ADVANCED DATA STRUCTURES

THEORY AND APPLICATIONS

SUMAN SAHA
SHAILENDRA SHUKLA

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

ADVANCED DATA STRUCTURES
THEORY AND APPLICATIONS

SUMAN SAHA
SHAILENDRA SHUKLA

CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Advanced Data Structures
Theory and Applications

Advanced Data Structures

Theory and Applications

Suman Saha
Shailendra Shukla

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-1-138-59260-5 (Hardback)

Library of Congress Cataloging-in-Publication Data

Names: Saha, Suman, author. | Shukla, S. K. (Shailendra Kumar), 1969-author.
Title: Advanced data structures : theory and applications / Suman Saha, Shailendra Shukla.
Description: Boca Raton : Taylor & Francis, a CRC title, part of the Taylor & Francis imprint, a member of the Taylor & Francis Group, the academic division of T&F Informa, plc, 2019. | Includes bibliographical references and indexes.
Identifiers: LCCN 2019011938| ISBN 9781138592605 (hardback : acid-free paper) | ISBN 9780429488757 (ebook)
Subjects: LCSH: Data structures (Computer science)
Classification: LCC QA76.9.D35 S23 2019 | DDC 005.7/3--dc23
LC record available at https://lccn.loc.gov/2019011938

Visit the Taylor & Francis Web site at

Dedicated to our students

Contents

Index

FOREWORD

Current trends in computer science and engineering are driven mostly by data science, machine learning, the Internet of Things, cryptography, and other practical uses. Therefore, a complete book on advanced data structures that will work as a backbone in explaining these modern subjects is much sought after.

However, writing such a book is challenging because it must meet two goals: allowing students to understand the fundamentals of advanced data structures and enabling them to apply in-depth knowledge in their chosen areas. This book is one of the first efforts in this direction and it will play an indispensable role for students in computer science and related disciplines to enhance their expertise in advanced data structures and applications.

The authors are members of the faculty at Jaypee University of Information Technology. They have more than ten years of teaching experience and have published a wide range of peer-reviewed articles in many areas of information technology. They possess unique skills in analyzing and explaining difficult algorithms and data structures simply and elegantly. Their expertise in all areas of computer science equips them to be suitable authors of a comprehensive book about data structures.

The book chapters are well organized. Part I covers theoretical advancements in basic data structures. Part II discusses evolving paradigms of data structures, and Part III details recent applications. All the material is organized logically and both students and researchers can benefit from reading this book and applying its concepts.

I hope this book will serve as a useful tool for all inquisitive students who plan to apply recent developments in data structures to their work.

A. Chattopadhyay

Dr. Amit Chattopadhyay
International Institute of Information Technology
Bangalore, India

ACKNOWLEDGMENTS

It has been almost two years since we started writing a book about advanced data structures. This book was kept on track and seen to completion with the support and encouragement of numerous family members, friends, colleagues, teachers, and students. It is a pleasure to express our thanks to all those who contributed in so many ways to our publication success.

We are very thankful to Dr. Amit Chattopadhyay of the International Institute of Information Technology in Bangalore for reviewing our work and writing a favorable foreword message. Our heartfelt thanks also extend to our friends and colleagues for their help and encouragement.

We express our sincere thanks to the students and colleagues of Jaypee University of Information Technology, Solan, Himachal. We also appreciate the excellent work of the editorial and production teams of CRC Press and Taylor & Francis. Thanks are also due to all the reviewers who evaluated this book and provided suggestions and advice.

Readers' constructive feedback and suggestions for improvement may be emailed to suman.saha@juit.ac.in or shailendra.shukla@juit.ac.in.

PREFACE

Advanced data structures are usually covered in core courses included in most graduate programs in computer science, engineering, and allied disciplines, during the first year or first semester of the curriculum. The objective of the course is to provide students with a much-needed foundation for advancing their technical skills and sharpening their problem-solving abilities in their respective disciplines.

Although many technical universities have offered advanced data structure courses for decades, major technology advances triggered a recent paradigm shift that focuses on huge databases and Internet-based technologies. Most technical universities have redefined their course contents to meet current needs and rely heavily on research papers because of the lack of comprehensive texts on advanced data structures.

To the best of our knowledge, existing textbooks on advanced data structures provide only partial coverage of the subject. This book evolved from materials the authors developed over several years while teaching advanced data structures at Jaypee University of Information Technology. The course was designed for advanced graduate students, although it has become accessible to advanced undergraduates and beginner researchers in the areas of computer science and bio-informatics.

This book emphasizes recent evolutions of data structures to fit new paradigms of computation and applications to various domains of computer science. We included illustrative problems, review questions, and programming projects to enable students to comprehend, implement, and appreciate advanced data structures.

The book is divided into three parts. Part I details basic data structure advancements such as cuckoo hashing, skiplists, tango trees, Fibonacci heaps, and index files. As an introduction, the first chapter discusses the need for advanced data structures and some basic concepts pertaining to amortized analysis of algorithms.

Part II covers data structures of evolving data domains and dedicates chapters to spatial and temporal data structures, external memory data structures, and distributed and streaming data structures.

Part III elucidates the applications of data structures to various areas of computer science—for example, cryptography, the Internet, networking, the Internet of Things, and images and graphics.

The concepts and techniques behind the data structures and their applications are explained. Every chapter includes a variety of illustrative problems pertaining to technical content, a summary of the concepts, and review questions to reinforce comprehension of the material.

This book may be used as an introductory or advanced-level text for undergraduates, graduate students, and participants in research programs that offer advanced data structures as a core or elective course. While the book is primarily intended for classroom use, it could also serve as a starting point for researchers working in specialized computer science areas.

AUTHORS

Dr. Suman Saha spent the last 14 years performing research in the areas of data and information science, specifically working on information retrieval, web mining, decision theory, social network analysis, and big data technologies. He started his research in the field of web mining by working as a senior research scientist at the Center for Soft Computing Research of the Indian Statistical Institute in Kolkata for almost five years. After that, he joined the Department of Computer Science at Jaypee University of Information Technology, Himachal, India as an assistant professor. For the past eight years, he has continued his research, taught classes, and handled other departmental responsibilities.

Dr. Saha earned a PhD from Jaypee University after pursuing a master's in computer science from the Indian Statistical Institute in Calcutta and a second master's in mathematics from the University of Calcutta. His thesis was titled "Community Detection in Complex Networks: Metric Space, Nearest Neighbor Search, Low-Rank Approximation and Optimality."

During his eight years at Jaypee University of Information Technology as an assistant professor, he taught courses in advanced web mining, cloud computing, advanced algorithms, fundamentals of algorithms, advanced data structures, and other subjects. He currently supervises two PhD students and has guided 15 master's candidates and 50 undergraduates. He can be reached at https://orcid.org/0000-0003-1492-2738.

Dr. Shailendra Shukla completed a master's program in information security at the Indian Institute of Information Technology in Allahabad and then earned a PhD in computer science from the Indian Institute of Technology in Patna. His doctoral work focused on boundary detection and localization in wireless sensor networks. His thesis proposed a collection of networking algorithms that address security problems like routing in the Internet of Things, localization, boundary node detection (surveillance), virtual coordinate assignments (geography, routing, and localizations), physical cyber systems, monitoring, and surveillance. His articles have appeared in journals of Elsevier, Springer, the Institute of Electrical and Electronic Engineers, and other publishers.

Dr. Shukla continues his work as an assistant professor at Jaypee University in Waknaghat. He currently supervises two PhD candidates and five master's program students. He can be reached at https://orcid.org/0000-0002-8316-8460.

Part I

Theoretical Advancements

Chapter 1
Introduction

This book emphasises recent theoretical developments, evolving data structures for different paradigms of computations and important applications to research domains of computer science. The whole book is divided into three parts. As an introduction, the need for advanced data structures and some basic concepts pertaining to amortized analysis of algorithms have been presented in the first chapter.

Part I details advancements on basic data structures, for example, cuckoo hashing, skiplists, tango trees and Fibonacci heaps and index files. Part II details data structures of different evolving data domains such as special data structures, temporal data structures, external-memory data structures, distributed and streaming data structures. Part III elucidates the applications of these data structures on different areas of computer science such as networks, the World Wide Web, database management systems (DBMSs), cryptography, and graphics to name a few. The concepts and techniques behind each data structure and their applications have been explained. Illustrative problems, review questions, and programming assignments are included to enable the students comprehend, implement and appreciate advanced data structures. In addition to illustrative problems, each chapter includes a detailed summary of the technical content and list of review questions to reinforced comprehension of the material.

Objective 1.1 — Advanced data structure. The rapid development of different domains (e.g. data science, the Internet of Things (IOT), artificial intelligence (AI), machine learning (ML), cloud computing and others) of computer science imposes new challenges to the field of data structures and algorithms. The traditional data structures are mainly developed to perform sequential point search on RAM models and assessment based on computational complexity is insufficient to meet the requirements of modern computer science. In this book we have incorporated important data structural concepts to be taught as an advanced course on data structure intended to meet contemporary computational challenges.

Note We assume students understand basic data structures like arrays, link lists, stacks, queues, trees, hashing, and other relevant concepts of data structures and algorithms. Those materials are not included in this book, but their advancements are demonstrated along with the other advanced topics.

In this chapter, the need for advanced data structures and some basic concepts pertaining to design of data structures, amortized analysis of queries over data structures and how to use data structures to solve computational problems have been presented. The last section of the chapter is included to describe organization of the book.

1.1 Data Structure

A data structure is a typical way of organizing data in a computer so that it can be accessed efficiently [1].

Definition 1.1.1 — Data structure. A data structure is an organization of data values, the relationships among them, and the functions to answer particular queries on that data.

A data structure can implement one or many abstract data types (ADT), which specify the operations that can be performed in that data structure. Moreover, the computational complexity of those operations, described in the ADT will be adopted in the data structure. In comparison, a data structure is a real implementation of the theoretical specification abstracted in an ADT.

Definition 1.1.2 — Abstract data type. An abstract data type (ADT) is a mathematical abstraction about data and its functions.

Different kinds of data structures are generally used in different problems. A data structure can implement an abstract data types, and adopt the specifications of operations and complexities. Different application tasks are often solved using various types of data structures, some of which are highly specialized for specific tasks. Data structures provide mechanisms to organize and manage large amounts of data efficiently for several uses in most domains of computer science. To be more specific, the efficient data structures are the key to designing efficient algorithms whatever the domain may be. Some formal design paradigm and programming languages put more emphasis on data structures than algorithms. They argue that data structures are the key organizing factors in software design. Data structures describe the access patterns of data stored in both main and secondary memories. To implement a data structure, we usually require to write a set of procedures that access and organize data instances of that structure. The efficiency of a data structure can be analyzed by counting the number of times a particular operation is performed. This is the underlying theoretical concept of an abstract data type. A data structure can be defined alternatively by the designs of its access mechanisms and operations sequences.

1.2 Design of Data Structure

This section focuses on data structure design. Practitioners understand that data should be specified on two levels based on the abstract user-oriented information structure and the concrete machine-oriented storage structure of the data. The design methodology of data structure is based on many views such as data reality, data abstraction, information structure, storage structure, and machine encoding to name a few. The design of a data structure should proceed through successive levels by specifying important aspects necessary within levels and binding the different aspects across the levels. Users must be able to consider all aspects of data and must be able to recognize uses for commonly studied data structures for compatibility of a data structure to an application algorithm. In order to do this, they must be aware of the various data structures which could be used to solve a problem, the particular processes required, what trade-offs may occur in the selection of one option over another, and how to make a reasonable choice given all of these considerations. In trying to get students to match a data structure to an application, some small activities can be used to provide a flavor of what is involved in data structure selection.

For example, within a programming assignment which requires that students keep track of passengers on a limited number of airline flights, one requirement is that passengers who wish to book a full flight be kept on a waiting list.

Another task that can be included in an assignment on binary trees is to design an algorithm to print the tree in graphic form.

Both of these activities require that a student recognize the particular properties of a problem and match the processing needs to a simple data structure.

1.3 Analysis of Data Structure

In this section, we discuss briefly the common practice and widely used notions to analyze a data structure, which is designed to handle a particular type of data (generally known as data domain) and also accept specific queries drawn from a domain of possible queries. Let D be the domain of data and Q be the domain of possible queries. In case of static data structure, let f be a function defined as $f{:}Q \times D \rightarrow A$, i.e. $a = f(x, y)$ is the answer to query x about data y. For example, if the function returns true or false, then A is Boolean, but it may be something else [2]. To understand the complexity of the process we need to include three more parameters s, b, and t, where s denotes the size of the memory cell containing b bits and t is the time for accessing data points to answer a query in a random access machine. In general, our goal is to optimize the parameters s, t, and b are predefined parameter of the model, usually set to $O(log\ |Q|)$ or $O(poly(log\ |Q|))$. The most widely used model for proving lower bound of data structure is the "cell-probe model", introduced by Yao [3], before introduction of the notion of communication complexity [4].

Minimizing the cost per query is usually trivial for static data structures. We can always pre-compute the answers to all queries and store the answers in a memory location corresponding to that query [5]. However, the solution is undesirable because it requires a large amount of preprocessing as well as a large amount of memory. Therefore for static data structures, good understanding of query time, preprocessing time and space requirement are necessary to implement a solution.

For dynamic data structures, the problem of minimizing time per operation often becomes nontrivial, even without considering the cost of preprocessing or space. If we try to minimize the costs of queries as in the static case by maintaining the answer to each query in memory, each update may change the answers many queries and thus require many memory locations to be rewritten [6]. On the other hand, the cost of updates can be minimized by simply memorizing each update in a list without doing any processing. Answering a query is very expensive because of the need to scan the entire history. The whole difficulty of dynamic data structure is the number of memory locations that is used to record updates and the number of memory locations that need to be accessed perform queries.

1.4 Amortized Complexity

The concept of amortized analysis is to compute the time complexity on a sequence of operations rather than finding a single case of worst case time, which may be very high but rare within the sequence. To explain the situation we can take an example of monthly expenditure of a student. In the notion of time complexity we should find out a typical day with maximum expenditure (analogous to worst case complexity) and multiply it 30 times. We then estimate monthly expenses but must consider possible overestimate based on the day the student paid rent. In the paradigm of time complexity her total monthly expense in the worst case (calculated on the day rent is paid) will ignore the fact that rent is paid only once a month. The actual daily expenditure must be total spending over the month divided by thirty, not just the first days expenditure. The same solution method is adopted in the notion of amortized complexity [7,8].

Objective 1.2 — Amortized analysis. Amortized analysis is designed to reduce the total complexity of a sequence of operation which is in practice confused with worst-case run time of the operation multiplied by sequence size. In typical algorithms, certain operations may be very costly, whereas other operations may not be as costly. In amortized analysis we consider both the costly and less costly operations together on a sequential run of the algorithm and compute their average, which often provides a tighter bound of worst case complexity.

The basic concept is that a wrost case operation generally changes the state although this is not frequent. We should consider amortizing costs in a worst case situation. The three common methods of amortized analysis are:

1. Aggregate method: determines the upper bound of the total cost for a sequence of operations followed by average computation.
2. Accounting method: assign the individual amortized cost of each operation, its stored credit for non-costly operations and debit the account when costly operation occurs. This method ensures that total credit is non-negative by means of creative assignment of amortized cost. Finally, we compute the amortized cost by taking the average over the sequence of operations.
3. Potential method: similar to accounting method, but cost is associated with data structure as a potential function that is updated after each operation by adding the potential difference per operation. It can be noted that potential difference may be negative but total potential must remain positive.

▪ **Example 1.1 — Amortized analysis of dynamic array.** A dynamic array grows in size with the insertion of more elements; a typical example is the array list in Java. Consider a dynamic array of five elements and a constant time requirement. Inserting a sixth element into the array would take more time because the dynamic array would have to double in size, map the old elements into the larger array, then insert the new element. The next four insertions will require constant time plus insertion of the new element will require costly doubling of the array size. Consider an array of size n. Note that insertion operations take constant time except for the every $(kn + 1)$th insertion, which takes $O(n)$ time to execute the size doubling operation. The average of all insertions represents constant time. ▪

▪ **Example 1.2 — Amortized analysis of red-blacktrees.** Red-black tree is a very popular balanced binary search tree that use color color convention and set of rules to ensure balance structure. During dynamic update by means of insert and delete, the resulting tree fails to follow the rules assumed in the definition, which follows recursive rotation and re-coloring to re-establish balanced structure. In red-black tree every operation cost $O(\log n)$ in the worst case. Let us consider the case of inserting n more elements in a red-black tree of size n. A conventional calculation of complexity may be $O(n\log n)$, but the amortized analysis shows us $O(n)$ operation for structural reform after any n consecutive insert; we can apply this to any operation. The amortized analysis of the above example takes into account the fact that $O(\log n)$ structural reform is infrequent enough to sum up to $O(n\log n)$, but the per operation cost is only $O(1)$. The analysis can be done easily by valuing black nodes with no red children as 0, black node with one red child as 1, and black nodes with two red children as 2. ▪

1.5 Computational Models

Computational complexity is examined in concrete and abstract terms. The concrete analysis of computational limits is done using models that capture the exchange of space for time. Although many computational models exist, we mention only three that are most widely used for providing bounds of data structures.

1.5.1 RAM model

The random access machine (RAM) is the most popular computation model. It assumes all simple operations are equally costly and require one unit of time, loops and subroutines are not simple, and the memory is sufficient for one unit of time to facilitate easy calculation for algorithm developers.

However, the cost of addition and the cost of multiplication, which involves several additions, are not same in reality. The memory access in cache and disk are not same, cache access is far faster (possibly 1,000 times or more) than disk access [1]. Although there are seemingly absurd assumptions in the RAM model, in practice it is most useful probably for its robustness. The model does not depend on the configuration of the machine and it clearly indicates how the algorithm will work. The model also allows a comparison of algorithms designed to perform the same task to enable users to determine which algorithm will perform better asymptotically.

1.5.2 Word RAM model

The word RAM model is a modified abstraction of a random access machine with some additional capabilities. It can handle words up to w bits in size and store integers up to size 2^w. The model assumes that the word size should be larger than the problem size, that is, for a problem of size n, $w \geq \log n$, the model is transdichotomous. The bitwise operations such as arithmetic and logical shifts are allowed in this model and require constant time for execution. The model satisfies the bound $U \leq 2^w$, where U is the number of possible values. The word RAM model performs integer sorting very efficiently, with expected running time $O(n\log\ \log n)$ in a randomized algorithm [10,9].

1.5.3 Cell-probe model of computation

The cell-probe model represents a modification of the random access machine; it assumes all operations except memory access are free. This model is generally used to provide lower bounds of data structure operations. Computational costs is assigned only to accessing units of memory (cells) and the model represents a minor upgrade of the random access machine model (a modification of the counter machine model). Computation is treated as a problem of querying a set of memory cells. Every problem has two phases: preprocessing and querying. The input of the preprocessing state is storing a set of data as a memory cell structure. The query phase input is a query element. The system determines whether the query element is included in the memory structure. All operations except memory cell access are free [3].

This model is useful in the analysis of data structural lower bounds. The model clearly shows a minimum number of memory accesses in stored data on which we run the queries.

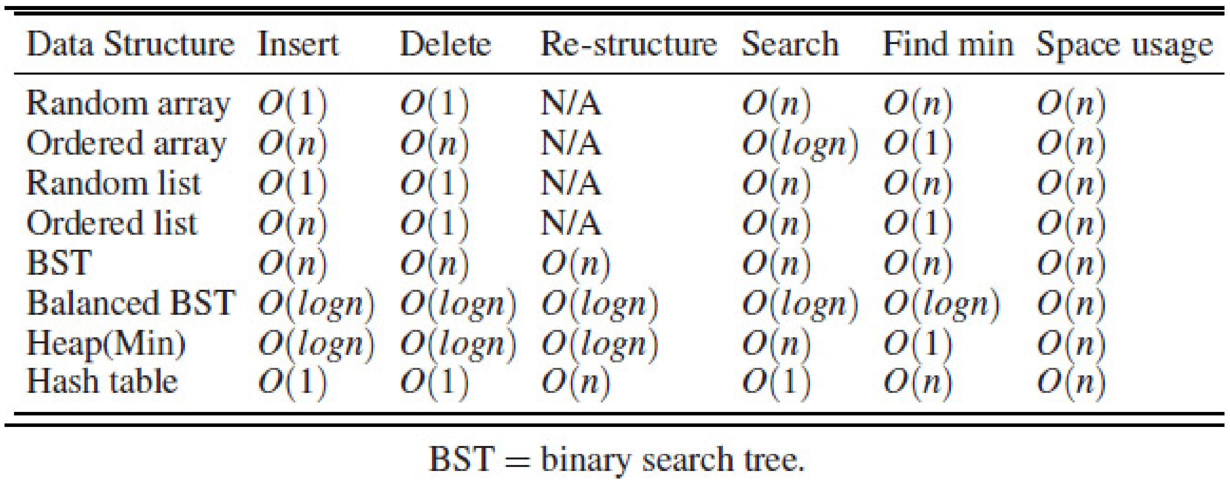
▪ **Example 1.3 — Dynamic partial sums.** A dynamic partial sum problem defines two operations: update and sum. Update (k, v) sets the value in an array A at index k to be v, whereas sum (k) returns the sum of the values in A at indices 0 through k. It take $O(1)$ time for Update and $O(n)$ time for sum.

If the values are stored as leaves in a tree and inner nodes store the values of the subtree which is rooted at that node, the Update requires $O(\log n)$ time to update each node in the leaf to root path, and sum requires $O(\log n)$ time to traverse the tree and sum the values of all subtrees left in the query index. The cell-probe model shows that the partial sums problem requires $\Omega\,(\log n)$ time per operation [11]. ▪

1.6 Bounds of Fundamental Data Structures

In this section, we discuss fundamental Data structures such as arrays, lists, trees, and queues and explain how they work and achievable results and bounds. The most common data structures used for searching unordered sequential lists of items are usually easy to utilize but less efficient. Finding the query item in such a list requires a number of operations proportional to the number of items (in worst case and average case) in the sequence, possibly with linear search method. Some good data structures developed for searching purpose provide faster retrieval, but generally increase overhead because creation and maintenance are costly. The cost of building such search data structures is at least proportional to number of elements, and often exceeds basic element costs. Static search data structures are developed for answering many queries on a fixed data set. However, dynamic structures provide the capabilities of insertion, deletion and modification of data items along with query retrieval. In the dynamic data structure, the cost of fixing the search structure also includes updates and accounting costs.

The approximate summary for basic data structure developed to perform specific queries appears in Table 1.1. There are special situations and different variants of the data structures that involve further costs. It is also possible to combine data structures to obtain lower costs [12,13].



Data Structure	Insert	Delete	Re-structure	Search	Find min	Space usage
Random array	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$
Ordered array	$O(n)$	$O(n)$	N/A	$O(logn)$	$O(1)$	$O(n)$
Random list	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$
Ordered list	$O(n)$	$O(1)$	N/A	$O(n)$	$O(1)$	$O(n)$
BST	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(logn)$	$O(n)$
Heap(Min)	$O(logn)$	$O(logn)$	$O(logn)$	$O(n)$	$O(1)$	$O(n)$
Hash table	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$

BST = binary search tree.

Table 1.1: Comparison of fundamental data structures

1.7 Lazy Delete

An alternative to a standard deletion strategy is known as lazy deletion. When deleting elements from a singly linked list, we delete them logically but not physically. This is done by marking the node as deleted (using a Boolean value). The numbers of deleted and not-deleted elements in the list are kept as part of the list. If at some point the number of deleted elements is equal to the number of not-deleted elements, we traverse the list and delete all "lazily deleted" elements. The technique has advantages and disadvantages. The main advantage is that marking a node as deleted takes less time than changing pointers. However, once a node is deleted lazily, still we need to traverse it while searching for another node and a lot of time is required to remove lazily deleted elements [14].

▪ **Example 1.4 — Hashing with lazy delete.** In hashing we index the elements of a set or collection into a table of larger size using some range bound function, which is subject to collisions. In collision resolution by probing, we shift the element to some other empty slot using some predefined strategy. During the removal of elements by hashing with collision resolution, if we remove an indexed element which was inserted first in the table followed by several collisions and consequently followed by several shiftings of later inserted elements, the later element will become inaccessible. A lazy delete is a good temporary solution for this problem without any burden of re-indexing, although it uses more memory. A delete flag will ensure the search process to answer queries correctly. ▪

1.8 Organization of Part I

Part I of this book (Chapters 1 through 4) discusses theoretical advancements in the basic data structure field, specifically in the areas of hashing, trees and lists. This chapter introduced the uses of advanced data structures, basic design concepts, amortized analysis of queries, and capabilities of data structures to solve computational problems.

The advanced hashing techniques are covered in chapter 2. The chapter first covers definitions and concepts, then proceeds to discuss collisions, birthday paradoxes, load factors, chaining and probing. It explains perfect hashing universal hashing, cuckoo hashing, bloom filters, and locality-sensitive hashing. Most of these topics were developed recently and are not covered in well known texts. Bipartite graph analysis in cuckoo hashing and probabilities of false positives in bloom filters are also included.

Chapter 3 discusses balanced binary search trees (BSTs) in depth. The chapter starts with a brief review and progresses to discussions of reducing $log(n)$ lower bounds, splay trees, tango trees, and skiplists. The final section covers static and dynamic optimality.

Chapter 4 explains some important data structures designed to answer complex point search queries. Disjoint sets and binomial heaps, both of which play important roles in major network algorithms, are detailed, along with Fibonacci heaps. Tries and inverted indices that handle keyword queries are detailed in the final section.

1.9 Exercises

Exercise 1.1 What is the difference between an array and a linked list? When might you use either data structure? ▪

Exercise 1.2 There is an array of numbers with all the elements appearing twice, and one element appearing once. How can you remove that element efficiently? ▪

Exercise 1.3 Explain whether a tree is or is not a BST. ▪

Exercise 1.4 Find the k closest points to a target. ▪

Exercise 1.5 Given a binary tree, find the greatest possible sum of the subtrees. ▪

Exercise 1.6 Remove duplicates from an unsorted linked list. ▪

Exercise 1.7 Find k-th largest element in an array. ▪

Exercise 1.8 During the execution of a task sequence, assume that every i^{th} task takes $O(1)$ time when i is not a power of two; otherwise, it takes i time. Show that amortized complexity of each task is $O(1)$. ▪

Exercise 1.9 Build a heap($[x_1, x_2, \ldots, x_n]$) operation call $n/2$ heapify(x) operation for every non-leaf elements in reverse direction, i.e. last root. The cost of heapifying (x) is height of subtree rooted at x with a maximum of $O(\log n)$ for root. Show that amortized complexity of build-heap($[x_1, x_2, \ldots, x_n]$) is $O(n)$. ▪

Chapter 2

O(1) Search by Hashing

Objective 2.1 In this chapter we have demonstrated the theoretical foundations of the fastest data structures (various types of hashing) for direct access to point query searches. We also discuss the relevant technical concepts readers will find useful for more advanced applications.

The chapter starts with definitions and concepts of basic hashing, collisions, birthday paradoxes, load factors, and well known concepts like chaining and probing. Sections are dedicated to recent topics: perfect hashing, universal hashing, cuckoo hashing, bloom filters, and locality-sensitive hashing. The chapter also covers related topics like bipartite graph analysis in cuckoo hashing and probabilities of false positives in bloom filters.

2.1 Basic Hashing

Definition 2.1.1 Hashing is a structure for distributing data entries in an array of buckets using an onto function f which ranges over the indices of arrays of buckets also known as hash tables.

Given a set of keys, a hashing algorithm computes indices that determine where the entry will be stored and can be found later. Generally, this is done in two steps: hash = hashfunc(key) followed by index = hash % array-size (or we can say index = $f(key, array_size))$. In this conventioonal practice, the hash value and array size are independent, and are then glued using the modulo operator (%) to fit in the hash table.

Often, array size is in the form 2^k and the remainder operation is simplified to masking, which speeds the algorithm but increases collision probability.

2.1.1 Hash function

Hash function is the most essential component of a hashing data structure (Figure 2.1). A good hash function utilizes hashing for fast look-ups but is difficult to create. The fundamental requirement is that a hash function should uniformly distribute values throughout a hash table. A non-uniform distribution increases the possibilities of collisions and increases the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated theoretically using statistical evaluations, like estimation.

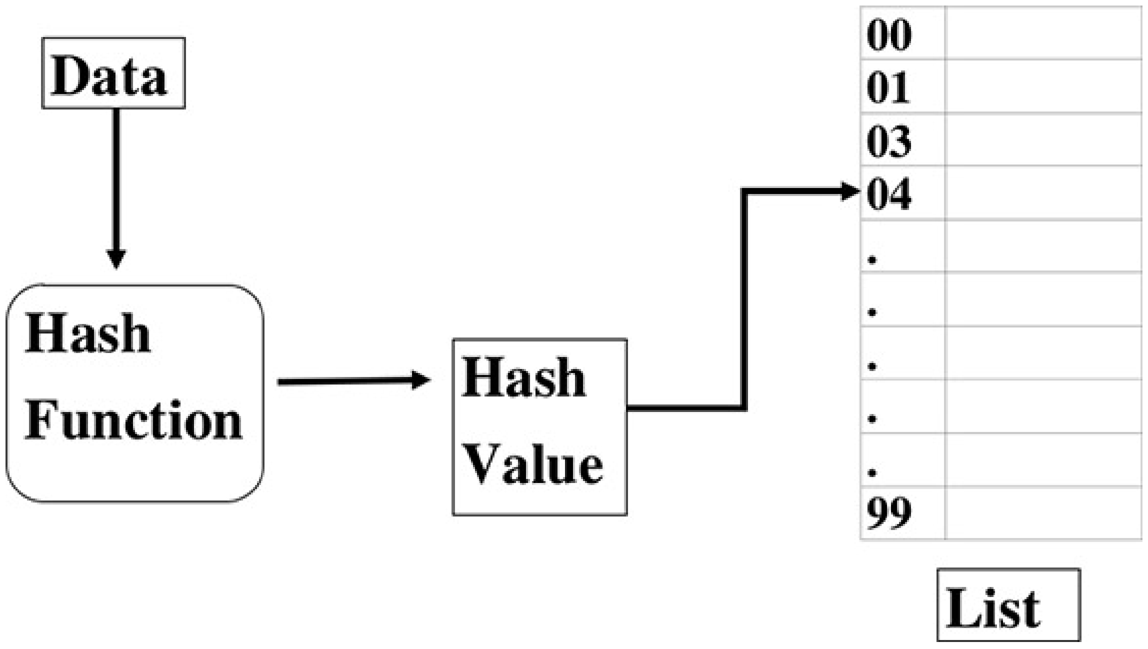


Figure 2.1: Pictorial representation of a simple hashing

To get a clear understanding, consider the example of dynamic resizing by means of exact doubling and halving of an n size table. The hash function needs to be uniform only when n is a power of two where the index can be computed as some range of bits of the hash function. However, the preferable hashing algorithms choose n as a prime number. The modulus operation provides some additional permutation and is very useful to improve poor hash functions.

Let us consider another example of very useful cryptographic hash function, the well known function for any table size n. These functions are generally created in combination of modulo operations and bit maskings. They are also expected to be appropriate under attack. Suppose a malicious user tries to access a network service by means of submitting predefined requests carefully designed to create a large number of collisions in the hash table. Choosing a random hash function from the universal family is a common practice to avoid these kinds of attacks. However, creating complex hash functions of computationally expensive and simpler hash functions are preferable.

2.1.2 Load factor

A critical statistic for a hash table is the load factor, defined as: $\alpha = \frac{n}{k}$; where, n is the number of elements and k is the size of hash table. As the load factor grows larger, the hash table becomes slower, and it may even fail to work (due to "birthday paradox"). To maintain the expected constant time lookup criteria in a hash table, the load factor is generally kept below some predefined bound to minimize the collision probability as computed by the birthday paradox. Given a fixed number of buckets, the time taken for a lookup operation grows proportionally with the number of entries and therefore the desired constant time constrained for search operation is not achieved. Moreover, the number of entries per bucket should not vary too much. For example, assume two tables both contain the same number of entries and the same number of buckets. The first table contains one entry in each bucket. All entries are mapped in the same bucket in the second table. Hashing will not work in the second table. A low load factor is not beneficial. As the load factor approaches zero, unused entries in the hash table increase; the cost of searching does not decrease and the result is a waste of memory.

2.1.3 Collision resolution

Let us assume that there are n available slots and m occupying items in a hashing scheme, then the following probabilities can be computed:

Probability of no collision after 1st insertion = n/n,

Probability of no collision after 2^{nd} insertion = $(n - 1)/n$,
Probability of no collision after 3^{rd} insertion = $(n - 2)/n$, and so on,
....
Probability of no collision after m^{th} insertion = $(n - (m - 1))/n$.
Therefore, the probability of no collision after m insertions in a hash table can be computed as a product of the above values, i.e.,

$$(n-1)!/((n-m)! * n^{(m-1)}).$$

In other words, the likelihood of a single collision is

$$1-(n-1)!/((n-m)! * n^{(m-1)}).$$

Therefore, the collisions in the hashing are unavoidable in reality, particularly when hashing a random subset of a large set. As a consequence, the collision resolution strategies are very common in most hashing techniques. All these methods require a pointer stored with the keys to denote the alternative arrangement. In the method known as separate chaining, each bucket of the hash table is independent and a list of entries with same hash value is maintained in the same index. The total time taken for a lookup operation in that hash table is the sum of time taken to find the bucket, i.e., constant time and the time taken for searching in the list maintained in that bucket, which may be linear in the worst case. In a good hash table, each bucket is expected to have one or no entries, and sometimes a very small number of elements, but not more than that. Therefore, a preferable structure for a hash table should be efficient with respect to time and space, not just large entries for each bucket. If these collisions happen frequently in some hash function we need to fix the function, not fix the table. Open addressing are from clustering, which may increase the lookup cost drastically, even if we choose to keep the load factor low. This phenomenon forces us to adopt multiplicative hash to avoid clustering.

2.2 Perfect Hashing

Perfect hash functions are used to implement a hash table with constant time access in worst cases. This special type of hash function is very useful for fast searching and does not require any collision resolution technique. A perfect hash function, whose values lies in a bounded range is used for efficient lookup, by mapping keys from S, the set of all elements, in a hash table indexed by the output of the hash function. Each lookup operation takes constant time in the worst case for perfect hashing.

Definition 2.2.1 — Perfect hashing. A hash function is called perfect for a set S of n elements that maps distinct elements in S to a hash table without any collisions. Mathematically, a perfect hash function must be an injective function.

2.2.1 Construction

A perfect hash function for a specific that can search elements in constant time and has values in a bounded range is generally created by a randomized algorithm. The original construction of Fredman et al. [15] uses a two-level scheme to map a set S of n elements to a range of $O(n)$ indices, and then maps each index to fit in the scope of hash table. The first level construction chooses a large prime p (larger than the size of the domain of S), and a coefficient k to maps each element x of S to the index $f(x) = ((kx\%p)\%n)$ Though k is chosen randomly, there are possibilities of collisions, but a positive factor is that the number of elements n_i which collide at i-th index is far fewer. In the second level construction a disjoint range of $O(n_i^2)$ integers is assigned to each index i. It uses a different set of linear modular functions for each different i-th index. Thus each member x of S map into the range associated with $f(x)$. As Fredman et al. [15] show, one option is to choose the parameter k such that the sum of the lengths of the ranges is $O(n)$ for the n different values of $f(x)$. In addition to that, for each value of $f(x)$, a linear modular function exists that maps the respective subset of S into the associated range of that value. Determination of k and the second-level functions is done in polynomial time for each value of $f(x)$ by random selection and manual verification. The space requirement of hash function is $O(n)$ including everything. A modified version of this two-level scheme with a larger number of values at the top level can be used to construct a perfect hash function that maps S into a smaller range of length $n + o(n)$.

> **Theorem 2.2.1** For any hash function f, chosen randomly from a universal family, which maps n keys to a table of size $m = n^2$, the probability of collisions is less than 1/2.

Proof. (Sketch) Expected number of collision is $\binom{n}{2\times\frac{1}{n^2}\leq\frac{1}{2}}$. ▪

2.2.2 Remarks

- A widely used perfect hashing with dynamic updates in Section 2.4 covering cuckoo hashing. This scheme maps keys to two or more locations within a range and provides constant time lookup.

- Lookups with this scheme are slower due to verification of multiple locations, but still perform in constant worst-case time.

2.3 Universal Hashing

Universal hashing is a way of choosing a hash function randomly from a special family of hash functions, where each member of the family satisfies certain mathematical properties (See Figure 2.2). The universal hashing guarantees a low number of expected collisions, even if the data is chosen by an unknown user. There are many universal families of hash functions that are well known for their efficient evaluation.

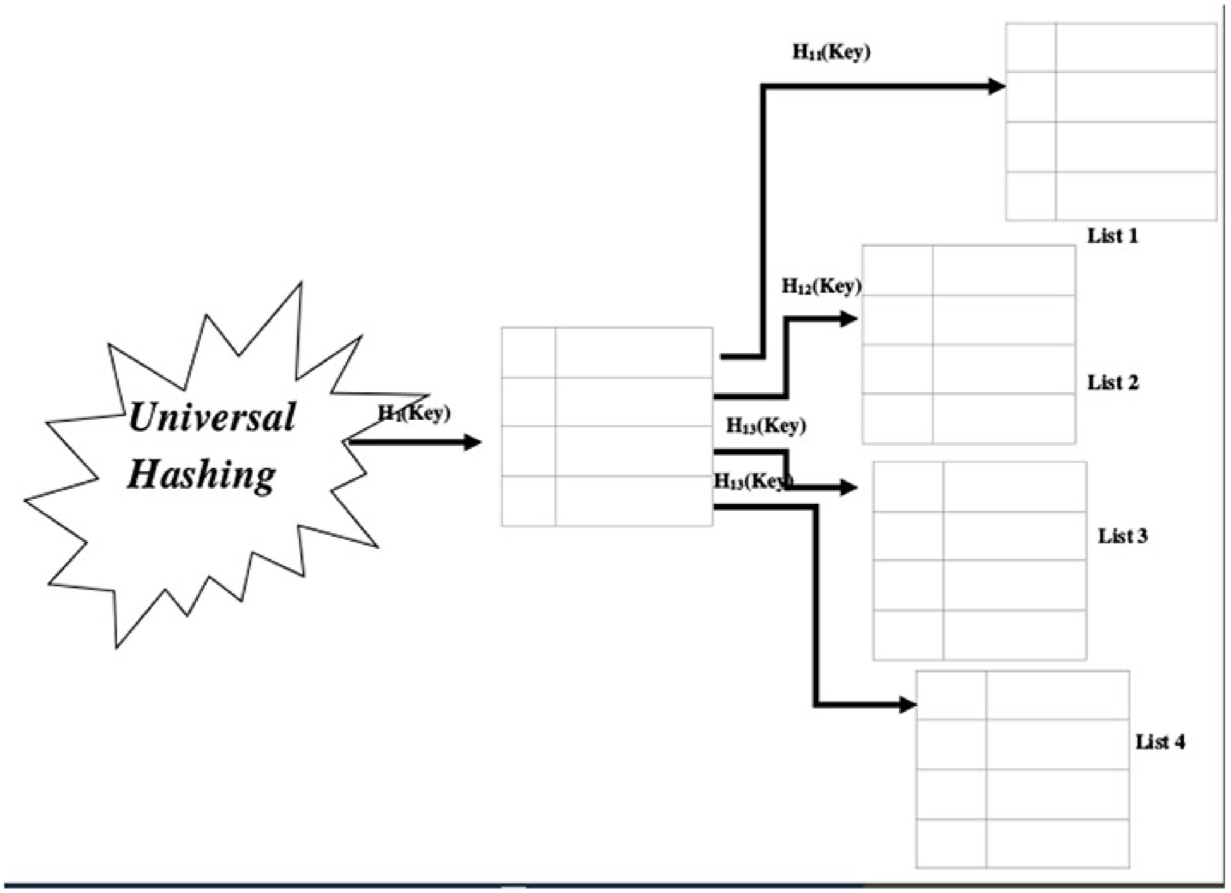


Figure 2.2: Universal hashing

Definition 2.3.1 — Universal hashing. A finite collection of hash functions $\mathscr{H}$, where each member $f \in \mathscr{H}$ maps the elements of a given universe U to a hash table of range $\{0, 1, ..., m-1\}$ is said to be universal if for every pair of elements $x, y \in U$, where $x \neq y$, the probability of collision is bounded by $|\mathscr{H}|/m$.

Universal hashing is used in many areas of computer science, for example in implementing hash tables to create randomized algorithms and to develop cryptographic schemes. Suppose a malicious user wants to access a network service by submitting predefined requests designed to create large numbers of collisions in a hash table. Choosing a random hash function from the universal family is a common practice to avoid these kind of attacks, by ensuring the that collision patterns are unpredictable. Universal hashing usually performs well no matter what keys are chosen by the adversary.

2.3.1 Important properties

There are different kind of universal hash families satisfying different sets of properties and providing different bounds on collision probability. Some of the most important properties are given in this subsection.

Definition 2.3.2 — Almost universality. If we choose a hash function randomly from universal family of hash functions, i.e. $h \in H$, then probability that two keys will collide is at most $O(1/m)$, where m is the number of buckets in the hash table.

Definition 2.3.3 — Uniform difference property. For randomly chosen $h \in H$ we have $\forall x, y \in U$, $x \neq y$, the difference $h(x) - h(y) \mod m$ is uniformly distributed in $[m]$.

Note: The universality property appears when $h(x) - h(y) = 0$, i.e. in case of collisions, but the uniform difference property is more general and satisfied in limited universal families.

Definition 2.3.4 — XOR universal property. A universal family can be XOR universal if $\forall x, y \in U$, $x \neq y$, the value $h(x) \oplus h(y) \mod m$ is uniformly distributed in $[m]$ where $\oplus$ is the bitwise exclusive OR operation.

Note: This is only possible if m is a power of two.

Definition 2.3.5 — Pairwise independence property. Pairwise independence is a strong condition that occurs when $P(h(x) = z_1 \wedge h(y) = z_2) = 1/m^2$ where z_1, z_2 are the hash values of x, y satisfying$\forall x, y \in U$, $x \neq y$, the probability of perfectly random.

Note: Pairwise independence is also known as strong universality.

Definition 2.3.6 — Uniformity property. A family of hash functions H is uniform if all hash values are equally likely, i.e. $P(h(x) = z) = 1/m$ for any hash value z of the point x. We can observe that strong universality may sometime (but not always) imply uniformity.

Note: Commonly used tricks such as adding a random constant to hash values of using a subfamily of a universal hash family may help us to achieve a desired property. These properties are very important for some specific application to meet expected collision bounds.

2.3.2 Mathematical guarantees

Theorem 2.3.1 Let $f \in H$ be chosen randomly, where f maps the the n elements of U into a table of size $m \geq n$, then for any $x \in U$ the probability of collision involving x is <1.

Proof. Let, $y, z \in U$ and $y \neq z$.

Let, R_{yz} be the random variable defined as: $R_{yz} = 1$ when $f(y) = f(z)$ and $R_{yz} = 0$ when $f(y) \neq f(z)$.

By property of universal hashing we have $E[R_{yz}] = 1/m$.

Let, R_x be the total number of collisions involving the element x.

Then, $E[R_x] = \sum_{y \in U, y \neq x} E[R_{xy}] = \frac{n-1}{m}$.

Since $n \leq m$, we have $E[R_x] < 1$. ▪

Note: A randomized hash function is usually created to handle at most $O(n)$ collisions. If too many collisions are observed then another random hash function is chosen from the family where universality guarantees the geometric randomness.

2.4 Cuckoo Hashing

Cuckoo hashing, introduced by Pagh and Rodler [16], replicates the activities of cuckoo birds that lay their eggs in crow nests after pushing out the crows' eggs.

> **Definition 2.4.1 — Cuckoo hashing.** This hashing scheme solves the dynamic dictionary problem using two hash tables and a carefully designed rule of eviction to achieve $O(1)$ worst-case time for lookup and deletes, whereas, $O(1)$ is the expected amortized cost for insertion mechanism.

Let f and g be two random hash functions f, g: $[u] \rightarrow [m]$ where $m = c \cdot n$, and an array $T[1 \ldots m]$ that stores items. The structure will maintain the invariant that the entry x will always be stored in either $T[f(x)]$ or $T[g(x)]$, so that queries will take constant time. Generally, f and g map to a table T with m rows. But here in cuckoo hashing, f and g hash to two separate hash tables, $T[f(x)]$ and $T[g(x)]$ respectively. The cuckoo part refers to movement of keys from one table to another following the alternative hash function in the event of collision, until the collision is resolved.

2.4.1 Operations

Cuckoo hashing. Like other hashing techniques, performs the basic operations of insertion, searching, and deletion. In a search operation, insertion, search and delete. In search operation, we need to check two positions in the hash tables, $T[f(q)]$ and $T[g(q)]$ for query element q 2. Delete operations remove elements after search results return true 3. To insert an element x, we need to check whether $T[f(x)]$ (index in first table) is empty or not. If entry in first table $T[f(x)]$ is empty, we put x in $T[f(x)]$ and are done. Otherwise, we need to evict some element, say y, from $T[f(x)]$ followed by occupancy of $T[f(x)]$ by x and occupancy of $T[g(y)]$ by y, provided $T[g(y)]$ is empty. In case $T[g(y)]$ is already occupied by some element, say z, we have to evict z before inserting y in the second table. The evicted element z will follow the mechanism of insertion similar to x from the beginning. We continue this procedure until we're done or forcefully stopped after a certain iteration occurs (generally *log* n multiplied by a predefined constant, possibly 4 or 6) and opts for a rehash.

Each element x must be at $T[f(x)]$ or $T[g(x)]$ and maintained to ensure the correctness of search and delete, as well as guarantee the $O(1)$

Note search and delete operation. However, the complexity analysis of insertion is difficult.

Insert(x) is not horribly slow because only a small number of items are bumped and we can rehash a table whenever m (may be $4n$) is large enough.

2.4.2 Bipartite graph of cuckoo hashing

We can always construct a bipartite or cuckoo graph, associated to every cuckoo hashing $2m$ vertices, realized by the locations in the array $T[f(x)]$, and $T[g(x)]$, where edges are characterized by the displacement of the elements in S through hash tables. To understand the insertion process of cuckoo hashing visually, we generally construct a cuckoo graph, which is a bipartite graph with m vertices namely, $1, 2, \ldots, m$ and edges are realized as ($f(x)$, $g(x)$) for all $x \in S$. Now, inserting a new element in cuckoo hashing corresponds to a walk in cuckoo graph.

Algorithm 1 Insertion algorithm of cuckoo hashing

```
1:  procedure CUCKOO-INSERT($x$)                    ▷ take an input element $x$
2:      $c = 0$                                       ▷ set counter to 0
3:      while $c \neq dlog(n)$ do      ▷ hard stop after $dlog(n)$ displacement, d
    predefined constant
4:          if $T[f(x)]$ is empty then index $x$ in $T[f(x)]$ return true  ▷ trivial
    case
5:          else y= $T[f(x)]$                          ▷ evict $y$, old element
6:              c=c+1                                  ▷ increment counter
7:              index $x$ in $T[f(x)]$
8:              if T[g(y)] is empty then index $y$ in $T[g(y)]$ return true ▷ put
    $y$ in second table
9:              else z= $T[g(x)]$       ▷ evict $z$ old element of $y$'s second table
10:                 c=c+1                              ▷ increment counter
11:                 index $y$ in $T[g(y)]$
12:                 insert(z,d)                        ▷ repeat procedure for z
13:     return false
```

Algorithm 2 Search algorithm of cuckoo hashing

```
1: procedure CUCKOO-SEARCH($q$)                              ▷ take a query element $q$
2:     if $T[f(q)]$ is non empty then return true   ▷ query element found in first table
3:     else if $T[g(q)]$ is non empty then return true ▷ query element found in second table
4:     else return false                                      ▷ query element not found
```

Algorithm 3 Delete algorithm of cuckoo hashing

```
1: procedure CUCKOO-DELETE($z$)                               ▷ element $z$ to be deleted
2:     if $T[f(z)]$ is non empty then delete $T[f(z)]$  ▷ deleted from first table
3:     else if $T[g(q)]$ is non empty then delete $T[g(z)]$     ▷ deleted from second table
4:     else nothing to delete                                   ▷ element not found
```

Theorem 2.4.1 Insertion in cuckoo hashing is successful only if the associated graph contains one cycle or no cycles.

Proof. If a cuckoo graph contains no cycle, no loop or rehash can occur in the insertion process. Let the single insertion cycle in cuckoo graph be $x_1, x_2, \ldots, x_k$. Then x_1 will evict x_2 and finally x_k will evict x_1. Both these evictions will take place at the same place, but the elements of evictions are different. The first element was x_2, and the final one was x_1; they go to different places and do not cycle back to their starting locations. Otherwise, a graph has two or more cycles. ▪

Definition 2.4.2 — Complex. A connected component of a graph is complex if it is neither a tree nor unicyclic.

Theorem 2.4.2 — Constant expected amortized insertion, random hash functions. Pr[Insert follows eviction path of length k] $\leq 1/2^k$.

Proof. (Sketch) For two hash functions f and g, where each has m values, and each of these m values has $\log m$ bits, we need $2n\log n$ bits to encode f and g.

■

Theorem 2.4.3 — Constant expected amortized insertion, rare existence of complex. Pr[a bipartite multigraph contains a complex]$\theta 1/m$.

Theorem 2.4.4 — Constant expected amortized insertion. The expected amortized cost of an insertion procedure in a cuckoo hashing is $O(1+\varepsilon+\varepsilon^{-1})$.

Note Detailed systematic analysis of cuckoo hashing and the proof of et al [18,17]

2.5 Bloom Filters

Definition 2.5.1 A bloom filter is a compact data structure designed for probabilistic representation of a set; it answers membership queries (i.e. whether a particular element belong to the set or not). The compact representation is subject to false positive penalties in membership queries.

Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is made sufficiently low. Burton Bloom introduced Bloom filters in the 1970s, and they have become very popular in database applications. A bloom filter is not just a data structure developed to support membership queries and its wide use is the result of its interesting characteristics. Its main properties are given below:

- The amount of space needed to store the bloom filter is small compared to the amount of data belonging to the set being tested.
- The time needed to check whether an element is a member of a given set is independent of the number of elements contained in the set.
- False negatives are not possible.

- False positives are possible, but their frequency can be controlled. In practice, a trade-off exists between space and time efficiency and false positive frequency.

2.5.1 Construction of bloom filter

A bloom filter is based on an array of m bits $(b_1, b_2, \ldots, b_m)$ that are initially set to zero (See Figure 2.4). To understand how a bloom filter works, it is essential to describe how these bits are set and checked. For this purpose, k independent hash functions $(h_1, h_2, \ldots, h_k)$, each returning a value between 1 and m, are used. In order to "store" a given element into the bit array, each hash function must be applied to it and, based on the return value r of each function $(r_1, r_2, \ldots, r_k)$, the bit with the offset is set to one. Since there are k hash functions, up to k bits in the bit array are set to one (it might be less because several hash functions might return the same value).

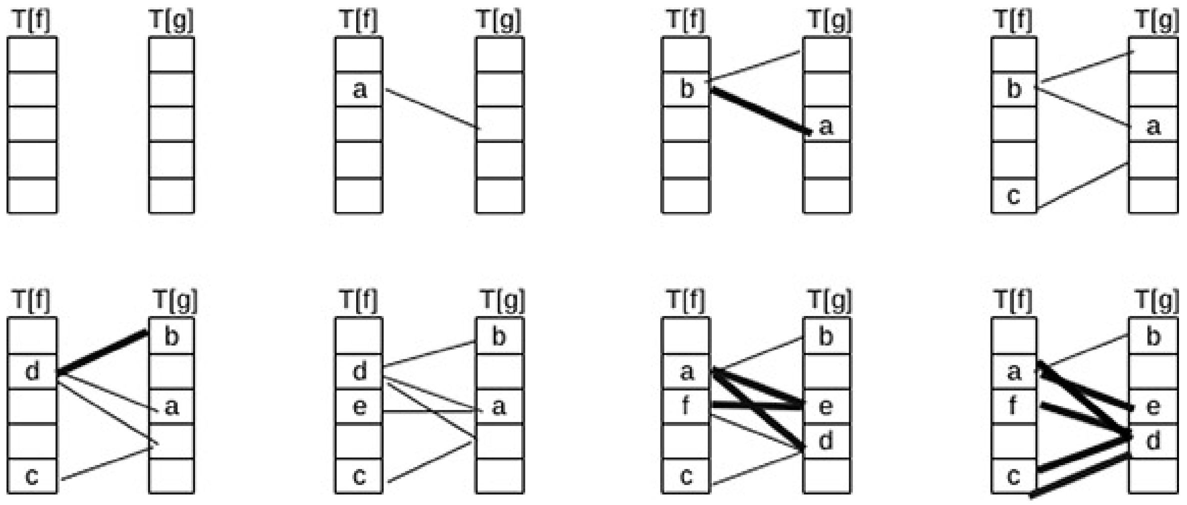


Figure 2.3: Pictorial representation of insertion in cuckoo hashing and associated bipartite graph

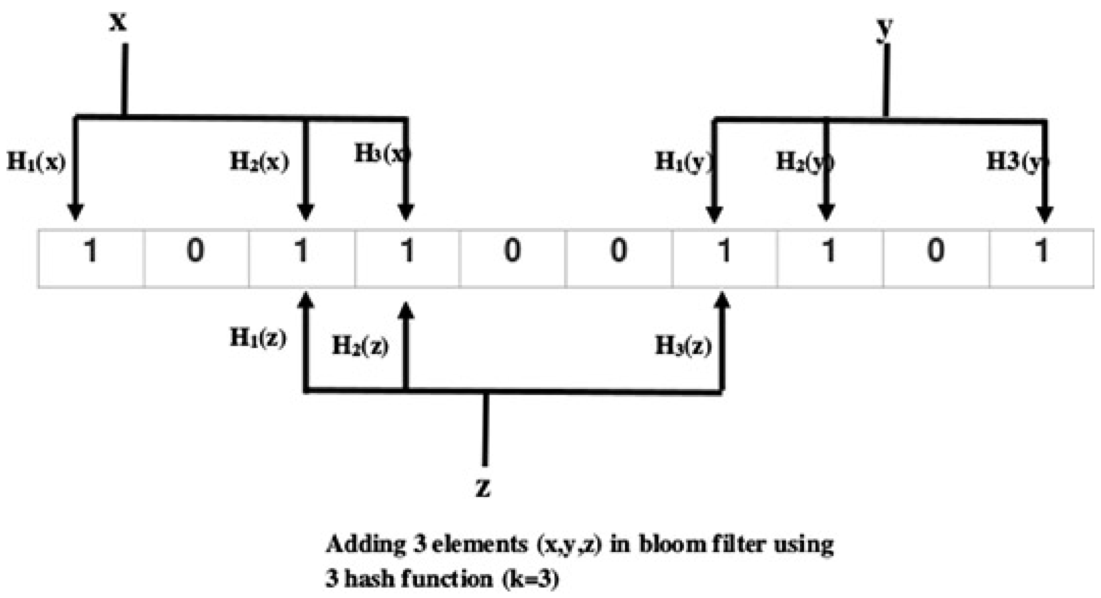


Figure 2.4: Pictorial representation of a bloom filter

The following procedure builds an *m* bits bloom filter *F*, corresponding to a set *S* of *n* elements, which describe membership information using *k* independent hash functions $(h_1, h_2, \ldots, h_k)$:

Algorithm 4 Algorithm to create an *m* bit bloom filter

```
1: procedure CREATEBLOOMFILTER(set $S$, hash-functions, $m$) ▷ take input set, hashings and $m$
2:     initialize $m$ bit filter $F$ to 0                         ▷ set $F$ to 0
3:     for all $e_i \in S$ do
4:         for all hash function $h_j$ do $F[h_j(e_i)] = 1$ ▷ set bit 1 for elements by all hashings
5:     return $F$
```

Therefore, if *e* is member of a set *S*, in the resulting bloom filter *F* all bits obtained corresponding to the hashed values of *e* are set to 1. Testing for membership of a query element e_q is equivalent to testing that all *k* bits corresponding to each $h_i(e_q)$ are set to 1 or not, in case of 0 at any one position; the testing procedure safely return false.

Algorithm 5 Algorithm for membership test in a bloom filter

```
1: procedure MembershipTestBloomFilter(filter $F$, hash-functions, query element $e_q$)
2:     for all hash function $h_j$ do
3:         if $F[h_j(e_q)] \neq 1$ then return False        ▷ return false and exit
4:     return True
```

Note A bloom filter can be built incrementally with the arrival of new elements to a set and the corresponding positions are computed through the hash functions and bits are set in the filter. Moreover, the union of two sets is simply computed as the bit-wise OR applied over the corresponding filters.

2.5.2 Probability of false positives

Lets assume that a hash function follows the uniform distribution over the filter, i.e. each bucket is equally probable. For $m = |F|$ and $h_i \in \{h_1, h_2, \ldots, h_k\}$ the probability of false positive is computed as follows:

A certain bit is set to 1 by a hash function during insertion and is then $(\frac{1}{m})$.

