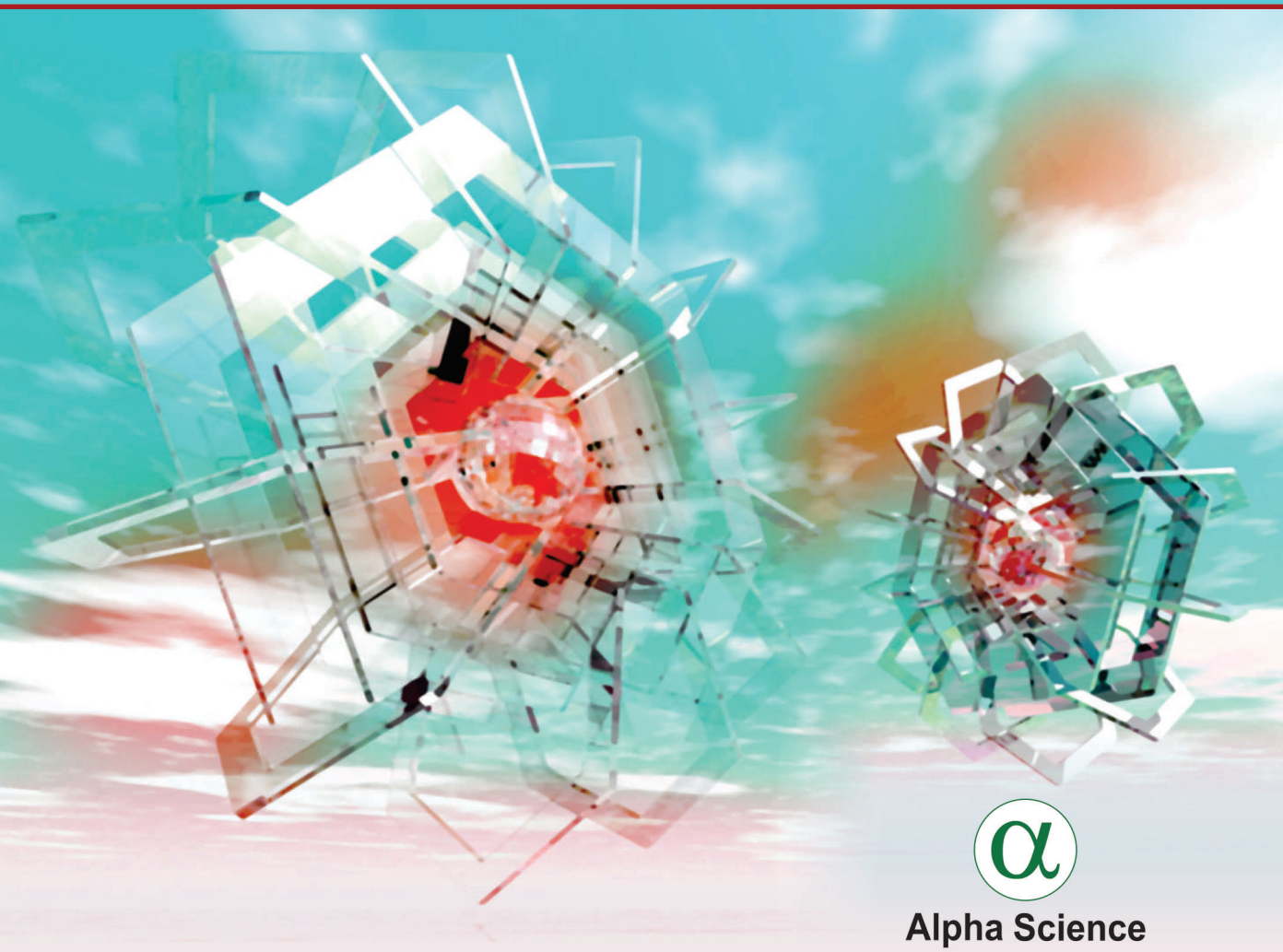
Jitendra Singh

Data Structure Simplified

Implementation using C++



Alpha Science

Data Structure Simplified
Implementation Using C++

Data Structure Simplified
Implementation Using C++

Jitendra Singh

Alpha Science International Ltd.
Oxford, U.K.

Data Structure Simplified
Implementation Using C++
368 pages

Jitendra Singh
Department of Computer Science
PGDAV College
University of Delhi

ALPHA SCIENCE INTERNATIONAL LTD.

7200 The Quorum, Oxford Business Park North
Garsington Road, Oxford OX4 2JZ, U.K.

www.alphasci.com

ISBN 978-1-78332-370-8

E-ISBN 978-1-78332-428-6

Printed from the camera-ready copy provided by the Author.

Dedicated to
my beloved sister Anita, my respected
father Sh. Jabar Singh Yadav
&
my mother Smt. R.B. Yadav

Preface

This book has been written to develop the necessary skill needed to understand the data structure concept and programming. Important topics of data structure are extensively covered. We have started our book from Abstract data type and algorithm. In abstract data type, we have discussed about the significance of Abstract data type in data structure. The second focus of this chapter is to highlight the significance of algorithm in designing and solving the computer problems. We believe that understanding of this topic before the implementation of other topics will enable the readers in effective reading as well as writing of code. Mere writing the algorithm is not enough, instead writing an efficient algorithm is the need of an hour. This can be accomplished with the help of an efficient and effective algorithm writing and same has been highlighted in the remaining chapters.

Arrays and pointers are one of the most important topics in data structure. Familiarization with these topics at early stage of learning will facilitates readers in comprehensive learning of complex code that is written using pointers and array. This chapter covers the usage of array and pointers. To ensure that the students have grasped the concept completely, enough exercises along with their answers have been included within the chapter as well as at the end of the chapter. Strong understanding of arrays and pointers will be extremely helpful in understanding the advance topics, including the link list, stack, trees among others.

In areas where access of one item after another is needed, link list is most appropriate for such type of requirements. Usage of link list allows the linear traversal. In link list, user can traverse with the help of next pointer that is stored in each node of a link list. Various aspects of link list including insertion of the node at any position and deletion of any node is discussed in detail. Majority of the topics related to the link list are extensively covered. This topic is discussed before the stack to highlight the concept of dynamic memory allocation and organizing the element in a dynamic data structure. This will also assist readers in the creation of dynamic stack that is widely used in subsequent chapters of the book.

The next two chapters are related to the arrangement and accessing of the item. We have started it from the data structure that is opened from one end. This data structure is termed as stack. The second data structure is also related to the arrangement of item(s). However, it is equipped with two ends and knows as queue. One end is utilized for insertion whereas the other one is utilized for the removal of items. We will be widely utilizing this data structure in the other chapters of this book wherever we need to store the element and access them in the order of insertion.

Linear data structure is followed by non-linear data structure and started with definition of trees and the need of the tree. Heading further, we have discussed the special case of a tree known as binary search tree (BST). Major concepts pertaining to BST including insertion of a node, traversal in a BST, deletion of the node have been extensively covered. In addition, we have also covered the programming of tree in detail. All the programs written in this chapter and the subsequent chapters are adequately tested (Turboc compiler) before including them in this text book.

Further extending the topic of tree, we have considered the advance tree in which the major limitations

of BST have been discussed. To overcome the limitation of BST, other advance tree has been discussed. Consequently, threaded tree and AVL tree are discussed in detail. BST, AVL tree, threaded tree etc. are of order 2, as a result, attaching more than two nodes is not possible. To accommodate more than two nodes multiway tree have been discussed. Range of topics pertaining to multiway, including B+ tree, B* trees have been described in detail.

Text book on data structure would not be complete without discussing the core concept of string and sorting. In order to abreast the readers with string and sorting, these topics are widely described in the chapter on sorting and strings. Readers can get equipped themselves with various sorting and searching techniques applied for the manipulation of string.

Eventually, we have discussed the graphs that are needed in the wide variety of applications, for instance, during routing in a network. The key focus of this chapter is to cover the basics of graphs, representations techniques of graphs, and various methods that are applied for the traversal in a graph etc. Finally, shortest path is determined from the source to the destination by applying the prominent techniques existing.

Acknowledgement

It was indeed motivation and encouragement from many of my students who have consistently inspired me for writing this book. Infact, students were aiming for the text book that should enrich them both theoretically as well as practically. At the same time, the book should be simple to read. Over and above, it should enable them to learn the domain where the data structures can be applied.

This journey would have not been completed without the support of many of my colleagues. It will not be possible to name all of them, indeed, it is worth naming few of them. First and foremost, I am extremely thankful to Dr. Vikas Kumar, who has taught me the art of presenting the text. He has also encouraged me to write few books after submission of my thesis. I am also grateful to my colleagues in PGDAV College (University of Delhi, India), Stratford University, USA (India Campus), who have always discussed me about data structure and the need of a book that should have wide coverage on the case study aspect of data structure.

I am extremely thankful to my wife Ms. Nitu and son Varunjit Singh for extending their cooperation and sparing me for long hours that were needed for writing this book. Finally, I devote this book to my parents who are the constant source of my motivation and inspiration. They have indeed encouraged me to contribute in computer domain in order to benefit the entire computer fraternity, and people below the bottom line in particular.

Jitendra Singh

Contents

Chapter 1

Basic Concepts

Chapter Objective

- Describe the abstract data type
- Usage of abstract data type
- Define the algorithm
- Characteristics of algorithm
- Types of algorithm
- Measuring the complexity of an algorithm.

1.1 INTRODUCTION

Data structure is widely used in several domains of computer science. With the advancement in computer electronics, processing speed and memory are no longer constraints for the computer system. Still, for massive data processing, efficient program plays a significant role in generating instant results. Therefore, it is significant that the algorithm employed should be efficient at the same time should require minimum time and space for execution.

It is equally important that the program serving one objective should solve other similar objective without undergoing major modifications. During the program implementation, underlying complexity should remain hidden from the user. To deal with all the aforementioned objectives, this chapter highlights the various aspects of algorithm that have significant impact on the performance of a program.

1.2 ABSTRACT DATA TYPE (ADT)

In a language, data types plays significant role in the representation of data. ADT can be defined as a mathematical representation that governs the various factors including:

- Type of information accepted.
- Range of permitted values.
- Memory space needed.
- Operations permitted on the data type.

Primitive data type includes integer, real, character etc. on such data types, only basic operations such as addition, subtraction, multiplication etc. are allowed. Consequently, complex operations cannot be applied on primitive data types. This limitation has been

overcome with the help of ADT as it needs to define the complex operations.

In abstract data type (ADT), users are provided with the limited details which are really needed to them; remaining information remains hidden and inaccessible to the users. Hiding implementation details from the user serve the wide purposes. For instance, modification in the code may be needed due to bugs or any other reasons. With the usage of ADT, only hidden details are changed without affecting the user level accessibility. Hence, user is not affected due to the modification or any inclusion of additional functionality.

Object oriented language has the capability to support the abstract data type. In object oriented language, complex as well as additional operations are required to be defined. ADT facilitates in defining additional operations that were not possible on primitive data types. Even the hiding of internal details is widely supported in the object oriented language. In object oriented approach, to hide the members, they are declared as private. In complex program, usage of friend function is not allowed since it allows the accessibility of private member. ADT offers flexibility to implement the definition for complex structure, although it might have implemented for primitive data type.

1.2.1 Need of an ADT

User needs to implement the abstract data types due to the variety of reasons. This is facilitated with the help of interface that is offered by the implementer. Abstract data type needs to implement various details. As a pre-requisite, code offered should be reliable and robust. Reliability can be attended with the help of data hiding. Consequently, user doesn't enjoy the flexibility to access the defined variable.

1.2.2 Implementation of an ADT

Even though, ADT is widely used due to the existence of its various promising features. However, before implementation, readers need to learn many of the significant terms and their meaning associated. Same has been defined herein:

Constructor/Modificators - In a language, constructor is a special method this is capable to build or change the state of an object. Name of the constructor cannot be assigned arbitrarily instead; its name is governed by the class name for which constructor is to be created. Constructor name is same as the class name. Constructors are invoked implicitly by their objects when they are created.

Accessors - Methods that can be accessed with the help of an object are known are accessors. Values to the variable can be assigned with the help of accessors. Consider a case of a queue implementation; an accessor can be developed to carry out the various operations needed. For instance, insertion of items in a queue is termed as en-queue whereas removal of item is known as de-queue (Readers who are not aware of the queue, please refer section 6.1, basic definitions, for clarity of the topic). Corresponding, enque and deque accessor can be implemented. Due to the usage of ADT, user does not need to know the implementation details. Instead, can concentrate on the usage of various functions

offered by enque and deque accessors. In addition to the aforementioned operations, one more operation can be applied that does not moves the item but let us know to determine the item(s) that is available at the front of the queue without removing it.

During the implementation of ADT, we need to learn the various input, output and changes that can be caused due to the insertion of the items. Indeed, we need to determine the final output after the changes are incorporated. Same has been enumerated in table 1.1 by considering the case of a queue.

Table 1.1: Queue interface

Name	Description	Input	Output	Changes
Enqueue	Places an item at the end of a queue, if queue is not full	Queue and an item	Integer value	Upon addition of an Item, rear will move one position backwards
Dequeue	Removes an item from the front of the queue, if queue is not empty	Queue	Integer value	Front item is removed and the front will point to the next element
Empty	Examines whether the queue is empty or not	Queue	A Boolean value	
Item	Returns the first element without making any changes at front	Queue	Front of the queue	

Axioms - Axioms are the mathematical representation of the various operations that can be carried out by utilizing ADT. Consider a case, if the queue is q1 and the item to be inserted is 'e' then the axioms are:

- Enque(q1,e)=e //Element inserted
- Dequeue (Enque(q1,e))=e //Element removed
- Empty (Enque(q1,e))=Boolean //status flag
- Item(Enque(q1,e))=e //first element of queue

Once the axioms have been written, they lay the fertile ground to denote the prototype of the functions that can be used to accomplish a specific task. Prototype of the above axioms can be the following:

```
Template<class T>
T enqueue (Q q, T t1); //insert the items in the queue.
T dequeue (Q q); //returns the item removed.
bool Empty (Q);   // To check whether the queue is empty
T Q (T item);     // to determine the first element of the queue
```

Above functions signifies that the usage of an ADT makes the programmer life easy. Any error or complexity can be analyzed well before time. Correspondingly, it saves a lot of time during the testing stage.

1.3 ALGORITHM

In computers, we need to solve wide variety of existing problems, majority of them fall under the category of simple or complex. Any problem can be solved by applying sequence of statements. Algorithms are comprising the set of statements needed to solve a given problem.

Algorithm - In majority of the cases, algorithms are written using pseudo code. Coder or developer will translates pseudo code used in the algorithm into a specific code of the target programming language. Good algorithm is a pre-requisite for good functions. To overcome a problem, sequential set of statements are grouped together, and may be categorized into simple statements, conditional statements, loops statements etc. It requires careful thought for writing an efficient and cost effective code by utilizing these language constructs. For instance, the written logic should not allow a loop to continue, once search is already accomplished. If the element is already found then the looping statement must terminate. A faulty algorithm to search an element has been discussed as follows:

```
Write an algorithm to search an item in a given array
int search (a [], element)
{
found=false;
k=0;
while (not end of a)
{
if (a[k] == element)
{
found=true;
} // end of if
increment k;
}// end of while
}// end of search
```

In the above algorithm, even though the element has been found at its first location, despite that searching will continue till the end an array is not reached. However, it is undesired; since loop allowed to execute without any objective. Consequently, we have wated the precious processor cycle without any aim. In order to gain high degree of efficiency, the above code needs to be modified in a manner so that any possible unnecessary cycle of loop can be eleminated. Same has been described in the following program.

```
Write an algorithm to search an item from the given array
search (a [], element)
{
found=false;
k=0;
while (not end of array a [])
      {
      if (a[k] == element)
      {
      found=true;
      break;      // come out of the loop
      }
      k++;  //increment the value of k
      } // end of while
}// end of search
```

Writing an efficient algorithm has major role in polarizing the program based on latency, throughput, space needed etc. Correspondingly, an algorithm needs to have some specific properties that include:

Input - An algorithm should be capable enough to accept the input. Based on the industry type, input can be supplied within the program or from an external world. Input supplied guides an algorithm for further processing.

Output - Algorithm should be capable to produce the verifiable output after processing of the given input.

Unambiguous - Language used to write an algorithm should have an explicit meaning. Ambiguous words leads to more than one interpretation. Consequently, it leads to errors in the resultant code.

Termination - All the algorithms are written to accomplish a specific task. Algorithm should be capable to accomplish that task in finite steps. Once the objective is achieved, an algorithm should terminate.

Feasibility - Algorithm written to solve the specific problem should be feasible for the language considered in order to get it implemented in real world with the assistance of contemporary languages.

1.3.1 Call based Algorithms

To solve any specific problem, we can apply range of approaches. Problem solving approaches have their own benefits and limitations. Broadly, algorithm based on call/flow can

be categorized into the following types:

- Non-recursive
- Recursive

1.3.2 Non-Recursive

In the non-recursive algorithm, we use conditional statement(s) or the looping statement(s) in order to solve the specific problem. This approach is considered as simple and widely utilized by the developers in their coding. Debugging the recursive algorithm is simple and needs less effort.

```
To find out the sum of n numbers
int sum_array (int x [])
{
int total=0;      //initialize the total to zero
while(x[i])
      {
      total=total+x[i]; //add the element with total
      i++;  //go to the next element
      }
return total;
}
```

Non-recursive algorithm has less overhead relative to recursive approach that utilizes the stack to store/save it various states. However, there are number of instances where solving the problem using the non-recursive approach is complex, for instance, traversal in a tree, Tower of Hanoi, factorial etc. In such situations, applying the recursive approach makes the programmer's life simple.

1.3.3 Recursive Approach

A problem that is solved using non-recursive approach can also be solved with the help of recursive approach. In the recursive algorithm, function is defined in its own term. However, function comprise of a constrain in order to prevent calling to itself infinitely. For instance, consider a case to add all the elements of an array.

```
int sum (int x [], int n)// Array element sum, recursively
{
if (n==0)   // base condition
return 0;
return total=sum(x, n-1) +x[n];     //decreasing n
}
```

In the above example, the entire problem has been solved in terms of sum function. Whenever, we are writing a recursive approach, it is imperative that two conditions should be essentially included and same have been mentioned herein:

- Base condition (Condition at which the function terminates).
- Subsequent call that leads to the base condition

In the above program, 'if n==0' is the base condition, whereas sum(x, n-1) is the call that leads towards the base condition. To accomplish this objective, decrementation in the value of 'n' is carried out by using the expression 'n-1'.

1.4 COMPLEXITY ANALYSIS

A number of methods can be applied to solve a problem. Programmers solve the problem(s) as per their perceptions and understandings. Due to the existence of variety of solutions, it becomes extremely challenging to select the solution that yields the best performance.

Selection of an optimum solution is governed by various factors, for instance, computation cycle needed to solve a problem, space requirement in memory, storage requirement etc. Solution that needs minimum time to compute, minimum space in memory is considered as the optimum solution.

Algorithms which are selected based on time and space requirement are known as asymptotic. In such type of algorithms, we compute the complexity in account of best or worst cases. Analysis is one of the major components of any algorithm written. In best cases, an algorithm can solve the given problem in few iteration or in best case even in one iteration. However, in worst case it can solve the considered problem in last step or penultimate step.

Other method to compute the efficiency is termed as amortized method. In the amortized method, we do not compute the efficiency of an algorithm in best or worst cases instead in terms of average time. However, in asymptotic, complexity efficiency is determined in terms of:

- Maximum time needed.
- Minimum time needed.
- Equal time needed.

Accordingly, they are categorized into lower bound, upper bound, equal bound etc. In this book, we have considered big (O) that represent the upper bound.

1.4.1 Big-Oh

Big Oh for a function with 'n' numbers can be defined as f (n) <=c*g (n) for all n, n>=n0;Big (Oh) acts as an upper bound that means it represent the complexity in the maximum terms. However, algorithm in real time may need lesser time than the one denoted by the Big (O).

Mastering big 'O' complexity is the major challenge to the students who have left the mathematics or could not remember many of the formulae. In big Oh, major complexities

results are O (1), O (n), O (n2), O (2n), O (log n). This section enumerates the Big Oh in profoundly simple terms with enough examples in order to offer profound simplicity.

A. O(1)

Big O (1) is the simplest among the other complexities in the Big Oh notation. O (1) denotes that only one comparison is needed. On various occasion, it has been observed that the program itself is of single statement. Consequently, complexity results in O (1). In the O (1) notation, computational time always remains same. Since, the item has been searched in the first step itself. For instance, consider the following example:

```
Problem: To find out the greater between two numbers
// Solution to find out the greater between two numbers
bool max_value (int a, int b) // function that accept two numbers
{
if (a> b)          // Compare the two numbers
return true;       // if first is greater return true
else
return false.      // otherwise return false
}
```

In the above example, only one comparison is needed therefore, overall performance will remain the same, irrespective of any other factors. So the complexity of this algorithm is O (1). Correspondingly, complexity is not dependent on the number of elements.

B. O(n)

O (n) is also known as linear complexity. In O (n), we start from the first element, afterwards second element and continue till the last element is reached. Upon reaching the last element, it may happen that item to be searched is not found. However, it may also happen that the item to be searched has been found before the n^{th} element, but we consider it as n^{th} only. Consequently, the complexity is represented as O (n). Same has been depicted in the following example.

```
// Program to search a number from a list of numbers
// number is to be searched, n is size //of number
bool search (int arr [], int number, int n)
{
bool found=false;
for (int i=0; i<n; i++)
      {
      if (arr[i] ==number)
      {
```

```
      found=true;
      break;
      }     // end of if
      }     // end of for
return found;
}     // end of function
```

In the above example, element to be searched may be in the first position, in that case the complexity would be O (1). However in the majority of the cases it will be in the second position, third position or may be upto the n^{th} position. However, we consider its complexity as O (n).

C. $O(n^2)$

Complexity of the type O (n^2) is known as quadratic complexity. O (n^2) results due to the nested loop involved in the algorithm. Consequently, for the 1^{st} iteration of the external loop, the inner loop execute for 'n' times. Same has been depicted in the following example.

```
Problem: Write a program to sort the series of numbers using Bubble sort
Solution: Bubble sort method to sort the given array
void array (int arr [], int n)
{
int i, j;
for (i=0; i<n; i++)     // start of outer loop
{
      for (j=1; j<n-i; j++)   // inner loop
      {
      if (arr [j+1] > arr[j]) // comparing the elements
      {                 // swapping if the adjacent is larger
      temp=arr [j+1];
      arr [j+1] =arr[j];
      arr[j] =temp;
      }     // end of if
      }     //end of inner for loop
}     // end of outer for loop
```

Above example signifies that for the one iteration of outer loop, the inner loop executes 'n-i' times. Therefore, total number of iteration can be computed as:

n [n-1+n-2+n-3+…….3+2+1].

=n (n+1)/2

=n^2 [we have considered that the value of 'n' is high (Here high signifies the value of 'n' is at-least of five digits i.e. 10000), in such cases value of 'n' will be negligible relative to n^2].

It is apparent from the above example that the power of 'n' will increase upon increasing the nesting. Consider a case that there are three nested loops, correspondingly the complexity of this algorithm will be O (n^3). If four nested loops are used then it will be O (n^4).

D. O(2^n)

O (2^n) types of complexity is termed as polynomial and will result if the consecutive terms are double to that of the previous one. Consider the example given herein:

No. of steps needed	2	4	8	16	32	64	128	256	512

In the above example, we can observe that the consecutive terms are double of the previous terms. Therefore, the complexity of such series will also result in $2(2^{n.-1} -1)$ i.e. O (2^n) considering the value of 'n' as sufficiently large.

E. O(log n)

Logarithm type of complexity results due to the split of the given series into two halves. Subsequently, from the two halves we select only one half. The half that is selected further divided into two halves i.e. left and right halves. This process continues till the objective to divide the series is not accomplished. This approach is also termed as divide and conquers method.

Consider the binary search, where we follow the aforementioned approach. To solve the binary search, firstly we would compute the mid by adding the index of lower and the upper (last) element in an array. Subsequently, result obtained is divided by 2. Here the number to be searched is compared with the mid element already computed. During comparison, we can met the following possibilities.

- Item to be searched is at the mid position of the list.
- Mid element is less than the number to be searched.
- Mid element is greater than the number to be searched.

If the number to be searched is greater than the mid element, in that case we follow the right half else we follow the left half of the series. This process continues till the number is not found or the series is not exhausted. Algorithm for the binary search is discussed as follows:

```
Program: Write an algorithm to search the element from the series by
using the binary search techniques.
bool bsearch (b [], n, low, upper)  //  'n'  is  the  number  to  be
searched)
{
bool found=false;
```

```
while (low<=upper)
{
mid= (low+upper)/2;            // find out the center
if (b [mid] ==n)
{
found=true;
break;
}
if (n < b [mid])
   {
   upper=mid-1;   // Select the left half
  }
      else
low=mid+1;         //Select the right half
 }    end of while loops
return found;
}
```

Upon application of the aforecited algorithm, the number of items that would remain in the series will be the half of the previous step. Therefore, total number of items remaining can be computed as:

n + n/2+ n/4……..4+2+1.

When we apply the definition of logarithm, and subsequently simplify it, the algorithm will result in O log (n)

EXERCISES

A. Descriptive Answers

1. What is the significance of data type in a programming language?
2. ADT is complement to the data type and not its substitution, justify.
3. How ADT is related to object oriented language?
4. What is an algorithm? How it is different with the program?
5. Efficient algorithm is the prerequisite for efficient program, justify?
6. Write an algorithm for binary sort?

7. What is asymptotic notation? How it computes the efficiency of a program?
8. How to define big 'O'?
9. How to define the upper and lower bound in complexity analysis?
10. Consider a case that in complexity computation, it has been computed in logarithm, what will be the effect on the value calculated, if the base of log is changed?
11. How complexity may differ in best case, average and worst case? Justify with help of suitable examples?
12. What will be the complexity of the following algorithms:
    - Insertion sort.
    - Linear search.
    - Binary search.
13. In the above question (12), how the efficiency will vary for each method? Which one among them is the best? Prove mathematically.

B. Multiple Choice Questions

(Mark the most appropriate answer. In case(s), if more than one options are true, select the most appropriate one.)

B.1 Abstract data type, allows defining:

i) New data type

ii) Extend the feature of an existing type.

iii) Both i and ii

iv) None of these

B.2 Abstract data type is widely utilized in modern programming:

i) True for all the language

ii) False for all the language.

iii) Both i) and ii)

iv) None of these.

B.3 In asymptotic notation, we compute

i) Time complexity.

ii) Space complexity.

iii) Running complexity.

iv) None of these

B.4 Big ‘O’ is the

i) Upper bound of an algorithm.

ii) Lower bound of an algorithm.

iii) both i) and ii)

iv) None of these.

B.5 Big ‘O’ of quicksort is

i) log n.

ii) n^2.

iii) n log n

iv) None of these.

B.6 In Big ‘O’, if the base is of log is changed

i) It will change the resultant value.

ii) It will not change the resultant value.

iii) It will change the value, however that change is negligible, therefore can be neglected.

iv) None of these.

Chapter 2

Arrays and Pointers

Chapter Objective

- Define array
- To describe the types of array
- Usage of single dimensional and two dimensional arrays
- Defining pointers
- Describing the single pointers and its usage
- Describing the pointer to pointer and its usage.

2.1 INTRODUCTION

Arrays are widely used in any programming language. It is extremely useful in cases where we need to store the similar set of elements. It helps in reducing the program complexity at the same time increases the programmer's productivity. In C++, it is used in implementing the string with the help of array of characters. Depending upon the element organization, arrays can be categorized into the following:

- Single Dimensional array.
- Double Dimensional array.
- Multidimensional array.

For this text book, multidimensional array is out of scope and here it has been discussed for the sake of completeness only. Remainder of the topics has been described in the upcoming sub-sections.

2.2 SINGLE DIMENSIONAL ARRAY

Consider a case that we need to store the marks of five students; in order to store the marks of these five students, we need to declare five different variables. Due to the involvement of homogeneous information that needs to be stored, variables declared should be of the same type. Program to store five elements has been depicted below:

```
Program: To accept marks of five subjects and display them (without
an array)
Solution:
int main ()
```

```
{
int marks1, marks2, marks3, marks4, marks5;
cout<<"enter marks1";
cin>>marks1;
cout<<"enter marks2";
cin>>marks2;
cout<<"enter marks3";
cin>>marks3;
cout<<"enter marks4";
cin>>marks4;
cout<<"enter marks5";
cin>>marks5;
return 0;
}
```

In the above program, we have observed that to store the marks of 05 different subjects, we have to create five distinct variables. Complexity of the above program will grow further upon increment of subjects. Correspondingly, to input the details of 100 students or more, there will be phenomenal growth in the complexity of the above program. To overcome such limitation, we need to evolve the aformentioned method in order to reduce the complexity at the same time to improve the programmer's productivity. Here the solutions lie with the usage of arrays. Once array is declared, space needed for an array is allocated contiguously. This space is allocated at the time of compilation.

Array can be defined as:

- Used to store, set of similar data types.
- In an array, elements are stored in continuous memory locations.
- Index, or subscript starts with 0.
- Size of the array should be constant.
- Character array terminates with the special symbols '\0', known as null.

Index is also known as subscript and starts from '0^{th} number. At index '0', first element is stored. Same has been illustrated in figure 2.2.

2.2.1 Declaration of an array

If we are intending to create an array that should be capable to store 10 elements. Consider that number to be stored is of integer type. To declare this array, following syntax can be used.

int number [10];

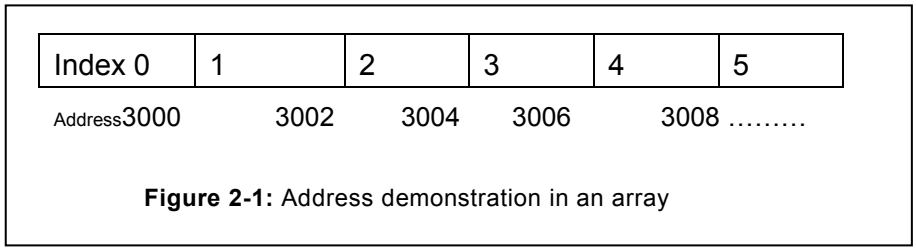


Figure 2-1: Address demonstration in an array

In the above statement, number is acting as an array identifier. Here an array of integer type is created, that is capable to store 10 integer type numbers. If we assume that the size of integer is 2; also assume that first address of an array is 3000 then we can compute the address of subsequent elements. Address computation of various elements of an array has been illustrated in the figure 2.1.

In an array, elements are placed one after another; however, storage at any index/subscript is possible. For instance, if the order of arrival of elements is 15, 10, and 7 then it is not mandatory that they have to be stored in the index 0, 1, 2 respectively. We can store them at any index of an array based on the availability. For instance, the first element can be stored in the array index 2, 10 at index 1 and the element 7 at the index number 0. Initialization of these elements is as follows:

array [2] =15;

array [0] =10;

array [1] =7.

However, as a good practice, it is suggested to use the contiguous address for storage. Correspondingly, to access an array element stored at 4^{th} index, syntax array [4] need to be used. It is worth mentioning that the element that is stored at 5^{th} location of an array will be accessed (Since index starts from 0). For instance, if the array is given as **int array1= {4, 9, 12, 67, 90, 45}**, in this case, array1 [4] will be 90. Since, the index 4 which is 5^{th} element of an array has value of 90.

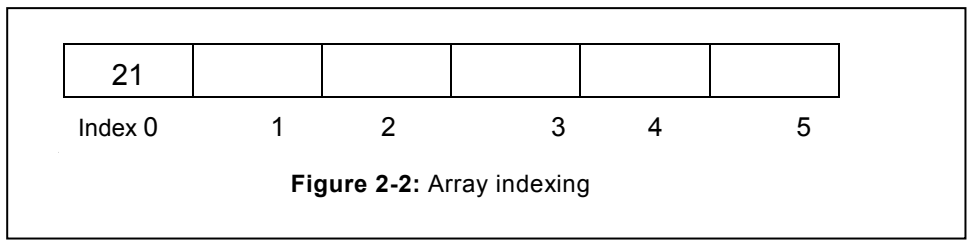


Figure 2-2: Array indexing

Array offers number of advantages such as it eliminates the need of creating many variables. Consequently, complexity of the program is reduced to a great extent. The other advantage is array is fast relative to the dynamically allocated variables. Following program demonstrates

the array usage.

```
Program: To accept and display five marks using array
int main ()
{
//Creation of an array to store five elements
int marks [5];
int i;
// input of five elements
cout<<”enter marks for five students”;
for (i=0; i<=4; i++)
cin>>marks[i];
cout<<”marks entered for five students are :”<<endl;
//Display of marks for five students
for (i=0; i<=4; i++)
cout<<marks[i] <<endl;
return 0;
}
```

Similarly, we can determine an element having the maximum value from the given set of elements. We would use an array to store the elements.

```
// Program: To input 5 random numbers and to find out the maximum element.
(Hint: Assume first element as maximum element)
int maximum (int a [])
{
int i=0;
int m=a [0];
while (a[i])      // examine all the elements of an array
{
if (a[i]>max)     // if array element is more than the max
max=a[i];   // Make this element as maximum
i++;  // increment the index
}
return max; //return the max
}
```

2.2.2 Initialization of an Array

Other than assigning the value using input technique, arrays can be initialized at the time of creation. Once array is initialized at the time of creation, there is no need to declare the size of an array.

int A [5] = {45, 6, 7, 8, 12}; (1)

int A [] = {45, 6, 7, 8, 12}; (2)

The above (1) is redundant, since the array is initialized at the time of creation. Therefore, it is suggested to use the syntax depicted in (2), since compiler automatically allocates the array size based on the number of elements initialized.

```
Program: Initializing of an array during creation
Solution:
void array1 ()
{
int arr [] = {12, 5, 67, 98, 54};
int i=0;
while(arr[i])
{
cout<<arr[i]<<endl;
i++;
} //end of while statement
} //end of function
```

In the above program, the array is initialized at the time of creation. Here the size of an array will be five, if user is attempting to initialize the size of an array say arr [3] then also it will be overruled by the compiler and the new size will be 05. This feature is extremely helpful in cases where we are aware the number of values to be enumerated.

```
Program: Write down the program that demonstrate the initializing of
an array at the time of creation.
void array1 ()
{
int arr [] = {12, 5, 67, 98, 54};
int i=0;
while(arr[i])
{
cout<<arr[i]<<endl;
i++;
```

```
} //end of while statement
} //end of function
```

A. Special cases of initialization

Beyond, the above methods of initialization, there is one more method of initialization that is worth discussing at this point in time. In this method, instead of initializing all the index individually, we are initializing only one and the remaining indexes are initialized by the compiler. Syntax for the program is shown in (3).

int arr [8] = {0}; (3)

In the above statement, all the array index will be initialized with 0 (i.e. arr[0] to arr[7])

But consider another example:

int arr [7] = {4}; (4)

In the above case (4), all the indexes are not initialized with data 4; instead seventh index will have the value 4, whereas others will have the value of 0 only. Method (3) is significantly helpful in cases where we need to initialize all the index of an array to 0 or any other value.

B. Problems based on integer array and their solutions

In this section, we will discuss some of the interesting problems related to the integer array and their solutions.

```
Program: Searching the element in an array.
void array1 (int arr [], int element) // element to be searched from
arr array
{
int i=0;
int found=0;            //initially, item is not found
while (arr[i])    // Run till item exist in the array
{
if (arr[i] ==element)   //Element found
 {
found=1;
break;            // terminate the further searching
}
i++;
}     //end of while statement
return found; // Now we need to return the status of found
} //end of function
```

Explanation: In this program, user is intending to search an element from an array. To

accomplish this objective, array is to be scanned from the first index till the last index is reached. During the search, there are two probabilities, i) element exist in the array. ii) It does not exist in an array, in the second case, found should return false value. In case element is found in an array, instead of continue to go upto the end, loop should terminate at the point where element is traced. Value of found to be changed to 1 that signifies success. Eventually, the value of false is returned to the caller.

```
Problem: Program to count the number of elements which are more than the given number
//Function to count the number of elements with values more than the given number 'n'.
int countNumbers(int array [], int number)
{
int count=0;
int i=0;
while(array[i])
  {
if (array[i]> number) // test if array element is more than number
 count++;   //increment the count
i++;  //increment the index of array
}
return count;     // return the count
}
```

Explanation: In this program, we intend to count the number of elements in the array which are more than the given number (n), in this case we have to scan the array from the first element and continued upto the last element. On each run, array element need to be compared with the given number, if it is more than the given number then the variable count is incremented each time. Eventually, variable holding the count is returned to the caller.

```
Program: Program to modify the array element to some specific multiple. For instance, if multiple is 5, then it should modify each element by multiplying them by 5.
void increase (int array1 [], int mul)
{
int i=0;
while(array1 [i])
{
//multiply the element with the given multiple
```

```
array1[i]=array1[i]*mul;
i++;
}
}
```

In this program, we intend to increase the value of array element by multiplying it with specific multiple. This can be achieved by accessing each element and multiplying them individually with the given multiple. Once the elements are accessed and multiplied by the given multiple, numbers need to be stored back at their respective locations.

```
Program:  To find out the average of given numbers in an array
int average(int num[])
{
int i=0,sum=0, avg = 0, count=0;
while(num[i])
{
sum=sum +num[i];
count++;
}
avg=sum/count;
return avg;
}
```

Consider a case in which we are not knowing the total number of elements available in the array. However, we have to determine the average of all the numbers stored in an array. To accomplish this task we have computed the sum of all numbers by using the variable 'Sum'. At the same time, number of elements in the array is also computed with the help of count variable. Eventually, total of all the numbers is divided by the count (that represent all the elements).

2.2.3 Character Array

In a single dimensional array, character elements can also be stored. Character elements are terminated by the special symbol '\0' also known as NULL. This special character has significant impact in determining the size of an array. **Character array is also known as String in C++**.

```
Program 2.6: To find out the length of a string.
int length(char name[])
{
int len=0;
```

```
while(name[len++]!='\0'); // From beginning to end
return len;              //return length.
}
```

In the above program, array is started from the first index and we increment the index till end of an array is reached. Once the end of an array is reached the length of the array is returned to the calling function.

These functions help enormously in other functions of string. Correspondingly, we can determine whether a given string is palindrome or not. A string is known as palindrome if reading from left or right returns the same result. For instance, madam, Malayalam, nitin, pop etc. is widely used string palindrome.

To determine whether the string is palindrome or not first element is compared with the last element, second element with the penultimate element, so on and so forth, till we don't reach to the mid element. If all these elements are same then the given string is known as palindrome otherwise it is not a palindrome. Function to determine the palindrome has been given below.

```
// Program to determine the string is palindrome, (// creating function palindrome and accepting one argument of string type)
int palindrome(char str[])
{
int l=length(str);      // finding out length of the string
int i=0, flag=1;
while(i<=l) // executing the loop upto center element
 {
if(str[i]!=str[l])
{
flag=0;           // setting the flag value to false and breaking the loop
break
}
i--;
l--;
}
return flag;      // returning the value of flag to the calling function
}
```

2.2.3.1 String Programs

In the above function, we have considered that the given string is palindrome, correspondingly flag has been set to true. In the while loop, we compare the first element with the last element, if both are same, we compare the second element with the penultimate element. If these two are also same then remainder elements are tested for the equality. This will continue till the string is not exhausted. At any point of time, if the characters that are matched are not same, then the flag is set to false and further test of the element is terminated. Eventually, flag is returned to the caller function where the status of flag is evaluated to flash the appropriate message (true or false).

In addition to the above discussed string program, there are many other interesting array program. Several of them have been discussed in detail.

A. Program to concatenate two strings

In our daily usage, we require that second string should be concatenated with the first. For instance, in a database that is storing the first name and the last name, here the need is to concatenate the second name with first and eventually, concatenated string is then displayed.

```
// Program to concatenate second string with first
// function to concatenate the string2 with string1
void concate(char s1[],char s2[])
 {
 int i=0,j=0;
 // Running upto the end of the first string
 while(s1[i]!='\0')
 i++;
// concatenate the second into first from the end
 while(s2[j])
 {
 s1[i]=s2[j];
 i++;
 j++;
 }
 s1[i]=s2[j];     // copying the null character into the first string
 }
int main()
{
clrscr();
```

```
char str[40];
cout<<"Enter string "<<endl;
cin>>str;
char str1[]="singh";
concate(str,str1);
cout<<"String after concatenation"<<str<<endl;
getch();
return 0;
}
```

B. Program to copy one string into another

In this program, we copy one string into another string; here we consider that first string is copied into the second string. To copy the string one (str1 into the string two (str2), we have to copy the characters of str1 into the string str2 till str1 is not exhausted. After completion of the copy, NULL character of str1 is also copied into the str2. Otherwise the resultant string will not have the NULL character. Consequently, it will pose the problems in further operations, such as display, evaluating string length etc. Same has been depicted in the following program.

```
//function to copy first string into second
void strcopy(char s1[],char s2[])
 {
 int i=0;
 while(s1[i]!='\0')
 {
 s2[i]=s1[i];
 i++;
 }
 s2[i]=s1[i]; //copy the null character
 }
```

2.3 TWO DIMENSIONAL ARRAY

In two dimensional array, elements are arranged in rows and columns. Indexing in two dimensional arrays starts with '0^{th}' index. Statement to create two dimensional arrays has been depicted below:

int A[row][column]; //declaration

int A[5][4]; // should have constant type parameters

In the above statements, a matrix with 5 rows and each row with 04 columns will be created. In computer, elements will be arranged in a single row and their indexing will start from 0. Same has been illustrated in the following figure.

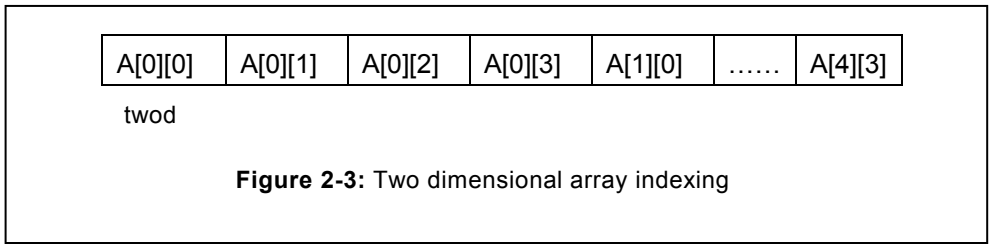


Figure 2-3: Two dimensional array indexing

Once the index of the columns reaches to the maximum value then only index for second row starts. In the above example, total numbers of columns that have been declared are 4 therefore, row 1 will start after the row '0' has exhausted its column's upper limit, i.e. 3. However, in the random storage, we can store the elements anywhere within the two dimensional array, without the completion of any column. It signifies that the statement like A[3][3]=90 is permitted irrespective of the state of elements in the previous indices.

2.3.1 Two dimensional integer array

Two dimensional arrays are also known as matrix. Similar to one dimensional array, two dimensional array can be initialized at the time of creation. Initialization of the matrix at the time of creation has been depicted as follow.

```
Int x[3][4]={
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {2, 4, 6, 3},
        };
```

The above matrix is having three rows and four columns. Correspondingly, we can also create the two dimensional array by:

- Declaring the two dimensional array
- Initializing the elements of the two dimensional array.

The following sub-section describes how to accept and display the matrix.

A. Program to accept the matrix

As described earlier, in the case of two dimensional array the first statement is used to declare that the array is two dimensional of specific data type. To accomplish this objective, we can use the syntax:

```
int mat[3][4];
```

Here the keywords ‘int’ used denotes that the two dimensional array is of integer type. Mat[3][4] signifies that it would comprise of three row and four column. Afterwards, we can use the nested loop in order to accept the elements of the matrix. The outer loop will denote the row whereas the inner loop will denote the column. The entire program has been depicted in subsequent page.

Program: Program to accept the matrix and to display it.

Solution:

```
void inmatrix(int mat[3][4]) //function inmatrix accept the matrix
 {
int i,j;
cout<<”enter the element of the matrix”;
for(i=0;i<=2;i++)       // start of outer loop
  {
for(j=0;j<4;j++)        //start of inner loop
  {
 cin>>mat[i][j];
  }
}
void display(int mat[3][4]) // function to display the elements input previously
  {
int i,j;
cout<<”element of the matrix are :”<<endl;
for(i=0;i<=2;i++)       // start of outer loop for the row
  {
for(j=0;j<4;j++)        //inner loop for columns
  {
 cout<<mat[i][j];
  }
//change the line after completion of the row
cout<<endl;
}     //completion of column loop
//void type therefore no need to return any value
}
```

B. Program that sums two Matrix

We have already learned during our mathematics course, the way matrix sum is carried out. As a elementary rule sum on two matrix can be performed by considering one element of row one column one, with the corresponding element of the second matrix. Result obtained after adding is stored in the third array. In our case, we have named the third array with the name 'result'. Same procedure will be carried out for the second and subsequent elements. Program will terminate when all the elements are exhausted.

```
Program: To sum two matrix
 void sum(int matrix1[2][3],int matrix2[2][3])
{
int result[3][4];
for(int i=0;i<3;i++)
  {
   for(j=0;j<=3;j++)
      { //storing the sum of matrix1 and matrix2 into result matrix
      result[i][j]=matrix1[i][j]+matrix2[i][j];
      }
}
}
```

C. Program that transpose the matrix

In a matrix, if we change the row of a matrix with the corresponding column, then it is known as transpose of a matrix. In the transpose of a matrix, we exchange the first row with first column, second row with second column, so on and so forth.

We can accomplish the transpose of the matrix by exchanging the row into column and column into row. To achieve this objective, we have created the function transpose that exchanges the row into column and stores the resultant element into a temporary matrix named tp. Once the matrix is transposed into the tp matrix, it is copied back to the original matrix. Same has been depicted in the following program.

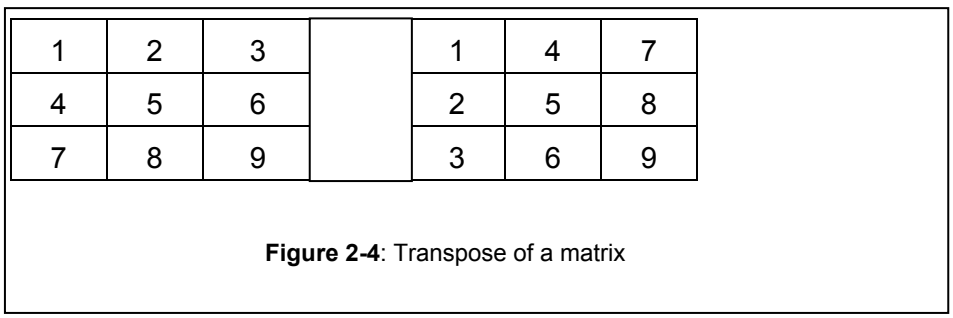


Figure 2-4: Transpose of a matrix

```
Program: To transpose the matrix; (elements of row and columns are
interchanged)
void transpose(int m[3][3]);
int main()
 {
 clrscr();
 int matrix[3][3]={{3,4,5},
            {6,7,8},
            {9,10,11}};
            int i, j;
            cout<<"Matrix output"<<endl;
      for(i=0;i<=2;i++)
      {
      for(j=0;j<=2;j++)
       {
      cout<<matrix[i][j]<<"\t";     // display of matrix element
      }
      cout<<endl;
      }
      transpose(matrix);      //transposing the matrix
      cout<<"Matrix after transpose"<<endl;
      for(i=0;i<=2;i++)
      {
      for(j=0;j<=2;j++)
       {
      cout<<matrix[i][j]<<"\t";
      }
      cout<<endl;
      }
      getch();
      return 0;
      }

//program to transpose the matrix
```

```
      void transpose(int matrix[3][3])
            {
            int tp[3][3];
            int i,j;
      for(i=0;i<=2;i++)
      {
      for(j=0;j<=2;j++)
      {
      tp[j][i]=matrix[i][j];  //  transposing  and  storing  in  tp
      matrix
      }
      }
      cout<<" Matrix after transpose"<<endl;
      for(i=0;i<=2;i++)
      {
      for(j=0;j<=2;j++)
      {
      matrix[i][j]=tp[i][j];  //  Reassigning  the  tp  back  to  the
      matrix
      }
          cout<<endl;
        }
}//end of the function
```

D. Program to multiply two matrices

We know that during the multiplication of two matrices, it is significant to determine the feasibility of the multiplication. Possibility of multiplication can be evaluated with the help of number of columns in the first row to that of number of rows in of the second matrix. If both of them are same, then only the multiplication can occur, otherwise multiplication wouldn't be possible. From the above rule of multiplication, we can determine the number of rows and columns in the resultant matrix.

```
Program : Program to multiply two matrices
void multiply(int a[r1][c1], int b[r2][c2])
   {
  int c[c1][r2];
int sum=0;
if(c1!=r2)
```

```
{
cout<<" multiplication not possible"<<endl;
getch();          // wait for the key to be pressed
return;           // return to the calling function
}
//if the matrix product is possible, in that case
for(int i=0;i<=c1;i++)
  {
   value=1;
for(j=0;j<=r2;j++)
   {
for(k=0;k<c2;k++)
sum=sum+a[i][k]*b[k][j];
}
c[i][j]=sum;
sum=0;
}
```

Diagonal elements of the matrix can easily be traced. In a matrix, there are two types of diagonal exist namely the forward and the backward diagonal. We have considered the forward diagonal for our discussion, same has been depicted in the following example.

```
Program to display the forward diagonal of the matrix
void fordiagonal(int matrix[][])
{
int row,col;
for(row=0;row<n;row++)
{
col=row;    // diagonal can be accessed if both row and
            //column index are same
cout<<matrix[row][col];
}
}}
```

Two dimensional character arrays have significant role in the storage of more than one names, since the name can only be stored in two dimensional arrays. Two dimensional character array can be created with the help of following syntax.

char name[number][length];

Here the first argument signifies the number of names to be entered, whereas the second argument[length] signifies the maximum length that a name can take.

Program that input 10 names has been discussed here.

```
Program: Write a program that can accept 10 names, also display the output
Solution
char name[10][40];
void main
{
int i=0, j=0;
cout<<"enter 10 names";
for(i=0;i<=9;i++)
cin>>name[i];
cout<<" here is the output of  names";
for(i=0;i<=9;i++)
cout<<name[i]<<endl;
}
```

It is significant that the reader should learn, that the elements of two dimensional character array can be easily accessed with the help of pointers.

2.4 POINTERS

Pointer is a variable that is capable to hold the address of another variable. Holding of addresses of another variable is needed in various instances that include:

- To access the array element
- To change the value of variable from function
- In dynamic allocation of memory.
- In complex programming, such as link list, tree, B tree etc.

To denote that a specific variable is acting as a pointer, they are preceded with the symbol '*'. For instance, int *x, Here the pointer is integer type. It means that this pointer can hold the address of integer type variable.

2.4.1 Declaring pointers

As already stated that pointer is a variable that is capable to hold the address of another variable, correspondingly, we need a variable whose address can be stored in the pointer and consequently, we can access the variable with the help of pointer. To declare the pointer, variable name is preceded with asterisk (*) symbol, same has been illustrated in the following program.

```
int x=8;
int *p;// variable that is pointer of int type
p=&x; //p now holds the address of variable x
cout<<p;     // print the address of x;
cout<<*p;    // print the value pointed by p;
```

Working of the above statements has been illustrated in the figure 2.5. Initially, the variable 'x' is declared and assumes that it has been allocated the address location 1000. Similarly, when int *p is declared, it is also allocated the address similar to any other variable. Here we have assumed that the pointer 'p' is allocated new address which is at memory location of 1010. Consequently, we can conclude that both are different variables and definitions applicable to variable is still applicable on pointer variables. In addition, operations applicable to pointer variable can also be applied on them.

2.4.2 Accessing the pointer variables

Pointer is also a variable but instead of holding the value it holds the address of another variable. We are assuming that it is holding the address of variable 'x' that is having the address 1000. Now, it will have the address of 1000 in its value part, as illustrated in the figure 2-5. Once user is displaying the value of this pointer, he will get this address as value.

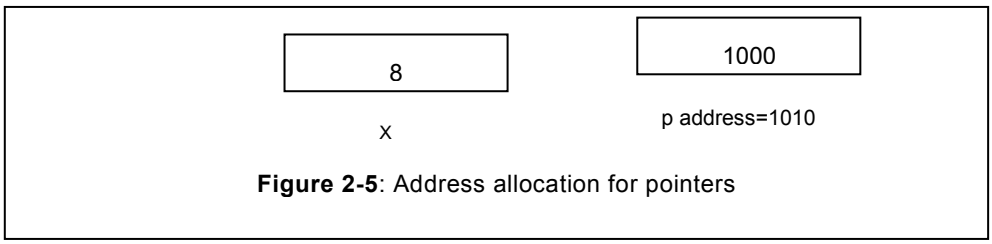


Figure 2-5: Address allocation for pointers

However, it facilitates to access the variable whose address is held by the pointer. To access the value stored in the address held by the pointer we use de-referencing operator. To denote the de-referencing, the statement *p is used. Consequently, it displays the value stored at that address. Usage of pointer has been depicted in the following program.

```
// Program to display the use of pointer
int x=8;
int *p;
p=&x;
cout<<" value of x "<<x // 8
cout<<" value of p="<<p //1000
cout<<" value pointed by *p"<<*p;   // 8 will be printed
```

```
*p=30;
cout<<" value of x"<<x; // 30
cout<<" value of p="<<p;            1000
cout<<" value of *p="<<*p;    // 30
```

Pointers are widely used in the character array (string). Similarly, pointers are also used in integer array. Pointer usage in array has been depicted in the subsequent sections.

2.4.3 Pointer to pointer

Sometimes, we need to store the address of a pointer, this can be accomplished with the help of pointer to pointer. Pointer to pointer is a variable that holds the address of another variable that is pointer type.

Declaring pointer to pointer

Declaring pointer to pointer is different from the normal pointer type. In pointer to pointer notation two asterisk (**) are preceded before the identifier.

int **pp; (2.4.1)

int *p; (2.4.2)

pp=&p; (2.4.3)

In the line 2.4.1, a variable declared is pointer to pointer type. It is capable of storing the address of another pointer variable. Notation to store the address of pointer in the 'pointer to pointer' has been depicted in the line 2.4.3. Usage of pointer to pointer type has been depicted with the help of following example.

```
// Program to display the use of double pointer
#include<iostream.h>
#include<conio.h>
int main()
{
int v=20;
int *p;
int **pp;
// Assigning address to pointer
p=&v;
pp=&p;
cout<<"value of v=:"<<v<<endl;
cout<<"Address of v=:"<<&v<<endl;
cout<<"Address held by p=:"<<p<<endl;
cout<<" Address of p=:"<<&p<<endl;
```

```
cout<<"Valued de-referenced of p=:"<<*p<<endl;
cout<<" Address of pp=:"<<&pp<<endl;
cout<<"Address hold by pp=:"<<pp<<endl;
cout<<"Valued de-referenced of *pp=:"<<*pp<<endl;
cout<<"Valued de-referenced of **pp=:"<<**pp<<endl;
getch();
return 0;
}
```

Output

```
Value of v=20;
Address of v=0xfc1fff4
Address of *p=0xfc1fff2 // Now address of p
Address held by p=0xfc1fff4   //Address hold by p (i.e. of v)
Valued de-referenced of p=20  //de-referencing the value of p (i.e. of v)
Address of pp=0xfc1fff0       // Now address of pp
Address held by pp=0xfc1fff4  //Address hold by pp (i.e. of p)
Valued de-referenced of pp=20 // de-referencing the value of pp(i.e. of v)
```

2.4.4 Use of Pointers in an Array

Pointers are widely used in arrays. To access the array element, we can initialized the index of an array to a pointer variable. For instance, in the array ‘a’, we have initialized it with list of numbers. Accessing of array element with the help of pointer has been depicted in the following program.

```
// Program to access the array element with the help of pointers
int main()
{
int a[]={10,50,20,40,70};
int *x;
x=&a[0];
cout<<x<<endl;
cout<<*x;          //output 10
cout<<*(x+2);            // output 20
cout<<a[2];        // output 20
cout<<*(x+3);            // output 40
```

```
cout<<a[3];        // output 40
return 0;
}
```

In the above program, pointer is initialized with the base address of an array. Consequently, statement *x displays the element stored at the index '0'. When the statement *(x+2) is used, it signifies that the 2 is to be added at the base address which results in x[2]. Consequently, the element accessed by the *(x+2) and x[2] gives the same result. In the same line, the statement *(x+3) will result in 0+3 i.e. 3rd index. Correspondingly, the element stored at the 3rd index will be accessed.

2.4.5 Dynamic Array creation

During the array creation, array size can be allocated either at runtime or at compile time. Accordingly, it can be classified as:

- Static array
- Dynamic array

Array whose size is allocated at compile time, such arrays are known as static array. Arrays that have been discussed in the previous sections fall under the category of static array.

Other type of array is dynamic array, space to dynamic array is allocated at run time. In C++, creation of dynamic array is accomplished with the help of new operator. Same has been depicted with help of following example.

Program: Create the dynamic array and discuss their various ways of initialization.

Solution:

```
#include<iostream.h>
#include<conio.h>
int main()
 {
int *x;
int *y;
y=new int(45);
x=new int[4];
clrscr();
// elements initialization will follow
x[0]=90;
x[1]=55;
x[2]=32;
```

```
x[3]=67;
// From this statement onwards dynamic array will be created.
int i=0;
cout<<"Dynamic array elements are"<<endl;
while(i<=3)
{
cout<<"x[i] i="<<i<<"  "<<x[i]<<endl;
i++;
}
cout<<"y ="<<*y<<endl;
getch();
return 0;
}
```

```
// Program to initialize the dynamic pointer.
#include<conio.h>
int main()
 {
int *y;
clrscr();
y=new int(45);
int i=0;
cout<<"y ="<<*y<<endl; //output will be 45
getch();
return 0;
}
```

2.4.6 Array of pointers

Array of pointers is an array that holds the pointers (addresses). Usage of array pointer is significant in cases where we need to store the addresses of various pointers of similar type. Array of pointer facilitate the ease of accessibility at the same time improves the efficiency of the program written. Primarily array of pointer is used to deal with double dimensional array type of problems. For instance, if we intend to store two or more arrays, then we can use pointer arrays. Example to access three array using array of pointers has been depicted in the following program.

```
// Program: Create a program to access integer arrays using array of pointer.
Solution:
#include<iostream.h>
#include<conio.h>
int main()
{
int a[]={26,89,4};
int b[]={98,5,42};
int c[]={78,6,23};
int *arrp[4];
arrp[0]=a;  // Storing the base address of array a
arrp[1]=b;  // Storing the base address of array b
arrp[2]=c;  // Storing the base address of array c
cout<<"Output of the array is"<<endl;
int i=0;
for(i=0;i<=2;i++)
  {
  for(int j=0;j<=2;j++)
    {
    cout<<arrp[i][j]<<" ";
    }
    cout<<endl;
    }
return 0;
}
```

In the above program, we have used array of pointers to store the address of three arrays. Since, naming the array denotes the first base address; therefore, these array pointers hold the address of the first address of assigned arrays. Using the array of pointers, we can access the individual element of an array as depicted in the above program.

The other potential use of pointer of arrays is in character array (also known as string), here the details of ‘n’ entities can be stored, and these details can easily be accessed using array of pointers.

In the following program, we have demonstrated how the various arrays holding the values can easily be linked with array of pointers and their values can be accessed easily.

```
Program : Accessing of character array(string) using array of pointers
#include<iostream.h>
#include<conio.h>
int main()
{
char ca[]="first";
char cb[]="second";
char cc[]="third";
char *cp[3];
cp[0]=ca;
cp[1]=cb;
cp[2]=cc;
cout<<"Output of the array is"<<endl;
int i=0;
    cout<<" Output of character array"<<endl;
    for(i=0;i<=2;i++)
    cout<<”output of array ”<<i+1<<cp[i]<<endl;
getch();
return 0;
}
```

Output

```
Output of array 1 first
Output of array 2 second
Output of  array 3      third
```

In the character pointer, statement cp[0]=ca holds the base address of character array ca, correspondingly, cp[1]=cb holds the base address of character array cb, and cp[2]=cc holds the address of character array cc. In the for loop, value at first address of the character array is shown, it continues to move further till it does not encounter the ‘\0’ character. As a result, outcome of first character is displayed. Similarly, output of other array elements will be displayed one by one in subsequent iteration.

EXERCISE

Descriptive Type Questions

1. How array is different from other variables.
2. What are the advantages of array?
3. An array is consisting of 10 elements. Delete the 4^{th} element; correspondingly move the remaining elements, so that now array should consist of 9 elements.
4. In the above array, insert the element at 3^{rd} position and move the third to n^{th} element to the right hand side, so that there is no overlapping.
5. Write a program (WAP) to find out the minimum element from the given array.
6. Write a program to increase the value of each array element by 2. For instance, if the array element were 2, 3, 9, and 4 then after adding 2 they should become 4, 5, 11, and 6.
7. WAP to find out the maximum and minimum element hold by the array. Use only one function, these minimum and maximum variable should be capable to be accessed from any other function.
8. Write a program to swap the first element with the second, third with forth, till the end of an array.
9. Write a program to swap first element with the last element, second element with the penultimate element and so forth. After making changes see the changes by creating a new function name it as display.
10. Write a program to merge the element of two arrays into the third array.
11. Write a program to store the string into the array without initializing it or accepting from the keyboard.
12. Write a program that eliminates the immediate repetition. For instance, if the string is 'jiitiiee' then the resultant string should be 'jitie'.
13. Write a function to copy the string into another array.
14. Write a program to capitalize the first element of an array.
15. Write a program that accepts numbers, character and special characters. Your program should count the number of characters, numbers and special characters.
16. Write a program that accept the string in small case and converts all the character into upper case.
17. Character array is of size 20; write a program that extracts the character from 4^{th} position to 9^{th} position.

Predict Output

1.
```
int x[]={3,8,5,90,12,34};
int y=2;
cout<<x<<endl;
```

cout<<*x<<endl;
cout<<*(x+3)<<endl;
cout<<*(x+y)<<endl;

2. `int *x=new int[4];
int *y=new int(4);

3. char x[]= “jitendra”;
cout<<x;
cout<<*x;
cout<<*(x+1);
cout<<(x+1);
cout<<*(x+3);
cout<<(x+3;
cout<<x+3;
cout<<*x+3;
cout<<(*x)+3;

Find out error(s), if any, in the following codes.

Problem 1

```
int x[5]={4,6,1.89,8};
cout<<x;
cout<<*(x+5);
cout<<*x+4;
```

Problem 2

```
int x[4]={67,9,2,78,45,54};
cout<<x;
for(int i=0;i<4;i++)
cout<<x[i];
```

Problem 3

```
int x[4]= {67,9,2,78,45,54};
int y=2;
cout<<*(x);
cout<<x;
cout<<*(x+y);
x=x+y;
cout<<x;
cout<<*x;
```

Problem 4

```
int x[2]= {67,9,2,78,45,54};
int y=2;
cout<<*(x);
cout<<x;
cout<<*x+y;
```

Chapter 3

Link List

Chapter Objective

- Describe and create link list
- Display/Traversal in a link list
- Inserting a node at various position
- Deletion of any node in a link list
- Reversing the link list
- Operator overloading in a link list
- Application of a link list
- Limitation of a link list and defining circular link list
- Benefits of circular link list .

3.1 INTRODUCTION

Array suffers from the size limitations, specifically, if the number of elements to be stored in the array is not known in advance. In such cases usage of array is inefficient. Consider a case in which array size allocated is more but items to be inserted are less. This will lead to wastage of array space. On the other hand, if size of an array defined is less and elements to be stored in the array are more, in that case many elements would not be placed inside an array. To address the variable need of data storage, a new data structure known as link list has been envisioned.

This chapter highlights many significant features of the link list that includes:

- Creating the link list.
- Display of the link list.
- Inserting the node in a link list.
- Deleting the node from a link list.
- Usage of operator overloading in a link list(+operator)
- Searching an item from a link list.
- Orderly link list
- Reversing the link list.
- Circular Link list

3.2 CREATION OF LINK LIST

The complex part of the link list is its creation. It can be accomplished by creating the nodes and then connecting them immediately after creation. In a link list, the smallest entity is node and is created dynamically. Each node consists of two parts

- Data Part
- Address part

Data part is capable to hold the data in a node. This data may be simple type or complex type like structure or any other data type created by the user. Whereas, address part represent the address of next node. Initially, there is only one node therefore, it's next pointer holds NULL. Structure of a node is illustrated in the figure 3.1.

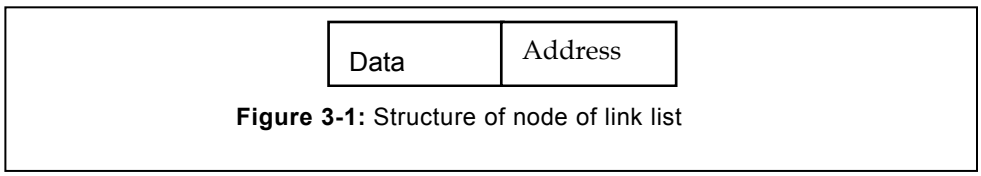


Figure 3-1: Structure of node of link list

In the figure 3.1, data part will hold the user's data, whereas address part points to the address of a next node. Structure of a node can be defined as:

```
// Creating the template for the node
class node
{
public:
int data;
node *next;
};
```

During the dynamic creation of first node 'new' operator is used for allocation of address of next node. It is introduced in C++, since it was lacking in 'C' language. New operator is an improvement over the earlier malloc function of 'C'. For instance, no casting is needed in 'new' operator, whereas it was needed in malloc function. To create a node, we can use the syntax as given below:

node *one;

one=new node;// node created

one->data=10;

one->next=NULL;

A. Creation type

During the creation of a link list, it can grow in forward or the backward direction. This is entirely governed by the method of insertion of a new node either:

- At the end of a link list, or
- At the beginning/front of a link list

Figure 3-2: Creation of a node in a link list

In the insertion from the end, a new node is placed at the end of an existing link list. For instance, inserting the node with value 20, in the link list already having the node 10 is illustrated below. Here 20, will be placed after 10 and the pointer will be appropriately adjusted. After inserting 20, consider that we have to insert 30 into the existing list. Same is illustrated in the figure 3.3(b). This is known as backward growth of a link list. Since new node is attached at the end.

Simirlarly, we can create a link list that can be forward growing. In forward list, the new node is inserted at the beginning, instead of at the end. Same has been illustrated in the figure 3.4. Consider the case of figure 3.4. In this case, a node with the value of 5 already existed. The new node having value 10 is created; correspondingly it will be inserted at the beginning of the existing link list. Similarly, the new node that has arrived is 20. It will be positioned at the beginning of 10 (refer figure 3.4). This type of growth in a link list is known as forward growing link list.

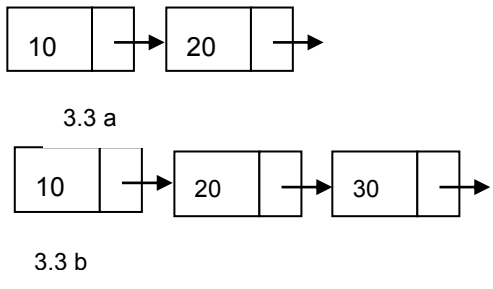


Figure 3-3: Creation of link list from the end

B. Display the node

Once the link list is created, irrespective of the forward growing or the backward growing, we would like to view the data that is hold by various node of a link list. To display the node element, we begin from the start i.e. first node and continue to move towards the end node. Next pointer of the last node points to the NULL that can be used to stop the traversal. Logic for the traversal has been depicted in the following program:

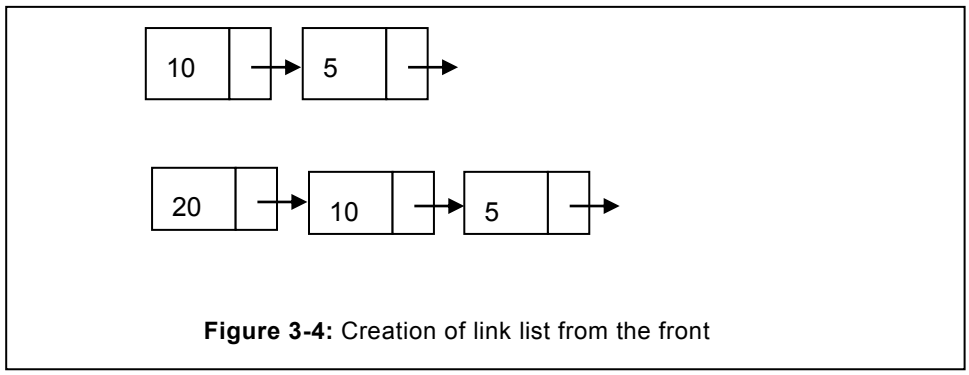


Figure 3-4: Creation of link list from the front

```
dnode *current;
current=first     //initialize with the first pointer
while (current!=NULL)   // moving upto the end of a list
 {
cout<<"current->info"<<current->info; //displaying info part
current=current->next; //move to the next node
}
```

The complete program for creation and display of a link list has been depicted as follows:

```
Program1: Create and display the node of a list. In the program, node is to be inserted from the end.
// class to define the structure of a node
class node
{
public:
int info;
node *next;
};
// class of a link list
class list
 {
node *first, *end;      // creation of first and end pointer
public:
//  Defining constructor for the list
list()
 {
```

```
first=NULL;
}
// function to create the link list from the end
void create_end(int);
// function to display the elements of link list
void display();
};

// function to position the new node at the end and connecting it
void list::create_end(int el)
       {
      node *current,*temp;
       if(first==NULL)  // if first node does not exist
        {
         first=new node;
         first->data=el;
         first->next=NULL;
         end=first; //initialize end with first (now both are same)
         return;
         }
         //if first node exists, follow the line given below
         current=new node;
         current->data=el;
         current->next=NULL;
         end->next=current;   // connecting end with the current
         end=current;         // making the current as end node
        } //end of create_end
```

```
           //function to display the content in a linked list
           void list::display()
           {
              node *current;
              current=first;
              cout<<"Linked list:"<<endl;
              while(current!=NULL)
```

```
            {
                cout<<current->data<<" -> ";
                current=current->next;
            }
        }
```

C. Creating the list by inserting at the beginning

Link list can also be created by inserting the element at the beginning. The new node that is attached becomes the first node of the list. For instance, if list is to be created for the series of item 10, 20, 30, 5, 9, 100, then the list will grow as illustrated in the figure 3.5.

