Data Structures and Program Design Using Python

Dr. Dheeraj Malhotra
Dr. Neha Malhotra

KHANNA PUBLISHERS®
Investing in Learning®

DATA STRUCTURES AND PROGRAM DESIGN USING PYTHON

A Self-Teaching Introduction

Dheeraj Malhotra, PhD
Neha Malhotra, PhD

KHANNA PUBLISHERS®
Investing in Learning®
Operational Office :
4575/15, Onkar House, Opp. Happy School,
Ground Floor, Daryaganj, New Delhi 110 002
Phones : 011-45033819 • *Mob*. 09811541460
email : contactus@khannapublishers.in

Published by :
Romesh Chander Khanna & Vineet Khanna
for KHANNA PUBLISHERS
2-B, Nath Market, Nai Sarak
Delhi- 110 006 (India)

Website : www.khannapublishers.in

© 1979 and onward

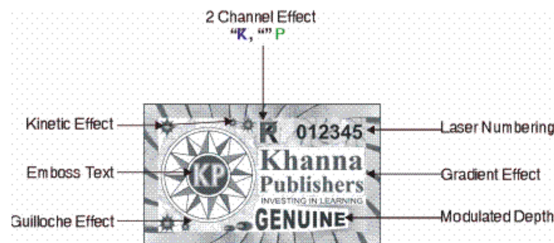
This book or part thereof cannot be translated or reproduced in any form without the written permission of the Authors and the Publishers. The right to translation, however, reserved with the authors alone.

Copyright: Authors and Publishers Jointly

Hologram & Description

To all readers of our books, to prevent yourself from being defrauded by piraters, please make sure that there is an Hologram on the cover of our books with the below specifications. If you find any book without Hologram and Description, please mail us at **contactus@khannapublishers.in**

Thanking you



ISBN No. : 978-93-92549-68-7

***First Edition :* 2024**

PREFACE

Data structures are the building blocks of computer science. The objective of this text is to emphasize the fundamentals of data structures as an introductory subject. It is designed for beginners who would like to learn the basics of data structures and their implementation using the Python programming language. With this focus in mind, we present various fundamentals of the subject, well supported with real-world analogies to enable a quick understanding of the technical concepts and to help the reader in quickly identifying appropriate data structures to solve specific, practical problems. This book will serve the purpose of a text or reference book and will be of immense help especially to undergraduate or graduate students of various courses in information technology, engineering, computer applications, and information sciences.

Key Features:

- **Practical Applications:** Real-world analogies as practical applications are given throughout the text to quickly understand and connect the fundamentals of data structures with day to day, real-world scenarios. This approach, in turn, will assist the reader in developing the capability to identify the most appropriate and efficient data structure for solving a specific, real-world problem.
- **Frequently Asked Questions:** Frequently asked theoretical or practical questions are integrated throughout the content of the book, within related topics to assist readers in grasping the subject.
- **Algorithms and Programs:** To better understand the fundamentals of data structures at a generic level-followed by their implementation in Python, syntax independent algorithms, as well as implemented programs in Python, are discussed throughout the book. This presentation will assist the reader in getting both algorithms and their corresponding implementation within a single book.

- *Numerical and Conceptual Exercises*: To assist the reader in developing a strong foundation of the subject, various numerical and conceptual problems are included throughout the text.
- *Multiple Choice Questions*: To assist students for placement-oriented exams in various IT fields, several exercises are suitably chosen and are given in an MCQ format.

September 2020

Dr. Dheeraj Malhotra
Dr. Neha Malhotra

CONTENTS

Preface xv
Acknowledgments xvii

Chapter 1: Introduction to Data Structures 1—16

1.1 Introduction 1
1.2 Types of Data Structures 2
1.2.1 Linear and Non-Linear Data Structures 2
1.2.2 Static and Dynamic Data Structures 3
1.2.3 Homogeneous and Non-Homogeneous Data Structures 3
1.2.4 Primitive and Non-Primitive Data Structures 3
1.2.5 Arrays/Lists 4
1.2.6 Stacks 4
1.2.7 Queues 5
1.2.8 Linked Lists 5
1.2.9 Trees 6
1.2.10 Graphs 7
1.3 Operations on Data Structures 8
1.4 Algorithms 9
1.4.1 Developing an Algorithm 9
1.5 Approaches for Designing an Algorithm 9
1.6 Analyzing an Algorithm 10
1.6.1 Time-Space Trade-Off 11

