DATA STRUCTURES & ALGORITHMS IN Ruby

Stack

push   pop

6   ← top
5
4
3
2
1

HEMANT JAIN

1ST EDITION

Data Structures & Algorithms in Ruby
First Edition

By Hemant Jain

Problems Solving in Data Structures & Algorithms in Ruby

Hemant Jain

ACKNOWLEDGEMENT

TABLE OF CONTENTS

CHAPTER 0: HOW TO USE THIS BOOK

What this book is about

This book introduces you to the world of data structures and algorithms. Data structure defines the way how data is arranged in computer memory for fast and efficient access while algorithm is a set of instruction to solve problems by manipulating these data structures.

Designing an efficient algorithm is a very important skill that all computer companies e.g. Microsoft, Google, Facebook etc. pursue. Most of the interviews for these companies are focused on knowledge of data structure and algorithm. They look for how candidates use these to solve complex problems efficiently, which is also very important in everyday coding. Apart from knowing, a programming language you also need to have good command on these key Computer fundamentals to not only qualify the interview but also excel in the top high paying jobs.

This book assumes that you are a Ruby language developer. You are not an expert in Ruby language, but you are well familiar with concepts of class, references, functions, list, tuple, dictionary and recursion. At the start of this book, we will be revising Ruby language fundamentals that will be used throughout this book. We will be looking into some of the problems in Lists and recursion too.

Then in the coming chapter we will be looking into Complexity Analysis. Followed by the various data structures and their algorithms. We will be looking into a Linked-List, Stack, Queue, Trees, Heap, Hash-Table and Graphs. We will also be looking into Sorting, Searching techniques.

We will be looking into algorithm analysis of various algorithm techniques. Such as, Brute-Force algorithms, Greedy algorithms, Divide and Conquer algorithms, Dynamic Programming, Reduction and Backtracking.

Preparation Plans

Generally you have limited time before your next interview, it is important to have a solid preparation plan. The preparation plan depends upon the time and which companies you are planning to target. Below are the three-preparation plan for 1 Month, 3 Month and 5 Month durations.

1 Month Preparation Plans

Below is a list of topics and approximate time users need to finish these topics. These are the most important chapters that must be prepared before appearing for an interview.

This plan should be used when you have a limited preparation time for an interview. These chapters cover 90% of data structures and algorithm based interview questions. In this plan, since we are reading about the various ADT and Ruby collections (or built in data structures.) in chapter 4 so we can use these datatype easily without knowing the internal details how they are implemented.

Time	Chapters	Explanation
Week 1	Chapter 1: Algorithms Analysis Chapter 2: Approach To Solve Algorithm Design Problems Chapter 3: Abstract Data Type & Ruby Collections	You will get a basic understanding of how to find complexity of a solution. You will come to know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 2	Chapter 4: Searching Chapter 5: Sorting Chapter 13: String Algorithms	Searching, Sorting and String algorithm consists of a major portion of the interviews.
Week 3	Chapter 6: Linked List Chapter 7: Stack Chapter 8: Queue	Linked list, Stack and Queue are some of the favourites in an interview.
Week 4	Chapter 9: Tree	In this portion you will read about Trees

3 Month Preparation Plan

This plan should be used when you have some time to prepare for an interview. This preparation plan includes nearly everything in this book except various algorithm techniques. Algorithm problems that are based on "dynamic programming", "divide & conquer" etc. Which are asked by vary specific companies like Google, Facebook, etc. Therefore, until you are planning to face interview with them you can withhold these topics for some time and should focus on the rest of the topics.

Time	Chapters	Explanation
Week 1	Chapter 1: Algorithms Analysis Chapter 2: Approach To Solve Algorithm Design Problems Chapter 3: Abstract Data Type & Ruby Collections	You will get a basic understanding of how to find complexity of a solution. You will know how to handle new problems. You will read about a variety of datatypes and their uses.
Week 2 & Week 3	Chapter 4: Searching Chapter 5: Sorting Chapter 13: String Algorithms	Searching, sorting and string algorithm consist of a major portion of the interviews.
Week 4 & Week 5	Chapter 6: Linked List Chapter 7: Stack Chapter 8: Queue	Linked list, Stack and Queue are some of the favourites in an interview.
Week 6 & Week 7	Chapter 9: Tree Chapter 10: Heap	In this portion you will read about trees and heap data structures.
Week 8 & Week 9	Chapter 11: Hash-Table Chapter 12: Graphs	Hash-Table is used throughout this book in various places, but now it is time to understand how Hash-Table is actually

		implemented. Graphs are used to propose a solution many real life problems.
Week 10, Week 11 & Week 12	Revision of the chapters listed above.	At this time, you need to revise all the chapters that we have gone through in this book. Whatever remains needs to be completed and the exercise that remain uncovered need to be solved in this time

5 Month Preparation Plan

This preparation plan is meticulously devised on top of 3-month plan. In this plan, students should look for algorithm design chapters. In addition, in the rest of the time they need to practise more and more from www.topcoder.com and other resources. If you are targeting for google, Facebook, etc., Then it is highly recommended to join topcoder and make practice as much as possible.

Time	Chapters	Explanation
Week 1 Week 2	Chapter 1: Algorithms Analysis Chapter 2: Approach To Solve Algorithm Design Problems Chapter 3: Abstract Data Type & Ruby Collections	You will get a basic understanding of how to find complexity of a solution. You will know how to handle unseen problems. You will read about a variety of datatypes and their uses.
Week 3 Week 4 Week 5	Chapter 4: Searching Chapter 5: Sorting Chapter 13: String Algorithms	Searching, sorting and string algorithm consists of a major portion of the interviews.
Week 6 Week 7 Week 8	Chapter 6: Linked List Chapter 7: Stack Chapter 8: Queue	Linked list, Stack and Queue are some of the favourites in an interview.
Week 9 Week 10	Chapter 9: Tree Chapter 10: Heap	This portion you will read about trees and priority queue.
Week 11 Week 12	Chapter 11: Hash-Table Chapter 12: Graphs	Hash-Table is used throughout in this book in various places, but now it is time to understand how Hash-Table are actually implemented. Graphs are used to propose a solution in many real life problems.
Week 13 Week 14 Week 15 Week 16	Chapter 14: Algorithm Design Techniques Chapter 15: Brute Force Chapter 16: Greedy Algorithm Chapter 17: Divide-And-Conquer, Decrease-And-Conquer Chapter 18: Dynamic Programming Chapter 19: Backtracking And Branch-And-Bound	These chapters contain various algorithms types and their usage. Once the user is familiar with most of these algorithms. Then the next step is to start solving topcoder problems from topcoder.

	Chapter 20: Complexity Theory And Np Completeness	
Week 17 Week 18 Week 19 Week 20	Revision of the chapters listed above.	At this time, you need to revise all the chapters that we have gone through in this book. Whatever remains needs to be completed and the exercise that may remain, needs to be solved in this period.

Summary

These are few preparation plans that can be followed to complete this book while preparing for the interview. It is highly recommended that you should read the problem statement, try to solve the problems by yourself and then only you should look into the solution to find the approach of this book. Practising more and more problems will increase your thinking power and you will be able to handle unseen problems in an interview. We recommend you to make practicing all the problems given in this book, then solve more and more problems from online resources like www.topcoder.com, www.careercup.com, www.geekforgeek.com etc.

CHAPTER 1: ALGORITHMS ANALYSIS

Introduction

We learn by experience. By looking into various problem solving algorithms or problem solving techniques we begin to develop a pattern that will help us in solving similar problems appear before us.

Algorithm

An algorithm is a set of steps to accomplish a task. Or an algorithm in a computer program in which a set of steps applied over a set of input to produce a set of output.

Knowledge of algorithm helps us to get desired result faster by applying the appropriate algorithm.

The most important properties of an algorithm are:
1. **Correctness**: The algorithm should be correct. It should be able to process all the given inputs and provide correct output.

2. **Efficiency**: The algorithm should be efficient in solving problems. Efficiency is measured in two parameters. First is Time-Complexity, how quick result is provided by an algorithm. And the second is Space-Complexity, how much RAM that an algorithm is going to consume to give desired result.

Time-Complexity is represented by function $T(n)$ - time versus the input size n.
Space-Complexity is represented by function $S(n)$ - memory used versus the input size n.

Asymptotic analysis

Asymptotic analysis is used to compare the efficiency of algorithm independently of any particular data set or programming language.

We are generally interested in the order of growth of some algorithm and not interested in the exact time required for running an algorithm. This time is also called Asymptotic-running time.

Big-O Notation

Definition: "f(n) is big-O of g(n)" or $f(n) = O(g(n))$, if there are two +ve constants c and n0 such that
$f(n) \leq c\ g(n)$ for all $n \geq n0$,

In other words, c g(n) is an upper bound for f(n) for all n ≥ n0
The function f(n) growth is slower than c g(n)

We can simply say that after a sufficient large value of input N the (c.g(n)) will always be greater than f(n).
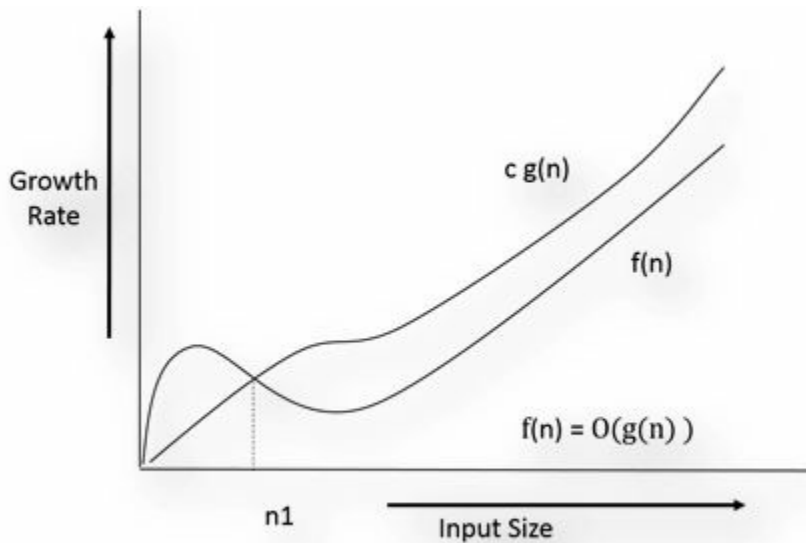
Example: $n^2 + n = O(n^2)$

Omega-Ω Notation

Definition: "f(n) is omega of g(n)." or $f(n) = \Omega(g(n))$ if there are two +ve constants c and n0 such that
c g(n) ≤ f(n) for all n ≥ n0

In other words, c g(n) is lower bound for f(n)
Function f(n) growth is faster than c g(n)
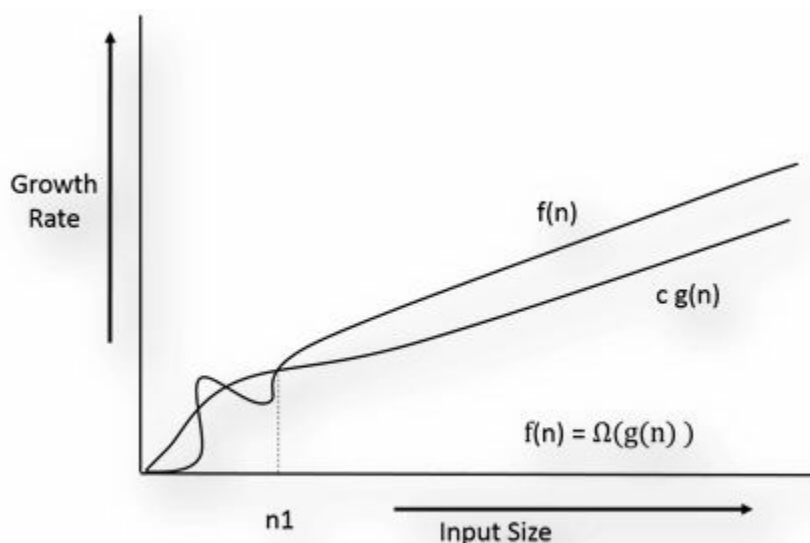
Find relationship of $f(n) = n^c$ and $g(n) = c^n$
$f(n) = \Omega(g(n))$

Theta-Θ Notation

Definition: "f(n) is theta of g(n)." or $f(n) = \Theta(g(n))$ if there are three +ve constants $c1$, $c2$ and $n0$ such that $c1\ g(n) \leq f(n) \leq c2\ g(n)$ for all $n \geq n0$

Function g(n) is an asymptotically tight bound on f(n). Function f(n) grows at the same rate as g(n).
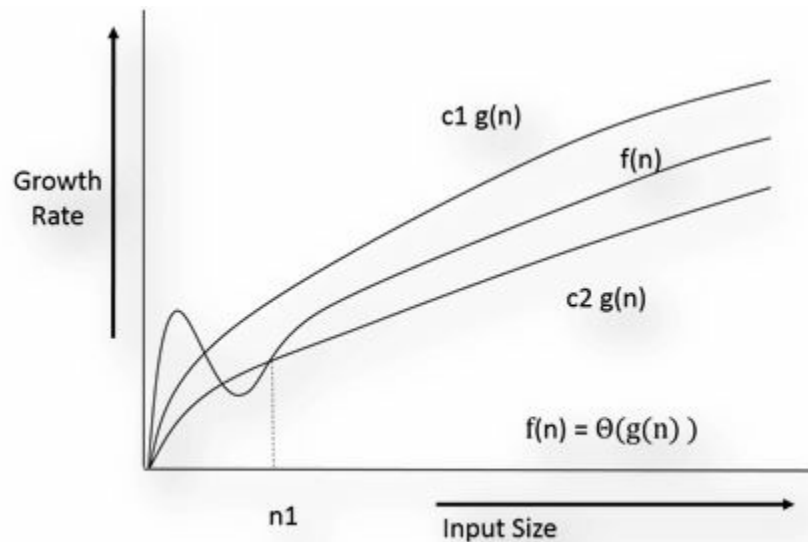
Example: $n^3 + n^2 + n = \Theta(n^3)$
Example: $n^2 + n = \Theta(n^2)$

Find relationship of $\quad f(n) = 2n^2 + n \quad$ and $g(n) = n^2$
$f(n) = O(g(n))$
$f(n) = \Theta(g(n))$
$f(n) = \Omega(g(n))$

Note:- Asymptotic Analysis is not perfect, but that is the best way available for analysing algorithms.

For example, say there are two sorting algorithms first take $f(n) = 10000*n*log(n)$ and second $f(n) = n^2$ time. The asymptotic analysis says that the first algorithm is better (as it ignores constants) but, actually for a small set of data when n is smaller than 10000, the second algorithm will perform better. To consider this drawback of asymptotic analysis case analysis of the algorithm is introduced.

Complexity analysis of algorithms

1. **Worst Case complexity:** It is the complexity of solving the problem for the worst input of size n. It provides the upper bound for the algorithm. This is the most common analysis used.
2. **Average Case complexity**: It is the complexity of solving the problem on an average. We calculate the time for all the possible inputs and then take an average of it.
3. **Best Case complexity**: It is the complexity of solving the problem for the best input of size n.

Time Complexity Order

A list of commonly occurring algorithm Time Complexity in increasing order:

Name	Notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
N-LogN	$O(n \log n)$
Quadratic	$O(n^2)$
Polynomial	$O(n^c)$ c is a constant & c>1
Exponential	$O(c^m)$ c is a constant & c>1
Factorial or N-power-N	$O(n!)$ or $O(n^n)$

Constant Time: O(1)

An algorithm is said to run in constant time if the output is produced in constant time regardless of the input size.

Examples:
1. Accessing n^{th} element of an array
2. Push and pop of a stack.
3. Enqueue and remove of a queue.
4. Accessing an element of Hash-Table.

Linear Time: O(n)

An algorithm is said to run in linear time if the execution time of the algorithm is directly proportional to the input size.

Examples:
1. Array operations like search element, find min, find max etc.
2. Linked list operations like traversal, find min, find max etc.

Note: when we need to see/ traverse all the nodes of a data-structure for some task then complexity is no less than **O(n)**

Logarithmic Time: O(logn)

An algorithm is said to run in logarithmic time if the execution time of the algorithm is proportional to the logarithm of the input size. Each step of an algorithm, a significant portion of the input is pruned out without traversing it.

Example: Binary search, we will read about these algorithms in this book.

N-LogN Time: O(nlog(n))

An algorithm is said to run in logarithmic time if the execution time of an algorithm is proportional to the product of input size and logarithm of the input size.

Example:
1. Merge-Sort
2. Quick-Sort (Average case)
3. Heap-Sort

Note: Quicksort is a special kind of algorithm to sort an array of numbers. Its worst-case complexity is $O(n^2)$ and average case complexity is O(n log n).

Quadratic Time: $O(n^2)$

An algorithm is said to run in quadratic time if the execution time of an algorithm is proportional to the square of the input size.

Examples:
1. Bubble-Sort
2. Selection-Sort
3. Insertion-Sort

Deriving the Runtime Function of an Algorithm

Constants

Each statement takes a constant time to run. Time Complexity is O(1)

Loops

The running time of a loop is a product of running time of the statement inside a loop and number of iterations in the loop. Time Complexity is O(n)

Nested Loop

The running time of a nested loop is a product of running time of the statements inside loop multiplied by a product of the size of all the loops. Time Complexity is **$O(n^c)$**. Where c is a number of loops. For two loops, it will be **$O(n^2)$**

Consecutive Statements

Just add the running times of all the consecutive statements

If-Else Statement

Consider the running time of the larger of if block or else block. Moreover, ignore the other one.

If each iteration the input size is decreased by a constant factors. Time Complexity = O(log n).

Time Complexity Examples

Example 1.1

```
def fun1(n)
    m = 0
    i = 0
    while i < n
        m += 1
        i += 1
    end
    return m
end
```

Time Complexity: **O(n)**

Example 1.2

```
def fun2(n)
    i = 0
    m = 0
    while i < n
        j = 0
        while j < n
            m += 1
            j += 1
        end
        i += 1
    end
    return m
end
```

Time Complexity: $O(n^2)$

Example 1.3

```
def fun3(n)
    m = 0
    i = 0
    while i < n
        j = 0
        while j < i
            m += 1
            j += 1
        end
```

```
      i += 1
   end
   return m
end
```

Time Complexity: $O(N+(N-1)+(N-2)+...) == O(N(N+1)/2) == O(n^2)$

Example 1.4
```
def fun4(n)
   m = 0
   i = 1
   while i < n
      m += 1
      i = i * 2
   end
   return m
end
```

Each time problem space is divided into half. Time Complexity: **O(log(n))**

Example 1.5
```
def fun5(n)
   m = 0
   i = n
   while i > 0
      m += 1
      i = i / 2
   end
   return m
end
```

Same as above each time problem space is divided into half. Time Complexity: **O(log(n))**

Example 1.6
```
def fun6(n)
   m = 0
   i = 0
   while i < n
      j = 0
      while j < n
         k = 0
         while k < n
            m += 1
            k += 1
         end
         j += 1
      end
      i += 1
```

```
      end
   return m
end
```

Outer loop will run for n number of iterations. In each iteration of the outer loop, inner loop will run for n iterations of its own. Final complexity will be n*n*n.

Time Complexity: **O(n³)**

Example 1.7
```
def fun7(n)
   m = 0
   i = 0
   while i < n
      j = 0
      while j < n
         m += 1
         j += 1
      end
      i += 1
   end
   i = 0
   while i < n
      k = 0
      while k < n
         m += 1
         k += 1
      end
      i += 1
   end
   return m
end
```

These two groups of for loop are in consecutive so their complexity will add up to form the final complexity of the program. Time Complexity: **O(n²) + O(n²) = O(n²)**

Example 1.8
```
def fun8(n)
   m = 0
   i = 0
   while i < n
      j = 0
      while j < Math.sqrt(n)
         m += 1
         j += 1
      end
      i += 1
   end
   return m
```

```
end
```

Time Complexity: $O(n * \sqrt{n}) = O(n^{3/2})$

Example 1.9
```
def fun9(n)
    m = 0
    i = n
    while i > 0
        j = 0
        while j < i
            m += 1
            j += 1
        end
        i /= 2
    end
    return m
end
```

Each time problem space is divided into half.   Time Complexity: $O(\log(n))$

Example 1.10
```
def fun10(n)
    m = 0
    i = 0
    while i < n
        j = i
        while j > 0
            m += 1
            j -= 1
        end
        i += 1
    end
    return m
end
```

O(N+(N-1)+(N-2)+...) = O(N(N+1)/2)   // arithmetic progression.
Time Complexity: $O(n^2)$

Example 1.11
```
def fun11(n)
    m = 0
    i = 0
    while i < n
        j = i
        while j < n
            k = j + 1
            while k < n
```

```
              m += 1
              k += 1
          end
          j += 1
      end
      i += 1
  end
  return m
end
```

Time Complexity: **O(n³)**

Example 1.12
```
def fun12(n)
  m = 0
  i = 0
  j = 0
  while i < n
      while j < n
          m += 1
          j += 1
      end
      i += 1
  end
  return m
end
```

Think carefully once again before finding a solution, j value is not reset at each iteration.
Time Complexity: **O(n)**

Example 1.13
```
def fun13(n)
  m = 0
  i = 1
  while i <= n
      j = 0
      while j <= i
          m += 1
          j += 1
      end
      i *= 2
  end
  return m
end
```

The inner loop will run for 1, 2, 4, 8,… n times in successive iteration of the outer loop.
Time Complexity: $T(n) = O(1+ 2+ 4+ ….+n/2+n) = $ **O(n)**

Example 1.14

```
# Testing Code
print "N = 100, Number of instructions :: " , fun1(100), "\n"
print "N = 100, Number of instructions :: " , fun2(100), "\n"
print "N = 100, Number of instructions :: " , fun3(100), "\n"
print "N = 100, Number of instructions :: " , fun4(100), "\n"
print "N = 100, Number of instructions :: " , fun5(100), "\n"
print "N = 100, Number of instructions :: " , fun6(100), "\n"
print "N = 100, Number of instructions :: " , fun7(100), "\n"
print "N = 100, Number of instructions :: " , fun8(100), "\n"
print "N = 100, Number of instructions :: " , fun9(100), "\n"
print "N = 100, Number of instructions :: " , fun10(100), "\n"
print "N = 100, Number of instructions :: " , fun11(100), "\n"
print "N = 100, Number of instructions :: " , fun12(100), "\n"
print "N = 100, Number of instructions :: " , fun13(100), "\n"
```

Output:

```
N = 100, Number of instructions :: 100
N = 100, Number of instructions :: 10000
N = 100, Number of instructions :: 4950
N = 100, Number of instructions :: 7
N = 100, Number of instructions :: 7
N = 100, Number of instructions :: 1000000
N = 100, Number of instructions :: 20000
N = 100, Number of instructions :: 1000
N = 100, Number of instructions :: 197
N = 100, Number of instructions :: 4950
N = 100, Number of instructions :: 166650
N = 100, Number of instructions :: 100
N = 100, Number of instructions :: 134
```

Master Theorem

The master theorem solves recurrence relations of the form: $T(n) = a\ T(n/b) + f(n)$

Where $a \geq 1$ and $b > 1$.
"n" is the size of the problem. "a" is a number of sub problem in the recursion. "n/b" is the size of each sub-problem. "f(n)" is the cost of the division of the problem into sub problem and merger of results of sub-problems to get the final result.

It is possible to determine an asymptotic tight bound in these three cases:

- Case 1: when $f(n) = O(n^{\log_b a - \epsilon})$ and constant $\epsilon > 1$, then the final Time Complexity will be: $T(n) = \Theta(n^{\log_b a})$

- Case 2: when $f(n) = \Theta(n^{\log_b a} \log^k a)$ and constant $k \geq 0$, then the final Time Complexity will be $T(n) = \Theta(n^{\log_b a} \log^{k+1} a)$

- Case 3: when $f(n) = \Omega(n^{\log_b a + \epsilon})$ and constant $\epsilon > 1$, Then the final Time Complexity will be:

T(n) = Θ(f(n))

T(n)= a T(n/b) + f(n)   Where a ≥ 1, b > 1

f(n) =O ($n^{\log_b a - \epsilon}$)

f(n) = Ω($n^{\log_b a + \epsilon}$)

f(n) = Θ($n^{\log_b a}$ $\log^k n$)

T(n) = Θ($n^{\log_b a}$)

T(n) =Θ($f(n)$)

Case 1

T(n) = Θ($n^{\log_b a}$ $\log^{k+1} n$)

Case 3
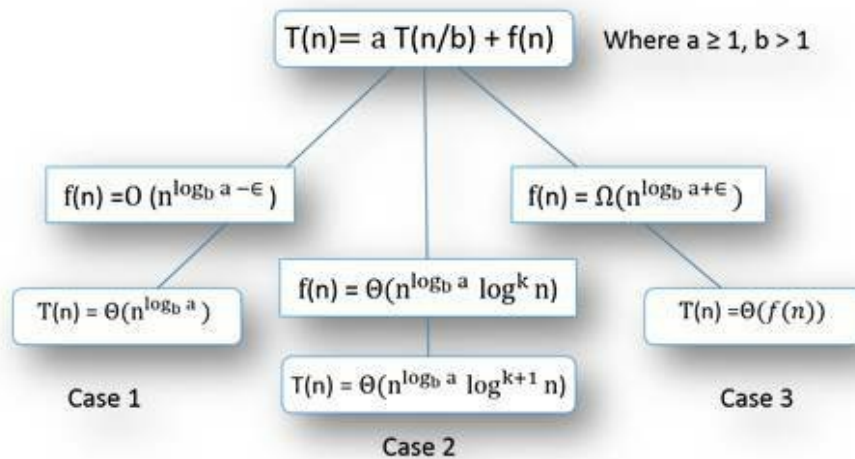
Case 2

Example 1.15: Take an example of Merge-Sort, T(n) = 2 T(n/2) + n
Sol:- $\log_b a = \log_2 2 = 1$

$f(n) = n = \Theta(n^{\log_2 2} \log^0 n)$

Case 2 applies and $f(n) = \Theta(n^{\log_2 2} \log^{0+1} n)$
T(n) = Θ (n log(n))

Example 1.16: Binary Search T(n) = T(n/2) + O(1)
Sol:- $\log_b a = \log_2 1 = 0$

$f(n) = 1 = \Theta(n^{\log_2 1} \log^0 n)$

Case 2 applies and $f(n) = \Theta(n^{\log_2 1} \log^{0+1} n)$
T(n) = Θ(log(n))

Example 1.17: Binary tree traversal T(n) = 2T(n/2) + O(1)
Sol:- $\log_b a = \log_2 2 = 1$

$f(n) = 1 = O(n^{\log_2 2 - 1})$

Case 1 applies and $f(n) = \Theta(n^{\log_2 2})$
T(n) = Θ(n)

Example 1.18: Take an example T(n) = 2 T(n/2) + n^2
Sol:- $\log_b a = \log_2 2 = 1$
$f(n) = n^2 = \Omega(n^{\log_2 2 + 1})$
Case 3 applies and T(n) = Θ(f(n))
T(n) = Θ(n^2)

Example 1.19: Take an example T(n) = 4 T(n/2) + n^2
Sol:- $\log_b a = \log_2 4 = 2$

$f(n) = n^2 = \Theta(n^{\log_2 4} \log^0 n)$

Case 2 applies and $f(n) = \Theta(n^{\log_2 4} \log^{0+1} n)$
T(n) = Θ(n^2 log n)

Modified Master theorem

This is a shortcut to solve the same problem easily and quickly. If the recurrence relation is in the form of $T(n) = a\,T(n/b) + dx^s$
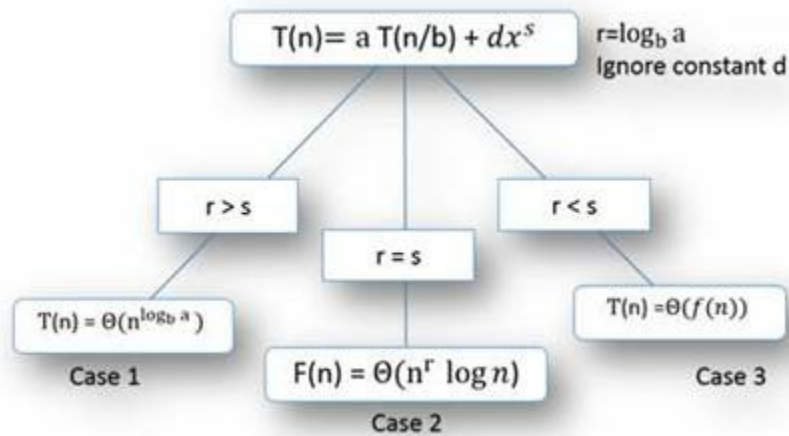
Example 1.20: $T(n) = 2\,T(n/2) + n^2$
Sol:- $r = \log_2 2$ $= 1$
$s = 2$
Case 3: $r < s$
$T(n) = \Theta\,(\,f(n)\,) = \Theta(n^2)$

Example 1.21: $T(n) = T(n/2) + 2n$
Sol:- $r = \log_2 1 = 0$
$s = 1$
Case 3
$T(n) = \Theta(n)$

Example 1.22: $T(n) = 16T(n/4) + n$
Sol:- $r = 2$
$s = 1$
Case 1
$T(n) = \Theta(n^2)$

Example 1.23: $T(n) = 2T(n/2) + n \log n$
Sol:-   There is logn in f(n) so use master theorem, shortcut will not work.
$f(n) = n \log(n) = \Theta(a^{\log_2 2} \log^1 a)$
$T(n) = \Theta(a^{\log_2 2} \log^{0+1} a) = \Theta(n \log(n))$

Example 1.24: $T(n) = 2\,T(n/4) + n^{0.5}$
Sol:- $r = \log_4 2 = 0.5 = s$
Case 2: $T(n) = \Theta(a^{\log_2 2} \log^{0.5+1} a) = \Theta(n^{0.5} \log^{1.5} n)$

Example 1.25: $T(n) = 2\,T(n/4) + n^{0.49}$

Sol:-   Case 1:

$T(n) = \Theta(a^{\log_a 2}) = \Theta(n^{0.5})$

Example 1.26: T (n) = 3T (n/3) + √ n

Sol:-   r = $\log_3 3$ = 1

s = ½

Case 1

T(n) = Θ(n)

Example 1.27:T (n) = 3T (n/4) + n log n

Sol:-   There is logn in f(n) so take a look if master theorem applies.

f(n) = n log n = $\Omega(a^{\log_4 3} \log^1 a)$

Case 3:

T(n) = Θ(n log(n))

Example 1.28: T (n) = 3T (n/3) + n/2

Sol:-   r=1=s

Case 2:

T(n) = Θ(n log(n))

Array Questions

The following section will discuss the various algorithms that are applicable to Arrays.

Sum Array

Write a method that will return the sum of all the elements of the integer array, given array as an input argument.

Example 1.29:
```
def SumArray(arr)
    size = arr.Length
    total = 0
    index = 0
    while index < size
        total = total + arr[index]
        index += 1
    end
    return total
end
```

Sequential Search

Example 1.30: Write a method, which will search an array for some given value.
```
def SequentialSearch(arr, value)
    size = arr.size
    i = 0
```

```
    while i < size
        if value == arr[i]
            return true
        end
        i += 1
    end
    return false
end
```

Analysis:
·   Since we have no idea about the data stored in the array, or if the data is not sorted then we have to search the array in sequential manner one by one.
·   If we find the value, we are looking for we return True.
·   Else, we return False in the end, as we did not find the value we are looking for.

Binary Search

If the data is sorted, a binary search can be used. We examine the middle position at each step. Depending upon the data that we are searching is greater or smaller than the middle value. We will search either the left or the right portion of the array. At each step, we are eliminating half of the search space, thereby, making this algorithm efficient as compared with the linear search.

Example 1.31: Binary search in a sorted array.
```
def BinarySearch(arr, value)
    size = arr.size
    low = 0
    high = size - 1
    while low <= high
        mid = low + (high - low) / 2 # To avoid the overflow
        if arr[mid] == value then
            return true
        elsif arr[mid] < value then
            low = mid + 1
        else
            high = mid - 1
        end
    end
    return false
end
```

Analysis:
·   Since we have data sorted in increasing / decreasing order, we can apply more efficient binary search. At each step, we reduce our search space by half.
·   At each step, we compare the middle value with the value we are searching. If mid value is equal to the value we are searching for then we return the middle index.
·   If the value is smaller than the middle value, we search the left half of the array.
·   If the value is greater than the middle value then we search the right half of the array.
·   If we find the value we are looking for then its index is returned or -1 is returned.

Rotating an array by K positions.

Given an array you need to rotate its elements K number of times. For example, an array [10,20,30,40,50,60] rotate by 2 positions to [30,40,50,60,10,20]

Example 1.32:

```
def reverseArray(a, start, end1)
    i = start
    j = end1
    while i < j
        temp = a[i]
        a[i] = a[j]
        a[j] = temp
        i += 1
        j -= 1
    end
end

def rotateArray(a, k)
    n = a.length()
    reverseArray(a, 0, k - 1)
    reverseArray(a, k, n - 1)
    reverseArray(a, 0, n - 1)
end

# Testing code
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
rotateArray(arr, 6)
print arr
```

Analysis:
·   Rotating array is done in two parts trick. In the first part, we first reverse elements of array first half and then second half.

   1,2,3,4,5,6,7,8,9,10   =>   5,6,7,8,9,10,1,2,3,4

   1,2,3,4,5,6,7,8,9,10   =>   4,3,2,1,10,9,8,7,6,5   =>   5,6,7,8,9,10,1,2,3,4

·   Then we reverse the whole array there by completing the whole rotation.

Find the largest sum contiguous subarray.

Given an array of positive and negative integers, find a contiguous subarray whose sum (sum of elements) is maximum.

Example 1.33:

```
def maxSubArraySum(a)
    size = a.length()
    maxSoFar = 0
    maxEndingHere = 0
```

```
        i = 0
        while i < size
            maxEndingHere = maxEndingHere + a[i]
            if maxEndingHere < 0
                maxEndingHere = 0
            end
            if maxSoFar < maxEndingHere
                maxSoFar = maxEndingHere
            end
            i += 1
        end
        return maxSoFar
end

# Testing code
arr = [1, -2, 3, 4, -4, 6, -4, 8, 2]
print maxSubArraySum(arr)
```

Analysis:
·   Maximum subarray in an array is found in a single scan. We keep track of global maximum sum so far and the maximum sum, which include the current element.
·   When we find global maximum value so far is less than the maximum value containing current value we update the global maximum value.
·   Finally return the global maximum value.

Recursive Function

A recursive function is a function that calls itself, directly or indirectly.

A recursive method consists of two parts: Termination Condition and Body (which includes recursive expansion).
1.   **Termination Condition:** A recursive method always contains one or more terminating condition. A condition in which recursive method processes a simple case and does not call itself.
2. **Body** (including recursive expansion): The main logic of the recursive method is contained in the body of the method. It also contains the recursion expansion statement that, in turn, calls the method itself.

Three important properties of recursive algorithm are:
1) A recursive algorithm must have a termination condition.
2) A recursive algorithm must change its state, and move towards the termination condition.
3) A recursive algorithm must call itself.

Note: The speed of a recursive program is slower because of stack overheads. If the same task can be done using an iterative solution (using loops), then we should prefer an iterative solution in place of recursion to avoid stack overhead.
Note: Without termination condition, the recursive method may run forever and will finally consume all the stack memory.

Factorial

Example 1.34: Factorial Calculation. N! = N* (N-1)…. 2*1.

```
def factorial(i)
    # Termination Condition
    if i <= 1 then
        return 1
    end
    # Body, Recursive Expansion
    return i * factorial(i - 1)
end

# Testing code
puts factorial(10)
```

Analysis: Each time method fn is calling fn-1. Time Complexity is **O(N)**

Print Base 16 Integers

Example 1.35: Generic print to some specific base method.

```
def printInt2(number, baseValue)
    conversion = "0123456789ABCDEF"
    digit = (number % baseValue)
    number = number / baseValue
    if number != 0 then
        printInt2(number, baseValue)
    end
    print conversion[digit]
end

# Testing code
print printInt2(75, 16)
```

Analysis:
·   Base value is provided along with the number in the function parameter.
·   Remainder of the number is calculated and stored in digit.
·   If the number is greater than base then, number divided by base is passed as an argument to the printInt() method recursively.
·   Number will be printed with higher order first then the lower order digits.
Time Complexity is **O(N)**

Tower of Hanoi

The **Tower of Hanoi** (also called the **Tower of Brahma**) We are given three rods and N number of disks, initially all the disks are added to first rod (the leftmost one) is in decreasing size order. The objective is to transfer the entire stack of disks from first tower to third tower (the rightmost one), moving only one disk at a time and never a larger one onto a smaller.
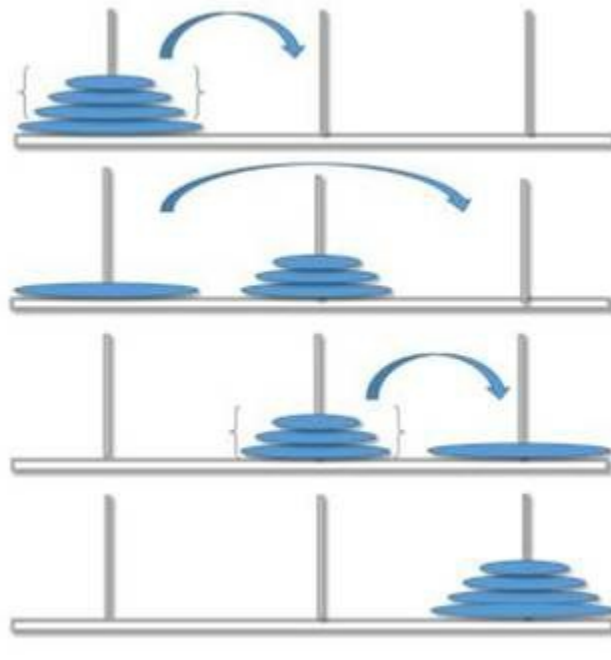
Example 1.36:

```
def towerOfHanoi(num, src, dst, temp)
    if num < 1
        return
    end
    towerOfHanoi(num - 1, src, temp, dst)
    print "\n Move " , num , " disk from peg " , src , " to peg " , dst
    towerOfHanoi(num - 1, temp, dst, src)
end


num = 4
print "The sequence of moves involved in the Tower of Hanoi are :"
towerOfHanoi(num, 'A', 'C', 'B')
```

Analysis: If we want to move N disks from source to destination, then we first move N-1 disks from source to temp, then move the lowest Nth disk from source to destination. Then it will move N-1 disks from temp to destination.

Greatest common divisor (GCD)

Example 1.37: Find the greatest common divisor.

```
def GCD(m, n)
    if m < n
        return GCD(n, m)
    end
    if m % n == 0
        return (n)
    end
    return GCD(n, m % n)
end
```

```
print "GCD : " ,GCD(7, 3)
```

Analysis: Euclid's algorithm is used to find gcd. GCD(n, m) == GCD(m, n mod m).

Fibonacci number

Example 1.38: Given N, find the Nth number in the febonacci series. .
```
def fibonacci(n)
    if n <= 1 then
        return n
    end
    return fibonacci(n - 1) + fibonacci(n - 2)
end

# Testing code
puts fibonacci(10)
```

Analysis: Fibonacci numbers are calculated by adding sum of the previous two numbers.

Note:- There is an inefficiency in the solution we will look better solution in coming chapters.

All permutations of an integer array

Example 1.39: Generate all permutations of an integer array.
```
def permutation(arr, i, length)
    if length == i
        printArray(arr)
        return
    end
    j = i
    j = i
    while j < length
        swap(arr, i, j)
        permutation(arr, i + 1, length)
        swap(arr, i, j)
        j += 1
    end
    return
end

arr = [1,2,3,4,5]
permutation(arr, 0, 5)
```

Analysis: In permutation method at each recursive call number at index, "i" is swapped with all the numbers that are right of it. Since the number is swapped with all the numbers in its right one by one it will produce all the permutation possible.

Binary search using recursion

Example 1.40: Search a value in an increasing order sorted array of integers.

```
# Binary Search Algorithm - Recursive Way
def BinarySearchRecursive(arr, value)
    return BinarySearcheUtil(arr, 0, arr.size, value)
end

def BinarySearcheUtil(arr, low, high, value)
    if low > high
        return false
    end
    mid = low + (high - low) / 2 # To avoid the overflow
    if arr[mid] == value then
        return true
    elsif arr[mid] < value then
        return BinarySearcheUtil(arr, mid + 1, high, value)
    else
        return BinarySearcheUtil(arr, low, mid - 1, value)
    end
end

# Testing Code
arr = [1,3,5,6,8,9,11,14,17,18]
puts BinarySearchRecursive(arr, 9)
puts BinarySearchRecursive(arr, 7)
```

Analysis: Similar iterative solution we have already seen. Now let us look into the recursive solution of the same problem. In this solution, we are diving the search space into half and discarding the rest. This solution is very efficient as at each step we are rejecting half the search space/ array.

Exercises

1. Find average of all the elements in an array.

2. Find the sum of all the elements of a two dimensional array.

3. Find the largest element in the array.

4. Find the smallest element in the array.

5. Find the second largest number in the array.

6. Using AllPermutation function discussed before, write a function, which give only distinct solutions.

7. Write a method to compute Sum(N) = 1+2+3+…+N.

8. Print all the maxima's in an array. (A value is a maximum if the value before and after its index are smaller than it does or does not exist.)
   Hint: Start traversing array from the end and keep track of the max element. If we encounter an element whose value is grater then max, print the element and update max.

9. Given an array of intervals, merge all overlapping intervals.
   Input: {[1, 4], [3, 6], [8, 10]},   Output: {[1, 6], [8, 10]}

10. Reverse an array in-place. (You cannot use any additional array in other wards Space Complexity should be **O(1).)**
    Hint: Use two variable, start and end. Start set to 0 and end set to (n-1). Increment starts and decrement ends. Swap the values stored at arr[start] and arr[end]. Stop when start is equal to end or start is greater than end.

11. Given an array of 0s and 1s. We need to sort it so that all the 0s are places before all the 1s.
    Hint: Use two variable, start and end. Start set to 0 and end set to (n-1). Increment starts and decrement ends. Swap the values stored at arr[start] and arr[end] only when arr[start] == 1 and arr[end] ==0. Stop when start is equal to end or start is greater than end.

12. Given an array of 0s, 1s and 2s. We need to sort it so that all the 0s are placed before all the 1s and all the 1s are placed before 2s.
    Hint: Same as above first think 0s and 1s as one group and move all the 2s on the right side. Then do a second pass over the array to sort 0s and 1s.

13. Find the duplicate elements in an array of size n where each element is in the range 0 to n-1.
    Hint:
    Approach 1: Compare each element with all the elements of the array (using two loops) $O(n^2)$ solution
    Approach 2: Maintain a Hash-Table. Set the hash value to 1 if we encounter the element for the first time. When we put same value again we can see that the hash value is already 1 so we can print that value. **O(n)** solution, but additional space is required.
    Approach 3: We will exploit the constraint "every element is in the range 0 to n-1".   We can take an array arr[] of size n and set all the elements to 0. Whenever we get a value say val1. We will increment the value at arr[var1] index by 1. In the end, we can traverse the array arr and print the repeated values. Additional Space Complexity will be **O(n)** which will be less than Hash-Table approach.

14. Find the maximum element in a sorted and rotated array. Complexity: **O(logn)**
    Hint: Use binary search algorithm.

15. Given an array with 'n' elements & a value 'x', find two elements in the array that sums to 'x'.
    Hint:
    Approach 1: Sort the array.
    Approach 2: Using a Hash-Table.

16. Write a method to find the sum of every number in an int number. Example: input= 1984, output should be 32 (1+9+8+4).

17. True or false
   a. $5n + 10n^2 = O(n^2)$
   b. $n \log n + 4n = O(n)$
   c. $\log(n^2) + 4 \log(\log n) = O(\log n)$
   d. $12 n^{1/2} + 3 = O(n^2)$
   e. $3^n + 11 n^2 + n^{20} = O(2^n)$

18. What is the best-case runtime complexity of searching an array?

19. What is the average-case runtime complexity of searching an array?

CHAPTER 2: APPROACH TO SOLVE ALGORITHM DESIGN PROBLEMS

Introduction

Theoretical knowledge of the algorithm is essential, yet it is not sufficient. When an interviewer asks to develop a program in an interview, than interviewee should follow our five-step approach to solve it. Master this approach and you will perform better than most of the candidates in interviews.

Five steps for solving algorithm design questions are:
1. Constraints
2. Ideas Generation
3. Complexities
4. Coding
5. Testing

Constraints

Solving a technical question is not just about knowing the algorithms and designing a good software system. The interviewer wants to know you approach towards any given problem. Many people make mistakes, as they do not ask clarifying questions about a given problem? They assume many things simultaneously and begin working with that. There is lot of data that is missing that you need to collect from your interviewer before beginning to solve a problem.

In this step, you will capture all the constraints about the problem. We should never try to solve a problem that is not completely defined. Interview questions are not like exam paper where all the details about a problem are well defined. In the interview, the interviewer actually expects you to ask questions and clarify the problem.

For example, when the problem statement says that write an algorithm to sort numbers.
1. The first information you need to capture is what kind of data is provided. Let us assume interviewer respond with the answer Integer.
2. The second information that you need to know what is the size of data. Your algorithm differs if the input data size if 100 integers or 1 billion integers.

Basic guideline for the Constraints for a list of numbers:
1. How many numbers of elements are there in the list?
2. What is the range of value in each element? What is the min and max value?
3. What is the kind of data in each element? Is it an integer or a floating point?
4. Does the list contain unique data or not?

Basic guideline for the Constraints for a list of string:
1. How many numbers of elements are there in the list?

2. What is the length of each string? What is the min and max length?
3. Does the list contain unique data or not?

Basic guideline for the Constraints for a Graph
1. How many nodes are there in the graph?
2. How many edges are there in the graph?
3. Is it a weighted graph? What is the range of weights?
4. Is the graph directed or undirected?
5. Is there is a loop in the graph?
6. Is there negative sum loop in the graph?
7. Does the graph have self-loops?

We will see this in graph chapter that depending upon the constraints the algorithm applied changes and so is the complexity of the solution.

Idea Generation

We will cover a lot of theoretical knowledge in this book. It is impossible to cover all the questions as new ones are created every day. Therefore, we should know how to handle new problems. Even if you know the solution of a problem asked by the interviewer then also you need to have a discussion with the interviewer and try to reach to the solution. You need to analyse the problem also because the interviewer may modify a question a little bit so the approach to solve it will vary.

How to solve an unseen problem? The solution to this problem is that you need to do a lot of practice and the more you will practise the more you will be able to solve any unseen question, which come before you. When you have solved enough problems, you will be able to see a pattern in the questions and will be able to solve unseen problems easily.

Following is the strategy that you need to follow to solve an unknown problem:
1. Try to simplify the task in hand.
2. Try a few examples
3. Think of a suitable data-structure.
4. Think about similar problems that you have already solved.

Try to simplify the task in hand

Let us look into the following problem: Husbands and their wives are standing in random in a line. They have been numbered, for husbands H1, H2, H3 and so on. Their corresponding wives have been numbered, W1, W2, W3 and so on. You need to arrange them so that H1 will stand first, followed by W1, then H2 followed by W2 and so on.
At the first look, it looks difficult, but it is a simple problem. Try to find a relation of the final position.
$P(Hi) = i*2 - 1$, $\qquad P(Wi) = i*2$
For rest of the algorithm we are leaving you to do something like Insertion-Sort and you are done.

In the above problem if you have tried it with some example for 3 husband-wife pair then you will reach to the same formula that we have shown in the previous section. Sometime applying some more examples will help you to solve the problem.

Think of a suitable data-structure

For some problems, it is straightforward to choose which data structure will be most suitable. For example, if we have a problem finding min/max of some given value, then probably heap is the data structure we are looking for. We have seen a number of data structure throughout this book. We have to figure out which data-structure will suite our need.

Let us look into a problem: We are given a stream of data, at any time we can be asked to tell the median value of the data and maybe we can be asked to pop median data.

We can think about some sort of tree, may be balanced tree where the root is the median. Wait! It is not so easy to make sure the tree root to be a median.

A heap can give us minimum or maximum so we cannot get the desired result from it too. However, what if we use two heap one max heap and one min heap. The smaller values will go to max heap and the bigger values will go to min heap. In addition, we can keep the count of how many elements are there in the heap. The rest of the algorithm you can think yourself.

For every unseen problem think about the data structures, you know and may be one of them or some combination of them will solve your problem.

Think about similar problems you have already solved. Let us suppose you are given, two linked list head reference and they meet at some point and need to find the point of intersection. However, in place of the end of both the linked list to be a null reference, there is a loop.

You know how to find intersection point of two intersecting linked-list, you know how to find if a linked list have a loop (three-reference solution). Therefore, you can apply both of these solutions to find the solution of the problem in hand.

Complexities

Solving a problem is not just finding a correct solution. The solution should be fast and should have reasonable memory requirement. You have already read about Big-O notation in the previous chapters. You should be able to do Big-O analysis. In case you think the solution you have provided is not optimal and there may be more efficient solution, then think again and try to figure out this information.

Most interviewers expect that you should be able to find the time and Space Complexity of the algorithms. You should be able to compute the time and Space Complexity instantly. Whenever you are solving any problem, you should find the complexity associated with it from this you would be able to choose the best solutions. In some problems there is some trade-offs between

space and Time Complexity, so you should know these trade-offs. Sometime taking some bit more space saves a lot of time and make your algorithm much faster.

Coding

At this point, you have already captured all the constraints of the problem, proposed few solutions, evaluated the complexities of the various solutions and picked the solution to do final coding. Never ever, jump into coding before discussing constraints, Idea generation and complexity with the interviewer.

We are accustomed to coding in an IDE like visual studio. So many people struggle when asked to write code on a whiteboard or some blank sheet. Therefore, we should have a little practice to the coding on a sheet of paper. You should think before coding because there is no back button in sheet of paper. Always try to write modular code. Small functions need to be created so that the code is clean and managed. If there is a swap function so just use this function and tell the interviewer that you will write it later. Everybody knows that you can write swap code.

Testing

Once the code is written, you are not yet done. It is most important that you test your code with several small test cases. It shows that you understand the importance of testing. It also gives confidence to your interviewer that you are not going to write a buggy code. Once you are done with, your coding, it is a good practice that you go through your code line-by-line with some small test cases. This is just to make sure that your code is working as it is supposed to work.

You should test few test cases.
Normal test cases: These are the positive test cases, which contain the most common scenario, and focus is on the working of the base logic of the code. For example, if we are solving some problems for linked list, then this test may contain, what will happen when a linked list with 3 or 4 nodes is given as input. These test cases you should always contemplate before stating that the code is done.

Edge cases: These are the test cases, which are designed to test the boundaries of the code. For the same linked list algorithm, edge cases may be created to test how the code behaves when an empty list is passed or just one node is passed. Edge cases may help to make your code more robust. Just few checks need to be added to the code to take care of these conditions.

Load testing: In this kind of test, your code will be tested with a huge data. This will allow us to test if your code is slow or too much memory intensive.

Always follow these five steps never jump to coding before doing constraint analysis, idea generation, and Complexity Analysis: At least, never miss the testing phase.

Example

Let us suppose the interviewer ask you to give a best sorting algorithm.
Some interviewee will directly jump to Quick-Sort **O(nlogn)**. Oops, mistake! You need to ask

many questions before beginning to solve this problem.

Questions 1: What is the kind of data? Are they integers?
Answer: Yes, they are integers.

Questions 2: How much data are we going to sort?
Answer: May be thousands.

Questions 3: What exactly is this data about?
Answer: They store a person's age

Questions 4: What kind of data-structure is used to hold this data?
Answer: Data are given in the form of some list

Questions 5: Can we modify the given data-structure? In addition, many, many more…?
Answer: No, you cannot modify the data structure provided

Ok, from the first answer, we will deduce that the data is integer. The data is not very big it just contains a few thousand entries. The third answer is interesting, from this we deduce that the range of data is 1-150. Data is provided in a list. From fifths answer we deduce that we have to create our own data structure and we cannot modify the list provided. So finally, we conclude, we can just use bucket sort to sort the data. The range is just 1-150 so we need just 151-capacity integral list. Data is under thousands so we do not have to worry about data overflow and we get the solution in linear time **O(N)**.

Note: We will read sorting in the coming chapters.

Summary

At this point, you know the process of handling unseen problems very well. In the coming chapter we will be looking into a lot of various data structures and the problems they solve. It may be possible that the user is not able to understand some portion of this chapter as knowledge of rest of the book is needed, so they can read this chapter again after they have read the rest of the data structures portion. A huge number of problems are solved in this book. However, it is recommended that first try to solve them by yourself, and then look for the solution. Always think about the complexity of the problem. In the interview interaction is the key to get problem described completely and discuss your approach with the interviewer.

CHAPTER 3: ABSTRACT DATA TYPE & RUBY COLLECTIONS

Abstract data type (ADT)

An abstract data type (ADT) is a logical description of how we view the data and the operations that are allowed on it. ADT is defined as a user point of view of a data type. ADT concerns about the possible values of the data and what are interface exposed by it.

ADT does not concern about the actual implementation of the data structure.

For example, a user wants to store some integers and find a mean of it. Does not talk about how exactly it will be implemented.
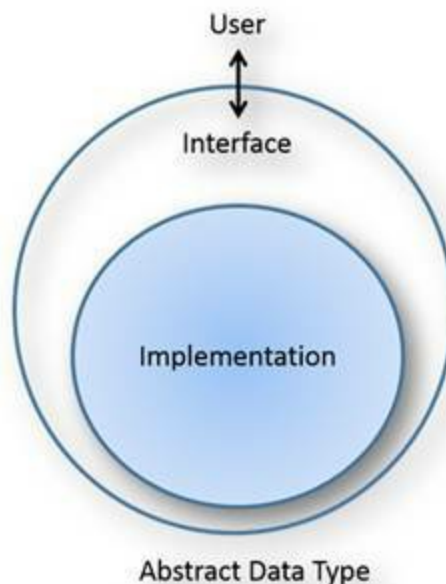
Data-Structure

Data structures are concrete representations of data and are defined as a programmer point of view. Data-structure represents how data will be stored in memory. All data-structures have their own pros and cons. Depending upon the type problem we pick a data-structure that is best suited for it.
For example, we can store data in an array, a linked-list, stack, queue, tree, etc.

Ruby Collection Framework

Ruby programming language provides a Ruby Collection Framework, which is a set of high quality, high performance & reusable data-structures and algorithms.
The following are the advantages of using a Ruby collection framework:
1. Programmers do not have to implement basic data structures and algorithms repeatedly.

Thereby it prevents the reinvention the wheel. Thus, the programmer can devote more effort in business logic.

2. The Ruby Collection Framework code is well-tested, high quality and high performance code there by enhance the quality of the programs.
3. Development cost is reduced as basic data structures and algorithms are implemented in Collections framework.
4. Easy for the reviewing and understanding other programs as other developers also use the Collection framework. In addition, collection framework is well documented.

Array

Array represents a collection of multiple elements of the same datatypes. Arrays are variable length data structure. The size of this data structure is variable and with the number of elements inside.

Array ADT Operations

Below is the API of array:

1. Adds an element at kth position. Value can be stored in array at Kth position in **O(1)** constant time. We just need to store value at arr[k].
2. Reading the value stored at kth position. Accessing the value stored at some index in array is also **O(1)** constant time. We just need to read value stored at arr[k].
3. Substitution: change the value stored in kth position with a new value. Time complexity: **O(1)** constant time.

Example 3.1

```
arr = Array.new(10)
i = 0
while i < 10
    arr[i] = i
    i += 1
end
print arr
```

Linked List
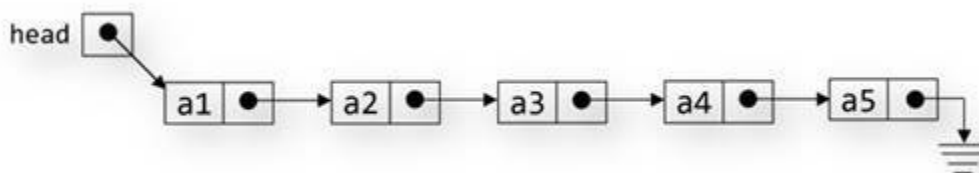
Linked lists are dynamic data structure and memory is allocated at run time. The concept of linked list is not to store data contiguously. Use links that point to the next elements.

Performance wise linked lists are slower than lists because there is no direct access to linked list elements. The linked list is a useful data structure when we do not know the number of elements to be stored ahead of time. There are many types of linked list: linear, circular, doubly, and doubly circular.

Linked List ADT Operations

Below is the API of Linked list.

Insert(k): adds k to the start of the list
Insert an element at the start of the list. Just create a new element and move references. So that this new element becomes the new element of the list. This operation will take **O(1)** constant time.

Delete(): delete element at the start of the list
Delete an element at the start of the list. We just need to move one reference. This operation will also take **O(1)** constant time.

PrintList(): display all the elements of the list.
Start with the first element and then follow the references. This operation will take **O(N)** time.

Find(k): find the position of element with value k
Start with the first element and follow the reference until we get the value we are looking for or reach the end of the list. This operation will take **O(N)** time.
Note: binary search does not work on linked lists.

FindKth(k): find element at position k
Start from the first element and follow the links until you reach the kth element. This operation will take **O(N)** time.

IsEmpty(): check if the number of elements in the list are zero.
Just check the head reference of the list it should be Null. Which means list is empty. This operation will take **O(1)** time.

Stack
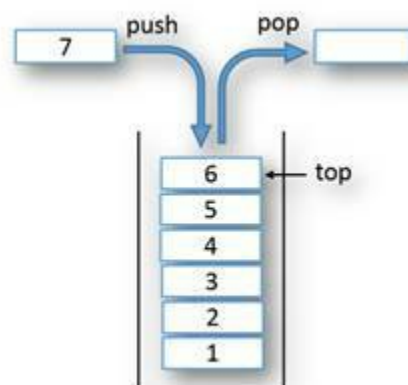
Stack is a special kind of data structure that follows Last-In-First-Out (LIFO) strategy. This means that the element that is added last will be the first to be removed.

The various applications of stack are:
1. Recursion: recursive calls are preformed using system stack.
2. Postfix evaluation of expression.

3. Backtracking
4. Depth-first search of trees and graphs.
5. Converting a decimal number into a binary number etc.

Stack ADT Operations

Push(k): Adds a new item to the top of the stack
Pop(): Removes an element from the top of the stack and return its value.
Top(): Returns the value of the element at the top of the stack
Size(): Returns the number of elements in the stack
IsEmpty(): determines whether the stack is empty. It returns 1 if the stack is empty or return 0.
Note: All the above Stack operations are implemented in **O(1)** Time Complexity.

Stack implementation using Ruby Array

Stack is implemented using Array collection. Array behave as stack if we add data using insert() and remove using pop() function.

Example 3.2
```
stk = []
stk.push(1)
stk.push(2)
stk.push(3)
stk.push(4)

size = stk.size
print "Element at top of stack ::" , stk[size -1], "\n"
i = 0
while i < size
    print "Pop from stack: " , stk.pop(), "\n"
    i += 1
end
```

Output
```
Element at top of stack ::4
Pop from stack: 4
Pop from stack: 3
Pop from stack: 2
Pop from stack: 1
```

Queue

dequeue   enqueue

A queue is a First-In-First-Out (FIFO) kind of data structure. The element that is added to the queue first will be the first to be removed and so on.

Queue has the following application uses:
1. Access to shared resources (e.g., printer)
2. Multiprogramming
3. Message queue

Queue ADT Operations:

Add(): Adds a new element to the back of the queue.

Remove(): Removes an element from the front of the queue and return its value.

Front(): Returns the value of the element at the front of the queue.

Size(): Returns the number of elements inside the queue.

IsEmpty(): Returns 1 if the queue is empty otherwise returns 0

Note: All the above Queue operations are implemented in **O(1)** Time Complexity.

Queue implementation in Ruby Collection

Deque is the class implementation of doubly ended queue. If we use append(), popleft() it will behave as a queue.

Example 3.3
```
que = Queue.new()
que.push(1)
que.push(2)
que.push(3)
que.push(4)
que.push(5)
size = que.size
i = 0

while i < size
    print "Dequeue from queue: " , que.pop(), "\n"
    i += 1
end
```

Output

Note:- Do not use normal List to implement queue. List is implemented using dynamic array so it is not optimized for queue Implementation. If we use List, then we need to do insertion and removal at different end and each insertion will do shifting of all the elements that will be inefficient and will take O(N) time.

Tree

Tree is a hierarchical data structure. The top element of a tree is called the root of the tree. Except the root element, every element in a tree has a parent element, and zero or more child elements. The tree is the most useful data structure when you have hierarchical information to store.
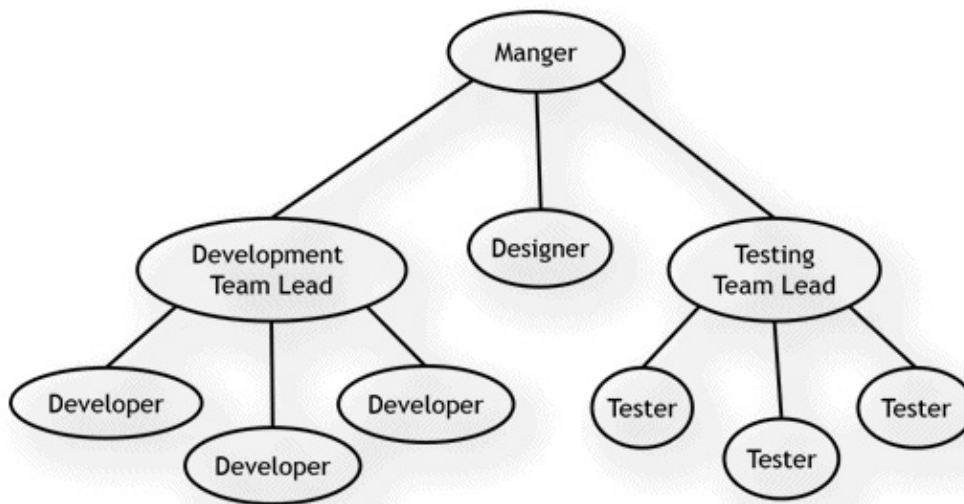
There are many types of trees, for example, binary-tree, Red-black tree, AVL tree, etc.

Binary Tree

A binary tree is a type of tree in which each node has at most two children (0, 1 or 2) which are referred as left child and right child.

Binary Search Trees (BST)
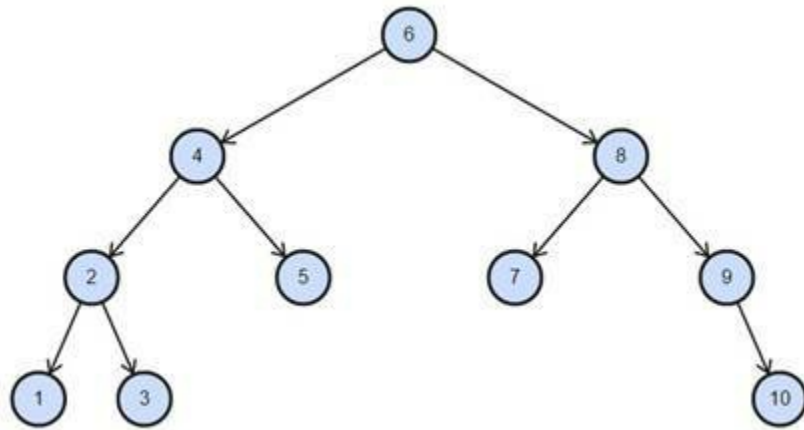
A binary search tree (BST) is a binary tree on which nodes are ordered in the following way:
1. The key in the left subtree is less than the key in its parent node.
2. The key in the right subtree is greater or equal to the key in its parent node.

Binary Search Tree ADT Operations

Insert(k): Inserts an element k into the tree.
Delete(k): Deletes an element k from the tree.
Search(k): Searches a particular value k into the tree if it is present or not.
FindMax(): Finds the maximum value stored in the tree.
FindMin(): Finds the minimum value stored in the tree.

The average Time Complexity of all the above operations on a binary search tree is O(log n), the case when the tree is balanced. The worst-case Time Complexity will be **O(n)** when the tree is skewed. A binary tree is skewed when tree is not balanced.

There are two types of skewed tree.
1. Right Skewed binary tree: A binary tree in which each node is having either a right child or no child.
2. Left Skewed binary tree: A binary tree in which each node is having either a left child or no child.

Balanced Binary search tree

There are few binary search tree, which always keeps themselves balanced. Most important among them are Red-Black Tree (RB-Tree) and AVL tree. Ordered dictionary in collections is implemented using RB-Tree.

Priority Queue (Heap)
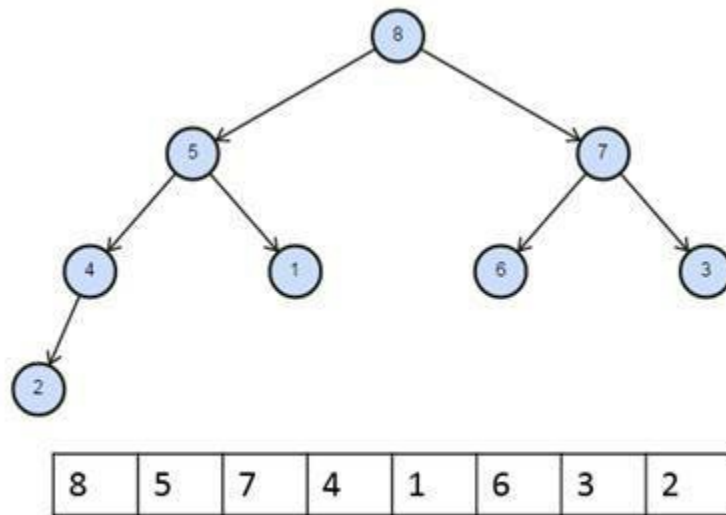
Max Heap
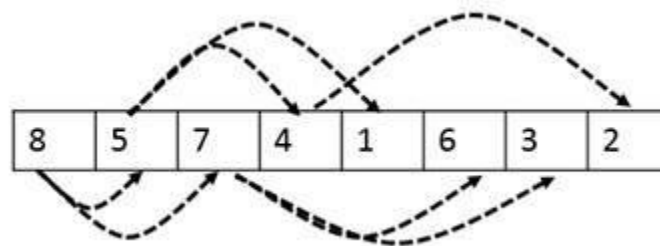
Priority queue is implemented using a binary heap data structure. In a heap, the records are stored in an array. Each node in the heap follows the same rule that the parent value is greater than its children.

There are two types of the heap data structure:
1. Max heap: each node should be greater than or equal to each of its children.
2. Min heap: each node should be smaller than or equal to each of its children.

A heap is a useful data structure when you want to get max/min value one by one from data. Heap-Sort uses max heap to sort data in increasing/decreasing order.

Heap ADT Operations

Insert() - Adds a new element to the heap. The Time Complexity of this operation is O(log(n))
Remove() - Extracts max for max heap case (or min for min heap case). The Time Complexity of this operation is O(log(n))
Heapify() – Converts a list of numbers in a list into a heap. This operation has a Time Complexity **O(n)**

Hash-Table
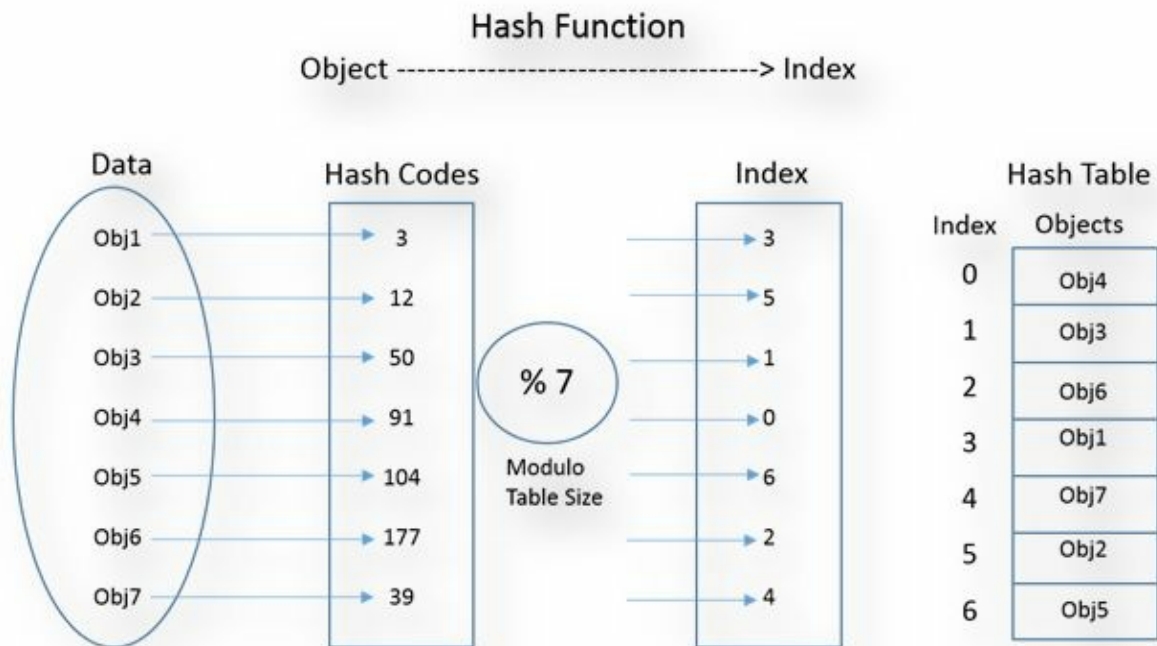
Hash Function

Object -------------------------------------> Index

A Hash-Table is a data structure that maps keys to values. Each position of the Hash-Table is called a slot. The Hash-Table uses a hash function to calculate an index of a list. We use the Hash-Table when the number of keys actually stored are small relatively to the number of possible keys.

The process of storing objects using a hash function is as follows:
1. Create a list of size M to store objects, this list is called Hash-Table.
2. Find a hash code of an object by passing it through the hash function.
3. Take module of hash code by the size of Hash-Table to get the index where objects will be stored.
4. Finally store these objects in the designated index.

The process of searching objects in Hash-Table using a hash function is as follows:
1. Find a hash code of the object we are searching for by passing it through the hash function.
2. Take module of hash code by the size of Hash-Table to get the index of the table where objects are stored.
3. Finally, retrieve the object from the designated index.

Hash-Table Abstract Data Type (ADT)

ADT of Hash-Table contains the following functions:
Insert(x): Adds object x to the data set.
Delete(x): Deletes object x from the data set.
Search(x): Searches object x in data set.

The Hash-Table is a useful data structure for implementing dictionary. The average time to search for an element in a Hash-Table is **O(1)**. A Hash Table generalizes the notion of a list.

Dictionary in Ruby Collection

A Dictionary is a data structure that maps keys to values. A Dictionary uses a hash table so the key value pairs are not stored in sorted order. Dictionary does not allow duplicate keys but values can be duplicate.

Example 3.4

```ruby
# Create a Dictionary or map.
hm = {} # (or hm = Hash.new)

# Put elements into the Dictionary or map
hm["Apple"] = 40
hm["Banana"] = 30
hm["Mango"] = 50
puts "Total number of fruits :: #{hm.size}"

hm.each do |key, value|
    puts "#{key} cost :#{value}"
end
hm.delete("Mango")
puts "Apple present :: #{hm.key?("Apple")}"
puts "Mango present :: #{hm.key?("Mango")}"
puts "Grape present :: #{hm.key?("Grape")}"
```

Output

```
Total number of fruits :: 3
Apple cost :40
Banana cost :30
Mango cost :50
Apple present :: true
Mango present :: false
Grape present :: false
```

Set

Set is a class, which is used to store only unique elements. Set is implemented using a hash table. Since Set is implemented using a hash table its elements are not stored in sequential order.

Sets
- Sets are similar to dictionaries in Ruby, except that they consist of only keys with no associated values.
- Essentially, they are a collection of data with no duplicates.
- They are very useful when it comes to remove duplicate data from data collections.

Example 3.5

```ruby
class Set
    def initialize()
        @hm = {}
    end
```

```
    def Insert(key)
        @hm[key] = 1
    end

    def Delete(key)
        @hm.delete(key)
    end

    def Has(key)
        return @hm.key?(key)
    end

    def Size()
        return @hm.size
    end
end

# Testing Code
cm = Set.new()
cm.Insert(2)
print "\n2 in set : " , cm.Has(2)
cm.Delete(2)
print "\n2 in set : " , cm.Has(2)
```

Output
```
2 in set : true
2 in set : false
```

Counter

Counters are used to count the number of occurrence of values.

Example 3.6
```
class Counter
    def initialize()
        @hm = {}
    end

    def add(key)
        if @hm.key?(key) then
            count = @hm[key]
            @hm[key] = count + 1
        else
            @hm[key] = 1
        end
    end

    def remove(key)
```

```ruby
        if @hm.key?(key) then
            if @hm[key] == 1 then
                @hm.delete(key)
            else
                count = @hm[key]
                @hm[key] = count - 1
            end
        end
    end

    def get(key)
        if @hm.key?(key) then
            return @hm[key]
        end
        return 0
    end

    def containsKey(key)
        return @hm.key?(key)
    end

    def size()
        return @hm.size
    end
end

# Testing Code
cm = Counter.new()
cm.add(2)
cm.add(2)
print "\n 2 count is : " , cm.get(2)
cm.remove(2)
print "\n 2 count is : " , cm.get(2)
```

Output

```
2 count is : 2
2 count is : 1
```

Dictionary / Symbol Table

A symbol table is a mapping between a string (key) and a value, which can be of any data type. A value can be an integer such as occurrence count, dictionary meaning of a word and so on.

