COMPUTER ORIENTED NUMERICAL ANALYSIS

H. Pandey
R.K.Mishra
Rakesh Kumar Yadav

Computer Oriented Numerical Analysis

H. Pandey
R.K.Mishra
Rakesh Kumar Yadav

Computer Oriented Numerical Analysis

H. Pandey
R.K.Mishra
Rakesh Kumar Yadav

Dominant
Publishers & Distributors Pvt Ltd
New Delhi, INDIA

Knowledge is Our Business

COMPUTER ORIENTED NUMERICAL ANALYSIS
By H. Pandey & R.K.Mishra, Rakesh Kumar Yadav

This edition published by Dominant Publishers And Distributors (P) Ltd
4378/4-B, Murarilal Street, Ansari Road, Daryaganj,
New Delhi-110002.

ISBN: 978-93-82007-34-0

Edition: 2022 (Revised)

Dominant
Publishers & Distributors Pvt Ltd

Registered Office: 4378/4-B, Murari Lal Street, Ansari Road,
Daryaganj, New Delhi - 110002.
Ph. +91-11-23281685, 41043100, Fax: +91-11-23270680

Production Office: "Dominant House", G - 316, Sector - 63, Noida,
National Capital Region - 201301.
Ph. 0120-4270027, 4273334

e-mail: dominantbooks@gmail.com
info@dominantbooks.com

w w w . d o m i n a n t b o o k s . c o m

CONTENTS

CHAPTER 1

INTRODUCTION TO DATA STRUCTURE

Rakesh Kumar Yadav, Associate Professor
Maharishi School of Engineering & Technology,Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-rkymuit@gmail.com

ABSTRACT:

This book offers implementations of both standard and rare algorithms in language-independent pseudocode that is suitable for straightforward porting to the majority of imperative programming languages. It is not a definitive work on the theory of algorithms and data structures.It is quite likely that our implementations deviate from those generally accepted because this book mostly covers implementations developed by the authors themselves based on the notions upon which the individual algorithms are built.This book should be used in conjunction with a book on the same topic that includes formal proofs of the relevant algorithms. To make the book more accessible to a wider audience, we employ the abstract big Oh notation to illustrate the run time complexity of algorithms in this book.

KEYWORDS:

Algorithmic Design, Data Structure, Efficiency Optimization, Fundamental Concepts, Organized Data.

INTRODUCTION

An expertly designed framework for arranging, processing, accessing, and storing data is called a data structure. Data structures come in both simple and complex forms, all of which are made to organize data for a certain use. Users find it simple to obtain and use the data they need in the right manner thanks to data structures.Although we made few assumptions about the reader when writing this book, some were essential to keep it as brief and user-friendly as possible.

The reader is assumed to be familiar with the following:

1. Big Oh symbol
2. A language for imperative programming
3. Concepts of object orientation

Big Oh Symbol:

Big O notation is a type of mathematical notation that expresses how a function limits itself when the argument goes to zero or infinity. It belongs to the group of notations known as Bachmann-Landau notation or asymptotic notation, which was created by Paul Bachmann, Edmund Landau, and others. The complexity of your code is expressed in algebraic terms using the Big O notation [1]–[3].We may look at a common example, O(n2), which is typically termed "Big O squared," to further grasp what Big O notation is. Here, the input size is denoted by the letter "n," and the function "g(n) = n2" within the "O()" tells us how complicated the method is in relation to the input size.

The selection sort algorithm is a classic example of an algorithm with O(n2) complexity. An iterative sorting technique called selection sort makes sure that each entry at position i is the ith lowest or biggest member in the list.

A Language for Imperative Programming:

The sequence of operations is critical for imperative programming since it precisely describes the processes that determine how the program will implement desired functionality. C++, Java, Fortran, and other imperative programming languages are examples.We are specific about this need since it underlies all of our implementations, which follow an imperative thinking paradigm. To create efficient solutions with regard to your functional language, whether it be Haskell, F#, etc., you will need to use a variety of functional paradigm principles.

The virtual machines that target two of the languages we've discussed C# and Java offer diverse features including security sandboxing and memory management through garbage collection methods. The translation of our implementations into various languages is simple. You must keep in mind to utilize pointers for certain items while porting to C++. For instance, in the context of a controlled environment, we say that a linked list node has a reference to the subsequent node. In C++, the reference should be seen as a pointer to the subsequent node, and so on. These subtleties won't be a problem for programmers who have some experience with their respective languages, which is why we stress that the reader needs to be familiar with at least one imperative language in order to successfully port the mock implementations.

Concepts of Object Orientation:

A computer programming paradigm known as object-oriented programming (OOP) arranges the architecture of software around data or objects rather than functions and logic. An object is a data field with particular characteristics and behavior.

Object-oriented programming is based on four core ideas:

1. Inheritance,
2. Encapsulation,
3. Polymorphism,
4. Data abstraction.

Understanding all of these is necessary in order to comprehend OOPs.

DISCUSSION

The outcome of such a conversation will reveal more about the high-level algorithm design than about the efficiency of the algorithm. Play the scene out in your thoughts, except this time each developer mentions the asymptotic run time of their algorithm in addition to discussing algorithm design. The latter method gives you vital efficiency information that helps you choose an algorithm for a goal more effectively in addition to giving you a strong overall understanding of the algorithm design.Some readers could even be employed by a product team where they get budgets for each feature. Each feature has a budget that corresponds to its current time limit. Saving time in one feature doesn't always translate into better performance in the other features. Imagine that you are a member of the team designing the procedures that will effectively spin up everything needed when the program is started. You are working on an application.

The effectiveness of each algorithm used in this example during startup is essential for a successful product. Even without these budgets, you should still aim for the best solutions.

Data Structure:

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures [4], [5].

1. **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
    a. Examples of linear data structures are array, stack, queue, linked list, etc.
2. **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
    a. An example of this data structure is an array.
3. **Dynamic data structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
    a. Examples of this data structure are queue, stack, etc.
4. **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
    a. Examples of non-linear data structures are trees and graphs.

Algorithm:

A data structure is used for more than just data organization. Additionally, it is employed for data processing, retrieval, and archiving. Almost all software systems and programs that have been built employ many basic and complex forms of data structures. Therefore, we need to be well-versed in data structures.

The term "linear data structure" refers to a data structure in which the components are ordered sequentially or linearly, with each element being connected to its immediately preceding and following neighboring elements. The array, stack, queue, linked list, and other linear data structures are a few examples.Data structures that are static have a set amount of memory. A static data structure's components are simpler to access.

An array is a kind of this data structure

Dynamic data structures: The size is variable with dynamic data structures. It may be randomly updated while the program is running, which is thought to be efficient given the code's memory (and space) complexity. Queue, stack, and other such data structures are examples.Data structures that do not arrange data pieces sequentially or linearly are referred to as non-linear data structures. We cannot explore every element of a non-linear data structure in a single operation [6]–[8].

Trees and graphs are two examples of non-linear data structures

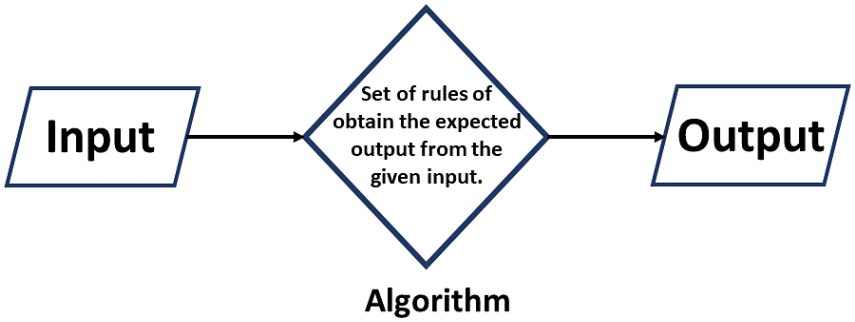


Figure 1: Working of an Algorithm[simplilearn.com].

Testing:

To construct the pseudocode algorithm, all the data structures and algorithms were evaluated on paper using a minimal test-driven development methodology. These tests are then converted into unit tests by individually fulfilling each one. We consider the method to have been sufficiently tested if all test cases have been gradually satisfied. Most algorithms have quite clear-cut situations that must be satisfied. Some, however, have several areas that may be more difficult to fulfill. When using such approaches, we will highlight the challenging test cases and the related sections of pseudocode that fulfill each case. Figure 1 shown working of an algorithm.

As you grow more familiar with the real issue, you will naturally be able to see potential issues with the implementation of your methods. In certain circumstances, this will provide an overwhelming number of issues, which will seriously impede your ability to develop an algorithm. When you are swamped with so many worries, take another look at the big picture and break it down into smaller issues. It is far simpler to solve the minor issues first and then put them together than it is to fill your head with too many little details [9], [10].Unit tests are the sole testing method that we employ in the execution of all the instructions in this book. We encourage the reader to see Appendix D, which goes into further detail on testing, because unit tests are such an essential component of producing software that is a little bit more stable.

CONCLUSION

A data structure is described as "a data format that aids developers in organizing, managing, and storing information." The relationships between the items, the actions that the structure supports, and the actual values of the items are used to explain computer data structures. Although many algorithms and data structures are included in the major programming languages, developers frequently invent new ones for applications.There are some linear data structures. This indicates that the elements are set up in a logical sequence. When the connections between the elements are crucial, non-linear methods should be applied. Arrays, stacks, queues, records, trees, graphs, linked lists, and hash tables are some of the most frequently used data structures. Choosing a data structure to employ involves a lot of different considerations. However, performance, usability, and memory use are the most crucial factors.

For resilience and reusability, data structures provide systematic data storage. Correct data structure construction makes it simple to compute and manipulate data as needed. There are hundreds of background software applications operating, and each one makes use of data

structures to increase efficiency with increasing space and time complexity. The accuracy of it determines how well the data and code work. Algorithms and data structures are the fundamental building blocks of computer coding. The implementation of the issue solution makes the most of the data structure and algorithm.

Data structures is a very broad field that includes much more than simply stacks, queues, and linked lists. There are several other data structures, such as Maps, Hash Tables, Graphs, Trees, and so on. Each data structure has its own benefits and drawbacks, and it must be utilized in accordance with the application's requirements. A student of computer science should at the very least be familiar with the fundamental data structures and the operations that go with them. Many of these data structures are included by default in many high level and object-oriented programming languages, such as C#, Java, and Python. As a result, understanding how things operate is crucial.

REFERENCES:

[1] C. A. Shaffer, "A Practical Introduction to Data Structures and Algorithm Analysis Third Edition (C ++ Version)," *Programming*, 2010.

[2] W. H. Finch, J. E. Bolin, and K. Kelley, "Introduction to Multilevel Data Structure," in *Multilevel Modeling Using R*, 2018. doi: 10.1201/b17096-5.

[3] J. A. Storer, *An Introduction to Data Structures and Algorithms*. 2002. doi: 10.1007/978-1-4612-0075-8.

[4] S. Sridharan and R. Balakrishnan, "Introduction to Algorithms and Data Structures," in *Foundations of Discrete Mathematics with Algorithms and Programming*, 2019. doi: 10.1201/9781351019149-5.

[5] "Enzymes: a practical introduction to structure, mechanism, and data analysis," *Choice Rev. Online*, 2001, doi: 10.5860/choice.38-3881.

[6] A. El-Atawy, "An introduction to data structures and algorithms by James A. Storer Birhauser," *ACM SIGACT News*, 2010, doi: 10.1145/1753171.1753175.

[7] K. Weiskamp, "Introduction to Data Structures," in *Advanced Programming with Microsoft Quickc*, 1989. doi: 10.1016/b978-0-12-742684-6.50007-8.

[8] M. T. Goodrich, "Data Structures and Algorithms in Python," *Wiley*, 2010.

[9] B. Work, "Advanced Smalltalk: Introduction to Data Structures and Algorithms with CAA," *Eur. J. Inf. Syst.*, 1997, doi: 10.1057/palgrave.ejis.3000268.

[10] A. Zare, A. Ozdemir, M. A. Iwen, and S. Aviyente, "Extension of PCA to higher order data structures: An introduction to tensors, tensor decompositions, and tensor PCA," *Proc. IEEE*, 2018, doi: 10.1109/JPROC.2018.2848209.

CHAPTER 2

A BRIEF DISCUSSION ON ANALYSIS OF ALGORITHMS

Vaishali Singh, Assistant Professor
Maharishi School of Engineering & Technology,Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-singh.vaishali05@gmail.com

ABSTRACT:

The Analysis of Algorithms stands as a critical discipline in computer science, enabling us to gauge the efficiency and performance of algorithms under different conditions. This chapter delves into the essence of algorithmic analysis, unraveling its significance, methodologies, and mathematical foundations. By exploring key concepts such as time complexity, space complexity, and big O notation, along with real-world applications, readers gain insights into the art of evaluating algorithms and making informed design decisions.

KEYWORDS:

Asymptotic Analysis, Algorithm Performance, Big O Notation, Computational Complexity, Efficiency, Space Complexity, Time Complexity.

INTRODUCTION

As algorithms form the bedrock of modern computational systems, the Analysis of Algorithms emerges as a crucial discipline that illuminates their performance and efficiency. The efficiency of an algorithm not only impacts its execution time but also influences resource utilization, scalability, and overall system behavior. This chapter embarks on a journey to understand the importance of algorithmic analysis, exploring its types, characteristics, real-world applications, and the key components that underlie its significance.

Types, Characteristics, Applications, and Key Components:

Types of Algorithmic Analysis:

The Analysis of Algorithms encompasses two fundamental dimensions: time complexity and space complexity. Time complexity measures the execution time of an algorithm as a function of the input size, providing insights into how the algorithm's efficiency scales with larger inputs. Space complexity, on the other hand, assesses the memory space an algorithm requires to solve a problem, reflecting its memory usage characteristics [1]–[3].

Characteristics and Importance:

The primary goal of algorithmic analysis is to provide a rigorous understanding of how an algorithm performs under different input conditions. By quantifying the relationship between input size and resource utilization, we gain insights into the algorithm's scalability, efficiency, and suitability for various applications. Algorithmic analysis empowers us to make informed design decisions, optimize code, and predict system behavior in real-world scenarios.

Applications in the Real World:

The impact of algorithmic analysis reverberates across industries and domains. In software development, it guides the choice of algorithms for specific tasks, ensuring that applications are responsive and resource-efficient. In system design, it aids in identifying performance bottlenecks and devising strategies for optimization. In theoretical computer science, algorithmic analysis provides a framework for comparing and contrasting different algorithms, allowing us to uncover their strengths and limitations.

Key Components of Algorithmic Analysis:

The core components of algorithmic analysis include the identification of relevant metrics for measuring efficiency, the utilization of mathematical techniques to quantify these metrics, and the comparison of algorithms under different input conditions. Asymptotic analysis, characterized by big O notation, provides a standardized way to describe algorithm performance as input size approaches infinity. This notation abstracts away constant factors and lower-order terms, focusing on the fundamental growth rate of algorithms.

DISCUSSION

The Analysis of Algorithms is the gateway to understanding the inner workings of algorithms and their impact on computational systems. Time complexity, as a key dimension of analysis, offers insights into an algorithm's efficiency as inputs grow. It answers crucial questions: How does an algorithm perform on larger datasets? Does its efficiency degrade gracefully or drastically? Time complexity classes, such as O(1), O(log n), O(n), O(n log n), O(n^2), and beyond, encapsulate different growth rates, allowing us to classify algorithms and anticipate their behaviors.

Space complexity, the counterpart to time complexity, delves into an algorithm's memory usage. Efficient memory utilization is critical, especially in resource-constrained environments. Algorithms with lower space complexity are favored, as they minimize the memory footprint and enable systems to operate smoothly. Balancing time and space complexity is a hallmark of adept algorithm design, ensuring that algorithms deliver performance without excessive memory usage.

Asymptotic analysis, characterized by big O notation, is a cornerstone of algorithmic analysis. It abstracts away irrelevant details, focusing on the dominant growth rate of an algorithm's performance. This abstraction provides a clear and concise language to communicate the scalability of algorithms. For instance, an algorithm with O(n log n) time complexity signifies a more favorable growth rate compared to an algorithm with O(n^2) time complexity.

The real-world implications of algorithmic analysis extend to every facet of computation. In the realm of databases, efficient search algorithms, such as binary search, optimize data retrieval. In web development, algorithms with optimal time and space complexity ensure responsive user experiences. In scientific simulations, efficient algorithms enable researchers to model complex phenomena in feasible timeframes. The healthcare sector leverages algorithms to analyze medical data, enhancing diagnostics and treatment planning. Algorithmic analysis drives the architecture of operating systems, networking protocols, and artificial intelligence systems, ensuring their reliability and efficiency.

Unveiling the Efficiency of Algorithms through Analysis:

In the vast realm of computer science, the Analysis of Algorithms stands as a fundamental discipline that offers a window into the performance and efficiency of algorithms under various conditions. As algorithms serve as the backbone of modern computing, understanding their efficiency becomes paramount in designing systems that are responsive, scalable, and optimized. This chapter delves into the essence of algorithmic analysis, exploring its significance, methodologies, mathematical foundations, and real-world implications.

The Core of Algorithmic Analysis: Unraveling Efficiency

Algorithmic analysis revolves around the core question of how efficiently an algorithm performs as the size of the input grows. This efficiency is paramount in determining an algorithm's suitability for real-world scenarios, as well as its impact on resource utilization and system behavior. By dissecting the performance of algorithms, we gain insights into how they scale and operate under different conditions.

At the heart of algorithmic analysis lie two fundamental dimensions: time complexity and space complexity. Time complexity measures the amount of time an algorithm takes to complete its task, usually as a function of the input size. Space complexity, on the other hand, quantifies the amount of memory space an algorithm requires to solve a problem. These dimensions provide a comprehensive view of an algorithm's behavior and its impact on the underlying system [4], [5].

Deciphering Time Complexity: Understanding Performance Scaling

Time complexity, often expressed using big O notation, is a cornerstone of algorithmic analysis. It offers a standardized way to describe how an algorithm's execution time scales as the input size increases. The big O notation abstracts away constant factors and lower-order terms, focusing on the dominant growth rate of the algorithm. For instance, an algorithm with O(n) time complexity indicates linear scaling, while O(n^2) indicates quadratic scaling.

Time complexity classes, such as O(1), O(log n), O(n log n), O(n^2), and beyond, categorize algorithms based on their performance characteristics. These classes provide a framework for comparing algorithms and making informed design decisions. Algorithms with lower time complexity are preferred, as they exhibit better scaling behavior and are more suitable for larger datasets.

Balancing Resources: Exploring Space Complexity

Space complexity complements time complexity by evaluating an algorithm's memory usage. Efficient memory utilization is vital, especially in resource-constrained environments. Algorithms with lower space complexity are preferable, as they minimize the memory footprint and enable systems to operate efficiently. Balancing time and space complexity is a hallmark of effective algorithm design, ensuring that algorithms deliver performance without excessive memory consumption.

Space complexity classes, such as O(1), O(n), O(n^2), reflect an algorithm's memory requirements as input size varies. Understanding an algorithm's space complexity is crucial for system architects, as it influences memory allocation, data structures, and overall system behavior.

Big O Notation: The Language of Asymptotic Analysis

Asymptotic analysis, embodied by big O notation, forms the language through which algorithmic analysis is communicated. It transcends language and implementation details, providing a clear and concise means to convey algorithm performance. By abstracting away constant factors and lower-order terms, big O notation focuses on the fundamental growth rate of algorithms, making it an essential tool for comparing and contrasting algorithmic behaviors. For instance, an algorithm with O(n log n) time complexity indicates better scalability than an algorithm with O(n^2) time complexity. This abstraction empowers designers to make informed decisions about algorithm selection and optimization, regardless of specific programming languages or hardware configurations [6], [7].

Real-World Applications: From Software Development to Scientific Simulations

The implications of algorithmic analysis extend far beyond theoretical constructs. In software development, the choice of algorithms impacts application responsiveness and resource utilization. Efficient search and sorting algorithms optimize data retrieval and management. In web development, algorithms with optimal time and space complexity ensure seamless user experiences. In scientific simulations, efficient algorithms enable researchers to model complex phenomena within reasonable timeframes. In healthcare, algorithms analyze medical data, facilitating diagnostics and treatment planning. System architects leverage algorithmic analysis to identify performance bottlenecks and optimize system behavior. The architecture of operating systems, networking protocols, and artificial intelligence systems hinges on efficient algorithmic design.

Navigating Efficiency in the Digital Era:

In the dynamic landscape of computing, the Analysis of Algorithms provides a roadmap to efficiency and scalability. Time complexity, space complexity, and big O notation serve as the compasses that guide us through this intricate terrain. As technology advances and computational challenges grow in complexity, the significance of algorithmic analysis becomes increasingly pronounced. Mastering the art of algorithmic analysis empowers individuals and industries to harness the full potential of algorithms. It transforms theoretical understanding into practical applications, ensuring that our algorithms meet the demands of the digital era. By quantifying performance, making informed design decisions, and optimizing resource utilization, algorithmic analysis drives the creation of systems that are not only innovative but also responsive and efficient [8]–[10]. In the ever-evolving world of computer science, the Analysis of Algorithms remains a critical discipline that bridges theory and practice. It empowers us to decode the efficiency of algorithms, making us architects of systems that seamlessly navigate the challenges of the digital age. With the insights gained from algorithmic analysis, we stand at the forefront of innovation, ready to propel computing to new heights of efficiency and excellence.

CONCLUSION

Decoding Efficiency through Algorithmic Analysis:

In the ever-evolving landscape of computer science, the Analysis of Algorithms holds the key to understanding the efficiency, scalability, and impact of algorithms. Time complexity, space complexity, and big O notation are the tools that unlock insights into algorithm performance, enabling us to make informed decisions about algorithm selection, design optimization, and

system behavior prediction. As technology advances and computational challenges grow in complexity, the significance of algorithmic analysis becomes even more pronounced. By mastering the art of algorithmic analysis, individuals and industries alike navigate the intricacies of efficient problem-solving. Algorithm analysis provides a lens through which we can dissect algorithms, revealing their underlying mechanics and guiding us to make strategic design choices. The integration of algorithmic analysis into the design process is not just a technical endeavor; it's a commitment to building systems that deliver optimal performance, scalability, and reliability.As we delve deeper into the complexities of modern technology, the Analysis of Algorithms continues to be a compass, pointing us towards solutions that are not only innovative but also efficient. It empowers us to bridge the gap between theoretical understanding and practical application, ensuring that our algorithms stand the test of time and meet the demands of a dynamic computational world.

REFERENCES

[1] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. Knowl. Data Eng.*, 2007, doi: 10.1109/TKDE.2007.250581.

[2] N. Kock, "Using WarpPLS in e-collaboration studies: Mediating effects, control and second order variables, and algorithm choices," *Int. J. e-Collaboration*, 2011, doi: 10.4018/jec.2011070101.

[3] N. Yang, R. Wang, X. Gao, and D. Cremers, "Challenges in monocular visual odometry: Photometric calibration, motion bias, and rolling shutter effect," *IEEE Robot. Autom. Lett.*, 2018, doi: 10.1109/LRA.2018.2846813.

[4] J. R. Gage and T. F. Novacheck, "An update on the treatment of gait problems in cerebral palsy," *Journal of Pediatric Orthopaedics Part B*. 2001. doi: 10.1097/01202412-200110000-00001.

[5] E. E. Smith, S. J. Reznik, J. L. Stewart, and J. J. B. Allen, "Assessing and conceptualizing frontal EEG asymmetry: An updated primer on recording, processing, analyzing, and interpreting frontal alpha asymmetry," *Int. J. Psychophysiol.*, 2017, doi: 10.1016/j.ijpsycho.2016.11.005.

[6] S. Das Choudhury, A. Samal, and T. Awada, "Leveraging image analysis for high-throughput plant phenotyping," *Frontiers in Plant Science*. 2019. doi: 10.3389/fpls.2019.00508.

[7] A. Rai, M. Yamazaki, and K. Saito, "A new era in plant functional genomics," *Current Opinion in Systems Biology*. 2019. doi: 10.1016/j.coisb.2019.03.005.

[8] M. Lakshmanan, G. Koh, B. K. S. Chung, and D. Y. Lee, "Software applications for flux balance analysis," *Brief. Bioinform.*, 2014, doi: 10.1093/bib/bbs069.

[9] A. Singh and S. Mahajan, "Data mining algorithms on prediction of cardiovascular diseases," *Int. J. Recent Technol. Eng.*, 2019, doi: 10.35940/ijrte.C6887.098319.

[10] G. Scalet and F. Auricchio, "Computational Methods for Elastoplasticity: An Overview of Conventional and Less-Conventional Approaches," *Arch. Comput. Methods Eng.*, 2018, doi: 10.1007/s11831-016-9208-x.

CHAPTER 3

INTRODUCTION TO LINKED LIST

Girija Shankar Sahoo, Assistant Professor
Maharishi School of Engineering & Technology,Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-gssahoo07@gmail.com

ABSTRACT:

From a high level perspective, linked lists may be considered to be a collection of nodes. Each node has at least one pointer to the next node and, in the event of the last node, a null pointer signifying that the linked list is complete. Our linked list implementations in DSA keep head and tail pointers at all times, making insertion at either the head or tail of the list a constant-time operation. The insertion procedure is the one that sticks out among the three. A typical insertion to the front or rear of the linked list is an O(1) operation because in DSA we decided to always preserve pointers (or more accurately references) to the node(s) at the head and tail of the linked list. Making an insertion before a node in a singly linked list that is neither the head nor the tail is an exception to this rule. The complexity is O(n) when the previous node we are adding is dispersed across the linked list (a process known as random insertion). We must navigate the linked list to find that node's current predecessor in order to add before the specified node. This traversal results in a run time of O(n).

KEYWORDS:

Doubly Linked List, Linked List, Node Connections, Operations, Singly Linked List.

INTRODUCTION

This does not apply to random insertion, which will be a linear process. Consequently, linked lists in DSA possess the following attributes:

1. Insertion is O(1)
2. Removal is O(n)
3. Searching is O(n)

Although this data structure is simple, linked lists contain a few distinctive features that might occasionally make them quite alluring: The list is dynamically resized, thus there is no copy cost like there would be for an array or vector in the long run. Additionally, insertion is O(1).

Singly Linked List:

One of the most basic data structures in this book is the singly linked list. A single linked list's nodes each contain a value and a reference to the next node, if any, in the list. In Figure 1 shown the Singly linked list process. Figure 1 shown Simply Linked List.

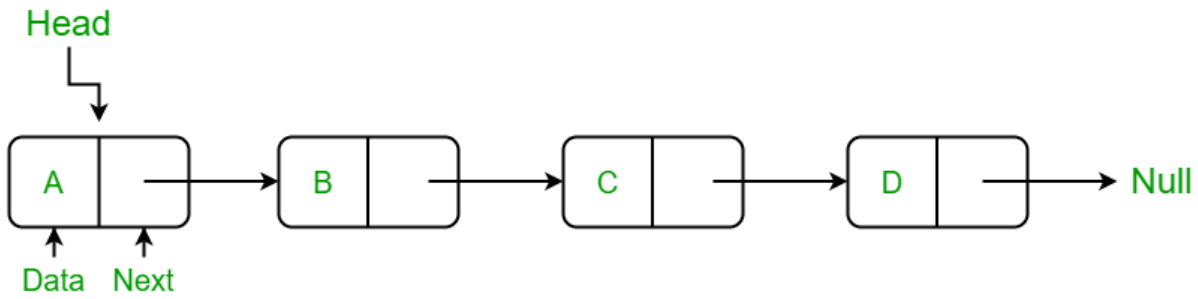


Figure 1: Singly linked list [geeksforgeeks.org].

Insertion:

It is quite easy to add a new element at the beginning of a single linked list. Only a few changes need to be made to the node connections. The actions listed below must be carried out in order to add a new node to the list at the start. Set aside space for the new node, then add data to its data section. The following statements will achieve this. Include a link in the new node linking to the list's original initial node. The next sentence will be used to accomplish this [1]–[3]. Finally, we need to set the new node as the list's first node. To accomplish this, use the following line.

Algorithm

1. STEP 1: IF PTR = NULL
2. SIGNIFY OVERFLOW
3. STEP 2: SET NEW_NODE = PTR
4. STEP 3: SET PTR = PTR NEXT
5. STEP 4: SET NEW_NODE DATA = VAL
6. STEP 5: SET NEW_NODE NEXT = HEAD
7. STEP 6: SET HEAD = NEW_NODE
8. STEP 7: GO OUT

Removal:

Although removing a node from a linked list is simple, there are a few situations that require attention: If any of the following conditions apply: the list is empty; the node to be removed is the sole node in the linked list; we are deleting the head node; we are removing the tail node; we are removing the node that is between the head and tail; or the item to be removed doesn't exist in the linked list. No matter where a node is in a list or whether it is the head or not, the method whose situations we have discussed will eliminate it. You can write far more succinct algorithms if you know that items will only ever be removed from the head or tail of the list. When the linked list is always removed from the front, deletion becomes an O(1) operation.

Algorithm

1. Step 1: IF HEAD = NULL.
2. Step 2: SET PTR = HEAD.
3. Step 3: Repeat Steps 4 and 5 while PTR -> NEXT!= NULL.
4. Step 4: SET PREPTR = PTR.

5. Step 5: SET PTR = PTR -> NEXT.
6. Step 6: SET PREPTR -> NEXT = NULL.
7. Step 7: FREE PTR.
8. Step 8: EXIT.

Searching:

An easy way to search a linked list is to simply go over each node, comparing the value we're looking for with each one's value. This section's algorithm and the traversal algorithm are quite similar [4], [5].

Algorithm

1. STEP 1: set PTR to HEAD.
2. STEP 2: Set I = 0.
3. STEP 3: If PTR is NULL
4. SCRABBLE "EMPTY LIST"
5. STEP 8 UP
6. FINAL IF
7. Repeat steps 5 through 7 until
8. STEP 4: PTR!= NULL
9. STEP 5: IF PTR DATA = ITEM type i+1
10. STEP 6: I = I + 1 concludes
11. STEP 7: PTR = PTR NEXT
12. [FINAL LOOP]
13. STEP 8: Exit.

Reversing the order of the list:

Traversing a singly linked list forward (i.e., from left to right) is straightforward. What would happen, though, if we needed to go through the linked list's nodes in reverse order for any reason? The procedure to carry out such a traversal is quite straight forward we will need to get a reference to a node's predecessor even though the fundamental properties of a singly linked list's nodes make this a costly operation. Finding a node's predecessor is an O(n) operation for each node, making the cost of traversing the whole list backwards O(n2).

The following procedure is applied to a linked list with the integers 5, 10, 1, and 40.

Algorithm

1. STEP 1: Set up three points.
2. STEP 2: curr as the head,
3. STEP 3: prev as NULL,
4. and next as NULL.
5. STEP 4: Go through the linked list iteratively. Perform the following in a loop:
6. STEP 5: Store the next node next = curr -> next prior to updating the next of curr.
7. STEP 6: Update the curr's next pointer to the prev now.
8. STEP 7: curr = next = previous
9. STEP 8: Update prev as next prev = curr and curr as prior
10. STEP 9: next = curr

DISCUSSION

Doubly Linked List:

Doubly linked lists and singly linked lists are quite similar. Each node contains a reference to both the next and previous nodes in the list, which is the sole difference**.**Figure 1 shown Doubly Linked List.

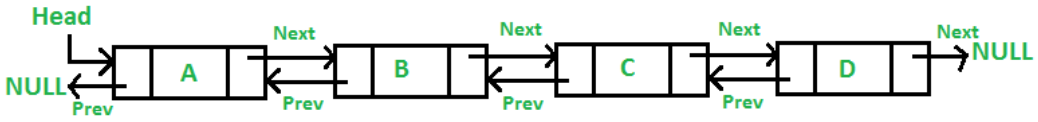


Figure 1: Doubly linked list [geeksforgeeks.org].

Insertion:

Due to the fact that each node in a doubly linked list has two pointers, maintaining a doubly linked list requires more pointers than maintaining a single linked list. Any element can be inserted into a doubly linked list in one of two ways. The list is either empty or has at least one element. To place a node at the start of a doubly linked list, follow these steps.

Algorithm

1. STEP 1: Write OVERFLOW IF ptr = NULL.
2. Enter Step 9 [END OF IF] now.
3. STEP 2: SET NEW_NODE = ptr.
4. STEP 3: SET ptr = ptr -> NEXT
5. STEP 4: SET NEW_NODE -> DATA = VAL
6. STEP 5: SET NEW_NODE -> BEFORE = NULL
7. STEP 6: NEXT = START SET NEW_NODE
8. STEP 7: Set the head to NEW_NODE = PREV
9. STEP 8: SET head = NEW_NODE
10. STEP 9: EXIT.

Searching:

To find a given element in the list, we only need to traverse the list. To search for a certain operation, carry out the following actions [6]–[8].

Algorithm

1. STEP 1: [END OF IF] IF HEAD == NULL
2. WRITE "UNDERFLOW" GOTO STEP 8
3. STEP 2: Set PTR = HEAD.
4. STEP 3. Set i = 0.
5. Repeat steps 5 through 7
6. STEP 4: while PTR!= NULL .
7. STEP 5: [END OF IF] IF PTR data = item return i

8. STEP 6: i = i + 1.
9. STEP 7: PTR = PTR following
10. STEP 8: Leave.

Removal:

As you might have suspected, the situations in which we delete items from a doubly linked list are the same as those described in Similar to insertion, we also need to bind a second reference (Previous) to the right value [9], [10]**.**

Algorithm

1. STEP 1: IF HEAD = NULL,
2. WRITE UNDERFLOW, THEN GO TO STEP 6.
3. STEP 2: SET PTR = HEAD.
4. STEP 3: SET HEAD = HEAD NEXT .
5. STEP 4: SET HEAD PREV = NULL
6. STEP 5: Free PTR
7. STEP 6: QUIT

CONCLUSION

When you need to store an undetermined number of things, linked lists are a smart choice. When using a data structure like an array, you must define the size in advance; if the size is exceeded, a resizing mechanism with a linear run time must be used. In order to ensure a consistent run time, linked lists should also be used when nodes will only be removed from the head or tail of the list.Maintaining pointers to the nodes at the head and tail of the list is necessary, but if you repeat this action frequently, the memory expense will pay for itself. The following are some things that linked lists are not particularly good at: searching, random insertion, and index access to nodes. You could keep a count variable that counts the number of items in the list at the cost of a small amount of memory (in most cases 4 bytes would suffice) and a few more read/write operations so that accessing such a basic property is a constant operation—you just need to update count during the insertion and deletion algorithms.

When doing simple queries, singly linked lists should be utilized. In general, doubly linked lists are more tolerant of actions on a linked list that are not trivial. When you need both forwards and backwards traversal, we advise using a doubly linked list. The majority of the time, this need is present. Take a token stream as an example, which you wish to parse using recursive descent. You may need to go back and forth at times to build the proper parse tree. A doubly linked list is ideal in this situation since it makes bi-directional traversal considerably simpler and faster than a single linked list.

REFERENCES:

[1] A. Zenetos, G. Apostolopoulos, and F. Crocetta, “Aquaria kept marine fish species possibly released in the Mediterranean Sea: First confirmation of intentional release in the wild,” *Acta Ichthyol. Piscat.*, 2016, doi: 10.3750/AIP2016.46.3.10.

[2] D. Hart *et al.*, “Integration of entrustable professional activities with the milestones for emergency medicine residents,” *West. J. Emerg. Med.*, 2019, doi: 10.5811/westjem.2018.11.38912.

[3] "Mastering algorithms with C," *Comput. Math. with Appl.*, 1999, doi: 10.1016/s0898-1221(99)91275-0.

[4] C. Unger, A. Freitas, and P. Cimiano, "An introduction to question answering over linked data," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 2014, doi: 10.1007/978-3-319-10587-1_2.

[5] L. M. Ochoa-Ochoa, C. A. Ríos-Muñoz, S. B. Johnson, O. A. Flores-Villela, J. Arroyo-Cabrales, and M. Martínez-Gordillo, "Invasive species: Legislation and species list considerations from Mexico," *Environmental Science and Policy*. 2019. doi: 10.1016/j.envsci.2019.03.002.

[6] F. Casbas Pinto, S. Ravipati, D. A. Barrett, and T. C. Hodgman, "A methodology for elucidating regulatory mechanisms leading to changes in lipid profiles," *Metabolomics*, 2017, doi: 10.1007/s11306-017-1214-y.

[7] J. M. Rude *et al.*, "Rapid response to meningococcal disease cluster in foya district, lofa county, liberia january to February 2018," *Pan Afr. Med. J.*, 2019, doi: 10.11604/PAMJ.SUPP.2019.33.2.17095.

[8] S. C. Sacleux and D. Samuel, "A Critical Review of MELD as a Reliable Tool for Transplant Prioritization," *Seminars in Liver Disease*. 2019. doi: 10.1055/s-0039-1688750.

[9] J. Lomax, "Get ready to GO! A biologist's guide to the Gene Ontology," *Brief. Bioinform.*, 2005, doi: 10.1093/bib/6.3.298.

