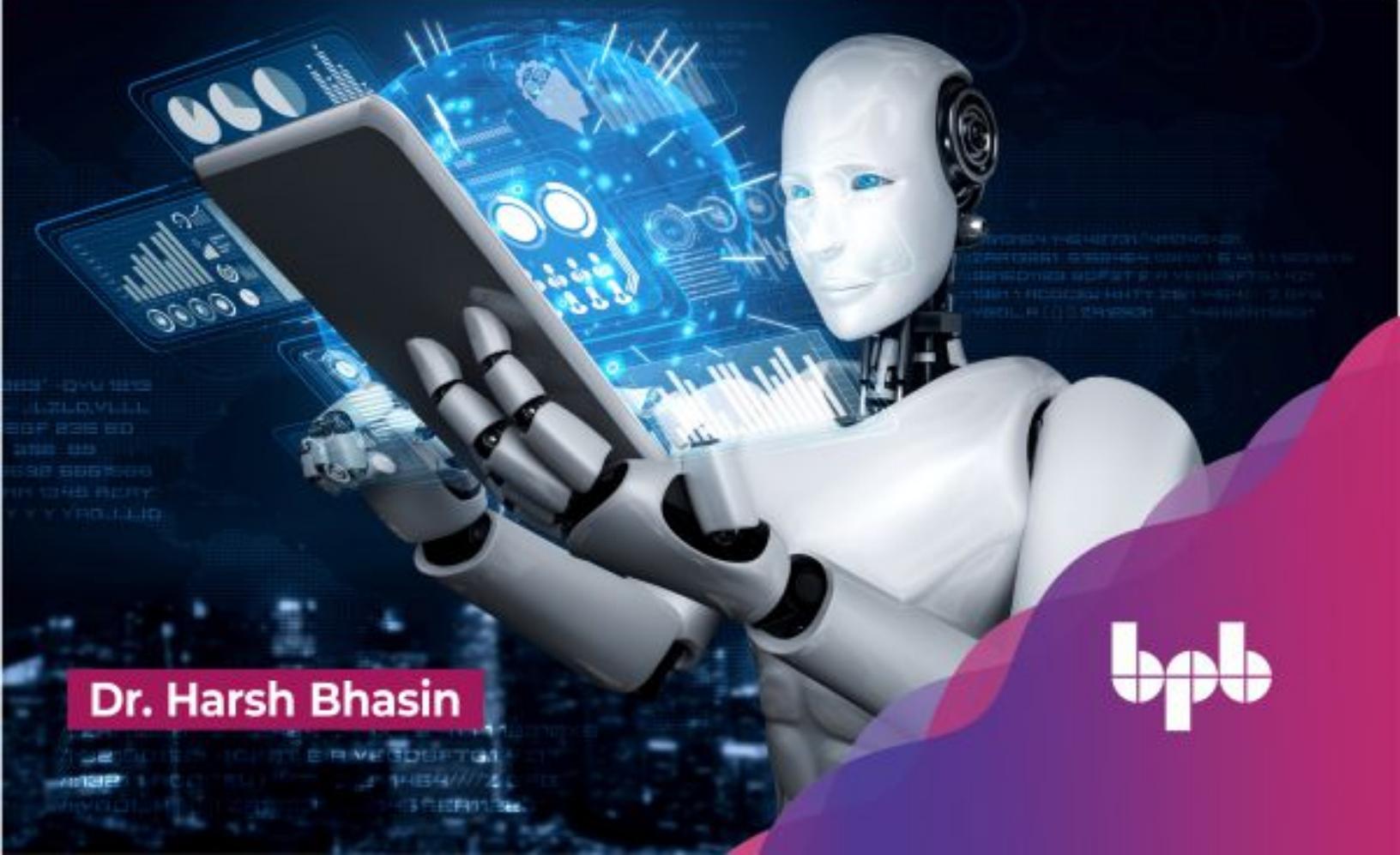


# Data Structures

—with—

# Python

Get familiar with the common Data Structures  
and Algorithms in Python



Dr. Harsh Bhasin

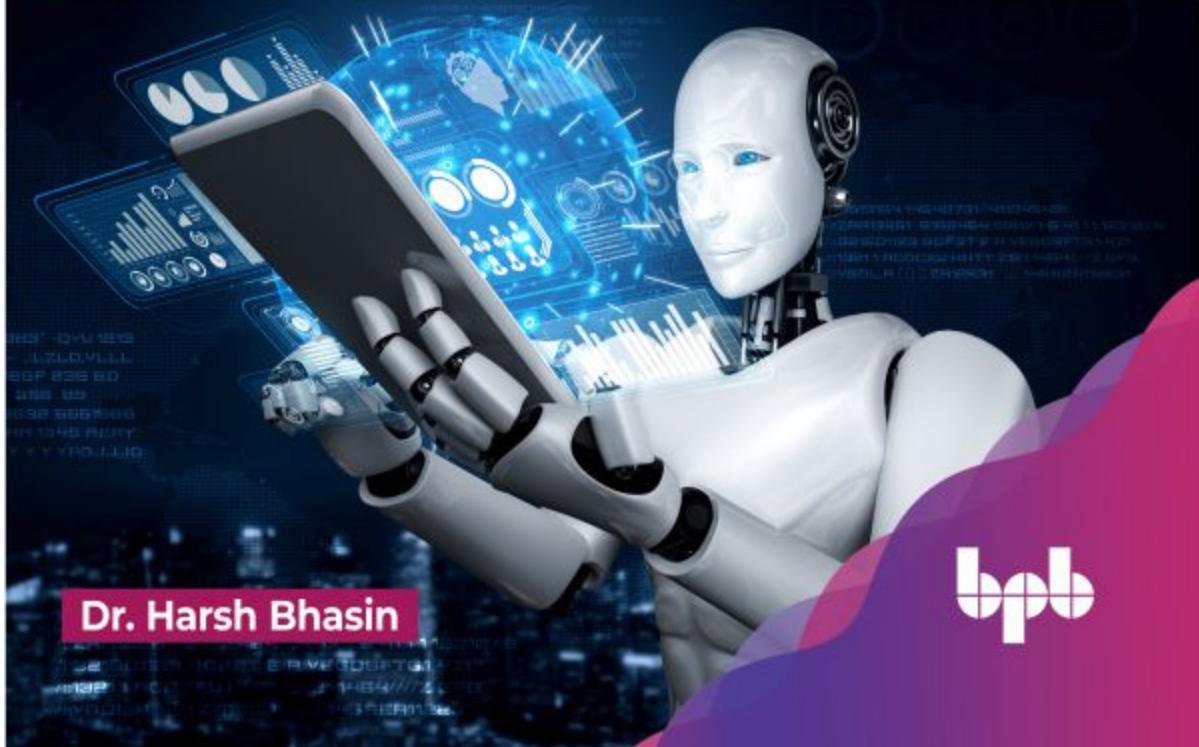
bpb

# Data Structures

— with —

# Python

Get familiar with the common Data Structures  
and Algorithms in Python



Dr. Harsh Bhasin

bpb

# **Data Structures with Python**

---

*Get familiar with the common Data  
Structures and Algorithms in Python*

---

**Dr. Harsh Bhasin**



[www.bpbonline.com](http://www.bpbonline.com)

Copyright © 2023 BPB Online

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

**First published:** 2023

Published by BPB Online  
WeWork  
119 Marylebone Road  
London NW1 5PU

**UK | UAE | INDIA | SINGAPORE**

ISBN 978-93-5551-331-1

**[www.bpbonline.com](http://www.bpbonline.com)**

**Dedicated to**

*My mother*

## About the Author

**Dr. Harsh Bhasin** is a researcher and practitioner. Dr. Bhasin is currently associated with the Center of Health Innovations, Manav Rachna Institutions. Dr. Bhasin has completed his Ph. D. in Mild Cognitive Impairment from Jawaharlal Nehru University, New Delhi. He worked as a Deep Learning consultant for various firms and taught at various Universities including Jamia Hamdard and DTU.

He has authored 11 books including Programming in C#, Oxford University Press, 2014; Algorithms, Oxford University Press, 2015; Python for Beginners, New Age International, 2018; Python Basics, Mercury, 2019; Machine Learning, BPB, 2020, to name a few.

Dr. Bhasin has authored 40 papers published in renowned journals including Alzheimer's and Dementia, Soft Computing, BMC Medical Informatics & Decision Making, AI & Society, etc. He is the reviewer of a few renowned journals and is the editor of a few special issues. He is a recipient of a distinguished fellowship.

His areas of expertise include Deep learning, Algorithms, and Medical Imaging. Outside work, he is deeply interested in Hindi Poetry: the progressive era, and Hindustani Classical Music: percussion instruments.

## About the Reviewer

**Sumeet Lalla** has done his masters of Data Science from Higher School Of Economics Moscow and Bachelors of Engineering in Computer Engineering from Thapar University. He also has 6 years of experience in Data Science and Software Engineering. His career graph includes working as a Data Scientist in Cognizant and as a Software Developer in Siemens Technology, along with working in Services and Technology Analyst in Deloitte Consulting and Pvt Ltd.

## Acknowledgement

**“Feeling gratitude and not expressing it is like wrapping a present and not giving it.”**

*– William Arthur Ward*

I am blessed to have met people who encouraged me to learn continuously. First of all, I would like to thank Professor Moin Uddin, former Pro-Vice-Chancellor, Delhi Technological University for his unconditional support. He has deposited his faith in me when no one else did. Had it not been for his encouragement I would not have been able to achieve whatever I did.

I would also like to thank Professor I. K. Bhat, Vice Chancellor, Manav Rachna University, India; and Professor Sameer Singh, Rail Vision, United Kingdom; for their continuous support and encouragement. I would also like to express my sincere gratitude to the late Professor A. K. Sharma, former Dean, and Chairperson, the Department of Computer Science, YMCA, Faridabad, for his constant encouragement. I have been able to write this book, author papers, and work on projects only because of the encouragement provided by him. I would also like to thank Professor Naresh Chauhan, former Head and Chairperson, Department of Computer Science, YMCA University of Science and Technology, and Dr. S. K. Pal, Scientist, Department of Defence and Research Organization, Govt. of India for their constant support.

I am thankful to Mr. Nishant Kumar, NCU, India, for his contribution to editing, formatting, and developing some programs for this book. I would also like to thank my students and colleagues, for their critical reviews. I am also very thankful to the editorial team of BPB Publications for providing valuable assistance.

I would like to express my sincere gratitude to my Mother, Sister, and the rest of the family including my pets: Zoe & Xena, and friends for their unconditional support to me.

I would be glad to receive your comments or suggestions which can be incorporated into future editions of the book.

# Preface

This book introduces the reader to Data Structures and Algorithms, the foundation stone of programming. The concepts discussed in this book will help the reader to understand various data structures, analyze the time and space complexity, and use these data structures for solving graded problems.

The first chapter introduces the reader to the fascinating world of Algorithms and Data Structures. The idea of complexity has been introduced in the chapter. It contains ample examples of finding the complexity of a given algorithm.

The next chapter takes the reader through various approaches to developing algorithms. The chapter introduces the Greedy approach, divide and conquer, dynamic programming, and backtracking. An introduction to branch and bound has also been included in the chapter.

Recursion has been introduced in the third chapter of this book. The chapter contains the mechanism, examples, and the process of finding the complexity of a recursive algorithm. The problems related to arrays have been discussed in the fourth chapter of this book. It presents insertion, deletion, and searching in arrays along with the complexities.

[\*\*Chapter 5\*\*](#) discusses Linked Lists. The algorithms, complexity, and problems of linked lists have been covered in this chapter. This chapter forms the basis of the following chapters. The next two chapters introduce stacks and queues. The applications of these data structures have also been included in the chapters. These chapters contain assorted problems related to stacks and queues.

[\*\*Chapters 8 and 9\*\*](#) deal with trees and heaps. The insertion and deletion in binary trees and other algorithms have been included in these chapters. Chapter ten introduces priority queues. The next chapter discusses graphs. It contains traversal algorithms, spanning tree algorithms, and shortest path algorithms.

[Chapter 11](#) contains eleven sorting techniques and discusses the related problems. selection has been dealt with in the next chapter. A very efficient searching technique called hashing has been explained in detail in the fourteen chapters. The last chapter deals with the String algorithms.

The book also contains four appendices containing Dijkstra's algorithm, all pairs' shortest path, and tree traversals using stacks and problems.

## Coloured Images

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

**<https://rebrand.ly/6rz6cpm>**

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](mailto:errata@bpbonline.com)

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

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to the eBook version at [www.bpbonline.com](http://www.bpbonline.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: [business@bpbonline.com](mailto:business@bpbonline.com) for more details.

At [www.bpbonline.com](http://www.bpbonline.com), you can also read a collection of free 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](mailto: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](http://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](http://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](https://discord(bpbonline).com)



# Table of Contents

## 1. Introduction to Data Structures

Structure  
Objectives  
Introduction  
Data types  
Types of data structures  
Game of clones  
The game of clones revisited  
Conclusion  
Multiple choice questions  
Theory-based questions  
Application-based questions

## 2. Design Methodologies

Structure  
Objectives  
Greedy approach  
Divide and conquer  
Backtracking and dynamic programming  
Longest common sub-sequence  
Conclusion  
Multiple choice questions  
Programming/application  
Further references

## 3. Recursion

Structure  
Objectives  
Exponentiation  
Tower of Hanoi  
Rabbit problem

[Generating binary numbers](#)

[Lists](#)

[Numbers](#)

[Conclusion](#)

[Multiple choice questions](#)

[Programming](#)

[Further references](#)

## **4. Arrays**

[Structure](#)

[Objectives](#)

[Introduction](#)

[Memory map](#)

[Address in column-major](#)

[Inserting and deleting](#)

[Operations on arrays](#)

[Linear search](#)

[Problems](#)

[Conclusion](#)

[Multiple choice questions](#)

[Programming](#)

## **5. Linked List**

[Structure](#)

[Objectives](#)

[One-way linked list](#)

[Traversing](#)

[Insertion and deletion](#)

[Two-way linked list](#)

[Traversing](#)

[Insertion and deletion](#)

[Cyclic list](#)

[Stacks and Queues](#)

[Reversing a linked list](#)

[Concatenate lists](#)

[Check cycle](#)  
[Conclusion](#)  
[Multiple choice questions](#)  
[Theory](#)  
[Problems](#)

## **6. Stacks**

[Structure](#)  
[Objectives](#)  
[Introduction](#)  
[Implementing two stacks using a single list](#)  
[Types and uses](#)  
[Reversing a string](#)  
[Expressions](#)  
    [Evaluation of postfix](#)  
    [Infix to postfix](#)  
    [Infix to prefix](#)  
[Problems](#)  
[Conclusion](#)  
[Multiple Choice Questions](#)  
[Problems](#)

## **7. Queues**

[Structure](#)  
[Objectives](#)  
[Introduction](#)  
[Algorithm and implementation](#)  
[Circular queue](#)  
[Doubly-ended queue: DEQueue](#)  
[Generating binary numbers using a queue](#)  
[Stack using two queues](#)  
[Stack from a single queue](#)  
[Scheduling](#)  
[Conclusion](#)  
[Multiple Choice Questions](#)

## Problems

### 8. Trees-I

Introduction

Structure

Objectives

Definition and terminology

Representation of a Binary Tree

Traversal

Post-order traversal

Pre-order traversal

Binary search tree

Insertion in a BST

Deletion

Leftmost node

Rightmost node

Conclusion

Multiple choice questions

Numerical/problems

### 9. Trees-II

Structure

Objectives

AVL trees

Insertion

Deletion

Insertion in an AVL tree

Deletion from an AVL Tree

B Trees

Conclusion

Multiple choice questions

Theory

Numericals

## 10. Priority Queues

[Structure](#)  
[Objectives](#)  
[Introduction to priority queues](#)  
    [Structure of heap](#)  
    [Operations](#)  
[Inserting an element in a heap](#)  
[Deletion](#)  
[Heap sort](#)  
[Problems](#)  
[Conclusion](#)  
[Multiple choice questions](#)  
[Programming](#)  
[Further references](#)

## **11. Graphs**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[Representation](#)  
[Traversals](#)  
[Depth First Search](#)  
    [Breadth First Search](#)  
    [Topological sort](#)  
[Spanning tree](#)  
[Kruskal's algorithm](#)  
[Conclusion](#)  
[Multiple choice questions](#)  
[Numerical/application based](#)  
[Programming](#)

## **12. Sorting**

[Structure](#)  
[Objectives](#)  
[Bubble sort](#)  
[Comb sort](#)

[Selection sort](#)  
[Insertion sort](#)  
[Radix sort](#)  
[Counting sort](#)  
[Merge and merge sort](#)  
[Partition and quick sort](#)  
[Conclusion](#)  
[Illustrations](#)  
[Multiple choice questions](#)  
[Theory](#)

## **13. Median and Order Statistics**

[Introduction](#)  
[Structure](#)  
[Objectives](#)  
[Introduction to median and order statistics](#)  
[Median of medians](#)  
[Median using heaps](#)  
[Median using insertion sort](#)  
[Median using partition](#)  
[Conclusion](#)  
[Solved problems](#)  
[Multiple choice questions](#)  
[Applications/implementation](#)  
[Bibliography](#)

## **14. Hashing**

[Structure](#)  
[Objectives](#)  
[Hash tables](#)  
[Storing information](#)  
[Sorted sequential array](#)  
[Linked list representation](#)  
[AVL trees](#)  
[Hashing](#)

[Hash function](#)  
[Collision resolution](#)  
[Selecting hash function](#)  
[Collisions](#)  
[Collision resolution](#)  
[Linear probing](#)  
[Quadratic probing](#)  
[Separate chaining](#)  
[Solved problems](#)  
[Conclusion](#)  
[Multiple choice questions](#)  
[Theory](#)  
[Problems](#)  
[Programming](#)

## **15. String Matching**

[Structure](#)  
[Objectives](#)  
[Introduction to string-matching](#)  
[Brute force method](#)  
[Rabin Karp](#)  
[Knuth–Morris–Pratt algorithm](#)  
[KMP method](#)  
[Conclusion](#)  
[Multiple choice questions](#)  
[Theory/applications](#)  
[Find errors/special cases](#)  
[References](#)

## **Appendix 1: All Pairs Shortest Path**

[Introduction](#)  
[All Pairs Shortest Path](#)

## **Appendix 2: Tree Traversals**

[Introduction](#)

[In-Order Traversal](#)  
[Pre-order traversal](#)  
[Post-order traversal](#)

### **Appendix 3: Dijkstra's Shortest Path Algorithm**

[Introduction](#)  
[Dijkstra's shortest path algorithm](#)

### **Appendix 4: Supplementary Questions**

[Arrays: Level 0](#)  
[Arrays: Level 1](#)  
[Stacks](#)  
[Linked List](#)  
[Trees](#)  
[Graphs](#)  
[Application based](#)

### **Index**

# CHAPTER 1

## Introduction to Data Structures

*R*ichard Buckland from the University of New South Wales often uses the acronym PAPP while teaching Data Structures and Algorithms. Here, the first P stands for the problem, the A stands for the algorithm, the second P for the program, and the third P for the process. An algorithm is the sequence of steps to accomplish the task at hand or solve the problem. The algorithm is then implemented in some programming language, thus, giving rise to a program. When you execute this program, it becomes a process.

You learn languages, say Python, to implement an algorithm; this takes us from the algorithm to the program. You execute this program, thus, creating a process. To understand the transition from a program to a process, you learn the basics of Compiler Design and Operating Systems. In this chapter, you will learn the transition from Algorithm to Program and to some extent, the problem to the algorithm. That is, you will be presented with algorithms related to some data structures, which you will implement. Furthermore, at times, you will be presented with some problems that you need to solve using the implemented data structures. This chapter defines the term data structure and presents a brief overview of the things to come.

The reader is expected to know Python; for that matter, he/she must be versed in at least one programming language. The following discussion will extensively use loops, nested loops, lists, tuples, dictionaries, arrays (both 1-dimensional and 2-dimensional), and the difference between reference types and value types.

### Structure

In this chapter, we will cover the following topics:

- Define data structures
- Define data type
- Classify data structures

- Learn a way to sort numbers

## **Objectives**

This chapter aims to introduce the fascinating subject of data structures to the readers. The chapter will deal with the basic data types, types of data structures, and a problem related to sorting.

After reading this chapter, the reader will be able to classify data structures and understand why this study is important. The reader will also learn when to use which data structure and what operations can be performed on them. This discussion will act as a foundation stone of the building called data structures.

## **Introduction**

The single value stored in a variable is called a datum. The set of values of a variable is called data. Note that data may contain many values, each of which is referred to as a data item. Each data item can further be divided into sub-items. For example, a variable called **name**, which stores the name of a student, may have the value “Brandon Walsh”. This data item can be divided into two sub-items, “Brandon” and “Walsh”, which are the first name and the last name, respectively, though some data items like PAN CARD NUMBER cannot be divided into sub-items.

Some of you might have studied Object Oriented Programming and have some basic idea of a “Class”, which is a real or a conceptual entity having importance to the problem at hand. An entity has attributes, each of which can be assigned some values and these values may belong to a particular data type. For example, in the following illustration, an entity called Movie has attributes name, year, genre, and so on. The data type of Name is a string, that of Year is an integer, and that of Genre, Director, and Music are strings. The values assigned to these variables are “Sairat”, 2016, “Don’t talk about it”, “Nagraj Manjule”, and “Ajay-Atul” respectively.

### **Movie**

Name : Sairat

Year : 2016

Genre : Don't talk about it

Director : Nagraj Manjule

Music : Ajay-Atul

The set of similar entities constitutes an entity set, which will help us to solve the preceding problem. This takes us toward meaningful data, which can be processed. Here comes information that can be considered as processed organized data. The organization is important, and this subject will teach you how to organize data.

Let us consider a file containing records of movies. Each record contains five fields: Name, Year, Genre, Director, and Music. So, we have a file containing records, and each record contains fields. This is an example of fixed-length records. There are files containing variable-length records as well. In such files, the maximum and minimum length is generally mentioned. The data needs to be organized to facilitate efficient and effective handling of this data. This subject teaches you data structures, which will help you to organize data efficiently and effectively.

**Data structures:** Data structures include the organization of records into complex structures, the implementation of such structures and the analysis of the amount of memory and time taken by such structures.

Having seen the definition of data structures, let us now move to the definition of data types.

## Data types

*"The data types constraints values that a variable can take and define operations that can be performed on it."* The basic data types such as int, char, and float are also called primitive data types. In older versions of C, for example, an integer is used to take two bytes of memory (16 bits). Out of these, one bit was reserved for the sign of the integer, and the remaining 15 bits were for storing the values. Note that the maximum value can therefore be  $2^{15} - 1 = 32767$  and the minimum value could be  $-2^{15} = -32768$ . Likewise, in the versions of C, which allot 4 bytes to integers, the maximum value can be  $2^{31} - 1$  and the minimum can be  $-2^{31}$ .

The **int** data type, therefore, constrains a) the types of values that can be stored in an integer type variable b) the range of values that can be stored in

such a variable, and c) the organization of memory (as in one bit will store the sign information and the rest binary equivalent of a given number).

The primitive data types are provided by the programming language, such as integer, character, float, Boolean and so on. The amount of memory allocated to each depends on the language and some other factors. For example, an **int** in C occupies two bytes in Turbo (DOS) and four in Visual Studio.

Most programming languages also allow user-defined data types as well. In Object Oriented Languages, classes provide a way to create user-defined data types. In C, structures can be used to create these.

Let us now have a brief overview of the types of Data Structures. The following section will give you a glimpse of things to come.

## Types of data structures

This subject primarily focuses on the organization of data. This organization can be modeled in different ways, each of which is referred to as a data structure. For example, a set of ordered numbers can be placed in a linear array or in a tree-like structure. The first method is easy but may require more time for some operations like insertion and deletion, whereas the second method though slightly complex, will help in easy insertion and deletion.

Any model that you choose must be representative of the relationship between the data members, and the structure should be simple, effective, and efficient. Some of the data structures that will be explored in the following chapters are as follows:

- **Arrays:** An array is one of the simplest data structures. It is homogeneous, and the elements are stored at consecutive memory locations. The elements of the array will be represented by the name of the array followed by square brackets ([ ]) containing the index of the element. Note that the first element is generally placed at index zero, the second at index one, and so on.
- **Two-dimensional arrays:** Two-dimensional arrays are table-like structures containing rows and columns. The elements in these matrix-like structures can be accessed by the name of the array, followed by

the two indices depicting the row index and the column index. Generally, in memory, the two-dimensional arrays are stored in the row-major or column-major format, as discussed in [Chapter 4, Arrays](#).

- **Linked list:** The linked list is a data structure in which the units called nodes are linked together. Each of these nodes contains at least two parts normally: (a) the information and (b) the address of the next node. The address of the last node is null/none. We may also have more than one container for the address in each node, like in the case of a doubly linked list. In a doubly linked list, each node contains the address of the previous node, information and that of the next node.
- **Stacks:** Stacks is a linear data structure that follows the principle of **Last in first out (LIFO)** or **First in last out (FILO)**. So, if you insert 51, 78, 90, and 49 in a stack, the order in which these elements would be removed is 49, 90, 78, and 51 (which is LIFO). Stacks is extremely useful while implementing recursion and in evaluating various types of expressions such as postfix, prefix, and so on.
- **Queue:** Queue is a linear data structure that follows the principle of **First in first out (FIFO)**. So, if you insert 51, 78, 90, and 49 in a queue, the order in which these elements would be removed is 51, 78, 90 and 49 (which is First In First Out). Queues are used in the implementation of scheduling algorithms such as FIFO and so on.
- **Graph:** Graph is a set containing  $\{V, E\}$ , where  $V$  is a finite non-empty set of vertices, and  $E$  is a finite non-empty set of edges. They are used in almost all facets of Computer Science, such as circuits and systems, networking, and so on.
- **Trees:** Trees generally contain nodes depicting hierarchical relationships. Technically, it is a graph that does not contain a cycle, isolated vertex, or isolated edge(s). Such data structure greatly helps us in searching and even sorting.

The operations that can be performed on each of the data structures are as follows:

- **Traversal:** A traversal defines a way to visit each element of a given data structure. A graph, for example, can be traversed using Depth First Search, Breadth First Search, Level First Search, and so on. A binary

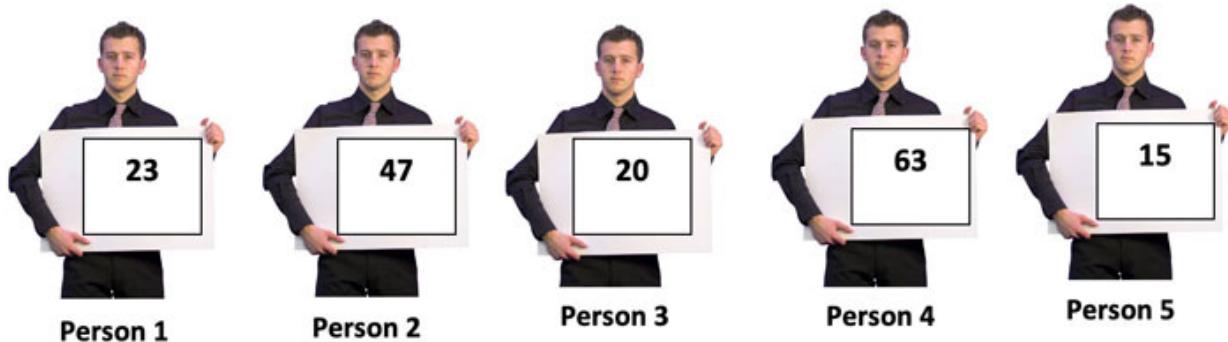
tree can be traversed using In-order, Post-order, or Pre-order traversal, and so on.

- **Insertion:** The process of inserting a node in a given data structure is important both in terms of time complexity and memory management. For example, inserting an element at the end of an array takes  $O(n)$  time, whereas inserting an element in a stack takes  $O(1)$  time.
- **Deletion:** Like in the case of insertion, the deletion of an element is important both in terms of time and space complexity. For example, deleting an element from the end of an array takes  $O(1)$  time, whereas deleting an element from the beginning of an array takes  $O(n)$  time.
- **Searching:** This is one of the most important operations in data structures. In fact, many data structures are designed so that an item can be efficiently searched from them.

Let us now have a look at one of the most important problems in data structures: Sorting.

## Game of clones

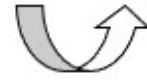
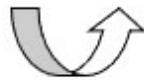
Consider the following situation. You have five people (all the same: clones?) with placards having a number written on them. All the cards have different numbers. The five people are assigned numbers indicating their positions, which are from 1 to 5 ([figure 1.1](#)). They start playing a game as per the following rules.



*Figure 1.1: Five persons holding different placards*

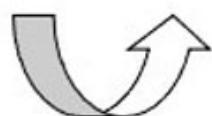
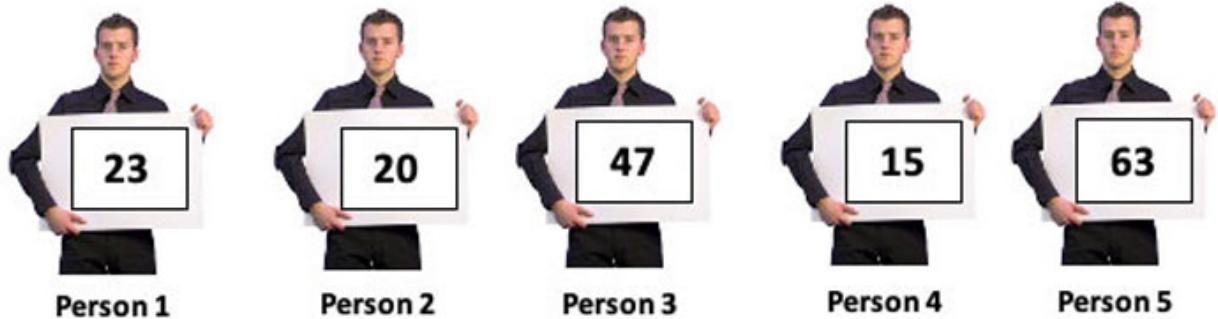
Starting from  $i=1$  (till  $i=4$ ), person  $i$  and  $(i+1)$  exchange their cards if the number on  $i$ 's card is greater than  $(i+1)$ 's card. That is, person 1 will

exchange the card with person 2 if person 1's card has a number greater than person 2's card; then person 2 will exchange the card with person 3 if person 2's card has a number greater than person 3's card, person 3 will exchange the card with person 4 if person 3's card has a number greater than person 4's card, and finally, person 4 will exchange the card with person 5 if person 4's card has a number greater than person 5's card. This way, person 5 will have the card having the greatest number after this step ([figure 1.2](#)):



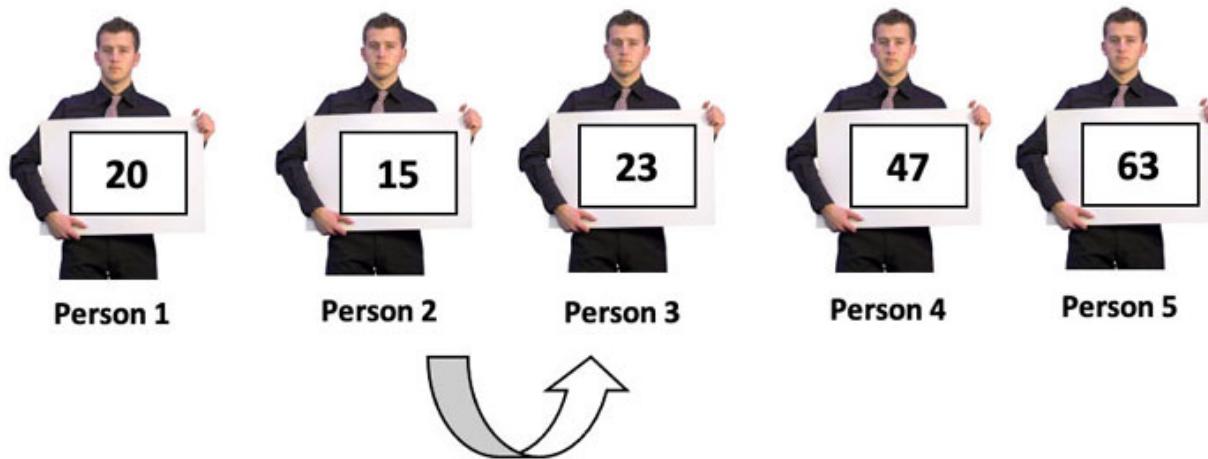
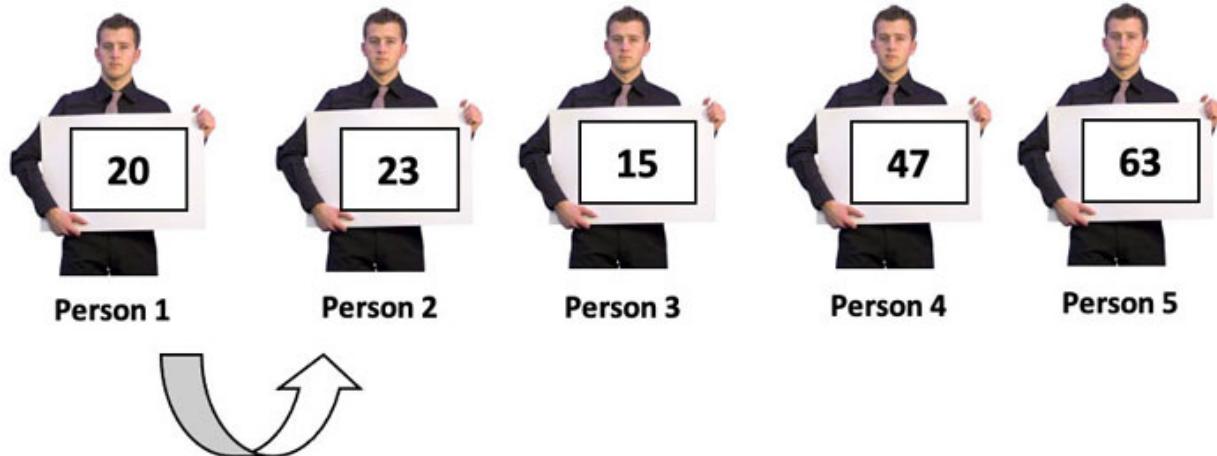
**Figure 1.2:** At the end of Step 1, person 5 will have the card having the largest number

In the next step, the preceding process is repeated, except for the exchange between persons 4 and 5, as person 5 already has the greatest number. After this, Step 4 will have the second largest number ([figure 1.3](#)):



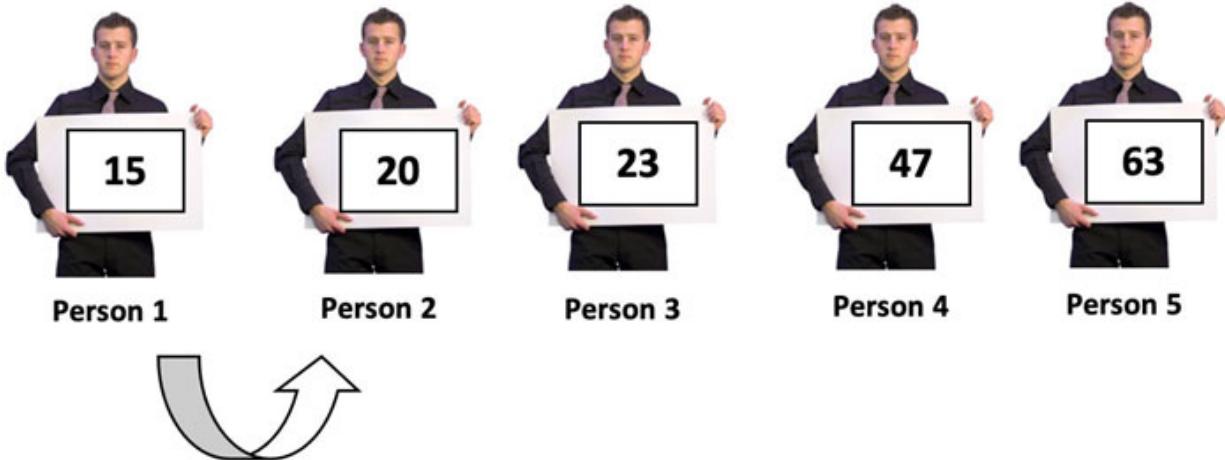
**Figure 1.3:** At the end of Step 2, person 4 will have the card having the second largest number

In the next step, the preceding process is repeated till all the numbers are sorted ([figure 1.4](#)):



**Figure 1.4:** At the end of Step 3, person 3 will have the card having the third largest number

Note that in the last step, only a single comparison is required ([figure 1.5](#)):



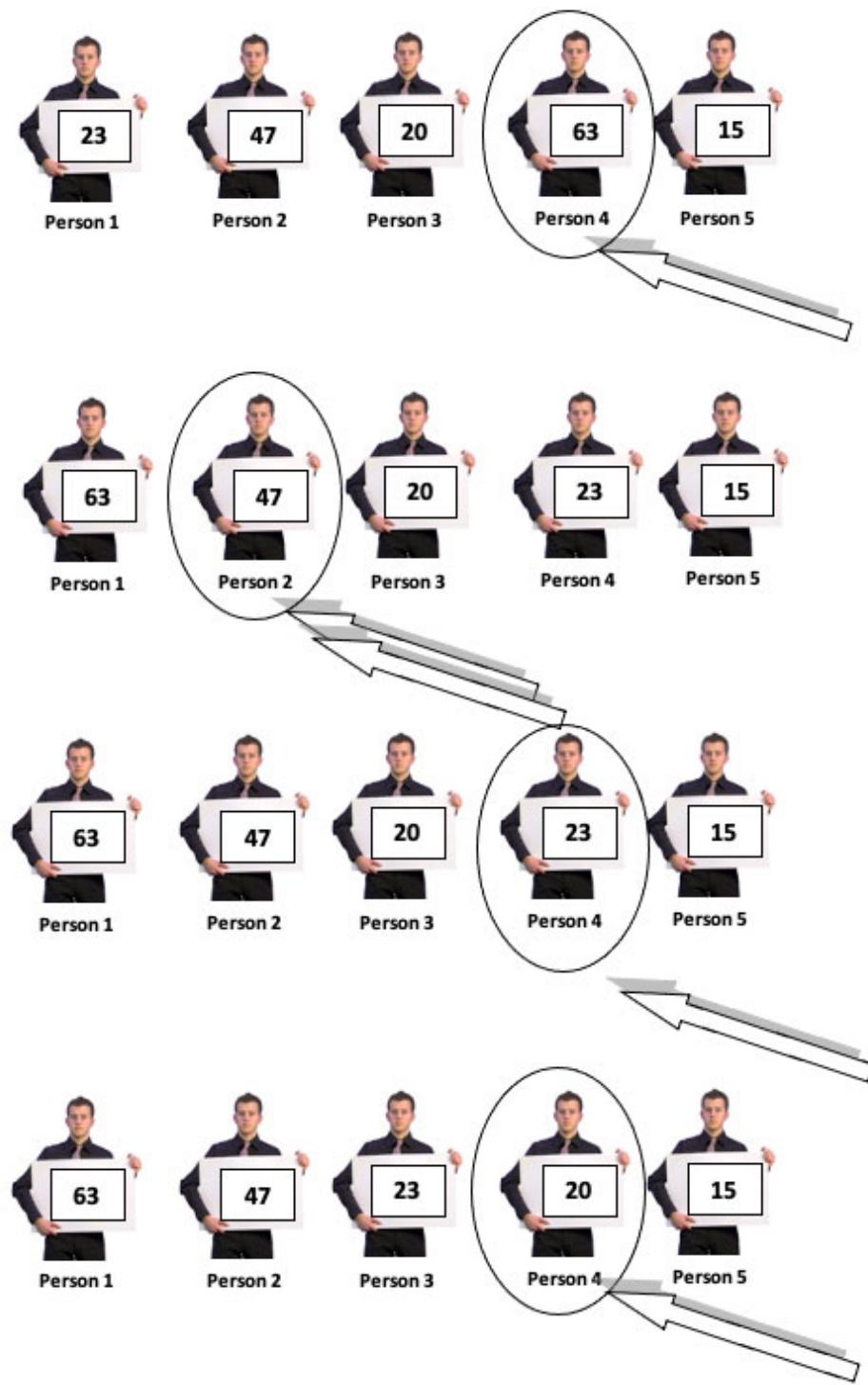
**Figure 1.5:** At the end of Step 4, person 2 will have the card having the fourth largest number

Note that after each step, the largest element of the remaining array is placed at the  $(i+1)^{\text{th}}$  last position. Here,  $i$  varies from 0 to  $(n-1)$ ,  $n$  being the number of terms. The complete process results in the sorted array. Note that in the preceding process, the total number of comparisons is  $4 + 3 + 2 + 1 = 10$ . Had there been  $n$  numbers, the total of comparisons would have been  $(n-1) + (n-2) + \dots + 1 = (n \times (n-1))/2$ . That is of order  $n^2$ . The process scales very poorly. When the number of elements doubles, the complexity changes by almost four times, provided  $n$  is a large number, and all the overheads take a negligible amount of time.

## The game of clones revisited

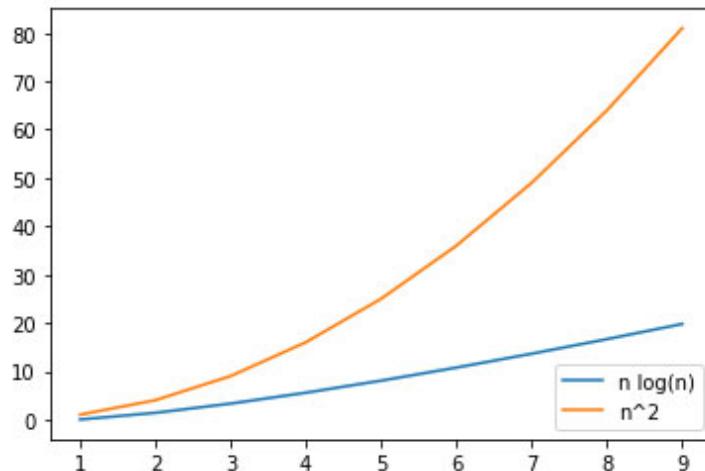
The preceding game was being watched by Phineas Fletcher. He approached the group of people playing the game and suggested a simple change. He suggested selecting the highest number in each step and replacing it with the  $i^{\text{th}}$  element,  $i$  starting from 0. The process is shown in the following figure.

In the figure that follows, the largest number (63) is selected in the first step and swapped with the element at the  $0^{\text{th}}$  index. In the next step, the second largest number is selected and swapped with the element at the  $1^{\text{st}}$  index. Likewise, in the successive steps, the largest numbers from the remaining array are chosen and swapped with the element at the  $i^{\text{th}}$  index ([figure 1.6](#)). Note that if selecting the largest element from the remaining array takes time,  $O(\log n)$  the whole process should take time of order  $O(n \log n)$ , which is much better than the previous method of sorting numbers.



**Figure 1.6:** Another way to sort numbers

The following graph ([figure 1.7](#)) shows the variation of  $n^2$  and  $(n \log n)$  with  $n$  and makes a strong case for the second method of sorting numbers:



**Figure 1.7:** Variation of  $n \log(n)$  and  $n^2$  with  $n$

## Conclusion

This chapter introduced the reader, the fascinating world of data structures. The chapter defined the term data structure, explained its types, stated the difference between data structures and data types, and presented a problem to explain the importance of time complexity. The reader should get hold of the idea of data structure, its types, and their applications after reading this chapter. This chapter is your first step towards becoming an accomplished programmer.

The next chapter takes the discussion further and introduces algorithms and complexity. The features of an algorithm, the types of algorithms, and asymptotic notations for complexity are discussed in the upcoming chapter.

## Multiple choice questions

1. Which of the following follows the principle of Last In First Out (LIFO)?
  - a. Stack
  - b. Queue
  - c. Linked List
  - d. None of above

**2. Which of the following follows the principle of First In First Out (FIFO)?**

- a. Stack
- b. Queue
- c. Linked List
- d. None of the above

**3. Which of the following is a linear data structure?**

- a. Stacks
- b. Queue
- c. Array
- d. All the above

**4. Which of the following is a non-linear data structure?**

- a. Tree
- b. Graph
- c. Plex
- d. All the above

**5. Which of the following data structure is used to evaluate a postfix expression?**

- a. Queue
- b. Stacks
- c. Array
- d. None of the above

**6. Which of the following data structure is used to evaluate a prefix expression?**

- a. Stacks
- b. Queue
- c. Array
- d. None of the above

- 7. Which of the following data structure is used to find the shortest path?**
- Trees
  - Graph
  - Plex
  - None of the above
- 8. Which of the following data structure is used for an efficiently searching an element?**
- Binary Search Trees
  - Queue
  - Stack
  - None of the above
- 9. An integer in Turbo C takes two bytes; what is the maximum value that can be stored in it?**
- 32767
  - 32768
  - 65536
  - 65535
- 10. In the preceding question, if it is an unsigned integer, what is the maximum value that can be stored in it?**
- 32767
  - 32768
  - 65536
  - 65535

### **Theory-based questions**

- What is a data structure?
- Define data type and differentiate between primary and secondary data types?

3. What are linear and non-linear data structures?
  4. Define Stack and give any two applications of Stacks?
  5. Define Queue and give any two applications of Queues?
  6. Define Trees and give any two applications of Trees?
  7. Define Graph and give any two applications of Graphs?
  8. How will you sort a list of numbers without spending time?
  9. Explain PAPP.
10. Write a short note on why you should study data structures.

## **Application-based questions**

- 1. Harsh keeps track of music videos by saving the following data on his PC:**
  - Name
  - Singers
  - Music Director
  - Album
  - Year
  - Duration
  - Lyricist
  - a. What steps would you take if you needed to store the preceding information of all the tracks in a file?
  - b. State the data types of all the attributes?
  - c. Give reasons for choosing a particular data type for an attribute? For example, explain why you choose a list/an array for the variable singers.
  - d. The records need to be stored in a file and accessed by a Python program. Can you suggest a better way of storing the data?
- 2. Listening to songs continuously motivated Harsh to keep track of albums as well.**

- a. What attributes must he store according to you?
  - b. State the data type of each of the attribute?
  - c. Can you suggest which attribute will always be unique for a particular record. What is such attribute called?
  - d. Can you link the previous question and this question with such unique attribute?
3. In the preceding two questions, segregate the attributes as variable length attributes and fixed length ones?
4. Give a brief description of how will you search a record in this file?
5. Harsh decides to sort the records alphabetically. Based on the techniques introduced in the chapter, can you suggest a method to him?
6. Consider the following expression:  $(a + b) - c$   
Create a tree for the same.
7. In your organization or University, there must be some hierarchy. For example, the Dean reports to the Vice Chancellor, the HOD (Head of Department) reports to the Dean and the faculty of a department reports to the HOD. Furthermore, each faculty may have a Teaching Assistance (TA). Create a tree representing this hierarchy.
8. You have given a list of numbers. Find out the subset of the list that sums to a particular number. Suggest a method for the same?  
For example, If the given list is [1, 2, 3, 4, 5, 6] and the sum is 6, then the subsets that sum to 6 are: {1, 5}, {2, 4}, {6}

## 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

### Design Methodologies

The previous chapter discussed the definition, types, and importance of data structures. Some approaches for sorting a list of numbers were also discussed. This chapter takes the discussion forward and introduces design techniques. The techniques, their applicability, and when to use them have been discussed in this chapter. Also, the chapter gives an ephemeral introduction to asymptotic complexity, which has been discussed in detail in the Appendix of this book.

Let us begin with an example. Suppose you need to find a number in a given list. What will you do? If the list is not sorted, you may match the element at each index with the given number. If they match, you will print the index. In the other case, you will move ahead. If you are not able to find the number, then you will print the message “Not Found”. This is Linear Search and takes time proportional to  $n$  (the number of elements in the list). If the list is sorted, we may apply Binary Search, which divides the list into two halves and reduce the size of the problem in half at each step. The algorithm takes time proportional to  $O(\log n)$ . Both these techniques are discussed in detail in the following chapters.

Now, hold and think about a situation where the sorted list is shifted by  $k$  positions (a finite number). How will you find a given element in this list? You may apply Linear Search, a variant of Binary Search, or a heap ([Chapters 3, Arrays](#) and [9, Heaps](#)). The chapters that follow will address all these questions. However, any solution that you think should be the following:

- a. Correct
- b. Efficient

The first and foremost thing in any solution that you propose is that it must be correct and must cater to all possible test cases. Efficiency comes next. The program should optimally use resources: both memory and time. Some of the approaches that you follow to solve a problem can be the following:

- Divide and conquer
- Greedy approach
- Dynamic programming

- Backtracking and branch and bound

Divide and Conquer is applied when a problem can be divided into similar sub-problems, and each sub-problem can be solved using that approach. Also, if required, there should be a way to club together the solutions to the sub-problems to return the solution to the original problem. Exploring each possibility and backtracking from the leaves to get the solution can be another approach, though expensive. If the sub-solutions can be memorized using a table, it leads to dynamic programming. The following sections will touch upon each of these approaches and will help you develop an understanding of these approaches and their applicability.

## Structure

In this chapter, we will cover the following topics:

- Divide and conquer
- Greedy approach
- Dynamic programming and backtracking
- Longest common subsequence

## Objectives

This chapter presents an overview of various methodologies, including the Greedy approach.

Divide and conquer, dynamic programming, and backtracking. This chapter also explains the difference between divide and conquer and dynamic programming and between dynamic programming and backtracking. Each methodology is explained using examples and numerical. Finally, the chapter will help you to move towards applying the learnt methods to solve problems.

## Greedy approach

Axl is a teenager, who lives in Orson. He decides to open his burger joint named “The HecksBurger” and employs his younger brother Brick to handle the cash counter. The cash counter has bills having denominations 500, 100, 50, 20, 10, 5, and 1. Assuming that the prices of the burgers are in natural numbers, how do you think he would return the change with a minimum number of bills?

Let us understand this with the help of an example: If he gets 1,000 and needs to return 657. He will start with 500 (the highest denomination) and find the number of bills of this denomination required by dividing the amount to be returned by the denomination (and then taking the floor), which is  $657/500$ , which is 1. Now, the remaining balance can be found by subtracting from 657, which gives 157. This is followed by repeating the process with the next highest denomination. That is, for finding the number of denominations of 100, divide the remaining, i.e., 157 by 100 (and then take the floor), which gives 1. The remaining amount in the next step will be. Likewise, the number of notes of 50, 20, 10, 5, and 1 will be 1, 0, 0, 0, and 2. That is, the number of bills required for change are [1, 1, 1, 0, 0, 0, 2].

The following Python code takes the amount to be returned as the input along with a list containing denominations in the sorted order and returns the list containing the number of denominations.

### Code:

```
def coin_changing(L, amount):
    denomination = []
    i=0
    while(i<len(L)):
        num = int(amount/L[i])
        amount = amount - num*L[i]
        denomination.append(num)
        i+=1
    return denomination
L=[500, 100, 50, 20, 10, 5, 2, 1]
print(len(L))
den=coin_changing(L, 657)
print(den)
```

### Output:

1. 8
2. [1, 1, 1, 0, 0, 1, 1, 0]

The preceding is an example of a Greedy approach, since at each step, the “best option at that point” is selected, and we proceed further. The approach is good but might not lead to an optimal solution always. We will explore this approach

further in the chapters that follow. Particularly, this approach would be used to solve the minimum spanning tree problem.

## Divide and conquer

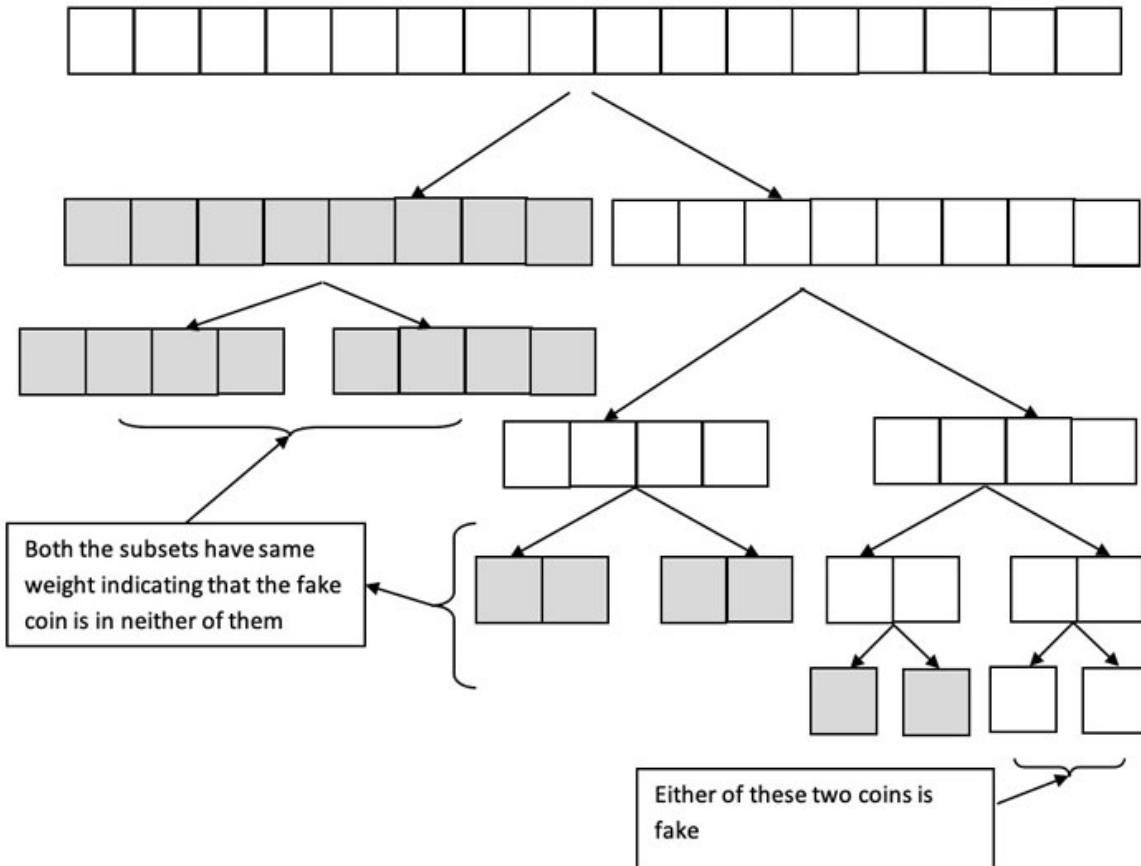
In order to understand divide and conquer, let us consider the following illustration. Once upon a time, there was a king named Harsh. He was overtly fascinated with himself, and hence, stopped the coins used earlier in the kingdom. The new coins had his photo engraved. Moreover, all the subjects of the kingdom were asked to do the transactions in  $2^k$  coins, where  $k \in \mathbb{Z}$ , for reasons which only future generations would understand.

The group of corrupt people called corrupts followed a protocol. They used to give one fake coin while doing a transaction. The weight of the fake coin was not the same as that of the original coin, and we do not know the weight of the valid coin. In order to identify the fake coin from the given coins in a transaction, the Algorithm department (he had established this department after selling the government IT company to his businessman friend) came up with an interesting solution, which is as follows:

Divide the set of  $n$  coins into two sets having  $n/2$  coins each. If the weights of the two sets are equal, then there is no fake coin in the given set of  $n$  coins. In the other case, divide each set of  $n/2$  coins into two sets of  $n/4$  coins. The part whose two subsets of  $n/4$  coins have equal weights do not have any fake coin, and the other set has a fake coin. The remaining coins can be divided further to reach a set of two coins, one of which is fake. Finally, the fake coin can be identified by comparing the remaining two coins with any coin in the set containing valid coins.

Figure 2.1 exemplifies the solution. In case there are 16 coins, then the coins can be divided into two sets having eight coins each. One of the sets has a fake coin. When the sets are further divided into two sets (we will have four sets of four coins). The two parts having the same weight (shown in gray) will not have any fake coin because there is only one fake coin, and the set having the fake coin will show different weights of its two subsets. The process is repeated till the last level (at which we have just one coin).

Note that the depth of the preceding tree is, as it the second level, there will be  $n/2$  elements in each subset; at the third, there will be  $n/2^2$ , in the next step, there will be  $n/2^3$  values, the process stops when only one value remains. That is, when  $n/2^i = 1$ , or  $\log_2 n$ .



*Figure 2.1: Finding a fake coin*

This was an example of divide and conquer. In divide and conquer, we divide the given problem into sub-problems of the same type. If needed, the solutions to all these problems are then clubbed, and the final solution is returned.

This technique can also be applied to problems like searching if the given array is sorted. The following chapters discuss some of the important applications of divide and conquer, such as merge sort, quick sort, matrix multiplication, exponentiation, and so on.

## Backtracking and dynamic programming

Mathematical researchers faced some problems in the 1950s. *Charles Erwin Wilson* was the Secretary of Defense under President Eisenhower. Earlier, he worked as the CEO of *General Motors*. According to Bellman, “*he had pathological fear and hatred of the word research*”. This gives an idea of his relationship with mathematics. Bellman was working with the RAND Corporation that was employed by the Air Force. To hide that he was doing

mathematics inside RAND, he devised the word Dynamic Programming so as to save the project that RAND got from being scrapped [1].

In **Dynamic Programming (DP)**, we divide the problem into smaller subproblems, and the solution to these smaller problems helps to construct that of the bigger one. Here also we divide the problem into subproblems like in the case of divide and conquer, but there is a difference. In divide and conquer, all the problems are solved individually, and then if needed, the solution is combined. In dynamic programming, on the other hand, the smaller solutions contribute to the larger ones, which contribute to still larger ones. This technique is effective, much more effective than techniques like backtracking.

To understand this, consider a simple program to find the factorial of a number. The factorial of a number is found by taking the product of all the numbers starting from 1 to that number, that is,

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

Note that,

$$n! = 1 \times 2 \times 3 \times \dots \times (n - 1) \times n = (n - 1)! \times n$$

That is,

$$\text{fac}(n) = n \times \text{fac}(n - 1)$$

factorial can be expressed in terms of itself and the base case, which is

$$\text{fac}(1) = 1$$

The following code assumes that the user enters a number  $\geq 1$ . For negative numbers, the code can be modified by including a branch in the if-elif-else ladder. The implementation of the preceding using recursion is shown as follows:

### Code:

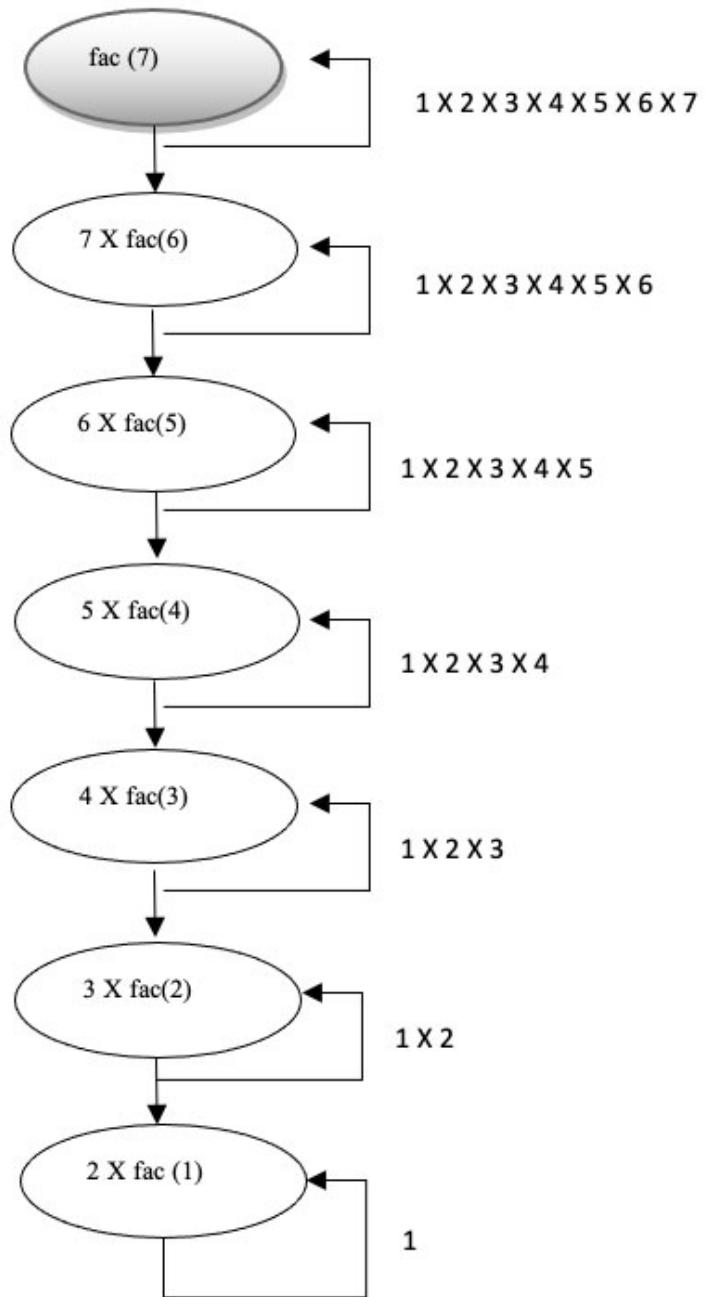
```
def fac(n):
    if n==1:
        return 1
    else:
        return fac(n-1)*n
```

### Output:

```
>>fac(1)
1
```

```
>>fac(8)
40320
>>fac(40)
815915283247897734345611269596115894272000000000
```

The preceding program is not the most efficient approach to solve this problem. The reason is that in calculating the factorial of higher numbers, that of the lower numbers is calculated again and again. To understand this, have a look at [figure 2.2](#). Note that in calculating a factorial of 7, that of 6 is required, for which that of 5, 4, 3, 2, and 1 are required. A factorial of 1 is the base case of this recursive algorithm.



*Figure 2.2: Factorial using recursion*

Now, consider the following implementation of factorial. Here, a list stores the factorial of a number and can be used to find the factorial of the later numbers.

### Code:

```

def fac1(n):
    fac_num=[]
    if n==1:

```

```

fac_num.append(1)
else:
    fac_num.append(1)
    for i in range(1, n):
        fac_num.append(fac_num[i-1]*(i+1))
return fac_num[-1]

```

### **Output:**

```

fac1(40)
815915283247897734345611269596115894272000000000

```

This solution requires  $O(n)$  time. Note that this solution calculates the factorial of a number, and the result is used to find the factorial of a larger number. That is, memoization helps in reducing the complexity of the algorithm. Let us consider another problem to understand the concept.

## Longest common sub-sequence

This section aims to find the longest common sub-sequence, given two strings. For example, if the first string is as follows:

```
str1 = "ghijklmno"
```

and the second string is:

```
str2 = "gijo"
```

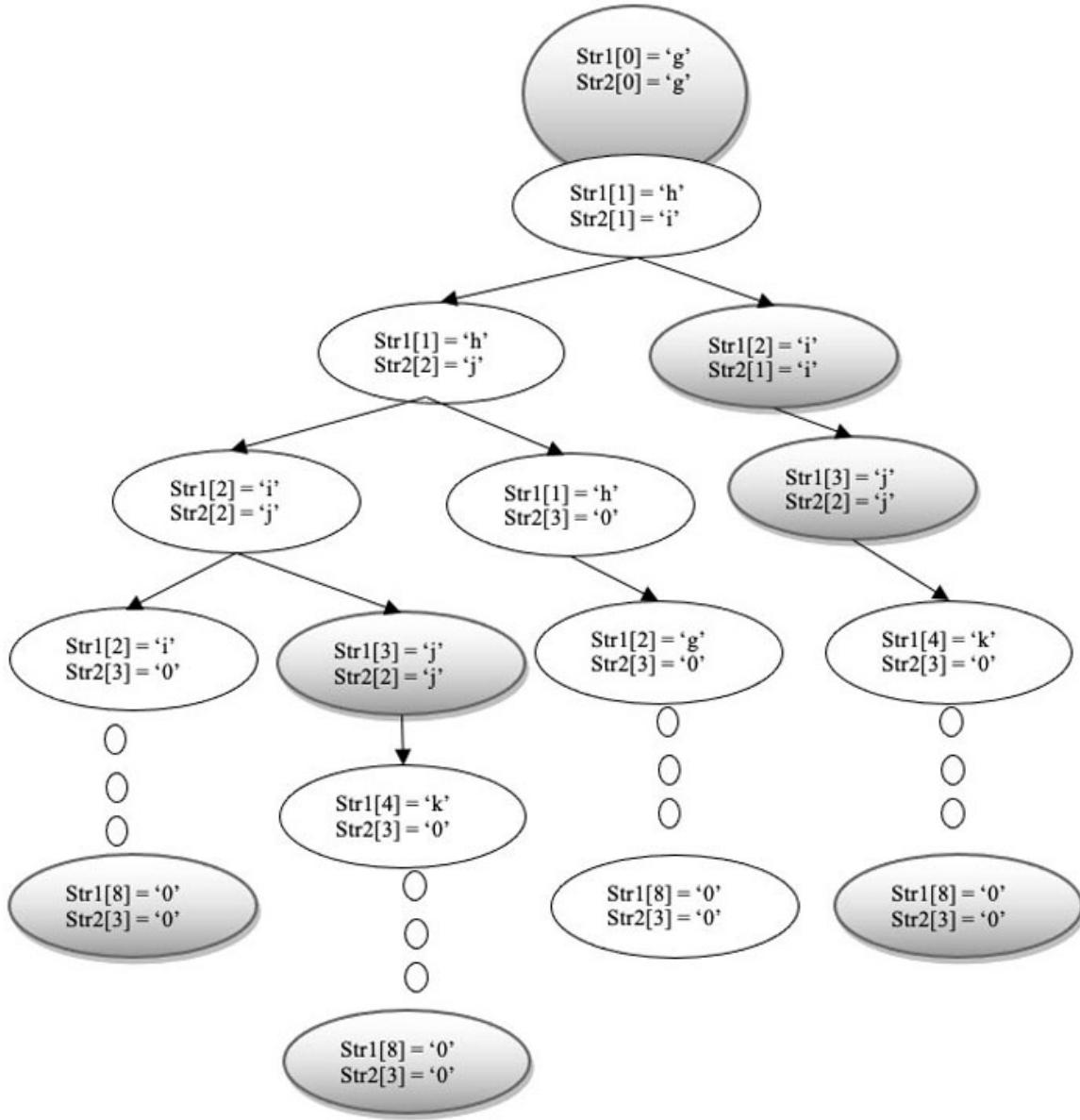
Then the longest common sub-sequence is “gijo”, as the alphabet “g” is at the 0th index in the first string, “i” is at the 2nd index, “j” at the 3rd, and “o” is at the 8th index. Note that the matching indices in the first string are {0, 2, 3, 8}, which is an increasing sequence. The sequence of matching indices should be ordered.

As such, the problem seems easy. Find all the common sub-sequences of the two strings and return the longest. Wait! Try and implement this solution, and you will realize that this solution has an exponential time complexity.

Another option can be to use backtracking. In backtracking, we start with the 0th index and check if the alphabets at this index match. If they match, we increment the value of indices in both the strings and proceed further, else we explore both options: that of incrementing the index of the first string, keeping that of the second same, and the other of incrementing the index of the second keeping that of the first same.

In [figure 2.3](#), the root node compares the character at the 0<sup>th</sup> index of the two strings. Since they are equal, the values of both i and j are incremented. Note that

$\text{str1}[1]$  is not the same as  $\text{str2}[1]$ ; therefore, two branches, one with  $i=1$  and  $j=2$ ; and the other with  $j=1$  and  $i=2$ , are created. This process continues till one of the strings ends. The nodes shown in the dark are those in which the characters match. Finally, the leaves (nodes at the last level) in which the two strings match can be backtracked to find the common sub-sequences and the longest common sub-sequence.



**Figure 2.3:** Longest common subsequence using backtracking

Note that the preceding approach is highly inefficient as a particular state is reached many times in the tree so formed. If we can store the required information in a table and use it later, we will arrive at the solution faster and more efficiently.

The memoization can be achieved by creating a two-dimensional array and filling it using the following procedure:

1. If the values at  $\text{str1}[i]$  and  $\text{str2}[j]$  are same then  $\text{LCS}[i, j]$  is one more than the value at  $\text{LCS}[i-1, j-1]$
2. Else, the value at  $\text{LCS}[i, j]$  is maximum of ( $\text{LCS}[i-1, j]$ ,  $\text{LCS}[i, j-1]$ )

Note that the first row and the first column of the table are all zeros (as one of the two strings is null).

For example, in the following case, the first string is “abcdefghi” and the second string is “cdgi”. The first row and the first column of the table are filled with zeros, and the rest are filled using the rules stated previously.

### Code:

```
import numpy as np
global string1
string1='abcdefghijklm'
global string2
string2='cdgi'
global table
table=np.zeros((len(string1)+1, len(string2)+1))
def LCS(i, j):
    if(string1[i-1]==string2[j-1]):
        table[i][j]= table[i-1][j-1]+1
    else:
        table[i][j]= max(table[i-1][j], table[i][j-1])
for i in range(1, len(string1)+1):
    for j in range(1, len(string2)+1):
        LCS(i, j)
print(table)
```

### Output:

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 1., 1., 1., 1.],
       [0., 1., 2., 2., 2.],
       [0., 1., 2., 2., 2.],
```

```
[0., 1., 2., 2., 2.],
[0., 1., 2., 3., 3.],
[0., 1., 2., 3., 3.],
[0., 1., 2., 3., 4.]])
```

Now, let us take it further. To find the largest common sub-sequence, identify the largest number in the last row ([table 2.1\(a\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.
i	0.	1.	2.	3.	4.

**Table 2.1(a):** Finding LCS using DP

Note that this number is not present at (previous row, same column) or (same row, previous column), so move to the (previous row, previous column) ([table 2.1\(b\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.

i	0.	1.	2.	3.	4.
---	----	----	----	----	----

**Table 2.1(b): Finding LCS using DP**

Note that this number is present at (previous row, same column), so move to the (previous row, same column) ([table 2.1\(c\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.
i	0.	1.	2.	3.	4.

**Table 2.1(c): Finding LCS using DP**

Note that this number (3) is not present at (previous row, same column) or (same row, previous column), so move to the (previous row, previous column) ([table 2.1\(d\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.
i	0.	1.	2.	3.	4.

**Table 2.1(d): Finding LCS using DP**

Note that this number (2) is present at (previous row, same column), so move to the (previous row, same column) ([table 2.1\(e\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.
i	0.	1.	2.	3.	4.

**Table 2.1(e): Finding LCS using DP**

Note that this number (2) is present at (previous row, same column) so move to the (previous row, same column) ([Table 2.1\(f\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.
i	0.	1.	2.	3.	4.

**Table 2.1 (f): Finding LCS using DP**

Note that this number (2) is not present at (previous row, same column) or (same row, previous column), so move to the (previous row, previous column) ([table 2.1 \(g\)](#)).

		c	d	g	i
	0.	0.	0.	0.	0.
a	0.	0.	0.	0.	0.
b	0.	0.	0.	0.	0.
c	0.	1.	1.	1.	1.
d	0.	1.	2.	2.	2.
e	0.	1.	2.	2.	2.
f	0.	1.	2.	2.	2.
g	0.	1.	2.	3.	3.
h	0.	1.	2.	3.	3.
i	0.	1.	2.	3.	4.

*Table 2.1(g): Finding LCS using DP*

The reader is expected to figure out why the common substring is “cdgi” (check the point at which we moved to the upper diagonal cell).

## Conclusion

The chapter introduced various methods of solving problems, including divide and conquer, Greedy approach, dynamic programming, and backtracking. The methods have been explained using some interesting problems such as coin changing, identifying fake coin, longest common sub-sequence, and so on. The exercises contain more such problems. The upcoming chapter introduces the theory and problems related to arrays. The chapter will help you to march toward competitive programming.

## Multiple choice questions

1. What is the complexity of linear search?

- a. O(n)
- b. O( $n^2$ )

- c.  $O(2^n)$
- d. None of the above

**2. What is the complexity of binary search?**

- a.  $O(n)$
- b.  $O(\log n)$
- c.  $O(2^n)$
- d. None of the above

**3. Refer to Question 2(b) of the following exercise. The procedure is called Ternary Search. What is its complexity?**

- a.  $O(n)$
- b.  $O(\log n)$
- c.  $O(2^n)$
- d. None of the above

**4. In finding the minimum from a given list, devise an algorithm using Divide and Conquer. What is the complexity?**

- a.  $O(n)$
- b.  $O(\log n)$
- c.  $O(2^n)$
- d. None of the above

**5. Refer to Question 3(a) of the exercise. In finding the  $n^{\text{th}}$  Fibonacci term, what is the complexity, using this approach?**

- a.  $O(n)$
- b.  $O(\log n)$
- c.  $O(2^n)$
- d. None of the above

**6. What is the complexity of finding the  $n^{\text{th}}$  Fibonacci term using recursion?**

- a.  $O(n)$
- b.  $O(\log n)$

- c.  $O(2^n)$
- d. None of the above

**7. Which of the following uses memoization?**

- a. Dynamic Programming
- b. Divide and Conquer
- c. Backtracking
- d. None of the above

**8. Which of the following uses division of problems into sub-problems?**

- a. Dynamic programming
- b. Divide and conquer
- c. Backtracking
- d. None of the above

**9. Which of the following uses strategy of choosing best option at that step?**

- a. Dynamic programming
- b. Divide and conquer
- c. Backtracking
- d. Greedy approach

**10. The longest common subsequence uses which of the following?**

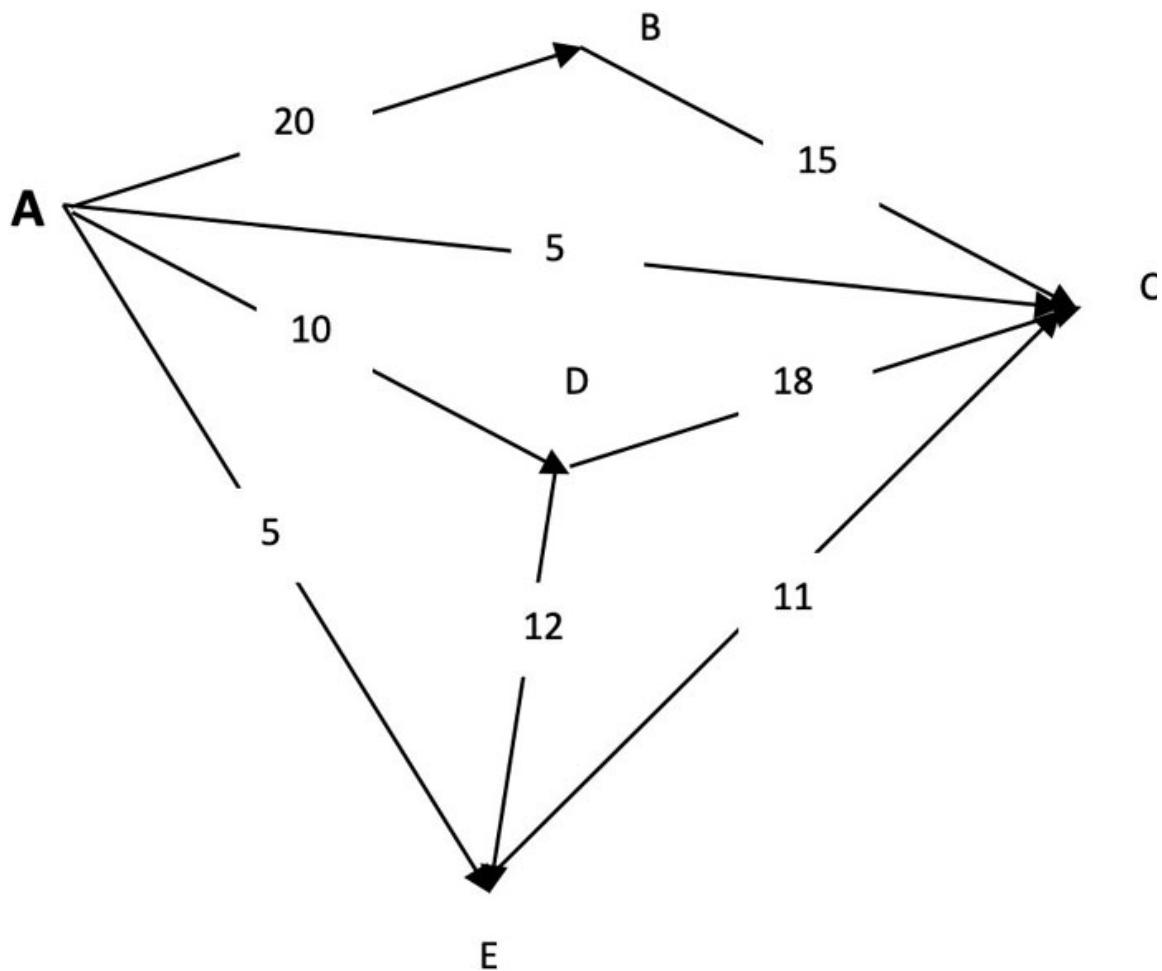
- a. Dynamic programming
- b. Divide and conquer
- c. Backtracking
- d. None of the above

## Programming/application

1. Consider the graph shown in [figure 2.4](#). The vertices of the graph are the cities, and the weight of each edge is the cost incurred to travel from one vertex to the other in 1,000's. "A" is the source vertex.
  - a. Find all possible paths from A to C and the corresponding costs. Find the path with minimum cost and that with the maximum cost.

- b. Now, start with the source node (in this case, A), find all adjacent edges, and then proceed through the edge with the minimum cost (provided it does not form a cycle). Repeat the process of reaching the next vertex. In reaching the destination (in this case, "C"), note the vertex in the path and print the sequence of vertices traversed while traveling from the source to the destination. Design an algorithm to find the tree (Spanning)
- c. In the preceding approach, the best possible option is considered at every point. Which approach is followed in designing the solution?
- d. Can the same approach be followed for finding the longest path?

The problem has also been discussed in graphs.



*Figure 2.4: Graph for Problem 1*

2. Consider a list of sorted numbers. To search for a number in this list, look at the first element, the last element, and the element in the middle

**of the list. If you can find what you are looking for, print the position, or else either move to the left half or the right half depending upon whether the number to be searched is less than that in the middle or bigger than the middle.**

- a. This is called binary search. Implement the preceding algorithm and find its complexity.
  - b. In the preceding question, divide the list into three parts and apply the process. What will be the complexity of the algorithm?
- 3.
- a. Write a recursive procedure to find the  $n^{\text{th}}$  Fibonacci term.
  - b. Draw a tree showing the calculation of the  $7^{\text{th}}$  term.
  - c. What is the complexity of this algorithm?
  - d. Now develop an iterative procedure to accomplish the preceding task.
  - e. Find the complexity of the preceding algorithm.
  - f. Which of the preceding techniques uses dynamic programming?
4. **The  $n^{\text{th}}$  term of a sequence can be calculated using the following formula:**
- $$\begin{aligned}f(1) &= 1, \\f(2) &= 1, \\f(n) &= 2 \times f(n - 1) + 3 \times f(n - 2)\end{aligned}$$
- a. Write a recursive procedure to find the  $n^{\text{th}}$  term.
  - b. Draw a tree showing the calculation of the  $4^{\text{th}}$  term.
  - c. What is the complexity of this algorithm?
  - d. Now develop an iterative procedure to accomplish the preceding task.
  - e. Find the complexity of the preceding algorithm.
  - f. Which of the preceding techniques uses dynamic programming?
5. Write a Greedy algorithm to solve the Knapsack problem. For reference, you may refer to reference 2.
6. In a plane, there are  $n$  points. Find the nearest pair using divide and conquer.
7. You are provided with a  $N \times N$  chessboard and need to place  $N$  Queens, one in each row. The queens should be placed such that none of them is in a

position to attack the other. This is called N-Queens' problem. Write an algorithm to solve the N Queen's problem using backtracking.

8. In the preceding procedure, can you make the backtracking solution more effective?

## Further references

- <https://www.cs.princeton.edu/~wayne/kleinberg-tardos/>
- <https://www.cse.iitd.ac.in/~ssen/csl356/root.pdf>
- <http://web.stanford.edu/class/archive/cs/cs161/cs161.1176/>

## **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:

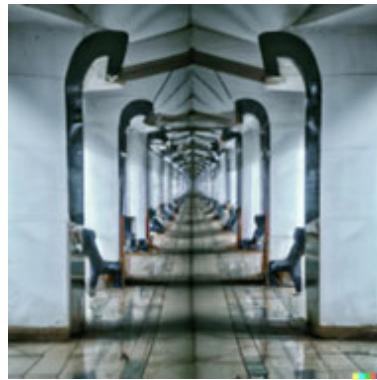
<https://discord.bpbonline.com>



## CHAPTER 3

### Recursion

Imagine a room having its walls covered with mirrors. How many images will you see? Now, consider the picture having two inclined mirrors and an object in between ([figure 3.1](#)). Do you find any commonality between your images in the room mentioned above and the picture? In both cases, an image acts as an object repeatedly.



*Figure 3.1: Parallel mirrors generate infinite images*

In this chapter, we will try to solve the problems by reducing them to their smaller versions. We will design a function that solves the problem by calling it within itself. This is not only a powerful instrument, but it also teaches us a lot about control flow.

The technique described in this chapter will help you to solve many problems. As a matter of fact, you need to know recursion to be able to use backtracking and some other assorted problem-solving techniques. In recursion, we write a function in terms of itself and state the base case. The mechanism of evaluation will become clear in the first example.

The chapter begins with exponentiation and builds the case for using the technique. This is followed by an informed discussion about the two most common problems: The Tower of Hanoi and the Rabbit Problem. We will then move to the generation of binary numbers, followed by problems

related to lists and numbers. As stated earlier, the chapter is important if you want to become an accomplished programmer.

## Structure

In this chapter, we will cover the following topics:

- Exponentiation
- Tower of Hanoi
- Rabbit problem
- Generating binary numbers
- Lists
- Numbers

## Objectives

This chapter introduces the fascinating concept of recursion. The ideas have been explained using famous problems, which not only demonstrate the ability of recursion to solve the problem in the most elegant way but also present it as the most natural solution to some of the problems.

The reader will be able to use recursion to solve some important problems after reading this chapter. This chapter will also form the basis of divide and conquer and dynamic programming. So let us dive into recursion.

## Exponentiation

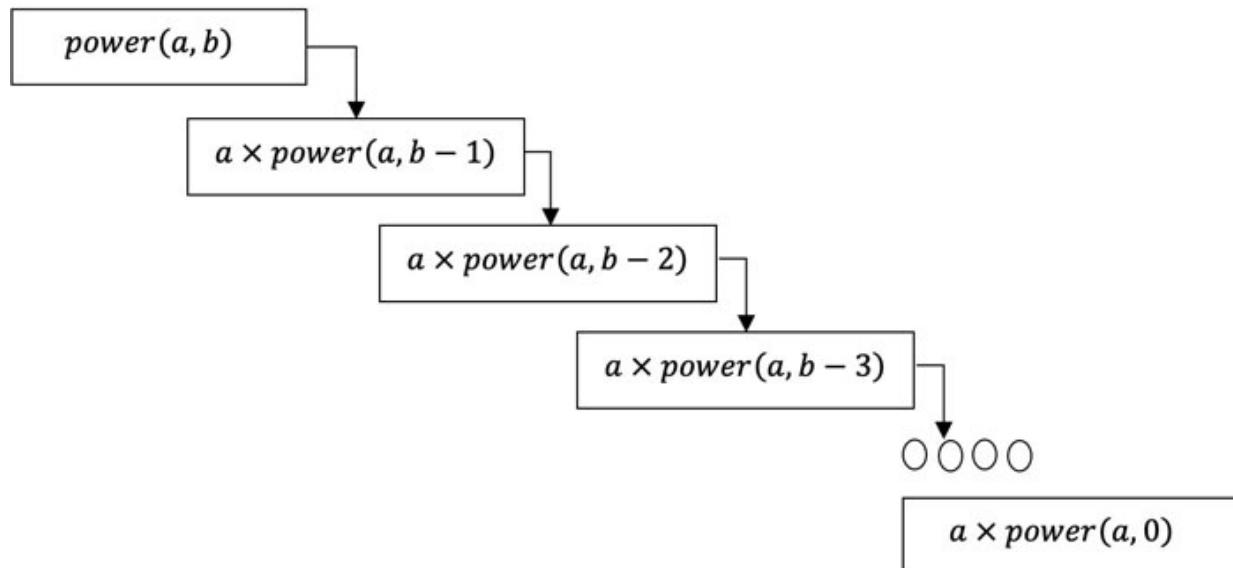
Recursion requires calling a function in itself and specifying the base case. For example, in calculating the power of a number to another, the following strategy can be applied.

$$\begin{aligned}a^b &= a \times a \times a \dots \times a (b \text{ times}) \\&= a \times (a \times a \dots \times a (b - 1 \text{ times})) \\&= a \times a^{b-1}\end{aligned}$$

That is if the function *power* (*a*, *b*) returns  $a^b$ , then the same result can be obtained by calling the same function with a reduced parameter (*a*, *b* - 1) and multiplying with the result so obtained.

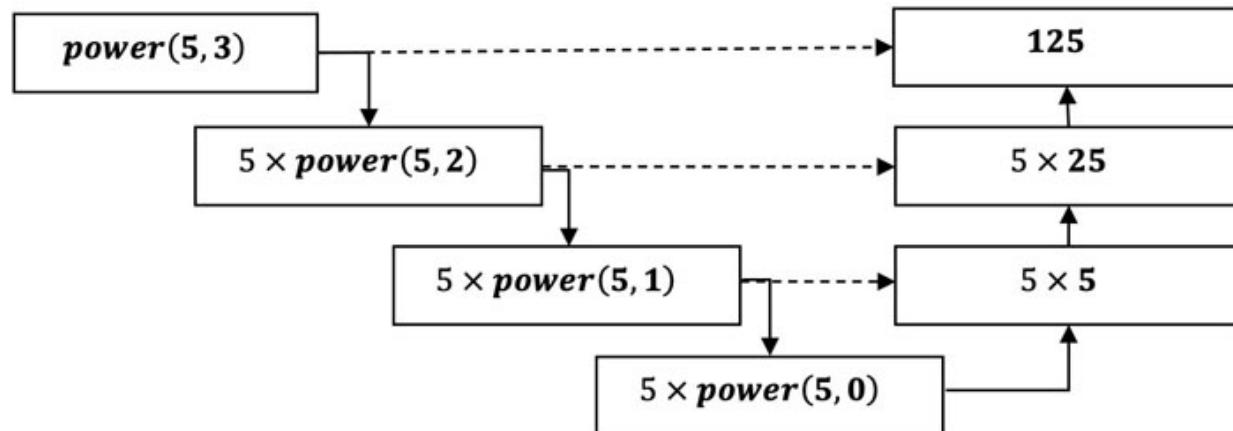
$$\text{power}(a,b) = a \times \text{power}(a, b-1)$$

Also note that if there is no base case, the depth of the recursion tree grows with the calls, thus consuming all the memory allocated to the function (Stack!). So, it needs to be told when to stop. In this case, the base case is one in which the value of  $b$  becomes 0 and  $\text{power}(a, b)$  returns a 1 ([figure 3.2](#)).



*Figure 3.2: Calculating using recursion*

The last instance of the function is evaluated first ( $\text{power}(a, 0)$ ) and returns 1 to the last but one instance. That is, it follows **Last In First Out (LIFO)**. [Figure 3.3](#) depicts calculating  $\text{power}(5, 3)$ . Note that the bold portions of the left cells are replaced by the corresponding bold ones on the right in the bottom-up fashion.



**Figure 3.3:** Calculating power (5, 3)

The code for calculating, using recursion as follows:

**Code:**

```
1. def power(a, b):  
2.     if(b==0):  
3.         return 1  
4.     else:  
5.         return (a*power(a, b-1))  
6. power(5,3)
```

**Output:**

**125**

Likewise, the value of the factorial can be calculated using the following formula:

$$fac(n) = n \times fac(n - 1)$$

and

$$fac(1) = 1$$

Note that the function factorial( $n$ ) calls factorial ( $n-1$ ), and the base case, factorial(1), returns 1. The following code calculates the factorial of a number using recursion.

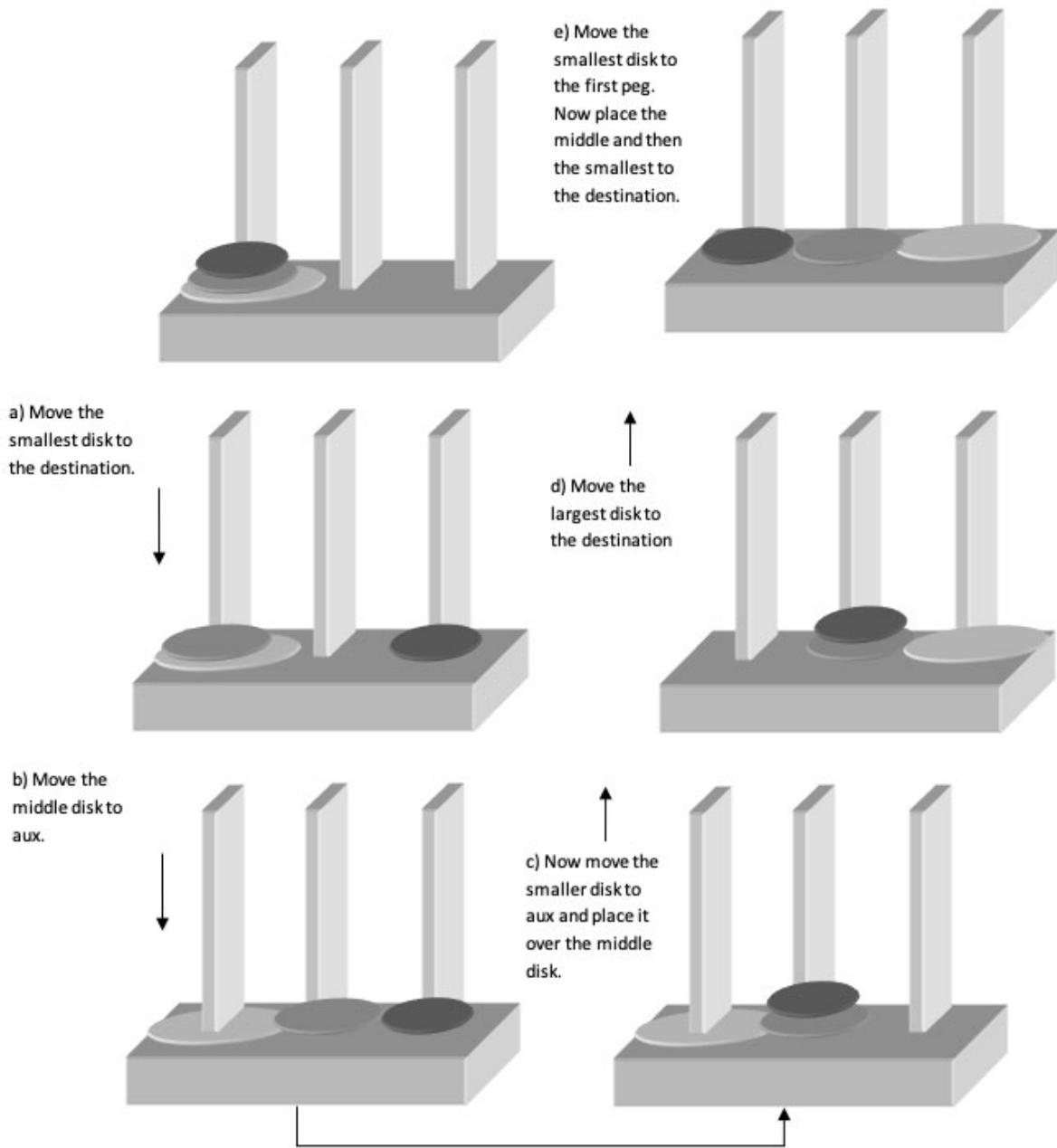
**Code:**

```
1. def factorial(n):  
2.     if(n==1):  
3.         return 1  
4.     else:  
5.         return (n*factorial(n-1))  
6. factorial(6)
```

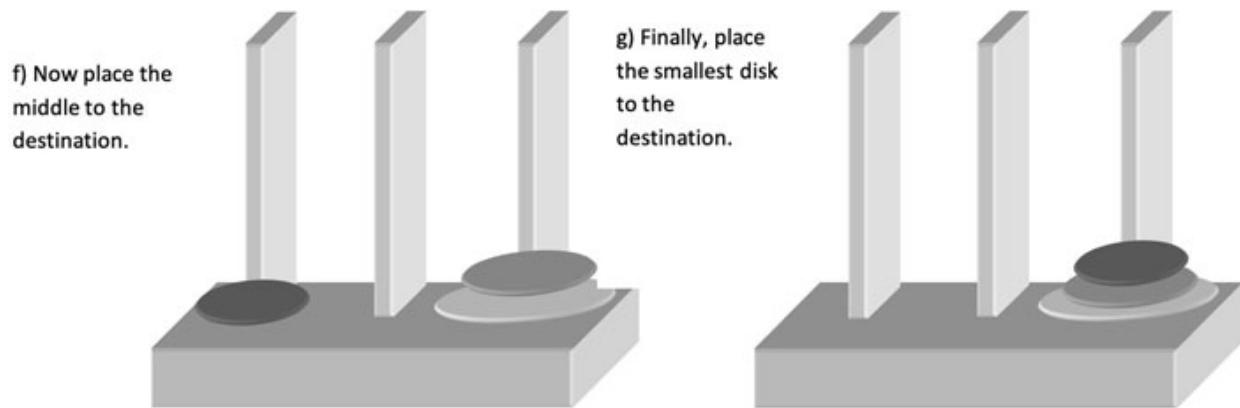
**Output:**

## Tower of Hanoi

Having seen a few simple problems, let us move to the fascinating problem “The Tower of Hanoi”. There are three pegs, and  $n$  disks are placed one over another in the first peg. The disks are placed in decreasing order of their radius. You are required to place the disks to the third peg using an auxiliary disk (aux1) in such a way that in no move, a bigger disk should be above the smaller one. [Figure 3.4](#) shows the process of transferring three disks from the source peg to the destination peg:



**Figure 3.4(a):** Solving the tower of Hanoi for  $n=3$  Steps (a)–(e)



**Figure 3.4(b):** Solving the tower of Hanoi for  $n=3$  Steps (f)–(g)

Note that for three disks in the source peg, seven moves are needed. We can easily establish that for  $n$  pegs  $2^n - 1$  moves are required to transfer the disks from the source to the destination.

According to a legend, 64 such disks are placed in a temple, and someone is entrusted with the responsibility of putting them in the destination peg. The complete process will take  $2^{64} - 1$  unit of time. If he takes a second to transfer a disk from one peg to another, the process will take 584942417355.072 years.

Now, let us develop a recursive solution to this problem. If there is a single disk in the source peg, then it needs to be moved from source to destination. In all other cases, move  $(n-1)$  disks from source to aux, then move the last disk to the destination, and then move the  $(n-1)$  disks from aux to destination. The following code implements the concept:

### Code:

```

1. def TowerOfHanoi(num, sour, dest, aux):
2.     if num==1:
3.         print('Move disk 1 from', sour, 'to ', dest)
4.         return
5.     else:
6.         TowerOfHanoi(num-1, sour, aux, dest)
7.         print('Move disk', num, 'from', sour, 'to', dest)
8.         TowerOfHanoi(num-1, aux, dest, sour)

```

```
9. TowerOfHanoi(4, 'A', 'C', 'B')
```

### Output:

```
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
Move disk 4 from A to C
Move disk 1 from B to C
Move disk 2 from B to A
Move disk 1 from C to A
Move disk 3 from B to C
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
```

## Rabbit problem

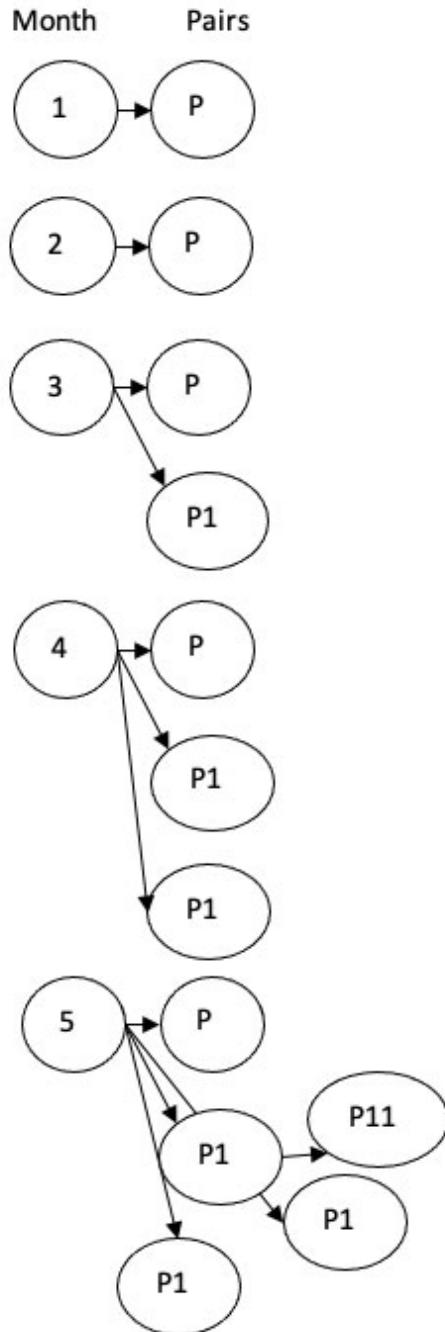
As explained earlier, recursion means calling a function in itself. There are two important ingredients of a program that uses recursion: the base case and expressing a function in terms of itself. To understand this concept, consider the famous “Rabbit Problem”, wherein a pair of rabbits (P1) does not breed for the first two months, after which it produces a pair of rabbits each month. That is, in the first two months, there is a single pair of rabbits. In the third month, the pair produces another pair (P11). In the forth month, the first pair produces a pair (P12), thus, three pairs in total. In the fifth month, the first pair produces another pair (P13), and the second pair also produces a pair (P111), thus, five pairs in total. [Figure 3.5](#) depicts the progression.

The variation of the number of pairs of rabbits with the month is, therefore,  
1, 1, 2, 3, 5, 8, 13, ...

Note that the first two terms of the preceding sequence are 1, and each term afterward is the sum of the previous two terms. That is,

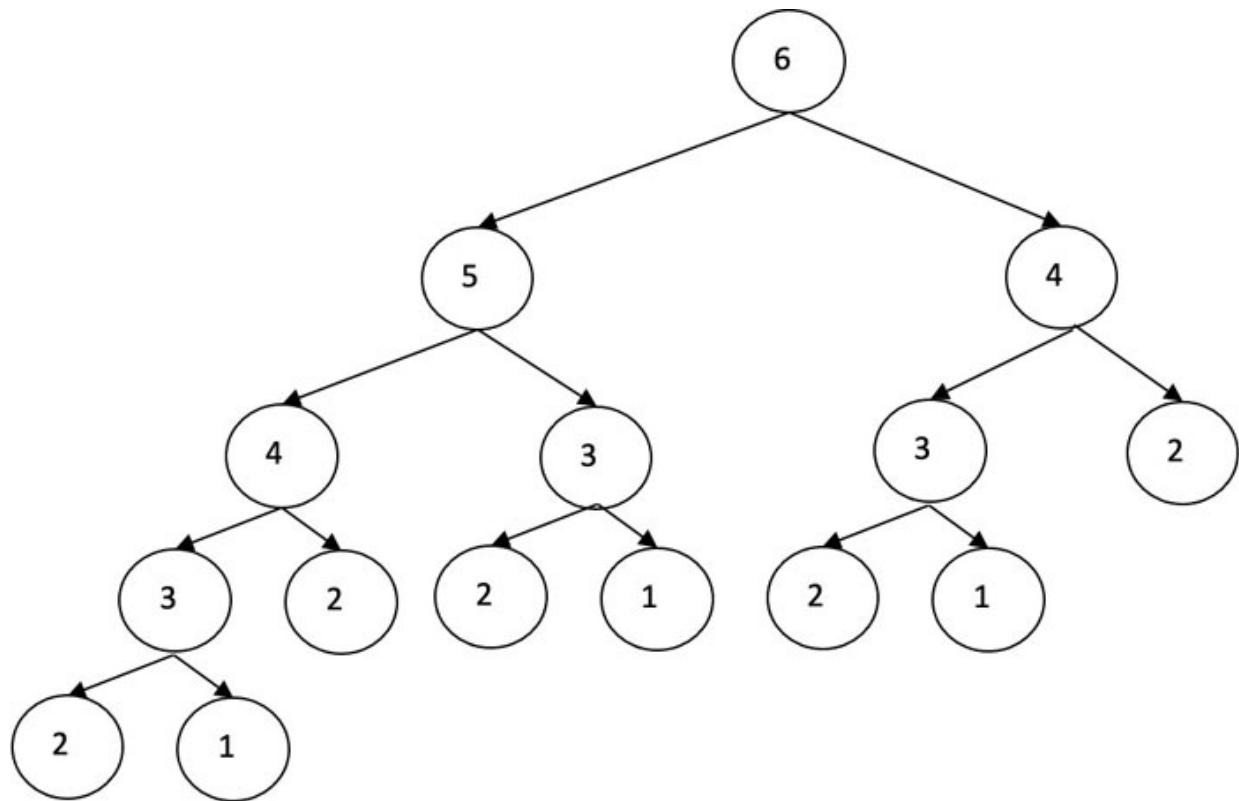
$$f(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ f(n - 1) + f(n - 2), & \text{otherwise} \end{cases}$$

Note the last part of the preceding function, in which  $f(n)$  is expressed in terms of itself. The values of the function at the first two time instances are 1 and 1 each. [Figure 3.6](#) shows the calculation of Fibonacci (6), which requires Fibonacci (5) and Fibonacci (4). Fibonacci (5) requires Fibonacci (4) and Fibonacci (3), and Fibonacci (4) requires Fibonacci (3) and Fibonacci (2).



**Figure 3.5:** The rabbit problem

Each node in the following diagram ([figure 3.6](#)) shows the number of rabbits in a particular month, and the left and the right child show the number of rabbits in the previous two months. Note that the root ( $\text{fib}(6)$ ) is the sum of  $\text{fib}(5)$  and  $\text{fib}(4)$ .



*Figure 3.6: Finding Fibonacci (6)*

Well, this seems good. Wait! There are two problems in the preceding approach, some of the terms are calculated a lot of times, and this method takes a lot of memory. Observe that the first code takes 0.60025 seconds for finding Fibonacci (30), whereas the one that does not use recursion takes 0.0009968 seconds for finding Fibonacci (100), which is amazing!

### Code 1:

```

1. def fibonacci(n):
2.     if n==1:
3.         return 1
4.     elif n==2:
5.         return 1
6.     else:
7.         return (fibonacci(n-1)+fibonacci(n-2))
8. tic=time.time()

```

```
9. fibonacci(30)
10. toc=time.time()
11. elapsed =toc-tic
12. print(elapsed)
```

**Output:**

**0.6002504825592041**

**Code 2:**

```
1. def fibonacci1(n):
2.     L=[]
3.     L.append(1)
4.     L.append(1)
5.     for i in range(2,n):
6.         L.append(L[i-1]+L[i-2])
7.     return L[n-1]
8. tic=time.time()
9. fibonacci1(100)
10. toc=time.time()
11. elapsed =toc-tic
12. print(elapsed)
```

**Output:**

**0.0009968280792236328**

This is because the complexity of Fibonacci, which uses recursion, is  $O((\phi)^n)$ , where  $\phi = (\sqrt{5}+1)/(\sqrt{5}-1)$ , whereas the complexity of Fibonacci1 is  $O(n)$ . This is one of the flip sides of using recursion.

## Generating binary numbers

The binary number system has two digits: 0 and 1. The single-digit numbers in this system are 0 and 1 in that order. The two digits numbers are formed by inserting a zero before these and then 1. That is,

One-bit binary numbers

0

1

Two-bit binary numbers

**00**

**01**

**10**

**11**

Likewise, three-bit binary numbers are formed by inserting a 0 before the two-bit numbers and then a 1. That is,

**000**

**001**

**010**

**011**

**100**

**101**

**110**

**111**

In the same way, an  $n$ -bit binary number can be formed by inserting a zero before all the  $(n-1)$  bit numbers and then a one before the  $(n-1)$  bit numbers. The following code implements this logic. The code asks the user to enter the value of  $n$  and inserts 0 and 1 at the beginning of the  $(n-1)$  bit numbers formed by calling the same function with reduced parameters. Note that one-bit binary numbers constitute the base case in the following example.

### Code:

```
1. def generate(n):  
2.     if n==0:  
3.         return ''  
4.     elif n==1:
```

```

5.         return ['0', '1']
6.     else:
7.         L=[]
8.         for i in generate(n-1):
9.             L.append('0'+i)
10.        for i in generate(n-1):
11.            L.append('1'+i)
12.        return L
13. generate(4)

```

### **Output:**

```
[ '0000',
  '0001',
  '0010',
  '0011',
  '0100',
  '0101',
  '0110',
  '0111',
  '1000',
  '1001',
  '1010',
  '1011',
  '1100',
  '1101',
  '1110',
  '1111']
```

Taking this discussion forward, let us consider an example. Hari, an intelligent scientist working in Computational Neurology, is an Indian version of Sheldon Cooper. He has a thing for numbers and decides to quit the decimal system used by Homo Sapiens. He comes up with “Hexal” numbers consisting of seven digits from 0 to 5. Use the preceding developed

logic and help him to generate  $n$ -digit “Hexal” numbers. Have a look at the following code and try decoding the mechanism:

### Code:

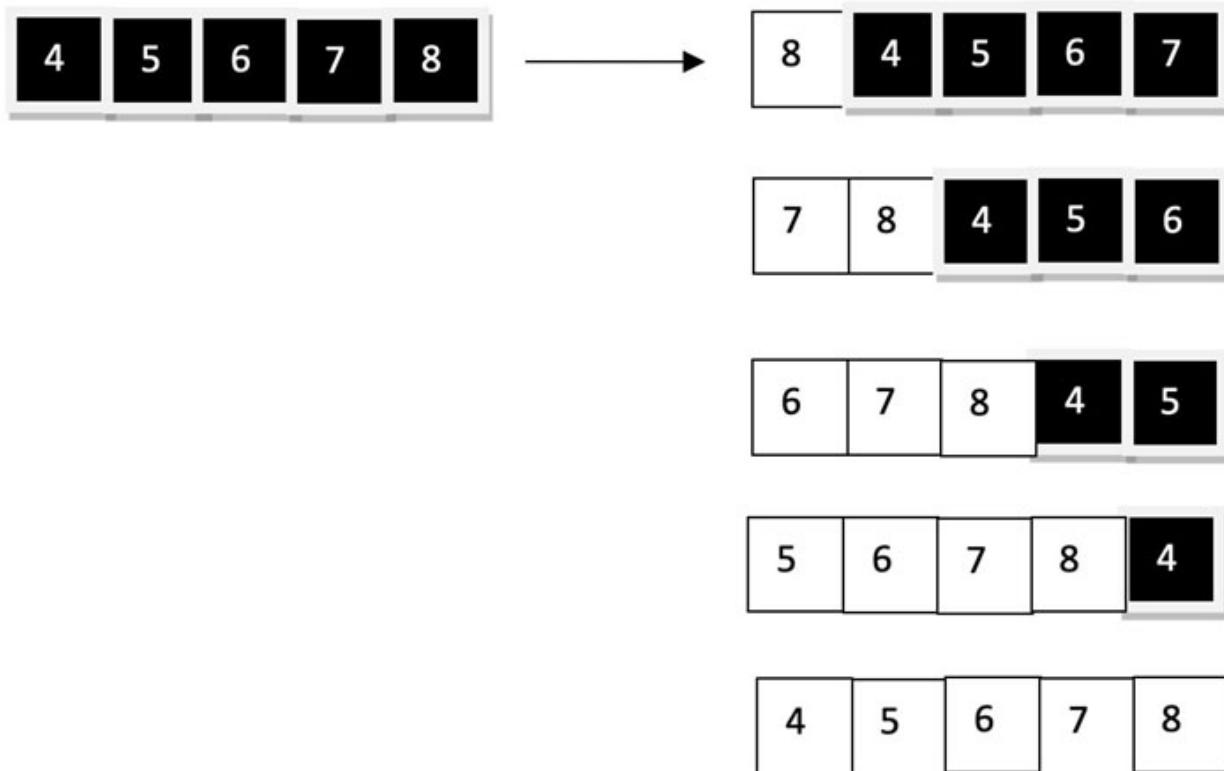
```
1. def generate1(n):
2.     if n==0:
3.         return ''
4.     elif n==1:
5.         return ['0', '1', '2', '3', '4', '5']
6.     else:
7.         L=[]
8.         for i in generate1(n-1):
9.             L.append('0'+i)
10.        for i in generate1(n-1):
11.            L.append('1'+i)
12.        for i in generate1(n-1):
13.            L.append('2'+i)
14.        for i in generate1(n-1):
15.            L.append('3'+i)
16.        for i in generate1(n-1):
17.            L.append('4'+i)
18.        for i in generate1(n-1):
19.            L.append('5'+i)
20.    return L
```

Well, you can generate any number system using the preceding logic. As a matter of fact, you will be doing so in the exercises.

## Lists

Consider reversing the order of elements in a given list. The task can be accomplished by taking out the last element and inserting it at the beginning of the list obtained by sending the list, except for the last element, in the same function. The procedure is depicted in [figure 3.7](#).

In the figure, the black portion of the list is passed as an argument to the function, and the last element is inserted in the beginning:



**Figure 3.7:** Reversing a list using recursion. The black portion is passed as an input to the function.

5  
6  
7  
8  
8  
4  
5  
6  
7  
7

```
7  
8  
4  
5  
6  
6  
7  
4  
5  
8  
5  
6  
4  
7  
8  
4  
5  
6  
7  
8
```

The program for reversing a list using recursion is as follows:

**Code:**

```
1. def reverse(L):  
2.     if(len(L)==1):  
3.         return L  
4.     else:  
5.         return([L[-1]]+reverse(L[:-1]))  
6. L = [1, 2, 3, 4, 5]
```

7. reverse(L)

**Output:**

[5, 4, 3, 2, 1]

The maximum element from a list can also be found in the same way. The first element can be compared with that obtained by passing the list, except for the first element, in the same function. The code implementing the logic is as follows:

**Code:**

```
1. def find_max(L):  
2.     if(len(L)==1):  
3.         return L[0]  
4.     else:  
5.         rest=find_max(L[1:])  
6.         return(L[0] if L[0]>rest else rest)  
7. find_max([2,3,4,5,6])
```

**Output:**

6

In the same way, the minimum element can be found.

## Numbers

Having learned recursion, let us play with numbers. We start with a simple problem (?) that of obtaining a number by reversing the order of digits of the given number. To accomplish the task, we first design a function that finds the number of digits of a given number. We use this function to find the number of digits of the number obtained by passing num/10 in the function and adding 10 to the power of d to that number. The code is as follows:

**Code:**

```
1. def digits(num):  
2.     d=1
```

```

3.     while(num//10!=0):
4.         d+=1
5.         num=num//10
6.     return d
7. def rev_number(num):
8.     if(num//10==0):
9.         return num
10.    else:
11.        d=digits(rev_number(num//10))
12.        num1=(num%10)*(10**d)+rev_number(num//10)
13.    return num1
14. rev_number(1234)

```

## **Output:**

**4321**

Well, this is just the beginning! You will be required to find the maximum digit, the sum of digits of a given number, and so on, in the exercises given at the end of this chapter.

## **Conclusion**

Recursion helps us to divide the problem into smaller problems of the same kind. It should have the base case(es) and a function expressed in terms of itself. This chapter introduced recursion. The chapter discussed the solutions and the implementations of some of the most important problems in recursion. The reader is advised to attempt the exercises before proceeding any further. You will design an efficient recursive solution to the exponentiation, generate hexadecimal numbers, and attempt a graph-based problem in the exercises. We have opened the door to the magical world of recursion. Keep in mind that this world has fascinating charms but, at the same time, may entice you into deadly traps. The upcoming chapter discusses arrays and uses the concepts studied so far to solve some interesting problems.

## Multiple choice questions

- 1. Which of the following can be solved using recursion?**
  - a. Finding the factorial of a number
  - b. Finding  $a$  to the power of  $b$
  - c. Reversing a number
  - d. All of the above
- 2. Which of the following is the more efficient implementation of Fibonacci?**
  - a. Iterative solution
  - b. Recursive solution
  - c. Both are equally good
  - d. Cannot compare
- 3. Which of the following can be solved using recursion?**
  - a. Finding the maximum from a given list
  - b. Reversing a list
  - c. Finding the minimum from a given list
  - d. All of the above
- 4. Which of the following generates smaller and more elegant code for finding the factorial of a given number?**
  - a. Iterative solution
  - b. Recursive solution
  - c. Both are equally good
  - d. Cannot compare
- 5. If the value of the  $n$ th term is generated using the following formula:**

$$f(n) = \begin{cases} f(n-1) + f(f-2) + f(n-3) \\ < \text{some value} >, \text{for } n = x? \end{cases}$$

What should be the value of  $x$ ?

- a. 1
- b. 2
- c. 3
- d. None of the above

**6. Which of the following is true with regard to recursion?**

- a. A function must be expressed in terms of itself
- b. The base case (es) must be specified
- c. Both
- d. None of the above

**7. Can we use recursion in every case?**

- a. Yes
- b. No

**8. Does recursion always reduce complexity?**

- a. True
- b. False

**9. Which of the following is used in backtracking?**

- a. Recursion
- b. Iteration
- c. Both
- d. None of the above

**10. Recursion uses**

- a. Stacks
- b. Queues
- c. Plex
- d. None of the above

## Programming

### **Level 0:**

1. Find the factorial of a number using recursion.
2. Find  $a^b$  using recursion. [Use  $a^b = a \times a^{b-1}$ ]
3. Find the sum of first n natural numbers using recursion.
4. Find the first  $n$  terms of the Fibonacci series using recursion.
5. Find the maximum element from a list using recursion.
6. Find the sum of elements of a given list using recursion.
7. Check if a given number is a palindrome using recursion.

### **Level 1:**

1. Find the sum of digits of a given number using recursion.
2. Obtain a number by reversing the order of digits of a given number.
3. Find the Greatest Common Divisor of two numbers using recursion.
4. Find the Lowest Common Multiple of two numbers using the preceding result.
5. Given a list of lists, find the list having a minimum sum.

### **Level 2:**

1. Generate n digit hexadecimal number using recursion.
2. Refer to Question 2, and devise a solution to the problem having lesser complexity. You may use the fact that

$$a^b = \begin{cases} (a^{b/2})^2, & \text{if } b \text{ is even} \\ a \times (a^{(b-1)/2})^2, & \text{if } b \text{ is odd} \end{cases}$$

### **Level 3:**

1. Given a two-dimensional array of 0's and 1's. Find the largest string of consecutive ones. These consecutive ones may be present horizontally, vertically, or diagonally.

**For example, the largest string of 1's is shown in bold in the following arrays.**

*Example 1:*

<b>1</b>	<b>1</b>	<b>1</b>	0
0	0	0	1
1	0	1	0

*Example 2:*

<b>1</b>	0	1	0
<b>1</b>	0	0	1
<b>1</b>	0	1	0

*Example 3:*

<b>1</b>	0	1	0
0	<b>1</b>	0	1
0	0	<b>1</b>	0

## Further references

- <https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1178/lectures/7-IntroToRecursion/7-IntroToRecursion.pdf>
- <http://www.cs.utah.edu/~germain/PPS/Topics/recursion.html>
- <https://www.csee.umbc.edu/~chang/cs202.f98/readings/recursion.html>

## **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

## Arrays

Consider a use case where you have to deal with the salary of a thousand employees of a company. You need to find the average salary, the median, the standard deviation, and so on. How will you begin?

The first step will be to store these values in a data structure. Note that all the salaries must be of the same type (for example, float), and it will be advantageous if the data structure stores them at consecutive memory locations. This will help us in accessing the elements easily and efficiently. This chapter introduces a data structure that will help you to deal with such problems. The data structure is an array. An array stores the same type of elements at consecutive memory locations.

### Structure

In this chapter, we will cover the following topics:

- Indexing
- Memory map of 2-D arrays
- Insertion and deletion in an array
- Problems

### Objectives

After reading this chapter, the reader will be able to understand the concept of arrays, and the memory map of 2-D arrays. You will also learn how to carry out insertion and deletion in an array, and solve problems using arrays

### Introduction

Arrays are used to store multiple values of the same type at consecutive memory locations. They store a fixed number of elements declared initially. For example, an integer array having size 5 can store a maximum of 5

integers. In Python, you can declare an array using the **array** module or can create more sophisticated arrays using the Numeric Python (**numpy**) package. The latter support vectorized operations and will also help you in Data Science and Machine Learning.

A list in Python can be converted into a **numpy** array by passing the list in the **np.array** function, where *np* is an alias for **numpy**. The following code creates a list having elements 9, 7, 5, 3, and 1. The list is then passed in the **np.array** function and gets converted into a **numpy** array called **arr1**. On printing the type of **arr1**, **numpy.ndarray** is shown, indicating it is an *n*-dimensional **numpy** array. Note that an element of this array can be accessed using the notation **<name of the array>[index]**. The index of the first element is 0, that of the second element is 1, and so on. So, if an array has **n** elements, the index of the last element is **(n-1)**.

If **arr1** is a numpy array, then **arr1.shape** gives its size. For a 1-D array, the shape would be of the form (<number of elements>). In the following code, the shape of **arr1** is, therefore, (5). So, the last element is at the 4<sup>th</sup> index. Hence, to access it we can write **arr1[shape[0] - 1]**, which is **arr1[5-1] = arr[4] = 1**.

### Code:

1. L=[9, 7, 5, 3, 1]
2. arr1=np.array(L)
3. print(arr1)
4. print(type(arr1))
5. print(arr1[0])
6. print(arr1[1])
7. print(arr1.shape)
8. print(arr1[arr1.shape[0]-1])

### Output:

```
[9 7 5 3 1]
<class 'numpy.ndarray'>
9
```

```
7  
(5, )  
1
```

It may be stated here that the slicing and negative indexing work as usual in **numpy** arrays. One can create a two-dimensional array in numpy by passing a list of lists in the **numpy.array** function. The following code creates a list of lists and passes it in **numpy.array** to create a 2-D array called **arr\_2d**. The shape of a 2D array is of the form (<number of rows>, <number of columns>). The elements of this array can be accessed using the **<name of the array>[ row index, column index]** notation. In Python, you can access a row of a 2D array by simply writing **<name of the array>[row index]**. So **arr\_2d[0]** can be used to access the first row of **arr\_2d**.

1. `L1=[[1,2,3], [4, 5, 6],[7, 8, 9], [10, 11, 12], [13, 14, 15]]`
2. `arr_2d= np.array(L1)`
3. `print(arr_2d)`
4. *#Shape of arr\_2d. This array has 5 rows and 3 columns, so the shape is (5, 3)*
5. `print(arr_2d.shape)`
6. *#Indexing arr\_2d[2, 1] will show the element at the third row and second column*
7. `print(arr_2d[2,1])`
8. *#To access the first row*
9. `print(arr_2d[0])`

### Output:

```
[[ 1   2   3]  
 [ 4   5   6]  
 [ 7   8   9]  
 [10  11  12]  
 [13  14  15]]  
(5, 3)
```

Let us now have a look at the memory map of a 2-D array.

## Memory map

A 2-D array can be stored in the memory in the row-major or column-major format. In the row-major format, the elements are stored row-wise as a 1-D array. For example, the elements of `arr_2d` are stored in the row-major format as follows:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

In the column-major format, on the other hand, the elements are stored column-wise as a 1-D array. For example, the elements of `arr_2d` are stored in the column-major format as follows:

1, 4, 7, 10, 13, 2, 5, 8, 11, 14, 3, 6, 9, 12, 15

## **Address in row-major**

If the number of rows in a given 2-D array is  $m$ , and the number of columns is  $n$ . Then the location of the  $(i, j)^{\text{th}}$  element in the row-major format is given by the following formula:

$$\text{location } (i, j) = \text{Base Address} + (i \times n + j) \times \text{size}$$

For example, if the base address is 2002, the given array has five rows, and three columns, then the address of  $(2, 1)^{\text{th}}$  element can be calculated using the following formula:

## Address in column-major

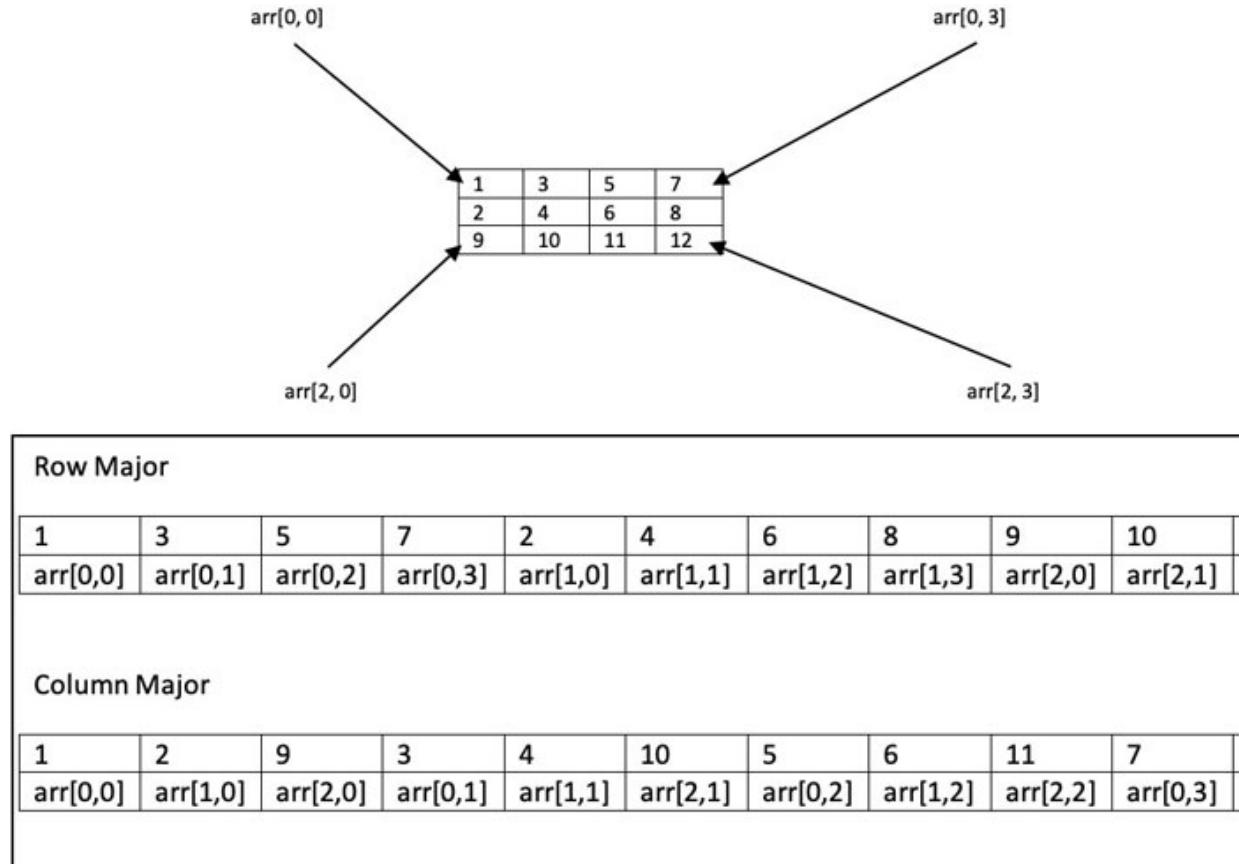
If the number of rows in a given 2-D array is  $m$ , and the number of columns is  $n$ . Then the location of the  $(i, j)^{\text{th}}$  element in the column-major format is given by the following formula:

$$\text{location } (i, j) = \text{Base Address} + (i + j \times m) \times \text{size}$$

For example, if the base address is 2002, the given array has five rows, and three columns, then the address of  $(2, 1)^{\text{th}}$  element can be calculated using the following formula:

$$\text{location}(2, 1) = 2002 + (2 + 1 \times 5) \times 2 = 2002 + 14 = 2016$$

[Figure 4.1](#) shows a two-dimensional array, and it is indexing and row-major and column-major representation:



*Figure 4.1: Indexing and memory map of a 2-D array*

We can take out required sub-arrays from a given array using slicing. The readers who are not versed with numpy are requested to go through the cheat sheet of numpy given in the Appendix of this book before proceeding any further.

## Inserting and deleting

You can add an element in a given array only if the number of elements in the array ( $n$ ) is not equal to the capacity of the array ( $max$ ). In inserting an element at the beginning, we shift all the elements to the right and then insert the item at the 0<sup>th</sup> index (refer to [figure 4.2](#)). Likewise, inserting an element after a particular value requires each element after that value to be shifted

one position to the right and then inserting item at that position, provided the  $n$  has not reached  $\max$  (refer to [figure 4.3](#)). Inserting an element at the end is relatively easy. If the value of  $n$  is not  $\max$ , then increment the value of  $n$  and insert an item at the  $n^{\text{th}}$  index (refer to [figure 4.4](#)).

An item can be deleted from a given array if it has at least one element. In deleting an element from the beginning, each element from the second position till the end is shifted one position to the left, and the value of  $n$  is decremented by 1 (refer to [figure 4.5](#)). In deleting an element from the end, the value of  $n$  is simply decremented by 1 (refer to [figure 4.6](#)). You can delete an element after a given element (**val**) only if **val** exists in the given array. Each element to the right of **val** is shifted one position to the left, after which the value of  $n$  is decremented by 1 (refer to [figure 4.7](#)). The algorithms for insertion and deletion in an array are given as follows:

**Algorithm: insert\_beg**

**Input:** Array, arr; item

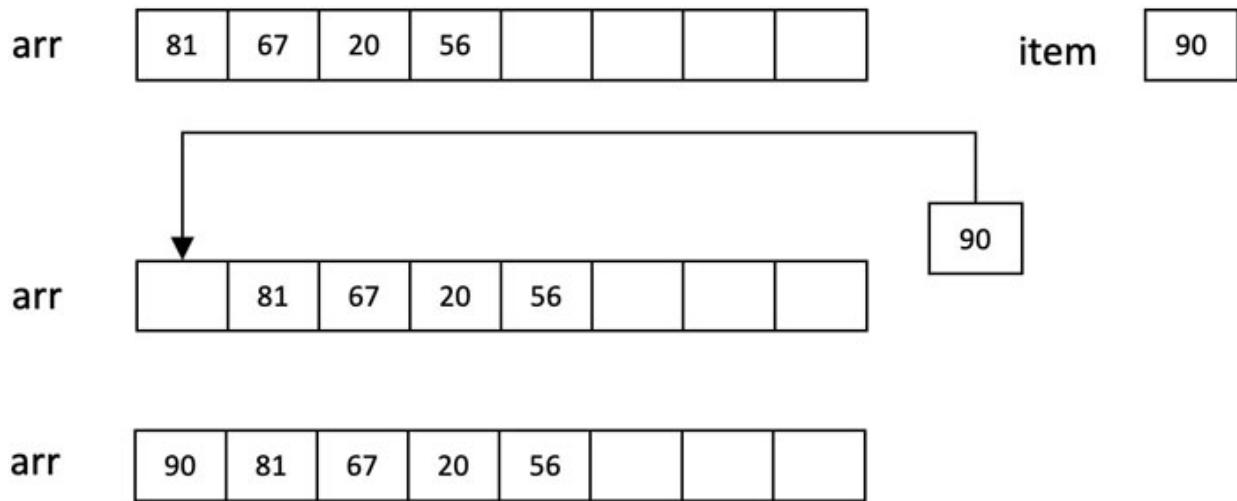
**Output:** Array with the item placed at the beginning

**Number of elements in the original array: n**

**Idea:** Shift all the elements one position to the right, and then place the item at the  $0^{\text{th}}$  index

```
def insert_beg(arr, n, item):
    if( n != max):
        i=n-1
        while(i>=0):
            arr[i+1]=arr[i]
            i-=1
        arr[i+1]=item
    n+=1
    else:
        print('Overflow')
```

**Example:**



*Figure 4.2: Insertion at the beginning*

### Algorithm: insert after a given element

**Input:** Array, **arr**; **item**

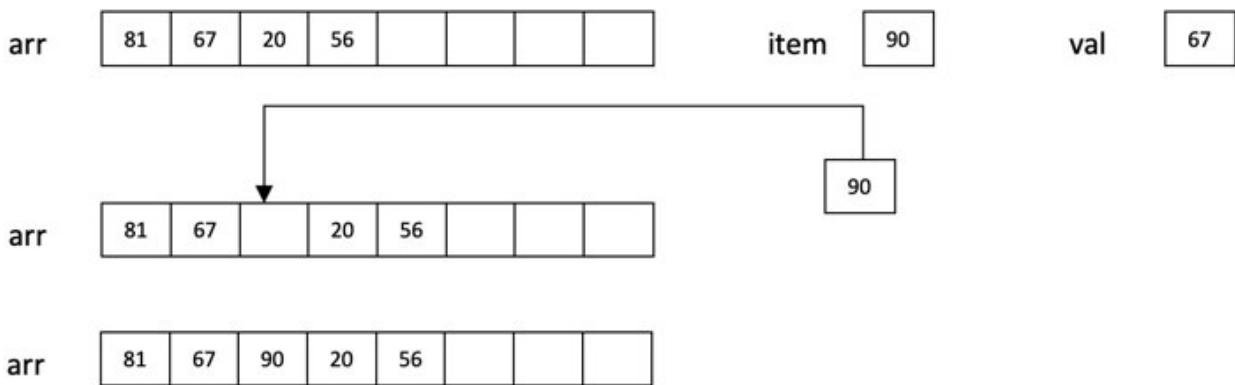
**Output:** Array with the item placed after **val**

**Number of elements in the original array:** **n**

**Idea:** Shift all the elements after **val** one position to the right, and then place the item at the position after **val**

```
def insert_after(arr, n, val, item):
    if( n != max):
        i=n-1
        while(arr[i] != val):
            arr[i+1]=arr[i]
            i-=1
        arr[i+1]=item
    n+=1
    else:
        print('Overflow')
```

**Example:**



*Figure 4.3: Insertion after a given element*

**Algorithm:** `insert_end`

**Input:** Array, **arr**; **item**

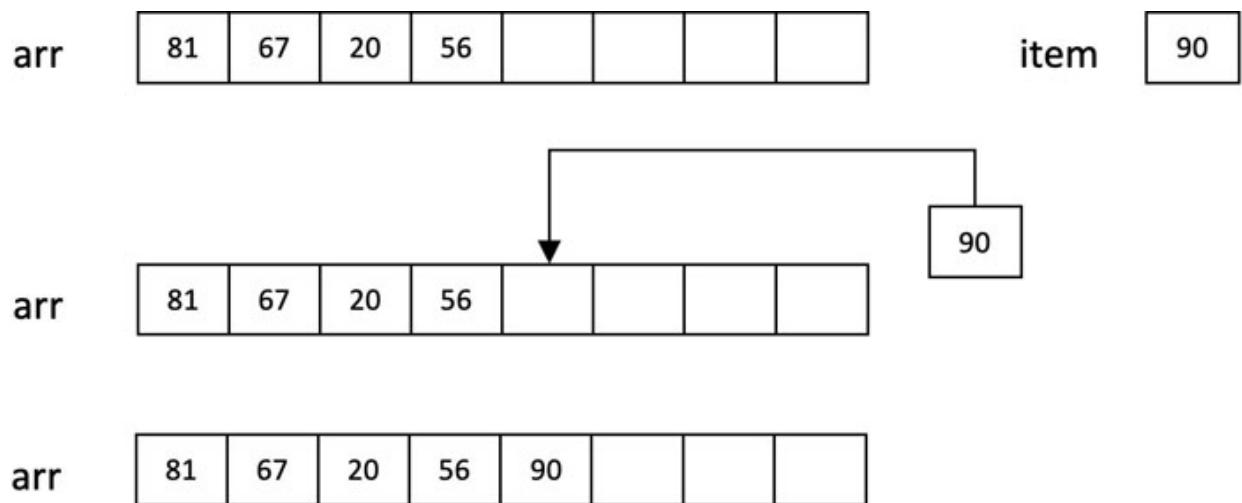
**Output:** Array with the item placed at the end

**Number of elements in the original array:** **n**

**Idea:** Place the item at the end

```
def insert_end(arr, n, item):
    if( n != max):
        arr[n]=item
    n+=1
    else:
        print('Overflow')
```

**Example:**



*Figure 4.4: Insertion at the end*

**Algorithm: del\_beg**

**Input:** Array, arr

**Output:** Array with the first item deleted

**Number of elements in the original array: n**

**Idea:** Shift all the elements one position to the left, starting from the second element

```
def del_beg(arr, n):  
    if( n != 0):  
        i=1  
        while(i<=n-1):  
            arr[i-1]=arr[i]  
            i+=1  
        n-=1  
    else:  
        print('Underflow')
```

**Example:**

arr	81	67	20	56				
-----	----	----	----	----	--	--	--	--

arr	67	20	56					
-----	----	----	----	--	--	--	--	--

*Figure 4.5: Deletion from the beginning*

**Algorithm:** Deletion from the position after the given value

**Input:** Array, arr, val

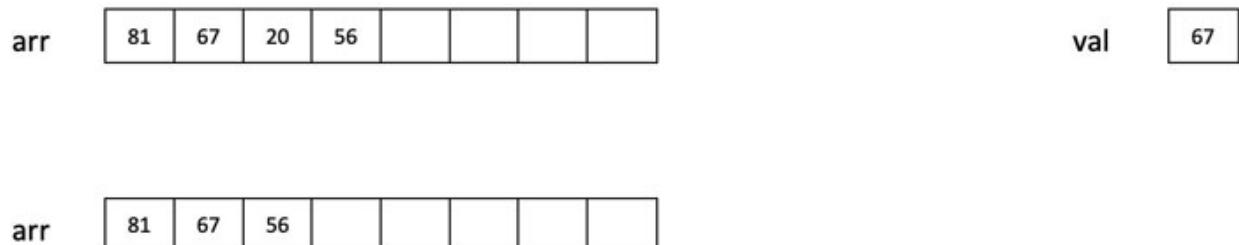
**Output:** Array with the item placed after val removed

**Number of elements in the original array: n**

**Idea:** Shift all the elements after **val** one position to the left, starting from left

```
def del_after(arr, n, val):
    if( found(val)):
        i=n-1
        while(arr[i] != val):
            i-=1
        i+=1
        while(i>n-1)
            arr[i]= arr[i+1]
            i+=1
        n-=1
    else:
        print('Not Found')
```

**Example:**



*Figure 4.6: Deletion from the position after a given value*

**Algorithm:** **del\_end**

**Input:** Array, **arr**

**Output:** Array with the last item removed

**Number of elements in the original array:** **n**

**Idea:** Remove the item placed at the end

```
def del_end(arr, n, item):
    if( n!= 0):
        n-=1
    else:
        print('Underflow')
```

**Example:**

arr	81	67	20	56				
-----	----	----	----	----	--	--	--	--

arr	81	67	20					
-----	----	----	----	--	--	--	--	--

*Figure 4.7: Deletion from the end*

The following program implements the preceding operations on an array. The code has eight functions: a constructor, three for insertions, three for deletions, and one for displaying the elements of the array.

### Code:

```

1. class Array:
2.     def __init__(self, max):
3.         self.max=max
4.         self.n=0
5.         self.arr=[0]*max
6.     def insert_beg(self, item):
7.         if(self.n != self.max):
8.             i=self.n-1
9.             while(i>=0):
10.                 self.arr[i+1]=self.arr[i]
11.                 i-=1
12.             self.arr[i+1]=item
13.             self.n+=1
14.         else:
15.             print('Overflow')

```

```
16.
17. def insert_after(self, val, item):
18.     if(self.n != self.max):
19.         i=self.n-1
20.         while(self.arr[i] != val):
21.             self.arr[i+1]=self.arr[i]
22.             i-=1
23.         self.arr[i+1]=item
24.         self.n+=1
25.     else:
26.         print('Overflow')
27.
28. def insert_end(self, item):
29.     if(self.n != self.max):
30.         self.arr[self.n]=item
31.         self.n+=1
32.     else:
33.         print('Overflow')
34. def del_beg(self):
35.     if(self.n!=0):
36.         i=1
37.         while(i<=self.n-1):
38.             self.arr[i-1]=self.arr[i]
39.             i+=1
40.         self.n-=1
41.     else:
42.         print('Underflow')
43. def del_after(self, val):
```

```

44.     if(val in self.arr):
45.         i=self.n-1
46.         while(self.arr[i] != val):
47.             i-=1
48.             i+=1
49.             while(i<self.n-1):
50.                 self.arr[i]= self.arr[i+1]
51.                 i+=1
52.                 self.n-=1
53.             else:
54.                 print('Not Found')
55.     def del_end(self):
56.         if(self.n!=0):
57.             self.n-=1
58.         else:
59.             print('Underflow')
60.     def display(self):
61.         print('Array\t:', self.arr[:self.n])

```

### **Testcase:**

1. Array1= **Array(5)**
2. Array1.insert\_beg(**5**)
3. Array1.insert\_beg(**6**)
4. Array1.insert\_beg(**10**)
5. Array1.display()

### **Output:**

**Array : [10, 6, 5]**

### **Test Case:**

1. `Array1.insert_after(10, 8)`
2. `Array1.display()`

**Output:**

`Array : [10, 8, 6, 5]`

**Testcase:**

1. `Array1.insert_end(3)`
2. `Array1.insert_end(1)`
3. `Array1.display()`

**Output:**

`Overflow`

`Array : [10, 8, 6, 5, 3]`

**Testcase:**

1. `Array1.del_beg()`
2. `Array1.display()`

**Output:**

`Array : [8, 6, 5, 3]`

**Testcase:**

1. `Array1.del_after(6)`
2. `Array1.display()`
3. `Array1.del_after(78)`

**Output:**

`Array : [8, 6, 3]`

`Not Found`

**Testcase:**

1. `Array1.del_end()`
2. `Array1.display()`

**Output:**

**Array :** [8, 6]

## Operations on arrays

The sum of the elements of a given array can be found by initializing the variable **sum** to 0 and adding each element of the array to this variable. The following code initializes the variable **sum** to 0, and the loop that follows adds each element of the array to this variable. Likewise, the product of the elements can be found by initializing the variable **prod** to 1 and multiplying each element of the array one by one by **prod**.

The maximum element of the given array can be found as follows:

- Set **max** to the first element of the array.
- Now, iterate over the remaining elements, and if the item at a particular position is greater than **max**, replace **max** with that element.

Likewise, the minimum element present in a given array can be found. The mean of the elements can be found by dividing the sum of the elements by the number of elements.

## Linear search

To implement search, we need a variable called **flag**, which should be initialized to 0. If we are able to find the item to be searched, we set the value of the **flag** to 1. At the end of the iteration, if the value of the **flag** is still 0, “Item not found” can be displayed. During the iteration, if the item is found, the position at which it is present can be printed. This will print all the positions at which the item is present, and if the item is not found, “item not found” will be printed only once.

The median of a given array can be found by sorting the elements of the array and printing the  $(n/2)^{\text{th}}$  element if  $n$  is even; else, the mean of  $(n+1)/2^{\text{nd}}$  and  $(n+3)/2^{\text{nd}}$  element is printed. Likewise, the first quartile is the  $(n+1)/4^{\text{th}}$  element; the third quartile is the  $(n+3)/4^{\text{th}}$  element in the sorted array. The difference between the third and the first quartile is the quartile deviation.

Note that [Chapter 11, Sorting](#), is dedicated to sorting. Here, we use the in-built function for sorting a given list. The following code implements the preceding functions:

## Code:

```
1. def find_max(arr, n):  
2.     max1=0  
3.     for i in range(1,n):  
4.         if(arr[i]>max1):  
5.             max1=arr[i]  
6.     return(max1)  
7. def find_min(arr, n):  
8.     min1=0  
9.     for i in range(1,n):  
10.        if(arr[i]<min1):  
11.            min1=arr[i]  
12.        return(min1)  
13. def find_sum(arr, n):  
14.     sum1=0  
15.     for i in range(n):  
16.         sum1+=arr[i]  
17.     return sum1  
18. def find_prod(arr, n):  
19.     prod1=1  
20.     for i in range(n):  
21.         prod1*=arr[i]  
22.     return prod1  
23. def linear_search(arr, n, item):  
24.     flag=0  
25.     for i in range(n):  
26.         if(arr[i]==item):  
27.             print('Found at \t:',i)
```

```

28.     flag=1
29.     if(flag==0):
30.         print('Not found')
31. def find_mean(arr, n):
32.     sum1=find_sum(arr, n)
33.     mean1=sum1/n
34.     return mean1
35. def find_median(arr, n):
36.     arr1=sorted(arr)
37.     print(arr1)
38.     if(n%2!=0):
39.         return (arr1[n//2])
40.     else:
41.         return ((arr1[(n//2)-1]+arr1[(n)//2])/2)
42. def find_quartiles(arr, n):
43.     arr1=sorted(arr)
44.     print(arr1)
45.     return (arr1[(n+1)//4], arr1[3*(n+1)//4])
46. def find_quar_devition(arr, n):
47.     q1, q3=find_quartiles(arr, n)
48.     qd=q3-q1
49.     return(qd)
50.

```

## Problems

Having seen the representation, operations, and algorithms of arrays, let us move to some problems related to arrays.

1. An array consisting of integers is given as input. Write a program to cluster all odd numbers on the left and even elements on the right side.

### **Solution:**

The simplest solution is to start traversing from the first element, and if it is an even number, we swap it with the nearest odd number; we continue to do so for the second position, third position, and so on. The following code implements this solution:

### **Code:**

```
1. for i in range(arr.shape[0]-1):
2.   if((arr[i]%2 ==0) and (i< (arr.shape[0]-1))):
3.     j=i+1
4.     while(arr[j]%2 ==0 and (j<(arr.shape[0] -1))):
5.       j= (j+1)
6.     (arr[i], arr[j])=(arr[j], arr[i])
7.   #print(arr)
```

2. In the preceding question, if the order of the integer needs to be maintained, how will you proceed?

### **Solution:**

Create a new array called **arr1**, having the same size as the given array. Now, traverse the array; if the number is odd, place it in **arr1**. Now traverse the array again, and this time put the even numbers in **arr1**. The following code implements this solution:

### **Code:**

```
1. k=0
2. arr1=np.zeros(arr.shape[0])
3. for i in range(arr.shape[0]):
4.   if(arr[i]%2 !=0):
5.     arr1[k]=arr[i]
6.     k+=1
7. for i in range(arr.shape[0]):
8.   if(arr[i]%2 ==0):
```

```
9. arr1[k]=arr[i]  
10. k+=1
```

3. If a set of numbers is given, some of which are positive and some of which are negative, how will you bring positive on the left and negative on the right side?

**Solution:**

The concept explained in the first and the second question can be used to solve this question.

4. If an array and a value are given, find all possible pairs of numbers in the array that sum to the given value.

**Solution:**

In the solution, we start with a particular element and see if the sum of that element, with any of the elements present later in the array, is equal to the given value. If this is the case, then the tuple is printed. The following code implements this solution. However, the problem with this solution is that it takes time of  $O(n^2)$ :

**Code:**

```
1. L=[]  
2. for i in range(arr.shape[0]):  
3.     for j in range(i+1, arr.shape[0]):  
4.         if(arr[i]+arr[j] ==val):  
5.             L.append((arr[i], arr[j]))  
6. L=set(L)  
7. print(L)
```

5. Suggest a way to improve the complexity of the preceding question.

**Solution:**

Sort the array in non-decreasing order. Place a pointer at the beginning and one at the end. If the sum of the numbers at the positions indicated by the pointers is the same as the required sum, print the numbers. If the sum so obtained is smaller than the required sum, move the left

pointer by one position to check the sum; else, move the right pointer by one position to the left and check the sum. Repeat the preceding process till the two pointers meet or the required sum is found. In the former case, the elements having the required sum cannot be found.

6. An array of integers is given. Find out the majority element.

**Solution:**

This is left as an exercise for the reader. The reader must try to find an  $O(n)$  solution to this problem. Kindly refer to the Appendix for the hint.

7. Find out the  $k^{\text{th}}$  maximum from a given array?

**Solution:**

Refer to the QuickSort algorithm discussed in the chapter on sorting. In the algorithm, a helper function called partition has been used. Now run a loop with different values of the index and compare the value of the key with the given value till you can get the key. The index for which the key matches the given value is the answer.

8. An array consists of only a few limited numbers that are they are repeated many times in the array. For example, zero's, one's, and two's are spread in an array having length 10. Sort this array.

**Solution:**

If you apply any sorting algorithm, you will get the answer. This solution, though correct, is inefficient. A better solution is based on the count sort discussed in the chapter on sorting.

9. An unsorted array of integers is given to you. Create a wave-like array from this array. That is, for an array, **array1** is in the required form if the first element is greater than or equal to the second, the second is less than or equal to the third, the third is greater or equal to the fourth, the fourth is less than or equal to the third, and so on.

**Solution:**

The simplest solution is to sort the array in ascending order and then swap two consecutive elements, starting from the beginning and moving with a stride of two.

**Code:**

```

1. def wave(arr, n):
2.     if(n==0 or n==1):
3.         return arr
4.     else:
5.         arr=np.sort(arr)
6.         #print(arr)
7.         if(n%2 == 0):
8.             for i in range(0, n, 2):
9.                 arr[i+1], arr[i]= arr[i], arr[i+1]
10.        else:
11.            for i in range(0, n-1, 2):
12.                arr[i+1], arr[i]= arr[i], arr[i+1]
13.    return arr

```

10. Can you improve the complexity of the preceding solution?

**Solution:**

For three elements, all possible combinations can be envisioned, and the array can be sorted in the requisite form. For a larger array, this helper function can be used to handle three elements at a time, moving with a stride of two. However, in this solution, the last element in the last call needs to be included among the three elements in the next call, as shown in the following code:

**Code:**

```

1. def wave_temp(arr):
2.     a = arr[0]
3.     b = arr[1]
4.     c = arr[2]
5.     # 1, 2, 3
6.     if(a < b and a < c and b < c):
7.         arr[0], arr[1]= arr[1], arr[0]

```

```
8.      # 1, 3, 2
9.      elif( (a < b) and (b > c) and (a < c)):
10.         arr[0], arr[1], arr[2] = c, a, b
11.         # 2, 1, 3
12.         elif((a>b) and (a<c) and (b<c)):
13.             pass
14.             # 2, 3, 1
15.             elif( (a < b) and (a > c) and (b > c)):
16.                 arr[0], arr[1], arr[2] = a, c, b
17.                 # 3, 1, 2
18.                 elif( (a > b) and (a > c) and (b < c)):
19.                     arr[0], arr[1], arr[2] = c, b, a
20.                     # 3, 2, 1
21.                     elif( (a > b) and (a > c) and (b > c)):
22.                         arr[0], arr[1], arr[2] = b, c, a
23.                         return arr
24. def wave1(arr, n):
25.     if(n==0 or n==1):
26.         return arr
27.     elif n==2:
28.         if(arr[1] > arr[0]):
29.             arr[0], arr[1] = arr[1], arr[0]
30.             return arr
31.     elif(n==3):
32.         arr=wave_temp(arr)
33.     if(n>=5):
34.         arr[:3]= wave_temp(arr[:3])
35.         i=2
```

```

36.     while(i+3<n):
37.         #print(i)
38.         d=arr[i]
39.         if( d > arr[i+1]):
40.             arr[i+1], arr[i] = arr[i], arr[i+1]
41.             arr[i:i+3]=wave_temp(arr[i: i+3])
42.             #print(arr[i:i+3])
43.             #print(arr)
44.             i+=2
45.         else:
46.             if(i+2 < n):
47.                 if(arr[i+1] > arr[i]):
48.                     arr[i], arr[i+1] = arr[i+1], arr[i]
49.                 if(n!2 !=0) and (arr[n-1]> arr[n-2]):
50.                     arr[n-1], arr[n-2] = arr[n-2], arr[n-1]
51.     return arr

```

Having seen some of the assorted problems, let us now conclude the discussion.

## Conclusion

Arrays are the building blocks of problem-solving and programming. They contain elements of the same type. These elements are stored at consecutive memory locations.

This chapter introduced arrays, discussed their memory mapping, explained insertion and deletion to/from an array, and discussed some common problems related to arrays.

The discussion continues in the following chapters, which use these arrays to implement sophisticated data structures and solve interesting problems. Furthermore, some of the problems solved in this chapter are discussed in the following chapter with new and better approaches.

The readers are expected to solve problems given in the exercises to get a good grasp of the topic. The Appendix of this book contains some more problems related to Arrays. So happy problem-solving!

Arrays are static data structures. The upcoming chapter deals with linked lists, which is a dynamic data structure. The chapter discusses the insertion, deletion, and other operations on linked lists.

## Multiple choice questions

- 1. Which of the following store's elements are at consecutive memory locations?**
  - a. List
  - b. Tuple
  - c. Array
  - d. None of the above
  
- 2. Which of the following stores only the same type of elements?**
  - a. List
  - b. Tuple
  - c. Dictionary
  - d. Arrays
  
- 3. What is the time complexity of inserting an element at the beginning, to an array?**
  - a.  $O(n^2)$
  - b.  $O(n)$
  - c.  $O(1)$
  - d. None of the above
  
- 4. What is the time complexity of inserting an element at the end, to an array?**
  - a.  $O(n^2)$
  - b.  $O(n)$

- c.  $O(1)$
- d. None of the above

**5. What is the time complexity of deleting an element from the beginning, from an array?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(1)$
- d. None of the above

**6. What is the time complexity of deleting an element from the end, from an array?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(1)$
- d. None of the above

**7. How many iterations are needed to reverse the order of elements of a given array?**

- a.  $n$
- b.  $n/2$
- c.  $n/4$
- d. None of the above

**8. What is the complexity of Linear Search?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(1)$
- d. None of the above

**9. Can you find the maximum element from a given array in  $O(\log n)$ ?**

- a. Yes

- b. No
  - c. Insufficient information
- 10. Refer to the problems section of the chapter. What is the time complexity of finding the tuple having a given sum from a given array?**
- a.  $O(n^2)$
  - b.  $O(n)$
  - c.  $O(1)$
  - d. None of the above

## **Programming**

### **Level 0**

1. From a given array, find all the elements greater than the mean of the elements.
2. An array stores the marks of 1,000 students in an exam. Find the index of the students who secured more than 99 percentile.
3. In the preceding question, find the index of the students who scored less than one percentile.

### **Level 1**

1. Refer to Question 2 of the Problems section of the chapter. Suggest a solution, which does not have quadratic time complexity, but at the same time does not use an auxiliary array.
2. Refer to Question 5 of the Problems section of the chapter. Suggest another solution to solve the problem which does not have quadratic time complexity.
3. Refer to problem 10 of the Problems section of the chapter. Can you reduce the size of the code?

### **Level 2**

1. Find if a given array has repeated elements.

2. Find the frequency of each element in a given array. The time complexity of the solution should not be quadratic.
3. An array of integers is given to you. Sort the elements in terms of their frequency.
4. An array of length  $(n-1)$  consists of consecutive numbers, with one of them missing. Find the missing number. Suggest three solutions.

## **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

## Linked List

So far, we have seen the implementation and usage of arrays. While defining an array, we allocate some memory to it, which may become problematic in case more memory is needed at the runtime or even if too little of it is used at the runtime. In addition to this, insertions and deletions in an array are computationally expensive. To handle these problems, lists or dynamic arrays may be used. This chapter focuses on the former. The reader may also note that the elements of an array are stored at consecutive memory locations, and hence, the linear relationship is manifested by the locations of the elements.

Lists are sequence objects in which insertions and deletions can be done at/from any position. This makes sense as, in most of the computing tasks, we need to store and process elements of a data structure at various positions. This chapter discusses various types of linked lists and their algorithms. The chapter assumes importance as lists are used in the implementation of Stacks, Queues, Trees, and Graphs.

### Structure

In this chapter, we will cover the following topics:

- One-way linked list
- Two-way linked list
- Circular linked list
- Stacks and Queues using linked list
- Reversing a linked list
- Assorted problems

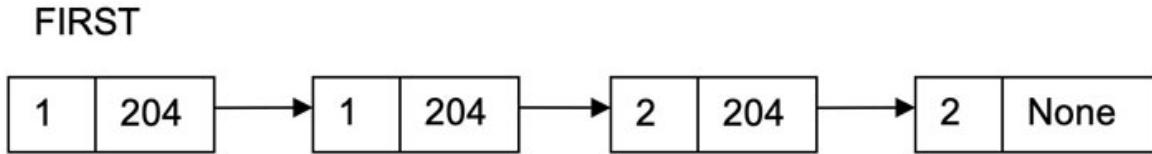
### Objectives

This chapter aims to prepare the reader to implement and use linked lists. The chapter starts with a one-way linked list and discusses insertion and deletion in it. The discussion then moves to slightly complex, two-way linked lists. This is followed by a brief discussion on circular linked lists. The implementation of

Stacks and Queues using a linked list is presented in the next section. The fourth section discusses the reversal of a list. This is followed by a brief discussion on some assorted problems related to the linked lists.

## One-way linked list

In a list, the first node is referred to as **FIRST**. The list is made up of units called nodes, which contain the data and a pointer to the next node. The last node in a list points to **NONE**. [Figure 5.1](#) shows a linked list in which the first node is referred to as **FIRST**, and the last node points to **NONE**. In the figure, the first node contains 10 as data and points to another node having the address 2048. The next node contains 15 and points to a node having address 2044. The third node contains 20 and points to 2040, and the last node contains 25 and points to **NONE**. Please refer to the following figure:



*Figure 5.1: An example of a linked list*

So, the preceding “One-way linked list” is a linear collection of nodes. Each node has two compartments: the left for the data and the right for the address of the next node. The arrow is drawn from one node to the next. Note that in some languages, **NULL** is used for the address part of the last node. Also, this text does not use the term “*pointers*” deliberately, as the implementations are done in Python.

## Traversing

Traversing a linked list makes use of the fact that the next part of the node points to the next node, and the last node points to **NONE**. In order to traverse a linked list, set a pointer **ptr** to the **FIRST** node and set **ptr** to **ptr.next** until **ptr** becomes **NONE**. In each iteration, the data of the node is processed. The algorithm for traversing a linked list is as follows:

1. Set **ptr** to **FIRST**
2. **while(ptr != NONE):**
  - a. **print(ptr.data)**
  - b. **ptr = ptr.next**

The preceding algorithm prints the data of each node in each iteration. However, in general, the processing of a node can be done as per the task at hand. In case the number of nodes in a list is to be counted, Step 2a will increment the value of a variable **count**, initialized to 0, in each iteration. The reader is requested to refer to the chapter on Arrays and write an algorithm to search a list for a given element.

## Insertion and deletion

Insertion and deletion are the most important operations in any data structure. This section explains the algorithms for insertion at the beginning, at the end, and at a given **location** and the corresponding delete operations.

In each case, we create a node (say **temp**) and put the given data in the node. Out of the three, insertion at the beginning is the easiest. In inserting a node at the beginning, we make a **temp** point to the **FIRST`node**. This is followed by renaming **temp** to **FIRST**.

In inserting a node at the end, we first set the pointer (say **ptr**) to the **FIRST** node and make it move till the last node is encountered. This is followed by setting the next pointer of **ptr** to **temp**.

In inserting an element at a given position, we set **ptr** to the **FIRST** and move it till the given data is encountered. Let the next of **ptr** be **ptr1**. We make **ptr** point to **temp** (by setting next of **ptr** to **temp**) and **temp** point to **ptr1** (by setting next of **temp** to **ptr1**). The algorithms for the preceding procedures follow:

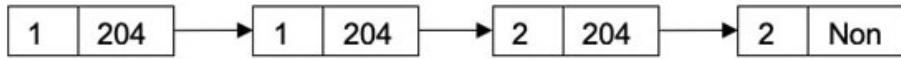
To insert a node at the beginning,

1. Create a new node called **temp** and insert the given data in **temp**.
2. Set the next pointer of **temp** to **first**.
3. Rename **temp** to **First**.

Figure 5.2 shows an example of inserting a node at the beginning of a given linked list:

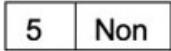
Insertion at the beginning

FIRST



Create a new node and set the data to

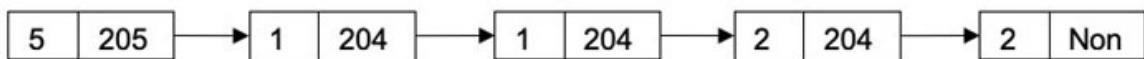
Temp



Set the next pointer of temp to FIRST

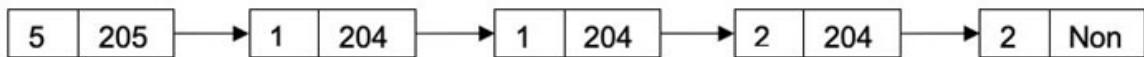
Temp

FIRST



Rename temp as FIRST

FIRST

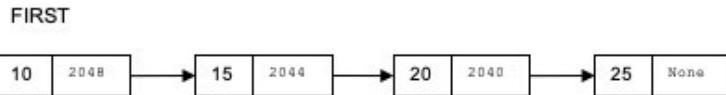


*Figure 5.2: Inserting node at the beginning of a linked list*

To insert a node at the end of a given linked list:

1. Create a new node called **temp** and insert the given data in **temp**.
2. Set a pointer **ptr** to the first node.
3. Move the pointer to the last node. This is done by setting the pointer **ptr** to its next till the next of **ptr** becomes None.
4. Now, set the next of **ptr** to **temp** and the next of **temp** to **None**.

### Insertion at the end

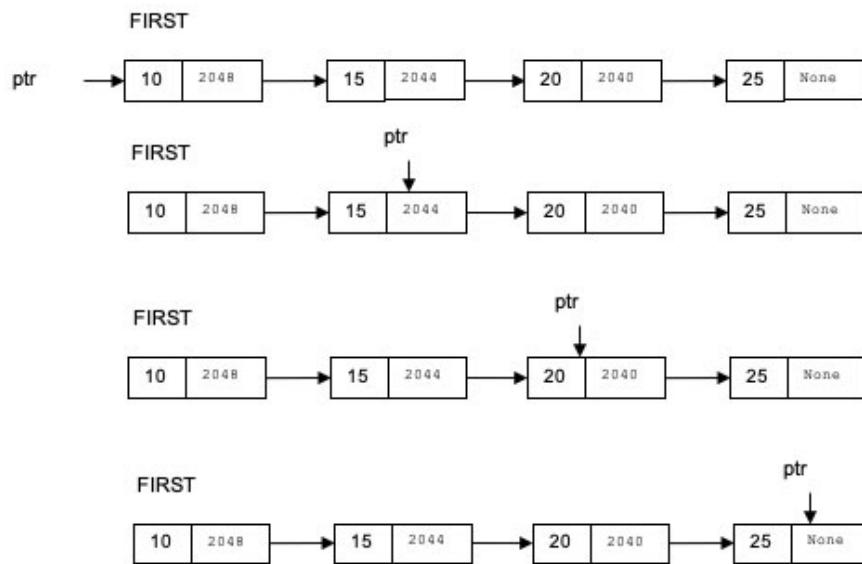


Create a new node and set the data to the given

Temp



Set the ptr pointer to FIRST



Set the next pointer of ptr to temp



*Figure 5.3: Inserting node at the end of a linked list*

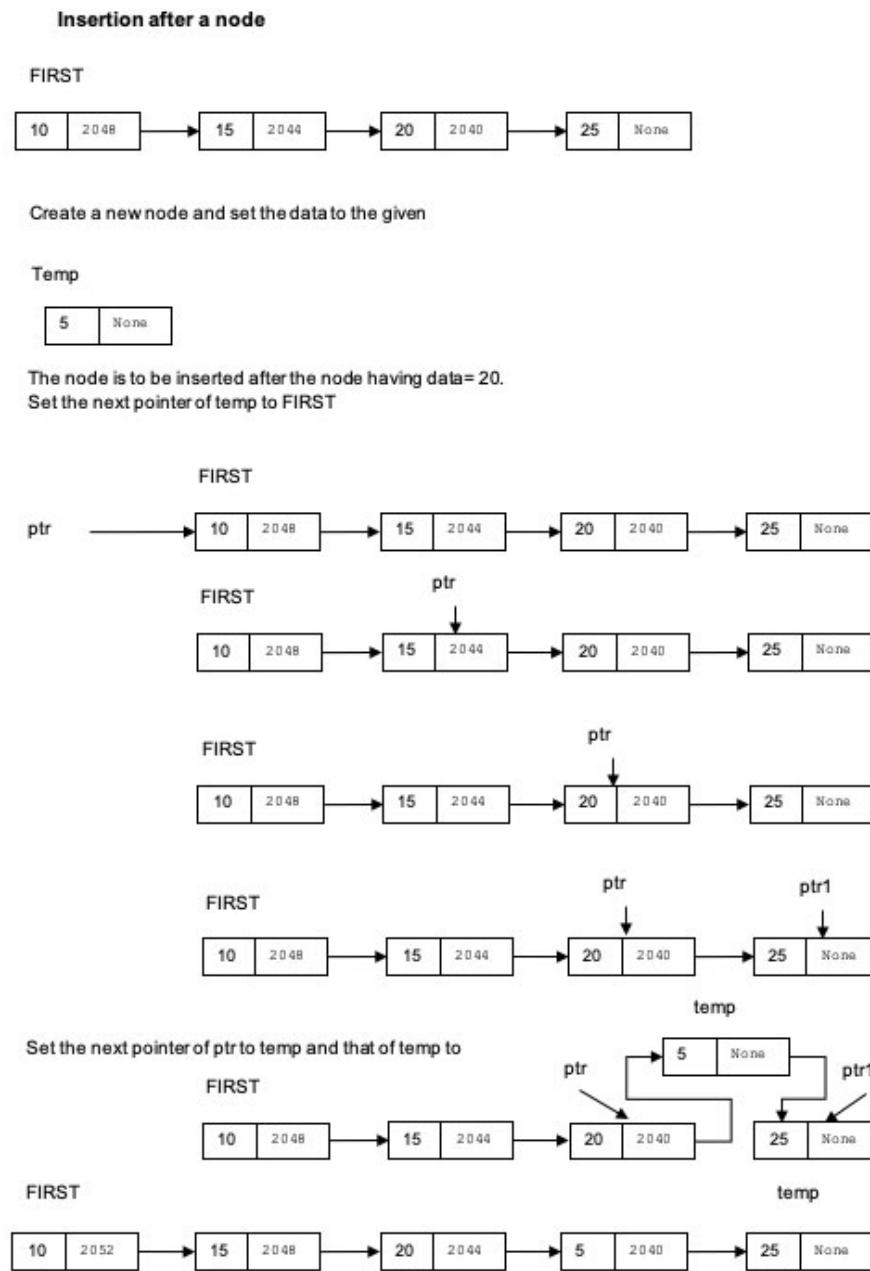
To insert a node after a given value in a linked list:

1. Create a new node called **temp** and insert the given data in **temp**.
2. Set a pointer **ptr** to the first node.
3. Move the pointer to the node having the given value. This is done by setting the pointer to its next till the data of **ptr** becomes equal to the given value.

4. Let the next of **ptr** be **ptr1**.

Now, set the next of **ptr** to **temp** and the next of **temp** to **ptr1**.

Figure 5.4 shows an example of inserting a node after the given value:



**Figure 5.4:** Inserting node after a given node in a linked list

Deleting **FIRST** is easy. We simply store the data of **FIRST** to a temporary variable (say **temp1**) and then set **FIRST** to **FIRST.next**.

Deleting the last node is not too difficult, either. Imagine you have to cut a branch of a tree. You will climb the tree and move to the branch before the branch to cut. We then cut the required branch. Likewise, to remove the last node, we move to the last but one node and set its pointer to NONE. This can be done by setting **ptr** to FIRST and moving till **ptr.next.next** becomes NONE.

If one wants to delete a node after a given value, set **ptr** to FIRST and move to the node having the required value. Let the next of **ptr** be **ptr1**, and the next of **ptr1** be **ptr2**. We set the next of **ptr** to **ptr2**. The algorithms for deletion are as follows:

Delete a node from the beginning

1. Save the data at FIRST in a temporary variable, say **temp1**.
2. Set FIRST to next of FIRST
3. Return **temp1**

Delete a node from the end

1. Set a pointer **ptr** to the first node.
2. Move the pointer to the second last node. This is done by setting the pointer to its next till the next of next of the **ptr** becomes **None**.
3. Save the data of the next of **ptr** in a temporary variable.
4. Now, set the next of **ptr** to **None**.

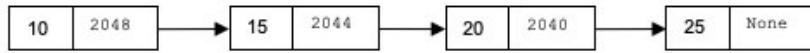
[Figure 5.5](#) shows an example of deleting a node from the end of a given linked list.

Delete node after a given value

1. Set a pointer **ptr** to the first node.
2. Move the pointer to the node having the given value. This is done by setting the pointer to its next till the value of the **ptr** becomes equal to the given data.
3. Let **ptr1** be the next of **ptr**.
4. Let **ptr2** be the next of **ptr1**.
5. Set next of **ptr** to **ptr2**.
6. Save the data of **ptr1** in a temporary variable.

### Delete from beginning

FIRST



Set FIRST to next of FIRST

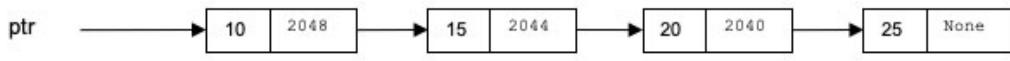
FIRST



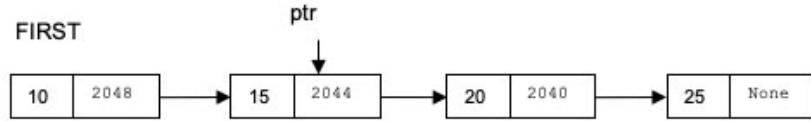
### Delete from the end

Set the next pointer ptr to FIRST

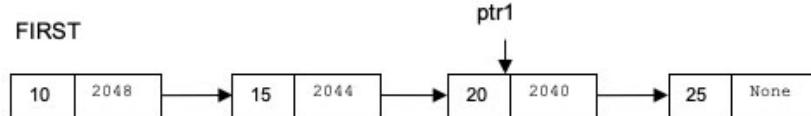
FIRST



FIRST



FIRST



FIRST

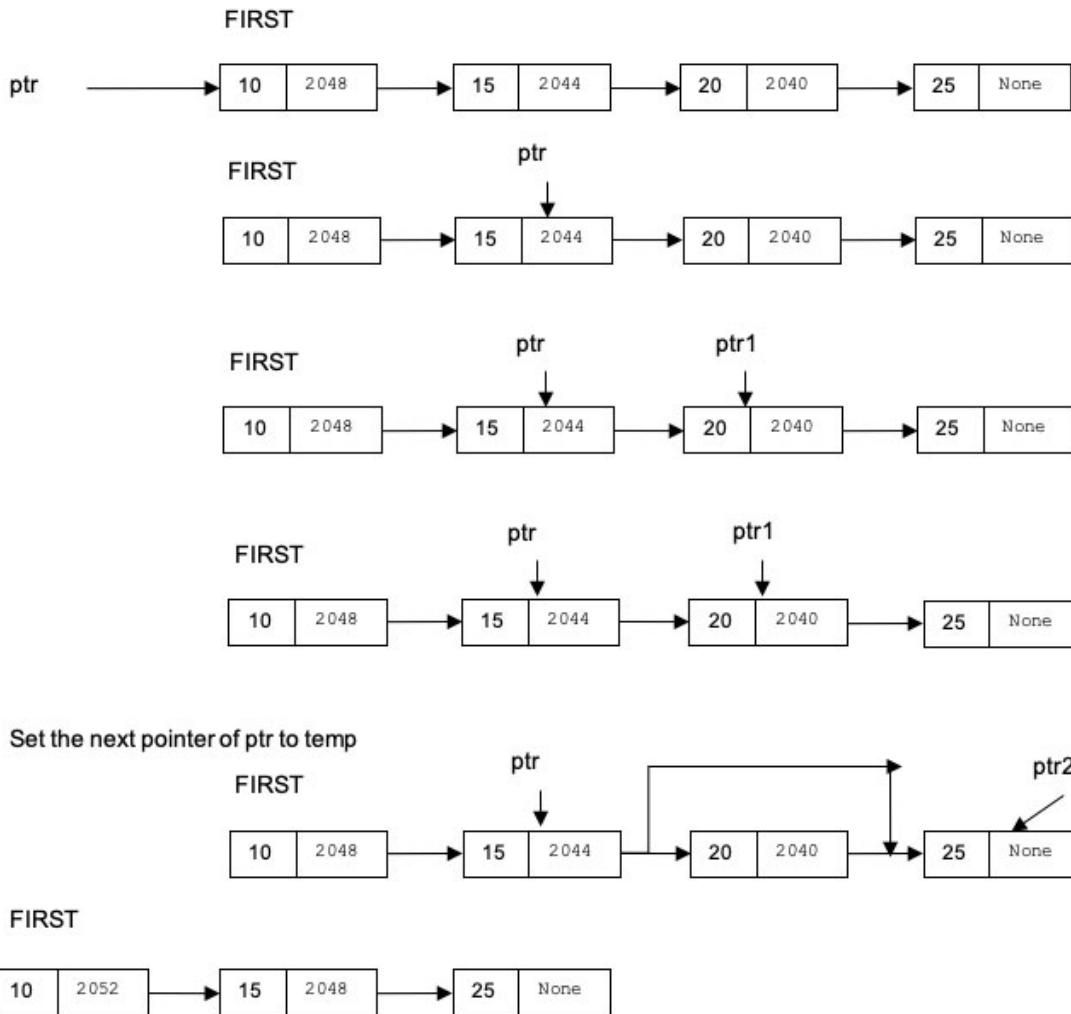


**Figure 5.5:** Deletion (a) from the beginning and (b) from the end

[Figure 5.6](#) shows an example of deleting a node after the given value from a given linked list. Please refer to the following figure:

### Delete after a given node

Set the next pointer **ptr** to **FIRST**



*Figure 5.6: Deletion after a given node*

The following program implements a “One-Way Linked List”:

```

1. class node:
2.     def __init__(self, data):
3.         self.data=data
4.         self.next=None
5.     def display_node(self):
6.         print(self.data, end=' ')
  
```

```
7. class linked_list:  
8.     def __init__(self, data1):  
9.         self.first=node(data1)  
10.    def insert_beg(self, data):  
11.        temp=node(data)  
12.        temp.next=self.first  
13.        self.first=temp  
14.    def display(self):  
15.        print('List\t:', end=' ')  
16.        ptr=self.first  
17.        while(ptr!=None):  
18.            ptr.display_node()  
19.            ptr=ptr.next  
20.        print()  
21.    def insert_end(self, data):  
22.        temp=node(data)  
23.        ptr=self.first  
24.        while(ptr.next!=None):  
25.            ptr=ptr.next  
26.        ptr.next=temp  
27.        temp.next=None  
28.    def insert_after(self, data, data1):  
29.        flag=0  
30.        temp=node(data)  
31.        ptr=self.first  
32.        while(ptr.next!=None):  
33.            if(ptr.data==data1):  
34.                flag=1  
35.                break  
36.            ptr=ptr.next
```

```
37.     if(flag==0):
38.         print('Search Failed!')
39.     else:
40.         ptr1=ptr.next
41.         ptr.next=temp
42.         temp.next=ptr1
43.     def del_first(self):
44.         if(self.first!=None):
45.             temp1=self.first.data
46.             self.first=self.first.next
47.             return temp1
48.     else:
49.         print('Underflow')
50.     def del_end(self):
51.         ptr=self.first
52.         if(ptr==None):
53.             print('Underflow')
54.         elif(ptr.next==None):
55.             temp1=ptr.data
56.             self.first=None
57.             return temp1
58.         else:
59.             while(ptr.next.next!=None):
60.                 ptr=ptr.next
61.                 temp1=ptr.next.data
62.                 ptr.next=None
63.             return temp1
64.     def del_after(self, data1):
65.         if(self.first==None):
66.             print('Underflow')
```

```

67.     else:
68.         flag=0
69.         ptr=self.first
70.         while(ptr!=None):
71.             if(ptr.data==data1):
72.                 flag=1
73.                 break
74.             else:
75.                 ptr=ptr.next
76.             if(flag==0):
77.                 print('Search Failed')
78.             else:
79.                 if(ptr.next==None):
80.                     print('Nothing after ',data1)
81.                 else:
82.                     ptr1=ptr.next
83.                     ptr2=ptr1.next
84.                     temp1=ptr1.data
85.                     ptr.next=ptr2
86.                     return temp1

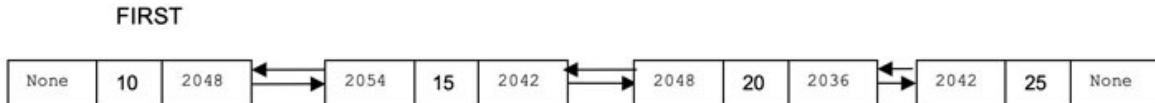
```

## Two-way linked list

The two-way linked list has three parts: the data, a pointer to the next node, and a pointer to the previous node. The first node is referred to as **FIRST**. The **prev** of **FIRST** points to **NONE**, and the **next** of the last node also points to **NONE**.

Figure 5.7 shows a linked list in which the first node is referred to as **FIRST**, and the **next** of the last nodes points to **NONE**. The **prev** of the **FIRST** node points to **NONE**. In the figure 5.7, the first node contains 10 as data and points to another node having the address 2048. The second node contains 15 and points to a node having address 2042. The third node contains 20 and points to 2036, and the last node contains 25 and points to **NONE**. The **prev** of the **FIRST** node points to **NONE**, that of the second points to 2054 (which is the address of the **FIRST**), that of the

next points to 2048, and that of the last points to 2042. Please refer to the following figure:



*Figure 5.7: An example of a two-way linked list*

So, the preceding “Two-way linked list” is a linear collection of nodes. Each node has three compartments: the left for the pointer to the previous node, the middle for the data, and the right for the address of the next node. The arrow is drawn from a node to the next node and from a node to the previous node. Note that in some languages, “NULL” is used for the address part of the last node. Also, this text does not use the term “pointers” deliberately, as the implementations are done in Python.

## Traversing

Traversing a two-way linked list is similar to that in a one-way linked list. It makes use of the fact that the next part of the node points to the next node, and the last node points to NONE. In order to traverse a linked list, set a pointer to the **FIRST** node and set **ptr** to **ptr.next** until **ptr** becomes **NONE**. In each iteration, the data of the node is processed. The algorithm for traversing a linked list is as follows:

1. Set **ptr** to **FIRST**
2. while(**ptr** != **NONE**):
  - a. **print(ptr.data)**
  - b. **ptr = ptr.next**

The preceding algorithm prints the data of each node in each iteration. However, in general, the processing of a node can be done as per the task at hand.

## Insertion and deletion

This section discusses the algorithms for insertion at the beginning, at the end, at a given location, and the corresponding delete operations for a two-way linked list. Note that the following algorithms are similar to those explained in the previous sections.

For inserting a node at the beginning, we create a node (say **temp**) and put the given data in the node. In inserting a node at the beginning, we make the **temp**

point to the **FIRST** node and the **prev** of the **FIRST** point to **temp**. This is followed by renaming **temp** to **FIRST**.

In inserting a node at the end, we first set the pointer (say **ptr**) to the **FIRST** node and make it move till the last node is encountered. This is followed by setting the next pointer of **ptr** to **temp** and the **prev** of **temp** to **ptr**.

In inserting an element at a given position, we set **ptr** to the **FIRST** and move it till the given data is encountered. Let the next of **ptr** be **ptr1**. We make **ptr** point to **temp** (by setting next of **ptr** to **temp**), **prev** of **temp** to **ptr**, **temp** point to **ptr1** (by setting **next** of **temp** to **ptr1**), and finally, **prev** of **ptr1** point to **temp**. The algorithms for the preceding procedures are as follows:

To insert a node at the beginning,

1. Create a new node called **temp** and insert the given data in **temp**.
2. Set the next pointer of **temp** to **FIRST**.
3. Set the **prev** of **FIRST** to **temp**.
4. Rename **temp** to **FIRST**.

To insert a node at the end of a two-way linked list:

1. Create a new node called **temp** and insert the given data in **temp**.
2. Set a pointer **ptr** to the first node.
3. Move the pointer to the last node. This is done by setting the pointer **ptr** to its next till the next of **ptr** becomes **None**.
4. Now, set the **next** of **ptr** to **temp**, **prev** of **temp** to **ptr**, and the **next** of **temp** to **None**.

To insert a node after a given value in a two-way linked list:

1. Create a new node called **temp** and insert the given data in **temp**.
2. Set a pointer **ptr** to the first node.
3. Move the pointer to the node having the given value. This is done by setting the pointer to its next till the data of **ptr** becomes equal to the given value.
4. Let the next of **ptr** be **ptr1**.
5. Now, set the **next** of **ptr** to **temp**, **prev** of **temp** to **ptr**, the **next** of **temp** to **ptr1**, and the **prev** of **ptr1** to **temp**.

Deleting **FIRST** is easy. We simply store the data of **FIRST** to a temporary variable (say **temp1**) and then set **FIRST** to **FIRST.next**. The new **FIRST** will have its **prev** point to **NONE**.

Deleting the last node is also not too difficult. To remove the last node, we move to the last but one node and set its pointer to **NONE**. This can be done by setting **ptr** to **FIRST** and moving till **ptr.next.next** becomes **NONE**.

If one wants to delete a node after a given value, set **ptrtoFIRST** and move to the node having the required value. Let the next of **ptr** be **ptr1**, and the next of **ptr1** be **ptr2**. We set the next of **ptr** to **ptr2**. The algorithms for deletion are as follows:

Delete a node from the beginning:

1. Save the data at **FIRST** to a temporary variable, say **temp1**.
2. Set **FIRST** to next of **FIRST**
3. Set **prev** of **FIRST** to **NONE**
4. Return **temp1**

Delete a node from the end:

1. Set a pointer **ptrto** the first node.
2. Move the pointer to the second last node. This is done by setting the pointer to its next till the next of next of the **ptr** becomes **NONE**.
3. Save the data of the next of **ptrto** a temporary variable, say **temp1**.
4. Now, set the next of **ptr** to **NONE**.

Delete node after a given value:

1. Set a pointer **ptr** to the first node.
2. Move the pointer to the node having the given value. This is done by setting the pointer to its **next** till the value of the **ptr** becomes equal to the given data.
3. Let **ptr1** be the next of **ptr**.
4. Let **ptr2** be the next of **ptr1**.
5. Set the next of **ptr** to **ptr2** and the **prev** of **ptr2** to **ptr**.
6. Save the data of **ptr1** in a temporary variable, say **temp1**.

The following program implements a doubly linked list:

```
1. class node:
2.     def __init__(self, data):
3.         self.data=data
4.         self.next=None
5.         self.prev=None
6.     def display_node(self):
7.         print(self.data, end=' ')
8. class doubly_linked_list:
9.     def __init__(self, data1):
10.        self.first=node(data1)
11.    def insert_beg(self, data):
12.        temp=node(data)
13.        temp.next=self.first
14.        self.first=temp
15.    def display(self):
16.        print('List\t:', end=' ')
17.        ptr=self.first
18.        while(ptr!=None):
19.            ptr.display_node()
20.            ptr=ptr.next
21.        print()
22.    def insert_end(self, data):
23.        temp=node(data)
24.        ptr=self.first
25.        while(ptr.next!=None):
26.            ptr=ptr.next
27.        ptr.next=temp
28.        temp.next=None
29.    def insert_after(self, data, data1):
```

```
30.     flag=0
31.     temp=node(data)
32.     ptr=self.first
33.     while(ptr.next!=None):
34.         if(ptr.data==data1):
35.             flag=1
36.             break
37.         ptr=ptr.next
38.     if(flag==0):
39.         print('Search Failed!')
40.     else:
41.         ptr\=ptr.next
42.         ptr.next=temp
43.         temp.prev=ptr
44.         temp.next=ptr1
45.         ptr\.\prev=temp
46.     def del_first(self):
47.         if(self.first!=None):
48.             temp\=self.first.data
49.             self.first=self.first.next
50.             return temp\
51.         else:
52.             print('Underflow')
53.     def del_end(self):
54.         ptr=self.first
55.         if(ptr==None):
56.             print('Underflow')
57.         elif(ptr.next==None):
58.             temp\=ptr.data
59.             self.first=None
```

```
60.         return temp1
61.     else:
62.         while(ptr.next.next!=None):
63.             ptr=ptr.next
64.             temp1=ptr.next.data
65.             ptr.next=None
66.         return temp1
67.     def del_after(self, data1):
68.         if(self.first==None):
69.             print('Underflow')
70.         else:
71.             flag=0
72.             ptr=self.first
73.             while(ptr!=None):
74.                 if(ptr.data==data1):
75.                     flag=1
76.                     break
77.                 else:
78.                     ptr=ptr.next
79.             if(flag==0):
80.                 print('Search Failed')
81.             else:
82.                 if(ptr.next==None):
83.                     print('Nothing after ',data1)
84.                 else:
85.                     ptr1=ptr.next
86.                     ptr2=ptr1.next
87.                     temp1=ptr1.data
88.                     ptr.next=ptr2
89.                     return temp1
```

If any node of a given linked list points to an earlier node, then the linked list contains a cycle.

## Cyclic list

The last node of a cyclic list points to FIRST. The reader is encouraged to write the algorithms for the insertion and deletion of elements in this list. In case of any problems, one may refer to the Appendix, which contains the program for a circular linked list.

## Stacks and Queues

Stacks are linear data structures that follow the principle of **Last In First Out (LIFO)**. The insertion and deletion in a stack are done only from the rear end. We have already seen the implementation of linked lists. A list that only supports the **insert\_end** and **del\_end** functions will act as a stack. The following program implements a stack using a linked list:

```
1. class node:  
2.     def __init__(self, data):  
3.         self.data=data  
4.         self.next=None  
5.     def display_node(self):  
6.         print(self.data, end=' ')  
7. class stack:  
8.     def __init__(self, data1):  
9.         self.first=node(data1)  
10.    def display(self):  
11.        print('List\t:', end=' ')  
12.        ptr=self.first  
13.        while(ptr!=None):  
14.            ptr.display_node()  
15.            ptr=ptr.next  
16.        print()  
17.    def insert_end(self, data):
```

```

18.     temp=node(data)
19.     ptr=self.first
20.     while(ptr.next!=None):
21.         ptr=ptr.next
22.         ptr.next=temp
23.         temp.next=None
24.     def del_end(self):
25.         ptr=self.first
26.         if(ptr==None):
27.             print('Underflow')
28.         elif(ptr.next==None):
29.             temp1=ptr.data
30.             self.first=None
31.             return temp1
32.         else:
33.             while(ptr.next.next!=None):
34.                 ptr=ptr.next
35.                 temp1=ptr.next.data
36.                 ptr.next=None
37.             return temp1

```

### Test:

```

1. s = stack(34)
2. s.insert_end(45)
3. s.insert_end(56)
4. s.insert_end(67)
5. s.insert_end(78)
6. s.display()
7. temp1=s.del_end()
8. print(temp1)

```

```
9. temp1=s.del_end()  
10. print(temp1)  
11. s.display()
```

### Output:

```
List :34 45 56 67 78
```

```
78
```

```
67
```

```
List :34 45 56
```

Queues are linear data structures that follow the principle of **First In First Out (FIFO)**. In a Queue, the insertion is done at the rear end, and deletion is done from the front end. We have already seen the implementation of a linked list. A list that only supports the `insert_end` and `del_first` functions will act as a queue. The following program implements a queue using a linked list:

```
1. class node:  
2.     def __init__(self, data):  
3.         self.data=data  
4.         self.next=None  
5.     def display_node(self):  
6.         print(self.data, end=' ')  
7. class Queue:  
8.     def __init__(self, data1):  
9.         self.first=node(data1)  
10.    def insert_end(self, data):  
11.        temp=node(data)  
12.        ptr=self.first  
13.        while(ptr.next!=None):  
14.            ptr=ptr.next  
15.        ptr.next=temp  
16.        temp.next=None  
17.    def display(self):  
18.        print('List\t:', end='')
```

```
19.     ptr=self.first
20.     while(ptr!=None):
21.         ptr.display_node()
22.         ptr=ptr.next
23.     print()
24. def del_first(self):
25.     if(self.first!=None):
26.         temp1=self.first.data
27.         self.first=self.first.next
28.         return temp1
29.     else:
30.         print('Underflow')
```

### Test:

```
1. q = Queue(34)
2. q.insert_end(45)
3. q.insert_end(56)
4. q.insert_end(67)
5. q.insert_end(78)
6. q.display()
7. temp1=q.del_first()
8. print(temp1)
9. temp1=q.del_first()
10. print(temp1)
11. q.display()
```

### Output:

List :34 45 56 67 78

34

45

List :56 67 78

Let us now have a look at some more problems related to linked lists, like reversing a linked list, concatenating a list, and checking if it contains a cycle.

## Reversing a linked list

This section discusses one of the easiest methods of reversing a linked list. The procedure uses three-pointers, initially pointing to the first three nodes. That is, **ptr** points to the first node, **ptr1** to the second, and **ptr2** to the third. The next of **ptr** is set to **None**, and the next of **ptr1** is set to **ptr**. This is followed by a loop that iterates till **ptr2** becomes **NONE**. In each iteration, the pointers shift one to the right. That is, **ptr** becomes **ptr1**, **ptr1** becomes **ptr2**, and **ptr2** becomes **ptr2** next. Also, the next of **ptr1** becomes **ptr**. The following code reverses a given linked list:

```
1. def rev_list(list1):
2.     if((list1.first==None) or (list1.first.next==None)):
3.         return list1
4.     else:
5.         ptr=list1.first
6.         ptr1=list1.first.next
7.         ptr2=ptr1.next
8.         #print(ptr.data, ptr1.data, ptr2.data)
9.         ptr1.next=ptr
10.        ptr.next=None
11.        while(ptr2!=None):
12.            ptr=ptr1
13.            ptr1=ptr2
14.            ptr2=ptr2.next
15.            ptr1.next=ptr
16.        list1.first=ptr1
17.        return list1
```

Let us now have a look at the concatenation of the two given lists.

## Concatenate lists

To concatenate two given lists, set a pointer **ptrtoFIRST** of the first list and bring it to the last node of the first list (See traversal of a list). Now, set the **next** of **ptr** to the **FIRST** of the next list. The reader is expected to create a function using the snippet given as follows to accomplish this task:

1. `ptr=list1.first`
2. `while(ptr.next!=None):`
3.   `ptr=ptr.next`
4. `ptr.next=list2.first`

Let us now see how to find a cycle in a given linked list.

## Check cycle

What if you are asked to devise a method to find out if a list contains a cycle or not? Since a linked list that does not contain a cycle, has its last node pointing to **NONE**. Can we use this to check if it contains a cycle?

NO! This is because if it contains a cycle, you will never find **NONE** (as you do not know the number of nodes in the list also). There is a trick, however. Save all the links in a set. While entering a link in the set, if you find that it already exists in the set, then the list contains a cycle. The following function checks if a given list contains a cycle:

1. `def check_cycle(list1):`
2.   `links=set()`
3.   `ptr=list1.first`
4.   `while(True):`
5.     `if ptr.next==None:`
6.       `print('No Cycle')`
7.       `break`
8.     `if ptr.next not in links:`
9.       `links.add(ptr.next)`
10.      `ptr=ptr.next`
11.     `else:`

```
12.     print("Cycle")
13.     break;
```

Having seen the intricacies of a linked list, let us now conclude this discussion.

## **Conclusion**

This chapter discussed linked lists. The importance of linked lists, insertion and deletion, and so on have been discussed in the chapter. The chapter also discusses some interesting problems related to linked lists. The lists presented in this chapter can be modified by creating an extra node called HEADER, which points to the FIRST of the lists explained in the chapter. The program for the same is included in the Web Resources of this book.

The reader will find the concepts learned in this chapter useful in the chapter that follows, such as Trees and Graphs. Also, the problem given in the Appendix extensively uses linked lists. The upcoming chapter discusses Stacks and presents an involved discussion on the need and applications of Stacks. Let us now hit the problems!

## **Multiple choice questions**

- 1. The next pointer of the last node of a One-Way Linked List points to**
  - a. None
  - b. FIRST
  - c. Both
  - d. None of the above
  
- 2. The prev pointer of the FIRST node of a Two-Way Linked list points to**
  - a. None
  - b. FIRST
  - c. Both
  - d. None of the above
  
- 3. Can you check if a linked list contains a cycle by looking for NONE in the next pointer?**
  - a. Yes

b. No

**4. To create a stack using a Linked List, which of the following methods are needed?**

- a. insert\_end
- b. insert\_beg
- c. del\_end
- d. del\_beg

**5. To create a Queue using a Linked List, which of the following methods are needed?**

- a. insert\_end
- b. insert\_beg
- c. del\_end
- d. del\_beg

**6. What is the complexity of insertion, in the beginning, in a linked List?**

- a. O(1)
- b. O(n)
- c. O(n<sup>2</sup>)
- d. None of the above

**7. What is the complexity of insertion at the end, in a linked List?**

- a. O(1)
- b. O(n)
- c. O(n<sup>2</sup>)
- d. None of the above

**8. What is the complexity of deletion from the beginning, in a linked List?**

- a. O(1)
- b. O(n)
- c. O(n<sup>2</sup>)
- d. None of the above

**9. What is the complexity of deletion from the end, in a linked List?**

- a. O(1)
- b. O(n)
- c. O( $n^2$ )
- d. None of the above

**10. Which of the following sorting algorithm is best suited for linked list?**

- a. Selection
- b. Bubble
- c. Insertion
- d. All of the above

## **Theory**

1. State some advantages and some disadvantages of an array.
2. How is a linked list better suited *vis-à-vis* an array?
3. Are there any downsides to linked lists?

**4. Write algorithms for the following for a One-Way linked list**

- a. Insertion at the beginning
- b. Insertion at end
- c. Insertion in between
- d. Deletion from the beginning
- e. Deletion from end
- f. Deletion from between

**5. Write algorithms for the following for a Two-Way linked list**

- a. Insertion at the beginning
- b. Insertion at end
- c. Insertion in between
- d. Deletion from the beginning
- e. Deletion from end
- f. Deletion from between

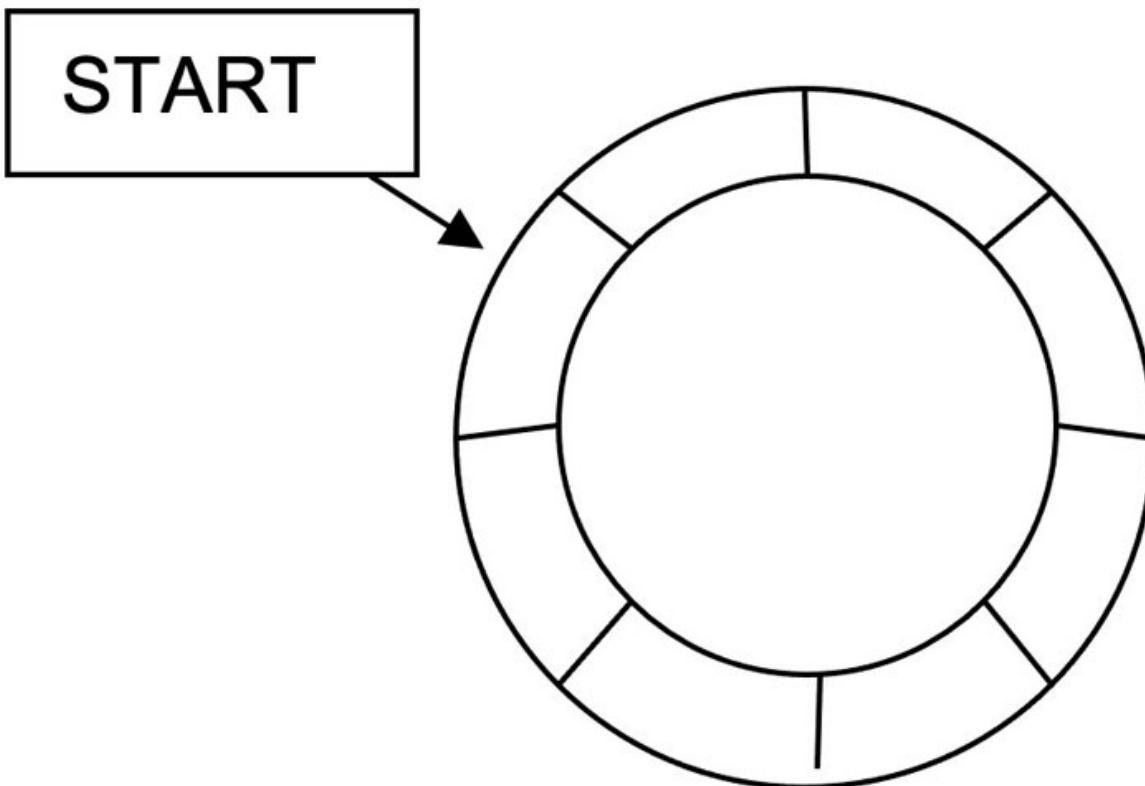
**6. Write algorithms for the following for a One-Way circularlinked list**

- a. Insertion at the beginning
- b. Insertion at end
- c. Insertion in between
- d. Deletion from the beginning
- e. Deletion from end
- f. Deletion from between

## Problems

### 1. Victor Meets Simon, Again

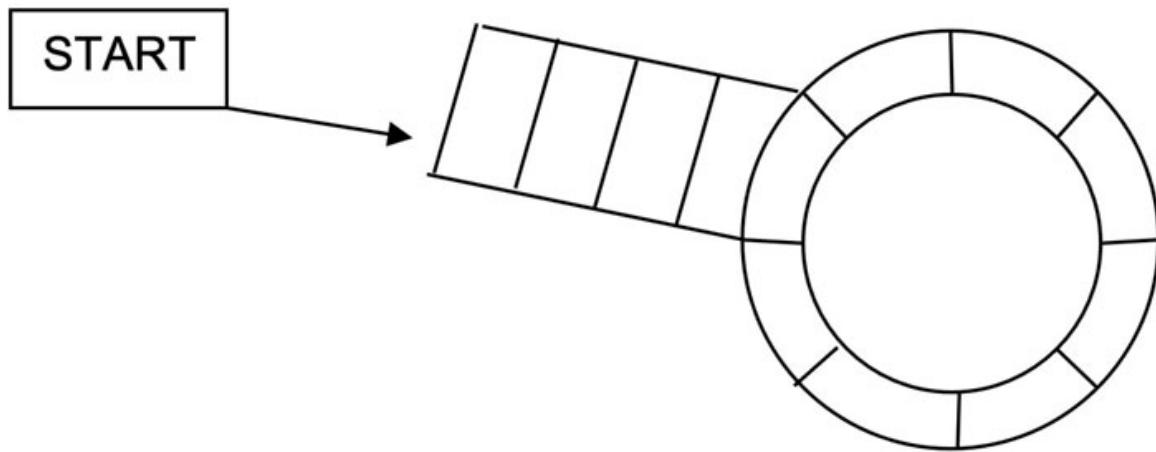
Simon and Victor were asked to begin from START ([figure 5.8](#)), cover eight sectors, come back to START, and repeat the same process until they are told to stop. Simon, being experienced in the game, covers two sectors in a unit time, whereas Victor covers one. They begin at time  $t=0$ , when will they meet again?



*Figure 5.8: Figure for Problem 1*

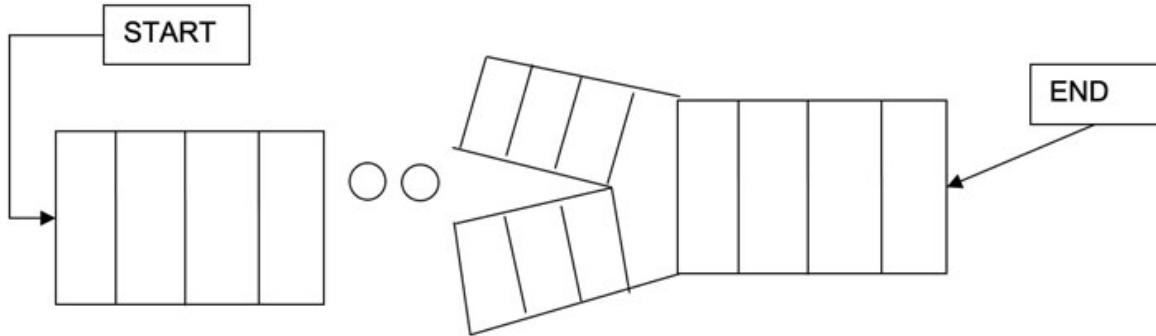
2. In the preceding question, find the sequence of times when they will meet if both start at  $t=0$ .

3. Can you use the preceding process to find a cycle in a linked list?
4. A path starts from START and leads to a loop ([figure 5.9](#)). Since the number of units in part are quite large, a person may start and continue moving in the path forever. Create a linked list of units and help him identify if there is a cycle in the path. You cannot use the method devised in the previous question.



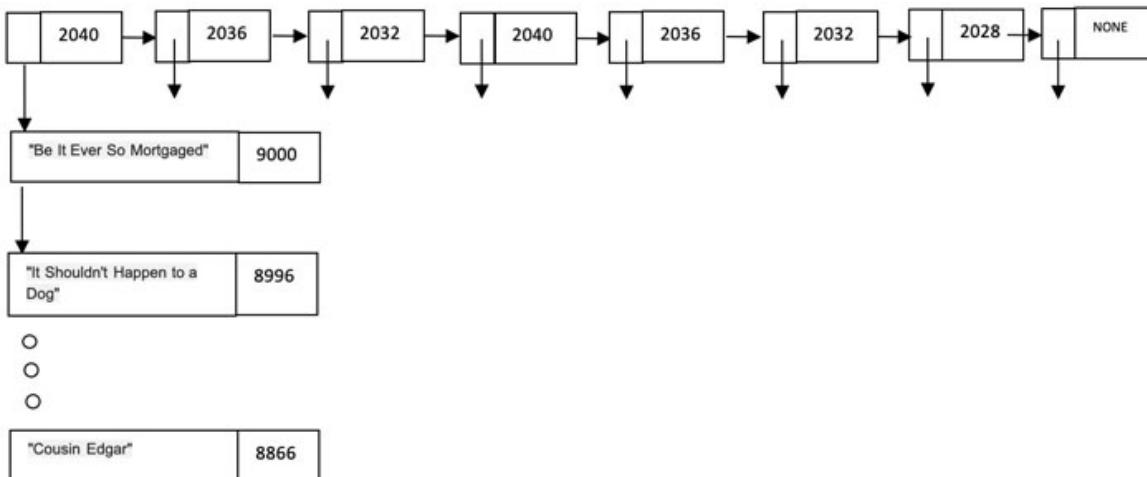
**Figure 5.9:** Figure for Problem 4

5. Consider the following linked list ([figure 5.10](#)), can you use stacks to find if there is a cycle in the path.



**Figure 5.10:** Figure for Problem 5

6. *Bewitched*, a comedy aired between 1964 and 1972 on ABC, had eight seasons. Create a linked list having a number of nodes equal to the number of seasons. Each node points to a linked list, having the names of the episodes of that season.



**Figure 5.11:** Figure for Problem 6

Now, flatten the list, creating a single list containing the names of the episodes in order of the date on which it is aired.

7. In one of the MCQs, you were asked to find the most appropriate sorting algorithm for linked lists. Justify your answer.
8. Swap the first half of the linked list with the second half.
9. Check if a given list contains duplicates. Now, write an algorithm to remove these duplicates.
10. Find the middle of a linked list without counting the number of nodes of the list.
11. Find the node of a linked list, where  $n$  is the number of nodes.
12. Perform the preceding task without counting the number of nodes.
13. If a linked list has  $n$  elements, then find the ( $i^{\text{th}}$  node) first node, which satisfies,  $i=n\%7$ .
14. In the preceding question, find the first such node from the end.
15. Given a linked list having a random number, bring the multiples of 7 at the beginning.

## 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:

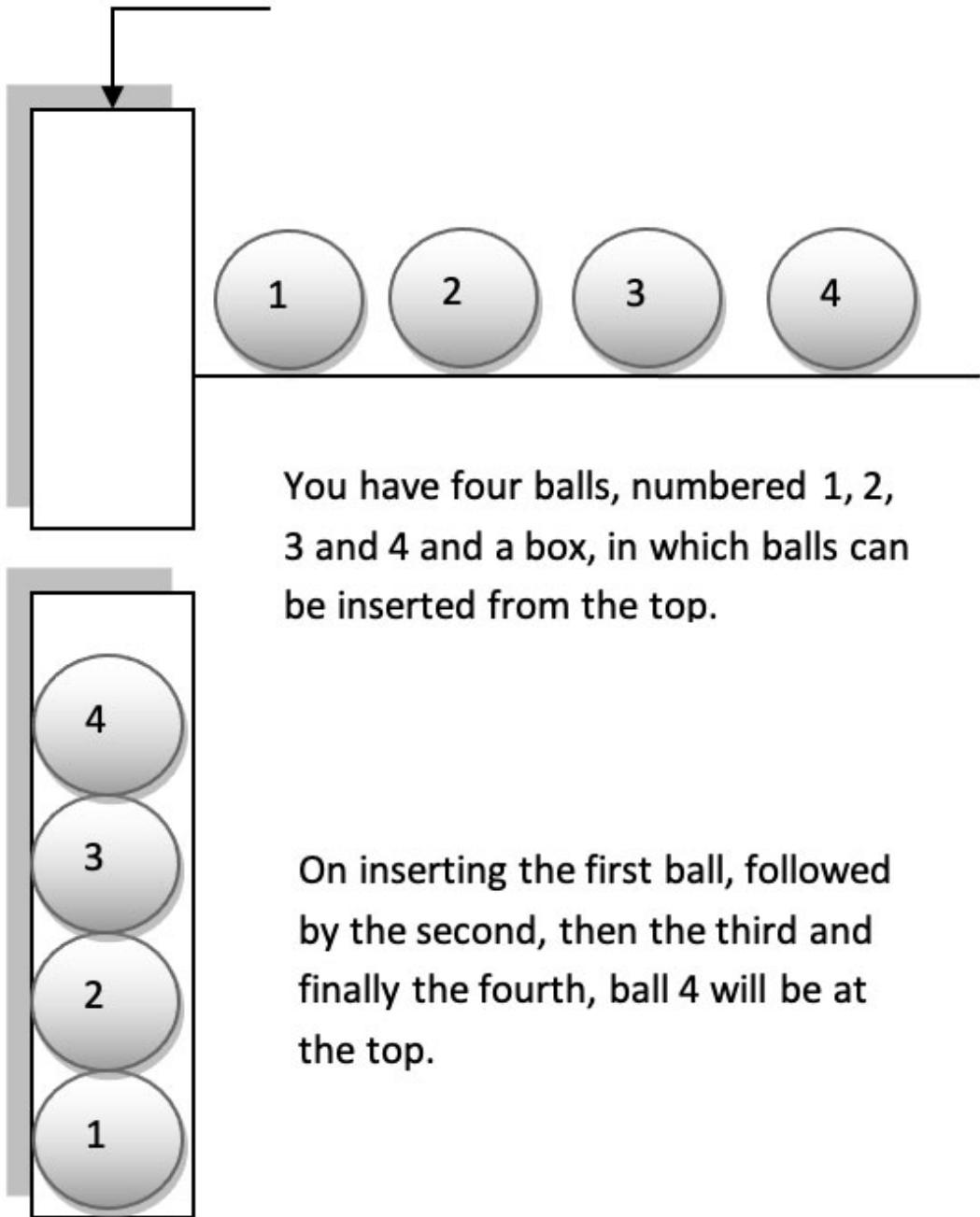
[https://discord\(bpbonline\).com](https://discord(bpbonline).com)



## CHAPTER 6

### Stacks

You have four balls numbered 1, 2, 3, and 4. A box, in which balls can be inserted or removed only from the top, is also given to you. On inserting the four balls in order, the box will have a ball numbered 4 at the top ([figure 6.1](#)). So, if you want to take out a ball, you can take out the fourth ball first. That is, the ball inserted at the end will be taken out first. This example follows the principle of **Last In First Out (LIFO)**. Please refer to the following figure:



*Figure 6.1: Putting balls in a box*

Now, revisit the chapter on recursions and have a look at the program that implements the factorial of a number. Note that, in evaluating the factorial of 5, factorial(4), followed by factorial(3), then factorial(2), and finally, factorial(1) is invoked. However, factorial(1) returns its value first, followed by others. Wow! Factorial also follows the principle of LIFO. Wait, this

principle is applicable in every program that uses recursion or sub-procedure call. This chapter introduces a linear data structure called stack that follows the principle of LIFO. Let us dive into the applications of stacks and empower ourselves with the power of this potent arm.

## Structure

This chapter covers the following topics:

- Introduction to stacks
- Reversing a string using a stack
- Implementation of two stacks using a list
- Evaluating postfix expressions
- Converting infix expression to postfix and prefix
- Implementing Queues using Stacks

## Objectives

This chapter talks about stacks, their efficient implementations, applications, evaluation of expressions, and implementation of Queues from Stacks. It forms the basis of the rest of the subject and is immensely important. It is even used in writing non-recursive procedures for recursive algorithms. The reader will learn the techniques of problem-solving using recursion and implementations.

## Introduction

A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. In a stack, items can be inserted only at the top of the stack and removed only from the top. To facilitate this, a variable called **TOP** is initialized to  $-1$  (indicating the stack to be empty). When an item is inserted in the stack, the variable **TOP** is incremented by  $1$ . When an item is removed from the stack, the value of **TOP** is decremented by  $1$ . It may be noted that you cannot remove an item from an empty stack; hence, if the value of **TOP** is  $-1$ , **UNDERFLOW** occurs. Also, note that you cannot insert an item into a filled stack (assume that it can hold **MAX** elements). That is, **OVERFLOW** occurs if

insertion is done when the value of **TOP** is **MAX-1**. The insertion in a stack is done using a function called **PUSH**, removal of an element is done using a function called **POP**.

The **IsOverflow** function returns a **TRUE** if the value of **TOP** is **MAX-1**; otherwise, it returns a **FALSE**. The **IsUnderflow** function returns a **TRUE** when the value of **TOP** is **-1**; otherwise, it returns a **FALSE**. You can use these two functions in **PUSH** and **POP** to check the conditions of underflow and overflow. The algorithms for **Push**, **Pop**, **IsOverflow**, and **IsUnderflow** are as follows:

### Algorithm: Stack using list

```
TOP = -1 and Stack = []
Push(S, TOP, MAX, item)
    if(TOP==MAX-1):
        print('Overflow')
    else:
        TOP+=1
        S[TOP] = item
Pop(S, TOP)
    if(TOP== -1):
        print('Underflow')
    else:
        temp=S[TOP]
        TOP-=1
IsOverflow(TOP)
    if(TOP==MAX-1)
        return TRUE
    else:
        return FALSE
IsUnderflow(TOP)
    if(TOP== -1)
        return TRUE
    else:
        return FALSE
```

The following code implements the stack using the Python List object:

## #Stack implementation using lists

```
1. def push(S, item, top, MAX):  
2.     if top==MAX-1:  
3.         print("Overflow")  
4.         return top  
5.     else:  
6.         top+=1  
7.         S.append(item)  
8.         return top  
9.  
10.    def pop(S, top, MAX):  
11.        if top== -1:  
12.            return (-1, -1)  
13.        else:  
14.            temp=S[top]  
15.            top-=1  
16.            del S[-1]  
17.            return (temp, top)  
18.    def disp(S, top):  
19.        print("Stack")  
20.        for i in range(top+1):  
21.            print(S[i], ' ')  
22.    def init():  
23.        S=[]  
24.        MAX=5  
25.        top=-1  
26.        return(S, MAX, top)  
27. s, MAX, top=init()
```

```
28. while(1):
29.     print('1\t:Push')
30.     print('2\t:Pop')
31.     print('3\t:Traverse')
32.     print('4\t:Exit')
33.     ch=int(input('Enter your choice\t:'))
34.     if(ch==1):
35.         item=int(input('Enter item\t:'))
36.         top=push(S, item, top, MAX)
37.     elif(ch==2):
38.         val, top=pop(S, top, MAX)
39.         if(top ==-1):
40.             print("Underflow")
41.         else:
42.             print(val)
43.     elif(ch==3):
44.         disp(S, top)
45.     elif(ch==4):
46.         print('Program ends...')
47.         break
48.     else:
49.         print('Incorrect input')
```

### Output:

```
1 :Push
2 :Pop
3 :Travrse
4 :Exit
Enter your choice :1
Enter item :23
```

```
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :1
Enter item :34
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :1
Enter item :45
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :1
Enter item :56
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :1
Enter item :78
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :1
Enter item :89
Overflow
1 :Push
2 :Pop
3 :Traverse
```

```
4 :Exit
Enter your choice :1
Enter item :90
Overflow
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :3
Stack
23
34
45
56
78
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :2
78
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :2
56
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :2
45
1 :Push
```

```

2 :Pop
3 :Traverse
4 :Exit
Enter your choice :2
34
1 :Push
2 :Pop
3 :Traverse
4 :Exit
Enter your choice :4
Program ends...

```

Having seen the implementation of the stack, let us now move to our next problem and explore if we can use a single list more efficiently.

## Implementing two stacks using a single list

You can implement two stacks using a single list. To do so, you need two TOPs: TOP1 and TOP2. TOP1 is initialized to -1, and TOP2 is initialized to MAX. If an element is inserted in the first stack, the value of TOP1 is incremented. On inserting an element in the second stack, the value of TOP2 is decremented by 1. An overflow occurs when  $\text{TOP1} + 1 = \text{TOP2}$ . The following program implements two stacks using a single list. Note that the program initializes the list to 0's and, on invoking pop sets the element at that position to 0.

### #Two stacks using a list

```

1. def push1(S, item, top1, top2, MAX):
2.     if top1+1==top2:
3.         print("Overflow")
4.         return top1
5.     else:
6.         top1+=1
7.         S[top1]= item
8.     return top1

```

```
9. def push2(S, item, top1, top2, MAX):
10.    if top1+1==top2:
11.        print("Overflow")
12.        return top2
13.    else:
14.        top2+=-1
15.        S[top2]=item
16.        return top2
17.
18. def pop1(S, top1, top2, MAX):
19.    if top1== -1:
20.        return (-1, -1)
21.    else:
22.        temp=S[top1]
23.        S[top1]=0
24.        top1-=1
25.        return (temp, top1)
26.
27. def pop2(S, top1, top2, MAX):
28.    if top2==MAX:
29.        return (-1, -1)
30.    else:
31.        temp=S[top2]
32.        S[top2]=0
33.        top2+=1
34.        return (temp, top2)
35.
36. def disp(S):
```

```
37. print("Stack")
38. for i in S:
39.     print(i, ' ')
40. def init():
41.     MAX=5
42.     S=MAX*[0]
43.     top1=-1
44.     top2=MAX
45.     return(S, MAX, top1, top2)
46. s, MAX, top1, top2=init()
47. while(1):
48.     print('1\t:Push in Stack')
49.     print('2\t:Pop from Stack')
50.     print('3\t:Push in Stack')
51.     print('4\t:Pop from Stack')
52.     print('5\t:Traverse')
53.     print('6\t:Exit')
54.     ch=int(input('Enter your choice\t'))
55.     if(ch==1):
56.         item=int(input('Enter item\t'))
57.         top1=push(s,item, top1, top2, MAX)
58.     elif(ch==2):
59.         val, top1=pop(s, top1, top2, MAX)
60.         if(top1 == -1):
61.             print("Underflow")
62.         else:
63.             print(val)
64.     elif(ch==3):
```

```

65.     item=int(input('Enter item\t:'))
66.     top2=top1
67.     elif(ch==4):
68.         val, top2=pop2(S, item, top1, top2, MAX)
69.         if(top2 ==MAX):
70.             print("Underflow")
71.         else:
72.             print(val)
73.     elif(ch==5):
74.         disp(S)
75.     elif(ch==6):
76.         print('Program ends...')
77.         break
78.     else:
79.         print('Incorrect input')
80.

```

The reader is expected to test the preceding code for all possible conditions. Let us now move to the uses of stacks.

## Types and uses

In languages like C, if stacks are implemented using arrays, then the memory is allocated at the compile time. Here, if one tries to push more elements than MAX (the maximum number of elements the given stack can store), OVERFLOW occurs.

If, on the other hand, if a stack is implemented using Linked Lists, then the memory is allocated at the run time. This leads to better memory management. This chapter discusses the implementation of stacks using lists. However, the variable MAX ensures that the stack cannot house more than the given number of elements. The chapter on Linked Lists revisits stacks and presents an alternate implementation of Stacks.

Stacks have numerous applications. Some of the most important ones are as follows:

- Reverse a string
- Convert an infix expression into a postfix and prefix
- Evaluate a postfix expression
- To convert a recursive procedure into a non-recursive one

The following sections discuss the preceding applications and lay the foundation for your becoming a programmer. Let us start!

## **Reversing a string**

The reversal of a string using a stack is quite easy. To do so, an empty stack is taken, and each character of a given string is pushed into the stack. The answer is stored in a string called R, which is initially empty. This is followed by taking out each character from the stack, one at a time, and appending it to R. The following program reverses a string using stack.

### **#Reverse a string using a stack**

```
1. string1=input('Enter string ')
2. s2=[]
3. top=-1
4. MAX=10
5. for i in string1:
6.     top=push(SY, i, top, MAX)
```

Now, let us move to a slightly involved application and explore expressions.

## **Expressions**

A binary operator requires two operands. If the binary operator is written between the two operands, the expression is referred to as an infix expression. If the operator precedes the two operands, the expression is called a prefix expression, and if the operator follows the two operands, the expression is called a postfix expression. That is, if  $a$  and  $b$  are the operands

and + is the operator, then the infix, prefix, and postfix expressions are as follows:

*Infix:  $a + b$ ,*

*Prefix:  $+ ab$ ,*

*Postfix:  $ab +$*

The concept can be applied to complex expressions also. For example, the expression,

$$(a+b)*(c/d)$$

can be converted into a postfix expression by converting  $(a+b)$  and  $(c/d)$  into postfix and treating the expressions so obtained as operands with \* as the operator. That is,

$$\begin{aligned} &= (ab+) * (cd/) \\ &= (ab+) (cd/)* \\ &= ab + cd/ * \end{aligned}$$

Likewise, the same expression can be converted into a prefix expression by converting  $(a+b)$  and  $(c/d)$  into a prefix and treating the expressions so obtained as operands with \* as the operator. That is,

$$\begin{aligned} &= (+ab) * (/cd) \\ &= * + ab / cd \end{aligned}$$

## Evaluation of postfix

Postfix expressions can be evaluated using a stack. This section discusses the conversion of a postfix expression into an infix expression. The algorithm for the same is as follows.

### **Algorithm: Evaluation of postfix**

1. Set the resultant expression, P to NULL. The following algorithm requires a stack, S, which initially is empty.
2. Scan the expression from left to right for each symbol x
3. If x is an operand, push it to the stack

4. If  $x$  is an operator, pop two symbols from the stack, say  $y$  followed by  $z$
5. Evaluate and push it to the stack.

For example, if  $x$  is  $+$  and the two topmost symbols in the stack are  $6(y)$  and  $5(z)$ , then push  $z + y = 5 + 6 = 11$ , to the stack. When  $x$  is null, then the last symbol left in the stack is the answer.

The following program implements the postfix to infix conversion. The test case uses the expression  $54 * 3 -$ .

- 5 and 4 are first pushed into the stack.
- On encountering  $*$ , 4, and 5 and popped from the stack, and  $(4*5)$ , that is 20 is inserted in the stack.
- Then 3 is pushed in the stack.
- On encountering  $-$ , 3 and 20 are popped, and  $(20 - 3 = 17)$  is pushed in the stack.

The following program implements the preceding algorithm.

## #Postfix to Infix

```
1. operator=['+', '-']
2. exp1=input('Enter postfix expression\n\t:')
3. s3=[]
4. top=-1
5. MAX=100
6. for i in exp1:
7.     if i in operator:
8.         b,top=pop(S3, top, MAX)
9.         a,top=pop(S3, top, MAX)
10.        if(i=='+'):
11.            temp=a+b
12.            #print(temp, top)
13.            top=push(S3, temp, top, MAX)
14.            #disp(S3, top)
```

```

15. elif(i == '-'):
16.     temp=a-b
17.     top=push(S¶, temp, top, MAX)
18.     #disp(S3, top)
19. else:
20.     top=push(S¶, int(i), top, MAX)
21. disp(S3, top)

```

### **Output:**

**Enter postfix expression :54\*3-**

**Stack**

**17**

The preceding code contains some comments. The reader can un-comment the statements to get hold of the working of the algorithm. Let us now move to a slightly complex conversion.

## **Infix to postfix**

To convert a given infix **expression** (E) to **postfix** (P), perform the following steps using a stack (initially empty):

### **Algorithm: Infix to Postfix**

1. Push an opening parenthesis into the stack and append a closing parenthesis at the end of the given expression.
2. Scan the given expression from left to right
  - a. If the scanned symbol is an opening parenthesis, push it into the stack.
  - b. If the scanned symbol is an operand, append it at the end of P.
  - c. If it is an operator, check the symbol at the top of the stack (say y).
    - i. If y is not an operator, push the incoming symbol in the stack.
    - ii. If y is an operator and its precedence is less than the incoming operator, push the incoming operator in the stack, else pop y till and append it to the end of P (till the top of the stack is an opening parenthesis).

stack contains an operator having precedence less than the incoming operator, or the top of the stack contains a symbol other than an operator).

- d. If the incoming symbol is a closing parenthesis, pop symbols from the stack one by one and appends them to the end of P.

The program for the conversion of an infix expression to the postfix is as follows:

### #Infix to postfix

```
1. def infix_to_postfix(E, prec):  
2.     # initialize P to a null string  
3.     P = ''  
4.     # extract operators from the prec dictionary  
5.     operator=prec.keys()  
6.     #print(operator)  
7.     # Initialize stack to [] and top to -1  
8.     stack\= []  
9.     top = -1  
10.  
11.    #Insert an '(' at the top of the stack and ')' at the end  
      # of the input string  
12.    stack\append('(')  
13.    top +=1  
14.    E+=''  
15.    step=1  
16.    # Scan the input string  
17.    for x in E:  
18.  
19.        # If x in '(' push it in the stack  
20.        if ( x =='(' ):
```

```

21.     stack\append(x)
22.     elif ( x in operator):
23.         # If x is an operator and the the symbol at the top
24.         # of the stack is an operator having precedence greater than
25.         # x, pop is and append it to P. Also put x in the stack.
26.         # Otherwise, push x in the stack
27.
28.     y=stack\[top]
29.     if y in operator:
30.         while ( prec[y] >= prec[x] ):
31.             y=stack\pop()
32.             top -= 1
33.             P += y
34.             y=stack\[top]
35.             if y not in operator:
36.                 break
37.             stack\append(x)
38.             top += 1
39.
40.     elif ( x == ')' ):
41.         # If x is ')' then pop everything from the stack till
42.         # you encounter '(' and append the symbols to P.
43.         while ( stack\[top] != '(' ):
44.             y = stack\pop()
45.             top -= 1
46.             P+=y

```

```

46.         y = stack1.pop()
47.         top -=1
48.         #P+=y
49.         #print(stack1, top, P)
50.
51.     else:
52.         # If x is an operand push it in stack
53.         P+=x
54.         print(stack1, top, P)
55.
56.         print('\nStep\t:',step, '\nSymbol\t:',x, '\nStack',
57.               stack1, '\nExpression \t:',P)
58.         step+=1
59.     return P
60. prec={ '+':1, '-':1, '*':2, '/':2}
61. E='a+b'
62. P=infix_to_postfix(E, prec)
63. print(P)

```

### **Output:**

```

['('] 0 a
Step : 1
Symbol : a
Stack [ '(' ]
Expression : a
Step : 2
Symbol : +
Stack [ '(', '+' ]
Expression : a
['(', '+'] 1 ab

```

```
Step : 3
Symbol : b
Stack ['(', '+']
Expression : ab
Step : 4
Symbol : )
Stack []
Expression : ab+
ab+
```

The reader may change the value of E to get the corresponding postfix expression. To understand the preceding algorithm, consider the following examples:

**Example 1:** Convert  $(a + b)$  to postfix

**Step 0:** Push ‘(‘ in the stack and ‘)’ at the end of the given expression, E.

**Step 1:** The stack has an opening parenthesis. The first symbol of the given expression is ‘a’, which is an operand and hence will be appended to P.

Symbol: a

Stack ['(']

Expression : a

**Step 2:** The next symbol is ‘+’, and the symbol at the top of the stack is not an operator, so ‘+’ is pushed into the stack.

Symbol: +

Stack ['(', '+']

Expression : a

**Step 3:** The next symbol is ‘b’. Since it is an operand, it is appended to P.

Symbol: b

Stack ['(', '+']

Expression : ab

**Step 4:** The last symbol is ‘)’, which pops everything from the stack till an ‘(‘ s encountered.

Symbol: )

Stack []

Expression : ab+

The resulting postfix expression is therefore ab+

**Example 2:** Convert to  $(a+b*c)$  postfix.

**Step 0:** Push '(' in the stack and ')' at the end of the given expression, E.

**Step 1:** The stack has an opening parenthesis. The first symbol of the given expression is 'a', which is an operand and hence will be appended to P.

Symbol: a

Stack ['(']

Expression : a

**Step 2:** The next symbol is '+', and the symbol at the top of the stack is not an operator, so '+' is pushed into the stack.

Symbol: +

Stack ['(', '+']

Expression : a

**Step 3:** The next symbol is 'b'. Since it is an operand, it is appended to P.

Symbol: b

Stack ['(', '+']

Expression : ab

**Step 4:** The next symbol is '\*', and the symbol at the top of the stack is an operator having precedence less than the incoming operator, so '\*' is pushed into the stack.

Symbol: \*

Stack ['(', '+', '\*']

Expression : ab

**Step 5:** The next symbol is 'c'. Since it is an operand, it is appended to the expression.

Symbol: c

Stack ['(', '+', '\*']

Expression : abc

**Step 6:** The last symbol is ‘)’, which pops everything from the stack till an ‘(‘ is encountered.

Symbol: )

Stack []

Expression : abc\*+

The resulting postfix expression is ‘abc\*+’

**Example 3:** Convert  $(a^*b+c)$  to postfix.

**Step 0:** Push ‘(‘ in the stack and ‘)’ at the end of the given expression, E.

**Step 1:** The stack has an opening parenthesis. The first symbol of the given expression is ‘a’, which is an operand and hence will be appended to P.

Symbol: a

Stack ['(']

Expression : a

**Step 2:** The next symbol is ‘\*’, and the symbol at the top of the stack is not an operator, so ‘\*’ is pushed into the stack.

Symbol: \*

Stack ['(', '\*']

Expression : a

**Step 3:** The next symbol is ‘b’. Since it is an operand, it is appended to P.

Symbol : b

Stack ['(', '\*']

Expression : ab

**Step 4:** The next symbol is ‘+’, and the symbol at the top of the stack is an operator having precedence greater than the incoming operator, so ‘\*’ is popped, appended to P and ‘+’ is pushed in the stack.

Symbol : +

Stack ['(', '+']

Expression : ab\*

**Step 5:** The next symbol is ‘c’. Since it is an operand, it is appended to the expression.

Symbol : c

Stack ['(', '+']

Expression : ab\*c

**Step 6:** The last symbol is ')', which pops everything from the stack till an '(' s encountered.

Symbol : )

Stack []

Expression : ab\*c+

The resulting postfix expression is 'ab\*c+'

Let us now move to the conversion of an infix expression to the corresponding prefix expression.

## Infix to prefix

To convert an infix expression to a prefix expression, reverse the input string, and follow the steps in “infix to prefix conversion”. Finally, reverse the expression so obtained.

To convert a given **infix expression (E)** to the **corresponding prefix (P)**, perform the following steps using a stack (initially empty):

### **Algorithm: Infix to prefix**

1. Push an opening parenthesis into the stack and append a closing parenthesis at the end of the given expression.
2. Reverse the input expression
3. Scan the given expression from left to right
  - a. If the scanned symbol is an opening parenthesis, push it into the stack.
  - b. If the scanned symbol is an operand, append it at the end of P.
  - c. If it is an operator, check the symbol at the top of the stack (say y).
    - i. If y is not an operator, push the incoming symbol in the stack.
    - ii. If y is an operator and its precedence is less than the incoming operator, push the incoming operator in the stack,

else pop y till and append it to the end of P (till the top of the stack contains an operator having precedence less than the incoming operator, or the top of the stack contains a symbol other than an operator).

- d. If the incoming symbol is a closing parenthesis, pop symbols from the stack one by one and appends them to the end of P.

4. Reverse P, so obtained.

The program for the conversion of an infix expression to the prefix is as follows:

### #Infix to Prefix

```
1. def infix_to_prefix(E, prec):  
2.     # initialize P to a null string  
3.     P = ''  
4.  
5.     #Reverse E  
6.     E=E[::-1]  
7.  
8.     # extract operators from the prec dictionary  
9.     operator=prec.keys()  
10.    #print(operator)  
11.    # Initialize stack to [] and top to -1  
12.    stack\ = []  
13.    top = -1  
14.  
15.    #Insert an '(' at the top of the stack and ')' at the end  
         of the input string  
16.    stack\ .append('(')  
17.    top +=1  
18.    E+=')'
```

```
19. step=1
20. # Scan the input string
21. for x in E:
22.
23.     # If x is '(' push it in the stack
24.     if ( x == '(' ):
25.         stack\append(x)
26.     elif ( x in operator):
27.         # If x is an operator and the symbol at the top
28.         # of the stack is an operator having precedence greater than
29.         # x, pop it and append it to P. Also put x in the stack.
30.         # Otherwise, push x in the stack
31.         y=stack\[top]
32.         if y in operator:
33.             while ( prec[y] >= prec[x] ):
34.                 y=stack\pop()
35.                 top -= 1
36.                 P += y
37.             y=stack\[top]
38.             if y not in operator:
39.                 break
40.             stack\append(x)
41.             top += 1
42.
43.
44.     elif ( x == ')' ):
```

```

45.          # If x is ')' then pop everything from the stack
        till you encounter '(' and append the symbols to P.
46.      while (stack1[top] != '('):
47.          y = stack1.pop()
48.          top -=1
49.          P+=y
50.          y = stack1.pop()
51.          top -=1
52.          #P+=y
53.          #print(stack1, top, P)
54.
55.      else:
56.          # If x is an operand push it in stack
57.          P+=x
58.          print(stack1, top, P)
59.
60.          print('\nStep\t:',step, '\nSymbol\t:',x, '\nStack',
        stack1, '\nExpression \t:',P)
61.      step+=1
62.
63.      #Reverse P
64.      P=P[::-1]
65.      return P

```

The reader should test the preceding code for various inputs, particularly for examples 1, 2, and 3 and above. He/she should also find the corresponding Prefix expressions manually. Let us now explore some standard problems and solve them using stacks.

## Problems

**Problem 1:** Given a list of elements, create a min stack that pops the minimum element in O(1) time.

**Solution:** To find the minimum element in O(1) time, we need two stacks. The first element is pushed to both stacks. After that, an incoming element is pushed into the first stack, and if it is smaller than that at the top of the second stack, it is pushed into the second stack as well. This way, we will have the smallest element at the top of the second stack. The `pop()` operation (second stack) will always pop the smallest element in O(1) time. The following program implements `pop()` in O(1) time. The reader is expected to implement the algorithm. In case of the problem he/she may refer to the Web resources.

**Problem 2:** Given a list of elements, implement the FIFO data structure using stacks.

**Solution:** To create a Queue using a stack, either of the following two approaches may be used. In the first approach, the push operation is implemented in the usual manner; however, the pop operation requires taking out everything from the first stack, putting it in the second stack, returning the top element of the second stack, and pushing back the remaining elements from the second stack to the first stack.

### #Queue from stack: method 1

```
1. def pop1(stack1, stack2, top1, max1):
2.     if top1== -1:
3.         return (stack1, top1, -2)
4.     elif top1== 0:
5.         top1=-1
6.         temp=stack1.pop()
7.         return (stack1, top1, temp)
8.     else:
9.         top1=-1
10.        while (top1!= -1):
11.            top1=-1
```

```
12.     temp=stack1.pop()
13.     stack2.append(temp)
14.     top2+=1
15.     top2-=1
16.     val=stack2.pop()
17.     while(top2!= -1):
18.         top2-=1
19.         temp=stack2.pop()
20.         stack1.append(temp)
21.         top1+=1
22.     return (stack1, top1, val)
23. def push1(stack1, top1, item, max1):
24.     if top1==max1 -1:
25.         print("Overflow")
26.     return (stack1, top1)
27. else:
28.     top1+=1
29.     stack1.append(item)
30.     return (stack1, top1)
31.
32. def init():
33.     stack1=[]
34.     stack2=[]
35.     MAX=5
36.     top1=-1
37.     top2=-1
38.     return(stack1, stack2, top1, top2, MAX)
```

The second approach requires the push operation to be altered. In this approach, the existing elements of the stack are pushed into another stack. The incoming element is then pushed into the first stack. Finally, all the elements from the second stack are pushed back into the first stack. The following program implements the solution.

### #Queue from stack: method 2

```
1. def push1(stack1, stack2, top1, top2, max1, item):
2.     if(top1==max1-1):
3.         print('Overflow')
4.     else:
5.         if(top1==-1):
6.             stack1.append(item)
7.             top1+=1
8.         else:
9.             while(top1!=-1):
10.                 top1-=1
11.                 temp=stack1.pop()
12.                 stack2.append(temp)
13.                 top2+=1
14.             top1+=1
15.             stack1.append(item)
16.             while(top2!=-1):
17.                 top2-=1
18.                 temp=stack2.pop()
19.                 stack1.append(temp)
20.             top1+=1
21.     return(stack1, stack2, top1, top2)
22. def pop1(stack1, top1, max1):
23.     if(top1==-1):
```

```

24.     return (stack1, top1, temp)
25. else:
26.     temp=stack1.pop()
27.     top1-=1
28.     return(stack1, top1, temp)
29.
30. def init():
31.     stack1=[]
32.     stack2=[]
33.     MAX=5
34.     top1=-1
35.     top2=-1
36.     return(stack1, stack2, top1, top2, MAX)

```

The exercises contain some more assorted problems on stacks. Remember that, stacks have just started!

## Conclusion

This chapter introduced stacks, which are linear data structures that follow the principle of Last In First Out. Stacks are used in converting infix expressions to postfix and prefix, in evaluating the postfix expressions, and for reversing strings. This chapter presents an informed discussion of the preceding topics. Some common problems related to stacks have also been discussed and implemented in the chapter.

The discussion continues in chapters on Queues and Linked lists, where some solutions will make use of stacks. The next chapter discusses Queues, which is a linear data structure that follows the principle of First In First Out. The implementation of Queues, circular Queues, and doubly ended Queues have also been included in the chapter. Your journey in becoming an accomplished programmer has just begun! The reader is advised to solve the problems given in the exercises to get a better hold of the topic.

## Multiple Choice Questions

1. The postfix expression  $4, 9, +, 5, 2, *, 4, /,-$  evaluates to
  - a. 4
  - b. 11.5
  - c. 0.75
  - d. None of the above
2. The postfix expression  $4, 9, * 5, 5, /,-$  evaluates to
  - a. 35
  - b. 0.2
  - c. 0.75
  - d. None of the above
3. A function  $f$  takes stack as an input argument and returns  $\min(f(S), i*i)$  if the stack is not null or 1 if the stack is null. Find the value  $f(S)$ , if  $S$  is  $[3, 1, 8, 9, 2, -1]$ .
  - a. 1
  - b. 2
  - c. 22
  - d. None of the above
4. In the preceding question, what is the value of,  $f(S)$ , if  $S$  is  $[-1]$ .
  - a. 0
  - b. 1
  - c. 2
  - d. None of the above
5. Which of the following data structures is best suited for evaluating recursive functions?
  - a. Stack
  - b. Queue
  - c. Both

d. None of the above

**6. Which of the following data structures is best suited for solving the stock span problem?**

- a. Stack
- b. Queue
- c. Both
- d. None of the above

**7. Which of the following can be solved using a stack?**

- a. Evaluating postfix expressions
- b. Converting expression to postfix
- c. Converting expression to prefix
- d. All of the above

**8. Consider the following algorithm,**

Input: Natural number  $n$

Stack,  $S = []$

Algorithm:

```
while( $n \neq 0$ )
    push ( $n \% 8$ ) to stack
    set  $n$  to  $n / 8$ 
while( $S$  is not null)
    pop from  $S$ 
```

What will be the output of the preceding algorithm?

- a. The octal equivalent
- b. The octal equivalent of a number in the reverse order
- c. The binary equivalent
- d. None of the above

**9. In the preceding algorithm if 8 is replaced by 2, what will be the output of the above?**

- a. The octal equivalent
  - b. The octal equivalent of a number in the reverse order
  - c. The binary equivalent
  - d. None of the above
10. In the linked list implementation of stack in C, what is the complexity of deletion and insertion?
- a. O(1) and O(1)
  - b. O(n) and O(1)
  - c. O(n) and O(1)
  - d. O(1) and O(1)
11. How many stacks are needed to implement a Queue (minimum number)?
- a. 1
  - b. 2
  - c. 3
  - d. None of the above
12. If two stacks are implemented using a single array, then for the most efficient implementation, which of the following can be true? (Symbols have usual meaning)
- a.  $\text{Top1} = \text{Top2} + 1$
  - b.  $\text{Top1} = 0, \text{Top2} = \text{Max} - 1$
  - c.  $\text{Top1} + \text{Top2} = \text{Max}$
  - d. None of the above
13. Which of the following permutation can be obtained in the same order using a stack, assuming that input is the sequence 1, 2, 3, 4, 5 in that order?
- a. 3, 5, 4, 2, 1
  - b. 1, 2, 3, 5, 4
  - c. 2, 1, 5, 4, 3

- d. None of the above
- 14. Which of the following can be used to efficiently convert a given decimal number to its hexadecimal equivalent?**
- a. Stack
  - b. Queue
  - c. Both
  - d. None of the above

## **Problems**

### **Level 1**

1. Write a program to implement stacks using arrays.
2. Write a program to evaluate a postfix expression.
3. Write a program to convert a given infix expression to its postfix equivalent.
4. Write a program to convert a given infix expression to its prefix equivalent.
5. Write a program to reverse a string using stacks.
6. Write a program to implement two stacks using a single list.
7. Write a program to implement a Queue using a stack?
8. For the preceding problem, can you suggest another method? Which of the two is better?

### **Level 2**

1. Write a program to convert a given decimal number to its binary equivalent.
2. Write a program to convert a given decimal number to its octal equivalent.
3. Write a program to convert a given decimal number to its hexadecimal equivalent.

4. Write a program to extract the minimum element in  $O(1)$  time using stacks.
5. Write a program to extract the middle element in  $O(1)$  time using stacks.

## Level 3

### Stock Span Problem

“The span of the stock’s price today is defined as the maximum number of consecutive days (starting from today and going backwards) for which the price of the stock was less than or equal to today’s price.”[1]

1. Write a program to find the span of a stock in a day.

### Non-recursive Merge Sort

2. Refer to the Merge Sort algorithm (Chapter: Sorting). Now, create a non-recursive procedure for the same using stacks.

### 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 7

## Queues

This chapter discusses a data structure called queues that follows the principle of **First In First Out (FIFO)**. In a simple queue, the elements are added at the rear end and removed from the front. This chapter discusses various types of queues, such as linear queues, circular queues, and doubly-ended queues. We then move to some exciting applications of queues, such as printing n-bit binary numbers, CPU scheduling, and creating a stack using a queue. The concepts studied in the chapter will be used in the following chapters as well. For example, Breadth-First Search, which is a type of Graph traversal, requires Queues.

### Structure

This chapter covers the following topics:

- Introduction to queues
- Circular queue and doubly-ended queue
- Generating binary numbers using a queue
- Implementing stack using two queue
- Implementing stack using a single queue

### Objectives

This chapter talks about queues, their efficient implementations, applications, generation of binary numbers, and implementation of stacks from queues. It forms the basis of the rest of the chapters and is immensely important. It is even used in CPU Scheduling and device drivers. The reader will learn the techniques of graph traversals using queues later in the book.

### Introduction

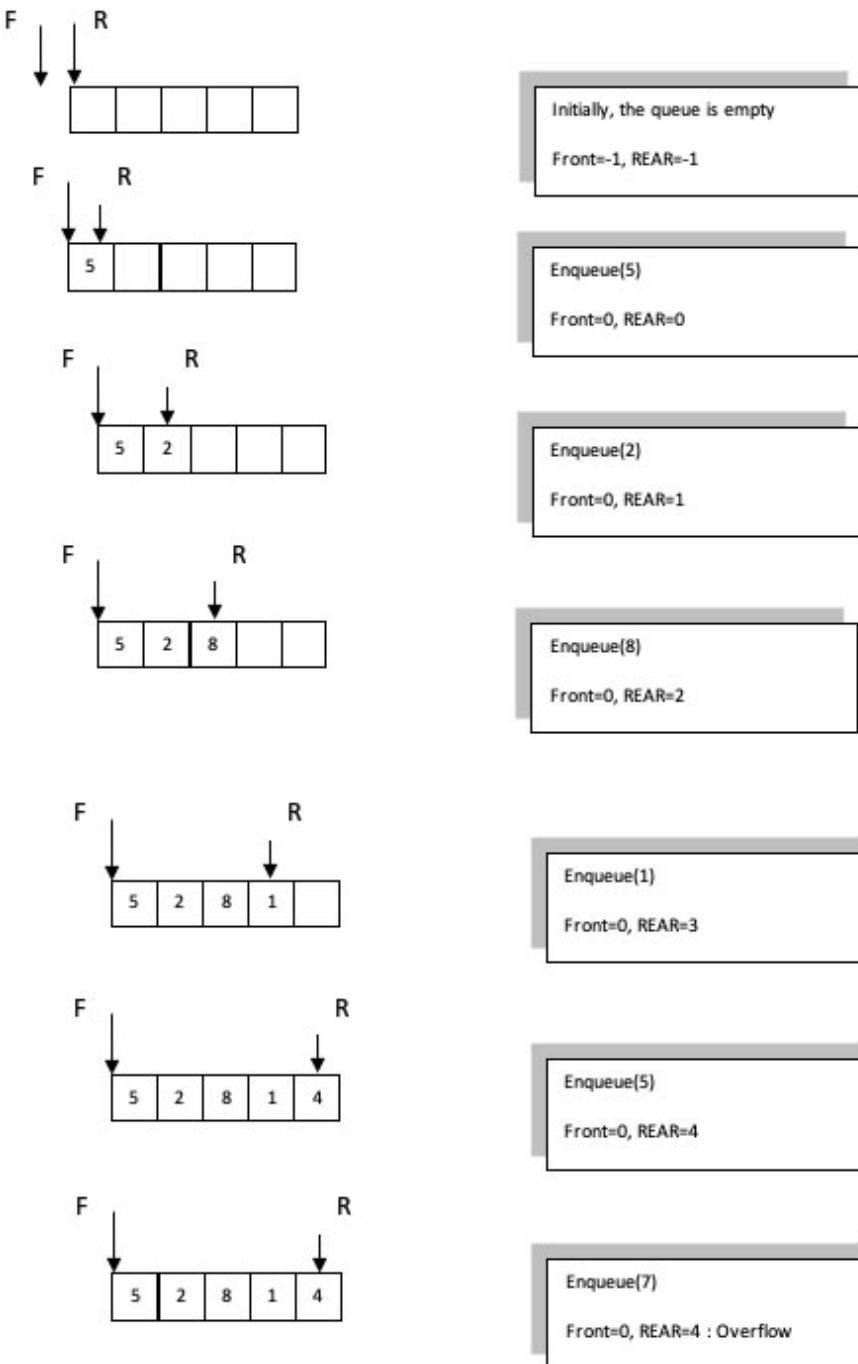
Imagine the following scenario. A new immunity booster is launched in the market by none other than one of the best actors the world has ever seen. It is exported to various countries, and we are particularly interested in two places: Shangri-La (of Lost Horizon by Hilton) and the Tri-State Area of the Phineas and Ferb Universe. The former is a utopian city, whereas the latter is, well, not utopic.

Shangri-La's local administration allows the sale of this amazing immunity booster in a shop, and people queue up to buy it. The one who comes first gets to buy it first; that means they follow First In First Out. This physical queue is like a data structure called a Linear Queue that follows the principle of FIFO. Note that the same principle is applied when you give print commands or in most of the device queues.

In the Tri-State Area, people also queue up to buy this amazing "medicine". They are amazed to see the great Dr. Heinz Doofenshmirtz in the queue. Haters question the veracity of his degree, but since they did not question all other things in the last century, so we ignore their concerns. To cover this big moment a "News Channel" called "Till today" sends its most honest journalists to request Dr. Doof to give a bite. He does, the reporters go away, and the shopkeeper begins by giving the medicine to Dr. Doof, who had 55 people in front of him, then. This is an example of a priority queue, which processes elements according to their priority. This is similar to the priority queues used in the processors. In Gotham City, they use circular queues for some reason. The details of which we will see later (the data structure, not the city).

Let us begin by taking an example of a Queue. In the example that follows ([figure 7.1](#)), initially, the queue is empty. The values of R (REAR) and F (FRONT) are initially -1. On inserting the first element in the queue, the values of F and R become 0. On further insertions, R is incremented by 1 till its value reaches MAX-1, after which "Overflow" occurs.

When the elements are popped, the value of the front is incremented by 1 till it becomes equal to R, after which both F and R become -1. Please refer to the following figure:



**Figure 7.1:** Inserting elements in a queue

Having seen the idea of Queues, let us now have a look at the formal algorithm and implementation of queues.

## Algorithm and implementation

A queue is a linear data structure that follows the principle of **FIFO**. For example, when we give the print commands, the printer processes the requests in the same order in which the commands are given (Spooling). Likewise, when the processor is assigned to different processes using the FIFO principle, the process which is admitted first is assigned to the processor earlier.

The following algorithm statically implements a queue. We need two indicators called FRONT (F) and REAR(R) for carrying out the preceding tasks.

Initially, the values of F and R are  $-1$ . When the first element is inserted, both F and R become  $0$ . Also, if the value of R is **MAX-1**, then no more elements can be added, and an error called “Overflow” occurs. In all other cases, R is incremented by 1, and a new item is added at R.

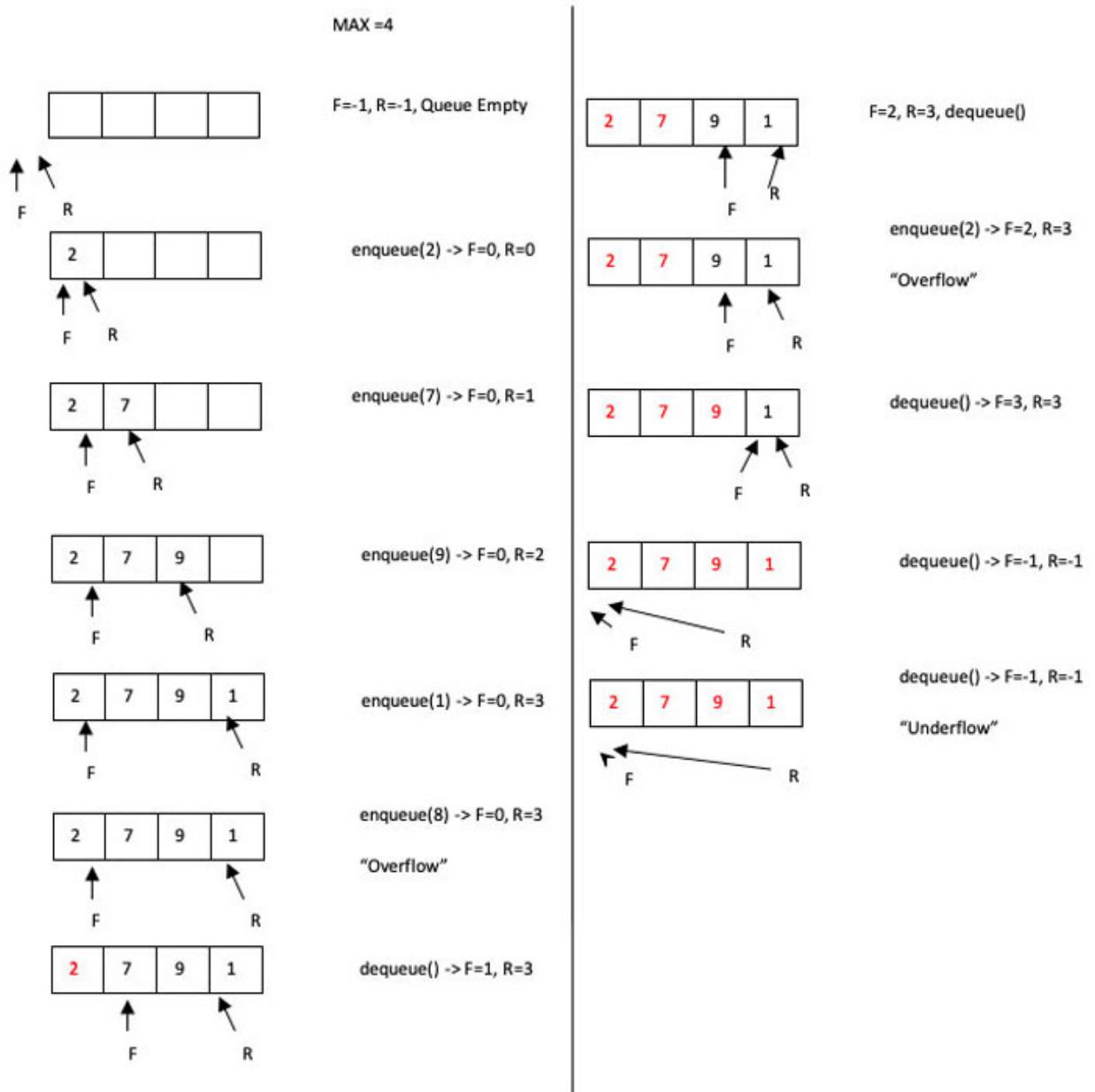
In the case of deletion, if both F and R have the same values, then both become  $-1$ . If both F and R are  $-1$ , then no more elements can be deleted, and an error called “Underflow” occurs. In all other cases, F is incremented by 1. [Figure 7.2](#) shows an example of a Queue. The algorithm is as follows.

### Algorithm:

Initially, the queue is empty, and the values of Rear (R) and Front (F) are  $-1$ . Also, the Q can have a maximum of MAX elements.