Binary Search Tree (BST) for Strings

Binary Search Tree (BST) is the simplest way to implement symbol table. Simple string compare function can be used to compare two strings. If all the keys are random, and the tree is balanced. Then on an average key lookup can be done in logarithmic time.
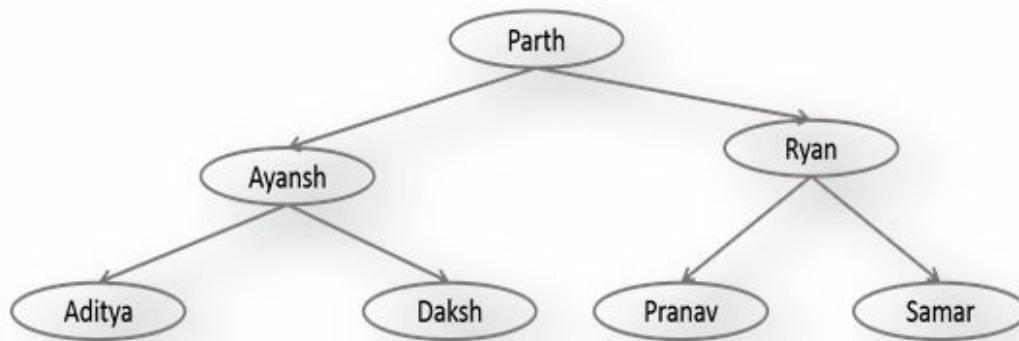
BINARY SEARCH TREE AS DICTIONARY

Hash-Table

The Hash-Table is another data structure, which can be used for symbol table implementation. Below in the Hash-Table diagram, we can see the name of that person is taken as the key, and their meaning is the value of the search. The first key is converted into a hash code by passing it to appropriate hash function. Inside hash function the size of Hash-Table is also passed, which is used to find the actual index where values will be stored. Finally, the value that is meaning of name is stored in the Hash-Table.
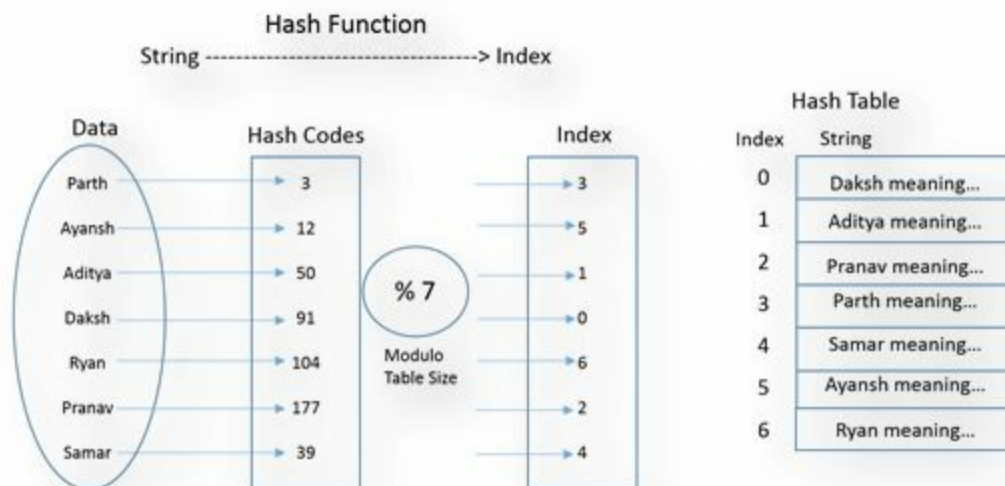
Hash-Table has an excellent lookup of constant time.

Let us suppose we want to implement autocomplete the box feature of Google search. When you type some string to search in google search, it proposes some complete string even before you have done typing. BST cannot solve this problem as related strings can be in both right and left subtree.

The Hash-Table is also not suited for this job. One cannot perform a partial match or range query on a Hash-Table. Hash function transforms string to a number. Moreover, a good hash function will give a distributed hash code even for partial string and there is no way to relate two strings in a Hash-Table.

Trie and Ternary Search tree are a special kind of tree, which solves partial match, and range query problem well.

Trie

Trie is a tree, in which we store only one character at each node. This final key value pair is stored in the leaves. Each node has K children, one for each possible character. For simplicity purpose, let us consider that the character set is 26, corresponds to different characters of English alphabets.

Trie is an efficient data structure. Using Trie, we can search the key in O(M) time. Where M is the maximum string length. Trie is suitable for solving partial match and range query problems.

TRIE DICTIONARY
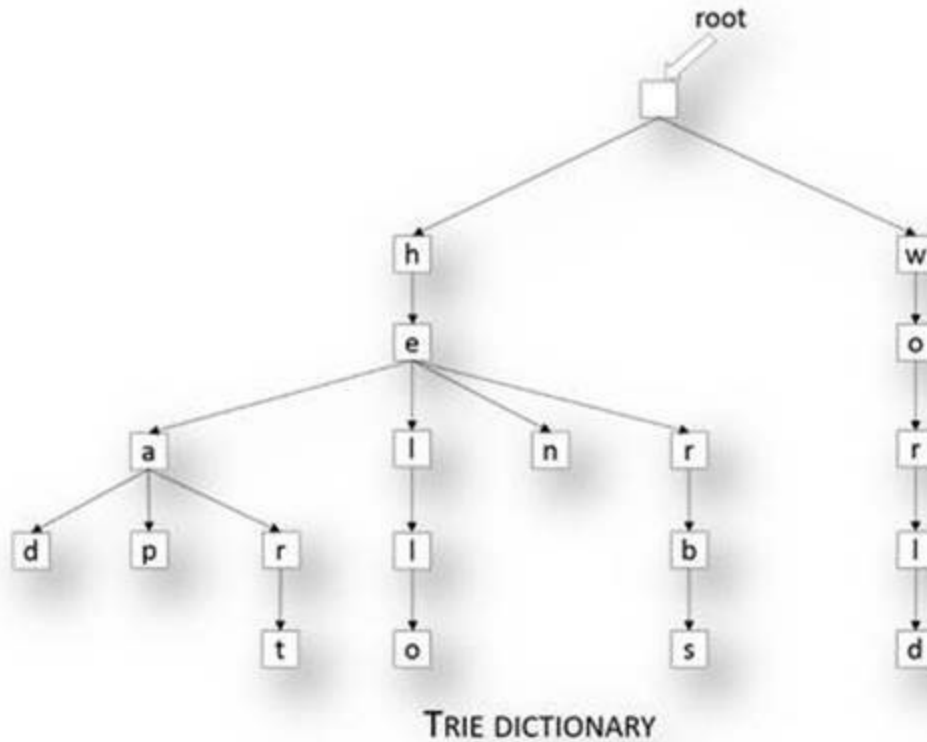
Ternary Search Trie/ Ternary Search Tree

Tries having a very good search performance of O(M) where M is the maximum size of the search string. However, tries having very high space requirement. Every node Trie contains references to multiple nodes, each reference corresponds to possible characters of the key. To avoid this high space requirement Ternary Search Trie (TST) is used. A TST avoids the heavy space requirement of the traditional Trie while keeping many of its advantages. In a TST, each node contains a character, an end of key indicator, and three references. The three references are corresponding to current char hold by the node(equal), characters less than and character greater than.

The Time Complexity of ternary search tree operation is proportional to the height of the ternary search tree. In the worst case, we need to traverse up to 3 times that many links. However, this case is rare. Therefore, TST is a very good solution for implementing Symbol Table, Partial match and range query.
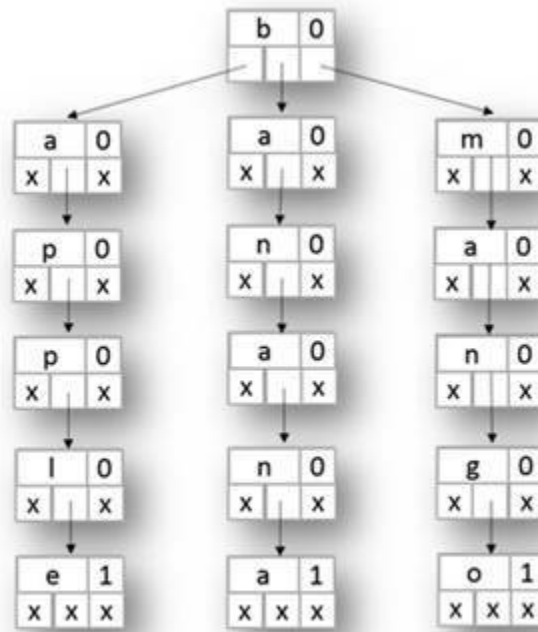
Ternary Search Tree
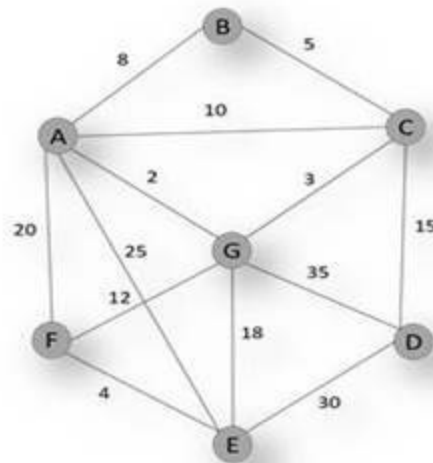
Graphs

A graph is a data structure that represents a network that connects a collection of nodes called vertices, and their connections, called edges. An edge can be seen as a path between two nodes. These edges can be either directed or undirected. If a path is directed then you can move only in one direction, while in an undirected path you can move in both the directions.

Graph Algorithms

(0,1)
(0,2)
(0,3)
(1,4)
(2,5)
(3,6)
(4,7)
(5,7)
(6,7)
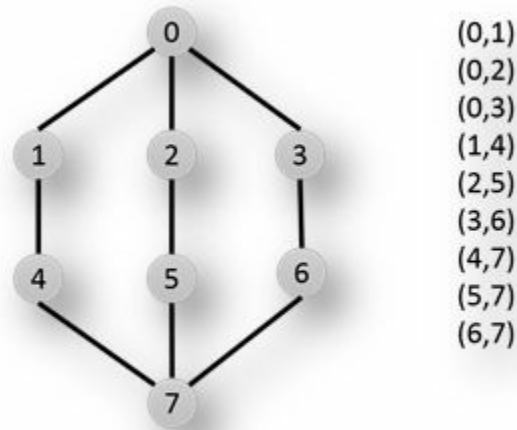
Depth-First Search (DFS)

In the DFS algorithm we start from starting point and go into depth of graph until we reach a dead end and then move up to parent node (Backtrack). In DFS, we use stack to get the next vertex to start a search. Alternatively, we can use recursion (system stack) to do the same.

(0,1)
(0,2)
(0,3)
(1,4)
(2,5)
(3,6)
(4,7)
(5,7)
(6,7)
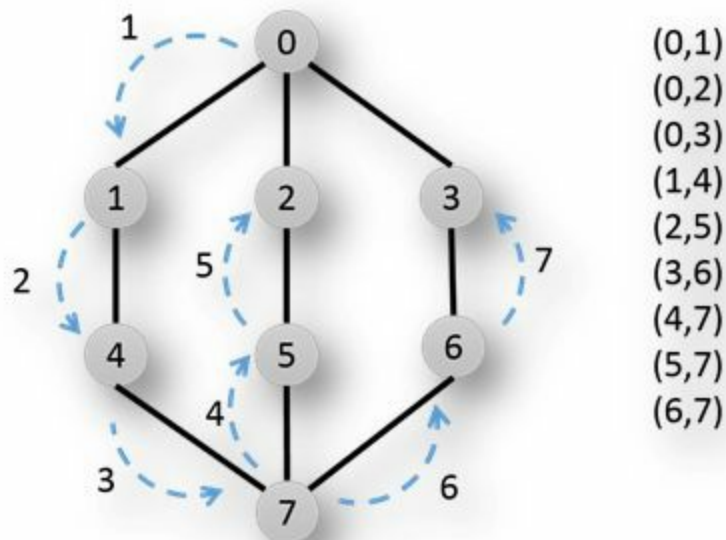
Depth First Traversal
0, 1, 4, 7 , 5, 2, 6, 3

Breadth-First Search (BFS)

In BFS algorithm, a graph is traversed in layer-by-layer fashion. The graph is traversed closer to the starting point. The queue is used to implement BFS.

```
        1  ---  0  --- 3           (0,1)
           ⌄              ⌄         (0,2)
         2                          (0,3)
       1      2      3              (1,4)
    4     5         6               (2,5)
           ⌄    ⌄         ⌄         (3,6)
     4     5    6                   (4,7)
                                    (5,7)
    7  ---  7                       (6,7)
```
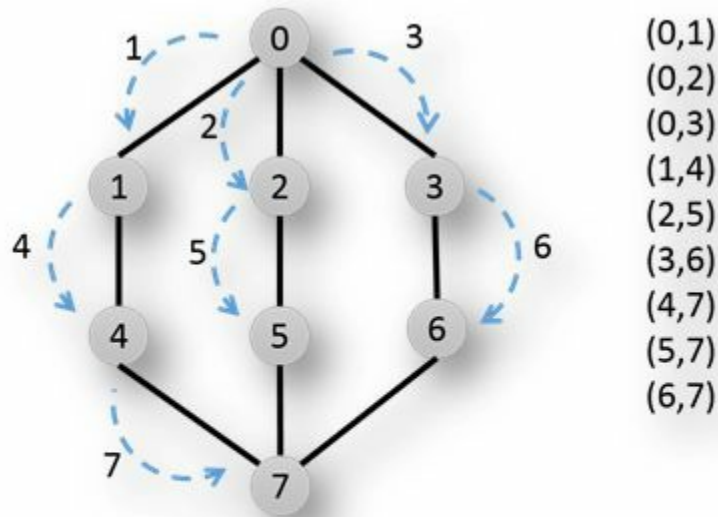
Breadth First Traversal
0, 1, 2, 3, 4, 5, 6, 7

Sorting Algorithms

| 5 | 8 | 9 | 10 | 1 | 6 | 7 | 2 | 3 | 4 |

Sorting

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
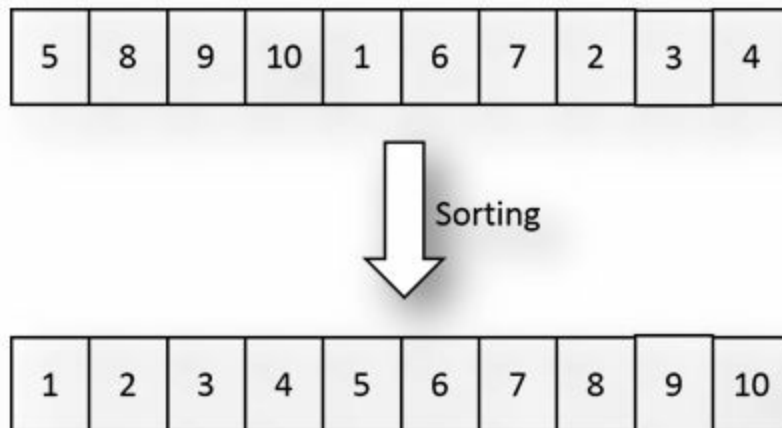
Sorting is the process of placing elements from a collection into ascending or descending order. Sorting arranges data elements in order so that searching become easier.

There are good sorting functions available which does sorting in **O(nlogn)** time, so in this book when we need sorting we will use sort() function and will assume that the sorting is done in **O(nlogn)** time.

Counting Sort

Counting sort is the simplest and most efficient type of sorting. Counting sort has a strict requirement of a predefined range of data.

Sort how many people are there in which age group. We know that the age of people can vary between 1 and 130.

| 2 | 6 | 4 | 1 | 5 | 8 | 1 | 4 | 6 | 1 | Input Data |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |
| 0 | 3 | 1 | 0 | 2 | 1 | 2 | 0 | 1 | 0 | Counts |

| 1 | 1 | 1 | 2 | 4 | 4 | 5 | 6 | 6 | 8 | Sorted Output |
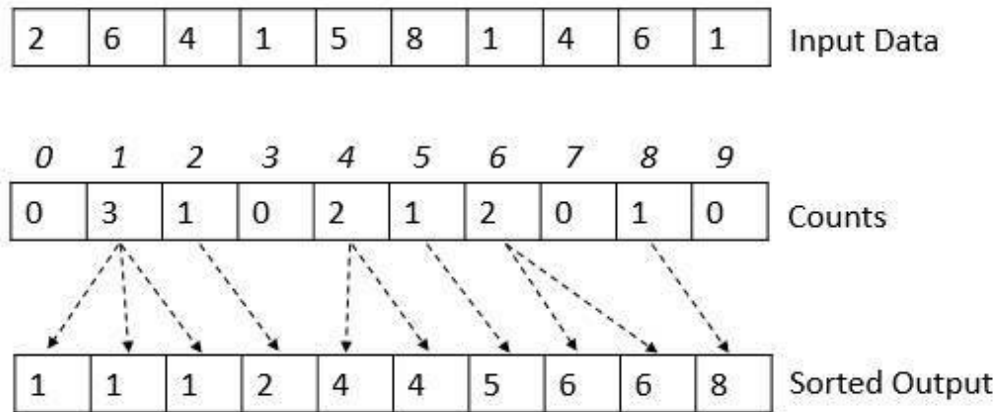
If we know the range of input, then sorting can be done using counting in $O(n+k)$. Where n is the number of people and k is the max age possible, let us suppose 130.

End note

This chapter has provided a brief introduction of the various data structures, algorithms and their complexities. In the next chapters we will look into all these data structure in details. If you know the interface of the various data structures, then you can use them while solving other problems without knowing the internal details and how they are implemented.

CHAPTER 4: SEARCHING

Introduction

Searching is the process of finding a particular item in a collection of items. The item may be a keyword in a file, a record in a database, a node in a tree or a value in an array etc.

Why Searching?

Imagine you are in a library with millions of books. You want to get a specific book with specific title. How will you find it? You will search the book in the section of library, which contains the books whose name starts with the initial letter of the desired book. Then you continue matching with a whole book title until you find your book. (By doing this small heuristic method you have reduced the search space by a factor of 26, consider we have an equal number of books whose title begin with particular char.)

Similarly, computer stores lots of information and to retrieve this information efficiently, we need very efficient searching algorithms. To make searching efficient, we keep the data in some proper order. There are certain ways of organizing the data. If you keep the data in proper order, it is easy to search required element. For example, Sorting is one of the process for making data organized.

Different Searching Algorithms

· Linear Search – Unsorted Input
· Linear Search – Sorted Input
· Binary Search (Sorted Input)
· String Search: Tries, Suffix Trees, Ternary Search.
· Hashing and Symbol Tables

Linear Search – Unsorted Input

When elements of an array are not ordered or sorted and we want to search for a particular value, we need to scan the full array until we find the desired value. This kind of algorithm is known as unordered linear search. The major problem with this algorithm is less performance or high Time Complexity in worst case.

Example 4.1
```
def linearSearchUnsorted(arr, value)
    i = 0
    size = arr.size
    while i < size
        if value == arr[i]
            return true
```

```
      end
      i += 1
   end
   return false
end
```

Time Complexity: O(n). As we need to traverse the complete array in worst case. Worst case is when your desired element is at the last position of the array. Here, 'n' is the size of the array.
Space Complexity: O(1). No extra memory is used to allocate the array.

Linear Search – Sorted

If elements of the array are sorted either in increasing order or in decreasing order, searching for a desired element will be much more efficient than unordered linear search. In many cases, we do not need to traverse the complete array. Following example explains when you encounter a greater value element from the increasing sorted array, you stop searching further. This is how this algorithm saves the time and improves the performance.

Example 4.2
```
def linearSearchSorted(arr, value)
   i = 0
   size = arr.size
   while i < size
      if value == arr[i]
         return true
      elsif value < arr[i]
         return false
      end
      i += 1
   end
   return false
end
```

Time Complexity: **O(n)**. As we need to traverse the complete array in worst case. Worst case is when your desired element is at the last position of the sorted array. However, in the average case this algorithm is more efficient even though the growth rate is same as unsorted.
Space Complexity: **O(1)**. No extra memory is used to allocate the array.

Binary Search

How do we search a word in a dictionary? In general, we go to some approximate page (mostly middle) and start searching from that point. If we see the word that we are searching is same then we are done with the search. Else, if we see the page is before the selected pages, then apply the same procedure for the first half otherwise to the second half. Binary Search also works in the same way. We get to the middle point from the sorted array and start comparing with the desired value.

Note: Binary search requires the array to be sorted otherwise binary search cannot be applied.

Example 4.3

```
# Binary Search Algorithm : Iterative Way
def Binarysearch(arr, value)
    low = 0
    high = arr.size - 1
    while low <= high
        mid = low + (high - low) / 2 # To avoid the overflow
        if arr[mid] == value
            return true
        elsif arr[mid] < value
            low = mid + 1
        else
            high = mid - 1
        end
    end
    return false
end
```

Time Complexity: O(logn). We always take half input and throw out the other half. So the recurrence relation for binary search is T(n) = T(n/2) + c. Using master theorem (divide and conquer), we get T(n) = O(logn)
Space Complexity: O(1)

Example 4.4: Binary search implementation using recursion.

```
# Binary Search Algorithm : Recursive Way
def BinarySearchRecursive(arr, value)
    return BinarySearchRecursiveUtil(arr, 0, arr.size - 1, value)
end

def BinarySearchRecursiveUtil(arr, low, high, value)
    if low > high
        return false
    end
    mid = low + (high - low) / 2 # To avoid the overflow
    if arr[mid] == value
        return true
    elsif arr[mid] < value
        return BinarySearchRecursiveUtil(arr, mid + 1, high, value)
    else
        return BinarySearchRecursiveUtil(arr, low, mid - 1, value)
    end
end
```

Time Complexity: O(logn). Space Complexity: O(logn) for system stack in recursion

String Searching Algorithms

Refer String chapter.

Hashing and Symbol Tables

Refer Hash-Table chapter.

How sorting is useful in Selection Algorithm?

Selection problems can be converted into sorting problems. Once the array is sorted, it is easy to find the minimum / maximum (or desired element) from the sorted array. The method 'Sorting and then Selecting' is inefficient for selecting a single element, but it is efficient when many selections need to be made from the array. It is because only one initial expensive sort is needed, followed by many cheap selection operations.

For example, if we want to get the maximum element from an array. After sorting the array, we can simply return the last element from the array. What if we want to get second maximum. Now, we do not have to sort the array again and we can return the second last element from the sorted array. Similarly, we can return the kth maximum element by just one scan of the sorted array.

So, with the above discussion, sorting is used to improve the performance. In general this method requires **O(nlogn)** (for sorting) time. With the initial sorting, we can answer any query in one scan, **O(n)**.

Problems in Searching

Print Duplicates in Array

Given an array of n numbers, print the duplicate elements in the array.

<u>First approach:</u> Exhaustive search or Brute force; for each element in array, find if there is some other element with the same value. This is done using two for loop, first loop to select the element and second loop to find its duplicate entry.

Example 4.5
```
def printRepeating(arr)
    size = arr.size
    print " Repeating elements are "
    i = 0
    while i < size
        j = i + 1
        while j < size
            if arr[i] == arr[j]
                print " " , arr[i]
            end
            j += 1
        end
        i += 1
    end
```

```
end
```

The Time Complexity is O(n^2) and Space Complexity is O(1)

Second approach: Sorting; Sort all the elements in the array and after this in a single scan, we can find the duplicates.

Example 4.6
```ruby
def printRepeating2(arr)
    size = arr.size
    arr = arr.sort()
    print " Repeating elements are "
    i = 1
    while i < size
        if arr[i] == arr[i - 1]
            print " " , arr[i]
        end
        i += 1
    end
end
```

Sorting algorithms take O(n log n) time and single scan take O(n) time.
The Time Complexity of an algorithm is O(n log n) and Space Complexity is O(1)

Third approach: Hash-Table, using Hash-Table, we can keep track of the elements we have already seen and we can find the duplicates in just one scan.

Example 4.7
```ruby
def printRepeating3(arr)
    size = arr.size
    set = Set.new
    print " Repeating elements are "
    i = 0
    while i < size
        if set.include?(arr[i])
            print " " , arr[i]
        else
            set.add(arr[i])
        end
        i += 1
    end
end
```

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Forth approach: Counting; this approach is only possible if we know the range of the input. If we know that, the elements in the array are in the range 0 to n-1. We can reserve an array of

length n and when we see an element, we can increase its count. In just one single scan, we know the duplicates. If we know the range of the elements, then this is the fastest way to find the duplicates.

Example 4.8

```
def printRepeating4(arr,range)
    size = arr.size
    count = Array.new(range + 1, 0)
    print " Repeating elements are "
    i = 0
    while i < size
        if count[arr[i]] == 1
            print " " , arr[i]
        else
            count[arr[i]] = count[arr[i]] + 1
        end
        i += 1
    end
end
```

Counting approach just uses an array so inserting and finding take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count array is also O(n)

Find max, appearing element in an array

In given array of n numbers, find the element, which appears maximum number of times.

First approach: Exhaustive search or Brute force; for each element in array, find how many times this particular value appears in array. Keep track of the maxCount and when some element count is greater than maxCount then update the maxCount. This is done using two for loop, first loop to select the element and second loop to count the occurrence of that element.

The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 4.9

```
def getMaxCount(arr)
    size = arr.size
    max = 0
    count = 0
    maxCount = 0
    i = 0
    while i < size
        j = i + 1
        while j < size
            if arr[i] == arr[j]
                count += 1
            end
            j += 1
```

```
            end
        if count > maxCount
            max = arr[i]
            maxCount = count
        end
        count = 0
        i += 1
    end
    return max
end
```

Second approach: Sorting; Sort all the elements in the array and after this in a single scan, we can find the counts. Sorting algorithms takes O(n.logn) time and single scan takes O(n) time. The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 4.10
```
def getMaxCount2(arr)
    size = arr.size
    max = arr[0]
    maxCount = 1
    curr = arr[0]
    currCount = 1
    arr = arr.sort()
    i = 1
    while i < size
        if arr[i] == arr[i - 1]
            currCount += 1
        else
            currCount = 1
            curr = arr[i]
        end
        if currCount > maxCount
            maxCount = currCount
            max = curr
        end
        i += 1
    end
    return max
end
```

Third approach: Counting, This approach is possible only if we know the range of the input. If we know that, the elements in the array are in the range 0 to n-1. We can reserve an array of length n and when we see an element, we can increase its count. In just one single scan, we know the duplicates. If we know the range of the elements, then this is the fastest way to find the max count.

Counting approach just uses array so to increase count take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count array is also

O(n)

Example 4.11

```
def getMaxCount3(arr, range)
    size = arr.size
    max = arr[0]
    maxCount = 1
    count = Array.new( range+1, 0)
    i = 0
    while i < size
        count[arr[i]] += 1
        if count[arr[i]] > maxCount
            maxCount = count[arr[i]]
            max = arr[i]
        end
        i += 1
    end
    return max
end
```

Majority element in an array

In given an array of n elements. Find the majority element, which appears more than n/2 times. Return 0 in case there is no majority element.

First approach: Exhaustive search or Brute force, for each element in array find how many times this particular value appears in array. Keep track of the maxCount and when some element count is greater than maxCount then update the maxCount. This is done using two for loop, first loop to select the element and second loop to count the occurrence of that element.

Once we have the final maxCount we can see if it is greater than n/2, if it is greater than we have a majority if not we do not have any majority.

The Time Complexity is $O(n^2) + O(1) = O(n^2)$ and Space Complexity is $O(1)$

Example 4.12
```
def getMajority(arr)
    size = arr.size
    max = 0
    count = 1
    maxCount = 0
    i = 0
    while i < size
        j = i + 1
        while j < size
            if arr[i] == arr[j]
                count += 1
            end
```

```
            j += 1
        end
        if count > size / 2
            return arr[i]
        end
        count = 1
        i += 1
    end
    print "no majority found"
    return -1
end
```

Second approach: Sorting, Sort all the elements in the array. If there is a majority then the middle element at the index n/2 must be the majority number. So, just single scan can be used to find its count and see if the majority is there or not.

Sorting algorithms take O(n.logn) time and single scan take O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 4.13
```
def getMajority2(arr)
    size = arr.size
    majIndex = size / 2
    count = 1
    arr = arr.sort()
    candidate = arr[majIndex]
    count = 0
    i = 0
    while i < size
        if arr[i] == candidate
            count += 1
        end
        i += 1
    end
    if count > size / 2
        return arr[majIndex]
    else
        print "no majority found"
        return -1
    end
end
```

Third approach: This is a cancelation approach (Moore's Voting Algorithm), if all the elements stand against the majority and each element is cancelled with one element of majority, if there is majority then majority prevails.
· Set the first element of the array as majority candidate and initialize the count to be 1.
· Start scanning the array.
    o If we get some element whose value is same as a majority candidate, then we increase

the count.

- o If we get an element whose value is different from the majority candidate, then we decrement the count.
- o If count become 0, that means we have a new majority candidate. Make the current candidate as majority candidate and reset count to 1.
- o At the end, we will have the only probable majority candidate.

· Now scan through the array once again to see if that candidate we found above have appeared more than n/2 times.

Counting approach just scans throw array two times. The Time Complexity of the algorithm is O(n) time. Space Complexity for creating count array is also O(1)

Example 4.14

```
def getMajority3(arr)
    size = arr.size
    majIndex = 0
    count = 1
    i = 1
    while i < size
        if arr[majIndex] == arr[i]
            count += 1
        else
            count -= 1
        end
        if count == 0
            majIndex = i
            count = 1
        end
        i += 1
    end
    candidate = arr[majIndex]
    count = 0
    i = 0
    while i < size
        if arr[i] == candidate
            count += 1
        end
        i += 1
    end
    if count > size / 2
        return arr[majIndex]
    else
        print "no majority found"
        return -1
    end
end
```

Find the missing number in an array

In given array of n-1 elements, which are in the range of 1 to n. There are no duplicates in the array. One of the integer is missing. Find the missing element.

First approach: Exhaustive search or Brute force, for each value in the range 1 to n, find if there is some element in array which have the same value. This is done using two for loop, first loop to select value in the range 1 to n and the second loop to find if this element is in the array or not.

The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 4.15
```
def findMissingNumber(arr, range)
    found = 0
    i = 1
    size = arr.size
    while i <= range
        found = 0
        j = 0
        while j < size
            if arr[j] == i
                found = 1
                break
            end
            j += 1
        end
        if found == 0
            return i
        end
        i += 1
    end
    print "missing number not found"
    return -1
end
```

Second approach: Sorting; Sort all the elements in the array and after this in a single scan, we can find the duplicates.

Sorting algorithms takes O(n.logn) time and single scan takes O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Third approach: Hash-Table, using Hash-Table; we can keep track of the elements we have already seen and we can find the missing element in just one scan.

Hash-Table insertion and finding take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Forth approach: Counting; we know the range of the input so counting will work. As we know that, the elements in the array are in the range 0 to n-1. We can reserve an array of length n and

when we see an element, we can increase its count. In just one single scan, we know the missing element.

Counting approach just uses an array so insertion and finding take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count array is also O(n)

Fifth approach: You are allowed to modify the given input array. Modify the given input array in such a way that in the next scan you can find the missing element.

When you scan through the array. When at index "index", the value stored in the array will be arr[index] so add the number "n + 1" to arr[arr[index]]. Always read the value from the array using a reminder operator "%". When you scan the array for the first time and modify all the values, then in one single scan you can see if there is some value in the array which is smaller than "n+1" that index is the missing number.
In this approach, the array is scanned two times and the Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Sixth approach: Summation formula to find the sum of n numbers from 1 to n. Subtract the values stored in the array and you will have your missing number.
The Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Seventh approach: XOR approach to find the sum of n numbers from 1 to n. XOR the values stored in the array and you will have your missing number.
The Time Complexity of this algorithm is O(n). Space Complexity is O(1)

Example 4.16

```
def findMissingNumber2(arr, range)
    size = arr.size
    xorSum = 0
    #get the XOR of all the numbers from 1 to range
    i = 1
    while i <= range
        xorSum ^= i
        i += 1
    end
    #loop through the array and get the XOR of elements
    i = 0
    while i < size
        xorSum ^= arr[i]
        i += 1
    end
    return xorSum
end
```

Note: Same problem can be asked in many forms (sometime you have to or don't have to do the xor of the range):
  1. There are numbers in the range of 1-n out of which all appears single time but there is one

that appear two times.
2.   All the elements in the range 1-n are appearing 16 times and one element appears 17 times. Find the element that appears 17 times.

Find Pair in an array

Given an array of n numbers, find two elements such that their sum is equal to "value"

First approach: Exhaustive search or Brute force, for each element in array find if there is some other element, which sums up to the desired value. This is done using two for loop, first loop is to select the element and second loop is to find another element.

The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 4.17

```
def FindPair(arr, value)
    size = arr.size
    i = 0
    while i < size
        j = i + 1
        while j < size
            if (arr[i] + arr[j]) == value
                puts "The pair is : #{ arr[i]}, #{arr[j]}"
                return 1
            end
            j += 1
        end
        i += 1
    end
    return 0
end
```

Second approach: Sorting, Steps are as follows:
1. Sort all the elements in the array.
2. Take two variable first and second. Variable first= 0 and second = size -1
3. Compute sum = arr[first]+arr[second]
4. If the sum is equal to the desired value, then we have the solution
5. If the sum is less than the desired value, then we will increase the first
6. If the sum is greater than the desired value, then we will decrease the second
7. We repeat the above process until we get the desired pair or we get first >= second

Sorting algorithms takes O(n.logn) time and single scan takes O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 4.18

```
def FindPair2(arr, value)
    size = arr.size
    first = 0
    second = size - 1
```

```
        arr = arr.sort()
        while first < second
            curr = arr[first] + arr[second]
            if curr == value
                puts "The pair is : #{ arr[first]}, #{arr[second]}"
                return 1
            elsif curr < value
                first += 1
            else
                second -= 1
            end
        end
        return 0
end
```

Third approach: Hash-Table, using Hash-Table; we can keep track of the elements we have already seen and we can find the pair in just one scan.

1. For each element, insert the value in Hashtable. Let's say current value is arr[index]
2. If value - arr[index] is in the Hashtable then we have the desired pair.
3. Else, proceed to the next entry in the array.

Hash-Table insertion and finding take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Example 4.19
```
def FindPair3(arr, value)
    size = arr.size
    set = Set.new
    i = 0
    while i < size
        if set.include?(value - arr[i])
            puts "The pair is : #{ arr[i]}, #{value - arr[i]}"
            return 1
        end
        set.add(arr[i])
        i += 1
    end
    return 0
end
```

Forth approach: Counting; this approach is only possible if we know the range of the input. If we know that, the elements in the array are in the range 0 to n-1. We can reserve an array of length n and when we see an element, we can increase its count. In place of the Hashtable in the above approach, we will use this array and will find out the pair.

Counting approach just uses an array so insertion and finding take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity for creating count array is also O(n)

Find the Pair in two Arrays

Given two array X and Y. Find a pair of elements (x_i, y_i) such that $x_i \in X$ and $y_i \in Y$ where $x_i + y_i = \text{value}$.

First approach: Exhaustive search or Brute force; loop through element xi of X and see if you can find (value – xi) in Y. Two for loop.
The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Second approach: Sorting; Sort all the elements in the second array Y. For each element of X you can see if that element is there in Y by using binary search.

Sorting algorithms take $O(m.logm)$ and searching will take $O(n.logm)$ time.
The Time Complexity of an algorithm is $O(n.logm)$ or $O(m.logm)$ and Space Complexity is $O(1)$

Third approach: Sorting, Steps are as follows:
1. Sort the elements of both X and Y in increasing order.
2. Take the sum of the smallest element of X and the largest element of Y.
3. If the sum is equal to the value, we got our pair.
4. If the sum is smaller than value, take next element of X
5. If the sum is greater than value, take the previous element of Y

Sorting algorithms take $O(n.logn) + O(m.logm)$ for sorting and searching will take $O(n+m)$ time.
The Time Complexity of an algorithm is $O(n.logn)$ Space Complexity is $O(1)$

Forth approach: Hash-Table, Steps are as follows:
1. Scan through all the elements in the array Y and insert them into Hashtable.
2. Now scan through all the elements of array X, let us suppose the current element is xi see if you can find (value - xi) in the Hashtable.
3. If you find the value, you got your pair.
4. If not, then go to the next value in the array X.

Hash-Table insertion and finding take constant time $O(1)$ so the total Time Complexity of the algorithm is $O(n)$ $O(n)$ time. Space Complexity is also $O(n)$

Fifth approach: Counting; this approach is only possible if we know the range of the input. Same as Hashtable implementation just use a simple array in place of Hashtable and you are done.

Counting approach just uses an array so insertion and finding take constant time $O(1)$ so the total Time Complexity of the algorithm is $O(n)$ time. Space Complexity for creating count array is also $O(n)$

Two elements whose sum is closest to zero

In given Array of integers, both +ve and -ve. You need to find the two elements such that their

sum is closest to zero.

First approach: Exhaustive search or Brute force; for each element in the array find the other element whose value when added will give minimum absolute value. This is done using two for loop, first loop to select the element and second loop to find the element that should be added to it so that the absolute of the sum will be minimum or close to zero.

The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Example 4.20

```
def minAbsSumPair(arr)
    size = arr.size
    # Array should have at least two elements
    if size < 2
        print "Invalid Input"
        return
    end
    # Initialization of values
    minFirst = 0
    minSecond = 1
    minSum = (arr[0] + arr[1]).abs
    l = 0
    while l < size - 1
        r = l + 1
        while r < size
            sum = (arr[l] + arr[r]).abs
            if sum < minSum
                minSum = sum
                minFirst = l
                minSecond = r
            end
            r += 1
        end
        l += 1
    end
    puts "The two elements with minimum sum are : #{arr[minFirst]},#{arr[minSecond]}"
end
```

Second approach: Sorting

Steps are as follows:
1. Sort all the elements in the array.
2. Take two variable firstIndex = 0 and secondIndex = size -1
3. Compute sum = arr[firstIndex]+arr[secondIndex]
4. If the sum is equal to 0 then we have the solution
5. If the sum is less than 0 then we will increase first
6. If the sum is greater than 0 then we will decrease the second
7. We repeat the above process 3 to 6, until we get the desired pair or we get first >= second

Example 4.21

```
def minAbsSumPair2(arr)
    size = arr.size
    # Array should have at least two elements
    if size < 2
        print "Invalid Input"
        return
    end
    arr = arr.sort()
    # Initialization of values
    minFirst = 0
    minSecond = size - 1
    minSum = (arr[minFirst] + arr[minSecond]).abs
    l = 0
    r = size - 1
    while l < r
        sum = (arr[l] + arr[r])
        if (sum).abs < minSum
            minSum = sum
            minFirst = l
            minSecond = r
        end
        if sum < 0
            l += 1
        elsif sum > 0
            r -= 1
        else
            break
        end
    end
    puts "The two elements with minimum sum are : #{arr[minFirst]},#{arr[minSecond]}"
end
```

Find maxima in a bitonic array

A bitonic array comprises of an increasing sequence of integers immediately followed by a decreasing sequence of integers.

Since the elements are sorted in some order, we should go for algorithm similar to binary search. The steps are as follows:

1. Take two variable for storing start and end index. Variable start=0 and end=size-1
2. Find the middle element of the array.
3. See if the middle element is the maxima. If yes, return the middle element.
4. Alternatively, if the middle element is in increasing part, then we need to look for in mid+1 and end.
5. Alternatively, if the middle element is in the decreasing part, then we need to look in the start and mid-1.

6. Repeat step 2 to 5 until we get the maxima.

Example 4.22
```
def SearchBotinicArrayMax(arr)
    size = arr.size
    start = 0
    end2 = size - 1
    mid = (start + end2) / 2
    maximaFound = 0
    if size < 3
        print "error"
        return 0
    end
    while start <= end2
        mid = (start + end2) / 2
        if arr[mid - 1] < arr[mid] and arr[mid + 1] < arr[mid]    #maxima
            maximaFound = 1
            break
        elsif arr[mid - 1] < arr[mid] and arr[mid] < arr[mid + 1]    #increasing
            start = mid + 1
        elsif arr[mid - 1] > arr[mid] and arr[mid] > arr[mid + 1]    #decreasing
            end2 = mid - 1
        else
            break
        end
    end
    if maximaFound == 0
        print "error"
        return 0
    end
    return arr[mid]
end
```

Search element in a bitonic array

A bitonic array comprises of an increasing sequence of integers immediately followed by a decreasing sequence of integers. To search an element in a bitonic array:

1. Find the index or maximum element in the array. By finding the end of increasing part of the array, using modified binary search.
2. Once we have the maximum element, search the given value in increasing part of the array using binary search.
3. If the value is not found in increasing part, search the same value in decreasing part of the array using binary search.

Example 4.23
```
def SearchBitonicArray(arr, key)
    size = arr.size
    max = FindMaxBitonicArray(arr, size)
```

```
        #puts max
    k = BinarySearch(arr, 0, max, key, true)
    if k != -1
        return k
    else
        return BinarySearch(arr, max + 1, size - 1, key, false)
    end
end

def FindMaxBitonicArray(arr, size)
    start = 0
    end2 = size - 1
    if size < 3
        print "error"
        return 0
    end
    while start <= end2
        mid = (start + end2) / 2
        if arr[mid - 1] < arr[mid] and arr[mid + 1] < arr[mid]     #maxima
            return mid
        elsif arr[mid - 1] < arr[mid] and arr[mid] < arr[mid + 1]     #increasing
            start = mid + 1
        elsif arr[mid - 1] > arr[mid] and arr[mid] > arr[mid + 1]     #decreasing
            end2 = mid - 1
        else
            break
        end
    end
    print "error"
    return 0
end

def BinarySearch(arr, start, end2, key, isInc)
    if end2 < start
        return -1
    end
    mid = (start + end2) / 2
    #puts "mid is #{mid}"
    if key == arr[mid]
        return mid
    end
    if (isInc == true and key < arr[mid] )or (isInc == false and key > arr[mid])
        return BinarySearch(arr, start, mid - 1, key, isInc)
    else
        return BinarySearch(arr, mid + 1, end2, key, isInc)
    end
end
```

Occurrence counts in sorted Array

Given a sorted array arr[] find the number of occurrences of a number.

First approach: Brute force, Traverse the array and in linear time we will get the occurrence count of the number. This is done using one loop.
The Time Complexity is O(n) and Space Complexity is O(1)

Example 4.24
```
def findKeyCount(arr, key)
    size = arr.size
    count = 0
    i = 0
    while i < size
        if arr[i] == key
            count += 1
        end
        i += 1
    end
    return count
end
```

Second approach: Since we have sorted array, we should think about some binary search.
1. First, we should find the first occurrence of the key.
2. Then we should find the last occurrence of the key.
3. Take the difference of these two values and you will have the solution.

Example 4.25
```
def findFirstIndex(arr, start, end2, key)
    if end2 < start
        return -1
    end
    mid = (start + end2) / 2
    if key == arr[mid] and (mid == start or arr[mid - 1] != key)
        return mid
    end
    if key <= arr[mid]   # <= is us the number.t in sorted array.
        return findFirstIndex(arr, start, mid - 1, key)
    else
        return findFirstIndex(arr, mid + 1, end2, key)
    end
end
```

```
def findLastIndex(arr, start, end2, key)
    if end2 < start
        return -1
    end
    mid = (start + end2) / 2
```

```
      if key == arr[mid] and (mid == end2 or arr[mid + 1] != key)
         return mid
      end
      if key < arr[mid]   # <
         return findLastIndex(arr, start, mid - 1, key)
      else
         return findLastIndex(arr, mid + 1, end2, key)
      end
end


def findKeyCount2(arr, key)
   size = arr.size
   firstIndex = findFirstIndex(arr, 0, size - 1, key)
   lastIndex = findLastIndex(arr, 0, size - 1, key)
   return (lastIndex - firstIndex + 1)
end
```

Separate even and odd numbers in Array

Given an array of even and odd numbers, write a program to separate even numbers from the odd numbers.

First approach: allocate a separate array, then scan through the given array, and fill even numbers from the start and odd numbers from the end.

Second approach: Algorithm is as follows.
1. Initialize the two variable left and right. Variable left=0 and right= size-1.
2. Keep increasing the left index until the element at that index is even.
3. Keep decreasing the right index until the element at that index is odd.
4. Swap the number at left and right index.
5. Repeat steps 2 to 4 until left is less than right.

Example 4.26
```
def seperateEvenAndOdd(arr)
   size = arr.size
   left = 0
   right = size - 1
   while left < right
      if arr[left] % 2 == 0
         left += 1
      elsif arr[right] % 2 == 1
         right -= 1
      else
         swap(arr, left, right)
         left += 1
         right -= 1
      end
   end
```

Stock purchase-sell problem

In given array, in which nth element is the price of the stock on nth day. You are asked to buy once and sell once, on what date you will be buying and at what date you will be selling to get maximum profit.

<div align="center">Or</div>

In given array of numbers, you need to maximize the difference between two numbers, such that you can subtract the number, which appears before form the number that appear after it.

First approach: Brute force; for each element in the array find if there is some other element whose difference is maximum. This is done using two for loop, first loop to select, buy date index and the second loop to find its selling date entry.

The Time Complexity is $O(n^2)$ and Space Complexity is $O(1)$

Second approach: Another clever solution is to keep track of the smallest value seen so far from the start. At each point, we can find the difference and keep track of the maximum profit. This is a linear solution.

The Time Complexity of the algorithm is $O(n)$ time. Space Complexity for creating count array is also $O(1)$

Example 4.27

```ruby
def maxProfit(stocks)
    size = stocks.size
    buy = 0
    sell = 0
    curMin = 0
    currProfit = 0
    maxProfit = 0
    i = 0
    while i < size
        if stocks[i] < stocks[curMin]
            curMin = i
        end
        currProfit = stocks[i] - stocks[curMin]
        if currProfit > maxProfit
            buy = curMin
            sell = i
            maxProfit = currProfit
        end
        i += 1
    end
    puts "Purchase day is- , #{buy} , at price , #{stocks[buy]}"
    puts "Sell day is- , #{sell} , at price , #{stocks[sell]}"
    return ( stocks[sell] - stocks[buy])
end
```

Find a median of an array

In given array of numbers of size n, if all the elements of the array are sorted then find the element, which lie at the index n/2.

First approach: Sort the array and return the element in the middle.
Sorting algorithms takes O(n.logn).
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 4.28
```
def getMedian(arr)
    size = arr.size
    arr = arr.sort()
    return arr[size / 2]
end
```

Second approach: Use QuickSelect algorithm. This algorithm we will look in the next chapter. In QuickSort algorithm just skip the recursive call that we do not need.

The average Time Complexity of this algorithm will be O(1)

Find median of two sorted Arrays.

First approach: Keep track of the index of both the array, say the index are i and j. keep increasing the index of the array which ever have a smaller value. Use a counter to keep track of the elements that we have already traced.

The Time Complexity of an algorithm is O(n) and Space Complexity is O(1)

Example 4.29
```
def findMedian(arrFirst, arrSecond)
    sizeFirst = arrFirst.size
    sizeSecond = arrSecond.size
        medianIndex = ((sizeFirst + sizeSecond) + (sizeFirst + sizeSecond) % 2) / 2 #cealing
function.
    i = 0
    j = 0
    count = 0
    while count < medianIndex - 1
        if i < sizeFirst - 1 and arrFirst[i] < arrSecond[j]
            i += 1
        else
            j += 1
        end
        count += 1
    end
    if arrFirst[i] < arrSecond[j]
```

```
        return arrFirst[i]
    else
        return arrSecond[j]
    end
end
```

Search 01 Array

In given array of 0's and 1's. All the 0's come before 1's. Write an algorithm to find the index of the first 1.

<div align="center">Or</div>

You are given an array which contains either 0 or 1, and they are in sorted order Ex. a [] = {1, 1, 1, 1, 0, 0, 0} How will you count no of 1`s and 0's?

<u>First approach</u>: Binary Search, since the array is sorted using binary search to find the desired index.
The Time Complexity of an algorithm is O(logn) and Space Complexity is O(1)

Example 4.30
```
def BinarySearch01(arr)
    size = arr.size
    if size == 1 and arr[0] == 1
        return 0
    end
    return BinarySearch01Util(arr, 0, size - 1)
end

def BinarySearch01Util(arr, start, end2)
    if end2 < start
        return -1
    end
    mid = (start + end2) / 2
    if 1 == arr[mid] and 0 == arr[mid - 1]
        return mid
    end
    if 0 == arr[mid]
        return BinarySearch01Util(arr, mid + 1, end2)
    else
        return BinarySearch01Util(arr, start, mid - 1)
    end
end
```

Search in sorted rotated Array

Given a sorted array s of n integer. s is rotated an unknown number of times. Find an element in the array.

<u>First approach</u>: Since the array is sorted, we can use modified binary search to find the element.

The Time Complexity of an algorithm is O(logn) and Space Complexity is O(1)

Example 4.31
```
def BinarySearchRotateArrayUtil(arr, start, end2, key)
    if end2 < start
        return -1
    end
    mid = (start + end2) / 2
    if key == arr[mid]
        return mid
    end
    if arr[mid] > arr[start]
        if arr[start] <= key and key < arr[mid]
            return BinarySearchRotateArrayUtil(arr, start, mid - 1, key)
        else
            return BinarySearchRotateArrayUtil(arr, mid + 1, end2, key)
        end
    else
        if arr[mid] < key and key <= arr[end2]
            return BinarySearchRotateArrayUtil(arr, mid + 1, end2, key)
        else
            return BinarySearchRotateArrayUtil(arr, start, mid - 1, key)
        end
    end
end

def BinarySearchRotateArray(arr, key)
    size = arr.size
    return BinarySearchRotateArrayUtil(arr, 0, size - 1, key)
end
```

First Repeated element in the array

In given unsorted array of n elements, find the first element, which is repeated.

First approach: Exhaustive search or Brute force; for each element in array find if there is some other element with the same value. This is done using two for loop, first loop to select the element and second loop to find its duplicate entry.

The Time Complexity is $O(n^2)$ and Space Complexity is O(1)

Example 4.32
```
def FirstRepeated(arr)
    size = arr.size
    i = 0
    while i < size
        j = i + 1
        while j < size
```

```
            if arr[i] == arr[j]
                return arr[i]
            end
            j += 1
        end
        i += 1
    end
    return 0
end
```

Second approach: Hash-Table; using Hash-Table, we can keep track number of times a particular element came in the array. First scan just populate the Hashtable. In the second, scan just look the occurrence of the elements in the Hashtable. If occurrence is more for some element, then we have our solution and the first repeated element.

Hash-Table insertion and finding take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n) for maintaining hash.

Transform Array

How would you swap elements of an array like [a1 a2 a3 a4 b1 b2 b3 b4] to convert it into [a1 b1 a2 b2 a3 b3 a4 b4]?
Approach:
· First swap elements in the middle pair
· Next swap elements in the middle two pairs
· Next swap elements in the middle three pairs
· Iterate n-1 steps.

Example 4.33
```
def transformArrayAB1(arr)
    size = arr.size
    n = size / 2
    i = 1
    while i < n
        j = 0
        while j < i
            swap(arr, n - i + 2 * j, n - i + 2 * j + 1)
            j += 1
        end
        i += 1
    end
end
```

Find 2nd largest number in an array with minimum comparisons

Suppose you are given an unsorted array of n distinct elements. How will you identify the second largest element with minimum number of comparisons?

First approach: Find the largest element in the array. Then replace the last element with the largest element. Then search the second largest element in the remaining n-1 elements.
The total number of comparisons is: (n-1) + (n-2)

Second approach: Sort the array and then give the (n-1) element. This approach is still more inefficient.

Third approach: Using priority queue / Heap; In this approach we will look into heap chapter. Use buildHeap() function to build heap from the array. This is done in n comparisons. Arr[0] is the largest number, and the greater among arr[1] and arr[2] is the second largest.
The total number of comparisons are: (n-1) + 1 = n

Check if two Arrays are permutation of each other

In given two integer Arrays; You have to check whether they are permutation of each other.

First approach: Sorting; Sort all the elements of both the Arrays and Compare each element of both the Arrays from beginning to end. If there is no mismatch, return true. Otherwise, false.
Sorting algorithms takes O(n.logn) time and comparison takes O(n) time.
The Time Complexity of an algorithm is O(n.logn) and Space Complexity is O(1)

Example 4.34
```
def checkPermutation(array1, array2)
    size1 = array1.size
    size2 = array2.size
    if size1 != size2
        return false
    end
    array1 = array1.sort()
    array2 = array2.sort()
    i = 0
    while i < size1
        if array1[i] != array2[i]
            return false
        end
        i += 1
    end
    return true
end
```

Second approach: Hash-Table (Assumption: No duplicates).
1. Create a Hash-Table for all the elements of the first array.
2. Traverse the other array from beginning to the end and search for each element in the Hash-Table.
3. If all the elements are found in, the Hash-Table return true, otherwise return false.

Hash-Table insert and find take constant time O(1) so the total Time Complexity of the algorithm is O(n) time. Space Complexity is also O(n)

Time Complexity = O(n) (For creation of Hash-Table and look-up),
Space Complexity = O(n) (For creation of Hash-Table).

Example 4.35

```
def checkPermutation2(array1, array2, range)
   size1 = array1.size
   size2 = array2.size
   if size1 != size2
       return false
   end
   count = Array.new(range + 1, 0)
   i = 0
   while i < size1
       count[array1[i]] += 1
       i += 1
   end
   i = 0
   while i < size2
       if count[array2[i]] == 0
           return false
       end
       count[array2[i]] -= 1
       i += 1
   end
   return true
end
```

Remove duplicates in an integer array

First approach: Sorting, Steps are as follows:
1. Sort the array.
2. Take two references. A subarray will be created with all unique elements starting from 0 to the first reference (The first reference points to the last index of the subarray). The second reference iterates through the array from 1 to the end.   Unique numbers will be copied from the second reference location to first reference location and the same elements are ignored.

Time Complexity calculation:
Time to sort the array = **O(nlogn)**.   Time to remove duplicates = **O(n).**
Overall Time Complexity = **O(nlogn)**.
No additional space is required so Space Complexity is **O(1)**.

Example 4.36

```
def removeDuplicates(arr)
   size = arr.size
   if size == 0
       return 0
   end
```

```
        j = 0
        arr = arr.sort()
        i = 1
        while i < size
            if arr[i] != arr[j]
                j += 1
                arr[j] = arr[i]
            end
            i += 1
        end
        puts j
        return arr[0..j+1]
end
```

Searching for an element in a 2-d sorted array

In given 2 dimensional array. Each row and column are sorted in ascending order. How would you find an element in it?

The algorithm works as:
1. Start with element at last column and first row
2. If the element is the value we are looking for, return true.
3. If the element is greater than the value we are looking for, go to the element at previous column but same row.
4. If the element is less than the value we are looking for, go to the element at next row but same column.
5. Return false, if the element is not found after reaching the element of the last row of the first column. Condition (row < r && column >= 0) is false.

Example 4.37
```
def FindElementIn2DArray(arr, r, c, value)
    row = 0
    column = c - 1
    while row < r and column >= 0
        if arr[row][column] == value
            return 1
        elsif arr[row][column] > value
            column -= 1
        else
            row += 1
        end
    end
    return 0
end
```

Running time = **O(N)**.

Exercise

1. In given array of n elements, find the first repeated element. Which of the following methods will work for us. And which if the method will not work for us. If a method works, then implement it.
   - Brute force exhaustive search.
   - Use Hash-Table to keep an index of the elements and use the second scan to find the element.
   - Sorting the elements.
   - If we know the range of the element then we can use counting technique.

   Hint: When order in which elements appear in input is important, we cannot use sorting.

2. In given array of n elements, write an algorithm to find three elements in an array whose sum is a given value.
   Hint: Try to do this problem using a brute force approach. Then try to apply the sorting approach along with a brute force approach.   The Time Complexity will be $O(n^2)$

3. In given array of –ve and +ve numbers, write a program to separate –ve numbers from the +ve numbers.

4. In given array of 1's and 0's, write a program to separate 0's from 1's.
   Hint: QuickSelect, counting

5. In given array of 0's, 1's and 2's, write a program to separate 0's , 1's and 2's.

6. In given array whose elements is monotonically increasing with both negative and positive numbers. Write an algorithm to find the point at which array becomes positive.

7. In given sorted array, find a given number. If found then return the index if not found then insert into the array.

8. Find max in sorted rotated array.

9. Find min in the sorted rotated array.

10. Find kth Smallest Element in the Union of Two Sorted Arrays

CHAPTER 5: SORTING

Introduction

Sorting is the process of placing elements from an array into ascending or descending order. For example, when we play cards, sort cards according to their value so that we can find the required card easily.

When we go to some library, the books are arranged according to streams (Algorithm, Operating systems, Networking etc.). Sorting arranges data elements in order so that searching become easier. When books are arranged in proper indexing order, then it is easy to find a book we are looking for.

This chapter discusses algorithms for sorting an array of N items. Understanding sorting algorithms are the first step towards understanding algorithm analysis. Many sorting algorithms are developed and analysed.

A sorting algorithm like Bubble-Sort, Insertion-Sort and Selection-Sort are easy to implement and are suitable for the small input set. However, for large dataset they are slow.

A sorting algorithm like Merge-Sort, Quick-Sort and Heap-Sort are some of the algorithms that are suitable for sorting large dataset. However, they are overkill if we want to sort the small dataset.

Some algorithm, which is suitable when we have some range information on input data.

Some other algorithm is there to sort a huge data set that cannot be stored in memory completely, for which external sorting technique is developed.

Before we start a discussion of the various algorithms one by one. First, we should look at comparison function that is used to compare two values.

Less function will return 1 if value1 is less than value2 otherwise it will return 0.
```
def less(value1, value2)
    return value1 < value2
end
```

More function will return 1 if value1 is more than value2 otherwise it will return 0.
```
def more(value1, value2)
    return value1 > value2
end
```

The value in various sorting algorithms is compared using one of the above functions and it will be swapped depending upon the return value of these functions. If more() comparison function is used, then sorted output will be increasing in order and if less() is used then resulting output will be in descending order.

Type of Sorting

Internal Sorting: All the elements can be read into memory at the same time and sorting is performed in memory.

1. Selection-Sort
2. Insertion-Sort
3. Bubble-Sort
4. Quick-Sort
5. Merge-Sort

External Sorting: In this, the dataset is so big that it is impossible to load the whole dataset into memory so sorting is done in chunks.

1. Merge-Sort

Three things to consider in choosing, sorting algorithms for application:

1. Number of elements in array
2. A number of different orders of array required
3. The amount of time required to move the data or not move the data

Bubble-Sort

Bubble-Sort is the slowest algorithm for sorting. It is easy to implement and used when data is small.

In Bubble-Sort, we compare each pair of adjacent values. We want to sort values in increasing order so if the second value is less than the first value then we swap these two values. Otherwise, we will go to the next pair. Thus, largest values bubble to the end of the array.

After the first pass, the largest value will be in the rightmost position. We will have N number of passes to get the array completely sorted.

5	1	2	4	3	7	6	Swap
1	5	2	4	3	7	6	Swap
1	2	5	4	3	7	6	Swap
1	2	4	5	3	7	6	Swap
1	2	4	3	5	7	6	No Swap
1	2	4	3	5	7	6	Swap
1	2	4	3	5	6	7	

Example 5.1

```
def BubbleSort(array)
    size = array.size
    i = 0
    while i < (size - 1)
        j = 0
        while j < size - i - 1
            if more(array[j], array[j + 1])
                # Swapping
                temp = array[j]
                array[j] = array[j + 1]
                array[j + 1] = temp
            end
            j += 1
        end
        i += 1
    end
end

# Testing code
array = [9, 1, 8, 2, 7, 3, 6, 4, 5]
bs = BubbleSort2(array)
print array
```

Analysis:
- The outer for loops represents the number of swaps that are done for comparison of data.
- The inner loop is actually used to do the comparison of data. At the end of each inner loop iteration, the largest value is moved to the end of the array. In the first iteration the largest value, in the second iteration the second largest and so on.
- more() function is used for comparison which means when the value of the first argument is greater than the value of the second argument then perform a swap. By this we are sorting in increasing order if we have, the less() function in place of more() then array will be sorted in decreasing order.

Complexity Analysis:
Each time the inner loop execute for (n-1), (n-2), (n-3)…(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n(n-1)/2

Worst case performance	$O(n^2)$
Average case performance	$O(n^2)$
Space Complexity	O(1) as we need only one temp variable
Stable Sorting	Yes

Modified (improved) Bubble-Sort

When there is no more swap in one pass of the outer loop. It indicates that all the elements are already in order so we should stop sorting. This sorting improvement in Bubble-Sort is extremely useful when we know that, except few elements rest of the array is already sorted.

Example 5.2

```
def BubbleSort2(array)
    size = array.size
    swapped = 1
    i = 0
    while i < (size - 1) and swapped == 1
        swapped = 0
        j = 0
        while j < size - i - 1
            if more(array[j], array[j + 1])
                # Swapping
                temp = array[j]
                array[j] = array[j + 1]
                array[j + 1] = temp
                swapped = 1
            end
            j += 1
        end
        i += 1
    end
end
```

By applying this improvement, best case performance of this algorithm is improved when an array is nearly sorted. In this case we just need one single pass and the best case complexity is **O(n)**

Complexity Analysis:

Worst case performance	$O(n^2)$
Average case performance	$O(n^2)$
Space Complexity	O(1)

Adaptive: When array is nearly sorted	O(n)
Stable Sorting	Yes

Insertion-Sort

Insertion-Sort Time Complexity is **O(n^2)** which is same as Bubble-Sort but perform a bit better than it. It is the way we arrange our playing cards. We keep a sorted subarray. Each value is inserted into its proper position in the sorted sub-array in the left of it.

5	6	2	4	7	3	1	Insert 5
5	6	2	4	7	3	1	Insert 6
2	5	6	4	7	3	1	Insert 2
2	4	5	6	7	3	1	Insert 4
2	4	5	6	7	3	1	Insert 7
2	3	4	5	6	7	1	Insert 3
1	2	3	4	5	6	7	Insert 1

Example 5.3
```
def InsertionSort(arr)
    size = arr.size
    i = 1
    while i < size
        temp = arr[i]
        j = i
        while j > 0 and more(arr[j - 1], temp)
            arr[j] = arr[j - 1]
            j -= 1
        end
        arr[j] = temp
```

```
        i += 1
    end
end

# Testing code
array = [9, 1, 8, 2, 7, 3, 6, 4, 5]
InsertionSort(array)
print array
```

Analysis:
- The outer loop is used to pick the value we want to insert into the sorted array in left.
- The value we want to insert we have picked and saved in a temp variable.
- The inner loop is doing the comparison using the more() function. The values are shifted to the right until we find the proper position of the temp value for which we are doing this iteration.
- Finally, the value is placed into the proper position. In each iteration of the outer loop, the length of the sorted array increase by one. When we exit the outer loop, the whole array is sorted.

Complexity Analysis:

Worst case Time Complexity	$O(n^2)$
Best case Time Complexity	$O(n)$
Average case Time Complexity	$O(n^2)$
Space Complexity	$O(1)$
Stable sorting	Yes

Selection-Sort

Selection-Sort searches, the whole unsorted array and put the largest value at the end of it. This algorithm is having the same Time Complexity, but performs better than both bubble and Insertion-Sort as less number of comparisons required. The sorted array is created backward in Selection-Sort.

5	6	2	4	7	3	1		Swap
5	6	2	4	1	3	7		Swap
5	3	2	4	1	6	7		Swap
1	3	2	4	5	6	7		No Swap
1	3	2	4	5	6	7		Swap
1	2	3	4	5	6	7		No Swap
1	2	3	4	5	6	7		

Example 5.4:

```
def SelectionSort(arr) #back array
    size = arr.size
    i = 0
    while i < size - 1
        max = 0
        j = 1
        while j <= size - 1 - i
            if arr[j] > arr[max]
                max = j
            end
            j += 1
        end
        temp = arr[size - 1 - i]
        arr[size - 1 - i] = arr[max]
        arr[max] = temp
        i += 1
    end
end

# Testing code
array =[9, 1, 8, 2, 7, 3, 6, 4, 5]
SelectionSort2(array)
print array
```

Analysis:
- The outer loop decides the number of times the inner loop will iterate. For an input of N elements, the inner loop will iterate N number of times.
- In each iteration of the inner loop, the largest value is calculated and is placed at the end of the array.
- This is the final replacement of the maximum value to the proper location. The sorted array is created backward.

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	$O(n^2)$
Average case Time Complexity	$O(n^2)$
Space Complexity	$O(1)$
Stable Sorting	No

The same algorithm can be implemented by creating the sorted array in the front of the array.

Example 5.5:

```
def SelectionSort2(arr) #front array
    size = arr.size
    i = 0
    while i < size - 1
        min = i
        j = i + 1
        while j < size
            if arr[j] < arr[min]
                min = j
            end
            j += 1
        end
        temp = arr[i]
        arr[i] = arr[min]
        arr[min] = temp
        i += 1
    end
end
```
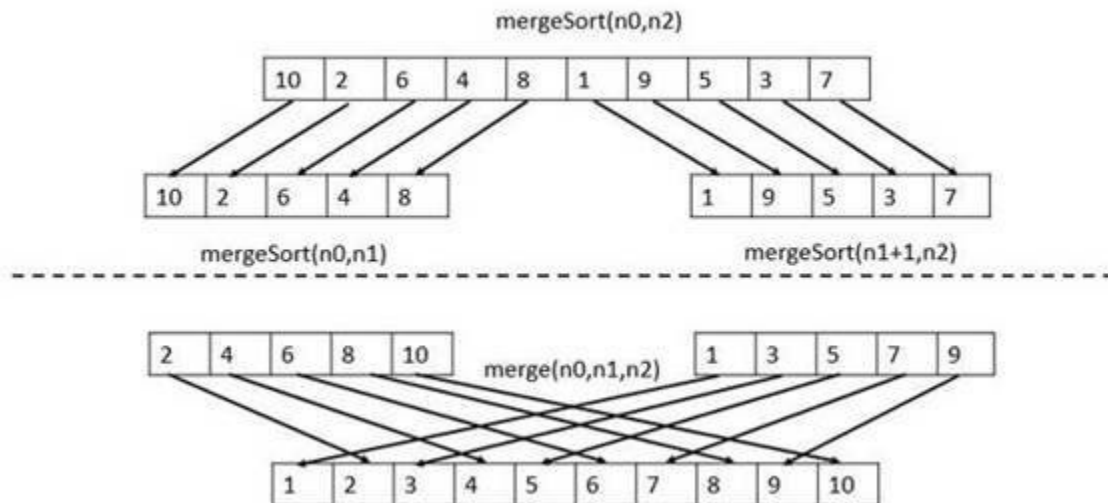
Merge-Sort

Merge sort divide the input into half recursive in each step. It sort the two parts separately recursively and finally combine the result into final sorted output.

mergeSort(n0,n2)

mergeSort(n0,n1) mergeSort(n1+1,n2)

merge(n0,n1,n2)

Example 5.6:

```
def MergeSort(arr)
    size = arr.size
    tempArray = Array.new(size,0)
    mergeSortUtil(arr, tempArray, 0, size - 1)
end

def mergeSortUtil(arr, tempArray, lowerIndex, upperIndex)
    if lowerIndex >= upperIndex
        return
    end
    middleIndex = (lowerIndex + upperIndex) / 2
    mergeSortUtil(arr, tempArray, lowerIndex, middleIndex)
    mergeSortUtil(arr, tempArray, middleIndex + 1, upperIndex)
    merge(arr, tempArray, lowerIndex, middleIndex, upperIndex)
end

def merge(arr, tempArray, lowerIndex, middleIndex, upperIndex)
    lowerStart = lowerIndex
    lowerStop = middleIndex
    upperStart = middleIndex + 1
    upperStop = upperIndex
    count = lowerIndex
    while lowerStart <= lowerStop and upperStart <= upperStop
        if arr[lowerStart] < arr[upperStart]
            tempArray[count] = arr[lowerStart]
            count += 1
            lowerStart += 1
        else
            tempArray[count] = arr[upperStart]
            count += 1
            upperStart += 1
        end
    end
    while lowerStart <= lowerStop
```

```
            tempArray[count] = arr[lowerStart]
            count += 1
            lowerStart += 1
        end
        while upperStart <= upperStop
            tempArray[count] = arr[upperStart]
            count += 1
            upperStart += 1
        end
        i = lowerIndex
        while i <= upperIndex
            arr[i] = tempArray[i]
            i += 1
        end
end

# Testing code
array = [3, 4, 2, 1, 6, 5, 7, 8, 1, 1]
MergeSort(array)
print array
```
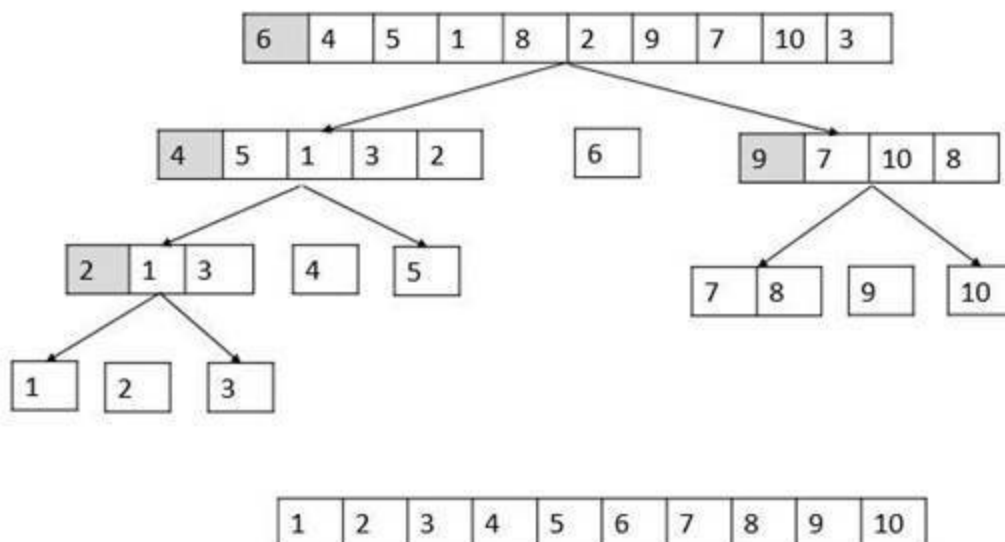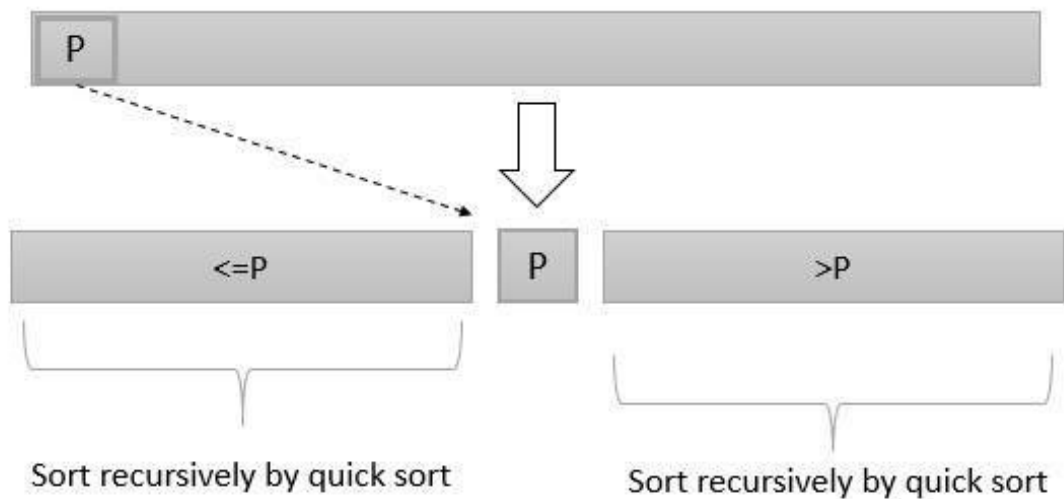
· The Time Complexity of Merge-Sort is **O(nlogn)** in all 3 cases (best, average and worst) as Merge-Sort always divides the array into two halves and takes linear time to merge two halves.
· It requires the equal amount of additional space as the unsorted array. Hence, it is not at all recommended for searching large unsorted arrays.
· It is the best Sorting technique for sorting Linked Arrays.

Complexity Analysis:

Worst Case Time Complexity	O(nlogn)
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(n)
Stable Sorting	Yes

Quick-Sort

P

<=P   P   >P

Sort recursively by quick sort   Sort recursively by quick sort

| 6 | 4 | 5 | 1 | 8 | 2 | 9 | 7 | 10 | 3 |

| 4 | 5 | 1 | 3 | 2 | | 6 | | 9 | 7 | 10 | 8 |

| 2 | 1 | 3 | | 4 | | 5 | | 7 | 8 | | 9 | | 10 |

| 1 | | 2 | | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Quick sort is also a recursive algorithm.
· In each step, we select a pivot (let us say first element of array).
· Then we traverse the rest of the array and copy all the elements of the array which are smaller than the pivot to the left side of array
· We copy all the elements of the array, which are greater than pivot to the right side of the array. Obviously, the pivot is at its sorted position.
· Then we sort the left and right subarray separately.
· When the algorithm returns the whole array is sorted.