1.7 Abstract Data Types 12
1.8 Big O Notation 12
1.9 Summary 13

Chapter 2: Introduction to Python **17-47**
2.1 Introduction 17
2.2 Python and its Characteristics 17
2.3 Python Overview 19
2.4 Tools For Python 19
2.5 Easy Install and Pip 20
2.6 Quotations and Comments in Python 20
2.7 Compiling the Python Program 21
2.8 Object-Oriented Programming 22
2.9 Character Set Used in Python 23
2.10 Python Tokens 23
2.11 Data Types in Python 27
2.12 Structure of a Python Program 28
2.13 Operators in Python 29
2.14 Decision Control Statements 33
2.15 Looping Statements 35
2.16 Loop Control Statements 38
2.17 Methods 39
2.18 Summary 43
2.19 Exercises 44
2.19.1 Theory Questions 44
2.19.2 Programming Projects 45
2.19.3 Multiple Choice Questions 46

Chapter 3: Arrays/Lists **48-67**
3.1 Introduction 48
3.2 Definition of an Array 48
3.3 Array/List Declaration 49
3.4 Array/List Initialization 49
3.5 Calculating the Address of Array Elements 50
3.6 Operations on Arrays/Lists 51

CHAPTER 1

INTRODUCTION TO DATA STRUCTURES

1.1 INTRODUCTION

A data structure is an efficient way of storing and organizing the data elements in a computer's memory. *Data* means a value or a collection of values. *Structure* refers to a method of organizing the data. The mathematical or logical representation of data in the memory is referred to as a *data structure*. The objective of a data structure is to store, retrieve, and update the data efficiently. A data structure can be considered as all the elements grouped under one name. The data elements are called *members*, and they can be of different types. Data structures are used in almost every program and software system. There are various kinds of data structures that are suited for different types of applications. Data structures are the building blocks of a program. For a program to run efficiently, a programmer must choose the appropriate data structures. A data structure is a crucial part of data management. As the name suggests, *data management* is a task that includes different activities, like the collection of data and the organization of data into structures. Data structures are used in stacks, queues, arrays, binary trees, linked lists, and hash tables.

A data structure helps us to understand the relationship of one element to another element and organize it within the memory. It is a mathematical or logical representation or organization of data in the memory. Data structures are extensively applied in the following areas:

- Compiler Design
- Database Management Systems (DBMS)
- Artificial Intelligence
- Network and Numerical Analysis

- Statistical Analysis Packages
- Graphics
- Operating Systems (OS)
- Simulations

There are many applications in which different data structures are used for their operations. Some data structures sacrifice speed for the efficient utilization of memory, while others sacrifice memory utilization and result in a faster speed. In today's world, programmers aim not just to build a program, but to build an effective program. As previously discussed, for a program to be efficient, a programmer must choose the appropriate data structures. Hence, data structures are classified into various types. Now, let us discuss and learn about different types of data structures.

Frequently Asked Questions

1. Define the term "data structure."

Answer:

A data structure is an organization of data in a computer's memory or disk storage. In other words, a logical or mathematical model of a particular organization of data is called a data structure. A data structure in computer science is also a way of storing data in a computer so that it can be used efficiently. An appropriate data structure allows a variety of important operations to be performed using both resources, that is, the memory space and execution time, efficiently.

1.2 TYPES OF DATA STRUCTURES

Data structures are classified into various types.

1.2.1 Linear and Non-Linear Data Structures

A *linear data structure* is one in which the data elements are stored in a linear, or sequential, order; that is, data is stored in consecutive memory locations. A linear data structure can be represented in two ways; either it is represented by a linear relationship between various elements utilizing consecutive memory locations as in the case of arrays, or it may be represented by a linear relationship between the elements utilizing links from one element to another as in the case of linked lists. Examples of linear data structures include arrays, linked lists, stacks, and queues.