```
Q=[], R=-1, F=-1
enqueue(item)
{
    if((F==1) && (R==-1))
    {
        F=0;
        R=0;
        Q[R]=item;
    }
    else if(R== MAX-1)
    {
        print("Overflow");
    }
    else
```

```
{  
    Q[++R] = item;  
}  
}  
  
dequeue()  
{  
    if((F==1) and (R==-1))  
    {  
        temp=-1;//indicating the Q is empty  
    }  
    else if (F==R)  
    {  
        temp=Q[F];  
        F=-1;  
        R=-1;  
    }  
    else  
    {  
        temp=Q[F++];  
    }  
}
```



**Figure 7.2:** An example of a linear queue

**Problem:** Even if the spaces in the beginning are empty, adding an element may result in an “Overflow”. For example, in the preceding example, if  $F=2$  and  $R=3$ , inserting an element leads to “Overflow” although the first two positions do not contain elements (!).

The following program implements a queue. The menu-driven program asks for choice and inserts elements in a linear queue, takes out element, or display the queue as per the choice entered by the user.

## Code:

```
1. def enqueue(Q, item, F, R, MAX):  
2.     if(R==MAX-1):  
3.         print("Overflow")  
4.         return (F, R)  
5.     elif(R==-1):  
6.         F=0  
7.         R=0  
8.         Q[R]=item  
9.         return (F, R)  
10.    else:  
11.        R+=1  
12.        Q[R]=item  
13.        return (F, R)  
14. def dequeue(Q, F, R):  
15.    if(F==-1):  
16.        return (-1, F, R)  
17.    elif(F==R):  
18.        temp=Q[F]  
19.        Q[F]=0  
20.        F=-1  
21.        R=-1  
22.        return (temp, F, R)  
23.    else:  
24.        temp=Q[F]  
25.        Q[F]=0  
26.        F+=1  
27.        return (temp, F, R)
```

```
28.  
29. def disp(Q, F, R, MAX):  
30.     for i in range(MAX):  
31.         print(Q[i], end=' ')  
32.     print("Front=", F, " Rear=", R)  
33. #Main  
34. MAX=5  
35. Q=[0]*MAX  
36. F=-1  
37. R=-1  
38. item=int(input('Enter item\t'))  
39. (F, R)=enqueue(Q, item, F, R, MAX)  
40. disp(Q, F, R, MAX)  
41. item=int(input('Enter item\t'))  
42. (F, R)=enqueue(Q, item, F, R, MAX)  
43. disp(Q, F, R, MAX)  
44. (item, F, R)=dequeue(Q, F, R)  
45. if(item==-1):  
46.     print("Underflow")  
47. else:  
48.     print(item)  
49.  
50. disp()
```

The reader is expected to run the program and observe the output. Note that even if some locations in a linear Queue are empty, “Overflow” can still occur. The next section deals with this problem.

## Circular queue

To handle the preceding problem gracefully, we use **Circular Queues**, in which a Queue is treated as a circular data structure. In this case, the elements are added at  $(R+1) \% \text{MAX}$ . If  $(R+1) \% \text{MAX}$  is the same as F, then overflow occurs. The elements can be deleted if the value of R is not -1. Also, if the value of both F and R are the same, then F and R both become -1. In all other cases, the value of F becomes  $(F+1) \% \text{MAX}$ . The algorithm for the circular queue is as follows:

### Algorithm:

Initially, the queue is empty, and the values of Rear (R) and Front (F) are -1. Also, the Q data structure can have a maximum of MAX memory locations.

```
Q=[], R=-1, F=-1
Cenqueue(item)
{
    if((F==1) && (R==-1))
    {
        F=0;
        R=0;
        Q[R]=item;
    }
    else if((R+1)%MAX ==F)
    {
        print("Overflow");
    }
    else
    {
        R=(R+1)%MAX;
        Q[R] = item;
    }
}
Cdequeue()
{
    if((F==1) and (R==-1))
    {
        temp=-1;//indicating the Q is empty
```

```

    }
else if (F==R)
{
    temp=Q[F];
    F=-1;
    R=-1;
}
else
{
    temp=Q[F];
    F=(F+1)%MAX;
}
}

```

The following program implements a Circular Queue. The menu-driven program asks for choice and inserts elements in a circular Queue, takes out element, or display the queue as per the choice entered by the user.

### **Code:**

1. **def enqueue(Q, item, F, R, MAX):**
2.   **if((R+1)%MAX == F):**
3.     **print("Overflow")**
4.     **return (F, R)**
5.   **elif(R==-1):**
6.     **F=0**
7.     **R=0**
8.     **Q[R]=item**
9.     **return (F, R)**
10.   **else:**
11.     **R=(R+1)%MAX**
12.     **Q[R]=item**
13.     **return (F, R)**
14. **def dequeue(Q, F, R):**

```
15. if(F==-1):
16.     return (-1, F, R)
17. elif(F==R):
18.     temp=Q[F]
19.     F=-1
20.     R=-1
21.     return (temp, F, R)
22. else:
23.     temp=Q[F]
24.     F=(F+1)%MAX
25.     return (temp, F, R)
26. #Main
27. MAX=5
28. Q=[0]*MAX
29. F=-1
30. R=-1
31. item=int(input('Enter item\t'))
32. (F, R)=enqueue(Q, item, F, R, MAX)
33. disp(Q, F, R, MAX)
34. item=int(input('Enter item\t'))
35. (F, R)=enqueue(Q, item, F, R, MAX)
36. disp(Q, F, R, MAX)
37. item=int(input('Enter item\t'))
38. (F, R)=enqueue(Q, item, F, R, MAX)
39. disp(Q, F, R, MAX)
40. item=int(input('Enter item\t'))
41. (F, R)=enqueue(Q, item, F, R, MAX)
42. disp(Q, F, R, MAX)
```

```

43. item=int(input('Enter item\t'))
44. (F, R)=enqueue(Q, item, F, R, MAX)
45. disp(Q, F, R, MAX)
46. item=int(input('Enter item\t'))
47. (F, R)=enqueue(Q, item, F, R, MAX)
48. disp(Q, F, R, MAX)

```

Let us now move to doubly-ended queues, in which insertions and deletions can be done at both ends.

## Doubly-ended queue: DEQueue

It is a data structure in which insertions and deletions can be made from both ends. **FRONT** and **REAR**, are initialized to -1. When insertion is made in the **FRONT**, then **FRONT** becomes **FRONT -1**, provided it is not 0. Likewise, if insertion is done at the **REAR**, **REAR** becomes **REAR+1**, provided it is not **MAX-1**. In case of deletion from the **FRONT**, **FRONT** becomes **FRONT +1** (provided **FRONT** is not -1), and in case of deletion from the **REAR**, **REAR** becomes **REAR -1** (provided **REAR** is not -1). In case of deletion, an element is stored in temp and returned. In case there is no element in the **queue**, -1 is returned.

### **Algorithm: Doubly-ended queue**

```

//INSERT AT THE BEGINNING
enqueue_front(item)
{
    if(FRONT!= 0)
    {
        FRONT=FRONT-1;
        Q[FRONT]= item;
    }
    else
    {
        print("Cannot insert at the beginning");
    }
}

```

```

//INSERT AT THE END
enqueue_rear(item)
{
    if( REAR != MAX-1 )
    {
        REAR=REAR +1;
        Q[REAR]= item;
    }
    else
    {
        print("Cannot insert at the end");
    }
}
//DELETE FROM THE BEGINNING
dequeue_front()
{
    if(FRONT!= -1)
    {
        if(FRONT == REAR)
        {
            temp=Q[FRONT]
            FRONT=-1;
            REAR=-1
        }
        else
        {
            temp=Q[FRONT];
            FRONT=FRONT+1;
        }
    }
    else
    {
        print("Cannot delete from the beginning");
        temp=-1
    }
}

```

```

    return temp;
}
//DELETE FROM THE END
dequeue_rear()
{
    if( REAR!= -1)
    {
        if(FRONT==REAR)
        {
            temp=Q[REAR];
            FRONT=-1;
            REAR=-1;
        }
    else
        {
            temp=Q[REAR];
            REAR=REAR-1;
        }
    }
    else
    {
        print("Cannot delete item");
        temp=-1
    }
    return temp;
}

```

The web resources contain the program that implements a doubly-ended queue. The menu-driven program asks for choice and inserts elements in a DEQueue, takes out element, or display the queue as per the choice entered by the user.

## **Stack and queue using DEQueue**

To create a stack using a **DEQueue**, the push operation can be implemented using the **enqueue\_rear** function, and the pop operation can be implemented

using the `dequeue_rear` function. Likewise, to create a Queue using a `DEQueue`, the enqueue operation can be implemented using the `enqueue_rear` function, and the dequeue operation can be implemented using the `dequeue_front` function.

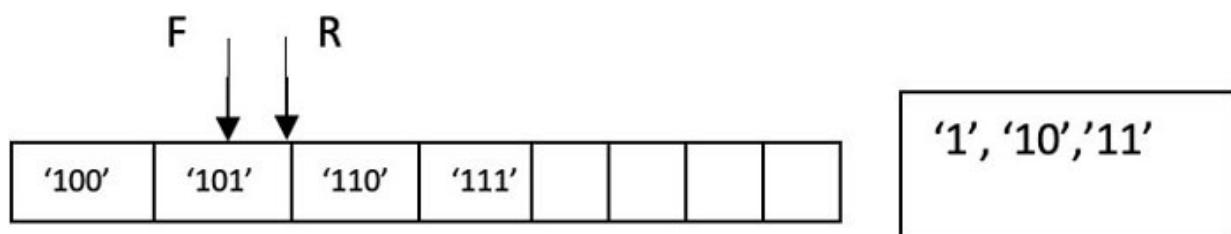
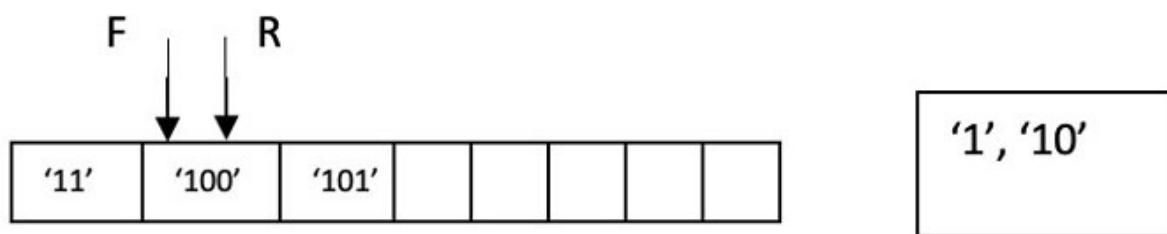
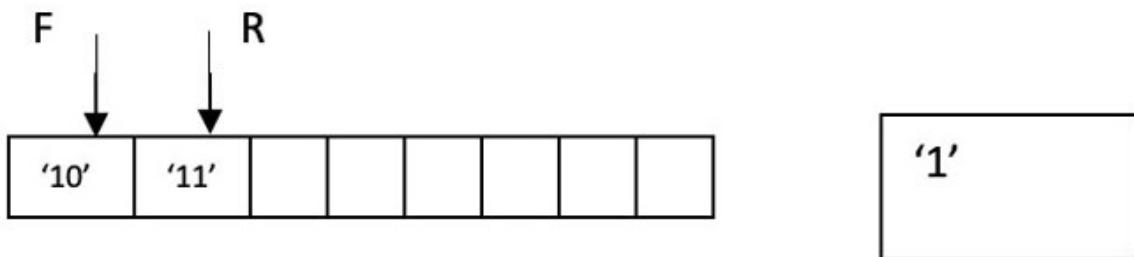
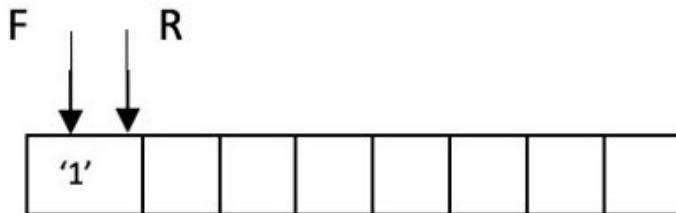
Let us now move to an exciting application of a Queue, which is generating Binary Numbers.

## Generating binary numbers using a queue

In this section, we will generate n-bit binary numbers using Queues. We will use a single Queue in the procedure. Initially, the queue is empty. Then we insert ‘1’ in the queue. Till there are elements in the queue, we dequeue an element from the queue (“x”) and insert “x0” and “x1” in the queue.

Let us see what happens if we repeat this process  $2^n - 1$  times.

To understand the process, let us take the value of  $n$  as 3. The first four steps of the algorithm are shown in [figure 7.3](#). Note that, initially, a ‘1’ is inserted in the queue. In the next step, ‘1’ is dequeued and printed. Now, ‘10’ and ‘11’ are inserted in the queue. The queue now has ‘10’ and ‘11’. In the next step, ‘10’ is dequeued and printed. Also, ‘100’ and ‘101’ are inserted in the queue. If the process continues  $2^n - 1$  times, n-bit binary numbers are generated. The code implementing this algorithm follows.



*Figure 7.3: Generating binary numbers using a queue*

### Code:

1. #Implement Queue
2. Q=[]
3. choice=0
4. F=-1
5. R=-1
6. MAX=5
7. while choice!=4:

```
8. print('Queue\n1\t:Enqueue\n2\t:Dequeue\n3\t:Dispaly\n4:Exit
 \nEnter choice\t:')
9. choice=int(input('Choice\t:'))
10. if choice==1:
11.     item=int(input('Entre item\t:'))
12.     if R==-1:
13.         R+=1
14.         F+=1
15.         Q.append(item)
16.     elif R==(MAX-1):
17.         R+=1
18.         Q.append(item)
19.     else:
20.         print('Overflow')
21. elif choice==2:
22.     if R==-1:
23.         print('Underflow')
24.     elif R==F:
25.         R=-1
26.         F=-1
27.         item=Q.pop(0)
28.         print(item,' popped')
29.     else:
30.         F+=1
31.         item=Q.pop(0)
32.         print(item,' popped')
33. elif choice==3:
34.     print('F=' , F, ' R=' , R)
```

```

35.     print('Queue')
36.     for i in Q:
37.         print(i, end=' ')
38.     elif choice==4:
39.         print('Program ends...')
40.     else:
41.         print('Incorrect choice')
42. #Main
43. Q=[]
44. n=int(input('Enter number of digits\t'))
45. print(n, ' bit binary numbers')
46. Q.append('1')
47. for i in range(2**n-1):
48.     item=Q.pop(0)
49.     print(item)
50.     Q.append(item+'0')
51.     Q.append(item+'1')

```

**Octal numbers** are numbers that use eight digits {‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’}. The counting in this number system reads like this:

‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘10’, ‘11’, ‘12’, ‘13’, ‘14’, ‘15’, ‘16’, ‘17’, ‘20’, ‘21’, ‘22’, ‘23’, ‘24’, ‘25’, ‘26’, ‘27’,... and so on.

Now, can you think of a procedure that generates octal numbers using the preceding method? Now, go back to the fictitious “Hexal numbers” studied in the previous chapter. Try to generate  $n$ -digit Hexal numbers using Queues. The following code may help you.

## Fictitious Hexal Numbers

### Code:

```

1. Q=['1','2','3','4','5']
2. n=int(input('Enter number of digits\t'))

```

```
3. print(n, ' hexal binary numbers>)
4. for i in range(6**n-1):
5.     item=Q.pop(0)
6.     print(item)
7.     Q.append(item+'0')
8.     Q.append(item+'1')
9.     Q.append(item+'2')
10.    Q.append(item+'3')
11.    Q.append(item+'4')
12.    Q.append(item+'5')
```

Let us now implement a stack using queue (s).

## Stack using two queues

To create a stack using two queues, say Q1 and Q2, first, the incoming element is added to Q2. Then all the elements are transferred from Q2 to Q1. This is followed by transferring all the elements from Q1 to Q2. In the example that follows, Initially, 1 is inserted in the queue, followed by the insertion of 2 in Q2, then transferring all the elements of Q2 to Q1 ([2, 1]), and then transferring all the elements of Q1 to Q2.

### **Example:**

#### **Step 1:**

```
Q1=[], Q2=[]
Q2=[1]
Q2=[], Q1=[1]
```

#### **Step 2:**

```
Q2=[2], Q1=[1]
Q2=[2, 1], Q1=[]
Q2[], Q1=[2, 1]
```

#### **Step 3:**

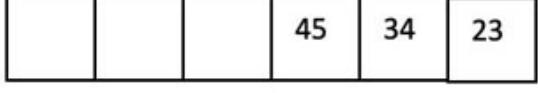
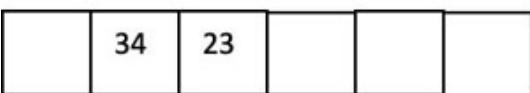
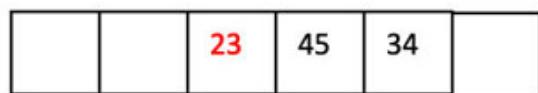
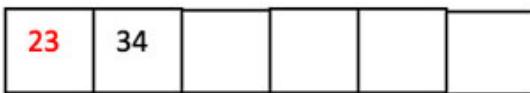
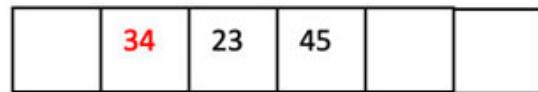
```
Q2=[3], Q1=[2, 1]
Q2=[3, 2, 1], Q1=[]
Q2[], Q1=[3, 2, 1]
```

The reader is expected to implement the algorithm. The web resources of this book, though, contain the program. Let us now see a slightly more complicated way of doing the same thing.

## Stack from a single queue

To create a stack from a single Queue, the **enqueue** function of a queue can be tweaked to create a **push** function. Here, first, the item is inserted in the usual way; second, taking out all the elements, except the item recently added, one by one and inserting it at the end of the queue.

In the example given in [figure 7.4](#), first, 23 is inserted in the queue, as usual. On inserting 43, all the elements except 34 are dequeued and enqueued. Thus, pushing 23 to the end. The same procedure is followed while adding 45. The program that follows implements a stack using a single Queue.



First, 23 is inserted in the Queue, as usual. On inserting 34, all the elements except 34 are dequeued and enqueued. Thus, pushing 23 to the end

On inserting 45, all the elements except 45 are dequeued and enqueued. Thus, pushing 23 to the end

**Figure 7.4:** Creating stack from a single queue

**Code:**

```
1. #Enqueue
2. def enqueue(Q, F, R, MAX, item):
3.     if F== -1:
4.         R+=1
5.         F+=1
6.         Q.append(item)
7.     elif R!=(MAX-1):
8.         R+=1
9.         Q.append(item)
10.    else:
11.        print('Overflow')
12.    return (Q, F, R)
13. #Dequeue
14. def dequeue(Q, F, R):
15.    if F== -1:
16.        print('Underflow')
17.        item=-1
18.    elif R==F:
19.        R=-1
20.        F=-1
21.        item=Q.pop(0)
22.        print(item, ' popped')
23.    else:
24.        F+=1
25.        item=Q.pop(0)
26.        print(item, ' popped')
```

```

27.    return (Q, F, R, item)
28. #Push
29. def push(Q, F, R, MAX, item):
30.     Q, F, R= enqueue(Q, F, R, MAX, item)
31.     n=R-F
32.     for i in range(n):
33.         Q, F, R, item\=dequeue(Q,F, R)
34.         if(item1!=\-1):
35.             Q, F, R=enqueue(Q, F, R, MAX, item\)
36.     return (Q, F, R)
37. #Main
38. q=[]
39. choice=\0
40. F=\-1
41. R=\-1
42. MAX=10
43. while choice!=4:
44.     print('Queue\n1\t:Enqueue\n2\t:Dequeue\n3\t:Dispaly\n4\t:Exit\nEnter choice\t:')
45.     choice=int(input('Choice\t:'))
46.     if choice==1:
47.         item=int(input('Entre item\t:'))
48.         Q, F, R = push(Q, F, R, MAX, item)
49.     elif choice==2:
50.         Q, F, R, item\= dequeue(Q, F, R)
51.     elif choice==3:
52.         print('F=\', F, ' R=\', R)
53.         print('Queue')

```

```

54.     for i in Q:
55.         print(i, end=' ')
56.     elif choice==4:
57.         print('Program ends...')
58.     else:
59.         print('Incorrect choice')

```

This was fun! Let us now look at scheduling.

## Scheduling

In a multi processing environment, a single CPU is allocated to various processes. If the processor is allocated in a First Come First Serve manner, and the processes that require a large time get the CPU in the beginning, the others may have to wait for a long, even when they require a little time to complete. To handle this, Round Robin may be used. This algorithm gives the CPU to each process for a fixed time. After every process has got the CPU, the cycle starts again till all the processes are over. The following code implements Round Robin. The reader is expected to observe the output to get hold of the mechanism.

### **Code:**

```

1. Q=[]
2. F=-1
3. R=-1
4. MAX=100
5. quantum = int(input('Enter the quantum\t'))
6. n=int(input('Enter the number of processes\t'))
7. b_t=[]
8. for i in range(n):
9.     str1='Enter burst time of process'+str(i)+'\t'
10.    burst_time=int(input(str1))
11.    b_t.append(burst_time)

```

12. `temp=n*[0]`

**Output:**

```
Enter the quantum 4
Enter the number of processes 5
Enter burst time of process0 12
Enter burst time of process1 8
Enter burst time of process2 6
Enter burst time of process3 5
Enter burst time of process4 9
```

## **Conclusion**

This chapter discussed Queues and their types. The linear Queues end up wasting memory spaces and may prove inefficient, particularly in terms of memory. To handle this, we may use circular Queues. This chapter also discussed Doubly Ended Queues. Some applications of Queues, such as generating binary numbers, creating stacks, scheduling, and so on, have been discussed in the chapter. As stated earlier, the concepts studied in the chapter will be used in Trees and Graphs also. In [chapter 5](#), we discussed Linked List and again implemented Stacks and Queues using Linked Lists. The reader is advised to attempt the problems given at the end of the chapter to get hold of the ideas explored in the chapter.

In the next chapter, we will discuss the concepts of Trees.

## **Multiple Choice Questions**

**1. A data structure in which insertion can be done from the end and deletion from the beginning is called**

- a. Stack
- b. Queue
- c. Tree
- d. None of the above

**2. A data structure in which insertion can be done from the beginning and deletion from the end is called**

- a. Stack
- b. Queue
- c. Tree
- d. None of the above

**3. What is a Ring Buffer?**

- a. Circular Queue
- b. Linear Queue
- c. Graph
- d. None of the above

**4. Which data structure follows the principle of FIRST IN FIRST OUT (FIFO)?**

- a. Queue
- b. Stack
- c. Tree
- d. None of the above

**5. Which of the following depicts the condition for overflow in a circular Queue?**

- a.  $(\text{REAR} + 1) \% \text{MAX} = \text{FRONT}$
- b.  $(\text{FRONT} + 1) \% \text{MAX} = \text{REAR}$
- c.  $\text{FRONT} == \text{REAR}$
- d.  $\text{REAR} = \text{MAX} - 1$

**6. Which of the following depicts the condition for underflow?**

- a.  $\text{REAR} == -1$
- b.  $\text{FRONT} == -1$
- c. Both
- d. None of the above

**7. Which of the following can be implemented using queues?**

- a. Breadth-First Search

- b. Depth First Search
- c. Both
- d. None of the above

**8. Which of the following uses queue (s)?**

- a. Spooling
- b. Round Robin
- c. Both
- d. None of the above

**9. Which of the following does not use queues?**

- a. Recursion
- b. Backtracking
- c. Both use queues
- d. None of the two use queues

**10. Which of the following can be used to implement Priority Queues?**

- a. Heap
- b. Binary Search Tree
- c. Graph
- d. None of the above

**11. What is the minimum number of stacks needed to implement a Queue?**

- a. 1
- b. 2
- c. 3
- d. None of the above

**12. What is the minimum number of queues needed to implement a Stack?**

- a. 1
- b. 2

- c. 3
- d. None of the above

## **Problems**

### **Level 1**

- 1. What is a Queue?**
  - a. Write an algorithm to implement a linear Queue.
  - b. What is the problem with this data structure?
  - c. State at least three applications of a linear Queue.
- 2. What is a circular Queue?**
  - a. Write an algorithm to implement a circular Queue.
  - b. State a few applications of the circular queue.
- 3. What is a Doubly Ended Queue?**
  - a. Write an algorithm to implement the doubly-ended queue.
  - b. State a few applications of doubly-ended queue.
- 4. Write an algorithm to implement a stack using a Queue.**

### **Level 2**

1. Write an algorithm to implement a stack using two queues.
2. Write an algorithm to print n-bit octal numbers using a queue.
3. Write an algorithm to reverse a queue.

### **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 8

## Trees-I

### Introduction

Consider the family tree of Simpsons: Abe Simpson has two sons, Homer and Herb; Homer has three children, Bart, Lisa, and Maggie. Since December 17, 1989, there has been no addition or subtraction in this tree. For those of us who grew up watching The Simpsons, Abe is the root of this family tree, and Bart, Lisa, and Maggie are the leaves. If you decide to watch this series from the beginning, this information would be helpful. Likewise, four generations of the late Prithviraj Kapoor, who established the Prithvi theatre in 1944, have been active in the Hindi film industry. These family trees have been part and parcel of the lives of those who are into Cinema. These trees are a subset of general trees. The other subsets include decision trees, search trees, parse trees, and expression trees (to name a few), which are the soul of various branches of computing. So, let us dive into this amazing data structure.

So far, we have studied linear data structures like Stacks and Queues. This chapter introduces the reader to trees, which is a non-linear data structure. A tree is essentially a graph having no cycle or no isolated edge or vertex. Generally, it is used to depict a hierarchical relationship between the nodes. Trees find applications in numerous problems: from compression using the Huffman algorithm to expression evaluation, from semantic analysis to Machine Learning; trees are used everywhere.

The chapter explains the fundamentals and then moves to a special type of tree called Binary Trees. The traversals of Binary Trees are followed by a searching, insertion, and deletion in Binary Search Trees. This is followed by an introduction to B trees.

### Structure

In this chapter, we will cover the following topics:

- Definition and representation of trees
- Binary trees

- Binary search trees

## Objectives

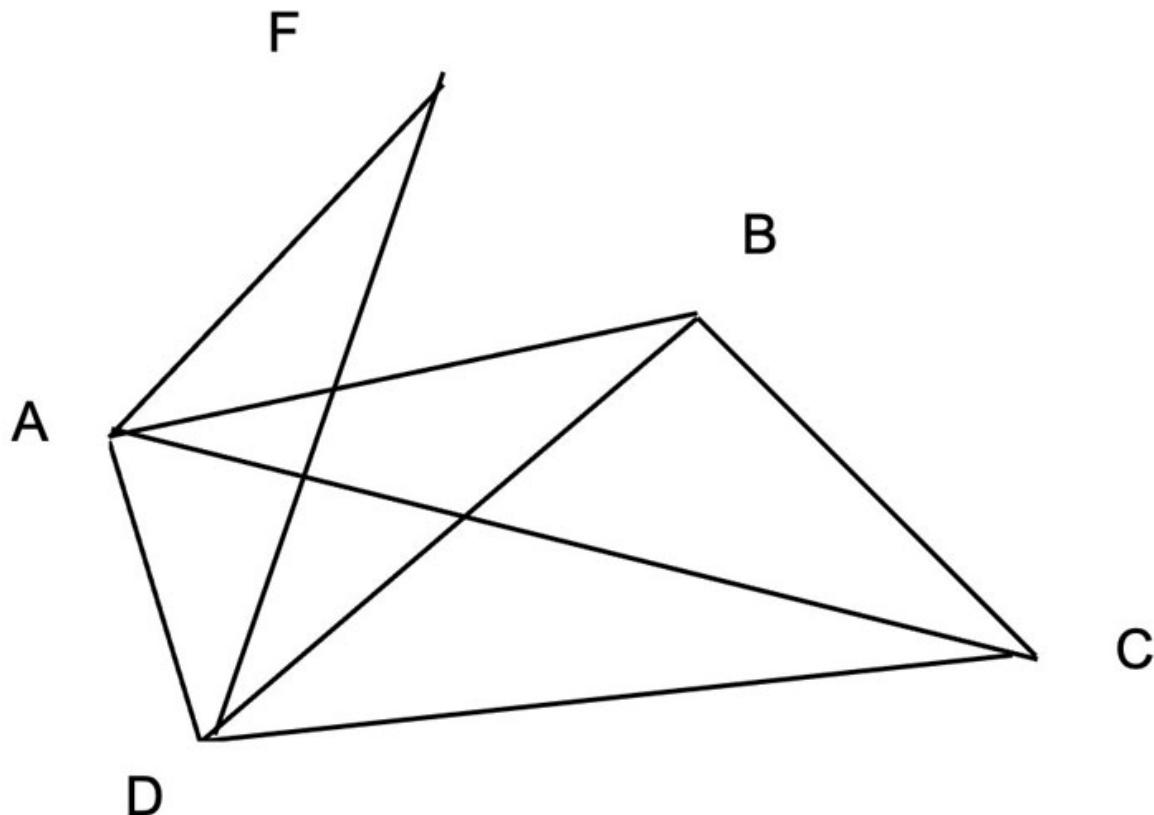
After reading this chapter, the reader will be able to understand the representation and the terminology of Trees. He/she will be able to define and create a Binary Search Tree and traverse these trees. The chapter also explains how to insert and delete elements in a Binary Search Tree and find the maximum and minimum from a tree. The reader will also learn to apply the preceding methods to solve problems.

## Definition and terminology

As stated earlier, trees are a subset of graphs, so let us begin with the definition of a Graph.

**Graph:** A graph is a set  $(V, E)$ , where  $V$  is a finite, non-empty set of vertices. The set  $E$  is a set consisting of tuples  $(x, y)$ , where  $x$  and  $y$  belong to set  $V$ .

Figure 8.1 shows a graph  $G = (V, E)$ , where  $V$  is  $\{A, B, C, D, F\}$  and the set  $E$  is  $\{(A, B), (A, C), (A, D), (A, F), (B, C), (B, D), (C, D), (D, F)\}$ :

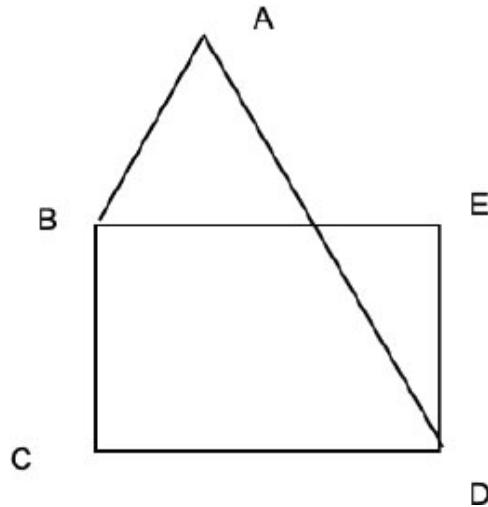


**Figure 8.1:** The graph  $G = (V, E)$ , where  $V = \{A, B, C, D, F\}$  and  $E = \{(A, B), (A, C), (A, D), (A, F), (B, C), (B, D), (C, D), (D, F)\}$ .

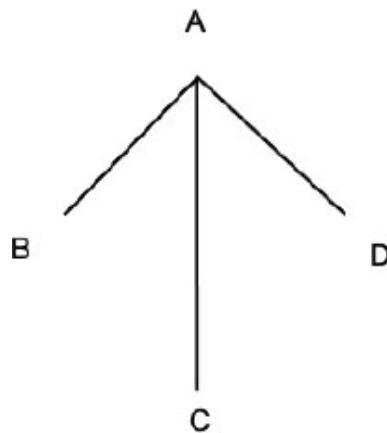
A graph is one of the most important data structures that is used in a wide variety of applications like finding shortest paths, ranking Web pages, and so on. As a matter of fact, there is a dedicated subject on graph algorithms. [Chapter 10, Priority Queues](#) of this book, discusses Graphs and their applications.

**Tree:** A Tree is a Non-Linear Data Structure. It is basically a graph that does not form any cycle and does not have isolated edges or vertices. Though, a tree can have a single vertex, in which case it will be the root.

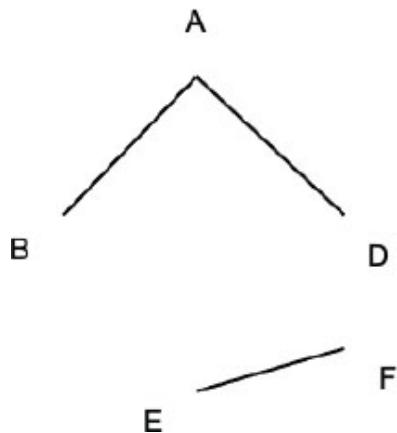
[Figure 8.2\(a\)](#) shows an example of a graph that is not a tree, as there exists a cycle in this graph. [Figure 8.2\(b\)](#) shows a tree as it does not contain any cycle or isolated vertex or edge. The graph in [figure 8.2\(c\)](#) is not a tree, as it contains an isolated edge:



**Figure 8.2 (a)**



**Figure 8.2 (b)**

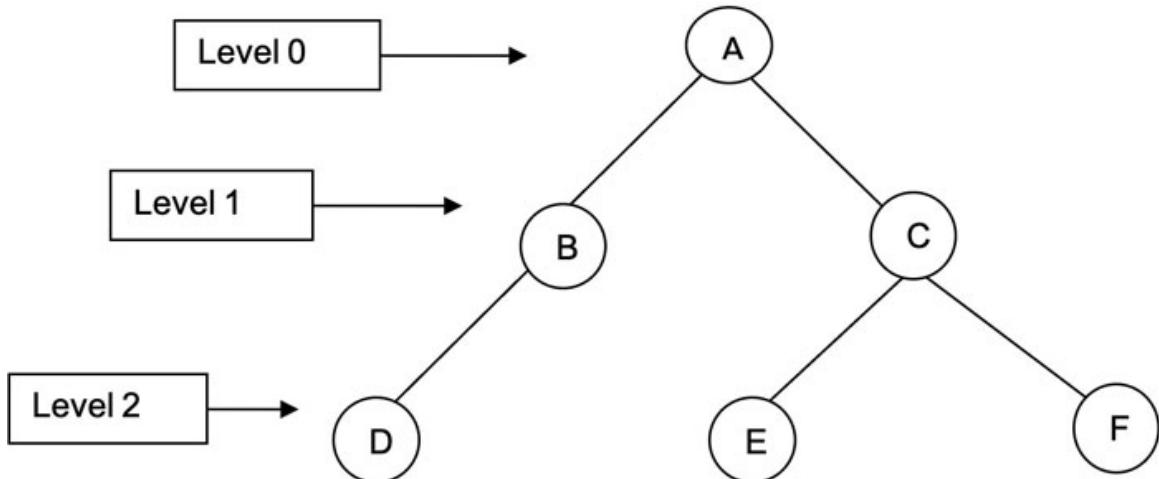


*Figure 8.2 (c)*

In this chapter, we will focus on the subset of trees called Binary Trees.

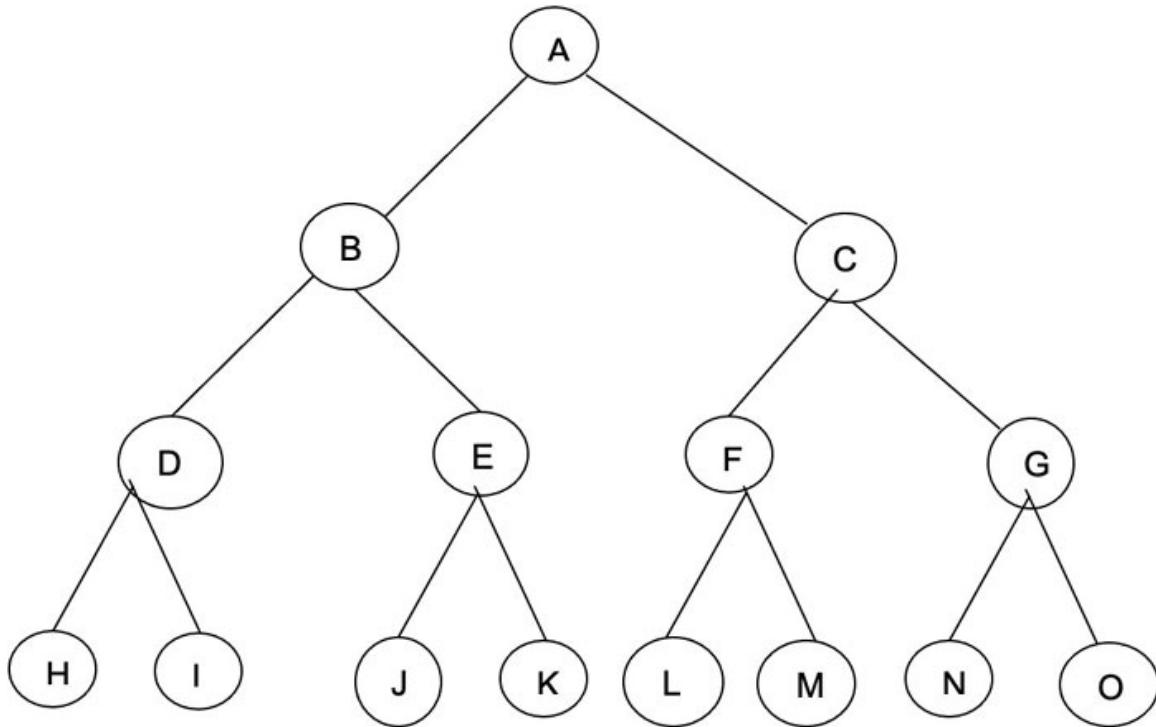
**Binary Tree:** A binary tree is one in which each node can have a maximum of two children.

The tree shown in [figure 8.3](#) is a binary tree, as each node has 0, 1, or 2 children. Note that node A is at level 0, nodes B and C are at Level 1, and nodes D, E, and F are at Level 2. If each node of a tree has two children except for possibly the last level, it is called a complete binary tree. Please refer to the following figure:



*Figure 8.3: Binary tree, each node has 0, 1, or 2 children*

[Figure 8.4](#) shows a **Complete Binary Tree**. The root of a tree is always at Level 0, the children of the root at Level 1, and so on. Please refer to the following figure:



**Figure 8.4:** Complete Binary Tree, each node has two children except for the leaves

The complete binary tree has a node at Level 0, two nodes at Level 1, four at level 2, and so on. The total number of nodes in a complete binary tree with  $l$  levels is  $2^l - 1$ .

The vertices of a tree would henceforth be referred to as nodes. A tree with a node designated as a root is called a **rooted tree**. Each node in a rooted tree can have a parent (except root) and children (0 or more). [Figure 8.5](#) shows a rooted tree in which A is the root, B and C are the children of A, and consequently, A is the parent of B and C. The nodes having the same parent are called **siblings**. The nodes having no child are called **leaves**. In [figure 8.5](#), the leaves are B, E, G, and H. The root is at Level 0. The children of the root are at Level 1, and so on. Examples of rooted trees include family trees, folders, sub-folder organization, and hierarchical structures in institutes. For a rooted tree, the terminology follows:

**Node:** The vertex of the tree (abstract data type) is referred to as a node.

**Root:** It is a node that does not have any parent.

**Siblings:** The nodes having the same parent are called siblings.

**Leaf:** A node with no children is called a leaf.

**Internal node:** A node that has child(children) is called an internal node.

**Depth:** The root node is at depth 0, its children at depth 1, and so on.

**Length of path:** A path is a sequence of vertices from the source node to the destination node. The number of nodes in a path minus 1 is the length of the path.

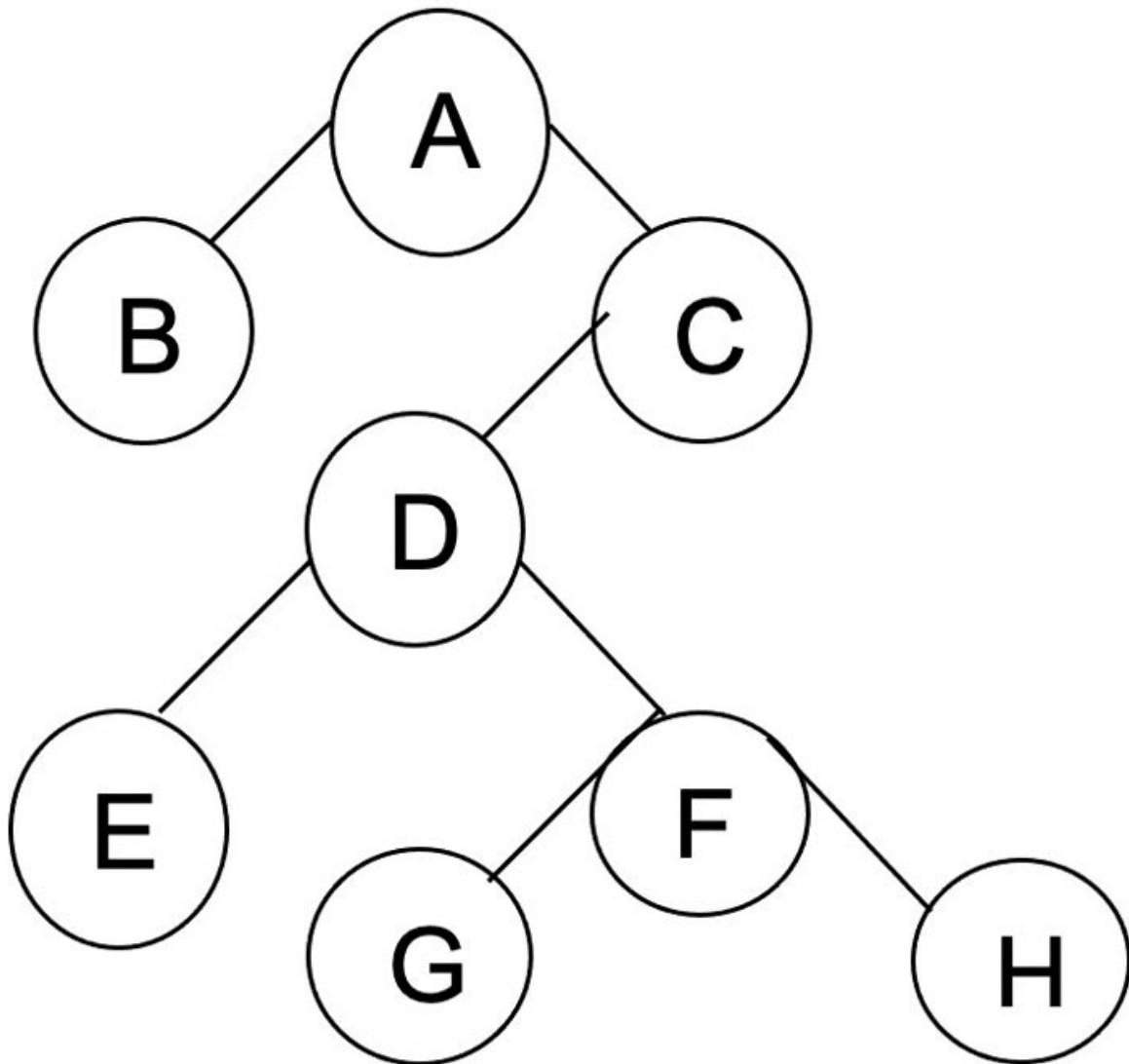
**Ancestor:**  $x$  is an ancestor of  $y$  if there is a path from  $x$  to  $y$ . Here,  $y$  is the descendent of  $x$ .

**Rooted tree:** This tree has either no node or a root node having 0 or more subtrees.

Having seen the terminology, let us move to the representation of a Binary Tree.

## Representation of a Binary Tree

A binary tree can be stored in a computer using an array or a linked list. The array representation of a binary tree requires the root to be stored at the 0<sup>th</sup> index. For each node stored at the  $n$ th index, its left child would be stored at the  $(2n+1)$ <sup>th</sup> index and the right child at  $(2n+2)$ <sup>th</sup> index. Please refer to the following figure:



**Figure 8.5:** Calculation of index for the array representation of a Binary Search Tree

For example, in the preceding tree ([figure 8.5](#)), the root would be stored at the 0<sup>th</sup> index. The left child of the root (B) would be stored at the index, given by the formula,  $2 \times n + 1 = 2 \times 0 + 1 = 1$ .

The right child of the root would be stored at the index, given by the formula,

$$2 \times n + 2 = 2 \times 0 + 2 = 2$$

That is, B would be stored at the first index and C at the second index. Likewise, the left child of C would be stored at the index given by the formula:

$$2 \times n + 1 = 2 \times 2 + 1 = 5$$

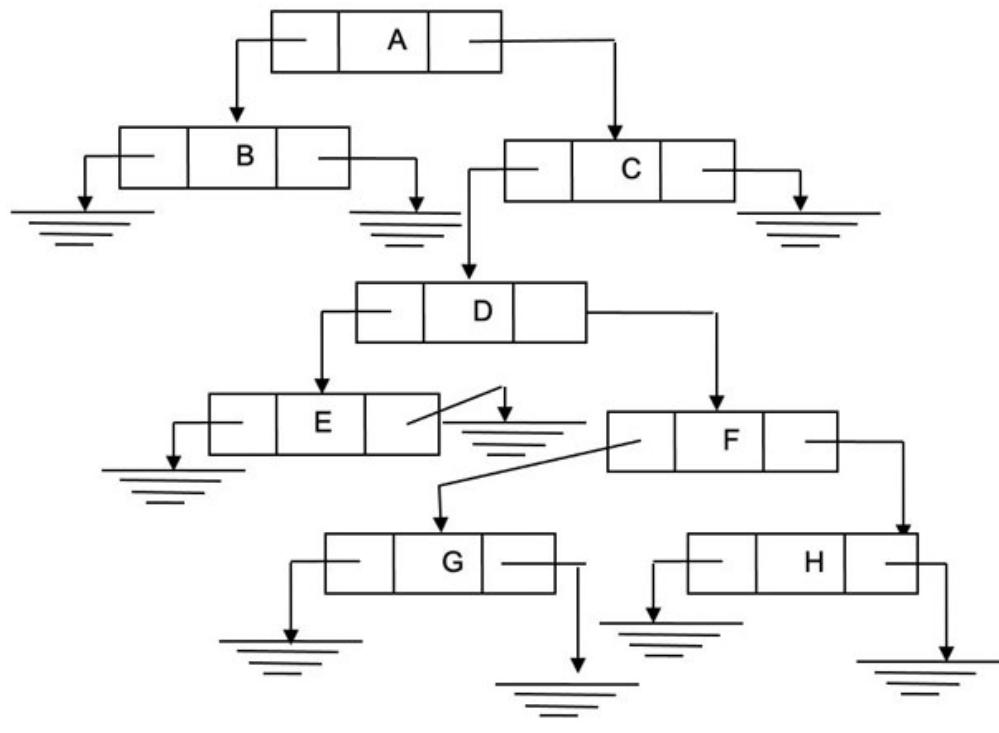
The left child of D would be stored at the 11<sup>th</sup> index, and the right child would be stored at the 12<sup>th</sup> index. Finally, the left and the right child of F would be stored at the 23<sup>rd</sup> and the 24<sup>th</sup> index. The array representation of the tree in [figure 8.5](#) is, therefore, as follows:

A	B	C			D					E	F			G	H
0	1	2	3	4	5	6	7	8	9	10	11	12	.....	23	24

*Table 8.1: Representation of the tree shown in [figure 8.5](#)*

Note that the representation is quite inefficient in terms of space. The eight values are stored in an array that has been allocated to at least 24 memory locations. That is, a lot of space is wasted if the given tree is not completely balanced. Note that in the case of a completely balanced tree, (almost) no space would be wasted in the array representation.

A tree can also be stored using the doubly-linked list. In the representation, a node's Left Child's address is stored at the previous pointer, and its Right Child's address is stored at the next pointer. [Figure 8.6](#) shows the linked list representation of the tree:



Represents

**Figure 8.6:** Linked list representation of a Binary Tree

Having seen the representation, let us move to the traversal of a tree.

## Traversal

A Binary tree can be traversed in three ways:

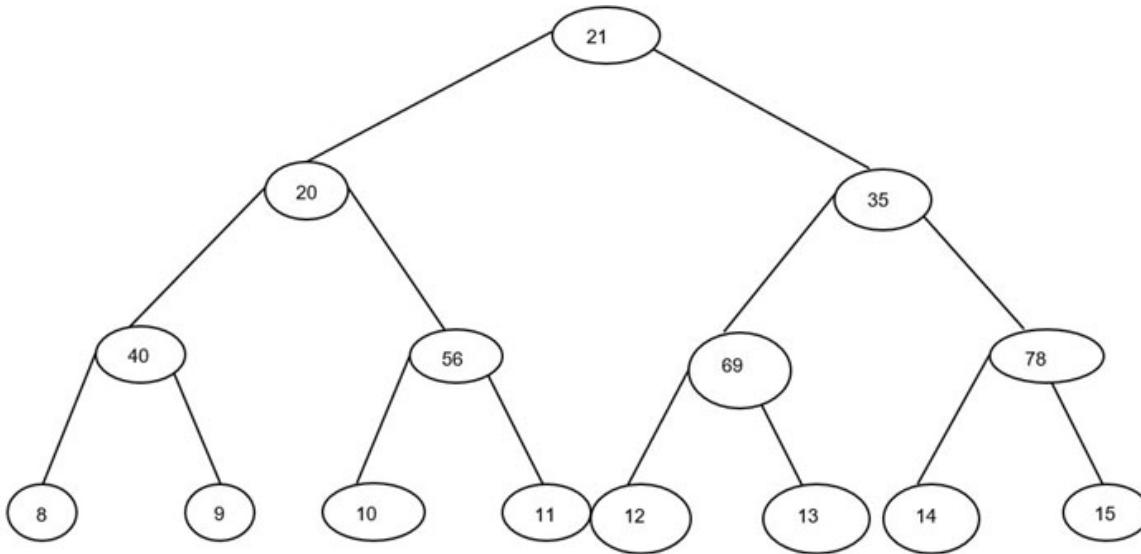
1. In-order
2. Post-order
3. Pre-order

### In-order traversal

The **in-order** traversal traverses the left sub-tree in **in-order** followed by processing the data at the root and, finally, the right sub-tree in **in-order**. The following recursive procedure can be used to print the **in-order** traversal of a given tree:

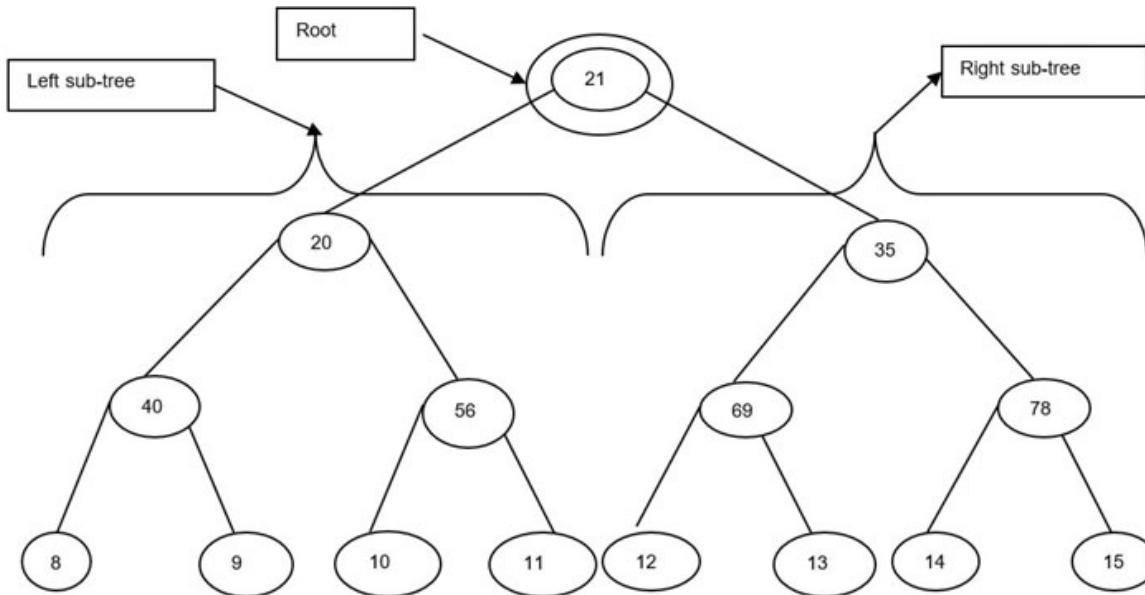
```
def inorder(T):  
    if (T == None) :  
        return  
    elif ((T.left==None ) and (T.right ==None)):  
        print(T.data)  
    else:  
        inorder(T.left)  
        print(T.data)  
        inorder(T.right)
```

Let us try to unwind the preceding procedure. Consider the tree shown in [figure 8.7](#). The **in-order** traversal prints the **in-order** traversal of the left subtree, followed by the data at the root (21, in this case) and then the **in-order** traversal of the right sub-tree.



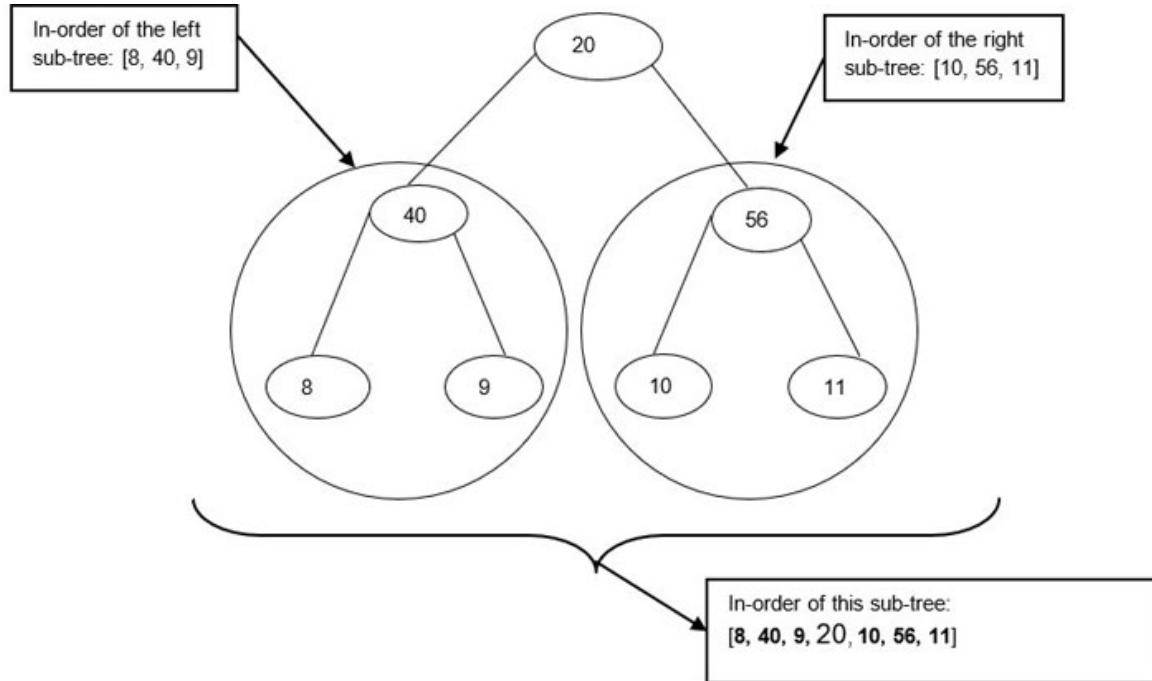
**Figure 8.7:** A binary tree

[Figure 8.8](#) depicts the procedure. Note that this procedure needs to be applied to the left and the right sub-tree as well:



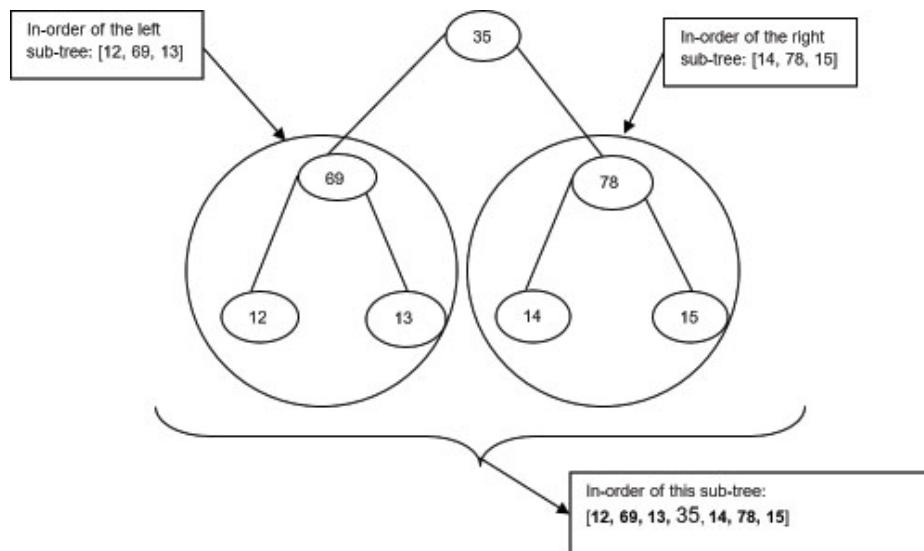
**Figure 8.8:** In-order traversal of tree in [figure 8.7](#)

The **in-order** traversal of the left subtree ([figure 8.9](#)), shown in [figure 8.7](#), prints the **in-order** traversal of the left tree ([8, 40, 9]), followed by the data at the root (20 in this case), and then the **in-order** traversal of the right sub-tree ([10, 56, 11]). The **in-order** traversal of the left sub-tree, therefore, becomes [8, 40, 9, 20, 10, 56, 11].



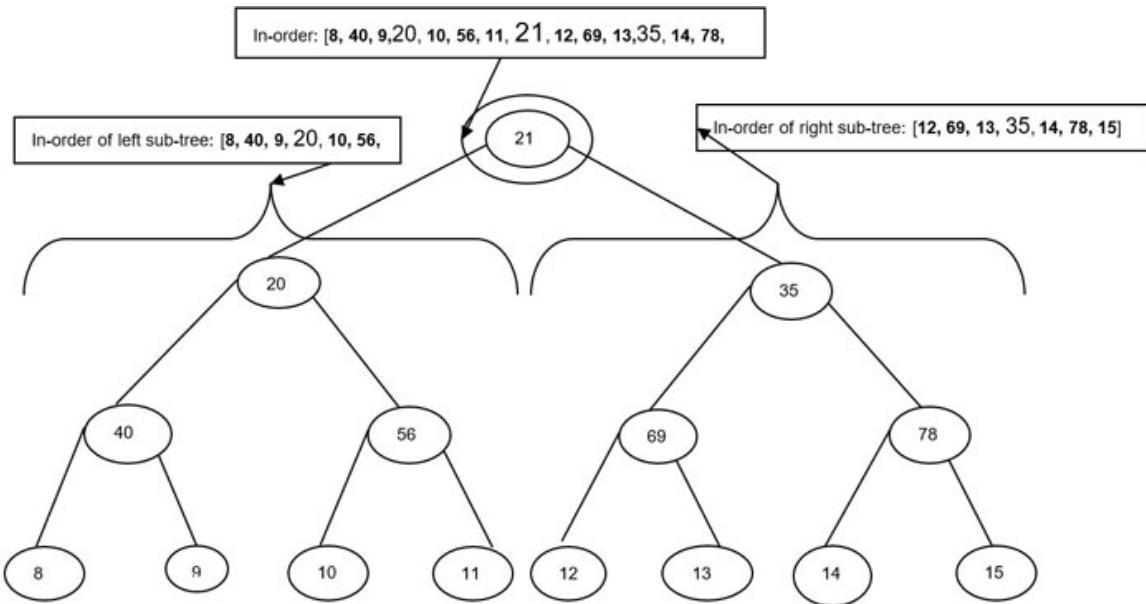
**Figure 8.9:** The In-order traversal of the left sub-tree

The **in-order** traversal of the right sub-tree ([figure 8.10](#)), shown in [figure 8.7](#), prints the in-order traversal of the left tree ([12, 69, 13]), followed by the data at the root (35 in this case), and then the **in-order** traversal of the right sub-tree ([14, 78, 15]). The **in-order** traversal of the right sub-tree, therefore, becomes **[12, 69, 13, 35, 14, 78, 15]**.



**Figure 8.10:** The in-order traversal of the right sub-tree

The **in-order** traversal of the tree, therefore, can be returned by getting the **in-order** traversal of the left tree ([8, 40, 9, 2, 10, 56, 11]), followed by the data at the root (21 in this case), and then the **in-order** traversal of the right sub-tree ([12, 69, 13, 35, 14, 78, 15]). The **in-order** traversal of the right sub-tree, therefore, becomes [8, 40, 9, 20, 10, 56, 11, 21, 12, 69, 13, 35, 14, 78, 15] ([figure 8.11](#)).



**Figure 8.11:** The in-order traversal of the tree shown in [figure 8.7](#)

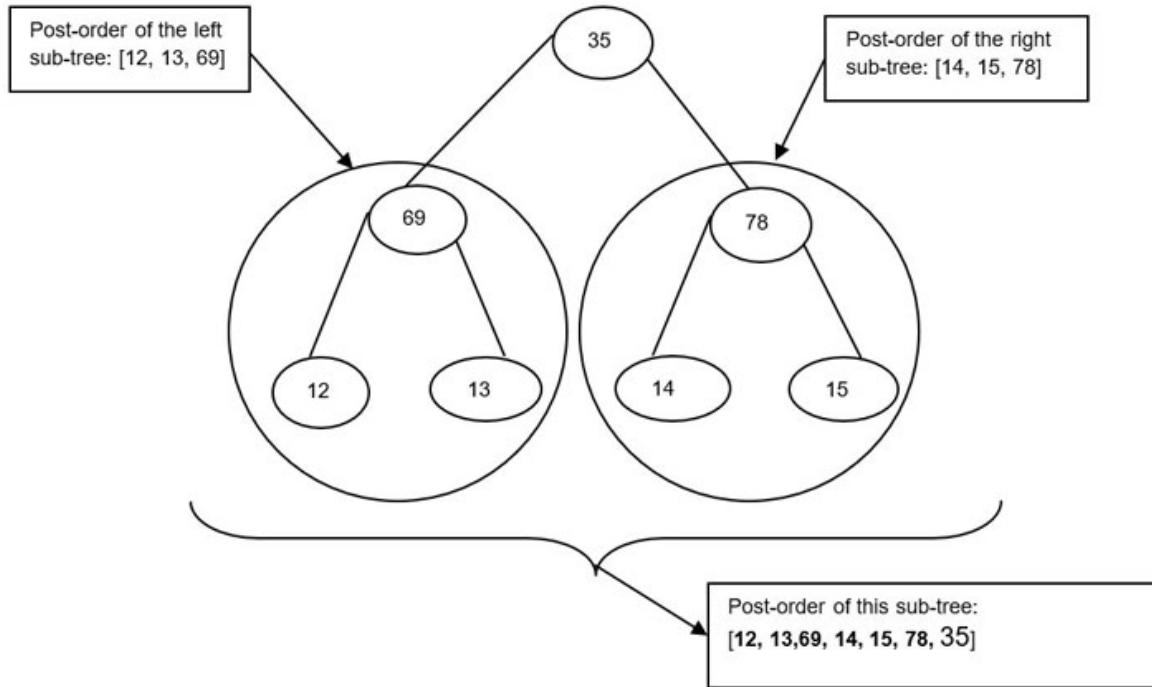
## Post-order traversal

The **post-order** traversal traverses the left sub-tree in **post-order**, followed by the **post-order** traversal of the right sub-tree, then processes the data at the root. The following recursive procedure can be used to print the **post-order** traversal of a given tree.

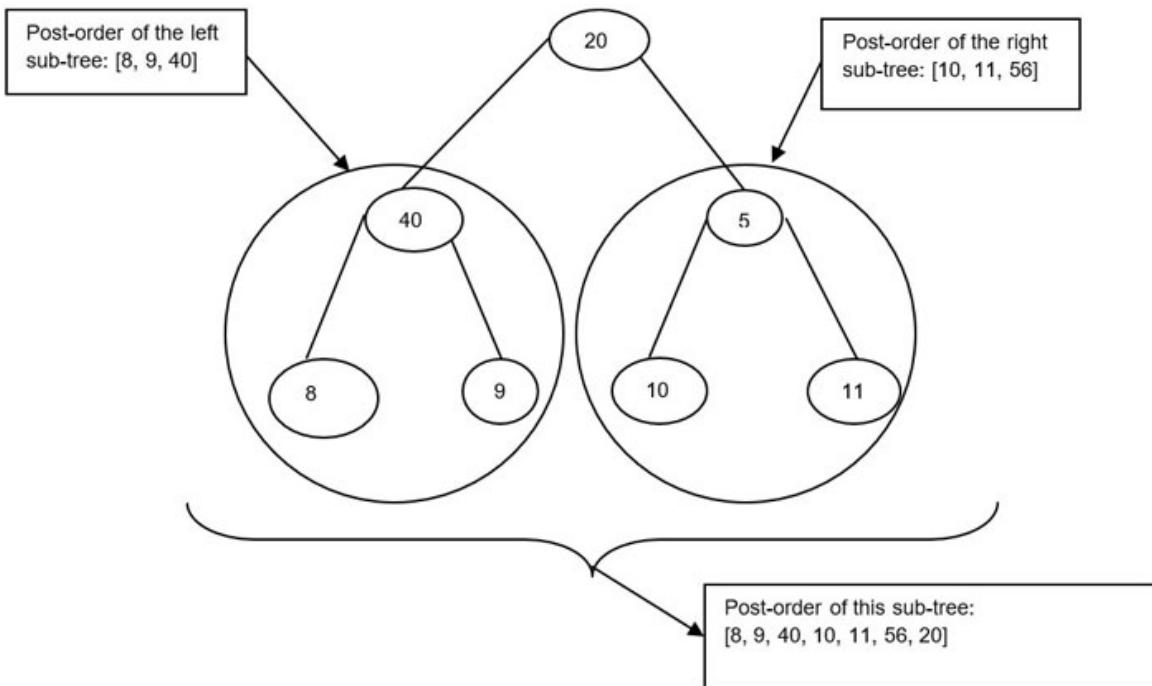
```

def postorder(T):
    if (T == None) :
        return
    elif ((T.left==None ) and (T.right ==None)):
        print(T.data)
    else:
        postorder(T.left)
        postorder(T.right)
        print(T.data)
  
```

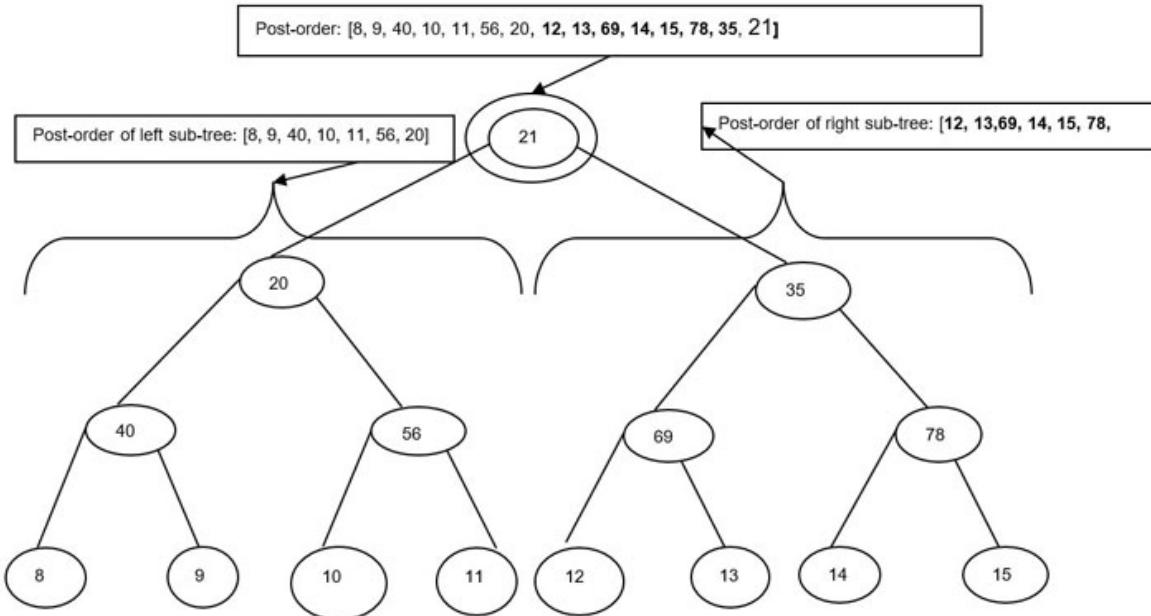
The following figures show the **post-order** traversal of the Binary Tree shown in [figure 8.7](#). [Figure 8.12](#) shows the **post-order** traversal of the left sub-tree. [Figure 8.13](#) shows the **post-order** traversal of the right sub-tree, and finally, [figure 8.14](#) shows the **post-order** traversal of the complete tree.



*Figure 8.12: The post-order traversal of the right sub-tree*



**Figure 8.13:** The post-order traversal of the left sub-tree



**Figure 8.14:** The post-order traversal of the tree shown in [figure 8.7](#)

## Pre-order traversal

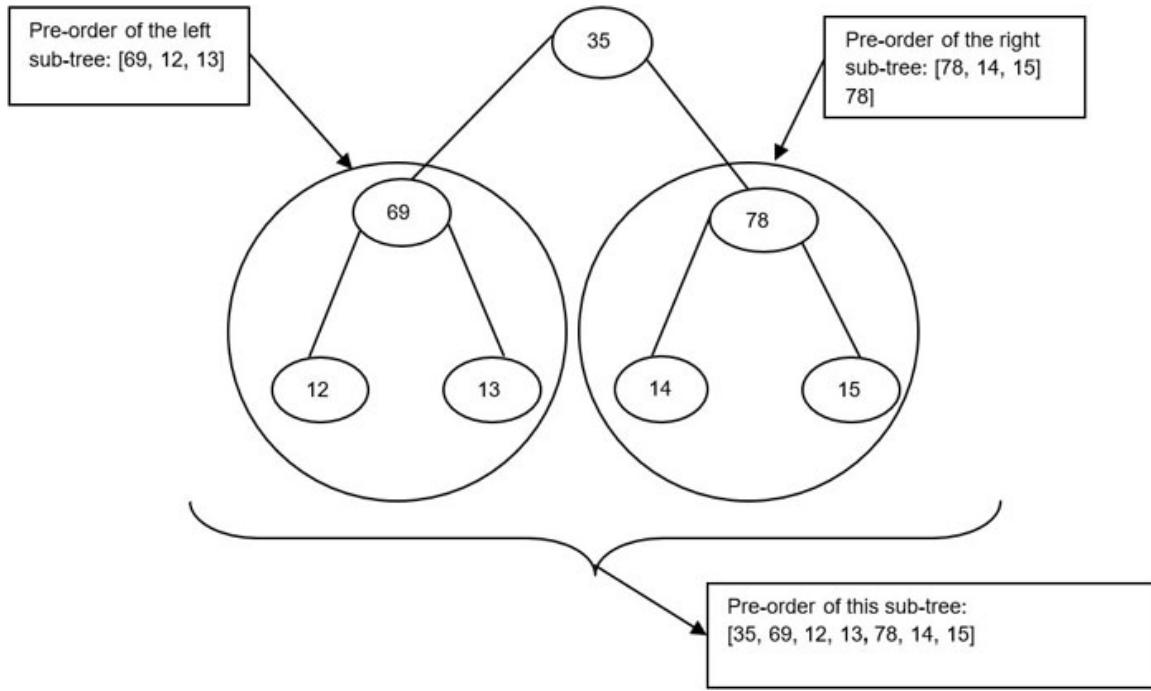
The **pre-order** traversal outputs the data at the root, followed by processing the left sub-tree in **pre-order** and finally the right sub-tree in **pre-order**. The following recursive procedure can be used to print the **pre-order** traversal of a given tree.

```

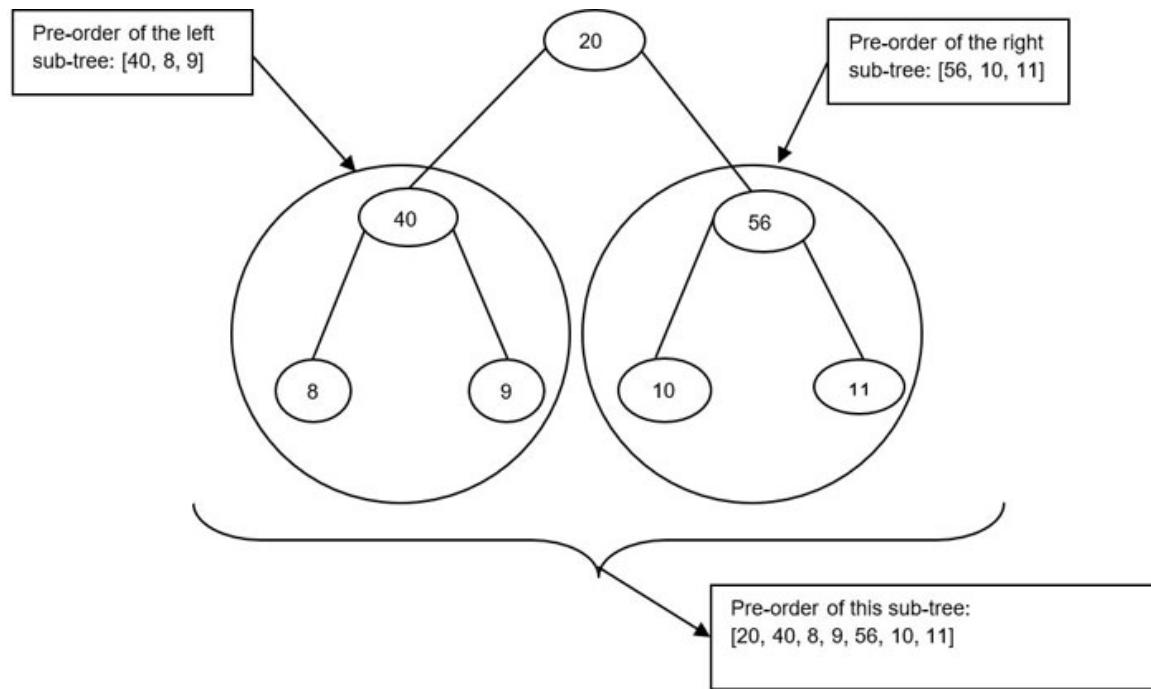
def preorder(T):
    if (T == None) :
        return
    elif ((T.left==None ) and (T.right ==None)):
        print(T.data)
    else:
        print(T.data)
        preorder(T.left)
        preorder(T.right)

```

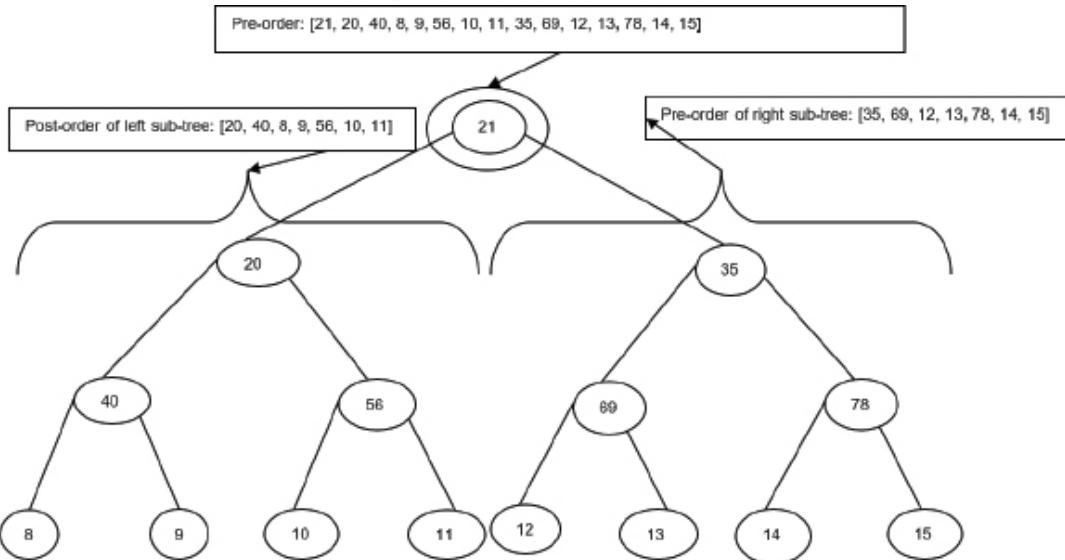
[Figure 8.15](#) shows the pre-order traversal of the BST shown in [figure 8.7](#):



*Figure 8.15: The pre-order traversal of the right sub-tree shown in [figure 8.7](#)*



*Figure 8.16: The pre-order traversal of the left sub-tree shown in [figure 8.7](#)*



**Figure 8.17:** The pre-order traversal of the tree shown in [figure 8.7](#)

The non-recursive procedures of the preceding algorithms use the Stack data structures and have been discussed in the upcoming chapter. Let us now move to a subset of Binary Trees called Binary Search Trees.

## Binary search tree

A **Binary Search Tree (BST)** is a binary tree in which the data of all the nodes in the left sub-tree of a given node are less than that of the node, and the data of all nodes to the right sub-tree are greater than that of the node.

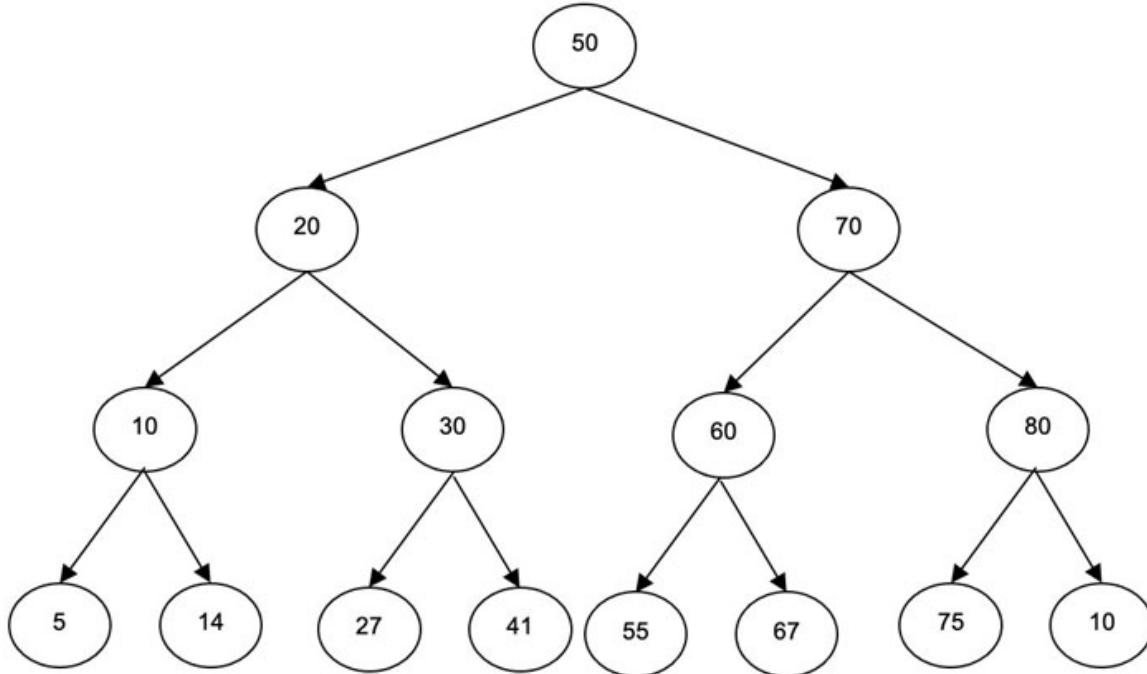
That is if a is a node and b is a node in the left sub-tree of that node (), then

$$b.\text{data} < a.\text{data}$$

On the other hand, if b is a node in the right sub-tree of that node (), then

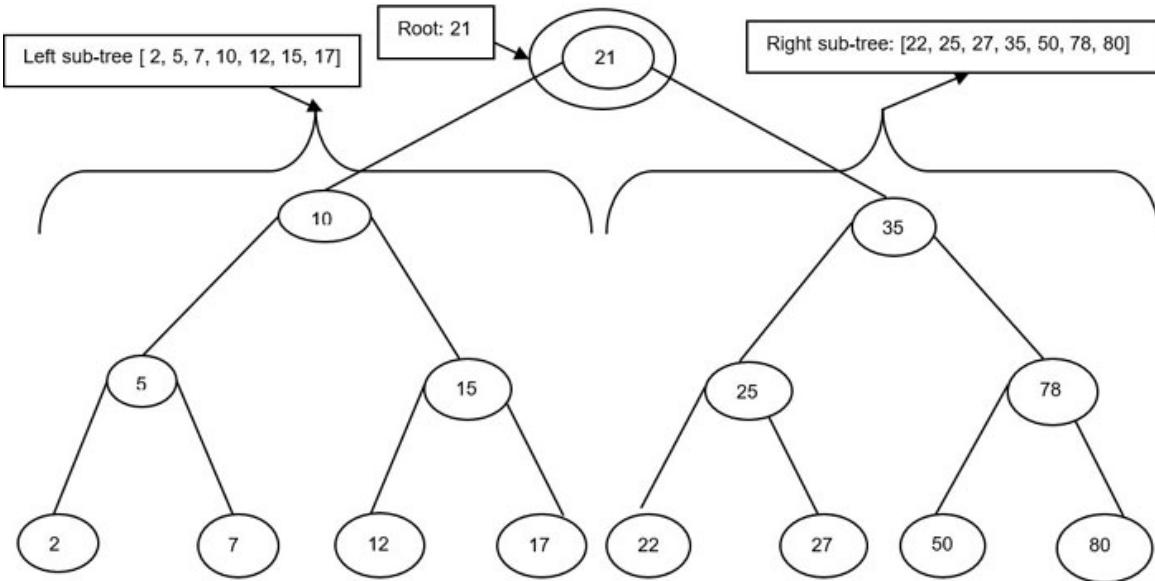
$$b.\text{data} > a.\text{data}$$

The tree shown in [figure 8.18](#) is a BST.



**Figure 8.18:** Binary Search Tree

Consider the in-order traversal of a Binary Search Tree shown in [figure 8.18](#). Note that this traversal ([ 2, 5, 7, 10, 12, 15, 17, 21, 22, 25, 27, 35, 50, 78, 80] ) gives the keys of a given tree in sorted order. Please refer to the following figure:



**Figure 8.19:** In-order traversal of a binary search tree generates a sorted sequence

We now come to the insertion of an element in a BST.

## Insertion in a BST

The insertion in a Binary Search Tree is simple. The first value becomes the root. For inserting the rest of the values, the correct position of the key is located. This is followed by inserting the new key at that position. The following example will help you understand the procedure. Let us consider a tree, initially empty. The algorithm for the insertion of a node in a BST is as follows:

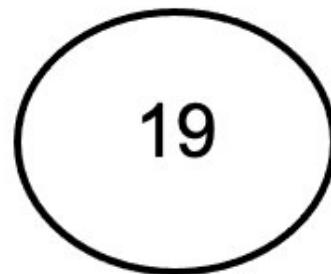
Create a new BST node and assign values to it.

```
def insert(node, key)
If root == None
    Set root = node
    return node
else if (key > root->data)
    insert(root->right, key)
else:
    insert(root->left, key)
```

The following example explains the process of inserting nodes and shows the configuration of the tree in each step.

### 1. Insert (19)

The first value of the tree becomes the data of the root, and the left and the right child of this node (root) are set to None ([figure 8.20](#)):



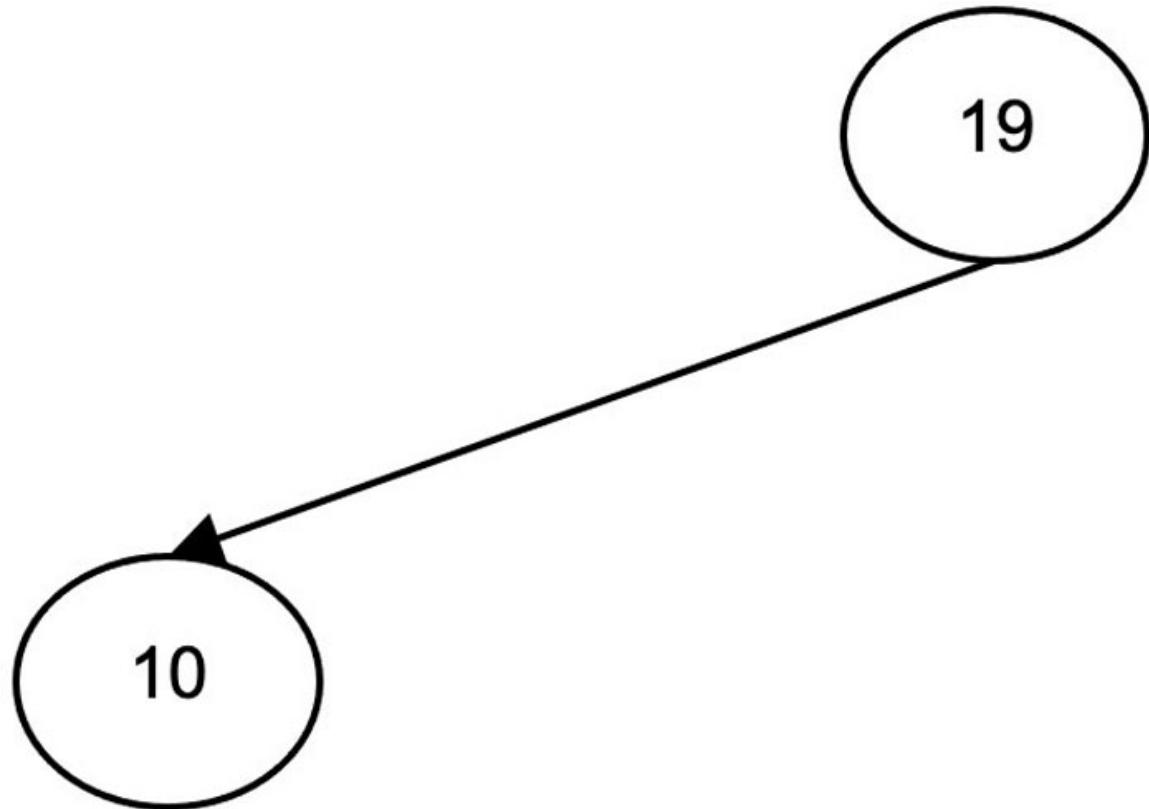
*Figure 8.20: Inserting 19 in a BST*

### 2. Insert (10)

Create a new node called ptr1 and set its value = 10:

```
ptr1= node(10)
```

Set `ptr= root`. Since the value to be inserted (10) < that at `ptr (19)`, so `ptr` becomes `ptr->left`. Since it is None, we set `ptr1 = ptr->left` ([figure 8.21](#)):



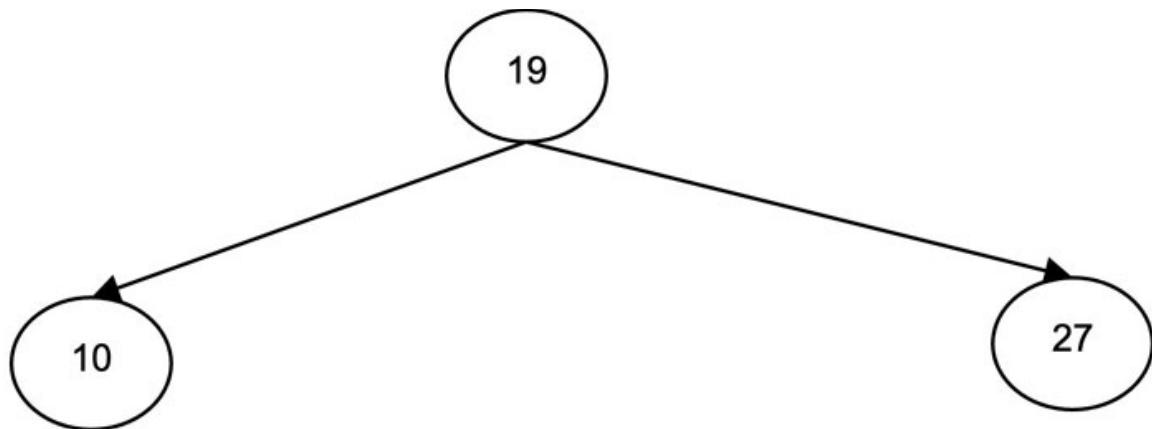
*Figure 8.21: Inserting 10 in the BST*

### 3. Insert (27)

Create a new node called `ptr1` and set its value = 27:

```
ptr1= node(27)
```

Set `ptr= root`. Since the value to be inserted (27) > value at `ptr(10)`, so `ptr` becomes `ptr->right`. Since it is None, we set `ptr1 = ptr->right`.



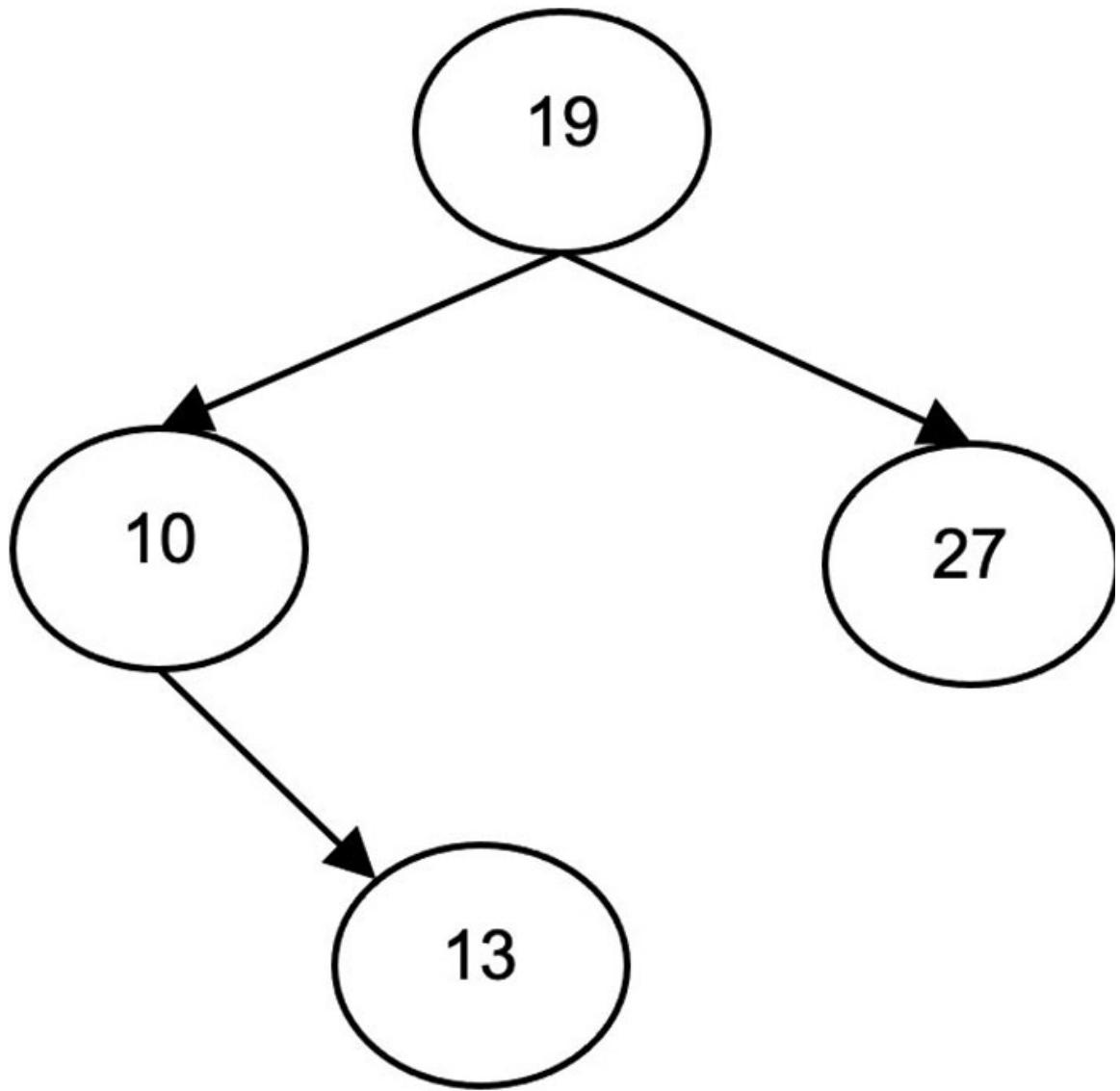
*Figure 8.22: Inserting 27 in the BST*

#### 4. Insert (13)

Create a new node called **ptr1** and set its **value = 13**:

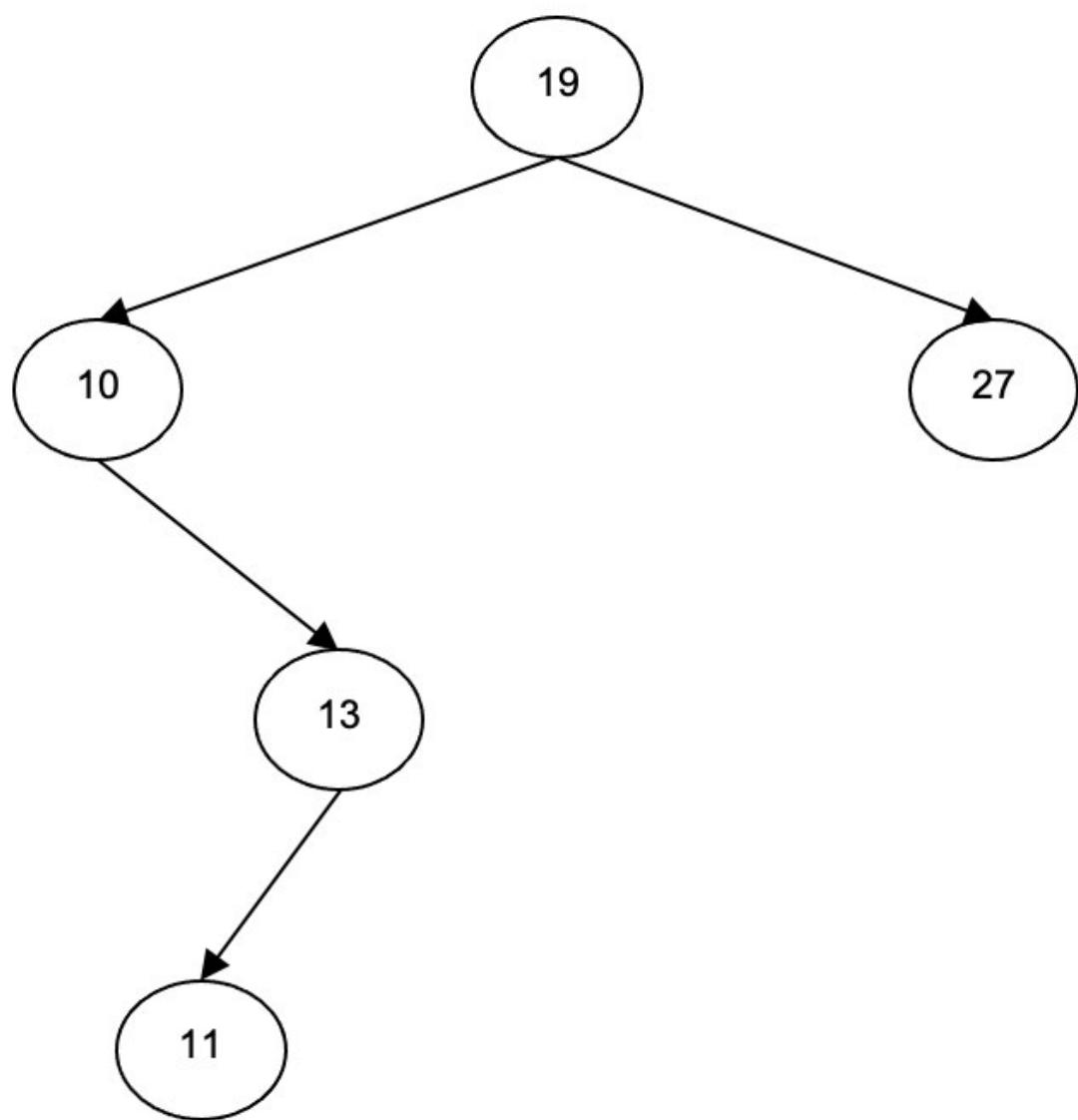
```
ptr1= node(13)
```

Set **ptr= root**. Since the value to be **inserted (13) < value at ptr (19)**, so **ptr** becomes **ptr->left**. Since the value at **ptr** is 10, which is less than 13, so **ptr** becomes **ptr->right**. Now, since **ptr-> right = None**, we set **ptr1 = ptr->right**.

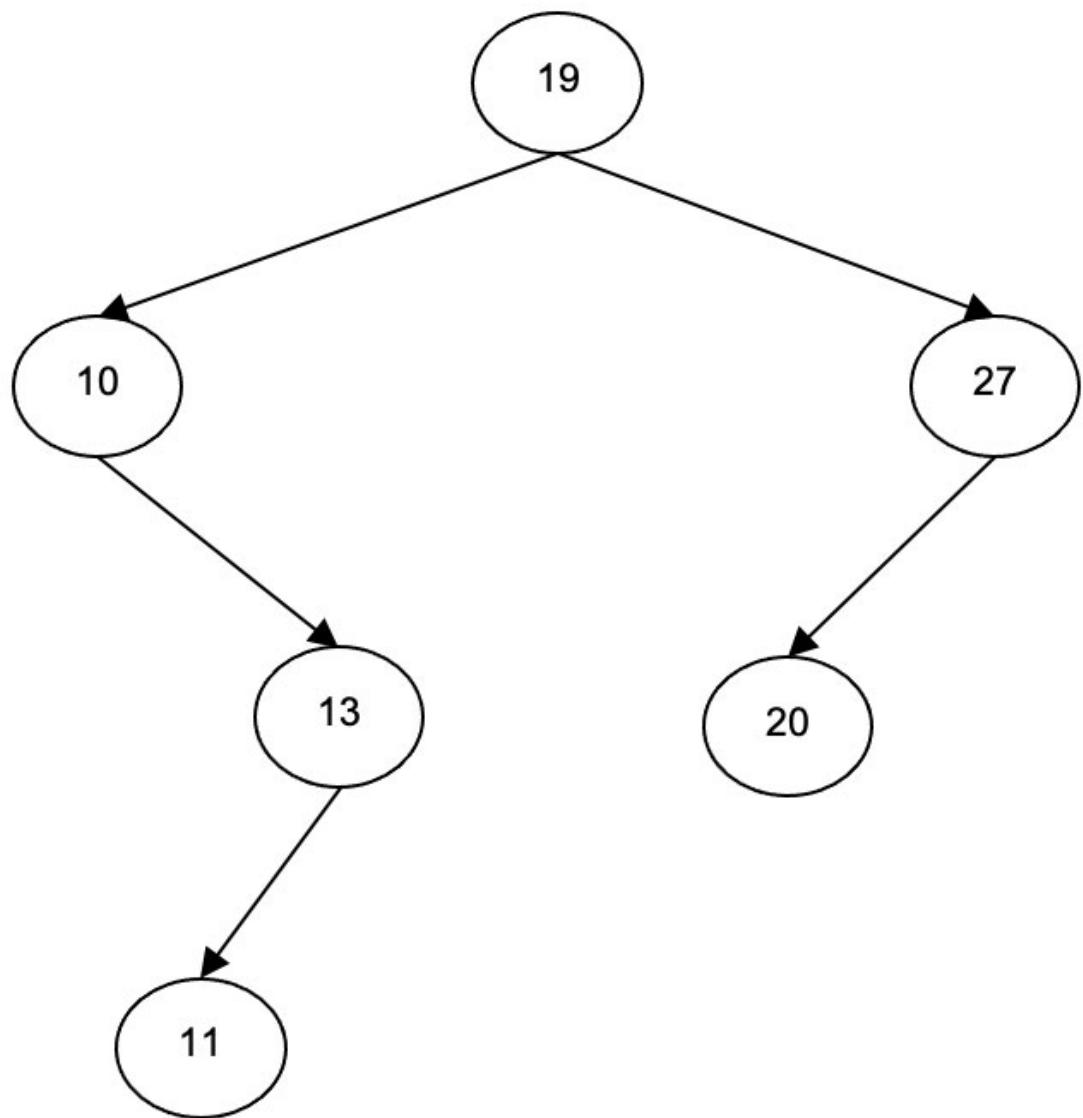


*Figure 8.23: Inserting 13 in the BST*

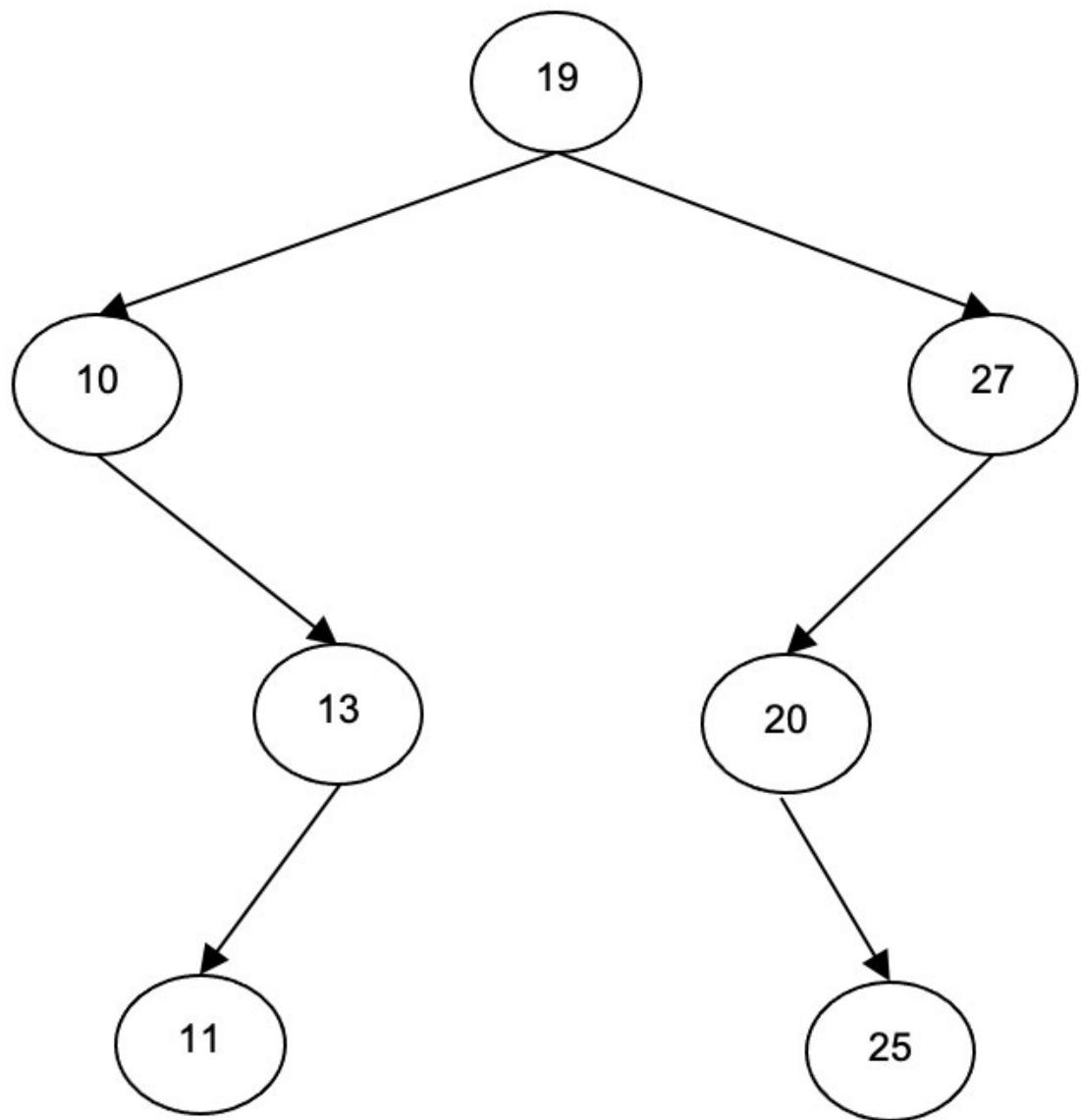
5. The following numbers are then inserted in the preceding tree: 11, 20, 25, 30, 47, and 29. The configurations of the tree after every insertion are shown in [figures 8.24](#) to [8.29](#). Note that at each step, the correct position of the node to be inserted is searched, and the node is then inserted at its correct position.



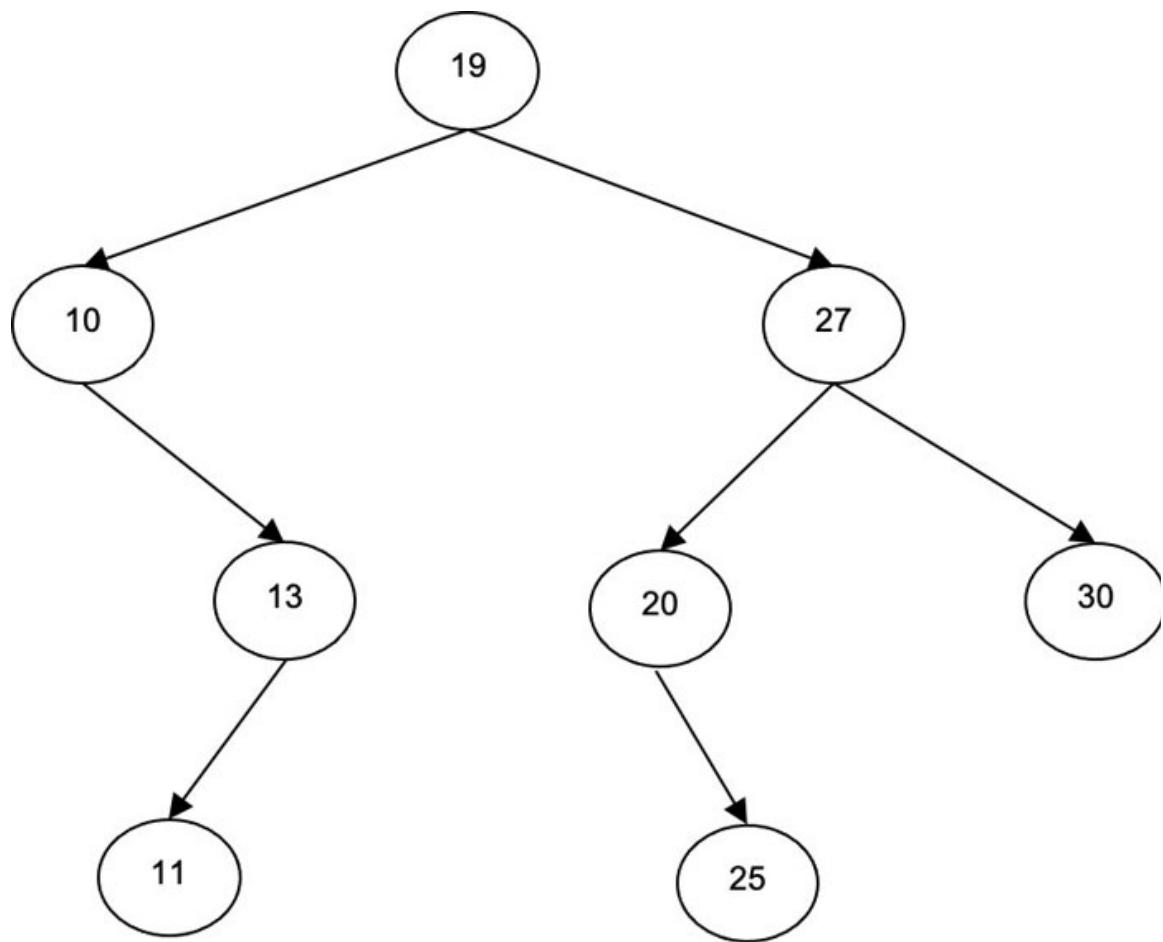
*Figure 8.24: Inserting 11 in the BST*



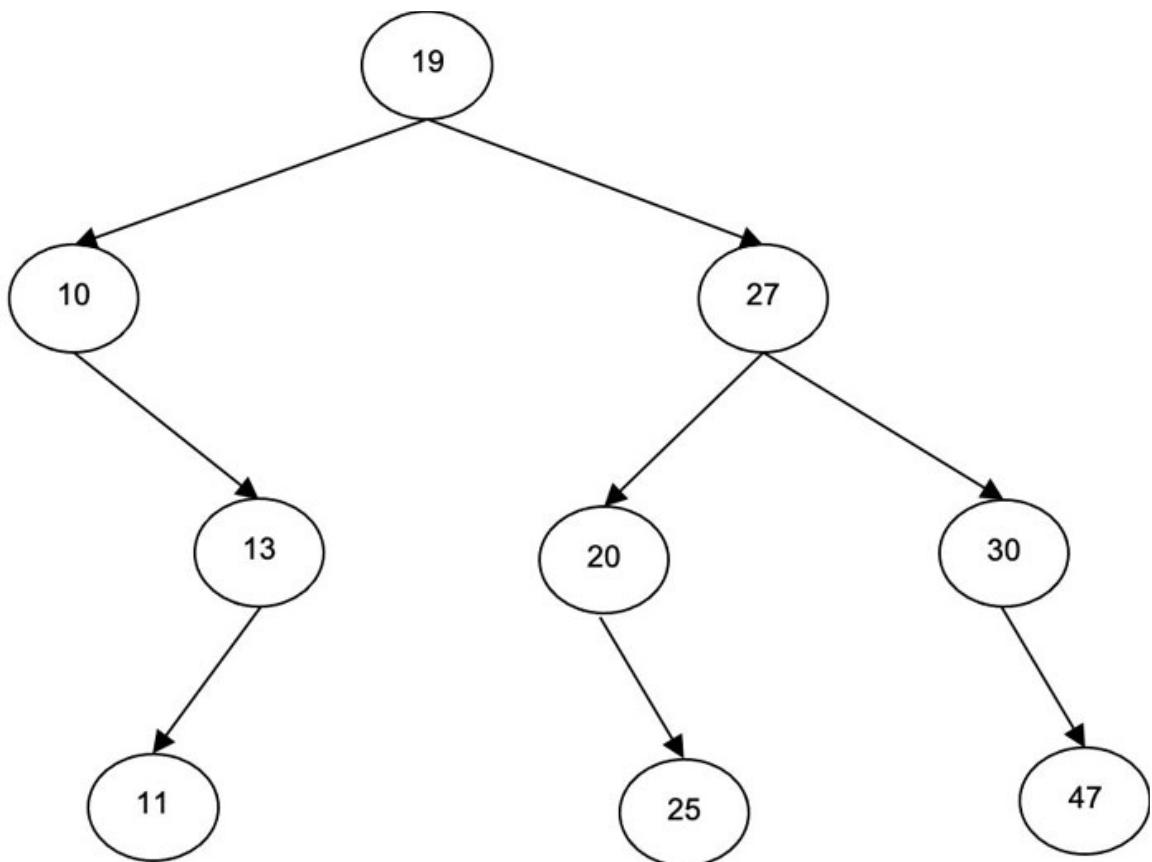
*Figure 8.25: Inserting 20 in the BST*



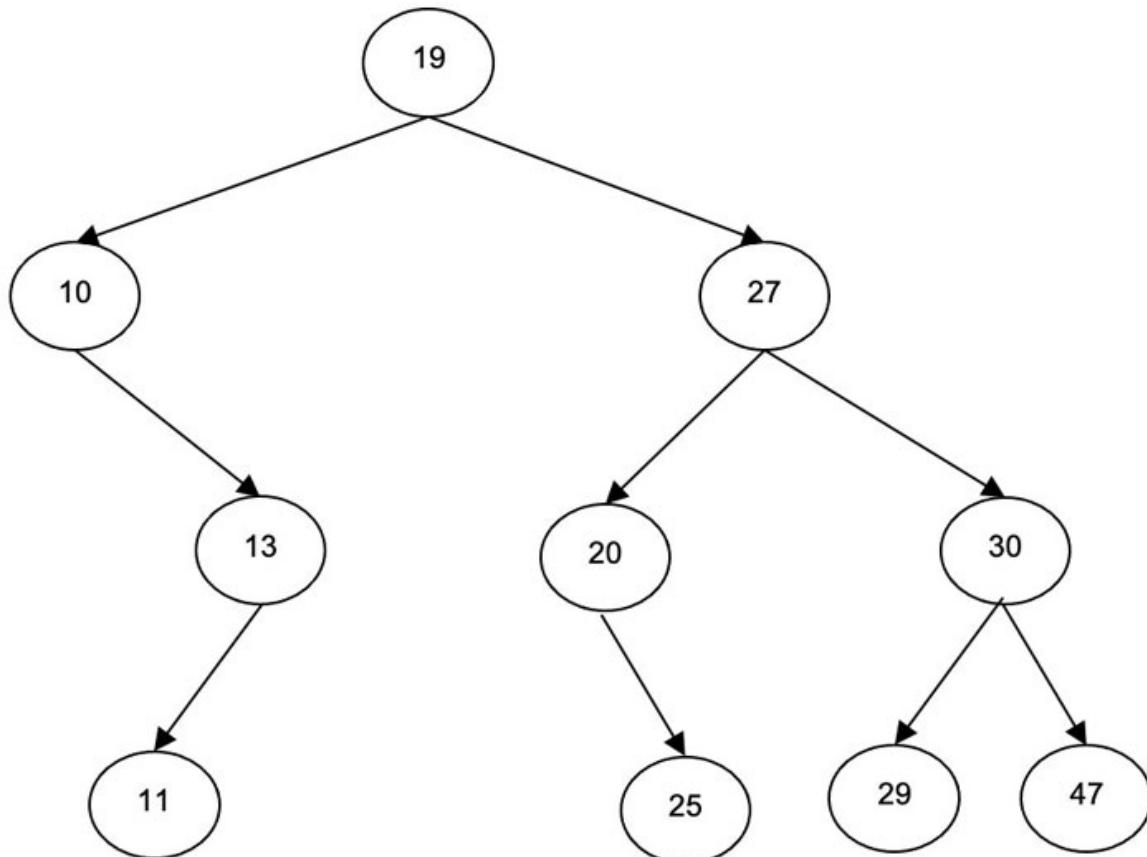
*Figure 8.26: Inserting 25 in the BST*



*Figure 8.27: Inserting 30 in the BST*



*Figure 8.28: Inserting 47 in the BST*



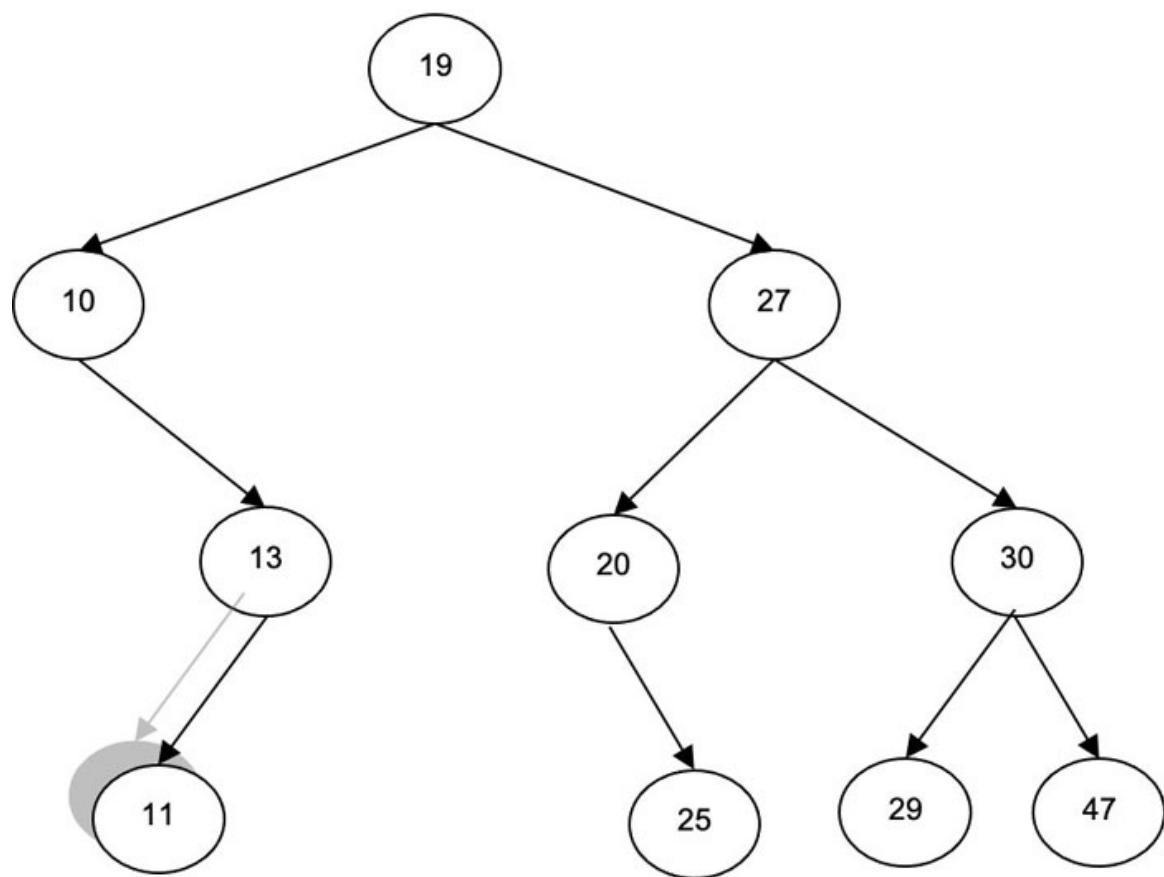
*Figure 8.29: Inserting 29 in the BST*

Having seen insertion in the BST, let us now move to the deletion of a node in the BST.

## Deletion

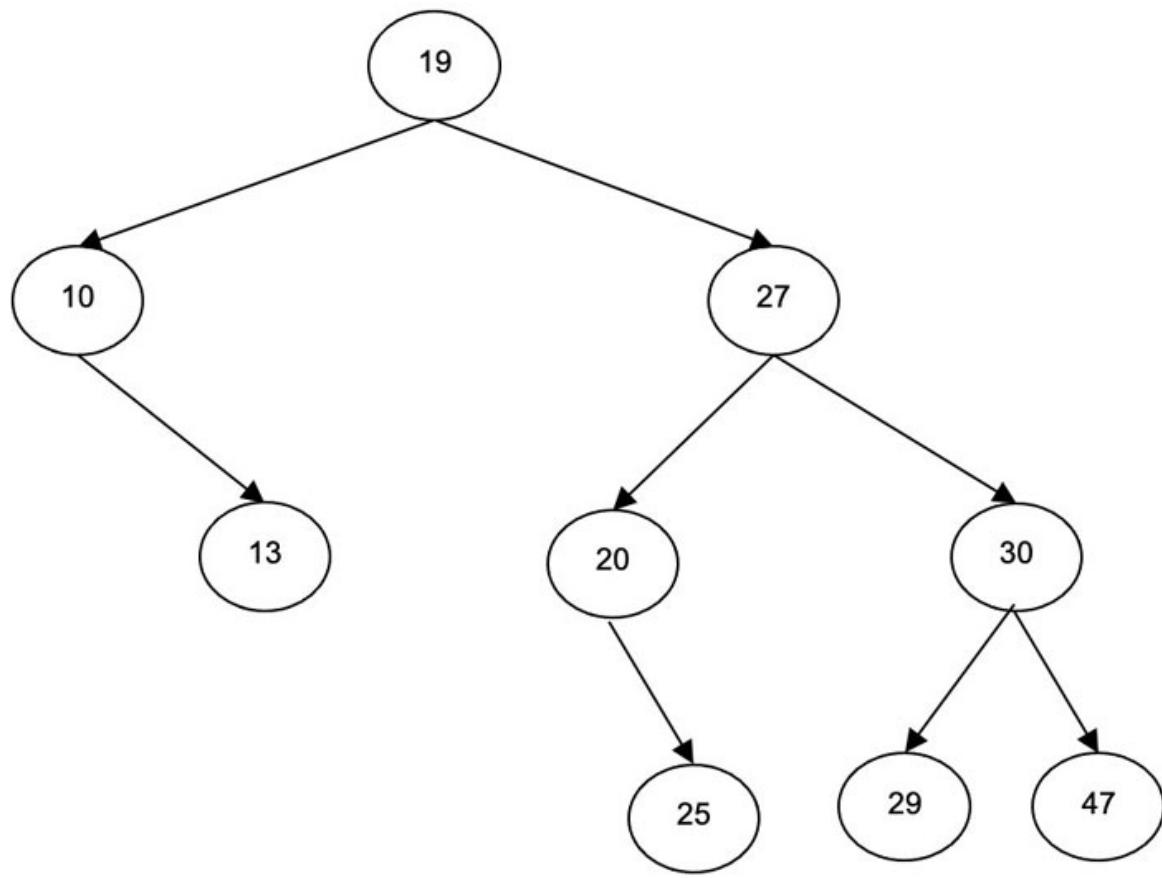
The deletion of a node from a BST is slightly complicated. The deletion of a leaf node is simple. The leaf node is simply deleted (Case 1). In case the node has two children, it may be replaced with the rightmost node of the left sub-tree (or the leftmost node of the right sub-tree)(Case 2). If the child of the node to be deleted has only one child, the parent of the node to be deleted is set to its child (Case 3). The following example depicts the three cases:

### **Case 1: Delete 11**



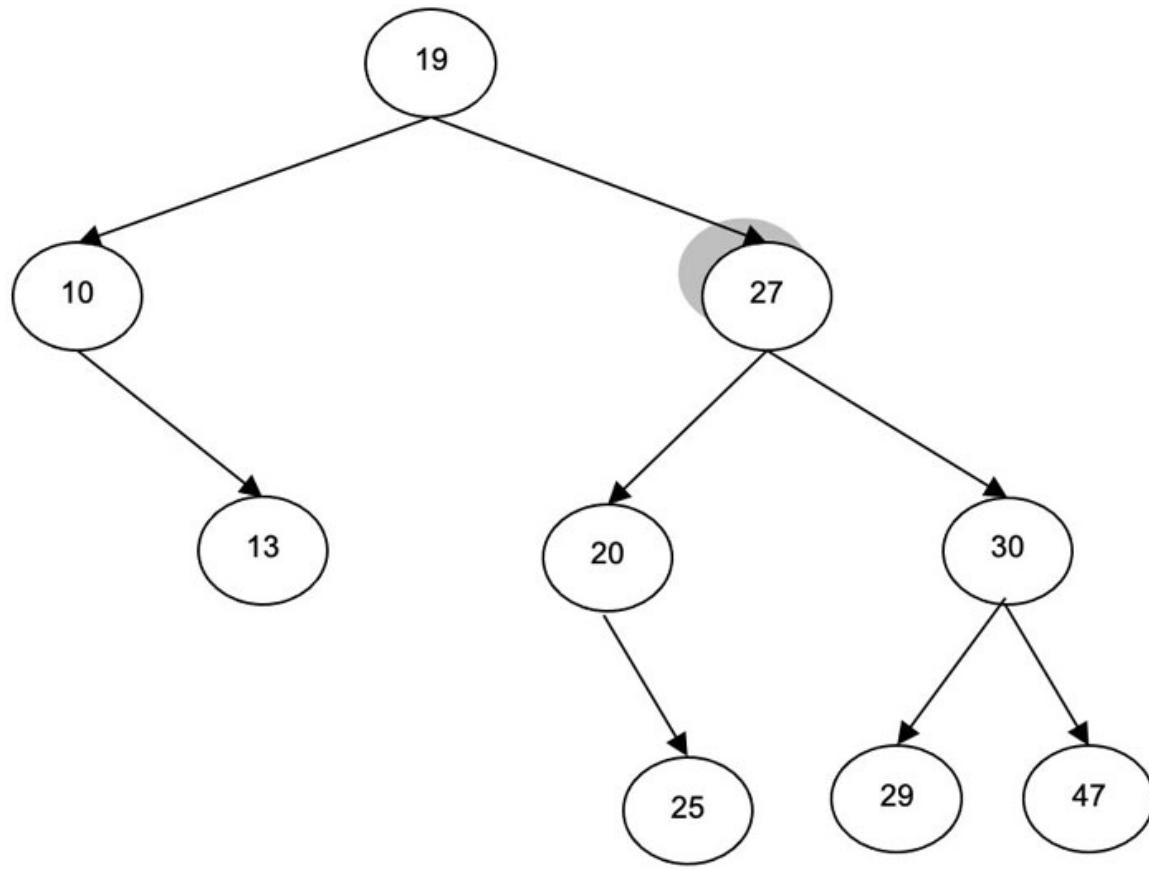
*Figure 8.30: Delete 11 from the BST*

Note that it is a left node, so it is simply deleted, as shown in [figure 8.31](#):



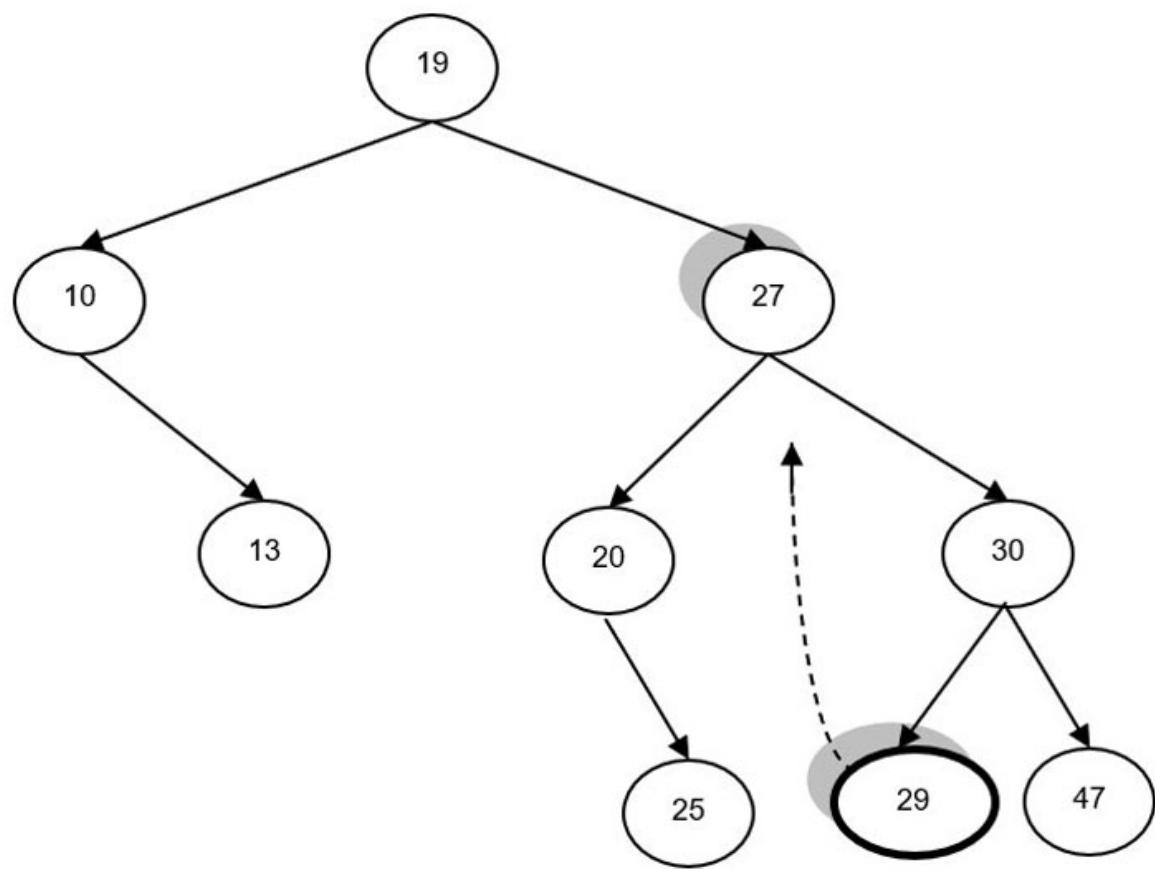
**Figure 8.31:** 11 deleted from the BST shown in [figure 8.30](#)

**Case 2: Delete 27**

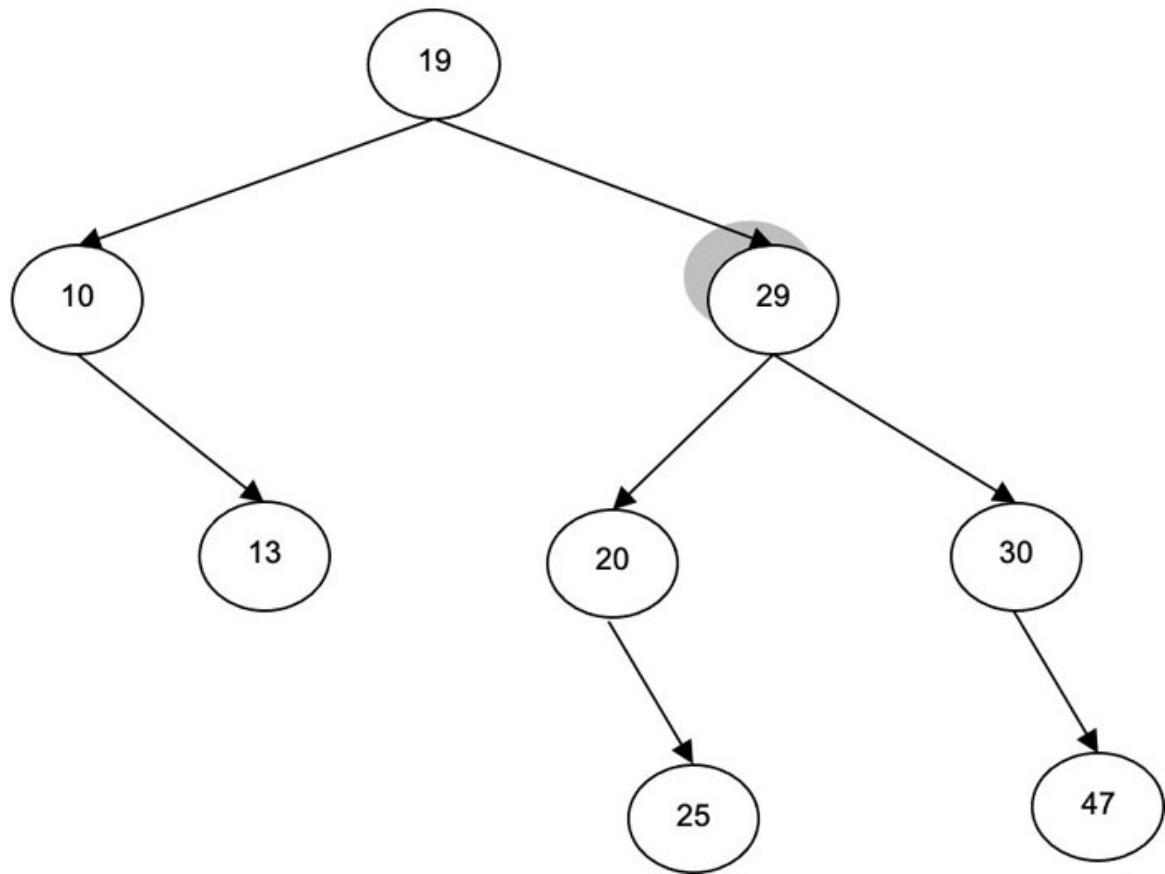


*Figure 8.32: Delete 27 from the BST*

Note that the leftmost node of the right sub-tree has 29, so this left is deleted, and 27 is replaced with 29, as shown in [figure 8.33](#):



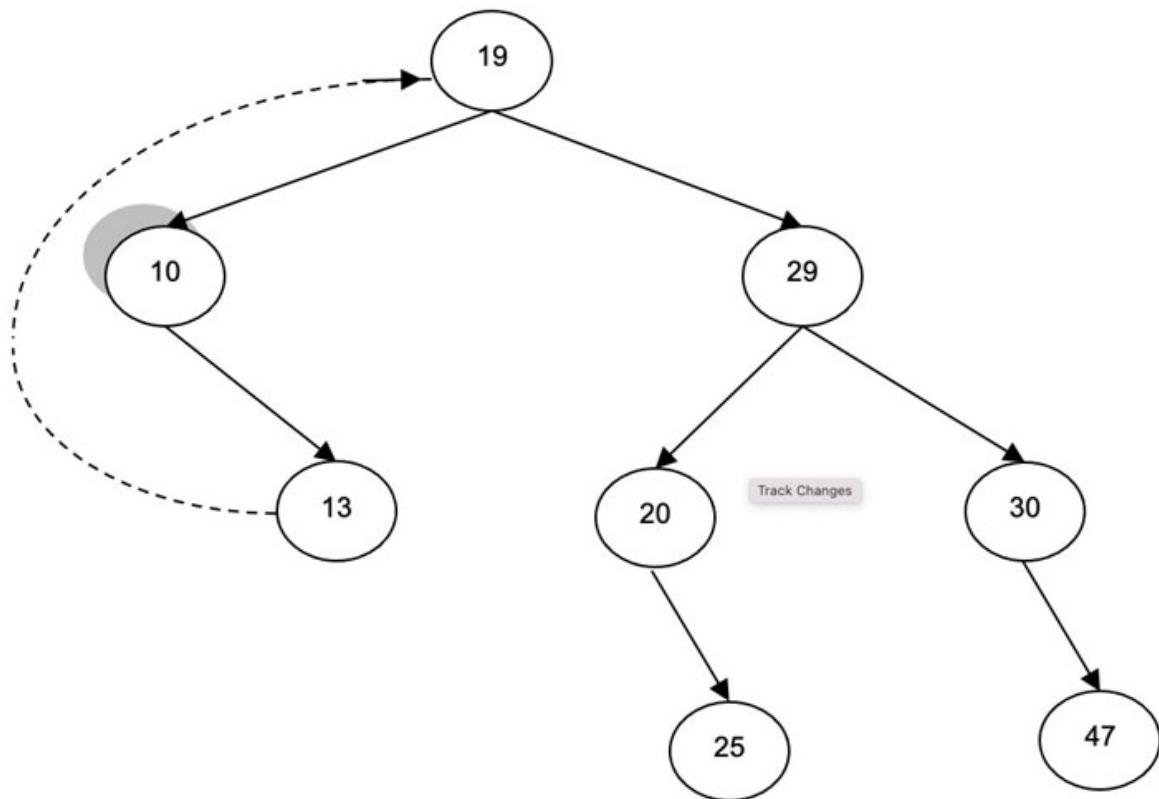
**Figure 8.33:** Replace 27 with 29



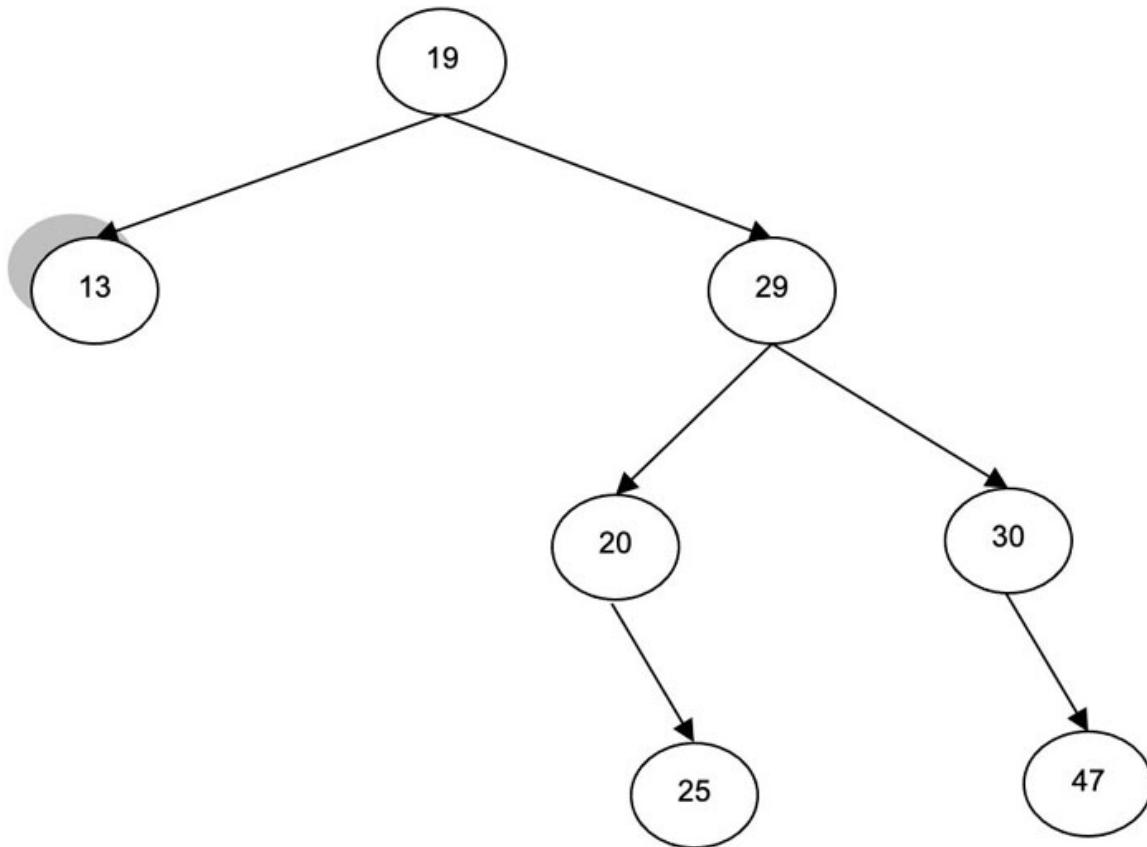
**Figure 8.34:** Delete 29

### Case 3: Delete 10.

Note that the node having 10 has just one child. On deleting 10, the parent (having value 19) starts pointing to its child (having value 13). Please refer to the following figure:



**Figure 8.35:** Delete 10



**Figure 8.36: 10 Deleted**

The following discussion presents the algorithms to find the leftmost node and rightmost node.

### Leftmost node

The leftmost node of a BST is the node having the minimum value. It can be found by setting ptr to the root and changing ptr to `ptr-> left` till `ptr-> left` becomes **NONE**.

The algorithm for finding the leftmost node is as follows:

```

def leftmost(root):
    ptr= root
    while(ptr->left != None):
        ptr= ptr->left
  
```

### Rightmost node

The rightmost node of a BST is the node having the maximum value. It can be found by setting ptr to the root and changing ptr to `ptr-> right` till `ptr-> right`

becomes **NONE**.

The algorithm for finding the leftmost node is as follows:

```
def rightmost(root):
    ptr= root
    while(ptr->right != None):
        ptr= ptr->right
```

The leftmost node of the right sub-tree can be found by setting **ptr=root->right** and passing ptr to the leftmost method created preceding. Likewise, the rightmost node of the left sub-tree can be found by setting **ptr=root->left** and passing **ptr** to the rightmost method created previously.

The following code implements a BST:

## Code

```
1. class node:
2.     def __init__(self, data):
3.         self.left=None
4.         self.right=None
5.         self.data=data
6.     def getLeft(self):
7.         return self.left
8.     def getRight(self):
9.         return self.right
10.    def getData(self):
11.        return self.data
12.    def setData(val):
13.        self.data=val
14. class BinaryTree2:
15.     def __init__(self, rootData):
16.         self.root=node(rootData)
17.         self.level=0
18.     def preOrderTraversal(self, root, trav):
19.         if root == None:
```

```
20.     return
21. else:
22.     stack_pre=[]
23.     stack_pre.append(root)
24.     while stack_pre:
25.         ptr=stack_pre.pop()
26.         trav.append(ptr.data)
27.         if(ptr):
28.             if ptr.right != None:
29.                 stack_pre.append(ptr.right)
30.             if ptr.left !=None:
31.                 stack_pre.append(ptr.left)
32. def inOrderTraversal(self, root, trav):
33.     if root == None:
34.         return
35.     else:
36.         stack_in=[]
37.         ptr=root
38.         while stack_in or ptr:
39.             if ptr != None:
40.                 stack_in.append(ptr)
41.                 ptr=ptr.left
42.             else:
43.                 ptr=stack_in.pop()
44.                 trav.append(ptr.data)
45.                 ptr=ptr.right
46. def postOrderTraversal(self, root, trav):
47.     if root == None:
48.         return
49.     else:
```

```
50.     stack_post=[]
51.     ptr=root
52.     visited=set()
53.     while stack_post or ptr:
54.         if(ptr):
55.             stack_post.append(ptr)
56.             ptr=ptr.left
57.         else:
58.             ptr=stack_post.pop()
59.             if(ptr.right != None) and (ptr.right not in visited):
60.                 stack_post.append(ptr)
61.                 ptr=ptr.right
62.             else:
63.                 visited.add(ptr)
64.                 trav.append(ptr.data)
65.                 ptr=None
66.
67.     def insertLeft(self, val):
68.         ptr=self.root
69.         if ptr.left==None:
70.             ptr.left=node(val)
71.         else:
72.             temp=node(val)
73.             temp.left=ptr.left
74.             ptr.left=temp
75.     def insertRight(self, val):
76.         ptr=self.root
77.         if ptr.right==None:
78.             ptr.right=node(val)
79.         else:
```

```
80.     temp=node(val)
81.     temp.right=ptr.right
82.     ptr.right=temp
83. Tree1=BinaryTree2(45)
84. inOrder=[]
85. Tree1.inOrderTraversal(Tree1.root, inOrder)
86. print(inOrder)
87. Tree1.insertLeft(5)
88. Tree1.insertLeft(7)
89. Tree1.insertRight(90)
90. Tree1.insertRight(67)
91. Tree1.insertLeft(56)
92. inOrder=[]
93. Tree1.inOrderTraversal(Tree1.root, inOrder)
94. print(inOrder)
95. preOrder=[]
96. Tree1.preOrderTraversal(Tree1.root, preOrder)
97. print(preOrder)
98. postOrder=[]
99. Tree1.postOrderTraversal(Tree1.root, postOrder)
100. print(postOrder)
```

### **Output:**