Example 5.7:

```
def QuickSort(arr)
    size = arr.size
    quickSortUtil(arr, 0, size - 1)
end

def swap(arr, first, second)
    temp = arr[first]
```

```
      arr[first] = arr[second]
      arr[second] = temp
end

def quickSortUtil(arr, lower, upper)
    if upper <= lower
        return
    end
    pivot = arr[lower]
    start = lower
    stop = upper
    while lower < upper
        while arr[lower] <= pivot and lower < upper
            lower += 1
        end
        while arr[upper] > pivot and lower <= upper
            upper -= 1
        end
        if lower < upper
            swap(arr, upper, lower)
        end
    end
    swap(arr, upper, start) #upper is the pivot position
    quickSortUtil(arr, start, upper - 1) #pivot -1 is the upper for left sub array.
    quickSortUtil(arr, upper + 1, stop)# pivot + 1 is the lower for right sub array.
end

# Testing code
array = [3, 4, 2, 1, 6, 5, 7, 8, 1, 1]
QuickSort(array)
print array
```

· The space required by Quick-Sort is very less, only O(nlogn) additional space is required.
· Quicksort is not a stable sorting technique. It can reorder elements with identical keys.

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(nlogn)
Stable Sorting	No

Quick Select

Quick select algorithm is used to find the element, which will be at the Kth position when the array will be sorted without actually sorting the whole array. Quick select is very similar to

Quick-Sort in place of sorting the whole array we just ignore the one-half of the array at each step of Quick-Sort and just focus on the region of array on which we are interested.

Example 5.8:

```
def QuickSelect(arr, k)
    quickSelectUtil(arr, 0, arr.size - 1, k)
    return arr[k]
end

def quickSelectUtil(arr, lower, upper, k)
    if upper <= lower
        return
    end
    pivot = arr[lower]
    start = lower
    stop = upper
    while lower < upper
        while arr[lower] <= pivot and lower < upper
            lower += 1
        end
        while arr[upper] > pivot and lower <= upper
            upper -= 1
        end
        if lower < upper
            swap(arr, upper, lower)
        end
    end
    swap(arr, upper, start) #upper is the pivot position
    if k < upper
        quickSelectUtil(arr, start, upper - 1, k)
    end #pivot -1 is the upper for left sub array.
    if k > upper
        quickSelectUtil(arr, upper + 1, stop, k)
    end
end

# pivot + 1 is the lower for right sub array.
def swap(arr, first, second)
    temp = arr[first]
    arr[first] = arr[second]
    arr[second] = temp
end

# Testing code
array = [3, 4, 2, 1, 6, 5, 7, 8, 10, 9]
val = QuickSelect(array, 5)
print "value at index 5 is : " , val
```

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	$O(\log n)$
Average Time Complexity	$O(\log n)$
Space Complexity	$O(n \log n)$

Bucket Sort

Bucket sort is the simplest and most efficient type of sorting. Bucket sort has a strict requirement of a predefined range of data.

Like, sort how many people are in which age group. We know that the age of people can vary between 0 and 130.
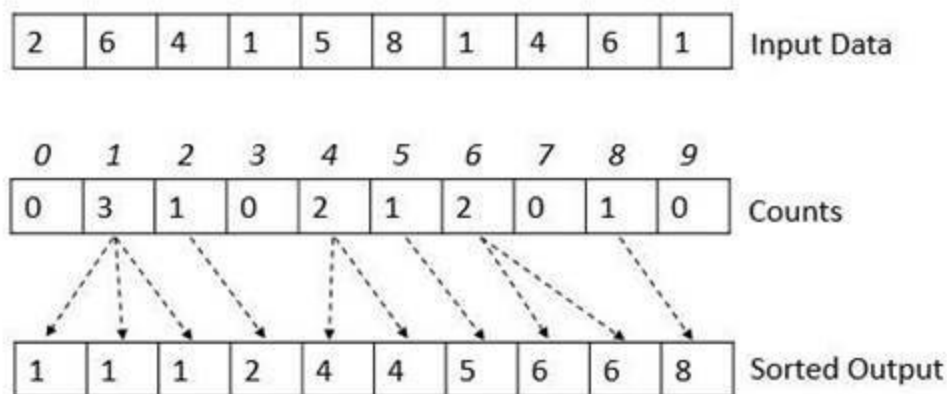
Example 5.9:

```
def BucketSort(array, lowerRange, upperRange)
    range = upperRange - lowerRange
    size = array.size
    count = Array.new(range,0)
    i = 0
    while i < size
        count[array[i] - lowerRange] += 1
        i += 1
    end
    j = 0
    i = 0
    while i < range
        while count[i] > 0
            array[j] = i + lowerRange
            j += 1
            count[i] -= 1
        end
        i += 1
    end
end
```

```
# Testing code
array = [23, 24, 22, 21, 26, 25, 27, 28, 21, 21]
BucketSort(array, 20, 30)
print array
```

Analysis:
- · We have created a count array to store counts.
- · Count array elements are initialized to zero.
- · Index corresponding to input array is incremented.
- · Finally, the information stored in count array is saved in the array.
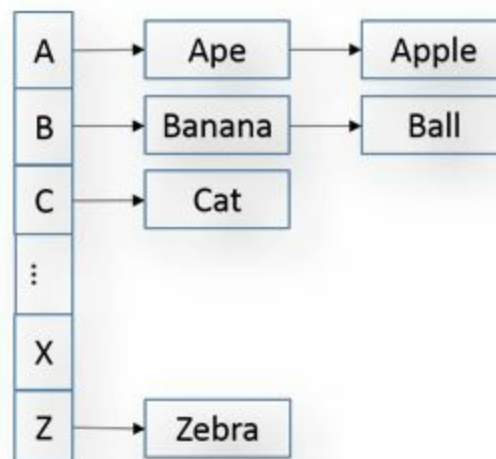
Complexity Analysis:

Data structure	Array
Worst case performance	O(n+k)
Average case performance	O(n+k)
Worst case Space Complexity	O(k)

k - Number of distinct elements.
n - Total number of elements in array.

Generalized Bucket Sort

There are cases when the element falling into a bucket are not unique but are in the same range. When we want to sort an index of a name, we can use the reference bucket to store names.

The buckets are already sorted and the elements inside each bucket can be kept sorted by using an Insertion-Sort algorithm. We are leaving this generalized bucket sort implementation to the reader of this book. The similar data structure will be defined in the coming chapter of Hash-Table using separate chaining.

Heap-Sort

Heap-Sort we will study in the Heap chapter.

Complexity Analysis:

Data structure	Array
Worst case performance	O(nlogn)
Average case performance	O(nlogn)
Worst case Space Complexity	O(1)

Tree Sorting

In-order traversal of the binary search tree can also be seen as a sorting algorithm. We will see this in binary search tree section of tree chapter.

Complexity Analysis:

Worst Case Time Complexity	$O(n^2)$
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(n)
Stable Sorting	Yes

External Sort (External Merge-Sort)

When data needs to be sorted is huge and it is not possible to load it completely in memory (RAM), for such a dataset we use external sorting. Such data is sorted using external Merge-Sort algorithm. First data is picked in chunks and it is sorted in memory. Then this sorted data is written back to disk. Whole data are sorted in chunks using Merge-Sort. Now we need to combine these sorted chunks into final sorted data.

Then we create queues for the data, which will read from the sorted chunks. Each chunk will have its own queue. We will pop from this queue and these queues are responsible for reading from the sorted chunks. Let us suppose we have K different chunks of sorted data each of length M.

The third step is using a Min-Heap, which will take input data from each of this queue. It will take one element from each queue. The minimum value is taken from the Heap and added to the final sorted element output. Then queue from which this min element is inserted in the heap will again popped and one more element from that queue is added to the Heap. Finally, when the data is exhausted from some queue that queue is removed from the input array. Finally, we will get a sorted data coming out from the heap.

We can optimize this process further by adding an output buffer, which will store data coming out of Heap and will do a limited number of the write operation in the final Disk space.
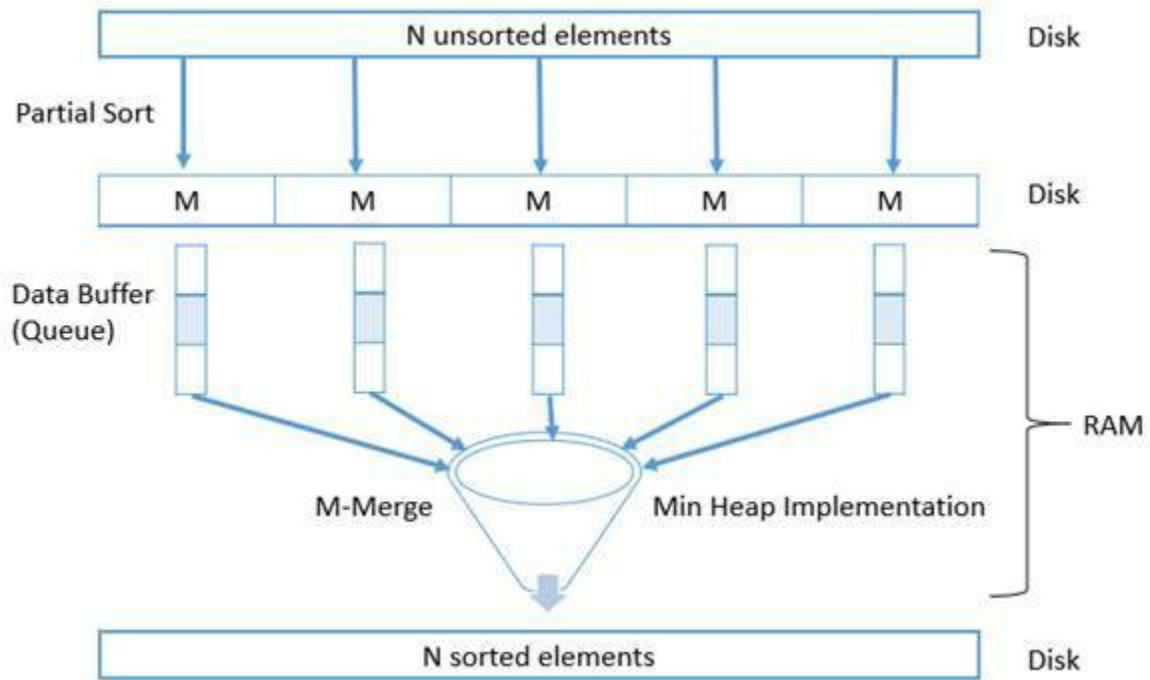
Note: No one will be asking to implement external sorting in an interview, but it is good to know about it.

Comparisons of the various sorting algorithms.

Below is comparison of various sorting algorithms:

Sort	Average Time	Best Time	Worst Time	Space	Stable
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Modified Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	Yes
Heap Sort	$O(n * \log(n))$	$O(n * \log(n))$	$O(n * \log(n))$	$O(1)$	No
Merge Sort	$O(n * \log(n))$	$O(n * \log(n))$	$O(n * \log(n))$	$O(n)$	Yes
Quick Sort	$O(n * \log(n))$	$O(n * \log(n))$	$O(n^2)$	$O(n)$ worst case $O(\log(n))$ average case	No
Bucket Sort	$O(n\ k)$	$O(n\ k)$	$O(n\ k)$	$O(n\ k)$	Yes

Selection of Best Sorting Algorithm

No sorting algorithm is perfect. Each of them has their own pros and cons. Let us read one by one:

Quick-Sort: When you do not need a stable sort and average case performance matters more than worst-case performance. When data is random, we prefer the Quick-Sort. Average case Time Complexity of Quick-Sort is **O(nlogn)** and worst-case Time Complexity is **O(n^2).** Space Complexity of Quick-Sort is O**(logn)** auxiliary storage, which is stack space used in recursion.

Merge-Sort: When you need a stable sort and Time Complexity of **O(nlogn)**, Merge-Sort is used. In general, Merge-Sort is slower than Quick-Sort because of lot of copy happens in the merge phase. There are two uses of Merge-Sort when we want to merge two sorted linked lists and Merge-Sort is used in external sorting.

Heap-Sort: When you do not need a stable sort and you care more about worst-case performance than average case performance. It has guaranteed to be **O(nlogn)**, and uses **O(1)** auxiliary space, means you will not unpredictably run out of memory on very large inputs.

Insertion-Sort: When we need a stable sort, When N is guaranteed to be small, including as the base case of a Quick-Sort or Merge-Sort. Worst-case Time Complexity is **O(n^2)**. It has a very small constant factor multiplied to calculate actual time taken. Therefore, for smaller input size it performs better than Merge-Sort or Quick-Sort. It is also useful when the data is already pre-sorted. In this case, its running time is **O(N)**.

Bubble-Sort: Where we know the data is nearly sorted. Say only two elements are out of place. Then in one pass, Bubble Sort will make the data sorted and in the second pass, it will see everything is sorted and then exit. Only takes 2 passes of the array.

Selection-Sort: Best, Worst & Average Case running time all are **O(n^2)**. It is only useful when you want to do something quick. They can be used when you are just doing some prototyping.

Counting-Sort: When you are sorting data within a limited range.

Radix-Sort: When log(N) is significantly larger than K, where K is the number of radix digits.

Bucket-Sort: When your input is more or less uniformly distributed.

Note: A stable sort is one that has guaranteed not to reorder elements with identical keys.

Exercise

1. In given text file, print the words with their frequency. Now print the kth word in term of frequency.
    Hint:-
    a) First approach may be you can use the sorting and return the kth element.
    b) Second approach: You can use the kth element quick select algorithm.

c) Third approach: You can use Hashtable or Trie to keep track of the frequency. Use Heap to get the Kth element.

2. In given K input streams of number in sorted order. You need to make a single output stream, which contains all the elements of the K streams in sorted order. The input streams support ReadNumber() operation and output stream support WriteNumber() operation.
Hint:-
a) Read the first number from all the K input streams and add them to a Priority Queue. (Nodes should keep track of the input stream)
b) Dequeue one element at a time from PQ, Put this element value to the output stream, Read the input stream number and from the same input stream add another element to PQ.
c) If the stream is empty, just continue
d) Repeat until PQ is empty.

3. In given K sorted Arrays of fixed length M. Also, given a final output array of length M*K. Give an efficient algorithm to merge all the Arrays into the final array, without using any extra space.
Hint: you can use the end of the final array to make PQ.

4. How will you sort 1 PB numbers? 1 PB = 1000 TB.

5. What will be the complexity of the above solution?

6. Any other improvement can be done on question 3 solution if the number of CPU cores is eight.

7. In given integer array that support three function findMin, findMax, findMedian. Sort the array.

8. In given pile of patient files of High, mid and low priority. Sort these files such that higher priority comes first, then mid and last low priority.
Hint: Bucket sort.

9. Write pros and cons of Heap-Sort, Merge-Sort and Quick-Sort.

10. In given rotated-sorted array of N integers. (The array was sorted then it was rotated some arbitrary number of times.) If all the elements in the array were unique find the index of some value.
Hint: Modified binary search

11. In the problem 9, what if there are repetitions allowed and you need to find the index of the first occurrence of the element in the rotated-sorted array.

12. Merge two sorted Arrays into a single sorted array.
Hint: Use merge method of Merge-Sort.

13. Given an array contain 0's and 1's, sort the array such that all the 0's come before 1's.

14. Given a list of English characters, sort the array in linear time.

15. Write a method to sort a list of strings so that all the anagrams are next to each other.
    Hint:-
    a) Loop through the list.
    b) For each word, sort the characters and add it to the hash map with keys as sorted word and value as the original word. At the end of the loop, you will get all anagrams as the value to a key (which is sorted by its constituent chars).
    c) Iterate over the hashmap, print all values of a key together and then move to the next key.
       Space Complexity: O(n), Time Complexity: O(n)

CHAPTER 6: LINKED LIST

Introduction

Let us suppose we have a list that contains following five elements 1, 2, 4, 5, 6. We want to insert a new element with value "3" in between "2" and "4". In the list, we cannot do it so easily. We need to create another list that is long enough to store the current values and one more space for "3". Then we need to copy these elements in the new space. This copy operation is inefficient. To remove this copy operation linked list is used.

Linked List

The linked list is a list of items, called nodes. Nodes have two parts, value part and link part. Value part is used to stores the data. Either the value part of the node can be a basic data-type like an integer or it can be some other data-type like an object of some class.
The link part is a reference, which is used to store addresses of the next element in the list.
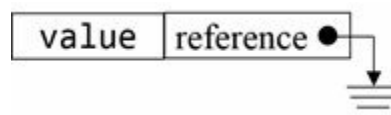
Types of Linked list

There are different types of linked lists. The main difference among them is how their nodes refer to each other.

Singly Linked List

Each node (Except the last node) has a reference to the next node in the linked list. The link portion of node contains the address of the next node. The link portion of the last node contains the value null
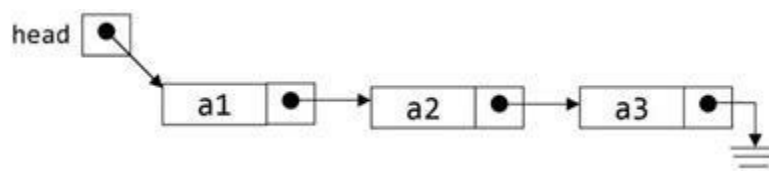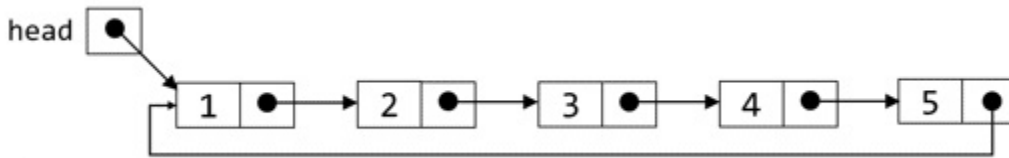

Doubly Linked list

The node in this type of linked list has reference to both previous and the next node in the list.
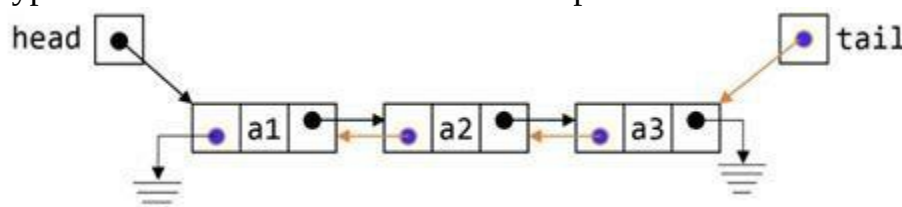
Circular Linked List

This type is similar to the singly linked list except that the last element have reference to the first node of the list. The link portion of the last node contains the address of the first node.


The various parts of linked list

1. Head: Head is a reference that holds the address of the first node in the linked list.
2. Nodes: Items in the linked list are called nodes.
3. Value: The data that is stored in each node of the linked list.
4. Link: Link part of the node is used to store the reference of other node.
   a. We will use "next" and "prev" to store address of next or previous node.

Singly Linked List
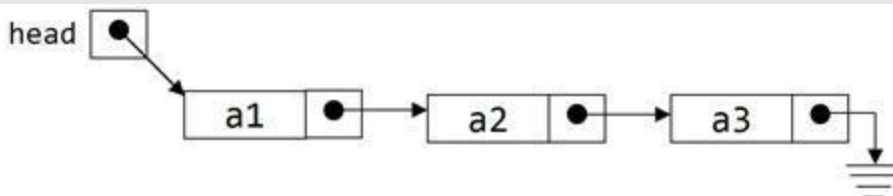
Let us look at the Node. The value part of node is of type integer, but it can be some other data-type. The link part of node is named as next in the below class definition.
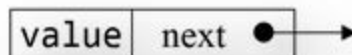
Note: For a singly linked, we should always test these three test cases before saying that the code is good to go. This one node and zero node case are used to catch boundary cases. It is always mandatory to take care of these cases before submitting code to the reviewer.
- Zero element / Empty linked list.
- One element / Just single node case.
- General case.

The various basic operations that we can perform on linked lists, many of these operations require list traversal:
- Insert an element in the list, this operation is used to create a linked list.
- Print various elements of the list.
- Search an element in the list.
- Delete an element from the list.
- Reverse a linked list.

You cannot use Head to traverse a linked list because if we use the head, then we lose the nodes of the list. We have to use another reference variable of same data-type as the head.

Example 6.1:

```
class LinkedList
    attr_accessor :head, :count
    def initialize()
        @head = nil
        @count = 0
    end

    class Node
        attr_accessor :value, :next
        def initialize(v, p = nil)
            @value = v
            @next = p
        end
    end

    #Other Methods.
end
```

Size of List

Example 6.2:

```
def size()
    return count
end
```

IsEmpty function

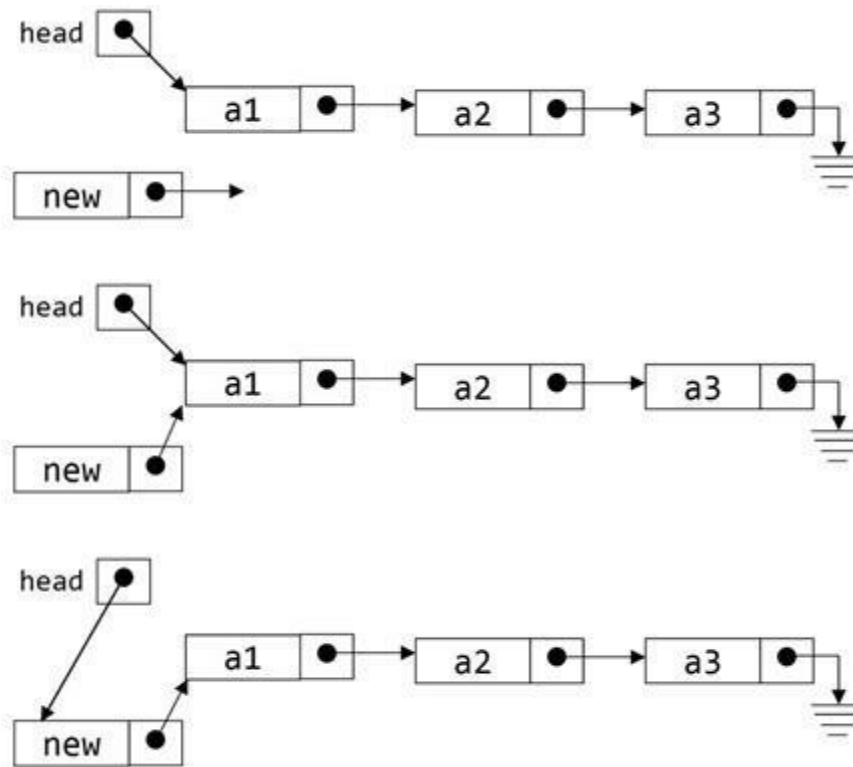**Example 6.3:**

```
def Empty
    return count == 0
end
```

Insert element in linked list

An element can be inserted into a linked list in various orders. Some of the example cases are mentioned below:

1. Insertion of an element at the start of linked list
2. Insertion of an element at the end of linked list
3. Insertion of an element at the Nth position in linked list
4. Insert element in sorted order in linked list
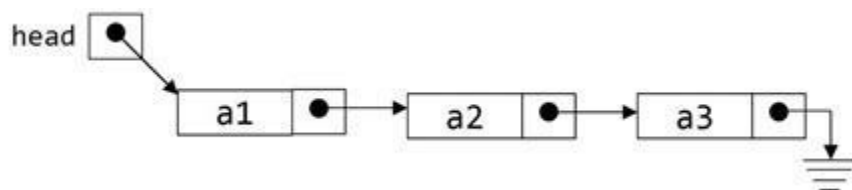
Insert element at the Head

Example 6.4:

```
def addHead(value)
    @head = Node.new(value, @head)
    @count += 1
end
```

Analysis:

· We need to create a new node with the value passed to the function as argument.
· While creating the new node the reference stored in head is passed as argument to Node() constructor so that the next reference will start pointing to the node or null which is referenced by the head node.
· The newly created node will become head of the linked list.
· Size of the list is increased by one.

Insertion of an element at the end

Example 6.5: Insertion of an element at the end of linked list

```
def addTail(value)
    newNode = Node.new(value, nil)
    curr = @head
    if @head == nil then
        @head = newNode
    end
    while curr.next != nil
        curr = curr.next
    end
```

```
    curr.next = newNode
    @count += 1
  end
```

Analysis:
- New node is created and the value is stored inside it.
- If the list is empty. Next of new node is null. And head will store the reference to the newly created node.
- If list is not empty then we have to traverse until the end of the list.
- Finally, new node is added to the end of the list.

Note: This operation is un-efficient as each time you want to insert an element you have to traverse to the end of the list. Therefore, the complexity of creation of the list is n^2. So how to make it efficient we have to keep track of the last element by keeping a tail reference. Therefore, if it is required to insert element at the end of linked list, then we will keep track of the tail reference also.

Traversing Linked List

Example 6.6: Print various elements of a linked list
```
  def printList()
    temp = @head
    while temp != nil
      print temp.value , " "
      temp = temp.next
    end
  end
```

Analysis:
We will store the reference of head in a temporary variable temp.
We will traverse the list by printing the content of list and always incrementing the temp by pointing to its next node.

Complete code for list creation and printing the list.

Example 6.7:
```
ll = LinkedList.new()
ll.addHead(1)
ll.addHead(2)
ll.addHead(3)
ll.printList()
```
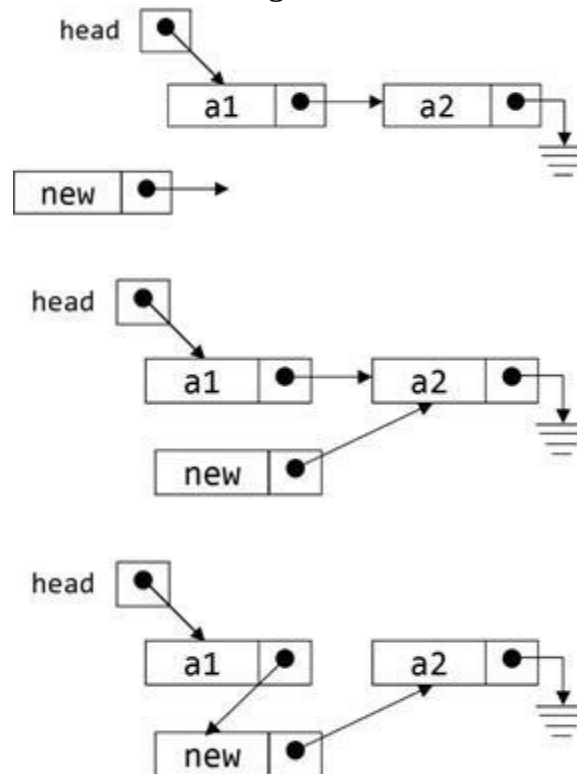
Analysis:
New instance of linked list is created. Various elements are added to list by calling addHead() method.
Finally all the content of list is printed to screen by calling print() method.

Sorted Insert

Insert an element in sorted order in linked list given Head reference

Example 6.8:

```
def sortedInsert(value)
    newNode = Node.new(value, nil)
    curr = @head
    if curr == nil or curr.value > value then
        newNode.next = head
        head = newNode
        return
    end
    while curr.next != nil and curr.next.value < value
        curr = curr.next
    end
    newNode.next = curr.next
    curr.next = newNode
end
```

Analysis:

·   Head of the list is stored in curr.
·   A new empty node of the linked list is created. And initialized by storing an argument value into its value. Next of the node will point to null.
·   It checks if the list is empty or if the value stored in the first node is greater than the current value. Then this new created node will be added to the start of the list. And head need to be modified.
·   We iterate through the list to find the proper position to insert the node.
·   Finally, the node will be added to the list.

Search Element in a Linked-List

Search element in linked list. Given a head reference and value. Returns true if value found in list else returns false.

Note: Search in a single linked list can only be done in one direction. Since all elements in the list have reference to the next item in the list. Therefore, traversal of linked list is linear in nature.
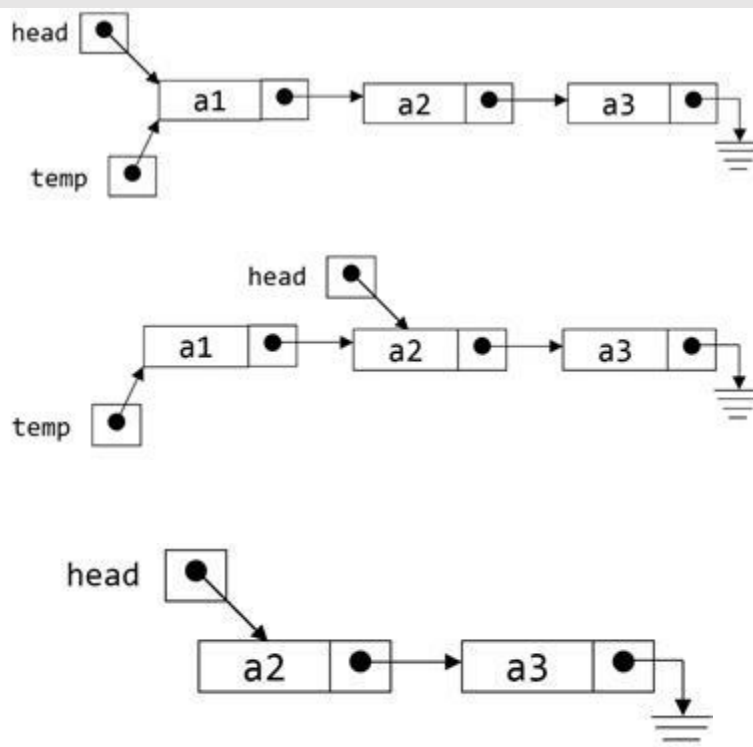
Example 6.9:

```
def isPresent(data)
    temp = @head
    while temp != nil
        if temp.value == data then
            return true
        end
        temp = temp.next
    end
    return false
end
```

Analysis:
- We create a temp variable, which will point to head of the list.
- Using a while loop we will iterate through the list.
- Value of each element of list is compared with the given value. If value is found, then the function will return true.
- If the value is not found, then false will be returned from the function in the end.

Delete element from the linked list

Delete First element in a linked list.

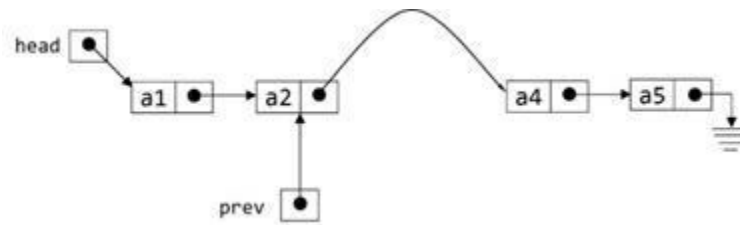**Example 6.10:**

```
def removeHead()
    if self.Empty then
        raise StandardError, "EmptyListException"
    end
    value = @head.value
    @head = @head.next
    @count -= 1
    return value
end
```

Analysis:

- First, we need to check if the list is already empty. If list is already empty then throw EmptyListException.
- If list is not empty then store the value of head node in a temporary variable value.
- We need to find the second element of the list and assign it as head of the linked list.
- Since the first node is no longer referenced so it will be automatically deleted.
- Decrease the size of list. Then return the value stored in temporary variable value.

Delete node from the linked list given its value.
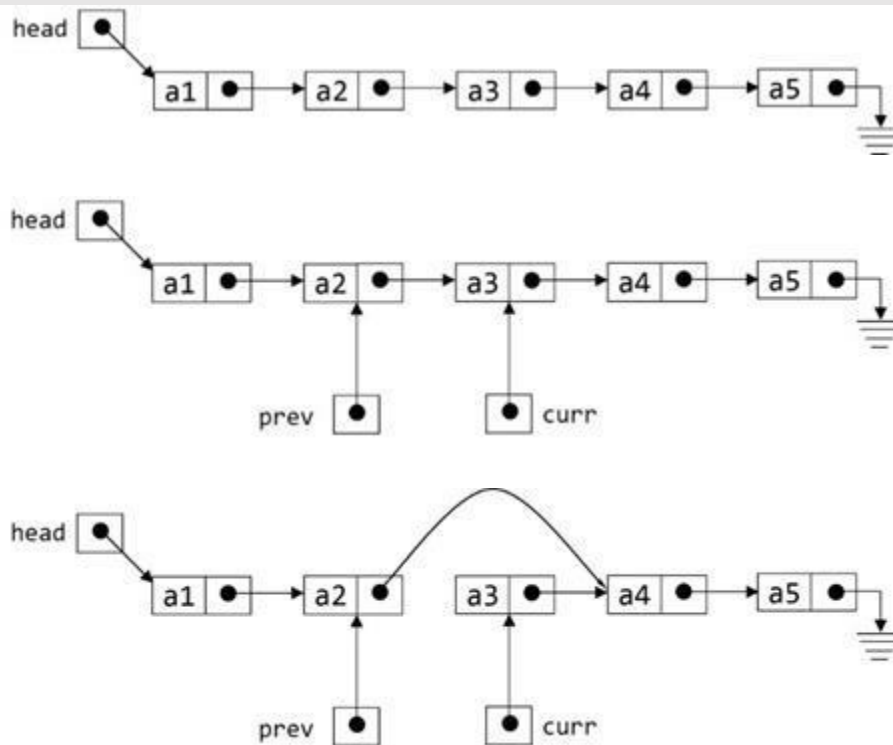
Example 6.11:

```
def deleteNode(delValue)
    temp = @head
    if self.Empty then
        return false
    end
    if delValue == @head.value then
        @head = @head.next
        @count -= 1
        return true
    end
    while temp.next != nil
        if temp.next.value == delValue then
            temp.next = temp.next.next
            count -= 1
            return true
        end
        temp = temp.next
    end
    return false
end
```

Analysis:

·   If the list is empty then we will return false from the function which indicate that the deleteNode() method executed with error.

·   If the node that need to be deleted is head node then head reference need to be modified and point to the next node.

·   In a while loop we will traverse the link list and try to find the node that need to be deleted. If the node is found then, we will point its reference to the node next to it and return true.

·   If the node is not found then we will return false.

Delete all the occurrence of particular value in linked list.

Example 6.12:

```
def deleteNodes(delValue)
    currNode = @head
    while currNode != nil and currNode.value == delValue #first node
        @head = currNode.next
        currNode = @head
    end
    while currNode != nil
        nextNode = currNode.next
```

```
            if nextNode != nil and nextNode.value == delValue then
                currNode.next = nextNode.next
            else
                currNode = nextNode
            end
        end
    end
```

Analysis:
- In the first while loop we will delete all the nodes that are at the front of the list, which have valued equal to delValue. In this, we need to update head of the list.
- In the second while loop, we will delete all the nodes that are having value equal to the delValue. Remember that we are not returning even though we have the node that we are looking for.

Delete a single linked list

Given a reference of head of linked list delete all the elements of a list.

Example 6.13:
```
    def freeList()
        head = nil
        count = 0
    end
```

Analysis: We just need to point head to null. The reference to the list is lost so it will automatically deleted.

Reverse a linked list.

Reverse a singly linked List iteratively using three Pointers

Example 6.14:
```
    def reverse()
        curr = @head
        prev = nil
        next1 = nil
        while curr != nil
            next1 = curr.next
            curr.next = prev
            prev = curr
            curr = next1
        end
        @head = prev
    end
```

Analysis: The list is iterated. Make next equal to the next node of the curr node. Make curr node's next point to prev node. Then iterate the list by making prev point to curr and curr point

to next.

Recursively Reverse a singly linked List

Reverse a singly linked list using Recursion.

Example 6.15:
```
def reverseRecurseUtil(currentNode, nextNode)
    if currentNode == nil then
        return nil
    end
    if currentNode.next == nil then
        currentNode.next = nextNode
        return currentNode
    end
    ret = self.reverseRecurseUtil(currentNode.next, currentNode)
    currentNode.next = nextNode
    return ret
end

def reverseRecurse()
    @head = self.reverseRecurseUtil(@head, nil)
end
```

Analysis:
·   ReverseRecurse function will call a reverseRecurseUtil function to reverse the list and the reference returned by the reverseRecurseUtil will be the head of the reversed list.
·   The current node will point to the nextNode that is previous node of the old list.

Note: A linked list can be reversed using two approaches the first approach is by using three references. The Second approach is using recursion both are linear solution, but three-reference solution is more efficient.

Remove duplicates from the linked list

Remove duplicate values from the linked list. The linked list is sorted and it contains some duplicate values, you need to remove those duplicate values. (You can create the required linked list using SortedInsert() function)

Example 6.16:
```
def removeDuplicate()
    curr = @head
    while curr != nil
        if curr.next != nil and curr.value == curr.next.value then
            curr.next = curr.next.next
        else
            curr = curr.next
        end
```

```
      end
   end
```

Analysis: While loop is used to traverse the list. Whenever there is a node whose value is equal to the next node's value, that current node next will point to the next of next node. Which will remove the next node from the list.

Copy List Reversed

Copy the content of linked list in another linked list in reverse order. If the original linked list contains elements in order 1,2,3,4, the new list should contain the elements in order 4,3,2,1.

Example 6.17:
```
def copyListReversed()
    ll = LinkedList.new()
    tempNode = nil
    tempNode2 = nil
    curr = @head
    while curr != nil
        tempNode2 = Node.new(curr.value, tempNode)
        curr = curr.next
        tempNode = tempNode2
    end
    ll.head = tempNode
    return ll
end
```

Analysis: Traverse the list and add the node's value to the new list. Since the list is traversed in the forward direction and each node's value is added to another list so the formed list is reverse of the given list.

Copy the content of given linked list into another linked list

Copy the content of given linked list into another linked list. If the original linked list contains elements in order 1,2,3,4, the new list should contain the elements in order 1,2,3,4.

Example 6.18:
```
def copyList()
    ll = LinkedList.new()
    headNode = nil
    tailNode = nil
    tempNode = nil
    curr = @head
    if curr == nil then
        return nil
    end
    headNode = Node.new(curr.value, nil)
    tailNode = headNode
```

```
            curr = curr.next
            while curr != nil
                tempNode = Node.new(curr.value, nil)
                tailNode.next = tempNode
                tailNode = tempNode
                curr = curr.next
            end
            ll.head = headNode
            return ll
    end
```

Analysis: Traverse the list and add the node's value to new list, but this time always at the end of the list. Since the list is traversed in the forward direction and each node's value is added to the end of another list. Therefore, the formed list is same as the given list.

Compare List

Compare the values of two linked lists given their head pointers.

Example 6.19: Compare two list given

```
    def compareList(ll)
        return self.compareListUtil(@head, ll.head)
    end

    def compareListUtil(head1, head2)
        if head1 == nil and head2 == nil then
            return true
        elsif (head1 == nil) or (head2 == nil) or (head1.value != head2.value) then
            return false
        else
            return self.compareListUtil(head1.next, head2.next)
        end
    end
```

Analysis:
·   List is compared recursively. Moreover, if we reach the end of the list and both the lists are null. Then both the lists are equal and so return true.
·   List is compared recursively. If either one of the list is empty or the value of corresponding nodes is unequal, then this function will return false.
·   Recursively calls compare list function for the next node of the current nodes.

Find Length

Example 6.20: Find the length of given linked list.

```
    def findLength()
        curr = @head
        count = 0
        while curr != nil
```

```
            count += 1
            curr = curr.next
        end
        return count
    end
```

Analysis: Length of linked list is found by traversing the list until we reach the end of list.

Nth Node from Beginning

Example 6.21: Find Nth node from beginning
```
    def nthNodeFromBegining(index)
        count = 0
        curr = @head
        while curr != nil and count < index - 1
            count += 1
            curr = curr.next
        end
        if curr == nil then
            raise StandardError, "null element"
        end
        return curr.value
    end
```

Analysis: Nth node can be found by traversing the list N-1 number of time and then return the node. If list does not have N elements then the method return null.

Nth Node from End

Example 6.22: Find Nth node from end
```
    def nthNodeFromEnd(index)
        size = self.findLength()
        if size != 0 and size < index then
            raise StandardError, "null element"
        end
        startIndex = size - index + 1
        return self.nthNodeFromBegining(startIndex)
    end
```

Analysis: First find the length of list, then nth node from end will be (length – nth +1) node from the beginning.

Example 6.23:
```
    def nthNodeFromEnd2(index)
        count = 0
        forward = @head
        curr = head
        while forward != nil and count < index - 1
```

```
            count += 1
            forward = forward.next
        end
        if forward == nil then
            raise StandardError, "null element"
        end
        while forward != nil
            forward = forward.next
            curr = curr.next
        end
        return curr.value
    end
```
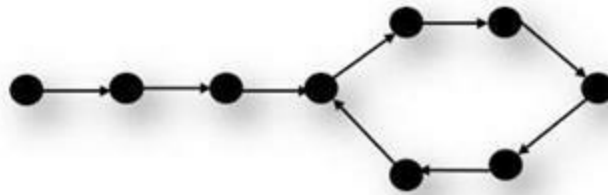
Analysis: Second approach is to use two references one is N steps / nodes ahead of the other when forward reference reach the end of the list then the backward reference will point to the desired node.

Loop Detect

Find if there is a loop in a linked list. If there is a loop, then return 1 if not, then return 0.

There are many ways to find if there is a loop in a linked list:
Approach 1: User some map or hash-table
   a) Traverse through the list.
   b) If the current node is, not there in the Hash-Table then insert it into the Hash-Table.
   c) If the current node is already in the Hashtable then we have a loop.

Approach 2: Slow reference and fast reference approach (SPFP)

Approach 3: Reverse list approach"

Slow reference and fast reference approach (SPFP)

We will use two references, one will move 2 steps at a time and another will move 1 step at time. If there is, a loop then both will meet at a point.
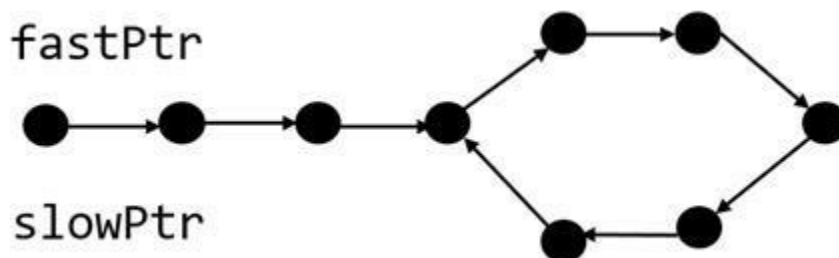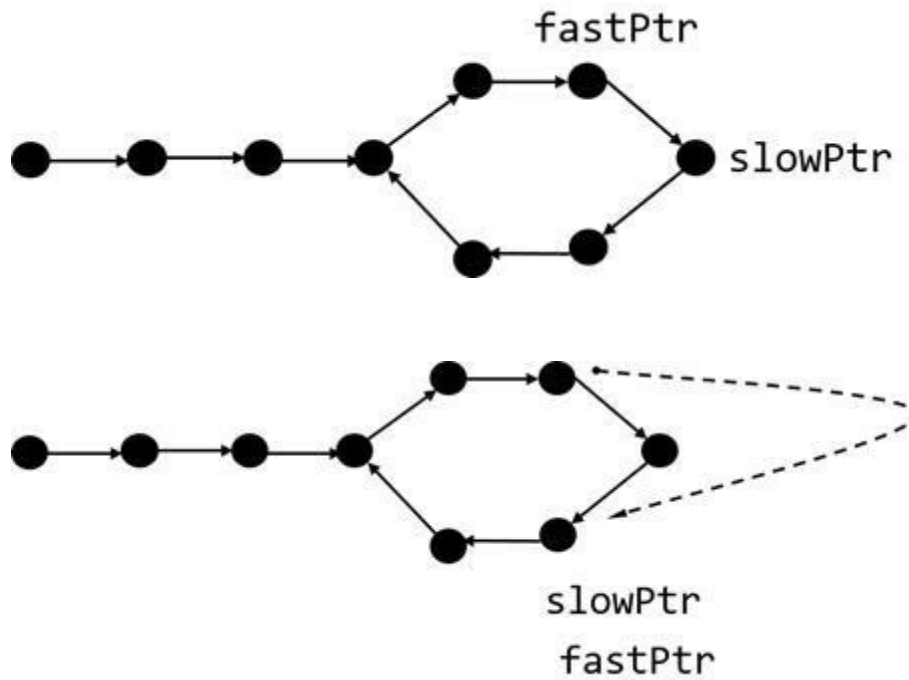
Example 6.24:

```
def loopDetect()
    slowPtr = fastPtr = @head
    while fastPtr.next != nil and fastPtr.next.next != nil
        slowPtr = slowPtr.next
        fastPtr = fastPtr.next.next
        if slowPtr == fastPtr then
            puts "loop found"
            return true
        end
    end
    puts "loop not found"
    return false
end
```

Analysis:

·   The list is traversed with two references, one is slow reference and another is fast reference. Slow reference always moves one-step. Fast reference always moves two steps. If there is no loop, then control will come out of while loop. So return false.

·   If there is a loop, then there comes a point in a loop where the fast reference will come and try to pass slow reference and they will meet at a point. When this point arrives, we come to know that there is a loop in the list. So return true.

Reverse List Loop Detect

If there is a loop in a linked list then reverse list function will give head of the original list as the head of the new list.

Example 6.25: Find if there is a loop in a linked list. Use reverse list approach.

```
def reverseListLoopDetect()
```

```
            tempHead = @head
            self.reverse()
            if tempHead == @head then
                self.reverse()
                puts "loop found"
                return true
            else
                self.reverse()
                puts "loop not found"
                return false
            end
        end
```

Analysis:
- Store reference of the head of list in a temp variable.
- Reverse the list
- Compare the reversed list head reference to the current list head reference.
- If the head of reversed list and the original list are same then reverse the list back and return true.
- If the head of the reversed list and the original list are not same, then reverse the list back and return false. Which means there is no loop.

Note: Both SPFP and Reverse List approaches are linear in nature, but still in SPFP approach, we do not require to modify the linked list so it is preferred.

Loop Type Detect

Find if there is a loop in a linked list. If there is no loop, then return 0, if there is loop return 1, if the list is circular then 2. Use slow reference fast reference approach.

Example 6.26:
```
    def loopTypeDetect()
        slowPtr = fastPtr = @head
        while fastPtr.next != nil and fastPtr.next.next != nil
            if @head == fastPtr.next or head == fastPtr.next.next then
                print "circular list loop found"
                return 2
            end
            slowPtr = slowPtr.next
            fastPtr = fastPtr.next.next
            if slowPtr == fastPtr then
                print "loop found"
                return 1
            end
        end
        print "loop not found"
        return 0
    end
```

Analysis: This program is same as the loop detect program only if it is a circular list than the fast reference reaches the slow reference at the head of the list this means that there is a loop at the beginning of the list.

Remove Loop

Example 6.27: Given there is a loop in linked list remove the loop.

```
def loopPointDetect()
    slowPtr = fastPtr = @head
    while fastPtr.next != nil and fastPtr.next.next != nil
        slowPtr = slowPtr.next
        fastPtr = fastPtr.next.next
        if slowPtr == fastPtr then
            return slowPtr
        end
    end
    return nil
end

def removeLoop()
    loopPoint = self.loopPointDetect()
    if loopPoint != nil then
        return
    end
    firstPtr = @head
    if loopPoint == @head then
        while firstPtr.next != head
            firstPtr = firstPtr.next
        end
        firstPtr.next = nil
        return
    end
    secondPtr = loopPoint
    while firstPtr.next != secondPtr.next
        firstPtr = firstPtr.next
        secondPtr = secondPtr.next
    end
    secondPtr.next = nil
end
```

Analysis:

· Loop through the list by two reference, one fast reference and one slow reference. Fast reference jumps two nodes at a time and slow reference jump one node at a time. The point where these two reference intersect is a point in the loop.

· If that intersection point is head of the list, this is a circular list case and you need to again traverse through the list and make the node before head point to null.

· In the other case, you need to use two reference variables one starts from head and another

starts form the intersection-point. They both will meet at the point of loop. (You can mathematically prove it ;))

Find Intersection
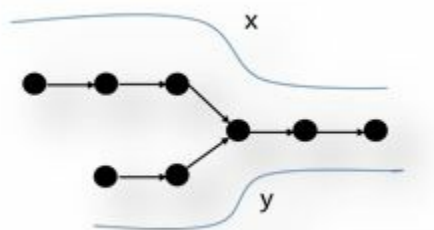
Example 6.28: In given two linked list which meet at some point find that intersection point.

```
def findIntersection(list2)
    l1 = 0
    l2 = 0
    tempHead = @head
    tempHead2 = list2.head
    while tempHead != nil
        l1 += 1
        tempHead = tempHead.next
    end
    while tempHead2 != nil
        l2 += 1
        tempHead2 = tempHead2.next
    end
    if l1 < l2 then
        temp = head
        head = head2
        head2 = temp
        diff = l2 - l1
    else
        diff = l1 - l2
    end
    while diff > 0
        head = head.next
        diff -= 1
    end
    while head != head2
        head = head.next
        head2 = head2.next
    end
    return head
    end
end
```

Analysis: Find length of both the lists. Find the difference of length of both the lists. Increment the longer list by diff steps, and then increment both the lists and get the intersection point.

Doubly Linked List

In a Doubly Linked list, there are two references in each node. These references are called prev and next. The prev reference of the node will point to the node before it and the next reference will point to the node next to the given node.
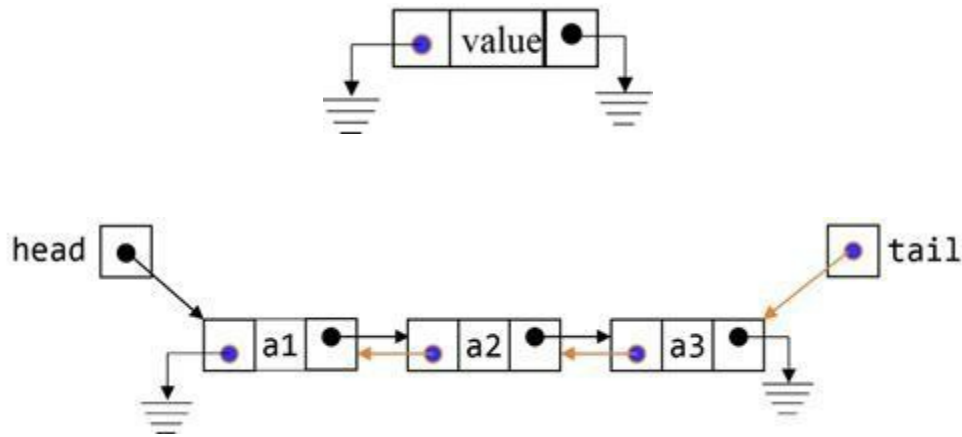
Let us look at the Node.   The value part of the node is of type integer, but it can be of some other data-type. The two link references are prev and next.

Search in a single linked list can only be done in one direction. Since all elements in the list has reference to the next item in the list. Therefore, traversal of linked list is linear in nature. In a doubly linked list, we keep track of both head of the linked list and tail of linked list.

In doubly linked list linked list below are few cases that we need to keep in mind while coding:
- · Zero element case (head and tail both can be modified)
- · Only element case (head and tail both can be modified)
- · First element (head can be modified)
- · General case
- · The last element (tail can be modified)

Note: Any program that is likely to change head reference or tail reference is to be passed as a double reference, which is pointing to head or tail reference.

Basic operations of Linked List

Basic operation of a linked list requires traversing a linked list. The various operations that we can perform on linked lists, many of these operations require list traversal:
1. Insert an element in the list, this operation is used to create a linked list.
2. Print various elements of the list.
3. Search an element in the list.
4. Delete an element from the list.
5. Reverse a linked list.

For doubly linked list, we have following cases to consider:

1. null values (head and tail both can be modified)
2. Only element (head and tail both can be modified)
3. First element (head can be modified)
4. General case
5. Last element (tail can be modified)

Example 6.29:

```ruby
class DoublyLinkedList
    attr_accessor :head, :tail, :count
    def initialize()
        @head = nil
        @tail = nil
        @count = 0
    end

    class Node
        attr_accessor :value, :next, :prev
        def initialize(v, n = nil, p = nil)
            @value = v
            @next = n
            @prev = p
        end
    end

    def size()
        return @count
    end

    def Empty
        return @count == 0
    end

    def peek()
        if self.Empty then
            raise StandardError, "EmptyListException"
        end
        return @head.value
    end
    #Other Methods.
end
```

Insert at Head
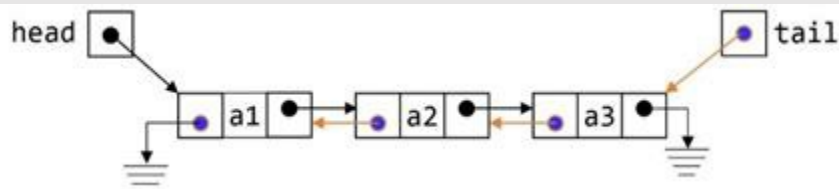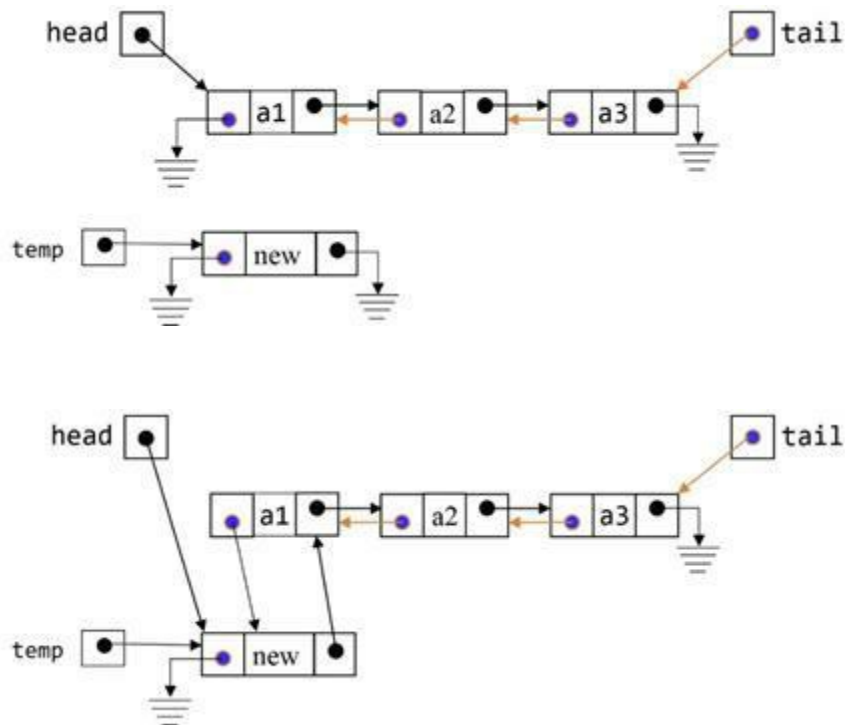
Example 6.30:

```
def addHead(value)
    newNode = Node.new(value, nil, nil)
    if @count == 0 then
        @tail = @head = newNode
    else
        @head.prev = newNode
        newNode.next = @head
        @head = newNode
    end
    @count += 1
end
```

Analysis: Insert in double linked list is same as insert in a singly linked list.
· Create a node assign null to prev reference of the node.
· If the list is empty then tail and head will point to the new node.
· If the list is not empty then prev of head will point to newNode and next of newNode will point to head. Then head will be modified to point to newNode.

Insert at Tail

Example 6.31: Insert an element at the end of the list.

```
def addTail(value)
    newNode = Node.new(value, nil, nil)
    if @count == 0 then
        @head = @tail = newNode
    else
        newNode.prev = @tail
        @tail.next = newNode
        @tail = newNode
```

```
            end
         @count += 1
      end
```

Analysis: Find the proper location of the node and add it to the list. Manage next and prev reference of the node so that list always remains double linked list.

Remove Head of doubly linked list

Example 6.32:
```
   def removeHead()
      if self.Empty then
         raise StandardError, "EmptyListException"
      end
      value = @head.value
      @head = @head.next
      if @head == nil then
         @tail = nil
      else
         @head.prev = nil
      end
      @count -= 1
      return @value
   end
```

Analysis:
·   If the list is empty then EmptyListException will be raised.
·   Now head will point to its next.
·   If head is null then this was single node list case, tail also needed to be made null.
·   In all the general case head. Prev will be set to null.
·   Size of list will be reduced by one and value of node is returned.

Delete a node given its value
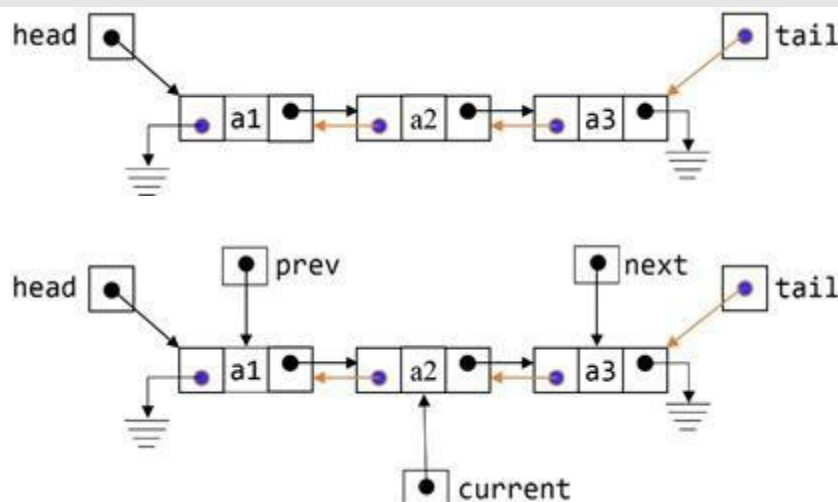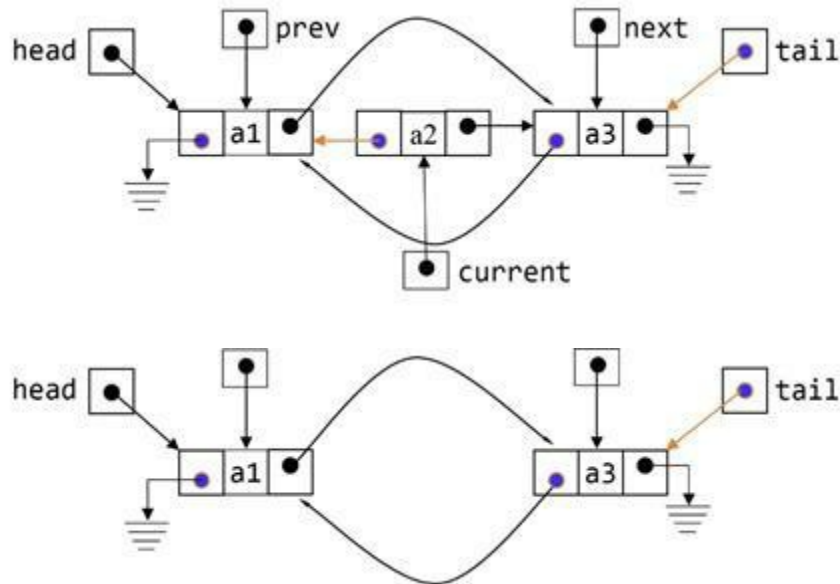
Example 6.33: Delete node in linked list

```
def removeNode(key)
    curr = @head
    if curr == nil then #empty list
        return false
    end
    if curr.value == key then #head is the node with value key.
        @head = @head.next
        @count -= 1
        if @head == nil then
            @tail = nil
        end # only one element in list.
        return true
    end
    while curr.next != nil
        if curr.next.value == key then
            curr.next = curr.next.next
            if curr.next == nil then #last element case.
                @tail = curr
            else
                curr.next = curr
            end
            @count -= 1
            return true
        end
        curr = curr.next
    end
    return false
end
```

Analysis: Traverse the list find the node which needs to be deleted. Then remove it and adjust next reference of the node previous to it and prev reference of the node next to it.

Search list

Example 6.34:
```
def isPresent(key)
    temp = @head
    while temp != nil
        if temp.value == key then
            return true
        end
        temp = temp.next
    end
    return false
end
```
Analysis: Traverse the list and find if some value is resent or not.

Free List

Example 6.35:
```
def freeList()
    @head = nil
    @tail = nil
    @count = 0
end
```

Analysis: Just head and tail references need to point to null. The rest of the list will automatically deleted by garbage collection.

Print list

Example 6.36:
```
def printList()
    temp = @head
    while temp != nil
        print temp.value , " "
        temp = temp.next
    end
    puts ""
end
```

Analysis: Traverse the list and print the value of each node.

Reverse a doubly linked List iteratively

Example 6.37:
```
# Reverse a doubly linked List iteratively
def reverseList()
    curr = @head
```

```
        while curr != nil
            tempNode = curr.next
            curr.next = curr.prev
            curr.prev = tempNode
            if curr.prev == nil then
                @tail = head
                @head = curr
                return
            end
            curr = curr.prev
        end
        return
    end
```

Analysis: Traverse the list. Swap the next and prev. then traverse to the direction curr.prev, which is next before swap. If you reach the end of the list then set head and tail.

Copy List Reversed

Example 6.38: Copy the content of the list into another list in reverse order.

```
    def copyListReversed()
        dll = DoublyLinkedList.new()
        curr = @head
        while curr != nil
            dll.addHead(curr.value)
            curr = curr.next
        end
        return dll
    end
```

Analysis:
· Create a DoublyLinkedList class object dll.
·   Traverse through the list and copy the value of the nodes into another list by calling addHead() method.
·   Since the new nodes are added to the head of the list, the new list formed have nodes order reverse there by making reverse list.

Copy List

Example 6.39:

```
def copyList()
    dll = DoublyLinkedList.new()
    curr = @head
    while curr != nil
        dll.addTail(curr.value)
        curr = curr.next
    end
    return dll
```

Analysis:

· Create a DoublyLinkedList class object dll.

· Traverse through the list and copy the value of the nodes into another list by calling addTail() method.

· Since the new nodes are added to the tail of the list, the new list formed have nodes order same as the original list.

Sorted Insert
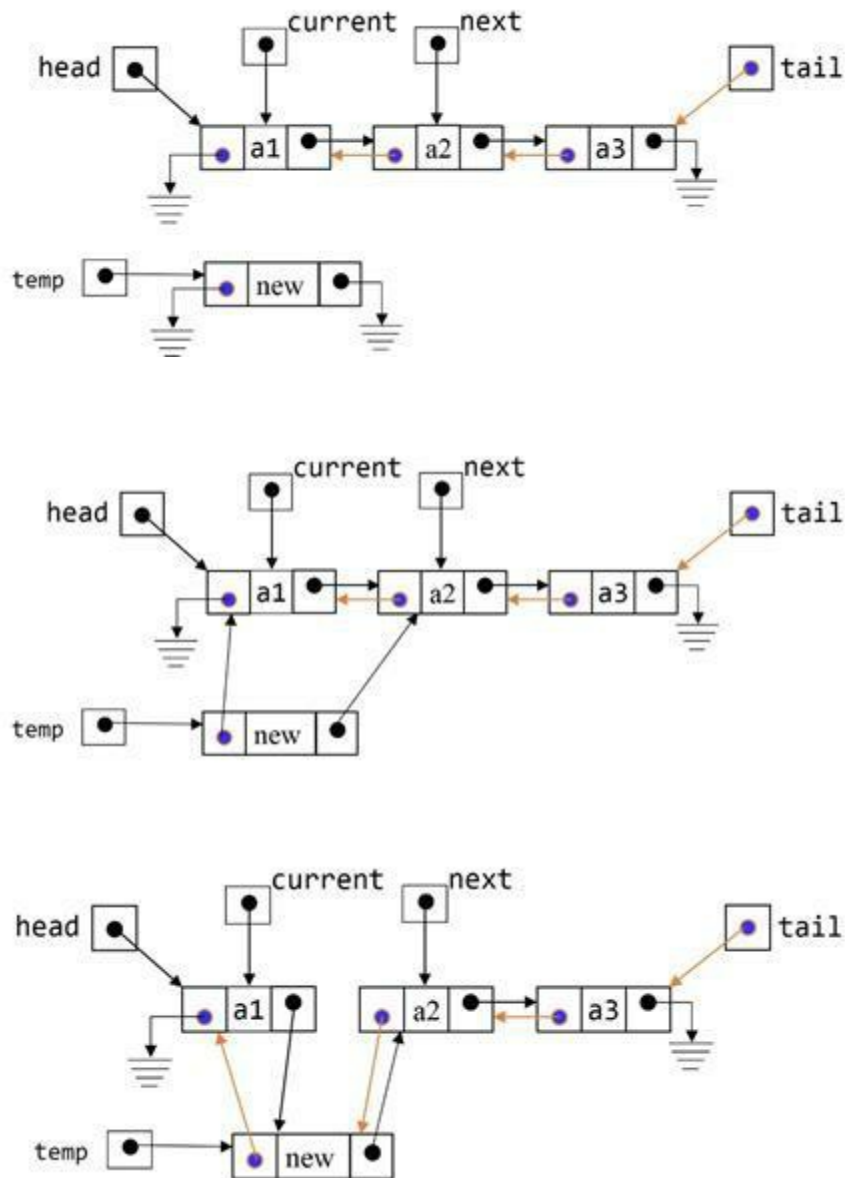
Example 6.40:

```
#SORTED INSERT DECREASING
    def sortedInsert(value)
        temp = Node.new(value)
        curr = @head
        if curr == nil then #first element
            @head = temp
```

```
            @tail = temp
        end
        if @head.value <= value then #at the begining
            temp.next = @head
            @head.prev = temp
            @head = temp
        end
        while curr.next != nil and curr.next.value > value #treversal
            curr = curr.next
        end
        if curr.next == nil then #at the end
            @tail = temp
            temp.prev = curr
            curr.next = temp
        else # all other
            temp.next = curr.next
            temp.prev = curr
            curr.next = temp
            temp.next.prev = temp
        end
    end
end
```

Analysis:
- We need to consider only element case first. In this case, both head and tail will modify.
- Then we need to consider the case when head will be modified when new node is added to the beginning of the list.
- Then we need to consider general cases
- Finally, we need to consider the case when tail will be modified.

Remove Duplicate

Example 6.41: Consider the list as sorted remove the repeated value nodes of the list.

```
# Remove Duplicate
def removeDuplicate()
    curr = @head
    while curr != nil
        if (curr.next != nil) and curr.value == curr.next.value then
            deleteMe = curr.next
            curr.next = deleteMe.next
            curr.next.prev = curr
            if deleteMe == @tail then
                @tail = curr
            end
        else
            curr = curr.next
        end
    end
end
```

Analysis:
· Removing duplicate is same as single linked list case.
· Head can never modify only the tail can modify when the last node is removed.

Circular Linked List

This type is similar to the singly linked list except that the last element points to the first node of the list. The link portion of the last node contains the address of the first node.
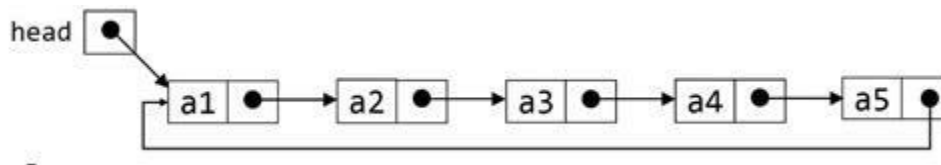
Example 6.42:

```ruby
class CircularLinkedList
    attr_accessor :tail, :count
    def initialize()
        @tail = nil
        @count = 0
    end

    class Node
        attr_accessor :value, :next
        def initialize(v, n = nil)
            @value = v
            @next = n
        end
    end

    def size()
        return @count
    end

    def Empty
        return @count == 0
    end

    def peek()
        if self.Empty then
            raise StandardError, "EmptyListException"
        end
        return tail.next.value
    end
    #Other Methods
end
```

Analysis: In the circular linked list, we just need the pointer to the tail node. As head node can be easily reached from tail node. Size(), isEmpty() and peek() functions remains the same.

Insert element in front
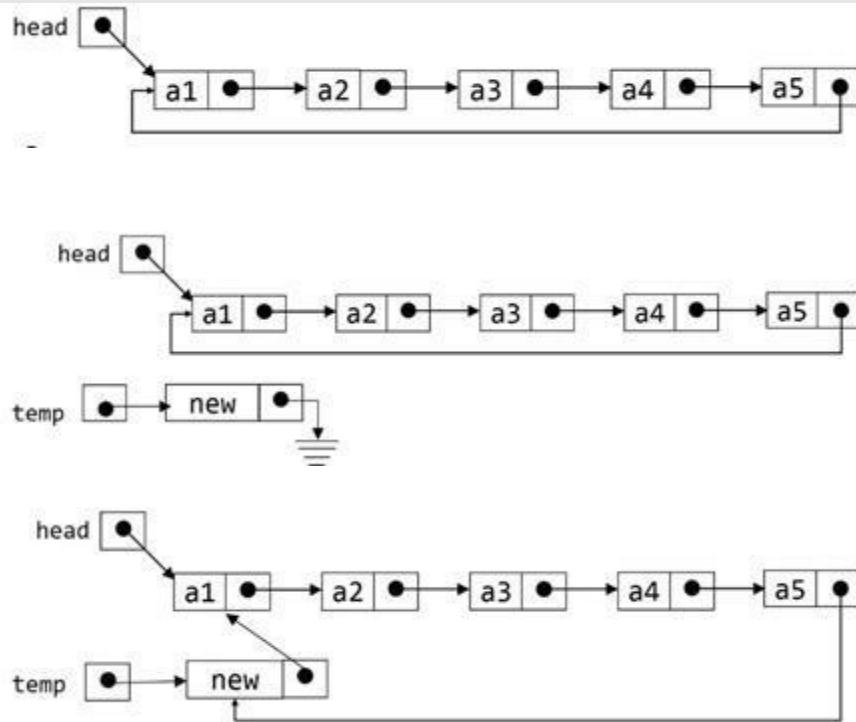
Example 6.43:
```
def addHead(value)
    temp = Node.new(value)
    if self.Empty then
        @tail = temp
        temp.next = temp
    else
        temp.next = @tail.next
        @tail.next = temp
    end
    @count += 1
end

# Testing code
ll = CircularLinkedList.new()
ll.addHead(1)
ll.addHead(2)
ll.addHead(3)
ll.addHead(4)
ll.addHead(5)
ll.addHead(6)
ll.printList()
```

Analysis:
· First, we create node with given value and its next pointing to null.

- If the list is empty then tail of the list will point to it. In addition, the next of node will point to itself
- If the list is not empty then the next of the new node will be next of the tail. In addition, tail next will start pointing to the new node.
- Thus, the new node is added to the head of the list.
- The demo program creates an instance of CircularLinkedList class. Then add some value to it and finally print the content of the list.

Insert element at the end

Example 6.44:

```
def addTail(value)
    temp = Node.new(value, nil)
    if self.Empty then
        @tail = temp
        temp.next = temp
    else
        temp.next = @tail.next
        @tail.next = temp
        @tail = temp
    end
    @count += 1
end
```

Analysis: Adding node at the end is same as adding at the beginning. We just need to modify tail reference in place of the head reference.

Search element in the list

Example 6.45:

```
def isPresent(data)
    temp = @tail
    i = 0
    while i < count
        if temp.value == data then
            return true
        end
        temp = temp.next
        i += 1
    end
    return false
end
```

Analysis: Iterate through the list to find if particular value is there or not.

Print the content of list

Example 6.46:

```
def printList()
    if self.Empty then
        return
    end
    temp = @tail.next
    while temp != @tail
        print temp.value , " "
        temp = temp.next
    end
    print temp.value
end
```

Analysis: In circular list, end of list is not there so we cannot check with null. In place of null, tail is used to check end of the list.

Remove element in the front

Example 6.47:
```
def removeHead()
    if self.Empty then
        raise StandardError, "EmptyListException"
    end
    value = @tail.next.value
    if @tail == @tail.next then
        @tail = nil
    else
        @tail.next = @tail.next.next
    end
    @count -= 1
    return @value
end
```

Analysis:
·   If the list is empty then exception will be thrown. Then the value stored in head is stored in local variable value.
·   If tail is equal to its next node that means there is only one node in the list so the tail will become null.
·   In all the other cases, the next of tail will point to next element of the head.
·   Finally, the value is returned.

Delete List

Example 6.48:
```
def freeList()
    @tail = nil
    count = 0
end
```

Analysis: The reference to the list is tail. By making tail null, the whole list is deleted.

Delete a node given its value
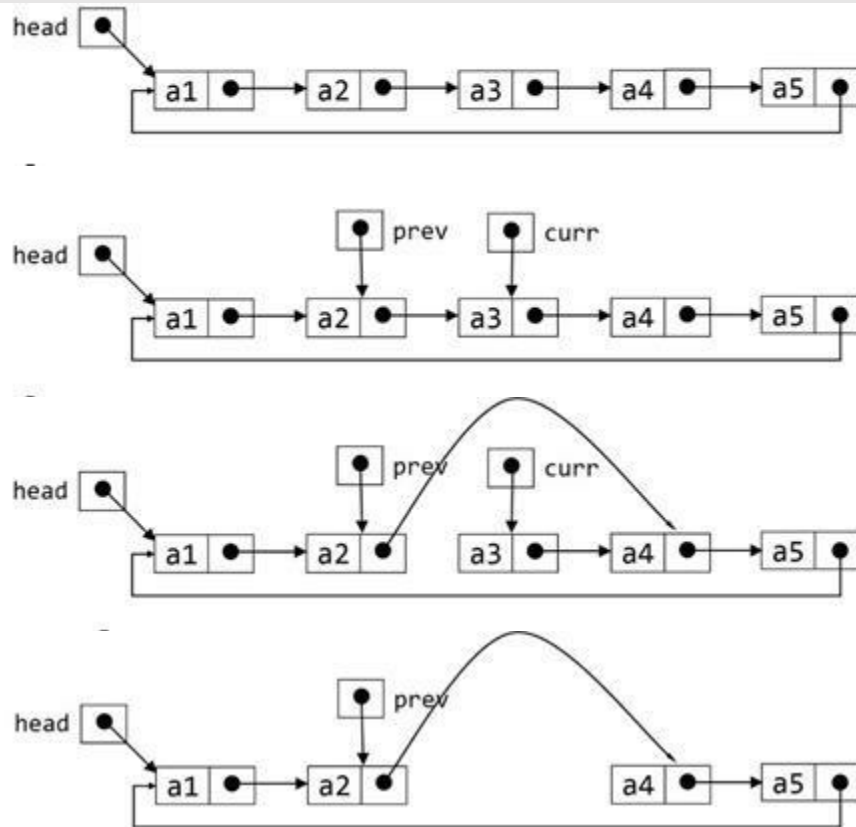
Example 6.49:
```
def removeNode(key)
    if self.Empty then
        return false
    end
    prev = @tail
    curr = @tail.next
    head = @tail.next
    if curr.value == key then #head and single node case.
        if curr == curr.next then #single node case
            @tail = nil
        else # head case
            @tail.next = @tail.next.next
        end
        @count -= 1
        return true
    end
    prev = curr
    curr = curr.next
    while curr != head
        if curr.value == key then
            if curr == @tail then
                @tail = prev
            end
```

```
            prev.next = curr.next
            @count -= 1
            return true
        end
        prev = curr
        curr = curr.next
    end
    return false
end
```

Analysis: Find the node that needs to be free. Only difference is that while traversing the list end of list is tracked by the head reference in place of null.