Therefore, a certain bit not set to 1 is a hash function during its insertion is $(1 - \frac{1}{m})$.

The probability that it is not set to 1 by any of the hash functions is $\left(1 - \frac{1}{m}\right)^k$.

If n elements are inserted, the probability that a certain bit is still 0 is $\left(1 - \frac{1}{m}\right)^{kn}$.

The probability that certain bit set to 1 is therefore $1 - \left(1 - \frac{1}{m}\right)^{kn}$.

The probability of all positions are 1 in array corresponding to a non member element is $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$.

Now, probability TestMembership(S, e_q) for a non member e_q is false, i.e. each of the k array positions is set to 1 due to other element is $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$. This is not an exact result because it assumes independence for the probabilities of each bit being set. However, this result is a close approximation and the probability of false positives decreases as m increases, and increases as n (the number of inserted elements) increases.

2.5.3 Optimal values of parameters

For a given *m* and *n*, the value of *k* (the number of hash functions) that minimizes the probability can be computed from the equation $p = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}$, where p is the probability of false positive. The formula can be approximated to $p = \left(1-e^{-kn/m}\right)^{k}$, using Starling's approximation.

Solving for optimal k we obtain: $k_{min} = \frac{mlog(2)}{n}$, i.e. number of required hash functions.

With optimal value of *k*, the equation of false positive can be written as: $p = 2^{\frac{-mlog(2)}{n}}$

Solving for optimal m we obtain: $m_{min} = \frac{nlog(1/p)}{log^2(2)}$, i.e. requirement of filter size.

Bits per element $b = m/n = \frac{log(1/p)}{log^2(2)}$.

Bloom filters are effective for representing sets with respect to space use. They present small risk of false positives because they do not store data. A Bloom filter with 1% error and an optimal value of *k*, determine from the set size and filter size, requires only about 10 bits per element, whatever the size of the elements may be. This advantage is achieved from its compact representation and probabilistic nature. If we want to reduce the false-positive rate to 0.01% we need to increase the bits per element into 17, which can be derived from the equation easily. Bloom filters are very efficient in terms of complexity of creation, *O*(*nk*), and complexity of look-up, *O*(*k*). Since *k* is a fixed constant, we can observe the creation process as linear and look-up as constant.

▪ **Example 2.1 — Database query verification.** A typical application receives lots of input data. It is verified before processing, whether that data is present in a given set, which is stored in a database table. Note that this validation process has very high rejection rate and the validation is performed remotely. But, the time required for the two way network access is very high and not possible to optimize without increasing the bandwidth. Using a cache is A trivial non feasible solution that come to mind. The general idea is to clone the data of remote server into a local cache and perform the validation locally but that may have a resource constraint. However, a bloom filter is very handy to solve the problem within the constraints of memory and network bandwdith. What about false positive? A bloom filter is intended to discard major amounts of input data before transmittal for remote verification. As a result, validation became faster and verifications became less necessary.

2.6 Locality-Sensitive Hashing

Locality sensitive hashing (LSH) is a data structure to perform probabilistic dimension reduction of high-dimensional data and efficiently handle similarity searches, thus defeating the "curse of dimensionality." The basic idea is to hash the input items so that similar items are mapped to the same buckets with high probability (the number of buckets being much smaller than the universe of possible input items).

Definition 2.6.1 — Locality sensitive hashing. Let h be a random hash function defined over a set S of n points. The hash function h is called locality-sensitive, if, for any pair of points $p, q \in S$ we have:
$Pr[h(p) = h(q)]$ is "high" if p is "close" to q
and $Pr[h(p) = h(q)]$ is "low" if p is "far" from q.

? Do such functions exist?

To understand locality-sensitive hashing, lets take an example.

▪ **Example 2.2 — LSH using hamming distance.** Consider the d dimensional hypercube $\{0, 1\}^d$ and hamming distance $D(p, q)$ = # positions on which p, q, h and R differ, where $p, q \in \{0, 1\}^d$. Define hash function h by choosing a set R of $k \leq d$ random coordinates, and setting $h(p)$ = projection of p on R. In a particular scenario say, d=10, p=0110010111, q=1100110011 and R is projection of points corresponding to the first, third, and fifth coordinates. Then $h(p)$ = 010 and $h(q)$ = 101. According to the definition of h: probability of equality of hash values of p and q is $Pr[h(p) = h(q)] = (1 - D(p, q)/d)^k$, i.e. $=(1 - 3/10)^3 = (7/10)^3$, which is quite high, so we should keep p and q in different buckets.

▪ **Example 2.3 — LSH for for high dimensional data.** Consider another example of real high dimensional data. We use l_p^d to denote the d-dimensional real space $\mathbb{R}^d$ under the l_p norm. For any point $\mathbf{v} \in \mathbb{R}^d$ the notation $||v||_p$ represents the l_p norm of the vector $\mathbf{v}$. The LSH scheme uses p-stable distributions as follows:

> Compute the dot products $(a.v)$ to assign a hash value to each vector $\mathbf{v}$.
> Each hash function defined as, $h_{a,b}(\mathbf{v}) : \mathbb{R}^d \rightarrow h_{j,i}^r(\mathbf{x}) = \frac{\langle \mathbf{x} | \mathbf{a}_{j,i} \rangle - b_{j,i}}{w}$.
> The hash function $h_{j,i}^r$ produces a real value which is subsequently rounded to an integer, as $h_i(\mathbf{x}) = \left\lfloor h_{j,i}^r(\mathbf{x}) \right\rfloor$.
> A single hash function of H being not discriminative enough by itself, a second level of hash functions is defined by concatenating functions $h_i \in H$.
> $\mathbf{g}_j^r(\mathbf{x}) = \left(h_{j,1}^r, \cdots, h_{j,k}^r\right)$ and after quantization:
> $\mathbf{g}_j = \left\lfloor \mathbf{g}_j^r \right\rfloor = (h_{j,1}, \cdots, h_{j,k})$.

At this point, a vector x of the dataset is indexed by a set of L vectors of integers $\mathbf{g}_j(\mathbf{x}) \in \mathbb{Z}^d$.

2.6.1 Use in nearest neighbor search problem

One of the main applications of LSH is to provide efficient nearest neighbor search algorithms. In practice we use locality sensitive hashing to report (R, ε)-approximate nearest neighbor queries, which returns a point within distance εR to the nearest neighbor of the query point. The goal of LSH based nearest neighbor search is to use hash functions $h_j(x) : \mathscr{R}^d \rightarrow \mathscr{N}^k$ with the locality-sensitive property i.e., probability of collision is higher for nearby points than for distant points. At the time of mapping, we populate the hash table by evaluating each hash function on collection of points. To perform a nearest neighbor query, we again evaluate the hash functions on the query point and retrieve the points in the same bucket of the query. One of these points is likely to be a ε-nearest neighbor of the query.

The above technique may suffer from edge effects, i.e., a query point may map in a bin other than the bin containing its nearest neighbor due to sensitivity threshold which can be tackled by increasing the number of hash functions that reduce the probability of edge effect.

Note Nearest neighbor search is very useful for real-time applications such as finding a school near your new home or finding a nearby bus stop or restaurant, and LSH is very efficient for such tasks in comparison to other tree based data structures.

2.7 Exercises

Exercise 2.1 Consider the universal family $H = \{(ax + b)\ modNmodm\}$ where $a, b \in \{0,..., N - 1\}$, $N = 97$ and $m = 7$. Assume we choose two hash functions $f, g \in H$ and the sequence of insertions: I(3), I(10), I(17), I(24), I(31), I(38), I(45). Find the collisions for your choice. ▪

Exercise 2.2 Consider objects of the form xy where x and y are k bit integers. Let $h(xy) = f(x, y)$ be a deterministic hash function. Show that there is a huge number of collisions. ▪

Exercise 2.3 Find the probability an ideal random hash function is perfect. ▪

Exercise 2.4 Prove or disprove:

a) Cuckoo hashing has a worst-case unbounded time complexity with regards to the insert operation.

b) Cuckoo hashing is able to rehash in constant time, which is why it has an average case constant time complexity. ▪

Exercise 2.5 Is it possible to delete an element from a bloom filter? ▪

Exercise 2.6 Explain how to determine the optimal size of a bloom filter (assume that other parameters are fixed). ▪

Exercise 2.7 Assume we have a corpus of 1000 documents. In order to check for similarities among them, we must compare 1000 × 1000 pairs. Justify the role of locality sensitive hashing towards reduction of this huge computational task. ▪

Chapter 3

O(log(n)) Ordered Search (Trees and Lists)

In this chapter we discuss well known parts of data structures, i.e., ordered searches for point queries based on trees and lists. However, more emphasis is given to select topics that have been widely used in the past and cover important theoretical foundations. In first section of this chapter, balanced BST is covered as a brief review of previously covered topics of basic data structure course. In section 2 randomized BST is discussed followed be dynamic optimality in the next section. The issues of reducing log(n) lower bound is described in section 4 and section 5 named as splay tree and tango tree respectively. Skiplist is presented in the last section, as a state of the art representative of linked based data structures, which provides O(log n) time search with a probabilistic guarantee.

3.1 Balanced Binary Search Trees (BSTs)

A self-balancing binary search tree is a data structure for solving point query searches. The tree maintains its height from foots to leaves and supports dynamic updates. A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1. A binary tree is balanced if for any two leaves the difference of the depth is at most 1. A binary search tree has the following properties. Each node in the tree stores an element and can have at most two child nodes. The tree does not contain any duplicate values. Elements in a node's left subtree are smaller than the node's element. Elements in a node's right subtree are greater than the node's element. There are also three main tasks a binary search tree can perform. Figure 3.1 depicts a BST. Figure 3.2 depicts a balanced BST.

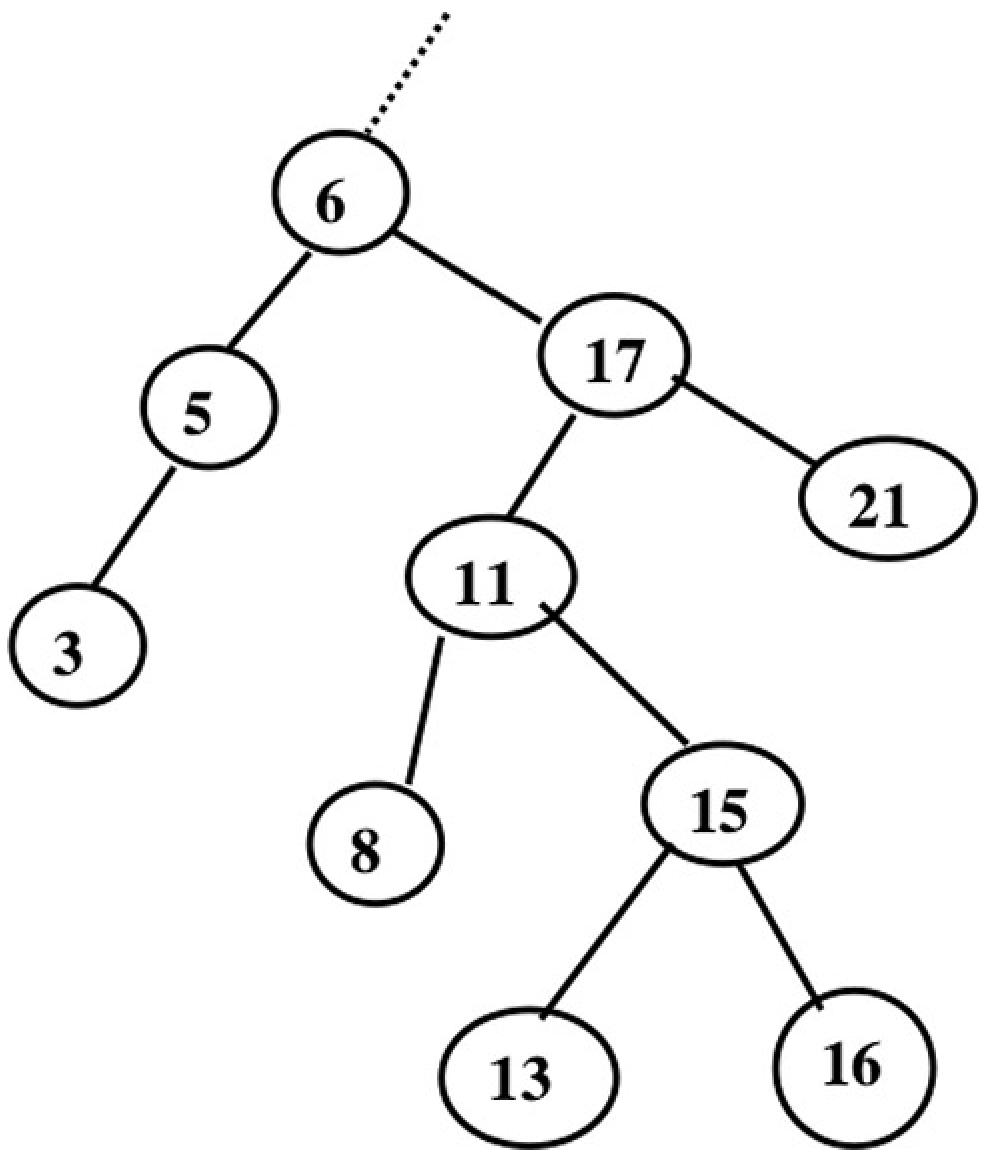


Figure 3.1: A binary search tree

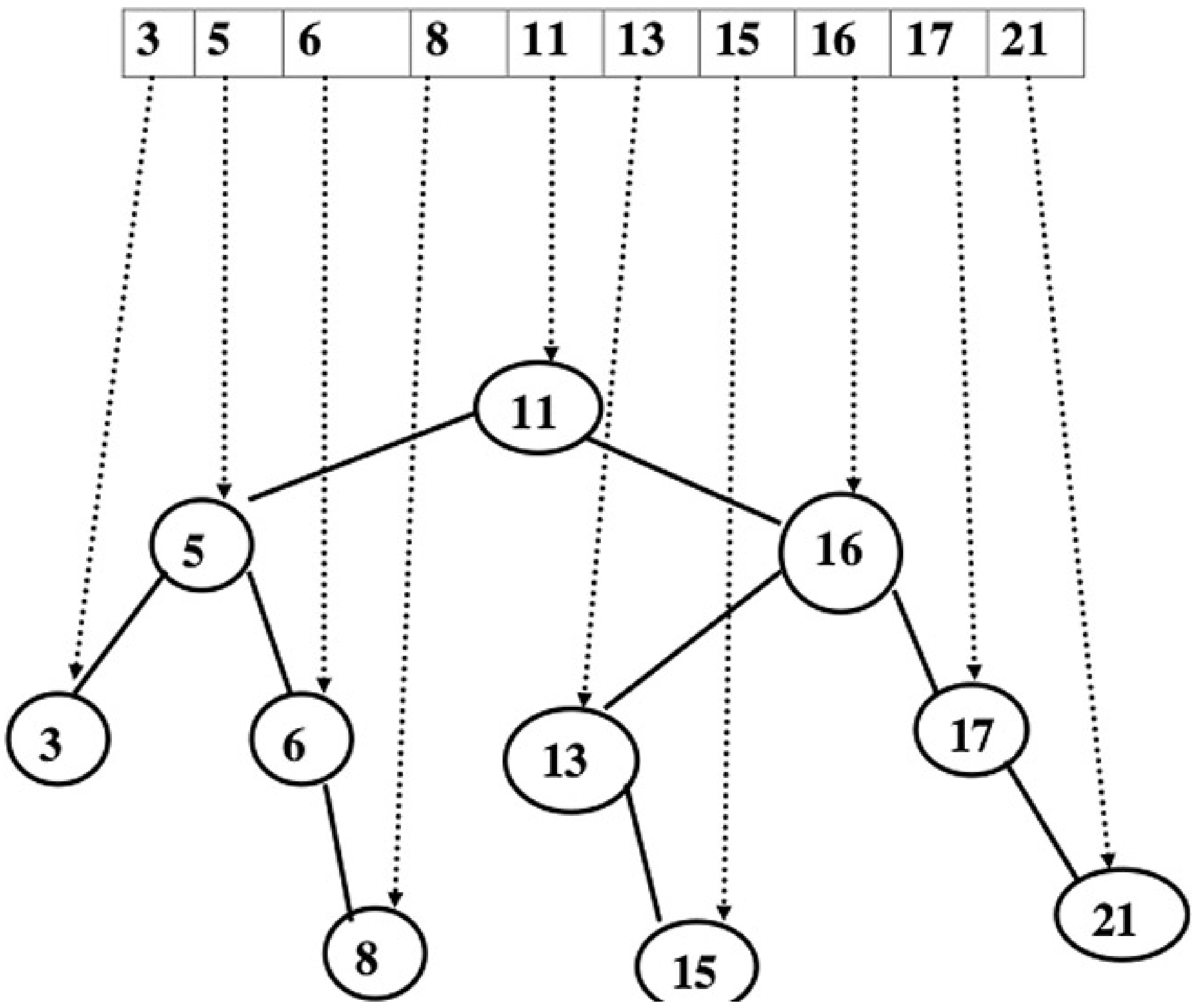


Figure 3.2: A balanced binary search tree for ordered search

Summary 3.1 — Search. Determine if a node containing a particular element exists in the tree by performing a binary tree search. This is done by starting at the root node and recursively searching the tree by selecting the left or right subtree based on the node's value and the value that we are searching for. We know that the value does not exist in the tree if we reach an external node and have not yet found it.

Summary 3.2 — Insert. Insert a new node in the correct place in the tree given its value. This is also done with a tree search. Assuming the element does not already exist in the tree, we search for the value that we wish to insert until we arrive at an external node at which point we add the new node as its right or left child depending on its value.

Summary 3.3 — Delete. Remove an element from the tree and rearrange the remaining nodes in order to keep the desired structure. If the node you wish to delete is an external node you simply remove it; however if it has children, the process is more complicated. One way to deal with this is to identify its predecessor in the left subtree. This is the greatest element in the left subtree which can be found by recursively selecting the right child within this subtree (in other words it is the right-most element in the left subtree). This in-order predecessor is then removed (it is an external node so this is simple) and is used to replace the node that is to be deleted. The opposite procedure would work as well (i.e. replacing the node with its in-order successor in the right subtree).

3.1.1 Height bound of balanced BST

Let T be a balanced binary search tree with n nodes whose maximum height is h. At the top level there is only one node, in the next level there are at most 2 nodes, followed by 4, 8, … up-to 2^h in the last level. Therefore a binary search tree with with height h contains at most $2^0 + 2^1 + \ldots. + 2^h = 2^{h+1} - 1$ nodes, which implies that $h = O(\log n)$ when a tree is balanced. The standard operations defined on a binary search tree (insert, delete, and search) takes at most the height of the tree, i.e., $O(\log n)$ time, but we need to ensure the tree is balanced. To keep this desired height of the balanced BST many strategies are adopted from time to time and most methods incur additional overhead cost for storing additional information such as weights and heights of subtrees, and red and black color conventions. Additionally, such operations also require complex balancing techniques to update the system after updates.

Note Most past research was intended to find efficient ways to balance BSTs after updates and led to small improvements. However, the main goal remains the development of a data structure that will provide ordered arrangement of data with a $O(\log n)$ reporting for point search queries. A clever way around these computational overhead issues is the use of alternative data structures such as randomized BSTs, self-balancing BSTs, and skiplists which also provide $O(\log n)$ reporting for point search queries in a probabilistic sense.

Complexity 3.1.1 — Balanced BST. All operations in a balanced BST involve $O(\log n)$. i.e. complexity of insertion = complexity of deletion = complexity of search. Complexity of re-balancing after every update is at most $O(\log n)$; however for some balanced BST the amortized cost of re-balancing is $O(1)$.

3.2 Randomized BSTs

Balanced BSTs provide $O(\log n)$ lookup for point search queries, by storing additional information in each node and executing complex re-balancing mechanisms for dynamic updates. However, a simple alternative to maintain the ordered index and provide – parallel construction $O(\log n)$ lookup, with a probabilistic guarantee, is randomized BST. This technique eliminates the need to store additional information in every node but requires a randomization step to ensure $O(\log n)$ expected search complexity.

Definition 3.2.1 A BST is considered randomized if any node of the tree is equally probable to become the root of the tree, i.e. in case of a BST with n elements the probability of any node being a root is $1/n$.

The main drawback considered in the simple binary search tree is that the worst case complexity of search is $O(n)$, i.e. the tree is fully skewed by containing no left child or no right child. The objective is to fold the tree properly by to keep height low ($O(\log n)$). However, skewed BSTs are very rare.

▪ **Example 3.1** Let, {7, 13, 9, 5, 6, 11, 3, 2} be a set of elements and we need to create a static binary search tree to enable search for a query element. We also want to prevent the tree from skewing. The structure of a BST depends on the sequence of its elements. A skewed BST based on the above sequence will appear as [2, 3, 5, 6, 7, 9, 11, 13] or [13, 11, 9, 7, 6, 5, 3, 2] among 40320 total possible cases. ▪

? How many permutations can we have in a set of size n? How many among them are skewed?

3.2.1 Static randomized BSTs

For static collection of elements we can create a randomized binary search tree very easily. Assume $S = \{x_1, x_2, \ldots, x_n\}$ is a collection of elements given in advance for creation of a randomized binary search tree to enable search in the collection. As the probability that the BST created is very low ($2/n!$ for fully skewed), any arbitrary order of insertion of the elements in a simple BST has high probability that the tree is near balanced. Therefore for static BST considering a random permutation of the elements of S before insertion provides a near balanced BST with high probability. As a consequence the search operation will also perform in $O(\log n)$ with high probability.

3.2.2 Dynamic randomized BSTs

Creation of a static randomized BST is very simple, but maintaining a random structure during dynamic update is complex. To create a randomized BST we need to make sure that every member is equally probable to become a root tree. To ensure this randomization, we generally insert a root with probability $1/n$, when $n - 1$ elements are already there in the tree. Assume that we want to insert x in a binary tree with elements $[x_1, x_2, \ldots, x_n]$. The first step is splitting the tree into two BSTS, one with all elements less than the split point and one with more elements than the split point; this operation is similar to quicksort. A simple implementation of split algorithm can be done using a stack. The algorithm starts with a search for new elements and stores the points of comparison in the stack up-to leaf. Now we extract elements from the stack and check whether two consecutive points cross the node or not, then we update the parent pointer of the node to the next element of stack from same side. The two subtrees found in this process will be inserted as the left subtree and as the right subtree of the new element, which will be inserted in the root.

Note Insertion in the root happens once after every $n - 1$ elements, whereas other elements are inserted with probability $1 - 1/n$ in left or right subtree.

3.2.3 Analysis of randomized BSTs

Theorem 3.2.1 In a randomized BST, the required time to report a search query is $O(\log n)$.

Proof. Let T be a randomized binary search tree with n elements $[x_1, x_2, \ldots, x_{n-1}]$, i.e. $T = \Phi$ or T is created with insertion in root with probability $1/n$. The main analysis of randomized BST involves probabilistic $O(\log n)$ search reporting for point queries in a random permutation setup. Without loss of generality, we can denote a random permutation of elements as $[x_1, x_2, \ldots, x_{n-1}]$ and represent the permutation as $[1, 2, \ldots, n-1]$. Instead of proving the probability in terms of $O(\log n)$ we use harmonic numbers $H_n = \sum_{i=1}^{n} \frac{1}{i}$, since they are bounded by each other and the next term of the sequence, i.e. $H_{n-1} \leq \log n \leq H_n \log n + 1$ (integral of a function) is the area bounded by the curve.

Assume that we want to search x in a random permutation $B = [1, 2, \ldots, n-1]$. First, we need to find the probability of any point $i \in B$ to be present in the search path of x. Without loss of generality, we assume that $i \leq x$; the opposite case $i \geq x$ is similar. If i appears in the search path of x then the points $\{i+1, i+2, \ldots, \lfloor x \rfloor\}$ will never appear before i;

otherwise i cannot appear in the search path of x. Similarly, $\{\lceil x\rceil, \lceil x\rceil+1, \lceil x\rceil+2, \ldots, i-2, i-1\}$ will never appear in the search path of x before i, if $i \geq x$. Therefore, the probability,

$$\Pr\{\, i \text{ appears in the search path of } x\} = \begin{cases} 1/\left(\lfloor x\rfloor - i + 1\right) & if\, i < x \\ 1/\left(i - \lceil x\rceil + 1\right) & if\, i > x \end{cases} .$$

We now define the random variable R_i s.t. $R_i = 1$ if i appears in the search path of x and $R_i = 1$ if i does not appear in the search path of x. By definition of R_i, for any x the total length of search path, $L(x)$, is equal to expected values of total sum of R_i for all points i, i.e.,

$$\begin{aligned}
L(x) &= E\left[\sum\nolimits_{i=1}^{n} R_i\right] \\
&= E\left[\sum\nolimits_{i=0}^{x-1} R_i + \sum\nolimits_{i=x+1}^{n-1} R_i\right] \\
&= \sum\nolimits_{i=0}^{x-1} E[R_i] + \sum\nolimits_{i=x+1}^{n-1} E[R_i] \\
&= \sum\nolimits_{i=0}^{x-1} \frac{1}{([x] - i + 1)} + \sum\nolimits_{i=x+1}^{n-1} \frac{1}{/}(i - \lceil x\rceil + 1) \\
&= \sum\nolimits_{i=0}^{x-1} 1/(x - i + 1) + \sum\nolimits_{i=x+1}^{n-1} 1/(i - x + 1) \\
&= \tfrac{1}{2} + \tfrac{1}{3} + \ldots + \tfrac{1}{x+1} + \tfrac{1}{2} + \tfrac{1}{3} + \ldots + \tfrac{1}{n-x} \\
&= H_{x+1} + H_{n-x} - 2 \\
&= O(\log n)
\end{aligned}$$

■

3.3 Splay Tree

In the previous sections we discussed balanced BSTs and randomized BSTs designed to provide $O(\log n)$ lookup time for point search queries.

? Is $O(\log n)$ search time the best result we can achieve in a binary search model?

▪ **Example 3.2 — Website query.** Consider an application to facilitate Internet browsing for an organization that wants to store IP and URL data of popular websites in a BST. Assume that tree support dynamic updates are always maintained in a balanced manner. Suppose a new website is stored in a leaf and added to a tree. Although the tree is balanced, the time required for search ($O(\log n)$) is not optimum because the node is the leaf. If the website is accessed frequently, placing it in the root can make the application faster ($O(1)$). An alternative arrangement placing frequently accessed elements near the root and rarely accessed elements near leaves can make searching faster even if the tree is not balanced.

Objective 3.1 We now discuss splay trees; these are important data structures based on binary searches. They exceed height balance and use access patterns to improve search parameters where query sequences are not random. Sleator and Tarjan developed splay trees in 1985 and they are important steps in the goal of achieving optimal BSTs.

Splay trees are self-adjusting, may be unbalanced, and are dynamic BST's intended to explore the access pattern of the search while providing the $O(\log n)$ amortized time complexity for lookup of point queries. The main technique used in the formation of this data structure is splaying to deliver a node to a root. Unlike moving a node to a root via simple rotation, splaying keeps the path length small while moving an element to a root.

3.3.1 Splaying

The restructuring heuristic, called splaying, is similar to moving a root in that it performs rotations along the access path from the bottom up and moves the accessed item all the way to the root. It differs in that it performs rotations in pairs in an order that depends on the structure of the access path. To splay a tree at a node x, we repeat the following splaying step until x is at the root of the tree. Instead of repeatedly using rotation to move x to the root, use the following three types of splaying operations to move x to the root.

Zig:	If parent(x) is the root, rotate parent(x) to get x to the root (this is a terminal case).
Zig-Zig:	If parent(x) is not the root and x and parent(x) are both left or both right children, rotate grandparent(x) followed by rotate parent(x).
Zig-Zag:	If parent(x) is not the root and x is a left child and parent(x) a right child or vice versa, rotate parent(x) and then rotate the new parent(x).

3.3.2 Splaying algorithms

In this subsection, the algorithms that perform splaying operations are listed. These algorithms are called in sequence to perform restructuring of splay trees after every operation. All the algorithms run in constant time and the length of a sequence determines the complexity of a particular operation. Figure 3.3 illustrates splaying operations.

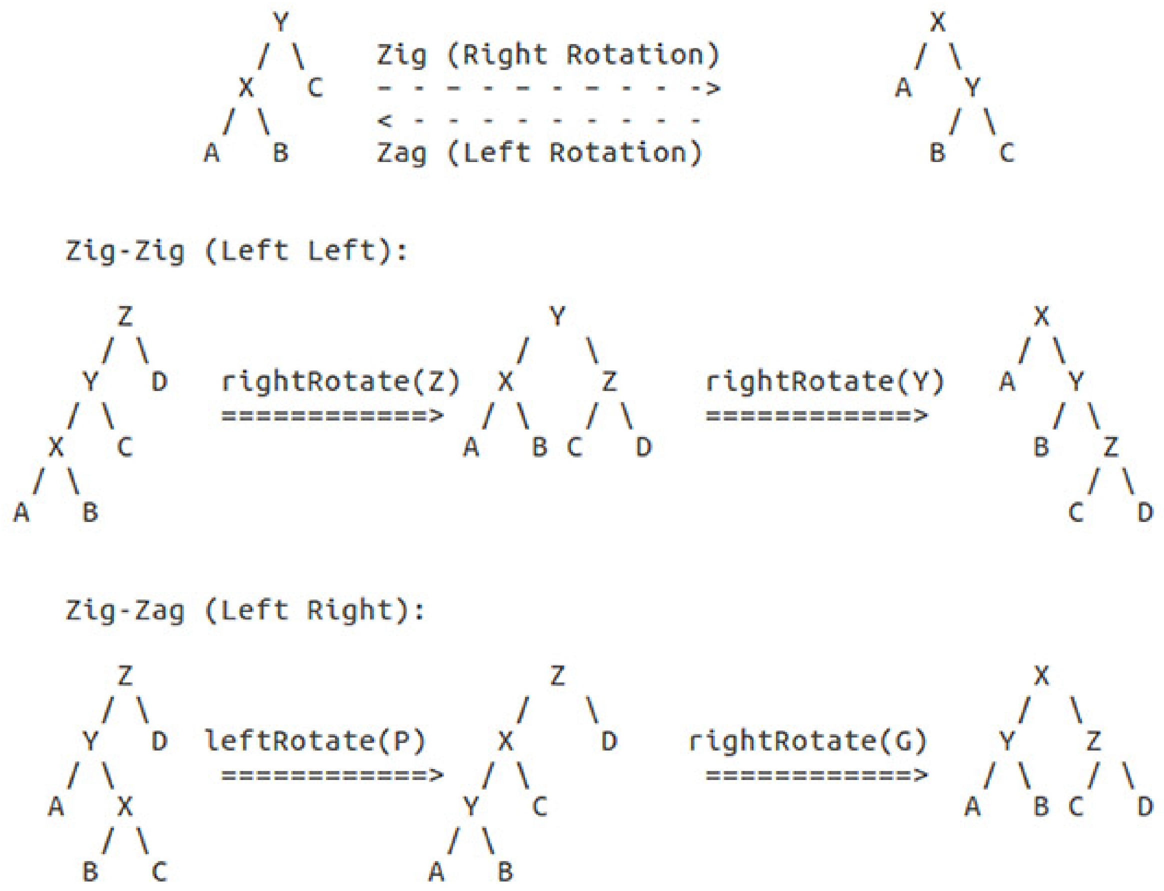


Figure 3.3: Splaying operations: Zig, Zag, Zig-Zig, and Zig-Zag.

Algorithm 6 Algorithm moving an element up-to root in splay tree

```
1: Procedure SPLAY-TO-ROOT ($x$, $T$) ▷ take input element $x$ to be deleted from splay tree, $T$
2:     While grandparent(x) exist do repeat ▷ Continue double rotations
3:         If x is left child of parent(x) and parent(x) is left child of grandparent(x) then Zig-Zig operation
```

```
4:         else if x is right child of parent(x) and parent(x) is right child of grandparent(x) then Zag-Zag operation
5:         else if x is left child of parent(x) and parent(x) is right child of grandparent(x) then Zig-Zag operation
6:         else if x is right child of parent(x) and parent(x) is left child of grandparent(x) then Zag-Zig operation
7:     If x is left child of parent(x) then Zig operation ▻ single rotate right
8:     else Zag operation ▻ single rotate left
9:     return modified $T$
```

Algorithm 7 Algorithm to perform Zig operation in splay tree

```
1:  Procedure ZIG($x$, $T$)                ▻ When parent is root and $x$ is left child
2:     State Make the x root
3:     Add parent as right child
4:     Add right subtree of x as left subtree of parent
5:     return modified $T$
```

Algorithm 8 Algorithm to perform Zag operation in splay tree

```
1:  Procedure ZAG($x$, $T$)                ▻ When parent is root and $x$ is right child
2:     Make the x root
3:     Add parent as left child
4:     Add left subtree of x as right subtree of parent
5:     return modified $T$
```

Algorithm 9 Algorithm to perform Zig-Zig operation in splay tree

```
1:  Procedure ZIG-ZIG($x$, $T$)            ▻ When $x$ is left child of parent(x) and parent(x) is left child of grandparent(x)
2:     Make the x root
3:     Add parent as right child of x
4:     Add grandparent(x) as right child of parent(x)
5:     Add right subtree of x as left subtree of parent(x)
```

```
6:    Add right subtree of parent(x) as left subtree of grandparent(x)
7:    return modified T
```

Algorithm 10 Algorithm to perform Zag-Zag operation in splay tree

```
1: Procedure ZAG-ZAG($x$, $T$)    ▷ When $x$ is right child of parent(x) and parent(x) is right child of grandparent(x)
2:    Make the x root
3:    Add parent as left child of x
4:    Add grandparent(x) as left child of parent(x)
5:    Add left subtree of x as right subtree of parent(x)
6:    Add left subtree of parent(x) as right subtree of grandparent(x)
7:    return modified T
```

Algorithm 11 Algorithm to perform Zig-Zag operation in splay tree

```
1: Procedure ZIG-ZAG($x$, $T$)    ▷ When $x$ is right child of parent(x) and parent(x) is left child of grandparent(x)
2:    Make the x root
3:    Add parent as left child
4:    Add grandparent(x) as right child of x
5:    Add left subtree of x as right subtree of parent(x)
6:    Add right subtree of x as left subtree of grandparent(x)
7:    return modified T
```

Algorithm 12 Algorithm to perform Zag-Zig operation in splay tree

```
1: Procedure ZAG-ZIG ($x$, $T$)    ▷ When $x$ is left child of parent(x) and parent(x) is right child of grandparent(x)
2:    Make the x root
3:    Add parent as right child
4:    Add grandparent(x) as left child of x
5:    Add right subtree of x as left subtree of parent(x)
6:    Add left subtree of x as right subtree of grandparent(x)
7:    return modified T
```

3.3.2.1 Insertion

Insertion in a splay tree is done in two phases. In the first phase, the element to be inserted is searched according to the binary search tree structure followed by insertion of the element as in the BST procedure. In the second phase, the newly inserted node must move up to the root using splaying operations.

Algorithm 13 Algorithm to insert new node in splay tree

```
1: Procedure INSERT-SPLAY($x$, $T$) ⊳ take input element $x$ and splay tree, $T$
2:     Search the location in $T$ to insert $x$
3:     Insert $x$ as new leaf in $T$
4:     SPLAY-to-root($x$)          ⊳ function move a point to make it root
5:     return modified $T$
```

3.3.2.2 Deletion

The deletion in a splay tree is also done in two phases. In the first phase, the element to be deleted is searched according to the binary search tree structure followed by removal of the element or a not-found report (as in BST method). In the second phase, the last access node before deletion or reported not found must move up to the root using splaying operations.

Algorithm 14 Algorithm to delete a node in splay tree

```
1: Procedure DELETE-SPLAY($x$, $T$) ⊳ take input element $x$ to be deleted from splay tree, $T$
2:     State Search $x$ in $T$
3:     If $x$ found then remove $x$
4:       SPLAY-to-root(parent of $x$) ⊳ function move a point to reach the root?
5:     return modified $T$
```

3.3.3 Performance

Performance of a splay tree depends on the splaying mechanism, in which frequently accessed nodes move towards the root where access is time constant. The worst-case of a splay tree is its height which may be O(n), but $O(\log n)$ in the average case.

3.3.3.1 Competitiveness

1. Implementation is simpler for splay trees than for other self-balancing binary search trees, such as red-black or AVL trees.
2. Average-case performance is as efficient for splay trees as it is for other trees.
3. Splay trees have small memory requirements because they present no need to store bookkeeping data.
4. The possibility of creating a persistent data structure (detailed in Part II of this book) will allow access to previous and new versions after updates. This can be useful in functional programming, and requires amortized $O(\log n)$ space per update.
5. Splay trees are very efficient when nodes contain identical keys. All tree operations preserve the order of the identical nodes within the tree, which is a property similar to stable sorting algorithms. A carefully designed find operation can return the leftmost or rightmost node of a given key.
6. A splay tree may be much more efficient if its usage pattern is non-uniform.

R Perhaps the most significant disadvantage of splay trees is that the height of a splay tree can be linear. For example, this will be the case after accessing all n elements in non-decreasing order. Since the height of a tree corresponds to the worst-case access time, this means that the actual cost of an operation can be low. However the amortized access cost of this worst case is logarithmic, $O(\log n)$. Also, the expected access cost can be reduced to $O(\log n)$ by using a randomized variant. A splay tree can be worse than a static tree by at most a constant factor.

Splay trees can change even when they are accessed in a read-only manner (i.e. by find operations). This complicates the use of such splay trees in a multi-threaded environment. Specifically, extra management is needed if multiple threads are allowed to perform find operations concurrently.

Individual operations within a sequence can be expensive which is a drawback for real time application.

For uniform access, the performance of a splay tree will be considerably worse.

3.4 Tango Tree

Tango tree was introduced in [11] to provide $O(\log \log n)$ competitive ratio with dynamically optimal BST. The tango tree data structure is very important with the theoretical point of view that it is a step forward towards the desired $O(1)$ "dynamic optimality conjecture" [7]. The center step of tango tree creation is the marking of preferred paths on a simple BST followed by decomposition of the said BST into auxiliary trees to form a tree of trees.

3.4.1 Creation of tango tree

Creation of a tango tree from a simple BST is a typical folding of that BST into a tree of trees defined by an access sequence already performed in the BST. This transformation involves three concepts: preferred path, auxiliary tree, and updating as described below.

The determination of preferred path is made on an augmented tree which can store additional bits in each node to store preferred child information. A preferred child of a node is decided from the access sequence and has nothing to do with keys associated in the node. Let, x be a node of consideration and designate $l(x)$ and $r(x)$ as left child and right child respectively. We will mark l(x) as preferred child of x if l(x) is last accessed; otherwise r(x) will be marked. This marking can be done by setting the bit value to 1. A preferred path extends from root to leaf following a preferred child. Nodes of the preferred path are compressed into an auxiliary BST and remaining subtrees are hung from an auxiliary tree. Preferred paths are depicted in Figure 3.4. An auxiliary tree may be maintained as a modified red-black tree for storing subtrees. The nodes of the preferred path are ordered and stored in the leaves of an auxiliary tree.

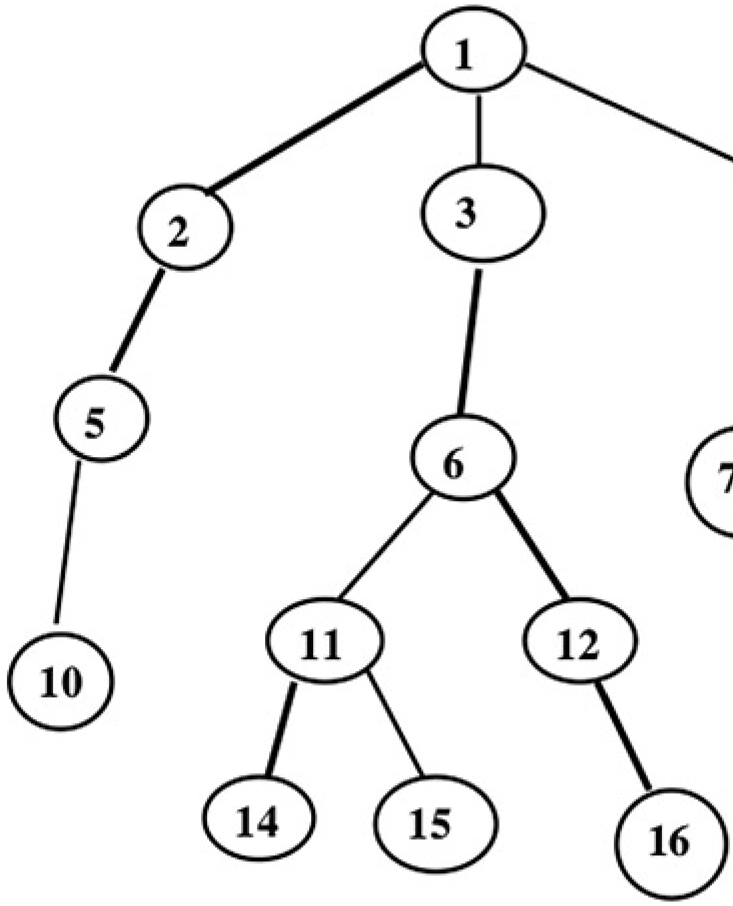


Figure 3.4: The preferred paths (dark edges) of a tango tree. Each node's preferred child is its most recently accessed child.

3.4.2 Tango analysis

To analyze the complexity of a tango tree, we need to compute the costs of point search queries and updating. Since the height of the augmented tree is $O(\log n)$, a search operation in an auxiliary tree can be done in $O(\log \log n)$.

Complexity 3.4.1 — Tango search. To search for a query point x, start from the topmost auxiliary tree and then move over the k number of subtrees where each subtree contains $O(\log n)$ nodes. The total cost of a point search is $O(k \log \log n)$.

Complexity 3.4.2 — Tango update. Cost of updating a tango tree is similar to searching, i.e. updating preferred child, split, join are all can be done with the same cost of searching, $O(k \log \log n)$, as split and join takes $O(k)$ time, where k is constant.

3.5 Skiplists

Skiplists are probabilistic data structures that may supplant balanced trees as implementations of choice in many applications. Skiplist algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster and use less space. Skiplists include algorithms for searching, insertion, and deletion and are as versatile as balanced trees. We describe and analyze algorithms so that searching for an element k away from the last element searched for takes O(log k) time. Operation like merge, split and concatenate skiplists, and implement linear list operations using skiplists (e.g., "insert this after the kth element of the list"). The skiplist algorithms for these actions are faster and simpler than their balanced tree cousins. The merge algorithm for skiplists we describe has better asymptotic time complexity than any previously described merge algorithm for balanced trees.

3.5.1 Skipping

A skiplist is a data structure for storing a sorted list of items using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. These auxiliary lists allow item lookup with efficiency comparable to balanced binary search trees (that is, with number of probes proportional to $\log n$ instead of n). Each link of the sparser lists skips over many items of the full list in one step, hence the structure's name. These forward links may be added in a randomized way with a geometric or negative binomial distribution. Insert, search and delete operations are performed in logarithmic expected time. The links may also be added in a non-probabilistic way so as to guarantee amortized (rather than merely expected) logarithmic cost. A skiplist maintains the same distribution of nodes, but without the requirement for the rigid pattern of node sizes:

1. 1/2 have 1 pointer
2. 1/4 have 2 pointers
3. 1/8 have 3 pointers
4. …..
5. $1/2^i$ have i pointers

It's no longer necessary to maintain the rigid pattern by moving values around for insert and remove. This gives us a high probability of still having O(lg N) performance. The probability that a skiplist will behave badly is very small. A skiplist is built in layers. The bottom layer is an ordinary ordered linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer i+1 with some fixed probability p (two commonly used values for p are 1/2 or 1/4). On average, each element appears in 1/(1−p) lists, and the tallest element (usually a special head element at the front of the skiplist) in lists. A search for a target element begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, or the search reaches the end of the linked list, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most 1/p, which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. Therefore, the total expected cost of a search in known when p is a constant. By choosing different values of p, it is possible to trade search costs against storage costs. Figure 3.5 details the structure and pointers.

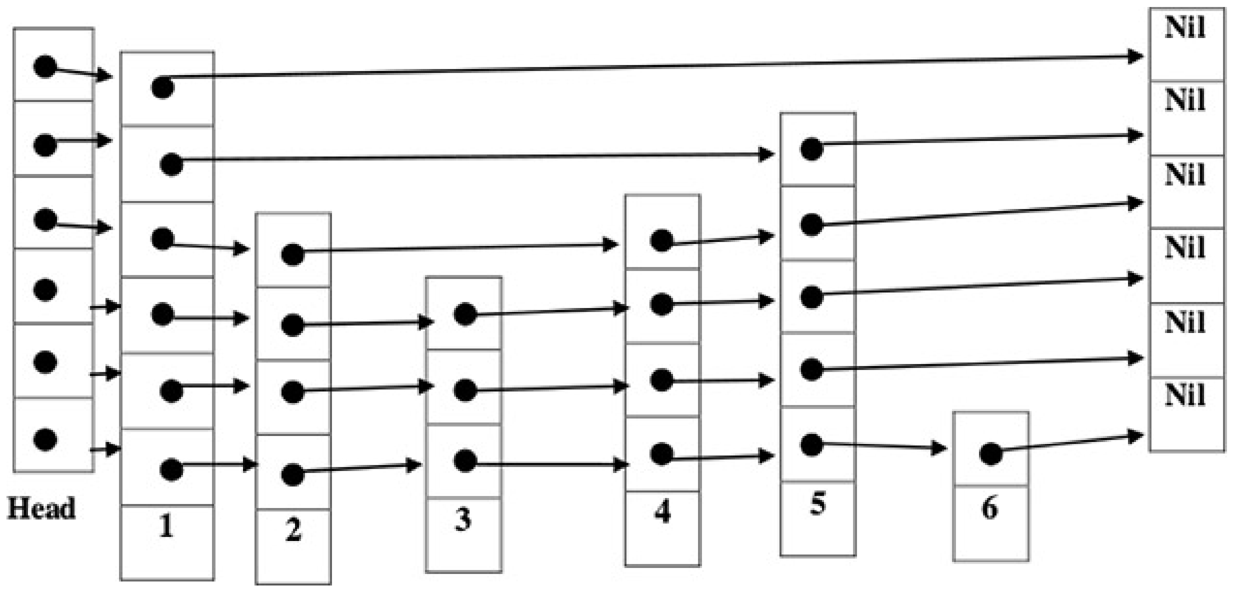


Figure 3.5: Skiplist node structure and skipping pointers

3.5.2 Dynamic updates

The elements used for a skiplist can contain more than one pointer since they can participate in more than one list. Insertions and deletions are implemented much like the corresponding linked-list operations, except that "tall" elements must be inserted into or deleted from more than one linked list. Operations, which force us to visit every node in ascending order (such as printing the entire list), provide the opportunity to perform a behind-the-scenes de-randomization of the level structure of the skiplist in an optimal way, bringing the skiplist to search time. (Choose the level of the *i*-th finite node to be 1 plus the number of times we can repeatedly divide i by 2 before it becomes odd. Also, i = 0 for the negative infinity header as we have the usual special case of choosing the highest possible level for negative and/or positive infinite nodes.) However this also allow us to know where all of the nodes above level 1 are and delete them.

3.5.2.1 Size of header node

The size of the header node (the number of forward references it has) is the maximum size of any node in the skiplist and is chosen when the empty skiplist is constructed (i.e. it must be predetermined). Dr. Pugh [193] has shown that the maximum size should be chosen as log 1/p N. For p = 1/2, the maximum size for a skiplist with 65,536 elements should be no smaller than log 2 65536 = 16.

3.5.2.2 Initialization

A new list is initialized as follows:

1. A node called NIL is created and its key is set to a value greater than the greatest key that could possibly be used in the list (i.e. if the list will contain only keys between 1 and 999, then 1000 may be taken as the key in NIL). Every level ends with NIL.
2. The level of a new list is 1
3. All forward pointers of the header point to NIL.

3.5.2.3 Searching

1. Start at the highest level of the list.
2. Move forward following the pointers at the same level until the next key is greater than the searched key.
3. If the current level is not the lowest, go down one level and repeat the search at that level from the current node.
4. Stop when the level is 1 and the next key is greater than the searched key.
5. If the current key is the searched key return the value of that node. Otherwise, return a failure.

3.5.3 Probabilistic analysis of skiplist

Theorem 3.5.1 Expected total number of pointers at all levels $O(2n)$.

Theorem 3.5.2 Expected height of skiplist is $O(\log n)$.

Theorem 3.5.3 Expected length of search path in skiplist is $O(\log n)$.

Proof. Proofs of these theorems are similarized to proofs for randomized BSTS and are left as exercises for readers. ▪

3.5.3.1 Time complexity

The expected time to find an element (and therefore to insert or remove) is O(lg N). It is possible for the time to be substantially longer if the configuration of nodes is unfavorable for a particular operation. Since the node sizes are chosen randomly, it is possible to get an unacceptable run of sizes. For example, it is possible that each node will be generated with the same size, producing the equivalent of an ordinary linked list. An unsatisfactory run will be less important in a long skiplist than in a short one. The probability of poor performance decreases rapidly as the number of nodes increases.

3.5.3.2 Comparison of BSTs and skiplists

The BST is efficient but may become easily unbalanced after several insertions and deletions. Balanced trees (2-3 trees and red-black trees) are guaranteed to remain balanced and as do the basic operations in O(log n) in the worst case. Theoretically, they are efficient but their implementation is complicated. On the other hand skiplists are easier to implement. The algorithms for insertions and deletions are simpler and faster. They do not guarantee O(log n) performance but they do have an O(log n) performance in the average case (for insert, delete, search) and the probability of a high deviation from the average is quite high.

3.6 Static and Dynamic Optimality

The search operation in a binary search tree starts at root and follows the child nodes successively till the query object is found. The cost of search operation in a binary search tree is computed as the number of comparisons, generally one comparison generally one comparison at every node of the search path, which is equal to the length of search path.

3.6.1 Search optimality in BST

Let's consider the search sequences of length m: X={X_1, X_2, … X_m}. To avoid issues involving small sequences and the initial state of the tree, we assume m is sufficiently long (often only $m = \Omega(n)$ is needed) and that the tree is in some canonical initial state. A BST-model algorithm is simply a way of choosing a sequence of BST unit cost primitives to execute each search. A BST-model algorithm is online if its choice of BST unit cost primitives to execute search X_i is a function of X_1, … X_i. The online BST model is still very permissive, as only unit cost operations are counted, and unlimited computation could be done to determine these operations. What is normally thought of as a BST is an online BST model algorithm that can be implemented on a BST where $O(\log n)$ bits of data can be augmented on every node, and where unit cost operations are chosen based on the current search, the contents of the node currently pointed to, including any augmented data, and $O(\log n)$ bits of global state. Such a BST algorithm is called a real-world BST.

We let $R_A(X)$ denote the cost in the BST model to execute X using some BST-model algorithm A. Let $OPT(X)$ be the fastest runtime of any BST that can execute X; that is $OPT(X) = min_A\ R_A\ (X)$. Given enough time (i.e. exponential in m), $OPT(X)$ can be computed exactly, and an offline algorithm A such that $R_A\ (X) = OPT(X)$ can be produced. Splay trees are a BST structures introduced by Sleator and Tarjan. They use a simple restructuring heuristic, to move the searched item to the root. This heuristic has the following effect on nodes other than the one searched: if the node x is at depth d and l of the ancestors of x are on the search path, after the search x will be at depth d + l 2 + O(1). The focus of this work is on the dynamic optimality conjecture.

Definition 3.6.1 — Dynamic optimality conjecture. We refer to any BST algorithm A such that $R_A(X) = O(OPT(X) + f(n))$ for some f(n) as dynamically optimal. Rather then focus on splay trees, we focus on whether there are any dynamically optimal BSTs. We present several different formulations of this, from weakest to strongest.

3.6.2 Static optimality

It is possible to compute in polynomial time (in, say, the RAM model) an algorithm A such that $R_A(X) = O(OPT(X))$. As we have noted that computing such an algorithm is possible, given enough time, this question concerns only running time, and is the easiest of the questions presented. We believe that computing OPT(X) exactly is likely to be NP-complete. NP completeness means that instead of a sequence of single searches to be executed on a BST, a sequence of sets of items are provided and the algorithm can order the searches in each set in whatever manner is beneficial to it. Computing the exact optimal cost for executing such a sequence of sets of searches may prove to be NP-complete.

3.6.3 Dynamic optimality

Is there an online BST algorithm A such that $R_A(X) = O(OPT(X))$? In this statement of the problem, A could do significant computation in order to determine which BST unit-cost operation to perform at every step, subject only to the requirement that it is online. This conjecture represents the claim that there is no asymptotic difference in power between online and offline algorithms in the BST model. Such equivalence in power between online and offline power is generally not possible in more permissive models, and is typically only found in very strict models such as the optimality of the move-to-front rule for search in a linked list. In more permissive models like the RAM, an offline algorithm could fill an array M such that $M[i] = xi$ and thus could trivially achieve offline performance that an online algorithm could never match.

Note The search cost of a search BST algorithm is simply the depth of the node to be searched. Any rotations or pointer movements off the search path are free; in this way the BST can be arbitrarily reconfigured between searches at zero cost. If one were to try to adapt this method to the standard online BST model cost function, a reasonable starting point would be to try to determine if there is any cohesion of the trees produced by the method from one search to the next, and to try to figure out if one could use such cohesion to transform one tree to the next in time proportional to the search cost.

3.7 Exercises

Exercise 3.1 A family of trees is called balanced if every tree in the family has height $O(\log n)$, where n is the number of nodes in the tree. For each property below, determine whether the family of binary trees satisfying that property is balanced. If your answer is "no", provide a counterexample. If your answer is "yes", give a proof.
(a) Every node of the tree is either a leaf or it has two children.
(b) The average depth of a node is $O(\log n)$. ■

Exercise 3.2 Let Φ be the potential function used to analyze splay trees, i.e., $\Phi = \sum_{v \in T} rank(v)$. Prove that the potential of a complete binary tree is O(n) and that the potential of a rooted path is O(nlog n). ■

Exercise 3.3 Show how to modify a balanced binary search tree to find max element <x query in O(log n) time. ■

Exercise 3.4 Describe an algorithm select(S, k) to find the kth sized element in a skiplist S with n elements. You can add a field to each node of S. The average time of the algorithm should be $O(\log n)$. ■

Exercise 3.5 Write an algorithm that builds a skiplist S from the given BST T with n elements, such that the worst query time in S will be $O(\log n)$. T can be unbalanced. The time complexity of the algorithm should be $O(n)$. ■

Chapter 4

Findset, Find Min, and Find Word

In this chapter, we explore data structures for point search queries and widely used data structures to answer some very important queries. In the first section we discuss disjoint-set data structure which is well known for its applications in network algorithms and provides find set queries. Sections 4.2 and 4.3 cover binomial heaps and Fibonacci heaps; both are known for their roles in creating priority queues. Binomial heaps efficiently perform delete min queries. Fibonacci heaps effectively decrease key operation. Section 4.4 covers various kinds of strings and their variants that are designed for membership queries. The final section covers the use of inverted indices for handling keyword queries. The chapter ends with a set of exercises.

4.1 Disjoint Sets

The disjoint-set data structure, also known as a union find data structure was developed to represent a disjoint collection of sets which allows simple operations of set theory such as creating singleton sets, unions of to sets, and most importantly, for determining to which set an element belongs. This data structure is well known for its wide applications in network algorithms.

Definition 4.1.1 A disjoint-set data structure organizes a collection, $S = S_1, S_2, \ldots, S_k$ of mutually disjoint sets, such that each element of any set is represented as an object and each set has a representative element inside the set. Moreover, it is possible to unite two disjoint sets S_i and S_j to form a new set $S_k = S_i \cup S_j$, which contains elements of both.

4.1.1 Operations on disjoint-set data structure

The disjoint-set data structure is characterized by its unique operations and provides the way to understand the data which is always arranged as a disjoint collection of sets. The operations are described below. Separate three operations below by vertical spaces so they're easier to read.

makeset(x): The makeset(x) operation uses an element *x* as input and returns a new singleton set represented by itself.

union(x,y): The union(x,y) operation uses two sets represented by *x* and *y* elements as inputs and unites those sets to return a new set consisting of elements from both the sets.

findset(x): The findset(x) operation uses an element as input and returns the representative of the set to which it belongs.

▪ **Example 4.1 — Find connected components of a graph** $G(V, E)$. In this example, we present an algorithm to compute connected components of a given graph. The algorithm is very straightforward and given below. ▪

Algorithm 15 Algorithm to determine connected components of a graph G

```
1: procedure CONNECTED-COMPONENTS G(V,E)    ▷ computes connected components of a graph
2:     for ∀ v ∈ V do
3:         makeset(v)          ▷ creates |V| number of singleton sets
4:         remember representatives
5:     for ∀ (u, v) ∈ E do
6:     if findset(u) ≠ findset(v) then
7:         union(u,v)          ▷ unite if different set
8:         reduce representative
9:     return representative        ▷ one representative for each component
```

4.1.2 Representations of disjoint sets

Given abstraction for disjoint sets, the next task is to implement this data structure in an efficient manner. A trivial implementation uses list representation, i.e., each set will be a link list and specify disjoint set operations there. Another implementation uses disjoint forests which are combinations of tree data structures. Both the representations are discussed below.