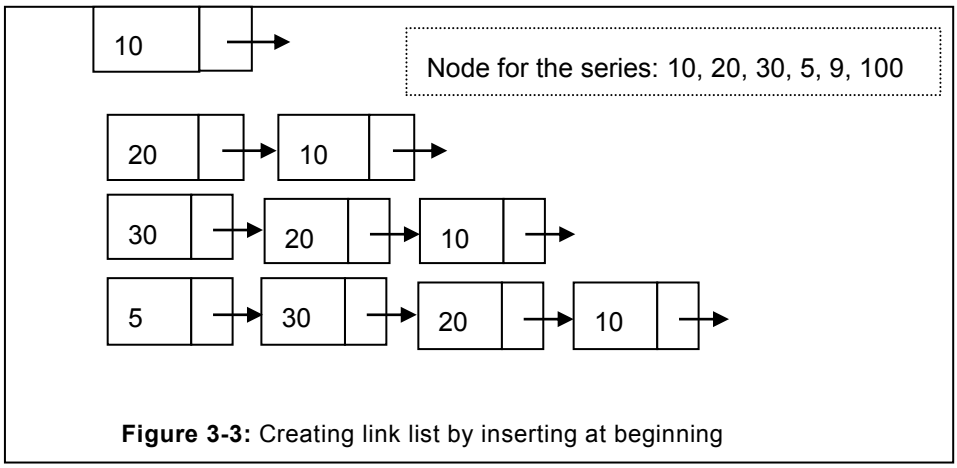


Figure 3-3: Creating link list by inserting at beginning

Once the new node is created then it will be connected to the first node of an already existing list, as a result the existing first node will move to the second position. Therefore, the new node should become the first node. The complete program for the insertion of a node in a link list has been depicted below:

```
//function to create the list from the beginning
      void list::create_beg(int el)
       {
        node *current,*temp;
      // if the list does not exist
      if(first==NULL)
        {
         first=new node;
         first->data=el;
         first->next=NULL;
         end=first;
```

```
        return;
        }
      // If the link list exist then program will start from here
       current=new node;
        current->data=el;
        current->next=first; //link current to first
        first=current; // make the current node as first node
       } //end of create_beg
```

3.3 INSERTING THE NODE IN AN EXISTING LINK LIST

In an existing link list, node can be inserted anywhere as intended by the user. However, for the program side, we need to consider two cases that will cover all positions of insertion. These situations can be:

- At the first position
- Position, other than the first.

All these cases along with the method of connecting them have been depicted in the following sub-sections.

3.3.1 Inserting at the first position

Consider a link list as illustrated in the figure 3.6. In this list, a new node is to be placed (inserted) at the beginning. To accomplish this objective, first we have to create the new node and attach this new node before the existing first node. This operation is illustrated in the figure 3.6.

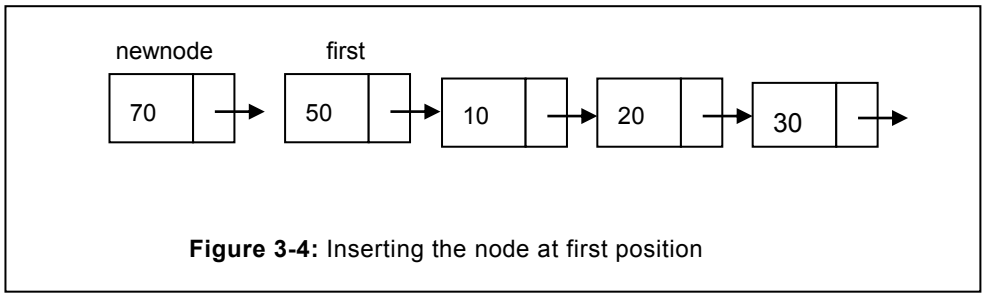


Figure 3-4: Inserting the node at first position

Due to insertion of a newnode at the beginning, status of existing first node will change. As first node will become the second node and the newnode will result into first node. Consequently, the first pointer has to be shifted to the newnode. The complete sequence of operation has been described as:

Method of inserting at first position

To insert the node at the first position, the following sequence needs to be followed.

- Create the newnode
- Link it to the first node
- Change the position of first node with this newnode.
- Return the pointer of this new first node

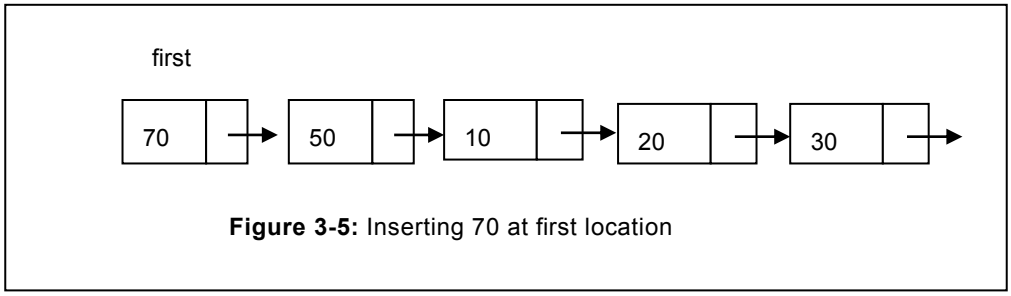


Figure 3-5: Inserting 70 at first location

Assume that the link list comprise of the ites as: 50,10,20,30. The new element is to be inserted is '70'. The first node is now moved to the second node and newnode has become the first node. The resultant link list has been illustrated in figure 3.5. Code is:

```
newnode=new node;
newnode->data=el;
newnode->next=first;
first=newnode;
```

3.3.2 Inserting the node other than the first position

To insert the node in between, we will determine the two nodes where new node is to be

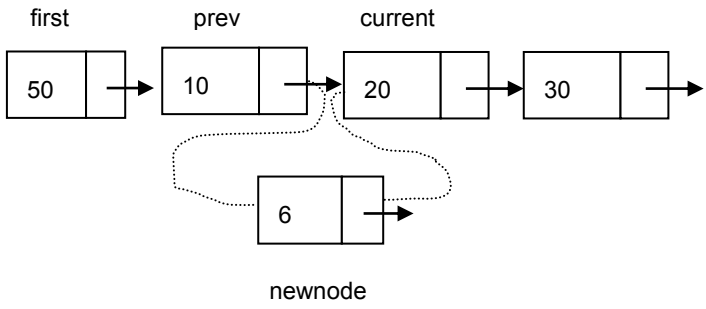


Figure 3-6 (a): Inserting the node in between

inserted. The node that will be before this node will be known as previous node and the node after this node will be known as forward node. In our program, we have named forward node as current node. Same is illustrated in the figure 3.6.

In the existing link list, pointer of previous node needs to be connected to the newnode, whereas pointer of the newnode to be connected to the current node.

```
prev->next=newnode;
newnode->next=current;
```

The complete method of inserting the node in a link list has been depicted in the insert function. In the discussed function, we accept the position, that is passed as an argument and examine it with the number of nodes that a link list is consisting of at that specific point in time. Users intended position should not be more than the existing list size (number of nodes) plus 1. For instance, if the total numbers of nodes are n, then the position supplied should not be more than n+1.

Program: Insert the node at any position in a list

Solution

```
//function to insert at particular position
       void list::insert(int el, int pos)
          {
        node *current,*newnode,*prev=NULL;
        int cnt=count();//count number of nodes
        if(pos<=cnt+1)
           {
            current=first;
            // Create a new node
           newnode=new node;
            newnode->data=el;
            newnode->next=NULL;
            int i=1;
            //moving to the appropriate position
            while(i<pos)
             {
            prev=current;
            current=current->next;
            i++;
            }//end of while
```

```
            if(prev==NULL)// to be inserted at first position
            {
             newnode->next=first;
             first=newnode;
             return;
            } //end of inner if
      // if node is to be inserted is other than the first node
      newnode->next=current;
          prev->next=newnode;
           }  //end of outer if
             else
             cout<<"Size less than position, Try again-..\n";
           }//end of function
```

The above program is well suited for insertion at last position. Since, during positioning at last position involves placing of the newnode at last, this is preceded by connecting the existing last node with the newnode. Insertion of a newnode is illustrated in the figure 3.7.

Code written for the in-between will also support the insertion of the node at last position.

```
prev->next=newnode;            (1)
newnode->next=current          (2)
```

However, the prev node will point to the last node, whereas the current will point to the last node. Since the code of (1) will connect the last node to the newnode therefore one objective will be completed. Whereas the code (2) will connect the newnode to the current node which is already resulted in NULL, as illustrated in the figure 3.7. Consequently, the code will work for inserting the node at any position other than the first position.

3.3.3 Deleting the node in a link list

Link list holds the data similar to other type with the difference that it is dynamic in nature. It may happen that the data stored in a link list may no longer needed. This leads to need of another operation term as deletion to be envisioned on a link list. Snce, link list is connected with the help of pointers, therefore, due care need to be exercised to ensure that link list remains connected even after the deletion of a node(s). During deletion of a node, the following possibilities related to the position need to be taken into account.

- Deleting first node.
- Deleting other than the first node.

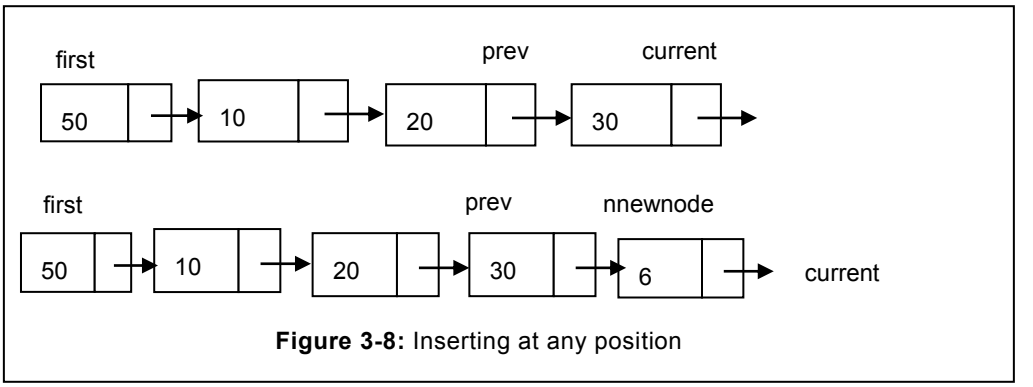


Figure 3-8: Inserting at any position

3.3.4 Deletion of first node

Once the user is aiming to delete the first node, in that case pointer of first node will be lost. Therefore, to delete the first node the following sequence of steps need to be adhered.

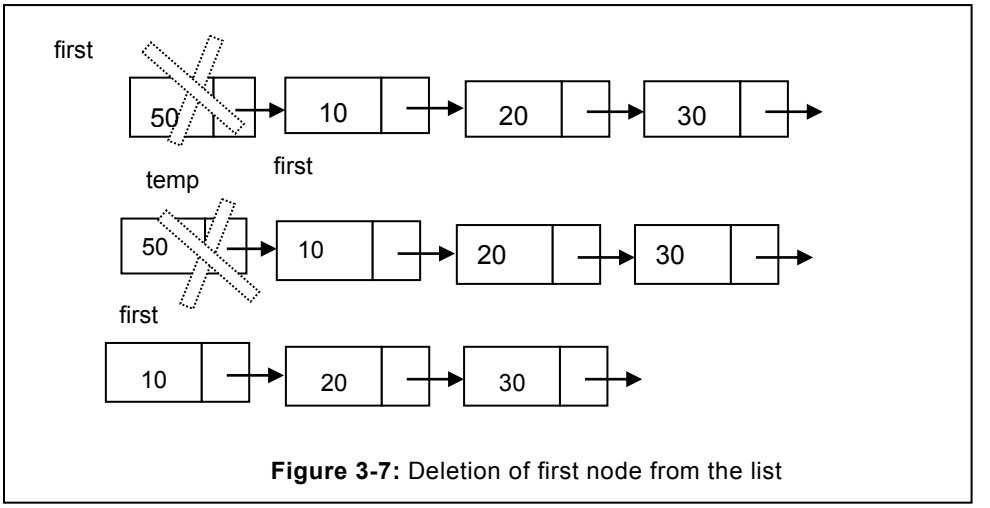


Figure 3-7: Deletion of first node from the list

- Hold the first node.
- Find out the second node.
- Store the first node into temp.
- Assign the second address to the first node.
- Delete the temp (the previous first) node from the list to free the resources.

Code to delete the first node is given here under.

```
node *temp;
temp=first; //storing the first node into temp
```

```
first=first->next; //moving the first pointer to the next node
delete temp; //delete the temp node and free the space
```

3.3.5 Deletion of other than the first node

The complete method of deleting the node has been illustrated in the figure 3.8.

Suppose users need to delete the third node that has the value 20. To delete this node, following steps need to be followed.

- Traversing upto the target node, at the same time holding pointer of the previous node.
- Store the pointer of target node (current node, node to be deleted i.e. 20) into temp.
- Connecting previous node to the next node of the current node(node to be deleted).

To delete the node, users need to traverse upto current node by preserving the pointer of the previous node. In case pointer of forward node is NULL, then previous pointer will point to the NULL. Similarly, if the node to be deleted is the first node, in that case previous will be NULL. In such cases first is moved to first->forward. The node that is just traversed i.e. previous node is deleted. Finally, to delete the node in between, previous next is connected to the next pointer of target node. At the same time, target i.e. current is deleted to release the free space from the heap.

```
temp=current;
prev->next=current->next;
delete temp;
```

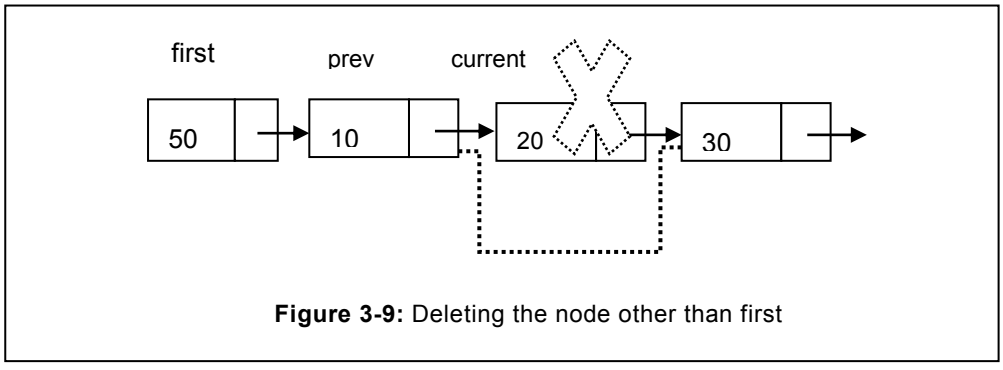


Figure 3-9: Deleting the node other than first

The complete program is discussed in the following program.

```
Program to delete the node from a link list
void list::deletenode(int element)
       {
       node* current,*prev=NULL;
       int found=0;
```

```
 current=first;
 while(current!=NULL)
{
if(current->data==element)
  {
  found=1;
  break;
  }
prev=current;
current=current->next;
}
if(found)   // if node to be deleted is found
{
if(prev==NULL)    // if first node
  {
  prev=current->next;
  first=prev;
}
else  // for other than the first node
prev->next=current->next;
delete current;
}
else
cout<<"element not in the list"<<endl;
 }//end of deletenode
```

3.4 SEARCHING THE ELEMENT IN A LINK LIST

Link list is a linear data structure; consequently elements need to be searched from the beginning. In the list, elements is to be searched at first node. This is followed by visiting the subsequent nodes till elements is not found or the list is not exhausted. In the best case, element to be searched may be found at first position itself. In worst case, element to be searched may be found at last position. Therefore, in best case, complexity will be O(1). If the number of nodes are ‘n’ then in the worst case the complexity will be O(n). Chances of in-between also exist. However, in asymptotic notation, only best and worst cases are only taken into account. Program to search an element has been depicted as follows:

```
Program to search the element in a link list
           int list::search(int el) //search function
           {
             //Initially no item is found, therefore its value is
initialize with zero.
              int found=0;
              node *current=first;
              while(current)
               {
                 if(current->data==el)
                 {
                  found=1;
                  break;
                  }
                 current=current->next;
           }
            return found;
         }//end of search function
```

3.5 ORDERED LINK LIST

In the orderly link list, items are arranged in certain order, i.e. in ascending or descending order. In orderly link list, overhead is involved at the time of creation. However, it offers excellent performance during searching, finding out the minimum, or the maximum element, etc. Since during search, all elements are not to be searched instead as soon as elements with the higher value is encountered loop needs to be terminated. Therefore, orderly link list gives better performance relative to ordinary link list.

In an orderly link list, items are arranged in ascending order from construction stage itself. Whenever a new node is to be inserted in an orderly link list, logic of insertion need to be applied. For instance, consider a list of numbers 20, 5, 13, 90, 100, 25 has to be inserted in an orderly list.

Detailed steps have been described as follows:

- If the link list is empty, insert the item at the first position.
- In all other cases, keep on moving to the right till element with greater value is not encountered.
- Once the larger element is encountered, terminate the traversal.

Create the node and insert the element into it. Traverse the appropriate position in the link list and connect the newnode to it, at the same time connect the nextnode of the earlier node to

the next pointer of the newnode. This process need to adopted for placing all the node in an orderly list. Creation of an orderly link list has been illustrated in the figure 3.10. Intially 20 is to be placed in an orderly link list. Since, list is empty therefore this node will become the first node. Afterwards, 5 is to be placed in this orderly list. Since, 5 is less than 20, therefore, it will

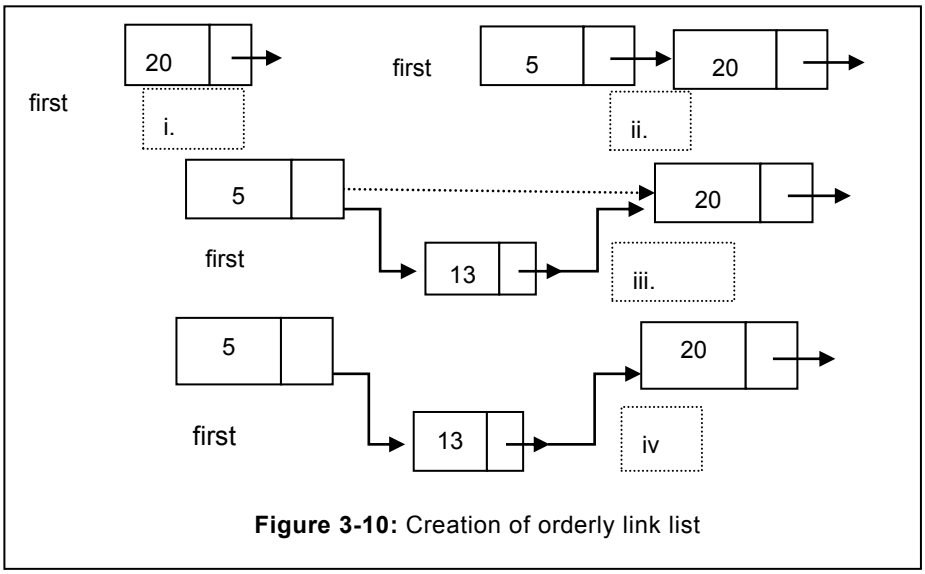


Figure 3-10: Creation of orderly link list

be placed before 20 and pointers to be adjusted as already depicted.

```
//Program to create and insert node in an orderly link list.
#include<iostream.h>
#include<conio.h>
class node
{
 public:
 int data; //data part of the list
 node *next;// address of next node
};

class list
{
  //creating the first pointer of the list
  node *first,*end;
  //public part of class
   public:
```

```
 //linked list constructor
  list()
    {
    first=NULL;
    }
    void order_list(int);
    void display();
};
void list::order_list(int element)
 {
    if(first==NULL)   // if link list is not existing
  {
  first=new node;
  first->data=element;
  first->next=NULL;
  return;
  }
  //if list is not empty
  node *temp,*prev=NULL,*current;
  temp=new node;
  temp->data=element;
  //temp->next=NULL;
  current=first;
  while((current->data < element)&& (current!=NULL))
  {
     prev=current;
     current=current->next;
     }
    if(prev==NULL)    // if node to be inserted at first position
    {
    temp->next=current;
    first=temp;
    }
    else
    {
```

```
      prev->next=temp;
      temp->next=current;
      }
 }// end of function
           void list::display()
           {
              node *current;
              current=first;
              cout<<"Ordered linked list:"<<endl;
              while(current!=NULL)
              {
                 cout<<current->data<<" -> ";
                 current=current->next;
              }
              cout<<endl;
            }
int main()
 {
 list l1;
 clrscr();
 cout<<"Creating ordered list"<<endl;
 l1.order_list(29);
 l1.order_list(14);
 l1.order_list(70);
 l1.order_list(12);
 l1.order_list(30);
 l1.order_list(121);
 l1.display();
 getch();
 return 0;
 }
```

3.6 REVERSE A LINK LIST

Reversing a link list is a complex operation and requires deep understanding of the pointers. Losing any pointer will result in the lost of a node and information. During the reversing a link

list, we traversed from the front towards the last node. During this traversal, the node that is accessed at last will be attached at the beginning of a new link list, similar to the insertion from the beginning operation. We need to reverse this link list as illustrated in figure 3.11. In the considered link list, the first node is 50, initially, it will be attached at the beginning.

On the arrival of newnode i.e. 10, it will be attached at the beginning of 50 as illustrated in the figure 3.11 and 3.12.

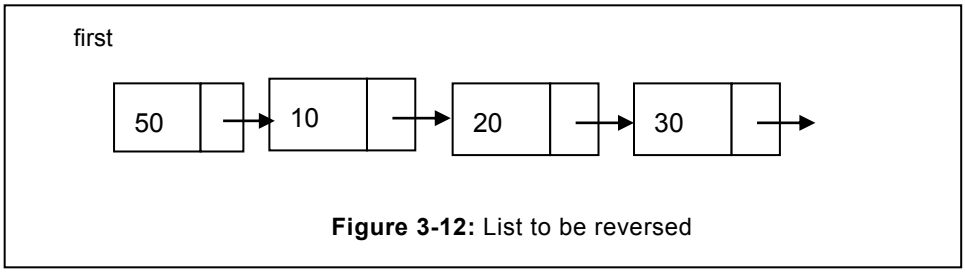


Figure 3-12: List to be reversed

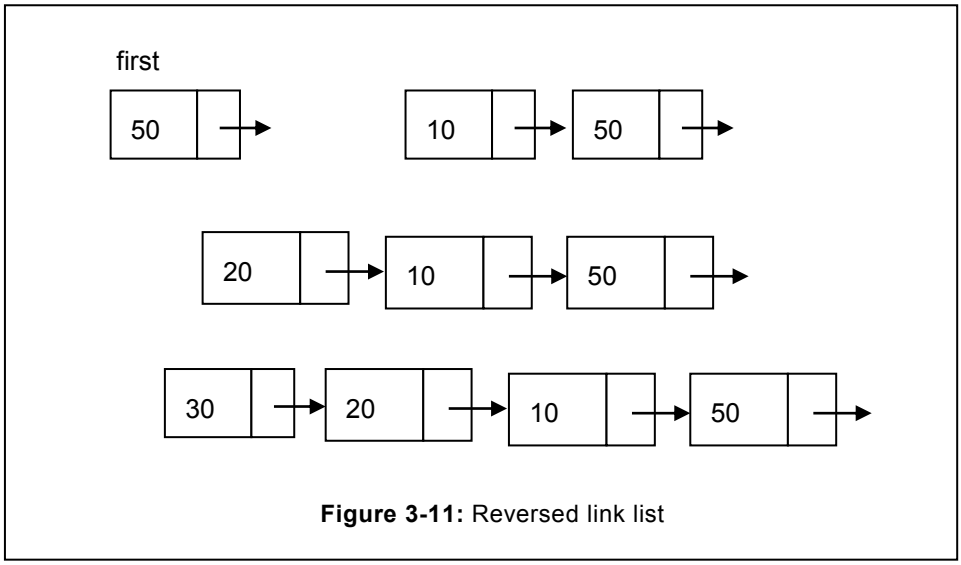


Figure 3-11: Reversed link list

Reversal of a link list is demonstrated in the figure in step by step manner.

```
Problem: Write a program to reverse a link list
Solution:
// Reversing a list where the first node is pointed by first pointer.
void list::reverse()
            {
```

```
            cout<<" Reverse of a list is ::"<<endl;
               // Initialize all the node with NULL.
                     node *current,*forward,*back=NULL;
               current=first;
               forward=current->next;
               while(forward!=NULL)
               {
                current->next=back;
                back=current;
                current=forward;
                forward=forward->next;
                }
                 current->next=back;
                 first=current;
}
}
```

3.7 MERGING TWO LINK LIST

Due to the variety of reasons, we need to merge the link list so that data available in more than one link list can be accommodated in a single list. During merging, due care need to be exercised in order to ensure that items of both the link list that need to be merged into third remain in the shorted order. Merge link list, would result in better performance and management. We have created the third list in which content of both the list is stored. Detailed method and program to merge the two link list has been described hereunder.

```
list list:: merge(list l1,list l2)// To merge list 1 and list 2.
         {
         list l3;
         node *one,*two,*current;
         one=l1.first;
         two=l2.first;
         if(one->data < two->data)    // if the first list is having
smaller //element
           {
           l3.first=current=one;
           one=one->next;// move to the next node
           }
```

```
         else
         {
         l3.first=current=two; // initialize with the first node of
second list
         two=two->next;
         }
         while((one!=NULL) && (two!=NULL))
           {
           if(one->data < two->data)
           {
          current->next=one;  // connect current to the one
          current=one;     // move one to the current
          one=one->next;  // move to the next node
           }
           else
           {
           current->next=two; // connect current to the two
          current=two;      // move two to the current
          two=two->next;  // move to the next node
           }
           }// end of while
           if(one)          // if list one is not exhausted
            {
            current->next=one;
            }
            else if(two)   // if list two is not exhausted
             {
             current->next=two;
             }
           return l3;
       }
```

3.8 USE OF OPERATOR OVERLOADING IN A LINK LIST

Operator overloading allows the programmer to define the operators to function as per programmer's way, instead of the way defined for the primitive data type. However, use of the overloading is restricted at the place and scope of definition, beyond which it can be used

according to its normal definition. For instance, '+' operator is defined to sum two integer number. However, it can also be used to concatenate the two strings or to join the two link list. Similarly, ++ operator can be used to move to the next node of the link list. Operator '=' can also be used to compare the number of nodes in two link list. It returns whether compared link list are equal in length or not.

We can use the '+' operator to join two link list. In our case, we would like to write the statement, List3=list1+list2, this should result in list3 that should consist of both list1 and list2. To accomplish the above objective, the code is written below:

```
list operator +(list l2)
 {
while(first->next!=NULL)      // Move to the last node of the list1
first=first->next;
first->next=l2.first;   // connect the last node to the first node
of list l2
return *this;     // return the first list
}
```

To achieve the above task, first we need to create the list l1 and the list l2. Afterwards, we can call them using the + overloaded operator. Code for the main function is described below:

```
int main()
{
list l1, l2, l3;
l1.create_beg(10);
l1.create_beg(20);
l2.create_beg(6);
l2.create_beg(33);
l3=l1+l2;
```

Using the same method, other operators like ==, < , > etc can be used. Readers are advised to write the program of operator overloading for the aforementioned operator.

3.9 SELF-ORGANIZING LIST

Link list is the linear data structure,as a reslt, whenever user needs to carry out any operation, he has to start from the first node and traverse the subsequent node one after another node till target node is not reached or list is exhausted. Therefore, efficiency of a list is dependent on the size of a list. If the list is large in that case, efficiency of the link list drops. To overcome this limitation, a new version of link list known as self-organizing list is devised. Self-organizing list is capable of reorganizing the node based on its usage criterian. Consequently, it yields better performance relative to the normal link list.

Based on our day today experience, we know that in a given list some items are more frequently used relative to others. Self-organizing list draws the advantage of this trend and organizes the node according to the accessibility. Possible method by which list can be organized are:

- Ordered based.
- FCFS based.
- Count method.
- Shift based method.

3.9.1 Ordered List

In the ordered list, items are arranged based on their values. This signifies that items are arranged in ascending or descending order. In the event of arrival of a new item, it is inserted in the existing link list based on its value. This position can be at the beginning, at the center or at the end. We have already discussed the orderly link list in detail in earlier section. However, differences has been discussed herein, and same is again illustrated in figure 3.13.

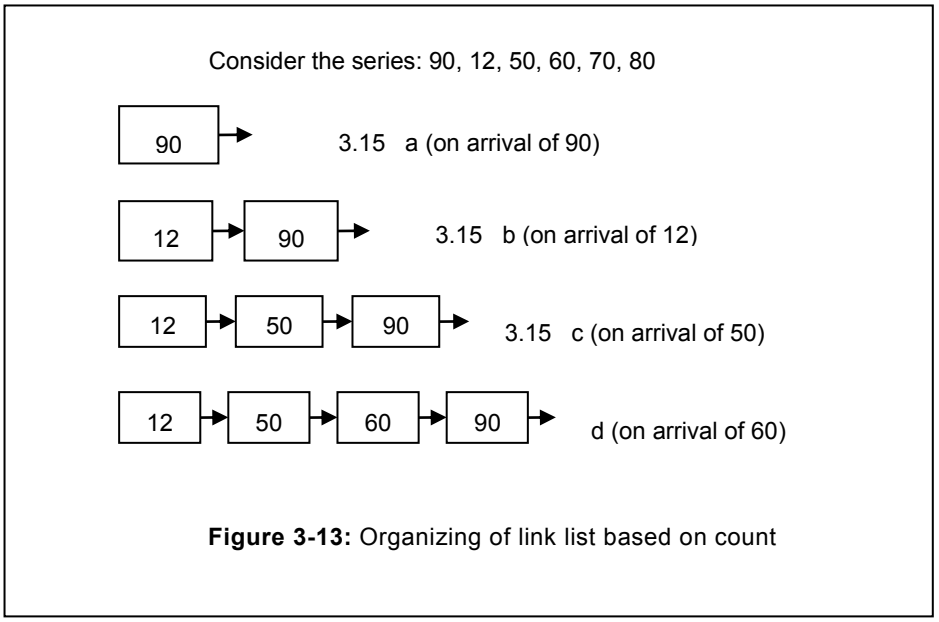


Figure 3-13: Organizing of link list based on count

Ordered link list has the following advantages over the ordinary link list.

- During insertion of the node, we traverse till a node with greater value is not encountered or the list is not exhausted.
- During removal again traversal is taking place till the node comprising of the target value is not found or nodes are less than the node to be searched, after which searching terminates.

Since, we do not need to traverse all the node, consequently, it yields better performance relative to simple link list.

3.9.2 Count based link list

In the count based link list, in each node an additional data member named ‘count’ is provisioned. Value of the count is assigned based on the accessibility of the node. On every access, value of the count is increased by 1. During arrangement, node with the maximum count is placed at the first position whereas, the node with minimum count is placed at the end.

In the 3.14 a, structure of the list has been demonstrated. In the figure 3.14 b, nodes are organized based on their count. Node with the maximum count is placed at the first position. In the 3.14 c, if the count changes, consequently, the nodes are rearranged based on their count. Same has been illustrated in figure 3.14 c.

3.9.3 First come first serve (FCFS)

In the first come first serve method, nodes are arranged on the order of their arrival. Nodes

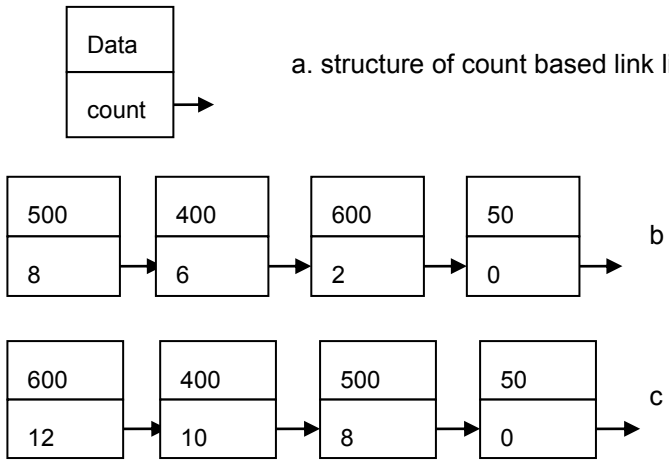


Figure 3-14: Organizing of link list based on count

arrived late are placed at the end. Creation of a link list based on the FCFS has been illustrated in the figure 3.15 . Initially, 500 is arrived, since there is no element therefore, it is placed at the first position. It is followed by 400, correspondingly, it is placed after the existing nodes, here it is 500, refer figure 3.15 b. Eventually, 12 is arrived at last, correspondingly, a new node is created and attached at the last of existing link list, refer figure 3.15 c. Same procedure will be applicable for other elements.

It is apparent from the above description that the FCFS follows the sequence similar to that of insertion at the end. FCFS based algorithms are immensely useful in the scenarios where we

are aiming to serve the elements based on their arrival time. For instance, in railway reservation.

3.9.4 Shift based method

In the shift based method, node accessed is shifted one position toward the beginning side. This method is fairly similar with the count based method. Working of shift based method has

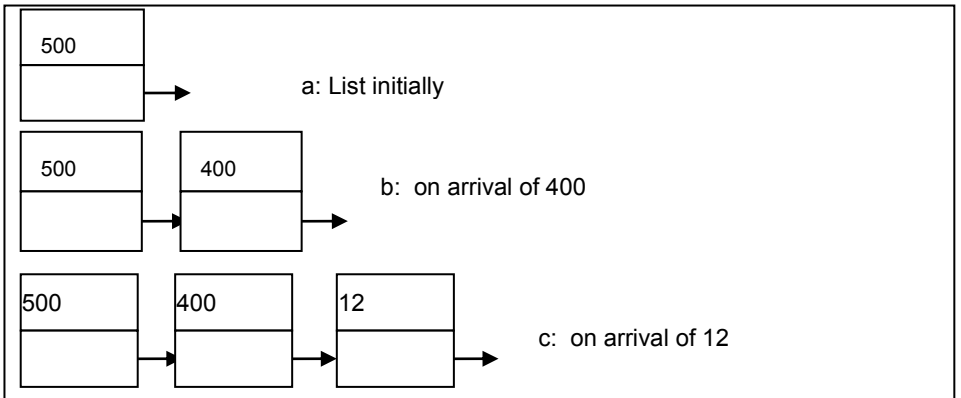


Figure 3-15: Organizing of link list based on FCFS

been demonstrated in the figure 3.16. However, in count based method, position is primarily decided by the count method, but in the shift based method, it is decided by the last accessibility. For instance the node which is available at the first position might have been accessed 5 times that's why it has been placed at first position whereas the second node might have not been accessed even once then also it is in the second position. As soon as second node is accessed it will replaced with the first node irrespective of the time it has been accessed previously.

3.9.5 Skip List

To significantly improve the performance a new type of list known as skip list has been devised. Skip list is the special case of orderly link list in which while organizing and traversing we are skipping some of the nodes. In this type of list, instead of traversing node by node, a number of nodes are skipped to improve the performance during traversal or searching of a node. This type of link list has substantial advantages over other types of link list described earlier. Two widely used skip lists are :

- Even skip list
- Random skip list

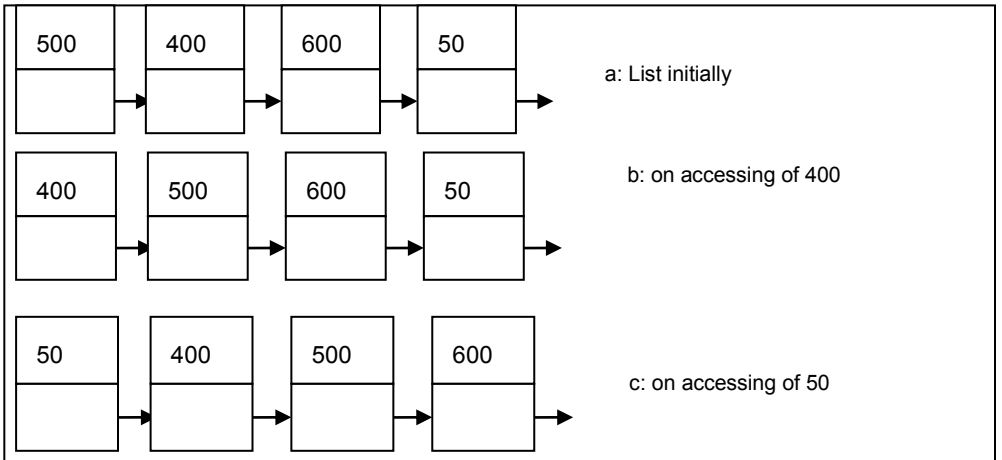


Figure 3-16: Shift based organization of a link list

3.10 CALLING FROM THE MAIN FUNCTION

Various functions are defined to conduct the various link list activities. These functions need to be called from the main function. Various parameters and method of calling the function has been defined as follows:

```
// Calling the function from the main function
int main()
{
int el;
list list1,list2,list3,list4;
clrscr();
//creating the list from beginning calling
list1.create_beg(40);
list1.create_beg(2);
list1.create_beg(14);
list1.display();
cout<<" inserting 100 at 1 position"<<endl;
list1.insert(100,1);
list1.display();
cout<<" inserting 34 at 3 position"<<endl;
list1.insert(34,3);
```

```
list1.display();
cout<<" inserting 200 at 6th position"<<endl;
list1.insert(200,6);
list1.display();
cout<<" Total number of nodes now:\t"<<list1.count()<<endl;
cout<<" List after deleting 200"<<endl;
list1.deletenode(500);
list1.display();
cout<<" List2 inserting from the end"<<endl;
list2.create_end(1);
list2.create_end(7);
list2.create_end(3);
list2.create_end(4);
cout<<" List2 Now :"<<endl;
list2.display();
cout<<" Inserting the node in list2"<<endl;
list2.insert(70,1);
list2.display();
list3=list1+list2;
cout<<" List3 after list1+list2"<<endl;
list1.display();
if(list2.search(30))
cout<<"Element found"<<endl;
else
cout<<"Element not in the list"<<endl;
if(list4.emptylist())
cout<<"List4 is empty"<<endl;
if(list1.emptylist())
cout<<"list1 is empty"<<endl;
else
cout<<"list1 not empty"<<endl;
getch();
return 0;
}
```

3.11 CREATING THE LINK LIST USING TEMPLATE

In the examples discussed earlier, we have considered the integer list to insert the items. Similarly, if we need to create the link list for the character, a new link list for the character type needs to be created. This results in rewriting of the existing code more than once. In the new written code, error may creep in that result in retesting of the program that was already tested. To overcome this limitation, we can create the link list using template. Complete program need to be written once, and will work for the all types of data. (Readers not aware of the concept of template are requested to refer the appendix A)

```
Link list using template
//Singly linked list using template
#include<iostream.h>
#include<conio.h>
template<class T>
class node
{
 public:
 T data; //data part of the list
 node<T> *next;// address of next node
};
template<class T>
class list
{
  //creating the first pointer of the list
  node<T> *first,*end;
  //public part of class
   public:
   //linked list constructor
    list()
      {
      first=NULL;
      }
      // create the link list by inserting at beginning
      void create_beg(T);
      // create the link list by inserting at end
      void create_end(T);
```

```
// Insert the element at specific position
void insert(T t1,int pos);
//To delete the node consisting the value
void deletenode(T el);
//To search the element in the list
int search(T);
// To count number of nodes
int count();
 //function to traverse the list
void display();
//function to reverse the list
void reverse();
//To find out list is empty
int emptylist();
// To concatenate the two list
 list operator+(list);
};
template<class T>
int list<T>::emptylist()
{
int empty=0;
if(first==NULL)
empty=1;
return empty;
}//end of emptylist
template<class T>
void list<T>::deletenode(T element)
 {
 node<T>* current,*prev=NULL;
 int found=0;
 current=first;
 while(current!=NULL)
{
if(current->data==element)
  {
```

```
        found=1;
        break;
        }
      prev=current;
      current=current->next;
      }
      if(found)
      {
      if(prev==NULL)
        {
        prev=current->next;
        first=prev;
      }
      else
      prev->next=current->next;
      delete current;
      }
      else
      cout<<"element not in the list"<<endl;
       }//end of deletenode
       //function to create the list from end
      template<class T>
      void list<T>::create_end(T el)
       {
       char ch;
       node<T> *current,*temp;
       if(first==NULL)
        {
         first=new node<T>;
         first->data=el;
         first->next=NULL;
         end=first;
         return;
         }
         current=new node<T>;
```

```
        current->data=el;
        current->next=NULL;
        end->next=current;
        end=current;
       } //end of create_beg

      //function for creating linked lists from
      template<class T>
      void list<T>::create_beg(T el)
      {
      char ch;
      node<T> *current,*temp;
      if(first==NULL)
       {
        first=new node<T>;
        first->data=el;
        first->next=NULL;
        end=first;
        return;
        }
        current=new node<T>;
        current->data=el;
        current->next=first;
        first=current;
       } //end of create_beg
     //function to insert at particular position
      template<class T>
      void list<T>::insert(T el, int pos)
         {
       node<T> *current,*newnode,*prev=NULL;
       int cnt=count();//count number of nodes
       if(pos<=cnt+1)
          {
           current=first;
           newnode=new node<T>;
```

```
            newnode->data=el;
            newnode->next=NULL;
            int i=1;
            //moving to the right position
            while(i<pos)
             {
            prev=current;
            current=current->next;
            i++;
            }//end of while
            if(prev==NULL)
            {
             newnode->next=first;
             first=newnode;
             return;
            } //end of inner if
          newnode->next=current;
          prev->next=newnode;
           }  //end of out if
           else
           cout<<"List size less than position, Try again……"
           }//end of function insert_beg

           //search function
           template <class T>
           int list<T>::search(T el)
           {
              int found=0;
              node<T> *current=first;
              while(current)
               {
                 if(current->data==el)
                 {
                  found=1;
                  break;
```

```
                }
               current=current->next;
        }
         return found;
      }//end of search function
        //overloading + operator
        template<class T>
        list<T> list<T>::operator +(list<T> l1)
        {
           node<T> *current;
           current=first;
           while(current->next!=NULL)
           current=current->next;
           current->next=l1.first;
           return *this;
        }
        //reverse function
        template<class T>
        void list<T>::reverse()
        {
           node<T> *a,*b,*temp;
           a=first;
           b=a->next;
           temp=b->next;
           a->next=NULL;
           while(temp!=NULL)
           {
               b->next=a;
               a=b;
               b=temp;
               temp=temp->next;
               //this->display();
           }
           b->next=a;
           first=b;
```

```
            }
            //counting nodes in a linked list
            template<class T>
            int list<T>::count()
            {
               node<T> *current;
               int c=0;
               current=first;
               while(current!=NULL)
               {
                   c++;
                   current=current->next;
               }
               return c;
            }
            //function to display linked list
            template <class T>
            void list<T>::display()
            {
               node<T> *current;
               current=first;
               cout<<"Linked list:"<<endl;
               while(current!=NULL)
               {
                  cout<<current->data<<" -> ";
                  current=current->next;
               }
               cout<<endl;
             }
// Declaring the main function
int main()
{
int el;
//creating the integer type template
list<int> list1;
```

```
clrscr();
//creating the list from beginning calling
list1.create_beg(40);
list1.create_beg(2);
list1.create_beg(14);
list1.display();
cout<<" inserting 100 at 1 position"<<endl;
list1.insert(100,1);
list1.display();
cout<<" inserting 34 at 3 position"<<endl;
list1.insert(34,3);
list1.display();
cout<<" inserting 200 at 6th position"<<endl;
list1.insert(200,6);
list1.display();
cout<<" Total number of nodes now:\t"<<list1.count()<<endl;
cout<<" List after deleting 200"<<endl;
list1.deletenode(500);
list1.display();
//list<int> list3;
list<char> list2;
cout<<" List2 inserting from the end"<<endl;
list2.create_end('A');
list2.create_end('Z');
list2.create_end('B');
list2.create_end('C');
cout<<" List2 Now :"<<endl;
list2.display();
cout<<" Inserting the node in list2"<<endl;
list2.insert('I',1);
list2.insert('T',3);
list2.display();
//list3=list1+list2;
cout<<" List3 after list1+list2"<<endl;
list1.display();
```

```
if(list2.search('I'))
cout<<"Element found"<<endl;
else
cout<<"Element not in the list"<<endl;
if(list2.emptylist())
cout<<"List4 is empty"<<endl;
if(list1.emptylist())
cout<<"list1 is empty"<<endl;
else
cout<<"list1 not empty"<<endl;
getch();
return 0;
}
```

3.12 CIRCULAR LINK LIST

In the link list, user can traverse only in one direction that is from the first node towards the last node. Once the last node is reached, there is no way to traverse backwards towards the first node, since link list allows to traverse only in one direction as it maintains the pointer of next node only. To overcome this limitation, a modification is suggested and same can be incorporated in the link list by connecting the last node pointer to the first node. List in which last node is connected to the first node is termed as circular link list.

3.12.1 Creating Circular link list

Creation of node in a circular link list is similar to linear link list. However, special attention is needed on the first node during the creation step. First node's next pointer in a circular link list will not be pointed to the NULL, instead it will be connected to itself as illustrated in the figure 3.17. When the circular list grows, new node is to be placed after this first node. Here again, next pointer of the newnode is to be pointed to the first node. This is accomplished by replacing the pointer of first node to the pointer of new node. Detail steps are illustrated in the following figures.

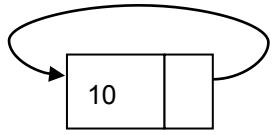


Figure 3-17: Connecting the first node to itself

To accomplish this task, code of the first node can be modified as follows:

```
first=new node;
 first->data=el;
 first->next=first;
 end=first;
// Now first and the end node is pointing to the same node.
```

At the time of adding a new node, next pointer of the last node needs to be pointed at new node, whereas the next pointer of the last node will be pointed to the first node. To accomplish this objective, we store the next address of the last node (which is pointing the first node) to the next node of a newnode. It is followed by connecting the last node with the newnode. Same is illustrated in the following figure 3.20.

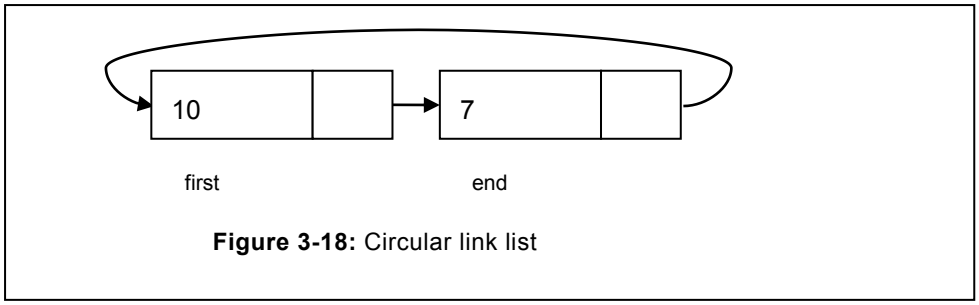


Figure 3-18: Circular link list

Attaching more node at the end will have to follow the same sequence of steps as the one already enumerated.

3.12.2 Inserting the node in between a circular list

Inserting the node in between the circular link list is not different than the simple link list till it does not disturb the first and last node.

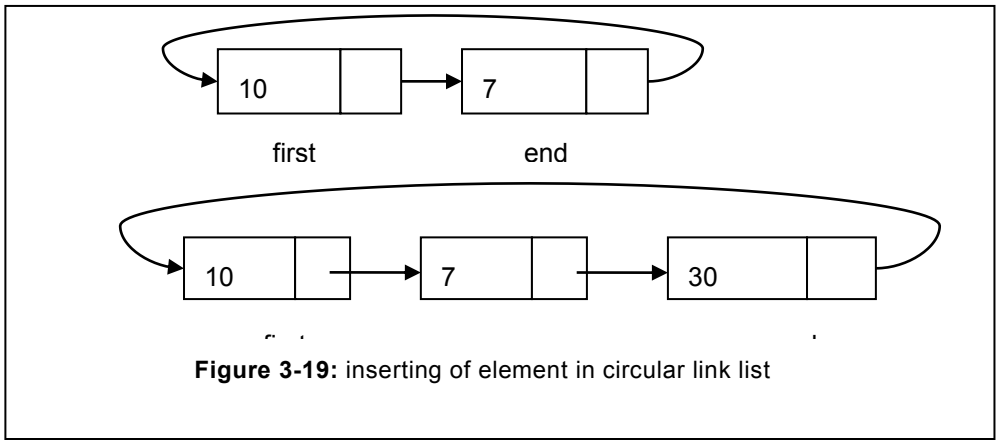


Figure 3-19: inserting of element in circular link list

In the event, if node to be inserted at the first position the following changes need to be incorporated:

- Connecting the new node to the first node.
- Connecting the pointer of the last node to this new first node.

Correspondingly, if the node to be inserted at the end of a list, in that case, following changes need to be carried out.

- Connecting the next pointer of the existing last node to the next pointer of a new last node.
- Connecting the existing last node pointer to this new last node.

The complete code to achieve the entire above objective has been described as follows.

```
// Program to attach the last node to the circular link list.
node *temp;
// create a new node.
temp=new node;
// assign data to the new node
temp->data=20.
//Connect the next node to the first node.
temp->next=end->next;
//connecting the node to the last node of the circular link list.
end->next=temp;
```

```
Program to attach a new node at the first position of a circular
link list.
node *temp;
// create a new node.
temp=new node;
// assign data to the new node
temp->data=20.
//Connect the temp node to the first node.
temp->next=first;
//connecting the last node  to the new first node (temp).
end->next=temp;
```

3.12.3 Traversal in circular link list

Traversal in a circular list is similar to that of linear link list. However, in a circular link list due care need to be exercised when the last node is reached, since the last node is connected to

the first node. Correspondingly, there is no end of the list and absence of NULL pointer at last node, during traversal if limit is set to NULL, therefore it will not terminate. Instead first node will be reached and traversal will run infinitely. In our code, we have started from the first node and traversed till the next node of the list is not pointing to the first node of the list. Since last node remained unvisited therefore provision is made to visit this last node. Code to traversed the circular linked list has been given herein.

```
while(current->next!=first)// Till next node is not the first node
              {
                 cout<<current->data<<" -> ";
                 current=current->next;
              }
              cout<<current->data<<" -> "; //visit the last node
```

Complete code for various operations incuding creation, traversal, insertion and deletion has been given hereunder. We have started with the structure of the node, that is followed by the other operations in a circular list.

```
//Creation of Circular link list
#include<iostream.h>
#include<conio.h>
class node
{
 public:
 int data; //data part of the list
 node *next;// address of next node
};

// Complete program and various operations on circular link list.
class list
{
  //creating the first pointer of the list
  node *first,*end;
  //public part of class
   public:
   //circular list constructor
    list()
      {
```

```
first=NULL;
}      // create the link list by inserting at end
void create_end(int);
// Insert the element at specific position
void insert(int el, int pos);
//To delete the node consisting the value
void deletenode(int el);
//To search the element in the list
int search(int);
// To count number of nodes
int count();
 //function to traverse the list
void display();
//function to reverse the list
void reverse();
//To find out list is empty
int emptylist();
// To concatenate the two list
 list operator+(list);
};
int list::emptylist()
{
int empty=0;
if(first==NULL)
empty=1;
return empty;
}//end of emptylist
void list::deletenode(int element)
 {
 node* current,*prev=NULL;
 int found=0;
 current=first;
 while(current!=NULL)
{
if(current->data==element)
```

```
    {
    found=1;
    break;
    }
prev=current;
current=current->next;
}
if(found)
{
if(prev==NULL)
    {
    prev=current->next;
    first=prev;
}
else
prev->next=current->next;
delete current;
}
else
cout<<"element not in the list"<<endl;
 }//end of deletenode
 //function to create the list from end
void list::create_end(int el)
 {
 char ch;
 node *current,*temp;
 if(first==NULL)
  {
   first=new node;
   first->data=el;
   first->next=first;
   end=first;
   return;
   }
   current=new node;
```

```
        current->data=el;
        current->next=end->next;
        end->next=current;
        end=current;
       } //end of create_beg

      //function for creating linked lists from
     /* void list::create_beg(int el)
      {
      node *current,*temp;
      if(first==NULL)
       {
        first=new node;
        first->data=el;
        first->next=NULL;
        end=first;
        return;
        }
        current=new node;
        current->data=el;
        current->next=first;
        first=current;
       } //end of create_beg  */
     //function to insert at particular position
      void list::insert(int el, int pos)
         {
       node *current,*newnode,*prev=NULL;
       int cnt=count();//count number of nodes
       if(pos<=cnt+1)
          {
           current=first;
           newnode=new node;
           newnode->data=el;
           newnode->next=NULL;
           int i=1;
```

```
            //moving to the right position
            while(i<pos)
             {
            prev=current;
            current=current->next;
            i++;
            }//end of while
            if(prev==NULL)
            {
             newnode->next=first;
             first=newnode;
             end->next=first;
             return;
            } //end of inner if
          newnode->next=current;
          prev->next=newnode;
           }  //end of out if
             else
             cout<<"List size less than position, try again..\n";
           }//end of function insert_beg
           //search function
           int list::search(int el)
           {
              int found=0;
              node *current=first;
              int i=0,cnt;
              cnt=count();
              while(i<=cnt)
               {
                 if(current->data==el)
                 {
                  found=1;
                  break;
                  }
                 current=current->next;
```

```
                i++;
            }
             return found;
          }//end of search function
            //counting nodes in a linked list
            int list::count()
            {
               node *current;
               int c=0;
               current=first;
               while(current->next!=first)
               {
                   c++;
                   current=current->next;
               }
               return c+1;
            }

            //function to display linked list
            void list::display()
            {
               node *current;
               current=first;
               cout<<"Linked list:"<<endl;
               while(current->next!=first)
               {
                  cout<<current->data<<" -> ";
                  current=current->next;
               }
               cout<<current->data<<" -> ";
               cout<<endl;
             }
int main()
{
int el;
```

```
list list1,list2,list3,list4;
clrscr();
//creating the list from beginning calling
list1.create_end(40);
list1.create_end(2);
list1.create_end(100);
list1.create_end(14);
list1.display();
cout<<" inserting 5 at 3 position"<<endl;
list1.insert(5,3);
list1.display();
if(list1.search(20))
cout<<"Found"<<endl;
else
cout<<"Element not in the list"<<endl;
/*cout<<" inserting 34 at 3 position"<<endl;
getch();
return 0;
}
```

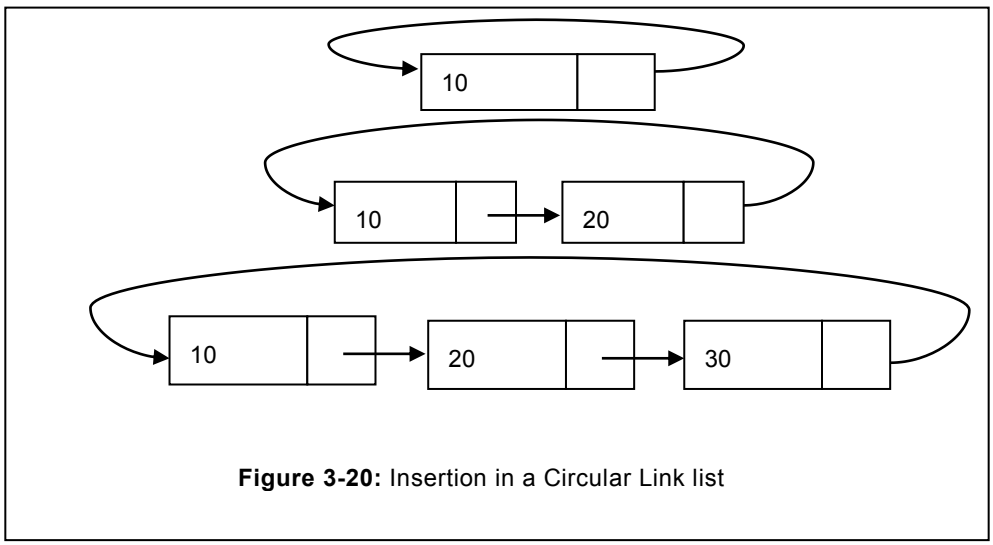


Figure 3-20: Insertion in a Circular Link list

3.13 CASE STUDY BASED ON LINK LIST

A financial institution is engaged in share trading. It serves it customer by offering the share trading application that can be accessed by them using internet. It utilizes the storage device for its data storage. Various information related to the client is stored that includes clients basic details, stock owned by them, value of their portfolio etc. Necessary information, their fields along with the short description has been enumerated herein:

Table 3.1: Client file

S.No	Field	Description
1.	Name	Name of the client
2.	Id	Unique id of the client
3.	Address	Unique address of the client
4.	DoB	Date of birth of a client

Table 3.2: Stocklist

S.No	Field	Description
1.	Stockid	Unique id of the stock
2.	Stock_name	Name of the stock
3.	Market_price	Current market price of the stock
4.	Total share	Total number of share of particular company

Table 3.3: Purchase table

S.No	Field	Description
1.	Stockid	Unique id of the stock
2.	Stock_name	Name of the stock
3.	Market_price	Current market price of the stock
4.	Qty_purchased	Total number of share purchased of a particular company

Table 3.4: Stock sold

S.No	Field	Description
1.	Stockid	Unique id of the stock
2.	Stock_name	Name of the stock
3.	Market_price	Current market price of the stock
4.	Qty_purchased	Total number of share sold of a particular company

User is engaged in stock trading business and need to store the data as described in the above table. He intends to store the record of his client once he opens the account to maintain the purchase or sale of the shares. Based on the above table, create the node of a link list that has the provision for the storage of the above data. Whenever a new client is joining the firm, he should be allotted a new customer id and added in the link list. Whatever the transaction he is performing that should also be linked to the client id. At any point of time, user should be able to view his portfolio that includes the number of share he holds, purchase price, total value of the inventory, etc.

Finally, if he quits the broker's firm, his records need to be deleted from the link list irrespective of his position in the link list (first, middle, or the last). All the deleted records should be stored in the deleted link list. This will enable the firm to track all those customers who had the account with them but no longer associated to them. This will enable the firm to gain insight from the data such as:

- To know the total number of account deleted
- Reason why clients are closing the account.
- To identify the gap in the services offered and the one demanded.
- To meet the regulatory compliances needed for maintaining the client details.

Beyond, the above details, implementer can assume any other information that is not enumerated in the above given case, but it may be useful in implementing the above case.

EXERCISE

Short Answer Type Question

1. What is the need of a link list?

2. What are the various overhead of a link list?

3. Describe the advantage of array over link list.

4. Describe the link list advantages over the array.

5. What are the advantages of having two pointers first and end in the list?

6. How inserting at the beginning is different from inserting at any other position in a link list?

7. Is inserting in between the link list is same as inserting at the end? How it can be managed in the insertion logic?

8. Write a program to swap the first node with the second node, third node with the forth node and so on, till the list is not exhausted.

9. Write a program to swap the first node with the last node, second node with the penultimate node and so on.

10. Write a program to add two list using the + operator (operator overloading).

11. Write a program to traverse the next node by overloading the ++ operator (operator overloading).

12. Write a program to overload the following operator in link list

    = (that checks whether number of nodes are same in two list).

    – (that traverse to one node back in a link list).

    > To check whether the link list one is of more size than the second one.

13. Why deleting the first node is different from deleting the any other node?

14. What are the precautions that need to be taken while deleting the first node?

15. How deleting in between node is different from deleting the last node?

16. What are the points that need to be borne in mind during the creation of an orderly list?

17. What are the major limitations of a link list?

18. Prove the advantage of self-organizing list over ordinary link list? Relate the self-organization method discussed with their computer usage.

19. Why circular link list is used over linear link list?

20. How circular link list is different from the linear link list?

21. What are the drawbacks of circular link list over linear link list?

22. How the display function of circular link list is different from the linear link list?

23. How to delete the last node of the circular link list? Discuss in detail about the pointer adjustment.

24. Describe the necessary changes you will incorporate, while inserting the node at the first position in a circular link list.

25. Consider two link list, list1 and list2. Delete the node of list1 from the position given in list2. For instance, if the list1=(m, n, o, p, q, r, s), list2=(1, 3, 4) then the resultant list will be {n, q, r, s}

26. Write a function that compares whether two given list (list1 and list2) are same or not, while comparing consider the content of the link list.

27. Write down the function to count the number of nodes using recursion.

28. Write a function to create the link list using recursion.

29. Write a function to reverse the link list by using
    - Iterative method
    - With recursion

30. Write a function that moves any given node of the link list to the first position. For instance, if user is giving the node 4, then the 4^{th} node should result as first node.

31. In the given link list, consider one more data member count. Initially, it is 0 for the node that is created. Once this node is accessed (if the given node to be searched or to be displayed etc.) the count should incremented by 1. Adjust the position of the nodes based on the count. For instance, node with the maximum count should be placed at the first position. Node with the second highest count should be placed at the second position. If the count is same then place them as per the following algorithms.

a) First come first serve technique to be used, means the node available at the position should not be changed.

b) Consider the value (data part), if it is less then place the node before the node having greater value.

A. Multiple Choice Questions

In the given questions one or more options may have the right answer. Select the one that best describes the answer.

1. In the list program, the new operator returns which of the following data types:

a) Int type c) void type

b) Node type d) None of these

2. Size of node used in the link list is

a) 02 bytes c) 04 bytes

b) 06 byes d) None of these

3. new is similar to its counterpart malloc function of ‘c’ language

a) True

b) False

4. Circular link list

a) Allows traversal in both the direction

b) Allows traversal only in one direction.

c) No movement is possible.

d) None of these

5. In orderly link list

a) Elements are arranged in ascending order.

b) Elements can be arranged in descending order.

c) Both ‘a’ or ‘b’

d) None of these

6. Complexity of traversing the linear link list in the worst case is:

a) O(1) c) $O(n^2)$

b) O(n) c) None of these

7. Worst case complexity of linear link list and the orderly link list is same:

a) True b) False

8. If in the link list, the searching method is based on binary search technique, then there is:

a) No effect on the performance.

b) Performance will be improved.

c) None of these

9. Reversing the link list using recursion is more effective than non-recursive algorithm:

a) True

b) False.

10. Arrays are faster than link list.

a) True

b) False.

Chapter 4

Doubly link list

Chapter Objective

- Defining the limitation of link list
- Defining the doubly link list and structure of a doubly link list.
- Creation of a doubly link list
- Traversal in a doubly link list
- Insertion in a doubly link list
- Deletion of the node in a doubly link list
- Circular doubly link list
- Display of circular doubly link list
- Inserting a node into a circular doubly link list
- Deletion from a circular doubly link list.

4.1 INTRODUCTION

Doubly link list is a special case of a link list in which each node will consist of two pointers instead of one pointer as the case with the singly link list. Among these two pointers, one point to the address of previous node, while the other one points to the address of next node. Doubly link list is needed to overcome various limitations of a link list that were encountered progressively.

4.2 LIMITATIONS OF LINK LIST

Although the link list is offering amazing result in the cases where we do not need know the number of elements in advance. Yet, it is suffering from the major limitation related to non-traversal in backward direction. Operations can be carried out at great cost in terms of performance. This is attributed to the fact that link list is maintaining only one pointer that holds the address of next node. Consequently, traversal was possible only from start node towards the end node (last node) side only. Even one step backward traversal is needed that also cannot be accomplished.

Limitation discussed above has been addressed by devising the doubly link list. It maintains two pointers instead of one pointer. Single node of a doubly link list comprise of three data members instead of two as in the singly link list. Data members encompassed in a single node are:

- Data part
- Two pointers (one for the next node and the other one for the previous node)

Structure of a doubly link list node is illustrated in figure 4.1. The data part of the link list may be primitive data type or complex or user defined data types.

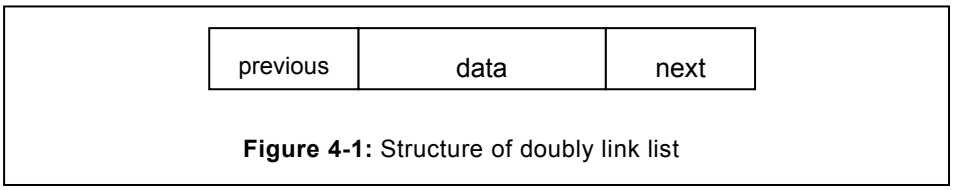


Figure 4-1: Structure of doubly link list

Both the pointers will hold the address of adjacent nodes. One pointing to the previous node and the other to the next node as illustrated in the figure 4.1.

4.3 CREATION OF A DOUBLY LINK LIST

During creation of a doubly link list, first structure of the doubly link list(DLL) node need to be defined. During creation of a DLL node, needed space will be allocated from the heap. It will be followed by connecting both pointers (previous and next) to the NULL, whereas the data part will be filled with the supplied data. Class and other functions needed for the above objectives have been depicted as follows:

```
// structure of a doubly link list
class dnode
 {
public:
int info;
dnode *prev,*next;
};
// class of doubly link list
class dlist
{
dnode *first, *end;
public:
dlist()
{
first=NULL;
}
};
```

During the creation of a doubly link list, we create it by creating one node at a time. Upon defining the node structure, needed space will be allocated from the memory. Allocation of memory to a newnode, connecting the previous and next pointer along with the node data has been depicted as follows:

```
dnode *newnode;
newnode=new dnode;
newnode->prev=NULL;
newnode->next=NULL;
newnode->data=5;
```

Complete operation for creating a node of a doubly link list, its data and allocation assignment has been illustrated in figure 4.2.

