Advanced Data Structures and Algorithms

Learn how to enhance data processing with more complex and advanced data structures

Abirami A
Priya R L

bpb

Advanced Data Structures and Algorithms

Learn how to enhance data processing with more complex and advanced data structures

Abirami A
Priya R L

bpb

Advanced Data Structures and Algorithms

Learn how to enhance data processing with more complex and advanced data structures

Abirami A

Priya R L

www.bpbonline.com

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-792-0

www.bpbonline.com

Dedicated to

Our beloved students

&

Our family members and friends

About the Authors

- **Mrs. Abirami A** has completed her M.E (Information Technology) from Mumbai University, and is currently pursuing Ph.D. (Information Technology), from Noorul Islam Centre for Higher Education, Kumaracoil, Tamil Nadu. Her research interests are Computer Networks, Cyber security, Network Forensics and Analysis of algorithms. She has 17 years of teaching experience and has published many papers in national and international conferences and journals and work as an editorial board member of the Editorial Board of various SCI/SCOPUS indexed journals.

  Asst. Professor III - Information Technology Department,

  Bannari Amman Institute of Technology, Sathyamangalam, Erode, Tamil Nadu.

- **Mrs. Priya R. L** is currently working as an Assistant Professor in V.E.S Institute of Technology, Mumbai, having 18 years of teaching experience for undergraduate students of Computer Engineering and Information Technology disciplines in different Engineering Institutes. She obtained her Master's degree in Information Technology Engineering from Mumbai University. She has also worked as a Software Engineer in different firms in Chennai and Mumbai for 4 years. Her research interest is more into Artificial Intelligence, Internet of Things (IoT), Software engineering, Data Structures, Data Mining and Next Generation Networks. She has published more than 40 papers in various reputed SCI / SCOPUS indexed journals and international conferences.

  Assistant Professor, Department of Computer Engineering,

  V.E.S. Institute of Technology, Mumbai, India.

About the Reviewers

- **Mrs. Lifna C S**, Assistant Professor at the Vivekanand Education Society's Institute of Technology, Mumbai. Her research interests are Big Data Analytics, Social Analytics, Machine Learning, UX Design, Blockchain. She has around 26+ research papers published in international conferences & journals. She is an Innovation Ambassador of VESIT - IIC (Institute Innovation Cell) formed under MHRD from 2019 onwards. She is also a member of the AI Research Group established at VESIT, in association with LeadingIndia.ai at Bennett University.

- **Dr. Lakshmanaprakash S** completed his Ph.D. from Curtin University, Sarawak, Malaysia in 2019. He is currently working as Associate Professor in the Department of IT, BIT, and Erode, India. He has published many papers in various reputed national/International conferences and Journals. Dr. Lakshmanaprakash has also been a reviewer in various SCI Journals. His research interest mainly focuses on Cybersecurity and Machine Learning.

Acknowledgements

We wish to express our gratitude to all those people who assisted and helped us to complete this book. We would like to show our appreciation to the members of our student team (Riya Bhanushali, Amisha Das, Asiya Khan, Disha Shah, Jay Visaria, and Mansi Thakkar of Shah and Anchor Kutchhi Engineering College, Mumbai, and Ajay Nair from Vivekanand Education Society's Institute of Technology, Mumbai). They helped us to understand their needs for the preparation of the contents associated with this book during the Engineering course. First and foremost, I want to express my gratitude to our friends and family members for their unwavering support to complete the book successfully. Without their encouragement, we would never have been able to finish it.

We owe a debt of gratitude to the student team who accompanied us in writing this book. We gratefully acknowledge Dr. Lakshmanprakash S and Mrs. Lifna C S for their kind technical scrutiny of this book.

Our gratitude also extends to the BPB team, who were kind enough to give us enough time to complete the book's first section and to approve its publication.

Preface

This book covers a collection of complex algorithms and helps to face the challenges in algorithmic analysis. Analysis of algorithms and handling sophisticated data structures focus on the fundamentals of the computer programming field. The book highlights how to find the best optimal solution to a real-world problem using an appropriate algorithm. The book provides theoretical explanations and solved examples of most of the topics covered.

This book also introduces the importance of performance analysis of an algorithm, which helps to increase efficiency and reduces time and space complexity. It shows how to create and design a complex data structure. This book solves the basic understanding of greedy and dynamic programming. It also gives importance to various divide-and-conquer techniques. This book gives information about string-matching methods as well.

This book is divided into six chapters. The reader will go through advanced data structures, greedy and dynamic programming, optimal solutions, string matching using various techniques, and calculations of time and space complexity using Asymptotic notations. To help learners better comprehend the material, each topic is handled with appropriate examples. The specifics are mentioned as follows.

Chapter 1 emphasizes the basics of algorithmic analysis. It will discuss the need for analysis of algorithms and help us to choose a better suitable algorithm for a given problem statement. In algorithmic design, the complexity of an algorithm plays an important aspect to justify the design decisions. Accordingly, algorithm efficiency is measured from two perspectives such as time and space complexity. Hence, the major focus of this chapter is on various types of asymptotic notations used for the estimation of the time complexity of an algorithm and is discussed with examples.

Chapter 2 discusses different complex data structures that may be used to effectively tackle difficult situations. AVL Tree, Huffman Coding, Redblack Tree, and several more search trees are among the advanced data structures addressed.

Chapter 3 discusses the divide and conquer technique with the basic introduction and various methods that are involved in it with suitable examples. Divide and Conquer is the simplest and easiest technique of decomposing a larger problem into simpler problems, to solve any given problem statement.

Chapter 4 will cover information about various greedy algorithms such as the knapsack problem, optimal merge pattern, subset cover problem, and so on, in detail with various solved examples.

Chapter 5 discusses Dynamic Programming. It describes various classical computer science problems and their optimal solutions using dynamic programming approaches along with their applications. The need for dynamic algorithms and introduction to NP-Hard & NP-Complete are discussed with examples.

Chapter 6 describes different string-matching algorithms with suitable examples. The chapter also provides description of genetic algorithms.

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/x0d84sg

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

CHAPTER 1

Analysis of Algorithm

Introduction

The chapter emphasizes the basics of algorithmic analysis. *Donald Knuth* defines the term *"analysis of algorithms"* as the common technique for theoretical estimation of required resources, that is used to provide a solution to any specific computational problem. It will discuss the need for analysis of algorithms and help us choose a more suitable algorithm for a given problem statement. In algorithmic design, complexity of an algorithm plays an important aspect in justifying the design decisions. Accordingly, algorithm efficiency is measured in two perspectives, such as time and space complexity. Hence, the major focus of this chapter is on various types of asymptotic notations used for the estimation of time complexity of an algorithm and is discussed with examples.

Structure

In this chapter, we will discuss the following topics:

- Analysis of algorithm
- Asymptotic Notations
- Time Complexity
- General Rules for time complexity calculation
- Recurrences

Objectives

This chapter discusses the basics of analysis of algorithm, the need for analysis of algorithms as well as the various notations, with examples. Apart from these, time complexity calculation is the major content focused on, in the chapter.

Analysis of algorithm

In this section, we give an overview of a generic framework on analysis of algorithms. It analyzes the efficiency of algorithms, in terms of space efficiency and time efficiency. To represent the complexity of an algorithm, asymptotic notations are a very crucial and important factor in designing algorithms. Therefore, various notations with solved examples are illustrated here.

Space complexity is defined as the amount of memory space required to execute an algorithm. Sometimes, space complexity is ignored as the space used is minimal. But time complexity refers to the amount of time required for the computation of an algorithm. Mostly, execution time depends on the various properties such as disk input/output speed, CPU speed, instructor set and so on. The calculation of time complexity and its rules with solved examples are also described in this chapter. It is followed by the recurrences in algorithm analysis and its three major types are discussed in the later sections of this chapter.

What is analysis of algorithms?

In general terms, analysis of algorithms discusses the efficiency of an algorithm. It tells the estimation of various resources required by an algorithm to crack a specific problem of computation. The resources are the necessary storage or the required time to execute a specific algorithm. The estimated running time of an algorithm is called time complexity and the estimated storage/memory needed for the execution of an algorithm is called space complexity.

Why to analyze algorithms?

Multiple solutions are available for a single problem; analysis will present the best algorithm to solve the given problem out of the multiple solutions. Even though the objective of the algorithms is to generate the expected output, the ways in which the outputs are generated, are different. The algorithm varies in the time and the space complexity. The various cases of analysis such as the worst, best and the average are performed with the help of asymptotic notations, to finalize the best algorithm.

Asymptotic notations

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms, that does not depend on machine specific constants, and neither requires algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

Θ Notation

In this *theta* notation, the function bounds between the upper and the lower bound, and so it is called as an average case of T(n).

Average-case T(n) = average expected time of algorithm over all inputs of size n.

$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c_1, c_2, \text{and } n_0 \text{ s.t.} \\ c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Figure 1.1 features the average case Theta Θ Notation:

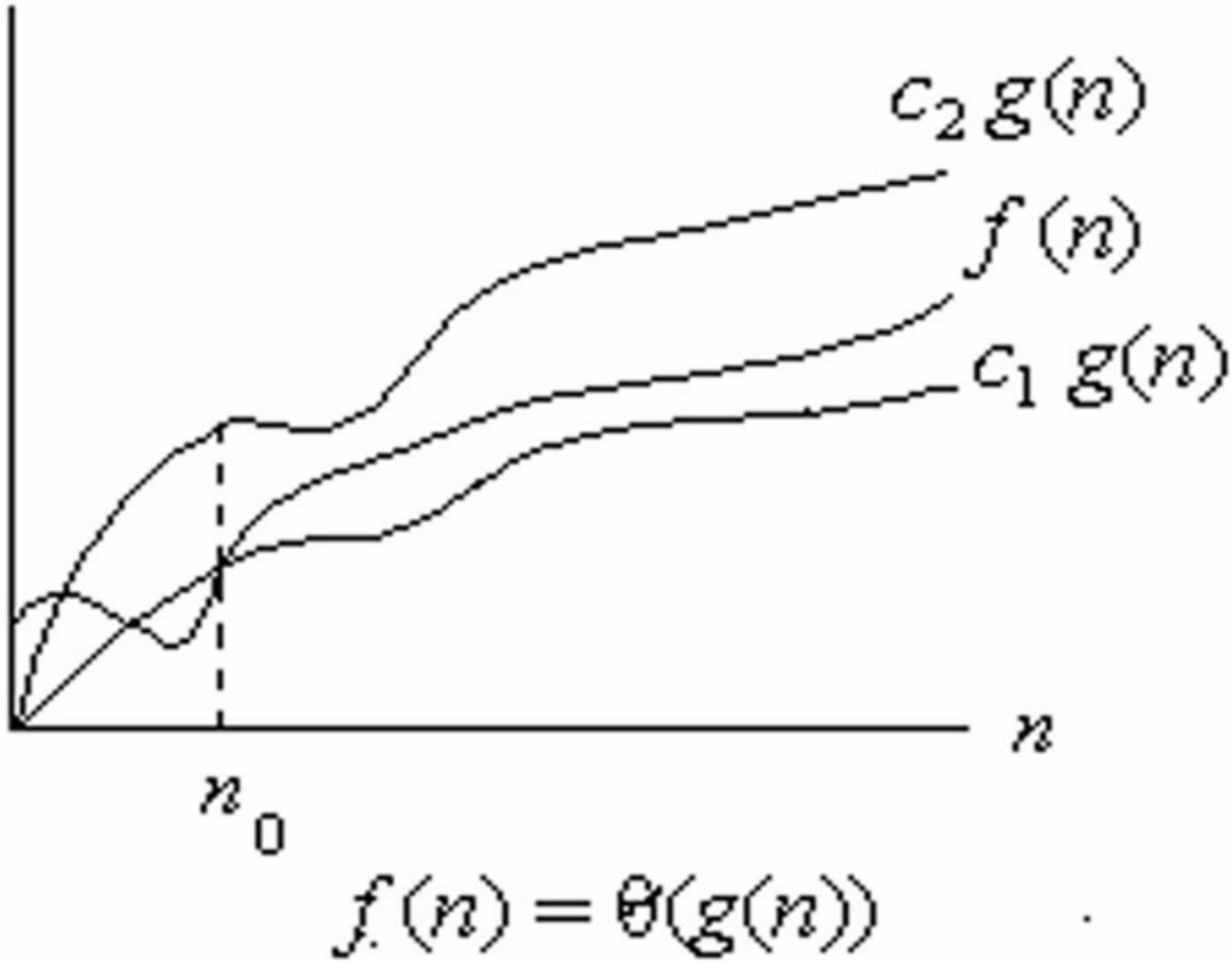


Figure 1.1: *Average case Theta Θ Notation*

Big O Notation

In this Big O notation, the function defines an upper bound, and so it is called the worst case of T(n).

$$T(n) = \text{maximum time of algorithm on any input of size } n.$$

$$O(g(n)) = \left\{ \begin{array}{l} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Figure 1.2 features the worst-case Big O Notation:

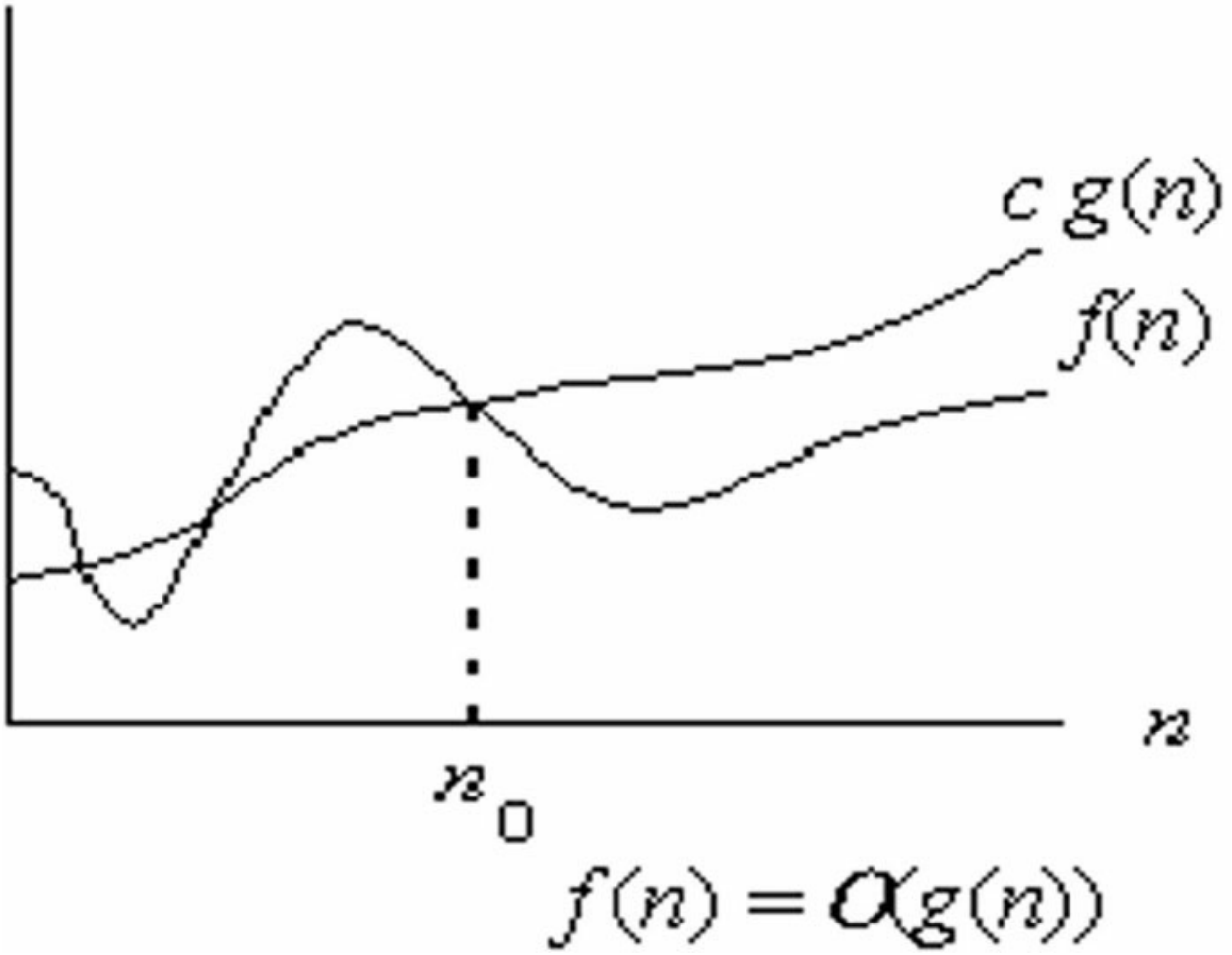


Figure 1.2: *Worst case Big O Notation*

Ω (Omega) Notation

In this Ω notation, the function defines a lower bound, and so it is called the best case of T(n).

T(n) = minimum time of algorithm on any input of size n.

It is a slow algorithm that works fast on some input.

$$\Omega(g(n)) = \left\{ \begin{array}{c} f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ s.t.} \\ cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Figure 1.3 features the best-case Omega Ω Notation:

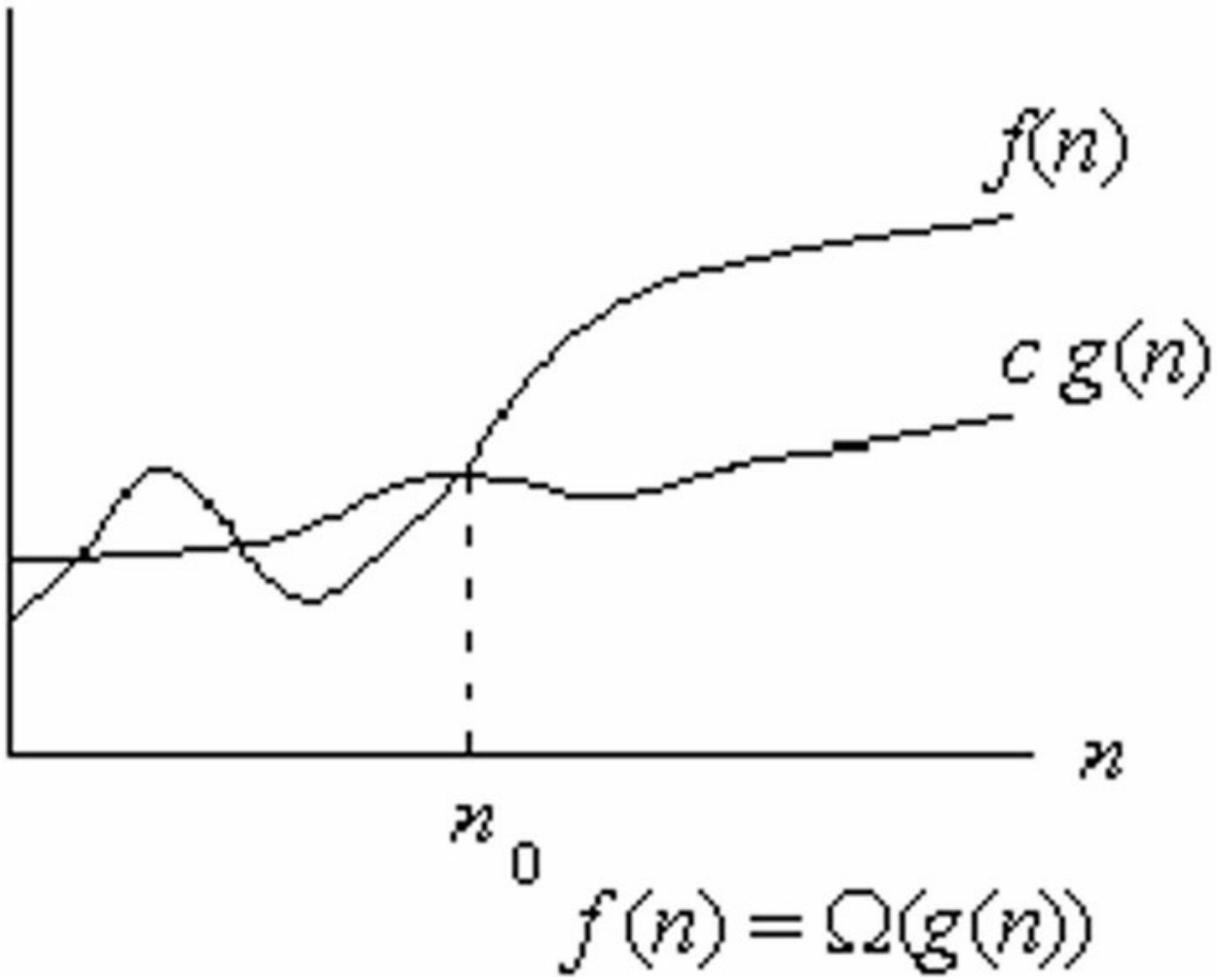


Figure 1.3: *Best case Omega Ω Notation*

Table 1.1 features the Asymptotic notations:

Asymptotic Notations	Case	Function Bound
Big O	Worst Case	Upper bound
Omega Ω	Best Case	Lower Bound
Theta Θ	Average Case	Tight Bound

Table 1.1: *Asymptotic Notations*

Example 1

Find Upper Bound, Lower Bound and Tight Bound for the following function 2n+5.

Solution 1:

Consider f(n)=2n+5,

g(n)=n {consider the highest degree of f(n)}

Refer to *Table 1.2*:

Lower Bound	Tight Bound	Upper Bound
Omega Ω	Theta Θ	Big O
Best Case	Average case	Worst case
2n	2n	3n

Table 1.2: *The constant value consideration for the asymptotic notations*

Big O:

*f(n)<=c*g(n)*

*2n+5<=c*n*

C=3// as per Table 1.2

*2n+5<=3*n*

For n=1, 7<=3 → False

For n=2, 9<=6 → False

For n=3, 11<=9 → False

For n=4, 13<=12 → False

For n=5, 15<=15 → True

Therefore f(n)=O(g(n))

2n+5=O(n) for all n>=5, C=3

Omega (Ω):

*f(n)>=c*g(n)*

*2n+5>=c*n*

C=2

*2n+5>=2*n*

For n=1, 7>=2 → True

Therefore f(n)= Ω(g(n))

2n+5= Ω(n) for all n>=1, C=2

Theta (Θ):

*C1*g(n) <= f(n) <= c2*g(n)*

*C1*n <= 2n+5 <= c2*n*

C1=2, c2=3

*2*n <= 2n+5 <= 3*n*

For n=1, 2 <= 7 <=3 → false

For n=2, 4 <= 9 <=6 → False

For n=3, 6 <= 11 <=9 → false

For n=4, 8 <= 13 <=12 → false

For n=5, 10<= 15 <=15 → true

Therefore f(n)=Θ(g(n))

2n+5= Θ(n) for all n>=5, c1=2, c2=3

Example 2

Find Upper Bound, Lower Bound and Tight Bound for the following function 3n+2

Solution 2:

Big O:

f(n)=O g(n);

3n+2 =O(n) for all n>=2, c=4

Omega Ω :

F(n)= Ω g(n);

3n+2= Ω(n) for all n>=1, c=3

Theta Θ:

F(n)= Θ g(n);

3n+2=Θ(n) for all n>=2, c1=3 & c2=4.

Example 3

Find Upper Bound, Lower Bound and Tight Bound for the following function 3n+3

Solution 3:

Big O:

f(n)=O g(n);

$$3n+3 = O(n) \text{ for all } n>=3, c=4$$

Omega Ω:

F(n)= Ω g(n);

$$3n+3 = \Omega(n) \text{ for all } n>=1, c=3$$

Theta Θ:

F(n)= Θ g(n);

$$3n+3 = \Theta(n) \text{ for all } n>=3, c1=3 \ \& \ c2=4$$

Example 4

Find Upper Bound, Lower Bound and Tight Bound for the following function $10n^2+4n+2$

Solution 4:

Consider

$$f(n)= 10n^2+4n+2,\ g(n)=n^2 \ \{consider\ the\ highest\ degree\ of\ f(n)\}$$

Refer to *Table 1.3*:

Lower Bound	Tight Bound	Upper Bound
Omega Ω	Theta Θ	Big O
Best Case	Average case	Worst case
$10n^2$	$10n^2$	$11n^2$

Table 1.3: *The constant value consideration for the asymptotic notations*

Big O:

f(n)=O g(n);

$$10n^2+4n+2 = O(n^2) \text{ for all } n>=5, c=11$$

Omega Ω:

F(n)= Ω g(n);

$$10n^2+4n+2 = \Omega(n^2) \text{ for all } n>=1, c=10$$

Theta Θ:

F(n)= Θ g(n);

$$10n^2+4n+2=\Theta(n^2) \text{ for all } n>=5, c1=10 \& c2=11$$

Time complexity

Time complexity is not the execution time of a particular algorithm; it informs about the rate of growth of an algorithm based on the inputs. Refer to *Figure 1.4*:

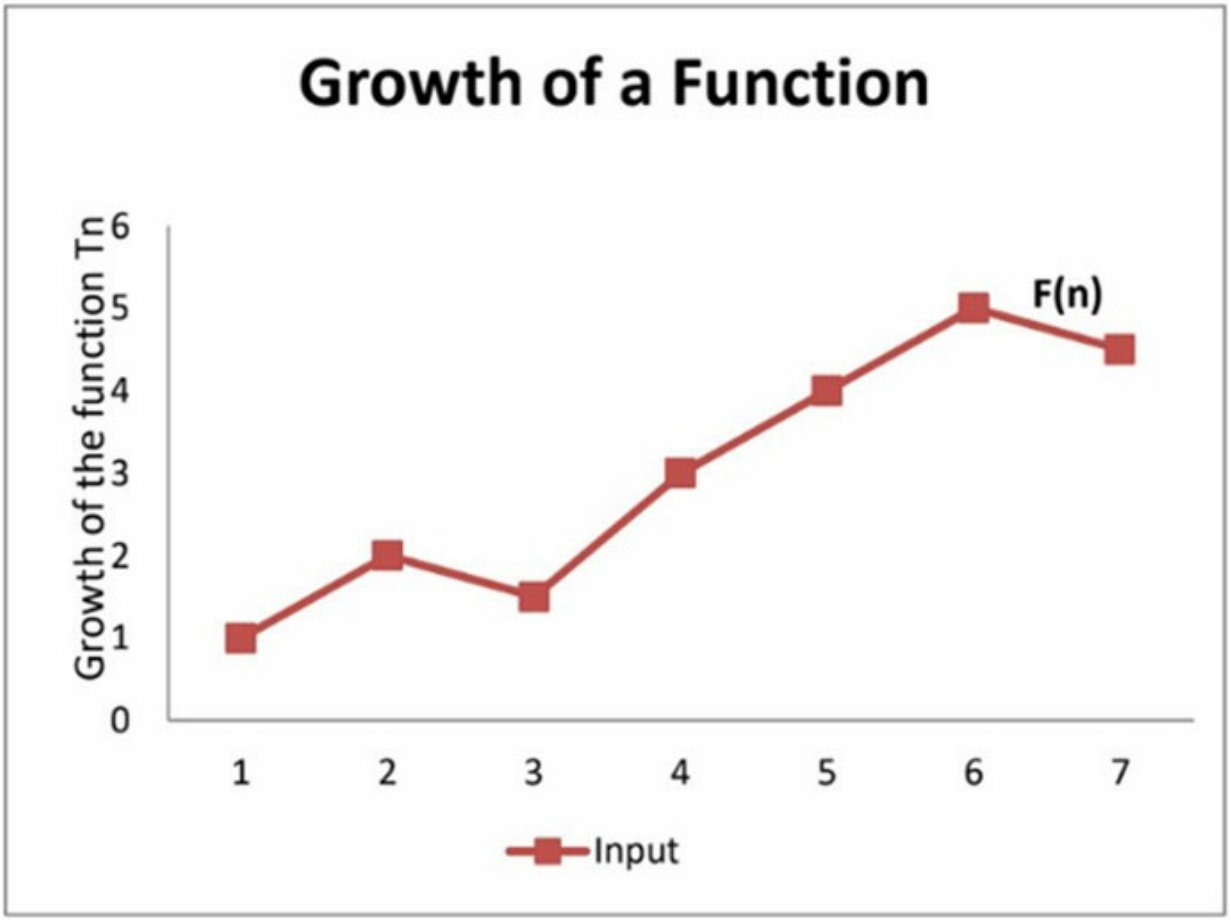


Figure 1.4: *Growth of a Function*

Consider the given example to calculate the time complexity of two different

methods, to solve the problem of identifying if the number is prime or not.

Prime number: A prime number cannot be formed by the product of smaller natural numbers, and it should be greater than 1.

For example, 7 is a prime number, as it is formed by the product of 1*7 and 7*1, and not formed by the product of smaller natural numbers.

Find if the given number is prime or not.

Algorithm 1	**Algorithm 2**
For i = 2 to n-1 *If i divides n* *n is not prime*	*For i = 2 to √n* *If i divides n* *n is not prime*

Assumptions: The computer takes one millisecond to perform the division operation.

Algorithm runs for n-2 times	**Algorithm runs for √n-1 times**
For n=7, *Takes 7-2=5ms* *For n=11,* *Takes 11-2=9ms* *For n=101,* *Takes 101-2=99ms* *For n=1000003,* *Takes ≈106 ms*	*For n=7;√7=2.6 [consider the integer value]* *Takes √7-1=2-1=1ms* *For n=11, √11=3.3* *Takes √11-1=3-1=2ms* *For n=101; √101=10.04* *Takes 10-2=9ms* *For n=1000003* *Takes ≈√106 ms≈103 ms*

From the analysis, it is clearly understood that algorithm 2 takes much less time to compute for larger input values. After computing the time complexity of both the algorithms, algorithm 2 runs efficiently. The time complexity of the program is how fast the output is generated for the given input, as the input size increases. The approximate time taken by algorithm 1 is proportional to n and is represented by *O(n)* in computational terms, which will be discussed later in this chapter. The approximate time taken by the algorithm 2 is proportional to √n and is represented by O(√n) in computational terms.

Calculating the time complexity

The running time of an algorithm depends on the following factors:

- Single vs multiprocessor
- Read/write speed of memory
- 32-bit architecture vs 64-bit architecture
- Input size

When we bother about the time complexity, all the preceding factors need not be considered. The factor that is very important is the input size, how an algorithm works for varying input, or the time taken by the algorithm for varying inputs. The basic consideration is the rate of growth of the time taken with respect to the input.

Calculating the rate of growth of an algorithm with respect to the input

For the calculation purpose, a model machine is to be considered with an assumption of Single processor, 32-bit architecture, and sequential execution of the Read/Write operation. The unit of time taken by the arithmetic and logical operations, assignment operations and return statement, are considered as one.

Example 5

Sum (a, b)

{

Return a + b

}

In *Example 5*, the return statement takes 1 unit of time and the addition operation takes 1 unit of time. So, the total time taken is 2 units.

As per this, the algorithm grows in constant time.

T(n)=1 ---------- *[1]*

Figure 1.5 features the rate of growth of algorithm:

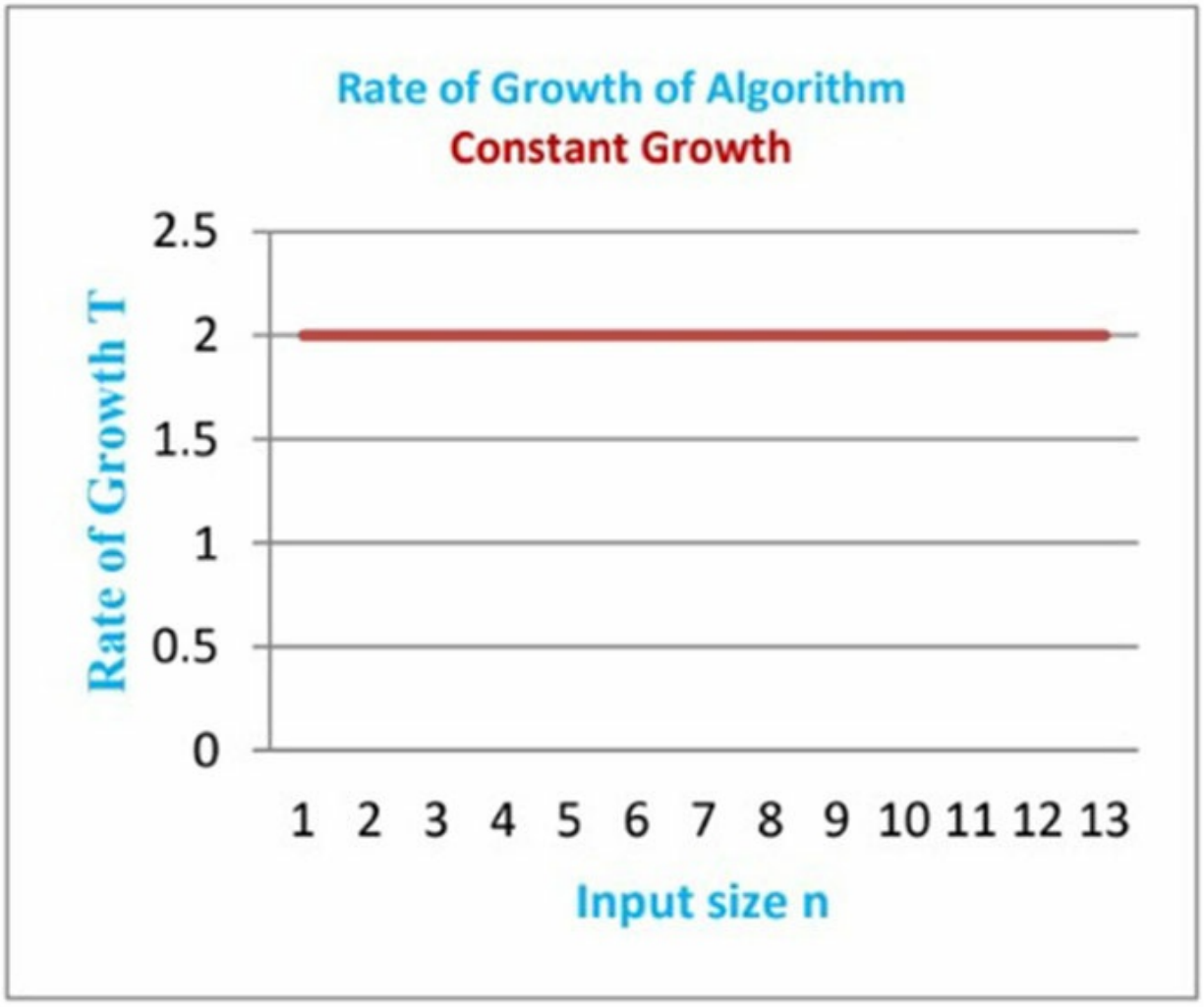


Figure 1.5: *Rate of growth of Algorithm for constant function- Example 5*

Example 6

Sum (Sum of elements of array A, n)

{

Statement No.	Statement Details	Cost of Execution	Time taken in units
1	Sum=0;	1 (C1)	1
2	For i=0 to n-1	2 (C2)	n+1
3	Sum=sum + A[i];	2 (C3)	n
4	Return Sum	1(C4)	1

}

In *Example 6*, the return statement 1 takes 1 unit of time and is executed once so that the total time taken is 1. In the statement 2 which is the “For” loop, there are 2 operation assignment and the increment, so the cost is 2 and is executed for n+1 times, which includes the false statement as well. Statement 3 which is an additional operation along with the assignment operation, takes the cost of execution as 2 and the time taken for the execution as n times. The 4^{th} statement is the return statement which is executed with the cost of 1 and the unit of time taken is also 1.

As per *Example 6*, the algorithm grows as follows,

$T(n) = 1+2(n+1) + 2n+1$

$= 4n + 4$

$= Cn + C` [C= C2+C3 \& C`=C1+C2+C4]$ ---------- *[2]*

As per equation [2], the growth of the algorithm is in the linear manner, as can be seen in *Figure 1.6:*

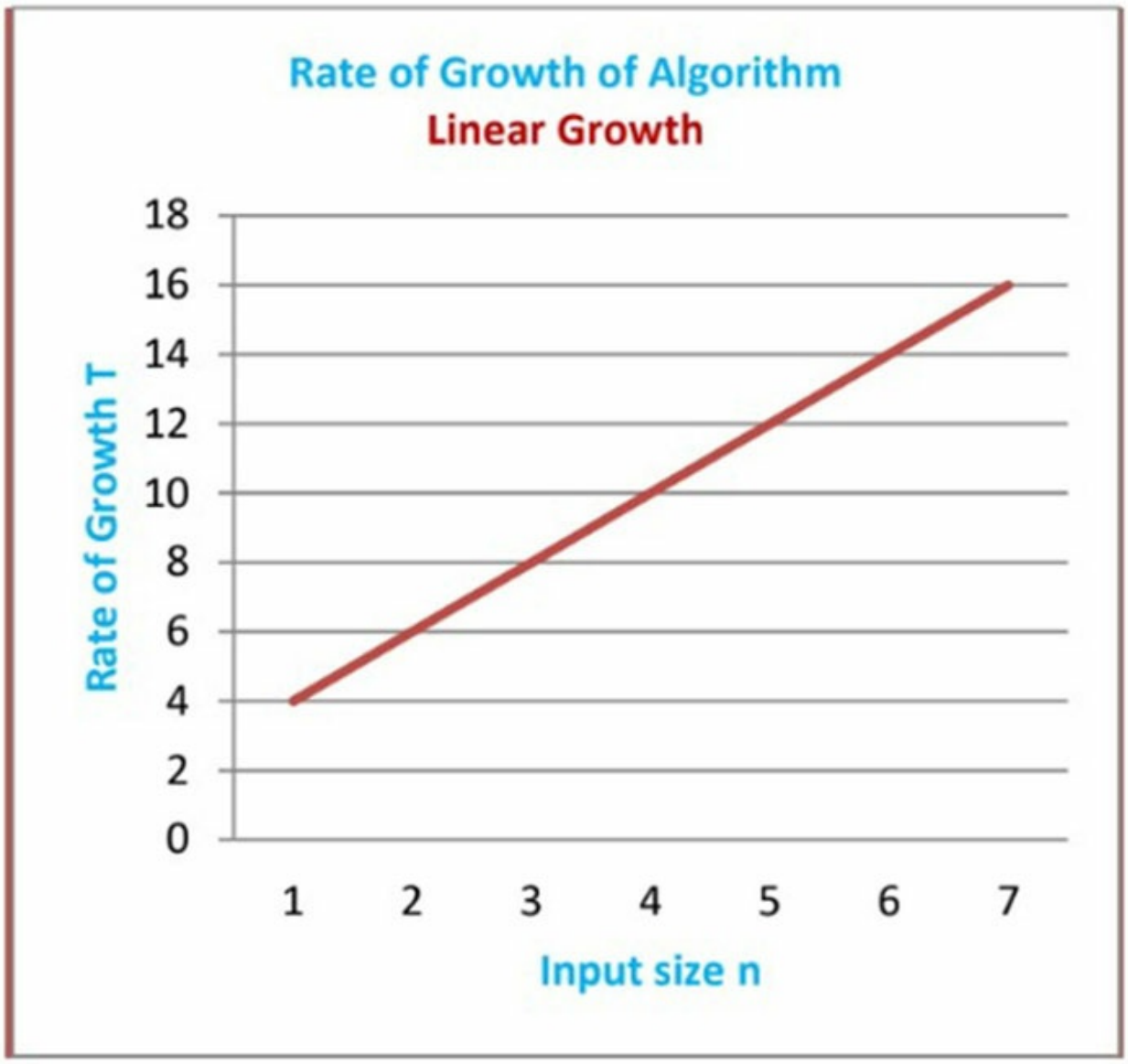


Figure 1.6: *Rate of growth of Algorithm for linear function - Example 6*

Example 7

For "Sum of Matrix "the expression will be

$T(n) = an^2 + bn + c$ ---------- *[3]*

As per equation [3], the growth of the algorithm is in the quadratic manner, as can be seen in *Figure 1.7:*

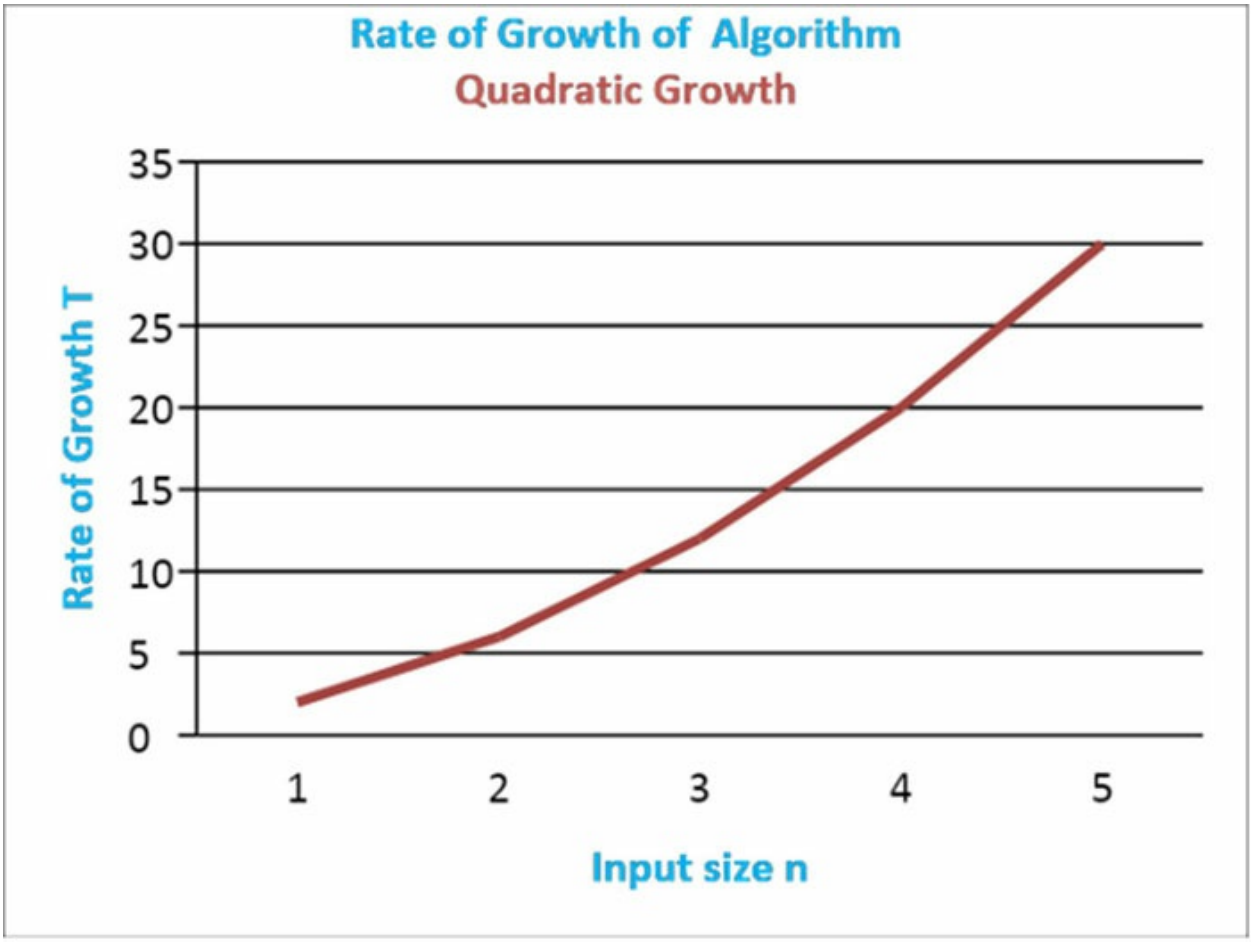


Figure 1.7: *Rate of growth of Algorithm - Example 7*

The other time complexities are discussed in the following *Table 1.4*:

Notation	Time Complexity
O(nc)	**Polynomial time**
O(log n)	**Logarithmic time**
O(n log n)	**Linearithmic**
O(2^n)	**Exponential time**
O(n!)	**Factorial time**

Table 1.4: *Asymptomatic notations and their complexities*

General rules for time complexity analysis

While calculating the time complexity, always consider a very large input size and the worst-case scenario.

Rule 1: Remove all lower order expressions & drop the constant multiplier

Example 8

$$T(n) = n^3 + 4n^2 + 7n + 2.$$

Applying Rule 1, T(n) is $\approx n^3$

In the worst-case scenario, the time complexity of the algorithm is **O(n^3)**.

Example 9

$$T(n) = 6n^2 + 9n + 3.$$

Applying Rule 1, T(n) is $\approx n^2$

In the worst-case scenario, the time complexity of the algorithm is **O(n^2).**

Example 10

$$T(n) = 5n + 1.$$

Applying Rule 1, T(n) is $\approx n$

In the worst-case scenario, the time complexity of the algorithm is **O(n).**

Rule 2: The running time of an algorithm is equal to the summation of running time of all the fragments of the algorithm.

$T(n)=\sum_0^n \quad fragments\ of\ the\ algorithm$

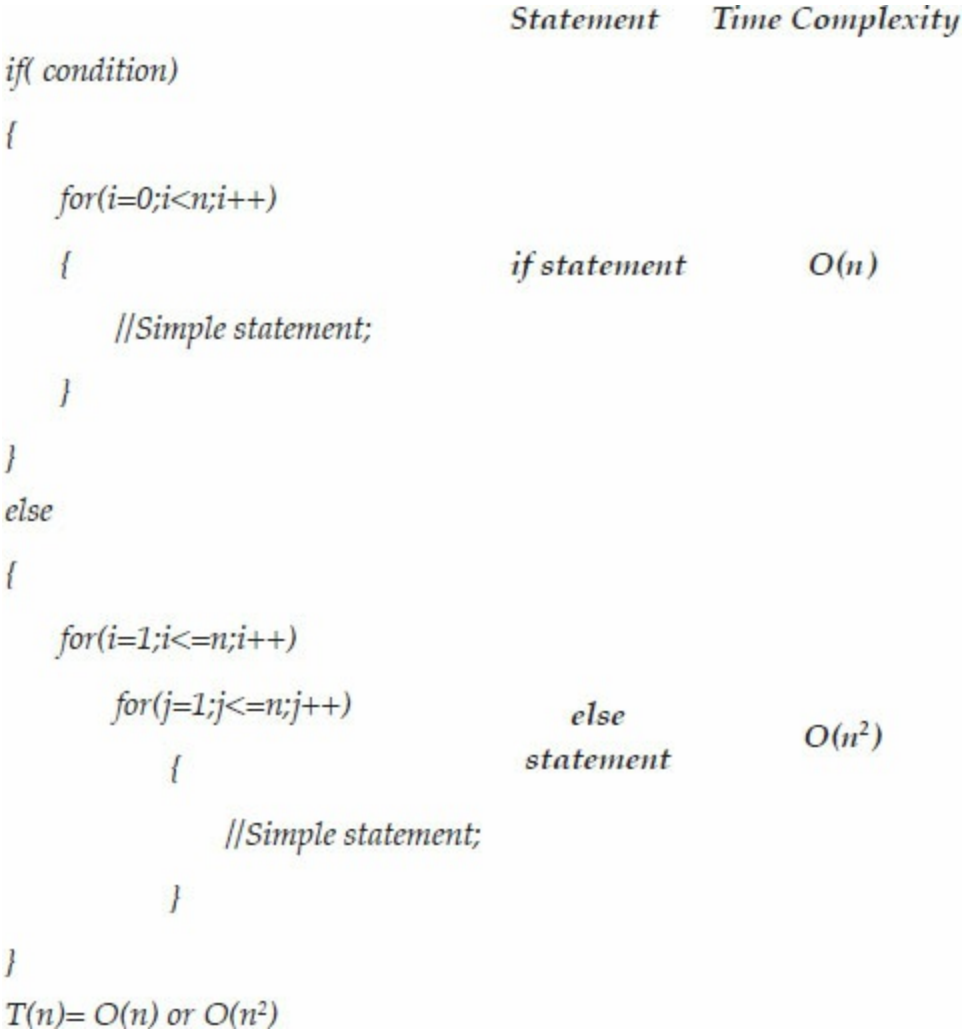
Example 11

Statement		Time Complexity
Fragment 1: *int a=10;*	*Simple statement*	O(1)
Fragment 2: *for(i=1;i<=n;i++)* *{* *//Simple statement;* *}*	*Simple loop*	O(n)
Fragment 3: *for(i=1;i<=n;i++)* *for(j=1;j<=n;j++)* *{* *//Simple statement;* *}*	*Nested loop*	$O(n^2)$

T(n)= O(1) + O(n) + $O(n^2)$// Sum of all the fragment's Time Complexity

Rule 3: In case of conditional statements; consider the complexity of the condition which is the worst case.

Example 12



Recurrences

A recurrence is a recursive description of a function, or in other words, a description of a function in terms of itself. A recurrence equation describes functions in terms of its value on smaller inputs. Recurrences generally use divide and conquer strategy. Let us consider the running time of an algorithm of size n as *T(n)*.

There are three types of recurrence methods:

- The Substitution method
- Recursive tree method
- Masters method

Substitution method

The **Substitution method** provides a condensed way of proving an asymptotic bound on recurrence by induction. We can use the substitution method to establish both upper and lower bounds on recurrences. This method is powerful, but it is only applicable to instances where the solutions can be guessed.

Example 13

$T(n) = 2\ T(\frac{n}{2}) + n\ ,\ T(1) = 1$

- Here ***T(n)*** is the run time of an algorithm. So, for the given information, we will use the substitution method.
- In the substitution method, we will use two columns: build solution and expand scratch.
- In build solution, we will be substituting the value of $T(\frac{n}{2})$ with the value returned from the expanded scratch.
- Initially, we will be substituting for $T(\frac{n}{2})$ in the build solution, the value for which will be fetched from scratch. In scratch solution, we will consider the equation, ***T(n) = 2 T(n/2) + n*** and in place of n, we will be substituting n/2. Hence, we get the result ***T(n/2) = 2 T(n/2^2) + (n/2)*** by substituting n with n/2.
- Now we will be substituting $T(\frac{n}{2})$ in the build solution from the result returned from the scratch solution. After substituting, we can see that the equation in the build solution becomes ***T(n) =2^2 T(n/2^2) + 2n*** highlighted as equation 1.
- Now in the equation, we have ***T(n/2^2)***. Hence, we will again be finding a value to substitute in its place, from the scratch solution. So again, consider the equation ***T(n) = 2 T(n/2) + n*** and in place of ***n*** substituted, ***n/2^2***.
- Once we get the result from scratch, substitute it in the build solution. Hence, after substitution, the equation becomes ***T(n) = 2^3 T(n/2^3) + 3n***.
- We can continue the same steps to substitute ***T(n/2^3)*** in the build solution. This process has to be done 3 or 4 times in the example.
- In this case, we have considered it 3 times. After substitution, we can

observe that it follows a specific pattern of substitution. For example, ***T(n) = 2 T(n/2) + n*** after substitution becomes ***2^2 T(n/2^2) + 2n***, that further, after substitution, becomes ***2^3 T(n/2^3) + 3n*** and that further after substitution becomes 2^4 T(n/2^4) + **4**n. So, we can observe that there is a number that always gets incremented after substitution. Hence, we can formulate a general formula as ***T(n) = 2^i T(n/2^i) + i.n***, where ***i*** is the number that gets incremented.

Solution 13:

Build Solution	**Expand Scratch**
$T(n) = 2\,T(n/2) + n$	$T(n/2) = 2\,T(n/22) + (n/2)$
$= 2\,[2\,T(n/22) + (n/2)] + n$	$T(n/22) = 2\,T(n/23) + (n/22)$
$= 22T(n/22) + 2n$ -------1	$T(n/23) = 2\,T(n/24) + (n/23)$
$= 22\,[2\,T(n/23) + (n/22)] + 2n$	
$= 23T(n/23) + 3n$ ---------2	
$= 23\,[2\,T(n/24) + (n/23)] + 3n$	
$= 24T(n/24) + 4n$ ----------3	

This implies $T(n) = 2^i\,T(n/2^i) + i.n$ ------ *4*

- After formulating the general formula, we will consider ***T(1) = 1***. Therefore, ***(n/2^i)*** from equation 4, will also be considered as 1.
- Hence, we will solve for ***(n/2^i) = 1*** and try to find a solution for which we took as an incremented variable previously.
- Hence, we get ***i = $\log^2 n$***. We will be substituting the value of ***i*** in the general formula, which we formulated, that is, equation 4.
- After substituting and solving the equation, we will get ***T(n) = n + n $\log^2 n$***, where the ***n*** before the addition sign can be ignored, as it is very small as compared to ***n $\log^2 n$***. Hence, the time complexity becomes ***O(n log n)***.

Now, *T(1) = 1*

Therefore, (n/2i) = 1

n = 2i

$i = log_2 n$

From equation 4,

$T(n) = 2log_2 n\ T(1) + (n)\ log_2 n$ …… *(Since we considered (n/2) = 1)*

$= n\ log_2^2\ (1) + n\ log_2 n$

$= n + n\ log_2 n$

This states that the complexity is ***O(n log n)***

Master theorem

Master theorem is used in calculating the time complexity of recurrence relations (divide and conquer algorithms) in a simple and quick way. It provides an asymptotic analysis of an algorithm.

There are 3 cases in the Master theorem that need to be considered and the equation that can be solved using the Master theorem will be of the form ***T(n) = a T(n/b) + f(n)***. Such types of equations can be solved using the Master theorem. The three cases which need to be checked are stated as follows.

General format

$T(n) = a\ T(n/b) + f(n)$

Where,

$a >= 1$

$b > 1$

f(n) should be positive.

There are three cases possible.

Case I:

$f(n) < n^{log_b a}$ *then*

$T(n) = Ø(n^{log_b a})$

Case II:

$f(n) = n^{log_b a}$ *then*

$T(n) = Ø(\ n^{log_b a}\ log\ n)$

Case III:

$f(n) > n^{\log_b a}$ *then*

$T(n) = Ø(f(n))$

Example 14

$T(n) = 2\ T(\frac{n}{2}) + n$

- This equation is of the form $T(n) = a\ T(n/b) + f(n)$, where a=2, b=2 and f(n)=n.

Now we will check for the 3 cases mentioned previously.

- **Case I:** $f(n) < n^{\log_b a}$. Therefore, in our case $n < n^{\log_2 2}$, that is, $n<n$ *(since* $\log_2 2 = 1$*)*. But the $n<n$ condition does not hold true. Hence, this case cannot be used.
- **Case II:** $f(n) = n^{\log_b a}$, therefore in our case $n = n^{\log_2 2}$, that is, $n=n$ *(since* $\log_2 2 = 1$*)*. The $n=n$ condition also holds true. Hence, this case can be used.
- **Case III:** $f(n) > n^{\log_b a}$, therefore in our case $n > n^{\log_2 2}$, that is, $n>n$ *(since* $\log_2 2 = 1$*)*. However, $n>n$ condition does not hold true. Hence, this case cannot be used.
- Hence in our equation only case II holds true. Therefore, we will be using case II for the solution of this problem.
- From case II, the run time of an algorithm becomes $T(n) = Ø(n^{\log_b a} \log n)$. Therefore, run time $T(n) = Ø(n^{\log_2 2} \log n)$, that is, $T(n) = Ø(n \log n)$.

Solution 14:

Here a=2 and b=2

$f(n) = n$

Therefore, $n^{\log_b a} = n^{\log_2 2}$

$f(n) = n^{\log_b a}$ ----------*Case II*

Therefore, $T(n) = Ø(n^{\log_b a} \log n)$

$T(n) = Ø(n \log n)$

Recursive Tree Method

A **recursion tree** shows the relationship between calls to the algorithm. Each item in the tree represents a call to the algorithm. A recursive algorithm can be developed in two main steps. Step 1 is developing the base part and Step 2 is developing the recursive part. Recursive Tree can be most used to generate a good guess, which can be cross checked if required, using the substitution method.