4.1.3 Link-list representations of disjoint sets

In a link-list representation of a disjoint set (Figure 4.1). Every set is realized as a link list, nodes of the lists are elements of the sets, representative elements should appear at the first location of a list and every node will have two pointers instead of one. One pointer will point to the next element, as in an ordinary link list. The second pointer will point to the representative element of the set (the first node).

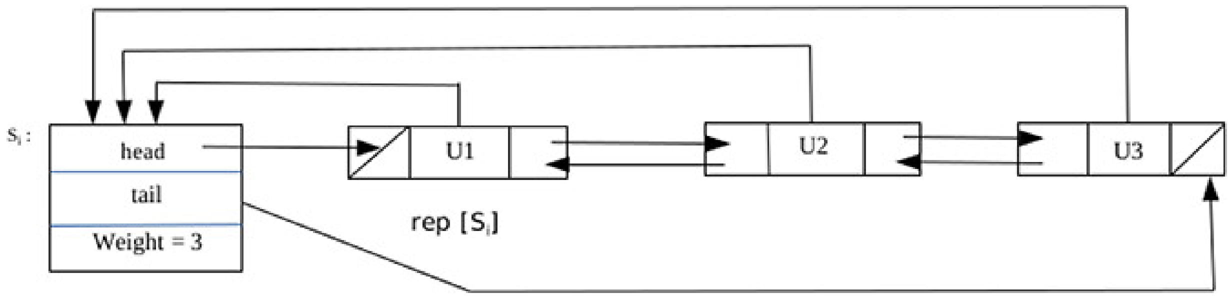


Figure 4.1: Disjoint sets with link representation

In this representation, makeset(x) creates a single node list with reference to itself. There is no operation to create a non-singleton set; larger sets are created via union operations. The union (x, y) joins two lists, generally the larger one appears first to reduce complexity. The findset operation is performed following the representative pointer associated with each element.

Definition 4.1.2 — Weighted union heuristic. Let $|S_1| = n$ and $|S_2| = m$ be the cardinality of two disjoint sets, where $m \leq n$. A union operation must append the smaller list, S_2 at the end of S_1 to minimize the pointer update, $O(m)$.

Complexity 4.1.1 — Disjoint sets with link representation.
Complexity of makeset operation is $O(1)$.

Complexity of findset operation is $O(1)$.
Complexity of union operation is $O(m + n)$.

Theorem 4.1.1 A union operation takes $\Omega(n)$ time, but a sequence of m disjoint set operations on a collection of n elements required $O(m + n\ log\ n)$ time.

Proof. If all operations are makeset or findset it can be at most $O(m)$. If m is large enough and maximum possible union operation is performed then at most $(n - 1)$ union is possible but the cost of union is only $O(1)$. The worst case of union complexity occurs when set sizes are equal, which will take time $O\ (n\log n)$ similar to merge sort. ▪

4.1.4 Forest representations of disjoint sets

Objective 4.1 — Disjoint forest representation. In link-list representation of a disjoint set, the makeset and findset perform in constant time whereas the union is almost linear. However, the findset and union are called in roughly the same order in many applications. Balancing the cost of union with findset operation is a good trade-off to minimize the total cost of application, which is the main motive of disjoint forest representation.

Realizing disjoint-set data structure with a collection of trees was first proposed to make proper use of pointers (Figures 4.2 and 4.3). In this representation sets are trees with reverse pointers, i.e., a parent pointer in every node instead of child pointers in which roots will point to themselves and represent the set as well. In makeset operation a root is created using an input element. The findset operation accesses the sequential parent pointer to reach the root element where the parent pointer points to itself and represents the set. The union operation joins the two tree the root of the smaller rank tree becomes the child of the root of the higher rank tree.

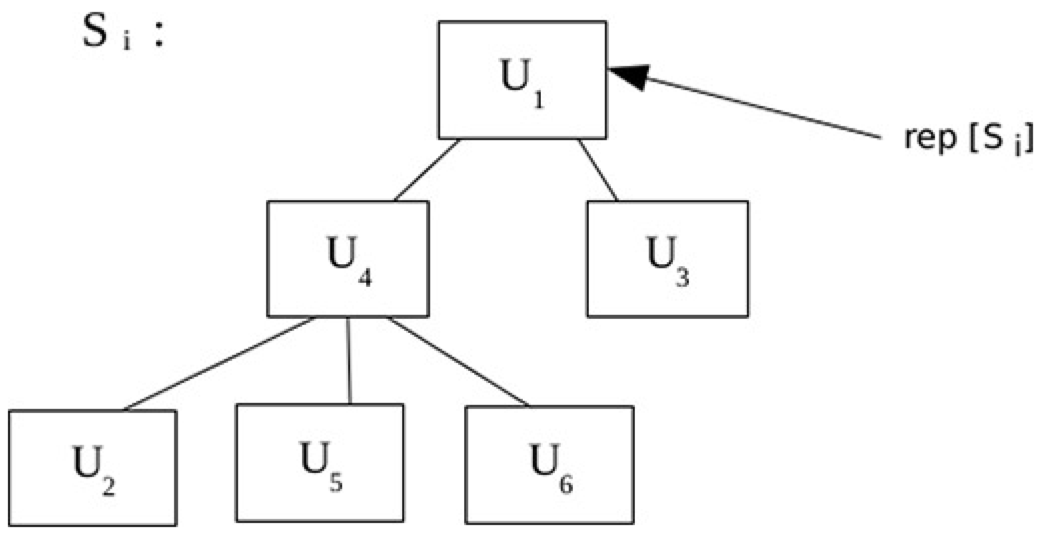


Figure 4.2: Disjoint sets with forest representation

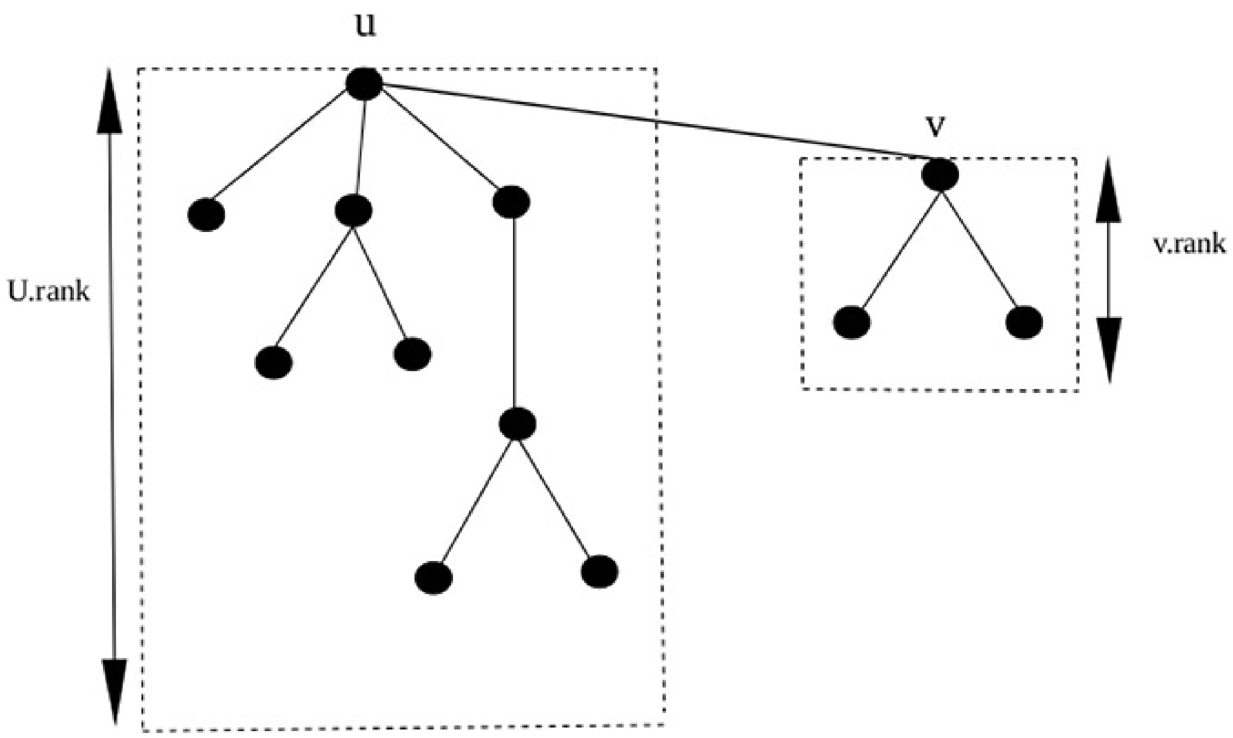


Figure 4.3: Union by rank in disjoint sets

Definition 4.1.3 — Union by rank heuristic. In union(x,y) operation of two disjoint sets represented by the elements x and y respectively in the tree. As illustrated in the diagram below representation the reference pointer associated with the tree with smaller rank is updated by pointing it to the root of the higher rank tree. Rank of the resulting tree is equal to the rank of the larger tree.

Definition 4.1.4 — Path compression heuristic. The findset operation searches for roots by following parent pointers. It detaches all nodes appearing in the path and makes them direct children of a root.

Theorem 4.1.2 For a sequence of m disjoint set operations on a collection of n elements, represented as disjoint tree forest supported with union by rank and path compression, where $m \geq n$ implement, the total cost is $O(m.\alpha(n))$, where $\alpha(n) = \min\{k : A_k(1) \geq n\}$, the inverse of the variant of Ackerman function $A_k(i)$.

Note $\alpha(n) \geq 4$ means n greater than the number of atoms in the universe.

Proof. Note that findset can use its worst case $O(n)$ it will happen only once due to the path compression heuristic, whereas makeset and union utilize constant time. If all operations are makeset or findset, the result can be $O(m)$ at most. See Figure 4.4.

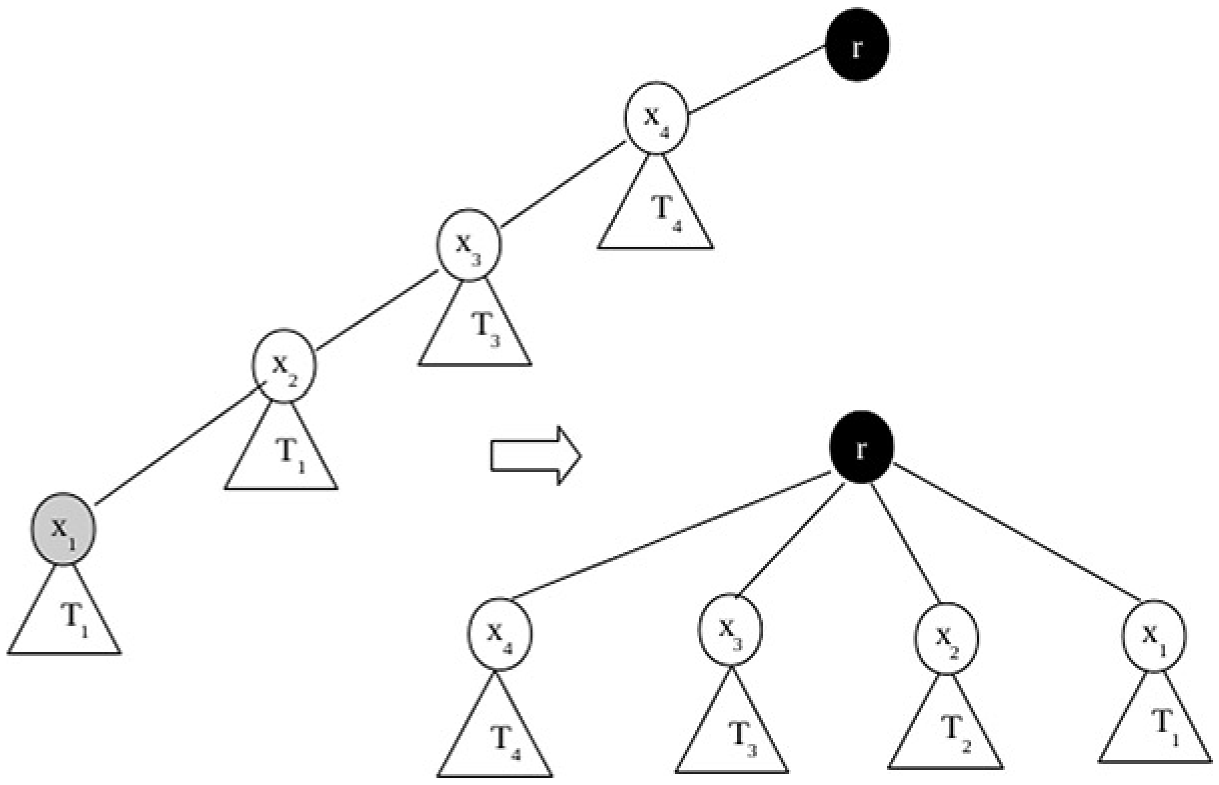


Figure 4.4: Disjoint-sets path compression during findset operation

4.2 Binomial Heap

Heaps are common data structures used mainly to implement priority queue operations. Binomial heaps are taught in basic data structure courses. They are binary trees used in moderately complex operations like findmin in $O(1)$, deletemin in $O(\log n)$, and buildheap in $O(n)$. However, binomial heaps are not suitable for dynamic update operations.

Objective 4.2 Binomial heap data structures provide amortized constant time finemin and deletemin operations and they efficiently perform insert and meld operations.

A binomial heap is a specific implementation of the heap data structure using a collection of binary trees connected by a circular link list in the roots of the respective trees. Each binary tree is a min heap (can be made max as well), i.e. minimum element in the root. The numbers on nodes in a binary tree may be 1, 2, 4, 8 etc. To create a binomial heap of any size we need to represent a chosen number as a sum of binary numbers and create trees accordingly.

4.2.1 Creation and updates of binomial heap

Binomial heaps are created as a collection of binomial trees and connected in roots by a circular link list. The sizes of the binomial trees are determined by the elements of binomial sequence $\{1, 2, 4, 8, \ldots\}$ i.e., $\{2^0, 2^1, 2^2, 2^3, \ldots, 2^k, \ldots\}$. A binomial tree B_k of size 2^k is called order k binomial tree or rank k binomial tree.

Binomial trees demonstrate remarkable properties because they are represented by binomial numbers. See Figures 4.5 and 4.6.

1. For every $k \in \mathcal{N}$, $B_k = B_{k-1} + B_{k-1}$.
2. A binomial heap H_1 of size $13 = 2^3 + 2^2 + 2^0$ can be written as $H_1 = B_3 + B_2 + B_0$.
3. Similarly, a binomial heap H_2 of size $23 = 2^4 + 2^2 + 2^1 + 2^0$ can be written as $H_1 = B_4 + B_2 + B_1 + B_0$.
4. $H_3 = meld(H_1, H_2)$ will have $36 = 2^5 + 2^2$ node and can be written as $H_3 = B_5 + B_2$.

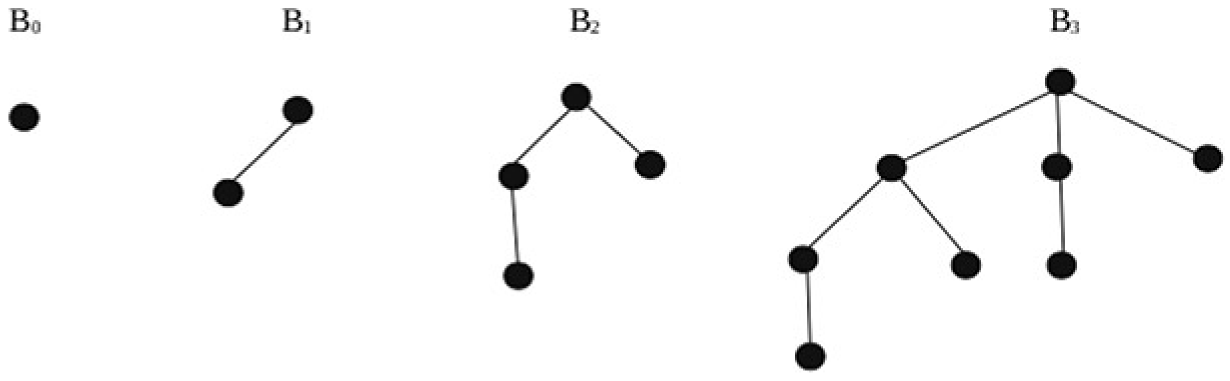


Figure 4.5: Binomial trees

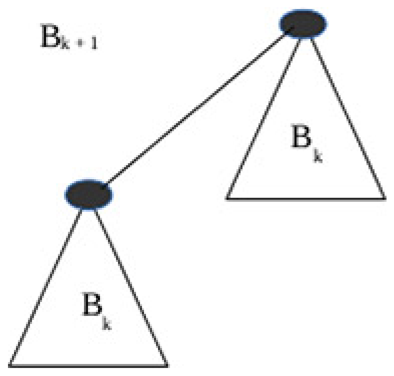


Figure 4.6: Merging two binomial trees of same order

4.2.2 Operations of Binomial Heap

Binomial trees are constructed to include heap properties, i.e., values of children are always greater than values of parents. See Figure 4.7. Creation, update and finding minimum value are generally done through the operations *create*(B_0), *meld*(H_1, H_2), *findmin*(), *deletemin*(), *deletenode*(x) and *decreasekey*(x, Δx).

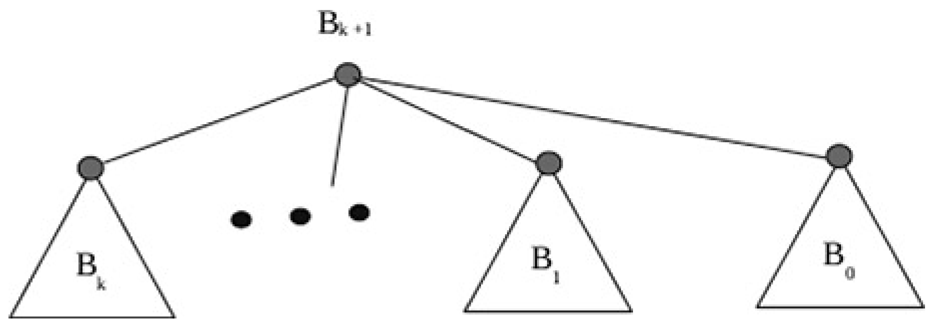


Figure 4.7: Merging binomial trees of lower order

4.2.2.1 *create(B_0)*

The (B_0) operation creates a binomial tree of size 1; it contains only one root as shown in Algorithm 16.

Algorithm 16 Algorithm to perform *create* (B_0) operation

```
1: Procedure CREATE $B_0$, $x$        ⊳Create a $B_0$ from the element $x$
2:     Make the x root
3:     Add key value of x
```

```
4:     return $B_0$
```

4.2.2.2 *meld(H_1, H_2)*

One useful operation of a binomial heap is *meld*(H_1, H_2). Other operations like deletemin and deletenode depend on the meld operation and their complexities are determined by the complexity of the meld. Algorithm 17 details the meld operation.

4.2.2.3 *findmin()*

To search and return the minimum value of a binomial heap we execute the findmin() operation that traverses the list of the tree roots. One of these nodes must have minimum value. This can be done in $O(1)$ amortized complexity though the worst case is $O(\log n)$. Algorithm 18 depicts the findmin function.

Algorithm 17 Algorithm to perform *meld*(H_1, H_2) operation

```
1: procedure H = MELD ($H_1$ $H_2$        ⊳ Join two binomial heaps
2:     for $k = 0 \rightarrow max\ \{\log |H_1|, \log |H_2|\}$ do do
3:         if more than one $B_k$ then join root to create a $B_{k+1}$
4:         Add $B_{k+1}$ to list of $H$
5:     return $H$
```

Algorithm 18 Algorithm to perform *findmin*() operation

```
1: procedure FINDMIN ($H$)        ⊳ Find and return the minimum value of heap
2:     Find minimum of link list
3:     return minimum value
```

4.2.2.4 *deletemin()*

In the *deletemin*() operation first we call findmin(H) to find the minimum root from the collection of binomial trees associated with the binomial heap H, then we delete that root and a new binomial tree followed by a meld operation. For example, if in a binomial heap

$H = B_k + B_{k-1} + \cdots + B_i + \cdots + B_1 + B_0$ and minimum is the root of B_i, then after removing root of B_i it will become $B_{i-1} + B_{i-2} + \cdots + B_1 + B_0$ and the new binomial heap will be obtained by $meld(H \setminus B_i, B_{i-1} + B_{i-2} + \cdots + B_1 + B_0)$. This operation has the same complexity as meld as findmin operates in constant time. Algorithm 19 depicts the deletemin operation.

Algorithm 19 Algorithm to perform *deletemin()* operation

```
1: procedure DELETEMIN (H) ▷ Delete the minimum element of the heap H
2:     root (B_i) = findmin(H)
3:     remove root of B_i
4:     return meld(H \ B_i, B_{i-1} + B_{i-2} + … + B_1 … B_0)
```

4.2.2.5 *decreasekey(x, Δx)*

Assume we want to decrease the value of node x in a *decreasekey*(x, Δx) operation but decreasing an intermediate key will violate the heap property. So we need compare the node with its parent in a recursive process up-to root or stop where the heat property is maintained. The worst case complexity of decreasekey operation is $O(\log n)$, but it is an overestimate. The operation will proceed in a binomial heap if a binary tree containing most elements and a decreasekey operation is called in leaf node; this is a rare situation. Algorithm details the decreasekey operation.

Algorithm 20 Algorithm to perform *decreasekey*(x, Δx) operation

```
1: procedure DECREASEKEY (x, Δx)    ▷ Decrease the value of a particular node
2:     Decrease x by Δx
3:   while parent(x) > current node do swap with parent
4:     Update current node to parent
5:   return New tree
```

4.2.2.6 *deletenode(x)*

To delete a particular node from a binomial heap which reside in some binary tree following steps are performed:

1) set the value of that node sufficiently small (may be 0 or negative).
2) a bubble up of node up-to root to maintain heap property.
3) extract-min() operation will delete the node as well as recreate the heap.

The worst case complexity of deletenode operation is O(logn), but it is an over estimate because. Algorithm 21 shows the deletenode operation.

Algorithm 21 Algorithm to perform *deletenode(x)* operation

```
1: procedure DELETENODE ($x$)    ▷ Delete an arbitrary node in binomial heap.
2:     decreasekey($x$,$\infty$)
3:     deletemin()
4:     return New Heap
```

4.2.3 Complexity

The complexity issues of the six binomial heap operation are summarized below.

1. Complexity of *create*(B_0) is $O(1)$.
2. Complexity of *meld*(H_1, H_2) is $O(\log n)$ worst case (over estimated).
3. Complexity of *findmin*() is $O(1)$ amortized.
4. Complexity of *deletemin*() is $O(\log n)$ over estimated.
5. Complexity of *deletenode*(x) is $O(\log n)$ over estimated.
6. Complexity of *decreasekey*(x, Δx) is $O(\log n)$ over estimated.

▪ **Example 4.2 — Prim's algorithm to find Minimal Spanning Tree.** Prim's algorithm uses binomial heaps to store the key values of unexplored vertices. The algorithm searches every iteration to look for minimum edges in explored vertices and unexplored vertics determined by the minimum values of the priority queue. The iterations continue until the priority queue is empty. If the graph under consideration is $G(V, E)$, then maximum size of priority queue is $|V|$, so number of findmin is $|V|$ and number of deletemin is also $|V|$. However, the number of decrease keys may be $O(E)$ which may cost up-to $O(E\log V)$ without amortized analysis.

4.3 Fibonacci Heaps

Objective 4.3 Priority queue is a ubiquitous data structure in theoretical computer science. Fibonacci heaps provide a fast and efficient solution for decrease key operation, thus making the network optimization algorithms faster.

The fibonacci heap data structure developed by Fredman and Tarjan in 1984 gives an efficient way to implement decrease keys of priority queues [19]. See Figure 4.8 goal is to find a way to minimize the number of operations needed to compute the Minimal Spanning Tree or Shortest Path tree, the kind of operations that we are interested in are *insert*, *decrease-key*, *merge*, and *delete-min*. We can achieve this minimization goal by using lazy operations to reduce amortization complexity.

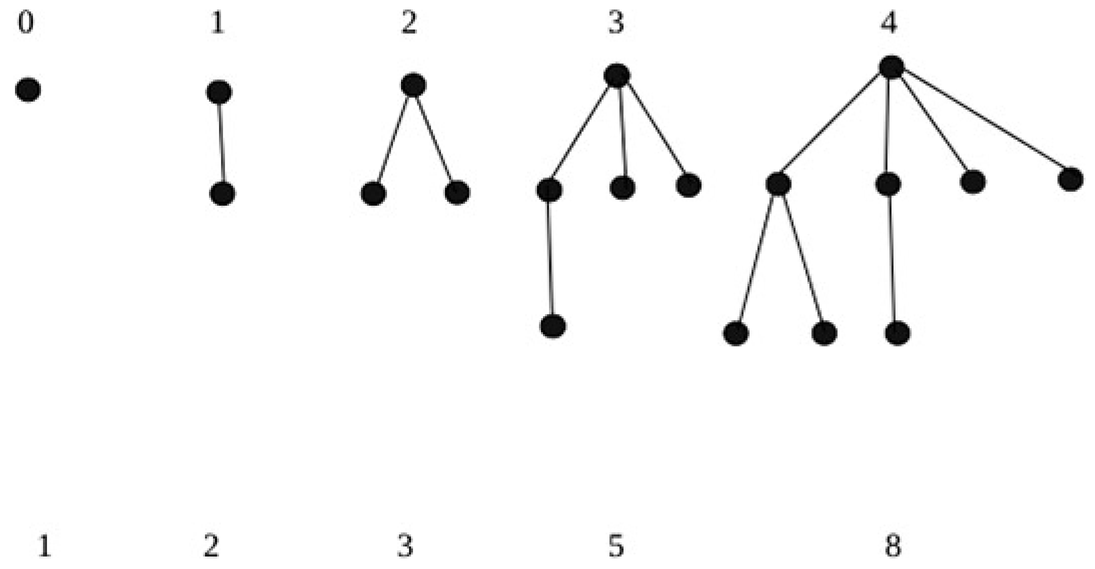


Figure 4.8: Fibonacci trees of different ranks and sizes

Fibonacci heaps make use of heap-ordered trees, but uses Fibonacci numbers instead of the binomial numbers used in binomial heaps.

4.3.1 Properties of a Fibonacci heap

1. The roots of these trees are kept in a doubly-linked list (the "root list" of H);
2. The root of each tree contains the minimum element in that tree (this follows from being a heap-ordered tree);
3. We access the heap by a pointer to the tree root with the overall minimum key;
4. For each node x, we keep track of the *rank* (also known as the *order* or *degree*) of x, which is just the number of children x has; we also keep track of the *mark* of x, which is a Boolean value whose role will be explained later.

Note For each node, we have at most four pointers that respectively point to the node's parent, to one of its children, and to two of its siblings. The sibling pointers are arranged in a doubly-linked list (the "child list" of the parent node).

4.3.2 Inserting, merging, cutting, and marking

Inserting a node *x*. We create a new tree containing only x and insert it into the root list of H; this is clearly an $O(1)$ operation.

Merging two trees. Let x and y be the roots of the two trees we want to merge; then if the key in x is no less than the key in y, we make x the child of y; otherwise, we make y the child of x. We update the appropriate node's rank and the appropriate child list; this takes $O(1)$ operations.

Cutting a node. If x is a root in H, we are done. If x is not a root in H, we remove x from the child list of its parent, and insert it into the root list of H, updating the appropriate variables (the rank of the parent of x is decremented, etc.). Again, this takes $O(1)$ operations. (We assume that when we want to find a node, we have a pointer hanging around that accesses it directly, so actually finding the node takes $O(1)$ time.)

Marking. We say that x is marked if its mark is set to “true”, and that it is unmarked if its mark is set to “false”. A root is always unmarked. We mark x if it is not a root and it loses a child (i.e., one of its children is cut and put into the root-list). We unmark x whenever it becomes a root. We will make sure later that no marked node loses another child before it is cut and reverts to unmarked status.

4.3.3 Decreasing keys and delete-min operation

At first, *decrease-key* does not appear to be any different than *merge* or *insert*; we simply need to find the node and cut it off from its parent, then insert the node into the root list with a new key. This requires removing it from its parent’s child list, adding it to the root list, updating the parent’s rank, and (if necessary) the pointer to the root of smallest key. This takes $O(1)$ operations.

The *delete-min* operation works in the same way as *decrease-key*: Our pointer into the Fibonacci heap is a pointer to the minimum keyed node, so we can find it in one step. We remove this root of smallest key, add its children to the root-list, and scan through the linked list of all the root nodes to find the new root of minimum key. Therefore, the cost of a *delete-min* operation is $O(\#\text{ of children})$ of the root of minimum key plus $O(\#\text{ of root nodes})$; in order to make this sum as small as possible, we have to add a few bells and whistles to the data structure.

4.3.4 Algorithm for Fibonacci heaps

Maintain a list of heap-ordered trees.

4.3.4.1 *insert*

Add a degree 0 tree to the list.

4.3.4.2 *delete-min*

We can find the node we wish to delete immediately since our handle to the entire data structure is a pointer to the root with minimum key. Remove the smallest root, and add its children to the list of roots. Scan the roots to find the next minimum. Then consolidate all the trees (merging trees of equal rank) until there is ≤ 1 of each rank.

Assuming that we have achieved the property that the number of descendants is exponential in the number of children for any node, as we did in the binomial trees, no node has rank $> c \log n$ for some constant c. From this assumption, we can conclude that every root (including the smallest) has $O(\log n)$ children, and that consolidation leaves us with $O(\log n)$ roots.

The consolidation is performed by allocating buckets of sizes up to the maximum possible rank for any root node, which we just showed to be $O(\log n)$. We put each node into the appropriate bucket, at cost $O(\log n)$ + O(# of roots). Then we march through the buckets, starting at the smallest one, and consolidate everything possible. This again incurs cost $O(\log n)$ + O(# of roots).

4.3.4.3 *decrease-key*

Cut the node, change its key, and insert it into the root list as before, Additionally, if the parent of the node was unmarked, mark it. If the parent of the node was marked, cut it off also. Recursively do this until we get up to an unmarked node. Mark it.

4.3.5 Amortized analysis for Fibonacci heaps

Define $\Phi(DS) = k \cdot$ (# of roots in DS + $2 \cdot$ # marked bits in DS). Note that *insert* and *delete-min* do not ever cause nodes to be marked - we can analyze their behaviour without reference to marked and unmarked bits. The parameter k is a constant that we will conveniently specify in the analysis below. We now analyze the costs of the operations in terms of their amortized costs (defined to be the real costs plus the changes in the potential function).

4.3.5.1 *insert*

The amortized cost is $O(1)$ which represents actual work plus k change in potential for adding a new root. Since k is a constant, $O(1) + k = O(1)$ total amortized cost.

4.3.5.2 *delete-min*

For every node that we put into the root list (the children of the node we have deleted), plus every node that is already in the root list, we do constant work putting that node into a bucket corresponding to its rank and constant work whenever we merge the node.

Let r be the number of roots on the root list before we add the minumum root's children. From our assumption that the size of a tree be exponential in its rank, we know that the number of children added is $O(\log n)$. Thus, our real costs are putting the roots into buckets ($O(r) + O(\log n)$), walking through the buckets ($O(\log n)$), and doing the consolidating tree merges ($O(r) + O(\log n)$). On the other hand, our change in potential is $k \cdot (O(\log n) - r)$ (since there are at most $O(\log n)$ roots after consolidation). Thus, total amortized cost is $O(r) + O(\log n) + k \cdot (O(\log n) - r) = O(\log n) + (O(r) - k \cdot r)$. Now, we see that if we set k to be greater than the constant term hidden in the $O(r)$ notation, that term disappears, and the amortized cost becomes O(log n).

4.3.5.3 *decrease-key*

The real cost is $O(1)$ for each individual cut, key decrease and re-insertion. The only problematic issue is the possibility of a "cascading cut" - a name we give to a cut that causes the node above it to cut because it was already marked, which causes the node above it to be cut since it too was already marked. This can increase the actual cost of the operation to O(# of nodes already marked). Every cost we incur from having to update pointers due to a marked node that was cut is offset by the decrease in the potential function when that previously marked node is now left unmarked in the root list.

More formally, let c be the number of nodes cut. The amount of real work done is $O(c)$. We clear $c - 1$ mark bits as we cascade up, but the parent of the last node that we cut may be marked. Also, we added c nodes to the root list. Thus, the change in potential is at most $k \cdot (c + 2(- (c - 1) + 1)) = -k \cdot (c -$

4), and the amortized cost is $(O(c) - k \cdot c) + O(1)$. As with *delete-min*, if we set k to be greater than the constant term hidden in the $O(c)$ notation, we're left with $O(1)$. This analysis also explains why we needed to use twice the number of marked bits in our potential function to cancel the addition of the nodes onto the root list.

4.3.6 Tree size

The only thing left to prove is that for every node in every tree in our Fibonacci heap, the number of descendants of that node is exponential to the number of children of that node, and this is true despite the cut rule for marked bits. We must prove this in order to substantiate our earlier assertion that all nodes have degree $\leq \log n$.

Consider the children of some node x in the order in which they were added to x.

Theorem 4.3.1 The i^{th} child to be added to x has rank at least $i - 2$.

Proof. Let y be the i^{th} child to be added to x. When it was added, y had at least $i - 1$ children. This is true because we can currently see $i - 1$ children that were added earlier, so they were there at the time of y's addition. This means that y had at least $i - 1$ children at the time of it merger because we only merge equal ranked nodes. Since a node could not lose more than one child without being cut, y must have at least $i - 2$ children ($i - 1$ from when it was added, and no more than a potential 1 subsequently lost). ▪

Note If we had been working with a binomial tree, the appropriate lemma would have been $rank = i - 1$ not $\geq i - 2$.

Let S_k be the minimum number of descendants of a node with k children. We have $S_0 = 1$, $S_1 = 2$ and, $S_k \geq \sum_{i=0}^{k-2} S_i$ This recurrence is solved by $S_k \geq F_{k+2}$, the $(k + 2)^{th}$ Fibonacci number.

4.4 Tries

The trie is a oldest known data structure, which implements set of words very efficiently report the set membership queries. Other than string membership search, the trie data structure is widely used in modern applications such as lexicon implementation, spelling correction, auto complete operations, and routing tables. Figure 4.9 depicts a trie.

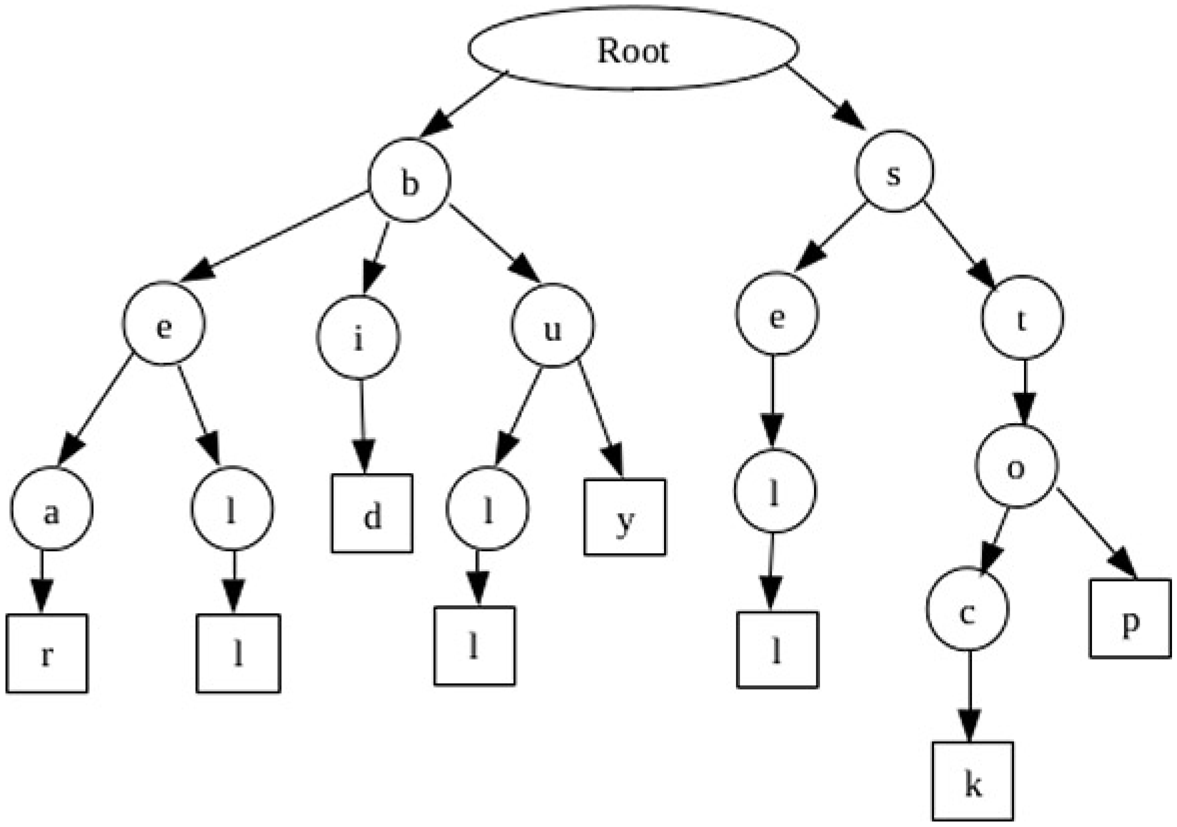


Figure 4.9: Trie

Consider a set $S = \{w_1, w_2, \ldots, w_n\}$ and query element w_q, where m =max $\{|w_1|, |w_2|, \ldots, |w_n|\}$ is the maximum length of the word. The problem is to verify the membership of query element in the collection i.e., to verify whether $w_q \in S$ or $w_q \notin S$. This simple problem can be solved by using many data structures. Let's consider the two best data structures $O(1)$-hashing and $O(\log n)$-bst and compare them with trie as shown in Table 4.1.

Data Structure	Create	Insert/Delete/ Search	Space usage	Storage location
$O(1)$-Hashing	$O(m \times n)$	$O(m)$	$O(k \times m)^*$	Random
$O(\log n)$-BST	$O(m \times n \log n)$	$O(m \log n)$	$O(m \times n \log n)$	Order
Trie	$O(m \times n)$	$O(m)$	$O(m \times n)$	Order

Table 4.1: Comparison of data structures for membership query

Set membership verification from a collection of data items is a classical problem in computer science and lots of data structures are developed with a common objective to reduce the query reporting time. However, finding a word from a collection of words can be done using a special data structure popularly known as TRIE or simply trie. Though the traditional data structures like BST or hashing are very good for searching but in case of string trie perform better, a comparison is shown in the table above.

The following are the main advantages of tries over binary search trees (BSTs). If we store keys in a binary search tree, a well balanced BST will need time proportional to $m\log n$, where m is maximum string length and n is number of keys in a tree. Using trie, we can search the key in $O(m)$ time, i.e., lookup time is faster. Looking up a key of length m takes worst case $O(m)$ time. A BST performs $O(\log (n))$ comparisons of keys, where n is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes $O(m\log n)$ time. Moreover, in the worst case $\log (n)$ will approach m.

Tries are more space-efficient when they contain a large number of short keys, since nodes are shared between keys with common initial subsequences. Tries facilitate longest-prefix matching. The following are the main advantages of tries over hash tables: Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their indices they they become full – a very expensive operation. Tries therefore have much better bounded worst-case time costs, which is important for latency-sensitive programs. Tries support ordered iteration, whereas iteration over a hash table will result in a pseudo-random order given by the hash function and there is no problem of collision even. Tries facilitate longest-prefix matching, but hashing does not.

The word trie is an infix of the word “retrieval” because the trie can find a single word in a collection of words with only a prefix of the word. The main idea of the trie data structure consists of the following parts: The trie is a tree where each vertex represents a single word or a prefix. The root represents an empty string (””), the vertices that are direct sons of the root represent prefixes of length 1, the vertices that are 2 edges of distance from the root represent prefixes of length 2, the vertices that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that is k edges of distance of the root has an associated prefix of length k.

4.4.1 Insertion

Inserting a key into trie is simple. Every character of input key is inserted as an individual trie node. Note that the children constitute an array of pointers to next level trie nodes. The key character acts as an index into the array of children. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is a prefix of an existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth. Algorithm 22 details the steps or inserting a word in a trie.

Algorithm 22 Algorithm to insert word in trie

```
1: procedure TRIE-INSERT (word)        ⊳ Insert a word in trie.
2.     Set pointer to root
3:     while word length ≥ 0 do repeat over character
4:     if character not exist in childs then insert new child for the character
5:         move to next character
6:         move to child
7:     Set end flag
8:     return New trie
```

4.4.2 Searching

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the value field of last node is non-zero, then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie. Algorithm 23 lists the search steps.

Algorithm 23 Algorithm to search word in trie

```
1: procedure TRIE-SEARCH word          ▷Search a word in a trie.
2:     Set pointer to root
3:     While word length ≥ 0 do repeat over character
4:         if character not exist in children then return not exist
5:         else move to next character
6:             move to child
7:     if Current character is end flag then return word found
8:     else return not exist
```

4.4.3 Deletion

During delete operation we delete the key in two phases using a stack. In the first phase we search for the word to be deleted and store the characters appearing in the search path. If the search is successful, we delete characters up to the previous end flag (Algorithm 24).

Algorithm 24 Algorithm to delete word from a trie

```
1: procedure TRIE-DELETE word          ▷ Delete a word from a trie.
2:     Set pointer to root
3:     while word length ≥ 0 do repeat over character
4:         push character in a stack
5:         if character not exist in childes then return not exist
6:         else move to next character
7:             move to child
8:     if Current character is end flag then stop the stack
```

```
9:          while character $\neq$ end flag do pop from stack
10:             remove character from trie
11:         return deleted
12:     else return not exist
```

4.4.4 Complexity

Insert and search operations costs equal O(key length). The memory requirement of a trie is O(alphabet size * key length * N) where N is the number of keys in trie. There are efficient representations of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize memory requirements. What makes the trie really perform well is the fixed cost of looking up a word of prefix; the cost depends only on the number of characters in a word, not on the size of the vocabulary.

Note We must take into account the worst case timing first and later convince ourselves of the practical timings. For every node in a trie, we have a collection that may be either a set or a list. If we choose the set option, the order of the operation will be in $O(1)$ time. If we use a linked list, the worst number of comparison will be 26 (the number of letters in the English alphabet). To move from node to another, at least 26 comparisons are required at each step.

Complexity 4.4.1 — Lexicon. To insert a word of length k, we need k * 26 comparisons. Applying the O notationy ields $O(k)$ which will become $O(1)$. Insert operations are performed in constant time regardless of the length of the input string (this might look lik an understatement, but if we make the length of the input string a worst case maximum, this sentence holds true). Same holds true for the search operation as well. The search operation exactly performs the way the insert does and its order is $O(k*26) = O(1)$.

Complexity 4.4.2 — Space requirement. We will use an English dictionary as an example to illustrate complexity. Considering space requirements (recall that *M* indicates the byte size of the dictionary), a trie could have *M* nodes in the worst case, if no two strings shared a prefix. A lot of compression must be done if we observe a large amount of redundancy in the dictionary. The English dictionary used in the example contains 935,017 bytes and requires 250,264 nodes, with a compression ratio of about 73%. However, despite the compression, a trie will usually require more memory than a tree array because each node requires at least 26 * sizes of pointer bytes plus overhead. On a 64-bit machine, each node requires more than 200 bytes, whereas a string character requires a single byte, or two if we consider UTF strings.

4.4.5 Compact trie

Consider a trie that is mostly static (all insertions or deletions of keys from a prefilled trie are disabled) and only lookups are needed. It is possible to compress the trie representation by merging the common branches to compress the representation. We can also improve the time and space performance metrics of a trie by eliminating all branch nodes that have only one child. The result is called a compressed trie. When branch nodes with a single child are removed from a trie, we need to keep additional information so that dictionary operations may be performed correctly.

4.4.6 Patricia

A Patricia is a compressed trie. Instead of storing all data at its edges and having empty internal nodes, Patricia stores data in every node. All operations are performed at worst in O(K) time, where K is the number of bits in the largest item in the tree. In practice, operations actually take O(A(K)) time, where A(K) is the average number of bits of all items in the tree. Most importantly, Patricia requires very few comparisons to keys during any operation. During a lookup, each comparison (K at most) will perform a single bit comparison against the given key, instead of comparing the entire key to another key. Patricia, edge labels (from, to) in a compressed operation are replaced by (T[from], to from + 1). To implement this idea we make a copy of each string s that is inserted into *T*. To label an edge with some substring of s, we use two pointers to the first and last characters of the label, thus ignoring the cost of storing extra strings. The size of each edge label is constant. Additional Patricia operations include (1) finding all strings with a common prefix; the system returns an array of strings that begin with the same prefix; (2) finding predecessors by locating the largest string that is smaller than the given string using lexicographic order; and (3) finding successors by locating the smallest string larger than the given string, also via lexicographic order.

4.4.7 Suffix tree

Let S denote a string, the length of which is n. Let $S[i, j]$ denote the substring of S from position i to position j. Before constructing the suffix tree, we concatenate a new character, \$ to S. The importance of this character is twofold. First, by adding it to the string, we avoid the undesirable situation in which a suffix is a prefix of another suffix, which is undesirable. Second, the generalization is also made easier by this operation. Now, we will define the suffix tree of a string. We also consider fixed size alphabets; unbounded alphabets are not discussed. A suffix tree is a rooted, directed tree (see Figure 4.10). It has n leaves labelled from 1 to n, and its edges are labelled by characters of the alphabet. The label of an edge e is denoted by $l(e)$. On a path from the root to the leaf j one can read the suffix $S[j, n]$ of the string and a \$ sign. Such a path is denoted by $P(j)$ and its label is referred as the path label of j and denoted by $L(j)$. We call a leaf w reachable from the node v, if there is a directed path from v to w.

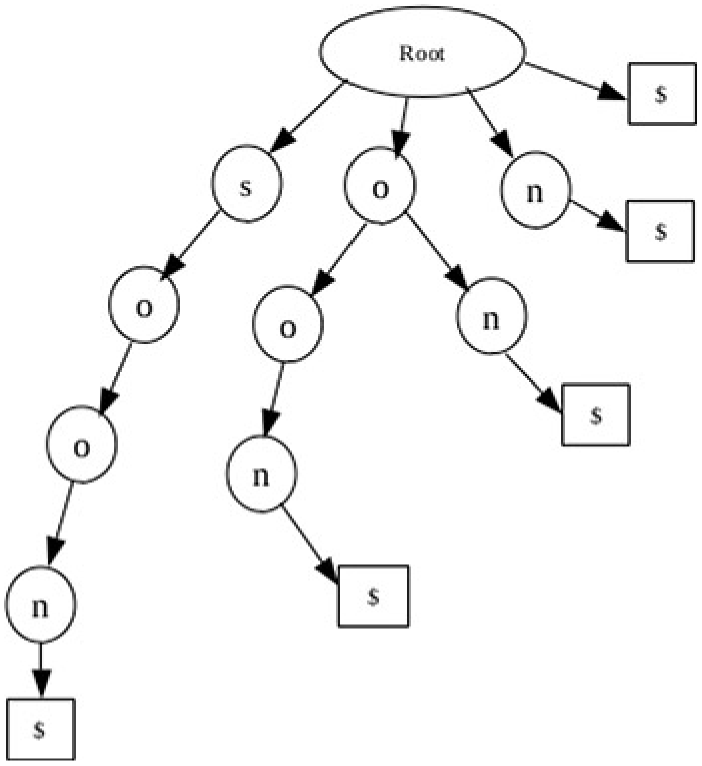


Figure 4.10: Suffix tree example (in reverse dictionary order)

Suffix trees can use quite a lot of space. There are long branches which could be compressed to achieve a compact suffix tree of a string. Formally, a compact suffix tree of *S* is a rooted directed tree with *n* leaves. Each internal node has at least two children (the root is not an internal node). Each edge has a label with the property that if *uv* and *uw* are edges, then the first characters of the label of *uv* and of *uw* are distinct. The label of a path is the concatenation of the labels on its edges.

4.5 Inverted Index

An inverted index data structure stores mappings from content such as words or numbers in a database, document, or set of documents (unlike a forward index that maps from documents to content). The purpose of an inverted index is to allow fastfull text searches, at a cost of increased processing when a document is added to the database. The inverted file may be the database file rather than itsindex. It is the most popular data structure used indocument retrieval systemsused on a large scale for example insearch engines.

4.5.1 Inverted index creation

In its basic form, an inverted index consists of postings lists, one associated with each term that appears in the collection. A postings list is comprised of individual postings, each of which consists of a document identification and payload information about occurrences of the term in the document. The simplest payload is nothing! For simple Boolean retrieval, no additional information is needed in the posting other than the document identification; the existence of the posting indicates the presence of the term in the document.

▪ **Example 4.3** Simple illustration of an inverted index. Each term is associated with a list of postings. Each posting is comprised of a document identification and a payload, denoted by *p*. An inverted index provides quick access to documents identifications that contain a term.

- term1 occurs in d1, d5, d6, d11,........,
- term2 occurs in d11, d23, d59, d84,....., and
- term3 occurs in d1, d4, d11, d19,.......

In an actual implementation, we assume that documents can be identified by a unique integer ranging from 1 to n, where n is the total number of documents. Generally, postings are sorted by document identification.

terms	postings
term1	d1p d5p d6p d11
term2	d11 p d23 p d59 p d84 p …
term3	d1 p d4 p d11 p d19 p …

▪

The size of an inverted index varies, depending on the payload stored in each posting. If only term frequency is stored, a well-optimized inverted index can be a tenth of the size of the original document collection. An inverted index that stores positional information would easily be several times larger than one that does not. Generally, it is possible to hold the entire vocabulary (i.e., dictionary of all the terms) in memory, especially with techniques such as front-coding. However, with the exception of well-resourced, commercial web search engines, postings lists are usually too large to store in memory and must be held on disk, usually in compressed form; this involves random disk access and decoding of postings. One important aspect of the retrieval problem is to organize disk operations such that random seeks are minimized.

4.5.2 Index compression

We return to the question of how postings are actually compressed and stored on disk. This chapter devotes a substantial amount of space to this topic because index compression is one of the main differences between a "toy" indexer and one that works on real-world collections. Generating or maintaining a large-scale search engine index represents a significant storage and processing challenge. Many search engines utilize a form of compression to reduce the size of the indices on disk.

▪ **Example 4.4** Consider the following scenario for a full text, Internet search engine.

- An estimated 2,000,000,000 different web pages existed as of the year 2000.
- Suppose there are 250 words on each webpage (based on the assumption they are similar to the pages of a novel.
- It takes 8 bits (or 1 byte) to store a single character. Some encodings use 2 bytes per character.
- The average number of characters in any given word on a page may be estimated at 5.
- The average personal computer comes with 100 to 250 gigabytes of usable space.

Given this scenario, an uncompressed index (assuming a non-conflated, simple, index) for 2 billion web pages would need to store 500 billion word entries. At 1 byte per character, or 5 bytes per word, this would require 2500 gigabytes of storage space alone, more than the average free disk space of 25 personal computers. This space requirement may be even larger for a fault-tolerant distributed storage architecture. ▪

4.5.3 Key words search

The point of using an index is to increase the speed and efficiency of searches of the document collection. Without some sort of index, a users query must sequentially scan the complete document collection, finding those documents containing the search terms. Consider the “Find” operation in Windows; a user search is initiated and a search starts through each file on the hard disk. When a directory is encountered, the search continues through each directory. With only a few thousand files on a typical laptop, a typical find operation takes a minute or longer. Currently, a web search covers at least one billion documents. Hence, a sequential scan is simply not feasible. Within the search engine domain, data are searched far more frequently than they are updated. An inverted index is able to do many accesses in $O(1)$ time at a price of significantly longer time to do an update, in the worst case $O(n)$. Where other data structures require a minimum of $O(\log n)$ time to perform any operation; best results were produced from balanced developed to improve database access. For many systems, the inverted index can be compressed to around 10% of the original document collection.

4.6 Exercises

Exercise 4.1 Consider an algorithm that computes the connected components of an undirected graph G using a forest of trees and union-find: Start with a partition of the n vertices into a forest of n trees, each consisting of a single vertex. Then, for each edge (i, j) in the graph G, apply union(i, j). Prove that this algorithm is correct, in that it indeed computes the connected components of G. ▪

Exercise 4.2 Consider the set of all trees of height h that can be constructed by a sequence of union-by-height operations. How many such trees are there? ▪

Exercise 4.3 Consider an arbitrary sequence of m makeset operations, followed by u union operations, followed by f find operations, and let $n = m + u + f$. Prove that if we use union by rank and find with path compression, all n operations are executed in $O(n)$ time. ▪

Exercise 4.4 Give an example of two binary heaps with n elements each such that build-heap takes (n) time on the concatenation of their arrays. ▪

Exercise 4.5 Discuss the relationship between inserting into a binomial heap and incrementing a binary number and the relationship between uniting two binomial heaps and adding two binary numbers. ▪

Exercise 4.6 Show that if only the mergeable-heap operations are supported, the maximum degree $D(n)$ in an n-node Fibonacci heap is at most $\lfloor \log n \rfloor$. ▪

Exercise 4.7 Given two strings S_1 and S_2, and a positive integer k, find the number of substrings of S_1 of length at least k that occur in S_2. Develop and analyze an algorithm to solve this problem in $O(|S_1| + |S_2| + \text{sort}(\Sigma))$ time. ▪

Exercise 4.8 Create a suffix tree for Hizbizbiz. ▪

Exercise 4.9 Create a trie, compact trie and Patricia for the word searching from a file titled “Education news covers the latest national and international education news”. ▪

Exercise 4.10 Describe inverted index data structure for text searching from a collection of documents. Compute construction cost and search cost of an inverted index. Construct an inverted index for the following collection.

- Doc1: breakthrough drug for schizophrenia
- Doc2: new schizophrenia drug
- Doc3: new approaches for treatment of schizophrenia
- Doc4: new hopes for schizophrenia patients ▪

Part II

Evolving Paradigms

Chapter 5

Evolving Paradigms of Data Structures

In Part II presents structures of various data domains in a problem-specific arrangement. Not all data structures work in all domains because data is domain-specific and query requirements differ. The chapters (5 through 10) of Part II describe various data structures: spatial, temporal, external memory, distributed, and synopsis types. All six chapters conclude with exercises.

This chapter details evolving paradigms and discusses topics of current interest. This chapter is relevant because numbers, sizes, and complexities of data structures continue to increase. This chapter focuses on geometric queries and the complexities of input and output operations and communications. The final section tackles large data problems.

Chapter 6 introduces data structures designed to handle spatial (multidimensional) data. It starts by explaining range queries and suggests four popular solutions: range search trees, KD trees, quadtrees and R trees. Theoretical bounds, examples, and construction details are explained.

Chapter 7 explores lesser known temporal data structures: partial, full, confluent, and functional persistent types. Retroactivity issues are also covered. Chapter 8 describes external memory models designed to reduce input and output operations related to queries. Cache aware and cache oblivious models are explained as are (a,b), B, B+, and buffer trees. The chapter includes theorems and examples.

Chapter 9 focuses on distributed data structures and the difficulties of implementing them, then explains distributed hashing, distributed lists and trees, and skip graphs. Theoretical details and recent developments are described. Chapter 10 covers synopsis data structures designed to deal with large amounts of streaming data. It includes definitions and notations. Specific sections discuss sampling, sketching, fingerprints, and wavelets.

5.1 Geometric Queries

All the data structures we studied in previous chapters were developed to handle one-dimensional data points. However, data points are not always one-dimensional in practice. The ability to partition search space is a common feature of most data structures. Most data structures are designed to perform point search (linear space) queries. The common partitioning strategy is dividing linear spaces around points into two halves. Instead of designing a new data structure for higher dimensional spaces, we developed a space partitioning device to transform points in linear space into two or more dimensions. Note that curves and other geometric shapes can be used for space partioning, but they are difficult to understand.

Orthogonal range queries are the most basic geometric inquiries. They are different from point search queries and are helpful for developing range search solutions and more advanced databases. Let $S \subseteq R^d$, where $|S| = n$ and R^d is a d dimensional real space having Cartesian coordinate system, i.e. S is a set of n points of d dimensional space. Assume that Q is a query box (hyper cube) in same d dimensional space whose sides are parallel to the axis of R^d. In the range search problem, we are supposed to return the points of $S \cap Q$ efficiently. In two dimensional case, Q is an interval say, $[q_1, q_2]$ and the points of S which are $\geq q_2$ and $\leq q_1$ should be reported.

▪ **Example 5.1** Range searching arises in many applications.

- A university administrator may wish to know students whose ages are between 21 and 24 years and whose grade point averages are greater than 3.5.
- In a geographic database of cities one might seek a list of cities whose latitude lies between 37′ and 41′ and longitude between 102′ and 109′.
- In data analysis it is often useful to do separate analyses on sets of data lying in different regions (ranges) of the observation space and

then compare (or contrast) the respective results.