[10] B. M. Arneth, "Neuronal Antibodies and Associated Syndromes," *Autoimmune Diseases*. 2019. doi: 10.1155/2019/2135423.

CHAPTER 4

A BRIEF STUDY ONBINARY SEARCH TREE

Ritesh Kumar, Assistant Professor
Maharishi School of Engineering & Technology,Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-riteshkumar268@gmail.com

ABSTRACT:

The concept of binary search trees (BSTs) is relatively straightforward. The left subtree of the root node, which has the value x, includes nodes with values x, and the right subtree contains nodes with values equal to x. Regarding nodes in their left and right subtrees, each node adheres to the same set of rules.Because its operations are advantageously quick insertion, lookup, and deletion may all be completed in O(log n) time BSTs are of interest. It is significant to note that the BST must be fairly balanced in order to achieve the O(log n) timings for these operations (for a tree data structure with self-balancing characteristics, see the AVL tree discussed in 7).In the examples that follow, you may assume that root refers to the tree's root node unless it is used as a parameter alias**.**

KEYWORDS:

Balanced Trees, Binary Search Tree, Insertion, Node Structure, Traversal.

INTRODUCTION

Insertion:

There is a valid reason why the insertion algorithm is divided. The first algorithm (non-recursive) verifies if the tree is empty, which is a fundamental basic case. Then we just make our root node and finish if the tree is empty. In all other circumstances, we use the recursive InsertNode technique, which merely directs us to the first suitable location to insert value in the tree.

A binary chop is performed at each stage, when we decide whether to recurse into the left or right subtree by comparing the new value to the node's existing value. No value can concurrently meet the requirements to insert it in both subtrees for any entirely ordered type [1]–[3].

Algorithm

1. STEP 1: IF TREE = NULL
2. Provide TREE with memory
3. Set tree to data equals item, tree to left equals tree to right equals null.
4. IF ITEM IS A TREE, THEN DATA
5. If not, insert (TREE -> LEFT, ITEM)
6. Embedding(TREE->RIGHT, ITEM)
7. [FINAL IF]
8. [FINAL IF]
9. STEP 2: Finish.

Searching:

Even more basic than insertion is searching a BST. Although the pseudocode is self-explanatory, we will nonetheless examine the algorithm's basic idea.

In earlier discussions, we discussed how when adding a node, we go either left or right, with the right subtree holding values equal to x, where x is the value of the node that has to be inserted.

The rules are slightly more atomicized while searching, and there are always four scenarios to take into account:

The value is not in the BST if: 1. the root =; or 2. root.Value root; Value = value, in which case value is in the BST.Value, we must look for it in the left subtree of root; otherwise, value must be greater than root.Value must be examined in the appropriate subtree of the root.

Algorithm:

1. STEP 1: If Root is equal to Data or Root is NULL.
2. Reply ROOT
3. IF ROOT ROOT -> DATA, ELSE
4. Return search (ROOT -> RIGHT, ITEM) IF NOT Return search (ROOT -> LEFT, ITEM)
5. [FINAL IF]
6. [FINAL IF]
7. STEP 2: Finish

Deletion:

There are four circumstances to take into consideration when removing a node from a BST, and they are as follows:

1. If a value has a left subtree but no right subtree, a left subtree but no right subtree, a value with both left and right subtrees but no leaf nodes, or a value with both left and right subtrees but no leaf nodes, we promote the largest value in the left subtree.
2. There is also an implicit fifth situation in which the single node in the tree that has to be deleted. Although the first already addresses this situation, it should still be considered a possibility.
3. Of course, a value can appear more than once in a BST. In this scenario, the first instance of that value in the BST will be eliminated.

Algorithm

1. STEP 1: IF TREE = NULL
2. If an item is not discovered in the tree, type "item not found in the tree" ELSE
3. If the item is more than the tree's data, delete (TREE->LEFT, ITEM).
4. (TREE -> RIGHT, ITEM) delete
5. IF NOT A TREE TO THE LEFT AND A TREE TO THE RIGHT
6. SET TREE -> DATA = TEMP -> DATA SET TEMP = findLargestNode(TREE -> LEFT)
7. Delete(TREE -> LEFT, TEMP -> DATA) IF TREE -> LEFT AND TREE -> RIGHT ARE BOTH NULL THEN SET TEMP = TREE
8. Set tree to zero.

9. ELSE SET TREE = TREE -> LEFT ELSE SET TREE = TREE -> RIGHT [END OF IF] IF TREE -> LEFT!= NULL
10. IF ENDS, FREE TEMP
11. STEP 2: Finish.

DISCUSSION

Identifying a particular node's parent

Simple reference (or pointer) to the parent node of the node with the specified value is the sole goal of this approach. We have discovered that such a method is really helpful, particularly when carrying out thorough tree transformations.

Algorithm

1. STEP 1: The FindParent(value, root)
2. STEP 2: Pre: value is the value of the node whose parent we're seeking to find.
3. STEP 3: Root is the root node of the BST and is! =
4. STEP 4: Post: a reference to the parent node of value if found; otherwise,
5. STEP 5: if value = root.Value
6. returns
7. STEP 6: tend if value equals root.Value
8. STEP 7: root.Left = return if root.Left,
9. STEP 8: else.Value = Value return the root
10. STEP 9: if not, then return
11. STEP 10: If root is true, FindParent(value, root.Left)
12. STEP 11: ends.Right = return
13. STEP 12: if root.Right otherwise.Value = Value return the root
14. STEP 13: if not, then return If 24) end if 25)
15. STEP 14: end if FindParent(value, root.Right) end FindParent

When the specified value does not exist in the BST, the aforementioned procedure has a specific case in which case we return. Unless they are already positive that a node with the specified value exists, callers to this algorithm must consider this option.

Acquiring a node reference

This procedure is except it returns a reference to the node itself rather than the parent of the node with the specified value. If the value can't be retrieved, is once more returned.

Algorithm

1. STEP 1: The FindNode(root, value) algorithm
2. STEP 2: Pre: value is the value of the node whose parent we're seeking to find.
3. STEP 3: Root is the BST's root node.
4. STEP 4: If a node of value is found, post: a reference to it; else,
5. STEP 5: if root = return end if root.Value is value.
6. return
7. STEP 6: root 10
8. STEP 7: if value is not root,
9. STEP 8: else.Value

10. STEP 9: Return FindNode(root.Left, value)
11. Otherwise
12. STEP 10: Return FindNode(root.Right, value)
13. STEP 11: terminate
14. STEP 12: if FindNode

The only difference between the FindNode method and the Contains algorithm (dened in 3.2) is that we are returning a reference to a node that is neither true nor false. Astute readers will have spotted this. The simplest approach to implement Contains with FindNode is to call FindNode and compare the return value with [4], [5].

Finding the binary search tree's lowest and greatest values

You can find the lowest value in a BST by simply traversing the nodes in its left subtree, always turning left when you come across a node, and stopping when you locate a node without a left subtree. Finding the greatest value in the BST has the opposite effect. The only reason these two algorithms are presented together is for completeness.We have reached the final node if the Left (FindMin) or Right (FindMax) node references are true, which is the fundamental case for both FindMin and FindMax algorithms.

Algorithm

1. STEP 1: The FindMin(root) algorithm
2. STEP 2: Before: root is the BST's root node
3. STEP 3: root= Post:
4. STEP 4: If root.Left = return root, the lowest value in the BST is found.End Value
5. STEP 5: if FindMin(root.Left) == 9)
6. STEP 1: The FindMax(root) algorithm
7. STEP 2: Before: root is the BST's root node
8. root=
9. STEP 3: Post: If root.Right = return root, the greatest value in the BST is found.End of value
10. STEP 4: if FindMax(root.Right) == 9)
11. STEP 5: FindMax

Tree Crossings

To navigate the objects in a tree, a variety of techniques can be used; the strategy you choose will depend on the node visitation order you need.

We will briefly discuss the traversals that DSA offers for all data structures descended from the BinarySearchTree in this section [6]–[8].

Preorder

The preorder algorithm requires you to first visit the root, then navigate the left subtree, and then navigate the right subtree.

Algorithm

1. STEP 1: Root is the BST's root node,
2. STEP 2: hence Preorder(root) = 2

3. STEP 3: The nodes of the BST have been visited in preorder
4. STEP 4: if root=yield root in steps three through eight.Value
5. STEP 5: if 9) end Preorder(root.Left) Preorder(root.Right)

Postorder

This approach is quite similar to that in except instead of traversing both subtrees before yielding the value of the node, it does so.

Algorithm

1. STEP 1: Postorder(root) 2) Pre: root is the BST's root node.
2. STEP 2: If root=Postorder(root.Left),
3. STEP 3: then all of the nodes in the BST have been visited in postorder.
4. STEP 4: Postorder(root.Right)
5. STEP 5: Returns
6. STEP 6: root.End Value if 9)
7. STEP 7: end Postorder

Inorder

Another variant of the techniques used in -order traversal, where the value of the current node is yielded depending on whether the left or right subtree is traversed.

Algorithm

1. STEP 1: algorithmInorder(root)
2. STEP 2: Before: root is the BST's root node
3. STEP 3: Post: The BST has had nodes visited in sequence
4. STEP 4: If root equals, then
5. STEP 5: Inorder(root.Left).
6. STEP 6: yieldroot.Value
7. STEP 7: Inorder(root.Right)

One of the charms of in order traversing a list of values that are yielded in their comparing order.In other words, they produced a sequence that would have attribute xi xi+1 while traversing a populated BST with the inorderstrategy..

BreadthFirst:

The values of the fall nodes of a certain depth in the tree are yielded by traversing a tree in breadth-first order before any deeper nodes. In other words, given a depth d, we would visit the values of the nodes that lie on the left and right, and we would continue to do so until we had no more nodes to visit. Breadth-first traversal is traditionally done using a list (vector, resizeable array, etc.) to store the value of visited nodes in breadth-first order and a queue to hold nodes that have not yet been visited [9], [10]..

Algorithm

1. STEP 1: BreadthFirst(root) algorithm
2. STEP 2: Before: root is the BST's root node
3. STEP 3: Post: the nodes in the BST have been visited in breadth first order q queue

4. STEP 4: while root=yield root.Value of the root.
5. STEP 5: If root is true, then Left=q.Enqueue(root.Left)
6. STEP 6: end if.End if if Right=q.Enqueue(root.Right)!If 18)
7. STEP 7: end while q.Dequeue() 19)
8. STEP 8: end q.IsEmpty() root otherwise root
9. STEP 9: end BreadthFirst

CONCLUSION

In conclusion, when you need to describe types that are arranged in accordance with certain, type-specific criteria, a binary search tree is an excellent choice. It is incredibly efficient to use logarithmic insertion, lookup, and deletion. There are several methods to go to a tree's nodes, but traversal is always linear. Since trees are recursive data structures, many of the algorithms that use them will likely be recursive as well.

The assumption that the left and right subtrees of the binary search trees are properly balanced is the basis for the run times reported in this chapter.Only when this is true are we able to achieve logarithmic run times for the techniques described before.

Such a condition is not enforced by a binary search tree, and the execution durations for these operations on a pathologically imbalanced tree become linear, thereby making the tree merely a linked list. We will look at an AVL tree that imposes self-balancing qualities later in chapter seven to assist achieve logarithmic run speeds.

REFERENCES:

[1] J. L. Bentley, “Multidimensional Binary Search Trees Used for Associative Searching,” *Commun. ACM*, 1975, doi: 10.1145/361002.361007.

[2] S. V. Nagaraj, “Optimal binary search trees,” *Theor. Comput. Sci.*, 1997, doi: 10.1016/S0304-3975(96)00320-9.

[3] B. Manthey and R. Reischuk, “Smoothed analysis of binary search trees,” *Theor. Comput. Sci.*, 2007, doi: 10.1016/j.tcs.2007.02.035.

[4] R. Aguech, A. Amri, and H. Sulzbach, “On Weighted Depths in Random Binary Search Trees,” *J. Theor. Probab.*, 2018, doi: 10.1007/s10959-017-0773-1.

[5] B. Kizilkaya, M. Caglar, F. Al-Turjman, and E. Ever, “Binary search tree based hierarchical placement algorithm for IoT based smart parking applications,” *Internet of Things (Netherlands)*, 2019, doi: 10.1016/j.iot.2018.12.001.

[6] D. D. Sleator and R. E. Tarjan, “Self-Adjusting Binary Search Trees,” *J. ACM*, 1985, doi: 10.1145/3828.3835.

[7] M. Kuba and A. Panholzer, “The left-right-imbalance of binary search trees,” *Theor. Comput. Sci.*, 2007, doi: 10.1016/j.tcs.2006.10.033.

[8] M. Komorowski and T. Trzciński, “Random Binary Search Trees for approximate nearest neighbour search in binary spaces,” *Appl. Soft Comput. J.*, 2019, doi: 10.1016/j.asoc.2019.03.031.

[9] A. B. A. Hassanat, "Norm-based binary search trees for speeding up KNN big data classification," *Computers*, 2018, doi: 10.3390/computers7040054.

[10] J. Kujala and T. Elomaa, "The cost of offline binary search tree algorithms and the complexity of the request sequence," *Theor. Comput. Sci.*, 2008, doi: 10.1016/j.tcs.2007.12.015.

CHAPTER 5

A BRIEF DISCUSSION ON HEAP

Chetan Chaudhary, Assistant Professor,
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India,
Email Id-ccnitj@gmail.com

ABSTRACT:

A heap can be compared to a straightforward tree data structure, although heaps often use one of two methods:a minimum heap, or a maximum heap. The characteristics of the tree and its values are determined by each method. Each parent node would have a value lower than its children if you choose to use the min heap technique. As an illustration, the node at the tree's root will have the lowest value. For the biggest heap technique, the reverse is true. Unless otherwise noted, a heap in this book is presumed to use the min heap technique. A heap is often implemented as an array rather than a collection of nodes, each of which has references to other nodes, like other tree data structures like the one described in 3 and other tree data structures. Though fundamentally identical, the nodes only have two kids total. Figure 4.1 illustrates the array representation of the tree (12 7(3 2) 6(9)), which is not a heap data structure. The array in Figure 4.1 was created by simply adding values from top to bottom and from left to right. Arrows are shown in Figure 4.2 pointing to the immediate left and right children of each value in the array.

KEYWORDS:

Binary Heap, Heapify, Min Heap, Max Heap, Priority Queue.

INTRODUCTION

This chapter is heavily focused on the idea of expressing a tree as an array since comprehending this chapter requires a grasp of this attribute. A step-by-step procedure for representing a tree data structure as an array.

You can suppose that our array's default capacity is eight

Being upfront about the size of the array to use for the heap makes using only an array frequently ineffective. When it comes to the size of its internal data structures, a program's run-time behavior is frequently unpredictable, thus we must pick a more dynamic data structure that has the following characteristics [1]–[3]:

In situations when we know the maximum amount of storage that is needed, we may define an initial size for the array. Additionally, the data structure has scaling methods that allow the array to grow as needed during runtime. Direct offspring of the nodes in a tree data structure represented as an array

1) Vector,
2) Array lists
3) List

How we would handle adding null references to the heap. These changes depending on the circumstance; in certain circumstances, null values are outright forbidden, while in others, we may treat them as less than any non-null value or even larger than any non-null value.

You must study your needs in order to address this uncertainty. By forbidding null values, we will avoid the problem and maintain clarity.

We require a method to get the index of a parent node and the children of a node because we are utilizing an array. The needed expressions for this for a node at index are denoted as follows:

Insertion

It is straightforward to design an algorithm for heap insertion, but we must make sure that heap order is maintained after each insertion. This is often a post-insertion surgery. It is easy to insert a value into the next available slot in an array; all we need to do is use the next available index as a counter and increment it after each insertion.

The first step of the method is adding our value to the heap; the second is confirming the heap order. If a kid's value is greater than or equal to its parent's value, then we must swap the values of the parent and child in the case of min-heap ordering. For each sub tree holding the value we just put, we must carry out this action**.**

Algorithm

1) STEP 1: The procedure for adding elements to the heap is the same as that for deleting them, as was just said. The purpose is to:
2) STEP 2: To make room for the additional element, first raise the heap size by 1.
3) STEP 3: At the end of the Heap, add the new element.
4) STEP 4: The attributes of Heap for its parents may be distorted by this recently added member. Therefore, heapify this newly added element using a bottom-up method to maintain the Heap's attributes

Removal

Similar to insertion, removing an item requires making sure heap ordering is maintained. Three stages make up the deletion algorithm:

1. Find the value to delete's index
2. place the last value in the heap at the item to be deleted's index point.
3. Examine the heap ordering for each sub tree that once contained the item.

Algorithm

1) STEP 1: Since removing an element from any intermediate location in the heap might be expensive, we can just swap out the element that needs to be destroyed for the final element in the heap and then delete it.
2) STEP 2: The last element should take the place of the root or deleted element.
3) STEP 3: Take the final thing out of the heap.
4) STEP 4: Because of this, the last element is now located at the root node's location. Thus, it might not adhere to the heap characteristic. So, heapify the most recent node to be placed in the root location.

Searching

Simply traversing the elements in the heap in order results in a search operation with an O(n) runtime complexity. The search may be conceptualized as a breadth-first traversal.

The flaw in the prior approach is that we failed to take use of the heap strategy's inherent ability to hold all values, which is the attribute at issue. For instance, if the number 4 wasn't present in the heap, we would need to use up the entire backing heap array before we could say that it wasn't. Taking into account what we know about the heap, we may improve the search algorithm by incorporating logic that takes use of the characteristics offered by a particular heap technique. Although it is not easy to optimize to deterministically say that a value is on the heap, the topic is quite intriguing. Think of a min-heap, for instance, that doesn't have the value 5. Only if 5 > the parent of the current node being examined and the current node being inspected nodes at the current level we are travelling can we conclude that the value is not in the heap. If so, then 5 cannot be in the heap, allowing us to respond without having to go through the rest of the heap. The method will actually revert to checking every node in the heap if this property is not satisfied for any level of nodes that we are investigating. The extra logic within the loop, in our opinion, is justified in order to prevent the costly worst-case run time because the optimization that we show might be fairly frequent [4], [5]. The next method was created specifically for a min-heap. The two comparison operations in the else if condition within the inner while loop should be equipped in order to adapt the algorithm for a max-heap.

Traversal:

The outcome is Starting at the first array index (usually 0 in most languages), you iterate through the array. after which go over each value in the array till you reach the upper bound off of the pile. You'll see that we employ Count in our search algorithm. as this upper bound as opposed to the allocated's physical bound array. Count is used to separate the literal array from the conceptual heap. the heap's implementation [6]–[8]: We only consider the objects in the heap, not the entire array the latter might include a variety of additional data because heap mutation, a heap that has likely undergone several mutations. For the purposes of this illustration, we might further assume that the items in the indexes Figure 4.8 shows that 3 5 truly included references to living objects of type T. The subscript is used to distinguish between different items of T. You have probably inferred that from everything you have read so far Any alternative sequence of traversal over the heap would be of little use. The mass attribute only applies to each node's sub tree, thus traversing a heap in Any other style needs some inventiveness. Usually, heaps are not navigated in any manner other than the one previously advised [9], [10].

CONCLUSION

Priority queues and heap sort are the two uses of heaps that are most frequently employed (see for an example implementation). According to the chosen ordering technique, a heap preserves heap order, as was covered in both the insertion and deletion sections. The terms min-heap, max heap, and Chapter 4: Heap are used to describe these techniques. The former method ensures that a parent node's value is lower than each of its children's values, whereas the latter technique enforces that a parent node's value is higher than each of its children's values.

You should presume that a heap is using the min-heap approach when you see one and are not told what strategy it enforces. You will frequently need to say this explicitly if the heap can be

constrained in another way, for as by using max-heap. When the insertion and deletion algorithms are called, the heap gradually adheres to a strategy. The drawback of such a strategy is that it forces us to use algorithms with logarithmic run-time complexity for each insertion and deletion. Even while it might not appear to be very expensive, maintaining the technique has a cost. At some point, we'll also need to account for the cost of dynamic array extension. This will happen if there are more objects in the heap than there is room for in the backup array. You might want to look at a reasonable initial beginning size for your heap array. This will help reduce the Effects of dynamic array resizing.

REFERENCES

[1] L. W. John, "The Art of Heap Leaching- The Fundamentals," *South African Inst. Min. Metall.*, 2011.

[2] T. Sprod, "'Nobody really knows': The structure and analysis of social constructivist whole class discussions," *Int. J. Sci. Educ.*, 1997, doi: 10.1080/0950069970190805.

[3] M. C. Pinotti and G. Pucci, "Parallel algorithms for priority queue operations," *Theor. Comput. Sci.*, 1995, doi: 10.1016/0304-3975(95)00039-Y.

[4] S. Trencansky, "Final Girls and Terrible Youth: Transgression in 1980s Slasher Horror," *J. Pop. Film Telev.*, 2001, doi: 10.1080/01956050109601010.

[5] A. K. Didwania, F. Cantelaube, and J. D. Goddard, "Static multiplicity of stress states in granular heaps," *Proc. R. Soc. A Math. Phys. Eng. Sci.*, 2000, doi: 10.1098/rspa.2000.0626.

[6] S. Cavagnetto and B. Gahir, "Game Theory - Its Applications to Ethical Decision Making," *Cris - Bull. Cent. Res. Interdiscip. Study*, 2014, doi: 10.2478/cris-2014-0005.

[7] T. Ida and T. Matsuno, "Overview of MC/LISP system," *J. Inf. Process.*, 1990.

[8] R. D. Barnett, "The Nimrud Ivories and the Art of the Phoenicians," *Iraq*, 1935, doi: 10.2307/4241579.

[9] W. Mostowski, "From Explicit to Implicit Dynamic Frames in Concurrent Reasoning for Java," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2020. doi: 10.1007/978-3-030-64354-6_7.

[10] S. Dalley, "The God Salmu and the Winged Disk," *Iraq*, 1986, doi: 10.2307/4200253.

CHAPTER 6

A BRIEF DISCUSSION ON SETS

Akhilendra Pratap Singh, Assistant Professor
Maharishi School of Engineering & Technology,Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-akhilendrasingh.muit@gmail.com

ABSTRACT:

Sets, an integral concept in mathematics and computer science, denote collections of distinct elements without a defined order. Elements either belong to a set or do not. Represented using curly braces, such as {1, 2, 3}, sets exhibit membership through notation: x ∈ A (x is in set A), x ∉ A (x is not in A). Set operations include union (A ∪ B), intersection (A ∩ B), complement (A'), and subset (A ⊆ B), along with the power set (P(A)) encompassing all subsets of A. Sets follow laws like distributive, associative, and commutative. Widely applicable, sets serve as foundational tools in mathematics essential in calculus and number theory as well as in computer science for modeling relationships, databases, and data structures. The cardinality of a set is its distinct element count, expressed as |A|. Sets can be finite or infinite, and they form a versatile framework for analyzing, organizing, and solving problems involving diverse data collections.

KEYWORDS:

Data Structure, Intersection Set Operations, Preordered, Postordered, Union.

INTRODUCTION

An unrestricted collection of values is called a set. Each value in the collection stands out from the others. The problem of repeated values is often avoided by the fact that set implementations check that a value isn't already present in the set before adding it. This section doesn't go into great detail on set theory; instead, it shows how to get the values of sets and how to apply basic operations to them. Given the previously denoted set A, we may state that 4 belongs to A signified by 4 A and that 99 does not belong to A denoted by 99 A. It can be tedious to individually list each member of a set-in order to deny it, and more crucially, a set may include many values. Defining a set and its members in terms of a set of qualities that their values must meet is a more succinct method to do so. For instance, the set A only contains positive integers that are even according to the definition A = xx > 0x% 2=0. The qualities that x must have in order to be in the set A are to the right of x, which is an alias for the current value we are reviewing. In this case, x must be greater than zero and the remainder of the mathematical equation x2 must equal zero. You may infer from the definition of the set A that it can have an infinite number of values and that all of those values will be even integers that are components of the set of natural numbers N, where N = 123.

Finally, in this succinct overview of sets, we'll talk about set intersection and union, two of the numerous operations that may be performed on sets. The following is a denial of the union set: A B intersection is defined as A B = x x A and x B. Graphic representations of set intersection and union. The union of the two sets, AB = 12369, and the intersection of the two sets, AB = 2, are determined by the set definitions A = 123 and B = 629.

Within the framework connected to mainstream languages, both set union and intersection are occasionally available. Due to the fact that these techniques are as extension methods defined in the type System.

Linq. Enumerable2, DSA does not offer implementations of these formulas. Most algorithms are contained in System. Instead than only dealing with sets, enumerable also deals with sequences [1]–[3]. A straightforward traverse of both sets may be used to construct set union, adding each item from the two sets to a new union set.

Algorithm

1) Step 1: (set1, set2) Union
2) Step 2: Pre: set1, and set2 =
3) Step 3: union is a set
4) Step 4: For each item in set1 union,
5) Step 5: a union of sets1 and set2 has been generated.Set2 union:
6) Step 6: Add(item)
7) Step 7: end for each
8) Step 8: for each item. Add(item) end of loop
9) Step 9: Restore the unity
10) Step 10: End Union 11

When m is the number of items in the first set and n is the number of items in the second set, the execution time of our Union method is O(m + n). This runtime is only applicable to sets with O(1) insertions.

Implementing set intersection is also simple. The only significant aspect of our method worth highlighting is that we traverse the set with the fewest elements. This is possible because there are no more objects that are members of both sets, hence there are no more items to add to the intersection set, if we have used up all the items in the smaller of the two sets.

Algorithm

1) Step 1: (set1, set2) Intersection
2) Step 2: Prior to set 1, set 2 equals
3) Step 3: junction, and smaller A set is a set
4) Step 4: Post: If set1.Count > set2.Count smaller Set,
5) Step 5: then an intersection of sets 1 and 2 has been produced
6) Step 6: if set1 set2 for each item is smaller,
7) Step 7; set end intersection of sets set1.Contains(item) and set2.Contains(item).
8) If
9) Step 8: end if (item)
10) Step 9: end of for each
11) Step 10: Return Intersection
12) Step 11: end intersection

Our intersection algorithm runs in O(n) time, where n is the total number of elements in the smaller of the two sets. A linear runtime can only be achieved when working with a set that has O(1) insertion, much like with our Union approach [4], [5].

DISCUSSION

Unordered

In general, sets do not need that its members be ordered explicitly. For instance, because it is not necessary, the members of B = 629 do not comply to any ordering system. DSA does not include implementations of unordered sets, as do the majority of libraries; we just mention it here to clarify the difference between an ordered set and an unordered set. Only insertion for an unordered set will be discussed, along with the benefits of using a hash table as the data structure for implementation.

Insertion:

A hash table can effectively be used as the supporting data structure for an unordered set. As previously indicated, we only add an item to a set if it isn't already there, therefore the underpinning data structure we employ has to be simple to search up and put [6], [7].

In general, a hash map offers the following:

1. One for insertion, O(1)
2. Advancing toward O(1) for a glance up.

The aforementioned is dependent on how effective the hashing algorithm of the hash table is, however the majority of hash tables use very efficient general purpose hashing algorithms, therefore the run time complexities for the hash table in your library of choice should be relatively similar in terms of efficiency.

Ordered:

While each member of an ordered set must undergo a predetermined comparison in order to generate a set whose members are properly arranged, ordered sets are comparable to unordered sets in that their members are distinct. The fundamental underpinning data structure for our ordered set in DSA 0.5 and earlier was a binary search tree (dened in 3). Since version 0.6, an AVL tree has taken the role of the binary search tree, mostly due to AVL's balanced nature. By conducting an in order traverse on the underlying tree data structure, which produces the proper ordered sequence of set members, the ordered set's order is realized [8]–[10]. You should read 7 to understand more about the run-time complexity involved with its operations as an ordered set in DSA is really a wrapper over an AVL tree that further ensures that the tree includes unique elements.

CONCLUSION

The use of sets enables the creation of either ordered or unordered collections of singular items. The right underpinning data structure must be chosen while implementing a set, whether it be ordered or unordered. We need this check to be as rapid as feasible, since we first check to see if the item is already present within the set before adding it. We may rely on the usage of a hash table for unordered sets and use an item's key to check whether it already exists within the set. This check has a nearly constant run time complexity when using a hash table. For this check, ordered sets are slightly more expensive, but the logarithmic growth we see by employing a binary search tree as its underlying data structure is acceptable.

The fact that both sets constructed using the method we describe have favorably quick look-up times is a crucial aspect of both. This run-time complexity for a hash table should be almost constant, much like the check before insert. When checking for the presence of an item, ordered sets as stated in 3 do a binary chop at each stage, resulting in a logarithmic run time. We can utilize sets to make many algorithms easier to understand than they would be without them. We build an algorithm that counts the amount of repeated words in a string with the help of an unordered collection.

REFERENCES:

[1] T. Klikauer, "What Is Managerialism?," *Crit. Sociol.*, 2015, doi: 10.1177/0896920513501351.

[2] R. D. Lankes, J. Silverstein, and S. Nicholson, "Participatory networks: The library as conversation," *Inf. Technol. Libr.*, 2007, doi: 10.6017/ital.v26i4.3267.

[3] K. Ryan and F. Whelan, *Freedom*. 2018. doi: 10.2307/139715.

[4] M. I. Z. Ridzwan, S. Shuib, A. Y. Hassan, A. A. Shokri, and M. N. Mohammad Ibrahim, "Problem of stress shielding and improvement to the hip implant designs: A review," *Journal of Medical Sciences*. 2007. doi: 10.3923/jms.2007.460.467.

[5] A. Romanova and V. Barzdenas, "A review of modern CMOS transimpedance amplifiers for OTDR applications," *Electronics (Switzerland)*. 2019. doi: 10.3390/electronics8101073.

[6] A. F. Kramer, K. I. Erickson, and S. J. Colcombe, "Exercise, cognition, and the aging brain," *Journal of Applied Physiology*. 2006. doi: 10.1152/japplphysiol.00500.2006.

[7] N. Naughton and L. Algar, "Linking commonly used hand therapy outcome measures to individual areas of the International Classification of Functioning: A systematic review," *J. Hand Ther.*, 2019, doi: 10.1016/j.jht.2017.11.039.

[8] J. Mitchell *et al.*, "Effectiveness and cost-effectiveness of a very brief physical activity intervention delivered in NHS Health Checks (VBI Trial): Study protocol for a randomised controlled trial," *Trials*, 2016, doi: 10.1186/s13063-016-1413-2.

[9] X. B. Huang and L. Watson, "Corporate social responsibility research in accounting," *J. Account. Lit.*, 2015, doi: 10.1016/j.acclit.2015.03.001.

[10] N. Siegelman, L. Bogaerts, O. Kronenfeld, and R. Frost, "Redefining 'Learning' in Statistical Learning: What Does an Online Measure Reveal About the Assimilation of Visual Regularities?," *Cogn. Sci.*, 2018, doi: 10.1111/cogs.12556.

CHAPTER 7

A BRIEF STUDY ON QUEUES

Neeraj Das, Assistant Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-dasium3000@gmail.com

ABSTRACT:

The chapter on queues in data structures presents a crucial linear arrangement following the First-In-First-Out (FIFO) principle. It details the enqueueing and dequeueing processes, elucidating the addition of elements at the rear and their removal from the front. The chapter delves into diverse implementations of queues, encompassing arrays and linked lists, and introduces circular queues optimized for space efficiency. Real-world applications, including their role in task scheduling and graph traversal using breadth-first search, underscore the practical significance of queues. A comprehensive grasp of queues not only fosters the development of efficient algorithms but also equips programmers to address a wide array of computational challenges.

KEYWORDS:

Breadth-First Search, Circular Queue, Dequeue, Enqueue, First-In-First-Out (FIFO), Priority Queue, Queue Operations, Task Scheduling.

INTRODUCTION

Queues are a fundamental data structure that may be found in a wide range of software, including kernel mode and user mode programs that are crucial to the system. They adhere to the first in, first out (FIFO) principle, which states that the item that is added to the queue first will be the item that is served first, followed by the item that is added second, and so on. In a conventional queue, you can only access the item at the head of the line; if you add another item, it is moved to the rear of the line. In the past, queues have always used the following three fundamental techniques:

1. Enqueue: to insert something at the end of a line;
2. Dequeue: takes the item at the head of the line and takes it out of the line;
3. Peek: 1 takes the first item from the front of the line without taking it out of the line.

Standard queue:

A queue inherently resembles the one previously discussed in this section. Because queues are so widely used and such a fundamental data structure, almost every major library offers a queue data structure that you may use with your preferred language. This is why DSA doesn't offer a standard queue. This section will go through how to create an efficient queue data structure if necessary.

The fact that we may access the item at the front of the line is a queue's key characteristic. An efficient way to create the queue data structure is to use a singly linked list (widened in 2.1).

Run-time complexity for insertion and deletion in a single linked list is O(1). Because we only ever delete items from the front of queues (using the Dequeue operation), deletion has an O(1) run time complexity. A singly linked list always has a pointer to the item at the top, therefore removing an item requires just returning the value of the old head node and changing the head pointer to go to the new head node. Searching a queue has the same run-time complexity as searching a singly linked list: O(n) [1]–[3].

Priority Queue:

In contrast to a typical line, where things are arranged according to who came first, a priority queue arranges its elements according to which item has the highest priority. The sole difference between a priority queue and a regular queue is that items in a priority queue are arranged according to priority; otherwise, both are identical.

A heap data structure is a reasonable choice for implementing a priority queue. By just returning the item at index 0 in the heap array, we can use a heap to look at the first item in the queue. We may create a priority queue using a heap in which the items with the greatest priority are either those with the smallest value or those with the largest [4], [5].

Double-ended queue:

A double ended queue, in contrast to the queues we have discussed thus far in this chapter, enables you to access the items at both the front and the rear of the line. The term "deque"which we shall use from here on is the common word for a double-ended queue.

Entries are added in order to either the front or rear of a deque; a deque does not prioritize its entries the way a priority queue does. The programmer uses the data structures' accessible interface to signify the former attributes of the deque.

DISCUSSION

Deques offer front- and rear-specific variants of frequently performed queue operations. For example, if you wanted to add an item to the front of the queue rather than the back, you would use a method with a name like EnqueueFront. In terms of behavior, the operations behave just like those in a regular queue, or priority queue. The collection of algorithms that add an item to the rear of the deque may occasionally go by the same names as they do for conventional queues, such as EnqueueBack being referred to as simply Enqueue and so on. Additionally, some frameworks outline precise behaviors that data structures must follow. This is unquestionably true in the.NET framework, since the majority of collections implement an interface that calls for the data structure to offer a common Add function. You may confidently assume that the Add function will just enqueue an item to the deque's back in this situation [6], [7].

A deque is equivalent to a regular queue in terms of the complexity of algorithms running at runtime. That is, adding an item to the rear of a queue takes just one action, and adding an item to the head of the queue takes only one operation as well. A deque is a wrapper data structure that makes use of a doubly linked list or an array. If the programmer could deterministically indicate the maximum number of items the deque will ever hold at any one moment, it would obviously be advantageous to use an array as the supporting data structure since it would force the programmer to be specific about the size of the array up front. Unfortunately, this doesn't always true. As a result, the backing array must bear the cost of calling a resizing method, which is

almost certainly an O(n) operation. The library developer would likewise be lost with such a strategy. In order to examine array reduction strategies, it is possible that after a number of iterations of the resizing process and other modifications on the deque, we will have an array occupying a large amount of memory even though we are only using a tiny portion of it. The given method would likewise be O(n), but its use would be more difficult to predict strategically [8]–[10].

A deque often employs a doubly linked list as its basic data structure to get over all the problems listed above. Although a node with two pointers uses more memory than its equivalent array item, this eliminates the need for pricey resizing procedures as the size of the data structure grows dynamically. Memory reclamation is an opaque process when a language is designed to run on a virtual machine that uses garbage collection because nodes that are no longer referenced become inaccessible and are flagged for collection the next time the garbage collection algorithm is called. When an object's memory may be released using C++ or any other language that allows explicit memory allocation and deal location, that decision will be up to the programmer.

CONCLUSION

We have observed that in regular lineups, those who come first are handled first, or in a first-in, first-out (FIFO) manner. In order to decide which process should be the next to use the CPU for a specific time period, the Windows CPU scheduler employs a different queue for each priority of task. Run times for insertion and deletion are constant in normal queues. It's odd to search a line of people. Usually, the item at the front of the line is the only one you are interested in. In spite of this, searching is frequently exposed on queues, and run times are typically linear.

We've also seen priority lines in this chapter, with those at the front of the line having the greatest priority and those at the rear having the lowest. The run durations for insertion, deletion, and searching are identical to those for a heap in one implementation of a priority queue since the backup store is a heap data structure. Although they are pretty rudimentary, queues are a highly natural data structure that may greatly simplify a variety of issues. For instance, the breadth first search described in Section extensively utilizes queues.

REFERENCES:

[1] M. Cildoz, A. Ibarra, and F. Mallor, “Accumulating priority queues versus pure priority queues for managing patients in emergency departments,” *Oper. Res. Heal. Care*, 2019, doi: 10.1016/j.orhc.2019.100224.