Example 15

$T(n) = 3\ T(n/4) + n^2$

- In this equation, $f(n) = n^2$ and we will be having 3 branches of it since we have $\boldsymbol{3}T(n/4)$ and n will be substituted by $n/4$.
- Now as we can observe in the following *Figure 1.8*, n^2 has three branches and n has been replaced by $n/4$.
- Now for each $(n/4)^2$, we will be having 3 branches and n will be replaced by $n/4$. Hence, here we already have $n/4$, therefore it will be $(n/4*4) = n/14$, that is, substituting $n/4$ for the n in the numerator.
- Now we have $(n/16)^2$ and each of it will have 3 branches. This recursive tree will have branches recursively until we have $T(1)$.
- Now we will be formulating a general formula from the tree. From the tree we can observe we have our root node as n^2, three $(n/4)^2$ branches, nine $(n/16)^2$ branches, will be having twenty-seven $(n/64)^2$ branches and so on.
- Adding all of them, a general formula needs to be formulated. After adding and solving the equation as done in the following *figure*, we have got the result $T(n) = 16n^2/13$. Here, we are ignoring 16/13 as it is a very small value as compared to n^2. Hence, we will have time complexity as $O(n^2)$, that is $T(n) = O(n^2)$.

Solution 15:

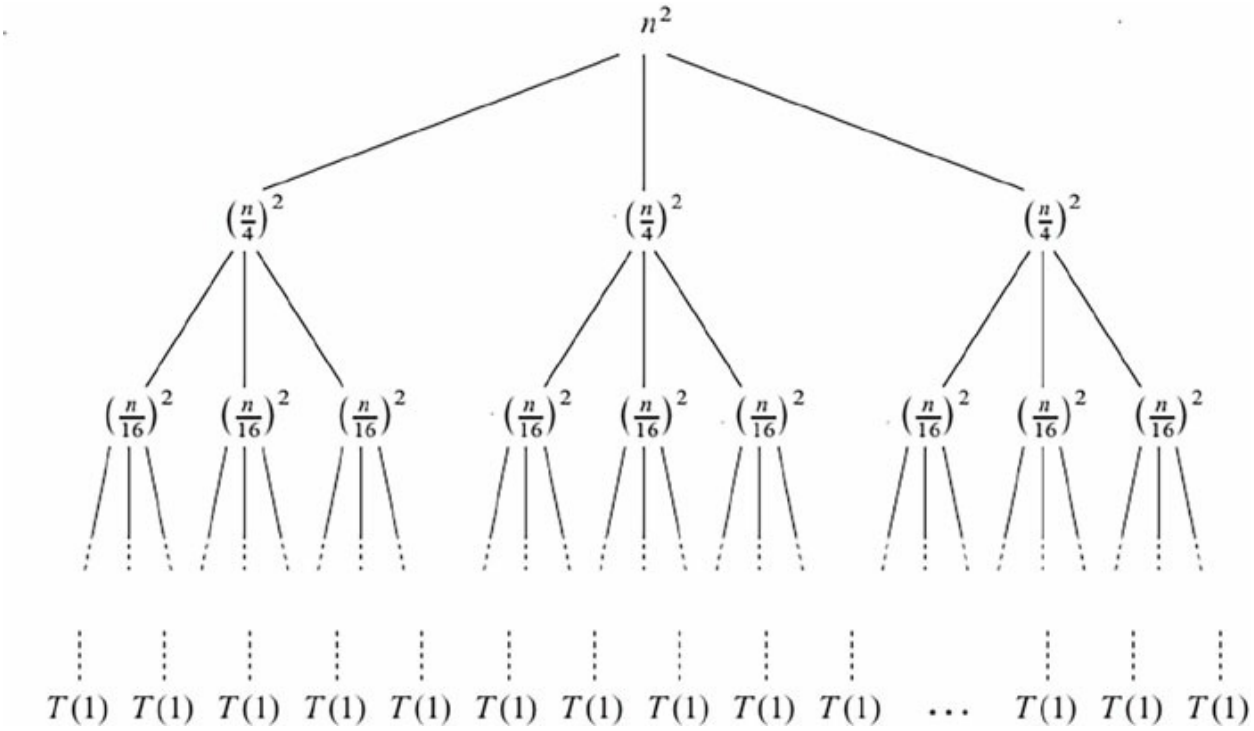


Figure 1.8: *Recursive Tree*

$T(n) = n^2 + 3\,[\,(n/4)^2\,] + 9\,[\,(n/16)^2\,] + 27\,[\,(n/64)^2\,] + \ldots\ldots\ldots$

$= n^2 + 3n^2/4^2 + 9n^2/16^2 + 27n^2/64^2 + \ldots\ldots$

$= n^2\,[\,1 + 3/16 + (3/16)^2 + (3/16)^3 + \ldots.]$

$= n^2\,[\frac{1}{1-\frac{3}{16}}]$ $\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots[\frac{a}{1-r}$ *in G.P* $]$

$= n^2\,[\frac{16}{13}] = 16n^2/13$

Therefore, $T(n) = O(n^2)$

Conclusion

In this module, we have discussed the need of analysis of algorithms and to choose a better algorithm for a particular problem. The main concern is the time complexity and space complexity which has been discussed previously in this chapter. We also learned about the measurement of an input size, that is, an algorithm's efficiency, as a function of some parameter, indicating the algorithm's input size. Asymptotic notations are the expressions which are used to represent the complexity of an algorithm, that is, Average case, Best case and Worst case.

Various Advanced Data Structures are explained in the next chapter, which are used to store and handle data in a very effective manner, for easier and faster access and alteration of data. AVL tree, B / B++ tree, Red Black tree, Tries, and other sophisticated data structures are explored in the next chapter.

Key facts

- An essential component of computational complexity theory is analysis of algorithms.
- Analysis of algorithms refers to the complexity calculation of an algorithm on the basis of time and space required for the execution.
- Certain algorithms need more space, but less time for execution or vice versa. It is better to take into account both time and space complexity while solving a problem.
- The time complexity of an algorithm quantifies how long it takes an algorithm to run in relation to the size of the input. We should be aware that the time to run depends on the length of the input rather than the machine's actual execution time.
- An algorithm's computational time for a given input, n, is expressed using asymptotic notation.
- There are three various asymptotic notations, such as Big O, Theta, and Omega. Big O represents the worst-case scenario of an algorithm, theta for average-case and Omega for best-case scenario of an algorithm.
- An equation or inequality known as a recurrence, defines a function in terms of the values it takes on smaller inputs.
- The major types of recurrence methods are substitution method, Iteration method, Recursion tree method and Master theorem.

Questions

1. Which are the different methods of solving recurrences? Explain with examples. **[MU DEC'18 (10 MARKS)] [MU DEC'19 (10 MARKS)]**
2. Compute the worst-case complexity of the following program segment: **[MU MAY'19 (5 MARKS)]**

```
Void fun(int n, intarr[]){
```

```
  Int i = 0, j = 0;
  for(; i< n; ++i)
   while(j < n && arr[i] < arr[j])
    j++;
}
```

3. Give asymptotic upper bound of *T(n)* for the following recurrences and verify your answer using suitable recurrence method: **[MU MAY'19 (10 MARKS)]**

   $T(n) = T(n-1) + n$

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2

Advanced Data Structures

Introduction

To store data effectively and organize it for quick access and change, data structures are utilized. Arrays, Linked Lists, Stacks, Queues, and other fundamental data structures are a few examples. This chapter discusses several sophisticated and complex data structures, including the AVL tree, the Huffman algorithm, Heap and many more.

A height-balanced binary search tree known as an AVL tree, is one in which each node is assigned a balance factor, which is determined by deducting the height of the node's right sub-tree from the height of its left sub-tree. A tree is said to be balanced if the balance factor for each node is between -1 and 1. Otherwise, rotations must be used to balance the tree.

Huffman coding is a lossless data compression algorithm. For each character entered into this algorithm, a variable-length code is assigned. The codes for the most common characters are the shortest, while the codes for the least common characters are longer. Essentially, there are two sections. A Huffman tree should be built first, and then another should search the tree for codes.

In data structures, a 2-3 Tree is a sort of tree where each node is either a 2 node or a 3 node. It is a particular variety of B-Tree with order three. A node with a value of two in the tree is one with two child nodes and one data component. A node with three child nodes and two data components is known as a "3 node" in the tree.

In terms of self-balancing binary search trees, red-black trees are one type. Each node of the binary search tree also contains an additional bit that represents "red" and "black," which aids in maintaining the tree's balance during insertions and deletions, in addition to the user data.

Tries are sometimes referred to as digital trees, radix trees, or prefix trees. Tries is a tree-based data structure for storing strings that enables rapid

pattern matching. A heap is a complete binary tree. In case of Min Heap, the parent node's value must be lower than or equal to that of either of its children. In case of Max Heap, the parent node's value must be greater than or equal to that of either of its children.

B-tree is a type of tree data structure that keeps its data sorted and offers logarithmic-time searches, sequential access, insertions, and removals. The binary search tree is generalized by the B-tree, which supports nodes with more than two children.

Structure

In this chapter, we will discuss the following topics:

- Advanced Data Structures
- AVL Tree
- Huffman Algorithm
- 2-3 Tree Operation
- Red-Black Trees
- Tries
- Heap Sort
- B Trees

Objectives

By the end of this chapter, the learner will understand and analyze, and be able to choose appropriate data structures and algorithms for the chosen problem statements. It helps the learners understand the required mathematical aspects, in order to design the algorithms for any given specific problem. The learners will be able to comprehend the design approaches with various performance analysis such as efficiency, correctness and so on.

Advanced data structures

Data Structures are used to store data and manage the data in a very efficient way for easy and faster access and for modification of data. Data Structures are used in every program. Some examples of data structures are arrays,

stacks, queues, linked lists, binary trees, and hash tables. This module consists of AVL tree, Red Black tree, B/B++ tree, Tries, and so on.

AVL Tree

The AVL Tree was developed by Russians, Adelson-Velsky and Landis (hence AVL). The key here is to keep the height of a binary search tree balanced.

Definition: A binary search tree called an AVL tree, has nodes with balance factors of 0, +1, or -1.

Balance factor= Height (left subtree) – Height (right subtree)

We perform rotation in AVL tree only in case the Balance Factor is other than -1, 0, and 1. There are basically four types of rotations, which are as follows:

- **LL rotation:** The node that was inserted is in the left subtree of the left subtree of A.
- **RR rotation**: The node that was inserted is in the right subtree of the right subtree of A.
- **LR rotation:** The node that was inserted is in the right subtree of the left subtree of A
- **RL rotation:** The node that was inserted is in the left subtree of the right subtree of A, where node A is the node whose balance Factor is not -1, 0, or 1.

The initial 2 rotations LL and RR are single rotations whereas the next 2 rotations LR and RL are double rotations. A tree's minimum height must be at least 2 to be unbalanced. Let us examine each rotation now.

RR Rotation

When a **Binary Search Tree** (**BST**) becomes imbalanced because an element is incorporated into the right subtree of the right subtree of A, we apply RR rotation, which is an anticlockwise revolution, to the edge under a node with a balance factor of -2. Refer to *Figure 2.1:*

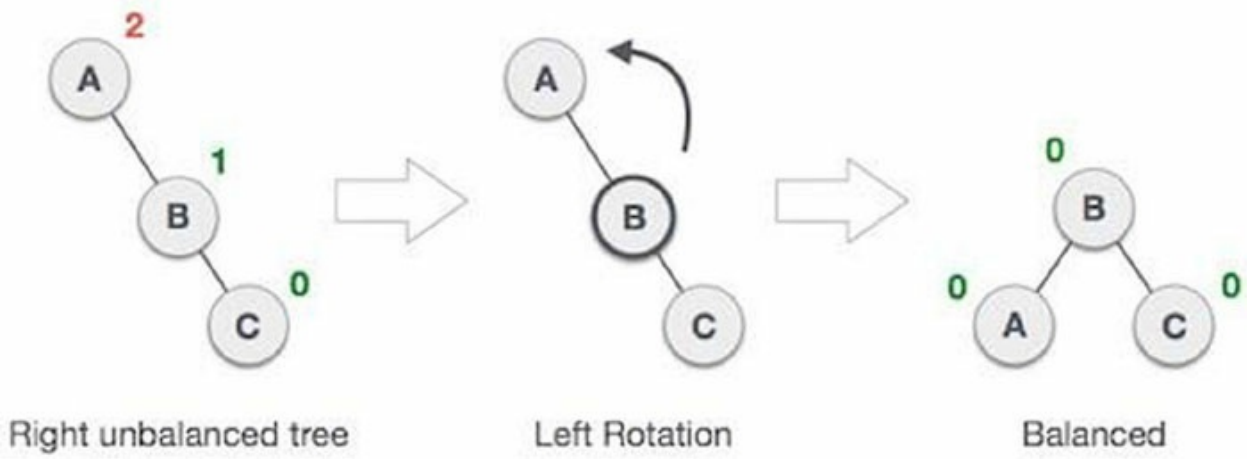


Figure 2.1: *RR Rotation*

In the preceding *Figure 2.1*, node A has balance factor of 2 on the grounds that a node C is embedded in the right subtree of A right subtree. We play out the RR turn on the edge below A.

LL Rotation

We perform LL revolution, which is a clockwise rotation, on the edge beneath a node with a balancing factor of 2, at the point when BST becomes uneven because an element is embedded into the left subtree of the left subtree of C. Refer to *Figure 2.2*:

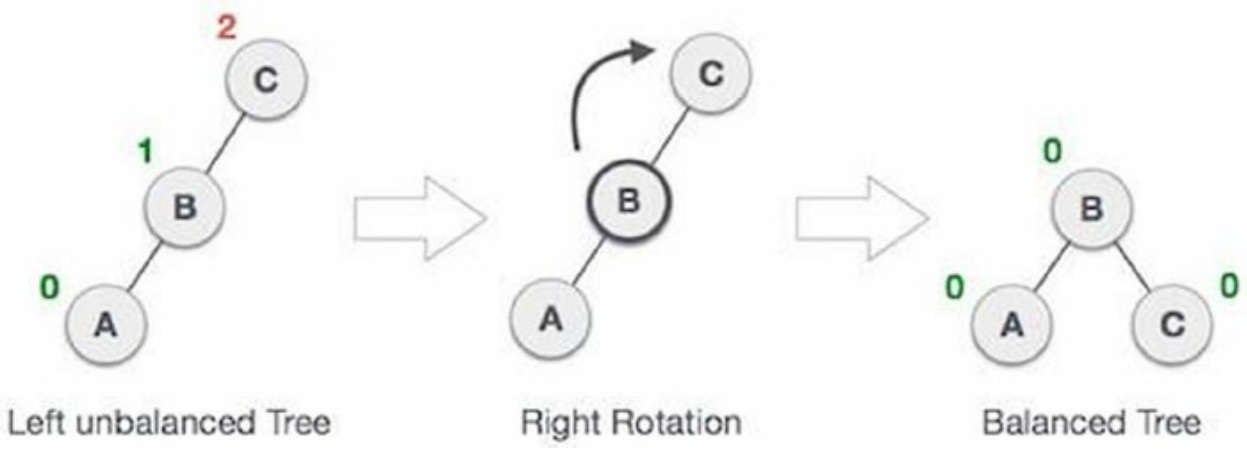


Figure 2.2: *LL Rotation*

Since a node A is embedded in the left subtree of the left subtree of node C in the model shown in *Figure 2.2*, node C has a balancing factor of 2. On the node under A, we perform the LL revolution.

LR Rotation

Without a doubt, double rotations are more difficult than one rotation. LR rotation = RR rotation + LL rotation, that is, first RR rotation is performed on subtree and afterward, LL rotation is performed on full tree. By full tree, we mean the first node from the way of embedded node, whose balance factor is not - 1, 0, or 1. Refer to *Figure 2.3*:

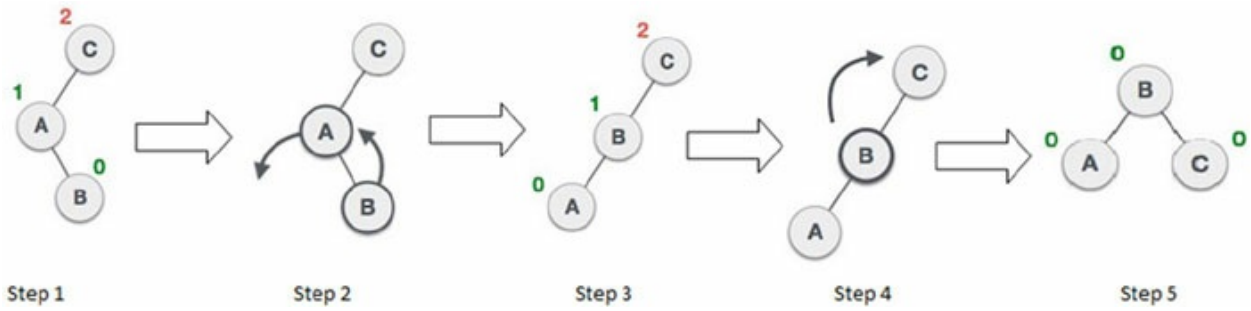


Figure 2.3: *LR Rotation*

Step 1: Because a node B was added to the left subtree of C and the right subtree of A, C developed a balance factor of 2, making it an unbalanced node. In *Figure 2.3*, the inserted node is in C's right subtree's left subtree.

Step 2: Because LR rotation = RR + LL rotation, RR (anticlockwise) is done first on the subtree rooted at A. After RR rotation, Node A has evolved into the left subtree of B.

Step 3: Node C is still unbalanced after RR rotation, with a balance factor of 2, because inserted node A is to the left of C.

Step 4: Starting at node C, we rotate the complete tree clockwise now. Node C is now the right subtree of node B, while A is the left subtree.

Step 5: Each node's balance factor is now -1, 0 or 1, indicating that BST is now balanced.

RL rotation

RL rotation = LL rotation + RR rotation, that is, first LL rotation is conducted on subtree and afterward, RR rotation is performed on the whole tree, which is the primary node from the way of the embedded node whose balancing factor is not 1, 0, or 1. Refer to *Figure 2.4*:

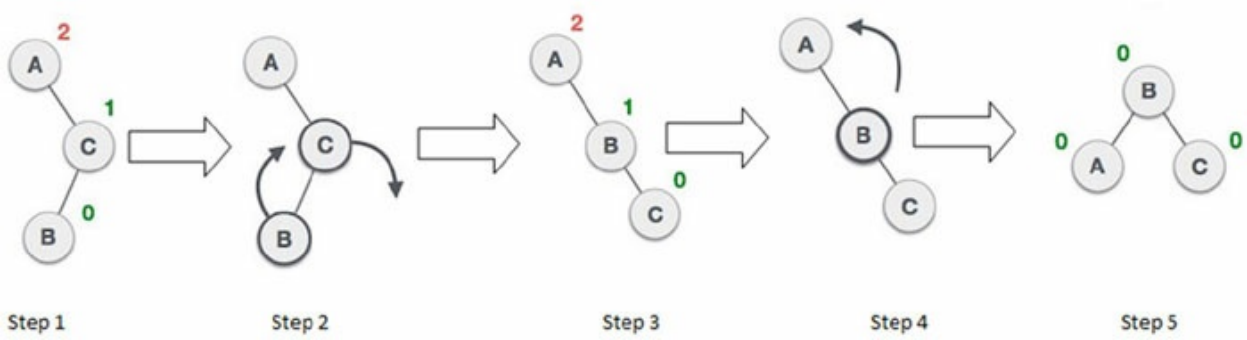


Figure 2.4: *RL Rotation*

Step 1: A became an unbalanced node with a balance factor of -2, as a result of a node B being added to the left subtree of C and the right subtree of A. The inserted node is in the left subtree of A's right subtree in *Figure 2.4*, of RL rotation.

Step 2: The subtree rooted at C is rotated in LL (clockwise) first, because *RL rotation =LL rotation +RR rotation*. After RR rotation, Node C has evolved into the right subtree of B.

Step 3: Because of the right-subtree of the right-subtree node A, node A remains unbalanced after executing LL rotation, with a balance factor of -2.

Step 4: We now conduct RR rotation (anticlockwise rotation) on the entire tree, starting with node A. Node C is now the right subtree of node B, while node A is the left subtree of B.

Step 5: Each node's balance factor is now either -1, 0 or 1, indicating that BST is now balanced.

Huffman algorithm

Huffman codes or Huffman algorithm is an entropy encoding algorithm. David Huffman invented it in 1952. It is used widely for data compression. The fundamental principle of Huffman coding is to employ shorter bit strings to encode the most common character, than to encode less common source characters.

Example 1

Generate Huffman code for the given input "**`mississippi`**".

Count the characters which are repeating, as shown in *Table 2.1*:

Character	Frequency
m	1
i	4
s	4
p	2

Table 2.1: *Character Frequency Count*

Arrange the frequencies either in ascending or descending order, as shown in *Table 2.2*:

Character	Frequency
m	1
p	2
i	4
s	4

Table 2.2: *Sorted Frequency Character Count Set*

Add the first two frequencies 1 + 2 = 3 and let us give it ‘o’, as shown in *Figure 2.5*:

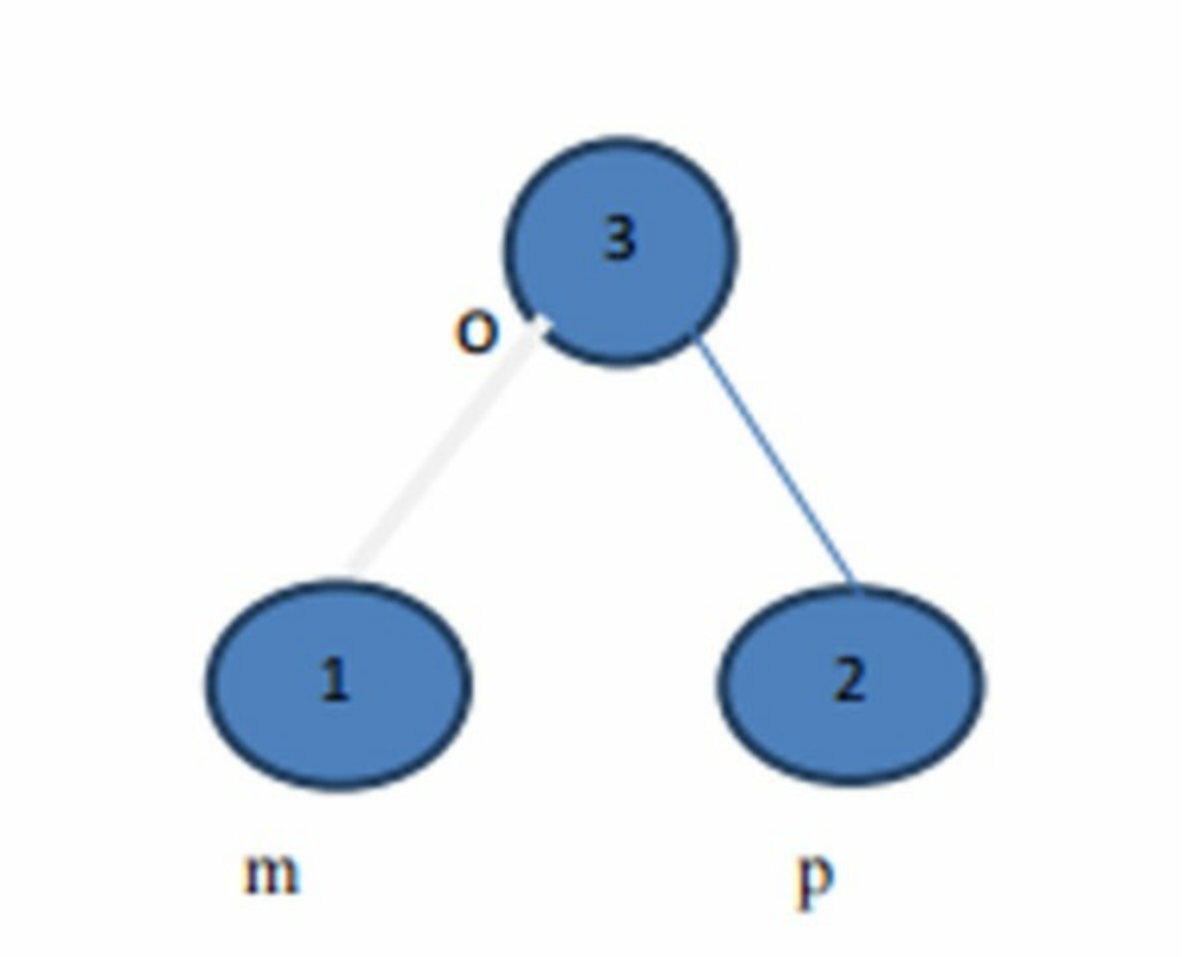


Figure 2.5: *Huffman tree after inserting m & p*

Refer to the following *Table 2.3*:

Character	Frequency
o	3
s	4
i	4

Table 2.3: *Sorted Frequency Character Count Set*

Now, in the next step, add the ‘o’ in ascending manner with the remaining characters and start adding the lowest frequencies, that is, 3+4 =7. Let us give it ’q’, as shown in *Figure 2.6*:

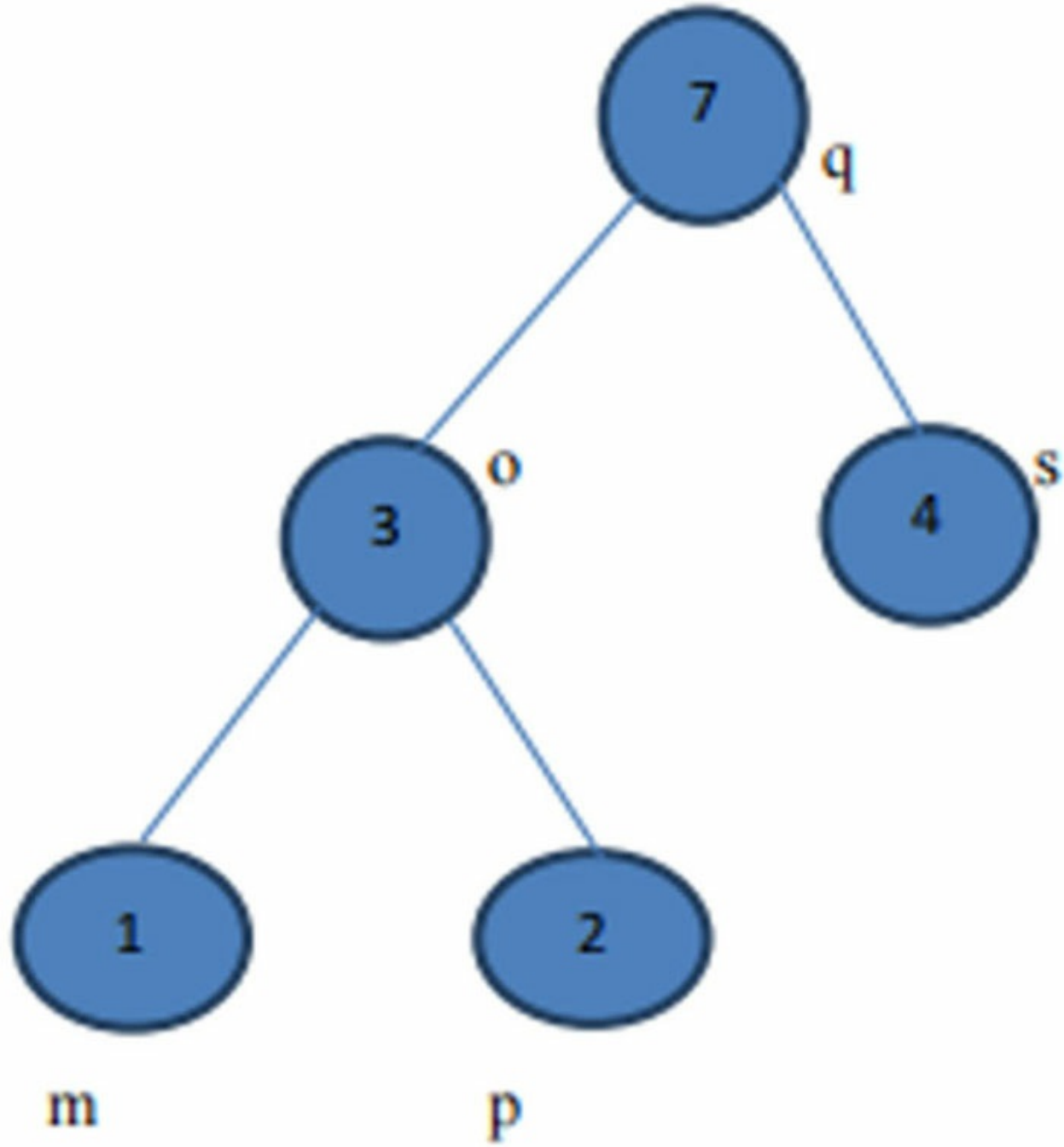


Figure 2.6: *Huffman Tree after adding o and s*

Now take the remaining two frequencies and add it and let's give it 'r', as shown in *Table 2.4:*

Character	Frequency
i	4
q	7

Table 2.4: *Sorted Frequency Character Count Set*

Refer to the following *Figure 2.7*:

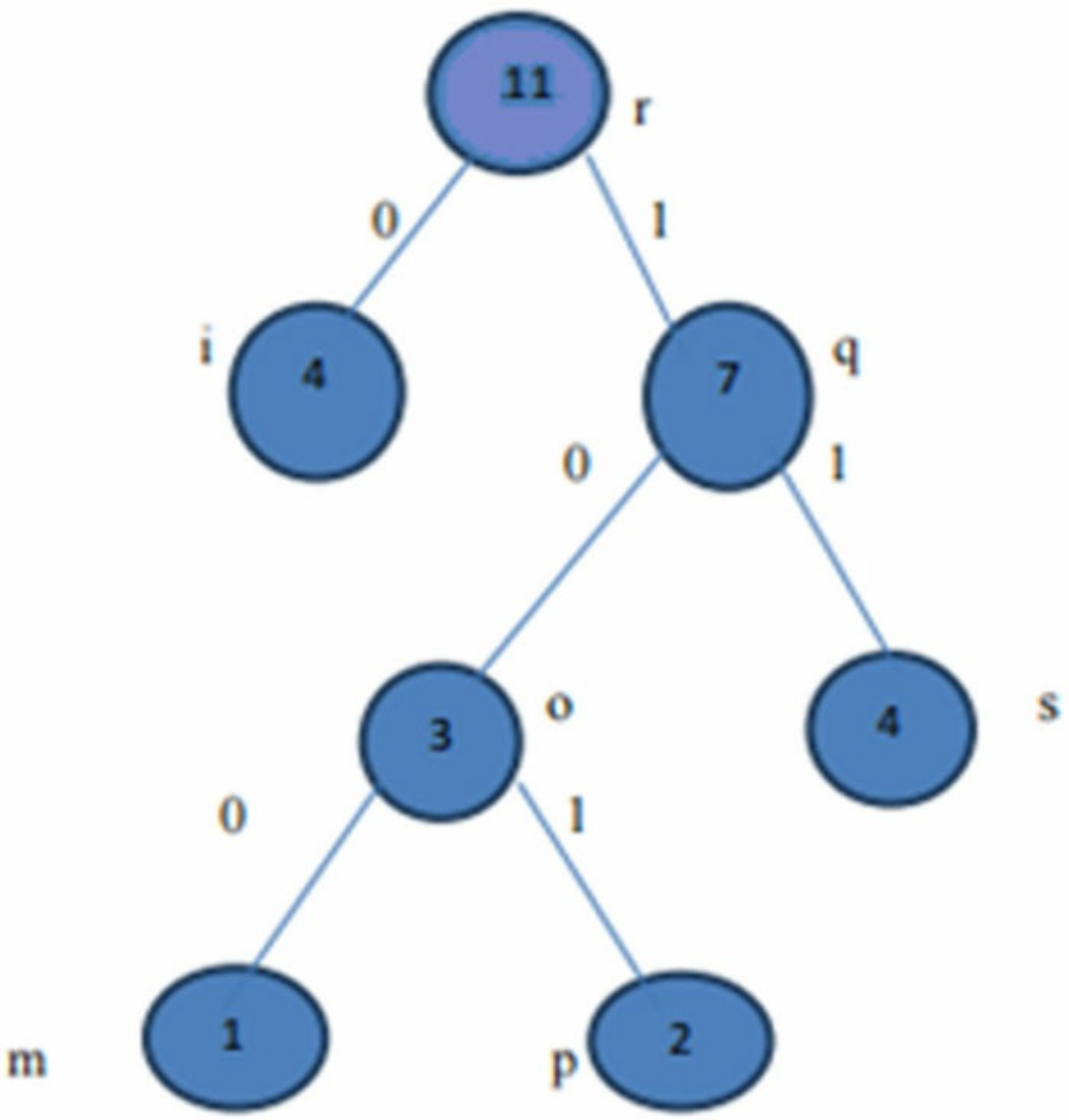


Figure 2.7: *Huffman Tree after adding i and q, then coded with 1 & 0*

Traverse the tree from the starting node. When we move to the left child, write 0 and when we move to the right child, write 1.

The codes are as shown in *Table 2.5*:

Character	Frequency
m	100
p	101
i	0
s	11

Table 2.5: *Huffman Code set*

Example 2

A networking company uses a compression technique to encode the message before transmitting over the network. Suppose the message contains the characters shown in *Table 2.6* with their frequency:

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Table 2.6: *Character Frequency Count Set*

Note that each character in the input message takes 1 byte.

Solution: First draw the Huffman tree.

Step 1: Add frequencies of ‘a’ and ‘b’. Let’ us give it ‘l’, as shown in the following *Figure 2.8:*

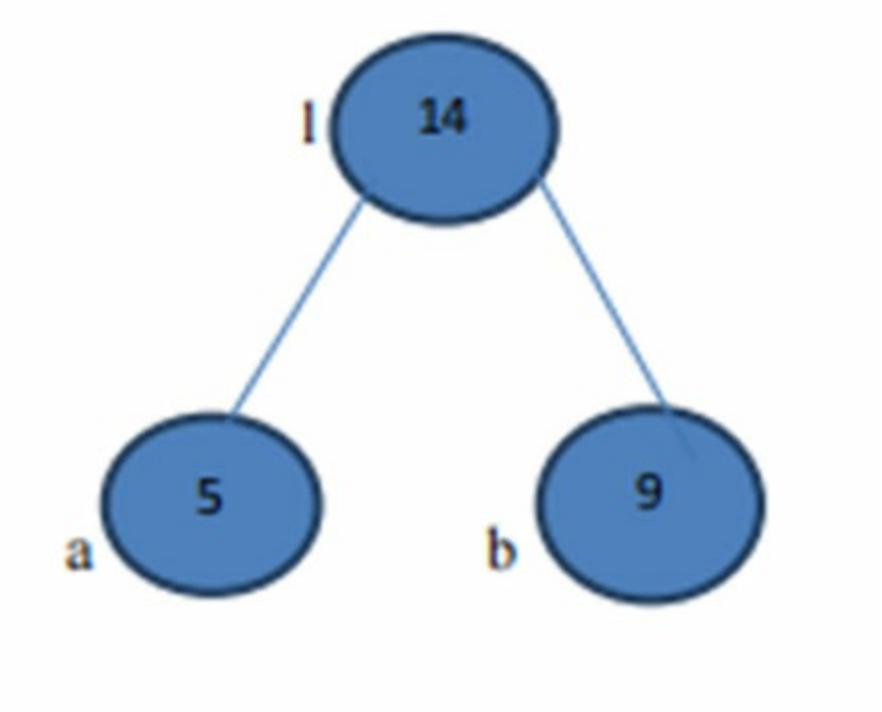


Figure 2.8: *Huffman Tree- 5 & 9 inserted*

Refer to the following *Table 2.7*:

Character	Frequency
l	14
c	12
d	13
e	16
f	45

Table 2.7: *Character Frequency Count Set*

Step 2: Now take the two least frequencies and add them, that is, ‘c’ and ‘d’. Let us give it ‘m’, as shown in the following *Figure 2.9*:

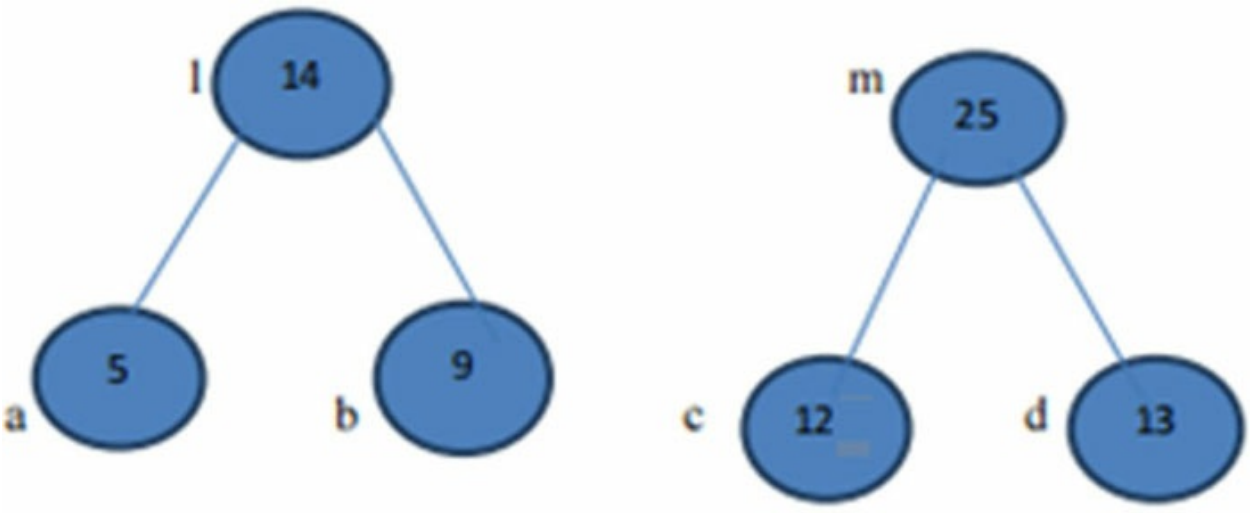


Figure 2.9: *Huffman Tree- 12 & 13 inserted*

Refer to the following *Table 2.8*:

Character	Frequency
l	14
m	25
e	16
f	45

Table 2.8: *Character Frequency Count Set*

Step 3: Add ‘l’ and ‘e’ frequencies and name them ‘n’, as shown in the following *Figure 2.10:*

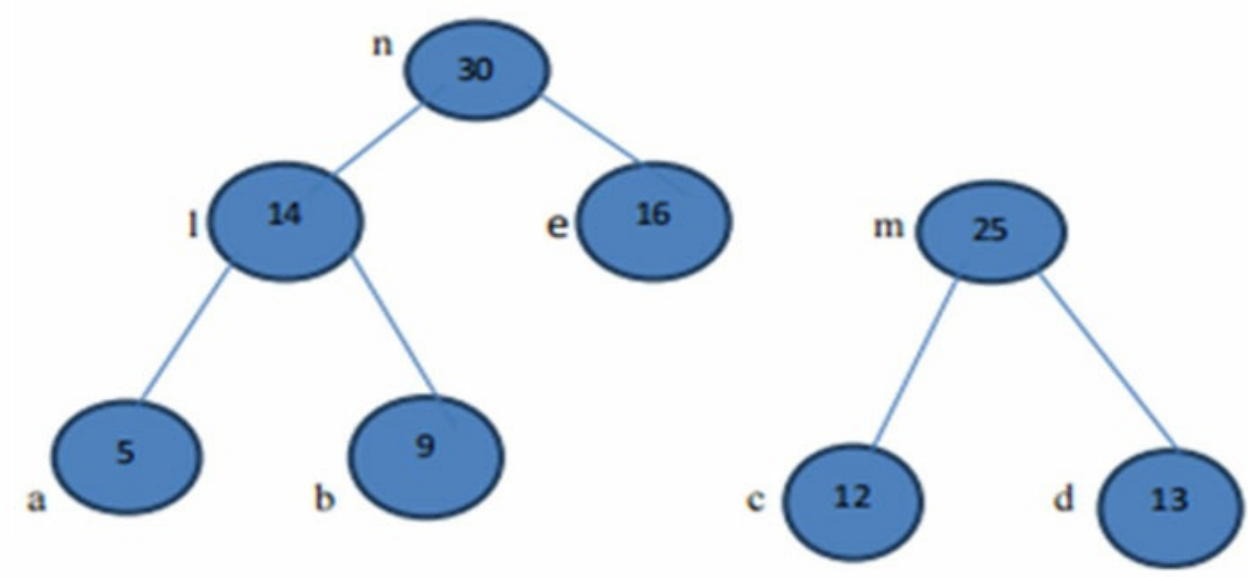


Figure 2.10: *Huffman Tree -16 inserted*

Refer to the following *Table 2.9*:

Character	Frequency
m	25
n	30
f	45

Table 2.9: *Character Frequency Count Set*

Step 4: Now add the frequencies of ‘m’ and ‘n’. Let’s give it ‘o’. Refer to the following *Figure 2.11:*

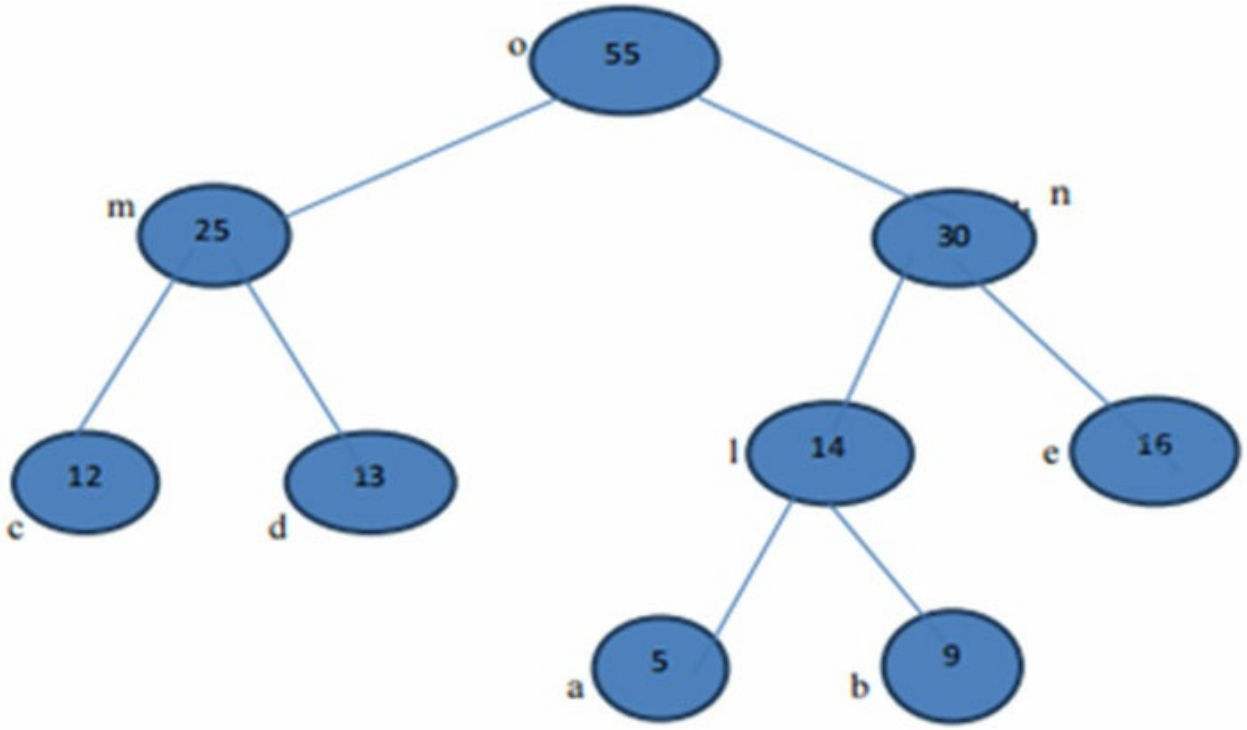


Figure 2.11: *Huffman Tree- 25 & 30 added*

Step 5: Traverse the tree from the starting node. When we move to the left child, write 0 and when we move to the right child, write 1. Refer to the following *Figure 2.12:*

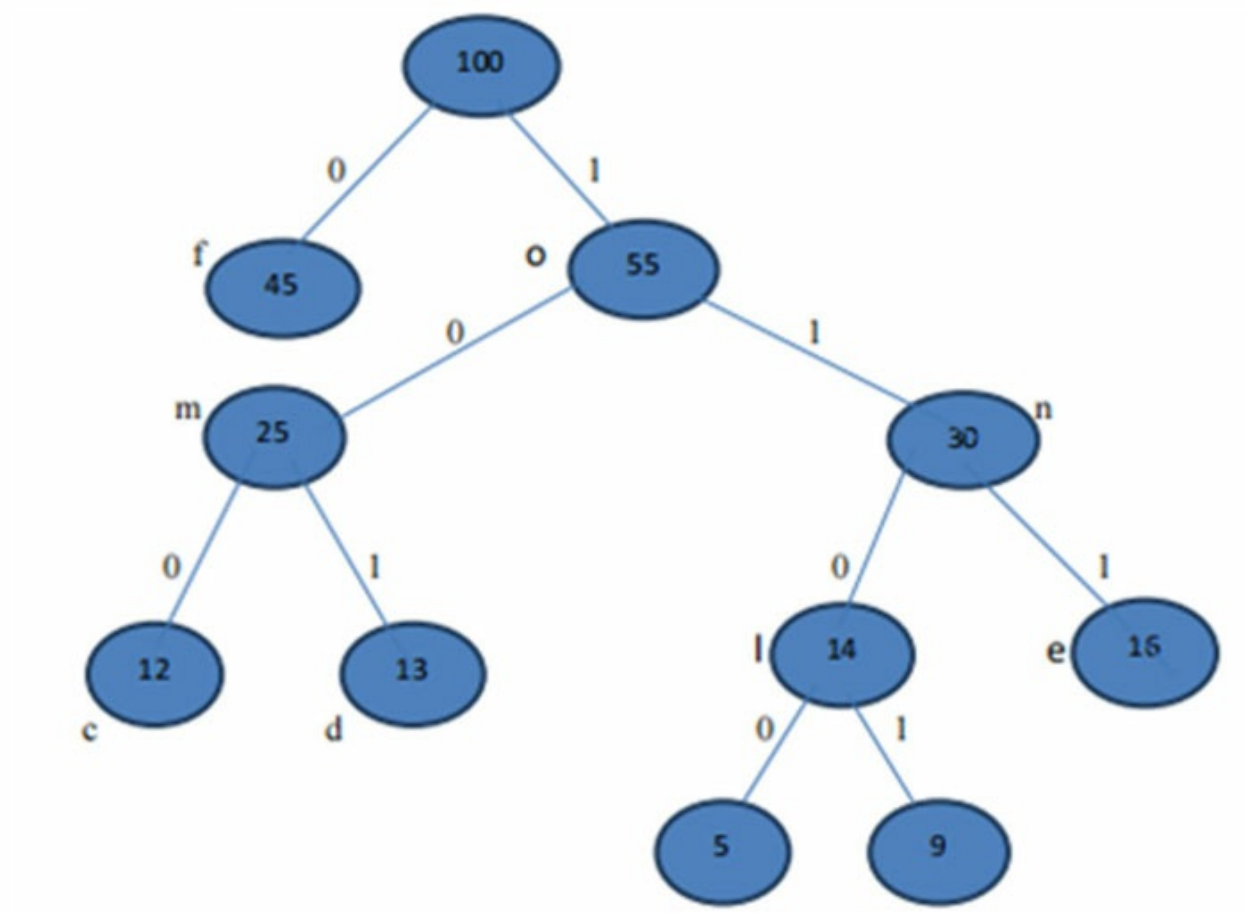


Figure 2.12: *Final Huffman Tree after inserting 45*

Refer to the following *Table 2.10*:

Characters	Code-word	Frequency	Total Number of Bits
a	1100=4 bits	5	4*5=45
b	1101=4 bits	9	4*9=36
c	100=3 bits	12	3*12=36
d	101=3bits	13	3*13=39
e	111=3bits	16	3*16=48
f	0=1bit	45	1*45=45

Table 2.10: *Huffman Code Set*

Total Number of Bits *= (4*5)+(4*9)+(3*12)+(3*13)+(3*16)+(1*45) =224*

*Size of 1 char = 1 byte= 8 bit = 8 *100 =800 bits*

Total number of bits saved by Huffman coding = 800 -224 = 576bits.

2-3 Tree operation

2-3 trees are also known as **3 order B-tree**. Every internal node has a maximum of two data elements and a maximum of three children in this tree. The main advantage of the 2-3 tree is that it is the most balanced as compared to BST, and hence, the complexity of insertion, deletion in 2-3 trees is O(log (n)).

For example, refer to the following *Figure 2.13:*

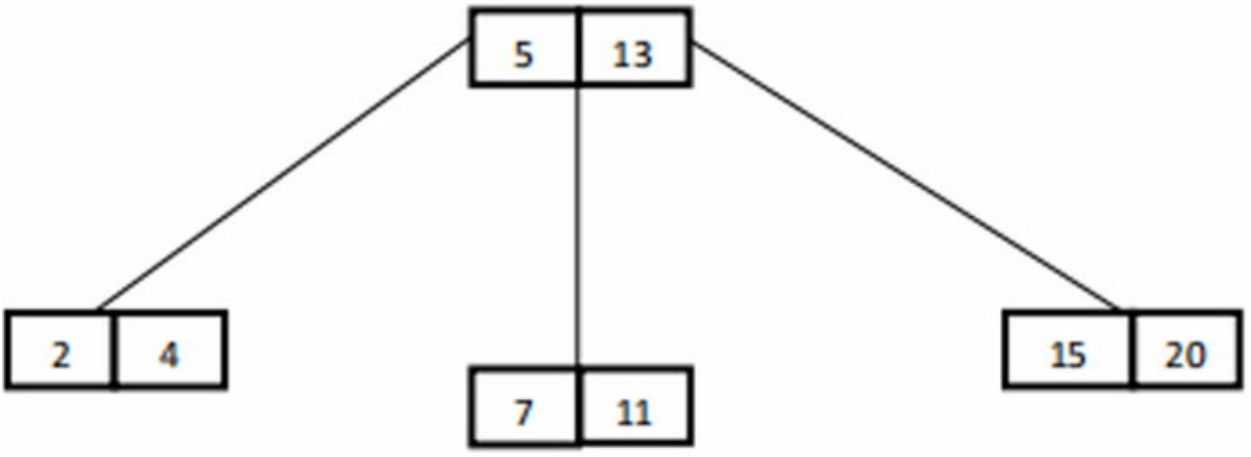


Figure 2.13: *2-3 Tree*

Searching an element in 2-3 tree

The Searching process in 2-3 trees is very much like that of a Binary Search Tree. Let us assume that we want to find the number 'X' in the tree, and then let M1 and M2 be the two values stored in the root node such that M1 < M2.

Then,

- If X < M1, search the X in the left child of the node.
- Otherwise, search in the right child of the node.
- If M1< X>M2, (that is, the node has three children) then search in the middle child of the node to find the X.

Search an element in 2-3 Tree:

Let us assume the following 2-3 tree in which we must find the location of 8 in the tree. Then, it can be searched as shown in *Figure 2.14*:

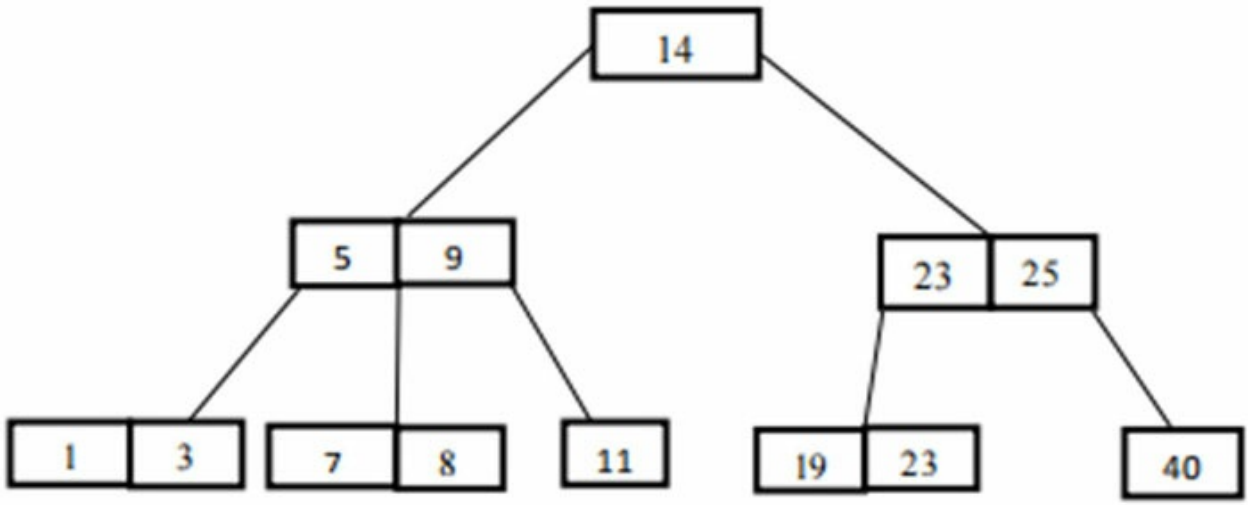


Figure 2.14: *2-3 Tree*

Step 1: Compare 8 with the root element. Here, 8< 14, therefore we will search for the element in the left child of the node. Refer to *Figure 2.15:*

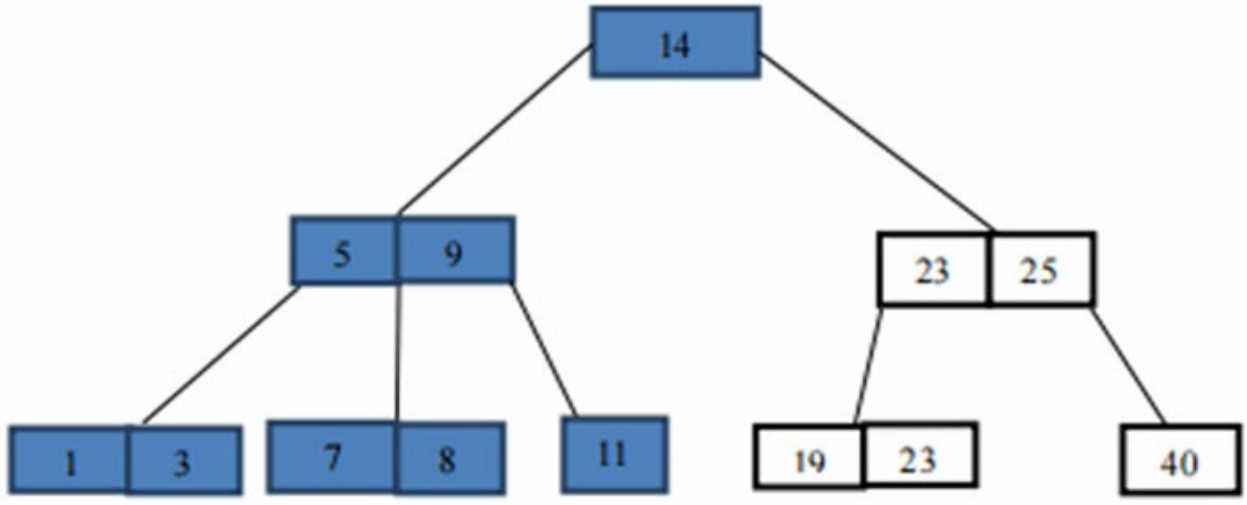


Figure 2.15: *2-3 Tree*

Step2: Now we will compare 8 with another node element, which is, (5, and 9). Since 5<8<9, we will search in the middle child of the node, as shown in *Figure 2.16:*

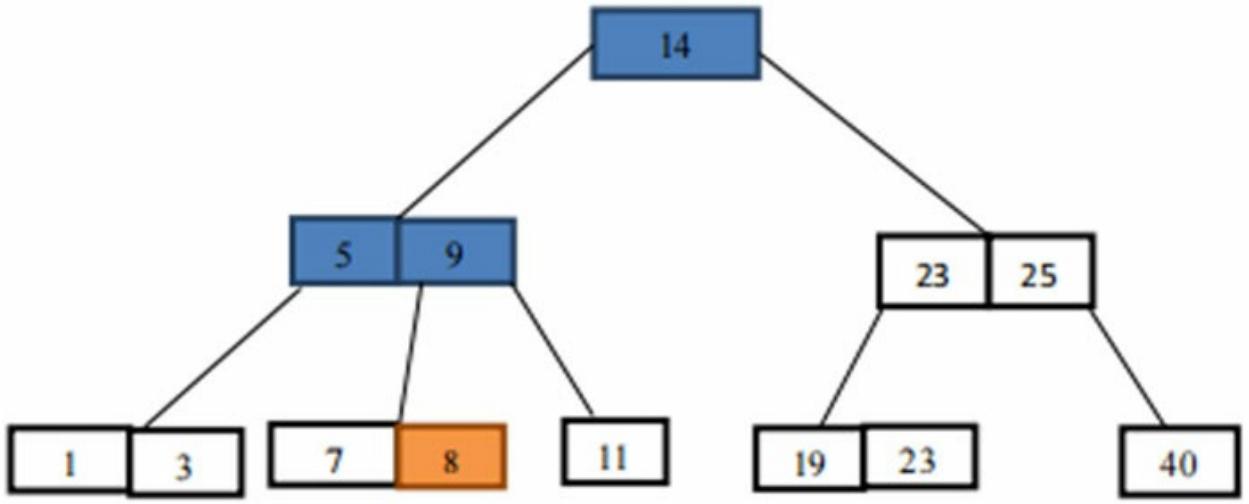


Figure 2.16: *2-3 Tree*

Hence, in this, we can carry out the searching operation easily using 2-3 tree.