- In statistics, range searching can be employed to determine the empirical probability content of a hyperrectangle, to determine empirical cumulative distributions, and to perform density estimation.

▪

5.2 I/O Complexities

The efficiencies of algorithms and data structure are generally measured in units of number of operations based on the assumption that the whole data structure is stored in memory. But for large data structures not all of the data can be stored in cache; the portions of the data structure must be stored in external memory, which is usually a disk. Accessing data from the disk is a slower operation than data in cache and measures; as units of time. Our objective is minimizing the number of disk transfers for creation updates and query reporting of data structure. Since accessing the external memory is so much slower than accessing RAM, the total number of block transfers between internal memory and external memory are only considered and the computations performed within the internal memory are assumed free. The notions of complexity we use to compute are meaningful but disk access is too slow to make a program usable. Analyzing the performance of an ADT by counting the number of disk transfers is called I/O complexity.

▪ **Example 5.2 — Systems log file.** A log file holds a tremendous number of system commands. Every record stores the detailed information of the program and its execution behavior. The huge number of records are arranged into a large file, stored in the disk. The disk writing process is formed block by block. When a new record must be stored, the last used block is checked for empty slots; a new block is initiated if the last block is filled. To access a stored record we need to load the particular block in the memory. When storing an unordered list, the $O(n/B)$ blocks are searched; n is the number of nodes and B is block size. We can say that I/O complexity is $O(n/B)$. Using a ordered list can reduce this I/O complexity of searching to

$O(\log n/B)$, but insertion will become $O(n/B)$. We can observed that sequential access is not good option for data stored in disk.

5.3 Communication Complexities

Definition 5.3.1 — Communication complexity. The communication complexity is a measure to quantify the total amount of communication made by an algorithm that executes on different systems over a network.

Communication complexity was introduced by Yao in 1979 (4) who described complexity as communication between multiple parties. Assume A and B are two parties; A possesses D_1 data and B possesses D_2 data. Our task is to compute a function using both sets of data, $f(D_1, D_2)$. Computation can be done anywhere at the location of A or B. We choose to compute at A. If B sends all data, A can compute $f(D_1, D_2)$ trivially, but the goal of communication complexity is to find an efficient solution: to compute a task with the least amount of communication between the parties.

▪ **Example 5.3 — Simple statistics.** Assume that we need to compute the average of a large collection D of n integers. The data resides with A and B. A possesses D_1 consisting of n_1 integers and B possesses D_2 consisting of n_2 integers, where $n_1 + n_2 = n$. Assume further that the task will be computed at A. A trivial solution will send all B's data to A, requiring (size of D_2)/(packet size) communication and computation of the average at A. A better solution is choosing the computation point wisely, i.e., node with smaller data will send. An even better solution is having B send the average and number of nodes to A and having A compute the joint average. ▪

5.4 Large Data Problem

The rapid growth of data size in various application domains demands more efficient data structures capable of processing petabytes and continuous sensor data. Large volumes of data that reside in disks or arrive continuously over a network are not accessible randomly in multiple passes. Processing of these kind of data using some data structure allows user to scan the data or part of it only once through a small window. Main challenge of the researchers is to minimize the access of the data and still allow the data structure to answer the desired query with some degree of guarantee. For static data *S*, residing in the disks and a class of queries *Q*, the goal is to develop a data structure for the query class *Q* that minimizes the reporting time as well as maximizes the accuracy and confidence of the answers. In case of dynamic data structure, data arriving online are stored in disks, then used to create and update data structures.

To address the above problem we need data structures with small footprints, popularly known as synopsis data structures which are substantively smaller than their base data sets. These are data structures for supporting queries to massive data sets while minimizing or avoiding disk access. They have the following advantages:

1. Fast processing: May reside in memory; provides fast processing of queries and updates itself.
2. Fast transfer: Resides in disks; can be swapped in and out of memory with minimal disk access.
3. Lower cost: Has minimal impact on space requirements of the data set and its supporting data structures.
4. Small surrogate: Can provide surrogate function for data set when the data set is currently expensive or impossible to access.

Since synopsis data structures are too small to maintain a full characterization of their base data sets, they must summarize the data set, and the responses they provide to queries will typically be approximate.

▪ **Example 5.4** An important application domain for synopsis data structure is approximate query answering for ad hoc queries of large data warehouses. In large data recording and warehousing environments, it is often advantageous to provide fast, approximated answers to complex decision support queries. The goal is to provide an estimated response in far less time than the time required to compute an exact answer by minimizing or eliminating the number of accesses to the base data. ▪

5.5 Exercise

Exercise 5.1 How can you search for all the points laying inside a circle? ▪

Exercise 5.2 How can you retrieve a previous edit in an image? ▪

Exercise 5.3 How do you multiply two matrices, each of which is larger than the computer memory? ▪

Exercise 5.4 Describe the procedure of executing a stack on a distributed environment. ▪

Exercise 5.5 How would you determine the most frequent caller on your phone since the day you purchased it? ▪

Chapter 6

Spatial Data Structures

Objective 6.1 All data structures we studied so far deal with one-dimensional data points, but real-life data are not always one-dimensional. Our primary concern in this chapter is developing efficient data structures to handle higher dimensional data points.

Note Partitioning search space is a common feature in most data structures designed to perform point search (linear space) queries. A common strategy s to divide the linear space into two halves. Rather than designing new logical steps for higher dimensional spaces, we utilize a space partitioning separator to handle two-dimensional or greater situations. Curves and other geometric shapes can partition space but they are difficult to understand.

Definition 6.0.1 — Range queries. Orthogonal range queries are basic geometric queries, unlike point search queries. They represent a fundamental step toward developing the requirements for solving range search problems. Let $S \subseteq R^d$, where $|S| = n$ and R^d is a d dimensional real space having Cartesian coordinates. S is a set of n points of d dimensional space and Q is a query box (hyper cube) in the same d dimensional space whose sides are parallel to the axis of R^d. In the range search problem, we should return the points of $S \cap Q$ efficiently. In the two-dimensional case, Q is an interval say, $[q_1, q_2]$ and the points of S which are $\geq q_2$ and $\leq q_1$ should be reported.

6.1 Range Search Trees

A range search tree on a set S of d dimensional points, is an ordered tree data structure generally implemented with balanced binary search trees in each dimension, where the data values are stored in the leaves and internal nodes keep values for comparison purposes, usually the largest value of its left subtree. A range search tree can be realized as a multi-level binary search tree on a set of points in d-dimensions, defined recursively over the dimensions as level. The first level is a binary search tree consisting of the first coordinate of each point of S. Each node x of this top level tree contains an associated range search tree data dimension created with the remaining $(d - 1)$ coordinates of the points of S stored in the subtree of x.

Definition 6.1.1 A range search tree is a binary tree data structure designed to report all points within a given range of multi-dimensional space.

Bentley introduced range trees in 1979 [50] to provide $O(\log^d n + k)$ times range query processing with $O(n \log^{d-1} n)$ space for n points of d space, reporting K points. Bernard Chazelle improved complexity capacity by achieving query time $O(\log^{d-1} n + k)$ and space requirement $O\left(n\left(\frac{\log n}{\log\log n}\right)^{d-1}\right)$.

6.1.1 Construction

A one-dimensional range search tree of n points is a simple binary search procedure that takes $O(n \log n)$ time to be constructed. A multi-dimensional range search tree is constructed recursively by implementing a balanced binary search tree on the first coordinate of the points, followed by construction of a $(d - 1)$-dimensional range search tree for the remaining coordinates contained in the subtree of each node recursively till last coordinate. This recursive construction of a range search tree requires. See Figure 6.1.

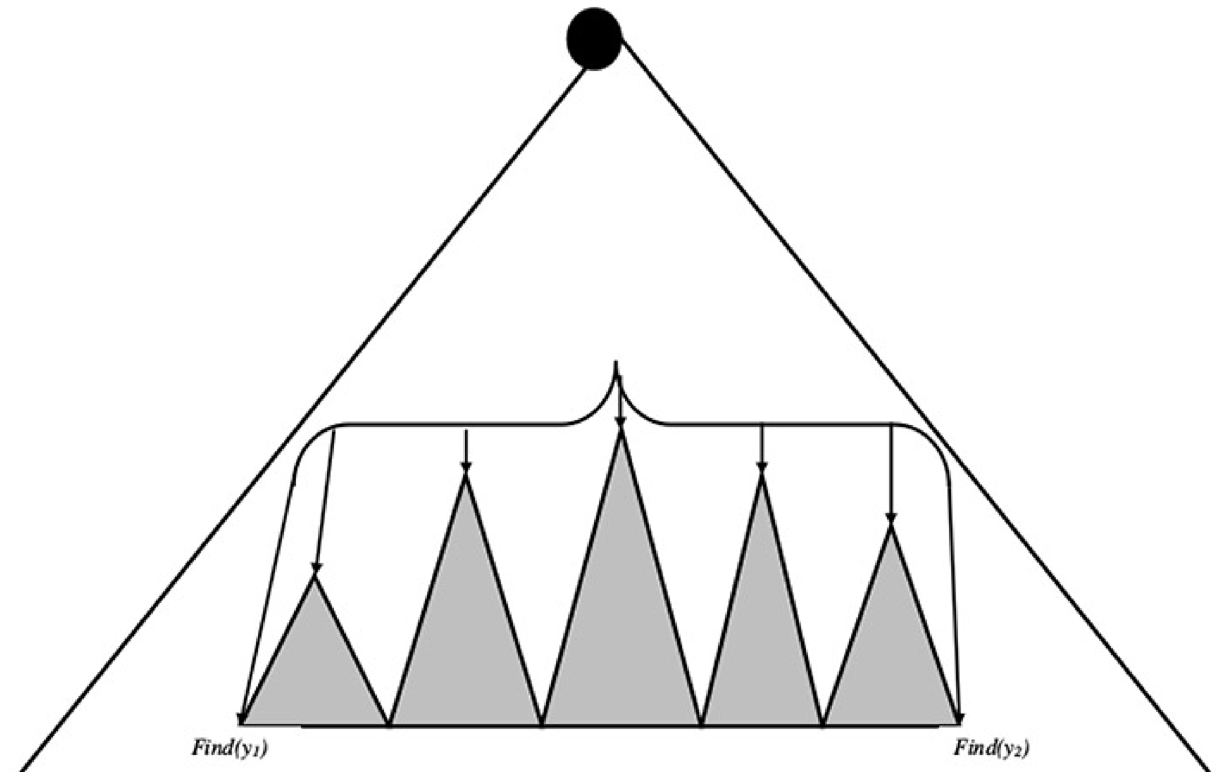


Figure 6.1: Range search tree

6.1.1.1 Two-dimensional range search tree in $O(n \log n)$

Let $S \subseteq R^2$.
If S is singleton then return a leaf with value of that point.
Otherwise, construct a one-dimensional range tree based on the y coordinates for $(x, y) \in S$.
Let $x_m = \{\text{median of all x}\}$.
Let $S_l \subseteq S$ s.t. $x \leq x_m \forall (x, y) \in S_l$.
Let $S_r \subseteq S$ s.t. $x > x_m \forall (x, y) \in S_r$.
Repeat above steps for S_l.
Repeat above steps for S_r.

Recursive equation of complexity is $T(n) = 2T(n/2) + n$, which resolves into $T(n) = O(n \log n)$. Time to construct is $O(n \log^{2-1} n) = O(n \log n)$.

6.1.2 Range query search

A range query Q on a range search tree T_R containing points of $S \subset R^d$, reports the set of points $S \cap Q$. For $d = 1$ Q is an interval, say $[x_1, x_2]$. To perform this range query, we first execute two point queries for the points x_1 and x_2 and then follow the steps below:

```
Let $x_{12}$ be the last node in the common search path of $x_1$ and $x_2$.
For every node $x$ after $x_{12}$ in the search path of $x_1$ do.
If $x \geq x_1$, report every point in the right-subtree of $x$ including $x$.
For every node $x'$ after $x_{12}$ in the search path of $x_2$ do.
If $x' \leq x_2$, report every point in the left-subtree of $x'$ including $x'$.
Return union of points.
```

A range search tree is a always a balanced binary search tree, so the maximum length of search paths for any point is $O(\log n)$. Reporting the points of a subtree is linear, the time of traversal. Therefore the time required to perform a range query is $O(\log n + k)$, where $k = |S \cap Q|$, the number of points inside the query interval.

A range query in d-dimensions performs in a recursive matter on the remaining structure and stops recursion via the above steps when the remaining dimension is 1. The recursion works as a d-dimensional range query reducing to $O(\log n)$ search and a $(d - 1)$-dimensional range query, which achieves a complexity of $O(n \log^d n + k)$.

Note This complexity can be reduced to $O(n \log^{d-1} n + k)$ using fractional cascading.

6.2 KD Trees

Definition 6.2.1 A k-dimensional tree, popularly known as KD tree is a geometric data structure widely used for organizing multi-dimensional points into an upgraded binary search tree capable of searching over dimensions alternatively while processing spatial search queries.

The KD tree is the most popular spatial data structure for performing range search queries and nearest neighbor queries. Each level of the tree partitions all children along a specific dimension and rotates the dimension over levels using a hyperplane perpendicular to the axis of consideration. At the root of the tree all points will be partitioned based on the first coordinate of the root node, i.e. all points of the right subtree have larger first coordinates than the roots have. Each level down the tree partitions the search space on the basis of next dimension with a periodical repetition, i.e. returning to the first dimension when the last one is finished. The efficient way to build a static KD tree is to use a partition method that places the median point; all data with smaller one-dimensional values moves to the right, and larger amounts move to the right. We then repeat this procedure recursively on both the left and right subtrees until single element remains. See Figure 6.2.

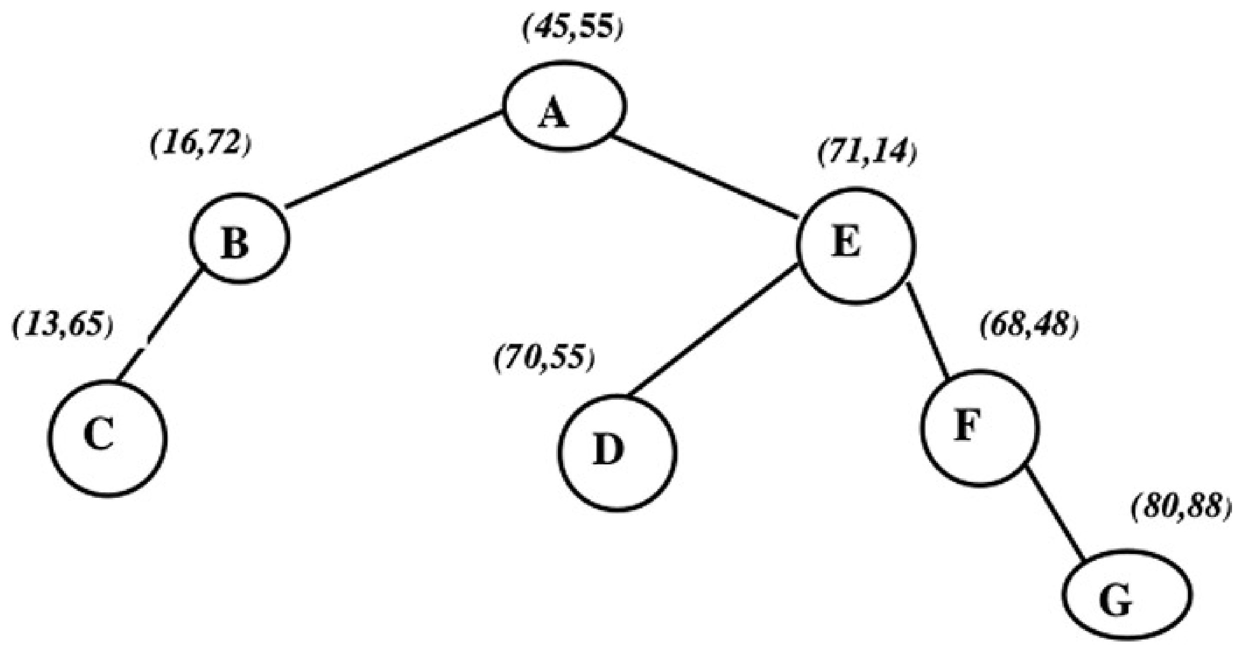


Figure 6.2: Simple KD tree example

Note Search space partitioning strategy used here is uses line separators in two dimensions and hyperplane separators for hyperspace, partitioning dimensions one after one alternatively.

6.2.1 Creation of KD tree

Since there are many possible ways to choose axis-aligned splitting planes, there are many different ways to construct KD trees. First, we will understand the creation method of a static KD tree before proceeding to dynamic updates.

6.2.1.1 Construction of static KD tree

Step 1: Let $S \subset R^d$ be a set of n points of d dimensional space, i.e. $S = \{x_1, x_2, \ldots, x_i, \ldots, x_n\}$, where each $x_i = (x_i^1, x_i^2, \ldots, x_i^d)$.

Step 2: The first partition of the data space takes place at the root of static KD tree using the hyperplane $x^1 = x_m^1$, where x_m^1 is the median of $\{x_1^1, x_2^1, \ldots, x_i^1, \ldots, x_n^1\}$.

Step 3: The left subtree of the root will contain all the points on the left side of the hyperplane $x^1 = x_m^1$, i.e. $x_i^1 \leq x_m^1$.

Step 4: Similarly, right subtree of the root will contain all the points lying to the right of the hyperplane $x^1 = x_m^1$, i.e. $x_j^1 > x_m^1$.

Step 5: Now, we have two disjoint subsets of point set, $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \phi$, where S_1 contains all the points satisfying $x_i^1 \leq x_m^1$ and S_2 contains all the points satisfying $x_j^1 > x_m^1$.

Step 6: The next level partitions of the data space take place at the children of the roots using the hyperplanes $x^2|_{S_1} = x_m^2|_{S_1}$, where $x_m^2|_{S_1}$ is the median of $\{x_1^2, x_2^2, \ldots, x_i^2, \ldots, x_n^2\} \cap S_1$, in the left child and $x^2|_{S_2} = x_m^2|_{S_2}$, where $x_m^2|_{S_2}$ is the median of $\{x_1^2, x_2^2, \ldots, x_i^2, \ldots, x_n^2\} \cap S_2$, in the right child.

Step 7: The process continues until the subsets become singletons and are inserted into the corresponding leaves when further partition is no longer required.

Note The above method creates a balanced static KD tree.

Complexity 6.2.1 The above recursive algorithm will generate the recursive equation $T(n) = 2T(n/2) + n \log n$. Solving this recursion we obtain $T(n) = \Omega(n \log n)$.

6.2.1.2 Point query in a KD tree

The point of searching with a KD tree is analogous to searching in other types of trees. For searching a query point $x_q = (x_q^1, x_i^q, \ldots, x_q^d)$ in a KD tree, we start the comparison of x_q^1 with the first coordinate of root, followed by the comparison of x_q^2 with the second coordinate of selected child of root and so on. Once the search process reaches the leaf, we decide whether the point exists in the tree or not based on an exact match or failure. The complexity of search process is $O(\log n)$ for a balanced KD tree consisting of n nodes.

6.2.1.3 Dynamic updates in a KD tree

Inserting a node in an empty KD tree is trivial, as it is for other data structures: create a root and insert the node. Inserting a new node in an existing KD tree is also similar; the first step is running a point search to find the location. The search process is coordinate wise of the point as well as level wise of the tree, as explained in the above paragraph. Once the search operation reach the leaf and element not already present in the tree, the new node is inserted based on comparison of next coordinates of current node and new node either as left or right child depending on the result of comparison.

Deleting a given node is similar to deletion processes for other tree structures. The first step is determining the element to be deleted. If the node is a leaf, simply delete it. If the element to be deleted is an internal node, the operation is more complex.

Note: Spatial data structures are generally created for large numbers of data points; lazy deletes are preferred to sequential deletes because they perform more efficiently.

? What about maintaining a balanced KD tree?

Note Dynamic updates may destroy the balanced structures of KD trees. Rebalancing is not a preferred option even though an unbalanced tree make increase search cost. A randomized KD tree is a logical alternative but it requires random shuffling during insertion.

Complexity 6.2.2 Inserting a new element in a balanced kd-tree takes $O(log\ n)$ time. Removing a given element from a balanced kd-tree also takes $O(log\ n)$ time. Querying a range parallel to the axes in a balanced kd-tree takes $O(n^{1-1/k} + m)$ time, where m is the number of the points to be reported in k^{th} dimensional points.

6.2.2 Range search in KD tree

Range searches in KD trees find all points in the circle of a radius R of a query q, typically achieved by using a stack. Starting from a root, we put all the nodes and their children arriving in the search path into the stack for further processing. When the search operation reaches the leaf, we compute the distances of query points in the stack by removing them from the top down. If the computed distance is less than R the point is accepted for range reporting.

? What to do with the node lying on the boundary?

We must consider all the points of the subtree rooted at that particular child node for inclusion in the stack followed by a range verification for a possible reporting.

Note Inclusion of a child node in a subtree is very expensive but the step must be performed to ensure accurate range reporting.

Note If a KD tree uses split hyperlanes parallel to an axis, inclusion of a subtree with a child node is made easier by subtraction of splitting coordinate of child node and query point.

Complexity 6.2.3 The worst case time complexity for range search query in a k-d tree containing n nodes is $t_{\text{worst}} = O(k \cdot n^{1-\frac{1}{k}})$. The curse of dimensionality leads most searches in high dimensional spaces to end up as brute force searches.

6.2.3 Nearest neighbor search in KD tree

A nearest neighbor search in a KD tree is similar to a range search except that the radius R is treated as a current minimum and rejection criterion rather than as an accepted condition. The nearest neighbor search is performed by using a stack. We start from the root and put all nodes and their children in the search path into the stack for selection as possible nearest neighbors sought by the query. After the search reaches the leaf, we compute the distances of query points starting by removing the points in the stack from the top down. The first distance computed between leaf and query point is considered as current minimum R and the leaf is considered the current nearest neighbor. We then compute the distances from the query point for all other points extracted from the top of the stack. If the computed distance is less than R the point updated as current nearest neighbor and the value of R are reduced to newly computed distance; otherwise the old parameters will remain. When the stack is empty the current nearest neighbor is reported.

Note Nearest neighbor search is very useful for real-time applications such as finding a nearby school or the nearest bus stop of restaurant.

6.3 Quadtree

A quadtree is a tree based spatial data structure in which each internal node has exactly four children. Quadtrees perform the two-dimensional space partition in each node dividing a two-dimensional point into four quadrants or regions; this process is similar to what binary search trees do in one dimension. Various methods can associate data with leaf nodes based on the requirements of the application.

The subdivided regions may be square or rectangular, or may have arbitrary shapes. This data structure was named a quadtree by Raphael Finkel and J.L. Bentley in 1974 [51].

A quadtree is a hierarchical data structure that represents spatial data based on the principle of recursive decomposition (similar to divide-and-conquer methods). Hierarchical data structures are useful because of their ability to focus on the interesting subsets of the data. This focusing results in an efficient representation and improved execution times. Thus they are particularly convenient for performing set operations. Data structures are also attractive because of their structural clarity and ease of implementation. The hierarchical representation using a tree data structure is the traditional also a way to represent quadtrees but a linear representation is more efficient and saves image storage space.

6.3.1 Inserting data into a quadtree

Starting at the root, determine which quadrant your point occupies. Recurse to that node and repeat, until you find a leaf node. Then, add your point to that node's list of points. If the list exceeds some predetermined maximum number of elements, split the node and move the points into the correct subnodes. To query a quadtree, starting at the root, examine each child node, and check if it intersects the area queried. If it does, recurse into that child node. Whenever you encounter a leaf node, examine each entry to see if it intersects with the query area, and return it if it does.

Theorem 6.3.1 Let T be a quadtree with m nodes. Then the balanced version of T has $O(m)$ nodes and can be constructed in $O((d+1)m)$ time.

6.3.2 Properties of quadtree

Quadtrees are hierarchical data structures whose common property is that they are based on the principle of decomposition of space (Figure 6.3). Their characterization is based on (1) the type of data they represent; (2) the principle guiding the decomposition process; and (3) whether the resolution is or is not variable. Currently quadtree is used for point data, areas, curves, surfaces and volumes. The prime motivation for the development of the quadtree is the desire to reduce the space necessary to store data through the use of aggregation of homogeneous blocks. An important by-product of this aggregation is the reduction of operating times of certain operations (e.g. connected component labeling and component counting).

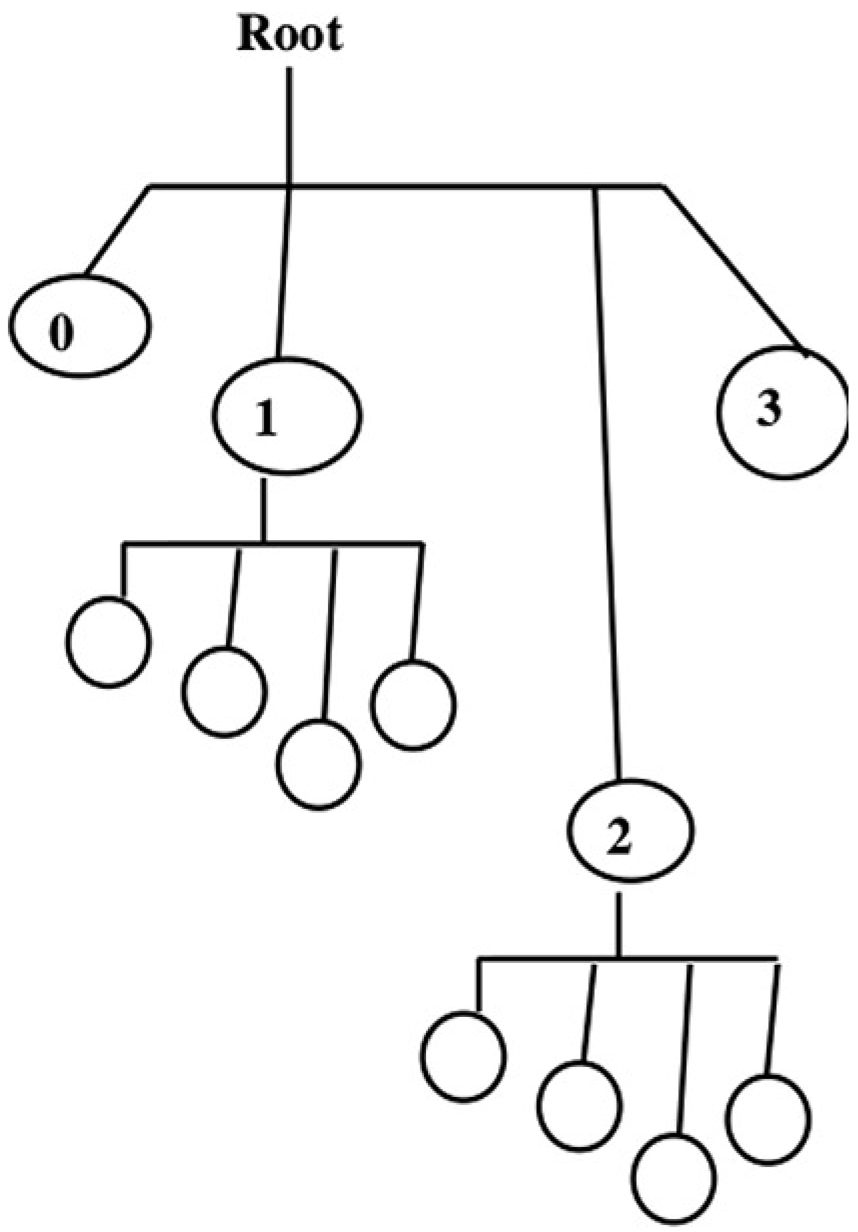


Figure 6.3: Example of quadtree

Quadtrees may be classified according to the type of data they represent, including areas, points, lines and curves. Quadtrees may also be classified by whether the shape of the tree is independent of the order in which data is processed.

6.3.3 Region quadtree

The region quadtree represents a partition of space in two dimensions by decomposing the region into four equal quadrants, subquadrants, and so on with each leaf node containing data corresponding to a specific subregion. Each node in the tree either has exactly four children, or has no children (a leaf node). The height of a quadtree that follows this decomposition strategy (i.e. subdividing subquadrants as long as there is interesting data in the subquadrant for which more refinement is desired) is sensitive to and dependent on the spatial distribution of interesting areas in the space being decomposed. The region quadtree is a type of trie.

A region quadtree with a depth of *n* may be used to represent an image consisting of $2n \times 2n$ pixels, where each pixel value is 0 or 1. The root node represents the entire image region. If the pixels in any region are not entirely 0s or 1s, it is subdivided. In this application, each leaf node represents a block of pixels that are all 0s or all 1s. Note the potential savings in terms of space when these trees are used for storing images; images often have many regions of considerable size that have the same colour value throughout. Rather than store a big two-dimensional array of every pixel in the image, a quadtree can capture the same information potentially several divisive levels higher than the pixel-resolution sized cells that we would otherwise require. The tree resolution and overall size are bounded by the pixel and image sizes.

A region quadtree may also be used as a variable resolution representation of a data field. For example, the temperatures in an area may be stored as a quadtree, with each leaf node storing the average temperature over the subregion it represents.

If a region quadtree is used to represent a set of point data (such as the latitude and longitude of a set of cities), regions are subdivided until each leaf contains at most a single point.

6.3.4 Point quadtree

The point quadtree is an adaptation of a binary tree used to represent two-dimensional point data. It shares the features of all quadtrees but is a true tree as the center of a subdivision is always on a point. It is often very efficient in comparing two-dimensional, ordered data points, usually operating in *O*(log *n*) time. Point quadtrees are worth mentioning for completeness, but they have been surpassed by KD trees as tools for generalized binary search.

Point quadtrees are constructed as follows. Given the next point to insert, we find the cell in which it lies and add it to the tree. The new point is added such that the cell that contains it is divided into quadrants by the vertical and horizontal lines that run through the point. Consequently, cells are rectangular but not necessarily square. In these trees, each node contains one of the input points.

Since the division of the plane is decided by the order of point-insertion, the tree's height is sensitive to and dependent on insertion order. Inserting in a "bad" order can lead to a tree of height linear in the number of input points (at which point it becomes a linked-list). If the point-set is static, pre-processing can be done to create a tree of balanced height.

A node of a point quadtree is similar to a node of a binary tree, with the major difference being that it has four pointers (one for each quadrant) instead of two (left and right) as in an ordinary binary tree. Also a key is usually decomposed into two parts, referring to x and y coordinates. Therefore, a node contains the following information: four pointers: quad[NW], quad[NE], quad[SW], and quad[SE] point; which in turn contains: key; usually expressed as x, y coordinates value.

▪ **Example 6.1 — Connected Component Labelling.** Consider two neighbouring black pixels in a binary image. They are adjacent if they share a bounding horizontal or vertical edge. In general, two black pixels are connected if one can be reached from the other by moving only to adjacent pixels (i.e. there is a path of black pixels between them where each consecutive pair is adjacent). Each maximal set of connected black pixels is a connected component. Using the quadtree representation of images, we can find and label these connected components in time proportional to the size of the quadtree. This algorithm can also be used for polygon colouring.

The algorithm works in three steps:

1. Establish the adjacency relationships between black pixels.
2. Process the equivalence relations from the first step to obtain one unique label for each connected component.

3. Label the black pixels with the label associated with their connected component.

To simplify the discussion, let us assume the children of a node in the quadtree follow the Z-order (SW, NW, SE, NE). Since we can count on this structure, for any cell we know how to navigate the quadtree to find the adjacent cells in the different levels of the hierarchy.

Step 1 is accomplished with a post-order traversal of the quadtree. For each black leaf *v* we look at the node or nodes representing cells that are northern neighbours and eastern neighbours (i.e. the northern and eastern cells that share edges with the cell of *v*). Since the tree is organized in Z-order, we have the invariant that the southern and western neighbours have already been taken care of and accounted for. Let the northern or eastern neighbour currently under consideration be *u*. If *u* represents black pixels:

- If only one of *u* or *v* has a label, assign that label to the other cell.
- If neither of them have labels, create one and assign it to both of them.
- If *u* and *v* have different labels, record this label equivalence and move on.

Step 2 can be accomplished using the union-find data structure. We start with each unique label as a separate set. For every equivalence relation noted in the first step, we union the corresponding sets. Afterwards, each distinct remaining set will be associated with a distinct connected component in the image.

Step 3 performs another post-order traversal. This time, for each black node *v* we use the union-find's find operation (with the old label of *v* to find and assign *v* its new label (associated with the connected component of which *v* is part).

6.4 R Tree

Objective 6.2 So far we have described many spatial data structures to report geometric queries, but all these data structures store multidimensional points, whereas R trees index special objects such as regions and answer geometric queries as objects or collections of objects.

An R tree is a spatial data structure developed for indexing multi-dimensional batch processing such as geographical information or graphics design. The R tree was invented by Antonin Guttman in 1984 and is very popular for storing spatial objects, the data structure provides solid theoretical foundation as well as useful applications.

Definition 6.4.1 R tree, or rectangle tree, is a highly balanced multiple-use tree data structure developed to index spatial objects using their geometric location. It is created in bottom up manner and very efficient for performing object, range object, and nearest object searches.

In R tree data structure, we group nearby objects and organize them with their minimum bounding rectangle (MBR) in the upper level of the tree. The whole tree is a hierarchical structure of inclusive MBRs except the leaves containing the objects.

Note Like other spatial data structures, R trees also partition search spaces in terms of MBRs. They create overlapping space partitioning and use optimization to minimize overlap instead of utilizing disjoint space partitioning.

6.4.1 Indexing structure of R tree

R tree is a multilevel height balance tree. Each node contains several keys and pointers for child nodes up to a determined maximum of *m*. A leaf level node contains information about objects. Internal nodes contain rectangles corresponding to children lying inside their parent nodes. Figure 6.4 shows the indexing and hierarchical bounding boxes of an R tree.

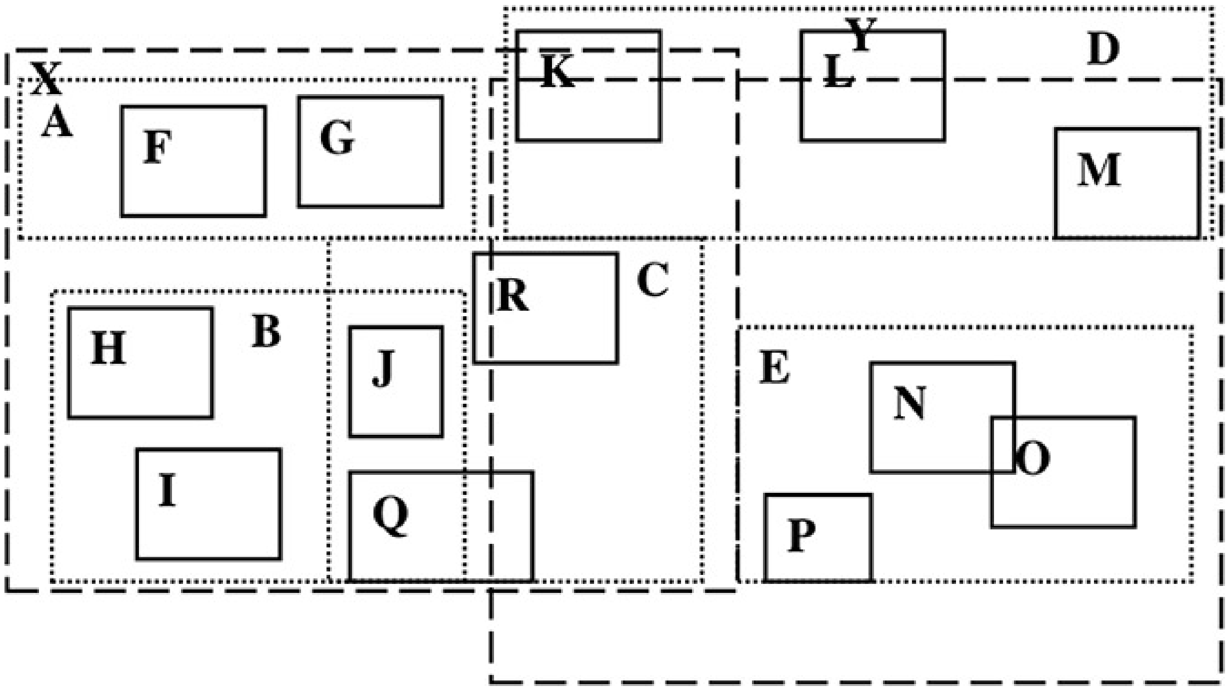


Figure 6.4: Construction of minimum bounding region of R tree.

6.4.1.1 Creation of R tree

Let, $S = \{O_1, O_2, O_3, \ldots O_n\}$ be a set of n objects of multidimensional space (R^d). Each object is associated with some arbitrary contiguous area in the space. task of R-tree is to index the objects to answer the query efficiently. Let $\{I_1, I_2, I3, \ldots, I_n\}$ be the identifiers corresponding to the objects, where each I_i is a hyper-rectangle or d-dimensional box covering the object O_i ($1 \leq i \leq n$) and I_i is a minimum box for a particular dimension u, (u_{min}, u_{max}) of O_i and (u_{min}, u_{max}) of I_i are the same; $u = u_{min}$ and $u = u_{max}$ are the boundary hyperplanes of I_i for u^{th} dimension. The leaf level entries are created as $E_i = (I_i$, tuple identifier); This identifier points to the object O_i. Non leaf entries are written as $E = (I$, child pointer), where child pointer and they point to a child of this node corresponding to the entry E and I is the minimum bounding region which encompasses all the regions corresponding to the entries of child node (Figure 6.5). We keep two parameters m and M to decide bounds for the number of entries in every internal node excluding root and a general convention is to take $M = 2m$, i.e. $m \leq$ number of entries $\leq M$.

In this process, the region of root is the minimum hyper-rectangle to encompass all identifiers as well all objects.

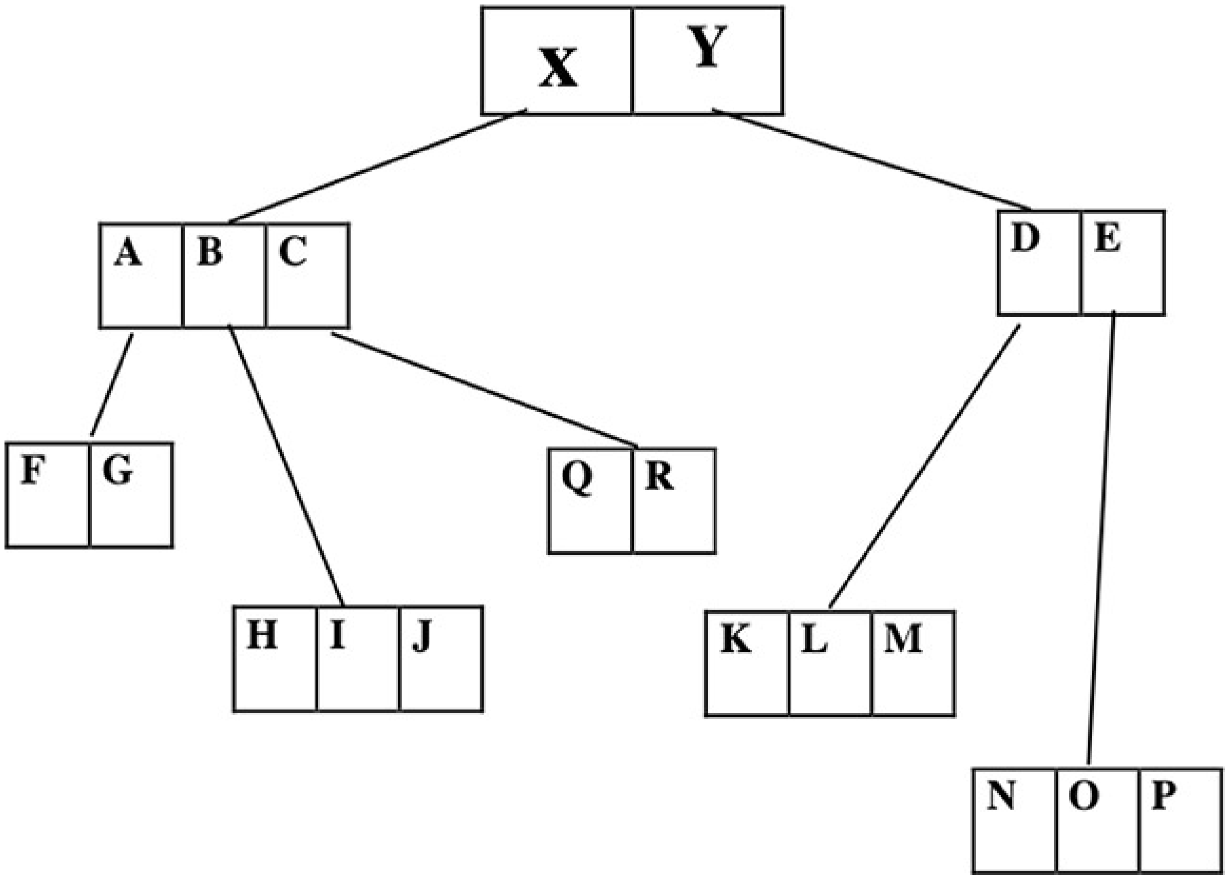


Figure 6.5: Example of R tree

6.4.2 Search in R tree

To search a given query object O_q, first we need to create MBR of O_q. Let I_q be the corresponding identifier. Start the search process by matching the region of I_q with MBRs of all entries of internodes, starting from the root. A successful match in internal node means inclusion of I_q within some MBR, (gets smaller at every match) and finally query O_q matches at leaf.

For range search we need to report all objects inside the range. The range search process follows a path from root to leaf and matches the region of query with MBRs of all entries of the internal nodes. The objective is to find the smallest MBR which contains the whole range and reports all the objects under the subtree rooted at that node.

In the case of a nearest neighbor search for a given point or region, the search process runs with the help of a priority queue. Starting from the root the algorithm inserts the nodes of the search path and their children into the priority queue. The priority values are decided by computing distance from the query point to the identifier. The find-min-return-by function at the end is the desired answer for nearest neighbor search in R tree.

6.4.3 Dynamic update of R tree

R trees can be updated dynamically through insert and delete operations. R trees are height balanced; the two-phase update method handles height without performing rotation or other external balancing steps.

6.4.3.1 Insertion in R tree

Insertion of a new object in the R tree is done by two phase traversal started from the root up to the leaf node of the search path. The first phase is similar to search and we create a new object and its identifier. In the second phase we traverse from leaf to upper level node until the identifier is included into a top level MBR. This process may require redefinition of the MBR and updating data in a reverse path up to the root. In some situations, we may need to split the node to maintain the upper bound *M* for number of entries. However, sharing of entry with the less crowded sibling is also possible.

Note Tree height may get increased in the second phase of insertion, when a split occurs at the root of the R tree but the system maintains height balance by increasing the heights of all the paths get increased.

6.4.3.2 Deletion in R tree

Deletion in R tree is similar to insertion and performed in two phases. In the first phase, a search operation is executed to find the object to be deleted. The second phase may be required only if the object if found and deleted. In this case, a bottom-up date of the MBRs in the affected path is required. In certain situations, nodes may be underfilled and may be joined with a sibling or take a sibling from a crowded node.

Note In the second phase of any update operation, MBRs must be updated optimally. The choice to share or merge siblings should be performed to maintain minimum overlaps between higher level MBRs.

6.5 Exercises

Exercise 6.1 Describe how to construct a d-dimensional range tree on n given points in $O(n\lg^{d-1} n)$ time. ▪

Exercise 6.2 Describe how to construct a d-dimensional layered range tree on n given points in $O(n\lg^{d-1} n)$ time. ▪

Exercise 6.3 What is a spatial data structure? Describe the applicability of the following data structures for various types of queries of special data.

1. Multi-dimensional range search tree
2. KD tree
3. Quadratic tree
4. R tree ▪

Exercise 6.4 What is nearest neighbour search problem? Write an algorithm for an approximate nearest neighbour search using the follow data structures:

1. Multi-dimensional range search tree
2. KD tree
3. Quadratic tree
4. R tree ▪

Exercise 6.5 What is an R tree? Explain the construction of R tree for special objects. How would you construct an internal MBR? Discuss the optimality issues for insertion in an R tree in the case of a split. Discuss the optimality issues for deletion in an R tree in the case of a join. ▪

Chapter 7

Temporal Data Structures

Objective 7.1 — Temporal Data Structures. We usually deal with data structures in traditional algorithmic settings. Updates are handled by modifying data or their underlying pointers. Information is lost because those processes do not retain previous data states. Temporal data structures preserve information from previous states so it is available for access.

In this chapter we discuss temporal data structures, which will allow us to view and/or modify past and present data structures. The specific behaviors of these data structures are defined below.

There are two primary models of temporal data structures. The first, called persistence, is based on the branching-universe model of time travel. In this model, going back in time and making changes creates a new branch of the data structure that differs from the original branch. The second, called retroactivity, works on the idea of round-trip time travel. Here, a time traveler goes back in time, makes a change, and then returns to observe the effects of his or her change. This model gives us a linear timeline with no branching.

We have vaguely referred to persistence as the ability to answer queries about the past states of the structure. Here we give several definitions of persistence.

7.1 Partial Persistence

In this persistence model we may query any previous version of the data structure, but we may only update the latest version. The relevant operations are read(var, version) and newversion = write(var, val).

? Can we use time travel models to review past data structures? ■

A persistent data structure that always preserves the previous version of itself when it is modified. Such data structures are effectively immutable, as their operations do not (visibly) update the structure in-place, but instead always yield a new updated structure. The term was introduced in 1986 by Driscoll et al. [86].

A data structure is partially persistent if all previous versions of the same data structure can be accessed for viewing (read only mode) but only the current version can be updated. The data structure is fully persistent if view and modification are possible in all the versions. If there is a special combination operation that can create a new version from two previous versions of the data structure, the system is a confluent persistent data structure. Data structures that are not persistent are ephemeral.

Persistent data structures are used in functional programming. Further, in the case of purely functional programs, all data is immutable, therefore all data structures are automatically fully persistent. Persistent data structures can also be created using in-place updating of data and these may, in general, use less time or storage space than their purely functional counterparts. Purely functional data structures are persistent in that they avoid the use of mutable state, but can still achieve attractive amortized time complexity bounds.

Definition 7.1.1 — Purely functional language. A purely functional language is a programming technique which does not allow any delete operation but can overwrite data by invoking a function call that returns newly computed data.

While persistence can be achieved by simple copying, this is inefficient in CPU and RAM usage, because most operations make only small changes to a data structure. A better method is to exploit the similarity between the new and old versions and share structures between them, such as using the same subtree in a number of tree structures. A garbage collection feature may be necessary if determining the number of previous versions that share parts of a structure is not feasible the user wishes to discard old versions. However, a sophisticated system such as the ZFS copy-on-write model can perform these tasks by directly tracking storage allocation.

7.1.1 Partial persistence

In the partial persistence model (Figure 7.1), we may query any previous version of the data structure, but we may only update the latest version. This implies a linear ordering among the versions.

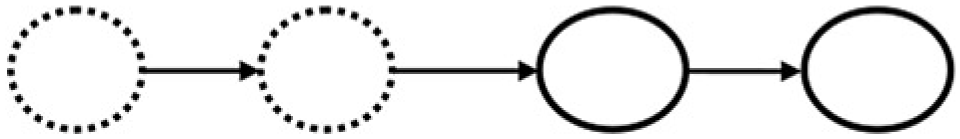


Figure 7.1: Partial persistence

7.1.1.1 Fat node method

This method records all changes made to nodes in node fields without erasing the old values of the fields. This requires that we allow nodes to become arbitrarily “fat”. In other words, each fat node contains the same information and pointer fields as an ephemeral nodes, along with space for an arbitrary number of extra field values. Each extra field value has an associated field name and a version stamp which indicates the version in which the named field was changed to have the specified value. Besides, each fat node has its own version stamp, indicating the version in which the node was created. The only purpose of nodes having version stamps is to make sure that each node only contains one value per field name per version. In order to navigate through the structure, each original field value in a node has a version stamp of zero.

Complexity 7.1.1 — Fat node method. Use of the fat node requires $O(a)$ space for every modification. Just store the new data. Each modification takes $O(1)$ additional time to store the modification at the end of the modification history. This is an amortized time bound, assuming we store the modification history in a growable array. For access time, we must find the right version at each node as we traverse the structure. If we made m modifications, then each access operation has $O(\log m)$ slowdown resulting from the cost of finding the nearest modification in the array.

7.1.1.2 Path copying

Path copy duplicates all nodes on the path containing the node about to be inserted or deleted. We must first cascade the change back through the data structure: all nodes that pointed to the old node must be modified to point to the new node instead. These modifications cause more cascading changes, and so on, until we reach the root. We maintain an array of roots indexed by timestamp. The data structure pointed to by time t's root is exactly time t's data structure.

Complexity 7.1.2 — Path copying method. The m modifications cost $O(\log m)$ lookup time. Modification time and space are bounded by the size of the structure, since a single modification may cause the entire structure to be copied. That is $O(m)$ for one update, and thus $O(n^2)$ preprocessing time.

7.1.1.3 A combination of fat node and path copying

Sleator and Tarjan (7) devised a way to combine the advantages of fat nodes and path copying, getting $O(1)$ access slowdown and $O(1)$ modification space and time.

In each node, we store one modification box. This box can hold one modification to a node (to a pointer, a key, another piece of node-specific data) along with a timestamp of the modification. Initially, every nodes modification box is empty.

Whenever we access a node, we check the modification box, and compare its timestamp against the access time. (The access time specifies the version of the data structure that we care about.) If the modification box is empty, or the access time is before the modification time, then we ignore the modification box and just deal with the normal part of the node. On the other hand, if the access time is after the modification time, then we use the value in the modification box, overriding that value in the node. (Say the modification box has a new left pointer. Then we'll use it instead of the normal left pointer, but we'll still use the normal right pointer.)

Modifying a node works like this. We assume that each modification touches one pointer or similar field. If the node's modification box is empty, then we fill it with the modification. Otherwise, the modification box is full. We make a copy of the node, but using only the latest values. That is, we overwrite one of the node's fields with the value that was stored in the modification box. Then we perform the modification directly on the new node, without using the modification box. We overwrite one of the new nodes fields, and its modification box stays empty. Finally, we cascade this change to the nodes parent, just like path copying. This may involve filling the parents modification box, or making a copy of the parent recursively. If the node has no parent-its the root-we add the new root to a sorted array of roots.

With this algorithm, given any time t, at most one modification box exists in the data structure with time t. Thus, a modification at time t splits the tree into three parts: one part contains the data from before time t, one part contains the data from after time t, and one part was unaffected by the modification.

7.1.1.4 Complexity of combination

Time and space for modifications require amortized analysis. A modification takes $O(1)$ amortized space, and $O(1)$ amortized time. To see why, use a potential function Φ, where $\Phi(T)$ is the number of full live nodes in T. The live nodes of T are just the nodes that are reachable from the current root at the current time (that is, after the last modification). The modification boxes of full live nodes are full.

Each modification involves some number of copies, say k, followed by one change to a modification box. You could add a new root but that would not change the argument. Consider each of the k copies. Each costs $O(1)$ space and time, but decreases the potential function by one. First, the node we copy must be full and live, so it contributes to the potential function. The potential function will only drop, however, if the old node isnt reachable in the new tree. Since the node is not reachable in the new tree, the next step in the algorithm will be to modify the nodes parent to point at the copy. Finally, we know the copys modification box is empty. Thus, weve replaced a full live node with an empty live node, and Φ goes down by one. The final step fills a modification box, which costs $O(1)$ time and increases Φ by one.

Putting it all together, the change in Φ is $\Delta\Phi = 1 - k$. Thus, weve paid $O(k + \Delta\Phi) = O(1)$ space and $O(k + \Delta\Phi + 1) = O(1)$ time.

7.1.2 Full persistence (see Figure 7.2)

In this model (Figure 7.1), both updates and queries are allowed on any version of the data structure. We have operations read(var, version) and newversion = write(var, version, val).

7.1.3 Confluent persistence

In this model (Figure 7.3) in addition to the previous operation, combination operations merge the inputs of previous versions into a single output version. We have operations read(var, version), newversion = write(var, version, val) and newversion = combine(var, val, version1, version2). Rather than a branching tree, combinations of versions induce a DAG (direct acyclic graph) structure on the version graph.

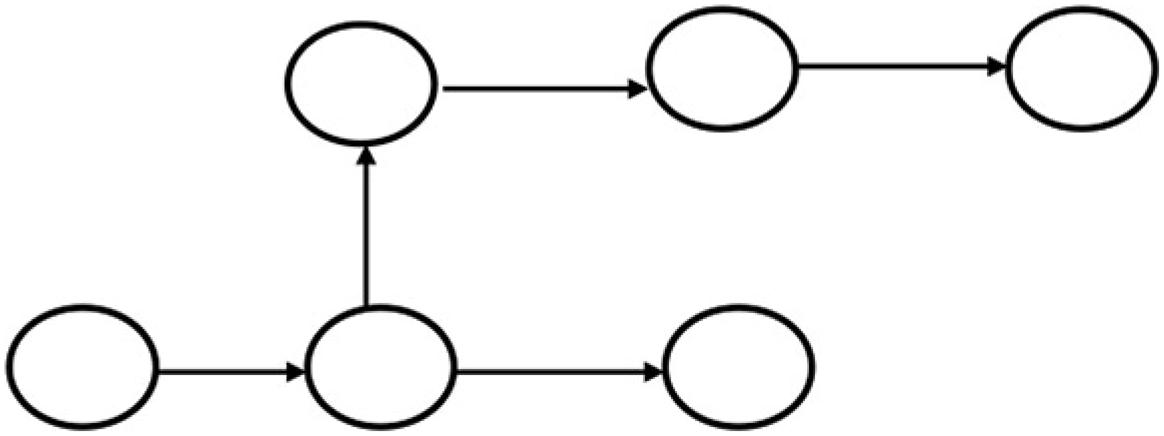


Figure 7.2: Full persistence

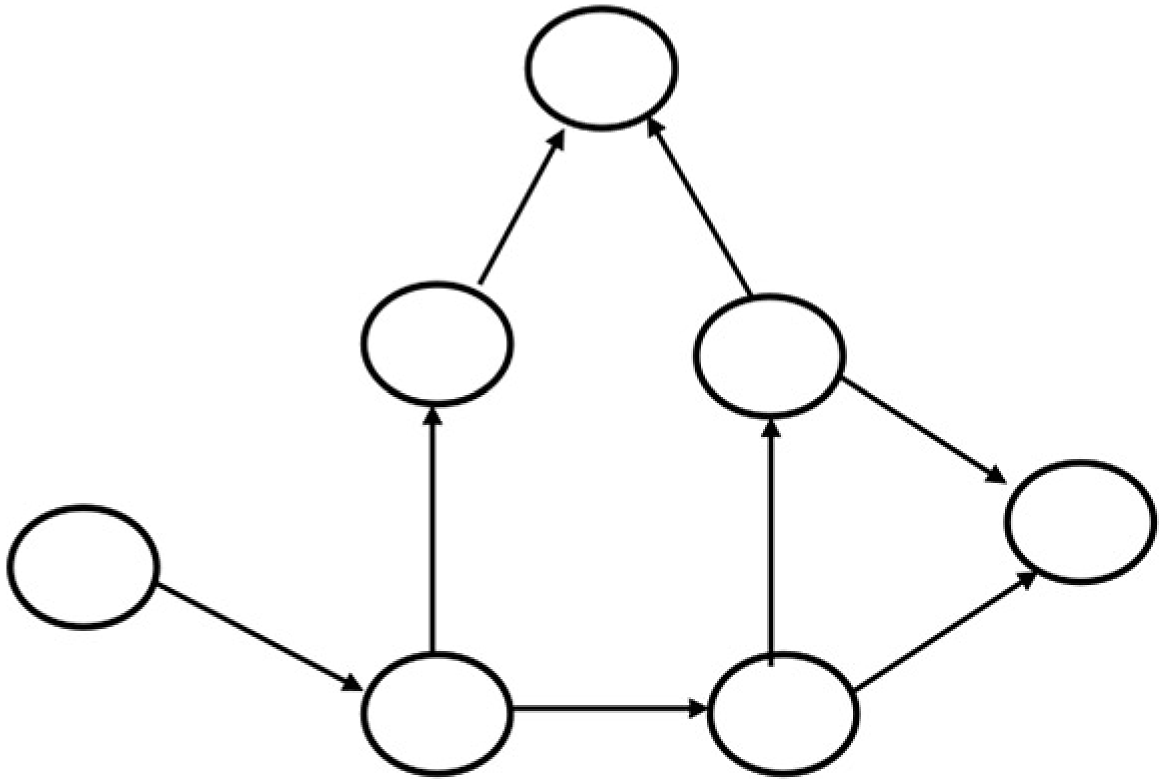


Figure 7.3: Confluent persistence

7.1.4 Functional persistence

This model takes its name from functional programming where objects are immutable (Figure 7.4). The nodes in this model are likewise immutable: revisions do not alter the existing nodes in the data structure but create new ones instead. The difference between functional persistence and the other types is the need to keep all the structures related to previous versions intact:

the only allowed internal operation is to add new nodes. The three previous types of persistence were far more flexible as long as we were able to implement the interface. Each of the succeeding levels of persistence is stronger than the preceding ones. Functional implies confluent, confluent implies full, and full implies partial. Functional implies confluent because we are simply restricting ways to implement persistence. Confluent persistence becomes full persistence if we do not use combinators. Full persistence when we limit our writing to the latest version.