[2] K. B. Aditya, Y. R. K, and Suraya, “Perbandingan Metode Simple Queue Dan Queue Tree Untuk Optimasi Manajemen Bandwidth Menggunakan Mikrotik (Studi Di Asrama Wisma Muslim),” *Jarkom*, 2019.

[3] K. Gao, F. Han, P. Dong, N. Xiong, and R. Du, “Connected vehicle as a mobile sensor for real time queue length at signalized intersections,” *Sensors (Switzerland)*, 2019, doi: 10.3390/s19092059.

[4] J. Wang and Y. P. Zhou, “Impact of queue configuration on service time: Evidence from a supermarket,” *Manage. Sci.*, 2018, doi: 10.1287/mnsc.2017.2781.

[5] R. J. Batt and C. Terwiesch, "Waiting patiently: An empirical study of queue abandonment in an emergency department," *Manage. Sci.*, 2015, doi: 10.1287/mnsc.2014.2058.

[6] S. T. O'Neil, "Implementing persistent O(1) stacks and queues in R," *R J.*, 2015, doi: 10.32614/rj-2015-009.

[7] N. I. Dirja, "Implementasi metode simple queue dan queue tree untuk optimasi manajemen bandwith jaringan komputer di Politeknik Aceh Selatan," *METHOMIKA J. Manaj. Inform. Komputerisasi Akunt.*, 2018.

[8] S. A. Bishop, H. I. Okagbue, P. E. Oguntunde, A. A. Opanuga, and O. A. Odetunmibi, "Survey dataset on analysis of queues in some selected banks in Ogun State, Nigeria," *Data Br.*, 2018, doi: 10.1016/j.dib.2018.05.101.

[9] R. Klimek, "Sensor-enabled context-aware and pro-active queue management systems in intelligent environments," *Sensors (Switzerland)*, 2020, doi: 10.3390/s20205837.

[10] E. N. Weiss and C. Tucker, "Queue management: Elimination, expectation, and enhancement," *Bus. Horiz.*, 2018, doi: 10.1016/j.bushor.2018.05.002.

CHAPTER 8

A BRIEF DISCUSSION ON ARRAY

Kalyan Acharjya, Assistant Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology,
Uttar Pradesh, India
Email Id-kalyan.acharjya@gmail.com

ABSTRACT:

An array is a group of objects kept in adjacent memory regions. The goal is to group objects of the same category for storage. As a result, it is simpler to determine each element's position by simply adding an offset to a base value, or the address in memory where the array's first element is stored. For ease of understanding, imagine an array as a set of stairs with a value placed on each step. Any of your pals may be located here by merely knowing how many steps they have left to go. As a result, it is simpler to determine each element's position by simply adding an offset to a base value, or the address in memory where the array's first element is stored.

KEYWORDS:

Array Operations, Array Manipulation, Indexing, Multi-dimensional Array, One-dimensional Array.

INTRODUCTION

Basic array terminology

The index of an element in an array is used to identify that element. The array index begins at 0. Elements in an array are things that can be retrieved by their index and are kept there. Range Length: The maximum number of elements that may be included in an array determines its length. An array's declaration might specify how it is represented. A declaration involves setting aside space for an array of a specific size. Different languages have different techniques of declaring arrays. Here are a few array declarations for several languages, for better example. A linear data structure called an array is a grouping of related data types. Arrays are kept in areas of memory that are close together. It has a fixed size and is a static data structure. Similar data kinds are combined [1]–[3].

Array

Array data structure applications include:

Data storage and retrieval: Arrays are used to keep and get data in a certain order. A weather station's temperature readings or a group of students' test results, for instance, may both be stored in an array.

Data may be sorted using arrays in either ascending or descending order. Arrays play a significant role in sorting algorithms like bubble sort, merge sort, and quicksort.

Searching: Algorithms like linear search and binary search may be used to search arrays for specific elements.

Matrices: In mathematical operations like matrix multiplication, linear algebra, and image processing, matrices are represented by arrays. The implementation of stacks and queues, which are often used in algorithms and data structures, uses arrays as the underlying data structure.

Graphs: In computer science, arrays may be used to represent graphs. The relationships between the nodes in the graph are represented by the values kept in the array, and each element in the array serves as a node in the graph.

Dynamic programming: In order to solve a larger problem, dynamic programming techniques frequently store the preliminary outcomes of smaller problems in arrays [4], [5].

Applications of Array in Real-Time:

Here are a few examples of arrays being used in real time.

In signal processing, arrays are used to represent a collection of samples that have been gathered over time. This may be used to various systems, including radar systems, image processing, and voice recognition.

Multimedia Applications: Arrays are used to hold the pixel or audio samples in multimedia applications like video and audio processing. For instance, the RGB values of a picture can be stored as an array. Large datasets are represented using arrays in data mining applications. This makes it possible to retrieve and analyze data effectively, which is crucial for real-time applications.

Robotics: In robotics, arrays are used to depict the location and orientation of objects in three-dimensional space. Applications like motion planning and object identification can make use of this.

Systems for real-time monitoring and control: Sensor data and control signals are stored in real-time monitoring and control systems using arrays. Real-time processing and decision-making are made possible, which is crucial for systems like industrial automation and aircraft systems.

Financial Analysis: To store past stock prices and other financial information, arrays are used in financial analysis. This makes it possible to obtain and analyze data effectively, which is crucial for real-time trading systems.

In scientific computing, arrays are used to represent numerical data, including measurements from simulations and experiments. This makes it possible to handle and visualize data effectively, which is crucial for scientific research and experimentation that takes place in real-time.

Applications of arrays in C++/C:

1. In the C++ STL, lists and vectors are implemented using arrays.
2. All sorting algorithms start with arrays as their building blocks.
3. The implementation of other DS, like as a stack, queue, etc., uses arrays.
4. utilized for creating matrices.

Due to the fact that arrays are simpler to manage than pointers, data structures like trees can also occasionally employ the array implementation. An array implementation is used, for instance, in segment trees.

Data structures like a heap, map, and set, which may be created using arrays, require binary search trees and balanced binary trees. Graphs can alternatively be constructed as adjacency matrices, which are arrays.

Java applications for array:

Data collections: Arrays are frequently used to hold data collections of the same kind. An array of integers, for instance, can be used to hold a collection of numerical numbers.

Implementing matrices and tables: Matrices and tables may be implemented using arrays. A matrix of numerical values can be stored, for instance, in a two-dimensional array. Data sorting and searching frequently include the usage of arrays. For instance, the Java Arrays class has functions like sort() and binary Search() to sort and search array members [6]–[8].

Implementing data structures: Several additional data structures, including as stacks, queues, and heaps, employ arrays as their underlying data structure. The elements of the stack, for instance, might be kept in an array-based version of the stack. Arrays are frequently used in image processing to store the pixel values of an image. For instance, the RGB values of a picture can be kept in a two-dimensional array.

Uses for Array in C#

Implementing dynamic programming methods: Arrays are frequently used by dynamic programming algorithms to store interim results. For instance, the values of the Fibonacci series are stored in an array in the well-known Fibonacci series method.

1. Arrays can be used in database programming to store the outcomes of database queries. For instance, the outcomes of a SELECT query can be saved as an array.
2. Arrays are a common tool in parallel programming for dividing duties among several threads. An array can be split up into several pieces, each of which can be handled by a distinct thread.

The Array Data Structure's Benefits

Direct and effective access to any element in the collection is made possible by arrays. An array's elements can be accessed using an O(1) operation, which means that the amount of time needed to do so is constant and independent of the array's size.

Fast data retrieval: Because the data is stored in contiguous memory regions, arrays provide quick data retrieval. This indicates that there is no requirement for intricate data structures or algorithms in order to access the data fast and effectively.

Memory effectiveness: Data storage using arrays is memory-efficient. The size of an array is known at compile time since its elements are stored in contiguous memory regions. As a result, less memory fragmentation results from the ability to allocate memory for the full array in a single block.

Versatility: Arrays may contain a wide variety of data types, including as characters, floating-point numbers, integers, and even complicated data structures like objects and pointers.

Simple to use: Arrays are a great choice for novices learning computer programming since they are simple to use and comprehend.

Hardware compatibility: The array data structure is a flexible tool for programming in a variety of situations since it works with the majority of hardware designs.

Array Data Structure Drawbacks:

Fixed size: At the moment of construction, an array's fixed size is chosen. This implies that if the array's size has to be extended, a new array will need to be made, and the data will need to be transferred from the old array to the new array, which may take a lot of time and memory.

Problems with memory allocation Large array allocation can be challenging, especially on systems with low memory. The system could run out of memory if the array size is too high, which could result in a software crash.

Problems with insertion and deletion: Adding or removing an element from an array can be time-consuming and inefficient since every element following the insertion or deletion point needs to be moved to account for the change.

Wasted space: The memory set aside for an array may contain wasted space if it is not entirely occupied. If you have poor recall, this can be a problem.

Limited data type support: Because an array's components must all be of the same data type, it has limited support for complicated data types like objects and structures[9], [10].

Lack of flexibility: Compared to alternative data structures like linked lists and trees, arrays might be less flexible due to their fixed size and restricted support for complicated data types.

Structure has several benefits over arrays

1. Unlike an array, which can only contain comparable data types, the structure may store diverse sorts of data.
2. The size of a structure is not constrained like an array.
3. Array elements are kept in contiguous places, whereas structure components may or may not be stored in contiguous areas.
4. In contrast to arrays, where objects cannot be created, structures allow for the creation of objects.

DISCUSSION

Search Operation: A linear traverse from the first element to the final element may be used to conduct a search operation on an unsorted array.

Add the following after:

Because we don't have to worry about the element's placement, insert operations in unsorted arrays are quicker than those in sorted arrays.

Place in any location:

By moving components to the right that are on the right side of the desired position, it is possible to conduct an insert operation in an array at any position.

Delete process: This process involves finding the element to be removed using a linear search, performing the delete, and then relocating the elements.

Search in a Sorted Array: Binary search may be used to do a search operation within a sorted array.

Fill in a Sorted Array with:

In a sorted array, a binary search is used to find the element's potential location, after which an insert operation is carried out, followed by the shifting of the elements. Because we don't have to worry about the element's placement, inserting data into an unsorted array is quicker than doing it into a sorted array.

In a Sorted Array, delete:

The delete operation involves finding the element to be destroyed using binary search, performing the delete operation, and then moving the elements.

CONCLUSION

One of the most used data structures in C programming is the array. It is an easy and quick approach to save several values under one name. In this article, we'll examine a variety of array-related topics in the C programming language, including array declaration, definition, initialization, kinds of arrays, array syntax, benefits and drawbacks, and more. The array must be declared in C before use, just like any other variable. An array can be declared by stating its name, the type of its components, and the dimensions of the array. When we declare an array in C, the compiler gives the array name a memory block with the given size. In C, initialization refers to the process of giving a variable a starting value. The array's elements include some junk value when it is declared or allocated memory. Therefore, we must set the array's initial value to something useful. In C, there are several different ways to initialize an array.

REFERENCES:

[1] B. Wilson and A. Chakraborty, “Planning Smart(er) Cities: The Promise of Civic Technology,” *J. Urban Technol.*, 2019, doi: 10.1080/10630732.2019.1631097.

[2] A. M. McQuillan, L. Byrd-Leotis, J. Heimburg-Molinaro, and R. D. Cummings, “Natural and Synthetic Sialylated Glycan Microarrays and Their Applications,” *Frontiers in Molecular Biosciences*. 2019. doi: 10.3389/fmolb.2019.00088.

[3] D. A. Dias, S. Urban, and U. Roessner, “A Historical overview of natural products in drug discovery,” *Metabolites*. 2012. doi: 10.3390/metabo2020303.

[4] W. R. Tobler, “Cellular geography.,” *Philos. Geogr.*, 1979, doi: 10.1007/978-94-009-9394-5_18.

[5] Y. Y. Broza and H. Haick, “Nanomaterial-based sensors for detection of disease by volatile organic compounds,” *Nanomedicine*. 2013. doi: 10.2217/nnm.13.64.

[6] P. Song, G. Yang, T. Lang, and K. T. Yong, "Nanogenerators for wearable bioelectronics and biodevices," *Journal of Physics D: Applied Physics*. 2019. doi: 10.1088/1361-6463/aae44d.

[7] R. J. Mailloux, "Electronically scanned arrays," *Synth. Lect. Antennas*, 2007, doi: 10.2200/S00081ED1V01Y200612ANT006.

[8] A. Missoum, "DNA Microarray and Bioinformatics Technologies: A mini-review," *Proc. Nat. Res. Soc.*, 2018, doi: 10.11605/j.pnrs.201802010.

[9] D. Giuffrida, P. Donato, P. Dugo, and L. Mondello, "Recent Analytical Techniques Advances in the Carotenoids and Their Derivatives Determination in Various Matrixes," *Journal of Agricultural and Food Chemistry*. 2018. doi: 10.1021/acs.jafc.8b00309.

[10] I. Altman and A. M. M. Hunter, "The employment and income effects of cleaner coal: The case of FutureGen and rural Illinois," *Clean Technol. Environ. Policy*, 2015, doi: 10.1007/s10098-014-0872-y.

CHAPTER 9

A BRIEF STUDY ON TREES

Rakesh Kumar Yadav, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-rkymuit@gmail.com

ABSTRACT:

This chapter delves into the pivotal realm of trees, an essential data structure profoundly influential in computer science. Trees, inspired by natural branching patterns, facilitate hierarchical data organization and manipulation. This exploration encompasses fundamental principles of tree structures, their versatile applications, traversal strategies, and the significance of diverse tree types. This comprehension equips learners with powerful tools for algorithm design and complex problem-solving.

KEYWORDS:

Algorithm Design, Binary Search Trees, Data Organization Trees, Hierarchical Structures.

INTRODUCTION

Trees stand as a foundational concept in data structures and algorithms, resembling the branching patterns found in nature. These structures offer hierarchical data organization and manipulation, supporting efficient operations. This chapter introduces the core of trees, elucidating their types, inherent characteristics, and extensive applications. Proficiency in trees enables adept algorithm design, data optimization, and effective solutions to computational challenges. The concept of trees is a fundamental cornerstone within the realm of data structures and algorithms, echoing the branching structures prevalent in the natural world.

Trees offer a hierarchical organizational framework that fosters efficient data manipulation and retrieval, making them a crucial element for programmers and computer scientists to comprehend. This chapter embarks on a comprehensive exploration of trees, encompassing their diverse array of types, intrinsic characteristics, and versatile applications. A solid grasp of trees not only equips individuals with the tools to design advanced algorithms and optimize data storage but also empowers them to navigate intricate computational challenges

Types and Characteristics:

At the heart of tree structures lies their intrinsic hierarchy. Trees consist of nodes interconnected by edges, originating from a single distinguished node known as the root. Each node can have child nodes, forming subtrees that extend downwards. This hierarchical arrangement enables the classification of trees into various types based on their properties [1]–[3].

One pivotal type is the binary tree, where nodes possess a maximum of two children: A left child and a right child. This simplicity gives rise to characteristics like ease of traversal and efficient searching. Binary search trees (BSTs) exemplify this type, incorporating a key property where values in the left subtree are smaller than the node's value, and values in the right subtree are larger. BSTs facilitate rapid search, insertion, and deletion operations.

Another vital aspect of tree structures is balance:

Balanced trees ensure that the height of subtrees remains consistent, preventing the degeneration of the tree into a linked list:

1. AVL trees
2. Red-Black trees

These are well-known balanced tree types, embodying self-adjustment mechanisms that sustain logarithmic time complexity even in worst-case scenarios.

Applications:

The significance of trees resonates across diverse domains, reflecting their adaptability to a plethora of applications. In file systems, trees emulate the hierarchical arrangement of directories and files, facilitating efficient storage and navigation. Similarly, in databases, trees are harnessed to organize data hierarchies, enabling rapid retrieval and manipulation. Trees also find a compelling role in artificial intelligence and decision-making processes. Decision trees are a prime example, where nodes represent choices, and edges symbolize outcomes, aiding in making informed decisions based on a series of conditions. Furthermore, hierarchical data representation in family trees, organizational charts, and taxonomies leverages trees to capture intricate relationships [4], [5]. In network routing and communication protocols, trees emerge as foundational structures. Spanning trees, for instance, establish efficient connections among network nodes while minimizing redundancy. In linguistics, syntax trees provide insights into the structure of sentences, aiding in parsing and analysis.

Trees are a cornerstone of data structures and algorithms, presenting an elegant solution for managing hierarchical data. Their diversity of types and characteristics empowers programmers to tailor solutions to specific needs, whether it's efficient searching in binary trees or self-balancing in AVL trees. Understanding trees transcends technicalities, finding application in diverse fields, from file systems to AI decision-making. The chapter on trees is a gateway to harnessing the power of hierarchical data representation, enabling individuals to navigate complex problems, design efficient algorithms, and contribute to the innovation that drives computer science forward. Embracing trees is not just about mastering a data structure; it's about embracing a versatile paradigm that enriches the computational landscape with its inherent structure, efficiency, and adaptability.

Key components of trees include:

Nodes: Nodes are the fundamental building blocks of a tree. Each node holds data and may have references to child nodes, forming the hierarchical structure.

Root: The root is the topmost node in a tree, serving as the starting point for traversal. It has no parent nodes.

Edges: Edges are the connections between nodes, defining the relationships and hierarchy within the tree.

Parent Node: A parent node is a node that has one or more child nodes emanating from it.

Child Nodes: Child nodes are nodes that are directly connected to a parent node through edges, forming subtrees.

Leaf Nodes: Leaf nodes are nodes that have no children, i.e., they are at the terminal ends of the tree structure.

Subtree: A subtree is a portion of a tree that consists of a node and all its descendants.

Depth: The depth of a node represents the length of the path from the root to that node. The depth of the root is typically considered to be 0.

Height: The height of a node is the length of the longest path from that node to a leaf. The height of the tree is the height of the root node.

Binary Tree: A binary tree is a tree where each node has at most two children: a left child and a right child.

Binary Search Tree (BST): A specific type of binary tree where each node's left subtree contains values less than the node's value, and the right subtree contains values greater.

Balanced Tree: A balanced tree is a tree where the height difference between the left and right subtrees of any node is limited, ensuring efficient operations.

Unbalanced Tree: An unbalanced tree is a tree where the height difference between the left and right subtrees of nodes can be significant, potentially leading to inefficient operations.

Traversal: Traversal refers to the process of visiting all the nodes of a tree systematically. Common traversal methods include in-order, pre-order, and post-order.

In-Order Traversal: In this method, nodes are visited in ascending order, which is often used to retrieve data in sorted order.

Pre-Order Traversal: In pre-order traversal, the root node is visited first, followed by its children. It is useful for creating a copy of the tree.

Post-Order Traversal: Post-order traversal visits the children of a node before visiting the node itself. It's often used in tasks like freeing memory in a tree structure.

Spanning Tree: In a graph theory context, a spanning tree is a tree that includes all the vertices of the graph, forming a connected subgraph.

Rooted Tree: A tree where one node is designated as the root, and the remaining nodes have a parent-child relationship.

Unrooted Tree: A tree where no node is explicitly designated as the root, and all nodes have a parent-child relationship.

These components collectively define the structure, relationships, and behavior of trees, making them a crucial concept in computer science and algorithms.

DISCUSSION

Trees encompass a diverse range of structures, each tailored to address specific computational challenges. One notable variant is the binary tree, characterized by nodes having at most two children: a left child and a right child. Binary trees find applications in a multitude of scenarios, with binary search trees (BSTs) being a prime example. In BSTs, every node holds a value, and

the left subtree contains values smaller than the node's value, while the right subtree holds values greater. This arrangement facilitates efficient search, insertion, and deletion operations.

Balanced trees emerge as a critical solution to mitigate performance issues associated with highly skewed trees. AVL trees and Red-Black trees are well-known instances of balanced trees. Maintaining a balanced structure ensures that worst-case operations remain efficient, and the tree does not degrade into a mere linked list. The elegance of balanced trees lies in their ability to self-adjust, guaranteeing logarithmic time complexity for essential operations.

In the realm of traversal algorithms, the in-order, pre-order, and post-order traversals hold significance. In-order traversal explores nodes in ascending order, making it ideal for scenarios such as retrieving data in sorted order. Pre-order traversal starts at the root and visits the nodes before their children, which is useful for copying the structure of a tree. Post-order traversal processes the children before the parent node and is valuable for tasks like deleting a tree.

Trees, as a foundational principle within the sphere of data structures and algorithms, echo the intricate branching patterns observed in the natural world. These hierarchical structures form the bedrock of efficient data organization and manipulation, serving as a linchpin in computer science. This segment delves comprehensively into the nuances of trees, unraveling their diverse types, inherent characteristics, and multifaceted applications that underscore their paramount significance [6]–[8].

Central to this concept is the essence of hierarchy. Trees comprise nodes interconnected by edges, emanating from a singular, distinguished node referred to as the root. Nodes within trees possess the capability to give rise to subtrees, establishing a branching structure that extends downward, akin to the natural branching patterns witnessed in various phenomena. Of particular prominence within the spectrum of tree structures is the binary tree. Here, each node can house a maximum of two children typically designated as the left child and the right child.

This intrinsic simplicity engenders remarkable properties that facilitate streamlined traversal, efficient searching, and effective sorting operations. The binary tree framework constitutes a cornerstone of computer science, fostering the genesis of numerous algorithms and methodologies for data storage. Within the realm of binary trees, Binary Search Trees (BSTs) emerge as a foundational application. These structures are characterized by their organization—each node retains a value, and the left subtree contains values less than the node's value, while the right subtree houses values greater. This configuration facilitates rapid search, seamless insertion, and efficient deletion of elements. However, it's important to underscore that without appropriate balance, a BST can devolve into an inefficient linked list, thereby rendering operations less effective. This brings into play the critical notion of balance, an essential factor within tree structures. The concept of balance assumes paramount importance in ensuring the efficacy of operations conducted within tree structures. An imbalanced tree can potentially lead to skewed performance, wherein specific operations experience disproportionate time consumption. To counteract this, an array of balanced tree structures has been conceptualized. Among these, AVL trees and Red-Black trees stand out as exemplary instances. These structures incorporate mechanisms that ensure automatic adjustment during insertion and deletion operations, thereby preserving a balanced state. This inherent ability to self-balance contributes to the maintenance of relatively uniform subtree heights, consequently enabling logarithmic time complexity for crucial operations.

Traversal algorithms present systematic techniques for navigating through the nodes of a tree. The in-order traversal, for instance, involves visiting nodes in ascending order, rendering it invaluable for endeavors requiring data retrieval in an organized fashion. Conversely, the pre-order traversal entails visiting the root node before its children—an approach particularly useful for tasks such as replicating a tree's structure. On the other hand, the post-order traversal entails traversing children before the parent node, often employed for tasks such as tree deletion.

Bearing in mind the intrinsic depth of trees, their applications extend significantly beyond the realm of technical intricacies. Trees serve as the backbone of file systems, mirroring the hierarchical arrangement of directories and files and thereby facilitating efficient storage and navigation. In the domain of databases, trees come to the fore, facilitating the structuring of data hierarchies and streamlining processes associated with retrieval and management. In the arena of artificial intelligence, decision trees assume a central role as models for decision-making based on sequential conditions, proving invaluable for tasks such as classification and prediction [9], [10].

In the intricate landscape of network protocols, trees play an instrumental role in establishing efficient connections and minimizing redundancy. Spanning trees, in particular, contribute to network routing by ensuring resilient communication pathways while minimizing resource overlap. Moreover, trees find utility within linguistics, where they serve as syntax trees, unraveling sentence structures and contributing to linguistic parsing and analysis.

CONCLUSION

In conclusion, trees stand as an indispensable concept in computer science, serving as a powerful means of organizing and processing hierarchical data. By grasping the fundamentals of trees, individuals gain access to a diverse toolkit for solving intricate problems efficiently.

The array of tree types, ranging from binary trees to balanced trees, caters to diverse computational needs, ensuring optimal performance across various scenarios. Traversal algorithms further enhance the utility of trees by providing systematic methods to navigate through nodes.

Beyond their role in data structures and algorithms, trees have a lasting impact on various domains, influencing database design, compiler construction, and network protocols. The knowledge of working with trees equips aspiring programmers and computer scientists with a versatile skill set, enabling them to craft elegant solutions to a myriad of computational challenges.

The chapter on trees illuminates not only the technical aspects but also the conceptual beauty that arises from the interplay of hierarchy, structure, and efficient algorithms. By embracing the depth of trees, individuals unlock the potential to design innovative algorithms, unravel complex problems, and contribute to the ever-evolving landscape of computer science.

Trees emerge as the cornerstone underpinning data structures and algorithmic design, embodying hierarchical data representation and manipulation. Their diverse typologies, from binary trees to balanced counterparts, equip programmers with the versatility to tailor solutions to diverse demands. Beyond the boundaries of technical understanding, trees permeate diverse domains, including file systems, artificial intelligence, and network protocols.

This chapter not only acquaints readers with the profound concept of trees but also unravels a universe characterized by hierarchical organization—a universe that constitutes the backbone of efficient algorithms, optimized data storage, and astute solutions to intricate computational challenges. Proficiency in trees extends beyond theoretical grasp, presenting an adaptable framework that enriches the computational vista through its inherent structure, efficiency, and adaptability.

REFERENCES

[1] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, 2006, doi: 10.1007/s10994-006-6226-1.

[2] D. Bertsimas and J. Dunn, "Optimal classification trees," *Mach. Learn.*, 2017, doi: 10.1007/s10994-017-5633-9.

[3] W. P. Maddison, "Gene trees in species trees," *Syst. Biol.*, 1997, doi: 10.1093/sysbio/46.3.523.

[4] J. M. Luna *et al.*, "Building more accurate decision trees with the additive tree," *Proc. Natl. Acad. Sci. U. S. A.*, 2019, doi: 10.1073/pnas.1816748116.

[5] K. R. Covey and J. P. Megonigal, "Methane production and emissions in trees and forests," *New Phytologist*. 2019. doi: 10.1111/nph.15624.

[6] L. A. Hug *et al.*, "A new view of the tree of life," *Nat. Microbiol.*, 2016, doi: 10.1038/nmicrobiol.2016.48.

[7] J. B. Turner-Skoff and N. Cavender, "The benefits of trees for livable and sustainable communities," *Plants People Planet*. 2019. doi: 10.1002/ppp3.39.

[8] D. J. P. Gonçalves, B. B. Simpson, E. M. Ortiz, G. H. Shimizu, and R. K. Jansen, "Incongruence between gene trees and species trees and phylogenetic signal variation in plastid genes," *Mol. Phylogenet. Evol.*, 2019, doi: 10.1016/j.ympev.2019.05.022.

[9] S. Geiselhart, K. Hoffmann-Sommergruber, and M. Bublin, "Tree nut allergens," *Mol. Immunol.*, 2018, doi: 10.1016/j.molimm.2018.03.011.

[10] Y. Malhi *et al.*, "New perspectives on the ecology of tree structure and tree communities through terrestrial laser scanning," *Interface Focus*. 2018. doi: 10.1098/rsfs.2017.0052.

CHAPTER 10

A BRIEF STUDY ON AVL TREE

Vaishali Singh, Assistant Professor,
Maharishi School of Engineering & Technology, Maharishi University of Information Technology,
Uttar Pradesh, India
Email Id-singh.vaishali05@gmail.com

ABSTRACT:

An AVL tree is described as a self-balancing Binary Search Tree (BST) in which there can never be more than one difference between the heights of the left and right subtrees for any given node. The 1962 publication "An algorithm for the organization of information" that its creators Georgy A delson-Velsky and Evgenii Landis wrote gave the AVL tree its name. The balance factor of a node in an AVL tree is the difference in height between that node's left subtree and right subtree. The balance factor is equal to either the height of the right or left subtree divided by the height of the left subtree. The balance factor keeps an avl tree's ability to balance itself in check. The balancing factor's value must always be either -1, 0 or +1.

KEYWORDS:

AVL Tree Operations, Deletion, Height-Balance Property, Insertion, Rotation, Self-Balancing Tree.

INTRODUCTION

The first self-balancing binary search tree data structure was created by G.M. Adelson-Velsky and E.M. Landis in the early 1960s and was known as the AVL Tree. A binary search tree (BST) with a self-balancing constraint that states the height differences between the left and right subtrees cannot be more than one is known as an AVL tree.

The general form of an AVL tree is forced by this condition, which is restored following each tree update. Let's first consider the significance of balance before moving on. Consider a binary search tree that was created by starting with an empty tree and adding values in the numbers 1, 2, 3, and 4. The worst-case scenario (BST), where all common operations such as search, insertion, and deletion take O(n) time to complete. We guarantee that each common operation's worst-case execution time is O(log n) by using a balancing condition.

Irrespective of the order in which values are put, the height of an AVL tree with n nodes is O(log n) [1]–[3]. The AVL balancing condition, sometimes referred to as the node balance factor, is a further piece of data that is recorded for every node. This is paired with a method that successfully restores the tree's balancing condition. The creators employ a well-known method called tree rotation in an AVL tree. An AVL is a self-balancing Binary Search Tree (BST) in which no node's left and right subtrees cannot differ in height by more than one.

Essential Points:

1. The tree's height is balanced.
2. A binary search tree is used.

3. The height difference between the left subtree and the right subtree is practically one in this binary tree.
4. Maximum depth from root to leaf is measured in height.

Features of the AVL Tree:

1. It adheres to the fundamental characteristics of a binary search tree.
2. Each branch of the tree is balanced, with a maximum height difference of one between the left and right branches.
3. When a new node is added, the tree automatically balances itself. Consequently, the insertion procedure takes a long time.

Use of the AVL Tree:

1. Avl trees are used to store the majority of in-memory sets and dictionaries.
2. AVL trees are also often used in database applications, where insertions and deletions are less common but frequent data lookups are required.
3. It is used in various applications that require greater searching in addition to database applications.
4. Red-black trees, as opposed to AVL trees, are used in the majority of STL implementations of the ordered associative containers (sets, multisets, maps, and multimaps) [4], [5].

AVL trees have the benefit:

1. Being self-balancing.
2. Additionally, quicker search procedures are provided.
3. Additionally, AVL trees may balance themselves via a different kind of rotation.
4. superior than other trees, such the binary Tree, in terms of searching time complexity.
5. Height must not exceed log(N), where N is the sum of all nodes in the tree.

AVL trees have the following drawbacks:

1. They are challenging to install
2. For some procedures, AVL trees have significant constant factors.

Maximum and minimum node counts:

1. 2H+1 - 1 is the maximum number of nodes.
2. minimum amount of height nodes Where H(0)=1, H = minimum number of nodes of height (H-1) + minimum number of nodes of height (H-2) +H(1)=2.

DISCUSSION

Rotations of Tree:

A tree rotation on a binary search tree is a constant-time operation that modifies the tree's form while maintaining common BST features. Both left and right rotations reduce a BST's height by shifting smaller subtrees downward and bigger subtrees upward**.**

Only when the Balance Factor is not one of the following values: -1, 0, or 1 do we rotate the AVL tree. Rotations may often be divided into four categories, which are as follows:

1. **Left-Left Rotation:** The newly added node is located in the left subtree of A's left subtree.
2. **Right-Right Rotation:** The newly added node is located in the right subtree of A's right subtree.
3. **Left-Right Rotation:** The inserted node is located in the left subtree of A's right subtree.
4. **Right-Left Rotation:** The inserted node is located in the right subtree of A's left subtree.

Algorithm for left rotation:

1. Step 1: Left Rotation(node).
2. Step 2: Prior to: node. Right! =∅.
3. Step 3: Node, post. The new root of the subtree is to the right.
4. Step 4: Node has changed to node. Left kid and right.
5. Step 5: The BST's characteristics are kept.
6. Step 6: end Right Node node. Right node. Right Right Node. Left node.
7. Step 7: end Left Rotation.

Algorithm for right rotation:

1. Step 1: Right Rotation(node)
2. Step 2: Node before. Left ! = ∅
3. Step 3: Node, post. The subtree's new root is on the left;
4. Step 4: node has changed to node's attributes, as well as the offspring of the left, are retained.
5. Step 5: node Left Node. Right node. Right Left Node. Left
6. Step 6: Right Node. Left Node node
7. Step 7: end Right Rotation

Tree Rebalancing: The procedure we outline in this section confirms that the height difference between the left and right subtrees is, at most, 1. If this characteristic is absent, the rotation is done correctly. You'll see that we employ two novel techniques to simulate double rotations. Left And Right Rotation and Right And Left Rotation are the names of these algorithms. The titles of the algorithms, such as Left And Right Rotation, serve as self-documentation. Conducts a left rotation first, then a right rotation, in that order [6]–[8].

Algorithm:

1. Step 1: CheckBalance(current) algorithm
2. Step 2: Pre: current is the node from which balancing should begin.
3. Step 3: Post: current height has been adjusted, and tree balance is being considered if necessary.
4. Step 3: recovered by rotations
5. Step 4: if up to date. Left = present and.Right = ∅
6. Step 5: Present. Height is one;
7. else
8. Step 6: Present. Height = Max((current.Left,current.Right)) + 1

9. Step 7: terminate if
10. Step 8: If Height(current. Left) - Height(current.Right) is more than one
11. Step 9: If Height(current.Left.Left) minus Height(current.Left.Right) is more than zero.
12. Step 10: (current) Right Rotation
13. Step 11: else
14. Step 12: (current) LeftAndRightRotation
15. Step 13: end if
16. Step 14: If Height(current.Left) - Height(current.Right) 1, then else
17. Step 15: If Height(current.Right.Left) - Height(current.Right.Right) > zero.
18. Step 16: (current) LeftRotation
19. Step 17: else
20. Step 18: RightAndLeftRotation(current)
21. Step 19: end if
22. Step 20: end if
23. Step 21: end Check Balance.

Insertion:

AVL insertion works by first inserting the provided value similarly to BST insertion and then, if necessary, by using rebalancing algorithms. The latter is only carried out if the AVL property no longer applies, that is, if there is a height difference of more than one between the left and right subtrees. Every time a node is added to an AVL tree:

As with BST insertion, we start by going down the tree to locate the best place to insert the node. Next, we travel up the tree from the inserted node to ensure that the node's balancing property has not been violated; if it hasn't, we don't need to rebalance the tree; however, if it has, we do. Figure 1 shown insertion for AVL Tree.

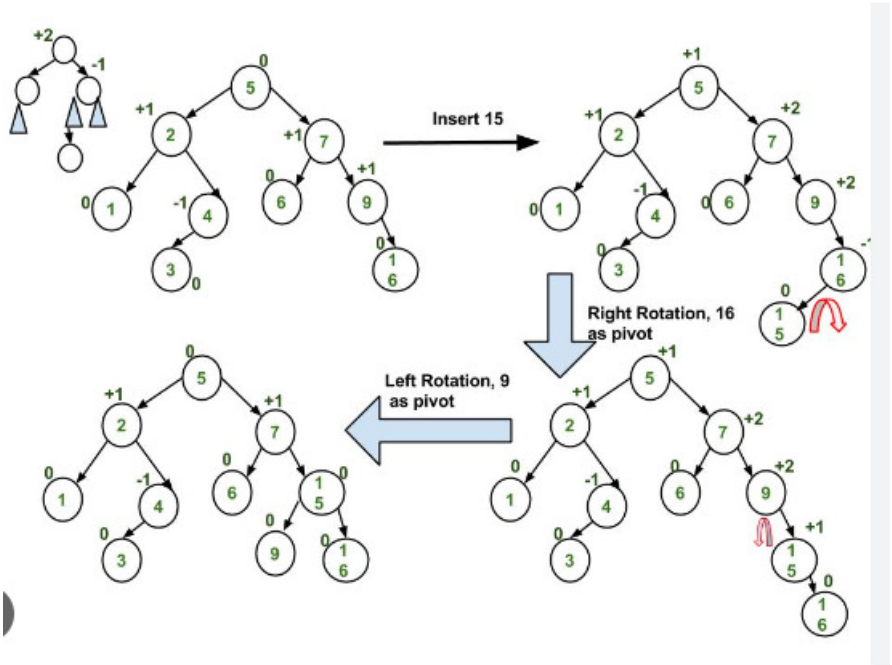


Figure 1: For Insertion OF AVL Trees (geeksforgeeks.org).

Deletion:

Our balancing method resembles the BST technique that was provided. The main distinction is that when the node is removed, we must make sure that the tree still complies with the AVL balance criterion. If the value we are deleting is contained within the tree and the tree doesn't need to be rebalanced, no additional steps are necessary. We must, however, make the appropriate rotation(s) if the value is already in the tree and its removal disturbs the AVL balance characteristic [9], [10].