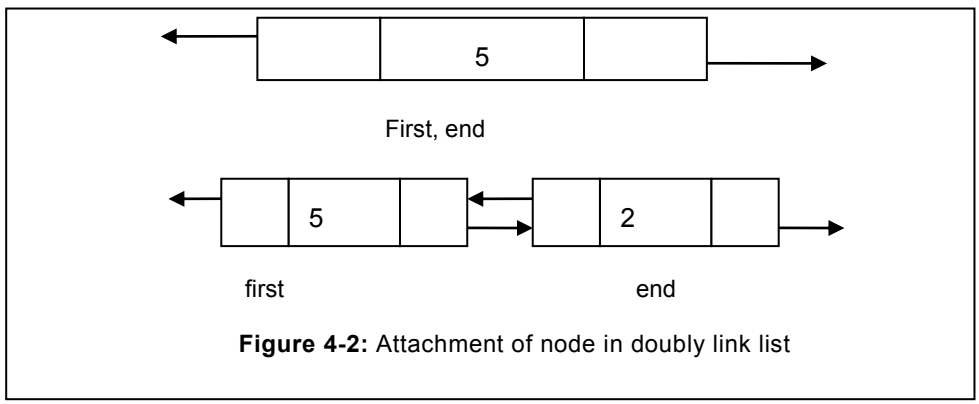


Figure 4-2: Attachment of node in doubly link list

4.4 DISPLAYING ITEMS OF A DOUBLY LINK LIST

In doubly link list, traversal is possible in both the direction. We can start from the first node and using the next pointer can reach upto the last node. Similarly, from the last node, we can reach to the first node by using the previous pointer. We have employed both the method to traverse the doubly link list. Each of them has been enumerated in the following programs.

```
Program: To display the item of a doubly link list
Solution:
//traversing from first to the last node using next pointer
void doubly_display()
{
dnode *current;
current=first;
while(current!=NULL)
{
```

```
cout<<current->data;
current=current->next;
}     // end of while loop
}     // end of function display
```

Similarly, traversal from the last node to the first node can be accomplished by using previous pointer as depicted in the following program:

```
Program: To display the content of the doubly link list
//traversing from last node to the first node using previous pointer
void display_end()
{
dnode *current;
current=end;
while(current!=NULL)
{
cout<<current->data;
current=current->previous;
}     // end of while
}     // end of function
```

```
//class for the node of a doubly link list
class dnode
{
public:
       int data;
       dnode *prev,*next;
};

class dlist
{
       int n;
       dnode *first,*last;
       public:
            //constructor to initialized the data member
            dlist();
```

```
            //To create the node in doubly link list
            void create(int);
            //To count the number of node
            int count();
            //To search element in doubly link list
            int search(int);
            //To insert the element at particular position
            void insert(int, int);
            //To reverse the doubly link list
            void reverse();
            //To concatenate two doubly link list
            dlist operator+(dlist);
            //To display content of doubly link list
            void display();
                  void display1();        //To display content from
last to first
};
```

Definition of the functions declared in the above class has been given in the following program.

```
// defining constructor for the doubly link list
dlist::dlist()
{
    first=NULL;
    last=NULL;
}
            //function to create the node of a doubly link list
            void dlist::create(int item) // creation of link list,
front to end
            {
                  dnode *current,*temp;
                  if(first==NULL) // if no node create the first node
                  {
                  first=new dnode;
                  first->data=item;
                  first->next=NULL;
```

```
                first->prev=NULL;
                last=first;
                }
                else
                {
                temp=new dnode;
                temp->data=item;
                temp->next=NULL;
                temp->prev=last;
                last->next=temp;
                last=temp;
                }
          }
         //function to display the element of doubly link list
         //Display from front to last using next pointer.
         void dlist::display()
               {
            dnode *current;
            current=first;
            cout<<"Data in doubly linked list(front to last):\n";
            while(current!=NULL)
            {
                  cout<<current->data<<" <-> ";
                  current=current->next;
            }
            cout<<endl;
         }
      // function to display the node element from last to first
   void dlist::display1()
         {
            dnode *current;
            current=last;
            cout<<"Data in doubly linked list from end to
            first:\n";
            while(current!=NULL)
```

```
        {
                cout<<current->data<<" <-> ";
                current=current->prev;
        }
        cout<<endl;
    }
```

4.5 INSERTING ITEM (NODE) IN A DOUBLY LINK LIST

In a doubly link list, a new node can be inserted at any position including first, middle or at last position. In a doubly link list insertion of the node is less cumbersome relative to the singly link list. Complexity is reduced due to the availability of two pointers that exist in the doubly link list. During insertion of a node in a doubly link list, following sequence of steps need to be followed:

- Accept the position where new node is to be inserted.
- Intending position where node is to be inserted should be less than or equal to length of the doubly link list, if it is more than total number of nodes, in that case node cannot be inserted.
- If the above condition is fulfilled, then traverse upto the node that is just before which the node is to be inserted. Assume that this node is termed as previous node.
- Find out the next pointer of the current node.
- Connect the new node with the forward node and the node lies before this current node i.e. previous node.
- If the node is to be inserted at the first position in that case the pointer of the first node should be changed to this new node.
- Similarly, if the node to be inserted is the last node, change the pointer of the last node to this new node.

All the above cases have been discussed in forthcoming sub-sections.

A. Insertion at the beginning

In a doubly link list, node can be inserted at the beginning of a link list but in that case the first pointer will be shifted to this new node that is attached at the first position. Complete process has been defined as follows:

- Create the new node.
- Connect its next pointer to the already existing first pointer.
- Change the exisiting first node’s pointer to the pointer of the node that has been just inserted.

Once it is connected to the first node, then correspondingly first pointer is moved to the newnode as this node has become the first node. Same has been illustrated with the help of an example. In the given example, node with value 3 is to be attached at the first position. After the attachment of this new node, it will become the first node.

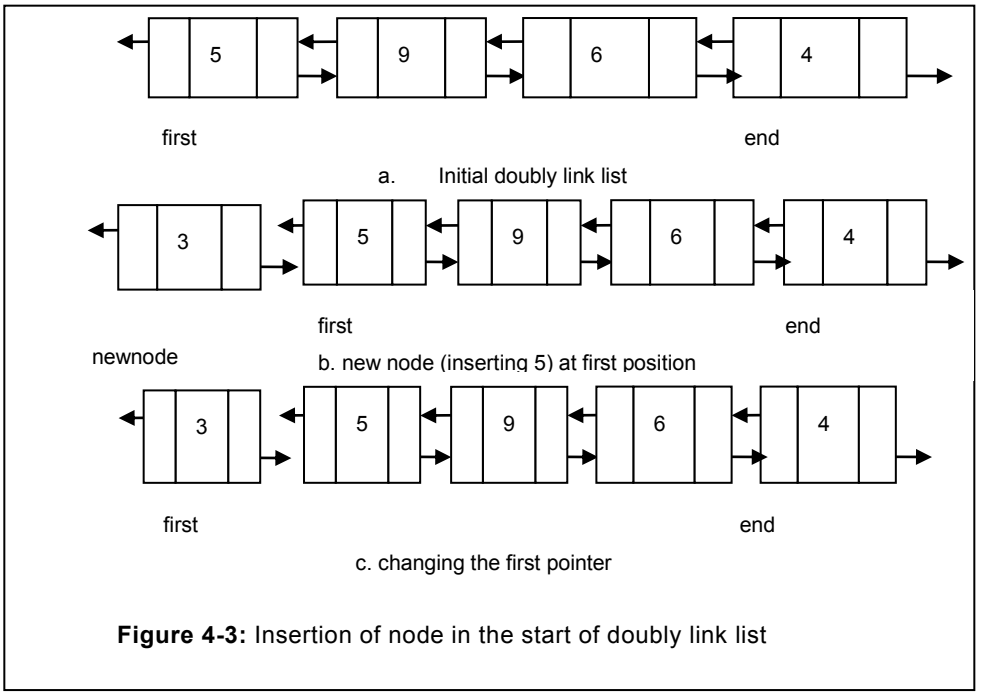


Figure 4-3: Insertion of node in the start of doubly link list

B. Insertion other than the first position

In this case, we have assumed that the node is to be inserted other than the first position in a doubly link list. It means that the position any value ranging from the second onwards including last. Whenever, we will insert the node at any other position after first node i.e. in the middle position or at the last position of the given link list, we need to create the space for positioning the new node and attaching this new node with the previous and the forward node. We will follow the following steps:

- Create a new node.
- Traverse to the position, where the node is to be inserted.
- Connect the next pointer of the newnode node to the forward node.
- Connect the previous pointer of the newnode to the current node.
- Finally, connect the next pointer of the reached node to this newnode.

Step by step, the entire procedure has been illustrated in the following figures.

By meticulous implementation of the logic, second and third conditions (insertion in between and last) can be merged into one. We have implemented the program considering all the insertion positions. Complete implementation has been depicted in the following program:

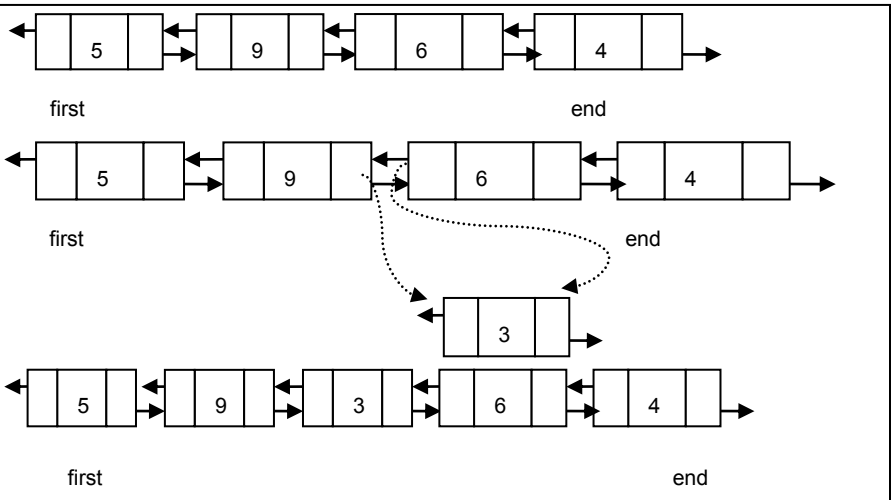


Figure 4-4: Insertion of node at any position of doubly link list

```
//function to insert the node at a specific location. This function,
takes two arguments, one is the position where the node is to be
inserted and the second is the elements value that need to be
inserted in a doubly link list.
void dlist::insert(int pos, int el)
           {
      int b=count();    //finding out the number of nodes
      cout<<"Length of list :"<<b<<endl;
      if(pos<=b+1) //checking insert location is less than length+1
        {
         cout<<"Inserting "<<el<<"  at position"<<pos<<endl;
         node *current,*forward,*temp; //pointer for first and next
node
         current=first;
         temp=new dnode;
         temp->data=el;
         temp->next=temp->prev=NULL;
            if(pos==1)  //checking the first position
```

```
            {
             temp->next=first;
              first->prev=temp;
               first=temp;
               }
       // if node to be inserted at any other location
             else if(pos<=b)
        {
        for(int i=1;i<pos-1;i++)
        current=current->next;
        forward=current->next;
        temp->next=forward;
        temp->prev=current;
        current->next=temp;
        forward->prev=temp;
        }
        else
        {
        last->next=temp;
        temp->prev=last;
        temp->next=NULL;
        last=temp;
        }
        cout<<"Node inserted at position "<<pos<<endl;
       }
       else
       cout<<"Can't be inserted\n";
}
```

4.6 DELETING THE ITEM (NODE) FROM DOUBLY LINK LIST

To delete a node from a doubly link list, various cases have been taken into account to achieve completeness. Various cases of deleting a node from the doubly link list includes the following:

- Deletion from the first position
- Deletion in any other location (other than the first)

4.6.1 Deleting the first node

To delete the first node, procedure to be employed is different from the one at any other node. Due to the deletion of the first node, the pointer holding the first node address will be lost. Consequently, first position needs to be shifted to the new first position before deletion. Required changes will be reflected in the existing doubly link list. Deleting a node from the doubly link list can be carried out by adhering to the steps mentioned below:

- Find out the node location that needs to be deleted. It should be less than or equal to total number of nodes in a doubly link list (number of node in a doubly link list).
- If the node to be deleted is the first node, save the first node in the temp pointer. Move the first node pointer one step ahead of the current position.
- Delete the first node that is stored in the temp node.

Since the node to be deleted is the first node, in this case, the first node is to be stored in the temp node. Move the first node to the second position of the existing list. However, previous node is still pointing to the first node. To remove this node from the list, store NULL to the previous of the new first node. Finally, remove the original first node of the link list.

To determine if the node is first node, we examine the previous pointer of the node. If the first pointer is NULL then it implies that the node to be deleted is the first node, therefore it is to be handled accordingly.

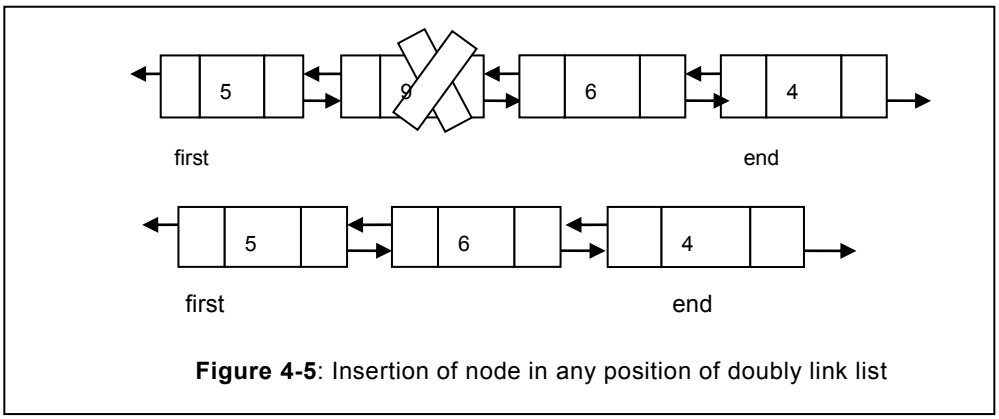


Figure 4-5: Insertion of node in any position of doubly link list

Deleting the node other than the first node has been discussed in the forthcoming sub-section.

4.6.2 Deleting the node other than the first node

If the node to be deleted is other than the first node (including the last node), in that case the node to be deleted is to be stored in the temp node. Afterwards, we can connect the previous node to the next of the current node. Finally, delete the temp node. Deletion of temp node will result in release of memory occupied by it. Complete function to accomplish this task has been depicted in the following program.

```
//function to delete node from doubly link list
void dlist::deletenode(int el)
{
//int found=0;
dnode *current,*temp;
current=first;
while(current!=NULL)
   {
   if(current->data==el)
     {
     //Node to be deleted is the last node
     if(current->next==NULL)
     {
     last=current->prev;
     current->prev->next=current->next;
     delete current;
     break;
     }
     else if(current->prev==NULL)// node is first node
       {
       current->next->prev=current->prev;
       first=current->next;
       delete current;
       break;
       }
       else // any other node to be deleted
       {
       current->prev->next=current->next;
     current->next->prev=current->prev;
     delete current;
     break;
     }
     }
   current=current->next;// otherwise continue to next node
   }
```

4.7 SEARCHING ELEMENT IN A DOUBLY LINK LIST

Doubly link facilitates the traversal in both the direction. For instance, we can traverse from front towards the end as well as from end towards first node. During the traversal, a node consisting of specific value can be located by matching the value held by the node. Due care is to be observed to ensure that traversal should be limited only to the number of nodes existing in the list. As soon as node consisting of the element encountered, searching should be terminated. The whole procedure is depicted in the following program.

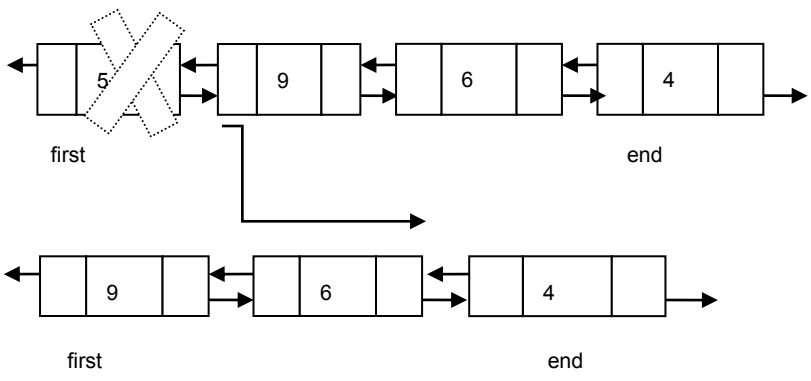


Figure 4-6: Deletion of the first node in a doubly link list

```
//function to search an element from a doubly link list
int dlist::search(int element)
           {
              int flag=0;
              dnode *current;
              int b=count();
              current=first;
              for(int i=1;i<=b;i++)
              {
                    if(current->data==element)
                    {
                    flag=1;
                    break;
                    }
                    current=current->next;
```

```
        }
    return flag;
    }
```

In the program depicted above, we have declared a variable flag to represent the status of item to be searched. We have initialized it to '0' to represent that the item is not present in the list. If item to be searched is found in the doubly link list, immediately loop is terminated with the help of break statement, and status of flag is set to the new value of 1 to represent that item is located within the link list. Finally, the status of the flag is returned to the caller.

4.8 COUNTING NUMBER OF NODES IN A DOUBLY LINK LIST

Counting the number of nodes in a doubly link is needed on many occasion for instance to learn about the length of doubly link list, during insertion of a node, during deletion of a node etc. Therefore, counting the number of node is the significant function in a doubly link list. Counting all the nodes of a doubly link list does not involve much complexity. Here the user has to start from the first node and go upto the last node. Counting the node is needed during traversal of nodes. Since, doubly link list facilitates traversal in both the direction; therefore we can also count the number of node by starting traversal from the end. During this traversal, we can use the previous pointer to reach upto first node, at the same time we can count the number of nodes in the doubly link list. However, in our example, we have considered traversal from the beginning towards the end node. Same has been depicted in the following program.

```
//Program to count the number of node in a doubly link list
Solution:
int count( )
{
dnode *current=first;
int cnt=0;
while(current!=NULL) // till the double link list is not exhausted
{
cnt++;            //count the node
current=current->next;  // move to the next node
}
return cnt;
}
```

4.9 CIRCULAR DOUBLY LINK LIST

Even though doubly link list permits the traversal in both the direction yet sometime we need to connect the last node to the first node. Such type of doubly link list is known as circular

doubly link list. Doubly link list has the following advantages.

- Does not have any null pointer.
- Improves the efficiency, specifically if required to be traversed from the last node towards the first node.
- Improves the performance, if required to be traversed from first node towards the last node side.

Circular doubly link list has been illustrated in the figure 4.7.

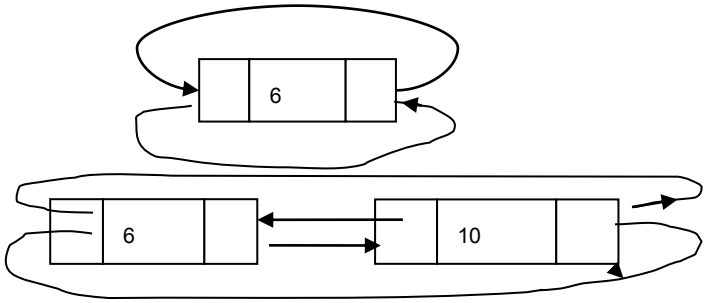


Figure 4-7: Circular doubly link list

4.9.1 Creation of circular Link list

Creation of a circular doubly link list is different from its counterpart doubly link list. Once the first node is created, unlike the doubly link list, we do not point to the previous and next pointer to the NULL. Instead, we connect them to the node itself. Same has been illustrated in the figure 4.7 (a). Once the second node is created, its previous node is connected to the first node. A pointer of the first node is also to be adjusted, because the next pointer of the first (existing) node will point to this newnode and the next pointer of this new node to be pointed to the first node. Functions such as insertions, removal, traversal can be perform similar to that of normal link list. Class that performs various activities (function) on circular doubly link list has been depicted in the following program.

```
#include<iostream.h>
#include<conio.h>
// class to define the structure of a node in a doubly link list
class dnode
{
public:
int data;
```

```
dnode *prev;
dnode *next;
};
// Class for the doubly link list
class dblyList
{
dnode *first;
dnode *end;
public:
dblyList()
{
first=NULL;
end=NULL;
}
void create(int);
void display();
void insert(int, int);
int count();
void deletenode(int);
};
```

4.9.2 Function to create circular doubly link list

In the creation of the circular doubly link list, we have assumed two pointers namely first and end. These two pointers represent the first and the last node of the circular doubly link list. By using the last's next, we can reach to the first node of the circular doubly link list. Correspondingly, we can also reach to last node by using the first's previous. When the first node is created and the doubly link list is having only one node, in that case both the first and the last node will point to the same node. To insert the other node, pointer of last and previous pointer of the first node need to be connected to the newly created newnode. Complete program has been depicted as follows.

```
Prob: Write a program to create first node or any other node
// To create the first node
void create(int element)
{
if(first==NULL)
 {
```

```
 first=new dnode;
 first->data=element;
 first->next=first;
first->prev=first;
end=first;
return;
} // end of if statement
dnode *newnode; //to create any other node
newnode=new dnode;
newnode->data=element;
newnode->next=end->next;
newnode->prev=end;
end->next=newnode;
first->prev=newnode;
end=newnode;
}
```

After inserting the node into the circular doubly link list, we need to view the data available in a circular doubly link list. This objective can be accomplished with the help of display function. In the display function, due diligence is to be exercised in order to avoid infinite looping. Since, in the circular doubly link list, last node is connected to the first node, therefore, loop must terminate at the 'end' node otherwise, it will lead to infinite loop.

```
void dblyList::display()
 {
 dnode *current=first;
 while(current->next!=first)
  {
  cout<<current->data<<"->";
  current=current->next;
  }
  //to display the data of last node
cout<<current->data<<"->";
}
```

4.9.3 To count the number of nodes in a circular doubly link list

To count the number of nodes in a circular doubly link list, we need to begin traversal either from the front or from the back. In a circular doubly link list since first is connected to the last

node and the last node is pointing to the first node, correspondingly, due care need to be observed in order to prevent the cycling during the traversal. In our case, we have applied the condition thereby it is examined that we have not reached to the node from the one we have began. Function for this effect has been depicted as follows:

```
// To count the number of node in a doubly link list
  int dblyList::count()
  {
// initialize the count to 0
 int c=0;
// store the address of first to the current node
dnode *current=first;
// Run the loop till we are not reaching to the first node
while(current->next!=first)
  {
  c++;      // increment the count
// move the list to the next node.
 current=current->next;
}
  //One node was left due to current->next, count that node also;
  c=c+1;
  return c;
}
```

The variable ‘c’ represents the number of count, initially this value will be zero, correspondingly, it has been initialized with the zero. During visit to each node, we increment the values of this variable by one. Since, we have terminated at the last node, therefore this node has not been reached. To count this node, we have incremented the current value of variable ‘c’ by 1. Finally, value of ‘c’ has been returned to the caller.

4.9.4 To insert a node at a given position

During inserting a node in a circular link list, we need to ensure that circular property is not lost upon insertion. This is particularly applicable during the insertion of a node at either the first position or at last position. When the node to be inserted is in between, then there is no difference between the circular doubly link list and the doubly link list. Complete procedure of inserting the node in a circular link list has been depicted as follows:

- Figure out the validity of position where new node is to be inserted(must lie between first and last node plus one).
- If the position where node to be inserted is first, connect the previous node of the

newnode to the previous node of the existing first node. Connect the next pointer to the first node. Name this newnode as first node by shifting the first pointer to this newnode. Similarly change the next pointer of the last node to this newnode(first node).

- To insert the node in-between (other than the first and the last node), replace the next pointer of newnode with the next pointer of current node(node after which the newnode is to be inserted). Connect the previous node of the newnode with the current node. Connect the next pointer of the current node to the newnode.
- To insert at the end position, replace the next pointer of this newnode with the existing next pointer of the end node (it is pointing to the first node). Connect the next pointer of the existing end node to the newnode (this will act as last node), connect the previous node of newnode to the existing last node. Finally, move the end node to this newnode.

Program to demonstrate the entire operation has been depicted herein:

```
 // To insert the node at a given position
void dblyList:: insert(int element, int pos)
    {
    int n=count();
     dnode *current,*temp;
    temp=new dnode;
    temp->data=element;
    if(pos<=(n+1))
    {
        if(pos==(n+1))
     {      // insert at the end
     temp->prev=end;
     temp->next=end->next;
    end->next=temp;
    first->prev=temp;
    end=temp;
    }
    else
      {
       current=first;
      int i=1;
      while(i<pos)
      {
```

```
      current=current->next;
            i++;
      }
      temp->next=current;
      temp->prev=current->prev;
      current->prev->next=temp;
      current->prev=temp;
      //if the position is 1, change the first pointer
      if(pos==1)
      first=temp;
     }
    }
    else
cout<<" Cannot be inserted beyond the list length"<<endl;
}
```

4.9.5 Delete a node from a circular link list

Deletion of a node from a circular doubly link list is substantially different from the doubly link list. Due care needs to be observed while deleting the first or the last node of the circular doubly link list. If the first node is to be deleted, in that case, first is to be moved to the one node ahead of the existing first node. Despite of deletion of the first node, circular property of the link list is to be retained. Therefore, this new first has to be connected to the end node. Correspondingly, the last node of the circular doubly link list need to be adjusted. Pointer of the last node i.e. the End node, need to be connected to the first node of the doubly link list. Same has been depicted in the following program.

```
//To delete the node based on a  user supplied value
    void dblyList:: deletenode(int element)
     {
     dnode *current,*prev1=NULL,*temp;
     current=first;
     do{
      if( current->data==element)
      {
      break;
      }
      prev1=current;
```

```
     current=current->next;
     }while(current!=first);
// if node to be deleted is first node
  if(prev1==NULL)
       {
       temp=first;
       current=first->next;
       //connecting the new first to the last node
        first->next->prev=first->prev;
       end->next=current;
       first=current;
      }
// if node to be deleted is the last node
else if(current==end)
      {
      temp=current;
//Connecting the first node to the last node
     first->prev=prev1;
     // connecting the last node to the first node
     prev1->next=first;
       end=prev1;
      }
// if the node to be deleted is the in-between node
      else
      {
      temp=current;
      prev1->next=current->next;
      current->next->prev=prev1;
      }
     delete temp;
     }
```

4.10 DOUBLY LINK LIST USING TEMPLATE

Doubly link that has been discussed in length, however, we have assumed data to be inserteds of integer data type. If we need to implement it for any other data type(s) then

program need to be modified accordingly. To address these situations we suggest implementation of template doubly link list. This template link list will get modified according to the type to be inserted. List can be implemented similar to link list implementation using pointers. Changes are needed to be made only due to the template formation only. Complete code has been depicted as follows for the convenient of the readers.

```
//operations on double ended linked list
#include<iostream.h>
#include<conio.h>
 template<class t>
class dnode
{
      public:
       t data;
       dnode *prev,*next;
};
template<class t>
class  dlist
{
      dnode<t> *first,*last;

       public:
            //constructor to initialize the data member
            dlist();
            //to create the doubly link list
            void create(t);
            //to count the number of node
            int count();
            //to search element in doubly link list
            int search(t);
            //to insert the element at particular position
            void insert(int, t);
            //to reverse the doubly link list
            void reverse();
            //to concatenate the doubly link list
            dlist operator+(dlist);
```

```
            //to display content of node
            void display();
            //to display content last to first
            void display1();
            void deletenode(t);
};
//constructor to initialize the data member
template<class t>
dlist<t>::dlist()
{
    first=null;
    last=null;
}

//function to delete node from doubly link list
 template<class t>
void dlist<t>::deletenode(t t1)
{
//int found=0;
dnode<t> *current,*temp;
current=first;
while(current!=null)
   {
   if(current->data==t1)
     {
     //node to be deleted is the last node
     if(current->next==null)
     {
     last=current->prev;
     current->prev->next=current->next;
     delete current;
     break;
     }
     else if(current->prev==null)// node is first node
       {
```

```
       current->next->prev=current->prev;
       first=current->next;
       delete current;
       break;
       }
       else
       {
       // any other node to be deleted
     current->prev->next=current->next;
     current->next->prev=current->prev;
     delete current;
     break;
     }
     }
   current=current->next;
   }
}
            //function to create the doubly link list
            template<class t>
            void dlist<t>::create(t item)
            {
                 dnode<t> *current,*temp;
                 if(first==null)
                 {
                 first=new dnode<t>;
                 first->data=item;
                 first->next=null;
                 first->prev=null;
                 last=first;
                 }
                 else
                 {
                 temp=new dnode<t>;
                 temp->data=item;
                 temp->next=null;
```

```
                temp->prev=last;
                last->next=temp;
                last=temp;
                }
         }
        //function to display the element of doubly link list
        template<class t>
        void dlist<t>::display()
        {
           dnode<t> *current;
           current=first;
           cout<<"data in doubly linked list(front to last):\n";
           while(current!=null)
           {
                 cout<<current->data<<" <-> ";
                 current=current->next;
           }
           cout<<endl;
        }
    // To display all the element of list
        template<class t>
        void dlist<t>::display1()
        {
           dnode<t> *current;
           current=last;
           cout<<"Doubly link list end to  first:\n";
           while(current!=null)
           {
                 cout<<current->data<<" <-> ";
                 current=current->prev;
           }
           cout<<endl;
        }
        //reverse function
        template<class t>
```

```
void dlist<t>::reverse()
{
int n;
   n=count();
   dnode<t> *current;
   current=last;
   cout<<"the data after reversing the linked list:\n";
   for(int i=1;i<=n;i++)
   {
         cout<<current->data<<" -> ";
         current=current->prev;
   }
}
//count function
template<class t>
int dlist<t>::count()
{
   int c=0;
   dnode<t> *current;
   current=first;
   while(current!=null)
   {
      c++;
      current=current->next;
   }
   return c;
}
//to insert el(element) at particular position
template<class t>
void dlist<t>::insert(int pos, t el)
{
   int b=count();
   cout<<"length of list :"<<b<<endl;
   if(pos<=b+1)
   {
```

```
            cout<<"inserting "<<el<<"  at position"<<pos<<endl;
                      dnode<t> *current,*forward,*temp;
                      current=first;
                      temp=new dnode<t>;
                      temp->data=el;
                      temp->next=temp->prev=null;
                      if(pos==1)
                      {
                            //temp->prev=null;
                            temp->next=first;
                            first->prev=temp;
                            first=temp;
                      }
                      else if(pos<=b)
                      {
                            for(int i=1;i<pos-1;i++)
                            current=current->next;
                            forward=current->next;
                            temp->next=forward;
                            temp->prev=current;
                            current->next=temp;
                            forward->prev=temp;
                           // forward->prev=temp;
                      }
                      else
                      {
                            last->next=temp;
                            temp->prev=last;
                            temp->next=null;
                            last=temp;
                      }
                     cout<<"node inserted at position "<<pos<<endl;
              }
              else
              cout<<"can't be inserted\n";
```

```
            }
            //search function
            template<class t>
            int dlist<t>::search(t element)
            {
               int flag=0;
               dnode<t> *current;
               int b=count();
               current=first;
               cout<<"searching element "<<element<<endl;
               for(int i=1;i<=b;i++)
               {
                     if(current->data==element)
                     {
                     flag=1;
                     break;
                     }
                     current=current->next;
               }
             return flag;
             }
            //overloading + operator
            template<class t>
            dlist<t> dlist<t>::operator +(dlist<t> l)
            {
               dlist l6;
               l6.first=first;
               l6.last=last;
               l6.last->next=l.first;
               l.first->prev=l6.last;
               return l6;
            }
main()
{
    int n,el;
```

```
    dlist<int> l1,l3,l2;// creating the doubly list for integer type
    clrscr();
    l1.create(34);
    l1.create(7);
    l1.create(153);
    l1.create(40);
    l1.display();
    //l1.insert(4,112);
  l1.display();
  if(l1.search(40))
  cout<<"element found"<<endl;
  else
  cout<<"element not in doubly link list"<<endl;
  cout<<"enter element to be deleted"<<endl;
  cin>>el;
  l1.deletenode(el);
  l1.display();
       }
```

4.11 CASE STUDY BASED ON DOUBLY LINK LIST

Refer the case already enumerated in the link list chapter and implement the same using doubly link list. If the same program is implemented using doubly link list, then what will be the effect on performance. Extend the doubly link program enumerated in the link list example by including more data members. These members should be relevant to the student details. These members have been depicted as follows:

Table 4.1: Student description

S.No	*Data members*	*Description*
1.	*Stud_id*	*Unique id of all the students.*
2.	*Student_name*	*Name of a student*
3.	*Subject*	*Name of the subject(each student will have 5 subjects)*
4.	*Marks*	*Marks of the subject (for all the above 05 subjects, there will be there corresponding marks).*
5.	*Total*	*Total of marks in all the above subjects*

- Each node should contain the above information. Consider the case that the total numbers of students are 20. Keep the data in the doubly link list itself. Write a function that should organize the records (nodes) of the doubly link list based on total marks. For instance, student secured the maximum marks to be put at first position, then the second highest, etc. Write a function, to search the node (record) in the given doubly link list.
- Extend the above program by including one more data member in the doubly link list. Name of this data member should be count. Based on the number of time the record (node) is searched, the value of count should increment by 1. Rearrange the nodes based on the count. For instance, if the 4th node has the maximum count then it is to be positioned at the first place.

EXERCISE

A. Descriptive Questions

1. Write a function that inserts the new node after the last node in a doubly link list without traversing the doubly link list.
2. Modify the above function, so that there is no need to check whether the first node is NULL or not.
3. Write a function that inserts a new node in the doubly link list. Function should manage the following:

a. Whether the list is existing or not.

b. Whether the position given is within the permissible range or not. For instance, if the total numbers of nodes are 5 and the given position where node to be inserted is 8 then it should not be allowed.

4. Write a function that deletes a node in the doubly link based on the following factors:

a. Based on existing list.

b. Consider the position given is within the permissible range or not. For instance, if the total numbers of nodes are 5 and the given position where node is to be deleted is 8 then it should not be allowed.

c. Delete the node based on the value received from the user.

5. Write a function to delete a node from a doubly link list.
6. To delete the first node of doubly link list.
7. To delete the node that is in between (other than first and last).
8. To delete the node existing at the end.
9. Once the node is deleted adjust the pointers accordingly. For instance, if the last node is deleted in that case the previous node has to be pointed to the NULL, since now penultimate node is acting as last node.

B. Multiple Choice Questions

1. What will be the size of the class used to define the structure of the doubly link list.

a. 02 bytes c. 40 bytes.

b. 06 bytes d. 12 bytes.

2. Can we modify the existing create function by including the pointer as an argument?

a. Yes

b. No.

(Write down the points in support and against of your statement and get it verified).

3. If the current is pointing to the current node of a doubly link list. To access the two nodes ahead of current node, following statement can be used.

a. current->next c. current->next->next

b. current->next->prev d. None of these

4. If the current is pointing to the current node and forward is pointing to one node ahead of current. Then the code forward->prev points to the

a. Current node. b. One node ahead of the forward node.

c. One node previous to the current node.

d. None of these

5. In the above question, to delete the node that lies between the first node and the last node of a doubly link list, the code current->next, deletes forward node:

a. True

b. False

Chapter 5

Stack

Chapter Objective

- Defining stack
- Defining various operations permitted on the stack
- Describing the push and pop function
- Describing the nop and returning top of the stack
- Displaying the items in a stack
- Describing the application of stack
- Creating the dynamic stack
- Creating the stack using template.

5.1 INTRODUCTION

In our daily life, during dinner time once we pick the plate(s), it is the top plate. Whenever caterers need to place more plates to replenish the existing plates, same is placed on the top of the existing plates. Same is the case with our new data structure that is popularly known as stack. It is widely used data structure in plenty of applications including tree traversal, balancing of the tree, expression computation, parenthesis mathing to name a few.

5.2 STACK DEFINITION

In line with the above example, we can define stack as a data structure considered to be opened from one end only. This open end is known as top of the stack. Insertion and deletion operations are permitted from this 'top' end only. Items are inserted one after another, where last element inserted remains at the top position of a stack. Consequently, the item inserted at last is removed first. Process in which we insert the item last is removed first is known as Last in First out (LIFO) order. Correspondingly, it can be concluded that stack works on LIFO order.

The complete process of inserting item(s) in a stack has been illustrated with the help of figure 5.1. Initially, we have assumed that the stack is empty, threrefore, top of the stack is initialized with -1, to represent the empty condition. In first step, on arrival of element 'A' the top moves to the 0^{th} index and item is placed at this position, as illustrated in the figure 5.1 (a). Similarly, other items 'B' and 'C' are also inserted into the stack, correspondingly, top also moves upwards, refer figure 5.1 (b, c). Before insertion of item in a stack, top is always examined that it has not reached to the capacity of the stack. Irrespective of operations

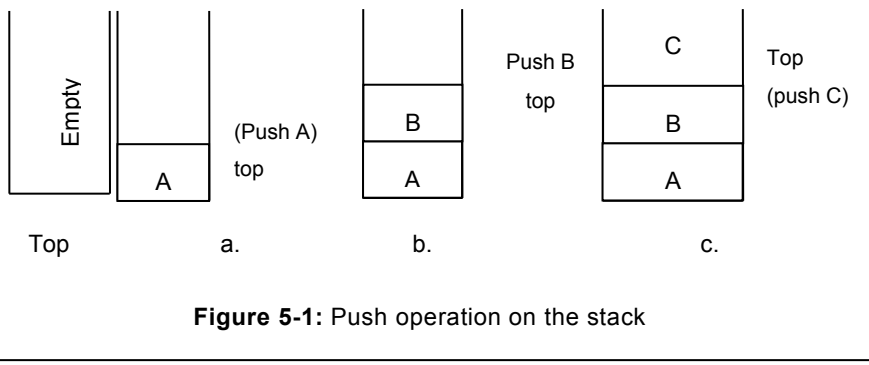


Figure 5-1: Push operation on the stack

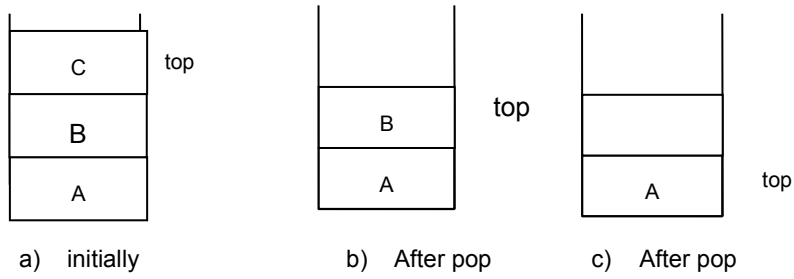


Figure 5-2: Pop operation

(push or pop) performed on the stack, top will always point to the last element of a stack.

5.3 OPERATIONS ON STACK

In a stack, two major operations are carried out and related to the insertion and removal of the items of a stack.

Push: Inserting the element(s) into the stack is known as push. Once the item is pushed into the stack, top is incremented by one, therefore top moves towards the capacity of the stack. Once the capacity is full and still we intend to insert more item(s) on the stack then this event is known as overflow. Since, overflow will lead to data loss, therefore provision is to be made to avoid the overflow. Push operation has been illustrated in the figure 5.1.

Pop: Removing items from the stack is known as pop operation. In the pop operation, items are removed from the top, correspondingly, top moves towards the base of the stack. Once the stack is empty and still we attempt to remove the elements from the stack, it is known as underflow. Pop has been illustrated in the figure 5.2.

Consider the case where we have already three elements in the stack, therefore stack will point to the top position. In the figure 5.2, the top element of the stack is 'C', therefore, top is at index 2. Once the one item is popped, then top will get decremented by one index and will point to 'B' (refer figure 5.2 b)

Nop: Beyond the push and pop operation, there is another possible operation that can be carried out on the stack, it is known as NoOperation(Nop). In Nop, there is no operation that takes place. In other terms, there is no movement of the top; instead, activities are carried out without the movement of the top. For instance, displaying the elements of the stack falls under this category, since it does not involve any movement of the top of the stack. The other example related to the Nop is to determine the top of the stack element, in which top element is to be returned without moving the top pointer.

5.4 IMPLEMENTATIONS OF STACK

Stack is widely used data structure in computer science, usage of stack improves the overall performance, at the same time simplifies the code. In stack, many times we know in advance the maximum number of elements that need to be inserted. However, many times the number of items to be inserted cannot be predicted accurately. Correspondingly, to cater these needs, stacks are categorized into:

- Static stack
- Dynamic stack

In stack, if total number of items that can be inserted is predicted in advance, then such type of stack can be implemented with the help of an array and termed as **static stack**. Such stacks give better performance relative to any other type of stack.

In cases, if the total number of items to be inserted is not known or variation of minimum number of items to be inserted to the maximum number of items to be inserted is huge, in such cases, we use another type of stack known as **dynamic stack**. But this stack involves the performance overhead related to memory allocation and de-allocation.

5.4.1 Static Stack

Static stack is implemented with the help of an array. Before creating the stack, its structure for the type of information that it will hold is to be defined. The other constraint is related to the top pointer that will point to the top of the stack. Class denoting the data member and the operations that can be accomplished on the data member has been given as follows:

```
Program for the push, pop and display function of the static stack.
// Program that depicts push and pop operation on the stack
const int size=5; // size of the stack
// Defining the structure of the stack
class stk
{
```

```
int stack[size];
int top;
public:
// constructor to initialize the value of top
stk()
{
top=-1;
}
// To push the element in the stack
int push(int);
//To pop the element from the stack and return the top/pop element
int pop();
// To display the items in the stack.
void display();
// To display the number of element in the stack.
void count_elements();
// To examine whether the stack is empty or not
int isEmpty();
};
```

Initially, the stack is empty and it is denoted by assigning the value of -1 to top. Whenever an item is inserted into the stack, it is examined for its capacity (represented by size) to avoid overflow. If the stack is not full then only item(s) is allowed to be inserted into the stack. Correspondingly, top pointer is incremented by +1 to denote the new position of the top.

Push function returns the integer value. This value is utilized to determine whether the push operation was successful or unsuccessful. If it is successful, push function returns 1, whereas, in case of failure, it returns the value 0.

```
// function to implement the push operation of stack.
int stacks::push(int element)
{
        if(top==(size-1))
        {
      cout<<"overflow, more elements cannot be inserted \n";
      return 0;
       }
       else
```

```
        {
        stack [++top] =element;
        }
      return 1;
}
```

To pop the element from the stack, first we would check for the empty condition. If the stack is empty then underflow message will be flashed to the user. To check the empty condition, we check the value associated to the top variable.

if (top== -1)

Value of -1 represents the empty condition. If there are more than zero elements in the stack, in that case pop operation can be performed. Before pop operation, the top element of the stack is stored in the temp variable. Afterwards, the top moves one pointer down and finally the value of the top stored in the temp variable is returned to the caller function, for the later usage, if any.

```
//function to pop the element from the stack
int stacks::pop ()
           {
              int temp=0;
             // Examine whether stack is empty or not.
                   if (top==-1)  // top=-1 indicates the empty state
of stack
              {
                        cout<<"underflow: \n";
            }
              else
              {
                  temp=stack [top--];
              }
             return temp;
           }
```

We have also designed the function isEmpty () to determine whether the stack is empty or not. isEmpty() function can be used in several places, for instance, to determine the number of elements in a stack, at any point in time. This function can be utilized before the pop operation is performed. If the stack is empty then pop should not be permitted, instead suitable message need to be flashed to the user. It returns 1 that represents true, if the stack is empty otherwise returns 0 that represents false. Same has been illustrated in following program.

```
// To examine whether the stack is empty.
int stacks::isEmpty ()
           {
              if (top==-1)
              return 1;
              else
              return 0;
           }
```

Despite of executing the push and pop operation, we are not aware of the elements that are available in the stack. To accomplish this objective, the function display () has been created. This function displays all the elements in the stack. It starts from the top and goes towards the bottom of the stack. In order to ensure that top is not shifted from the current position, a new variable with the name 'i' has been used and the value of top is assigned to this variable.

```
// Program to display the element(s) of the stack.
void stacks::display ()
           {
              for (int i=top; i>=0; i--)
              cout<<stack[i]<<" ";
              cout<<endl;
           }
```

On various occasions, we would need to determine the number of elements in the stack at any particular point in time. To accomplish this objective, we have created the count_elements () function.

5.4.1.1 Counting the number of elements

To count the number of elements in a stack, we have started from the top and moved towards the base. During this looping, variable cnt that counts the element of stack is incremented by 1. Finally, when all the elements are exhausted value of 'cnt' variable is returned to the caller function.

```
//function to count the number of element(s) in the stack;
int stacks::count_elements()
   {
//initialize the t with top and cnt with 0.
int cnt=0,t=top;
// Continue till stack is not empty.
while(t>=0)
```

```
   {
   cnt++;   // increment the cnt
   t--;           // move towards the base
   }
   return cnt;
  }
```

All the functions discussed above can be called from the main function. As we know that the main function acts as a single point of execution for all the functions declared and defined earlier. Once this function is created, we can make usage of all the functions already defined.

```
// Using the functions already defined
int main()// Calling all the functions from the main function
{
int n,n1;
stacks s1;
// If the stack is empty
if (s1.isEmpty ())
cout<<"Stack is empty"<<endl;
else
cout<<"Stack is not empty";
s1.push (10);
s1.push (5);
//s1.push (7);
s1.push (14);
cout<<"Total number of elements in stack are: "<<s1.count elements()
<<endl;
s1.push (90);
s1.push (299);
cout<<" Element(s) in the stack are :"<<endl;
s1.display ();
n=s1.pop ();
cout<<" Element poped is "<<n<<endl;
getch ();
return 0;
}
```

5.4.2 Dynamic stack

Although, static stack is widely used data structure, however, static stack suffers with the following limitations:

- Number of elements to be inserted should be known before creation of stack.
- If the size allocated is smaller than number of elements to be inserted. In that case, many elements will not be inserted in the stack. At the same time, if the space allocation requested for the stack is much more than the number of elements to be inserted in that case huge space of the array cannot be utilized by the stack.

To address the above issues, dynamic stack can be used. Dynamic stack is implemented with the help of a node that is dynamically provisioned when the need for data storage is realized. (Users who are not familiar with link list are requested to please refer the link list chapter before reading this section for rich understanding)

Dynamic stack consist of two parts.

- Data part (represent data)
- Next pointer (Reflects the address of next node from the top).

Data part holds the data that is to be inserted in a stack. Top points to the current position of the stack and facilitates in insertion of the elements inside the stack. Whenever, any item is removed or inserted, it is top that is moved towards the beginning or end respectively. Working of the dynamic stack is depicted in the following program:

```
// Class to define the node of the stack
class stacknode
  {
  public:
  int data;
  stacknode*next;
  };
//Class stack to perform various operation
class stack
{
stacknode *top;
public:
stack();
void push(int);
int pop();
void display();
};
```

Initially, the dynamic stack will be empty. To highlight this condition, we have initialized the top with the NULL. Since, all the stacks will be created from the empty condition, therefore; we have initialized it with the help of constructor.

Once the items are inserted (push) into the stack, top will move towards the new position. Insertion of the stack is illustrated in the following figure 5.3. Initially, the stack is empty; therefore top will be pointing to the NULL. When 10 is inserted, top will point to the new position. Similarly, when 20 is inserted the new position of top is demonstrated in the figure 5.3 (c).

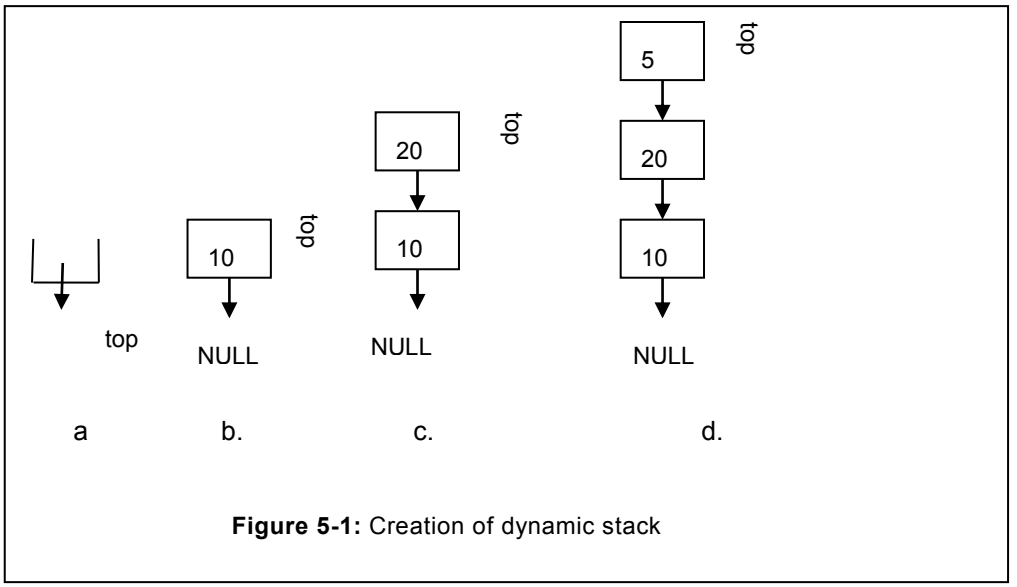


Figure 5-1: Creation of dynamic stack

```
stack::stack()
{
top=NULL;
}
void stack::push(int element)
 {
  if(top==NULL)
   {
   top=new stacknode;
   top->data=element;
   top->next=NULL;
   }
   else
```

```
   {
   stacknode *temp;
   temp=new stacknode;
   temp->data=element;
   temp->next=top;
   top=temp;
   }
 }
```

To display the current items in a dynamic stack, we use the current pointer, which accepts the address of the top and facilitates in retaining the value of top. In our program, temporary variable has been represented with the name current. To traverse the entire list of stack, while loop is used that executes till all the elements are not accessed.

```
void stack::display()
   {
   stacknode *current;
   current=top;   // storing the address of top to the current
   cout<<" Elements in the stack are: "<<endl;
   while(current!=NULL)
     {
     cout<<current->data<<"    ";  //display the data
     current=current->next;   // move to the next node, towards the
base
     }
     cout<<endl;
  }
```

To pop the items from the stack, we have used the pop function. This function will be used to remove the elements from the stack. Before taking out (pop) any element, stack is first examined for the availability of any. In case if no item is available in the stack, in that case message for the stack empty is displayed and the failure status is returned from the function. In case, if one or more item(s) are available in the stack, the item that is removed is returned. Same will be received by the caller for the future use. A function that describes the pop operation has been described as follows.

```
int stack::pop()// function to pop the element of the dynamic stack
   {
   if(top==NULL)
   {
```

```
   cout<<"Stack is empty "<<endl;
   return 0;
   }
   stacknode* temp=top;
   int el=temp->data;
   top=top->next;
   delete temp;
   return el;
}
```

Similarly, we can create the size () function for the dynamic stack that count the number of elements currently in the stack. To accomplish this task, we have started from the top and moved towards the base. During this traversal, variable representing the 'count' is incremented by one. Function to represent the size of a dynamic stack is depicted below:

```
// function to determine the size (number of count) of the dynamic
stack
int stack::size()
   {
int count=0;
if(top==NULL)
   {
   cout<<"No element in the stack "<<endl;
   return count;
   }
   stacknode *current;
   current=top;   // storing the address of top to the current
    while(current!=NULL)
     {
     count++;
     }
 return count;
 }
```

Sequence of calling the stack function is depicted in the main function. Calling of the functions sequence is entirely governed by the specific operation to be accomplished. However, first it will start with the push, since, first few elements need to be inserted before removing the element from the stack.

```
int main()// program to call the function from the main
  {
  stack s1;
  clrscr();
  s1.push(90);
   s1.push(5);
  s1.push(32);
cout<<"Element before pop"<<endl;
  s1.display();
cout<<"pop element "<<s1.pop()<<endl;
  cout<<"pop element "<<s1.pop()<<endl;
  s1.display();
  getch();
  return 0;
  }
```

5.5 STACK IMPLEMENTATION USING TEMPLATE

In earlier examples, we have implemented the stack depending upon the type of data to be inserted, for instance stack data is to be created integer type if the integer value is to be inserted or character type if character value need to be inserted. Now if the requirement changes where we need to implement stack for the complex data type or any other data type, in that case program need to be modified/rewritten again, this is followed by retesting. To avoid such situation, a stack for generic type may be created; such stacks are capable to implement the various types of data. Since, implementation will change for the dynamic relative to the static stack, therefore, we have separately implemented each of them. Same has been depicted in the upcoming sub-section.

5.5.1 Static stack using template

```
#include<iostream.h>
#include<conio.h>
const int size=5;
template <class t>
class stacks
{
      t stack[size];
      int top;
      public:
```

```
            stacks()
            {
               top=-1;
            }
      int push(t);
      t pop();
      int isEmpty();
      void display();
      int count_elements();
 };
// function to count the number of elements
template<class t>
 int stacks<t>::count_elements()
   {
   int cnt=0,t1=top;
   while(t1>=0)
   {
   cnt++;
   t1--;
   }
   return cnt;
  }
            template<class t>
            int stacks<t>::push(int element)
            {
               if(top==(size-1))
               {
                   cout<<"overflow element not entered\n";
                   return 0;
               }
               else
               {
                   stack[++top]=element;
               }
             return 1;
```

```
            }
      template<class t>
          t stacks<t>::pop()
            {
               t a=stack[top];
               if(top==-1)
               {
                   cout<<"underflow:\n";
               }
               else
               {
                   a=stack[top--];
               }
              return a;
            }
      //function to display the item in the stack.
                template<class t>
            void stacks<t>::display()
            {
               for(int i=top;i>=0;i--)
               cout<<stack[i]<<" ";
               cout<<endl;
            }
             template<class t>
            int stacks<t>::isEmpty()
            {
               if(top==-1)
               return 1;
               else
               return 0;
            }
// Completion of all function and starting of main function
int main()
{
int n,n1;
```

```
stacks<int> s1;
clrscr();
if(s1.isEmpty())
cout<<"Stack is empty"<<endl;
else
cout<<"Stack is not empty";
s1.push(10);
s1.push(5);
//s1.push(7);
s1.push(14);
cout<<"Total number of elements in stack are:
"<<s1.count_elements()<<endl;
s1.push(90);
//s1.push(299);
cout<<" Element(s) in the stack are:"<<endl;
s1.display();
n=s1.pop();
cout<<" Element pop is "<<n<<endl;
getch();
return 0;
}
```

5.5.2 Dynamic stack using template

```
#include<iostream.h>
#include<conio.h>
//using namespace std;
const int size=5;
template <class t>
class stacks
{
      t stack[size];
      int top;
      public:
           stacks()
           {
```

```
            top=-1;
           }
      int push(t);
      t pop();
      int isempty();
      void display();
      int count_elements();
 };
template<class t>
 int stacks<t>::count_elements()
   {
   int cnt=0,t1=top;
   while(t1>=0)
   {
   cnt++;
   t1--;
   }
   return cnt;
  }
           template<class t>
           int stacks<t>::push(int element)
           {
              if(top==(size-1))
              {
                  cout<<"overflow element not entered\n";
                  return 0;
              }
              else
              {
                  stack[++top]=element;
              }
            return 1;
           }
      template<class t>
          t stacks<t>::pop()
```

```
            {
               t a=stack[top];
               if(top==-1)
               {
               cout<<"underflow:\n";
               }
               else
               {
                   a=stack[top--];
               }
              return a;
            }
            template<class t>
            void stacks<t>::display()
            {
               for(int i=top;i>=0;i--)
               cout<<stack[i]<<" ";
               cout<<endl;
            }
             template<class t>
            int stacks<t>::isempty()
            {
               if(top==-1)
               return 1;
               else
               return 0;
            }
int main()
{
int n,n1;
stacks<int> s1;
clrscr();
if(s1.isempty())
cout<<"stack is empty"<<endl;
else
```

```
cout<<"stack is not empty";
s1.push(10);
s1.push(5);
//s1.push(7);
//s1.push(14);
cout<<"total number of elements in stack are: "<<s1.count_elements()<<endl;
s1.push(90);
//s1.push(299);
cout<<" element(s) in the stack are:"<<endl;
s1.display();
n=s1.pop();
cout<<" element pop is "<<n<<endl;
n=s1.pop();
cout<<" element pop is "<<n<<endl;
n=s1.pop();
cout<<" element pop is "<<n<<endl;
if(s1.isempty())
cout<<"stack under flow"<<endl;
else
{
n=s1.pop();
cout<<" element pop is "<<n<<endl;
}
getch();
return 0;
}
```

5.6 APPLICATION OF STACK

Stack is applied in variety of applications. In majority of applications, usage of stack is either explicit or implicit. For instance, in non-recursive traversal of a tree this data structure is explicitly used. In the recursion, it is implicitly used by the program. Stack is widely utilized in the application such as parenthesis matching, infix to postfix conversion, evaluation of complex expression etc. Applications utilizing stack have been discussed in the upcoming sub-section.

5.6.1 Parenthesis matching

To compute the complex expression, we enclose it in parenthesis to ensure that the expression is computed in correct order. It is equally important that correct number of parenthesis are used in right order. We have already learned in mathematics that the order of closing the bracket is governed by the order of opening brackets. We close the bracket that is opened last. It is illustrated with the help of following example.

2+ [4.9+1-{23*5-(54+12) +90/15} +98]

In the aforementioned example, the bracket '(' is opened at the end, accordingly it has to be closed first. Similarly, same rule is applicable for the other brackets. It is equally significant that the bracket opened is also closed. Otherwise, this will also leads to the parenthesis mismatch. In brief, followings points are to be examined for correctness of parenthesis:

- Brackets are opened and closed in the correct sequence.
- Numbers of open brackets have the correspondingly closing brackets.
- Closing bracket should not be more that the brackets opened.

To accomplish the above objectives, stack is extremely useful. To evaluate any expression, we consider that the expression is made of operand and operator. Symbols '+', '.', '*' etc. are known as operator. We scan the given expression, character by character; stack pushes the opening brackets. Once it encounters the closing bracket, it pops an item from the stack. The bracket that is received from the pop operation should be corresponding to the closing bracket. If these criteria are fulfilled, then it moves forward with other elements. In case of non-match, program terminates at this point itself and flashes the suitable message related to mismatch. To implement this task, we have used two functions based on brackets.

- Bracket checking.
- Expression evaluation

In the bracket checking program, each brackets are assigned priority. In the program, due care was taken to ensure that the brackets with the higher priority is not replacing the bracket with the lower priority. In other terms '('should not come before '['or '{'. Various possibilities that may take place with the parenthesis mismatching have been illustrated in the figure 5.4.

In our program, we would scan character by character from the beginning of the expression. Upon encountering the opening bracket, we will push it into the stack. Further, we would continue to scan the expression. Upon encountering the closing bracket, the top element of the stack will be popped by using the pop function. The item popped will be checked with the closing brackets, if it is correct matching then we will proceed further otherwise we terminate here itself and the error message will be flashed to the user.

To meet the case of extra opening bracket(s), if there is no element in the expression but stack is empty, this will imply that there are extra brackets in the expression. Similarly, if the closing bracket is encountered and the stack is empty, it is the case of extra closing brackets. All these cases have been considered in the following figure 5.4

To evaluate the expression considering the various cases discussed above, we have used the dynamic stack (linked stack) that was previously created by us. Program is discussed in the following example.

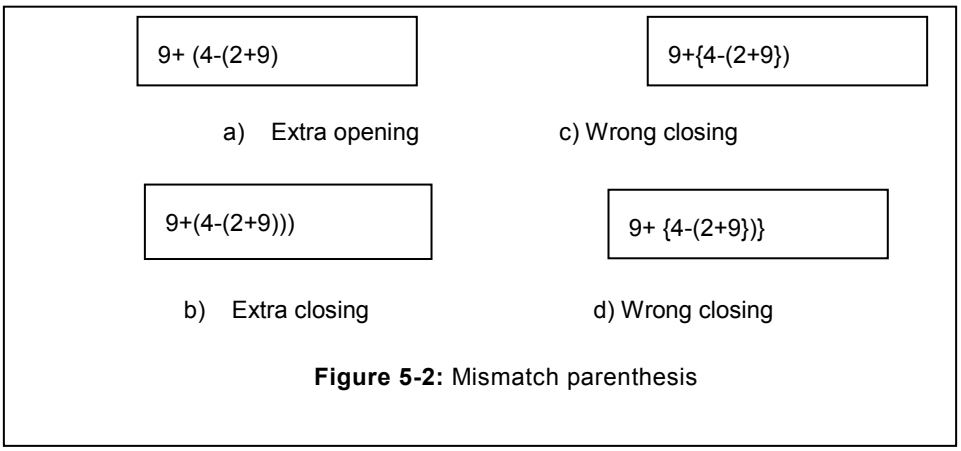


Figure 5-2: Mismatch parenthesis

```
Program: To match the parenthesis in the given expression
Solution:
#include<dyn.h>
class appl
{
public:
void parenth(char*); // function to match the parenthesis
int open_brack(char);   // To determine whether the opening of a
bracket
int close_brack(char); // To determine whether the closing of a
bracket
int matching(char, char); // To examine that opening bracket has the
……
                  // corresponding closing brackets
};
int appl::matching(char sym, char expr)
 {
 int ans=0; // initially, mismatch
 if((sym=='('  &&  expr==')')||(sym=='{'  &&  expr=='}')||(sym=='['  &&
expr==']'))
 ans=1;           // Matching exist
 return ans;
```

```
 }
 int appl::open_brack(char b)
  {
  int flag=0;
  if(b=='('|| b=='{'||b=='[')
  flag=1;
  return flag;
  }
  int appl::close_brack(char b)
  {
  int flag=0;
  if(b==')'|| b=='}'||b==']')
  flag=1;
  return flag;
  }
void appl::parenth(char expr[])
 {
 int i=0;
 stacks<char> s1;
 while(expr[i])
 {
 if(open_brack(expr[i]))
 s1.push(expr[i]);
 else
  {
 if(close_brack(expr[i]))
 {
 if(!s1.empty())
  {
 char symbol=s1.pop();
   if(!matching(symbol,expr[i]))
      {
      cout<<" Not a matching parenthesis"<<endl;
      return;
     // break;
```

```
      }
     // symbol='$';
  }//  empty
  else
  {
  cout<<"Extra closing"<<endl;
  return;
  }// else of empty
  }  // close brack
  }  //else
   i++;
 }        // while
 if(s1.empty())
  cout<<" Brackets are matching"<<endl;
  else
  cout<<" Non Matching Brackets"<<endl;
 }
int main()
{
clrscr();
appl a1;
char x[]="[(3+(9.4)+]";
a1.parenth(x);
getch();
return 0;
}
```

5.6.2 Conversion of infix expression to postfix expression

We evaluate the mathematical expression, in order to determine its value. Infix notation is widely used across the world in order to compute the expression. The key reason behind this is that it is human brain's friendly. We as a human being are much comfortable with the infix notation. However, computing the infix notation in computer poses tremendous challenge. Computer can use other notations for instance, pre-fix or postfix notations for the computational purpose. However, human being is comfortable in writing the infix expressions. Consequently, there is a need to change the expression written in infix notation into prefix or postfix notation. Stack is widely used data structure to convert the infix notation into prefix or postfix notation.

Prior to exploring the conversion methods, it is worth to understand that any expression is consist of two components namely operator and operand. Operators denotes the operations to be performed on operand. Consider the following expressions

3+6*5-56/28

We can classify them as

Operator: +, *, -, /

Operand: 3, 6, 5, 56, 28

During the conversion phase, priority of the expression operators aligning with the one BODMAS is taken into account. During the scanning phase, operators are pushed into the stack one by one. As soon as operator with lower priority from the one available at the top of the stack is encountered, previously pushed operators are popped from the stack. In the expression 3+6*5-56/28, first + will be pushed into the stack, this is followed by '*'. When – is encountered in that case, all the operators pushed earlier will be popped from the stack, since operator with lower priority from the top i.e. (*) of stack is encountered. Consequently, the expression will appear as

3,6,5 *+

Afterwards, push operation will commence again and rest of the operators will be pushed. Since, no operator with the lower priority is encountered but the expression is exhausted therefore, all the operators pushed into the stack will be popped. As a result, the complete conversion will result into the expression as:

3,6,5*+56,28/-

5.7 CASE STUDY BASED ON STACK

An airbase has the automatic parking system for the airplane. After completing the sortie, airplanes are parked at their respective places. These aircrafts are parked one behind another. Lane where the aircraft are taxing is sufficient to accommodate only one aircraft at a time for moving to and out from the parking (Refer the figure 5.5). Consequently, the aircraft that is parked at the last will go for the flying first. In each lane, only fixed number of aircraft can be parked. There are total 3 lanes (lane1, lane2 and lane3) and each of them is having the capacity of 4, 5, and 4 respectively. Diagram of parking area has been illustrated in the following figure 5.5.

Initially, airplanes parking start with one lane, upon filling the used lane only it proceed for the next land. Aircraft are parked first in lane1, once it is full to its capacity then the aircraft are parked in lane2; so on and so forth. You have to design a program to accomplish the following objective.

i) To park the aircraft in the respective lanes.

ii) Determine the number of aircraft parked in a particular lane.

iii) Change the lane of a new aircraft arriving, if the previous lane is full to its capacity.

iv) Determine the total number of aircrafts in all the lanes.

v) Generate the suitable message, if the parking lane is full to its capacity and more aircraft (s) are seeking the same lane for parking.

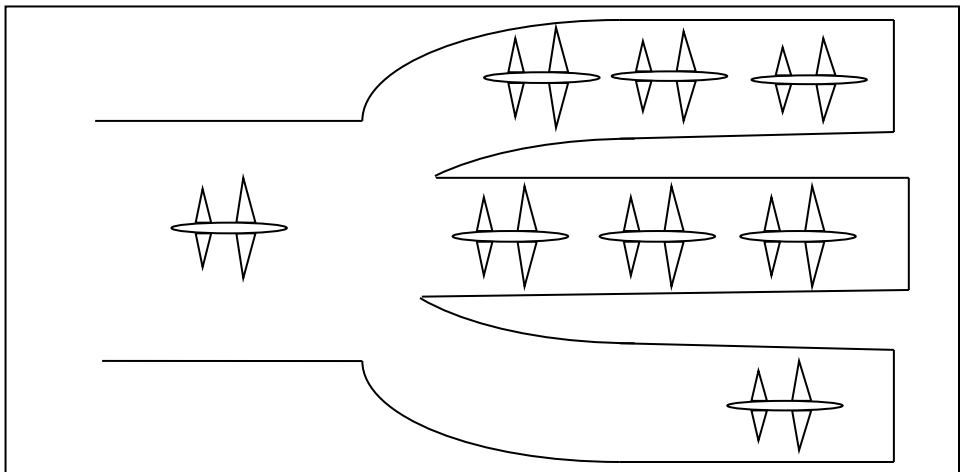


Figure 5-3: Aircraft parking area

EXERCISE

A. Descriptive Answers

1. What do you understand by stack? Write down the applications of stack.
2. What do you understand by FIFO. How its is applicable in the case of stack?
3. Write down few applications in real life where FIFO is implemented.
4. In static stack, do you think FIFO order can be violated? Justify your answer.
5. Write a stack program, in which top is initialize with 0 instead of -1 as given in the example. Correspondingly, write down other functions such as push and pop.
6. Write a program to display the content of a stack.
7. Discuss the drawback, if any, of initializing the stack top from 0.
8. Can we modify the static stack, so that the value of its size is supplied by the user at run time? Justify your answer.

9. Reverse the content of a given stack by using

a) One Stack

b) By using two stacks

10. Write a stack based program to find out whether the string is palindrome or not.

11. How we can we implement the stack using the queue based approach (If the concept pertaining to queue is not known, refer the next chapter and its implementation).

12. Is it possible to remove the in-between element(s) from the stack? Justify your answer in both the cases.

13. Write down a large number comprising of more than 5 digits. Store this number into stack, digit by digit. For instance, if number is 95487245 then it should be pushed in the stack in the following sequence 9, 5, 4, 8, 7, 2, 4, and 5.

14. In the above program, find out the sum of all the digits that are pushed inside the stack.

15. Write a program that stores two large numbers. Sum these two numbers using stack.

16. Accept the string; push this string character by character in the stack. Sequence to push the characters should be the reverse order of their arrival. For instance, if “Push” is to be inserted then insertion sequence should be h, s, u, P.

17. Extend the above program by finding out the substring from the string pushed into the stack. Substring is the partial string that consists of the part of the string and may be located at the starting, in between or at the last of the string. For instance, if the word inserted is ‘Ramayana’ then word such as ‘Rama’ ; ’maya’ are its various substring, existing at various position within the given string.

18. Create a function of the stack that will be utilized by us to determine the total number of element(s) in a stack. To accomplish this task, define the:

a) Count variable as member of the stack class.

b) Without including the count as data member.

19. How will you define the size of the dynamic stack? Discuss various constraints associated with the size of the stack.

20. Write a program to delete the elements from the stack. Upon deletion of the item, it should be inserted into another stack. When needed item deleted can be displayed for the audit purpose. Implement the case both in static and dynamic stack.

21. How the display of item is different in static stack with the dynamic stack.

22. Write down the cases where dynamic stack is well suited.

23. Write down the cases where static stack is ill-suited.

24. How the underflow condition is dealt in dynamic stack? Briefly, discuss various conditions associated with it.

B. Correct/Modify the given code

(Assume that variable are declared, and necessary variable(s) are in place)

1. In a given stack, top is initialized with 0. To push the element in the stack the following code is written:

```
int push(int element)
{
if(top==size)
cout<<”stack is full”;
return;
else
stack[++top]=element;
}
```

2. Check the correctness of the following pop function.

```
int pop()
{
int x=stack[top];
if(top<0)
{
cout<<”stack underflow”;
}
return x;
}
```

3. To pop the element from the stack the following code is written;

```
int pop()
{
int x=stack[top--];
if(top>size)
{
cout<<”stack underflow”;
}
return x;
}
```

4. In the display function, following code has been written:

```
void display()
{
int beg=0;
while(beg<=top)
cout<<stack[beg++]<<endl;
}
```

```
void display()
{
int t=top;
while(t>=0)
cout<<stack[t++]<<endl;
}
```

After correcting error from both the functions (if any), which version of them will be preferred?

5. In the dynamic stack, the code written is:

```
void push(int element)
{
snode * temp;
temp=new snode;   // creation of a new node
snode->data=element;    // assigning the numeric value
snode->next=NULL; // pointing the next to the NUL
top=temp;   // assigning the top to this new position
}
```

6. In the dynamic stack, to pop the element from the stack the following code is written

```
int pop()
{
int x=top->data;
if(top==NULL)
cout<<" Stack underflow"<<endl;
return;
delete top; // Remove the top node
top=top->next;    // move the top to the previous position
}
```

7. In a stack based template, pop function is written; find out the possible problem that may occur in this code.

```
t pop()
{
t x=top->data;
if(top==NULL)
cout<<” Stack underflow”<<endl;
delete top; // Remove the top node
top=top->next;    // move the top to the previous position
}
```

Chapter 6

Queue

Chapter Objective

- Defining queue with various operations permitted on a queue
- Creation of a static and dynamic queue
- Traversal in a queue
- Insertion and removal of items from a queue
- Circular queue
- Creating the queue using template
- Application of a queue.