Inserting an element in a 2-3 Tree:

Let the elements to be inserted are 1, 12, 8, 2, 25, 5, 14, 28

Step 1: Inserting 1. Refer to *Figure 2.17*:

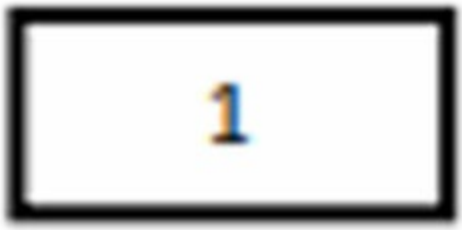


Figure 2.17: *Insertion of 1 in 2-3 Tree*

Step 2: Inserting 12. Refer to *Figure 2.18*:

Figure 2.18: *Insertion of 12 in 2-3 Tree*

Step 3: Inserting 8. Refer to *Figure 2.19*:

Figure 2.19: *Insertion of 8 in 2-3 Tree*

Step 4: Inserting 2. Refer to *Figure 2.20*:

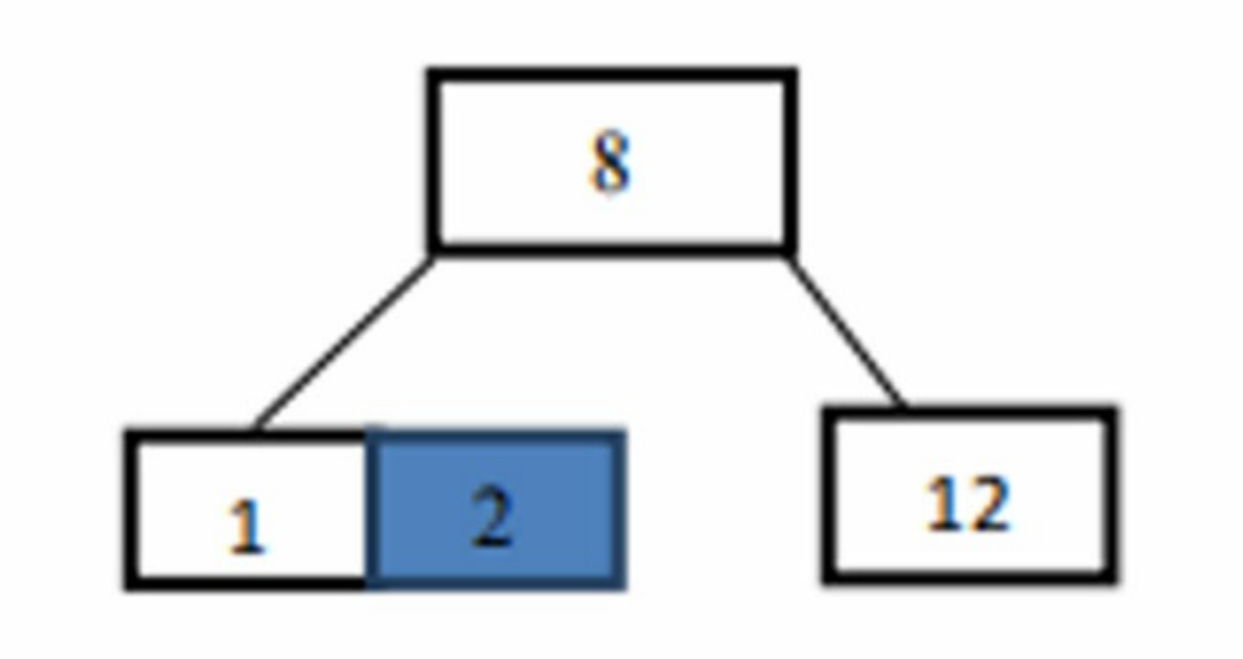


Figure 2.20: *Insertion of 2 in 2-3 Tree*

Step 5: Inserting 25. Refer to *Figure 2.21*:

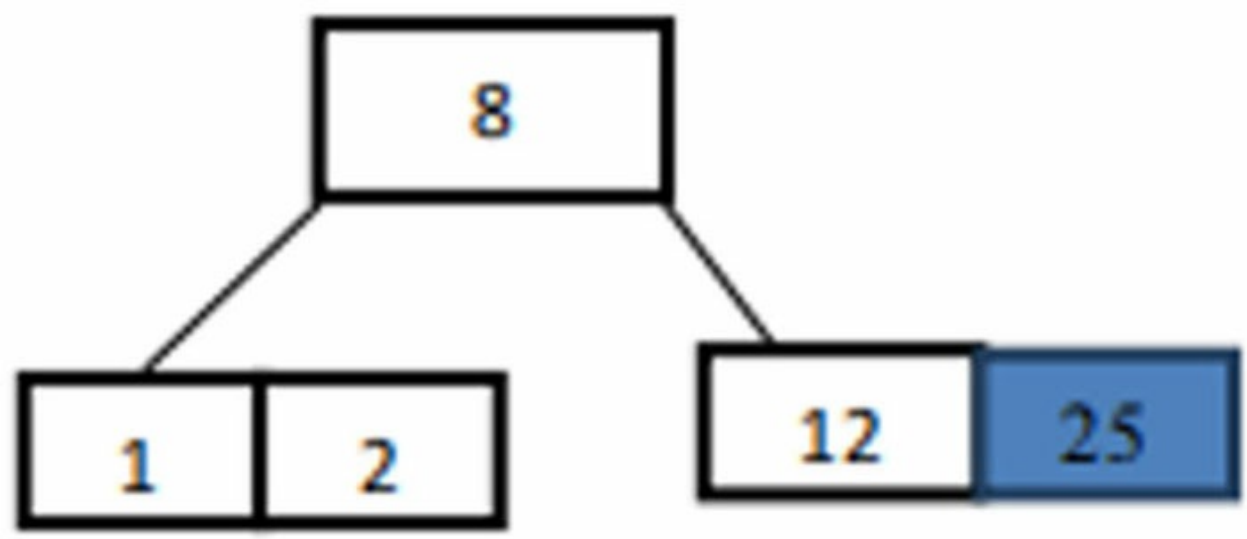


Figure 2.21: *Insertion of 25 in 2-3 Tree*

Step 6: Inserting 5. Refer to *Figure 2.22*:

Figure 2.22: *Insertion of 5 in 2-3 Tree*

Step 7: Inserting 14. Refer to *Figure 2.23*:

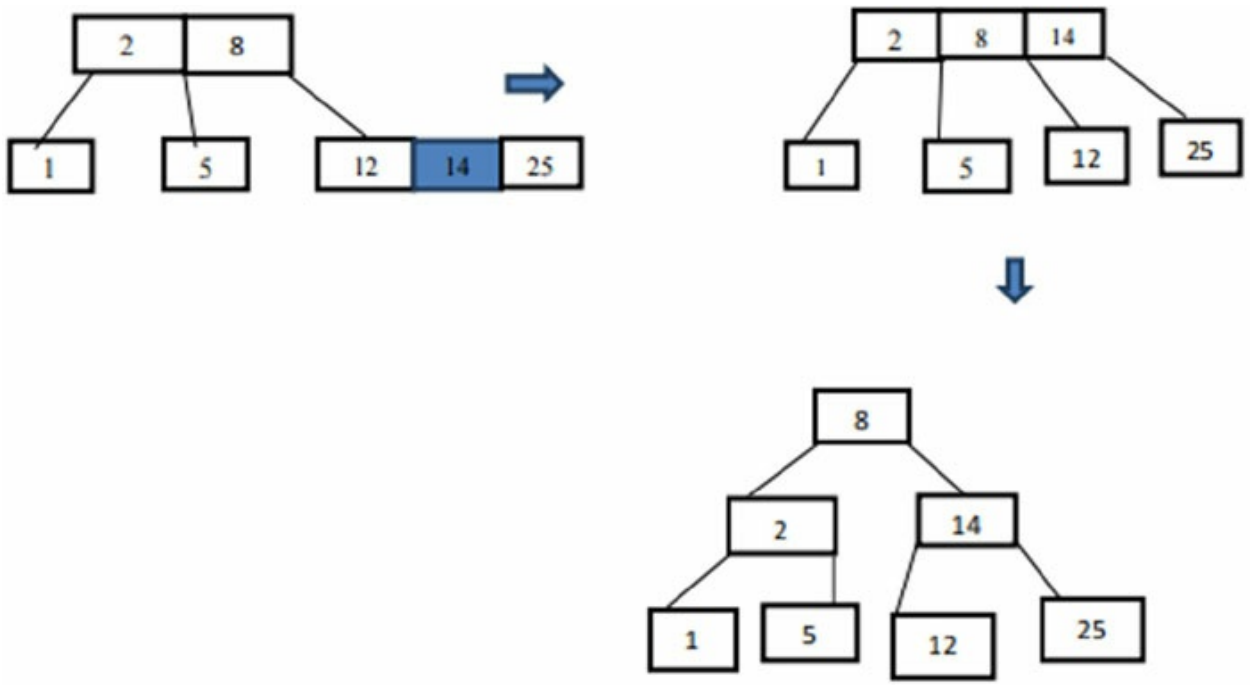


Figure 2.23: *Insertion of 14 in 2-3 Tree*

Step 8: Inserting 28. Refer to *Figure 2.24*:

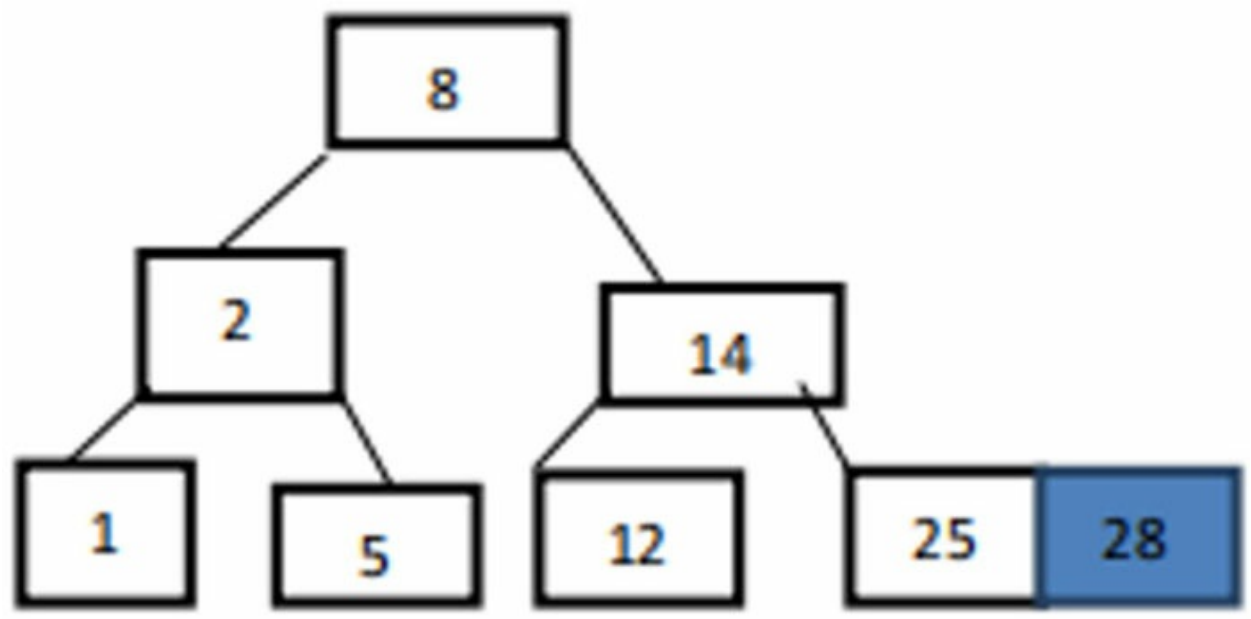


Figure 2.24: *Insertion of 28 in 2-3 Tree*

Hence, in this way we can insert various elements into the 2-3 Tree.

Red-Black Trees

Red-Black Tree is a self-balancing binary search tree. Each node has an additional node that indicates whether it is red or black in color. These colors indicate whether the tree is balanced, every time a node is inserted or deleted.

Rules for insertion of node:

- If it is an empty tree, then create a black root node.
- If there is no empty node, insert the new node or new leaf as red. There are two possibilities which are checked after inserting the red leaf.
    - If the parent of the red node is black, then no changes need to be made.
    - If the parent of the red node is red, then we need to check the following things for the red node:
        - If the parent node has black or absent sibling, then we need to rotate and then recolor it.
        - If the parent node has a red sibling, then recolor and check again whether the tree satisfies all the conditions.

Example 3

Construct a red-black tree for the following sequence: 10, 20, -10, 15, 17, 40, 50, 60.

1. As it is an empty tree, 10 will be inserted as the root node colored as black, as shown in *Figure 2.25*:

Figure 2.25: *Construction of red-black tree – inserted 10*

2. 20 will be inserted to the right as following the rules of binary search tree and colored as red leaf node, as shown in *Figure 2.26*:

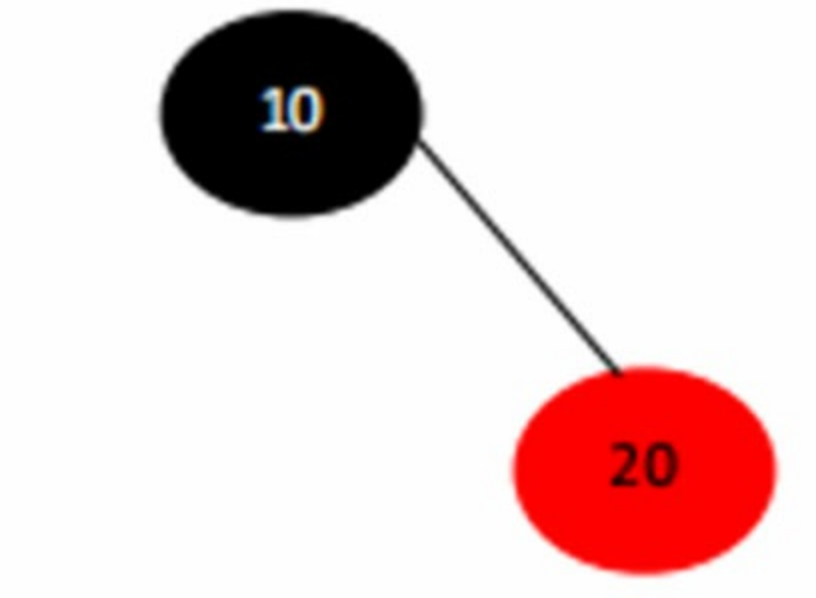


Figure 2.26: *Construction of red-black tree – inserted 20*

3. -10 will be added to the left as a red node, as shown in *Figure 2.27*:

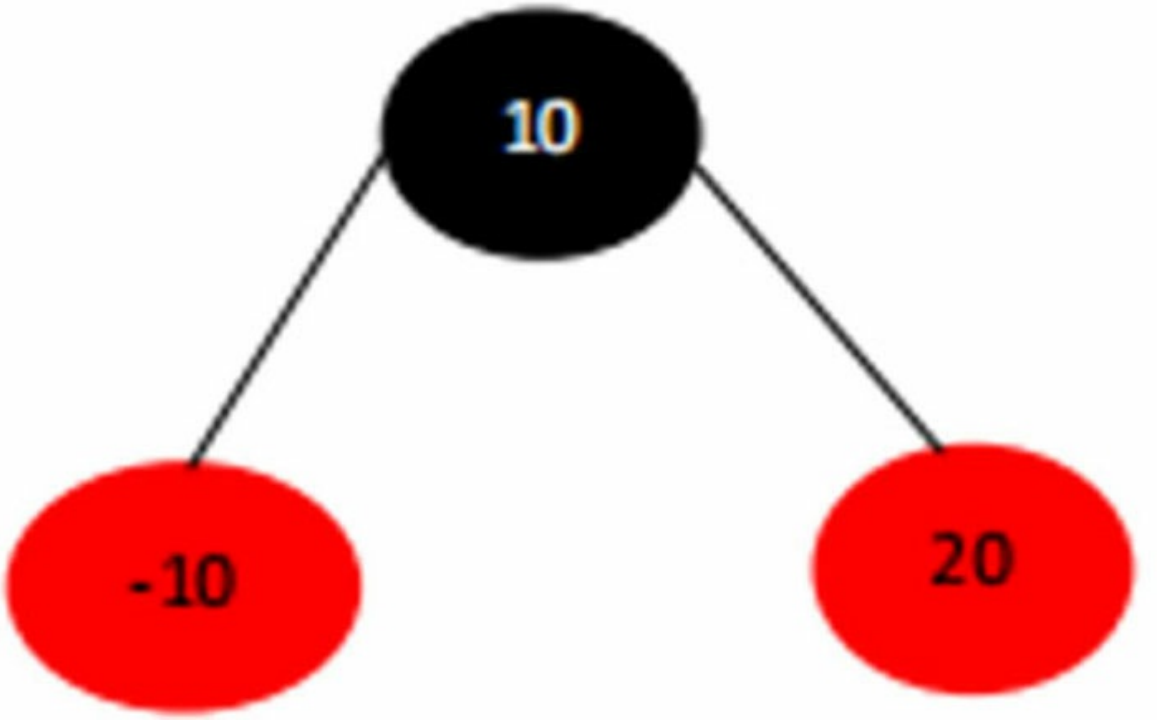


Figure 2.27: *Construction of red-black tree – inserted 10*

4. 15 will be added to the left of 20 as a red node. Since a red leaf cannot have a red parent, changes need to be made. Here, the changes are made according to top rule 2-> b -> ii, as shown in *Figure 2.28*:

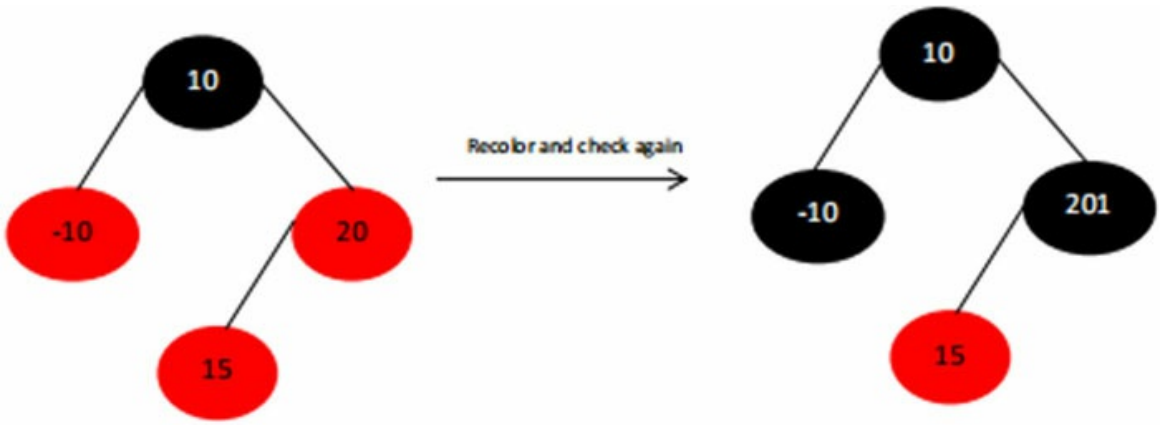


Figure 2.28: *Construction of red-black tree – inserted 15*

5. 17 will be added to the right of 15 as a red node. Since a red leaf cannot have a red parent, changes need to be made. Here the changes are made according to top rule 2-> b -> i, as shown in *Figure 2.29*:

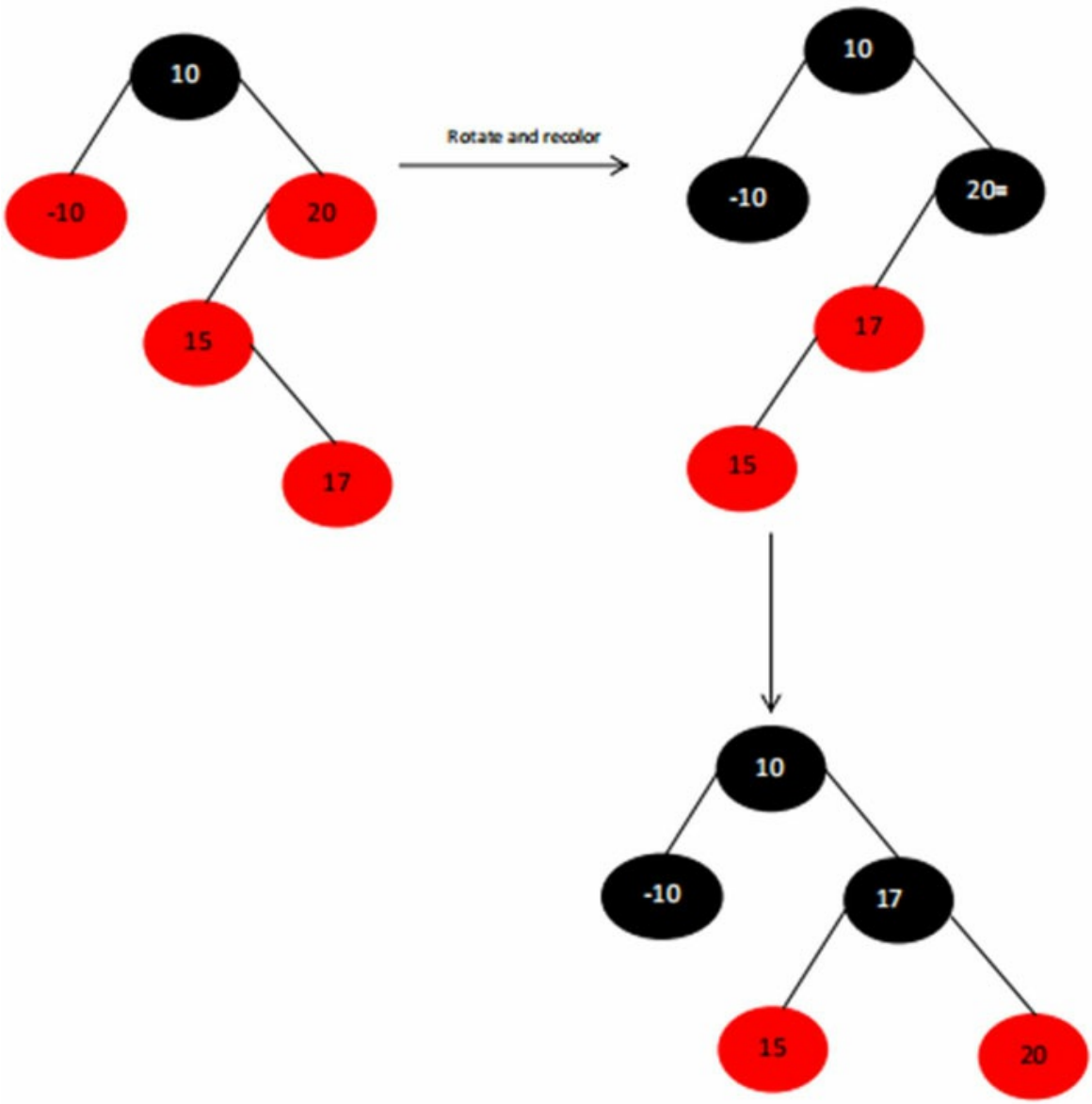


Figure 2.29: *Construction of red-black tree – inserted 17*

6. 40 will be added to the right of 20. Since a red leaf cannot have a red parent, we need to make changes. Here the changes will be made according to rule 2 -> b -> ii, as shown in *Figure 2.30*:

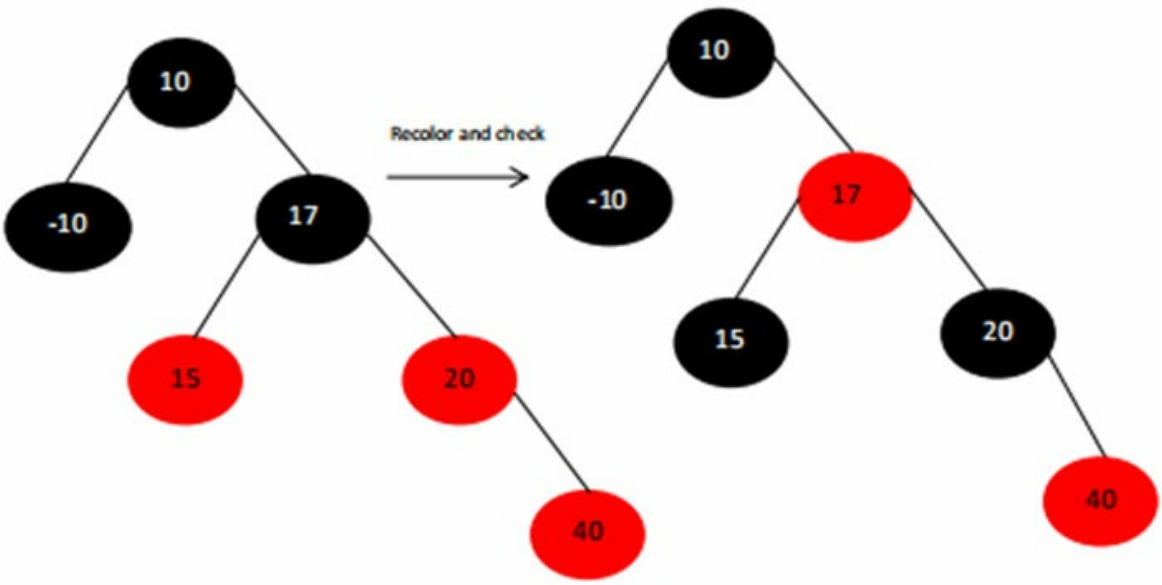


Figure 2.30: *Construction of red-black tree – inserted 40*

7. 50 will be added to the right of 40. Since a red leaf cannot have a red parent, we need to make changes. Here, the changes will be made according to rule 2 -> b -> ii, as shown in *Figure 2.31*:

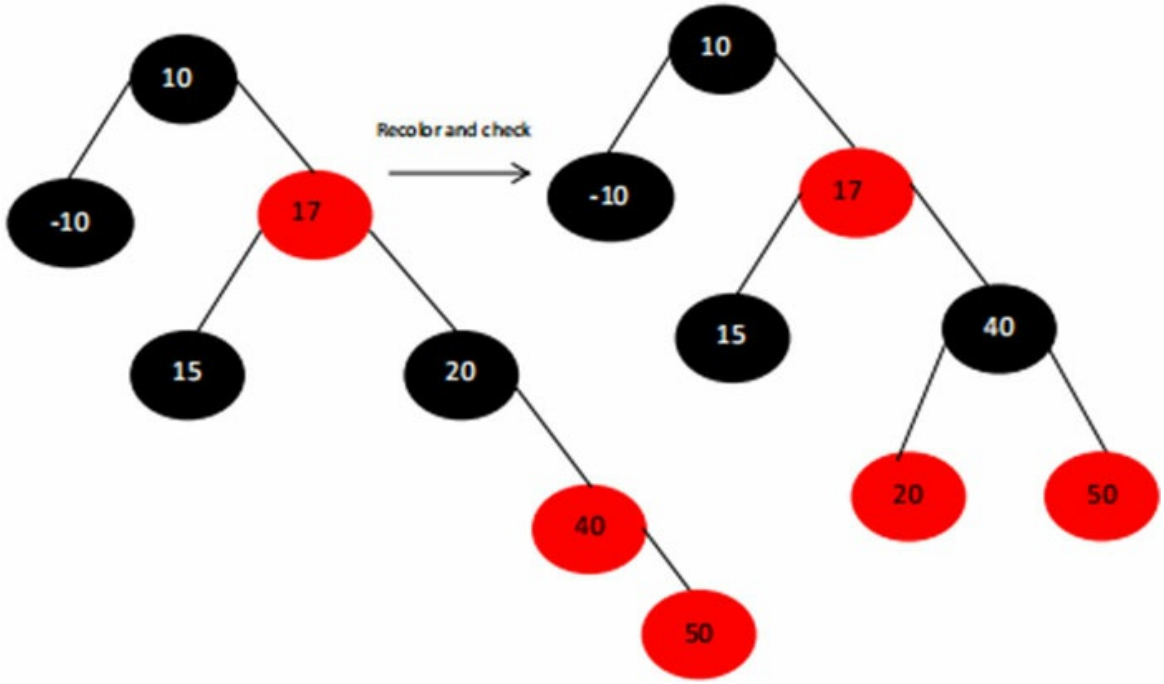


Figure 2.31: *Construction of red-black tree – inserted 50*

8. 60 will be added to the right of 50. Since a red leaf cannot have a red parent, we need to make changes. Here, the changes will be made

according to rule 2 -> b -> ii, as shown in *Figure 2.32*:

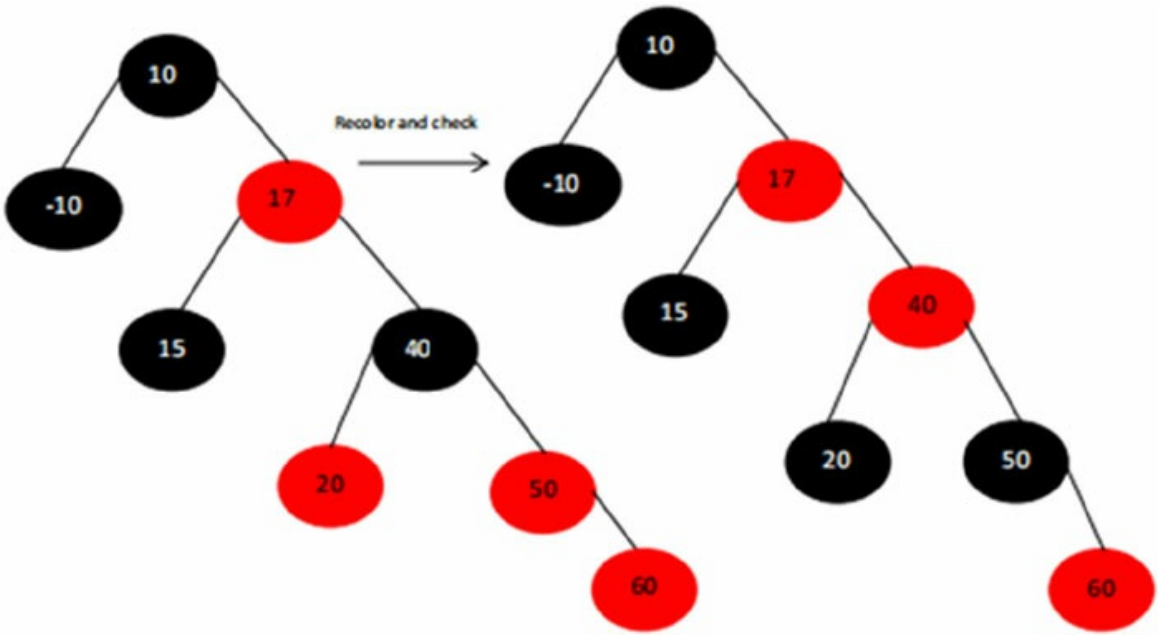


Figure 2.32: *Construction of red-black tree – inserted 60*

Now, if we check the tree again, we can see that the tree does not satisfy all the conditions of red black tree since 40, that is, the red node has 17 as a parent node, which is also a red node. Hence, changes must be made. Changes will be made according to rule 2 -> b ->I (rotate and recolor), as shown in *Figure 2.33*:

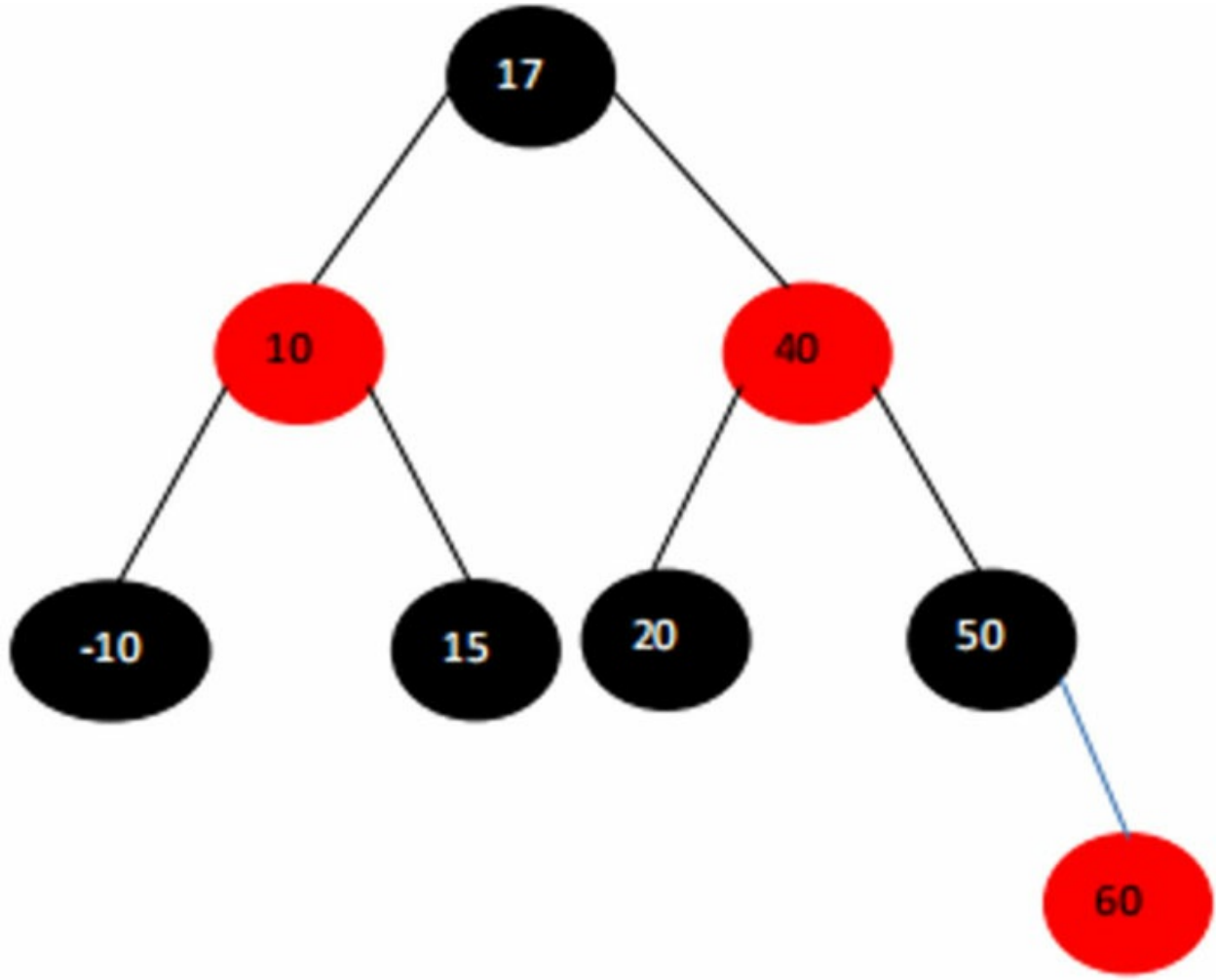


Figure 2.33: *Construction of red-black tree*

Example 4

Construct a red-black tree for the following sequence: 50,60,70,90,20,15,10.

1. Inserting 50 as root node, as shown in *Figure 2.34*:

Figure 2.34: *Insertion of 50 in red-black tree*

2. Inserting 60, as shown in *Figure 2.35*:

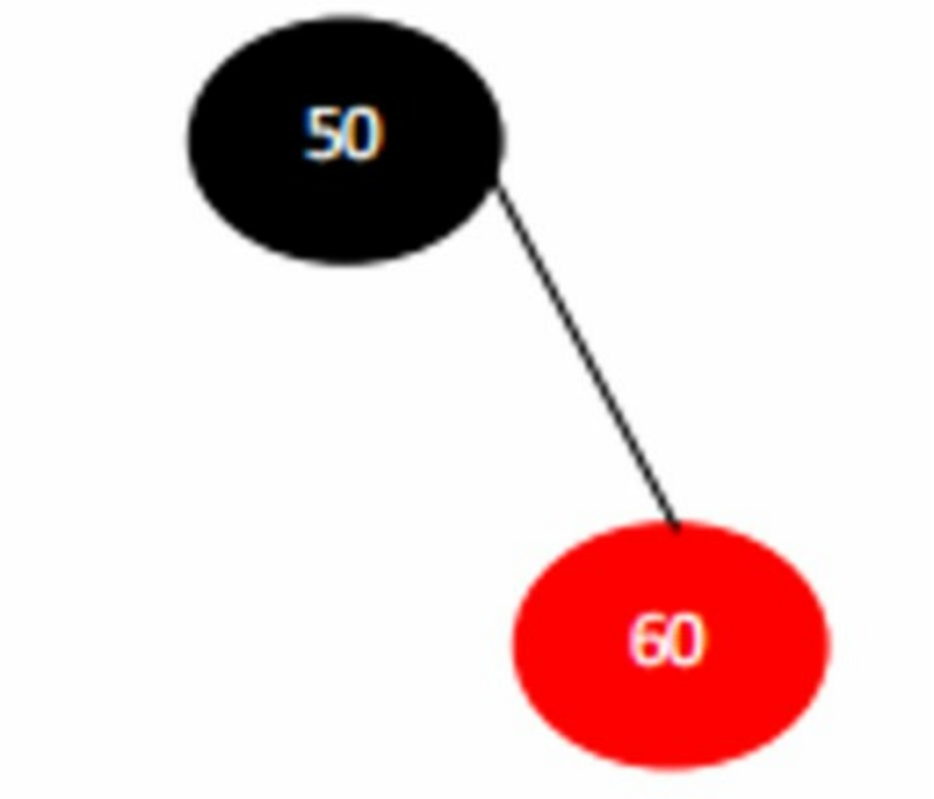


Figure 2.35: *Insertion of 60 in red-black tree*

3. Inserting 70, as shown in *Figure 2.36*:

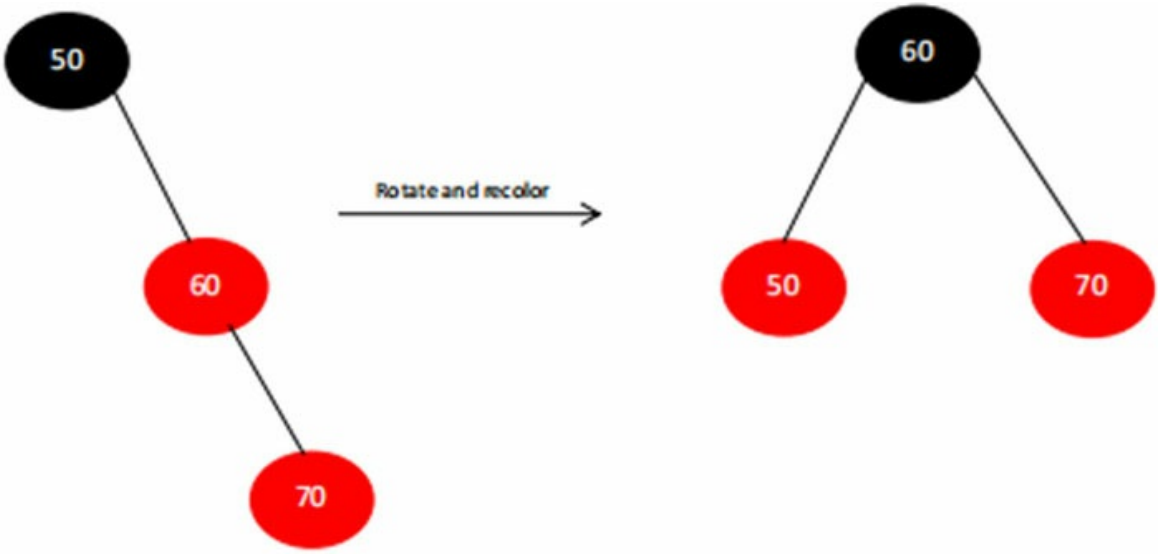


Figure 2.36: *Insertion of 70 in red-black tree*

4. Inserting 90, as shown in *Figure 2.37*:

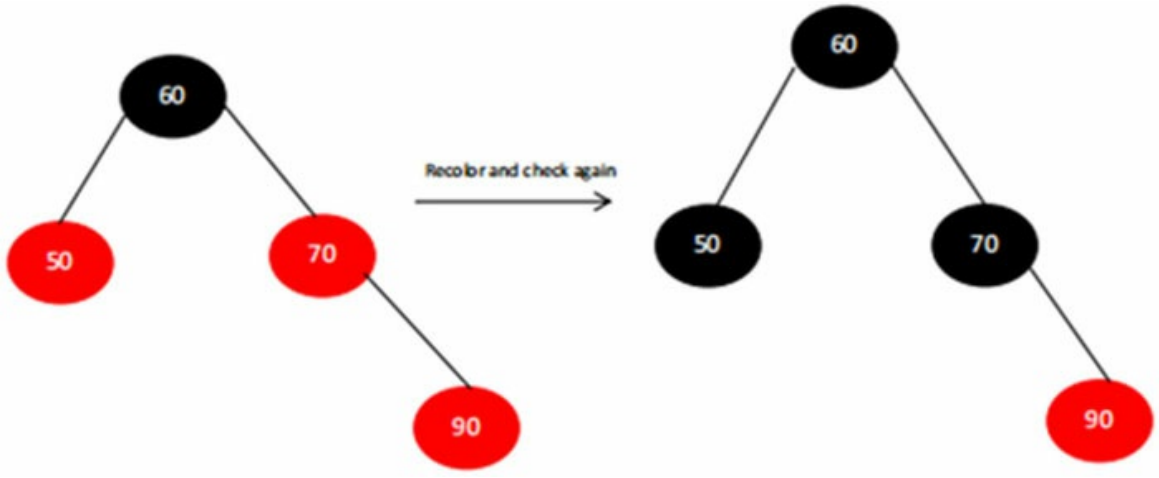


Figure 2.37: *Insertion of 90 in red-black tree*

5. Inserting 20, as shown in *Figure 2.38*:

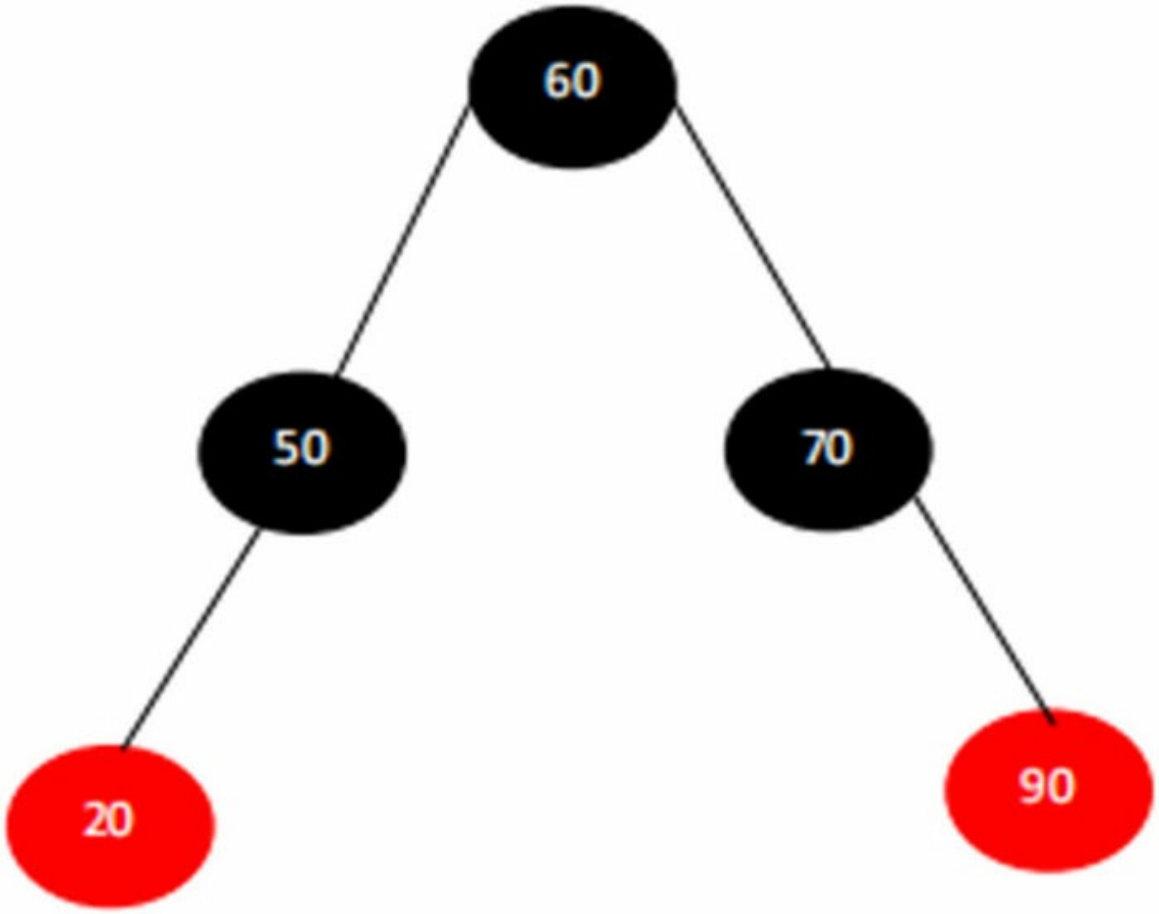


Figure 2.38: *Insertion of 20 in red-black tree*

6. Inserting 15, as shown in *Figure 2.39*:

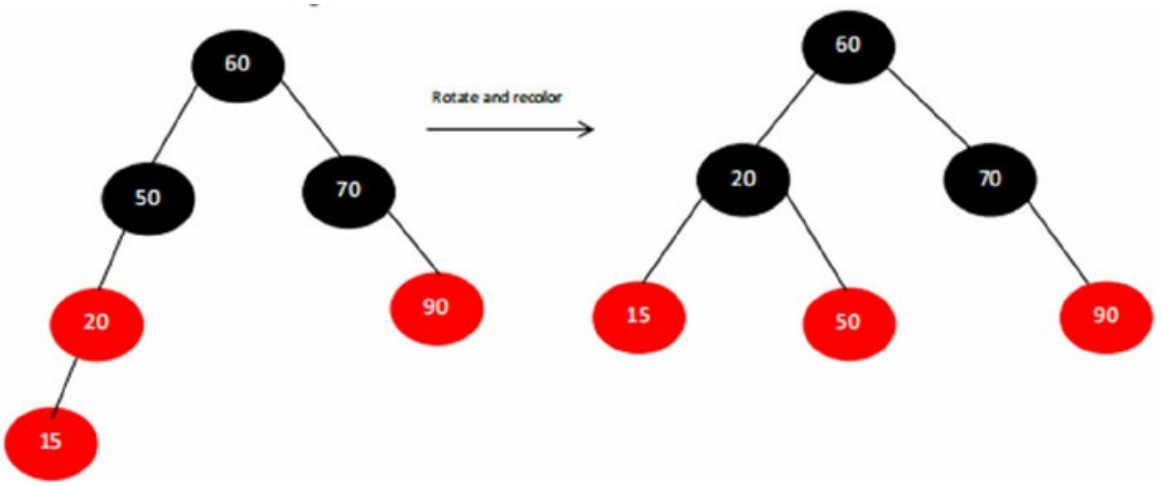


Figure 2.39: *Insertion of 15 in red-black tree*

7. Inserting 10, as shown in *Figure 2.40*:

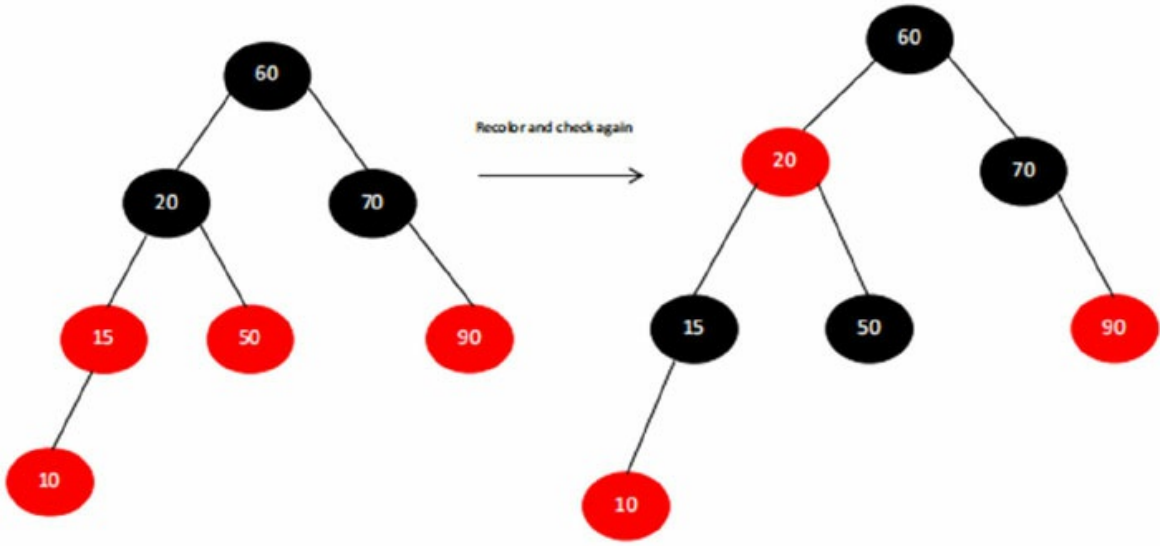


Figure 2.40: *Insertion of 10 in red-black tree*

Tries

A tries is also known as a digital tree, radix tree, or prefix tree in computer science. To allow quick pattern matching, tries is a tree-based data structure for storing strings. Tries are used to retrieve information. The character, and not the key, is stored in each node using tries. The key relates to the path from the root to the node. Tries uses a key's character to direct the search. The string associated with that node's prefix is shared by all the node's

descendants.

Types of tries

Let us now discuss the three different tries of Tries.

Standard tries

The Standard tries for a set of strings S is an ordered tree in which:

- Each node, other than the root, is assigned a character label.
- A node's children are arranged alphabetically.
- The strings of S are produced by the paths leading from the external nodes to the root.

Compressed tries

By compressing chains of redundant nodes, compressed tries are created from regular tries. A requirement to be a compressed tries is:

- Tries should have nodes of degree of at least 2.

Suffix tries

The features of suffix tries are as follows:

- A suffix is a compressed trie that contains all of the text's suffixes.
- The suffix trie for a text X of size n from an alphabet of size d.
- Suffix tries stores all the $n(n-1)/2$ suffixes of X in O(n) space.
- Suffix tries supports prefixes matching queries and arbitrary pattern matching in O(dm) time, where m is the length of the pattern.
- It is possible to construct suffix tries in O(dn) time.
- Applications:
- Word matching.
- Prefix matching.

Heap Sort

The (binary) heap is an array object that is a complete or almost complete binary tree. The leaves are on at most two different levels. The tree is completely filled on all nodes except the lowest.

Types of heaps

There are two types of heaps: Maximum and Minimum heaps.

Maximum Heap

Max heap is a binary tree in which every parent node contains a value greater than or equal to the child nodes.

Key-value of all nodes > key value of sub-tree

Refer to Figure 2.41:

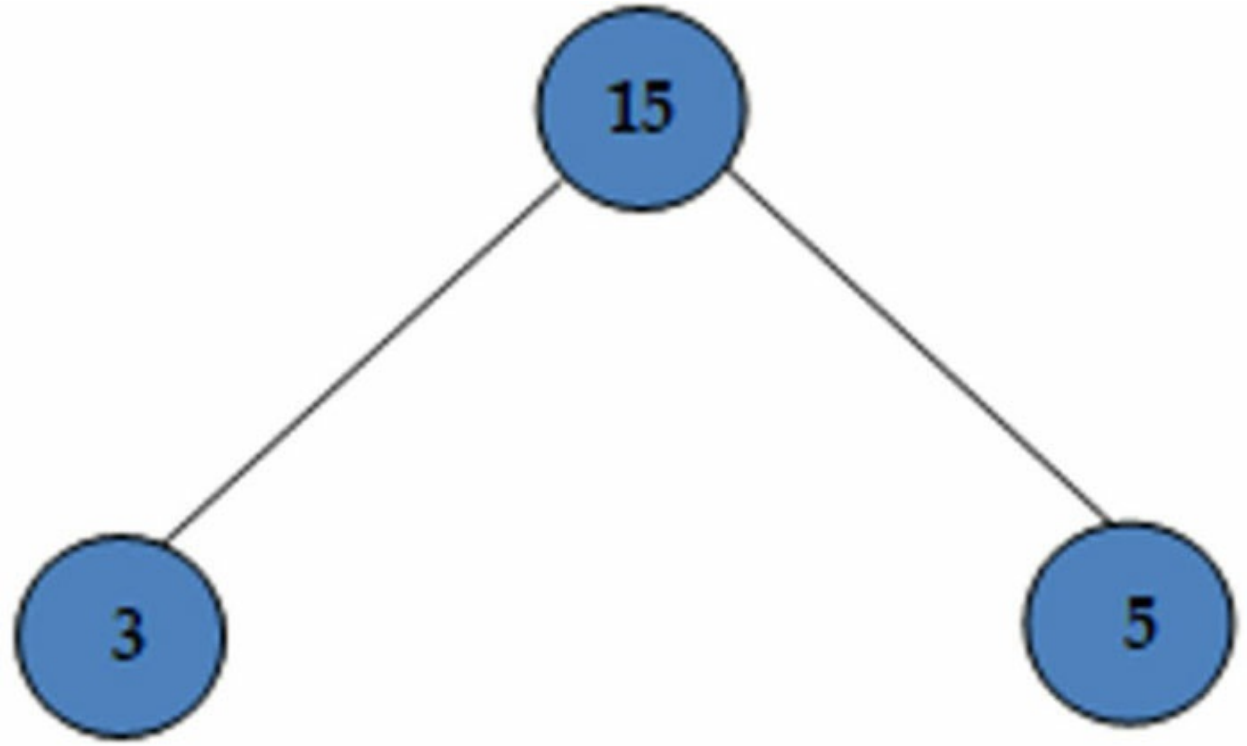


Figure 2.41: *Maximum Heap*

Minimum heap

Min heap is a binary tree in which every parent node contains values smaller than or equal to the child nodes.