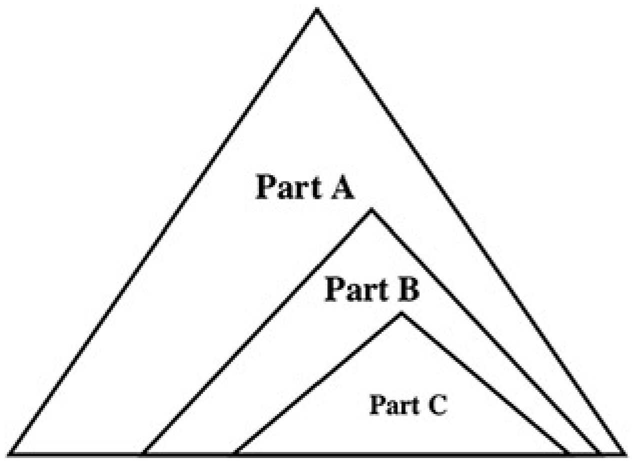


Figure 7.4: Different versions (Part A, Part B, Part C) of data structure in functional persistence

7.2 Retroactivity

A retroactive data structure (Figure 7.5) supports simple operations such as insertion (t, update), deletion (t), and query (t, query). An uppercase insert indicate an operation on a retroactive data structure. A lowercase update denotes an operation on the current data structure. Think of time t as an integer but a better approach is to use an order maintenance data structure to avoid using non-integers (in case you want to insert an operation between times t and $t + 1$).

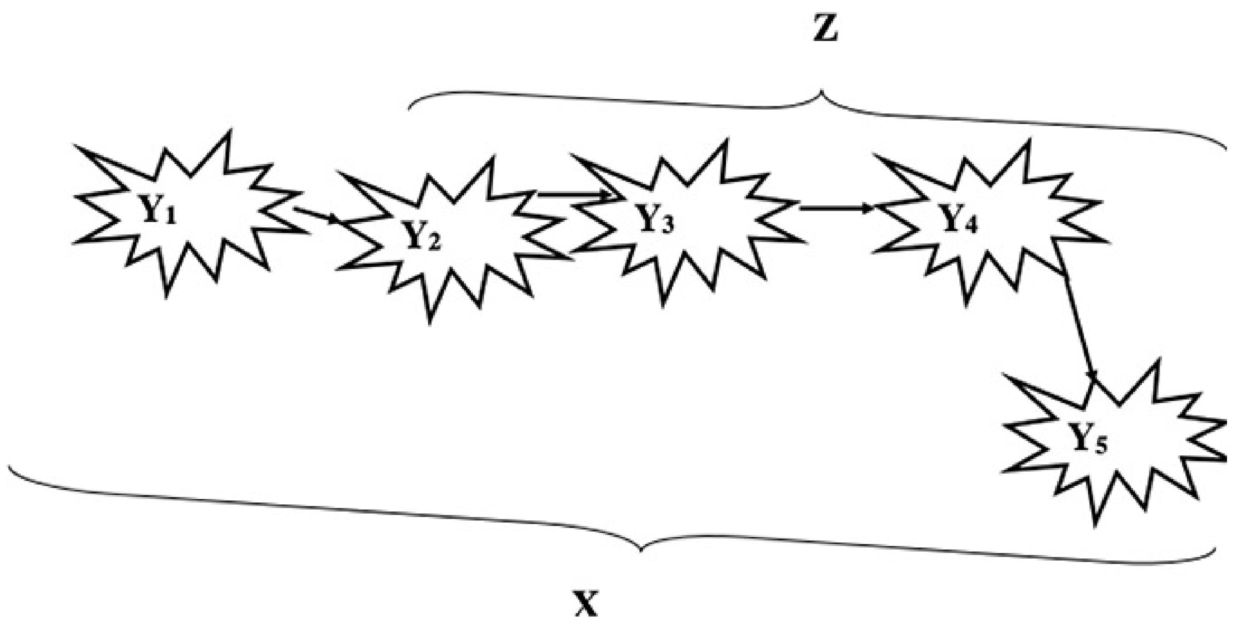


Figure 7.5: Retroactivity

7.2.1 Decomposable search problem

This is similar to a simple search problem but requires that queries must satisfy the following equation: $query(x, A \cup B) = f(query(x, A), query(x, B))$, for some function f computed in $O(1)$ (sets A and B may overlap). Examples of problems with such a function include dynamic nearest neighbor, successor on a line, and point location. We want to build a balanced search tree on time (leaves represent time). Every element "lives" in the data structure on the interval of time, corresponding to its insertion and deletion. Each element appears in $O(\log\ n)$ nodes. To query on this tree at time t, we want to know what operations have been done from the beginning of time to t. Because the query is decomposable, we can look at $O(\log n)$ different nodes and combine the results (using the function f).

▪ **Example 7.1 — Priority queue.** Priority queues are data structures in which retroactive operations potentially create chain reactions but still produce acceptable results. The main operations are insert and delete-min which we would like to retroactively Insert and Delete. It is possible to implement a partially retroactive priority queue with only $O(\log n)$ overhead per partially retroactive operation. Because of the presence of delete-min, the set of operations on priority queues is non-commutative. The order of updates now clearly matters, and inserting a delete-min retroactively has the potential to cause a chain reaction which changes subsequent results. ▪

Summary 7.1 — Temporal data structure. Temporal data structures are used to create timestamped indices of data and are useful in applications relevant to time travel operations.

Note: In this chapter we have discussed basics of temporal data structures. Interested readers are referred to Okasaki's book on functional data structures [88].

7.3 Exercises

Exercise 7.1 Given an ordered universe U of keys, develop and analyze a fully retroactive data structure that maintains $S \subseteq U$ and supports the following operations:

- `insert`(k) : `Insert` $k \in U$ `into` S
- `delete`(k) : `Remove` $k \in U$ `from` S
- `successor`(k) : `Return` $\min\{k' \in S \mid k' \geq k\}$

under the constraint that all `insert` operations must occur at time $-\infty$. All operations should run in time $O(\log m)$, where m is the total number of updates performed in the structure (retroactive or not). Observe that such a structure is sufficient to answer the "rightward ray shot" queries needed for the nonoblivious retroactive priority queue. ▪

Exercise 7.2 Create persistence versions of the following data structures:

1. Array
2. Link list

3. Binary search tree
4. Skiplist
5. Trie ▪

Chapter 8

External Memory Data Structures

There exist collections of data so large that no computer has enough memory to store them. In such cases, the application must resort to storing the data on some external medium such as a hard disk, a solid state disk, or even a network file server (which has its own external storage). Accessing an item from external storage is extremely slow. The hard disk attached to the computer on which this book was written has an average access time of 19 ms and the solid state drive attached to the computer has an average access time of 0.3 ms. In contrast, the random access memory in the computer has an average access time of less than 0.000113 ms.

Accessing RAM is more than 2500 times faster than accessing the solid state drive and more than 160000 times faster than accessing the hard drive. These speeds are fairly typical; accessing a random byte from RAM is thousands of times faster than accessing a random byte from a hard disk or solid-state drive. Access time, however, does not tell the whole story.

When we access a byte from a hard disk or solid state disk, an entire block of the disk is read. Each of the drives attached to the computer has a block size of 4096; each time we read one byte, the drive gives us a block containing 4096 bytes. If we organize our data structure carefully, each disk access could yield 4096 bytes that are helpful in completing whatever operation we are doing. This is the idea behind the external memory model of computation.

In this model, the computer has access to a large external memory in which all of the data resides. This memory is divided into memory blocks each containing B words. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are free; they take no time at all. The fact that internal memory computations are free may seem a bit strange, but it simply emphasizes the fact that external memory is so much slower than RAM.

In the external memory model, the size of the internal memory is also a parameter. However, for the data structures described in this chapter, it is sufficient to have an internal memory of size $O(B + \log_B n)$. That is, the memory needs to be capable of storing a constant number of blocks and a recursion stack of height $O(\log_B n$. In most cases, the $O(B)$ term dominates the memory requirement. For example, even with the relatively small value $B = 32$, $B \geq \log_B n$ for all $n \leq 2^{160}$.

8.1 Input/Output (I/O) Model

We will be working in the standard I/O model introduced by Aggarwal and Vitter [133]. The model has the following parameters: N = number of elements in the problem instance, M = number of elements that can fit into main memory, B = number of elements per block, where $M < N$ and $1 \leq B \leq M/2$. An I/O operation (or I/O) is a swap of B elements from internal memory with B consecutive elements from external memory (disk). The measure of performance we consider is the number of such I/Os needed to solve a problem.

As we shall see shortly, N/B (the number of blocks in the problem) and M/B (the number of blocks that fit into internal memory) play an important role in the study of I/O complexity. Therefore, we use n as shorthand for N/B and m for M/B. We say that an algorithm uses a linear number of I/Os if it uses $O(n)$ I/Os.

Early work on I/O algorithms concentrated on algorithms for sorting and permutation-related problems in the single disk model, as well as in the extended version of the I/O model. In the single disk model, external sorting requires $O(n \log_m N)$ I/Os, which is the external equivalent of the well-known $(N \log N)$ internal sorting bound. Searching for an element among N elements requires $O(\log_B N)$ I/Os. More recently external algorithms and data structures have been developed for a number of problems in different areas.

An important consequence of the fundamental external memory bounds for sorting and searching is that, unlike in internal memory, use of on-line efficient data structures in algorithms for batched problems often leads to inefficient algorithms. Consider for example sorting N elements by performing N insert followed by N delete operations on a B tree. This algorithm perform s$O(N\log_B N)$ I/Os, which is a factor B log Bm from optimal. Thus while for on-line problems $O(\log_B N)$ is the I/O bound corresponding to the $O(\log_2 N)$ bound on many internal memory data structure operations, for batched problems the corresponding bound is $O((\log_m N)/B)$. The problem with the B tree in a batched context is that it does not take advantage of the large main memory effectively; it works in a model where the size of the main memory is equal to the block size.

8.2 Cache Oblivious Algorithms

8.2.1 Cache aware model

The memory system of most modern computers consists of a hierarchy of memory levels, with each level acting as a cache for the next; for a typical desktop computer the hierarchy consists of registers, level 1 cache, level 2 cache, level 3 cache, main memory, and disk. The model defines a computer as having two levels: 1. The cache which is near the CPU, cheap to access, but limited in space. 2. The disk which is distant from the CPU, expensive to access, but nearly limitless in space. The main aspect of this model is that transfers between cache and disk involve blocks of data. As a consequence of this, the memory access pattern of an algorithm has a major influence on its practical running time. If the program is aware of the cache hardware, the information can be used to optimize the cache complexity for the particular cache size and line length.

8.2.2 Cache oblivious model

There are situations where the memory transfer between two levels of the memory hierarchy dominates the running time, which is often the case when the size of the data exceeds the size of main memory. Besides, every machine has a different memory hierarchy and cache size. Therefore, it is not always possible to know the right parameters to achieve the optimal run time.

8.2.2.1 Consequences of using Cache Oblivious

First, if a cache-oblivious algorithm performs well between two levels of the memory hierarchy (cache and disk), then it must automatically work well between any two adjacent levels of the memory hierarchy, because blocks in memory levels nearer the CPU store subsets of memory levels farther from the CPU. Second, we can design and analyze algorithms in a two-level memory model, and obtain results for an arbitrary many-level memory hierarchy, i.e., algorithm formulated in the RAM model but analyzed in the I/O model. Why? Since the I/O-model analysis holds for any block and memory size, it holds for all levels of a multi-level memory hierarchy.

8.2.2.2 Assumptions

We do not know the page-replacement strategy cache hardware is using. We can make two assumptions. Firstly, that the page replacement is optimal, and second, that the cache is fully associative.

Optimal page replacement specifies that the future is known and it always evicts the page that will be accessed farthest in the future. Least Recently Used (LRU) is a more realistic solution because future is not known.

Full associativity allows any block to be stored anywhere in a cache. Typical real-world caches are either directed mapped ($c = 1$) or two-way associative ($c = 2$). It is not uncommon to assume that a cache is taller than it is wide, that is, the number of blocks, M/B, is larger than the size of each block, B.

Cache oblivious models work by recursive divide-and-conquer algorithms. A problem is divided into smaller and smaller subproblems until the system reaches a subproblem size that fits into the cache, regardless of the cache size.

8.3 B, B+ Tree

The binary tree was developed to perform search operations in the internal memory, generally in the assumption of RAM model that supports search queries for n items in $O(\log n)$ time, which is optimal for the number of comparisons required for computation (Figure 8.1). A common adaptation of the concept of the external memory model is the B tree. Data are transferred in blocks of B items and provide an $\Omega(\log_B N)$ I/O lower bound. These lower bounds depend mainly on the parameters of the model.

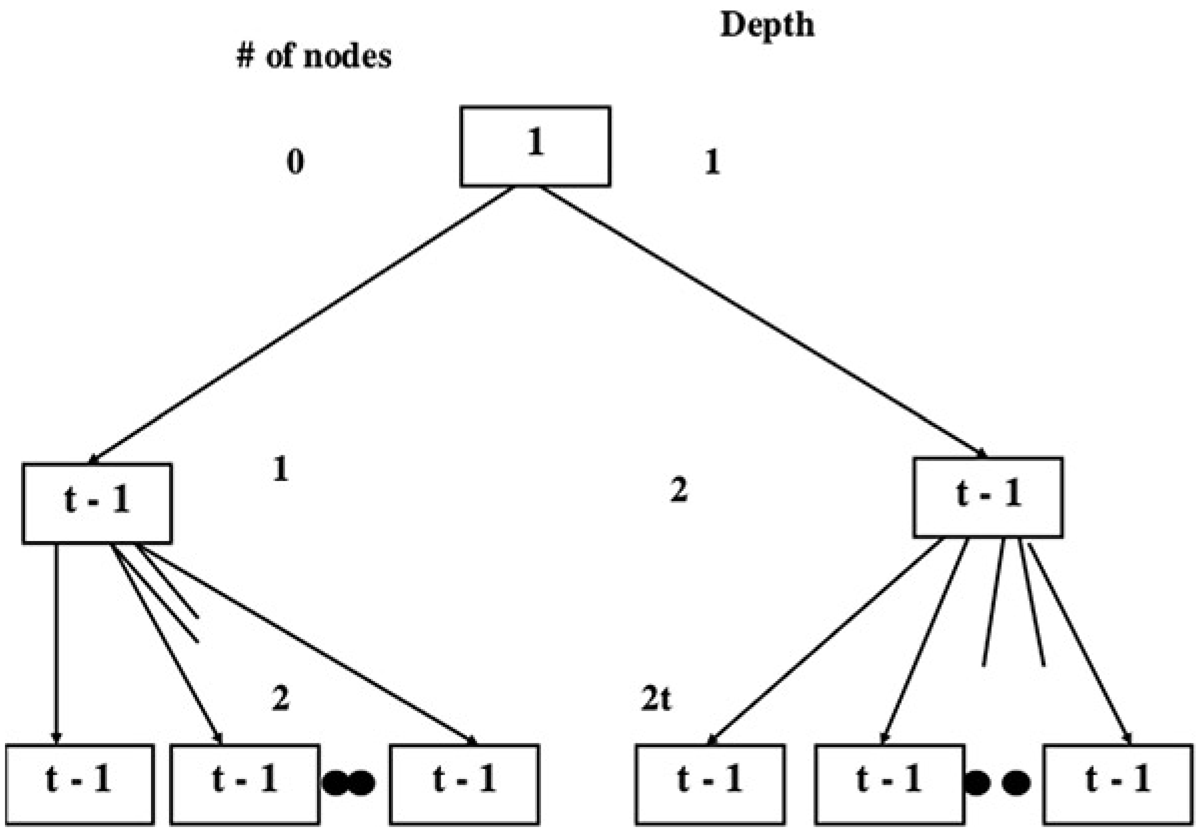


Figure 8.1: B-tree node structure

To facilitate block transfers for data structure operations involving external memory access, a node of the tree represented by a block that can store $\theta(B)$ data values and pointers is required for $\theta(B)$-way branching. The well-known balanced multiway B tree due to Bayer and McCreight [194], is the most widely used nontrivial external memory data structure. The degree of each node in the B tree (with the exception of the root) is required to be $\theta(B)$, which guarantees that the height of a B tree storing N items is roughly $O(\log_B N)$. B trees support dynamic dictionary operations and one-dimensional range search optimally in linear space using $O(\log_B N)$ I/Os per insert or delete and $O(\log_B(N) + z)$ I/Os per query, where $Z = zB$ is the number of items output. When a node overflows during an insertion, it splits into two half-full nodes, and if the splitting causes the parent node to have too many children and overflow, the parent node splits, and so on. Splittings can thus propagate up to the root, which is how the tree grows in height. Deletions are handled in a symmetric way by merging nodes.

8.3.1 Searching

The implementation of the find(x) operation generalizes the find(x) operation in a binary search tree. The search for x starts at the root and uses the keys stored at a node, u, to determine in which of u's children the search should continue.

More specifically, at a node u, the search checks if x is stored in $u.\mathit{keys}$. If so, x has been found and the search is complete. Otherwise, the search finds the smallest integer, i, such that $u.\mathit{keys}[i] > x$ and continues the search in the subtree rooted at $u.\mathit{children}[i]$. If no key in $u.\mathit{keys}$ is greater than x, then the search continues in u's rightmost child. Just like binary search trees, the algorithm keeps track of the most recently seen key, z, that is larger than x. In case x is not found, z is returned as the smallest value that is greater than or equal to x.

We can analyze the running time of a B tree find(x) operation both in the usual word-RAM model (where every instruction counts) and in the external memory model (where we only count the number of nodes accessed). Since each leaf in a B tree stores at least one key and the height of a B tree with ℓ leaves is $O(\log_B \ell)$, the height of a B tree that stores n keys is $O(\log_B n)$. Therefore, in the external memory model, the time taken by the find(x) operation is $O(\log_B n)$. To determine the running time in the word-RAM model, we have to account for the cost of calling $\text{find\_it}(a, x)$ for each node we access, so the running time of find(x) in the word-RAM model is $O(\log_B n) \times O(\log B) = O(\log n)$.

8.3.2 Insertion

One important difference between B trees and binary search trees is that the nodes of a B tree do not store pointers to their parents. The lack of parent pointers means that the add(x) and remove(x) operations on B trees are most easily implemented using recursion.

Like all balanced search trees, some form of re-balancing is required during an add(x) operation. In a B tree, this is done by splitting nodes. Although splitting takes place across two levels of recursion, it is best understood as an operation that takes a node u containing $2B$ keys and having $2B + 1$ children. It creates a new node, w, that adopts u.

Note The splitting operation modifies three nodes: u, u's parent, and the new node, w. This is why it is important that the nodes of a B tree do not maintain parent pointers. If they did, then the $B + 1$ children adopted by w would all need to have their parent pointers modified. This would increase the number of external memory accesses from 3 to $B + 4$ and would make B trees much less efficient for large values of B.

The add(x) method in a B tree finds a leaf, u, at which to add the value x. If this causes u to become overfull (because it already contained $B - 1$ keys), then u is split. If this causes u's parent to become overfull, then u's parent is also split, which may cause u's grandparent to become overfull, and so on. This process continues, moving up the tree one level at a time until reaching a node that is not overfull or until the root is split. In the former case, the process stops. In the latter case, a new root is created whose two children become the nodes obtained when the original root was split.

The executive summary of the add(x) method is that it walks from the root to a leaf searching for x, adds x to this leaf, and then walks back up to the root, splitting any overfull nodes it encounters along the way.

? How do we implement this operation recursively?

The real work of add(x) is done by the $\text{add\_recursive}(x, ui)$ method, which adds the value x to the subtree whose root, u, has the identifier ui. If u

is a leaf, then x is simply inserted into $u.keys$. Otherwise, x is added recursively into the appropriate child, u', of u. The result of this recursive call is normally *nil* but may also be a reference to a newly-created node, w, that was created because u' was split. In this case, u adopts w and takes its first key, completing the splitting operation on u'.

After the value x has been added (either to u or to a descendant of u), the $\text{add\_recursive}(x, ui)$ method checks to see if u is storing too many (more than $2B - 1$) keys. If so, then u needs to be split with a call to the $u.\text{split}()$ method. The result of calling $u.\text{split}()$ is a new node that is used as the return value for $\text{add\_recursive}(x, ui)$.

The $\text{add\_recursive}(x, ui)$ method is a helper for the add(x) method, which calls $\text{add\_recursive}(x, ri)$ to insert x into the root of the B tree. If $\text{add\_recursive}(x, ri)$ causes the root to split, then a new root is created that takes as its children both the old root and the new node created by the splitting of the old root.

The add(x) method and its helper, $\text{add\_recursive}(x, ui)$, can be analyzed in two phases.

8.3.2.1 Downward phase

During the downward phase of the recursion, before x has been added, the system accesses a sequence of B tree nodes and calls find_it(a,x) on each node. As with the find(x) method, this takes $O(\log_B n)$ time in the external memory model and $O(\log n)$ time in the word-RAM model.

8.3.2.2 Upward phase

During the upward phase of the recursion, after x has been added, these methods perform a sequence of at most $O(\log_B n)$ splits. Each split involves only three nodes, so this phase takes $O(\log_B n)$ time in the external memory model. However, each split involves moving B keys and children from one node to another, so in the word-RAM model, this takes $O(B\log n)$ time.

Note The value of B can much larger than even $\log n$. Therefore, in the word-RAM model, adding a value to a B tree can be much slower than adding into a balanced binary search tree. The amortized number of split operations done during an add(x) operation is constant. This shows that the (amortized) running time of the add(x) operation in the word-RAM model is $O(B + \log n)$.

8.3.3 Removal

The remove(x) operation in a B tree is, again, most easily implemented as a recursive method. Although the recursive implementation of remove(x) spreads the complexity across several methods. By shuffling keys around, removal is reduced to the problem of removing a value, x', from some leaf, u. Removing x' may leave u with less than $B - 1$ keys; this situation is called an underflow.

When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's grandparent to underflow. This process works its way back up to the root until there is no more underflow or until the root has its last two children merged into a single child. When the latter case occurs, the root is removed and its lone child becomes the new root.

Next we delve into the details of how each of these steps is implemented. The first job of the remove(x) method is to find the element x that should be removed. If x is found in a leaf, then x is removed from this leaf. Otherwise, if x is found at $u.keys[i]$ for some internal node, u, then the algorithm removes the smallest value, x', in the subtree rooted at $u.children[i + 1]$. The value x' is the smallest value stored in the B tree that is greater than x. The value of x' is then used to replace x in $u.keys[i]$.

After recursively removing the value x from the ith child of u, remove_recursive(x, ui) needs to ensure that this child still has at least $B - 1$ keys. In the preceding code, this is done using a method called check_underflow(x, i), which checks for and corrects an underflow in the ith child of u. Let w be the ith child of u. If w has only $B - 2$ keys, then this needs to be fixed. The fix requires using a sibling of w. This can be either child $i + 1$ of u or child $i - 1$ of u. We will usually use child $i - 1$ of u, which is the sibling, v, of w directly to its left. The only time this doesn't work is when $i = 0$, in which case we use the sibling directly to w's right.

In the following, we focus on the case when $i \neq 0$ so that any underflow at the ith child of u will be corrected with the help of the $(i - 1)$st child of u. The case $i = 0$ is similar and the details can be found in the accompanying source code.

To fix an underflow at node w, we need to find more keys (and possibly also children), for w. There are two ways to do this:

8.3.3.1 Borrowing

If w has a sibling, v, with more than $B - 1$ keys, then w can borrow some keys (and possibly also children) from v. More specifically, if v stores size(v) keys, then between them, v and w have a total of $B - 2 + \text{size}(w) \geq 2B - 2$ keys. We can therefore shift keys from v to w so that each of v and w has at least $B - 1$ keys.

8.3.3.2 Merging

If v has only $B - 1$ keys, we must do something more drastic, since v cannot afford to give any keys to w. Therefore, we merge v and w. The merge operation is the opposite of the split operation. It takes two nodes that contain a total of $2B - 3$ keys and merges them into a single node that contains $2B - 2$ keys. (The additional key comes from the fact that, when we merge v and w, their common parent, u, now has one less child and therefore needs to give up one of its keys.)

The remove(x) method in a B tree follows a root to leaf path, removes a key x' from a leaf, u, and then performs zero or more merge operations involving u and its ancestors, and performs at most one borrowing operation. Since each merge and borrow operation involves modifying only three nodes, and only $O(\log_B n)$ of these operations occur, the entire process takes $O(\log_B n)$ time in the external memory model. Again, however, each merge and borrow operation takes $O(B)$ time in the word-RAM model, so (for now) the most we can say about the running time required by remove(x) in the word-RAM model is that it is $O(B\log_B n)$.

8.3.4 Amortized analysis of B trees

In the external memory model, the running time of find(x), add(x), and remove(x) in a B tree is $O(\log_B n)$. In the word-RAM model, the running time of find(x) is $O(\log n)$ and the running time of add(x) and remove(x) is $O(B\log n)$.

Theorem 8.3.1 Starting with an empty B tree and performing any sequence of m add(x) and remove(x) operations results in at most $3m/2$ splits, merges, and borrows being performed.

Proof. The proof of this theorem using accounting method is given here. Each split, merge, or borrow operation is paid for with two credits, i.e., a credit is removed each time one of these operations occurs and at most three credits are created during any add(x) or remove(x) operation.

Since at most $3m$ credits are ever created and each split, merge, and borrow is paid for with with two credits, it follows that at most $3m/2$ splits, merges, and borrows are performed.

To keep track of these credits the proof maintains the following credit invariant: Any non-root node with $B - 1$ keys stores one credit and any node with $2B - 1$ keys stores three credits. A node that stores at least B keys and most $2B - 2$ keys need not store any credits. What remains is to show that we can maintain the credit invariant and satisfy properties 1 and 2, above, during each add(x) and remove(x) operation.

8.3.4.1 Amortized analysis of insertion

The add(x) method does not perform any merges or borrows, so we need only consider split operations that occur as a result of calls to add(x).

Each split operation occurs because a key is added to a node, u, that already contains $2B - 1$ keys. When this happens, u is split into two nodes, u' and u'' having $B - 1$ and B keys, respectively. Prior to this operation, u was storing $2B - 1$ keys, and hence three credits. Two of these credits can be used to pay for the split and the other credit can be given to u' (which has $B - 1$

keys) to maintain the credit invariant. Therefore, we can pay for the split and maintain the credit invariant during any split.

The only other modification to nodes that occur during an add(x) operation happens after all splits, if any, are complete. This modification involves adding a new key to some node u'. If, prior to this, u' had $2B-2$ children, then it now has $2B-1$ children and must therefore receive three credits. These are the only credits given out by the add(x) method.

8.3.4.2 Amortized analysis of deletion

During a call to remove(x), zero or more merges occur and are possibly followed by a single borrow. Each merge occurs because two nodes, v and w, each of which had exactly $B-1$ keys prior to calling remove(x) were merged into a single node with exactly $2B-2$ keys. Each such merge therefore frees up two credits that can be used to pay for the merge.

After any merges are performed, at most one borrow operation occurs, after which no further merges or borrows occur. This borrow operation only occurs if we remove a key from a leaf, v, that has $B-1$ keys. The node v therefore has one credit, and this credit goes towards the cost of the borrow. This single credit is not enough to pay for the borrow, so we create one credit to complete the payment.

At this point, we have created one credit and we still need to show that the credit invariant can be maintained. In the worst case, v's sibling, w, has exactly B keys before the borrow so that, afterwards, both v and w have $B-1$ keys. This means that v and w each should be storing a credit when the operation is complete. Therefore, in this case, we create an additional two credits to give to v and w. Since a borrow happens at most once during a remove(x) operation, this means that we create at most three credits, as required.

If the remove(x) operation does not include a borrow operation, this is because it finishes by removing a key from some node that, prior to the operation, had B or more keys. In the worst case, this node had exactly B keys, so that it now has $B-1$ keys and must be given one credit, which we create.

In either case, whether the removal finishes with a borrow operation or not–at most three credits need to be created during a call to remove(x) to maintain the credit invariant and pay for all borrows and merges that occur. This completes the proof of the lemma.

Note In the word-RAM model the cost of splits, merges and joins during a sequence of m add(x) and remove(x) operations is only $O(Bm)$. That is, the amortized cost per operation is only $O(B)$, so the amortized cost of add(x) and remove(x) in the word-RAM model is $O(B + \log n)$.

Corollary 8.3.2 — External Memory B Trees. In the external memory model, a B tree supports the operations add(x), remove(x), and find(x) in $O(\log_B n)$ time per operation.

Corollary 8.3.3 — Word-RAM B Trees. A B tree supports the operations add(x), remove(x), and find(x) in $O(\log n)$ time per operation in the word-RAM model where the cost of splits, merge and share is ignored. For any sequence of n add(x) and remove(x) operations the total time required to perform splits, merges and shares is $O(B \times n)$.

8.3.5 B+ tree

In the B+ tree variant, all the items are stored in the leaves, and the leaves are linked together in symmetric order to facilitate range queries and sequential access. The internal nodes store only key values and pointers and thus can have a higher branching factor. In the most popular variant of B+ trees, called B* trees, splitting can usually be postponed when a node overflows, by sharing the node's data with one of its adjacent siblings. The node needs to be split into two nodes when sibling is also full and in that case the keys of the node and its full sibling are redistributed evenly to make all three nodes to about 2/3 full, which allows space for future sharing. This local optimization reduces the number of times new nodes must be created and thus increases storage capacity. And since there are fewer nodes in the tree, search I/O costs are lower. Storage utilization can be increased further by sharing among several siblings, at the cost of more complicated insertions and deletions. Some helpful space-saving techniques borrowed from hashing are partial expansions and use of overflow nodes.

8.4 (a,b) Tree

In this section we explore the use of weak B trees for the representation of linear lists (Figure 8.2). In weak B trees all leaves have the same depth and every interior node has at least a and at most b children for some constants a, b with $a \geq 2$ and $2a - 1 \leq b$. We analyze the cost of sequences of insertions and deletions into weak B trees and show that this cost is linear in the length of the sequence when the initial tree is empty, and sublinear when b-2a is sufficiently large. In the case of an arbitrary starting tree we derive a bound in terms of the positions of the insertions and deletions. We also show that the numbers of insertions and deletions which require k re-balancing operations (i.e., the last k nodes of the path to the root are effected by local changes) decreases exponentially with k.

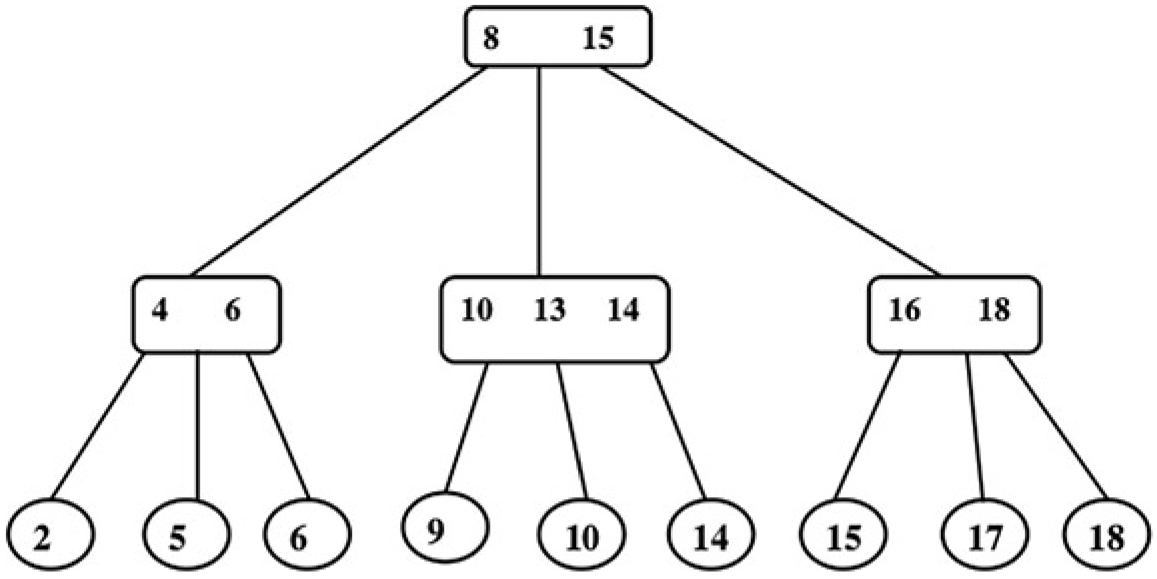


Figure 8.2: (a,b) tree

We conclude that weak B trees support a high degree of concurrency even in the presence of insertions and deletions. In B trees all leaves have the same depth and each internal node has at least a and at most $2a$-1 children where a is some constant, the order of the tree. In weak B trees we allow for a wider range of flexibilities of numbers of keys in a node. Assume that the node term refers only to internal nodes and not leaves and $p(v)$ denotes the number of children of node V and T_1 denotes the number of leaves of T.

Definition 8.4.1 Let a and b be integers with $a \geq 2$ and $2a - 1 \leq b$. A tree T is an (a, b)-tree if

1. All leaves of T have the same depth.
2. All nodes v of T satisfy $p(v) \leq b$.
3. All nodes v except the root satisfy $p(v) \geq a$.
4. The root r can have one key.

Note The B tree described in the section above is a special case of an (a,b) tree with b = 2a-1, i.e., B tree is a (a, 2a-1) tree.

The number of leaves in an (a,b) tree is logarithmic. Insertion and deletion into (a,b)-trees is quite similar to the corresponding operations in B trees. An insertion means the addition of a new leaf at a given position in the tree, a deletion means the pruning of an existing leaf at a given position in the tree. Note that we treat the searches for these positions separately in what follows, i.e., for the moment we concentrate at the re-balancing aspect of (a, b)-trees.

8.4.1 Insertion

An insertion is accomplished by a sequence of node expansions and node splittings, terminating in a balanced (a, b)-tree. Let w be any leaf of T and suppose that a new leaf is to be inserted to the right (left) of w. Let v be the parent of w. Expand v, i.e., make the new leaf an additional child of v. The expansion of v increases $p(v)$ by i. If $p(v)$ is still $<b$ then re-balancing is

complete. Otherwise v needs to be split. Since splitting may propagate we formulate it as a loop.

8.4.2 Deletion

A deletion is accomplished by a sequence of node shrinking and node fusing possibly followed by one node sharing. Deletion has two parameters, the sharing threshold t, which specifies when to share or fuse and the shifting parameter s, which specifies the number of children to shift when sharing. Let w be any leaf of T (the leaf to be deleted) and let v be the parent of w, in first step we shrink v by means of pruning the w. This decreases the value of $p(v)$ by 1. If $p(v)$ is still $> a$ or the height of v is 1 then re-balancing is completed, because we represent the empty tree by a single node. Otherwise, v needs to be rebalanced by either fusing or sharing. Let u be any sibling of v, when $p(v) = a - 1$ and $p(u) = a + j$ during deletion, the algorithm performs a node fusing if $j < t$, otherwise a sharing will take place based on the value of s.

Theorem 8.4.1 If $b \geq 2a$, then i insertions and d deletions perform at most $O(\delta h\ (i + d))$ splits and fusions at height h, where $\delta < 1$ depends on a and b.

Theorem 8.4.2 Searching for an element among N elements in external memory requires $\Omega(\log_{B+1} N)$ I/Os.

Note: The lower bound holds even if an I/O can read B arbitrary elements from memory.

8.5 Buffer Tree

An important paradigm for constructing algorithms for batched problems in an internal memory setting is to use a dynamic data structure to process a sequence of updates. For example, we can sort N items by inserting them one by one into a priority queue, followed by a sequence of N delete-min operations. Similarly, many batched problems in computational geometry can be solved by dynamic plane sweep techniques. Orthogonal segment intersections can be tracked dynamically by active vertical segments (i.e., those hit by the horizontal sweep line); we mentioned a similar algorithm for orthogonal rectangle intersections.

However, if we use this paradigm naively in an External Memory (EM) setting, with a B tree as the dynamic data structure, the resulting I/O performance will be highly non-optimal. For example, if we use a B tree as the priority queue in sorting or storing the active vertical segments hit by the sweep line, each update and query operation will take $O(\log_B(N))$ I/Os, resulting in a total of $O(N \log_B N)$ I/Os, which is larger than the optimal bound by a substantial factor of B approximately.

One solution suggested is to use a binary tree data structure in which items are pushed lazily down the tree in blocks of B items at a time. The binary nature of the tree results in a data structure of height $O(\log n)$, yielding a total I/O bound of $O(n \log n)$, which is still non optimal by a factor of $\log m$.

The buffer tree data structure (Figure 8.3) was developed to support batched dynamic operations, as in the sweep line example, where the queries do not have to be answered right away or in any particular order. The buffer tree is a balanced multiway tree, but with degree $\theta(m)$ rather than degree $\theta(B)$, except possibly for the root. Its key distinguishing feature is that each node has a buffer that can store $\theta(M)$ items (i.e., $\theta(m)$ blocks of items).

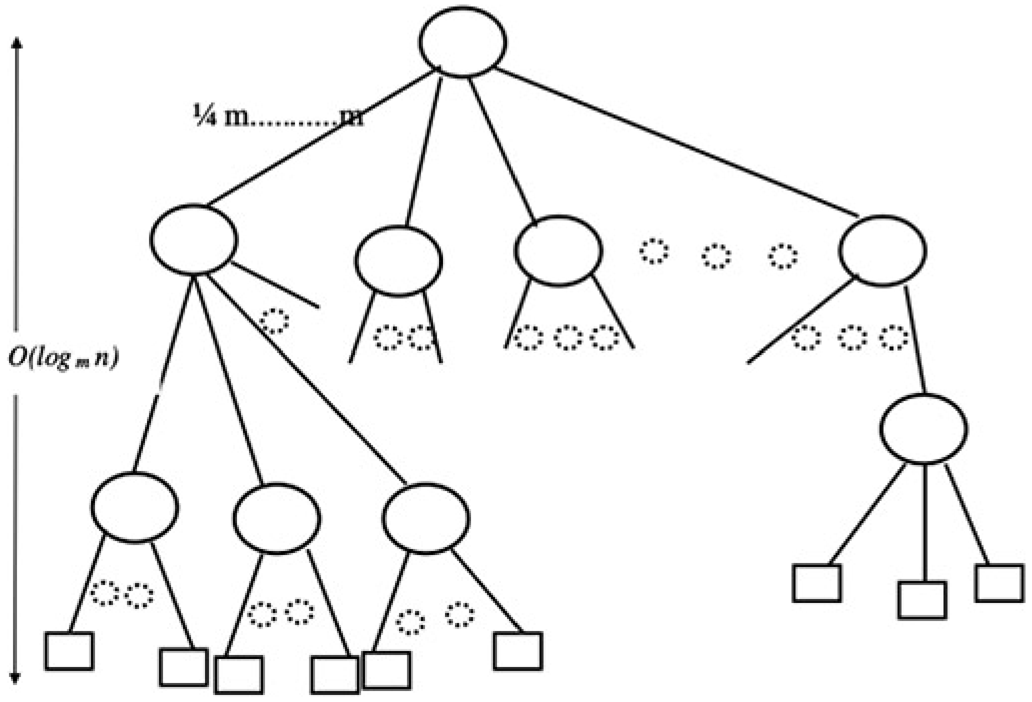


Figure 8.3: Buffer tree

Items in a node are pushed down to the children when the buffer fills. Emptying a full buffer requires $\theta(m)$ I/Os, which amortizes the cost of distributing the M items to the $\theta(m)$ children. Each item thus incurs an amortized cost of $O(\text{m/M}) = O(1/\text{B})$ I/Os per level, and the resulting cost for queries and updates is $O(1/\text{B}) \log_m n$ I/Os amortized.

The number of applications for buffer trees continues to increase. Buffer trees can be used as subroutines in the standard sweep line algorithm in order to get an optimal external memory algorithm for orthogonal segment intersection.

Buffer trees provide a natural amortized implementation of priority queues for application which involve time dependent processing like discrete event simulation, line sweeping, and list ranking. To perform B insertions and delete-min operations in a priority queue using buffer tree requires only $O(\log_m n)$ I/Os.

Theorem 8.5.1 For an arbitrary sequence of N insert and delete operations on an initially empty buffer tree which uses $O(n)$ *space*, the requirement of I/O is $O(n\log_m n)$, i.e., the amortized cost per operation is $O((log_m n)/B)$ I/Os.

Proof. As a buffer-emptying process on a node containing $x > m$ blocks uses $O(x)$ I/O, without counting the I/Os used for re-balancing, the total cost of all buffer-emptying processes on nodes with full buffers is bounded by $O(n\log_m n)$ I/Os. In an (a,b) tree, we know that if $b > 2\ a$, the number of re-balancing operations in a sequence of K updates in an initially empty (a,b)-tree is bounded by O(K/(b/2-a)). As we are inserting blocks in the (m/4,m) tree underlying the buffer tree, the total number of re-balance operations in a sequence of N updates on the buffer tree is bounded by O(n /m). Each re-balance operation takes O (m) I/Os (O(m) I/Os may be used by each delete re-balance operation to empty a non-full buffer), and thus the total cost of the re-balancing is O(n). This proves the first part of the theorem. The structure uses linear space since the tree uses O(n) blocks, the O(n/m) non-full buffers use O(m) blocks each, and each element is stored in O(1) places in the structure. ▪

8.6 Exercises

Exercise 8.1 Develop and analyze a cache-oblivious first-in-first-out queue. Both the enqueue and the dequeue operations should take $O(1/B)$ amortized memory transfers. Your data structure should only use external memory indices in $\{0, 1, \cdots, O(N)\}$, where N is the maximum number of elements stored in the queue at once. ▪

Exercise 8.2 Given an unordered array of N elements, develop and analyze a cache-oblivious algorithm to find the median of the array in $O(\lceil N/B \rceil)$ memory transfers. In your solution, you may assume knowledge of the standard median-of-medians deterministic selection algorithm. ▪

Exercise 8.3 Given a set of N horizontal and vertical line segments, develop and analyze a cache-oblivious algorithm to find the number of vertical segments intersecting each horizontal segment in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ memory transfers. You may assume that the endpoints of any two different line segments do not have the same x or y value. ▪

Exercise 8.4 What is the maximum number of internal nodes in a B tree that stores n keys (as a function of n and B)? ▪

Exercise 8.5 Develop and analyze a word-RAM data structure to maintain a set of disjoint intervals of the form $[a, b)$ such that $a, b \in \mathscr{U}$. Your data structure should support the following operations in $O(\lg\lg u)$ time:

- make(a, b) : Create the interval $[a, b)$ (must not overlap existing intervals).
- union(a, b, c) : Merge the adjacent intervals $[a, b)$ and $[b, c)$ into $[a, c)$.
- split(a, b, k) : For $k \in [a, b)$, split the interval $[a, b)$ into $[a, k)$ and $[k, b)$.
- find(k) : Return the interval $[a, b)$ that contains k, or report that no interval contains k. ▪

Exercise 8.6 Design a modified version of a B tree in which nodes can have anywhere from B up to $3B$ children (and hence $B - 1$ up to $3B - 1$ keys). Show that this new version of B trees performs only $O(m/B)$ splits, merges, and borrows during a sequence of m operations. (Hint: For this to work, you will have to be more agressive with merging, sometimes merging two nodes before it is strictly necessary.) ▪

Exercise 8.7 Develop and analyze a data structure that supports insert, delete, successor and predecessor in the word-RAM model in $O(\lg\lg u)$ worst-case time. Your data structure should use $O(u)$ bits of space. ▪

Chapter 9

Distributed Data Structures (DDSs)

In this chapter, different types of distributed data structures are presented. Distributed architecture and the difficulties of implementation in that environment are described in the first section. Later sections discuss distributed hashing, lists, trees, and skiplists. The chapter also covers theoretical details, current state of the art, and open questions.

The evolution of the Internet and other advanced technologies led to the development of network computing. This framework lets powerful, low-cost workstations connect quickly via terabytes of memory, petabytes of disk space, and groups of processing units. Every node in a network is either a client or server, based on whether it accesses or manages data.

Every server provides storage space in the form of buckets, to keep a portion of the file under maintenance. Servers communicate by sending and receiving point-to-point messages. The network performing the communications is assumed to be error-free. The efficiency of the system is judged by the number of messages communicated by the servers regardless of message length or network topology.

The algorithms and data structures are specified as distributed algorithms and distributed data structure respectively. The distributed data structures are designed and implemented to allow easy addition of new servers to balance loads on a particular server. The access and maintenance responsibilities require atomic updates to multiple machines. A data structure that meets these constrains is generally known as Distributed Data Structure (DDS).

9.1 Descriptions of Structures

A DDS is a self-managing storage layer developed to run on a collection of workstations inter connected by an underlying network. A DDS is designed to take care of high throughput, high concurrency, availability, incremental scalability, and data consistency. Users see the interface of a distributed data structure as a conventional data structure, such as a hash table, tree or a list. The DDS interface hides all of the mechanisms used to access, partition, replicate, scale, and recover data from a user whose only concern is consistent service to meet his or her specific requirements. A user expects all difficulties of managing to be handled by a DDS interface. Databases and file systems have managed storage layers and other durable components for many years. The advantage of a DDS is the level of abstraction it provides to users. A DDS handles access behavior (concurrency and throughput demands) and other requirements based on its expected runtime and the types of failures it can correct.

Objective 9.1 A distributed data structure provide better services transmitted over the Internet by means of a new persistent storage layer of data which is durable, available, consistent, and independent of service constraints. A DDS automates replication, partitions data, and distributes data over servers that ensure high availability and automatic recovery.

9.1.1 Properties of DDS

Strict consistency of DDS: All operations on the elements of DDS are atomic; they complete a step or are rejected. DDS data elements are replicated among services and a client can see one copy of a logical data item. A two-phase procedure maintains coherent replicas to provide single copy interfaces to clients. A DDS does not typically support accessing of multiple elements or operations. Its protocol may be designed to deny multiple access to maintain efficiency and simplicity. The atomic single element updates and coherence provided in DDS are good enough to support various applications.

Fault tolerance of DDS: A distributed system generally remains operational even if an individual server fails. In a large distributed system server failures are expected from time to time, but a DDS must tolerate limited hardware failures and still support access to data elements at all times and provide data availability, generally by using efficient data replications. A server must immediately detect an operational failure or crash of a connected server. This is certainly a reasonable assumption for local networks, but it is unrealistic for globally distributed databases.

9.2 Distributed Hashing

A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. DHTs form an infrastructure that can be used to build more complex services, such as cooperative Web caching, distributed file systems, domain name services, instant messaging, multicast, and also peer-to-peer file sharing and content distribution systems. Notable distributed networks that use DHTs include BitTorrent's distributed tracker, the Coral Content Distribution Network, the Kad network, the Storm botnet, the Tox instant messenger, Freenet and the YaCy search engine.

9.2.1 Structure of distributed hashing

The structure of a DHT can be decomposed into several main components (Figure 9.1). The foundation is an abstract keyspace, such as the set of 160-bit strings. A keyspace partitioning scheme splits ownership of this keyspace among the participating nodes. An overlay network then connects the nodes, allowing them to find the owner of any given key in the keyspace. The keyspace partitioning and overlay network components are described below with the goal of capturing the principal ideas common to most DHTs; many designs differ in the details.

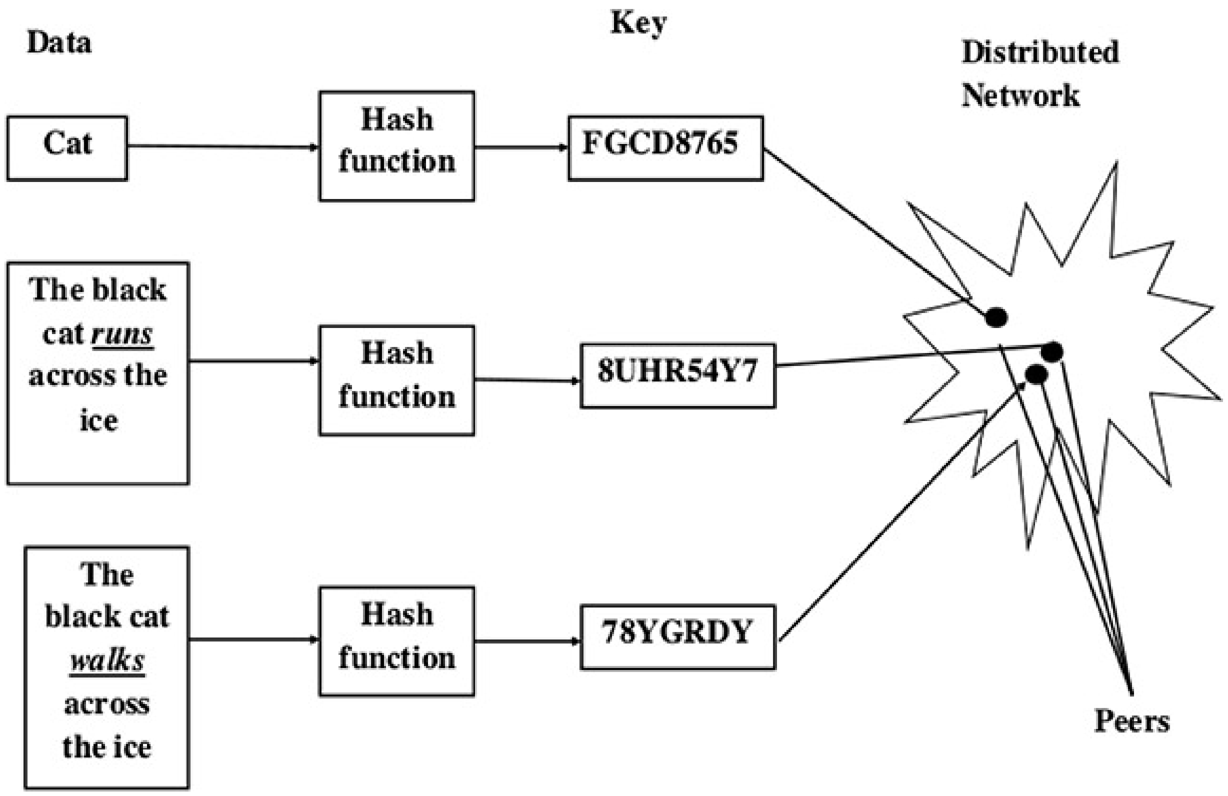


Figure 9.1: Distributed hashing

9.2.1.1 Keyspace partitioning

Most DHTs use some variant of consistent hashing or rendezvous hashing to map keys to nodes. The two algorithms appear to have been devised independently and simultaneously to solve the distributed hash table problem. Both consistent hashing and rendezvous hashing have the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. Contrast this with a traditional hash table in which addition or removal of one bucket causes nearly the entire keyspace to be remapped.

9.2.1.2 Consistent hashing

Consistent hashing is defined as a distance function between the keys. The distance function is independent of geographical distance or network latency.

Each node is assigned a unique key called identifier. A node owns all the keys closest measured according to the distance function. For example, the Chord DHT uses consistent hashing, which treats keys as points on a circle, and is the distance measured clockwise around the circle. Thus, the circular keyspace is split into contiguous segments whose endpoints are the node identifiers.

9.2.1.3 Rendezvous hashing

A rendezvous hashing is a distributed technique in which a common predefined hash function is used by all the clients to a key to some server decided by a common list of server identifiers $\{S_1, S_2, \ldots, S_n\}$ maintained by all clients. For a given key k and the hash function h, the clients compute n hash maps $v_i = h(S_i, k)$ where, $i \in \{1, 2, \ldots, n\}$. The server assignment is determined by the highest value of v_i, i.e. $argmax\ v_i = h(S_i, k)$. For this reason rendezvous hashing sometimes called highest random weight hashing.

9.2.1.4 Overlay network

All the nodes in the overlay network (e.g. Peer to Peer networks) maintains a set of links with its neighbors and nodes in its routing table. The network formed by these set of links is known as an overlay network. DHT is used to create the topological structure (node ID which owns the key (k) or has a link to those nodes who are closer to key (k)). Data sharing in the overlay network becomes easy via DHT. As at every step, data is forwarded to the neighbor node which is closer to the key (k). When the message arrives at the neighbor closed to the k and no other neighbor exists which is closed to the key then the node owns the key. This approach is also known as Greedy approach or key-based routing using DHT.

9.2.1.5 DHT implementation

Notable differences encountered in practical instances of DHT implementations include the following:

- The address space is one of the parameters of DHT and various real-world DHT use either 128-bit or 160-bit key space. Some of the real-world DHTs use hash functions instead of SHA-1. In the real world, DHT keys are the hash of file content instead of file name, so the name does not affect the operation (search).

- Some real world DHT may also publish objects of different types. Like, the key could be the node ID and related data could illustrate how to contact this node. This allows publication-of-presence information and often used in Instant messaging (IM) technology. The node ID is simply a random number that is directly used as a key (so in a 128 bit DHT, node ID will be a 128-bit number chosen randomly).
- To improve reliability in DHT, redundancy can be added. In which the key pair can be stored in more than one node corresponding to the key. Usually, rather than selecting just one node, real-world DHT algorithms selects the 'n' suitable nodes, with 'n' being an implementation-specific parameter of the DHT.
- Some advanced DHTs like Kademlia perform iterative lookups through the DHT first in order to select a set of suitable nodes and send key and data messages only to those nodes. This drastically reduces useless traffic since published messages are sent only to nodes that seem suitable for storing the key. Furthermore, iterative lookups cover only a small set of nodes rather than the entire DHT, thus reducing useless forwarding.

9.2.1.6 Security in DHT

There are three main security issues related to DHTs mentioned in literature. These are called Sybil attacks, Eclipse attacks and routing and storage attacks. In Sybil attacks the idea is that an attacker generates large number of nodes in the network in order to subvert the reputation system or mechanisms based on redundancy.

Eclipse attack is based on poisoning the routing tables of honest nodes. As there are many nodes joining and exiting from the DHT all the time, nodes need to actively update and synchronize their routing tables with their neighbors in order to keep lookup system functional.

In a routing and storage attack, a single node does not follow the protocol. Instead of forwarding the lookup requests, it may drop the messages or pretend that is responsible for the key. Hence, it may provide corrupted or malicious data such as viruses or Trojan horses as a response. Sybil and Eclipse attacks do not directly break the DHT nor damage the other peers.

Security Mechanisms: Some practical examples of security solutions in DHTs are listed below.

1. Sybil attacks: As a defense against Sybil attack, there are several different approaches. Borisov [202 proposes a challenge response protocol based on computational puzzles. The idea is that every node should periodically send computational puzzles to its neighbors. Solving the puzzle proves that the node is honest and trustworthy, but it also requires CPU cycles. The goal is to make organizing Sybil attacks more difficult: running one peer client does not require much CPU power, but running thousands of active virtual nodes is computationally infeasible.

2. Eclipse attacks: An obvious way to shield against Eclipse attacks is to add some redundancy in routing. This approach is utilized by Castro [203 who proposed two routing tables: the optimized routing table and the verified routing table.

3. Routing and storage attacks: Ganesh and Zhao [204 proposed having nodes sign proof-of-life certificates that are distributed to randomly chosen proof managers. A node making a lookup request can request the certificates from the managers and detect possibly malicious nodes.

9.3 Distributed Trees

This section discusses a unique distributed solution for searching problems: the optimal binary search tree (BST) problems presented and analysed. Implemented as a VLSI array, the algorithm for building the optimal BST uses $O(n^2)$ processor and has the parallel time complexity $O(n)$. A search is solved in $O(\log n)$ time. Every site in a network is either a server managing data or a client requesting access to data. Every server provides a storage space of b (bucket) data elements to accommodate a part of the file under maintenance. Sites communicate by sending and receiving point-to-point messages. The distributed algorithms and data structures in such an environment must be designed and implemented so that (a) they expand to new servers gracefully, and only when servers already used are efficiently loaded and (b) their access and maintenance operations never require atomic updates to multiple clients. If all the hypotheses used to efficiently manage search structures in the single processor case are carried over to a distributed environment, then a tight lower bound of $\Omega(n)$ holds for the height of balanced search trees.

9.3.1 Construction of distributed BST

Let T be any binary search tree with 'n' leaves and $n - 1$ internal nodes. Let $f_1, f_2, f_3 .....f_n$ be the leaves and $t_1, t_2, \ldots t_n$ be the internal node. To each leaf a bucket capacity of 'b' storage is associated. Let $s_1, s_2 \ldots s_n$ be the n servers managing the search tree. We can define leaf association in the form of pair (f, s), where 's' represents the server manages the leaf 'f' and the node association is represented in the form of pair (t,s), where 's' represents the server manages internal node 't'. In an equivalent way, we denote the two functions as

1) $t(s_j) = t_i$, where (t_i, s_j) is a node association,
2) $f(s_j) = f_i$, where (f_i, s_j) is a leaf association.