6.1 INTRODUCTION

We have seen that in a stack, insertion and removal of the item was permitted only from one end. The end from where insertion and removal of the item was permitted was termed as top. Consequently, item inserted at last removed first from the stack. To ensure that the items are removed in the order they have inserted, a new data structure popularly known as queue can be used.

Queue is a data structure that can be considered as open from both the ends. Insertion is accomplished from one end known as rear, whereas removal of the item is taking place on the other end known as front. Same has been illustrated in the following figure.

Figure 6-1: Queue with its end

6.2 STATIC AND DYNAMIC QUEUES

Based on the creation method, queue can be categorized into static and dynamic queue. Static queue is assigned the size at the beginning of the program, whereas the dynamic stack

grows one element at at time based on the arrival of item only. Both of them are having their own pros and cons. Upcoming, sub-section presents both types of queue in detail.

6.2.1 Static Queue

If the size of the queue is declared at the time of creation, then such type of queue is known as static queue. In this queue, elements are inserted from the end known as rear, once the items are inserted, corresponding to that rear pointer moves from the front towards the rear end. Once the rear has reached to the extreme end (threshold of the queue) then more items cannot be inserted. If we attempt to insert more items in a queue that is already full then it is known as overflow.

Insertion of the items inside the queue is also known as enqueue. Initially, front and rear has

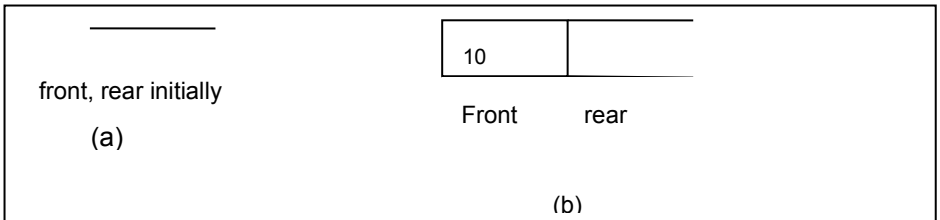


Figure 6-2: insertion into a queue

been initialized with -1. Therefore, they are not pointing to any of the index of a queue and denoting the empty condition. Once the first item is inserted than first is initialize with 0, correspondingly, rear is also incremented. Enqueue of item governs the position/movement of rear index of a queue. During insertion, maximum value of rear can reached equal to the size of the queue. Once rear is reached to the last position, insertion of any further elements will leads to overflow.

Complete working of a queue during insertion phase has been illustrated in the figure 6.3.

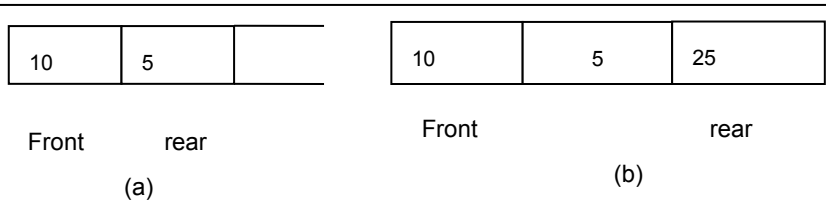


Figure 6-3: insertion into a queue to its capacity

In the static queue, first and foremost requirement is to define the size of the queue in order to ensure that the size needed for the queue can be allocated at the compile time itself.

During insertion of each elements, rear is checked to determine whether additional items can be inserted or it is already reached to its threshold. If it has not reached at the end position, in that case additional items can be inserted into the queue, otherwise suitable message (queue is full) is generated. Program depiting the static queue has been given as follows.

```
const int size=5;
class Q
{
      int front, rear;
      int queue[size];
      public:
      Q( );
      // To insert the element in the queue
      void enqueue(int);
      // To remove the element from the queue
      int dequeue();
      // To display the elements in the queue
      void display();
      //To investigate, if queue is empty
      int isEmpty();
      // To count the number of elements in the queue
      int count();
      // To shift the front to the 0th index
      void shiftElements();
…..// function to find out the first element, without changing the first   //order
int FrontElement();
};
Q::Q()
    {
     front=rear=-1;
    }

     void Q::enqueue(int element)
    {
     if(((rear-front)==(size-1))&&(rear==(size-1)))
```

```
    {
      cout<<"overflow element cannot be entered\n";
      return;
    }
    else if(front==-1)
    {
    queue[++rear]=element;
    front=0;
    }
     else if(rear<(size-1))
      {
      queue[++rear]=element;
      }
    }
```

6.2.1.1 Display of queue Elements

To display the queue elements, it's worth making use of front and rear index which denotes the range where data elements are lying. During the display of queue element(s), we have to start from the front and reach towards the rear. During this process, front or rear should not be shifted anywhere. To accomplish this objective, we can use the local variables as depicted in the following example.

```
void Q::display()
           {
           if(front>=0) //if any item in the queue
           for(int i=front;i<=rear;i++)
           cout<<queue[i]<<" ";
           }
```

6.2.1.2 Removing elements from a queue

Queue works on First in first out(FIFO) order. This implies that an element inserted at first will be removed first. Since, queue is having two ends (front and rear), it removes the elements from the front, consequently, front will move towards the rear. Before, removing any item from the queue, it is to be ascertained that the queue is not empty. Element removed from from a queue is to be stored in a temporary variable and same can be returned to the caller. Deletion of a queue result in creation of a space at a first position of a queue. Complete function of dequeue has been presented herein:

```
int Q::dequeue()
      {
       int a;
       if((rear== -1)||(front== -1))
      {
        cout<<"underflow:\n";
        return 0;
      }
        else
        {
          a=queue[front];
          front++;
         if(front>rear)//after removal if queue is empty
          {
            front=rear=-1;
          }
        }
        return a;
       }
```

6.2.1.3 Counting number of elements

To count the number of elements in a queue, we have to begin from the front. Since front may be at 0^{th} index or may be away from it, if some of the items are removed (dequeue). Counting should continue upto the rear element as it denotes the last position (index) of an element.

```
// Program to count the number of elements in the queue
int Q::count()
{
int f,r, c=0;
f=front,r=rear;// initialize f with front and r with rear
while(f++<=r) //execute the loop till last element is not reached
{
c++;  //count the elements
}
return c;   // return the count
}
```

6.2.1.4 To find out the first element of a Queue

We can determine the first element in a queue without changing the first index. Once the element(s) of a queue is displayed, it will again display all the element(s) including the first element that has been returned by the FirstElement() function. Code of the function is depicted in the following example:

```
// Program : Write the program to determine the first element in a queue without changing the first index.
int firstelement()
 {
if(first==0)
{
cout<<" queue is empty";
return 0;
}
int x=queue[first];     // Store the first element of the stack
return x;   // return the first element
}
```

6.2.2 Main function

Functions defined in the preceding sub-sections have been performing their objectives. To call them, we can make use of main function, where they can be called as illustrated in the following example.

```
int main()
{
clrscr();
Q q1;
q1.enqueue(90);
q1.enqueue(5);
q1.enqueue(78);
q1.enqueue(35);
q1.enqueue(10);
q1.enqueue(14);
cout<<" Total number of elements "<<q1.count()<<endl;
q1.display();
cout<<"Dequeue item: "<<q1.dequeue()<<endl;
cout<<"Dequeue item: "<<q1.dequeue()<<endl;
```

```
cout<<" Total number of elements "<<q1.count()<<endl;
q1.display();
if(q1.isEmpty())
   cout<<"Queue is empty"<<endl;
  else
  cout<<"Not empty"<<endl;
getch();
return 0;
}
```

6.2.3 Limitation of queue

Queue suffers from the major limitation due to dequeue operation performed on a queue. In dequeue, front is moved towards the rear. As a result, there will not be any item from the 0^{th} index upto front. Although, space exist but user will not be able to utilize that space, since front is moved ahead of the zeroth index. To overcome this limitation, elements can be moved from their middle position to zero indexes onwards. Consequently, front and rear value needs to be changed.

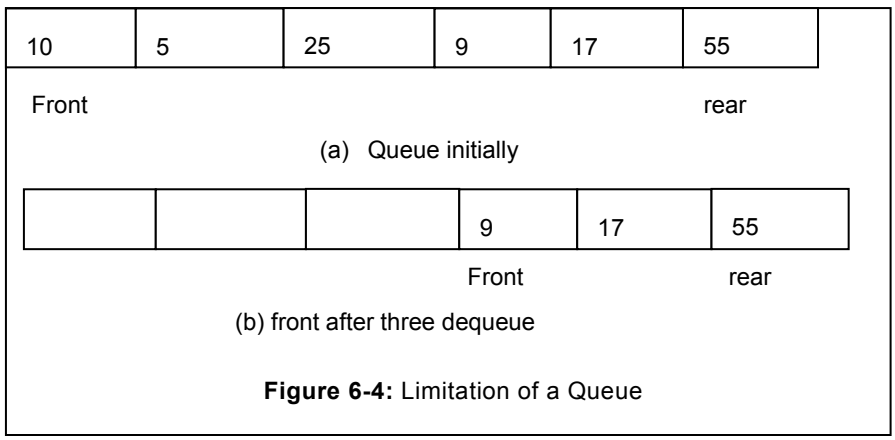


Figure 6-4: Limitation of a Queue

Shifting operation may be preferred for the small size of list; however for the queue of huge size, shifting element will be an expensive affair and will need huge resources for shifting the elements.

```
// To shift the Element after dequeue
void Q::shiftElements()
{
int f=front,t=-1;
while(f<=rear)
```

```
 {
 queue[++t]=queue[f++]; // initialize the front member in the 0th
index
 }
 front=0;
 rear=t;
 cout<<endl<<"Items shifted in the Q"<<endl;
```

6.3 DYNAMIC QUEUE

Limitation of static queue is overcome with the help of dynamic queue. Dynamic queue can grow to any extent. Only limitation is the hardware limit. In dynamic queue there is no need for testing the full condition, as it is governed by the hardware capacity. However, empty condition need to be tested similar to previous case.

Structure of the dynamic queue has been defined with the help of following class. Various functions along with their objectives have been given in the following program.

```
//Structure of queue node
class qnode
{
public:
int info;
qnode *next;
};
class queue            //class for queue
{
//defining the front and rear pointer
qnode *front,*rear;
public:
//constructor for queue
queue();
//function to enqueue in dynamic queue
void enqueue(int);
//function to deque in the queue
int dequeue();
//To display the items in the queue
void display();
};
```

Initially, there are no elements in the queue; therefore, front and rear have been initialized with NULL. To accomplish the task of initialization, constructor has been used.

To enqueue the element, two cases have been considered:

- No element(s) in the queue.
- There is one or more element(s) in the queue.

Once, there is no element, at that point in time, front and rear both will be NULL. Therefore, while inserting the first element, rear is initialized and allocated the space. In this case, front and rear will be pointing to the same element.

In the second condition, element will be inserted from the rear. Consider the node to be inserted is termed as temp, connect the rear to this temp and move the rear to this last position. Same has been depicted in the following function.

```
void queue::enqueue(int x) // To enqueue the element in a queue
{
if(rear==NULL)    // If queue is empty
 {
 rear=new qnode;
 rear->next=NULL;
 rear->info=x;
  front=rear;
 }
else        // if more than one element are in the queue
 {
  qnode *temp;
temp=new qnode;
temp->info=x;
temp->next=NULL;
rear->next=temp;
rear=temp;  // moving the rear to new last position
}
}
```

To dequeue elements from a queue, the element inserted first is removed first. In dequeue, three cases may occur:

- Queue is empty.
- Queue has only one element
- Queue has more than one element

If the queue is empty, it signifies there is no item in the queue. Consequently, no items can be removed and unsuccessful flag(0) is returned.

In the second case, item is removed from the front. Upon dequeue of the item, the queue will be empty. To reflect this change, first and rear need to be initialized with NULL.

In the third case, where the queue has more than one elements, front has to be moved to the next available slot. Data item and the pointer of the previous front that is stored in the temp have to be deleted, in order to free the space that was earlier occupied by the front pointer.

```
// Function to dequeue the element from the queue
int queue::dequeue()
{
int x=front->info;
if(front==NULL)
 {
 cout<<"Queue is empty"<<endl;
 }
 else
 {
 qnode *temp=front;
 if(front==rear)
 {
 front=NULL;
 rear=NULL;
 delete temp;
 return x;
 }
 front=front->next;
  }
 return x;
 }
```

During enqueue items are inserted, whereas in the dequeue items are removed from the queue. After the enqueue and dequeue operation, in majority of the cases there will be some items that are left in the queue. To witness the change that is reflected, we can use the function display. This function demonstrates the items currently in the dynamic queue.

To display the items, we begin from the front and continue to access the node till we are not reaching the end of a queue. The end position of queue is represented by the rear.

```
void queue::display()
 {
 qnode *current;
 if(front==NULL)
 {
 cout<<"queue is empty"<<endl;
 return;
 }
 current=front;
  while(current!=rear)
 {
 cout<<current->info<<endl;
current=current->next;
}
cout<<current->info<<endl;
}
```

The isEmpty () is one of the other functions that has the significant role in this program. It can supports both the enqueue and dequeue. During enqueue, it can be used to check whether the queue is empty or not. Correspondingly, the next course of action can be initiated.

This function also plays a significant role while calling the dequeue function. We can include isEmpty() into other functions as a condition. If the queue is empty then dequeue operation cannot be performed. isEmpty() function has been defined in the following program.

```
int queue::isEmpty()
  {
  int flag=0;
  if(front==NULL)
   {
   flag=1;
   }
   return flag;
   }
```

All the above discussed functions can be called from the main function. Order and arguments to be passed have been given in the main function. Same has been depicted in the following main function.

```
// Main function to call other queue function
void main()
 {
 clrscr();
 queue      q1;
q1.enqueue(40);
q1.enqueue(27);
if(q1.isEmpty())
 {
 cout<<"Q is empty"<<endl;
 }
//q1.enqueue(12);
//q1.enqueue(64);
q1.display();
int j;
j=q1.dequeue();
cout<<" element deleted "<<j<<endl;
q1.display();
j=q1.dequeue();
cout<<" element deleted "<<j<<endl;
if(q1.isEmpty())
 {
 cout<<"Q is empty"<<endl;
 }
q1.display();
j=q1.dequeue();
cout<<" element deleted "<<j<<endl;
q1.display();
getch();
}
```

6.4 QUEUE USING TEMPLATE

Dynamic queue has the wide usage in data structure. Dynamic queue is primarily used in traversal of a tree, particularly for the breadth wise traversal. At the same time, we enjoy the flexibility of unlimited size of the queue. Queue presented in the preceding section suffers from the limitation that for the different data type, program need to be modified suitability. For

instance, the program has been discussed used integer data type, if we have to use the character data type, enumerated program needs to be suitably modified.

In order to address such issues and to adopt the program for any data type, we can use template. Program that depicts the use of queue template t has been given as follows:

```
#include<conio.h>
#include<iostream.h>
//Structure of queue node
template<class t>
class qnode
{
public:
t info;
qnode<t> *next;
};
//class for queue
template<class t>
class queue
{
//defining the front and rear pointer
qnode<t> *front,*rear;
public:
//constructor for queue
queue();
//function to enqueue in dynamic queue
void enqueue(t);
//function to deque in the queue
t dequeue();
//To display the items in the queue
void display();
//To check for queue empty or not
int isEmpty();
};
template<class t>
queue<t>::queue()
{
```

```
front=NULL;
rear=NULL;
}
template<class t>
void queue<t>::enqueue(t x)
{
if(rear==NULL)
 {
 rear=new qnode<t>;
 rear->next=NULL;
 rear->info=x;
  front=rear;
 }
else
 {
  qnode<t> *temp;
temp=new qnode<t>;
temp->info=x;
temp->next=NULL;
rear->next=temp;
rear=temp;
}
}
template<class t>
void queue<t>::display()
 {
 qnode<t> *current;
  if(front==NULL)
 {
 cout<<"queue is empty"<<endl;
 return;
 }
 current=front;
  while(current!=rear)
 {
```

```
 cout<<current->info<<endl;
current=current->next;
}
cout<<current->info<<endl;
}
template<class t>
t queue<t>::dequeue()
{
t x=front->info;
if(front==NULL)
 {
 cout<<"Queue is empty"<<endl;
 }
 else
 {
 qnode<t> *temp=front;
 front=front->next;
 if(front==NULL)// if no node in the bst
 rear=NULL; // point the rear also to the NULL
 delete temp; // Delete the node
 }
 return x;
 }
 template<class t>
 int queue<t>::isEmpty()
  {
  int flag=0;
  if(front==NULL)
   {
   flag=1;
   }
   return flag;
   }

void main()
```

```
 {
 clrscr();
 queue<int> q1;
q1.enqueue(40);
q1.enqueue(27);
if(q1.isEmpty())
 {
 cout<<"Q is empty"<<endl;
 }
//q1.enqueue(12);
//q1.enqueue(64);
q1.display();
int j;
j=q1.dequeue();
cout<<" element deleted "<<j<<endl;
q1.display();
j=q1.dequeue();
cout<<" element deleted "<<j<<endl;
if(q1.isEmpty())
 {
 cout<<"Q is empty"<<endl;
}
q1.display();
j=q1.dequeue();
cout<<" element deleted "<<j<<endl;
q1.display();
getch();
}
```

6.5 CIRCULAR QUEUE

As mentioned earlier, queue suffers with a limitation due to the movement of front towards rear. Consequently, empty space created due to the movement of front is not utilized. Even though shifting has been suggested as one of the methods to remove the empty space at the front of a queue, however, overhead caused due to shifting is large and cannot be ignored. To address this issue, circular queue is used. Functionality of a circular queue has been discussed as follows.

In a circular queue, items are still inserted from the rear. Insertion operation remains continue till front is not encountered. In a circular queue, value of rear moves between '0' to the 'size - 1'. When rear attends the maximum value, its next value is reinitialized with zero. This is accomplished with the usage of mod operator. We are aware of that the mod is widely used in calculating the remainder and this remainder will not go beyond the value of its right side number. For instance, if the size of the queue is 4 then due to the usage of remainder method, the potential values will lie in the range of 0 to 3 only.

To insert the element in the queue, we have created the 'insert' function. This function accepts one argument as integer type and does not return a value. To insert the values, we have increment the rear. Once rear is incremented, it is always examined with the front; so that it should not overlap the front (does not insert the value more than the capacity of the queue). Complete function has been enumerated in the following example.

```
// Program to insert the item in a circular queue.
 void insert (int &n)
           {
              if ((rear+1) %size! =front)
              {
               rear= (rear+1) %size;
              queue [rear] =n;
              if (front<0)
               front=0;
              }
              else
              {
                 cout<<"Queue overflow, element not inserted: \n";
              }
           }
```

After the elements are inserted, then the other major task is to remove the items from the queue. Before removing the item from the queue, it is examined that queue is not empty. During removal of the item, it is removed from the front of the queue. Correspondingly, front moves towards the rear index. Here, we have considered the two significant cases of queue:

- Queue has only one element (front and rear at the same position).
- There is more than one element in the queue.

Both these cases have been considered in the remove () function discussed herein:

```
// Program to remove the items from the queue
 int remove ()
           {
```

```
       int a=0;
       if (! empty ())
       {
        if (front==rear)            //only one element in the queue
         {
          a=queue [front];
          front=rear=-1;
          }
          else
          {       // More than one element in the queue
           a=queue [front];
           front= (front+1) %size;
          }
       }
      else
      {
         cout<<"Queue underflow: \n";
       }
       return a;
      }
```

The other significant function is empty function. This function ascertains whether the queue is empty or not. If the queue is empty it returns ‘1’ otherwise it returns ‘0’.

```
// function to check if any item in the list or empty
 int empty ()
           {
             if (front==-1)
             return 1;
             else
             return 0;
           }
```

Finally, it is worth discussing the display function. Although, it appears to be simple function similar to the one we have learned other display function. However, it is significantly different since value need to hover between ‘0’ and the ‘size-1’. Upon attending the maximum value, it further gets initialized with zero. In the discussed function, we have started from the front position and continued till we have not reached to the tail (last element). Once last element is reached, loop is terminated and the remaining items are displayed. Before, displaying the

element(s), it is ascertained whether the queue is empty or not. If the queue is empty, no more items are displayed; instead corresponding message is displayed (that the queue is empty).

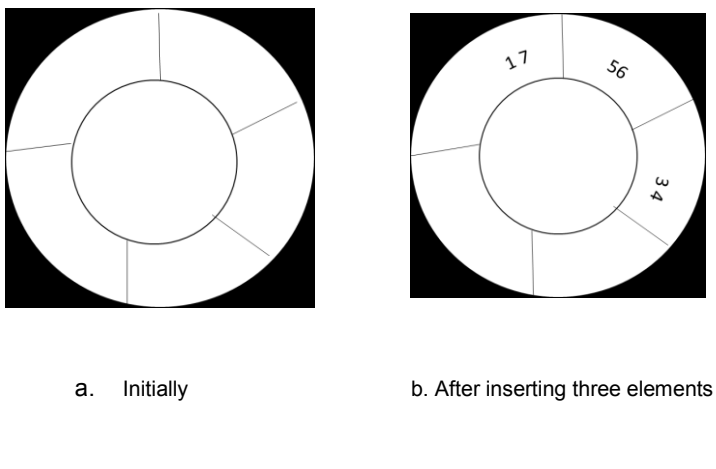
```
     void display ()
           {
              if (empty ())
              {
                cout<<"Empty queue: \n";
              }
              else
              {
                  int r, f;
                  r=rear;
                  f=front;
                 // examine that both front and rear are not at same
position
                              while (f! =r)
                  {
                  cout<<queue[f] <<endl;
                  f= (f+1) %size;
                  }
                  cout<<queue[f] <<endl;
              }
           }
```

Function discussed above can be included in the class as shown in the following program. The class members can be called by the class instance from the main function. Calling of various members has been illustrated in the following program.

```
#include<iostream.h>
#include<conio.h>
const int size=4; // Size of the queue
class CQ
{
      int queue [size];
      int front, rear;
      public:
           CQ ()
```

```
            {
              front=-1;
              rear=-1;
            }
// function to insert the element
void insert (const int &);
// function to display the element
void display ();
// function to remove the element
void remove ();
// To examine whether the queue is empty or not
int empty ();
};
int main ()
{
 CQ q1;     // Creating instance of the queue
clrscr ();
q1.insert (10);   //inserting the element in the queue
q1.insert (45);
q1.insert (90);
q1.insert (21);
cout<<"Elements in the queue are "<<endl;
q1.display ();    // display of the queue
q1.remove ();
q1.remove ();
q1.remove ();
cout<<"Elements after removal"<<endl;
q1.display ();
q1.insert (17);
q1.insert (56);
q1.insert (34);
cout<<"Elements in the queue are "<<endl;
q1.display ();
q1.remove ();
q1.remove ();
```

```
cout<<"Elements after removal"<<endl;
q1.display ();
getch ();
return 0;
}
```



a. Initially b. After inserting three elements

Figure 6-5: Circular Queue

6.6 PRIORITY QUEUE

In our daily life, we do not follow the first come first serve sequence always. Instead, we have to assigned priority to one service over the other. For instance, at the toll booth, police vehicle, ambulance, VIP's vehicle etc. do not need to pay the taxes, at the same time they receive more priority over others. Similarly, there are many other instances where one task is receiving higher priority over others. In such cases ordinary queue does not meet the needed purpose, instead a new queue termed as priority queue is used.

In the priority queue, elements with higher priority are moved towards front relative to the elements with lower priority. This arrangement ensures that the elements with higher priority are served first. To achieve this objective, priority is assigned to every elements of a queue. Before they are served, if any element with the higher priority arrives it displaces the element with lower priority towards back. Such queue where the elements are arranged in the order of priority instead of arrival time is known as priority queue.

Priority queue can be implemented with the help of heap. Complete implementation has been enumerated in the section describing heap.

6.7 CASE STUDY BASED ON QUEUE

In a hospital there are a number of department based on disease. Major department exist are Dermatology, Cardiology, Gynecology, Pathology, etc. Each department is capable to serve the patient in increasingly fair manner. To avail the treatment facility, patient visits to the relevant department where (s)he is allotted a token number. Allotment of token number is purely on the basis of arrival time. Separate queue is maintained for each department.

Hospital maintains the following information about the patient.

Table 6.1: Patient Information

S.No	Information	Type of information
1.	Patientid	Stores in alphanumeric
2.	Social_id	Numeric value
3.	Occupation	Character value
4.	Disease_reported	Character value
5.	Treatment prescribed	Character value
6.	Previous treated	Yes/No
7.	Token number	Number
8.	dateOfVisit	Date

Based on the above information, solve the following challenges the hospital has to encounter.

- Allot the unique token number to each patient.
- Store all the information related to the above field.
- Total number of patient in a particular queue.
- How many patients waiting in the hospital.
- How many patients arrive in a day (you can take the average of week or month)?
- Which type of data structure is preferred by you? Explain its advantages over any other data structure which you may consider near appropriate.

EXERCISE

A. Short Answer Type Questions

1. How to define the queue? Write down the various operations that are permitted in a queue?
2. How many ends are used to insert and remove the elements from the queue? How this phenomenon is implemented in the queue?
3. What are the major drawbacks of a queue? How these drawbacks are addressed?
4. What are the various methods with the help of which we can overcome the drawback of queues?
5. Implement the function that counts the number of element in the queue?
6. How circular queue is different from the ordinary queue?
7. How the circular feature in a circular queue can be implemented in a queue, justify?
8. During the insertion of elements in the circular queue, what are major considerations that need to be addressed?
9. Write a function to search an element in a circular queue?
10. Write a program to reverse the stack elements using queue?
11. Write a program to concatenate the element of two queues using operator (+) overloading.
12. Write a program to compare the two arrays on the basis of element count, by using operator (>) overloading.
13. Write a function to implement the priority while inserting the elements in the queue.
14. In a circular queue, how to ensure that once all the elements are dequeue then it is pointing to the Null condition.

B. Correct the code, if any

1. In the given queue, front and rear has been initialized with 0. Insert function of the queue has been given as:

```
void insert (int x)
{
if (rear+1==size)
cout<<”queue overflow”<<endl;
++rear=x;
}
```

2. In the circular queue, front and rear has been initialized with -1. Insert function of the queue has been given as:

```
void insert (int x)
```

```
{
if (rear+1==size)
cout<<”queue overflow”<<endl;
rear+1=x;
}
```

3. In the circular queue, front and rear has been initialized with -1. Insert function of the queue has been given as:

```
void insert (int x)
{
if (rear+1==front)
cout<<”queue overflow”<<endl;
rear+1=x;
}
```

4. In dynamic queue, front and rear has been initialized with NULL. Correct the following code for enqueue.

```
void enqueue (int x)
{
if (front==rear)
{
front=new qnode;
front->data=x;
front->next=null;
}
else
{
qnode *temp;
temp=new qnode;
temp->data=x;
temp->next=null;
rear->next=temp;
}
```

5. In dynamic queue, front and rear has been initialized with NULL. Correct the following code for dequeue.

```
qnode* dequeue ( )
{
if (front==rear)
```

```
{
cout<<”queue is empty”;
}
else
{
qnode *temp;
temp=front;
rear=rear->next;
}
}
```

6. Is it possible to remove the item from the middle of the queue? True/False

7. In queue, inserting the first element and other element is to be considered in the same manner. True/False

8. What is the reason for using a "circular queue" instead of a regular one?

a) The running time of enqueue () is improved

b) Reuse empty spaces

c) You can traverse all the elements more efficiently

d) None of the above

9. One difference between a queue and a stack is:

A. Queues require linked lists, but stacks do not.

B. Stacks require linked lists, but queues do not.

C. Queues use two ends of the structure; stacks use only one.

D. Stacks use two ends of the structure, queues use only one.

10. If the characters 'D', 'C', 'B', 'A' are placed in a queue (in that order), and then removed one at a time, in what order will they be removed?

A. ABCD

B. ABDC

C. DCAB

D. DCBA

11. Suppose we have a circular array implementation of the queue class, with ten items in the queue stored at data [2] through data [11]. The current capacity is 42. Where does the insert method place the new entry in the array?

A. data [1]

B. data [2]

C. data [11]

D. data [12]

12. Consider the implementation of the Queue using a circular array. What goes wrong if we try to keep all the items at the front of a partially-filled array (so that data [0] is always the front)?

A. The constructor would require linear time.

B. The getFront method would require linear time.

C. The insert method would require linear time.

D. The isEmpty method would require linear time.

13. In the linked list implementation of the queue class, where does the insert method place the new entry on the linked list?

A. At the head

B. At the tail

C. After all other entries those are greater than the new entry.

D. After all other entries those are smaller than the new entry.

14. In the circular array version of the Queue class, which operations require linear time for their worst-case behavior?

A. getFront

B. insert when the capacity has not yet been reached

C. isEmpty

D. None of these operations require linear time.

15. In the linked-list version of the Queue class, which operations require linear time for their worst-case behavior?

A. getFront

B. insert

C. isEmpty

D. None of these operations require linear time.

16. If data is a circular array of CAPACITY elements, and rear is an index into that array, what is the formula for the index after rear?

A. (rear % 1) + CAPACITY

B. rear % (1 + CAPACITY)

C. (rear + 1) % CAPACITY

D. rear + (1 % CAPACITY)

17. I have implemented the queue with a circular array, keeping track of front, rear, and Many Items (the number of items in the array). Suppose front is zero, and rear is one less than the current capacity. What can you tell me about many Items?

A. manyItems must be zero.

B. manyItems must be equal to the current capacity.

C. count could be zero or the capacity, but no other values could occur.

D. None of the above.

18. I have implemented the queue with a linked list, keeping track of a front node and a rear node with two reference variables. Which of these reference variables will change during an insertion into a NONEMPTY queue?

A. Neither changes

B. Only front changes.

C. Only rear changes.

D. Both change.

19. Priority Queues suppose getFront is called on a priority queue that has exactly two entries with equal
priority. How is the return value of getFront selected?

A. One is chosen at random.

B. The one which was inserted first.

C. The one which was inserted most recently.

D. This can never happen (violates the precondition)

Chapter 7

Tree

Chapter Objective

- Defining tree and its terminology
- Defining Binary search tree (BST)
- Creation of the BST using recursive and non-recursive method
- Traversal of a BST using recursive and non-recursive method
- Deletion a node from a BST
- Constructing the BST based on traversal series
- Searching the element in a BST
- Description of various methods to determine the height of a BST
- To determine the mirror of a tree.

7.1 TREE

Data structure can grow in linerar or non-linear fashion, accordingly data structure can be categorized into two as mentioned hereunder:

- Linear.
- Non-linear.

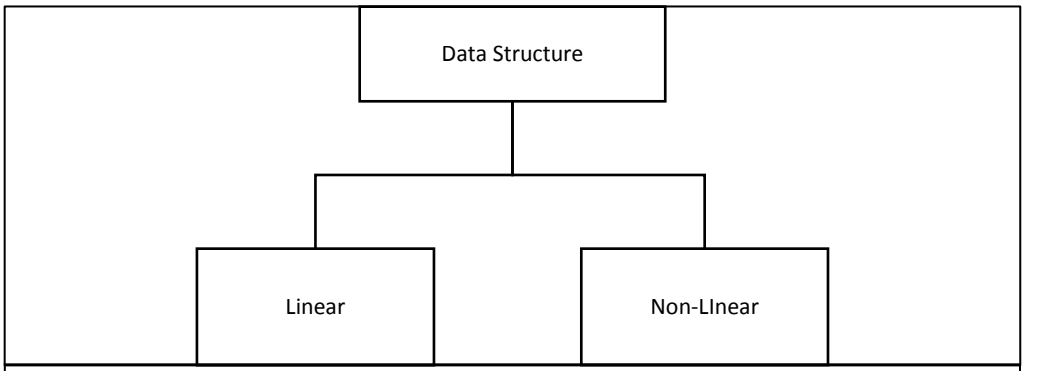


Figure 7-1: Types of data structure

Various topics discussed earlier in the book including Stacks, Queue, Link list would fall under the category of linear data structure. In linear data structure, items are arranged one after the other. During, accessing of item, traversal takes place by visiting all the nodes previously placed. Complexity, of linear data structure is O(n). Correspondingly, performance is the major bottleneck in linear data structure. To improve the performance, a new data structure known as tree is devised. Tree is an example of non-linear data structure. In non-linear data structure, data are arranged in parent-child relationship. Same has been

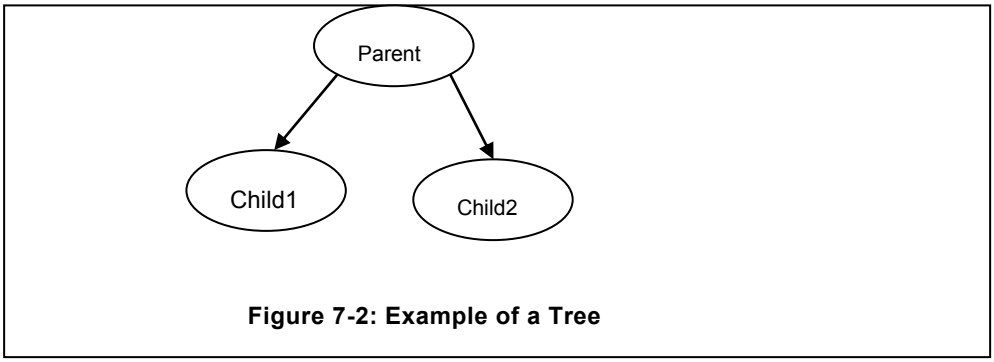


Figure 7-2: Example of a Tree

demonstrated in the following figure.

Tree categorization is determined by the maximum number of child that any parent can accommodate at any point in time. However, a node may have less number of nodes than the maximum number of a node that it can accommodate. Trees are further defined based on their degree. For instance tree with degree '2' is termed as binary, tree of degree '3' is termed ternary, whereas the tree of degree 'n' is termed as n-ary tree. For this chapter, we have considered the binary tree only.

7.2 BASIC TERMINOLOGY

At any point in time, a tree may hold a number of nodes, which are arranged in parent and child relationship. The node from where tree originates that node is termed as root. Node after the root node is child of another node, this will continue till we do not reach at the leaf node. Each node is acting as child of other node (except the root node). Therefore, tree can be defined as non linear data structure, organized in parent child relationship. Each node is again acting as a tree.

Before discussing the tree, it is worth discussing some of the terms that will be frequently used throughout our discussion.

Siblings: Children of same parent are known as siblings. In the tree, illustrated in the figure 7.3, the two nodes with the values 5 and 40 are siblings, since both of them are children of the same parent i.e. node 20.

Non-terminal node: Nodes in a tree with at least one child are known as non-terminal node. It can also be defined as all the nodes other than the terminal nodes are termed as non-

terminal node. In figure 7.4, nodes with the value 20, 5, 40 are the example of non-terminal nodes.

Leaf Node: Leaf nodes are the one in a tree that don't have any child node. Leaf nodes are also termed as terminal node. In figure 7.4, node with the value 9 and 40 are examples of terminal nodes, since they don't have any child node.

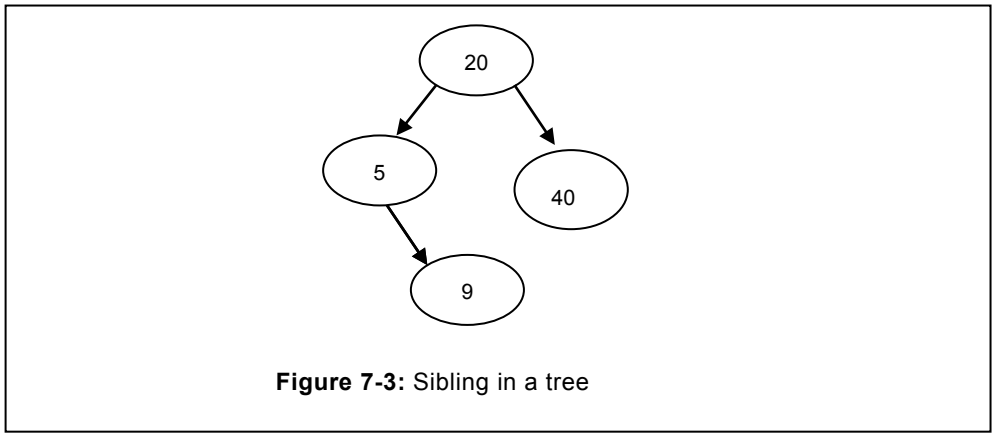


Figure 7-3: Sibling in a tree

Degree of a tree

In a tree the maximum number of child nodes that a particular node can have is termed as degree of a tree. If a node can have maximum two children then its degree is two, it is also known as binary node. Examples of a degree have been given in the following figure 7.4.

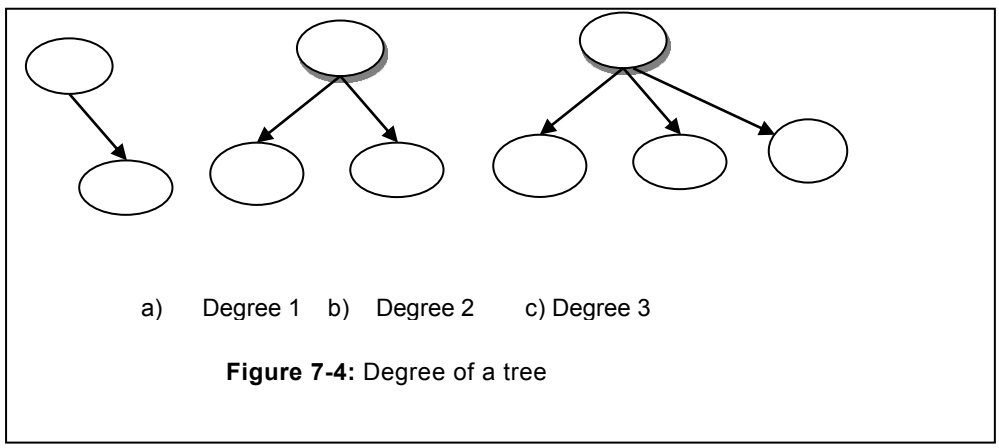
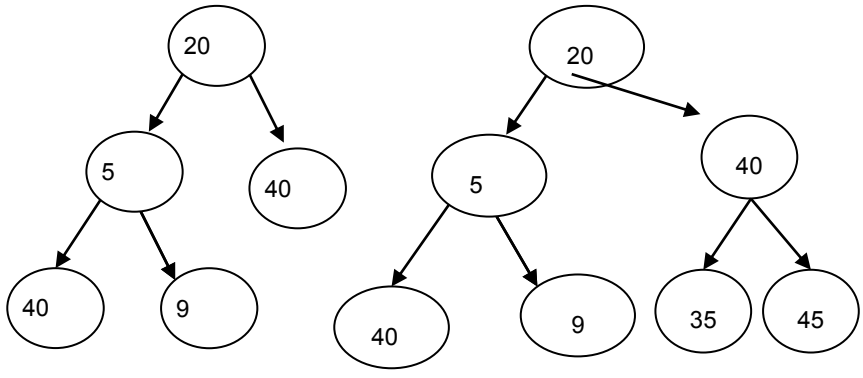


Figure 7-4: Degree of a tree

Strictly Balanced Binary tree: Trees in which all the children have at-least two nodes or no node (in case of leaf node). Such a tree is known as strictly binary tree. In the following figure strictly tree has been illustrated. Strictly balance tree is also known as almost balanced tree.

Completely Balanced Binary tree: A binary tree can be termed as completely balance binary tree, if all its nodes, other than the leaf nodes are having both the nodes. In addition, all the leaf nodes must be at the same level. This property makes the completely balanced tree different from the almost balanced binary tree.



a) Strictly Balance binary Tree b. Completely Balance Tree

Figure 7-5: Balance binary tree

7.3 BINARY SEARCH TREE

Binary search tree (BST) is the special case of a binary tree in which nodes are iplaced depending upon their values. BST has to follow additional restrictions that are applicable to the binary tree:

- All the right nodes are more than the parent node.
- All the left nodes are less than the parent node.
- Comparison begin from the root, and continues upto leaf node is not encountered. i.e left pointer or the right pointer is not NULL.

Therefore, while creating the BST special attention is to be paid, as explained in the upcoming section.

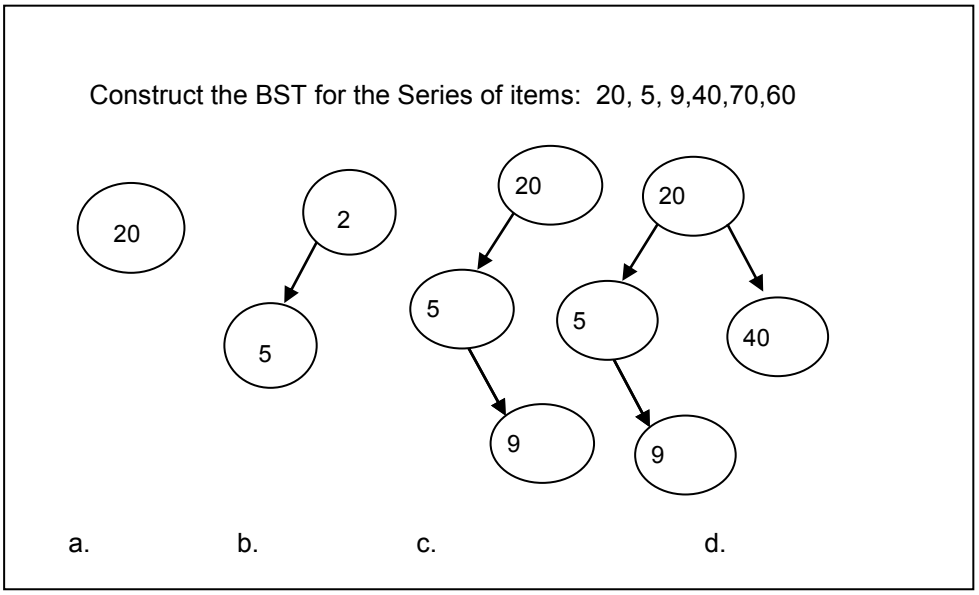
7.3.1 Creation of Binary search tree

Consider that the list of items (20, 5, 9,40,70,60) as shown in figure 7.6, is required to be inserted into the BST. The first item of the list is assigned to the root node of the BST, rest are compared with the root element. According to their values, elements are routed to the left or right hand side of the root according to the tree definition.

As illustrated in the figure 7.6, first the root node is created and it is having the value of 20 and its left and right node will be pointing to the NULL. The other elements are compared with this node. Element with greater to its parent node, they will be attached to the right, otherwise

it will be attached to the left. Consequently, on the arrival of next item i.e. 5, it will be compared with the root node (20), since it is less than the root node therefore, it will head towards the left hand side. Since, no left node is existing so this node will be attached to the left hand side and will have the value 5. The next number that is arrived is 9, again the comparison starts from the root node(20), since 9 is less than 20 so it will be heading towards the left, after heading one level down left, node with value of 5 is reached, since, 9 is more than 5 therefore it will head towards the right of 5, since, no node is attached to the right of 5, that's why this node will be attached to the right of 5, refer figure 7.6 b.

The next number in the series is 40, again the comparison will start from the root i.e. 20, since 40 is greater than the 20 it will head towards the right of 20, since the tree don't have any node attached to the right of 20, the new node having value of 40 will be attached to the right of 20, it is illustrated in the figure 7.6 'c'. Continuing to create the tree, the next node to be inserted in the tree is with the value of 70. Corresponding to the previous node, comparison will start from the root node (20), since, 70 is more than 20 it will head towards the right hand side, going one level down, it is again compared with the node having value 40, since still it is greater, therefore it will head towards the right of this node also, since its right is NULL, consequently, it will be attached to the right of this node, same has been illustrated in the figure 7.6 'd'. The next node to be inserted is having value 60. Corresponding to the previous insertion, comparison will restart from the root node(20), since it is more than 20, therefore it will head towards the right of this node, during heading one level down towards the right, 40 is encountered, since 60 is more than the 40, therefore, it will head towards the right of it. Now, 70 will be encountered, since 60 is less than the 70, therefore, it will head towards the left of it. Since, its left is NULL, consequently, 60 will be attached to this node. Final tree has been illustrated in the figure 7.6 'e'.



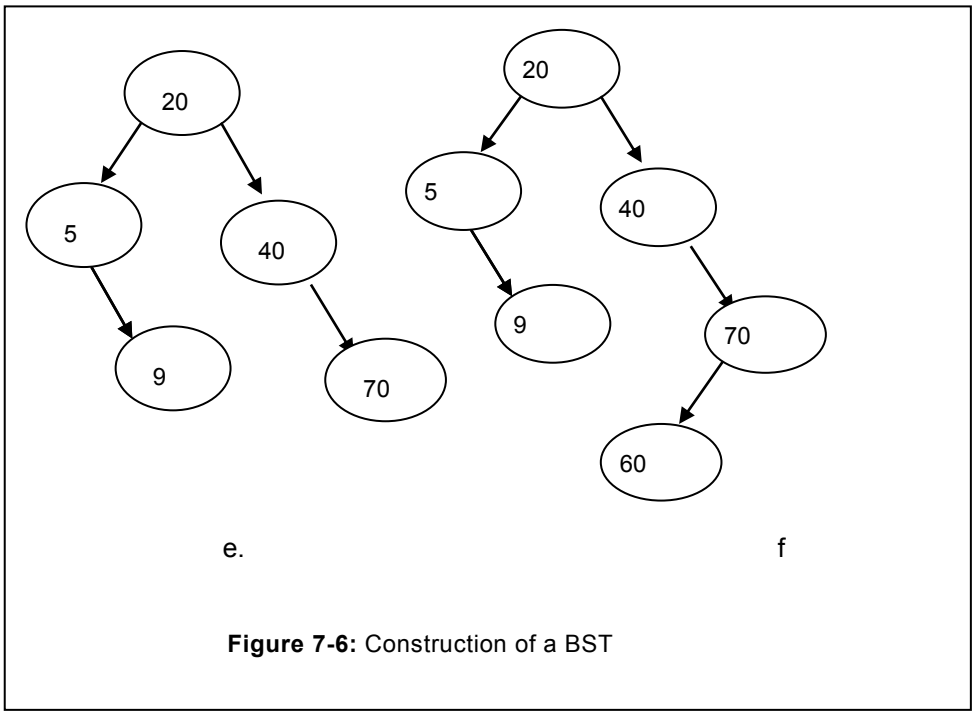


Figure 7-6: Construction of a BST

A. Program to create BST (Non-recursive)

First we would like to define the structure of the BST. The structure of the BST comprise of three items i.e. left child pointer, right child pointer, and the data part. Structure of the BST, as discussed above has been defined as follows:

```
class tnode
{
public:
int info;
tnode *left, *right;
};
```

It is significant to understand that the member created here must be public, otherwise they will not be accessible outside the class.

Class to create the BST, along with the members has been shown below:

```
class bst
{
tnode *root;
void create(int);
};
```

Creation of a tree is started with the root node. This can be accomplished by the following statements.

```
root=new tnode;
root->left=root->right=NULL;
root->data=element;
```

Consequently, structure as illustrated in the following diagram will be created.

```
// Program to create a BST
void bst::create(int element)
 {
 //creating the root of bst
 if(root==NULL)
 {
 root=new bnode;
 root->info=element;
 root->left=root->right=NULL;
 return;
 }
```

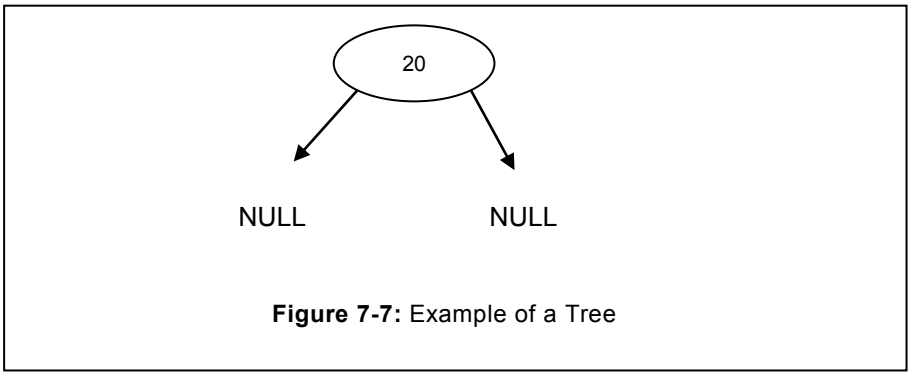


Figure 7-7: Example of a Tree

```
 //creating a node to be inserted in a bst(root already exist)
bnode *temp,*r;
temp=new bnode;
temp->info=element;
temp->left=temp->right=NULL;
//initializing the r with root
r=root;
while(r!=NULL)
{
// if element is less than r then move to the left
if(element < r->info)
   {
   // if left is NULL attach the temp node to the left side.
  if(r->left==NULL)
  {
  r->left=temp;
  break;
  }
  else
    r=r->left;   // Move to the left
 }
 else if(element > r->info)
    {
    // if element is more than r move toward the right
    if(r->right==NULL)
```

```
     {
     //if right is NULL attach temp to the right
      r->right=temp;
     break;
      }
      else
      r=r->right; // go to the right node
    }
 }
}
```

7.3.2 Recursive BST Creation

In the recursive BST creation, the BST will be created that will be similar with the one created with the help of non-recursive method. However, in the recursive function we are making use of recursive call. In our program, we have divided the working of the program as:

- Creation of root.
- Inserting the left or right node in a BST.

To accomplish first objective, we have created the rec_create (int data) function and same has been depicted in the following program. Logic begins by taking the root into account. If the root is not NULL (tree is already existing), in that case rec_create (bnode*, int) is called, otherwise, the root will be created and the program will return to the caller function.

```
// Function for recursive call
void bst::rec_create (int data)
{
if (root==NULL)   // if no
 {
 root=new bnode;
 root->left=root->right=NULL;
 root->info=data;
 return;
 }
 bnode *r=root;
rec_create(r, data);
}
```

If the tree already exists, then the new value will be compared with the root value, if the new value is less, it will be routed towards the left side. If the left side is NULL, then the new node will be created and it will be attached to the left side.

In the other case, if the new value to be inserted is greater than the root node, it will be routed to the right side. If the right side is NULL then a new node with the passed value will be created and attached to the right side of the tree.

Recursive function has been depicted in the following program.

```
void bst::rec_create (bnode *r, int data)
  {
 if (data < r->info)
   {
   if(r->left==NULL)
     {
     bnode *temp;
     temp=new bnode;
     temp->info=data;
     temp->left=temp->right=NULL;
     r->left=temp;
     return;
     }
     rec_create(r->left, data);
  }
  else
    {
    if (data> r->info)
     {
     if(r->right==NULL)
       {
       bnode *temp=new bnode;
       temp->left=temp->right=NULL;
       temp->info=data;
       r->right=temp;
       return;
       }
       rec_create(r->right, data);
```

```
    }
    }
    }
```

7.4 TRAVERSAL IN A BST

Visiting each node of a tree once is known as traversal of a tree. To display the value of each node, we need to visit each node of a BST. Different types of traversal method exist, there categorization is governed by the sequence of visiting the node and its branches. Different types of traversal that can be accomplished are:

- Pre order traversal
- In order traversal
- Post Order traversal

Each of them has been discussed in the following sub-section.

7.4.1 Pre-order Traversal

In the pre-order traversal, we are visiting the information part first then move towards the left side. Once the extreme left node is reached (next left is NULL), then we try to move towards the right. After reaching to the right again, we attempt to move to its extreme left. During the left side traversal, we continue to access the information as we have done before. If no right node is available then we reach to one level up and try to visit its right node. This process continues till all the nodes are not visited.

With the aforementioned discussion, we can conclude the sequence of traversal in pre-order as follows:

- Info
- Left node
- Right node

To execute the above steps, we can follow recursive as well as non-recursive approach. Recursive approach is extremely simple and needs few line of codes, whereas non-recursive approach is lengthy and requires usage of another data structure that includes stack as well as queue. Program to create the stack and queue has already been defined in the respective chapters. In our discussion, we have considered both the approaches.

7.4.2 Recursive approach for Pre-Order traversal

In the recursive approach, we visit the node, access the information and continue to traverse towards the left till we do not encountered the NULL. This condition has been given in the base condition. If the NULL is encountered, it traverses the right side but only once and again continues traversal towards the left as discussed before. The traversal method is illustrated in the following figure 7.8.

Consider the case of figure 7.8, here the traversal will commence from the root node. In this case it is 60, data of this node will be displayed and the node will be pushed into the stack. Now, we will be heading towards left since the visited node was not NULL, this new node is 40, and again it will be pushed into the stack. Correspondingly, we will continue to move towards the left till the visited node is not NULL, at the same time, node visited is inserted into the stack. Consequently, next traversed nodes are 30, 25. After reaching to 25, we cannot go to the left further, since its left is NULL. Then we will attempt to move towards the right of the node that was inserted in the stack. In this case, it is 25; the right node to this node is 27. This node is visited, and we would like to continue towards the left of this current node, since, there is no left node therefore, false condition will be encountered. From the stack, we will pop the other node here the node is 30. Since its left is already visited, then we would attempt to traverse to its right. Since it right is not NULL, therefore, the next visited node will be 35. However, its left is NULL; therefore we would not be able to head towards the left. Instead, we would access the right node, in the given tree it is 37. Since, it doesn't have any child node; therefore we would pop the node from the stack. Now, we will get the node with the value 40. For this node, we would go one step right and attempt heading towards the left as earlier.

In the recursive function, we have to just write three statements in sequence of calling and rest of the actions are implicit. However, in the case of non-recursive method, programmer has to provision for the complete logic. Non-recursive function of the BST has been enumerated in the next sub-section. Traversal in pre-order is enumerated immediately after the following program.

Pre-order traversal of the BST illustrated in the figure has been given as follows:

It will begin with root node i.e. 60, continue to traverse towards the left till Null is not encountered. Same has been highlighted with arrow. Pre-order traversal will be:

60, 40,30, 25, 27, 35, 37, 40, 49, 60, 80, 90

```
//Recursive function for the pre order traversal.
void bst::pre_order (const tnode *roote)
           {
              if (roote! =NULL)
              {
                        cout<<roote->data<<endl;
                        pre_order (roote->left);
                        pre_order (roote->right);
              }
           }
```

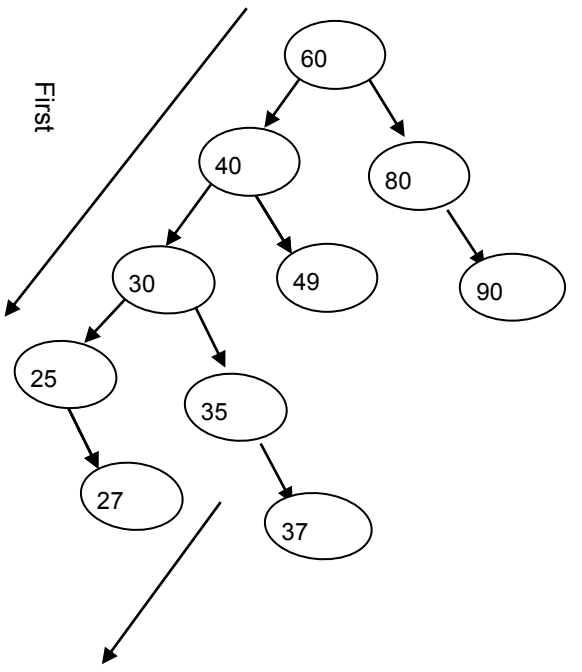


Figure 7-8: Pre-order traversal in a BST

7.4.3 Non-recursive traversal

We can also traverse in pre-order by using non-recursive approach. Traversal method in non-recursive method is not different with the one using the recursive approach. However, it is much more complex relative to the recursive approach defined earlier. In the non-recursive function, we initialize the pointer of root into the new pointer 'x'. To simplify our work, we can import the file that enumerates the dynamic stack defined in the stack chapter.

In the pre-order non-recursive approach, we use the stack to push the tree node explicitly. In this approach, we visit the node and move towards left by inserting the node into the stack. If the NULL is encountered, the node that is pushed at last is pop first. We set the flag as visited to avoid traversal again towards the left side. The node that is popped will be checked for the presence of right node, if the right side is present then it traverses towards the right. After visiting the right, there is possibility that it may have left node(s), consequently, the visit flag is set to false again- Now, traversal will continue towards left. This process will continue till the stack will not become empty. The entire function has been depicted in the following program.

```
Program: To traverse the tree in pre-order, using non- recursive approach.
void bst::pre_orderNR ()
{
bnode *x=root;    //initializing the x with the root
int visited=0;          // initializing the visited as false
stacks<bnode *> s;      // declaring the stack as BST node type
s.push(x);        // Push the root into the stack.
do
{
while ((x! =NULL) && (! visited))
{
cout<<"info ="<<x->info<<endl;      // Traverse the node
s.push(x);        // Push the node into the stack
x=x->left;  // continue to move left
}
visited=1;  // Make the visited true, to avoid re-traversal
x=s.pop (); // Pop one node from the stack.
if(x->right! =NULL)     // if right node exists
 {
 x=x->right;      // move towards the right
visited=0;  // set the visited value to false
}
} while (! s.empty ()); // Continue till node in the stack exist
}// End of function
```

7.4.4 In-order Traversal

In the in-order traversal, the following sequence is followed to visit the nodes:

- Left node
- Info
- Right node

In the in-order traversal, we head towards the extreme end of the left branch of the tree. During this traversal, nodes are stored in the stack. Once the extreme end of the left is reached then we can access the information. Afterwards, if the right node of the tree is not empty then we traverse towards the right. Again, we start traversal towards the left hand side by pushing the nodes into the stack, as earlier. Upon encounter of NULL, this process is over,

then we pop the one node and access the information, after that we check for the presence of right node, if it exists then move to the extreme end to its left branch else pop other node stored in the stack. In-order traversal of a BST illustrated in the figure 7.9 has been given as follows:

```
In-order Traversal of a BST (figure 7.9)
25, 27, 30, 35, 37, 40, 49, 60, 80, 90
```

It is apparent from the earlier discussion that we need to follow a repetitive approach for the traversal. Therefore, we will use the following function to accomplish the in-order traversal:

- Recursive
- Non-recursive

7.4.5 Recursive Approach

In the recursive approach, we use recursive function to traverse the tree. In the recursive approach, function has the BST node type pointer as an argument. It accepts the root node, following the in-order definition discussed earlier; traversal is taking place in the BST.

The function in-order (root->left) facilitates in heading towards the left side of a tree, level by level. Once the extreme end of the left branch is reached, after that we move one step right, if it exist. It is followed by extreme end of the left branch of this new right node. After reaching to the extreme end information is displayed of a node that doesn't have left or right node, we move one level up, visit the information part after that check for the presence of right node. Again procedure discussed above is repeated.

In the recursive function, stack is utilized; however usage of stack remains implicit to the user. Therefore, complexity of managing the stack by the programmer is eliminated. We can conclude that during the in-order traversal, information can be visited only if all the branches of left side do not exist.

```
In-order Traversal of a BST (figure 7.9)
25, 27, 30, 35, 37, 40, 49, 60, 80, 90
```

Recursive function of the BST has been depicted as follows:

```
Program: Recursive in-order approach to traverse the BST
void inorder (tnode *root)
{
if (root! =NULL)
{
inorder (root->left);   // visit the left node
cout<<"info ="<<root->info;   // visit the information
inorder (root->right);  // visit the right node
}}
```

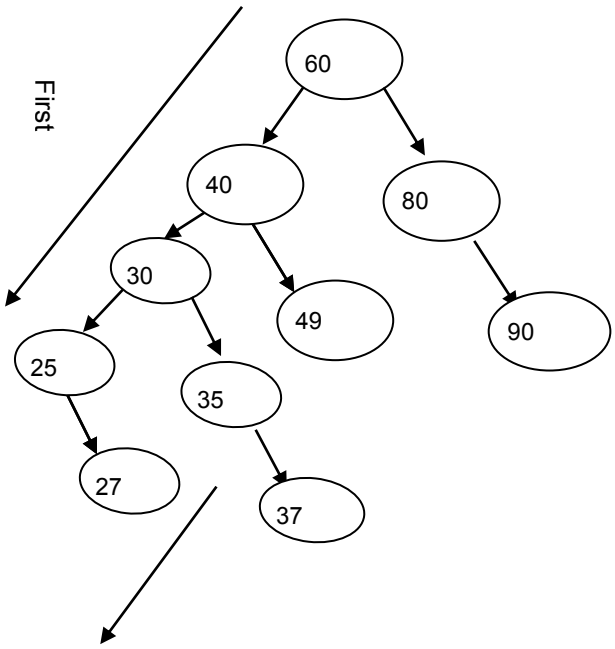


Figure 7-9: In-order traversal in a BST

7.4.6 **Non-recursive approach**

In the non-recursive approach, programmer has to manage the various sequence of operation needed for the traversal that include creating the stack, push and pop of BST node required for insertion and deletion respectively. Traversal will take place as already explained in the recursive approach. Managing the stack has been depicted in the following program.

```
// Program: Non-recursive approach for the in-order traversal of a BST
#include<stack.h> //include the stack header file defined earlier
void bst:: inorderNR()
{
//initializing the root with r
bnode *x, *r=root;
//initializing the variable visited with 0
int visited=0;
//creating the stack for the BST node type
```

```
stacks<bnode*> s1;
x=r;
//pushing the first element into the stack
s1.push(x);
while(!s1.empty())
  {
  //x=s1.pop();
  //Go to the left if not already visited and
  //Traversed at the extreme end of left branch
  while((x->left!=NULL)&& (!visited))
  {
  // Move to the left
  x=x->left;
  // Push the node in the stack
  s1.push(x);
  }
  // Make the visit true as left side is visited
  visited=1;
  // pop the last node from the stack
  x=s1.pop();
  // visit the extreme left node popped
  cout<<x->info<<endl;
  // check for the presence of right node
  if(x->right!=NULL)
   {
   x=x->right;
   s1.push(x);
   //if right node present, make the visit false
   visited=0;
   }
 }
```

7.4.7 Post order Traversal

In the post order traversal, sequence of visiting various nodes and information has been given as follows:

- Left node
- Right node
- Info

From the above sequence, it is apparent that first we will move towards the extreme left node, then right node and again towards extreme left node. Once no child is available or the child already visited then only the info is traversed. After traversing the extreme left node (leaf node here) the node stored in the stack is popped and same procedure is to be applied. We continue to apply this procedure till all the nodes are not visited.

```
void bst::post_order(const tnode *roote) // Program to traverse in
post order
           {
              if(roote!=NULL)
              {
             post_order(roote->left);
             post_order(roote->right);
             cout<<roote->data<<endl;
              }
           }
```

Traversal of the nodes has been given as follows: 27,25, 30, 35, 37, 40,49,

```
27, 25, 37, 35, 30, 49, 40, 90,80, 60
```

7.4.8 Constructing the tree from the given traversal

We can construct the tree, if we have given the traversal such as in-order and pre-order, or in-order and post-order. Consider a case that the pre-order and in-order traversal has been given as follows:

Pre-order traversal of a tree is: 60, 30, 20, 10, 25, 27, 60, 70, 65, 68, 80

In-order traversal of a tree is: 10, 20, 25, 27, 30, 60, 65, 68, 70, 80

It is possible to construct the BST from the given traversals. To construct the BST, if two traversals have been given, one is in-order the other is pre-order or the post-order in such case, BST can be easily constructed. We know from the various traversal enumerated earlier in the chapter that during pre-order traversal, first node that is traversed is the root node. Locate this value in the in-order traversal and divide the given traversal in left and right half.

In the in-order traversal, values that are less than the value of pre-order is to be placed in left half whereas, values greater than the first node of the pre-order value are placed in the right half. Same has been depicted in the following figures.

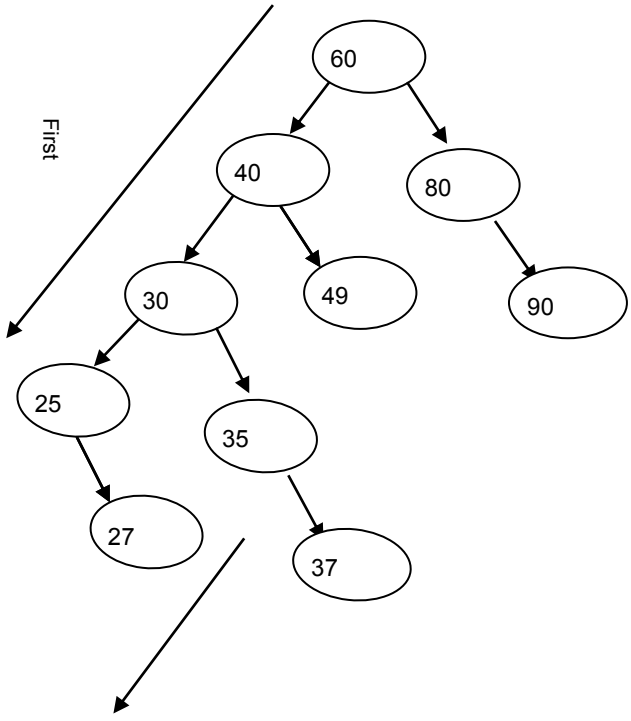
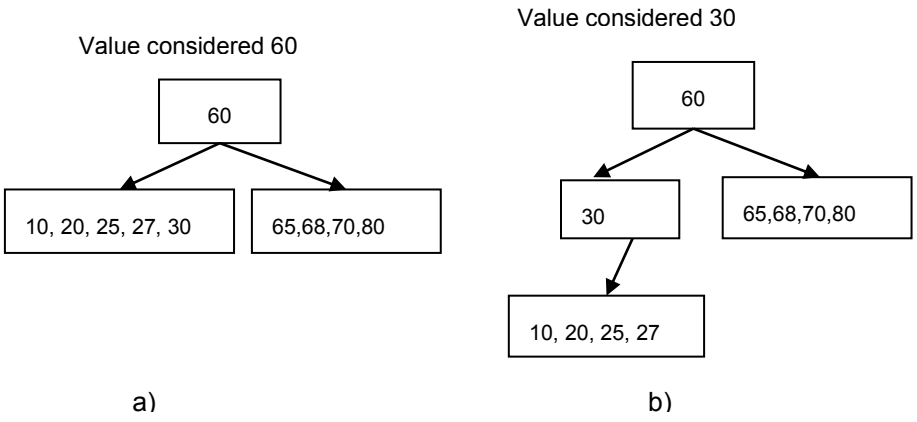


Figure 7-10: Post-order traversal in a BST

Pre-order traversal of a tree is: 60, 30, 20, 10, 25, 27, 60, 70, 65, 68, 80

In-order traversal of a tree is: 10, 20, 25, 27, 30, 60, 65, 68, 70, 80



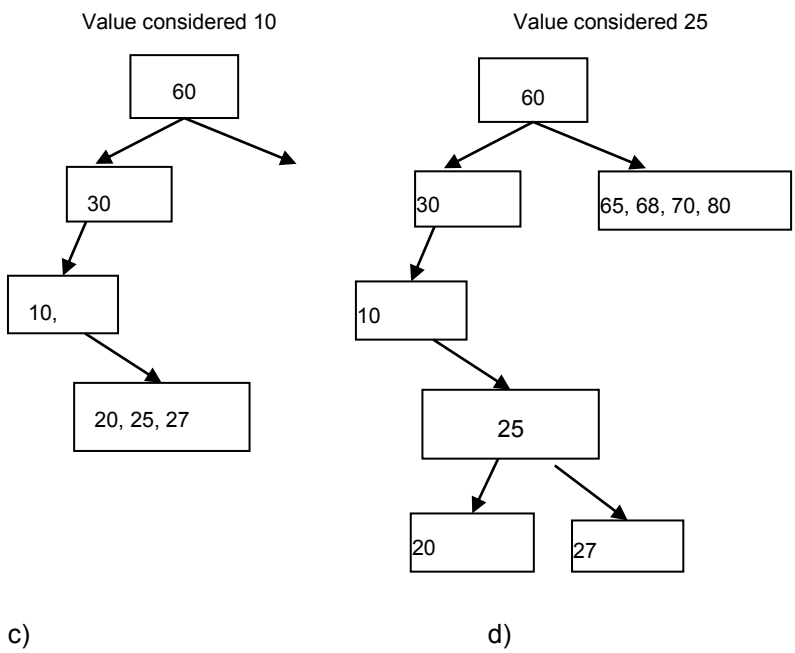


Figure 7-11: Constructing the tree from given in-order and pre-order

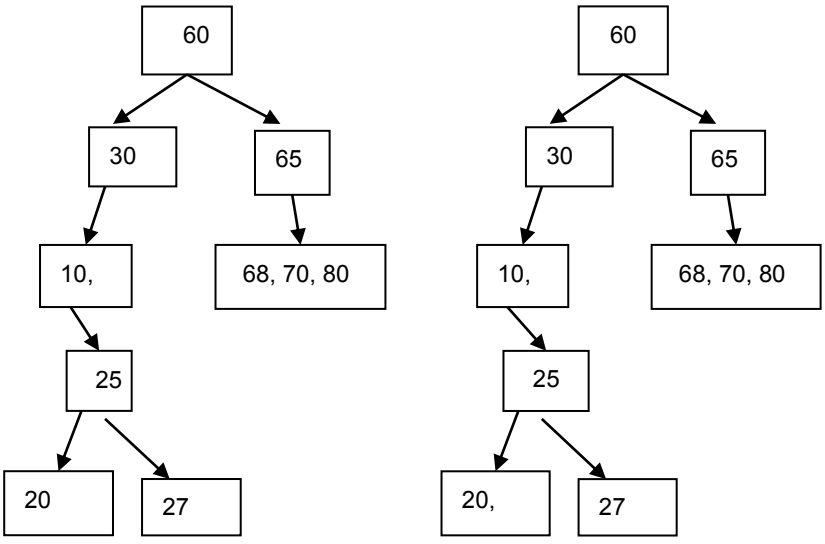


Figure 7-12: Constructing of BST from the given traversal

7.5 DELETION OF A NODE

Data inserted in the BST is required to be deleted in the event when it is no longer needed. Deletion of a node in a BST is more cumbersome relative to the insertion of the node. The node to be deleted is termed as target node. During deletion of a target node, followings are the various cases that have been considered:

- Target node does not have any child node (leaf node).
- Target node has only one child (may be left or right).
- Target node has both the children.