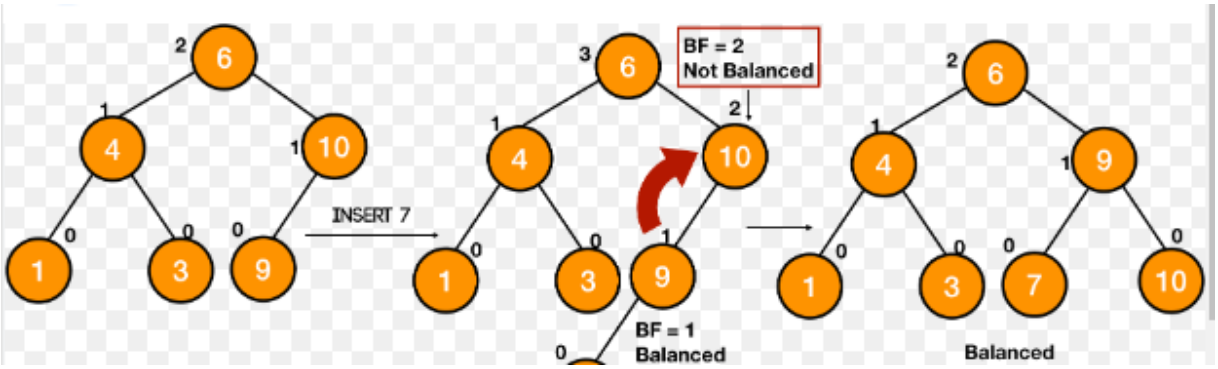


Figure 2: For Deletion OF AVL Tree (codesdope.com).

CONCLUSION

A complex self-balancing tree is the AVL tree. It may be compared to the binary search tree's younger, wiser brother. The AVL tree, in contrast to its elder sibling, avoids worst-case linear complexity runtimes for its operations. The AVL tree ensures that the difference between the heights of the left and right subtrees is at most 1, resulting in a runtime complexity that is at most logarithmic. This is done by enforcing balancing methods.

REFERENCES:

[1] M. Amani, "Gap terminology and related combinatorial properties for AVL trees and Fibonacci-isomorphic trees," *AKCE Int. J. Graphs Comb.*, 2018, doi: 10.1016/j.akcej.2018.01.019.

[2] R. W. Irving and L. Love, "The suffix binary search tree and suffix AVL tree," *J. Discret. Algorithms*, 2003, doi: 10.1016/S1570-8667(03)00034-0.

[3] M. Long, Y. Li, and F. Peng, "Dynamic provable data possession of multiple copies in cloud storage based on full-node of AVL tree," *Int. J. Digit. Crime Forensics*, 2019, doi: 10.4018/IJDCF.2019010110.

[4] C. C. Foster, "A Generalization of AVL Trees," *Commun. ACM*, 1973, doi: 10.1145/355609.362340.

[5] K. S. Larsen, "AVL trees with relaxed balance," *J. Comput. Syst. Sci.*, 2000, doi: 10.1006/jcss.2000.1705.

[6] M. Amani, K. A. Lai, and R. E. Tarjan, "Amortized rotation cost in AVL trees," *Inf. Process. Lett.*, 2016, doi: 10.1016/j.ipl.2015.12.009.

[7] A. K. Tsakalidis, "AVL-trees for localized search," *Inf. Control*, 1985, doi: 10.1016/S0019-9958(85)80034-6.

[8] L. Bounif and D. E. Zegour, "Toward a unique representation for AVL and red-black trees," *Comput. y Sist.*, 2019, doi: 10.13053/CyS-23-2-2840.

[9] M. Amani, "New Terminology and Results for AVL Trees," *Electron. Notes Discret. Math.*, 2017, doi: 10.1016/j.endm.2017.11.004.

[10] S. T. Klein, "AVL Trees," in *Basic Concepts in Data Structures*, 2016. doi: 10.1017/cbo9781316676226.006.

CHAPTER 11

A BRIEF STUDY ON STACKS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology,
Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

There are several instances of stacks in everyday life. Take the canteen's plate stacking arrangement as an illustration. The plate that is placed at the bottom of the stack stays there for the greatest amount of time since the plate at the top is the first to be taken out.

Therefore, it is clear that it follows the LIFO/FILO (First In, Last Out) sequence. The Last-In-First-Out (LIFO) concept is used by Stacks, a type of linear data structure. The Queue has two ends (front and back), but the Stack only has one.

It only has one pointer, top pointer, which points to the stack's topmost member. When an element is added to the stack, it is always added to the top, and it can only be removed from the stack. To put it another way, a stack is a container that allows insertion and deletion from the end known as the stack's top.

KEYWORDS:

Expression Evaluation, Function Call Management, Last-In-First-Out (LIFO), Push, Pop, Peek, Stack Operations, Undo Operations.

INTRODUCTION

A stack is a linear data structure in which adding new elements and removing old ones happen at the same end, which is symbolized as the stack's top. Because we can only access the elements on the top of the stack, maintaining the reference to the top of the stack, which is the last element to be placed, is necessary to implement the stack.

Last In First Out (LIFO):

According to this method, the piece that was added last will appear first. As an actual illustration, consider a stack of dishes stacked on top of one another. We may claim that the plate we put last comes out first because the plate we put last is on top and we take the plate at the top [1]–[3].

Basic Stack Operations:

Certain actions are made available to us so that we can manipulate a stack.

1. Push() to add a component to the stack
2. top() returns the element at the top of the stack, pop() removes an element from the stack.
3. is Empty() returns true if the stack is empty and false otherwise.
4. The stack's size is returned via size().

Several important aspects of stack:

It is known as a stack because it functions just like actual stacks, such as heaps of books. As an abstract data type with a predetermined capacity, a stack can only hold components that of a certain size. It is a data structure that inserts and deletes elements in a specific sequence, which can either be LIFO or FILO.

Working of Stack:

The LIFO pattern is used in stacking. The stack has a size of 5, because there are five memory blocks in it, as seen in the image below. Assume we wish to keep the components in a stack, which is now empty. As illustrated below, we have taken a stack of size 5, and we are adding items one at a time till the stack is filled.

Operation PUSH:

The following list of stages outlines the PUSH operation:

1. We determine whether a stack is full before adding an element to it.
2. The overflow problem arises if we attempt to put the element into a full stack.
3. To ensure that a stack is empty, we begin it with a top value of -1.
4. When a new element is added to a stack, the top value is first increased, top=top+1, and then the element is added at the new top position.
5. Up until the stack's maximum size, the items will be added.

Operation POP:

The following list of stages describes the POP operation:

1. We verify if the stack is empty before removing the piece from it.
2. The underflow problem manifests itself if we attempt to remove the element from the empty stack.
3. The element indicated by the top of the stack is the first thing we access if the stack is not empty.
4. The top is decreased by 1 when the pop operation is completed, or top=top-1.

Applications of Stack:

The stack's applications include the following:

Symbol balancing: A symbol is balanced using a stack. Each program contains an opening and a closing brace, which are respectively pushed into a stack when the opening brace appears and removed when the closing brace does.

As a result, the net value is equal to zero. If any symbol remains in the stack, some syntax in the program has taken place. Prior to reaching the null character, we push every character of the string onto a stack [4], [5].

Once all the characters have been pushed, we begin removing them one at a time until we reach the bottom of the stack.

UNDO/REDO:

It may also be utilized to carry out UNDO/REDO activities. As an illustration, suppose we had an editor where we write "a," "b," and "c"; the result is the text abc. As a result, three states—a, ab, and abc are kept in a stack. Two stacks would be present, one of which would display the UNDO state and the other the REDO state. If 'ab' state is what we're going for and UNDO operation is what we want to do, then pop operation is what we put into practice.

Recursion: When a function recurses, it calls itself one more. The compiler builds a system stack in which all of the function's prior records are kept in order to maintain the previous states.

DFS (Depth First Search): This search is carried out using the stack data structure on a graph.

Backtracking: Let's say we need to design a route to tackle a maze-related issue. if we are traveling along a certain road when we realize we have taken the wrong turn. We must utilize the stack data structure to return to the path's beginning and establish a new path [6]–[8].

Conversion of expressions: Stack can also be used for this. One of the most significant applications of stack is this. Following is a list of expression conversions:

1. Prefix to infix
2. Postfix to infix
3. Infix to prefix
4. postfix to prefix
5. Infix to Postfix.

DISCUSSION

Implementation of Stack in an array

When utilizing an array, the stack is created using the array. Arrays are utilized for all stack-related activities. Let's see how the array data structure may be used to accomplish each action on the stack.

Push operation: Adding an element to the stack

1. Push operations refer to adding an element to the top of the stack. Two steps are involved in a push operation.
2. Increase the value of the Top variable so it may now point to the following memory address.
3. Add an element at the top-incremented position. Adding a new element at the top of the stack is what this is known as.
4. As a result, our primary function must always avoid the stack overflow circumstance. Stack overflow occurs when we attempt to put an element into a fully loaded stack.

Algorithm:

1. Checks if the stack is full.
2. If the stack is full, produces an error and exit.
3. If the stack is not full, increments top to point next empty space.
4. Adds data element to the stack location, where top is pointing.
5. Returns success.

Elimination of a stack element (Pop operation):

Pop operation refers to the removal of an element from the top of the stack. Every time a piece of the stack is removed, the variable top's value is increased by 1. The stack's topmost element is saved in a different variable, after which the top is decremented by 1. The outcome of the operation is the erased value that was previously saved in another variable.

The underflow condition develops when we attempt to remove an element from a stack that is already empty.

Algorithm:

1. Checks if the stack is empty.
2. If the stack is empty, produces an error and exit.
3. If the stack is not empty, accesses the data element at which top is pointing.
4. Decreases the value of top by 1.
5. Returns success.

In a peek action, the element that is currently at the top of the stack is returned without being deleted. If we attempt to return the top element from an already empty stack, an underflow condition can result.

(STACK, TOP) PEEK

Algorithm:

Step 1: function PEEK(STACK, TOP):

Step 2: if TOP >= 0:

Step 3: return STACK[TOP]

Step 4: else:

Step 5: return "Stack is empty"

Implementation of stack using a linked list:

To implement a stack, we may alternatively use a linked list rather than an array. Memory is allocated dynamically using linked lists. However, the temporal complexity for all three operations push, pop, and peek is the same in both scenarios.

The nodes in a linked list implementation of a stack are kept apart from one another in memory. Each node on the stack has a reference to the node that will be its immediate successor. If there isn't enough room on the memory heap to build a node, the stack is said to have overflown.

Implementation stack for DS Linked lists:

The address field of the node at the top of the stack is always empty. Let's talk about how the linked list implementation of the stack carries out each action.

Stacking up a node (Push operation):

Push operations are used to add nodes to the stack. In contrast to array implementation, pushing an element to a stack in a linked list implementation is different. The following actions are required to put an element into the stack [9], [10].

Prior to allocating memory, create a node

If the list is empty, the item should be pushed as the list's start node. This entails giving the data portion of the node a value and giving the address portion of the node a null value.

To avoid violating the stack's property, we must add the new element at the beginning of the list if there are already some nodes there. Make the new node the starting node of the list by setting the address of the initial element in the address field of the new node.

Implementation in stack C of the DS Linked list:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;
};
struct Stack {
   struct Node* top;
};
struct Node* newNode(int data) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->data = data;
   node->next = NULL;
   return node;
}
struct Stack* createStack() {
   struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
   stack->top = NULL;
   return stack;
}
```

```
int isEmpty(struct Stack* stack) {
    return stack->top == NULL;
}
void push(struct Stack* stack, int data) {
    struct Node* newNode = newNode(data);
    newNode->next = stack->top;
    stack->top = newNode;
}
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return -1; // You can choose a suitable value to indicate an error
    }
    return stack->top->data;
}
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return -1; // You can choose a suitable value to indicate an error
    }
    struct Node* temp = stack->top;
    int data = temp->data;
    stack->top = temp->next;
    free(temp);
    return data;
}
int main() {
    struct Stack* stack = createStack();
    push(stack, 10);
    push(stack, 20);
```

```
    push(stack, 30);
    printf("Top element: %d\n", peek(stack));
    printf("Popped element: %d\n", pop(stack));
    printf("Popped element: %d\n", pop(stack));
    printf("Top element: %d\n", peek(stack));
    return 0;
}
```

Taking a node off of the stack (POP operation)

Pop action refers to removing a node from the top of the stack. The process of removing a node from the stack's linked list implementation differs from the array implementation. The actions below must be taken in order to pop an element from the stack:

Check for the underflow condition: When we attempt to pop from an already empty stack, the underflow problem arises. If the list's head pointer points to null, the stack will be empty.

Set the head pointer appropriately: The head pointer's value must be removed, and the node must be released since in a stack, the items are only popped from one end. The head node is now the next node in the chain.

Nodes will be shown (Traversing)

In order to display every node of a stack, you must traverse every node of the linked list that is set up as a stack. The measures below must be taken for this aim.

Create a temporary pointer by copying the head pointer. Print the value field associated with each node by iterating the temporary pointer over the list's nodes.The evaluation of expressions with operands and operators can be done using a stack. Stacks can be used for backtracking, or to verify if an expression's parentheses match. It can also be used to change the phrase from one form to another. It may be applied to formally manage memories.

CONCLUSION

In conclusion, the chapter on stacks in data structures has provided a thorough understanding of this essential linear data structure. Stacks, with their Last-In-First-Out (LIFO) principle, have been explored in terms of their basic concept, operations, and various applications. The chapter covered push and pop operations, illustrating the insertion and removal of elements, along with their implementations using arrays and linked lists. The significance of stacks in solving a range of computational problems has been highlighted, from managing function calls and expressions in programming to undo mechanisms in applications. The balance between simplicity and utility offered by stacks makes them a valuable tool in algorithm design. Furthermore, the chapter touched upon practical considerations such as handling stack overflow and underflow situations, emphasizing the importance of proper error handling. Readers have gained insights into how stacks can be used creatively to solve complex challenges and optimize processes in diverse domains. By grasping the concepts presented in this chapter, readers are equipped with a strong foundation in stacks, which will not only aid in building efficient algorithms but also foster a

deeper understanding of data structures as a whole. As they move forward, readers can confidently apply stack-based solutions to various programming scenarios, knowing the advantages stacks bring to problem-solving and code organization.

REFERENCES:

[1] J. R. Paris, J. R. Stevens, and J. M. Catchen, "Lost in parameter space: a road map for stacks," *Methods Ecol. Evol.*, 2017, doi: 10.1111/2041-210X.12775.

[2] M. J. Wang, R. Choudhury, and J. Sakamoto, "Characterizing the Li-Solid-Electrolyte Interface Dynamics as a Function of Stack Pressure and Current Density," *Joule*, 2019, doi: 10.1016/j.joule.2019.06.017.

[3] S. Aggarwal and J. Verma, "Comparative analysis of MEAN stack and MERN stack," *Int. J. Recent Res. Asp.*, 2018.

[4] J. L. Barton and F. R. Brushett, "A one-dimensional stack model for redox flow battery analysis and operation," *Batteries*, 2019, doi: 10.3390/batteries5010025.

[5] M. Riedel, M. P. Heddrich, A. Ansar, Q. Fang, L. Blum, and K. A. Friedrich, "Pressurized operation of solid oxide electrolysis stacks: An experimental comparison of the performance of 10-layer stacks with fuel electrode and electrolyte supported cell concepts," *J. Power Sources*, 2020, doi: 10.1016/j.jpowsour.2020.228682.

[6] A. Tremouli, J. Greenman, and I. Ieropoulos, "Investigation of ceramic MFC stacks for urine energy extraction," *Bioelectrochemistry*, 2018, doi: 10.1016/j.bioelechem.2018.03.010.

[7] R. M. Spratt and L. E. Lisiecki, "A Late Pleistocene sea level stack," *Clim. Past*, 2016, doi: 10.5194/cp-12-1079-2016.

[8] A. J. Calderón, F. J. Vivas, F. Segura, and J. M. Andújar, "Integration of a multi-stack fuel cell system in microgrids: A solution based on model predictive control," *Energies*, 2020, doi: 10.3390/en13184924.

[9] M. E. S. Youssef, R. S. Amin, and K. M. El-Khatib, "Development and performance analysis of PEMFC stack based on bipolar plates fabricated employing different designs," *Arab. J. Chem.*, 2018, doi: 10.1016/j.arabjc.2015.07.005.

[10] J. Catchen, P. A. Hohenlohe, S. Bassham, A. Amores, and W. A. Cresko, "Stacks: An analysis tool set for population genomics," *Mol. Ecol.*, 2013, doi: 10.1111/mec.12354.

CHAPTER 12

A BRIEF DISCUSSION ON HASHING IN DATA STRUCTURE

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology,Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

A data structure called a hash table uses associative coding to store data. Each data value in an array format is kept in a hash table with a distinct index value. If we are aware of the index for the needed data, access to it becomes quite quick. A method called hashing turns a range of key values into a range of array indexes. The modulo operator will be used to obtain a range of key values. Take a look at a hash table example with a size of 20, and the following things need to be kept. The format of the items is (key,value). As a result, it develops into a data structure in which insertion and search operations, regardless of the quantity of the data, are very quick.

A hash table employs an array as its store medium and generates an index from which an element may be accessed or inserted using a hashing algorithm.

KEYWORDS:

Chaining, Collision Resolution, Hash Function, Hash Map, Open Addressing.

INTRODUCTION

Every day, the amount of data on the internet increases tremendously, making it challenging to adequately store it all. Even though this amount of data may not be large in day-to-day programming, it must be conveniently saved, retrieved, and processed.

The Array data structure is one of the most popular ones used for this kind of function. Now the issue is raised: why create a new data structure if Array already existed? The term "efficiency" has the solution to this. While searching in an array requires at least O(log n) time, storing in one just requires O(1) time.

Although this time seems insignificant, it can really cause a lot of issues for a huge data collection, which renders the Array data structure ineffective. We are now searching for a data structure that can store the data and perform searches within it in O(1) time, or constant time.

The Hashing data structure was developed in this manner. Data may now be readily stored in constant time and retrieved in constant time thanks to the invention of the Hash data structure [1]–[3].

Components of Hashing:

Hashing primarily consists of three parts:

Key: A key is any text or number that is provided as input into a hash function, a method for determining an item's index or placement in a data structure.

Hash Function: The hash function takes a key as input and outputs the index of a hash table's entry. The index is sometimes referred to as a hash index.

Hash Table: A hash table is a type of data structure that uses a unique function called a hash function to map keys to values. Hash holds the data in an array in an associative fashion, giving each data value a distinct index.

Collision:

Since the hashing procedure yields a tiny number for a large key, it is possible for two keys to provide the same result. When a freshly inserted key corresponds to an existing occupied slot, a collision handling technique must be used to address the problem.

Hashing's benefits for data structures:

1. **Support for key-value data structures:** Hashing is a great choice for key-value data structures. Quick access to items with constant-time complexity is made possible via hashing.
2. **Efficiency:** The activities of inserting, deleting, and searching are quite effective. Memory consumption reduction: Since hashing allots a defined place for storing components, it uses less memory.
3. **Scalability:** Hashing works well with big data sets and keeps the access time constant. Secure data storage and data integrity checking depend on hashing, which also provides security and encryption.

Using linear probing:

As we can see, it is possible for the hashing method to be used to produce an array index that has previously been used. If this is the case, we can look at each subsequent cell until we discover one that is empty before moving on to the next empty spot in the array. This method is known as linear probing [4], [5].

Basic Procedures:

The fundamental principal operations of a hash table are listed below.

1. **Search:** Looks up information about a hash table element.
2. **Insert:** adds a new element to the hash table.
3. **Delete:** Removes a hash table element.

Define a data item with some data and a key so that it may be used as the basis for a hash table search.

Hash Technique

Create a hashing algorithm to calculate the key's hash value for the data item. the following: int hashCode(int key) return key% SIZE;

Operation of a Search:

Find the element using the hash code of the key supplied as an index in the array whenever an element has to be found. If the element cannot be located at the computed hash code, use linear probing to advance it.

Example: BinarySearch(arr, n, target):

Step 1: low = 0

Step 2: high = n - 1

Step 3: while low <= high:

Step 4: mid = (low + high) / 2

Step 5: if arr[mid] == target:

Step 6: return mid

Step 7: else if arr[mid] < target:

Step 8: low = mid + 1

Step 9: else:

Step 10: high = mid - 1

Step 11: return -1

Every time an element has to be placed, determine the hash code of the key that was supplied and use that hash code as an index in the array to find the appropriate location. If an element is located at the computed hash code, use linear probing to search for empty locations.

Removal Operation:

Find the index using the hash code of the key supplied and use that as an index in the array whenever an element has to be eliminated. In the event that an element cannot be located at the computed hash code, use linear probing to advance the element. Once located, put a fake item there to maintain the performance of the hash table.

Example: void delete(struct DataItem* item) {

Step 1: int key = item->key;

Step 2: int hashIndex = hashCode(key);

Step 3: while (hashArray[hashIndex] != NULL) {

Step 4: struct DataItem* temp = hashArray[hashIndex];

Step 5: if (temp->key == key) {

Step 6: hashArray[hashIndex] = dummyItem;

Step 7: return temp;