Copy List Reversed

Example 6.50:
```
def copyListReversed()
    cl = CircularLinkedList.new()
    curr = @tail.next
    head = curr
    if curr != nil then
        cl.addHead(curr.value)
        curr = curr.next
    end
    while curr != head
        cl.addHead(curr.value)
        curr = curr.next
    end
end
```

Analysis: The list is traversed and nodes are added to new list at the beginning. There by making the new list reverse of the given list.

Copy List

Example 6.51:
```
def copyList()
    cl = CircularLinkedList.new()
    curr = @tail.next
    head = curr
    if curr != nil then
        cl.addTail(curr.value)
        curr = curr.next
    end
    while curr != head
        cl.addTail(curr.value)
        curr = curr.next
```

```
      end
end
```

Analysis:

List is traversed and nodes are added to the new list at the end. There by making the list whose value are same as the input list.

Doubly Circular list

1. For any linked list there are only three cases zero element, one element, general case
2. To doubly linked list we have a few more things
   a) null values
   b) Only element (it generally introduces an if statement with null)
   c) Always an "if" remains before "while". Which will check from this head.
   d) General case (check with the initial head kept)
   e) Avoid using recursion solutions it makes life harder
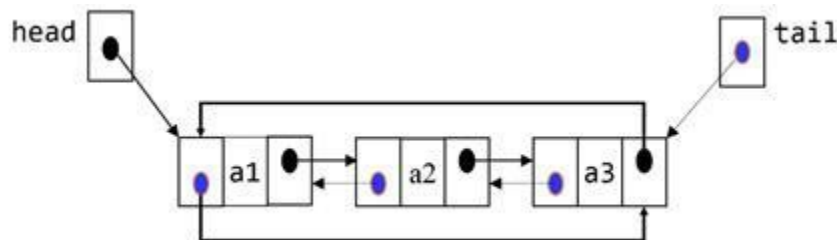
Example 6.52:
```
class DoublyCircularLinkedList

    attr_accessor :head, :tail, :count
    def initialize()
        @head = nil
        @tail = nil
        @count = 0
    end

    class Node
        attr_accessor :value, :next, :prev
        def initialize(v, n = nil, p = nil)
            @value = v
            @next = n
            @prev = p
        end
    end

    def size()
        return @count
    end

    def Empty
```

```
        return @count == 0
    end

    def peekHead()
        if self.Empty then
            raise StandardError, "EmptyListException"
        end
        return @head.value
    end

#Other Methods.
end
```

Insert Node at head

Example 6.53: Insert value at the front of the list.
```
    def addHead(value)
        newNode = Node.new(value)
        if @count == 0 then
            @tail = @head = newNode
            newNode.next = newNode
            newNode.prev = newNode
        else
            newNode.next = @head
            newNode.prev = @head.prev
            @head.prev = newNode
            newNode.prev.next = newNode
            @head = newNode
        end
        @count += 1
    end
```

Analysis:
·   A new node is created and if the list is empty then head and tail will point to it. The newly created newNode's next and prev also point to newNode.
·   If the list is not empty then the pointers are adjusted and a new node is added to the front of the list. Only head needs to be changed in this case.
·   Size of the list is increased by one.

Insert Node at tail

Example 6.54:
```
    def addTail(value)
        newNode = Node.new(value)
        if @count == 0 then
            @head = @tail = newNode
            newNode.next = newNode
            newNode.prev = newNode
```

```
        else
            newNode.next = @tail.next
            newNode.prev = @tail
            @tail.next = newNode
            @head.prev = newNode
            @tail = newNode
        end
        @count += 1
    end
```

Analysis:
·   A new node is created and if the list is empty then head and tail will point to it. The newly created newNode's next and prev also point to newNode.
·   If the list is not empty then the pointers are adjusted and a new node is added to the end of the list. Only tail needs to be changed in this case.
·   Size of the list is increased by one.

Print List

Example 6.55:
```
    def printList()
        if self.Empty then
            print "empty list"
            return
        end
        temp = @head
        begin
            print temp.value , " "
            temp = temp.next
        end while temp != @head
    end
```

Analysis: Traverse the list and print its content. Do..while is used as we want to terminate when temp is head. Moreover, want to process head node once.

Search value

Example 6.56:
```
    def isPresent(key)
        temp = @head
        if @head == nil then
            return false
        end
        begin
            if temp.value == key then
                return true
            end
            temp = temp.next
```

```
        end while temp != @head
        return false
    end
```

Analysis: Traverse through the list and see if given key is present or not. We use do..while loop as initial state is same as our termination state.

Delete list

Example 6.57:
```
    def freeList()
        @head = nil
        @tail = nil
        @count = 0
    end
```

Analysis: Remove the reference and list will be freed.

Delete head node

Example 6.58:
```
    def removeHead()
        if @count == 0 then
            raise StandardError, "EmptyListException"
        end
        value = @head.value
        @count -= 1
        if @count == 0 then
            @head = nil
            @tail = nil
            return value
        end
        nextNode = @head.next
        nextNode.prev = @tail
        @tail.next = nextNode
        @head = nextNode
        return value
    end
```

Analysis: Delete node in a doubly circular linked list is just same as delete node in a circular linked list. Just few extra reference need to be adjusted.

Delete tail node

Example 6.59:
```
    def removeTail()
        if count == 0 then
```

```
            raise StandardError, "EmptyListException"
        end
        value = @tail.value
        @count -= 1
        if @count == 0 then
            @head = nil
            @tail = nil
            return value
        end
        prev = @tail.prev
        prev.next = @head
        @head.prev = prev
        @tail = prev
        return value
    end
```

Analysis: Delete node in a doubly circular linked list is just same as delete node in a circular linked list. Just few extra reference need to be adjusted.

Exercise

1. Insert an element at k^{th} position from the start of linked list. Return 1 if success and if list is not long enough, then return -1.
   Hint: Take a reference of head and then advance it by K steps forward, and inserts the node.

2. Insert an element at k^{th} position from the end of linked list. Return 1 if success and if list is not long enough, then return -1.
   Hint: Take a reference of head and then advance it by K steps forward, then take another reference and then advance both simultaneously, so that when the first reference reaches the end of a linked list then second reference is at the point where you need to insert the node.

3. Consider there is a loop in a linked list, Write a program to remove loop if there is a loop in this linked list.

4. In the above SearchList program return, the count of how many instances of same value are found else if value not found then return 0. For example, if the value passed is "4". The elements in the list are 1,2,4,3 & 4. The program should return 2.

   Hint: In place of return 1 in the above program increment a counter and then return counter at the end.

5. Given two linked list head pointer and they meet at some point and need to find the point of intersection. However, in place of the end of both the linked list to be a null pointer there is a loop.
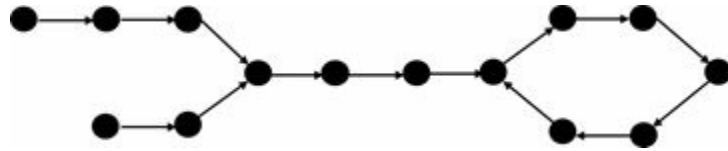
6. If linked list having a loop is given. Count the number of nodes in the linked list

7. We were supposed to write the complete code for the addition of polynomials using Linked Lists. This takes time if you do not have it by heart, so revise it well.

8. In given two linked lists. We have to find whether the data in one is reverse that of data in another. No extra space should be used and traverse the linked lists only once.

9. Find the middle element in a singly linked list. Tell the complexity of your solution.
   Hint:-
   Approach 1: Find the length of linked list. Then find the middle element and return it.
   Approach 2: Use two reference one will move fast and another will move slow, make sure you handle border case properly. (Even length and odd length linked list cases.)

10. Print list in reverse order.
    Hint: Use recursion.

CHAPTER 7: STACK

Introduction

A stack is a basic data structure that organizes items in last-in-first-out (LIFO) manner. Last element inserted in a stack will be the first to be removed from it.

The real-life analogy of the stack is "stack of plates". Imagine a stack of plates in a dining area everybody takes a plate at the top of the stack, thereby uncovering the next plate for the next person.

Stack allow to only access the top element. The elements that are at the bottom of the stack are the one that is going to stay in the stack for the longest time.
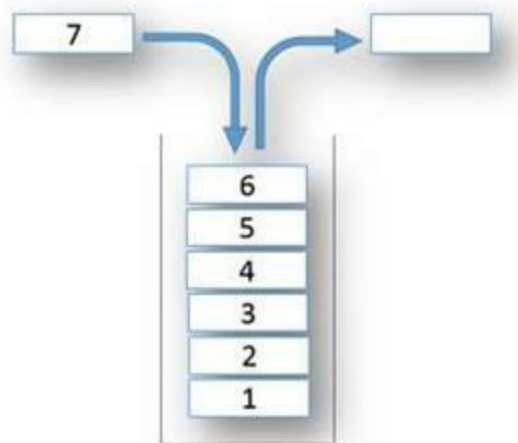
Computer science also has the common example of a stack. Function call stack is a good example of a stack. Function main() calls function foo() and then foo() calls bar(). These function calls are implemented using stack. First, bar() exists, then foo() and then finally main().

As we navigate from web page to web page, the URL of web pages are kept in a stack, with the current page URL at the top. If we click back button, then each URL entry is popped one by one.

The Stack Abstract Data Type

Stack abstract data type is defined as a class, which follows LIFO or last-in-first-out for the elements, added to it.
The stack should support the following operations:
1. Push(): Which adds a single element at the top of the stack
2. Pop(): Which removes a single element from the top of a stack.
3. Top(): Reads the value of the top element of the stack (does not remove it)
4. isEmpty(): Returns 1 if stack is empty
5. Size(): Returns the number of elements in a stack.
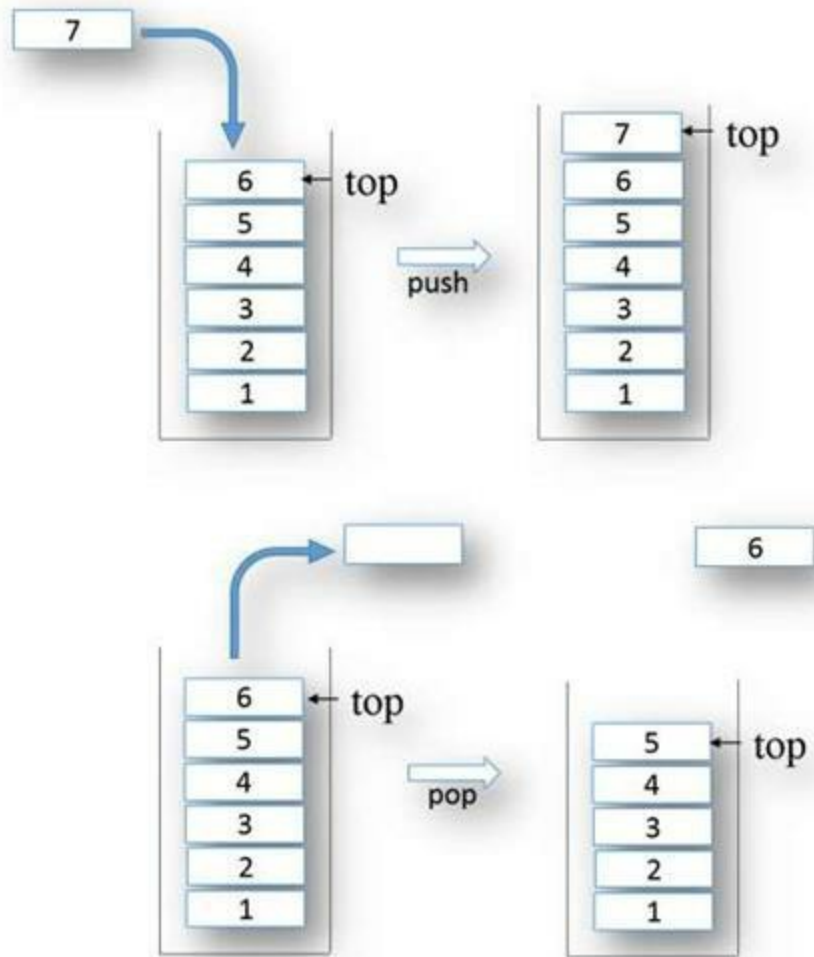
Add n to the top of a stack
def Push(value)

Remove the top element of the stack and return it to the caller function.
def Pop()

The stack can be implemented using an array or a linked list.
1. When stack is implemented using array, we will take care of its size.
2. In case of a linked list, there is no such limit on the number of elements it can contain.

When a stack is implemented, using an array, top of the stack is managed using an index variable called top.
When a stack is implemented using a linked list, push() and pop() is implemented using insert at the head of the linked list and remove from the head of the linked list.

Stack using Array

Implement a stack using a fixed length array.
Example 7.1:
class Stack
   def initialize(capacity=1000)
      @capacity = capacity
      @top = -1

```ruby
        @data = Array.new(@capacity,0)
    end
end
```

If user does not provide the max capacity of the array. Then an array of 1000 elements is created.

The top is the index to the top of the stack.

Number of elements in the stack is governed by the "top" index and top is initialized to -1 when a stack is initialized. Top index value of -1 indicates that the stack is empty in the beginning.

```ruby
    def empty
        return (@top == -1)
    end
```

isEmpty() function returns 1 if stack is empty or 0 in all other cases. By comparing the top index value with -1.

```ruby
    def size()
        return (@top + 1)
    end
```

size() function returns the number of elements in the stack. It just returns "top+1". As the top is referring the array index of the stack top variable so we need to add one to it.

```ruby
    def display()
        i = @top
        while i > -1
            print " " , @data[i]
            i -= 1
        end
        puts ""
    end
```

The display function will print the elements of the array.

```ruby
    def push(value)
        if self.size() == @data.size then
            raise StandardError, "StackOvarflowException"
        end
        @top += 1
        @data[@top] = value
    end
```

push() function checks whether the stack has enough space to store one more element, then it increases the "top" by one. Finally sort the data in the stack "data" array. In case, stack is full then "stack overflow" message is printed and that value will not be added to the stack and will be ignored.

```ruby
    def pop()
        if self.empty then
            raise StandardError, "StackEmptyException"
        end
        topVal = @data[@top]
        @top -= 1
        return topVal
    end
```

In the pop() function, first it will check that there are some elements in the stack by checking its top index. If some element is there in the stack, then it will store the top most element value in a variable "value". The top index is reduced by one. Finally, that value is returned.

```ruby
    def peek()
        if self.empty then
            raise StandardError, "StackEmptyException"
        end
        return @data[@top]
    end
```

top() function returns the value of stored in the top element of stack (does not remove it)

```ruby
# Testing code
s = Stack.new(1000)
s.push(1)
s.push(2)
s.push(3)
print s.pop()
```

Analysis:
· The user of the stack will create a stack local variable.
· Use push() and pop() functions to add / remove variables to the stack.
· Read the top element using the top() function call.
· Query regarding size of the stack using size() function call
· Query if stack is empty using isEmpty() function call

Stack using linked list

Example 7.2: Implement stack using a linked list.
```ruby
class ListStack
    attr_accessor :head, :count
    def initialize()
        @head = nil
        @count = 0
    end

    class Node
```

```ruby
        attr_accessor :value, :next
        def initialize(v, n = nil)
            @value = v
            @next = n
        end
    end

    def size()
        return @count
    end

    def empty
        return @count == 0
    end

    def peek()
        if self.empty then
            raise StandardError, "ListStackEmptyException"
        end
        return @head.value
    end

    def push(value)
        @head = Node.new(value, @head)
        @count += 1
    end

    def pop()
        if self.empty then
            raise StandardError, "ListStackEmptyException"
        end
        value = @head.value
        @head = @head.next
        @count -= 1
        return value
    end

    def insertAtBottom(value)
        if self.Empty then
            self.Push(value)
        else
            temp = self.Pop()
            self.insertAtBottom(value)
            self.Push(temp)
        end
    end

    def display()
```

```
            temp = @head
            while temp != nil
                print temp.value , " "
                temp = temp.next
            end
        end
end

# Testing code
s = ListStack.new()
s.push(1)
s.push(2)
s.push(3)
print s.pop()
print s.pop()
```

Analysis:
·   Stack implemented using a linked list is simply insertion and deletion at the head of a singly linked list.
·   In push() function, memory is created for one node. Then the value is stored into that node. Finally, the node is inserted at the beginning of the list.
·   In pop() function, the head of the linked list starts pointing to the second node there by releasing the memory allocated to the first node (Garbage collection).

Problems in Stack

Balanced Parenthesis

Example 7.3: Stacks can be used to check a program for balanced symbols (such as {}, (), []). The closing symbol should be matched with the most recently seen opening symbol.
Example: {()} is legal, {() ({})} is legal, but {((} and {()} are not legal

```
def isBalancedParenthesis(expn)
    stk = []
    expn.split("").each  do |ch|
        case ch
        when '{', '[', '('
            stk.push(ch)
        when '}'
            if stk.pop() != '{' then
                return false
            end
        when ']'
            if stk.pop() != '[' then
                return false
            end
        when ')'
            if stk.pop() != '(' then
```

```
            return false
          end
        end
      end
    end
    return stk.size == 0
end

# Testing code
expn = "{()}[]"
value = isBalancedParenthesis(expn)
puts "Given Expn: #{expn}"
puts "Result after isParenthesisMatched: #{value}"
```

Analysis:

·   Traverse the input string when we get an opening parenthesis we push it into stack. Moreover, when we get a closing parenthesis then we pop a parenthesis from the stack and compare if it is the corresponding to the one on the closing parenthesis.

·   We return false if there is a mismatch of parenthesis.

·   If at the end of the whole staring traversal, we reach to the end of the string and the stack is empty then we have balanced parenthesis.

Infix, Prefix and Postfix Expressions

When we have an algebraic expression like A + B then we know that the variable is being added to variable B. This type of expression is called **infix** expression because the operator "+" is there between operands A and operand B.

Now consider another infix expression A + B * C. In the expression there is a problem that in which order + and * work. Are A and B are added first and then the result is multiplied. Alternatively, B and C are multiplied first and then the result is added to A. This makes the expression ambiguous. To deal with this ambiguity we define the precedence rule or use parentheses to remove ambiguity.

So if we want to multiply B and C first and then add the result to A. Then the same expression can be written unambiguously using parentheses as A + (B * C). On the other hand, if we want to add A and B first and then the sum will be multiplied by C we will write it as (A + B) * C. Therefore, in the infix expression to make the expression unambiguous, we need parenthesis.

Infix expression: In this notation, we place operator in the middle of the operands.
< Operand > < operator > < operand >

Prefix expressions: In this notation, we place operator at the beginning of the operands.
< Operator > < operand > < operand >

Postfix expression: In this notation, we place operator at the end of the operands.
< Operand > < operand > < operator >

Infix Expression	Prefix Expression	Postfix Expression

A + B	+ A B	A B +
A + (B * C)	+ A * B C	A B C * +
(A + B) * C	* + ABC	A B + C *

Now comes the most obvious question why we need so unnatural Prefix or Postfix expressions when we already have infix expressions which words just fine for us. The answer to this is that infix expressions are ambiguous and they need parenthesis to make them unambiguous. While postfix and prefix notations do not need any parenthesis.

Infix-to-Postfix Conversion

Example 7.4:

```ruby
def precedence(x)
    if x == '(' then
        return (0)
    end
    if x == '+' or x == '-' then
        return (1)
    end
    if x == '*' or x == '/' or x == '%' then
        return (2)
    end
    if x == '^' then
        return (3)
    end
    return (4)
end

def infixToPostfix(expIn)
    expn = expIn.split("")
    return infixToPostfixUtil(expn)
end

def infixToPostfixUtil(expn)
    stk = []
    output = ""
    expn.each  do |ch|
        if ch <= '9' and ch >= '0' then
            output = output + ch
        else
            case ch
            when '+', '-', '*', '/', '%', '^'
                while stk.size != 0 and precedence(ch) <= precedence(stk[stk.size - 1])
                    temp = stk.pop()
```

```ruby
                    output = output + " " + temp
                end
                stk.push(ch)
                output = output + " "
            when '('
                stk.push(ch)
            when ')'
                while stk.size != 0 and (temp = stk.pop()) != '('
                    output = output + " " + temp + " "
                end
            end
        end
    end
    while stk.size != 0
        temp = stk.pop()
        output = output + temp + " "
    end
    return output
end
# Testing code
expn = "10+((3))*5/(16-4)"
value = infixToPostfix(expn)
puts "Infix Expn: #{expn}"
puts "Postfix Expn: #{value}"
```

Analysis:
- Print operands in the same order as they arrive.
- If the stack is empty or contains a left parenthesis "(" on top, we should push the incoming operator in the stack.
- If the incoming symbol is a left parenthesis "(", push left parenthesis in the stack.
- If the incoming symbol is a right parenthesis ")", pop from the stack and print the operators until you see a left parenthesis ")". Discard the pair of parentheses.
- If the precedence of incoming symbol is higher than the precedence of operator at the top of the stack, then push it to the stack.
- If the incoming symbol has, an equal precedence compared to the top of the stack, use association. If the association is left to right, then pop and print the symbol at the top of the stack and then push the incoming operator. If the association is right to left, then push the incoming operator.
- If the precedence of incoming symbol is lower than the precedence of operator on the top of the stack, then pop and print the top operator. Then compare the incoming operator against the new operator at the top of the stack.
- At the end of the expression, pop and print all operators on the stack.

Infix-to-Prefix Conversion

Example 7.5:
```ruby
def infixToPrefix(expn)
    arr = expn.split("")
```

```ruby
        reverseString(arr)
        replaceParanthesis(arr)
        arr = infixToPostfixUtil(arr)
        reverseString(arr)
        return arr
end

def replaceParanthesis(a)
        lower = 0
        upper = a.size - 1
        while lower <= upper
                if a[lower] == '(' then
                        a[lower] = ')'
                elsif a[lower] == ')' then
                        a[lower] = '('
                end
                lower += 1
        end
end

def reverseString(expn)
        lower = 0
        upper = expn.size - 1
        while lower < upper
                tempChar = expn[lower]
                expn[lower] = expn[upper]
                expn[upper] = tempChar
                lower += 1
                upper -= 1
        end
end

# Testing code
expn = "10+((3))*5/(16-4)"
value = infixToPrefix(expn)
puts "Infix Expn: #{expn}"
puts "Prefix Expn: #{value}"
```

Analysis:
1. Reverse the given infix expression.
2. Replace '(' with ')' and ')' with '(' in the reversed expression.
3. Now, apply infix to postfix subroutine already discussed.
4. Reverse the generated postfix expression and this will give required prefix expression.

Postfix Evaluate

Write a postfixEvaluate() function to evaluate a postfix expression. Such as: 1 2 + 3 4 + *

Example 7.6:
```ruby
def postfixEvaluate(expn)
    stk = []
    expn.split(" ").each do |token|
        if "+-*/".include?(token) then
            num1 = stk.pop()
            num2 = stk.pop()
            case token
            when "+"
                stk.push(num1 + num2)
            when "-"
                stk.push(num1 - num2)
            when "*"
                stk.push(num1 * num2)
            when "/"
                stk.push(num1 / num2)
            end
        else
            stk.push(token.to_i)
        end
    end
    return stk.pop()
end

# Testing code
expn = "6 5 2 3 + 8 * + 3 + *"
value = postfixEvaluate(expn)
puts "Given Postfix Expn: #{expn}"
puts "Result after Evaluation: #{value}"
```

Analysis:
1) Create a stack to store values or operands.
2) Scan through the given expression and do following for each element:
    a) If the element is a number, then push it into the stack.
    b) If the element is an operator, then pop values from the stack. Evaluate the operator over the values and push the result into the stack.
3) When the expression is scanned completely, the number in the stack is the result.

Min stack

Design a stack in which we can get minimum value in stack should also work in **O(1)** Time Complexity.

Hint: Keep two stack one will be general stack, which will just keep the elements. The second will keep the min value.

1. Push: Push an element to the top of stack1. Compare the new value with the value at the top of the stack2. If the new value is smaller, then push the new value into stack2. Or push the value at the top of the stack2 to itself once more.
2. Pop: Pop an element from top of stack1 and return. Pop an element from top of stack2 too.

3. Min: Read from the top of the stack2 this value will be the min.

Palindrome string

Find if given string is a palindrome or not using a stack.
Definition of palindrome: A palindrome is a sequence of characters that is same backward or forward.
Eg. "AAABBBCCCBBBAAA", "ABA" & "ABBA"

Hint: Push characters to the stack until the half-length of the string. Then pop these characters and then compare. Make sure you take care of the odd length and even length.

Depth-First Search with a Stack

In a depth-first search, we traverse down a path until we get a dead end; then we backtrack by popping a stack to get an alternative path.
· Create a stack
· Create a start point
· Push the start point onto the stack
· While (value searching is not found and the stack is not empty)
    o Pop the stack
    o Find all possible points after the one which we just tried
    o Push these points onto the stack

Stack using a queue

How to implement a stack using a queue. Analyse the running time of the stack operations.
See queue chapter for this.

Stack Span Problem

In given list of daily stock price in a list A[i].   Find the span of the stocks for each day. A span of stock is the maximum number of days for which the price of stock was lower than that day.
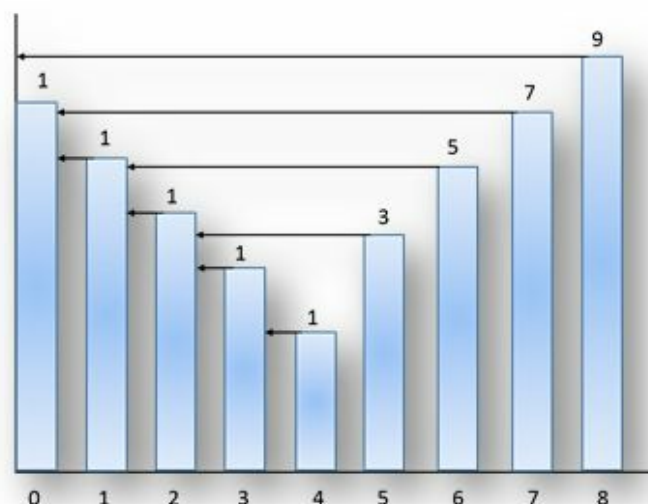
Example 7.7: Approach 1

```
def StockSpanRange(arr)
    sr = Array.new(arr.size)
    sr[0] = 1
    i = 1
    while i < arr.size
        sr[i] = 1
        j = i - 1
        while (j >= 0) and (arr[i] > arr[j])
            sr[i] += 1
            j -= 1
        end
        i += 1
    end
    return sr
end

# Testing code
arr = [8, 6, 5, 3, 2, 4, 6, 8, 9]
value = StockSpanRange(arr)
print "StockSpanRange: " , value
```

Example 7.8: Approach 2:

```
def StockSpanRange2(arr)
    stk = []
    sr = Array.new(arr.size)
    stk.push(0)
    sr[0] = 1
    i = 1
    while i < arr.size
        while stk.size != 0 and arr[stk[stk.length() - 1]] < arr[i]
            stk.pop()
        end
        sr[i] = (stk.size == 0) ? (i + 1) : (i - stk[stk.length() - 1])
        stk.push(i)
        i += 1
    end
    return sr
end
```

Get Max Rectangular Area in a Histogram

In given histogram of rectangle bars of each one unit wide. Find the maximum area rectangle in the histogram.
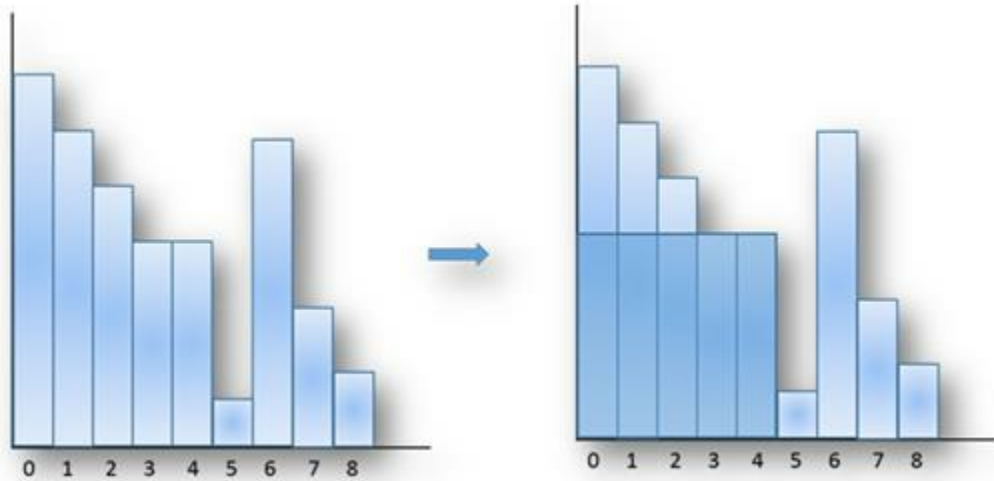
Example 7.9: Approach 1

```
def GetMaxArea(arr)
    size = arr.size
    maxArea = -1
    minHeight = 0
    i = 1
    while i < size
        minHeight = arr[i]
        j = i - 1
        while j >= 0
            if minHeight > arr[j] then
                minHeight = arr[j]
            end
            currArea = minHeight * (i - j + 1)
            if maxArea < currArea then
                maxArea = currArea
            end
            j -= 1
        end
        i += 1
    end
    return maxArea
end

# Testing code
arr = [7, 6, 5, 4, 4, 1, 6, 3, 2]
value = GetMaxArea (arr)
print "GetMaxArea: " , value
```

Approach 2: Divide and conquer

Approach 3
Example 7.10:

```
def GetMaxArea2(arr)
    size = arr.size
```

```
        stk = []
        maxArea = 0
        i = 0
        while i < size
            while (i < size) and (stk.size == 0 or arr[stk[stk.length() - 1]] <= arr[i])
                stk.push(i)
                i += 1
            end
            while stk.size != 0 and (i == size or arr[stk[stk.length() - 1]] > arr[i])
                top = stk.pop()
                topArea = arr[top] * (stk.size==0 ? i : i - stk[stk.length()-1]-1)
                if maxArea < topArea then
                    maxArea = topArea
                end
            end
        end
        return maxArea
end
```

Uses of Stack

- · Recursion can also be done using stack. (In place of the system stack)
- · The function call is implemented using stack.
- · When we want to reverse a sequence, we just push everything in stack and pop from it.
- · Grammar checking, balance parenthesis, infix to postfix conversion, postfix evaluation of expression etc.

Exercise

1. Converting Decimal Numbers to Binary Numbers using stack data structure.
   Hint: store reminders into the stack and then print the stack.

2. Convert an infix expression to prefix expression.
   Hint: Reverse given expression, Apply infix to postfix, and then reverse the expression again.
       Step 1. Reverse the infix expression.
               5^E+D*) C^B+A (
       Step 2. Make Every '(' as ')' and every ')' as '('
               5^E+D*(C^B+A)
       Step 3. Convert an expression to postfix form.
       Step 4. Reverse the expression.
               +*+A^BCD^E5

3. Write an HTML opening tag and closing tag-matching program.
   Hint: parenthesis matching.

4. Write a function that will transform Postfix to Infix Conversion

5. Write a function that will transform Prefix to Infix Conversion

6. Write a palindrome matching function, which ignores characters other than English alphabet and digits. String "Madam, I'm Adam." should return true.

7. In the Growing-Reducing Stack implementation using array. Try to figure out a better algorithm which will work similar to Vector<> or ArrayDeque<>.

CHAPTER 8: QUEUE

Introduction

A queue is a basic data structure that organizes items in first-in-first-out (FIFO) manner. First element, inserted into a queue, will be the first to be removed. It is also known as "first-come-first-served".

The real life analogy of queue is typical lines in which we all participate time to time.
- We wait in a line of railway reservation counter.
- We wait in the cafeteria line.
- We wait in a queue when we call to some customer-care.

The elements, which are at the front of the queue, are the one that stayed in the queue for the longest time.
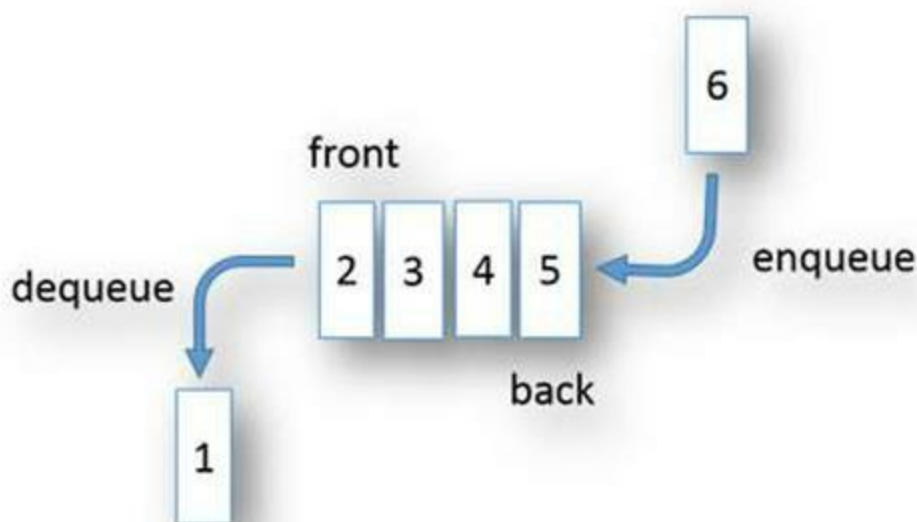
Computer science also has many common examples of queues. We issue a print command from our office to a single printer per floor. The print task are lined up in a printer queue. The print command that is issued first will be printed before the next commands in line.

In addition to printing queues, operating system is also using different queues to control process scheduling. Processes are added to processing queue, which is used by an operating system for various scheduling algorithms.

Soon we will study about graphs and will come to know about breadth-first traversal, which uses a queue.

The Queue Abstract Data Type

Queue abstract data type is defined as a class whose object follows FIFO or first-in-first-out for the elements, added to it.

Queue should support the following operations:
1. add(): Which adds a single element at the back of a queue
2. remove(): Which removes a single element from the front of a queue.
3. isEmpty(): Returns 1 if the queue is empty
4. size(): Returns the number of elements in a queue.

Queue Using Array

Circular view of arrays.
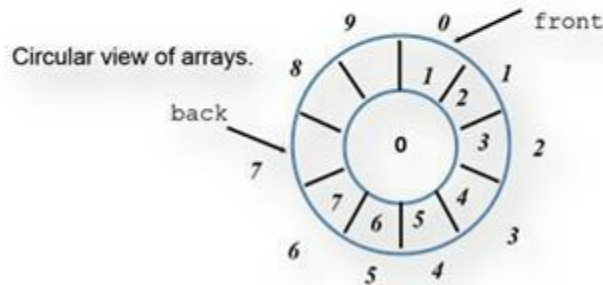
Example 8.1:

```ruby
class Queue
    def initialize(size = 100)
        @capacity = size
        @front = 0
        @back = 0
        @count = 0
        @data = Array.new(size,0)
    end

    def add(value)
        if @count >= @capacity then
            raise StandardError, "QueueFullException"
        else
            @count += 1
            @data[@back] = value
            @back = (@back += 1) % (@capacity - 1)
        end
    end

    def remove()
        if @count <= 0 then
            raise StandardError, "QueueEmptyException"
        else
            @count -= 1
            value = @data[@front]
```

```
            @front = (@front += 1) % (@capacity - 1)
        end
        return value
    end

    def empty
        return @count == 0
    end

    def size()
        return @count
    end
end

# Testing code
q = Queue.new()
q.add(1)
q.add(2)
q.add(3)
print q.remove()
print q.remove()
```

Analysis:
· Hear queue is created from a list.
· Add() to insert one element at the back of the queue.
· Remove() to delete one element from the front of the queue.

Queue Using linked list

Example 8.2:
```
class Queue
    attr_accessor :head, :tail, :count
    def initialize()
        @head = nil
        @tail = nil
        @count = 0
    end

    class Node
        attr_accessor :value, :next
        def initialize(v, n = nil)
            @value = v
            @next = n
        end
    end

    def size()
        return @count
```

```
    end

    def Empty
        return @count == 0
    end

    def peek()
        if self.Empty then
            raise StandardError, "QueueEmptyException"
        end
        return @head.value
    end

    def display()
        temp = head
        while temp != nil
            print temp.value , " "
            temp = temp.next
        end
    end

#other methods.
end
```

Add

Enqueue into a queue using linked list. Nodes are added to the end of the linked list. Below diagram indicates how a new node is added to the list. The tail is modified every time when a new value is added to the queue. However, the head is also updated in the case when there is no element in the queue and when that first element is added to the queue both head and tail will be pointing to it.
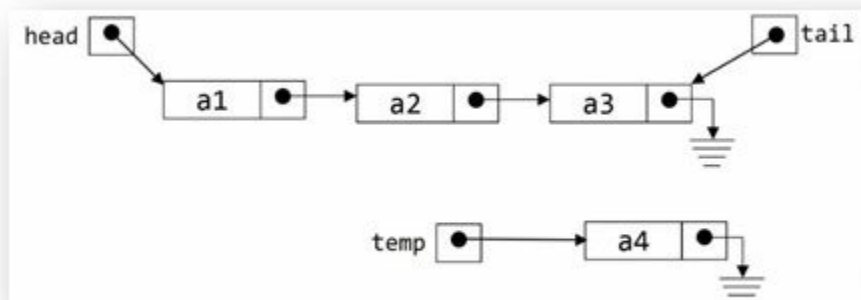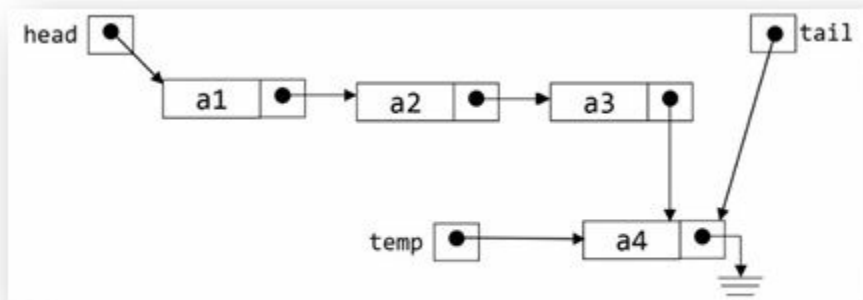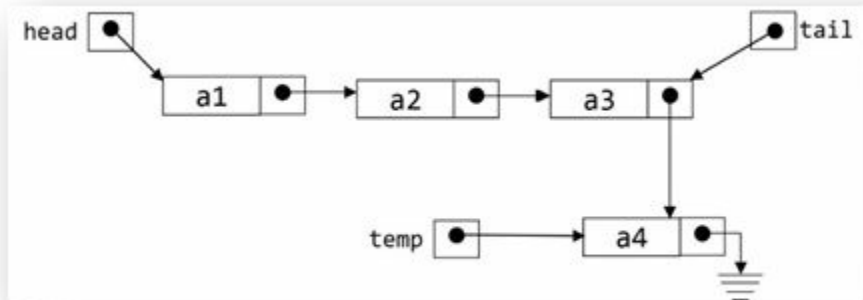
Example 8.3:

```
def add(value)
    temp = Node.new(value)
    if @head == nil then
        @head = @tail = temp
    else
        @tail.next = temp
        @tail = temp
    end
    @count += 1
end
```

Analysis: add operation add one element at the end of the Queue (linked list).

Remove
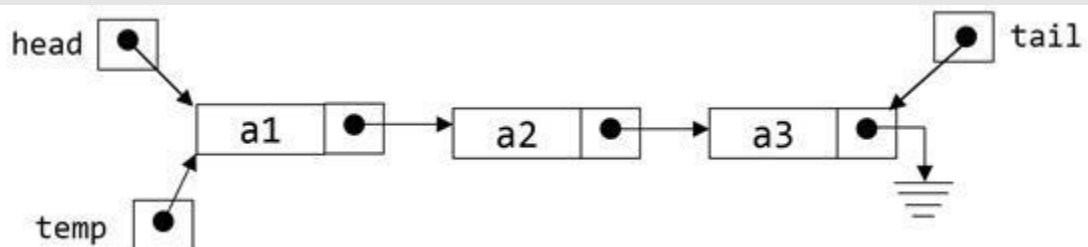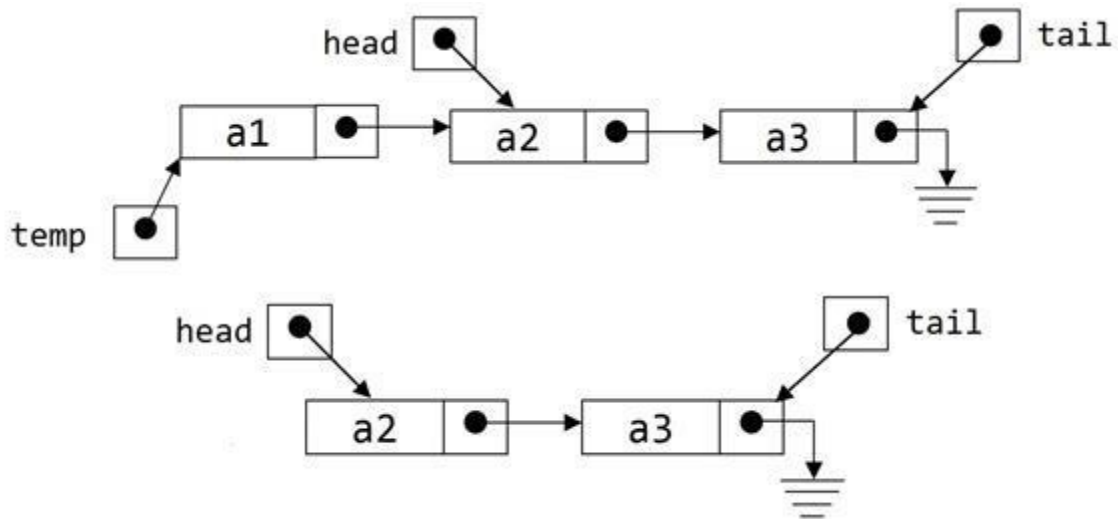
In this we need the tail reference, as it may be the case, there was only one element in the list and the tail reference will also be modified in case of the removal.

Example 8.4:
```
def remove()
    if self.Empty then
        raise StandardError, "QueueEmptyException"
    end
    value = @head.value
    @head = @head.next
    @count -= 1
    return value
end
```

Analysis: Remove operation removes first node from the start of the queue(linked list).

Problems in Queue

Queue using a stack

How to implement a queue using a stack. You can use more than one stack.

Solution: We can use two stack to implement queue.
1. **Enqueue Operation**: new elements are added to the top of first stack.
2. **Dequeue Operation**: elements are popped from the second stack. When second stack is empty then all the elements of first stack are popped one by one and pushed into second stack.

Example 8.5:
```
class QueueUsingStack
    def initialize()
        @stk1 = []
        @stk2 = []
    end
```

```ruby
    def add(value)
        @stk1.push(value)
    end

    def remove()
        if @stk2.size != 0 then
            return @stk2.pop()
        end
        while @stk1.size != 0
            value = @stk1.pop()
            @stk2.push(value)
        end
        return @stk2.pop()
    end
end

# Testing code
que = QueueUsingStack.new()
que.add(1)
puts que.remove()
```

Analysis: All add() happens to stack 1. When remove() is called removal happens from stack 2. When the stack 2 is empty then stack 1 is popped and pushed into stack 2. This popping from stack 1 and pushing into stack 2 revert the order of retrieval there by making queue behaviour out of two stacks.

Stack using a Queue

Implement stack using a queue.

Solution 1: use two queues
Push: add new elements to queue1.
Pop: while size of queue1 is bigger than 1. Push all items from queue 1 to queue 2 except the last item. Switch the name of queue 1 and queue 2. Then return the last item.
Push operation is **O(1)** and Pop operation is **O(n)**

Solution 2: This same can be done using just one queue.
Push: add the element to queue.
Pop: find the size of queue. If size is zero then return error. Else, if size is positive then remove size- 1 elements from the queue and again add to the same queue. At last, remove the next element and return it.
Push operation is **O(1)** and Pop operation is **O(n)**

Solution 3: In the above solutions the push is efficient and pop is inefficient can we make pop efficient **O(1)** and push inefficient **O(n)**
Push: add new elements to queue2. Then add all the elements of queue 1 to queue 2. Then switch names of queue1 and queue 2.
Pop: remove from queue1

Reverse a stack

Reverse a stack using queue
Solution 1:
- Pop all the elements of stack and add them into a queue.
- Then remove all the elements of the queue into stack
- We have the elements of the stack reversed.

Solution 2:
- Since dynamic list or [ ] list is used to implement stack in Ruby, we can iterate from both the directions of the list and swap the elements.

Reverse a queue

Reverse a queue-using stack
Solution:
- Dequeue all the elements of the queue into stack (append to the Ruby list [ ])
- Then pop all the elements of stack and add them into a queue. (pop the elements from the list)
- We have the elements of the queue reversed.

Breadth-First Search with a Queue

In breadth-first search, we explore all the nearest nodes first by finding all possible successors and add them to a queue.
- Create a queue
- Create a start point
- Enqueue the start point onto the queue
- while (value searching not found and the queue is not empty)
    - o Dequeue from the queue
    - o Find all possible points after the last one tried
    - o Enqueue these points onto the queue

Josephus problem

There are n people standing in a queue waiting to be executed. The counting begins at the front of the queue. In each step, k number of people are removed and again added one by one from the queue. Then the next person is executed. The execution proceeds around the circle until only the last person remains, who will be freedom.

Find that position where you want to stand and gain your freedom.
Solution:
- Just insert integer for 1 to k in a queue. (corresponds to k people)
- Define a Kpop() function such that it will remove and add the queue k-1 times and then remove one more time. (This man is dead.)
- Repeat second step until size of queue is 1.

· Print the value in the last element. This is the solution.

Exercise

1. Implement queue using dynamic memory allocation, such that the implementation should follow the following constraints.
    a. The user should use memory allocation from the heap using new operator. In this, you need to take care of the max value in the queue.
    b. Once you are done with the above exercise and you are able to test your queue. Then you can add some more complexity to your code. In add() function when the queue is full, in place of printing, "Queue is full" you should allocate more space using new operator.
    c. Once you are done with the above exercise. Now in remove function once you are below half of the capacity of the queue, you need to decrease the size of the queue by half. You should add one more variable "min" to queue so that you can track what is the original value capacity passed at initialization() function. Moreover, the capacity of the queue will not go below the value passed in the initialization.

    (If you are not able to solve the above exercise, then have a look into stack chapter, where we have done similar problems for stack)

2. Implement the below function for the queue:
    d. IsEmpty: This is left as an exercise for the user. Take a variable, which will take care of the size of a queue if the value of that variable is zero, isEmpty should return 1 (true). If the queue is not empty, then it should return 0 (false).
    e. Size: Use the size variable to be used under size function call. Size() function should return the number of elements in the queue.

3. Implement stack using a queue. Write a program for this problem. You can use just one queue.

4. Write a program to Reverse a stack using queue

5. Write a program to Reverse a queue using stack

6. Write a program to solve Josephus problem (algorithm already discussed.). There are n people standing in a queue waiting to be executed. The counting begins at the front of the queue. In each step, k number of people are removed and again added one by one from the queue. Then the next person is executed. The elimination proceeds around the circle until only the last person remains, who will be given freedom. Find that position where you want to stand and gain your freedom.

7. Write a CompStack() function which takes reference to two stack as an argument and return true or false depending upon whether all the elements of the stack are equal or not. You are given isEqual(int, int) which will compare and return 1 if both values are equal and 0 if they are different.

CHAPTER 9: TREE

Introduction

We have already read about various linear data structures like a list, linked list, stack, queue etc. Both list and linked list have a drawback of linear time required for searching an element.

A tree is a nonlinear data structure, which is used to represent hierarchical relationships (parent-child relationship). Each node is connected by another node by directed edges.

Example 1: Tree in organization
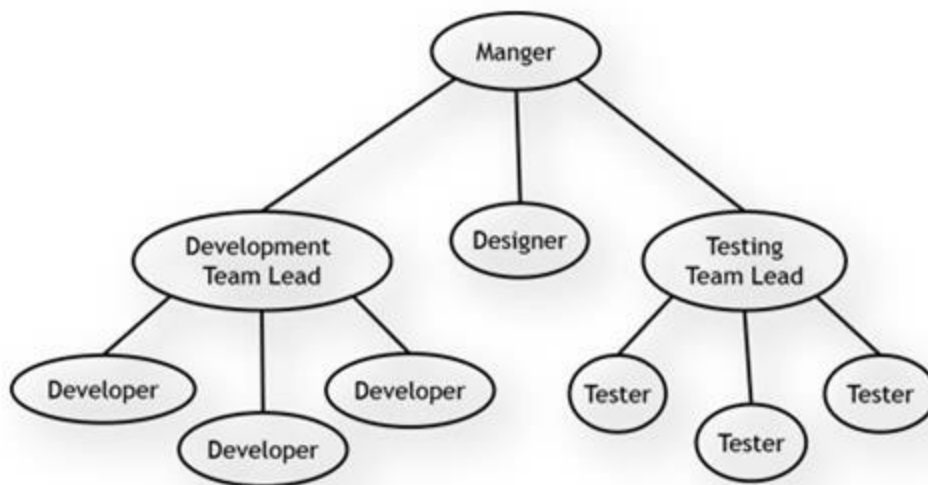
Example 2: Tree in a file system
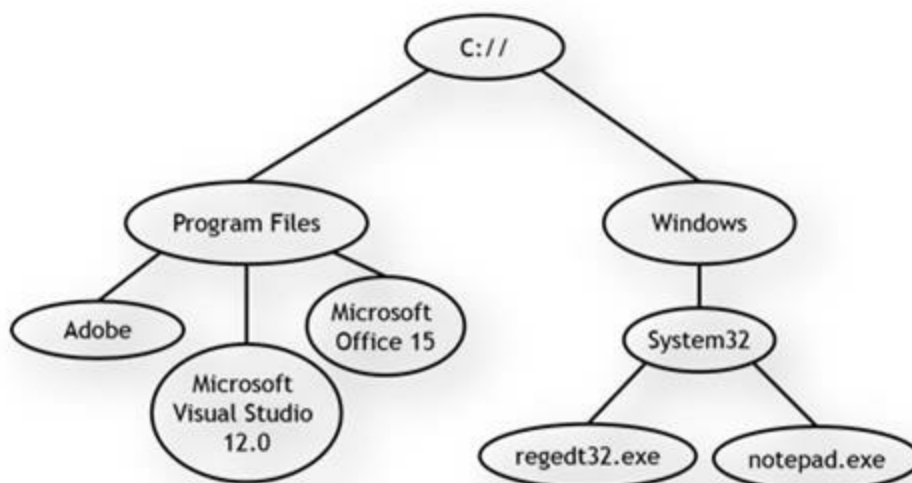
Terminology in tree
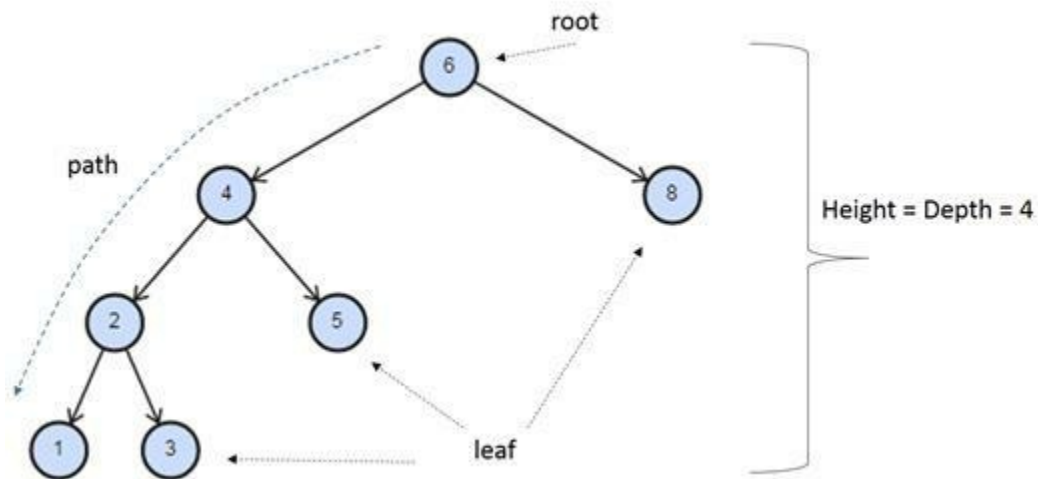
Root: The root of the tree is the only node with no incoming edges. It is the top node of a tree.

Node: It is a fundamental element of a tree. Each node has data and two references that may point to null or its children

Edge: It is also a fundamental part of a tree, which is used to connect two nodes.

Path: A path is an ordered list of nodes that are connected by edges.

Leaf: A leaf node is a node that has no children.

Height of the tree: The height of a tree is the number of edges on the longest path between the root and a leaf.

The level of node: The level of a node is the number of edges on the path from the root node to that node.
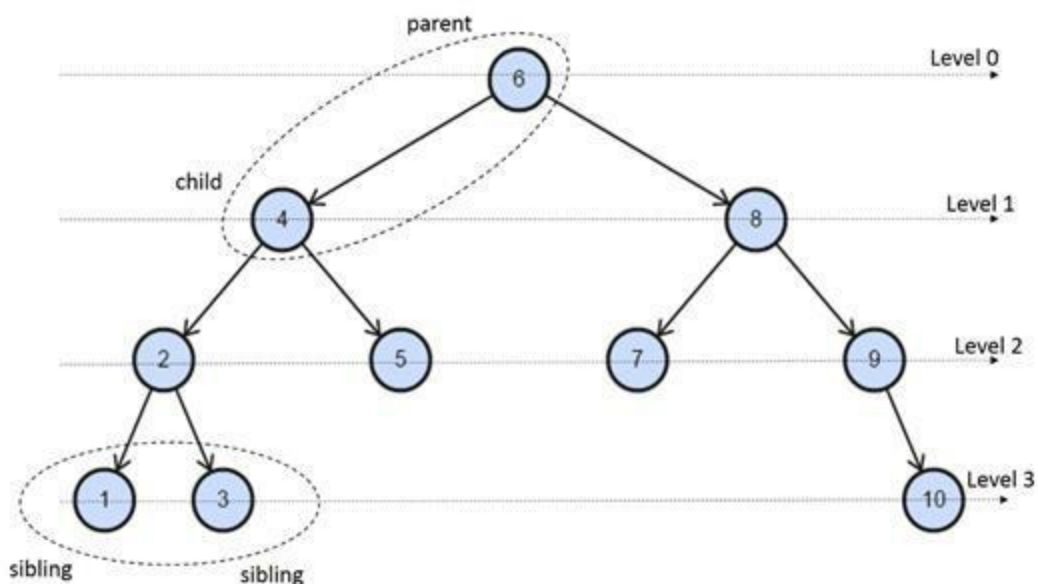
Children: Nodes that have incoming edges from the same node is said to be the children of that node.

Parent: Node is a parent of all the child nodes that are linked by outgoing edges.

Sibling: Nodes in the tree that are children of the same parent are called siblings.

Ancestor: A node reachable through repeated moving from child to parent.

Binary Tree

A binary tree is a type tree in which each node has at most two children (0, 1 or 2), which are referred to as the left child and the right child.

Below is a node of the binary tree with "a" stored as data and whose left child (lChild) and whose right child (rchild) both are pointing towards null.
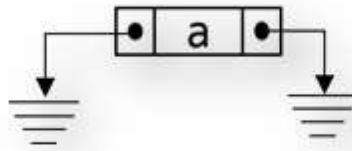
Below is a class definition used to define node.

```ruby
class Tree
    class Node
        attr_accessor :value, :lChild, :rChild
        def initialize(v, left = nil, right = nil)
            @value = v
            @lChild = left
            @rChild = right
        end
    end

    attr_accessor :root

    def initialize()
        @root = nil
    end

    # Other methods.
end
```

Below is a binary tree whose nodes contains data from 1 to 10

6,4,2,5,1,3,8,7,9,10
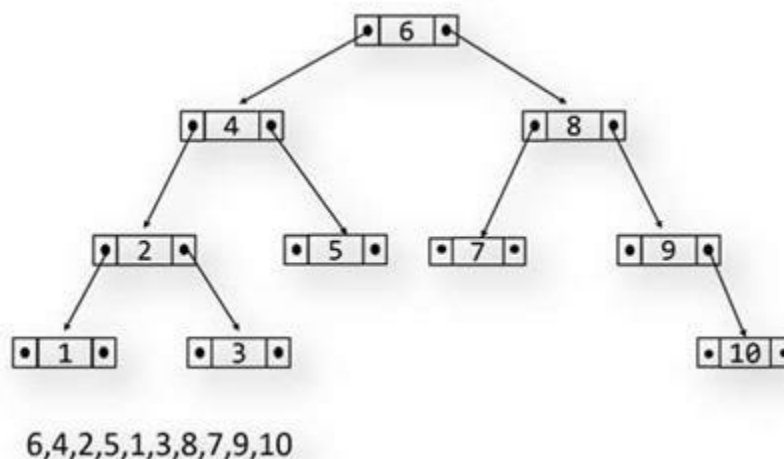
In the rest of the book, binary tree will be represented as below:
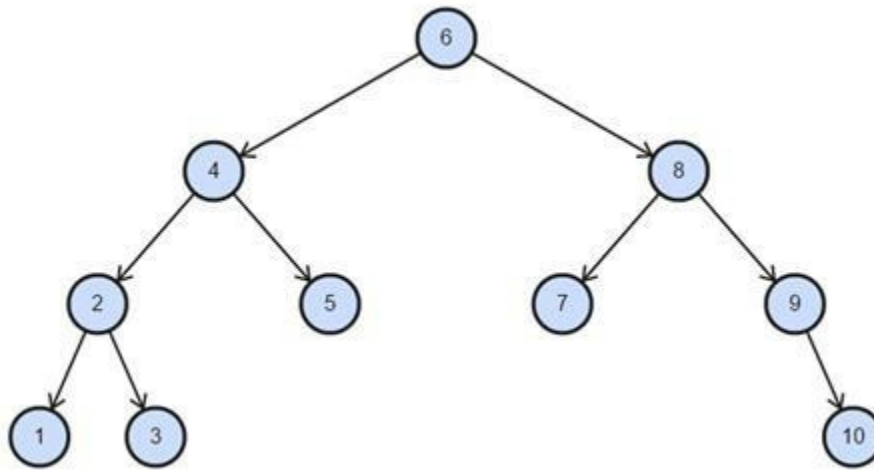
Properties of Binary tree are:

1. The maximum number of nodes on level i of a binary tree is 2^i , where i >= 1
2. The maximum number of nodes in a binary tree of depth k is 2^{k+1}, where k >= 1
3. There is exactly one path from the root to any nodes in a tree.
4. A tree with N nodes have exactly N-1 edges connecting these nodes.
5. The height of a complete binary tree of N nodes is $\log_2 N$.

Types of Binary trees

Complete binary tree

In a complete binary tree, every level except the last one is completely filled. All nodes in the left are filled first, then the right one. A binary heap is an example of a complete binary tree.
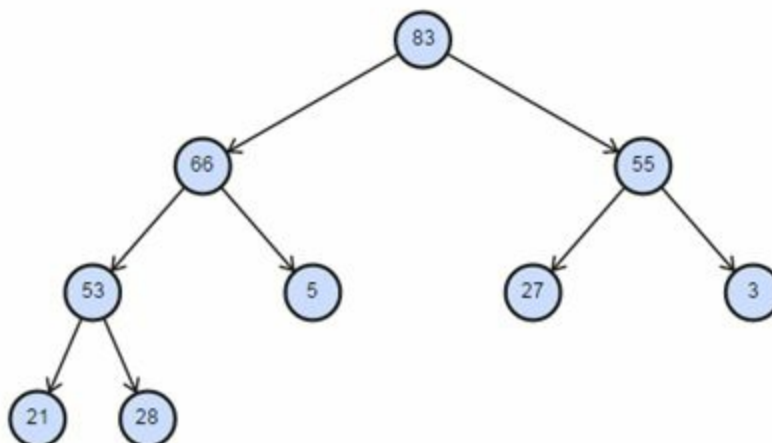
Full/ Strictly binary tree

The full binary tree is a binary tree in which each node has exactly zero or two children.
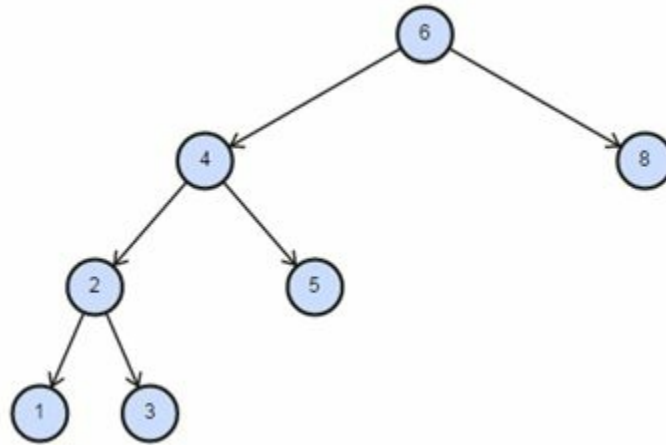
Perfect binary tree

The perfect binary tree is a type of full binary tree in which each non-leaf node has exactly two child nodes. All leaf nodes have identical path length and all possible node slots are occupied
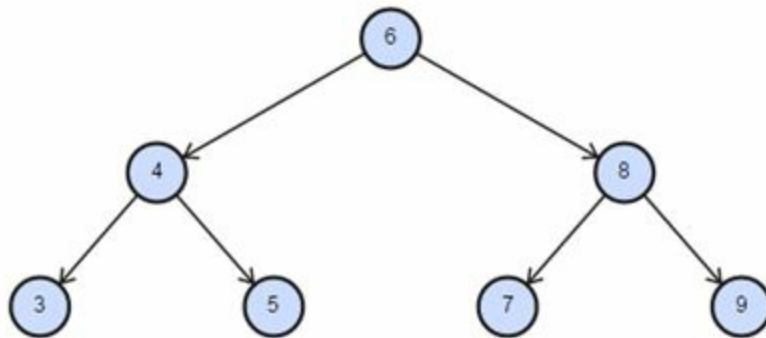
Right skewed binary tree

A binary tree in which either each node is has a right child or no child (leaf) is called as right skewed binary tree
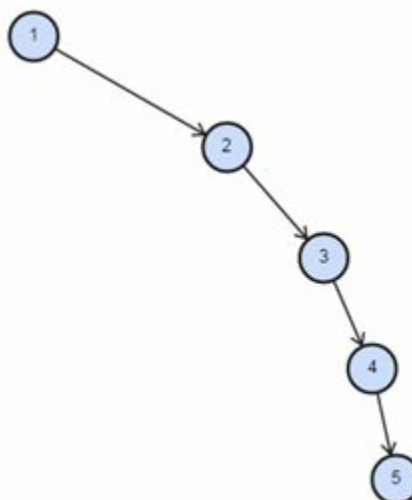
Left skewed binary tree

A binary tree in which either each node is has a left child or no child (leaf) is called as Left skewed binary tree
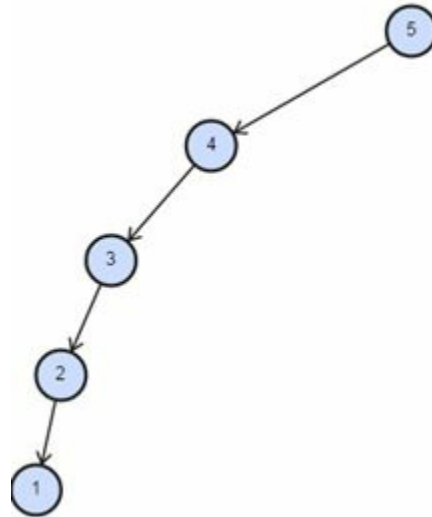
Height-balanced Binary Tree

A height-balanced binary tree is a binary tree such that the left & right subtrees for any given node differs in height by max one. AVL tree and RB tree are an example of height balanced tree we will discuss these trees later in this chapter.

Note: Each complete binary tree is a height-balanced binary tree
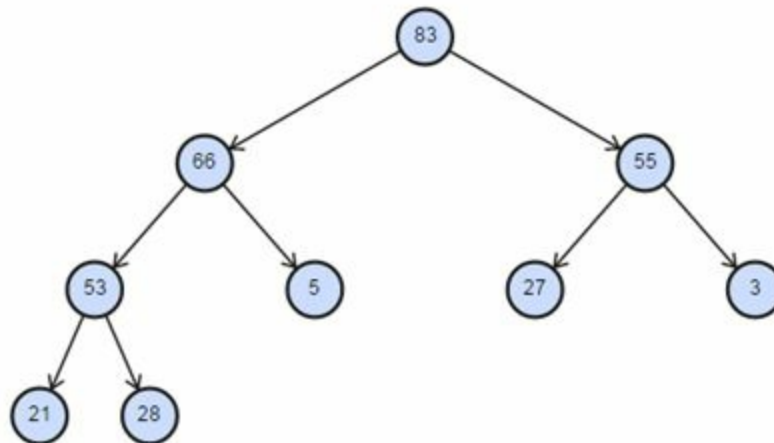
Problems in Binary Tree

Create a Complete binary tree

Create a binary tree given a list of values.

Solution: Since there is no order defined in a binary tree, so nodes can be inserted in any order so it can be a skewed binary tree. But it is inefficient to do anything in a skewed binary tree so we will create a Complete binary tree. At each node, the middle value stored in the array is assigned to node. The left of array is passed to the left child of the node to create left sub-tree and the right portion of array is passed to right child of the node to create right sub-tree.

Example 9.1:

```
def levelOrderBinaryTree(arr)
    @root = self.levelOrderBinaryTreeUtil(arr, 0)
end

def levelOrderBinaryTreeUtil(arr, start)
    size = arr.size
    curr = Node.new(arr[start])
    left = 2 * start + 1
    right = 2 * start + 2
    if left < size
        curr.lChild = self.levelOrderBinaryTreeUtil(arr, left)
    end
    if right < size
        curr.rChild = self.levelOrderBinaryTreeUtil(arr, right)
    end
    return curr
end

# Testing code
arr = [6, 4, 8, 2, 5, 7, 9, 1, 3]
t = Tree.new()
t.levelOrderBinaryTree(arr)
```

Complexity Analysis: This is an efficient algorithm for creating a complete binary tree.
Time Complexity: O(n), Space Complexity: O(n)

Pre-Order Traversal

Traversal is a process of visiting each node of a tree. In Pre-Order Traversal parent is visited / traversed first, then left child and then right child. Pre-Order traversal is a type of depth-first traversal.
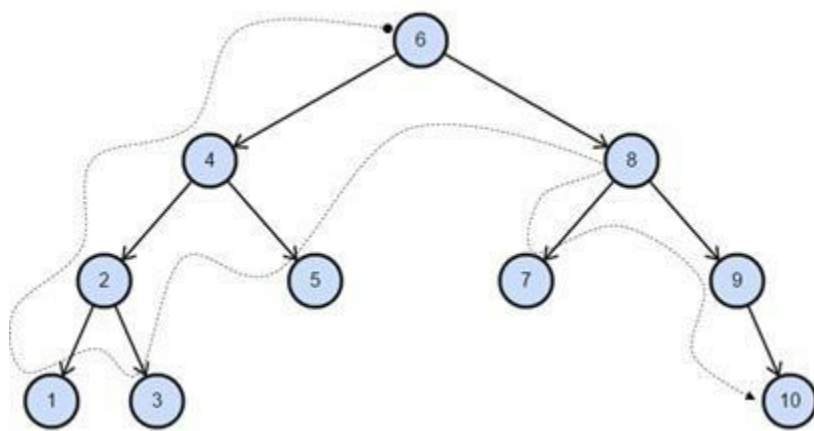
Solution: Preorder traversal is done using recursion. At each node, first the value stored in it is printed and then followed by the value of left child and right child. At each node its value is printed followed by calling printTree() function to its left and right child to print left and right sub-tree.

Example 9.2:

```
def PrintPreOrder(node = @root)
   if node != nil
      print node.value, " "
      self.PrintPreOrder(node.lChild)
      self.PrintPreOrder(node.rChild)
   end
end
```

Output:

6 4 2 1 3 5 8 7 9 10

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Note: When there is an algorithm in which all nodes are traversed then complexity cannot be less than O(n). When there is a large portion of the tree, which is not traversed, then complexity reduces.

Post-Order Traversal

In Post-Order Traversal left child is visited / traversed first, then right child and then parent Post-Order traversal is a type of depth-first traversal.
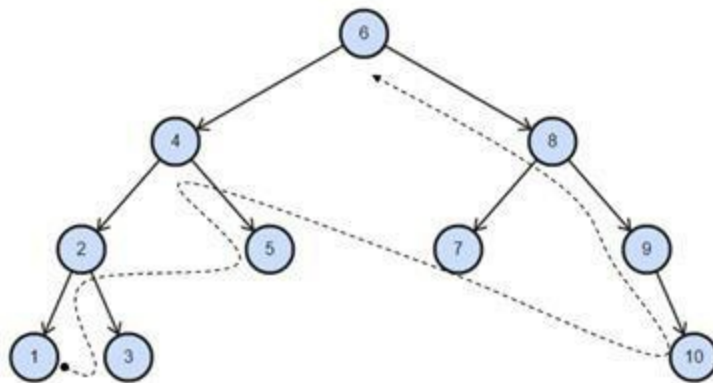
Solution: At each node, first the left child is traversed then right child and in the end, current node value is printed to the screen.

Example 9.3:

```
def PrintPostOrder(node = @root)
   if node != nil
      self.PrintPostOrder(node.lChild)
      self.PrintPostOrder(node.rChild)
      print node.value, " "
   end
end
```

Output:

1 3 2 5 4 7 10 9 8 6

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

In-Order Traversal

In In-Order Traversal, left child is visited / traversed first, then the parent value is printed and last right child is traversed.
In-Order traversal is a type of depth-first traversal. The output of In-Order traversal of BST is a sorted list.

Solution: In In-Order traversal first, the value of left child is traversed, then the value of node is printed to the screen and then the value of right child is traversed.
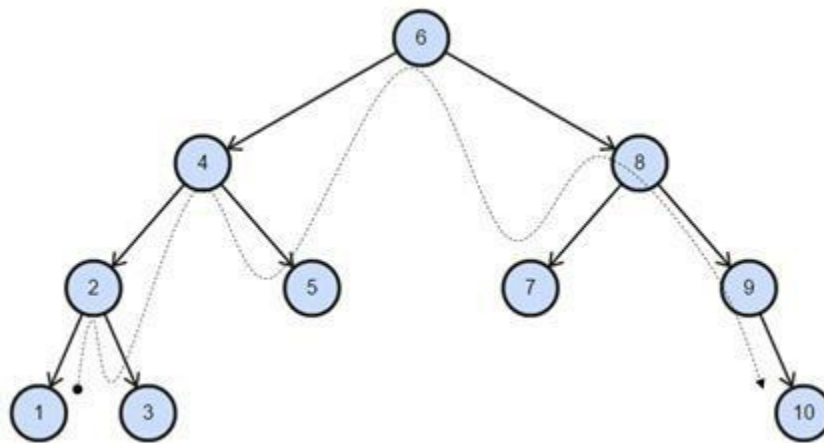
Example 9.4:
```
def PrintInOrder(node = @root)
    if node != nil
        self.PrintInOrder(node.lChild)
        print node.value, " "
        self.PrintInOrder(node.rChild)
    end
end
```

Output:
```
1 2 3 4 5 6 7 8 9 10
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Note: Pre-Order, Post-Order, and In-Order traversal are meant for all binary trees. They can be used to traverse any kind of a binary tree.

Level order traversal / Breadth First traversal

Write code to implement level order traversal of a tree. Such that nodes at depth k is printed before nodes at depth k+1.
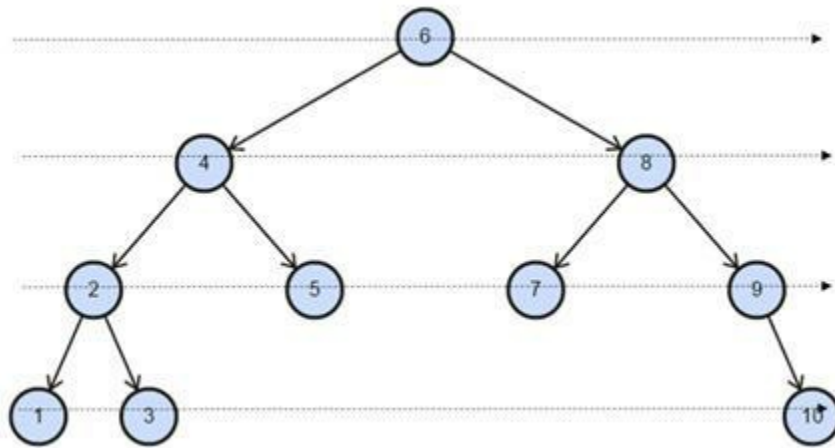
Solution: Level order traversal or Breadth First traversal of a tree is done using a queue. At first, the root node reference is added to a queue. The traversal of tree is done until the queue is empty. When we traverse the tree, we first remove an element from the queue, print the value stored in that node and then its left child and right child will be added to the queue.

Example 9.5:

```
def PrintBredthFirst()
    que = Queue.new()
    if @root != nil
        que.push(@root)
    end
    while que.size != 0
        temp = que.pop()
        print temp.value, " "
        if temp.lChild != nil
            que.push(temp.lChild)
        end
        if temp.rChild != nil
            que.push(temp.rChild)
        end
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Print Depth First without using the recursion / system stack.