A search tree is binary in that every node represents an interval of the data domain. Moreover, the overall data organization satisfies the invariant that the interval managed by a child node lies inside the father node's interval. Hence the search process visits a child node only if the searched key is inside the father node's interval. It is not possible, in the distributed case, to directly make use of rotations for balancing a distributed search tree while guaranteeing the straight guiding property. A lower bound of $O(n)$ holds for the height of balanced search trees if the straight guiding property has to be satisfied. The relaxed balanced search tree (RBST), upon accepting a violation of the straight guiding property, keeps tree height logarithmic. All update operations have logarithmic cost but the upper bound on the complexity of the search process is $O(log2n)$.

The distributed data structure we focus on is a binary search tree, where data are stored in the leaves and internal nodes contain only routing information. Every node has zero or two children. For a binary search tree T we denote with $h(T)$ the height of T, that is the number of internal nodes on a longest path from the root to a leaf. Every server s but one, with leaf node association (t, s) and leaf association (f, s), records at least the following information:

1. An internal node $t = t\ (s)$ and the associated interval of key's domain $I\ (t)$.
2. The server $p(s)$ managing the parent node $pn(t)$ of t, if t is not the root node.
3. The server $l(s)$ (resp., $r\ (s)$) managing the left child l$s(t)$ (resp., right child $rs(t)$) of t, and the associated interval $I\ (t)$ (resp., $I\ (t)$).
4. A leaf $f = f\ (s)$ and the associated interval of key's domain $I\ (f)$.
5. The server $pf\ (s)$ managing the father node $pn(f)$ of f, if f is not the unique node of global tree (initial situation).

This information constitutes the local tree lt (s) of server s. Since in a global tree of n nodes there are $n - 1$ internal nodes, there is one server $s0$ managing only a leaf association, hence lt (s') is made up by only the two last pieces of information in the above list. We say a server s is pertinent for a key k, if s manages the bucket to which k belongs, in our case, if $kcI(f\ (s))$. Moreover we say a server s is logically pertinent for a key k, if k is in the key interval of the internal node associated to s, that is if $kcI(t(s))$.

9.3.2 Insertion

Step 1: Insert: We search for the leaf where the new key has to be inserted and insert it. We assume that this insert generates an overflow, that is the key to be inserted is the $(b + 1)$th key assigned to that bucket.

Step 2: Manage the overflow: Leaf f, managed by server s, goes into overflow. In this case we have to decide whether s has to be split or if it is possible to transfer its keys to adjacent nodes. Details about this aspect have been discussed in the previous section. Assume then the decision was to split the node. Then s must perform a function called split. Leaf f splits in two new leaves $f1$ and $f2$. A new internal node $t + 1$ replaces f in the tree. A new server $s + 1$ is called to manage the new internal node and one of the new leaves. Server s releases the old leaf f and manages the other new leaf. In conclusion we delete leaf association (f, s) and add two leaf associations $(f1, s)$ and $(f2, s + 1)$ and one node association $(t + 1, s + 1)$ (Figure 9.2). The old interval $I(f)$ is divided in the new intervals $I(f1)$ and $I(f2)$, such that $I(f1) [I(f2) = I(f)]$.

Stpe 3: Balance the distributed BST function starting from $t + 1$.

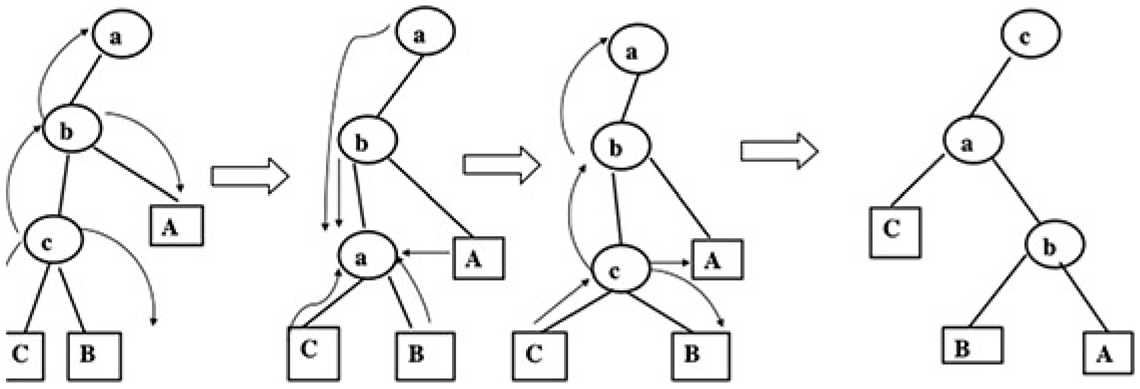


Figure 9.2: Distributed BST

9.3.3 Deletion

Stpe 1: Delete: We search for the leaf where the key has to be deleted and delete it. We assume that this generates an underflow, that is, by deleting that key the bucket has fewer than two keys.

Step 2: Manage the underflow: The leaf f, managed by server s, goes into overflow. In this case we have to decide whether s has to be released or if it is possible to transfer keys from the adjacent leaves, without releasing s. Details on this aspect have been discussed in Section 9.3.4. Assume then the decision was to release s. Then s performs a function called merge. If f is the root, the distributed BST is composed by one node, no action IS performed. If the distributed BST is composed by the root r and two leaves f and x, there are only two servers s and $s0$. Then s is released and after the communication to $s0$ and the deletion of r, x becomes the root. All the keys of f are sent to x. In the general case f is the leaf in figure 9.2. The case with f as left child is analogous. We assume b is the server such that $t(b)$ is the father node of $f(s)$

and c is the server such that $t(c)$ is the father node of $t(b)$. Note that $t(a)$ can be a leaf or an internal node. The steps to achieve are

1. Release server s and delete leaf $f = f(s)$.
2. Since node $t(b)$ has now one child, then delete $t(b)$ and replace it with $t(a)$ as the child of $t(c)$.
3. If s managed an internal node $t = t(s)$, then from now on t is managed by server b (note that b has just released its internal node $t(b)$).

Step 3: Balance the distributed BST by starting the balancing function at $t(c)$.

9.3.4 Rotation

Rotations in a distributed environment are performed via message exchanges between servers. Since we are in a concurrency framework, in the sense that various clients independently manipulate the structure, each rotation must be preceded by a lock of the servers involved. We can show that the cost of one rotation is a constant and then if a balancing strategy uses a logarithmic number of rotations for operation, the overall cost is kept logarithmic. Suppose that node a must rotate with node *b*.

1. *a* sends messages to (client) nodes A, B and to (server) node *b*, to notify that a lock must be created. After having received these messages, nodes A, B, and *b* stop routing messages towards *a* and send a lock acknowledgement to *a*.
2. *b* sends messages to (client) node C and to (server) node *c*, to notify that a lock must be created and that acknowledgement must be sent to *a*. After this message, nodes C and *c* stop routing messages towards *b*.
3. Every server answers to *a* (see second sketch of Figure 9.2) to acknowledge the lock state.
4. a notices to all servers involved in the rotation which modifications are needed and after all servers have been confirmed a releases all locks (Figure 9.2, third sketch).

5. When locks are released the situation is shown in the rightmost sketch of Figure 9.2 and all servers restart to route messages.

9.4 Skip Graphs

Skip graphs are novel distributed data structures based on skiplists, that provide the full functionality of a balanced tree in a distributed system where resources are stored in separate nodes that may fail at any time. Skip graphs are designed for use in searching peer-to-peer systems. By providing the ability to perform queries based on key ordering, they improve on existing search tools that provide only hash table functionality. Unlike skiplists or other tree data structures, skip graphs are highly resilient, tolerating a large fraction of failed nodes without losing connectivity. In addition, simple algorithms can be used to construct a skip graph, insert new nodes into it, search it, and detect and repair errors from node failures.

9.4.1 Design

As in a skiplist, each of the n nodes in a skip graph (Figure 9.3) is a member of multiple linked lists. The level 0 list consists of all nodes in sequence. Where a skip graph is distinguished from a skiplist is that there may be many lists at level i, and every node participates in one of these lists, until the nodes are splintered into singletons after $O(\log n)$ levels on average. A skip graph supports search, insert, and delete operations analogous to the corresponding operations for skiplists; indeed, algorithms for skiplists can be applied directly to skip graphs, as a skip graph is equivalent to a collection of n skiplists that happen to share some of their lower levels.

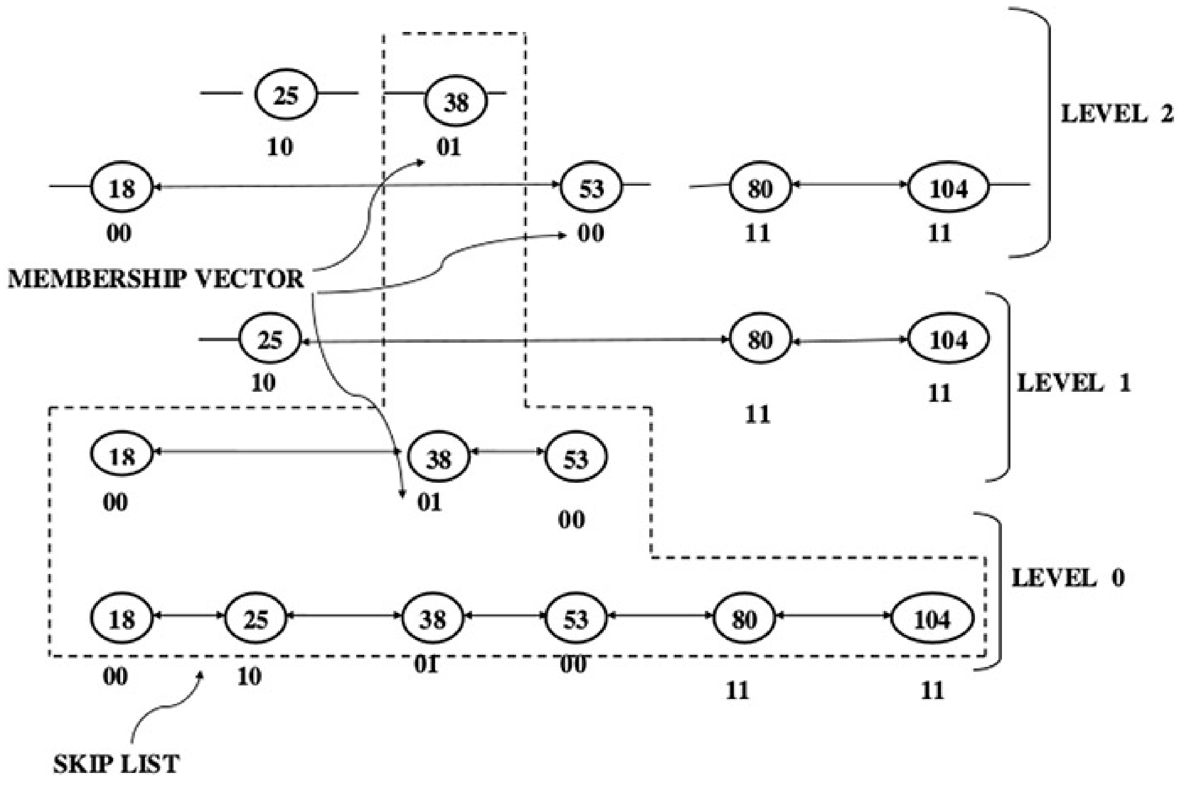


Figure 9.3: Distributed skip graph

Because there are many lists at each level, the chances that any individual node participates in some search is small, eliminating both single points of failure and hot spots. Furthermore, each node has Ω(log *n*) neighbors on average, and with high probability (1 − [1/*p*]log *n*), where *p* is the probability of node failure), no node is isolated. In Section 9.4.5 we observe that skip graphs are resilient to node failures and have an expansion ratio of Ω(*l* log *n*) with *n* nodes in the graph.

In addition to providing fault tolerance, having an Ω(log *n*) degree to support *O*(log *n*) search time appears to be necessary for distributed data structures based on nodes in a one-dimensional space linked by random connections satisfying certain uniformity conditions. While this lower bound requires some independence assumptions that are not satisfied by skip graphs, there is enough similarity between skip graphs and the class of models considered in the bound that an Ω(log *n*) average degree is not surprising.

We now give a formal definition of a skip graph. Precisely the list to which node x belongs is controlled by a membership vector $m(x)$. We think of $m(x)$ as an infinite random word over some fixed alphabet, although in practice, only an $O(log\ n)$ length prefix of $m(x)$ needs to be generated on average. The idea of the membership vector is that every linked list in the skip graph is labeled by some finite word w, and a node x is in the list labeled by w if and only if w is a prefix of $m(x)$.

9.4.2 Search

The search operation is identical to the search in a skiplist with only minor adaptations to run in a distributed system. The search is started at the topmost level of the node seeking a key and it proceeds along each level without overshooting the key, continuing at a lower level if required, until it reaches level 0. Either the address of the node storing the search key, if it exists, or the address of the node storing the largest key smaller than (or the smallest key larger than) the search key is returned. Skip graphs can support range queries to find a key $\geq x$, a key $\leq x$, the largest key $< x$, the least key $> x$, some key in the interval $[x, y]$, all keys in $[x, y]$, and so forth. For most of these queries, the procedure is an obvious modification and runs in $O(log\ n)$ time with $O(log\ n)$ messages. For finding all nodes in an interval, we can use a search operation to find a single element of the interval (which takes $O(log\ n)$ time and $O(log\ n)$ messages). With r nodes in the interval, we can then broadcast the query through all the nodes (which takes $O(log\ r)$ time and $O(rlogn)$ messages). If the originator of the query is capable of processing r simultaneous responses, the entire operation still takes $O(log\ n)$ time.

9.4.3 Insertion

A new node u knows some introducing node v in the network that will help it to join the network. Node u inserts itself in one linked list at each level till it finds itself in a singleton list at the topmost level. The insert operation consists of two stages: (1) Node u starts a search for itself from v to find its neighbors at level 0, and links to them. (2) Node u finds the closest nodes s and y at each level ≥ 0, $s < u < y$, such that $m(u)$ (+ 1) = $m(s)$ (+ 1) = $m(y)$ (+

1), if they exist, and links to them at level +1. Because each existing node v does not require $m(v) + 1$ unless there exists another node u such that $m(v)$ (+ 1) = $m(u)$ (+ 1), it can delay determining its value until a new node arrives asking for its value; thus at any given time only a finite prefix of the membership vector of any node needs to be generated.

In the absence of concurrency, the insert operation in a skip graph S with n nodes takes expected $O(log\ n)$ messages and $O(log\ n)$ time.

With concurrent inserts, the cost of a particular insert may be arbitrarily large. In addition to any costs imposed by the underlying doubly-linked list implementation, there is the possibility of starvation in line 5 as new nodes are inserted faster than u can skip over them. We do not expect this problem to be common in practice. With m machines and n resources in the system, most DHTs such as CAN, Pastry and Tapestry take $O(logm)$ time for insertion; an exception is Chord which takes $O(log2m)$ time. An $O(logm)$ time bound improves on the $O(log\ n)$ bound for skip graphs when m is much smaller than n. However, the cost of this improvement is losing support for complex queries and spatial locality, and the improvement is only a constant factor on machines not capable of storing superpolynomial numbers of resources.

9.4.4 Deletion

The delete operation is very simple. When node u wants to leave the network, it deletes itself in parallel from all lists above level 0 and then deletes itself from level 0. In the absence of concurrency, the delete operation in a skip graph S with n nodes takes expected $O(log\ n)$ messages and $O(1)$ time. We have assumed that each linked-list delete operation takes $O(1)$ messages and $O(1)$ time starting from u; since all but one of these operations proceed in parallel the total time is $O(1)$ while the total messages (summing over all $O(log\ n)$ expected levels) is $O(log\ n)$. The performance of a delete operation in the presence of concurrency depends on the costs of the underlying linked-list delete operation.

9.4.5 Correctness and concurrency

In this subsection, we prove the correctness of the search, insert and delete algorithms described above. We show that both insert and delete maintain the following simple invariant, and then use this invariant to argue that search operations eventually find their target node or correctly report that it is not present in the skip graph.

Note It can be observed that, at any time during the execution of any number of concurrent insert or delete operations, the set of nodes in any higher level of the list is a subset of the set of nodes in the list of level 0.

Consider a search operation with target *x*. If the search operation returns a node with key *x*, then some such node existed in the graph at some time during the execution of the search. If the search operations returns notFound, then there is a time during its execution at which no such node is present. The reason is that *x* is present if and only if it appears in the level 0 list. In order to return notFound, the search algorithm must reach the bottom level without finding *x*. Thus at the time it queries *x*'s level-0 predecessor and finds a successor (or vice versa), *x* is not present in the graph. If the search operation finds *x*, it must send a "found" message back to the source node. This can only occur if *x* has previously received a "found" message, which must have been sent by *x*'s successor or predecessor in some list. At the time this message is sent, *x* is still an element of that list and thus present in the graph. This implies that any search operation can be linearized with respect to inserts and deletes. In effect, the skip graph inherits the atomicity properties of its bottom layer, with upper layers serving only to provide increased efficiency.

9.5 Exercises

Exercise 9.1 Assume that the following solution to the problem of distributing a hash directory is proposed: each node maintains the hash value and location of its successor. Discuss the advantages and disadvantages of this solution, and describe the cost of the dictionary operation (insert, delete, search) and network maintenance operations (join and leave). ▪

Exercise 9.2 Compare index structures presented in this chapter and identify, beyond their differences, some of the common design principles adopted to cope with the distribution problem. ▪

Exercise 9.3 Describe possible solutions to perform distributed range queries. ▪

Exercise 9.4 How we can create a distributed priority queue? ▪

Exercise 9.5 How we can create a distributed stack? ▪

Chapter 10

Synopsis Data Structures

Synopsis data structures are substantially smaller than their base data sets. These are data structures for supporting queries to massive data sets while minimizing or avoiding disk accesses. They have the following advantages:

1. Fast processing: May reside in main memory; provides fast disk processing of queries and data structures updates.
2. Fast swap/transfer: Synopsis data structure that resides in disks can be swapped in and out of memory with minimal disk accesses.
3. Lower cost: Has minimal impact on space requirements of the data set and its supporting data structures.
4. Small surrogate: Can serve a data set when the data set is currently expensive or impossible to access.

Since synopsis data structures are too small to maintain a full characterization of their base data sets, they must summarize the data set, and the responses they provide to queries will typically be approximate ones.

? What synopsis of the full data must be kept in the limited space to maximize the accuracy of response? How can you efficiently compute a synopsis and maintain it during updates to the data set?

10.1 Data Synopsis

There are a number of data sets $S_1, S_2, \ldots, S_l$ and sets of query classes $Q_1, Q_2, \ldots, Q_k$ on these data sets. In the static scenario, the data sets are given as input residing in the disks. Given a class of queries Q, the goal is to design a data structure for the class Q that minimizes the response time to answer queries from Q, maximizes the accuracy and confidence of the answers, and minimizes the processing time needed to build the data structure. In dynamic scenario, which models the ongoing loading of new data into the data set, the data sets arrive online in the memory, and are stored on the disks.

Definition 10.1.1 An $f(n)$ synopsis data structure for a class Q of queries is a data structure for providing answers to queries from Q that uses $f(n) = O(n^{\varepsilon})$ space for a data of size n.

A synopsis data structure can be evaluated according to five metrics:

1. Coverage: the range and importance of the queries in Q
2. Answer quality: the accuracy and confidence of its answers to queries in Q
3. Footprint: its space bound
4. Query time
5. Computation time

10.1.1 Synopsis methods

10.1.1.1 Sampling methods

Sampling methods are among the most simplest methods for synopsis construction in data streams in that they use the same multi-dimensional representation as the original data points.

10.1.1.2 Sketches

Sketch-based methods derive their inspiration from wavelet techniques. In fact, sketch based methods can be considered randomized versions of wavelet techniques, and are among the most space efficient of all methods. However, because of the difficulty of intuitive interpretations of sketch based representations, they are sometimes difficult to apply to arbitrary applications.

10.1.1.3 Histograms

Histogram based methods are widely used for static datasets. However most traditional algorithms on static data sets require super-linear time and space. This is because of the use of dynamic programming techniques for optimal histogram construction. Their extension to the data stream case is a challenging task.

10.1.1.4 Wavelets

Wavelets have traditionally been used in a variety of image and query processing applications. In particular, the dynamic maintenance of the dominant coefficients of the wavelet representation requires some novel algorithmic techniques.

10.1.2 Application

10.1.2.1 Approximate query estimation

Query estimation is possibly the most widely used application of synopsis structures. The problem is particularly important from an efficiency point of view, since queries usually have to be resolved in online time. Therefore, most synopsis methods such as sampling, histograms, wavelets and sketches are usually designed to solve the query estimation problem.

10.1.2.2 Approximate join estimation

The efficient estimation of join size is a particularly challenging problem in streams when the domain of the join attributes is particularly large.

10.1.2.3 Computing aggregates

In many data stream computation problems, it may be desirable to compute aggregate statistics over data streams. Some applications include estimation of frequency counts and quintiles

10.1.2.4 Data mining applications

A variety of data mining applications such as change detection do not require to use of individual data points, but only require a temporal synopsis which provides an overview of the behavior of the stream. Methods such as clustering and sketches can be used for effective change detection in data streams.

10.2 Sampling

Sampling is a utility task used in diverse applications such as data mining, query processing and sensor data management. There are many different algorithms dedicated to effectively building a random sample of a small fixed size whose efficiency varies and depends on how data is stored and accessed. It is however less obvious to incrementally build sample in a single pass or, more generally, to maintain a random sample when the dataset is updated. This is for instance the case when dealing with data streams.

10.2.1 Sampling technique

The right choice of elements of a sample is necessary to achieve good representation of a population. The following issues should be clearly defined.

1. The selection method for the elements of the population (sampling method to be used).
2. Sample size.
3. Reliability of the conclusions and estimates of probable errors.

How can we classify the different ways of choosing a sample?

10.2.1.1 Probability based sampling

In probability sampling, every unit in a population has a chance (greater than zero) of being selected in the sample, and this probability can be accurately determined. The combination of these traits makes it possible to produce unbiased estimates of population totals, by weighting sampled units according to their probability of selection.

▪ **Example 10.1** We want to estimate the total income of adults living in a given street. We visit each household in that street, identify all adults living there, and randomly select one adult from each household. For example, we can allocate each person a random number, generated from a uniform distribution between 0 and 1, and select the person with the highest number in each household. We then interview the selected person and determine his or her income. People living on their own are certain to be selected, so we simply add their income to our estimate of the total. But a person living in a household of two adults has only a one-in-two chance of selection. To reflect this, we would count the selected person's income twice towards the total. The person who is selected from that household can be loosely viewed as also representing the person who isn't selected. ▪

In the above example, not everybody has the same probability of selection; what makes it a probability sample is the fact that each person's probability is known. When every element in the population does have the same probability of selection, this is known as an equal probability of selection (EPS) design. Such designs are also referred to as self-weighting because all sampled units are given the same weight. Probability sampling includes: Simple random sampling, systematic sampling, stratified sampling, probability proportional to size sampling, and cluster or multistage sampling have two common characteristics:

1. Every element has a known nonzero probability of being sampled.
2. Random selection occurs at some point.

10.2.1.2 Non-probability-based sampling

Non-probability-based sampling is any method in which some elements of a population have no chance of selection (these are sometimes referred to as out of coverage or undercovered elements), or where the probability of selection can't be accurately determined. Sampling involves the selection of elements based on assumptions regarding the population of interest, which forms the criteria for selection. Hence, because the selection of elements is nonrandom, nonprobability sampling does not allow the estimation of sampling errors. These conditions give rise to exclusion bias, placing limits on how much information a sample can provide about the population. Information about the relationship between sample and population is limited, making it difficult to extrapolate from the sample to the population.

▪ **Example 10.2** We visit every household in a given street, and interview the first person to answer the door. In any household with more than one occupant, this is a non-probability sample, because some people are more likely to answer the door (e.g. an unemployed person who spends most of his or her time at home is more likely to answer than an employed housemate who might be at work when the interviewer calls) and it's not practical to calculate these probabilities.

Note Probability sampling is useful because it assures that a sample is representative and we can estimate the errors for the sampling.

10.2.2 Reservoir sampling

In this subsection, we discuss simple reservoir sampling algorithm (internal memory), which works as follows: Let t denote the number of the data in the dataset that have been processed. If $t + 1 \leq n$, the $(t + 1)$th data is directly inserted into the reservoir. Otherwise, the data is made a candidate and replaces one of the old candidates in the reservoir with probability $n/(t + 1)$. The replaced data is uniformly selected from the reservoir.

Algorithm 25 Reservoir sampling algorithm

```
1: procedure RESERVOIR($k$, $S$)    ▷ take a random sample from the dataset $S$
2:     initialize reservoir of size $k$
```

```
3:  for $i = 1$ to $i \le |S|$ do
4:    old = $i$-th item
5:    if $i \le k$ then R[$i$]= old
6:    else generate a random integer from 1 to $x$
7:      if $x \le k$ then R[$i$]= old
```

The basic idea behind reservoir algorithms is to select a sample of size $2n$, from which a random sample of size n can be generated. A reservoir algorithm is defined as follows:

Definition 10.2.1 — Reservoir. The first step of any reservoir algorithm is to put the first n records of the file into a reservoir. The rest of the records are processed sequentially; records can be selected for the reservoir only as they are processed.

An algorithm is a reservoir algorithm if it maintains the invariant that after each record is processed a true random sample of size n can be extracted from the current state of the reservoir. At the end of the sequential pass through the file, the final random sample must be extracted from the reservoir. The reservoir might be rather large, and so this process could be expensive. The most efficient reservoir algorithms avoid this step by always maintaining a set of n designated candidates in the reservoir, which form a true random sample of the records processed so far. When a record is chosen for the reservoir, it becomes a candidate and replaces one of the former candidates; at the end of the sequential pass through the file, the current set of n candidates is output as the final sample.

If the internal memory is not large enough to store the n candidates, the algorithm can be modified as follows: The reservoir is stored sequentially on secondary storage; pointers to the current candidates are stored in internal memory in an array, which we call I. (We assume that there is enough space to store n pointers.) Suppose during the algorithm that record R' is chosen as a candidate to replace record R, which is pointed to by $I[k]$. Record R is written sequentially on secondary storage, and $I[k]$ is set to point to R. The above algorithm can be modified by replacing the initial for loop.

Algorithm 26 Reservoir sampling algorithm (external memory)

```
1: procedure RESERVOIR($k$, $S$) ▷ take a random samples(external) from the dataset $S$
2:     for $i$=1 to $i \leq |S|$ do
3:         Copy the $j$th record onto secondary storage;
4:         Set $I[j]$ to point to the $j$th record;
5:         Copy the next record onto secondary storage;
6:         Set $I[H]$ to point to that record        ▷ $H$ is a harmonic coefficient
```

Retrieval of the final sample from secondary storage can be sped up by retrieving the records in sequential order. The sort should be very fast because it can be done in internal memory. The basic idea of Algorithm 25 is to skip data that are not going to be selected, and rather select the index of next data. A random variable $\phi(n, t)$ is defined to be the number of data that are skipped over before the next data is chosen for the reservoir, where n is the size of the sample and t is the number of data items processed so far. This technique reduces the number of data items that need to be processed and thus the number of calls to RANDOM (RANDOM) is a function to generate a uniform random variable between 0 and 1).

Complexity 10.2.1 — Reservoir sampling algorithm. It is clear that Algorithm 26 runs in time $O(N)$ because the entire file must run in time $O(N)$ and be scanned since each record can be processed in constant time. This algorithm can be reformulated easily by using the framework that we develop in the next section, so that the I/O time is reduced from $O(N)$ to $O(n(1 + \log(N/n)))$.

10.2.3 Sampling with updates

As we discussed above, the reservoir algorithm can only produce samples of the insertion-only dataset. Deletions are supported by two algorithms: random pairing (RP) and resizing samples (RS).

10.2.3.1 Random pairing

The basic idea behind random pairing is to avoid accessing the base data set by considering the new insertion as a compensation for the previous deletion. In the long term, every deletion from the data set is eventually compensated by a corresponding insertion. The algorithm maintains two counters c_1 and c_2, which respectively denote the numbers of uncompensated deletions in the sample S and in the base data set R. Initially c_1 and c_2 are both set to 0. If $c_1 + c_2 = 0$, the reservoir algorithm is applied and new data has a probability $c_1 /(c_1 + c_2)$ to be chosen for S; otherwise, it is excluded. Then c_1 and c_2 are modified accordingly. When the transaction consists of a sequence deletion then the last element immediately compensated by an insertion.

10.2.3.2 Resizing samples

The general idea of any resizing algorithm is to generate a sample S of size at most n from the initial dataset R and after some finite transactions of insertions and deletions, produce a sample S of size n from the new base dataset R, where $n < n < |R|$. The proposed algorithm follows this general idea by using a random variable based on the binomial distribution.

10.2.4 Sliding window sampling

In this subsection, we focus on algorithms, which generate a sample of size n from a window of size w. Two types of sliding windows are defined: (i) the sequence-based window and (ii) the timestamp-based window.

10.2.4.1 Simple algorithm

The simple algorithm first generates a sample of size n from the first w data using the reservoir algorithm. Then, the window moves. The sample is maintained until the entry of new data cause old data in the sample to expire. The new data is then inserted into the sample and the expired data is discarded. This algorithm can efficiently maintain a uniform random sample of the sliding window. However, the sample design is reproduced for every tumbling window. If the ith data is in the sample for the current window, the $(i + cw)$th data is guaranteed to be included into the sample at some time in the future, where c is an arbitrary integer constant.

10.2.4.2 Chain-sample algorithm

The chain-sample algorithm generates a sample of size 1 for each chain. So in order to get our sample of size n, n chains need to be maintained. When the ith data enters the window, it is selected to be the sample with probability Min(i,w). If the data is selected, the index of the data w that replaces it when it expires is uniformly chosen from $i + 1$ to $i + w$. When the data with the selected index arrives, the algorithm puts it into the sample and calculates the new replacement index. Thus, a chain of elements that can replace the outdated data is built.

10.3 Sketching

In recent years, significant research has focused on developing compact data structures. Families of such data structures are called sketches. Recent research to develop data structures to summarize massive data streets and compute statistics from such summaries has been ongoing. One statistic that is commonly sought is the total count associated with a key in a data stream, i.e., the sum of the values of keys. A sketch can estimate the count associated with a key in a process known as answering point queries. In networking, sketches have known applications in estimating the size of the largest traffic flows in routers, in detecting changes in traffic streams, in adaptive traffic-sampling techniques, and in worm fingerprinting. More generally, sketches find applications in systems that require online processing of large data sets [145, 146, 144]. Sketches use the same underlying hashing scheme for summarizing data. They differ in how they update hash buckets and use hashed data to derive estimates. Among the different sketches, the one with the best time and space bounds is the count-min sketch.

10.3.1 Count-min sketches

Definition 10.3.1 The count-min (CM) sketch is a compact summary data structure capable of representing high-dimensional vectors and answering queries on such vectors, in particular point queries and dot product queries, with strong accuracy guarantees. Figure 10.1 depicts the structure.

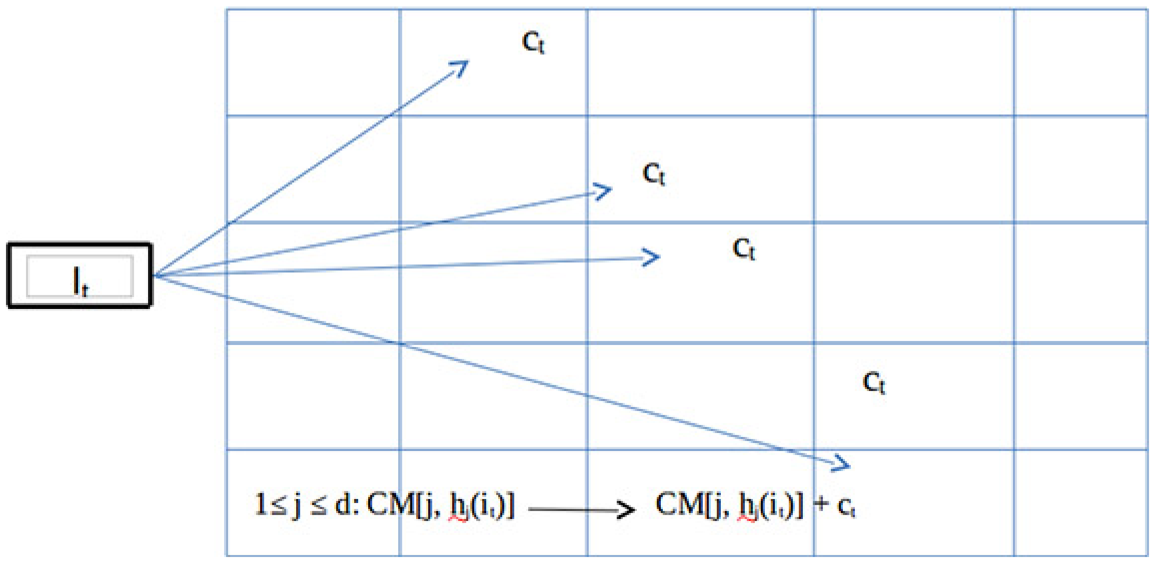

c_t
c_t
c_t
c_t
I_t
$1 \le j \le d$: $CM[j, h_j(i_t)] \longrightarrow CM[j, h_j(i_t)] + c_t$

Figure 10.1: Count-min sketch

Point queries are at the core of many computations, so the structure can be used in order to answer a variety of other queries, such as frequent items (heavy hitters), quintile finding, join size estimation, and more. Since the data structure can easily process updates in the form of additions or subtractions to dimensions of the vector (which may correspond to insertions or deletions, or other transactions), it is capable of working over streams of updates, at high rates.

The data structure maintains the linear projection of the vector with a number of other random vectors. These vectors are defined implicitly by simple hash functions. Increasing the range of the hash functions increases the accuracy of the summary, and increasing the number of hash functions decreases the probability of a bad estimate. These tradeoffs are quantified precisely below. Because of this linearity, CM sketches can be scaled, added and subtracted, to produce summaries of the corresponding scaled and combined vectors. The CM sketch was first proposed in 2003 as an alternative to several other sketch techniques, such as the count sketch and the AMS sketch [145]. The goal was to provide a simple sketch data structure with a precise characterization of the dependence on the input parameters. The sketch has also been viewed as a multistage-filter, which requires only limited independence AND randomness to show strong, provable guarantees. The simplicity of creating and probing the sketch has led to its wide use in many areas [145, 146, 143].

The CM sketch is simply an array of counters of width w and depth d, $CM[1,1],\ldots,CM[d,w]$. Each entry of the array is initially zero. Additionally, d hash functions $h_1,\ldots,h_d:\{1,\ldots,n\}\rightarrow\{1,\ldots w\}$ are chosen uniformly at random from a pairwise independent family. Once w and d are chosen, the space required is fixed: the data structure is represented by $w \times d$ counters and d hash functions.

10.3.1.1 Update procedure

Consider an implicit and incremental vector a of dimension n. Assume that its current state at time t is $a(t)=[a_1(t),\ldots,a_i(t),\ldots a_n(t)]$. Initially, $a(0)$ represents zero vectors, so $a_i(0)$ is 0 for all i. Updates to individual entries of the vector are presented as a stream of pairs. The t^{th} update is (i_t, c_t), meaning that $a_{i_t}(t)=a_{i_t}(t-1)+c_t$ and $a_i(t)=a_i(t-1)$, when $i \neq i_t$.

Note Generally, for convenience of reference, t is dropped, and the current state of the vector is written as just a.

When an update (i_t, c_t) appears, c_t is added to one count in each row of the CM sketch and the counter is determined by h_j.

Complexity 10.3.1 Computing each hash function takes $O(1)$ (constant) time and the total time to perform an update is $O(d)$, independent of w. Since d is typically small in practice (often less than 10), updates can be processed at high speed.

10.3.1.2 Estimation procedure

The process is similar for estimating (i) operations. For each row, the corresponding hash function is applied to i to look up one of the counters.

Across all rows, the estimate is found as the minimum of all the probed counters. In the example above, we examine each place where *i* was mapped by the hash functions. Each of these entries has a counter which has added up all the updates that were mapped there, and the estimate returned is the smallest of these.

Algorithm 27 Initializing array C of $w \times d$ counters to 0, and picking values for hash functions based on prime p.

```
1: procedure CountMinInit ($w$, $d$, $p$)          ⊳ Initialization of CM sketch
2:    $C[d, w] = 0$
3:    for i=1 to d do
4:        Choose $a_j$, $b_j$ and prime p       ⊳ different prime p may be $2^31 - 1$
or else
5:       Total count, N=0
```

Algorithm 28 Updating (i c) by updating N with c.

```
1: procedure CountMinUpdate($i$, $c$)          ⊳ Update of CM sketch
2:    $N = N + c$
3:    for i=1 to d do
4:        $h_j(i) = (a_j \times i + b_j)$ mod p mod w          ⊳ different prime p for
different j
5:        $C[j, h_j(i)] = C[j, h_j(i)] + c$;
NB: The loop hashes its counter in each row and updates it there.
```

Algorithm 29 Estimating (i for given i by performing hashing and tracking minimum value of $C[j, h_j(i)]$ over d values of j

```
1: procedure CountMinEstimate($i$)          ⊳ Estimation of CM sketch
2:    $e \leftarrow 100$
3:    for i=1 to d do
4:        $h_j(i) = (a_j \times i + b_j)$ mod p mod w          ⊳ different prime p for
different j
5:        $e = min(e, C[j, h_j(i)])$
6:    return e
```

Note To speed implementation, mod p can be replaced with a shift and mod w can be replaced with a bitmask operation when w for certain choice of p and w.

Theorem 10.3.1 In a sketch of size $w \times d$ and total count N, any estimate has error at most $2N/w$ with an error probability $1 - (\frac{1}{2})^d$.

Proof. Proof of the theorem can be found in Cormode et al. [144].

R If we set large w and d, we can achieve high accuracy in comperatively little space.

10.4 Fingerprint

This section describes the fingerprint synopsis data structure that maintains a small data footprint while representing it accurately.

Objective 10.1—Fingerprint. Analysis of huge groups of complex objects is now common practice. The fingerprint technique eliminates the "curse of dimensionality" in costly object comparisons by comparing the smallest fingerprint first.

Definition 10.4.1 — Fingerprint. Small fingerprints effectively represent large data objects based on the concept that the probability that two objects have the same fingerprint should be far smaller if the two fingerprints are different from the corresponding objects.

A fingerprinting scheme is a certain collection of functions $F = \{f{:}\Delta \rightarrow \{0, 1\}^k\}$ where Δ is the set of all possible objects of interest and k is the length of the fingerprint.

Complexity 10.4.1 The space complexity of a fingerprint is 2^k.

10.4.1 Fingerprinting scheme of Rabin

Rabin's scheme [152] can produce a simple real-time string matching algorithm and a method for securing files against unauthorized changes. This fingerprinting scheme is based on the arithmetic modulo of a polynomial with coefficient in Z_2. Let $A = (a_1, a_2, \ldots, a_m)$ be a binary string. We associate the string A with a polynomial $A(x)$ of degree m 1 with coefficients in Z_2 as: $A(x) = a_1x^{m-1} + a_2x^{m-2} + \cdots + a_{m-1}x + a_m$ Let $P(x)$ be an irreducible polynomial of degree k, over Z_2. For a fixed P, we define the fingerprint of A to be the polynomial $F(A) = A(x)\ mod\ P(x)$.

Definition 10.4.2 — Irreducible polynomial. A given polynomial $P(x)$ defined on a finite field Z_2 is irreducible if $P(x)$ is not evaluted to 0 by any values from Z_2.

? What will be the value of k, the degree of irreducible polynomial $P(x)$?

Note Any value of k can be used, but implementation if more convenient if we choose k as prime number.

Theorem 10.4.1 Let S be a set on an n string of length m. For an irreducible $P(x)$, the probability that a pair of distinct strings in S has the same fingerprint is $< \frac{n^2m}{2^k}$.

Proof. Let $\Phi_s(x) = \prod_{A \neq B \in S} (A(x) - B(x))$.

Collision implies $\Phi_s(x) = O mod\ P(x)$ by equivalence of string and polynomial.

Therefore, P(x) is a factor of $\Phi_s(x)$.

Degree $\Phi_s(x)$ is $n^2\ m$ and # of irreducible degree k factor of $\Phi_s(x)$ is $\frac{n^2m}{k}$.

The number of irreducible degree k polynomials $>(2^k - 2^{k/2})/k$.

Therefore the probability that a random $P(x)$ divides $\Phi_s(x)$ is $< \frac{n^2m}{2^k}$. ▪

10.4.1.1 Usefull properties for application development

1. $f(A) \neq f(B)$.
2. $Pr(f(A)) = (f(B) \mid A \neq B)$ = very small.
3. The string A and the polynomial $A(x)$ with coefficient over Z_2 are identical.
4. Fingerprinting is distributive over addition (in Z_2) : $f(A + B) = f(A) + f(B)$.
5. Fingerprints can be computed in linear time.
6. The fingerprint of the concatenated strings can be computed as: $f(concat(A, B)) = f(concat\ (f(A), B))$.
7. For f(A) and f(B), and the length n = length (B), we have $f\ (concat(A, B)) = A(x) * x^n + B(x)\ mod\ P(x)$.

Note The operations on polynomials have simple implementations: addition is equivalent to bit-wise exclusive OR. And multiplication by t is equivalent to a one-bit left shift.

Hashing $h \in \omega\{0,1\}^k$	Fingerprinting $f \in \Omega\{0,1\}^k$
Uniform distribution over $\{0,1\}^k$	Distribution free over $\{0,1\}^k$
Bound collisions	Avoid collisions

Table 10.1: Comparison of hashing and fingerprinting

▪ **Example 10.3 — Duplicate detection in large document corpus.** Collections of documents are often very large. Duplicate detection using naive search is very costly. Fingerprinting is a good alternative. It matches fingerprints instead of documents and checks documents for possible duplications only when fingerprint matches are found. ▪

10.5 Wavelets

Wavelets are common mathematical functions often used by database researchers for summarization and decomposition. Wavelet algorithms are capable of processing data at various scales and resolution levels.

Objective 10.2 — Wavelet as synopsis data structure. Wavelets are used to capture crucial information about data such as broad trends and local characteristics of higher and lower order coefficients. The function of the coefficients is to answer queries about data within bounded spaces.

We rely on windows to observe gross characteristics of signals; small windows are used to examine small characteristics. This section discusses the use of Haar 195 wavelets for simple hierarchical decompositions of streaming and other large data collections.

10.5.1 Wavelet decomposition

Wavelet decompositions are special mathematical transforms for capturing data trends in numerical sequences (Figure 10.2). If a few are significant, no need to say that some aren't: Only a few wavelet coefficients of data sets are significant. Only small numbers of significant coefficients are stored to approximate data sequences. Although wavelets represent a vast number of operations, we generally use only basic wavelet decomposition for data structures.

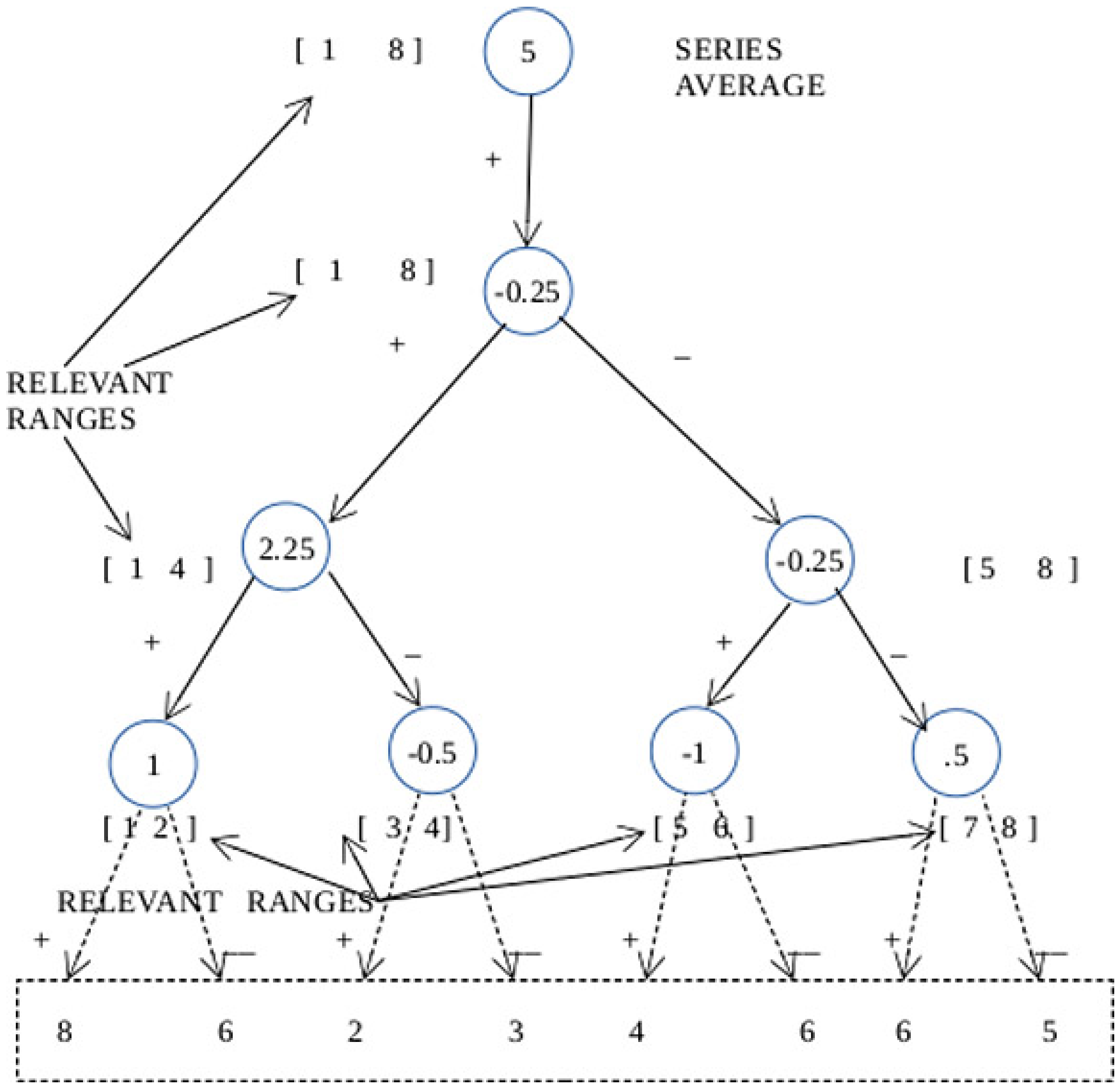


Figure 10.2: Wavelet decomposition

1. The decomposition is computed by convolving the signal with the low pass filter $\{1/\sqrt{2}, 1/\sqrt{2}\}$ and the high pass filter $\{-1/\sqrt{2}, 1/\sqrt{2}\}$ followed by down-sampling by two.
2. In the discrete case if there are N values in the array, this process yields N = 2 averages and N = 2 differences (the averages and differences are scaled by suitable scaling factors).
3. We store the differences as the wavelet coefficients at this level. We repeat the computations of averages and differences until we

are left with only one average and *N*1 differences over *logN* scales or resolutions.

4. The total collection of all the differences over the *log N* scales together with the final average gives the Haar wavelet transform of the input signal.

The entire computation can be quite naturally represented by a binary tree over the signal array, each node in the tree representing the average of the nodes under it and the difference between the left and right child.

▪ **Example 10.4 — Computer vision.** Marr's theory was that image processing in the human visual system has a complicated hierarchical structure that involves several layers of processing. At each processing level, the retinal system provides a visual representation that scales progressively in a geometrical manner. His arguments hinged on the detection of intensity changes. He theorized that intensity changes occur at different scales in an image, so that their optimal detection requires the use of operators of different sizes. He also theorized that sudden intensity changes produce a peak or trough in the first derivative of the image. These two hypotheses require that a vision filter have two characteristics:

1. It should be a differential operator.
2. It should be capable of tuning to act at any desired scale. Marr's operator is now referred to as the Marr wavelet. ▪

▪ **Example 10.5 — Wavelet tree for text compression.** A wavelet data structure was initially proposed for text compression applications and for other uses in text indexing and retrieval. A wavelet tree can simply store texts and provide indexing and compression mechanisms. It can also store auxiliary information for compressed algorithms. Another use is as a general tool to reduce computation needed to compress a string from an arbitrary alphabet to binary strings.

Wavelet trees can be used as a means of reorganizing natural text that has been word-compressed in order to guarantee self-synchronization, even for compression algorithms that are not self-synchronized. Self-synchronization for compressed text permits fast search and random access. The compressed text resulting from this reorganization is synchronized, even for codes that are not self-synchronized. All the advantages (good compression, fast search and random access ability) can be gained simultaneously. The first step is compressing the text and then reorganizing the bytes of all code words in the order in which they appear in the nodes of a structure closely resembling a wavelet tree. ▪

10.6 Exercises

Exercise 10.1 What is a synopsis data structure? What is the need for using synopsis data structures? ▪

Exercise 10.2 Describe the window sampling method and its uses on massive data. ▪

Exercise 10.3 Describe the random sampling method and its uses on massive data. ▪

Exercise 10.4 Describe the reservoir sampling method and its uses on massive data. ▪

Exercise 10.5 Describe the random sketches method and its uses on massive data. ▪

Exercise 10.6 Describe the sketching of frequency moments method and its uses on massive data. ▪

Exercise 10.7 Describe the fingerprint method and its uses on massive data. ▪

Exercise 10.8 Describe the histogram method and its uses on massive data. ▪

Part III

Recent Applications

Chapter 11

Introduction to Applications

Objective 11.1 Part III elucidates the applications of the data structures discussed earlier chapters covering networks, the WWW, DBMS, cryptography, graphics machine learning, operating systems, and other topics. Part III covers recent advances in applications of data structures to all these important computer science areas.

11.1 Various Domain Applications

Chapter 12 addresses applications of data structures to cryptographic techniques. The chapter includes three case studies and describes the use of hash collision as a cryptographic function and hashing for password verification.

Chapter 13 focuses applying data structures to the World Wide Web. It covers uses of tries and skiplists to create inverted indices along with case studies on splay trees, URL hashing for lookup functions, and various string data structures used for semantic alignments.

Applications of data structures to networks and communications are the subjects of Chapter 15. The contents include applications of binomial heaps in network algorithms and acceleration of union and merger operations. Bloom filters for optimizing caches in proxy servers and as summary vectors in ad hoc networks are described. A case study on disjoint sets for greedy algorithms is included.

Chapter 17 details the ways data structures are used for databases, specifically indexing and block searching functions of B trees, B+ trees, and BSTs. The section dedicated to buffer trees explores access paths and other aspects of database management. The final section explains applications of bloom filter join methods for query processing in distributed databases.

Chapter 14 explores machine learning applications such as use of KD tree structures in multidimensional searches, a case study of locality-sensitive hashing (LSH) in nearest neighbor searches and the Nystrom method for generating low-rank matrix operations in learning applications.

Chapter 18 addresses image processing and graphics. It discusses R trees used for spatial indexing and storing geometric and multi-dimensional data in geographic information systems and other trees for handling region searches and predictive queries. The final section covers ray shooting.

Chapter 16 covers applications of data structures in operating and communication systems and includes case studies on caching queues in paging systems, cache size management, and optimizing techniques. It includes case studies on sketching for heavy hitters and sampling and histogram creation for optical signal monitoring.

11.2 Project

Project 11.1 — Artificial mind of a machine Creating an artificial mind for a computing system requires the installation in modules of data structures and algorithms designed to perform the needed functions based on the steps below:

1. Gather data to be processed by machine and store it; design system to perform organized retrieval.
2. Create domain ontology (concept hierarchies).
3. Model interaction of machine with environment.
4. Model interaction of machine with user.
5. Model decision problems to capabilities of machine.
6. Design machine learning algorithm and train it on past instances(may be simulated).
7. Test and re-train adaptively.

Chapter 12

Applications to Cryptography

Cryptography current fulfills a vital function in a world where millions of people use the Internet to exchange personal and business information. It is imperative for companies, banks, governments, and other institutions to ensure security of the networks while providing users with speedy output of information. Confidentiality, data integrity, secrecy, and validation are provided by cryptographic algorithms. A mathematical formula scrambles plain text to produce ciphertext. The conversion achieved by a cryptographic algorithm is known as decryption.

Hash texts (also known as message digests or one-way encryptions) have no keys. A fixed-length hash value is computed based on the plaintext. The major purpose of hash functions in cryptography is message integrity. The hash value affords a digital fingerprint of a message's contents, which ensures that the message has not been changed by an intruder, virus, or by other means.

Objective 12.1 In this chapter, we address the applications of hashing for cryptographic functions. Each section includes notable case studies. The first section discusses the MD5 (message digest 5) algorithm. The next section covers hashing applications to secure socket layers, along with hash collisions and cryptographic hash functions. The chapter concludes with explanations of hashing in block chains and digital signatures.

12.1 MD5

MD5 is a cryptographic algorithm that uses an arbitrary length input to generate a message digest 128 pieces long [12], [155]. The algorithm operates at 128 bits divided into four 32-bit words denoted U,V,W and X. These words are initialized to some fixed constants. Each input message is broken into chunks of 512 bit block (pieces) in turn to change the state (each message block consists of four similar stages, called Cycles). Each round of algorithm consists of 16 similar operations based on a nonlinear function F, a modular addition, and left rotation. There are four possible functions; a different function $\oplus, \wedge, \vee, \neg$ is used in each round to denote the xor, and, or, and not operations respectively.

$$\begin{aligned}
F(B,C,D) &= (B \wedge C) \vee (\neg B \wedge D) \\
G(B,C,D) &= (B \wedge D) \vee (C \wedge \neg D) \\
H(B,C,D) &= B \oplus C \oplus D \\
I(B,C,D) &= C \oplus (B \vee \neg D)
\end{aligned}$$

12.1.1 Password hashing

The algorithms for hash functions have one-way functional behavior. The main function of a hash is to convert any length of data into a fixed-length fingerprint that cannot be reversed. If the input changes even by a tiny bit, the resulting hash is completely different. A hash is designed to protect stored passwords even if their files are compromised. A secure system must also be able to verify user passwords. The list below shows the work flow for account registraation and authentication using a hash-based algorithm.

1. User creates an account.
2. User enters a password that is hashed and stored in the database. After hashing, the password is also converted into a random fixed-length fingerprint and stored on the hard drive.
3. Next time when user attempts to log in, the hash of the password is checked against the hash of the user's real password (retrieved from the database).
4. If the entered value matches with the stored hashes, the user is granted access. If not, the user is advised that he or she entered an invalid login.

5. Steps 3 and 4 repeat whenever a user tries to login to his or her account.

Note Step 4 is designed to display a generic message like "Invalid username or password" if a user enters a wrong password or name. This feature of the operating system prevents attackers from using valid usernames without knowing user passwords.

Note Hash functions used for password protection are different from the hash functions used in a data structure. Hash functions used to execute data structures (e.g. hash tables) are designed to be fast, not secure. The cryptographic hash functions(such as SHA256, SHA512, RipeMD and WHIRLPOOL) are only used for password hashing.

It's easy to believe that all you need to do is run the password through a cryptographic hash function and the passwords of your users will be safe. That assumption is far from the truth. There are many ways to recover simple hash passwords very quickly. There are several easy-to-use techniques that make these attacks much less effective. There are method to retrieve data using various dictionary attacks.

12.2 Secure Socket Layers (SSLs)

The major goal of the SSL [RFC0793] protocol is to provide security between channels. An SSL is composed of two layers. Its function is encapsulation of various higher level protocols. The SSL handshake protocol provides the authentication between the server and client and then negotiates the encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data. One advantage of SSLs is that they are application protocol independent. Their reliability is maintained through message integrity checks using keyed message authentication codes (MACs) [RFC2104] [157, 156, 158].

12.2.1 Data structure of open SSL

The current data structures of open SSL library functions are published on the Internet: (https://www.openssl.org/docs/man1.0.2/ssl/ssl.html):

1. SSL_METHOD describes the internal SSL library methods and functions which implement the various protocol versions (SSLv1, SSLv2, and TLSv1). It's needed to create an SSL_CTX.
2. SSL_CIPHER retains the information of a particular cipher which is a core part of the SSL/TLS protocol. The available ciphers are configured on an SSL_CTX basis and the used ones become part of the SSL_SESSION.
3. SSL_CTX is a global context structure created by a server or client over the lifetime. It also holds mainly default values for the SSL structures which are later created for the connections.
4. SSL_SESSION maintains current TLS/SSL session details for the connection: SSL_CIPHERs, server and client certificates, keys, etc.
5. SSL is the main connection structure of SSL/TLS. It is created by a server or client per established connection. Under run-time, the application usually deals with this structure which has links to most other system structures.

12.3 Block Chains

The distributed and decentralized ledger system is one of the best advancements since the invention of the WWW [159, 160, 161]. Over the years it has found many applications and one of them is "currency" In this section we explore the main data structures found in almost any cryptocurrency based on the block chain technology. A block chain, in general, is a hash pointer-based data structure composed of a block with the following features:

1. Index is the position of the block on the block chain. The first block has a 0 index.
2. The hash function applies to block components. For example, the hash function used in Bitcoin is a variant of the SHA2 with 256 bits (SHA256). In Ethereum, SHA3 is used.