```
    }
  Step 8:  ++hashIndex;
  Step 9:   hashIndex %= SIZE;
  }
  }
```

DISCUSSION

Index Mapping:

In Index Mapping, sometimes referred to as Trivial Hashing, the data is simply mapped to an index in a hash table. This approach commonly uses the identity function as the hash function, which translates the input data to itself. In this instance, the hash table's index is created using the data's key, and the value is saved there.

The naive hashing function would simply map the key "a" to the index "a" in the hash table and store the value "apple" at that index, for instance, if we had a hash table of size 10 and wanted to store the value "apple" with the key "a".

Index Mapping's simplicity is one of its key benefits. The data may be simply accessed using the key, and the hash function is simple to comprehend and use. It does, however, have certain restrictions. The biggest drawback is that because the size of the hash table must match the number of keys, it can only be utilized for modest data sets. Furthermore, it is incapable of handling collisions, which means that if two keys map to the same index, one of the data will be overwritten [6]–[8].

Given a restricted range, an array's elements can be either positive or negative, i.e., between -MAX and +MAX. In O(1) time, our objective is to determine if a given integer is present in the array or not. Since the range is constrained, index mapping (or simple hashing) can be used. In a large array, we utilize values as the index. As a result, searching and inserting items both take O(1) time.

Algorithm:

1. Give the hash matrix's whole range a value of 0.
2. navigating the provided array
3. Assign if the element ele is not negative.
4. Hash[ele][0] as 1 means.
5. Alternatively, take ele's absolute value and
6. Give hash[ele][1] a value of 1.

To look up any x-value in the array:

1. Check to see whether hash[X][0] is 1 or not if X is not negative. The number is present if hash[X][0] is 1, otherwise it is not.
2. If X is negative, determine its absolute value before determining whether or not hash[X][1] is 1. The number is present if hash[X][1] is one.

Time Complexity preceding approach has an O(N) time complexity, where N is the size of the supplied array. Because we are utilizing an array with a maximum size, the above approach has an O(N) space complexity.

Separate Chaining: The goal of separate chaining is to represent the array as a chain, which is a linked list. One of the most popular and often employed methods for handling accidents is separate chaining.

This method is implemented using the linked list data structure. As a result, when numerous entries are hashed into the same slot index, those elements are added to a chain, which is a singly-linked list.

Here, a linked list is created out of all the entries that hash into the same slot index. Now, using merely linear traversal, we can search the linked list with a key K. It signifies we have located our entry if the intrinsic key for any entry equals K.

The entry does not exist if we have searched all the way to the end of the linked list and still cannot find it. In separate chaining, we thus get to the conclusion that if two different entries have the same hash value, we store them both in the same linked list one after the other.

Advantages:

1. Easy to put into practice.
2. We can always add additional items to the chain, thus the hash table never runs out of space.
3. less susceptible to load factors or the hash function.
4. When it is unclear how many or how frequently keys could be added or removed, it is typically utilized.

Disadvantages:

1. Chaining's cache performance is poor since keys are kept in a linked list. Since everything is stored in the same table, open addressing improves cache speed.
2. Space wastage (some of the hash table's parts are never used)
3. In the worst situation, search time might become O(n) as the chain lengthens.
4. additional space is used for connections.

Execution of Chaining:

The effectiveness of hashing may be assessed under the basic uniform hashing assumption, which states that each key has an equal likelihood of being hashed to any position in the table.

There are m slots in a hash table.

1. N is the quantity of keys to be added to the hash table.
2. The load factor is n/m.
3. Expected search time = O(1 +)
4. Expected deletion time = O(1 +)
5. Insertion time: O(1)
6. If is O(1), then search insert and delete time complexity is O(1).

Chain-Storing Data Structures:

1. Lists with links

O(l) search, where l is the length of the linked list

Remove O(l)

Adding: O(l)

Not cache-compatible

2. Dynamic Sized Arrays (List in Python, ArrayList in Java, and Vectors in C++)

O(l), where l is the array length

Remove O(l)

Adding: O(l)

friendly to cache

Self-balancing BST (Red-Black Trees, AVL Trees)

O(log(l)) search, where l is the length of the linked list

Subtract: O(log(l))

Adding: O(l)

Hash tables employ the collision resolution method known as double hashing. It operates by calculating two distinct hash values for a given key using two hash algorithms. The initial hash value is calculated using the first hash function, and the step size for the probing sequence is calculated using the second hash function. Due to the employment of two hash algorithms to determine the step size and hash value, double hashing has the potential to have a low collision rate. This means that compared to other collision resolution strategies like linear probing or quadratic probing, the likelihood of a collision occuring is smaller [9], [10]. Double hashing does have certain disadvantages, though. First, it necessitates the employment of two hash algorithms, which can make insertion and search operations more computationally complicated. To obtain optimal speed, a suitable selection of hash functions is necessary. The collision rate might still be large if the hash algorithms are poorly constructed.

Benefits of using double hashing:

1. The benefit of double hashing is that it produces a consistent distribution of records across a hash table, making it one of the greatest methods of probing.
2. Clusters are not produced by this method.
3. It is one of the most efficient ways to deal with collisions.
4. You can perform a double hash using:
5. %TABLE_SIZE (hash1(key) + i * hash2(key))
6. The hash functions here are hash1() and hash2(), and TABLE_SIZE.
7. is the hash table's size.
8. (We continue by raising i whenever a collision happens.)

CONCLUSION

As a result, hashing is a practical method for ensuring that files are accurately transferred between two sites. Without opening and comparing the files, it may also be used to determine if they are identical. asking everything into account is a useful way to ensure that the information is correctly copied between two sources. Without opening and comparing them, it may also determine if the information is indistinguishable. Hashing is mostly used for database recovery since it is easier to find an item using a small hash key than it is to find it using the original value.

REFERENCES:

[1] R. Nasr, T. Kristensen, and P. Baldi, "Tree and hashing data structures to speed up chemical searches: Analysis and experiments," *Mol. Inform.*, 2011, doi: 10.1002/minf.201100089.

[2] S. Han, S. Kim, J. Hoon Jung, C. Kim, K. Yu, and J. Heo, "Development of a hashing-based data structure for the fast retrieval of 3D terrestrial laser scanned data," *Comput. Geosci.*, 2012, doi: 10.1016/j.cageo.2011.05.005.

[3] J. Zhang and Y. Peng, "SSDH: Semi-Supervised Deep Hashing for Large Scale Image Retrieval," *IEEE Trans. Circuits Syst. Video Technol.*, 2019, doi: 10.1109/TCSVT.2017.2771332.

[4] J. Schneider and P. Rautek, "A Versatile and Efficient GPU Data Structure for Spatial Indexing," *IEEE Trans. Vis. Comput. Graph.*, 2017, doi: 10.1109/TVCG.2016.2599043.

[5] R. Nasr, D. S. Hirschberg, and P. Baldi, "Hashing algorithms and data structures for rapid searches of fingerprint vectors.," *J. Chem. Inf. Model.*, 2010, doi: 10.1021/ci100132g.

[6] C. S. Ellis, "Concurrency in Linear Hashing," *ACM Trans. Database Syst.*, 1987, doi: 10.1145/22952.22954.

[7] S. S. Abdul-Jabbar and L. E. George, "Fast Dictionary Construction using Data Structure and Numeration Methodology with Double Hashing," *Int. J. Sci. Res.*, 2015, doi: 10.21275/ART20173976.

[8] G. Marçais, B. Solomon, R. Patro, and C. Kingsford, "Sketching and Sublinear Data Structures in Genomics," *Annual Review of Biomedical Data Science*. 2019. doi: 10.1146/annurev-biodatasci-072018-021156.

[9] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, 2004, doi: 10.1016/j.jalgor.2003.12.002.

[10] L. Chi and X. Zhu, "Hashing techniques: A survey and taxonomy," *ACM Computing Surveys*. 2017. doi: 10.1145/3047307.

CHAPTER 13

A BRIEF DISCUSSION ON GRAPHS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

A non-linear data structure made up of vertices and edges is called a graph. In a graph, the edges are the lines or arcs that link any two nodes, while the vertices are occasionally also referred to as nodes. A graph is more precisely made up of a collection of vertices (V) and a set of edges (E). Graph G(E, V) is used to represent it. A graph data structure is made up of a number of nodes that are interconnected and contain data. Let's use an illustration to try to comprehend this. Everything on Facebook is a node. Everything with data is a node, including User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, and Note. Every connection between two nodes is an edge. If you share a photo, join a group, "like" a page, etc., that relationship gains a new edge.

KEYWORDS:

Algorithms, Breadth-First Search, Depth-First Search, Edge, Graph Representation, Shortest Path, Vertex.

INTRODUCTION

A set of vertices and the edges used to link them can be referred to as a graph. In a graph, instead of having a parent-child relationship, the vertices (also known as Nodes) retain any complicated relationship among themselves.

An ordered set G(V, E) can be used to create a graph, where V(G) denotes the set of vertices and E(G) is the set of edges used to link these vertices. The graph G(V, E) in the image below has five vertices (A, B, C, D, and E) and six edges ((A, B), (B, C), (C, E), (E, D), (D, B), and (A, A)).

Sections of a Graph:

Vertices: The basic building blocks of a graph are called vertices. Vertices are sometimes referred to as nodes or vertices. Any node or vertex can have a label or not.

Edges: In a graph, edges are lines that link two nodes together. In a directed graph, it may be an ordered pair of nodes. Any two nodes can be connected by edges in any manner. No regulations exist. Edges are sometimes also referred to as arcs. Any edge may be tagged or left unlabeled [1]–[3].

Many difficulties in everyday life may be resolved with graphs. Networks are depicted using graphs. The networks might be telephone, circuit, or city-wide route networks. Graphs are also employed in social media sites like Facebook and linkedIn. For instance, each individual is represented as a vertex (or node) on Facebook. Each node is a structure that houses data like a person's name, gender, location, and other attributes.

Terminology for Graphs:

If there is an edge connecting two vertices, they are said to be next to one another. Due to the lack of an edge between them, vertices 2 and 3 are not close by.

Path: A path is a collection of edges that connects vertex A with vertex B. The pathways from vertex 0 to vertex 2 are 0-1, 1-2, and 0-2.

An edge (u,v) on a directed graph does not always imply that there is an edge (v,u) as well. In this type of graph, the edges are shown as arrows to indicate their orientation.

Graph, both directed and undirected:

Both directed and undirected graphs exist. In contrast, edges in an undirected graph are not connected to the directions that they are connected to. The graph in the above illustration is an undirected one since none of the edges are connected to any of the directions. Vertex A and B may be crossed from B to A as well as A to B if there is an edge between them.

Edges make up an ordered pair in a directed graph. A particular route from one vertex A to another vertex B is represented by an edge. While node B is known as the terminal node, node A is known as the beginning node [4], [5]. Figure 1 shown undirected graph and Figure 2 shown directed graph.The graph in the accompanying image is directed.

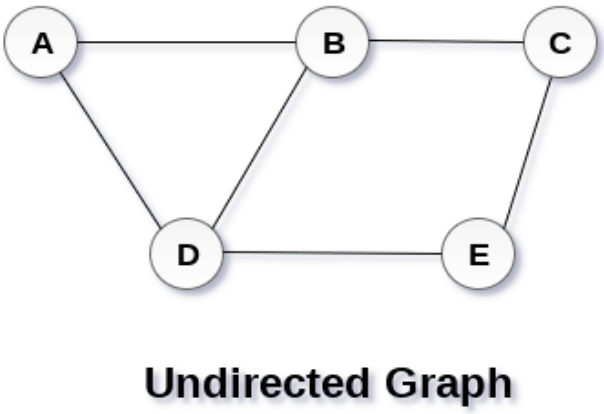


Figure 1:Undirected Graph (geeksforgeeks.org).

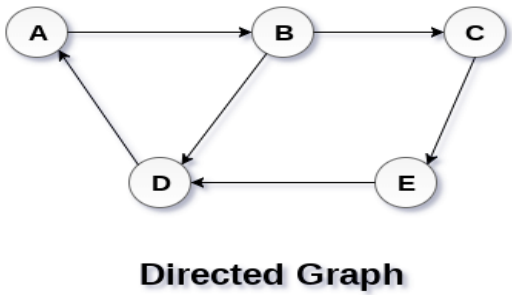


Figure 2:Directed Graph (geeksforgeeks.org).

Representation in Graphs

Graphs are often shown in one of two ways:

1.Adjacency Matrix

First A 2D array of V x V vertices is known as an adjacency matrix. A vertex is represented by each row and column. Any element whose value is 1 is said to have an edge linking its vertex i and vertex j. The graph we produced above has the following adjacency matrix: The value of the matrix element for the row and column with edges is 1 for the row and column with edges, and 0 for the row and column without edges, according to the graph adjacency matrix for the example graph.

Adjacency matrix for graphs

Since the graph is undirected, we must also label edge (2,0) for edge (0,2) in order to make the adjacency matrix symmetric around the diagonal [6]–[8]. In adjacency matrix format, edge lookup (determining if an edge exists between vertex A and vertex B) is incredibly quick, but more space is needed since we must set aside space for every potential link between all vertices (V x V).

2. Adjoining List

A graph is represented as an array of linked lists by an adjacency list.

Each item in the linked list that the index of the array indicates represents a different vertex that forms an edge with that vertex.

DISCUSSION

BFS Algorithm (Breadth first search):

We will talk about the BFS algorithm in the data structure in this post. A graph traversal technique called breadth-first search investigates every node in the graph starting with the root node. Then it chooses the closest node and investigates every undiscovered node. Any node in the graph can serve as the root node when employing BFS for traversal. There are other techniques to navigate the graph, however BFS is the one that is most frequently applied. The process of searching every vertex in a tree or graph data structure is recursive. Every vertex in the graph is divided into two groups by BFS: visited and non-visited. A single node in a network is chosen, and then all of the nodes next to that node are visited.

Uses for the BFS algorithm

The following are some examples of breadth-first algorithm applications:

1. From a given source location, BFS may be used to locate nearby sites.
2. The BFS algorithm may be used as a traversal method in a peer-to-peer network to locate every nearby node. This method is used by the majority of torrent applications, including BitTorrent, uTorrent, etc., to locate "seeds" and "peers" on the network.

3. BFS may be used in web crawlers to build indexes for web pages. It is one of the most important algorithms for indexing web pages. It begins its journey on the source page and then navigates across the page's links. Every web page is viewed in this case as a node in the graph.
4. The shortest path and least spanning tree are found using BFS.
5. Cheney's method of duplicating trash collection also employs BFS.
6. It may be used to calculate the maximum flow in a flow network using the Ford-Fulkerson approach.

Algorithm

The following are the stages the BFS algorithm takes to explore a graph:

1. Step 1: For each node in G, set STATUS to 1 (ready status).
2. Step 2: Enqueue the first node A and set its STATUS to 2 (waiting state).
3. Step 3: Keep going through Steps 4 and 5 until the QUEUE is empty.
4. Step 4: Dequeue a node N. Set its STATUS to 3 (processed state) after processing it.
5. Step 5: Put all of N's neighbors in line who are ready (their STATUS = 1) and set
6. whose STATUS IS 2
7. (State of Waiting)
8. [FINAL LOOP]
9. Step 6: GO OUT

Complexity of BFS:

BFS's time complexity is dependent on the graph representation's data structure. Since the BFS algorithm searches every node and edge in the worst scenario, its time complexity is O(V+E). The number of edges in a graph is O(E), but the number of vertices in a graph is O(V).

The number of vertices, V, determines the BFS's space complexity, which is denoted by the symbol O(V).

The adjacency list is how we describe our graph in this code. Since we only need to go over the list of nodes connected to each node once the node is dequeued from the head (or start) of the queue, implementing the Breadth-First Search algorithm in Java makes dealing with the adjacency list considerably simpler.

DFS Complexity(Depth-First Search):

the data structure's DFS algorithm. A tree data structure or a graph's whole vertex set can be searched using a recursive approach. Beginning with the first node of graph G, the depth-first search (DFS) method digs down until it reaches the target node, also known as the node with no children. The DFS method may be implemented using a stack data structure due to its recursive nature. The DFS algorithm implementation procedure is comparable to that of the BFS algorithm [9], [10].

1. The following is the step-by-step procedure to implement the DFS traversal:
2. Create a stack with all of the graph's vertices in it first.
3. Select whatever vertex you want to use as the first vertex in the traverse, and add it to the stack.
4. After that, raise a non-visited vertex that is close to the vertex at the top of the stack.

5. Repeat steps 3 and 4 up until there are no more vertices to visit from the vertex at the top of the stack.
6. Go back and pop a vertex off the stack if there isn't any vertex remaining.
7. Till the stack is completely empty, repeat steps 2, 3, and 4.

Uses for the DFS algorithm:

1. The following list of uses for the DFS algorithm includes:
2. The topological sorting may be implemented using the DFS algorithm.
3. It may be applied to determine the routes connecting two vertices.
4. It may also be used to find graph cycles.
5. DFS technique is also applied to puzzles with a single solution.
6. If a graph is bipartite or not, it may be determined using DFS.

Algorithm:

1. Step 1: Every node in G should have STATUS set to 1 (ready state).
2. Step 2: Push the first node A into the stack and set its STATUS value to 2 (waiting state).
3. Step 3: Carry out Steps 4 and 5 once more up until STACK is empty.
4. Step 4: Pop the top node N and Set its STATUS to 3 (processed state) after processing it.
5. Step 5: Push all of N's neighbors who are in the ready state (their STATUS = 1) onto the stack and change their STATUS to 2 (the waiting state).
6. [FINAL LOOP]
7. Step 6: EXIT Depth-first search method complexity

The DFS algorithm has an O(V+E) time complexity, where V is the number of graph vertices and E is the number of graph edges. The DFS algorithm has a space complexity of O(V).

CONCLUSION

A useful idea in data structures is graphs. It is used in practically every field in a practical way. Therefore, it is crucial to comprehend the fundamentals of graph theory in order to comprehend the algorithms for graph structures. Graphs were just introduced in this essay. Just a stepping stone, really. It is advised to delve further deeper into the subject of graph theory.

REFERENCES:

[1] H. Cai, V. W. Zheng, and K. C. C. Chang, “A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications,” *IEEE Trans. Knowl. Data Eng.*, 2018, doi: 10.1109/TKDE.2018.2807452.

[2] P. Goyal and E. Ferrara, “Graph embedding techniques, applications, and performance: A survey,” *Knowledge-Based Syst.*, 2018, doi: 10.1016/j.knosys.2018.03.022.

[3] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, “Graph summarization methods and applications: A survey,” *ACM Computing Surveys*. 2018. doi: 10.1145/3186727.

[4] A. Ortega, P. Frossard, J. Kovacevic, J. M. F. Moura, and P. Vandergheynst, “Graph Signal Processing: Overview, Challenges, and Applications,” *Proc. IEEE*, 2018, doi: 10.1109/JPROC.2018.2820126.

[5] J. Yan, C. Wang, W. Cheng, M. Gao, and A. Zhou, "A retrospective of knowledge graphs," *Frontiers of Computer Science*. 2018. doi: 10.1007/s11704-016-5228-9.

[6] T. Wu, G. Qi, C. Li, and M. Wang, "A survey of techniques for constructing Chinese knowledge graphs and their applications," *Sustain.*, 2018, doi: 10.3390/su10093245.

[7] K. Wongsuphasawat *et al.*, "Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow," *IEEE Trans. Vis. Comput. Graph.*, 2018, doi: 10.1109/TVCG.2017.2744878.

[8] P. Veličković, A. Casanova, P. Liò, G. Cucurull, A. Romero, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. doi: 10.1007/978-3-031-01587-8_7.

[9] Y. Li, L. Zhang, and Z. Liu, "Multi-objective de novo drug design with conditional graph generative model," *J. Cheminform.*, 2018, doi: 10.1186/s13321-018-0287-6.

[10] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, 2018, doi: 10.1145/3298989.

CHAPTER 14

A BRIEF DISCUSSION ON SORTING ALGORITHMS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

A 32-bit integer is frequently utilized in all the sorting algorithms in this chapter since its related operations (such as, >, etc.) behave in a manner that is obvious to the user. The presented algorithms may simply be converted into universal sorting algorithms in your chosen language.Rearranging an array or list of elements according to a comparison operator on the elements is done using a sorting algorithm. The new order of the items in the relevant data structure is determined using the comparison operator.

KEYWORDS:

Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Selection Sort.

INTRODUCTION

Bubble Sort:

Comparing each item in a list with every other item is one of the simplest techniques of sorting, however as the description may suggest, it is not especially efficient O(n 2). The simplest way to construct bubble sort is with two loops [1]–[3].

Algorithm:

1) Step 1: BubbleSort(list) algorithm

2) Step 2: Before: 6 =

3) Step 3: Post: The values in the list have been arranged in ascending order.

4) Step 4: List for i 0.To list j 0, count 1

5) Step 5: first.If list[i] > list[j], count = 1

6) Step 6: List[i], List[j] Swap

8) Step 7: End If;

9) Step 8: End For;

10)Step 9: End For;

11) Step 10: Return List;

12) Step 11: End BubbleSort**.**

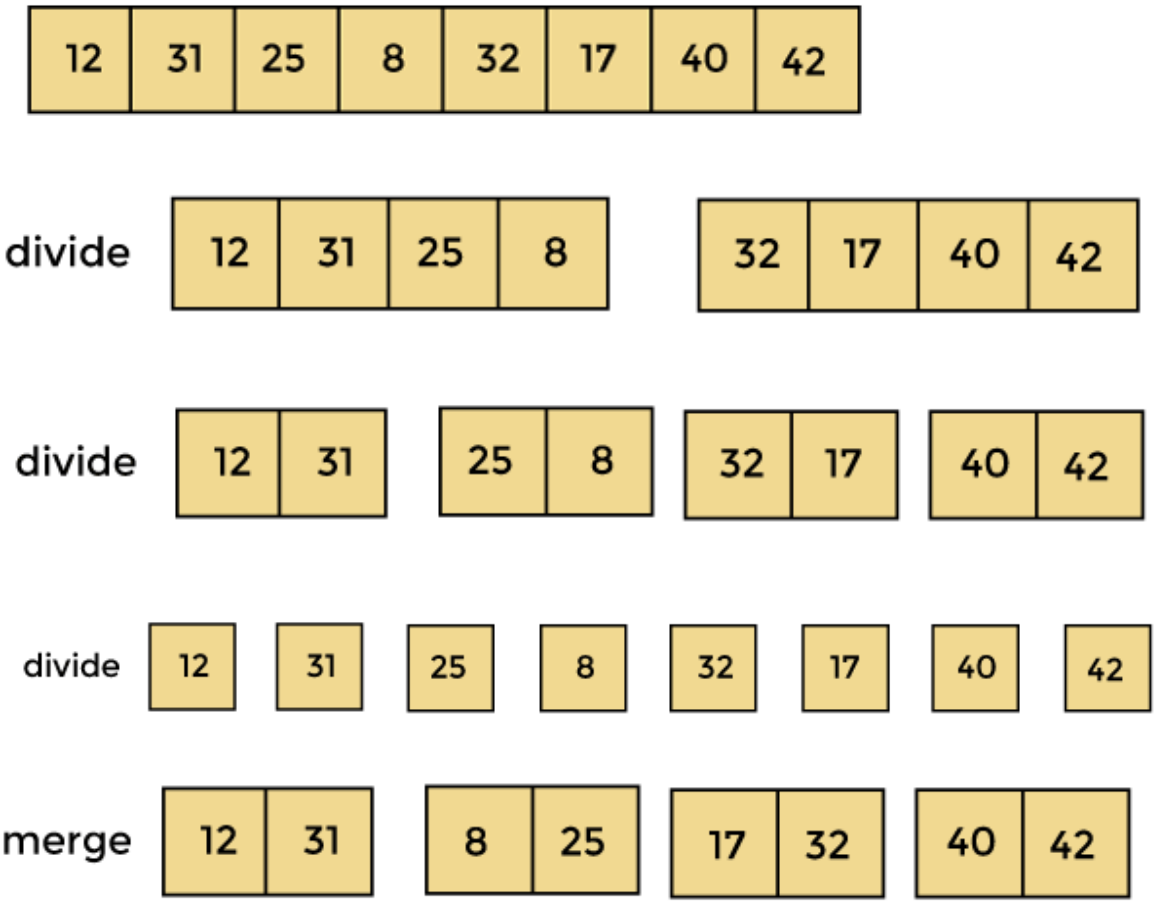
Merge Sort

Merge Sort is a recursive method that repeatedly divides an array in half until there is only one element remaining, at which point it can no longer be divided (an array with one element is always sorted). The sorted subarrays are then combined into a single array. Figure 1 shown working of merge sort.

Algorithm

1. Step 1: MERGE_SORT(arr, beg, end) is the
2. Step 2: If start > end, set mid = (start + end)/2.
3. Step 3: MERGE_SORT (arr, beg, mid)
4. Step 4: MERGE_SORT (arr, mid + 1)
5. Step 5: MERGE_SORT (arr, end)
6. Step 6: End of if: MERGE (arr, beg, mid)
7. Step 7: BOTTOM MERGE_SORT

Working of Merge Sort:



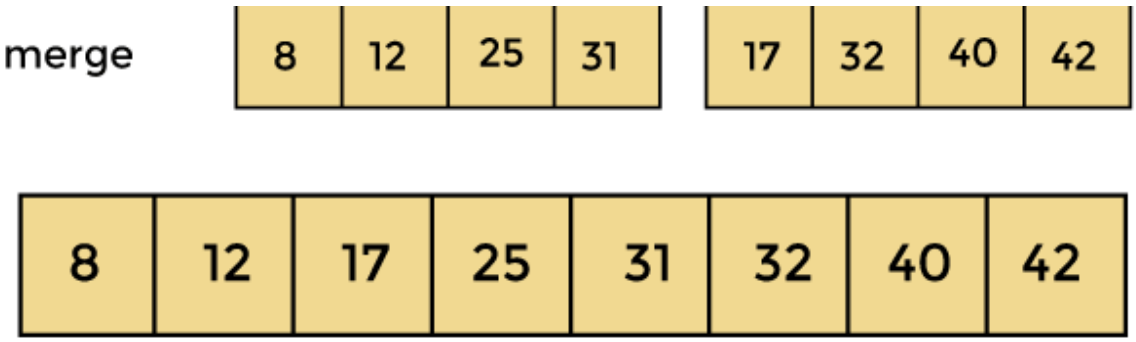


Figure 1: Working of merge sort (javatpoint.com).

QUICK Sort

One of the most often used sorting algorithms is quick sort, which has an O(n log n) complexity and is based on the divide et impera method. The pivot element is chosen by the algorithm as the initial element, and all smaller components are moved in front of it while all larger elements are moved in back. Partition is the primary quick sort process that is repeatedly applied to smaller and larger sublists until their size equals one or zero, at which point the list is implicitly sorted [4], [5].

It is crucial to pick a suitable pivot, such as the median element, to prevent substantially lowered performance of O(n 2). Figure 2 shown working of quick sort.

Algorithm

function QuickSort(arr, low, high):

Step 1: if low < high:

Step 2: pivotIndex = Partition(arr, low, high)

Step 3: QuickSort(arr, low, pivotIndex - 1)

Step 4: QuickSort(arr, pivotIndex + 1, high)

Step 5: function Partition(arr, low, high):

Step 6: pivot = arr[high]

Step 7: i = low - 1

Step 8: for j = low to high - 1:

Step 9: if arr[j] < pivot:

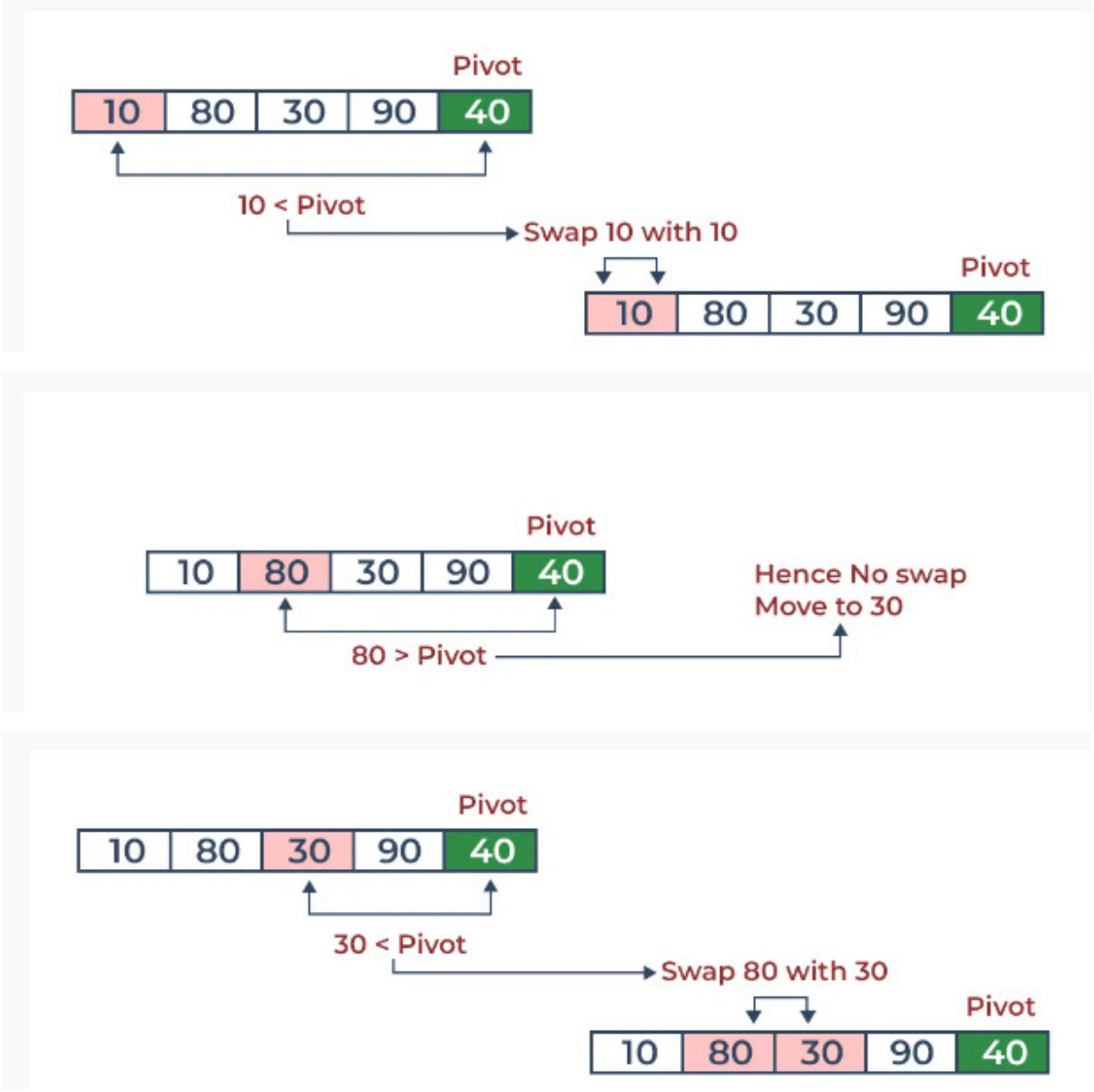
Step 10: i = i + 1

Step 11: Swap(arr[i], arr[j])

Step 12: Swap(arr[i + 1], arr[high])

Step 13: return i + 1

Working of Quick Sort:



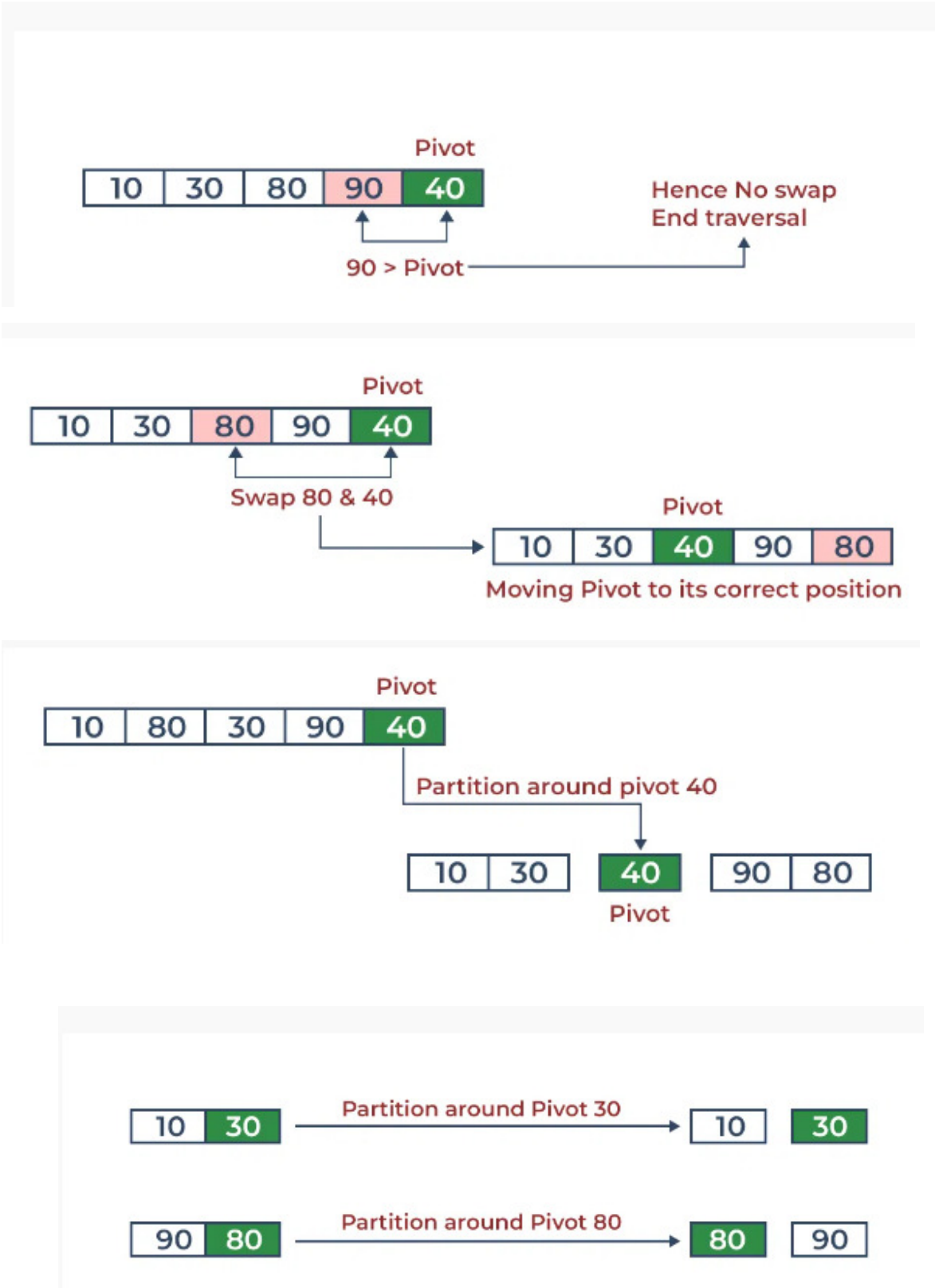


Figure 2: working of Quick Sort(geeksforgeeks.org).

DISCUSSION

Insertion Sorts:

We shall talk about the Insertion sort algorithm in this article. The way that insertion sort operates is similarly straightforward. Students who could encounter an insertion sort question in their exams will find this essay to be both highly informative and engaging. So it's crucial to have a conversation about it. Similar to how playing cards are sorted in hands, insertion sort operates similarly. In the card game, it is expected that the initial card is already sorted before we choose an unsorted card.

The chosen unsorted card will be positioned to the right if it is larger than the first card and to the left if it is not. Similar to that, all unsorted cards are grabbed and placed exactly where they belong. The insertion sort uses the same strategy. The insertion sort works by iterating one element at a time over the sorted array. Despite being easy to use, insertion sort is not suitable for huge data sets since its temporal complexity in both the best and worst cases is O(n2), where n is the number of items. In comparison to other sorting algorithms like heap sort, fast sort, merge sort, etc., insertion sort is less effective.

1. Insertion sort offers a number of benefits, including:
2. straightforward execution
3. effective for little data sets
4. It is adaptive, meaning that it works with data sets that have previously been considerably sorted.
5. Let's now examine the insertion sort method.

Algorithm

1) The following is a list of the easy methods to implement the insertion sort:
2) Step 1: assume that the element is already sorted if it is the first element. Get 1 back.
3) Step 2: Pick the next componentand place it in a key independently.
4) Step 3: Right now, evaluate the key against each component of the sorted array.
5) Step 4: Move on to the next element if the element in the sorted array is smaller than the one you are currently on. If not, move the array's larger items to the right.
6) Step 5: Insert the value.
7) Step 6 - Continue doing this until the array is sorted.
8) The Insertion Sort Algorithm in Action.

Selection Sort Algorithm

We shall talk about the Selection sort algorithm in this article. The selecting sort's operational process is very straightforward. Students who might have to answer a question on selection sort in their exams will find this material to be both highly informative and engaging. So it's crucial to have a conversation about it. In a selection sort, the unsorted array items with the smallest values are picked out in each pass and moved to the proper location inside the array. The algorithm is also the simplest.

It is an algorithm for in-place comparison sorting. In this approach, the array is split into two sections: one that is sorted and the other that is not. The array's sorted portion is initially empty, while its unsorted portion contains the specified array. The sorted portion is positioned to the left,

and the unsorted portion is positioned to the right. In a selection sort, the unsorted array's first element is chosen from its smallest elements and given the top spot. The second-smallest piece is then chosen and put in the second slot after that. The procedure is repeated until the full array has been sorted. Selection sort has an average and worst-case complexity of O(n2), where n is the total number of elements. This makes it unsuitable for huge data collections [6]–[8].

Typically, selection sort is used for:

1. Sorting a tiny array is required.
2. Cost of swapping is irrelevant.
3. Every component must be checked.
4. Let's now examine the selection sort algorithm.

Algorithm

SELECTION SORT (ARR, n)

1) Step 1: For i = 0 to n-1, repeat Steps 2 and 3.
2) Step 2: CALL SMALLEST(arr, i, n, pos)
3) Step 3: SWAP arr[i] with arr[pos]
4) [FINAL LOOP]
5) Step 4: EXIT

(arr, i, n, pos) SMALLEST

1. Step 1: [INITIATE] CONFIGURE SMALL = arr[i]
2. Step 2: [INITIATE] LIMIT pos to i
3. Step 3: Repeat if (SMALL > arr[j]) for j = i+1 to n.
4. CONFIGURE SMALL = arr[j]
5. [END OF IF] SET pos = j
6. [FINAL LOOP]
7. Step 4: RETURN the position
8. How the Selection Sort Algorithm Works [9], [10]
9. Let's now examine how the Selection sort Algorithm functions.

CONCLUSION

In conclusion, the performance of several computer science sorting algorithms as well as code samples in various languages were covered in this book. Sorting is the process of moving data into a predetermined order, such as ascending or descending. As long as it is a linear, or complete, ordering meaning that any two items may be ordered the precise order doesn't actually important.Choosing the best sorting algorithm is typically determined only by efficiency; for example, merge sort is always preferred over shell sort.

But there are also other elements to consider, and they are dependent on the execution. Recursion is a highly elegant way to define some algorithms, but these algorithms should still be quite efficient; for example, using recursion to construct a linear, quadratic, or slower algorithm would be a very terrible choice.

REFERENCES:

[1] R. Mavrevski, M. Traykov, and I. Trenchev, "Interactive approach to learning of sorting algorithms," *Int. J. online Biomed. Eng.*, 2019, doi: 10.3991/ijoe.v15i08.10530.

[2] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, 1992, doi: 10.1145/146370.146381.

[3] F. Gebali, M. Taher, A. M. Zaki, M. Watheq El-Kharashi, and A. Tawfik, "Parallel Multidimensional Lookahead Sorting Algorithm," *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2920917.

[4] A. A. Nasar, "A Mathematical Analysis of student-generated sorting algorithms," *Math. Enthus.*, 2019, doi: 10.54870/1551-3440.1460.

[5] Y. Ben Jmaa, R. Ben Atitallah, D. Duvivier, and M. Ben Jemaa, "A comparative study of sorting algorithms with FPGA acceleration by high level synthesis," *Comput. y Sist.*, 2019, doi: 10.13053/CyS-23-1-2999.

[6] H. Wang *et al.*, "PMS-sorting: A new sorting algorithm based on similarity," *Comput. Mater. Contin.*, 2019, doi: 10.32604/cmc.2019.04628.

[7] H. M. Walker, "Sorting algorithms," *ACM Inroads*, 2015, doi: 10.1145/2727125.

[8] A. DEV MISHRA and D. GARG, "Selection of Best Sorting Algorithm," *Int. J. Intell. Inf. Process.*, 2008.

[9] J. Sukiban *et al.*, "Evaluation of Spike Sorting Algorithms: Application to Human Subthalamic Nucleus Recordings and Simulations," *Neuroscience*, 2019, doi: 10.1016/j.neuroscience.2019.07.005.

[10] S. Thengumpallil, J. F. Germond, J. Bourhis, F. Bochud, and R. Moeckli, "Impact of respiratory-correlated CT sorting algorithms on the choice of margin definition for free-breathing lung radiotherapy treatments4DCT sorting algorithms in margin definition," *Radiother. Oncol.*, 2016, doi: 10.1016/j.radonc.2016.03.015.

CHAPTER 15

A BRIEF DISCUSSION ON SEARCHING ALGORITHMS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology,
Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

One or more elements from a dataset can be found using the searching techniques. These kinds of algorithms are employed to locate components inside certain data structures. Searching may or may not be consecutive. If the dataset's data are random, sequential searching is required. If not, we can apply various approaches to simplify the situation. In order to find an element or get it from any data structure where it is stored, searching algorithms are created**.**

KEYWORDS:

Linear Search, Binary Search, Hash Table Search, Tree-based Search, Search Efficiency.

INTRODUCTION

Sequential Search

a straightforward algorithm that looks for a certain item within a list. It works by repeatedly iterating through each element in order to find matches or reach the finish.

Algorithm

1) Step 1: SequentialSearch(list, item)
2) Step 2: Before: 6 =
3) Step 3: Post: if item index is discovered,
4) Step 4: return it; if not, return -1
5) Step 5: index = 0;
6) while index = list;.
7) Step 6: List and count [index] 6 = item
8) index + index plus one
9) If index > list, then
10) Step 7: end while
11) Step 8: List[index] = item and count
12) Step 9: Return Index
13) Step 10: If Return 1, end
14) Step 11: end SequentialSearch.

Probability Search

A statistical sequential searching algorithm is probability search. Along with looking for a particular item, it also accounts for its frequency by switching it out for its predecessor in the list. Although the non-uniform items search still has an O(n) algorithm complexity, list scanning time is reduced since the more common items are in the initial slots [1]–[3].

Notice how the searched items' search probabilities have grown after each search operation in which depicts the list's final state after finding two items.

1) algorithm ProbabilitySearch(list, item)

2) Step 1: Pre: list 6= ∅

3) Step 2: Post: a boolean indicating where the item is found or not;

in the former case swap founded item with its predecessor

4) Step 3: index ← 0

5) Step 4: while index < list.Count and list[index] = item

6) Step 5: index ← index + 1

7) Step 6: end while

8) Step 7: if index ≥ list.Count or list[index] = item

9) Step 8: return false

10)Step 9: end if

11)Step 10: if index > 0

12)Step 11: Swap(list[index], list[index − 1])

13)Step 12: end if

14)Step 13: return true

15)Step 14: end ProbabilitySearch.

DISCUSSION

Binary Search

We may use the binary search approach to find things on the list after it has been sorted. The full list is split into two sub-lists in this method. If the item is located in the center position, the location is returned; if not, the procedure moves to the left or right sub-list and is repeated until the item is located or the range has been exceeded [4], [5].

Binary Search Technique complexity

1. O(1) is the best estimate for time complexity. The worst-case scenario is O(log2 n).
2. O(1) algorithm binarySearch (array, start, end, key) has a space complexity of 1.
3. The search key, a start and finish location, and a sorted array are the inputs.
4. If the key is present, output the location; else, erroneous location.

Algorithm

1) Step 1: Start the expression
2) Step 2: if start = end,
3) Step 3: then mid:= start + (end - start) /2.

4) Step 4: If array[mid] = key, then return the midpoint.
5) Step 5: When array[mid] is greater than key,
6) Step 6: call binarySearch(array, mid+1, end, key);
7) Step 7: else,
8) Step 8: call binarySearch(array, start, mid-1, key);
9) Step 9: else, return invalid location.
10) Step 10: Resolve linear search

Linear Search:

The most basic method of searching is linear. This method involves individually searching each item. This process may be used with a collection of unsorted data as well.

Sequential search is another name for linear search. Because of its n O(n) order n O(n) temporal complexity, it is known as linear.

Linear Search Technique complexity

O(n) Time Complexity

Complexity of Space: O(1)

Algorithm

1) Step 1: array, size, and key; linear Search
2) Step 2: An input consisting of a sorted array, the array's size, and the search key
3) Step 3: If the key is present,
4) Step 4: output the location;
5) Step 5: else, erroneous location.
6) Step 6: Start with for i:= 0 to size -1,
7) Step 7: then do if array[i] equals key,
8) Step 8: then return i,
9) Step 9: done returning invalid location.
10) Step 10: End.
11) Step 11: Search Exponentially

Exponential Search:

Other names for exponential search include doubling and galloping search. This method is used to determine the possible range for the search key. L and U are both powers of two if L and U are the list's upper and lower bounds, respectively.

The U is listed at the very end of the final section. It is referred to as exponential because of this. It employs the binary search method to pinpoint the precise location of the search key after determining the precise range [6]–[8].

Exponential Search Technique's complexity

O(1) is the best estimate for time complexity. O(log2 i) in the average or worst case scenario. where the search key is located at position i.O(1)

Algorithm

binarySearch (array, start, end, key) has a space complexity of 1.

the search key, a start and stop location, and a sorted array as input

A. If the key is present,
B. output the location;
C. else, erroneous location.

1) Step 1: Start the expression if start = end,
2) Step 2: then mid:= start + (end - start) /2.
3) Step 3: If array[mid] = key,
4) Step 4: then return the midpoint.
5) Step 5: When array[mid] is greater than key,
6) Step 6: call binarySearch(array, mid+1, end, key);
7) Step 7: else, call binarySearch(array, start, mid-1, key);
8) Step 8: else,
9) Step 9: return invalid location.
10) Step 10: End.
11) Step 11: Toggle Search

Jump search

For ordered lists, the jump search strategy also works. It makes a block and looks inside of it for the element. If the item is not present in the block, the block as a whole is moved. The list size determines the block size. The block size will be n if the list size is n. It uses a linear search approach to locate the object after locating a proper block. According to its performance, the jump search falls between linear search and binary search [9], [10].

Jump Search Technique complexity

O(n) Time Complexity

O(1) Space Complexity jumpSearch(array, size, key)

An ordered array, its size, and the search key are provided as input.

If the key is present, output the location; else, erroneous location.

Algorithm

1) Step 1: size start:= 0
2) Step 2: end:= block
3) Step 3: blockSize:=Size
4) Step 4: when array[end] equals key AND
5) Step 5: end size when start == end
6) Step 6: end == end + blockSize
7) Step 7: if end is more
8) Step 8: If array[i] has the key,
9) Step 9: then return i after doing if size - 1
10) Step 10: then end:= size done for i:= start to

11) Step 11: end -1 do
12) Step 12: if End
13) Step 13: return invalid location
14) Step 14: End

CONCLUSION

In conclusion, We have introduced a few cutting-edge searching algorithms in this chapter. Avl and BST trees employ the logarithmic searching technique, which we have previously described as being more effective. We choose not to discuss a search method called binary chop (sometimes known as binary search; often used to refer to its array counterpart). The underlying data structure being utilized to store the data has a significant impact on the effectiveness of search algorithms. For instance, searching a BST takes less time than searching a linked list, and determining if an item is in a hash table takes less time than determining whether it is in an array. We highly suggest you to sit down and understand the data structures accessible to you if you plan to often search for data. Using a list or any other purely linear data structure is typically a sign of ignorance. Research the data structures that best suit your circumstance after modeling your data.

REFERENCES:

[1] S. Li, Y. Wang, W. Wu, and Y. Liang, "Predictive searching algorithm for Fourier ptychography," *J. Opt. (United Kingdom)*, 2017, doi: 10.1088/2040-8986/aa95d5.

[2] B. Subbarayudu, L. Lalitha Gayatri, P. Sai Nidhi, P. Ramesh, R. Gangadhar Reddy, and C. Kishor Kumar Reddy, "Comparative analysis on sorting and searching algorithms," *Int. J. Civ. Eng. Technol.*, 2017.

[3] H. Zhang and Q. Hui, "Many objective cooperative bat searching algorithm," *Appl. Soft Comput. J.*, 2019, doi: 10.1016/j.asoc.2019.01.033.

[4] Y. Liang, X. Da, J. Wu, R. Xu, Z. Zhang, and H. Liu, "WFRFT modulation recognition based on HOC and optimal order searching algorithm," *J. Syst. Eng. Electron.*, 2018, doi: 10.21629/JSEE.2018.03.03.

[5] C. Zalka, "Grover's quantum searching algorithm is optimal," *Phys. Rev. A - At. Mol. Opt. Phys.*, 1999, doi: 10.1103/PhysRevA.60.2746.

[6] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Commun. ACM*, 1977, doi: 10.1145/359842.359859.

[7] E. Asenova, H. Y. Lin, E. Fu, D. V. Nicolau, and D. V. Nicolau, "Optimal fungal space searching algorithms," *IEEE Trans. Nanobioscience*, 2016, doi: 10.1109/TNB.2016.2567098.

[8] A. Niewola and L. Podsedkowski, "L* Algorithm—A Linear Computational Complexity Graph Searching Algorithm for Path Planning," *J. Intell. Robot. Syst. Theory Appl.*, 2018, doi: 10.1007/s10846-017-0748-6.

[9] D. Hu, T. Long, Y. Xiao, X. Han, and Y. Gu, "Fluid-structure interaction analysis by coupled FE-SPH model based on a novel searching algorithm," *Comput. Methods Appl. Mech. Eng.*, 2014, doi: 10.1016/j.cma.2014.04.001.

[10] Y. Wang, G. Kora, B. P. Bowen, and C. Pan, "MIDAS: A database-searching algorithm for metabolite identification in metabolomics," *Anal. Chem.*, 2014, doi: 10.1021/ac5014783.

CHAPTER 16

A BRIEF DISCUSSION ON NUMERIC

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

Data that is expressed as numbers rather than in any linguistic or descriptive form is referred to as numerical data. Numerical data, also known as quantitative data, is gathered in number form and differs from all other sorts of number data because it can be computed mathematically and statistically.

KEYWORDS:

Arithmetic Algorithms, Floating-Point Numbers, Integer Operations,Numeric Precision, Number Theory.

INTRODUCTION

Primality Test:

Using a straightforward technique, it is possible to establish if a given integer, is a prime number. For example, the numbers 2, 5, 7, and 13 are all prime numbers, however the number 6 is not since it can come from the product of two less-than-6-digit integers. The upper bound is set to the value of n in an effort to slow down the inner loop. A technique called the primality test may be used to determine if a positive integer is a prime number or not. There are several ways to determine if a number is a prime number, including the Miller-Rabin and Fermat primality tests, the School Method, the Trial Division Method, and the Fermat primality test [1]–[3].

Algorithm

1) Step 1:The IsPrime(n)
2) Step 2: Post: It is determined whether or not n is a prime.
3) Step 3: If i j = n, then
4) Step 4: If j 1 to sqrt(n), then
5) Step 5: If i j = n
6) Step 6: Send back fake end if
7) Step 7: end for
8) Step 8: end for
9) Step 9: end if
10) Step 10: end the Prime

Conversions Of Bases:

Several methods in DSA can convert a base-10 integer into its equivalent in binary, octal, or hexadecimal form. For instance, the binary equivalent of 7810 is 10011102.

Algorithm

1) ToBinary(n),
2) Step 1: Pre: n ≥ 0
3) Step 2: Post: n is now represented as a base-2 number.
4) Step 3: while n > 0,
5) Step 4: Add(n% 2)
6) Step 5: n ← n/2
7) Step 6: End while
8) Step 7: and return Reverse(list).
9) Step 8: finish the ToBinary n list

Different number systems, such as Decimal, Hexadecimal, Octal, Binary, or even Duodecimal, or the less well-known but more appropriate moniker Uncial, may be used in electronic and digital systems. Other than decimal, all other bases are produced by computers.

The base-12 equivalent of Decimal, called Uncial (from the Latin word for 1/10, "decima"), gets its name from the word "uncia" meaning one-twelfth). N can be expressed as follows in base or radix b:

(N)b = dn-1, dn-2, d1, d0. d-1 d-2 -- -- -- -- d-m

In the example above, the integer portion is from dn-1 to d0, followed by a radix point, and the fractional portion is from d-1 to d-m.

Most significant bit (MSB) = dn-1

Least Significant Bit (LSB) = d-m

The following examples:

1. **Decimal to Binary**

(10.25)10

Continue multiplying the fractional component by 2 until you get the decimal part 0.00 [4], [5].

(0.25)10 = (0.01)2

The formula is (10.25)10 = (1010.01)2

10.01

2. Binary to Decimal

8+0+2+0+0+0.25 = 10.25 (1010.01) is the result of 1x23 + 0x22 + 1x21 + 0x20 + 0x2 -1 + 1x2 -2.2 = (10.25)10

3. Octal to Decimal

(10.25)10 (10)10 = (12)8

Part in fractions: 0.25 x 8 = 2.00

Reminder: Continue multiplying the fractional component by 8 until you reach the decimal part.00.

(.25)10 = (.2)8

Answer: (12.2)8 + 10.25 =

4. Decimal to Octal

(12.2)8 1 x 81 + 2 x 80 +2 x 8-1 = 8+2+0.25 = 10.25 (12.2)8 = (10.25)10