Solution: Depth first traversal of the tree is done using recursion by using system stack. The same can be done using stack. In the beginning, root node reference is added to the stack. The whole tree is traversed until the stack is empty. In each iteration, an element is popped from the stack, its value is printed to screen. Then right child and then left child of the node is added to stack.

Example 9.6:

```
def PrintDepthFirst()
    stk = []
```

```
        if @root != nil
            stk.push(@root)
        end
        while stk.size != 0
            temp = stk.pop()
            print temp.value, " "
            if temp.rChild != nil
                stk.push(temp.rChild)
            end
            if temp.lChild != nil
                stk.push(temp.lChild)
            end
        end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Tree Depth

Solution: Depth of tree is calculated recursively by traversing left and right child of the root. At each level of traversal depth of both left & right child is calculated. The greater depth among the left and right child is added by one (which is the depth of the current node) and this value is returned.

Example 9.7:
```
def TreeDepth(curr = @root)
    if curr == nil
        return 0
    else
        lDepth = self.TreeDepth(curr.lChild)
        rDepth = self.TreeDepth(curr.rChild)
        if lDepth > rDepth
            return lDepth + 1
        else
            return rDepth + 1
        end
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth Pre-Order

Solution: We want to print the node, which will be at the nth index when we print the tree in PreOrder traversal. Therefore, we keep a counter to keep track of the index. When the counter is equal to index, then we print the value and return the Nth preorder index node.

Example 9.8:

```
def NthPreOrder(index)
   counter = [0]
   self.NthPreOrderUtil(@root, index, counter)
end
def NthPreOrderUtil(node, index, counter)
   if node != nil
      counter[0] += 1
      if counter[0] == index
         print node.value
      end
      self.NthPreOrderUtil(node.lChild, index, counter)
      self.NthPreOrderUtil(node.rChild, index, counter)
   end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth Post Order

Solution: We want to print the node that will be at the nth index when we print the tree in post order traversal. Therefore, we keep a counter to keep track of the index, but at this time, we will increment the counter after left child and right child traversal. When the counter is equal to index, then we print the value and return the nth post-order index node.

Example 9.9
```
def NthPostOrder(index)
   counter = [0]
   self.NthPostOrderUtil(@root, index, counter)
end
def NthPostOrderUtil(node, index, counter)
   if node != nil
      self.NthPostOrderUtil(node.lChild, index, counter)
      self.NthPostOrderUtil(node.rChild, index, counter)
      counter[0] +=  1
      if counter[0] == index
         print node.value
      end
   end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Nth In Order

Solution: We want to print the node that will be at the nth index when we print the tree in in-order traversal. Therefore, we keep a counter to keep track of the index, but at this time, we will increment the counter after left child traversal but before the right child traversal. When the counter is equal to index, then we print the value and return the nth in-order index node.

Example 9.10:
```
def NthInOrder(index)
    counter = [0]
    self.NthInOrderUtil(@root, index, counter)
end

def NthInOrderUtil(node, index, counter)
    if node != nil
        self.NthInOrderUtil(node.lChild, index, counter)
        counter[0] += 1
        if counter[0] == index
            print node.value
        end
        self.NthInOrderUtil(node.rChild, index, counter)
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Copy Tree

Solution: Copy tree is done by copy nodes of the input tree at each level of the traversal of the tree. At each level of the traversal of nodes of the tree, a new node is created and the value of the input tree node is copied to it. The left child tree is copied recursively and then reference to new subtree is returned which will be assigned to the left child reference of the current new node. Similarly for the right child node too. Finally, the tree is copied.

Example 9.11:
```
def CopyTree()
    tree2 = Tree.new()
    tree2.root = self.CopyTreeUtil(@root)
    return tree2
end

def CopyTreeUtil(curr)
    if curr != nil
        temp = Node.new(curr.value)
        temp.lChild = self.CopyTreeUtil(curr.lChild)
        temp.rChild = self.CopyTreeUtil(curr.rChild)
        return temp
    else
        return nil
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Copy Mirror Tree

Solution: Copy mirror image of the tree is done same as copy tree, but in place of left child pointing to the tree which is formed by left child traversal of input tree. This time left child points to the tree formed by right child traversal of the input tree. Similarly right child points to the tree formed by the traversal of the left child of the input tree.

Example 9.12:
```
def CopyMirrorTree()
    tree2 = Tree.new()
    tree2.root = self.CopyMirrorTreeUtil(@root)
    return tree2
end

def CopyMirrorTreeUtil(curr)
    if curr != nil
        temp = Node.new(curr.value)
        temp.rChild = self.CopyMirrorTreeUtil(curr.lChild)
        temp.lChild = self.CopyMirrorTreeUtil(curr.rChild)
        return temp
    else
        return nil
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Number of Element

Solution: Number of nodes at the right child and the number of nodes at the left child is added by one and we get the total number of nodes in any tree / sub-tree.

Example 9.13:
```
def numNodes(curr = @root)
    if curr == nil
        return 0
    else
        return (1 + self.numNodes(curr.rChild) + self.numNodes(curr.lChild))
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Number of Leaf nodes

Solution: If we add the number of leaf node in the right child with the number of leaf nodes in the left child, we will get the total number of leaf node in any tree or subtree.

Example 9.14:

```
def numLeafNodes(curr = @root)
    if curr == nil
        return 0
    end
    if curr.lChild == nil and curr.rChild == nil
        return 1
    else
        return (self.numLeafNodes(curr.rChild) + self.numLeafNodes(curr.lChild))
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Identical

Solution: Two trees have identical values if at each level the value is equal.

Example 9.15:

```
def isEqual(tree2)
    return self.isEqualUtil(@root, tree2.root)
end

def isEqualUtil(node1, node2)
    if node1 == nil and node2 == nil
        return true
    elsif node1 == nil or node2 == nil
        return false
    else
        return (self.isEqualUtil(node1.lChild, node2.lChild) and self.isEqualUtil(node1.rChild,
node2.rChild) and (node1.value == node2.value))
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Free Tree

Solution: You just need to make the root of the tree point to nil. The system will do garbage collection and recover the memory assigned to the tree. You had done a single act and because of this action, the time complexity is constant.

Example 9.16:

```
def Free()
    @root = nil
end
```

Complexity Analysis: Time Complexity: O(1), Space Complexity: O(1)

Print all the paths

Print all the paths from the roots to the leaf

Solution: Whenever we traverse a node, we add that node to the list. When we reach a leaf, we print the whole list. When we return from a function, then we remove the element that was added to the list when we entered this function.

Example 9.17:
```
def printAllPath()
    stk = []
    self.printAllPathUtil(@root, stk)
end

def printAllPathUtil(curr, stk)
    if curr == nil
        return
    end
    stk.push(curr.value)
    if curr.lChild == nil and curr.rChild == nil
        print stk, "\n"
        stk.pop()
        return
    end
    self.printAllPathUtil(curr.rChild, stk)
    self.printAllPathUtil(curr.lChild, stk)
    stk.pop()
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Least Common Ancestor

Solution: We recursively traverse the nodes of a binary tree. We find any one of the input nodes for which we are searching a common ancestor then we return that node. When we get both the left and right as some valid reference location other than null, we will return that node as the common ancestor.

Example 9.18:
```
def LCA(first, second)
    ans = self.LCAUtil(@root, first, second)
    if ans != nil
        return ans.value
    else
        return -1000000
    end
end
```

```
def LCAUtil(curr, first, second)
    if curr == nil
        return nil
    end
    if curr.value == first or curr.value == second
        return curr
    end
    left = self.LCAUtil(curr.lChild, first, second)
    right = self.LCAUtil(curr.rChild, first, second)
    if left != nil and right != nil
        return curr
    elsif left != nil
        return left
    else
        return right
    end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Find Max in Binary Tree

Solution: We recursively traverse the nodes of a binary tree. We will find the maximum value in the left and right subtree of any node then will compare the value with the value of the current node and finally return the largest of the three values.

Example 9.19:
```
def findMaxBT(curr = @root)
    if curr == nil
        return -1000000
    end
    max = curr.value
    left = self.findMaxBT(curr.lChild)
    right = self.findMaxBT(curr.rChild)
    if left > max
        max = left
    end
    if right > max
        max = right
    end
    return max
end
```

Search value in a Binary Tree

Solution: To find if some value is there in a binary tree or not it is done using exhaustive search of the binary tree. First, the value of current node is compared with the value, which we are

looking for. Then it is compared recursively inside the left child and right child.

Example 9.20:
```
def searchBT(value, curr = @root)
    if curr == nil
        return false
    end
    if curr.value == value
        return true
    end
    left = self.searchBT( value, curr.lChild)
    if left
        return true
    end
    right = self.searchBT( value, curr.rChild)
    if right
        return true
    end
    return false
end
```

Maximum Depth in a Binary Tree

Solution: To find the maximum depth of a binary tree we need to find the depth of the left tree and depth of right tree then we need to store the value and increment it by one so that we get depth of the given node.

Example 9.21:
```
def TreeDepth(curr = @root)
    if curr == nil
        return 0
    else
        lDepth = self.TreeDepth(curr.lChild)
        rDepth = self.TreeDepth(curr.rChild)
        if lDepth > rDepth
            return lDepth + 1
        else
            return rDepth + 1
        end
    end
end
```

Number of Full Nodes in a BT

Solution: A full node is a node that has both left and right child. We will recursively traverse the whole tree and will increase the count of full node as we find them.

Example 9.22:

```
def numFullNodesBT(curr = @root)
    if curr == nil
        return 0
    end
    count = self.numFullNodesBT(curr.rChild) + self.numFullNodesBT(curr.lChild)
    if curr.rChild != nil and curr.lChild != nil
        count += 1
    end
    return count
end
```

Maximum Length Path in a BT/ Diameter of BT

Solution: To find the diameter of BT we need to find the depth of left child and right child then will add these two values and increment it by one so that we will get the maximum length path (diameter candidate) which contains the current node. Then we will find max length path in the left child sub-tree. We will also find the max length path in the right child sub-tree. Finally, we will compare the three values and return the maximum value out of these, this will be the diameter of the Binary tree.

Example 9.23:
```
def maxLengthPathBT(curr = @root) # diameter
    if curr == nil
        return 0
    end
    leftPath = self.TreeDepth(curr.lChild)
    rightPath = self.TreeDepth(curr.rChild)
    max = leftPath + rightPath + 1
    leftMax = self.maxLengthPathBT(curr.lChild)
    rightMax = self.maxLengthPathBT(curr.rChild)
    if leftMax > max
        max = leftMax
    end
    if rightMax > max
        max = rightMax
    end
    return max
end
```

Sum of All nodes in a BT

Solution: We will find the sum of all the nodes recursively. sumAllBT() will return the sum of all the node of left and right subtree then we will add the value of current node and will return the final sum.

Example 9.24:
```
def sumAllBT(curr = @root)
    if curr == nil
```

```
      return 0
   end
   rightSum = self.sumAllBT(curr.rChild)
   leftSum = self.sumAllBT(curr.lChild)
   sum = rightSum + leftSum + curr.value
   return sum
end
```

Iterative Pre-order

Solution: In place of using system stack in recursion, we can traverse the tree using stack data structure.

Example 9.25:
```
def iterativePreOrder()
   stk = []
   if @root != nil
      stk.push(@root)
   end
   while stk.size != 0
      curr = stk.pop()
      print curr.value , " "
      if curr.rChild != nil
         stk.push(curr.rChild)
      end
      if curr.lChild != nil
         stk.push(curr.lChild)
      end
   end
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Binary Search Tree (BST)

A binary search tree (BST) is a binary tree on which nodes are ordered in the following way:
·   The key in the left subtree is less than the key in its parent node.
·   The key in the right subtree is greater the key in its parent node.
·   No duplicate key is allowed.

Note: there can be two separate key and value fields in the tree node. But for simplicity, we are considering value as the key. All problems in the binary search tree are solved using this supposition that the value in the node is key for the tree.

Note: Since binary search tree is a binary tree. So all the above algorithm of a binary tree are applicable to a binary search tree.
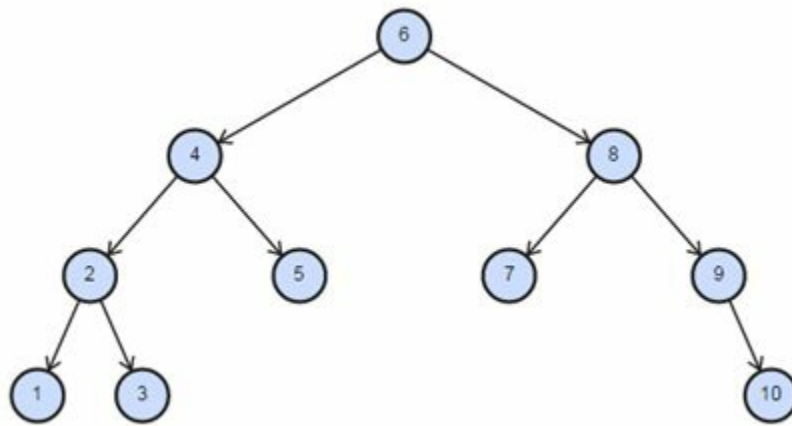
Problems in Binary Search Tree (BST)

All binary tree algorithms are valid for binary search tree too.

Create a binary search tree from sorted list

Create a binary tree from list of values in sorted order. Since the elements in the list are in sorted order and we want to create a binary search tree in which left subtree nodes are having values less than the current node and right subtree nodes have value greater than the value of the current node.

Solution: We have to find the middle node to create a current node and send the rest of the list to construct left and right subtree.

Example 9.26:

```
def CreateBinaryTree(arr)
    @root = self.CreateBinaryTreeUtil(arr, 0, arr.size - 1)
end

def CreateBinaryTreeUtil(arr, start, end2)
    curr = nil
    if start > end2
        return nil
    end
    mid = (start + end2) / 2
    curr = Node.new(arr[mid])
    curr.lChild = self.CreateBinaryTreeUtil(arr, start, mid - 1)
    curr.rChild = self.CreateBinaryTreeUtil(arr, mid + 1, end2)
    return curr
end

# Testing code
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
t2 = Tree()
t2.CreateBinaryTree(arr)
```

Insertion

Nodes with key 6,4,2,5,1,3,8,7,9,10 are inserted in a tree. Given below is a step by step tree after inserting nodes in the order.
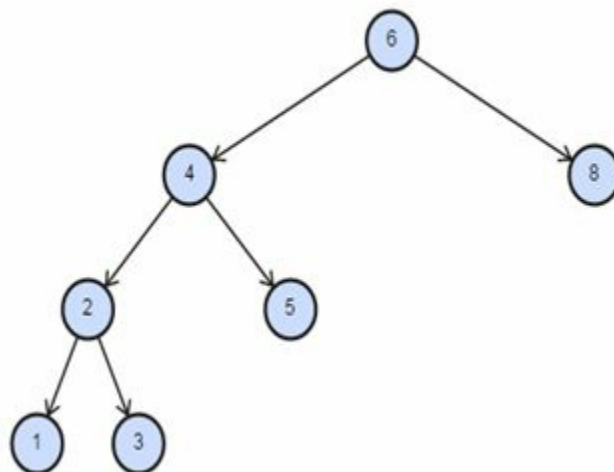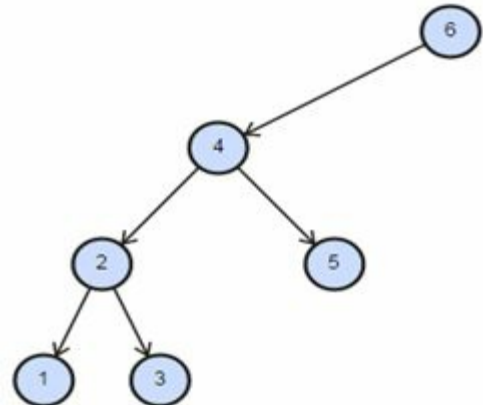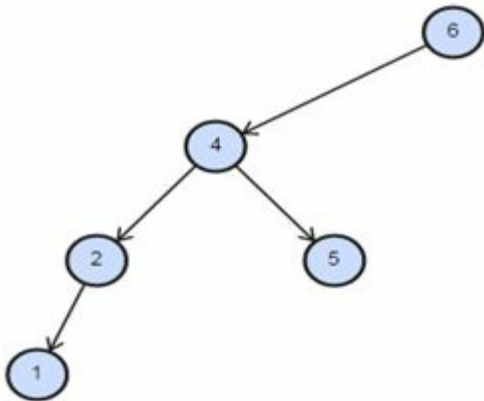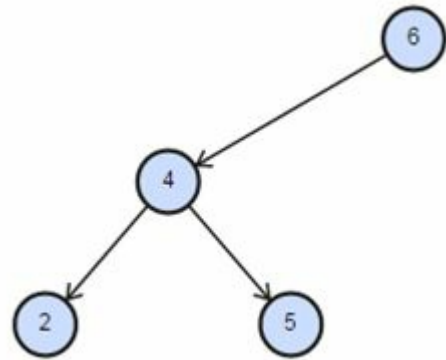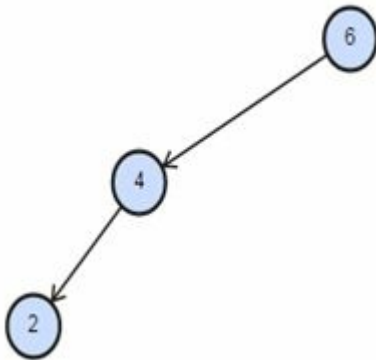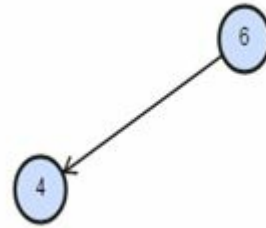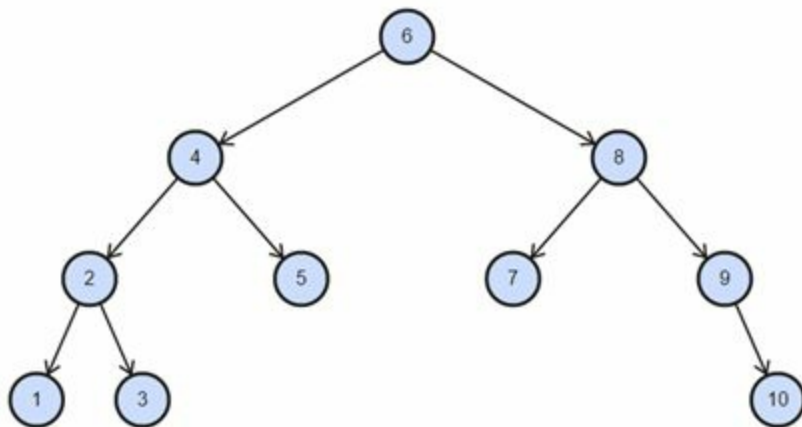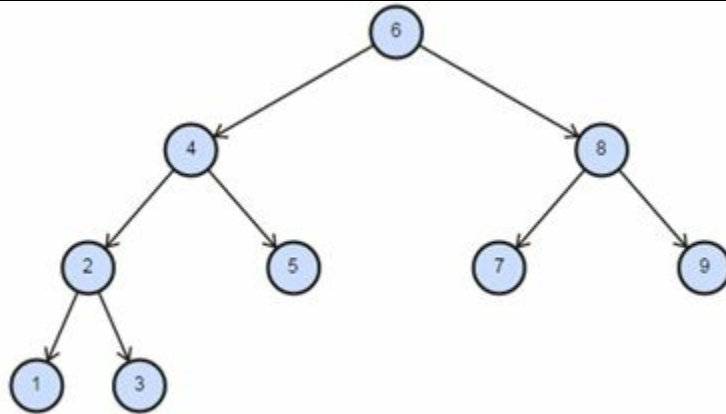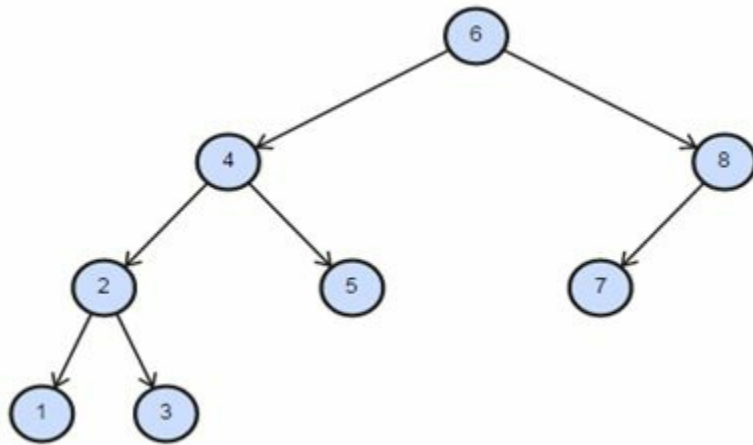
Solution: Smaller values will be added to the left child sub-tree of a node and greater value will be added to the right child sub-tree of the current node.

Example 9.27:
```ruby
def InsertNode(value)
    @root = self.InsertNodeUtil(value, @root)
end

def InsertNodeUtil(value, node)
    if node == nil
        node = Node.new(value, nil, nil)
    else
        if node.value > value
            node.lChild = self.InsertNodeUtil(value, node.lChild)
        else
```

```
            node.rChild = self.InsertNodeUtil(value, node.rChild)
        end
    end
    return node
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

Find Node

Solution: The value greater than the current node value will be in the right child sub-tree and the value smaller than the current node is there in the left child sub-tree. We can find a value by traversing the left or right subtree iteratively.

Example 9.28: Find the node with the value given.
```
def Find(value)
    curr = @root
    while curr != nil
        if curr.value == value
            return true
        elsif curr.value > value
            curr = curr.lChild
        else
            curr = curr.rChild
        end
    end
    return false
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Find Min

Find the node with the minimum value.
Solution: left most child of the tree will be the node with the minimum value.
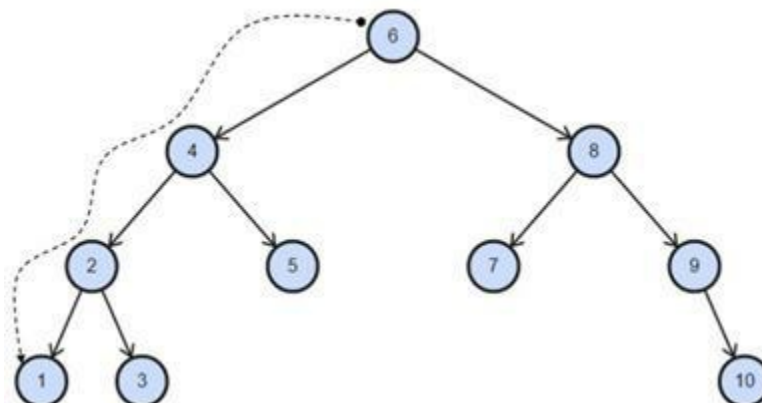
Example 9.29:

```
def FindMin(node = @root)
    if node == nil
        return 1000000
    end
    while node.lChild != nil
        node = node.lChild
    end
    return node.value
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Find Max

Find the node in the tree with the maximum value.
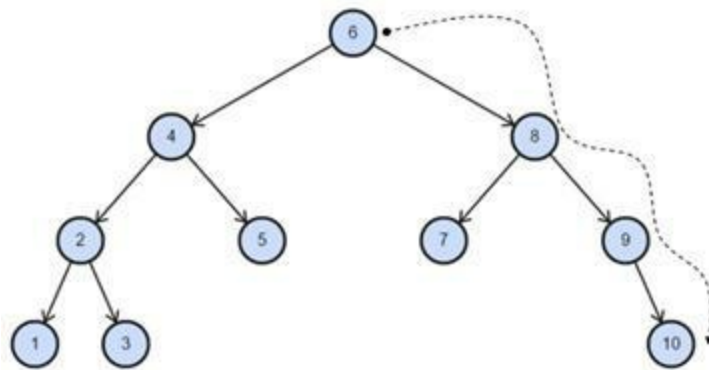
Solution: Right most node of the tree will be the node with the maximum value.

Example 9.30:
```
def FindMax(node = @root)
    if node == nil
        return -1000000
    end
    while node.rChild != nil
        node = node.rChild
    end
    return node.value
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(1)

Is tree a BST

Approach 1: At each node we check whether, max value of left subtree is smaller than the value of current node and min value of right subtree is greater than the current node or not.

Example 9.31:
```
def isBST3(curr = @root)
    if curr == nil
```

```
        return true
    end
    if curr.lChild != nil and self.FindMax(curr.lChild).value > curr.value
        return false
    end
    if curr.rChild != nil and self.FindMin(curr.rChild).value < curr.value
        return false
    end
    return (self.isBST3(curr.lChild) and self.isBST3(curr.rChild))
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n)

The above solution is correct but it is not efficient, as same tree nodes are traversed many times.

Approach 2: A better solution will be the one in which we will look into each node only once. This is done by narrowing the range. We will be use an isBSTUtil() function which takes the max and min range of the values of the nodes. The initial value of min and max will be INT_MIN and INT_MAX.

Example 9.32:
```
def isBst()
    return self.isBSTUtil(@root, -1000000, 1000000)
end

def isBSTUtil(curr, min, max)
    if curr == nil
        return true
    end
    if curr.value < min or curr.value > max
        return false
    end
    return self.isBSTUtil(curr.lChild, min, curr.value) &&        self.isBSTUtil(curr.rChild,
curr.value, max)
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n) for stack

Approach 3: Above method is correct and efficient but there is an easy method to do the same. We can do in-order traversal of nodes and see if we are getting a strictly increasing sequence

Example 9.33:
```
def isBST2()
    c = [-10000000]
    return self.isBST2Util(@root, c)
end

def isBST2Util(curr, count) #  in order  traversal
```

```
    if curr != nil
        ret = self.isBST2Util(curr.lChild, count)
        if not ret
            return false
        end
        if count[0] > curr.value
            return false
        end
        count[0] = curr.value
        ret = self.isBST2Util(curr.rChild, count)
        if not ret
            return false
        end
    end
    return true
end
```

Complexity Analysis: Time Complexity: O(n), Space Complexity: O(n) for stack

Delete Node

Description: Remove the node x from the binary search tree, reorganize nodes of binary search tree to maintain its necessary properties.

There are three cases in delete node, let us call the node that need to be deleted as x.
Case 1: node x has no children. Just delete it (i.e. Change parent node so that it does not point to x)
Case 2: node x has one child. Splice out x by linking x's parent to x's child
Case 3: node x has two children. Splice out the x's successor and replace x with x's successor

When the node to be deleted has no children
This is a trivial case, in which we directly delete the node and return null.

When the node to be deleted has only one child.
In this case, we save the child in a temp variable, then delete current node, and finally return the child.

We want to remove node with value 9. The node has only one child.
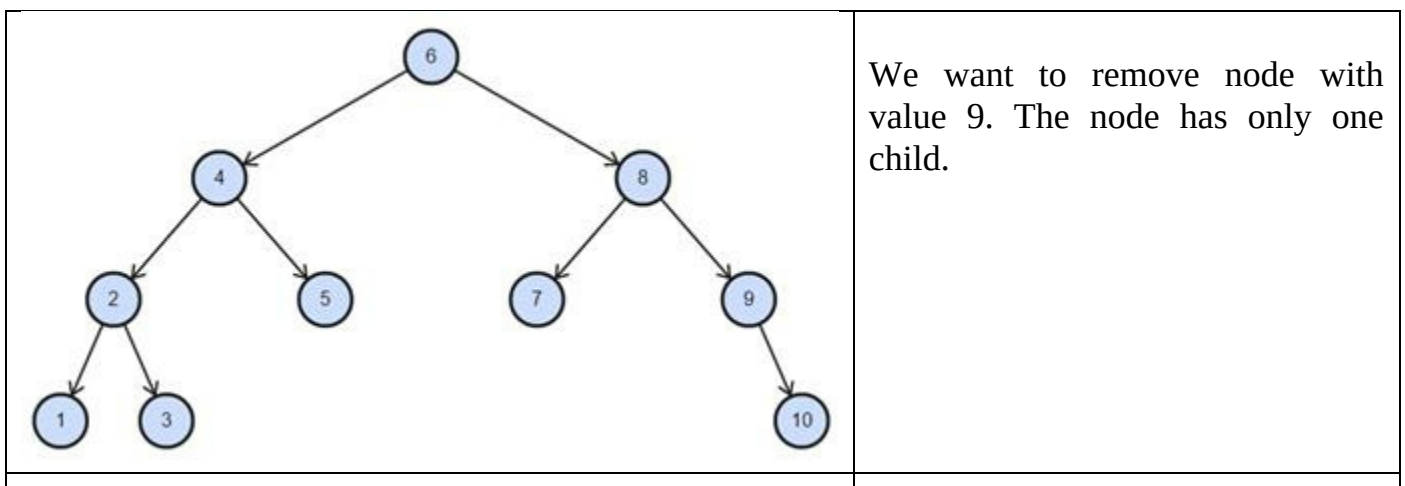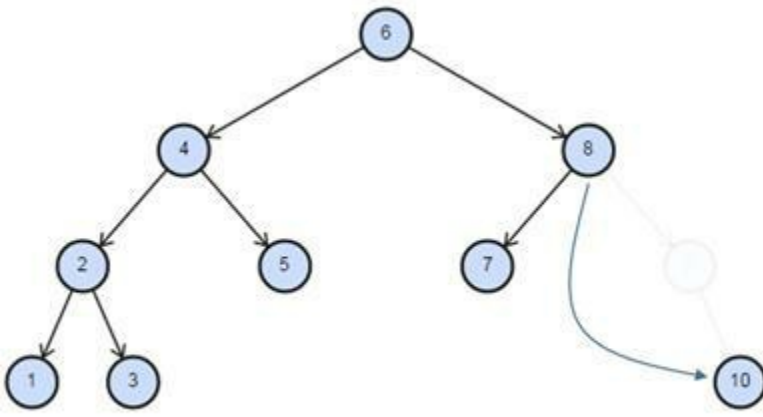
	Right child of the parent of node with value 9 that is the node with value 8 will point to the node with value 10.
	Finally, node with value 9 is removed from the tree.
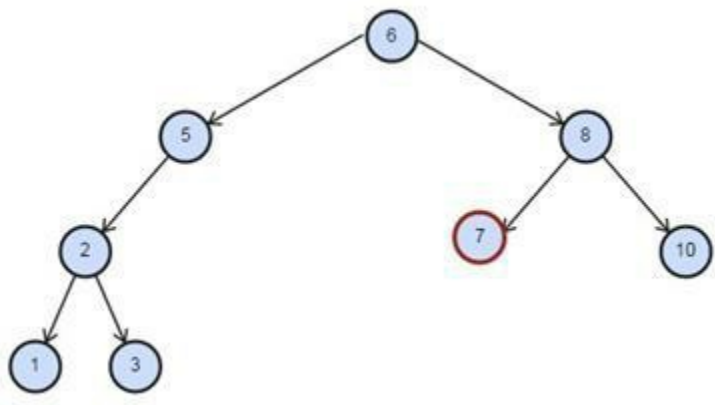
When the node to be deleted has two children.
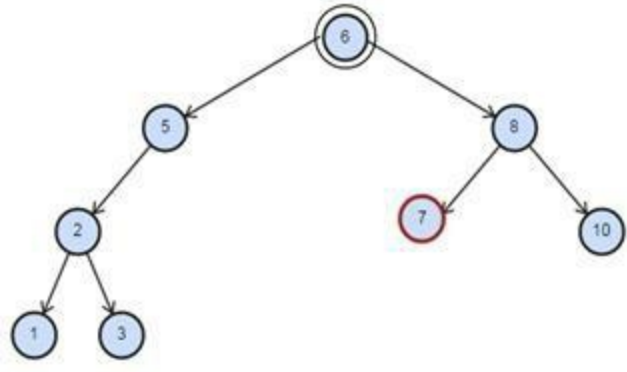
	We want to delete node with value 6. Which have two children.
	We have found minimum value node of the right child of node with value 6.
	Minimum value is copied to the node with value 6.
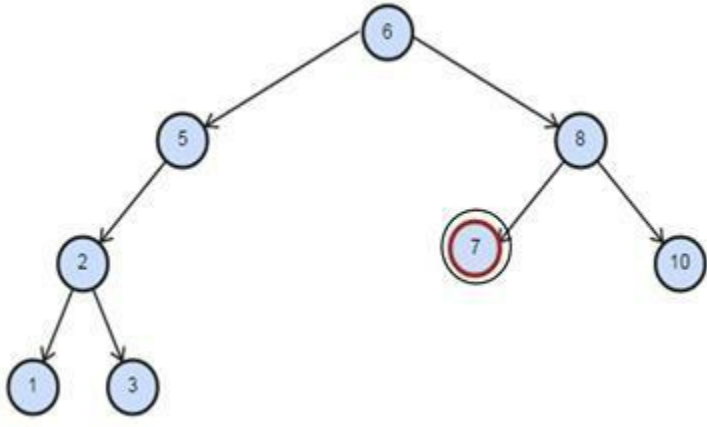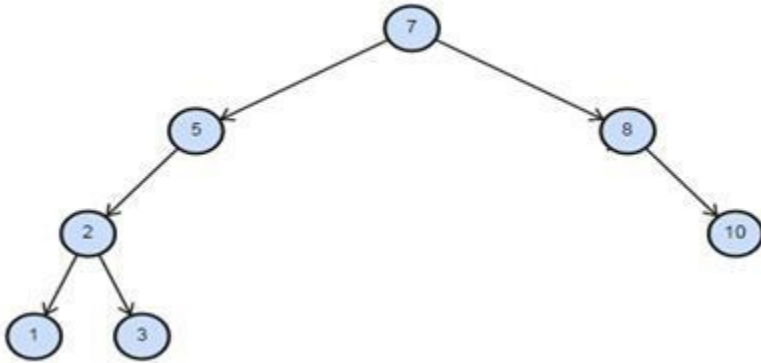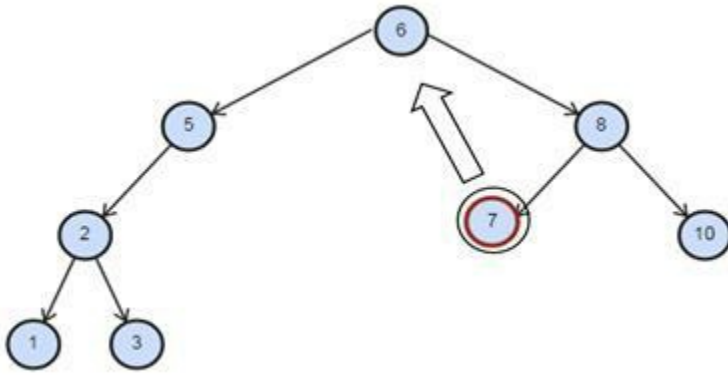
	Delete node with minimum value 7 is called over the right child tree of the node. Finally the tree with both the children is created.

Example 9.34:
```
def DeleteNode(value)
    @root = self.DeleteNodeUtil(@root, value)
end

def DeleteNodeUtil(node, value)
    temp = nil
    if node != nil
        if node.value == value
            if node.lChild == nil and node.rChild == nil
                return nil
            else
                if node.lChild == nil
                    return node.rChild
                end
                if node.rChild == nil
                    return node.lChild
                end
                minNode = self.FindMin(node.rChild)
                minValue = minNode.value
                node.value = minValue
                node.rChild = self.DeleteNodeUtil(node.rChild, minValue)
            end
        else
            if node.value > value
                node.lChild = self.DeleteNodeUtil(node.lChild, value)
            else
```

```
                node.rChild = self.DeleteNodeUtil(node.rChild, value)
            end
        end
    end
    return node
end
```

Analysis: Time Complexity: O(n), Space Complexity: O(n)

Least Common Ancestor

In a tree T. The least common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants.

Example 9.35:
```
def LcaBST(first, second)
    return self.LcaBSTUtil(@root, first, second)
end

def LcaBSTUtil(curr, first, second)
    if curr == nil
        return 1000000
    end
    if curr.value > first and curr.value > second
        return self.LcaBSTUtil(curr.lChild, first, second)
    end
    if curr.value < first and curr.value < second
        return self.LcaBSTUtil(curr.rChild, first, second)
    end
    return curr.value
end
```

Trim the Tree nodes which are Outside Range

In given range as min, max. We need to delete all the nodes of the tree that are out of this range.

Solution: Traverse the tree and each node that is having value outside the range will delete itself. All the deletion will happen from inside out so we do not have to care about the children of a node as if they are out of range then they had already had deleted themselves.

Example 9.36:
```
def trimOutsideRange(min, max, curr = @root)
    if curr == nil
        return nil
    end
    curr.lChild = self.trimOutsideRange(min, max, curr.lChild)
    curr.rChild = self.trimOutsideRange(min, max, curr.rChild)
    if curr.value < min
```

```
      return curr.rChild
   end
   if curr.value > max
      return curr.lChild
   end
   return curr
end
```

Print Tree nodes which are in Range

Print only those nodes of the tree whose value is in the given range.

Solution: Just normal inorder traversal and at the time of printing we will check if the value is inside the given range.

Example 9.37:
```
def printInRange(min, max, curr = @root)
   if curr == nil
      return
   end
   self.printInRange(min, max, curr.lChild)
   if curr.value >= min and curr.value <= max
      print curr.value , " "
   end
   self.printInRange(min, max, curr.rChild)
end
```

Find Ceil and Floor value inside BST given key

In given tree and a value we need to find the ceil value of node in tree which is smaller than the given value and need to find the floor value of node in tree which is bigger. Our aim is to find ceil and floor value as close as possible then the given value.

Example 9.38:
```
def FloorBST(val)
   curr = @root
   ceil = -1000000
   floor = 1000000
   while curr != nil
      if curr.value == val
         ceil = curr.value
         floor = curr.value
         break
      elsif curr.value > val
         ceil = curr.value
         curr = curr.lChild
      else
         floor = curr.value
```

```
            curr = curr.rChild
        end
    end
    return floor
end
```

```
def CeilBST(val)
    curr = @root
    ceil = -1000000
    floor = 1000000
    while curr != nil
        if curr.value == val
            ceil = curr.value
            floor = curr.value
            break
        elsif curr.value > val
            ceil = curr.value
            curr = curr.lChild
        else
            floor = curr.value
            curr = curr.rChild
        end
    end
    return ceil
end
```

Segment Tree

Segment tree is a binary tree that is used to make multiple range queries and range update in an array.

Examples of problems for which Segment Tree can be used are:
1. Finding the sum of all the elements of an array in a given range of index
2. Finding the maximum value of the array in a given range of index.
3. Finding the minimum value of the array in a given range of index (also known as Range Minimum Query problem)

Properties of Segment Tree:
1. Segment tree is a binary tree.
2. Each node in a segment tree represents an interval in the array.
3. The root of tree represents the whole array.
4. Each leaf node represents a single element.

Note:- Segment tree solves problems which can be solve in linear time by just scanning and updating the elements of array. The only benefit we are getting from segment tree is that it does update and query operation in logarithmic time that is more efficient than the linear approach.

Let us consider a simple problem:

Given an array of N numbers. You need to perform the following operations:
1. Update any element in the array
2. Find the maximum in any given range (i, j)

Solution 1:
Updating: Just update the element in the array, a[i] =x. Finding maximum in the range (i, j), by traversing through the elements of the array in that range.
Time Complexity of Update is O(1) and of Finding is O(n)

Solution 2: The above solution is good. However, can we improve performance of Finding?
The answer is yes. In fact, we can do both the operations in O(log n) where n is the size of the array. This we can do using a segment tree.

Let us suppose we are given an input array A = {1, 8, 2, 7, 3, 6, 4, 5}. Moreover, the below diagram will represent the segment tree formed corresponding to the input array A.

Input Array : A = {1,8,2,7,3,6,4,5}
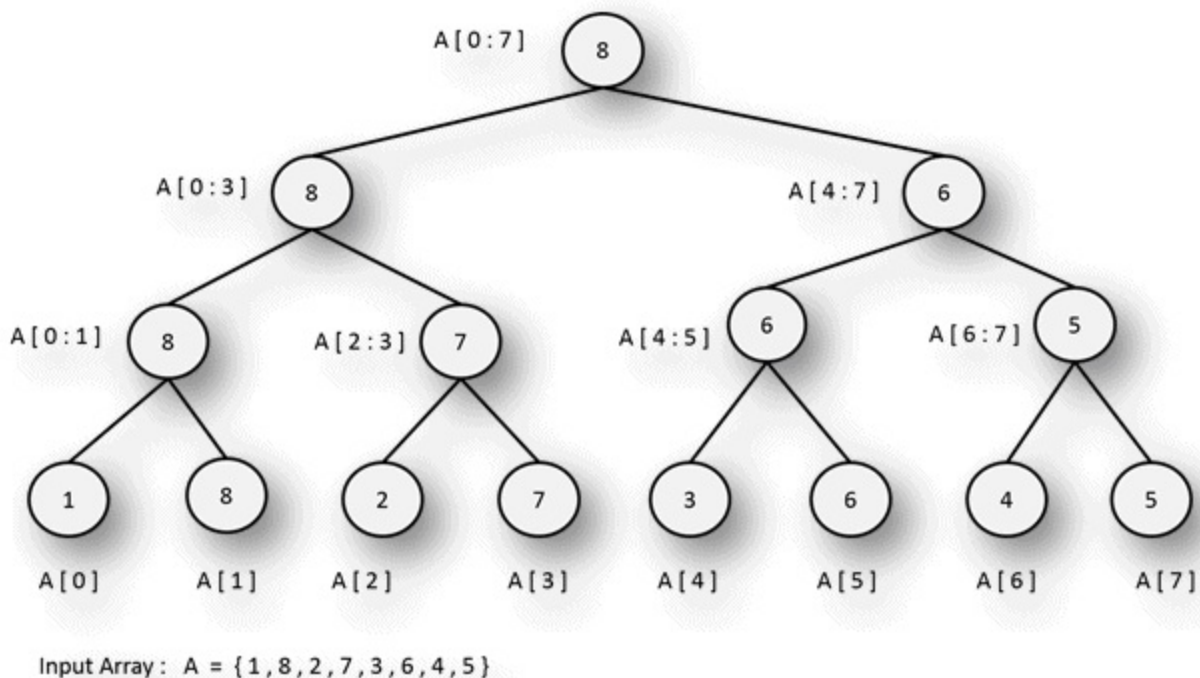
AVL Trees

An AVL tree is a binary search tree (BST) with an additional property that the subtrees of every node differ in height by at most one. An AVL tree is a height balanced BST.

AVL tree is a balanced binary search tree. Adding or removing a node form AVL tree may make the AVL tree unbalanced. Such violations of AVL balance property is corrected by one or two simple steps called rotations. Let us assume that insertion of a new node converted a previously balanced AVL tree into an unbalanced tree. Since the tree is previously balanced and a single new node is added to it, the unbalance maximum difference in height will be 2.

Therefore, in the bottom most unbalanced node there are only four cases:
Case 1: The new node is left child of the left child of the current node.
Case 2: The new node is right child of the left child of the current node.
Case 3: The new node is left child of the right child of the current node.

Case 4: The new node is right child of the right child of the current node.
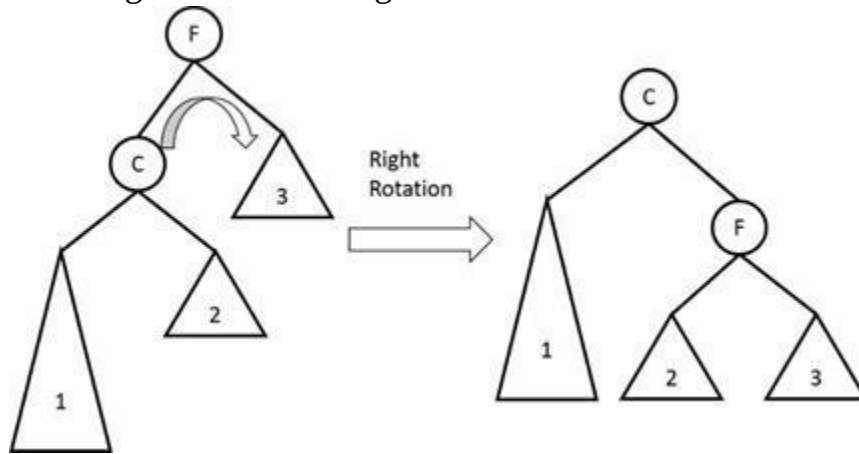
Case 1 can be re-balanced using a single Right Rotation.
Case 4 is symmetrical to Case 1: can be re-balanced using a single Left Rotation
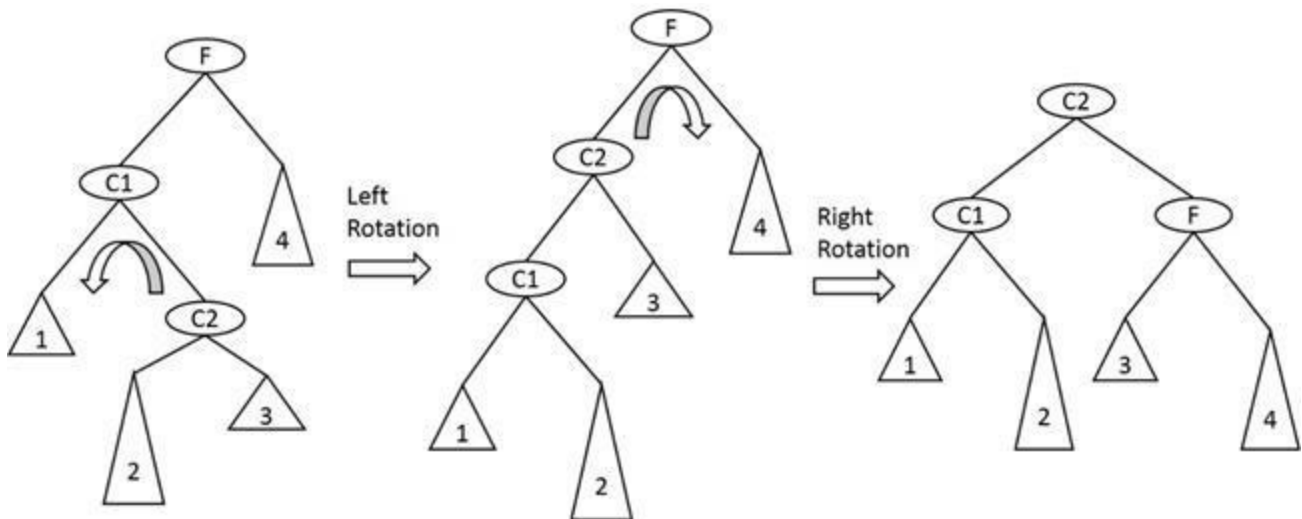
Case 2 can be re-balanced using a double rotation. First, rotate left than rotation right.
Case 3 is symmetrical to Case 2: can be re-balanced using a double rotation. First, rotate right than rotation left.

Time Complexity of Insertion: To search the location where a new node needs to be added is done in O(log(n)). Then on the way back, we look for the AVL balanced property and fixes them with rotation. Since the rotation at each node is done in constant time, the total amount of word is proportional to the length of the path. Therefore, the final time complexity of insertion is O(log(n)).

Red-Black Tree

The red-black tree contains its data, left and right children like any other binary tree. In addition to this its node also contains an extra bit of information which represents colour which can either red or black. Red-Black tree also contains a specialized class of nodes called NULL nodes. NULL nodes are pseudo nodes that exists at the leaf of the tree. All internal nodes have their own data associated with them.

Red-Black tree has the following properties:
1. Root of tree is black.
2. Every leaf node (NULL node) is black.
3. If a node is red then both of its children are black.
4. Every path from a node to a descendant leaf contains the same number of black nodes.

The first three properties are self-explanatory. The forth property states that, from any node in the tree to any leaf (NULL), the number of black nodes must be the same.
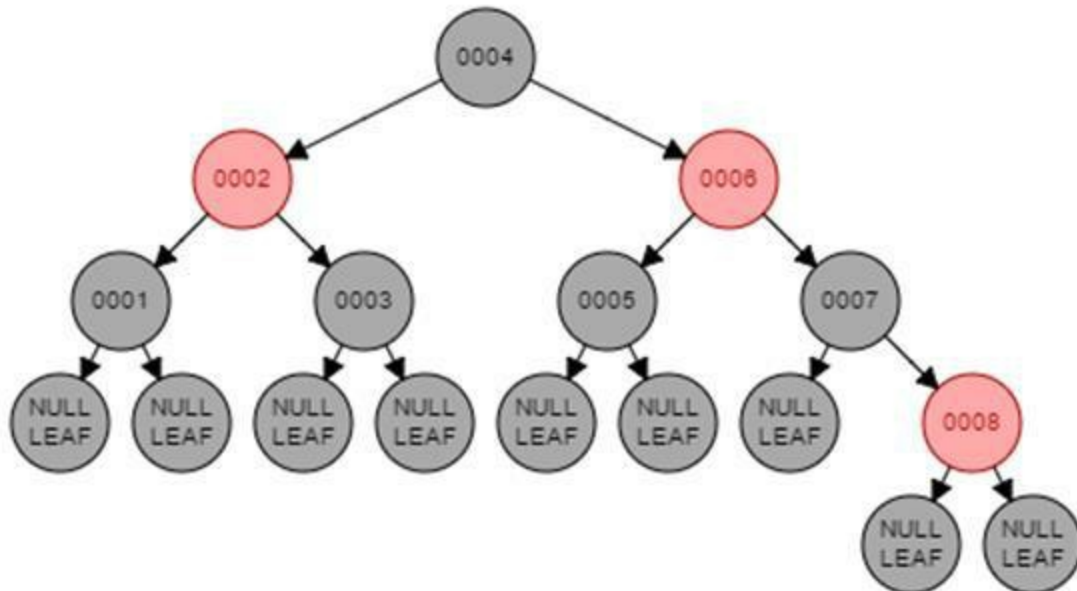
In the above figure, from the root node to the leaf node (NULL) the number of black node is always three nodes.

Like the AVL tree, red-black trees are also self-balancing binary search tree. Whereas the balance property of an AVL tree has a direct relationship between the heights of left and right subtrees of each node. In red-black trees, the balancing property is governed by the four rules mentioned above. Adding or removing a node form red-black tree may violate the properties of a red-black tree. The red-black properties are restored through recolouring and rotation. Insert, delete, and search operation time complexity is O(log(n))

Splay tree

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time.
Elements of the tree are rearranged so that the recently accessed element is placed at the top of the tree. When an element is searched then we use standard BST search and then use rotation to bring the element to the top.

	Average Case	Worst Case
Space complexity	O(n)	O(n)
Time complexity search	O(log(n))	Amortized O(log(n))
Time complexity insert	O(log(n))	Amortized O(log(n))

Time complexity delete	O(log(n))	Amortized O(log(n))

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap.

B-Tree

As we had already seen various types of binary tree for searching, insertion and deletion of data in the main memory. However, these data structures are not appropriate for huge data that cannot fit into main memory, the data that is stored in the disk.
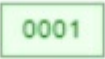
A B-tree is a self-balancing search tree that allows searches, insertions, and deletions in logarithmic time. The B-tree is a tree in which a node can have multiple children. Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write entire blocks (page) of data. The read - write operation from disk is very slow as compared with the main memory. The main purpose of B-Tree is to reduce the number of disk access. The node in a B-Tree has a huge number of references to the children nodes. Thereby reducing the size of the tree. While accessing data from disk, it makes sense to read an entire block of data and store into a node of tree. B-Tree nodes are designed such that entire block of data (page) fits into it. It is commonly used in databases and filesystems.
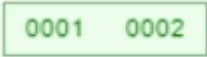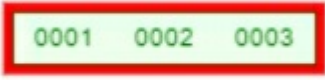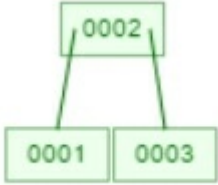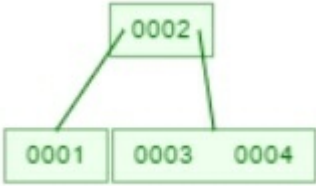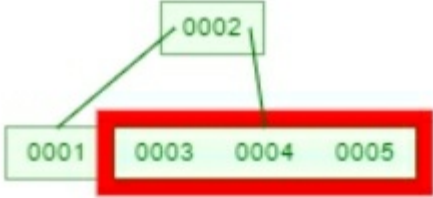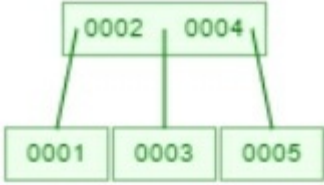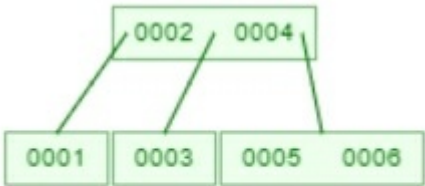
B-Tree of minimum degree d has the following properties:
1. All the leaf nodes must be at same level.
2. All nodes except root must have at least (d-1) keys and maximum of (2d-1) keys. Root may contains minimum 1 key.
3. If the root node is a non-leaf node, then it must have at least 2 children.
4. A non-leaf node with N keys must have (N+1) number of children.
5. All the key values within a node must be in Ascending Order.
6.   All keys of a node are sorted in ascending order. The child between two keys, K1 and K2 contains all keys in range from K1 and K2.

B-Tree	Average Case	Worst Case
Space complexity	O(n)	O(n)
Time complexity search	O(log(n))	O(log(n))
Time complexity insert	O(log(n))	O(log(n))
Time complexity delete	O(log(n))	O(log(n))

Below is the steps of creation of B-Tree by adding value from 1 to 7.

1		Insert 1 to the tree.	Stable
2		Insert 2 to the tree.	Stable

	0001 0002		
3	0001 0002 0003	Insert 3 to the tree.	Intermediate
4	0002 / 0001 0003	New node is created and data is distributed.	Stable
5	0002 / 0001 0003 0004	Insert 4 to the tree.	Stable
6	0002 / 0001 0003 0004 0005	Insert 5 to the tree.	Intermediate
7	0002 0004 / 0001 0003 0005	New node is created and data is distributed.	Stable
8	0002 0004 / 0001 0003 0005 0006	Insert 6 to the tree.	Stable
9		Insert 7 to the tree. New node is created and data is distributed.	Intermediate

10		After rearranging the intermediate node also have more than maximum number of keys.	Intermediate
11		New node is created and data is distributed. The height of the tree is increased.	Stable

Note:- 2-3 tree is a B-tree of degree three.

B+ Tree

B+ Tree is a variant of B-Tree. The B+ Tree stores records only at the leaf nodes. The internal nodes store keys. These keys are used for insertion, deletion and search. The rules of splitting and merging of nodes are same as B-Tree.

b-order B+ tree	Average Case	Worst Case
Space complexity	O(n)	O(n)
Time complexity search	O($\log_b$(n))	O($\log_b$(n))
Time complexity insert	O($\log_b$(n))	O($\log_b$(n))
Time complexity delete	O($\log_b$(n))	O($\log_b$(n))

Below is the B+ Tree created by adding value from 1 to 5.

1.		Value 1 is inserted to leaf node.
2.		Value 2 is inserted to leaf node.
3.		Value 3 is inserted to leaf node.

		Content of the leaf node passed the maximum number of elements. Therefore, node is split and intermediate / key node is created.
4.		Value 4 is further inserted to the leaf node. Which further splits the leaf node.
5.		Value 5 is added to the leaf node the number of nodes in the leaf passed the maximum number of nodes limit that it can contain so it is divided into 2. One more key is added to the intermediate node, which also make it passed maximum number of nodes it can contain, and finally divided and a new node is created.

B* Tree

The B* tree is identical to the B+ tree, except for the rules for splitting and merging of nodes. Instead of splitting a node into two halves when it overflows, the B* tree node tries to give some of its records to its neighbouring sibling. If the sibling is also full, then a new node is created and records are distributed into three.

Exercise

1. Construct a tree given its in-order and pre-order traversal strings.
   - o inorder: 1 2 3 4 5 6 7 8 9 10
   - o pre-order: 6 4 2 1 3 5 8 7 9 10

2. Construct a tree given its in-order and post-order traversal strings.
   - o inorder: 1 2 3 4 5 6 7 8 9 10
   - o post-order: 1 3 2 5 4 7 10 9 8 6

3. Write a delete node function in Binary tree.

4. Write a function print depth first in a binary tree without using system stack
   - Hint: you may want to keep another element to tree node like visited flag.

5. Check whether a given Binary Tree is Complete or not
   - o In a complete binary tree, every level except the last one is completely filled. All nodes

in the left are filled first, then the right one.
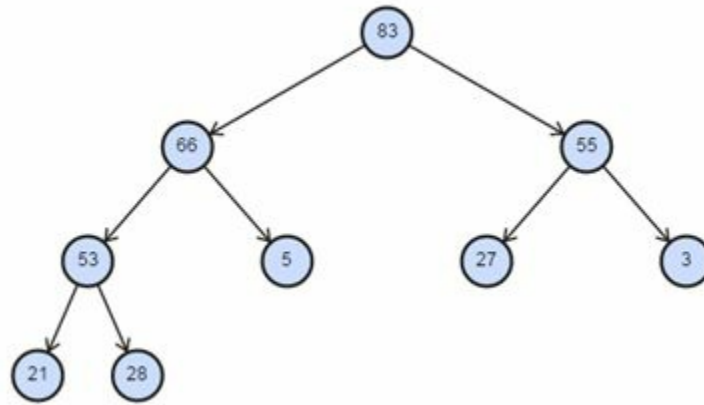
6. Check whether a given Binary Tree is Full/ Strictly binary tree or not. The full binary tree is a binary tree in which each node has zero or two children.
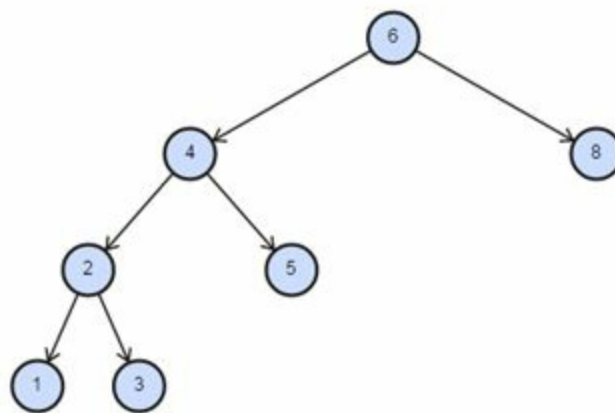
7. Check whether a given Binary Tree is a Perfect binary tree or not. The perfect binary tree- is a type of full binary trees in which each non-leaf node has exactly two child nodes.

8. Check whether a given Binary Tree is Height-balanced Binary Tree or not. A height-balanced binary tree is a binary tree such that the left & right subtrees for any given node differs in height by not more than one
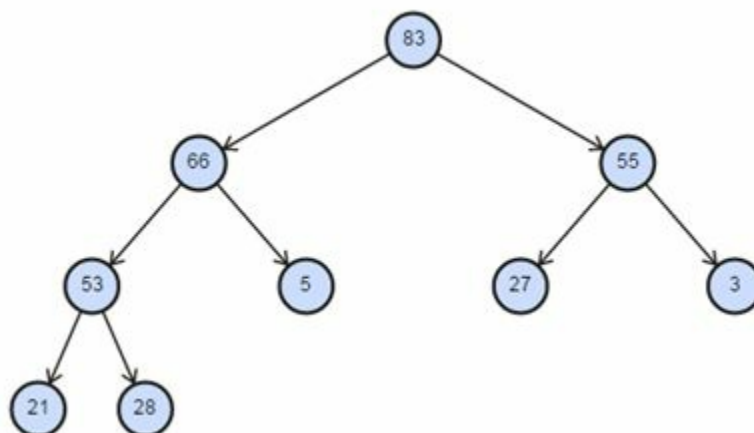
9. Isomorphic: two trees are isomorphic if they have the same shape, it does not matter what the value is. Write a program to find if two given tree are isomorphic or not.

10. The worst-case runtime Complexity of building a BST with n nodes

- o $O(n^2)$
- o $O(n * \log n)$
- o $O(n)$
- o $O(\log n)$

11. The worst-case runtime Complexity of insertion into a BST with n nodes is
   - o $O(n2)$
   - o $O(n * \log n)$
   - o $O(n)$
   - o $O(\log n)$

12. The worst-case runtime Complexity of a search of a value in a BST with n nodes is:
   - o $O(n^2)$
   - o $O(n * \log n)$
   - o $O(n)$
   - o $O(\log n)$

13. Which of the following traversals always gives the sorted sequence of the elements in a BST?
   - o Preorder
   - o Ignored
   - o Postorder
   - o Undefined

14. The height of a Binary Search Tree with n nodes in the worst case?
   - o $O(n * \log n)$
   - o $O(n)$
   - o $O(\log n)$
   - o $O(1)$

15. Try to optimize the above solution to give a DFS traversal without using recursion use some stack or queue.

16. This is an open exercise for the readers. Every algorithm that is solved using recursion (system stack) can also be solved using user defined or library defined stack. So try to figure out what all algorithms that uses recursion and try to figure out how you will do this same using user defined stack.

17. In a binary tree, print the nodes in zigzag order. In the first level, nodes are printed in the left to right order. In the second level, nodes are printed in right to left and in the third level again in the order left to right.
    Hint: Use two stacks. Pop from first stack and push into another stack. Swap the stacks alternatively.

18. Find nth smallest element in a binary search tree.
    Hint: Nth inorder in a binary tree.

19. Find the floor value of key that is inside a BST.

20. Find the Ceil value of key, which is inside a BST.

CHAPTER 10: PRIORITY QUEUE

Introduction

A Priority-Queue, also known as Binary-Heap, is a variant of queue. Items are removed from the beginning of the queue. However, in a Priority-Queue the logical ordering of objects is determined by their priority. The highest priority item is at the front of the Priority-Queue. When you add an item to the Priority-Queue, the new item can more to the front of the queue. A Priority-Queue is a very important data structure.   Priority-Queue is used in various Graph algorithms like [Prim's Algorithm](#) and [Dijkstra's algorithm](#). Priority-Queue is also used in the timer implementation etc.

A Priority-Queue is implemented using a Heap (Binary Heap). A Heap data structure is an array of elements that can be observed as a complete binary tree. The tree is completely filled on all levels except possibly the lowest. Heap satisfies the heap ordering property. In max-heap, the parent's value is greater than or equal to its children value. In min-heap, the parent's value is less than or equal to its children value. A heap is a complete binary tree so the height of tree with N nodes is always **O(logn)**.

Max Heap
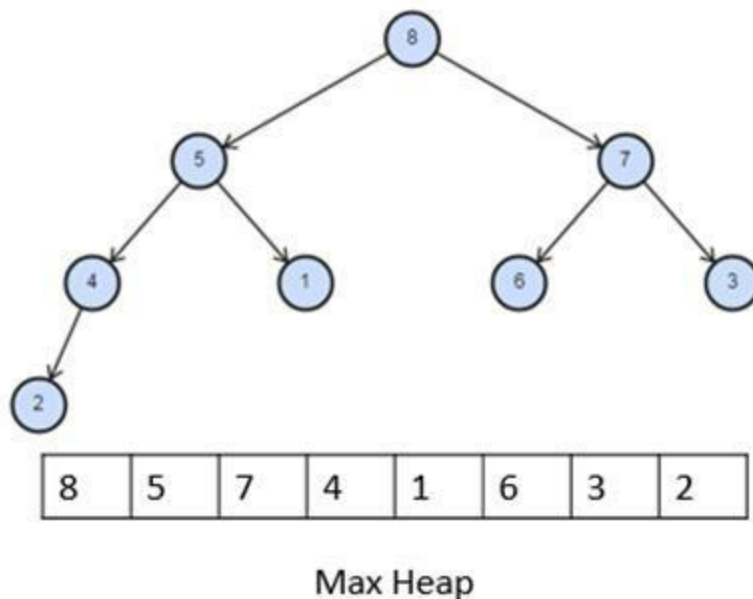
A heap is not a sorted data structure and can be regarded as partially ordered. As you can see in the picture, there is no relationship among nodes at any given level, even among the siblings.

Heap is implemented using an array. Moreover, because heap is a complete binary tree, the left child of a parent (at position x) is the node that is found in position 2x in the array. Similarly, the right child of the parent is at position 2x+1 in the array. To find the parent of any node in the heap, we can simply make division. In given index y of a node, the parent index will by y/2.
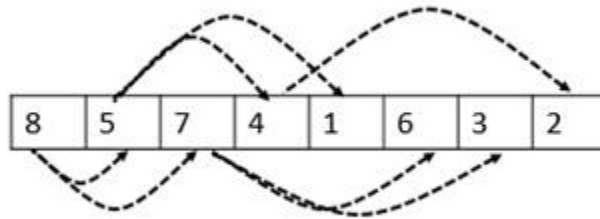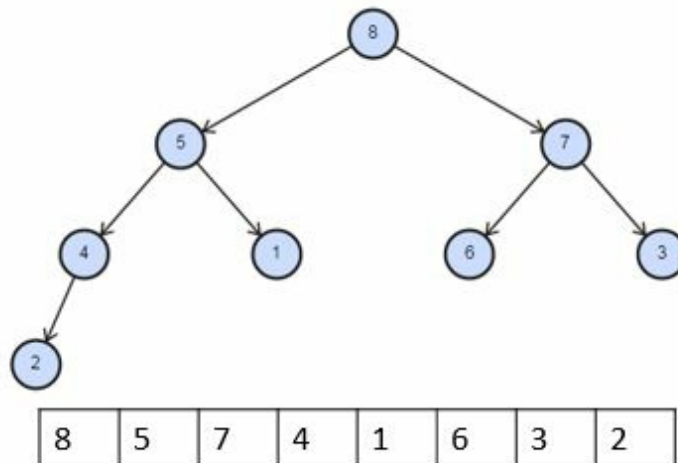
Types of Heap

There are two types of heap and the type depends on the ordering of the elements. The ordering can be done in two ways: Min-Heap and Max-Heap

Max Heap

Max-Heap: the value of each node is less than or equal to the value of its parent, with the largest-value element at the root.
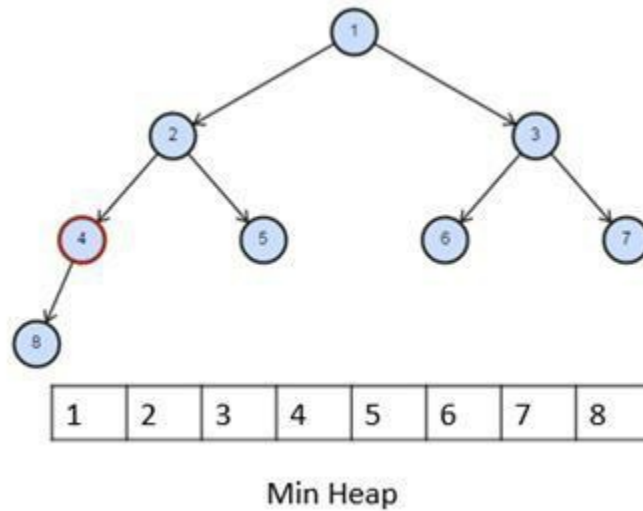

Max Heap

Max Heap Operations

Insert	O(logn)
DeleteMax	O(logn)
Remove	O(logn)
FindMax	O(1)

Min Heap

Min-Heap: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.

Min Heap

Use it whenever you need quick access to the smallest item, because that item will always be at the root of the tree or the first element in the array. However, the remainder of the array is kept partially sorted. Thus, instant access is only possible for the smallest item.

Min Heap Operations

Insert	O(logn)
DeleteMin	O(logn)
Remove	O(logn)
FindMin	O(1)

Throughout this chapter, the word "heap" will always refer to a max-heap. The implementation of min-heap is left for the user to do it as an exercise.

Heap ADT Operations

The basic operations of binary heap are as follows:

Binary Heap	Creates a new empty binary heap	O(1)
Insert	Adding a new element to the heap	O(logn)
DeleteMax	Deletes the maximum element form the heap.	O(logn)
FindMax	Finds the maximum element in the heap.	O(1)
isEmpty	Returns true if the heap is empty else return false	O(1)
Size	Returns the number of elements in the heap.	O(1)
BuildHeap	Builds a new heap from the array of elements	O(logn)

Operation on Heap

Create Heap from an array

Heapify is the process of converting an array into Heap. The various steps are:
1. Values are present in the array.
2. Starting from middle of the array move downward towards the start of the array. At each

step, compare parent value with its left child and right child. In addition, restore the heap property by shifting the parent value with its largest-value child. Such that the parent value will always be greater than or equal to left child and right child.

3. For all elements from middle of the array to the start of the array. We make comparisons and shift, until we reach the leaf nodes of the heap. The Time Complexity of build heap is **O(N)**.

|	
	Given an array as input to create heap function. Value of index i is compared with value of its children nodes that is at index (i*2 + 1) and (i*2 + 2). Middle of array N/2, that is index 3, is comapred with index 7. If the children node value is greater than parent node then the value will be swapped.
	Similarly, value of index 2 is compared with index 5 and 6. The largest of all the values is 7 which will be swapped with the value at the index 2.
	Similarly, value of index 1 is compared with index 3 and 4 The largest of all the values is 8 which will be swapped with the value at the index 1.
	Percolate down function is used to subsequently adjust the value replased in the previous step by comparing it with its children
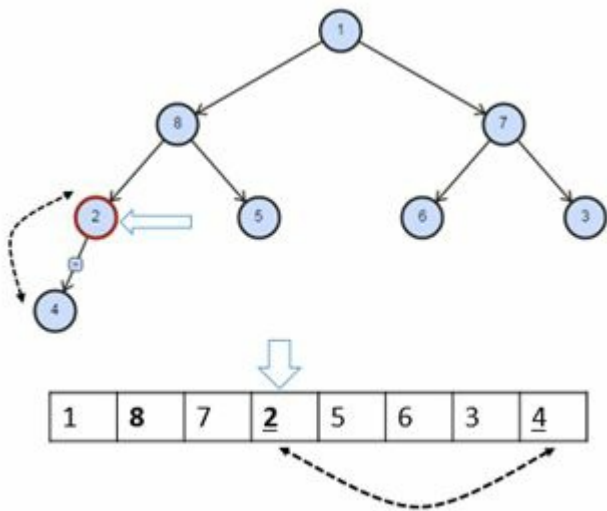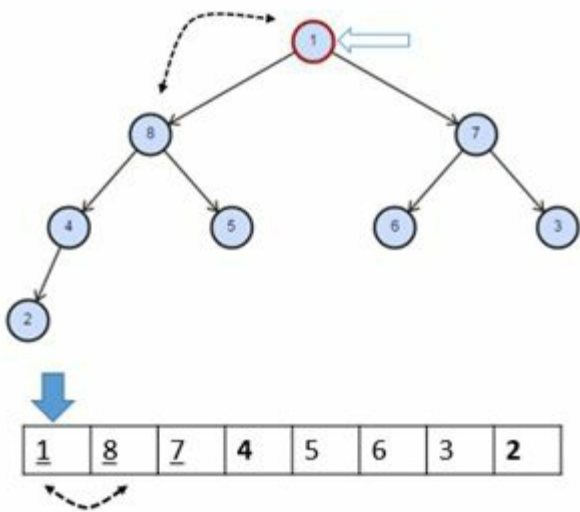
nodes.

Now value at index 0 is comared with index 1 and 2. 8 is the largest value so it is swapped with the value at index 0.
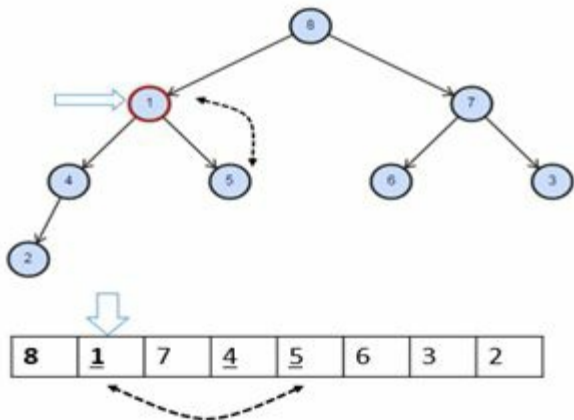
Percolate down function is used to further compare the value at index 1 with its children nodes at index 3 and 4.
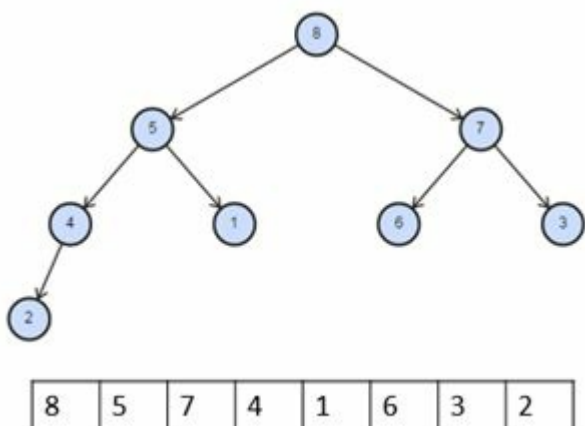
In the end max heap is created.

Example 10.1:

```
class Heap
    # Number of elements in Heap
    # The Heap array
    def initialize(array, isMinHeap = true)
        @size = array.size
        @arr = array.clone
        @arr.unshift(1) #we do not use 0 index
        @isMinHeap = isMinHeap
        #Build Heap operation over array
        i = (@size / 2)
        while i > 0
            self.proclateDown(i)
            i -= 1
        end
    end

    def comp(first, second)
        if @isMinHeap then
            return (@arr[first] > @arr[second])
        else
            return (@arr[second] > @arr[first])
        end
    end

#Other Methods.
End
```

```
def proclateDown(position)
    lChild = 2 * position
    rChild = lChild + 1
    small = -1
    if lChild <= @size then
        small = lChild
    end
    if rChild <= @size and (self.comp(rChild, lChild)== false) then
        small = rChild
    end
    if small != -1 and (self.comp(small, position)== false) then
        temp = @arr[position]
        @arr[position] = @arr[small]
        @arr[small] = temp
        self.proclateDown(small)
    end
end
```

```
def proclateUp(position)
```

```
            parent = position / 2
            if parent == 0 then
                return
            end
            if self.comp(parent, position) == true then #parent grater then child.
                temp = @arr[position]
                @arr[position] = @arr[parent]
                @arr[parent] = temp
                self.proclateUp(parent)
            end
        end

def display()
    i = 1
    while i <= size + 1
        print "value is :: " , arr[i]
        i += 1
    end
end

def isEmpty()
    return (@size == 0)
end

def peek()
    if self.isEmpty() then
        raise StandardError, "HeapEmptyException"
    end
    return @arr[1]
end

def size()
    return @size
end
```

Enqueue / Insert

1. Add the new element at the end of the array. This keeps the structure as a complete binary tree, but it might no longer be a heap since the new element might have a value greater than its parent's value.
2. Swap the new element with its parent until it has value greater than its parent's value.
3. Step 2 will be terminated when the new element reaches the root or when the new element's parent has a value greater than or equal to the new element's value.

Let us take an example of the Max heap created in the above example.

| 8 | 5 | 7 | 4 | 1 | 6 | 3 | 2 |
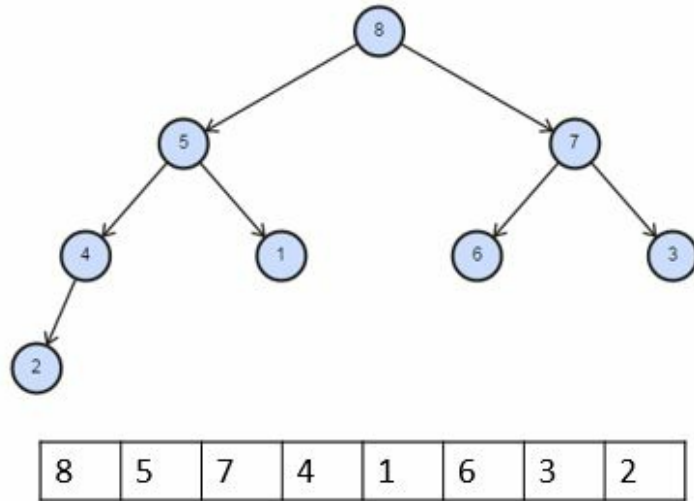
Let us take an example by inserting element with value 9 to the heap. The element is added to the end of the heap array. Now the value will be percolated up by comparing it with the parent. The value is added to index 8 and its parent will be (N-1)/2 = index 3.

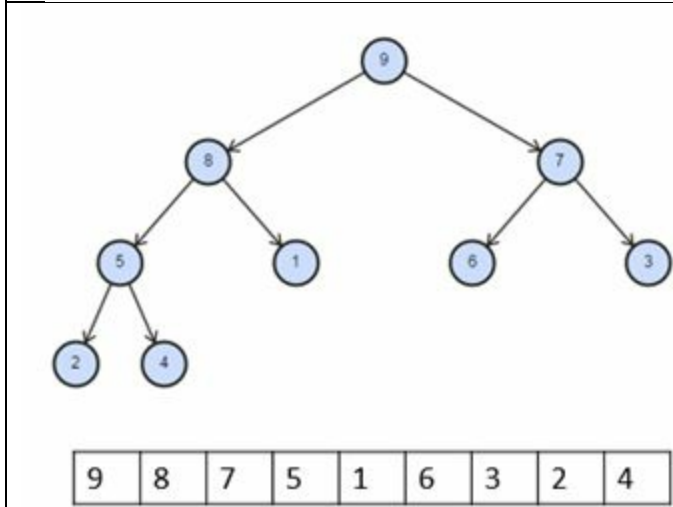|	Since the value 9 is greater then 4 it will be swapped with it.
	Percolate up is used and the value is moved up until heap property is satisfied.
	Now the value at index 1 is compared with
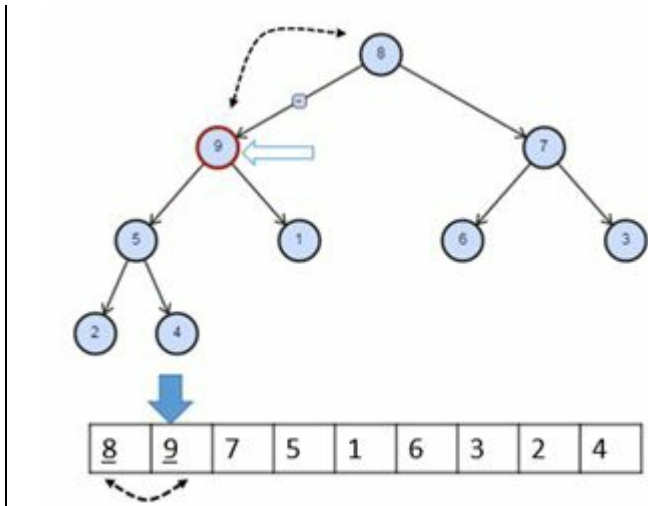
| | index 0 and to satisfy heap property it is further swapped. |

| | Now, finally max heap is created by inserting new node. |

Example 10.2:

```
def add(value)
    @size += 1
    @arr[@size] = value
    self.proclateUp(@size)
end
```

Dequeue / Delete

1. Copy the value at the root of the heap to the variable that will be used to return the value.
2. Copy the last element of the heap to the root, and then reduce the size of heap by 1. This element is called the "out-of-place" element.
3. Restore heap property by swapping the out-of-place element with its greatest-value child. Repeat this process until the out-of-place element reaches a leaf or it has a value that is greater or equal to all its children.
4. Return the answer that was saved in Step 1.

To remove an element from the heap its top value is swapped to the end of the heap array and size of heap is reduced by 1.
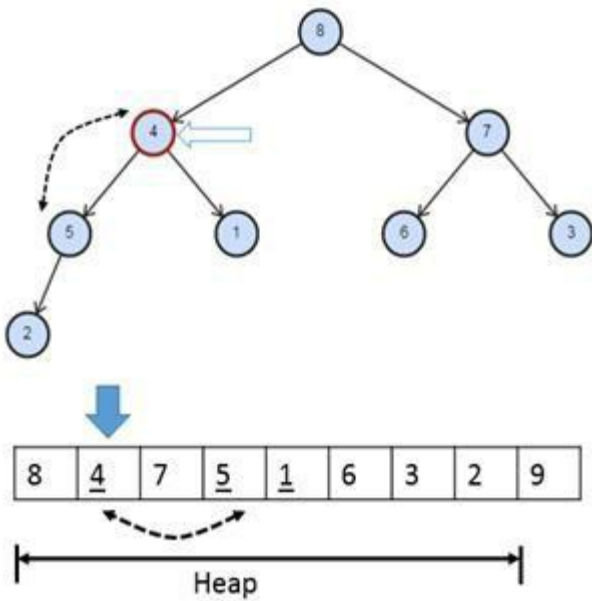
| | Since value at end of the heap is copied to head of the heap. Heap property is disturbed so we |
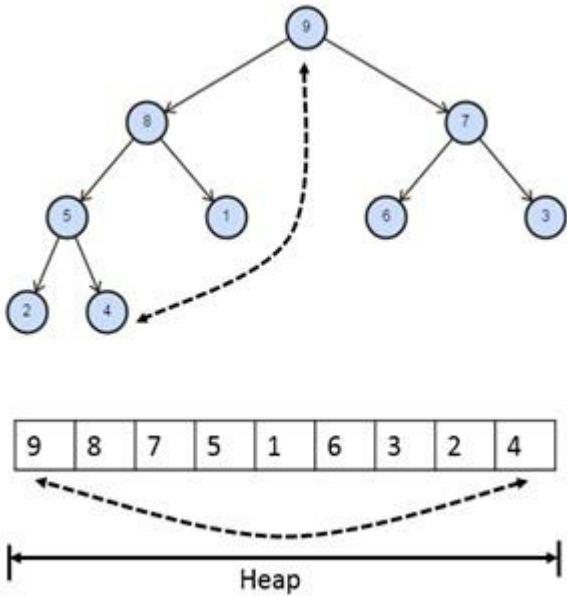
need to percolate down by comparing node with its children nodes and restore heap property.
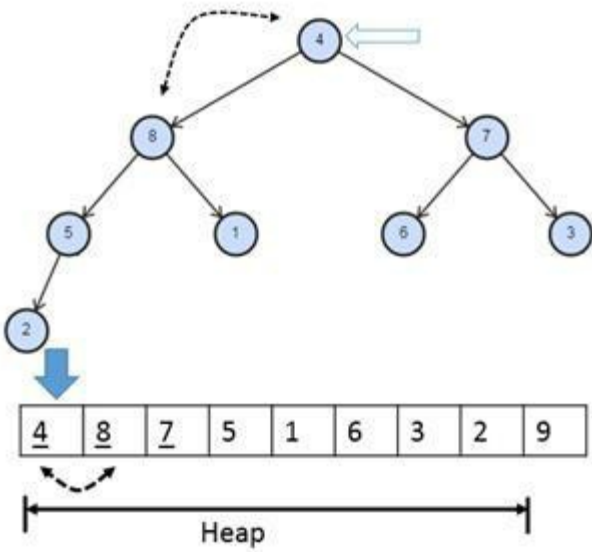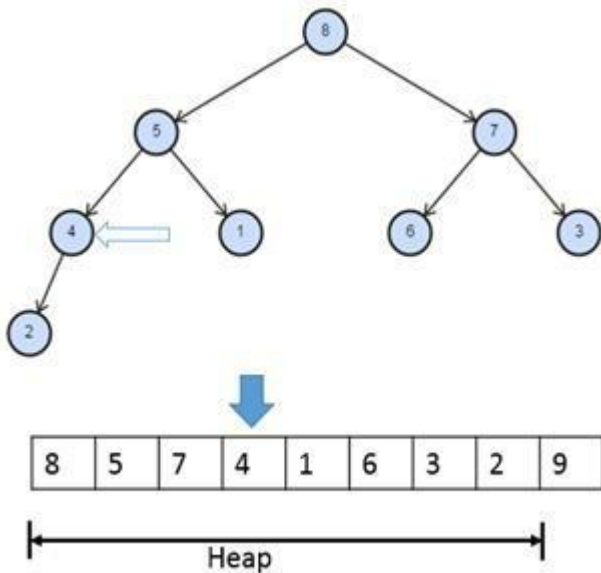

Percolate down is continued by comparing with its children nodes.

Percolate down

Percolate down Complete

Example 10.3

```
def remove()
    if self.isEmpty() then
        raise StandardError, "HeapEmptyException"
    end
    value = @arr[1]
    @arr[1] = @arr[size]
    @size -= 1
    self.proclateDown(1)
    return value
end

# Testing code
a = [9, 8, 10, 7, 6, 1, 4, 2, 5, 3]
pq = Heap.new(a, true)
pq.add(2);
pq.add(3);
count = pq.size()
i = 0
while i < count
    print "value is :: " , pq.remove(), "\n"
    i += 1
end
```

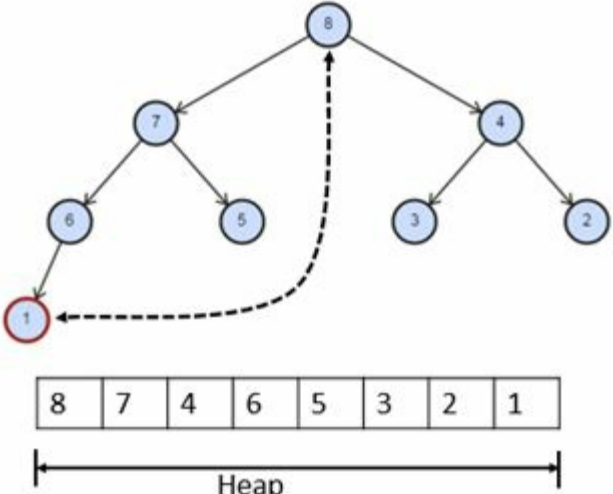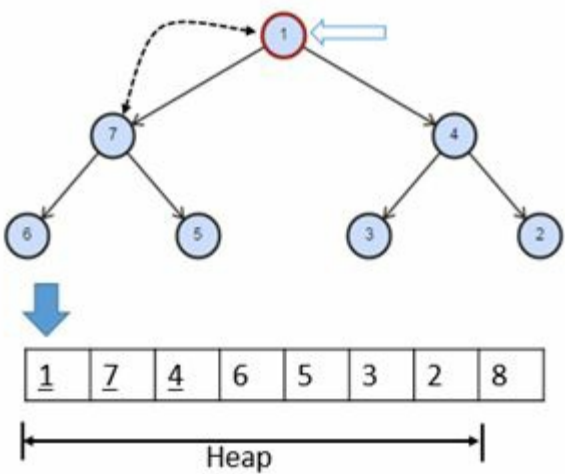## Heap-Sort

1. Use create heap function to build a max heap from the given array of elements. This operation will take **O(N)** time.
2. Dequeue the max value from the heap and store this value to the end of the array at location arr[size-1]
   a) Copy the value at the root of the heap to end of the array.
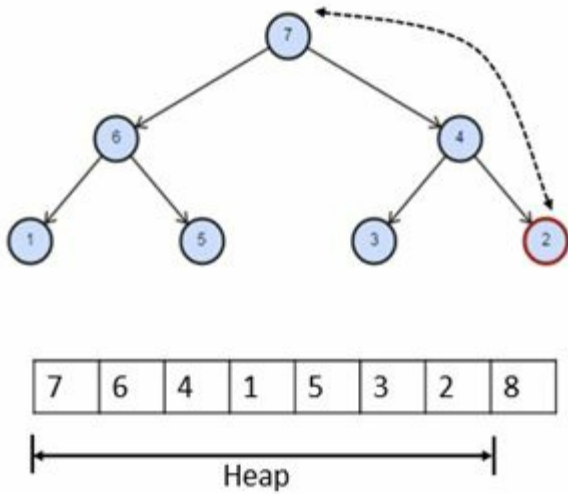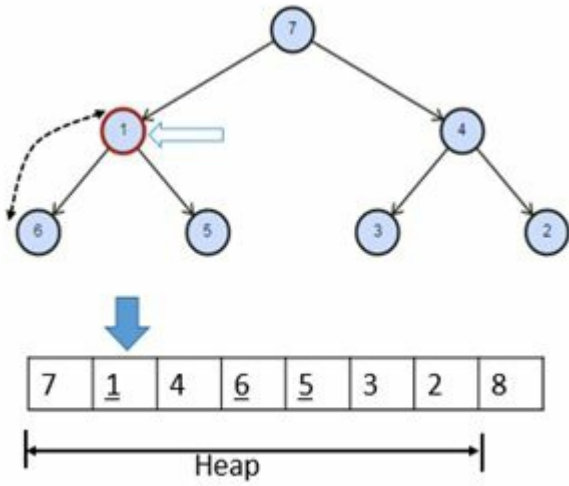   b) Copy the last element of the heap to the root, and then reduce the size of heap by 1.

This element is called the "out-of-place" element.
   c)   Restore heap property by swapping the out-of-place element with its greatest-value child. Repeat this process until the out-of-place element reaches a leaf or it has a value that is greater or equal to all its children
3. Repeat this operation until there is just one element in the heap.

Let us take an example of the heap that we had already created at the start of the chapter. Heap sort is algorithm starts by creating a heap of the given array, which is done in linear time. Then at each step head of the heap is swapped with the end of the heap and the heap size is reduced by 1. Then percolate down is used to restore the heap property. Moreover, the same is done multiple times until the heap contain just one element.

	We had started with max heap. The maximum value as the first element of the Heap array is swapped with the last element of the array. Now the largest value is at the end of the array. Then we will reduce the size of the heap by one.
	Since 1 is at the top of the heap. Moreover, heap property is lost we will use Percolate down method to regain the heap property.
	Percolate down cont.

Since heap property is regained. Then we will copy the first element of the heap array to the second last position.
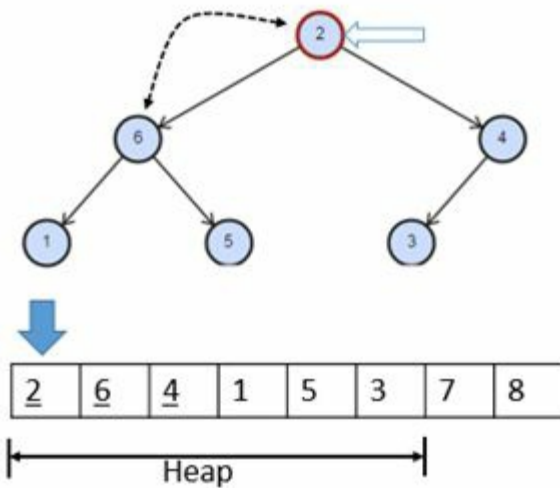
Heap size is further reduced and percolate down cont.

Percolate down cont.

	Again swap.
	Size of heap is reduced and percolate down.
	Again swap.

6	2	4	1	5	3	7	8

Heap

6	5	4	1	2	3	7	8

Heap

3	5	4	1	2	6	7	8
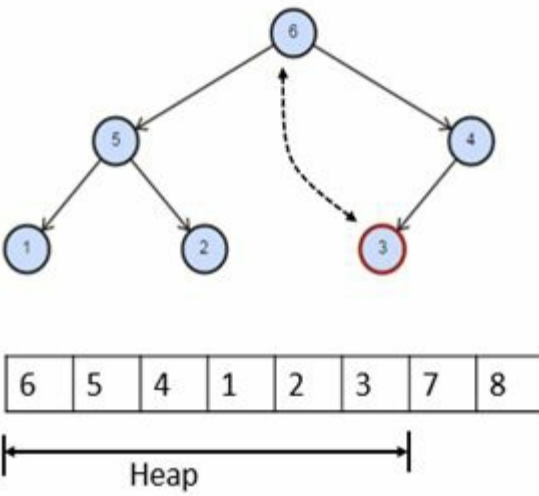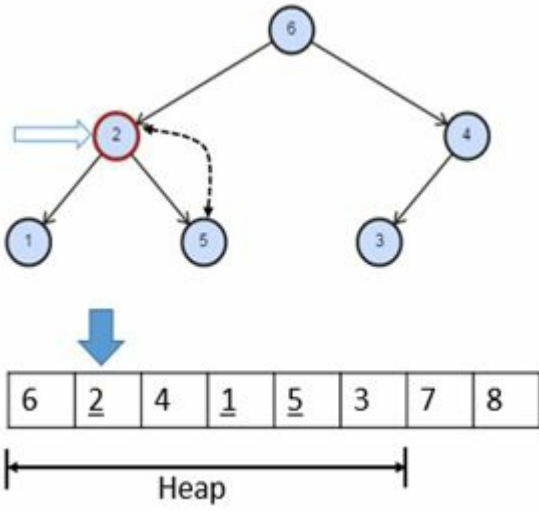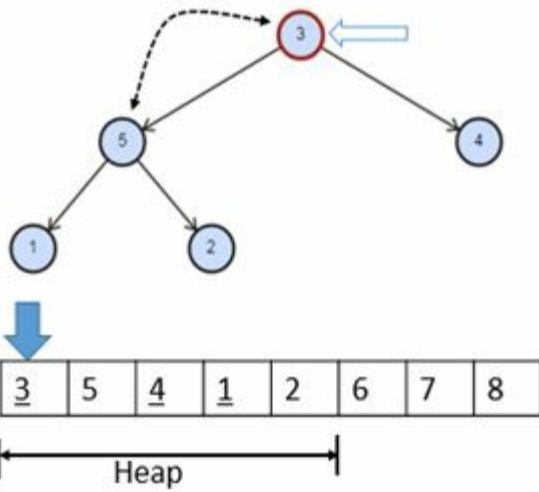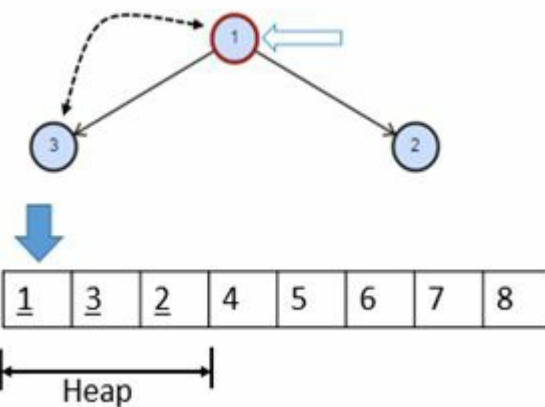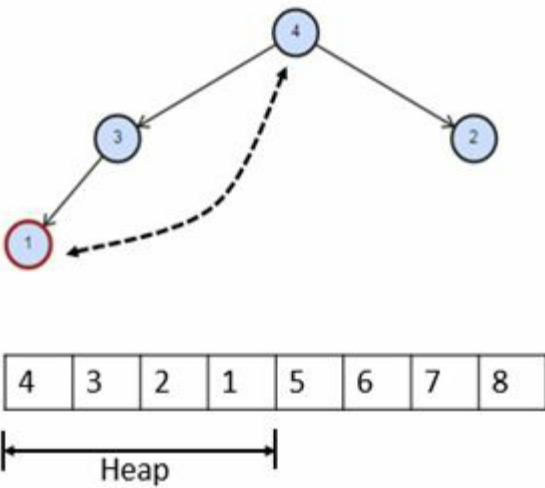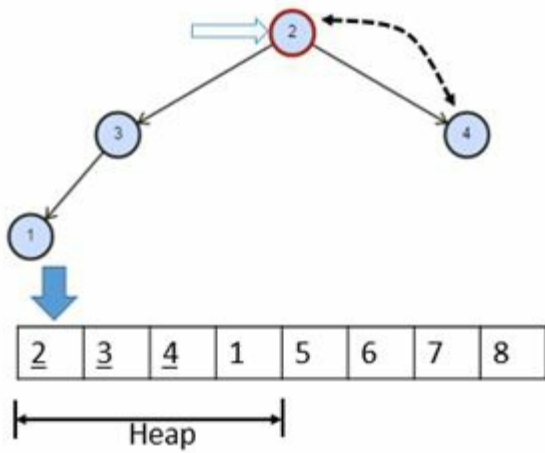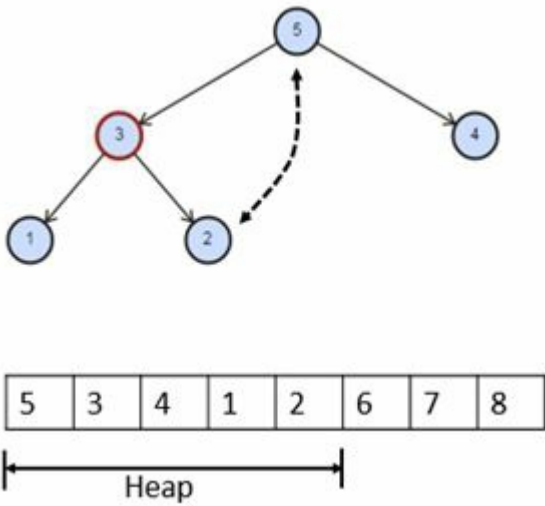
Heap

	Size of heap is reduced and percolate down.
	Again swap.
	Size of heap is reduced and percolate down.
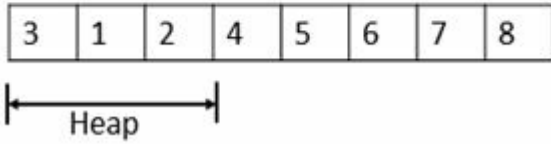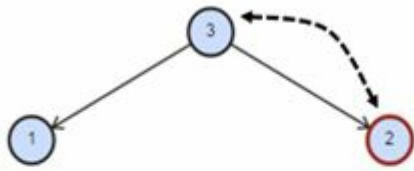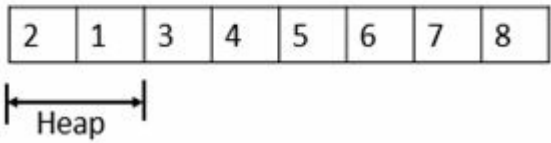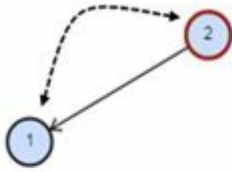
	Again swap.
	Again swap.
	End.
	Final array, which is sorted in increasing order.
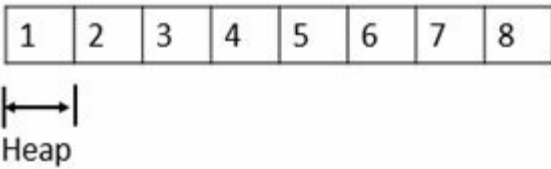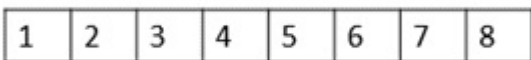
Example 10.4:

```
def HeapSort(array)
    hp = Heap.new(array)
    i = 0
    while i < array.size
        array[i] = hp.remove()
        i += 1
    end
end

# Testing code
```

```
a = [1, 9, 6, 7, 8, 0, 2, 4, 5, 3]
heapSort(a)
print a,
```

Data structure	Array
Worst Case Time Complexity	O(nlogn)
Best Case Time Complexity	O(nlogn)
Average Time Complexity	O(nlogn)
Space Complexity	O(1)

Note: Heap-Sort is not a Stable sort and does not require any extra space for sorting an array.

Uses of Heap

1. **Heapsort**: One of the best sorting methods being in-place and log(N) time complexity in all scenarios.

2. **Selection algorithms**: Finding the min, max, both the min and max, median, or even the kth largest element can be done in linear time (often constant time) using heaps.

3. **Priority Queues**: Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority. Schedulers, timers

4. **Graph algorithms**: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal

5. Because of the lack of references, the operations are faster than a binary tree. In addition, some more complicated heaps (such as binomial) can be merged efficiently, which is not easy to do for a binary tree.

Problems in Heap

Kth Smallest in a Min Heap

Just call DeleteMin() operation K-1 times and then again call DeleteMin() this last operation will give Kth smallest value. Time Complexity O(KlogN)

Kth Largest in a Max Heap

Just call DeleteMax() operation K-1 times and then again call DeleteMax () this last operation will give Kth smallest value.   Time Complexity O(KlogN)

100 Largest in a Stream

There are billions of integers coming out of a stream some getInt() function is providing integers

one by one. How would you determine the largest 100 numbers?

Solution: Large hundred (or smallest hundred etc.), such problems are solved very easily using a Heap. In this case, we will create a min heap.
1. First from 100 first integers build a min heap.
2. Then for each coming integer compare if it is greater than the top of the min heap.
3. If not, then look for next integer. If yes, then remove the top min value from the min heap, insert the new value at the top of the heap, use procolateDown, and move it to its proper position, down the heap.
4. Every time you have largest 100 values stored in your head

Merge two Heap

How can we merge two heaps?

Solution: There is no single solution for this. Let us suppose the size of the bigger heap is N and the size of the smaller heap is M.
1. If both heaps are comparable in size, then put both heap arrays in same bigger Arrays. Alternatively, in one of the Arrays if they are big enough, then apply CreateHeap() function which will take theta(N+M) time.
2. If M is much smaller than N then add() each element of M array one by one to N heap. This will take O(MlogN) the worst case or O(M) the best case.

Get Median function

Give a data structure that will provide median of given values in constant time.

Solution: We will use two heaps, one min heap and other max heap. Max heap will contain the first half of data and min heap will contain the second half of the data. Max heap will contain the smaller half of the data and its max value that is at the top of the heap will be the median contender. Similarly, the Min heap will contain the larger values of the data and its min value that is at its top will contain the median contender. We will keep track of the size of heaps. Whenever we insert a value to heap, we will make sure that the size of two heaps differs by max one element, otherwise we will pop one element from one and insert into another to keep them balanced.

Example 10.5:
```
class MedianHeap
    def initialize()
        @minHeap = Heap.new([])
        @maxHeap = Heap.new([],false)
    end


    def insert(value)
        if @maxHeap.size() == 0 or @maxHeap.peek() >= value then
            @maxHeap.add(value)
        else
            @minHeap.add(value)
```

```ruby
            end
            #size balancing
            if @maxHeap.size() > @minHeap.size() + 1 then
                value = @maxHeap.remove()
                @minHeap.add(value)
            end
            if @minHeap.size() > @maxHeap.size() + 1 then
                value = @minHeap.remove()
                @maxHeap.add(value)
            end
        end

        def median()
            if @maxHeap.size() == 0 and @minHeap.size() == 0 then
                raise StandardError, "EmptyException"
            end
            if @maxHeap.size() == @minHeap.size() then
                return (@maxHeap.peek() + @minHeap.peek()) / 2
            elsif @maxHeap.size() > @minHeap.size() then
                return @maxHeap.peek()
            else
                return @minHeap.peek()
            end
        end
end

# Testing code
arr = [1, 9, 2, 8, 3, 7, 4, 6, 5, 1]
hp = MedianHeap.new()
i = 0
while i < 10
    hp.insert(arr[i])
    print "\n Median after insertion of " , arr[i] , " is " , hp.median()
    i += 1
end
```

Traversal in Heap

Heaps are not designed to traverse, to find some element. They are made to get min or max element quickly. Still if you want to traverse a heap just traverse the array sequentially. This traversal will be level order traversal. This traversal will have linear Time Complexity.

Deleting Arbiter element from Min Heap

Again, heap is not designed to delete an arbitrary element, still if you want to do so. Find the element by linear search in the heap array. Replace it with the value stored at the end of the Heap value. Reduce the size of the heap by one.   Compare the new inserted value with its parent. If its value is smaller than the parent value, then percolate up. Else if its value is greater

than its left and right child then percolate down. Time Complexity is **O(logn)**

Deleting Kth element from Min Heap

Again, heap is not designed to delete an arbitrary element, still if you want to do so. Replace the kth value with the value stored at the end of the Heap value. Reduce the size of the heap by one.   Compare the new inserted value with its parent. If its value is smaller than the parent value, then percolate up. Else if its value is greater than its left and right child then percolate down. Time Complexity is **O(logn)**

Print value in Range in Min Heap

Linearly traverse through the heap and print the value that are in the given range.

Exercise

1. What is the worst-case runtime Complexity of finding the smallest item in a min-heap?

2. Find max in a min heap.
   Hint: normal search in the complete array. There is one more optimization you can search from the mid of the array at index N/2

3. What is the worst-case time Complexity of finding the largest item in a min-heap?

4. What is the worst-case time Complexity of deleteMin in a min-heap?

5. What is the worst-case time Complexity of building a heap by insertion?

6. Is a heap full or complete binary tree?

7. What is the worst time runtime Complexity of sorting an array of N elements using heapsort?

8. In given sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9
   a. Draw a binary Min-heap by inserting the above numbers one by one
   b. Also draw the tree that will be formed after calling Dequeue() on this heap

9. In given sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9
   a. Draw a binary Max-heap by inserting the above numbers one by one
   b. Also draw the tree that will be formed after calling Dequeue() on this heap

10. In given sequence of numbers: 3, 9, 5, 4, 8, 1, 5, 2, 7, 6. Construct a Min-heap by calling CreateHeap function.

11. Show an array that would be the result after the call to deleteMin() on this heap

12. In given array: [3, 9, 5, 4, 8, 1, 5, 2, 7, 6]. Apply heapify over this to make a min heap and sort the elements in decreasing order?

13. In Heap-Sort once a root element has been put in its final position, how much time, does it take to re-heapify the structure so that the next removal can take place? In other words, what is the Time Complexity of a single element removal from the heap of size N?

14. What do you think the overall Time Complexity for heapsort is? Why do you feel this way?

CHAPTER 11: HASH-TABLE

Introduction

In the searching chapter, we have gone through into various searching techniques. Consider a problem of searching a value in an array. If the array is not sorted then we have no other option, but to look into each element one by one so the searching Time Complexity will be **O(n)**. If the array is sorted then we can search the value in **O(logn)** logarithmic time using binary search.

What if the possible location / index of the value that we are looking in the array is returned by a magic function in constant time? We can directly go into that location and tell whether the value we are searching for is present or not in just **O(1)** constant time. Such a function is called a Hash function.

The process of storing objects using a hash function is as follows:
1. Create a list of size M to store objects; this list is called Hash-Table.
2. Find a hash code of an object by passing it through the hash function.
3. Take module of hash code by the size of Hashtable to get the index of the table where objects will be stored.
4. Finally store these objects in the designated index.

The process of searching objects in Hash-Table using a hash function is as follows:
1. Find a hash code of the object we are searching for by passing it through the hash function.
2. Take module of hash code by the size of Hashtable to get the index of the table where objects are stored.
3. Finally, retrieve the object from the designated index.

Hash-Table

A Hash-Table is a data structure that maps keys to values. Each position of the Hash-Table is called a slot. The Hash-Table uses a hash function to calculate an index of a list. We use the Hash-Table when the number of keys, actually stored, is small relatively to the number of possible keys.

Hash-Table Abstract Data Type (ADT)

ADT of Hash-Table contains the following functions:
1. Insert(x), add object x to the data set.
2. Delete(x), delete object x from the data set.
3. Search(x), search object x in data set.

Hash Function

A hash function is a function that generates an index in a table for a given object.
An ideal hash function that generate a unique index for every object is called the perfect hash function.

Example 11.1: Most simple hash function
```
def computeHash(key) #division method
    return key % @tableSize
end
```

There are many hash functions. The above function is a very simple hash function. Various hash generation logics will be added to this function to generate a better hash.

Collisions

When a hash function generates the same index for the two or more different objects, the problem is known as the collision. Ideally, hash function should return a unique address for each key, but practically it is not possible.

Properties of good hash function:

1. It should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them.
2. Choose a hash function, which can be computed quickly and returns values within the range of the Hash-Table.
3. Chose a hash function with a good collision resolution algorithm which can be used to compute alternative index if the collision occurs.
4. Choose a hash function, which uses the necessary information provided in the key.
5. It should have high load factor for a given set of keys.

Load Factor

Load factor = Number of elements in Hash-Table / Hash-Table size

Based on the above definition, Load factor tells whether the hash function is distributing the keys uniformly or not. Therefore, it helps in determining the efficiency of the hashing function. It also works as decision parameter when we want to expand or rehash the existing Hash-Table entries.

Collision Resolution Techniques

Hash collisions are practically unavoidable when hashing large number of objects. Techniques that are used to find the alternate location in the Hash-Table is called collision resolution. There are a number of collision resolution techniques to handle the collision in hashing.

Most common and widely used techniques are:
· Open addressing
· Separate chaining

Hashing with Open Addressing

When using linear open addressing, the Hash-Table is represented by a one-dimensional list with indices that range from 0 to the desired table size-1.

One method of resolving collision is to look into a Hash-Table and find another free slot to hold the object that have caused the collision. A simple way is to move from one slot to another in some sequential order until we find a free space. This collision resolution process is called Open Addressing.

Linear Probing

In Linear Probing, we try to resolve the collision of an index of a Hash-Table by sequentially searching the Hash-Table free location. Let us assume, if k is the index retrieved from the hash function. If the kth index is already filled then we will look for (k+1) %M, then (k+2) %M and so on. When we get a free slot, we will insert the object into that free slot.

Example 11.2: The resolver function of linear probing
```
def resolverFun(i)
    return i
end
```

Quadratic Probing

In Quadratic Probing, we try to resolve the collision of the index of a Hash-Table by quadratically increasing the search index of free location. Let us assume, if k is the index retrieved from the hash function. If the kth index is already filled then we will look for (k+1^2) %M, then (k+2^2) %M and so on. When we get a free slot, we will insert the object into that

free slot.

Example 11.3: The resolver function of quadratic probing
```
def resolverFun2(i)
    return i * i
end
```

Table size should be a prime number to prevent early looping it should not be too close to 2powN

Linear Probing implementation

Example 11.4: Below is a linear probing collision resolution Hash-Table implementation.
```
class HashTable
    def initialize(tSize)
        @EMPTY_NODE = -1
        @LAZY_DELETED = -2
        @FILLED_NODE = 0
        @tableSize = tSize
        @Arr = Array.new(tSize + 1)
        @Flag = Array.new( tSize + 1)
        i = 0
        while i <= tSize
            @Flag[i] = @EMPTY_NODE
            i += 1
        end
    end
    # Other methods.
end
```

Table list size will be 50 and we have defined two constant values EMPTY_NODE and LAZY_DELETED.
```
def computeHash(key)
    return key % @tableSize
end
```

This is the most simple hash generation function, which just takes the modulus of the key.
```
def resolverFun(index)
    return index
end
```

When the hash index is already occupied by some element the value will be placed in some other location to find that new location resolver function is used.

Hash-Table has two component one is table size and another is reference to list.

Example 11.5:

```
def insert(value)
    hashValue = self.computeHash(value)
    i = 0
    while i < @tableSize
        if @Flag[hashValue] == @EMPTY_NODE or @Flag[hashValue] == @LAZY_deleteD
then
            @Arr[hashValue] = value
            @Flag[hashValue] = @FILLED_NODE
            return true
        end
        hashValue += self.resolverFun(i)
        hashValue %= @tableSize
        i += 1
    end
    return false
end
```

An insert node function is used to add values to the list. First hash is calculated. Then we try to place that value in the Hash-Table. We look for empty node or lazy deleted node to insert value. In case insert did not success, we try new location using a resolver function.

Example 11.6:
```
def find(value)
    hashValue = self.computeHash(value)
    i = 0
    while i < @tableSize
        if @Flag[hashValue] == @EMPTY_NODE then
            return false
        end
        if @Flag[hashValue] == @FILLED_NODE and @Arr[hashValue] == value then
            return true
        end
        hashValue += self.resolverFun(i)
        hashValue %= @tableSize
        i += 1
    end
    return false
end
```

FindNode function is used to search values in the array. First hash is calculated. Then we try to find that value in the Hash-Table. We look for over desired value or empty node. In case we find the value that we are looking for, then we return that value or in case it is not found we return -1. We use a resolver function to find the next probable index to search.

Example 11.7:
```
def delete(value)
    hashValue = self.computeHash(value)
    i = 0
```

```
        while i < @tableSize
            if @Flag[hashValue] == @EMPTY_NODE then
                return false
            end
            if @Flag[hashValue] == @FILLED_NODE and @Arr[hashValue] == value then
                @Flag[hashValue] = @LAZY_deleteD
                return true
            end
            hashValue += self.resolverFun(i)
            hashValue %= @tableSize
            i += 1
        end
        return false
end
```

Delete node function is used to delete values from a Hashtable. We do not actually delete the value we just mark that value as LAZY_DELETED. Same as the insert and search we use resolverFun to find the next probable location of the key.

Example 11.8:
```
def display()
    i = 0
    while i < @tableSize
        if @Flag[i] == @FILLED_NODE then
            print "\n Node at index [" , i , " ] :: " , @Arr[i]
        end
        i += 1
    end
end

# Testing code
ht = HashTableSC.new()
ht.insert(100)
print "\nsearch 100 :: " , ht.find(100)
print "\nremove 100 :: " , ht.delete(100)
print "\nsearch 100 :: " , ht.find(100)
print "\nremove 100 :: " , ht.delete(100)
```

Output:
```
search 100 :: true
remove 100 :: true
search 100 :: false
remove 100 :: false
```

Print method print the content of hash table. Main function demonstrating how to use hash table.

Quadratic Probing implementation.

Everything will be same as linear probing implementation only resolver function will be changed.

```
def resolverFun(index)
    return index * index
end
```

Hashing with separate chaining

Another method for collision resolution is based on an idea of putting the keys that collide in a linked list. This method is called separate chaining. To speed up search we use Insertion-Sort or keeping the linked list sorted.
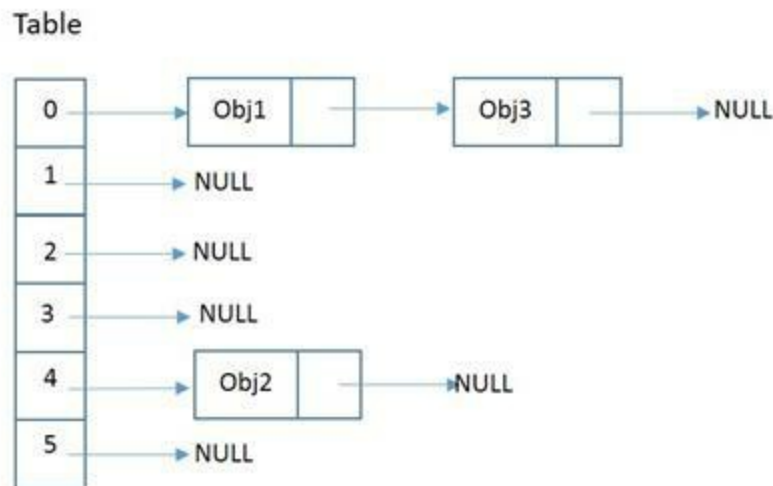
Separate Chaining implementation

'**Example 11.9:** Below is separate chaining implementation of hash tables.

```
class HashTableSC
    class Node
        attr_accessor :value, :next
        def initialize(v, n = nil)
            @value = v
            @next = n
        end
    end #double pointer

    attr_accessor :tableSize, :listArray

    def initialize(tSize = 512)
        @tableSize = tSize
        @listArray = Array.new(tSize + 1, nil)
    end

    # Other methods.

    def ComputeHash(key) #division method
```

```
        return key % @tableSize
end

def resolverFun(i)
        return i
end

def resolverFun2(i)
        return i * i
end

def insert(value)
        index = self.ComputeHash(value)
        @listArray[index] = Node.new(value, @listArray[index])
end

def delete(value)
        index = self.ComputeHash(value)
        head = @listArray[index]
        if head != nil and head.value == value then
            @listArray[index] = head.next
            return true
        end
        while head != nil
            nextNode = head.next
            if nextNode != nil and nextNode.value == value then
                head.next = nextNode.next
                 return true
            else
                head = nextNode
            end
        end
        return false
end

def display()
        i = 0
        while i < tableSize
            print "Printing for index value :: " + i + "List of value printing :: "
            head = @listArray[i]
            while head != nil
                print head.value
                head = head.next
            end
            i += 1
        end
end
```

```
    def find(value)
        index = self.ComputeHash(value)
        head = @listArray[index]
        while head != nil
            if head.value == value then
                return true
            end
            head = head.next
        end
        return false
    end
end

# Testing code
ht = HashTableSC.new()
ht.insert(100)
print "\nsearch 100 :: " , ht.find(100)
print "\nremove 100 :: " , ht.delete(100)
print "\nsearch 100 :: " , ht.find(100)
print "\nremove 100 :: " , ht.delete(100)
```

Output:
```
search 100 :: true
remove 100 :: true
search 100 :: false
remove 100 :: false
```

Note: It is important to note that the size of the "skip" must be such that all the slots in the table will eventually be occupied. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size must be a prime number. This is the reason we are using 11 in our examples.

Problems in Hashing

Anagram solver

An anagram is a word or phrase formed by reordering the letters of another word or phrase.

Example 11.10: Two words are anagram if they are of same size and their characters are same.
```
def isAnagram(str1, str2)
    size1 = str1.size
    size2 = str2.size
    if size1 != size2
        return false
    end
    cm = Counter.new()
    index = 0
    while index < size1
```

```
            ch = str1[index]
            cm.add(ch)
            index += 1
        end

    index = 0
    while index < size2
        ch = str2[index]
        if cm.containsKey(ch) then
            cm.remove(ch)
        else
            return false
        end
        index += 1
    end
    return (cm.size() == 0)
end
```

Remove Duplicate

Solution: We can use a second list or the same list, as the output list. In the following example Hash-Table is used to solve this problem.

Example 11.11: Remove duplicates in a list of numbers
```
def removeDuplicate(str)
    hs = Set.new
    out = ""
    index = 0
    size = str.size
    while index < size
        ch = str[index]
        if hs.include?(ch) == false then
            out = out + ch
            hs.add(ch)
        end
        index += 1
    end
    return out
end
```

Find Missing

Example 11.12: There is a list of integers we need to find the missing number in the list.
```
def findMissing(arr, start, end2)
    hs = Set.new
    index = 0
    size = arr.size
    while index < size
```

```
      i = arr[index]
      hs.add(i)
      index += 1
   end
   curr = start
   while curr <= end2
      if hs.include?(curr) == false then
         return curr
      end
      curr += 1
   end
   return start - 1
end
```

All the elements in the list is added to a HashTable. The missing element is found by searching into HashTable and final missing value is returned.

Print Repeating

Example 11.13: Print the repeating integer in a list of integers.
```
def printRepeating(arr)
   hs = Set.new
   print "\n Repeating elements are:"
   index = 0
   size = arr.size
   while index < size
      val = arr[index]
      if hs.include?(val)
         print "   " , val
      end
      hs.add(val)
      index += 1
   end
end
```

All the values are added to the hash table,   when some value came which is already in the hash table then that is the repeated value.

Print First Repeating

Example 11.14: It is same as the above problem in this we need to print the first repeating number. Care should be taken to find the first repeating number. It should be the one number that is repeating. For example, 1, 2, 3, 2, 1. The answer should be 1 as it is the first number, which is repeating.

```
def printFirstRepeating(arr)
   size = arr.size
   hs = Counter.new()
```

```
    i = 0
    while i < size
        hs.add(arr[i])
        i += 1
    end
    i = 0
    while i < size
        hs.remove(arr[i])
        if hs.containsKey(arr[i]) then
            print "\n First Repeating number is : " , arr[i]
            return
        end
        i += 1
    end
end
```

Add values to the count map the one that is repeating will have multiple count. Now traverse the list again and see if the count is more than one. So that is the first repeating.

Exercise

1. Design a number (ID) generator system that generates numbers between 0-99999999 (8-digits).
   The system should support two functions:
   a. int   getNumber();
   b. int   requestNumber();
   getNumber() function should find out a number that is not assigned, then mark it as assigned and return that number. requestNumber() function checks the number if it is assigned or not. If it is assigned returns 0, else marks it as assigned and return 1.

2. In given large string, find the most occurring words in the string. What is the Time Complexity of the above solution?
   Hint:-
   a. Create a Hashtable which will keep track of   <word, frequency>
   b. Iterate through the string and keep track of word frequency by inserting into Hash-Table.
   c. When we have a new word, we will insert it into the Hashtable with frequency 1. For all repetition of the word, we will increase the frequency.
   d. We can keep track of the most occurring words whenever we are increasing the frequency we can see if this is the most occurring word or not.
   e. The Time Complexity is **O(n)** where n is the number of words in the string and Space Complexity is the O(m) where m is the unique words in the string.

3. In the above question, What if you are given whole work of OSCAR WILDE, most popular playwrights in the early 1890s.
   Hint:-
   a. Who knows how many books are there, let us assume there is a lot and we cannot put everything in memory. First, we need a Streaming Library so that we can read section

by section in each document. Then we need a tokenizer that will give words to our program. In addition, we need some sort of dictionary let us say we will use HashTable.

b. What you need is - 1. A streaming library tokenizer, 2. A tokenizer 3. A hashmap
Method:
1. Use streamers to find a stream of the given words
2. Tokenize the input text
3. If the stemmed word is in hash map, increment its frequency count else add a word to hash map with frequency 1

c. We can improve the performance by looking into parallel computing. We can use the map-reduce to solve this problem. Multiple nodes will read and process multiple documents. Once they are done with their processing, then we can do the reduce operation by merging them.

4. In the above question, What if we want to find the most common PHRASE in his writings.
Hint: - We can keep <phrase, frequency> Hash-Table and do the same process of the 2nd and 3rd problems.

5. Write a hashing algorithm for strings.
Hint: Use Horner's method

```
def hornerHash(key, tableSize)
  size = key.size
  h = 0
  i = 0
  while i < size
     h = (32 * h + key[i]) % tableSize
     i += 1
  end
  return h
end
```

6. Pick two data structures to use in implementing a Map. Describe lookup, insert, & delete operations. Give time & Space Complexity for each. Give pros & cons for each.
Hint:-
a) Linked List
  I.   Insert is **O(1)**
  II.   Delete is **O(1)**
  III.   Lookup is **O(1)** auxiliary and **O(N)** worst case.
  IV.   Pros: Fast inserts and deletes, can use for any data type.
  V.   Cons: Slow lookups.

b) Balanced Search Tree (RB Tree)
  I.   Insert is **O(logn)**
  II.   Delete is **O(logn)**
  III.   Lookup is **O(logn)**
  IV.   Pros: Reasonably fast inserts/deletes and lookups.
  V.   Cons: Data needs to have order defined on it.

CHAPTER 12: GRAPHS

Introduction

In this chapter, we will study about Graphs. Graphs can be used to represent many interesting things in the real world. Flights from cities to cities, rods connecting various towns and cities. Even the sequence of steps that we take to become ready for jobs daily, or even a sequence of classes that we take to become a graduate in computer science. Once we have a good representation of the map, then we use a standard graph algorithms to solve many interesting problems of real life.

The flight connection between major cities of India can also be represented by the graph given below. Each node is a city and each edge is a straight flight path from one city to another. You may want to go from Delhi to Chennai, if this data is given in good representation to a computer, through graph algorithms the computer may propose shortest, quickest or cheapest path from source to destination.
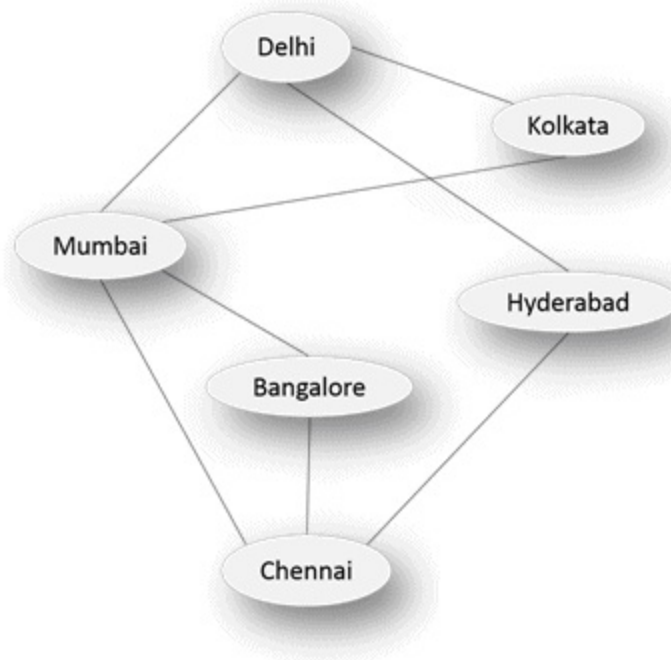
Google map that we use is also a big graph of lots of nodes and edges. That suggests shortest and quickest path to the user.

Graph Definitions
A Graph is represented by G where G = (V, E), where V is a finite set of points called **Vertices** and E is a finite set of **Edges**.

Each **edge** is a tuple (u, v) where u, v $\in$ V. There can be a third component weight to the tuple. **Weight** is cost to go from one vertex to another.

Edge in a graph can be directed or undirected. If the edges of graph are one way, it is called **Directed graph** or **Digraph**. The graph whose edges are two ways is called **Undirected graph**
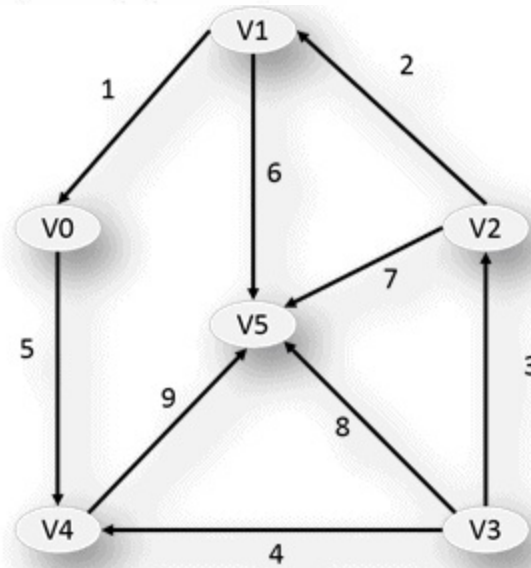
or just graph.

A **Path** is a sequence of edges between two vertices. The length of a path is defined as the sum of the weight of all the edges in the path.

Two vertices u and v are **adjacent** if there is an edge whose endpoints are u and v.

In the below graph:
V = {V1, V2, V3, V4, V5, V6, V7, V8, V9},

$$E = \begin{Bmatrix} (V1,V0,1),(V2,V1,2),(V3,V2,3),(V3,V4,4),(V5,V4,5), \\ (V1,V5,6),(V2,V5,7),(V3,V5,8),(V4,V5,9) \end{Bmatrix}$$

The **in-degree** of a vertex v, denoted by indeg(v) is the number of incoming edges to the vertex v. The **out-degree** of a vertex v, denoted by outdeg(v) is the number of outgoing edges of a vertex v. The **degree** of a vertex v, denoted by deg(v) is the total number of edges whose one endpoint is v.

deg(v) = Indeg (v) + outdeg (v)

In the above graph
deg(V4) =3, indeg(V4) =2 and outdeg(V4) =1

A **Cycle** is a path that starts and ends at the same vertex and includes at least one vertex.

An edge is a **Self-Loop** if two if its two endpoints coincide. This is a form of a cycle.

A vertex v is **Reachable** from vertex u or "u reaches v" if there is a path from u to v. In an undirected graph if v is reachable from u then u is reachable from v. However, in a directed graph it is possible that u reaches v but there is no path from v to u.

A graph is **Connected** if for any two vertices there is a path between them.

A **Forest** is a graph without cycles.

A **Sub-Graph** of a graph G is a graph whose vertices and edges are a subset of the vertices and edges of G.

A **Spanning Sub-Graph** of G is a graph that connects all the vertices of G.

A **tree** is an acyclic connected graph.

A **Spanning tree** of a graph is a tree that connects all the vertices of the graph. Since a Spanning-Tree is a tree, so it should not have any cycle.

Graph   Spanning Tree
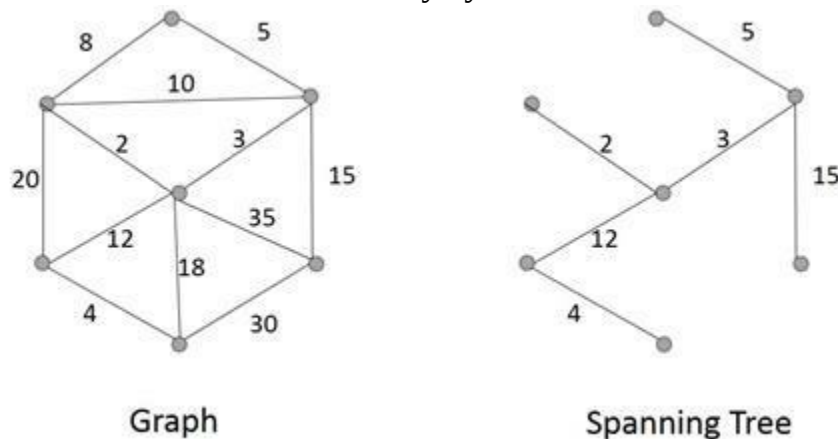
Graph Representation

In this section, we have introduced the data structure for representing a graph. In the below representations we maintain a collection to store edges and vertices of the graph.

Adjacency Matrix

One of the ways to represent a graph is to use two-dimensional matrix. Each combination of row and column represents a vertex in the graph. The value stored at the location row v and column w is the edge from vertex v to vertex w. The nodes that are connected by an edge are called adjacent nodes. This matrix is used to store adjacent relation so it is called the Adjacency Matrix. In the given below diagram, we have a graph and its Adjacency matrix.
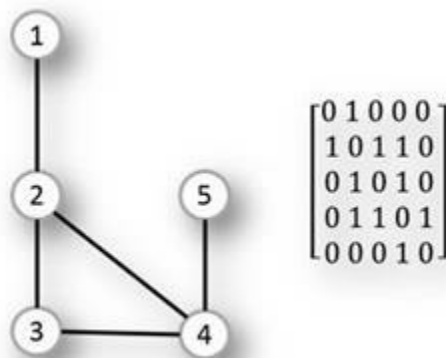
In the above graph, each node has weight 1 so the adjacency matrix has just 1s or 0s. If the edges are of different weights then that weight will be filled in the matrix.

Pros: Adjacency matrix implementation is simple. Adding/Removing an edge between two vertices is just **O(1)**. Query if there is an edge between two vertices is also **O(1)**

Cons: It always consumes $O(V^2)$ space, which is an inefficient way to store when a graph is a sparse.

Sparse Matrix: In a huge graph, each node is connected with fewer nodes. So most of the places in adjacency matrix remain empty. Such matrix is called sparse matrix. In most of the real world problems adjacency matrix is not a good choice for sore graph data.

Adjacency List

A more space efficient way of storing graph is adjacency list. In adjacency we have list of references to a linked list. Each reference corresponds to vertices in a graph.   Each reference will then point to the vertices that are connected to it and stores this as a list.

In the below diagram node 2 is connected to 1, 3 and 4. Therefore, the reference at location 2 is pointing to a list that contains 1, 3 and 4.
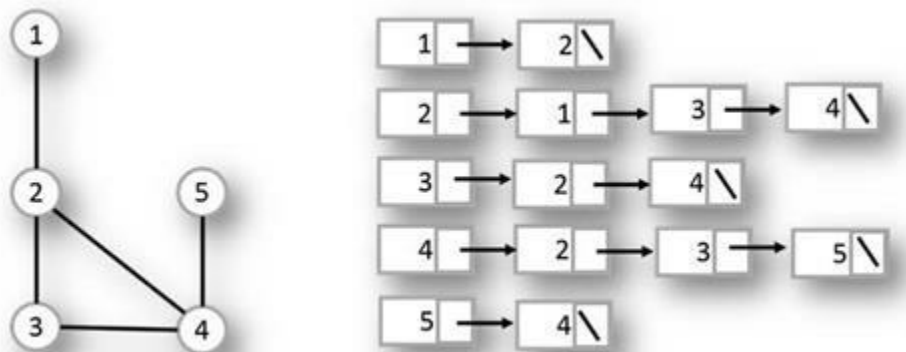
The adjacency list helps us to represent a sparse graph. An adjacency list representation also allows us to find all the vertices that are directly connected to any vertices by just one link list scan. In all our programs, we are going to use the adjacency list to store the graph.

Example 12.1: adjacency list representation of an undirected graph

```ruby
class AdjNode
    attr_accessor :source, :destination, :cost, :next
    def initialize(src, dst, cst=1)
        @source = src
        @destination = dst
        @cost = cst
        @next = nil
    end

    def compare(other)
        return @cost - other.cost
    end
end

class AdjList
    attr_accessor :head
    def initialize()
        @head = nil
```

```ruby
        end
end

class Graph
    attr_accessor :size, :list
    def initialize(cnt = 0)
        @size = cnt
        @list = Array.new(cnt, nil)
        i = 0
        while i < cnt
            @list[i] = AdjList.new()
            @list[i].head = nil
            i += 1
        end
    end

    def AddEdge(source, destination, cost = 1)
        node = AdjNode.new(source, destination, cost)
        node.next = @list[source].head
        @list[source].head = node
    end

    def AddBiEdge(source, destination, cost = 1) #bi directional edge
        AddEdge(source, destination, cost)
        AddEdge(destination, source, cost)
    end

    def Print()
        i = 0
        while i < size
            ad = @list[i].head
            if ad != nil then
                print "Vertex " , i , " is connected to : \n"
                while ad != nil
                    print ad.destination , " "
                    ad = ad.next
                end
                puts ""
            end
            i += 1
        end
    end
end
```

Graph traversals

The **Depth first search (DFS)** and **Breadth first search (BFS)** are the two algorithms used to traverse a graph. These same algorithms can also be used to find some node in the graph, find if

a node is reachable etc.

Traversal is the process of exploring a graph by examining all its edges and vertices.
A list of some of the problems that are solved using graph traversal are:
1. Determining a path from vertex u to vertex v, or report an error if there is no such path.
2. Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph.
3. Testing if a graph G is connected.
4. Finding a spanning tree of a Graph.
5. Finding if there is some cycle in the graph.

Depth First Traversal

We start the DFS algorithm from starting point and go into depth of graph until we reach a dead end and then move up to parent node (Backtrack). In DFS, we use stack to get the next vertex to start a search. Alternatively, we can use recursion (system stack) to do the same.

Depth First Traversal
0, 1, 4, 7 , 5, 2, 6, 3
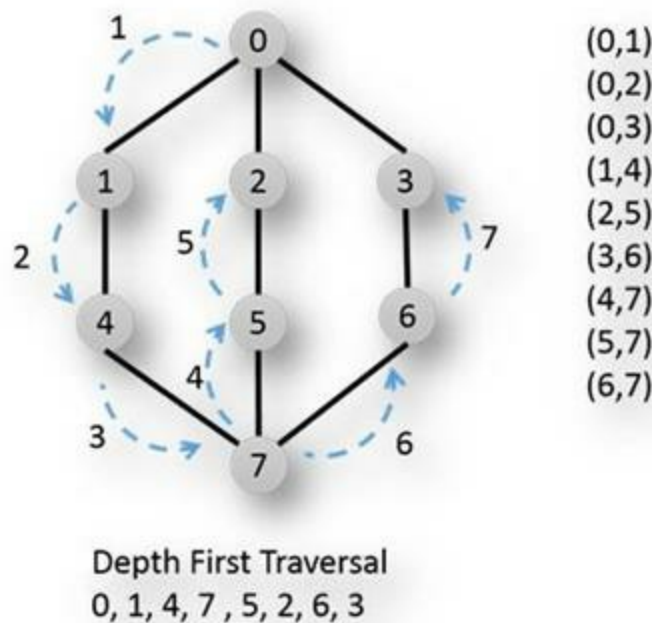
Algorithm steps for DFS
1. Push the starting node in the stack.
2. Loop until the stack is empty.
3. Pop node from the stack inside loop, call this node current.
4. Process the current node. //Print, etc.
5. Traverse all the child nodes of the current node and push them into stack.
6. Repeat steps 3 to 5 until the stack is empty.

Stack based implementation of DFS

Example 12.2:
```
def DFSStack(gph)
    size = gph.size
    visited = Array.new( size)
    stk = []
```

```
        i = 0
        while i < size
            visited[i] = 0
            i += 1
        end
        visited[0] = 1
        stk.push(0)
        while stk.size > 0
            curr = stk.pop()
            head = gph.list[curr].head
            while head != nil
                if visited[head.destination] == 0 then
                    visited[head.destination] = 1
                    puts "visited dfs stack #{head.destination}"
                    stk.push(head.destination)
                end
                head = head.next
            end
        end
end
```

Recursion based implementation of DFS

Example 12.3:
```
def DFS(gph)
    size = gph.size
    visited = Array.new( size)
    i = 0
    while i < size
        visited[i] = 0
        i += 1
    end
    i = 0
    while i < size
        if visited[i] == 0 then
            visited[i] = 1
            DFSRec(gph, i, visited)
        end
        i += 1
    end
end


def DFSRec(gph, index, visited)
    head = gph.list[index].head
    while head != nil
        if visited[head.destination] == 0 then
            visited[head.destination] = 1
```

```
            #puts "visited stk rec #{head.destination}"
          DFSRec(gph, head.destination, visited)
       end
       head = head.next
    end
end
```

Breadth First Traversal

In BFS algorithm, a graph is traversed in layer-by-layer fashion. The graph is traversed, closer to the starting point. The queue is used to implement BFS.

Algorithm steps for BFS
1. Push the starting node into the Queue.
2. Loop until the Queue is empty.
3. Remove a node from the Queue inside loop, and call this node current.
4. Process the current node.//print etc.
5. Traverse all the child nodes of the current node and push them into Queue.
6. Repeat steps 3 to 5 until Queue is empty.
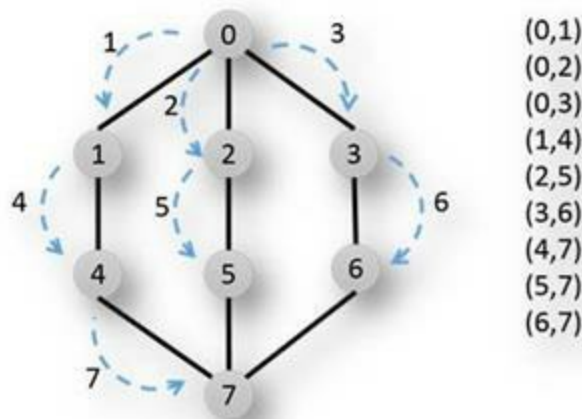
Breadth First Traversal
0, 1, 2, 3, 4, 5, 6, 7

Example 12.4:
```
def BFS(gph)
    size = gph.size
    visited = Array.new( size)
    i = 0
    while i < size
        visited[i] = 0
        i += 1
    end
    i = 0
    while i < size
        if visited[i] == 0 then
            BFSQueue(gph, i, visited)
        end
        i += 1
```

```
        end
end

def BFSQueue(gph, index, visited)
    que = Queue.new()
    visited[index] = 1
    que.push(index)
    while que.size > 0
        curr = que.pop()
        head = gph.list[curr].head
        while head != nil
            if visited[head.destination] == 0 then
                puts "visited #{head.destination}"
                visited[head.destination] = 1
                que.push(head.destination)
            end
            head = head.next
        end
    end
end
```

A runtime analysis of DFS and BFS traversal is O(n+m) time, where n is the number of edges reachable from source node and m is the number of edges incident on s.

The following problems have O(m+n) time performance:
1. Determining a path from vertex u to vertex v, or report an error if there is no such path.
2. Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph.
3. Testing of a graph G is connected.
4. Finding a spanning tree of a Graph.
5. Finding if there is some cycle in the graph.

Problems in Graph

Determining a path from vertex u to vertex v

IF there is a path from u to v and we perform DFS from u then v must be visited. Moreover, if there is no path then report an error.

Example 12.5:
```
def PathExist(gph, source, destination)
    size = gph.size
    visited = Array.new( size)
    i = 0
    while i < size
        visited[i] = 0
        i += 1
    end
```

```
        visited[source] = 1
        DFSRec(gph, source, visited)
        return visited[destination]
end
```

Given a starting vertex s, finding the minimum number of edges from vertex s to all the other vertices of the graph

Look for single source shortest path algorithm for each edge cost as 1 unit.

Testing of a graph is connected.

IF there is a path from u to v and we are doing DFS from u then v must be visited. Moreover, if there is no path then report an error.

Example 12.6:
```
def isConnected(gph)
    size = gph.size
    visited = Array.new( size)
    i = 0
    while i < size
        visited[i] = 0
        i += 1
    end
    visited[0] = 1
    DFSRec(gph, 0, visited)
    i = 0
    while i < size
        if visited[i] == 0 then
            return false
        end
        i += 1
    end
    return true
end
```

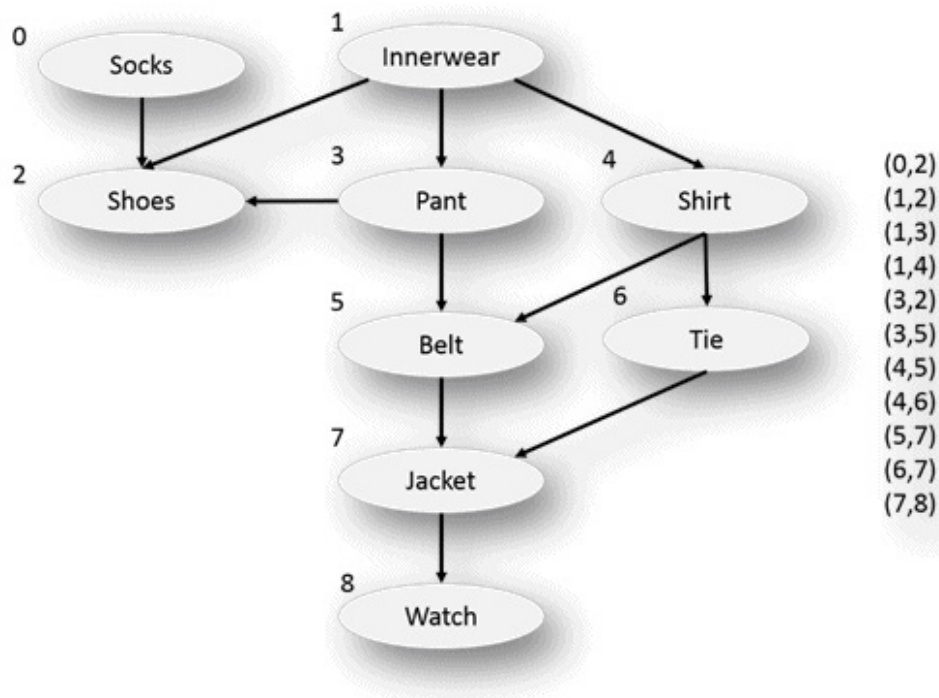## Finding if there is some cycle in the graph.

Modify DFS problem and get this done.

Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a directed graph with no cycle. A DAG represents relationship, which is more general than a tree. Below is an example of DAG, this is how someone becomes ready for work. There are N other real life examples of DAG such as coerces selection to be a graduate from college

```
0    Socks         1    Innerwear
2    Shoes    3    Pant         4    Shirt      (0,2)
                                               (1,2)
                                               (1,3)
                                               (1,4)
          5    Belt    6    Tie                (3,2)
                                               (3,5)
                                               (4,5)
                                               (4,6)
          7    Jacket                          (5,7)
                                               (6,7)
          8    Watch                           (7,8)
```

Topological Sort

A topological sort is a method of ordering the nodes of a directed graph in which nodes represent activities and the edges represent dependency among those tasks. For topological sorting to work it is required that the graph should be a DAG which means it should not have any cycle. Just use DFS to get topological sorting.

Example 12.7:
```ruby
def TopologicalSort(gph)
    stk = []
    size = gph.size
    visited = Array.new(size)
    i = 0
    while i < size
        visited[i] = 0
        i += 1
    end
    i = 0
    while i < size
        if visited[i] == 0 then
            visited[i] = 1
            TopologicalSortDFS(gph, i, visited, stk)
        end
        i += 1
    end
    print "topology order is : "
    while stk.size != 0
        print " " , stk.pop()
    end
    puts ""
end
```

```
def TopologicalSortDFS(gph, index, visited, stk)
    head = gph.list[index].head
    while head != nil
        if visited[head.destination] == 0 then
            visited[head.destination] = 1
            TopologicalSortDFS(gph, head.destination, visited, stk)
        end
        head = head.next
    end
    stk.push(index)
end
```

Topology sort is DFS traversal of topology graph. First, the children of node are added to the stack then only the current node is added. So the sorting order is maintained. Reader is requested to run some examples to understand this algo.
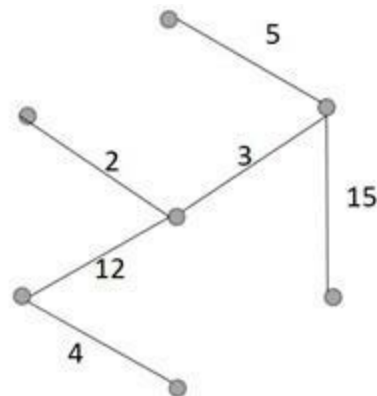
Minimum Spanning Trees (MST)

A Spanning Tree of a graph G is a tree that contains all the edges of the Graph G.
A Minimum Spanning Tree is a spanning-tree whose sum of length/weight of edges is minimum as possible.

For example, if you want to setup communication between a set of cities, then you may want to use the least amount of wire as possible. MST can be used to find the network path and wire cost estimate.
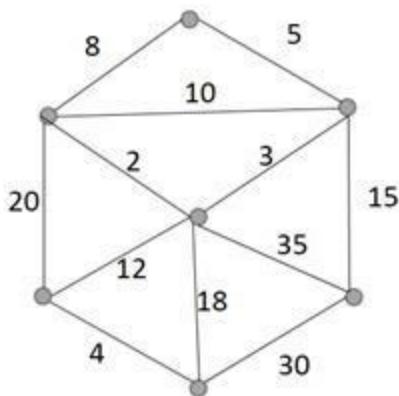
Graph   Minimum Spanning Tree

Prim's Algorithm for MST

Prim's algorithm grows a single tree T, one edge at a time, until it becomes a spanning tree.
We initialize T with zero edges and U with single node. Where T is spanning tree edges set and U is spanning tree vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other

not in us. Since each edge adds one new vertex to U, after n − 1 additions, U contains all the vertices of the spanning tree and T becomes a spanning tree.

Example 12.8:
```
// Returns the MST by Prim's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST
Algorithm Prim(G)
    T = {}
    Let r be any vertex in G
    U = {r}
    for i = 1 to |V| - 1 do
        e = minimum-weight edge (u, v)
            With u in U and v in V-U
        U = U + {v}
        T = T + {e}
    return T
```

Prim's Algorithm using a priority queue (min heap) to get the closest fringe vertex
Time Complexity will be O(m log n) where n vertices and m edges of the MST.

Example 12.9:
```
def Prims(gph)
    previous = Array.new(gph.size)
    dist = Array.new(gph.size)
    source = 1
    i = 0
    while i < gph.size
        previous[i] = -1
        dist[i] = 999999
        i += 1
    end
    dist[source] = 0
    previous[source] = -1
    queue = PriorityQueue.new([])
    node = AdjNode.new(source, source, 0)
    queue.add(node)
    while queue.size() != 0
        node = queue.peek()
        queue.remove()
        if dist[node.destination] < node.cost then
            next
        end
        dist[node.destination] = node.cost
        previous[node.destination] = node.source
        adl = gph.list[node.destination]
        adn = adl.head
        while adn != nil
```