Key-value of all nodes < key value of sub-tree

Refer to Figure 2.42:

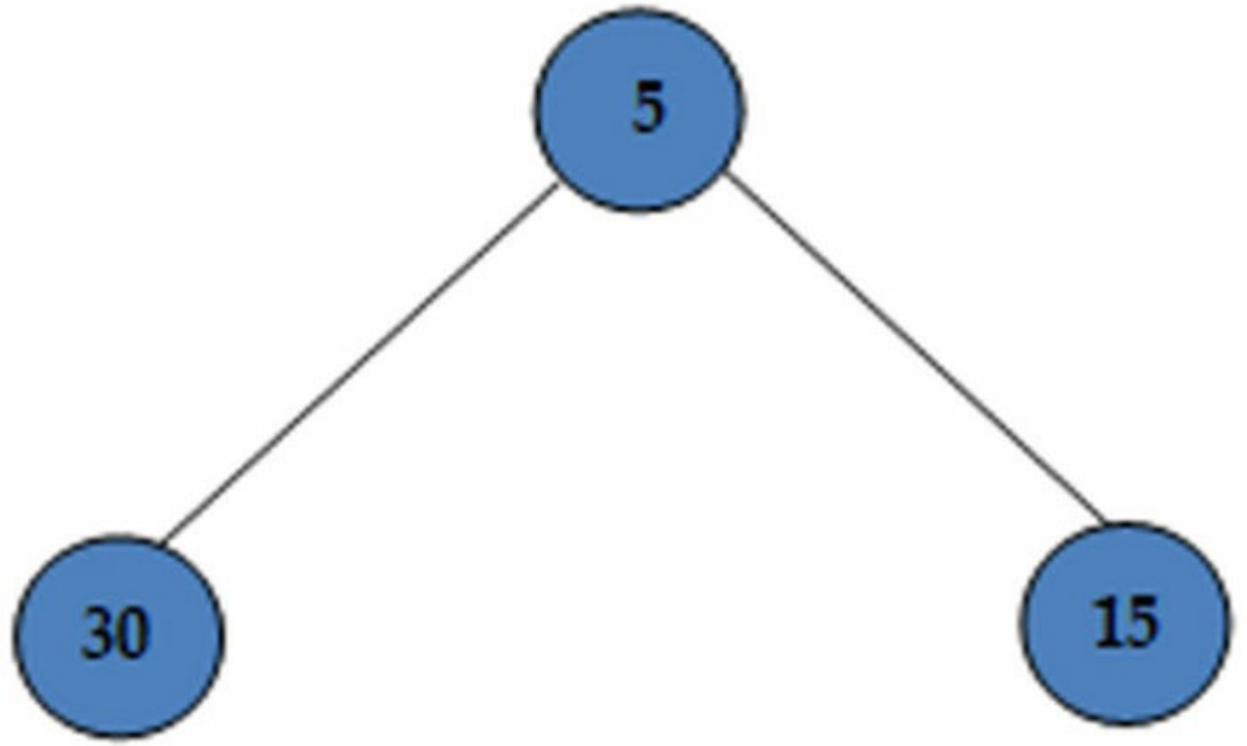


Figure 2.42: *Minimum Heap*

ReHeap-Up

Assume we have an almost complete binary tree with N elements, whose 1st N-1 elements satisfy the order property of the heap, but the last element does not. The ReHeap-up operation repairs the structure so that it is a heap, by floating the last element up the tree, until that element is in its correct location in the tree.

Consider a maximum heap in which we want to insert a node value 35, using ReHeap-Up, as shown in *Figure 2.43*:

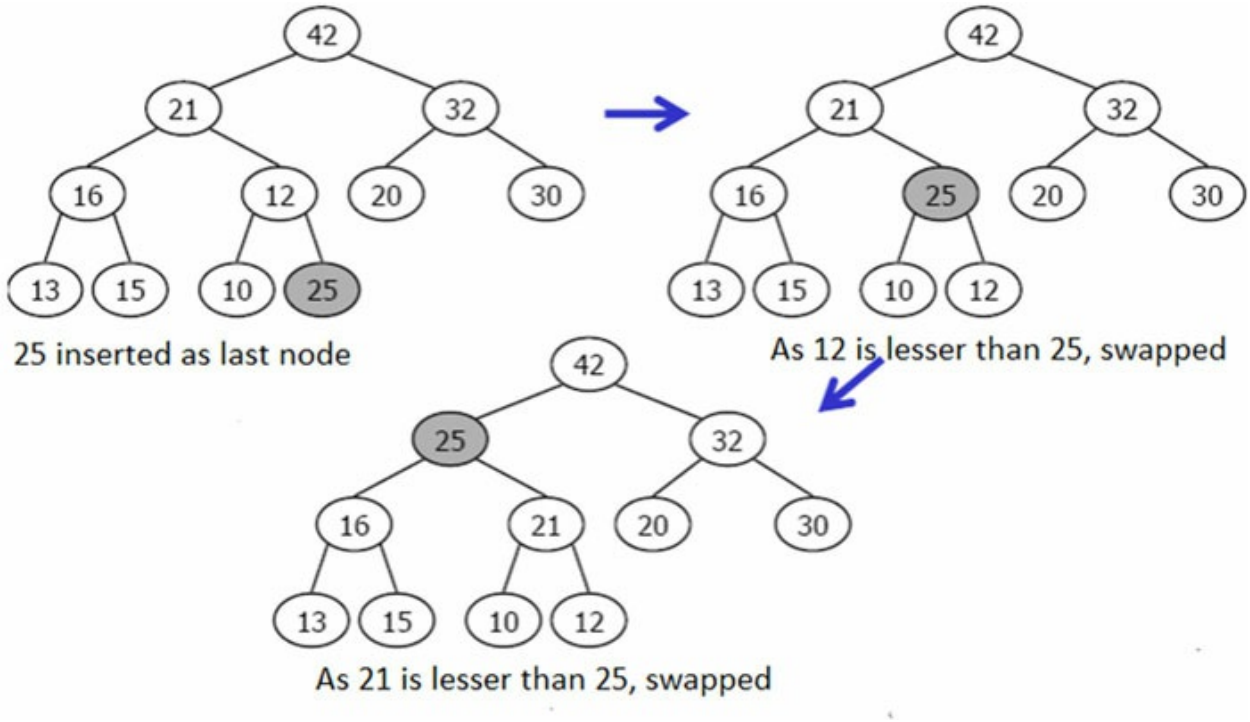


Figure 2.43: *ReHeap-Up Operation*

ReHeap-Down

Assume we have an almost complete binary tree that satisfies the order property of the heap, except in the root position. This situation occurs when the root is deleted from the tree, leaving 2 disjoined heaps. To correct the situation, we move the data in the last tree node to the root. Obviously, this action destroys the heap properties. To restore the heap property, we use re-heap down.

Consider a minimum heap in which we want to delete a node, Using ReHeap-Down, as shown in *Figure 2.44*:

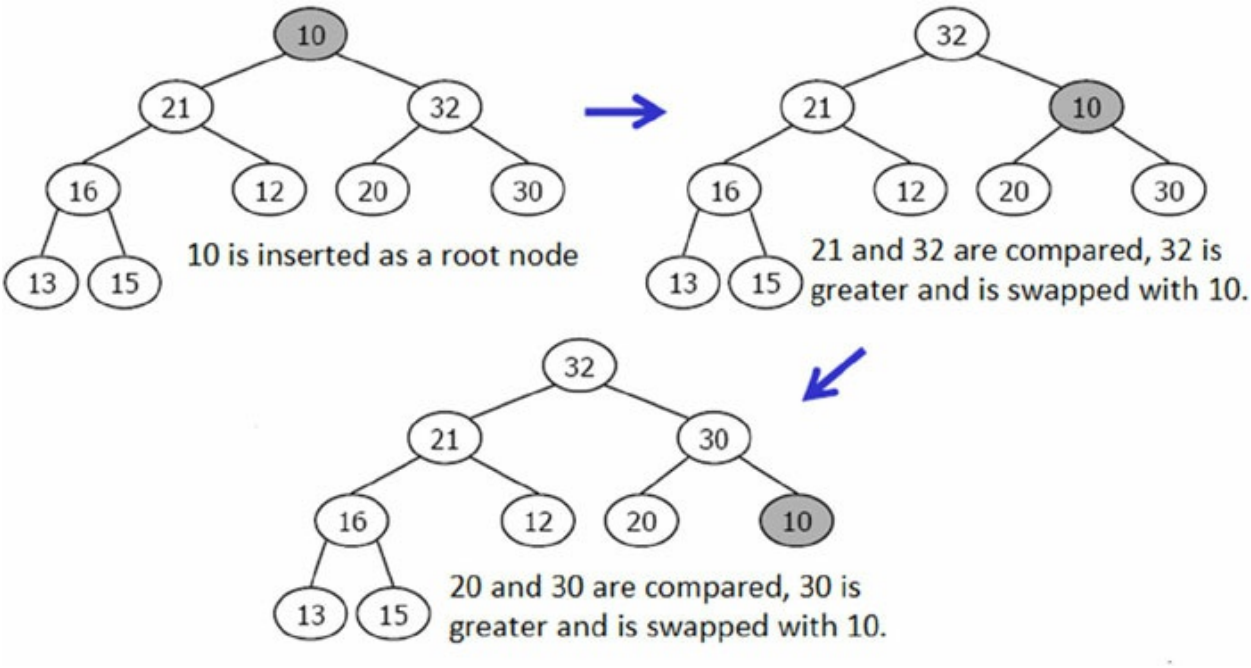


Figure 2.44: *ReHeap-Down Operation*

Algorithm for heap sort

Let us now look at the algorithm:

```
void heap(int a[], int n, int m)
{
 int largest = m;
 int l = 2*m+ 1;
 int r = 2*m + 2;
 // if left child node is larger than root node
 if (l < n && a[l] > a[largest])
  largest = l;
 // if right child node is larger than largest so far
 if (r < n && a[r] > a[largest])
  largest = r;
 // if largest node is not root node
 if (largest != m)
 {
  swap(a[m], a[largest]);
heap(a, n, largest);
 }
}
void heapsort(int a[], int n)
{
 for (int m = n / 2 - 1; m >= 0; m--)
heap(a, n, m);
```

```
  for (int m=n-1; m>=0; m--)
  {
   swap(a[0], a[m]);
heap(a, m, 0);
  }
}
int main()
{
  int a[] = {121, 20, 130,46, 27};
  int n = sizeof(a)/sizeof(a[0]);
heapsort(a, n);
  count << “Sorted array is \n”;
printArray(a, n);
}
```

Example 5

Create a Max-Heap & Min-Heap for the following sequence: 10, 20, 4, 8, 21, 50, 17.

Max-Heap:

Refer to the following *Figure 2.45*:

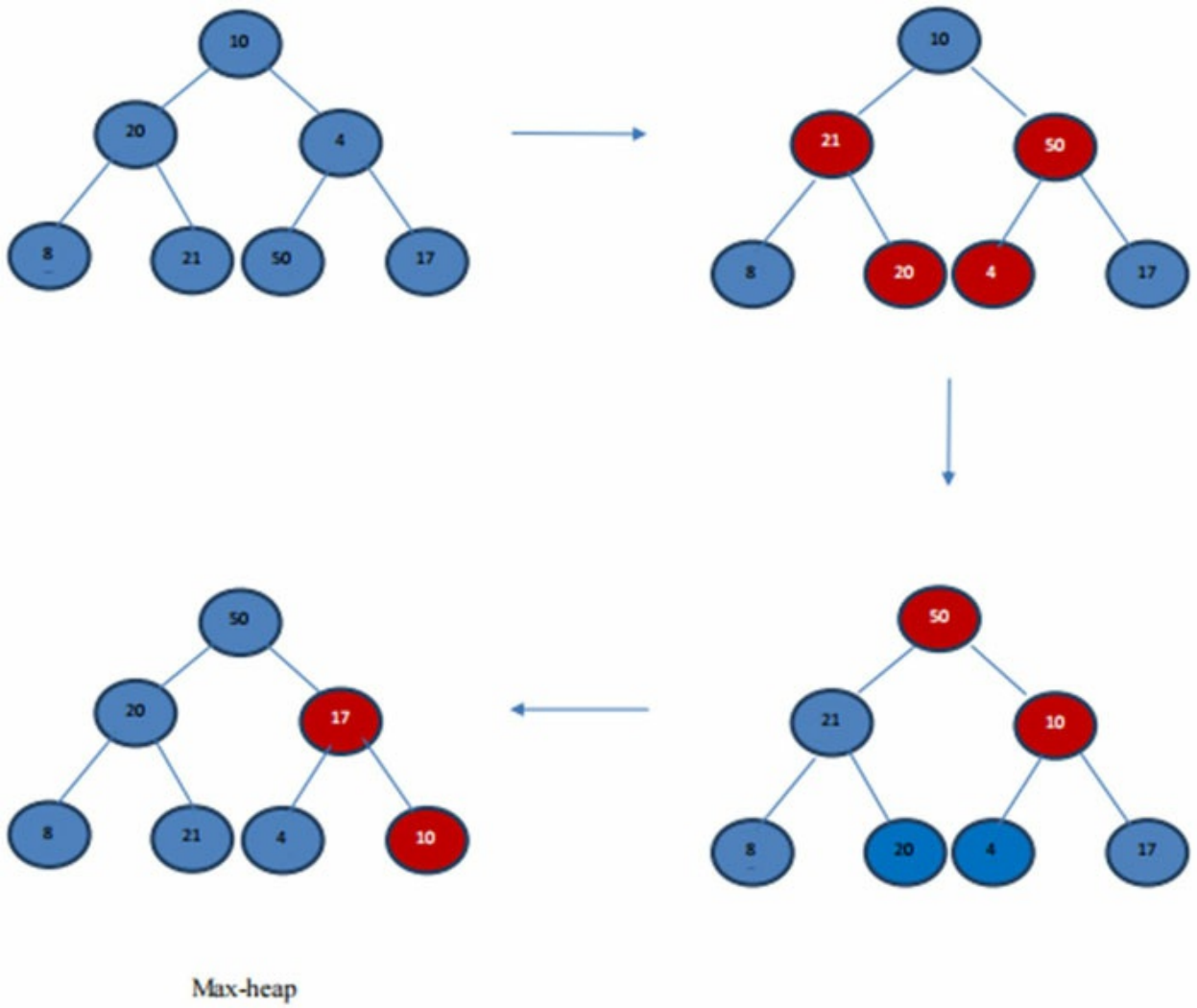


Figure 2.45: *Construction of Max-Heap*

Here,

- 8 & 21 are compared, 21 is greater- swapped with its parent node 20.
- 50 &17 are compared, 50 is greater- swapped with its parent node 4.
- 21 & 50 are compared, 50 is greater- swapped with its parent node 10.
- 17 & 4 are compared, 17 is greater- swapped with its parent node 10.

Min-Heap:

Refer to the following *Figure 2.46*:

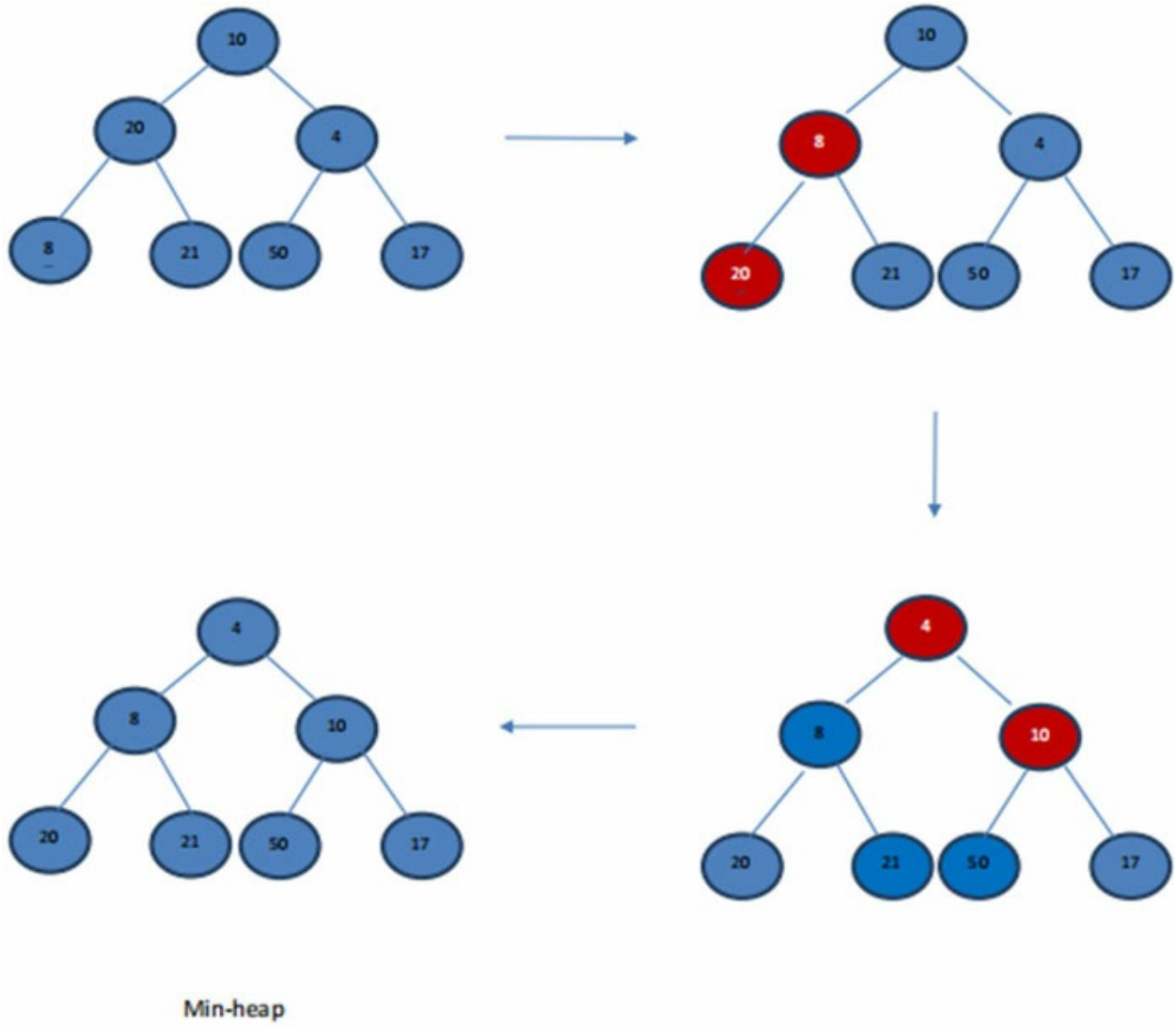


Figure 2.46: *Construction of Min-Heap*

- 8 & 21 are compared, 8 is lesser- swapped with its parent node 20.
- 50 &17 are compared, 17 is lesser- as its parent node 4 which is lesser, and thus, it is not swapped.
- 8 & 4 are compared, 4 is lesser, and so it is swapped with its parent node 10.
- 50 & 17 are compared, 17 is lesser, as its parent node 10 which is lesser. Thus, it is not swapped.

B Trees

An m-way tree, or a tree where each node may have up to ‘m’ children, is

referred to as a B-tree of order m.

- Each non-leaf node has one fewer key than the number of children it has, and these keys divide the keys in the children into groups resembling a search tree.
- Every leaf is at the same height.
- Every non-leaf node has at least [m/2] children, with the exception of the root.
- The root either has two to m children, or it is a leaf node.
- A leaf node can hold m-1 keys at the most.
- “m” should always be an odd number.

Refer to the following *Figure 2.47* for an example of B-tree:

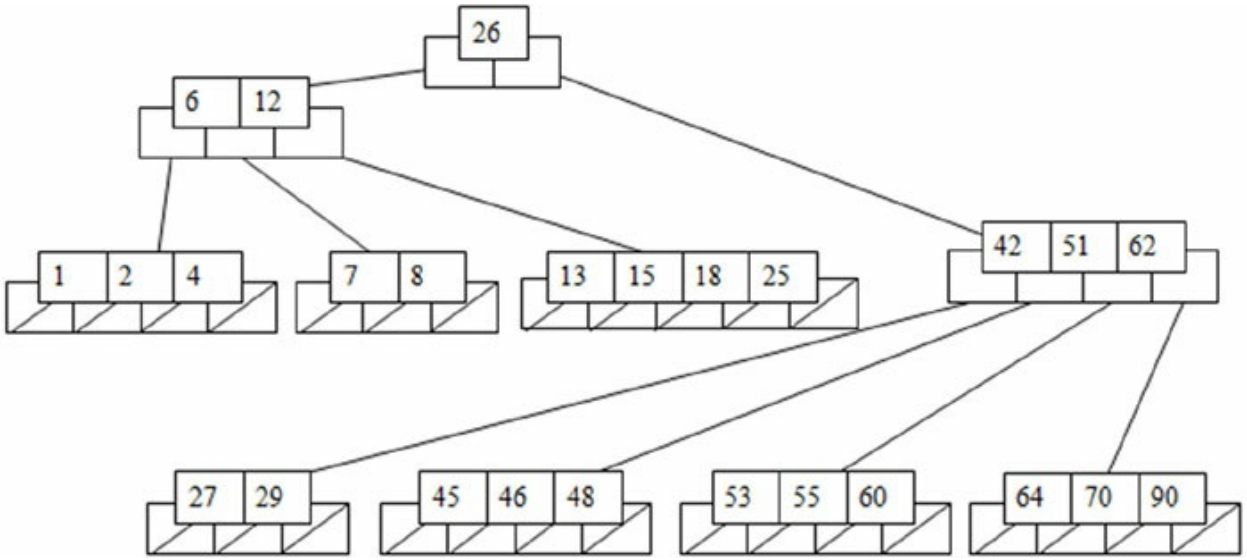


Figure 2.47: *Example of a B-tree of order 5*

Constructing a B-Tree

Let us say the B-tree is empty when we begin, and the key appears as follows:

1,12,8,25,5,14,28,17,7,52,16,48,68,3,26,29,53,55,45

Let us consider we want to build a B-tree of order 5. The first four keys go into the root, as shown in *Figure 2.48:*

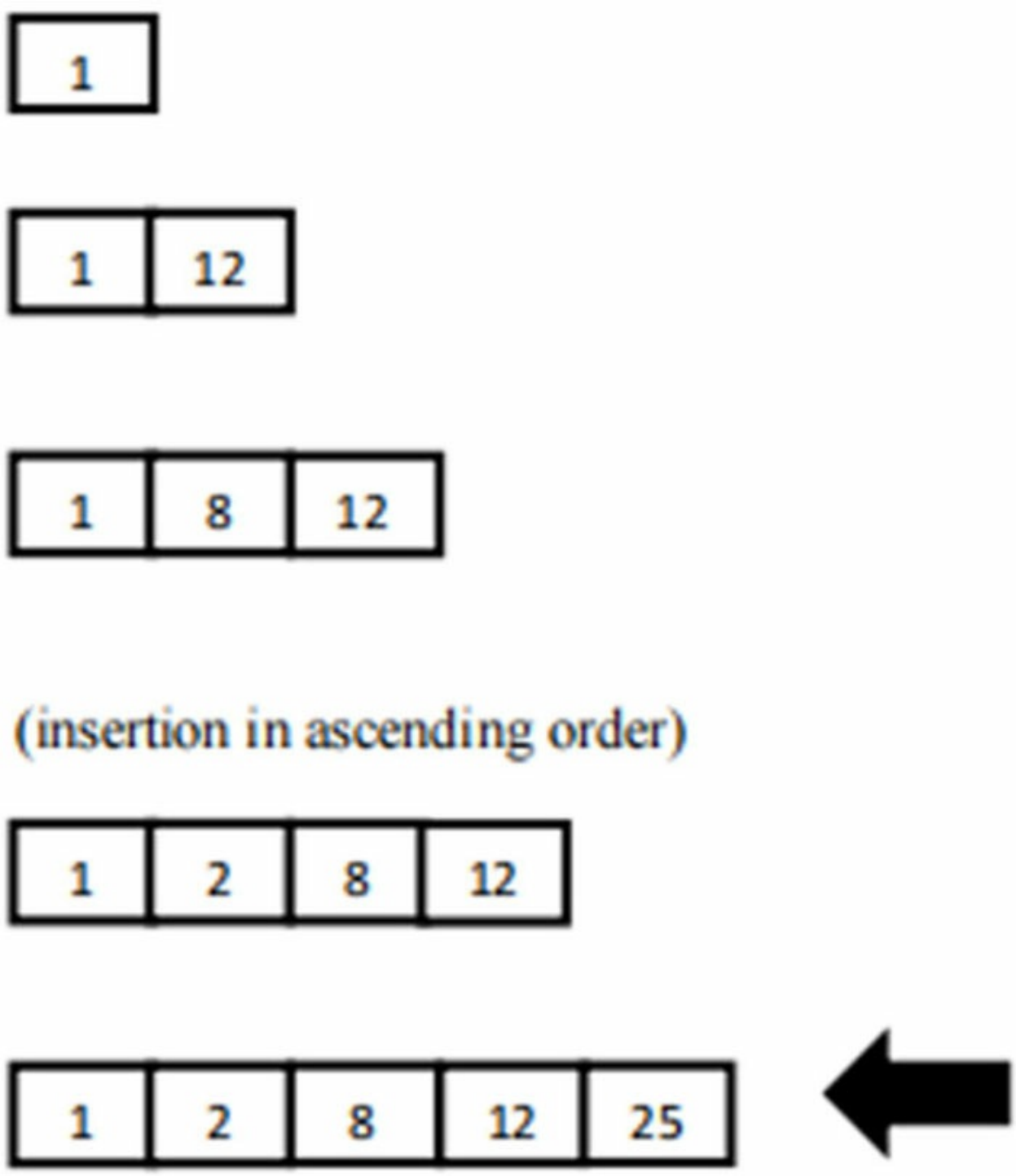


Figure 2.48: *Insertion of 1,12,8,2,25 into the B-tree*

We have to split the 5 keys once added because it is a B-tree of order 5, and hence the root node can only have a maximum of 4 keys.

To put the fifth key in the root would violate condition 5. Therefore, when 25

arrives, pick the middle key to make a new root, as shown in *Figure 2.49*:

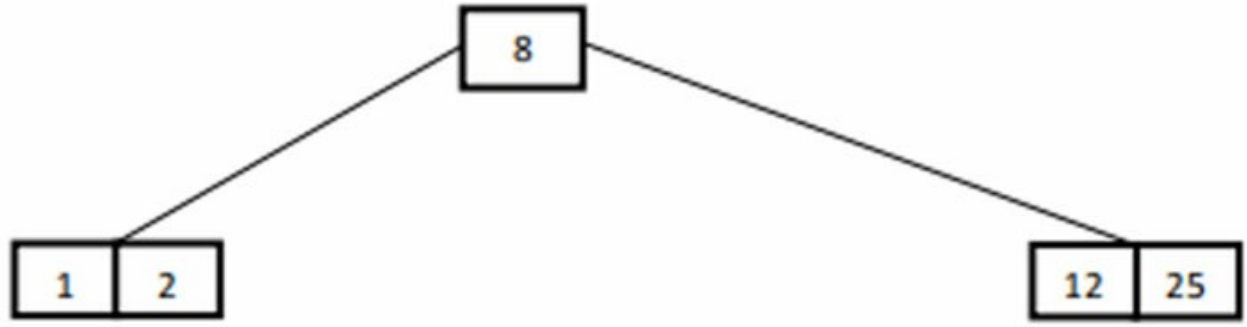


Figure 2.49: *Splitting of a B-Tree*

5, 14, 28 goes into the leaf node, as shown in *Figure 2.50*:

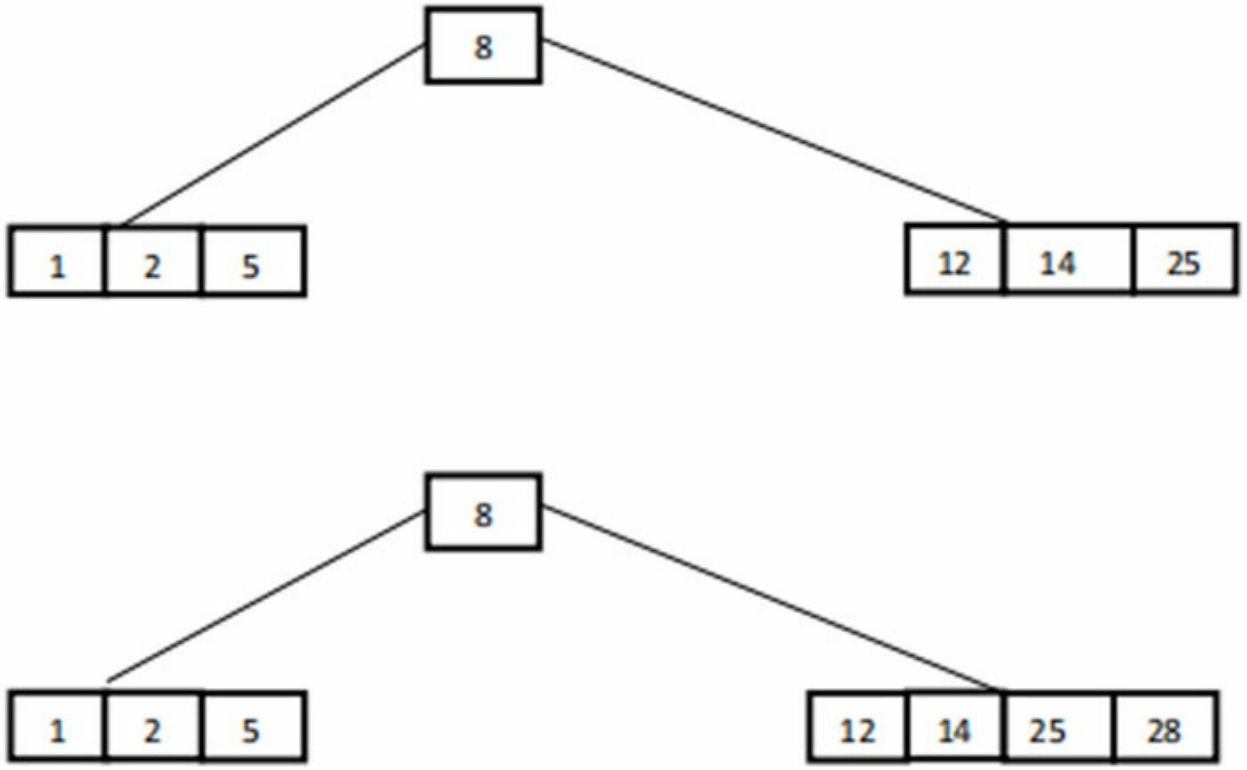


Figure 2.50: *Insertion of 5, 14, and 28into the B-tree*

We take the middle key, promote it (to the root), and split the leaf because adding 17 to the right leaf node would overfill it. Refer to the following *Figure 2.51*:

Figure 2.51: *Insertion of 17 into the B-tree*

Split the lead as shown in the following *Figure 2.52*:

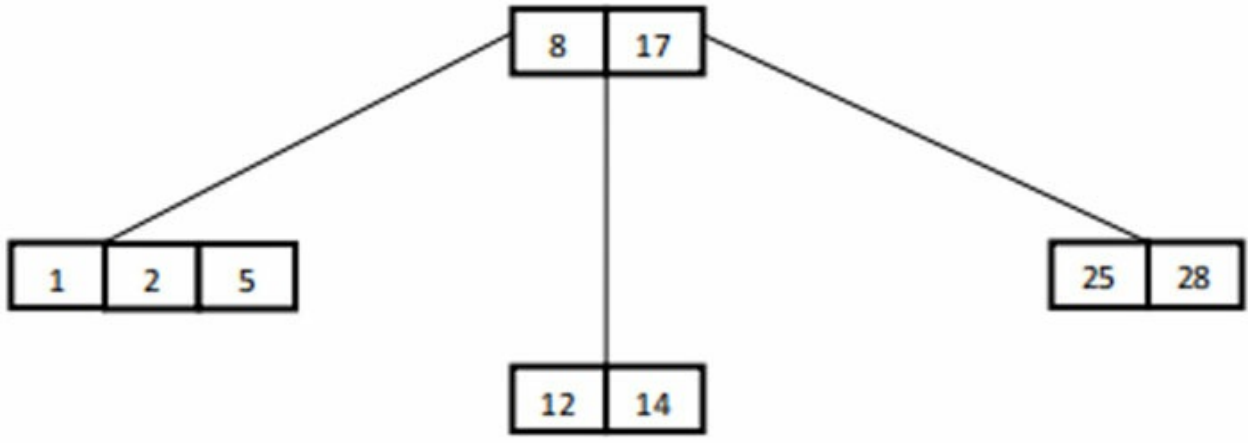


Figure 2.52: *Splitting of a B-tree*

7 and 52 get added to the leaf nodes, as shown in *Figure 2.53*:

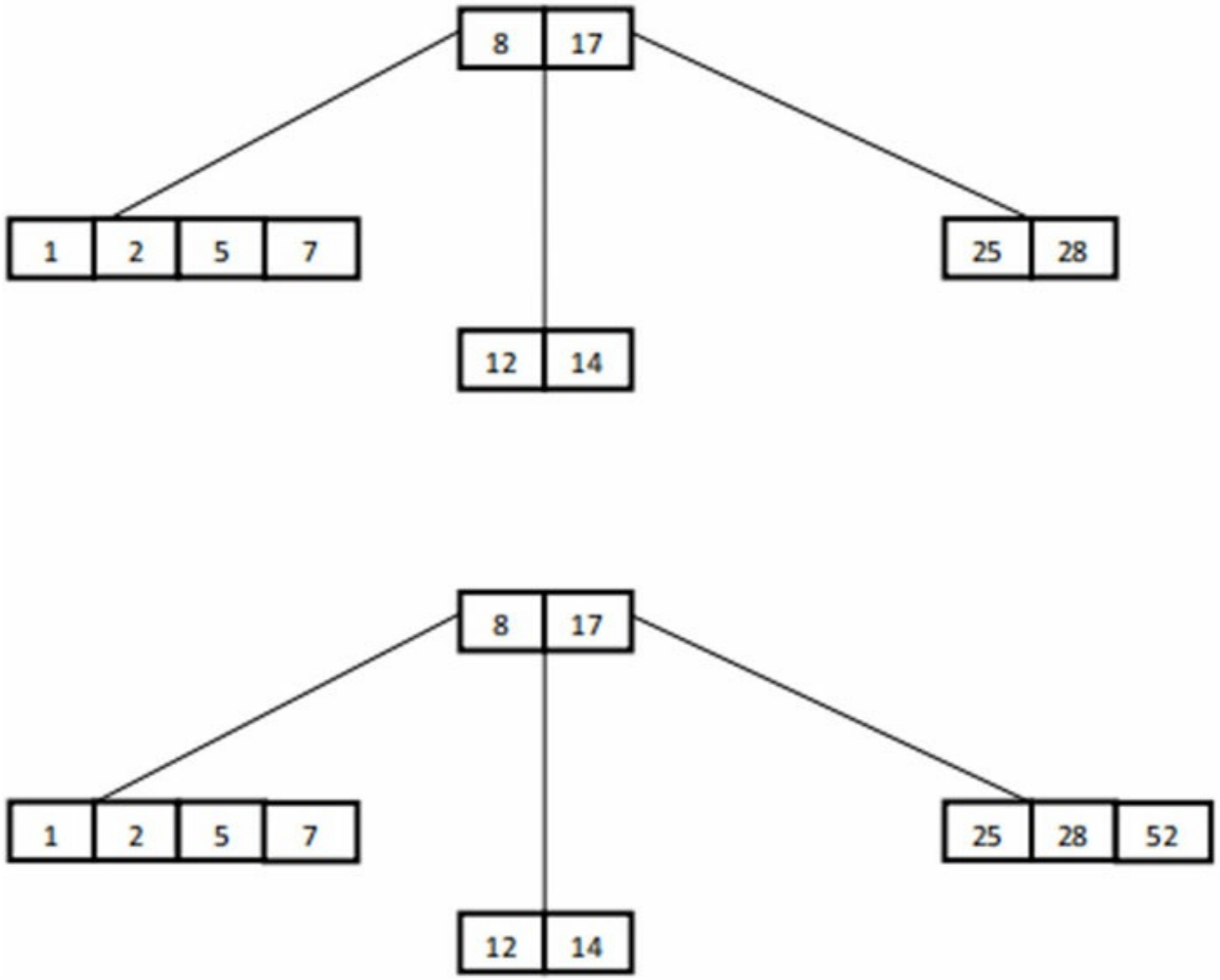


Figure 2.53: *Insertion of 7 and 52into the B-tree*

16 and 48 get added to the leaf nodes, as shown in *Figure 2.54*:

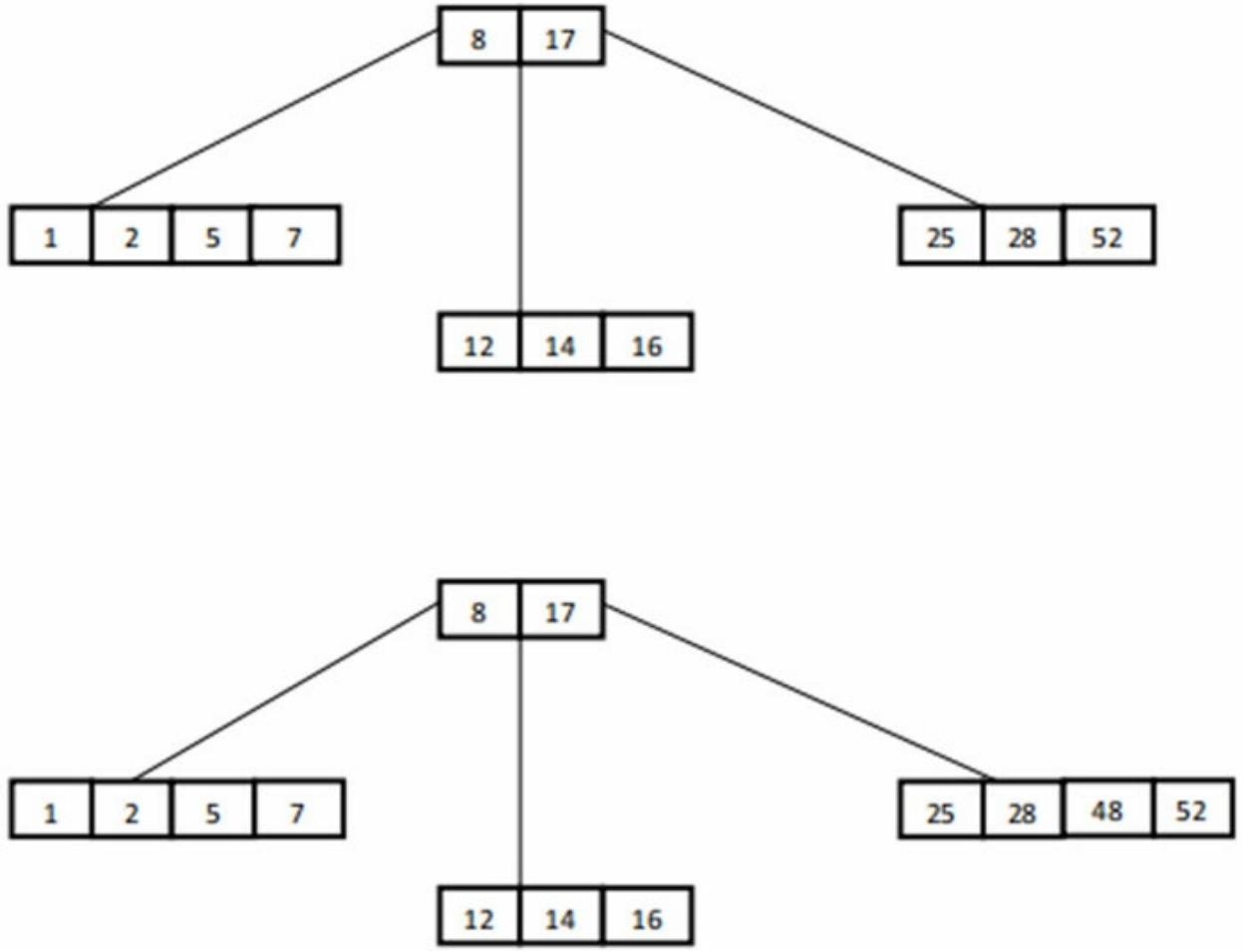


Figure 2.54: *Insertion of 16 and 48 into the B-tree*

We divide the rightmost leaf by adding 68. We promote 48 to the root, and then we split the leftmost leaf by adding 3>, as shown in the following *Figure 2.55*:

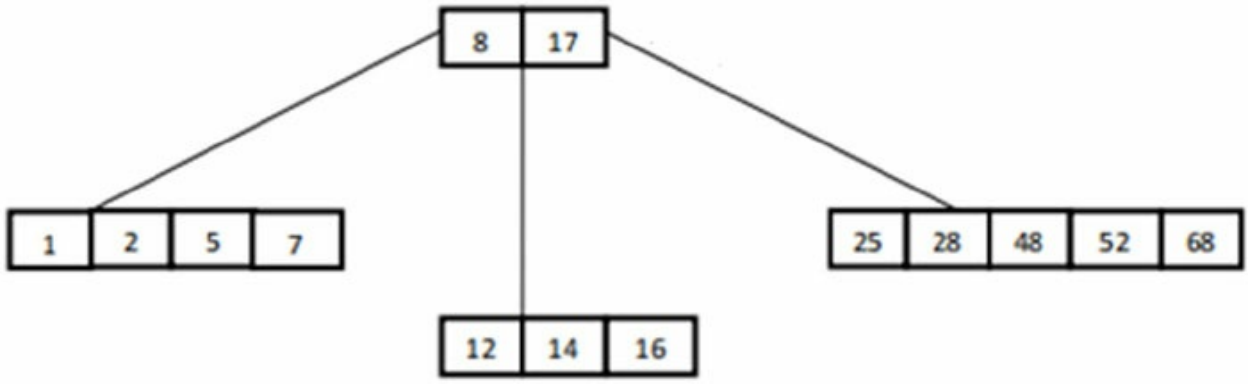


Figure 2.55: *Insertion of 68 into the B-tree*

Also refer to *Figure 2.56*:

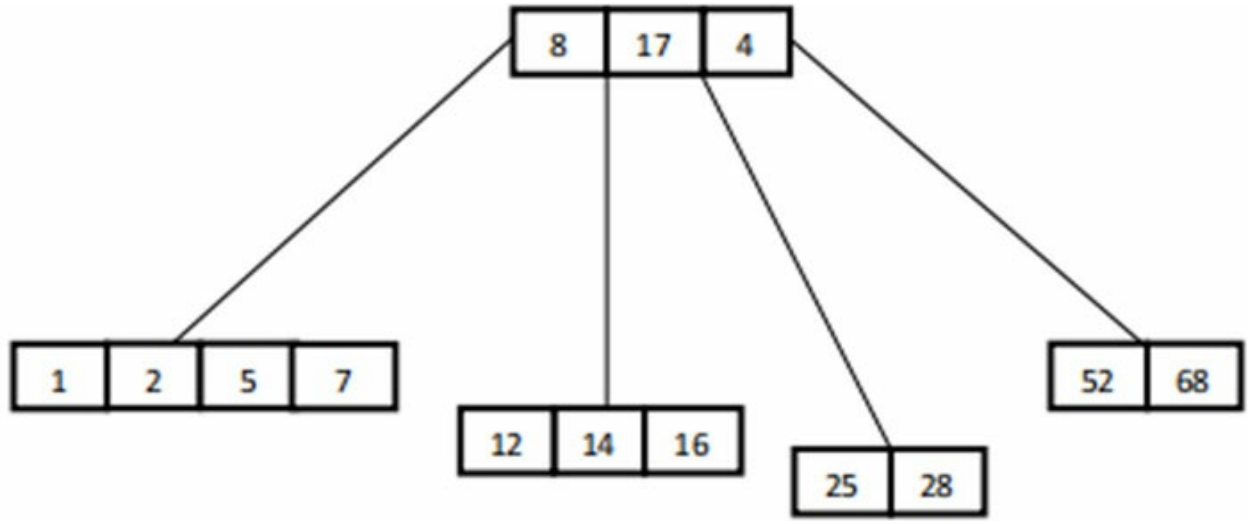
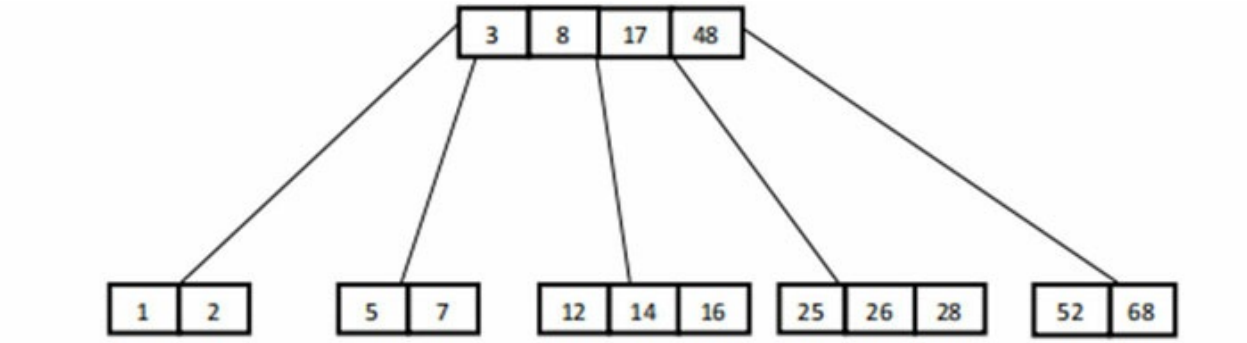


Figure 2.56: *Splitting of a B-tree*

Then, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves. Refer to the following *Figure 2.57*:



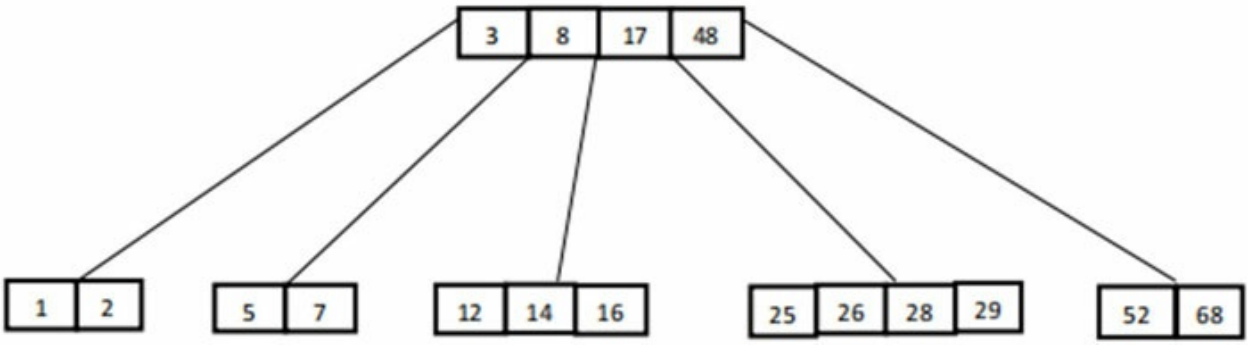


Figure 2.57: *Insertion of 3 into the B-tree and split*

Also, refer to the following *Figure 2.58*:

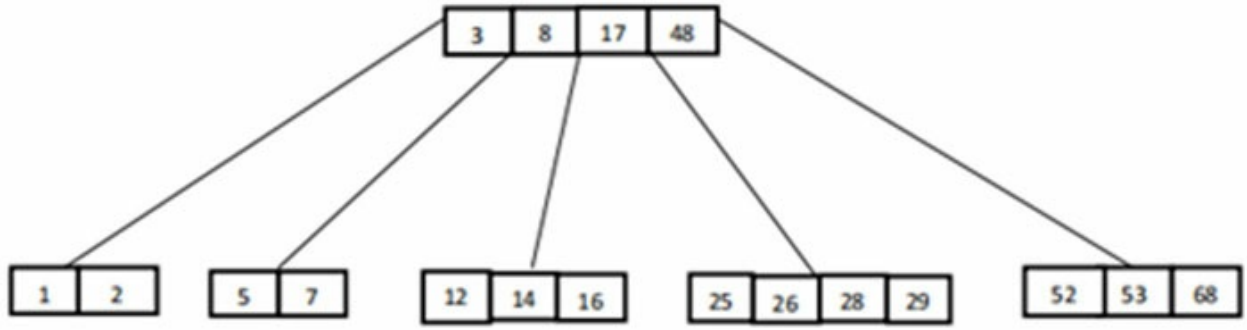


Figure 2.58: *Insertion of 26, 29, 53 and 55 into the B-tree*

Adding 45 causes a split of 25,26,28,29 and promoting 28 to the root then causes it to split. Refer to the following *Figure 2.59*:

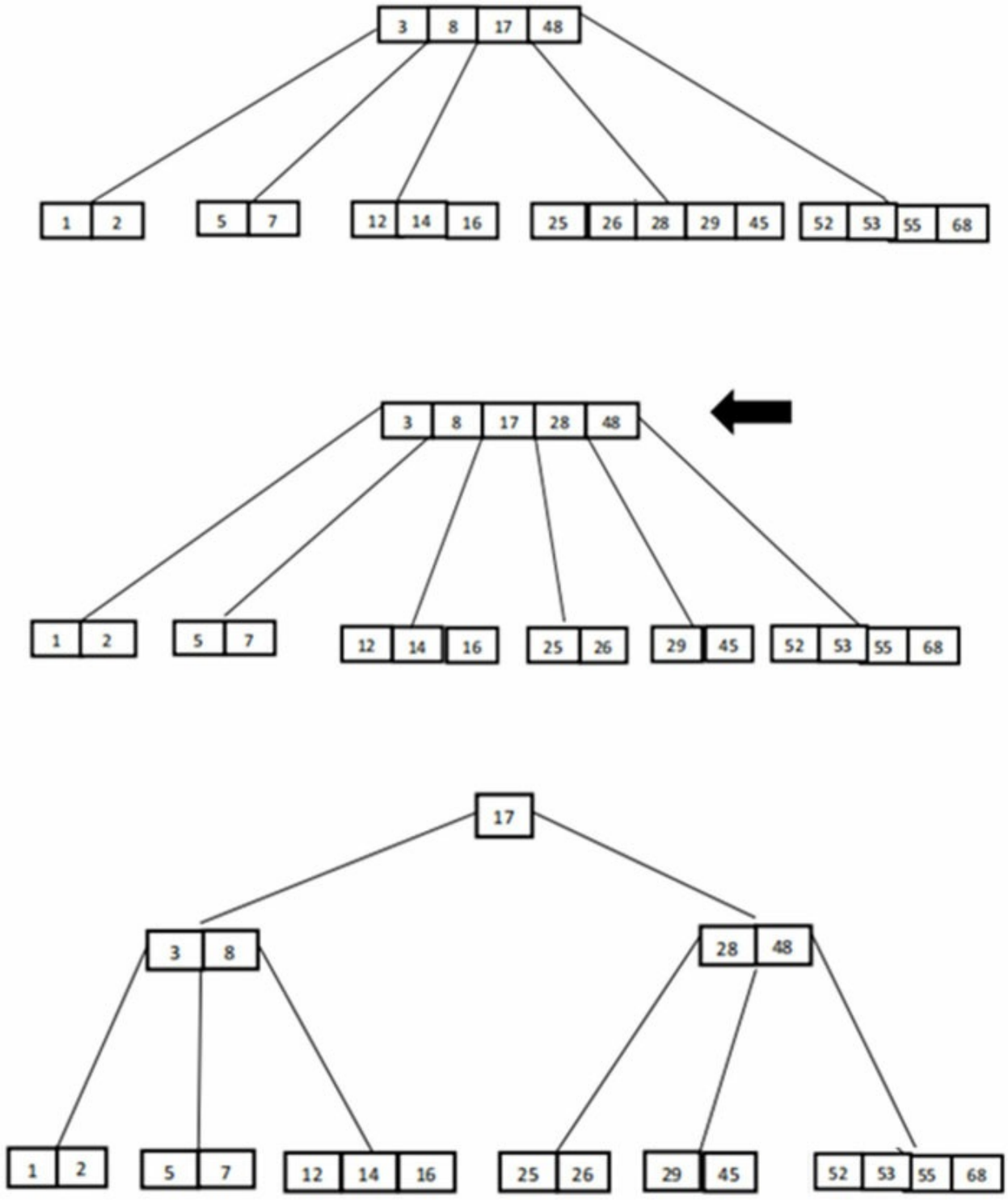


Figure 2.59: *Splitting of a B-tree*

Conclusion

A data structure is of 2 types: Primitive and Non-Primitive. Non-primitive

data structures are divided into 2 categories: linear and non-linear data structures. An array is a collection of similar data elements. A stack follows LIFO, where operations are done at the only one end. A queue follows a FIFO, where operations can be done at both the ends, such that every example of data structure has its advantages as well as disadvantages. AVL trees require O(log n) time for simpler operations. AVL trees are useful for searching elements. Balancing capabilities of AVL tree is the major factor, that is, if the tree is unbalanced, then it will take longer time for operations to be performed. Red-Black trees are similar for simpler operations that take O(log n) time as AVL trees. The properties should be noted while inserting the nodes. B-tree offers O(log n) for deletion and insertion of the elements. Unlike a hash table, B-tree provides an ordered sequential access to the index. B-tree provides iteration of keys as well. Tries is also a tree that stores characters or strings. It offers O(n), where *n* is the length of the key for insertion, searching and deletion operations. Unlike hash tables, Tries easily print every word in alphabetical order. But it needs a lot of memory for storing the characters. Therefore, we can say that tries are faster but require a huge memory for storing characters. Huffman algorithm is easy to implement and it helps in shortening down the messages.

Key facts

- Advanced data structure is the one used for organizing, storing, and managing the data in a more efficient manner with ease of access and modification.
- AVL Tree is determined as a height balanced binary search tree, where each node is associated with balance factor. It is computed by finding the difference between the height of the left subtree and height of the right subtree. The balance factor should be -1,0 or 1 for an AVL tree.
- Huffman Algorithm is a kind of data compression technique, used to encode the most frequent characters by shorter length of bits of characters without losing much data. It takes a variable-length code to encode any given source character.
- 2-3 Tree is a type of tree structure, in which each internal node represents either two children with one data element or three children with two data elements.

- Red-Black Trees, a self-balancing type of binary search tree used to represent each node of the tree with an extra bit, where an extra bit is often interpreted as either in red or black color.
- Tries is known for sorted tree-based data structure or radix trees. It is typically used to store a set of dictionary words, in which any word in the dictionary can be searched using the word's prefix. It helps to generate the finite alphabet completion list in an efficient way.
- Heap Sort is one of the popular and more efficient sorting algorithms in data structure. It is a kind of selection sort, where the minimum element is located and placed in the beginning of the tree. The same process is repeated for the remaining elements until all elements are placed in the tree structure. Hence, it is known as an in-place algorithm.
- B-Trees is a special kind of m-way based tree structure applied mostly for disk access. A B-Tree of order m can have a maximum of 'm-1' number of keys with 'm' children. It is capable of organizing and to manage enormous data storage and facilitates search operation.