5. Binary to Hexadecimal

Write the 4-bit binary equivalent of hexadecimal to convert it to binary.

(3A)16 = (00111010)2

6. Hexadecimal to Binary

Writing the corresponding hexadecimal for the 4-bit binary requires arranging the bits in groups of four starting from the right end. To reposition the groupings, add more 0s to the left.

1111011011 0011 1101 1011 (001111011011)2 = (3DB)16

7.Binary to Octal

Write the corresponding octal value for the 3-bit binary value after grouping the bits into groups of 3 from the right end.

binary. To change the groupings, add 0s to the left.

Example:

111101101

111 101 101

(111101101)2 = (755)8

DISCUSSION

Getting the biggest common factor of two numbers:

Finding the greatest common denominator of two integers is a very typical issue in mathematics. What we are essentially looking for is the largest number that is a multiple of both, for example, the greatest common denominator of 9 and 15 is 3. Based on Euclid's technique, one of the most elegant solutions to this issue has a run-time complexity of O(n 2) [6], [7].

Calculating the highest possible value for a number in a base with N digits:

This procedure determines the highest possible value of a number for a particular set of digits, for example, using the base-10 system, the highest possible number we may have is 999910. Similarly, the largest base-2 number that may include 4 digits is 11112, or 1510. This maximum value for N digits may be calculated using the formula: BN 1. B is the number base and N is the number of digits in the previous phrase. For instance, if we wanted to get the highest value for a six-digit, base-16 hexadecimal integer, the equation would be: 166 1. The preceding example's

highest value might be expressed as F F F F F F16, which results in 1677721510. The following procedure should be understood to limit numberBase to the numbers 2, 8, 9, and 16. Due to this, numberBase has an enumeration type in our real implementation.

Base enumeration type is described as follows:

Base is equal to "Binary 2, Octal 8, Decimal 10, and Hexadecimal 16."

Instead of utilizing multiple tests to find the best basis to utilize, we present the definition of basis to give you an idea of how this method may be modelled in a more legible way. For the sake of our implementation, we convert numberBase's value to an integer, and as a result, we extract the value related to the appropriate choice from the Base enumeration. As an illustration, the value 8 would result from casting the option "Octal" to an integer. The cast is implied in the procedure below, so we just use the real input numberBase.

Algorithm:

1) Step 1: MaxValue algorithm (numberBase, n)
2) Step 2: Pre: The number system to use is numberBase, and n is the number of digits.
3) Step 3: Post: The numberBase's maximum value with n digits is calculated.
4) Step 4: Return Power(numberBase, n) 1
5) Step 5: end MaxValue

Number's factororial

A simple mathematical procedure is finding a number's factorial. Since the issue is recursive in nature, many factorial method solutions are also recursive; nevertheless, in this case, we provide an iterative approach. Given that it is also simple to construct and does not suffer from the use of recursion (for more information on recursion, see C), the iterative approach is offered [8]–[10].

Factorial of 0 and 1 equals zero. The aforementioned serves as a foundation upon which we shall build. The factorial of two is two times the factorial of one, and the factorial of three is three times the factorial of two, and so on. When pursuing the factorial of a number, we may use the notation N!, where N stands for the number we want to factorialize. Although our algorithm doesn't utilize this notation, knowing it is helpful.

Algorithm:

1) Step 1: Factorial(n)
2) Step 2: Pre: n 0, n is the number to use to calculate the factorial of
3) Step 3: Post: n's factorial is calculated
4) Step 4: if n < 2
5) Step 5: return 1;
6) Step 6: end if;
7) Step 7: fibonacci 1 ;
8) Step 8: for i 2 to n
9) Step 9: "f actorial" "f actorial"
10) Step 10: "i," "end for,"
11) Step 11: "return f actorial," etc.
12) Step 12: end Factorial

CONCLUSION

Numerous numerical algorithms have been provided in this chapter, the most of which are only there because their creation was enjoyable. The reader's takeaway from this chapter may be that many domains can use algorithms to make work in that particular area possible.

Some of the most sophisticated systems on the globe compute data like weather forecasts using numerical algorithms, in particular. The numerous numeric data types utilized in SQL are the main topic of this chapter.

In SQL, numbers can be either accurate or approximative. A precise numerical value has a scale and a precision. The precision, which is a positive integer, specifies how many significant digits there are in a given radix.

Exact numerical types include NUMERIC, DECIMAL, INTEGER, BIGINT, and SMALLINT. Although an integer's scale is zero, its syntax only employs the term INTEGER or the acronym INT.

Although SMALLINT has a scale of zero, the implementation allows for a range of values that is either smaller than or equal to that of INTEGER. Similar to INTEGER, BIGINT also has a scale of 0, but its range of possible values is bigger than or equal to that of INTEGER in the implementation. A TINYINT precise numeric type with a range of 0 to 255 may also be available in SQL. For each numeric type, the minimum and maximum values are both larger than zero.

Since host languages do not allow NULLs, the programmer must either substitute missing values with those that can be expressed in the host language or utilize INDICATOR variables to instruct the host program to take specific action for them. The NULL in SQL is the only mechanism to handle missing data.

REFERENCES:

[1] K. Moon *et al.*, “Expanding the role of social science in conservation through an engagement with philosophy, methodology, and methods,” *Methods Ecol. Evol.*, 2019, doi: 10.1111/2041-210X.13126.

[2] C. Le Cornec *et al.*, “Is intravenously administered, subdissociative-dose KETAmine non-inferior to MORPHine for prehospital analgesia (the KETAMORPH study): Study protocol for a randomized controlled trial,” *Trials*, 2018, doi: 10.1186/s13063-018-2634-3.

[3] S. C. Bailey *et al.*, “Expanding the Universal Medication Schedule: A patient-centred approach,” *BMJ Open*, 2014, doi: 10.1136/bmjopen-2013-003699.

[4] J. C. Austin, C. Hippman, and W. G. Honer, “Descriptive and numeric estimation of risk for psychotic disorders among affected individuals and relatives: Implications for clinical practice,” *Psychiatry Res.*, 2012, doi: 10.1016/j.psychres.2012.02.005.

[5] A. Louw, K. Zimney, E. A. Johnson, C. Kraemer, J. Fesler, and T. Burcham, “De-educate to re-educate: aging and low back pain,” *Aging Clin. Exp. Res.*, 2017, doi: 10.1007/s40520-017-0731-x.

[6] D. P. Adams, “Reactive multilayers fabricated by vapor deposition: A critical review,” *Thin Solid Films*. 2015. doi: 10.1016/j.tsf.2014.09.042.

[7] R. Meister *et al.*, "Myocardial Infarction - Stress PRevention INTervention (MI-SPRINT) to reduce the incidence of posttraumatic stress after acute myocardial infarction through trauma-focused psychological counseling: Study protocol for a randomized controlled trial," *Trials*, 2013, doi: 10.1186/1745-6215-14-329.

[8] "Dyadic Pain Management Program for Older Adults and Informal Caregivers With Chronic Pain," *Case Med. Res.*, 2019, doi: 10.31525/ct1-nct04106271.

[9] P. Macaruso, "Developmental Dyscalculia and Cognitive Neuropsychology," *Dev. Neuropsychol.*, 1994, doi: 10.1080/87565649409540593.

[10] C. Prati *et al.*, "Effects of endurance training on quality of life in patients with amyotrophic lateral sclerosis," *Amyotroph. Lateral Scler. Front. Degener.*, 2013.

CHAPTER 17

A BRIEF DISCUSSION ON STRINGS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

Whether as a literal constant or as some other variable, a string is often a collection of computer science instructions. The latter could be able to modify the duration and evolve its components, or it might be fixed (after formation). Most people agree that a string qualifies as an information form. In order to maintain a number of components, often letters, it is frequently expressed as a byte (or term) range information structure. Other kinds and configurations of sequence (or array) information can also be designated by the sequence, such as more generalized arrays. A variable is described as a sequence that, depending on the programming language and appropriate information type utilized, may either cause immediate space to be permanently allocated to a set maximum size or implement an active dynamic allocation to include a dynamic number of elements.A sequence is categorized as a particular string or an invisible string depending on whether it appears explicitly in the code. A string is a finite sequence of symbols chosen from an alphabet-like structure used in mathematical expressions and conceptual computer science.

KEYWORDS:

String Manipulation, String Matching, Substrings, String Compression, Text Algorithms.

INTRODUCTION

Considering how frequently programs employ string operations and transformations, strings receive their own chapter in this book. The provided algorithms were either created to pique the authors' curiosity or are based on issues they have already encountered. Strings are often expressed as arrays of bytes (or words) that store a series of characters and are regarded as a general-purpose data type. The definition of a string is a collection of characters. A string differs from a character array in that it is finished with the unique character "0."

The foundation of computer languages and the building blocks of communication are the String data format. One of the most fundamental and often used tools in computer science and programming is the string data structure. They make it possible to represent and manipulate text and character sequences in a number of different ways.

Large volumes of text data, ranging from basic strings to complicated words, paragraphs, and even whole books, may be stored and processed using the string data structure, which is a potent tool. This string of letters stands in for text or other types of data. It is a basic data structure that is used to store and modify text-based data in various computer languages. Strings are typically implemented as an array of characters in computer languages, with each character having a specific index location inside the array [1]–[3].

String's use cases:

Plagiarism Checker: By employing string matching techniques, strings may be utilized to quickly detect plagiarism in programs and materials. Using this, the computer could quickly inform us of the proportion of code and text that any two users' contributions match.

Encoding/Decoding (Cipher Text Generation): Strings can be used for encoding and decoding in order to ensure that no one standing in the path of the transmission of your data is able to read it and launch active or passive assaults. Your messaged text is encrypted at the sender's end and decrypted at the receiver's end.

Information Retrieval: String applications and the matching/retrieval module for strings enable us to obtain information from unidentified data sources (huge datasets used as input).

Better Filters For The Approximate Suffix-Prefix Overlap Problem: Strings and its applications to algorithms enable us to offer better filters for the approximate suffix-prefix overlap problem. Finding all pairings of strings from a given set such that a prefix of one string is similar to a suffix of the other is known as the approximation suffix-prefix overlap issue.

A. Data exchanged over networks, such HTTP requests and answers, may be encoded and decoded using strings.
B. Strings are used to read and write files, change file paths and names, and handle files.
C. **Data analysis:** Natural language processing and sentiment analysis may both be done on massive volumes of text data using strings.

Real-Time String Application:

Spam detection: Since the idea of a string-matching method will be employed in this case, strings may be used as a spam detection system. Unwanted emails, or spam, may result in significant financial loss.

The idea of string matching is used by all spam filters to locate and eliminate spam. Strings are applicable to the science of bioinformatics (DNA sequencing). Finding patterns in DNA and resolving challenges with genetic sequences may both be done using the string matching module.

Strings are a possible component of intrusion detection systems. String matching methods are used to find packets that include intrusion-related phrases.

Lookup Tools: Strings are used in a variety of search engine strategies. The majority of the data is available online as textual data. It is really challenging to search for a certain piece of material since there is so much uncategorized text data available. String matching algorithms are used to categorize the material and organize it using web search engines.

String operations:

Users can perform a number of operations on strings. Among the significant ones are:

1) The length of the string may be determined using the size() method.
2) The function substr() is used to locate a substring of a specific length beginning at a specific index.
3) +: This operator joins two strings together.

4) S1 and S2 are compared using the function s1. compare(s2) to determine which string is lexicographically bigger and which is smaller.
5) Reversing a string is possible with the reverse() method.
6) Sorting a text in lexicographic order is done with the function sort().

Benefits of string:

Text Processing: In computer languages, text is represented by strings. They may be used to search, replace, parse, and format text, among other text manipulation and processing operations. Strings may be used to represent several forms of data, including numbers, dates, and times. For instance, a string can be used to represent a time in the format "HH:MM:SS" or a date in the format "YYYY-MM-DD". Strings are simple to use and to modify.

Among other things, they can be inverted, cut, and concatenated. Additionally, they offer an easy-to-understand syntax that makes them usable by programmers of all ability levels. Strings are a common data type in all computer languages due to their compatibility. As a result, strings are a dependable and effective way to communicate and share data since they can be readily moved between various systems and platforms.

Memory Efficiency: Strings are easy to allocate and deallocate since they are frequently kept in contiguous blocks of memory. Because they don't use much memory, they may be used to represent vast quantities of data.

Negative aspects of string:

Memory Usage: Working with large or numerous strings might cause strings to use up a lot of memory. In memory-constrained settings like embedded systems or mobile devices, this may be an issue.

Immutability: Strings can never be modified after they have been produced in many computer languages. This might cause inefficiencies and memory cost when working with big or complicated strings that call for frequent alterations.

Performance Overhead: When working with lengthy or intricate strings, string operations may take longer than operations on other data types. This is due to the fact that string operations frequently require copying and memory reallocation, both of which take time.

Overhead in Encoding and Decoding: Strings may use multiple character encodings, which might increase conversion time.

Working with data from several sources or dealing with systems that employ various encodings can be challenging because of this.

Security Flaws: If strings are not handled appropriately, they may be subject to security flaws such buffer overflows or injection attacks. This is because attackers may alter strings to run arbitrary code or access private information [4], [5].

DISCUSSION

Changing the word order in a sentence:

It is easy to define algorithms for basic string operations, such as extracting a sub-string of a string, however other methods call for more creative thinking can be little trickier. The technique shown here doesn't only flip the letters in a instead, it flips the order of the words within a string.

This program is based on the idea that white space between words serves as a word boundary. We only need a few markers to indicate the beginning and end of words so we may reverse them.

Algorithm:

1) ReverseWords (value)
2) Step 1: Pre: sb is a string buffer,
3) Step 2: value 6=Post: The words' values have been switched around.
4) Step 3: length of last value: 1
5) Step 4: begin end
6) Step 5: While final, 0
7) // skip blank lines
8) Step 6: with value[start] = whitespace and start 0
9) Step 7: Begin Begin 1
10) Step 8: Finish while
11) Step 9: final start
12) // move the word's start position down to the index
13) Step 10: while start 0 and start 6 equal whitespace
14) Step 11: begin; begin; and one
15) Step 12: finish though
16) // Add the characters from start + 1 to length + 1 to the string buffer sb.
17) Step 13: for i first + last
18) Step 14: sb.Append(value[i]),
19) Step 15: finish with
20) // Add some whitespace after the word in the buffer if this isn't the string's final word
21) Step 16: if start > 0, then
22) Step 17: (' ') sb.Append
23) Step 18: end if
24) Step 19: final start 1
25) Step 20: first to last
26) Step 21: finish though
27) // Determine whether we introduced too much whitespace to sb.
28) Step 22: If sb[sb.Length 1] = whitespace,
29) // remove any blank spaces
30) Step 23: sb.Length sb.Length 1 (30)
31) Step 24: end if
32) Step 25: send sb back
33) Step 26: ending ReverseWords

Finding a palindrome

Despite not being a frequently used algorithm in real-world situations. The technique for palindrome detection is entertaining and, as it turns out, rather simple design.The complexity of the method we provide in terms of execution time is O(n). Our approach places two points at the ends of the string that we are examining to see whether it is a palindrome. or not. These points advance near one another while constantly making sure that

Algorithm

1) IsPalindrome(value)
2) Step 1: Pre: value 6 equals
3) Step 2: Post: value's palindrome status is determined
4) Step 3: value.Strip().ToUpperCase(); word;
5) Step 4: left ← 0
6) Step 5: The proper term, length 1.
7) Step 6: while lef t right and word[lef t] = word[right]
8) Step 7: (Left Left + 1)
9) Step 8: Right Right Right 1
10) Step 9: Finish while
11) Step 10: Return word[lef t] = word[right].
12) Step 11: ending IsPalindrome

We use a technique called Strip in the IsPalindrome algorithm. This algorithm eliminates all punctuation, including white space, from the string. The outcome is Each word in word includes a highly compressed version of the original string.It is represented as an uppercase version of the character. White space, punctuation, and case are eliminated in palindromes. Allows us to create a robust algorithm while keeping our design simple. In terms of the palindromes it will find.

Calculating how many words are in a string:

Although counting the amount of words in a string may appear simple at first,

We need to be mindful of the following cases:

1. Monitoring our position in a string.
2. Updating the word count where it belongs
3. Omitting the blank spaces between the words

Three variables can be used to control the points listed:

1. Index
2. WordCount
3. With Word

Keeps track of the current index we are at in relation to the previously mentioned index. The string, wordCount is an integer that records how many words we've used and lastly, inW ord is a Boolean flag that indicates if or not [6]–[8].

We are currently within a word of whether that is the case. Unless we are already striking, If we are in a word and there is a white space, the reverse is true from the current index. Reaching a blank place.

Algorithm:

1) WordCount(value)
1) Step 1: Pre: value 6 equals
2) Step 2: In the post, the value's word count is calculated.
3) four) in Word true
4) Step 3: words or less
5) Step 4: index ← 0
6) // skip the first blank line
7) Step 5: While index value.Length 1 and value[index] = whitespace
8) Step 6: index = index plus one
9) Step 7: Finish while
10) // Was there only whitespace in the string?
11) Step 8: If value[index] = whitespace and index = value.Length
12) Step 9: give 0
13) Step 10: terminate if
14) Step 11: while index = length.Value
15) Step 12: if value[index] = blank
16) Skip all whitespace
17) Skip while index value.Length 1 and value[index] = whitespace
18) Step 13: index + 1 index
19) Step 14: finish though
20) (in Word false)
21) Step 15: WordCount WordCount + 1
22) Step 16: else
23) Step 17: (Word inW ord: true)
24) Step 18: end if
25) Step 19: Index Index + 1
26) Step 20: finish though
27) // Last word might not have had a space after it
28) Step 21: if inWord.
29) Step 22: wordCount wordCount + 1
30) Step 23: end if
31) Step 24: return wordCount
32) Step 25: end WordCount

Counting the amount of words that are repeated inside of a string:

With the use of an unordered collection and a word-splitting algorithm. This algorithm is simple to implement within a string using a designated delimiter Implement.

If all of the words were divided by a single white space, as our delimiter, we receive back all of the words in the string as components of a variety. Then, if we repeat these phrases, we may add them to a collection that includes only unique strings, we can determine how many distinct words there are from the string.

The final step is to deduct the overall word count from the unique word count. Number of stings in the array that the split operation returned [9], [10]. The

Algorithm:

1) RepeatedWordCount(value)
2) Step 1: Pre: value 6 equals
3) Step 2: Post: The value of the number of repeated words is returned.
4) Step 3: Words, split using "
5) Step 4: Uniques (5) Set
6) Step 5: Words for each word
7) Step 6: uniques.Add(word.Strip())
8) Step 7: finish each
9) Step 8: return unique words, length, and count
10) Step 9: End RepeatedWordCount at 10.

CONCLUSION

We hope the reader has now seen how entertaining algorithms for string data types may be. Since strings are likely to be the most prevalent data type (and data structure; keep in mind that we're working with an array), it's critical that you develop your creative thinking skills when working with them. Strings are intriguing, at least to us.

A quick Google search on string differences between languages and encodings will turn up a ton of issues. Now that we've given you some inspiration with our basic algorithms, you can come up with some of your own.

REFERENCES:

[1] D. Szklarczyk *et al.*, "STRING v11: Protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets," *Nucleic Acids Res.*, 2019, doi: 10.1093/nar/gky1131.

[2] F. F. Real, A. Batou, T. G. Ritto, and C. Desceliers, "Stochastic modeling for hysteretic bit–rock interaction of a drill string under torsional vibrations," *JVC/Journal Vib. Control*, 2019, doi: 10.1177/1077546319828245.

[3] S. I. Hakak, A. Kamsin, P. Shivakumara, G. A. Gilkar, W. Z. Khan, and M. Imran, "Exact String Matching Algorithms: Survey, Issues, and Future Research Directions," *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2914071.

[4] H. Tang, E. L. Day, M. B. Atkinson, and N. J. Pienta, "GrpString: An R package for analysis of groups of strings," *R J.*, 2018, doi: 10.32614/rj-2018-002.

[5] D. Szklarczyk *et al.*, "The STRING database in 2017: Quality-controlled protein-protein association networks, made broadly accessible," *Nucleic Acids Res.*, 2017, doi: 10.1093/nar/gkw937.

[6] R. A. Wagner and R. Lowrance, "An Extension of the String-to-String Correction Problem," *J. ACM*, 1975, doi: 10.1145/321879.321880.

[7] M. P. J. van der Loo, "The stringdist package for approximate string matching," *R J.*, 2014, doi: 10.32614/rj-2014-011.

[8] H. Wickham, "Stringr: Modern, consistent string processing," *R J.*, 2010, doi: 10.32614/rj-2010-012.

[9] R. A. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *J. ACM*, 1974, doi: 10.1145/321796.321811.

[10] H. Erbin, J. Maldacena, and D. Skliros, "Two-point string amplitudes," *J. High Energy Phys.*, 2019, doi: 10.1007/JHEP07(2019)139.

CHAPTER 18

A BRIEF DISCUSSION ON GREEDY ALGORITHM

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

An algorithmic paradigm known as a greedy algorithm adopts the problem-solving heuristic of selecting the locally best option at each stage in the hopes of discovering a global optimum. In other words, a greedy algorithm selects the best option available at each stage without taking into account how that selection may affect subsequent phases. When a problem can be broken down into smaller subproblems and the solutions to each subproblem can be integrated to solve the larger problem, the greedy method is helpful. The greedy method can be used to resolve optimization issues where the goal is to select the best answer from a large pool of alternatives.It's possible that not all issues can be solved using this approach. It does this because it constantly seeks to create the optimal outcome at the local level. But we can find out if the technique works for any issue.

KEYWORDS:

Activity Selection, Greedy Approach, Huffman Coding, Knapsack Problem, Optimal Choice.

INTRODUCTION

The "coin change" problem is a well-known illustration of a challenge that may be resolved by a greedy algorithm. Making change for a certain sum of money with the fewest number of coins is the challenge. The greedy algorithm, for instance, would select the largest coin at each step if the sum was 25 cents and the available coins were 1, 5, and 10 cents. For a total of three coins, it would first choose a 10-cent coin, then another 10-cent coin, and finally a 5-cent coin. The greedy algorithm might not always find the best answer, though. For instance, the greedy algorithm will choose one 4-cent coin and two 1-cent coins when the available coins are 1 cent, 3 cents, and 6 cents, although the ideal answer would be to utilize two 3-cent coins. Therefore, it's critical to establish the validity of a greedy algorithm and comprehend its constraints. Numerous fields, including scheduling, graph theory, and dynamic programming, can benefit from the greedy method. The definition of a greedy algorithm is a strategy for solving optimization problems by selecting actions that, regardless of the outcome, produce the most obvious and immediate advantage. It is effective in situations when minimization or maximizing yields the desired outcome [1]–[3].

Features of the greedy algorithm:

A issue must have a few key qualities in order to be solved using the Greedy method:

1. Resources are listed in order of profit, cost, worth, etc.
2. The maximum amount of each resource (maximum profit, maximum worth, etc.) is used.
3. For instance, the largest value or weight is taken first in the fractional knapsack issue based on the available capacity.

Application of the greedy algorithm:

The greedy algorithm is a technique for solving optimization issues where the objective is to select the option that is locally optimal at each stage in the pursuit of a global optimum. It is referred described as "greedy" because it strives to discover the optimum answer by selecting the best option at each stage, without taking into account the implications of the present choice or subsequent actions.

The greedy algorithm has several typical applications, including:

Scheduling and Resource Allocation: The greedy method may be used to effectively schedule tasks or distribute resources.

A graph's smallest spanning tree: the subgraph that links all vertices with the least amount of edge weight can be discovered using the greedy technique.

Coin Change Issue: By constantly selecting the coin with the greatest value that is less than the remaining amount to be changed, the greedy algorithm may be utilized to make change for a given amount with the fewest possible coins.

By building a binary tree so that the frequency of each character is taken into account, the greedy approach may be utilized to develop a prefix-free code for data compression.

It's crucial to remember that not every optimization problem can be resolved by a greedy algorithm, and there are some situations in which the greedy strategy might result in less-than-ideal solutions. However, the greedy method is a helpful tool for swiftly and effectively resolving optimization issues as it frequently offers a decent approximation to the ideal solution [4], [5].

Every greedy algorithm has the same fundamental framework:

1. Declare a result of zero.
2. We make a hasty decision to choose, and if it is practical, we add it to the outcome.
3. return the outcome.

A few sacrifices in the greedy technique may make it acceptable for optimization. To promptly arrive at the best possible solution is one important factor. If more activities can be completed in the same amount of time as the present activity in the activity selection issue (explained below), they can. There is no need to integrate all of the answers when you can partition a problem recursively depending on a condition. The "recursive division" stage in the activity selection issue is completed by scanning a list of items just once and taking into account specific activities.

Unfair algorithm Examples of well-known issues that may be resolved using the greedy techniqand demonstrate the optimal substructure property include:

1) **Job schedulability:** By ranking the tasks in decreasing order of profit, avariciously chose the jobs with the highest profit first. By selecting the project with the highest profit potential for each available time slot, this would ultimately serve to optimize the total profit.
2) **Prim's Minimum Spanning Tree Algorithm:**The spanning tree is empty in the beginning. Maintaining two sets of vertices is the goal. The first set comprises the vertices that are already part of the MST, whereas the second set contains the vertices that are not yet part of the MST. It selects the minimal weight edge from among all the edges

that join the two sets at each stage. It selects the edge and then adds the opposite endpoint to the set containing MST.
It can be challenging and occasionally even unsuccessful to choose to use the greedy strategy when no careful consideration has been given to the matter. Making the local best decision occasionally results in the loss of the global best choice.

DISCUSSION

Greedy algorithm, divide and conquer algorithm, and dynamic programming algorithm are three common algorithmic paradigms used to solve problems.

Here's a comparison among these algorithms:

Approach:

Greedy algorithm: Makes locally optimal choices at each step with the hope of finding a global optimum. Divide and conquer algorithm: Breaks down a problem into smaller subproblems, solves each subproblem recursively, and then combines the solutions to the subproblems to solve the original problem.

Dynamic programming algorithm: Solves subproblems recursively and stores their solutions to avoid repeated calculations.

Goal:

Greedy algorithm: Finds the best solution among a set of possible solutions.

Divide and conquer algorithm: Solves a problem by dividing it into smaller subproblems, solving each subproblem independently, and then combining the solutions to the subproblems to solve the original problem.

Dynamic programming algorithm: Solves a problem by breaking it down into smaller subproblems and solving each subproblem recursively [6]–[8].

Time complexity:

Greedy algorithm: O(nlogn) or O(n) depending on the problem. Divide and conquer algorithm: O(nlogn) or O(n^2) depending on the problem.

Dynamic programming algorithm: O(n^2) or O(n^3) depending on the problem .

Space complexity:

Greedy algorithm: O(1) or O(n) depending on the problem.

Divide and conquer algorithm: O(nlogn) or O(n^2) depending on the problem.

Dynamic programming algorithm: O(n^2) or O(n^3) depending on the problem.

Optimal solution: Greedy algorithm: May or may not provide the optimal solution.

Divide and conquer algorithm: May or may not provide the optimal solution.

Dynamic programming algorithm: Guarantees the optimal solution. Examples: Greedy algorithm: Huffman coding, Kruskal's algorithm, Dijkstra's algorithm, etc.

Divide and conquer algorithm: Merge sort, Quick sort, binary search, etc.

Dynamic programming algorithm: Fibonacci series, Longest common subsequence, Knapsack problem, etc. In summary, the main differences among these algorithms are their approach, goal, time and space complexity, and their ability to provide the optimal solution. Greedy algorithm and divide and conquer algorithm are generally faster and simpler, but may not always provide the optimal solution, while dynamic programming algorithm guarantees the optimal solution but is slower and more complex.

Greedy Algorithm: Greedy algorithm is defined as a method for solving optimization problems by taking decisions that result in the most evident and immediate benefit irrespective of the final outcome. It is a simple, intuitive algorithm that is used in optimization problems.

Divide and conquer Algorithm: Divide and conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy. A typical Divide and Conquer algorithm solve a problem using the following three steps [9], [10]:

Divide: This involves dividing the problem into smaller sub-problems.

Conquer: Solve sub-problems by calling recursively until solved.

Combine: Combine the sub-problems to get the final solution of the whole problem.

Dynamic Programming: Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has sometimes repeated calls for the same input states, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

CONCLUSION

Greedy algorithms are an easy technique to addressing optimization issues, providing a minimum or maximum result.This book provided several examples of greedy algorithms and the technique to tackle each problem. By knowing how a greedy algorithm problems works you may better comprehend dynamic programming.A greedy algorithm is a method of problem-solving that chooses the best choice at the time. It is unconcerned with whether the most excellent outcome at the moment will produce the final best result. Even if the option was the erroneous one, the algorithm never goes back and changes its mind. It functions in a top-down manner.

REFERENCES:

[1] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller, “A greedy algorithm for aligning DNA sequences,” *Journal of Computational Biology*. 2000. doi: 10.1089/10665270050081478.

[2] S. A. Curtis, “The classification of greedy algorithms,” *Sci. Comput. Program.*, 2003, doi: 10.1016/j.scico.2003.09.001.

[3] J. Bang-Jensen, G. Gutin, and A. Yeo, “When the greedy algorithm fails,” *Discret. Optim.*, 2004, doi: 10.1016/j.disopt.2004.03.007.

[4] M. Steel, “Phylogenetic diversity and the greedy algorithm,” *Syst. Biol.*, 2005, doi: 10.1080/10635150590947023.

[5] A. Jain, M. Saini, and M. Kumar, "Greedy Algorithm," *J. Adv. Res. Comput. Sci. Eng. (ISSN 2456-3552)*, 2015, doi: 10.53555/nncse.v2i4.451.

[6] A. Vince, "A framework for the greedy algorithm," *Discret. Appl. Math.*, 2002, doi: 10.1016/S0166-218X(01)00362-6.

[7] V. N. Temlyakov and P. Zheltov, "On performance of greedy algorithms," *J. Approx. Theory*, 2011, doi: 10.1016/j.jat.2011.03.009.

[8] A. Buffa, Y. Maday, A. T. Patera, C. Prud'Homme, and G. Turinici, "A priori convergence of the Greedy algorithm for the parametrized reduced basis method," *ESAIM Math. Model. Numer. Anal.*, 2012, doi: 10.1051/m2an/2011056.

[9] J. Zhou, X. Zhao, X. Zhang, D. Zhao, and H. Li, "Task allocation for multi-agent systems based on distributed many-objective evolutionary algorithm and greedy algorithm," *IEEE Access*, 2020, doi: 10.1109/ACCESS.2020.2967061.

[10] A. V. Dereventsov and V. N. Temlyakov, "A unified way of analyzing some greedy algorithms," *J. Funct. Anal.*, 2019, doi: 10.1016/j.jfa.2019.108286.

CHAPTER 19

A BRIEF DISCUSSION ON DYNAMIC PROGRAMMING

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

Dynamic programming is a technique that divides the issues into smaller ones and saves the answer for use at a later time without having to recalculate it. The optimum substructure property refers to the optimization of the subproblems in order to maximize the overall solution. Dynamic programming is mostly used to address optimization issues. In this context, optimization challenges refer to situations where we are attempting to determine the minimal or maximum solution to a problem. If a solution does exist, dynamic programming ensures that the best solution will be found. According to the definition of dynamic programming, it is a method for solving difficult problems by first decomposing them into a number of simpler subproblems, solving each subproblem only once, and then storing the answers to prevent having to do the same calculations repeatedly.

KEYWORDS:

Coin Change Problem, Longest Common Subsequence, Memoization, Optimal Substructure, Overlapping Subproblems.

INTRODUCTION

The main benefit of dynamic programming is an improvement over simple recursion. Whenever we observe a recursive solution with repeated calls for the same inputs, we can use dynamic programming to improve it. The goal is to just save the solutions to the subproblems so that we won't have to recompute them in the future. The time complexity is decreased from exponential to polynomial with this straightforward improvement.

For instance, writing a straightforward recursive solution for Fibonacci numbers results in exponential time complexity; however, if this solution is optimized by storing the answers to its subproblems, the time complexity is reduced to linear [1]–[3].

Algorithm for dynamic programming features:

1. Dynamic programming (DP) is one of the most effective methods for tackling a certain class of issues.
2. The technique can be expressed elegantly, the thought process is relatively straightforward, and the coding is simple.
3. In essence, it is a straightforward concept: save the solution you come up with after working through a problem using the provided input as a reference for later use. (Remember your Past.
4. If the provided problem can be divided into smaller subproblems, and those smaller subproblems can be further divided into even smaller ones, and throughout this process, you observe some overlapping subproblems, that is a significant clue for DP.

5. The best solution to the main problem is also influenced by the best answers to the subproblems, which is known as the Optimal Substructure Property.
6. To prevent unnecessary calculation, the answers to the subproblems are either bottom-up (tabulation) or memoized in a table or array.
7. The answers to the subproblems can be used to build the solution to the main issue.

Using an iterative algorithm, which finds solutions by going through the subproblems in a particular order, or a recursive algorithm, which finds solutions to subproblems recursively, are two ways to implement dynamic programming.

The concepts that govern dynamic programming are as follows:

1. Create a mathematical model of the solution by describing the structure of the ideal solution.
2. Define the value of the ideal response recursively.
3. Calculate the value of the best solution for each potential subproblem using a bottom-up technique.
4. Using the data computed in the preceding phase, create an ideal solution to the original problem.

Applications:

Problems involving optimization are resolved using dynamic programming. Many issues in daily life are resolved using it, including, Making a modification issue, the rucksack issue, best-practice binary search tree.

Recursion is when a function is called and executed on its own. In dynamic programming, problems are handled by breaking them down into smaller ones to solve the larger ones. Although recursion techniques are not required for dynamic programming to work, since the goal of dynamic programming is to optimize and speed up the process, programmers typically utilize recursion techniques to speed up and turn the process efficiently. Get the name of the recursive function when a function has the ability to carry out a particular task by calling itself. This function calls itself when it needs to be executed in order to carry out and complete the work. In order to solve an issue, you can divide it into smaller components known as subproblems using dynamic programming.

In dynamic programming, the problem is initially solved, and the solutions are then memorized. Recursion is used to automate a function, but dynamic programming is an optimization approach used to solve issues, hence this is the major distinction between the two techniques. When necessary, recursive functions execute by themselves before ceasing to operate. The function executes itself by calling itself when it recognizes when it is required; this is known as a recursive situation.

As a result, in the base case, the function must terminate when the work is finished. Dynamic programming recognizes the issue and breaks it down into smaller issues in order to address the entire scene. A mathematical relationship between these subproblems or variables must be established by the programmer after they have been resolved. Not to mention, these answers and outcomes are saved as algorithms so they may be retrieved later on without having to redo the entire problem [4], [5].

Methods for resolving issues with dynamic programming

To start, memoization:

To start solving the given problem, break it down. Return the saved solution if you observe that the issue has already been resolved. Solve it and save it if it hasn't already. This is known as memorization, and it is typically simple to think of and quite intuitive.

Bottom-Up (Dynamic Programming): Start with the simplest subproblem and work your way up to the given problem by analyzing the problem and observing the order in which the subproblems are resolved. This procedure makes sure that the little issues are resolved prior to the major issues. Dynamic programming is the name for this.

Memorization vs. Tabulation (Dynamic Programming):

There are two distinct methods for storing the values so that they can be applied to another sub-problem. I'll cover two approaches to solve dynamic programming (DP) issues here:

Summary: Bottom Up

Keep in mind: Top Down

DISCUSSION

Using dynamic programming approach:

The steps that dynamic programming takes are as follows:

1. It divides the challenging issue into more manageable issues.
2. It resolves these related issues in the best possible way.
3. It memorizes the solutions to subproblems. Memorization is the process of retaining the answers to subproblems.
4. The identical sub-problem is calculated more than once as a result of their reuse.
5. Calculate the outcome of the challenging problem last.

The fundamental steps for dynamic programming are the five listed above. Dynamic programming is applicable to things with characteristics like:

Those issues when the subproblems overlap and the substructures are ideal. In this context, optimum substructure denotes the fact that it is possible to solve optimization issues by simply integrating the best solutions to each of their constituent subproblems. As we save the intermediate outcomes in the case of dynamic programming, the space complexity would grow, but the time complexity would decrease.

Dynamics of programming strategies:

Two methods exist for dynamic programming:

1. Top-down strategy
2. Bottom-up strategy

While the bottom-up strategy uses the tabulation method, the top-down approach uses memorization. Recursion plus caching in this case equals memorizing. While caching entails keeping the interim results, recursion entails calling the method directly.

Advantages

1. It is really simple to comprehend and put into practice.
2. The subproblems are only resolved when necessary.
3. It's simple to debug.

Disadvantages

1. It makes use of the recursion approach, which utilizes additional call stack memory. Occasionally, the stack overflow issue will occur due to excessive recursion.
2. It uses up more memory, which lowers performance as a whole.

Let's use an example to illustrate dynamic programming.