```
            if previous[adn.destination] == -1 then
                node = AdjNode.new(adn.source, adn.destination, adn.cost)
                queue.add(node)
            end
            adn = adn.next
        end
    end
    # Printing result.
    size = gph.size
    i = 0
    while i < size
        if dist[i] == 999999 then
            print " node id " , i , " prev " , previous[i] , " distance : Unreachable \n"
        else
            print " node id " , i , " prev " , previous[i] , " distance : " , dist[i] ,"\n"
        end
        i += 1
    end
end

# Testing code
gph = Graph.new(9)
gph.AddBiEdge(0, 2, 1)
gph.AddBiEdge(1, 2, 5)
gph.AddBiEdge(1, 3, 7)
gph.AddBiEdge(1, 4, 9)
gph.AddBiEdge(3, 2, 2)
gph.AddBiEdge(3, 5, 4)
gph.AddBiEdge(4, 5, 6)
gph.AddBiEdge(4, 6, 3)
gph.AddBiEdge(5, 7, 5)
gph.AddBiEdge(6, 7, 7)
gph.AddBiEdge(7, 8, 17)
Prims(gph)
```

Kruskal's Algorithm

Kruskal's Algorithm repeatedly chooses the smallest-weight edge that does not form a cycle.
Sort the edges in non-decreasing order of cost: c (e1) ≤ c (e2) ≤ · · · ≤ c (em).
Set T to be the empty tree. Add edges to tree one by one, if it does not create a cycle.

Example 12.10:
```
// Returns the MST by Kruskal's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST
Algorithm Kruskal(G)
    Sort the edges E by their weights
    T = { }
```

```
while |T | + 1 < |V | do
    e = next edge in E
    if T + {e} does not have a cycle then
        T = T + {e}
return T
```

Kruskal's Algorithm is O(E log V) using efficient cycle detection.

Shortest Path Algorithms in Graph

Single Source Shortest Path

For a graph G= (V, E), the single source shortest path problem is to find the shortest path from a given source vertex s to all the vertices of V.

Single Source Shortest Path for unweighted Graph.

Find single source shortest path for unweighted graph or a graph with all the vertices of same weight.

Example 12.11:
```
def ShortestPath(gph, source) # unweighted graph
    size = gph.size
    distance = Array.new( size)
    path = Array.new( size)
    que = Queue.new()
    i = 0
    while i < size
        distance[i] = -1
        i += 1
    end
    que.push(source)
    distance[source] = 0
    while que.size > 0
        curr = que.pop()
        head = gph.list[curr].head
        while head != nil
            if distance[head.destination] == -1 then
                distance[head.destination] = distance[curr] + 1
                path[head.destination] = curr
                que.push(head.destination)
            end
            head = head.next
        end
    end
    i = 0
    while i < size
        print path[i] , " to " , i , " weight " , distance[i], "\n"
```

```
        i += 1
    end
end
```

Dijkstra's algorithm

Dijkstra's algorithm is used for single-source shortest path problem for weighted edges with no negative weight. Given a weighted connected graph G, find shortest paths from the source vertex s to each of the other vertices. Dijkstra's algorithm is similar to prims algorithm. It maintains a set of nodes for which shortest path is known.
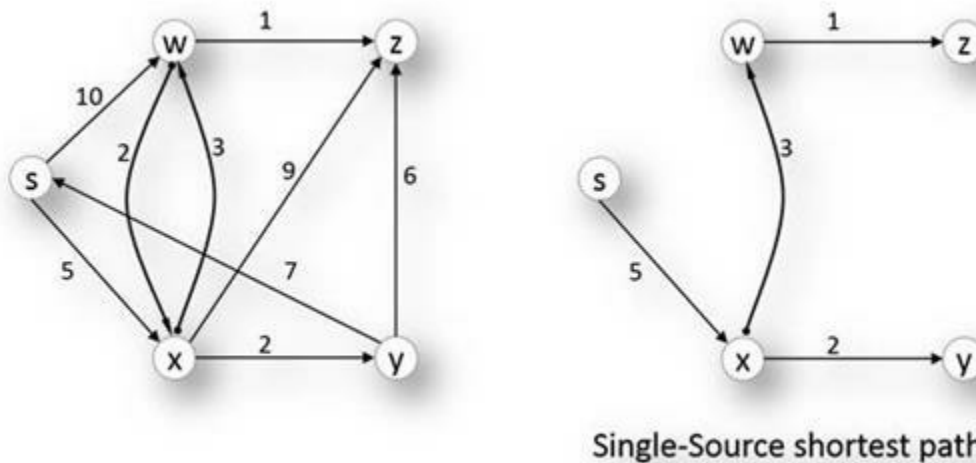
Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning as we do not know the actual path to the nodes and parents of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)

At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the distance and parent list fully populated.

Example 12.12:
```
// Solves SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph G = (V, E)
// with no negative weights, and source vertex v
// Output: The length and path from s to every v

Algorithm Dijkstra(G, s)
for each v in V do
    D[v] = infinite // Unknown distance
    P[v] = null    //unknown previous node
    add v to PQ     //adding all nodes to priority queue

D[source] = 0 // Distance from source to source

while (PQ is not empty)
    u = vertex from PQ with smallest D[u]
    remove u from PQ
```

```
    for each v adjacent from u do
        alt = D[u] + length ( u , v)
        if   alt < D[v] then
            D[v] = alt
            P[v] = u
Return D[] , P[]
```

Time Complexity will be O(|E|log|V|)

Note: Dijkstra's algorithm does not work for graphs with negative edges weight.
Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Example 12.13:
```
def Dijkstra(gph, source)
    previous = Array.new(gph.size)
    dist = Array.new(gph.size)
    i = 0
    while i < gph.size
        previous[i] = -1
        dist[i] = 999999
        i += 1
    end #infinite
    dist[source] = 0
    previous[source] = -1
    queue = PriorityQueue.new([])
    node = AdjNode.new(source, source, 0)
    queue.add(node)
    while queue.size != 0
        node = queue.peek()
        queue.remove()
        adl = gph.list[node.destination]
        adn = adl.head
        while adn != nil
            alt = adn.cost + dist[adn.source]
            if alt < dist[adn.destination] then
                dist[adn.destination] = alt
                previous[adn.destination] = adn.source
                node = AdjNode.new(adn.source, adn.destination, alt)
                queue.add(node)
            end
            adn = adn.next
        end
    end
    size = gph.size
    i = 0
    while i < size
        if dist[i] == 999999 then
            print " node id " , i , " prev " , previous[i] , " distance : Unreachable \n"
```

```
        else
            print " node id " , i , "  prev " , previous[i] , " distance : " , dist[i], "\n"
        end
        i += 1
    end
end
```

Bellman Ford Shortest Path

The bellman ford algorithm works even when there are negative weight edges in the graph. It does not work if there is some cycle in the graph whose total weight is negative.

Example 12.14:
```
def BellmanFordShortestPath(gph, source)
    size = gph.size
    distance = Array.new( size)
    path = Array.new( size)
    i = 0
    while i < size
        distance[i] = 999999
        i += 1
    end
    distance[source] = 0
    i = 0
    while i < size - 1
        j = 0
        while j < size
            head = gph.list[j].head
            while head != nil
                newDistance = distance[j] + head.cost
                if distance[head.destination] > newDistance then
                    distance[head.destination] = newDistance
                    path[head.destination] = j
                end
                head = head.next
            end
            j += 1
        end
        i += 1
    end
    i = 0
    while i < size
        print path[i] , " to " , i , " weight " , distance[i], "\n"
        i += 1
    end
end
```

All Pairs Shortest Paths

Given a weighted graph G(V, E), the all pair shortest path problem is used to find the shortest path between all pairs of vertices u, v ϵ V. Execute n instances of single source shortest path algorithm for each vertex of the graph.

The complexity of this algorithm will be $O(n^3)$

Exercise

1.   In the various path-finding algorithms, we have created a path array that just stores immediate parent of a node, print the complete path for it.

2. All the functions are implemented considering as if the graph is represented by adjacency list. Write all those functions for graph representation as adjacency matrix.

3. In a given start string, end string and a set of strings, find if there exists a path between the start string and end string via the set of strings.

    A path exists if we can get from start string to end the string by changing (no addition/removal) only one character at a time. The restriction is that the new string generated after changing one character has to be in the set.

    Start: "cog"
    End: "bad"
    Set: ["bag", "cag", "cat", "fag", "con", "rat", "sat", "fog"]
    One of the paths: "cog" -> "fog" -> "fag" -> "bag" -> "bad"

CHAPTER 13: STRING ALGORITHMS

Introduction

Every word processing program has a search function in which you can search all occurrences of any particular word in a long text file. The user enter some word that he want to search in text document and the find functionality finds its occurrences in the text. An efficient string-matching algorithm for this problem can increase the efficiency of a text editor.

In this chapter we start with brute-force algorithm for the string-matching problem, which has worst-case running time $O(n*m)$. Then we will look into Robin-Karp string matching algorithm which has much better average time performance but has same worst case running time $O(n*m)$. Then we will be reading KMP or Knuth-Morris-Pratt algorithm, which has a much better run time of $O(n+m)$.

Autocomplete is a feature that suggests a complete word or phrase after a user has just typed it small portion. Autocomplete functionality is commonly found on address bar of web-browsers, text box of search engines and messaging apps. We will be looking into some of the algorithms that can be utilized for autocomplete functionality.

String Matching Algorithms

The various string-matching algorithms that we are going to study in this chapter.
1. Brute Force Search Algorithm
2. Robin-Karp Algorithm
3. Knuth-Morris-Pratt Algorithm
4. Boyer More Algorithm

Brute Force Search Algorithm

We have a pattern that we want to search in the text. The pattern is of length m and the text is of length n. Where m < n.

The brute force search algorithm will check the pattern at all possible value of "i" in the text where the value of "i" ranges from 0 to n-m. The pattern is compared with the text, character by character from left to right. When a mismatch is detected, then pattern is compared by shifting the compare window by one character.

Example 13.1:
```
def BruteForceSearch(text, pattern)
    i = 0
    j = 0
    n = text.size
    m = pattern.size
```

```
    while i <= n - m
        j = 0
        while j < m and pattern[j] == text[i + j]
            j += 1
        end
        if j == m
            return (i)
        end
        i += 1
    end
    return -1
end
```

Worst case Time Complexity of the algorithm is O(m*n), we get the pattern at the end of the text or we do not get the pattern at all.

Best case Time Complexity of this algorithm is O(m), The **average case** Time Complexity of this algorithm is O(n)

Robin-Karp Algorithm

Robin-Karp algorithm is somewhat similar to the brute force algorithm. Because the pattern is compared to each portion of the text of length m. Instead of pattern at each position a hash code is compared, only one comparison is performed. The hash code of the pattern is compared with the hash code of the text window. We try to keep the hash code as unique as possible.

The two features of good hash code are:
· The collision should be excluded as much as possible. A collision occurs when hash code matches, but the pattern does not.
· The hash code of text must be calculated in constant time.

Hash value of text of length m is calculated. Each time we exclude one character and include next character. The portion of text that need to be compared moves as a window of characters. For each window calculation of hash is done in constant time, one member leaves the window and a new number enters a window.

Multiplication by 2 is same as left shift operation. Multiplication by 2^{m-1} is same as left shift m-1 times. We want this multiple times so just store it in variable pow(m) = 2^{m-1}

We do not want to do large multiplication operations so modular operation with a prime number is used.

Example 13.2:
```
def RobinKarp(text, pattern)
    n = text.size
    m = pattern.size
    prime = 101
    powm = 1
```

```
            textHash = 0
        patternHash = 0
        if m == 0 or m > n
            return -1
        end
        i = 0
        while i < m - 1
            powm = (powm << 1) % prime
            i += 1
        end
        i = 0
        while i < m
            patternHash = ((patternHash << 1) + pattern[i].to_i) % prime
            textHash = ((textHash << 1) + text[i].to_i) % prime
            i += 1
        end
        i = 0
        while i <= (n - m)
            if textHash == patternHash
                j = 0
                while j < m
                    if text[i + j] != pattern[j]
                        break
                    end
                    j += 1
                end
                if j == m
                    return i
                end
            end
            textHash = (((textHash - text[i].to_i * powm) << 1) + text[i + m].to_i) % prime
            if textHash < 0
                textHash = (textHash + prime)
            end
            i += 1
        end
        return -1
end
```

Worst case Time Complexity of the algorithm is O(m*n), we get the pattern at the end of the text and we are getting hash code match at each step.

Average case Time Complexity of algorithms is O(n). Most go the time we have good hashing function so we will get hash match only when we had found the pattern.

Knuth-Morris-Pratt Algorithm

There is an inefficiency in the brute force method of string matching. After a shift of the pattern,

the brute force algorithm forgotten all the information about the previous matched symbols. This is because of which its worst case Time Complexity is O(mn).

The Knuth-Morris-Pratt algorithm makes use of this information that is computed in the previous comparison. It never re compares the whole text. It uses pre-processing of the pattern. The pre-processing takes O(m) time and whole algorithm is O(n). So the total running time is O(n + m) and considering the text is large compared to pattern it will be O(n).

Pre-processing step: we try to find the border of the pattern at a different prefix of the pattern.

A **prefix** is a string that comes at the beginning of a string.
A **proper prefix** is a prefix that is not the complete string. Its length is less than the length of the string.
A **suffix** is a string that comes at the end of a string.
A **proper suffix** is a suffix that is not a complete string. Its length is less than the length of the string.
A **border** is a string that is both proper prefix and a proper suffix.

Example 13.3:

```
def KMPPreprocess(pattern, shiftArr)
    m = pattern.size
    i = 0
    j = -1
    shiftArr[i] = -1
    while i < m
        while j >= 0 and pattern[i] != pattern[j]
            j = shiftArr[j]
        end
        i += 1
        j += 1
        shiftArr[i] = j
    end
end
```

We have to loop outer loop for the text and inner loop for the pattern when we have matched the text and pattern mismatch, we shift the text such that the widest border is considered and then the rest of the pattern matching is resumed after this shift. If again a mismatch happens then the next mismatch is taken.
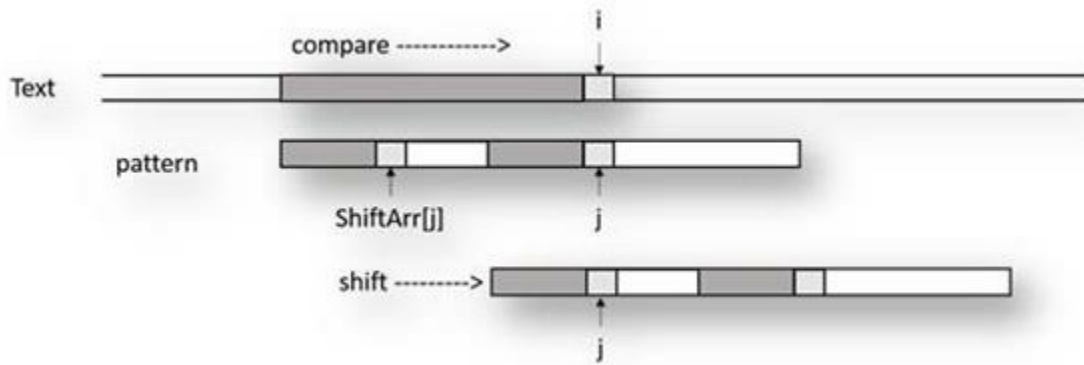
Example 13.4:

```
def KMP(text, pattern)
    i = 0
    j = 0
    n = text.size
    m = pattern.size
    shiftArr = Array.new(m + 1)
    KMPPreprocess(pattern, shiftArr)
    while i < n
        while j >= 0 and text[i] != pattern[j]
            j = shiftArr[j]
        end
        i += 1
        j += 1
        if j == m
            return (i - m)
        end
    end
    return -1
end
```

Example 13.5: Use the same KMP algorithm to find the number of occurrences of the pattern in a text.

```
def KMPFindCount(text, pattern)
    i = 0
    j = 0
    count = 0
    n = text.size
    m = pattern.size
    shiftArr = Array.CreateInstance(System::Int32, m + 1)
    self.KMPPreprocess(pattern, shiftArr)
    while i < n
        while j >= 0 and text[i] != pattern[j]
            j = shiftArr[j]
        end
        i += 1
        j += 1
        if j == m
            count += 1
```

```
            j = shiftArr[j]
        end
    end
    return count
end
```

Dictionary / Symbol Table

A symbol table is a mapping between a string (key) and a value that can be of any type. A value can be an integer such as occurrence count, dictionary meaning of a word and so on.

Binary Search Tree (BST) for Strings

Binary Search Tree (BST) is the simplest way to implement symbol table. Simple strcmp() function can be used to compare two strings. If all the keys are random and the tree is balanced, then on an average key lookup can be done in O(logn) time.
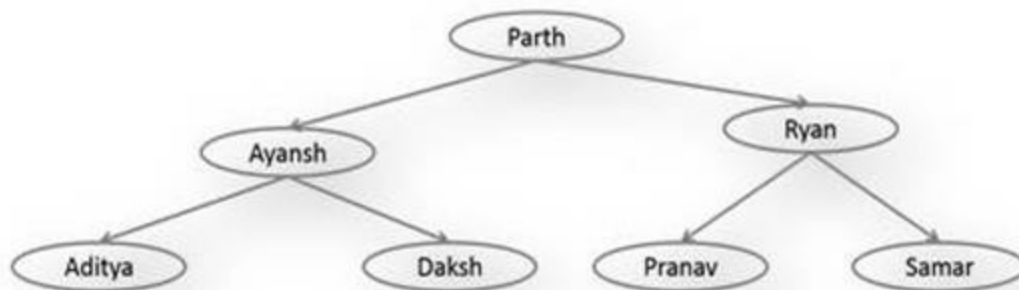
BINARY SEARCH TREE AS DICTIONARY

Below is an implementation of binary search tree to store string as key. This will keep track of the occurrence count of words in a text.

Example 13.6:
```
class StringTree
    class Node
        attr_accessor :value, :count, :lChild, :rChild
        def initialize(v, left = nil, right = nil)
            @value = v
            @count = 1
            @lChild = left
            @rChild = right
        end
    end

    attr_accessor :root
    def initialize()
        @root = nil
    end
```

```ruby
def printTree(curr=@root) # pre order
    if curr != nil
        print " value is ::" , curr.value
        print " count is :: " , curr.count
        self.printTree(curr.lChild)
        self.printTree(curr.rChild)
    end
end

def insert(value)
    @root=insertUtil(value, @root)
end

def insertUtil(value, curr)
    if curr == nil
        curr = Node.new(value)
        curr.count = 1
    else

        compare = ( curr.value <=> value )
        if compare == 0
            curr.count += 1
        elsif compare == 1
            curr.lChild = self.insertUtil(value, curr.lChild)
        else
            curr.rChild = self.insertUtil(value, curr.rChild)
        end
    end
    return curr
end

def freeTree()
    root = nil
end

def find(value, curr=@root)
    if curr == nil
        return false
    end
    compare = ( curr.value <=> value )
    if compare == 0
        return true
    else
        if compare == 1
            return self.find(value, curr.lChild)
        else
            return self.find(value, curr.rChild)
```

```ruby
                end
            end
        end

    def frequency(value, curr=@root)
        if curr == nil
            return 0
        end
        compare = ( curr.value <=> value )
        if compare == 0
            return curr.count
        else
            if compare > 0
                return self.frequency(value, curr.lChild)
            else
                return self.frequency(value, curr.rChild)
            end
        end
    end
end

# Testing code
tt = StringTree.new()
tt.insert("banana")
tt.insert("apple")
tt.insert("mango")
tt.insert("banana")
print ("\nSearch results for apple, banana, grapes and mango :\n")
puts tt.find("apple")
puts tt.find("banana")
puts tt.find("grapes")
puts tt.find("mango")
puts tt.find("banan")
puts tt.frequency("apple")
puts tt.frequency("banana")
puts tt.frequency("mango")
```

Output:
```
Search results for apple, banana, grapes and mango :
true
true
false
true
false
1
2
1
```

Hash-Table

The Hash-Table is another data structure that can be used for symbol table implementation. Below Hash-Table diagram, we can see the name of that person is taken as key, and their meaning is the value of the search. The first key is converted into a hash code by passing it to appropriate hash function. Inside hash function the size of Hash-Table is also passed, which is used to find the actual index where values will be stored. Finally, the value, which is the meaning of name is stored in the Hash-Table, or you can store a reference to the string which stores meaning is stored into the Hash-Table.
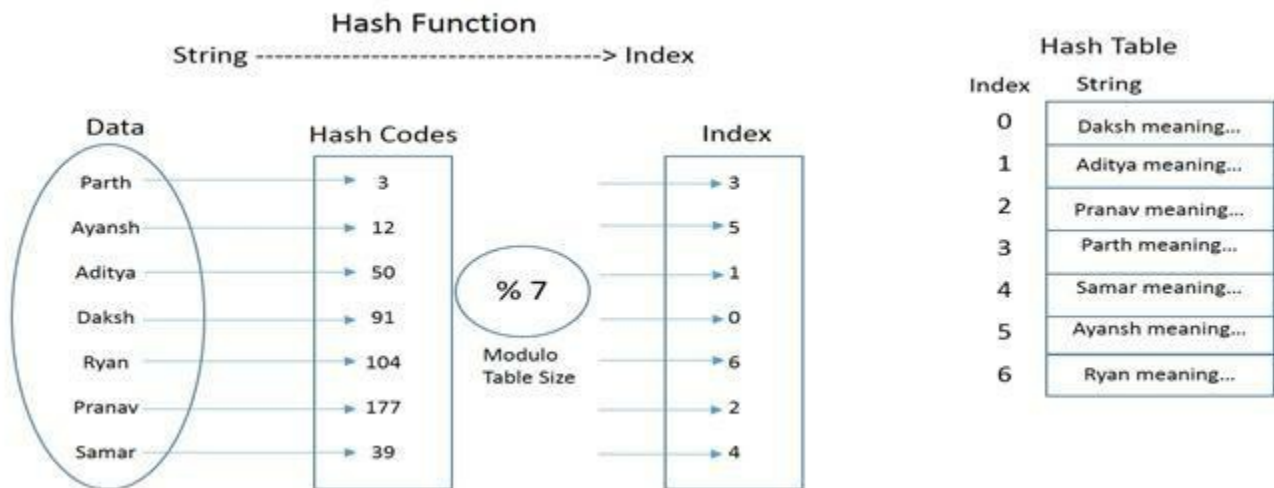
Hash-Table has an excellent lookup of O(1).

Let us suppose we want to implement autocomplete the box feature of Google search. When you type some string to search in google search, it propose some complete string even before you have done typing. BST cannot solve this problem as related strings can be in both right and left subtree.

The Hash-Table is also not suited for this job. One cannot perform a partial match or range query on a Hash-Table. Hash function transforms string to a number. Moreover, a good hash function will give a distributed hash code even for partial string and there is no way to relate two strings in a Hash-Table.

Trie and Ternary Search tree are a special kind of tree that solves partial match and range query problem efficiently.

Trie

Trie is a tree, in which we store only one character at each node. The final key value pair is stored in the leaves. Each node has R children, one for each possible character. For simplicity purpose, let us consider that the character set is 26, corresponds to different characters of English alphabets.

Trie is an efficient data structure. Using Trie, we can search the key in O(M) time. Where M is the maximum string length. Trie is also suitable for solving partial match and range query problems.

TRIE DICTIONARY
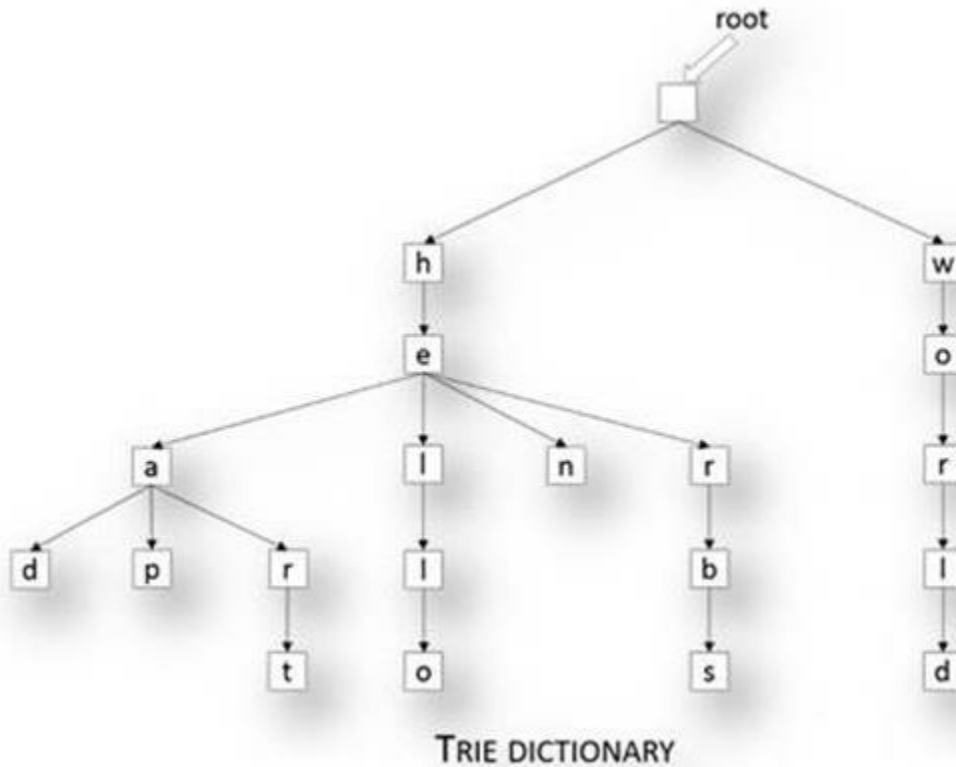
Example 13.7:

```ruby
class Trie
   class Node
      attr_accessor :child, :isLastChar
      def initialize(isLastChar = false)
         @child = Array.new(26, Node)
         i = 0
         while i < 26
            @child[i] = nil
            i += 1
         end
         @isLastChar = isLastChar
      end
   end
```

```ruby
   #first node with dummy value.
   def Insert(str)
      if str == nil
         return @root
      end
      @root = self.InsertUtil(@root, str.downcase(), 0)
   end
```

```ruby
   def InsertUtil(curr, str, index)
      if curr == nil
         curr = Node.new()
      end
      if str.size == index
         curr.isLastChar = true
```

```ruby
        else
            chIndex = str[index].ord() - 'a'.ord()
            curr.child[chIndex] = self.InsertUtil(curr.child[chIndex], str, index + 1)
        end
        return curr
    end

    def Remove(str)
        if str == nil
            return
        end
        str = str.downcase()
        self.RemoveUtil(@root, str, 0)
    end

    def RemoveUtil(curr, str, index)
        if curr == nil
            return
        end
        if str.size == index
            if curr.isLastChar
                curr.isLastChar = false
            end
            return
        end
        self.RemoveUtil(curr.child[str[index].ord() - 'a'.ord()], str, index + 1)
    end

    def Find(str)
        if str == nil
            return false
        end
        str = str.downcase()
        return self.FindUtil(@root, str, 0)
    end

    def FindUtil(curr, str, index)
        if curr == nil
            return false
        end
        if str.size == index
            return curr.isLastChar
        end
        return self.FindUtil(curr.child[str[index].ord() - 'a'.ord()], str, index + 1)
    end
end

# Testing code
```

```
t = Trie.new()
a = "apple"
b = "app"
c = "appletree"
d = "tree"
t.Insert(a)
t.Insert(d)
puts t.Find(a)
puts t.Find(b)
puts t.Find(c)
puts t.Find(d)
```

Output:
```
true
false
false
true
```

Ternary Search Trie/ Ternary Search Tree

Tries have a very good search performance of O(M) where M is the maximum size of the search string. However, tries have very high space requirement. In every node Trie contains references to multiple nodes, each reference corresponds to possible characters of the key.   To avoid this high space requirement Ternary Search Trie (TST) is used.

 A TST avoids heavy space requirement of traditional Trie, still keeping many of its advantages. In a TST, each node contains a character, an end of key indicator and three references. The three references are corresponding to current char hold by the node(equal), characters less than and character greater than.

The Time Complexity of ternary search tree operation is proportional to the height of the ternary search tree. In the worst case, we need to traverse up to 3 times the length of largest string. However, this case is rare. Therefore, TST is a very good solution for implementing Symbol Table, Partial match and range query.
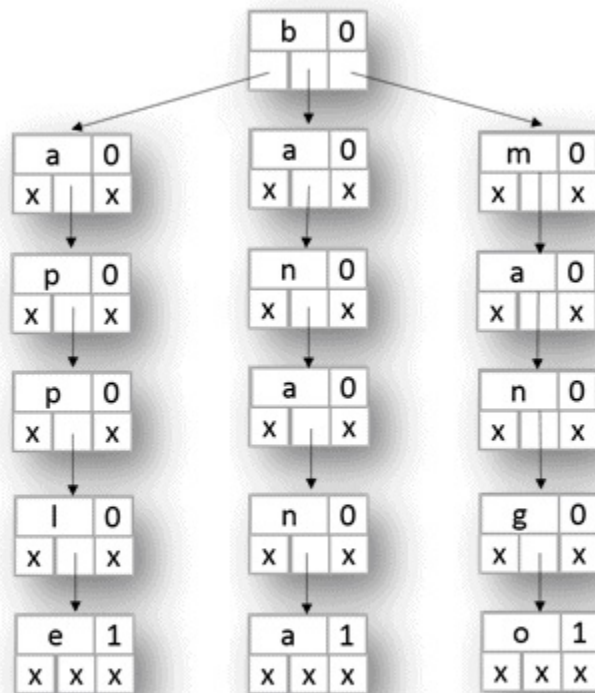
Ternary Search Tree

Example 13.8:

```
class TST
    def initialize()
    end

    class Node
        attr_accessor :data, :isLastChar, :left, :equal, :right
        def initialize(d)
            @data = d
            @isLastChar = false
            @left = @equal = @right = nil
        end
    end

    def insert(word)
        @root = self.insertUtil(@root, word, 0)
    end

    def insertUtil(curr, word, wordIndex)
        if curr == nil
            curr = Node.new(word[wordIndex])
        end
        if word[wordIndex] < curr.data
            curr.left = self.insertUtil(curr.left, word, wordIndex)
        elsif word[wordIndex] > curr.data
            curr.right = self.insertUtil(curr.right, word, wordIndex)
        else
```

```ruby
            if wordIndex < word.size - 1
                curr.equal = self.insertUtil(curr.equal, word, wordIndex + 1)
            else
                curr.isLastChar = true
            end
        end
        return curr
    end

    def find(curr, word, wordIndex)
        if curr == nil
            return false
        end
        if word[wordIndex] < curr.data
            return self.find(curr.left, word, wordIndex)
        elsif word[wordIndex] > curr.data
            return self.find(curr.right, word, wordIndex)
        else
            if wordIndex == word.size - 1
                return curr.isLastChar
            end
            return self.find(curr.equal, word, wordIndex + 1)
        end
    end

    def findWrapper(word)
        ret = self.find(@root, word, 0)
        print (word + " :: ")
        if ret
            print " Found \n"
        else
            print "Not Found \n"
        end
        return ret
    end
end

# Testing code
tt = TST.new()
tt.insert("banana")
tt.insert("apple")
tt.insert("mango")
tt.findWrapper("apple")
tt.findWrapper("banana")
tt.findWrapper("mango")
tt.findWrapper("grapes")
```

Output:
```
apple :: Found
banana    :: Found
mango :: Found
grapes    :: Not Found
```

Problems in String

Regular Expression Matching

Implement regular expression matching with the support of '?' and '*' special character.
'?' Matches any single character.
'*' Matches zero or more of the preceding element.

Example 13.9:
```
def matchExp(exp, str)
    return matchExpUtil(exp, str, 0, 0)
end

def matchExpUtil(exp, str, i, j)
    if i == exp.size and j == str.size
        return true
    end
    if (i == exp.size and j != str.size) or (i != exp.size and j == str.size)
        return false
    end
    if exp[i] == '?' or exp[i] == str[j]
        return matchExpUtil(exp, str, i + 1, j + 1)
    end
    if exp[i] == '*'
        return (matchExpUtil(exp, str, i + 1, j) or
        matchExpUtil(exp, str, i, j + 1) or
        matchExpUtil(exp, str, i + 1, j + 1))
    end
    return false
end
```

Order Matching

In given long text string and a pattern string, find if the characters of pattern string are in the same order in text string.
Eg. Text String: ABCDEFGHIJKLMNOPQRSTUVWXYZ, Pattern string: JOST

Example 13.10:
```
def match(source, pattern)
    iSource = 0
    iPattern = 0
    sourceLen = source.size
```

```
        patternLen = pattern.size
        iSource = 0
        while iSource < sourceLen
            if source[iSource] == pattern[iPattern]
                iPattern += 1
            end
            if iPattern == patternLen
                return 1
            end
            iSource += 1
        end
        return 0
end
```

Unique Characters

Write a function that will take a string as input and return 1 if it contains all unique characters, else return 0.

Example 13.11:
```
def isUniqueChar(str)
    bitarr = Array.new(26)
    i = 0
    while i < 26
        bitarr[i] = 0
        i += 1
    end
    size = str.size
    i = 0
    while i < size
        c = str[i].ord
        if 'A'.ord <= c and 'Z'.ord    >= c
            c = (c - 'A'.ord )
        elsif 'a'.ord   <= c and 'z'.ord    >= c
            c = (c - 'a'.ord )
        else
            print "Unknown Char!\n"
            return false
        end
        if bitarr[c] != 0
            print "Duplicate detected!\n"
            return false
        end
        bitarr[c] += 1
        i += 1
    end
    print "No duplicate detected!\n"
    return true
end
```

Permutation Check

Example 13.12: Function to check if two strings are permutation of each other.

```ruby
def isPermutation(s1, s2)
    count = Array.new( 256)
    length = s1.size
    if s2.size != length
        return false
    end
    i = 0
    while i < 256
        count[i] = 0
        i += 1
    end
    i = 0
    while i < length
        ch = s1[i].ord
        count[ch] += 1
        ch = s2[i].ord
        count[ch] -= 1
        i += 1
    end
    i = 0
    while i < 256
        if count[i] != 0
            return false
        end
        i += 1
    end
    return true
end
```

Palindrome Check

Example 13.13: Find if the string is a palindrome or not

```ruby
def isPalindrome(str)
    i = 0
    j = str.size - 1
    while i < j and str[i] == str[j]
        i += 1
        j -= 1
    end
    if i < j
        print "String is not a Palindrome"
        return false
    else
        print "String is a Palindrome"
```

```
        return true
    end
end
```

Time Complexity is **O(n)** and Space Complexity is **O(1)**

Power function

Example 13.14: Function which will calculate x^n, Taking x and n as argument.

```
def pow(x, n)
    if n == 0
        return (1)
    elsif n % 2 == 0
        value = pow(x, n / 2)
        return (value * value)
    else
        value = pow(x, n / 2)
        return (x * value * value)
    end
end
```

String Compare function

Write a function strcmp() to compare two strings. The function return values should be:
   a) The return value is 0 indicates, that both first and second strings are equal.
   b) The return value is negative it indicates, the first string is less than the second string.
   c)   The return value is positive it indicates, that the first string is greater than the second string.

Example 13.15:

```
def strcmp(a, b)
    index = 0
    len1 = a.size
    len2 = b.size
    minlen = len1
    if len1 > len2
        minlen = len2
    end
    while index < minlen and a[index] == b[index]
        index += 1
    end
    if index == len1 and index == len2
        return 0
    elsif len1 == index
        return -1
    elsif len2 == index
        return 1
    else
```

```
        return a[index].ord  - b[index].ord
    end
end
```

Reverse String

Example 13.16: Reverse all the characters of a string.
```
def reverseString1(a)
    lower = 0
    upper = a.size - 1
    while lower < upper
        tempChar = a[lower]
        a[lower] = a[upper]
        a[upper] = tempChar
        lower += 1
        upper -= 1
    end
end

def reverseString(a, lower, upper)
    while lower < upper
        tempChar = a[lower]
        a[lower] = a[upper]
        a[upper] = tempChar
        lower += 1
        upper -= 1
    end
end
```

Reverse Words

Example 13.17: Reverse order of words in a string sentence.
```
def reverseWords(a)
    length = a.size
    upper = -1
    lower = 0
    i = 0
    while i <= length
        if a[i] == ' ' or i == length
            reverseString(a, lower, upper)
            lower = i + 1
            upper = i
        else
            upper += 1
        end
        i += 1
    end
    reverseString(a, 0, length - 1)
```

```
    return a
    # -1 because we do not want to reverse \0
end
```

Print Anagram

Example 13.18: Given a string as character list, print all the anagram of the string.

```
def printAnagram(a)
    n = a.size
    printAnagramUtil(a, n, n)
end

def printAnagramUtil(a, max, n)
    if max == 1
        print a,"\n"
    end
    i = -1
    while i < max - 1
        if i != -1
            temp = a[i]
            a[i] = a[max - 1]
            a[max - 1] = temp
        end
        printAnagramUtil(a, max - 1, n)
        if i != -1
            temp = a[i]
            a[i] = a[max - 1]
            a[max - 1] = temp
        end
        i += 1
    end
end
```

Exercise

1. In given string, find the longest substring without reputed characters.

2. The function memset() copies ch into the first 'n' characters of the string

3. Serialize a collection of string into a single string and de serializes the string into that collection of strings.

4. Write a smart input function, which takes 20 characters as input from the user. Without cutting some word.
   User input: "Harry Potter must not go"
   First 20 chars: "Harry Potter must no"
   Smart input: "Harry Potter must"

5. Write a code that returns if a string is palindrome and it should return true for below inputs too.

> Stella won no wallets.
> No, it is open on one position.
> Rise to vote, Sir.
> Won't lovers revolt now?

6. Write an ASCII to integer function, which ignore the non-integral character and give the integer. For example, if the input is "12AS5" it should return 125.

7. Write code that would parse a Bash brace expansion.
Example: the expression "(a, b, c) d, e" and would give output all the possible strings: ad, bd, cd, e

8. In given string write a function to return the length of the longest substring with only unique characters

9. [Replace all occurrences of "a" with "the"](#)

10. Replace all occurrences of %20 with ' '.
E.g. Input: www.Hello%20World.com
Output: [www.Hello](#) World.com

11. Write an expansion function that will take an input string like "1..5,8,11..14,18,20,26..30" and will print "1,2,3,4,5,8,11,12,13,14,18,20,26,27,28,29,30"

12. Suppose you have a string like "Thisisasentence". Write a function that would separate these words. Moreover, will print whole sentence with spaces.

13. In given three string str1, str2 and str3. Write a complement function to find the smallest sub-sequence in str1 which contains all the characters in str2 and but not those in str3.

14. In given two strings A and B, find whether any anagram of string A is a sub string of string B.
For eg: If A = xyz and B = afdgzyxksldfm then the program should return true.

15. In given string, find whether it contains any permutation of another string. For example, given "abcdefgh" and "ba", the function should return true, because "abcdefgh" has substring "ab", which is a permutation of the given string "ba".

16. In give algorithm which removes the occurrence of "a" by "bc" from a string? The algorithm must be in-place.

17. In given string "1010101010" in base2 convert it into string with base4. Do not use an extra space.

18. In Binary Search tree to store strings, delete() function is not implemented, implement the

same.

19. If you implement delete() function, then you need to make changes in find() function. Do the needful.

CHAPTER 14: ALGORITHM DESIGN TECHNIQUES

Introduction

In real life, when we are asked to do some work, we try to correlate it with our experience and then try to solve it. Similarly, when we get a new problem to solve. We first try to find the similarity of the current problem with some problems for which we already know the solution. Then solve the current problem and get our desired result.

This method provides following benefits:
1) It provides a template for solving a wide range of problems.
2) It provides us an idea of the suitable data structure for the problem.
3) It helps us in analysing space and Time Complexity of algorithms.

In the previous chapters, we have used various algorithms to solve different kind of problems. In this chapter, we will read about various techniques of solving algorithmic problems.

Various Algorithm design techniques are:
1) Brute Force
2) Greedy Algorithms
3) Divide-and-Conquer, Decrease-and-Conquer
4) Dynamic Programming
5) Reduction / Transform-and-Conquer
6) Backtracking and Branch-and-Bound

Brute Force Algorithm

Brute Force is a straightforward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem. It is useful for solving small size dataset problem.

Some examples of brute force algorithms are:
· Bubble-Sort
· Selection-Sort
· Sequential search in a list
· Computing pow(a, n) by multiplying a, n times.
· Convex hull problem
· String matching
· Exhaustive search: Traveling salesman, Knapsack, and Assignment problems

Greedy Algorithm

Greedy algorithms are generally used to solve optimization problems. In greedy algorithm, solution is constructed through a sequence of steps. At each step, choice is made which is locally optimal.

Note:- Greedy algorithms does not always give optimum solution.

Some examples of greedy algorithms are:
· Minimal spanning tree: Prim's algorithm, Kruskal's algorithm
· Dijkstra's algorithm for single-source shortest path problem
· Greedy algorithm for the Knapsack problem
· The coin exchange problem
· Huffman trees for optimal encoding

Divide-and-Conquer, Decrease-and-Conquer

Divide-and-Conquer algorithms involve basic three steps. First, split the problem into several smaller sub-problems. Second, solve each sub-problem. Finally, combine the sub-problems results to produce the desired result.

In divide-and-conquer the size of the problem is reduced by a factor (half, one-third, etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.

Examples of divide-and-conquer algorithms:
· Merge-Sort algorithm (using recursion)
· Quicksort algorithm (using recursion)
· Computing the length of the longest path in a binary tree (using recursion)
· Computing Fibonacci numbers (using recursion)
· Quick-hull

Examples of decrease-and-conquer algorithms:
· Computing pow(a, n) by calculating pow(a, n/2) using recursion.
· Binary search in a sorted list (using recursion)
· Searching in BST
· Insertion-Sort
· Graph traversal algorithms (DFS and BFS)
· Topological sort
· Warshall's algorithm (using recursion)
· Permutations (Minimal change approach, Johnson-Trotter algorithm)
· Computing a median, Topological sorting, Fake-coin problem (Ternary search)

Consider the problem of exponentiation Compute x^n

Brute Force:	n-1 multiplications
Divide and conquer:	$T(n) = 2*T(n/2) + 1 = n-1$
Decrease by one:	$T(n) = T(n-1) + 1 = n-1$
Decrease by constant factor:	$T(n) = T(n/a) + a-1$ $= (a-1) n$ $= n$ when a = 2

Dynamic Programming

While solving problems using Divide-and-Conquer method, there may be a case when recursively sub-problems can result in the same computation being performed multiple times. This problem arises when there are identical sub-problems arise repeatedly in a recursion.

Dynamic programming is used to avoid the requirement of repeated calculation of same sub-problem. In this method, we usually store the result of sub - problems in a table and refer that table to find if we have already calculated the solution of sub - problems before calculating it again.

Dynamic programming is a bottom up technique in which the smaller sub-problems are solved first and the result of these are sued to find the solution of the larger sub-problems.

Examples:
- Fibonacci numbers computed by iteration.
- Warshall's algorithm for transitive closure implemented by iterations
- Floyd's algorithms for all-pairs shortest paths

```
def fibonacci(n)
    if n <= 1 then
        return n
    end
    return fibonacci(n - 1) + fibonacci(n - 2)
end
```
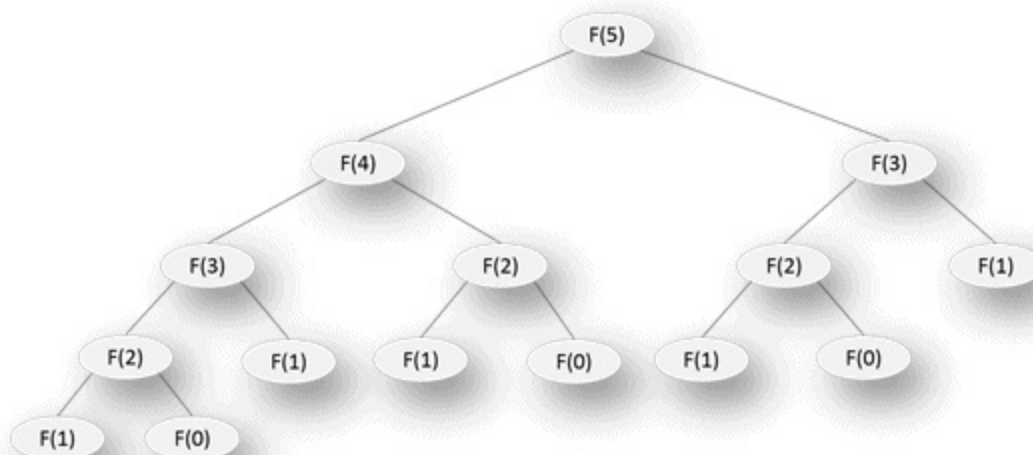

Using divide and conquer the same sub problem is solved again and again, which reduce the performance of the algorithm. This algorithm has an exponential Time Complexity and linear Space Complexity.

```
def fibonacci(n)
    first = 0
    second = 1
    temp = 0
    if n == 0 then
        return first
```

```
    elsif n == 1 then
        return second
    end
    i = 2
    while i <= n
        temp = first + second
        first = second
        second = temp
        i += 1
    end
    return temp
end
```

Using this algorithm, we will get Fibonacci in linear Time Complexity and constant Space Complexity.

Reduction / Transform-and-Conquer

These methods work as a two-stage procedure. First, the problem is transformed into a known problem for which we know optimal solution. In the second stage, the problem is solved.

The most common type of transformation is sorting of a list. For example, in a given list of numbers finds the two closest number.

Brute-force solution, we will find distance between each element in the list and will keep the minimum distance pair. In this approach total Time Complexity will be **O(n^2)**

Transform and conquer solution, we will first sort the list in **O(nlogn)** time and then find the closest number by scanning the list in another single pass with time complexity **O(n)**. Thus the total Time Complexity will be **O(nlogn)**.

Examples:
·   Gaussian elimination
·   Heaps and Heapsort

Backtracking

In real life, let us suppose someone has given you a lock with a number (three digit lock, number range from 1 to 9). Moreover, you do not have the exact password key for the lock. You need to test every combination until you got the right one. Obviously, you need to test starting from something like "111", then "112" and so on. You will get your key before you reach "999". Therefore, what you are doing is backtracking.

Suppose the lock produces some sound "click" if correct digit is selected for any level. If we can listen to this sound such intelligence/ heuristics will help you to reach your goal much faster. These functions are called Pruning function or bounding functions.

Backtracking is a method by which solution is found by exhaustively searching through large but finite number of states, with some pruning or bounding function we can narrow down our search.

For all the problems (like NP hard problems) for which there does not exist any other efficient algorithm we use backtracking algorithm.

Backtracking problems have the following components:
1. Initial state
2. Target / Goal state
3. Intermediate states
4. Path from the initial state to the target / goal state
5. Operators to get from one state to another
6. Pruning function (optional)

The solving process of backtracking algorithm starts with the construction of state's tree, whose nodes represents the states. The root node is the initial state and one or more leaf node will be our target state. Each edge of the tree represents some operation. The solution is obtained by searching the tree until a Target state is found.

Backtracking uses depth-first search:
1) Store the initial state in a stack
2) While the stack is not empty, repeat:
3) Read a node from the stack.
4) While there are available operators, do:
   a. Apply an operator to generate a child
   b. If the child is a goal state – return solution
   c. If it is a new state, and pruning function does not discard it, than push the child into the stack.

There are three monks and three demons at one side of a river. We want to move all of them to the other side using a small boat. The boat can carry only two persons at a time. Given if on any shore the number of demons will be more than monks then they will eat the monks. How can we move all of these people to the other side of the river safely?

Same as the above problem there is a farmer who has a goat, a cabbage and a wolf. If the farmer leaves, goat with cabbage, goat will eat the cabbage. If the farmer leaves wolf alone with goat, wolf will kill the goat. How can the farmer move all his belongings to the other side of the river?

You are given two jugs, a 4-gallon one and a 3-gallon one. There are no measuring markers on jugs. A tap can be used to fill the jugs with water. How can you get 2 gallons of water in the 4-gallon jug?

Branch-and-bound

Branch and bound method is used when we can evaluate cost of visiting each node by a utility functions. At each step, we choose the node with the lowest cost to proceed further. Branch-and bound algorithms are implemented using a priority queue. In branch and bound, we traverse the

nodes in breadth-first manner.

A* Algorithm

A* is a sort of an elaboration on branch-and-bound. In branch-and-bound, at each iteration we expand the shortest path that we have found so far. In A*, instead of just picking the path with the shortest length so far, we pick the path with the shortest estimated total length from start to goal, where the total length is estimated as length traversed so far plus a heuristic estimate of the remaining distance from the goal.

Branch-and-bound will always find an optimal solution, which is the shortest path. A* will always find an optimal solution if the heuristic is correct. Choosing a good heuristic is the most important part of A* algorithm.

Conclusion

Usually a given problem can be solved using a number of methods; however, it is not wise to settle for the first method that comes to our mind. Some methods result in a much more efficient solution than others do.

For example, the Fibonacci numbers calculated recursively (decrease-and-conquer approach), and computed by iterations (dynamic programming). In the first case, the complexity is **$O(2^n)$,** and in the other case, the complexity is **$O(n)$**.

Another example, consider sorting based on the Insertion-Sort and basic bubble sort. For almost sorted files, Insertion-Sort will give almost linear complexity, while bubble sort sorting algorithms have quadratic complexity.

So the most important question is how to choose the best method?
First, you should understand the problem statement.
Second by knowing various problems and their solutions.

CHAPTER 15: BRUTE FORCE ALGORITHM

Introduction

Brute Force is a straight forward approach of solving a problem based on the problem statement. It is one of the easiest approaches to solve a particular problem. It is useful for solving small size dataset problem.

Most of the cases there are other algorithm techniques can be used to get a better solution of the same problem.

Some examples of brute force algorithms are:
- Bubble-Sort
- Selection-Sort
- Sequential search in a list
- Computing pow (a, n) by multiplying a, n times.
- Convex hull problem
- String matching
- Exhaustive search
- Traveling salesman
- Knapsack
- Assignment problems

Problems in Brute Force Algorithm

Bubble-Sort

In Bubble-Sort, adjacent elements of the list are compared and are exchanged if they are out of order.

```
// Sorts a given list by Bubble Sort
// Input: A list A of orderable elements
// Output: List A[0..n - 1] sorted in ascending order

Algorithm BubbleSort(A[0..n - 1])
    sorted = false
    while !sorted do
        sorted = true
        for j = 0 to n - 2 do
            if A[j] > A[j + 1] then
                swap A[j] and A[j + 1]
                sorted = false
```

The Time Complexity of the algorithm is $\Theta(n^2)$

Selection-Sort

The entire given list of N elements is traversed to find its smallest element and exchange it with the first element. Then, the list is traversed again to find the second element and exchange it with the second element. After N-1 passes, the list will be completely sorted.

```
//Sorts a given list by selection sort
//Input: A list A[0..n-1] of orderable elements
//Output: List A[0..n-1] sorted in ascending order

Algorithm SelectionSort (A[0..n-1])
    for i = 0 to n - 2 do
        min = i
        for j = i + 1 to n - 1 do
            if A[j] < A[min]
                min = j
        swap A[i] and A[min]
```

The Time Complexity of the algorithm is $\Theta(n^2)$

Sequential Search

The algorithm compares consecutive elements of a given list with a given search keyword until either a match is found or the list is exhausted.

```
Algorithm SequentialSearch (A[0..n], K)
    i = 0
    While A [i] ≠ K do
        i = i + 1
    if   i < n
        return i
    else
        return -1
```

Worst case Time Complexity is $\Theta(n)$.

Computing pow (a, n)

Computing a^n (a > 0, and n is a nonnegative integer) based on the definition of exponentiation. N-1 multiplications are required in brute force method.

```
// Input: A real number a and an integer n = 0
// Output: a power n

Algorithm Power(a, n)
    result = 1
    for i = 1 to n do
```

```
        result = result * a
    return result
```

The algorithm requires Θ (n)

String matching

A brute force string matching algorithm takes two inputs, first text consists of n characters and a pattern consist of m character (m<=n). The algorithm starts by comparing the pattern with the beginning of the text. Each character of the pattern is compared to the corresponding character of the text. Comparison starts from left to right until all the characters are matched or a mismatch is found. The same process is repeated until a match is found. Each time the comparison starts from one position to the right.

```
//Input: A list T[0..n - 1] of n characters representing a text
// a list P[0..m - 1] of m characters representing a pattern
//Output: The position of the first character in the text that starts the first
// matching substring if the search is successful and -1 otherwise.

Algorithm BruteForceStringMatch (T[0..n - 1], P[0..m - 1])
    for i = 0 to n - m do
    j = 0
    while j < m and P[j] = T[i + j] do
    j = j + 1
    if j = m then
        return i
    return -1
```

In the worst case, the algorithm is O(mn).

Closest-Pair Brute-Force Algorithm

The closest-pair problem is to find the two closest points in a set of n points in a 2-dimensional space.
A brute force implementation of this problem computes the distance between each pair of distinct points and find the smallest distance pair.

```
// Finds two closest points by brute force
// Input: A list P of n >= 2 points
// Output: The closest pair

Algorithm BruteForceClosestPair(P)
    dmin = infinite
    for i = 1 to n - 1 do
        for j = i + 1 to n do
            d = (xi - xj)² + (yi - yj)²
            if d < dmin then
                dmin = d
```

```
            imin = i
            jmin = j
     return imin, jmin
```

In the Time Complexity of the algorithm is $\Theta(n^2)$

Convex-Hull Problem

Convex-hull of a set of points is the smallest convex polygon that contains all the points. All the points of the set will lie on the convex hull or inside the convex hull. The convex-hull of a set of points is a subset of points in the given sets.

How to find this subset?
Answer: Subset points are the boundary of the Convex Hull. We take any two consecutive points of the bounty, and the rest of the points of the set will lie on its one side.
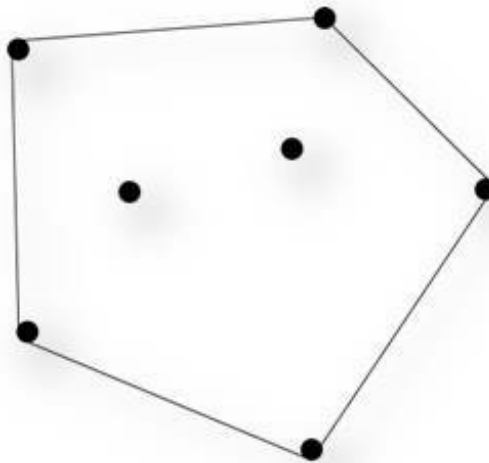
Two points (x1, y1), (x2, y2) make the line ax + by = c
Where a = y2-y1, b = x1-x2, and c = x1y2 - y1x2

And divides the plane by ax + by - c < 0 and ax + by - c > 0
So, we need to check ax + by - c for the rest of the points

If we find all the points in the set lies one side of the line with either all have ax + by - c < 0 or all the points have ax + by - c > 0 then we will add these points to the desired convex hull point set.

For each of n (n -1) /2 pairs of distinct points, one needs to find the sign of ax + by - c in each of the other n - 2 points.
What is the worst-case cost of the algorithm: O(n3)

```
Algorithm ConvexHull
    for i=0 to n-1
        for j=0 to n-1
            if (xi,yi) !=(xj,yj)
                draw a line from (xi,yi) to (xj,yj)
```

```
            for k=0 to n-1
                if(i!=k and j!=k)
                    if ( all other points lie on the same side of the
                        line (xi,yi) and (xj,yj))
                        then add   (xi,yi) to (xj,yj) to the convex hull set
```

Exhaustive Search

Exhaustive search is a brute force approach applies to combinatorial problems.
In exhaustive search, we generate all the possible combinations. At each step, we try to find if
the combinations satisfy the problem constraints. Either, we get a desired solution which satisfy
the problem constraint or there is no solution.

Examples of exhaustive search are:
· Traveling salesman problem
· Knapsack problem
· Assignment problem

Traveling Salesman Problem (TSP)

In the traveling salesman problem we need to find the shortest tour through a given set of N
cities that salesperson visits each city exactly once before returning to the city where he has
started.

Alternatively, finding the shortest Hamiltonian circuit in a weighted connected graph. A cycle
that passes through all the vertices of the graph exactly once.
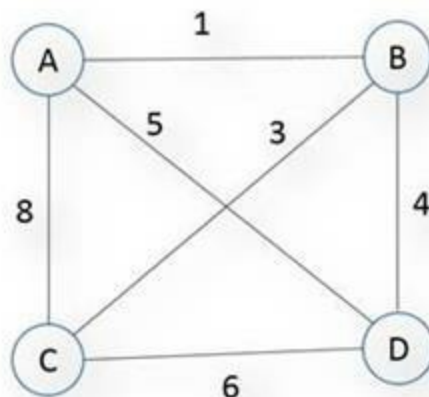
Tours where A is starting city:

Tour   Cost
A→B→C→D→A   1+3+6+5 = 15
A→B→D→C→A   1+4+6+8 = 19
A→C→B→D→A   8+3+4+5 = 20
A→C→D→B→A   8+6+4+1 = 19
A→D→B→C→A   5+4+3+8 = 20
A→D→C→B→A   5+6+3+1 = 15

```
Algorithm TSP
    Select a city
    MinTourCost = infinite
    For ( All permutations of cities ) do
        If( LengthOfPathSinglePermutation < MinTourCost )
            MinTourCost = LengthOfPath
```

Total number of possible combinations = (n-1)!
Cost for calculating the path: $\Theta(n)$
So the total cost for finding the shortest path: $\Theta(n!)$

Knapsack Problem

Given an item with cost C1, C2,..., Cn, and volume V1, V2,..., Vn   and knapsack of capacity Vmax, find the most valuable (max $\sum Cj$) that fits in the knapsack ($\sum Vj \leq Vmax$).

The solution is one of the subset of the set of object taking 1 to n objects at a time, so the Time Complexity will be $O(2^n)$

```
Algorithm KnapsackBruteForce
    MaxProfit = 0
    For ( All permutations of objects ) do
        CurrProfit = sum of objects selected
        If( MaxProfit < CurrProfit )
            MaxProfit = CurrProfit
            Store the current set of objects selected
```

Conclusion

Brute force is the first algorithm that comes into mind when we see some problem. They are the simplest algorithms that are very easy to understand. However, these algorithms rarely provide an optimum solution. In many cases we will find other effective algorithm that is more efficient than the brute force method.

CHAPTER 16: GREEDY ALGORITHM

Introduction

Greedy algorithms are generally used to solve optimization problems. To find the solution that minimizes or maximizes some value (cost/profit/count etc.).

In greedy algorithm, solution is constructed through a sequence of steps. At each step, choice is made which is locally optimal. We always take the next data to be processed depending upon the dataset which we have already processed and then choose the next optimum data to be processed.

Greedy algorithms does not always give optimum solution. For some problems, greedy algorithm gives an optimal solution. They are useful for fast approximations.

Greedy is a strategy that works well on optimization problems with the following characteristics:
1. Greedy choice: A global optimum can be arrived at by selecting a local optimum.
2. Optimal substructure: An optimal solution to the problem is made from optimal solutions of sub-problems.

Some examples of brute force algorithms are:
Optimal solutions:
- · Minimal spanning tree:
  - o Prim's algorithm,
  - o Kruskal's algorithm
- · Dijkstra's algorithm for single-source shortest path
- · Huffman trees for optimal encoding
- · Scheduling problems

Approximate solutions:
- · Greedy algorithm for the Knapsack problem
- · Coin exchange problem

Problems on Greedy Algorithm

Coin exchange problem

How can a given amount of money N be made with the least number of coins of given denominations D= {d1… dn}?

The Indian coin system {5, 10, 20, 25, 50,100}

Suppose we want to give change of a certain amount of 40 paisa.

We can make a solution by repeatedly choosing a coin ≤ to the current amount, resulting in a new amount. In the greedy algorithm, we always choose the largest coin value possible without exceeding the total amount.

For 40 paisa: {25, 10, and 5}
The optimal solution will be {20, 20}

The greedy algorithm did not gave us optimal solution, but it gave us a fair approximation.

```
Algorithm  MAKE-CHANGE (N)
   C = {5, 20, 25, 50, 100}   // constant denominations.
   S = {}            // set that will hold the solution set.
   Value = N
   WHILE Value != 0
       x = largest item in set C such that x < Value
       IF no such item THEN
           RETURN    "No Solution"
       S = S + x
       Value = Value - x
   RETURN S
```

Minimum Spanning Tree

A spanning tree of a connected graph is a tree containing all the vertices.
A minimum spanning tree of a weighted graph is a spanning tree with the smallest sum of the edge weights.
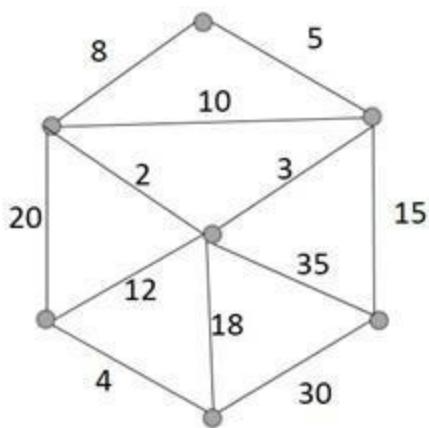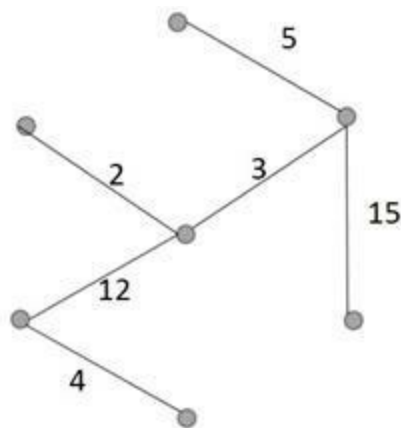
Graph   Minimum Spanning Tree

Prim's Algorithm

Prim's algorithm grows a single tree T, one edge at a time, until it becomes a spanning tree.
We initialize T with zero edges and U with single node. Where T is spanning tree edges set and U is spanning tree vertex set.

At each step, Prim's algorithm adds the smallest value edge with one endpoint in U and other not in us.
Since each edge adds one new vertex to U, after n − 1 additions, U contains all the vertices of

the spanning tree and T becomes a spanning tree.

```
// Returns the MST by Prim's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Prim(G)
    T = {}
    Let r be any vertex in G
    U = {r}
    for i = 1 to |V| - 1 do
        e = minimum-weight edge (u, v)
            With u in U and v in V-U
        U = U + {v}
        T = T + {e}
    return T
```

Prim's Algorithm using a priority queue (min heap) to get the closest fringe vertex
Time Complexity will be O(m log n) where n vertices and m edges of the MST.

Kruskal's Algorithm

Kruskal's Algorithm is used to create minimum spanning tree. Spanning tree is created by choosing smallest weight edge that does not form a cycle. And repeat this process until all the edges from the original set is exhausted.

Sort the edges in non-decreasing order of cost: $c(e1) \leq c(e2) \leq \cdots \leq c(em)$.
Set T to be the empty tree. Add edges to tree one by one if it does not create a cycle. (If the new edge form cycle then ignore that edge.)

```
// Returns the MST by Kruskal's Algorithm
// Input: A weighted connected graph G = (V, E)
// Output: Set of edges comprising a MST

Algorithm Kruskal(G)
    Sort the edges E by their weights
    T = {}
    while |T| + 1 < |V| do
        e = next edge in E
        if T + {e} does not have a cycle then
            T = T + {e}
    return T
```

Kruskal's Algorithm is O(E log V) using efficient cycle detection.

Dijkstra's algorithm for single-source shortest path problem

Dijkstra's algorithm is used for single-source shortest path problem for weighted edges with no

negative weight. It determines the length of the shortest path from the source to each of the other nodes of the graph. In a given weighted graph G, we need to find shortest paths from the source vertex s to each of the other vertices.
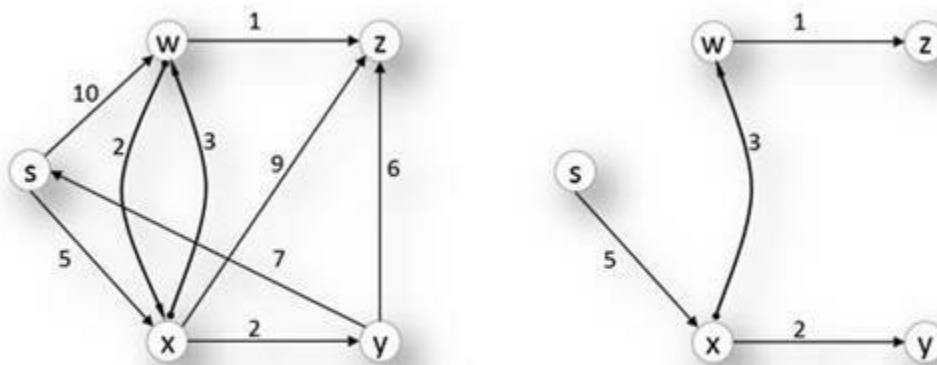
Single-Source shortest path

The algorithm starts by keeping track of the distance of each node and its parents. All the distance is set to infinite in the beginning, as we do not know the actual path to the nodes and parent of all the vertices are set to null. All the vertices are added to a priority queue (min heap implementation)
At each step algorithm takes one vertex from the priority queue (which will be the source vertex in the beginning). Then, update the distance list corresponding to all the adjacent vertices. When the queue is empty, then we will have the distance and parent list fully populated.

```
// Solves SSSP by Dijkstra's Algorithm
// Input: A weighted connected graph G = (V, E)
// with no negative weights, and source vertex v
// Output: The length and path from s to every v

Algorithm Dijkstra(G, s)
    for each v in V do
        D[v] = infinite // Unknown distance
        P[v] = null     // Unknown previous node
        add v to PQ     // Adding all nodes to priority queue
    D[source] = 0 // Distance from source to source
    while (PQ is not empty)
        u = vertex from PQ with smallest D[u]
        remove u from PQ
        for each v adjacent from u do
            alt = D[u] + length ( u , v)
            if   alt < D[v] then
                D[v] = alt
                P[v] = u
    Return D[] , P[]
```

Time Complexity will be O(|E|log|V|).
Note: Dijkstra's algorithm does not work for graphs with negative edges weight.
Note: Dijkstra's algorithm is applicable to both undirected and directed graphs.

Huffman trees for optimal encoding

Encoding is an assignment of bit strings of alphabet characters.

There are two types of encoding:
· Fixed-length encoding (eg., ASCII)
· Variable-length encoding (eg., Huffman code)

Variable length encoding can only work on prefix free encoding. Which means that no code word is a prefix of another code word.

Huffman codes are the best prefix free code. Any binary tree with edges labelled as 0 and 1 will produce a prefix free code of characters assigned to its leaf nodes.

Huffman's algorithm is used to construct a binary tree whose leaf value is assigned a code, which is optimal for the compression of the whole text need to be processed. For example, the most frequently occurring words will get the smallest code so that the final encoded text is compressed.

Initialize n one-node trees with words and the tree weights with their frequencies. Join the two binary tree with smallest weight into one and the weight of the new formed tree as the sum of weight of the two small trees. Repeat the above process N-1 times and when there is just one big tree left you are done.

Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Word	Frequency
Apple	30
Banana	25
Mango	21
Orange	14
Pineapple	10
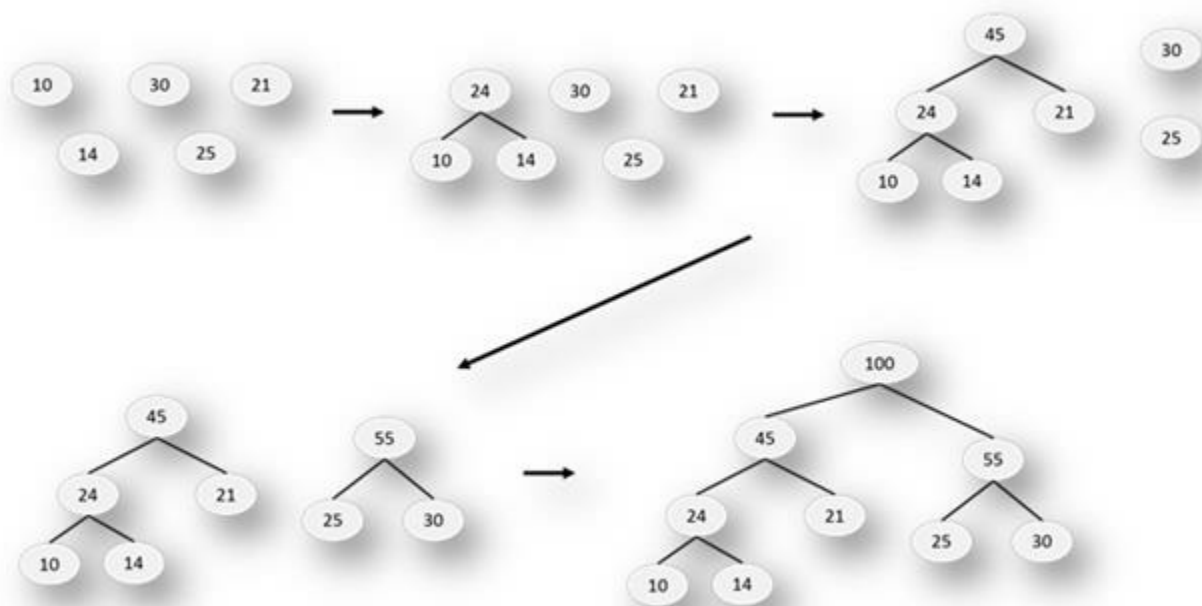
Word	Value	Code
Apple	30	11
Banana	25	10
Mango	21	01
Orange	14	001
Pineapple	10	000

It is clear that more frequency words gets smaller Huffman's code.

```
// Computes optimal prefix code.
// Input: List W of character probabilities
// Output: The Huffman tree.

Algorithm Huffman(C[0..n - 1], W[0..n - 1])
    PQ = {} // priority queue
    for i = 0 to n - 1 do
        T.char = C[i]
        T.weight = W[i]
        add T to priority queue PQ

    for i = 0 to n - 2 do
        L = remove min from PQ
        R = remove min from PQ
        T = node with children L and R
        T.weight = L.weight + R.weight
        add T to priority queue PQ
    return T
```
The Time Complexity is **O(nlogn)**.

Activity Selection Problem

Suppose that activities require exclusive use of common resources, and you want to schedule as many activities as possible.
Let $S = \{a_1,..., a_n\}$ be a set of n activities.

Each activity a_i needs the resource during a time period starting at s_i and finishing before f_i, i.e., during $[s_i, f_i)$.
The optimization problem is to select the non-overlapping largest set of activities from S.

We assume that activities $S = \{a_1,..., a_n\}$ are sorted in finish time $f_1 \le f_2 \le ... f_{n-1} \le f_n$ (this can be done in $\Theta(n \lg n)$).

Example: Consider these activities:

I	1	2	3	4	5	6	7	8	9	10	11
S[i]	1	3	0	5	3	5	6	8	8	2	11

F[i]	4	5	6	7	8	9	10	11	12	13	14

Here is a graphic representation:
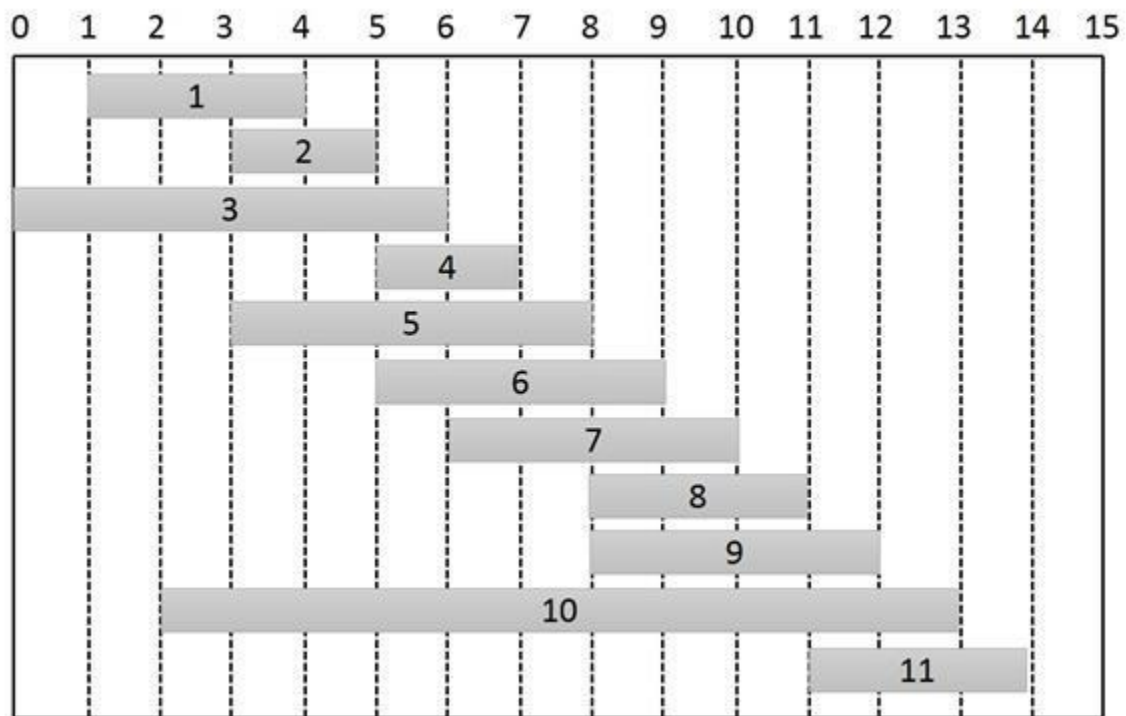
We chose an activity that starts first, and then look for the next activity that starts after it is finished. This could result in {a4, a7, a8}, but this solution is not optimal.

An optimal solution is {a1, a3, a6, a8}. (It maximizes the objective function of a number of activities scheduled.)

Another one is {a2, a5, a7, a9}. (Optimal solutions are not necessarily unique.)

How do we find (one of) these optimal solutions? Let us consider it as a dynamic programming problem.

We are trying to optimize the number of activities. Let us be greedy!

· The time left after running an activity can be used to run subsequent activities.

· If we choose the first activity to finish, the more time will be left.

· Since activities are sorted by finish time, we will always start with a_1.

· Then we can solve the single sub problem of activity scheduling in this remaining time.