A *non-linear data structure* is one in which the data is not stored in any sequential order or consecutive memory locations. The data elements in this structure are represented by a hierarchical order. Examples of non-linear data structures include graphs and trees.

1.2.2 Static and Dynamic Data Structures

A *static data structure* is a collection of data in memory that is fixed in size and cannot be changed during runtime. The memory size must be known in advance, as the memory cannot be reallocated later in a program. One example is an *array*.

A *dynamic data structure* is a collection of data in which memory can be reallocated during the execution of a program. The programmer can add or remove elements according to his/her need. Examples include linked lists, graphs, and trees.

1.2.3 Homogeneous and Non-Homogeneous Data Structures

A *homogeneous data structure* is one that contains data elements of the same type (for example, arrays).

A *non-homogeneous data structure* contains data elements of different types (for example, structures).

1.2.4 Primitive and Non-Primitive Data Structures

Primitive data structures are the fundamental data structures or predefined data structures that are supported by a programming language. Examples of primitive data structure types are integer, float, and char.

Non-primitive data structures are comparatively more complicated data structures that are created using primitive data structures. Examples of non-primitive data structures are arrays, files, linked lists, stacks, and queues.

The classification of different data structures is shown in Figure 1.1.

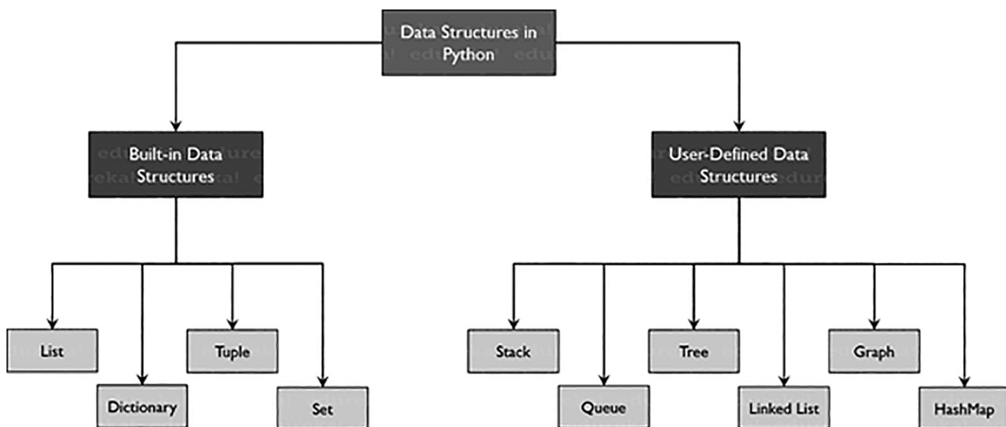


FIGURE 1.1 Classification of different data structures

Python supports various data structures. We now introduce all these data structures, and they are discussed in detail in the upcoming chapters.

Frequently Asked Questions

2. What is the difference between primitive data structures and non-primitive data structures?

Answer:

The data structures that are typically directly operated upon by machine-level instructions, that is, the fundamental data types such as int, float, and char, are known as primitive data structures. The data structures that are not fundamental are called non-primitive data structures.

Frequently Asked Questions

3. What is the difference between linear and non-linear data structures?

Answer:

The main difference between linear and non-linear data structures lies in the way in which data elements are organized. In the linear data structure, elements are organized sequentially, and therefore they are easy to implement in a computer's memory. In non-linear data structures, a data element can be attached to several other data elements to represent specific relationships existing among them.

1.2.5 Arrays/Lists

The array structure looks very similar to Python's list structure. That's because the two structures are both sequences that are composed of multiple sequential elements that can be accessed by position. But there are two major differences between the array and the list. First, an array has a limited number of operations, which commonly include those for array creation, reading a value from a specific element, and writing a value to a specific element. The list provides a large number of operations for working with the content of the list. Second, the list can grow and shrink during execution as elements are added or removed while the size of an array cannot be changed after it has been created.

Python's list structure is a mutable sequence container that can change size as items are added or removed. It is an abstract data type that is implemented using an array structure to store the items contained in the list.