3. The previousHash function links a block to its predecessor.
4. The Unix UTC timestamp shows when a block was created.
5. The nonce is a 32- to 64-bit integer used in data mining.
6. numTx indicates the number of transactions in a block.
7. The transactions feature is an array of all the transactions found in a block.

Note For transactions data, some implementations use Merkle trees for space optimization.

Blocks in a block chain contain valid transactions that are encoded and hashed by Merkle trees. Merkle tree is also known as hash tree, where leaf node is labeled with hash of data block and non leaf nodes are labelled with cryptographic hash. Markel tree helps in efficient and secure verification of large data structure (as illustrated in Figure 12.2). Each block has the cryptographic hash value of the previous block in the block that unites the two. The connected blocks cascade to form a chain. This iterative process confirms the integrity of the previous block to the original generation block [162]. Sometimes, the blocks can be produced at the same time, creating a temporary bifurcation. Each block has a specific algorithm to evaluate different versions of the history, so a higher value can be chosen. Blocks not selected for inclusion in the chain are called “orphans”.

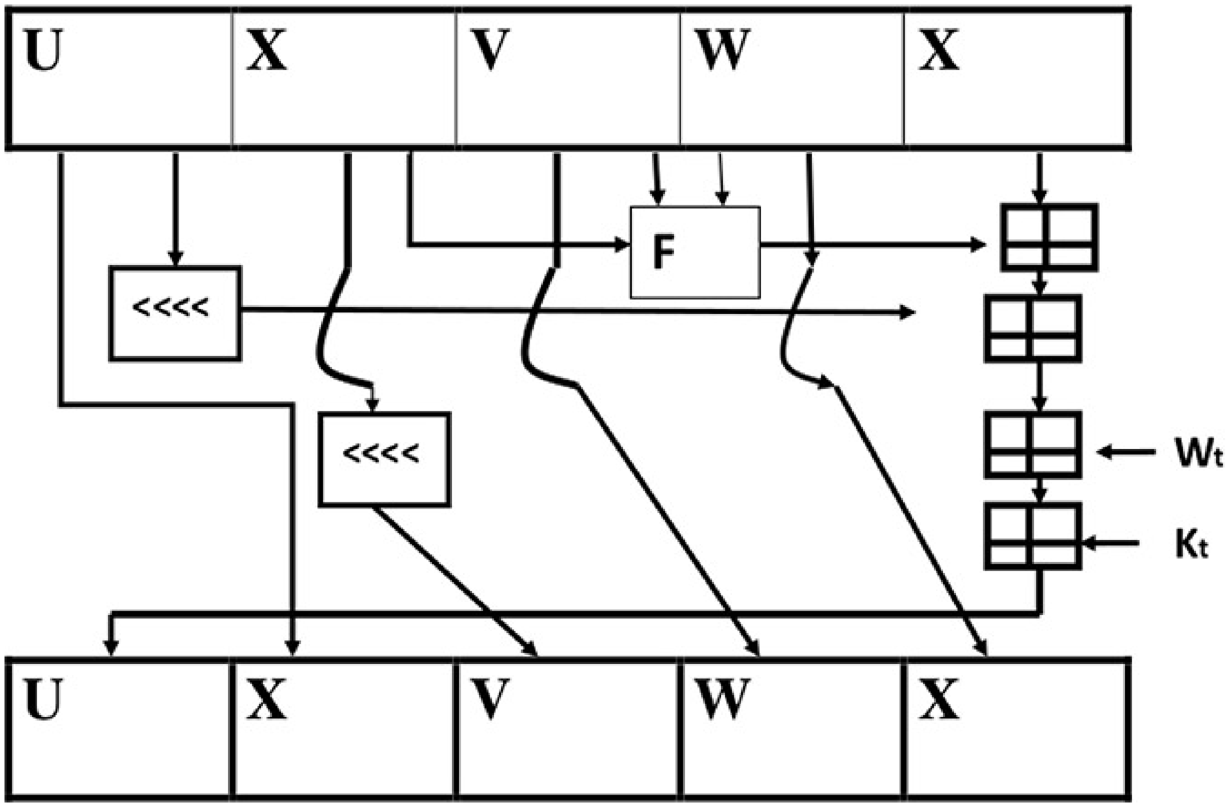


Figure 12.1: MD5

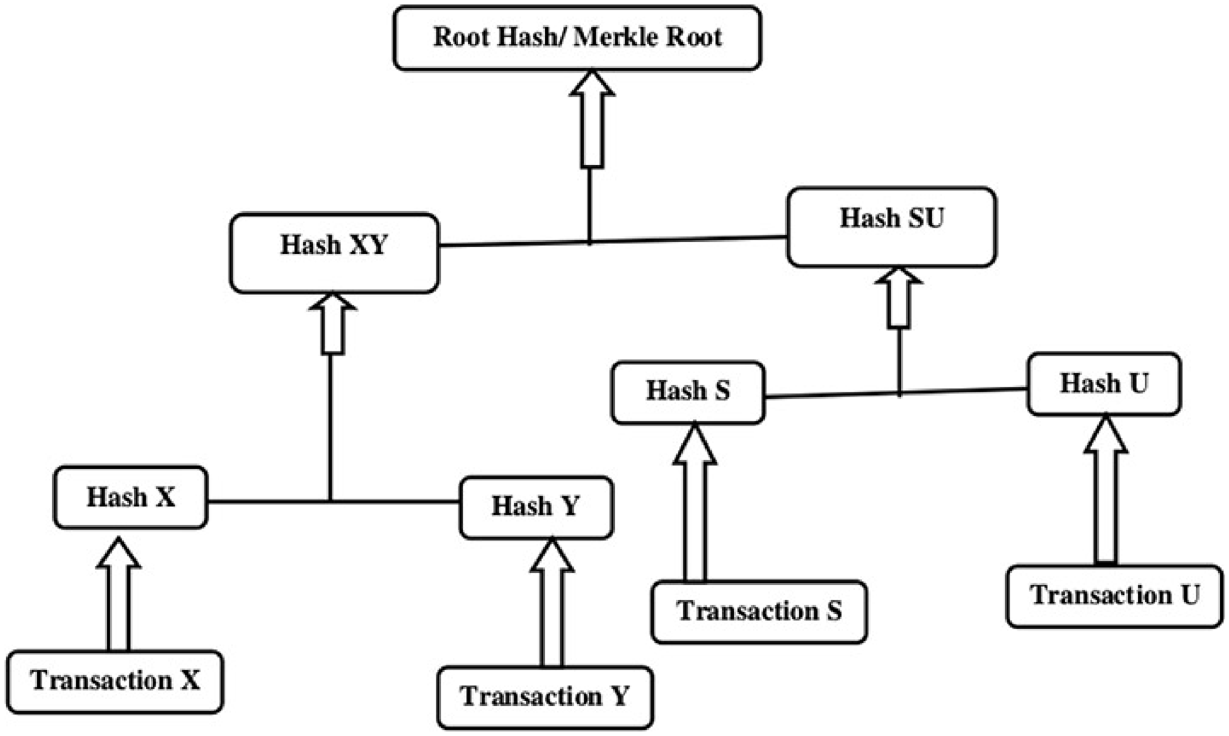


Figure 12.2: Merkle tree of transactions X, Y, S, and U.

The partners that support the database have different versions of the story from time to time. They maintain the highest version of the database they know. Whenever a block receives a higher version of score it extends or overwrites its own database and retransmits the improvement to its peers. There is never an absolute guarantee that a particular item will remain forever in a story.

Block chains are generally created to add new blocks to old blocks. Built-in incentives motivate block chains to extend new blocks instead of overwriting old blocks. The probability of replacing an entry decreases exponentially as more blocks are built. For example, in a block chain using the proof-of-work system, the chain with the most cumulative proof-of-work is always considered the valid one by the network. There are a number of methods that can be used to demonstrate a sufficient level of computation. Within a block chain the computation is carried out redundantly rather than in the traditional segregated and parallel manner [163].

12.4 Digital Signature

The unique characteristics of a hand-written signature are not easily imitated and thus allow a person to conduct business without having his or her identity questioned. Those characteristics allow a signature to be verified as genuine or identified as a forgery [165, 166].

A digital signature works like an electronic stamp or fingerprint; it is the electronic equivalent of a hand-written signature. The major difference between digital and hand-written signatures is that a digital signature changes on every use even if the signer and key pairs are the same. A digital signature authenticates data origins and protects data integrity. An example of how a hash function of a digital signature works is shown below.

1. Assume Sarah is the sender and signer of a document. She has the private and public key pair, the hash function for creating the message digest, and the document.
2. Remy is the recipient of the document. Sarah starts the digital process by generating the hash value of the message of document to be transmitted to Remy.

3. Sarah uses her private key to encrypt the message digest to produce the signature.
4. Sarah appends the digital signature to the document.
5. Finally, she encrypts the signed document with her private key and transmits it to Remy.
6. Remy receives the ciphertext and decrypts it using Sarah's public key to access the signed document [164].

Summary 12.1 The advanced hashing techniques are covered in Chapter 2. This chapter discusses four important cryptographic applications: hashing in SSLs, hashing using the MD5 algorithm, block chain hashing, and method to hash digital signatures; a practice project is presented below.

12.5 Projects

Project 12.1 — Graphical password strategy. Assume you want to maximize password space while facilitating memorization of entered secrets. [Hint: use graphical: Use a graphical password system along with the hash function because a graphical password system is considered difficult to crack by brute force, search, dictionary, social engineering, and spyware attacks.]

Project 12.2 — File encryption using Fibonacci series. Use the Fibonacci series technique to encrypt and decrypt a file. [Hint: Start the Fibonacci series from 1 instead of 0. The element which is at the odd position in the ciphertext is forwarded by the current Fibonacci term.]

Project 12.3 — Hybrid AES DES encryption algorithm. The advanced encryption standard (AES) and the data encryption standard (DES) are used to encrypt and transfer data. Combine both algorithms in an efficient structure to create a strong encryption algorithm.

Project 12.4 — Mobile Self Encryption Project. Use a stream cipher to encrypt data on a mobile phone. The key is stored on a server, If a user loses the phone, he reports the loss to the server that then destroys the key and phone data remains confidential.

Chapter 13

Application to IR and WWW

Web-based applications have increased enormously in the last few years. Use of search engines, online simulations, language translators, games, AI-based Web pages continues to grow and new technology continues to develop. Advances such as crawl frontiers, posting list intersections, text retrievals via inverted indices, and autocomplete functions using tries have revolutionized Web access. This chapter summarizes some of the methods and their corresponding data structures.

Objective 13.1 This chapter addresses the applications of priority queues (crawl frontiers), skiplists, skip pointers and tries. It contains case studies of priority queues and posting list intersections. The final sections cover text retrieval through inverted indices and autocomplete using tries.

13.1 Crawl Frontier

Crawl frontier is a data structure (also known as a priority queue) designed to store URLs and support their addition and crawl selection functions. The crawl frontier on a node receives a URL from its crawl feature or a host separator from another crawl. The crawl frontier retains the URLs and releases them in a certain order whenever a search thread seeks a URL.

Two important issues govern the order in which the frontier returns the URLs:

1. Pages that change frequently are given priority for frequent scans; page priority combines quality and rate of change. The combination

is essential because each search means processing large numbers of changing pages.

2. Politeness [172] is another crawl frontier feature. Repeated fetch requests in a short period should be avoided, primarily because the requests impact many URLs linked on a single host. A URL frontier implemented as a simple priority queue may cause a burst of fetch requests to a host.

This scenario may even occur if a constraint is imposed on a crawler so that at most one thread could fetch from any single host at any time. A heuristic approach is to insert a gap between two consecutive fetch requests to that host which has an order of magnitude larger than the time taken for the most recent fetch from that host. Figures 13.1 and 13.2 depict details of a crawl frontier [169]–[171], [174].

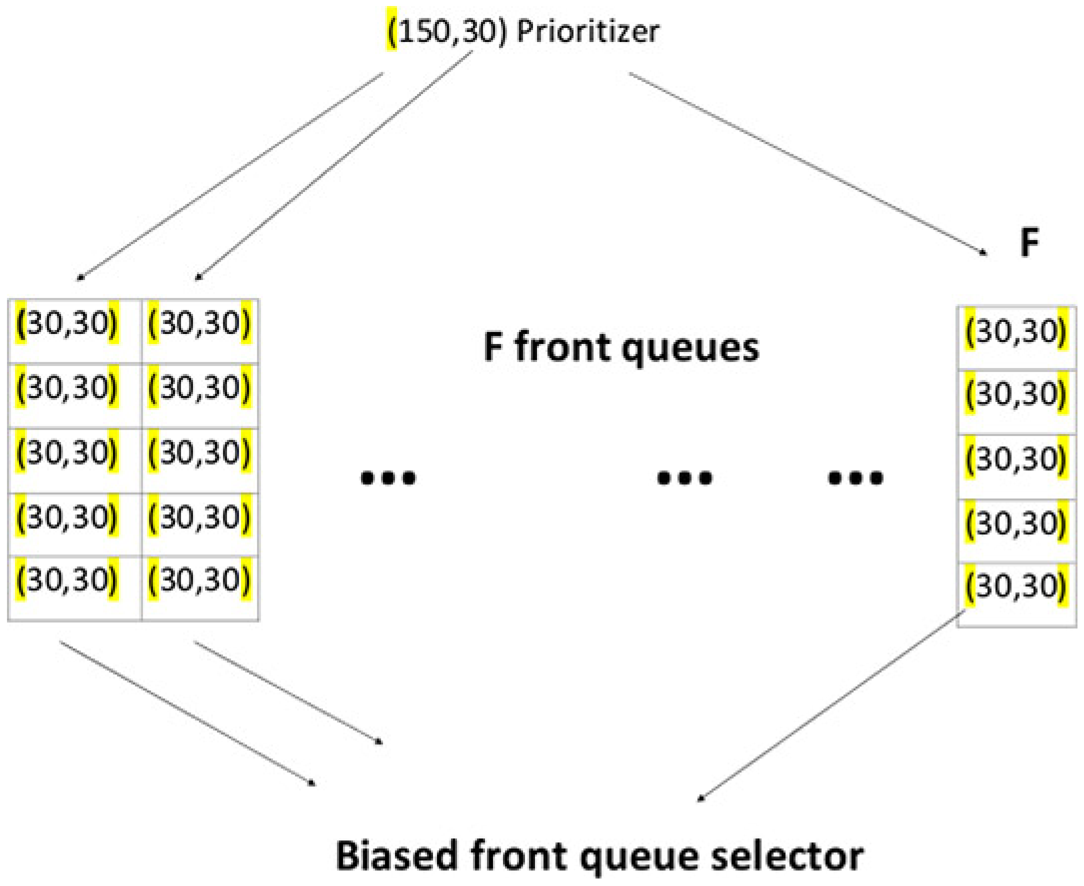


Figure 13.1: Biased front queue of crawl frontier

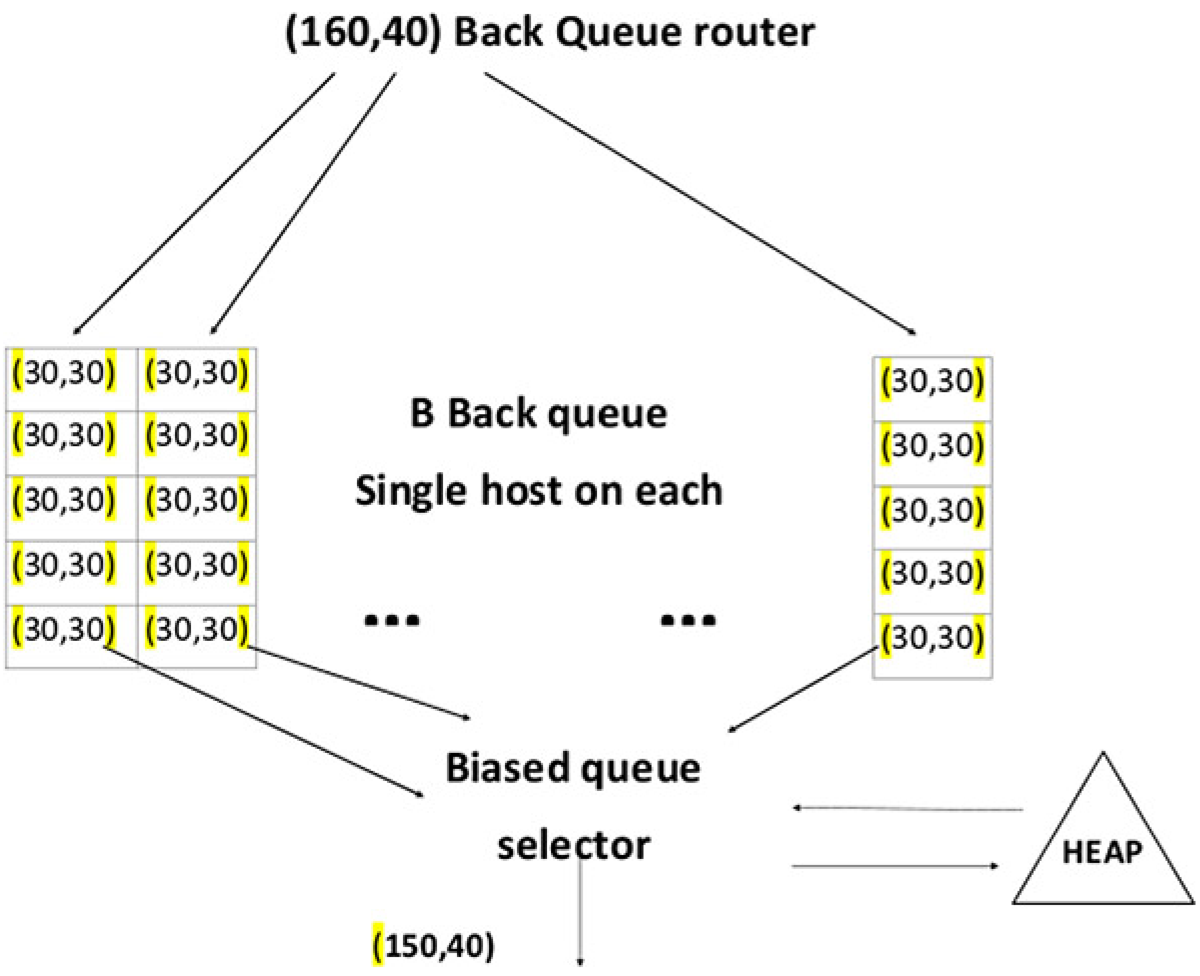


Figure 13.2: Back queue of crawl frontier

13.2 Posting List Intersection

The basic operation of a posting list intersection is to walk through the two postings lists, in linear time within a total number of postings entries. If the list lengths of entries are m and n, then the intersection will take $O(m + n)$ operations. Now the question is, can it be done better than this? Can the process postings list intersection be done in sublinear time? The answer to both questions is yes provided the indexes are not changed frequently. One way to deal with it to use skip list (augmenting postings lists with skip pointers) as illustrated in Figure 13.2. Skip pointers (Figure 13.3) avoid processing but the issues of placing and merging them must also be handled. Information about operations of skiplists and skip pointers can be found at https://nlp.stanford.edu/IR-book/html/htmledition/faster-postings-list-intersection-via-skip-pointers-1.html.

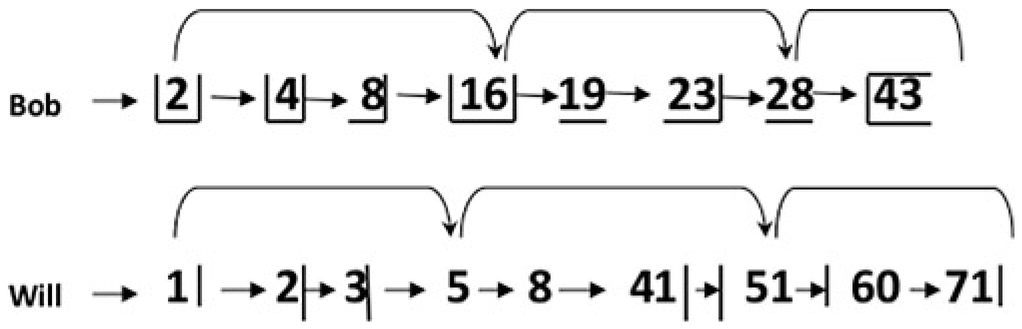


Figure 13.3: Data structure of skip pointer

13.3 Text Retrieval from Inverted Index

Retrieval involves a search for lists postings corresponding to query terms (document score and return of results to the user). Searching for postings involves the random search of the disk, since postings may be too large to fit in memory (without considering cached storage and other special cases). Searching at this level is beyond the capability of many systems. The best solution is breaking down indices and distributing retrieval results over a number of machines.

The two main partitioning approaches for distributed data retrieval are document partitioning and term partitioning. In document partitioning, the entire collection is broken into multiple sub-collections and then assigned to the server. In term partitioning, each server is responsible for the subset of the entire collection, i.e., the server holds the postings for all documents in the collection for the subset of terms. The two techniques use different retrieval techniques and present different trade-offs. Document partitioning uses a query broker that forwards a user query to all partition servers, then merge the partitions results and returns them. Document partitioning typically yields shorter query latencies as they operate independently and they traverse postings in parallel.

13.4 Auto Complete Using Tries

Auto complete functionality is used widely in mobile applications and text editor. A trie is an efficient data structure commonly used to implement auto complete functionality. Trie provides an easy way to search for the possible dictionary words to complete a query for the following reasons. Looking up data in a trie is faster in the worst case O(n) (n = size of the string involved in the operation) time compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. A trie can provide an alphabetical ordering of the entries by key. Searching in a trie helps trace pointers to reach a node entered by the user. By exploring a trie traversing down the tree, we can easily enumerate all strings that complete the user input. The steps for resolving a query are:

1. Search for given query using the standard trie search algorithm.
2. If query prefix is not present, indicate it by returning −1.
3. If a query is the end of a word in trie, print the query. The end of the query can be checked quickly by checking whether the last matching node has an end word flat. Tries use such flags to mark the ends of word nodes during searches.
4. If last matching node of query has no children, return.
5. Otherwise, recursively print all nodes under subtree of last matching node.

13.5 Projects

Project 13.1 — Document Sentiment Analysis. The aim of sentiment analysis is to detect the attitude of a writer, speaker or another subject with respect to some topic or context. Sentiment analysis utilizes natural language processing to extract subjective information from the content. Write a project for document sentiment analysis. The project should involve scanning of user documents and breaking down of comments to check the sentiment in the documents. If the keywords are found, the comments contain sentiment. [Hint: for morphological analysis, tries can be useful.]

Project 13.2 — Twitter Trend Analysis. The hashtag (#) has become an important medium for sharing common interests. Enough individual participation in social media eventually creates a social trend. Through the social trends government and other institutions get an idea of the popularity of any event or activity in a specific moment. Consider a Twitter microblogging site and introduce a typology that includes the following four types of uses: news, ongoing events, memes, and commemoratives. Input will be given by the user in the form of a keyword that will be used to search for latest trends. The system will search for the similar words in the database and it will summarize the total count of the trending tweets associated with the keyword. The summarized trending tweets will be displayed on the screen.

Project 13.3 — Website Evaluation Using Opinion Mining. Propose an advanced website evaluation system for an electronic commerce company. The project must rate the company's website based on user and customer opinions. The evaluation criteria should consider the following factors: (1) timely delivery of product; (2) website support; (3) accuracy and scope of the website. The proposed system should rate the website by evaluating the opinions and comments of users and customers. [Hint: use opinion mining methodology and database of sentiment based keywords. Positive and negative weights in the database are evaluated and sentiment keywords of users are used for ranking].

Project 13.4 — Detecting E Banking Phishing Websites. The number of online banking users has increased greatly over the past few years. Banks that provide online transactions ask their customers to submit certain credentials (user name, password, credit card data). Malicious users may try to obtain credentialing information by disguising themselves as trustworthy entities; their operations are known as phishing websites.

Propose a system based on a classification data mining algorithm to predict and detect e-banking phishing websites. The system should detect

URL, domain identity, security features, and encryption criteria characteristics. [Hint: Read the 2020 paper of Aburrous et al. [196].]

Chapter 14

Applications to Data Science

Data sciences represent smart world and artificial intelligence of the future. They blend the areas of data inference, algorithm development, and electronic advances to solve complex problems and add value in all areas of government and commerce.

Objective 14.1 This chapter covers the applications of important concepts of synopsis data structures and various hashing techniques for analysis of large amounts of data in reasonable time.

14.1 Heavy Hitters and Count-Min Structures

A paper by Cormode et al. [186] describes the count-min sketch data structure that does more than approximating data distributions. The functions are described below.

- A heavy hitter query (HH(k) request) seeks a set of high frequency elements (say $\frac{1}{k}$ of the total frequency). Count tracking can be used to answer the query directly, by considering the frequency of each item. When there are many possible elements, the query response can be significantly slower. At the expense of collecting additional groups, the process can be speeded up keeping additional information about the frequencies of the element groups [186].
- Finding heavy-hitters is also of interest in the context of signal processing. Since data distribution is defined by signals, the recovery of heavy goods is the key to the best signal convergence. Consequently

the graph-min function can be used in a compressed sensor paradigm to obtain a recently processed signal [187].

- If $p \notin B(q, r_2)$ then $Pr_H[h(q) = h(p)] \leq p_2$.
- Applications like no loopy peas (NLP) generate large amounts of data. Therefore, it is important to store statistics by combining words, such as pairs or triple words that appear in the sequence. In one experiment, the researchers compacted a 90 GB data load to 8 GB graphic sketches.
- Another use of a count-min sketch is in password design. The count-min structure can be used for count tracking (see http://www.youtube.com/watch?v=qo1cOJFEF0U). The good feature with it is the impact of a false positive.

14.2 Approximate Nearest Neighbor Searches

The nearest neighbor (NN) search is an important step in stuctural analysis and processing of other types of queries. NN search is very useful for dealing with massive data sets, but it suffers from the curse of dimension [34, 36]. NN searches were previously used for low dimensional data. A recent surge of results shows that the NN search is also useful for analyzing large data collections if a suitable data structure (KD tree, quadtree, R tree, metric tree, locality sensitive hashing (LSH)) is used [42, 41, 40, 38]. One more advantage of using NN search for large data analysis is the availability of efficient approximation scheme, which provides almost the same results in very less time [49, 37].

14.2.1 Approximate nearest neighbor

Consider a metric space (S, d) and some finite subset S_D of data points $S_D \subset S$ on which the nearest neighbor queries are to be made. The aim of approximate NN to organize S_D such that answer the NN queries can be done more efficiently. For any $q \in S$, the NN problem consists of finding a single minimal located point $p \in S_D$ s.t. $d(p, q)$ minimum over all $p \in S_D$. We denote this by $p = NN(q, S_D)$.

An ε approximate NN of $q \in S$ is to find a point $p \in S_D$ s.t. $d(p, q) \leq (1 + \varepsilon)\, d(x, d)\ \forall\, x \in S_D$.

14.2.2 Locality-sensitive hashing (LSH)

Several methods to compute first nearest neighbor query are cited in the literature and locality-sensitive hashing (LSH) is most popular because of its dimension independent run time [46, 45]. In a locality sensitive hashing, the hash function has the property that close points are hashed into same bucket with high probability and distance points are hashed into same bucket with low probability. Mathematically, a family $H = \{h : S \rightarrow U\}$ is called (r_1, r_2, p_1, p_2)-sensitive if for any $p, q \in S$

- if $p \in B(q, r_1)$ then $Pr_H[h(q) = h(p)] \geq p_1$
- if $p \notin B(q, r_2)$ then $Pr_H[h(q) = h(p)] \leq p_2$

where $B(q, r)$ denotes a hypersphere of radius r centered at q. In order for a locality-sensitive family to be useful, it has to satisfy inequalities $p_1 > p_2$ and $r_1 < r_2$ when D is a distance, or $p_1 > p_2$ and $r_1 > r_2$ when D is a similarity measure [27, 29]. The value of $\delta = log(1/P_1)/log(1/P_2)$ determines search performance of LSH. Defining a LSH as a $(r, r(1 + \varepsilon), p1, p2)$, the $(1 + \varepsilon)$ NN problem can be solved via hashing and searching within the buckets [44, 43, 40].

14.3 Low Rank Approximation by Sampling

Traditionally in information retrieval and machine learning, data is represented in the form of vectors. These vector collections are then stored in the single metric $A \in R^{n \times m}$, where each column of A corresponds to a vector in the n dimensional space. The advantage of using a vector space model is that its paradigm can be exploited for the solution [205]. However, the information contained in the data must not be removed. The widely used method for this purpose is to estimate a single data matrix, A with a lower rank matrix. Mathematically, according to the Frobenius criteria, the optimum rank estimate of the matrix A can be computed as follows: Find a matrix $B \in R^{n \times m}$ with rank $(B) = k.$, such that $\|A - B\|$ is minimum. The matrix B can be readily obtained by computing the singular value decomposition (SVD) of A, as stated by Golub and Van Loan [206]. For any approximation M of A, we call $\|A - M\|_F$ the reconstruction error of the approximation.

SVD calculation is interesting from a theoretical view because it provides the closest matrix of a given rank. For many applications where the data matrix is large, the SVD calculation can be impractical because it requires a large number of operations and large memory. Recent studies have focused on algorithms which are not optimal in the sense that they compute a lower-grade matrix which is not close to the original matrix. Reported work shows that they have an advantage over SVD based algorithms as they require less memory. Low-rank approximations have various applications like latent semantic indexing, support vector machine training, machine learning, computer vision, and web search models. In these applications, the data consist of a matrix of pairwise distances between the nodes of a complex network and approximated by a low-rank matrix for fast community detection using a distance-based partitioning algorithm. Calculating such a low-rank approximation can reveal the underlying structure of the data and allow for fast computations.

14.3.1 Nystrom approximation

Any symmetric positive semi-definite (SPSD) matrix can be approximated as a subset of its columns using Nystrom methods. Specifically, given an $m \times m$ SPSD matrix A, the method requires sampling c ($< m$) columns of A to construct an $m \times c$ matrix C. We always assume that C consists of the first c columns of A without loss of generality. We partition A and C as

$$A = \begin{pmatrix} W & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} \quad and\ C = \begin{pmatrix} W \\ A_{21} \end{pmatrix}$$

where W and A_{21} are of size $c \times c$ and $m - c \times c$ respectively [31].

Definition 14.3.1 The standard Nystrom approximation of A is

$$A_c^{nys} = \begin{pmatrix} W & A_{21}^T \\ A_{21} & A_{21} W A_{21}^T \end{pmatrix}$$

Since the running time complexity of SVD on W is $O(kl^2)$ and matrix multiplication with C takes $O(kln)$, the total complexity of the Nystrom approximation computation is in $O(kln)$[32, 33].

14.3.2 Random sketching

In random sketching, a relation is modelled as defining a vector or matrix. The sketch is formed by multiplying the data by a vector [21]. A sketch vectorforms a synopsis of the data ad the synopsis is smaller than all the original data. Data mining algorithms can now be applied on the sketch vector.

Definition 14.3.2 Johnson-Lindenstrauss Lemma 1. For any set of n points $S \in R^d$, there is a $(1 + \varepsilon)$-distortion embedding of X into R^k $(k < d)$, for $k = O(\frac{log(n)}{\varepsilon^2})$.

Definition 14.3.3 Johnson-Lindenstrauss Lemma 2. There is a distribution over random linear mappings A: $R^d \to R^k$, $(k < d)$ such that for any vector x we have $\|Ax\| = (1 \pm \varepsilon)\ \|x\|$ with probability $1 - e^{-Ck\varepsilon^2}$.

The Johnson Linderstrauss theorems [30] for sketches ensure the preservation of distance during lower dimensional approximation and provide minimum lower bounds of approximation.

We have used random sketches for low rank approximations of the dataset.

- Step 1: counts the number of attributes in the dataset
- Step 2: sketch matrix is generated and number of rows equals the number attributed in the dataset. Sketch matrix have three sets of values: (1) between 0 and 1 (2) 0 and 1 (3)between maximum and minimum values of the dataset.
- Step 3: multiplies each row of the dataset with the sketch matrix in order to generate sketch vector.

After a sketch vector is generated machine learning algorithms are applied to the sketch matrix.

14.4 Near-Duplicate Detection by Min Hashing

The MinHash technique invented by Andrei Broder can quickly estimate similarities between two sets [188, 189]. Initially, it was used in the AltaVista search engine to detect duplicate web pages, clustering documents by the similarities of their sets of words.

Note MinHash uses the Jaccard similarity coefficient and hash functions. The coefficient is used to find similarities between two sets (A and B). A similarity of 0 indicates they have no elements in common. A similarity of 1 indicates both sets contain the same elements.

MinHash methodology can be classified into two types: variants with many hash functions and variants with only one.
In the many hash functions variation, the MinHash scheme uses k different hash functions. Here the k (a fixed integer parameter) represents each set S by the k values of $h_{min}(S)$ for these k functions. To detect the Jaccard similarity coefficient J(A,B) assume y is the total number of hash functions for which $h_{(min)}(A) = h_{(min)}(B)$, and use $\frac{y}{k}$. This is the average prediction of k hash at different 0-1 random variables. When $h_{min}(A) = h_{min}(B)$ and zero otherwise, and each of which is an unbiased estimator of J(A, B). Since the whole methodology works on the random number, the expected error is $O(1/\sqrt{k})$ (by standard Chernoff bounds for sums of 0-1 random variables). For example, to find the J(A, B) of 400 hashes would require error value of 0.5 or less. Similarly, a variant with a single hash function can be computed [188, 189].

14.5 Projects

Project 14.1 — Crime Rate Prediction Using K Means. The rate of crime in the past few years has increased in many countries. The high crime rates worldwide require effective protection. Create a crime prediction system by using a user's crime data to compute future crime rates. Use the K-mean algorithm and suitable data structure to identify and store different crime patterns, hidden links, link prediction, and statistical analysis of crime data. Administrators can see criminal historical data. Crime prediction relies on historical crime statistics along with geospatial and demographic data.

Project 14.2 — Online test system using clustering algorithm. Create a user-friendly online test system using interactive web applications and software. The system should provide fast access and information retrieval. The test result should show ranking and weights. [Hint: use any clustering algorithm for classification of question papers.]

Project 14.3 — Informtion leak detection system. Consider a scenario in which a sender wants to transmit confidential to a number of receivers. Due to an accident, the information is leaked. The sender wants to determine whether the leaked data came from one of more of his receivers. Design an information leak detection system that allows data allocation. Treat data allocation as an input that will help identify data leaks. You can also use also insert realistic but fake data records to further improve the chances of identifying unknown data leakages and also the receiver responsible for it.

Chapter 15

Application to Network and IOT

In recent years, networks, cloud computing, and the Internet of Things (IOT) have reshaped the world. Estimates indicate that a billion devices will be connected and generate trillions of terabytes of data by 2023. Among the applications of these systems are airline operations, the insurance industry, the media and entertainment field, and advertising and marketing of products and services.

Objective 15.1 This chapter addresses the applications of indices, skiplists, query resolution via hashing, and security issues affecting cloud computomg. networks and the IOT. Every section includes case studies. The chapter starts with use of bloom filters for click-stream processing. The remaining sections cover fast IP-address lookup and use of integrity verification for cloud and IOT data.

15.1 Click-Stream Processing Using Bloom Filters

The quick growth of online advertisement over the internet plays an important role in the success of the advertising market. One approach for generating revenue is the pay-per-click model. The advertising entity is charged for each key word click. Unfortunately, the increase of click frauds is slowly destroying the entire online advertising market. One step toward solving the problem is detection of duplicate clicks by window systems described below.

Two bloom filter-based algorithms have been proposed to detect duplicates in pay-per-click streams. The group bloom filter (GBF, also known as a jumping window) and the timing bloom filter (TBF, also known as a sliding window) process click streams via small numbers of sub-windows, use memory and computation time economically, and do not produce false negatives.

Note Burton H. Bloom proposed his space-efficient filter in 1970 [64] and it is commonly used in networking and database operations. The filter structure uses a set of n elements to respond to membership queries. Chapter 2 covers bloom filters in depth.

15.1.1 GBF Algorithm

Bloom filters use windows to detect duplicates in click streams. Clicks, like cookies, have their source IP addresses. Detection of duplicate cookies by passing its identifier through a bloom filter and if the identifier already exists, the duplicate is flagged. Another approxach is to evenly divide a jumping window into sub-windows, each of which has its own bloom filter.

To reduce the complexity of the system, all bloom filters should use the same hash function. Assume a jumping window of size N is divided into Q sub-windows. The bloom filter will expire when the window is full. This means the entire memory space of the expired filter must be cleaned. Cleaning is time-consuming (it requires $O(n)$ time). Extra queue space is required to handle new activity during the cleaning operation.

The extra space required during cleaning is provided by dividing available memory into $Q + 1$ pieces where Q represents bloom filter space divided by Q active sub-windows. The extra pieces serve as elements in the sub-windows during cleaning and allow more time for the cleaning operation.

Note GBFs were designed to fill the need for extra queue space during cleaning operations.

GBFs can significantly reduce the memory required to detect duplicates in click streams. The GBF operates by grouping bits with the same indices in bloom filters and nopt in the main memory. This allows the CPU to process the groups instead of individual bits.

For example, let Q active sub-windows = 31 bits and a word in memory = 32 bits. The 32-bit word is generated by the AND operation and held in the bloom filter. The bits constituting a word in an expired bloom filter are masked by setting corresponding bits to 0. If the value of a word is non-zero, the new element is considered a duplicate. For a zero value, set the corresponding bits to 1 and return them to memory [191].

15.2 Fast IP-Address Lookup Using Tries

With the evolution of the Internet of Things (IOT), the number of devices and Internet speed have increased exponentially. The computing world needs fast processing routers and high-speed IP address lookup engine. The role of the router has become crucial since routers forward packets to the appropriate interfaces (ports). Routing millions of packets based on addresses has become extremely difficult. The increase in traffic led to development of an efficient IP lookup algorithm derived from tree-based and trie-based structures [192].

Note Tree- and trie-based data structures are used in network prefixes. Both structures are discussed in Part I.

The difference between tree- and trie-based approaches is information distribution. Trees hold information by nodes; the number of nodes shows the number of network prefixes. The depth of a binary tree is log (N). Distribution of trie data occurs on the edge.

The right edge of a binary tree handles one bit; the left age is for zero bits. The path from the root to a node defines the network prefix. A prefix in a table should be labeled in a trie tree. The longest prefix indicates the depth of the table. The latency of the lookup algorithm in a binary trie is measured in amount of memory accessed which is equal to data structure depth. Disadvantages of a binary tree are rebalance overload, and smaller depth than a binary trie.

Memory consumption of tree and tries is calculated by using the number of nodes in the data structure and sizes of all nodes. Tries hold more nodes than trees since some nodes in a trie do not correspond to valid prefixes; the numbers correspond in a tree.

A tree may seem to require less memory than a trie, but node size is smaller in tries. A tree requires 64 bits per node; a trie requires only 32. The multibit trie is an improvement of a binary trie in that it increases speed. The Depth of Multibit tries is lesser as compared to binary tries. It makes the multibit tree faster as compare to binary tries. For example, the IP address is of 32 bit which can be represented as 192.16.73.2. The IP address segment can be represented as four octets(192,16,73,2). The depth of the tree in multibit trie is equaled to the four octets. The memory utilized in multibit tries depends on the number of octets and the length of each octet.

Note It is difficult to state definitive information about memory consumption of trees, tries, and multibit tries because memory consumption always depends on the memory needs of an operation.

Note A combination of tree and trie data structures can provide maximum compression.

Figure 15.1 compares memory consumption of a tree with the consumption achieved by a combination of a tree and a trie. The most significant 16 bits are considered at the first step; the depth of the structure should never exceed 18 bits [192, 168].

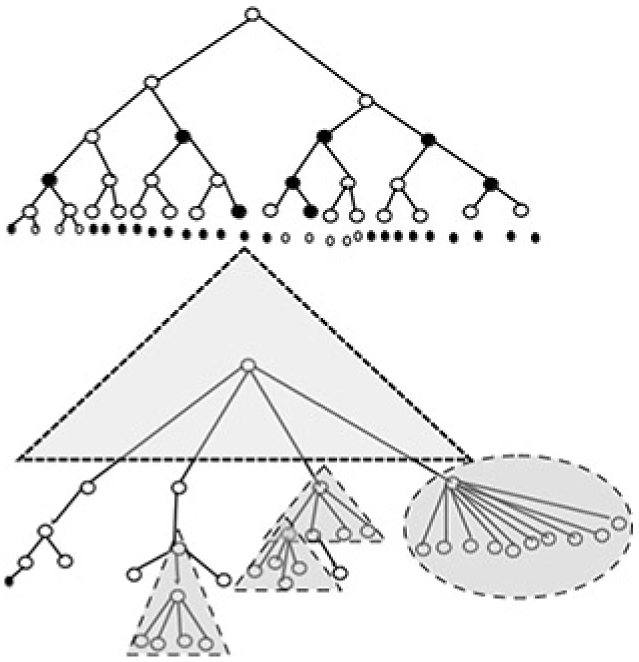


Figure 15.1: (a) Binary trie showing network prefixes in 6-bit address space; (b) Combination of tree and trie shown in (a)

15.3 Integrity Verification: Cloud and IOT Data

Data protection is vital for users. Cloud computing is effective because it provides quality service while maintaining user privacy and security. Data stored in the cloud are generated by the IOT, businesses, governments, and individuals. Data integrity is vital, especially for outsourced data [197]. Much recent research has focused on data integrity verification.

Data integrity is achieved by preventing alteration by an intruder. Proposed data integrity measures include checksums, trapdoor hash functions, message authentication codes (MACs), digital signatures and other hash-based approaches. While many methods can ensure data integrity, cloud users must still have a trusted party verify their data remotely.

Note A cloud server must verify data from an external party independent of the cloud service provider.

One approach is retrieving and downloading all data to be verified from its server but this approach is not feasible. The large amount of data increases time consumption and communication overheads.

An authenticated data structure based on data positions and paths from roots can be used. Two examples are the Merkle hash tree (MHT) and the rank-based authenticated skiplist (RASL).

The MHT [164] is an authentication structure used to verify dynamic data updates. It is similar to a binary tree in which each node N can have a maximum of two child nodes. Information stored in each N node in MHT T is H hash value. In a tree construction a parent of node $N_1 = \{H_1, r_{N_1}\}$ and $N_2 = \{H_2, r_{N2}\}$ is constructed as $N_P = \{h(H_1 \parallel H_2\}$ where H_1 and H_2 are information pieces stored in N_1 and N_2 nodes respectively. Each leaf node (LN) is based on a message m_1.

The RASL authenticates data content and indices of data blocks. The RASL skiplist [167] included a provable data possession (PDP) scheme to provide efficient integrity checks of outsourced data. The RASL was the first scheme to provide simultaneous authentication and public verifiability. Its logarithmic complexity is similar to that of the block system of the MHT.

15.4 Projects

Project 15.1 — Attacker tracing using packet marking. Marking schemes for network packets are used extensively for network security. They simplify the tasks of packet analysis and IP source tracking [198].

Propose a marking scheme that will provide a convenient way for a user to accurately identify an attacker. Your solution should be flexible: change as situations change and modify its marking style based on router loads. Your project should include the ability to retrace sources even during heavy router loads (hint: use hash technique).

Project 15.2 — Controlling network congestions using internet border patrolling. Congestion control is a vital factor in network implementation. The efficiency of existing algorithms in controlling congestion remains to be proven. Your project is to propose an Internet border patrol scheme [199] to perform surveillance of packets and questionable data at network borders. To increase efficiency and stabilize bandwidth flow by allocating resources, use the enhanced core stateless fair queuing (ECSFQ) algorithm [200].

Project 15.3 — Delay-tolerant networks and epidemic routing. End-to-end connections occur rarely or never in delay-tolerant networks (VANET and interplanetary networks for example). One solution [NOTE: you haven't stated a problem that needs solution.] is to use epidemic routing [201] based on a flooding-based algorithm that replicates, stores and forwards data. Replication is achieved by exchanging summary vectors (data structures for tracking unavailable data in nodes). A normal bloom filter serves as the data structure. However, summary vectors suffer from collision. Your project is to design a modified summary vector that can accommodate large amounts of data. [Hint: Read Chapter 2.]

Project 15.4 — Network-based stock price system. Assume that a stock market needs to track and update its records constantly and these tasks require frequent server interactions. Stock prices change often so accuracy of update information is vital. Choose a data structure to design a robust price update system and explain why the chosen data structure is appropriate for this scenario.

Chapter 16

Applications to Systems

The system is a set of items that works collaboratively as part of a larger construction. Computer science utilizes many systems that manage router packets, operating functions, distributions, and other tasks. Each module of a system involves complex processes that require optimized data structures.

Objective 16.1 This chapter addresses applications of data structures to various system modules. It first covers queue-spilling algorithms for maintaining packet queues in routers and switches. Subsequent sections explain use of data structures in schedulers, distributed caching, and file system applications.

16.1 Queue Spilling

Iyer et al. [177] demonstrated that router and switch efficiency can be increased by using expensive SRAM, economical DRAM and queue-spilling algorithms. Queue-caching algorithms were proposed to manage unconsumed data that exceeded queue space and had to be spilled into secondary memory to be retrieved later. The huge amounts of data processed by the Internet make solution of the heuristic queue-spilling problem essential. Motwani et al. [178] proposed a model to manage queue spilling.

These algorithms and models work in a variety of data streaming systems. The proposed systems allow efficient management of queues in memory by using buffers for secondary storage. Caching queues in memory buffers are triggered whenever an certain number of data items (tuple) enters a queue. Only one tuple is consumed by the header because queues have first-in-first-out (FIFO) properties. Tuples are assumed to be of the same size M and n represents he number of queues available in the buffer. The process assumes that some unbounded tuples may need to be transmitted to main memory.

The algorithm can read the tuple from the secondary memory whenever required. The queue spilling algorithm should be able to decide in an on-line manner about the tuple state (read or write). The half algorithm is proposed for caching queues systems because it keeps the two active ends of the queue buffered in memory. The algorithm divides an unconsumed tuple into a head, spilled portion, and tail. The head contains the oldest tuple. The tail portion contains the most recent tuple. The tail and head reside in the main memory and the spilled portion resides on a disk.

Newly arrived tuples reside at the head segment; the tail is empty. The head portion expands as additional tuples arrive. The write operation on tuples continues until the queue is full, at which time the algorithm writes out $\frac{\mathrm{M}}{2}$ for the newest tuple to spill. The sizes of the head and spilled portion are the same i.e. $\frac{\mathrm{M}}{2}$ at that point and the newly arrived tuple is sent to the tail.

The proposed algorithm will ensure the following invariants invariants during read and write operations:

- Invariant 1: The tuples in the head are always older than the tuples in the spilled portion and the tail.
- Invariant 2: If the spilled portion is empty, the tail will also be empty.
- Invariant 3: The maximum size of the spilled portion can reach $\frac{\mathrm{M}}{2}$, i.e., the maximum size of the head.

The steps required to maintain these invariants [178] are as follows:

[$Write - Out$]: Once the tuples reach the $\frac{\mathrm{M}}{2}$ and the spilled portion is empty, write the $\frac{\mathrm{M}}{2}$ data from the tail to the spilled portion.

[$Read - In$]: If the head is emptied before the tail reaches $\frac{\mathrm{M}}{2}$, the half algorithm will read in $\frac{\mathrm{M}}{2}$ the oldest header from the spilled part to the head.

16.2 Completely Fair Schedulers in Kernels

A completely fair scheduler (CFS) maintains fairness among processes during running states [175]. Fairness provides execution time to processes in danger of becoming out of balance. A CFS maintains virtual run time (time needed for a specific task) and sleeper fairness to ensure that waiting after resumption of running state gets a fair share of processor operation.

Note A CFS uses time-ordered red-black trees rather than queues to maintain running states of processes.

Figure 16.1 illustrates a red-black tree that has specific properties: (1) it is self balance; no path in the tree will be used more than twice; and (2) insertion and deletion tasks require only $O(\log n)$ time.

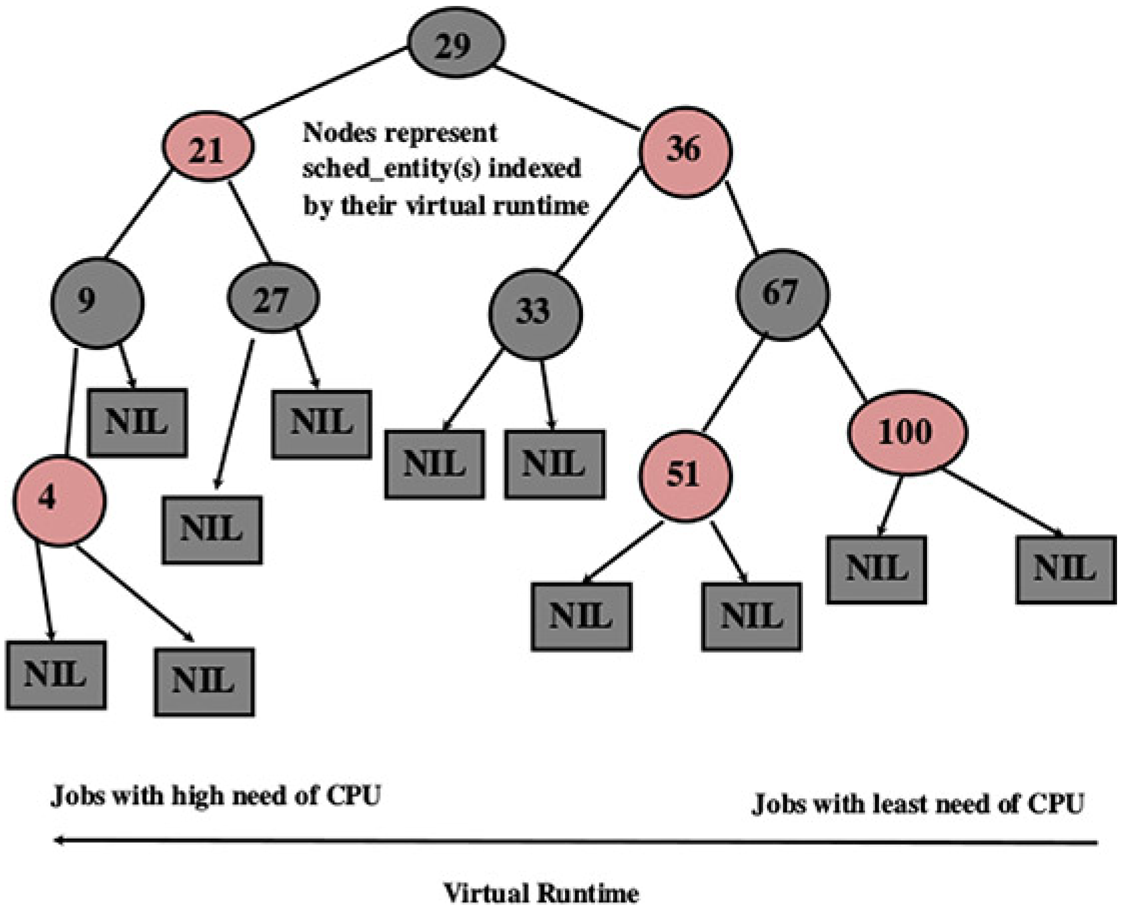

29
21
36
Nodes represent sched_entity(s) indexed by their virtual runtime
9
27
33
67
NIL
4
51
100
Jobs with high need of CPU
Jobs with least need of CPU
Virtual Runtime

Figure 16.1: Red-back tree stucture used with completely fair scheduler

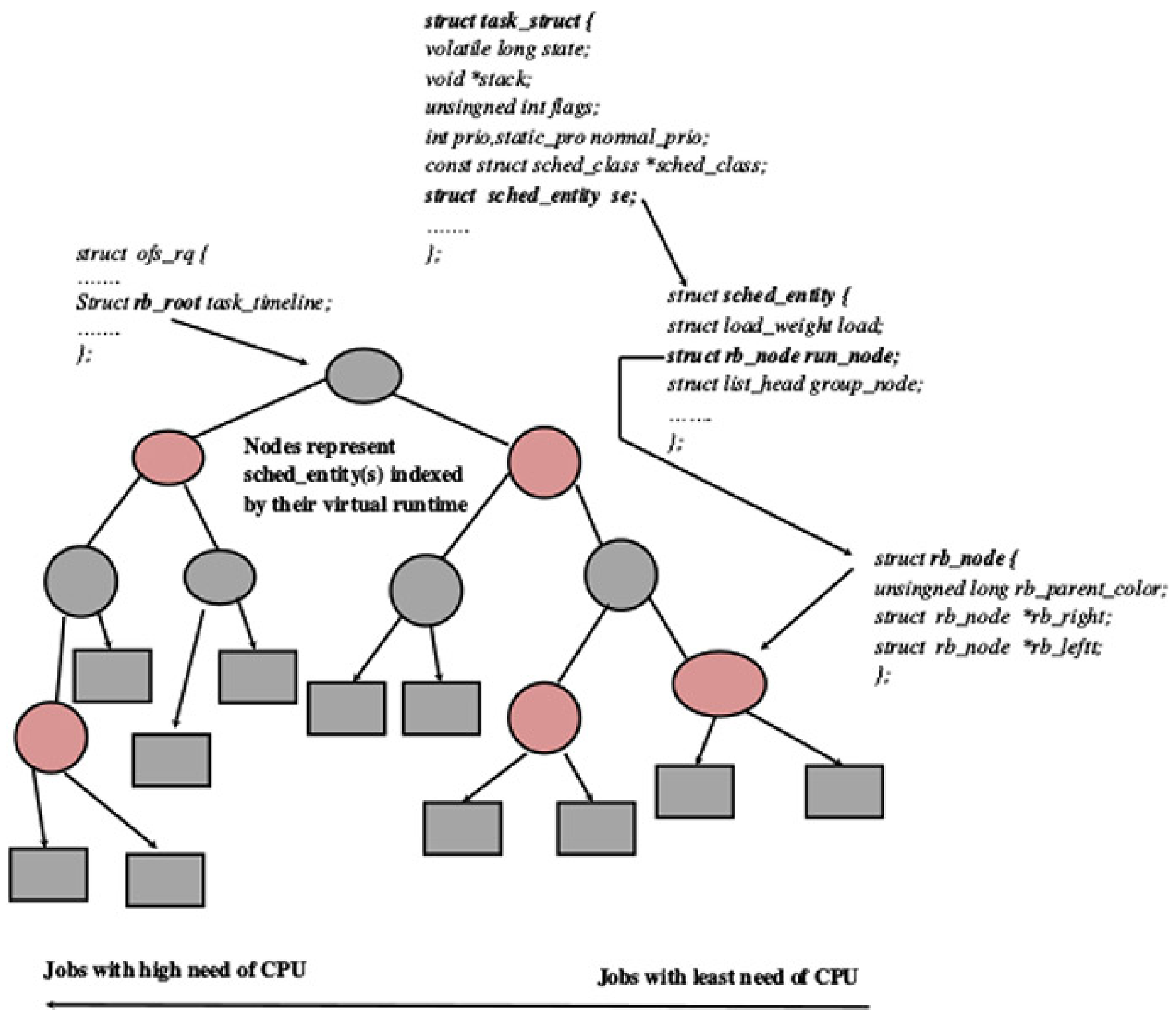
Figure 16.1 represents the task as sched_entity objects stored in a red-black tree. Tasks with fewer processor needs are stored on the right side of the tree; tasks with more needs are stored on the left side. The scheduler selects the left-most nodes of the tree first, then the remaining left nodes.

16.2.1 CFS internals

A task in a link is handled by a task_struct function that represents all associated attributes (description, current state, stack order, static and dynamic priorities, and process flag).

Note Most of the task related structure can be found in a header file at./linux/include/linux/sched.h. No tasks related to the CFS will be run.

The root of the red-black tree in Figure 16.2 uses the rb_root element from the cfs_rq structure in ./kernel/sched.c). The internode (rb_node) represents a runnable task. The leaves mean “no information.” Internal node(rb_node) resides within the $sched_entity$ structure. It includes the rb_node reference, load weight, and a variety of statistical data. The $sched_entity$ contains the *vruntime* (64-bit field), it indicates the amount of time occupied by the processor and index [175,176].



struct task_struct {
volatile long state;
void *stack;
unsingned int flags;
int prio,static_pro normal_prio;
const struct sched_class *sched_class;
struct sched_entity se;
....
};
struct ofs_rq {
....
Struct rb_root task_timeline;
....
};
struct sched_entity {
struct load_weight load;
struct rb_node run_node;
struct list_head group_node;
....
};
Nodes represent sched_entity(s) indexed by their virtual runtime
struct rb_node {
unsingned long rb_parent_color;
struct rb_node *rb_right;
struct rb_node *rb_leftt;
};
Jobs with high need of CPU
Jobs with least need of CPU

Figure 16.2: Relations of red-black tree structures

16.3 Distributed Caching

A distributed system is a collection of independent computers connected by some medium. Data caching provides solutions to many serious problems in distributed environments.