```
Algorithm ActivitySelection(S[], F[], N)
    Sort S[] and F [] in increasing order of finishing time
    A = {a1}
    K = 1
    For m = 2 to N do
        If S[m] >= F[k]
            A = A + {am}
            K = m
    Return A
```

Knapsack Problem

A thief enters a store and sees a number of items with their mentioned cost and weight. His Knapsack can hold a max weight. What should he steal to maximize profit?

Fractional Knapsack problem

A thief can take a fraction of an item (they are divisible substances, like gold powder).
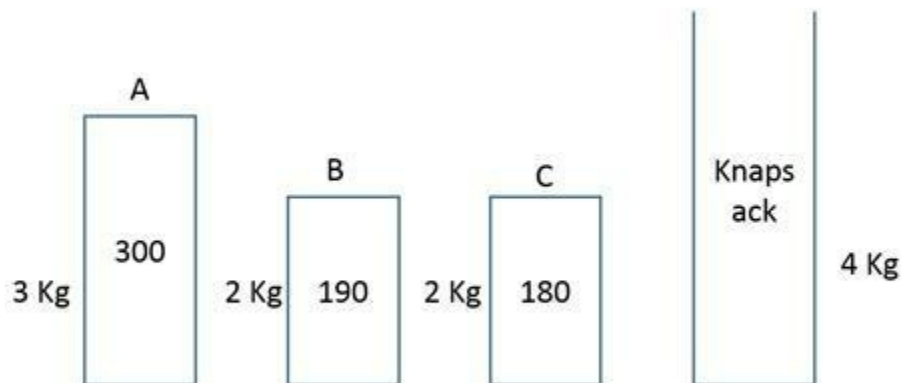
The fractional knapsack problem has a greedy solution one should first sort the items in term of cost density against weight. Then fill up as much of the most valuable substance by weight as one can hold, then as much of the next most valuable substance, etc. Until W is reached.

Item	A	B	C
Cost	300	190	180
Weight	3	2	2
Cost/weight	100	95	90

For a knapsack of capacity of 4 kg.
The optimum solution of the above will take 3kg of A and 1 kg of B.
Algorithm FractionalKnapsack(W[], C[], Wk)
   For i = 1 to n do
      X[i] = 0
   Weight = 0
   //Use Max heap
   H = BuildMaxHeap(C/W)
   While Weight < Wk do
      i = H.GetMax()
      If(Weight + W[i] <= Wk) do
         X[i] = 1
         Weight = Weight + W[i]
      Else
         X[i] = (Wk – Weight)/W[i]
         Weight = Wk
   Return X

0/1 Knapsack Problem

A thief can only take or leave the item. He cannot take a fraction.

A greedy strategy same as above could result in empty space, reducing the overall cost density of the knapsack.

In the above example, after choosing object A there is no place for B or C. So, there leaves empty space of 1kg. Moreover, the result of the greedy solution is not optimal.

The optimal solution will be when we take object B and C. This problem can be solved by dynamic programming that we will see in the coming chapter.

CHAPTER 17: DIVIDE-AND-CONQUER, DECREASE-AND-CONQUER

Introduction

Divide-and-Conquer algorithms works by recursively breaking down a problem into two or more sub-problems (divide step), until these sub-problems become simple enough so that they can be solved directly (conquer step). The solution of these sub-problems is then combined to give a solution of the original problem.

Divide-and-Conquer algorithms involve basic three steps
1. Divide the problem into smaller problems.
2. Conquer by solving these problems.
3. Combine these results together.

In divide-and-conquer the size of the problem is reduced by a factor (half, one-third etc.), While in decrease-and-conquer the size of the problem is reduced by a constant.

Divide-and-Conquer algorithms
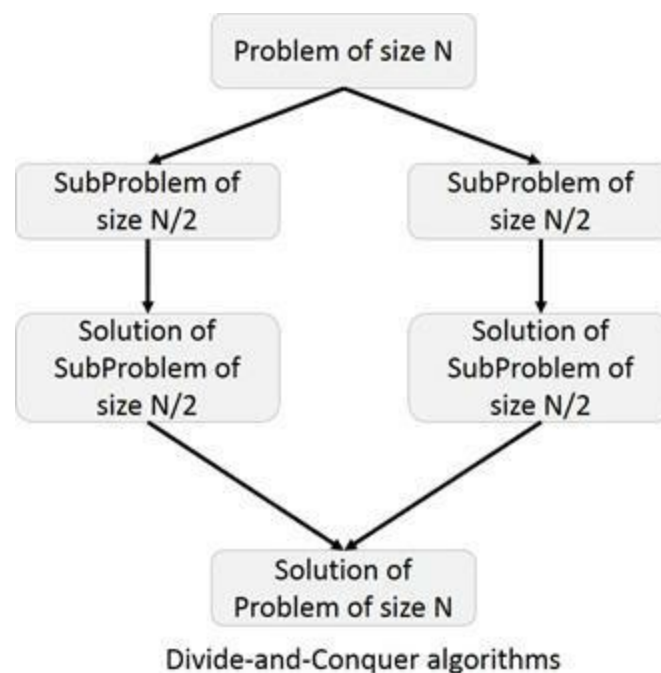
Examples of divide-and-conquer algorithms:
· Merge-Sort algorithm (recursion)
· Quicksort algorithm (recursion)
· Computing the length of the longest path in a binary tree (recursion)
· Computing Fibonacci numbers (recursion)
· Convex Hull

Examples of decrease-and-conquer algorithms:
· Computing POW (a, n) by calculating POW (a, n/2) using recursion

- Binary search in a sorted list (recursion)
- Searching in BST
- Insertion-Sort
- Graph traversal algorithms (DFS and BFS)
- Topological sort
- Warshall's algorithm (recursion)
- Permutations (Minimal change approach, Johnson-Trotter algorithm)
- Fake-coin problem (Ternary search)
- Computing a median

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$
- Where $a \geq 1$ and $b > 1$.
- "n" is the size of a problem.
- "a" is a number of sub-problem in the recursion.
- "n/b" is the size of each sub-problem.
- "f(n)" is the cost of the division of the problem into sub problem or merge of the results of sub-problem to get the final result.

Master Theorem

The master theorem solves recurrence relations of the form:
$T(n) = aT(n/b) + f(n)$
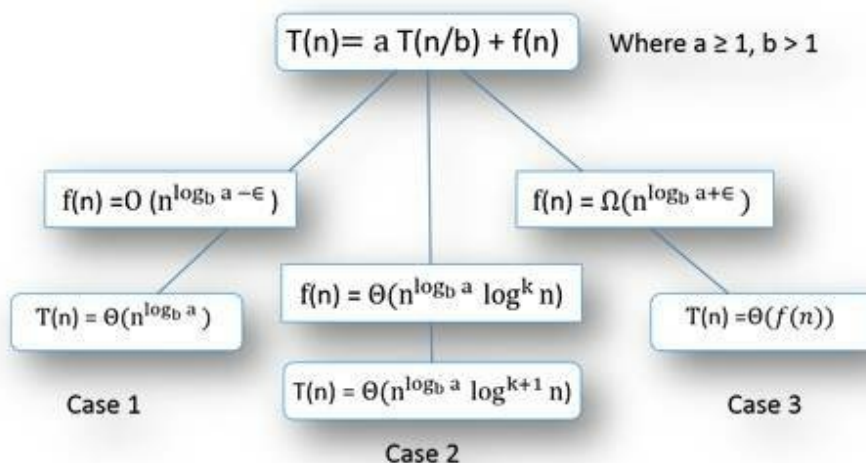
It is possible to determine an asymptotic tight bound in these three cases:

Case 1: when $f(n) = O(n^{\log_b a - \epsilon})$ and constant $\epsilon > 1$, then the final Time Complexity will be:
$T(n) = \Theta(n^{\log_b a})$

Case 2: when $f(n) = \Theta(n^{\log_b a} \log^k n)$ and constant $k \geq 0$, then the final Time Complexity will be:
$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

Case 3: when $f(n) = \Omega(n^{\log_b a + \epsilon})$ and constant $\epsilon > 1$, Then the final Time Complexity will be:
$T(n) = \Theta(f(n))$

Modified Master theorem: This is a shortcut to solve the same problem easily and quickly. If the recurrence relation is in the form of $T(n) = a\,T(n/b) + dx^s$
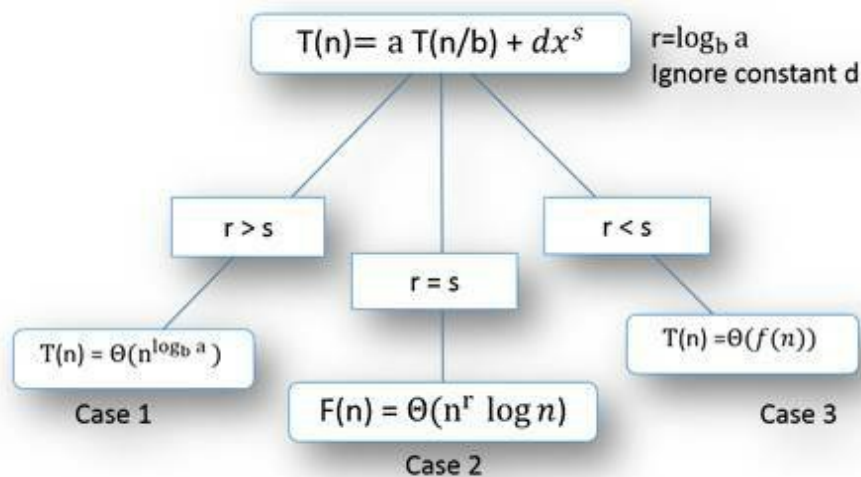
Example 1: Take an example of Merge-Sort, $T(n) = 2T(n/2) + n$
Sol:-
$Log_b a = \log_2 2 = 1$

$f(n) :: n :: \Theta(n^{\log_b a} \log^0 n)$

Case 2 applies and $T(n) :: \Theta(n^{\log_b a} \log^{0+1} n)$
$T(n) = \Theta(n \log(n))$

Example 2: Binary Search $T(n) = T(n/2) + O(1)$
Sol:-
$Log_b a = \log_2 1 = 0$

$f(n) :: 1 :: \Theta(n^{\log_b a} \log^0 n)$

Case 2 applies and $T(n) :: \Theta(n^{\log_b a} \log^{0+1} n)$
$T(n) = \Theta(\log(n))$

Example 3: Binary tree traversal $T(n) = 2T(n/2) + O(1)$
Sol:-
$Log_b a = \log_2 2 = 1$

$f(n) :: 1 :: \Theta(n^{\log_b a - 1})$

Case 1 applies and $T(n) :: \Theta(n^{\log_b a})$
$T(n) = \Theta(n)$

Problems on Divide-and-Conquer Algorithm

Merge-Sort algorithm

mergeSort(n0,n2)

| 10 | 2 | 6 | 4 | 8 | 1 | 9 | 5 | 3 | 7 |

mergeSort(n0,n1)   mergeSort(n1+1,n2)

merge(n0,n1,n2)
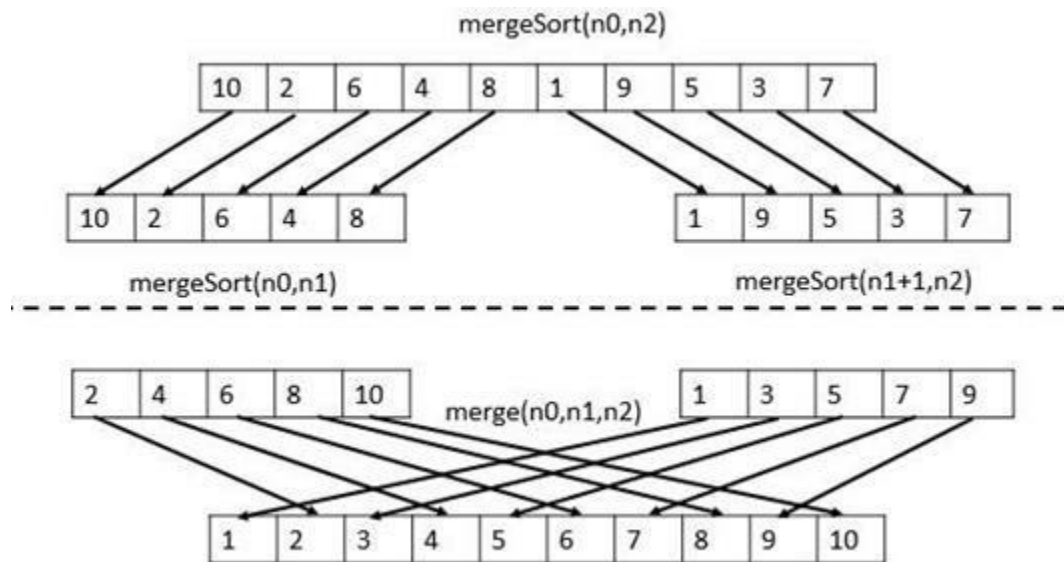
```
// Sorts a given list by mergesort
// Input: A list A of orderable elements
// Output: List A[0..n − 1] in ascending order

Algorithm Mergesort(A[0..n − 1])
    if n ≤ 1 then
        return;
    copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
    copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
    Mergesort(B)
    Mergesort(C)
    Merge(B, C, A)
```

```
// Merges two sorted arrays into one list
// Input: Sorted arrays B and C
// Output: Sorted list A

Algorithm Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])
    i = 0
    j = 0
    for k = 0 to p + q − 1 do
        if i < p and (j = q or B[i] ≤ C[j]) then
            A[k] = B[i]
            i = i + 1
        else
            A[k] = C[j]
            j = j + 1
```

Time Complexity: **O(nlogn)** & Space Complexity: **O(n)**
The Time Complexity of Merge-Sort is **O(nlogn)** in all 3 cases (worst, average and best) as Merge-Sort always divides the list into two halves and take linear time to merge two halves.

It requires equal amount of additional space as the unsorted list. Hence, it is not at all recommended for searching large unsorted lists.

Quick-Sort
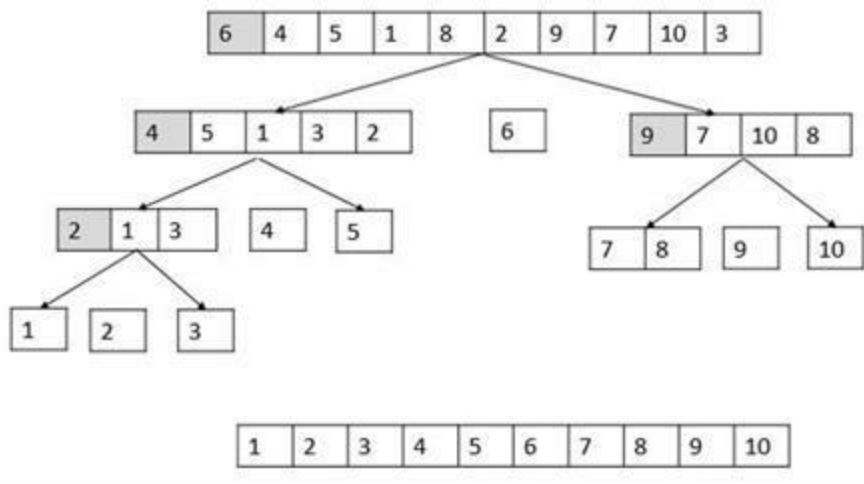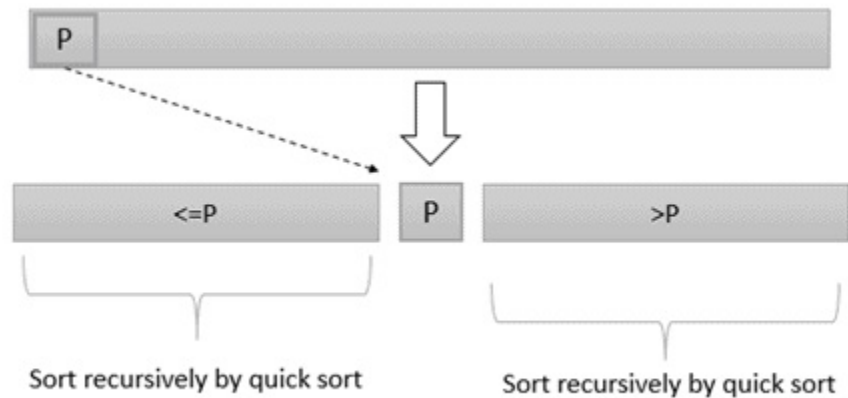

```
// Sorts a subarray by quicksort
// Input: An subarray of A
// Output: List A[l..r] in ascending order

Algorithm Quicksort(A[l..r])
    if l < r then
        p ← Partition(A[l..r]) // p is index of pivot
        Quicksort(A[l..p − 1])
        Quicksort(A[p + 1..r])
```

```
// Partitions a subarray using A[..] as pivot
// Input: Subarray of A
// Output: Final position of pivot

Algorithm Partition(A[], left, right)
    pivot = A[left]
    lower = left
    upper= right
    while lower < upper
        while A[lower] <= pivot
```

```
        lower = lower + 1
      while A[upper] > pivot
          upper = upper – 1
      if lower < upper then
          swap A[lower] and A[upper]
  swap A[lower] and A[upper] //upper is the pivot position
  return upper
```

Worst Case Time Complexity: **O(n²),**
Average Time Complexity: **O(nlogn),**
Space Complexity: **O(nlogn),** The space required by Quick-Sort is very less, only **O(nlogn)** additional space is required.
Quicksort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

External Sorting

External sorting is also done using divide and conquer algorithm.
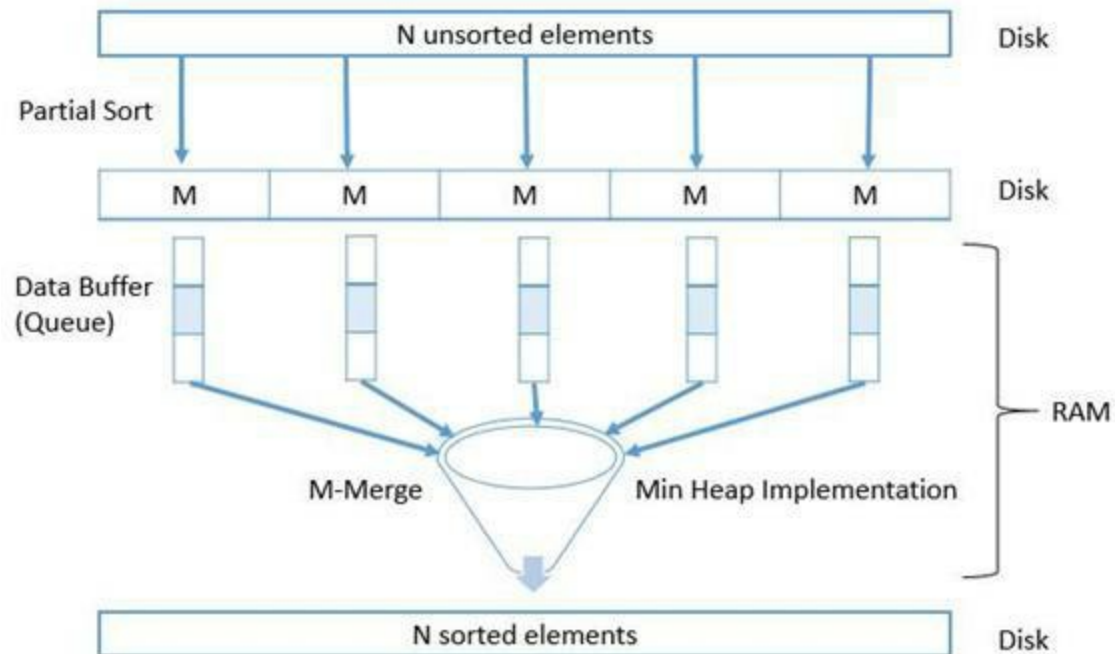
Binary Search

We get the middle point from the sorted list and start comparing with the desired value.
Note: Binary search requires the list to be sorted otherwise binary search cannot be applied.

```
// Searches a value in a   sorted list using binary search
// Input: An sorted list A and a key K
// Output: The index of K or −1

Algorithm BinarySearch(A[0..N − 1], N, K) // iterative solution
   low = 0
   high = N-1
   while low <= high do
```

```
        mid = ⌊ (low + high)/2 ⌋
        if K = A[mid] then
            return mid
        else if A[mid] < K
            low = mid + 1
        else
            high = mid - 1
    return −1
```

```
// Searches a value in a   sorted list using binary search
// Input: An sorted list A and a key K
// Output: The index of K or −1

Algorithm BinarySearch(A[], low, high, K) //Recursive solution
    If low > high
        return -1
    mid = ⌊ (low + high)/2 ⌋
    if K = A[mid] then
        return mid
    else if A[mid] < K
        return BinarySearch(A[],mid + 1, high, K)
    else
        return BinarySearch(A[],low, mid - 1, K)
```

Time Complexity: **O(logn)**. If you notice the above programs, you should keep in mind that we always take half input and throwing out the other half. So the recurrence relation for binary search is $T(n) = T(n/2) + c$. Using a divide and conquer master theorem, we get $T(n) =$ **O(logn)**.
Space Complexity: **O(1)**

Power function

```
// Compute Nth power of X using divide and conquer using recursion
// Input: Value X and power N
// Output: Power( X, N)

Algorithm Power( X, N)
    If N = 0
        Return 1
    Else if N % 2 == 0
        Value = Power(X, N/2)
        Return Value * Value
    Else
        Value = Power(X, N/2)
        Return Value * Value * X
```

Convex Hull

Sort points by X-coordinates. Divide points into equal halves A and B. Recursively compute HA and HB. Merge HA and HB to obtain CH
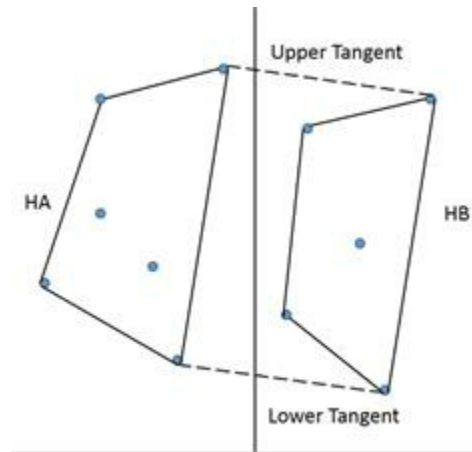

```
LowerTangent(HA, HB)
    A = rightmost point of HA
    B = leftmost point of HB
    While ab is not a lower tangent for HA and HB do
        While ab is not a lower tangent to HA do
            a = a − 1 (move a clockwise)
        While ab is not a lower tangent to HB do
            b = b + 1 (move b counterclockwise)
    Return ab
```

Similarly find upper tangent and combine the two hulls.
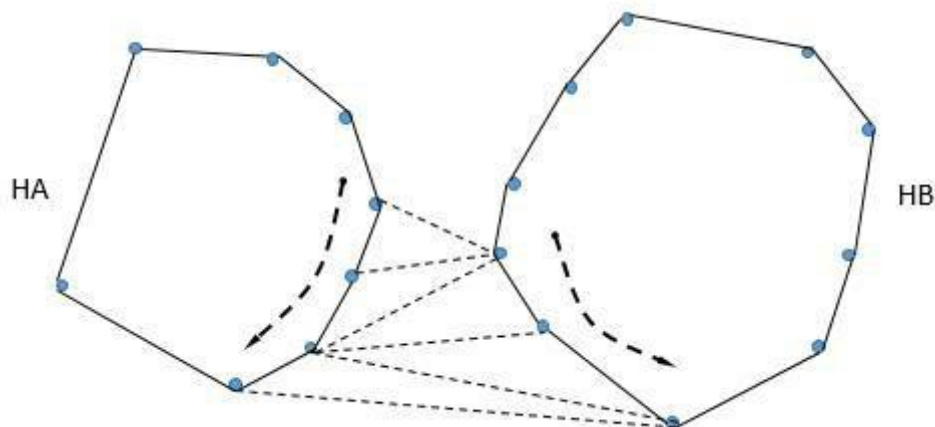
Initial sorting takes **O(nlogn)** time
Recurrence relation T (N) = 2T (N/2) + **O(N)**
Where, **O(N)** time is for tangent computation inside the merge step.
Final Time Complexity will be T (N) = **O(nlogn)**.

Closest Pair

Given N points in 2-dimensional plane, find two points whose mutual distance is smallest.
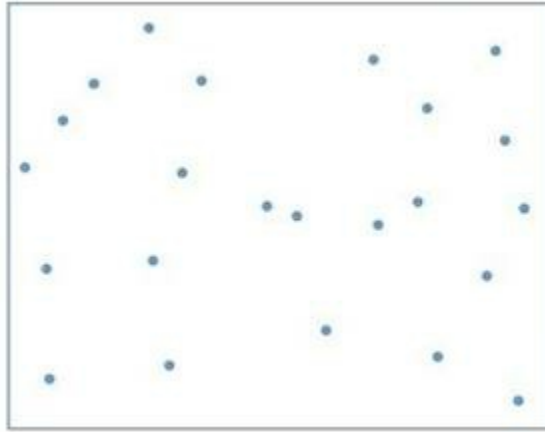
A brute force algorithm takes every point and find its distance with all the other points in the plane. In addition, keep track of the minimum distance points and minimum distance. The closest pair will be found in $O(n^2)$ time.

Let us suppose that there is a vertical line, which divides the graph into two separate parts (let us call it left and right part). In the brute force algorithm, we will notice that we are comparing all the points in the left half with the points in the right half. This is the point where we are doing some extra work.
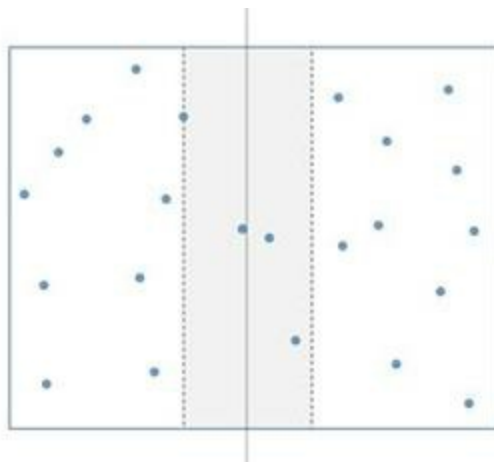
To find the minimum we need to consider only three cases:
1) Closest pair in the right half
2) Closest pair in the left half.
3) Closest pair in the boundary region of the two halves. (Gray)

Every time we will divide the space S into two parts S1 and S2 by a vertical line. Recursively we will compute the closest pair in both S1 and S2. Let us call minimum distance in space S1 as $\delta 1$ and minimum distance in space S2 as $\delta 2$.
We will find $\delta = \min (\delta 1, \delta 2)$

Now we will find the closest pair in the boundary region. By taking one point each from S1 and S2 in the boundary range of δ width on both sides.

The candidate pair of point (p, q) where p $\in$ S1 and q $\in$ S2.
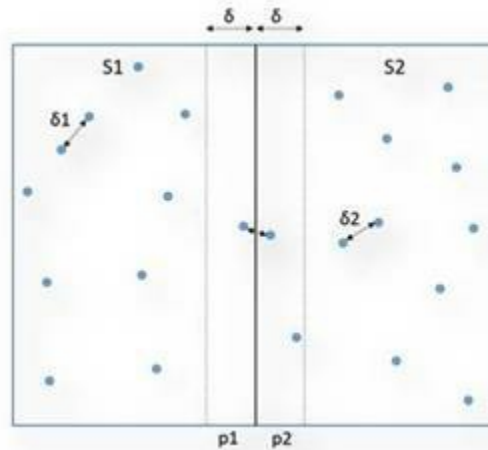
We can find the points that lie in this region in linear time O(N) by just scanning through all the points and finding that all points lie in this region.

Now we can sort them in increasing order in Y-axis in just **O(nlogn)** time. Then scan through them and get the minimum in just one linear pass. Closest pair cannot be far apart from each other.

Let us look into the next figure.
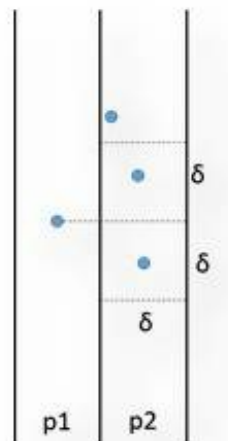
Then the question is how many points we need to compare. We need to compare the points sorted in Y-axis only in the range of δ. Therefore, the number of points will come down to only 6 points.

By doing this, we are getting equation.
$T(N) = 2T(N/2) + N + NlogN + 6N = O(n(logn)^2)$

Can we optimize this further?
Yes

Initially, when we are sorting the points in X coordinate we are sorting them in Y coordinate too.

When we divide the problem, then we traverse through the Y coordinate list too, and construct the corresponding Y coordinate list for both S1 and S2. And pass that list to them.

Since we have the Y coordinate list passed to a function the δ region points can be found sorted in the Y coordinates in just one single pass in just **O(N)** time.

T(N) = 2T(N/2) + N + N + 6N = **O(nlogn)**

```
// Finds closest pair of points
// Input: A set of n points sorted by coordinates
// Output: Distance between closest pair

Algorithm ClosestPair(P)
    if n < 2 then
        return ∞
    else if n = 2 then
        return distance between pair
    else
        m = median value for x coordinate
        δ1 = ClosestPair(points with x < m)
        δ2 = ClosestPair(points with x > m)
        δ = min(δ1, δ2)
        δ3 = process points with m −δ < x < m + δ
        return min(δ, δ3)
```

First pre-process the points by sorting them in X and Y coordinates. Use two separate lists to keep these sorted points.

Before recursively solving sub-problem pass the sorted list for that sub-problem.

CHAPTER 18: DYNAMIC PROGRAMMING

Introduction

While solving problems using Divide-and-Conquer method, there may be a case when recursively sub-problems can result in the same computation being performed multiple times. This problem arises when there are identical sub-problems arise repeatedly in a recursion.

Dynamic programming is used to avoid the requirement of repeated calculation of same sub-problem. In this method, we usually store the result of sub - problems in some data structure (like a table) and refer it to find if we have already calculated the solution of sub - problems before calculating it again.

Dynamic programming is applied to solve problems with the following properties:
1. Optimal Substructure: An optimal solution constructed from the optimal solutions of its sub-problems.
2. Overlapping Sub-problems: While calculating the optimal solution of sub-problems same computation is repeated repeatedly.

Examples:
1. Fibonacci numbers computed by iteration.
2. Assembly-line Scheduling
3. Matrix-chain Multiplication
4. 0/1 Knapsack Problem
5. Longest Common Subsequence
6. Optimal Binary Tree
7. Warshall's algorithm for transitive closure implemented by iterations
8. Floyd's algorithms for all-pairs shortest paths
9. Optimal Polygon Triangulation
10. Floyd-Warshall's Algorithm

Steps for solving / recognizing if DP applies.
1. Optimal Substructure: Try to find if there is a recursive relation between problem and sub-problem.
2. Write recursive relation of the problem. (Observe Overlapping Sub-problems at this step.)
3. Compute the value of sub-problems in a bottom up fashion and store this value in some table.
4. Construct the optimal solution from the value stored in step 3.
5. Repeat step 3 and 4 until you get your solution.

Problems on Dynamic programming Algorithm

Fibonacci numbers

```
def fibonacci(n)
    if n <= 1 then
        return n
    end
    return fibonacci(n - 1) + fibonacci(n - 2)
end)
```
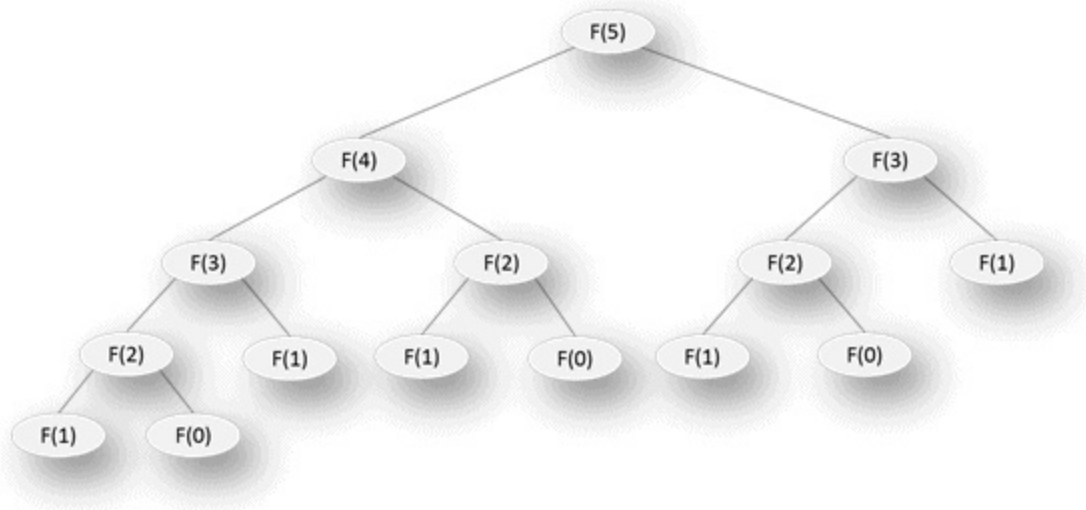

Using divide and conquer same sub-problem is solved again and again, which reduce the performance of the algorithm. This algorithm has an exponential Time Complexity.

Same problem of Fibonacci can be solved in linear time if we sort the results of sub-problems.

```
def fibonacci(n)
    first = 0
    second = 1
    temp = 0
    if n == 0 then
        return first
    elsif n == 1 then
        return second
    end
    i = 2
    while i <= n
        temp = first + second
        first = second
        second = temp
        i += 1
    end
    return temp
end
```

Using this algorithm, we will get Fibonacci in linear Time Complexity and constant Space Complexity.

Assembly-line Scheduling
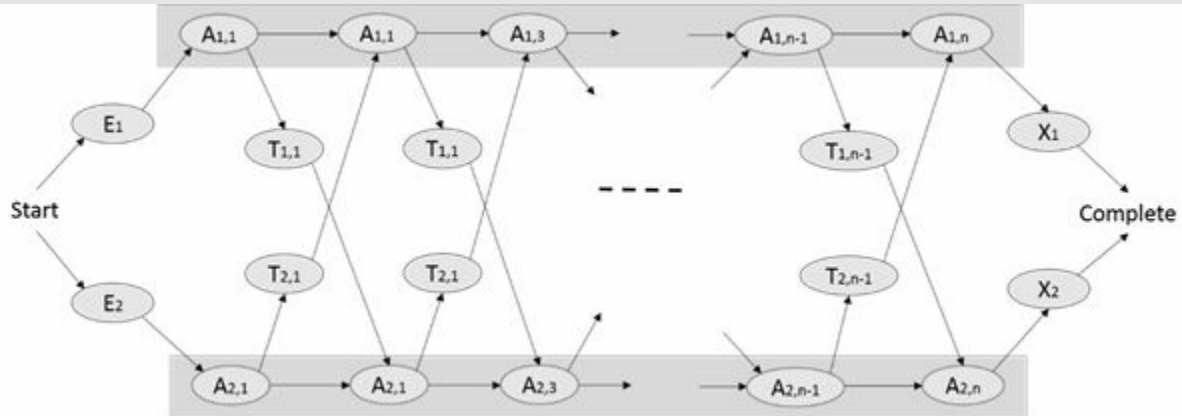
We consider the problem of calculating the least amount of time necessary to build a car when using a manufacturing chain with two assembly lines, as shown in the figure
The problem variables:
· e[i]: entry time in assembly line i
· x[i]: exit time from assembly line i
· a[i,j]: Time required at station S[i,j] (assembly line i, stage j)
· t[i,j]: Time required to transit from station S[i,j] to the other assembly line

Your program must calculate:
· The least amount of time needed to build a car
· The list of stations to traverse in order to assemble a car as fast as possible.
The manufacturing chain will have no more than 50 stations.

If we want to solve this problem in the brute force approach, there will be in total 2^n Different combinations so the Time Complexity will be O(2^n)

Step 1: Characterizing the structure of the optimal solution
To calculate the fastest assembly time, we only need to know the fastest time to $S1_n$ and the fastest time to $S2_n$, including the assembly time for the nth part. Then we choose between the two exit points by taking into consideration the extra time required, x1 and x2.   To compute the fastest time to $S1_n$ we only need to know the fastest time to $S1_{n-1}$ and to $S2_{n-1}$. Then there are only two choices.

Step 2: A recursive definition of the values to be computed

$$f1[j] = \begin{cases} e1 + a1,1 \text{ if } j = 1 \\ \min(f1[j-1] + a1,j, f2[j-1] + t2,j-1 + a1,j) \text{ if } j \geq 2 \end{cases}$$

$$f2[j] = \begin{cases} e2 + a2,1 \text{ if } j = 1 \\ \min(f2[j-1] + a2,j, f1[j-1] + t1,j-1 + a2,j) \text{ if } j \geq 2 \end{cases}$$

Step 3: Computing the fastest time finally, compute f* as

Step 4: Computing the fastest path compute as li[j] as the choice made for fi[j] (whether the first or the second term gives the minimum). Also, compute the choice for f* as l*.

```
FASTEST-WAY(a, t, e, x, n)
    f1[1] ← e1 + a1,1
    f2[1] ← e2 + a2,1
    for j ← 2 to n
        do if f1[j - 1] + a1,j ≤ f2[j - 1] + t2,j-1 + a1,j
            then f1[j] ← f1[j - 1] + a1, j
            l1[j] ← 1
        else f1[j] ← f2[j - 1] + t2,j-1 + a1,j
            l1[j] ← 2
        if f2[j - 1] + a2,j ≤ f1[j - 1] + t1,j-1 + a2,j
            then f2[j] ← f2[j - 1] + a2,j
            l2[j] ← 2
        else f2[j] ∞ f1[j - 1] + t1,j-1 + a2,j
            l2[j] ← 1
    if f1[n] + x1 ≤ f2[n] + x2
            then f* = f1[n] + x1
            l* = 1
    else f* = f2[n] + x2
        l* = 2
```

Matrix chain multiplication

Same problem is also known as Matrix Chain Ordering Problem or Optimal-parenthesization of matrix problem.

Given a sequence of matrices, M = M1,…, Mn. The goal of this problem is to find the most efficient way to multiply these matrices. The guild is not to perform the actual multiplication, but to decide the sequence of the matrix multiplications, so that the result will be calculated in minimal operations.

To compute the product of two matrices of dimensions pXq and qXr, pqr number of operations will be required. Matrix multiplication operations are associative in nature. Therefore, matrix multiplication can be done in many ways.

For example, M1, M2, M3 and M4, can be fully parenthesized as:
(M1· (M2· (M3·M4)))
(M1· ((M2·M3)· M4))
((M1·M2)· (M3·M4))
(((M1·M2)· M3)· M4)
((M1· (M2·M3))· M4)

For example,
Let M1 dimensions are 10×100, M2 dimensions are 100×10, and M3 dimensions are 10×50.
((M1·M2)· M3) = (10*100*10) + (10*10*50) = 15000
(M1· (M2·M3) = (100*10*50) + (10*100*50) = 100000

Therefore, in this problem we need to parenthesize the matrix chain so that total multiplication cost is minimized.

Given a sequence of n matrices M1, M2,… Mn. And their dimensions are p0, p1, p2,…, pn. Where matrix Ai has dimension $p_{i-1} \times p_i$ for $1 \le i \le n$. Determine the order of multiplication that minimizes the total number of multiplications.

If you try to solve this problem using the brute-force method, then you will find all possible parenthesization. Then you will compute the cost of multiplications. Thereafter you will pick the best solution. This approach will be exponential in nature.

There is an insufficiency in the brute force approach. Take an example of M1, M2,…, Mn. When you have calculated that ((M1·M2) · M3) is better than (M1· (M2·M3) so there is no point of calculating then combinations of (M1· (M2·M3) with (M4, M5…. Mn).

Optimal substructure:
Assume that M (1, N) is the optimum cost of production of the M1,…, Mn.

A list p [] to record the dimensions of the matrices.
P [0] = row of the M1
p[i] = col of Mi 1<=i<=N

For some k
M(1,N) = M(1,K) + M(K+1,N) + p0*pk*pn

If M (1, N) is minimal then both M (1, K) & M (K+1, N) are minimal.

Otherwise, if there is some M'(1, K) is there whose cost is less than M (1.. K), then M (1.. N) can't be minimal and there is a more optimal solution possible.

For some general i and j.
M(i,j) = M(i,K) + M(K+1,j) + pi-1*pk*pj

Recurrence relation:

$$M(i,j) = \begin{cases} 0 \text{ if } i = j \\ \min \{M(i,k) + M(k,j) + pi-1 * pk * pj\} \ i \le k < j \end{cases}$$

Overlapping Sub-problems:

M(1,4)

M(1,1) M(2,4)

M(1,2) M(3,4) M(1,3) M(4,4)

M(2,2) M(3,4) M(2,3) M(4,4)
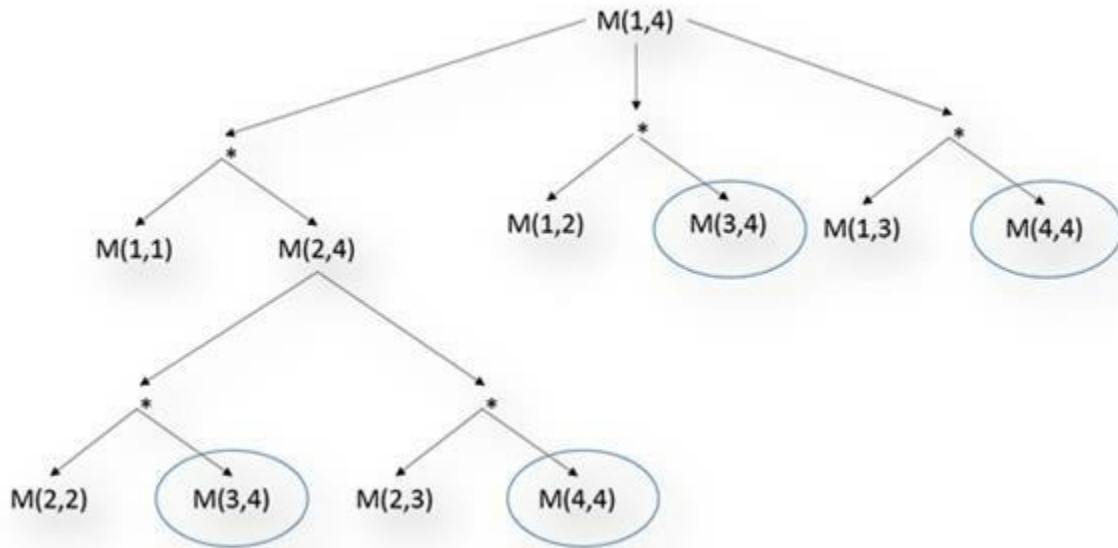
Directly calling recursive function will lead to calculation of same sub-problem multiple times. This will lead to exponential solution.

```
Algorithm MatrixChainMultiplication(p[])
    for i := 1 to n
        M[i, i] := 0;
        for l = 2 to n // l is the moving line
            for i = 1 to n − l +1
                j = i + l − 1;
                M[i, j] =  min {M(i, k) + M(k, j) + pi − 1 * pk * pj }
i ≤ k < j
```

Time Complexity will $O(n^3)$
Constructing optimal parenthesis Solution

Use another table s[1..n, 1..n]. Each entry s[i, j] records the value of k such that the optimal parenthesization of Mi Mi+1...Mj splits the product between Mk and Mk+1.

```
Algorithm MatrixChainMultiplication(p[])
    for i := 1 to n
        M[i, i] := 0;
    for l = 2 to n // l is the moving line
        for i = 1 to n − l +1
            j = i + l − 1;
            M[i, j] =  min {M(i, k) + M(k, j) + pi − 1 * pk * pj }
i ≤ k < j

            S[i, j] = k for min {M(i, k) + M(k, j) + pi − 1 * pk * pj }
i ≤ k < j
```

```
Algorithm MatrixChainMultiplication(p[])
    for i := 1 to n
    M[i, i] := 0;
    for l = 2 to n // l is the moving line
```

```
        for i = 1 to n − l +1
            j = i + l − 1;
            for k = i to j
                if( (M(i, k) + M(k, j) + pi − 1 * pk * pj ) < M[i, j]
                    M[i, j] = (M(i, k) + M(k, j) + pi − 1 * pk * pj )
                    S[i, j] = k
```

Algorithm PrintOptimalParenthesis(s[], i, j)
   If i = j
      Print Ai
   Else
      Print "("
      PrintOptimalParenthesis(s[], i,s[i, j])
      PrintOptimalParenthesis(s[], s[i, j],j)
      Print ")"

Longest Common Subsequence

Let X = {x1, x2,...., xm} is a sequence of characters and Y = {y1, y2,..., yn} is another sequence.

Z is a subsequence of X if it can be driven by deleting some elements of X. Z is a subsequence of Y if it can be driven by deleting some elements form Y. Z is LCS of it is subsequence to both X and Y, and length of all the subsequence is less than Z.

Optimal Substructure:

Let X = < x1, x2, ..., xm > and Y = < y1, y2, ..., yn > be two sequences, and let Z = < z1, z2, ..., zk > be a LCS of X and Y.

- If xm = yn, then zk = xm = yn ⟹ Zk−1 is a LCS of Xm−1 and Yn−1
- If xm != yn, then:
  - o zk != xm ⟹ Z is an LCS of Xm−1 and Y.
  - o zk != yn ⟹ Z is an LCS of X and Yn−1.

Recurrence relation
Let c[i, j] be the length of the longest common subsequence between X = {x1, x2,...., xi} and Y = {y1, y2,..., yj}.
Then c[n, m] contains the length of an LCS of X and Y

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } xi = yj \\ max(c[i-1, j], c[i, j-1]) & \text{otherwise} \end{cases}$$

Algorithm LCS(X[], m, Y[], n)
   for i = 1 to m
      c[i,0] = 0
   for j = 1 to n
      c[0,j] = 0;

```
    for i = 1 to m
        for j = 1 to n
            if X[i] == Y[j]
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = ↖
            else
                if c[i-1,j] ≥ c[i,j-1]
                    c[i,j] = c[i-1,j]
                    b[i,j] = ↑
                else
                    c[i,j] = c[i,j-1]
                    b[i,j] = ←
```

```
Algorithm PrintLCS(b[],X[], i, j)
    if i = 0
        return
    if j = 0
        return
    if b[i, j] = ↖
        PrintLCS (b[],X[], i − 1, j − 1)
        print X[i]
    else if b[i, j] = ↑
        PrintLCS (b[],X[], i − 1, j)
    else
        PrintLCS (b[],X[], i, j − 1)
```

Coin Exchanging problem

How can a given amount of money N be made with the least number of coins of given denominations D= {d1… dn}?

For example, Indian coin system {5, 10, 20, 25, 50,100}. Suppose we want to give change of a certain amount of 40 paisa.

We can make a solution by repeatedly choosing a coin ≤ to the current amount, resulting in a new amount. The greedy solution always choose the largest coin value possible.
For 40 paisa: {25, 10, and 5}

This is how billions of people around the globe make change every day. That is an approximate solution of the problem. But this is not the optimal way, the optimal solution for the above problem is {20, 20}

Step (I): Characterize the structure of a coin-change solution.
Define C [j] to be the minimum number of coins we need to make a change for j cents.

If we knew that an optimal solution for the problem of making change for j cents used a coin of denomination di, we would have:
$C[j] = 1 + C[j − di]$

Strep (II): Recursively defines the value of an optimal solution.

$$c[j] = \begin{cases} \text{infinite} & \text{if } j < 0 \\ 0 & \text{if } j = 0 \\ 1 + \min(c[j - d_i]) \ 1 \le i \le k & \text{if } j \ge 1 \end{cases}$$

Step (III): Compute values in a bottom-up fashion.
Algorithm CoinExchange(n, d[], k)
   C[0] = 0
   for j = 1 to n do
      C[j] = infinite
   for i = 1 to k do
      if j < di and 1+C[j − di] < C[j] then
         C[j] = 1+C[j − di]
   return C
Complexity: O(nk)

Step (iv): Construct an optimal solution
We use an additional list Deno[1.. n], where Deno[j] is the denomination of a coin used in an optimal solution.
Algorithm CoinExchange(n, d[], k)
   C[0] = 0
   for j = 1 to n do
      C[j] = infinite
   for i = 1 to k do
      if j < di and 1+C[j − di] < C[j] then
         C[j] = 1+C[j − di]
         Deno[j] = di
   return C
Algorithm PrintCoins(Deno[], j)
   if j > 0
      PrintCoins (Deno, j −Deno[j])
   print Deno[j]

CHAPTER 19: BACKTRACKING

Introduction

Suppose there is a lock, which produce some "click" sound when correct digit is selected for any level. To open it you just need to find the first digit, then find the second digit, then find the third digit and done. This will be a greedy algorithm and you will find the solution very quickly.

However, let us suppose the lock is some old one and it creates same sound not only at the correct digit, but also at some other digits. Therefore, when you are trying to find the digit of the first ring, then it may product sound at multiple instances. So, at this point you are not directly going straight to the solution, but you need to test various states and in case those states are not the solution you are looking for, then you need to backtrack one step at a time and find the next solution. Sure, this intelligence/ heuristics of click sound will help you to reach your goal much faster. These functions are called Pruning function or bounding functions.

Problems on Backtracking Algorithm

N Queens Problem

There are N queens given, you need to arrange them in a chessboard on NxN such that no queen should attach each other.

```
def Feasible(queen, k)
    i = 0
    while i < k
        if queen[k] == queen[i] or (queen[i] - queen[k]).abs == (i - k).abs then
            return false
        end
        i += 1
    end
    return true
end

def NQueens(queen, k, n)
    if k == n then
        printQueens(queen, n)
        return
    end
    i = 0
    while i < n
        queen[k] = i
        if Feasible(queen, k) then
            NQueens(queen, k + 1, n)
        end
        i += 1
```

```
    end
end

queen = Array.new(8)
NQueens(queen, 0, 8)
```

Tower of Hanoi

The Tower of Hanoi puzzle, disks need to be moved from one pillar to another such that any large disk cannot rest above any small disk.

This is a famous puzzle in the programming world; its origins can be tracked back to India. "There is a story about an Indian temple in Kashi Viswanathan which contains a large room with three timeworn posts in it surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient Hindu prophecy, have been moving these disks, in accordance with the immutable rules of the Brahma the creator of the universe, since the beginning of time. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the prophecy, when the last move of the puzzle will be completed, the world will end."   ;) ;) ;)
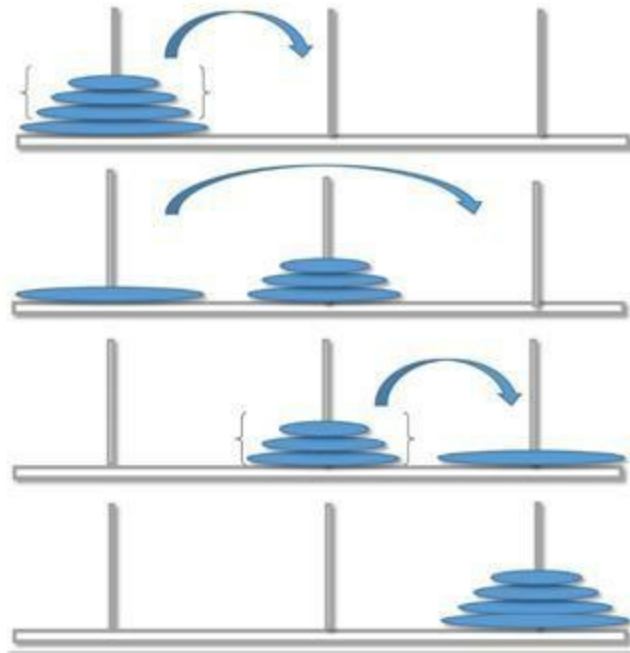

```
def TOHUtil(num, from, to, temp)
    if num < 1 then
        return
    end
    TOHUtil(num - 1, from, temp, to)
    print "Move disk " , num , " from peg " , from , " to peg " , to, "\n"
    TOHUtil(num - 1, temp, to, from)
end
def TowersOfHanoi(num)
    print "The sequence of moves involved in the Tower of Hanoi are : \n"
    TOHUtil(num, 'A', 'C', 'B')
end
```

CHAPTER 20: COMPLEXITY THEORY AND NP COMPLETENESS

Introduction

Computational complexity is the measurement of how much resources are required to solve some problem.

There are two types of resources:
1. Time: how many steps are taken to solve a problem
2. Space: how much memory is taken to solve a problem.

Decision problem

Much of Complexity theory deals with decision problems. A decision problem always has a yes or no answer.

Many problems can be converted to a decision problem that have answer as yes or no. For example:
1. Searching: The problem of searching element can be a decision problem if we ask, "Find if a particular number is there in the list ?".

2. Sorting of list and find if the list is sorted, you can make a decision problem, "Is the list is sorted in increasing order?".

3. Graph colouring algorithms: this is can also be converted to a decision problem. Can we do the graph colouring by using X number of colours?

4. Hamiltonian cycle: Is there is a path from all the nodes, each node is visited exactly once and comes back to the starting node without breaking?

Complexity Classes

Problems are divided into many classes based on their difficulty. Or how difficult it is to find if the given solution is correct or not.

Class P problems

The class P consists of a set of problems that can be solved in polynomial time. The complexity of a P problem is $O(n^k)$ where n is input size and k is some constant (it cannot depend on n).

Class P Definition: The class P contains all decision problems for which a Turing machine

algorithm leads to the "yes/no" answer in a definite number of steps bounded by a polynomial function.
For example:
Given a sequence a1, a2, a3…. an. Find if a number X is there in this list.
We can search, the number X in this list in linear time (polynomial time)

Another example:
Given a sequence a1, a2, a3…. an. If we are asked to sort the sequence.
We can sort and array in polynomial time using Bubble-Sort, this is also linear time.

Note: **O(logn)** is also polynomial. Any algorithm, which has complexity less than some O (nk), is also polynomial.

Some problem of P class is:
1. Shortest path
2. Minimum spanning tree
3. Maximum problem.
4. Max flow graph problem.
5. Convex hull

Class NP problems

Set of problems for which there is a polynomial time checking algorithm. Given a solution if we can check in a polynomial time if that solution is correct or not then, the problem is NP problem.

Class NP Definition: The class NP contains all decision problems for which, given a solution, there exists a polynomial time "proof" or "certificate" that can verify if the solution is the right "yes/no" answer
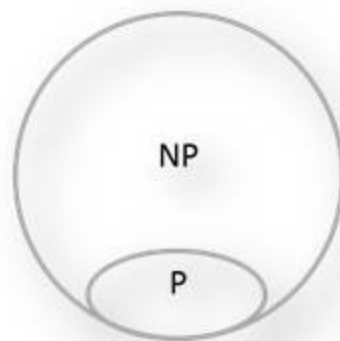
Note: There is no guarantee that you will be able to solve this problem in polynomial time. However, if a problem is an NP problem, then you can verify an answer in polynomial time.

NP does not means "non-polynomial". Actually, it is "Non-Deterministic Polynomial" type of problem. They are the kind of problems that can be solved in polynomial time by a Non-Deterministic Turing machine. At each point, all the possibilities are executed in parallel. If there are n possible choices, then all n cases will be executed in parallel. We do not have non-deterministic computers. Do not be confused with parallel computing because the number of CPU is limited in parallel computing it may be 16 core or 32 core, but it cannot be N-Core.
In short NP problems are those problems for which, if a solution is given, We can verify that

solution (if it is correct or not) in polynomial time.

Boolean Satisfiability problem

A Boolean formula is satisfied if there exist some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.

$$(A1 \lor A2 \ldots) \land (A2 \lor A4..) \ldots \land (.. \lor AN)$$

There are in total N Different Boolean Variables A1, A2… AN. There are an M number of brackets. Each bracket has K variables.

There is N variable so the number of solutions will be 2^n

And to verify if the solutions really evaluate the equation to 1 will take total ($2^n * km$) steps
In given solution of this problem you can find if the formula satisfies or not in KM steps.

Hamiltonian cycle

Hamiltonian cycle is a path from all the nodes of a graph, each node is visited exactly once and come back to the starting node without breaking.

This is an NP problem, if you have a solution to it, then you just need to see if all the nodes are there in the path and you came back to where you have started and you are done? The checking is done in linear time and you are done.

Determining whether a directed graph has a Hamiltonian cycle or not does not have a polynomial time algorithm. O(n!)

However, if someone has given you a sequence of vertices, determining whether that sequence forms a Hamiltonian cycle can be done in polynomial time (Linear time).
Hamiltonian cycles are in NP

Clique Problem

In a graph given if there is a clique of size K or more. A clique is a subset of nodes, which are completely connected to each other.
This problem is NP problem. Given a set of nodes, you can very easily find out whether it is a clique or not.

For example:

Clique of size 4
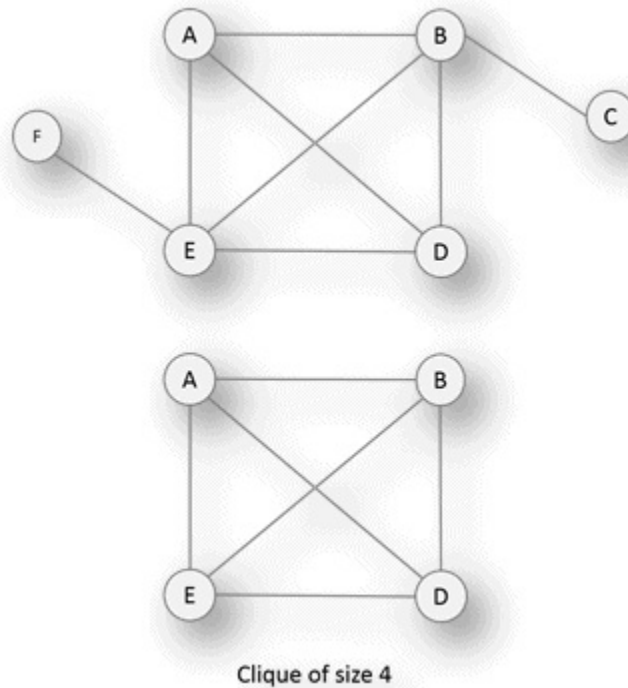
Prime Number

Finding Prime number is NP. Given a solution, it is easy to find if it is a Prime or not in polynomial time. Finding prime numbers is important which is heavily used in cryptography.

```
def isPrime(n)
    answer = (n > 1) ? true : false
    i = 2
    while i * i <= n
        if n % i == 0 then
            answer = true
            break
        end
        i += 1
    end
    return answer
end
```

Checking will happen until the square root of number so the Time Complexity will be O(√n). Hence, prime number finding is an NP problem as we can verify the solution in polynomial time.

Graph theory have wonderful set of problems
- Shortest path algorithms?
- Longest path is NP complete.
- Eulerian tours is a polynomial time problem.
- Hamiltonian tours is a NP complete

Class co-NP

Set of problems for which there is a polynomial time checking algorithm. Given a solution, if we can check in a polynomial time if that solution is incorrect the problem is co-NP problem.

Class co-NP Definition: The class co-NP contains all decision problems such that there exists a polynomial time proof that can verify if the problem does not have the right "yes/no" answer.
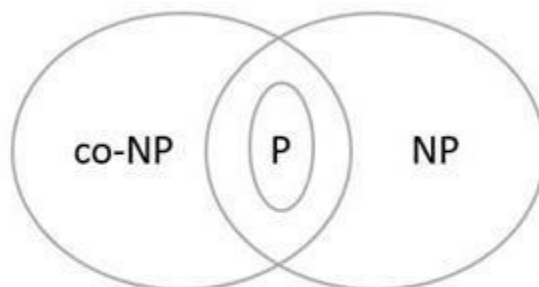
Relationship between P, NP and co-NP

Class P is Subset of Class NP

All problems that are P also are NP ($P \subseteq NP$). Problem set P is a subset of problem set NP.
Searching
If we have some number sequence a1, a2, a3…. an.   We already know that searching a number X inside this list is of type P.
If it is given that number X is inside this sequence, then we can verify by looking into every entry again and find if the answer is correct in polynomial time (linear time.)

Sorting
Another example of sorting a number sequence, if it is given that the list b1, b2, b3.. bn is a sorted then we can loop through this given list and find if the list is really sorted in polynomial time (linear time again.)

NP–Hard:

A problem is NP-Hard if all the problems in NP can be reduced to it in polynomial time.
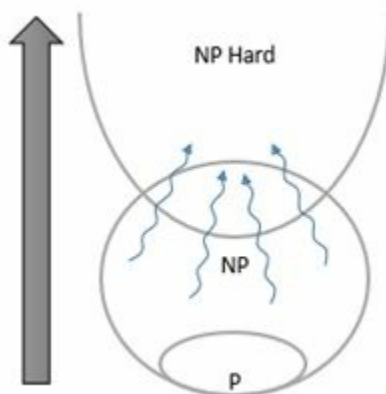
NP–Complete Problems

Set of problem is NP-Complete if it is an NP problem and an NP-Hard problem.
It should follow both the properties:

1) Its solutions can be verified in a polynomial time.
2) All problems of NP are reduced to NP complete problems in polynomial time.

You can always reduce any NP problem into NP-Complete in polynomial time. In addition, when you get the answer to the problem, then you can verify this solution in polynomial time.
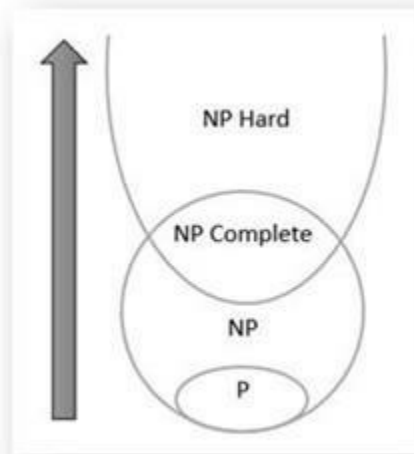
Any NP problem is polynomial reduced to NP-Complete problem, if we can find a solution to a single NP-Complete problem in polynomial time, then we can solve all the NP problems in polynomial time. However, so far no one is able to find any solution of NP-Complete problem in polynomial time.

$P \neq NP$

Reduction

It is a process of transformation of one problem into another problem. The transformation time should be polynomial. If a problem A is transformed into B and we know the solution of B in polynomial time, then A can also be solved in polynomial time.

For example,

Quadratic Equation Solver: We have a Quadratic Equation Solver, which solves equation of the form $ax^2 + bx + c = 0$. It takes Input a, b, c and generates output r1, r2.

Now try to solve a linear equation 2x+4=0. Using reduction second equation can be transformed to the first equation.

$2x+4 = 0$

$X^2 + 2x + 4 = 0$

ATLAS: We have an atlas and we need to colour maps so that no two countries have the same colour. Let us suppose below is the various countries. Moreover, different patterns represent different colour.
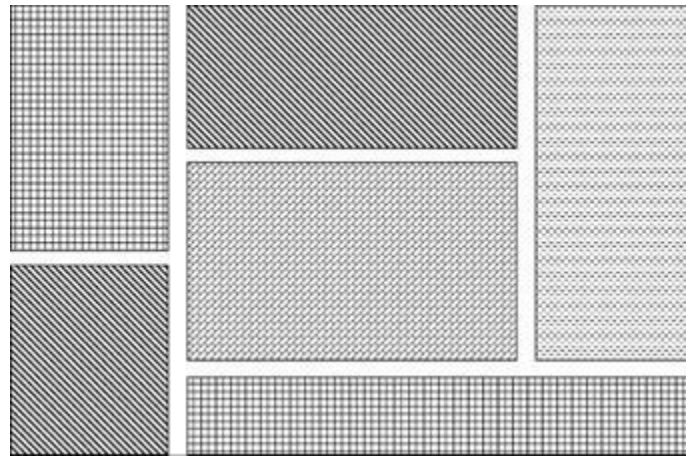
We can see that same problem of atlas colouring can be reduced to graph colouring and if we know the solution of graph colouring then same solution can work for atlas colouring too. Where each node of the graph represents one country and the adjacent country relation is represented by the edges between nodes.
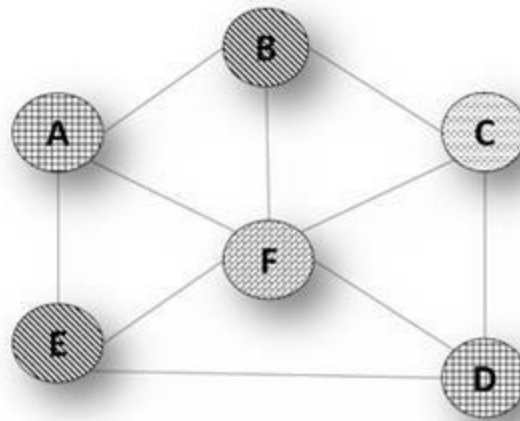
The sorting problem reduces (≤) to Convex Hull problem.
SAT reduces (≤) to 3SAT

Traveling Salesman Problem (TSP)

The traveling salesman problem tries to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.
Alternatively find the shortest Hamiltonian circuit in a weighted connected graph. A cycle that passes through all the vertices of the graph exactly once.

```
Algorithm TSP
Select a city
MinTourCost = infinite
For ( All permutations of cities ) do
    If( LengthOfPathSinglePermutation < MinTourCost )
        MinTourCost = LengthOfPath
```

Total number of possible combinations = (n-1)!
Cost for calculating the path: $\Theta(n)$
So the total cost for finding the shortest path: $\Theta(n!)$

It is an NP-Hard problem there is no efficient algorithm to find its solution. Even if some solution is given, it is equally hard to verify that this is a correct solution or not.   However, some approximate algorithms can be used to find a good solution. We will not always get the best solution, but will get a good solution.

Our approximate algorithm is based on the minimum spanning tree problem. In which we have to construct a tree from a graph such that every node is connected by edges of the graph and the total sum of the cost of all the edges is minimum.
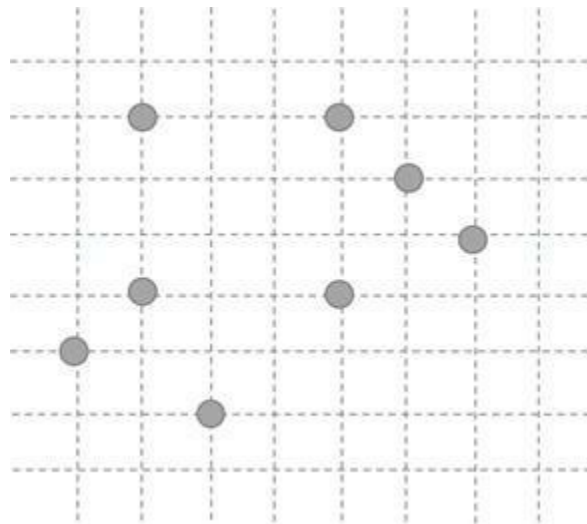
In the above diagram, we have a group of cities (each city is represented by a circle.) Which are located in the grid and the distance between the cities is same as per the actual distance. And there is a path from each city to another city which is a straight path from one to another.
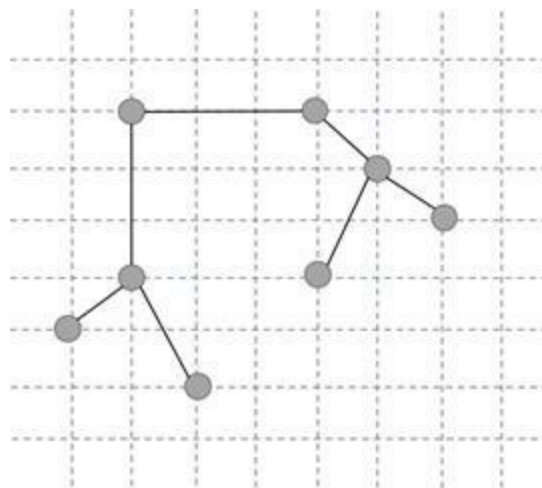
We have made a minimum spanning tree for the above city graph.

What we want to prove that the shortest path in a TSP will always be greater than the length of MST. Since in MST all nodes are connected to the next node, which is also the minimum distance from the group of node. Therefore, to make it a path without repeating the nodes we need to go directly from one node to other without following MST. At that point, when we are not following MST we are choosing an edge, which is grater, then the edges provided by MST. So TSP path will always be greater than or equal to MST path.
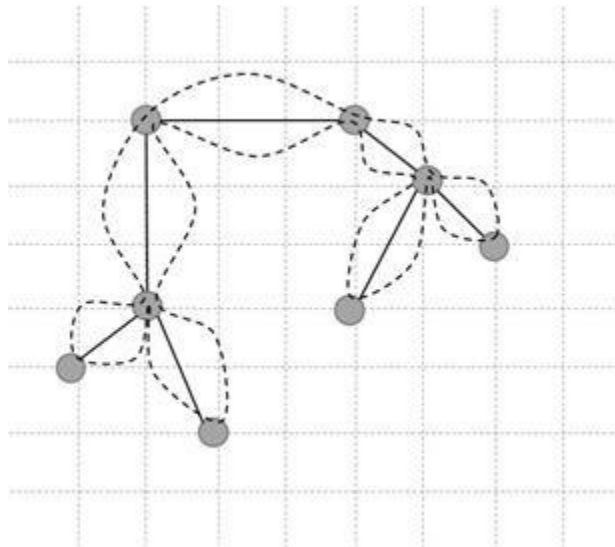
Now let us take a path from starting node and traverse each node on the way given above and then come back to the starting node. The total cost of the path is 2MST. The only difference is that we are visiting many nodes multiple times.
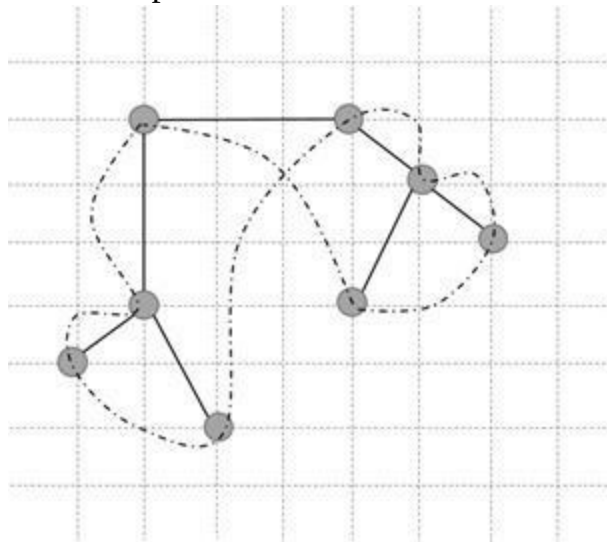
Now let us change our traversal algorithm so that it will become TSP in our traversal, we did not visit an already visited node we will skip them and will visit the next unvisited node. In this algorithm, we will reach the next node by as shorter path. (The sum of the length of all the edges of a polygon is always greater than a single edge.) Ultimately, we will get the TSP and its path length is no more than twice the optimal solution. Therefore, the proposed algorithm gives a good result.

End Note

Nobody has come up with such a polynomial-time algorithm to solve a NP-Complete problem. Many important algorithms depends upon it. However, at the same time nobody has proven that no polynomial time algorithm is possible. There is a million US dollars for anyone who can solve any NP Complete problem in polynomial time. The whole economy of the world will fall as most of the banks depend on public key encryption which will be easy to break if P=NP solution is found.

APPENDIX

Appendix A

Algorithms	Time Complexity
Binary Search in a sorted array of N elements	O(logN)
Reversing a string of N elements	O(N)
Linear search in an unsorted array of N elements	O(N)
Compare two strings with lengths L1 and L2	O(min(L1, L2))
Computing the Nth Fibonacci number using dynamic programming	O(N)
Checking if a string of N characters is a palindrome	O(N)
Finding a string in another string using the Aho-Corasick algorithm	O(N)
Sorting an array of N elements using Merge-Sort/Quick-Sort/Heap-Sort	O(N * LogN)
Sorting an array of N elements using Bubble-Sort	O(N!)
Two nested loops from 1 to N	O(N!)
The Knapsack problem of N elements with capacity M	O(N * M)
Finding a string in another string – the naive approach	O(L1 * L2)
Three nested loops from 1 to N	$O(N^3)$
Twenty-eight nested loops … you get the idea	$O(N^{28})$
Stack	
Adding a value to the top of a stack	O(1)
Removing the value at the top of a stack	O(1)
Reversing a stack	O(N)
Queue	
Adding a value to end of the queue	O(1)
Removing the value at the front of the queue	O(1)
Reversing a queue	O(N)
Heap	
Adding a value to the heap	O(logN)
Removing the value at the top of the heap	O(logN)
Hash	

Adding a value to a hash	O(1)
Checking if a value is in a hash	O(1)

Table of Contents