```
1) int fib(int n)
2) if(n0) error;
3) if(n==0)
4) return 0;
5) if(n==1)
6) return 1;
7) sum = fib(n-1) + fib(n-2);
```

The recursive method is what we used in the code above to determine the Fibonacci series. The number of function calls and computations will both increase as the value of 'n' rises. The temporal complexity in this situation rises exponentially to 2n. The use of dynamic programming is one technique to solving this issue. We can reuse the previously calculated value rather than repeatedly creating the recursive tree. The time complexity would be O(n) if we used the dynamic programming methodology.

The code would seem as follows when the Fibonacci series is implemented using dynamic programming:

```
1) If memo[n]!= NULL,
2) return memo[n];
3) count++;
4) if(n==0)
5) return 0;
6) if(n==1)
7) return 1;
8) sum = fib(n-1) + fib(n-2);
9) memo[n] = sum;
10) static int count = 0;
11) int fib(int n)
12) if(memo[n]!= NULL)
13) return memo[n];
```

In the code above, we've employed the memorizing strategy, storing the outcomes in an array so that we can reuse the values. This strategy, in which we start at the top and divide the problem into smaller issues, is also known as a top-down approach [6]–[8].

Bottom-Up strategy

One of the methods that can be utilized to implement dynamic programming is the bottom-up strategy. It applies the dynamic programming strategy using the tabulation method. It eliminates the recursion while still solving the same kind of issues. Recursive functions have no overhead or stack overflow problem if recursion is removed. In this method of tabulation, we answer the issues and put the solutions into a matrix.

The two methods for using dynamic programming are as follows:

1. Top-Down
2. Bottom-Up

To prevent recursion and conserve memory, the bottom-up method is utilized. While the recursive algorithm begins at the end and works backward, the bottom-up algorithm begins at the beginning. In the bottom-up method, we begin with the simplest instance and work our way up to the solution. The Fibonacci series' base cases are 0 and 1, as is well known. We'll begin with 0 and 1 as the bottom method starts with the base cases [9], [10].

Major points

Before moving on to the larger problems using smaller sub-problems, we solve all the smaller sub-problems that will be required to solve the larger sub-problems.

1. To iterate through the sub-problems, we utilize a for loop.
2. The tabulation or table filling method is another name for the bottom-up methodology.
3. Let's use an illustration to clarify.

Assume we have an array with 0 and 1 values, respectively, at a[0] and a[1] places, as illustrated below:

Programmatically changing

The values at a[0] and a[1] are added to obtain the value of a[2] because the bottom-up approach starts from the lower values, as demonstrated below:

Programmatically changing

Adding a[1] and a[2] will yield the value of a[3], which is 2 as seen below:

Programmatically changing

By combining a[2] and a[3], the value of a[4] will be found to be 3 as illustrated below:

Programmatically changing

When the values of a[4] and a[3] are added, the value of a[5] equals 5, as seen below:

Programmatically changing

The following contains the bottom-up approach's implementation of the Fibonacci series in code:

A[0] = 0, A[1] = 1; for(i=2; i=n; i++); int fib(int n) int A[]; A[0] = 0, A[1] = 1.

Return A[n] by formulating A[i] as A[i] = A[i-1] + A[i-2];

The basis cases in the code above are 0 and 1, and we then used a for loop to find further Fibonacci series values..

CONCLUSION

In conclusion, dynamic programming is a superior form of recursion that overcomes its limitations. However, DP can occasionally be challenging to comprehend, making it a well-liked option for coding interviews. Understanding how DP functions can be useful to everyone, whether they are a professional or a student getting ready for the workforce. The greatest approach to learn something new is to solve issues on your own, therefore visit these websites and resources to better comprehend DP.

REFERENCES:

[1] P. Bouman, N. Agatz, and M. Schmidt, "Dynamic programming approaches for the traveling salesman problem with drone," *Networks*, 2018, doi: 10.1002/net.21864.

[2] Ö. U. Nalbanto□lu, "Dynamic programming," *Methods Mol. Biol.*, 2014, doi: 10.1007/978-1-62703-646-7_1.

[3] J. N. Tsitsiklis and B. Van Roy, "Feature-based methods for large scale dynamic programming," *Mach. Learn.*, 1996, doi: 10.1007/BF00114724.

[4] M. P. Deisenroth, C. E. Rasmussen, and J. Peters, "Gaussian process dynamic programming," *Neurocomputing*, 2009, doi: 10.1016/j.neucom.2008.12.019.

[5] W. B. Powell, "What you should know about approximate dynamic programming," *Nav. Res. Logist.*, 2009, doi: 10.1002/nav.20347.

[6] B. Doerr, A. Eremeev, F. Neumann, M. Theile, and C. Thyssen, "Evolutionary algorithms and dynamic programming," *Theor. Comput. Sci.*, 2011, doi: 10.1016/j.tcs.2011.07.024.

[7] I. Fawwaz, A. Winarta, Selvianna, J. J. Ramli, and L. M. Waruwu, "Implementation Of Dynamic Programming Algorithm For Npc Movement In Police And Thief Game," *J. Inf. Technol. Educ. Res.*, 2019, doi: 10.31289/jite.v2i2.2169.

[8] F. Y. Wang, J. Zhang, Q. Wei, X. Zheng, and L. Li, "PDP: Parallel dynamic programming," *IEEE/CAA J. Autom. Sin.*, 2017, doi: 10.1109/JAS.2017.7510310.

[9] M. W. Ulmer, J. C. Goodson, D. C. Mattfeld, and M. Hennig, "Offline–online approximate dynamic programming for dynamic vehicle routing with stochastic requests," *Transp. Sci.*, 2019, doi: 10.1287/trsc.2017.0767.

[10] J. Zou, S. Ahmed, and X. A. Sun, "Stochastic dual dynamic integer programming," *Math. Program.*, 2019, doi: 10.1007/s10107-018-1249-5.

CHAPTER 20

A BRIEF DISCUSSION ON RECURSIVE AND ITERATIVE SOLUTIONS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

The paper on recursive and iterative solutions in data structures examines two core problem-solving methods: recursion and iteration. Recursion involves breaking down a problem into smaller instances, while iteration uses loops for repetitive execution. The chapter explores their characteristics, benefits, and suitable applications. It discusses the elegance of recursion but warns about potential stack overflow issues. Iteration's efficiency and clarity are highlighted, along with its relevance in optimizing certain algorithms. Through examples like factorial computation and tree traversal, the chapter contrasts the implementation of both approaches. By grasping these techniques, readers gain a versatile problem-solving toolkit for different scenarios in computer science and programming.

KEYWORDS:

Base Case, Iterative Loops, Problem Solving Techniques, Recursion, Recursive Functions.

INTRODUCTION

We will discuss iterative solutions now that we have quickly explained what a recursive algorithm is and why you might want to employ such an approach for your algorithms. There is absolutely no recursion in an iterative solution.current programming languages like C++, C#, and Java (among many others) allow you to design methods that reference themselves; these methods are known as recursive methods. This is one of the most concise aspects of current programming languages. One of the major benefits of recursive approaches is that they frequently produce simpler, easier-to-read solutions to issues. So, a method that defines itself in terms of itself is recursive.

Recursive algorithms typically have two key characteristics:

A recursive case in addition to one or more base cases.

For the time being, let's quickly go over these two recursive algorithmic features. We should be getting closer to our base case with each recursive iteration in order to avoid problems. We shall explain why the issue we are referring to generally appears as a stack overflow in the future.

All that is required for an iterative solution are loops (such as for, while, do-while, etc.). Iterative algorithms have the drawback of not always being as transparent in their operation as their recursive cousins. The quickness of iterative solutions is its main benefit. The majority of production software that you will encounter employs very little to no recursive algorithms. The latter attribute may occasionally be a requirement for a company before code can be checked in, for example, a static analysis tool may ensure that the developer's code does not contain

recursive algorithms. This zero tolerance criterion for recursive algorithms is typically applied to systems level programs [1]–[3].

Recursion should only ever be used with quick algorithms; it has the following runtime drawbacks that you should avoid:

1. O(n 2)

2. O(n 3).

3. O(2n)

You are asking for trouble if you utilize recursion for any of the aforementioned run-time efficient algorithms. These algorithms are expanding quickly, and they frequently rely significantly on divide-and-conquer strategies. Although it's best practice to continually break down larger problems into smaller ones, doing so in these situations will result in a lot of method calls.

Method calls aren't cheap, so all this cost will rapidly add up and either make your algorithm run much slower than it should or, worse, run out of stack space. The operating system will terminate the thread when you use more stack space than is permitted. This is true regardless of the platform you employ, such as native C++ or.NET. You can request a larger stack size, but you should normally only do so if you have an extremely compelling reason to do so.

Records of Activations

Every time a method is called, an activation record is produced. In order to facilitate method invocation, an activation record is just anything that is added to the stack. The creation of activation records is quick and easy because they are so lightweight.The following is a typical activation record for a method call (this is quite general):The top-of-stack index is increased by the complete amount of memory needed by the method's local variables, the actual method parameters are pushed onto the stack, the return address is pushed onto the stack, and a jump is made to the method [4], [5].

You will quickly run out of stack space in many recursive algorithms that operate on huge data structures or inefficient methods. Consider an algorithm that generates numerous recursive calls when used with a certain value. In this scenario, a sizable portion of the stack will be used. The call chain's nested methods must exit and return to their respective callers before the activation records may begin to be unwound. A method's activation record is unwound when it terminates. An activation record can be unwound in various steps:

1. The total amount of memory used by the technique is used to decrement the top-of-stack index.
2. The stack's return address is popped. The total amount of memory used by the real parameters decreases the top-of-stack index.

Although an effective mechanism to support method calls, activation records can quickly accumulate. If given the chance, recursive algorithms can quickly deplete the thread's allotted stack size. We should be dusting out the Fibonacci method, a classic illustration of an iterative vs. recursive solution, right about now. This is a well-known illustration because it demonstrates the advantages and disadvantages of recursive algorithms. The iterative solution completes the task much more quickly, however it is not as attractive or self-documenting. The Fibonacci

process has an O(g n) run time, so if we were to input, say, 60, we would have to wait a while to obtain the value back. On the other hand, the iterative variant has an O(n) run time. Do not be deterred from recursion by this. Instead of scaring programmers away, this example mostly serves to scare them into thinking about the implications of recursion [6]–[8].

DISCUSSION

Some issues have a recursive nature:

Some data structures and algorithms are actually recursive in nature, so keep that in mind. A tree data structure is the ideal illustration of this. A common tree node typically includes a value, two pointers to additional nodes of the same node type, and a value. As you can see, the tree's structure is recursive, with each node having the potential to point to two additional nodes.

It makes sense to use recursive algorithms while working with trees because you are merely following the data structure's intrinsic nature. Of course, there are some drawbacks because we are still constrained by the restrictions outlined earlier in this chapter. We can also consider sorting techniques like rapid sort and merge sort. Since the design of both of these algorithms is recursive, modeling them recursively makes sense.

The set of instructions is repeated using both recursion and iteration. When a statement within a function repeatedly invokes itself, recursion has taken place. Iteration happens when a loop runs repeatedly until the controlling condition is satisfied. Recursion and iteration vary fundamentally in that iteration is used to apply a set of instructions that we want to perform again whereas recursion is a procedure that is always applied to a function.

Recursion:

Recursion is the process of a function repeatedly calling itself. Selection structure is used in recursion. An infinite recursion happens if the recursion step fails to reduce the issue in a way that converges on the base condition. The system can be destroyed by an infinite recursion. A base case is identified when recursion comes to an end. Recursion typically takes longer than iteration due to the cost of maintaining the stack. Recursion consumes more memory than iteration as well. However, it reduces the size of the code, making it an incredible approach that makes the code simpler to read and write [9], [10].

Iteration:

Iteration is the repetition of a computational or mathematical procedure that lasts until the controlling condition ceases to hold true, according to the definition. The repetition structure is used. If the loop condition test never returns false, iteration results in an infinite loop. CPU cycles are continuously used in an infinite loop. When the loop condition is violated, an iteration comes to an end. Iteration uses less memory, but lengthens the code, making it more challenging to comprehend and write.

CONCLUSION

Recursion is a strong tool that all programmers should be familiar with. Recursion is useful if you don't use it to create an algorithm with a quadratic run time or higher because readability and performance are frequently tradeoffs in software projects. Of course, this isn't a general rule; we're simply being reckless here. Always, defensive coding will win out. Many times, recursive

data structures and algorithms which are by their very nature recursive have a natural place for recursion. In such cases, recursion is entirely appropriate. Recursion is a bit overkill for tasks like traversing linked lists. There are presumably fewer lines of code in its iterative counterpart than in its recursive version. You should speak with your compiler and run-time environment for more information as we can only discuss the effects of employing recursion abstractly. It's possible that your compiler recognizes and can optimize things like tail recursion. This is common; in fact, the majority of commercial compilers will do it.

REFERENCES:

[1] Z. He and D. Ding, "Efficient recursive-iterative solution for EM scattering problems," *Electron. Lett.*, 2015, doi: 10.1049/el.2014.4466.

[2] N. Baaziz and C. Labit, "Multiconstraint Wiener-Based Motion Compensation using Wavelet Pyramids," *IEEE Trans. Image Process.*, 1994, doi: 10.1109/83.334975.

[3] Y. Yang and W. Chen, "Taiga: Performance optimization of the C4.5 decision tree construction algorithm," *Tsinghua Sci. Technol.*, 2016, doi: 10.1109/TST.2016.7536719.

[4] B. Sanchez, C. Rausch, and C. Haas, "'Deconstruction programming for adaptive reuse of buildings,'" *Autom. Constr.*, 2019, doi: 10.1016/j.autcon.2019.102921.

[5] H. Liu, X. Gao, P. H. Schimpf, F. Yang, and S. Gao, "A recursive algorithm for the three-dimensional imaging of brain electric activty: Shrinking LORETA-FOCUSS," *IEEE Trans. Biomed. Eng.*, 2004, doi: 10.1109/TBME.2004.831537.

[6] V. Kralev, "An analysis of a recursive and an iterative algorithm for generating permutations modified for Travelling Salesman Problem," *Int. J. Adv. Sci. Eng. Inf. Technol.*, 2017, doi: 10.18517/ijaseit.7.5.3173.

[7] R. A. Obando, "Teaching object-oriented recursive data structures simplifying the definition and implementation of the data structures," in *ACMSE 2019 - Proceedings of the 2019 ACM Southeast Conference*, 2019. doi: 10.1145/3299815.3314432.

[8] B. Xu, X. Wang, A. A. Razzaqi, and X. Zhang, "Topology optimisation method for MACL formation based on acoustic measurement network," *IET Radar, Sonar Navig.*, 2019, doi: 10.1049/iet-rsn.2018.5384.

[9] E. Morales-Casique and S. P. Neuman, "Laplace-transform finite element solution of nonlocal and localized stochastic moment equations of transport," *Commun. Comput. Phys.*, 2009, doi: 10.4208/cicp.2009.v6.p131.

[10] J. C. Herz, "Recursive Computational Procedure for Two-dimensional Stock Cutting," *IBM J. Res. Dev.*, 2010, doi: 10.1147/rd.165.0462.

CHAPTER 21

A BRIEF DISCUSSION ON DIVIDE AND CONQUER

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

The "Divide and Conquer" paradigm is a fundamental algorithmic technique that involves breaking down complex problems into smaller, more manageable subproblems, solving these subproblems independently, and then combining their solutions to solve the original problem. This chapter explores the principles of the Divide and Conquer approach, its applications, benefits, and challenges. By understanding this technique, readers gain a powerful tool for solving intricate problems efficiently and optimizing algorithm design.

KEYWORDS:

Algorithmic Technique,Divide and Conquer, Efficiency, Optimization, Problem Solving, Subproblems.

INTRODUCTION

In the domain of algorithmic problem-solving, the "Divide and Conquer" strategy stands out as a pivotal technique, offering a robust framework for grappling with intricate challenges by skillfully breaking them down into smaller, more manageable subproblems. Rooted in the timeless adage that dissecting a convoluted task into simpler components often paves the way for enhanced efficiency, this approach epitomizes the quintessential wisdom of strategic simplicity. At its core, the essence of Divide and Conquer resides in its remarkable aptitude for orchestrating complex solutions through a two-fold process: the recursive resolution of subproblems, followed by the astute amalgamation of their solutions, culminating in the resolution of the overarching problem at hand.

In its fundamental form, the mechanics of the Divide and Conquer strategy are elegantly straightforward, yet remarkably potent. It adheres to a distinct three-step process—namely, "divide," "conquer," and "combine." The "divide" phase constitutes the initial dissection of a multifaceted problem into a constellation of smaller, distinct subproblems. These subproblems are meticulously crafted to encapsulate the essence of the original challenge, albeit on a reduced scale.

This strategic disassembly of complexity lays the groundwork for the "conquer" phase, wherein each of the subproblems is methodically addressed using the most suitable approach or algorithm. This often involves the judicious application of recursion, whereby the same Divide and Conquer strategy is employed iteratively to unravel subproblems until they attain a level of simplicity that permits direct, self-contained solutions. Once these subproblems are individually resolved, the "combine" phase enters the scene. Here, the focus shifts to the careful integration of the solutions to the subproblems, a process that synthesizes the separate threads of solution into a coherent and all-encompassing resolution to the original problem [1]–[3].

However, it's crucial to recognize that the utility of the Divide and Conquer approach extends far beyond its basic mechanics. This technique is adaptable, offering various nuanced variants that cater to the diverse spectrum of challenges that algorithmic problem-solving presents. The "pure" iteration adheres to the principle of uniform division, wherein the problem is dissected into subproblems of equal or nearly equal size, allowing for standardized solutions and consolidation.

On the other hand, the "Divide and Conquer with Optimization" variant introduces an additional layer of sophistication. It strategically harnesses selective combination of specific subproblems to optimize performance, ensuring that redundancy is minimized and efficiency is maximized. This diversity of types reflects the technique's inherent flexibility and its ability to cater to the unique intricacies of distinct problems.

In practice, the applications of the Divide and Conquer strategy span a multitude of domains, underscoring its versatility and efficacy. Consider sorting algorithms such as Merge Sort, where this technique is conspicuously manifest. The unsorted array undergoes a process of division, as its elements are segmented and sorted independently.

The subsequent merging phase brings these ordered segments together to produce a fully sorted array. Similarly, Quick Sort adheres to the Divide and Conquer approach, partitioning the array and autonomously sorting the partitions before amalgamating them into a comprehensive solution.

Furthermore, the application of Divide and Conquer is especially pronounced in searching algorithms, epitomized by the Binary Search technique. Here, the search space is progressively narrowed down by dividing it into smaller regions and making informed comparisons. This process aligns harmoniously with the tenets of Divide and Conquer, illustrating how this strategy efficiently navigates through complex problems.

In sum, the Divide and Conquer approach stands as a testament to the power of strategic simplicity in algorithmic problem-solving. By embracing the philosophy of dissecting intricate problems into manageable fragments and then reassembling the solutions, this technique offers an invaluable tool for optimizing solutions to multifaceted challenges. Its recursive nature, adaptability across various problem types, and tangible impact on algorithmic advancements solidify its significance in the field of computer science. Understanding the intricacies of the Divide and Conquer paradigm empowers individuals to approach convoluted computational obstacles with a systematic and efficient strategy.

Types and Characteristics:

The Divide and Conquer technique encompasses various forms, each tailored to the nature of the problem at hand. One notable variant is the "pure" Divide and Conquer, where the problem is divided into equal or nearly equal-sized subproblems, solved independently, and merged to obtain the final solution. Another variant is "Divide and Conquer with Optimization," where certain subproblems are strategically combined to optimize performance and reduce redundancy.

Characteristic to this approach is its recursive nature, which often involves applying the same technique to the subproblems until they become trivial enough to be solved directly. Additionally, Divide and Conquer thrives on the ability to transform intricate problems into smaller, independent instances that can be solved efficiently [4], [5].

Applications:

Divide and Conquer finds applications across diverse domains, underlining its versatility and effectiveness. In the realm of sorting algorithms, techniques like Merge Sort and Quick Sort exemplify this strategy. These algorithms divide an unsorted array into smaller segments, sort them independently, and then merge them into a fully sorted array.

Binary Search, a widely used searching algorithm, employs Divide and Conquer to narrow down search regions efficiently. By consistently halving the search space based on comparison results, this technique ensures optimal searching in a sorted dataset. In the realm of computer graphics, rendering complex scenes is streamlined using the Divide and Conquer approach. The scene is broken into smaller portions, each processed separately, and then combined to create the final visual output.

Key Components:

The key components of the Divide and Conquer technique include the "divide" step, where the problem is decomposed into smaller subproblems; the "conquer" step, where these subproblems are independently solved using appropriate methods; and the "combine" step, where solutions of subproblems are merged to yield the solution to the original problem. Furthermore, the efficiency of Divide and Conquer hinges on the ability to balance the size of subproblems and manage the overhead associated with recursion. Smartly identifying and addressing overlapping subproblems can also significantly enhance the technique's performance. In the forthcoming exploration, we will delve deeper into the principles and strategies of the Divide and Conquer paradigm. By gaining an understanding of its nuances, applications, and limitations, readers will empower themselves with a versatile toolset for addressing intricate computational challenges.

DISCUSSION

The "Divide and Conquer" paradigm is an algorithmic strategy that embodies the essence of efficiency through decomposition. Rooted in the concept of breaking down complex problems into simpler, more manageable subproblems, this technique is a cornerstone of algorithm design. Its power lies in the ability to navigate intricate challenges by leveraging the advantages of independent problem-solving and intelligent combination of solutions. In this section, we delve deeper into the intricacies of the Divide and Conquer approach, exploring its mechanics, types, applications, and its impact on algorithmic problem-solving.

Mechanics of Divide and Conquer:

At its core, the Divide and Conquer strategy follows a structured methodology that can be distilled into three distinct phases: "divide," "conquer," and "combine." In the "divide" phase, the complex problem is dissected into smaller, more manageable subproblems. These subproblems are designed to be independent instances of the original problem, lending themselves to autonomous resolution.

The "conquer" phase involves solving each subproblem using the most suitable technique. This often entails recursive application of the same Divide and Conquer strategy to further break down the subproblems until they reach a level of simplicity that permits direct solutions. Finally, in the "combine" phase, the solutions of the subproblems are adeptly merged to form the solution to the overarching problem [6]–[8].

Types of Divide and Conquer:

The Divide and Conquer technique isn't a monolithic approach; rather, it manifests in several forms to cater to diverse problem domains. The "pure" form involves uniform division of the problem into equal or nearly equal-sized subproblems, followed by their independent solution and merging. On the other hand, "Divide and Conquer with Optimization" emphasizes selective combination of certain subproblems to optimize performance and minimize redundancy. This diversity of types underscores the adaptability of the technique to address a wide array of challenges.

Applications of Divide and Conquer:

The power of Divide and Conquer is manifested in its wide-ranging applications across various domains. In sorting algorithms, Merge Sort exemplifies the essence of this technique. The unsorted array is divided into smaller segments, sorted independently, and then combined through merging to achieve a fully sorted array. Quick Sort, another sorting algorithm, also follows the Divide and Conquer approach by partitioning the array and sorting the partitions independently.

Binary Search is another prime application, particularly in searching algorithms. It employs the Divide and Conquer strategy to efficiently narrow down the search space. By halving the search region based on comparison results, Binary Search ensures optimal search performance, making it a fundamental algorithm for fast information retrieval.

Beyond these classic examples, Divide and Conquer has a profound impact in various fields. In computer graphics, complex scene rendering benefits from this technique. The scene is divided into smaller sections, each processed individually, and then reassembled to create the final visual output.

Efficiency and Limitations:

The efficiency of Divide and Conquer lies in its capacity to manage complexity by addressing smaller, more manageable units. However, it's important to note that this approach isn't a universal panacea. The overhead associated with dividing and combining subproblems can impact performance for problems where subproblems are too small or the combination process is computationally intensive. Additionally, identifying efficient division points can be non-trivial, making some problems less amenable to this technique [9]–[11].

Illustrative Example: Merge Sort

As an illustrative example, consider the Merge Sort algorithm. Given an unsorted array, Merge Sort employs Divide and Conquer. The array is divided into halves, each half is recursively sorted, and then the sorted halves are merged to obtain the final sorted array in C.

1. function MergeSort(arr):
2. Step 1: if length of arr <= 1:
3. Step 2: return arr

4. Step 3: mid = length of arr / 2
5. Step 4: left_half = arr[0 to mid - 1]
6. Step 5: right_half = arr[mid to end]
7. Step 6: sorted_left = MergeSort(left_half)
8. Step 7: sorted_right = MergeSort(right_half)
9. Step 8: sorted_arr = Merge(sorted_left, sorted_right)
10. Step 9: return sorted_arr
11. Step 10: function Merge(left, right):
12. Step 11: result = empty array

13. Step 12: while left is not empty and right is not empty:
14. Step 13: if first element of left <= first element of right:
15. Step 14: append first element of left to result
16. Step 15: remove first element from left
17. Step 16: else:
18. Step 17: append first element of right to result
19. Step 18: remove first element from right
20. Step 19: append remaining elements of left to result
21. Step 20: append remaining elements of right to result
22. Step 21: return result

CONCLUSION

In conclusion, the Divide and Conquer paradigm epitomizes an elegant and efficient approach to problem-solving. By deconstructing complex problems into smaller, solvable components and subsequently merging their solutions, this technique offers a versatile tool for algorithm design. Its recursive nature, adaptability through various types, and broad applications underscore its significance in the field of algorithms. Understanding the mechanics, applications, and limitations of Divide and Conquer empowers individuals to approach complex computational challenges with a systematic and optimized strategy.

REFERENCES:

[1] E. K. Molloy and T. Warnow, "Statistically consistent divide-and-conquer pipelines for phylogeny estimation using NJMerge," *Algorithms Mol. Biol.*, 2019, doi: 10.1186/s13015-019-0151-x.

[2] P. Yang, K. Tang, and X. Yao, "A Parallel Divide-and-Conquer-Based Evolutionary Algorithm for Large-Scale Optimization," *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2938765.

[3] A. Farzan and V. Nicolet, "Synthesis of divide and conquer parallelism for loops," *ACM SIGPLAN Not.*, 2017, doi: 10.1145/3062341.3062355.

[4] A. Napov, "A divide-and-conquer bound for aggregate's quality and algebraic connectivity," *Discrete Math.*, 2017, doi: 10.1016/j.disc.2017.05.003.

[5] P. Zhuang and X. Ding, "Divide-and-conquer framework for image restoration and enhancement," *Eng. Appl. Artif. Intell.*, 2019, doi: 10.1016/j.engappai.2019.08.008.

[6] C. Ye and T. Tian, "New insights into divide-and-conquer attacks on the round-reduced Keccak-MAC," *Chinese J. Electron.*, 2019, doi: 10.1049/cje.2019.04.002.

[7] E. D. Russell, "Resisting Divide and Conquer: Worker/Environmental Alliances and the Problem of Economic Growth," *Capital. Nature, Social.*, 2018, doi: 10.1080/10455752.2017.1360924.

[8] P. Cignoni, C. Montani, and R. Scopigno, "DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed," *CAD Comput. Aided Des.*, 1998, doi: 10.1016/S0010-4485(97)00082-1.

[9] S. Itzhaky *et al.*, "Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations," *ACM SIGPLAN Not.*, 2016, doi: 10.1145/2983990.2983993.

[10] X. Chang, S. B. Lin, and Y. Wang, "Divide and conquer local average regression," *Electron. J. Stat.*, 2017, doi: 10.1214/17-EJS1265.

[11] J. Feng, L. Wang, H. Yu, L. Jiao, and X. Zhang, "Divide-and-conquer dual-architecture convolutional neural network for classification of hyperspectral images," *Remote Sens.*, 2019, doi: 10.3390/rs11050484.

CHAPTER 22

A BRIEF DISCUSSION ON ADVANCE DATA STRUCTURE

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology,
Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

Advanced Data Structures, such as B-Trees and Skip Lists, transcend the realm of basic data structures, offering enhanced efficiency and versatility. This chapter delves into these sophisticated structures, exploring their intricate design, unique characteristics, wide-ranging applications, and the key components that define their functionality.

By comprehending these advanced data structures, readers gain a powerful toolkit for handling complex data scenarios and optimizing performance in various computational contexts.

KEYWORDS:

Advanced Data Structures, Applications, B-Trees, Efficiency, Key Components, Skip Lists, Versatility.

INTRODUCTION

In the landscape of data structures, a new tier of sophistication is achieved with the advent of Advanced Data Structures. These structures, such as B-Trees and Skip Lists, go beyond the elementary building blocks and harness intricate design principles to optimize data storage and manipulation.

This chapter embarks on a journey to explore these advanced data structures, shedding light on their varied types, distinguishing characteristics, far-reaching applications, and the pivotal components that underpin their effectiveness.

Types:

Advanced Data Structures stand as a testament to the evolution of computational tools, providing solutions to complex scenarios where conventional structures fall short. Among these, B-Trees emerge as a versatile type, renowned for their balanced hierarchical organization.

Their balanced nature enables efficient insertion, deletion, and search operations, making them suitable for applications like databases and file systems [1]–[3].

Characteristics:

Characteristics like variable node capacity and multiple keys per node empower B-Trees to adapt to varying data loads.

Skip Lists, another noteworthy structure, offer an intriguing alternative. With probabilistic design, they strike a balance between simplicity and efficiency.

Applications:

Skip Lists are particularly useful for applications that require rapid search operations. Their multi-level structure and probabilistic approach enable efficient search and insertion, making them a viable choice for tasks like implementing dynamic data structures and maintaining sorted lists.

Key Components:

The key components of these advanced structures constitute their essence. In B-Trees, nodes are pivotal, with a hierarchical arrangement ensuring balance. The branching factor, determined by the type of B-Tree, affects the structure's depth and overall performance. Leaf nodes house data pointers, enabling rapid data retrieval. In Skip Lists, levels and nodes are central components. Levels define the hierarchical structure, while nodes contain data and pointers to elements in the same and lower levels. The number of levels influences the search and insertion complexity.

DISCUSSION

Advanced Data Structures elevate the landscape of data manipulation by offering tailored solutions for intricate scenarios. B-Trees, for instance, thrive in applications where data must be stored and retrieved efficiently, such as databases. The balanced nature of B-Trees guarantees logarithmic time complexity for essential operations, a critical factor in managing large datasets. Their adaptability to changes in data volume and dynamic reorganization makes them indispensable in scenarios with fluctuating data loads.

Skip Lists, on the other hand, introduce an ingenious probabilistic approach to optimization. Their simplicity of design doesn't compromise efficiency; they strike a balance between the straightforward linked list and the intricate balanced tree. Skip Lists excel in search-intensive applications, capitalizing on their multi-level structure to accelerate search operations. This makes them a valuable asset in data structures where rapid searching is paramount.

Both B-Trees and Skip Lists contribute to database management systems, ensuring efficient indexing and query processing. In file systems, B-Trees aid in organizing directories and files, facilitating rapid data retrieval. Skip Lists find use in areas like peer-to-peer networking and maintaining sorted data streams [4], [5]. Figure 1 shown insertion of B-tree.

Code Example: B-Tree Insertion

1) Step 1: class BTreeNode:
2) Step 2: def __init__(self, leaf=True):
3) Step 3: self.keys = []
4) Step 4: self.children = []
5) Step 5: self.leaf = leaf
6) class BTree:
7) def __init__(self, t):
8) Step 1: self.root = BTreeNode()
9) Step 2: self.t = t
10) def insert(self, key):
11) Step 1: root = self.root
12) Step 2: if len(root.keys) == (2 * self.t) - 1:

13) Step 3: new_root = BTreeNode(leaf=False)
14) Step 4: new_root.children.append(root)
15) Step 5: self.split_child(new_root, 0)
16) Step 6: self.insert_non_full(new_root, key)
17) Step 7: self.root = new_root
18) Step 8: else:
19) Step 9: self.insert_non_full(root, key)
20) def insert_non_full(self, x, k):
21) Step 1: i = len(x.keys) - 1
22) Step 2: if x.leaf:
23) Step 3: x.keys.append(None)
24) Step 4: while i >= 0 and k < x.keys[i]:
25) Step 5: x.keys[i + 1] = x.keys[i]
26) Step 6: i -= 1
27) Step 7: x.keys[i + 1] = k
28) Step 8: else:
29) Step 9: while i >= 0 and k < x.keys[i]:
30) Step 10: i -= 1
31) Step 11: i += 1
32) Step 12: if len(x.children[i].keys) == (2 * self.t) - 1:
33) Step 13: self.split_child(x, i)
34) Step 14: if k > x.keys[i]:
35) Step 15: i += 1
36) self.insert_non_full(x.children[i], k)
37) def split_child(self, x, i):
38) Step 1: t = self.t
39) Step 2: y = x.children[i]
40) Step 3: z = BTreeNode(leaf=y.leaf)
41) Step 4: x.children.insert(i + 1, z)
42) Step 5: x.keys.insert(i, y.keys[t - 1])
43) Step 6: z.keys = y.keys[t:]
44) Step 7: y.keys = y.keys[:t - 1]
45) Step 8: if not y.leaf:
46) Step 9: z.children = y.children[t:]
47) Step 10: y.children = y.children[:t]
48) # Example usage
49) Step 11: b_tree = BTree(t=2)
50) Step 12: keys = [3, 8, 12, 15, 20, 25, 30, 35, 38]
51) Step 13: for key in keys:
52) Step 14: b_tree.insert(key).
53) Step 15: B-Trees: Balancing for Efficiency.

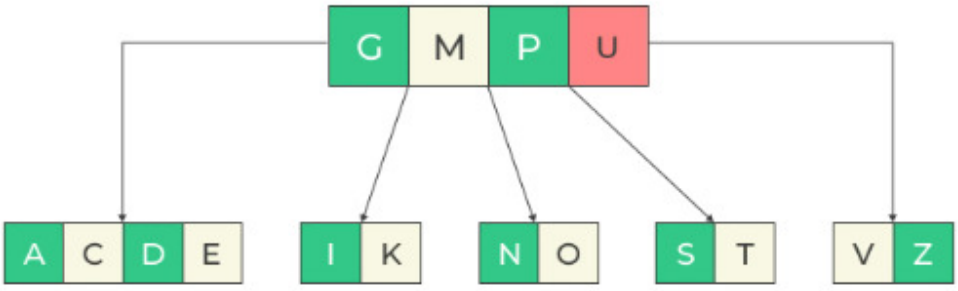


Figure 1: For Insertion in B-tree (prepinsta.com).

B-Trees, a pinnacle of balanced hierarchical structures, epitomize the marriage of efficiency and adaptability. These trees are tailored for scenarios where large volumes of data need to be managed and retrieved with logarithmic time complexity.

A defining characteristic of B-Trees is their balanced nature, meticulously orchestrated to ensure that the depth of the tree remains optimal. This balance bestows them with a remarkable capability: regardless of the volume of data, B-Trees maintain efficient search, insertion, and deletion operations.

The versatility of B-Trees is a testament to their adaptive capacity. Unlike binary trees, B-Trees can have multiple keys per node, and the number of keys per node is defined by the branching factor, which determines the maximum number of children a node can have.

This adaptability allows B-Trees to flexibly adjust to variations in data load, a critical aspect in scenarios where data flows fluctuate. As data is inserted or removed, B-Trees reorganize themselves, ensuring that the balance is preserved and that operations remain efficient.

B-Trees find widespread applications in various domains. In database management systems, where efficient indexing and query processing are paramount, B-Trees offer a potent solution. They provide a means to organize data in a way that minimizes the number of comparisons required for searching, translating to faster query responses. Furthermore, B-Trees shine in file systems, where they streamline the organization of directories and files. This hierarchical structure mirrors the way directories are structured in operating systems, allowing for quick navigation and data retrieval [6]–[8].

Skip Lists: Probability and Efficiency

In contrast to the deterministic nature of B-Trees, Skip Lists introduce a probabilistic dimension to data structure design. Skip Lists are a fusion of linked lists and balanced trees, offering a pragmatic compromise between simplicity and efficiency. At the heart of Skip Lists is the ingenious use of probability to dictate the layout of the structure. This probabilistic approach leads to a multi-level design, where elements are interconnected across various levels, allowing for rapid traversal and search.

Skip Lists are particularly effective in applications where rapid search operations are crucial. The multi-level structure empowers them to swiftly zero in on desired elements without exhaustive comparisons, leading to improved time complexity for searches. While B-Trees demand stricter adherence to balanced hierarchies, Skip Lists embrace a more relaxed balance, permitting variations in node heights that still result in efficient operations.

One notable application of Skip Lists lies in dynamic data structures. When maintaining sorted lists that require frequent insertions and deletions, Skip Lists offer an elegant solution. Their structure accommodates changes while still ensuring efficient search and traversal. Additionally, Skip Lists find utility in peer-to-peer networking, where elements need to be located swiftly to establish connections.

Key Components: Enabling Efficiency

The key components of these advanced structures form the pillars of their functionality. In the realm of B-Trees, nodes are of central significance. The hierarchical arrangement of nodes ensures balance, maintaining a consistent depth and optimizing performance.

The branching factor, determined by the type of B-Tree, significantly influences the tree's depth and overall efficiency. Moreover, the leaf nodes play a vital role, housing pointers to data blocks and enabling rapid data retrieval.

For Skip Lists, the critical components include levels and nodes. Levels define the structure's hierarchy, with higher levels encompassing fewer elements. Nodes house data elements and maintain pointers to elements in the same level and lower levels.

The number of levels has a direct impact on the structure's efficiency; more levels allow for faster searches but also increase space complexity [9], [10].