Note The hash-based bloom filter is used extensively to manage data caching in distributed environments.

Feng et al. [179] proposed a system using bloom filters to distribute data cache information. A summary cache system generates a question to determine whether another station's cache holds desired data if a local cache misses a step. Communication and time costs for accessing data from the original station are reduced. To reduce message traffic, stations use periodic broadcasts of a bloom filter that represents a cache instead of transferring the entire content of the cache.

Each station verifies bloom filter of other stations for any data availability. False positivity and false negativity will trigger delays due to hash collisions caused by the bloom filter's limited buffer size. Distributed caching has proven useful in Google's BigTable, Google Maps, Google Earth, Web indexing and other distributed storage systems for structured data. These applications utilize bloom filters to reduce disk lookups. Summary cache systems are used extensively in cloud computing, mapping, and reduction paradigms. Bloom filters optimize reduction operations. Summary caches divide applications into small chunks to achieve parallel efficiency [180].

16.4 Data Structures for Building File Systems

Disk file systems use bitmaps to track free blocks and handle queries related to specific disk blocks. Disk files need good data structures to store directories and efficiently handle queries and fast lookups. Microsoft's early FAT32 system used arrays for file allocations. The *ext* and *ext*2 systems use link lists. The XFS and NTFS systems use B+ trees for directory and security-related metadata indexing. The *ext*3 and *ext*4 file systems use modified B+ trees (also known as H trees) for file indexing [181].

16.5 Projects

Project 16.1 — Android Task Monitoring. Modern life demands completion of multiple tasks every day. Attending meetings, taking medications, paying bills, planning trips, going to classes and other activities represent daily challenges. The human brain is not designed to handle multiple tasks at the same time. Your project is to design a task alert system to broadcast a reminder whenever an important task must be done. The project should include a type of artificial intelligence assistant that will scan your android phone and create a schedule. The assistant can use interactive techniques and must ensure that you complete all tasks managed by the alert system.

Project 16.2 — Android Home Automation. Create a home automation project using an android phone. The major task of the system is to control home devices (lights, heat, alarms, air conditioning, appliances). You can choose any data structure to build your system.

Project 16.3 — File System. Create a file system using binary search trees and Linux. [Hint: Start by copying an existing file system to see how it works.]

Project 16.4 — Wearable Health Monitoring System. Create a health monitoring system using android phones. The objective is to maintain health records, coordinate appointments of multiple doctors, and schedule medicine intakes. The system should include update capabilities.

Project 16.5 — Firewall System. Create a signature-based firewall security system that will prevent unauthorized access to or from a private network. [Hint use the bloom filter to detect signature.]

Chapter 17

Applications to Databases

Databases are designed to maintain data for businesses, governments, and individuals. User requirements continue to increase and database capacity and efficiency must improve constantly to meet such requirements.

Objective 17.1 This chapter discusses applications of B trees, bloomjoins, and the CouchDB structure to resolve search-related database issues.

17.1 Database Problems

17.1.1 Searching sorted files

The complexity of a sorting and searching algorithm depends on the number of comparisons performed. For example, for a binary search of 1 million records, a specific record can be located with at most 20 comparisons. The large database is usually kept in the disk drives and the time to read a record on a disk drive (magnetic tapes) far exceeds the time needed to compare keys once the record is available. The time to read an element from a hard disk involves seek time and rotation time. Seek time is usually 0 to 20 ms, and sometimes more. The rotation time for a Seagate ST350032NS 7200 rpm drive is around 8.33 ms. For simplicity, assume that seeking data from a hard disk requires 10 ms. The time to process 20 disk reads to locate a single record among 1 million (at 10 ms per read) is 0.2 second. A little time saving results because individual records are grouped into disk blocks that hold approximately 16 Kb. About 100 records per block could be saved if each record holds 160 bytes.

17.1.2 Index for first search

If the index of the database is improved then the time consumed during searching can also be improved. Search range in the above example can be improved by creating an auxiliary index containing the first record in each disk block. This is known as a sparse index. The auxiliary index decreases search time but it occupies around 1% of the total database. It also identifies the block where the search is to be done, eliminating the cost of searching an enormous database. The auxiliary index can hold up to 10,000 entries, so it would take at most 14 comparisons. The index could be searched in about 8 disk blocks and the desired record will be accessed in around 9 disk reads. The trick to optimize the search is to create a new auxiliary index of an auxiliary index. This will reduce search further and it can be accommodated on one disk. So instead of reading all 14 disk blocks to find a desired record, we need only 3 block reads. Auxiliary indices can reduce search time to one-fifth of the processing time with no indices. If the main database is used frequently, the auxiliary disk will reside at the disk cache to avoid repetitive disk reads.

17.1.3 Insertion deletion in database

Index compilation becomes easy what a database is fixed. However, if a database changes over time, index management becomes complex. Operations like deleting a record have little impact because only the record in the index will be marked as deleted. If deletions become frequent, search operations become less efficient. Insertion of data in a sorted record is expensive because it requires shifts of all previous records.

Note We suggest storing all records sparsely in a block to create free space between the records. The free space can handle insertions and deletions efficiently.

If inserted data will not fit on a block after use of the above technique, free space on an out-of-sequence disk may be used.

17.2 B and B+ Trees for Database Creation and Block Search

Most of the self-balanced search trees (like AVL and red-black trees) assume that all data is stored in the main memory. This is a constraint when large data operations must be performed.

Note Disk access requires more time than accessing main memory.

B trees can reduce the frequency of disk access. Search, insert, delete, max, and min require $O(h)$ accesses where h is tree height. The high of a B tree can be reduced by applying the largest key possible to its nodes. Generally, B tree node size is equivalent to block size. B tree height is low in comparison to other BSTs like AVL and red-black [12].

17.2.1 Applications of B trees in databases and file systems

- B trees keep the keys in sorted order so that the sequential traversal becomes easy.
- To minimize the number of disk reads, B trees use hierarchical indexing.
- B trees use partially full blocks to speed insertions and deletions.
- B trees utilize elegant recursive algorithms to keep their indices balanced.

Note B trees utilize only half their interior nodes t minimize time requirements.

Note B trees can also handle any number of insertions and deletions.

Note A binary search of a sorted table (*N* records) requires roughly (*log*2 *N*) comparisons.

Note B trees can serve as file systems to quickly access arbitrary blocks without need to access the main database.

17.3 CouchDB

Apache CouchDB is a new product intended to enhance database management. It uses a B tree data structure to index its documents and views. It also acts as a B tree manager with an HTTP interface.

Note CouchDB uses a B+ tree, a variation of the B tree that trades a bit of (disk) space for speed,

CouchDB uses a B+ structure to store huge amounts of data. B trees typical have single-digit heights even when storing millions of entries. The advantage of a B tree is the ability to store leaves on a slow medium such as a hard drive. CouchDB utilizes that feature. An operating system is likely to cache the upper tree nodes so only the final leaf nodes are stored on a hard disk. B tree access time is fewer than 10 ms, even for extremely large amounts of data. CouchDB, B tree appends data only to the database file that keeps the B tree on disk and append-only leaf nodes. B Tree provides a robust database file. Inclusion of B tree features in CouchDB helps avoid data corruption by not overwriting existing data on hard disks.

CouchDB restarts after problems like power failures by backward reading of database files. If the first 2*k* (footer with checksum) is corrupted, CouchDB relaces it with the second footer. If that footer is corrupted, the first 2*k* is copied. When both footers are replaced successfully to the disk, the write operation is successful. Documents indexed in a B tree are given names (DocID) and sequence identifications. Each database update generates a new sequence number that will be used later to find changes in the database. Indices are updated by saving or deleting documents.

Note Index updates occur after files are appended or modified.

17.4 Bloomjoins

A bloomjoin is a algorithm used for combining database attributes in distributed environments. The sequence of steps is shown below. Consider an Internet scenario involving Site 1 and Site 2; both have data sets designated T1 and T2 respectively.

- Site 1 computes a bloom filter $F(T1)$-based table of T1 records by hashing h1(T1) and sends the table to Site 2.
- Site 2 computes a bloom filter $F(T2)$ and filters out all records that do not belong to $F(T1)$. Assume T′ is the required record.
- Site 2 then sends T′ to Site 1 where the join is computed.

This algorithm is not limited to only 2 sites. However, the algorithm does not specify any method to minimize the network cost. Assigning optimal configurations to records such as table structures, limiting numbers of records in tables, and joining data can reduce message size [190].

17.5 Projects

Project 17.1 — Question paper generator system. Create a smart exam paper generation system that will allow an administrator to input questions and answers. The system should restrict the ability to determine weights and complexities of questions to the administrator. The system will be capable of incorporating modifications to the exam after it is converted to pdf format and emailed to colleges. [Hint: Use a bloomjoin.]

Project 17.2 — Online tutorial using Couch DB. Use CouchDB to implement membership and revenue aspects of an online tutorial system. A user will register on a website, then pay a fee, after which the user is treated as a member and can view and download educational content. Include a procedure that will allow a user to refer new members and and collect a 30% rebate if a potential member joins. Your design should include four referral levels with decreasing rebate levels.

Project 17.3 — College governance system. Create a chart of accounts for various departments (finance, computer science, administration) with access to all accounts available to the top official. The first task is to choose any data structure you want. The requirements are listed below:

- Separate sections for all departments (you can choose any number of departments.
- Online access and processing of data restricted to certain officials.
- Administrator accounts for top officials.
- Separate sections for communications to and from various entities: A2S = Administraton to Students; A2F = Administration to Faculty; S2A = Students to Administration; A2P = Administration to Parents.

Project 17.4 — Farming assistance application. Design a Web project to increase profitability of farmers through direct communications between farmers and between farmers and supplies. Dealers should have the capability to advertise to farmers. Farmers should be able to rate dealer products and services. All parties will access the system via login identifications and passwords. [Hint: Database should utilize red-black trees.]

Chapter 18

Applications to Images and Graphics

This chapter demonstrates several applications of data structures to image and graphics processing operations. We selected three applications (among many) that are useful in these industries: (1) R trees for searching objects in maps; (2) KD trees for optimizing computations in geographic information systems (GISs); and (3) ray shooting for graphics rendering. The chapter ends with a group of projects.

Objective 18.1 This chapter addresses computer graphics applications of R trees, KD trees and octrees.

18.1 R Trees for Map Searches

Nearest neighbor and certain other searches involve degree measurements and parallelograms. A basis node is inserted into the priority queue. The search of closest entries in the queue continues until the queue is empty or results are returned. Various aspects of R trees (children, node operations, leaves) are detailed in Section 6.4. This approach is useful for obtaining geographic information by calculating distance metrics.

18.1.1 R trees for mapping

R trees provide spatial information by calculating geographical coordinates and geometric structures.

R trees can store abstract objections like restaurant locations and map insets and answer queries like finding museums within a certain radius, illustrating the best route to a specific location, and finding the nearest gas station.

R trees can accelerate nearest neighbor searches for various distance metrics including great-circle distances.

18.1.2 Insertion

Insertion of an object requires the recursive traversal of the tree from the root. At each step, the rectangles on the current directory are examined and the candidate is chosen using heuristic approach (e.g., choosing a rectangle with minimum enlargement). The search proceeds until it reaches the leaf node. The tree should be notified of full nodes before insertion. A heuristic approach splits a full node in half to prevent need for an exhaustive search. The newly created node traverses to the previous level; the overflow propagates to the root note [182].

18.1.3 Deletion

Deleting the entry from a page requires the adaptation of the parent page. A full page will not balance with its neighbors. It will dissolve and the nodes will be reinserted. If a root node contains fewer elements, the tree height will decrease.

18.1.4 Search

R tree searching is similar to B+ tree searching; both start at the roots. All nodes contain sets of rectangles and pointers to indicate child nodes. Every leaf node contains rectangles of spatial objects. Every rectangle in a node must decide whether to overlap. Corresponding nodes must also be searched if overlap occurs. Searching continues recursively until the child note is traversed. When a leaf node is searched, the boundary box is also searched and its results are included if they lie within the search rectangle.

18.2 Spatial Proximity in GIS

GIS studies are essential in fields like geography, cartography, civil engineering, and image processing. Processing of geographic information involves integrated database techniques and pictorial data processing. Recent research has developed many methods for handling map data. Map information is stored graphically as points, sequences of straight line segments, and regions (polygons). Distances and other spatial data are stored in matrices.

18.2.1 GIS objects

Points, line polygons, and other graphics are used to map geographic objects. KD trees can optimize GIS search queries significantly. KD trees work recursively by partitioning two-dimensional maps into rectangular blocks using parallel lines and coordinates. Each block represents a page in disk storage. Sets of records are clustered into small groups based on spatial proximity. Partitioning by various tree structures can reduce the time needed for answering queries.

18.2.2 Data access in GIS

GISs store large amounts of data and require secondary memories to store excess data. Paging (partitioning records into small groups) is one way to organize huge data quantities. Using search keys to handle GIS queries is an important function. One issue is the difficulty of calculating and storing distances between objects in advance of queries. A GIS will typically have to read many records in disks and calculate distances whenever a metric response is needed. This type of processing involves expensive computation time and access to huge numbers of pages.

18.2.3 Computational requirements

Matsuyama et al. [183] tried to facilitate query responses in GISs by reducing the number of accessed pages. Although a user can inquire about any area on a two-dimensional map, his queries usually focus on a small area. To respond to such limited inquiries, information about neighboring records should be stored on the same page, in a pattern identical to clustering data for identification. Geographic data processing requires dynamic file structures so that data can be inserted, modified, and deleted easily.

18.2.4 Solution using k-d tree

To achieve dynamic partition of a record, A KD tree search [183] divides two-dimensional map spaces into rectangular blocks by setting lines parallel to rectangular axes. Each block matches a page on a hard disk where all records including block items are stored. Accessing a disk on a hard page requires verification of the corresponding rectangle on a map. Processing can proceed quickly as soon as the page is accessed from main memory.

1. Lengths of the vertical and horizontal sides of the rectangle are compared for partitioning.
2. The rectangle is divided into two parts with the help of a straight line perpendicular to the longer side. The dividing line must ensure that each portion contains the same amount of data.
3. Repeat Steps 1 and 2, for each rectangle until the amount of data in any rectangle becomes less than the capacity of a page.

The position of the rectangle can be represented by double trees. The node of the tree represents a rectangular area generated by the division process. The root node points to the whole map and leaf nodes represent blocks related to pages.

If excessive insertions cause page overflow, divide the page into two pages using the above steps. Deleting entries is more complicated because the user must decide whether a page is associated with another page by checking neighboring trees.

Whenever a deletion creates an empty page, the leave node of the tree is marked “NIL”. A tree will become unbalanced after many insertions and removals and the database may need restructuring.

A file system can be reorganized by (1) partitioning map space as described above; or (2) saving the binary tree used in partitioning to control page access. File organization can significantly improve handling of distance queries. To find all points in a specific rectangle, we perform a comprehensive search using recursive algorithms to compare rectangle applications. If the range is contained on both sides of the split line, only the child of the related node should be searched.

▪ **Example 18.1 — Partitioning city map.** After a map is partitioned into blocks, KD tree partitioning is applied, using the centroids of the blocks as data points. A secondary KD tree handles page allocation. Each block corresponds to a disk track and pages whose centroids are contained in a block are allocated to the same track. Quadtrees can simplify searches for adjacent pages [183]. ▪

18.3 Ray Shooting

Roth [184] described ray casting (also known as ray shooting) in detail. The technique is used to process queries in computational geometry. A set of objects is assigned to d-dimensional space where the objects are preprocessed and sent to a data structure. The first object of each query ray is subject to a "fast hit."

18.3.1 Rays

To achieve select pixels in any order, then go from top to bottom and from left to right. The camera in the world coordinate system provides the rays. The direction is calculated by locating the four corners of the virtual image in the first world space, then splitting. The normalized direction is calculated from the position of the camera to the virtual pixel.

18.3.2 Camera-ray intersections

The initial camera is tested for intersections with three-dimensional visuals containing triangles and other graphics primitives. If the ray does not hit an object, pixels in a certain area are colored. To locate an intersection, the system must review color, structure, content and other relevant data. If the ray hits the center of a triangle, information will be calculated by interpolating the data from the top.

18.3.3 Shadow rays

Shadow is an important light effect that can be easily calculated by tracking the rays. If shading illumination for a given point is required the system must capture additional rays between the point and light source. Light can only contribute in the final color if nothing happens between the ray point and the light source. The equation of light which is a straight line in the quarter, can be easily adapted to handle ray because if the light is blocked, it will be 0.

18.3.4 Reflection rays

Another powerful feature with ray tracing is the exact reflection of complex surfaces. To create an ideal mirror surface rather than computing light through the normal equation, create a new ray of reflection and find it on the stage. All reflected, broken, and other rays are called secondary rays. Rays mirrored in the form of shade should be moved slightly on the surface to prevent the surface from crossing again.

18.3.5 Transmission rays

Radiation due to defraction can be used to tilt light to focus on transparent surfaces. The process is called transmission. When a beam attacks a transparent surface (like glass or water), then a new refracted ray is generated. We will assume that the transmitted beam will follow Snell's law as ($n_1 sin\theta_1 = n_2 sin\theta_2$), where n_1 and n_2 are the indices of refraction for the two materials.

18.3.6 Recursive ray tracing

The classical ray tracking algorithms have features like shadow, reflection and refraction. Depending on the number of lights and type of material, a primary ray can produce many secondary and shadow rays. These beams can be considered as the structure of the tree.

18.3.7 Ray intersection

The ray intersect routine uses a ray as input, returns a true value if the object is hit, and a false return if the object is missed. If an object is hit, the data is transmitted into the intersection class.

18.3.8 Bounding volume hierarchies

The basic concept of the binding volume hierarchy is inserting a complex object into a hierarchical structure. For example, if the size of a bounding volume hierarchy is limited, the system can provide more areas until the lower containing real geometry is reached. In this area, there are many other areas, until we finally reach the lower level, where there is real geometry like triangles in the globular circles. To test a ray in a scene, we cross the highest level hierarchy. Whenever an area is attacked, we test its spheric structures and finally tests triangles and other primitives. Normally, the volume limit can reduce beam crossing time from O (n) to O ($log\ n$), where the n is the number of primitives used in the scene. This reduction in linearity for logarithmic performance improves capacity and makes creation of scenes with millions of primitives possible.

18.4 Data Structures Used in Ray Shooting

18.4.1 Octrees

Octree construction starts with placing a cube around a scene. If more than a certain number of primitives (e.g., 10) occupy the cube, then it is evenly divided into 8 cube, which is re-tested and finally retrieved. The area is a more regular area structure and gives a clear rule of subdivision and does not overlap between cells. The octree is an attractive option but it is not yet ideal for ray shooting operations.[207]

18.4.2 KD trees

KD tree construction starts with placing a box (essentially a cube) around a scene. If the box contains many primitives, other systems would divide into octets. A KD tree splits the box into two boxes that do not have to be equal. Any arbitrary point in a partitioned box can be at X, Y, or Z. This quality makes KD trees adaptable and better suited to process wrong geometries. They are effective for tracking rays. Their main disadvantage is that the depth of the trees can be deeper, because of the intersection of the Beam to spend more time handling test intersections and less time on older ones.

18.4.3 BSP trees

A binary space partitioning (BSP) tree is similar to a KD tree. Both split spaces into two boxes that are not necessarily equal. KD tree splitting is limited to the XYZ axis. A BSP tree allows the splitting plane to be placed anywhere and aligned in any direction. The disadvantage is that the number of heuristics makes choosing a splitting plane location difficult. Both BSP and KD trees perform ray tracing well.

18.4.4 Uniform grids

It is possible to separate a location in a unified network instead of hierarchically; the result is a uniform grid. The process is fast but it incurs high memory costs, especially for large and complex scenes. The performance of a uniform grid decreases in computations involving wide varieties of shapes and places of primitives so they are not practical for use in general ray accelerator structures.

18.4.5 Hierarchical grids

A hierarchical network starts with a grid. Each cell containing many primitives is subdivided. One example is a hierarchical network limited to 2 × 2 × 2 subdivisions. Some hierarchical networks can support subdivisions in any number of cells. They perform beam tracking very well, especially when processing relatively similar shapes.

18.5 Projects

Project 18.1 — Emergency Assistance. Create an android application that allows users to connect directly with hospitals, fire stations, and police stations by pressing a panic button. The application must allow users to automatically call emergency help and convey user locations to family members. [Hint: Use a KD tree and virtual coordinate system (to compensate for GPS unavailability) in your design.]

Project 18.2 — RSSI based Indoor Positioning System. GPS usually tracks android phone locations on Google maps. Consider a scenario when GPS does not work or you want to track a location on a map. You have to create a local RSSI-based system to handle both situations. [Hints: Your system should gather and map data it using the indoor environment is estimated by FMM which is simultaneous algebraic reconstruction technique (SART). You can use any data structure to implement your system and refer to Sugano [185].]

Project 18.3 — Graphical Password via Image Segmentation. Consider a user who can enter his password as an image. When the system receives the image, it breaks it into segments and stores them in an array. The next time the user authenticates his password, the segmented image is shuffled and returned to the user. If the user selects part of the image, he is either authenticated or stopped from using the system. [Hint: The system uses image segmentation based on coordinates of the segmented image stored in different parts of the system.]

Project 18.4 — Image Processing to Detect Expressions. Emotions can be expressed in many ways. Create a system that can analyze emotions based on expression recognition. The system will try to find an emotion based on an input image. Images are prone to noise so a preprocessing step must be incorporated to remove noise and ensure a correct result. Your system should achieve a 50 to 60% success rate. [Hint: use an effective expression recognition algorithm.]

Bibliography

[1] D. Knuth, *The Art of Computer Programming*. Boston, Addison Wesley-Longman, 1998, Vols. 1–3.

[2] A. C.-C. Yao, A study of concrete computational complexity, PhD thesis, University of Illinois, Champaign-Urbana, 1975.

[3] A. C.-C. Yao, Should tables be sorted? *J. ACM*, vol. 28, no. 3, 1981. pp. 615–628. DOI: 10.1145/322261.322274.

[4] A. C. Yao, Complexity in information theory, in *Computational Information Theory*, Springer, New York, 1988, pp. 1–15.

[5] C. A. R. Hoare, Notes on data structuring, in *Structured Programming*, Academic Press, London, 1972, pp. 83–174.

[6] D. E. Knuth, Optimum binary search trees, *Acta Inf.*, vol. 1, no. 1, 1971, pp. 14–25. DOI: 10.1007/BF00264289.

[7] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. ACM*, vol. 32, no. 3, 1985, pp. 652–686.

[8] D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Commun. ACM*, vol. 28, no. 2, 1985, pp. 202–208. DOI: 10.1145/2786.2793.

[9] M. Thorup, Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations, *J. Algorithms*, vol. 42, no. 2, 2002, pp. 205–230.

[10] M. L. Fredman and D. E. Willard, Blasting through the information theoretic barrier with fusion trees, in *Proceedings of the Twentysecond Annual ACM Symposium on Theory of Computing*, DOI: 10.1145/100216.100217.

[11] E. D. Demaine, D. Harmon, J. Iacono, and Pătraşcu, Dynamic optimality - almost, *SIAM J. Comput.*, vol. 37, no. 1, 2007, pp. 240–251.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, MIT Press, 2009.

[13] R. Sedgewick, Algorithms. Addison-Wesley, 1983.

[14] C. Okasaki, The role of lazy evaluation in amortized data structures, in *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, 1996, pp. 62–71, DOI: 10.1145/232627.232636.

[15] M. L. Fredman, J. Komlós, and E. Szemerédi, Storing a sparse table with 0(1) worst case access time, *J. ACM*, vol. 31, no. 3, 1984, pp. 538–544. DOI: 10.1145/828.1884.

[16] R. Pagh and F. F. Rodler, Cuckoo hashing, *J. Algorithms*, vol. 51, no. 2, 2004, pp. 122–144.

[17] M. Drmota and R. Kutzelnigg, A precise analysis of cuckoo hashing, *ACM Trans. Algorithms*, vol. 8, no. 2, 2012, 11:1–11:36, DOI:10.1145/2151171.2151174.

[18] R. Kutzelnigg, Bipartite Random Graphs and Cuckoo Hashing, in *Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, P. Chassaing et al., Eds., Nancy, France, 2006, pp. 403–406.

[19] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, in *FOCS*, IEEE Computer Society, 1984, pp. 338–346.

[20] C. Faloutsos, Indexing and mining streams, in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, New York, 2004, p. 969.

[21] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine, Synopses for massive data: Samples, histograms, wavelets, sketches, *Foundations and Trends in Databases*, 2012, pp. 1–294.

[22] W. Johnson and J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space, in *Conference on Modern Analysis and Probability, American Mathematical Society*, 1984, pp. 189–206.

[23] M. N. Garofalakis, J. Gehrke, and R. Rastogi, Querying and mining data streams: You only get one look a tutorial, in *SIGMOD Conference*, 2002, p. 635.

[24] J. Abello, P. M. Pardalos, and M. G. C. Resende, Eds., *Handbook of Massive Data Sets*, Kluwer, Norwalk, 2002.

[25] P. Ciaccia, M. Patella, and P. Zezula, M-tree: An efficient access method for similarity search in metric spaces, in *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, Athens, Morgan Kauffman, 1997, pp. 426–435.

[26] P. Ciaccia, M. Patella, and P. Zezula, A cost model for similarity queries in metric spaces, in *Proceedings of the 16th ACM SIGACTSIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, Seattle, ACM Press, 1998, pp. 59–68.

[27] P. Indyk and R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, New York, ACM, 1998, pp. 604–613.

[28] P. Indyk, A sublinear time approximation scheme for clustering in metric spaces, in *Symposium on Foundations of Computer Science*, 2000, pp. 154–159.

[29] A. Gionis, P. Indyk, and R. Motwani, Similarity search in high dimensions via hashing, in *Proceedings of the 25th International Conference on Very Large Data Bases*, San Francisco, Morgan Kaufmann, 1999, pp. 518–529.

[30] W. Johnson and J. Lindenstrauss, Extensions of Lipschitz mappings into a Hilbert space, in *Conference on Modern Analysis and Probability*, New Haven, American Mathematical Society, 1984, pp. 189–206.

[31] S. Kumar, M. Mohri, and A. Talwalkar, Sampling methods for the nystrom method, *J. Mach. Learn. Res.*, vol. 13, no. 1, 2012, pp. 981–1006.

[32] P. Drineas and M. W. Mahoney, On the nystrom method for approximating a gram matrix for improved kernel-based learning, *J. Mach. Learn. Res.*, vol. 6, 2005, pp. 2153–2175.

[33] D. Achlioptas and F. Mcsherry, Fast computation of low-rank matrix approximations, *J. ACM*, vol. 54, no. 2, 2007.

[34] J. K. Uhlmann, Satisfying general proximity/similarity queries with metric trees, *Info. Proc. Lett.*, vol. 40, no. 4, 1991, pp. 175–179.

[35] J. K. Uhlmann, Metric trees, *Applied Mathematics Letters*, vol. 4, no. 5, 1991, pp. 61–62.

[36] E. V. Ruiz, An algorithm for finding nearest neighbours in (approximately) constant average time, *Pattern Recogn. Lett.*, vol. 4, no. 3, 1986, pp. 145–157.

[37] T. Liu, A. W. Moore, E. Gray, and K. Yang, An investigation of practical approximate nearest neighbor algorithms, in *NIPS2004*, MIT Press, 2004, pp. 825–832.

[38] S. Dasgupta and Y. Freund, Random projection trees and low dimensional manifolds, in *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, New York, ACM, 2008, pp. 537–546.

[39] P. N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, SIAM, 1993, pp. 311–321.

[40] A. Gionis, P. Indyk, and R. Motwani, Similarity search in high dimensions via hashing, in *Proceedings of the 25th International Conference on Very Large Data Bases*, San Francisco, Morgan Kaufmann, 1999, pp. 518–529.

[41] P. Indyk and R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, New York, ACM, 1998, pp. 604–613.

[42] R. Panigrahy, Entropy based nearest neighbor search in high dimensions, in *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, Philadelphia, SIAM, 2006, pp. 1186–1195.

[43] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, New York, ACM, 2004, pp. 253–262.

[44] A. Joly and O. Buisson, A posteriori multi-probe locality sensitive hashing, in *Multimedia*, A. El-Saddik et al., Eds., ACM, 2008, pp. 209–218.

[45] L. Paulevé, H. Jégou, and L. Amsaleg, Locality sensitive hashing: A comparison of hash function types and querying mechanisms, *Pattern Recogn. Lett.*, vol. 31, no. 11, 2010, pp. 1348–1358.

[46] R. Motwani, A. Naor, and R. Panigrahy, Lower bounds on locality sensitive hashing. *SIAM J. Discrete Math.*, vol. 21, no. 4, 2007, pp. 930–935.

[47] A. Andoni and P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, *Commun. ACM*, vol. 51, no. 1, 2008, pp. 117–122.

[48] M. S. Charikar, Similarity estimation techniques from rounding algorithms, in *Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing*, New York, ACM, 2002, pp. 380–388.

[49] L. Akoglu, R. Khandekar, V. Kumar, S. Parthasarathy, D. Rajan, and K.-L. Wu, Fast nearest neighbor search on large time-evolving graphs, in *Proceedings of the European Conference*

on Machine Learning and Knowledge Discovery in Databases, Nancy, France, Springer, 2014, pp. 17–33.

[50] J. L. Bentley, Multidimensional binary search trees used for associative searching, *Commun. ACM*, vol. 18, no. 9, 1975, pp. 509–517.

[51] J. H. Friedman, J. L. Bentley, and R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, *ACM Trans. Math. Softw.*, vol. 3, no. 3, 1977, pp. 209–226.

[52] D. P. Mehta and S. Sahni, *Handbook Of Data Structures And Applications*, Chapman & Hall, Boca Raton, 2004.

[53] P. Brass, *Advanced Data Structures*, Cambridge University Press, New York, 2008.

[54] R. Motwani and P. Raghavan, *Randomized Algorithms*. Cambridge University Press, New York, 1995.

[55] In, G. F. Italiano and R. Raman, Algorithms and theory of computation handbook, in *Topics in Data Structures*, Chapman & Hall, Boca Raton, 2020, p. 5.

[56] D. Golovin, Uniquely represented data structures with applications to privacy, PhD thesis CMU-CS-08–135, Carnegie Mellon University, Pittsburgh, 2008.

[57] J. L. Carter and M. N. Wegman, Universal classes of hash functions, in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, Boulder, ACM, 1977, pp. 106–112.

[58] R. Sprugnoli, Perfect hashing functions: A single probe retrieving method for static sets, *Commun. ACM*, vol. 20, no. 11, 1977, pp. 841–850.

[59] M. Aumüller, M. Dietzfelbinger, and P. Woelfel, Explicit and efficient hash families suffice for cuckoo hashing with a stash, *Algorithmica*, vol. 70, no. 3, 2014, pp. 428–456.

[60] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, *SIAM J. Comput.*, vol. 23, no. 4, 1994, pp. 738–761.

[61] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger, Hash, displace, and compress, *Lecture Notes in Computer Series*, Springer, 2009, pp. 682–693.

[62] R. Pagh and F. F. Rodler, Cuckoo hashing, *J. Algorithms*, vol. 51, no. 2, 2004, pp. 122–144.

[63] R. Kutzelnigg, Bipartite Random Graphs and Cuckoo Hashing, in *Proceedings of Fourth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, Nancy, France, 2006, pp. 403–406.

[64] B. H. Bloom, Space/time trade-offs in hash coding with allowable errors, *Commun. ACM*, vol. 13, no. 7, 1970, pp. 422–426.

[65] A. Kirsch and M. Mitzenmacher, Less hashing, same performance: Building a better bloom filter, *Random Struct. Algorithms*, vol. 33, no. 2, 2008, pp. 187–218.

[66] C. Martiénez and S. Roura, Randomized binary search trees, *J. ACM*, vol. 45, no. 2, 1998, pp. 288–323. DOI: 10.1145/274787.274812.

[67] B. Reed, The height of a random binary search tree, *J. ACM*, vol. 50, no. 3, 2003, pp. 306–332. DOI: 10.1145/765568.765571.

[68] L. Devroye, A note on the height of binary search trees, *J. ACM*, vol. 33, no. 3, 1986, pp. 489–498. DOI: 10.1145/5925.5930.

[69] M. Drmota, An analytic approach to the height of binary search trees II, *J. ACM*, vol. 50, no. 3, 2003, pp. 333–374, DOI: 10.1145/765568.765572.

[70] D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. ACM*, vol. 32, no. 3, 1985, pp. 652–686. DOI: 10.1145/3828.3835.

[71] A. Elmasry, On the sequential access theorem and deque conjecture for splay trees, *Theor. Comput. Sci.*, vol. 314, no. 3, 2004, pp. 459–466. DOI: 10.1016/j.tcs.2004.01.019.

[72] R. E. Tarjan, Sequential access in splay trees takes linear time, *Combinatorica*, vol. 5, no. 4, 1985, pp. 367–378. DOI: 10.1007/BF02579253.

[73] A. Blum, S. Chawla, and A. Kalai, Static optimality and dynamic search-optimality in lists and trees, in *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, SIAM, 2002, pp. 1–8.

[74] D. R. Morrison, Patricia and — Practical algorithms to retrieve information coded in alphanumeric, *J. ACM*, vol. 15, no. 4, 1968, pp. 514–534. DOI: 10.1145/321479.321481.

[75] E. Fredkin, Trie memory, *Commun. ACM*, vol. 3, no. 9, 1960, pp. 490–499. DOI: 10.1145/367390.367400.

[76] E. Ukkonen, Approximate string matching over suffix trees, in *Proceedings of the Fourth Annual Symposium on Combinatorial Pattern Matching*, Springer, 1993, pp. 228–242.

[77] G. Salton, E. A. Fox, and H. Wu, Extended Boolean information retrieval, *Commun. ACM*, vol. 26, no. 11, 1983, pp. 1022–1036. DOI: 10.1145/182.358466.

[78] J. H. Lee, Properties of extended Boolean models in information retrieval, in *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Dublin, Springer, 1994, pp. 182–190.

[79] J. L. Bentley and J. H. Friedman, Data structures for range searching, *ACM Comput. Surv.*, vol. 11, no. 4, 1979, pp. 397–409. DOI: 10.1145/356789.356797.

[80] J. L. Bentley, Multidimensional binary search trees used for associative searching, *Commun. ACM*, vol. 18, no. 9, 1975, pp. 509–517, DOI: 10.1145/361002.361007.

[81] E. Langetepe and G. Zachmann, *Geometric Data Structures for Computer Graphics*. A. K. Peters, Natick, 2006.

[82] O. Fries, K. Mehlhorn, and S. Näher, Dynamization of geometric data structures, in *Proceedings of the First Annual Symposium on Computational Geometry*, Baltimore, ACM, 1985, pp. 168–176. DOI: 10.1145/323233.323256.

[83] G. E. Blelloch, D. Golovin, and V. Vassilevska, Uniquely represented data structures for computational geometry, in *Proceedings of the Eleventh Scandinavian Workshop on Algorithm Theory*, Gothenburg, Springer, 2008, pp. 17–28.

[84] T. M. Chan, K. Larsen, and M. Pătraşcu, Orthogonal range searching on the RAM, revisited, in *Proceedings of the Twenty-seventh ACM Symposium on Computational Geometry*, arXiv:1011.5200, 2011, pp. 354–363.

[85] E. D. Demaine, D. Harmon, J. Iacono, D. Kane, and M. Pătraşcu, The geometry of binary search trees, in *Proceedings of the 20th ACM/SIAM Symposium on Discrete Algorithys*,Society for Industrial and Applied Mathematics, Philadelphia, 2009, pp. 496–505.

[86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, Making data structures persistent, *J. Comput. Syst. Sci.*, vol. 38, no. 1, 1989, pp. 86–124. DOI: http://dx.doi.org/10.1016/0022-0000(89)90034-2">10.1016/0022-0000(89)90034-2.

[87] N. I. Sarnak, Persistent data structures, AAI8706779, PhD thesis, New York, 1986.

[88] C. Okasaki. *Purely Functional Data Structures*, Cambridge University Press, New York, 1999.

[89] J. Iacono and M. Pătraşcu, Using hashing to solve the dictionary problem (in external memory), in *Proceedings of the Twenty-third ACM/SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 2012.

[90] M. Dietzfelbinger and F. Meyer auf der Heide, A new universal class of hash functions and dynamic hashing in real time, in *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, Warwick, UK, Springer, 1990, pp. 6–19.

[91] G. H. Gonnet, Expected length of the longest probe sequence in hash code searching, *J. ACM*, vol. 28, no. 2, 1981, pp. 289–304. DOI: 10.1145/322248.322254.

[92] M. Mitzenmacher, The power of two choices in randomized load balancing, *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, 2001, pp. 1094–1104. DOI: 10.1109/71.963420.

[93] A. Ostlin and R. Pagh, Uniform hashing in constant time and linear space, in *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, San Diego, San Diego, ACM, 2003, pp. 622–628. DOI: 10.1145/780542.780633.

[94] A. Siegel, On universal classes of extremely random constant-time hash functions, *SIAM J. Comput.*, vol. 33, no. 3, 2004, pp. 505–543. DOI: 10.1137/S0097539701386216.

[95] M. Aumüller, T. Christiani, R. Pagh, and F. Silvestri, Distance-sensitive hashing, in *PODS, the Symposium on Principles of Database Systems*, ACM, 2018, pp. 89–104.

[96] R. Pagh, N. Pham, F. Silvestri, and M. Stöckel, I/O-efficient similarity join, *Algorithmica*, vol. 78, no. 4, 2017, pp. 1263–1283.

[97] J. Gudmundsson and R. Pagh, Range-efficient consistent sampling and locality-sensitive hashing for polygons, 2017, arXiv preprint arXiv:1701.05290.

[98] T. D. Ahle, M. Aumüller, and R. Pagh, Parameter-free locality sensitive hashing for spherical range reporting, in SIAM, 2017, pp. 239–256.

[99] M. Goswami, R. Pagh, F. Silvestri, and J. Sivertsen, Distance sensitive bloom filters without false negatives, in SIAM, 2017, pp. 257–269.

[100] R. E. Tarjan and A. C. Yao, Storing a sparse table, *Commun. ACM*, vol. 22, no. 11, 1979, pp. 606–611.

[101] R. J. Lipton, A. L. Rosenberg, and A. C. Yao, External hashing schemes for collections of data structures, *J. ACM*, vol. 27, no. 1, 1980, pp. 81–95.

[102] M. Bădoiu and E. D. Demaine, A simplified and dynamic unified structure, in *Theoretical Informatics*, Springer, Heidelbert, 2004, pp. 466–473.

[103] R. Sundar, On the deque conjecture for the splay algorithm, *Combinatorica*, vol. 12, no. 1, 1992, pp. 95–124. DOI: 10.1007/BF01191208.

[104] R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. S. Rao, Asymptotically optimal encodings of range data structures for selection and top-k queries, *ACM Trans. Algorithms*, vol. 13, no. 2,

2017, 28:1–28:31.

[105] P. Bose, K. Douïeb, J. Iacono, and S. Langerman, The power and limitations of static binary search trees with lazy finger, *Algorithmica*, vol. 76, no. 4, 2016, pp. 1264–1275.

[106] E. D. Demaine, J. Iacono, and S. Langerman, Worst-case optimal tree layout in external memory, *Algorithmica*, vol. 72, no. 2, 2015, pp. 369–378.

[107] P. Bose, K. Douïeb, J. Iacono, and S. Langerman, The power and limitations of static binary search trees with lazy finger, in *ISAAC: International Symposium on Algorithms and Computation*, Springer, 2014, pp. 181–192.

[108] J. Iacono and Pătraşcu, Using hashing to solve the dictionary problem, in SIAM, 2012, pp. 570–582.

[109] S. Collette, J. Iacono, and S. Langerman, Confluent persistence revisited, in SIAM, 2012, pp. 593–601.

[110] G. S. Brodal, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro, Cache-oblivious dynamic dictionaries with update/query tradeoffs, in SIAM, 2010, pp. 1448–1456.

[111] K. Mehlhorn, Best possible bounds for the weighted path length of optimum binary search trees, in *Automata Theory and Formal Languages*, vol. 33, Springer, 1975, pp. 31–41.

[112] K. Mehlhorn, Nearly optimal binary search trees, *Acta Inf.*, vol. 5, 1975, pp. 287–295.

[113] K. Mehlhorn, Dynamic binary search, in *International Colloquium on Automata, Languages and Programming (ICALP)*, vol. 52, Springer, 1977, pp. 323–336.

[114] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*. Monographs on Theoretical Computer Science. Springer, 1984, vol. 3.

[115] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Monographs on Theoretical Computer Science. Springer, 1984, vol. 2.

[116] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*. Monographs on Theoretical Computer Science. Springer, 1984, vol. 1.

[117] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, in *Symposium on Foundations of Computer Science*, 1988, pp. 524–531.

[118] K. Mehlhorn and A. K. Tsakalidis, Data structures, in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, Elsevier, 1990, pp. 301–342.

[119] C. C. Wang, J. Derryberry, and D. D. Sleator, $O(\log \log n)$-competitive dynamic binary search trees, in *Symposium on Discrete Algorithms (SODA), ACM*, 2006, pp. 374–383.

[120] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, Making data structures persistent, *J. Comput. Syst. Sci.*, vol. 38, no. 1, 1989, pp. 86–124.

[121] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The pairing heap: A new form of self-adjusting heap, *Algorithmica*, vol. 1, no. 1, 1986, pp. 111–129.

[122] D. D. Sleator and R. E. Tarjan, Self-adjusting heaps, *SIAM J. Comput.*, vol. 15, no. 1, 1986, pp. 52–69.

[123] D. D. Sleator, and R. E. Tarjan, Self-adjusting binary trees, in *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, ACM, 1983, pp. 235–245.

[124] L. Arge, G. S. Brodal, J. Truelsen, and C. Tsirogiannis, An optimal and practical cache-oblivious algorithm for computing multiresolution rasters, in *European Symposium on Algorithms (ESA)*, vol. 8125, Springer, 2013, pp. 61–72.

[125] L. Arge and M. Thorup, Ram-efficient external memory sorting, in *ISAAC: International Symposium on Algorithms and Computation*, vol. 8283, Springer, 2013, pp. 491–501.

[126] L. Arge and K. G. Larsen, I/O-efficient spatial data structures for range queries, *SIGSPATIAL Special*, vol. 4, no. 2, 2012, pp. 2–7.

[127] P. K. Agarwal, L. Arge, and K. Yi, I/o-efficient batched union-find and its applications to terrain analysis, *ACM Trans. Algorithms*, vol. 7, no. 1, 2010, 11:1–11:21.

[128] L. Arge, M. de Berg, and H. J. Haverkort, Cache-oblivious R-trees, *Algorithmica*, vol. 53, no. 1, 2009, pp. 50–68.

[129] L. Arge, External geometric data structures, in *International Conference on Computing and Combinatorics*, vol. 3106, Springer, 2004, p. 1.

[130] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi, The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, 2008, p. 9.

[131] L. Arge, G. S. Brodal, and R. Fagerberg, Cache-oblivious data structures, in *Handbook of Data Structures and Applications*, Chapman & Hall, Boca Raton, 2004.

[132] L. Arge, The buffer tree: A technique for designing batched external data structures, *Algorithmica*, vol. 37, no. 1, 2003, pp. 1–24.

[133] J. S. Vitter, Algorithms and data structures for external memory, *Found. Trends Theor. Comput. Sci.*, vol. 2, no. 4, 2008, pp. 305–474. DOI: 10.1561/0400000014.

[134] H. Zhang, Y. Wen, H. Xie, and N. Yu, *Distributed Hash Table: Theory, Platforms and Applications*, Springer, Heldelberg, 2013.

[135] C. Leng, J. Wu, J. Cheng, X. Zhang, and H. Lu, Hashing for distributed data, in *Proceedings of the Thirty-second International Conference on Machine Learning*, Lille, France, 2015, pp. 1642–1650.

[136] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, A practical concurrent binary search tree, *SIGPLAN Not.*, vol. 45, no. 5, 2010, pp. 257–268. DOI: 10.1145/1837853.1693488.

[137] B. K. Shrivastava, G. Khataniar, and D. Goswami, Binary search tree: An efficient overlay structure to support range query, in *Proceedings of Twenty-seventh International Conference on Distributed Computing Systems Workshops, (ICDCSW'07)*, IEEE, 2007, p. 77. DOI: 10.1109/ICDCSW.2007.99.

[138] J. Aspnes and G. Shah, Skip graphs, *ACM Trans. Algorithms*, vol. 3, no. 4, 2007. DOI: 10.1145/1290672.1290674.

[139] A. Disterhoft, A. Funke, and K. Graffi, Packetskip: Skip graph for multidimensional search in structured peer-to-peer systems, in *Proceedings of Eleventh IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, IEEE, September 2017, pp. 21–30. DOI: 10.1109/SASO.2017.11.

[140] C. C. Aggarwal, *Data Streams: Models and Algorithms (Advances in Database Systems)*, Springer, Heidelberg, 2006.

[141] S. Muthukrishnan, Data streams: Algorithms and applications, *Found. Trends Theor. Comput. Sci.*, vol. 1, no. 2, 2005, pp. 117–236. DOI: 10.1561/0400000002.

[142] W. Hu and B. Zhang, Study of sampling techniques and algorithms in data stream environments, in *Proceedings of the Ninth International Conference on Fuzzy Systems and Knowledge Discovery*, IEEE, 2012, pp. 1028–1034. DOI: 10.1109/FSKD.2012.6234278.

[143] G. Cormode, Data sketching, *Commun. ACM*, vol. 60, no. 9, 2017, pp. 48–55. DOI: 10.1145/3080008.

[144] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine, Synopses for massive data: Samples, histograms, wavelets, sketches, *Foundations and Trends in Databases*, vol. 4, no. 1–3, 2012, pp. 1–294. DOI: 10.1561/1900000004.

[145] N. Alon, Y. Matias, and M. Szegedy, The space complexity of approximating the frequency moments, in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, Philadelphia, ACM, 1996, pp. 20–29. DOI: 10.1145/237814.237823.

[146] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, Models and issues in data stream systems, in *Proceedings of the Twenty-first Symposium on Principles of Database Systems*, Madison, Wisconsin: ACM, 2002, pp. 1–16. DOI: 10.1145/543613.543615.

[147] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, Surfing wavelets on streams: One-pass summaries for approximate aggregate queries, in *Very Large Database Conference*, 2001, pp. 79–88.

[148] P. Chaovalit, A. Gangopadhyay, G. Karabatis, and Z. Chen, Discrete wavelet transform-based time series analysis and mining, *ACM Comput. Surv.*, vol. 43, no. 2, 2011, 6:1–6:37. DOI: 10.1145/1883612.1883613.

[149] S. Guha, N. Koudas, and K. Shim, Data-streams and histograms, in *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, Greece, ACM, 2001, pp. 471–475, DOI: 10.1145/380752.380841.

[150] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, Histosketch: Fast similarity-preserving sketching of streaming histograms with concept drift, in *2017 IEEE International Conference on Data Mining (ICDM)*, 2017, pp. 545–554. DOI: 10.1109/ICDM.2017.64.

[151] R. B. Rusu, J. Bandouch, F. Meier, I. Essa, and M. Beetz, Human Action Recognition using Global Point Feature Histograms and Action Shapes, *Advanced Robotics journal, Robotics Society of Japan (RSJ)*, 2009.

[152] M. Rabin, *Fingerprinting by Random Polynomials*, ser. Center for Research in Computing Technology: Center for Research in Computing Technology. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[153] A. Z. Broder, Some applications of rabin's fingerprinting method, in *Sequences II: Methods in Communications, Security, and Computer Science*, Springer-Verlag, 1993, pp. 143–152.

[154] R. Rivest, *The md5 message-digest algorithm*, United States, 1992.

[155] D. Rachmawati, J. T. Tarigan, and A. B. C. Ginting, A comparative study of message digest 5(md5) and sha256 algorithm, *Journal of Physics: Conference Series*, vol. 978, no. 1, 2018, p.

012 116.

[156] P. Rogaway and J. Steinberger, Constructing cryptographic hash functions from fixed-key blockciphers, in *Advances in Cryptology - CRYPTO 2008*, D. Wagner, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 433–450.

[157] B. Preneel, Analysis and design of cryptographic hash functions, PhD thesis, 1993.

[158] F. Bauspiess and F. Damm, Requirements for cryptographic hash functions, *Comput. Secur.*, vol. 11, no. 5, 1992, pp. 427–437, DOI: 10.1016/0167-4048(92)90007-E.

[159] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, http://bitcoin.org/bitcoin.pdf, 2008.

[160] M. Swan, *Blockchain: Blueprint for a New Economy*, 1st. O'Reilly Media, Inc., 2015.

[161] D. Tapscott and A. Tapscott, *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World*. Brilliance Audio, 2016.

[162] A. Narayanan, Blockchains: Past, present, and future, in *PODS*, ACM, 2018, p. 193.

[163] A. Narayanan and J. Clark, Bitcoin's academic pedigree, *Commun. ACM*, vol. 60, no. 12, 2017, pp. 36–45.

[164] R. C. Merkle, A certified digital signature, in *Proceedings on Advances in Cryptology*, ser. CRYPTO '89, Santa Barbara, California, USA: Springer-Verlag New York, Inc., 1989, pp. 218–238.

[165] R. L. Rivest, A. Shamir, and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Commun. ACM*, vol. 21, no. 2, 1978, pp. 120–126, DOI: 10.1145/359340.359342.

[166] R. C. Merkle, A digital signature based on a conventional encryption function, in *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, ser. CRYPTO '87, London, UK, UK: Springer-Verlag, 1988, pp. 369–378.

[167] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia, Dynamic provable data possession, *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 4, 2015, 15:1–15:29.

[168] M. Tahir and S. Ahmed, Tree-combined trie: A compressed data structure for fast ip address lookup, *International Journal of Advanced Computer Science and Applications*, vol. 6, no. 12, 2015. DOI: 10.14569/IJACSA.2015.061223.

[169] B. Pinkerton, Finding what people want: Experiences with the WebCrawler, in *Proceedings of the 2nd International World Wide Web*, Anonymous, Ed., ser. Online & CDROM review: the international journal of, vol. 18(6), Medford, NJ, USA: Learned Information, 1994.

[170] J. Cho, H. Garcia-Molina, and L. Page, Efficient crawling through URL ordering, *Computer Networks*, vol. 30, no. 1–7, 1998, pp. 161–172.

[171] H. Garcia-Molina, Challenges in crawling the web, in *BNCOD*, ser. Lecture Notes in Computer Science, vol. 2712, Springer, 2003, p. 3.

[172] A. Heydon and M. Najork, Mercator: A scalable, extensible web crawler, *World Wide Web*, vol. 2, no. 4, 1999, pp. 219–229, DOI: 10.1023/A:1019213109274.

[173] S. Melnik, S. Raghavan, B. Yang, and H. Garcia-Molina, Building a distributed full-text index for the web, in *WWW*, ACM, 2001, pp. 396–406.

[174] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[175] C. S. Pabla, Completely fair scheduler, *Linux J.*, vol. 2009, no. 184, 2009.

[176] J. C. Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias, Acfs: A completely fair scheduler for asymmetric single-isa multicore systems, in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, Salamanca, Spain: ACM, 2015, pp. 2027–2032, DOI: 10.1145/2695664.2695714.

[177] S. Iyer, R. R. Kompella, and N. McKeowa, Analysis of a memory architecture for fast packet buffers, in *2001 IEEE Workshop on High Performance Switching and Routing (IEEE Cat. No.01TH8552)*, 2001, pp. 368–373. DOI: 10.1109/HPSR.2001.923663.

[178] R. Motwani and D. Thomas, Caching queues in memory buffers, in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '04, New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2004, pp. 541–549.

[179] W.-c. Feng, K. G. Shin, D. D. Kandlur, and D. Saha, The blue active queue management algorithms, *IEEE/ACM Trans. Netw.*, vol. 10, no. 4, 2002, pp. 513–528, DOI: 10.1109/TNET.2002.801399.

[180] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, Summary cache: A scalable wide-area web cache sharing protocol, *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, 2000, pp. 281–293, DOI: 10.1109/90.851975.

[181] D. Giampaolo, *Practical File System Design with the Be File System*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[182] A. Guttman, R-trees: A dynamic index structure for spatial searching, *SIGMOD Rec.*, vol. 14, no. 2, 1984, pp. 47–57, DOI: 10.1145/971697.602266.

[183] T. Matsuyama, L. V. Hao, and M. Nagao, A file organization for geographic information systems based on spatial proximity, *Computer Vision, Graphics, and Image Processing*, vol. 26, no. 3, 1984, pp. 303–318. DOI: 10.1016/0734-189X(84)90215-9.

[184] S. D. Roth, Ray Casting for Modeling Solids, vol. 18, no. 2, 1982, pp. 109–144. DOI: 10.1016/0146664X(82)90169–1.

[185] M. Sugano, Indoor localization system using rssi measurement of wireless sensor network based on zigbee standard, in *Wireless and Optical Communications*, IASTED/ACTA Press, 2006, pp. 1–6.

[186] G. Cormode and M. Muthukrishnan, Approximating data with the count-min sketch, *IEEE Softw.*, vol. 29, no. 1, 2012, pp. 64–69, DOI: 10.1109/MS.2011.127.

[187] A. C. Gilbert and P. Indyk, Sparse recovery using sparse matrices, *Proceedings of the IEEE*, vol. 98, no. 6, 2010, pp. 937–947.

[188] A. Z. Broder, On the resemblance and containment of documents, in *In Compression and Complexity of Sequences (SEQUENCES'97*, IEEE Computer Society, 1997, pp. 21–29.

[189] O. Chum, J. Philbin, and A. Zisserman, Near duplicate image detection: Min-hash and tf-idf weighting. In *BMVC*, M. Everingham, C. J. Needham, and R. Fraile, Eds., British Machine Vision Association, 2008, pp. 1–10.

[190] L. F. Mackert and G. M. Lohman, R* optimizer validation and performance evaluation for local queries, *SIGMOD Rec.*, vol. 15, no. 2, 1986, pp. 84–95.

[191] L. Zhang and Y. Guan, Detecting click fraud in pay-per-click streams of online advertising networks, in *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ser. ICDCS '08, Washington, DC, USA: IEEE Computer Society, 2008, pp. 77–84.

[192] S. Nilsson and G. Karlsson, Ip-address lookup using lc-tries, *IEEE J.Sel. A. Commun.*, vol. 17, no. 6, 2006, pp. 1083–1092.

[193] W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *Commun. ACM*, vol. 33, no. 6, 1990, pp. 668–676.

[194] R. Bayer and E. McCreight, Organization and maintenance of large ordered indices, in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIG-FIDET '70, Houston, Texas: ACM, 1970, pp. 107–141.

[195] A. Haar, Zur Theorie der orthogonalen Funktionensysteme, Mathematische Annalen, vol. 69, no. 3, 1910, pp. 331–371, DOI:10.1007/BF01456326

[196] Aburrous et al. Predicting phishing websites using classification mining techniques with experimental case studies. In *Proceedings of 7th International Conference on Information Technology: New Generations*, IEEE, 2010.

[197] C. Liu, C. Yang, X. Zhang, and J. Chen, External integrity verification for outsourced big data in cloud and IoT: A big picture, *Future generation computer systems*, vol. 49, 2015, pp. 58–67.

[198] Belenky, Andrey, and Nirwan Ansari. IP traceback with deterministic packet marking. *IEEE communications letters 7.4*, 2003, 162–164.

[199] C. Albuquerque, B. J. Vickers, and T. Suda, Network border patrol, in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, IEEE, vol. 1, 2000, pp. 322–331.

[200] I. Stoica, S. Shenker, and H. Zhang, Core-Stateless Fair Queuing: A Scalable Architecture to Approximate Fair Bandwidth Allocations in High Speed Networks, *IEEE/ACM Transaction on Networking*, vol. 11, no. 1, 2003, pp. 33–46.

[201] A. Vahdat, and B. David, Epidemic routing for partially connected ad hoc networks, 2000.

[202] N. Borisov, Computational puzzles as sybil defenses. *Sixth IEEE International Conference on Peer-to-Peer Computing (P2P'06)*, IEEE, 2006.

[203] A. Singh, M. Castro, P. Druschel, and A. Rowstron, Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, ACM, September 2004, p. 21.

[204] L. Ganesh, and Y. Z. Ben, Identity theft protection in structured overlays. *1st IEEE ICNP Workshop on Secure Network Protocols*, (NPSec), IEEE, 2005.

[205] M. W. Berry, T. D. Susan, and A. L. Todd, Computational methods for intelligent information access. Supercomputing'95: *In Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, IEEE, 1995.

[206] G. H. Golub, and C. F. Van Loan. *Matrix Computations*, JHU press, Maryland, Vol. 3, 2012.

[207] M. Agate, R. L. Grimsdale, P. F. Lister. The HERO Algorithm for Ray-Tracing Octrees. In: Grimsdale, R.L., Straßer, W. (eds), *Advances in Computer Graphics Hardware IV. Eurographic Seminars (Tutorials and Perspectives in Computer Graphics)*, Springer, Berlin, Heidelberg, 1991.

Index