Deletion of a node for all the above cases has been discussed as follows:

Case a): To delete a node that does not have any child, we need to determine its parent. Once the parent is known, simply make the child pointer of this parent node as NULL, since it will not have any other node.

Case b): In this case, we have considered that the node to be deleted has only one child. This child node may be at the left or the right branch of the target node of a BST. In this case, we need to determine the parent node of the node to be deleted. Once, the parent node is determined, we have to find out whether the node to be deleted is left of the parent or the right node. Connect the parent node of the target node (node to be deleted) to the child node of the node to be deleted. Same has been illustrated in the figure 7.15 (a).

Resultant tree has been illustrated in the figure 7.15 (b).

Case c): In the third case, we have considered that the target node (node to be deleted) has both the children (left and right child). In this case also, we need to determine the parent of the target node. Since, the node to be deleted has two children; therefore both the children cannot be attached to the parent node of the target node. Since, the parent may already have one child. As a result, total number of nodes to be attached to the parent node will become three. Which is not possible, due to the structure limitation, and structure is governed by the tree definition. To delete such a node, we can follow any one of the following methods.

- Delete by copy
- Delete by merging

7.5.1 Delete by Copying

To delete the node by copying, we do not delete the node physically. Instead, we find out the in-order predecessor or the in-order successor as well as their parent. Value of the node to be deleted is copied with the node of in-order predecessor or in-order successor already

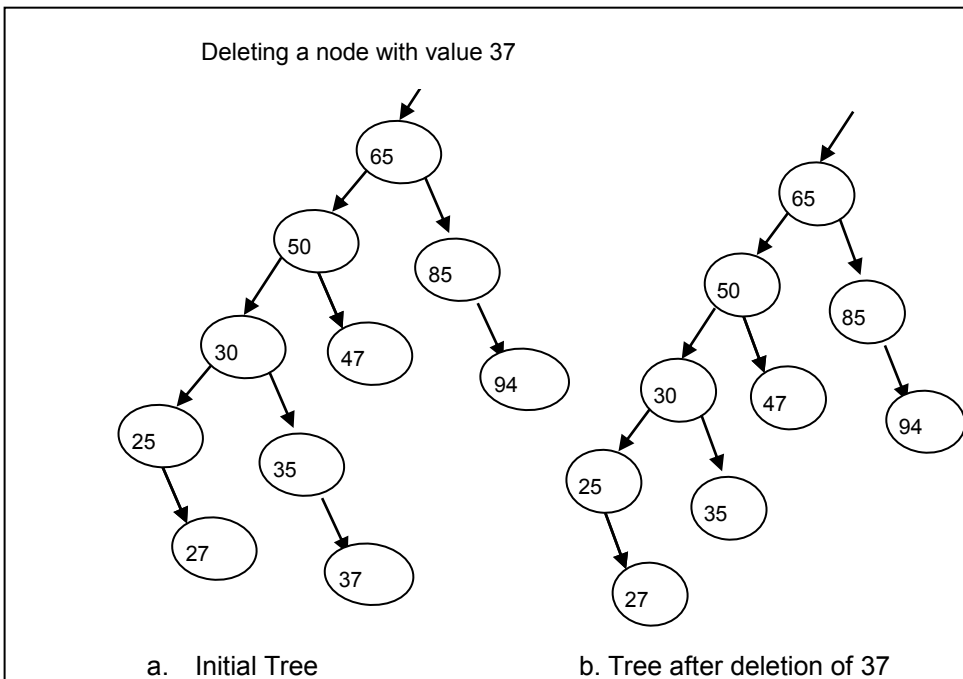


Figure 7-13: Deletion of a node with one child

determined. Replace the value of in-order successor node to the value of the node to be deleted and set this node as NULL. Complete program to delete the node using copying method is given in the following program.

[In our approach, to delete the target node, we will find out the in-order successor (instead of in-order predecessor) approach. The in-order successor node is the left most node of the right child of the target node. Function to determine the in-order successor has been depicted in the following program.

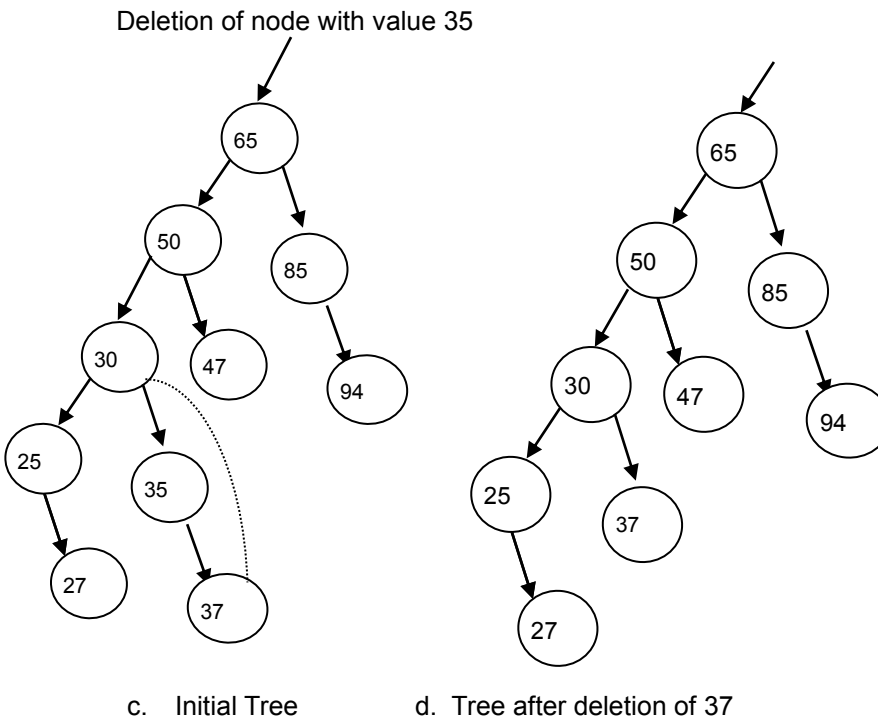


Figure 7-14: Deletion of a leaf node from a BST

```
// function to determine the in-order successor
bnode *bst::inSuccessor (bnode *current)
  {
  bnode *prev=NULL;
  bnode *insucc;
    current=current->right;
  while (current->left! =NULL)        // Go to the extreme left of
the BST
```

```
    {
    prev=current;
    current=current->left;
    }
    insucc=current;
    if (prev) // if prev is not NULL (prev exist)
     {
     prev->left=NULL;   //set the left child of the prev as NULL
     }
     return insucc;
}
```

Consider the BST illustrated in the figure 7.16. Node to be deleted (Target node) is 50, if we need to determine the in-order successor then we have to go to the right from where we have to head towards the extreme left end. In the given figure 32 is in-order successor.

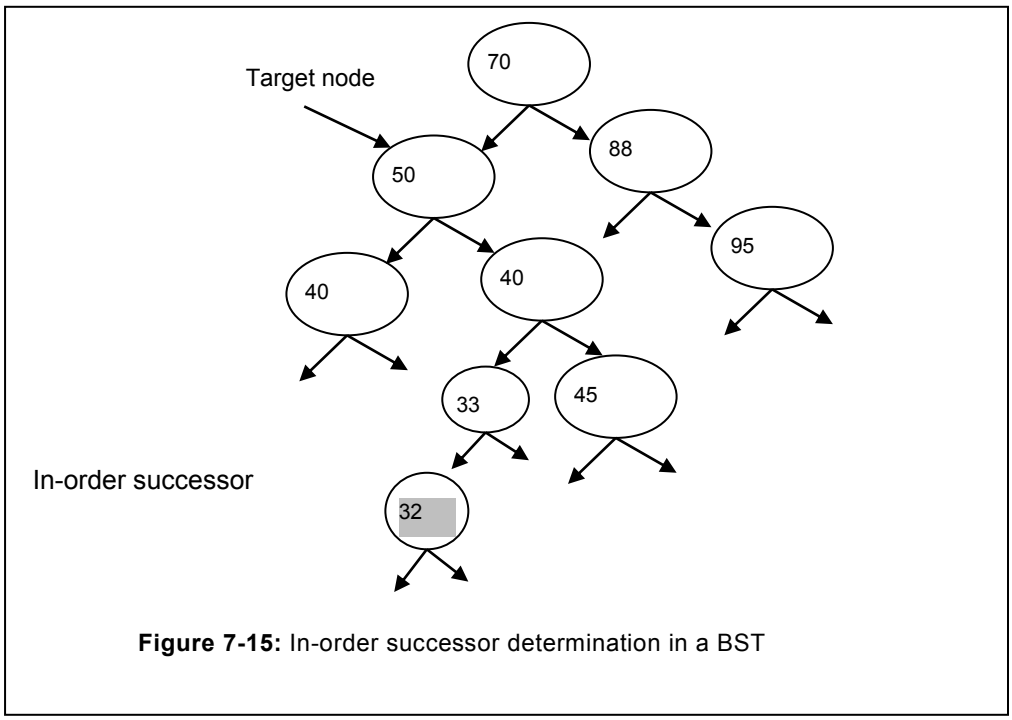


Figure 7-15: In-order successor determination in a BST

Correspondingly, consider the node to be deleted is 30, since it is having two children therefore; case ‘c’ will be applicable. First we have to determine the in-order successor. In the considered tree it will be 35, since the node 35 does not have any left child. Consequently, the node ‘35’ itself will act as an in-order successor.

Once we determine the in-order successor, it is obvious that it will not have any left node. Therefore, we will connect the left node of the target node to the left of in-order successor. Whereas right node will be connected to the in-order successor of this node.

7.5.2 Delete by merging

In delete the node by merging method the target node is deleted physically. Since, node is to be deleted will have the two children and if its parent is already contains one node then attaching these nodes to the parent is not possible. To address this problem, we have to find out the in-order predecessor or the in-order successor. Consider a case that we are aiming to determine the in-order successor, since we have reached to the left most node then we are sure that it will not have any left node. Connect the right node of the node to be deleted to the left side of the in-order successor that we have just attached. The above case has been depicted in the following program.

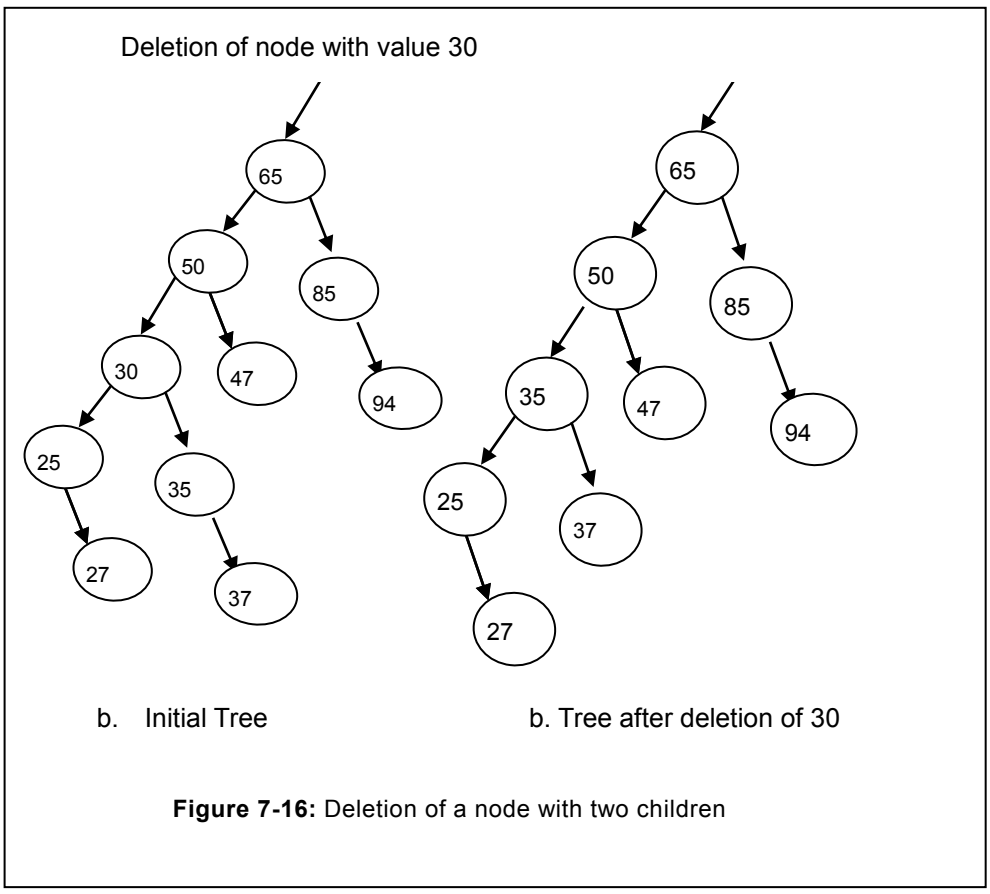


Figure 7-16: Deletion of a node with two children

```
void bst::deletenode (int node_data)
{
bnode*r,*parent=NULL,*temp;
r=root;
while(r! =NULL)
  {
  if(r->info==node_data)
  break;
  parent=r;
  if (node_data < r->info)
  r=r->left;
  else
  r=r->right;
  }// end of while
  if(r==NULL)
  {
  cout<<" Data is not in the BST"<<endl;
  cout<<"Press any key to continue...."<<endl;
  getch ();
  return;
  }
  else if ((r->left! =NULL) && (r->right! =NULL))
      {
       bnode *insucc=inSuccessor(r);
       cout<<"     we     are     evaluating     Inorder     successor=
......."<<insucc->info<<endl;
      getch ();
      deleteMerge (parent, r, insucc);
      }
 else if (parent->info < r->info) // if target(r) node is to the
towards right
   {
   if ((r->right==NULL) && (r->left! =NULL)) // if r has only left
child
     {
     temp=r;
```

```
     parent->right=r->left;
     delete temp;
     }
     else if ((r->right! =NULL) && (r->left==NULL)) // if r has only
right child
     {
     temp=r;
     parent->right=r->right;
     delete temp;
     }
     else if ((r->left==NULL) &&(r->right==NULL)) // if r is leaf
node
      {
      temp=r;
      parent->right=NULL;
      delete temp;
      }
   }
  else if (parent->info > r->info) // if target(r) node is to the
towards left
   {
   if ((r->right==NULL) && (r->left! =NULL)) // if r has only left
child
     {
     temp=r;
     parent->left=r->left;
     delete temp;
     }
     else if ((r->right! =NULL) && (r->left==NULL)) // if r has only
right child
     {
     temp=r;
     parent->left=r->right;
     delete temp;
     }
```

```
     else if ((r->left==NULL) &&(r->right==NULL)) // if r is leaf
node
      {
      temp=r;
      parent->left=NULL;
      delete temp;
      }
   }
    void bst::deleteMerge (bnode* parent, bnode *r, bnode *insucc)
  {
  bnode *temp=r;
  insucc->left=r->left;
  if (insucc==r->right)// if insucc is right child itself
  insucc->right=NULL;
  else
  insucc->right=r->right;
  if (parent->info < insucc->info)
  parent->right=insucc;
  else
  parent->left=insucc;
  delete temp;

}
```

7.6 SEARCHING AN ITEM IN A BST

Searching in a BST is used to determine the existence of a specific node (data) in the considered BST. To accomplish the searching, we will follow the technique of in-order traversal. As soon as the node to be deleted is determined, further searching will be terminated and the node consisting of the needed value is returned.

During the search process, if no node is remaining then we can conclude that the target node does not exist in the given tree. In this case NULL will be returned to the caller.

Program to depict the searching in a BST has been discussed as follows:

```
Program: Write a program to search a node that consists of a value
passed by the user.
tnode* bst::search (int el)
           {
```

```
tnode*rt;
tnode *found=NULL;
 rt=root;    //initialize it with root
while (rt! =NULL)  // continue search till node is null
{
if (rt->data==el)
{
break;
found=rt;
}
if (el < rt->data)
rt=rt->left;
else
rt=rt->right;
}
return found
}
```

7.7 COUNTING NUMBER OF NODES IN A BST

There are many instances in which we need to count the nodes of a BST. Total nodes of a BST consist of the sum of left half nodes and right half nodes of a tree. Method to count the number of nodes has been depicted in the upcoming sub-section.

7.7.1 Counting the left half nodes

To count all the nodes attached to the left side of a BST, we can make use of Queue. Whenever any node is encountered (should not be NULL), it is enqueue into the queue. Afterwards, we dequeue the node and check for the left node, if the left node exist, count is incremented by one. When the queue underflow occurs, we terminate counting and return the total left node count to the caller. In this program, we will use the dynamic queue that has already been created by us in the queue section. Complete program is depicted as follows:

```
int bst:: leftCount()
 {
 int count=0;     // initializing the count to 0
 bnode *x=root;   // initializing the x with the root
 queue<bnode*> q1;      //defining the queue of bnode type
 q1.enqueue(x); // inserting the node into queue
 while(!q1.isEmpty()) // Till queue is not empty
```

```
 {
 x=q1.dequeue(); // insert the node into the queue
 if(x->left!=NULL) // if left node exist
 {
  q1.enqueue(x->left); // insert the left node in q
  count++;          // increase the count
  }
  if(x->right!=NULL) // if right node exist
  q1.enqueue(x->right);  // insert the right node in q
  }
 return count;
}
```

7.7.2 Counting the right half nodes

Corresponding to the left side node, we can compute the total number of nodes attached to the right side of a BST. We can make use of Queue as shown in the above example. Whenever any node is encountered(should not be NULL), it is enqueue into the queue. Afterwards, we dequeue the node and check for the right node, If it is right node, count is incremented by one. When the queue underflow occurs, we terminate counting and return the total number of right count to the caller. Complete code to accomplish this task has been depicted as follows:

```
// Function to count the right hand side node in a BST
int bst:: rightCount()
 {
 int count=0;
 bnode *x=root;
 queue<bnode*> q1;
 q1.enqueue(x); // inserting the node into queue
 while(!q1.isEmpty()) // Till queue is not empty
  {
  x=q1.dequeue();
  if(x->left!=NULL) // if left node exist
  {
   q1.enqueue(x->left); // insert the left node in q
 }
   if(x->right!=NULL) // if right node exist
```

```
{
q1.enqueue(x->right);  // insert the right node in q
count++; // count the right node
}
   }
  return count;
 }
```

7.7.3 Counting all the nodes in a BST

To count the total number of nodes in a BST, we can compute the left half node as well as right half node of a BST. Afterwards, we can compute the total number of nodes by performing the sum of left half and right half. However, we have not counted the root node. Therefore, total node in a BST can be computed by

Total number of node=total number of left node + total number of right node +root node

Consequently, we can use the functions written above, to compute the total number of nodes.

- leftTotal= int bst:: leftCount();
- rightTotal=int bst:: rightCount();
- Total no. of nodes=leftTotal + rightTotal +1;

7.8 DETERMINING THE HEIGHT OF A TREE

Height of a tree is important to understand the exact depth of a tree. Efficiency of a tree is governed by the height o a tree. If a tree is containing so many branches which are not full in that case the height of a tree will grow tremendously where as the performance of the tree will reduce drastically. Therefore, we need to understand the height of a tree and how it can be controlled.

Height of a BST is the longest path existing in its left or the right side. For instance, the maximum path may be available towards the right side of the left node. Height of a BST has been explained with the help of following examples.

7.8.1 Non-recursive method

In the non-recursive approach of determining the height of a tree, we are visiting the tree level by level. Once new level is reached, height is incremented by one. To traverse the level by level, we can utilize the dynamic queue already created in the previous chapter. In this method, if the tree exists the root node is inserted into the queue. This is followed by counting the number of nodes in a queue. We can figure out the total number of nodes at a particular level with the help of a queue function ‘size ()’ that determine the number of nodes in a queue. Initially, number of nodes are 1 (only the root node). We can dequeue the root and

check the presence of its left or right node; if they are available they are inserted in the queue. Now the total number of nodes in a queue will be the nodes that are not null and available in the queue. We also increase the height once we access all the node of a tree. This entire process continues till all the nodes of a tree are not accessed. Finally, we return

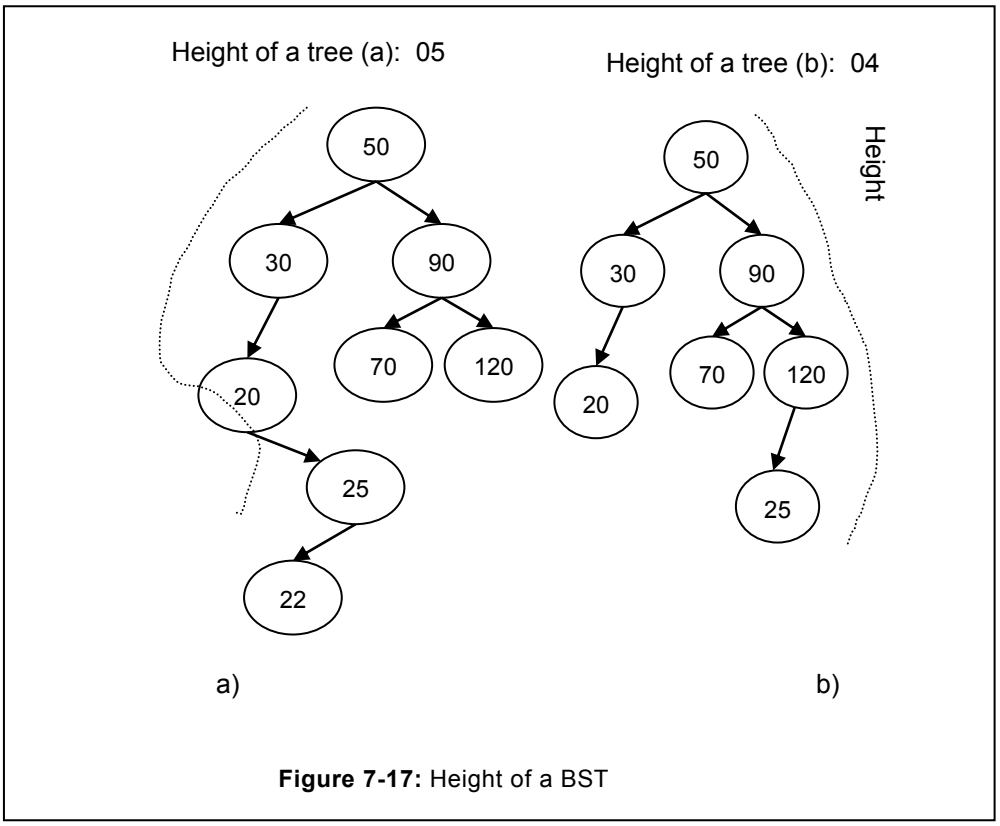


Figure 7-17: Height of a BST

the height of a tree to the caller function. Complete program to determine the height of a BST is given as follows

```
Program: Determine the height of tree (non recursive method)
// Non recursive function to determine the height of a tree
int bst::height_NR ()
 {
 int ht=0;  // initially height is 0
 queue<bnode*> q1; // create the queue of tree node type
 int count=0;     //total number of node in a queue
 bnode *temp;     // temporary variable to store bnode
 if (root! =NULL) // if tree exist
```

```
 q1.enqueue (root);     // insert the root in the queue
 while (! q1.isEmpty ())      // Till queue is not empty
 {
  ht++;     // increase the height
 count=q1.size ();      // count number of node in a queue
 while (count>0)  // while count is not zero (all the node at
  {               //particular level are taken out
  temp=q1.dequeue ();   // remove the bst node from the queue
  if (temp->left! =NULL)
  q1.enqueue (temp->left);
  if (temp->right! =NULL)
  q1.enqueue (temp->right);
  count--;  // reduce the count
  }
 }
return ht;  // finally return the height to the caller
}
```

7.8.2 Recursive method for height

In the recursive method of determining the height, we will compute the left height and the right height individually. Once both the heights of a tree are known we can compare them and return the maximum of left and right node to the caller. Following are the functions created using the object oriented approach.

```
Program: Program to determine the height of a tree recursively
// function called from the main function
int bst::height ()
{
bnode*r=root;
int h=height1(r);
return h;
}
// function called by the height without the knowledge of main
function
int bst::height1 (bnode * r)
{
   if (r==NULL)   // if root is null
```

```
        return 0;
    else
    {
        /* compute the depth of each sub tree */
           int lheight = height1(r->left);
        cout<<" after left call"<<lheight<<endl;
        int rheight = height1(r->right);
        cout<<" after right call"<<rheight<<endl;
        /* Find the larger one */
        if (rheight > lheight)
          return (rheight +1);
        else return (lheight+1);
    }
}
```

7.9 MIRROR OF A TREE

Once we stand in front of the mirror, we observe that our left hand appears to be as right hand and vice versa. Correspondingly, in the mirror of a tree, the given BST gets transposed. The left node will appears as the right node and the right node appears as left node.

Consider the BST of figure 7.19 (a) it represent the original tree. Mirror of the tree has been illustrated in the figure 7.19 (b). Mirroring can be achieved by connecting the right node to the left node and left node to the right side. To accomplish mirroring we can adopt the top to bottom approach or we can start from the bottom. Mirroring can be accomplished with the help of recursive or non-recursive function. We have commenced our discussion with non-recursive approach.

7.9.1 Non-recursive method

In the non-recursive method of mirroring, we will traverse level by level and transpose the child attached to right and left side. In our approach, we have inserted the node into the queue, we examine the queue in each iteration to determine whether queue is exhausted or not. If the queue is empty then it signifies that all the nodes have been traversed and the considered tree is transposed.

From the queue, we remove the node one by one which is followed by the exchange of its left and right child. After that we have to examine the availability of left and right node, if any one of them is available it is inserted into the queue. In our discussed approach, we have followed the top down approach, which means the mirroring has been started from the top and continued upto the bottom. Step by step method has been illustrated in the figure 7.19

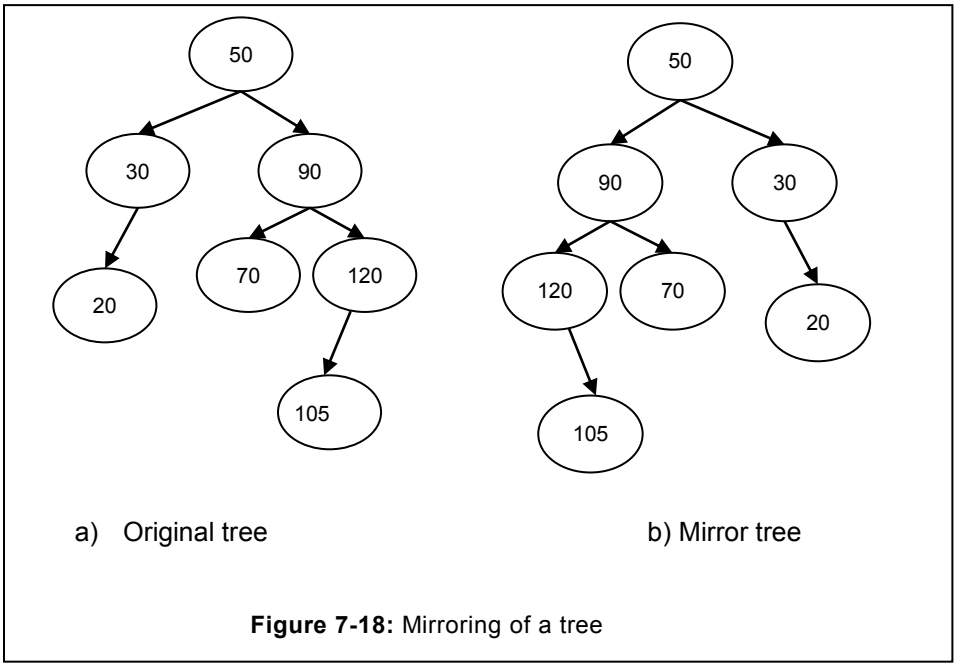


Figure 7-18: Mirroring of a tree

```
// function to find out the mirror of a BST
void bst::mirror ()
  {
  bnode *current,*temp, *x=root;
 queue<bnode*> q1;
 q1.enqueue(x); // inserting the node into queue
 while (! q1.isEmpty ()) // Till queue is not empty
  {
  current=q1.dequeue ();      // take out one node from a queue
  temp=current->left;         // swap left and right node
   current->left=current->right;
   current->right=temp;
  if (current->left! =NULL)// if left is not NULL insert in queue
  q1.enqueue (current->left); //insert left node in a queue
  if(current->right!=NULL)    // if right is not NULL
  q1.enqueue(current->right);  // insert right node in a queue
  }// end of while statement
 }    // end of function
```

7.9.2 Recursive Approach for Mirroring

In the recursive method, we follow the bottom up approach thereby nodes at the highest level (bottommost) is transposed first. This is followed by the nodes above it. Complete program to demonstrate the mirroring has been depicted below:

```
Problem: Recursive function for the mirroring
void bst:: mirror(bnode *root)
{
if(root==NULL)
return;
else
{
bnode *temp;
mirror(root->left);
mirror(root->right);
temp=root->left;
root->left=root->right;
root->right=temp;
}
}
```

To understand the functioning of mirroring using recursive function, consider the tree illustrated in the figure 7.19. Recursive function mirror(root->left) will enable us to reach the left most node of a tree. It will be followed by right side traversal. Since, it does not have the right node. Consequently, node with the value 20 will act as root. Since, it does not consist of any children therefore no changes will be visible.

It is followed by the node 30, since node with the value 30 doesn't have the right node, therefore, no rightward movement will take place. Now 30 is acting as the root node. Following statements such as temp=root->left will hold the address of the left node, since it does have the NULL node at the right. Consequently, 20 will be attached to the right and NULL will be attached to the left of the root node(30).

Afterwards, 50 will be returned, it is followed by one step right that is 90. Followed by reaching to the extreme left. In the given tree it is 70. Since 70 doesn't have the right and left child consequently, no change will be visible. Next node accessed will be 90, it is followed by shifting one step right i.e. node 120 and followed by extreme left. In the given case it is 105. Since, 105 doesn't have any child therefore, changes made by left and right attachment(swap) will not be visible. It is followed by the node just above i.e. 120. Node 105 will result as right node of 120.

Afterwards, node 90 will be accessed, the current state of the 90 is illustrated in the following figure 7.20. Its left side branch will be connected to the right side whereas the right side will result into left side due to the existence of the following statements.

```
temp=root->left;
root->left=root->right;
root->right=temp;
```

Consequently, the node will result as illustrated in figure 7.20.

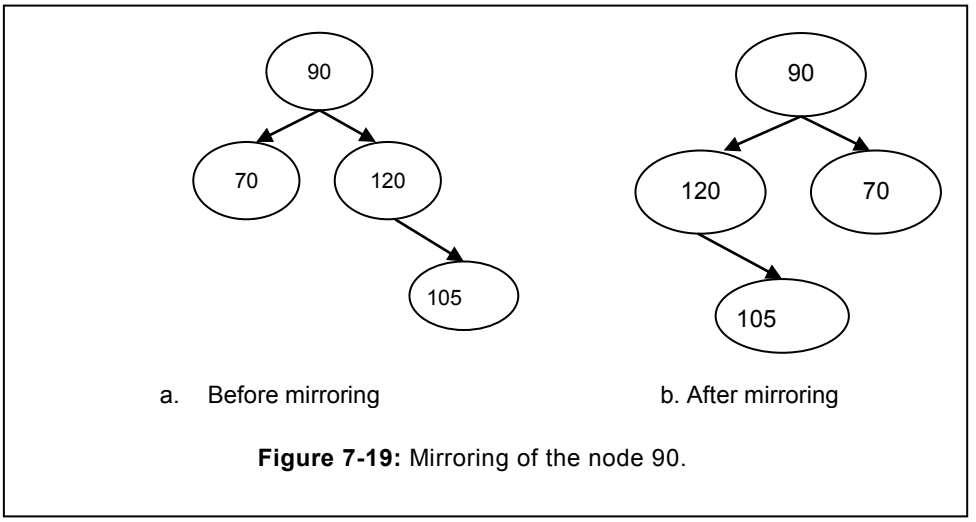


Figure 7-19: Mirroring of the node 90.

Eventually, 50 will be reached. Its left and right side nodes will be swapped as discussed above. Finally, the tree will be resulted as illustrated in the figure 7.20.

EXERCISE

A. Descriptive Type Questions

1. Create a class for a BST. In order to create the node at the time of object creation, modify the constructor suitably.
2. Create a function to visit each and every node using depth wise traversal.
3. Modify the function breadth first traversal. In the modified function, traverse level by level from bottom to top.

4. Write a program, where the display also appears as level by level. (Hint use two queue)
5. Write a program for the pre-order traversal that uses queue to insert the element. Once traversal is complete display each node stored in the queue.
6. Write a function that count and displays the number of nodes level by level in a BST.
7. Write a program that merges two BST. Ensure that after merging, the resultant tree should be BST.
8. Write a function to determine the following for any specific node of a BST

a) In-order predecessor

b) In-order successor

9. Write a program that should count the number of leaves node in a BST.
10. Write a program to determine the number of level in a BST.

Chapter 8

Advance Tree

Chapter Objective

- Defining the limitation of a BST
- Defining the threaded tree
- Defining the various types of threaded tree
- Defining the limitation of BST related to the height
- Defining the AVL tree
- Inserting the node in an AVL tree
- Discuss various types of rotation in an AVL tree
- Adjusting the balancing factor of an AVL tree.

8.1 ADVANCE TREE

BST that is discussed in the previous chapter suffers from various limitations which include inability of backward traversal, existence of many NULL nodes etc. Another major limitation of the BST is that it acts as a link list, if BST is to be created from the series that is in ascending or descending order. To overcome these limitations, advance tree such as threaded tree and AVL tree can be used. Threaded tree is useful in addressing the number of NULL nodes in a BST, at the same time facilitates the backward traversal. AVL addresses the BST behavior of skewing in one direction. Both of these topics have been described in the rest of the chapter.

8.2 THREADED TREE

Threaded tree is a special case of a BST, correspondingly, it requires additional component to represent thread. Variety of threaded trees are existing, these threaded tree can be used for their specific scenario. Major types of threaded tree that exist include:

- Left threaded tree
- Right threaded tree
- In-order threaded

BST in which only left pointer (having NULL) of a node is acting as a thread such tree is known as Left-in threaded tree. Similarly, BST in which only right pointer (having NULL) of a node is acting as thread, such BST is known as right-in threaded tree. Tree in which both left and right pointers of a node are acting as thread then it is known as in-threaded tree. Considering the complexity of in-threaded tree, we have described this (in-threaded)

threaded tree in the rest of our chapter.

8.2.1 Need of a threaded tree

It is apparent from our earlier discussion that the leaf doesn't have the child. Consequently, it's left and right pointer represents the NULL. Number of NULL pointers increases with the increase in level. Traversal, only in one direction is the other major limitation of a BST. To address all these issues, threaded tree is used. In the threaded tree, we connect the NULL pointers of the leaf as follows:

- Left pointer is connected to the in-order predecessor.
- Right pointer is connected to the in-order successor.
- If no predecessor or successor is present, in that case NULL pointer is connected to the root node.

Attachment of pointers to the predecessor and successor not only eliminates the NULL pointer, but also facilitates in backward traversal. In the rest of our discussion, we have considered the in-threaded tree. Once the readers can implement the in-threaded tree, implementing the left-in threaded tree or the right-in threaded tree can easily be mapped.

8.2.2 Creation of a threaded tree

In-threaded tree is popularly known as threaded tree. Structure of a threaded tree and the insertion of a node have been discussed in the following sub-section.

8.2.2.1 Representing the in-threaded tree

In-threaded tree is the advance version of a BST in which two additional components are also included. In addition to left and right pointer, provision for two threads i.e. left thread and right thread also managed. Complete structure of the in-threaded tree has been illustrated in the figure 8.1.

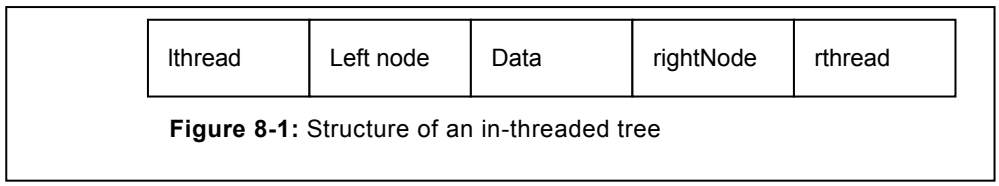


Figure 8-1: Structure of an in-threaded tree

In the above figure, symbols used have the following meaning:

lthread: denotes the left thread

leftnode: denotes the left node

info: data part of the tree

rightnode denotes the right node

rthread denotes the right thread

If the rthread=0, it denotes that the node has the right child and does not have right thread.

If the rthread=1, it denotes that the node has the right thread but does not have right child.

If lthread=0, it denotes that the node does not have the left thread as it is having the left child.

If the lthread=1, it denotes that the node does not have the left child consequently, it is having the left thread.

Class to represent the threaded tree node has been depicted as follows:

```
class tnode
{
public:
int lthread;
int rthread;
int info;
tnode *left;
tnode *right;
};
```

In threaded tree, there will be frequent need to determine the in-order successor and in-order predecessor. In-order successor is the next node to be visited during the in-order traversal. Correspondingly, the in-order predecessor in a threaded tree is the node that would be traversed just before the newnode during the in-order traversal of the BST. Functions to determine the in-order predecessor and in-order successor have been depicted as follows:

```
Program: To determine the in-order successor in a threaded tree
tnode* insucc( )
{
if(root->lthread==1)
return root;
else
{
tnode *succ=root->right;
while(succ->lthread!=1)
 {
succ=succ->left;
}
}
return succ;
}
```

```
Pre-order successor in a Threaded tree
tnode * presucc( )
{
if(p->lthread==0)
return p->left;
else
{
tnode *succ=p->right;
while(p->rthread==1)
p=p->right;
}
return p;
}
```

8.3 CLASS FOR THE NODE OF A THREADED TREE

Class to represent the structure of a threaded node as demonstrated in the figure 8.1, has been described in the upcoming program. Similar to Binary search tree (BST), members of the tnode (Thread node) should be public, otherwise these members will not be visible outside this class.

```
class tnode //class to create the thread tree node
{
public:
int info;
tnode *left,*right;
int lthread, rthread;   // to represent the left and right thread
tnode ()          //default constructor
{ }
tnode (int);
};
// Defining the constructor outside the class
tnode::tnode (int x)    //constructor with one argument
{
this->info=x;
lthread=rthread=1;
}
```

8.3.1 Threaded Tree member

In line with the threaded node, a class can be created that can represent the threaded tree. Class will consist of thread node and the function needed to conduct the various activities in a threaded tree. These activities include insertion of items in a threaded tree, traversal in a threaded tree, deletion of an item in a threaded tree, etc.

```
// Class to represent the threaded tree
class thread
{
tnode *root;      // defining the threaded node
public:
thread ();
void insert (int);      //   function   to   insert   the   element   in
threaded tree
void inorder ();
void in_order(tnode*);
void inorder_nr();// Non-recursive in order traversal
};
thread::thread()  //initializing the root to zero
{
root=NULL;
}
```

In the above class, root has been initialized with NULL. Consequently, it can represent that the threaded tree is empty.

8.4 INSERTING THE NODE(S) IN A THREADED TREE

Insertion of an item/node in a threaded tree is a complex procedure and precautions need to be exercised due to the following reasons:

- Identifying the location where node is to be inserted in a threaded tree.
- Adjusting the left and right side pointer of this new node.
- Adjusting the pointers and thread of a node acting as a parent node of a new node.
- Consider a case that only root node exists in the in-order threaded tree, in this case the left and right node will not have the node. Consequently, it's left and right thread will point to the NULL.
- In all other cases left thread is connected to the in-order predecessor and right thread is connected to the in-order successor. In the event of no predecessor or successor exist. In these cases, the thread will be connected to the root node.

Class to defining the various data members and functions needed for the threaded tree has been depicted as follows:

```
Problem: To create threaded tree and inserting the node
class threadtree
{
threadnode *root; // Root of a threaded tree
void insert (int);      // function to insert the node
void inorder ();  // function for the in-order traversal
};
void threatree::insert (int x) // inserting the node in a threaded
tree
{
if (root==NULL)
{
root=new threadnode;
root->data=x;
root->left=root;
root->right=root;
root->lthread=1;
root->rthread=1;
return;
}
r=root;
threadnode *newnode;
while (1)
{
if(x < r->info)
{
if(r->lthread! =0)
{
newnode->left=r->left;
r->lthread=0;
newnode->lthread=1;
newnode->right=r;
newnode->rthread=1;
```

```
break;
else
r=r->left;
}
else
{
if(r->rthread==1)
 {
newnode->left=r->left;
r->lthread=0;
newnode->lthread=1;
 newnode->right=r->right;
newnode->rthread=1;
r->rthread=0;
break;
}
else
r=r->right;
}     //end of else
}// end of while
```

8.5 TRAVERSAL IN A THREADED TREE

Traversal in threaded tree is a complex procedure. It requires variety of factors to be considered during traversal. Since, thread of a child may connects to the root therefore, cycle will be formed that will result in an infinite loop. Therefore, due care need to be exercised to prevent this infinite looping.

In the ordinary BST, during traversal we are only examine the presence of NULL where we terminate the traversal. Similarly, in the threaded tree, we continue to head till thread is not encountered. Inorder traversal in the inthreaded tree is depicted as follows:

```
void thread::inorder()
{
tnode *r=root;
in_order(r);
}
void thread::in order(tnode *root) // start of in order threaded
function
```

```
{
if(!root->lthread)
{
cout<<"display of lthread"<<endl;
in_order(root->left);
cout<<"info="<<root->info<<endl;
cout<<"left     and     right     thread"<<root->lthread<<"\t"<<root-
>rthread<<endl;
cout<<"Press any key to continue........."<<endl;
getch();
if(!root->rthread);
in_order(root->right);
}
cout<<"info="<<root->info<<endl;
cout<<"left     and     right     thread"<<root->lthread<<"\t"<<root-
>rthread<<endl;
cout<<"Press any key to continue........."<<endl;
}//end of function
void inorder(threadtree *root) // function for the inorder traversal
{
threadtree *r=root;
while(*r!=*root)
 {
cout<<r->info;
r=insucc(r);
}
}
```

8.6 NON RECURSIVE TRAVERSAL IN A THREADED TREE

```
// Program: Traversal in a Threaded tree
void thread::inorder_nr()
{
tnode *r1;
int count=0;
r1=root;
while(r1->lthread==0)
```

```
{
r1=r1->left;
}
while(count<2)//regulating the loop with
  {       //the help of root node visit
   cout<<"info ="<<r1->info<<endl;
   if(r1->rthread==1) //go one step right
   {
   r1=r1->right;
   if(r1==root)
   count++;
   }
   else
    {
    r1=r1->right;
   while(r1->lthread==0)
       { //if right child exist go the extreme left
       r1=r1->left;
       }//end of while
    } //end of else
    }//end of outer while
  }//end of function
```

8.7 CALLING FROM THE MAIN FUNCTION

```
int main ()
{
clrscr ();
thread t1;
t1.insert (20);
t1.insert (30);
t1.insert (25);
t1.insert (32);
t1.insert (15);
t1.inorder_nr ();
```

```
getch ();
return 0;
}
```

8.8 AVL TREE

BST proves to be effective in terms of performance relative to the linear link list. However, it suffers from many limitations. For instance, consider a case that the list is in the sorted order (items are already arranged in ascending or descending order), in such cases BST performs like a link list. This phenomenon has been illustrated with the help of figure 8.2.

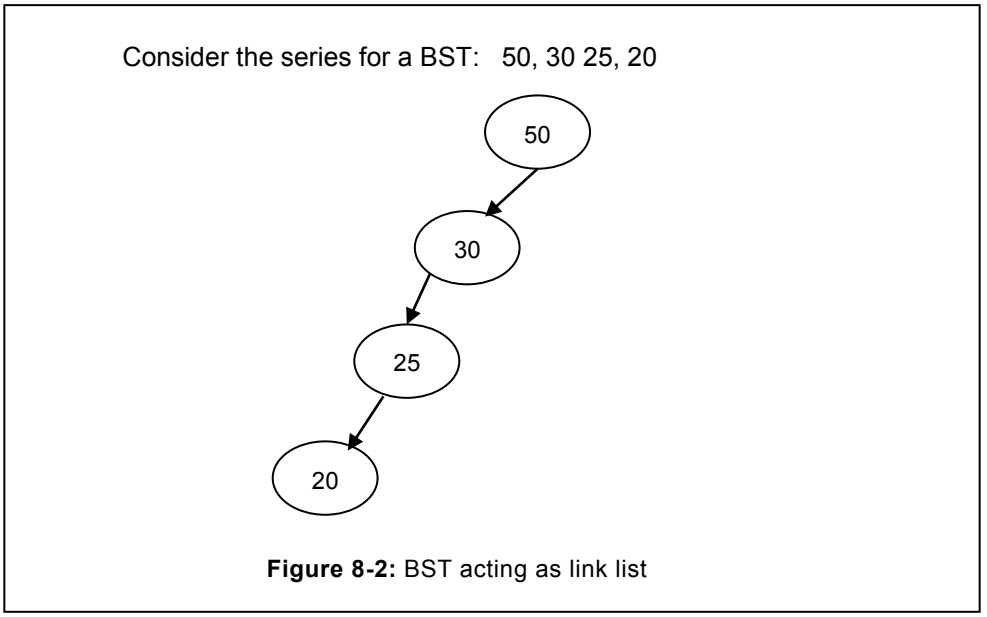


Figure 8-2: BST acting as link list

In the above figure, the series is in descending order; therefore BST is skewed towards left and acting like a link list. For the simplicity, we have considered the small series; impact will be increasingly higher in cases where the size of the series is large. Consequently, it loses the performance advantage of a tree.

8.8.1 Homogeneous unbalance tree

To address the above issue, modified version of BST known as AVL tree is used. AVL's name comes from its developer's G. M. Adel' son, Vel'ski and E. M. Landis. AVL is a special case of BST in which a new factor known as balancing factor is included. This balancing factor can receive the value between '-1' to '1' i.e. '-1', '0', or '1'. For instance, if the height of left side of a tree is one more than the one at right hand side than balancing factor is increased by +1. If the level of right side nodes is greater than one from the left side, -1 is added to it. However, if both the left and right side is at the same level, parent will have the 0 balancing factor. In brief, the balancing factor can receive the value as (1, 0,-1}, from left to

right. AVL tree along with the balancing factor has been illustrated in the following example.

In many books, the symbol {/, 0, \} are used to represent the left heavy, balanced and right heavy respectively. However, in this book we have used the numeric value discussed earlier. There may be various cases in which left or right node may be heavy. Correspondingly, AVL may be homogeneously or non-homogeneously balanced. Same has been described in detail in the upcoming section.

In the homogeneous unbalance AVL tree, it is unbalanced due to the unbalancing factor at the same side. For instance, consider the case of figure 8.2 (e & f), both are unbalanced AVL tree. The tree one illustrated in the figure 8.2 (e) is left heavy and the balance factor of the

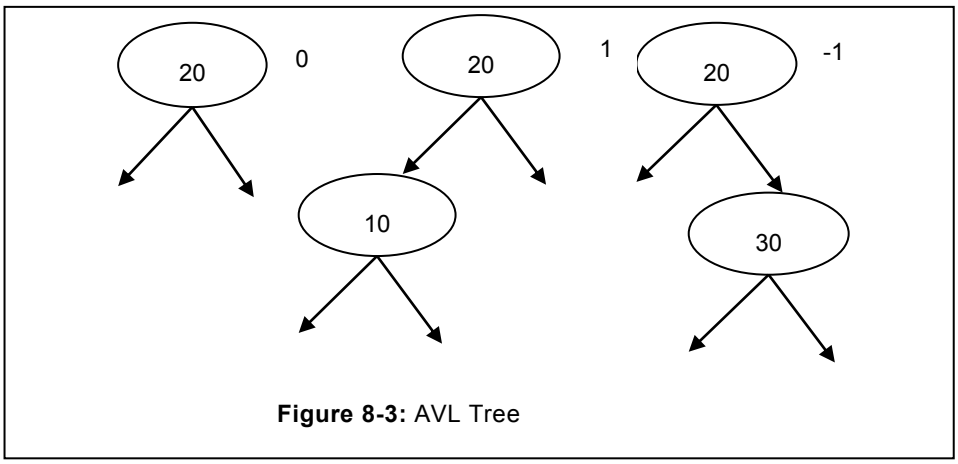
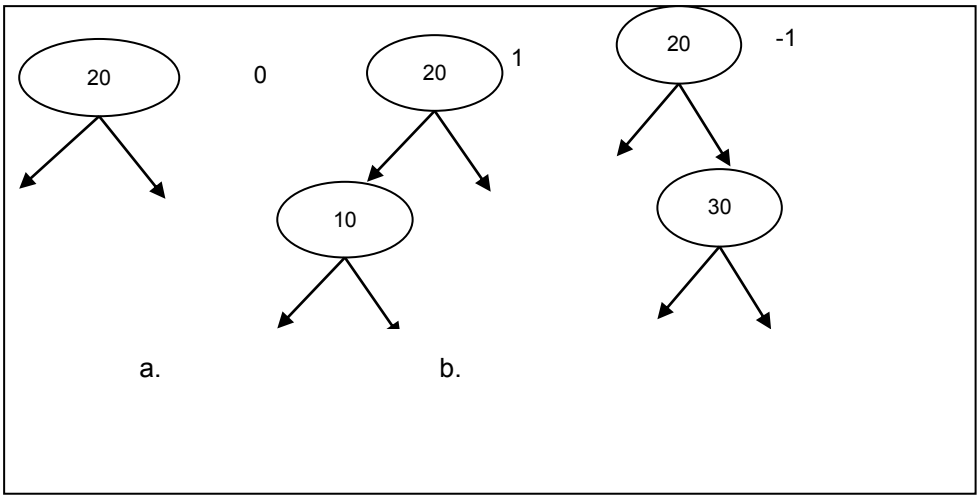


Figure 8-3: AVL Tree

grand parent of node 5 is 2. Similarly, upon computing the balance factor of its parent ‘5’ it is +1. Therefore, such type of unbalance tree is known as homogeneous unbalanced tree. Since the tree is unbalance to homogeneously attachment of the nodes. Correspondingly, we can define that the AVL tree shown in figure 8.2 (f) is homogeneous right unbalanced tree.



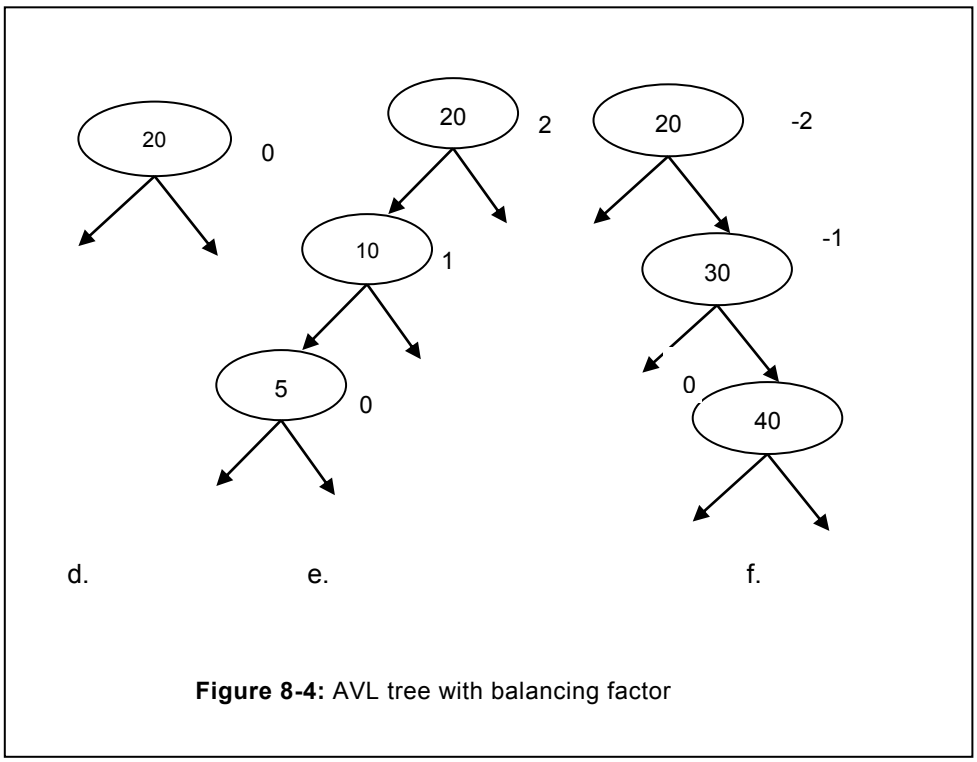


Figure 8-4: AVL tree with balancing factor

8.8.2 Hetrogeneous unbalance tree

In the heterogeneous unbalanced tree, unbalancing effect is attributed due to the heterogeneous imbalance. For instance, consider the case of figure 8.5 (a) which is left heavy, however its child is right heavy. Similarly, in the figure 8.5 (b) root is right heavy, but when we reaches to its child 30, it's left heavy. The above discussed cases of unbalanced tree are termed as heterogeneous unbalancing. To balance such type of AVL tree we have to follow the specific method to balance and the same has been discussed in the rotation section of this chapter

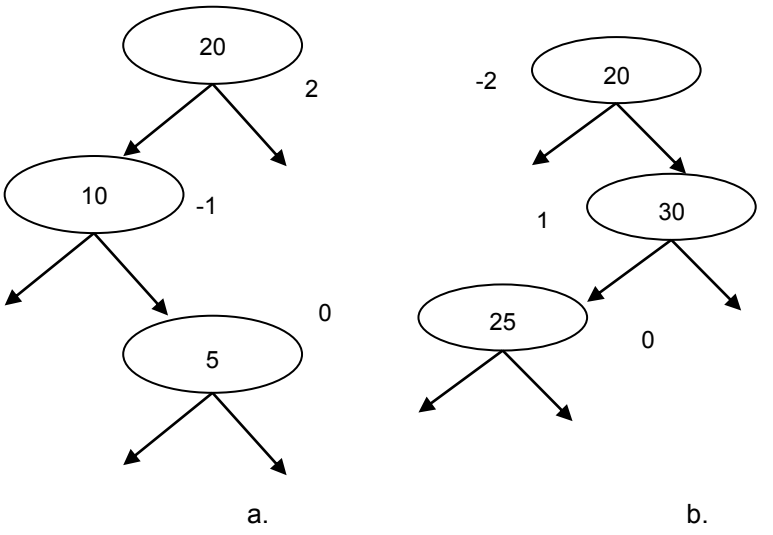


Figure 8-5: Heterogeneous unbalance AVL tree

8.8.3 Creation of an AVL tree

In order to ensure that BST is not acting as a linear link list, additional member, term as balancing factor is accommodated in the structure of AVL tree. We monitor the balance factor regularly, in order to ensure that the tree is not skewed in one direction. This balancing factor will change with the attachment of a node in an AVL tree. Data members of an AVL tree will be as:

```
class AVLnode
{
public:
// for the data part of the AVL Tree
```

```
int info;
// for the balancing factor
int bfactor;
AVLnode *left, *right;  // left and right node of the AVL
};
```

To understand, how the AVL tree will grow, let's understand it with the help of an example. Consider the series, 80, 25, 66, 90, 32, 60, 11 while inserting/creating 80 in the AVL tree, there are no node that are existing, therefore, root is to be created and the balance factor of this node will be 0. Consider another number 25, this node will be attached to the left side of the 80 and its balance factor will be 0. Whenever a new node is attached to the AVL, its balance factor is always equal to zero. However, the balance factor of its predecessor(s) needs to be changed. This balance factor will propagate upwards upto the grand parent, if exist. In case the balancing factor of the grandparent is violated, AVL need to be suitably rotated so that it can continue to qualify the definition of AVL tree. Creation of an AVL tree has been illustrated in the following figure 8.6.

Initially, 80 is inserted and its balance factor is changed to '0'. Since, it is the root node, consequently balancing cannot propagate upwards.

After 80, another number 25 is being inserted in figure 8.6 (b), since, it is less than 80, it will be directed towards the left hand side and get attached with 80. Balancing factor of this new node will be '0'. Balancing factor will be propagate upwards, Since only one node exist above 25, therefore its balancing factor can be determined as

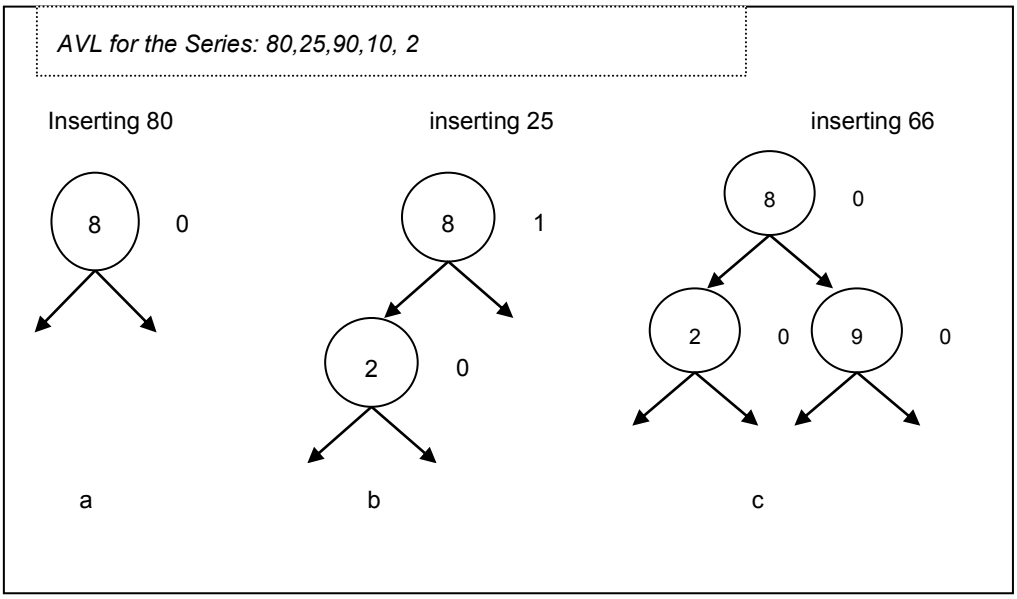
```
New balancing factor=old balancing factor + 1
New balancing factor=0 + 1=1
```

Therefore, new balancing factor of 80 will be 1.

Take another case (figure c), in which 90 is being inserted. This node will move towards the right side of the root. Balancing factor of this new node (90) will be 0. Upon insertion of this node, balancing factor will propagate towards updirection and its parent's balancing factor will be increased by 1. Since, node is attached to the right therefore increment will be -1. New balancing factor can be computed by:

```
New balancing factor =old balancing factor + (-1)
New balancing factor =1-1
                     =0
```

This propagation will not terminate here, instead will reach towards the left side of the root. New balancing factor of any node can be computed with the help of formula already given above. Therefore, it is apparent that balancing an AVL tree is a complex procedure and need promising algorithm inorder to balance the current node and subsequent nodes. However, once the AVL is created, it gives performance which is better than the BST, since the tree remain height balanced.



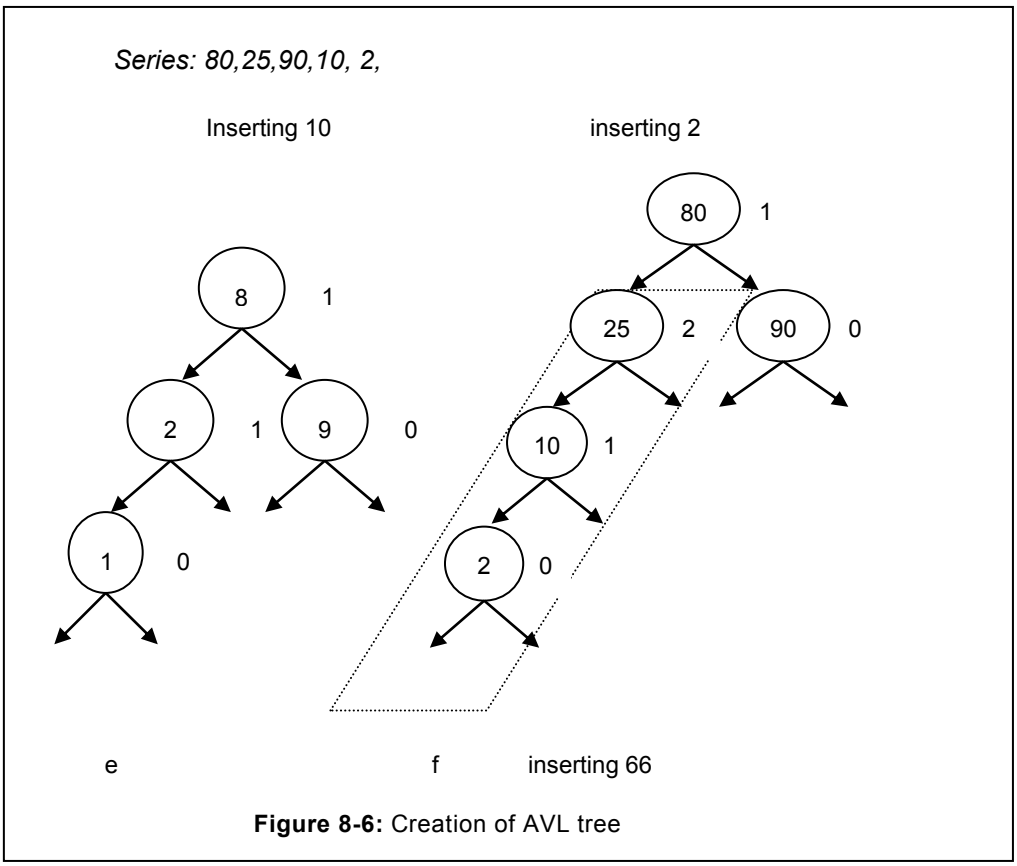


Figure 8-6: Creation of AVL tree

Finally, with the insertion of 10 and subsequently 2, balancing factor of the node 2's grandparent will be changed to 2 and this condition is unacceptable. Therefore, we need to balance this tree in order to ensure that it continue to qualify the definition of an AVL, we rotate the grandparent considering the homogeneous and heterogeneous unbalancing.

Class for the AVL tree and functions has been given as follows:

```
class avl
{
avlnode *root;
public:
avl ();
void insert (int);
avlnode*lrotate (avlnode*);
avlnode*rrotate (avlnode*);
void inorder ();
void in_order (avlnode*);
void leftbalance (stack<avlnode*>&, int);
void preorder ();
void pre_order (avlnode*);
void rightbalance (stack<avlnode*>&, int);
```

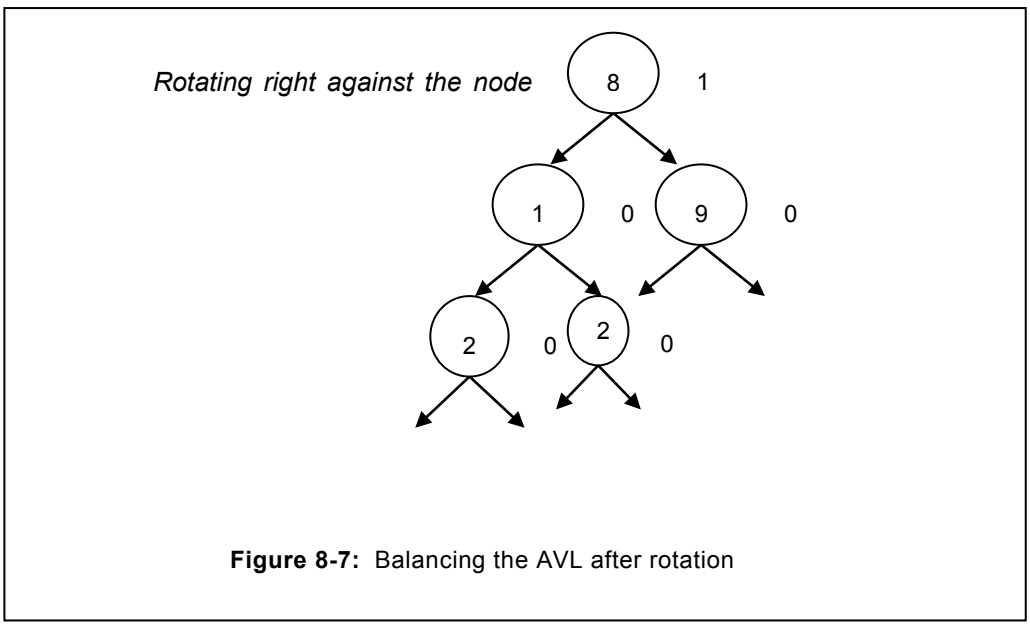


Figure 8-7: Balancing the AVL after rotation

```
//avlnode *leftrotate (avlnode*);
//avlnode *rightrotate (avlnode*);
avlnode *lrrotate (avlnode*);
avlnode *rlrotate (avlnode*);
};
```

8.8.4 Left Rotation

Left rotation is to be given, when the right side of the AVL tree is heavier. Consider the tree illustrated in the following figure. In the tree of figure 8.8, once 98 is inserted, as a result this tree becomes unbalanced. Node with the value of 90 will have new balancing factor which is equal to 2. Since, this is right heavy, therefore left rotation is to be given against the node 90. Consider in the figure 8.8 (a), 98 as child, 94 as parent of 98, and 90 as grandparent (gp) of 98. Rotating left against gp node can be accomplished by the following function:

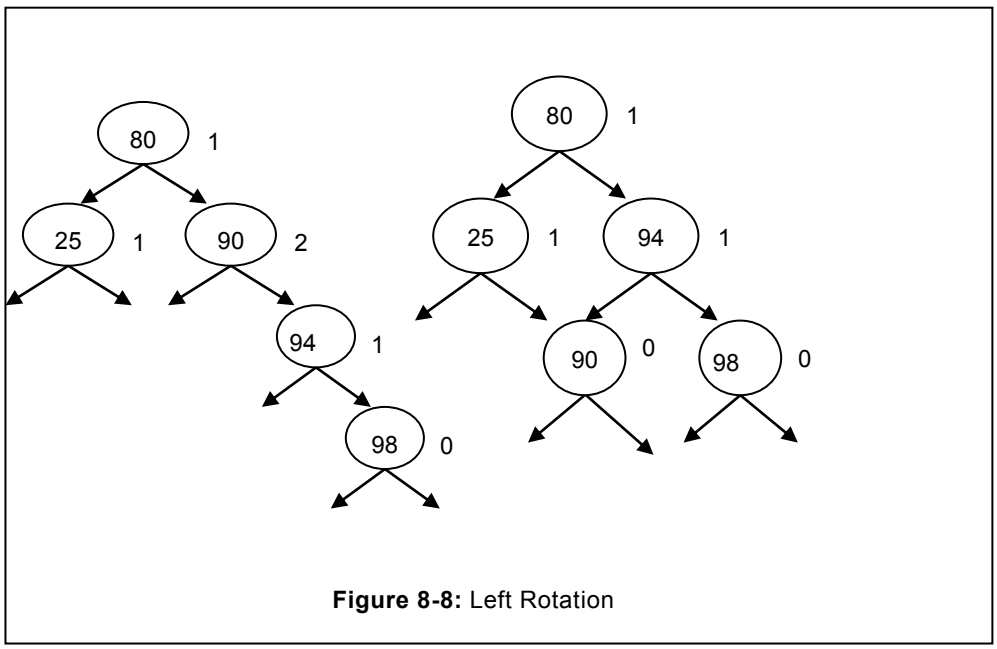


Figure 8-8: Left Rotation

```
avlnode* avl::lrotate (avlnode *gp)
  {
  avlnode *child;
  child=gp->right;
  gp->right=child->left;
  child->left=gp;
  child->bfactor=child->bfactor-1;
```

```
  gp->bfactor=gp->bfactor-1;
  return child;
//end of leftrotate
}
```

8.8.5 Right rotation

After inserting the node, if the left node is heavy then it is to be rotated right in order to ensure that the resultant tree should be balance.

```
avlnode *avl::rrotate (avlnode *gp)
  {
  avlnode *child;
  child=gp->left;
  gp->left=child->right;
  child->right=gp;
  child->bfactor=child->bfactor+1;
  gp->bfactor=gp->bfactor+1;
return child;
}
```

8.9 RIGHT LEFT ROTATION

In the right left rotation, first BST has to be rotated to the right. Completion of left rotation is followed by left rotation. This type of rotation is to be given in the cases of heterogeneous unbalanced tree. By heterogeneous, we mean if the right is heavy, but traversing downwards it is known that it is resulted due to the left heavy node. BST has already illustrated in the figure 8.5 (a and b) that is heterogeneously balanced.

```
/function for right left rotation
avlnode* avl::rlrotate (avlnode *gp)
{
avlnode*parent;
parent=gp->left;
gp->left=lrotate (parent);
gp=rrotate (gp);
return gp;
}
```

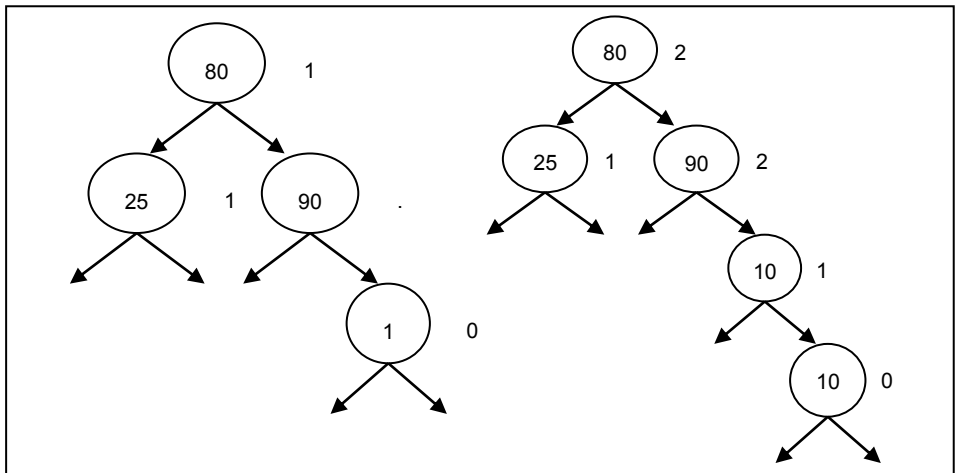


Figure 8-9: Right rotation

```
//Process of double rotation.
// function for left right rotation
avlnode* avl::lrrotate (avlnode *gp)
{
avlnode *parent;
parent=gp->right;
gp->right=rrotate (parent);
gp=lrotate (gp);
return gp;
}
```

8.9.1 Left Right Rotation

In the section 8.7.1, we have considered that both grandparent and parent are right heavy. However, this may be not the case always. It may also happen that grandparent is right heavy but parent may be left heavy. Balancing to the second case is complicated and requires double rotation. Initially, right rotation is given against the parent node. After that it is left rotated against the grandparent node. Same has been illustrated in the following example.

```
avlnode* avl::lrrotate (avlnode *gp)
{
avlnode *parent;
parent=gp->right;
```

```
gp->right=rrotate (parent);
gp=lrotate (gp);
 return gp;
}
```

If we consider the AVL tree given in the above figure 8.8, it has been observed that due to insertion of the node having the value 92 to the node 94. Its grandparent 90 has the balancing factor of 2 which is out of the tolerance limit. But when we considered the parent of 92, it has been observed that it is having the balance factor of -1. This indicates that the node is left heavy, so both are differently balanced AVL. In such cases, we have to give double rotation. Initially, the parent has to be given right rotation whereas the grandparent will be given left rotation after the first rotation only.