Questions

1. Define AVL tree. Construct AVL tree for the following data:

   21,26,30,9,4,14,28,18,15,10,2,3,7 **[MU MAY'19(10 MARKS)]**
2. Write a note on AVL trees **[MU DEC'18 (5 MARKS)]**
3. What is the optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers? **[MU MAY'19 (5 MARKS)]**

   a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21
4. Explain Huffman Algorithm. Construct Huffman tree for **MAHARASHTRA** with its optimal code. **[MU DEC'18 (10 MARKS)]**
5. Explain Red Black Trees [MU DEC'18 (5 MARKS)] [(MU MAY'19 (5 MARKS)][MU DEC'19 (5 MARKS)]
6. Define AVL tree. Construct AVL tree for the following data:

   63,9,19,27,18,108,99,81 **[MU DEC'19(10 MARKS)]**
7. Explain Huffman Algorithm. Construct Huffman tree & find Huffman code for the message: **KARNATAKA [MU DEC'19 (10 MARKS)]**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3

Divide and Conquer

Introduction

This module discusses divide and conquer technique with the basic introduction and various methods that are involved in it with suitable examples. In computer science, divide and conquer is an algorithm design paradigm. Divide and Conquer is the simplest and easiest technique of decomposing a larger problem into simpler problems, to solve any given problem statement.

Binary Search is a searching algorithm for finding an element's position in a sorted array. It follows the divide and conquer approach, in which the list is divided into two halves, and the item is compared with the middle element of the list.

The Max-Min problem is used to find a maximum and minimum element from the given array. We can effectively solve it using this approach.

Merge sort is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves.

Quicksort is the widely used sorting algorithm that makes **`nlogn`** comparisons in the average case, for sorting an array of n elements. It picks an element as pivot and partitions the given array around the picked pivot.

Strassen algorithm is a faster recursive method for matrix multiplication, where it divides the matrix into four sub-matrices of dimensions n/2 x n/2 in each recursive step.

Structure

In this chapter, we will discuss the following topics:

- Divide and Conquer
- Binary Search

- Finding the minimum and maximum
- Merge-Sort
- Quick Sort
- Strassen's Matrix Multiplication

Objectives

By the end of this chapter, the learner will understand the concept of Divide and Conquer techniques. The learner will get a clear view on the various algorithms that come under Divide and Conquer techniques. The Time complexity of various algorithms is represented with different cases.

Divide and conquer

The divide and conquer technique divides the larger problem into smaller subproblems and the solution is the combination of those smaller subproblems.

Divide and Conquer solves a problem in 3 steps, as depicted in *Figure 3.1*:

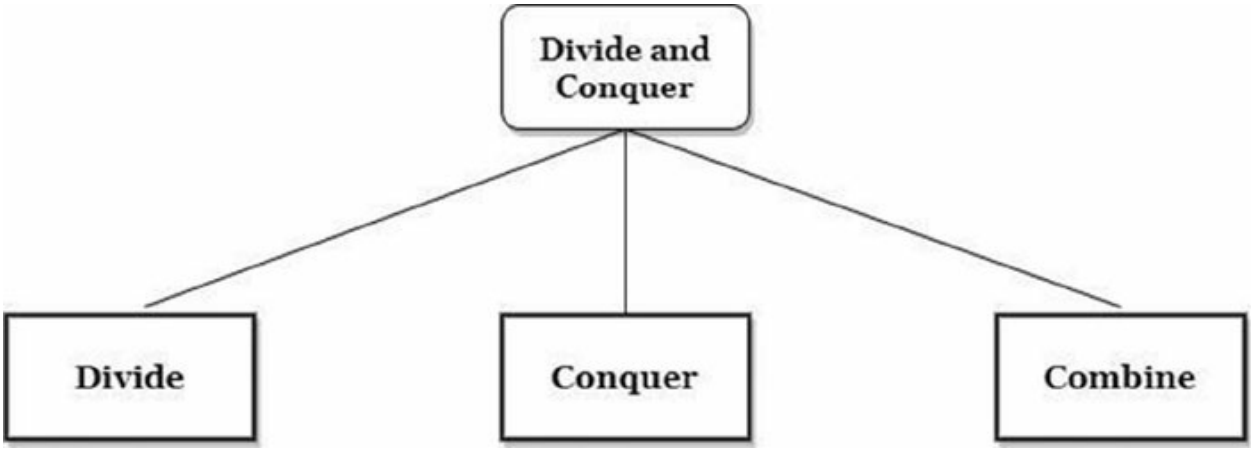


Figure 3.1: *Major steps of Divide and Conquer Technique*

The steps in Divide and Conquer are as follows:

1. **Divide:** This step breaks the larger problem recursively into smaller problems.
2. **Conquer:** This step involves recursively solving the sub-problems independently.

3. **Combine**: In this step, combine solutions of the smaller subproblems to get the solution of the original big problem.

Subproblems are like the original problem having smaller arguments and therefore they can be solved independently. When the subproblem comes to the smallest possible size, then they can be combined to generate a proper solution for bigger or the given original problems.

Divide and conquer should be used when the same subproblems are not evaluated many times.

In this module, some divide and conquer algorithms are explained such as binary search, max-min problem, sorting algorithms, that is, quicksort and merge-sort, as well as Multiplication of matrices (Strassen's algorithm).

Binary search

Binary Search is a search algorithm that finds the position of the target value from the sorted array. Binary Search is also known as Half-Interval Search or Logarithmic Search. This algorithm works on the principle of Divide and Conquer, and is implemented only on a sorted list of an array. This algorithm compares the target value to the middle value of the array. If they are not equal, the half in which the target value is not present, is removed, and the search continues on the remaining half. This procedure is repeated until the target value is found.

Algorithm

Let us now look at the algorithm:

```
Input :   A = sorted array
N = size of array
Key = value to be searched
Search (A, Key, N)
Set start = 0
Set end = N-1
While (start < end)
  Set mid = (start + end) /2
    If (Key == A[mid]) then
    Return mid
    Else if (Key < A[mid]) then
    Set end = mid – 1
    Else
```

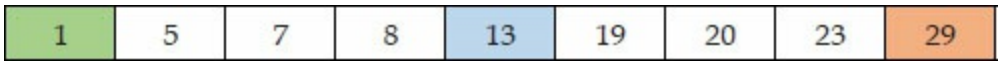
```
  Set start = mid + 1
  End
End
Return (-1)    //value not found
```

Example 1

Sorted array

Value to be searched is **Key = 23**

Step 1



Here, **A[mid] = 13 <23**

That is, **A[mid] < Key**

Hence, **start = mid+1 = 5 and mid = (start +end) /2 = 13/2= 6**

Step 2

Here, **A[mid] = 20 <23**

That is, **A[mid] < Key**

Hence, **start = mid + 1 = 7 and mid = (start +end) /2 = 15/2 = 7**

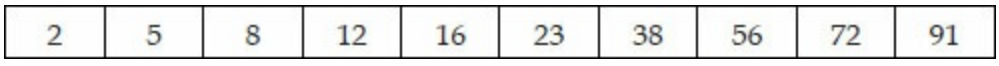
Step 3

Here, **A[mid] = 23 = Key**

Hence, Value is found in the array at position 7.

Example 2

Sorted array



Value to be searched is Key = 23

Step 1

Here, **A[mid] = 16 <23**

That is, **A[mid] < Key**

Hence, **start = mid+1 =5 and mid = (start +end) /2 = (9+5)/2 =7**

Step 2

Here, **A[mid] = 56>23**

That is, **A[mid] > Key**

Hence, **end = mid-1 =6 and mid = (start +end) /2 = (5+6)/2 =5**

Step 3

Here, **A[mid] = 23 = Key**

Hence, Value is found in the array at position 5.

Analysis of Algorithm

Worst-case time complexity = O (log n) comparisons: In the worst case, the value n does not exist in the array at all. Through each recursion or iteration of Binary Search, the size of the admissible range is halved. This halving can be done O (log n) times. Thus, O (log n) comparisons are required.

Best-case time complexity = O (1) comparisons: In the best case, the value n is the middle element in the given array. Hence, only 1 comparison is required.

Finding the minimum and maximum

The Max Min problem is to find the maximum and minimum value from the given array. There are two approaches for finding the minimum and maximum value in an array, and they are explained as follows.

Naive method

It is a basic method in which minimum and maximum value can be found by simply comparing each value of an array. Hence, this approach is simple but requires 2(n-1) comparisons, that is, (n-1) comparison to find maximum number and (n-1) comparison to find minimum number.

The number of comparisons can be reduced by using Divide and Conquer approach.

Divide and conquer approach

In this approach, there are the following three cases:

- If there is only one element in an array, then it is said to be both minimum and maximum.
- If the array contains only two elements, then maximum and minimum value can be decided using only one comparison.
- If the array contains more than two elements, then the array is divided into two halves from the middle. Then using a recursive approach, minimum and maximum are found in each half independently. Later, return the maximum of two maxima of each half and minimum of two minima of each half.

Algorithm

Let us now look at the algorithm:

```
//Using Divide and Conquer Approach.
MaxMin (a, low, high)
If (n==1) then
  return (a[0], a[1])
else if (n==2) then
  if (a[1] < a[2]) then
    return (a[1], a[2])
  else
    return (a[2], a[1])
else
  mid = (low + high)/2
  [Left_Min, Left_Max] = MaxMin (a, low, mid)
  [Right_Min, Right_Max] = MaxMin (a, mid+1, high)
If (Left_Max>Right_Max) then
  max = Left_Max
```

```
else
  max = Right_Max
end
If (Left_Min<Right_Min) then
  min = Left_Min
else
  min = Right_Min
end
return (min, max)
end
```

Analysis of Algorithm

From the preceding algorithm, **`MaxMin`** creates two problems of size n/2, and thus the recurrence relation is:

$T(n) = 2T(n/2) + 2 \quad , \quad n > 2$

$T(n) = 1 \quad , \quad n = 2$

$T(n) = 0 \quad , \quad n = 1$

When ‘n’ is a power of 2, n=2k for some positive integer ‘k’, then:

$T(n) = 2T(n/2) + 2$

$= ٢)٢T(n/٢ + (٢ + (٤$

$= ٤T(n/٢ + ٤ + (٤$

$= ٢k-١\ T(٢) + \Sigma\ ١ \le I \le k-٢ \ge ١i$

$= ٢k-٢ + ١k - ٢$

$T(n) = (3n/2) – 2$

Hence, (3n/2) - 2 is the best-case, average-case, and worst-case number of comparisons.

Merge Sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Algorithm

Let us now look at the algorithm:

```
MergeSort(A)
{
  n = length(A)
  if (n<2) then
    return mid = n/2
  left  = array of size(mid)
  right = array of size(n-mid)
  for i=0 to mid-1
    left[i] = A[i]
  for i = mid to n-1
    right[i-mid] = A[i]
  MergeSort(left)
  MergeSort(right)
  Merge(left,right,A)
}
Merge(L,R,A)
{
  nL = length(L)
  nR = length(R)
  i = j = k = 0
  while (i<nL && j<nR)
  {
    If (L[i] <= R[j]) then
    {
      A[k] = L[i]
      k = k +1
      i = i+1
    }
    else
    {
      A[k] = R[j]
      j = j+1
    }
    k = k+1
  }
  while (i<nL)
  {
    A[k] = L[i]
  i = i+1
    k = k+1
  }
  while (j < nR)
  {
    A[k] = R[j]
```

```
    j = j+1
    k = k+1
  }
}
```

Analysis of Merge Sort

In MergeSort(A), the first 5 lines are simple statements or simple operations, and so, it will take time, say 'C1'. The two for loops consist of executing simple statements. Hence, together it will take some constant time say 'C2.n'

If it takes T(n) time for an array of size n, then the left sub-array will take cost T(n/2) and the other half will take T(n/2).

All merge function does, is it picks one element from either left or right sub-array and then writes it into another array.

So, it would take cost 'n.C3 + C4'

Therefore,

$T(n) = 2\,T(n/2) + C2.n + n.C3 + C4$

$= 2\,T(n/2) + n\,(C2 + C3) + C4$

Let us consider,

$T(1) = C$ *(for n=1)*

Therefore,

$T(n) = 2T(n/2) + n.C$ *(Removing the constant values)*

Now using Substitution method

$T(n) = 2\{2T(n/4) + C.n/2\} + C.n$

$= 4T(n/4) + 2C.n$

$= 4\{2T(n/8) + C.n/4\} + 2C.n$

$= 8T(n/8) + 3C.n$

$= 16T(n/16) + 4C.n$ *(By Substitution)*

If we generalize, we can observe that,

$T(n) = 2^k T(n/2^k) + K.C.n$ ----------*(i)*

Since,

$T(1) = C$ *(as considered)*

Therefore,

$n/2^k = 1$

$n = 2^k$

Therefore, $k = \log_2 n$

Hence from (i)

$T(n) = 2^k T(n/2^k) + k.C.n$

$T(n) = 2^{\log_2 n}.T(n/2^{\log_2 n}) + \log_2 n.C.n$

$T(n) = n.T(1) + n.\log_2 n.C$ ----------(Since $2^{\log_2 n} = n$)

$T(n) = n.C + C.n.\log_2 n$

Here, by dropping the lower order terms, that is, n. C and by dropping the constant multiplier the **Time Complexity** is equal to O(n.logn) or we can also say, **Time Complexity** is equal to O(n.logn).

The best case as well as worst case Time complexity of **Merge Sort** is O(n.logn).

Quick Sort

Quick sort is also one of the divide and conquer algorithms. It is also called the Partition exchange algorithm. For sorting, it uses the special variable called Pivot. After sorting the elements, the left of the pivot is smaller than pivot and the elements to the right are greater than it. The pivot can be selected in any of the ways; first element as pivot and last element as pivot, middle element as pivot or any random element as pivot.

Algorithm

Let us now look at the algorithm:

```
Quicksort(A, start, end)
{
  If (start < end)
  {
    PIndex ← Partition (A, start, end)
    Quicksort(A, start, PIndex-1)
    Quicksort(A, PIndex+1, end )
  }
}
```

```
Partition(A, start, end)
{
 Pivot ← A[end]                  ……(1)
 PIndex  ← start                 ……(2)
 For i ← start to end -1
 {
  If ( A[i] <= pivot )
  {
   Swap ( A[i], A[index])         …….(3)
   PIndex ← PIndex +1              …….(4)
  }
 }
Swap (A. [PIndex], A[end] )
return PIndex
}
```

Analysis of algorithm

Let us consider the best case as well as the worst-case scenario.

Best case

As Quick Sort works on divide and conquer rule, for best case, the array will be already sorted. So, we go on dividing the array into a number of sub-arrays. Hence, there will be no need to perform any sorting operation, while conquering the sub-arrays.

In the partition (A, start, end) function, the statements which are numbered as (1), (2), (3) and (4), are simple statements which will take some constant time and together. Let us say they cost us “b”.

The **for** loop again consists of some simple statements which will cost, say “a “. Hence, the **for** loop will take together “an’’ time. The **for** loop will be executed one extra time but we neglect such a small cost while calculating the rate of growth for very high values of n.

Therefore, the time expression here will be:

$$T = an+b$$

Here, the complexity is computed using the Substitution method, but it can also be solved using Master’s method and Recursion tree method.

Now, the first statement is simple and will take the constant time C1.

The statement inside the condition will have time complexity as follows:

- The first statement will take **`an+b`** cost, as discussed earlier.
- The second and third statements will be the partitioning of an array. It consists of an element pivot. For us, it will be best if we consider a balanced partition, that is, Arrays to the left and right of pivot will have value n/2 each.
- Therefore, the time taken by quicksort is T(n/2) each.

$T(n) = an+b$

$T(n) = 2T(n/2) + an+b+c$

$T(n) = 2T(n/2) + Cn$

$T(1) = C1$

$T(n) = 2T(n/2) + Cn \quad \ldots\ldots(1)$

$= 2(2T(n/4) + Cn/2)+Cn \Rightarrow 4T(n/4) +2\,Cn$

$= 4(2T(n/8) + Cn/4)+2Cn \Rightarrow 8T(n/8) +3\,Cn$

$T(n) = 2^kT(n/2^k) + KCn$

$Let,\ n/2^k = 1$

Therefore, $n = 2^k$

$K = log_2n$

$T(n) = 2log_2n\ T(1) + log_2n$

$= nC1 + Cn\ log_2n$

$T(n) = O(nlogn)$

Hence, the time complexity is O(nlogn) for the best case.

Worst case

As we have already discussed about the best case, the worst case will be when we have totally unbalanced partitioning.

In this case, in **`quicksort(A,start,end)`**

- The if statement has complexity C1.
- The first statement in **if** condition has time complexity as **`an+b`**, same as in best case.
- The next statement will have time complexity **`T(n-1)`** and the next

statement will cost the same.

$T(n) = T(n-1)+Cn$

$= (T(n-2)+(Cn-1))+Cn \Rightarrow T(n-2)+2Cn-C$

$= (T(n-3)+(Cn-2))+2Cn-2 \Rightarrow T(n-3)+3Cn-3C$

$T(k)=T(n-k)+kCn-(k(k-1)/2)C$

$Let, n-k=1$

$n=k$

$T(n)=T(1)+Cn^2-(n(n-1)/2)C$

$T(n)=C1+Cn(n+1)/2$

$T(n)=Cn^2/2+Cn/2+C1$

$T(n)=O(n^2)$

Hence, the time complexity is $O(n^2)$ for the worst case.

Strassen's matrix multiplication

Strassen's Matrix is based on divide and conquer strategy, which includes a smaller number of multiplications, in comparison to the traditional way of matrix multiplication. The multiplication method is defined as:

$[Z_{11}\ Z_{12}\ Z_{21}\ Z_{22}] = [X_{11}\ X_{12}\ X_{21}\ X_{22}] \times [Y_{11}\ Y_{12}\ Y_{21}\ Y_{22}]$`

$Z_{11} = S_1 + S_4 - S_5 + S_7$

$Z_{12} = S_3 + S_5$

$Z_{21} = S_2 + S_4$

$Z_{22} = S_1 + S_3 - S_2 + S_6$

Where,

$S_1 = (X_{11} + X_{22}) * (Y_{11} + Y_{22})$

$S_2 = (X_{21} + X_{22}) * Y_{11}$

$S_3 = X_{11} * (Y_{12} - Y_{22})$

$S_4 = X_{22} * (Y_{21} - Y_{22})$

$S_5 = (X_{11} + X_{12}) * Y_{22}$

$S_6 = (X_{21} - X_{11}) * (Y_{11} + Y_{12})$

$S_7 = (X_{12} - X_{22}) * (Y_{21} + Y_{22})$

Example 3

Multiply the given two matrix using Strassen's Matrix Multiplication:

$$X = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 4 & 3 \\ 2 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} 0 & 0 & 4 & 3 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 2 \\ 3 & 4 & 0 & 0 \end{bmatrix}$$

Let $Z = X \times Y = [X_{11}\ X_{12}\ X_{21}\ X_{22}] \times [Y_{11}\ Y_{12}\ Y_{21}\ Y_{22}]$`

Z= [[0 0 3 4][1 2 0 0][0 0 2 1][4 3 0 0]] × [[0 0 2 1][4 3 0 0][0 0 3 4][1 2 0 0]]

$S_1 = (X_{11} + X_{22}) * (Y_{11} + Y_{22})$

= ([0 0 3 4] + [4 3 0 0]) * ([0 0 2 1] + [1 2 0 0])

= [4 3 3 4] * [1 2 2 1]

= [10 11 11 10]

$S_2 = (X_{21} + X_{22}) * Y_{11}$

= ([0 0 2 1] + [4 3 0 0]) * [0 0 2 1]

= [4 3 2 1] * [0 0 2 1]

= [6 3 2 1]

$S_3 = X_{11} * (Y_{12} - Y_{22})$

= [0 0 3 4] * ([4 3 0 0] – [1 2 0 0])

= [0 0 3 4] * [3 1 0 0]

= [0 0 9 3]

$S_4 = X_{22} * (Y_{21} - Y_{11})$

= [4 3 0 0]* ([0 0 3 4]-[0 0 2 1])

= [4 3 0 0] * [0 0 1 3]

= [3 9 0 0]

$S_5 = (X_{11} + X_{12}) * Y_{22}$

= ([0 0 3 4]+[1 2 0 0]) * [1 2 0 0]

= [1 2 3 4] * [1 2 0 0]

= [1 2 3 6]

$S_6 = (X_{21} - X_{11}) * (Y_{11} + Y_{12})$

= ([0 0 2 1] – [0 0 3 4]) * ([0 0 2 1]+[4 3 0 0])

= [0 0 -1 -3] * [4 3 2 1]

= [0 0 -10 -6]

$S_7 = (X_{12} - X_{22}) * (Y_{21} + Y_{22})$

= ([1 2 0 0] – [4 3 0 0]) * ([0 0 3 4] + [1 2 0 0])

= [-3 -1 0 0] * [1 2 3 4]

= [-6 -10 0 0]

$Z_{11} = S_1 + S_4 - S_5 + S_7$

= [10 11 11 10] + [3 9 0 0] – [1 2 3 6] + [-6 -10 0 0]

= [6 3 8 4]

$Z_{12} = S_3 + S_5$

= [0 0 9 3] + [1 2 3 6]

= [1 2 12 9]

$Z_{21} = S_2 + S_4$

= [6 3 2 1] + [3 9 0 0]

= [9 12 2 1]

$Z_{22} = S_1 + S_3 - S_2 + S_6$

= [10 11 11 10] + [0 0 9 3] – [6 3 2 1] + [0 0 -10 -6]

= [4 8 8 6]

Z = [[8 3 8 4][1 2 12 9][11 7 2 1][4 8 8 6]]

$$Z=\begin{bmatrix} 6 & 8 & 1 & 2 \\ 8 & 4 & 12 & 9 \\ 9 & 12 & 4 & 8 \\ 2 & 1 & 8 & 6 \end{bmatrix}$$

Analysis of algorithm

T(n) = 7T(n/2) + O(n^2) = O(n^log(7)) runtime.

Approximately O(n^2.8074) which is better than O(n^3)

Conclusion

Searching is the operation of finding the elements from the given set of data. Binary search is more efficient than linear search. In binary search, the elements in the array must be sorted, which is not in the case of linear search. Binary search comes under divide and conquer using search technique. In each step, the binary search algorithm divides the array into two parts and checks if the element to be searched, is on the upper or lower half of the array. Best case is O(1), and average case and worst case for binary search is $O(\log_2 n)$.

In the Max-min problem, for a given array, we must find the maximum and minimum elements. Before the divide and conquer approach, the maximum and minimum elements could be found by comparing every element in the list, but by using divide and conquer, the comparisons have reduced.

Merge sort sorts the list, using the divide and conquer technique. Merge sort uses a two-way merge process. The best case and worst case for this sorting technique is $O(n\log_2 n)$.

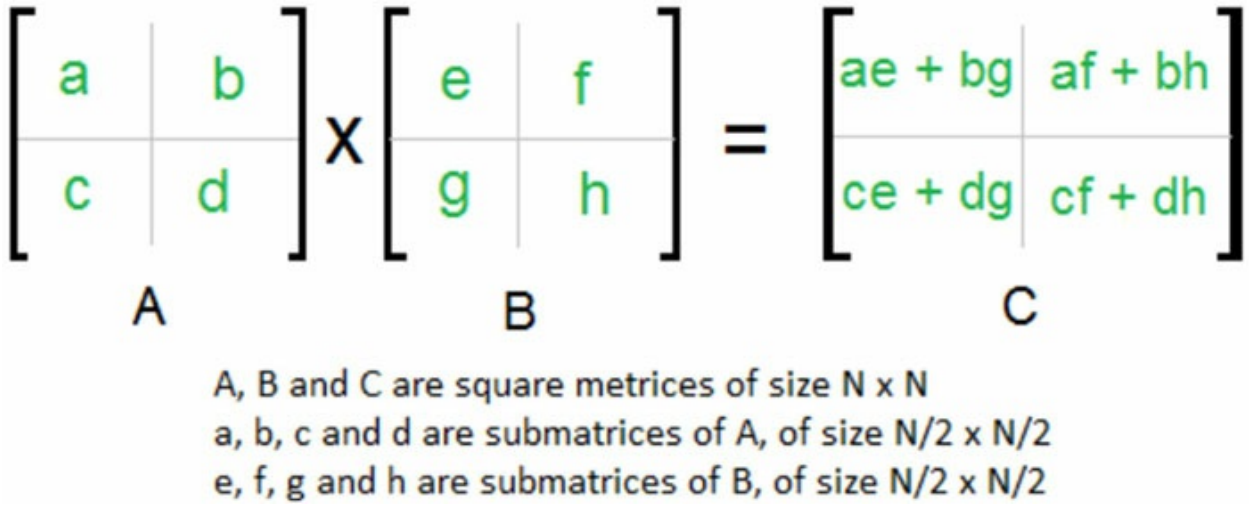
Quick sort selects one pivot element and then moves it to the correct position by fixing the pivot, then dividing the list into equal or unequal size. Best case is $O(n\log_2 n)$, and worst case is $O(n^2)$.

Key facts

- **Divide and Conquer:** It repeatedly divides a problem into two or more fragments of the same or closely similar type, until these are sufficiently straightforward to be solved by themselves. The answer to the main

problem is then produced by combining the answers to the fragments.

- **Binary Search:** A target value's location within a sorted array is discovered using this search process. The middle element of the array is used as a comparison point for the target value in binary search. If the target value and the middle element are not equal, the half where the target cannot lie is omitted, and the search is then repeated on the other half, using the middle element to compare to the target value once again, until the target value is identified. The target is not in the array if the remaining half is empty when the search is finished.
- **Finding the minimum and maximum:** Suppose we have an array of size N. Using the minimum number of comparisons, the aim is to determine the array's maximum and minimum elements by Divide and Conquer Technique.
- **Merge-Sort:** The Divide and Conquer paradigm serves as the foundation for the sorting algorithm known as the Merge Sort. The array is divided into two equal halves at first, and then they are merged in a sorted fashion.
- **Quick Sort:** QuickSort is a Divide and Conquer algorithm, like Merge Sort. It divides the given array around a particular element that it selects as a pivot. A partition() is QuickSort's most important process. Given an array and the pivot element, x, partitions aim to place x in the correct order in a sorted array, with all smaller elements preceding x and all greater elements following x.
- **Strassen's Matrix Multiplication:** It is a simple Divide and Conquer method to multiply two square matrices. Divide matrices A and B in 4 sub-matrices of size N/2 x N/2. Calculate the multiplication values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Questions

1. Explain with example how divide and conquer strategy is used in Binary Search? **[MU DEC'18 (5 MARKS)]**
2. Write an algorithm for finding minimum and maximum numbers from a given set. **[MU DEC'18 (5 MARKS)] [MU DEC'19 (5 MARKS)]**
3. Sort the following numbers using Quick Sort. Also derive time complexity of quick sort: 27,10,36,18,25,45 **[MU DEC'18 (10 MARKS)]**
4. Explain divide and conquer approach. Write a recursive algorithm to determine the max and min from the given elements. **[MU MAY'19 (10 MARKS)]**
5. Write an algorithm for implementing Quick Sort. Also comment on its complexity. **[MU DEC'19 (10 MARKS)]**
6. Write short note on Merge Sort **[MU DEC'19 (5 MARKS)]**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

Greedy Algorithms

Introduction

A greedy algorithm is a method of problem-solving that chooses the best choice available at the moment. It is unconcerned with whether the most excellent result at the moment will produce the final best result.

The knapsack problem is stated as, given a set of items, with each having a weight and a value, choose the number of each item to include in collection, such that the total weight is less than or equal to a given limit and the total value is as high as it can be.

The main goal of Job sequencing problems is finding a sequence of jobs that is finished within deadlines and yields the most profit.

A subset of edges in a connected weighted undirected graph that joins all the vertices with the least amount of edge weight is known as a **Minimum Spanning Tree** (**MST**). Prim's algorithm or Kruskal's algorithm can be used to derive an MST.

Optimal Storage on tapes is one of the applications in the Greedy method. The objective of this algorithm is to find the Optimal retrieval time for accessing programs that are stored on tape.

Create a single sorted file by combining several sorted files of varying lengths. To build the final file in the shortest amount of time, we must identify the optimal merge pattern. Given a collection of elements, the subset covering problem aims to find the minimum number of sets that incorporate (cover) all these elements.

The goal of an algorithm is to find the optimum solution to a certain problem. When using a greedy method, choices are drawn from the available solution domain. Being greedy, the closest option that appears to offer the optimum solution is picked.

Structure

In this chapter, we will discuss the following topics:

- Knapsack problem
- Job Sequencing with deadlines
- Minimum Cost Spanning Tree
- Optimal Storage on Tapes
- Optimal Merge Pattern
- Subset cover Problem
- Container Loading problem

Objectives

By the end of this chapter, the reader will learn about various greedy algorithms such as the knapsack problem, optimal merge pattern, subset cover problem and so on, in detail with various examples.

Knapsack problem

Given a set of items, each is associated with a particular weight with it. The knapsack problem finds the set of items whose total weight is less than or equal to the weight limit and the profit earned should be maximum.

There are mainly two types of Knapsack problem:

- **0/1 (Binary) Knapsack:** Items cannot be divided further.
- **Fractional Knapsack:** Items can be divided further.

In this chapter, fractional Knapsack problem is discussed. As the name implies, there is division among the items in the fractional knapsack problem. If it is not possible to take the entire object, we can even put a portion of it in the knapsack. Among the given set of items, the item that needs to be selected first is the item with the Maximum ratio.

$$Ratio = \frac{Profit}{Weight}$$

In the Knapsack problem, ‘W’ represents the weight of the knapsack, ‘P’ represents the Profit, ‘n’ represents the number of items. The fractional

knapsack problem is solved as follows with different examples.

Example 1

Consider the total number of item as n=3, and weight limit of the Knapsack=20. The profit of each item is (v_1,v_2,v_3) = (25,24,15), and the weight of each item is (w_1,w_2,w_3)=(18,15,10).

Solution

Ratio is calculated as follows:

$$\textit{The ratio of Item-1 as, } X1 = \frac{25}{18} = 1.38$$

$$\textit{The ratio of Item-2, } X2 = \frac{24}{15} = 1.6$$

$$\textit{The ratio of Item-3, } X3 = \frac{15}{10} = 1.5$$

Optimal Solution:

$$\textit{Selection of Item } X1 = 0 = 0$$

$$\textit{Selection of Item } X2 = 1 = 24$$

$$\textit{Selection of Item } X3 = \frac{1}{2} = 7.5$$

The total profit in this case = 0 + 24 + 7.5 = **31.5**

Example 2

Knapsack Capacity = 30. Refer to *Table 4.1* to solve the Fractional Knapsack problem:

Item	A	B	C	D
Value	50	140	60	60
Size	5	20	10	12
Ratio	10	7	6	5

Table 4.1: *Knapsack Example*

Solution:

Optimal Solution:

Selection of Item, A = 1 = 50

Selection of Item B = 1 = 140

Selection of Item C = $\frac{1}{2}$ = 30

Selection of Item D = 0 = 0

The total profit in this case =50 + 140 + 30 + 0 = 220

Example 3

Let us consider the capacity of Knapsack as W=60. Refer to *Table 4.2* to solve the Fractional Knapsack problem:

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio	7	10	6	5

Table 4.2: *Knapsack Example*

Solution:

Optimal Solution:

Selection of Item A = 1 = 40

Selection of Item B = 1 = 10

Selection of Item C = 1/2 = 10

Selection of Item D = 0 = 0

The total profit in this case = 280 + 100 + 60 + 0 = **440**

Job Sequencing with deadlines

The primary purpose of job sequencing problems is to identify a sequence of jobs that is completed on time and generates the greatest profit.

According to the Job Sequencing Problem:

- On a machine, there are ‘n’ jobs that need to be processed.
- Each job ‘i’ has a profit pi ≥ 0 and a deadline di ≥ 0.

- Profit is only earned if the job is finished by the deadline.
- If the job is processed for a unit of time on a machine, it is considered as finished.
- There is only one machine accessible to process jobs.
- On the machine, only one work is being processed at once.
- A subset of jobs J that allows each task to be finished by its deadline, constitutes a feasible solution.
- A workable solution with maximum profit value is an optimal solution.

Algorithm

Let us now look at the algorithm:

```
AlgorithmJS(D,J,n)
{
D[0] := J[0] :=0;
for i := 1 to n
{
T[i] = -1; //create an empty time slot array
}
J[1] := 1;
for i := 2 to n
{
k := min(Dmax,D[i]); // select between maximum deadline and
deadline of ith job
if (k>-1) then
{
if (T[k] == -1) then //check if time slot is empty
{
T[k] := J[k]; //assign job to that time slot
}
Else
k--;
}
}
```

Example 4

Refer to *Table 4.3* for this job sequencing problem:

Job	1	2	3	4	5
Profit	20	15	10	5	1

Deadline	2	2	1	3	3

Table 4.3: *Job sequencing with deadlines question*

The jobs are arranged in descending order as per the profit assigned with the jobs. As per the previous example job, 1 is yielding more profit, with the maximum deadline of 2. Thus, this job is considered first and assigned slot of (1, 2). Next, job 2 is considered with the profit of 15 which is the next highest profit. The deadline assigned is 2, as the slot (1, 2) is assigned to job 1, and slot (0, 1) is assigned to job 2. Now the next job which gives the maximum profit is job 3 with the deadline of 1. As the slot (0, 1) is already assigned to job 2, job3 gets rejected and this continues. Thus, the solution is the sequence of jobs (1, 2, 4), which are being executed within their deadline and gives the maximum profit.

Refer to *Table 4.4* for the solution:

Job	Assigned Slot	Job Considered	Action	Profit
{-}	-	1	Assign[1,2]	-
{1}	[1,2]	2	Assign[0,1]	20
{1,2}	[0,1][1,2]	3	reject	20+15=35
{1,2}	[0,1][1,2]	4	Assign [2,3]	35
{1,2,4}	[0,1][1,2][2,3]	5	reject	35+5=40
{1,2,4}	[0,1][1,2][2,3]	-	-	40

Table 4.4: *Job sequencing with deadlines answer*

Total profit of this sequence is **40.**

Minimum Cost Spanning Tree

A subset of the edges of a connected, undirected graph that joins all the vertices together, without any cycles, and with the least amount of total edge weight, is known as a minimal cost spanning tree. The Minimum Cost Spanning Tree is a spanning tree for a connected, undirected graph that has the lowest cost. *Figure 4.1* features a complete graph:

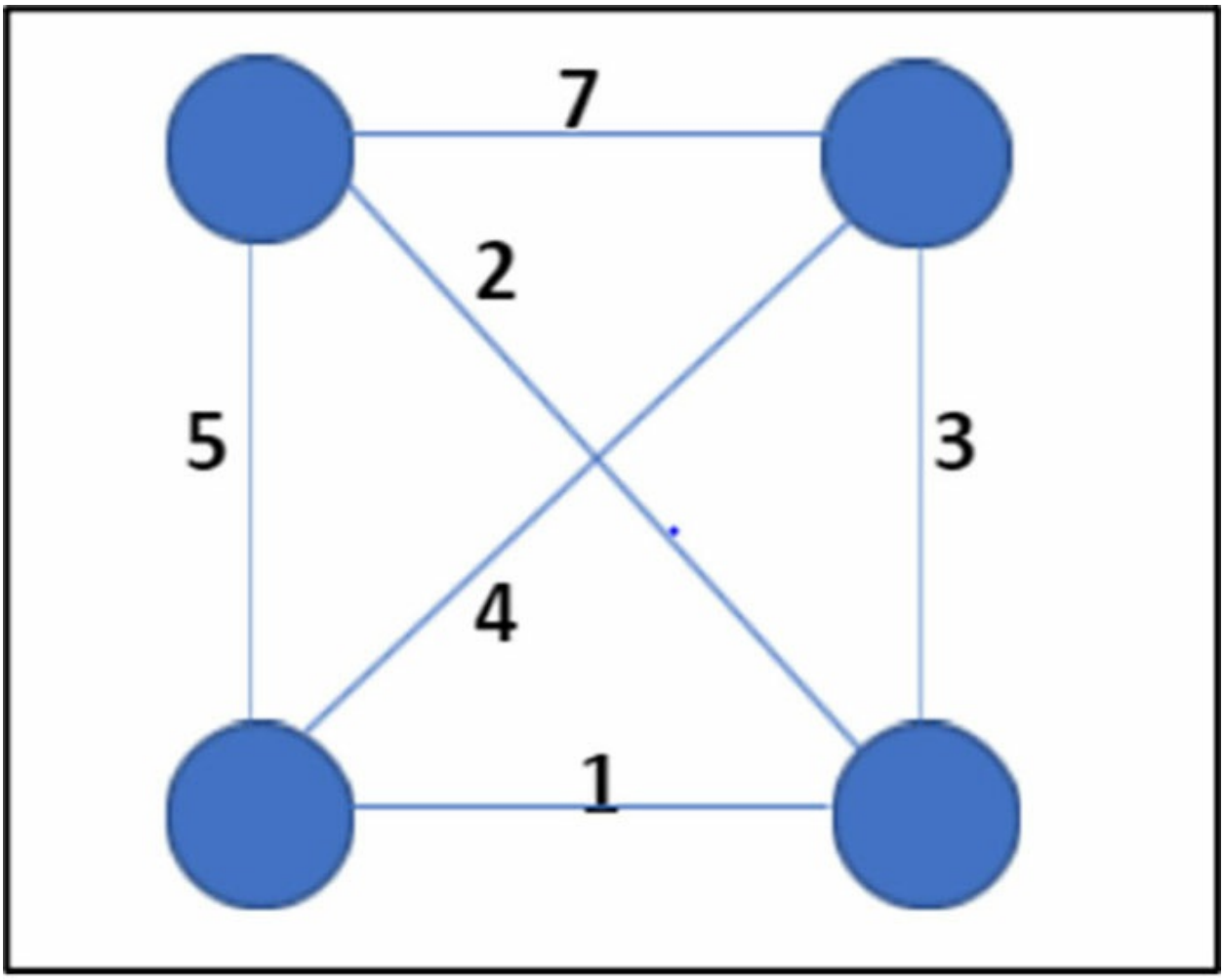


Figure 4.1: *Complete Graph*

Figure 4.2 Features a Minimum Spanning Tree:

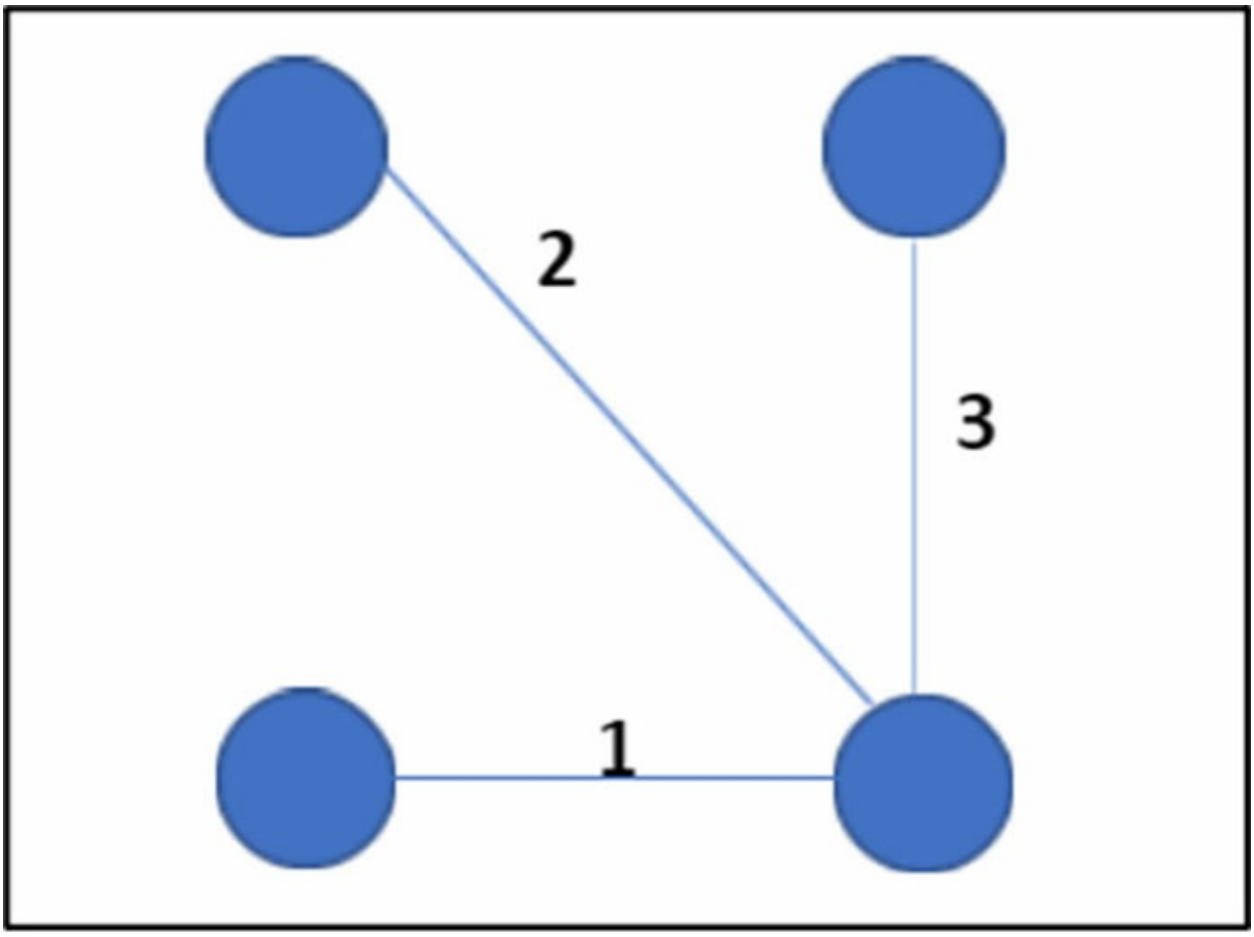


Figure 4.2: *Minimum Spanning Tree*

There are two basic algorithms for finding minimum cost spanning trees.

Kruskal's algorithm

The result of this algorithm is a forest of trees. The forest is initially made up of *n* single node trees (and no edges). We link two trees together without cycles, by adding one edge (the cheapest one) at each step.

Algorithm

Step 1: Order all the edges according to non-decreasing weight.

Step 2: Select the smallest edge. Test to see if a cycle is built with the currently formed spanning tree. Include this edge if a cycle does not develop. Throw it away if not.

Step 3: Carry out ***Step 2*** again and again until the spanning tree has (n-1)

edges.

Example 5

Consider the graph shown in the following *Figure 4.3* to construct the Minimum Spanning Tree with Minimum cost.

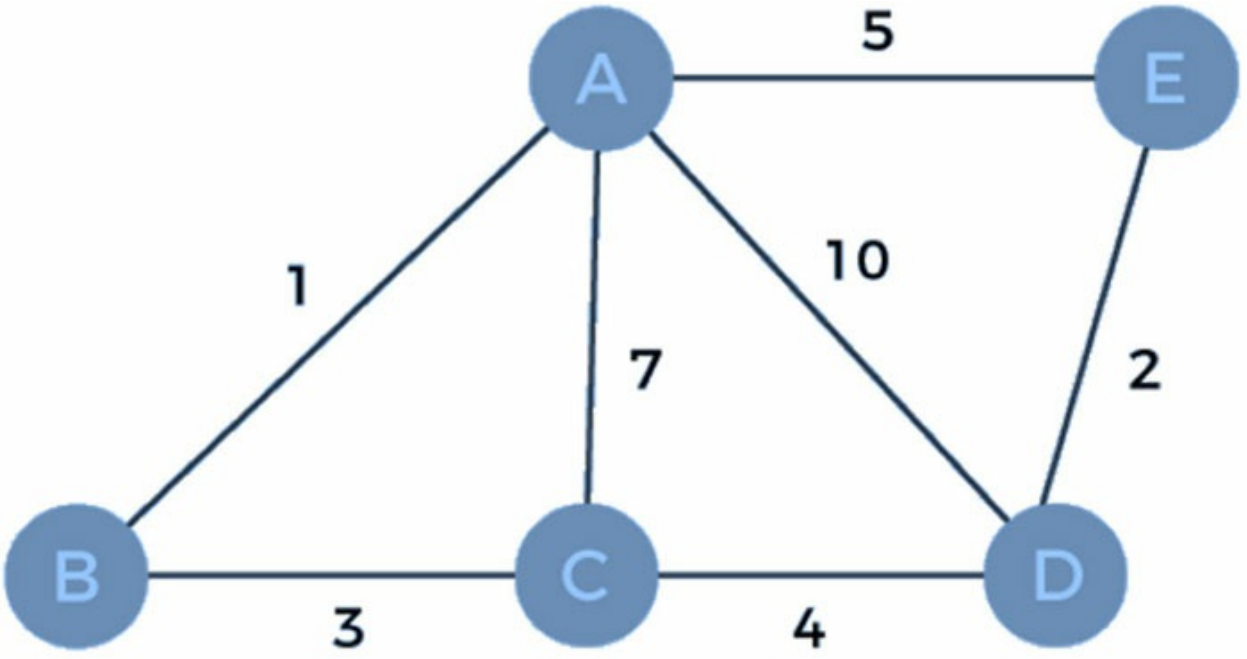


Figure 4.3: *A weighted graph*

For the given graph in *Figure 4.3*, the weight of the edges is tabulated in the following *Table 4.5*:

Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

Table 4.5: *Weight of the edges*

Based on the weights assigned for the edges, arrange the edges in ascending order, as shown in the following *Table 4.6*:

Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

Table 4.6: *Weight of the edges in ascending order*

Let us now see the Minimum Spanning Tree construction using Kruskal's Algorithm.

Step 1: The edge with the Minimum cost is **AB**, with the weight of value 1, is to added 1st in the Minimum Spanning Tree. Refer to the following *Figure 4.4*:

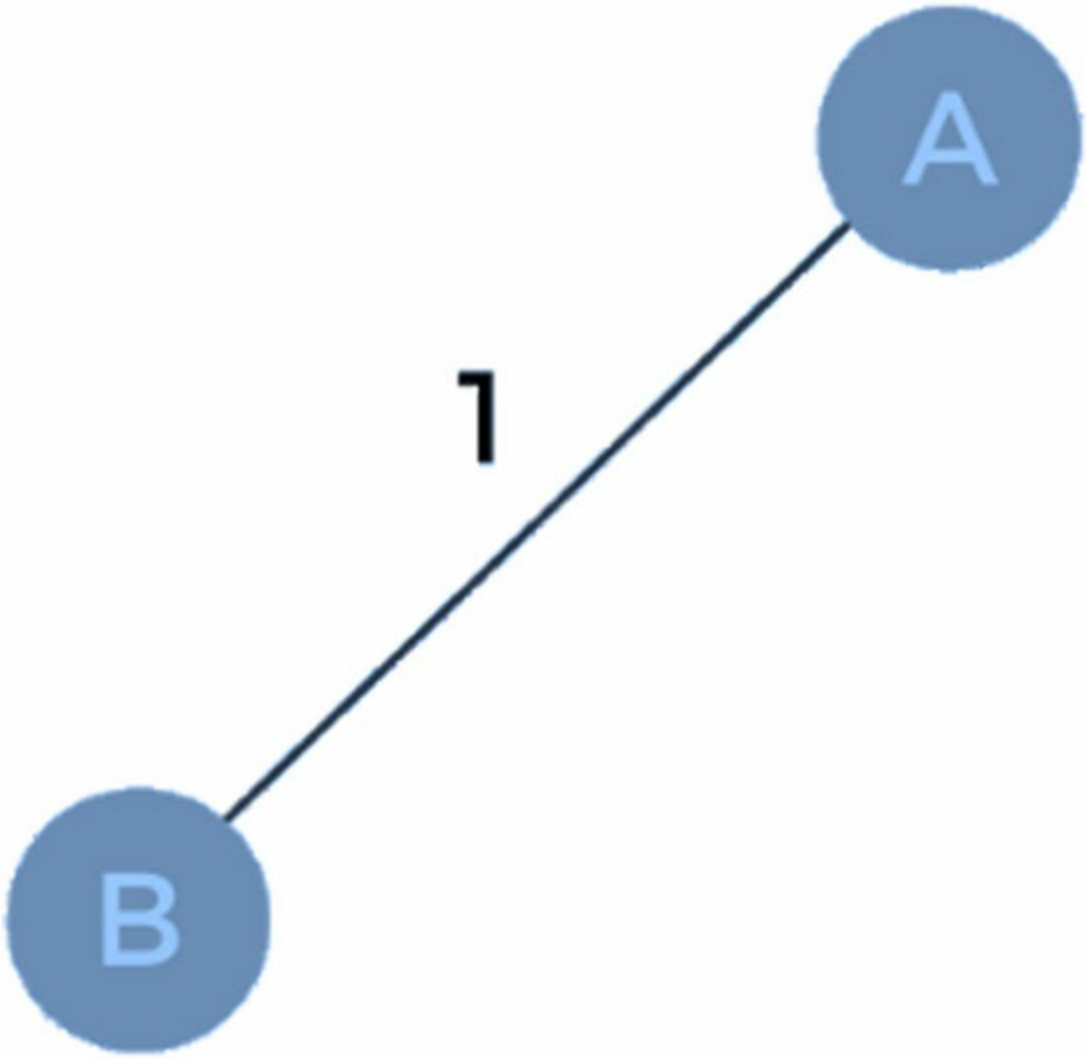


Figure 4.4: *Edge AB added to the Minimum Spanning Tree*

Step 2: The next minimum weighted edge **DE**, can be added to the Minimum Spanning Tree, and after adding, it must be checked whether it follows the condition of Minimum Spanning Tree or not. In this case, as it is not forming any cycle, it is added as the 2nd edge to the Minimum Spanning Tree. Refer to the following *Figure 4.5*:

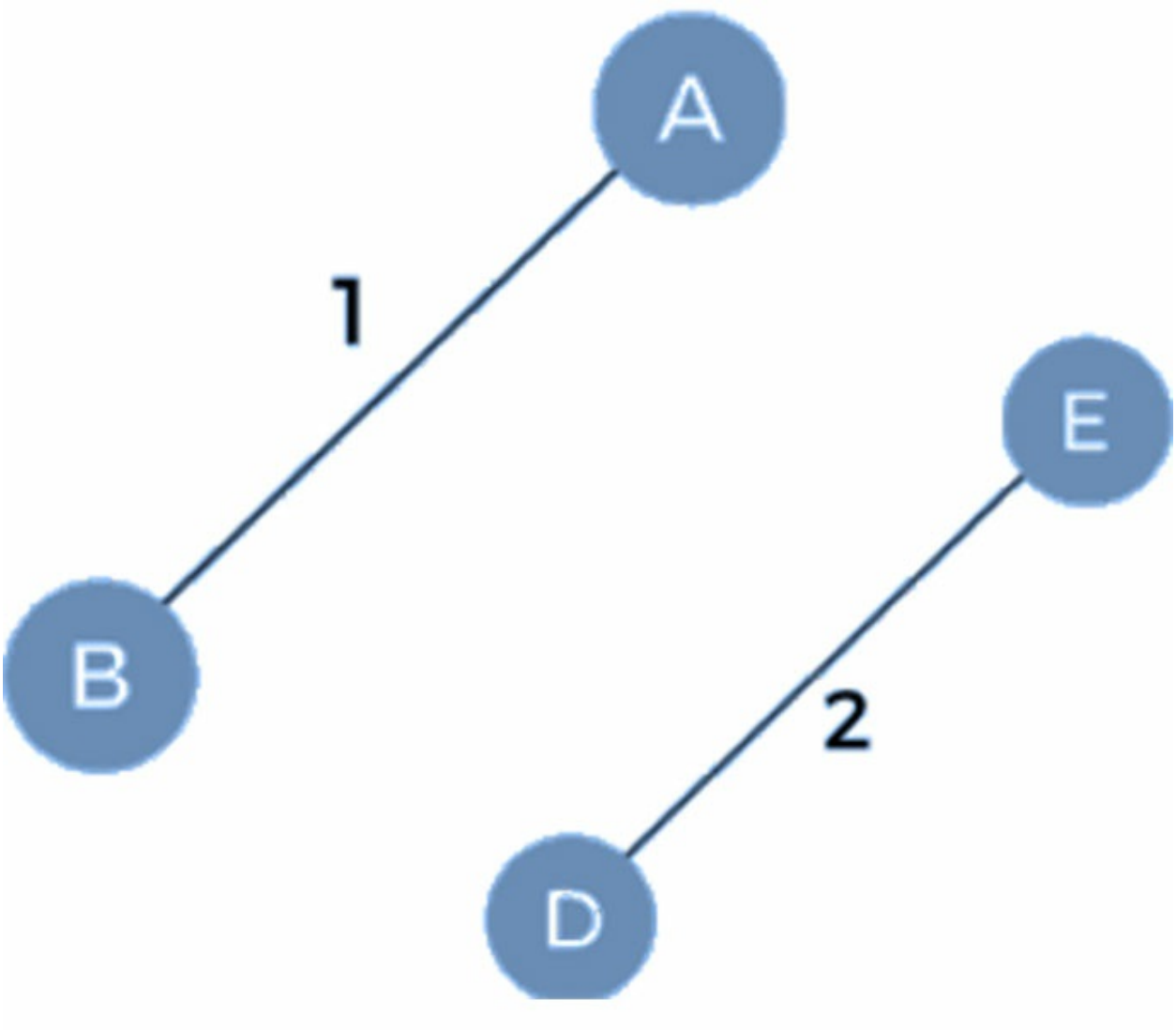


Figure 4.5: *Edge DE added to the Minimum Spanning Tree*

Step 3: Add the next minimum edge which is **BC**. It can be added to the Minimum Spanning Tree as it is not forming any cycle. Refer to the following *Figure 4.6:*

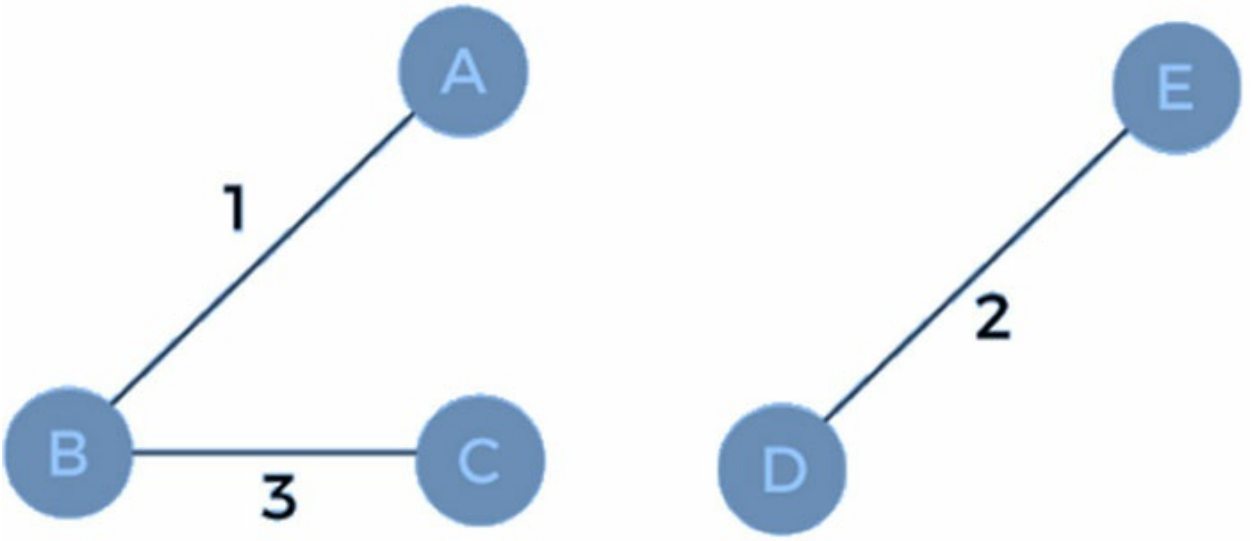


Figure 4.6: *Edge BC added to the Minimum Spanning Tree*

Step 4: The next edge with the Minimum cost is **CD**, and we will add it in the Minimum spanning Tree as it is also not forming any cycle. Refer to the following *Figure 4.7:*

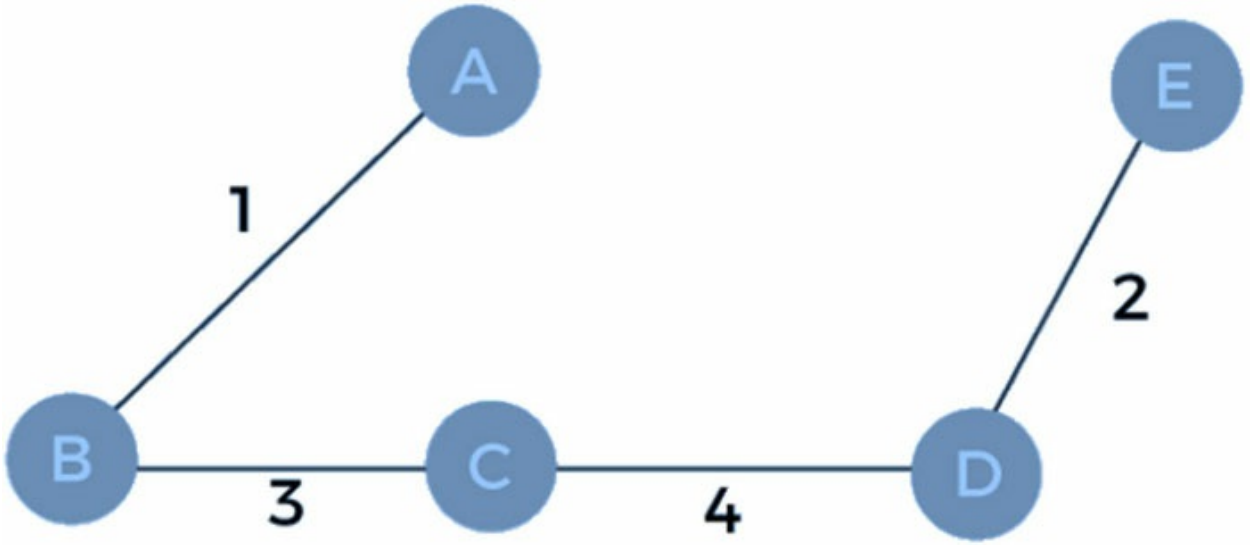


Figure 4.7: *Edge CD added to the Minimum Spanning Tree*

Step 5: The next edge that needs to be added to the Minimum Spanning Tree is AE, but this edge is to be discarded as it forms the cycle. It is the similar case with the edges AC and AD. The final Minimum Spanning Tree using Kruskal's algorithm is shown in Figure 4.8:

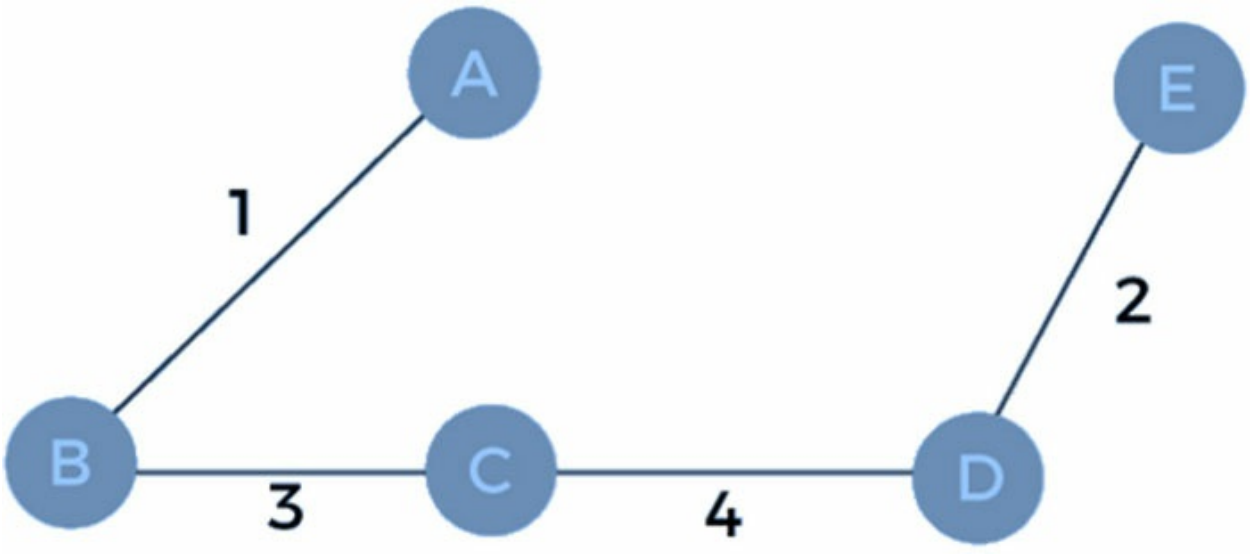


Figure 4.8: *Final Minimum Spanning Tree*

The Minimum Spanning cost is calculated by adding the weights of all the edges in the Minimum spanning Tree.