```
[45]
[5, 7, 56, 45, 67, 90]
[45, 56, 7, 5, 67, 90]
[5, 7, 56, 90, 67, 45]
```

## **Conclusion**

This chapter introduced trees. The definition, terminology, and types of these data structures have been explored in the chapter. The primary focus of the chapter is

Binary Search Trees. The reader will be able to implement the binary trees and BSTs and carry out some basic tasks using these data structures.

The problems related to BST have been discussed in the next chapter as well. The upcoming chapter revisits BST and introduces balanced trees and B Trees. Furthermore, the Huffman algorithm has also been discussed in the upcoming chapter.

## Multiple choice questions

- 1. A tree does not have a**
  - a. Cycle
  - b. Isolated vertices
  - c. Isolated edge
  - d. All of the above
- 2. A binary tree has a depth 8. Each level has only one node. If this tree is represented using an array, what should be the minimum size of the array?**
  - a. 64
  - b. 128
  - c. 8
  - d. None of the above
- 3. In the preceding question, what is the amount of space wasted?**
  - a. 56
  - b. 120
  - c. 0
  - d. None of the above
- 4. In Question 2, if each level is completely filled what is the amount of space wasted?**
  - a. 56
  - b. 120
  - c. 0
  - d. None of the above

**5. In a complete binary tree with eight levels, what will be the number of nodes in the fifth level?**

- a. 15
- b. 31
- c. 63
- d. None of the above

**6. The leftmost node of a BST has the**

- a. Minimum value
- b. Maximum value
- c. None of the above

**7. The right-most node of a BST has the**

- a. Minimum value
- b. Maximum value
- c. None of the above

**8. The in-order traversal of a BST generates**

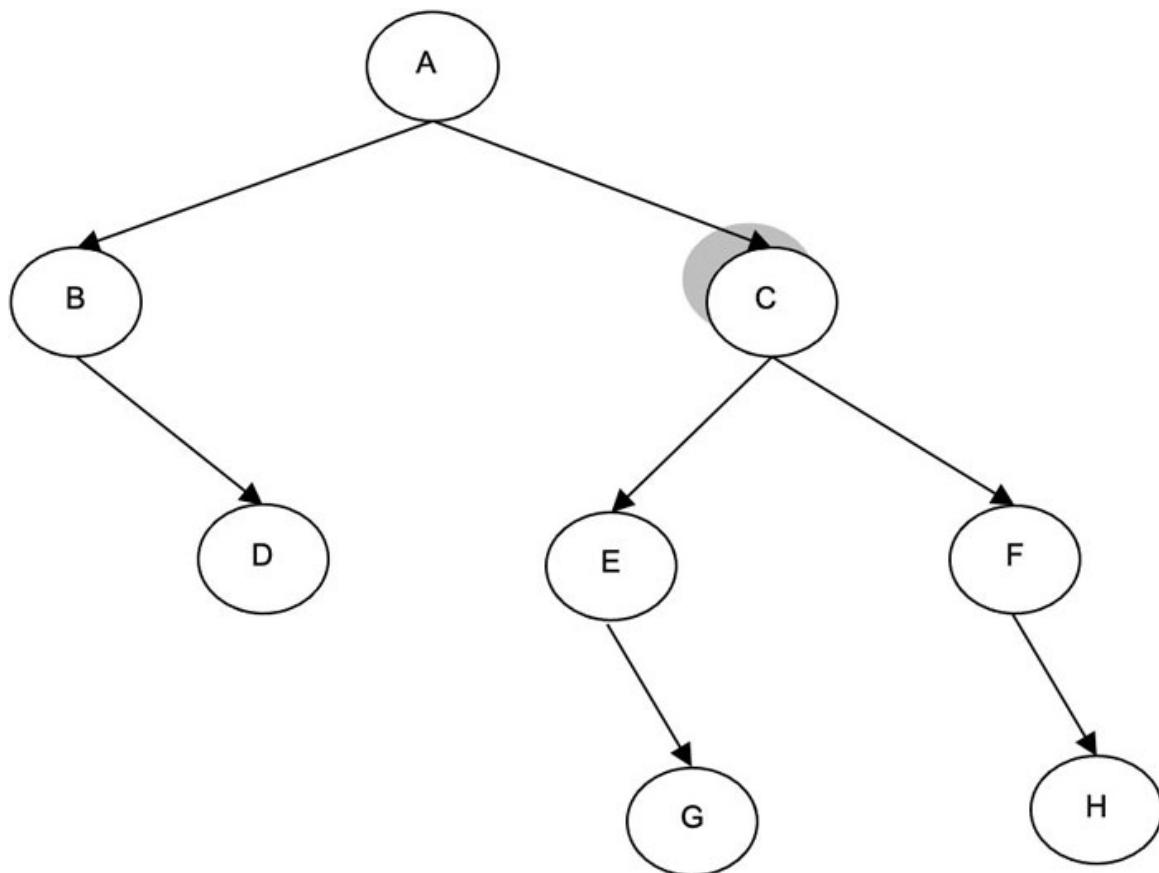
- a. Sorted sequence
- b. Zig-zag sequence
- c. None of the above

**9. A BST has 45 as its root, which of the following cannot be in the left sub-tree?**

- a. 56
- b. 12
- c. 31
- d. 19

### Numerical/problems

Consider the tree shown in [figure 8.37](#), and answer the following questions:



**Figure 8.37: Binary Tree**

1. What is the root of this tree?
2. Name the leaves of this tree.
3. Are B and C siblings?
4. At what Level G is present?
5. What is the depth of this tree?
6. Is it a complete tree?
7. Write the pre-order traversal of this tree.
8. Write the post-order traversal of this tree.
9. Write the in-order traversal of this tree.
10. Using the pre-order and in-order traversal of this tree, create the tree.
11. Using the post-order and in-order traversal of this tree, create the tree.
12. Which is the left-most node of this tree? Which is the right-most?
13. What is the successor of A?

14. Find the total number of nodes required to convert to a Complete Binary Tree.
15. Store the tree in an array.
16. In the preceding question, find the percentage occupancy of the array.
17. Represent the above tree using a linked list.
18. Can you state some disadvantages of representing trees using an array?
19. Can you state some disadvantages of representing trees using a linked list?
20. Can you represent the above tree using a singly linked list?

Consider the following sequence of numbers:

10, 67, 24, 76, 90, 56, 223, 89, 113, 91

1. Create a Binary Search Tree using the above values.
2. Write a procedure to search 90.
3. Write a procedure to find the minimum element from a BST.
4. Write a procedure to find the maximum element from a BST.
5. Traverse the BST in in-order.
6. Insert 08 in the BST.
7. Insert 02 in the BST.
8. Insert 78 in the BST.
9. Delete 20 from the BT.
10. Delete 10 from the BST.
11. Delete 56 from the BST.
12. Can you merge two BSTs?
13. Can you find the predecessor of 76?
14. Can you merge two BSTs?
15. Create all possible BSTs using the above numbers.

## 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>



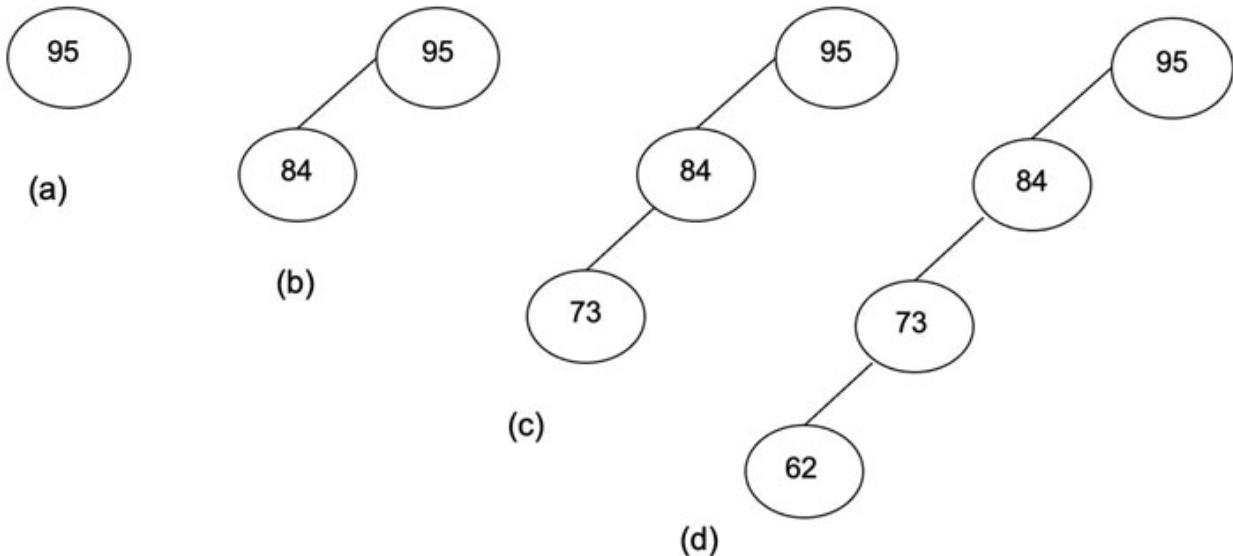
## CHAPTER 9

### Trees-II

In the previous chapter, we discussed the creation of a **Binary Search Tree (BST)**. The algorithms for insertion and deletion in this data structure were also discussed in the chapter. The advantage of these trees was that the average case complexity of search, insertion, and deletion was  $O(\log n)$ . However, at times this data structure loses its charm. Consider inserting the following numbers in a BST:

95, 84, 73, 62

The root of the BST will have 95. Since 84 is less than 95, it will become the left child of 95. Likewise, 73 and 62 will become the left child of 84 and 73, respectively, as shown in [figure 9.1](#):



*Figure 9.1: An example of a skewed BST*

Note that the BST so formed has depth = number of elements. Likewise, the BST formed by inserting the following sequence also has depth = number of elements.

95, 184, 273, 362, 451, 540

You must have inferred that the BST formed by increasing or decreasing sequences is always skewed. Thus, the complexity of the search operation will become  $O(n)$  instead of  $O(\log n)$ . Even for an almost sorted sequence, the average complexity of insertion, deletion, and the search will be  $O(n)$ . In such cases, using BST is of no use, as linear time complexity for searching and insertion is provided even by an array.

To handle this problem, we will explore two data structures, which are variants of binary search trees.

- The first is a balanced tree in which the height is proportional to  $O(\log n)$ . Adelson-Velskii and Landis proposed these balanced trees known as AVL Trees.
- The second is B-trees, which can store more than a single element in a node.

The reader will be able to create their trees, carry out insertion and deletion operations, and consequently handle the problem stated previously.

## Structure

In this chapter, we will cover the following topics:

- Definition and representation of AVL tree
- Insertion in an AVL tree
- Deletion from an AVL tree
- Insertion in a B tree

## Objectives

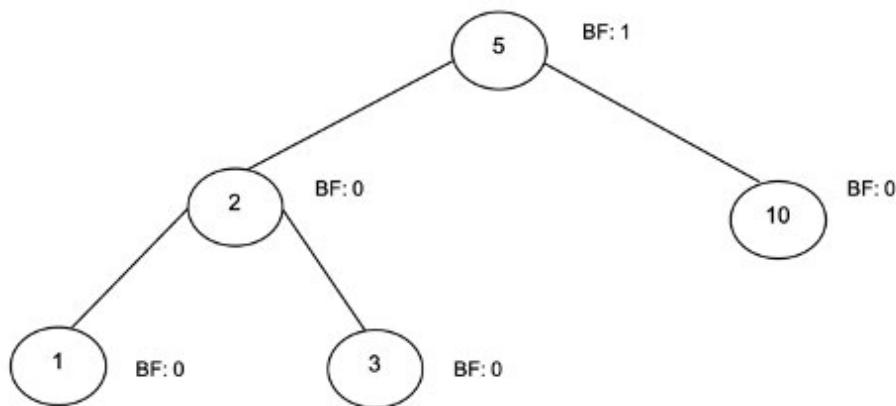
This chapter begins with the AVL trees and the definition of the Balance Factor. This is followed by a brief discussion around the insertion in this data structure and deletion from the AVL trees. The chapter then discusses examples pertaining to the above and then moves to B Tree.

## AVL trees

### *Definition*

- An empty binary tree is an AVL tree
- A non-empty tree where the difference between the heights of the left sub-tree and the right sub-tree for each node is maximum 1 is an AVL tree.

For a particular node, the difference between the height of the left and the right sub-tree is called the **Balance Factor (BF)**. In an AVL tree, the balance factor of any node can be 0, 1, or -1. For example, the tree shown in [figure 9.2](#) is an AVL tree:



*Figure 9.2: An example of a balanced tree*

Let us have a brief overview of insertion and deletion in AVL trees.

## Insertion

If insertion in a BST results in an unbalanced tree, to make it balanced again, one of the following operations is carried out. Note that in the following discussion, A is the root of the sub-tree, which becomes unbalanced.

### **LL**

If the new element is inserted in the left subtree of the left sub-tree of the A, then the balance factor of the closest ancestor becomes 2.

### **RR**

If the new element is inserted in the right sub-tree of the right sub-tree of the A, then the balance factor of the closest ancestor becomes -2.

### **RL**

If the new element is inserted in the right sub-tree of the left sub-tree of the A, then the tree is no longer balanced.

## LR

If the new element is inserted in the left sub-tree of the right sub-tree of the A, then the tree is no longer balanced.

To restore the balance of each of these mentioned elements, the resultant tree can be rotated, as explained in the following section.

## Deletion

Deletion from an AVL tree can be done by identifying the closest ancestor whose BF becomes +2 or -2; this is followed by identifying whether deletion is from the left sub-tree or from the right sub-tree. The last step is to see the root of the left of the right sub-tree and identify whether the BF is {0, 1, -1}.

This results in 6 cases, namely:

- L0
- L1
- L-1
- R0
- R1
- R-1

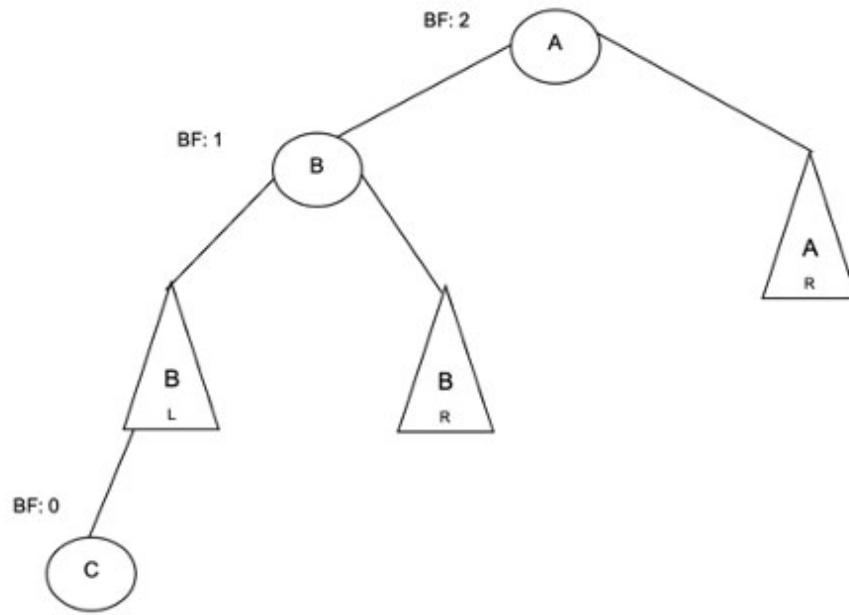
The following section explains each of the preceding operations in detail.

## Insertion in an AVL tree

In inserting a node in an AVL tree, it is first inserted as in the case of a BST, and the **Balance Factor (BF)** of each node is then calculated. If the balance factor of any node is not 0, 1, or -1, either LL, LR, RL, or RR rotation is carried out. Let us have a look at each of these cases.

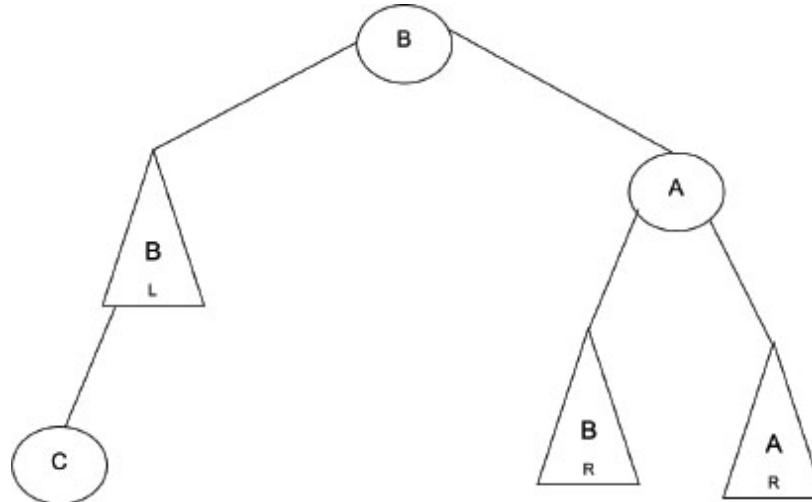
In the LL rotation, if the tree is initially balanced and a node is added to the left of the left sub-tree of the root (LL) ([figure 9.3](#)), which results in making

the BF of the root 2, the resultant tree is no longer a balanced tree. The balancing is carried out as follows:



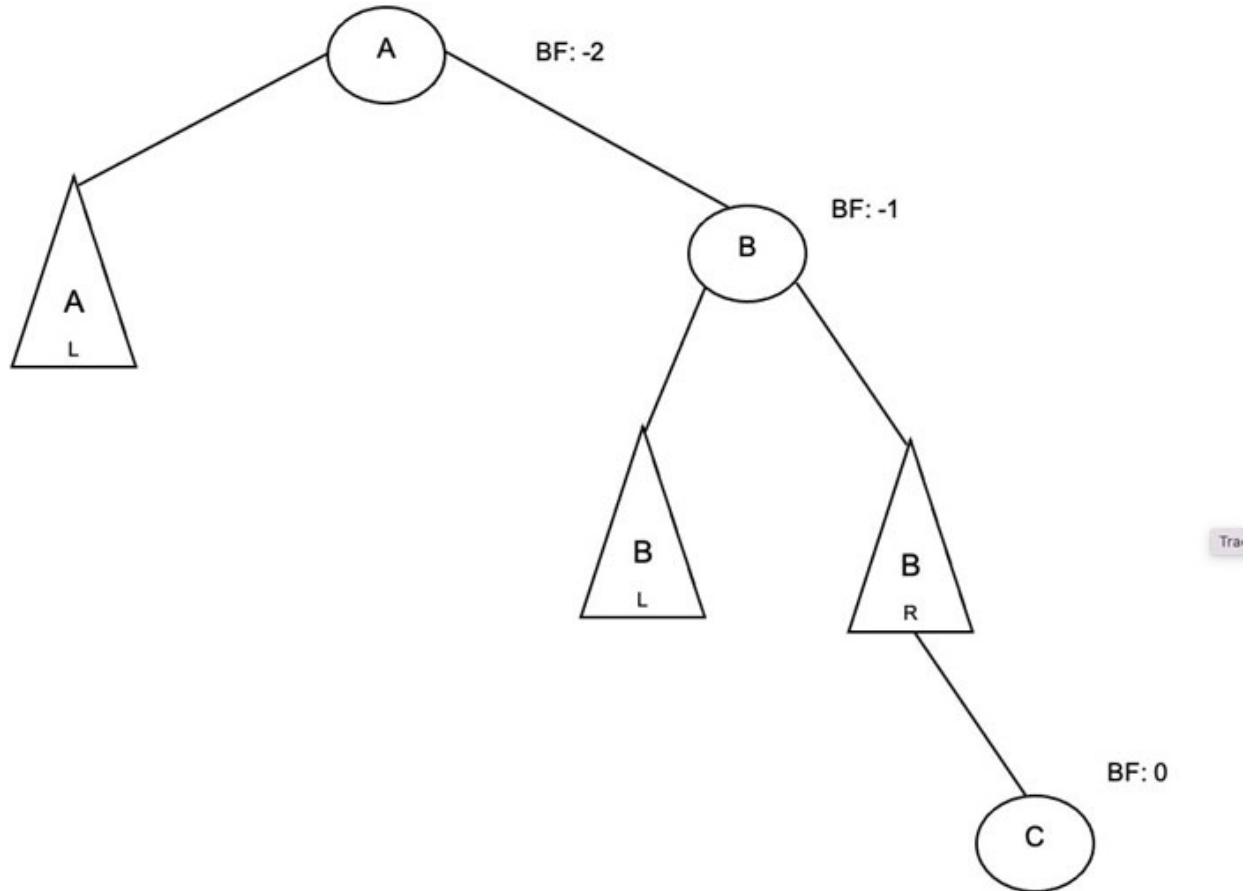
**Figure 9.3: LL: Inserting node in the left of the left sub-tree in a balanced tree**

To convert the tree shown in [figure 9.3](#) to a balanced tree, B is made the root,  $B_L$  remains the left sub-tree of B, C remains as it is, A becomes the right node of B, and  $A_R$  remains the right sub-tree of A. Finally,  $B_R$  becomes the left sub-tree of A. The resultant tree is shown in [figure 9.4](#).



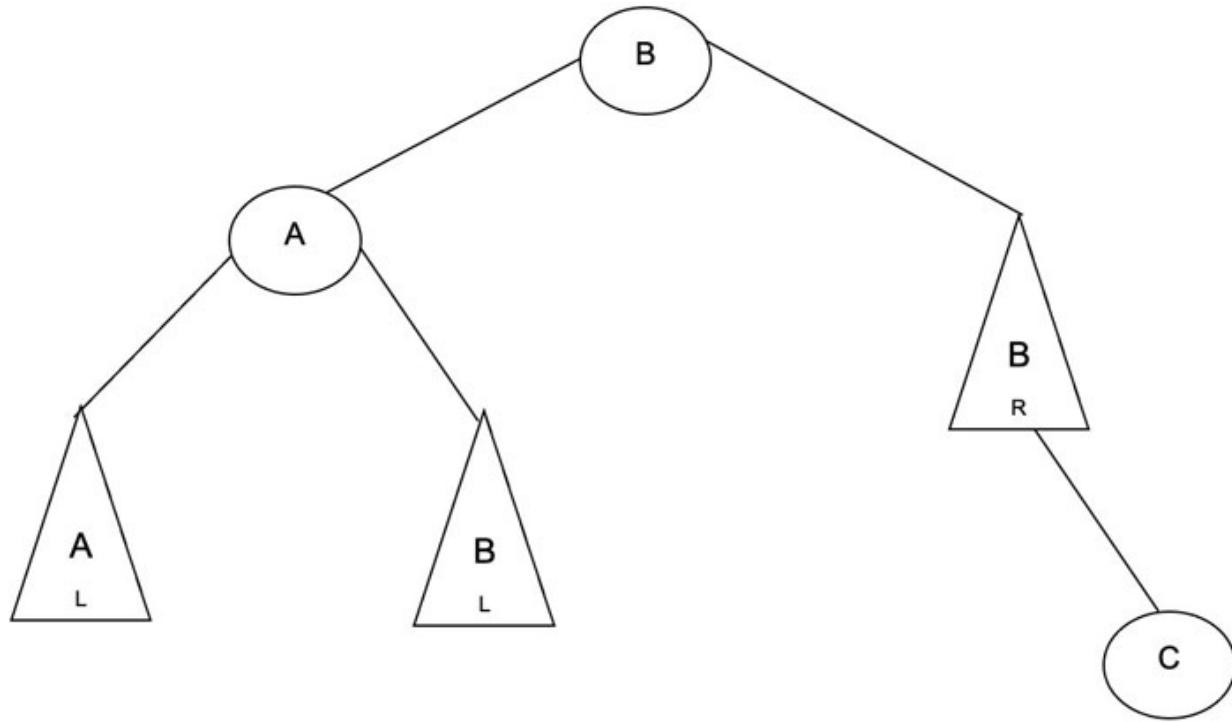
**Figure 9.4: LL rotation**

In the RR rotation, if the tree is initially balanced and a node is added to the right of the right sub-tree of the root (RR) ([figure 9.5](#)), which results in making the BF of the root -2. The resultant tree is no longer a balanced tree. The balancing is carried out as follows:



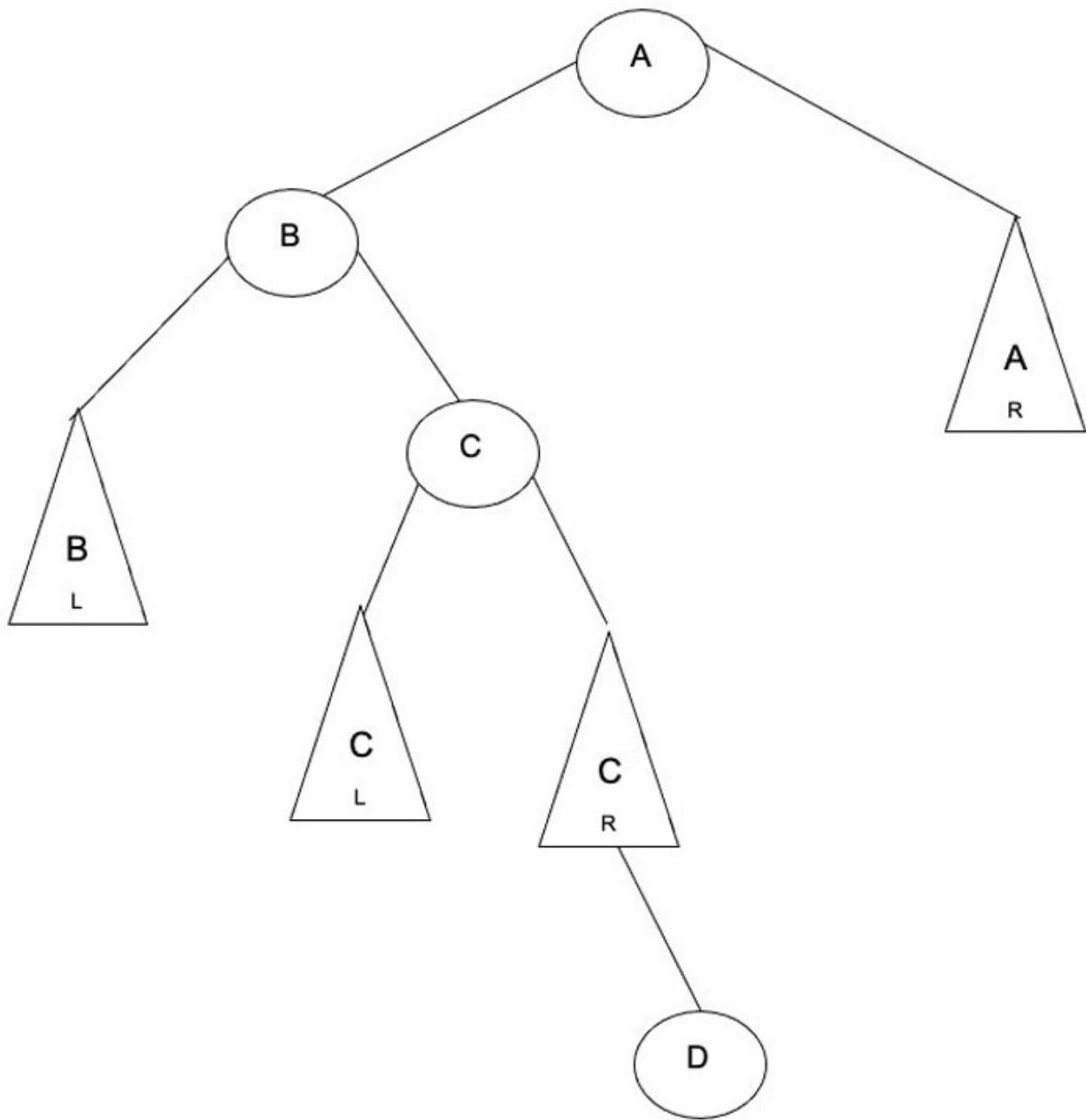
**Figure 9.5:** RR: Inserting node in a balanced tree

To convert the tree shown in [figure 9.5](#) to a balanced tree, B is made the root,  $B_R$  remains the right sub-tree of B, C remains as it is, A becomes the left node of B, and  $B_L$  becomes the right sub-tree of A.  $A_L$  remains as it is. The resultant tree is shown in [figure 9.6](#).



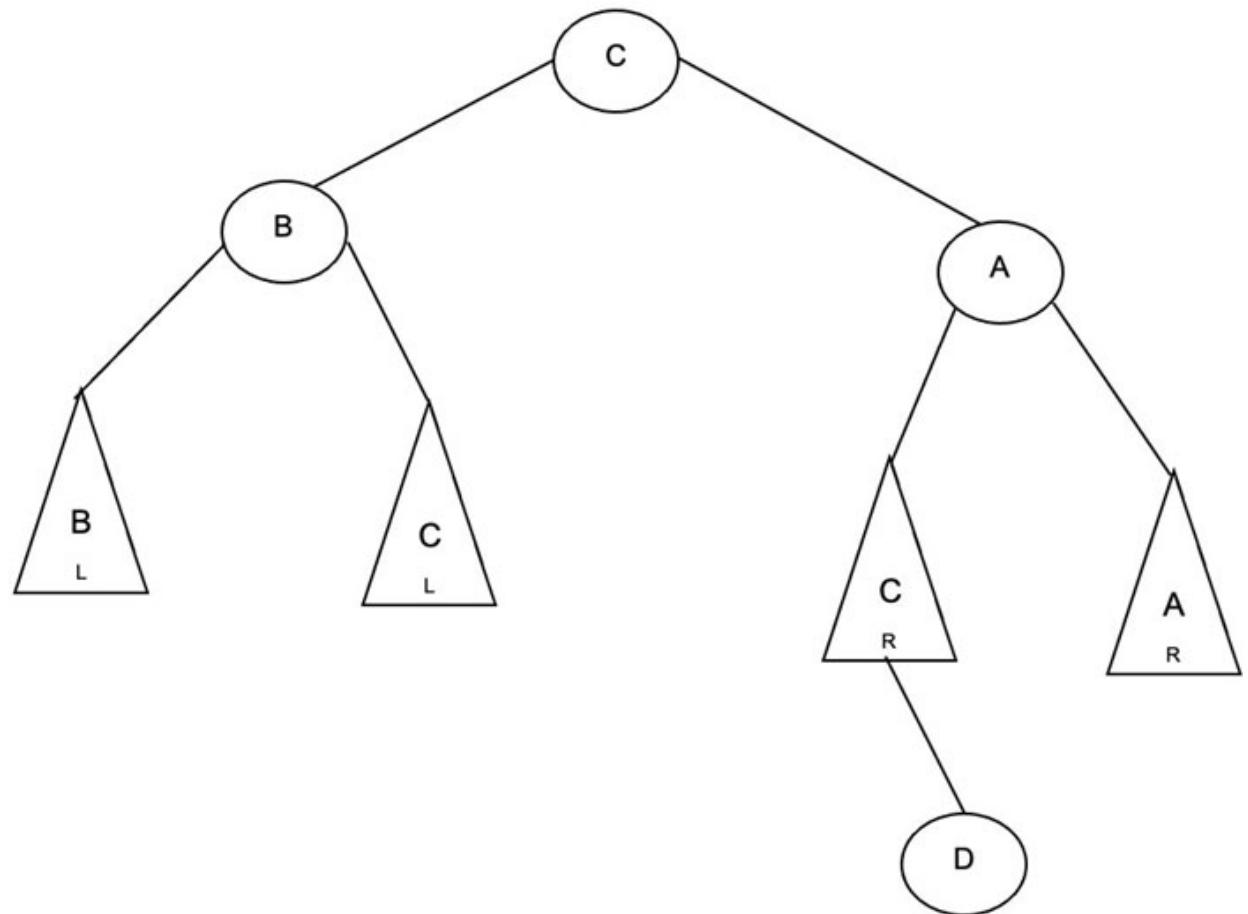
*Figure 9.6: RR rotation*

[Figure 9.7](#) to 9.10 show the LR and RL rotation. Note that initially, the tree was balanced, and after the insertion of a new node, it has at least one node having BF other than  $\{-1, 0, 1\}$ , and is no longer a balanced tree. The subsequent rotations help restore the balance of these trees.



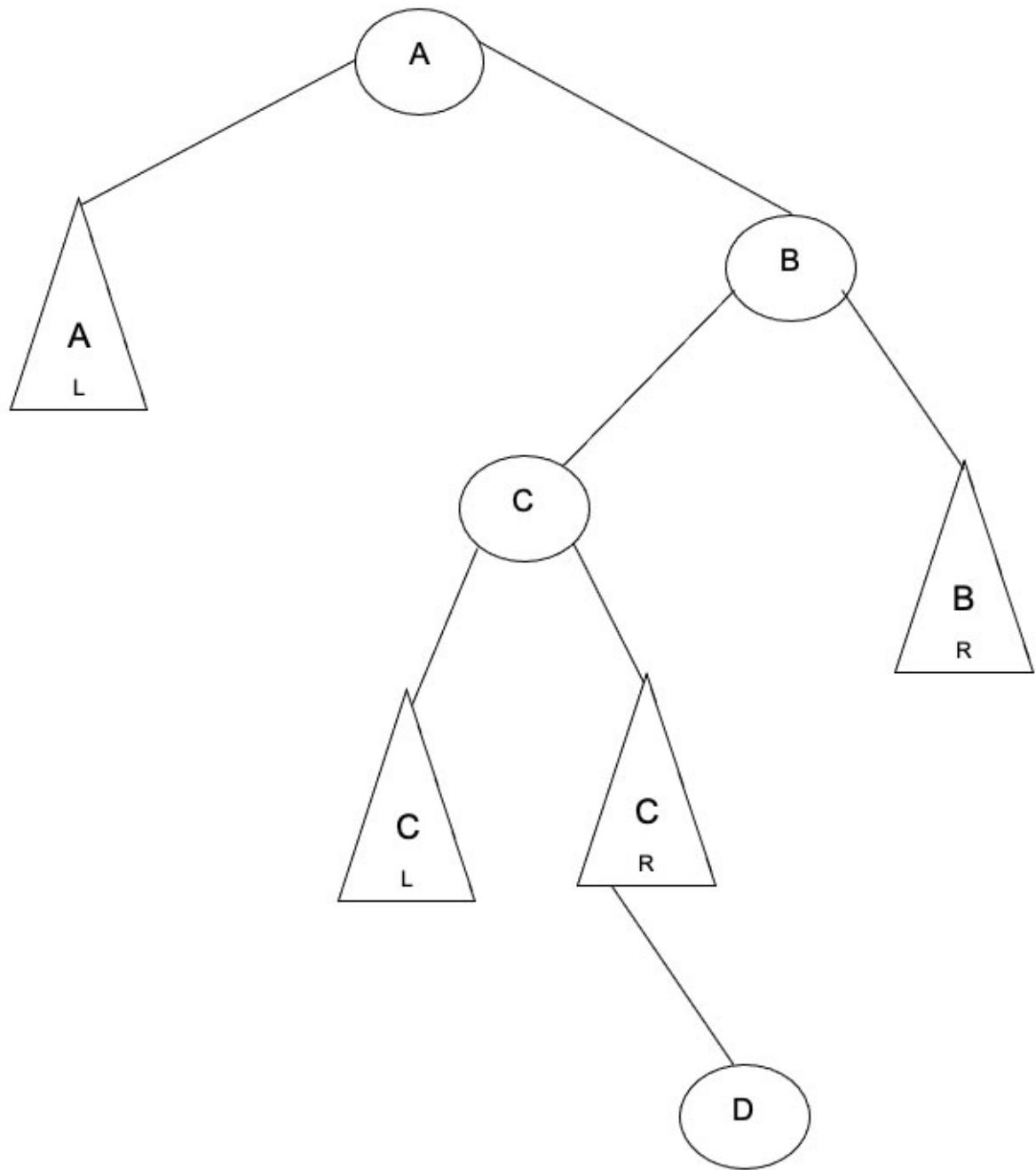
*Figure 9.7: LR: Insertion*

The LR rotation can be seen in the following figure:

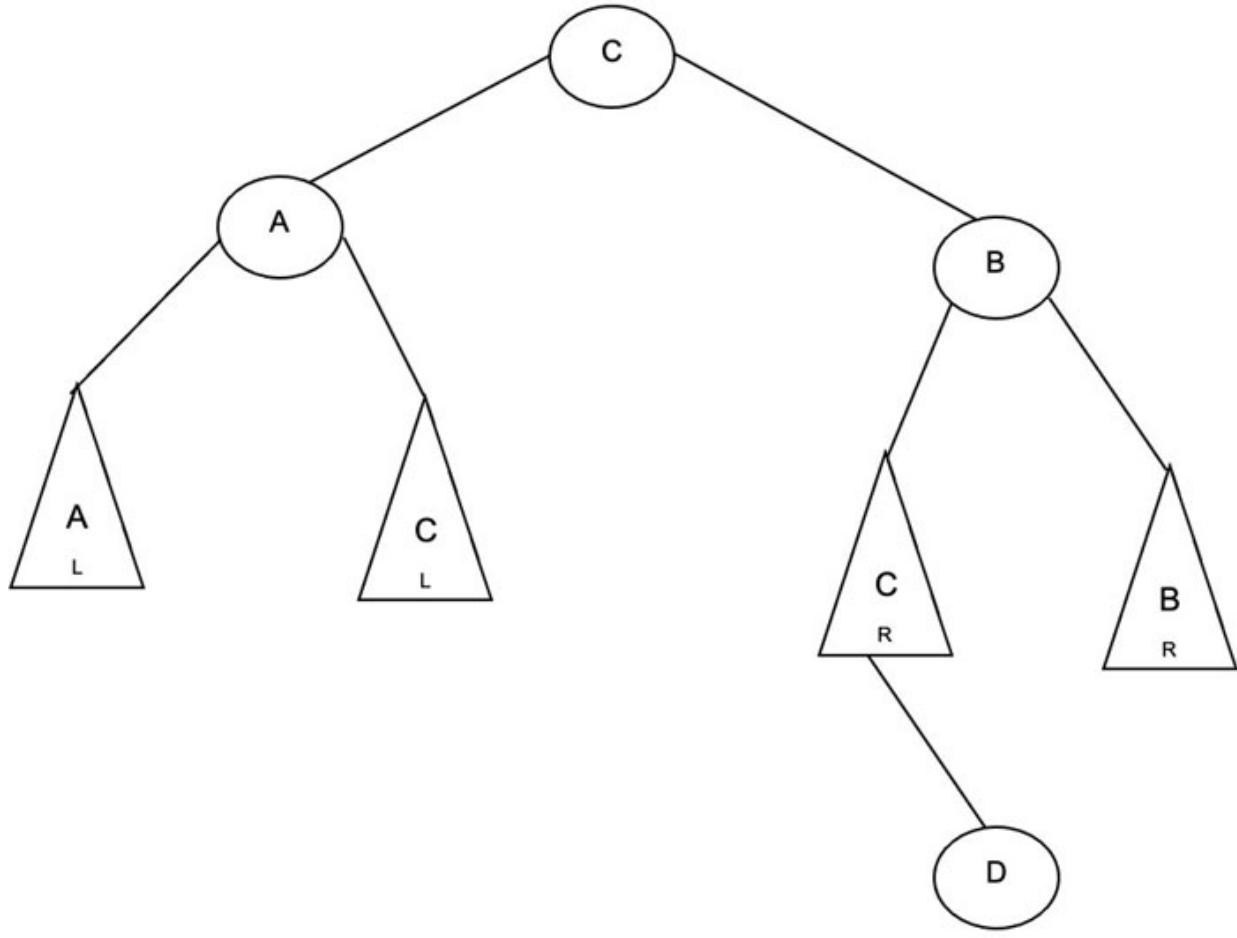


*Figure 9.8: LR rotation*

The LR insertion can be seen in the following figure:



*Figure 9.9: RL: Insertion*



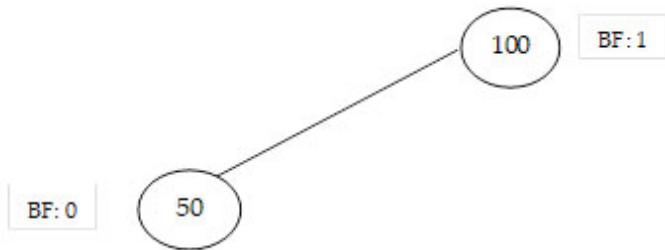
**Figure 9.10:** RL rotation

To understand insertion in an AVL tree, consider the following example. The first node becomes the root. The balance factor of this node is 0, as shown in [figure 9.11 \(a\)](#). The next item is less than the value at the root; hence, it becomes the left child of the root, as represented in [figure 9.11\(b\)](#). Likewise, since 30 is less than both 100 and 50, it becomes the left child of 50.

Root node 100



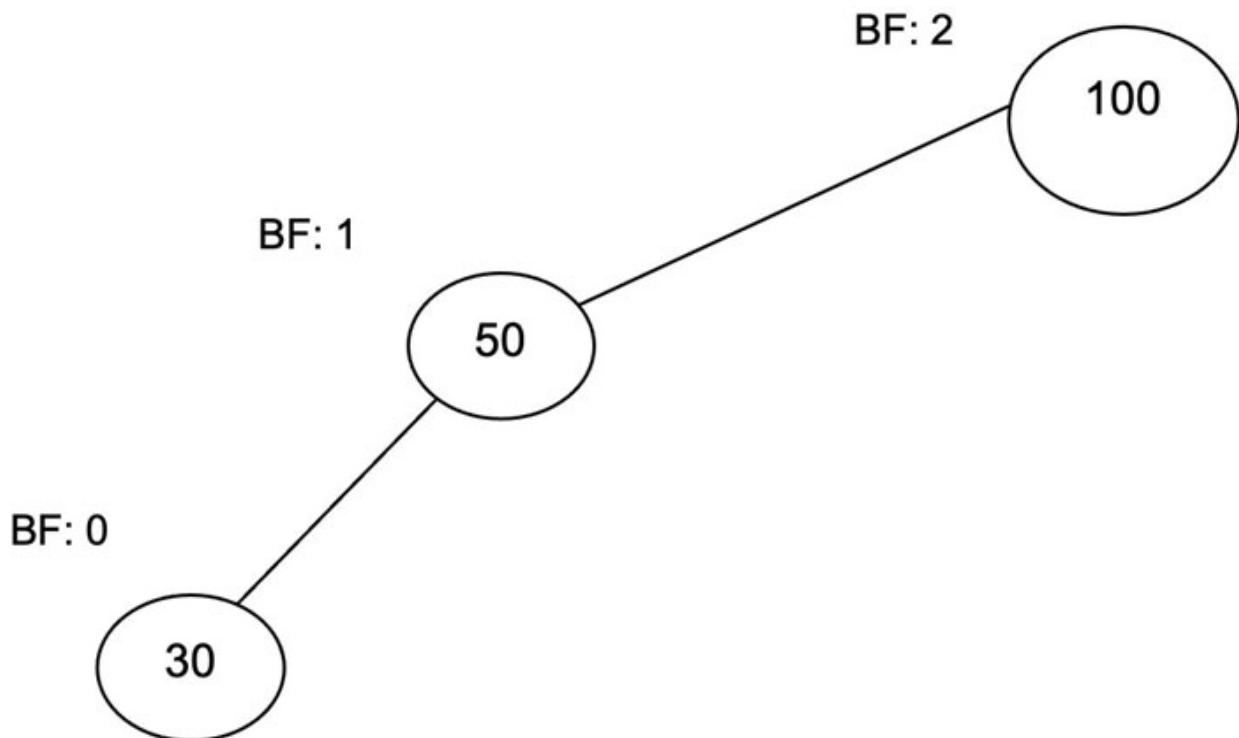
**Figure 9.11 (a):** The balance factor of the root is 0



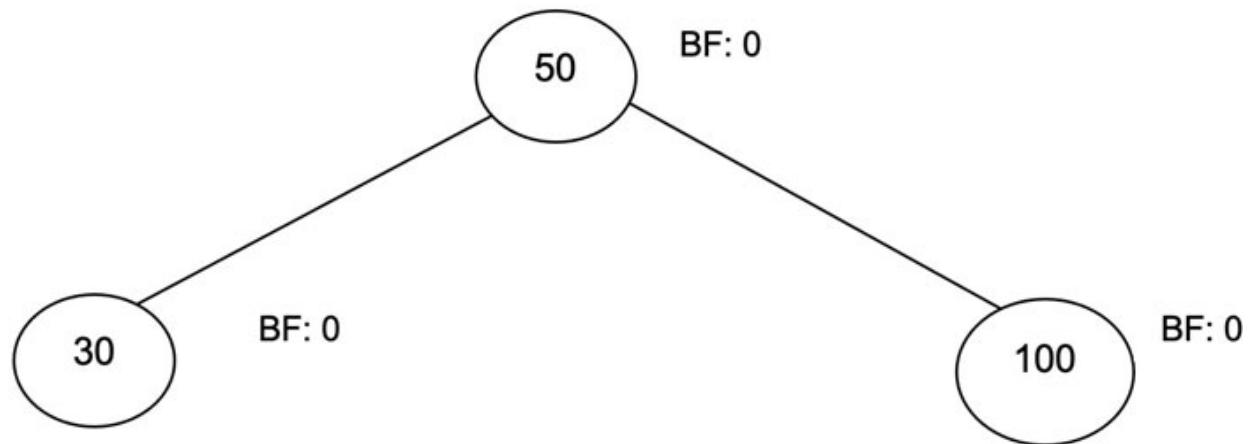
**Figure 9.11 (b):** The balance factors of nodes are 1 and 0, respectively

On inserting 30, the balance factor of the root becomes 2 ([figure 9.12\(a\)](#)). Note that the BF of the root is  $>1$ , and the node to be inserted is less than the value of the root, so LL rotation will be carried out ([figure 9.12 \(b\)](#)). From the last level, the node having  $\text{BF} = 2$  will be mapped

Inserting 30

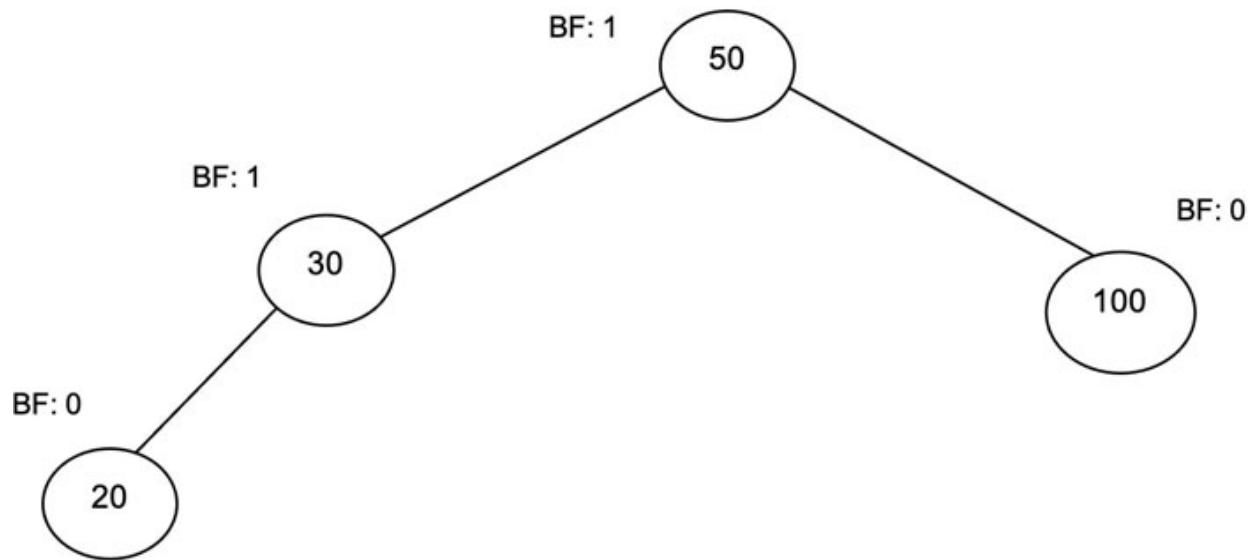


**Figure 9.12 (a):** The balance factors of nodes are 2, 1, and 0, respectively

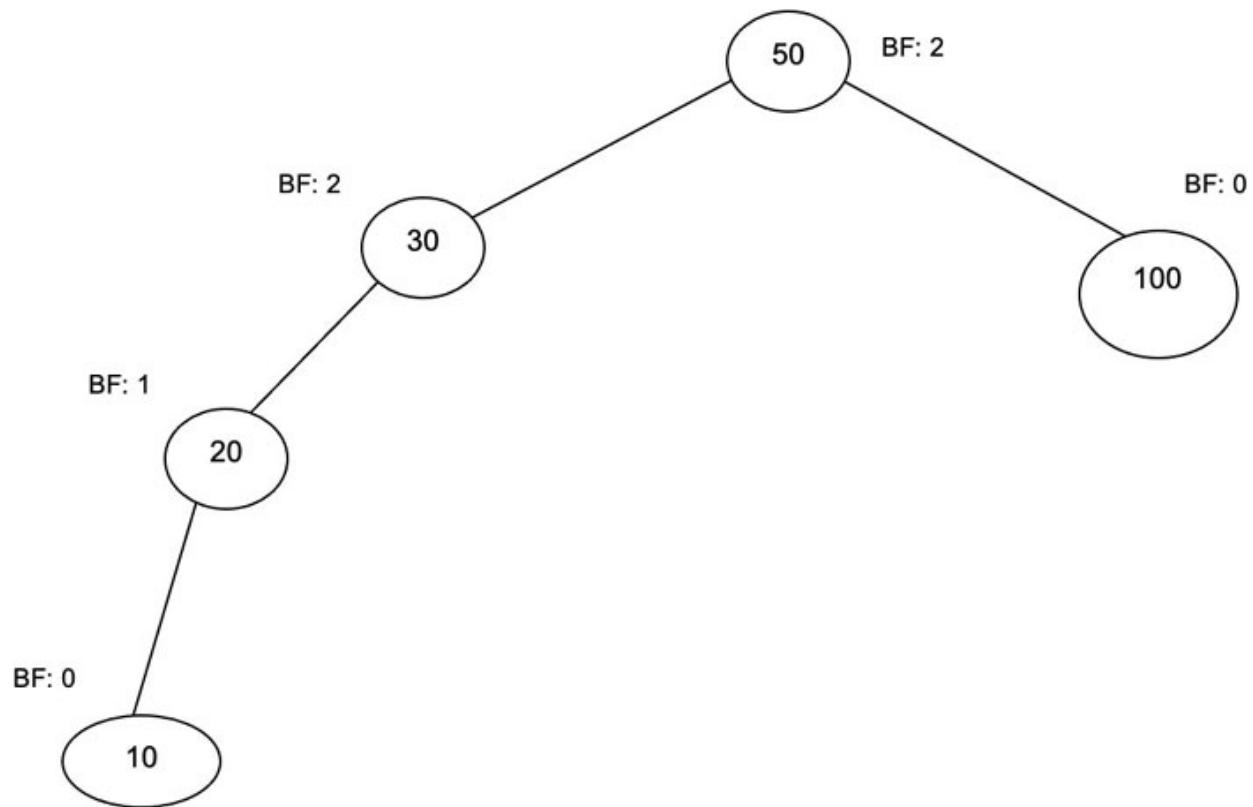


*Figure 9.12 (b): The balanced tree after LL rotation*

Likewise, inserting 20 followed by 10 would result in an unbalanced tree again ([figure 9.13](#)):

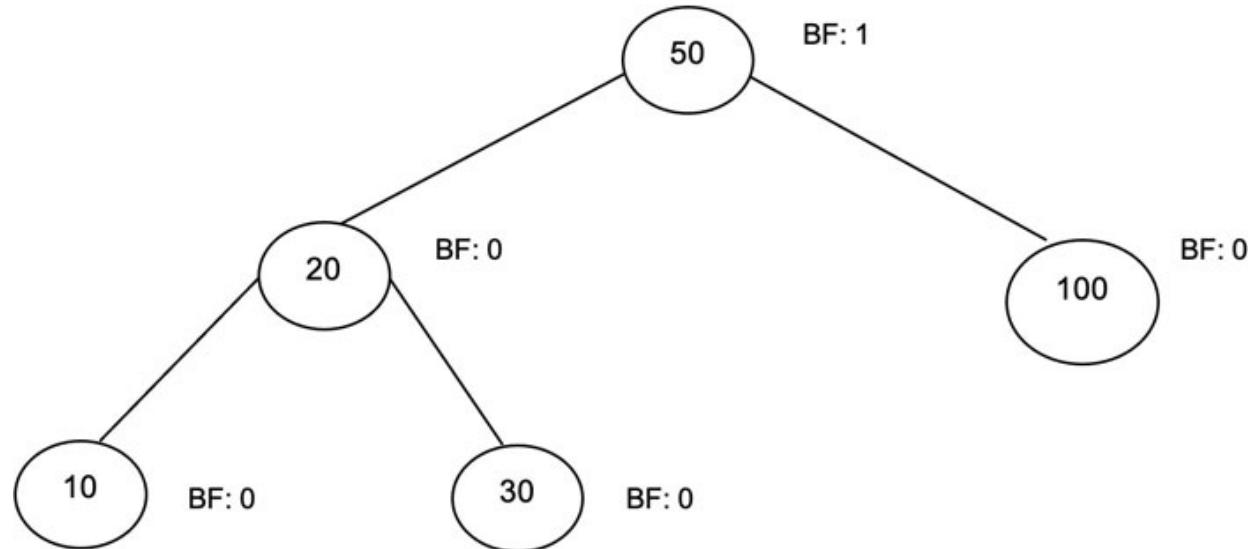


*Figure 9.13 (a): Inserting 20*



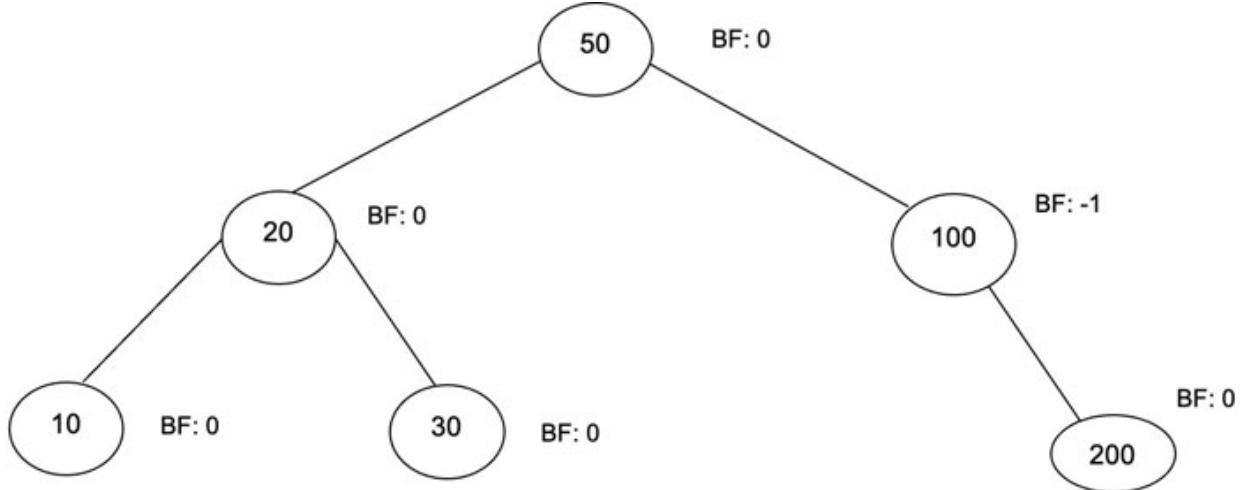
*Figure 9.13 (b): Inserting 10 would make the tree unbalanced*

Note that the first node from the last level, whose BF is not  $\{-1, 0, 1\}$ , needs to be handled. In this case, the balancing can be carried out using LL rotation on the sub-tree rooted at 30 ([figure 9.14](#)):

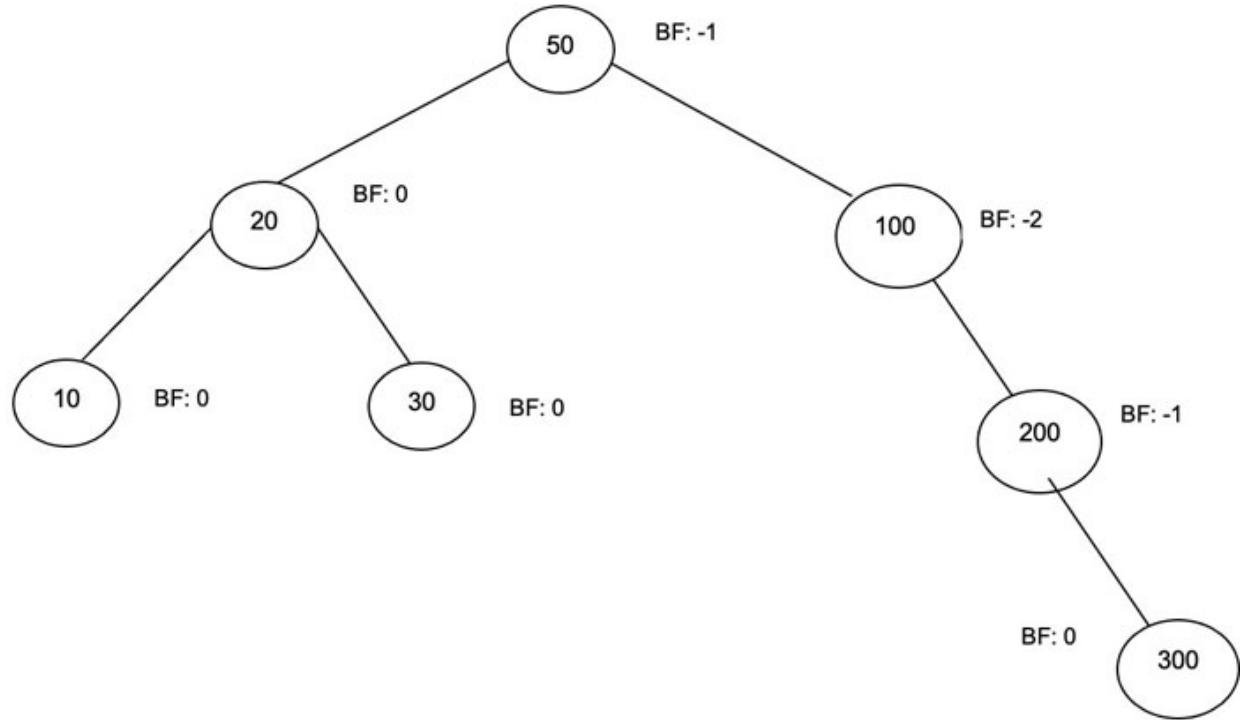


*Figure 9.14: LL rotation at the sub-tree rooted at 30*

Having seen two examples of LL rotation, let us move to the RR rotation. Inserting 200 ([figure 9.15 \(a\)](#)), followed by 300, would result in an unbalanced tree, as shown in [figure 9.15 \(b\)](#):

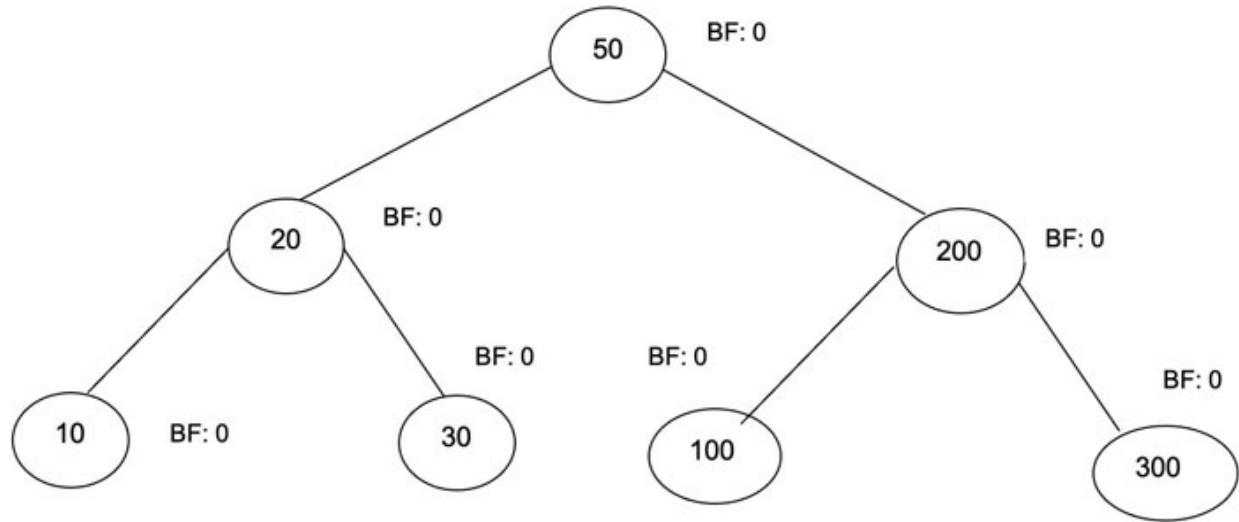


*Figure 9.15 (a): Inserting 200 in BST*



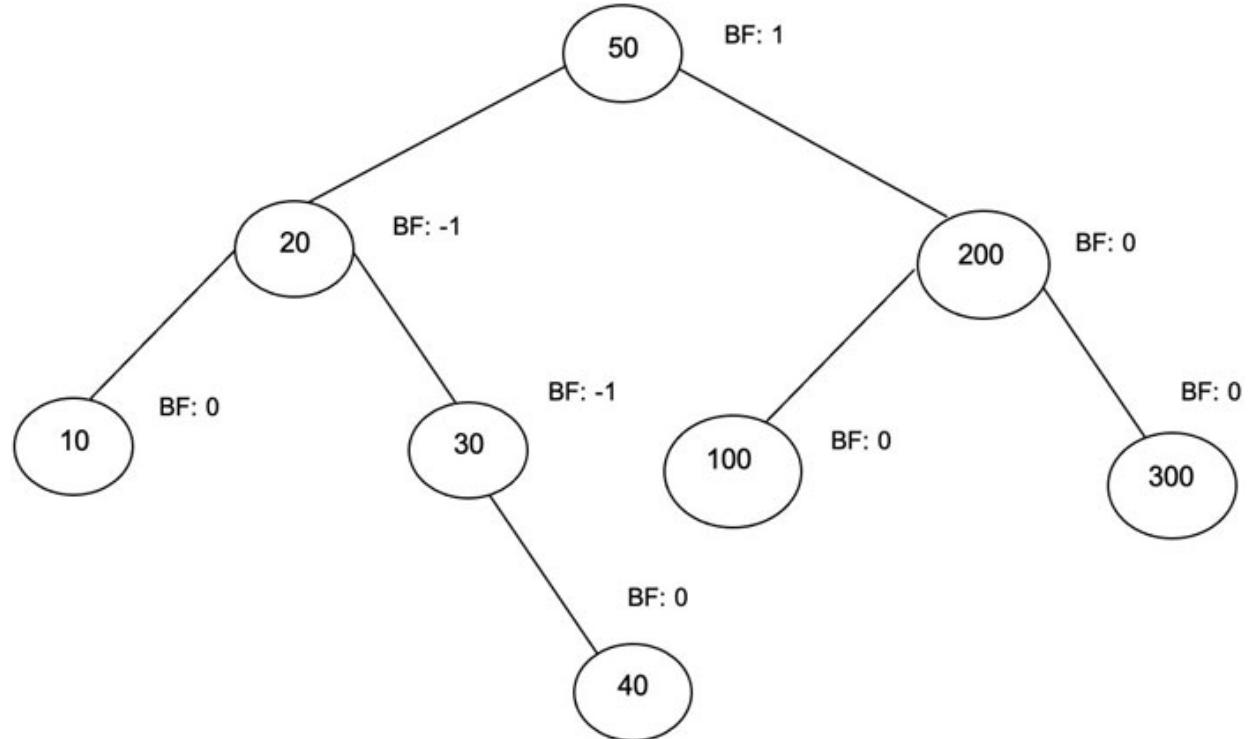
*Figure 9.15 (b): Inserting 300 in BST*

Note that the first node from the last level, whose BF is not  $\{-1, 0, 1\}$ , needs to be handled. In this case, the balancing can be carried out using RR rotation on the sub-tree rooted at 100, as shown in [figure 9.16](#):

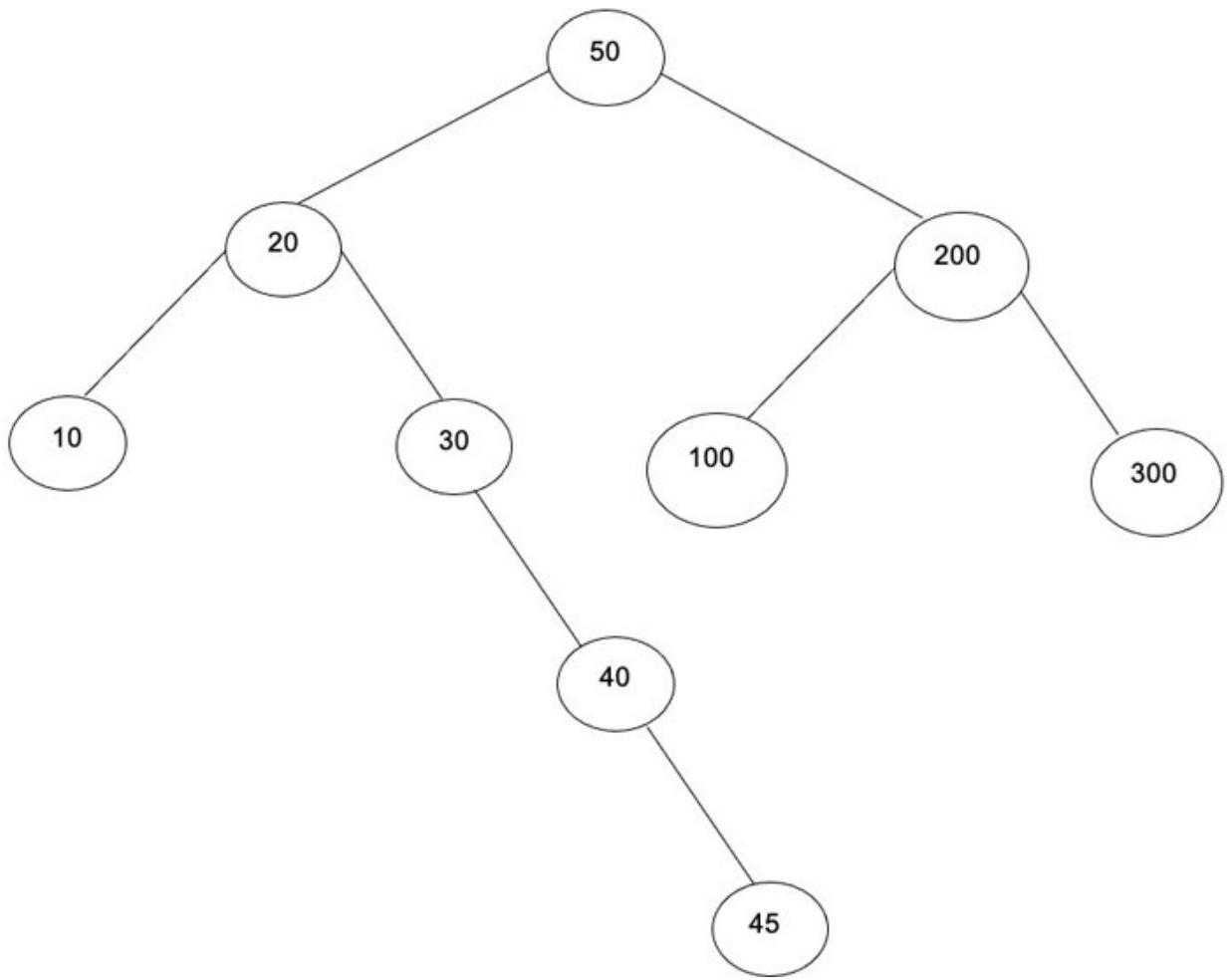


*Figure 9.16: Balanced Tree*

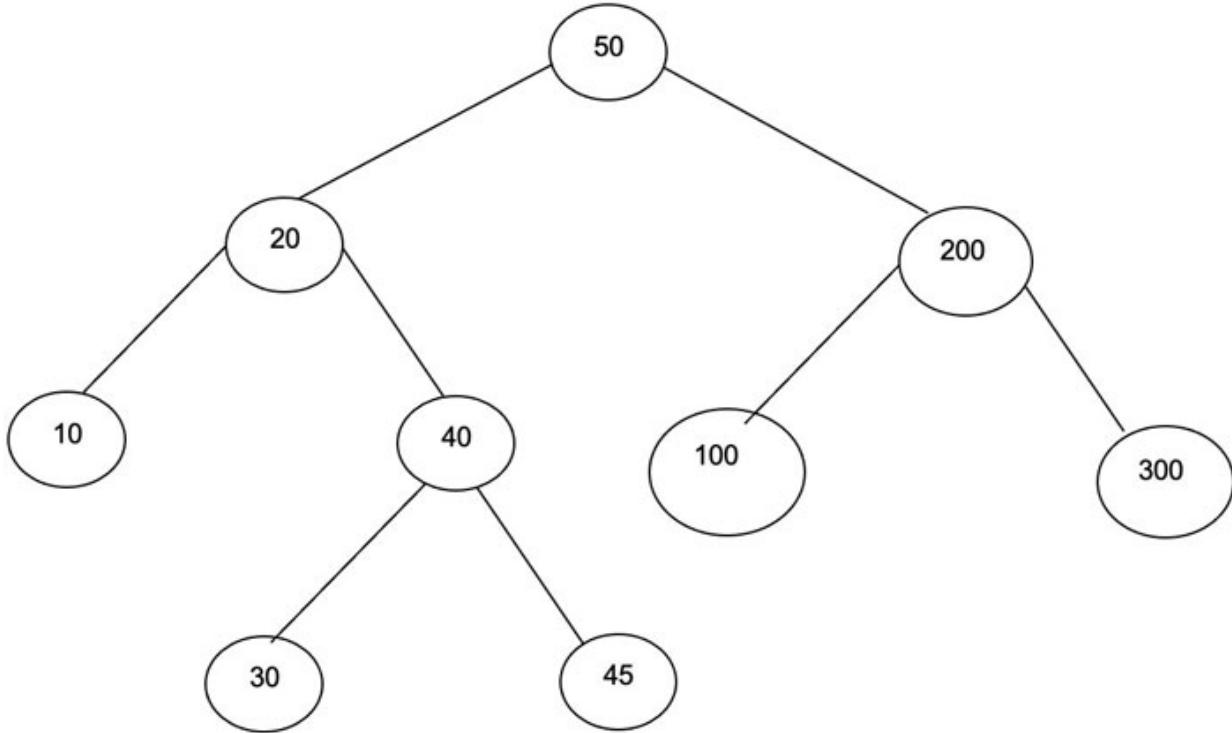
In the figures that follow, inserting 40, followed by 45, makes the tree unbalanced, as shown in [figure 9.17](#). The balance can be resorted by applying the RR rotation. Please refer to the following figure:



*Figure 9.17 (a): Inserting 40*



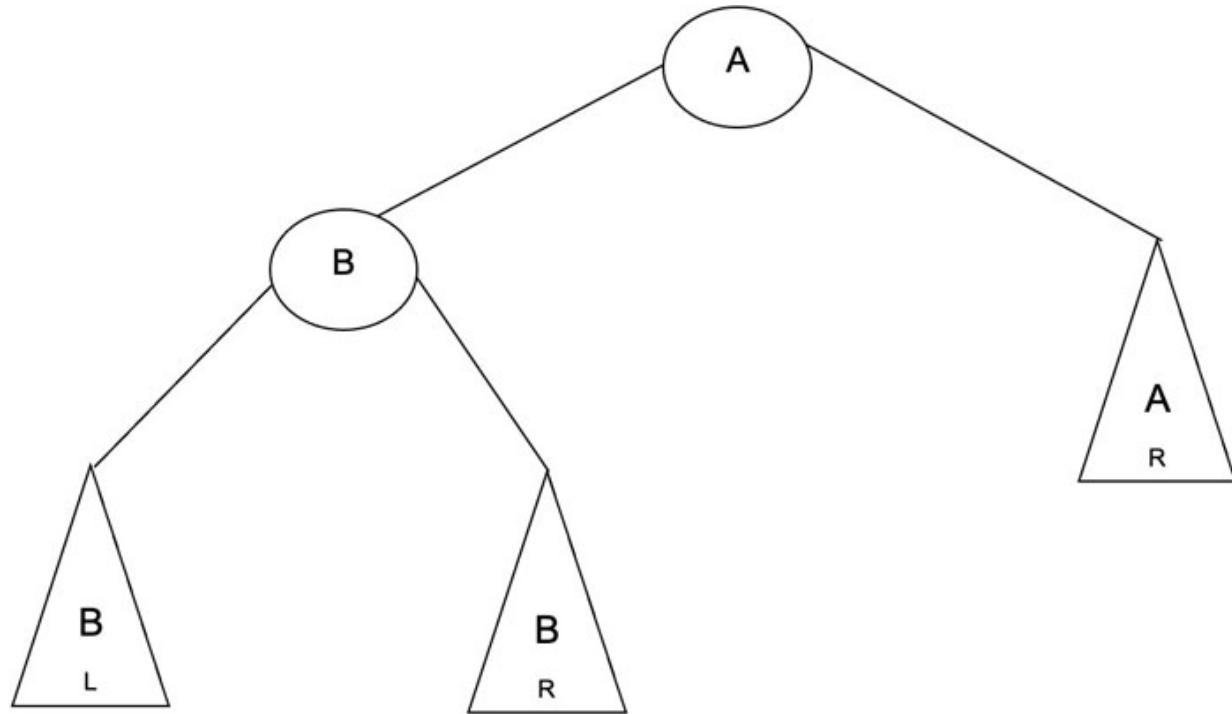
**Figure 9.17 (b):** Inserting 45



**Figure 9.17 (c):** Balance restored by applying RR rotation to the sub-tree rooted at 30

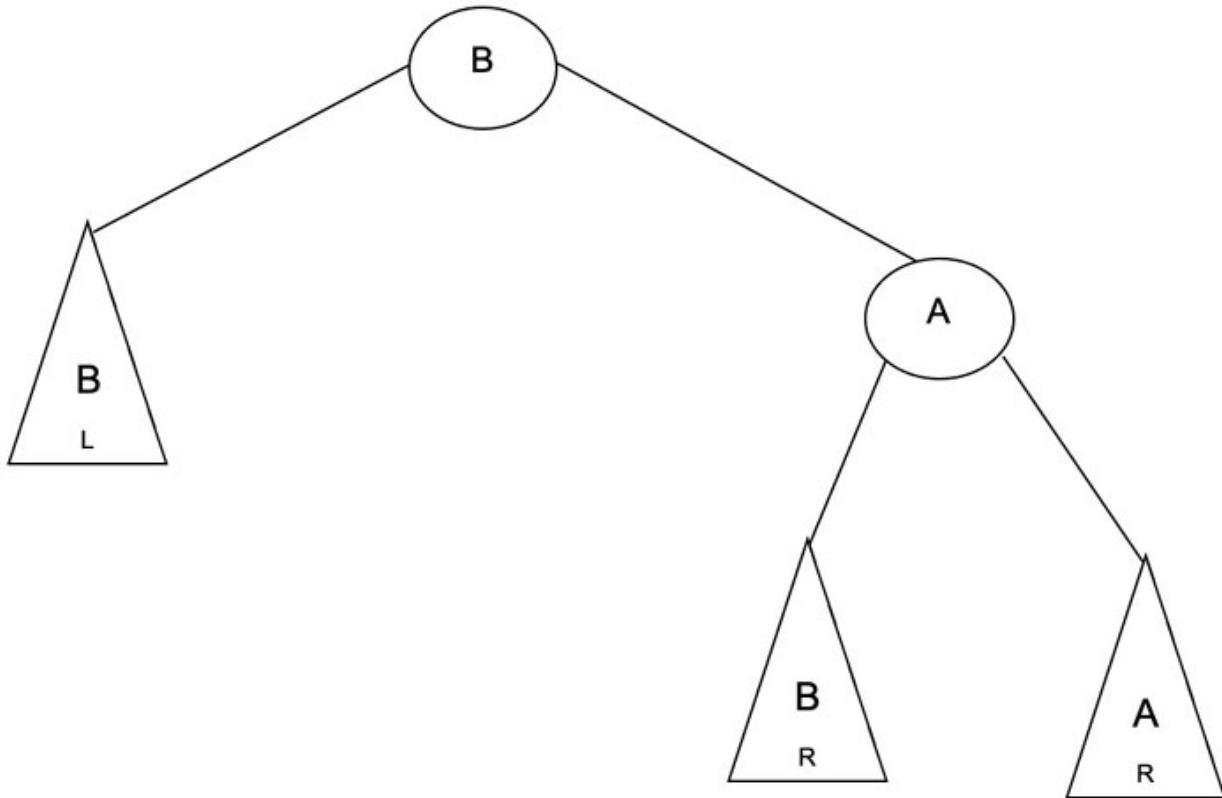
### Deletion from an AVL Tree

Note that, in the tree shown in [figure 9.18](#), if the heights of sub-trees BL, BR, and AR are  $h$  each, then the left sub-tree will have a height of  $(h+1)$ , and the right sub-tree will have a height of  $h$ . The deletion of the node from the left sub-tree would lead to the BF of A becoming 2. That is, the tree will no longer be balanced. In this case, the tree became unbalanced since a node was removed from the right sub-tree (hence, R), and the BF of the root of the left sub-tree was 0. Therefore, the method to restore the balance of this sub-tree would henceforth be referred to as R0 rotation. Please refer to the following figure:



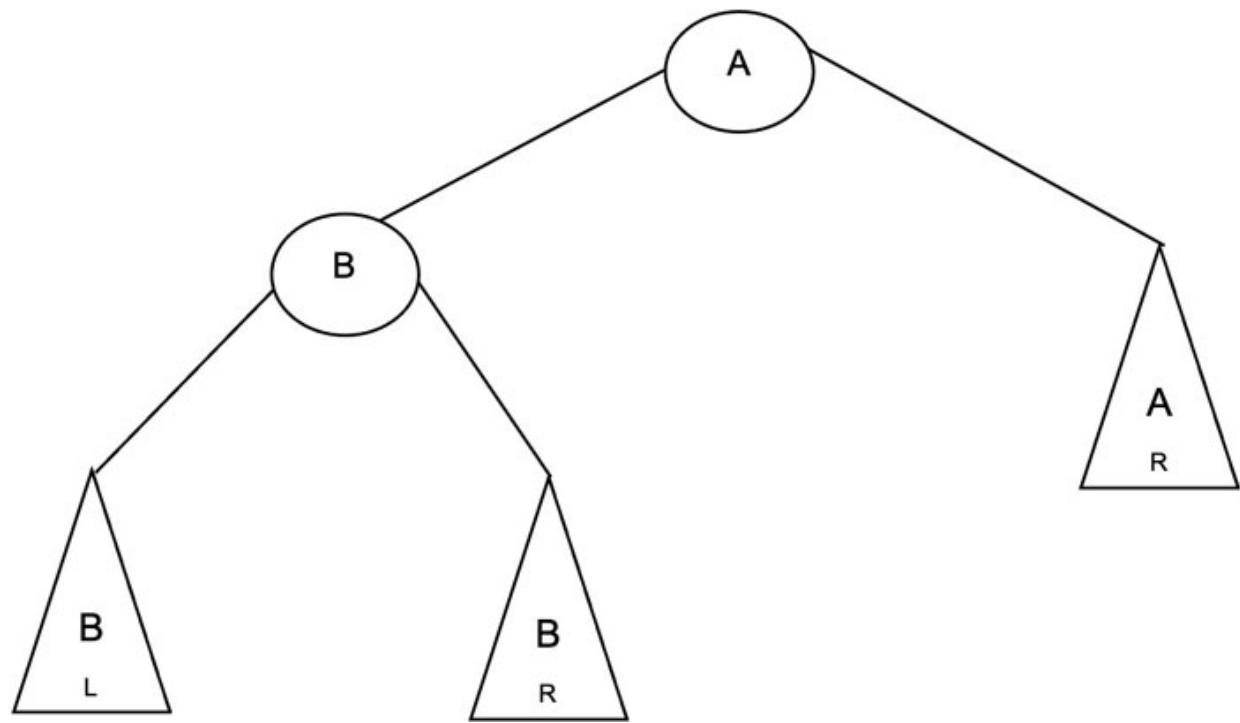
**Figure 9.18:** The deletion of a node from the right sub-tree will make it unbalanced

To make it balanced, the rotation shown in [figure 9.19](#) is carried out. Note that after the rotation, the left sub-tree has depth = $h$ , and the right sub-tree has depth =  $(h+1)$ . Hence, the difference between the depths of the two sub-trees is 1, and the tree becomes balanced. Please refer to the following figure:

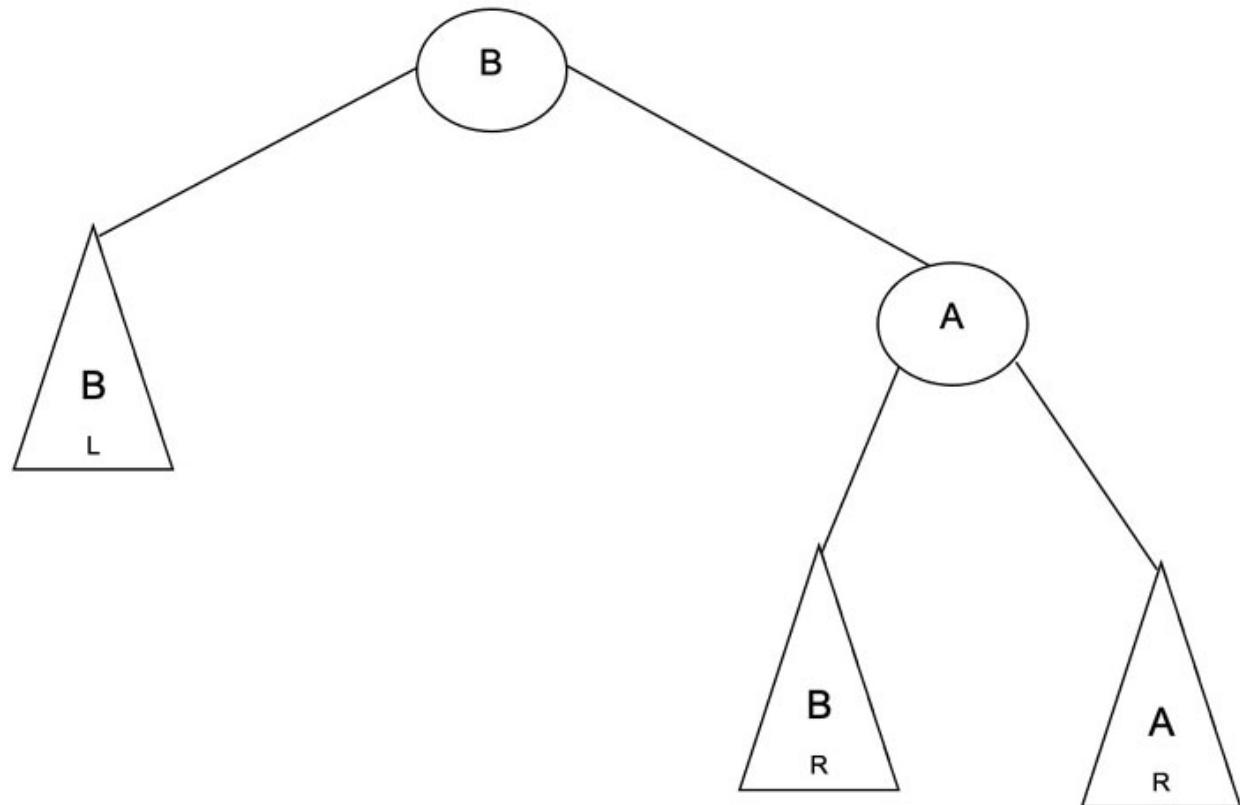


**Figure 9.19:** R0 rotation after deletion from right sub-tree

We will now generalize the preceding algorithm. The R1 rotation is carried out when an item is deleted from the right sub-tree, and the BF of the root of the left sub-tree is 1; that is, the left sub-tree of B as shown in [figure 9.20 \(a\)](#), it has depth  $(h+1)$  and the right sub-tree of B has depth  $h$ . [Figure 9.20 \(b\)](#) shows the balance restored after carrying around the R1 rotation:

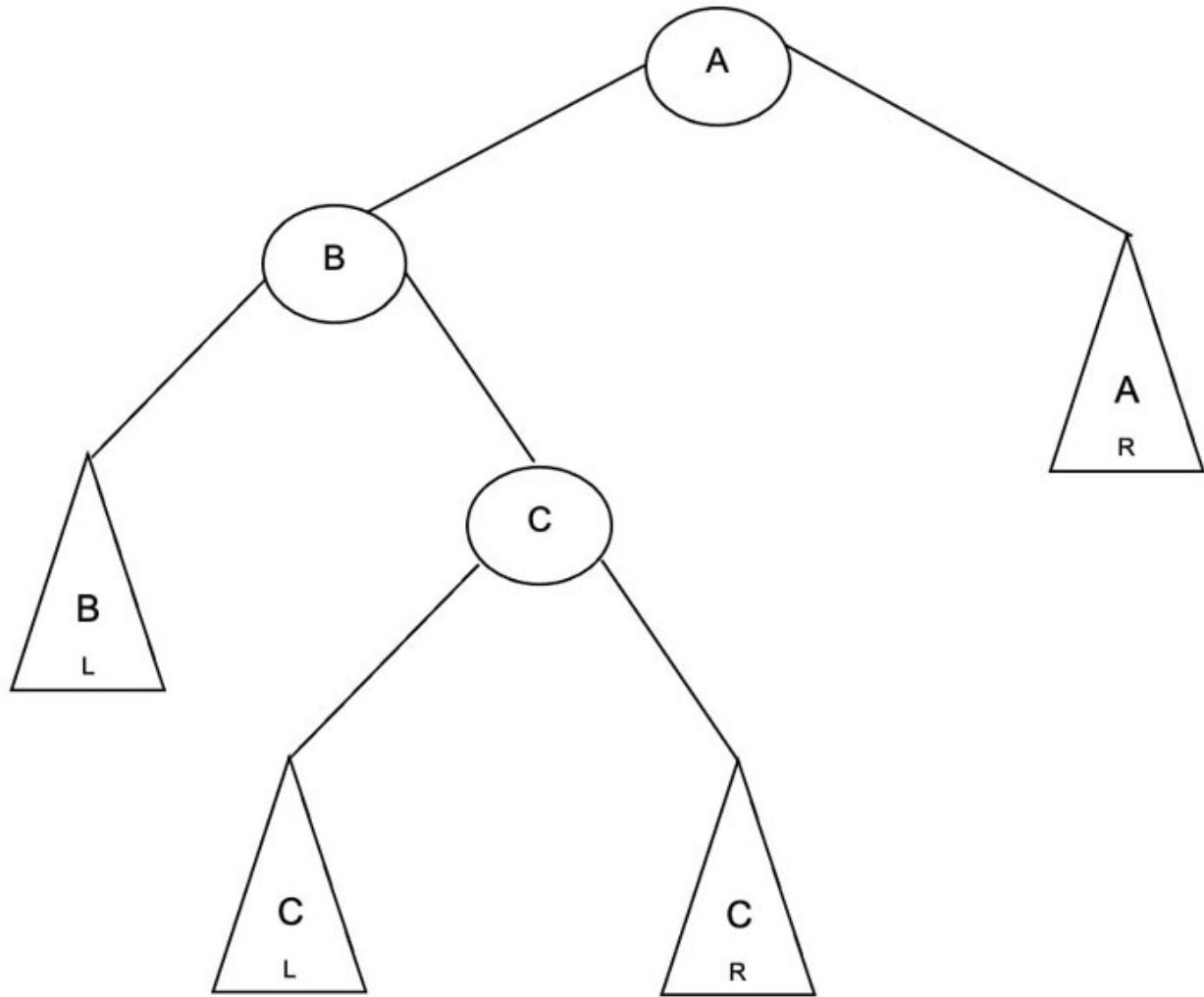


*Figure 9.20 (a): BL has depth  $h$ , and BR has depth  $(h-1)$*

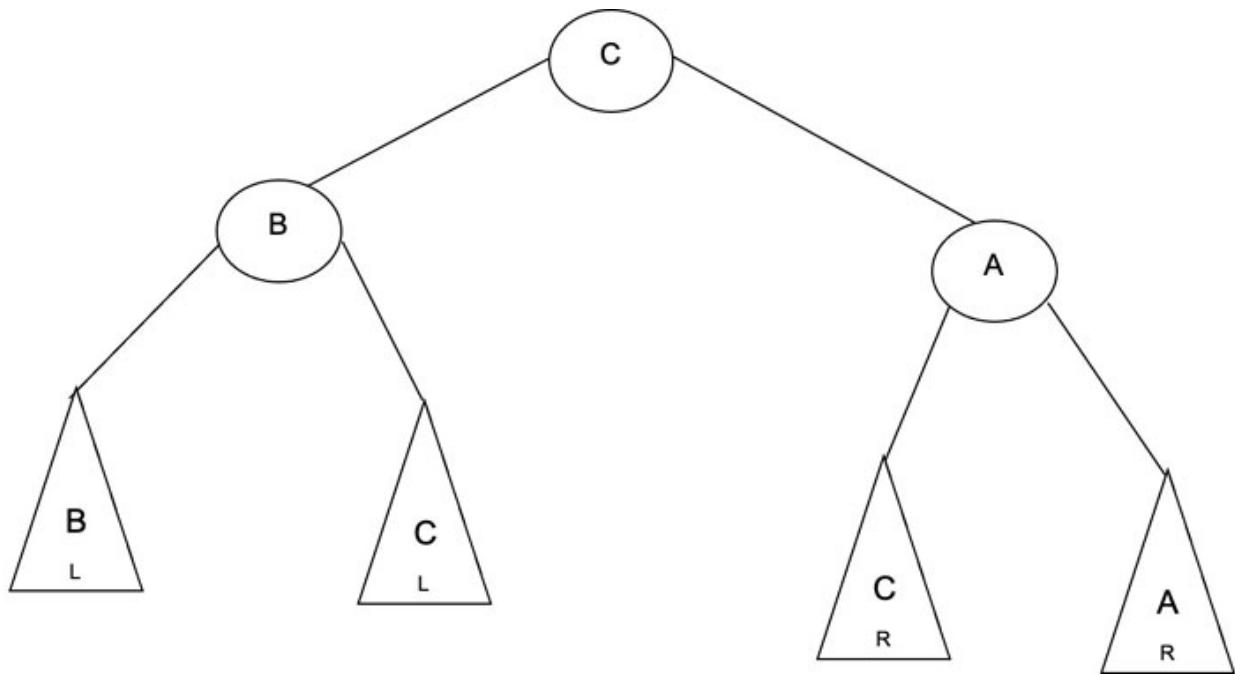


*Figure 9.20 (b): The balance is restored after carrying out the R1 rotation*

Likewise, in the case of the tree shown in [figure 9.21\(a\)](#), the BF of the root of the left sub-tree is  $-1$ . The tree becomes unbalanced after deleting an element from the right sub-tree. [Figure 9.21\(b\)](#) shows the tree after applying R-1 rotation. Please refer to the following figure:

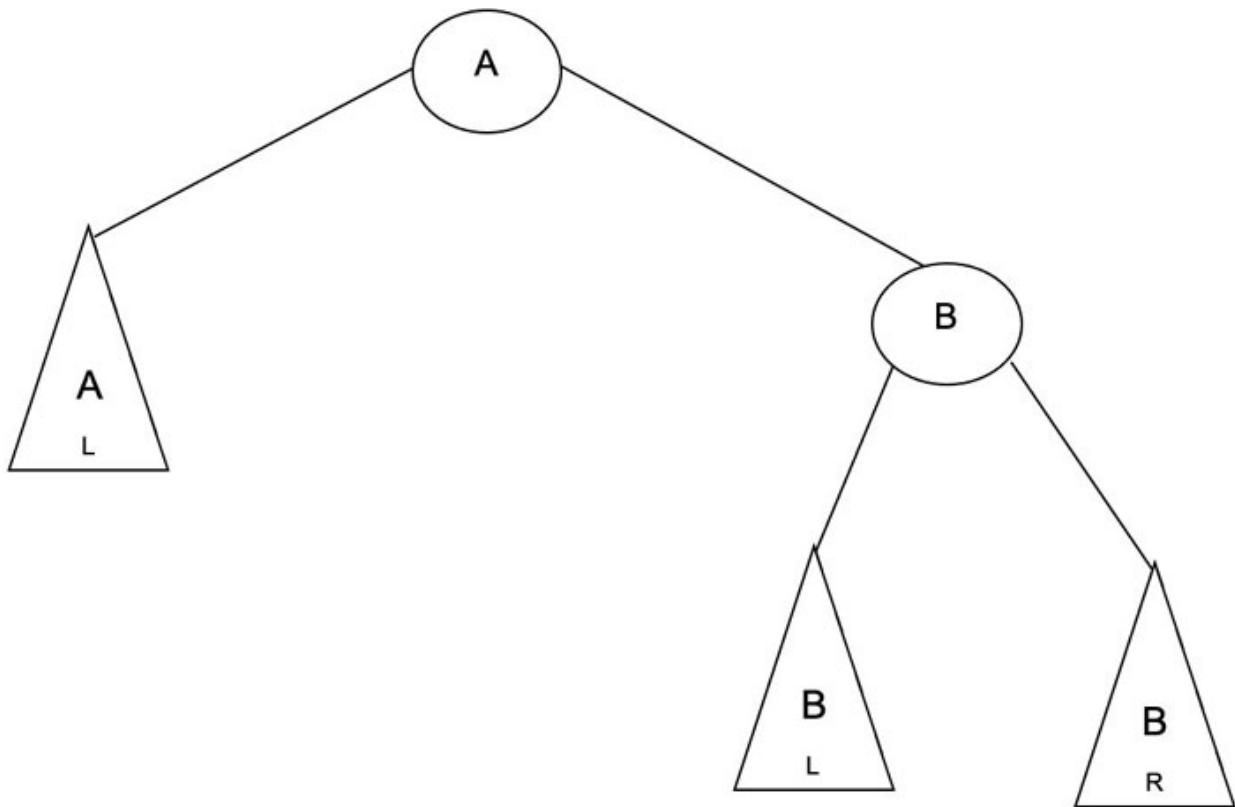


**Figure 9.21 (a):** Deleting node from the right sub-tree results in an unbalanced tree

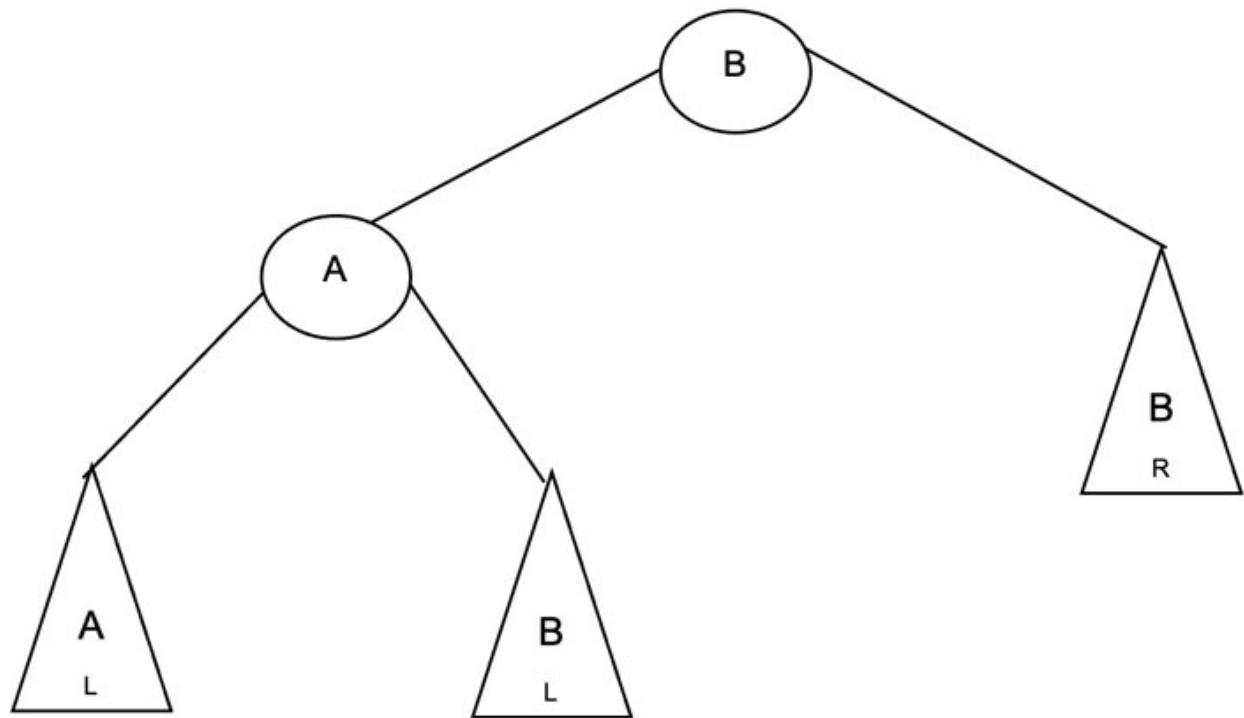


*Figure 9.21 (b): Resorting balance after carrying out the R-1 rotation*

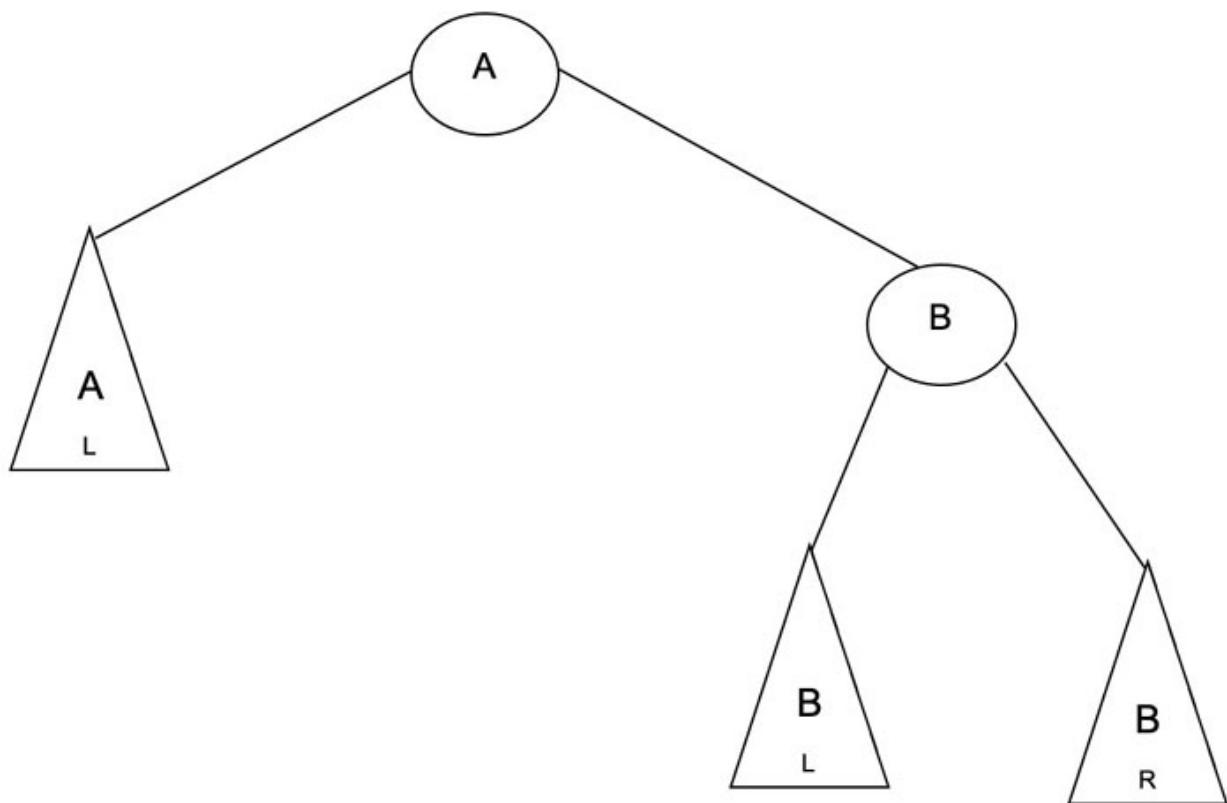
In the same way, the L0, L1, and L-1 rotations have been shown in [figures 9.22 \(a\)](#) to [9.22 \(f\)](#):



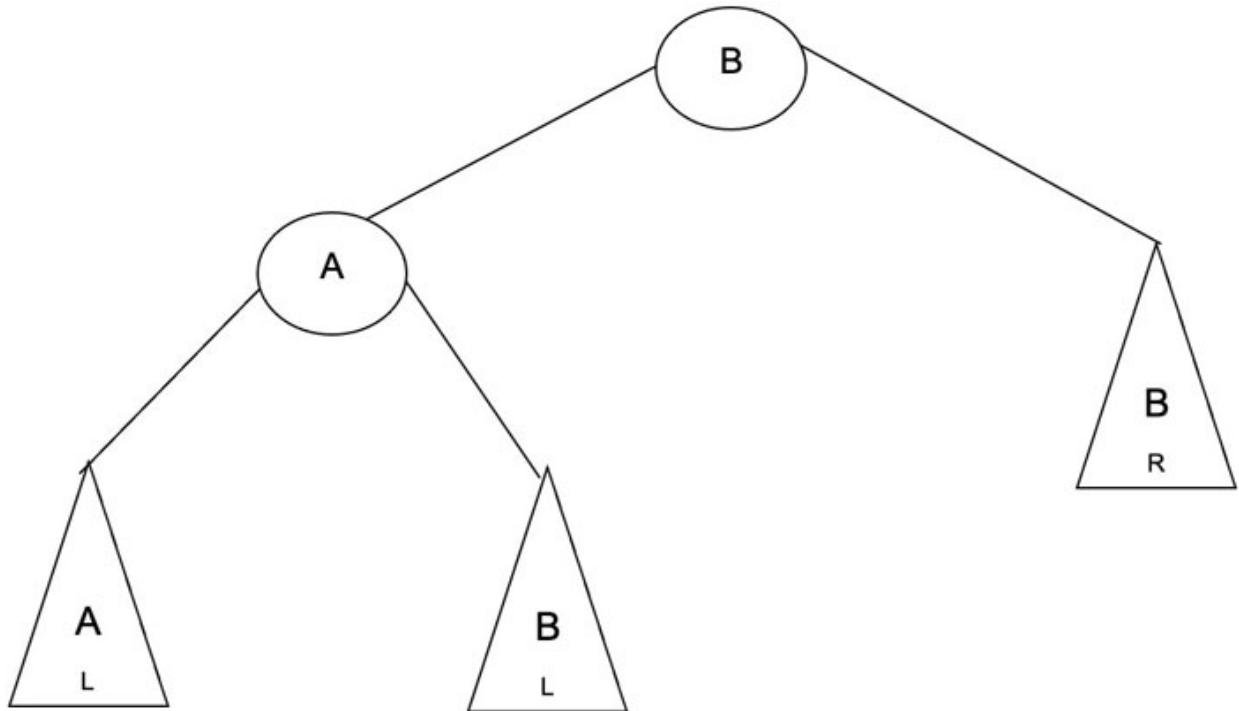
**Figure 9.22 (a):** Deleting the node from the left sub-tree would result in an unbalanced tree. The BF of B is 0.



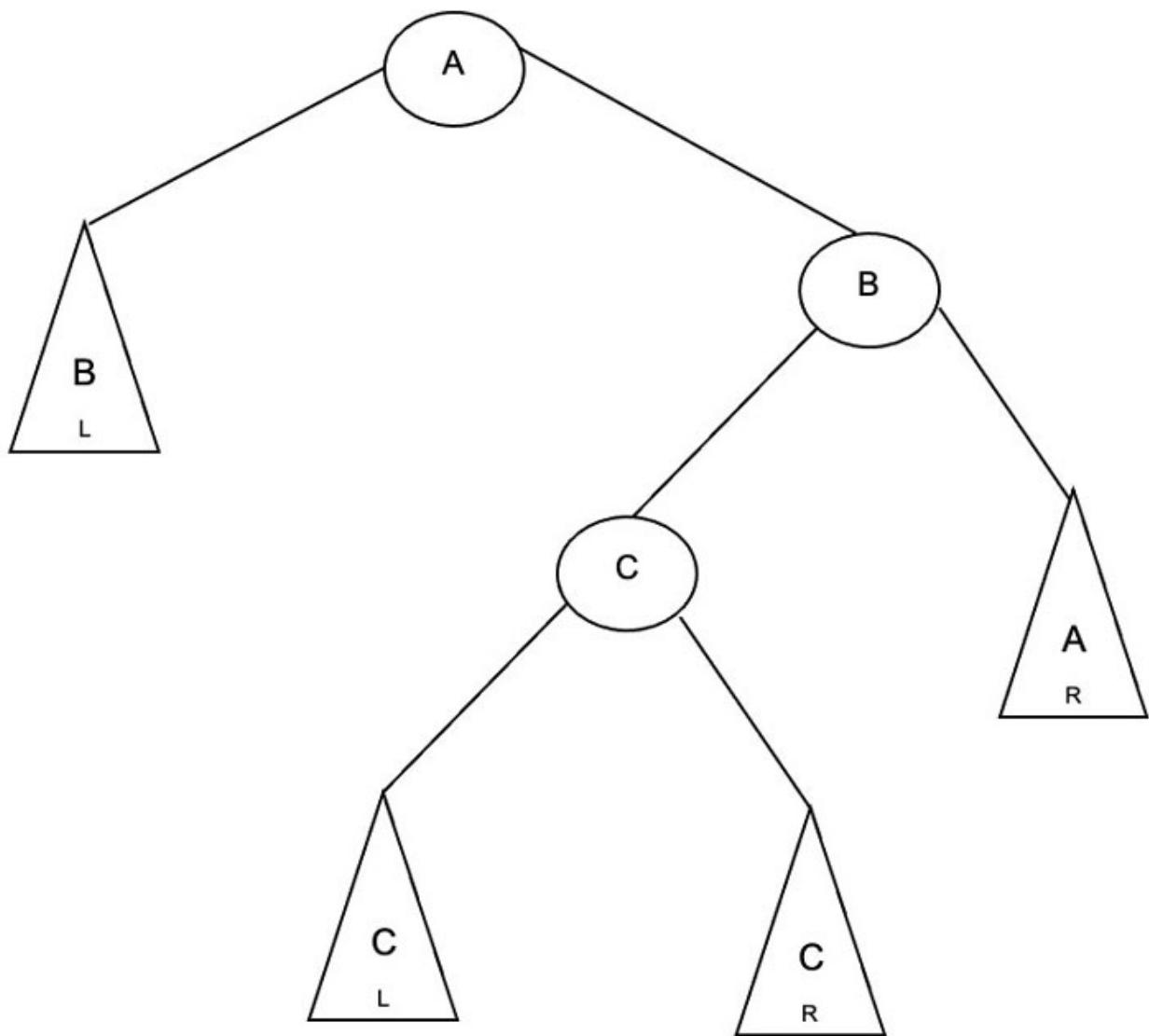
**Figure 9.22 (b):** L0 rotation



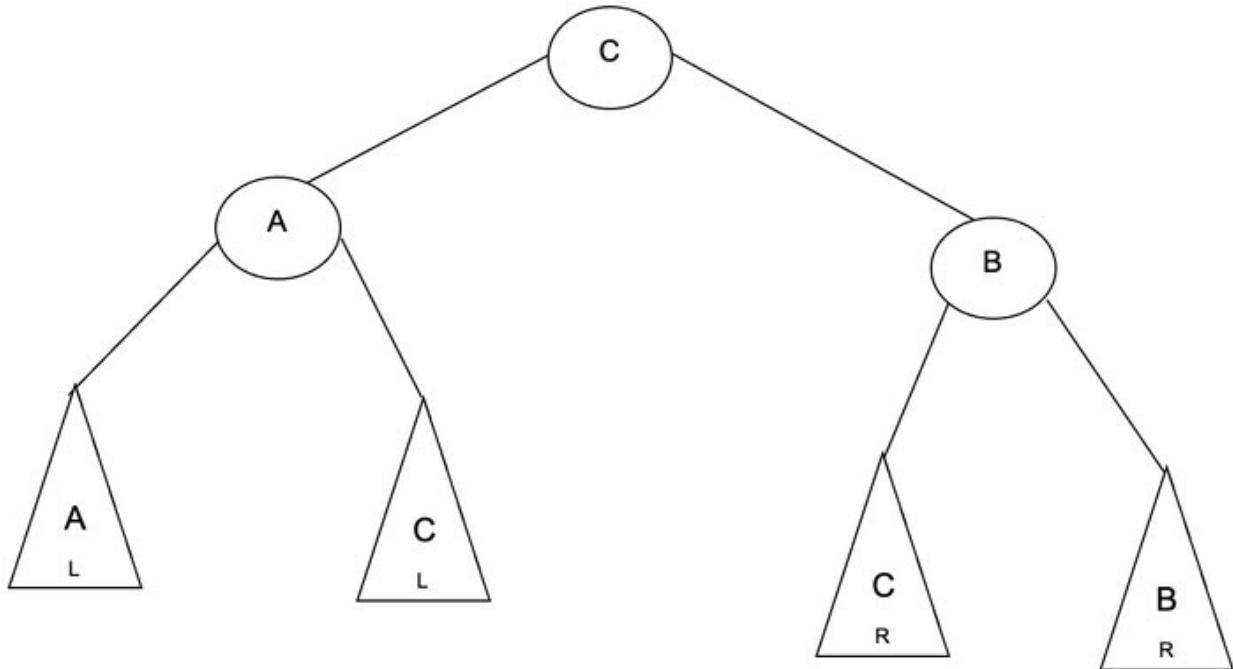
**Figure 9.22 (c):** Deleting the node from the left sub-tree would result in an unbalanced tree. The BF of B is 1. The depth of BL is  $(h+1)$ , and that of BR is  $h$ .



**Figure 9.22 (d):** L-1 Rotation



**Figure 9.22 (e):** Deleting the node from the left sub-tree would result in an unbalanced tree. The BF of B is -1. The depth of BL is  $h$ , and that of BR is  $(h+1)$

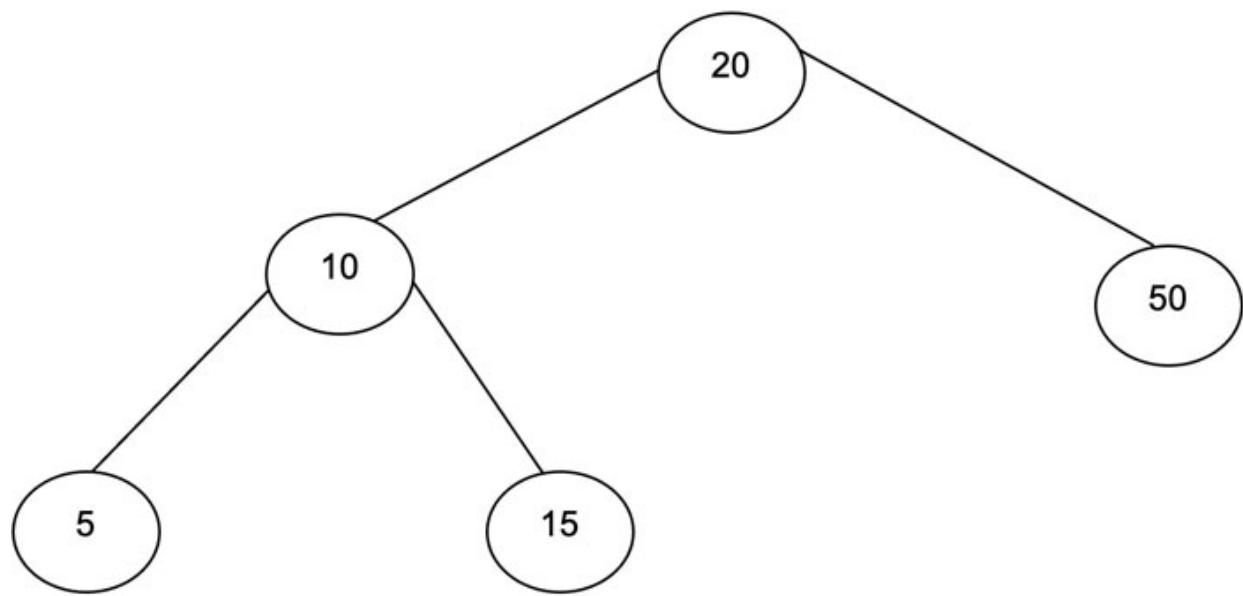


*Figure 9.22 (f): L-1 rotation*

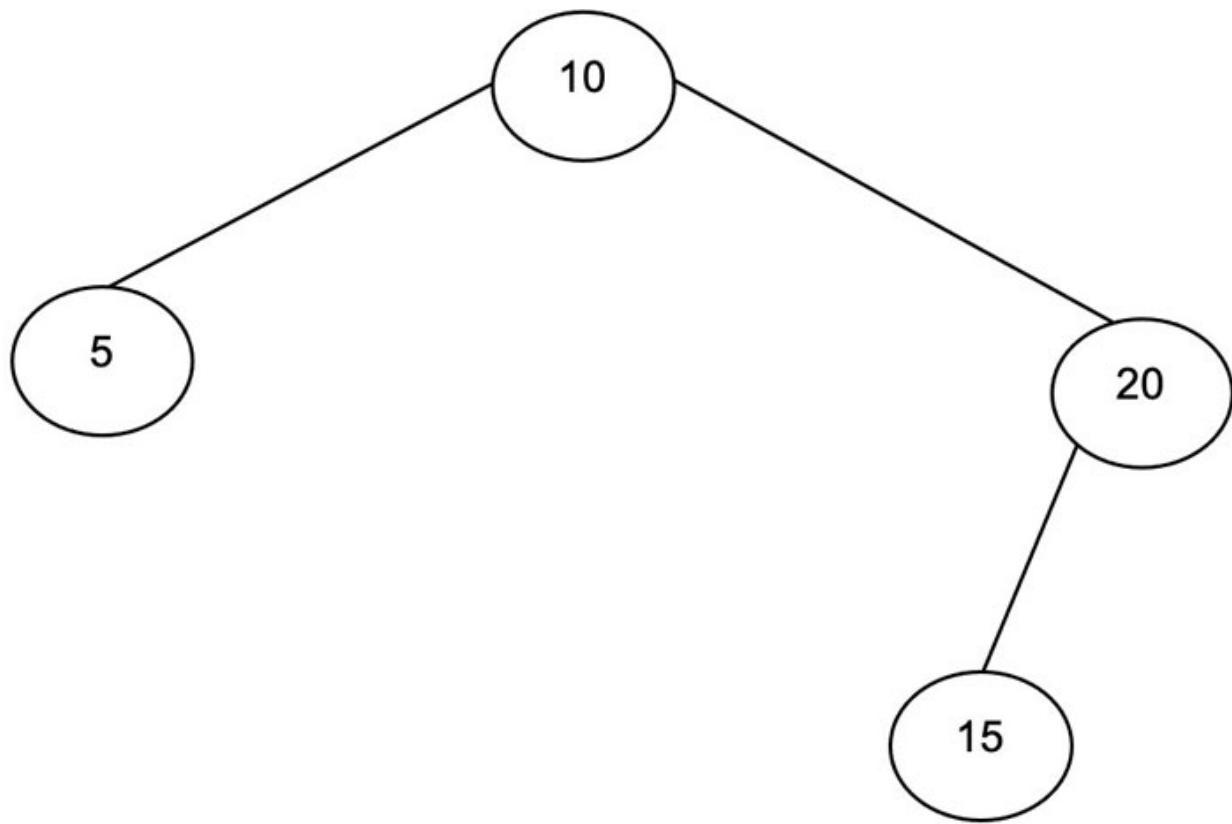
To understand these deletions, let us consider the following examples. In the first example, the R0 rotation restores the balance after deleting 50 from the tree.

### **Example 1: R0 rotation**

Note that initially, the tree shown in [figure 9.23 \(a\)](#) is balanced. The deletion of 50 makes it unbalanced. The balance can be restored by R0 rotation ([figure 9.23 \(b\)](#)).



**Figure 9.23 (a): Deleting 50**

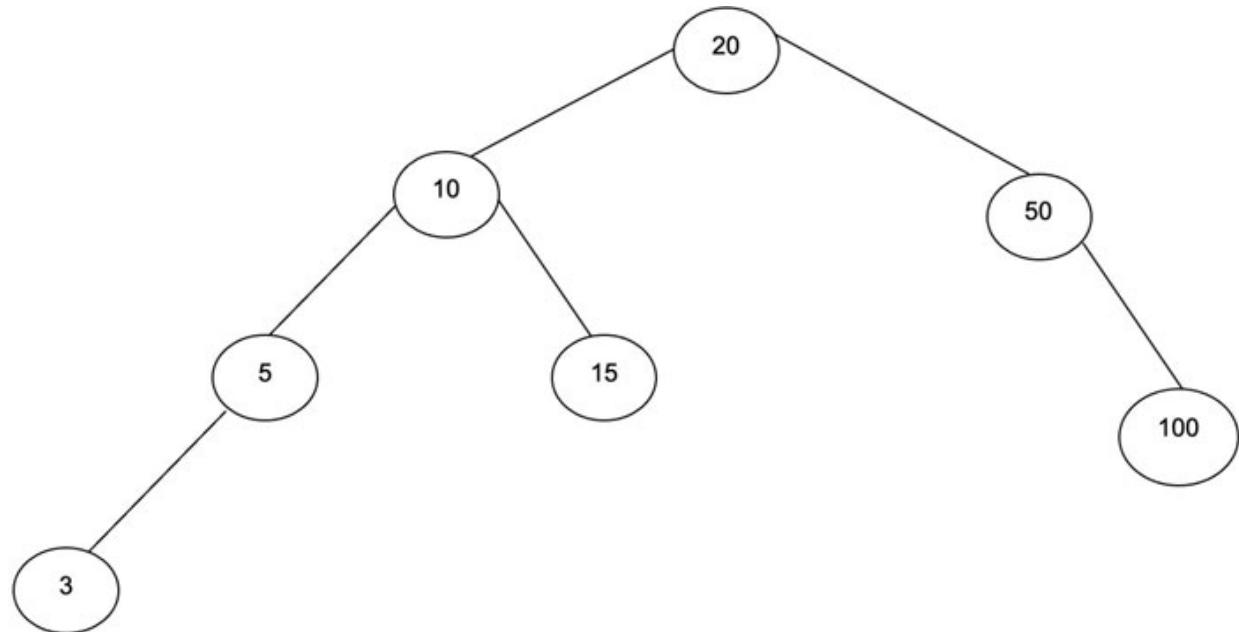


**Figure 9.23 (b): Balanced tree**

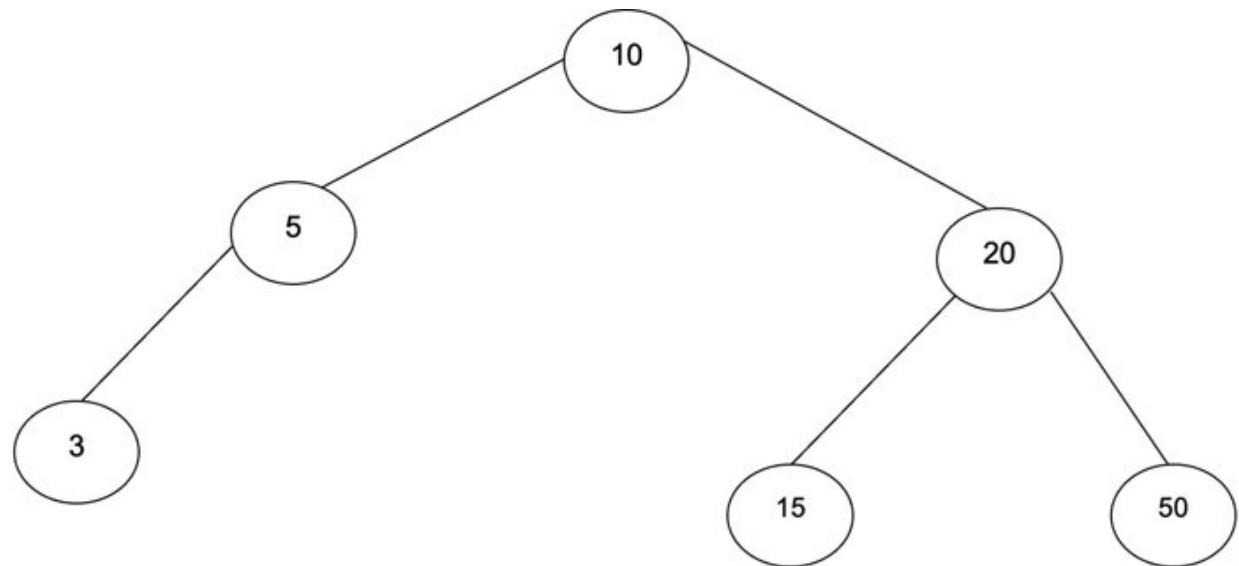
In the next example, the R1 rotation restores the balance after deleting 100 from the tree.

### Example 2: R1 rotation

Note that initially, the tree shown in [figure 9.24 \(a\)](#) is balanced. The deletion of 100 makes it unbalanced. The balance can be resorted by R1 rotation ([figure 9.24 \(b\)](#)).



*Figure 9.24 (a): Deleting 100*

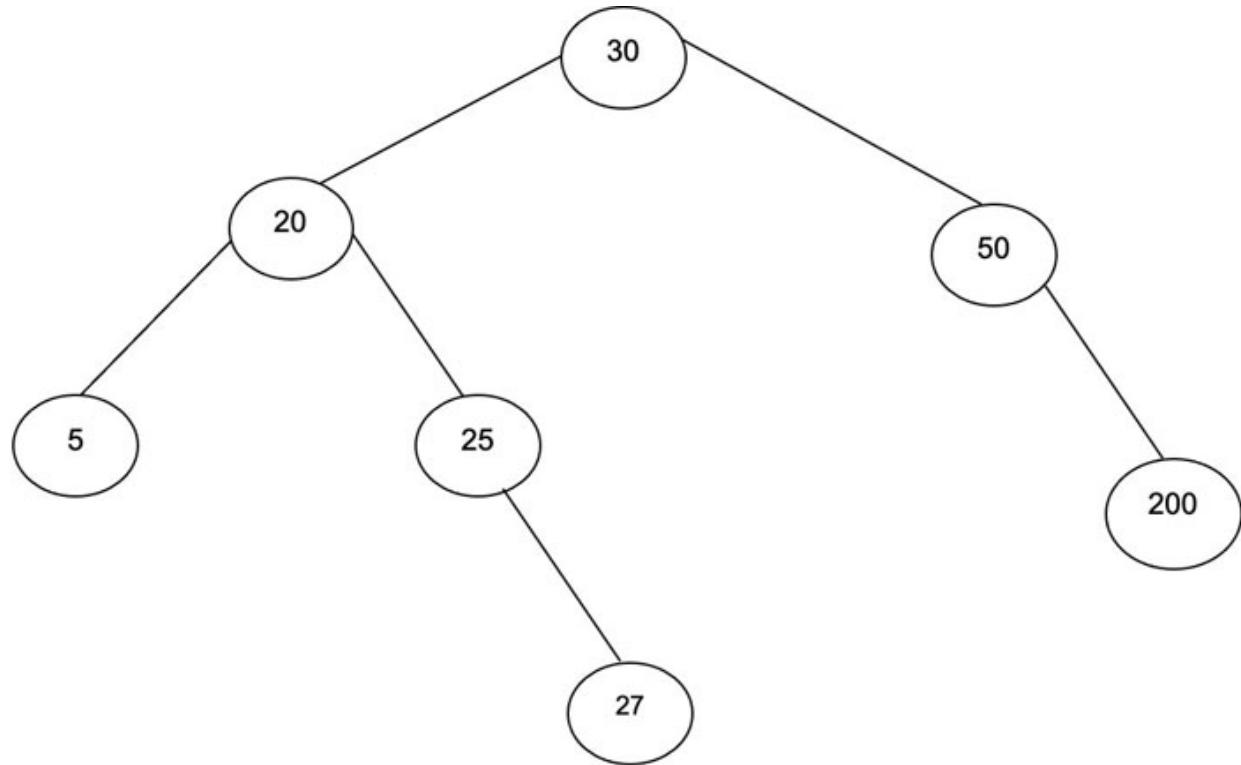


*Figure 9.24 (b): R1 rotation*

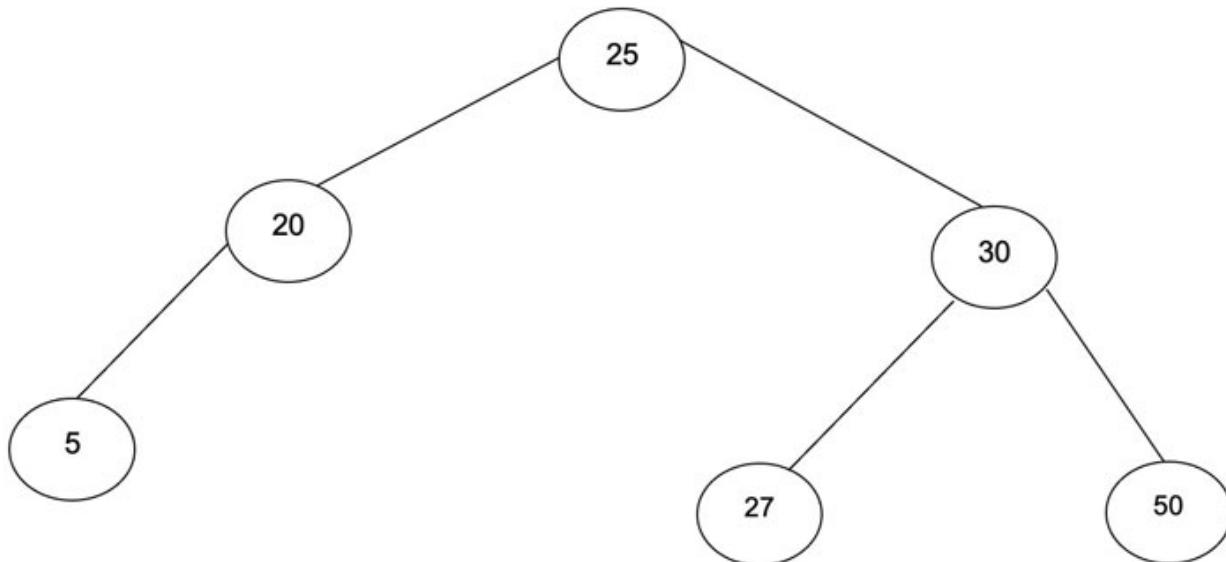
Example 3 depicts the R-1 rotation

### Example 3: R-1 Rotation

Note that initially, the tree shown in [figure 9.25 \(a\)](#) is balanced. The deletion of 200 makes it unbalanced. The balance can be resorted by R–1 rotation ([figure 9.25 \(b\)](#)).



*Figure 9.25 (a): Deleting 200*

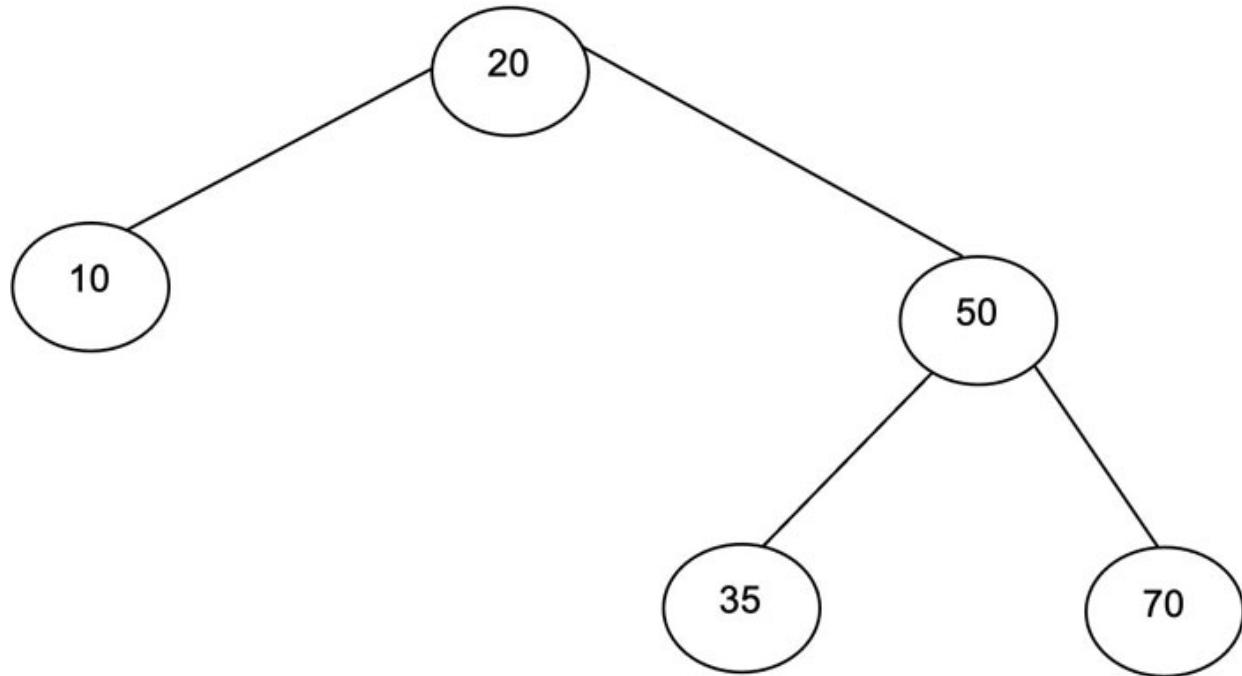


*Figure 9.25 (b): R–1 rotation*

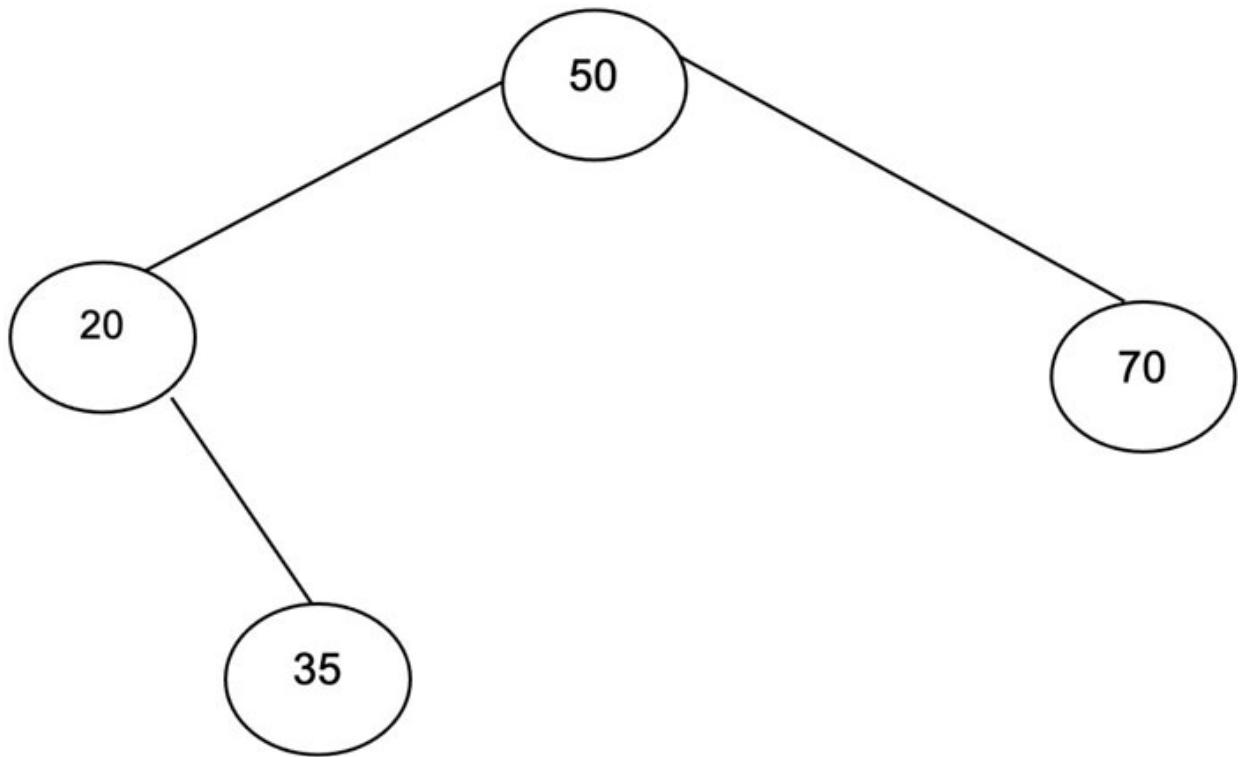
Having seen an example of each of R0, R1, and R–1 rotations, let us move to the L0, L1, and L–1 rotations.

#### Example 4: L0 Rotation

Note that initially, the tree shown in [figure 9.26 \(a\)](#) is balanced. Deletion of 10 makes it unbalanced. The balance can be resorted by L0 rotation ([figure 9.26 \(b\)](#)).



*Figure 9.26 (a): Deleting 10*

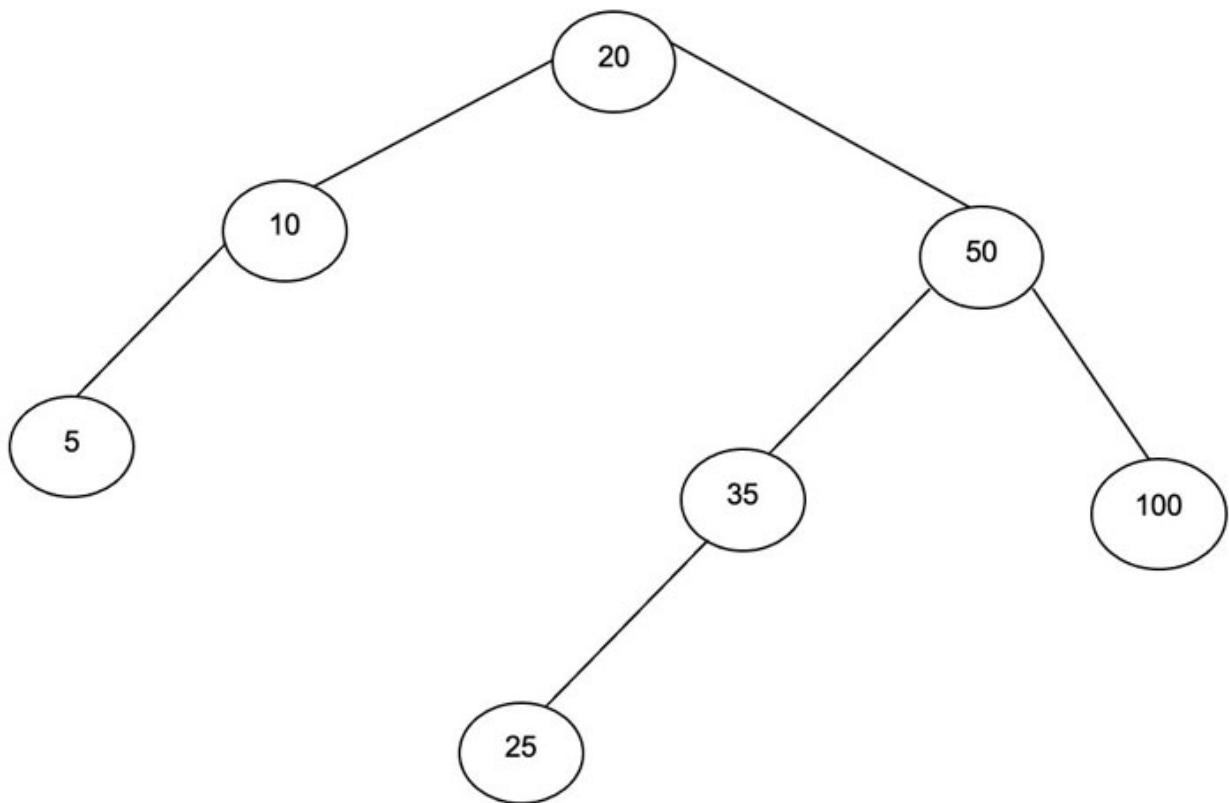


*Figure 9.26 (b): Balanced resorting using L0 rotation*

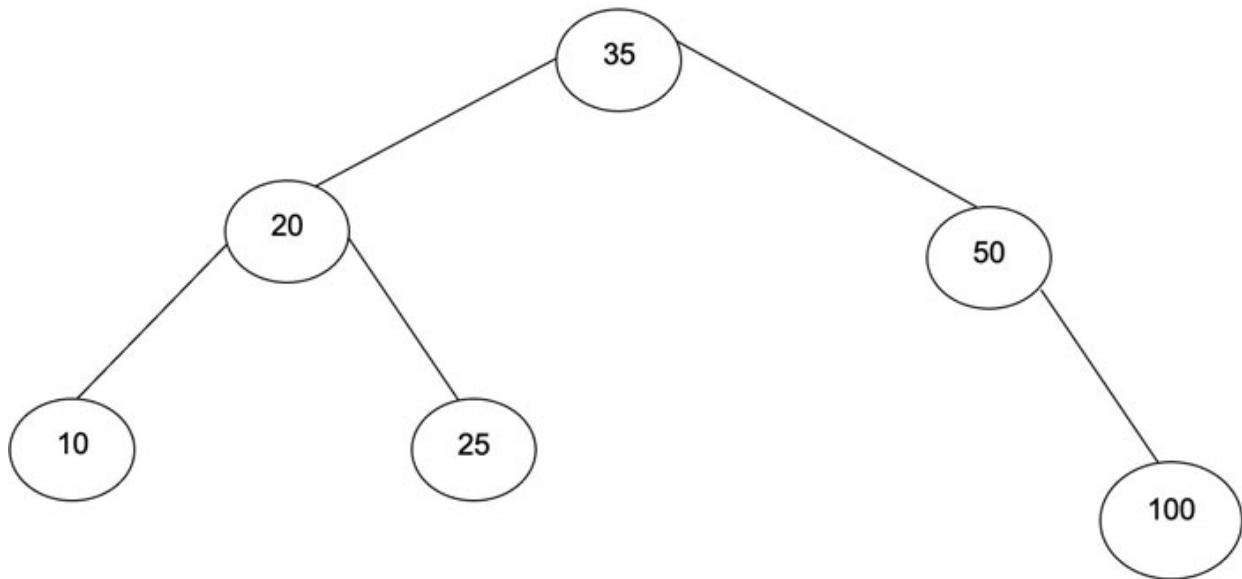
The next example shows L1 rotation.

### **Example 5: L1 Rotation**

Note that initially, the tree shown in [figure 9.27 \(a\)](#) is balanced. Deletion of 5 makes it unbalanced. The balance can be restored by L1 rotation ([figure 9.27 \(b\)](#)).



*Figure 9.27 (a): Deleting 5*

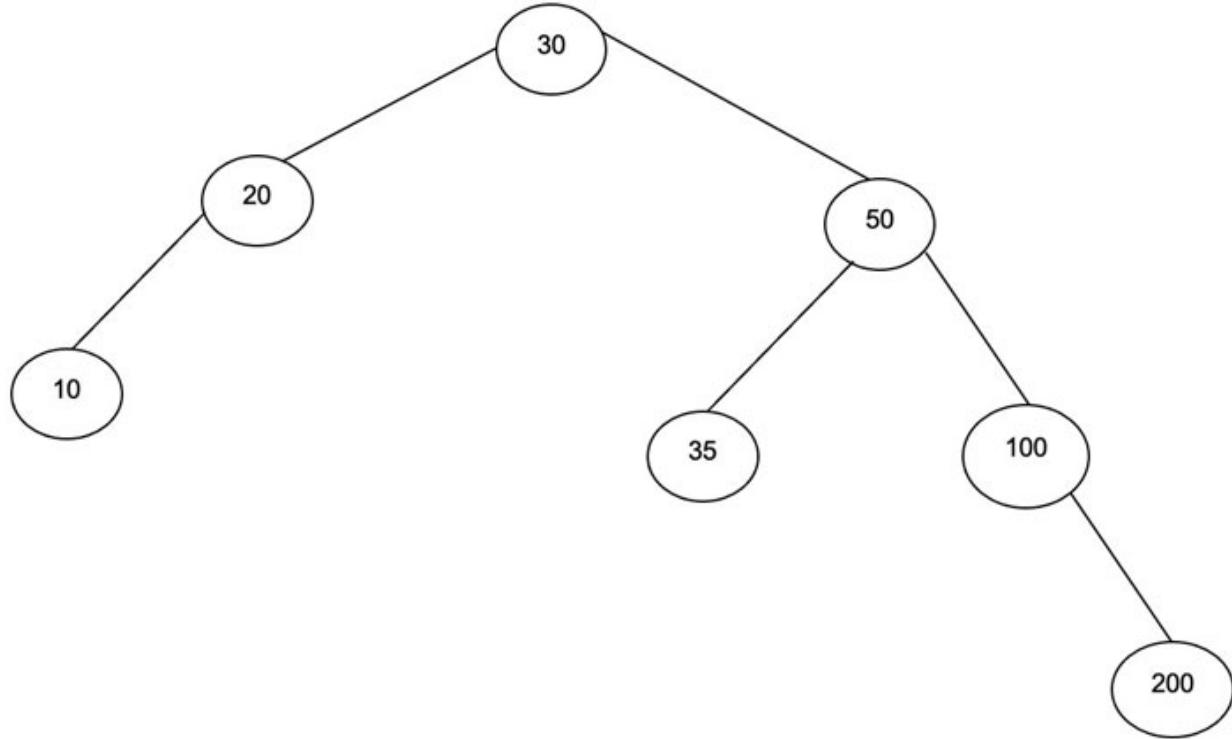


*Figure 9.27 (b): Balance resorted using L1 rotation*

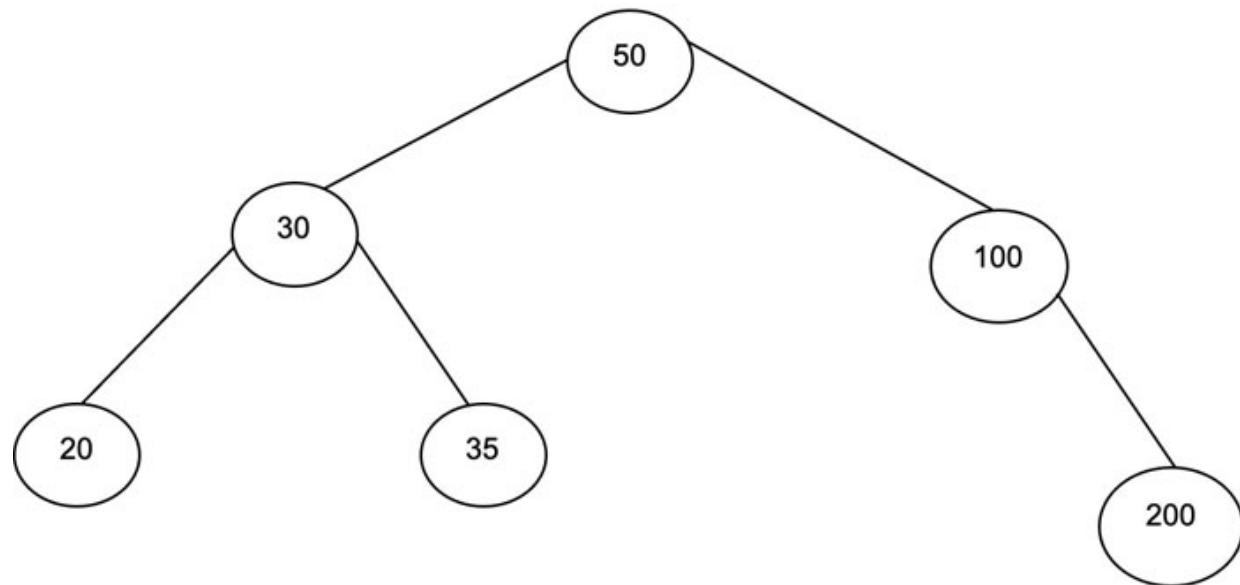
Example 6 shows an example of L-1 rotation.

### **Example 6: L-1 Rotation**

Note that initially, the tree shown in [figure 9.28 \(a\)](#) is balanced. Deletion of 10 makes it unbalanced. The balance can be resorted by L–1 rotation ([figure 9.28 \(b\)](#)).



*Figure 9.28 (a): Deleting 10*

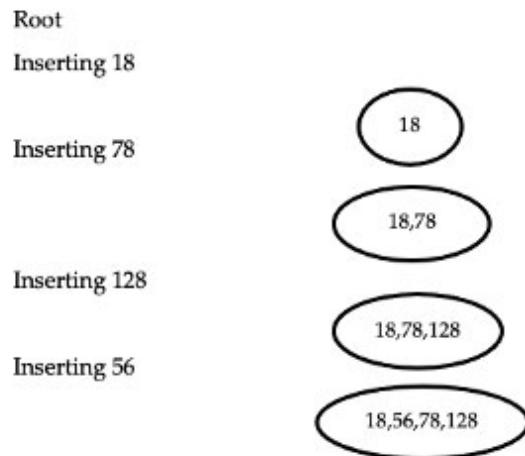


*Figure 9.28 (b): Balance restored using L–1 rotation*

## B Trees

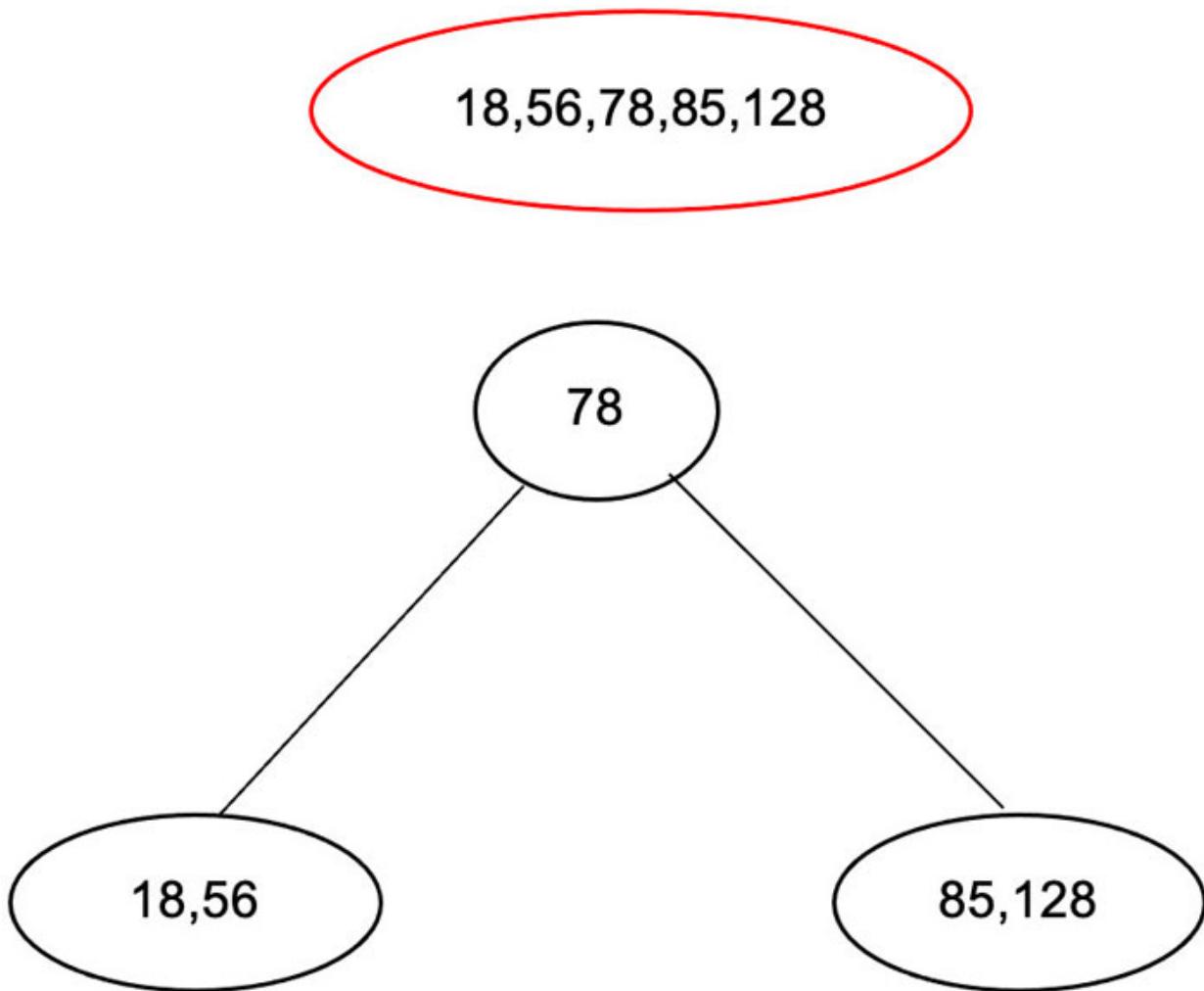
Let us now have a look at another search tree, B tree. A node in a B tree can have  $M$  items and a maximum of  $(M+1)$  children. Therefore, there can be a maximum of  $(M+1)$  pointers pointing to its children. The insertion in these trees is interesting. Initially, there is a root, and we start putting elements in the root till it has  $(M+1)$  items, after which it splits from the median, creating two children having  $M/2$  nodes each.

As an illustration, consider a B tree having  $M=4$ . On inserting items: 18, 78, 128, and 56 the root gets filled, as shown in [figure 9.29](#):



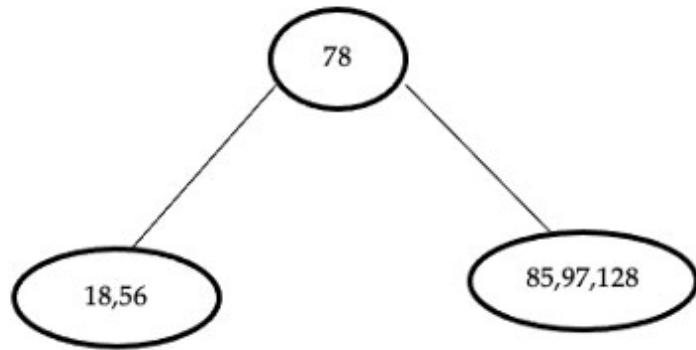
*Figure 9.29: Inserting 4 items in a B tree having  $M=4$*

On inserting 85, the number of items in the root exceeds the limit. At this point, the node splits into three parts. The first part will have  $M/2$  items, the second will have the item at the median, and the third will have  $M/2$  elements after the median, as shown in [figure 9.30](#).

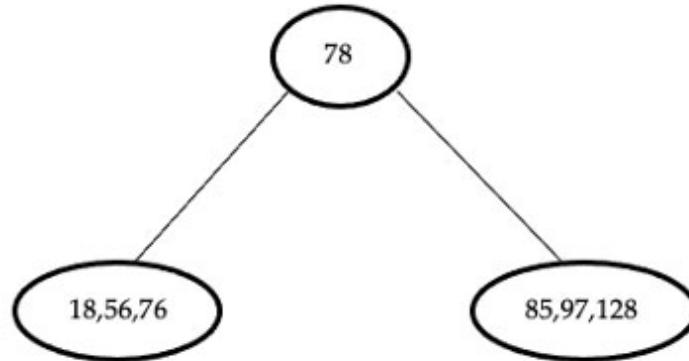


**Figure 9.30:** The root of the B tree with  $M=4$  splits on inserting 5 elements

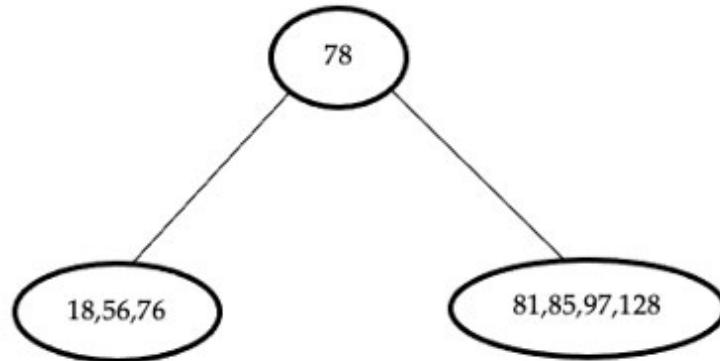
Now, inserting the subsequent elements in the B tree will follow the same procedure as a BST. In the B tree shown in [figure 9.31 \(a\)](#), 97 will go to the right child of the node as shown in [figure 9.31\(a\)](#), 76 to the left ([figure 9.31 \(b\)](#)), 81 to the right ([figure 9.31 \(c\)](#)), and 92 to the right.



*Figure 9.31 (a): Inserting 97*



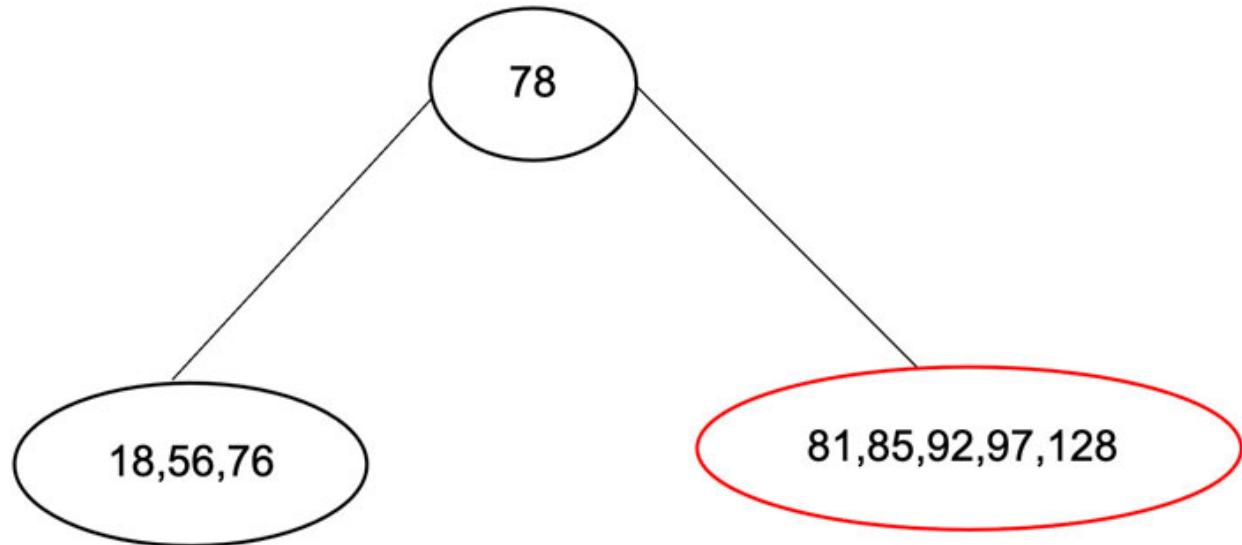
*Figure 9.31 (b): Inserting 76*



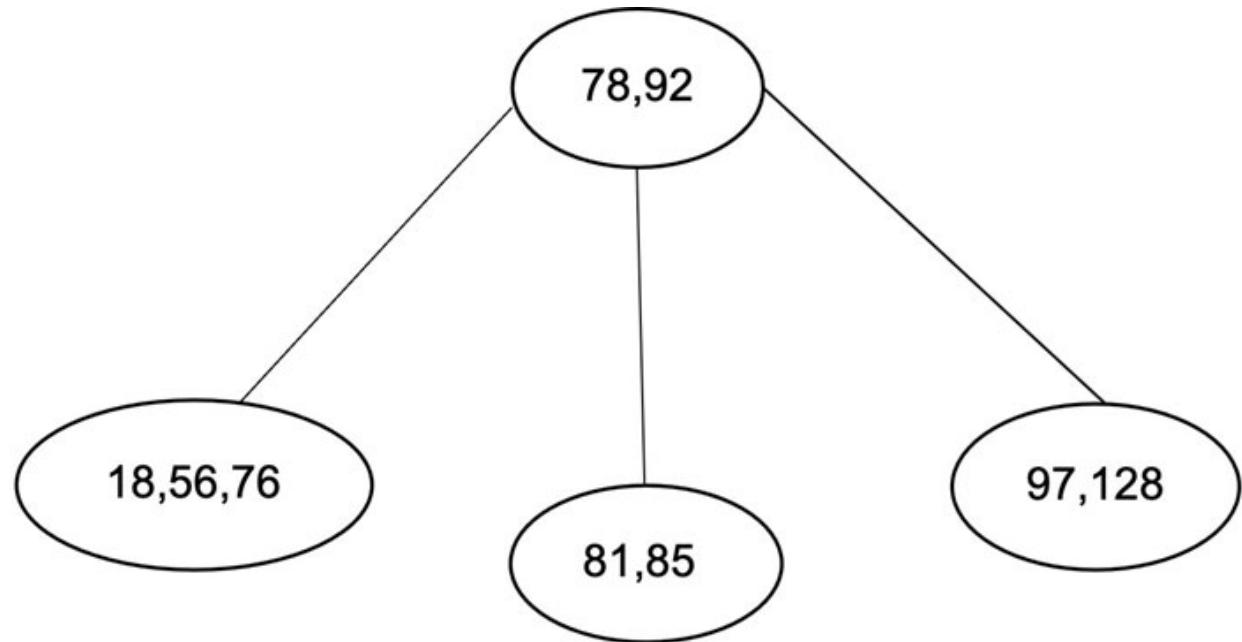
*Figure 9.31 (c): Inserting 81*

On inserting 92, however, the right child will have five items ([Figure 9.32 \(g\)](#)), and hence, it splits. This time, its parent has just one item. So, the item

at the median of the node that splits goes to the parent, and two new nodes are created, as shown in [figure 9.32 \(b\)](#):

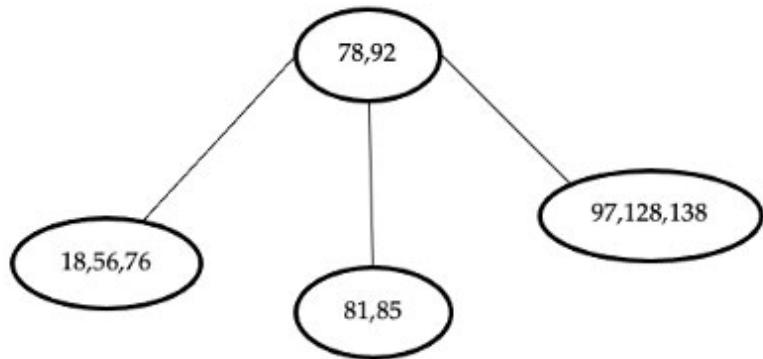


*Figure 9.32 (a): Inserting 92*



*Figure 9.32 (b): The right child of the root splits*

The insertion of 138, 90, and 111 is done in the usual manner ([figures 9.33 \(a\)](#) to [9.33 \(c\)](#)). However, inserting 120 again lands us in a problem:



**Figure 9.33 (a): Inserting 138**

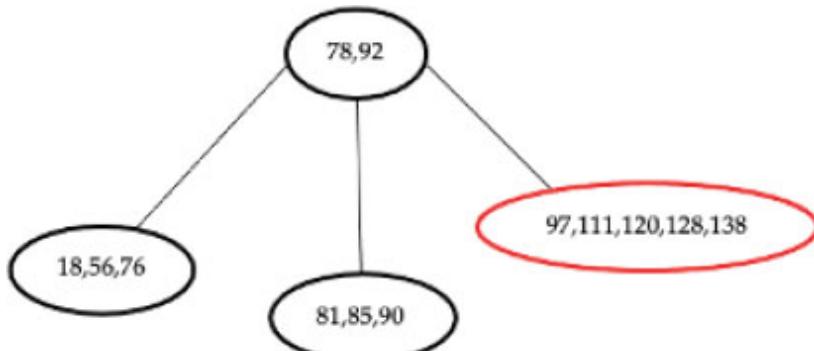


**Figure 9.33 (b): Inserting 90**

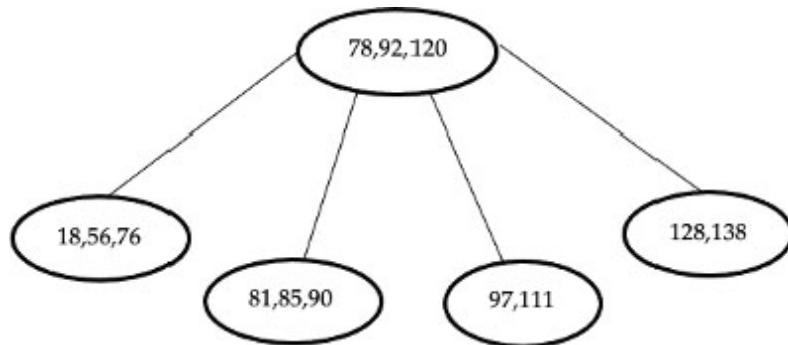


**Figure 9.33 (c): Inserting 111**

On inserting 120, the right-most child of the root will have five elements ([figure 9.34 \(a\)](#)), and hence, it splits from the median. The item at the median goes to the parent, as shown in [figure 9.34 \(b\)](#).



**Figure 9.34 (a):** Inserting 120 in a B tree having  $M=4$

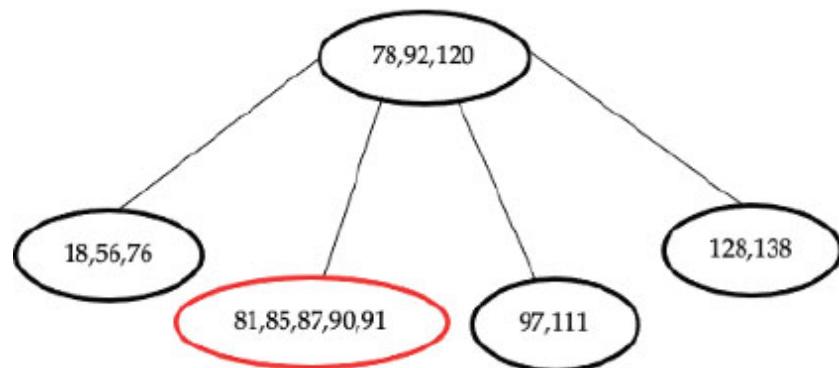


**Figure 9.34 (b):** The node splits around 120

The reader is requested to have a look at [figures 9.35\(a\) to 9.35\(g\)](#). These insertions follow the same rules as explained in the preceding discussion:



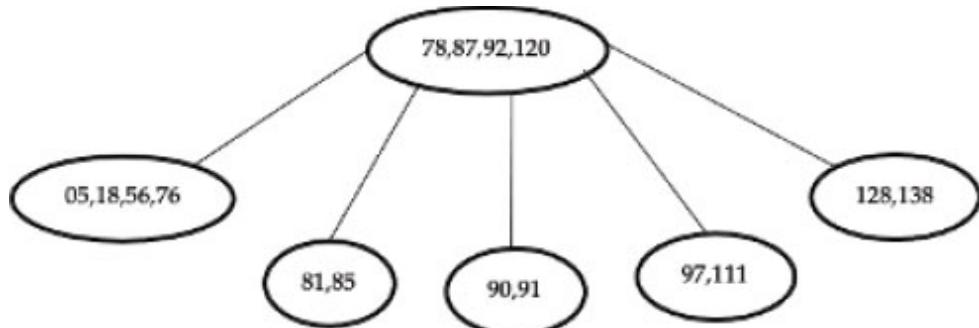
**Figure 9.35 (a):** Inserting 8



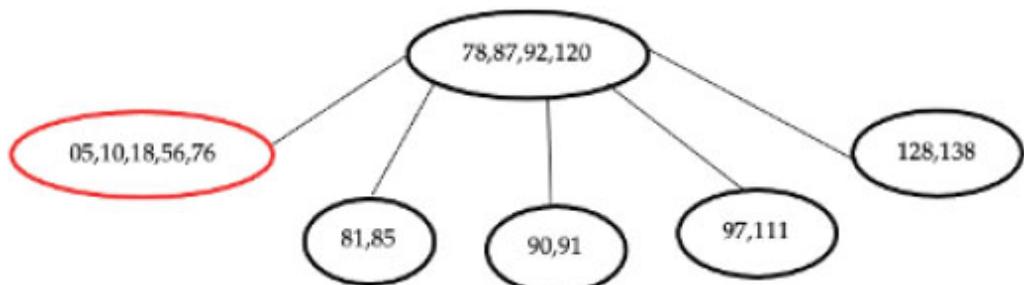
**Figure 9.35 (b):** Inserting 91



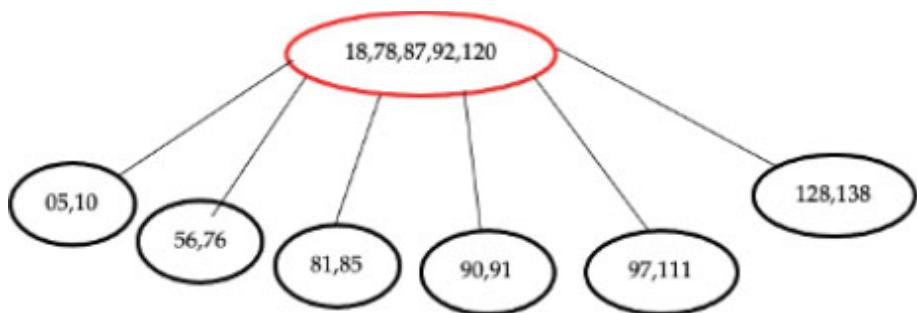
**Figure 9.35 (c):** After inserting 91, the node splits



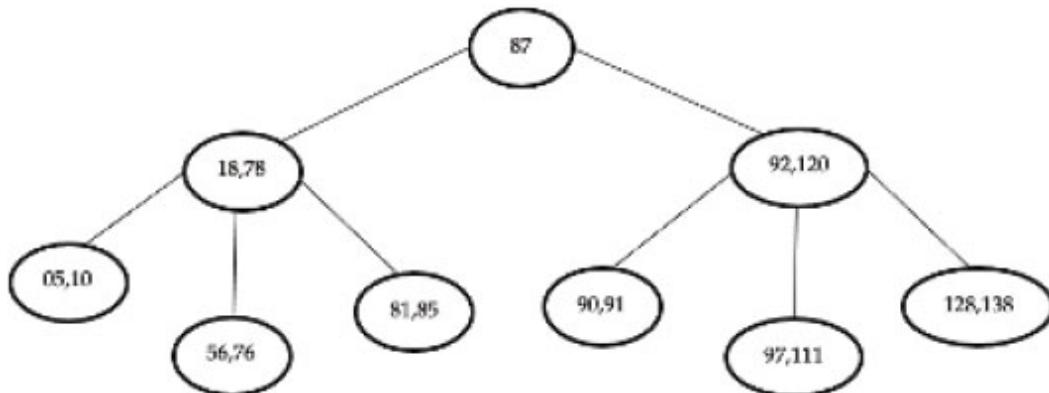
**Figure 9.35 (d):** Inserting 05



**Figure 9.35 (e):** Inserting 10



*Figure 9.35 (f): The node splits*



*Figure 9.35 (g): Final B tree*

The user can refer to the Web resources for the Jupyter notebook containing the implementation of B Tree.

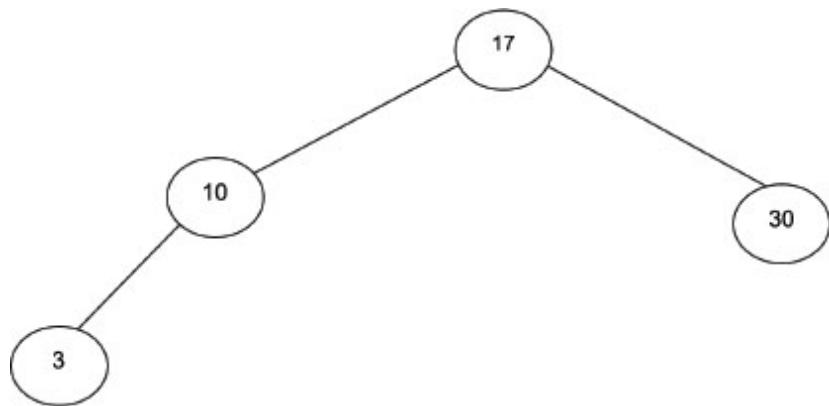
## Conclusion

This chapter introduced the reader to AVL trees and B Trees. The insertion and deletion in these trees have been discussed in detail in this chapter. The Web resources contain the Jupyter notebook containing the implementation of these trees. The reader will be able to carry out basic operations and implement these trees after reading this chapter.

The upcoming chapter discusses another important data structure called Graphs. The trees and graphs are the foundation stones of competitive programming. Let us now hit the exercises.

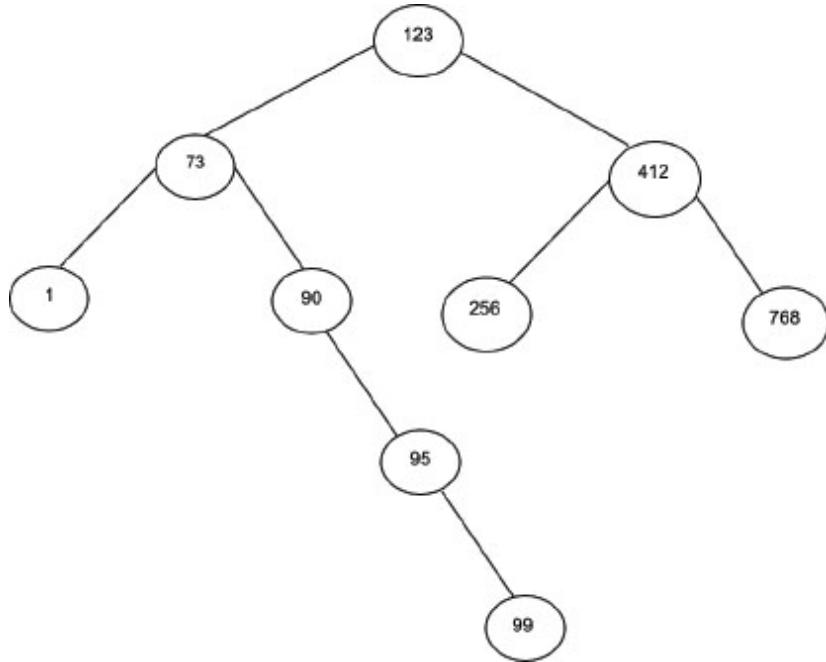
## Multiple choice questions

1. What is the BF of the node having value 17 ([figure 9.36](#))?



**Figure 9.36:** Figure for Q1

- a. 0
  - b. 1
  - c. 2
  - d. -1
2. In the preceding question, which rotation is needed to make the tree balanced?
- a. LL
  - b. LR
  - c. RL
  - d. RR
3. In the following tree, which rotation is needed to make the tree balanced ([figure 9.37](#))?



**Figure 9.37:** Figure for Q3

- a. LL
  - b. LR
  - c. RL
  - d. RR
4. Which of the following is not a method for deletion from an AVL tree?
- a. L0
  - b. L1
  - c. L-1
  - d. L2
5. Which of the following is not a method for deletion from an AVL tree?
- a. R0
  - b. L1
  - c. R-1
  - d. R3

**6. How many elements can a B tree of order 5 have in each node?**

- a. 5
- b. 6
- c. 7
- d. 4

**7. In the preceding question, what is the maximum number of children a node can have?**

- a. 5
- b. 4
- c. 6
- d. None of the above

**8. In a B Tree, all the leaves are at the same level?**

- a. True
- b. False

**9. Which of the following is a solution of an unbalanced tree?**

- a. AVL
- b. B Tree
- c. Both
- d. None of the above

**10. Which of the following values of BF can a node in an AVL tree have?**

- a. 1
- b. 2
- c. 0
- d. -1

## Theory

1. Explain the problems in BSTs? Give an example to show that a BST created from an increasing or decreasing sequence will be skewed.
2. Write an algorithm for the following:
  - a. LL
  - b. RR
  - c. LR
  - d. RL
3. For each algorithm in the preceding question, find the complexity.
4. Write an algorithm for each of the following:
  - a. L0
  - b. L1
  - c. L-1
  - d. R0
  - e. R1
  - f. R-1
5. For each of the preceding algorithms find the complexity.
6. Write an algorithm for inserting an element in a B Tree.
7. Define an AVL tree. How does it help to handle the problems in BSTs?
8. Define B tree. How does it help to handle the problems in BSTs?

## Numericals

1. Create a BST from the following sequence:  
2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
2. How much time does it take, on an average, to delete an element from the preceding tree?
3. Now create an AVL from the preceding values.
4. In the preceding question, insert the following and show the tree after each step:
  - a. 2048

b. 4096

c. 1

d. 8192

5. From the AVL tree created in the preceding question, delete the following:

a. 8

b. 512

c. 4096

d. 1

6. Now create a sequence. Create a B tree and show the tree at each step.

7. Create a B Tree from the following sequence

3, 8, 1, 78, 623, 24, 90, 23, 89, 12, 7, 52, 167, 233, 190.

8. From the tree created in the above question, delete 52 and 167.

## **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 10

## Priority Queues

We have already studied Stacks, which are linear data structures that follow the principle of Last In First Out, and Queues, which are linear data structures that follow the principle of First In First Out. From a Stack, the element at the top can be popped in O(1) time, whereas from a Queue, the element at the front can be popped in O(1) time. This chapter introduces a data structure that can pop the maximum element in it in O(1) time (or, for that matter, the minimum element). In general, the elements from this data structure can be popped based on the priority of the elements. These are called priority queues. Heap, an implementation of a Priority Queue, is an immensely useful data structure. This chapter presents the algorithms to deal with heaps. This will act as the foundation for the rest of the chapters and will help you to solve many interesting problems.

### Structure

In this chapter, we will cover the following topics:

- Introduction to priority queues and heaps
- Insertion in a heap
- Deletion from a heap
- Heapsort
- Problems related to heap

### Objectives

This chapter introduces the heap data structure to the reader. Note that a heap is a type of priority Queue. Its unique structure allows the removal of the maximum or minimum element only. After reading this chapter, the reader will be able to carry out insertion and deletion in a heap. The reader will also be able to implement the heap sort and solve some interesting problems

using heaps. So, let us discover the mesmerizing concept and equip ourselves with this potent data structure.

## Introduction to priority queues

The priority queue is a data structure in which elements can be returned in order of their priority. That is, the element with maximum priority can be returned in  $O(1)$  time. This priority can be having the maximum or the minimum element in constant time. If we wish to extract the maximum element each time the delete operation is invoked, then the data structure should support the following operations:

- **Insert(D, key):** This operation inserts the key into the set D
- **Max(D):** This operation returns the maximum element from the set D
- **Delete\_Max(D):** This operation should return the maximum element from D and remove it from the set D.

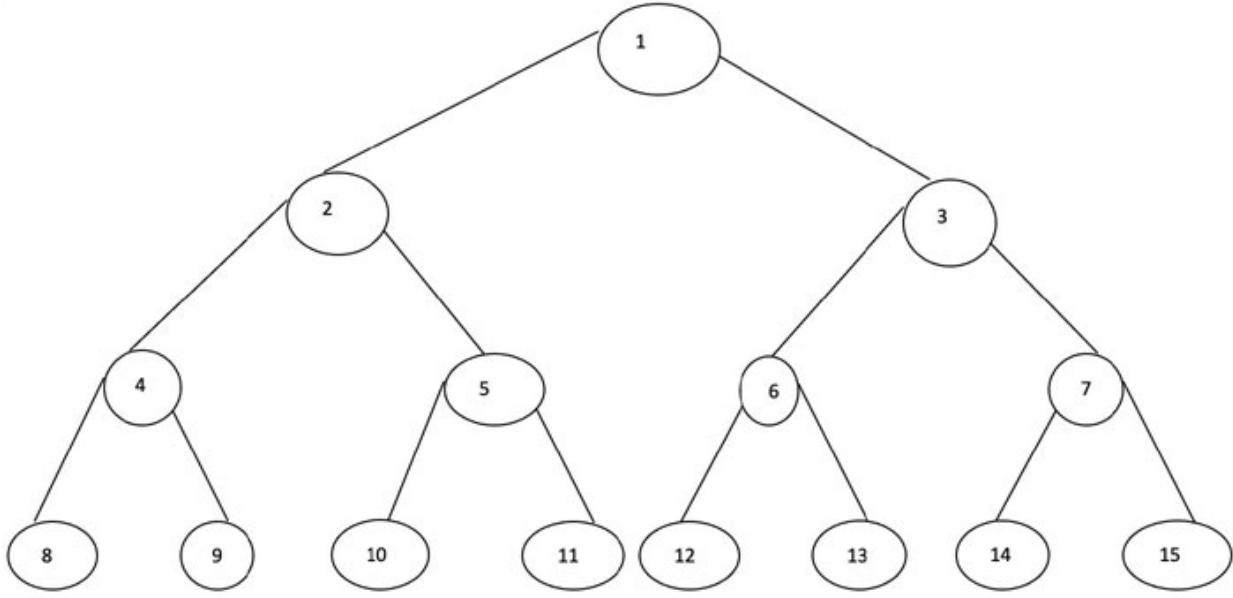
In case, the minimum element needs to be returned each time the delete operation is called. The data structure should support the following operations:

- **Insert(D, key):** This operation inserts the key into the set D
- **Min(D):** This operation returns the minimum element from the set D
- **Delete\_Min(D):** This operation returns the minimum element from D and removes it from the set D.

This chapter deals with heap, which is an implementation of a priority queue.

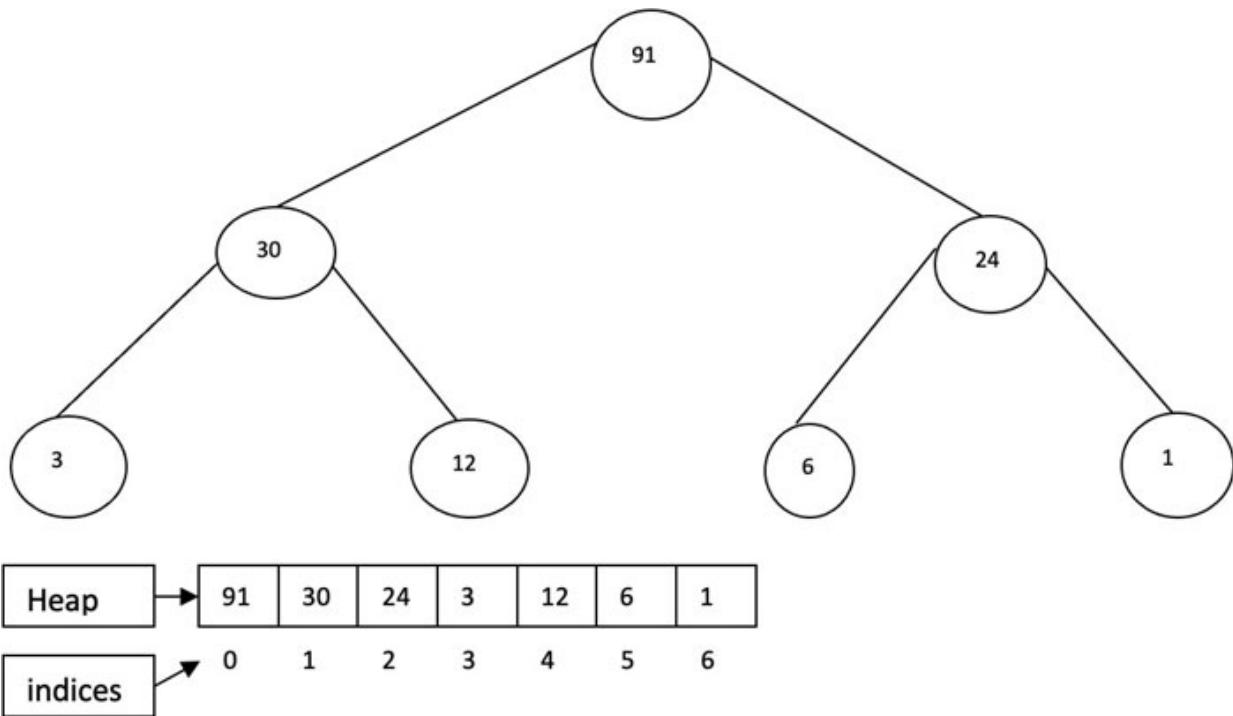
## Structure of heap

A heap is a complete binary tree. In this data structure, the elements are inserted in the level order. The numbers at the nodes in [figure 10.1](#) represent the order in which the elements should be inserted:



**Figure 10.1:** Elements in a heap are inserted in level-order

Since it is a complete binary tree, the elements of a heap can be stored in an array. That is, the root is stored at the  $0^{th}$  index. If a particular node is stored at the  $i^{th}$  index, its left child is placed at the  $2i + 1^{th}$  index, and the right child is placed at the  $2i + 2^{nd}$  index. [Figure 10.2](#) shows a heap and the corresponding array. Note that, in such implementation, the parent of the element at the  $i^{th}$  index is located at the  $((i-1)/2)^{th}$  index.



**Figure 10.2:** A heap can be stored using an array

A heap can be classified into two categories: Max-heap and Min-heap. A heap must satisfy the heap property, which requires the value at each node to be greater than either of its child, in the case of Max-heap. In the case of Min-heap, the value at each node should be lesser than either of its child. Note that it is markedly different from a Binary Search Tree, in which the left node is less than the root, whereas the right is greater than the root.

In the case of Max-heap, the greatest element is placed at the root. So, it can be returned in  $O(1)$  time. Likewise, in the case of Min-heap, the smallest element is placed at the root. Therefore, from a Min-heap, the smallest element can be returned in  $O(1)$  time.

Note that if a heap has  $2^n - 1$  elements, the last level contains  $2^{n-1}$  elements. The reader is expected to work out why in the case of Max-heap, the minimum element is one of these  $2^{n-1}$  elements. In the case of Min-heap, the maximum element is one of these elements.

## Operations

A heap supports the following operations:

- **Insert:** This operation inserts the key at the appropriate location to maintain the sanctity of the heap.
- **Delete:** This operation deletes the root and re-heapifies the rest of the element.
- **Peak:** This operation returns the element at the root but does not delete it.

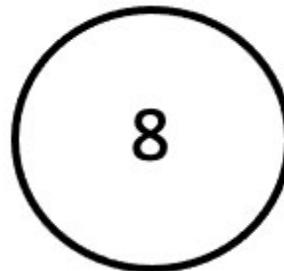
The next section explains the insertion in a Max-heap.

## Inserting an element in a heap

This section illustrates the procedure of the formation of a heap out of the given set of elements. Note that the elements are inserted in the level order, and in case an element violates the Max-heap property, the following procedure corrects the violation by trickling the element down the tree [Corman].

To understand the formation of a heap, let us have a look at the creation of a heap from the elements of a set {8, 18, 6, 89, 32, 9, 4}.

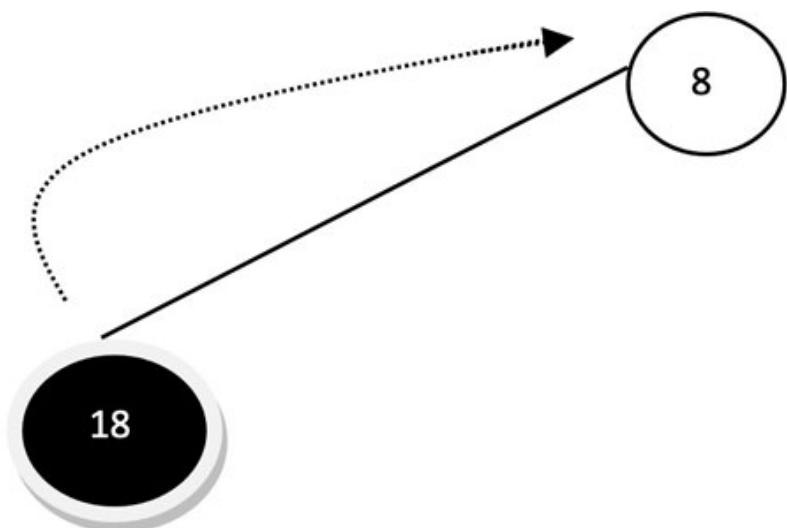
The first element becomes the root of the heap. That is, 8 is placed at the root of the heap ([figure 10.3](#)):



*Figure 10.3: Inserting the first element in the heap*

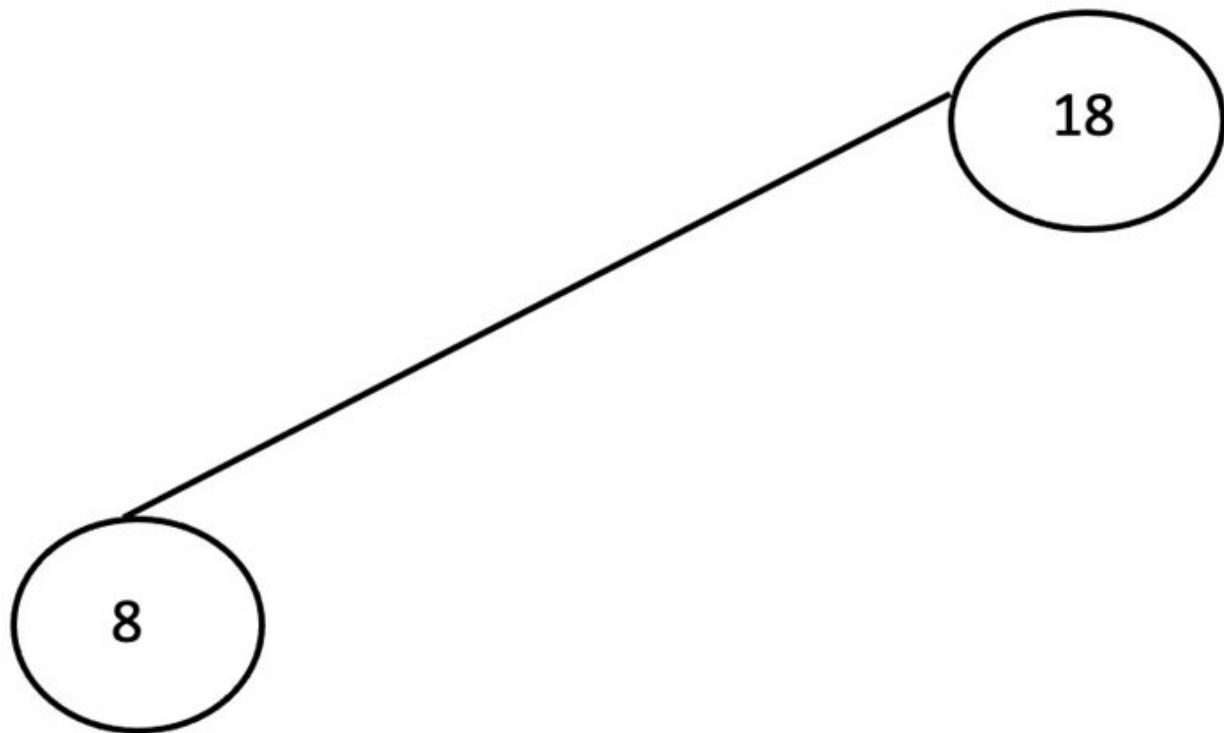
The next element becomes the left child of the root. In case it is greater than that at the root, the elements are swapped. The next element is 18, and because it is greater than 8, the two are swapped ([figure 10.4](#)):

The child is greater than the parent, therefore they are swapped.



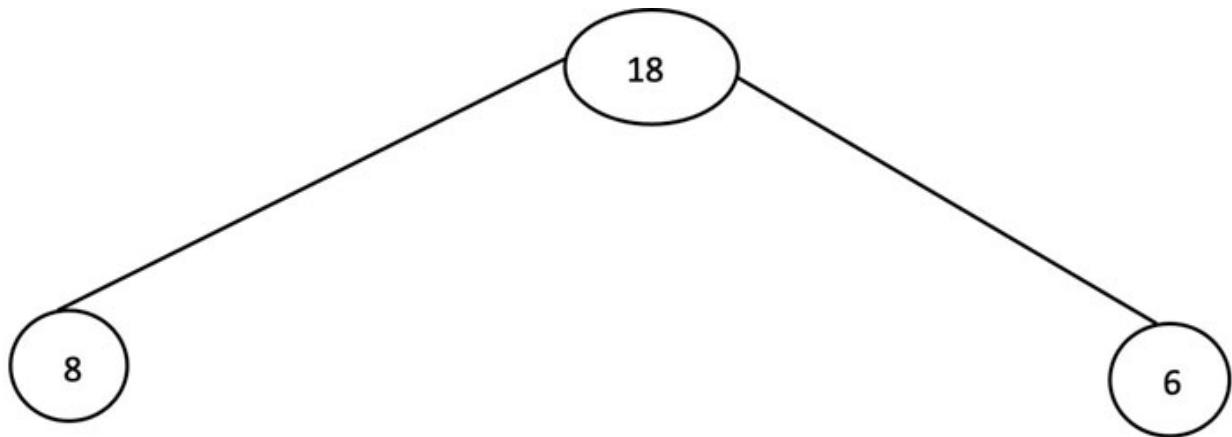
*Figure 10.4: Swapping 18 and 8*

Note that, in a heap, each child of a node is less than the node itself ([figure 10.5](#)):



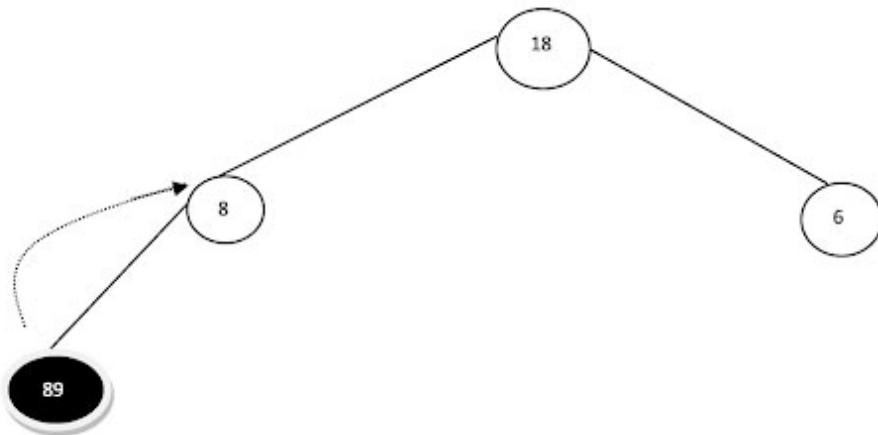
*Figure 10.5: Heap having two elements*

The next element becomes the right child of the root. In case it is greater than that at the root, the elements are swapped. The next element is 6; since it is less than that at the root, no swapping is required ([figure 10.6](#)):



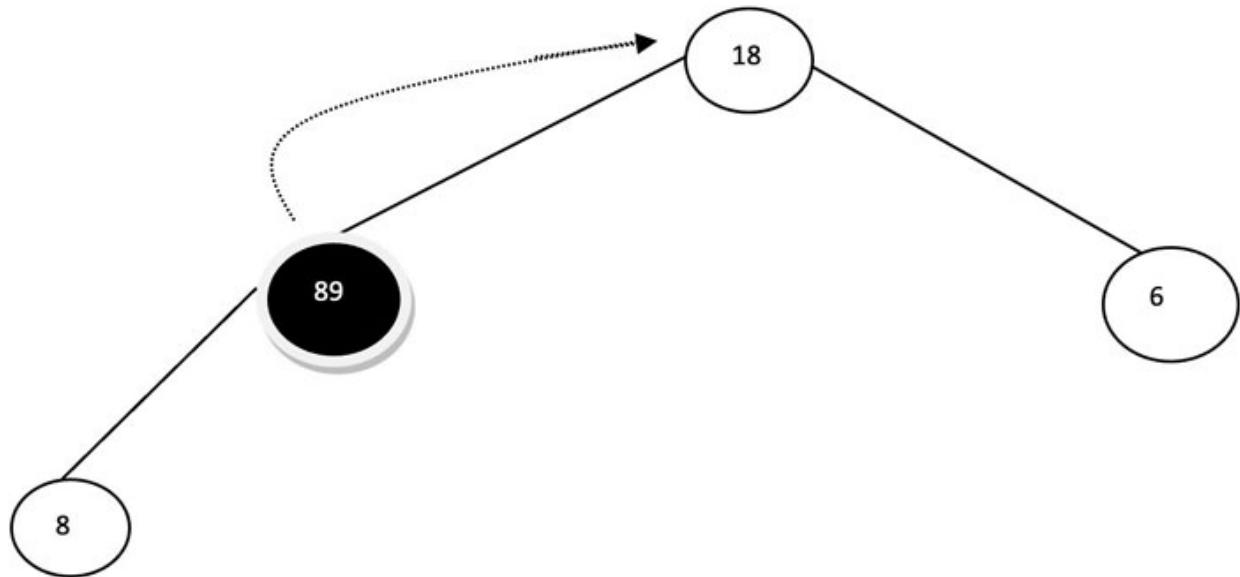
**Figure 10.6:** Inserting the third element in the heap

The next element will now become the left child of “8”. If it is greater than its parent, the values are swapped, and the process continues till the root. The next element is 89, and it is greater than 8; therefore, the two are swapped ([figure 10.7](#)):



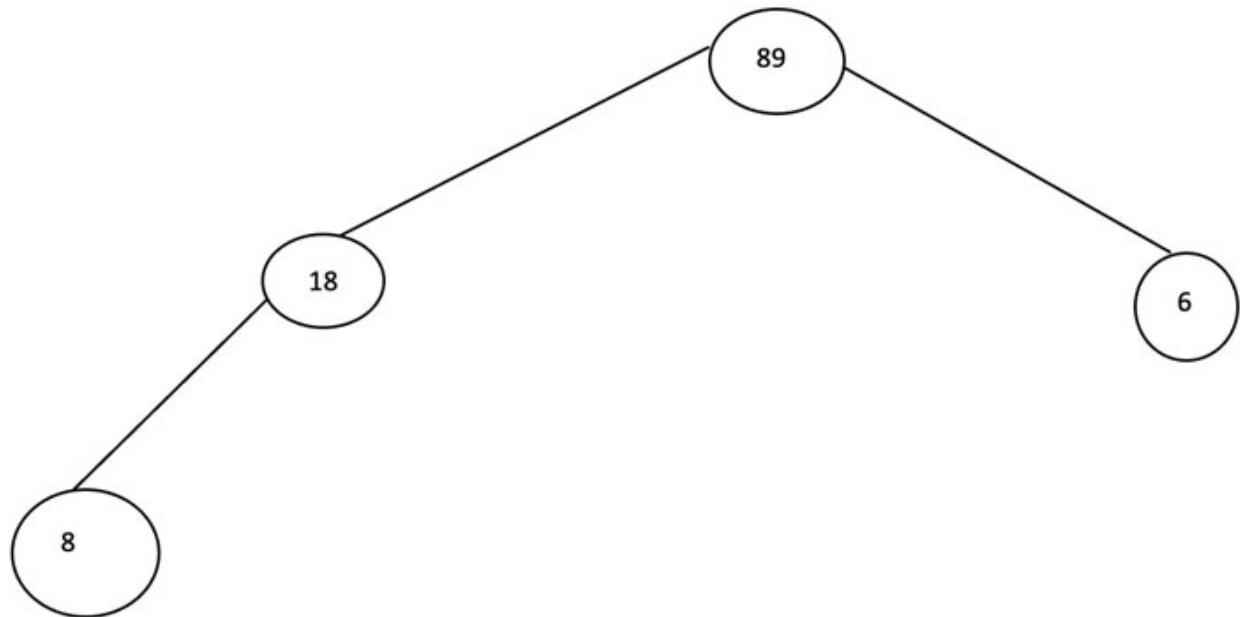
**Figure 10.7:** Inserting the fourth element in the heap

Note that still, the child of the root is greater than the value stored in it; therefore, the two would be swapped ([figure 10.8](#)):



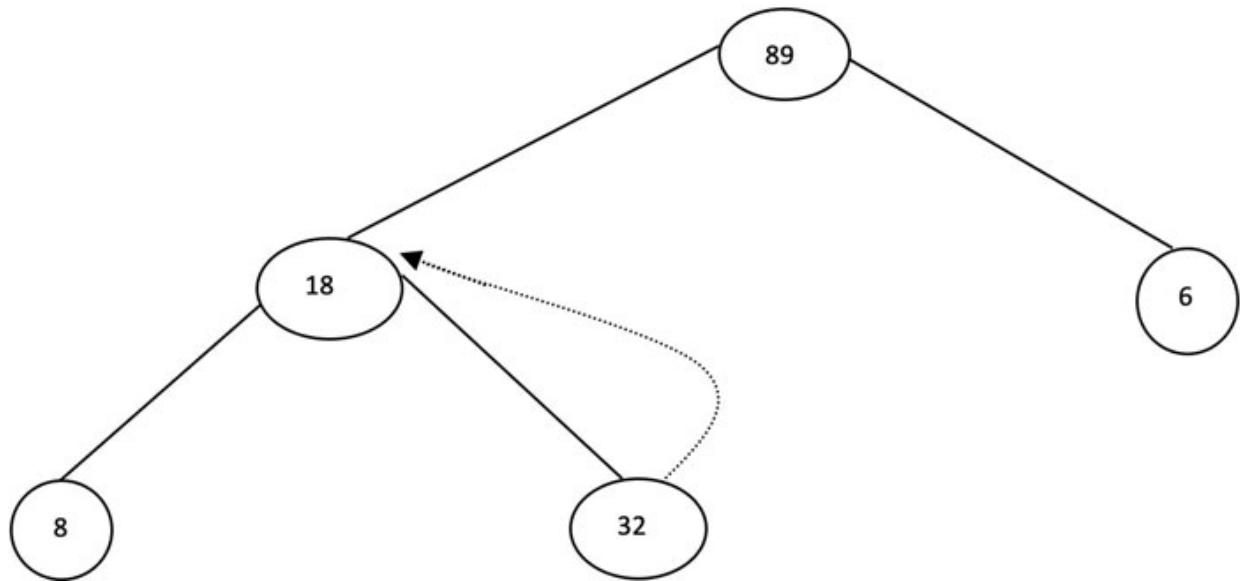
*Figure 10.8: Swapping 89 and 18*

The heap formed so far is shown in [figure 10.9](#):



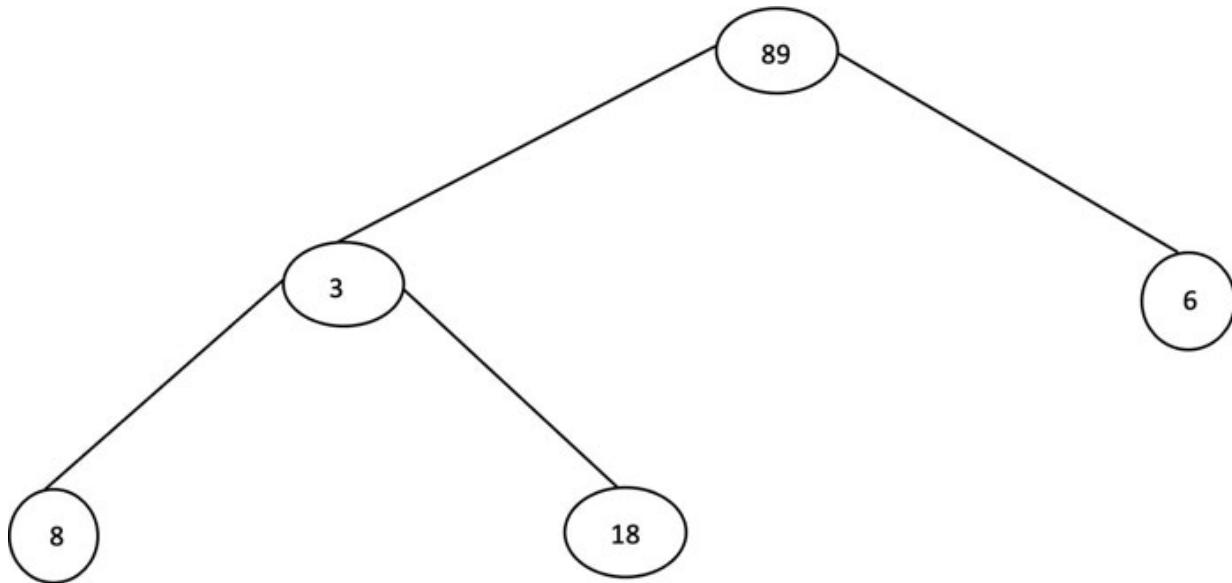
*Figure 10.9: Heap having four elements*

The next element becomes the right child of the node having value 18. Since the element being inserted is greater than 18, it is swapped with its parent ([figure 10.10](#)):



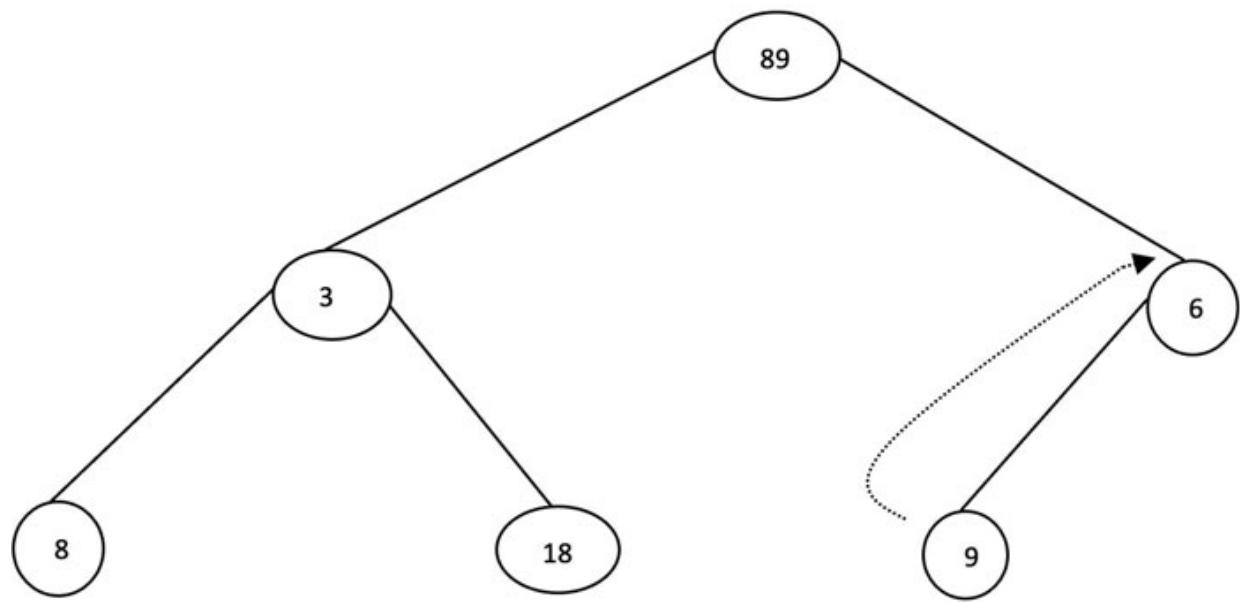
**Figure 10.10:** Inserting the fifth element in the heap

The heap so formed is shown in [figure 10.11](#):

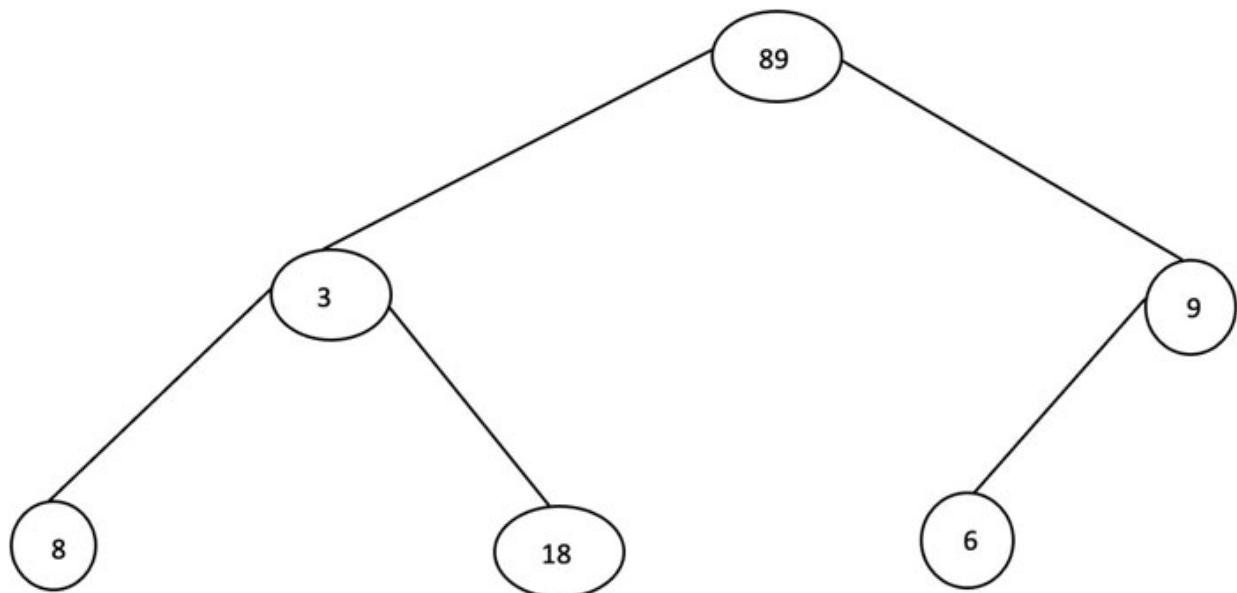


**Figure 10.11:** Heap having five elements

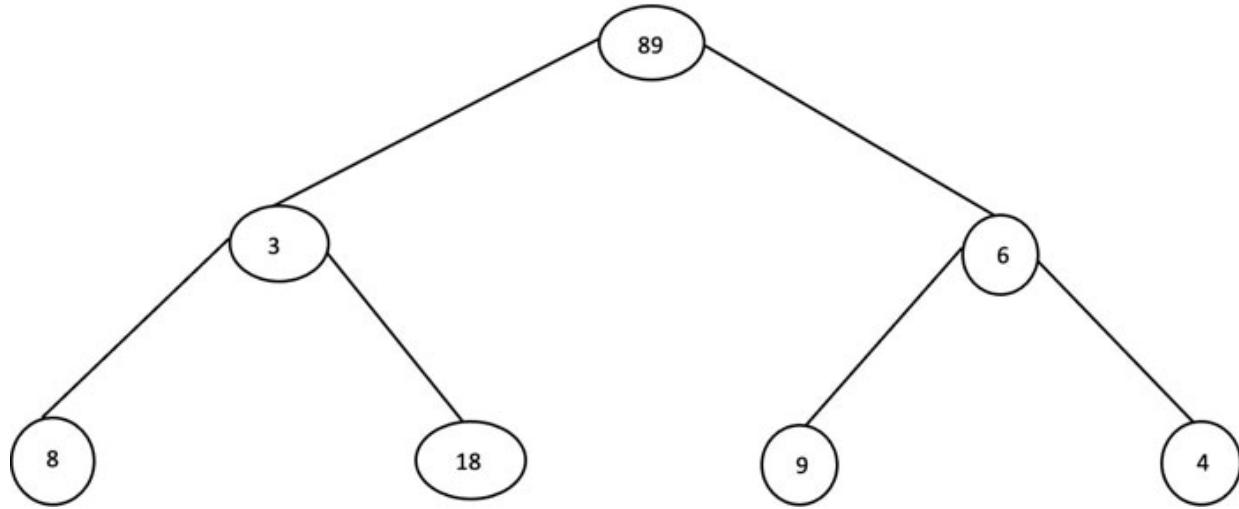
The following two figures show the insertion of 9 and 4 in this heap ([figures 10.12\(a\)](#) to [10.12\(c\)](#)):



**Figure 10.12(a): Inserting 9 in the heap**



**Figure 10.12(b): Heap having six elements**



*Figure 10.12(c): Final heap*

## Complexity

If a heap has  $N$  values, its height is,  $\lceil \log_2 N \rceil + 1$ , therefore, insertion requires at most  $O(\log_2 N)$  swaps. Therefore, the complexity of inserting an element in a heap is  $O(\log_2 N)$ . Note that in the following code, the heap list has been initialized to 0.

The following code inserts an element in a heap:

```

1. def insert(heap, val):
2.     if(heap[0] ==0):
3.         heap[0]=val
4.     else:
5.         index=0
6.         while(heap[index]!=0):
7.             index+=1
8.             heap[index]=val
9.             pos=index
10.            while(pos!=0):
11.                if(heap[(pos-1)//2]<heap[pos]):
12.                    temp=heap[pos]

```

```

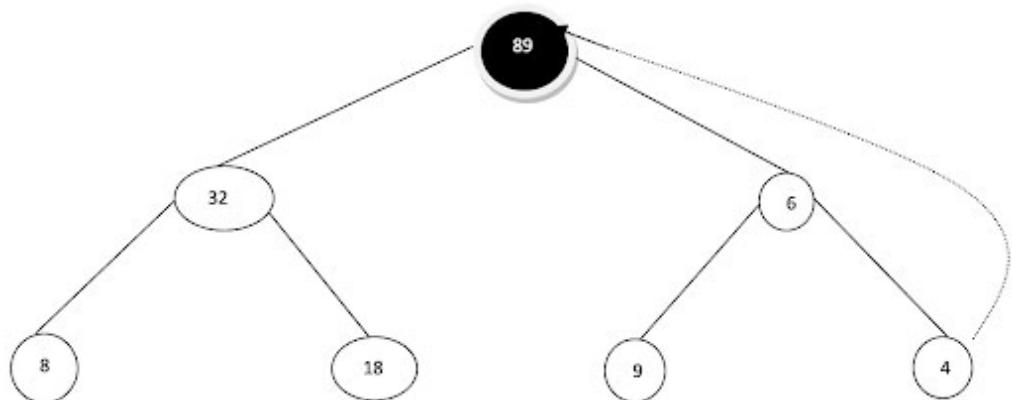
13.    heap[pos]=heap[(pos-1)//2]
14.    heap[(pos-1)//2]=temp
15.    pos=(pos-1)//2
16. else:
17.     break

```

Now, let us move to the deletion of an element from a heap.

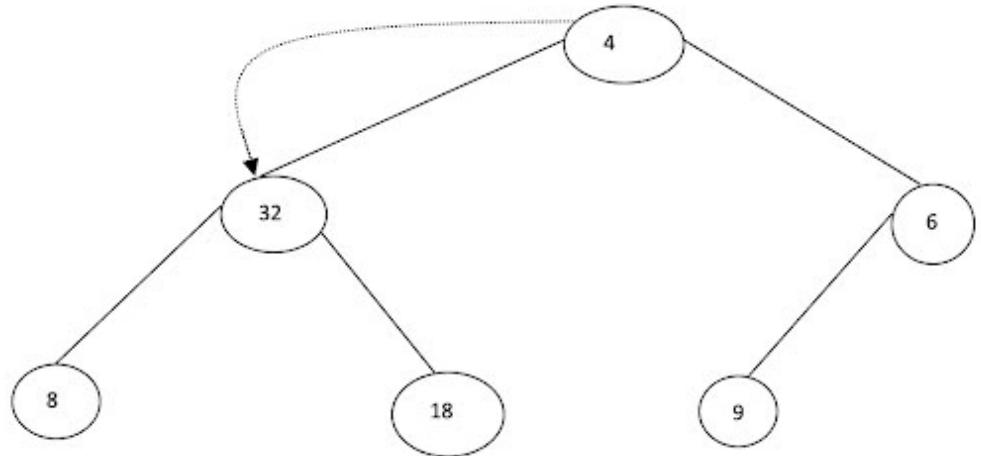
## Deletion

Let us now delete an element from the heap formed in the previous section. Note that we can only delete the element at the root. Therefore, in a max-heap, we get the maximum element on deleting ([figure 10.13](#)):



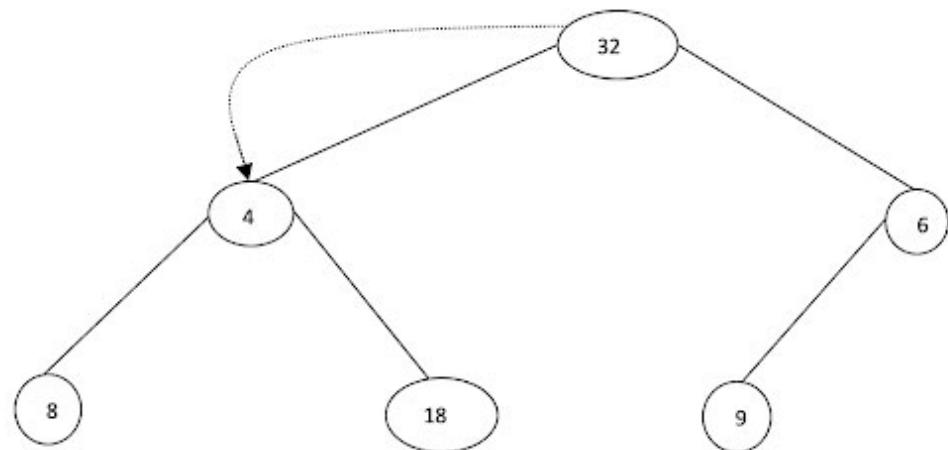
*Figure 10.13: Deleting a node*

We replace the element at the root with the last element of the heap. This may violate the heap property, in which case the root is swapped by the child having greater value. In this case, 4 is swapped with 32 ([figure 10.14](#)):



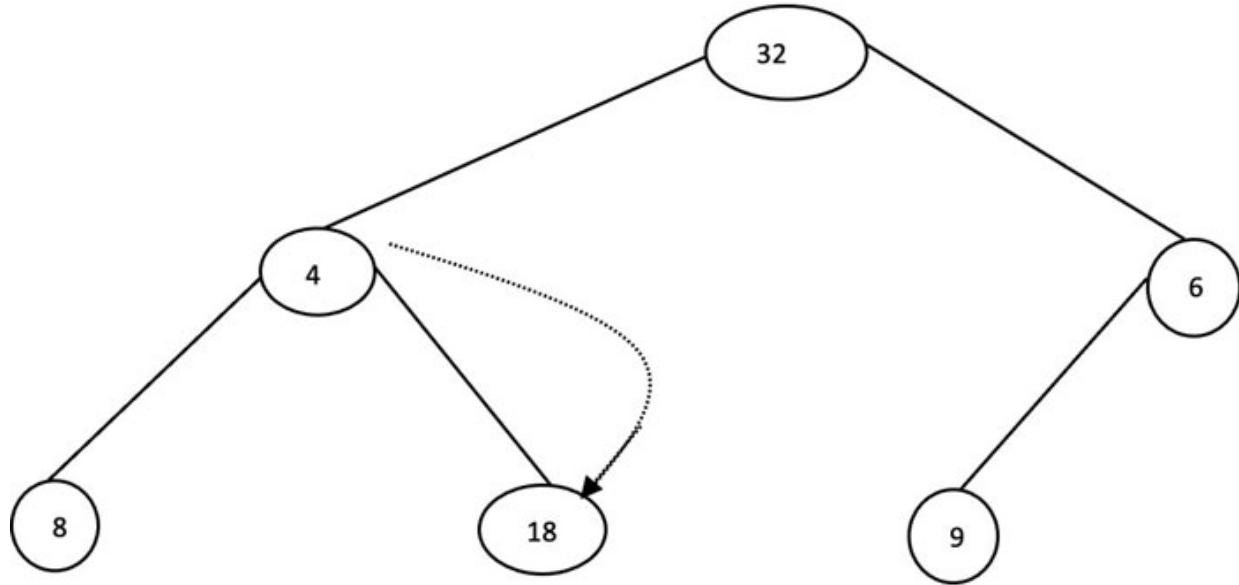
**Figure 10.14:** Placing the last element at the node

The resultant heap is shown in [figure 10.15](#):



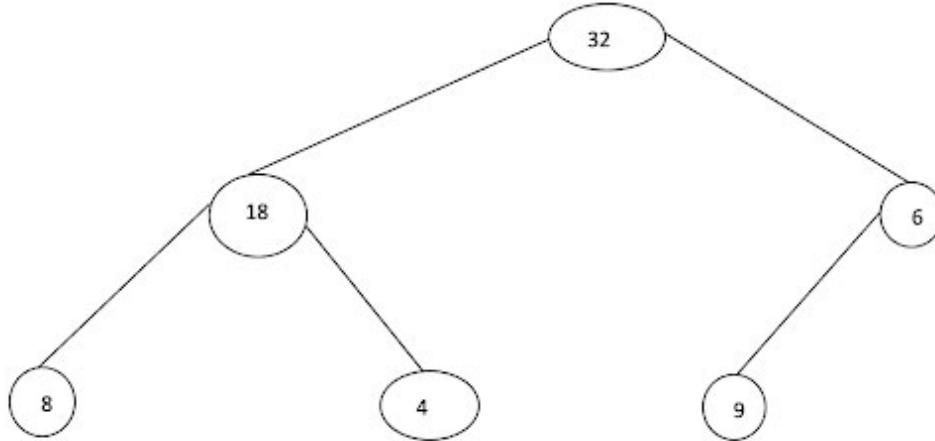
**Figure 10.15:** Swapping 4 and 32

Note that another swapping is needed so that the sanctity of the heap can be maintained. [Figure 10.16](#) shows the swapping of 4 with 18 (child having greater value):



**Figure 10.16:** Swapping 4 and 18

Finally, the following heap ([figure 10.17](#)) is formed after deletion. Note that, after deletion, the second largest element of the original heap is placed at the root. One may also appreciate the fact that the preceding procedure helps us to create a heap out of the remaining values efficiently and effectively:



**Figure 10.17:** Heap formed after deletion

## Complexity

If a heap has  $N$  values, its height is  $\lceil \log_2 N \rceil + 1$ , therefore, deletion requires at most  $O(\log_2 N)$  swaps. Therefore, the complexity of deleting an element in a heap is  $O(\log_2 N)$ .

The following code implements the delete operation:

```
1. def delete(heap):
2.     index=0
3.     while(heap[index]!=0):
4.         temp=heap[index]
5.         index\=index
6.         index+=1
7.         if(index1==0):
8.             temp\=heap[0]
9.             heap[0]=0
10.            return temp
11.        else:
12.            temp\=heap[0]
13.            heap[0]=temp
14.            pos=0
15.            print(index1)
16.            while(pos!=index1):
17.                if(heap[pos]<heap[2*pos+1]):
18.                    heap[pos],heap[2*pos+1]=heap[2*pos+1], heap[pos]
19.                    pos=2*pos+1
20.                    print(pos)
21.                elif(heap[pos]<heap[2*pos+2]):
22.                    heap[pos],heap[2*pos+2]=heap[2*pos+2], heap[pos]
23.                    pos=2*pos+2
24.                    print(pos)
25.            return temp
```

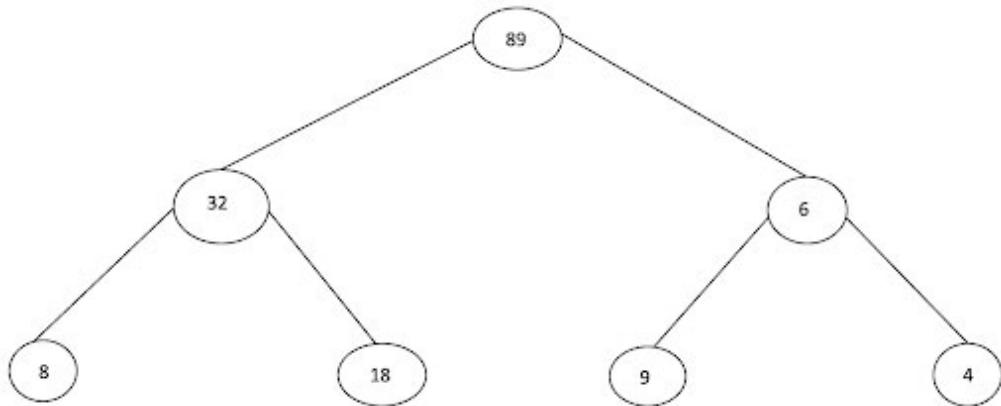
Having seen insertion and deletion from a heap, let us now move to the heap sort.

## Heap sort

Let us now have a look at how a heap can help us in sorting elements in time. For this, we delete one element at a time and re-heapify the rest of the elements. Repeating this  $n$  times will give us a sorted list. To understand this, let us consider the following illustration.

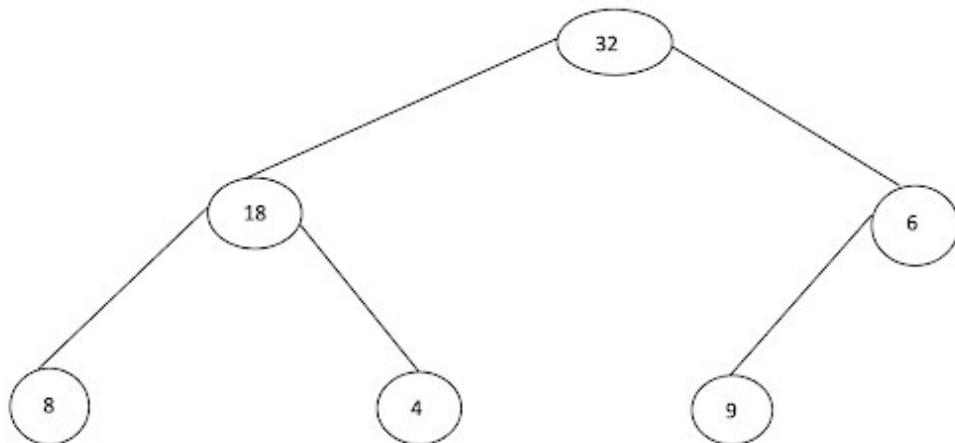
The elements of a set  $\{8, 18, 6, 89, 32, 9, 4\}$  are to be sorted using heap sort. The following steps illustrate the procedure of using heaps to sort the given set:

**Step 1:** In the first step, we create a heap out of the given set ([figure 10.18](#)):



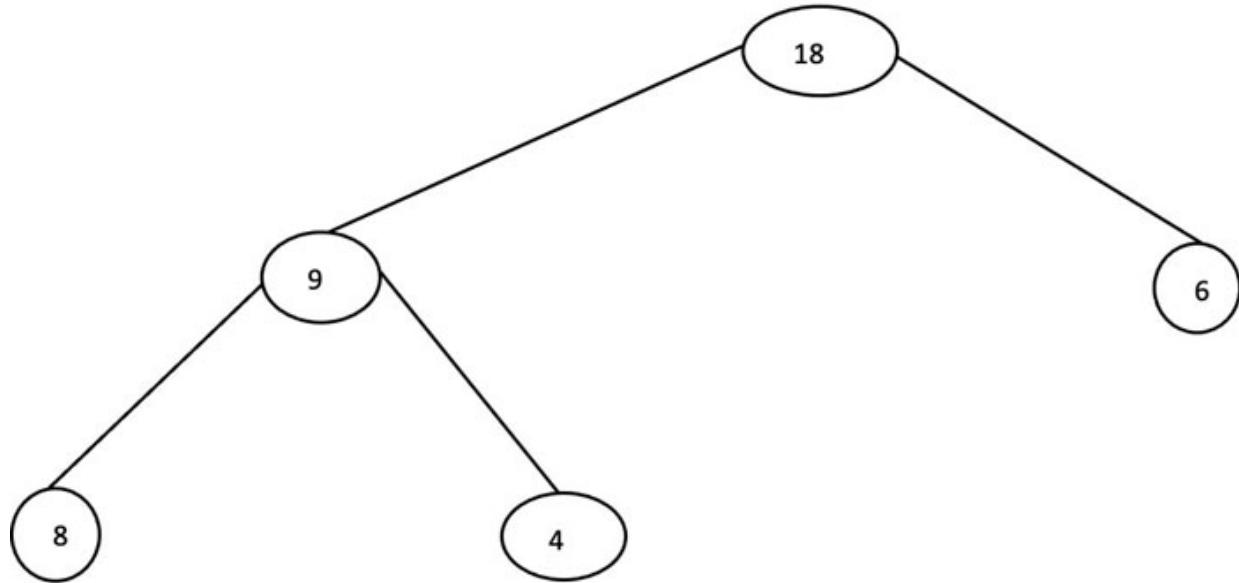
*Figure 10.18: Form a heap out of given elements*

**Step 2:** The element at the root, that is 89, is deleted, and the rest of the elements are re-heapified ([figure 10.19](#)). The list of elements deleted so far is [89]:



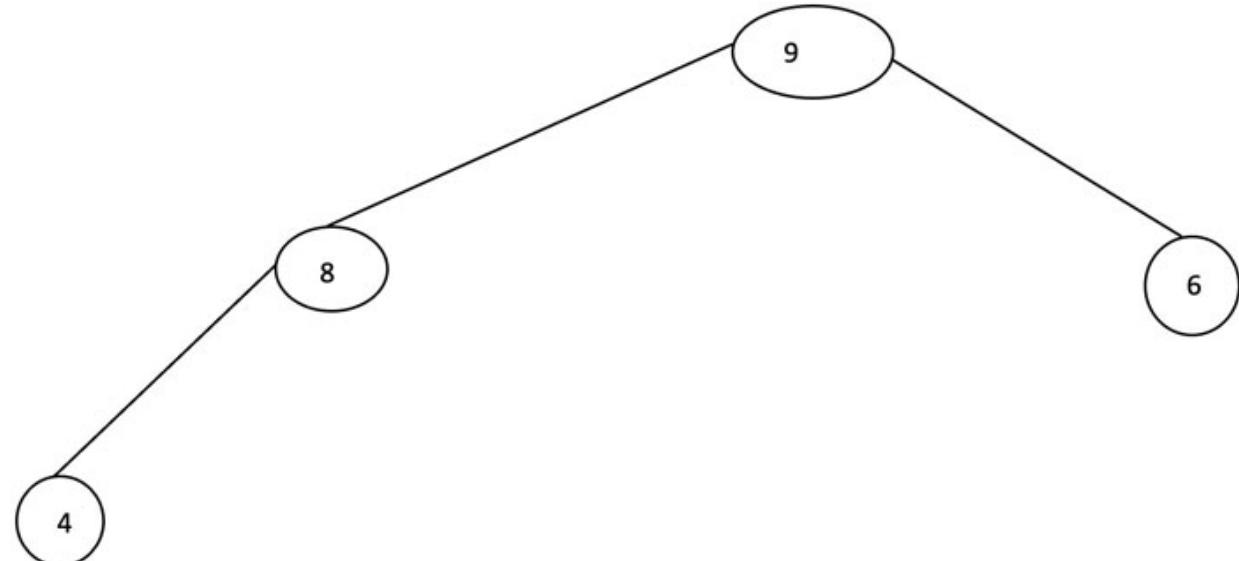
**Figure 10.19:** Delete root and re-heapify

Step 3: Now, 32 is deleted, and the rest of the elements are re-heapified ([figure 10.20](#)). The list of elements deleted so far is [89, 32]:



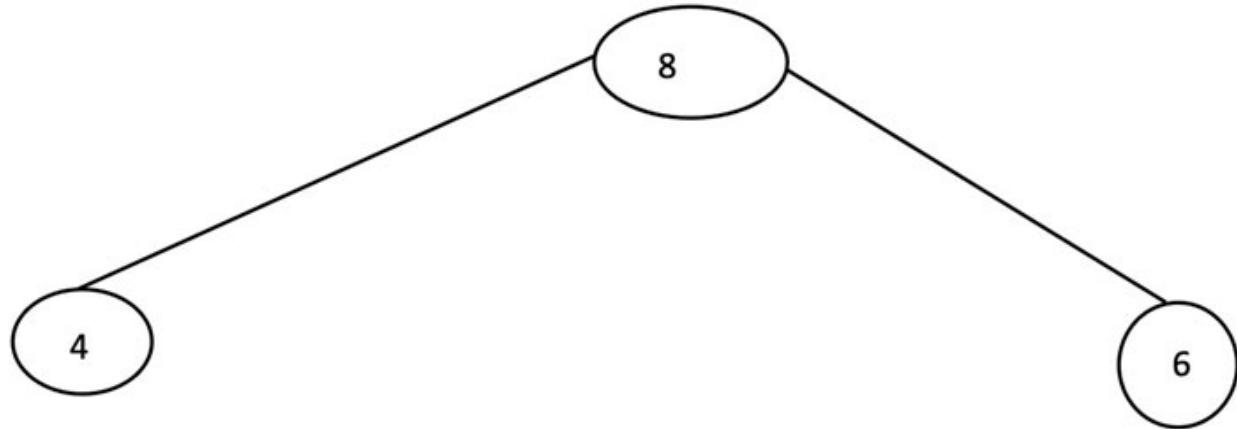
**Figure 10.20:** Delete root and re-heapify

**Step 4:** In the next step, 18 is deleted, and the rest of the elements are re-heapified ([figure 10.21](#)). The list of elements deleted so far is [89, 32, 18]:



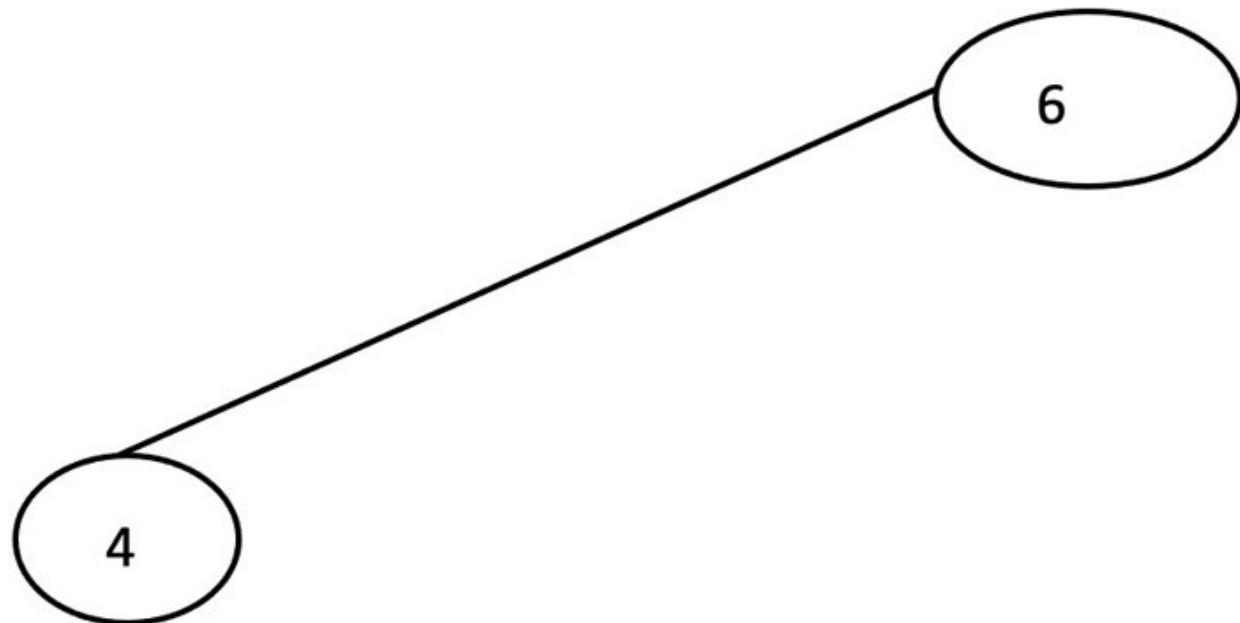
**Figure 10.21:** Delete root and re-heapify

**Step 5:** In this Step, 9 is deleted, and the rest of the elements are re-heapified ([figure 10.22](#)). The list of elements deleted so far is [89, 32, 18, 9]:



*Figure 10.22: Delete root and re-heapify*

**Step 6:** Now, 8 is deleted, and the rest of the elements are re-heapified ([figure 10.23](#)). The list of elements deleted so far is [89, 32, 18, 9, 8]:



*Figure 10.23: Delete root and re-heapify*

**Step 7:** In this step, 6 is deleted. The list of elements deleted so far is [89, 32, 18, 9, 8, 6].

**Step 8:** Finally, 4 is deleted. The list of elements deleted so far is [89, 32, 18, 9, 8, 4].

Note that the list so formed is a sorted list. Since each deletion requires, therefore, the complexity of deleting  $n$  elements from a heap to get a sorted list is  $O(n \log n)$ .

The following algorithm enumerates the steps of Heapsort:

1. First of all, we build a Max-heap from the given elements.
2. Delete an element from the heap using the Delete procedure. The root is returned, and the rest of the elements are re-heapified.
3. Repeat Step 2 till the heap contains elements.
4. The elements returned in Step 2 will form the sorted list.

The reader is expected to implement heap-sort. In case help is needed, the code has been included in the web resources.

## Problems

1. **Can we delete any element from a Max-heap (Complexity is not a consideration)?**

**Solution:** Yes! First, we will find the element to be deleted. This is followed by performing heapify at that position.

2. **Suggest an algorithm that finds all the elements greater than “item” from a Max-heap.**

**Solution:** Start from the root. If the root is greater than “item”, print the value and apply the procedure to the left or/and the right child, whichever is greater than the root. Continue this till elements less than “item” are encountered.

3. **Can you merge two heaps?**

**Solution:** Place the elements of the second heap after the element’s first heap in an array, and heapify the array.

4. **The height of a heap is  $H$ . How many elements can the last level have?**

**Solution:** The heap is a complete binary tree. The last level has a maximum of  $2^{H-1}$  elements. The minimum number of elements in this level can be 1.

**5. From the preceding answer, infer the maximum and the minimum number of elements in a heap of height H.**

**Solution:** Kindly refer to the text and figure out the solution.

**6. At which level would you find the minimum element in a Max-heap?**

**Solution:** Last level

**7. At which level would you find the maximum element in a Min-heap?**

**Solution:** Last level

**8. Can the in-order traversal of a heap generate a sorted list? Give a reason in support of your answer.**

**Solution:** No, the root is either the maximum or minimum element of the heap. In the in-order traversal, the root will appear between the same traversals of the left and the right sub-tree.

**9. How can we obtain the  $i$ th maximum element from a Max-heap?**

**Solution:** Perform  $i$  deletions to get the requisite element.

**10. How can we implement a stack using a heap?**

**Solution:** We can store the item number along with the item and set it as the priority.

## Conclusion

This chapter introduced the heap data structure, which is an implementation of the priority queue. There are two types of heaps: max-heap and Min-heap. The maximum element from a Max-heap can be returned in  $O(1)$ ; likewise, the minimum element from a Min-heap can be returned in  $O(1)$ . The insertion in a heap can be done in  $O(\log n)$  time, and the complexity of deletion and re-heapifying is also  $O(\log n)$ . This data structure can be used to sort a given list in  $O(n \log n)$  time.

The following chapters use heap often; therefore, this chapter forms the basis of some assorted topics that we are going to study. The data structure can also be used to solve some interesting problems.

The upcoming chapter takes the discussion forward and introduces Graphs, and takes you to a whole new world. The reader is expected to attempt the problems given at the end of this chapter for better understanding.

## Multiple choice questions

**1. Which of the following is an implementation of a priority queue?**

- a. Binary Search Tree
- b. Graph
- c. Heap
- d. None of the above

**2. In a Max-heap, the delete operation returns**

- a. The maximum element
- b. The minimum element
- c. Both
- d. None of the above

**3. In a Max-heap, the minimum element is present at**

- a. The last level
- b. The last but one level
- c. Can be present at any level
- d. None of the above

**4. The complexity of insertion into a heap is**

- a.  $O(1)$
- b.  $O(n)$
- c.  $O(\log n)$
- d. None of the above

**5. The complexity of deletion from a heap is**

- a.  $O(1)$
- b.  $O(n)$

- c.  $O(\log n)$
- d. None of the above

6. **The complexity of finding the maximum element from a heap is**

- a.  $O(1)$
- b.  $O(n)$
- c.  $O(\log n)$
- d. None of the above

7. **The complexity of Heap sort is**

- a.  $O(n)$
- b.  $O(n^2)$
- c.  $O(n \log n)$
- d. None of the above

8. **The complexity of converting a max heap to a min-heap is**

- a.  $O(n)$
- b.  $O(n^2)$
- c.  $O(n \log n)$
- d. None of the above

9. **The complexity of finding the minimum element from a max heap is**

- a.  $O(n)$
- b.  $O(n^2)$
- c.  $O(n \log n)$
- d. None of the above

10. **The complexity of finding the maximum element from a min-heap is**

- a.  $O(n)$
- b.  $O(n^2)$
- c.  $O(n \log n)$

d. None of the above

## **Programming**

### **Level 0:**

1. Write a program to create a heap out of the given set of numbers.
2. Write a program to delete an element from the preceding tree.
3. Write a program to implement heap sort.

### **Level 1:**

1. Find the maximum element from a min-heap.
2. Write a program to form a min tree from a max tree.
3. Implement stack using heaps.
4. Implement Queue using heaps.

### **Level 2:**

1. Write a program to merge two heaps.
2. Find all the elements greater than the “item” from a given heap.

## **Further references**

- [https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6\\_006F11\\_lec04.pdf](https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec04.pdf)
- <https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture4.pdf>

## **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 11

## Graphs

### Introduction

Trees and graphs are examples of non-linear data structures. The previous two chapters discussed Trees and their applications. These chapters focused on the representation of trees, their traversal, and their types. The focus was particularly on Binary Search Trees and B-Trees. This chapter takes the discussion forward and introduces the graph data structures.

Have you ever wondered how apps like Google maps find the shortest path from one point to another or LinkedIn finds your first, second, and further connections? These tasks are accomplished using graphs. These data structures not only work wonders in the preceding problems, but it also finds applications in circuit theory, discrete mathematics, and so on.

This chapter introduces the graph data structures and discusses their representation, traversal, and applications like finding the shortest path. The reader will be able to implement the preceding algorithms and apply them to solve complex problems using them.

### Structure

The main topics covered in this chapter are as follows:

- Representation
- Traversals
  - Depth first search
  - Breadth-first search
  - Topological sort
- Spanning trees
- Kruskal's algorithm

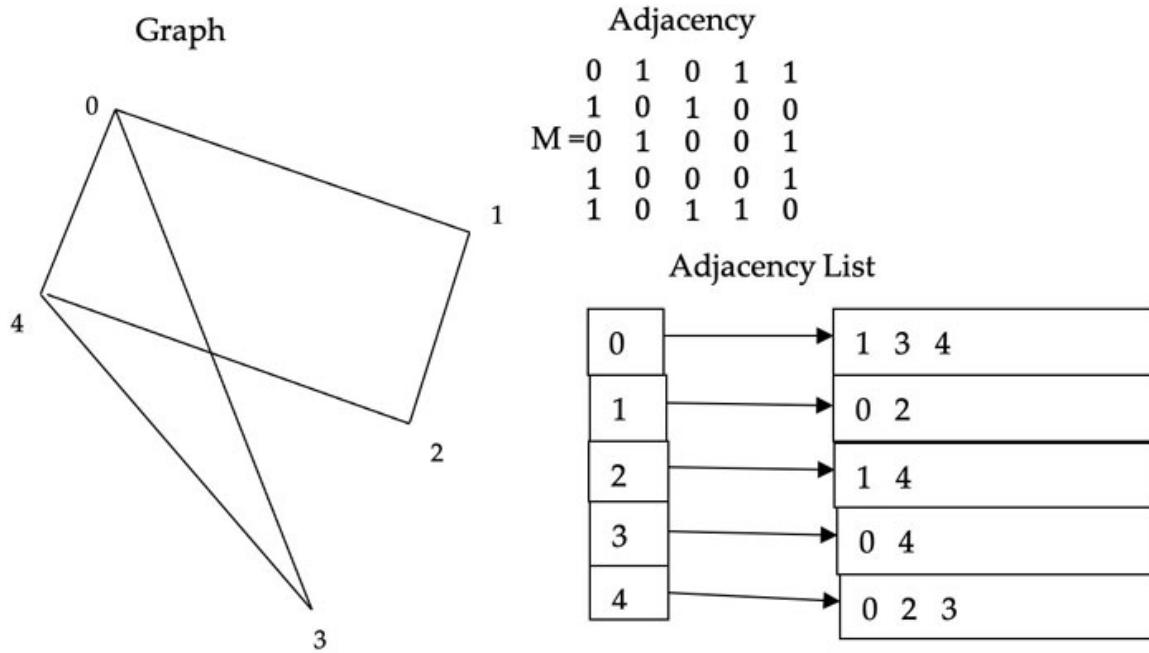
### Objectives

After reading this chapter, the reader will be able to understand the representation, traversal, and applications of graph data structures. The chapter discusses the adjacency matrix and adjacency list representations of graphs. This is followed by the three most common traversals, namely, Depth First Search, Breadth First Search (for general graphs), and Topological Sort for Directed Acyclic Graphs. The spanning tree and two greedy approach-based algorithms for the same follow.

## Representation

The graphs can be represented using an adjacency matrix or adjacency list. The adjacency matrix of a given graph can be created by a matrix of order  $n \times n$ , where  $n$  is the number of vertices in the graph. For example, the graph shown in [figure 11.1](#) has five vertices. The corresponding matrix has a 1 at  $(i, j)$  if there is an edge from vertex  $i$  to vertex  $j$ , else the  $(i, j)^{\text{th}}$  element is 0.

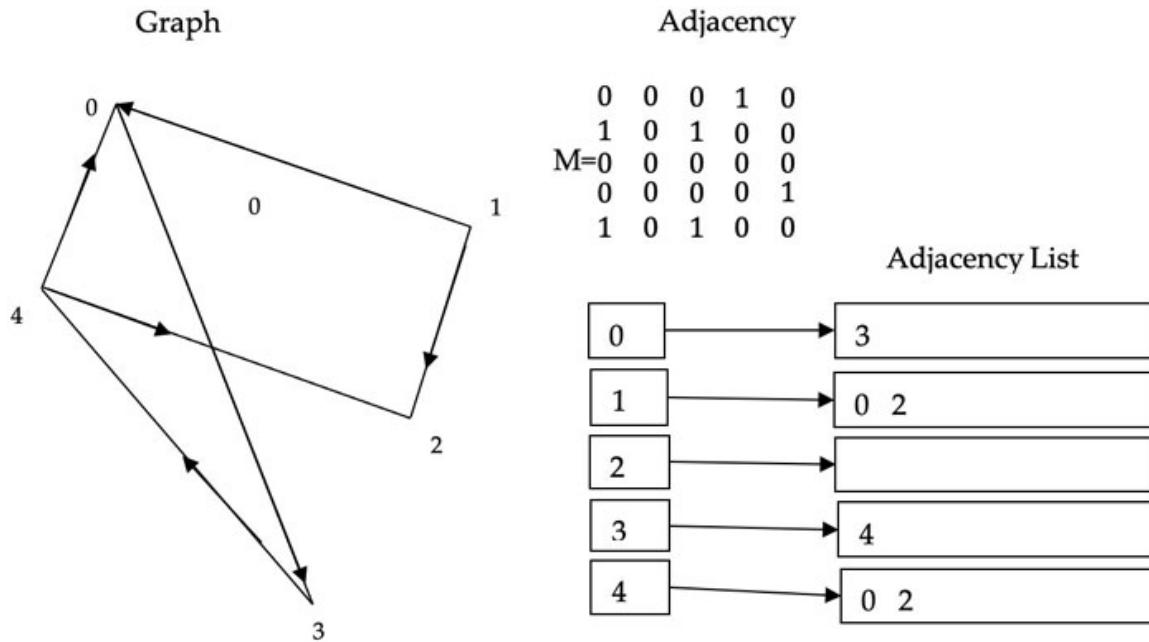
The adjacency list representation has a linked list attached to each node. Each linked list contains the nodes adjacent to the given node. For example, in [figure 11.1](#), the nodes adjacent to 0 are 1, 3, and 4; those adjacent to 1 are 0 and 2; those adjacent to 2 are 1 and 4; to 3 are 0 and 4; and those adjacent to 4 are 0, 2, and 3:



*Figure 11.1: Adjacency matrix and adjacency list of a given graph*

In a directed graph, the edges are from a source node to the destination node, and the opposite may not be true. Therefore, the adjacency matrix corresponding to a directional graph is generally not symmetric. Note that the matrix corresponding

to the graph shown in [figure 11.2](#) is not symmetric. The list representation of this graph is also shown in the following figure:



*Figure 11.2: Adjacency matrix and adjacency list of a given graph*

The following code stores the graph entered by the user in a matrix of order ( $V, V$ ), where  $V$  is the number of vertices. Note that in each iteration, the user is asked to enter whether the given edge exists (1) or not (0). Since the graph is assumed to be bidirectional, only the upper half of the matrix needs to be filled, and since it is symmetric,  $M(j,i)$  is assigned the value stored in  $M(i,j)$ .

### Code 1:

```

1. v = int(input('Enter the number of vertices\t:'))
2. M = np.zeros((v,v))
3. for i in range(v):
4.     for j in range(v):
5.         if(i< j):
6.             str1='Edge from '+ str(i)+ ' to '+ str(j)+ '\t(1/0)\t:'
7.             M[i, j]= int(input(str1))
8.             M[j, i]= M[i, j]
9. print(M)

```

The following code converts the matrix created in the preceding code to the adjacency list representation. Here, each node is stored as the key to the dictionary called Graph1, and the list corresponding it can be created by storing the indices of the corresponding row in the matrix in a list which becomes the value for that particular key.

### Code 2:

```
1. Graph1={}
2. for i in range(M.shape[0]):
3.     list1=[j for j in range(M.shape[1]) if M[i, j]==1]
4.     Graph1[i]=list1
5. print(Graph1)
```

To find that there is an edge from  $i$  to  $j$  in the adjacency matrix representation of the graph, the element corresponding to  $M[i,j]$  should be 1. Code 3 implements the **isEdge()** function for an adjacency matrix. Likewise, to find if there is an edge in the adjacency list representation, the list corresponding to the particular key is checked for the destination vertex. For example, to find if there is an edge from 1 to 2, the list corresponding to 1 is checked for 2. Code 4 implements **isEdge1()** for adjacency list representation:

### Code 3:

```
1. def isEdge(M, i, j):
2.     if(M[i, j]==1):
3.         return True
4.     else:
5.         return False
```

### Code 4:

```
1. def isEdge1(Graph1, i, j):
2.     if j in Graph1[i]:
3.         return True
4.     else:
5.         return False
```

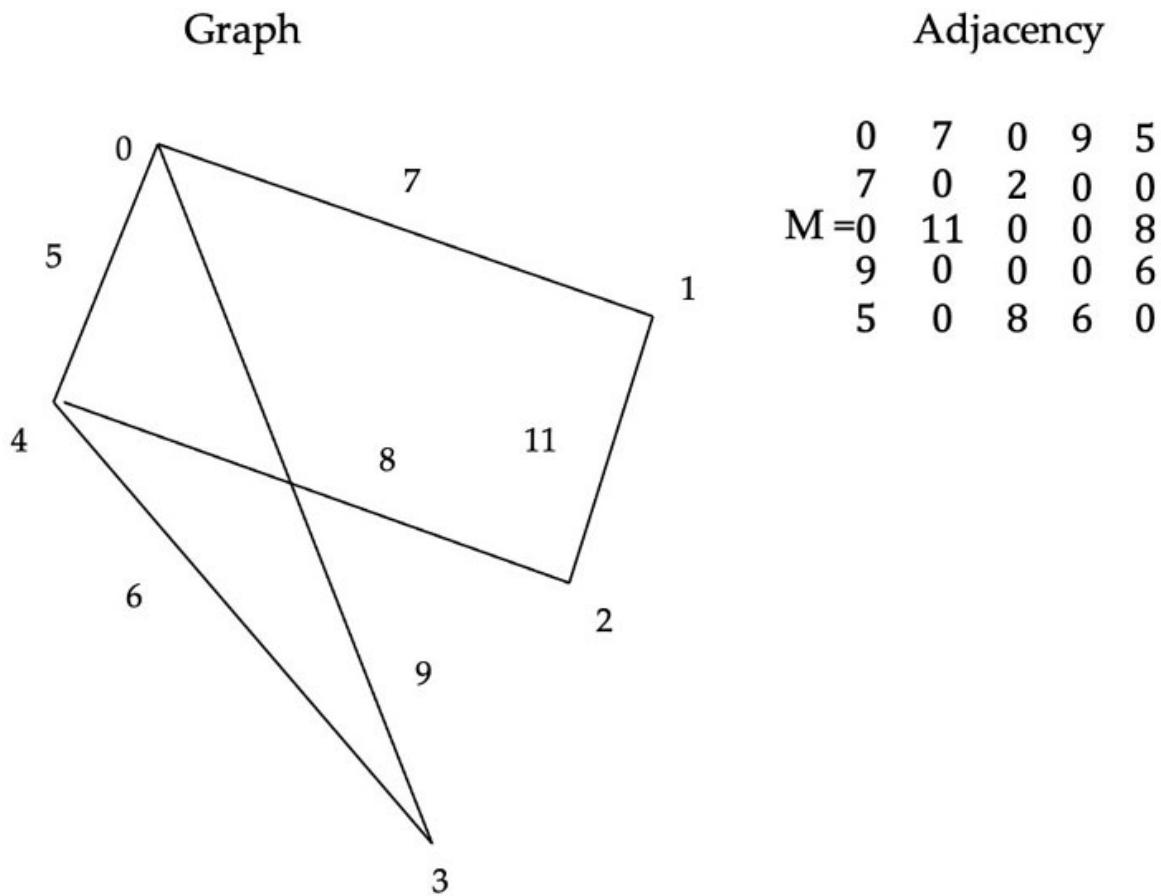
It is easy to insert an edge in the matrix representation. In order to insert an edge from  $i$  to  $j$ , we set  $M[i, j]$  to 1. Likewise, to insert an edge in the list representation, we add the destination vertex in the list corresponding to the source vertex.

We can also calculate the out edges and in edges in the adjacency list representation by displaying the list corresponding to the vertex in the adjacency list representation.

### Code 5:

```
1. # For Adjacency Matrix Representation  
2. def insertEdge(M, i, j):  
3.     M[i, j]=1  
4. # For Adjacency List Representation  
5. insertEdge(M, 2, 3)
```

In the case of a weighted graph, the adjacency matrix of a given graph can be created by a Matrix of order  $n$ , where  $n$  is the number of vertices in the graph. For example, the graph shown in [figure 11.3](#) has five vertices. The corresponding matrix has a at  $(i, j)$  if the weight of the edge from vertex  $i$  to vertex  $j$ , else the  $(i, j)^{th}$  element is 0:



*Figure 11.3: Adjacency matrix of a given graph*

Having seen the two most common representations of the graph, let us now move to the traversals of graphs.

## Traversals

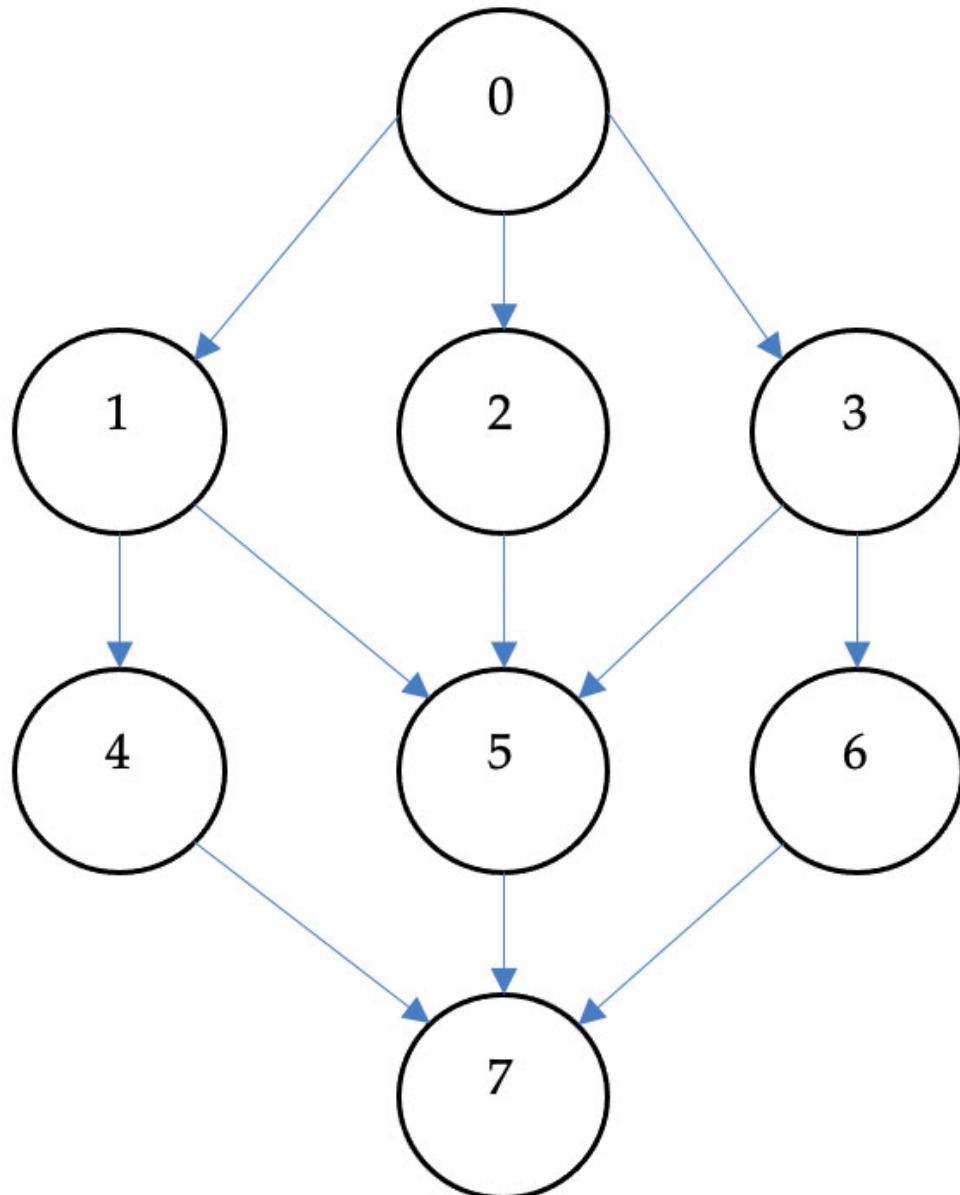
Like in the case of trees, traversal is the most important task in Graphs. This section discusses two traversals for directed graphs, namely, Depth First Search and Breadth First Search, and an algorithm for Directed Acyclic Graph called topological sorting.

## Depth First Search

The **Depth First Search (DFS)** traverses the given graph by processing the source node, followed by its first neighbor. The neighbors of the node that is processed next are found, and the process is repeated till no node is left. The algorithm for finding the DFS of a given graph is as follows:

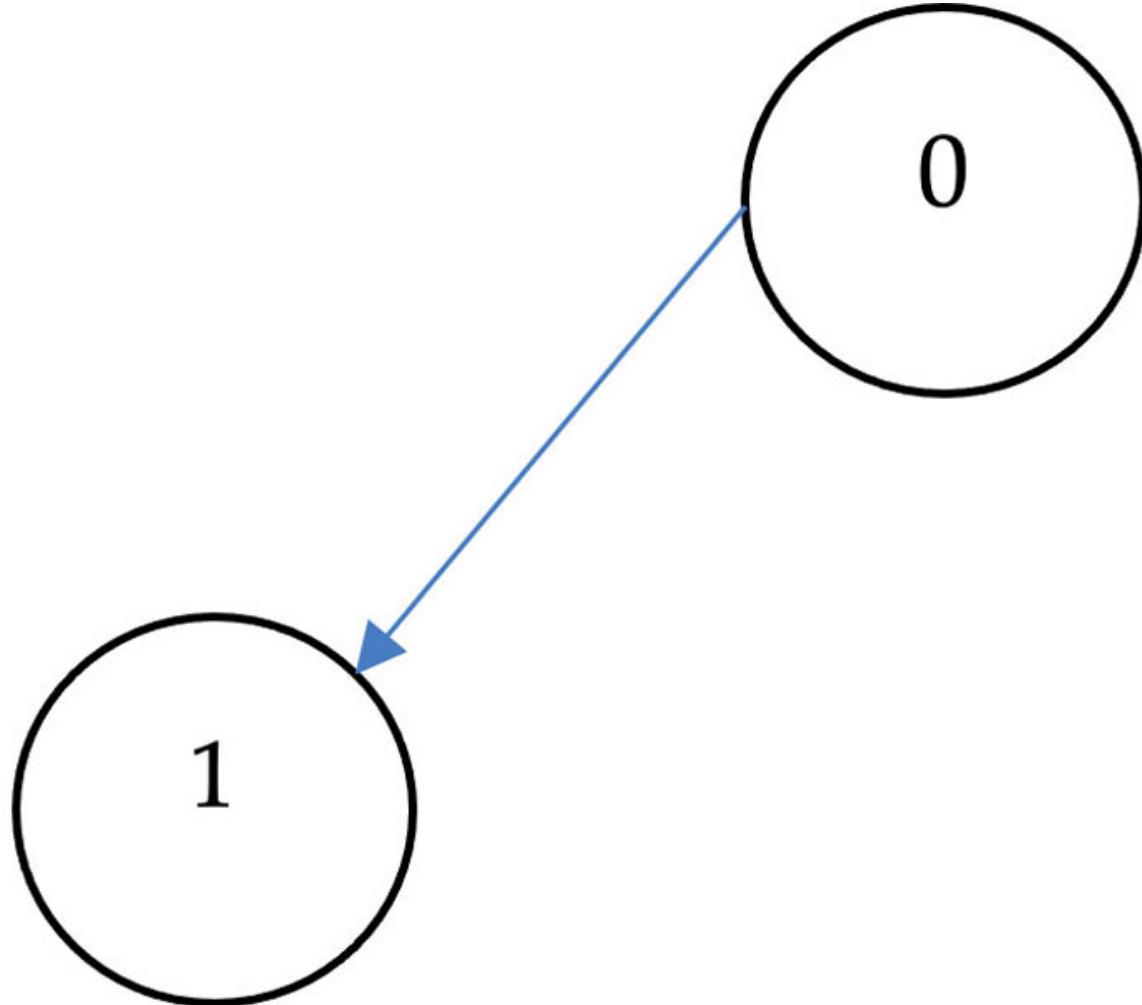
- For a given graph, set the source vertex as visited and find its adjacent vertices.
- For each adjacent vertex (if it is not visited), set it as visited and then recursively call DFS.
- To call DFS, initially set all the nodes to not visited, then for each node, set visited = False and call DFS.

For example, consider the graph shown in [figure 11.4](#). The steps of DFS are shown in [figures 11.5](#) to [11.12](#):



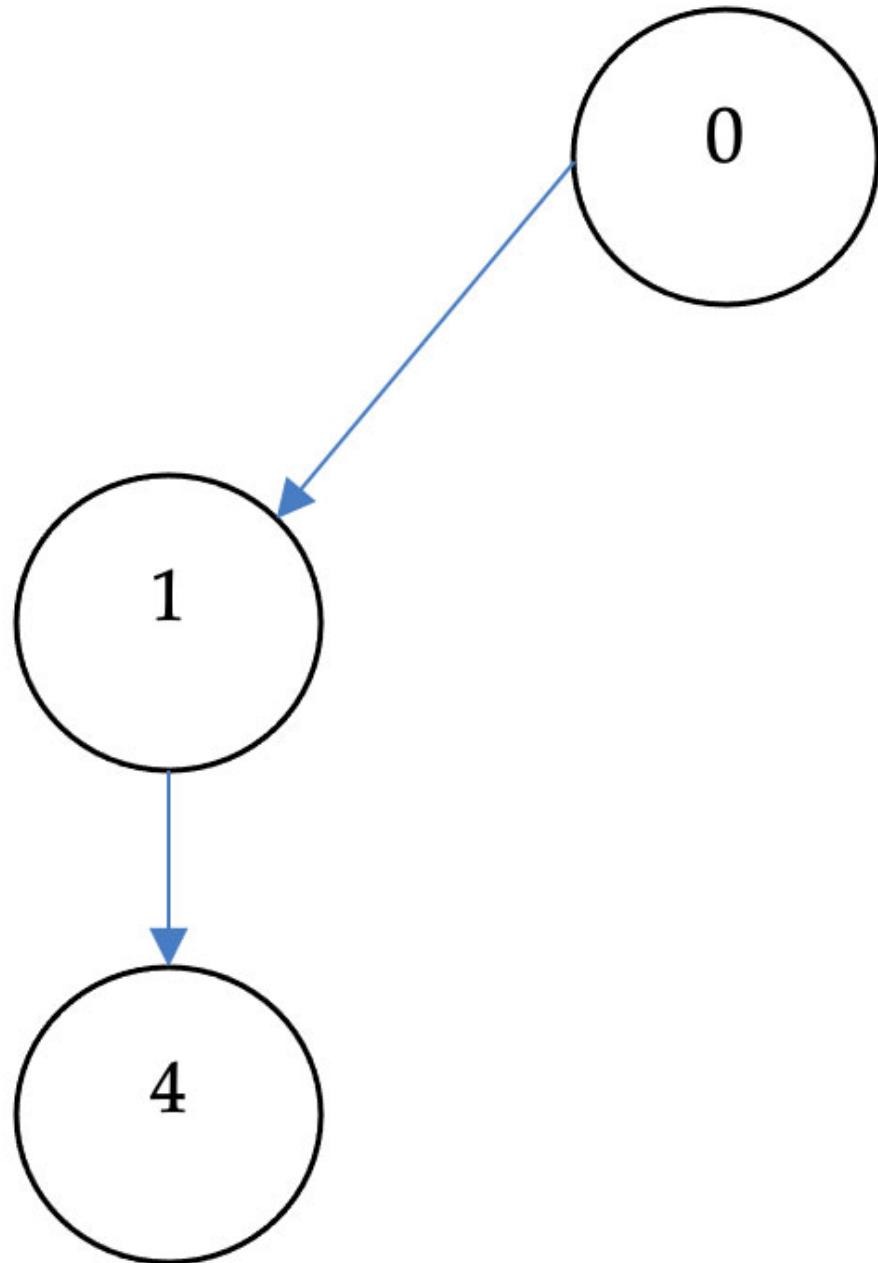
**Figure 11.4:** Graph for DFS

**Step 1:** 0 is the source, and 1 is the first neighbor of 0 ([figure 11.5](#)). So, the process begins by processing 0, followed by 1:



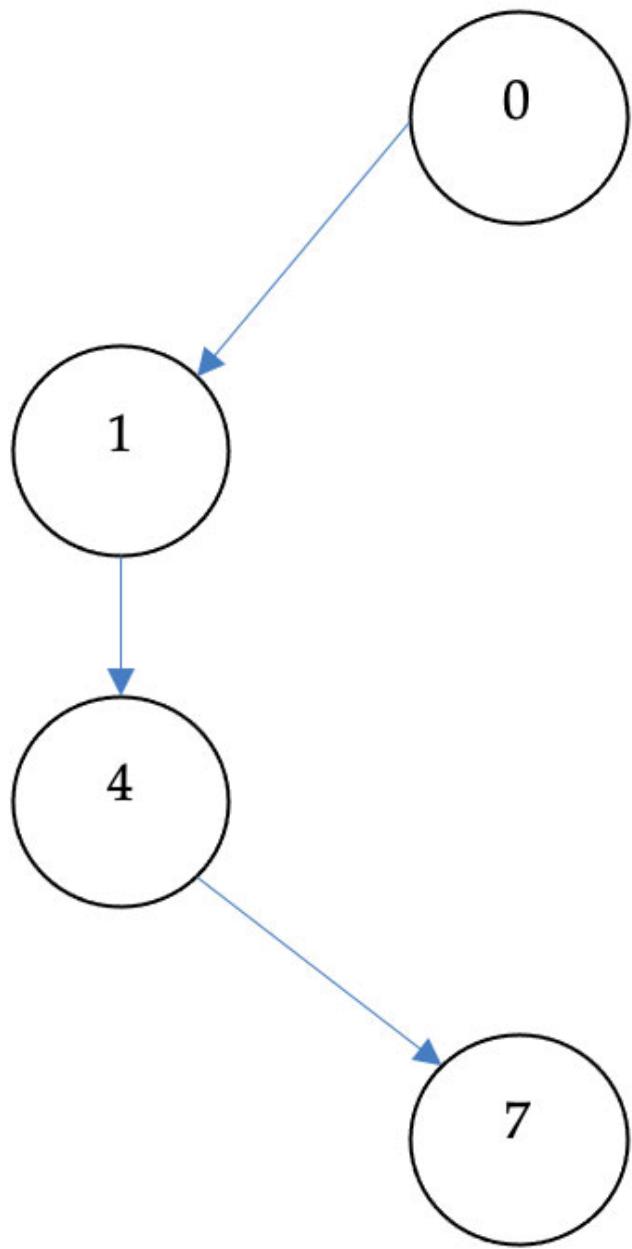
*Figure 11.5: Figure for Step1*

**Step 2:** The first neighbor of 1 is 4 ([figure 11.6](#)). That is, 4 is processed next:



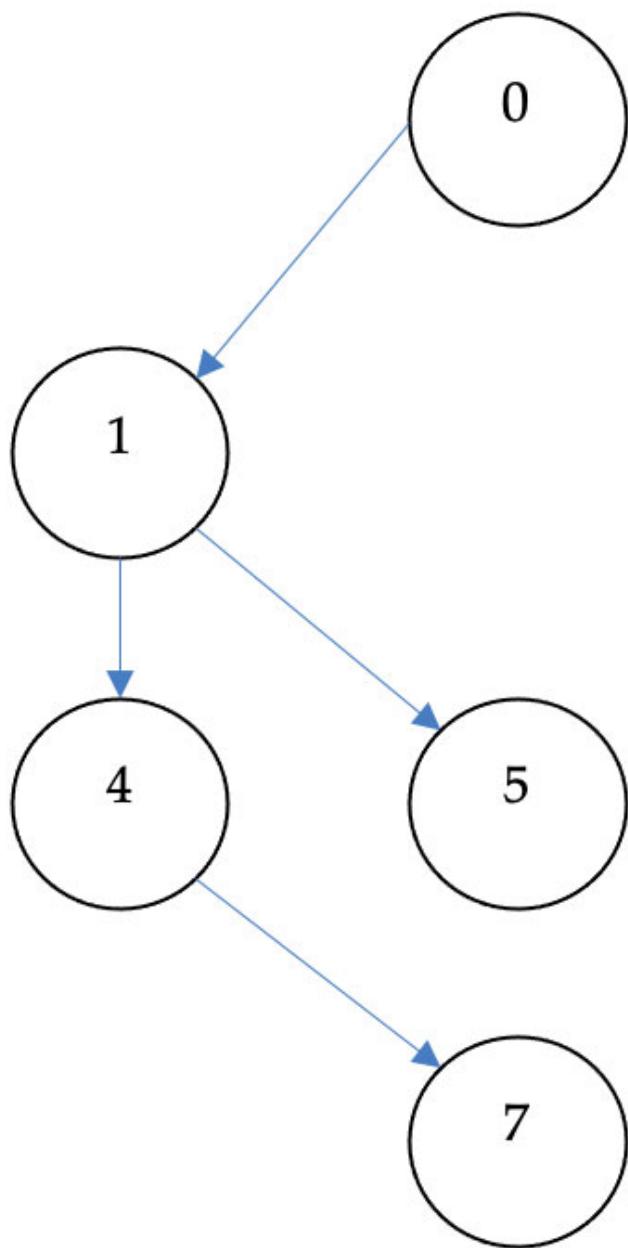
*Figure 11.6: Figure for Step2*

**Step 3:** The first neighbor of 4 is 7 ([figure 11.7](#)). Therefore, 7 is processed after 4:



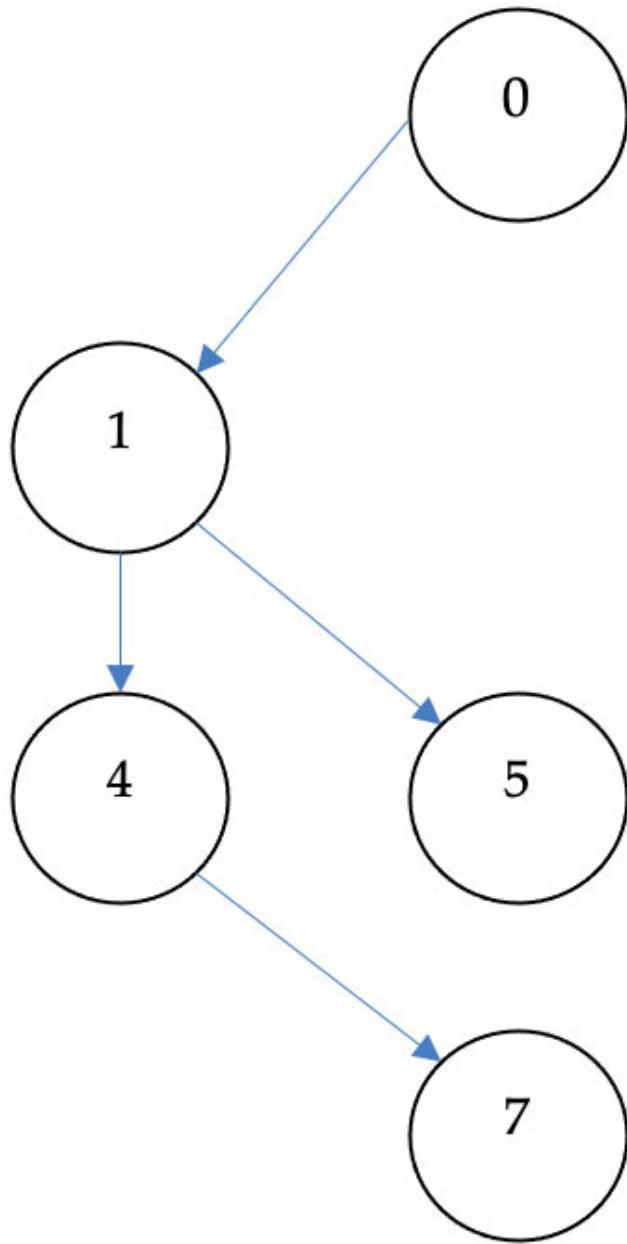
*Figure 11.7: Figure for Step3*

**Step 4:** Note that 7 does not have any neighbor that has not been processed, so we backtrack to 4 and then 1, from where we move to 5, as shown in [figure 11.8](#):



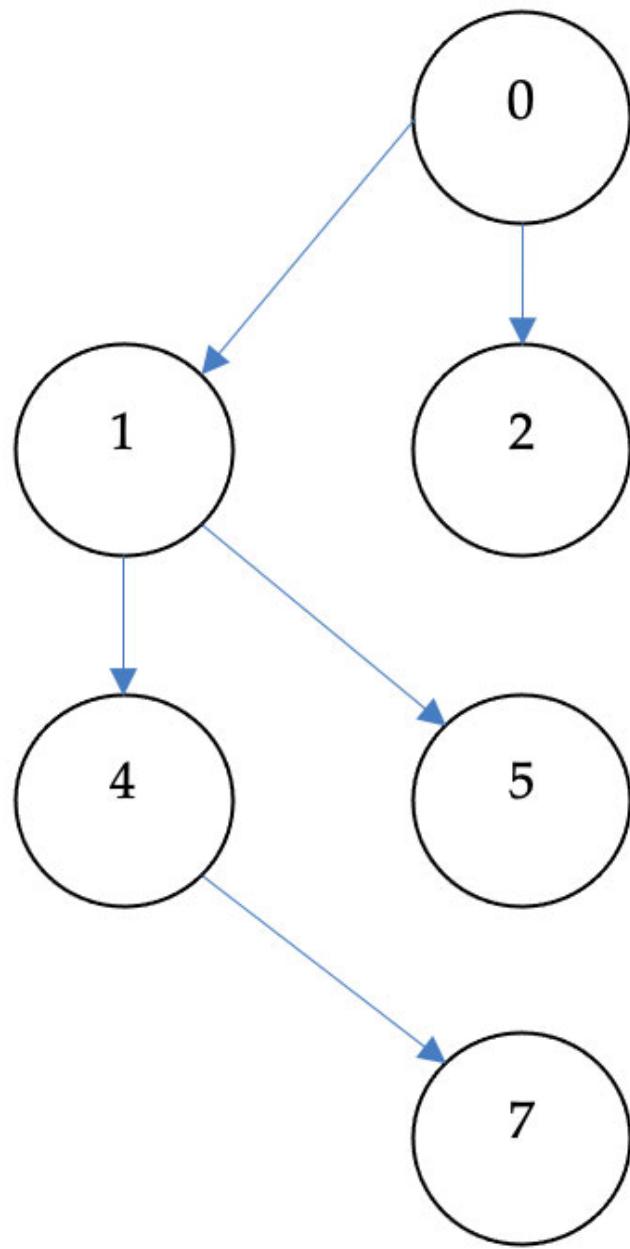
*Figure 11.8: Figure for Step4*

**Step 5:** The first neighbor of 5 is 7, but it has already been processed, as shown in [figure 11.9](#):



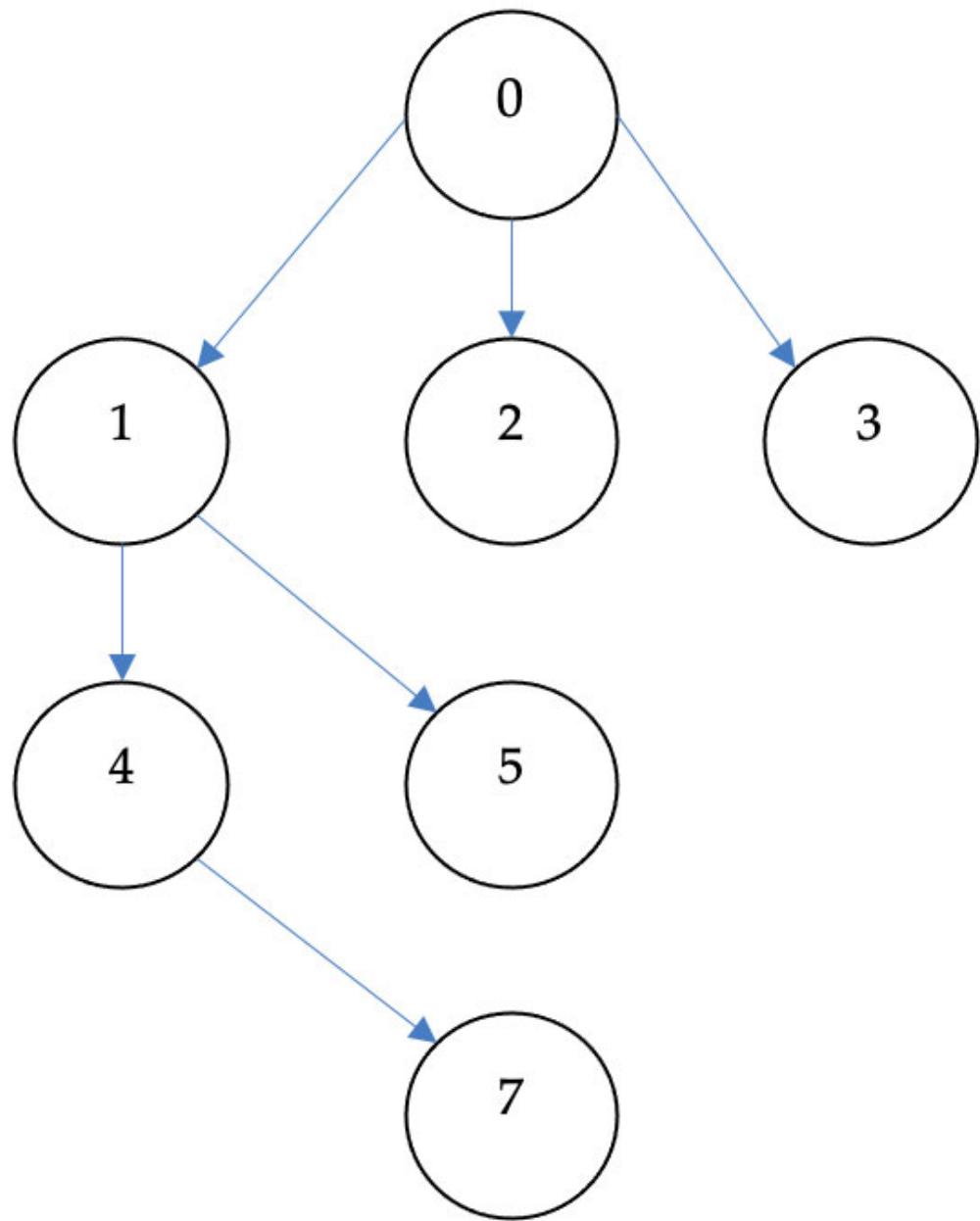
*Figure 11.9: Figure for Step5*

**Step 6:** Likewise, we backtrack to 1 followed by 0, and then move to 2 ([figure 11.10](#)).The next node to be processed is 2:



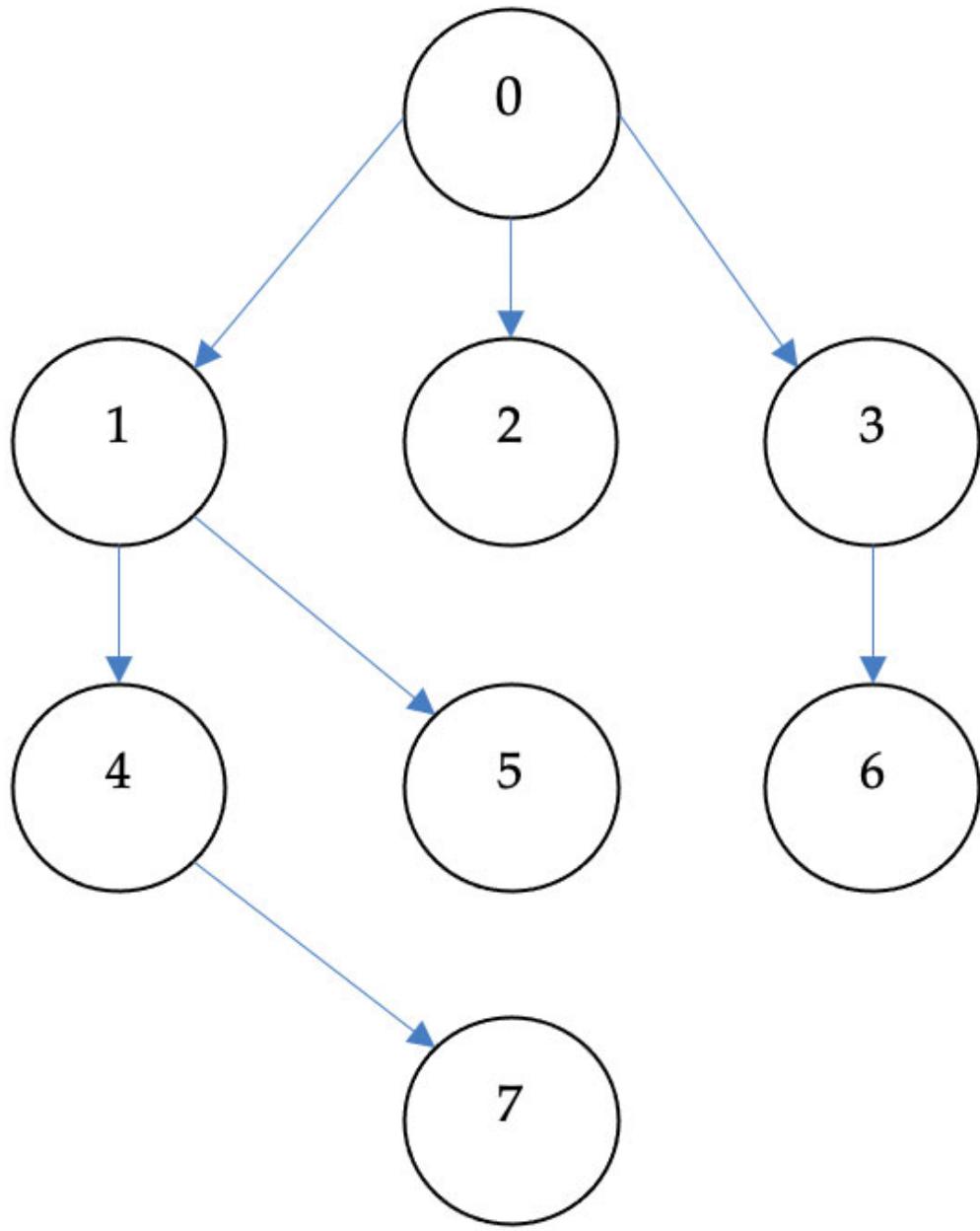
*Figure 11.10: Figure for Step6*

**Step 7:** Finally, we backtrack to 0, followed by processing 3, as shown in [figure 11.11](#):



*Figure 11.11: Figure for Step 7*

**Step 8:** Finally, node 6 is processed. The DFS of the given tree is, therefore, [0, 1, 4, 7, 5, 2, 3, 6] ([figure 11.12](#)):



**Figure 11.12:** Figure for Step8

The algorithm has been implemented as follows:

**Code 6:**

1. `from collections import defaultdict`
2. `class Graph:`
3.  `def __init__(self):`

```

4.     self.graph = defaultdict(list)
5.     def insertEdge(self,u,v):
6.         self.graph[u].append(v)
7.     def DFS(self, NameError):
8.         V = N
9.         # Set the vertices to 'not visited'
10.        visitedNodes =[False]*(V)
11.        print(self.graph)
12.        print(visitedNodes)
13.        for i in range(V):
14.            if visitedNodes[i] == False:
15.                self.DFS1(i, visitedNodes)

```

### **Output:**

```

Enter the number of vertices8
defaultdict(<class <list'>, {0: [1, 2, 3], 1: [4, 5], 2: [5], 3: [5, 6], 4: [7],
5: [7], 6: [7]})

[False, False, False, False, False, False, False, False]
0
1
4
7
5
2
3
6

```

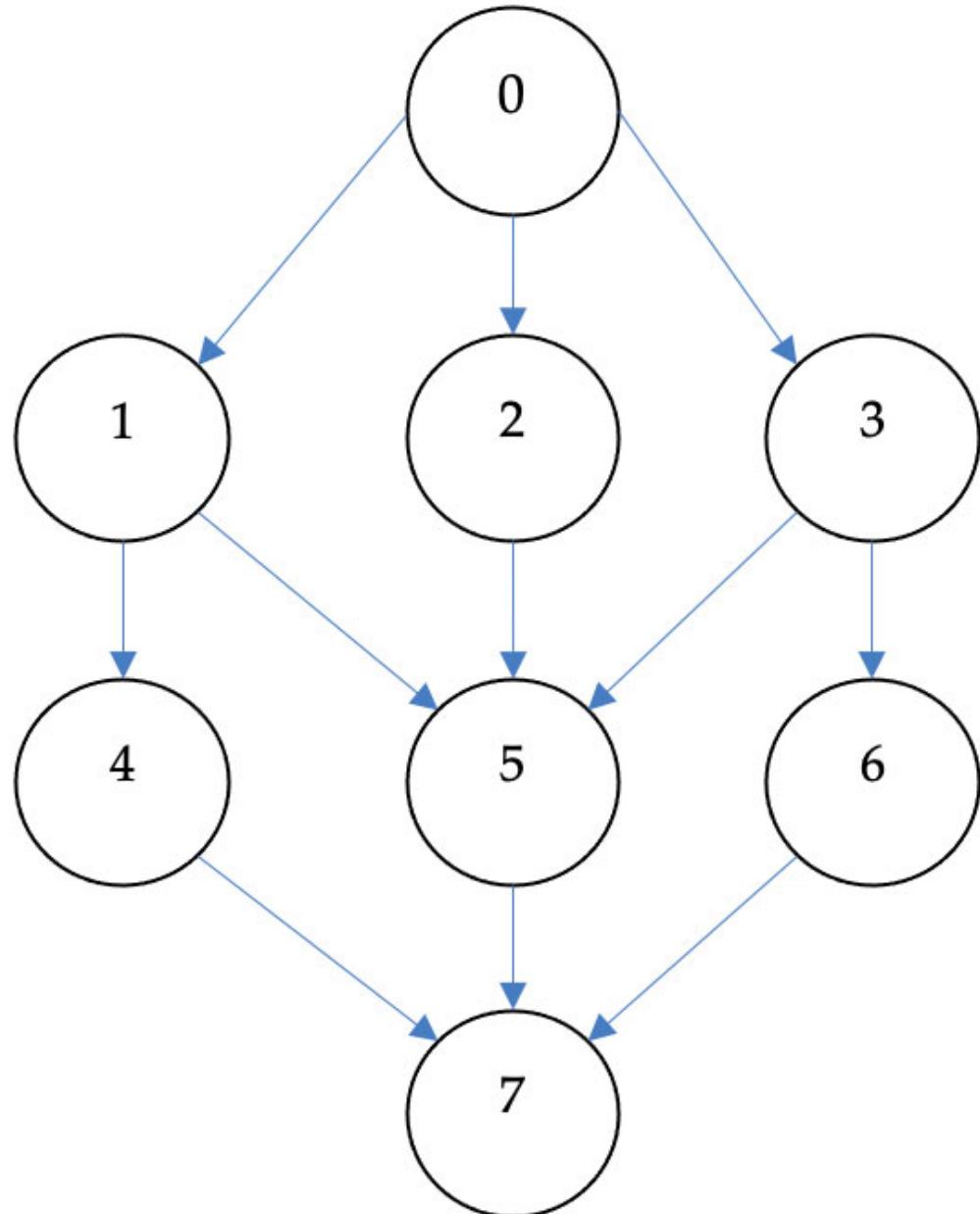
Note that DFS has been implemented using Stacks. The next traversal Breadth First Search is implemented using Queues.

### **Breadth First Search**

The **Breadth First Search (BFS)** traverses the node, followed by all its neighbors at the next level. It can be implemented using a queue. The algorithm for the same is as follows:

- Initially, all the nodes are set to not visited; we insert the source node in the queue, pop it, print it, and then enqueue its neighbors in the queue.
- This is followed by popping out a vertex from the queue, setting it to visited, and putting its neighbors in the queue till all the nodes have been visited.

For example, consider the graph shown in [figure 11.13](#):



*Figure 11.13: Graph for BFS*

**Step 1:** Initially, the root is put into the queue, and the visited array does not contain anything.

**Queue:**

0
---

**Visited:**

--	--	--	--	--	--	--	--

**Step 2:** In the next step, we put all the neighbors of 0 into the queue.

**Queue:**

1	2	3
---	---	---

**Visited:**

0							
---	--	--	--	--	--	--	--

**Step 3:** The first element of the queue is 1 (as shown previously). This element is popped, and all its neighbors are put into the queue. That is 1 is processed, and 5 is inserted in the queue.

**Queue:**

2	3	4	5
---	---	---	---

**Visited:**

0	1						
---	---	--	--	--	--	--	--

**Step 4:** The first element of the queue, formed so far, is 2 (as shown previously). The element is popped out and marked as visited, and all its neighbors are put in the queue.

**Queue:**

3	4	5
---	---	---

**Visited:**

0	1	2					
---	---	---	--	--	--	--	--

**Step 5:** The first element of the queue formed so far is 3 (as shown previously). The element is popped out and marked as visited, and all its neighbors are put in the queue.

**Queue:**

--	--

4	5	6
---	---	---

**Visited:**

0	1	2	3				
---	---	---	---	--	--	--	--

**Step 6:** The first element of the queue, formed so far, is 4 (as shown previously). The element is popped out and marked as visited, and all its neighbors are put in the queue.

**Queue:**

5	6	7
---	---	---

**Visited:**

0	1	2	3	4			
---	---	---	---	---	--	--	--

**Step 7:** The first element of the queue, formed so far, is 5 (as shown previously). The element is popped out and marked as visited, and all its neighbors are put in the queue.

**Queue:**

6	7
---	---

**Visited:**

0	1	2	3	4	5		
---	---	---	---	---	---	--	--

**Step 8:** The first element of the queue, formed so far, is 6, as shown previously. The element is popped out and marked as visited, and all its neighbors are put in the queue.

**Queue:**

7
---

**Visited:**

0	1	2	3	4	5	6	
---	---	---	---	---	---	---	--

**Step 9:** The first element of the queue formed so far is 7, as shown previously. The element is popped out and marked as visited, and all its neighbors are put in the queue. Note that the queue is now empty after this step. The visited array contains [0, 1, 2, 3, 4, 5, 6, 7]

**Queue:**

--

**Visited:**

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

The following code shows the implementation of BFS:

**Code:**

```
1. from collections import defaultdict
2. class Graph:
3.     def __init__(self):
4.         self.graph = defaultdict(list)
5.     def insertEdge(self, u, v):
6.         self.graph[u].append(v)
7.     def BFS(self, source, N):
8.         # All the nodes are initially marked as 'not visited'
9.         visitedNodes = [False] * N
10.        # queue1 is a queue for implementing BFS
11.        queue1 = []
12.        # The source node is marked as 'visited' and inserted in
queue1
13.        queue1.append(source)
14.        visitedNodes[source] = True
15.        while queue1:
16.            # From queue1 take out a node and print it
17.            source = queue1.pop(0)
18.            print(source, end = " ")
19.            #Find all the adjacent vertices of the vertex just removed.
20.            #For all such nodes which have not been visited mark visited
and put in queue1
21.            # visited and enqueue it
22.            for i in self.graph[source]:
23.                if visitedNodes[i] == False:
```

```
24.         queue.append(i)
25.         visitedNodes[i] = True
```

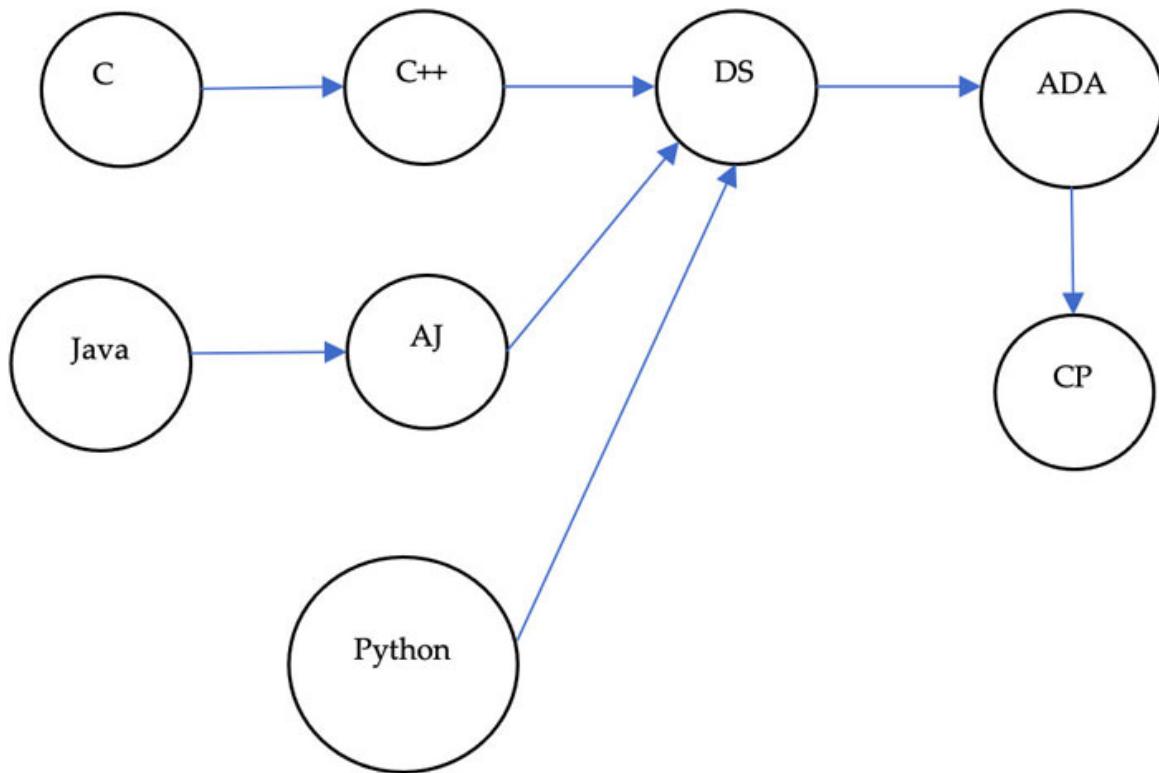
### Output:

0 1 2 3 4 5 6 7

So far, we have seen the traversals of directed graphs. Now, let us move to Directed Acyclic Graphs and explore one of the most important traversal algorithms of such graphs.

### Topological sort

The topological sort generates sequence(s) in which every vertex appears before vertices succeeding it. For example, consider the **Directed Acyclic Graph(DAG)** shown in [figure 11.14](#), in which nodes represent subjects and edges represent the prerequisites. That is, if there is an edge from A to B, then a student must complete A before they start with B:



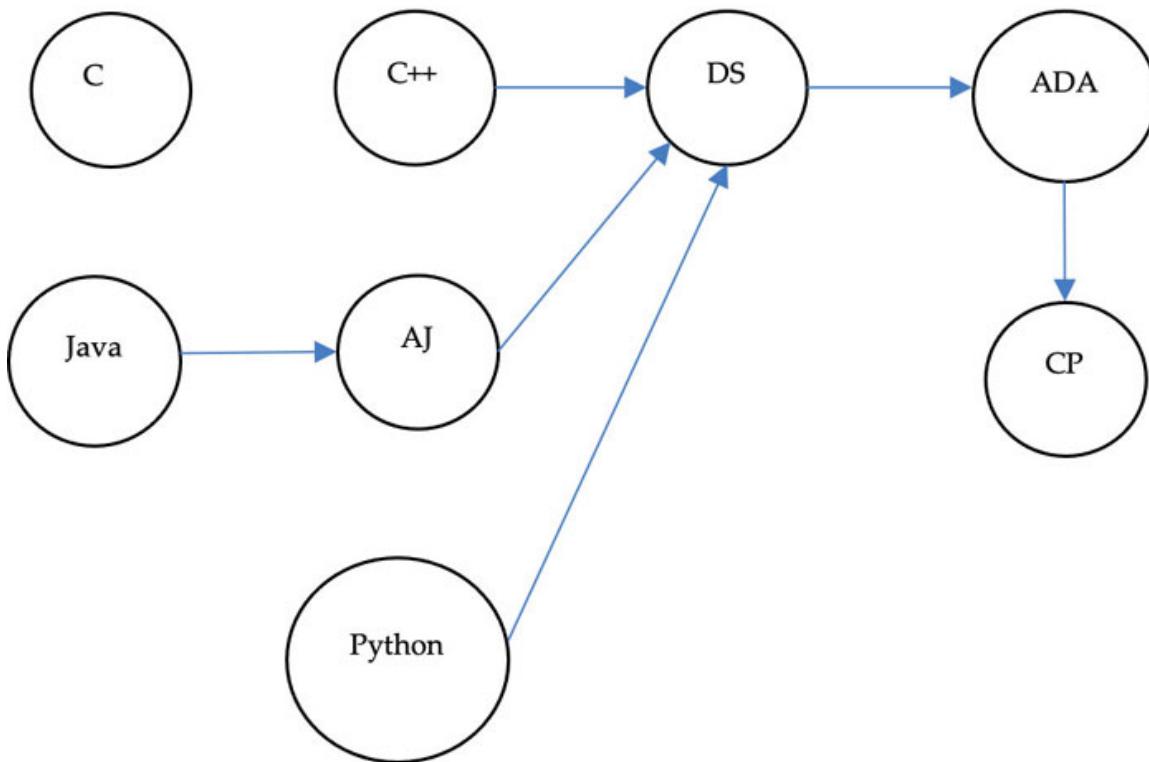
*Figure 11.14: Graph for topological sort*

The following steps may be followed to find the topological sort of a given graph:

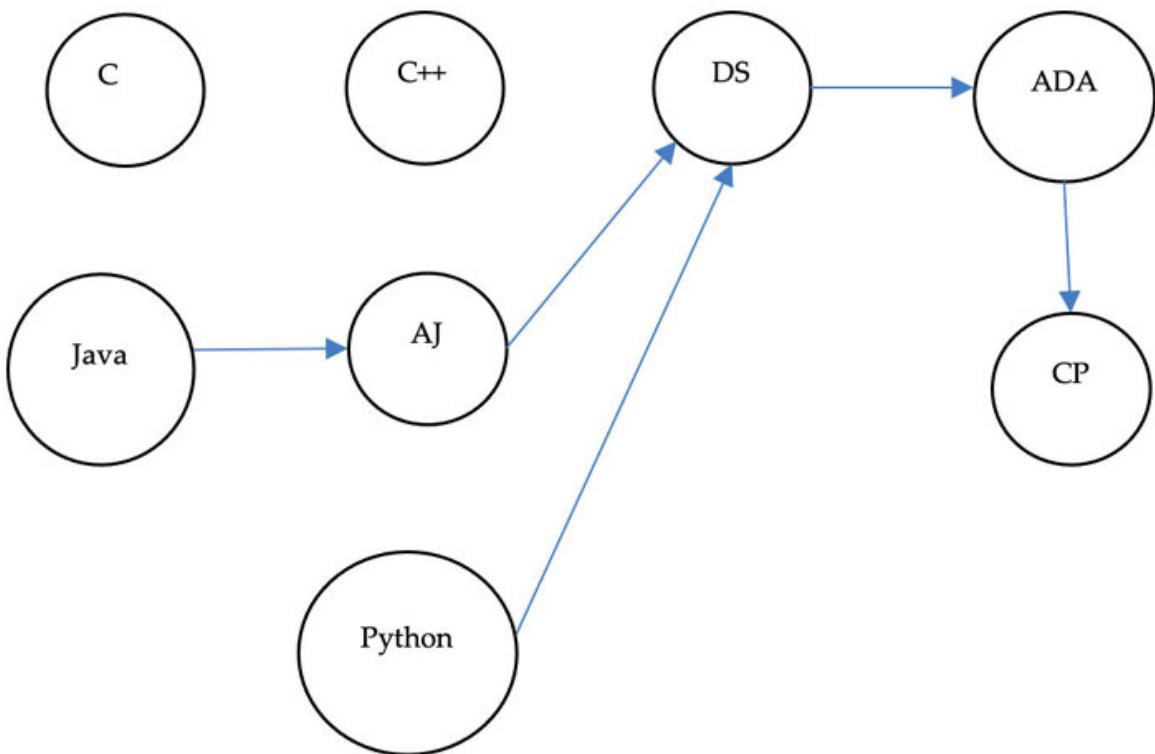
- Search for the node that has no incoming edge.

- Process it, and from that node, remove all outgoing edges.
- Repeat Steps 1 and 2 for the remaining graph till no node is left.

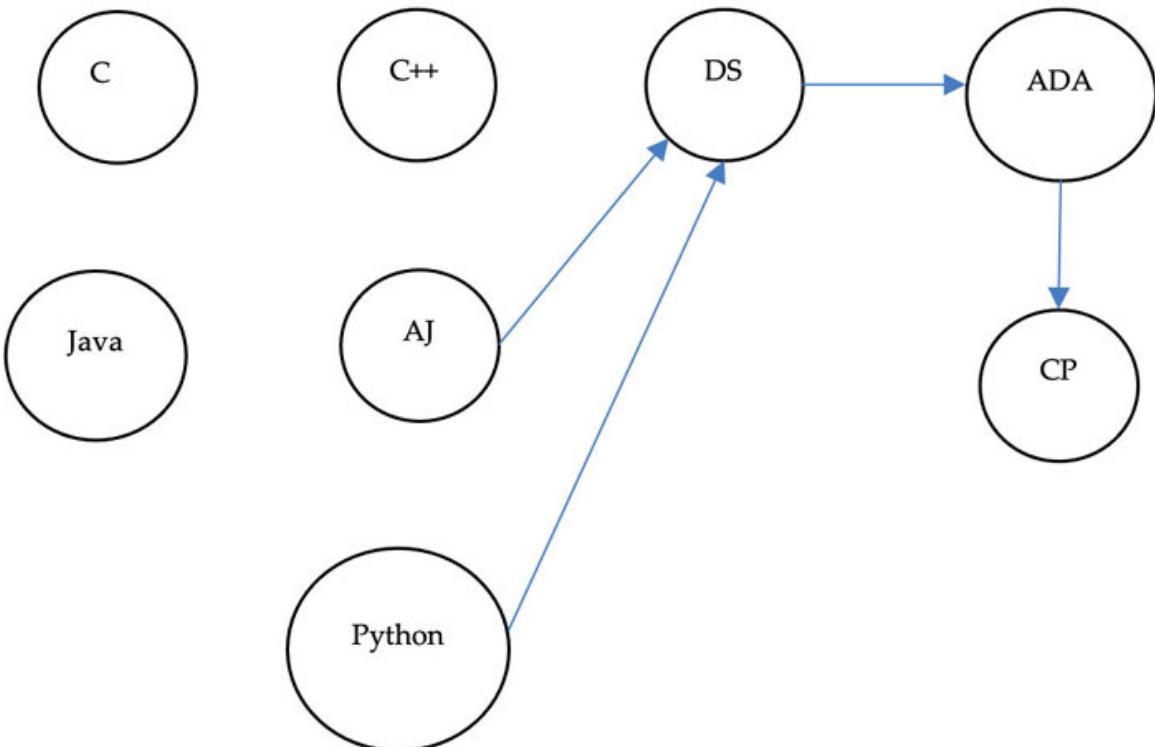
For example, in the graph shown in [figure 11.14](#), C is one of the vertices which has no incoming edge. From this node, all the outgoing edges are removed ([figure 11.15\(a\)](#)). This is followed by selecting C++ ([figure 11.15\(b\)](#)), JAVA ([figure 11.15\(c\)](#)), Advanced JAVA ([figure 11.15\(d\)](#)), Python ([figure 11.15\(e\)](#)), Data structures ([figure 11.15\(f\)](#)), ADA ([figure 11.15\(g\)](#)), and finally, Competitive Programming.



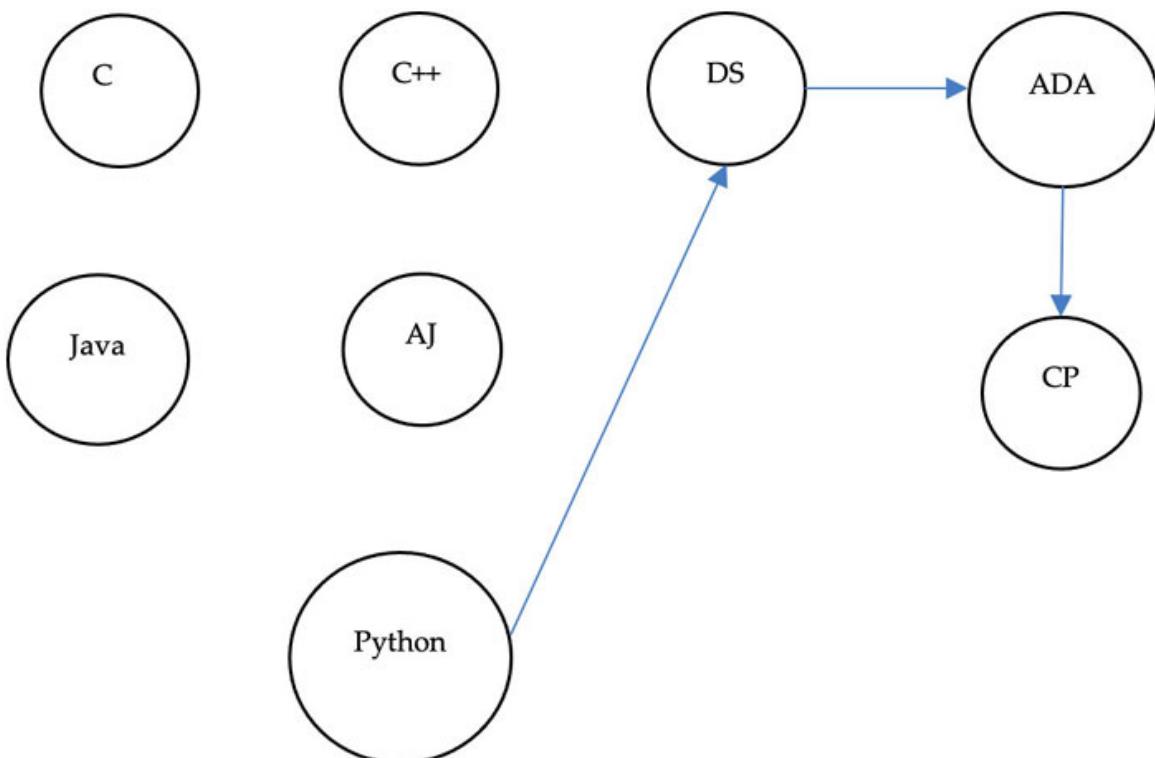
**Figure 11.15(a): Subjects and prerequisites**



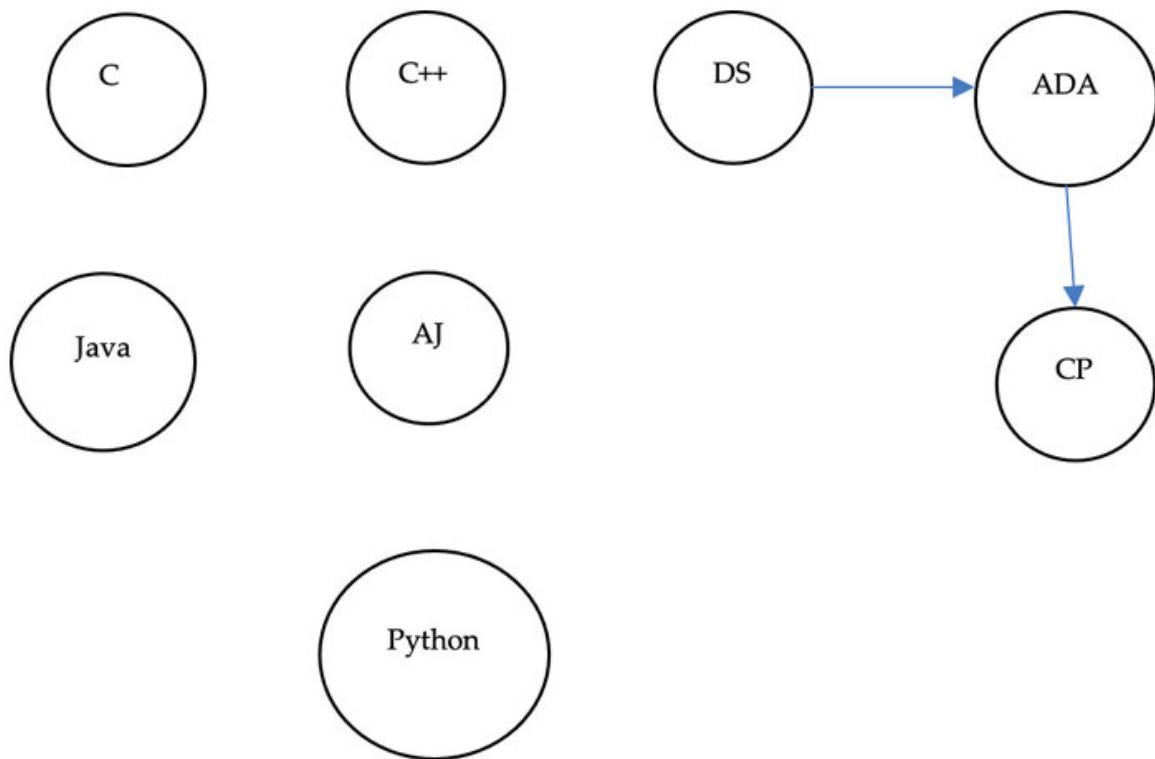
**Figure 11.15(b): Select C, followed by C++**



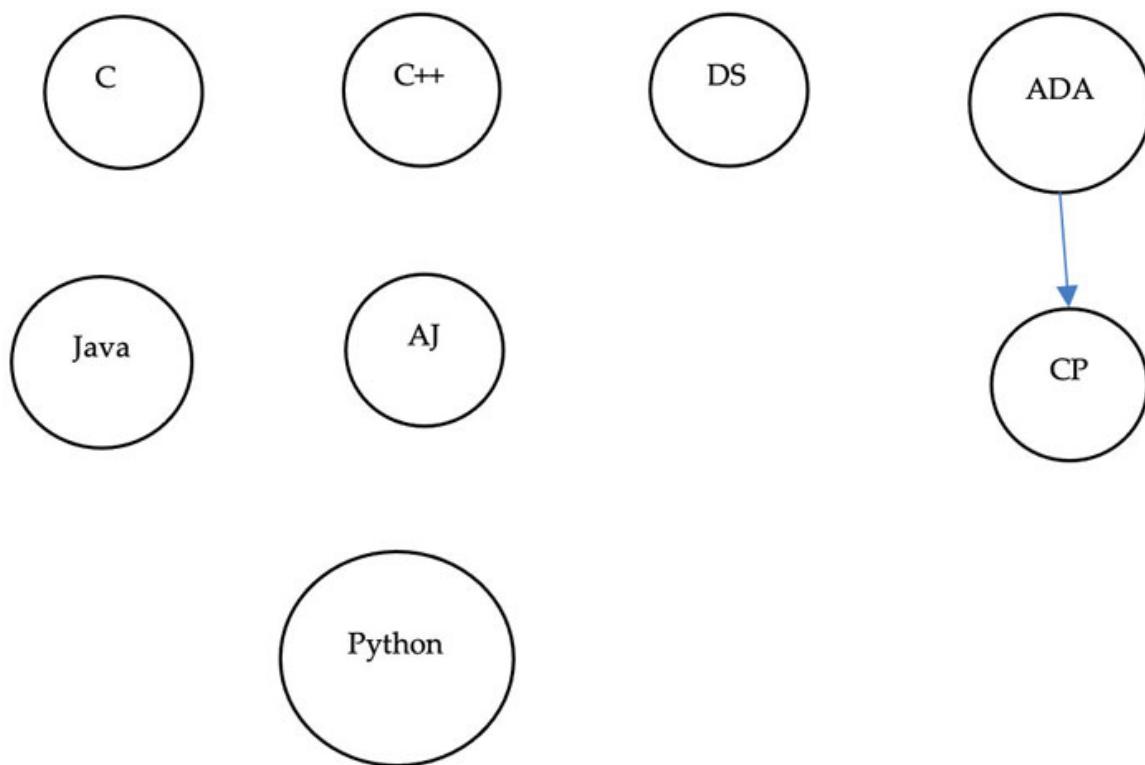
**Figure 11.15(c): Select Java**



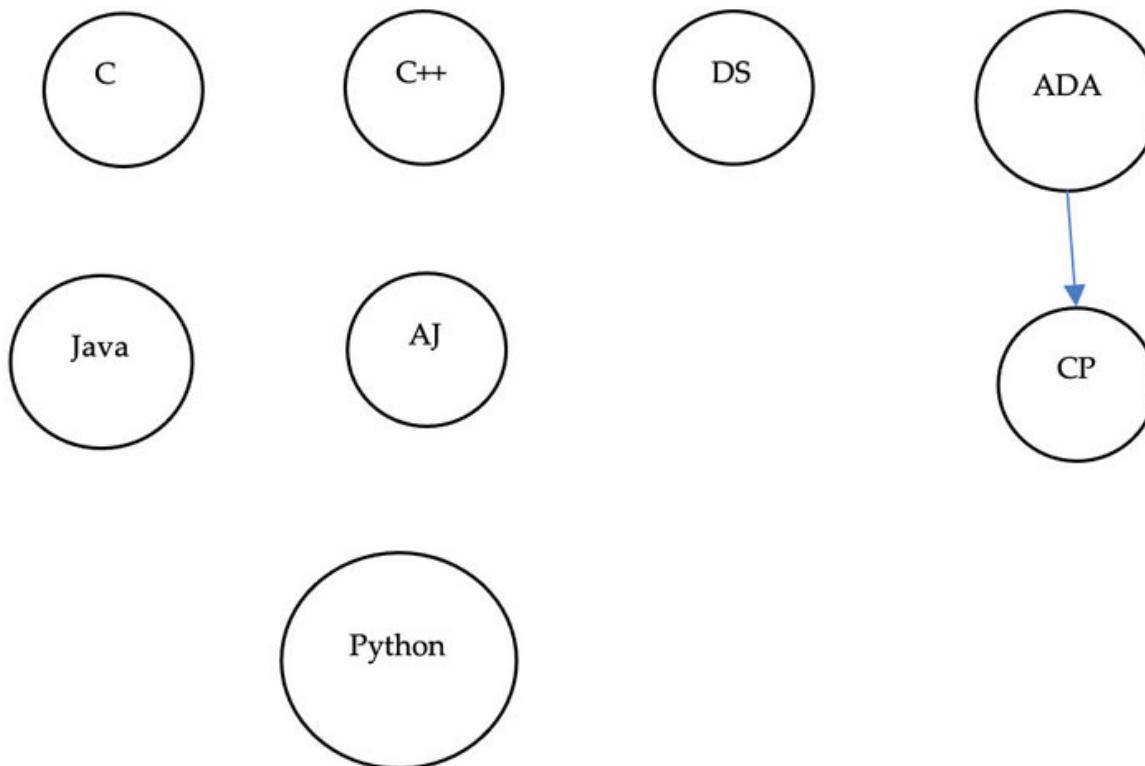
**Figure 11.15(d): Select AJ**



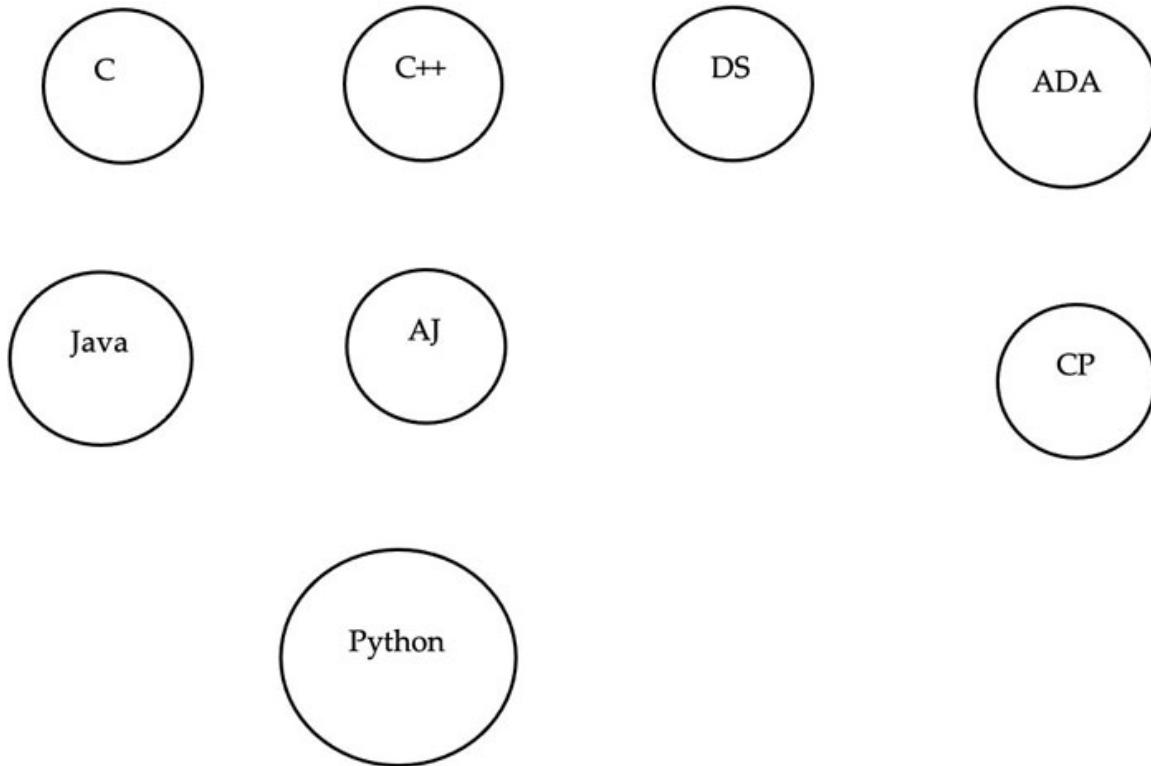
**Figure 11.15(e): Select Python**



**Figure 11.15(f): Select DS**



**Figure 11.15 (g): Select ADA**

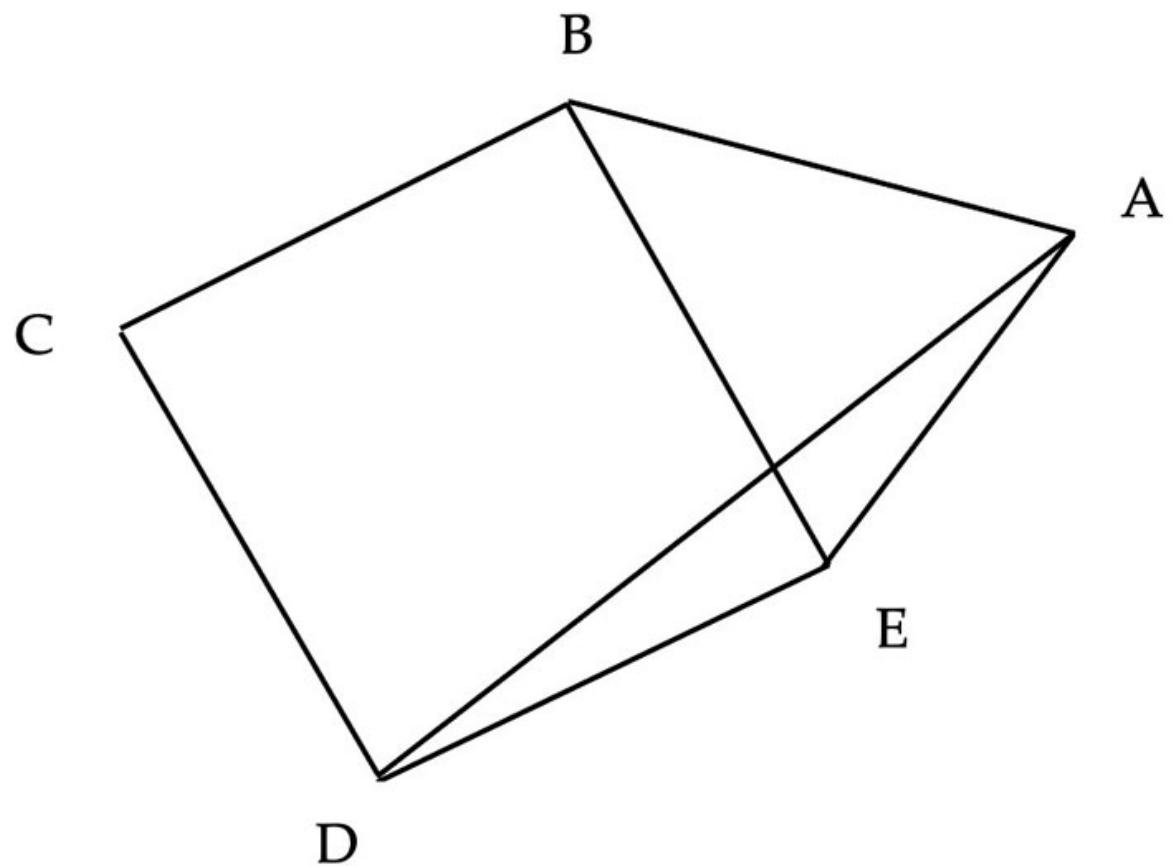


*Figure 11.15 (h): Select CP*

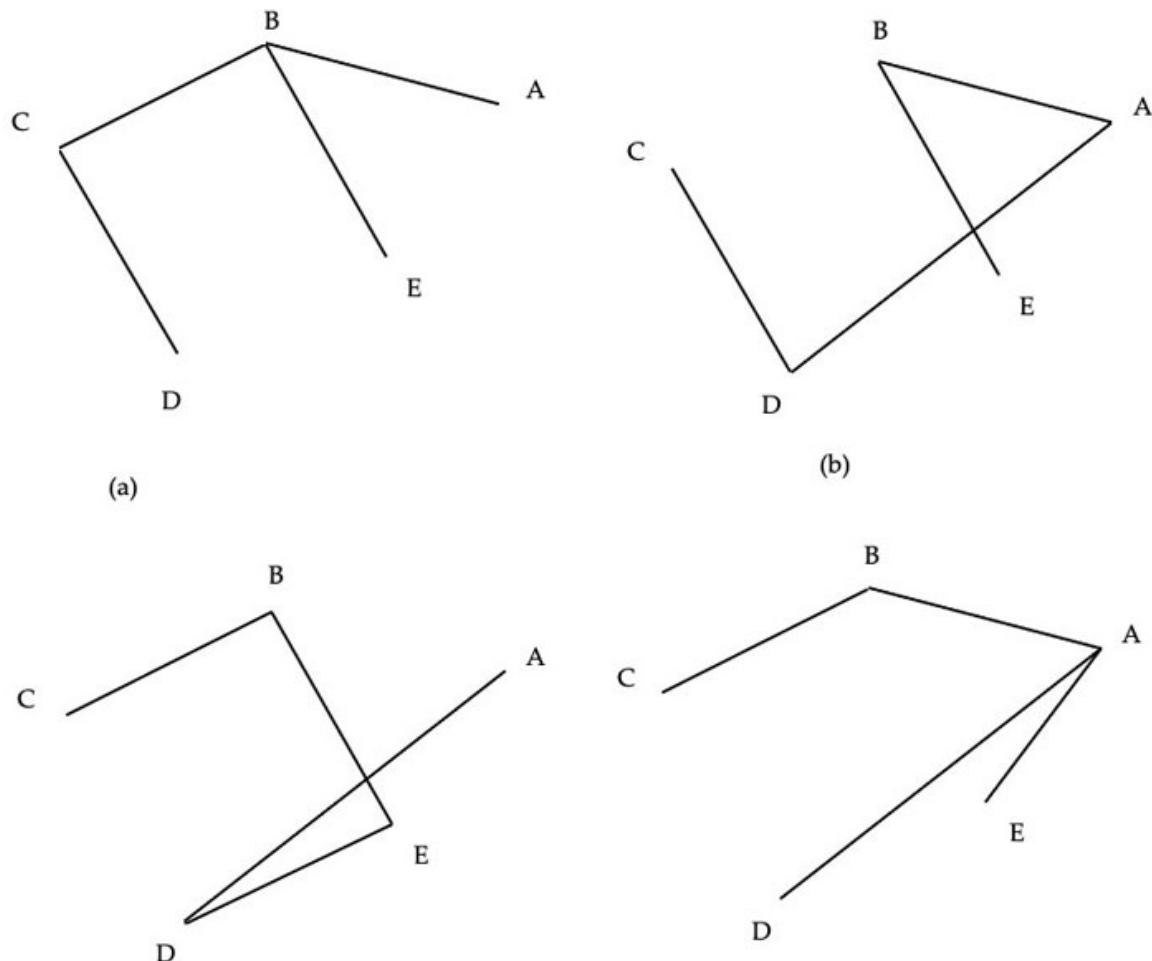
One of the most important applications of Graphs is finding the shortest path. The next section introduces Spanning Trees and discusses one of the most important algorithms for finding the minimum cost spanning tree.

## Spanning tree

Note that a tree is a graph having no cycle or no isolated edge or vertex. A spanning tree of a graph is a tree having  $(n-1)$  edges. For example, consider a graph shown in [figure 11.16](#); some of the spanning trees of this graph are shown in [figure 11.17](#). Note that each spanning tree has four edges.



*Figure 11.16: Graph*

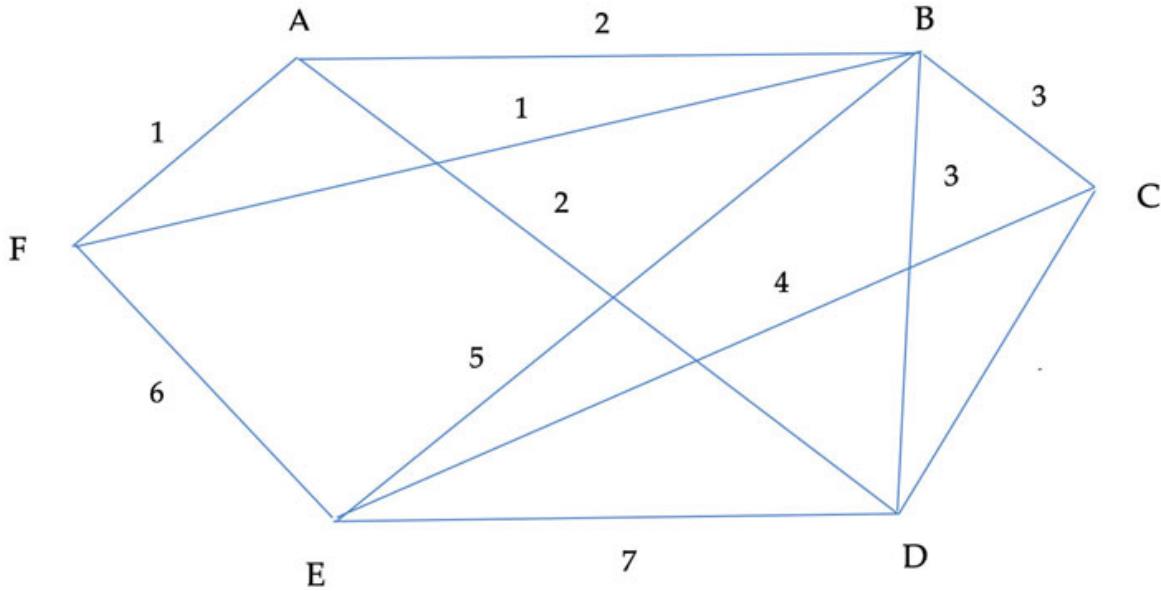


*Figure 11.17: Spanning trees for graph in [figure 11.16](#)*

Now, if a weighted graph is given as input, then each possible spanning tree will have some cost associated with it. We need to find a spanning tree having minimum cost. The following algorithms will help us to find the minimum cost-spanning trees:

### Kruskal's algorithm

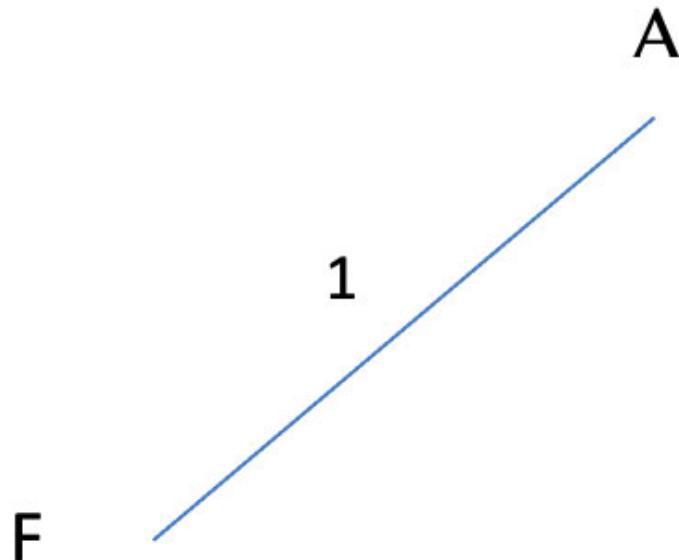
Kruskal's algorithm finds the minimum cost-spanning tree of a given weighted graph using the Greedy Approach. The process starts with arranging the edges in non-descending order. This is followed by picking an edge at each step, provided it does not form a cycle. This process continues till  $(n-1)$  edges are selected; else, no spanning tree is possible for the given graph.



**Figure 11.18:** Weighted Graph A is the source

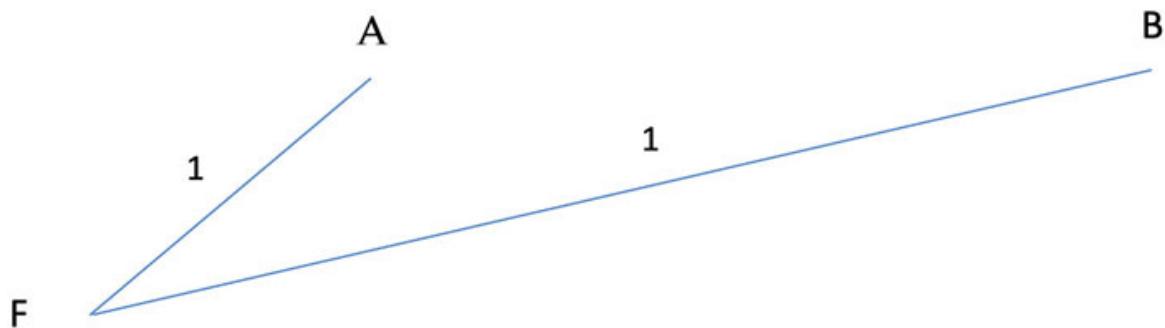
For the graph shown in [figure 11.18](#), the edges in the sorted order are {AF, FB, AB, AD, BC, BD, CD, EC, BE, FE, ED}. Initially, the spanning tree does not contain any edge.

**Step 1:** Take edge AF and add it to the spanning tree ([figure 11.19](#)):



**Figure 11.19:** Figure for Step1, minimum cost spanning tree

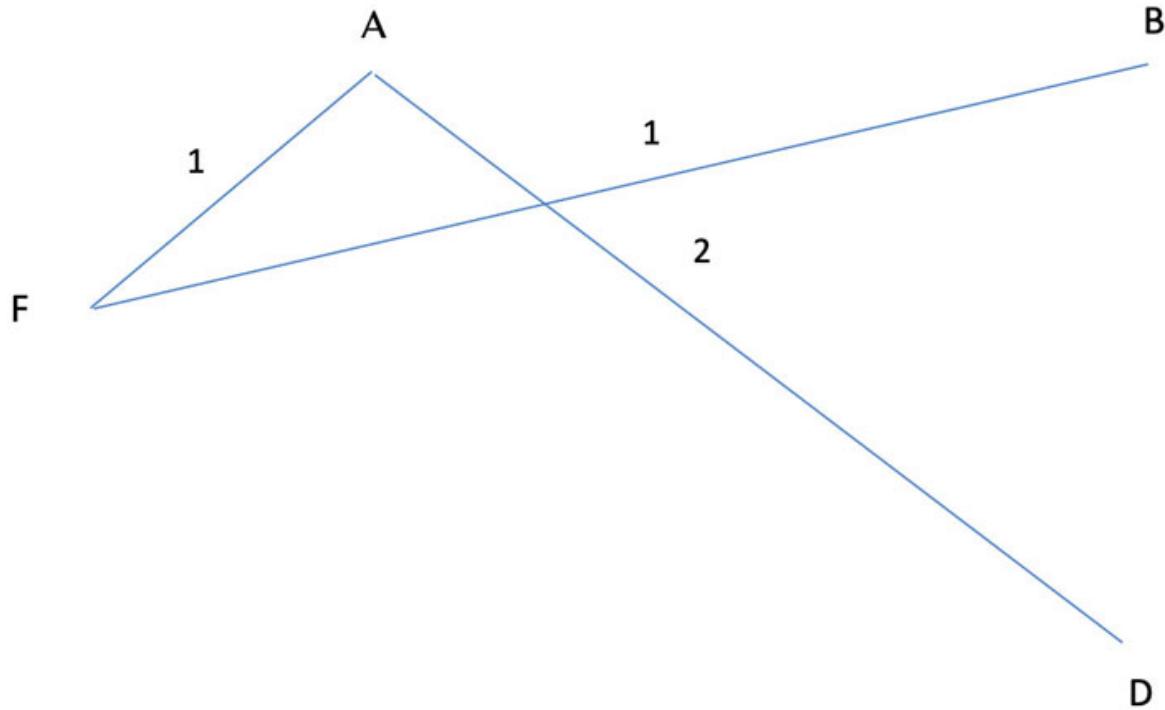
**Step 2:** Take edge FB and add it to the spanning tree since it does not form a cycle with the spanning tree created so far, as shown in [figure 11.20](#):



**Figure 11.20:** Figure for Step2, minimum cost spanning tree

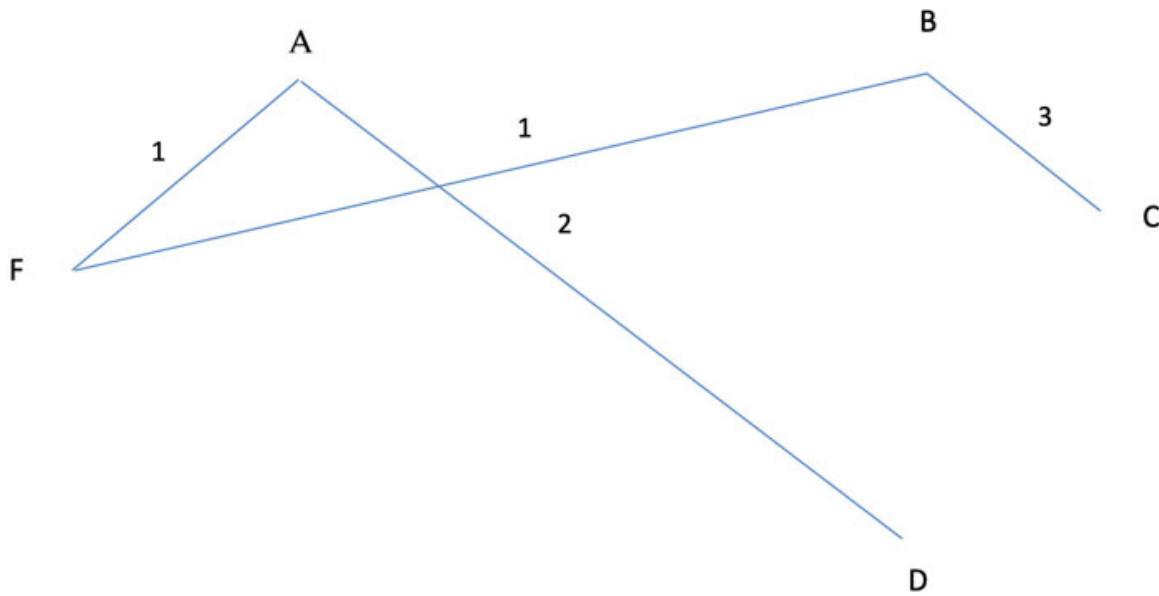
**Step 3:** Take edge AB and add it to the spanning tree. Since it will form a cycle with the spanning tree formed so far, so it is rejected.

**Step 4:** Take edge AD and add it to the spanning tree since it does not form a cycle with the spanning tree created so far, as shown in [figure 11.21](#):



**Figure 11.21:** Figure for Step4minimum cost spanning tree

**Step5:** Take the next edge, that is, BC, and add it to the spanning tree as shown in [figure 11.22](#):

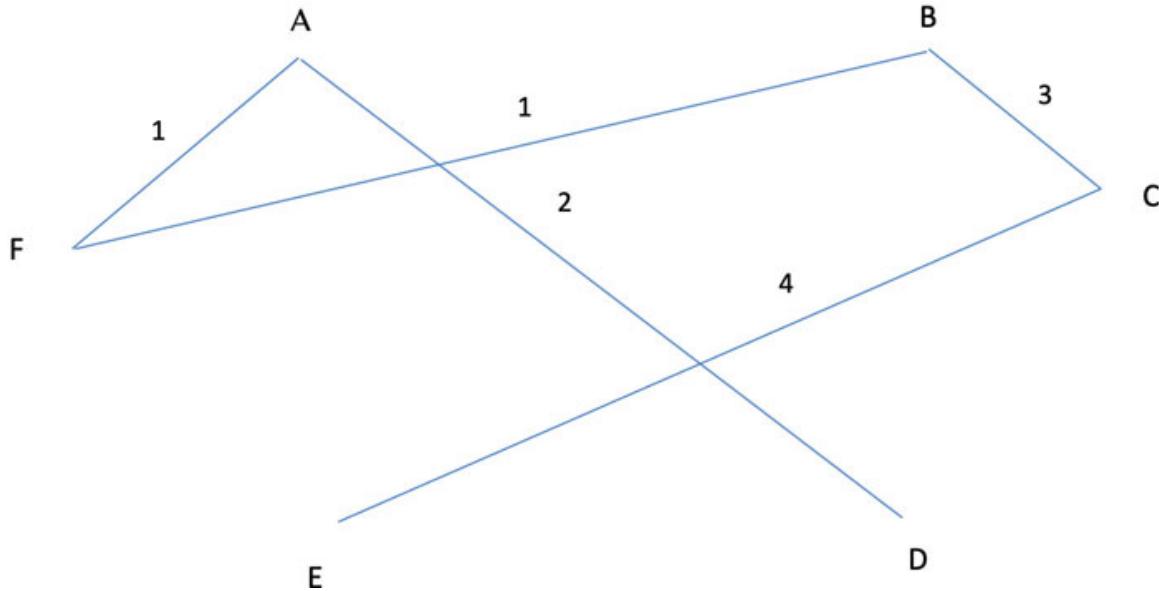


**Figure 11.22:** Figure for Step 5 minimum cost spanning tree

**Step 6:** Take edge BD and add it to the spanning tree. Since it will form a cycle with the spanning tree formed so far, so it is rejected.

**Step 7:** Take the edge CD and add it to the spanning tree. Since it will form a cycle with the spanning tree formed so far, so it is rejected.

**Step 8:** Take edge EC and add it to the spanning tree as shown in [figure 11.23](#):



**Figure 11.23:** Figure for Step 8 minimum cost spanning tree

Since the total number of vertices is 6, the edges in the minimum spanning tree would be 5. Note that the cost of the formed MST is.

The implementation of the algorithm is given in the accompanying Jupyter notebook. The Appendix contains a brief discussion of Prim's algorithm and all pair's shortest paths.

## **Conclusion**

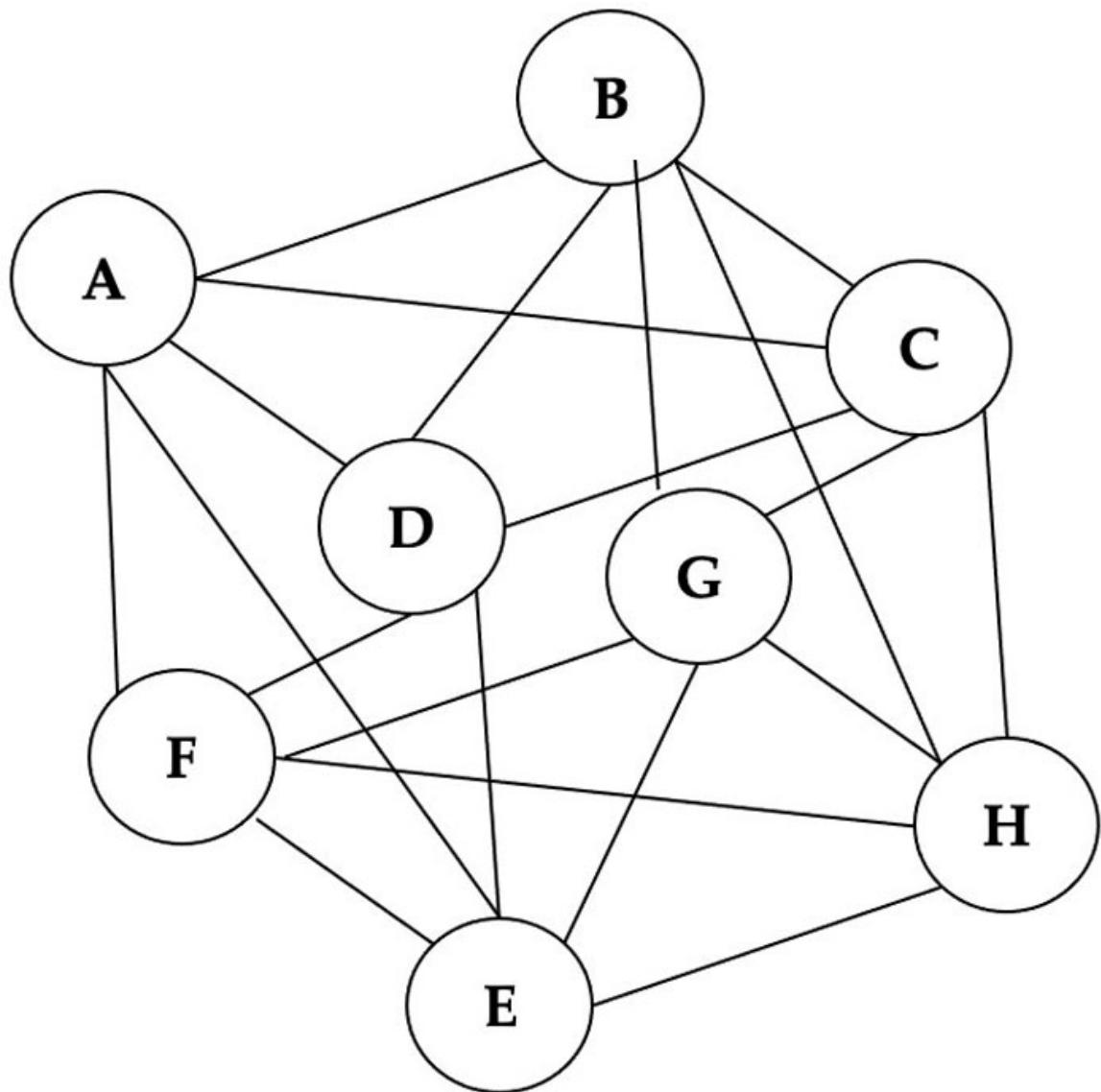
The chapter introduced the reader to Graphs, one of the most important data structures. The chapter discussed the two most important representations of graphs and the issues related to them. This was followed by the Depth First Search, Breadth First Search, and Topological Sorting. The discussions on spanning tree and the minimum cost-spanning trees followed.

The reader will be able to implement the algorithms given in the chapter and use them for solving various problems. Note that some of the topics related to graphs have been included in the Appendix. For programs, one may refer to the accompanying notebooks.

The upcoming chapter discusses sorting and introduces the reader to some of the most amazing algorithms. The chapter discusses  $O(n^2)$ ,  $O(n \log n)$ , and  $O(n)$  algorithms to accomplish sorting. Let us now hit the exercises.

## **Multiple choice questions**

1. For the graph shown in [\*figure 11.24\*](#), which representation is better?



*Figure 11.24: Graph*

- a. Adjacency matrix
  - b. Adjacency list
  - c. Both are appropriate
  - d. None of the above two
2. To implement DFS, which of the following can be used?
- a. Stack
  - b. Queue
  - c. Both

d. Tree

**3. To implement BFS, which of the following can be used?**

- a. Stack
- b. Queue
- c. Both
- d. Tree

**4. Topological sort can be implemented in**

- a. Directed Acyclic Graph
- b. Undirected graph
- c. Cyclic Graph
- d. None of the above

**5. How many edges are there in the spanning tree of a graph having n vertices?**

- a. n
- b.  $(n-1)$
- c.  $(n-2)$
- d. None of the above

**6. How many spanning trees are possible for a fully connected graph having n vertices?**

- a. n
- b.  $n^2$
- c.  $n^{n-2}$
- d. None of the above

**7. Which of the following can be used to find the minimum cost Spanning tree?**

- a. Kruskal's
- b. Prim's
- c. DFS
- d. All of the above

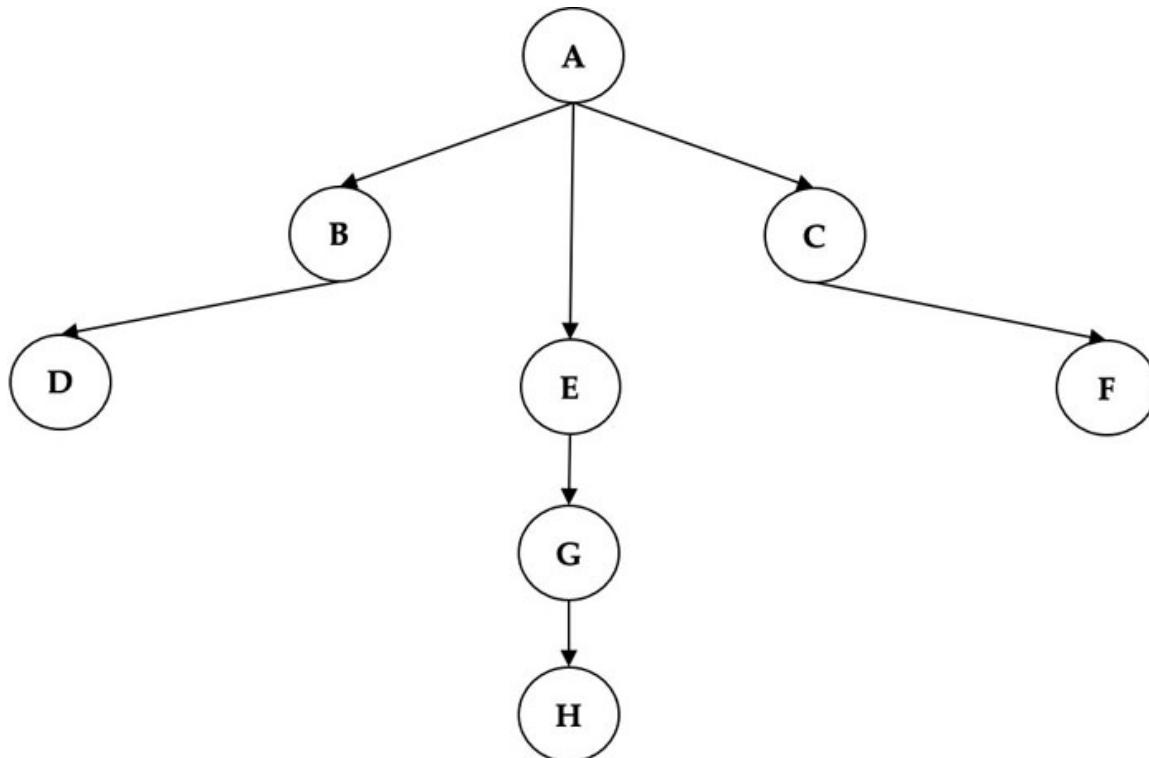
8. Which of the following can be implemented by sorting the edges and then picking the sorted edges one by one, provided they do not form a cycle?

- a. Kruskal's
- b. Prim's
- c. DFS
- d. All of the above

9. A spanning tree

- a. Can have a cycle
- b. Cannot have a cycle

10. Consider the tree shown in [figure 11.25](#). Find two sets U and V, such that they do not have anything in common and the vertices in U do not have any edge between them. Likewise, V does not have any edge between them.



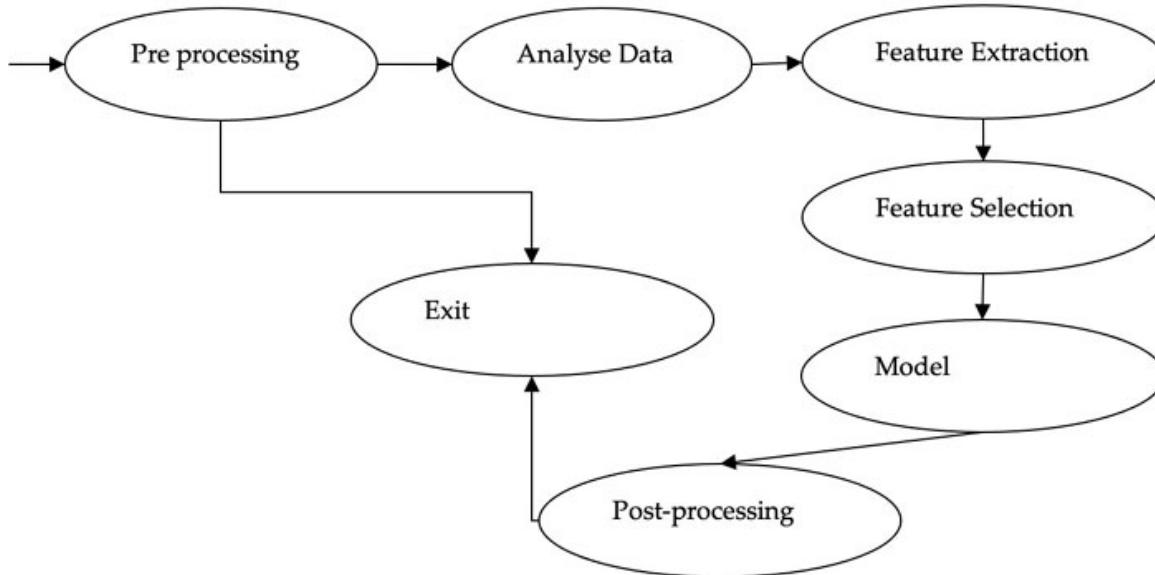
*Figure 11.25: Graph for question 10*

Which of the following is the correct answer for the preceding question?

1.  $U = \{A, D, E, F, H\}$ ,  $V = \{B, C, G\}$
2.  $U = \{A, B, C, F, H\}$ ,  $V = \{D, E, G\}$
3.  $U = \{A, B, D\}$ ,  $V = \{E, G, H, C, F\}$
4. None of the above

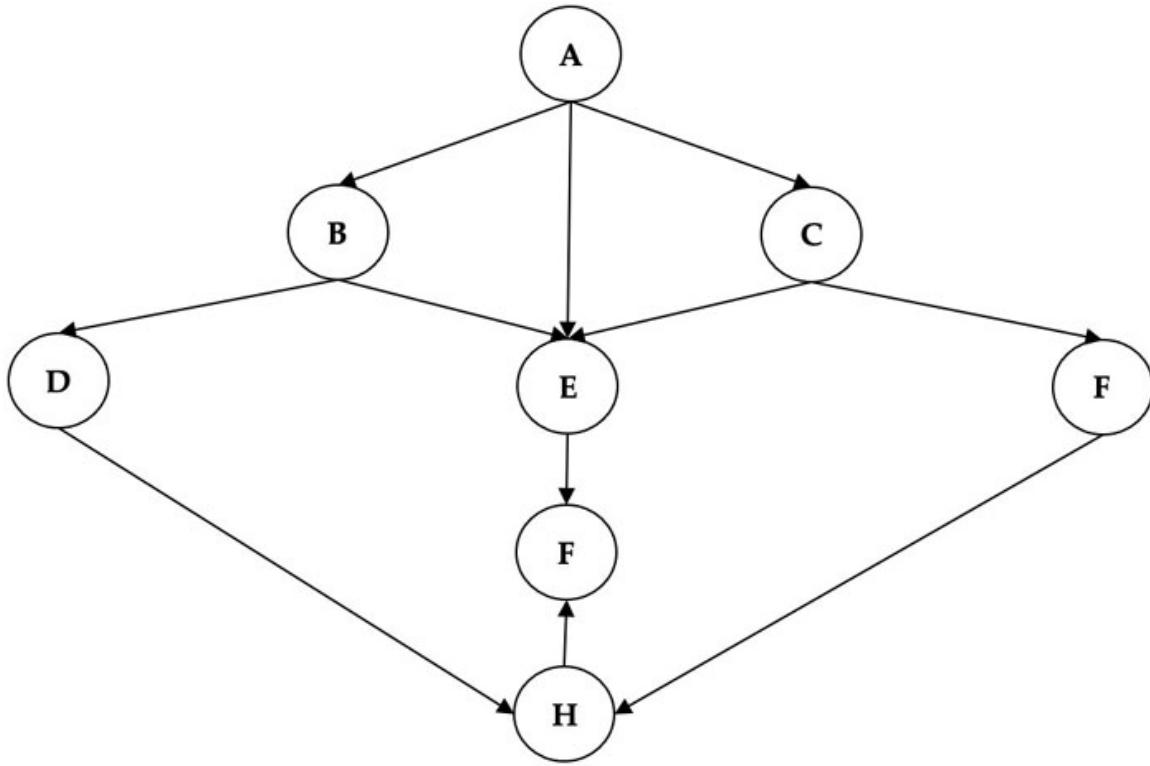
## Numerical/application based

1. Consider a graph, as shown in [figure 11.26](#). The graph shows the steps of the Machine Learning pipeline. The process starts with pre-processing. This is followed by feature extraction and feature selection. The model is crafted, followed by post-processing.



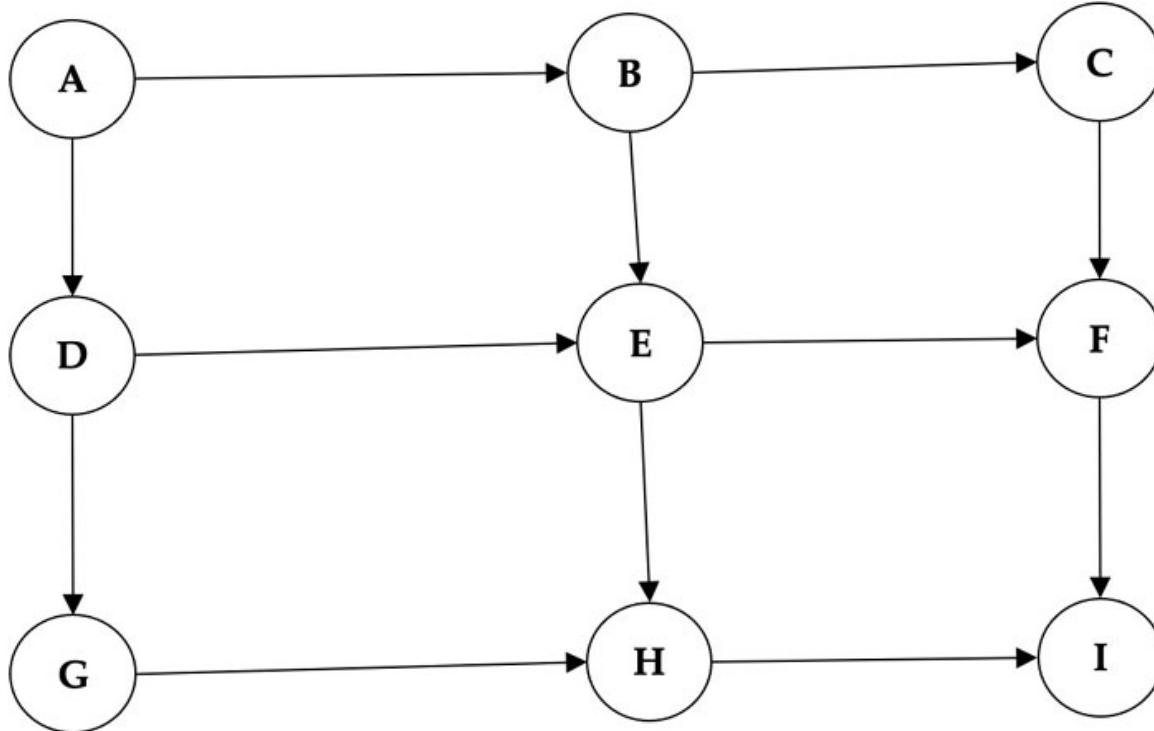
*Figure 11.26: Machine learning pipeline*

- a. Can we apply topological sort to this graph?
- b. Write the Breath First Search traversal of this graph.
- c. Write the Depth First Search traversal of this graph.
2. Consider the graph shown in [figure 11.27](#). Now, consider the sequences [A, B, C, D, E, F, G, H] and [A, B, D, H, G, E, C, F]. State the type of traversals and the sequence of steps to reach these sequences, starting from A.



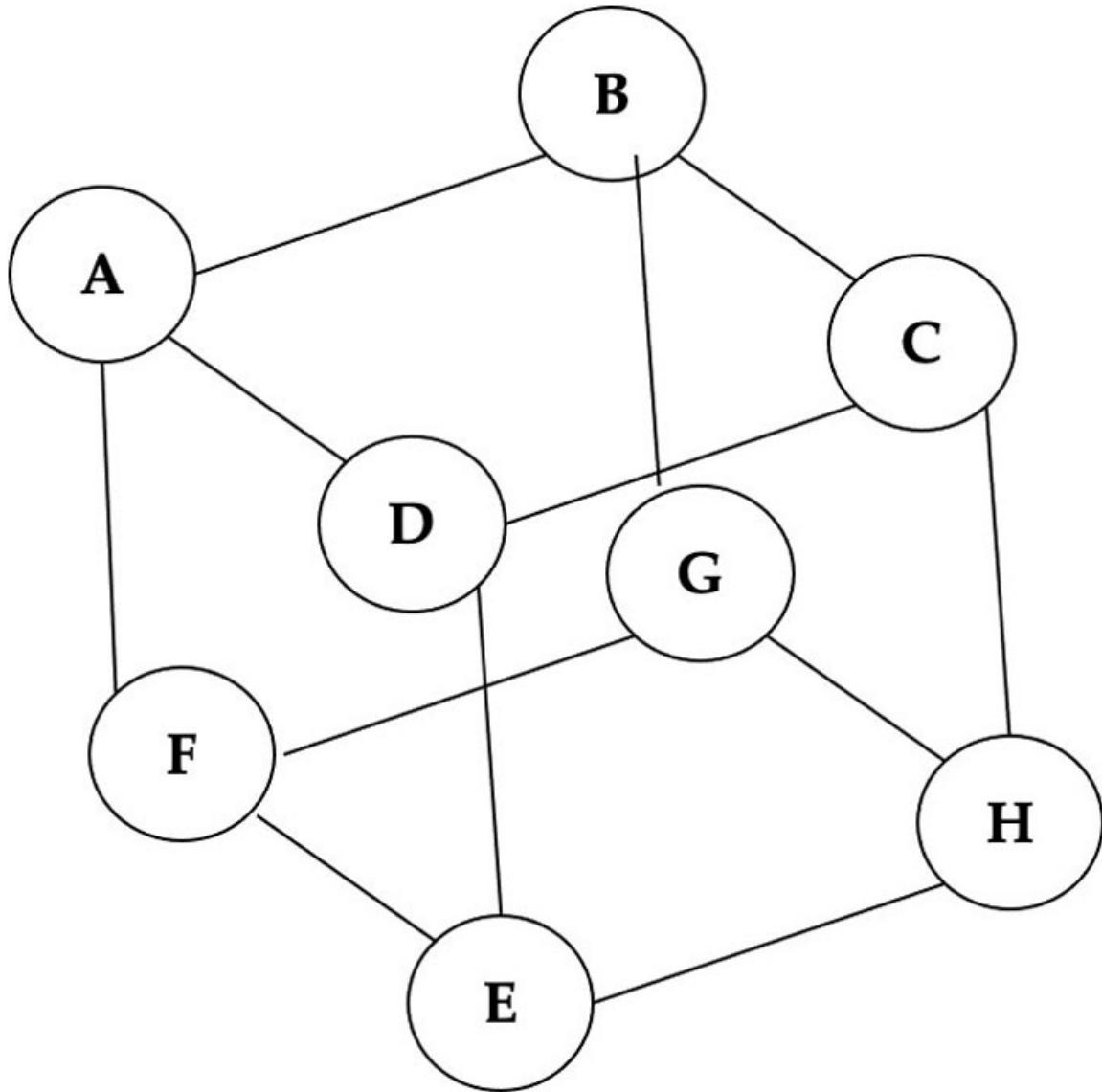
**Figure 11.27:** Graph for problem 2

3. Consider the graph shown in [figure 11.28](#). Find the DFS and BFS of this graph:



*Figure 11.28: Graph for problem 3*

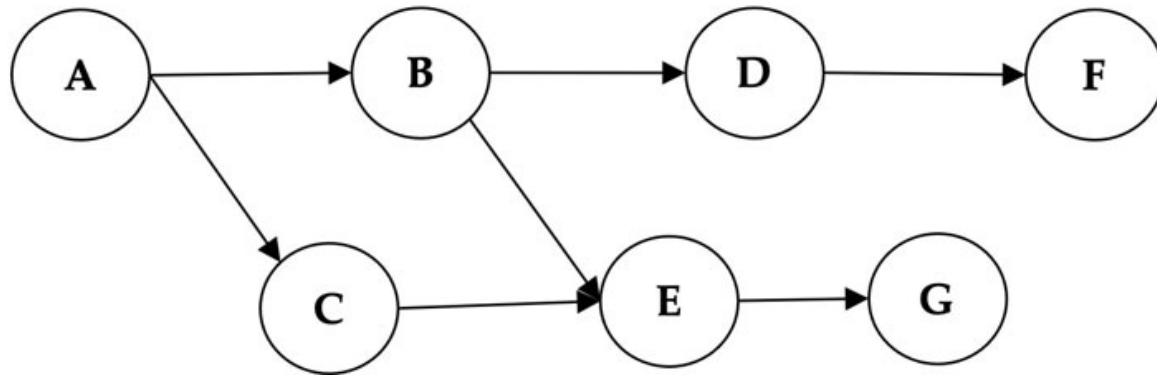
4. Consider the graph shown in [figure 11.29](#). Find the DFS and BFS of this graph.



*Figure 11.29: Graph for problem 4*

Also, write all possible topological sorts of the preceding graph.

5. Represent the graph shown in [figure 11.29](#) using (a) an adjacency matrix and (b) an adjacency list. Which of the two is better? Give reasons in support of your answer.
6. Write all possible topological sorts of the preceding graph shown in [figure 11.30](#):



*Figure 11.30: Graph for problem 6*

7. For the graph shown in [figure 11.31](#), apply Kruskal's Algorithm to show each step:

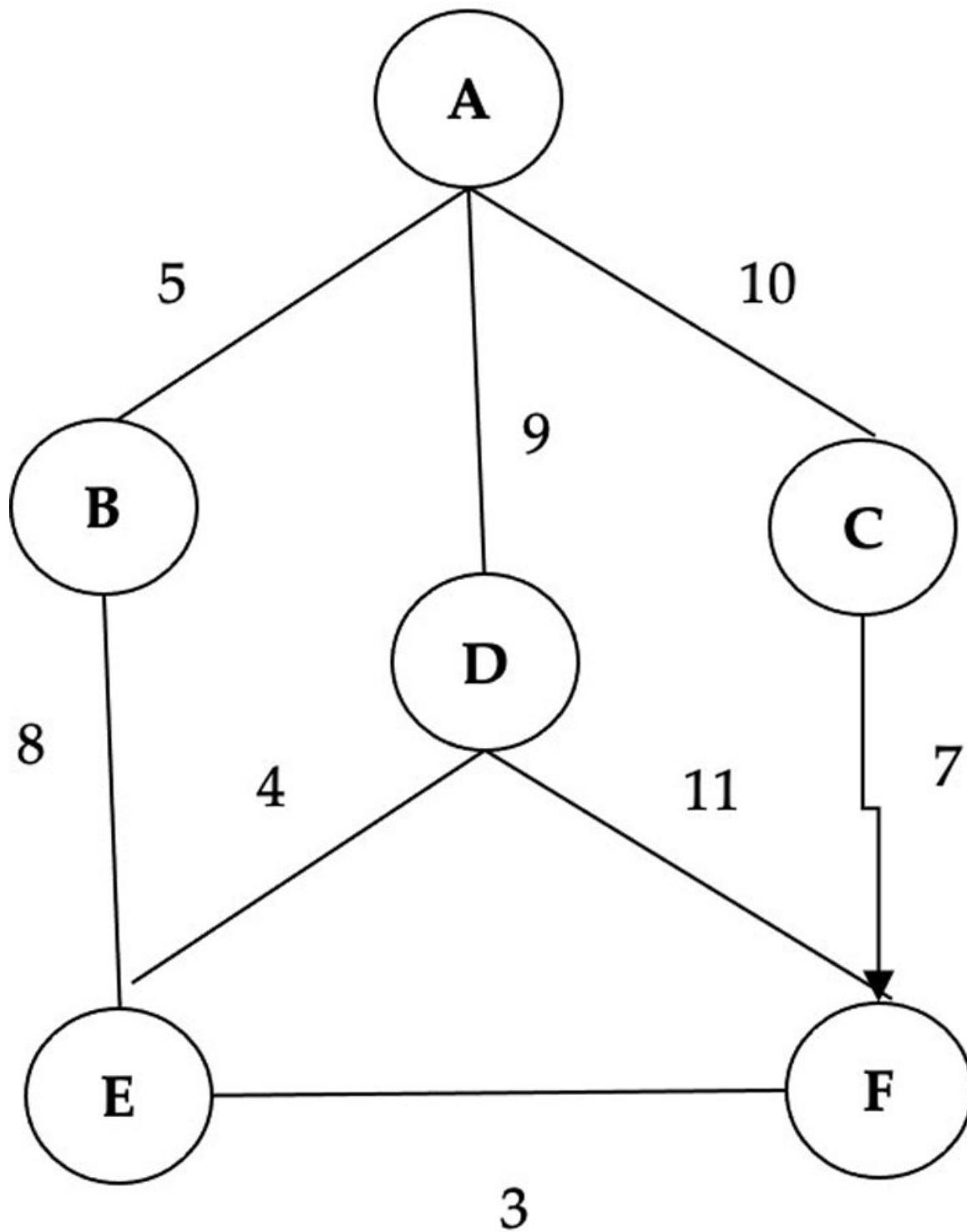


Figure 11.31: Graph for problem 7

8. For the graph shown in [figure 11.29](#), assume each edge has weight 1. Apply Kruskal Algorithm to find the minimum cost-spanning tree. Show each step.

## Programming

1. Write a program to implement Depth First Search.
2. Write a program to implement Breadth First Search.
3. Write a program to implement Kruskal's algorithm.
4. Write a program to implement Prim's algorithm.
5. Write a program to implement Topological Sort.
6. Write a program to find if there is a cycle in the given graph.
7. Write a program to convert an adjacency matrix to an adjacency list.
8. Using any of the preceding representations
  - a. Insert an edge in a given graph.
  - b. Delete an edge in a given graph.
  - c. Find if there is an edge from one vertex to another in a given graph.

## **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 12

## Sorting

Have a look at the contacts of your mobile phone, are they sorted (lexicographic ordering)? Take a moment and think about a scenario wherein these contacts are stored in random order. Can you figure out what will happen in that case? Yes! searching for contact will take more time. Likewise, the list of students in a class, the books in a library, and items in inventory are all sorted in some order to facilitate search. This chapter introduces sorting, which is a building block of Data Structures.

The chapter discusses some common sorting algorithms, their implementations, complexities, and downsides.

This chapter begins with Bubble Sort, one of the simplest sorting algorithms. The number of comparisons in the algorithm can be significantly reduced by a slight change introduced in Comb Sort. The chapter then moves to Selection Sort, which is elegant, but not as good as the next algorithm, which is insertion sort. This is followed by two linear-time complexity algorithms, namely, Radix sort and Counting Sort. We then move to Merge Sort, and finally, we introduce Quick Sort.

This chapter will not only help you strengthen your armor in your battles ahead but will also prove to be mentally stimulating.

### Structure

This chapter covers the following topics:

- Bubble sort
- Comb sort
- Selection sort
- Insertion sort
- Radix sort
- Counting sort

- Merge sort
- Quick sort

## Objectives

After reading this chapter, you will be able to understand, implement, and compare various sorting algorithms. You will also learn the time complexity and downsides of various algorithms.

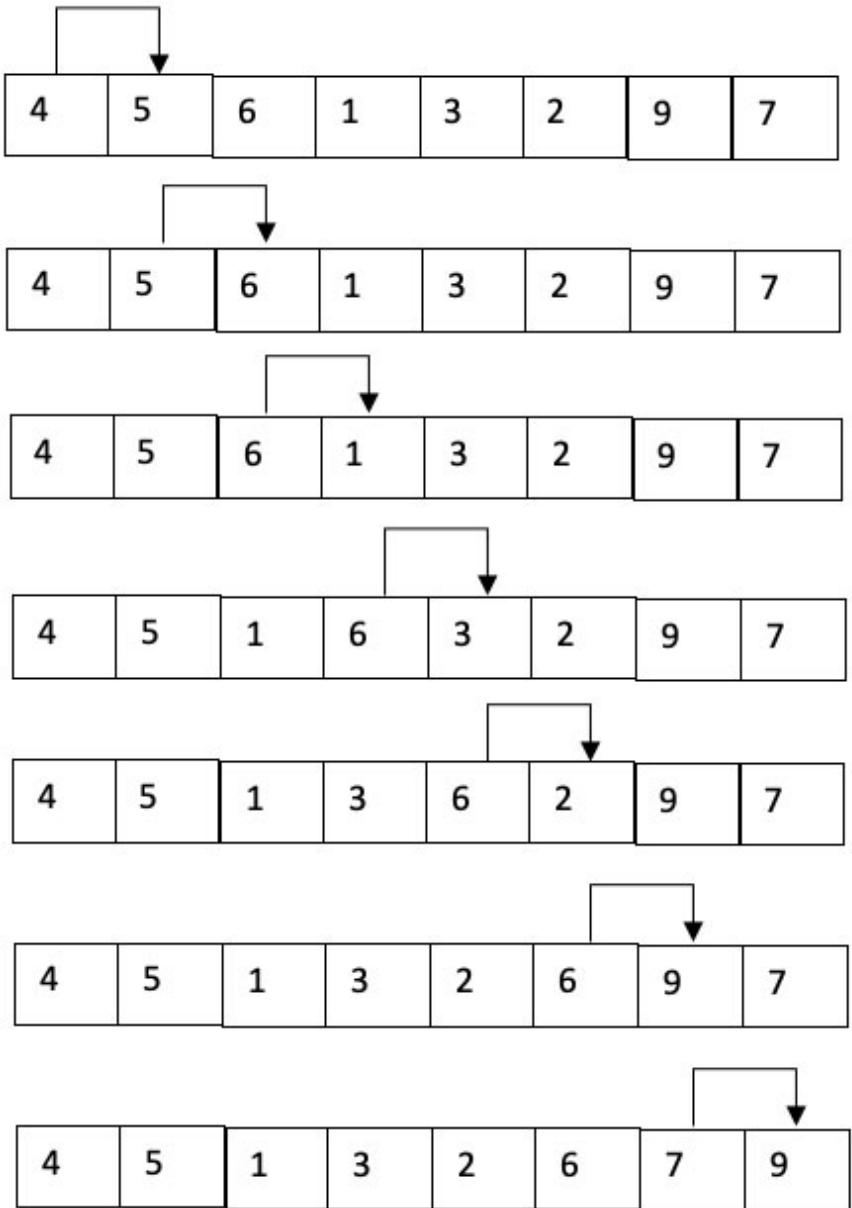
## Bubble sort

In Bubble sort, the number of iterations is  $(n-1)$ , where  $n$  is the number of elements in the array. In the first iteration, the first element is compared with the second, second with the third, third with the fourth, and so on. In the end, the second-last element is compared with the last one. Each comparison leads to swapping the elements being compared if the second element is smaller than the first. This results in the largest element being placed in the last position.

In the next iteration, the first element is compared with the second, second with the third, third with the fourth, and so on. In the end, the third-last element is compared with the second-last. Each comparison leads to swapping the elements if the second element is smaller than the first. This results in the second largest element being placed at the last but one position. Therefore, in the  $i$ th iteration, the largest element is placed at the  $(n-i-1)^{\text{th}}$  position. This process, as stated earlier, is repeated  $(n-1)$  times.

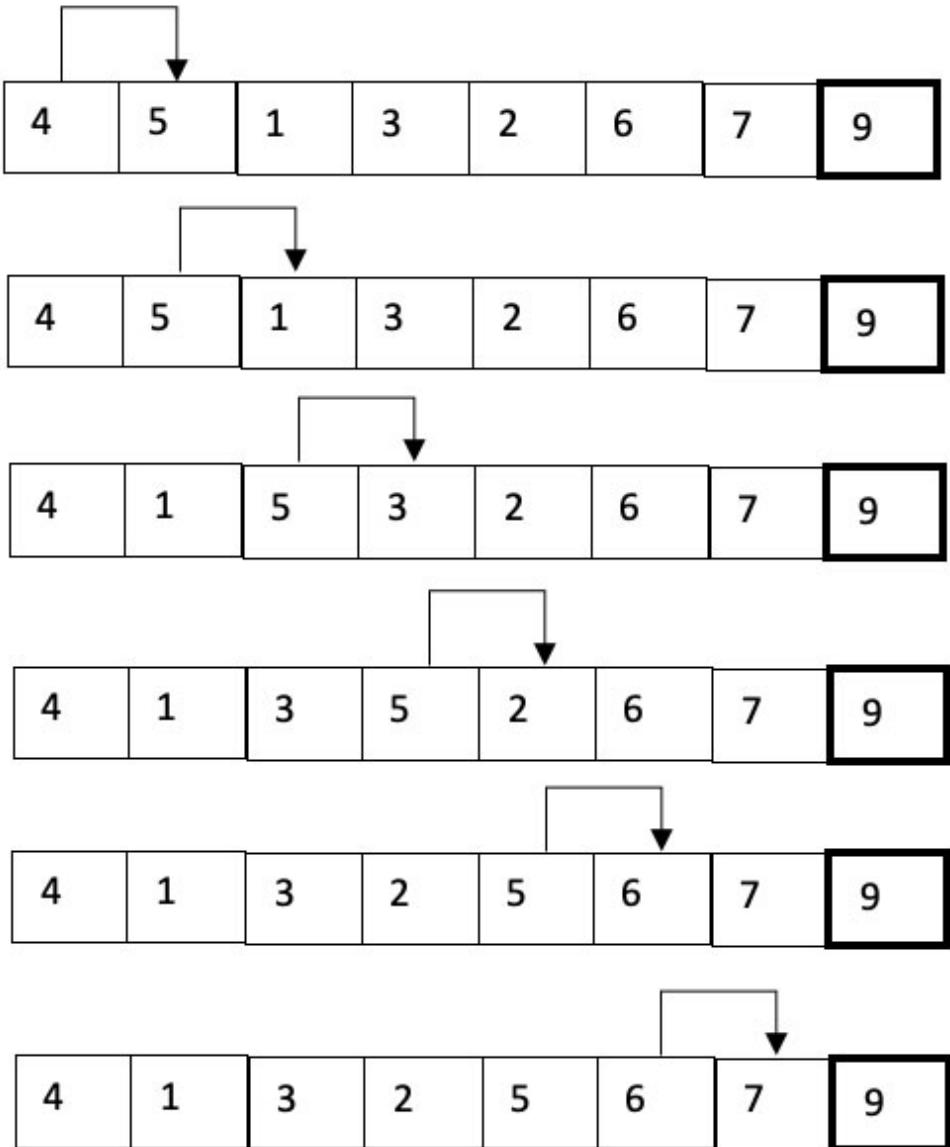
The process is depicted in [figure 12.1 \(a\)](#) to [12.1\(g\)](#). Note that in each iteration, the largest element of the first  $(n-i)$  positions is placed at the  $(n-i-1)^{\text{th}}$  position.

**Iteration 1**



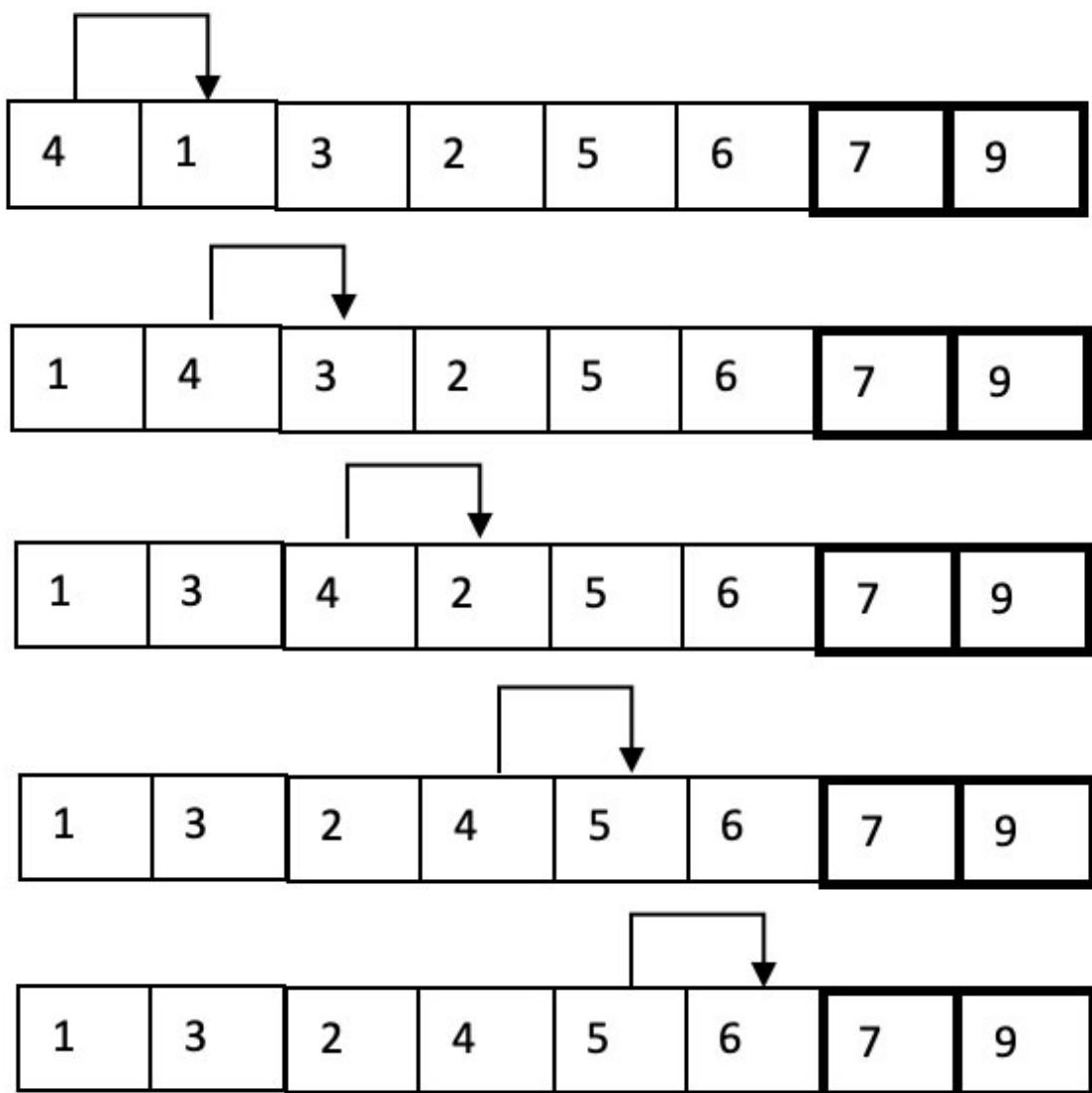
**Figure 12.1 (a): Bubble Sort, Iteration 1**

**Iteration 2**



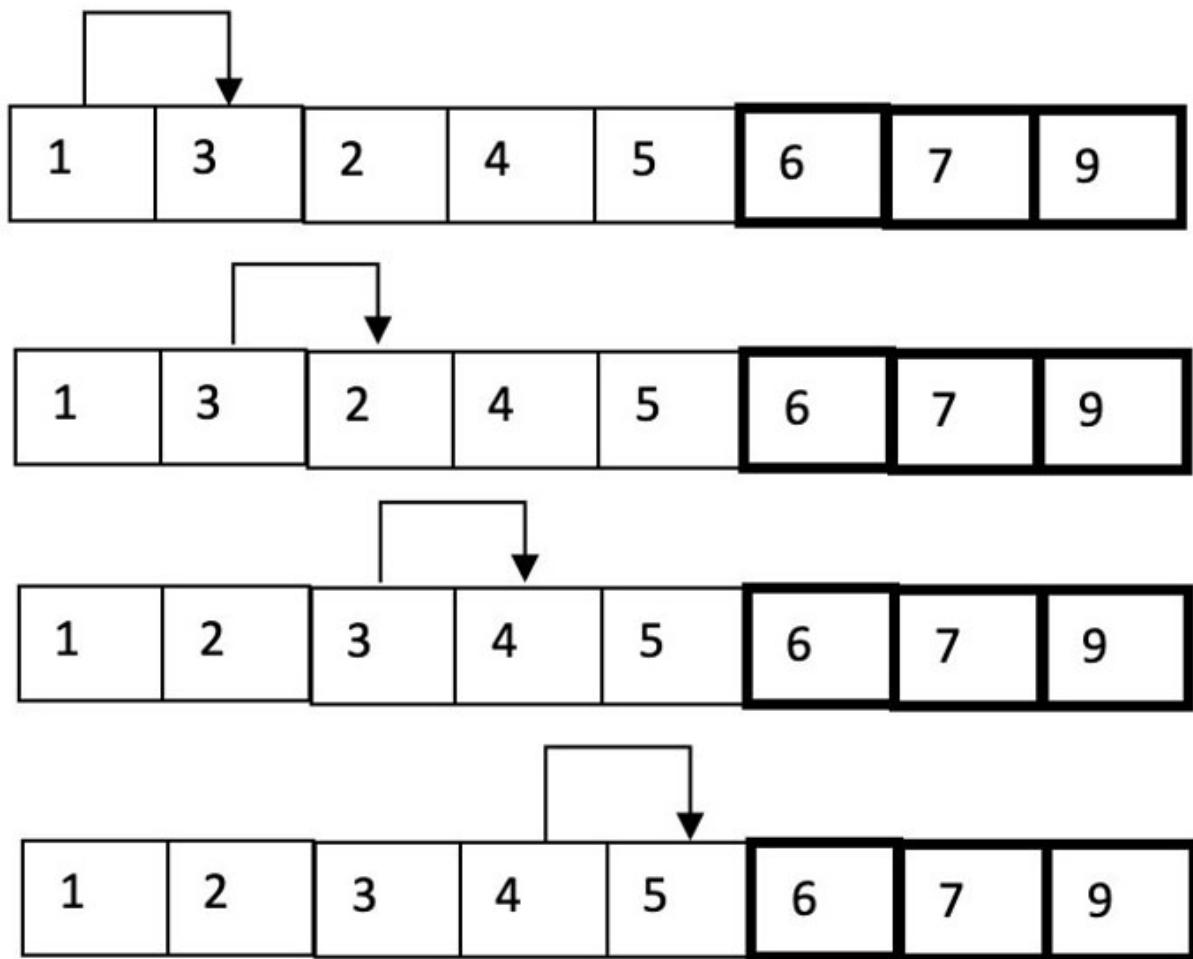
*Figure 12.1 (b): Bubble Sort, Iteration 2*

**Iteration 3**



*Figure 12.1 (c): Bubble Sort, Iteration 3*

**Iteration 4**



*Figure 12.1 (d): Bubble Sort, Iteration 4*

## Iteration 5

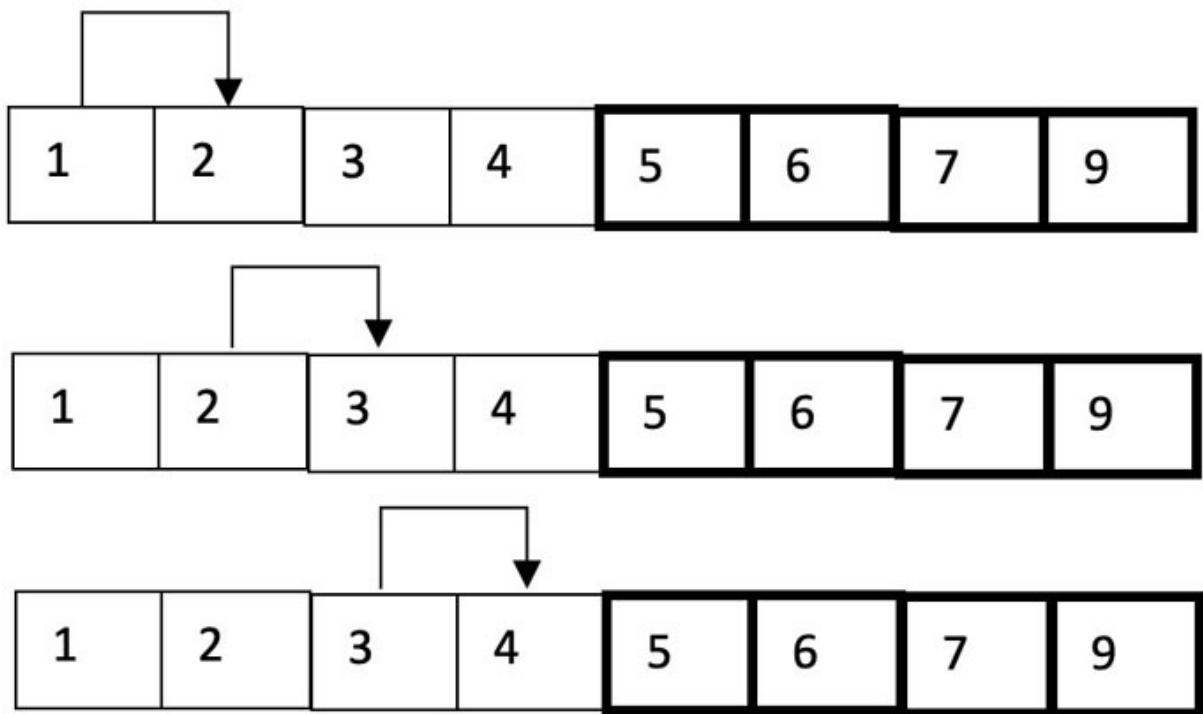
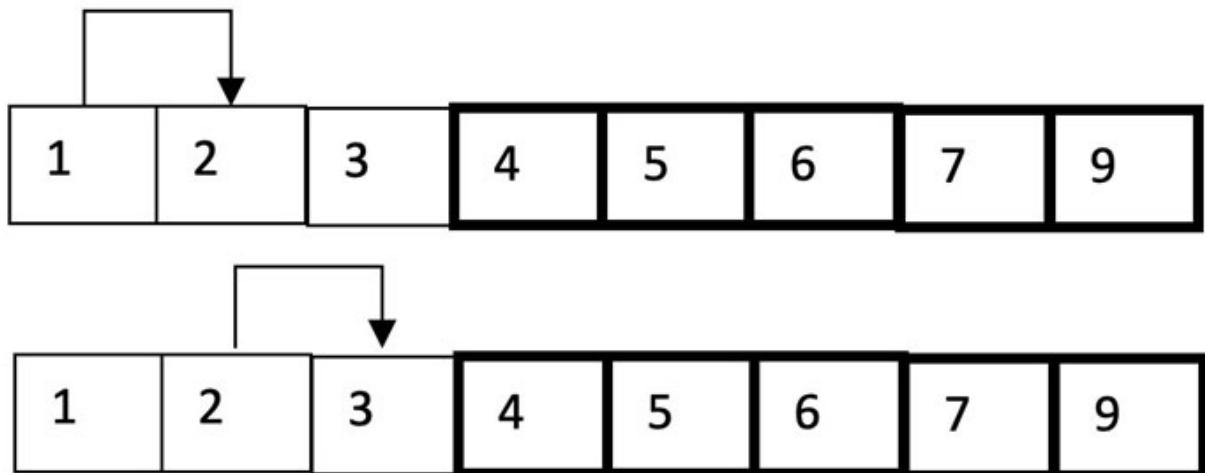


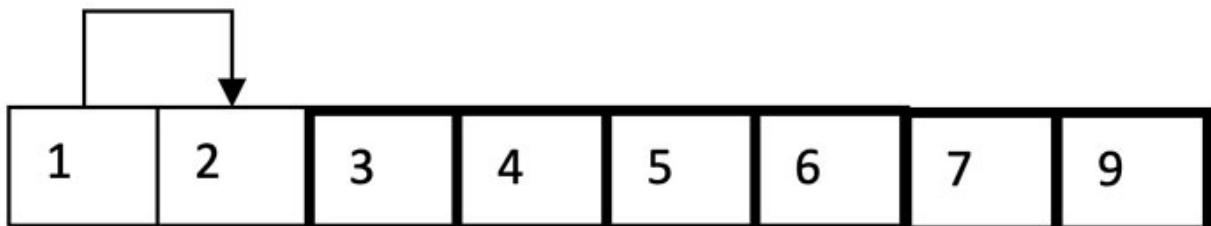
Figure 12.1 (e): Bubble Sort, Iteration 5

## Iteration 6



*Figure 12.1 (f): Bubble Sort, Iteration 6*

## Iteration 7



*Figure 12.1 (g): Bubble Sort, Iteration 7*

[Table 12.1](#) shows the number of comparisons in each iteration and the total number of comparisons in Bubble Sort:

Iteration Number	Comparisons
1	$(n-1)$
2	$(n-2)$
3	$(n-3)$
4	...
	$(n-1) \quad 1$
<b>Total</b>	$n \times \frac{n-1}{2} = \Theta(n^2)$

**Table 12.1:** Number of comparisons in Bubble Sort

Note that the number of swaps depends on the elements of the array itself. In the worst case, each comparison may require a swap, in which case the complexity is  $O(n^2)$ . In the best case, none of them requires a swap, in which case the complexity is  $O(n)$ . The code of Bubble Sort is as follows:

### Code:

```

1. def bubble_sort(arr1):
2.     n=len(arr1)
3.     for i in range(n):
4.         for j in range(0, n-i-1, 1):
5.             if arr1[j+1]<arr1[j]:
6.                 arr1[j+1], arr1[j]= arr1[j], arr1[j+1]
7. L=[5, 4, 6, 1, 3, 2, 9, 7]
8. bubble_sort(L)
9. print(L)

```

The simplicity of this algorithm is its strength, but its complexity is its downside. Can we reduce the number of swaps in the preceding algorithm? Let us see!

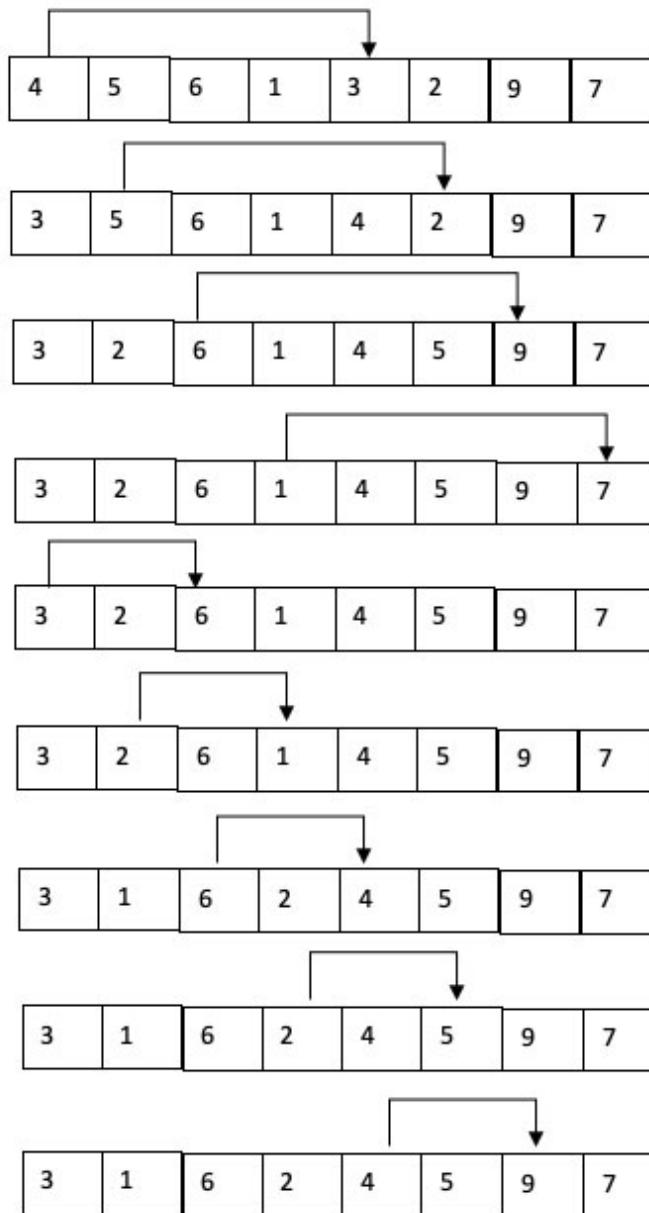
### Comb sort

How can you make the previous process more efficient? Instead of comparing the consecutive elements, compare the first element with the  $n/2^{th}$ , the second with  $(n/2 + 1)^{th}$ , and so on.

In the next iteration, reduce this distance to  $n/4$ . That is, compare the first element with the  $n/4$ th element, the second with  $(n/4 + 1)$ th element, and so on. In the last iteration, you will end up comparing the consecutive elements. Note that, in the end, you will have a bubble sort like situation, but the number of swaps would be greatly reduced.

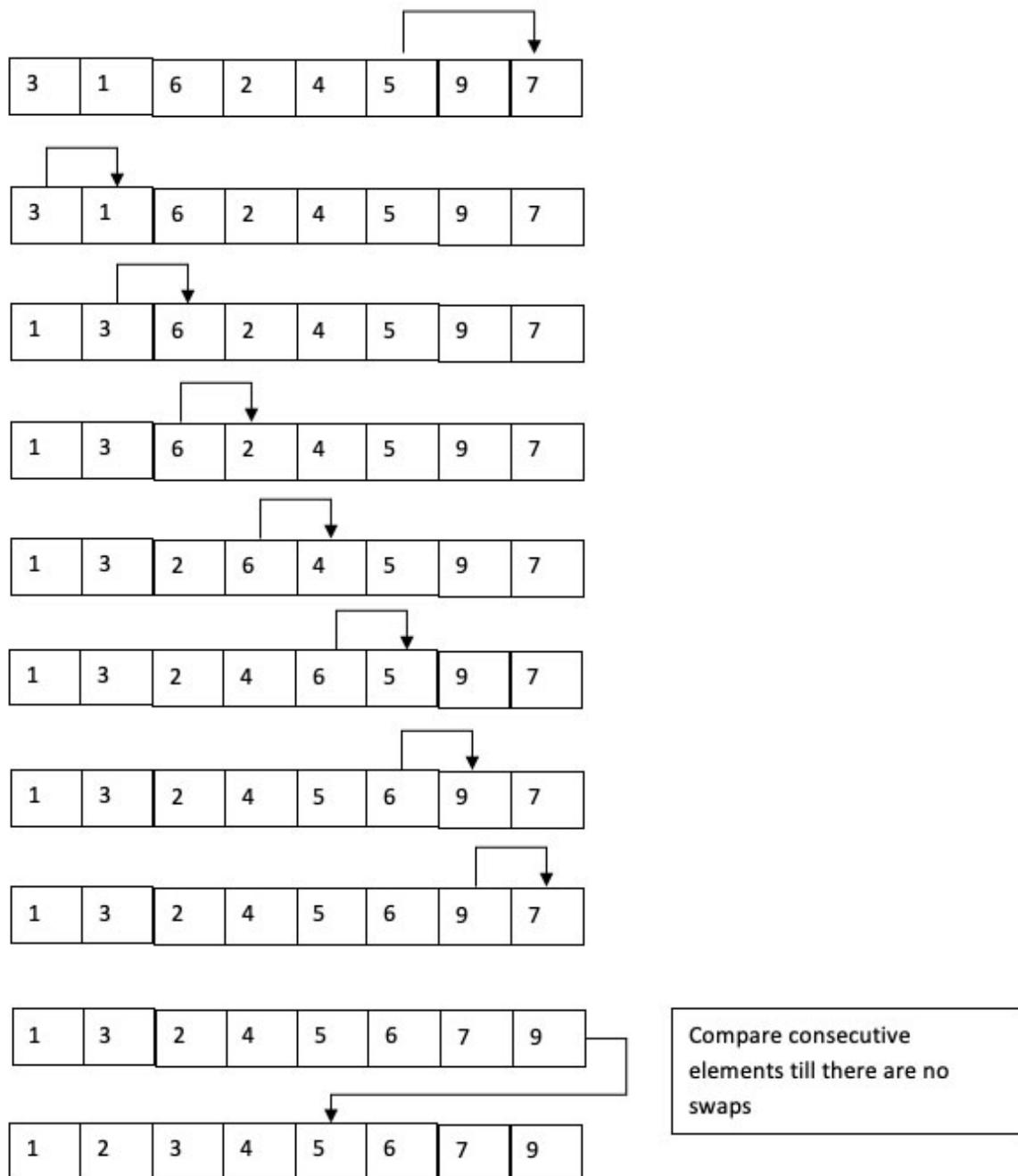
This will markedly reduce the number of comparisons, and the complexity may reduce to  $O(n)$ . However, in the worst case, the complexity would still be  $O(n^2)$ . *Figures 12.2(a) and 12.2(b)* show an example of Comb Sort:

Iteration 1



Iteration 2

Figure 12.2 (a): Comb Sort, Iterations 1 and 2



**Figure 12.2 (b):** Comb Sort, Iterations 3 and 4

Note that the last step of the preceding example is done using Bubble Sort. The reader is expected to write the code of Comb Sort. In case you are stuck, refer to the web resources.

So, we have been able to handle the problem of complexity in Bubble Sort. But can we sort the numbers using an algorithm that works more efficiently?

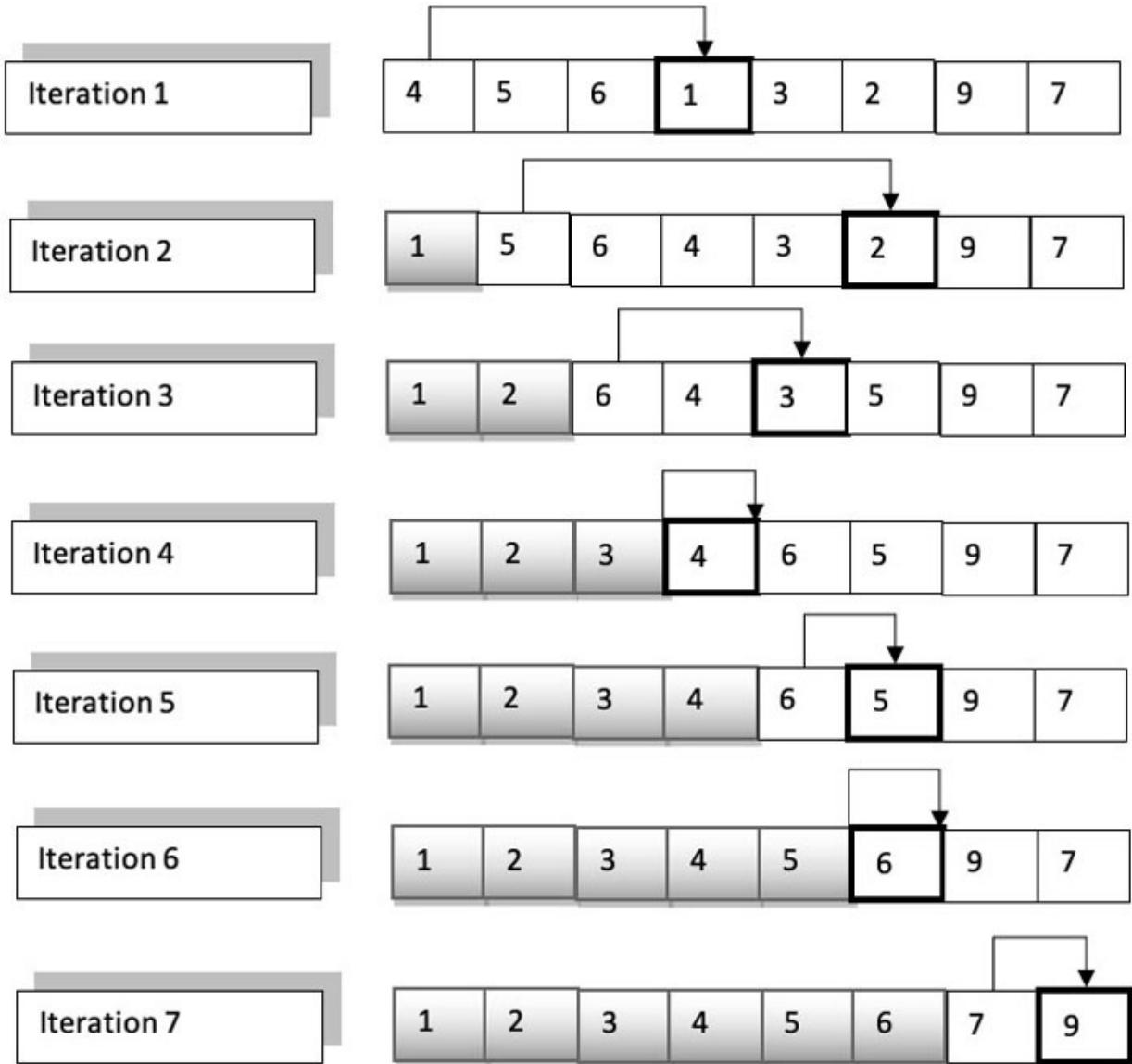
Let us see!

## **Selection sort**

Let us try out another sorting procedure! Here, we compare the first element with the rest of the elements. If in comparing num1 and num2, num2 turns out to be smaller, num1 and num2 are swapped. This way, we will have the smallest element in the first position.

In the next iteration, we compare the second element with the remaining elements (from the third element to the last) and follow the preceding protocol. At the end of the second iteration, we will have the second smallest element at the second position. Likewise, we repeat the process with the rest of the elements. This is called selection sort.

The process is depicted in [figure 12.3](#). Note that in each iteration, the smallest element of the last( $n-i$ ) positions of the given array is placed at the ( $i$ )<sup>th</sup> position.



*Figure 12.3: Selection Sort-I*

[Table 12.2](#) shows the number of comparisons in each iteration and the total number of comparisons in Selection Sort:

Iteration Number	Comparisons
1	(n-1)
2	(n-2)
3	(n-3)
4	...
(n-1)	1
Total	$n \times \frac{n-1}{2} = O(n^2)$

**Table 12.2:** Number of comparisons in Selection Sort-I

Note that the number of swaps depends on the elements of the array itself. In the worst case, each comparison may require a swap, in which case the complexity is  $O(n^2)$ . In the best, none of them require a swap, in which case the complexity is  $O(n)$ . The following code implements selection sort:

```

1. def sel_sort(arr1):
2.     n=len(arr1)
3.     for i in range(n):
4.         for j in range(i+1, n, 1):
5.             if arr1[j]<arr1[i]:
6.                 arr1[i], arr1[j]= arr1[j], arr1[i]

```

Time of execution of the algorithm for 10,000 numbers between 1 and 1,000:

```

1. arr1=np.random.randint(1,1000,10000)
2. L=list(arr1)
3. tic=time.time()
4. sel_sort(L)
5. toc=time.time()
6. print(toc-tic)

```

## Output:

**8.28629755973816**

Let us use the power of Python in optimizing the preceding procedure. Note that the following program places the smallest element from the  $(i+1)^{\text{th}}$  to the  $(n-1)^{\text{th}}$  position of the given array at the  $(i)^{\text{th}}$  position. That is, find the smallest element from the remaining array and place it at the  $(i)^{\text{th}}$  position.

**Note: The following program uses slicing and the min function of the list. The time of execution of the two versions is stated after the program; note the remarkable change in the execution time! Wonderful, is it not?**

The revised program of selection sort is as follows:

```
1. def sel_sort1(arr1):
2.     n=arr1.shape[0]
3.     for i in range(n-1):
4.         m1=np.argmin(arr1[i+1:])
5.         if(arr1[m1+i+1]<arr1[i]):
6.             temp=arr1[i]
7.             arr1[i]=arr1[m1+i+1]
8.             arr1[m1+i+1]=temp
```

Time of execution of the algorithm for 10,000 numbers between 1 and 1,000:

```
1. arr1=np.random.randint(1,1000,10000)
2. tic=time.time()
3. sel_sort1(arr1)
4. toc=time.time()
5. print(toc-tic)
```

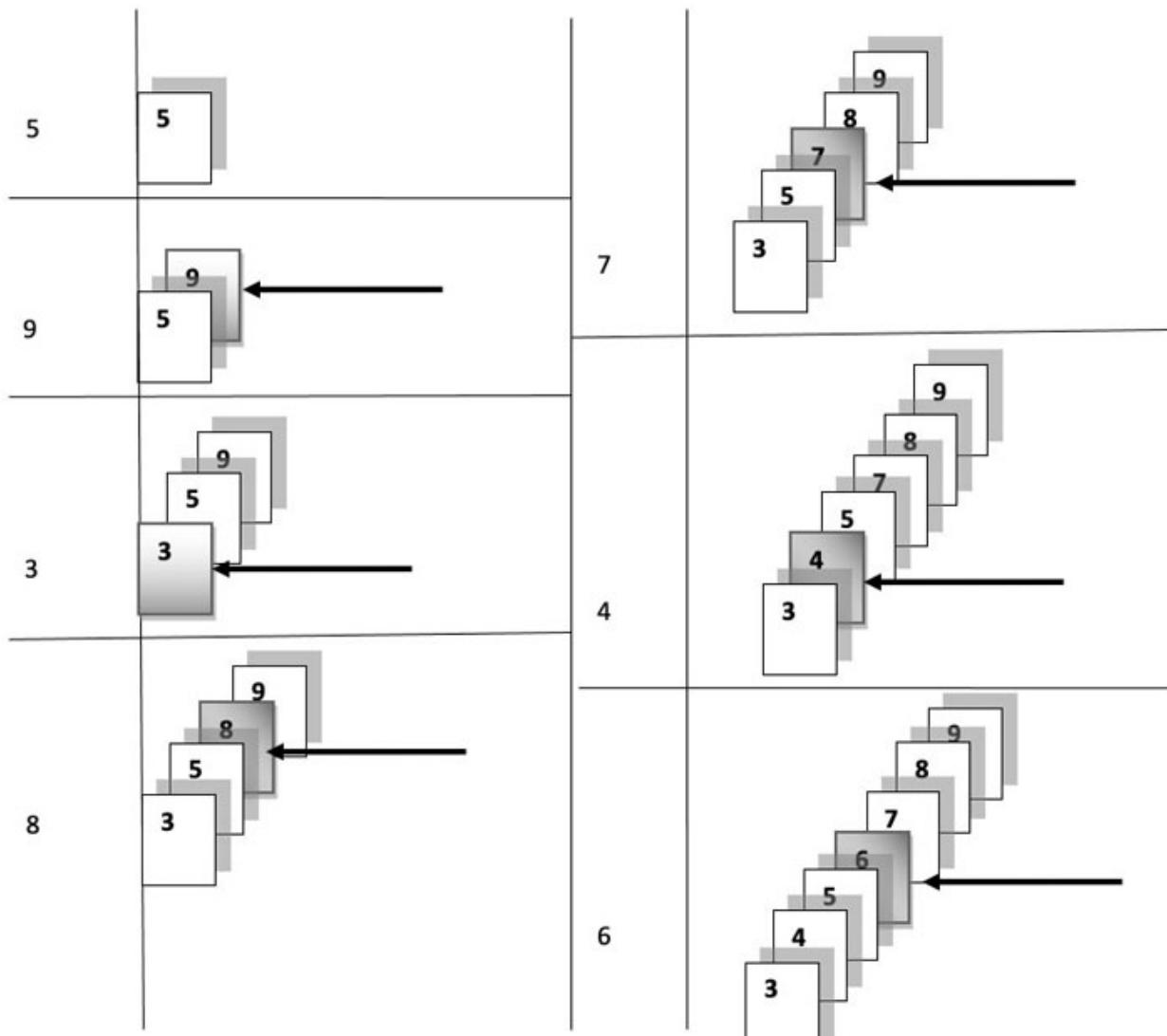
### Output:

**0.31936120986938477**

Having seen the three most important algorithms having quadratic time complexity, let us move to the algorithm which is the most efficient among the quadratic time complexity algorithms.

## Insertion sort

Have you ever played cards? Even if you have not, how would you arrange the given random cards in increasing order? Take the first card, and when you get the second card, place it before the first card; if the second card is smaller; else place it after the first card. As and when you get the next card, place it in its appropriate position. This is insertion sort. [Figure 12.4](#) shows an example of insertion sort:



*Figure 12.4: Insertion Sort*

Note that the worst-case complexity of this algorithm is  $O(n^2)$ . Every time, we get a new card, we begin with the last card in the subset of cards already sorted, compare it with the elements till we get an element greater than the

new one, and place the new card before it. However, the number of swaps is less than the preceding three algorithms.

The reader is expected to write the code himself/herself. However, he/she may refer to the Appendix (Sorting Revisited) for a detailed discussion.

Now, let us move to some more efficient algorithms. The next two sections present Radix Sort and Count Sort, which have a time complexity of  $O(n)$ . Linear time complexity sounds wonderful. However, there is a catch! Let us see.

## Radix sort

In Radix sort, we count the number of digits in the maximum element of the given input (say  $d$ ). The algorithm is  $O(n \times d)$ , where  $n$  is the number of elements in the list. Here, the value of  $d$  is finite and small; hence, the complexity becomes  $O(n)$ , which makes it a linear time complexity algorithm.

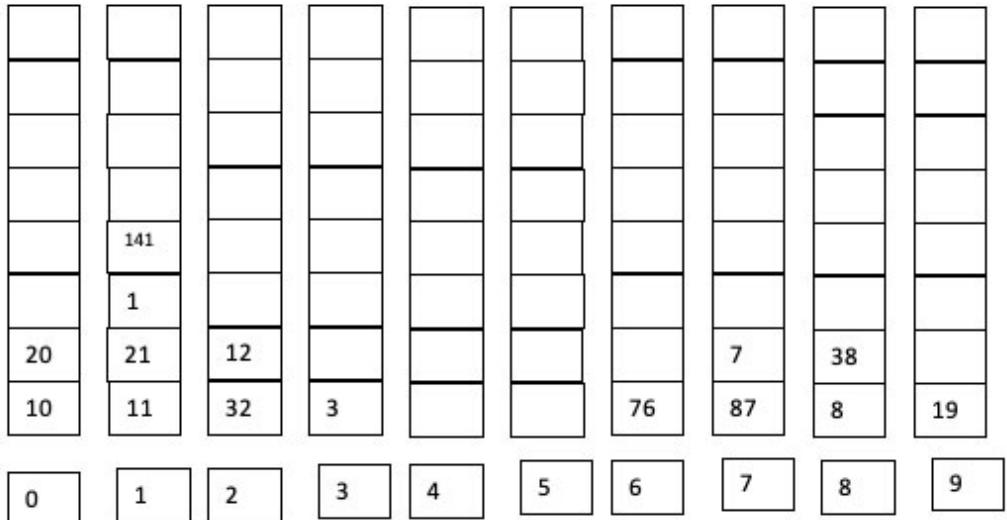
The algorithm works as follows:

- In the first iteration, the elements of the input array are placed in one of the 10 buckets, each representing the digit at the units' place.
- The numbers are then read in order; that is, numbers of bucket 0, followed by that of bucket 1, and so on. In the next iteration, the read numbers are placed in one of the 10 buckets, each representing the digit at the tens place.
- The process continues for  $d$  times.

*Figures 12.5(a) and 12.5(b) exemplify the process:*

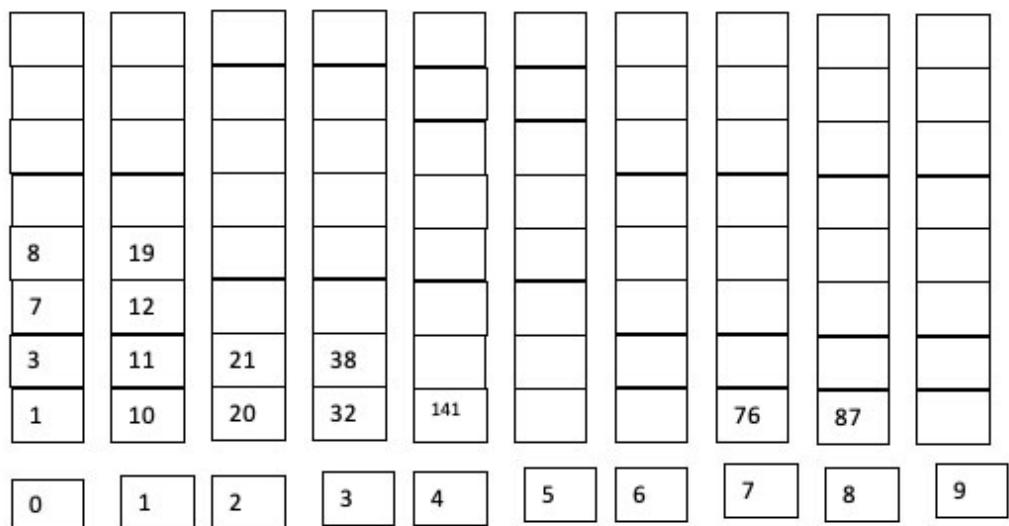
Iteration 1

3, 8, 10, 32, 20, 11, 87, 76, 38, 19, 21, 12, 1, 7, 141



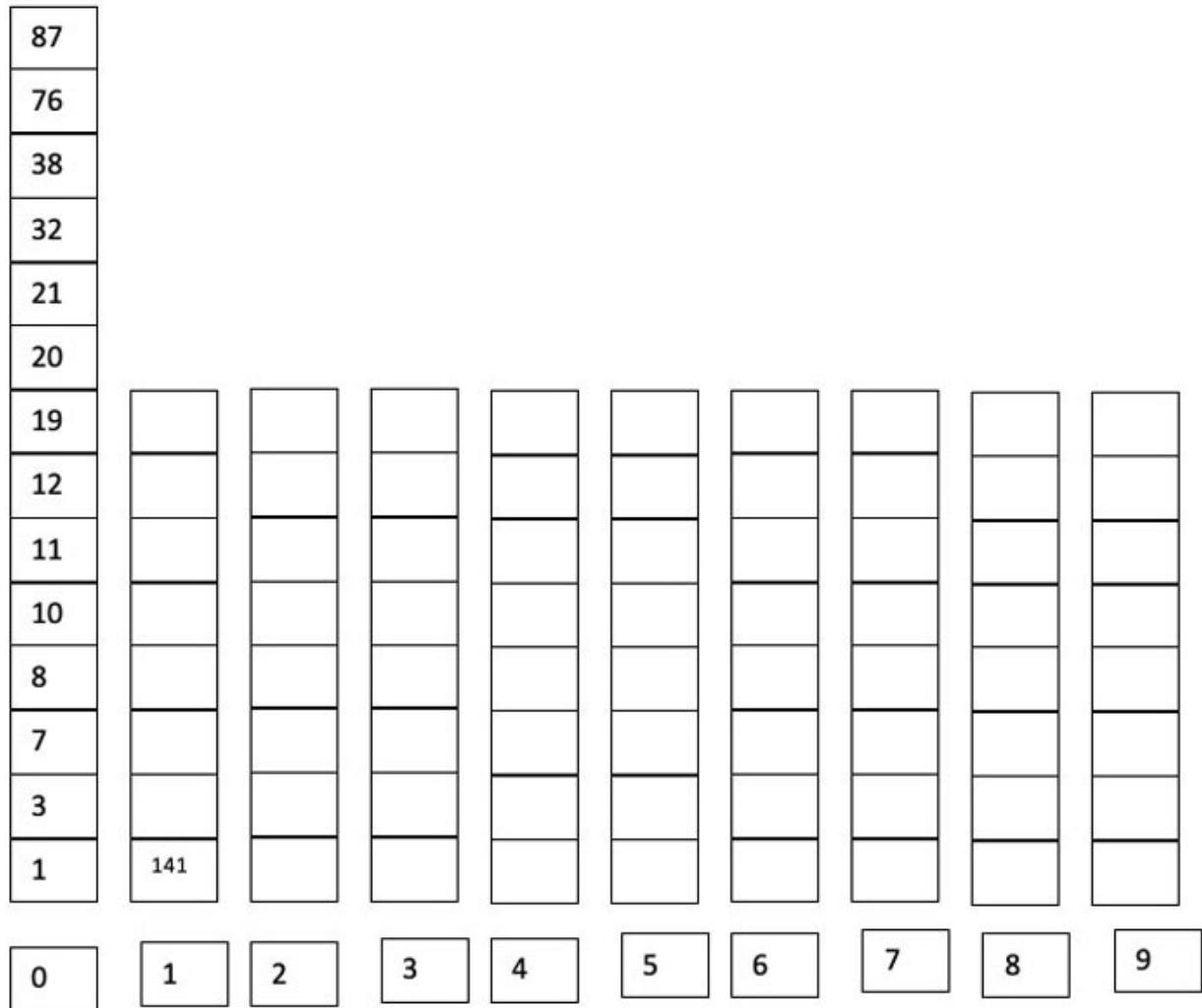
Iteration 2

10, 20, 11, 21, 1, 141, 32, 12, 3, 76, 87, 7, 8, 38, 19

**Figure 12.5 (a): Radix Sort, Iterations 1 and 2**

Iteration 1

1, 3, 7, 8, 10, 11, 12, 19, 20, 21, 32, 38, 141, 76, 87



**Figure 12.5 (b): Radix Sort, Iterations 3 and 4**

The following code implements Radix sort. Have a look at the code, and appreciate the elegance of Python:

### Code:

1. `import math`
2. `input1=[2,14,3,65,8,78,900,32,11]`
3. `L= [[0 for i in range(1)] for j in range(10)]`

```

4. n=max(input1)
5. d1=math.floor(math.log10(n)+1)
6. for d in range(d1):
7.     for i in range(len(input1)):
8.         p=(input1[i]/(10**d))%10
9.         L\=L[p]
10.        L\.append(input1[i])
11.    input1 = [sub[j] for sub in L for j in range(1, len(sub))]
12.    L=[[0 for i in range(1)] for j in range(10)]
13. print(input1)

```

Note that this algorithm cannot be used in each and every situation. For example, this algorithm should be used to sort a list of 10 numbers having the largest number containing hundred digits. Let us now have a look at another linear time complexity called counting sort.

## Counting sort

Counting sort counts the occurrence of an element in the input array, places the count in an auxiliary array, and proceeds in a simple yet efficient manner to produce a sorted array. The following are the steps involved in counting sort:

**Step 1:** Find the maximum number in the given input, and create an array having a length equal to one more than that number.

**Step 2:** Find the count of each element in the input and store it at the same index in the count array.

**Step 3:** In the count array, starting from the first index, find the cumulative sum. This will help us to find out the position of each element in the resultant array.

**Step 4:** For each element of the count array, place the index at the position specified arr[index] in the resultant array.

The process is shown in [figure 12.6](#):

Input array	<table border="1"><tr><td>3</td><td>2</td><td>2</td><td>1</td><td>7</td><td>4</td><td>7</td><td>1</td><td>6</td><td>5</td></tr></table>	3	2	2	1	7	4	7	1	6	5										
3	2	2	1	7	4	7	1	6	5												
Count	<table border="1"><tr><td>0</td><td>2</td><td>2</td><td>1</td><td>1</td><td>1</td><td>1</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	0	2	2	1	1	1	1	2	0	1	2	3	4	5	6	7				
0	2	2	1	1	1	1	2														
0	1	2	3	4	5	6	7														
Input array	<table border="1"><tr><td>0</td><td>2</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>10</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	0	2	4	5	6	7	8	10	0	1	2	3	4	5	6	7				
0	2	4	5	6	7	8	10														
0	1	2	3	4	5	6	7														
Cumulative Sum	<table border="1"><tr><td>0</td><td>1</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td></tr></table>	0	1	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8		
0	1	3	4	5	6	7	8	9													
0	1	2	3	4	5	6	7	8													
Cumulative Sum	<table border="1"><tr><td>0</td><td>0</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>9</td><td>7</td><td>8</td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table>	0	0	2	3	4	5	6	9	7	8	0	1	2	3	4	5	6	7	8	9
0	0	2	3	4	5	6	9	7	8												
0	1	2	3	4	5	6	7	8	9												
	Resultant Array																				
	<table border="1"><tr><td></td><td>1</td><td></td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td></td><td>7</td></tr></table>		1		2	3	4	5	6		7										
	1		2	3	4	5	6		7												
	<table border="1"><tr><td>1</td><td>1</td><td>2</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>7</td></tr></table>	1	1	2	2	3	4	5	6	7	7										
1	1	2	2	3	4	5	6	7	7												

*Figure 12.6: Count Sort*

The following program implements counting sort. Note that the complexity is  $O(n)$ , owing to the absence of any nested loops.

### Code:

```

1. input1=[2, 14, 3, 5, 8, 2, 6, 3, 2]
2. max1=max(input1)
3. count=[0]*max1
4. for i in input1:
5.     count[i-1]+=1
6. for i in range(1, len(count)):
7.     count[i]+=count[i-1]
8. final=[0]*len(input1)
9. for i in range(len(input1)-1, -1, -1):
10.    final[count[input1[i] - 1]-1]=input1[i]
11.    count[input1[i]-1] -= 1
12. print(final)

```

### Output:

[2, 2, 2, 3, 3, 5, 6, 8, 14]

The algorithm is simple and has complexity  $O(n)$ . But what if the maximum number in the input array is 567890, and the number of elements is 10. In such situations using this algorithm would waste a lot of memory. So, it has some limitations. The following algorithms, though  $O(n \log n)$ , have no such constraints.

## Merge and merge sort

Given two sorted arrays, the Merge algorithm generates a sorted array consisting of elements of the two input arrays, with each element appearing only once. The algorithm works as follows:

Initially, two pointers  $i$  and  $j$ , point to the respective first elements of the two arrays (arr1 and arr2). The resultant array is arr3. The initial value of  $k$  is 0. If arr1[i] is less than arr2[j], then arr1[i] is stored in arr3[k], and the values of  $i$  and  $k$  are incremented by 1. If arr2[j] is less than arr1[i], then arr2[j] is stored in arr3[k], and the values of  $j$  and  $k$  are incremented by 1. In case both arr1[i] and arr2[j] are the same, one of them is stored in arr3[k], and the values of  $i$ ,  $j$ , and  $k$  are incremented by 1.

The preceding iterations continue till  $((i < n_1) \text{ and } (j < n_2))$ , where  $n_1$  is the number of elements in the first array and  $n_2$  is the number of elements in the second array. When this loop ends, the remaining elements of either array are copied to the resultant array. The following code implements the Merge algorithm:

### **Code:**

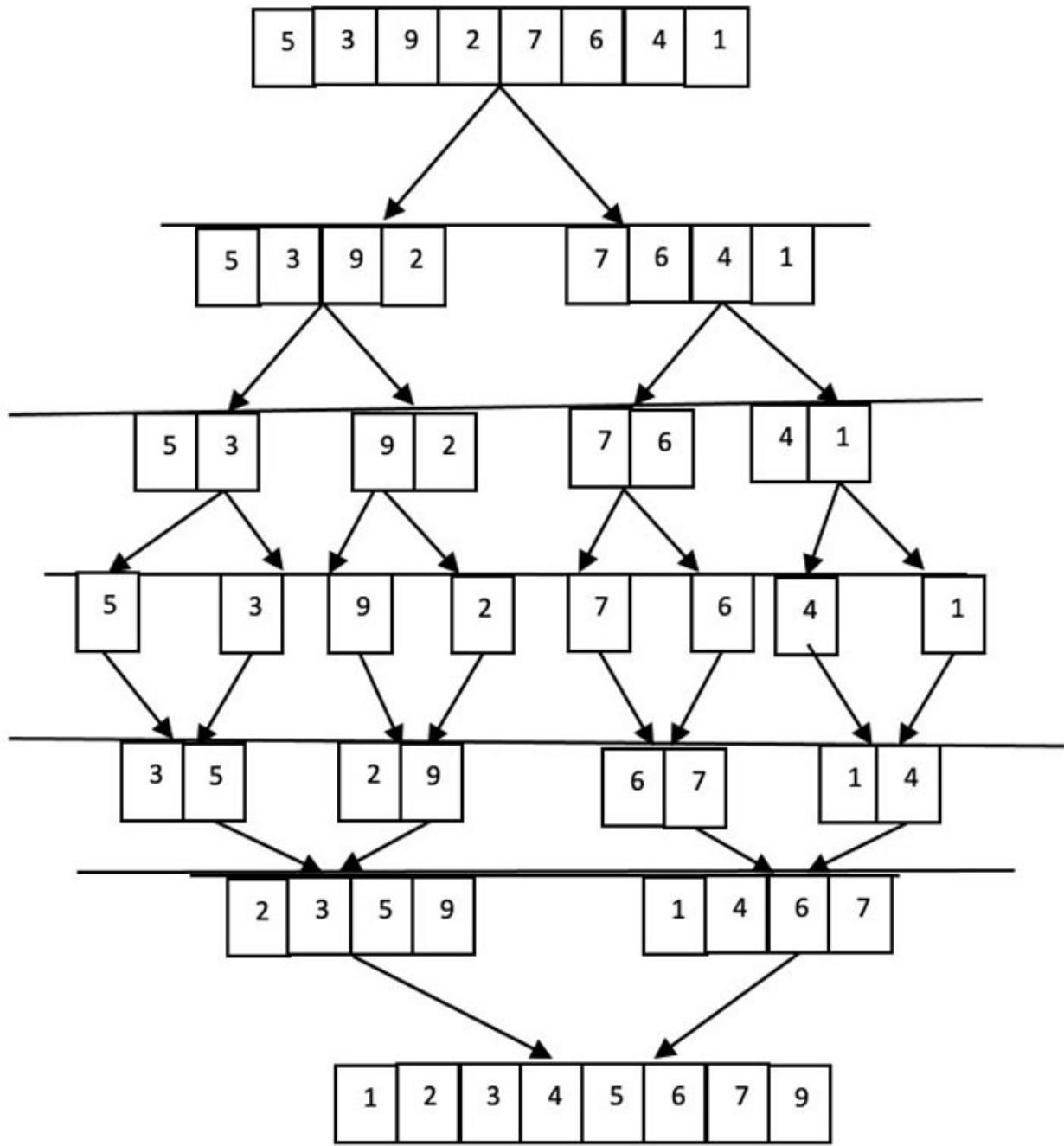
```
1. def merge(L1, L2):  
2.     L=[]  
3.     n1=len(L1)  
4.     n2=len(L2)  
5.     i=0  
6.     j=0  
7.     k=0  
8.     while((i<n1)and (j<n2)):
```

```

9.     if(L1[i]<L2[j]):
10.        L.append(L1[i])
11.        i+=1
12.        k+=1
13.    elif(L2[j]<L1[i]):
14.        L.append(L2[j])
15.        j+=1
16.        k+=1
17.    else:
18.        L.append(L1[i])
19.        i+=1
20.        j+=1
21.        k+=1
22.    while(i<n1):
23.        L.append(L1[i])
24.        i+=1
25.        k+=1
26.    while(j<n2):
27.        L.append(L2[j])
28.        j+=1
29.        k+=1
30.    return L

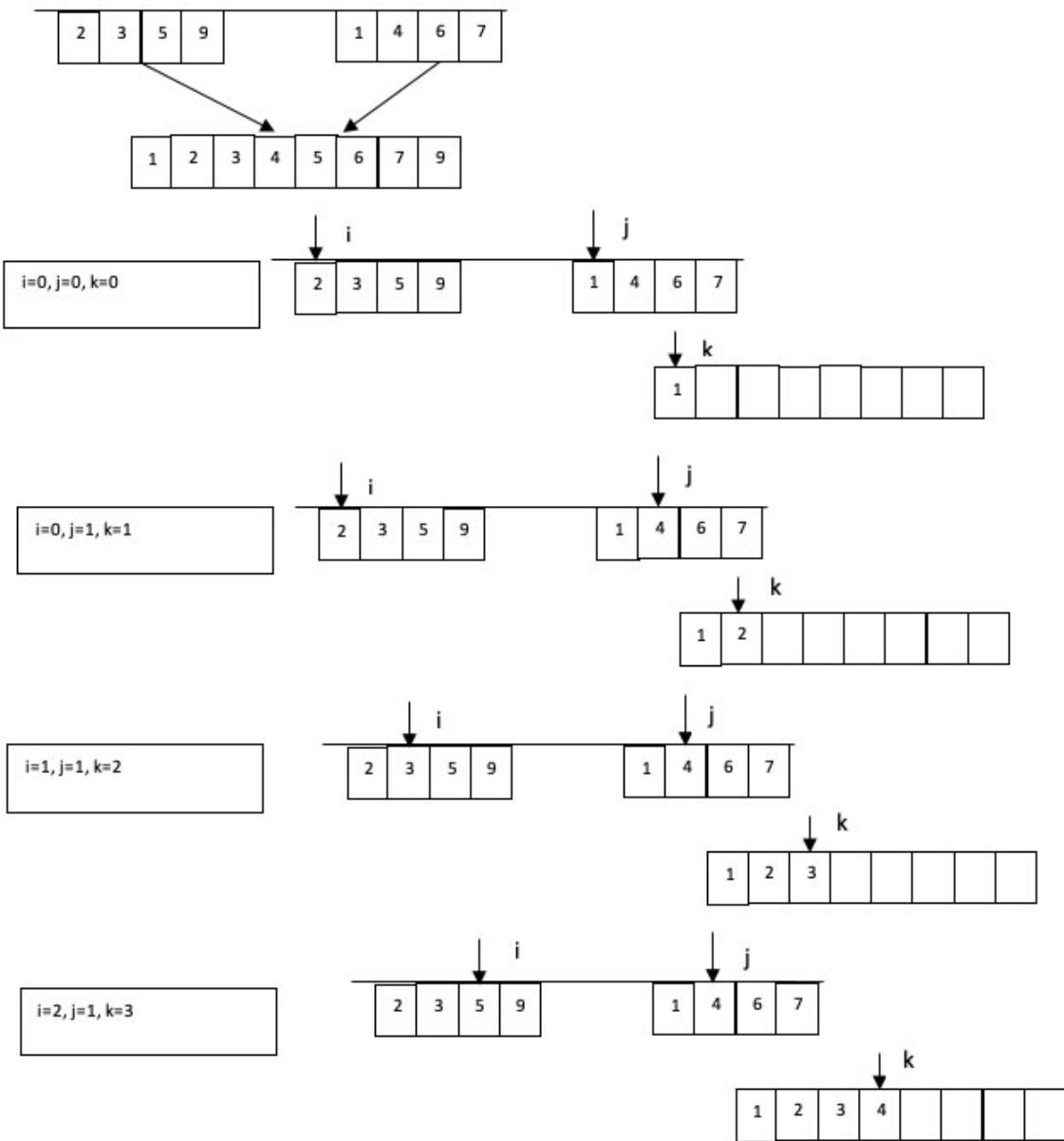
```

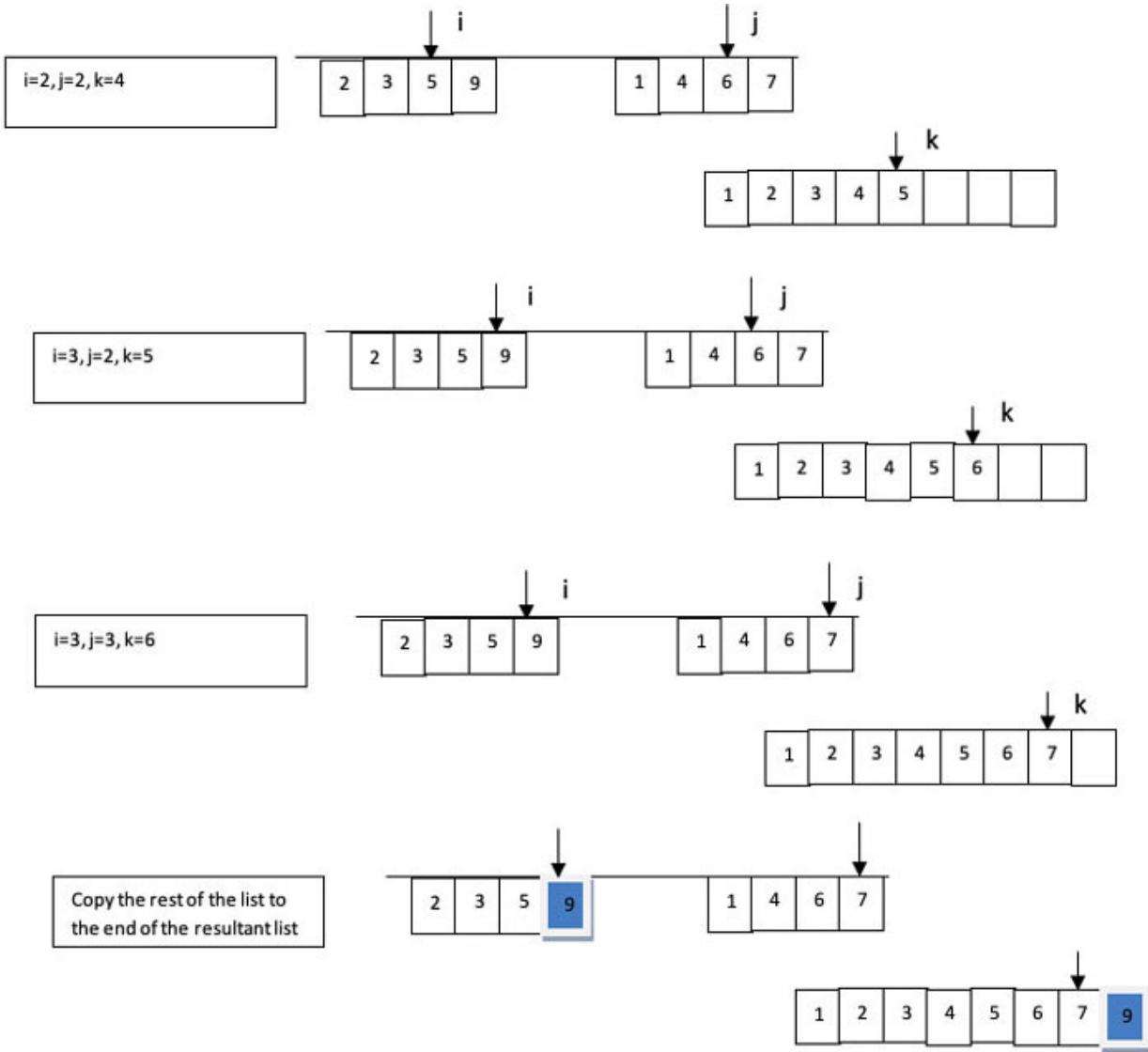
The merge sort algorithm can be perceived as a combination of two parts. The first part divides the given array into two parts until a single element is left. Note that an array containing a single element is deemed to be sorted. The second part takes two smaller sorted arrays in each step and merges them. To understand the algorithm, consider the example shown in [figure 12.7](#):



*Figure 12.7: An example of the Merge Sort*

[Figure 12.8](#) shows the working of the Merge procedure in the following example:





*Figure 12.8: Example of Merge*

The code of Merge Sort follows:

### Code:

```

1. def mergesort(L):
2.     if((len(L)==1) or (len(L)==0)):
3.         return L
4.     else:
5.         low=0

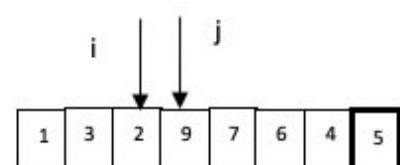
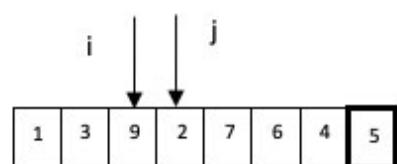
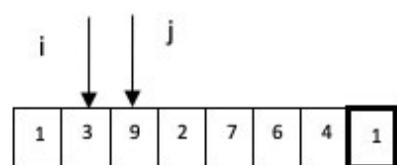
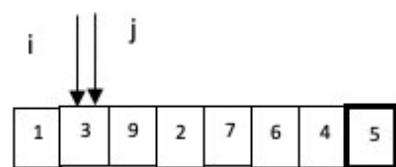
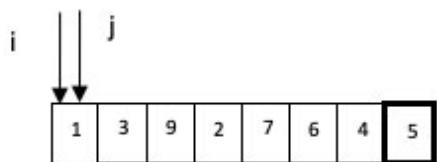
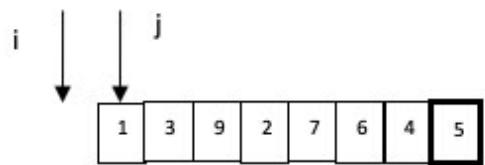
```

```
6.     high=len(L)-1
7.     mid=(low+high)//2
8.     L1=mergesort(L[:mid+1])
9.     L2=mergesort(L[mid+1:])
10.    L_=merge(L1, L2)
11.    return L_
```

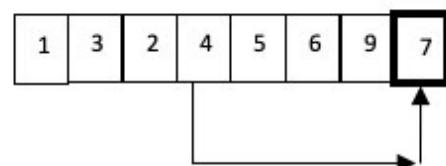
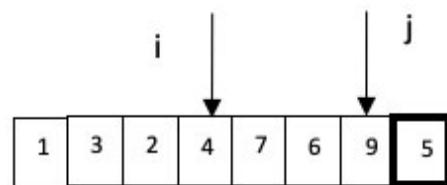
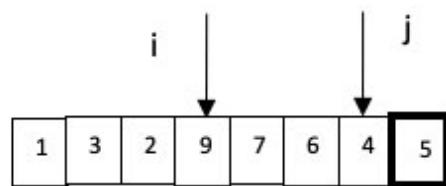
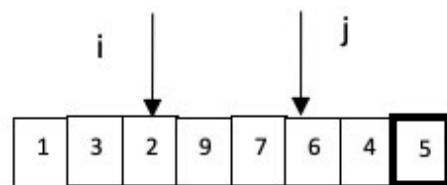
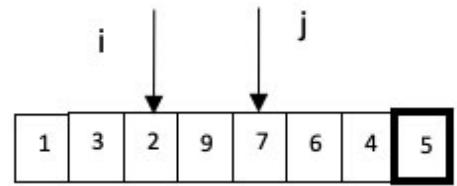
Merge sort is efficient compared to Bubble, Selection, or Insertion Sort in terms of the time of execution, as its time complexity is  $O(n \log n)$ . However, it requires auxiliary memory. Actually, a lot of it. The reader is expected to work out the memory complexity of this algorithm, and once you are ready, check your answer in the last section of this chapter. Having seen the algorithm for merge and merge sort, let us now move to partition and quick sort.

## Partition and quick sort

The partition algorithm finds the correct position of the pivot element. In the discussion that follows, the last element of the input array is taken as the pivot, and the resultant array will have numbers smaller than the pivot on the left of the pivot and all the numbers greater than the pivot on the right of it. The algorithm that follows uses two variables  $i$  and  $j$ . Initially,  $i$  is set to  $-1$  and  $j$  to  $0$ . If  $\text{arr}[j] < \text{pivot}$ , then  $i$  is incremented by  $1$ , and  $\text{arr}[i]$  and  $\text{arr}[j]$  are swapped. In each iteration,  $j$  is incremented by  $1$ . In the end,  $\text{arr}[i+1]$  is swapped with pivot. *Figures 11.9(a) and 11.9(b)* exemplify partition:



**Figure 12.9 (a): Example of partition**



*Figure 12.9 (b): Example of partition continued*

The following program implements the partition algorithm. Note that the complexity of this algorithm is  $O(n)$ . Furthermore, note that this is one of the versions of the partition algorithm; there is another version of partition presented in the Appendix of this book.

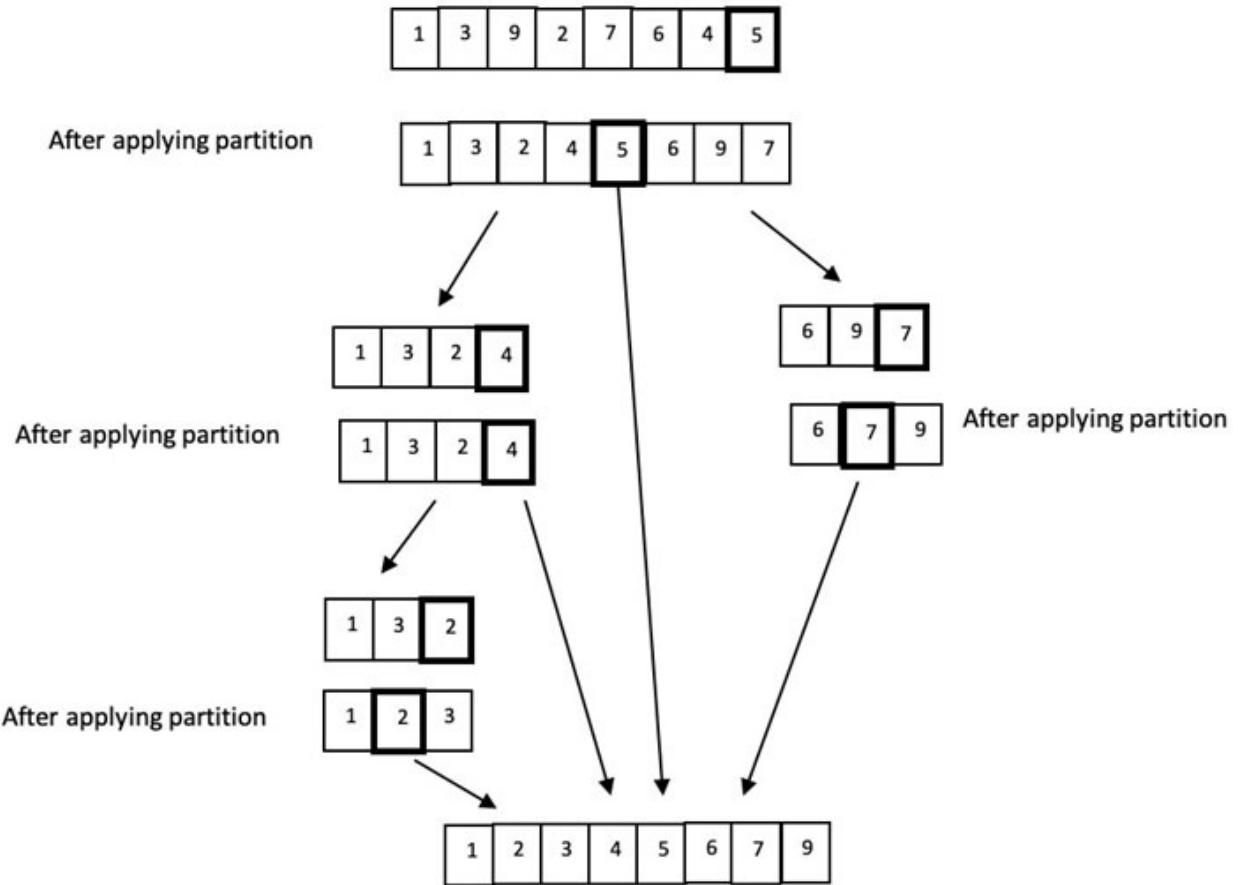
**Code:**

```

1. def partition(L):
2.     if(L==[]):
3.         return 0
4.     low=0
5.     high=len(L)-1
6.     pivot = L[high];
7.     i = (low - 1)    #indicates the right position of pivot found
so far
8.     for j in range(low, high):
9.         if (L[j] < pivot):
10.             i+=1
11.             L[i], L[j]= L[j], L[i]
12.             L[i + 1], L[high]=L[high], L[i+1]
13.     return (i + 1)

```

The quick sort uses the partition algorithm to sort the array. Initially, the complete array is passed into quick sort, which, in turn, calls partition. The partition algorithm places the last element at its correct position, say at pos. The input array's left part (from index 0 to pos), and the right part (from index pos+1 to (n-1)) are then passed as an argument to the quick sort. The process is shown in [figure 12.10](#):



*Figure 12.10: Quick Sort*

The following program implements quick sort:

### Code:

```

1. def quick_sort(L):
2.     low=0
3.     high=len(L)-1
4.     if(low<high):
5.         #print(low, high)
6.         pos=partition(L)
7.         #print(low, pos, high, L)
8.         quick_sort(L[low:pos])
9.         quick_sort(L[pos+1: high+1])

```

In the previous chapters, we saw that the quick sort has an average-case time complexity of  $O(n \log n)$  and a worst-case time complexity of  $O(n^2)$ . It is memory efficient compared to Merge Sort.

## Conclusion

This chapter introduced some sorting algorithms. The procedures are explained, exemplified, and implemented in the chapter. Let us have a quick look at the complexities of various algorithms and summarize the chapter ([table 12.3](#)):

Algorithm	Worst Case Complexity	Best Case Complexity	Space Complexity
Bubble Sort	$O(n^2)$	$O(n)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nd)$	$O(nd)$	$O(nd)$ , d being digits in maximum number
Counting Sort	$O(n+k)$	$O(n+k)$	$O(k)$ , k being range
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(1)$

*Table 12.3: Comparison of sorting algorithms*

The following points are worth noting in sorting:

1. Bubble Sort, Selection Sort, Insertion Sort, and Quick Sort are in-place, whereas Merge Sort requires auxiliary arrays and is, therefore, not in-place.
2. Bubble Sort, Insertion Sort, and Merge Sort are stable, whereas Selection Sort and Quick Sort are not stable.
3. If an array is almost sorted, then insertion sort is preferred.

The upcoming chapter focuses on Searching and Selection. The Appendix also explores some issues related to Sorting. Heap Sort has not been covered in the chapter, as this book contains a dedicated chapter on the heap. Also,

the average case complexity of some of the algorithms has been dealt with in the Appendix (sorting revisited). The Appendix also contains two more sorting algorithms, namely, Shell sort and Bucket sort. It also has a programming exercise based on sorting.

## Illustrations

### **Question 1. Sorting by frequency**

**Problem:** Given a set of numbers, sort them in order of the frequency in which they appear in the input.

**Example:**

Input : [1, 4, 6, 3, 2, 4, 1, 4, 2, 2, 2, 3, 7]

Output: [2, 2, 2, 2, 4, 4, 4, 1, 1, 3, 3, 6, 7]

**Solution:**

The problem can be handled using the following approach:

1. Begin by sorting the input array using any sorting algorithm.
2. Scan the sorted array and construct a 2D array  $\text{arr}(n,2)$  such that  $n$  is the number of elements, and the second column represents the frequency of occurrence of each element.
3. Sort the array  $\text{arr}(n,2)$  with respect to the frequency  $f$  that is using the second column.
4. Unwind the  $\text{arr}(n,2)$  to get the desired sequence.

**Note: If the frequency of two elements is the same in Step 3, the element with the lower index value should be processed first.**

### **Question 2. Repeated Numbers in a given range**

**Problem:** A list contains only 0's, 1's, and 2's. How will you sort it out in the best way?

**Solution:**

The preceding problem can be handled using the Counting Sort, explained in the chapter.

**Question 3. Problem:** In an election of  $k$  candidates,  $n$  people voted, and their votes were stored in  $\text{votes}[0,1,2..n-1]$ . How will you sort the array of votes?

### Solution:

The goal of this problem is to find the element that has the maximum occurrences in the votes array.

Some of the interesting approaches to handle the problem are as follows:

#### Method 1: Nested Loop

The easiest approach would be to craft a nested loop where the outer loop iterates through  $k$  elements of the candidate array, and in each iteration, a comparison is made with the  $n$  elements of the votes array to ultimately find the maximum occurrences of the element.

```
max=0
for i in range(k):
    max1=0
    for j in range(n):
        if(arr[j]==i):
            max1+=1
        if(max1>max):
            max=max1
```

But the preceding approach is costlier than it looks. Because of the introduction of nested loops, the complexity of the code is  $O(n^2)$ . In fact, there is a better way to handle the problem.

#### Method 2: Sort and linear traversal

An alternate approach would be to sort the given array of votes using a sorting algorithm and linearly traverse to calculate the maximum frequency of each element and find the max frequency out of the same. This brings down the time complexity to  $O(n\log n)$ .

#### Method 3: Counting Sort

The best approach would be to use the counting sort, owing to the reasons explained earlier.

### Question 4: Application of Radix Sort

**Problem:** Given an array of numbers  $\text{arr}[1, n^4]$ . What is the best way to sort this array?

**Solution:**

Radix Sort takes  $O(l*(n+k))$  time to sort an array of  $n$  elements,  $l$  is the number of digits in each element, and  $k$  is the base, i.e., the number of values each digit can have.

The most efficient way to handle this problem is:

1. Decrement 1 from each element of the given array such that the new array becomes  $\text{arr\_new}[0, n^4 - 1]$
2. Now the maximum number in the given range is in the form  $n^R - 1$ , which has  $\log_b(n^R - 1)$  digits.
3. Based on the preceding conclusion, convert every element of the given array to base 4.
4. Performing Radix sort on the newly converted array. This would lead to the maximum time complexity of  $O(4n+k)$ . In other words, we are calling counting sort (complexity =  $n + k$ ) one time for each of the  $l$  digits in the input element.
5. Add one to the sorted array obtained in the previous step to obtain the original array.

**Question 5.** Which sorting algorithms are stable?

**Answer:** The stability of a sorting algorithm is concerned with how the algorithm treats equal (repeated) elements. Stable sorting algorithms preserve the relative order of equal elements, whereas unstable sorting algorithms do not. In other words, stable sorting maintains the position of two equal elements relative to one another.

Examples of stable sorting algorithms include, Merge Sort, Counting Sort, Insertion Sort, and Bubble Sort. On the other hand, Quicksort, Heapsort, and Selection Sort are unstable.

**Question 6.** When does the stability of an algorithm matter?

**Answer:** When the collection already has some order, then sorting on another key must preserve that order.

**Question 7.** Sorting for Linked List

**Problem:** Which of Merge Sort or Quick sort is more suitable for the linked list and why?

**Solution:**

Quick sort requires a lot of random access of elements, therefore, escalating the overhead that comes with a linked list. Merge sort, however, relies on sequential access of data (mostly), and the need for random access is reasonably low; therefore, it can be used with a linked list.

## Multiple choice questions

- 1. The students of a class are asked to vote for one of the three candidates, having IDs {1, 2, 3}. Their response is stored in an array. Which sorting algorithm would you use to sort the array?**
  - a. Counting
  - b. Radix
  - c. Quick
  - d. None of the above
  
- 2. An array is almost sorted; which algorithm would you use to complete sort it?**
  - a. Bubble
  - b. Selection
  - c. Comb
  - d. None of the above
  
- 3. What is the complexity of Bubble Sort?**
  - a.  $O(n^2)$
  - b.  $O(n)$
  - c.  $O(n \log n)$
  - d. None of the above
  
- 4. Which of the following can significantly reduce the number of swaps in Bubble Sort?**

- a. Selection Sort
- b. Comb Sort
- c. Quick Sort
- d. None of the above

**5. What is the best case complexity of Selection Sort?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(n \log n)$
- d. None of the above

**6. You are given a list of numbers in which the maximum number is 9,819. The list has 1,000 elements; which algorithm would you use to sort the input?**

- a. Radix sort
- b. Count Sort
- c. Comb sort
- d. None of the above

**7. What is the average-case complexity of merge?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(n \log n)$
- d. None of the above

**8. What is the average-case complexity of Merge Sort?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(n \log n)$
- d. None of the above

**9. What is the best-case complexity of Quick Sort?**

- a.  $O(n^2)$

- b.  $O(n)$
- c.  $O(n \log n)$
- d. None of the above

**10. What is the worst-case complexity of Quick Sort?**

- a.  $O(n^2)$
- b.  $O(n)$
- c.  $O(n \log n)$
- d. None of the above

**11. Which of the following is stable?**

- a. Quick
- b. Merge
- c. Selection
- d. All of the above

**12. Which of the following is stable?**

- a. Bubble
- b. Comb
- c. Insertion
- d. All of the above

**13. Which of the following is stable?**

- a. Radix
- b. Count
- c. Both
- d. None of the above

**14. Which of the following is in-place?**

- a. Quick
- b. Merge
- c. Selection

d. All of the above

**15. Which of the following is in-place?**

- a. Bubble
- b. Comb
- c. Insertion
- d. All of the above

**16. Which of the following is in-place?**

- a. Radix
- b. Count
- c. Both
- d. None of the above

**17. Which of the following requires auxiliary memory?**

- a. Merge sort
- b. Quick Sort
- c. Both
- d. None of the above

**18. Which of the following can be used when multiple processors are available?**

- a. Merge sort
- b. Quick Sort
- c. Both
- d. None of the above

**19. You are given a sorted list (almost); which of the following should not be used?**

- a. Merge sort
- b. Quick Sort
- c. Both
- d. None of the above

**20. Which of the following should be considered for comparing sorting algorithms?**

- a. Number of swaps
- b. Number of comparisons
- c. Stability
- d. All of the above

## **Theory**

Write the algorithm for the following. Derive the average-case complexity, and state the downsides of the algorithm. Also, state its space complexity.

- 1. Bubble Sort
- 2. Comb Sort
- 3. Selection Sort
- 4. Insertion Sort
- 5. Radix Sort
- 6. Counting Sort
- 7. Merge Sort
- 8. Quick Sort

## **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 13

## Median and Order Statistics

### Introduction

The previous chapters discussed various searching and sorting algorithms. In a sorted sequence, the median is the element at the  $n/2^{\text{th}}$  position (in case  $n$  is even), where  $n$  is the number of elements. The first and the third quartile are the elements at the  $(n/4)^{\text{th}}$  and the  $(3n/4)^{\text{th}}$  position. However, sorting takes time  $O(n \log n)$  (assume we used Heap Sort). There can be efficient methods of finding the median, quartiles, and so on. This chapter discusses algorithms for order statistics, which find application in various domains, including Machine Learning.

### Structure

In this chapter, we will cover the following topics:

- Introduction to median and other statistics
- Median of medians
- Median using heaps
- Median using insertion sort
- Median using partition

### Objectives

After reading this chapter, you will understand the various methods of finding the median. Some of the methods, such as median using heaps and median using insertion sort, can also be used to find the median of a stream of numbers. The median of the median method can be used to find the  $k^{\text{th}}$  smallest or largest element. The order statistics discussed in this chapter are immensely important and form the foundation stone of many algorithms in signal and systems and machine learning.

### Introduction to median and order statistics

So far, we have discussed many approaches and design paradigms. Let us revisit the simplest problem: finding the minimum element from a given list. The time complexity of the brute force approach for finding the minimum element from a given list is  $O(n)$ . Likewise, finding the maximum element will take  $O(n)$  time. One can also find the second minimum or the second maximum element in  $O(n)$  time. Finding the median is slightly difficult. If the given list has odd number of elements, one needs to sort the numbers and then find the middle element of the list. In case the list has even number of elements, then the average of the  $(n/2)^{th}$  element and  $(n/2 + 1)^{th}$  element is the median. For example, consider the following list having fifteen elements.

5, 91, 71, 2, 34, 10, 11, 15, 8, 67, 90, 21, 89, 78, 40

To find the median of the list, we need to sort the list and then return the middle element, which in this case is 34.

2, 5, 8, 10, 11, 15, 21, 34, 40, 67, 71, 78, 89, 90, 91

The code follows:

### Code:

```
1. arr4=np.sort(arr)
2. if (arr4.shape[0]%2 ==0):
3.     median = (arr4[arr4.shape[0]//2 -1] + arr4[arr4.shape[0]//2
        ])/2
4. else:
5.     median = arr4[arr4.shape[0]//2 ]
6. print(median)
```

### Output:

6.5

Sorting takes a minimum of  $O(n \log n)$  time, and locating the  $n/2^{th}$  element will take  $O(1)$  time. The complexity of this method is, therefore,  $O(n \log n)$ . The complexity of finding the median, quartiles, deciles, and other statistics can be reduced to  $O(n)$  using the methods discussed in this chapter. The chapter focuses on finding the median of a list and the  $k^{th}$  largest (or, for that matter, smallest) element and that of a stream, but similar techniques can be used to find other such statistics also.

## Median of medians

Sorting numbers takes  $O(n \log n)$  time using techniques like heap sort. However, sorting a few numbers (Say 5) can be done in constant time. Let us use this intuition to reduce the time complexity of this problem. Consider the following set of numbers:

5, 91, 71, 2, 34, 10, 11, 15, 8, 67, 90, 21, 89, 78, 40

Let us divide them into groups of fives (in case the number of elements in the list is not divisible by 5, the last column can have remaining elements in the middle of the last column and zeros elsewhere).

5	10	90
91	11	21
71	15	89
2	8	78
34	67	40

We then sort each column (in the constant time since each column has only five elements). The third row of the resultant table is then sorted, and the middle element is returned as the median. The third row has 34, 11, and 78. The median of the given set of numbers is, therefore, 34.

2	8	21
5	10	40
34	11	78
71	15	89
91	67	90

Note that sorting each small group takes  $O(1)$  time. Hence, sorting of  $(n/5)$  columns will take  $O(n)$  time.

2	8	21
5	10	40
34	11	78
71	15	89
91	67	90
2	8	21
5	10	40
11	34	78

71	15	89
91	67	90

The preceding method finds the median of the medians. The code is as follows:

### Code:

```

1. def rearrange(arr):
2.     num_last_col = len(arr)%5
3.     remaining_arr = arr[:len(arr)-num_last_col]
4.     arr1 = np.reshape(remaining_arr, (len(arr)//5, 5))
5.     arr1 = arr1.T
6.     print(arr1)
7.     if(num_last_col != 0):
8.         up = (5 - num_last_col)//2
9.         arr3 = arr[5*((len(arr)//5)):]
10.        arr3= np.sort(arr3)
11.        last_col = [0]*up + list(arr3) + [0] * (5-(len(arr[5*((len(arr)//5)):])))
12.        print(last_col)
13.        #arr1 = np.hstack((arr1,np.reshape(np.array(last_col).T,
14.                                         (len(last_col),1))))
14.        print(arr1)
15.        arr1=np.sort(arr1, axis=0)
16.        arr1 = np.hstack((arr1,np.reshape(np.array(last_col).T,
17.                                         (len(last_col),1))))
17.        print(arr1)
18.        arr2 = np.sort(arr1[arr1.shape[0]//2, :])
19.        print(arr2)
20.        if (arr2.shape[0]%2 ==0):
21.            median = (arr2[arr2.shape[0]//2 -1] + arr2[arr2.shape[0]//2
22.                ])/2
22.        else:

```

```
23. median = arr2[arr2.shape[0]//2 ]  
24. print(median)  
25. arr = [2, 7, 6, 8, 10, 1, 3, 5, 4, 9, 11, 15]  
26. print(arr)  
27. rearrange(arr)
```

### Output:

```
[2, 7, 6, 8, 10, 1, 3, 5, 4, 9, 11, 15]  
[[ 2 1]  
 [ 7 3]  
 [ 6 5]  
 [ 8 4]  
 [10 9]]  
[0, 11, 15, 0, 0]  
[[ 2 1]  
 [ 7 3]  
 [ 6 5]  
 [ 8 4]  
 [10 9]]  
[[ 2 1 0]  
 [ 6 3 11]  
 [ 7 4 15]  
 [ 8 5 0]  
 [10 9 0]]  
[ 4 7 15]  
7
```

We can also create two sets, L and R, by using the median of medians,  $m$ , obtained previously. The set  $L$  will contain all the elements less than  $m$ , and the set R will contain all the elements greater than  $m$ . For the preceding example, the sets  $L$  and  $R$  are as follows:

$L = [2, 8, 5, 10, 11, 15, 21]$

$R = [71, 91, 67, 40, 78, 89, 90]$

These sets can further be used to calculate the first quartile, the third quartile, and so on. As a matter of fact, the procedure can be used to find the  $k^{\text{th}}$  smallest element using the algorithm stated as follows:

## **K<sup>th</sup> smallest element using Median of Medians**

- If the number of elements,  $n$ , in the given list is small, one can sort the list using any technique and return the median.
- For the larger value of  $n$ , partition the numbers into groups of five and sort the numbers within each group. Select the middle element of each group.
- Find all the elements less than the median of medians and call it  $L$ . Likewise, find  $R$  by finding all the elements greater than  $m$ .
- The rank of  $r$  is  $r = |L| + 1$ .
- To find the  $k^{\text{th}}$  smallest element
  - if  $r == k$  return.
  - If  $k < r$ , then return  $k^{\text{th}}$  smallest element of the set  $L$ .
  - If  $k > r$ , then return  $(k - r)^{\text{th}}$  smallest element of the set  $R$ .

The reader is expected to apply the preceding method to find the third smallest element from the preceding list using this method.

## **Median using heaps**

Having seen two methods of finding the median, let us move to another method that can also be used to find the median of an inline list. This method involves creating two heaps: a max-heap and a min-heap. The max-heap will contain the smaller half of the numbers obtained so far, and the min-heap will contain the larger half. The algorithm for finding the median of the number stream is as follows:

## **Median using heaps**

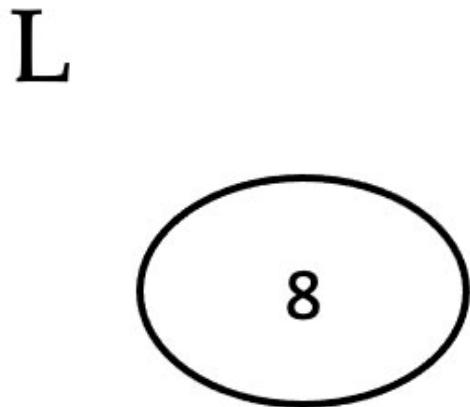
For each incoming integer:

- If the integer is greater than the root of the min-heap, add it to the min-heap, or else to the max heap.
- If, after the previous step, the difference between the numbers of elements in the two heaps is greater than 1, then we take out an element from the heap containing more elements and rearrange the heaps.
- Finally, if the number of elements in the two heaps is the same, then the average of the roots of the two heaps is returned; else, the root of the heap having a greater number of elements is returned.

To understand this algorithm, consider the following example:

**Problem:** You are required to find the median at each step for a number stream: [8, 11, 5, 15, 4] using the preceding method.

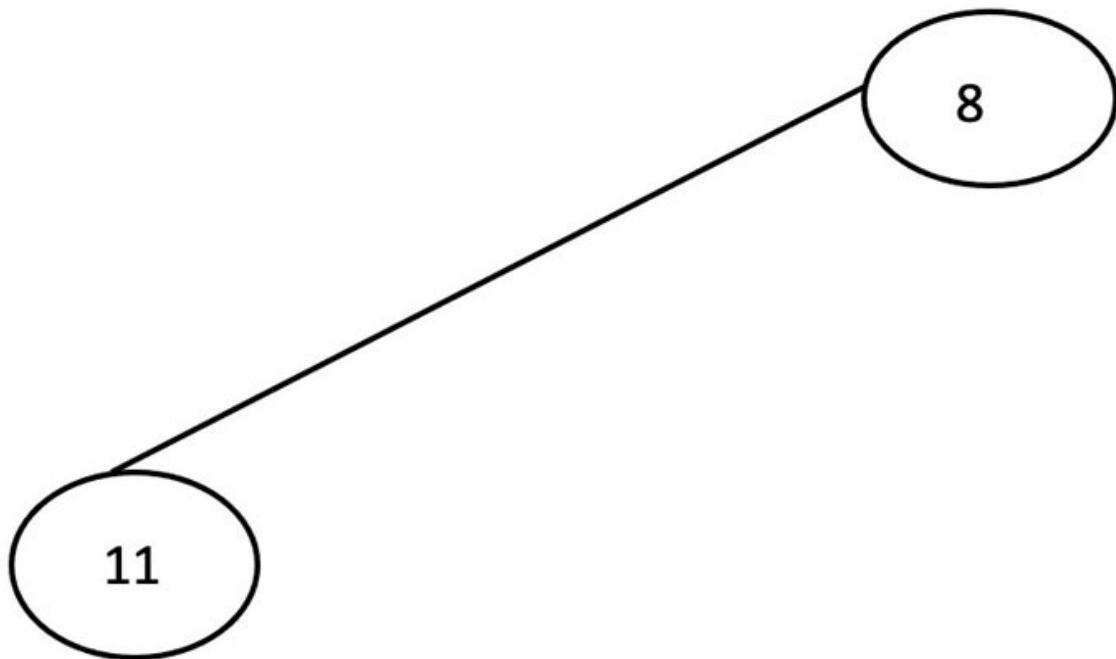
**Step 1:** The first element (8) input into the min-heap as shown in [figure 13.1](#):



*Figure 13.1: The configuration of the min-heap after inserting 8*

**Step 2:** The second element (11) is greater than the root of the min-heap, so it is inserted into the min-heap. After this insertion, the difference in the number of elements in the two heaps is greater than one, as shown in [figure 13.2](#):

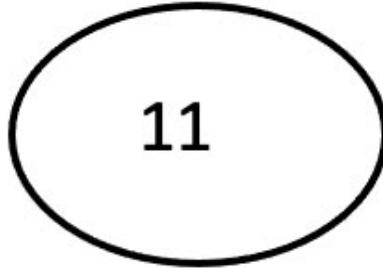
L



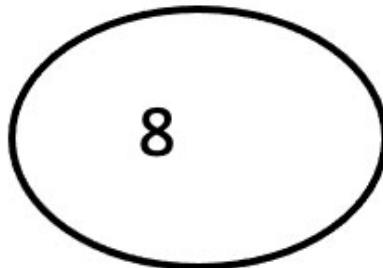
**Figure 13.2:** The configuration of the min-heap after inserting 11

Therefore, we pop an element from the min-heap heap and put it in the max-heap, as shown in [figure 13.3](#):

L

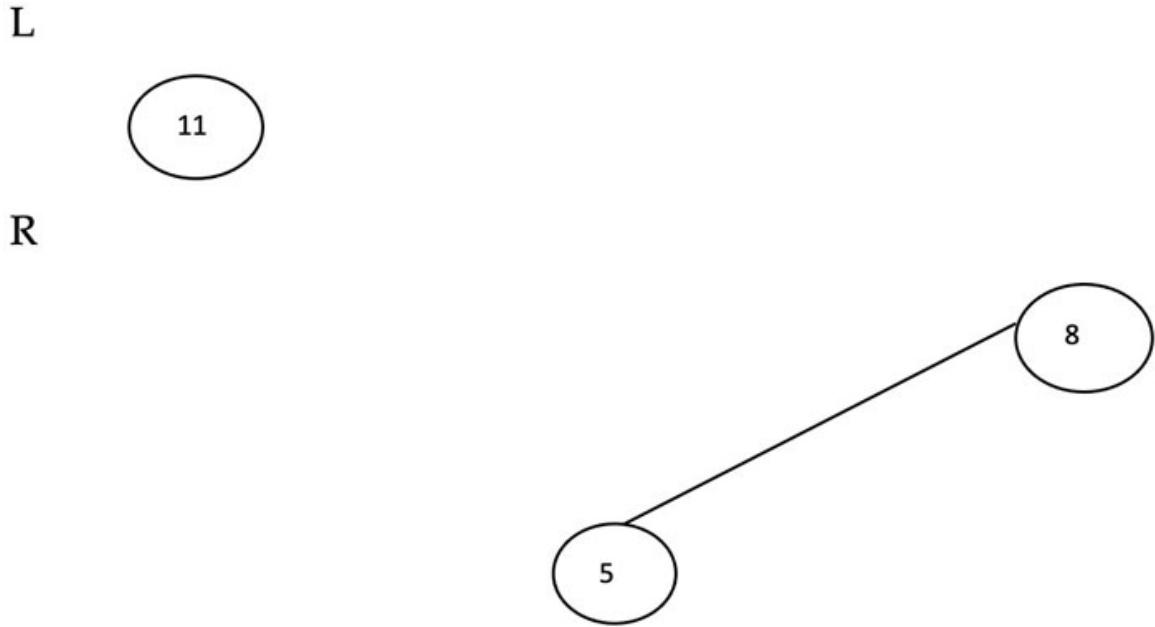


R



*Figure 13.3: The configuration of the min-heap and the max-heap after inserting 8 and 11*

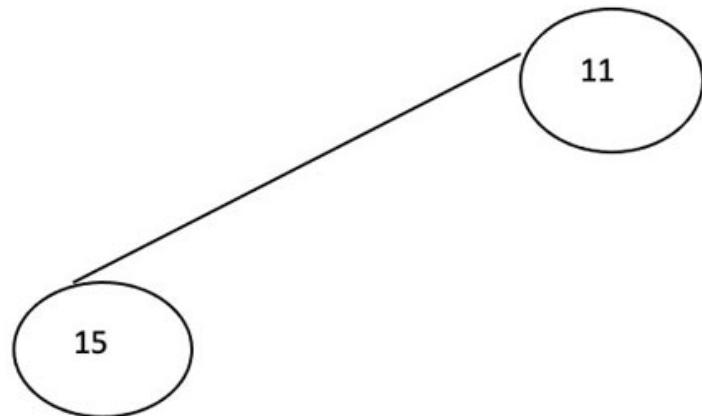
**Step 3:** The next element is 5. Since it is less than the root of the max-heap, it is pushed into the max-heap as shown in [figure 13.4](#):



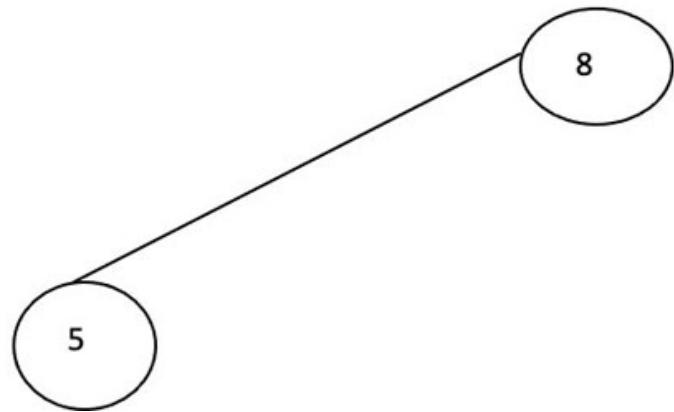
**Figure 13.4:** The configuration of the min-heap and the max-heap after inserting 8, 11, and 5

**Step 4:** The next element (15) is greater than the root of the min-heap; hence, it is inserted into the min-heap as shown in [figure 13.5](#):

L

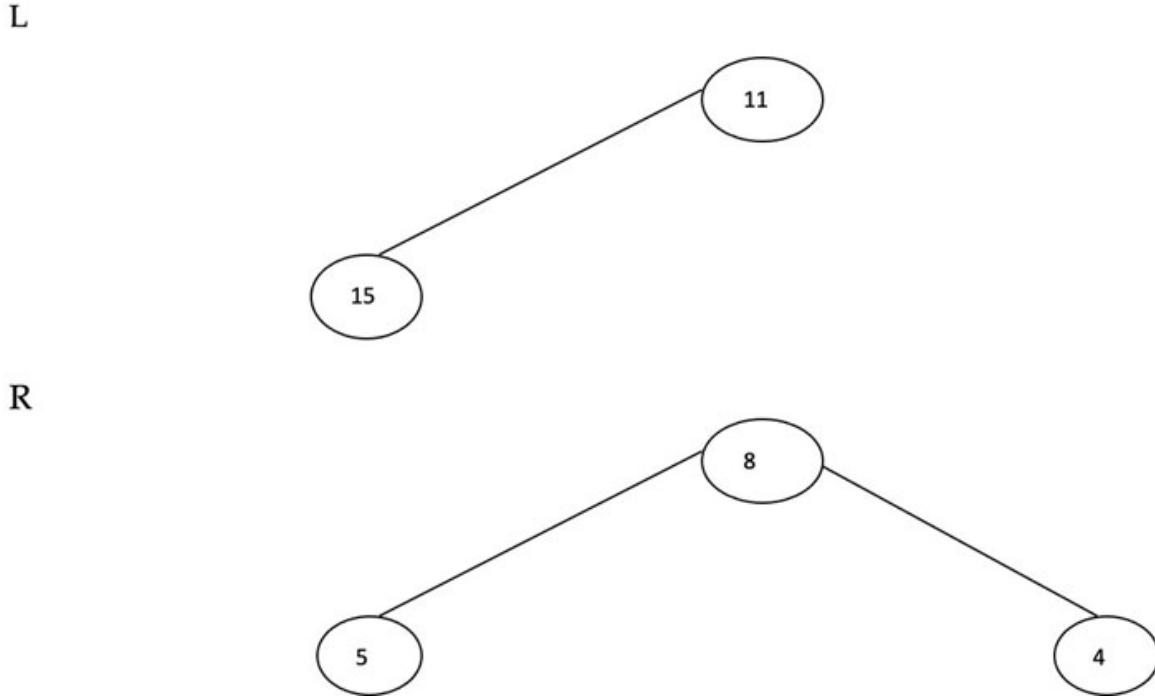


R



**Figure 13.5:** The configuration of the min-heap and the max-heap after inserting 8, 11, 5, and 15

**Step 5:** The next element (4) is less than the root of the max-heap; hence, it is inserted in the max-heap as shown in [figure 13.6](#):



**Figure 13.6:** The configuration of the min-heap and the max-heap after inserting 8, 11, 5, 15, and 4

The median at each step is shown in [table 13.1](#). The table also states how the median at each step is calculated:

Step	Median	Comments
Step 1	8	Only one element in L
Step 2	9.5	Both heaps have one element each; hence, the median is the average of the elements that are the two roots.
Step 3	8	L has two elements, and R has one. The median is the element at the root of L.
Step 4	9.5	Both heaps have two elements each; hence, the median is the average of the elements that are the two roots.
Step 5	8	L has three elements, and R has two. The median is the element at the root of L.

**Table 13.1:** Median at each step

The code is as follows:

### Code:

```

1. from heapq import heappush, heappop, heapify
2. def FindMedian_Heap(arr, N):

```