8.9.2 Inserting the elements in an AVL tree

Inserting the element(s) in the AVL tree is a complex procedure and requires a number of considerations. For instance, when the item is inserted in a balance tree, where the balance factor is '0', in such cases the element inserted will not result in the imbalance of the AVL.

In case if the balance factor is already 1, in that case a number of probability will exist. Consequently, rotation if needed would be govern by the rule of the rotations already specified in the section 8.7

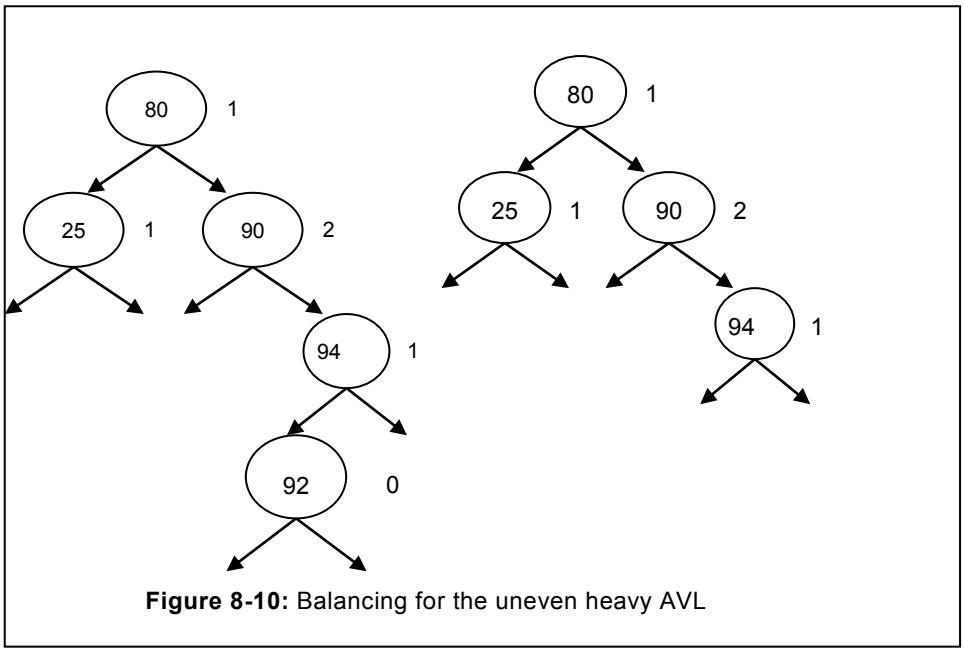


Figure 8-10: Balancing for the uneven heavy AVL

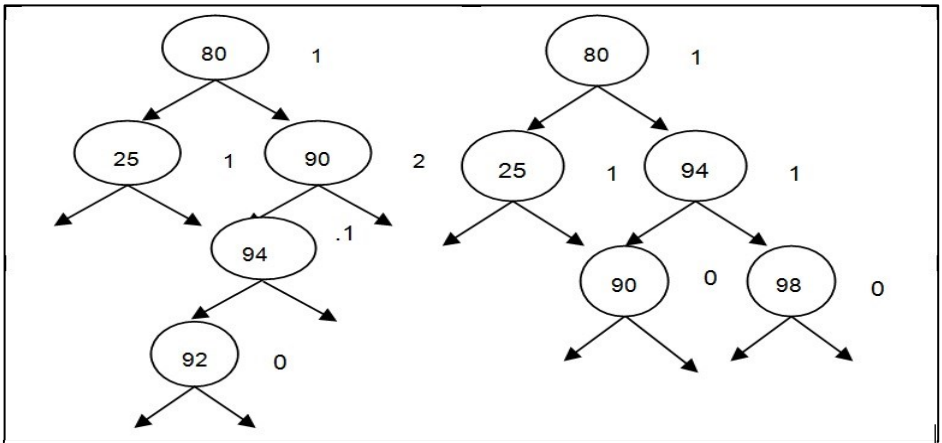


Figure 8-11: Balancing of an AVL tree

```
void avl::insert (int x)
{
if (root==NULL)
{
root=new avlnode(x);
return;
}
avlnode * temp,*root1,*parent;
root1=root;
stack<avlnode*> stack1;
while (root1! =NULL)
  {
stack1.push (root1);
if(x < root1->info)
root1=root1->left;
else
root1=root1->right;
} //end of while loop
parent=stack1.pop ();
if(x< parent->info)
```

```
 {
parent->left=new avlnode(x);
parent->bfactor=parent->bfactor-1;
if (! stack1.isEmpty ())
leftbalance (stack1, x);
}
else
{
parent->right=new avlnode(x);
parent->bfactor=parent->bfactor+1;
if (! stack1.isEmpty ())
rightbalance (stack1, x);
}
}
```

8.9.3 Left-Balancing of an AVL

Program for the left rotation of an AVL tree has been depicted as follows.

```
void avl::leftbalance (stack<avlnode*> &stack1, int x)
{
avlnode *gp;
int rotated=false;
while ((! rotated) && (! stack1.isEmpty ()))
 {
gp=stack1.pop ();
switch (gp->bfactor)
{
case -1: gp=rrotate (gp); rotated=true; break;
case 1: gp=lrrotate (gp); rotated=true; break;
case 0: gp->bfactor=gp->bfactor-1; break;
} //end of switch
}//end of while
if (stack1.isEmpty ())
root=gp;
else
{
```

```
avlnode *temp=stack1.pop ();
if (gp->info<temp->info)
temp->left=gp;
else
temp->right=gp;
}//end of else
} //end of function leftbalance
```

8.9.4 Right-Balancing of an AVL

Program for the right rotation of an AVL tree has been depicted as follows.

```
// Program to write the right balance
void avl::rightbalance (stack<avlnode*>&stack1, int x)
{
avlnode *gp;
int rotated=false;
while ((! rotated) && (! stack1.isEmpty ()))
 {
gp=stack1.pop ();
switch (gp->bfactor)
{
case -1: gp=rlrotate (gp); rotated=1; break;
case 1: gp=lrotate (gp); rotated=1; break;
case 0: gp->bfactor=gp->bfactor+1; break;
} //end of switch
}//end of while
if (stack1.isEmpty ())
{
root=gp;
}
else
{
avlnode *temp=stack1.pop ();
if (gp->info<temp->info)
temp->left=gp;
else
```

```
temp->right=gp;
}
}
```

8.9.5 Traversal in an AVL tree

Traversal in an AVL tree is not different with the one in BST. Indeed, it is more efficient to traverse in the AVL since it is almost balance tree where the depth variation of only 1 may exist. Popular traversal techniques including In-order, pre-order, and post-order traversal applicable in the BST are equally applicable in the AVL tree. Readers are requested to please refer the BST portion for explanation of this section.

```
void avl::preorder ()
{
avlnode *root1=root;
pre_order (root1);
}
void avl::pre_order (avlnode *root1)
{
if (root1! =NULL)
 {
 cout<<"info="<<root1->info<<endl;
 pre_order (root1->left);
 pre_order (root1->right);
 }
 }
 void avl::in_order (avlnode *root)
   {
   if (root! =NULL)
     {
     in_order (root->left);
     cout<<"info ="<<root->info<<"\t";
     cout<<"balance factor="<<root->bfactor<<endl;
     in_order (root->right);
      }
   }
   void avl::inorder ()
    {
```

```
    avlnode *root1=root;
    in_order (root1);
    }
```

8.9.6 Call from the main function

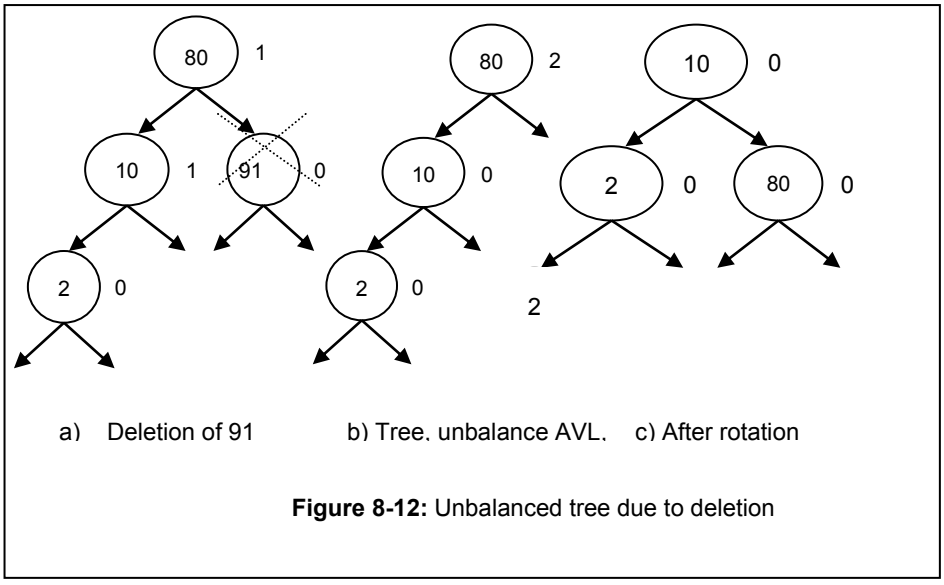
```
// Calling the aforementioned functions from the main functions in order it is needed
int main ()
    {
    clrscr ();
    avl avl1;
    avl1.insert (40);
    avl1.insert (50);
   avl1.insert (45);
   avl1.insert (60);
   avl1.insert (80);
   avl1.insert (90);
  // avl1.insert (2);
  // display of data from the tree.
   cout<<"inorder traversal "<<endl;
    avl1.inorder ();
    cout<<"Pre-order traversal "<<endl;
    avl1.preorder ();
    getch ();
    return 0;
    }
```

8.9.7 Deleting a node in an AVL tree

Deleting the node from the AVL tree carries the similar amount of complexity as with the insertion of a node. Once the node is deleted, AVL need to be evaluated for the balance factor, since deletion of the node may lead to the unbalancing of an AVL tree. As soon as, it is learned that AVL is unbalanced, it is to be suitably rotated so that the balance factor should not cross the permissible limit.

Consider a case as depicted in the figure 8.13, after the deletion of the node, balancing factor is not crossing the permissible limit, consequently no rotation is needed. However, in the case of figure 8.12 ‘a’, deleting a node leads to change in balancing factor. On evaluating the balancing factor, it is learned that it has crossed the permissible limit of balancing factor;

therefore, this node is to be rotated. New balancing factors of various nodes of an AVL has been illustrated in the figure 8.12. In the 8.12, the node 91 has been deleted, as a result, the



a) Deletion of 91 b) Tree, unbalance AVL, c) After rotation

Figure 8-12: Unbalanced tree due to deletion

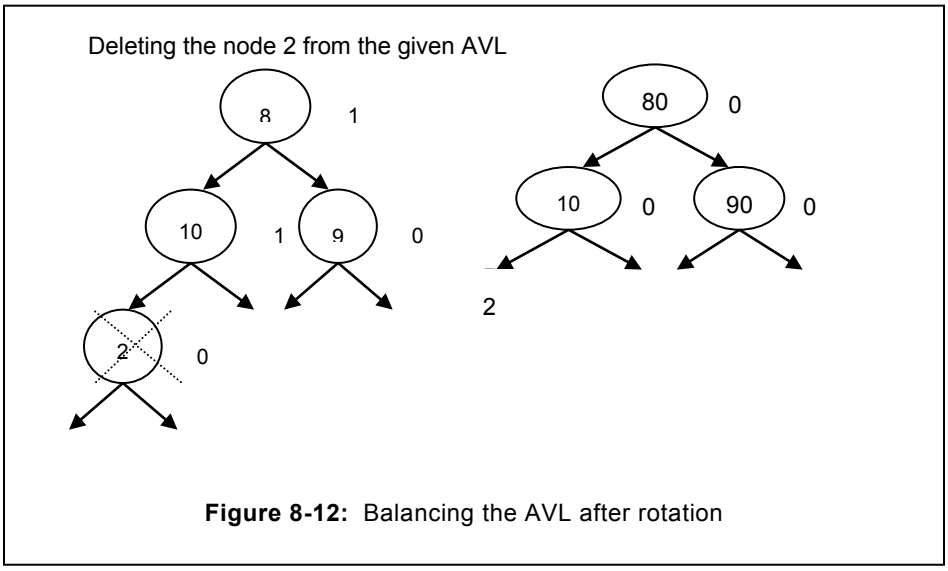


Figure 8-12: Balancing the AVL after rotation

node 80 will become unbalance and its new balancing factor will be 2. Since, it is left heavy; therefore, right rotation is to be given. Eventually, the node will be balanced after giving the right rotation, as illustrated in the figure.

EXERCISE

A. Descriptive Type Questions

1. What is an AVL tree? What is the significance of balancing factors in an AVL tree?
2. How AVL tree is different from BST? Jusity your answer.
3. What are the advantages that can be leveraged, if the user is applying the AVL data structure in his application? Justify.
4. What are the different types of rotations applicable in an AVL tree? Explain-
5. Discuss in detail when the left-right. (LR rotation is needed in an AVL tree?
6. How the traversal in an AVL tree is different from that of BST?
7. Discuss the various conditions applicable during the deletion of the node from an AVL tree.
8. What are the various limitations of an AVL tree?
9. Compute the complexity of inserting a node in an AVL tree.
10. Discuss the cases during insertion when no rotation occurs.
11. Discuss the number of levels above it will be affected once the AVL tree becomes unbalance during the insertion of the elements.

B. Short Answer Type Questions

(Follow the instruction to solve the given questions)

i. AVL tree is also a BST [True/False].

ii. AVL tree yields better performance relative to BST [True/False].

iii. AVL Tree is slow in relative to BST during:

a. Traversal c. insertion time

b. Display d. None of these

iv. AVL yields good performance during:

a. Searching c. Deletion

b. Insertion d. All the above

v. It is possible to give left-left-right rotation in an AVL tree [True/ False].

vi. It is possible to give left-right-right rotation in an AVL tree [True/ False].

vii. It is possible to give right-left-right rotation in an AVL tree [True/ False].

viii. Traversal in an AVL tree is similar to that of BST [True/ False].

ix. Rotation is needed during traversal of an AVL Tree [True/False].

x. Rotation may be given, if needed, during the following operation.

a. Traversal c. both a and b

b. Insertion d. none of these

xi. In an AVL tree, balancing factor may propagate upto

a. Up one level c. Up three level

b. Up two level d. None of these

xii. In an AVL tree, on inserting of a node, root is having the balance factor of 1, whereas its right child is having the balance factor of -1. Suggest the rotation needed so that the tree can qualify the definition of an AVL tree.

a. Left rotation b. right rotation

c. Left right rotation d. right left rotation

Chapter 9

Multi way tree

Chapter Objective

- Defining the multi-way tree
- Describing the various application of multi-way tree
- Defining and describing the B tree
- Insertion and deletion in a B tree
- Describing the B* tree and B+ tree
- Describing the various application of B tree.

9.1 MULTI-WAY TREE

Binary trees are performing profoundly well in cases where the depth of a tree is not substantial. However, as the depth of a binary tree grows, performance begins to deteriorates. To address this issue, multi-way tree can be utilized. In a multi-way tree, a node can accommodate greater number of elements relative to the one in BST. This number may reach upto 'n'. Similar to AVL tree, multi-way tree always remains height balanced, this is considerable advantage over other types of tree. Due to height balance property, it yields better performance relative to the one offered by the binary search tree.

9.1.1 Significance of Multi-way tree

There are various implementation of a multi-way tree, for instance, B tree, B+ tree and B* trees etc. All these trees are having their own pros and cons. Consequently, it depends upon their implementers to opt the multi-way tree that is well suiting for their specific need. Each of these trees have been discussed in the upcoming section.

9.2 B TREE

B Tree is an example of a multi-way tree. This data structure is widely used for storing the data into the storage device, for instance hard disk. Due to its height balanced property, B-Tree organizes the data at almost the same level. This leads to great performance improvement. at the same time depth remains controlled.

A 'B' tree of order 'n' will hold the following rules:

- In a B-tree, keys are arranged in sorted order
- Root node will have at least two nodes, if it is not acting as the leaf node.
- Each non-leaf node will have at least n/2 child at each level. However, it is not applicable in the case of root node.
- Total number of keys in a node should be less than 'n'. For instance, if a tree is of 'n' order, total number of keys should be less than n. The B tree of order n can have maximum n-1 keys.
- All the keys in the left nodes are less than the parent keys whereas the keys to the right are more than the parent key.
- All leaf nodes are at the same level.
- All the nodes consist of at-least (n/2) -1 keys, except the root node that can have one key.

It is suggested that the number 'n' taken for the B tree should be odd.

Structure of a B tree is depicted as follows:

```
// Defining the structure of B tree
const int n=5;    // order of a B tree
// naming the node of bst
class bnode
{
// total number of keys=n-1
int keys [n-1];
// defining total number of nodes =n
bnode *next[n];
};
```

9.2.1 Creation of a B Tree

To understand the B tree, consider the following series.

78, 5, 56, 90, 12, 25, 10, 45, 98, 100, 88,

Consider that we have a B Tree of order 5 i.e n=5; therefore total number of keys which it can have is 4, since formula (n-1) is applied. Initially, 78, 5, 56, 90 can be inserted as illustrated in the figure 9.1(a). However, on the arrival of 12, condition (maximum four key) will be violated. Therefore, from all the keys avalilable i.e. [5, 12, 56, 78, 90] branching will occur and 56 will be acting as root node and the data less than 56 will be placed to the left node whereas the keys greater than 56 will be placed towards the right hand side. Upon arrival of 56, a new node will be created if not existing. If the node already exist. 56 will be placed at appropriate place according to its weight and the previous value available in the B tree. Resultant tree, without any previous node has been illustrated in the figure 9.1(a).

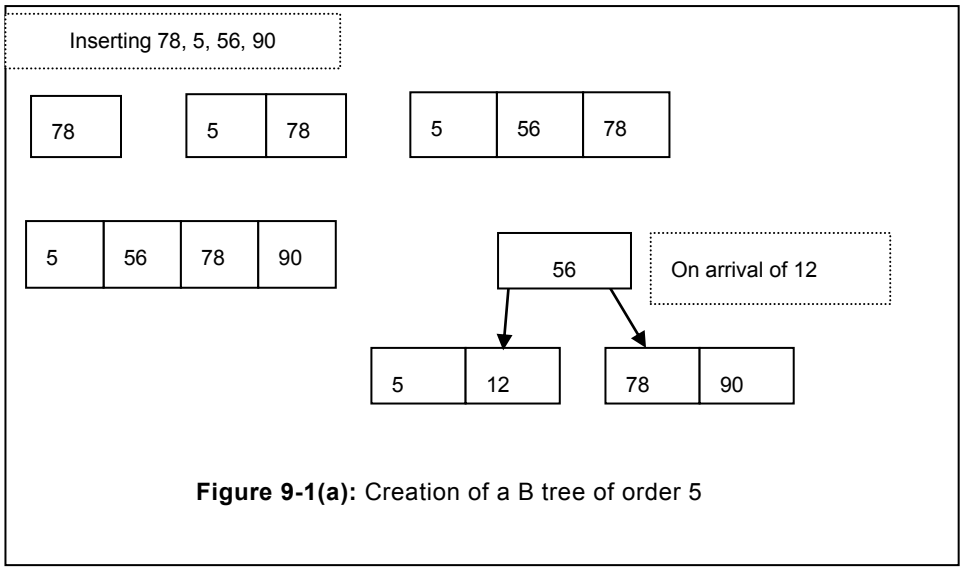


Figure 9-1(a): Creation of a B tree of order 5

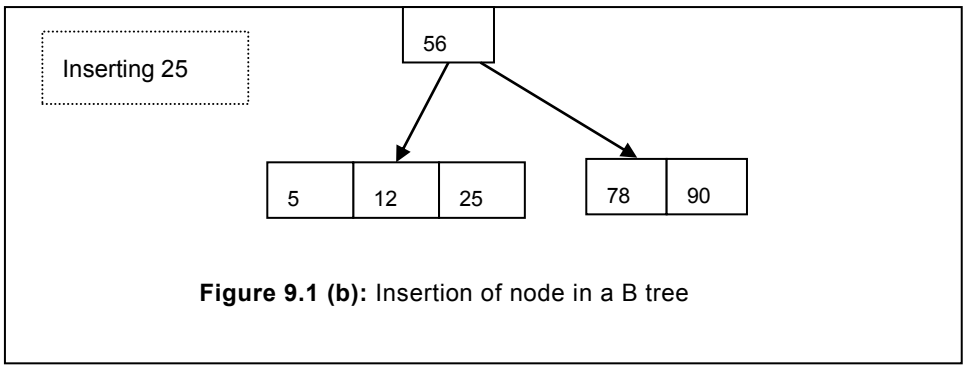


Figure 9.1 (b): Insertion of node in a B tree

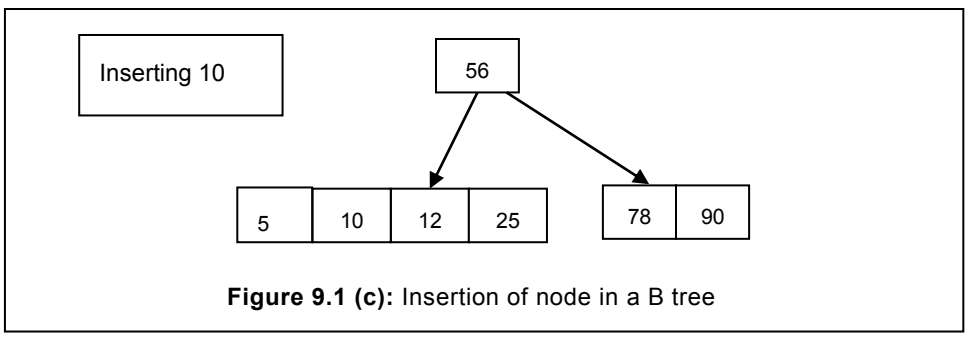


Figure 9.1 (c): Insertion of node in a B tree

After that we would like to insert 25 that will head towards the left of 56 and will result in the B-tree as illustrated in the figure 9.1(b). On the arrival of the item 10, it will head towards left of 56. Resultant B-tree is illustrated in figure 9.1(c).

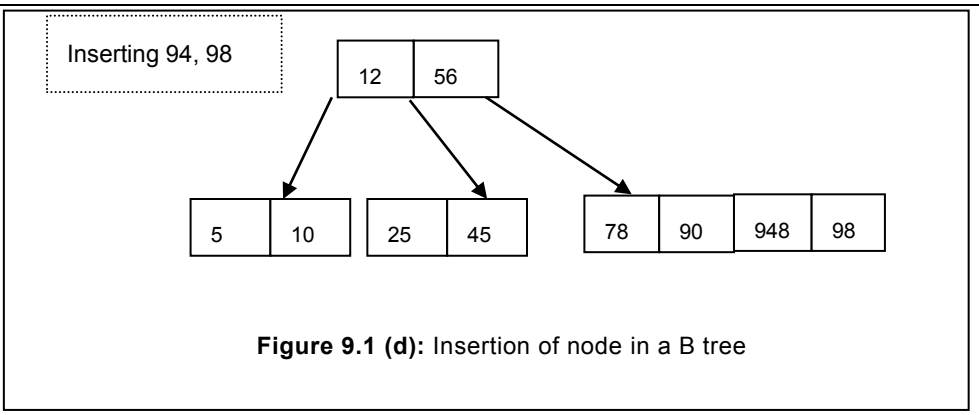


Figure 9.1 (d): Insertion of node in a B tree

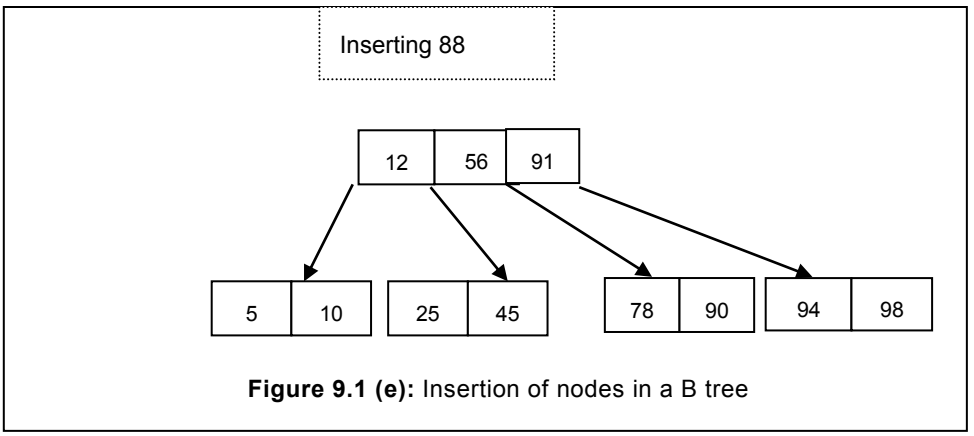


Figure 9.1 (e): Insertion of nodes in a B tree

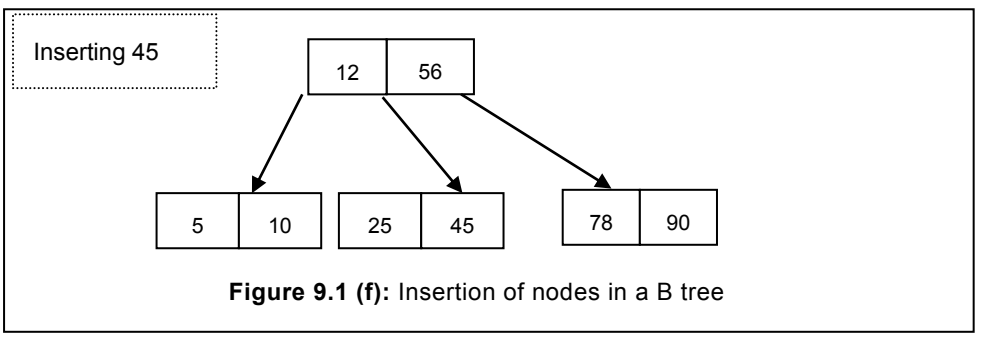


Figure 9.1 (f): Insertion of nodes in a B tree

On the arrival of 45, it will be directed towards the left branch. Left child is already full to its capacity; consequently, it will split the node in two branches. Resultant tree is illustrated in the figure 9.1(d).

In the above, consider a case; if number of keys in the parent is four in that case it won't be able to include all the five keys. As a result, it will split. Finally, the parent will also split to accommodate the key.

Finally, the B tree is illustrated in the figure 9.1(e). Sometime split also propagates to the upper level. Consider the B tree illustrated in the figure 9.2. Due to the insertion of 33, the second branch will split; consequently, the mid element i.e. 33 will go to the upper node.

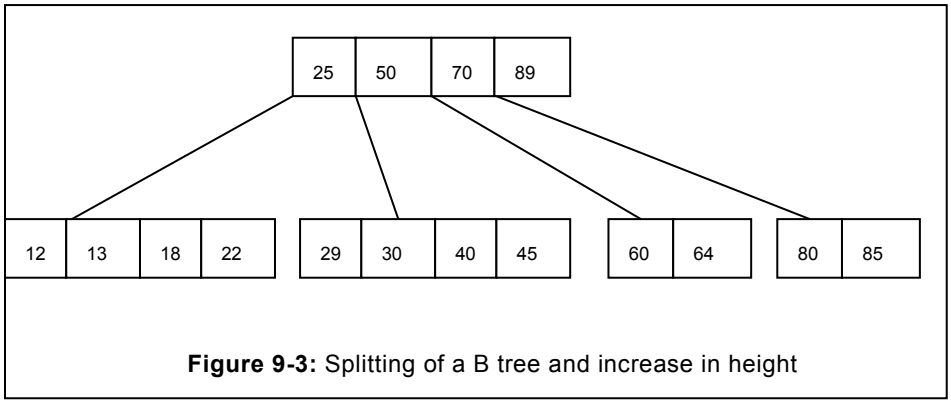


Figure 9-3: Splitting of a B tree and increase in height

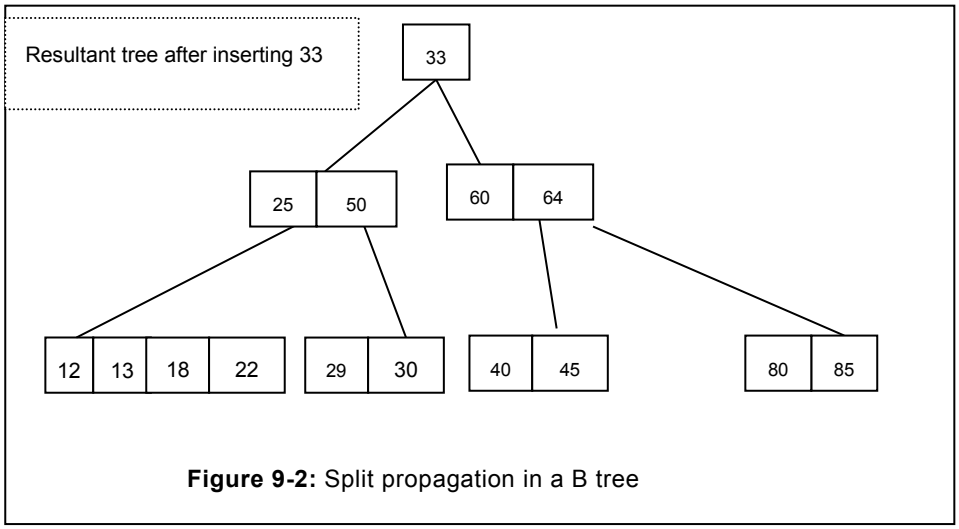


Figure 9-2: Split propagation in a B tree

However, the upper node is also full to its capacity. As a result, another split will occur on this level. Resultant tree is illustrated in the figure 9.3.

C. Deletion in a B tree

Deletion in a B tree is profoundly complex relative to the creation of a B tree. During insertion key is inserted at leaf, correspondingly, key has to be removed from the leaf. Complexity in the B tree lies in the merging when the underflow occurs in a B tree.

During deletion in a B tree, following four cases may occur:

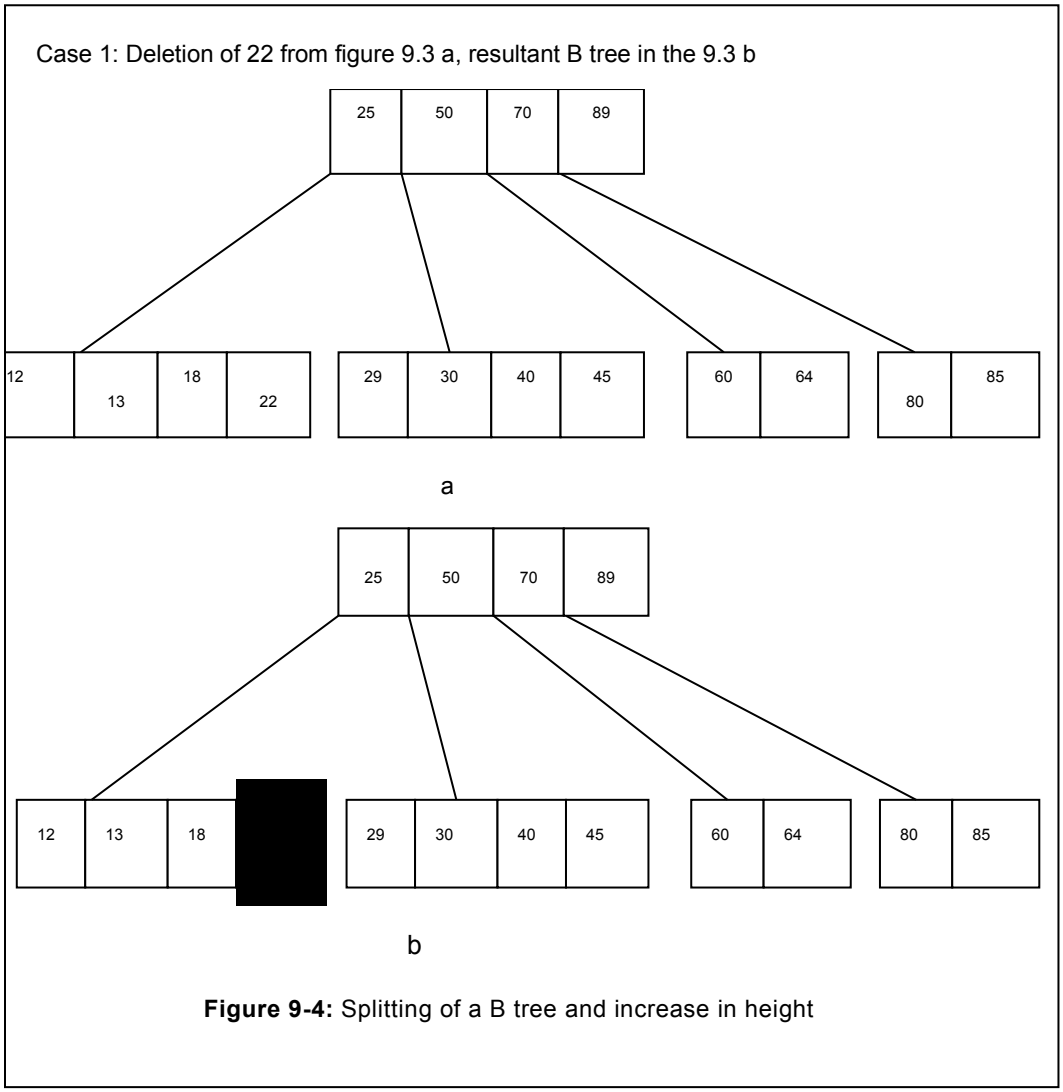


Figure 9-4: Splitting of a B tree and increase in height

- During deletion of a key, if the numbers of keys remaining are not violating any condition (n/2-1) then B tree is allowed to be as it is.
- If the key is not in the leaf node in that case it is guaranteed that its predecessor or successor will be in a leaf node in that case we promote the predecessor or successor to the leaf node.

Case 2: Deletion of 25 from figure 9.4 a, resultant B tree in the 9.4 b

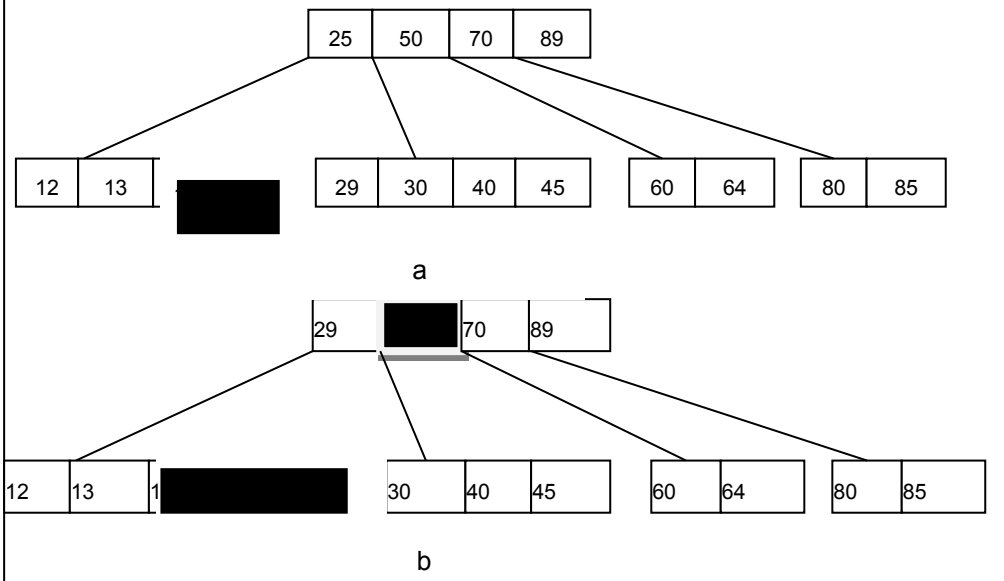


Figure 9-6: Splitting of a B tree and increase in height

Case 3: Deletion of 12 from figure 9.5 a, resultant B tree in the 9.5 b

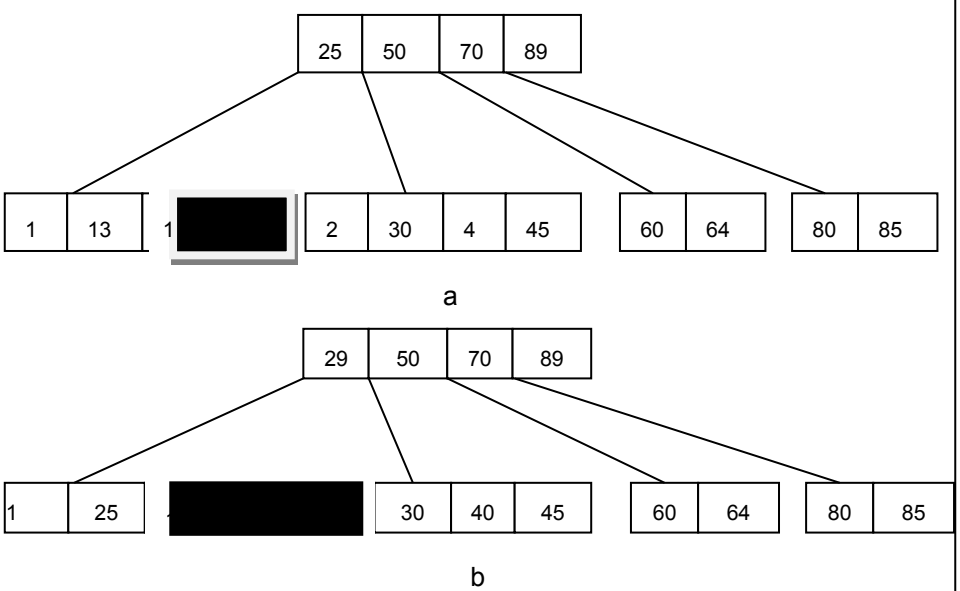


Figure 9-5: Splitting of a B tree and increase in height

During the above two, if the condition of minimum number of keys to be maintained is violated in that case, we look for the neighboring nodes (sibling) that are required to be

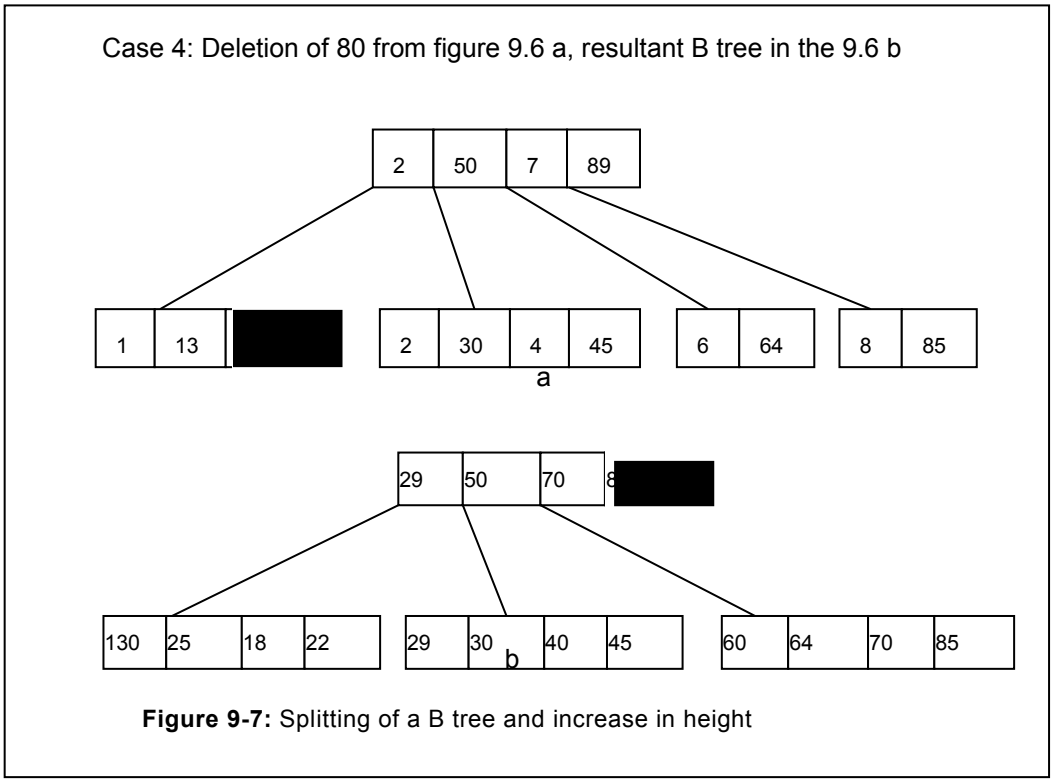


Figure 9-7: Splitting of a B tree and increase in height

merged.

- If one of the neighbors has more than the minimum key in that case, one of the key is promoted in the leaf node and the parent node is shifted in the leaf node.
- If both of the neighbor nodes do not have more than the minimum node in that case both the nodes are merged together.

All the above cases have been illustrated from figures 9.4 onwards.

9.3 B* TREE

Node of a B tree represents the block to be traversed for searching the needed information. Consequently, seek time will rise. To overcome this limitation, a new version of B tree known as B* tree is used. In a B* tree, splitting is delayed till all the nodes are not filled to 2/3 of its capacity. Therefore, low depth in a B* tree, may result in short seek time. The other measure is related to the number of split taking place. In case of B tree, one node splits into 2. However, in case of B* tree, two (2) nodes split into 3. Therefore, this split is also minimized. Consider a B* tree of order 9. In a B* tree of order 9, maximum number of keys that can be accommodated are 8. In the above figure 9.8 a, B* tree is illustrated. If the next element that

needs to be inserted is 10, it will be directed towards the left side. Since, left node is already full to its capacity. Therefore, inserting 10 will result in the split of the node. However, it has been notices that its right sibling is not full. Consequently, in the B* tree the split will be deferred till all the nodes of considered level are not full to 2/3. Since, in considered B* tree, its right node is not 2/3 full. Therefore, all the nodes will be linearly arranged, mid element of the linear order will be inserted at the top, whereas smaller will be arranged in the left and greater will be placed to the right. Similarly, if the nodes of the B* tree are full, in that case it will split. As depicted earlier.

9.4 B+ TREE

B+ tree is the special case of a B tree in which data is organized at the leaf of the tree. Interior nodes maintain only the reference. All the leaves are connected together to form a link list. Consequently, sequential accessibility is possible in the B+ tree. Since B+ tree is widely used in the storage of data in the storage devices. Therefore, items need to be

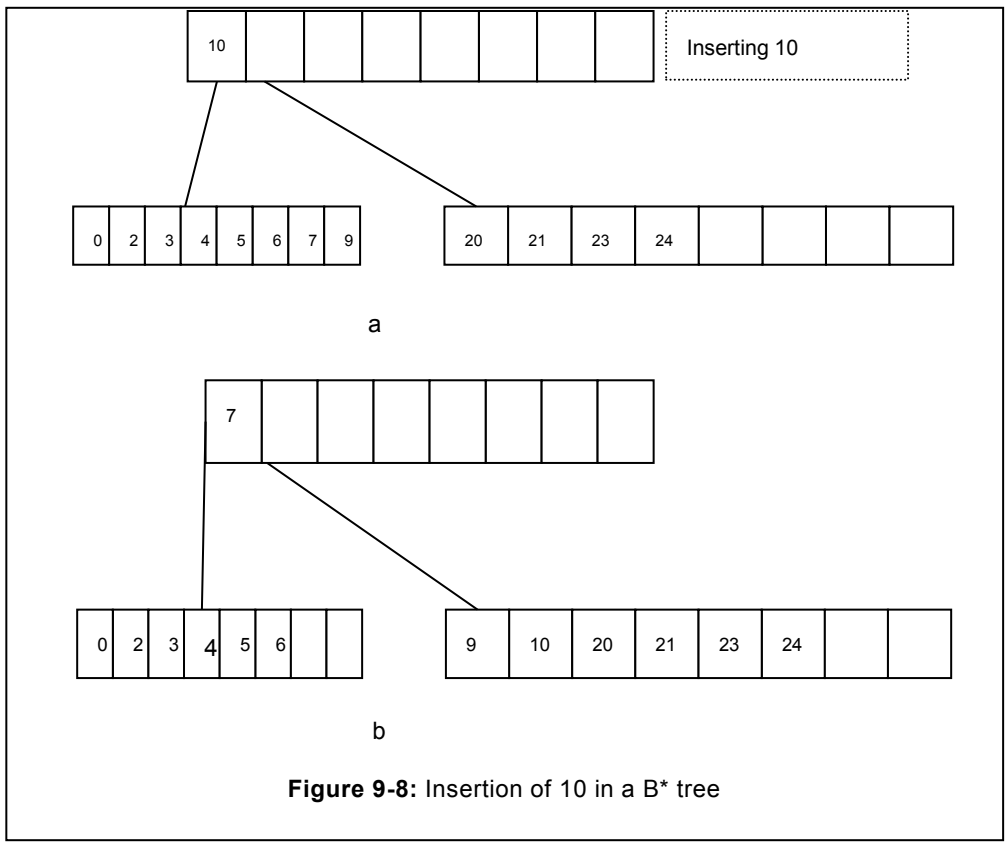


Figure 9-8: Insertion of 10 in a B* tree

arranged in orderly manner in order to ensure that it behave as an orderly tree. As we know that the orderly tree yields better performance relative to an ordinary list.

For a tree, to be qualified the definition of B+ tree, it should meet the following properties.

Consider a B+ tree of order ‘n’

- Root acts as a leaf node, if it is having at-least one element.
- Each node contains not more than n child and n-1 keys.
- Non leaf node consist of (n/2)-1 keys.
- Leaf nodes are connected to each other.

B+ tree is the widely used data structure for the storage, due to the availability of the data at the leaf node. Reference available at the top provides the routing capability.

Insertion in a B+ tree: Consider the B+ tree of the figure 9.9, from the illustrated figure it is apparent that all the nodes exist at the leaf node, and various leaves are connected with the help of a link.

Consider the case that ‘9’ is to be inserted into the B+ tree. Consequently, it will result in a split as illustrated in the figure 9.10.

Therefore, it is inferred that after split, copy of the key will also propagate to the upper level.

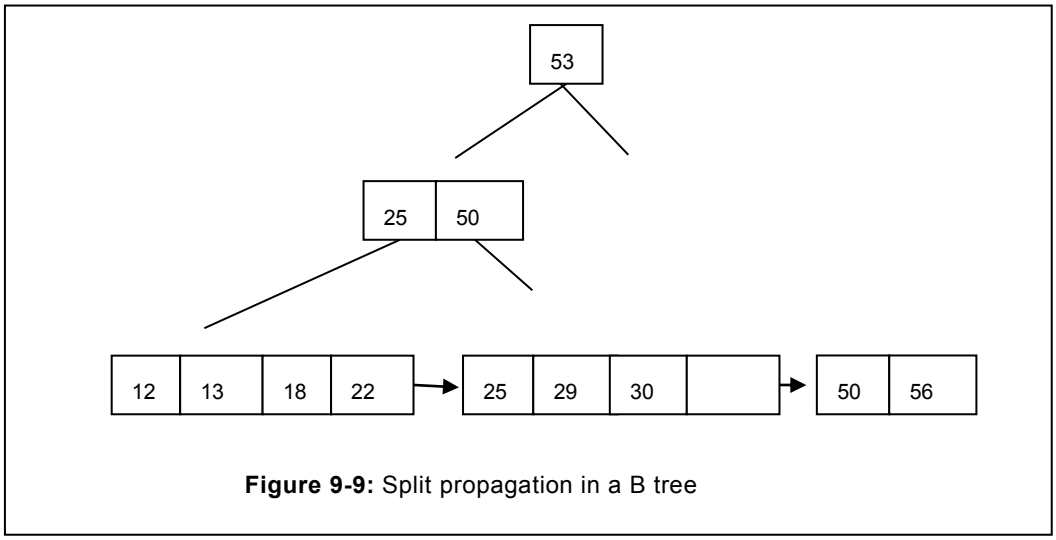


Figure 9-9: Split propagation in a B tree

Deletion in a B+ tree: Deletion in a B+ tree is similar to the B tree. No merging occurs if the deletion of a key does not violate the rule necessary for the B+ tree. However, if deletion of a node leads to a node that the number of keys remaining violates the condition of the B+ tree in that case the nodes may be merged.

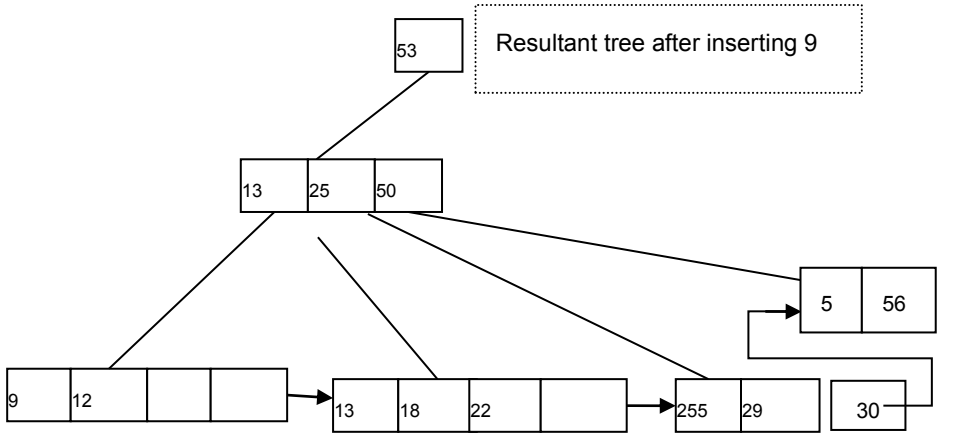


Figure 9-10: Split propagation in a B tree

EXERCISES

A. Descriptive Type Questions

1. What do you understand by multi-way tree? Explain when a tree can be classified as a multi-way tree? Justify.
2. Distinguish the multi-way tree and a Binary search tree.
3. What are the restrictions applicable to a B tree?
4. Define the B+ tree. Explain the significant advantages of B+ tree.
5. Define the B* tree. Discuss the significant advantages and disadvantages of B* tree.
6. Write down the 3 major uses of the B tree. How the performance is governed by these multi-way trees?
7. Distinguish between B+ tree and B* tree.
8. Consider that you have been given a problem after analysing the problem; you could ascertain that you need to apply the multi-way tree. Which multi-way tree you will like to apply. Justify your answer.

B. Short Answer Type Questions

1. B tree is also a BST [True/False].
2. All the BST can be termed as B Tree [True/False].

3. Tree having…….can be termed as multi-way tree.

a. n number of children
c. n number of keys.
b. Both a and c
d. None of these

4. In a B tree of order N, it can have…….minimum children:

a. N
c. 2^N
b. 2^{N-1}
d. None of these

5. In a B+ tree number, of order N, total number of leaf node would be

a. N/2
c. 2^N
b. Cannot be predicted
d. None of these.

6. In the B tree, split may take place if the number of node is less than n/2 [True/False].

7. In a B+ tree, split is deferred till the children are not filled to ….

a. 2/3
c. ½
b. Both a and c
d. none of these.

Chapter 10

Searching and Sorting

Chapter Objective

- Defining the searching and its significance
- Describing the various types of searching
- Defining the sorting and its various types
- Describing the simple sorting methods
- Describing the complex sorting .

10.1 INTRODUCTION

Sorting and searching are the basic operations for any collection of records maintained by any business entity. Query or record requested should be accessed in the minimum time. Delay in the accessibility result in poor experience. Sorted data facilitates in fast accessibility/searching of records. These two operations are complement to each other. The upcoming sections highlight the many of the prominent searching and sorting techniques that have significance for this topic.

10.2 SEARCHING

Sorting and searching are profoundly significant in data structure. In case of query, we need to search the record from the existing database. Searching time will be minimum in sorted list, relative to the list that is unsorted. In additions, accessing all the nodes or escaping some of them have great impact on the performance. According, searching is categorized into the following types.

10.2.1 Linear Search

In the linear search, data are randomly placed based on their arrival and no rule is applicable. In this method, each elements of the list is matched with the item to be searched. This process continues till element is not found or the list is not exhausted. Linear search is a simple method to implement but performance is low. Performance further degrades upon increasing the size of the list.

```
Program: To search the element from the given list and return the
search status

int search (int list [], int element)
 {
```

```
int i=0, found=0;
while (list[i])
 {
if (list[i] ==element)  // if the element is found, terminate
  {
found=1;
break;
}
i++;
}
return found;
}
```

We can also return the value of index, instead of returning the status of flag. In a list, if more than two numbers with same value exist in that case, number searched place will be returned. To return the index instead of status from the above program, it can be suitably modified as depicted in the following program.

```
Program: To search the element from the given list and return the index where item is found
int search (int list [], int element)
 {
int i=0, index=-1;
while (list[i])
 {
if (list[i] ==element)  //  if  the  element  is  found,  terminate checking
  {
index=i;
break;
}
i++;
}
return index;
}
```

Performance of search program is governed by the number of comparison needed to trace the target element. If the target element is found at the first position itself then its complexity will be O (1). Similarly, if it is found at the last position then it's complexity will be O (n).

In the worst case, linear search is having the complexity of O (n). It signifies that delay in searching is purely governed by the number of elements. However, based on the present demand of performance, the delay caused is significant. **If the item does not exist in the list**, in that case also all the elements need to be compared with the target item.

To receive the rich performance experience, we can use sorted linear list. The other solution based on the accessibility exists. In this method, elements are arranged based on their accessibility. It is observed that in a given list 20 % items are frequently accessed. If we can identify these 20 % item and arrange them at the beginning then the performance of the linear search can improve significantly. Therefore, it is required that the list be arranged for the better performance.

10.2.2 Binary search

Binary search is another method that is used for searching. However, it is much efficient relative to linear search. Arranging the items in sorted order is the pre-requisite for a Binary search. In binary search, we do not trace the target element one after the other as we do in linear search. Instead, we follow the approach similar to the one used in our dictionary, where we open the random page, look for the word, if word is on the same page then start searching the word. Otherwise we are moving left if the alphabet is greater or right if the alphabet is smaller from the word to be searched. In binary search, we follow the same approach. Binary search can be implemented with the help of recursive and non-recursive approach.

10.2.2.1 Recursive approach

In the recursive approach, we determine the mid element of the array. If the mid element is equal to the target item in that case we terminate here itself, else we recursively call the function. At any point in time, lower index exceeds the upper index, binary search is terminated and the message of item not found is suitably returned. We place the testing of lower and upper in the base condition.

```
//function for the recursive search
int rec_bsearch (int x [], int l, int u, int num)
{
int mid= (l+u)/2;
cout<<" l ="<<l<<" u="<<u<<endl;
 if(x [mid] ==num)
 return 1;
 if (l>u)
```

```
 return 0;
 if (num < x [mid])
  return rec_bsearch(x, l, mid-1, num);
  else
  return rec_bsearch(x, mid+1, u, num);
}
```

10.2.2.2 Non-recursive approach

In the non-recursive approach, we initialize the lower and the highest element and follow the same approach as followed in recursive method, but using non-recursive technique. In this method, we terminate the searching procedure immediately upon the item is traced. In order to achieve this objective, we have used the variable 'flag' to determine whether the item has been located in the given list or not. Initially, we consider that the item does not exist in the list; correspondingly, we set the value to 0. Once the element is located, in that case flag is set to 1.

```
// Non-recursive function to search an element
int bsearch (int x [], int num)
{
int lower, upper, mid, flag=0;
lower=0;
upper=9;
while (lower<=upper)
{
mid= (lower+upper)/2;
if(x [mid] ==num)
 {
 flag=1;
 break;
 }
 if (num < x [mid])   // Element may be at left hand side
   upper=mid-1; // move the highest to the left side
   else           // Element may be at right hand side
   lower=mid+1; // move the lower to the right side
 }
 return flag; // Finally, return the flag
}
```

10.3 SORTING

Items arranged in a list are in random order and yields lower performance during searching. Although, once we compute the complexity of sorted or unsorted array, it gives the same result; however, practically sorted array gives better performance relative to the unsorted array. Sorting can be broadly categorized into two:

- External sorting
- Internal sorting

Sorting in which all the elements are not loaded into memory instead are available in secondary storage then such type of sorting is known as **external sorting**.

The other type of sorting is known as internal sorting in which elements are stored in the memory for sorting. Due to availability of items in primary memory, internal sorting is faster relative to the external sorting. Sorting techniques that are discussed in this book are example of internal sorting.

A. Bubble Sort

Bubble sort functions similar to the bubble that has the characteristics of rising to the upper surface due to light weight, in other words heavier elements settles downwards. Bubble sort uses the aforesaid technique. In the first pass, the heaviest element is arranged at the right most position. In the second pass, second highest element is arranged at the penultimate location of the list and so forth. Working of the bubble sort has been illustrated in the figure 10.1.

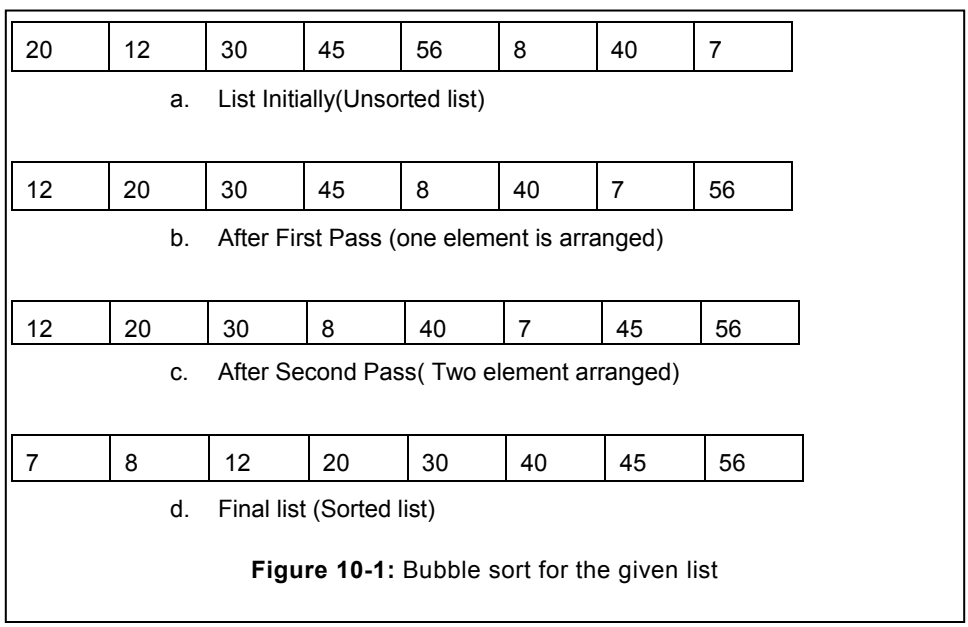


Figure 10-1: Bubble sort for the given list

In the bubble sort, number of iteration of a loop is entirely governed by the items that are arranged. Since, in each pass, one element will be placed at the appropriate position, therefore in the second pass onwards this element will not be disturbed. Accordingly, we make the provision to ensure that iteration does not reach to this element.

Other case may be that the array given is already sorted. In this case no items will be swapped. In order to ensure that the written program remain efficient, we need to determine whether the swapping of element is taking place or not. If it is not occurring then it denotes that the elements are sorted. Correspondingly, we terminate the loop here itself. Working of the program has been demonstrated in the following program.

```
// Problem: Program to sort the series using bubble sort
void bubble (int a [])
{
int i, j, temp;
for (i=0; i<8; i++)
{
for (j=0; j<7.i; j++)
 {
  if (a[j]>a [j+1]) // if item is more than the next item
   {
   temp=a[j];     // swapping if the right element is more
   a[j] =a [j+1];
   a [j+1] =temp;
   }
 }
 cout<<endl;
}
}
```

To determine the complexity of the bubble sort, we observed that for first pass the inner loops run for n-1. For the second pass, it runs for the n-2. It means for all the elements total numbers of iteration are 1+2+3+……..n-1, that result in n (n+1)/2. Therefore, we can conclude that the complexity of bubble sort is O (n^2).

B. Insertion sort

In the insertion sort, elements are placed in appropriate position from left to right. Initially, we consider that the first element is at appropriate place. Sorted elements are arranged in the list known as left hand side. After the first element, we consider the second element and compare it with the items available to its left hand side. During the scanning of the items, it is placed at the appropriate place. All the remaining items are shifted toward the right hand

side. Working of the insertion sort is demonstrated in the following program.

```
// Program to sort the array
void insert (int a [])  //function to insert the array
 {
 int k, index, temp, item, i;
 for (k=0; k<n; k++)    // Starting from the first to n element of
the array
   {
   item=a[k];     // storing the current element at item
   for (i=0; i<k; i++)  // To determine the position
     {
     if (a[i] > a[k])   // if the large element is found
      {
      for (int t=k; t>i; t--) // shift all the elements to the right
      a[t] =a [t-1];          // element by element
      a[i] =item;       // Place the item at right position
      break;                  // Stop the inner loop
      }
    }
  }
}
```

The outer for loop is used to sort all the elements of the array one by one. It is followed by the inner loop that runs from the 0^{th} index to the position under consideration. For instance, if the index/location under consideration is 5, then the element has to be inserted between the 0^{th} to the 4^{th} location. Consequently, element is to be inserted at the position where the element with greater value from the current element is encountered. Third loop with the variable 't' starts with the last index and moves the element one by one. This results in creation of the space for the item under consideration. Eventually, element is placed at the hole created.

C. Selection sort

In the selection sort, smallest item is selected from the entire list under consideration. In each pass, this smallest element is placed at appropriate place. In the first pass, we find out the smallest items from all the available items in the array. Once it is traced, then we can place it at first position. In the second pass, we try to find out the smallest items from the remaining list (other than the first item that is already smallest), once it is traced then it is placed at the second position and so forth. Here we need to note that the element under consideration is swapped with the last smallest element. Working of the selection sort is demonstrated in the following program.

```
Program: Write down the program for selection sort
/* This program sorts the array using selection sort. It swaps the item wherever fewer items are encountered. */
void selection (int a [])
 {
 int i, j, temp;
 for (i=0; i<=7; i++)
  {
  for (j=i+1; j<=7; j++)
   {
   if (a[i] > a[j])
     {
     temp=a[i];
     a[i] =a[j];
     a[j] =temp;
     cout<<" a[i] "<<a[i] <<"i="<<i<<endl;
     }
   }
}
}
```

```
Program: Write down the program for selection sort
/*This function also sorts the item using selection sort, but it identifies the minimum item. At the end swap it with the array element.*/
void selection2 (int a [])
 {
 int i, j, index, min;
 for (i=0; i<=7; i++)
  {
  min=a[i];
  index=i;
  for (j=i+1; j<=7; j++)
   {
```

```
if (min > a[j])
  {
  min=a[j];
  index=j;
  cout<<" Min "<<min<<"index="<<j<<endl;
  }
}
if (i! =j)//item is not the minimum
  {
  a [index] =a[i];
  a[i] =min;
  }
}
```

10.4 EFFICIENT SORTING ALGORITHMS

All the algorithms that are discussed above worked on the principle of exchange, as a result, the algorithm were less efficient. All of them were having the complexity of O (n^2). Their efficiency having compared hereunder:

Table 10.1: Values for various N

N= / efficiency	1	100	1000
N	1	100	1000
N^2	1	10000	1000000
logN	0	4.60	6.9

Therefore, we need algorithm that should result in better performance. Algorithm discussed below gives better performance.

A. Bucket sort

In the bucket sort algorithm, we arrange the items in groups based on their category. These categories of items are decided by the start value. For instance, consider the series 28, 17, 40, 99, 27, 22, 96, 98. In this series, items are arranged as per their weight in a group. Series with 1 in tenth places is grouped in one bucket, similarly numbers at tenth position is 2, and it will be grouped into 2^{nd} group. Process will continue upto the value of 9. For the given series, bucket has been illustrated in the following figure.

Initial list
28,
17,
40,
99,
27,
22,
96,
98

Elements after completion of phase 1		
17,		
22,	27,	28,
40,		
96,	98,	99

Figure 10-2: Bucket sort

Within the bucket the elements are arranged in sorted order. For instance, if the number starts with the number 2 and it is having 3 elements as 28, 22, 27. These numbers will be arranged in sorted order as 22, 27, 28. After the completion of placing phase, we start with the positioning phase. Here we start with the picking the items from the first bucket, second bucket third bucket etc. till all the buckets are not exhausted. Due care is exercised while placing the item in an array that the sequence of item within the bucket is not changed. Since, within the bucket items are placed in sorted order, therefore, all the number will be in sorted order.