Minimum Spanning Cost = AB + DE + BC + CD = 1 + 2 + 3 + 4 = 10.

Prim's algorithm

Prim's algorithm starts with one node. Then, one by one, it adds disconnected nodes to the new graph, choosing from among the available nodes, the node whose connecting edge has the smallest weight each time.

Algorithm

Step 1: First, pick a starting vertex.

Step 2: Until there are fringe vertices, repeat ***Step 3*** and ***Step 4***.

Step 3: Choose a minimum-weight edge that connects the tree vertex with the fringe vertex.

Step 4: Include the chosen vertex and edge in the shortest spanning tree T [END OF LOOP]

Step 5: EXIT

Example 6

Consider the graph shown in the following *Figure 4.9* to construct the

Minimum Spanning Tree with Minimum cost:

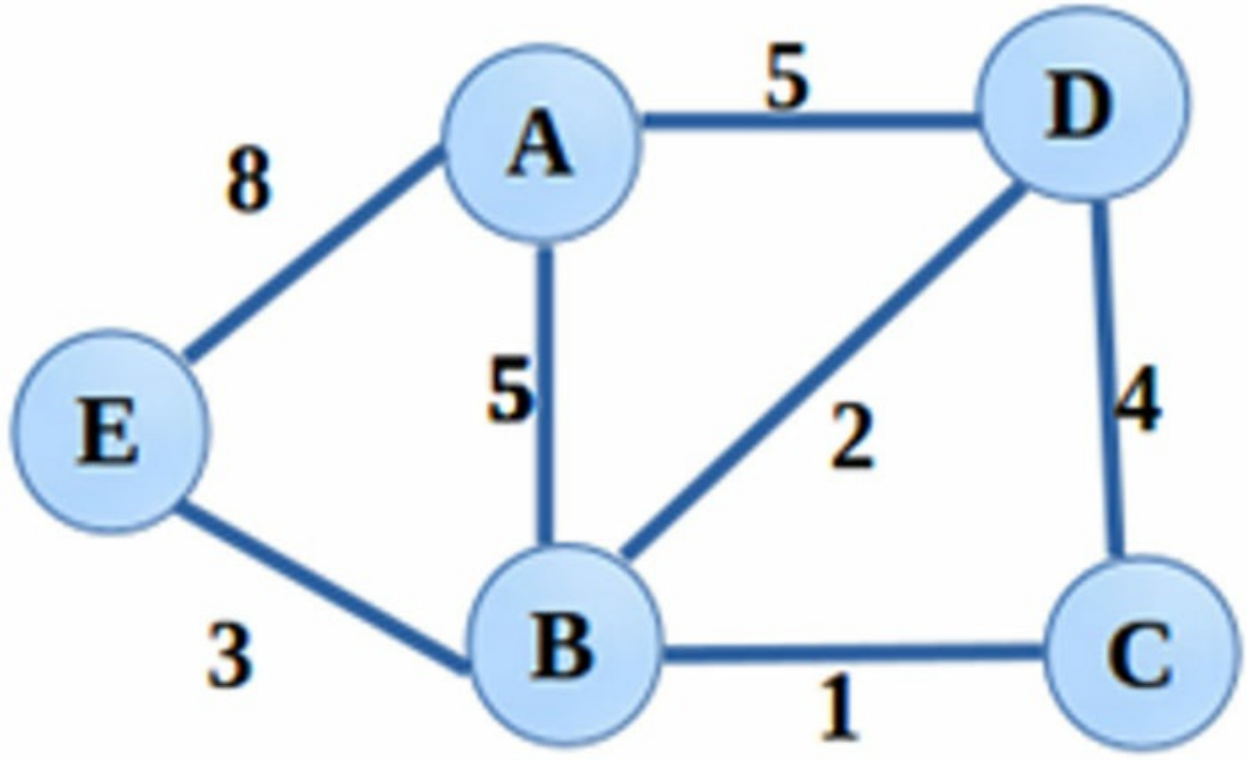


Figure 4.9: *A weighted graph*

Let us now see the Minimum Spanning Tree construction using Prim's Algorithm.

Step1: Select any one vertex as a starting vertex. In this example shown in *Figure 4.10*, Vertex **E** is considered:

Figure 4.10: *Vertex E added to the Minimum Spanning Tree*

Step 2: With the selected vertex, now compare all the adjacent edges. The edge with the Minimum weight is to be added to the Minimum Spanning Tree. In this case, E has 2 adjacent edges EA and EB, where EA has weight

of 8 and EB has weight of 3. While comparing the weights of EA and EB, EB is having the Minimum weight of 3, and so, EB will be added to the Minimum Spanning Tree. Refer to the following *Figure 4.11*:

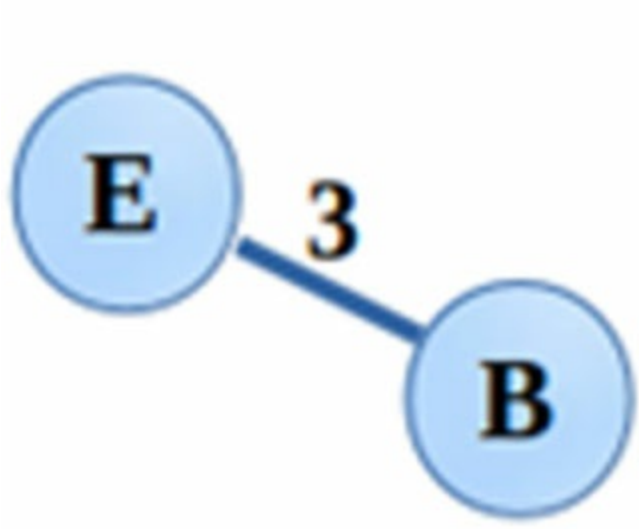


Figure 4.11: *The Edge EB is now the part of the Minimum Spanning Tree*

Step 3: Now, for both the vertices **E** and **B**, find the adjacent edges with the minimum value and add it in the Minimum Spanning Tree. In this case, **BC** has the Minimum weight of 1 and it satisfies the condition of Minimum Spanning Tree. Refer to the following *Figure 4.12*:

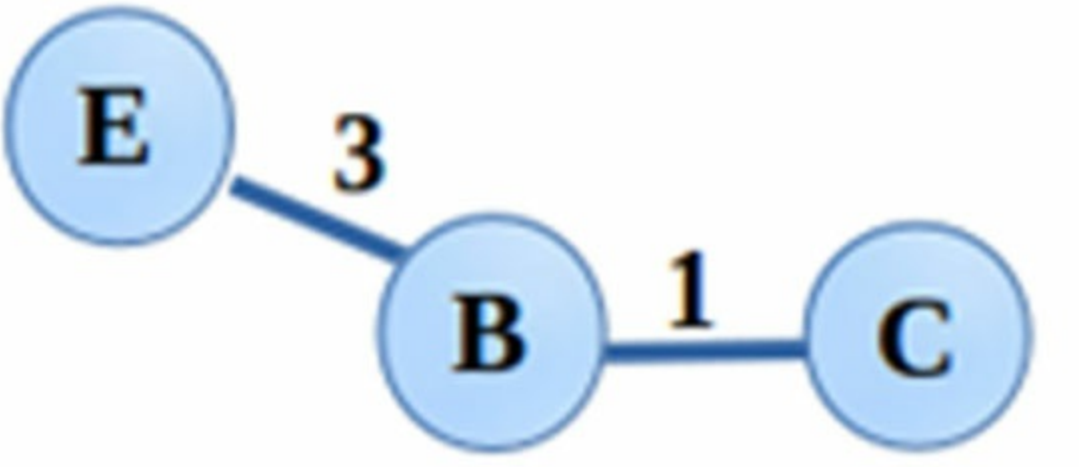


Figure 4.12: *The Edge BC added in the Minimum Spanning Tree*

Step 4: In this step, the edges **E**, **B** and **C** are considered and found to be the

Minimum adjacent edge. The edge with the Minimum weight is **BD**, which does not form any cycle. So add the edge **BD** to the Minimum Spanning Tree. Refer to the following *Figure 4.13:*

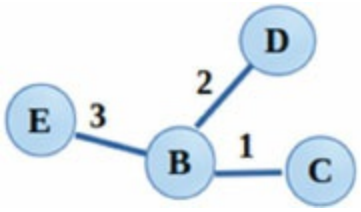


Figure 4.13: *The Edge BD added in the Minimum Spanning Tree*

Step 5: In this step, the edges **E**, **B, C** and **D** are considered and found to be the Minimum adjacent edge. The edge with the Minimum weight is **DC,** but it forms a cycle and is thus discarded. The next edge with the Minimum cost is **AB** and **AD**. **AD** forms a cycle if added, and so, it is discarded. Finally, the edge **AB** is added to the Minimum Spanning Tree. After adding **AB,** all the vertices are included in the Minimum Spanning Tree. Moreover, there is no cycle. Refer to the following *Figure 4.14:*

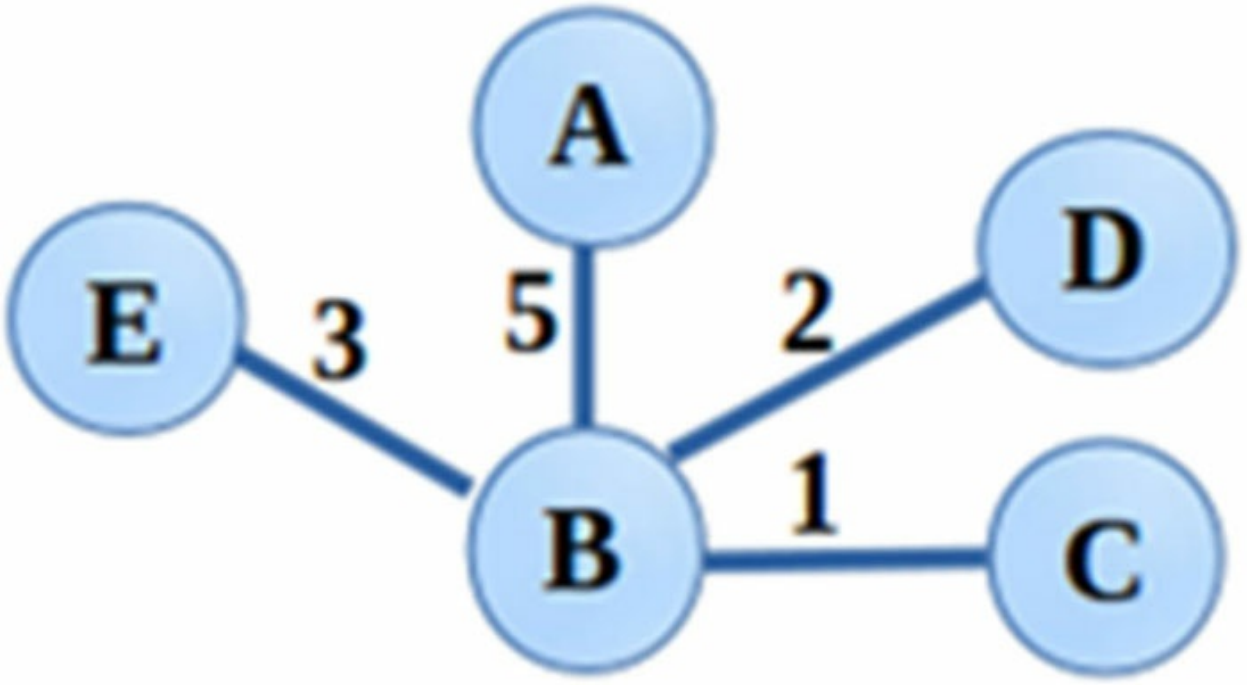


Figure 4.14: *Final Minimum Spanning Tree*

Since all vertices are included in the tree, it is our final Minimum Spanning Tree.

The Minimum Spanning Cost = BC + BD + EB + AB = 1 + 2 + 3 + 5 = 11

Optimal Storage on Tapes

Consider, given with "n" programs $P_1, P_2, P_3, \ldots, P_n$ of length $L_1, L_2, L_3 \ldots\ldots, L_n$ respectively, and store them on tapes of length L such that **Mean Retrieval Time** (**MRT**) is minimized. Retrieval of the j^{th} program is the summation of the length of the first *j* programs on tape, such that $1 \leq i \leq n$ and $\Sigma\ 1 \leq k \leq j\ Li \leq 1$.

Let us consider the magnetic tapes that provide the linear or the sequential way of playing music. In an audio cassette, if there are four songs, then the fourth song cannot be played directly. The first three songs are completed and then it goes to the fourth song. Suppose the lengths of the songs are 6,4,5, and 3 minutes respectively. To play the fourth song of 6+4+5=15 minutes, position the tape ahead. That is, the retrieval time taken to retrieve the fourth song entirely is 15+3 =18 mins.

Now for the sequential access of data, there are some limitations such as, for getting MRT, the sequences must have some order.

Optimal storage on single tapes

To reduce the MRT after storing all the programs on a single tape, we must find the permutation of the program order.

Example 7

Suppose there are 3 programs of length 2, 5, 4 respectively. Find the Optimal result.

Refer to *Table 4.7*:

Order of the program	Total Retrieval Time	Mean Retrieval Time
123	2+(2+5)+(2+5+4) = 20	20/3
132	2+(2+4)+(2+4+5) = 19	19/3
213	5+(5+2)+(5+2+4) = 23	23/3
231	5+(5+4)+(5+4+2) = 25	25/3

312	4+(4+2)+(4+2+5) = 21	21/3
321	4+(4+5)+(4+5+2) = 24	24/3

Table 4.7: *Optimal Storage on Tapes example*

From the preceding table, we get the MRT 19/3, which is achieved by storing the programs in the ascending order of their length.

The preceding program can be computed as:

```
Tj=0
for i=1 to n do
  for j=1 to i do
  Tj=Tj +L[j]
  End
End
MRT=sum(T)/n
```

Example 8

Explain optimal storage on tapes and find the optimal order for n=3 of length 5, 10, 3.

Refer to *Table 4.8*:

Order of the program	Total Retrieval Time	Mean Retrieval Time
123	5+(5+10)+(5+10+3)=38	38/3
132	5+(5+3)(5+3+10)=31	31/3
213	10+(10+5)+(10+5+3)=43	43/3
231	10+(10+3)+(10+3+5)=41	41/3
312	3+(3+5)+(3+5+10)=29	29/3
321	3+(3+10)+(3+10+5)=34	34/3

Table 4.8: *Optimal Storage on Tapes example*

From the preceding solution, we get the MRT 29/3, which is achieved by storing the programs in the ascending order of their length.

Optimal Storage on Multiple Tapes

The problem of minimizing MRT when retrieving the program from multiple tapes is the same as the concept of optimal storage on single tapes.

The programs are stored one at a time on each tape after being sorted, by increasing program length. In the following example, working is explained.

Example 9

There is a program of length L={12,34,56,73,24,11,34,56,79,92,34,92,46}. Store the programs on three tapes and minimize MRT.

Solution:

Here, the Given Data can be seen in *Table 4.9*:

P_i	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}
L_i	12	34	56	73	24	11	34	56	79	92	34	92	46

Table 4.9: *Optimal Storage on Multiple Tapes example*

The first step is to sort the data in ascending order of their size. Therefore, the sorted data is as shown in *Table 4.10*:

P_i	P_6	P_1	P_5	P_2	P_7	P_{11}	P_{13}	P_3	P_8	P_4	P_9	P_{10}	P_{12}
L_i	11	12	24	34	34	34	46	56	56	73	79	92	92

Table 4.10: *Optimal Storage on Multiple Tapes example*

Now distribute the files among all three tapes, as shown in *Table 4.11*:

Tape 1	P_6	P_2	P_{13}	P_4	P_{12}
Tape 2	P_1	P_7	P_3	P_9	
Tape 3	P_5	P_{11}	P_8	P_{10}	

Table 4.11: *Optimal Storage on Multiple Tapes example*

MRT_{Tape1} = ((11)+(11+34)+(11+34+46)+(11+34+46+73)+(11+34+46+73+92)) / 5

= 567 / 5 = 113.4

MRT_{Tape2} = ((12)+(12+34)+(12+34+56)+(12+34+56+79)) / 4

= 341 / 4 = 85.25

MRT_{Tape3} = ((24)+(24+34)+(24+34+56)+(24+34+56+92)) / 4

= 402 / 4 = 100.5

MRT = (MRT_{Tape1}_+ MRT_{Tape2} +MRT_{Tape3}) / 3 = (113.4 + 85.25 + 100.5) /3

= 299.15 / 3 = 95.28

Optimal Merge Pattern

Optimal merge pattern is a pattern that involves merging two or more sorted files in a single sorted file. There are many ways to do the merging, such as the two-way merging method. Different pairing requires different amounts of computing time.

Example 10

Consider the given files f_1, f_2, f_3, f_4, f_5 with 25,40,13,5, and 40 numbers of elements.

Solution:

If merge operations are performed according to the sequence, then,

M1 = merging f1 and f2 => 25+40 =>65

M2 = merging M1 and f3 => 65+13 =>78

M3 = merging M2 and f4 => 78+5 =>83

M4 = merging M3 and f5 => 83+40 =>123

Hence the total number of operations: 65+78+83+123 =349

Another solution is sorting the numbers in the ascending order, such as 5, 13, 25, 40, 40 until f_1, f_2, f_3, f_4, f_5.

M1 = merging f4 and f3 =>5+13 =>18

M2 = merging M1 and f1 =>18+25=>43

M3 = merging M2 and f2 => 43+ 40=>83

M4 = merging M3 and f5 => 83+40=>123

Hence the total number of operations 18+43+83+123 =267

Binary merge trees can be used to depict two-way merge patterns. Let us

consider a set of **n** sorted files **{f_1, f_2, f_3, …, f_n}**. This is initially thought of as a binary tree with a single node for each element. The next algorithm is used to locate this optimal solution.

Algorithm Tree (n)

Let us now look at the algorithm:

```
for i := 1 to n – 1 do
    declare new node
node.leftchild := least (list)
node.rightchild := least (list)
node.weight := ((node.leftchild).weight) +
((node.rightchild).weight)
    insert (list, node);
return least (list);
```

Example 11

Using the algorithm, the problem can be solved as shown in the following steps:

Initial Set:

Refer to *Figure 4.15*: *the sequence is sorted*

Figure 4.15: *Optimal Merge – sequence sorted*

Step1:

Refer to *Figure 4.16*:

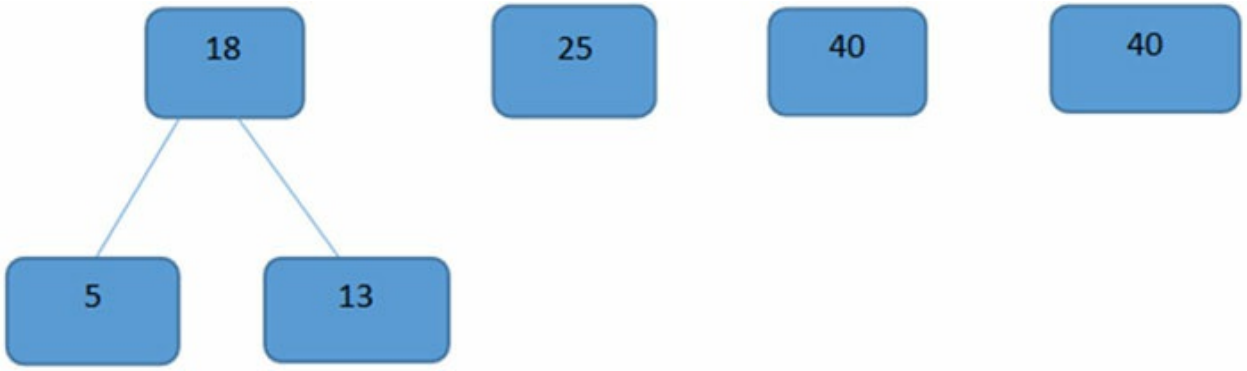


Figure 4.16: *5 and 13 are Merged*

Step2:

Refer to *Figure 4.17*:

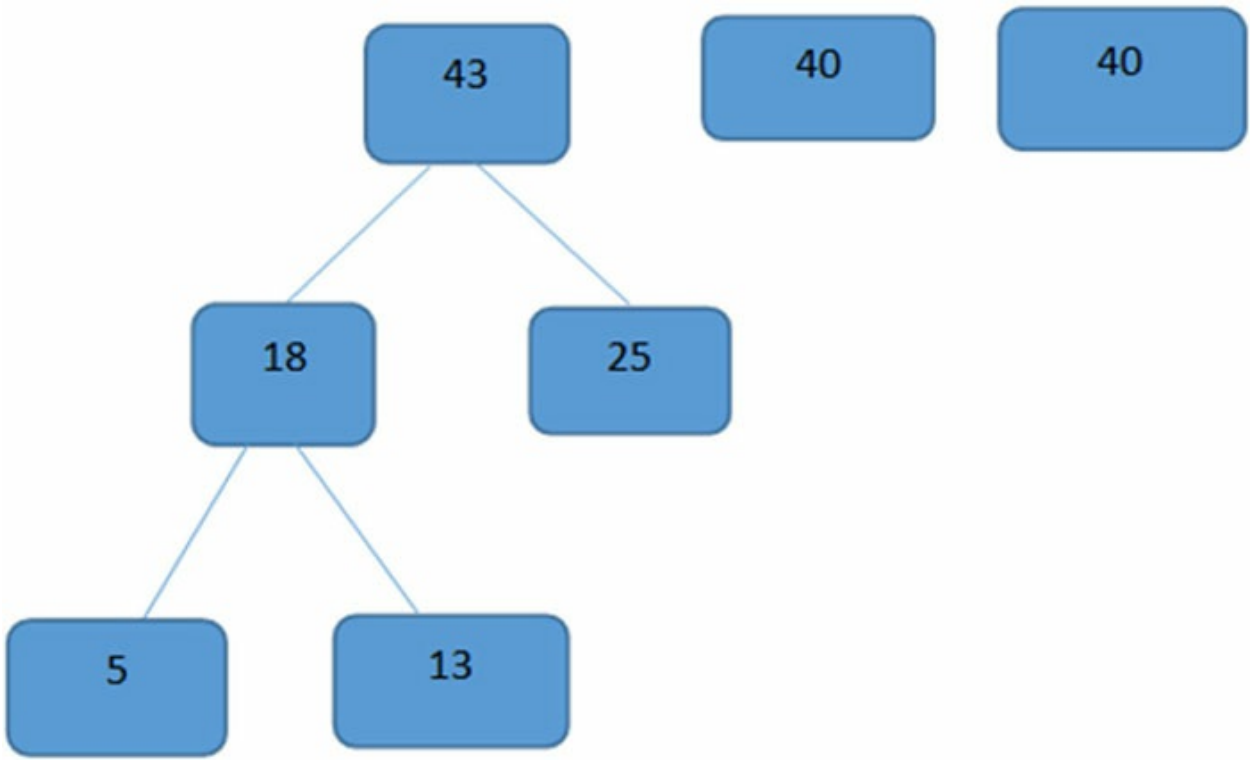


Figure 4.17: *18and 25 are Merged*

Step3:

Refer to *Figure 4.18*:

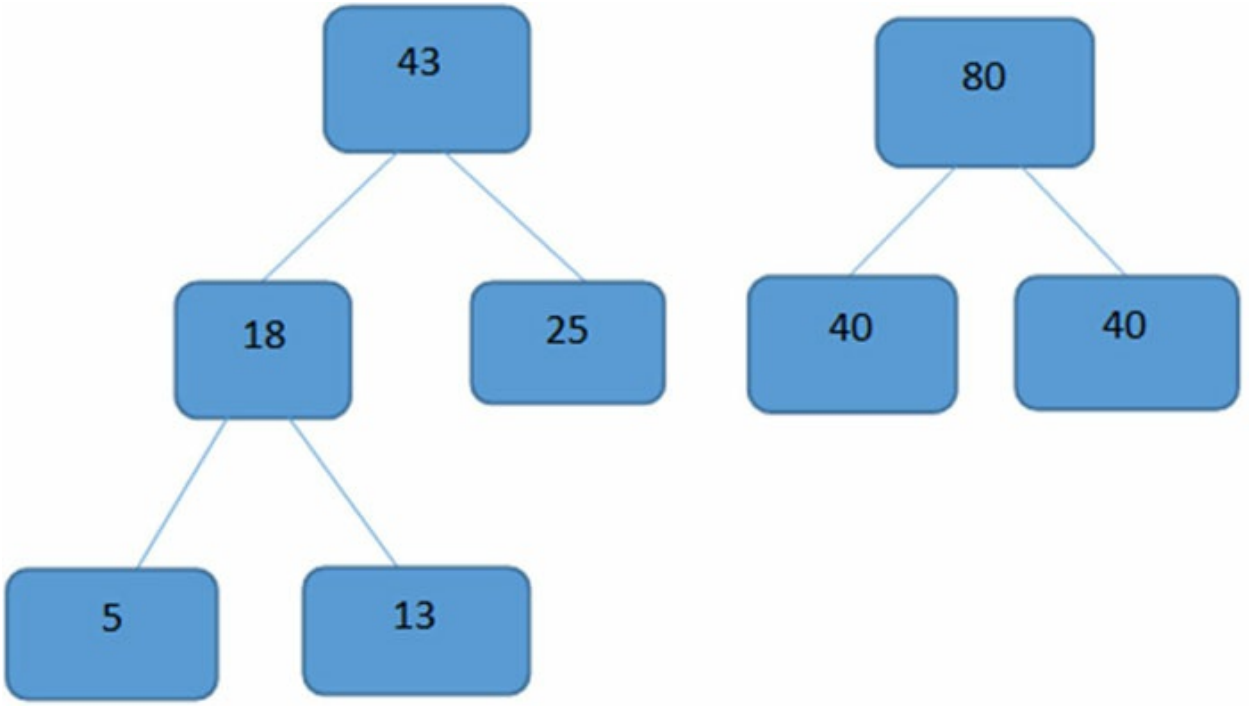


Figure 4.18: *40 and 40 are Merged*

Step 4:

Refer to *Figure 4.19*:

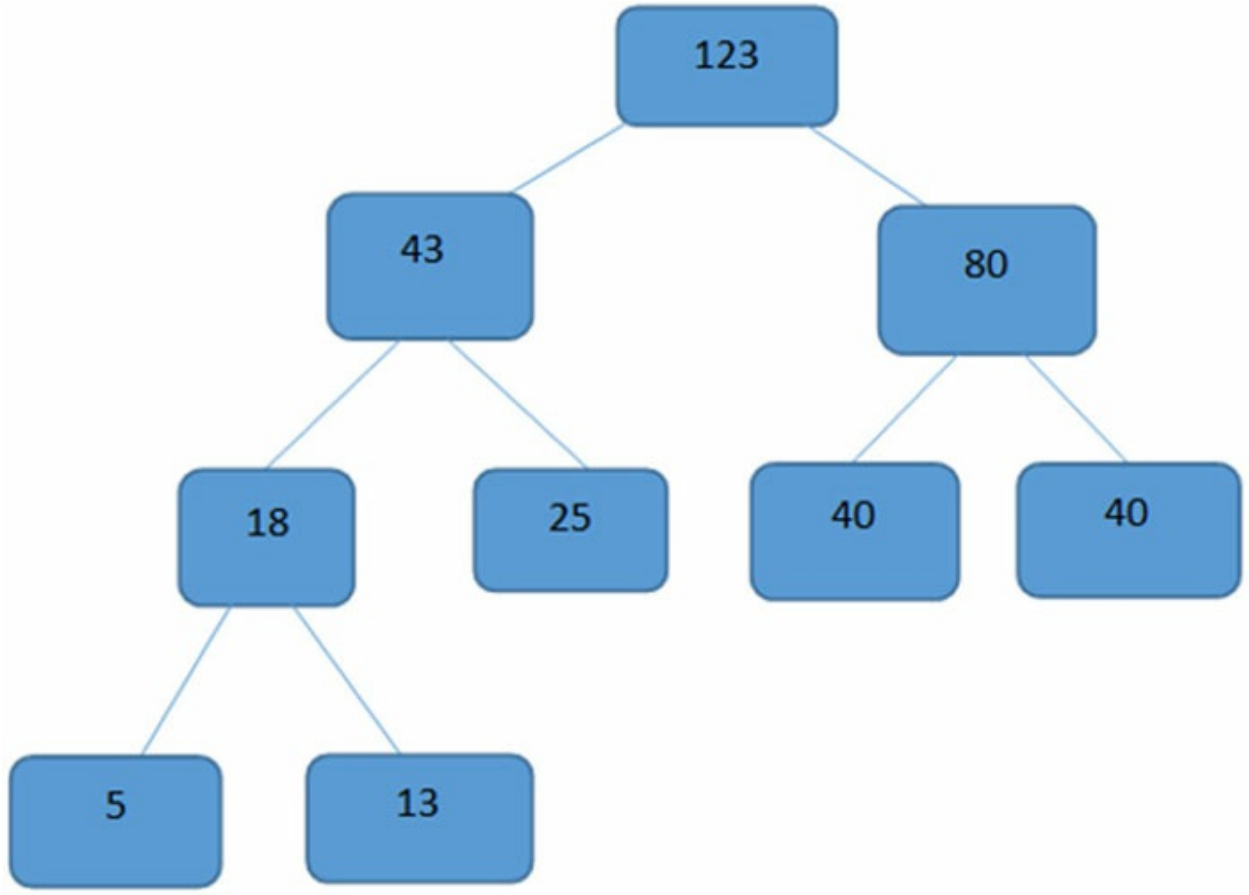


Figure 4.19: *43 and 80 are Merged*

Hence, the preceding problem takes 18 + 43 + 80 + 123= 264 numbers of comparisons.

Example 12

Given a set of unsorted files: 6,2,3,7,10,14

Solution

Arrange these elements in ascending order that is,2,3,6,7,10.

Step 1: Merged 2 and 3. Refer to *Figure 4.20:*

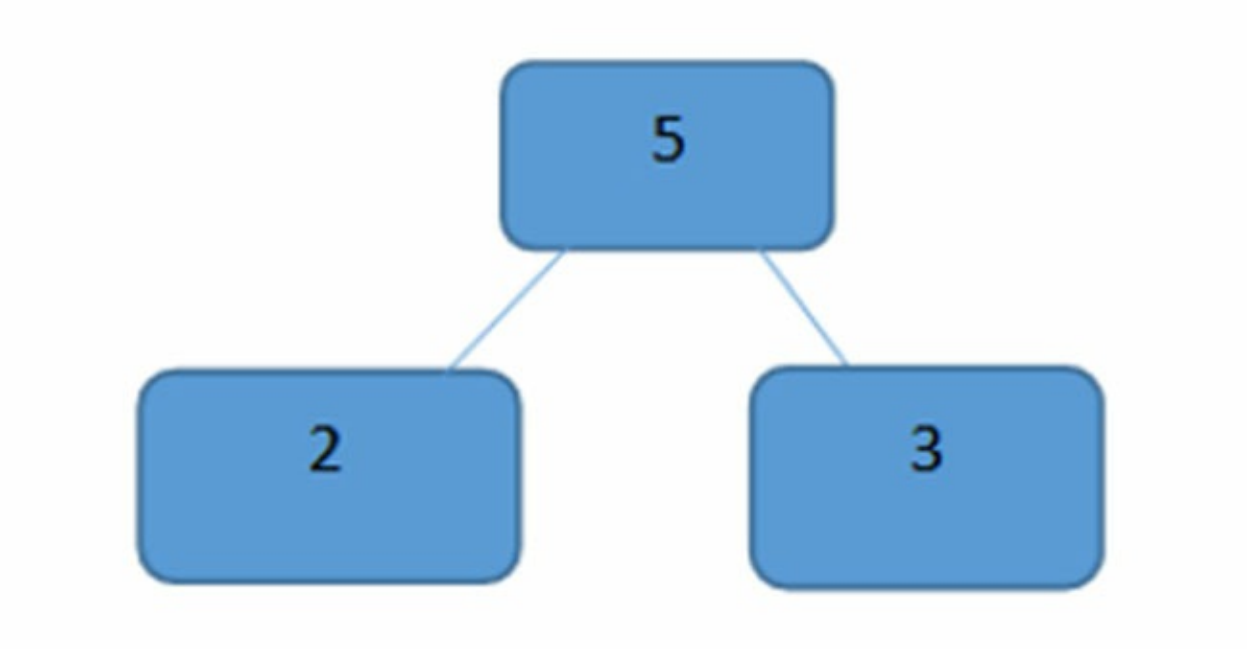


Figure 4.20: *Optimal Merge Pattern example*

Step 2: Merged 5 and 6. Refer to *Figure 4.21*:

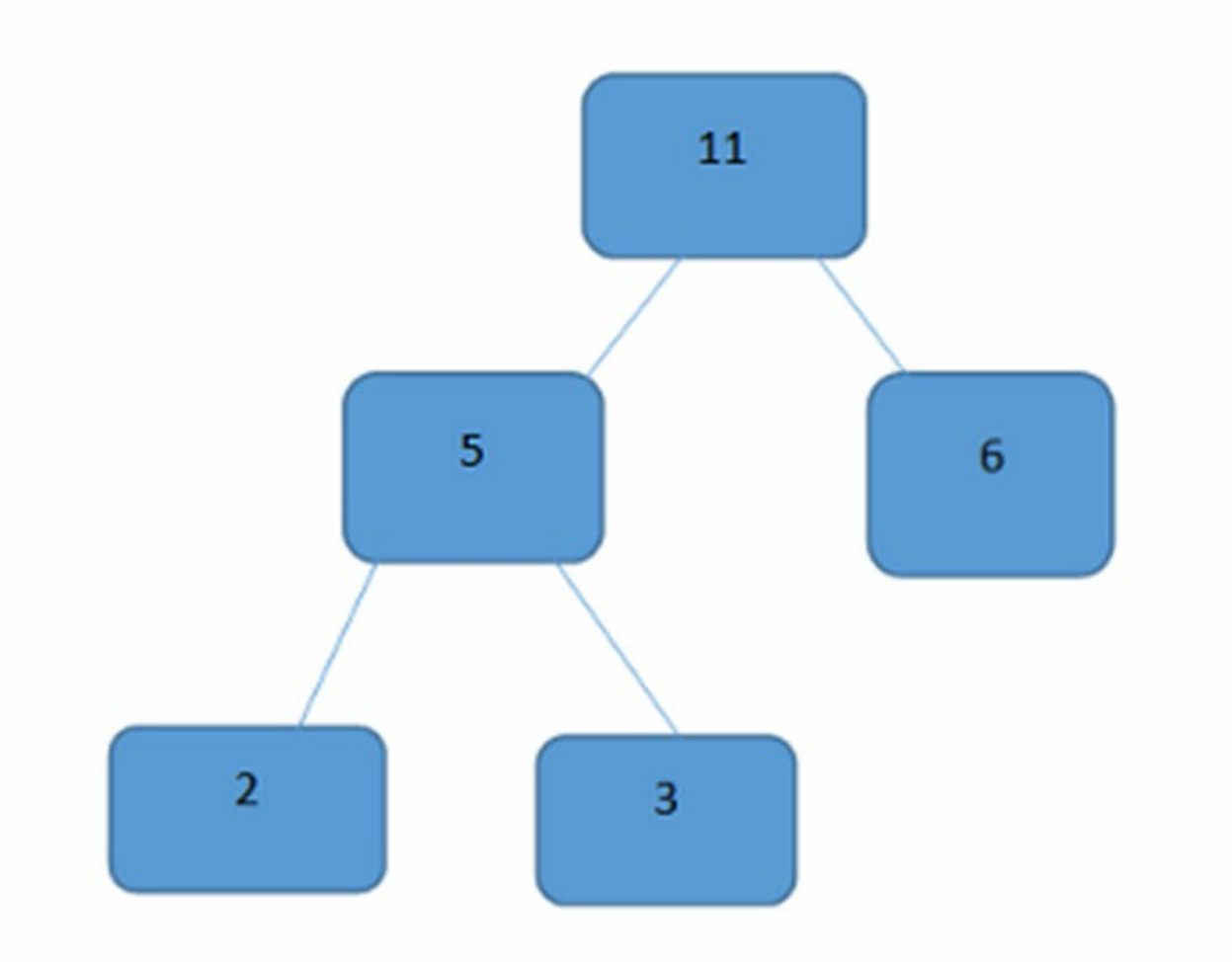


Figure 4.21: *Optimal Merge Pattern example*

Step 3: Merged 7 and 10. Refer to *Figure 4.22*:

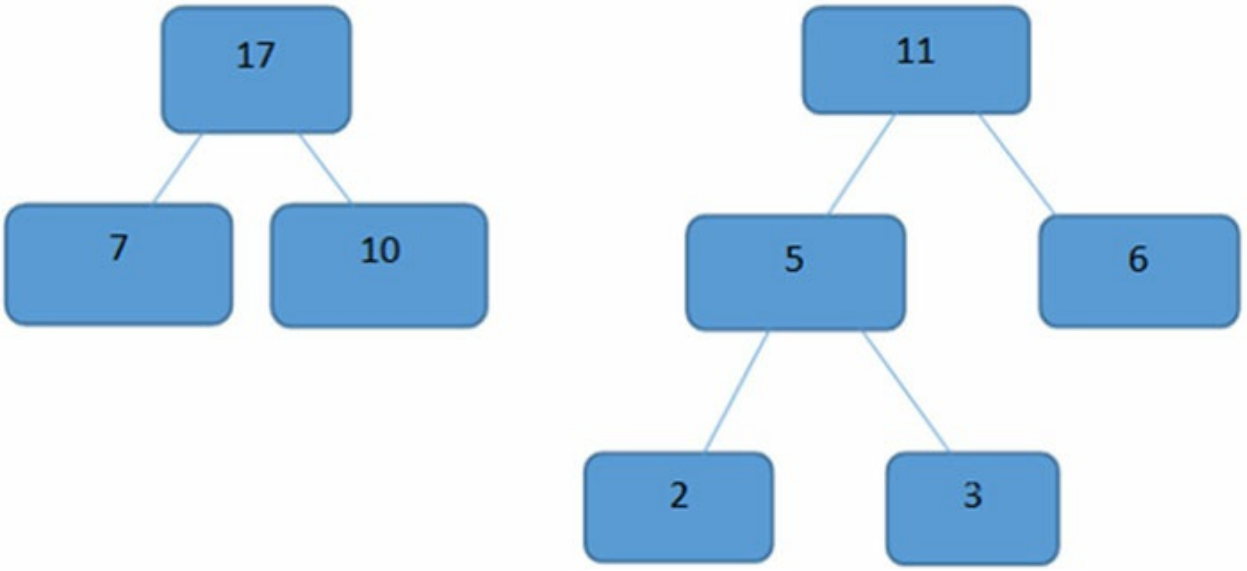


Figure 4.22: *Optimal Merge Pattern example*

Step 4: Optimal Merge Pattern Refer to *Figure 4.23*:

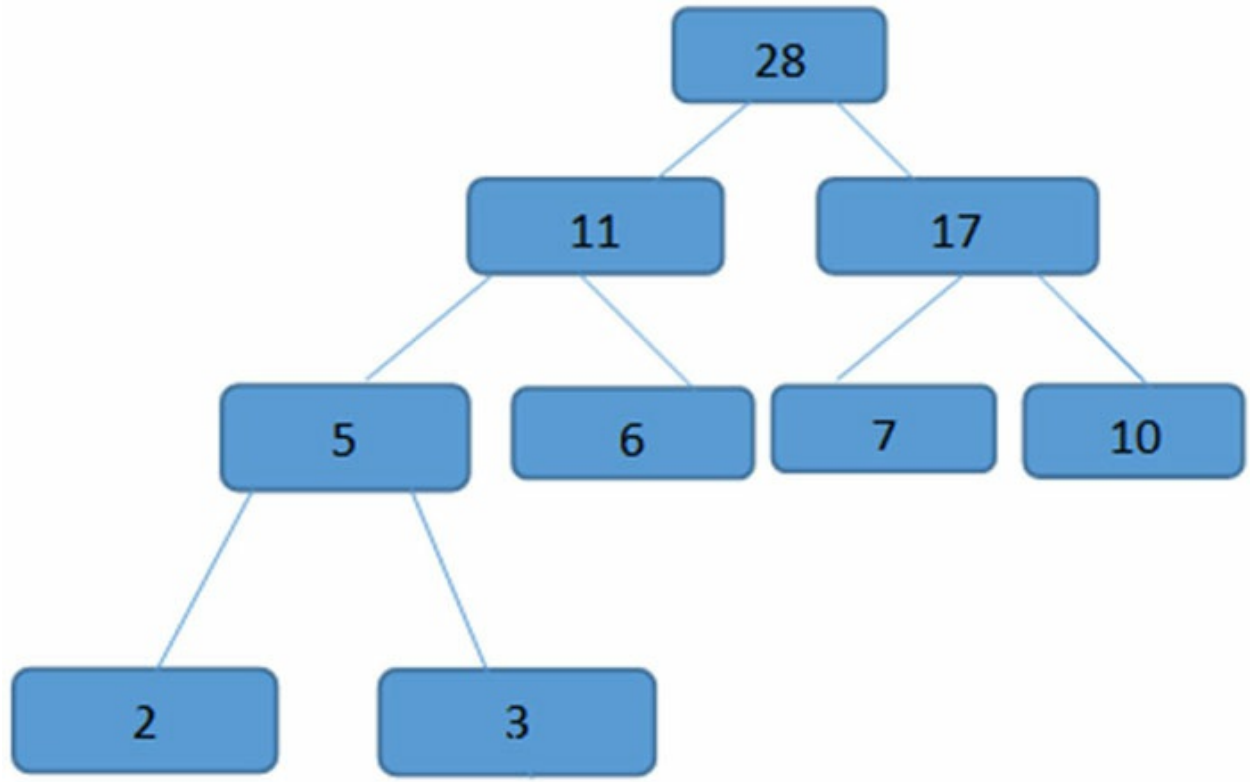


Figure 4.23: *Optimal Merge Pattern example*

Hence, the merging cost is 5+11+17+28 = 61

Subset cover problem

The set covering problem is an optimization problem. Given a pair (X, F), where X = $\{x_1, x_2, x_3, \ldots\ldots, x_m\}$ is a finite set (domain of element) and F=$\{ s_1, s_2, s_3, \ldots\ldots, s_n \}$ is a family of subset X, such that every element of x belongs to at least one set of F.

Consider a subset C ⊆ F (this is a collection of set over X). Therefore, we can say that C covers the domain if every element of X is in the same set of C.

Algorithm of set cover problem

The Greedy method selects the set S, at each stage that covers the greatest

number of remaining uncovered components. Let us now look at the algorithm:

```
Greedy-set-cover(X,F)
{
  U ← X
  C ← Φ
  While (U!= Φ)
  do-select an S ϵ F that maximize |S∩U|
  U ← U-S
  C C∪(S)
  Return (C)
}
```

Example 13

Problem:

Let *U={1,2,3,4,5} and S={S1,S2,S3}*

S1={4,1,3} cost=5

S2={2,5} cost=10

S3={1,4,3,2} cost=3

Find the minimum cost and maximum cost set cover.

Solution:

1. { S1,S2 } = S1∪S2 ={1,2,3,4,5}

   Since $S_1 \cup S_2 = U$, therefore, it is a set cover.

   Total cost = cost(S1)+cost(S2) =5+10=15

2. $\{S_1, S_3\} = S_1 \cup S_3 = \{1,2,3,4\}$

   Since $S_1 \cup S_3 = U$, therefore, it is not a set cover.

3. *{S2, S3} = S2∪S3 = {1,2,3,4,5}*

   Since $S_2 \cup S_3 = U$, therefore, it is a set cover.

   Total cost= cost(S2) +cost(S3) =10+3=13

4. *{S1, S2, S3} = S1∪S2∪S3 = {1,2,3,4,5}*

   Since $S_1 \cup S_2 \cup S_3 = U$, therefore, it is a set cover.

   Total cost = cost(S1) +cost(S2) +cost(S3) =5+10+3=18

Therefore, *Minimum cost=13* and *set cover is {S2, S3}*

And *Maximum cost =18* and *set cover is* $\{S_1, S_2, S_3\}$

Hence, we discussed that it was one of Karp's NP-complete problems and its other application is edge covering vertex covering. It is an NP-Hard problem because there is no polynomial time solution available for this problem. There is a polynomial time greedy approximate algorithm that provides a *[log(n)]* approximate time complexity.

If the cost of every subset is '1', then we can solve this problem in polynomial time.

If the cost for every subset is different, then we must pick the minimum cost and it is a non-polynomial time problem.

Container Loading Problem

The cargo will be put onto a big cargo ship. The cargo is divided into identically sized containers and is containerized. Weights of various containers may vary. Let us consider that the ship has a "c" cargo capacity and an integer array of container "i", with weights of "wi". The ship must be loaded with as much cargo as possible until the sum of the container weights < c.

Algorithm of Container Loading Problem

The containers are arranged in ascending order and loaded in the ship.

Let xi ∈ {0,1}

xi = 1 – ith container loaded in the ship.

xi = 0 – ith container is not loaded in the ship.

The value is assigned such that,

$$\sum_{i=1}^{n} xiwi \leq c, and \sum_{i=1}^{n} xi \quad is\ maximized$$

Example 14

Consider the capacity of the cargo ship c=400 and the number of containers m=8.

Refer to *Table 4.12*:

W1	W2	W3	W4	W5	W6	W7	W8
100	200	50	150	90	50	20	80

Table 4.12: *Containers with corresponding weights*

Step 1:

Arrange the containers in ascending order as per the weights, as shown in the following *Table 4.13*:

W7	W3	W6	W8	W5	W1	W4	W2
20	50	50	80	90	100	150	200

Table 4.13: *Containers arranged in ascending order*

Step 2:

Add the container with the minimum value into the ship, which is *W7* with the weight of 20, which is less than 400.

Step 3:

Add the next container with the next minimum value into the ship. These are W3 and W6, and since both have the weight of 50, either of one can be added.

Add W3 first, 20 + 50 = 70 which is less than 400.

Step 4:

Now add W6, 70 + 50 = 120 which is less than 400.

Step 5:

Now add W8, 120 + 80 = 200 which is less than 400.

Step 6:

Now add W5, 200 + 90 = 290 which is less than 400.

Step 7:

Now add W1, 290 + 100 = 390 which is less than 400.

Step 8:

Now select the next minimum value container W4, 390 + 150 = 540 which exceeds 400. Hence, stop the process. The selected containers for loading are

W7, W3, W6, W8, W5 and W1.

Conclusion

In combinational optimization, the knapsack problem is a well-known example, for when an optimal item or finite solution is required, and yet an exhaustive search is impractical. Job Sequencing with deadlines is used to understand which job provides us a maximum profit and at the same time, needs to be done within the range of the deadline provided. Minimum cost spanning trees help us reach the target node with the minimum cost possible among all the nodes. It consists of two algorithms – Kruskal's and Prim's. Optimal Storage on tapes is one of the applications of the Greedy Method. It is used for finding the optimal retrieval time for accessing programs that are stored on tape. The least number of computations required to create a single file from two or more sorted files is referred to as the optimal merging pattern. Subset cover problems find their applications in many different fields, such as data association problems in bioinformatics and computational biology.

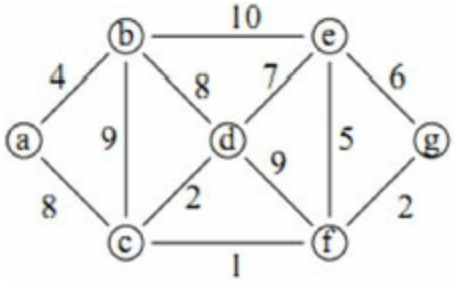
Key facts

- **Greedy Algorithm:** Any algorithm that makes the locally optimal decision at each stage when addressing a problem is said to be greedy.
- **Knapsack Problem:** The fundamental tenet of the greedy strategy is to compute the value/weight ratio for each item and then to arrange the items according to this ratio. Then add the item with the highest ratio to the knapsack until we can no longer add the next thing in its entirety. Finally, add the next item as much as you can.
- **Job Sequencing with deadlines:** Given a variety of jobs, each of which has a deadline and an associated profit if completed earlier than the deadline.
- **Minimum Cost Spanning Tree:** The weights of all the edges in the tree are added to determine the cost of the spanning tree. Many spanning trees may exist. The spanning tree with the lowest cost among all other spanning trees is known as a minimum spanning tree. Prims's and Kruskal's algorithms are used to construct the Minimum Spanning Tree.

- **Optimal Storage on Tapes:** Find the sequence in which the programs should be recorded in the tape for which the Mean Retrieval Time is reduced, given *n* programs that are stored on a computer tape.
- **Optimal Merge Pattern:** A binary merge tree with the shortest weighted external path length is an ideal merge pattern.
- **Subset cover Problem:** A set of subsets of a universe U of n elements, say S = S1, S2…,Sm, are given, where each subset Si has a cost. Find a S subcollection with a minimum cost that includes every aspect of U.
- **Container Loading problem:** A sizable cargo ship will be loaded with the container. The maximum number of containers must be put into the ship based on the capacity of the ship.

Questions

1. Compare Greedy and Dynamic programming approach for algorithm design. Explain how both can be used to solve knapsack problems. **[MU DEC'18 (10 MARKS)]**
2. Explain Job Sequencing with deadlines. Let n = 4, (P_1, P_2, P_3, P_4) = (100, 10, 15, 27) and (D_1, D_2, D_3, D_4) = (2, 1, 2, 1). Find a feasible solution. **[MU DEC'18 (10 MARKS)]**
3. Differentiate between greedy methods and dynamic programming? **[MU MAY'19 (5 MARKS)] [MU DEC'19 (5 MARKS)]**
4. Consider the instance of knapsack problem where n = 6, M = 15, profits are (P_1, P_2, P_3, P_4, P_5, P_6) = (1, 2, 4, 4, 7, 2) and weights are (W_1, W_2, W_3, W_4, W_5, W_6) = (10, 5 4, 2, 7, 3). Find maximum profit using fractional Knapsack. **[MU MAY'19 (10 MARKS)]**
5. Construct a minimum spanning tree using Kruskal's and Prim's algorithm and find out the cost with all intermediate steps. **[MU MAY'19 (10 MARKS)]**



6. Given a set of 9 jobs (J1,J2,J3,J4,J5,J6,J7,J8,J9) where each job has a deadline (5,4,3,3,4,5,2,3,7) and profit (85,25,16,40,55,19,92,80,15) associated to it. Each job takes 1 unit of time to complete and only one job can be scheduled at a time. We earn the profit if and only if the job is completed by its deadline. The task is to find the maximum profit and the number of jobs done. **[MU MAY'19 (10 MARKS)]**
7. Explain the knapsack problem with an example. **[MU DEC'19 (10 MARKS)]**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 5

Dynamic Algorithms and NP-Hard and NP-Complete

Introduction

This chapter talks about Dynamic Algorithms. The need for dynamic algorithms and introduction to NP-Hard & NP-Complete are discussed with examples.