```

3. heapL = []
4. heapR = []
5. heappush(heapL, -arr[0])
6. if(arr[1]< heapL[0]):
7.     heappush(heapL, -arr[1])
8.     heappush(heapR, -heappop(heapL))
9. else:
10.    heappush(heapR, arr[1])
11. for i in range(2, len(arr)):
12.     #print(heapL[0], heapR[0], arr[i])
13.     if(arr[i]<-1*heapL[0]):
14.         heappush(heapL, -1*arr[i])
15.     elif (arr[i]> heapR[0]):
16.         heappush(heapR, arr[i])
17.     if (len(heapL) > (len(heapR)+1)):
18.         heappush(heapR, -1*heappop(heapL))
19.     if (len(heapR) > (len(heapL)+1)):
20.         heappush(heapL, -1*heappop(heapR))
21.     if len(heapR) != len(heapL):
22.         print('Median\t', -heapL[0])
23.     else:
24.         print('Median\t:', (heapR[0] - heapL[0])/2)
25.         print('Left\t', heapL)
26.         print('Right\t', heapR)
27. if __name__ == '__main__':
28. arr = [14, 17, 21, 89, 2, 3, 78, 90, 23, 90, 12, 65, 23]
29. n = len(arr)
30. FindMedian_Heap(arr, n)

```

Note that the median of the stream of elements can also be found using insertion sort, as discussed in the following section.

## Median using insertion sort

We have already studied insertion sort in *Chapter 11, Sorting*. At each step in the insertion sort, the incoming element is inserted at its appropriate position. The middle element of the list created so far is the median at this step. The reader is requested to revisit the chapter for the same.

## Median using partition

Consider the partition algorithm of Quick Sort. The algorithm finds out the correct position of the pivot element in the given array. That is, the algorithm places the pivot at position  $k$  such that all the elements to the left of the pivot are smaller than the pivot, and all to the right are greater than the pivot.

Now, assume that you need to find the  $k^{\text{th}}$  smallest element in a given array. You choose an element, find its correct position if the position matches  $k$ , and return the element; else, we check if the value returned by the algorithm (ptr)  $< n/2$  (where  $n$  is the length of the array), set  $l = \text{ptr} + 1$ ; else if (ptr)  $> n/2$ , set  $r = \text{ptr} - 1$ .

The code is as follows:

### Code:

```
1. def partition(l, r, arr1):
2.     pivot, ptr = arr1[r], l
3.     for i in range(l, r):
4.         if arr1[i] <= pivot:
5.             arr1[i], arr1[ptr] = arr1[ptr], arr1[i]
6.             ptr += 1
7.     arr1[ptr], arr1[r] = arr1[r], arr1[ptr]
8.     return ptr
9. def FindMedian(arr):
10.    l = 0
11.    r = len(arr)-1
12.    while(1):
13.        ptr = partition(l, r, arr)
14.        if (ptr == len(arr)//2):
```

```

15. print(arr[ptr])
16. break
17. elif( ptr <len(arr)//2):
18. l = ptr + 1
19. else:
20. r = ptr-1
21. print(l, r, ptr)
22. arr = [2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
23. FindMedian(arr)

```

### **Output:**

0 7 8  
0 5 6  
5 5 4  
6

## **Conclusion**

In this chapter, we studied about finding the median, quartile, and so on, using a conventional algorithm that has a computational complexity of  $O(n \log n)$ . These are immensely important tasks, and hence, efficient and effective methods are needed to accomplish these tasks. This chapter discussed methods to find the  $k^{\text{th}}$  order statistics and those of finding the median of number streams. Our journey has not ended.

The upcoming chapter discusses hashing, which can be used to solve numerous problems efficiently. This technique is more efficient than linear and binary search.

The Appendix given at the end of this book has some assorted topics and questions based on the topics discussed in this book.

## **Solved problems**

1. You are given a list of unsorted elements. How will you find the  $k^{\text{th}}$  largest element? State the complexity of the algorithm.

**Solution:** The elements can be sorted using heap sort (non-increasing order). The  $k^{\text{th}}$  element from this sorted list is the required element.

**Complexity:** The complexity of sorting is  $O(n \log n)$ , and the selection of the  $k^{\text{th}}$  element from the sorted list can be done in  $O(1)$  time. The auxiliary space requirement is  $O(1)$ .

2. The partition algorithm of Quick Sort is used to find the  $k^{\text{th}}$  largest element. What is the best-case complexity of the algorithm?

**Solution:** The best-case complexity of the selection algorithm for finding the  $k^{\text{th}}$  largest element is  $O(1)$ .

3. In the preceding question, what is the worst-case complexity?

**Solution:** The worst-case complexity of the selection algorithm for finding the  $k^{\text{th}}$  largest element is  $O(n^2)$ .

4. What is the auxiliary space requirement for the preceding algorithm?

**Solution:** The auxiliary space requirement for the preceding algorithm is  $O(1)$ .

5. Binary search is used to find the  $(n/4)^{\text{th}}$  smallest element from a sorted array. What is the complexity of the algorithm?

**Solution:** The complexity of this procedure is  $O(1)$ .

6. In the preceding question, if one uses the max-heap. What will be the complexity?

**Solution:** The given elements can be inserted in a max heap. This is followed by extracting  $K$  elements from the heap. The last element extracted from the heap is the  $k^{\text{th}}$  largest element.

**Complexity:** The creation of a heap takes  $O(\log n)$  time. Extracting  $k$  elements requires re-heapifying the heap  $k$  times. The time complexity of the given algorithm is, therefore,  $O(k \log n + n)$ . The auxiliary space requirement is  $O(n)$ .

7. Can we accomplish the preceding task using a single min heap?

**Solution:** Yes, we can accomplish the task using a min-heap. However, the number of extractions from the heap will be  $(n-k)$  in place of  $k$ . Thus, making the complexity of the algorithm as  $O((n-k) \log n)$ , provided  $k$  is small as compared to  $n$ .

8. Will the complexity improve by using two heaps: one max-heap and one min-heap, to find the  $k^{\text{th}}$  largest element?

**Solution:** The reader is expected to read the chapter and figure out the answer.

9. Consider a list of 15 numbers [21, 14, 4, 6, 9, 8, 11, 3, 89, 1, 7, 56, 2, 23, 16]. Split the numbers into a group of three, sort them individually, and take out the second row of the matrix so formed. Now, find the median.

**Solution:**

[[ 4 6 3 1 2]

[14 8 11 7 16]

[21 9 89 56 23]]

The middle row is sorted)

[7 8 11 14 16]

Finally, the median of the middle row is

11

10. Repeat the preceding question taking five elements in each column.

**Solution:**

[[ 4 1 2]

[ 6 3 7]

[ 9 8 16]

[14 11 23]

[21 89 56]]

[ 8 9 16]

9

11. The preceding two methods (Q9 and Q10) give different answers. Why is it the case? Which do you think is correct and why?

**Solution:** The sorted array is

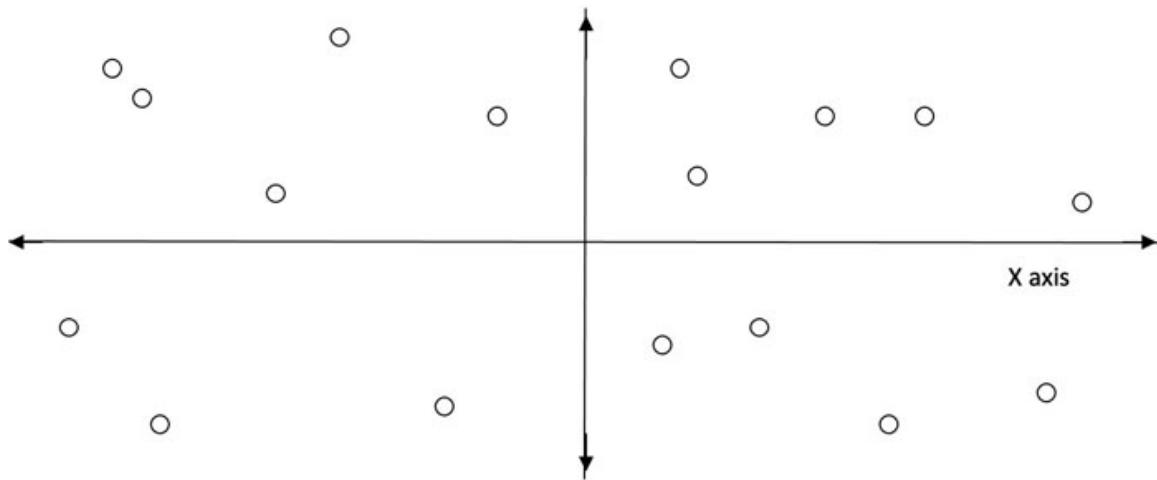
[ 1 2 3 4 6 7 8 9 11 14 16 21 23 56 89]

The median is, therefore, the 8th element, which is

9

So, the method adopted in Q10 is correct. The readers are expected to figure out why taking columns of length three does not work.

12.  $N$  people (each depicted by a circle), standing as shown in the following figure, are waiting for a bus. Each person has to walk in a straight line towards the path at which a bus will come (X1 axis). Find the condition that the total distance traveled by all is the minimum.



**Figure 13.7:** Diagram for Q12

**Solution:** The coordinates of all the persons are given. Note that the X1 axis should be in the median of the y coordinates to minimize the distance.

13. In the preceding question, if all the people are required to travel along a line at an angle of 30 degrees to the original Y axis. How will you find X1?

**Solution:** The reader is expected to figure out the solution by himself/herself. (Hint: Use transformation of coordinates).

14. Two sorted lists are given; find the median of all the elements taken together.

**Solution:** Merge two sorted lists and find the median of the resultant.

15. Can you find the median in the preceding question using heaps?

**Solution:** Yes.

## Multiple choice questions

1. An array of 10,000 numbers is given. We need to find the median by sorting the numbers and selecting the middle element if the number of elements is odd (or the average of the two middle elements if the number of elements is even). The complexity of the preceding method (provided we selected the most suitable sorting algorithm is

- a.  $O(n)$
- b.  $O(n \log n)$
- c.  $O(n^2)$
- d. None of the above

**2. What is the time complexity in which the preceding task can be accomplished, using the median of median method?**

- a.  $O(n)$
- b.  $O(n \log n)$
- c.  $O(n^2)$
- d. None of the above

**3. What is the time complexity in which the preceding task can be accomplished using two heaps?**

- a.  $O(n)$
- b.  $O(n \log n)$
- c.  $O(n^2)$
- d. None of the above

**4. To find the median of a stream of numbers, which of the following sorting algorithm can be used?**

- a. Insertion sort
- b. Selection sort
- c. Bubble sort
- d. None of the above

**5. To find the median of a stream of numbers, which of the following data structure is most appropriate?**

- a. Heap
- b. Stack
- c. Queue
- d. None of the above

**6. Which of the following can be used to find the  $k^{th}$ -order statistics?**

- a. Partition
- b. Merge
- c. Both
- d. None of the above

**7. To find the second smallest number, which of the following can be used?**

- a. Sort the numbers in ascending order and select the second element from the sorted list.
  - b. Create a heap and perform two deletions
  - c. Use partition and print the element if the number returned by the algorithm is 2
  - d. All of the above
- 8. From a list of 100,000 numbers, the 9/10<sup>th</sup> largest number is to be found. Can we use the median of medians for this?**
- a. Yes
  - b. No
- 9. For the preceding question, can we use the “Median using Heaps” method?**
- a. Yes
  - b. No
- 10. In Question 8, the number of elements doubles after every run of the algorithm. In this case, would you use sorting to find the 9/10<sup>th</sup> largest number?**
- a. Yes
  - b. No

## Applications/implementation

1. Find the first and the third Quartile (Q1 and Q3) by
  - a. Sorting numbers
  - b. Using Median of Medians
  - c. Using min-heap and max-heap method
  - d. Using Partition
2. Find the best and the worst-case complexity of each of the above.
3. Generate 1,000 random numbers and run the preceding programs. Compare the running time of each.
4. Now repeat the same task with 10,000, 100,000, and 1,000,000 numbers and plot the graph of variation of running time with the number of elements for

each.

## Bibliography

1. MIT Open courseware: <https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/resources/lecture-6-order-statistics-median/>
2. Williams.edu:  
[https://web.williams.edu/Mathematics/sjmiller/public\\_html/probabilityifesaner/supplementalchap\\_mediantheoremandorderstatistics.pdf](https://web.williams.edu/Mathematics/sjmiller/public_html/probabilityifesaner/supplementalchap_mediantheoremandorderstatistics.pdf)
3. NTU:  
<http://www.math.ntu.edu.tw/~hchen/teaching/LargeSample/notes/noteorder.pdf>
4. IITKGP:  
<http://www.facweb.iitkgp.ac.in/~sourav/Lecture-06.pdf>
5. Colorado:  
[https://www.colorado.edu/amath/sites/default/files/attached-files/order\\_stats.pdf](https://www.colorado.edu/amath/sites/default/files/attached-files/order_stats.pdf)
6. Duke  
<https://www2.stat.duke.edu/courses/Spring12/sta104.1/Lectures/Lec15.pdf>
7. Penn State:  
<https://online.stat.psu.edu/stat415/book/export/html/834>
8. University of South Carolina:  
<https://cse.sc.edu/~fenner/csce750/OKane-Fall-2020/notes-selection.pdf>
9. Durham University:  
[https://www.maths.dur.ac.uk/stats/courses/Proba34/1617Probability34\\_H.pdf](https://www.maths.dur.ac.uk/stats/courses/Proba34/1617Probability34_H.pdf)
10. Bookdown.org  
<https://bookdown.org/jkang37/stat205b-notes/lecture02.html>
11. University of Washington  
[http://faculty.washington.edu/yenchic/20A\\_stat512/Lec9\\_Order.pdf](http://faculty.washington.edu/yenchic/20A_stat512/Lec9_Order.pdf)
12. University of California, Irvine

<https://www.ics.uci.edu/~eppstein/161/960125.html>

13. University of California, Berkeley

<https://www.stat.berkeley.edu/~aditya/resources/FullLectureNotes201AFall2019.pdf>

14. Stanford

<https://web.stanford.edu/class/archive/cs/cs109/cs109.1218/files/studentdrive/5.10.pdf>

## **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 14

## Hashing

So far, we have discussed various search methods. Linear Search takes  $O(n)$  time and can be applied to an unordered sequence. Binary Search, on the other hand, takes  $O(\log n)$  time and can be applied to a sorted sequence. Other methods, like AVL trees, have also been discussed in the previous chapters. This chapter takes the discussion forward and introduces hashing.

After reading this chapter, the readers will be able to implement hash tables and handle problems encountered in creating such tables. Furthermore, many problems can be solved using hashing. This chapter also discusses some of these interesting problems.

The chapter also throws light on the selection of the hash function, selecting the size of the table, and avoiding collisions, which are some of the key issues in hashing.

### Structure

In this chapter, we will cover the following topics:

- Hash tables
- Storing information
- Hashing
- Collision resolution
- Problems

### Objectives

This chapter begins with the hash tables. After reading this chapter, you will be able to understand the importance of these tables, with the next section revisiting various methods of storing information and the complexity of the operations, such as insertion, deletion, and searching. This is followed by Hashing and the issues encountered in implementing it. An informed

discussion on various collision resolution techniques follows. The next section presents some of the interesting problems and uses hashing to solve these problems.

## **Hash tables**

Information may be considered as a list of records, each one of which contains some fields. This can be represented as a table having the number of rows equal to the number of records and the number of columns equal to the number of fields. For example, consider a table having records of employees; each record having name, date of birth, salary, department, phone number, and address. You can search the record of an employee by using his/her name; that is, to search the record of a particular employee, we do not need all fields but only the key (which, in this case, is the name). The common operations of this abstract data type (table) are as follows:

- **Create table:** This operation defines the structure of the table
- **Insert:** This operation inserts a record in the table, given its key and value
- **Search:** The search operation searches the value corresponding to a given key from a table

The creation of this table depends on the following factors:

- The number of records to be inserted in the table/deleted from the table
- How the records would be searched from the table
- The number of keys
- Whether the table is small enough to fit in the memory
- The duration for which it will stay in the memory

This table can be created in many ways, some of which are discussed in the next section.

## **Storing information**

The abstract data type called table can be represented in many ways. The records can be stored as a table having index, key, and entity as fields. For

example, the following table ([table 14.1](#)) contains four records  $\{(0, 14, 1234), (1, 45, 2345), (2, 19, 3456), \text{ and } (3, 27, 4567)\}$

Index	Key	Entity
0	14	1234
1	45	2345
2	19	3456
3	27	4567

**Table 14.1:** An example of an unsorted sequential array

The complexity of initializing this table is  $O(n)$ , that of finding it is `IsEmpty` is  $O(1)$ , inserting an element is  $O(1)$ , finding an element is  $O(n)$ , and removing an element is  $O(n)$ . Note that to check if the array is empty, we need to check the first record only. Likewise, to search for an element from the array, the complexity will be  $O(n)$ , as shown in [table 14.2](#):

Operation	Complexity
Initialize	$O(1)$
Is Empty	$O(1)$
Insert	$O(1)$
Find	$O(n)$
Remove	$O(n)$

**Table 14.2:** Complexity of various operations in unsorted sequential arrays

## Sorted sequential array

Another way of storing records would be storing the keys in sorted order, as shown in [table 14.3](#). This will help in applying algorithms like Binary Search while searching the keys.

Index	Key	Entity
0	14	1234
1	19	3456
2	27	4567
3	45	2345

**Table 14.3:** Example of sorted sequential array

The complexity of initializing this table is  $O(n)$ , that of finding it is `IsEmpty` is  $O(1)$ , inserting an element is  $O(1)$ , finding an element is  $O(\log n)$ , and removing an element is  $O(n)$ . Note that to check if the array is empty, we need to check the first record only. Likewise, to search an element from the array, the complexity will be  $O(\log n)$ , as shown in [table 14.4](#):

Operation	Complexity
Initialize	$O(1)$
<code>IsEmpty</code>	$O(1)$
Insert	$O(1)$
Find	$O(\log(n))$
Remove	$O(n)$

**Table 14.4:** Complexity of various operations in sorted sequential arrays

## Linked list representation

These records can also be stored using a linked list. The complexity of various operations is shown in [table 14.5](#):

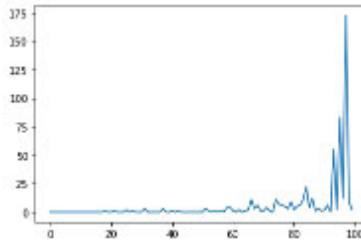
Operation	Complexity
Initialize	$O(n)$
<code>IsEmpty</code>	$O(1)$
Insert	$O(n)$
Find	$O(n)$
Remove	$O(n)$

**Table 14.5:** Complexity of various operations using linked list representations

## AVL trees

AVL trees can also be used for representing this data. This will make insertion, removal, and finding element  $O(\log n)$ .

[Figure 14.1](#) shows the variation of time of execution with the number of elements:



*Figure 14.1: Example of Ordered Binary Tree*

[Table 14.6](#) shows the complexity of various operations in an ordered binary tree:

Operation	Complexity
IsEmpty	$O(1)$
Insert	$O(\log n)$
Find	$O(\log n)$
Remove	$O(\log n)$

*Table 14.6: Complexity of various operations in an ordered binary tree*

Even the preceding complexity is not good. The following discussion throws light on the technique called hashing, which will further reduce the complexity of various operations.

## Hashing

Hashing is a technique that helps achieve the complexity of  $O(1)$  in search operations. Hashing can be implemented using arrays. In implementing hashing using arrays, we need a hash function that takes the key as the input and produces the index at which the data is to be placed, as shown in [figure 14.2](#):



*Figure 14.2: Hash function*

The key calculated using the hash function is called Hashed Key. Note that hashing is used if the searching is frequent, whereas if insertions and deletions are frequent, then ordered binary trees are good.

## Hash function

A good hash function should distribute keys uniformly. If the hash function generates the same index for two or more keys, it is called a **collision**. A good hash function should result in a minimum number of collisions. Along with these properties, it should be easy to calculate. To summarize:

A good Hash function has the following features:

- Easy to calculate
- Should result in the minimum number of collisions
- Should distribute keys uniformly

As stated, a hash function may result in a collision. The following sub-section throws light on the collision resolution techniques.

## Collision resolution

Collisions can be reduced using the following methods:

**Open addressing:** If the index generated by the hash function for a particular key is the same, then the key can be stored in the next free location found by incrementing the index generated by a certain number.

**Re-Hashing:** In re-hashing, we create a table of larger size and copy the contents of the current hash table in it. Note that the size of the table, is typically a prime number. The reader may refer to Dynamic Arrays for getting hold of this concept.

**Chaining:** This method requires a list to be mainlined with each index. In case of collision, the elements are added to this list.

These techniques have been discussed in detail in the following section.

## Selecting hash function

The hash function can be calculated using the following functions:

## **Modular Arithmetic**

In this method, the key is divided by a number, and the remainder is taken.

## Truncate

In this method, we can ignore some parts of the key and take the rest as the index. For example, we can take the last three digits of the given key if the table size is 1,000. However, it may not lead to even distribution throughout the table.

## Folding

In folding, the key is divided into some parts, and the parts are subjected to some mathematical operations to generate a key.

The Appendix gives a list of various hash functions and their examples.

## Collisions

To understand the concept, consider an array of size 10 and hash function as follows:

**key mod 10**

Now, consider the following keys:

23, 12, 18, and 27

The indices of the preceding keys are then calculated as follows:

$$23\%10 = 3$$

$$12\%10 = 2$$

$$18\%10 = 8$$

and  $26\%10 = 6$ . So far, none of the indices generated by the hash function are repeated ([table 14.7](#)).

0	
1	
2	12
3	23
4	
5	
6	26
7	

8	18
9	

**Table 14.7:** An example of a Hash table

Now, consider key 38; the index generated by the hash function is 8.

This results in a collision. Assume that we use linear probing (explained in the next section) and, in case of collision, store the item in the next available index. The table will become [table 14.8](#):

0	
1	
2	12
3	23
4	
5	
6	26
7	
8	18
9	38

**Table 14.8:** Handling collision using Linear probing

The following section discusses collision resolution in detail.

## Collision resolution

This section discusses various techniques of collision resolution and presents the implementation. Note that the *load factor* of a Hash table may be defined as the number of positions filled to the size of the hash table. It is generally denoted by  $\lambda$ . I, in the first technique,. Note that the first two techniques are examples of *Open Addressing*.

### Linear probing

Linear probing is one of the collision resolution methods. In this method, if a collision occurs (that is, the hash function generates the same index for more than one key), the index at which the key is to be inserted becomes:

$$\text{index} = \text{index} + f(i)$$

The value of  $f(i)$  is typically  $i$ . That is, if the index generated by the hash function is, say, 8, and this location is already occupied, then the index becomes:

$$\text{index} = \text{index} + f(i) = 8 + 1 = 9$$

If this location is also occupied, then the index increments by 1, and so on. Note that in this example, we increment the index by 1 each time collision occurs, but this number can also be any other constant number (other than 1). This method leads to a problem called **Primary Clustering**. Here, many filled slots are created near the indices generated by the hash functions.

The following program implements Linear Probing and analyses the number of collisions as the array gets filled. Note that as the array is getting filled, the number of collisions increases substantially, as shown in [figure 14.3](#).

The code is as follows:

## Code

#Importing Libraries

```
1. import numpy as np  
2. from matplotlib import pyplot as plt  
3. import time
```

#Hash Function

```
1. def findHash(num, table1):  
2.     return (num%table1.shape[0])
```

#Linear Probing Function

```
1. def insertVal(num, table1):  
2.     miss=0  
3.     index = findHash(num, table1)  
4.     #print(index)  
5.     while(table1[index]!=0):
```

```

6. index = (index+1)%table1.shape[0]
7. miss+=1
8. table1[index]=num
9. return miss

```

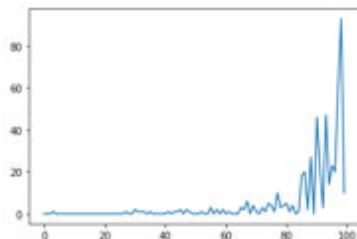
### #Implementing Linear Probing Function

```

1. t1= time.time()
2. table1 = np.zeros(100)
3. numbers = np.random.randint(1, 10000, 100)
4. #Basic function
5. misses=[]
6. for num in numbers:
7.     m=insertVal(num, table1)
8.     misses.append(m)
9. t2= time.time()
10. t=t2-t1
11. print(t)
12. x=np.arange(table1.shape[0])
13. plt.plot(X,misses)

```

### #Output



**Figure 14.3:** Variation of the number of collisions in Linear Probing

Having seen Linear Probing, let us now move to Quadratic Probing.

## Quadratic probing

Quadratic probing is another collision resolution method. In this method, if a collision occurs (that is, the hash function generates the same index for more than one key), the index at which the key is to be inserted becomes:

$$\text{index} = \text{index} + f(i)$$

The value of  $f(i)$  is typically  $i^2$ . That is, if the index generated by the hash function is, say, 8, and this location is already occupied, then the index becomes:

$$\text{index} = \text{index} + f(i) = 8 + 1^2 = 9$$

If this location is also occupied, then the index increments by 4, and so on (note that the order is  $1^2, 2^2, 3^2$  and so on). Here, can also be any other function that is quadratic. This method solves the problem of **Primary Clustering** but may lead to cluttering at some positions other than those generated by the hash function. The problem is generally referred to as **Secondary Clustering**.

The following program implements Quadratic probing and analyses the number of collisions as the array gets filled. Note that as the array is getting filled, the number of collisions increases substantially, as shown in [figure 14.4](#):

#Hash Function

```
1. def findHash(num, table1):  
2.     return (num%table1.shape[0])
```

#Quadratic Probing Function

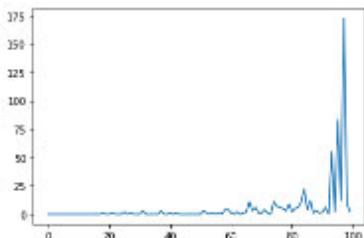
```
1. def insertVal1(num, table1):  
2.     miss=0  
3.     index = findHash(num, table1)  
4.     #print(index)  
5.     i=1  
6.     while(table1[index]!=0):  
7.         index = (index+(i**2))%table1.shape[0]  
8.         miss+=1
```

```
9.    i+=1  
10.   table1[index]=num  
11.   return miss
```

## #Implementing Quadratic Probing Function

```
1. t1=time.time()  
2. table1 = np.zeros(100)  
3. numbers = np.random.randint(1, 10000, 100)  
4. #Basic function  
5. misses1=[]  
6. i=1  
7. for num in numbers:  
8.     #print(i)  
9.     m=insertVal1(num, table1)  
10.    misses1.append(m)  
11.    i+=1  
12.    print(i)  
13. t2=time.time()  
14. t=t2-t1  
15. print(t)  
16. x=np.arange(table1.shape[0])  
17. plt.plot(x,misses1)
```

## #Output

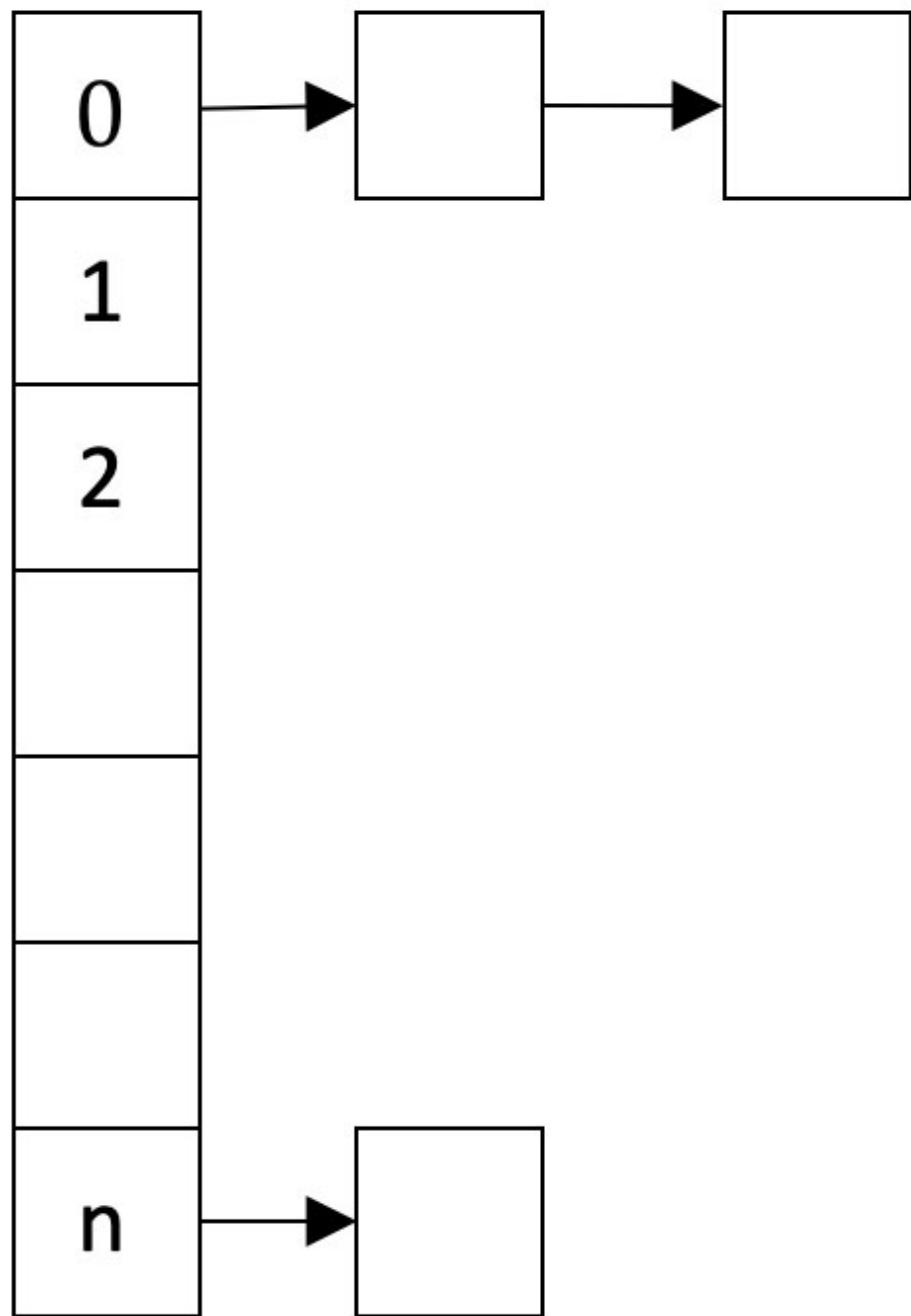


**Figure 14.4:** Variation of the number of collisions in quadratic probing

The following collision resolution technique uses a linked list to handle the problem stated previously.

### Separate chaining

In this method, a linked list is created with each index. If the **hash** function generates the same index for more than one key, the generated keys are inserted in the liked list corresponding to the index. [Figure 14.5](#) shows an example of separate chaining:



*Figure 14.5: Chaining*

Having seen various collision resolution techniques let us solve some interesting problems using Hashing.

## Solved problems

**Problem 1:** Given an array of random numbers, find the frequency of each element in the array.

**Solution:**

**Method 1:** The given task can be accomplished using brute force. This method takes the  $i^{\text{th}}$  element of the array ( $i$  varying from 0 to size -1) and compares it with the rest of the elements. This process can be implemented using a nested loop in  $O(n^2)$  time. The code is as follows:

**Code:**

```
1. import numpy as np
2. arr = np.random.randint(1, 100, 200)
3. count1={}
4. i=0
5. arr2= np.copy(arr)
6. #print(arr2)
7. for x in arr2:
8.     for j in range(i+1,arr2.shape[0]):
9.         item=arr2[j]
10.        if item== x:
11.            if x in count1:
12.                count1[x]+=1
13.                arr2[j]=-1
14.                print(j)
15.        else:
16.            count1[x]=1
17.    i+=1
```

**Method 2:** The task can also be accomplished using Counting Sort, explained in the Appendix of this book. Note that the second step of Counting Sort finds the frequency of each element.

**Method 3:** The problem can also be solved using sorting. The given array is sorted, and then the consecutive elements can be compared. This way, we will be able to find the repeated elements.

The code is as follows:

**Code:**

```
1. import numpy as np  
2. arr = np.random.randint(1, 100, 200)  
3. arr1 = np.sort(arr)  
4. count={}  
5. count[arr1[0]]=1  
6. for i in range(1, arr1.shape[0]):  
7.     if(arr1[i] not in count):  
8.         count[arr1[i]]=1  
9.     if(arr1[i] == arr1[i-1]):  
10.        count[arr1[i]]+=1
```

**Method 4:** The fourth method of doing this task is to use a hash table. The elements can be inserted in the hash table, and a counter can be maintained. While inserting a new element in the table, we check if the number exists in the hash table. If the number already exists in the hash table, the value of the count corresponding to that number can be increased by 1.

The code is as follows:

**Code:**

```
1. import numpy as np  
2. from collections import defaultdict  
3. arr = np.random.randint(1, 100, 200)  
4. table1 = defaultdict()  
5. count2={}  
6. for x in arr:
```

```
7. index = hash(x)%arr.shape[0]
8. if x not in table1:
9.     table1[index]= x
10.    count2[x]=1
11. else:
12.     count2[x]+=1
```

**Problem 2:** Given an array containing random numbers, how do you remove duplicates from this array?

**Solution:** Each of the previous strategies can be applied to accomplish this task. Note that the elements of the hash table in Method 4 are actually the unique elements of the given array.

**Problem3:** Given two arrays, X and Y, how do you find if the two arrays contain the same elements (not necessarily in the same order).

**Solution:** Create a hash table and corresponding track of counts of keys for X. Now, check if each element of Y exists in the hash table of X, also create a count track for the elements of Y. If any element of Y is not in the hash table, print “Not Same”, else if the count array of X is same as that of Y, declare the two to be same.

The code is as follows:

### Code:

```
1. import numpy as np
2. from collections import defaultdict
3. arr = np.random.randint(1, 100, 200)
4. table1 = defaultdict()
5. countX={}
6. countY={}
7. for x in X:
8.     index = hash(x)%arr.shape[0]
9.     if x not in table1:
```

```

10.     table1[index]= x
11.     countX[x]=1
12. else:
13.     countX[x]+=1
14. for y in Y:
15.     #print(y)
16.     if (y in table1):
17.         #print(<In table>)
18.         if y in countY:
19.             #print(<In countY'>)
20.             countY[y]+=1
21.             #print(<Count <,countY[y]>)
22.         else:
23.             countY[y]=1
24.             #print(<Count <,countY[y]>)
25.     else:
26.         print('Not same')
27.
28. if(countX == countY):
29.     print('Same')

```

**Problem 4:** Given two arrays, X and Y, find the common elements.

**Solution:** Create a hash table for X and one for Y. Now for each find if exists in the hash table of Y, in which case print.

**Code:**

```

1. import numpy as np
2. from collections import defaultdict
3. X = np.random.randint(1, 100, 200)
4. Y = np.random.randint(1, 100, 200)

```

```

5. tableX = defaultdict()
6. for x in X:
7.     index = hash(x)%X.shape[0]
8.     if x not in tableX:
9.         tableX[index]= x
10. tableY = defaultdict()
11. for y in Y:
12.     index = hash(y)%Y.shape[0]
13.     if y not in tableY:
14.         tableY[index]= y
15. k=50
16. for x in tableX:
17.     if k+x in tableY:
18.         print('(',x, k+x, ')')

```

## Conclusion

The chapter discussed hashing and compares various searching techniques *vis-à-vis* the complexity of operations such as insertion, deletion, and searching. The characteristics of a good hash function and its implementation were also discussed in the chapter. Collision resolution techniques such as linear probing, quadratic probing, and chaining have been discussed and implemented. Furthermore, the application of hashing to solve problems has also been discussed.

The reader should be able to implement hash tables and use hashing to solve problems efficiently after reading this chapter.

The upcoming chapter takes the discussion further and discusses various algorithms related to String Matching. For now, let us hit the exercises.

## Multiple choice questions

1. Hashing results in an average search time of:

- a.  $O(1)$
  - b.  $O(n)$
  - c.  $O(\log n)$
  - d. None of the above
- 2. If the insertion and deletion are to be done frequently, then which of the following data structures is appropriate?**
- a. Hash tables
  - b. AVL trees
  - c. Binary Search Trees
  - d. None of the above
- 3. A good hash function should**
- a. Distribute the keys uniformly in the table
  - b. Should result in less collisions
  - c. Should be easy to compute
  - d. All of the above
- 4. Which of the following results in primary clustering?**
- a. Linear Probing
  - b. Quadratic Probing
  - c. Chaining
  - d. None of the above
- 5. Which of the following results in secondary clustering?**
- a. Linear Probing
  - b. Quadratic Probing
  - c. Chaining
  - d. None of the above
- 6. For which of the following load factor can be greater than 1?**
- a. Linear Probing
  - b. Quadratic Probing

- c. Chaining
- d. None of the above

**7. Finding if an element is repeated in a given array takes what time using Hashing?**

- a.  $O(1)$
- b.  $O(n)$
- c.  $O(n^2)$
- d. None of the above

**8. Is hashing appropriate for sorting the elements efficiently?**

- a. No
- b. Yes

**9. Which of the following is a method of collision resolution?**

- a. Linear probing
- b. Quadratic probing
- c. Chaining
- d. All of the above

**10. Which of the following should preferably be the size of the hash table?**

- a. Prime number
- b. Non-prime number
- c. Any of the following
- d. Depends on the problem

## **Theory**

1. What is hashing? What is a hash table? State the characteristics of a good hash function.
2. What is meant by collision? State various methods of collision resolution.

3. What is linear probing? It may lead to primary clustering. Explain.
4. What is Quadratic probing? What is the downside of resolving collision using this method?
5. What is Chaining? What is the downside of resolving collision using this method?

## **Problems**

1. Given an array of random numbers, find if any number is repeated thrice.
2. Given an array containing random numbers, how do you find unique elements from this array in  $O(n)$  time?
3. Given two arrays, X and Y. How do you find if none of the elements in the two arrays is the same?
4. Given a string, how do you find out repeated characters from it?
5. Given two arrays, X and Y, find the elements which are in y, but not in x.

## **Programming**

1. Create an array of 100 random numbers. Now, create a hash function of your choice. Using this hash function to insert elements in the hash table and keep track of the number of collisions. Plot a graph of the load factor and the number of collisions.
2. Repeat the preceding experiment using Quadratic probing.
3. Implement chaining and repeat the experiment given in 1.
4. Implement the problems given in the previous section.

## **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 15

## String Matching

The string matching algorithms aim to find a string, P (pattern), in a larger string, T (text). This method is used in many applications, such as for DNA matching in bioinformatics, for detecting plagiarism, in spam filtering, in digital forensics, and so on. This chapter discusses various string-matching algorithms, their advantages, and their shortcomings. The chapter begins with a brute force algorithm and then moves to more efficient implementations to accomplish this task.

In this chapter, the reader will be able to implement string-matching algorithms and, in the process, use techniques like hashing and Dynamic Programming. The topic is important as it has many applications, but the algorithms used for string matching will give a whole new perspective of the applications of the techniques learnt so far.

### Structure

This chapter covers the following topics:

- Introduction to string-matching
- Brute force algorithm
- Rabin Karp algorithm
- Knuth–Morris–Pratt algorithm

### Objectives

This chapter talks about various string-matching algorithms, their efficient implementations, and their applications. It forms the basis of the methods used in areas like Bioinformatics. In the discussion that follows, take note of the fact that when the complexity of the algorithms decreases, it becomes efficient and is able to handle the shortcomings of the previous algorithms.

## Introduction to string-matching

The matching algorithms can be broadly classified into three categories:

- Comparing characters, like in the case of
  - Brute Force Algorithm, discussed in the next section, and
  - **Knuth–Morris–Pratt (KMP)** algorithm, which makes use of the fact that in the Brute Force algorithm, the first mismatch is preceded by some earlier matches. This information can be used to make the matching efficient.
- Hashing-based algorithms, like in the case of
  - Rabin Karp Algorithm, which uses the hash value of the pattern to locate the pattern, P in the text T.
- Deterministic Finite Automata, which match the pattern by starting from the start state and matching the succeeding symbols thereafter.

## Brute force method

**Problem statement:** Two strings P and T are given. The length of P is **m**, and the length of T is **n**. The goal is to find all the occurrences of P in T.

This section discusses the Brute Force algorithm that has a time complexity of  $O(nm)$ . The algorithm uses nested loops to find the location of P in T. The index **i** starts from 0 and goes to **len(P)–1**. For each character at **i**, **j** varies from 0 to **len(P)–1**. The following code shows the implementation of the algorithm, and a sample run follows:

**Code:**

```
1. def matchBrute(T, P):  
2.     n = len(T)  
3.     m = len(P)  
4.     for i in range(n-m):  
5.         j=0  
6.         while(j<m):
```

```

7. if(T[i+j] != P[j]):
8.     break
9. j+=1
10. if(j==m):
11.     print('Match at ', i)
12. T='babababbababbabbab'
13. P='abba'
14. matchBrute(T, P)

```

## Output

**Match at 1**

**Match at 6**

**Match at 11**

**Match at 14**

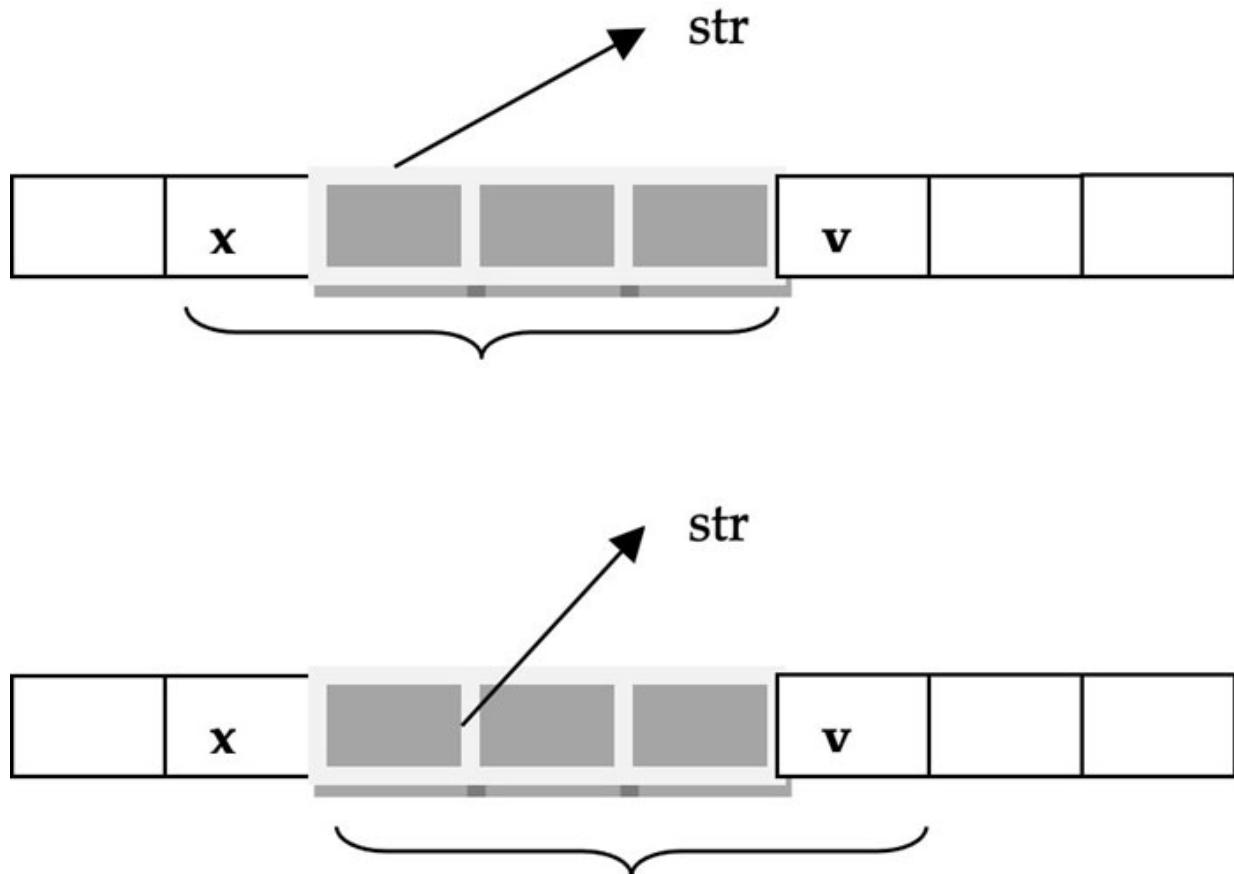
Note that the preceding algorithm takes  $O(n \times m)$  time, which can be further reduced using the algorithms discussed in the following sections.

## Rabin Karp

**Problem statement:** Two strings  $P$  and  $T$  are given. The length of  $P$  is  $m$ , and the length of  $T$  is  $n$ . The goal is to find all the occurrences of  $P$  in  $T$ .

We earlier accomplished this task using the Brute Force algorithm, which has a time complexity of  $O(n \times m)$ . The algorithm discussed in this section solves the same problem in time  $O(n)$  in the average case.

The concept is based on **rolling hash**. To understand this, assume that you are traversing the text  $T$ , and at some instance, you have a character  $x$  concatenated with string  $str$ . Somehow, the hash of  $(x + str)$  has been generated (depending upon the hash function). We shift the window by one to the right, thus leaving  $x$  and including  $y$ . To generate the hash of  $(str - x + y)$  where  $y$  is another character ([figure 15.1](#)), we need to conceptually subtract  $x$  and add  $y$ . Here,  $|x + str| = |y + str| = m$ .



**Figure 15.1:** Rolling hash

On obtaining this hash value, we compare it with the hash value of the given pattern. Now, if the two hash values match, then the pattern may be the same as that found in the text. Else, we continue finding the hash value of the shifted string in the text. To understand this, consider the following example, in which we have taken a very simple hash function that finds the sum of the ASCII values of the string. The program follows:

T = ‘abcabdacb’

P = ‘abd’ =  $97+98+100 = 295$

Rolling hash function:

‘abc’ =  $97+98+99 = 294$

‘bca’ =  $98+99+97 = 294$

‘cab’ =  $99+97+98 = 294$

‘abd’ =  $97+98+100 = 295$

‘bda’ = 98+100+97 = 295

‘dac’ = 100+97+99 = 296

‘acb’ = 97+99+98 = 294

### Code:

```
1. def findHash(P):  
2.     h = 0  
3.     for i in P:  
4.         h+= ord(i)  
5.     return h  
6. def findHash1(P):  
7.     h = 0  
8.     j=0  
9.     for i in P:  
10.        h+= ord(i) * (2**j)  
11.        j+=1  
12.    return h  
13. def findPattern(T, P):  
14.     h2 = findHash(P)  
15.     for i in range(len(T)-1):  
16.         str1 = T[i:i+len(P)]  
17.         h1 = findHash(str1)  
18.         if h1 == h2:  
19.             print('Hit at ', i)  
20. T = 'aabbbaababbababbaab'  
21. P = 'abab'  
22. findPattern(T, P)  
23. Hit at 0  
24. Hit at 3
```

```
25. Hit at 4
26. Hit at 6
27. Hit at 8
28. Hit at 10
29. Hit at 11
30. Hit at 13
31. Hit at 14
32. Hit at 15
```

Note that this method of finding out the hash of a given string is not good owing to the reasons stated in [Chapter 14, Hashing](#). This is the reason for getting many spurious hits in the preceding sample run. A string that is any permutation of a given pattern P will have the same hash value and will result in a spurious hit. A better hash function can be the following:

$$P = [P_1 \ P_2 \ P_3 \dots \ P_m]$$

$$h = ord(P_1) + 10 \ ord(P_2) + 10^2 \ ord(P_3) + \dots + 10^{m-1} \ ord(P_m).$$

In fact, the hash function can be any number raised to the power of the index of the character at the string. For the sake of simplicity, we take this number as 2 in the following program. This will result in different hash values for permutations of a given pattern. Note that you can also take the mod with a prime number of the resultant value to bring the so-obtained hash values within a limit.

The code is as follows:

### Code:

```
1. def findPattern(T, P):
2.     h2 = findHash1(P)
3.     for i in range(len(T)-1):
4.         str1 = T[i:i+len(P)]
5.         h1 = findHash1(str1)
6.         if h1 == h2:
```

```

7.   print('Hit at ', i)
8. T = 'aabbaababbababbaab'
9. P = 'abab'
10. findPattern(T, P)
11. Hit at 6
12. Hit at 11

```

It can be seen that the performance of this algorithm depends on the chosen hash function. This drawback calls for a better algorithm that takes time and does not depend on the choice of the hash function. The following section discusses one such algorithm.

## Knuth–Morris–Pratt algorithm

**Problem statement:** Two strings P and T are given. The length of P is **m**, and the length of T is **n**. The goal is to find all the occurrences of P in T.

In the Brute Force algorithm, as soon as a mismatch occurs, we move back to the starting index **i** and continue matching the pattern. However, at this point, some part of the pattern, P, has already been matched, say

$$T[s, s + 1, \dots, s + k] = P[1, 2, \dots, k]$$

That is,

$$T[s + k + 1] \neq P[k + 1]$$

We need to shift our window to  $s'$  such that,

$$T[s', s' + 1, \dots, s' + l] = P[1, 2, \dots, l]$$

Such that,

$$s' + l = s + k$$

This method uses an array  $\Pi$  created by finding all the prefixes in P, which are also the suffixes. The Knuth–Morris–Pratt algorithm uses this array resulting in the best-case time complexity of  $O(n)$ .

To understand the creation of  $\Pi$ , let us consider an example where P is “abaababb”. To accomplish this task, let us initialize the array  $\Pi$  to

[0,0,0,0,0,0,0,0]. Note that the number of zeros in  $P_i$  is the same as the length of the pattern.

**Procedure:** Creation of  $P_i$

Initially, the value of  $l$  is 0 and  $i$  is 1.

If  $P[i] \neq P[l]$  then

we set  $P_i[i]$  as 0 and increment the value of  $i$  by 1.

Else If these two are equal then

We increment  $l$  by 1, set  $P_i[i] = l$  and then increment  $i$  by 1.

In case  $P[i]$  is not the same as  $P[l]$ , and  $l$  is not 0, then

we set  $l$  as  $P_i[l]$ .

Consider the dry run of the preceding algorithm for the computation of  $P_i$ .

**Step 1:**  $i = 1$ ,

$l = 0$

$P[i], P[l] = b\ a$

$P[i] \neq P[l]$  ( $= b \neq a$ ) and

$l$  is 0;

Set  $P_i[i] = 0$  and

Increment the value of  $i$

**Step 2:**  $i = 2, l = 0$

$P[i], P[l] = a$

$P[i] = P[l]$  ( $= a$ )

Increment  $l$ ,

set  $P_i[i] = l$  and then Increment  $i$

**Step 3:**  $i = 3$ ,

$l = 1$

$P[i], P[l] = a\ b$

$P[i] \neq P[l]$  ( $a \neq b$ ) and

$l$  is not 0;

Set  $l$  as  $P_i[l-1] = 0$

**Step 4:**  $i = 3$ ,

$l = 0$

$P[i], P[l] = a, a$

$P[i] = P[l] (= a)$

Increment  $l$ ,

Set  $P[i] = l$  and then Increment  $i$

**Step 5:**  $i = 4$ ,

$l = 1$

$P[i], P[l] = b, b$

$P[i] = P[l] (= b)$

Increment  $l$ ,

Set  $P[i] = l$  and then Increment  $i$

**Step 6:**  $i = 5$ ,

$l = 2$

$P[i], P[l] = a, a$

$P[i] = P[l] (= a)$

Increment  $l$ ,

Set  $P[i] = l$  and then Increment  $i$

**Step 7:**  $i = 6, l = 3$   $P[i], P[l] = b, a$

$P[i] \neq P[l] = b \neq a$  and  $l$  is not 0; Set  $l$  as  $P[i-1]$  1

**Step 8:**  $i = 6$ ,

$l = 1$

$P[i], P[l] = b, b$

$P[i] = P[l] (= b)$

Increment  $l$ ,

set  $P[i] = l$  and then Increment  $i$

**Step 9:**  $i = 7$ ,

$l = 2$

$P[i], P[l] = b, a$

$P[i] \neq P[l]$  ( $b \neq a$ ) and  
 $l$  is not 0;

Set  $l$  as  $Pi[l-1]$  0

**Step 10:**  $i = 7$ ,  
 $l = 0$

$P[i], P[l] = b, a$

$P[i] \neq P[l]$  ( $b \neq a$ ) and  
 $l$  is 0;

Set  $Pi[i] = 0$  and increment the value of  $i$

The preceding steps generate the array **Pi** for the pattern ‘abaababb’, which is [0, 0, 1, 1, 2, 3, 2, 0].

Note that the first element of this array is 0, as no suffix of ‘a’ is the same as the prefix of ‘a’. Likewise, the second element is also 0, as no suffix of ‘ab’ is the same as the prefix of ‘ab’. The third element is 1, as in the string ‘aba’ ‘a’ is the suffix as well as the prefix of this string. In the next step, the value remains one, after which it becomes 2, as in ‘abaab’ ‘ab’ is the suffix as well as the prefix of the string. The process is shown in [table 15.1](#):

Step Number	String	Pi
1.	‘a’	[0]
2.	‘ab’	[0, 0]
3.	‘aba’	[0, 0, 1]
4.	‘abaa’	[0, 0, 1, 1]
5.	‘abaab’	[0, 0, 1, 1, 2]
6.	‘abaaba’	[0, 0, 1, 1, 2, 3]

**Table 15.1:** Creation of Pi array for the pattern

The following code implements the preceding method as follows:

### Code:

1. `def computePi(P, Pi):`
2. `l = 0 # length of the matched so far`

```

3. m = len(P)
4. Pi[0] = 0#Pi[0]
5. i = 1
6. # calculate Pi[0]
7. while i < m:
8.     print('i = ', i, ', l = ', l, ' P[i], P[l] =', P[i], P[l])
9.     if P[i]== P[l]:
10.         print('\t P[i] = P[l] =', P[i], 'Increment l, set
11.             Pi[i] = l and then Increment i')
12.         l += 1
13.         Pi[i] = l
14.         i += 1
15.     else:
16.         if l != 0:
17.             print('\t P[i] != P[l] =', P[i], ' != ', P[l], ' and
18.                 l is not 0; Set l as Pi[l-1]', Pi[l-1])
19.             l = Pi[l-1]
20.         else:
21.             print('\t P[i] != P[l] =', P[i], ' != ', P[l], ' and
22.                 l is 0; Set Pi[i] =0 and Increment the value of i')
23.             Pi[i] = 0
24.             i += 1
25. P='abaababb'
26. Pi= [0]*len(P)
27. m = len(P)
28. computePi(P, Pi)
29. print(Pi)

```

Initially, the value of l is 0, and i is 1. If  $P[i] \neq P[l]$ , then we set  $Pi[i]$  as 0 and increment the value of i by 1. If these two are equal, then we increment l by 1, set  $Pi[i] = l$ , and then increment i by 1. In case  $P[i]$  is not the same as  $P[l]$ , and l is not 0, then we set l as  $Pi[l]$ .

## **KMP method**

The KMP method uses the Pi array created previously to find the occurrences of the Pattern, P, in the Text, T.

### **Algorithm: KMP**

Set the initial value of  $i = 0$ . Here, i is the initial index of T. Also, set the initial value of j as 0.

Repeat the following steps while  $(n - i) \geq (m - j)$

if  $P[j] == T[i]$  then

Increment i by 1

Increment j by 1

if  $j == m$ :

The pattern is found at»,  $(i-j), \leftarrow$  and set j as  $Pi[j-1], \leftarrow, Pi[j-1]$ )

Set  $j = Pi[j-1]$

elif  $i < n$  and  $P[j] \neq T[i]$ :

if  $j \neq 0$ :

$j = Pi[j-1]$

else:

$i += 1$

To understand the preceding algorithm, let us consider an example. The text, T, is  $T = \text{'babbababbababbabbab'}$  and the pattern to be searched in the text is  $P = \text{'abba'}$ .

The steps for the KMP Method are as follows:

Step 1:  $i = 0$   $j = 0$   $P[j] = a$   $T[i] = b$

In the case of mismatch if j is not 0, set j as  $Pi[j-1] - 1$  else increment i

Step 2:  $i = 1$   $j = 0$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment i and j

Step 3:  $i = 2$   $j = 1$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 4:  $i = 3$   $j = 2$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 5:  $i = 4$   $j = 3$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment  $i$  and  $j$

Since  $j = m$ , print Found pattern at 1 and set  $j$  as  $P[i[j-1]] - 1$

Step 6:  $i = 5$   $j = 1$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

In the case of mismatch if  $j$  is not 0, set  $j$  as  $P[i[j-1]] - 0$  else increment  $i$

Step 7:  $i = 6$   $j = 0$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 8:  $i = 7$   $j = 1$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 9:  $i = 8$   $j = 2$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 10:  $i = 9$   $j = 3$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment  $i$  and  $j$

Since  $j = m$ , print Found pattern at 6 and set  $j$  as  $P[i[j-1]] - 1$

Step 11:  $i = 10$   $j = 1$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

In the case of mismatch if  $j$  is not 0, set  $j$  as  $P[i[j-1]] - 0$  else increment  $i$

Step 12:  $i = 11$   $j = 0$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 13:  $i = 12$   $j = 1$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 14:  $i = 13$   $j = 2$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment  $i$  and  $j$

Step 15:  $i = 14$   $j = 3$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment i and j

Since  $j = m$ , print Found pattern at 11 and set j as  $Pi[j-1] - 1$

Step 16:  $i = 15$   $j = 1$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment i and j

Step 17:  $i = 16$   $j = 2$   $P[j] = b$   $T[i] = b$

$P[j] = T[i]$ , increment i and j

Step 18:  $i = 17$   $j = 3$   $P[j] = a$   $T[i] = a$

$P[j] = T[i]$ , increment i and j

Since  $j = m$ , print Found pattern at 14 and set j as  $Pi[j-1] - 1$

The code for the preceding method is as follows.

### Code:

```
1. def KMP(T, P):
2.     m = len(P)
3.     n = len(T)
4.     Pi = [0] * m
5.     j = 0 # initial index for j
6.
7.     computePi(P, Pi)
8.     print(Pi)
9.     i = 0 # initial index of T
10.    while (n - i) >= (m - j):
11.        print('i = ', i, ' j = ', j, ' P[j] = ', P[j], ' T[i] = ', T[i])
12.        if P[j] == T[i]:
13.            print(' P[j] = T[i], increment i and j')
14.            i += 1
15.            j += 1
16.
```

```

17.     if j == m:
18.         print("Since j = m, print Found pattern at", (i-j), ' '
19.         and set j as Pi[j-1] ', Pi[j-1])
20.
21.     # mismatch
22.     elif i < n and P[j] != T[i]:
23.         print(' In the case of mismatch if j is not 0, set j
24.             as Pi[j-1] ', Pi[j-1], ' else increment i')
25.         if j != 0:
26.             j = Pi[j-1]
27.         else:
28.             i += 1
29. T='babbababbababbabbab'
30. P='abba'
31. Pi= [0]*len(P)
32. m = len(P)

```

## Conclusion

This chapter discussed various string-matching algorithms, their implementations, and their drawbacks. The reader is expected to explore other algorithms given in the references of this chapter. The reader will be able to use these in applications related to bioinformatics and network security. The exercises given in the Appendix will focus on some advanced problems related to this topic.

## Multiple choice questions

1. Which of the following takes time, where  $n$  is the length of the text and  $m$  is the length of the pattern?

- a. Brute Force
  - b. Rabin Karp
  - c. KMP
  - d. None of the above
2. **Which of the following takes time in the average case, where  $n$  is the length of the text and  $m$  is the length of the pattern?**
- a. Brute Force
  - b. Rabin Karp
  - c. KMP
  - d. None of the above
3. **Which of the following makes use of Hashing?**
- a. Brute Force
  - b. Rabin Karp
  - c. KMP
  - d. None of the above
4. **When a mismatch occurs in Brute Force, some of the characters have already been matched. Which of the following makes use of this information?**
- a. Brute Force
  - b. Rabin Karp
  - c. KMP
  - d. None of the above
5. **In the Rabin Karp algorithm, can the sum of the ASCII values be considered a reliable hash function?**
- a. Yes
  - b. No
6. **Which of the following depends on the properties of the pattern?**
- a. Brute Force

- b. Rabin Karp
- c. KMP
- d. None of the above

**7. For ‘abaaba’, the value of Pi used in KMP is**

- a. [0, 0, 1, 1, 2, 3]
- b. [0, 0, 0, 1, 1, 1]
- c. [0, 1, 2, 3, 4, 5]
- d. None of the above

**8. Can we modify brute force to find the pattern in time for a pattern in which all the characters are different?**

- a. Yes
- b. No

**9. Which of the following can be used to implement KMP?**

- a. DFA
- b. Push Down Automata
- c. Both
- d. None of the above

**10. Which of the following can be used for checking spelling?**

- a. Tries
- b. Suffix Trees
- c. Both
- d. None of the above

## Theory/applications

1. State the algorithm for string matching using Brute Force. What is the complexity of this algorithm?
2. Can we improve the preceding algorithm for a pattern in which all the characters are different?

3. Find the number of comparisons for  $P = \text{'abbab'}$  and  $T = \text{'ababbbbababba'}$  in the Brute Force Algorithm.
4. In the preceding question, trace the steps of the KMP algorithm.
5. What is the intuition behind the KMP algorithm? State its complexity.
6. In Question 3, if we apply Rabin Karp using the sum of ASCII values as the hash function, what will be the problem?
7. For the preceding question, state a better hash function.
8. Explain the concept of rolling hash. Does it always work?
9. What is the complexity of Rabin Karp? Explain.
10. Suggest an algorithm to find the occurrences of a given pattern,  $P$ , in text  $T$ , given that the pattern  $P$  has some special characters which match with any (and any number of characters).

## Find errors/special cases

In the questions that follow, find the error or the special conditions in which they will produce correct results. Also, if you think that they will accomplish the given task, craft various test cases and observe the output.

### 1. Matching strings

```

1. def matchBrute(T, P):
2.     n = len(T)
3.     m = len(P)
4.     for i in range(n-m):
5.         j=0
6.         while(j<m):
7.             if(T[i+j] != P[j]):
8.                 break
9.             j+=1
10.            if(j==m):
11.                print('Match at ', i)
12. T='babbababbababbab'

```

```
13. P='abba'  
14. matchBrute(T, P)  
15. #Output  
16. Match at 1  
17. Match at 6  
18. Match at 11  
19. Match at 14
```

2. Finding P in T if all the characters of P are different.

```
1. def findPattern(P, T):  
2.     i=0  
3.     j=0  
4.     m=len(P)  
5.     while(i<len(T)):  
6.         index= T.index(P[0])  
7.         #print('Index at ', index)  
8.         if(T[index:index+m]==P):  
9.             print('Found at ', index)  
10.            T=T[index+1:]  
11.        else:  
12.            print('Not Found')  
13.            break  
14.        i+=1  
15.    #Output  
16.    P = 'defg'  
17.    T = 'abcdefghijklm'  
18.    findPattern(P, T)
```

3. String matching using hashing

```
1. def findHash(P):
```

```

2. h = 0
3. for i in P:
4.     h+= ord(i)
5. return h
6. def findHash1(P):
7.     h = 0
8.     j=0
9.     for i in P:
10.        h+= ord(i) * (2**j)
11.        j+=1
12.    return h
13. def findPattern(T, P):
14.     h2 = findHash1(P)
15.     for i in range(len(T)-1):
16.         str1 = T[i:i+len(P)]
17.         h1 = findHash1(str1)
18.         if h1 == h2:
19.             print('Hit at ', i)

```

## References

1. The Hong Kong University of Science and Technology:  
<https://home.cse.ust.hk/~dekai/271/notes/L16/L16.pdf>.
2. University of Illinois:  
<https://jeffe.cs.illinois.edu/teaching/algorithms/notes/07-strings.pdf>.
3. UNIVERSITY OF HAWAI'I AT MĀNOA:  
<http://courses.ics.hawaii.edu/ReviewICS311/morea/230.string-matching/reading-notes.html>.
4. University of Auckland:

<https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf>.

5. Purdue University:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap11.pdf>.

6. Carnegie Mellon University:

<http://www.cs.cmu.edu/afs/cs/academic/class/15451-s14/www/LectureNotes/lecture39.pdf>.

## **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>



## APPENDIX 1

### All Pairs Shortest Path

#### Introduction

Single Source Shortest Path finds the shortest distances from a given point. This path can be found using algorithms like Dijkstra's, Bellman Ford, and so on. If we need to find the shortest distance between each pair, you can apply algorithms like Dijkstra's, taking each node as the source. The complexity of this brute force approach can be found by multiplying 'n' with the complexity of Single Source Shortest Path algorithm being used, where 'n' is the number of vertices. This appendix gives a brief overview to handle this problem in a slightly efficient way.

#### All Pairs Shortest Path

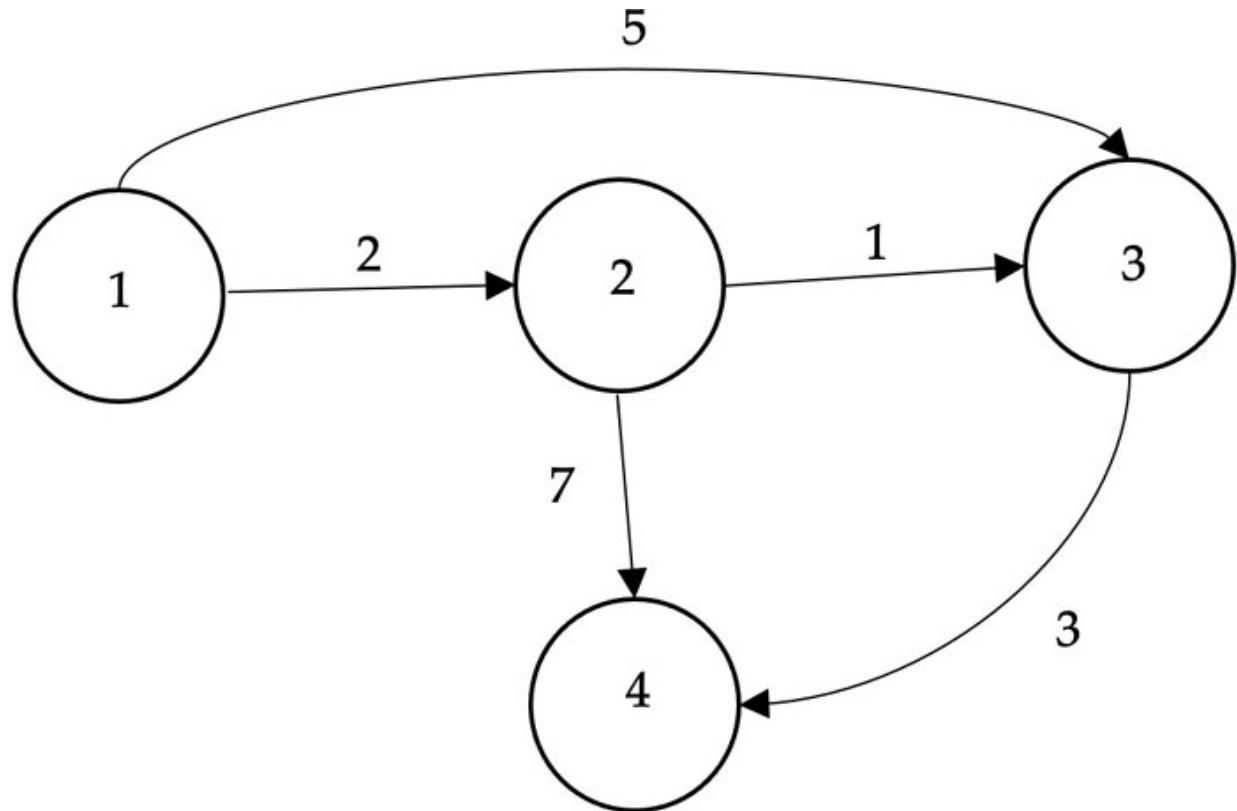
In the graph shown in [Figure Appendix 1.1](#), you need to find the shortest path between each pair of nodes. To accomplish this task you can either call Single Source Shortest Path many times or use the Floyd-Warshall algorithm, which uses dynamic programming. The code and the implementation of the following algorithm is given in the web resources of this book. The idea behind the algorithm is given in the following snippet:

```
def floyd_marshall(graph):
    n = len(graph)
    for k in range(n):
        for i in range(n):
            for j in range(n):
                graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j])
    return graph
```

To understand this, let us consider a graph with 4 nodes (1, 2, 3, and 4) and the following edges:

- 1 ->2 with weight 2,

- 1 ->3 with weight 5,
- 2 ->3 with weight 1,
- 2 ->4 with weight 7, and
- 3 ->4 with weight 3 ([Figure Appendix 1.1](#)).



*Figure Appendix 1.1: Graph*

The algorithm uses a matrix to keep track of the shortest path between all the pairs of nodes. The steps to find the All-Pairs shortest path are as follows:

**Step 1:** Initialize the matrix with the weights of the edges. Let the matrix be called "dist". The matrix has the same number of rows and columns as the number of nodes in the graph. Each cell of the final matrix represents the shortest distance between two nodes. For example,  $\text{dist}[i][j]$  represents the shortest distance between node i and node j. Initially,

$$\text{dist} = \begin{matrix} & 0 & 2 & 5 & \infty \\ 0 & \infty & 0 & 1 & 7 \\ \infty & \infty & 0 & 3 & \\ \infty & \infty & \infty & 0 & \end{matrix}$$

Note that, if there is no direct edge between the two nodes, then the weight of the corresponding edge is infinity ( $\infty$ ).

**Step 2:** For each node  $k$  in the graph, we need to find if there exists a shorter path between  $i$  and  $j$  through the node  $k$ , in which case the value of  $\text{dist}[i][j]$  is updated with the value.

For example, we check if the direct distance between 1 and 3 is shorter or via 2 is shorter. That is, since  $2 + 1 = 3 < 5$  (the current distance between 1 and 3), we update  $\text{dist}[1][3]$  as 3. Likewise, we find the distance between 1 and 3 via various nodes and update the value with the least one.

$$\text{dist} = \begin{matrix} & 0 & 2 & 3 & \infty \\ 0 & \infty & 0 & 1 & 7 \\ \infty & \infty & 0 & 3 & \\ \infty & \infty & \infty & 0 & \end{matrix}$$

We repeat this process for all nodes  $k$ .

**Step 3:** After all iterations, the matrix  $\text{dist}$  contains the shortest distance between all pairs of nodes. In our example, the final matrix is:

$$\text{dist} = \begin{matrix} & 0 & 2 & 3 & 6 \\ 0 & \infty & 0 & 1 & 4 \\ \infty & \infty & 0 & 3 & \\ \infty & \infty & \infty & 0 & \end{matrix}$$

This means that the shortest path:

- Between 1 and 2 is 2,
- Between 1 and 3 is 3,
- Between 1 and 4 is 6 (through 2 and 3),
- Between 2 and 3 is 1,
- Between 2 and 4 is 4 (through 3), and
- Between 3 and 4 is 3.

The reader is advised to implement this algorithm before referring to the web resources.

## APPENDIX 2

### Tree Traversals

#### Introduction

We have already seen some of the important applications of stacks like expression evaluation and recursion. Stacks are also used for traversing other data structures like trees and graphs. We have already seen graph traversal using stacks in [Chapter 11, Graphs](#). This appendix explains the use of stacks in traversal of binary trees.

#### In-Order Traversal

The in-order traversal of a Binary Tree has already been explained in [Chapter 8, Trees](#) of this book. The following discussion explains a non-recursive procedure of this algorithm using Stacks.

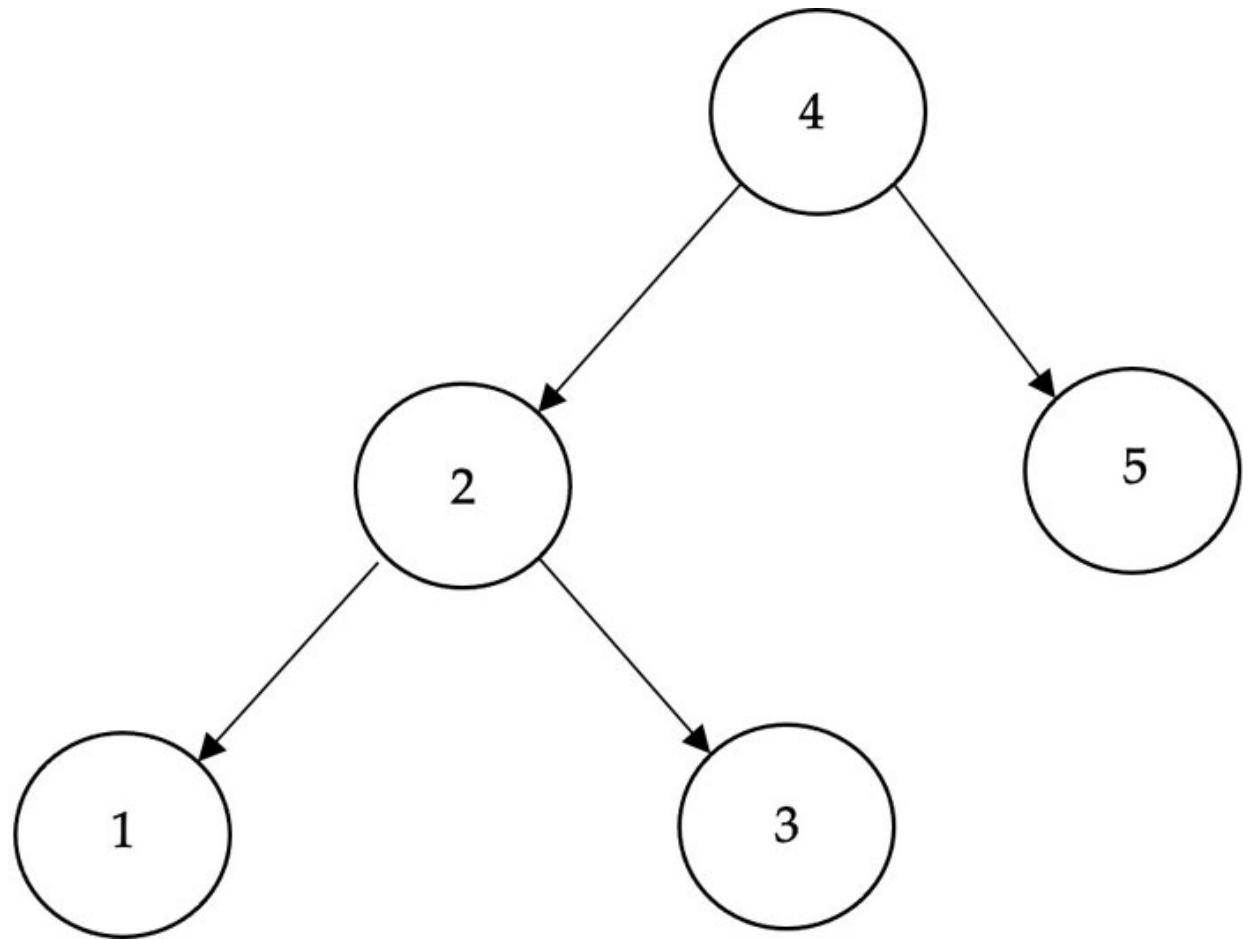
**Step 1:** We start with an empty stack and the current node is set to the root of the Binary Tree.

**Step 2:** While the current node is not null or the stack is not empty, perform the following steps:

1. If the current node is not null, push it to the stack and move to its left child.
2. If the current node is null, pop a node from the stack and print its value.
3. The current node is then set to its right child.

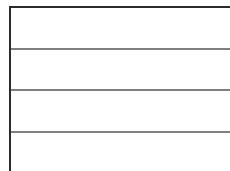
**Step 3:** Repeat step 2 until the stack is empty and the current node is null.

For example, consider the tree given in [Figure Appendix 2.1](#):

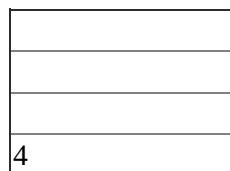


*Figure Appendix 2.1: In-Order Traversal Example*

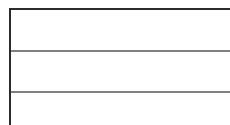
1. Start with an empty stack and current node set current to 4 (root of the Binary Tree).



2. Push 4 to the stack and move to its left child (2).



3. Push 2 to the stack and move to its left child (1)



2
4

4. Push 1 in the stack, 1 does not have any left child so print 1, set current node to its right child (null).

2
4

5. Pop 2 from the stack, print 2, set current node to its right child (3).

2
4

6. Push 3 in the stack, 3 does not have any left child so print 3, set current node to its right child (null).

7. Pop 4 from the stack, print 4, set current node to its right child (5).


8. Push 5 to the stack, print 5, set current node to its right child (null).

9. Stack is empty. The in-order traversal of the given tree is therefore: 1, 2, 3, 4, 5.

## Pre-order traversal

The pre-order traversal of a Binary Tree has already been explained in [Chapter 8, Trees](#) of this book. The following discussion explains a non-recursive procedure of this algorithm using Stacks.

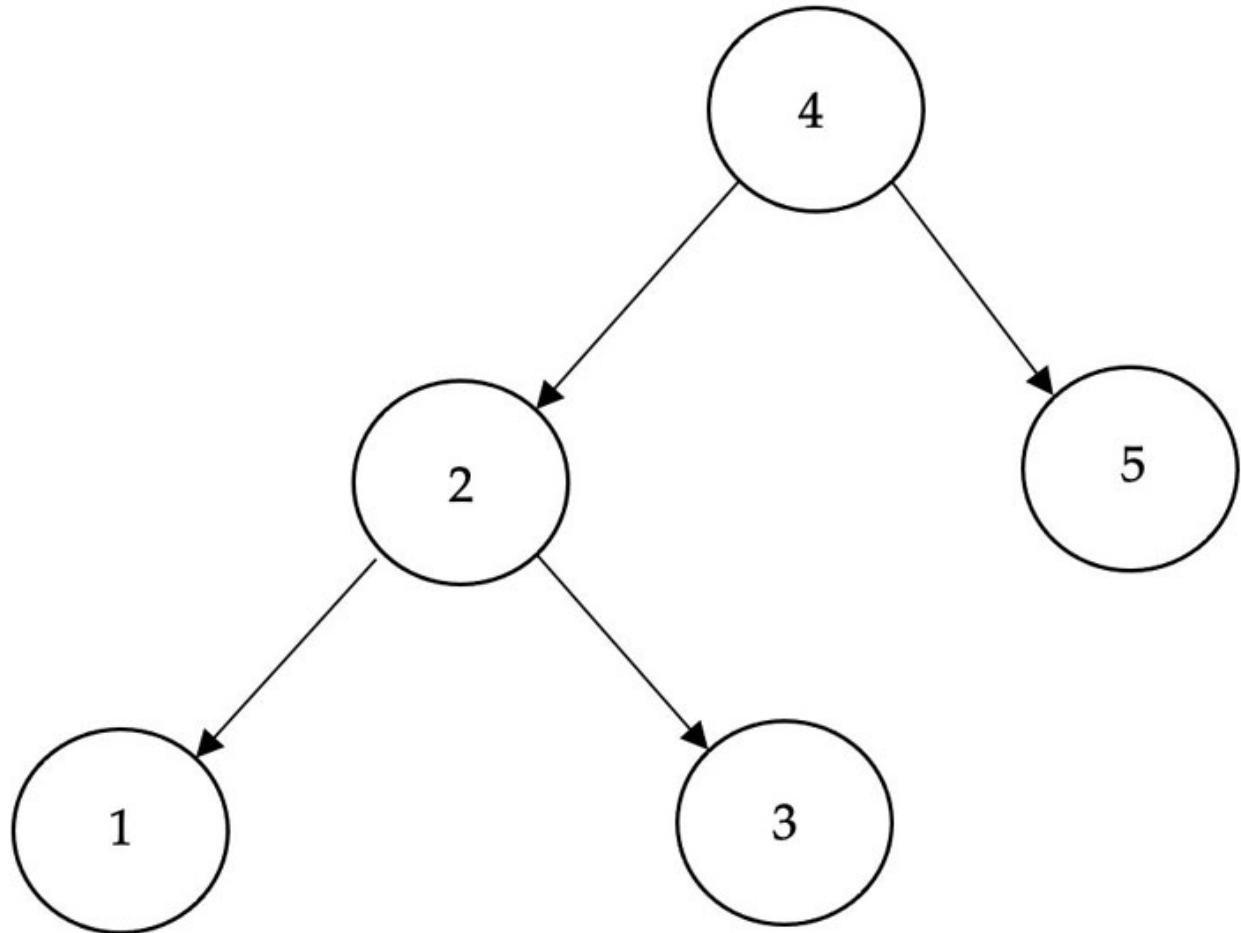
**Step 1:** Create an empty stack and push the root node onto it.

**Step 2:** While the stack is not empty, perform the following steps:

- Pop the top node from the stack and print its value.
- Push the right child of the popped node onto the stack.
- Push the left child of the popped node onto the stack.

**Step 3:** Repeat step 2 until all nodes have been visited and printed.

For Example, Consider the tree given in [Figure Appendix 2.2](#):

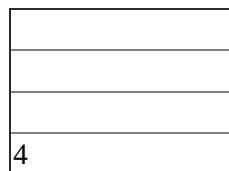


*Figure Appendix 2.2: Example of Pre-order*

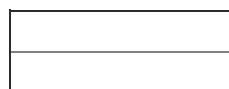
The pre-order traversal using stack would result in:

**Steps:**

1. The root node (4) is pushed onto the stack.



2. The top node (4) is popped and printed. The right child (5) and left child (2) are pushed onto the stack in that order.



2
5

3. The top node (2) is popped and printed. The right child (3) and left child (1) are pushed onto the stack in that order.

1
3
5

4. The top node (1) is popped and printed.

3
5

5. The top node (3) is popped and printed.

5

6. The top node (5) is popped and printed.

5

7. The stack is now empty. The pre-order traversal of the given tree is therefore: 4, 2, 1, 3, 5.

## Post-order traversal

The post-order traversal of a Binary Tree has already been explained in [Chapter 8, Trees](#) of this book. The following discussion explains a non-recursive procedure of this algorithm using Stacks.

**Step 1:** Create two stacks, S1 and S2.

**Step 2:** Push the root node of the tree into S1.

**Step 3:** While stack1 is not empty, repeat the following steps:

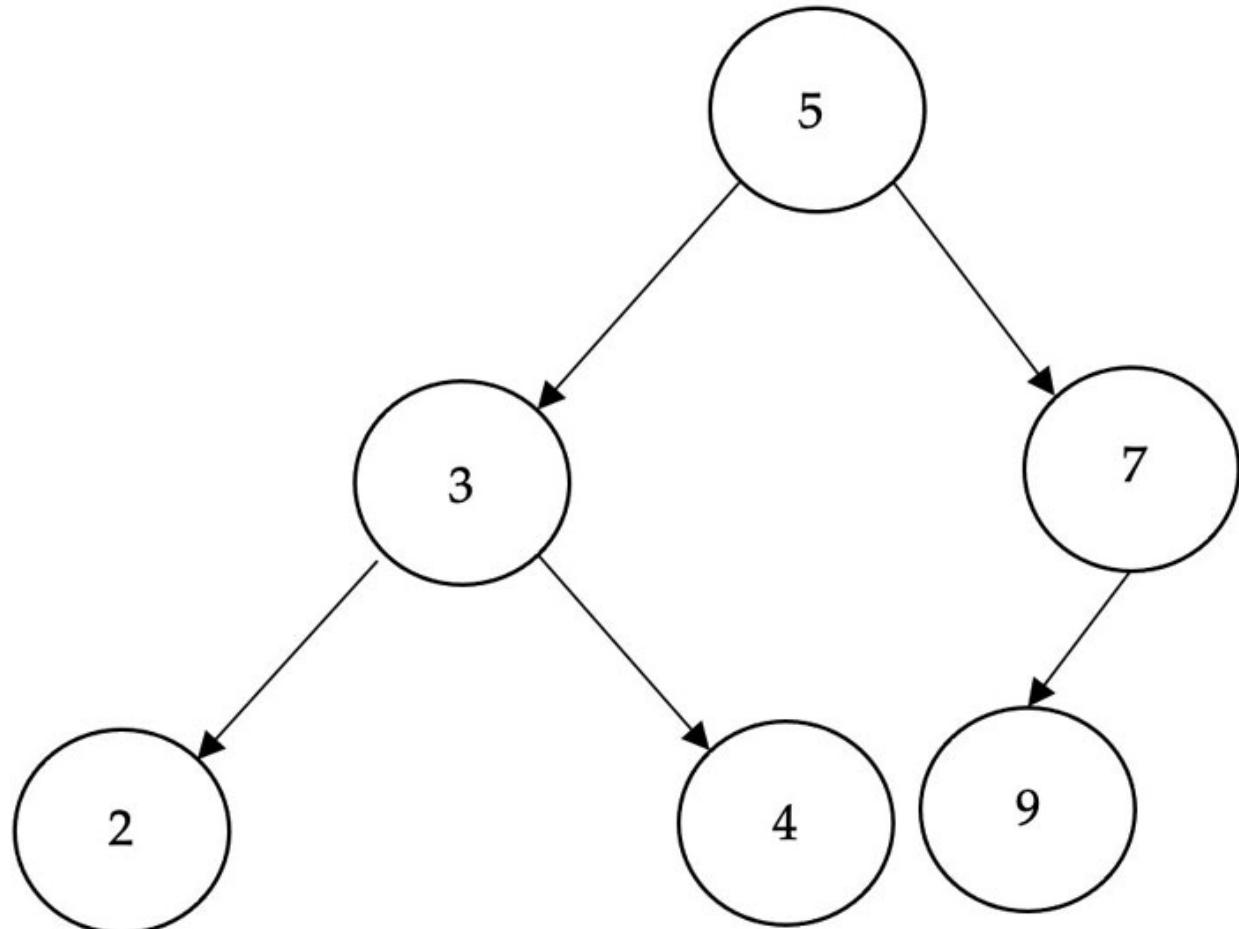
- a. Pop the top node from stack1 and push it into S2.

- b. Push the left child of the popped node into S1, if it exists.
- c. Push the right child of the popped node into S, if it exists.

**Step 4:** While S2 is not empty, repeat the following steps:

- a. Pop the top node from S2 and print its value.

For example, Consider the tree given in [Figure Appendix 2.3](#):



*Figure Appendix 2.3: Example post-order*

The application of the above algorithm results in the following configurations of the stacks:

1. Initially:

S1: [5]

Stack2: []

2. Pop 5 from stack1 and push it into stack2

Stack1: []

Stack2: [5]

3. Push 3 (left child of 5) and 7 (right child of 5) into stack1

Stack1: [3, 7]

Stack2: [5]

4. Pop 3 from stack1 and push it into stack2

Stack1: [7]

Stack2: [5, 3]

5. Push 2 (left child of 3) and 4 (right child of 3) into stack1

Stack1: [7, 2, 4]

Stack2: [5, 3]

6. Pop 7 from stack1 and push it into stack2

Stack1: [2, 4]

Stack2: [5, 3, 7]

7. Push 9 (right child of 7) into stack1

Stack1: [2, 4, 9]

Stack2: [5, 3, 7]

8. Pop 2, 4, and 9 from stack1 and push them into stack2

Stack1: []

Stack2: [5, 3, 7, 9, 4, 2]

9. Pop 2, 4, 9, 7, 3, and 5 from stack2 and print their values

Output: 2 4 9 7 3 5

## APPENDIX 3

### Dijkstra's Shortest Path Algorithm

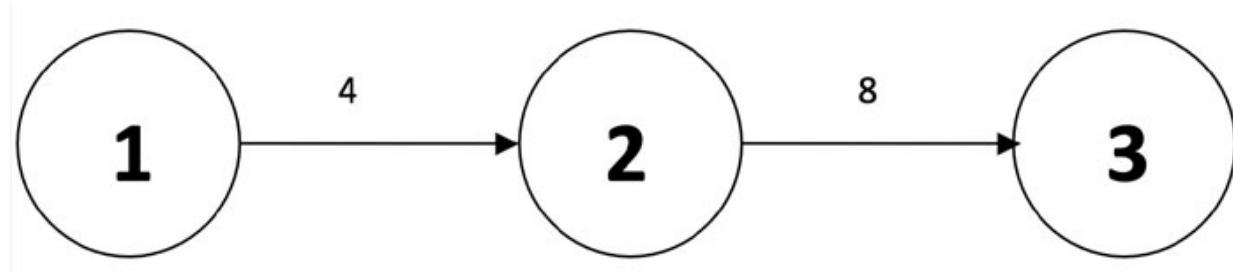
#### Introduction

[Chapter 11, Graphs](#) explains procedure of finding the shortest path using Prim's and Kruskal's algorithm. These two were greedy approaches as at each step the best possible solution at that particular point was chosen. The greedy approach might not work in every situation. Consider, for example, a graph containing four vertices {A, B, C, D} and edges {(A, B), (A, C), (B, D), (C, D)}. We need to find the shortest path from A to D. In going from vertex, A to vertex D of a given graph, the greedy algorithms may not produce the most optimal result. In the given graph, the cost of A to B is 1 and A to C is 3. The greedy approach will select vertex B, but the cost from B to D is 10, and C to D is 5. In this case, the greedy approach might come up with an incorrect answer. Other approaches work well in such cases. This chapter discusses Dijkstra's algorithm which is not greedy.

#### Dijkstra's shortest path algorithm

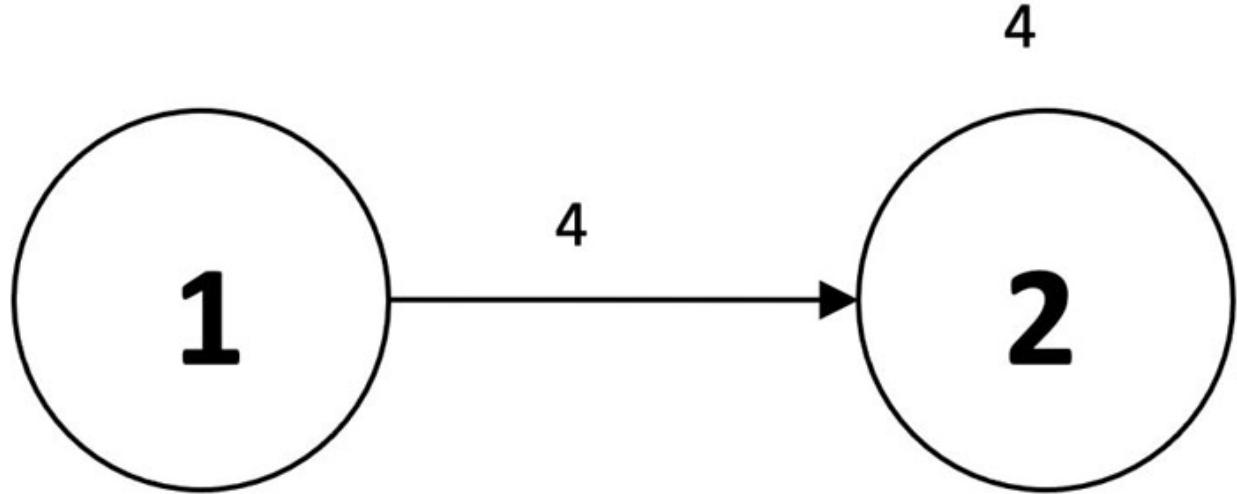
Dijkstra's algorithm finds the shortest path from a vertex to all other vertices. Consider the following example to understand the working of Dijkstra's Algorithm.

In the graph shown in [Figure Appendix 3.1](#), vertex 1 is the starting vertex. We aim to find the shortest path from vertex 1 to vertex 2 and vertex 3. The weights of the respective edges are mentioned in the following graph:



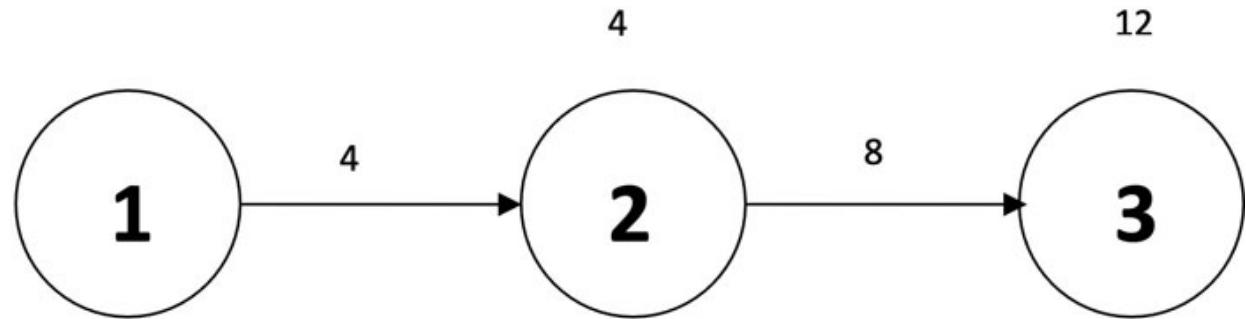
*Figure Appendix 3.1: Graph*

Let us begin from vertex 1. In moving from 1 to 2, the only option that we have, currently, is to move from 1 to 2 via the edge 1-2 as shown in [Figure Appendix 3.2](#):



*Figure Appendix 3.2: Shortest distance between vertex 1 and 2*

As per the Dijkstra algorithm once one edge has been selected (and hence the vertex), the rest of the edges can be selected via the selected vertex. Now, since the cost of moving from 2 to 3 is 8, and that from moving from 1 to 2 is 4, the cost of moving from 1 to 3 will be 12. Note that there is no direct edge between 1 and 3, hence the cost of the direct edge between 1 and 3 is infinity. Out of the two 12 is less as shown in [Figure Appendix 3.3](#):



*Figure Appendix 3.3: The labels above vertices 2 and 3 depict the distance of moving from 1 to 2 and 3*

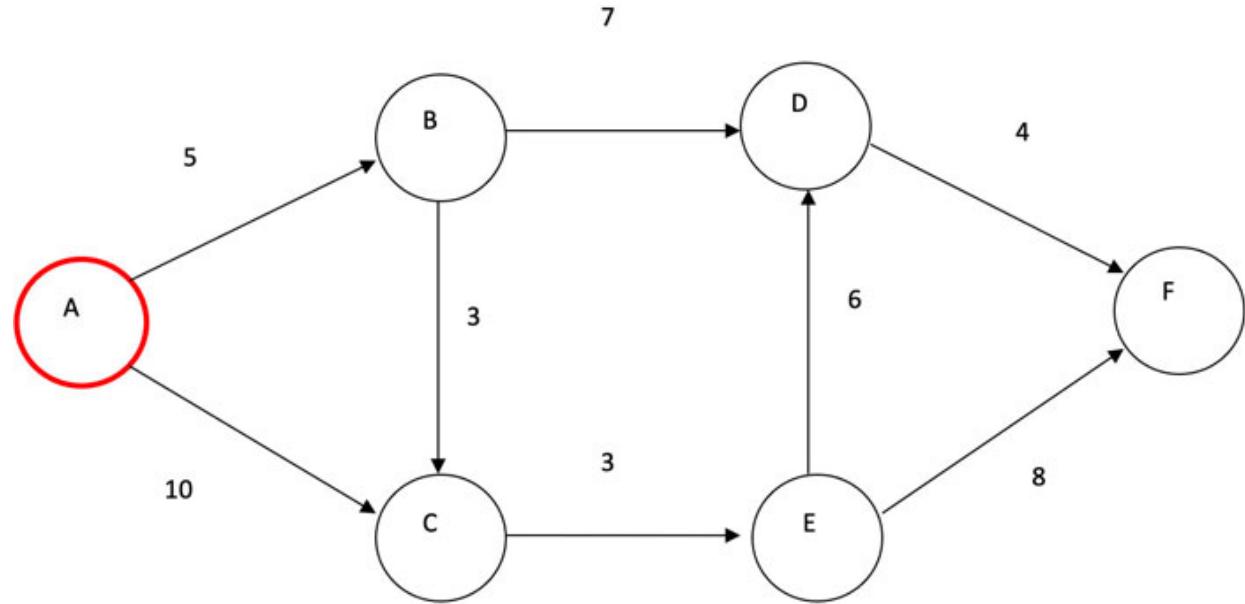
Updating (infinity) to 12 is known as Relaxation. The above point can be summarized as follows:

For two vertices and

If,  $(\text{distance}[u] + \text{cost}[u,v] < \text{distance}[v])$ :

$$\text{Distance}[v] = \text{distance}[u] + \text{cost}[u,v]$$

Let us apply this concept to find the shortest path from 1 to all other vertices, in the graph shown in [Figure Appendix 3.4](#):

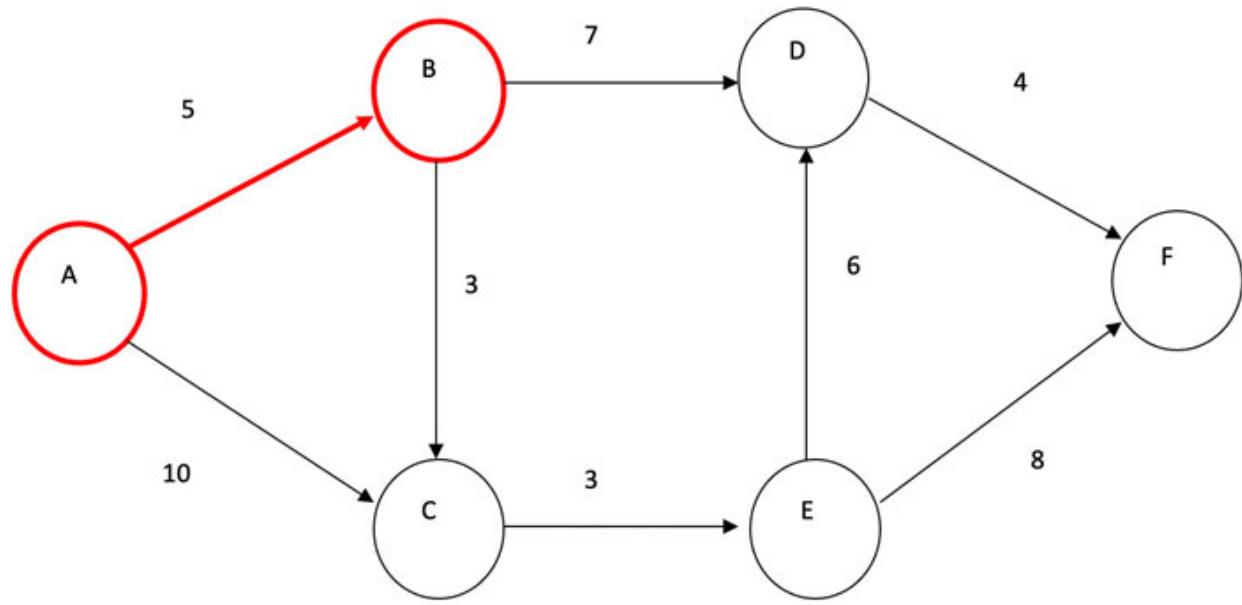


[Figure Appendix 3.4: Source node A](#)

Note that the source vertex is A, and the cost of shortest path from A to all other vertices are:

Selected path	B	C	D	E	F
	5	10	$\infty$	$\infty$	$\infty$

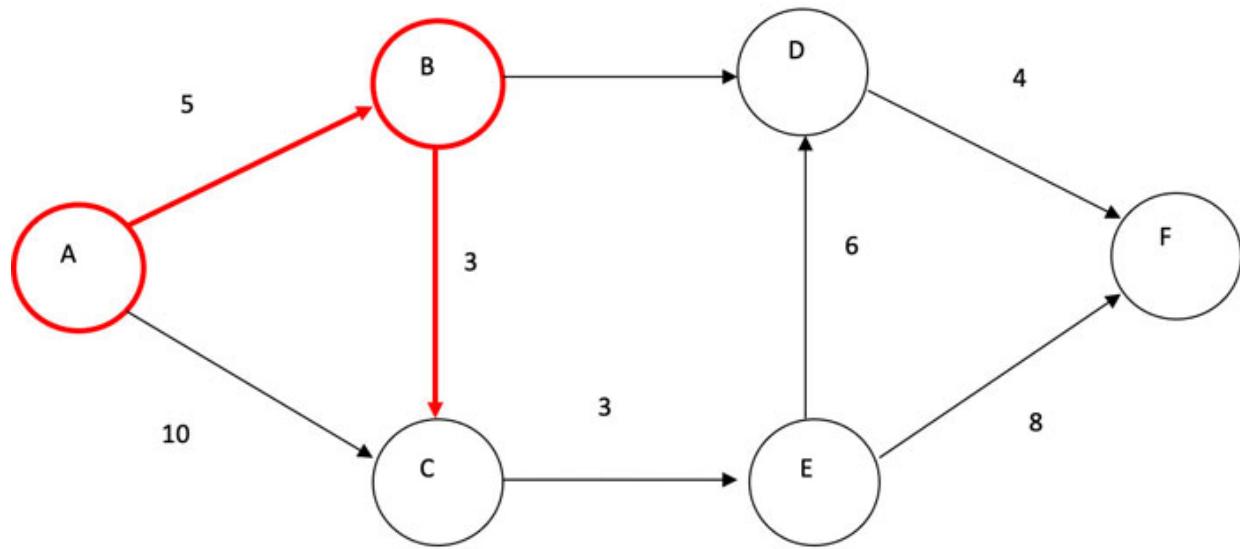
From amongst the above, the path from A to B is the shortest, having cost = 5 as shown in [Figure Appendix 3.5](#):



*Figure Appendix 3.5: Moving from A to B*

Selected path	B	C	D	E	F
	5	10	$\infty$	$\infty$	$\infty$

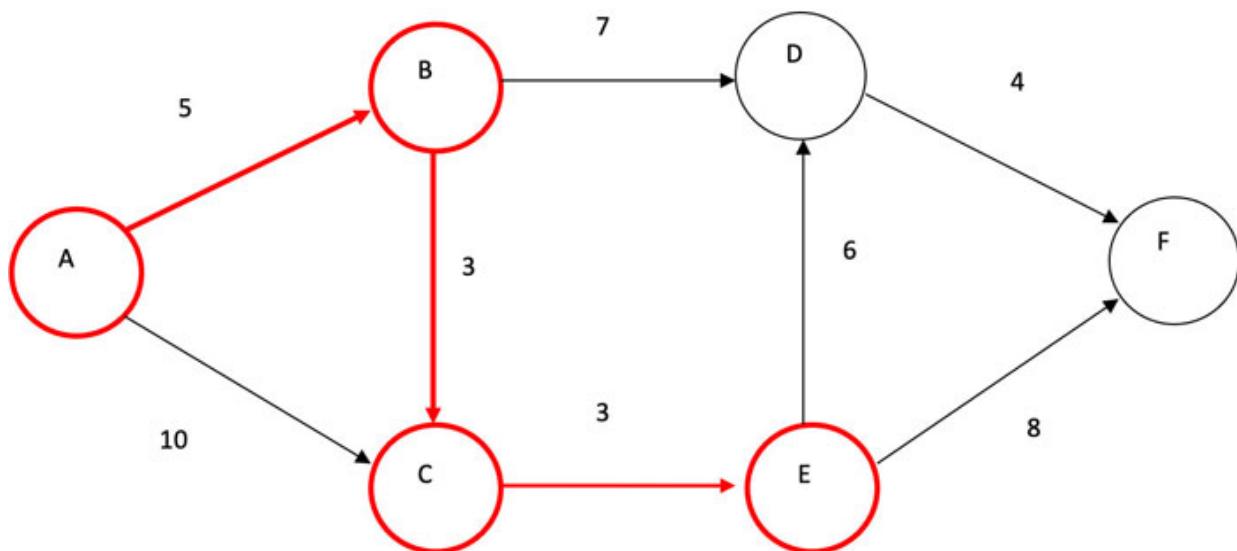
Note that since B has been selected, we can move to C via B also, the cost of which is 8, which is less than the cost of the direct edge from A to C as shown in [Figure Appendix 3.6](#):



*Figure Appendix 3.6: Moving from A to B and then to C*

Selected path	B	C	D	E	F
B	5	10	$\infty$	$\infty$	$\infty$
C	5	8	$\infty$	$\infty$	$\infty$

From C, we can now move to E, and the total cost of moving to E, via C is 11 as shown in [Figure Appendix 3.7](#):



*Figure Appendix 3.7: Moving from A to B and then to C, then E*

The updated table would be:

<b>Selected path</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
B	5	10	$\infty$	$\infty$	$\infty$
C	5	8	$\infty$	$\infty$	$\infty$
E	5	8	$\infty$	11	$\infty$

The reader is expected to apply the algorithm to select the last two vertices.

## APPENDIX 4

### Supplementary Questions

#### Arrays: Level 0

1. Write an algorithm for Selection sort that requires  $O(n)$  swaps.
2. At times, the given array is sorted in the first few iterations in Bubble Sort. In such cases, the rest of the iterations are redundant. Write the algorithms for Bubble Sort in which such redundant comparisons are not carried out.
3. You are provided with a sorted array. You must insert an element in the sorted array at an appropriate position. Which algorithm would you use (or a part of it)?
4. An un-sorted array and an element in that array are provided to you. Rearrange the array in a way that all the numbers less than that element are to the left of that element, and all the numbers greater than the element are to the right.
5. It is required to find the minimum element in a given array, remove that element, and then find the minimum element from the remaining. The process is to be repeated  $k$  times, where. Which algorithm would you use to accomplish this task efficiently?

#### Arrays: Level 1

1. Find if a given array contains duplicate elements.
2. The above problem can be done by sorting the array. Can you suggest an algorithm that accomplishes this task in  $O(n)$  time?
3. Find the frequency of each element in the given array.
4. An array and a sum ( $s$ ) are given to you. Find the subset of the array having sum =  $s$ .
5. An array and a sum( $s$ ) are given to you. Find the smallest subset (having minimum length) of the array having sum =  $s$ .

#### Stacks

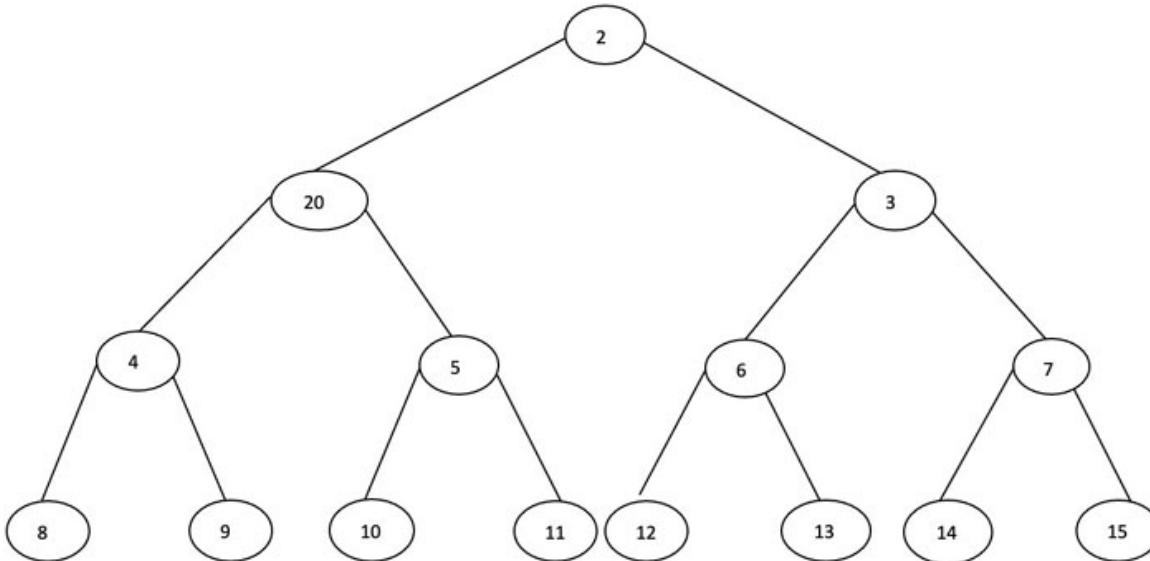
1. Implement a dynamic stack in which a single placeholder is added when overflow occurs.
2. Implement a dynamic stack in which the number of placeholders is doubled when overflow occurs.
3. Implement a dynamic stack in which the number of placeholders randomly increases when overflow occurs.
4. Using a stack, convert an infix expression into a postfix expression.
5. Using stacks, convert an infix expression into a prefix expression.
6. Using stacks, find the nth Fibonacci term.
7. Using stacks, find the number obtained by reversing the order of digits for a given number.
8. Using Queues, implement priority.
9. Using Queues, implement First Come, First Serve Scheduling.
10. Using Queues, implement First Come, First Serve with a time slice.

## Linked List

1. Write a program to find whether a given linked list has a cycle.
2. Write a program to join two linked lists.
3. Write a program to merge two linked lists.
4. Write a program to remove duplicate elements from a given linked list.
5. Write a program to find the second maximum element from a given linked list.
6. Write a program to find the element greater than the mean (assume that the linked list has only integers in the data part).
7. Write a program to find the common elements from two given linked lists.
8. Write a program to find the union of elements of two linked lists.
9. Write a program to arrange the linked list elements in descending order.
10. Write a program to partition a linked list as per the algorithm in the following reference.

## Trees

1. A Binary Search Tree is to be created from numbers 1, 2, 3...n in that order. The tree is stored using array representation. In [Chapter 8, Trees](#) it was stated that in such representation, the left child is stored at, whereas the right child is stored at.
2. For n = 500, find the sequence of the indices at which the numbers will be stored and the percentage of empty spaces in the so-formed array.
3. Can we generalize the above (Q1) for n? You may want to use reoccurrence relation of the sort.
4. You are given a binary tree to find a traversal that only prints the rightmost element at each level. For example, the traversal shown in [Figure Appendix 4.1](#) is 21, 35, 78, and 15:



**Figure Appendix 4.1:** Figure for questions 12 and 13

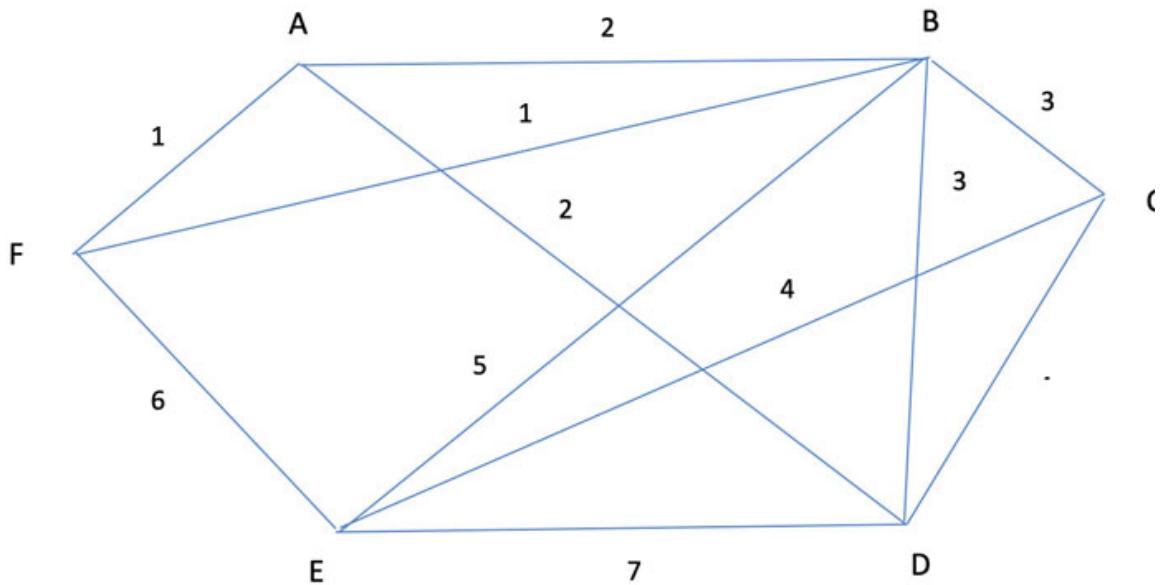
5. You are given a binary tree to find a traversal that only prints the leftmost element at each level. For example, the traversal shown in [Figure Appendix 4.2](#) is 21, 20, 40, and 8.
6. Two Binary Search Trees are given to you to create a new BST by merging those two trees.
7. Can the above question be done by creating two arrays consisting of the in-order traversal of those two trees and merging those two arrays?
8. Consider the following sequence 8, 10, 12, 14, 5, 2, 3, 9, 4, 6, 15, 1, 7, 11, 13, and find out the number of rotations required to create a balanced binary tree.

9. From the sequence given in *Question 8* create a B-tree of order 5.
10. You provided a list of numbers: 100, 95, 90.....5. Create a B-tree of order 5 and another B-tree of order 3. Compare the heights of the two trees.
11. For the sequence given in Question 10 Create an AVL tree and find the difference in the heights of the Binary Search Tree and the AVL tree created from the above sequence.
12. A tree can be represented using a two-dimensional array having n rows and two columns. In each row, the first column is i, and the second column is j, which means that there is an edge from i to j. Ask the user to enter the requisite data and display the tree (just the list of vertices and edges associated with them).
13. Create a binary tree using a doubly linked list. For this tree, accomplish the following tasks:
  - a. Write a program to implement the post-order traversal of a binary tree.
  - b. Write a program to implement the pre-order traversal of a binary tree.
  - c. Write a program to implement the in-order traversal of the tree.
  - d. Check if the given tree is a Binary Search Tree.
14. In a given Binary Search Tree, find the leftmost node of the right sub-tree of a given node.
15. In a given Binary Search Tree, find the rightmost node of the left sub-tree of a given node.
16. Write a program to create a heap from a given list.
17. Implement heap sort.

## **Graphs**

1. A graph can be represented using a two-dimensional array. The array would contain 0s and 1s. If the element at the  $i^{\text{th}}$  row and the  $j^{\text{th}}$  column has 1, it indicates the presence of an edge from vertex  $i$  to  $j$ . Ask the user to enter the number of vertices of a graph and create a two dimensional array depicting the graph.
2. Find the number of edges in the graph. (Note that the number of 1s in the 2-D array is not the same as the number of edges in the graph).
3. Find the vertex connected to the maximum number of edges.

4. Find if the graph has a cycle.
5. Ask the user to enter the initial vertex and the final vertex and find if there is a path from the initial to the final vertex.
6. In the above question (Question 5), find whether there are more than one paths from the initial to the final vertex, in which case, find the shortest path.
7. Now, in place of 1s, ask the user to enter a finite number representing the cost of the edge from vertex i to vertex j. Find the shortest path from the source vertex to all other vertices.
8. Write a program to find the spanning tree of the graph.
9. Write a program to find whether the graph is a tree.
10. For the graph given in [Figure Appendix 4.2](#), find the Minimum Cost Spanning tree:



*Figure Appendix 4.2: Figure for Questions 20, 21, 22 and 23*

11. For the graph given in [Figure Appendix 4.2](#), find the All-Pair Shortest Path.
12. For [Figure Appendix 4.2](#), will there be any difference in the Minimum Cost Spanning Tree formed by Kruskal and Prims Algorithm.
13. Apply Dijkstra's algorithm taking A as the source in Figure 2.
14. Instead of Minimum Cost Spanning Tree, if the Maximum Cost Spanning tree is to be found, will the greedy approach work?

## Application based

1. Generate an array of 500 random integers between 0 and 10000. Now apply the following sorting algorithms to sort the numbers.
  - a. Bubble Sort
  - b. Selection Sort
  - c. Insertion Sort
  - d. Quick Sort
  - e. Merge Sort
  - f. Heap Sort
  - g. Counting Sort
2. Run each algorithm five times and note the average execution time of each of the above.
3. Repeat the process with an array containing 1000 random integers, 2000 random integers, 5000 random integers, and 10000 random integers.
4. Plot the graph of the number of elements v/s execution time of each algorithm and compare the respective variations.
5. On the basis of the above, answer the following
  - a. Why do you think each algorithm was run five times and average execution time was noted?
  - b. If the number of elements is small, will there be an appreciable difference in the running times?
  - c. The variation of the execution time with the number of elements is predictable. For example, in the case of selection, bubble, and insertion, the sort variation should be parabolic. Why do you think then there is a difference in the graphs representing these variations for selection, bubble, and insertion sort?
6. Refer to Page 41 of <https://home.iitk.ac.in/~peeyush/102A/Lecture-notes.pdf>. Find the minor and cofactor of each element of a given square matrix.
7. Using the cofactors found in the previous question, find the determinant of the matrix.
8. Find the inverse of the given square matrix.

9. Read an image in Python and generate a Gray Level Co-occurrence matrix, as explained in [https://en.wikipedia.org/wiki/Co-occurrence\\_matrix](https://en.wikipedia.org/wiki/Co-occurrence_matrix). Now find Haralick features (Robert M Haralick; K Shanmugam; Its'hakDinstein (1973). "**Textural Features for Image Classification**" *IEEE Transactions on Systems, Man, and Cybernetics*. SMC-3 (6): 610–621. doi:10.1109/TSMC.1973.4309314).
10. Can you optimize the above process (*Question 2*)?
11. Apply Gray Level Co-occurrence matrix to 20 images of your face and 0 images that do not contain your face. Use the arrays so obtained to classify the two datasets.

# Index

## Symbols

2-D array  
indexing [59](#)  
memory map [59](#)

## A

all pairs shortest path  
finding [373-375](#)  
arrays [56, 57](#)  
elements, deleting [59-64](#)  
elements, inserting [59](#)  
linear search [68](#)  
operations [67, 68](#)  
problems [70-75](#)  
AVL trees [336, 337](#)  
definition [198, 199](#)  
deletion [200](#)  
node deletion [209-218](#)  
node insertion [199-208](#)

## B

backtracking [21](#)  
Balance Factor (BF) [200](#)  
binary numbers  
generating [44-47](#)  
Binary Search Tree (BST) [178, 179](#)  
leftmost node [188, 189](#)  
node deletion [185-188](#)  
node insertion [180-184](#)  
rightmost node [188](#)  
binary tree [167, 168](#)  
in-order traversal [171-174](#)  
post-order traversal [174, 175](#)  
pre-order traversal [176-178](#)

representation [169-171](#)  
traversal [171](#)  
Breadth First Search (BFS) [262-266](#)  
Brute force Algorithm [354](#)  
B tree [219](#)  
elements, inserting [220-225](#)  
example [219](#)  
node [219](#)  
Bubble sort [282](#), [286](#)

## C

chaining [338](#)  
circular queue [146-149](#)  
collision [338-340](#)  
collision resolution [340](#)  
linear probing [340-342](#)  
quadratic probing [342-344](#)  
separate chaining [344](#)  
solved problems [345-348](#)  
column-major format [58](#)  
Comb sort [286-288](#)  
Counting sort [295](#), [296](#)  
cyclic list [96](#)

## D

data [2](#)  
data item [2](#)  
data structures [3](#)  
arrays [4](#)  
deletion [5](#)  
graph [5](#)  
insertion [5](#)  
linked list [4](#)  
queue [5](#)  
searching [5](#)  
stacks [5](#)  
traversal [5](#)  
trees [5](#)  
two-dimensional arrays [4](#)

data types [3](#)  
int data type [3](#)  
primitive data types [3](#)  
datum [2](#)  
Depth First Search (DFS) [256-262](#)  
Dijkstra's shortest path algorithm [385-390](#)  
Directed Acyclic Graph(DAG) [266](#)  
divide and conquer approach [20, 21](#)  
doubly-ended queue (DEQueue) [149](#)  
algorithm [150, 152](#)  
for generating binary numbers [152-156](#)  
using [152](#)

## **Dynamic Programming (DP) [21-24](#)**

### **E**

exponentiation [36-38](#)  
expressions [117, 118](#)  
infix to postfix [120-126](#)  
infix to prefix [126-129](#)  
postfix expression [118-120](#)

### **F**

First in last out (FILO) [5](#)

## **folding method [339](#)**

### **G**

game of clones [6-10](#)  
revisited [10-12](#)  
graph [5, 166, 251](#)  
representation [252-256](#)  
spanning tree [270, 271](#)  
traversals [256](#)  
greedy approach [19, 20](#)

### **H**

hash function [338](#)

collision resolution [338](#)  
selecting [338](#)  
hashing [333](#), [337](#)  
hash tables [334](#)  
    information, storing [335](#)  
heap [232-234](#)  
    element, deleting [240-243](#)  
    element, inserting [234-239](#)  
    operations [234](#)  
    problems [246](#)  
heap sort  
    implementing [243-246](#)

## I

in-order traversal [171-174](#)  
non-recursive procedure, using stacks [377-379](#)  
Insertion sort [291](#), [292](#)  
int data type [3](#)

## K

KMP method [363-365](#)  
Knuth-Morris-Pratt (KMP) algorithm [354-363](#)  
Kruskal's algorithm [271-273](#)

## L

Last in first out (LIFO) [5](#), [37](#)  
leaves [169](#)  
linear probing [340-342](#)  
linear queue example [144](#)  
linked list [4](#), [79](#)  
    concatenation [100](#)  
    cycle, checking [101](#)  
    one-way linked list [80](#)  
    reversing [99](#), [100](#)  
    two-way linked list [90](#)  
linked list representation [336](#)  
lists  
    reversing [47](#), [48](#)  
longest common sub-sequence

finding [24-30](#)

## M

median

and order statistics [314, 315](#)

using heaps [318-322](#)

using insertion sort [323](#)

using partition [323, 324](#)

median of medians [315-318](#)

Kth smallest element [318](#)

memory map [58](#)

Merge algorithm [296-299](#)

Merge sort [300, 301](#)

Modular Arithmetic method [338](#)

## O

one-way linked list [80](#)

node, deleting [81, 85-87](#)

node, inserting [81-84](#)

traversing [80, 81](#)

open addressing [338](#)

## P

partition algorithm [301, 302](#)

postfix expression [118](#)

post-order traversal [174, 175](#)

non-recursive procedure, using stacks [381-383](#)

pre-order traversal [176-178](#)

non-recursive procedure, using stacks [379-381](#)

Primary Clustering [341, 342](#)

primitive data types [3](#)

priority queue [232](#)

heap structure [232-234](#)

## Q

quadratic probing [342-344](#)

queues [5, 98, 99, 139, 140](#)

algorithm [142-146](#)

scheduling [159](#)  
quick sort [303, 304](#)

## R

Rabbit problem [41-44](#)  
Rabin Karp algorithm [355-358](#)  
Radix sort [292-295](#)  
recursion [35](#)  
    numbers, playing with [48, 49](#)  
re-hashing [338](#)  
rolling hash [355](#)  
row-major format [58](#)

## S

Secondary Clustering [343](#)  
Selection sort [289-291](#)  
separate chaining [344](#)  
    example [344](#)  
siblings [169](#)  
sorted sequential array [335, 336](#)  
sorting  
    Bubble sort [282, 286](#)  
    Comb sort [286, 287, 288](#)  
    Insertion sort [292](#)  
    Merge sort [296-301](#)  
    programming exercise based exercise [305-308](#)  
    quick sort [301-304](#)  
    Radix sort [292-296](#)  
    Selection sort [289-291](#)  
spanning tree [270](#)  
stack [5, 96, 108, 109](#)  
    creating, from single queue [156-159](#)  
    implementing [109](#)  
    implementing, with single list [114](#)  
    problems [129-132](#)  
    types [116](#)  
    uses [117](#)  
string  
    reversing [117](#)

string matching [353](#), [354](#)

## T

topological sort [266](#)  
graph [267-270](#)  
Tower of Hanoi [38-40](#)  
traversals  
    Breadth First Search (BFS) [262-266](#)  
    Depth First Search (DFS) [256-262](#)  
trees [5](#), [166](#), [167](#)  
    binary tree [167](#), [168](#)  
    leaves [169](#)  
    siblings [169](#)  
truncate method [339](#)  
two-dimensional arrays [4](#)  
two-way linked list [90](#)  
    node, deleting [92](#), [93-96](#)  
    node, inserting [91](#), [92](#)  
    traversing [91](#)

## U

unsorted sequential array  
example [335](#)