B. Count sort

Count sort is another sorting method that is increasingly efficient. In the count sort, we consider the ‘n’ elements ranging between ‘0’ to ‘m’. We find out all the elements that are less than the particular element. Here in this case, we have considered that the numbers are equal. Here first count the number of particular digit that is followed by commutation of the numbers. To solve the count sort, we would consider two arrays, one consisting of the elements whereas the other stores the count of particular number. Consider a series consisting of 10 numbers. All these numbers are ranging from 0 to 7. Working of count sorting has been illustrated in the following figure 10.3.

n the considered case:

n=10

Range=0.6

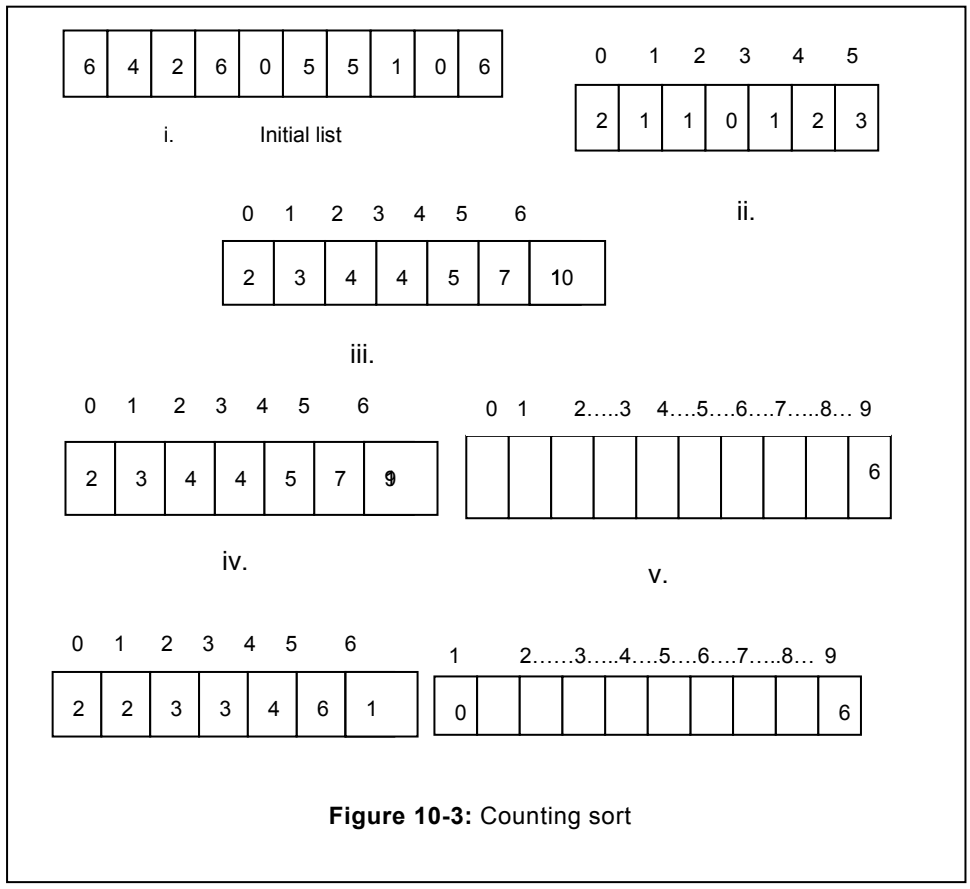


Figure 10-3: Counting sort

Therefore, we will count the numbers which are equal. Correspondingly, the count is written as illustrated in figure ii. Finally the count is commutative that assist in determining the number of elements less than the current number. Refer the list given in the i) again- Element is 6, Go to the 6^{th} index of iii. In this case value is 10. Consequently, 6 is to be placed at the 10^{th} position of the sorted list. Since, one item is already placed. Reduce the sum by 1 at the index position considered. This is followed by propagation of (-1) till the last element is not reached. In the considered example, since the last item is considered therefore, there is no propagation and reduction in value will occur.

C. Radix sort

In the radix sort, numbers are arranged based on their radix. We define the radix as base. For instance, in decimal number radix is 10. In radix sort, numbers are arranged based on the positional value of a number. First on the most significant value, and afterwards move towards least significant value. In each pass, we arrange them based on one positional value. At the end, whole series is available in the sorted order. This series make the usage of count sort and arrange the series based on the count method.

D. Quick Sort

Quick sort is a widely used technique to sort series. In each pass, series is divided into parts. Pivot element governs the place from where the series is to be divided. Pivot element can be defined as the first element of the series that need to be placed at appropriate place. All the elements left to it are smaller than the pivot element whereas all the element greater than the pivot elements are placed to its right.

Placing the pivot element at appropriate place need number of iterations. Movement terminates as soon as item encountered is greater than the pivot element. Index of this element is stored and termed as lower index. After that pivot element is moved from the extreme right towards the left till the elements is greater than the pivot element. As soon as the item less than the pivot element is encountered than we terminate here. This place is termed as upper index. If lower index is less than the upper index than both the items are swapped. This is followed by comparing the elements from the left side where we have left (left index) and applying the procedure from the right as one already discussed. After the complete process, the pivot element is placed at exact position. Items to the left of the pivot will be less and the items right of the pivot elements will be greater. Consider the series of

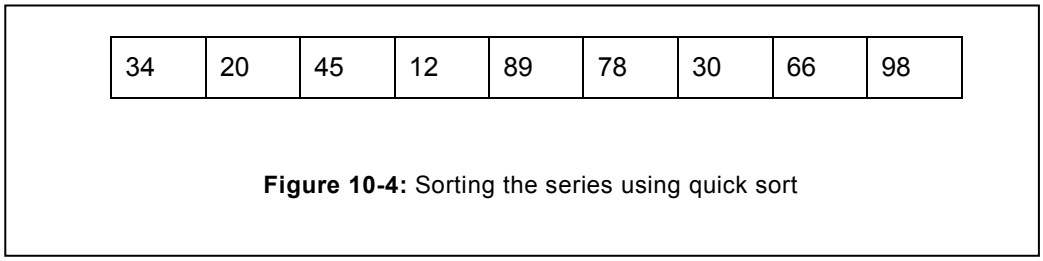


Figure 10-4: Sorting the series using quick sort

figure 10.4.

In this series there are total 09 elements. Therefore, lower index will be 0, whereas the upper index will be 07. In the given series, the first element is 34, this element is known as pivot element. This element is compared with the other elements of the series (starting from the left side of the series) one by one. This comparison starts from left; in our case we have considered the left hand side as starting point. During movement towards the right, we ensure that the element encountered is less than the pivot element. If the element is not smaller, our traversal terminates and we have to check starting from the right most position. From right side, we continue to move towards the left till elements encountered are less than the pivot element is encountered. The moment element that is smaller than the pivot is encountered, we terminate the movement, and here we exchange both the elements (greater of the left side to the smaller of right side) that are items at lower index with the item at upper index.

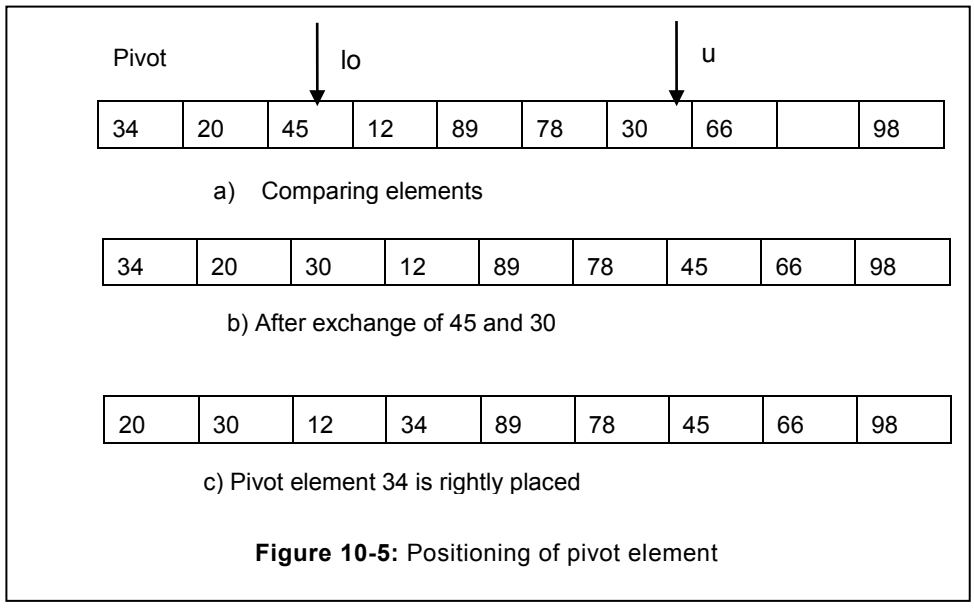


Figure 10-5: Positioning of pivot element

```
// program for the quick sort using recursive method
void quicksort (int ar [], int lower, int upper)
 {
int i;
if (upper > lower)// number of elements greater than 1
{
i=partition (ar, lower, upper);
quicksort (ar, lower, i-1); // quicksort for the left half
quicksort (ar, i+1, upper); // quicksort for the right half
}
}
```

Partition function is given as

```
// function to partition after placing the pivot element at the
appropriate place.
int partition (in ar, int left, int right)
{
int l, r, temp, i; // l for the left, r for right
l=left;
r=right +1;
while(r>= l)
```

```
 {
while (ar [++l] < i); // move right till elements are smaller than pivot
while (ar [--r] > i); // move towards left till elements are greater than pivot
if (l > u)
  {
t=ar[l];
ar[l] =ar[r];
ar[r] =t;
}
}
t=ar [lower];
ar [lower] =ar[r];
ar[r] =t;
return r;
}
```

E. Merge sort

In the merge sort, we follow divide and conquer method. In divide and conquer, first we divide the series into individual number, then we conquer them we arranging the items in groups. Finally, we follow the merge method to create the complete list. Therefore, merge sort is the popular example of the divide and conquer method. Functioning of the merge sort is dependent upon the count and radix sort. It is considered as one of the most efficient algorithm that falls under the category of asymptotically algorithms. In the given series of figure 10.4, the total number of elements is 8; therefore it will be divided into two groups.

One list will consisting of first four elements, while the second series consisting of the rest of the four elements. Now both the series are (28, 17, 40, 99) and (27, 22, 96, 98). In divide and conquer method, we go upto the lowest level, it means that the series have to be divided further. Since, each series comprises of 4 elements therefore respective series is to divide into the group of 02. (((28, 17), (40, 99)), ((27, 22), (96, 98)). Finally, we divide into the group of 1 as (((28), (17)), (40), (99)), ((27), (22), (96), (98)). Now they are divided into the lowest component. Here, each series is consisting of one element therefore, it is considered as sorted. Afterwards, elements are grouped in two and they are sorted as (17, 28) (40, 99) (22, 27), (96, 98). After that they are merged into the group of four and sorted among them as (17, 28, 40, 99) (22, 27, 96, 98). Now both the series are in sorted order. Finally, we group both the series and sort them. Consequently, the series obtained will be in the sorted order as (17 22 27 28 40 96 98 99). This approach is known as bottom up approach method.

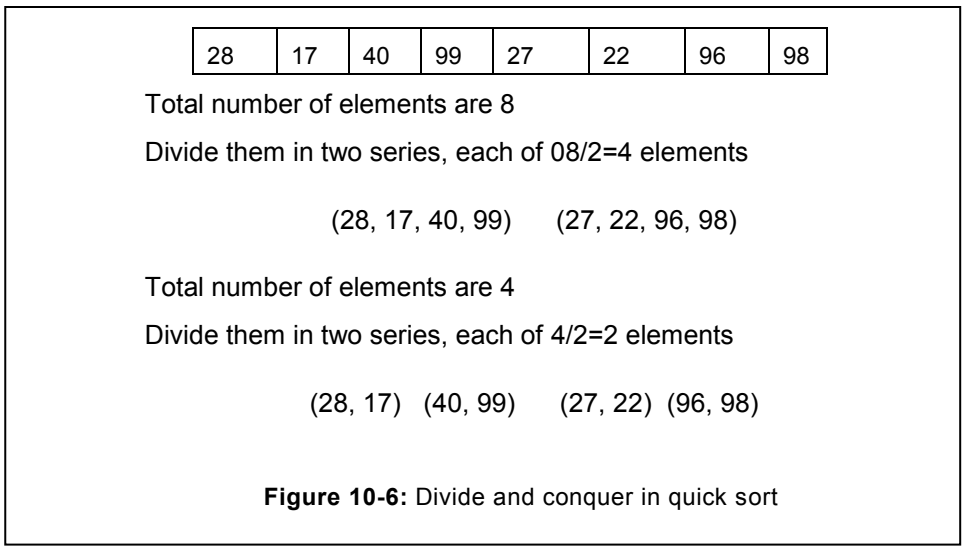


Figure 10-6: Divide and conquer in quick sort

Program to depict the merge sort has been given as follows.

```
// Program to sort the array using mergesort
void mergesort (int ar [], int lower, int upper)
 {
int mid;
if (upper > lower)
 {
mid= (lower+upper)/2;
mergesort (ar, lower, mid);   // divide the series into two (left side)
mergesort (ar, mid+1, upper); // divide the series into two (right side)
merge (ar, lower, mid, mid+1, upper); // merge the half
}     // end of if
}     // end of function
```

F. Heap sort

Heap is a data structure that is widely utilized in priority queue. Heap is defined as follows:

- A large dynamic area of memory that can be utilized by the programmer and same may be de-allocated when not needed.
- A balanced, left justified binary tree in which no node has a value greater than the value in its parents.

Heap sorts follow the second definition. Quick sort proved to be immensely effective in sorting algorithm. Same is apparent with the performance that yield performance of O (n log

n), however, in the worst case it slows down to O (n^2). New sorting method known as heap sort guaranteed O (n log n). Consequently, it is widely used in the critical applications where the time is a significant factor.

To know the heap sort, we need to know certain definitions.

Heap Properties

In order to qualify for a heap following conditions need to be satisfied:

- Node in a tree follows the heap property if its root is greater than the children
- All the leaf nodes follow the heap property.

If a tree does not follow the heap property, an operation known as shift operation has to be applied, that enables the given tree to qualify as heap. Once this operation is applied at one level, node at the upper level may violate the heap definition. Therefore, shift-up operation is repeatedly applied till it does not follow the heap property or it has not reached the root node. Tree fulfilling the heap sort has been illustrated in the following figure.

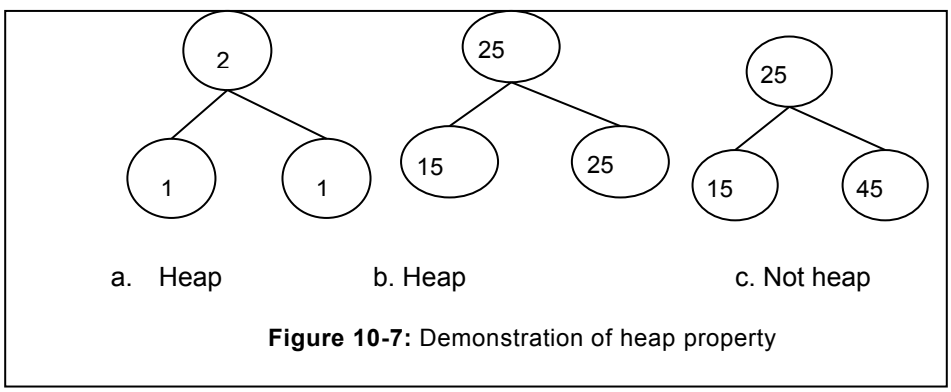


Figure 10-7: Demonstration of heap property

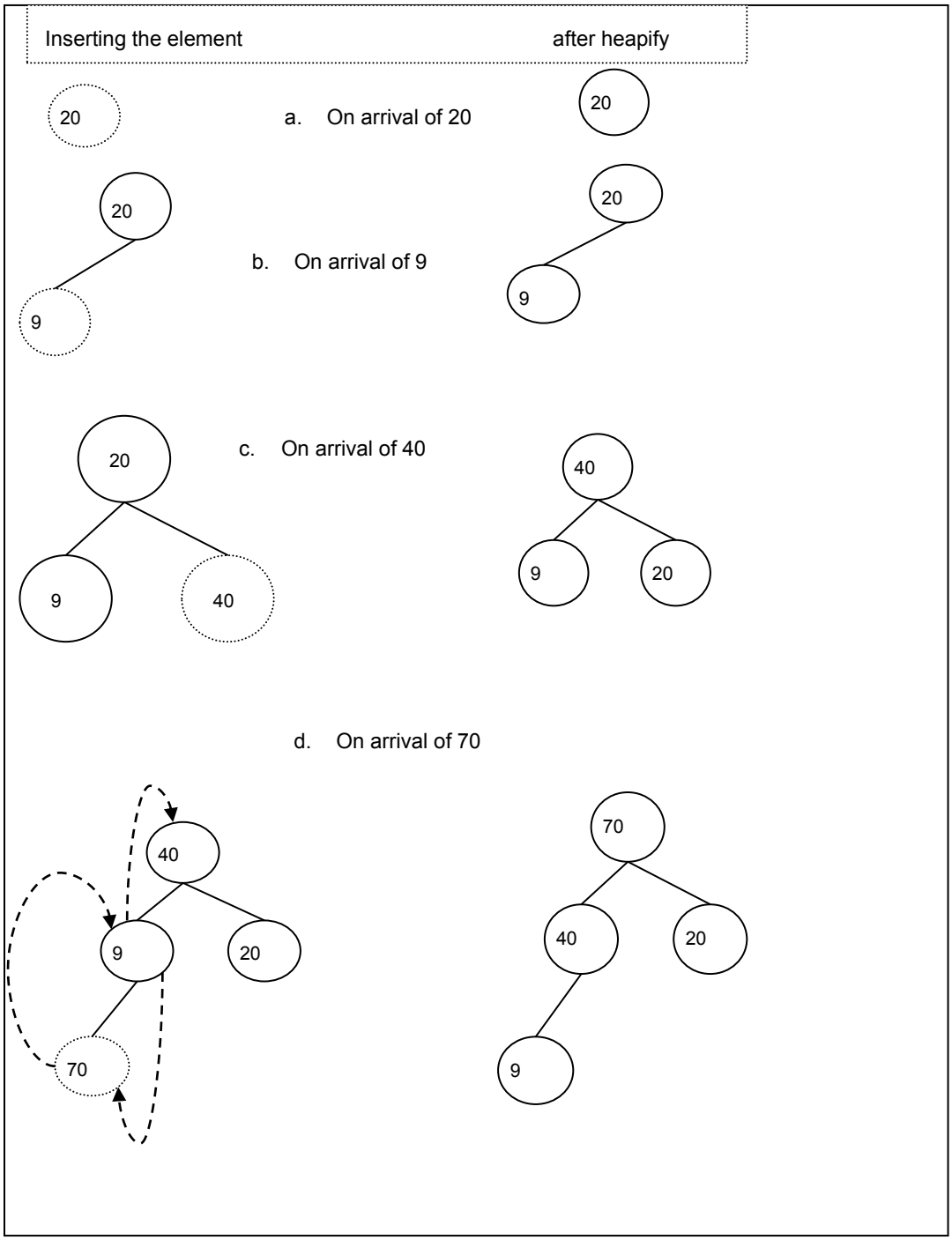
Constructing the Heap

Now, we would like to learn about the heap with the help of an example. Consider the series, 20, 9, 40, 70, 56, 34, 90, 12, 76, 80, 33 for which heap has to be constructed that will lead to the list that will be in sorted order.

On the arrival of 20, since no root is existing, consequently, the root will be created. On the arrival of 9, it will be routed to the left and will act as the left node of the heap tree. Since, attachment of the 9 does not violate any condition. Therefore, it will remain to the left of the existing tree as illustrated in the figure 10.7(b). On the arrival of 40, it will be attached to the right of the node, since the heap definition violates, consequently, shift-up operation is to be carried out. During shift operation, its parent node (i.e. 20) will occupy its position whereas the new node 40 will act as parent node. Same has been illustrated in the figure 10.7(c).

Upon arrival of 70, it will be inserted at the extreme left as illustrated in the figure 10.7 (d). insertion of this node will violate the definition of heap. Consequently, shift-up operation is to be carried out till it does not adhering to the heap definition. On shifting one level up, we

observe that it violates the heap definition. Consequently, it is shifted-up again- Here it is acting as the root node. Since, no upper level is available; therefore it is allowed to remain there. It is also observed that now it is not violating any heap definition.



This process continues till all the elements are not inserted in the heap. Complete heap is illustrated in the figure 10.8.

Next task is to select the items from the heap so that they can be stored in an array in sorted order. During the selection of the elements root is selected first, since it is the maximum element. It is placed at the last index of the array, as illustrated in the figure 10.10.

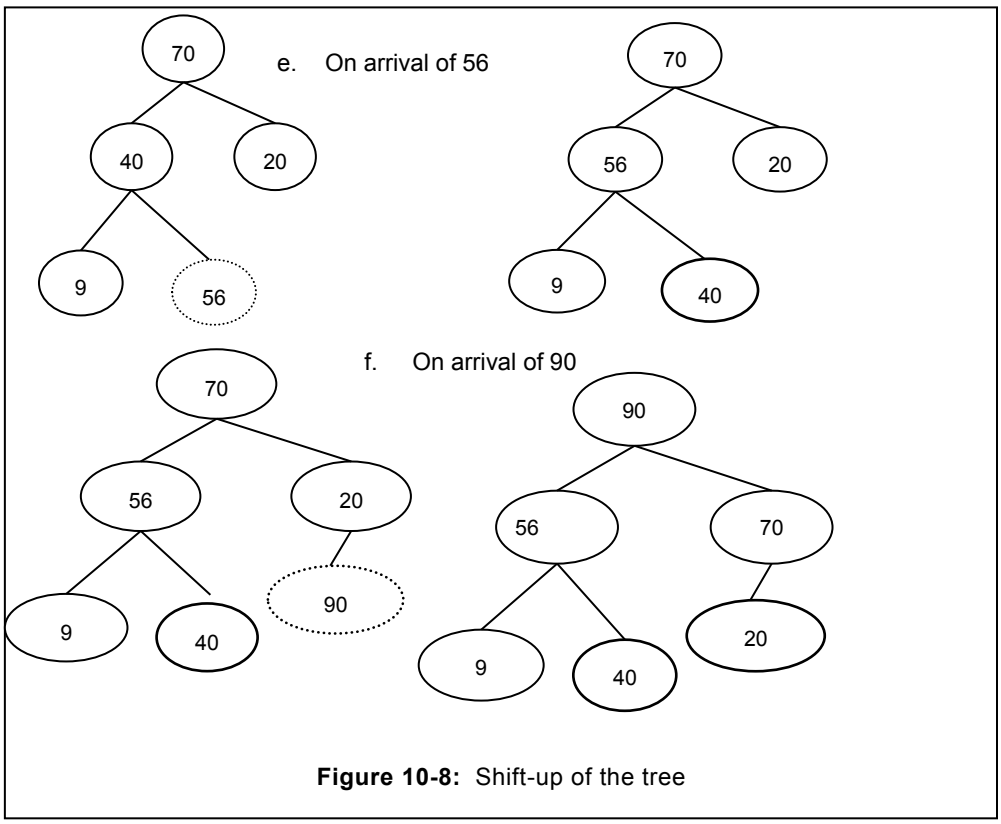


Figure 10-8: Shift-up of the tree

This is followed by determining the right most node of the **lowest level** of the tree. If the rightmost is not available than the left node of the same parent is considered for replacement with the root as illustrated in the 10.9(b). In the considered figure the rightmost node at the lowest level does not exist. However, the left node (20) of the same parent i.e. 70 is to be placed in the root.

Replacement of item may lead to the violation of heap definition. Consequently, again all the elements are checked for their values. If element is more, shift up operation results. S ince, in the resultant tree of 10.9 (b) heaps definition violates due to the availability of items in the tree which are more than the root node. Consequently, shift up operation is to be applied. Resultant tree is illustrated in the figure 10.11.

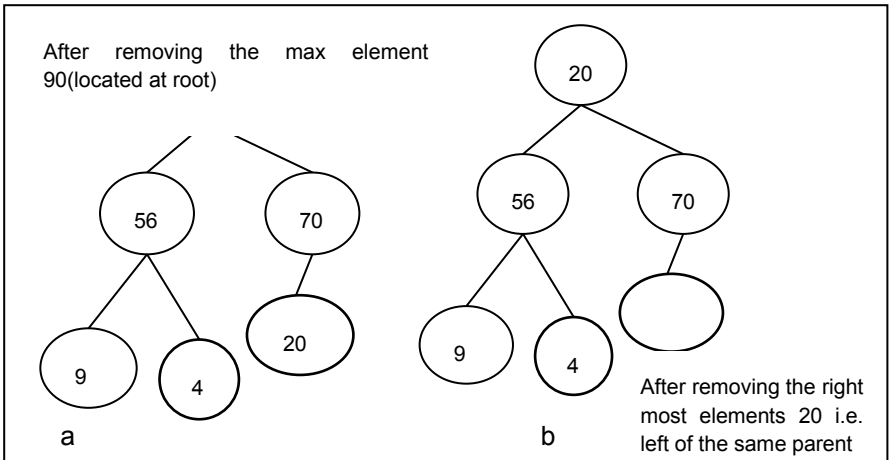


Figure 10-10: Removing maximum element

Similarly, we would remove the next root element from the heap, and place it in the array (at the first position available at the extreme right of the node, i.e. the last index of array). Now the highest element that is available in the root is 70, therefore this item is selected and placed in an array. It is apparent from the ongoing list that the elements are arranged in the

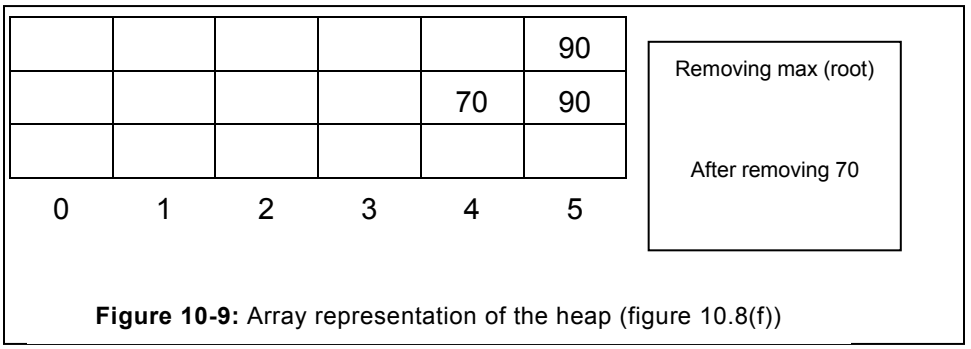


Figure 10-9: Array representation of the heap (figure 10.8(f))

descending order from the last index of the array.

After removal of the root element, the resultant tree has been illustrated in the figure 10.12 (a). Afterwards, we figure out the right most element of the lowest level. In this case, it is 40; therefore, it will be placed at the root of a tree. Again, we would examine the remaining tree whether it qualifies the definition of heap or not, if it definition of heap is violated, shift up operation is carried out inorder to allow it to further qualify as heap. Consequently, the resultant tree is illustrated in the figure 10.12(c).

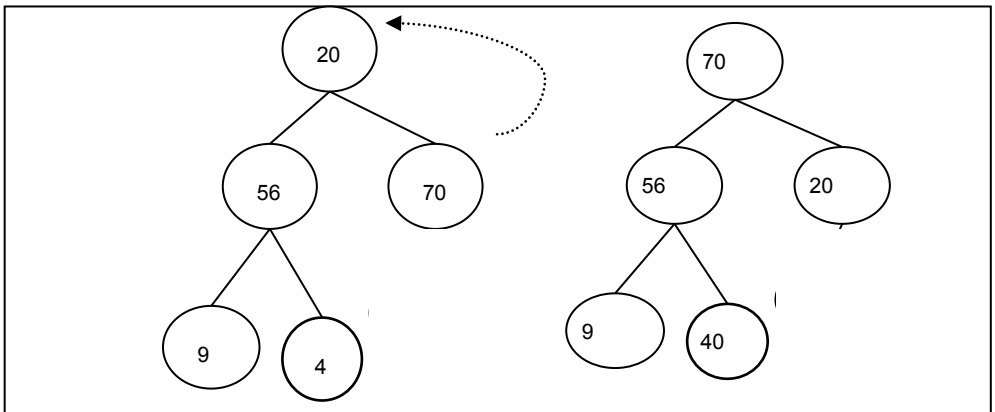


Figure 10-11: Shift up operation after removal of max element

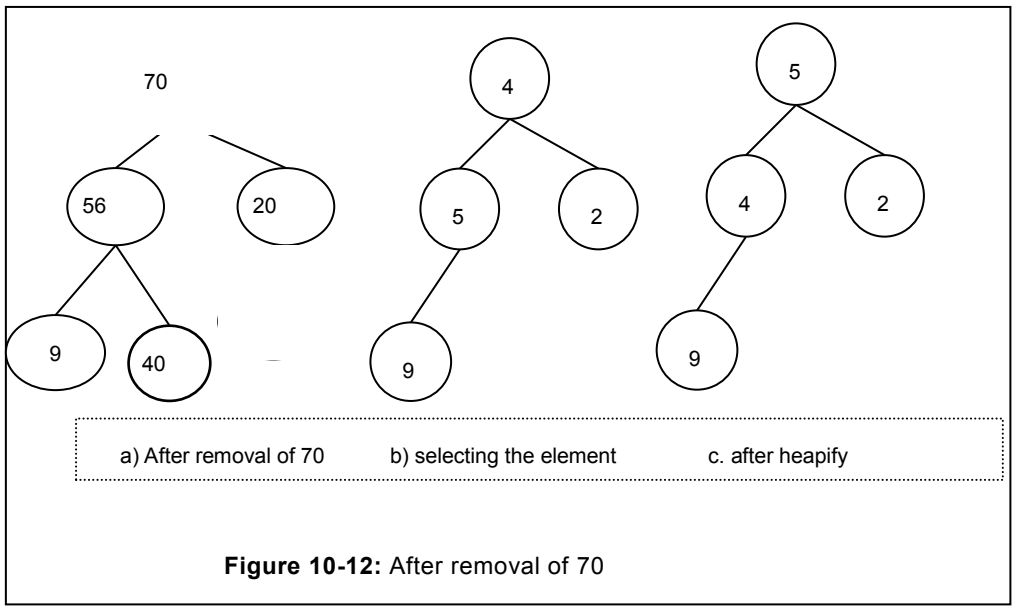


Figure 10-12: After removal of 70

Now again, since root is the highest element, in this case it is 56, it would be selected and placed in an array. Consequently the resultant array will be

			56	70	90
0	*1*	*2*	*3*	*4*	*5*

However, removal of 56 will leave the heap as illustrated in the figure 10.13 a. Again, we have to figure out the right most element of the lowest level. Since, right most element does

not exist, whereas the left element of the same parent is 9. Consequently, it will result in the heap of 10.13(b). Since, this heap violates the condition; therefore, the resultant tree has been illustrated in 10.13(c). Now again, since root is the highest element, in this case it is 40, it would be selected and placed in an array. Consequently the resultant array will be

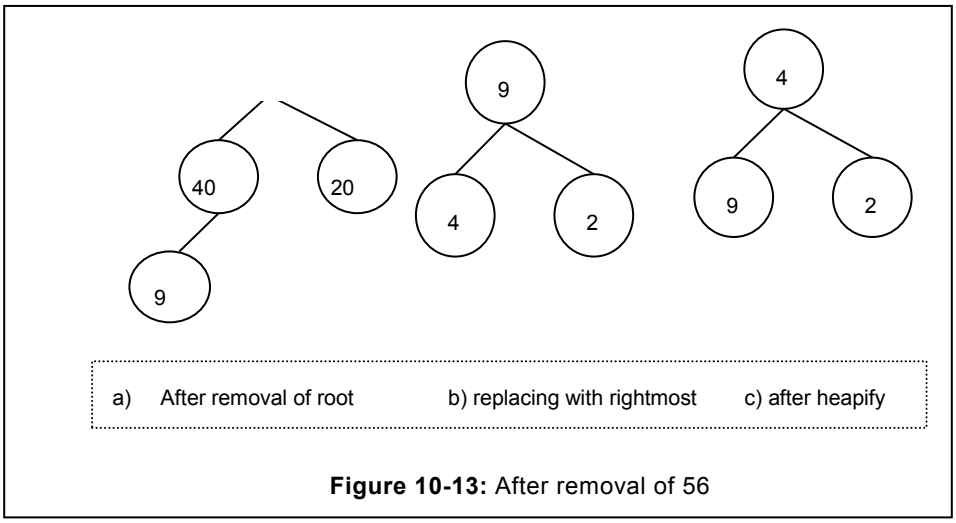


Figure 10-13: After removal of 56

		40	56	70	90
0	*1*	*2*	*3*	*4*	*5*

However, removal of 40 will leave the heap as illustrated in the figure 10.14 a. Again, we have to figure out the right most element of the lowest level. In this case it is 20. Consequently, it will result in the heap of 10.14(b). Since, this heap does not violate the condition; therefore, the resultant tree has been illustrated in 10.14(b).

Now again, since root is the highest element, in this case it is 20, it would be selected and placed in an array. Consequently the resultant array will be

	20	40	56	70	90
0	*1*	*2*	*3*	*4*	*5*

However, removal of 40 will leave the heap as illustrated in the figure 10.14 a. Again, we have to figure out the right most element of the lowest level. Since, right most element does not exist, whereas the left element of the same parent is 9. Consequently, it will result in the heap of 10.15(b). Since, this heap does not violates the condition, therefore, the resultant tree has been illustrated in 10.15(b). However, removal of 20 will leave the heap as illustrated in the figure 10.15 a. Again, we have to figure out the right most element of the lowest level. In this case it is 20. Consequently, it will result in the heap of 10.14(b). Since, no more nodes exist therefore, this heap does not violate the condition, and the resultant tree has been

illustrated in 10.15(b).

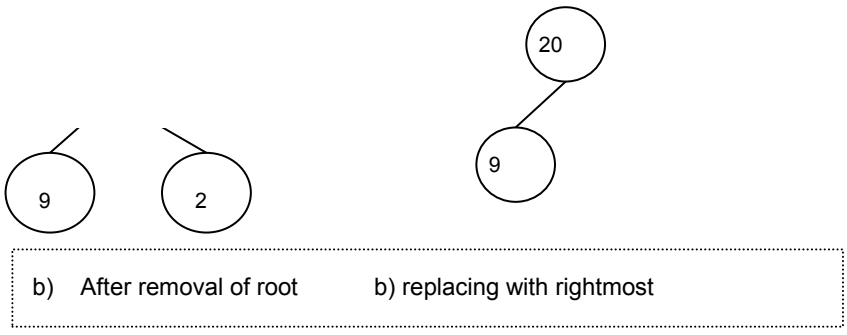


Figure 10-14: After removal of 40

Eventually, only one node is left and one space is left therefore, the remaining node will be selected and placed in the array as shown below.

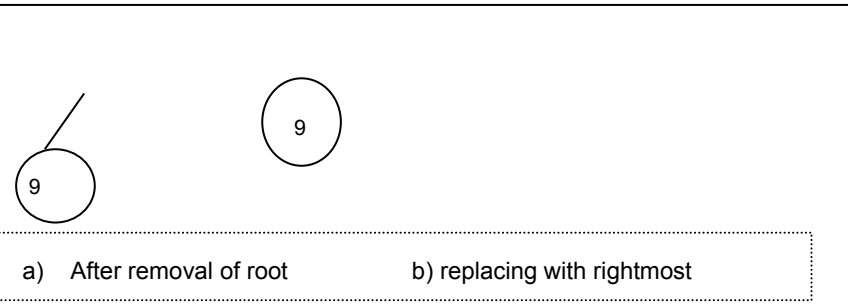


Figure 10-15: After removal of 20

9	20	40	56	70	90
0	*1*	*2*	*3*	*4*	*5*

It is obvious from the above discussion that the heap sort result in a sorted array as shown above.

EXERCISE

1. A series has been given as 34, 6, 12, 90, 33, 86, 76 what will the output of the given series after 3rd pass in

i) Insertion sort

ii) Selection sort

iii) Bubble sort

2. For the given series, 34, 6, 12, 90, 33, 86, 76 which algorithm (insertion, selection, bubble) will be best algorithm. Justify your case.

3. In the bubble sort, is it possible that the series is already sorted before completion of final pass?

4. If the answer of the above question 2 is yes, how to resolve the issue I order to ensure that there is performance improvement?

5. In all the discussed sorting algorithm, which is giving the best performance? Justify.

6. What is the difference between the internal and external sort? Which one is better justify?

7. Discuss the significance of internal and external sorting?

8. What is quick sort? How the quick sort functions explain?

9. What is the pivot element in the quick sort? How it function in the quick sort?

10. What is the best case performance of quick sort? How it varies? Discuss.

11. What is heap? What is the significance of heap?

12. What is merge-sort? Compute its complexity.

13. How the counting sort works? Discuss its advantages and disadvantages.

14. What is the use of heapify in the heap sort? Explain with the help of suitable example.

State True and False

I. Insertion sort is better than the selection sort.

II. Insertion sort is the example of internal sorting.

III. Insertion, bubble and selection sort will have the same performance.

IV. In the heap after removal of the items from the root, definition of the heap may be violated.

V. In the quick-sort, pivotal element places the item at the extreme right of the array.

VI. Heap sort will have always the complexity of O (n log n).

VII. Quick-sort is to be used where the time is the critical component.

Chapter 11

String

Chapter Objective

- Defining the string in C++
- Defining the various functions on string such as upper, lower, mid, left
- Searching of pattern in a string
- Discussing the brute force, KMP and Rabin karp algorithm.

11.1 INTRODUCTION

Strings are widely used in text based component of data structure. In languages like C and C++, it is represented with the help of character array. To manipulate the string, a number of functions already exist in the string header file. However, we may need more functions to accomplish many of the routine tasks such as extracting the string, searching for the specific pattern, conversion from one case to another case, etc.

In string, to achieve a single task many functions exist and each of them are representing various approaches based on their expert perspective. However, usage of these methods is governed by various factors including processing time, ease of use and resources capability.

Many string functions needed in our variety of needs have been described in the upcoming sub-section.

11.2 SIGNIFICANT FUNCTIONS OF STRING

To accomplish any task related to string, either we have to use the functions that are already defined in the string header file or we have to create our own functions. However, this section describes some of the complex functions that are widely needed for the string.

11.2.1 Converting the string case into other case

Due to variety of reasons, we need to convert the string text in upper, lower or sentence case. Therefore, function must exist that should facilitate the conversion from one case to another case. Various popular case conversion functions such as in lower case, upper case, sentence case, etc. have been defined as follows.

A. Upper case conversion

Upper case conversion function is used to convert the string characters from lower case to

upper case. To accomplish this purpose, we have used the ASCII values of character. If the character is falling within the range of small letter, it is converted into capital letter by subtracting 32 from the character value. Upper function has been enumerated in the following program.

```
// Program to convert the given string into upper characters
// function to convert the string into upper case
void toupper (char str [])
 {
 int i=0;
 while (str[i]! ='\0')// continue till end of the string is not encountered
 {
 if ((str[i]>=97) && (str[i] <=122))
 str[i] =str[i]-32;     // change it into corresponding upper case
 i++;
 }
 }
```

B. Lower case Conversion

Similar to upper case conversion, we can also convert the string into lower case, if the user has entered the string in capital letters. Corresponding to upper case conversion, we have used the ASCII table to convert the string character into lower case, if it does not fall under the category of lower case. The following program describes the conversion from the upper case to the lower case.

```
// Program to convert the upper case letter in lowercase
 //function to convert into lower case
 void tolower (char str [])
 {
 int i=0;
 while (str[i]! ='\0')
 {
 if ((str[i]>=65) && (str[i] <=92))
 str[i] =str[i] +32;
 i++;
 }
 }
```

11.2.2 Extracting substring

In addition to converting the string from one case to another case, we can also extract the group of characters from the string. These characters may be from the starting of the string, from the mid of the string or from the end of the string. In addition, there is a need to truncate the leading spaces entered by the user. Correspondingly, various functions to accomplish the above objective have been defined as follows:

A. Removing leading space

At the time of string input, user may inputs the spaces in a string. These leading spaces are undesired and affect the length of the string. Even worst, on various occasion does not give the correct result due to the presence of leading spaces.

To remove the leading space, we have defined the function trim (char []) that accepts the character array (string), and removes the leading spaces.

To remove the leading space, we have used the flag ‘start’ that denotes that this is the start of the string. To figure out that the present character is space, we have again used the ASCII table where the space has the value 32. Correspondingly, we have used the logic that if it is start of string and it is space, string should continue to move to the next character. Once any other character is arrived then it is copied. At the same time ‘start’ will be set to false, since the characters have started. Finally, the NULL character is also copied.

```
// Function to remove the leading space from a string
void trim (char str [])
  {
  int i=0, j=0, start=1;
  while (str[i]! ='\0')
   {
   if ((str[i] ==32) && (start))    // if space at the beginning
    {
    i++;
    //cout<<"blank found ";
    continue;
    }
    str[j] =str[i];
    start=0;      // Now if the space exist, it is in-between
    j++;
    i++;
   // cout<<"in trim"<<endl;
 //   cout<<endl;
```

```
}
str[j] ='\0';
}
```

Extracting the character from the start

We can extract the character from the start of the string. To accomplish this task, we have created a temporary array with the name 'temp'. We read the given string character by character upto the character intended by the user. For instance, if the user is intending to extract the first 4 characters, then it should start from the 0^{th} index and should continue upto the 3^{rd} index (Now total will be 4 characters). The entire program is enumerated as follows.

```
// Program to extract the first n character from the given string
// given string is st and number of characters are n
void left (char st [], int n)     {
    int i=0;
    char temp [20];
    while (i<n)//copying the string into temp
     {
     temp[i] =st[i];
     i++;
     }
     temp[i] ='\0';
     strcpy (st, temp); //copying the temp string back to the st
   }
```

B. Extracting the character from the middle

We can also extract the string from the middle. To accomplish this task, we have used the mid function. Here it is significant that the difference of indexing and the user supplied position should not have any mismatch. Consider that user is aiming to extract from the 2^{nd} position to 5^{th} position. In the array, index of the supplied argument will be 1^{st} and 4^{th}. Consequently, argument has to be indexed to hide the difference between the indexing and the user's perception of character location.

```
void mid (char str [], int start, int last)
    {
    int i=start-1, j=0;
    char temp [20];
    while (i<last)//copying the string into temp
     {
     temp[j] =str[i];
```

```
     i++;
     j++;
     }
     temp[j] ='\0';
     strcpy (str, temp); //copying the temp string back to the st
    }
```

In the above program, we have extracted the character from the start till the end into a temp string. Eventually, the temp string is copied into the original string by using the strcpy () function, defined in the string.h header file.

C. Extracting the string from the end

Extracting the string from the end is fairly complex relative to the other extraction methods discussed earlier. During extraction of a string from the end, we can use the logic described in the following program.

```
// Program to extract the 'n' character from the end of a string
void last (char str [], int n)
 {
 int len=strlen (str); // determine the length of the string
 int sp=len-n; //find out the starting position from where the…
// string extraction start
 char temp [30];
 int i=0;
 while (str [sp]! ='\0')
 {
 temp[i] =str [sp];
 i++;
 sp++;
 }
 temp[i] =str [sp]; // copying the NULL character
 strcpy (str, temp); // copying the extracted string back to the
original string
}
```

In the above program, first we determine the length of the string, which can be determined by the string function strlen () defined in the string.h. Afterwards, we have computed from last where we have to start extracting. To accomplish this objective, we have subtracted the 'n' value passed by the user from the string length already computed. For instance, if user is aiming to extract the last 4 characters from the string, first we determine the length of the

string, consider it is 15. Based on the available data, starting point of an array can be computed by subtracting 15-4=11. It signifies that we have to start from the 11th index and continue upto the end i.e. 14th index.

11.3 STRING MATCHING ALGORITHM

In the String matching, we compare the string with the supplied pattern to figure out, whether a particular pattern exist in the considered string or not. If the pattern exist then it should flash the suitable message, otherwise message for non presence of the pattern should be flashed. Popular string matching algorithms are discussed in the upcoming sub-section.

11.3.1 Brute force string matching

In the brute force string matching algorithm, we try to recognize the presence of a pattern in the given string. In this method, string is examined for the presence of a pattern from the starting index '0', if the first character of the string and the pattern matches then we compare the next character, if any of the character of string and pattern is not matching in that case further examination is terminated. Afterwards, examination starts from 1st index of the string with 0th index of the pattern to examine the presence of pattern. We continue till pattern is not exhausted or the string is not exhausted. If the pattern is exhausted it indicates that the pattern exist in the given string. Program for the brute force has been described as follows.

```
// Program to find out the pattern based on brute force algorithm
int bforce(char st[], char pattern[])
 {
int i=0, status=0;
int index;
while (st[i]!= '\0')
 {
j=0;
index=i;
  {
  while ((st[i]!='\0')&& (pattern[j]!='\0'))
{
if (st[i]==pattern[j])
{
i++;
j++;
}
} // end of inner while loop
```

```
if(pattern[i]=='\0')    // if the pattern is exhausted
{
status=1;          // change the status as successful
break;
}
i=index;    // set the i with the value as before inner while loop
i++;  // increment the index so that next element can be checked
}     // end of outer loop
return status;
} // end of function
```

We have used two loops in our program, outer and inner while loops respectively. Outer loop is governed by the length of the string whereas inner loop is governed by the pattern and string to be examined. If the matching is continued then the inner loop continues. After the exit of the inner loop, pattern is examined for whether it is exhausted or not. If it is exhausted, further checking is terminated.

If the pattern is not exhausted, in that case, we restore the value of i with the index that represents the value of i before the starting of the inner loop. Finally, 'i' is incremented so that checking can be started with the next element. This examination will continue till pattern or string is not exhausted. If the pattern is exhausted it represent the status as 1, otherwise status will be 0.

11.3.2 Knuth Morris Pratt (KMP)

Brute force algorithm is suffering from the major limitation that it is evaluating the string from the beginning and continues to match till mismatch does not occurs or the string is not exhausted. In case of mismatch, it will start checking the 1^{st} index (or the next address from where the last time it has started). It never consider that possibility of pattern existence will be less from the 1^{st} index once the matching already taken place upto 4^{th} or 5^{th} character. Consequently, it is termed that brute force algorithm never applies any intelligence. For instance, consider a case that we have started comparison from the beginning, first character of the string and the given pattern is matching, second is also matching, and third is also matching. However, mismatch has occurred at the fourth location. In this case there is no probability of string existence from the second element. Any probability of matching will start from the 5^{th} character only. However, this case has not been considered by the Brute force, instead it will start searching from the next (here in this case second) character.

The above limitation has been addressed by the Knuth Morris Pratt or (KMP). In this algorithm, we start from the beginning and compare the pattern with the string by also applying the intelligence. Working of KMP has been depicted as follows:

```
// Program to display the KMP
int KMP(char str[], char patt[])
{
int sl=strlen(str);
int pl=strlen(patt);
cout<<"String length ="<<sl<<endl;
cout<<"Pattern length ="<<pl<<endl;
int index=-1;
int i=0,j=0;
while(i<=sl)      // Check till the i is less then string
 {
j=0;
while((str[i]==patt[j]) && (j<=pl)) //  if  string  and  pattern  is
matching
{                        // at the same time, less than pattern
i++;              // go to the next character of string
j++;              // go to the next character of pattern
}
if(j>=pl)         // if pattern completely found in the string
{
index=i-pl; // first index of the string where matching found
break;
}
i++;  // if no matching found, go to the next character
}
return index;     // return the starting index if found, otherwise
return -1
}
```

In the above KMP algorithm, first we determine the length of string as well as of the pattern. The outer loop is governed by the total length of the string whereas the inner loop by the pattern length. If the matching character is found in that case index of both the string and the pattern is increased. If the pattern is completely checked (no more character exist) in that case further comparison is to be prevented, this is considered with the statement if (j>=pl). The used conditional statement signifies that the pattern is exhausted. To stop the further comparison ‘break’ statement is used that prevents any further comparison. Eventually, string index from where the pattern start is returned.

In case, if the pattern does not have the matching character next character of string is to be evaluated. This is achieved with the help of i++ statement available just before the closing of outer loop.

EXERCISE

1. What is the objective of Brute force string pattern matching algorithm?
2. How the KMP works? Determine the complexity of KMP.
3. What is the difference between the brute force algorithm and KMP?
4. How the efficiency is achieved with the usage of KMP.
5. How the pattern matching is different from the string matching? Discuss.
6. What are the various methods exist for pattern matching?
7. Take a String RAMAYAN, corresponding to the string, consider the patterns “RAM” and “MAY”. Carry out the dry run for pattern matching using Brute force algorithm and KMP. Also find out the comparison needed in case of Brute force and KMP algorithms.

Chapter 12

Hashing

Chapter Objective

- Defining hashing and significance of hashing
- Describing various hashing functions
- Describing the collision
- Describing the various collision handling techniques in hashing.

12.1 HASHING

In the linear search or the binary search, performance is the major challenge. Although linear search serve the purpose of searching, however performance is the major concern. Binary search yields better performance relative to the one offered by the linear search, however, in linear search the major limitation is related to the prerequisite that the series should be in sorted order. To overcome this challenge, another method known as hashing exist that allows to create and access the element on the basis of direct index, therefore, the performance of the algorithm is believed to be O(1), instead of O(n) or O(log n). To organize the element in the hashing, we use hash functions.

12.2 HASHING FUNCTION

Hashing functions are used to generate the unique index for the element. Correspondingly, elements are allocated to this new index/address. It is increasingly important that the function selected should generate the unique index, in order to ensure that the elements are placed at unique location within an array.

Hashing function is acting as the key driver in performance improvement. It allows us to place the item at appropriate location within an array, thereby entails us to search in minimum time, in the event of searching from an array. Although, various functions exist that entail us to create the unique address, however, we have limited ourselves upto prominent hashing functions, and same has been discussed in the upcoming section.

12.2.1 Division method

The key purpose of this function is to generate the unique index, in order to place the items at an appropriate location. In the division method, there are two numbers:

- Key that is used to generate an index.
- Total number of elements that are to be allocated (size) in a given table.

In this method, we divide the key by the size and find out the remainder. The computed remainder acts as an index of the given key. Since, it work on the remainder, therefore, it is also termed as remainder method. Syntax to generate the index is:

Index= K mod size.

Consider the series 34, 78, 47, 90, 12. Allocation of index using k mod n method. Here let n is 10 for which we have to allocate the address

Numbers	90		12		34			47	78
Index	0	1	2	3	4	5	6	7	8

Figure 12-1: Allocation of item using remainder method

For instance, consider that the total numbers of elements that need to be allocated are 100. Number (Key) for which we have to allocate the address is 34. In this case remainder will be 34 (34 mod 100). Consequently, 34 will be allocated the index 34. Now, consider the number 254. In this case, 254 mod 100, remainder is 54; therefore, 254 will be allocated the index 54. Therefore, a number may be sufficiently large; however, the remainder will be between 0 to (size-1).

In the figure 12.1, allocation of index using remainder method has been illustrated. In the considered series, for the first element, index is computed as '34 mod 10', therefore its index results in 4^{th} location. Correspondingly, index for the remaining items can be computed and would be result at the index 8, 7, 0, 2 respectively.

Hashing function does not always provide the unique address. For instance, consider a new number 154, applying the aforementioned rule; the address '4' will be generated. However, 4 is already occupied by the another number. If same index is allocated for more than one element then this phenomenon is known as **collision**. It is always undesired to have the collision, consequently, great care need to be exercised during the selection of the number for divisor. It is suggested to select the number of the size that should be a prime number. It is believed that if the size is prime number then it will generate less repeated index. However, collision is not altogether eliminated. To address the collision, numerous methods exist and some of the prominent methods have been discussed in the section 12.3.

12.2.2 Mid Square method

It is apparent from the above discussion that the division method results in excessive collision, since; we are directly operating on the key itself. Index can also be computed with the help of mid square method. In this method, instead of directly applying the rule, we

square the given key. Once the square of a key is known then we take the mid of this number based on the length of index. For instance, consider the number 311. First we square the considered key, it will result in 96721. From the result computed, we take the mid of this number. Number of digits to be extracted is governed by size of the table. In our case, consider the size of the table as 1000, and then the mid number is 672. It can be written as H (311) =672. The index generated for the aforementioned number will be 672. Now, it is apparent that the index generated is entirely different with the key. In this method, probability of generating the same index has been reduced significantly.

12.2.3 Folding method

In this method the numbers are folded so that computation can be performed. We can apply two types of folding methods:

- Shift folding
- Boundary folding

In the folding method, the number computed is divided by the size to determine the index of the key. It is considered as the hashing since it involves division and grouping of the number. Consider the number 254/32/6547/251, for this number we can apply any of the two types of folding methods. In the **shift folding**, the number can be shifted in other group as well. Consider the number is divided into the group of 03 digits, then the resultant number will be 254, 326, 547, 251. This is followed by adding these three numbers. In the case of on-going discussion, it will be 1374 (254+ 326+ 547+ 251). To compute the index, we can use the division method as (number mod size). If we consider the size as 100, then the index generated will be '1374 mod 100=74'. Index generated with the help of shift method is not unique instead more than one element may have the same index.

In the **folding method**, we divide the number into the groups of equal part and then fold the given number as with the folded paper. Afterwards, the number obtained is added. Result thus obtained undergoes the modulo operation. For instance, consider the number 123674 987, Here the number is divided into the group of three 123, 674, 987. We carry out the addition on the number thus obtained i.e. 123+674+987=1784. Finally, modulo size is carried out on the resultant number, in this number (key) is 1784 and the size we had considered was 100, therefore, the result would be '1784 mod 100=84'. Thereby, comparing the result of shift and folding method, it has been revealed that the index generated by both the method is different.

12.2.4 Extraction

In the extraction method, we extract only partial digit of a key to compute the address, whereas remainder digits of the key are omitted. For instance, if the number is 785/897/345, we can extract the first three digit or the first six, or four number or any other combination. This extracted number will be acting as a key and used to generate the index. It has been observed that the index thus generated yield good result and minimizes the collision, if carefully chosen. For instance, consider the employee_id assigned to an employee of an

organization, where few digits/character are common to all, whereas few digits are unique. Consider that the employee_id is CB_KNP_208010. In this case the initial number/character are common, therefore these common characters(CB_KNP) can be avoided as a part of the key. Correspondingly, in India, bank account number is allocated considering the location id that is dependent on state, district and on branch id. Now digits representing these items will be common for the majority of the customers, specifically in the same city and branch. Consequently, common digits should be omitted while generating the number to compute the index.

12.2.5 Radix transformation

In the radix transformation method, we change the radix of the given key. Afterwards, number is divided by modulo size. The number thus computed act as an index of the key. Consider, a key $(101)_2$ is the given number, since it is in base 2, therefore we will change it to the base of 10. The resultant number thus generated will be in decimal number, in the considered case it is 5. Afterwards, to compute the index (address) we compute the index as 5 modulo size=0, after considering size as 5 of the total element that can be stored. In our case, the address computed is 0.

12.3 COLLISION HANDLING TECHNIQUE IN HASHING

In the aforementioned hashing functions, we have witnessed that the hashing functions are not always generating the unique address. Instead, more than one number may fall in the same index. During the allocation of index, if more than one elements falls to the same index, then it is known as collision and undesired phenomenon. Consequently, collision leads to reduction in the performance of hashing and need to be resolved at appropriate level. Following are some of the popular methods that can be employed for handling the collision.

- Linear method
- Chaining method
- Coalesced Chaining method.

Methods used to resolve collision are termed as collision handling techniques. Following are some of the popular methods which can be employed to handle the collision:

A. Linear probe method

In the linear probe method, when two or more elements leads to the same index, in that case to allocate the unique index to the second element, we start probing the next empty cell from the point of collission. When the empty cell is discovered, element is allocated to this new available cell. Although, this method is simple and appears to yield good result, however, performance degrades upon increase in number. Items that are allocated at shifted cell are termed as clustered elements. These clustered number increase with greater pace when the shifting of items first start.

Consider a case in which a series is encompassing the numbers 21, 42, 14, 65, 32. All these

numbers are arranged using the 'x mod 10', where 'x' is the number whereas the 10 is size used to compute the remainder. Index generation of the various numbers including the 32 have been illustrated in the figure 12.2 a. On the arrival of 32, this will also result at the 2nd Index, however same is not to be permitted as the number at that index already exist. To resolve the collision, the linear probe method probes for the unallocated cell starting from the cell of collision. In the considered list, the next available cell is 3rd, therefore the number will be allocated at 3rd index and same has been illustrated in the figure 12.3 b. Now, consider that the next element is 53, in this case index will be '53 mod 10' that will be equal to 3. Since, the index 3 is already occupied, therefore, it will start probing linearly from the point of collision. Therefore, it will be allocated the index 6. Consequently, many item will not be allocated the actual index, hence it leads to further degradation in performance.

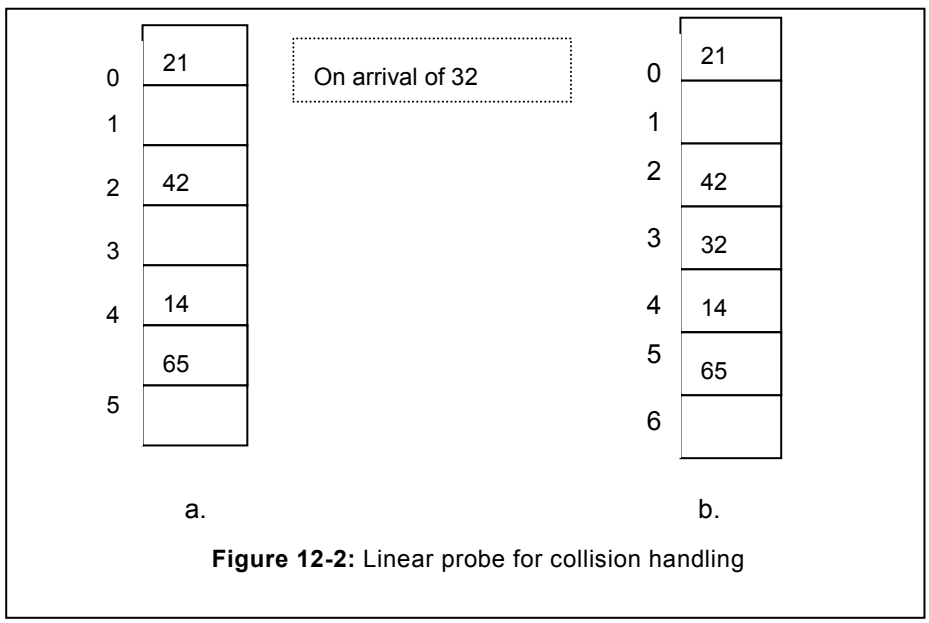


Figure 12-2: Linear probe for collision handling

B. Chaining method

In the chaining method, when the second item result in the same address, a link list is created at that index and termed as chain- All the elements that are leading to the address where the element is already stored, such elements are stored at the node of a link list. On the arrival of any other element at the same address, again a new node is created where the item is stored. Therefore, elements are arranged in an un-order form. During the accessing of the element, we continue to search the element till the item is not found or the considered list is not exhausted. This method of collision handling results in better performance relative to the linear probe method. However, major overhead lies during the insertion time (Chaining) since creation of node requires time.

Consider the list as illustrated in the figure 12.4, a node will be created similar to link list node

that will have the link at the starting index. All the other nodes that will result in the same index will have the node connected with the existing link list. Chaining method has been illustrated with the help of figure 12.3.

Consider the series of figure 12.4, on the arrival of 52, it will be allocated at the second index, since that index is already occupied therefore, collision will occur. To resolve it, a link node will be created where this new item will be allocated. Correspondingly, on the arrival of new item 12, a new node will be created at the index 2, where the number 12 will be placed refer figure 12.4.

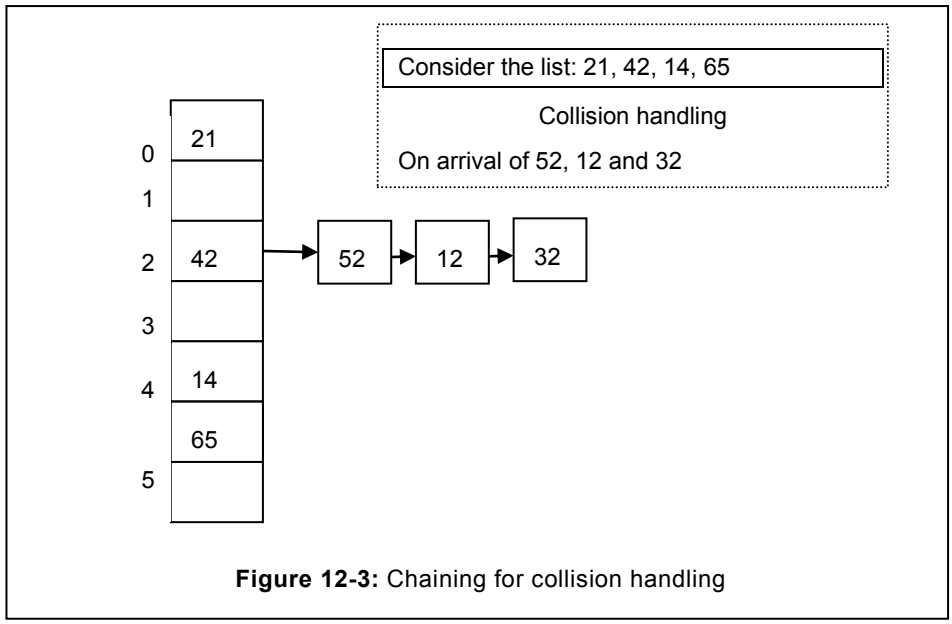


Figure 12-3: Chaining for collision handling

C. Double chaining method

In the chaining method, elements that are leading to the same index are connected together with the chain. In this method; a new node is created for all those elements that result in the same index, similar to that of link list. While arranging items in this method, if more than two variables need to be assigned the same index, double chaining method is used. In double chaining method elements are stored in sorted manner instead of order of their arrival. Sorted link list method leads to greater improvement in performance.

EXERCISE

Descriptive Type Questions

1. What is hashing? How it is different from the other methods of finding out the index?
2. Can be defined the hashing as search? Justify
3. What are the various methods of Hashing? Define advantages and disadvantages of each of them?
4. What is collision? How it occurs in hashing? Justify with the help of suitable examples.
5. Discuss the various methods to resolve the collision? Discuss their advantages and disadvantaged.
6. Study the various cases where hashing has been used for the implementation. Enumerate its advantages and disadvantages.
7. What are the major limitations of the linear probe method? How it is has been resolved by the subsequent methods.
8. Differentiate between the linear probe method and chaining method. Discuss their strength and weakness.
9. How the limitations of the chaining methods have been overcome?

Chapter 13

Elementary Graphs

Chapter Objective

- Defining the graph and its terminology.
- Discussing the various types of graph
- Representing the graph
- Traversal in a graph
- Minimum spanning tree in a graph
- Shortest path in a graph

13.1 INTRODUCTION

Previous chapters were entirely dedicated to linear data structure or the tree data structure. Tree forms one-to-many relationship from parent to the child. At the same time, it does not form the cycle. Even though, tree is one of the widely used data structures, yet does not fulfills many of the objective. For instance, consider the case if the data need to be send to the shortest path, at the same time, acknowledge of success or failure need to be returned. In such requirement, there is a specific need of a path that should have multiple connections. At the same time, should have way for the return. To fulfill this objective, we use a new data structure known as Graph.

13.2 BASIC DEFINITIONS

A graph consist of vertex (similar to node of a tree), various vertices are connected together with the help of edges. In any graph, we have collection of vertices and edges.

A. Graph

A graph 'G' can be defined as a data structure encompassing of finite set of non-empty vertices (V) and set of edges 'E'. Graph 'G' is represented by a tuple (V, E). In a graph, vertices have names and can represent the other properties. In graph, edges are used to represent the relationship between two objects. For instance, distance between the two vertices. Figure 13.1 consist of 08 vertices namely 'a', 'b', 'c','d', 'e', 'f', 'g', 'h'. Each vertex is connected with pair of vertices.

B. Directed Graph

In a graph, if we denote the direction in edges then it is termed as directed graph, it is illustrated in the figure 13.2. Such kind of graph is also known as ordered graph. In the directed graph of figure 13.2, the edge ‘ad’ is not same as ‘da’. We can distinguish the edges from one another with the help of the direction given.

Consider a case, if there is a need to visit from vertex ‘a’ to ‘f’. In this case, one path is ‘ac’, ‘ce’, ‘ef’ and the other possible path exist is ‘ab’, ‘bf’. Whereas, if we aim to reach ‘a’ from ‘f’

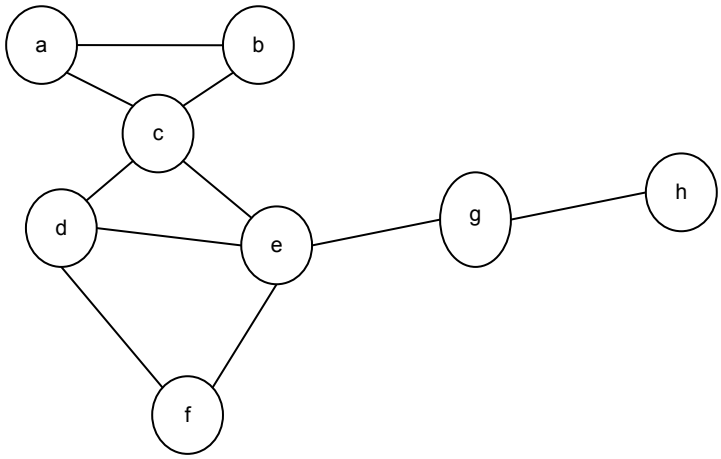


Figure 13-1: Graph with vertices and edges

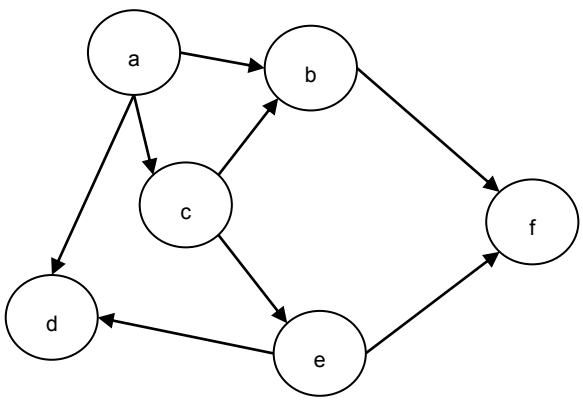


Figure 13-2: Directed graph with six vertex

then it is not possible. Since, edges directed towards the destination ‘a’ do not exist.

In a graph, if the edge V_i and V_j exist, at the same time the edge V_j to V_i also exist. In a graph, if all the edges follow the above type of edges then such type of graph is known as symmetric graph. Otherwise it is known as asymmetric graph. The graph illustrated in the figure 13.2 is not symmetric since the edge V_i and V_j exist while the edge V_j to V_i does not exist. Symmetric graph has been illustrated in the figure 13.3, here we can see that graph is having 03 vertices, each are attached with the help of edges. Each node is having two pointers from a single node, one is falling on the node where the other one is heading away from the node.

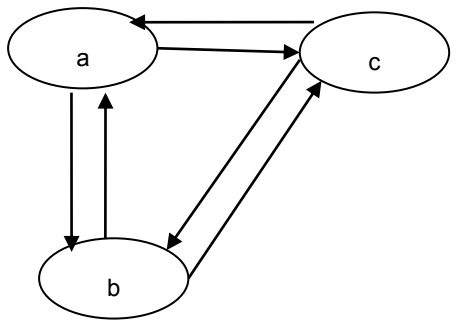


Figure 13-3: Symmetric graph

Such type of graph is termed as symmetric graph.

C. Undirected graph

In the figure 13.1, direction of edges is not denoted. In such type of graph order does not have any significance, therefore they are known as unordered. When the edges of a graph are unordered pairs then the graph is known as undirected graph. In the undirected graph, the edges such as ‘ac’, ‘ca’ is undistinguishable.

D. Degree

In a graph, degree of the vertex is governed by the number of edges and the type of graph(symmetric or asymmetric). In the case of undirected graph, all the edges are the part of the degree. Whereas, in the case of directed graph, degree is categorized into in-degree and out-degree. Vertex with the edges directed towards it is known as In-degree. In the out-degree, all the edges are directed away from the vertex. In the given figure 13.2, in-degree and out-degree of each vertex has been enumerated in the table 13.1.

E. Incidence

In a graph, the edge connecting the adjacent vertex is known as incidence. Consider the graph of figure 13.1, the two vertexes ‘a’ and ‘b’ are connected together with the help of an edge. This edge is known as incidence of ‘a’ and ‘b’.

F. Adjacency

In an undirected graph, if an edge 'e' is connected directly with the help of two vertices V_i, and V_j then both the edges are known as adjacent. Consider the graph illustrated in the figure 13.1, the edge 'ab' and 'ac ' both are adjacent to 'a'.

Table 13.1: Degree of graph in figure 13.2

Vertex	In-degree	Out-degree	Degree
A	0	2	2
B	2	2	4
C	1	2	3
D	2	0	2

G. Path

Path in a graph is a set of vertices used to reach from the source to the destination. For instance, consider the graph of figure 13.2 the path between the sources 'a' to the destination 'f' is 'ac-> ce-> ef'.

The graphs in which the path encompasses of all the vertices of a graph then it is known as Hamiltonian path.

H. Cycle

In any graph, if we start from the source vertex and return to the vertex from where we have started(source vertex), then we term that the cycle is formed. In the directed graph, we may have a path from source to destination that may not form the cycle. However, in the undirected graph probability that the cycle is formed is increasingly higher, since, undirected graph does not involve the direction.

I. Weighted graph

Graphs in which edges are assigned the weight, such kind of graphs are termed as weighted edges. Weighted graph is illustrated in the figure 13.4. In the illustrated figure, weight of the edges E_{12}=7, and E_{13}=5.

13.3 REPRESENTATION OF GRAPHS

Graphs that were created in the earlier sections are in the pictorial form. Same cannot be meaningful for the computer. We need to evolve the strategy in order to ensure that the graphs can be represented in the computer; this is known as representation of a graph. Following are the various methods which can be used to represent the graph.