The **All Pairs Shortest Path (APSP)** algorithm is an algorithm that can be used to find the shortest path between all pairs of nodes in a graph. The APSP problem is important in a variety of contexts, including transportation networks, communication networks, and social networks. **The 0/1 Knapsack problem** involves selecting a subset of items from a given set of items, such that the total value of the selected items is maximized, subject to the constraint that the total weight of the selected items does not exceed a given capacity. It has many practical applications, including inventory management, resource allocation, and selection of investments.

The **Travelling Salesman Problem (TSP)** involves finding the shortest possible route that visits a given set of cities and returns to the starting city, subject to the constraint that each city must be visited exactly once. The TSP has many practical applications, including logistics, transportation, and routing. The **Coin Changing Problem** involves finding the minimum number of coins needed to make a given amount of money, given a set of denominations of coins. It has many practical applications, including currency exchange and vending machines.

Matrix chain multiplication is an optimization problem that involves finding the most efficient way to multiply a sequence of matrices. It has many practical applications, including computer graphics, machine learning, and scientific computing. **Flow shop scheduling** involves scheduling a set of jobs on a set of machines, such that the total processing time is minimized. It is an important problem in manufacturing and production, where it is used to

optimize the use of resources and minimize the time required to produce a product.

An **optimal Binary Search Tree (BST)** is a binary tree that is used to store a sorted list of items such that the search time for any item is minimized. Optimal BSTs are used in a variety of applications, including databases, search engines, and data structures.

Structure

In this chapter, we will discuss the following topics:

- Dynamic Algorithms
- All Pair Shortest Path Algorithm
- 0/1 Knapsack
- Traveling Salesman problem
- Coin Changing problem
- Matrix Chain Multiplication
- Flow shop scheduling
- Optimal Binary Search Tree
- NP – HARD & NP – COMPLETE

Objectives

By the end of this chapter, the reader will learn about various classical computer science problems and their efficient solutions, as well as using dynamic programming approaches with their applications. Lastly, the chapter will also discuss NP-Hard & NP-Complete problems.

Dynamic algorithms

Dynamic Algorithm is mainly an optimization algorithm. Dynamic Programming can be used to optimize any recursive solution that contains repeated calls for the same inputs. Instead of repetitive re-computation for the results of subproblems at a later stage, the idea is to just store them. This module consists of All Pair Shortest Path, 0/1 Knapsack, Travelling Salesman Problem, Coin Changing Problem and many more such algorithms.

Moreover, this chapter introduces NP-Hard & NP-Complete problems.

All Pair Shortest Path Algorithm

Some features of the All Pair Shortest Path algorithm are as follows:

- The Algorithm is used to find the all pairs shortest path problems from a given weighted graph.
- This algorithm will generate a matrix that represents the shortest path between any node and every other node in the network.
- A weighted graph will be given to us, and looking at the graph, we must build a matrix of all the paths and its costs.
- Each step will be explained in detail in the following example solved.

Example 1

Refer to *Figure 5.1* for this example:

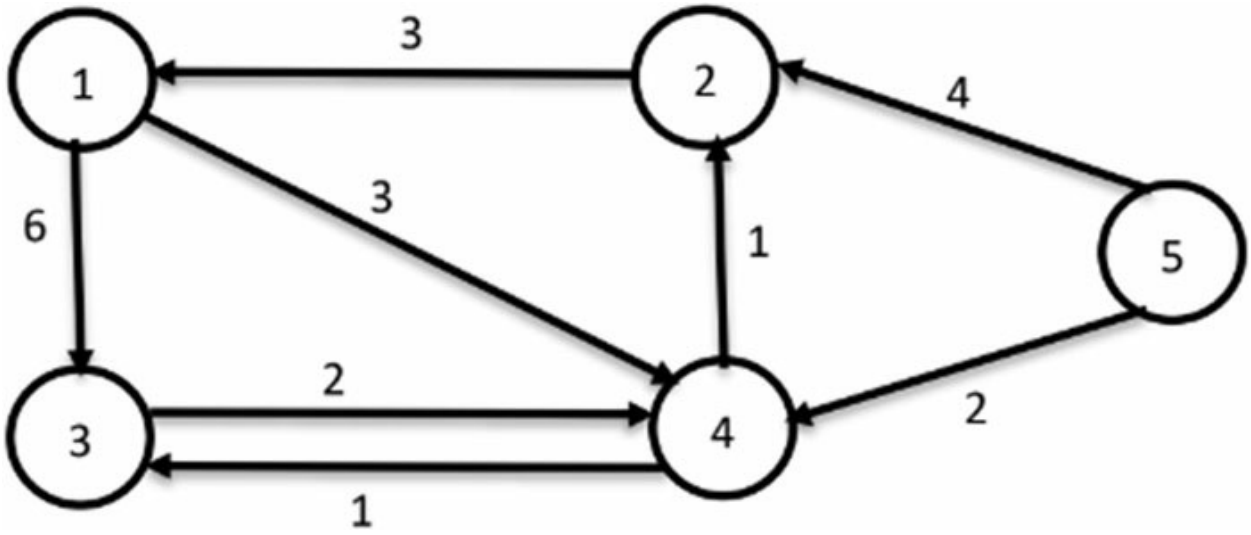


Figure 5.1: *APSP example*

Now, we will keep the first column and first row same for the new matrix we will be building, that is, A^1. Refer to the following matrix in *Figure 5.2*:

$$A^0 = \begin{pmatrix} 0 & \infty & 6 & 3 & \infty \\ 3 & 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & 2 & \infty \\ \infty & 1 & 1 & 0 & \infty \\ \infty & 4 & \infty & 2 & 0 \end{pmatrix}$$

Figure 5.2: *Matrix*

We will now fill other rows and columns by comparing the shortest distances between the two points.

We will be looking at matrix A^0.

Diagonal elements will always be 0.

We are using 1 as an intermediate branch, as we are calculating A^1.

$A^0[2,3] = \infty$

$A^0[2,1] + A^0[1,3] = 3 + 6 = 9$

$A^0[2,3]$ $A^0[2,1] + A^0[1,3]$

$A^0[2,4] = \infty > A^0[2,1] + A^0[1,4] = 3 + 3 = 6$

$A^0[2,5] = \infty$ $A^0[2,1] + A^0[1,5] = 3 + \infty = \infty$

$A^0[3,2] = 0$ $A^0[3,1] + A^0[1,2] = \infty + \infty = \infty$

$A^0[3,4] = 2 < A^0[3,1] + A^0[1,4] = \infty + 3 = \infty$

$A^0[3,5] = \infty$ $A^0[3,1] + A^0[1,5] = \infty + \infty = \infty$

$A^0[4,2] = 1 < A^0[4,1] + A^0[1,2] = \infty + \infty = \infty$

$A^0[4,3] = 1 < A^0[4,1] + A^0[1,3] = \infty + \infty = \infty$

$A^0[4,5] = \infty$ $A^0[4,1] + A^0[1,5] = \infty + \infty = \infty$

$A^0[5,2] = 4 < A^0[5,1] + A^0[1,2] = \infty + \infty = \infty$

$A^0[5,3] = \infty \quad A^0[5,1] + A^0[1,3] = \infty + \infty = \infty$

$A^0[5,4] = 2 \; < \; A^0[5,1] + A^0[1,4] = \infty + \infty = \infty$

Now, we will fill the preceding values in the A^1 matrix, as shown in *Figure 5.3*:

$$A^1 = \begin{pmatrix} 0 & \infty & 6 & 3 & \infty \\ 3 & 0 & 9 & 6 & \infty \\ \infty & \infty & 0 & 2 & \infty \\ \infty & 1 & 1 & 0 & \infty \\ \infty & 4 & \infty & 2 & 0 \end{pmatrix}$$

Figure 5.3: *Matrix*

Now, we will keep the second row and second column of A^1 the same for the new matrix we will build, that is, A^2.

We will also fill other rows and columns by comparing the shortest distances between two points.

We will be looking at matrix A^1.

Diagonal elements will always be 0. We are using 2 as an intermediate branch as we are calculating A^2.

$A^1[1,3] = 6 \; < \; A^1[1,2] + A^1[2,3] = \infty + 9 = \infty$

$A^1[1,4] = 3 \; < \; A^1[1,2] + A^1[2,4] = \infty + 3 = \infty$

$A^1[1,5] = \infty \quad A^1[1,2] + A^1[2,5] = \infty + \infty = \infty$

$A^1[3,1] = \infty \quad A^1[3,2] + A^1[2,1] = \infty + 3 = \infty$

$A^1[3,4] = 2 \; < \; A^1[3,2] + A^1[2,4] = \infty + 6 = \infty$

$A^1[3,5] = \infty \quad A^1[3,2] + A^1[2,5] = \infty + \infty = \infty$

$A^1[4,1] = \infty > A^1[4,2] + A^1[2,1] = 1 + 3 = 4$

$A^1[4,3] = 1 < A^1[4,2] + A^1[2,3] = 1 + 9 = 10$

$A^1[4,5] = \infty \quad A^1[4,2] + A^1[2,5] = 1 + \infty = \infty$

$A^1[5,1] = \infty > A^1[5,2] + A^1[2,1] = 4 + 3 = 7$

$A^1[5,3] = \infty > A^1[5,2] + A^1[2,3] = 4 + 9 = 13$

$A^1[5,4] = 2 < A[5,2] + A^1[2,4] = 4 + 6 = 10$

Now, we will fill the preceding values in A^2 matrix, as shown in *Figure 5.4*:

$$A^2 = \begin{pmatrix} 0 & \infty & 6 & 3 & \infty \\ 3 & 0 & 9 & 6 & \infty \\ \infty & \infty & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 7 & 4 & 13 & 2 & 0 \end{pmatrix}$$

Figure 5.4: *Matrix*

Now, we will keep the third row and third column of A^2 the same for the new matrix we will build, that is, A^3.

We will now fill other rows and columns by comparing the shortest distances between two points.

We will be looking at matrix A^2.

Diagonal elements will always be 0.

We are using 3 as intermediate branch as we are calculating A^3.

$A^2[1,2] = \infty \quad A^2[1,3] + A^2[3,1] = 6 + \infty = \infty$

$A^2[1,4] = 3 < A^2[1,3] + A^2[3,4] = 6 + 2 = 8$

$A^2[1,5] = \infty \quad A^2[1,3] + A^2[3,5] = 6 + \infty = \infty$

$A^2[2,1] = 3 \; < \; A^2[2,3] + A^2[3,1] = 9 + \infty = \infty$

$A^2[2,4] = 6 \; < \; A^2[2,3] + A^2[3,4] = 9 + 2 = 11$

$A^2[2,5] = \infty \quad A^2[2,3] + A^2[3,5] = 9 + \infty = \infty$

$A^2[4,1] = 4 \quad A^2[4,3] + A^2[3,1] = 1 + \infty = \infty$

$A^2[4,2] = 1 \; < \; A^2[4,3] + A^2[3,2] = 1 + \infty = \infty$

$A^2[4,5] = \infty \quad A^2[4,3] + A^2[3,5] = 1 + \infty = \infty$

$A^2[5,1] = 7 \; < \; A^2[5,3] + A^2[3,1] = 13 + \infty = \infty$

$A^2[5,2] = 4 \; > \; A^2[5,3] + A^2[3,2] = 13 + \infty = \infty$

$A^2[5,4] = 2 \; < \; A^2[5,3] + A^2[3,4] = 13 + 2 = 15$

Now, we will fill the preceding values in A^3 matrix, as shown in *Figure 5.5*:

$$A^3 = \begin{pmatrix} 0 & \infty & 6 & 3 & \infty \\ 3 & 0 & 9 & 6 & \infty \\ \infty & \infty & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 7 & 4 & 13 & 2 & 0 \end{pmatrix}$$

Figure 5.5: *Matrix*

Now, we will keep the fourth row and fourth column of A^3 the same, for the new matrix we will be building, that is, A^4.

We will now fill other rows and columns by comparing the shortest distances between two points.

We will be looking at matrix A^3.

Diagonal elements will always be 0.

We are using 4 as intermediate branch as we are calculating A^4.

$A^3[1,2] = \infty > A^3[1,4] + A^3[4,2] = 3 + 1 = 4$

$A^3[1,3] = 6 > A^3[1,4] + A^3[4,3] = 3 + 1 = 4$

$A^3[1,5] = \infty \quad A^3[1,4] + A^3[4,5] = 3 + \infty = \infty$

$A^3[2,1] = 3 < A^3[2,4] + A^3[4,1] = 6 + 4 = 10$

$A^3[2,3] = 9 > A^3[2,4] + A^3[4,3] = 6 + 1 = 7$

$A^3[2,5] = \infty \quad A^3[2,4] + A^3[4,5] = 6 + \infty = \infty$

$A^3[3,1] = \infty > A^3[3,4] + A^3[4,1] = 2 + 4 = 6$

$A^3[3,2] = \infty > A^3[3,4] + A^3[4,2] = 3 + 1 = 4$

$A^3[3,5] = \infty \quad A^3[3,4] + A^3[4,5] = 2 + \infty = \infty$

$A^3[5,1] = 7 > A^3[5,4] + A^3[4,1] = 2 + 4 = 6$

$A^3[5,2] = 4 > A^3[5,4] + A^3[4,2] = 2 + 1 = 3$

$A^3[5,3] = 13 > A^3[5,4] + A^3[4,3] = 2 + 1 = 3$

Now, we will fill the preceding values in A^4 matrix, as shown in *Figure 5.6*:

$$A^4 = \begin{pmatrix} 0 & 4 & 4 & 3 & \infty \\ 3 & 0 & 7 & 6 & \infty \\ 6 & 3 & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 6 & 3 & 3 & 2 & 0 \end{pmatrix}$$

Figure 5.6: *Matrix*

Now, we will keep the fifth row and fifth column of A^4 the same for the new matrix that we will build, that is, A^5.

We will also fill other rows and columns by comparing the shortest distances between the two points.

We will be looking at matrix A^4.

Diagonal elements will always be 0.

We are using 5 as intermediate branchm as we are calculating A^5.

$A^4[1,2] = 4 < A^4[1,5] + A^4[5,2] = \infty + 3 = 4$

$A^4[1,3] = 4 < A^4[1,5] + A^4[5,3] = \infty + 3 = \infty$

$A^4[1,4] = 3 < A^4[1,5] + A^4[5,4] = \infty + 2 = \infty$

$A^4[2,1] = 3 < A^4[2,5] + A^4[5,1] = \infty + 6 = \infty$

$A^4[2,3] = 7 < A^4[2,5] + A^4[5,3] = \infty + 3 = \infty$

$A^4[2,4] = 6 < A^4[2,5] + A^4[5,4] = \infty + 2 = \infty$

$A^4[3,1] = 6 > A^4[3,5] + A^4[5,1] = \infty + 6 = \infty$

$A^4[3,2] = 3 > A^4[3,5] + A^4[5,2] = \infty + 3 = \infty$

$A^4[3,4] = 2 < A^4[3,5] + A^4[5,4] = \infty + 2 = \infty$

$A^4[4,1] = 4 > A^4[4,5] + A^4[5,1] = \infty + 6 = \infty$

$A^4[4,2] = 1 > A^4[4,5] + A^4[5,2] = \infty + 3 = \infty$

$A^4[4,3] = 1 > A^4[4,5] + A^4[5,3] = \infty + 2 = \infty$

Now, we will fill the preceding values in A^5 matrix, as shown in *Figure 5.7*:

$$A^5 = \begin{pmatrix} 0 & 4 & 4 & 3 & \infty \\ 3 & 0 & 7 & 6 & \infty \\ 6 & 3 & 0 & 2 & \infty \\ 4 & 1 & 1 & 0 & \infty \\ 6 & 3 & 3 & 2 & 0 \end{pmatrix}$$

Figure 5.7: *Matrix*

Thus, the shortest distance between all the pairs is obtained.

0/1 Knapsack Problem

Let us consider an algorithm.

Algorithm

```
int w, k;
for (w=0; w <= W; w++)
   B[w] = 0
for (k=0; k<n; k++)
    {
   for (w = W; w>= w[k]; w--)
   {
    if (B[w – w[k]] + v[k] > B[w])
    B[w] = B[w – w[k]] + v[k]
   }
    }
```

Example 2

Weights: 2 3 3 4 6

Values: 1 2 5 9 4

Knapsack Capacity (W) = 10

Refer to *Table 5.1*:

	0	1	2	3	4	5	6	7	8	9	10
2(1)	0	0	1	1	1	1	1	1	1	1	1
3(2)	0	0	1	2	2	3	3	3	3	3	3
3(5)	0	0	1	5	5	6	7	7	8	8	8
4(9)	0	0	1	5	9	9	10	14	14	15	16
6(4)	0	0	1	5	9	9	10	14	14	15	16

Table 5.1: *Binary Knapsack example*

Example 3

W=10

Refer to the following *Table 5.2*:

i	Item	w_i	v_i
0	I_0	4	6
1	I_1	2	4
2	I_2	3	5
3	I_3	1	3
4	I_4	6	9
5	I_5	4	7

Table 5.2: *Binary Knapsack example*

Solution:

Refer to the following *Table 5.3*:

Item	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	6	6	6	6	6	6	6
1	0	0	4	4	6	6	10	10	10	10	10
2	0	0	4	5	6	9	10	11	11	15	15
3	0	3	4	7	8	9	12	13	14	15	18
4	0	3	4	7	8	9	12	13	14	16	18
5	0	3	4	7	8	10	12	14	15	16	19

Table 5.3: *Binary Knapsack example*

Example 4

Here we have a selection of n=4 items, and capacity of knapsack M=8. Refer to *Table 5.4*:

Item i	Value vi	Weight wi
1	15	1
2	10	5
3	9	3
4	5	4

Table 5.4: *Binary Knapsack example*

Refer to *Table 5.5*:

		Capacity Remaining								
		g=0	g=1	g=2	g=3	g=4	g=5	g=6	g=7	g=8
k=0	f(0,g)=	0	0	0	0	0	0	0	0	0
k=1	f(1,g)=	0	**15**	15	15	15	15	15	15	15
k=2	f(2,g)=	0	**15**	15	15	15	15	25	25	25
k=3	f(3,g)=	0	15	15	15	**24**	24	25	25	25
k=4	f(4,g)=	0	15	15	15	14	24	25	25	**29**

Table 5.5: *Binary Knapsack example*

Traveling Salesman Problem

The **Traveling Salesman Problem** (**TSP**) is a great dynamic programming

algorithm, and its objective is based on optimization. TSP is the method to easily represent the location of all the nodes as a graph structure.

For example, consider a group of cities and the distance between each pair of cities. The aim is to find the minimum distance of route that includes all the city only once and returns to the first city.

Example 5

Refer to *Figure 5.8*:

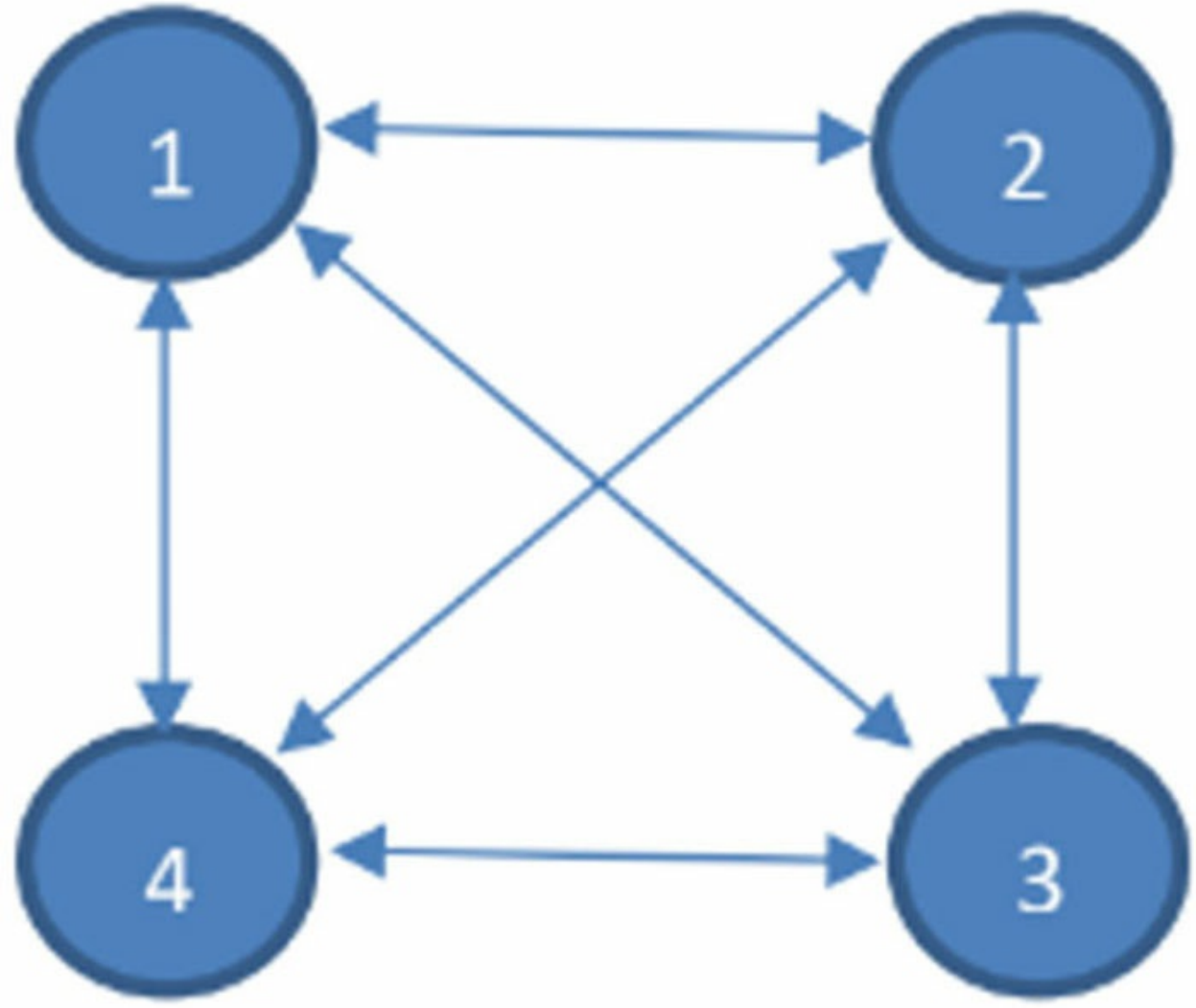


Figure 5.8: *TSP Example*

Refer to *Table 5.6*:

	1	2	3	4

1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Table 5.6: *TSP example*

Refer to the following *Figure 5.9*:

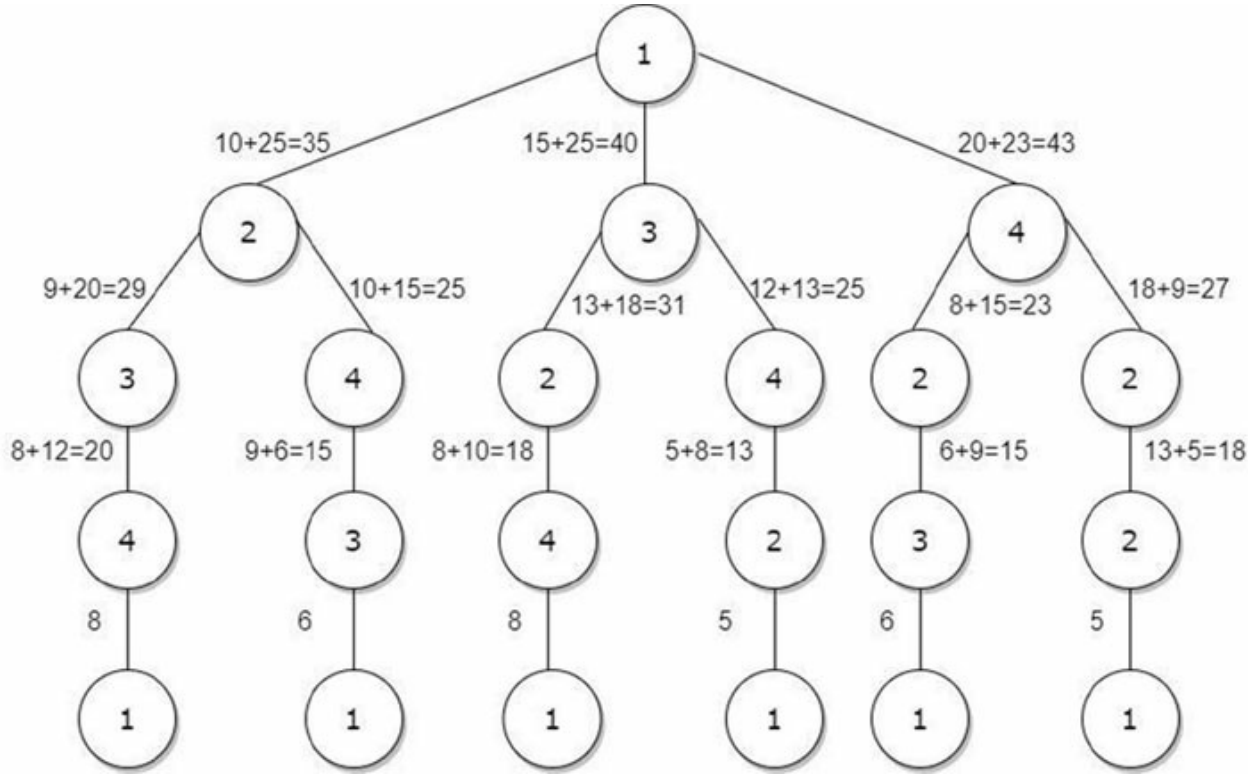


Figure 5.9: *TSP Example*

g(i,s) = min{Cik + g(k,s-{k})}

g(1,{2,3,4}) = min{C1k + g(k,{2,3,4}-{k}}

g(2,1) = 5

g(3,1) = 6

g(4,1) = 8

g(2,{3}) =15

g(2,{4}) =18

$g(3,\{2\}) = 18$

$g(3,\{4\}) = 20$

$g(4,\{2\}) = 13$

$g(4,\{3\}) = 15$

$g(2,\{3,4\}) = 25$

$g(3,\{2,4\}) = 25$

$g(4,\{2,3\}) = 23$

$$g(1,\{2,3,4\}) = \min\{C12 + g(2,\{,3,4\}), C13 + g(3,\{2,4\}), C14 + g(4,\{2,3\})\}$$
$$= \min\{35,40,43\}$$
$$= 35$$

Sequence = $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$

Example 6

Refer to *Figure 5.10*:

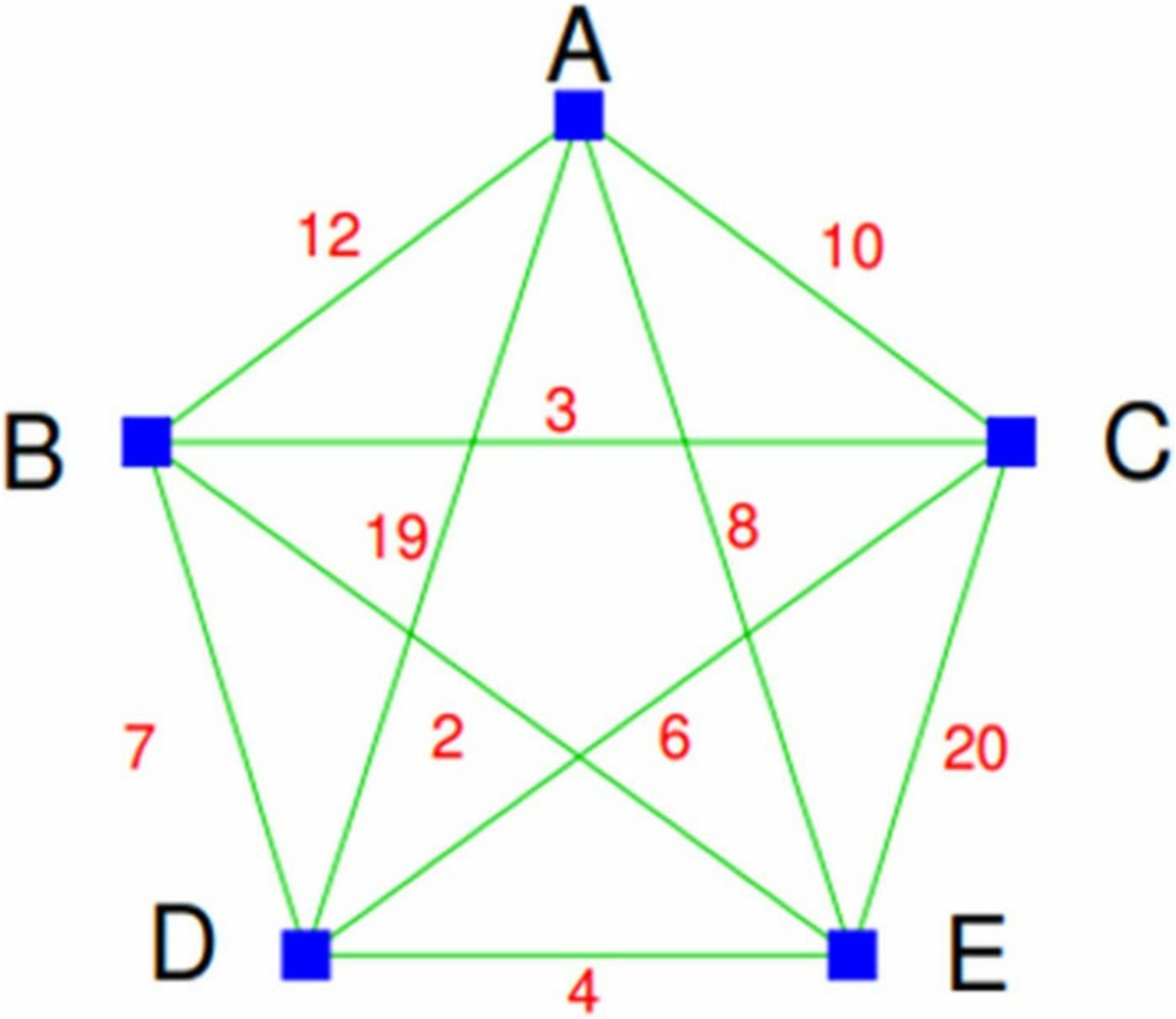


Figure 5.10: *TSP example*

Refer to *Table 5.7*:

	A	B	C	D	E
A	0	12	10	19	8
B	12	0	3	7	2
C	10	3	0	6	20
D	19	7	6	0	4
E	8	2	20	4	0

Table 5.7: *TSP example*

$g(i,s) = min\{Cik + g(k,s-\{k\})\}$

g(A,{B,C,D,E}) = min{C1k + g(k,{B,C,D,E}-{k}}

g(B,A) = 12
g(C,A) = 10
g(D,A) = 19
g(E,A) = 8

g(B,{C}) = 3+10=13
g(B,{D}) =7+19=26
g(B,{E}) =2+8=10
g(C,{B}) =3+12=15
g(C,{D}) =6+19=25
g(C,{E}) =20+8=28
g(D,{B}) =7+12=19
g(D,{C}) =6+10=16
g(D,{E}) =4+8=12
g(E,{B}) =2+12=14
g(E,{C}) =20+10=30
g(E,{D}) =4+19=23

g(B,{C,D}) =7+16=23
g(B,{D,E}) =7+12=19
g(B,{C,E}) =3+28=31
g(C,{B,D}) =6+19=25
g(C,{D,E}) =6+12=18
g(C,{B,E}) =3+10=13
g(D,{B,C}) =7+13=20
g(D,{C,E}) =6+28=34
g(D,{B,E}) =7+10=17
g(E,{B,C}) =2+13=15
g(E,{C,D}) =20+16=36

$g(E,\{B,D\}) = 2+19=21$

$g(B,\{C,D,E\}) = 3+18=21$

$g(C,\{B,D,E\}) = 6+17=23$

$g(D,\{B,C,E\}) = 6+13=19$

$g(E,\{B,C,D\}) = 4+20=24$

$g(A,\{B,C,D,E\}) = min\{21,23,19,24\}+19$

$= 19+19$

$= 38$

Sequence = $A \rightarrow E \rightarrow B \rightarrow C \rightarrow D \rightarrow A$

Coin Changing problem

The Coin Changing Problem helps find the minimum number of ways of making changes, using a given set of coins for the amount.

Follow the given steps:

- With each coin, we can wither include the coin in the table, or exclude it.
- We have to first check the value of the coin. If the coin value is less than or equal to the required amount, then we go for the following cases to include or exclude the value.
    - **Include**: Subtract the value of amount by value of coin and include it in the sub problem solution (amount-v[i]).
    - **Exclude**: Do not consider that the coin and solution will be the same amount.
- If the coin value is greater than the required amount, then exclude the value.

Example 7

Consider Total Amount = 5

Coins = {1, 2, 3}

Refer to *Table 5.8:*

	Amount					
	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	3	4	5

Table 5.8: *Coin Changing Problem example*

Example 8

Amount = 10

Coins = {2, 5, 3, 6}

Refer to *Table 5.9*:

	Amount									
	1	2	3	4	5	6	7	8	9	10
2	0	1	0	1	0	1	0	1	0	1
5	0	1	0	1	1	1	1	1	1	2
3	0	1	1	1	2	2	1	3	2	3
6	0	1	1	1	2	3	2	4	3	4

Table 5.9: *Coin Changing Problem example*

Minimum number of ways = 4

Matrix Chain Multiplication

The major features of Matrix Chain multiplication are as follows:

- Dynamic programming can be used to solve matrix chain multiplication.
- The objective here is to locate the most effective approach to duplicate these matrices.
- Choosing the order of the matrix multiplications involved is the challenge and doing the multiplications is not the problem.
- Multiplication of matrices is associative, meaning that no matter how

we solve it, the outcome will always be the same.

- The prerequisite for multiplying two matrices is that the first matrix's number of columns should match the second matrix's number of rows, that is, a*b b*c.
- When you multiply two matrices, the dimension of resultant matrix will be a*c.
- In matrix chain multiplication, the number of matrices as well as their dimensions will be given. They will be arranged such that they can be multiplied.
- We must take a pair at a time, and so your problem is to recognize which pair we should select such that the total cost of multiplying them is minimized.
- The problem is to know how to multiply them, meaning how we should parenthesize them.
- We try all possibilities and pick the best one.
- Dynamic programming uses tabulation.

Refer to the following *Figure 5.11*:

m	1	2	3	4
1				
2				
3				
4				

s	1	2	3	4
1				
2				
3				
4				

Figure 5.11: *Matrix Chain Multiplication example*

Example 9

M[1,4] Matrix Chain Multiplication for A_1, A_2 ,A_3 ,A_4 with $P_0 = 5$, $P_1 = 4$, $P_2 = 6$, $P_3 = 2$, $P_4 = 7$

Refer to the following *Table 5.10*:

A_1	A_2	A_3	A_4
5 × 4	4 × 6	6 × 2	2 × 7

Table 5.10: *Matrix Chain Multiplication example*

STEP 1:

$m(1,1) = A_1 = 0$

$m(2,2) = A_2 = 0$

$m(3,3) = A_3 = 0$

$m(4,4) = A_4 = 0$

STEP 2:

$m(1,2) = A_1 \times A_2$

$= 5 \times 4 \times 4 \times 6$

$= 5 \times 4 \times 6$

$= 120$

STEP 3:

$m(2,3) = A_2 \times A_3$

$= 4 \times 6 \times 6 \times 2$

$= 4 \times 6 \times 2$

$= 48$

STEP 4:

$m(3,4) = A_3 \times A_4$

$= 6 \times 2 \times 2 \times 7$

$= 6 \times 2 \times 7$

$= 84$

STEP 5:

$m(1,3) = min\ [A_1 \times A_2 \times A_3]$

$= (A_1 \times A_2) \times A_3$

$= [(A_1 \times A_2) + A_3] + [(A_1 \times A_2) \times A_3]$

$= [m(1,2) + m(3,3)] + [m(1,2)\ A_3]$

$= [(5 \times 4 \times 4 \times 6) + 0] + [\mathbf{(5 \times 4 \times 4 \times 6) \times (6 \times 2)}]$

[consider only bold elements because of m(1,3) that is, we will only take

elements of A_1 and A_3]

= [(5 × 4 × 6) + 0] + [5 × 6 × 2]

= 120 + 60

= 180

ii. *= A_1 × (A_2 × A_3)*

= [A_1 + (A_2 × A_3)] + [A_1 × (A_2 × A_3)]

= [m(1,1) + m(2,3)] + [A_1 m(2,3)]

= [0 + (4 × 6 × 6 × 2)] + [(5 × 4) × (4 × 6 × 6 × 2)]

[consider only bold elements because of m(1,3) that is, we will only take elements of A_1 and A_3]

= [(4 × 6 × 2) + 0] + [5 × 4 × 2]

= 48 + 40

= 88

Now, m(1,3) = min [A_1 × A_2 × A_3] = min[180,88] = 88.

STEP 6:

m(2,4) = min [A_2 × A_3 × A_4]

= (A_2 × A_3) × A_4

= [(A_2 × A_3) + A_4] + [(A_2 × A_3) × A_4]

= [m(2,3) + m(4,4)] + [m(2,3) A_4]

= [(4 × 6 × 6 × 2) + 0] + [(4 × 6 × 6 × 2) × (2 × 7)]

[consider only bold elements because of m(2,4) that is, we will only take elements of A_2 and A_4]

= [(4 × 6 × 2) + 0] + [4 × 2 × 7]

= 48 + 56

= 104

ii. *= A_2 × (A_3 × A_4)*

= [A_2 + (A_3 × A_4)] + [A_2 × (A_3 × A_4)]

= [m(2,2) + m(3,4)] + [A_2 m(3,4)]

= [0 + (6 × 2 × 2 × 7)] + [(4 × 6) × (6 × 2 × 2 × 7)]

[consider only bold elements because of m(2,4) that is, we will only take elements of A_2 and A_4]

= [(6 × 2 × 7) + 0] + [4 × 6 × 7]

= 84 + 168

= 414

Now, m(2,3) = min [$A_2 \times A_3 \times A_4$] = min[104,414] = 104

STEP 7:

m(1,4) = min [$A_1 \times A_2 \times A_3 \times A_4$]

= ($A_1 \times A_2 \times A_3$) × A_4

= [($A_1 \times A_2 \times A_3$) + A_4] + [($A_1 \times A_2 \times A_3$) × A_4]

= [m(1,3) + m(4,4)] + [($A_1 \times A_2 \times A_3$) × A_4]

*= [(88) + 0] + [(5 × 4 * 4 × 6 * 6 × 2) × 2 × 7]*

[consider only bold elements because of m(1,4) that is, we will only take elements of A_1 and A_4]

= [88] + [5 × 2 × 7]

= 88 + 70

= 158

ii. *= $A_1 \times (A_2 \times A_3 \times A_4)$*

= [A_1 + ($A_2 \times A_3 \times A_4$)] + [$A_1 \times (A_2 \times A_3 \times A_4)$]

= [m(1,1) + m(2,4)] + [$A_1 \times (A_2 \times A_3 \times A_4)$]

*= [0 + (104)] + [(5 × 4 * 4 × 6 * 6 × 2)] × 2 × 7]*

[consider only bold elements because of m(2,4) i.e we will only take elements of A_2 and A_4]

= [104] + [5 × 4 × 7]

= 104 + 140

= 244

iii. *= ($A_1 \times A_2$) × ($A_3 \times A_4$)*

$= [(A_1 \times A_2) \times (A_3 \times A_4)] + [(A_1 \times A_2) \times (A_3 \times A_4)]$

$= [m(1,2) + m(3,4)] + [(A_1 \times A_2) \times (A_3 \times A_4)]$

$= [(5 \times 4 \times 4 \times 6) + (6 \times 2 \times 2 \times 7)] + [(5 \times 4 * 4 \times 6) \times (6 \times 2 * 2 \times 7)]$

[consider only bold elements because of m(1,4) i.e we will only take elements of A_2 and A_4 and the common element between the two groups.]

$= [(5 \times 4 \times 6) + (6 \times 2 \times 7)] + [(5 \times 6 \times 7)]$

$= 120 + 84 + 210$

$= 414$

Now, $m(1,4) = \min [A_1 \times A_2 \times A_3 \times A_4] = \min[158, 244, 414] = 158$

Now, we fill the values in the tabular columns:

- We have obtained values for the 'm' table.
- For 's' table, we will see what the first element of the equation by which minimum value was obtained, was. We also have to consider the highest element in the bracket.

Refer to *Figure 5.12*:

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Figure 5.12: *Matrix Chain Multiplication example*

$\therefore$ *Result* $= [A_1 \times (A_2 \times A_3)] \times A_4$ *…. [By backtracking]*

[We used the equation $\{A_1 \times (A_2 \times A_3)\}$ in $\{(A_1 \times A_2 \times A_3) \times A_4\}$, in order to obtain the minimum value, that is, 158 for m(1,4)].

Refer to the following *Figure 5.13*:

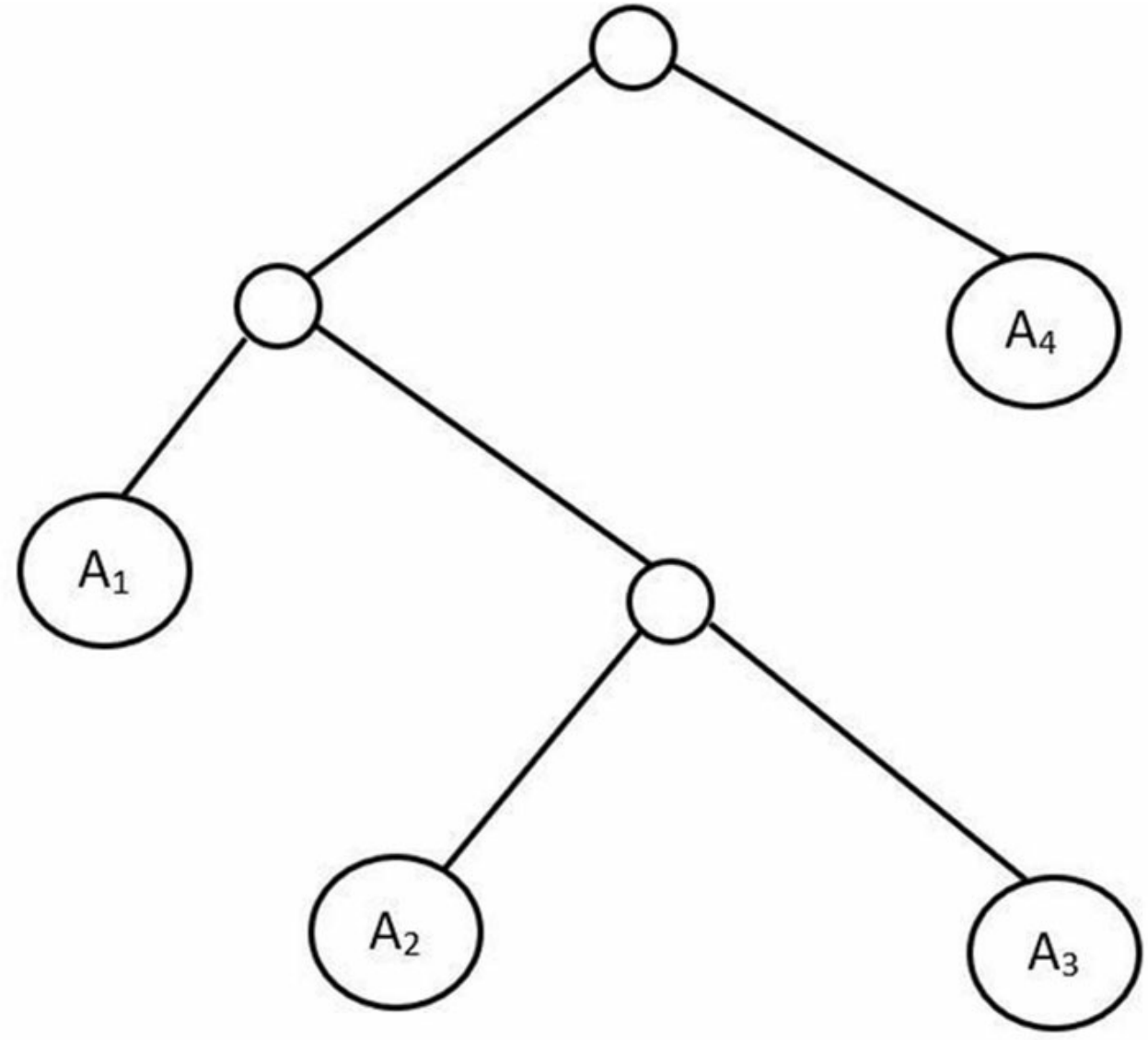


Figure 5.13: *Matrix Chain Multiplication optimal result*

Flow Shop scheduling

Flow Shop Scheduling problems are a collection of scheduling problems with a workshop or a group shop. If there are several machines and there are a number of jobs, then all of them must be executed by at least one system or machine, that is, implementation of all jobs must be computed. Flow shop scheduling is an exception to job scheduling, in which there are strict rules that the function should be performed on all jobs. Some methods to solve flow shop scheduling problems are Branch and Bound, Dynamic Programming, Heuristicz algorithm and Meta-heuristics.

In a Flow Shop Scheduling, all jobs must be executed on a set of systems in identical order. The aim is to identify the job sequence for optimization of certain objective functions. Flow shop scheduling problems broadly exist in the production of industries and mechanical developments.

Algorithm

Let us now go over the algorithm.

Algorithm Flow Shop Scheduling (P, Q)

Let us consider 'P' as an array to represent time of jobs, where each column denotes time period on machine, Mi.

Let us consider 'Q' as a job queue.

```
Q = Φ
for j = 1 to n do
  p = minimum machine time scanning in both columns
  if p occurs in column 1 then
    Add a Job j to the first empty slot of Q
  else
    Add a Job j to the last empty slot of Q
  end
  Remove processed job from consideration
end
return Q
```

Optimal Binary Search Tree

A tree in which the values are kept in the internal nodes is known as a **Binary Search Tree** (**BST**). Null nodes make up the exterior nodes. All the nodes in a binary search tree's left subtree have values that are lower than the root nodes. In accordance with this, every node in the right subtree has a value that is larger than or equal to the root node.

An optimal binary search tree is a BST which consists of the minimum possible cost of locating all nodes. Time complexity of BST is O(n), whereas time complexity for Balanced-BST is O(log n). The time complexity for Optimal Binary Search Tree can be improved by locating the most frequently used value near to the root element.

Algorithm

Let us now go over the algorithm.

Algorithm Optimal Binary Search Tree (x, y, n)

Let us consider :

```
o[1…n+1, 0…n ] as Optimal subtree,
p[1…n+1, 0…n] as Sum of probability,
root[1…n, 1…n] as To construct OBST.
for i = 1 to n + 1 do
o[i, i – 1] = yi – 1;
p[i, i – 1] = yi – 1;
end
for m = 1 to n do
for i = 1 to n – m + 1 do
j = i + m – 1 ;
o[i, j] = ∞ ;
p[i, j] = p[i, j – 1] + xj + yj;
for r = i to j do
t = o[i, r – 1] + o[r + 1, j] + p[i, j] ;
if t < o[i, j] then
o[i, j] = t ;
    root[i, j] = r ;
    end
   end
  end
end
return (o, root)
```

Example 10

Find the optimal Binary search tree for the below given input keys= {10, 12}, frequency = {34, 50}

There can be following two possible BSTs:

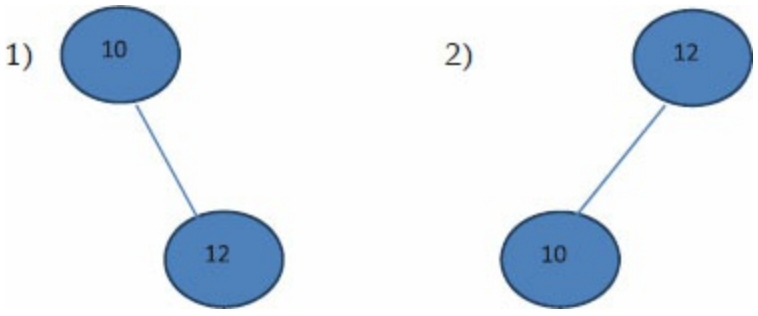


Figure 5.14: *Possible Binary Search Tree*

The frequency of 10 and 12 are 34 and 50 respectively,

For tree 1, cost = 34*1 + 50*2 = 134

For tree 2, cost = 50*1 + 34*2 =118

Hence, the cost of tree 2 is minimal.

Example 11

Find the optimal binary tree for the given input

Input: keys= {5, 15, 25}, frequency = {30, 10, 55}

There can be following possible BSTs

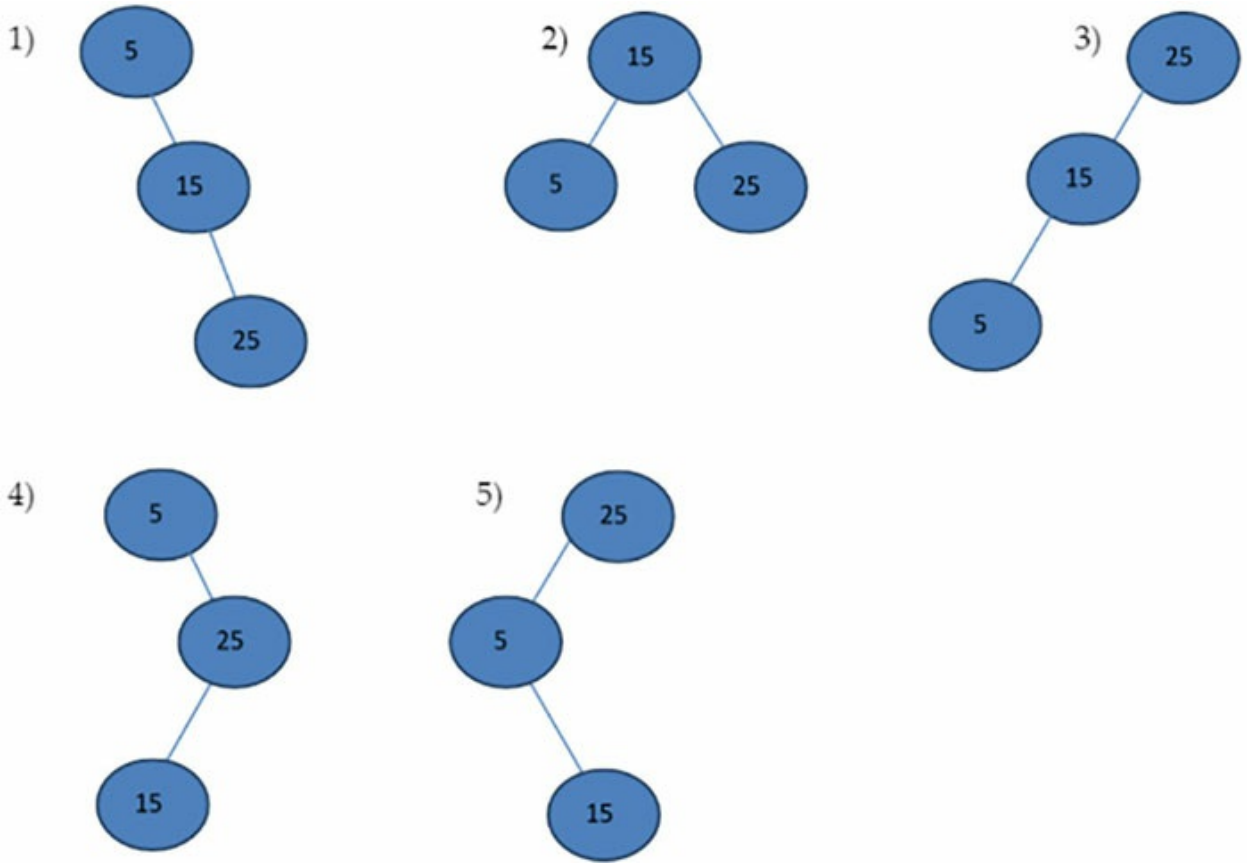


Figure 5.15: *Possible Binary Search Tree*

*For tree 1, cost = 30*1 + 10*2 + 55*3 = 215*

*For tree 2, cost = 10*1 + 30*2 + 55*3 = 235*

*For tree 3, cost = 55*1 + 10*2 + 30*3 = 165*

*For tree 4, cost = 30*1 + 55*2 + 10*3 = 170*

*For tree 5, cost = 55*1 + 30*2 + 10*3 = 145*

Hence, the cost of tree 5 is minimal.

NP – HARD & NP – COMPLETE

NP (**nondeterministic polynomial time**) is a complexity class used to classify decision problems in computational complexity theory. The group of decision problems known as NP are those for which, cases of the problem where the response is "yes", contain proofs that can be checked in polynomial time by a deterministic Turing machine.

A non-deterministic Turing machine may solve a collection of decision problems in polynomial time, which is an analogous definition of NP. The acronym **NP**, which stands for "**nondeterministic polynomial time**," is based on this notion. Because the algorithm based on the Turing machine has two phases, the first of which consists of a non-deterministic guess about the solution, and the second of which consists of a deterministic algorithm that verifies whether the guess is a solution to the problem, these two definitions are equivalent. Based on the fastest known methods, decision problems are categorized into complexity classes (such as NP). Therefore, if faster algorithms are developed, decision issues may be categorized differently.

A problem is in the class NPC if it is in NP. Even though a problem is not NP itself, it is NP-Hard, if all NP problems can be reduced to it in polynomial time. Refer to *Figure 5.14*:

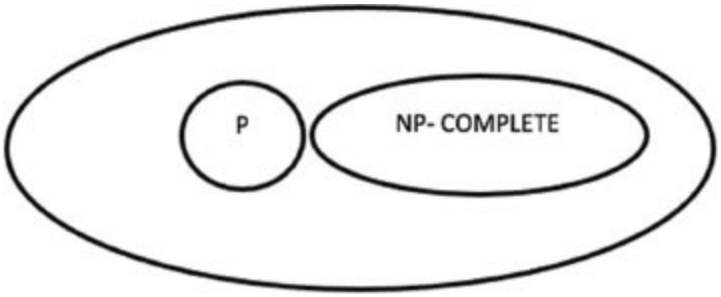


Figure 5.16: *Relationship*

All problems in NP would be polynomial time solvable if any polynomial time exists for any of these problems. These problems are referred to as NP-Complete. Both theoretically and practically, NP-Completeness is an important phenomenon.

If a language B meets the following two requirements:

- B is in NP.
- Every A in NP is polynomial time reducible to B.

Language B is referred to as NP-Hard if it meets the second property but not necessarily the first. If there is some NP-Complete problem, then a search problem B is informally said to be NP-Hard. It is impossible to solve the NP-Hard problem in the P = NP polynomial time unit.

NP-COMPLETE problems

The NP-Complete problems listed as follows, have no known polynomial time algorithm:

- Determine whether a graph has a Hamiltonian cycle.
- Determine whether a Boolean formula is satisfied, and so on.