Advancing Computational Efficiency:

In conclusion, Advanced Data Structures, epitomized by B-Trees and Skip Lists, provide a pivotal shift in data structure design, offering solutions to intricate scenarios where conventional structures fall short.

B-Trees harness balanced hierarchies to optimize data storage and retrieval, making them indispensable in database management and file systems. Skip Lists introduce a probabilistic approach, striking a balance between efficiency and simplicity, and excel in applications where swift search operations are essential.

Understanding the intricacies, characteristics, applications, and key components of these advanced structures empowers individuals to tackle the ever-evolving landscape of data manipulation with a strategic advantage.

These structures serve as testaments to the continuous evolution of computational tools, offering tailored solutions that meet the demands of modern computing challenges. By harnessing these advanced constructs, individuals can optimize data management, improve efficiency, and navigate complex computational scenarios with finesse.

CONCLUSION:

Advanced Data Structures, exemplified by B-Trees and Skip Lists, offer a deeper level of sophistication in managing complex data scenarios. B-Trees, with their balanced hierarchical design, excel in applications that demand efficient storage and retrieval of data. Skip Lists, with their probabilistic approach, shine in tasks requiring fast search operations. Understanding the characteristics, applications, and key components of these advanced structures equips individuals with the tools to optimize data manipulation in diverse computational contexts. These structures stand as a testament to the ever-evolving landscape of data structures, where tailored solutions meet the demands of modern computing challenges.

REFERENCES:

[1] P. Chang, M. Gohain, M. R. Yen, and P. Y. Chen, "Computational Methods for Assessing Chromatin Hierarchy," *Computational and Structural Biotechnology Journal*. 2018. doi: 10.1016/j.csbj.2018.02.003.

[2] H. Mavoa *et al.*, "Knowledge exchange in the Pacific: The TROPIC (Translational Research into Obesity Prevention Policies for Communities) project," *BMC Public Health*, 2012, doi: 10.1186/1471-2458-12-552.

[3] K. Grochowska *et al.*, "Properties of plasmonic arrays produced by pulsed-laser nanostructuring of thin Au films," *Beilstein Journal of Nanotechnology*. 2014. doi: 10.3762/bjnano.5.219.

[4] P. Mylonas, "The evolution of contextual information processing in informatics," *Inf.*, 2018, doi: 10.3390/info9030047.

[5] R. van Woesik, "Processes Regulating Coral Communities," *Comments* *Theor. Biol.*, 2002, doi: 10.1080/08948550214054.

[6] M. Raskovic, "Economic sociology and the ara interaction model," *J. Bus. Ind. Mark.*, 2015, doi: 10.1108/JBIM-09-2011-0123.

[7] J. G. Kruth, "Five qualitative research approaches and their applications in parapsychology," in *Journal of Parapsychology*, 2015.

[8] R. Gani, "Group contribution-based property estimation methods: advances and perspectives," *Current Opinion in Chemical Engineering*. 2019. doi: 10.1016/j.coche.2019.04.007.

[9] G. A. Hajj *et al.*, "COSMIC GPS ionospheric sensing and space weather," *Terr. Atmos. Ocean. Sci.*, 2000, doi: 10.3319/TAO.2000.11.1.235(COSMIC).

[10] A.-M. Hoskinson, L. Conner, S. Hester, M. B. Leigh, A. P. Martin, and T. Powers, "Coevolution or not? Crossbills, squirrels and pinecones," *CourseSource*, 2014, doi: 10.24918/cs.2014.4.

CHAPTER 23

GRAPH ALGORITHMS (DIJKSTRA, BFS, DFS AND BFA)

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

Graph Algorithms, including Breadth-First Search, Depth-First Search, Dijkstra's Algorithm, and Bellman-Ford Algorithm, serve as vital tools in solving complex problems involving interconnected data structures.

This chapter explores these graph algorithms in depth, elucidating their underlying principles, distinctive characteristics, diverse applications, and the essential components that drive their efficiency. Understanding these algorithms equips individuals with the ability to traverse, analyze, and optimize data within intricate networks.

KEYWORDS:

Bellman-Ford Algorithm, Breadth-First Search, Depth-First Search, Dijkstra's Algorithm, Graph Algorithms, Interconnected Data Structures.

INTRODUCTION

In the realm of algorithmic problem-solving, Graph Algorithms play an indispensable role in untangling the complexities of interconnected data structures. These algorithms, which include Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's Algorithm, and the Bellman-Ford Algorithm, form the backbone of graph-based computations.

This chapter delves into these fundamental graph algorithms, shedding light on their diverse types, defining characteristics, wide-ranging applications, and the key components that drive their efficacy.

Types, Characteristics, Applications:

Graphs, as versatile data structures, give rise to a myriad of computational challenges, from network analysis to route optimization. Breadth-First Search (BFS) is a graph traversal algorithm known for its systematic exploration of nodes level by level. This methodical approach renders BFS ideal for scenarios where the shortest path between nodes or levels needs to be identified, making it a staple in shortest path problems and network analysis [1]–[3].

Depth-First Search (DFS), in contrast, takes an exploratory approach, delving deep into a graph's branches before backtracking. This algorithm excels in scenarios that require exhaustive exploration of all possible paths, making it invaluable for tasks like topological sorting, maze-solving, and cycle detection.

Dijkstra's Algorithm, a cornerstone of pathfinding, prioritizes the exploration of paths with the lowest cumulative cost. Its application extends beyond traditional graphs to scenarios involving

weighted graphs, such as finding optimal routes in maps, networking, and transportation systems. The Bellman-Ford Algorithm, another pathfinding gem, caters to graphs containing negative-weight edges. It efficiently identifies shortest paths while accounting for negative weights, making it suitable for applications like network routing and resource allocation.

Key Components:

The key components of these graph algorithms establish their unique functionalities. For BFS and DFS, the traversal process hinges on the selection of starting nodes and the systematic exploration of adjacent nodes. In BFS, a queue guides the exploration, while DFS employs a stack or recursion to facilitate traversal. Dijkstra's Algorithm relies on priority queues to manage the selection of the next node for exploration, ensuring the shortest path is progressively discovered. The Bellman-Ford Algorithm, characterized by relaxation steps, iteratively refines distance estimates to obtain optimal paths.

DISCUSSION

Graph Algorithms, as powerful tools in computational problem-solving, offer distinct approaches to traversing and analyzing interconnected data structures. Breadth-First Search (BFS) is renowned for its methodical exploration, rendering it indispensable in scenarios that require the shortest path between nodes. Its application spans from social network analysis to web crawling, where understanding relationships and navigation efficiency are paramount. Depth-First Search (DFS), on the other hand, showcases its prowess in exhaustive path exploration. The algorithm's capacity to traverse deep into branches makes it particularly adept at maze-solving, topological sorting, and cycle detection. DFS finds use in various domains, including compiler design, artificial intelligence, and graph theory [4], [5]. Dijkstra's Algorithm stands as a paragon of pathfinding algorithms, optimizing route discovery in weighted graphs. Its prioritized exploration of nodes based on cumulative costs underpins efficient navigation systems, driving its applications in maps, logistics, and transportation networks. Additionally, Dijkstra's Algorithm finds utility in computer networks, optimizing data packet routing and minimizing latency.

The Bellman-Ford Algorithm offers a critical solution for scenarios involving graphs with negative-weight edges. Its iterative relaxation steps enable the discovery of optimal paths while accounting for negative weights, ensuring its applicability in scenarios like financial modeling, where investments and debts are assigned weights.

Example: Dijkstra's Algorithm

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Breadth-First Search (BFS):

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

Bellman-Ford Algorithm:

Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

CONCLUSION

In conclusion, Graph Algorithms encompass a diverse range of tools that navigate and analyze interconnected data structures. Breadth-First Search, Depth-First Search, Dijkstra's Algorithm, and the Bellman-Ford Algorithm each contribute unique strategies to solving problems within graphs. These algorithms serve as pillars of efficiency, optimizing network analysis, route discovery, and resource allocation. By delving into the intricacies of these algorithms

understanding their characteristics, applications, and key components individuals gain a potent toolkit for handling complex data scenarios within interconnected systems. As technology advances, the relevance of Graph Algorithms continues to grow, empowering computational problem solvers with the means to unravel the intricacies of intricate networks.

REFERENCES:

[1] R. Dondi, G. Mauri, and I. Zoppis, "Graph Algorithms," in *Encyclopedia of Bioinformatics and Computational Biology: ABC of Bioinformatics*, 2018. doi: 10.1016/B978-0-12-809633-8.20424-X.

[2] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale graph algorithms," *Parallel Comput.*, 2011, doi: 10.1016/j.parco.2011.02.004.

[3] T. A. Davis, "Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear Algebra," *ACM Trans. Math. Softw.*, 2019, doi: 10.1145/3322125.

[4] K. Wehmuth, É. Fleury, and A. Ziviani, "Multi aspect graphs: Algebraic representation and algorithms," *Algorithms*, 2017, doi: 10.3390/a10010001.

[5] D. Breuker, P. Delfmann, H. A. Dietrich, and M. Steinhorst, "Graph theory and model collection management: conceptual framework and runtime analysis of selected graph algorithms," *Inf. Syst. E-bus. Manag.*, 2015, doi: 10.1007/s10257-014-0243-6.

[6] M. Erwig, "Inductive graphs and functional graph algorithms," *J. Funct. Program.*, 2001, doi: 10.1017/S0956796801004075.

[7] S. Yang, B. Yang, H. S. Wong, and Z. Kang, "Cooperative traffic signal control using Multi-step return and Off-policy Asynchronous Advantage Actor-Critic Graph algorithm," *Knowledge-Based Syst.*, 2019, doi: 10.1016/j.knosys.2019.07.026.

[8] R. Sarno and K. R. Sungkono, "A survey of graph-based algorithms for discovering business processes," *Int. J. Adv. Intell. Informatics*, 2019, doi: 10.26555/ijain.v5i2.296.

[9] P. F. Felzenszwalb and R. Zabih, "Dynamic programming and graph algorithms in computer vision," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2011, doi: 10.1109/TPAMI.2010.135.

[10] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub)graph isomorphism algorithm for matching large graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, 2004, doi: 10.1109/TPAMI.2004.75.

CHAPTER 24

A BRIEF DISCUSSION ON NP-COMPLETENESS AND COMPUTATIONAL INTRACTABILITY

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

NP-Completeness and Computational Intractability represent critical concepts in the realm of computational theory. This chapter delves into the depths of these concepts, unraveling the intricacies of NP-Completeness and its implications on problem-solving complexity. By exploring the foundational principles, types of problems, characteristics, and real-world applications, readers gain a comprehensive understanding of the challenges posed by NP-Completeness and the broader landscape of computational intractability.

KEYWORDS:

Complexity Theory, Computational Intractability, Decision Problems, NP-Completeness, P vs. NP, Polynomial Time, Reductions, Real-World Applications.

INTRODUCTION

In the landscape of computational theory, NP-Completeness and Computational Intractability stand as monumental pillars that shape our understanding of the inherent complexity of certain problems. These concepts delve into the heart of decision problems and the daunting challenges that arise when dealing with problems that seem to defy efficient solutions. This chapter embarks on a journey through the intricacies of NP-Completeness and the broader realm of computational intractability, shedding light on their typologies, defining characteristics, real-world applications, and the essential components that underpin their significance [1]–[3].

Types, Characteristics, Applications, and Key Components:

Types of Problems:

NP-Completeness orbits around a distinct class of decision problems. These problems are characterized by the ability to verify potential solutions efficiently, albeit without a known efficient algorithm to find the solutions themselves. The class NP (nondeterministic polynomial time) encapsulates problems for which a proposed solution can be verified in polynomial time by a deterministic machine. The enigma of P vs. NP lies at the heart of computational theory, raising the fundamental question of whether problems that can be verified quickly can also be solved quickly.

Characteristics of NP-Completeness:

One hallmark of NP-Completeness is its contagious nature. Problems can be reduced to one another through polynomial-time transformations, forming a web of complexity that showcases the interconnectedness of these challenges. This web of reductions forms the foundation of NP-

Completeness proofs, where demonstrating that a problem is NP-Complete involves showcasing its equivalence to an already established NP-Complete problem.

Applications in the Real World:

The impact of NP-Completeness reverberates across various domains. Problems arising in scheduling, optimization, and cryptography often fall within the NP-Complete realm. The Traveling Salesman Problem, the Knapsack Problem, and the Boolean Satisfiability Problem are all exemplars of NP-Complete problems with tangible applications in logistics, resource allocation, and circuit design.

Key Components of NP-Completeness:

The crux of NP-Completeness lies in reductions—transformations that map one problem to another. A reduction from problem A to problem B involves crafting an algorithm that translates instances of problem A into instances of problem B in polynomial time. The goal is to demonstrate that if an efficient algorithm exists for problem B, then an efficient algorithm exists for problem A as well. This forms the foundation of NP-Completeness proofs and showcases the interconnectedness of problems within the NP-Complete class [4], [5].

DISCUSSION

The concept of NP-Completeness unravels a world of computational complexity, revealing the fascinating interplay between solvability and verification. It stems from the recognition that some problems pose a dual challenge: while verifying solutions is relatively straightforward, finding those solutions efficiently remains elusive. This dichotomy exposes the boundaries of computational tractability and prompts a deep exploration of problem classes and their inherent intricacies.

One of the cornerstones of NP-Completeness is the concept of polynomial-time reductions. These reductions serve as bridges between problems, highlighting their computational equivalence. A reduction from problem A to problem B asserts that if problem B can be solved efficiently, so can problem A. This web of reductions forms the backbone of NP-Completeness proofs, demonstrating the interconnectedness of problems within this class.

The question of P vs. NP, an unsolved enigma, encapsulates the heart of computational complexity theory. It asks whether problems that can be efficiently verified can also be efficiently solved. The distinction between P (problems solvable in polynomial time) and NP has profound implications for the limits of computation and the scope of human understanding in addressing complex problems.

Real-World Impact:

The real-world implications of NP-Completeness reverberate across industries and disciplines. NP-Complete problems serve as benchmarks for computational hardness, helping define the boundaries of what is achievable within realistic timeframes. The Traveling Salesman Problem, for instance, finds applications in route optimization, DNA sequencing, and logistics. The Knapsack Problem influences resource allocation in finance and inventory management. The Boolean Satisfiability Problem plays a pivotal role in circuit design and artificial intelligence planning. In the vast landscape of computational theory, the concepts of NP-Completeness and Computational Intractability stand as towering constructs that illuminate the complex interplay

between problem solvability and the limits of computation. These concepts form the foundation of our understanding of the inherent complexity of certain problems, offering insights into the daunting challenges posed by decision problems that seem to defy efficient solutions. As we embark on a journey through the depths of NP-Completeness and computational intractability, we uncover their significance, implications, and the intricate web of connections that define the boundaries of computation.

Understanding NP-Completeness: A Journey into Complexity

NP-Completeness arises from a fundamental question: Can every problem that can be verified efficiently also be solved efficiently? This question lies at the heart of the distinction between two classes of problems: P and NP. The class P encompasses problems that can be solved in polynomial time, signifying their tractability within reasonable timeframes as input size grows. On the other hand, the class NP consists of problems for which potential solutions can be verified quickly, even though finding the solutions themselves efficiently remains a puzzle.

Within the class NP, there exists a subset of problems that are exceptionally complex. These problems, known as NP-Complete problems, possess a unique property—they are believed to be as hard as the hardest problems in NP. To demonstrate that a problem is NP-Complete, one must prove both its membership in NP and its transformation into an already-established NP-Complete problem through polynomial-time reductions. This process of reduction unveils a fascinating web of interconnected problems, demonstrating that the complexity of NP-Complete problems is intrinsically linked [6]–[8].

The concept of NP-Completeness presents an elegant yet daunting challenge to the realm of computation. It reveals the profound interconnectedness of problems and lays the groundwork for understanding the limits of what can be achieved within realistic timeframes. The famous question of whether P equals NP, or more specifically, whether problems that can be efficiently verified can also be efficiently solved, remains an open problem that holds the key to unlocking the mysteries of computation's boundaries.

Polynomial-Time Reductions: The Threads of Complexity

At the heart of NP-Completeness lies the art of polynomial-time reductions—a technique that connects problems and showcases their computational equivalence. A reduction from problem A to problem B asserts that if problem B can be solved efficiently, then problem A can also be solved efficiently. This relationship underpins the concept of NP-Completeness, allowing us to establish the complexity of new problems by relating them to already-known NP-Complete problems.

Polynomial-time reductions form the building blocks of NP-Completeness proofs. By crafting algorithms that transform instances of one problem into instances of another in polynomial time, we create a bridge between problems. The transitivity of reductions enables us to navigate the intricate web of interconnected problems, demonstrating their equivalency and complexity. This reductionist approach offers a methodical framework for tackling NP-Completeness, providing insights into the boundaries of computation and the class of problems that pose universal challenges.

Real-World Implications: NP-Completeness in Practice

While the realm of NP-Completeness may seem abstract, its implications reverberate across diverse fields and industries. Many problems encountered in real-world scenarios fall into the NP-Complete category, showcasing the inherent complexity of tasks that, on the surface, appear straightforward. For instance, the Traveling Salesman Problem, a classic NP-Complete problem, involves finding the shortest route that visits a set of cities and returns to the starting point. This seemingly simple problem has applications in logistics, DNA sequencing, and circuit design optimization.

Similarly, the Knapsack Problem, where items with varying weights and values must be selected to maximize value without exceeding a weight limit, impacts resource allocation, portfolio optimization, and financial decision-making. The Boolean Satisfiability Problem, dealing with the question of whether a given logical formula can be satisfied by assigning truth values to its variables, has crucial applications in circuit design verification, AI planning, and constraint satisfaction.

The ubiquity of NP-Completeness underscores the importance of understanding and grappling with computational intractability. These problems serve as benchmarks for hardness, helping delineate the boundaries of what can be feasibly computed. The challenges posed by NP-Complete problems influence algorithm design, optimization strategies, and the very fabric of computational problem-solving across various domains [9], [10].

P vs. NP: The Grand Enigma

One of the most profound questions in computational theory revolves around P vs. NP—a question that encapsulates the essence of NP-Completeness and the broader landscape of computational intractability. The question asks whether every problem that can be verified efficiently (in NP) can also be solved efficiently (in P). In essence, P vs. NP delves into the heart of whether computational complexity can be bridged with computational efficiency. Resolving the P vs. NP question would have far-reaching implications for computer science, cryptography, optimization, and many other fields. If P equals NP, it would imply that efficient algorithms exist for a range of problems that currently appear intractable, revolutionizing various industries. On the other hand, if P does not equal NP, it confirms the existence of problems that inherently resist efficient solutions, highlighting the robustness of NP-Completeness as a concept that reflects the boundaries of computation.

Navigating the Uncharted Seas of Complexity: A Concluding Reflection

In the tapestry of computational theory, NP-Completeness and Computational Intractability emerge as intricate threads that weave together the complexity of problem-solving and the limitations of computation. These concepts illuminate the challenges of verifying and finding solutions within reasonable timeframes, offering insights into the boundaries of what is achievable within the realm of algorithms. The concept of polynomial-time reductions showcases the interconnectedness of problems, revealing the web of complexity that underlies seemingly disparate challenges.

As technology advances and our understanding of NP-Completeness deepens, the real-world implications become increasingly profound. NP-Complete problems influence decision-making processes, optimization strategies, and the design of algorithms that power modern

computational systems. The question of P vs. NP continues to challenge the very nature of computation, urging us to explore the frontiers of complexity, efficiency, and the fundamental limits of human knowledge.

In the uncharted seas of computational intractability, the pursuit of understanding continues, shaping the trajectory of research and innovation. The world of algorithms and complexity theory is ever-evolving, driven by the curiosity to unravel the mysteries of complexity and efficiency. As we navigate this intricate landscape, we gain not only a deeper appreciation for the intricacies of problem-solving but also a profound recognition of the boundaries that define the remarkable field of computational theory.

CONCLUSION

In conclusion, NP-Completeness and Computational Intractability illuminate the intricate landscape of decision problems that perplex even the most sophisticated computational minds. The quest to understand the boundaries of efficient solvability has led to the formation of NP-Completeness as a cornerstone concept. The interconnectedness of NP-Complete problems through reductions serves as a testament to the complexity of computation itself.

As technology advances and our understanding of NP-Completeness deepens, the real-world implications become increasingly profound. NP-Complete problems touch every facet of human endeavor, from supply chain optimization to cryptography. The enigma of P vs. NP continues to challenge the very nature of computation, urging us to unravel the mysteries of complexity, efficiency, and the limits of human knowledge.

In the uncharted seas of computational intractability, the pursuit of understanding continues, paving the way for new discoveries, methodologies, and breakthroughs in our ever-evolving world of algorithms and complexity theory.

REFERENCES

[1] A. M. Tillmann, “On the computational intractability of exact and approximate dictionary learning,” *IEEE Signal Process. Lett.*, 2015, doi: 10.1109/LSP.2014.2345761.

[2] A. Atiia, C. Hopper, and J. Waldispühl, “Computational intractability generates the topology of biological networks,” in *ACM-BCB 2017 - Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, 2017. doi: 10.1145/3107411.3107453.

[3] I. Diakonikolas, G. Kamath, D. Kane, J. Li, A. Moitra, and A. Stewart, “Robust estimators in high-dimensions without the computational intractability,” *SIAM J. Comput.*, 2019, doi: 10.1137/17M1126680.

[4] C. Rojas and M. Yampolsky, “Computational intractability of attractors in the real quadratic family,” *Adv. Math. (N. Y).*, 2019, doi: 10.1016/j.aim.2019.04.033.

[5] M. Blokpoel, M. van Kesteren, A. Stolk, P. Haselager, I. Toni, and I. van Rooij, “Recipient design in human communication: Simple heuristics or perspective taking?,” *Front. Hum. Neurosci.*, 2012, doi: 10.3389/fnhum.2012.00253.

[6] I. van de Pol, I. van Rooij, and J. Szymanik, "Parameterized Complexity of Theory of Mind Reasoning in Dynamic Epistemic Logic," *J. Logic, Lang. Inf.*, 2018, doi: 10.1007/s10849-018-9268-4.

[7] I. van Rooij, C. D. Wright, J. Kwisthout, and T. Wareham, "Rational analysis, intractability, and the prospects of 'as if'-explanations," *Synthese*, 2018, doi: 10.1007/s11229-014-0532-0.

[8] A. T. Buba and L. S. Lee, "A differential evolution for simultaneous transit network design and frequency setting problem," *Expert Syst. Appl.*, 2018, doi: 10.1016/j.eswa.2018.04.011.

[9] P. Hennig and C. J. Schuler, "Entropy search for information-efficient global optimization," *J. Mach. Learn. Res.*, 2012.

[10] R. Hoshino and M. Notarangelo, "Computational intractability and solvability for the birds of a feather game," in *33rd AAAI Conference on Artificial Intelligence, AAAI 2019, 31st Innovative Applications of Artificial Intelligence Conference, IAAI 2019 and the 9th AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019*, 2019. doi: 10.1609/aaai.v33i01.33019648.

CHAPTER 25

A BRIEF DISCUSSION ONALGORITHM DESIGN TECHNIQUES AND PATTERNS

Dr. Trapty Agrawal, Associate Professor
Maharishi School of Engineering & Technology, Maharishi University of Information Technology, Uttar Pradesh, India
Email Id-trapty@muit.in

ABSTRACT:

Algorithm design is the cornerstone of efficient problem-solving, driving innovation across diverse domains. This chapter delves into Algorithm Design Techniques and Patterns, elucidating strategies that guide the creation of robust, optimized algorithms.

By exploring key techniques, such as Divide and Conquer, Dynamic Programming, Greedy Algorithms, and Backtracking, along with their real-world applications, readers gain insights into the art of crafting algorithms that tackle challenges effectively. This chapter also highlights the significance of these techniques in the landscape of computational problem-solving.

KEYWORDS:

Algorithm Design, Backtracking, Divide and Conquer, Dynamic Programming, Greedy Algorithms, Problem-Solving.

INTRODUCTION

In the realm of computational problem-solving, algorithm design stands as the driving force behind the creation of efficient solutions that power modern technology. Algorithm Design Techniques and Patterns offer structured approaches to crafting algorithms that address challenges across diverse domains. These techniques encapsulate a wealth of strategies, each tailored to specific problem characteristics and requirements [1]–[3].

Types, Characteristics, Applications, and Key Components:

Types of Algorithm Design Techniques:

Algorithm Design Techniques span a spectrum of methodologies, each catering to distinct problem scenarios.

1) Divide and conquer,
2) Dynamic Programming
3) Greedy Algorithms,
4) and Backtracking

These are key paradigms that form the foundation of algorithmic problem-solving.

Characteristics and Strengths:

Divide and Conquer entails breaking down complex problems into smaller, more manageable subproblems. This technique excels in scenarios where subproblems are similar and can be solved independently. Dynamic Programming, on the other hand, emphasizes the reuse of

solutions to overlapping subproblems, mitigating redundancy and enhancing efficiency. Greedy Algorithms prioritize locally optimal choices at each step, aiming for an overall optimal solution. Backtracking explores all possible solutions recursively, efficiently pruning unviable paths.

Applications in the Real World:

Divide and conquer finds application in sorting algorithms like Merge Sort and Quick Sort, as well as in advanced data structures like B-Trees. Dynamic Programming drives solutions for tasks like optimal pathfinding in graphs, sequence alignment in bioinformatics, and resource allocation. Greedy Algorithms optimize activities such as scheduling, Huffman coding, and minimum spanning tree construction. Backtracking is pivotal in solving problems like the N-Queens puzzle, Sudoku, and constraint satisfaction.

Key Components of Algorithm Design Patterns:

The key components of Algorithm Design Patterns revolve around the unique strategies employed by each technique. Divide and Conquer hinges on breaking problems into smaller parts, solving them, and combining solutions. Dynamic Programming involves defining subproblems, establishing recurrence relations, and storing solutions in a table for reuse. Greedy Algorithms demand a greedy choice strategy and a proof of optimality. Backtracking entails systematically exploring possible solutions, employing constraints, and efficiently pruning unfeasible paths [4], [5].

DISCUSSION

Algorithm Design Techniques and Patterns epitomize the art of crafting efficient solutions to computational challenges. Divide and conquer, with its recursive breakdown of problems, offers elegant solutions to tasks like sorting and searching. Dynamic Programming's optimal substructure and overlapping subproblems property enable efficient solutions to optimization problems. Greedy Algorithms balance local and global optimization, providing efficient solutions in scenarios where greedy choices lead to overall optimality. Backtracking's exhaustive exploration is pivotal in solving constraint-based problems where systematic search is the key.

Divide and conquer, exemplified by Merge Sort, illustrates the power of problem decomposition. By breaking sorting into smaller tasks, efficiency is achieved through parallel processing. Dynamic Programming shines in applications like sequence alignment, where reusing optimal solutions leads to computational efficiency. Greedy Algorithms, seen in Huffman coding, demonstrate the power of incremental decision-making, optimizing data compression. However, these techniques aren't without limitations. Divide and Conquer may suffer inefficiencies due to the overhead of subproblem creation. Dynamic Programming's memory demands can be substantial, and it requires problems to exhibit overlapping substructures. Greedy Algorithms can sometimes lead to suboptimal solutions if the greedy choice doesn't always yield the best global solution. Backtracking, while exhaustive, can be computationally intensive for complex problems.

Real-World Impact:

Algorithm Design Techniques and Patterns are ubiquitous across industries. Financial optimization employs greedy algorithms for tasks like portfolio rebalancing. Dynamic Programming optimizes travel routes in maps, improving delivery logistics. Divide and Conquer

streamlines data processing in databases and file systems. Backtracking is instrumental in AI decision-making and solving puzzles. Algorithm design is the beating heart of computational problem-solving, driving innovation and progress across a myriad of domains. Within this landscape, Algorithm Design Techniques and Patterns serve as the guiding stars, offering structured approaches to crafting algorithms that effectively tackle complex challenges. By delving into key techniques such as Divide and Conquer, Dynamic Programming, Greedy Algorithms, and Backtracking, along with their real-world applications, we embark on a journey to understand the art of algorithmic creation that optimizes efficiency and fosters innovation [6]–[8].

Divide and Conquer: Decomposition for Elegance and Efficiency

Divide and conquer is a timeless technique that hinges on breaking down intricate problems into smaller, more manageable subproblems. This approach follows the age-old wisdom that tackling smaller tasks can lead to a more efficient solution for the overall problem. One of its shining exemplars is the Merge Sort algorithm, which demonstrates how dividing an unsorted array into smaller subarrays, sorting them, and merging them can lead to an optimized sorting process. By exploiting parallel processing capabilities, Divide and Conquer accelerates the computation of solutions. Divide and Conquers elegance lies in its ability to transform a seemingly complex problem into a series of more digestible subproblems. It's particularly potent when subproblems share structural similarities, allowing for efficient recombination of solutions. This technique shines in scenarios where solving smaller parts is more straightforward than solving the entire problem at once.

Dynamic Programming: Memorization and Optimization Unleashed

Dynamic Programming, often referred to as "smart brute force," tackles optimization problems by reusing solutions to overlapping subproblems. It hinges on breaking down a problem into smaller subproblems and storing their solutions in a table for later use. As a result, redundant calculations are avoided, and optimization is achieved. Classic examples of Dynamic Programming include the Fibonacci sequence and the Longest Common Subsequence problem. One of Dynamic Programming's defining characteristics is its emphasis on optimal substructure and overlapping subproblems. It's ideally suited for problems that can be broken down into smaller, related subproblems. This technique harnesses the power of memorization, enabling the efficient reuse of previously calculated solutions to build towards the overall optimal solution. However, the technique's memory demands can be substantial, and its applicability depends on the presence of these overlapping subproblems.

Greedy Algorithms: Balancing Local and Global Optimality

Greedy Algorithms follow a simple yet powerful mantra: make the locally optimal choice at each step to ultimately achieve a globally optimal solution. This technique prioritizes immediate gains, trusting that making the best choice in the current moment will lead to the best overall outcome. Classic examples of Greedy Algorithms include Huffman coding for data compression and the Minimum Spanning Tree construction. The strength of Greedy Algorithms lies in their efficiency and simplicity.

They're particularly effective when the problem at hand can be broken down into a sequence of decisions, and each decision contributes incrementally to the final solution. However, Greedy Algorithms don't guarantee global optimality in all scenarios. There are cases where the locally optimal choices don't lead to the best overall solution, highlighting the importance of carefully choosing problems that align with the greedy strategy.

Backtracking: Systematic Exploration of Solution Space

Backtracking, often likened to a "brute force with intelligence," explores all possible solutions by iteratively trying various options and abandoning paths that prove infeasible. This technique is particularly suited for problems with constraints, where solutions need to satisfy specific conditions. Examples of Backtracking applications include solving puzzles like the N-Queens problem and the Sudoku puzzle. Backtracking's hallmark is its exhaustive exploration of solution space. It's akin to traversing a labyrinth, testing different paths until the solution emerges or the infeasibility of all paths becomes evident. The efficiency of Backtracking lies in its ability to prune unviable paths early, narrowing down the search space. While it's a powerful technique, its computational intensity can grow quickly with problem complexity, making careful optimization and constraint management crucial.

Applications in the Real World: Impact and Significance

Algorithm Design Techniques and Patterns extend far beyond theoretical exercises—they form the backbone of modern technology and innovation. Divide and Conquer, as seen in Quick Sort and Merge Sort, optimizes data processing and management. Dynamic Programming fuels optimal route planning in maps, improving delivery logistics. Greedy Algorithms optimize resource allocation, influence data compression techniques, and enhance network infrastructure. Backtracking is instrumental in AI decision-making, solving puzzles, and constraint satisfaction problems. The real-world impact of these techniques spans industries and disciplines. Finance leverages Greedy Algorithms for portfolio optimization, while Dynamic Programming optimizes supply chain management. E-commerce platforms employ Divide and Conquer to manage vast datasets, and Backtracking plays a pivotal role in AI-driven recommendation systems. The significance of these techniques lies in their ability to convert complex real-world challenges into structured computational solutions [9], [10].

Navigating the Design Landscape

Algorithm Design Techniques and Patterns are the compass that guides the journey through the realm of computational challenges. These structured strategies provide systematic approaches to solving problems, each catering to specific problem characteristics. From Divide and Conquer's elegance to Dynamic Programming's reuse of solutions, from Greedy Algorithms' local optimization to Backtracking's systematic exploration, these techniques illuminate the path to optimized solutions. As technology advances and complexity grows, the significance of Algorithm Design Techniques becomes even more pronounced.

The ability to identify the right technique for a problem is a hallmark of adept algorithm designers. These techniques transcend domains, forming the bedrock of computational innovation. By mastering these techniques, individuals wield the power to navigate the ever-evolving landscape of algorithmic problem-solving with finesse and strategic prowess

CONCLUSION

Algorithm Design Techniques and Patterns serve as a compass in the complex landscape of computational problem-solving. They empower us to navigate through challenges systematically, crafting solutions that optimize efficiency and drive innovation. From the elegance of Divide and Conquer to the ingenuity of Dynamic Programming, the strategic nature of Greedy Algorithms, and the exhaustive exploration of Backtracking, each technique caters to specific problem characteristics.

The mastery of these techniques is a hallmark of adept algorithm designers. Choosing the right technique for a given problem is an art that involves understanding the problem's nature, constraints, and desired outcomes. As technology advances and challenges grow in complexity, the significance of Algorithm Design Techniques becomes even more pronounced.

They are the tools that enable us to bridge the gap between computational theory and real-world problem-solving, ensuring that our technological advancements are not just innovative but also efficient and impactful.

By understanding and wielding these techniques, individuals and industries alike can traverse the intricate landscape of algorithmic problem-solving with strategic finesse and innovation-driven precision.

REFERENCES:

[1] Y. Probst, D. T. Nguyen, M. K. Tran, and W. Li, “Dietary assessment on a mobile phone using image processing and pattern recognition techniques: Algorithm design and system prototyping,” *Nutrients*, 2015, doi: 10.3390/nu7085274.

[2] S. Mahapatra, S. Panda, and S. C. Swain, “A hybrid firefly algorithm and pattern search technique for SSSC based power oscillation damping controller design,” *Ain Shams Eng. J.*, 2014, doi: 10.1016/j.asej.2014.07.002.

[3] M. Y. Mhawish and M. Gupta, “Software Metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns,” *SN Appl. Sci.*, 2020, doi: 10.1007/s42452-019-1815-3.

[4] M. R. Kellman, E. Bostan, N. A. Repina, and L. Waller, “Physics-Based Learned Design: Optimized Coded-Illumination for Quantitative Phase Imaging,” *IEEE Trans. Comput. Imaging*, 2019, doi: 10.1109/tci.2019.2905434.

[5] D. J. Lohan, E. M. Dede, and J. T. Allison, “Topology optimization for heat conduction using generative design algorithms,” *Struct. Multidiscip. Optim.*, 2017, doi: 10.1007/s00158-016-1563-6.

[6] P. B. T. Kumbara and M. A. I. Pakereng, “Perancangan Teknik Kriptografi Block Cipher Berbasis Pola Permainan Tradisional Rangku Alu,” *J. Tek. Inform. dan Sist. Inf.*, 2019, doi: 10.28932/jutisi.v5i2.1714.

[7] S. Hussain, O. Ameri Sianaki, and N. Ababneh, “A Survey on Conversational Agents/Chatbots Classification and Design Techniques,” in *Advances in Intelligent Systems and Computing*, 2019. doi: 10.1007/978-3-030-15035-8_93.

[8] "Brief Papers," *Brain Cogn.*, 1996, doi: 10.1006/brcg.1996.0066.

[9] S. Enshaeifar *et al.*, "Machine learning methods for detecting urinary tract infection and analysing daily living activities in people with dementia," *PLoS One*, 2019, doi: 10.1371/journal.pone.0209909.

[10] S. Jafar-Zanjani, S. Inampudi, and H. Mosallaei, "Adaptive Genetic Algorithm for Optical Metasurfaces Design," *Sci. Rep.*, 2018, doi: 10.1038/s41598-018-29275-z.