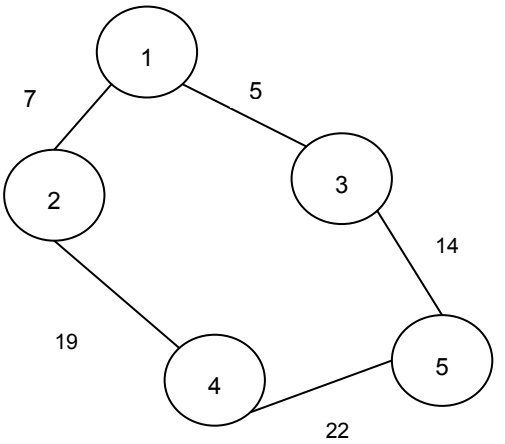


Figure 13-4: Weighted graph

13.3.1 Adjacency matrix

Adjacency matrix uses the matrix form to represent the edges between the two vertexes. If the vertex V_i is connected to the Vertex V_j with the help of a link known as edge then we represent this edge as '1'. In case, if it is not connected then it is represented as '0'. We have enumerated the adjacency matrix of the graph of figure 13.1 in the following table.

Table 13.2: Adjacency matrix of the figure 13.1

	a	b	c	d	e	f	G	h
a	0	1	1	0	0	0	0	0
b	1	0	1	0	0	0	0	0
c	1	1	0	1	1	0	0	0
d	0	0	1	0	1	1	0	0
e	0	0	1	0	1	1	1	0
f	0	0	0	1	1	0	0	0
g	0	0	0	0	1	0	1	0
h	1	0	0	0	0	0	0	0

In the undirected graph, if the edge exists between the vertexes A_i to the Vertex A_j then 1 is to be placed in the edge A_{ij} as well as the edge A_{ji}, since direction does not have any significance in undirected graph. Adjacency matrix obtained for the undirected graph is symmetric that implies that if the edge exist from the vertex A_i to A_j then it also applies for the

A_j to A_i. Degree of any vertex can be determined with the help of adjacency matrix. **Degree of any node is the sum of all the 1 in a particular row of the vertex**. In table 13.2, we can determine the degree of vertex 'c' as 4. Correspondingly, degree of other vertex can be determined.

In the directed graph, adjacency matrix is significantly different from that of undirected graph. In case of directed graph, only those edges are considered that represent the edge between the two vertices of the graph. For instance, if we say 'ab' it means we assume that the edge will be directed from 'a' to 'b'. if we consider 'ba' it is significantly different from the 'ab' where the direction is from source 'a' towards the destination 'b', whereas in the 'ba' edge is from the 'b' to the 'a'.

Table 13.3: Adjacency matrix of figure13.2

	a	b	c	d	e	f	g
a	0	1	0	1	0	0	0
b	0	0	0	0	0	1	0
c	0	0	0	0	1	0	0
d	0	0	0	0	0	0	0
e	0	0	0	1	0	1	0
f	0	0	0	0	0	0	0

In the directed graph, an edge may exist from the vertex V_i to V_j, however, that may not be true for the edge V_j to V_i. Therefore, adjacency matrix for the directed graph is not symmetric.

Degree of directed graph can be defined as sum of in-degree and out-degree of any vertex. In-degree of any directed graph is the total number of '1' in that column, whereas out-degree in a directed graph is the total number of '1' in the row. Consequently, degree of any node can be computed as:

Degree of vertex (V_i) = in-degree of vertex (V_i) + out-degree of vertex (V_i)

13.3.2 Adjacency List

In the adjacency list, we create the link list for the vertex and all other nodes adjacent to it. For instance, each vertex represents the link list, nodes of this vertex is represented by the adjacent vertex. Therefore, all the nodes that are adjacent can be accessed from the particular node is taken into the single adjacency list. Adjacency list of figure 13.1 is illustrated in figure 13.5. Adjacency list can be created using the link list program. Similarly, adjacency list for the directed graph can be created like undirected graph. However, there is a minor difference that is attributed due to the involvement of directional component of the

directed graph. Adjacency list of the directed graph as illustrated in the figure 13.2 has been illustrated in the following figure 13.6

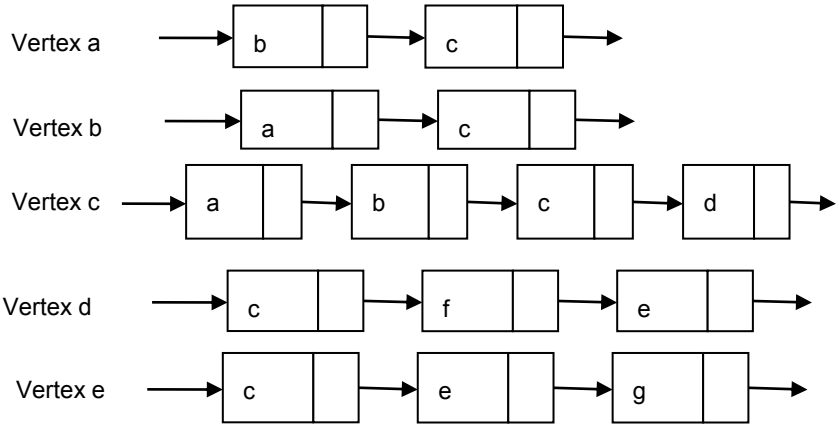


Figure 13-6: Adjacency list of figure 13.1

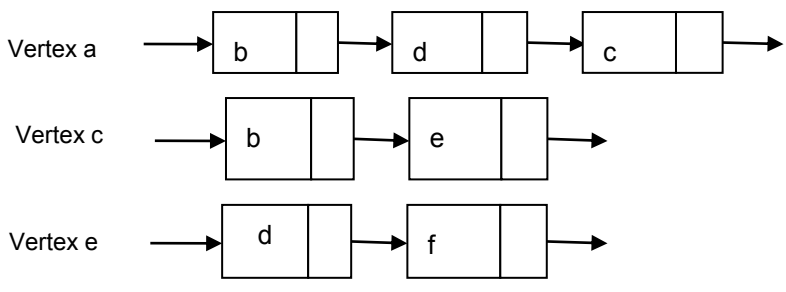


Figure 13-5: Adjacency list for the directed graph of figure 13.2

In the adjacency list, vertex 'b' and'd' does not have any adjacency list consequently, they will point to the NULL.

13.3.3 Incidence matrix

Consider the graph 'G' with "n" vertices and "ne" edges but does not have any self-loop. Consider the matrix A= (a_{ij}) with n x ne, where n is the number of vertices and ne is the number of edges as follows.

A_{ij}=1 if "ej" is incident to the vertex "v_i"

=0, otherwise

Such a matrix is known as the incidence matrix of a graph. **In other terms incidence matrix**

are the edges that are incident on a particular vertex. Therefore, we write all the incidence edges and all the vertex. If the incidence edge belongs to that particular vertex then 1 is placed otherwise 0. Consider the graph of the figure 13.7, In this graph, incidences are (1, 2), (2, 3), (2, 4) (3, 5) (3, 6) (3, 7). Incidence of figure 13.7 has been depicted in the following table 13.4.

Adjacency incidence matrix for the directed graph is different from the undirected graph. For any edge a_{ij} of a graph, incidence matrix is derived as per the following rule.

A_{ij}=1 if "e_i" is the incident out of V_i.

=-1 if "e_i" is the incident into V_i

= 0 if "ei" is not an incident to V_i.

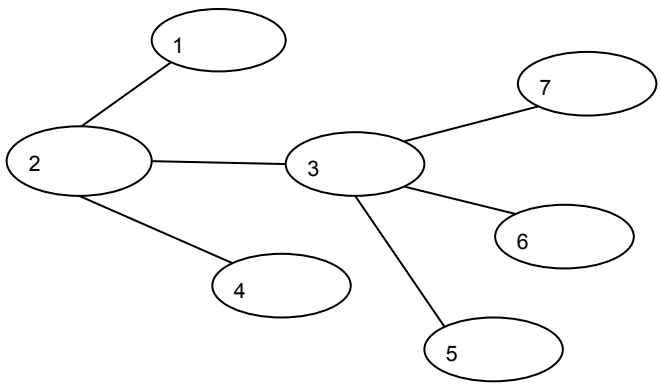


Figure 13-7: Sample graph

Table 13.4: **Incidence matrix for the figure 13.7**

	(1,2),	(2,3)	(2,4)	(3,5)	(3,6)	(3,7)
1.	1	0	0	0	0	0
2.	1	1	1		0	0
3.	0	1		1	1	1
4.	0	0	1	0	0	0
5.	0	0	0	1	0	0
6.	0	0	0	0	1	0
7.	0	0	0	0	0	1

13.4 MINIMUM SPANNING TREE

A graph “G” has been illustrated in the figure 13.8, this graph is forming the cycle. Graphs have drawn lot of attention in recent time. Many times it is compared with the tree due to the parent child relationship. But there is significant difference between the graph and tree. One significant difference between graph and tree is pertaining to the cycle. In graph, cycle may be formed; however, cycle cannot be formed in case of tree. Consider the graph of figure 13.8 (a), corresponding tree has been drawn in the figure 13.8(b). Tree of the figure 13.8(b) is similar to that of the tree that has been discussed in the earlier chapters. But the major difference is that it does not have explicit root. In the undirected graph, parent child relationship is also not apparent. Consequently, we can term tree as a graph which does not have a cycle.

A sub-graph of figure 13.8(a) that comprises of all the vertex and not forming the cycle has been illustrated in the figure 13.8 (b).

A minimum spanning tree is the sub-graph (tree) of a graph that forms a tree by selecting the minimum weight of a sub-graph. There are various algorithm existing that can be used in creating minimum spanning tree. Each of them create the minimum spanning tree in their own way. Some of the prominent algorithms have been discussed in the upcoming sub-section.

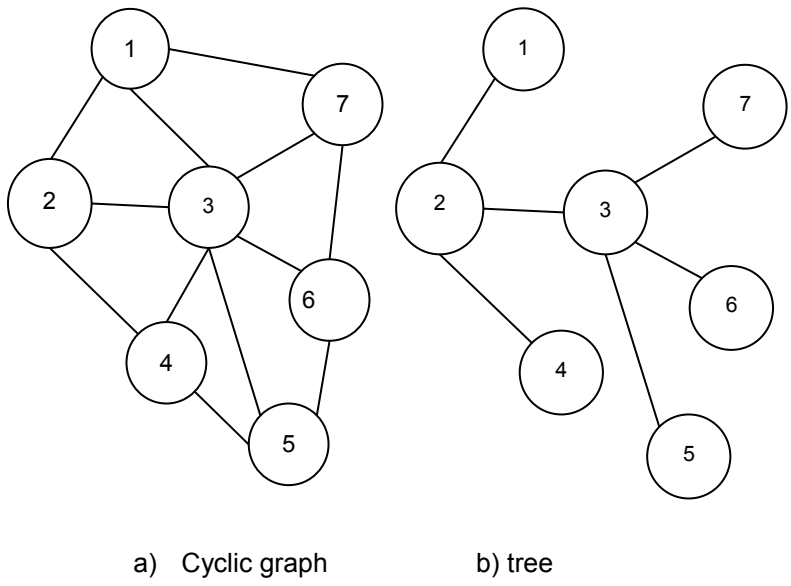


Figure 13-8: Graph to tree conversion

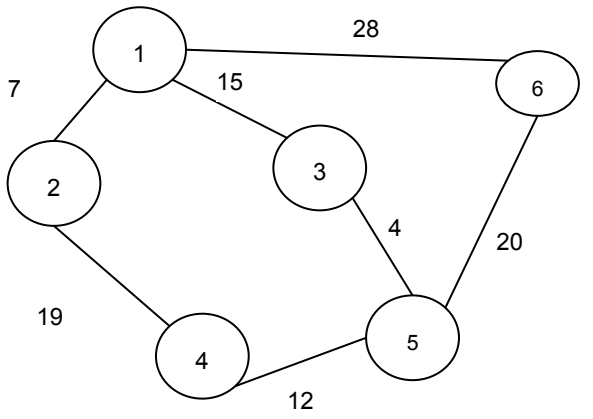


Figure 13-9: Weighted graph

13.4.1 Kruskal's Algorithm

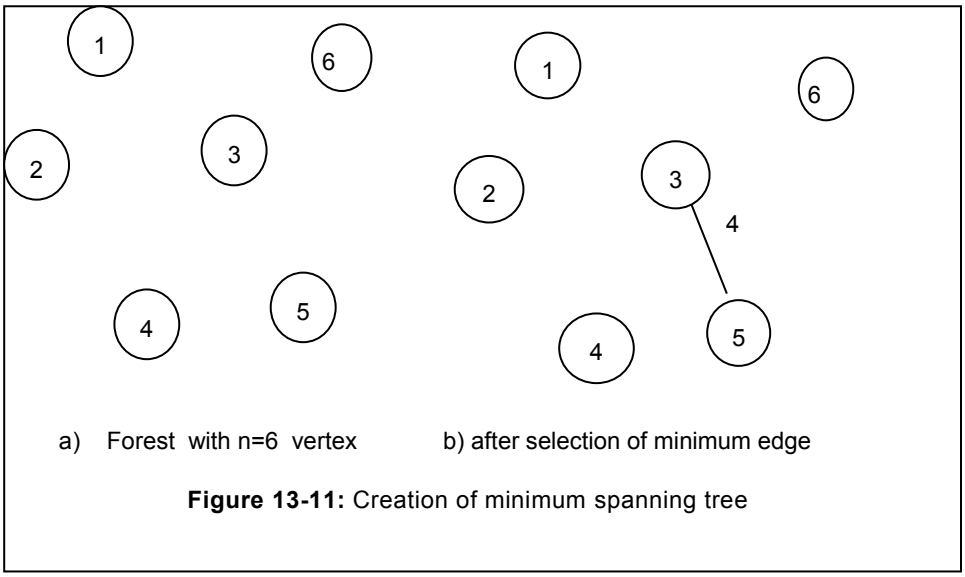
In the Kruskal's algorithm, to determine the minimum spanning tree, weight are considered in the increasing order. A graph having 'n' vertex will have a forest of 'n' tree. In each step, one edge is considered that is minimum from all available edges. Minimum edge is considered only if it is not forming the cycle in a tree.

In the graph of figure 13.9, the edge that will have minimum weight will be selected. In the considered graph the edge of vertex $V_{3,5}$ is having the minimum weight, i.e. 4. Consequently, it will be selected first. Same has been illustrated in the figure 13.10 (b).

In the next step, the other minimum edge is considered from the available edges. Now, the minimum node among the available set is 7, that is the edge connecting the vertex $V_{1,2}$. Resultant spanning tree will be as illustrated in the figure 13.11. Similarly, other edges are considered in the increasing order. The other edge that will be considered is $E_{1,3.}$, weight associated with this vertex is 15. After selection of 15, the next edge is $E_{2,4}$ which is 19, but if this edge will be selected the cycle will be formed, therefore, this edge will be dropped. The next weight is 20 that is related to the edge "$e_{5,6}$" is selected. Even though, other edges are left but they cannot be selected as they will result in the cycle. Finally, weight associated with this graph (MST) can be computed by adding all the weight of the MST thus obtained.

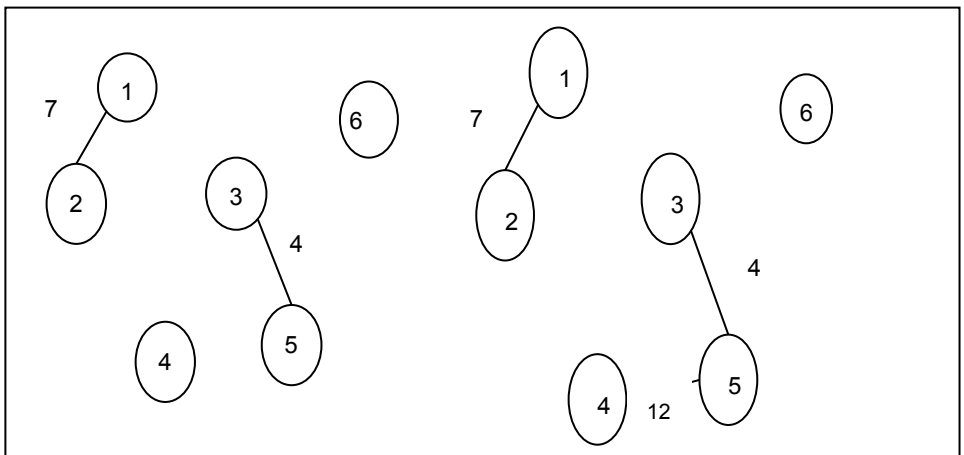
Total weight = 4+7+12+15+20

=58



a) Forest with n=6 vertex b) after selection of minimum edge

Figure 13-11: Creation of minimum spanning tree

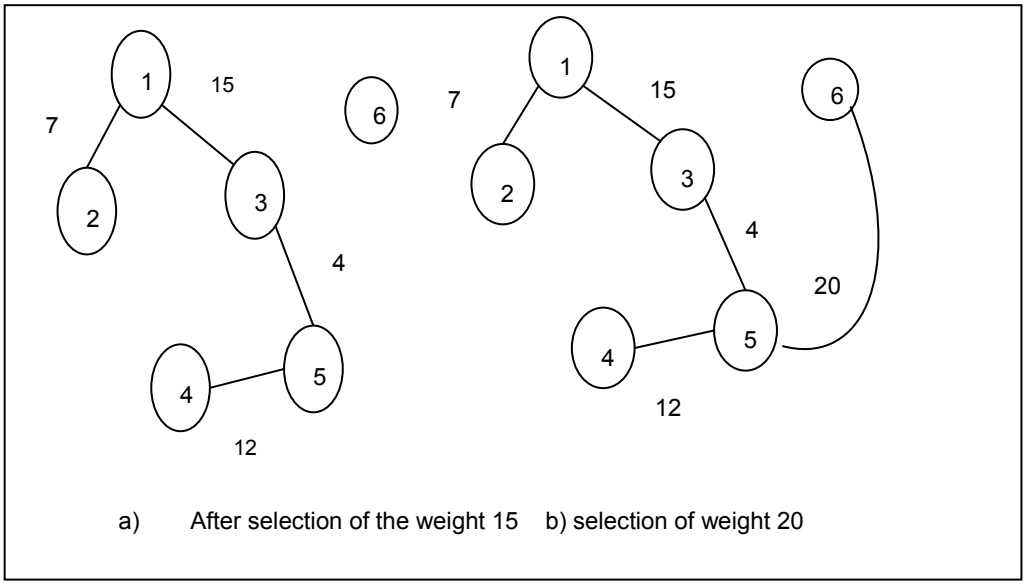


a) After selection of 7 b) After selection of 12

Figure 13-10: Creation of tree

13.4.2 Prim's Algorithm

Prim's algorithm is the other method utilized to construct the minimum spanning tree (MST). Unlike the Kruskal's algorithm, in the Prim's algorithm, we select any of the edges from the graph. This is known as arbitrary selection of edges from the graph. From this edge, we can select the edge from the available edges. However, the edge with the minimum weight is selected. This process will continue till all the vertex are not traversed. Functioning of the prim's method is depicted in the following figures.



a) After selection of the weight 15 b) selection of weight 20

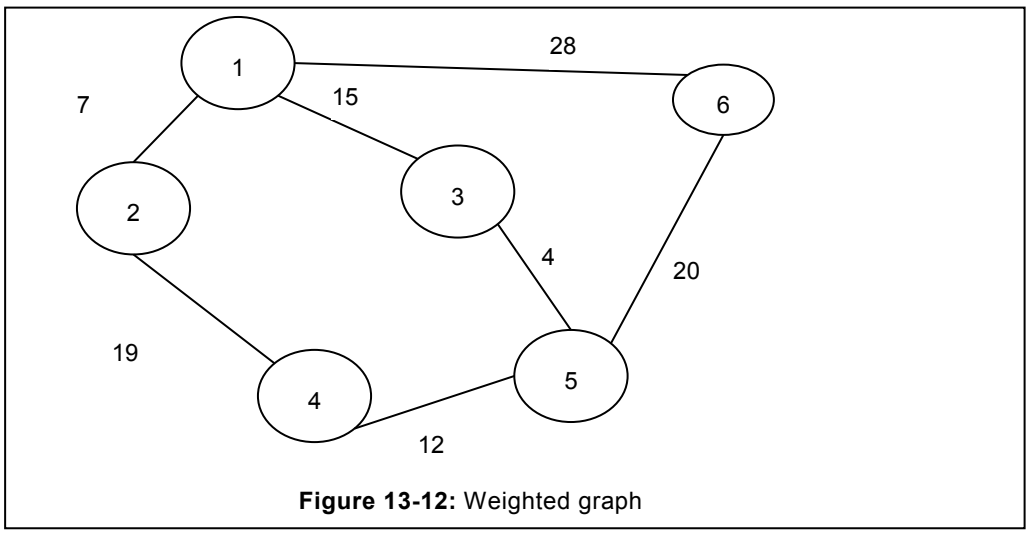


Figure 13-12: Weighted graph

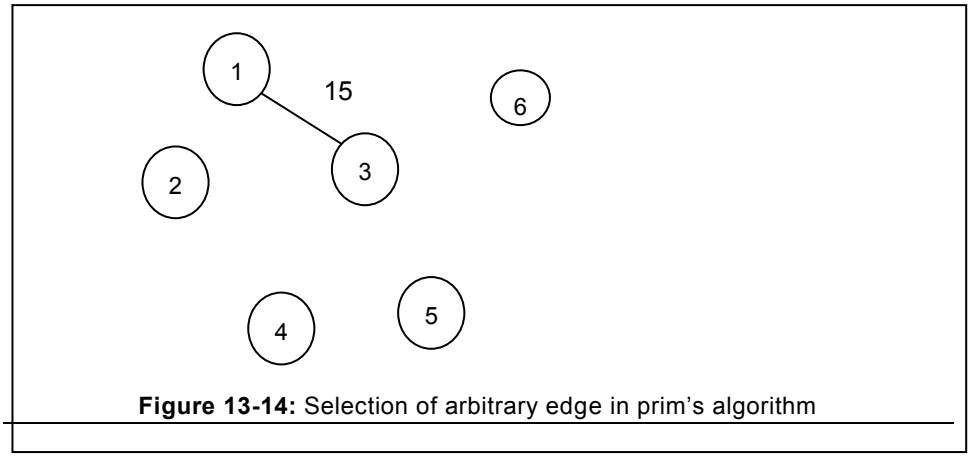


Figure 13-14: Selection of arbitrary edge in prim's algorithm

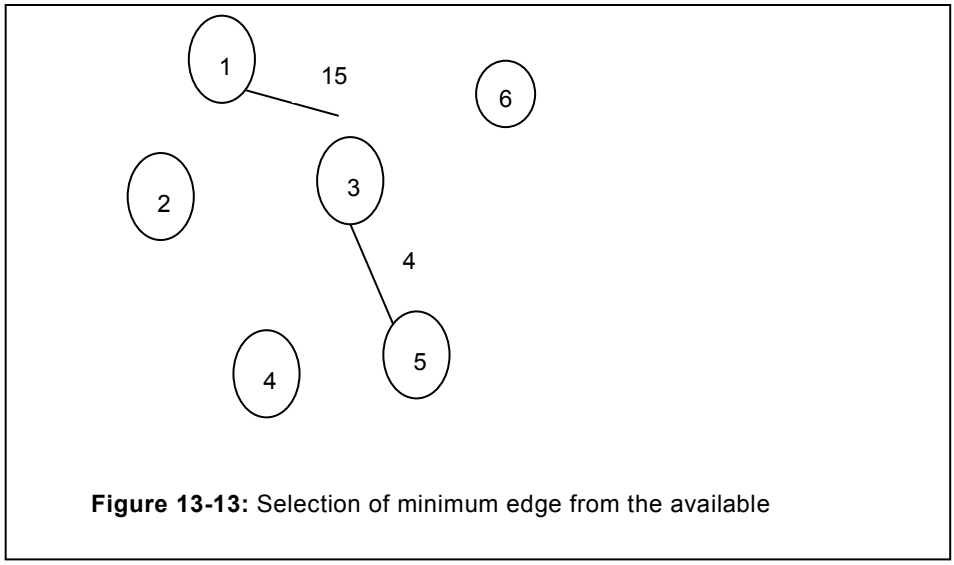


Figure 13-13: Selection of minimum edge from the available

Consider the graph of figure 13.12 from which we have to determine the minimum spanning tree. In this method, we will opt for any arbitrary edge, in our case we have considered the edge "e_{13}" and having the weight of 15.

Now the available edges of the graph are E_{12}, E_{16} and $E_{35,}$ these edges are having the weight 7, 28 and 4. Since, in the available weight 4 is minimum that is of the edge $E_{3,5}$ therefore, the next edge that will be selected isE_{35}. Consequently, the new spanning tree will be as illustrated in the figure 15.14. Afterwards, we will check for the available edges which are E_{12}, E_{16}, E_{54}, E_{56} having the weight 7, 28, 12 and 20 respectively. Among the available edges and their respective weight, the edge E_{12} is having the minimum weight i.e. 7.

Now the available edges are E_{56}, E_{45}, E_{12}, E_{16}. In the available edges weight associated to them are 20, 12, 7 and 28 respectively. Among the available weight edge E_{12} is with the

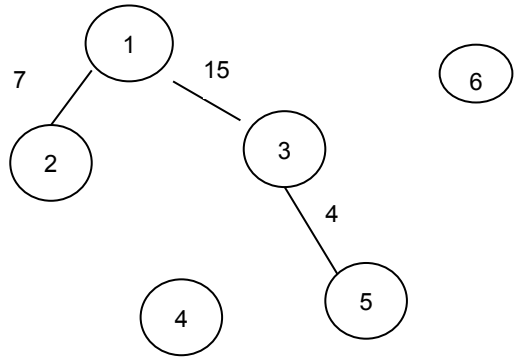


Figure 13-15: Selection of minimum edge from the available

minimum weight. Consequently, the next edge E_{12} will be selected. Same has been illustrated in the following figure.

After this step, now we are left with the E_{56}, E_{45}, E_{24}, $E_{16,}$ Weight associated to them are 20, 12, 19 and 28 respectively. Therefore, the minimum weight available is 12 and the edge associated to it is E_{45}.

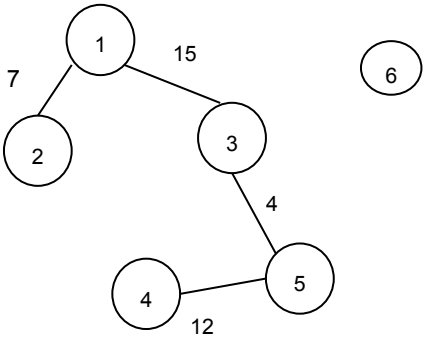


Figure 13-16: Selection of minimum edge from the available

Afterwards, leftover edges available are E_{56}, E_{24}, $E_{16,}$ among these edges E_{24} cannot be considered since it does make the cycle. Therefore, edges left are E_{56}, E_{16}, and the weight associated to them are 20 and 28 respectively. Among them, minimum is 20 therefore, it will

be selected. Finally, minimum spanning tree is illustrated in the figure 13.16. Since, edges that are left are forming the cycle, consequently they cannot be considered. Total weight associated to this graph can be computed as:

MST =4+7+12+15+20

=58

It is apparent from both the method that the minimum spanning tree is having the same value 58. Consequently, it signifies that the MST does not change with the change of algorithm.

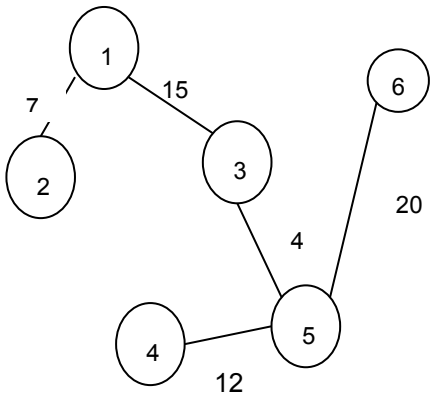


Figure 13-17: MST of the given graph (using prim's algorithm)

13.5 TRAVERSAL IN A GRAPH

Visiting each and every node once in a graph is termed as traversal. In the graph traversal, none of the node should be visited more than once and none of the node should remain unvisited. In graphs, there are two prominent traversal techniques that are widely used. Each of them has been discussed in the upcoming section.

13.5.1 Depth first search

In the depth first search method, we start from the one vertex and continue to traverse up-to the deepest level. At a time, only side is selected for the traversal and reaches to the deepest level. This allows to access vertex based on their depth.

Consider the case of the figure 13.18, in this graph, we can start from any of the vertex, however, we are starting from the vertex-1. This is followed by all the nodes at the next level of this node. In this case, they are 2, 5, 6. Afterwards, we would like to traverse all the vertex of the other vertex directly to the vertex 2. In the figure 13.18, it is 3, since; visit to 6 is already accomplished. Afterwards, we again consider, the other vertex directly connected to vertex-1. In the considered figure, it is 4. We would not consider the node 6, since; it is already marked as visited. Therefore, the entire depth first traversal is 1, 2, 5, 6, 3, 4. Readers are requested

to refer the depth first search method explained and coded in tree traversal (section 7.4).

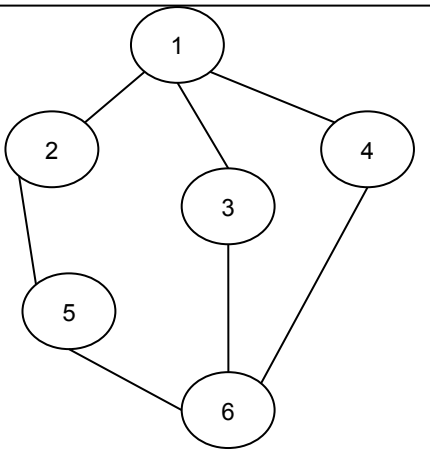


Figure 13-18: Depth first traversal

In the breadth first search method, we start from the vertex v_i, once all the neighbors of the vertex V_i are visited then we visit the neighbour of neighbours, and will continue till each node is not visited. During visit of the node due care is to be exercised to avoid visiting any node twice. Breadth first search is similar to the BFS of binary tree. While traversing the vertex Vi, we identify all the neighbours of this vertex, once the neighbours are identified then they are inserted in a queue. Therefore, the edges inserted at the end are removed at end. To enable the level by level traversal, we can use two queue. Consider the graph of figure 13.18, the breadth first traversal can be pursued as follows:

i) Assume that we have visited first vertex and it is V1.

ii) Neighbours of the V1 are V2, V3, V4 therefore all of them need to be inserted into a queue.

iii) After insertion, one element from the queue is removed (de-queue), the item available at first position of the queue is removed first. In this case, the first vertex is V2 therefore it is removed and marked as visited.

iv) Neighbours of V2 are inserted at the rear side of the queue, in this case it is V5. Therefore, the resultant queue will be V3, V4, V5

v) This is followed by de-queue of the neighbors already inserted i.e. V3. Its neighbor is V6, therefore, it would be inserted in the queue. Consequently, the resultant queue would be V4,V5, V6.

vi) In the next step, we would de-queue from the queue, consequently, we would obtain V4. Neighbour of this node is V6 that is already visited therefore, no more node will be added. Consequently, the resultant queue would be V5, V6.

vii) In the next step, again the next vertex is de-queued, in this case it is V5. Neighbour of the V5 is only V6 which is already visited. Consequently, no more nodes will be added in the existing queue which will have only vertex V6.

viii) Eventually, we would de-queue the V6 from the queue. Since, it does not have any neighbour, therefore, no more vertex can be en-queue. Since, the queue is also empty, therefore, the entire operation will be terminated. The resultant queue of the breadth first traversal is: V1, V2, V3, V4, V5, V6.

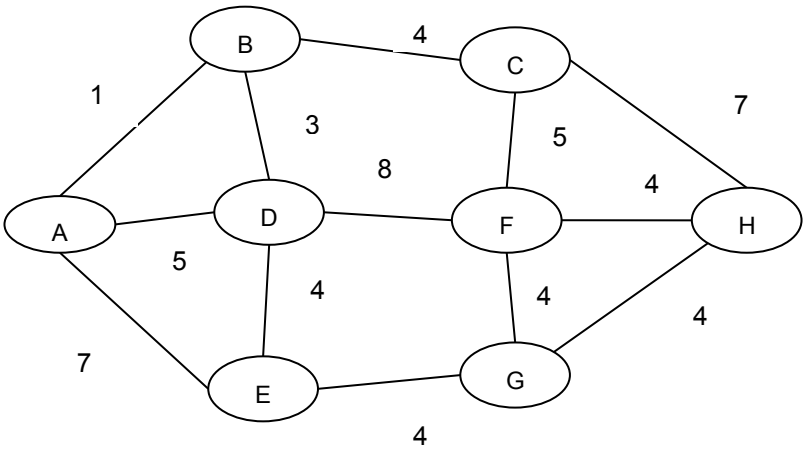


Figure 13-19: Graph to determine shortest path from A to H

Readers are requested to refer the section 7.4, for the implementation of the program. However, one more data member visited is to be included. By default it would have the value of '0' that represents the non-traversal of the vertex. Once the vertex is visited, its value will be set to 1. This will prevent the repeated visit to the same vertex.

13.6 SHORTEST PATH ALGORITHM

Shortest path algorithm determines the shortest path in a graph from the selected source to any given destination. Many a time, we need to learn the shortest distance from the source to the destination to minimize the cost of traversing or to reduce the latency.

To determine the shortest path from source to destination numerous algorithms exist. These algorithm can be significantly useful in reducing the overhead/cost associated in traversal from given source to the destination. Some of the prominent algorithm has been described in the following sub-sections.

13.6.1 Dijkshtra's Algorithm

Dijkshtra's algorithm is profoundly popular and widely used to determine the shortest path from the given source. This algorithm is applied in wide applications. For instance, in networking where we need to select the shortest path among the available list of paths.

Efficiency of shortest path algorithm is determined with the help of

- Accuracy of algorithm
- Efficiency of the path selected.

The principal objective is to figure out the shortest path in order to minimize the latency.

In the Dijkshtra's algorithm other than the source, we consider all the nodes are at the infinite distance. Consequently, infinity is assigned to all nodes. Consider the figure of 13.19; we have to determine the shortest path from the source A to the destination H. Complete procedure has been depicted as follows:

i. Find out all the adjacent nodes from the source.

ii. Figure out the instance from source to the other adjacent node and select the one with minimum value. Consider it as B. Mark this node as visited and the distance value will be 1.

iii. Find out the total distances of all the open nodes. Here, open nodes from B are C and D, with total distance would be 5 and 4 respectively. Similarly open nodes from A are D and E. Total distance from D and E is 5, 7 respectively.

iv. Among the available options, smallest is 4 therefore next milestone D will be selected. Value of D will be 4.

v. Now the backward path, i.e. from A to D will not be considered since, shortest path from A to D is already known.

vi. Now the open path is C, F, E.

EXERCISE

1. Define graph. How graphs are different from the tree? Explain by citing the suitable example(s).
2. What do you understand by Hamiltonian graph? Define the pros and cons of Hamiltonian graph?
3. Explain the degree, traversal, cycle in a graph? How it is different from the tree?
4. What do you understand by traversal in a graph? Define them with the help of suitable example?
5. What do you understand by graph representation? Why it is needed? Justify.
6. What are the major differences among the various graph representation techniques? Describe them with the help of suitable example(s).

7. Do the graph representation of directed and undirected graph will be same? Justify with the help of suitable examples.
8. How to compute the degree of symmetric and asymmetric graph? How to compute the degree with the help of graph representation method?
9. How the Kruskal's algorithm differ from the prim's algorithm? Describe with the help of suitable examples.
10. What are the avoidable conditions in Kruskal's algorithm? How these conditions can be fulfilled while defining the algorithm?
11. What do you understand by shortest path? Name the various algorithms existing for the shortest path? Compare the performance of each of them?

Chapter 14

Recursion

Chapter Objective

- Defining recursion
- Working with Active record
- Base condition
- Defining types of recursion
- Discussing the prominent recursion examples.

14.1 RECURSION

Computer programs are used to solve the problem undertaken. Program comprise of functions, which are set of statements that need to be executed. To execute the set of statements in function various methods exist. Broadly, the statements executed falls into the following categories:

- Iterative
- Recursive

In the iterative method, loops are used to execute the statement(s) more than once. Another approach to execute the group of statements is profoundly popular and known as recursion. It is equally significant and widely used.

14.2 DEFINING RECURSION

In recursion, function calls to itself number of times. In other terms, function solves the problems in its own terms. Recursion like other function accepts the parameters, compute the value of variables (or complete the task undertaken) and return the computed value to the caller. As the definition of recursion suggests that function calls itself more than once, therefore, we need to regulate the call in order to avoid the infinite calling. This is achieved with the help of base condition.

User need to learn few basic terms for the deep understanding of recursion. This will greatly enhance the users capability of writing effective recursive functions. These basic terms include:

- Base condition
- Increment/decrement of call

14.2.1 Base condition

Base condition governs the number of times the function would call to itself, by specifying the condition where the loop terminates. It helps in preventing the infinite loop. Recursive functions are terminated by specifying the **base condition**. Syntax for the base condition appears as follows:

If(x==0)

return x;

The above condition is known as the base condition and permits to return from the function upon meeting the specified condition. In the given case, it is 0.

14.2.2 Incrementing/decrementing of function call

Since, recursive function, call itself and can be terminated once the base condition is reached. Consequently, there must be a statement that should leads towards the base condition. This base condition can be reached with the help of incrementing/decrementing the statement in the function call. For instance

print(x-1)

If the value of x is 8, and this value will be decremented by one in each call. Eventually, it will fulfill the condition given in the base condition. This type of statement that leads towards the base condition by decrementing the variable value are termed as decrementing function call. Consider the other call as

print(x+1)

In the above call, base condition can be reached by incrementing the value from the calling function.

Complete program using the base condition as well as incrementing/decrementing function call has been depicted as follows:

```
Program to print the number from 10 to 1
void print (int x)// value of x passed is 1
 {
if(x! =10)  //base condition
{
cout<<"x="<<x;
print(x+1); // incrementing call
}
}
```

In the above program, print(x+1) is increment type of recursive call. We are considering that the value of x passed initially is 1. With the help of this program, we are aiming to print the number from 10 to 1. To terminate the recursive call, base condition has been specified as 10. As soon as the value of x will reach 10, calling of the function will be terminated due to the base condition involved.

14.3 ACTIVATION RECORD

In the modern programming language, functions are widely used to accomplish a specific task. Functions invoked consist of local variables that are utilized in that specific call. These local variables are destroyed at the time of exit. However, in recursion, these local variables need to exist so that they can be utilized at substantive calls. In recursion, function returns only the base condition is met. Consequently, all these calls are stored in the stack, so that they should work in last in first out (LIFO) order. The portion of the stack used for function invocation is known as stack frame or activate record.

Block of information (“frames”) associated with each function call include:

- Parameters
- Local variables
- Return address after call.
- Location to put return value, when function exits.
- Control link to the caller’s activation record.
- Saved registers.
- Intermediate results.

14.4 TYPES OF RECURSION

To meet the diverse requirement, various types of recursive function exist and are used based on their requirement. Prevailing recursion types can be categorized into the following:

- Linear recursion
- Tail recursion
- Non tail recursion
- Binary recursion
- Mutual recursion
- Indirect recursion

14.4.1 Linear recursion

In the linear recursion, recursive function calls to itself and placed at the end of the function. It continues to execute the group of statements till the base condition is not reached. It is the simplest type of recursion and it is widely used.

Consider the case to determine the factorial of a given number ‘n’. It is already known to us

that the factorial is obtained by multiplying the number from n to 1 i.e. n*(n-1)*(n-2)*(n*3).....-1. Therefore, factorial can be defined as:

$$f(n) = \{1 \quad if\ n = 0$$

$$else\ n * f(n-1)$$

From the above depicted equation, recursive function can easily be written. Here If (n==0) will act as the base condition, once base condition is met it will return to the caller. Complete program to determine the factorial is depicted below:

```
// Program to determine the factorial of a number n.
int factorial (int n)
{
if (n==0)
return 1;
else
return n*factorial (n-1);
}
```

Above function facilitates in determining the factorial of a given number that is passed as an argument. Recursively, this function will be called by decrementing the variable 'n'. Once 'n' reaches to '0', it will return 1 to the caller i.e. factorial (2). This in turn return 2 to the function called it i.e. factorial (3). This process will continue till all the calls that are saved in stack are not exhausted.

14.4.2 Tail recursion

Tail recursion is the other category of recursive function that is widely used. In the tail recursion, recursive call is placed at the end of all the statements. Statement before the recursive function is executed similar to the loops. Once the base condition is satisfied, recursive call will be terminated.

```
Program to print the number from 10 to 1
void print (int x)// value of x passed is 10
 {
if(x! =0)          //base condition
{
cout<<"x="<<x;
print(x-1); //decrementing call
}
}
```

14.4.3 Non-tail recursion

In the non-tail recursion, one or more statements exist after the recursive call. Consequently, all the statements after the recursive call will execute depending upon the calls made by the recursive function. For instance, if the recursive call has been executed four times then the statement following the recursive function will be executed four times. Example of non-tail function has been depicted below:

```
//program of non-tail recursion
void nontail1 (int x)   // value of x passed is 5
{
if(x! =0)
nontail1(x-1);
cout<<"x="<<x<<"\t";
}
Output for x=5:  0      1     2     3     4     5
```

From the above function, it is apparent that the recursive call is stored in the stack. Consequently, they are popped one by one till stack is not empty. Consider the other example of non-tail recursive function with the return type available at the end.

```
//Non-tail recursion with return statement
int nontail (int x)
{
if(x! =0)   // if x is not zero
nontail(x-1);     // continue to call non-tail function
return x;   // Eventually, return x
}
Output: 5 will be returned to the caller
```

It is extremely important to understand that we can use the return statement for returning the value. For instance, in the above program all the values from '0' to '5' will be returned to the caller. However, the one that is returned last will be available for access. Since, other values are overlapped. Similarly, the above function will return 7, if the argument is passed as 7.

14.4.4 Mutual recursion

In the mutual recursion, two different functions are calling each other to ensure that recursion takes place. Here one function func1 () calls to another function func2 (). Whereas, function func2 () will call the function func1 (). This calling will continue till the base condition of the calling function is not reached. To display the number from 'n' to 1, mutual function has been depicted in the following example.

```
// Program of mutual function to print the value from n to 1.
int fun1 (int a)
{
if (a==1)
return a;
cout<<a<<endl;
return fun2 (a-1);
}
int fun2 (int b)
{
if (b==1)
return b;
cout<<b<<endl;
return fun1 (b-1);
}
int main ()
{
clrscr ();
int x=fun1 (10);
cout<<"Resultant value of x is"<<x;
getch ();
return 0;
}
```

In the above program two functions func1 () and func2 () are acting as mutual function and facilitating in recursion. From the main function, if 10 is passed as an argument to the function func1 (). Then from the function func1 () another function func2 () is called, which in turn again calls to the function func1 (). During these calls, value of parameter is decremented by 1 so that the function can reach to base condition. On reaching the base condition call terminates, eventually control is passed to the main program. Since, n (here 10) is passed as an argument and in each call it is decremented by 1, consequently value from 10 to 1 will get printed by the recursive functions.

14.4.5 Binary recursion

In many occasion there is a need when first two terms of the series is used to generate the remainder of the series. For instance, Fibonacci series. In Fibonacci series, third term is the resultant of the previous two terms, and the remainder of the series can be generated by

using the immediate last two terms. Consequently, binary recursion can be applied in the case of Fibonacci series. In binary recursive function, two different conditions are acting as the base condition and facilitating in the termination of binary function.

Binary recursion results in the tree like structure. The hierarchical/tree like structure is generated due to the one call leads into two call. For instance, consider the case of Fibonacci series. If the n^{th} term of the Fibonacci is to be determined, in that case fibo (n) will be called. If 'n' is 4 then as per the definition, it will result in two call fib (3) and fib (2). Both the functions will not terminate here since the base condition is 0 and 1. Each of these functions (fib (3) and fib (2)) will act as an independent function and will generate two more functions from each of them. In the case of Fibonacci call, termination can take place only if the value of n is

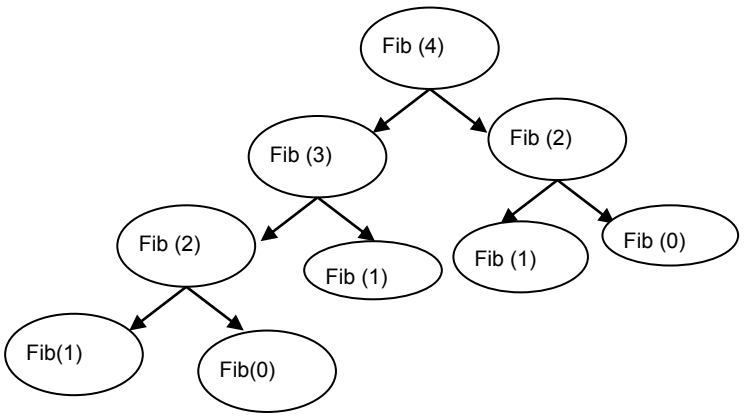


Figure 14-1: Hierarchical diagram demonstrating recursion

either '0' or '1' as specified in the base condition. Resultant tree have been illustrated in the following figure.

Complete program highlighting the definition and its calling has been depicted in the following program.

```
// Recursive function to compute the nth term of a Fibonacci series.
int fibo (int n)
 {
 if (n==0)  //base condition one, here it is n=0
 return 0;
 if (n==1) // base condition two, here n is 1
 return 1;
 return fibo (n-1) +fibo (n-2);     // Branching of fibo in two
```

```
 }
int main ()
{
clrscr ();
int n=6;
cout<<endl<<"Output of n series in Fibonacci is :"<< endl;
// Computing the Fibonacci series upto n^th integer
for (int i=0; i<n; i++)
{
int x=fibo (i);
cout<<x<<"\t";
}     // end of for clause
getch ();
return 0;
}     //end of main function
```

14.4.6 Indirect Recursion

Indirect recursion is the special case of recursion in which function call to itself but after calling some of the other functions. For instance, if rec1 () is the recursion function, it will call function A (), which may call some other function and finally the recursive function rec1 () will be called. Example of the indirect recursion has been depicted below:

rec1()→A()→B()→C()→D()→rec1()

In the above function, first recursive function rec1 () is called. It is followed by A (), and A () calls B (), followed by C (), followed by D (), Eventually, D () calls rec1 (). In such type of function calls where the cycle is formed is known as Indirect recursion. Figure 14.2 highlights the cycle that is formed by the Indirect recursion.

14.5 PROMINENT EXAMPLES OF RECURSION

Recursion can be used in variety of needs. Majority of the program can be solved with the help of recursion. It is highly preferred by the programmer/developer due to the ease of implementation and less code needed for solving the problem. However, usage of recursion should be deferred till it is not absolutely needed, since recursion involves performance overhead. Recursive function is slower relative to iterative function. Prominent recursive functions with their usage have been depicted in the following examples.

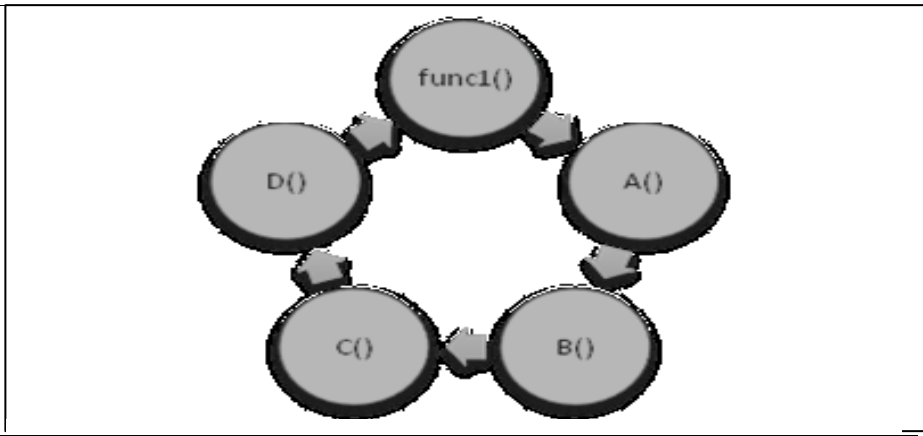


Figure 14-2: Cycle in recursion

A. Recursive function to compute the power

Consider a case to compute the power of a given number x. Value of power can be specified by the user and consider it as 'n'. Therefore, the recursive function should be capable of computing x^n.

To accomplish the above objective, a recursive function has been described below.

```
//power of x in 'n' terms or x to power n
int power(int x, int n)// raising the power of x to n
{
if(n==1)
return x;
else
return x*power(x, n-1);
}
```

To raise the power of a number 'x' to n i.e. x^n, the number x has to be multiplied 'n' times. To accomplished this objective, call using x*power(x,n-1) has been invoked. In order to ensure that the based condition is reached 'n' is decremented by 1 in each call. When the 'n' reaches to 1, 'x' is returned. Consequently, x is multiplied by n times. Eventually, the result will be returned to the caller.

B. Recursive function to multiply two numbers

Corresponding to the power function that is defined and discussed earlier, multiply function can also be defined in terms of recursion. This function will have two numbers in its argument. Since, multiplication of two numbers means sum of the number to the number of times to another number. Therefore, recursive function can be defined in these terms. Recursive function to multiply two numbers has been described in the following example.

```
// recursive function to multiply two numbers.
int multiply(int number, int times)
{
if(times==0)
return 0 ;
return number + multiply(number,times-1);
}
```

In the above function, a variable ‘times’ is used to multiply the number ‘number’ with the ‘times’. Variable ‘times’ value is decremented by one in each call. As soon as times reaches to 0, the value 0 is returned.

C. Recursive function to compute the sum of two numbers

Two numbers can be added by using the recursive function. Consider the function named as sum that accepts two arguments ‘a’ and ‘b’ of integer type. To sum these two numbers ‘a’ and ‘b’, we add ‘1’ to a ‘b’ times. Eventually, total of these two numbers can be returned to the caller.

```
//function to return sum of a and b
int sum(int a, int b)
{
if(b==0)
return a;
return 1+sum(a,b-1);
}
```

In aforementioned function, number of calls will be governed by the value of ‘b’ . For instance, if b=4, then function will be executed four times, once b is reached at 0, the value of first number i.e. a will be returned. Therefore, total will be a+1+1+1+1, eventually, ‘a+4’ will be returned to the caller.

D. Recursive function to find a character in a string

Recursive function can also be used to determine, whether a given character exist in the string or not. For instance, if it is intended to compute whether character ‘c’ exist in the string “jitendra”, recursive function can be used. This function as described in the following example returns the integer value. If the character exist then it returns 1 else returns 0.

From the calling function three arguments are passed, first is string, second is the character and finally the integer ‘i’ that is equal to 0. Here ‘i’ is incremented in each call. Correspondingly, character passed as an argument is compared with each character of the given string one by one. As soon as character matches function terminates and returns

success. If there is no much, Null will be encountered, correspondingly unsuccessful is returned to the caller.

```
//function to find out the existence of a character in a string
int pattern(char str[20], char p, int i)
{
if(str[i]=='\0')  // No match
return 0;   //return unsuccessful
if(str[i]==p)     // matches
return 1;   //return successful
pattern(str,p,i+1);     //  call  the  same  function,  increment  the value of 'i'
}
```

E. Recursive function to reverse a string

String can also be reversed with the help of recursion. In this case we can make use of '\0' character inserted at the end of the string. This will facilitate in the termination of the string once the end of the string is reached. Complete program depicting the string reversal has been described below:

```
// to compute the reverse of the string
void reverse(char str[], int i=0)
{
if(str[i]!='\0')
reverse(str,i+1);
cout<<str[i];//Reversing the string character by character
}
```

In the above program, string is received in the reverse function. Here default function has been used. That initializes the value of 'i' to 0 . In each call, value of i is incremented by 1. Once the string will reach to '\0' then the recursion will terminate. It is followed by pop operation that will take place on recursive calls that has already occurred. Eventually, character by character will be displayed from the last value of 'i'. Consequently, string will be reversed. Alternatively, we can also store it in temp variable and same can be utilized by the caller function.

F. Function from decimal to binary

Converting from the decimal to binary using manual method is fairly simple. However, implementing it on computer is notably complex. This complexity has been eliminated with the help of recursive function. As in the manual method, we compute the result till the number is not reaching to 0. Upon reaching to zero, the binary number is returned. If the number has

not reached to zero in that case remainder is computed and the numbers are adjusted based on their significance (Based on position). From the main function, value of b and factor is passed as 0 and 1. Entire function has been depicted below.

```
//function to convert the decimal into binary
int dTobinary(int dec, int b, int factor)
{
if(dec<=0)
return b;
int rem=dec%2;
b=b+rem*factor;
return dTobinary(dec/2,b,factor*10);
}
```

G. Function to reverse the digit

Recursion can also be used to compute the reverse of a given number, digit by digit. For instance, if the number is 704, then the answer will result into 407. Recursive function receives the value of a number and result from the main function. Value of number may be anything, whereas the value of variable 'result' is passed as 0. When number reaches to 0 it returns the result. Eventually, the result is returned to the caller. Entire function has been depicted below.

```
 //function to reverse the digit
int rev_digit(int number, int result)
 {
 if(number==0)
   {
   return result;
   }
   int rem=number%10;
   result=result*10+rem;
   return rev_digit(number/10,result);
   }
```

H. Digit sum

To compute the sum of the digit, complete function has been depicted below. Corresponding to non-recursive function, majority of the logic remains same. Sum is computed by adding the variable the sum with the remainder. Consider that the value is passed is 365, so initially, sum would be 5, number will be reduced to 36 by the call sum_digit(num/10,sum), statement

sum=sum + remainder will result in 5+6=11 since new remainder is 6. Finally, 3 will be send to a new call. It is not 0, consequently, rest of the statements would execute i.e. *11+3=14*,since new remainder will be 3. In the next call, num will become 0, therefore sum will be result to the caller. In this case it is 14.

```
//Digit sum of a number using recursion
 int sum_digit(int num, int sum)
  {
  if(num==0)
    {
    return sum;
    }
    sum=sum+num%10;
   return sum_digit(num/10,sum);
    }   //end of function
```

In this function, from the main we have to pass two numbers, one for which we have to compute the sum. In the given case it is 365, the second number is sum that should be 0.

I. Other recursive function

Many of the recursive function has already been discussed in this text book but are not included in order to mitigate redundancy. These recursive functions include

- Traversal of a BST.
- Creation of a tree.
- Height of a tree.
- Mirroring of a Tree.
- Binary search in a BST

Readers are requested to please refer these programs on the related chapters for the deeper understanding of the recursion.

EXERCISE

A. Descriptive Type Questions

1) What is recursion? Discuss the significance of recursion in modern programming language.
2) What is the significance of active record. What are the various components of active record.
3) What is the role of base condition? What will be the repercussion, if base condition is i) Not written ii) poorly written
4) How mutual function is different from the indirect function?
5) What are the usage of binary function? Discuss the base condition execution in the binary function.
6) Discuss the performance overhead involved in case of recursion.
7) What is stack and heap in recursive function. What is their significance in recursive function?
8) Explain the functioning of the tail recursion.
9) Write down the program to reverse the link list using recursion.
10) Write down the program to compute the GCD using recursion.
11) Compute the permutation of a number using recursion?
12) Compute the greater of two number using recursion (Don't use the if statement).
13) Write down the recursive function for the following equation.

    0 if n=0

    f(n)= f(2n) if n is odd

    1+f(n-1) if n is even

14) Write down the recursive function for the following equation.

    if n=0

    f(n)= n+ f(2n) if n is odd

    f(n-1) +f(n-2) if n is even

B. Multiple Choice Answers

1. In recursion, function can call to another function. (True/False).
2. There may be recursive function that may not return the value to the caller (True/False).
3. Recursive function have better performance relative to the iterative function.
4. Recursive functions are always concise relative to iterative function (True/False).

5. In recursion, incremental/decremented statements leads to

a) Infinite looping
c) base condition
b) Both a and c
d) None of these

6. Predict the output of the following recursive function. Consider that the value of x is 2.

```
rec(int x)
{
if(x<1)
return 1;
if(x%2==0)
rec(x*x+1);
return rec(x/2);
}
```

a) 24
c) infinite call
b) 1
d) None of these

7. Predict the output for the question number 13, if the n=1, n=2, n=8, n=9.

8. Predict the output for the question number 14, if the n=1, n=2, n=8, n=9.

APPENDIX

Template

Objective

- Defining template
- Scenario of templates
- Usage of template
- Creation of template
- Discussing the prominent examples.

INTRODUCTION

Templates are used for creating the generic class and function. It is recently added features in C++. Templates offer the following advantages:

- Avoids re-writing of the code.
- Eliminates the testing of re-written code.
- Prevents any error that may creep due to re-writing of the code.

Aforementioned reasons are among the various reasons due to which templates have gained the wide popularity. We can understand the limitations that exist due to non-usage of template with the help of following sub-sections.

NEED OF A TEMPLATE

Consider a case in which user is intending to initialize the integer in a function call. Similarly, same array size is used to inititalize the float members. In each case argument is passed in a function depending upon the type of data. Consequently, to accomplish this objective, (s) he has to define the following program.

```
#include<iostream.h>
#include<conio.h>
const int size=4;
class one
{
int a [size];
float b [size];
public:
one (int []);
one (float []);
```

```
void sum ();
void sum (float);
};
void one::sum (float s1)
 {
 float s=0.0;
 cout<<" In the float sum"<<endl;
 for (int i=0; i<size; i++)
 s=s+b[i];
 cout<<" Float sum"<<s<<endl;
}
one::one (int aa [])
{
 for (int i=0; i<size; i++)
 a[i] =aa[i];
 }
 one::one (float bb [])
 {
 for (int i=0; i<size; i++)
 b[i] =bb[i];
 }
 void one::sum ()
 {
 int s=0;
 for (int i=0; i<size; i++)
 s=s+a[i];
 cout<<" Integer sum"<<s<<endl;
 }
 int main ()
 {
 clrscr ();
 int arr [] = {2,3,4,5};
 float farr[]={4.2,6.3,4.7,9.4};
 float s=0.0;
 one a(arr);
```

```
 one b(farr);
a.sum();
 b.sum(s);
 getch();
 return 0;
 }
```

It is apparent from the above code that redundancies exist in function definition due to different types of data. Such redundancy do not only requires re-writing of the code but also increases the effort of re-writing the code. In addition, error may creep in during the re-writing of the code.

USE OF A TEMPLATE

Template enable us to write the code once that is applicable for all the data types provided the number of parameters are same. Template has been discussed in detail in the upcoming section. Broadly, templates can be categorized into

- Class template
- Function template

Class templates are used in the cases where we need to send the data type for the entire class as with the above example. Similarly, function templates can be used as a function that accept the argument of any type and perform the function needed. Eventually, function template terminates with the return type and capable of returning any type of value similar to other functions.

CLASS TEMPLATE

Class template is used in cases where the data type is to be created that need to be accessed by many members of the class. To create the template we have to use the syntax

Class template with single argument type

template<class type>

Here the type may be the build-in type or the user defined type. In our case, we have represented it with the small ‘t’, consequently, the statement will be

template<class t>

we can define more than one type in the arguments, if needed. Correspondingly, class can accept more than one data type. Syntax for more than one data types is:

template< class t1, class t2,……., class tn>

In the above template, upto ‘n’ types of arguments have been passed, however, we will be restricted upto two argument types.

In the following example, we have depicted how the class template can be used to define a type that will be applicable and accessible through the class:

```
Program: Use of class template
const int size=4;
template<class t>
class one
{
t a[size];  // declaring the t type
public:
one(t[]);
void sum();
};
template<class t>
one<t>::one(t aa[])
{
 for(int i=0;i<size;i++)
 a[i]=aa[i];
 }
template<class t>
 void one<t>::sum()     // defining the function outside the class
 {
 t s=0;
 for(int i=0;i<size;i++)
 s=s+a[i];         // adding the number one by one
 cout<<" sum of all numbers: "<<s<<endl;
 }
 int main()
 {
 clrscr( );
 int arr[]={2,3,4,5};
 float farr[]={2.5,3.2,4.6,5.9};
 one<int> a(arr); // creating the template as integer type
 one<float> b(farr);    // creating the template as float type
 a.sum();
 cout<<" Sum of float array"<<endl;
```

```
b.sum( );
getch( );
return 0;
}
```

In the above example, we have used the template class and applied the same definition for the integer type and float type. From the above example it is apparent that same definition can be applied on more than one type without any need for modification of the code. Thereby, it results in great cost saving in terms of human effort required in code re-writing and testing of the code.

Class template with two arguments

In many cases, we need to create the template for more than two data types. This can be accomplished in the class data type with the help passing two arguments. To achieve the above cited requirement, we can use the following syntax.

template<class t1, class t2>

Consider the following example in which programmer is aiming to accept two different types of arguments. In this case, we can use the template that can accept two arguments instead of one. Complete program for the above objective can be written as:

```
// Program to enumerate the use of two argument type in the class
template<class type1, class type2>
class one
{
type1 first;
type2 second;
public:
one (type1 o, type2 t);
void display ();
};
template<class type1, class type2>
one<type1, type2>: one (type1 o, type2 t)
{
first=o;
second=t;
 }
template<class type1, class type2>
 void one<type1, type2>: display ()
```

```
 {
 for (int i=0; i<second; i++)
 cout<<first<<"\t";
  }
 int main ()
 {
 clrscr ();
 one<char*, int> o1 ("jitendra", 2);
 one<int, int> o2 (30, 4);
 o1.display ();
 cout<<endl;
 o2.display ();
 getch ();
 return 0;
}
Output
Jitendra          jitendra
30    30    30    30
}
```

In the above example, we have used the class template that has accepted the two arguments. In first object i.e. o1, we have passed two arguments of different type, the first argument was string and the second was integer type. Argument passed first i.e. string is repeated number of times the value of second argument.

In the second object, again we have passed two arguments, however, this time types are same. It means that the argument of same types can also we passed even though we have defined two arguments. Precisely, speaking, number of template arguments governs the number of arguments that can be passed instead of number of data types. In our case, the first argument is the number to be printed and the second argument is the number of times the first argument is to be printed.

Function template

Function template is used to create the generic function. Consequently, same function can be used for wide variety of data types. To declare that the function is template type, we define the function with the keyword template similar to that of class type. Syntax for declaring the generic function can be written as follows:

template<class t>

return-type function_name (class t);

Function template using one argument

Generic function of one argument will enable the user to support the single argument for various data type. Need of a function template can be learned with the help of following example.

Consider that user intend to pass character array or integer array along with the element to be searched from within the supplied array. If character is to be searched from the character array and integer number to be searched from integer array in that case two different functions have to be created as described in the following program.

```
// Program to search the element from the integer/character array
const int size=4;
//function to search the integer type element from integer array
void search(int a[size], int element)
 {
 int i=0;
 int flag=0;
 while(i<size)
    {
    if(a[i]==element)
    {
    flag=1;
    break;
    }
    i++;
 }  //end of while
 if(flag)
 cout<<" item found";
 else
 cout<<"item not in the list";
 }//end of function
//function to search the char type element from character type array
void search(char a[size], char element)
 {
 int i=0;
 int flag=0;
 while(i<size)
```

```
    {
    if(a[i]==element)
    {
    flag=1;
    break;
    }
    i++;
 }  //end of while
 if(flag)
 cout<<" item found";
 else
 cout<<"item not in the list";
 }//end of function
 int main()
 {
 clrscr();
 int arr[]={2,3,4,5};
 float farr[]={2.5,3.2,4.6,5.9};
 char a[]={'a','c','e','g'};
 cout<<"integer result"<<endl;
 search(arr,14);
 cout<<"character result"<<endl;
 search(a,'c');
 getch();
 return 0;
 }
```

To avoid the re-writing of the same code, function template can be used. In the upcoming program, we have **taken two arguments in a template, however, both of them are same type**.

```
// Program to search the element using generic function
const int size=4;
template<class t>
 void search(t a[size], t element)
 {
```

```
 int i=0;
 int flag=0;
 while(i<size)
    {
    if(a[i]==element)
    {
    flag=1;
    break;
    }
    i++;
 }  //end of while
 if(flag)
 cout<<" item found";
 else
 cout<<"item not in the list";
 }//end of function
 int main()
 {
 clrscr();
 int arr[]={2,3,4,5};
 float farr[]={2.5,3.2,4.6,5.9};
 char a[]={'a','c','e','g'};
 cout<<"integer result"<<endl;
 search(arr,14);
 cout<<"character result"<<endl;
 search(a,'c');
cout<<"Float result"<<endl;
search(farr,3.2);
getch();
 return 0;
 }
```

In the above example, we have declared the function as generic type. Consequently, in the call of integer type i.e. search(arr,14), it will get converted into integer type. Whereas at the time of character array i.e. search(a, 'c') it will be converted into character type. Therefore, to call different type of data same call can be used due to creation of template.

Function template using two arguments

Similar to one argument type template function, we can create two argument type template functions. Thereby, it will enable us to pass two arguments of different data types.

Consider the case that we are aiming to print the first argument based on the value of second argument i.e. the number of times the value of second variable. First argument may be any data type. Same has been illustrated with the help of following example:

```
// Function template type, accepting two different types of arguments
template<class type1, class type2>
 void display (type1 one, type2 two)
 {
 int i=0;
 while (i<two) {
 cout<<one<<"\t";
 i++;
 } //end of while
 }//end of function
 int main ()
 {
 clrscr ();
 display ("jitendra", 5);
 display (58, 3);
 getch ();
 return 0;
}
Output
jitendra    jitendra    jitendra    jitendra    jitendra
58    58    58
 }
```

In the above program, first we pass the string and the number of times passed string is to be printed. Consequently, at the time of called function display (“jitendra”, 5), it will display the name ‘jitendra 5’ times. In the second call we pass the integer type by display (58, 3), therefore, 58 is printed 3 times.

Salient Features

Data structure is an extremely important subject in computer science. A number of books on this subject are available and are enriching the students based on their own methodology. However, this book is written in a manner so that readers can learn data structure with the help of this book itself. Other features of this book are:

- Use of simple language for coding.
- Major focus on link list, tree, stack, and queue.
- Special emphasis on recursion.
- In majority of the cases, both recursive and non-recursive functions have been used for single operation.
- Object oriented approach using C++ to code the program.
- Objective of each chapter discussed in the beginning.
- Code is substantiated with ample of figures and description.
- All the written codes are tested and executed.
- Equally beneficial for the novice as well as experts.
- Each chapter is followed by case study.
- Inclusion of exercise at the end of the chapter to enable readers to evaluate themselves.

References and Bibliograpy

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. Data Structures and Algorithms. Addison-Wesley, 1983.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

Algorithms, and Applications. Prentice Hall, 1993.

Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. Communications of the ACM, 31(9):1116–1127, 1988.

Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in linear time? Journal of Computer and System Sciences, 57:74–93, 1998.

Arne Andersson. Balanced search trees made simple. In Proceedings of the Third Workshop on Algorithms and Data Structures, volume 709 of Lecture Notes in Computer Science, pages 60–71. Springer, 1993.

Arne Andersson. Faster deterministic sorting and searching in linear space. In Proceedings of the 37th Annual Symposium on Foundations of Computer Science, pages 135–141, 1996.

Bollobas, B. (2012). *Graph theory: an introductory course* (Vol. 63). Springer Science & Business Media.

Bondy, J. A., & Murty, U. S. R. (1976). *Graph theory with applications* (Vol. 290). London: Macmillan.

Coplien, J. (1997, November). Advanced C++ programming styles and idioms. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25, Proceedings* (pp. 352-352). IEEE.

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press…

Drozdek, A. (2004). *Data structures and algorithms in Java*. Cengage Learning.

Drozdek, A. (2012). *Data Structures and algorithms in C++*. Cengage Learning.

Eppstein, D., Goodrich, M. T., & Sun, J. Z. (2005, June). The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proceedings of the twenty-first annual symposium on computational geometry* (pp. 296-305). ACM.

Frakes, W. B., & Baeza-Yates, R. (1992). Information retrieval: data structures and algorithms.

Gusfield, D. (1997). *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press.

Horowitz, E., Sahni, S., & Anderson-Freed, S. (1983). *Fundamentals of data structures* (No. 04; QA76. D35, H6.). London: Pitman.

Horowitz, E., Sahni, S., & Rajasckaran, S. (1996). *Computer algorithms: C++*. WH Freeman & Co..

Kanetkar, Y. (2008). Data structure through C++.

Kanetkar, Y. P. (2016). *Let us C*. BPB publications.

Lafore, R. (1997). *Object-oriented programming in C++*. Pearson Education.

Lafore, R. (1999). *Sams Teach Yourself Data Structures and Algorithms in 24 Hours with Cdrom*. Sams.

Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. Annals of Mathematics,160(2):781–793, 2004.

Mehta, D. P., & Sahni, S. (Eds.). (2004). *Handbook of data structures and applications*. CRC Press.

Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. Journal of the ACM, 37:213–223, 1990.

Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network Flows: Theory,

Selim G. Akl. The Design and Analysis of Parallel Algorithms. Prentice Hall, 1989.

Stroustrup, B. (1995). *The C++ programming language*. Pearson Education India.

Structures, D. (1998). Algorithms in C++. *Adam Drozdek*, *4*.

Vandevoorde, D., & Josuttis, N. M. (2002). *C++ Templates*. Addison-Wesley Longman Publishing Co., Inc..

West, D. B. (2001). *Introduction to graph theory* (Vol. 2). Upper Saddle River: Prentice hall.

Index

D

E

F

G

H

I

K

L

M

N

O

T

U

W