NP-HARD problems

Following are NP-Hard problems:

- The circuit – satisfiability problem
- Set cover
- Vertex cover
- Traveling salesman problem

Conclusion

By breaking the problem down into smaller and smaller potential sub-problems, the approach of dynamic programming is comparable to the strategy of divide and conquer. Contrary to divide and conquer, these subproblems cannot be resolved separately. Instead, solutions to these more manageable sub-problems are retained and applied to subsequent or related sub-problems. When we have difficulties that can be broken down into smaller, related problems so that the solutions can be reused, we employ dynamic programming. These algorithms are typically employed for optimization. The dynamic algorithm will attempt to review the outcomes of the previously solved sub-problems before solving the current sub-problem. To find the best solution, the subproblems' solutions are combined. Dynamic

algorithms are driven by a problem-wide optimization, in contrast to greedy algorithms that focus on local optimization. The shortest path algorithm uses the previous found data and finds the shortest path between the two pairs. 0/1 knapsack problem finds the optimal solution of the weights that leads to maximum capacity. Traveling salesman problem algorithm's main aim is to find the minimum cost path between the source and destination. Coin Changing Problem is to find the minimum number of ways of making changes, using a given set of coins for the amount. In a Flow Shop Scheduling, all jobs must be executed on a set of systems in identical order. A BST with the lowest cost of finding every node is an ideal binary search tree.

Key facts

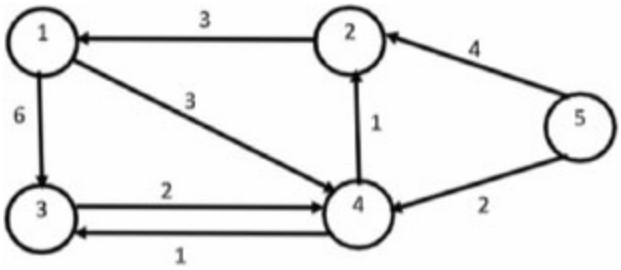
- **Dynamic Programming**: It is a method for solving difficult problems by first decomposing them into a number of simpler subproblems, solving each subproblem only once, and then storing the answers to prevent having to perform the same calculations repeatedly.
- **All Pair Shortest Path Algorithm**: It attempts to determine the shortest route between every pair of vertices. As we need one spanning tree for each vertex, storing all the paths can be highly memory expensive. These are typically referred to as all pairs-shortest distance problems, since they only seek to determine the distance between each node to another.
- **0/1 Knapsack**: The 0/1 knapsack problem means that the items are either completely filling the knapsack, or no items are filled in the knapsack. No fractional values are considered in this method.
- **Traveling salesman problem**: Issues with travelling salesmen are restricted to a salesman and a group of cities. Beginning in one city, the salesman must travel to all of them before returning to that same city. The difficulty with the situation is that the travelling salesman must reduce the length of the trip.
- **Coin Changing Problem**: If coin N is supplied, different coins are included in the array. The objective is to modify N using the array's coins. Make a change so that a minimal number of coins are utilized.
- **Matrix Chain Multiplication**: It is a Dynamic Programming technique

where the previous output serves as the following step's input. Chain in this case denotes that the column of one matrix equals the row of another matrix.

- **Flow shop scheduling**: A group of scheduling issues with a workshop or group of shops, is referred to as Flowshop scheduling problem. If there are several machines and numerous jobs, then each one of them must be carried out by at least one system or machine.
- **Optimal Binary Search Tree**: The idea of a binary search tree is expanded upon in an optimal binary search tree. The frequency and key value of a node's key value determine the cost of finding that node in a tree, which is a very important factor. Often, we want to lower the cost of searching, and an optimum binary tree helps us do so.
- **NP – HARD & NP – COMPLETE**: If a problem is in NP, it belongs to the class NPC. A problem is NP-Hard even though it is not NP in and of itself, if all NP problems can be reduced to it in polynomial time.
- If there is any polynomial time for any of these problems, then all NP problems would be solvable in polynomial time. These issues are known as NP-Complete.

Questions

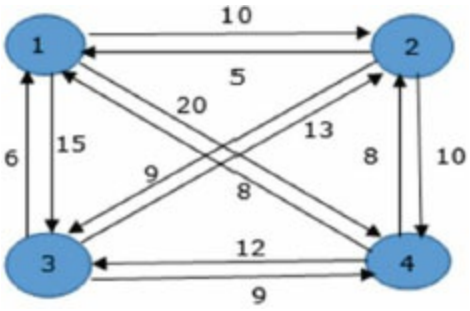
1. Explain Flow Shop Scheduling technique. **[MU DEC'18 (5 MARKS)]**
2. Apply All Pair Shortest Path on the following graph: **[MU DEC'18 (10 MARKS)]**



3. Given a chain of four matrices A_1, A_2, A_3, A_4 with $P_0 = 5$, $P_1 = 4$, $P_2 = 6$, $P_3 = 2$, $P_4 = 7$. Find m[1,4] using matrix chain multiplication. **[MU DEC'18 (10 MARKS)]**
4. Compare Greedy and Dynamic programming approaches for an

algorithm design. Explain how both can be used to solve knapsack problems. **[MU DEC'18 (10 MARKS)]**

5. Differentiate between greedy methods and dynamic programming? **[MU MAY'19 (5 MARKS)] [MU DEC'19 (5 MARKS)]**
6. A traveler needs to visit all the cities from a list (figure) where distances between all the cities are known and each city should be visited just once. What is the shortest possible route that he visits each city exactly once and returns to the origin city? **[MU MAY'19 (10 MARKS)]**
7. What is an Optimal Binary Search Tree? Explain with the help of an example. **[MU MAY'19 (10 MARKS)]**



8. Explain Coin Chaining Problem. **[MU DEC'19 (5 MARKS)]**
9. Explain Flow shop Scheduling Technique. **[MU DEC'19 (5 MARKS)]**
10. Explain Traveling Salesman Problem with an example. **[MU DEC'19 (10 MARKS)]**
11. Explain the knapsack problem with an example. **[MU DEC'19 (10 MARKS)]**
12. Write a short note on Optimal Binary Search Tree. **[MU DEC'19 (5 MARKS)]**

CHAPTER 6

String Matching

Introduction

String matching operation is one of the basic parts in many text processing applications. The main aim of string-matching algorithm is to find the pattern P from the given text T. Mostly, the length of the pattern P is much smaller than the length of the text T, that is, P << T. String matching problem is defined as “given some text or some string T[1….n] of size n, find all occurrences of pattern P[1….m] of size.

There are a few algorithms such as naïve string-matching algorithms, **Knuth–Morris–Pratt**(**KMP**) Algorithm, Rabin-Karp algorithm, and so on, which will be discussed with problems clearly.

Structure

In this chapter, we will discuss the following topics:

- The Naïve String-Matching Algorithm
- Rabin Karp Algorithm
- The Knuth-Morris-Pratt Algorithm
- Longest Common Subsequence
- Genetic Algorithms

Objectives

By the end of this chapter, the reader will learn about various string-matching algorithms with suitable examples. Readers will also learn about genetic algorithms.

The Naïve String-Matching Algorithm

In this algorithm, it first compares the first character of pattern with searchable text. If a match is not found, pointers of both the strings are moved forward. If a match is found, the pointer of text is incremented and the pointer of pattern is reset. This process is repeated till the end of the text. Naïve approach never requires any preprocessing. It starts directly comparing the pattern P and text T, character by character. After each comparison, the pointer shifts the pattern string one position to the right.

Example 1

Perform pattern matching using the naïve algorithm.

String: a b c d a d c a b c d f

Pattern: a b c d f

Solution: t_i and p_j are indices of text and pattern respectively.

Step 1: T[1] =P[1], t_i ++, p_j++

Refer to *Figure 6.1*:

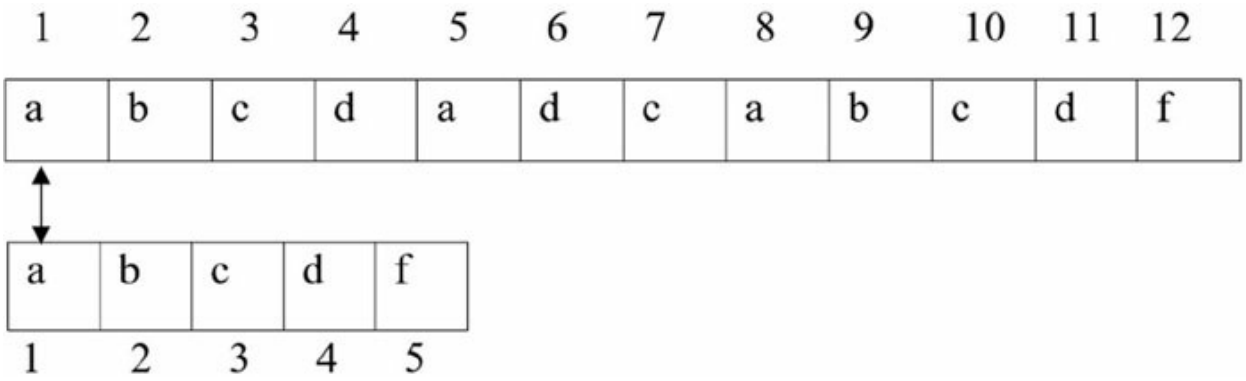


Figure 6.1: *T[1] compared with P[1]*

Step 2: T[2]=p[2], t_i ++, p_j++

Refer to *Figure 6.2*:

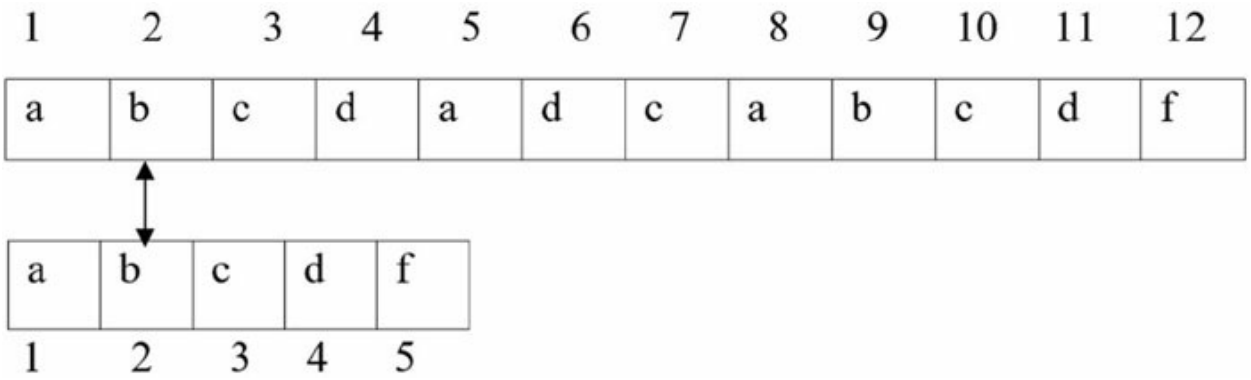


Figure 6.2: *T[2] compared with P[1]*

Step 3: T[3]= P[3], t_i ++, p_j++

Refer to *Figure 6.3*:

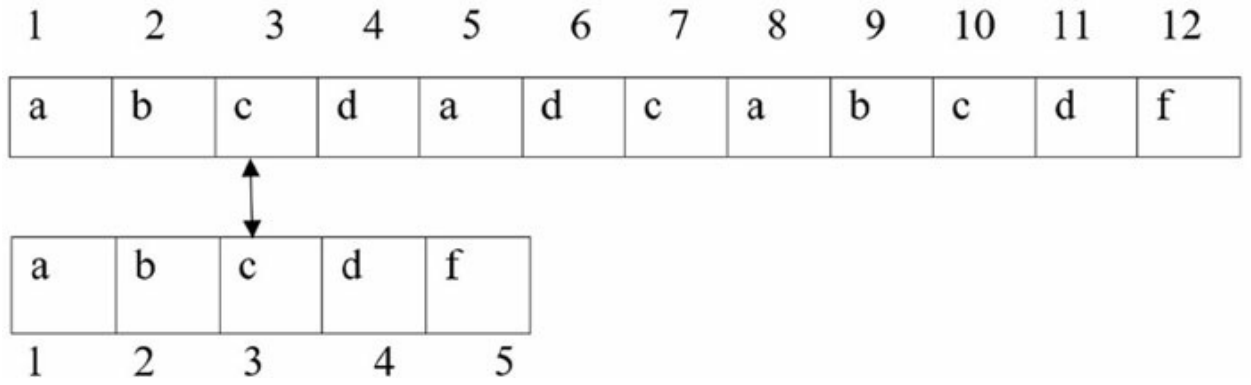


Figure 6.3: *T[3] compared with P[3]*

Step 4: T[4]= P[4], t_i ++, p_j++

Refer to *Figure 6.4*:

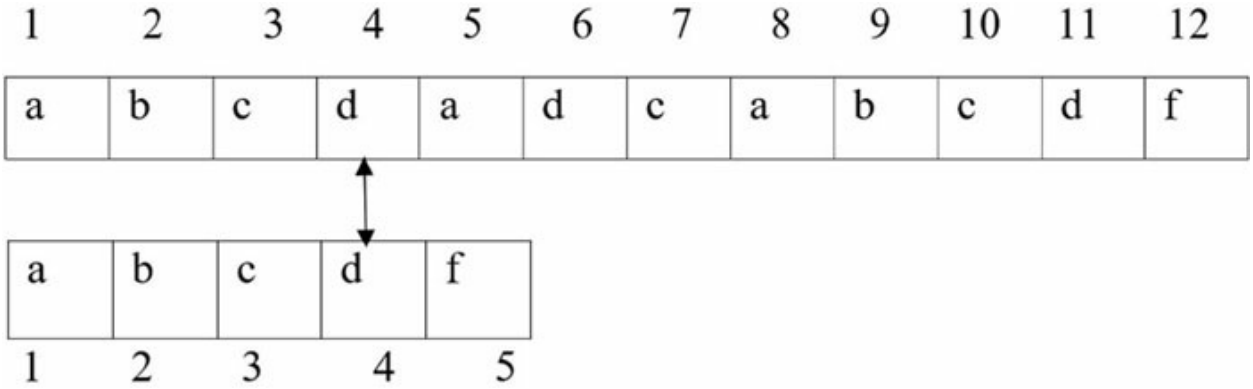


Figure 6.4: *T[4] compared with P[4]*

Step 5: T[5] ≠ P[5], t_i ++, p_j=1

Refer to *Figure 6.5*:

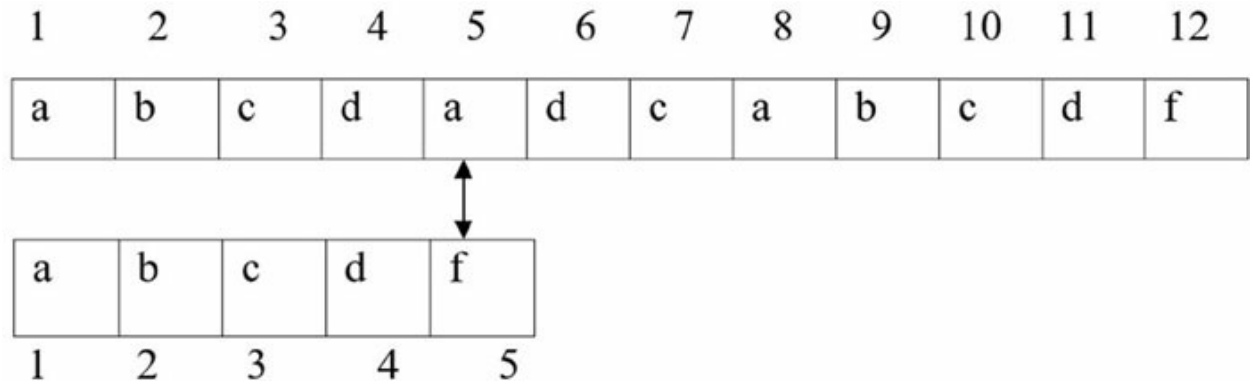


Figure 6.5: *T[5] compared with P[5]*

Step 6: T[2] ≠ P[1], t_i ++

Refer to *Figure 6.6*:

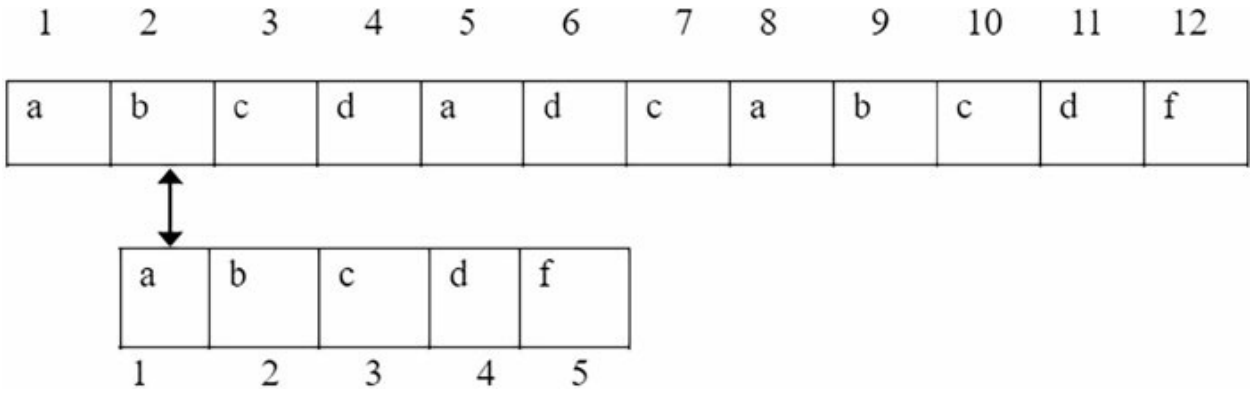


Figure 6.6: *T[2] compared with P[1]*

Step 7: T[3] ≠ P[1], t_i ++

Refer to *Figure 6.7*:

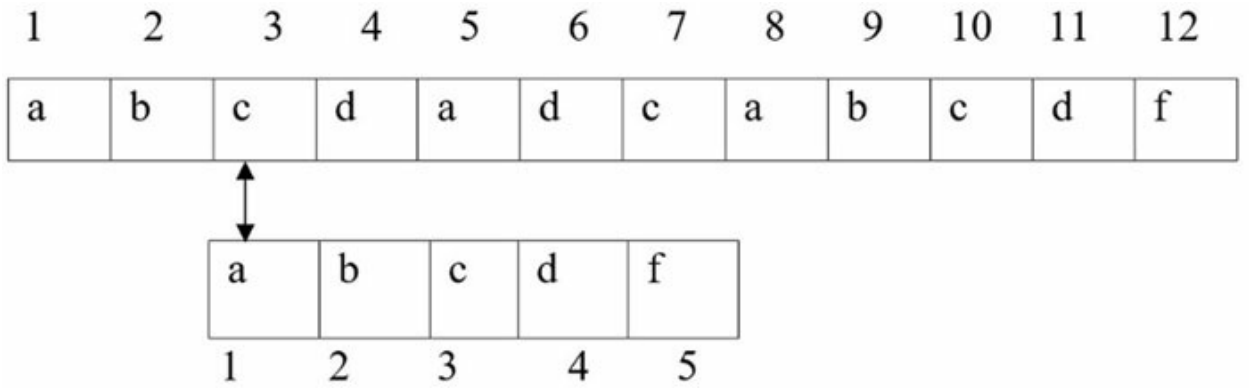


Figure 6.7: *T[3] compared with P[1]*

Step 8: T[4] ≠ P[1], t_i ++

Refer to *Figure 6.8*:

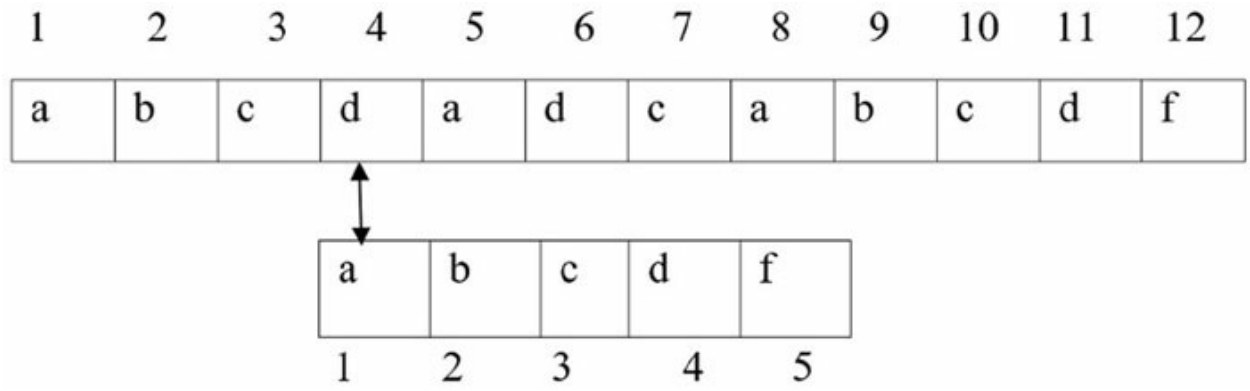


Figure 6.8: *T[4] compared with P[1]*

Step 9: T[5] = P[1], t_i ++, p_j++

Refer to *Figure 6.9*:

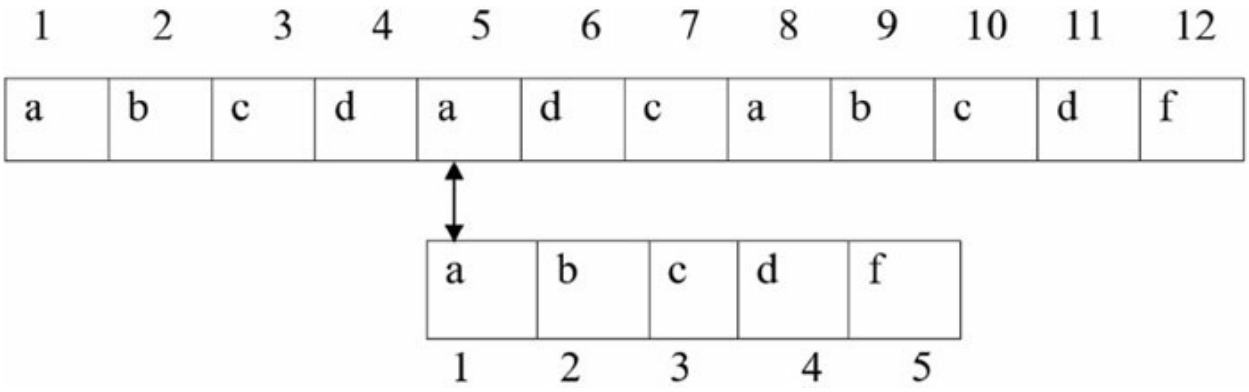


Figure 6.9: *T[5] compared with P[1]*

Step 10: T[6] ≠ P[2], t_i ++

Refer to *Figure 6.10*:

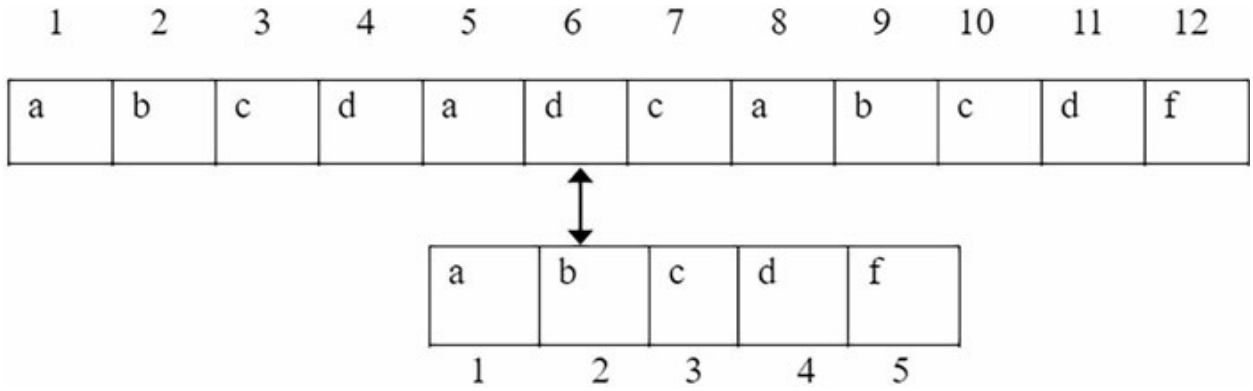


Figure 6.10: *T[6] compared with P[2]*

Step 11: T[6] ≠ P[1], t_i ++

Refer to *Figure 6.11*:

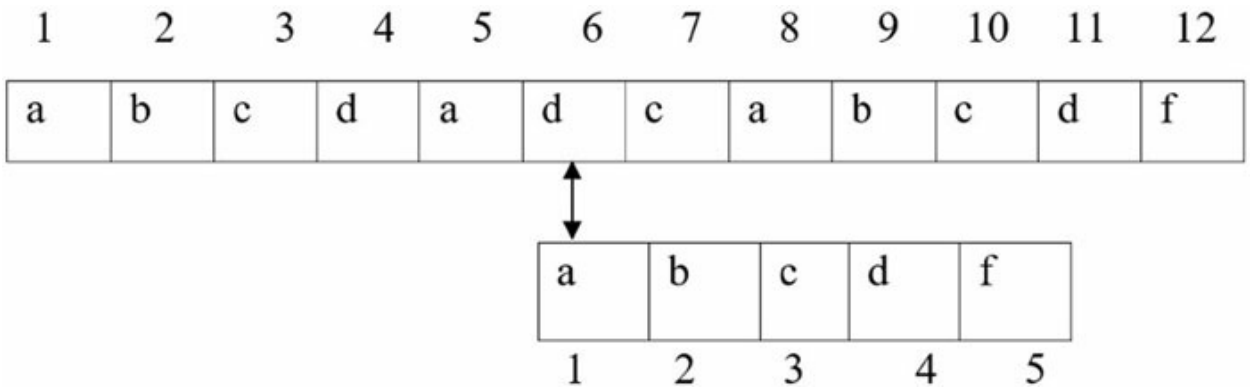


Figure 6.11: *T[6] compared with P[1]*

Step 12: T[7]≠ P[1], t_i ++

Refer to *Figure 6.12*:

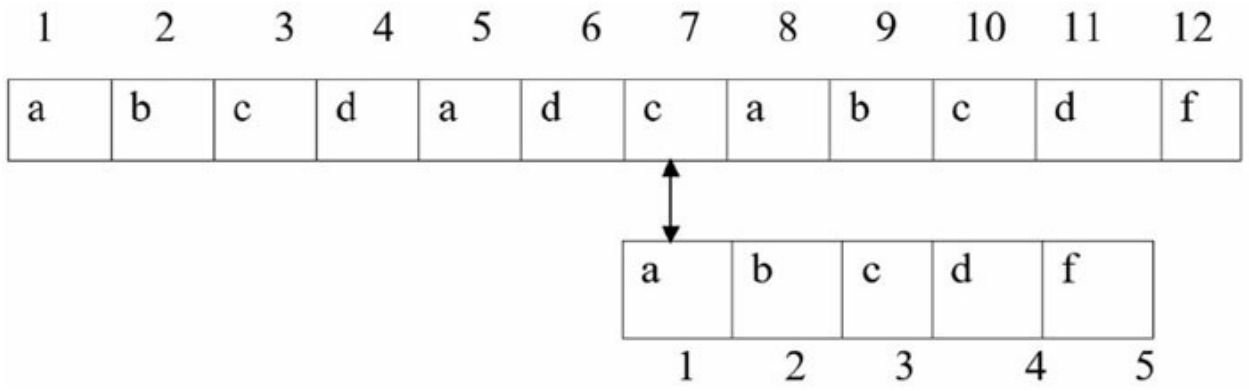


Figure 6.12: *T[7] compared with P[1]*

Step 13: T[8] =P[1], t_i ++, p_j++

Refer to *Figure 6.13*:

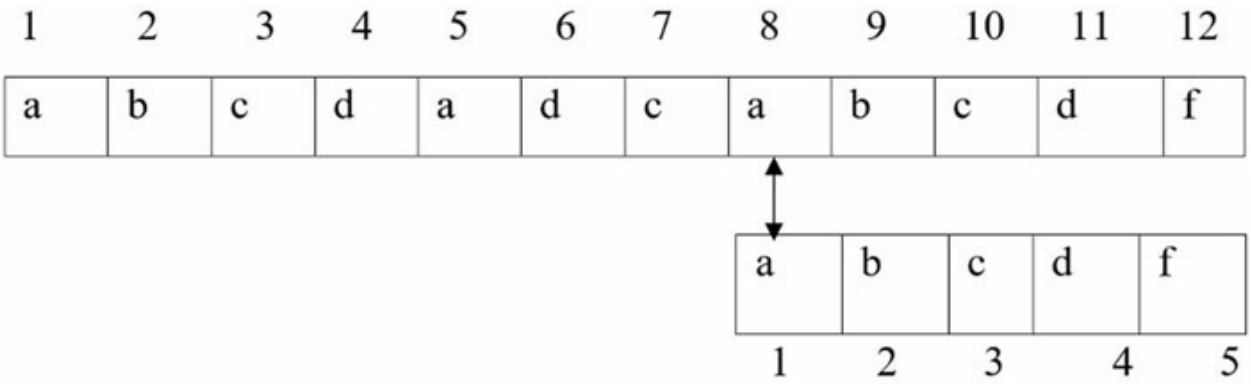


Figure 6.13: *T[8] compared with P[1]*

Step 14: T[9]=P[2], t_i ++, p_j++

Refer to *Figure 6.14*:

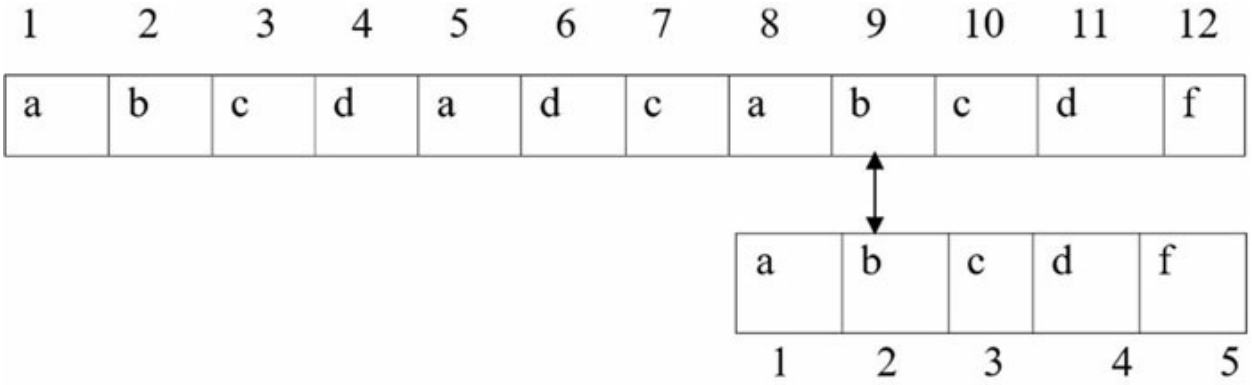


Figure 6.14: *T[9] compared with P[2]*

Step 15: T[10]=P[3], t_i ++, p_j++

Refer to *Figure 6.15*:

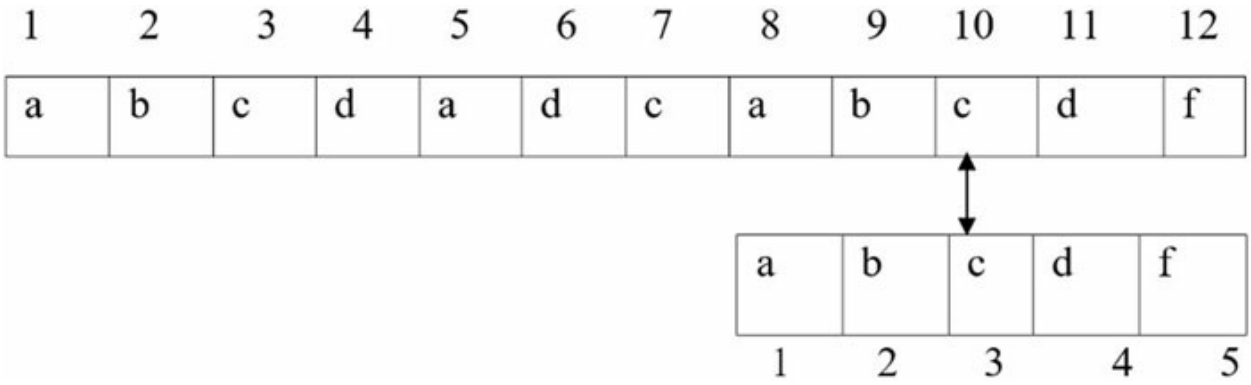


Figure 6.15: *T[10] compared with P[4]*

Step 16: T[11]=P[4], t_i ++, p_j++

Refer to *Figure 6.16*:

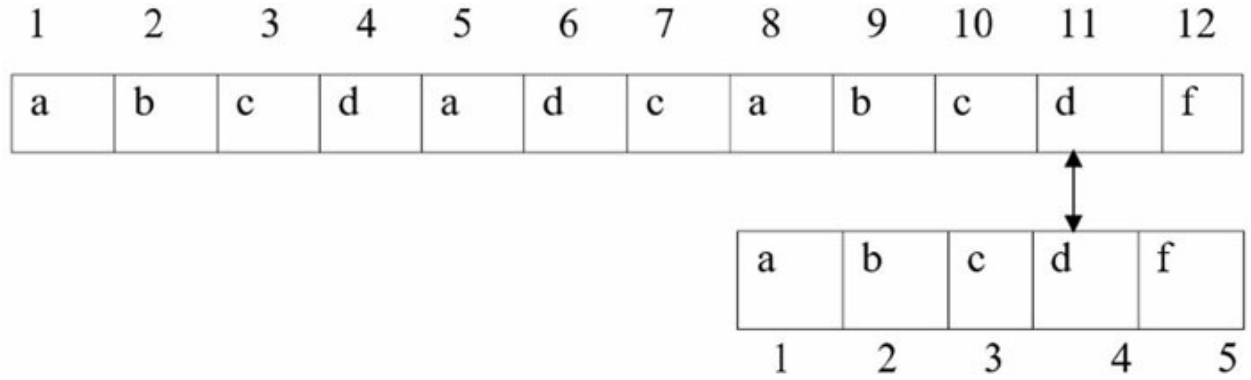


Figure 6.16: *T[4] compared with P[4]*

Step 17: T[12]=P[5], t_i ++, p_j++

Refer to *Figure 6.17*:

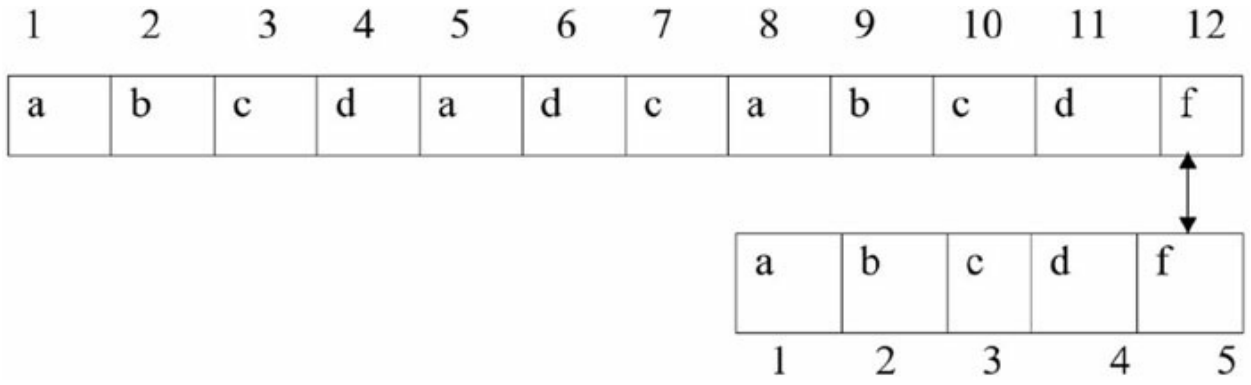


Figure 6.17: *T[12] compared with P[5]*

Thus, the string is matched.

Algorithm for Naïve bayes string matching

The algorithm for Naïve bays string matching is as follows:

```
NAÏVE_STRING_MATCH(T,P)
for i=0 to i=n-m do
   if P[1…m] == T[i+1…i+m]then
     print “Match found”
   end
end
```

The best case of is O(n) and the worst case is O(m*(n-m+1)).

Rabin Karp Algorithm

One of the string-matching algorithms is the Rabin-Karp algorithm. However, instead of matching every element of the string, it uses numbers, that is, the hash value to match the string. If the hash value is matched, then only it matches each character of that string, one by one. Hence, because of these methods, it is one of the most efficient methods of string matching, having the time complexity of O(n-m+1), but for worst case, it is O(mn).

Suppose, A represents values of the pattern A[1…..m] of length m, and T be the values of the substring.

The Rabin Karp algorithm first calculates the hash value of A and T. If the hash value is the same, that is, hash(A)=hash(T), then we match that part of the string like the naive algorithm.

On hash match, actual characters of both strings are compared using brute force approach. If pattern is found, then it is called hit else spurious hit.

Example 2

Compute the pattern matching using Rabin Karp algorithm for the given data:

Text= CCACCAAEDBA

Pattern= DBA

Solution

Let A=1; B=2; C=3; D=4; E=5; F=6; G=7; H=8; I=9; J=10

Step 1: Calculates the hash value of the pattern

```
Hash code h(P)=value (D)+value (B)+value (A)=4+2+1 = 7
```

Step 2: Calculate the hash value of the substring

CCA = 3+3+1 = 7 = h(p) (however the string does not match)

CAC = 3+1 +3 = 7 = h(p) (however the string does not match)

ACC = 1 +3+3 = 7 = h(p) (however the string does not match)

CCA = 3+3+1 = 7 = h(p) (however the string does not match)

CAA = 3+1+1 = 5 ≠ h(p)

Here, as we can see that most of the hash value of the substring gives the same hash code of the pattern, such a situation is called a spurious hit. To overcome such situation, we use another method;

Step 1: Pattern = DBA = $4*10^2 + 2*10^1 + 1*10^0$

= 400 + 20 + 1 = 421 = h(p)

Step 2:

CCA = $3*10^2 + 3*10^1 + 1*10^0$ = 300 + 30 + 1 = 331 ≠ h(p)

CAC = $3*10^2 + 1*10^1 + 3*10^0$ = 300 + 10 + 3 = 313 ≠ h(p)

ACC = $1*10^2 + 3*10^1 + 3*10^0$ = 100 + 30 + 3 = 133 ≠ h(p)

CCA = $3*10^2 + 3*10^1 + 1*10^0$ = 300 + 30 + 1 = 331 ≠ h(p)

CAA = $3*10^2 + 1*10^1 + 1*10^0$ = 300 + 10 + 1 = 311 ≠ h(p)

AAE = $1*10^2 + 1*10^1 + 5*10^0$ = 100 + 10 + 5 = 115 ≠ h(p)

AED = $1*10^2 + 5*10^1 + 4*10^0$ = 100 + 50 + 4 = 154 ≠ h(p)

EDB = $5*10^2 + 4*10^1 + 2*10^0$ = 500 + 40 + 2 = 542 ≠ h(p)

DBA = $4*10^2 + 2*10^1 + 1*10^0$ = 400 + 20 + 1 = 421 = h(p)

Hence, the match is found.

Example 3

Compute the pattern matching using RabinKarp algorithm for the given data:

Text= DABCADADDIT

Pattern= ADD

Solution:

Let A=1;B=2;C=3;D=4;E=5;F=6;G=7;H=8;I=9;J=10

Step 1: Calculate the hash value of the pattern:

$h(p) = ADD = 1*10^2 + 4*10^1 + 4*10^0$

$= 100 + 40 + 4 = 144$

Step 2: Calculate the hash value of the substrings.

$DAB = 4*10^2 + 1*10^1 + 2*10^0 = 400 + 10 + 2 = 412 \neq h(p)$

$ABC = 1*10^2 + 2*10^1 + 3*10^0 = 100 + 20 + 3 = 123 \neq h(p)$

$BCA = 2*10^2 + 3*10^1 + 1*10^0 = 200 + 30 + 1 = 231 \neq h(p)$

$CAD = 3*10^2 + 1*10^1 + 4*10^0 = 300 + 10 + 4 = 314 \neq h(p)$

$ADA = 1*10^2 + 4*10^1 + 1*10^0 = 100 + 40 + 1 = 141 \neq h(p)$

$DAD = 4*10^2 + 1*10^1 + 4*10^0 = 400 + 10 + 4 = 414 \neq h(p)$

$ADD = 1*10^2 + 4*10^1 + 4*10^0 = 100 + 40 + 4 = 144 = h(p)$

Hence, the match is found.

Hence, in this way, we can easily find the matching pattern from the whole complete string easily.

The Knuth-Morris-Pratt Algorithm

The Knutt-Morris-Pratt algorithm is also called as KMP algorithm. This is the first linear time algorithm for string matching. The main task of these algorithms is to find out whether the given pattern is existing inside the string or not. However, if it is there, then find out its index.

The fundamental tenet of the KMP method is that it avoids back trashing, unlike the naive bayes technique, if the initial portion of the string appears elsewhere in the string. O(m+n) is the time complexity of the KMP algorithm. In the worst case, KMP algorithm examines all the characters of the input pattern and text exactly once. This improvement is achieved by using the auxiliary function π or also known as LPS table.

The acronym LPS stands for Longest Proper Prefix, which is also suffix. A proper prefix is one that does not contain the entire string. For example,

prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC” and “ABC”.

Example 4

Let us assume the pattern be P_1 = ABCDABEABF.

Then the LPS table for the pattern P_1 would be as the following *Table 6.1*:

Index	1	2	3	4	5	6	7	8	9	10
pattern	A	B	C	D	A	B	E	A	B	F
LPS value	0	0	0	0	1	2	0	1	2	0

Table 6.1: *LPS table*

Since, till index [1,2,3,4] any of the character does not repeat itself, the LPS value is given as 0.However, at index [5] the character A is repeated and hence, it is given the value equal to the index of the first A, that is, 1 similarly character B at position [6] is given value as 2 and so on.

Hence, in this way we can calculate the LPS table or π table.

Example 5

Compute the π table for the following AABCADAABE

Solution:

Refer to the following *Table 6.2:*

index	1	2	3	4	5	6	7	8	9	10
Pattern	A	A	B	C	A	D	A	A	B	E
LPS value	0	1	0	0	1	0	1	2	3	0

Table 6.2: *LPS table*

Example 6

Check the pattern = ABABD in the text = ABABCABABD using the KMP algorithm.

Solution:

Step 1: Draw the LPS table for the pattern.

Refer to the following *Table 6.3*:

index	1	2	3	4	5
pattern	A	B	A	B	D
LPS value	0	0	1	2	0

Table 6.3: *Pattern Table*

Step 2: Initialize " i " and " j "

Refer to the following *Table 6.4*:

i									
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

	j					
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.4: *Initialize i and j*

Step 3: now, a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.5*:

i									
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

	j					
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.5: *Compare a[i] with a[j+1]*

Step 4:a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.6*:

	i								
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

		j				
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.6: *Compare a[i] with a[j+1]*

Step 5: a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.7:*

		i							
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

			j			
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.7: *Compare a[i] with a[j+1]*

Step 6:a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.8:*

			i						
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

				j		
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.8: *Compare a[i] with a[j+1]*

Step 7: a[i] ≠ a[j+1] ; move j to index pointed by the position j is now present

that is, move j to index 2.

Refer to *Table 6.9*:

				i					
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

					j	
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.9: *Compare a[i] with a[j+1]*

Step 8: a[i] ≠ a[j+1] ; move j to index pointed by the position j is now present that is move j to index 0.

Refer to *Table 6.10*:

				i					
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

			j			
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.10: *Compare a[i] with a[j+1]*

Step 9:a[i] ≠ a[j+1] ;but j is already at a[0] therefore increment i.

Refer to *Table 6.11*:

				i					
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

	j					
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.11: *Compare a[i] with a[j+1]*

Step 10: a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.12*:

					i				
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

	j					
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.12: *Compare a[i] with a[j+1]*

Step 11: a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.13*:

						i			
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

		j				
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.13: *Compare a[i] with a[j+1]*

Step 12:a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.14*:

							i		
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

			j			
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.14: *Compare a[i] with a[j+1]*

Step 13:a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.15*:

								i	
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

				j		
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.15: *Compare a[i] with a[j+1]*

Step 14: a[i] == a[j+1] ;i++ ; j++

Refer to *Table 6.16*:

									i
1	2	3	4	5	6	7	8	9	10
A	B	A	B	C	A	B	A	B	D

					j	
index	0	1	2	3	4	5
pattern		A	B	A	B	D
LPS value		0	0	1	2	0

Table 6.16: *Compare a[i] with a[j+1]*

Hence, in this way we can match the string using KMP algorithm.

Longest Common Subsequence (LCS)

LCS problems consist of two sequences of which, we need to find out the common longest subsequent present among them.

Though dynamic programming method in LCS follows a bottom-up approach, the matrix table gets filled from top-bottom. It is an approach followed to solve the problem. In the matrix table, rows are equal to the number of alphabets present in String 1 and columns are equal to the number of alphabets in String 2. To start with the algorithm, let us initialize the first row and first column of the matrix (that is, index 0 of the matrix) values as zero. Next, compare between the elements of string 1 and the elements of string 2.If there is a match, then add value 1 with the value of the previous diagonal element. If no match exists, then consider the maximum value out of either previous row element or column element. The process continues till all the columns and rows are filled. To find the longest common subsequence, start tracing back from the longest value towards diagonal element and thus the results are obtained. The time complexity of LCS using dynamic programming approach is defined as O(m x n), where ‘m’ denotes number of alphabets present in String 1, and ‘n’ denotes as number of alphabets in String 2.

Let us consider an example for better understanding.

Example 7

Find the LCS of the strings given as follows:

String 1: stone

String 2: longest

Solution:

Refer to *Table 6.17:*

		0	1	2	3	4	5	6	7
			l	o	n	g	e	s	t
0		0	0	0	0	0	0	0	0
1	s	0	0	0	0	0	0	1	1
2	t	0	**0**	0	0	0	0	1	2
3	o	0	0	**1**	1	1	1	1	2
4	n	0	0	1	**2**	**2**	2	2	2
5	e	0	0	1	2	2	**3**	3	3
				o	n		e		

Table 6.17: *LCS table*

Therefore, LCS = one

Example 8

Find LCS of the strings given as follows:

String 1: mlnom

String 2: mnom

Refer to *Table 6.18*:

		0	1	2	3	4	5
			m	l	n	o	m
0		**0**	0	0	0	0	0
1	m	0	**1**	**1**	1	1	1
2	n	0	1	1	**2**	2	2
3	o	0	1	1	2	**3**	3
4	m	0	2	2	2	3	**4**
			m		n	o	m

Table 6.18: *LCS table*

Therefore, LCS = mnom

Genetic Algorithms

Most evolutionary algorithms are adaptive heuristic search algorithms, known as **Genetic Algorithms** (**GA**). It is predicated on the concepts of

natural selection and genetics. They are frequently employed to produce excellent solutions for optimization and search-related issues.

Genetic algorithms are nothing but the natural selection of the species which can survive and adapt to the changing environment and could reproduce and go to the next generation. In other words, it can be termed as "survival of the fittest" among individuals of the consecutive generations, of the problem to be solved.

The following analogy serves as the foundation for genetic algorithms:

1. Population members fight for resources and mates.
2. The successful (fittest) individuals then mate to have more children, than other individuals.
3. Genes from the "fittest" parent spread throughout the generation, which means that occasionally parents produce offspring who are superior to both parents.
4. As a result, each next generation becomes more environment-friendly.

Search Space

The population of the individual is preserved in the search space. Each person will be represented by a vector of components with a finite length. These elements will resemble genes in certain ways. As a result, different or various genes make up each chromosome. Refer to *Figure 6.18*:

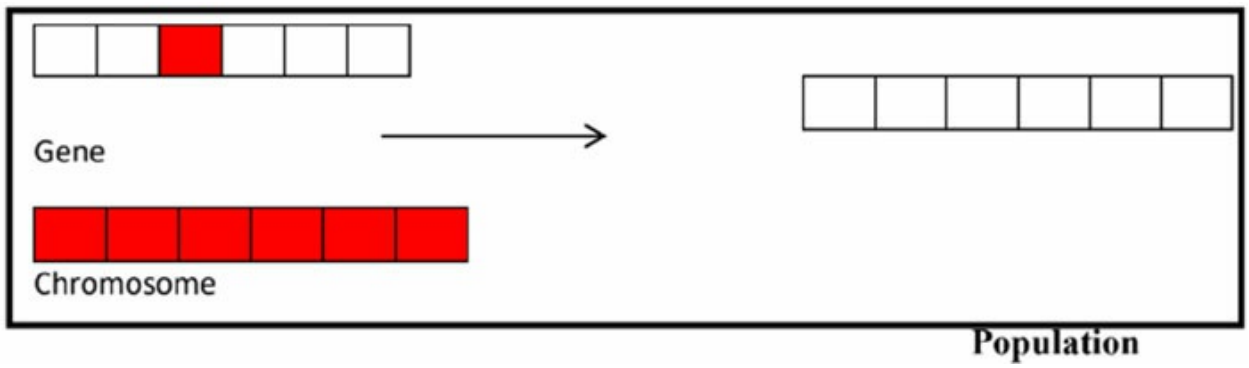


Figure 6.18: *Genetic Algorithm*

Fitness score

Each person receives a fitness score that indicates how well-equipped they

are, to compete in the changing environment. The GAs maintains the population, along with the fitness score of each individual. The individual having higher fitness score than others will be given more chance to reproduce, as compared to others. The candidate with the highest fitness score is chosen because they can unite with their mate and combine their chromosomes to make superior children. Since the population is static, space must be made for newcomers. As a result, some people pass away and are replaced by newcomers, eventually giving rise to a new generation, once the previous population had all its chances to mate. Better genes were passed down from members of the previous generation to each subsequent generation. As a result, younger generations always have more effective **"partial solutions"** than earlier ones. The population has converged once the progeny produced show no discernible difference from progeny from earlier populations. It is said that the algorithm has "converged" on a set of possible solutions to the issue.

Operators of genetic algorithms

Let us now see the different operators of genetic algorithms:

- **Selection Operator:** The goal is to favor the individual who scored higher on the fitness scale, allowing them to procreate and pass on their genes to the following generation.
- **Crossover Operator:** This is the union of the chosen operators. By using a selection operator, two operators are picked, and the crossover sites are then determined at random. Then, the genes are switched, producing an entirely new individual.

  Figure 6.19 Features the Crossover operation:

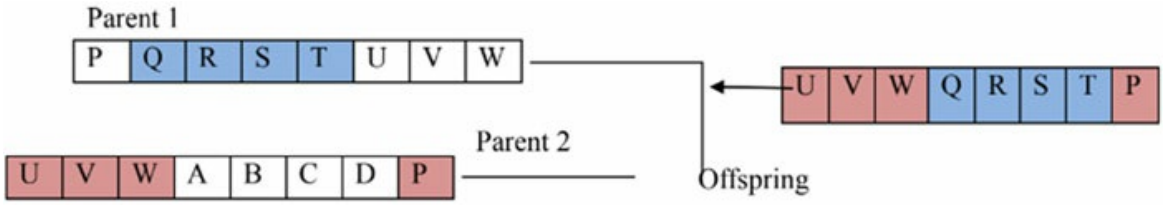


Figure 6.19: *Crossover operation*

- **Mutation Operator:** The goal is to maintain genetic diversity by randomly introducing genes into offspring. By doing so, early convergence can be avoided, as shown in the following *Figure 6.20*:

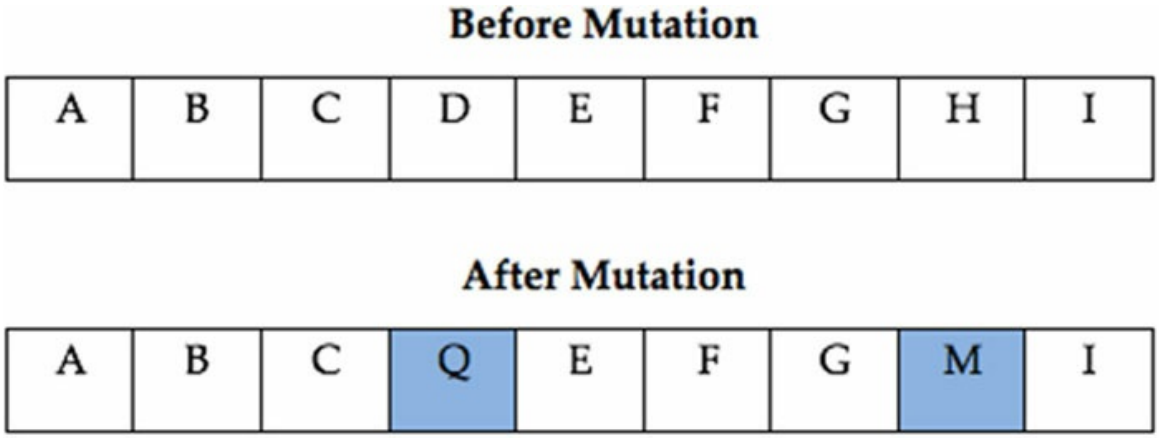


Figure 6.20: *Mutation*

Conclusion

From the preceding algorithms, we can say that the Rabin-Karp algorithm and KMP algorithm are the effective ones as compared to others, when the patterns are small, and the alphabets are small. The naïve algorithm is the solution with the slowest execution time.

Key Facts

- The string-matching algorithms are used for finding the occurrence of a pattern string within a larger text string. These algorithms can be used for a variety of applications, including text processing, data compression, and pattern recognition.
- The naive string-matching algorithm is a simple algorithm that has a time complexity of O(n * m). It is suitable for small to medium-sized strings, but it can be slow for larger strings.
- The Rabin-Karp algorithm is an improvement over the naive string-matching algorithm. The Rabin-Karp algorithm is a useful tool for text processing, data compression, and pattern recognition. It has a time complexity of O(n + m).
- The **Knuth-Morris-Pratt** (**KMP**) algorithm is an improvement over the naive string-matching algorithm and the Rabin-Karp algorithm. It has a time complexity of O(n + m) and is more efficient for large strings.
- The **Longest Common Subsequence** (**LCS**) is a problem in computer

science and operations research, that involves finding the longest subsequence that is common to two or more strings. It is an example of a problem that can be solved using optimization techniques, which are used to find the best solution to a problem given a set of constraints.

- Genetic algorithms are a type of optimization algorithm that is inspired by the process of natural evolution. Genetic algorithms are used in a variety of applications, including optimization, machine learning, and artificial intelligence. They are a useful tool for finding good solutions to difficult problems and are often used as a last resort when other algorithms are not effective.

Questions

1. Write a short note on Rabin Karp Algorithm **[MU DEC'18(10 MARKS)]**
2. Explain Rabin Karp Algorithm in detail [MU DEC'19(10 MARKS)] [MU MAY'19(10 MARKS)]
3. write short note on Knuth-Morris-Pratt Algorithm **[MU DEC'18(10 MARKS)] [MU MAY'19(5 MARKS)]**
4. What is the longest common subsequence problem? Find LCS for the following string:**[MU DEC'18(10 MARKS)][MU DEC'19(10 MARKS)]**

   String x = ACBAED

   String y = ABCABE
5. What is the longest common subsequence problem? Find LCS for the following string: **[MU MAY'19(5 MARKS)]**

   String X = ABCDGH

   String Y = AEDFHR
6. Explain Genetic Algorithm**[MU DEC'19(10 MARKS)]**
7. Write short note on Genetic Algorithm **[MU MAY'19(5 MARKS)]**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

Symbols

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T