In Python, a list is declared using the following syntax:

Syntax – pyList = [4, 12, 2, 34, 17]

1.2.6 Stacks

A *stack* is a collection of objects that are inserted and removed according to the Last-In, First-Out (LIFO) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack). The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack. When we need a new plate from the dispenser, we "pop" the

top plate off the stack, and when we add a plate, we "push" it down on the stack to become the new top plate.

> **Practical Application:**
>
> A real-life example of a stack is a pile of plates arranged on a table. A person will pick up the first plate from the top of the stack.

The Stack ADT can be implemented in several ways. The two most common approaches to implement Stack ADT in Python include the use of a Python list and a linked list. The choice depends on the type of application involved.

1.2.7 Queues

Another fundamental data structure is the *queue*. It is a close cousin of the stack, as a queue is a collection of objects that are inserted and removed according to the First-In, First-Out (FIFO) principle. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed. We usually say that elements enter a queue at the back and are removed from the front. A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the front of the line.

> **Practical Application:**
>
> For a simple illustration of a queue, imagine there is a line of people standing at the bus stop and waiting for the bus. The first person standing in the line will get into the bus first.

The Queue ADT can be implemented in several ways. The two most common approaches in Python include the use of a Python list and a linked list. The choice depends on the type of application involved.

1.2.8 Linked Lists

The major drawback of the array is that the size or the number of elements must be known in advance. Thus, this drawback gave rise to the new concept of a linked list. A *linked list* is a linear collection of data elements. These data elements are called *nodes*, which store the address of the next node. A linked list is a sequence of nodes in which each node contains one or more than one data field and an address field that stores the address of the next node. Linked lists are dynamic; that is, memory is allocated when required.

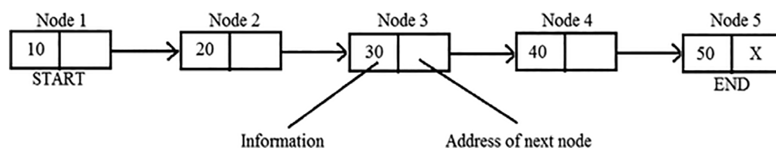


FIGURE 1.2 Memory representation of a linked list

Figure 1.2 shows a linked list in which each node is divided into two slots:

1. The first slot contains information/data.
2. The second slot contains the address of the next node.

Practical Application:

A simple real-life example is a train; here, each train car is connected to the previous one and next one (except the first car (the engine) and the last car (the coach)).

The address part of the last node stores a special value called NULL, which denotes the end of the linked list. The advantage of a linked list over arrays is that now it is easier to insert and delete data elements, as we don't have to do shifting each time. Yet searching for an element is difficult. More time is required to search for an element, and it requires a large amount of memory space. Hence, linked lists are used where a collection of data elements is required but the number of data elements in the collection is not known to us in advance.

Frequently Asked Questions

4. Define the term "linked list."

Answer:

A linked list or one-way list is a linear collection of data elements called nodes, which give a linear order. It is a popular dynamic data structure. The nodes in the linked list are not stored in consecutive memory locations. For every data item in a node of the linked list, there is an associated address field that gives the address location of the next node in the linked list.

1.2.9 Trees

A *tree* is a popular non-linear data structure in which the data elements or the nodes are represented in a hierarchical order. Here, one of the nodes is shown as the root node of the tree, and the remaining nodes are partitioned into two disjointed sets such that each set is a part of a sub-tree. A tree makes the search process very easy, and its recursive programming makes a program optimized and easy to understand.

A binary tree is the simplest form of a tree. A *binary tree* consists of a root node and two sub-trees known as the left sub-tree and the right sub-tree, where both sub-trees are also binary trees. Each node in a tree consists of three parts, that is, the extreme left part stores the address of the left sub-tree, the middle part consists of the data element, and the extreme right part stores the address of the right sub-tree. The root is the topmost element of the tree. When there are no nodes in a tree, that is, when ROOT = NULL, then it is called an *empty tree*.

For example, consider a binary tree where R is the root node of the tree. LEFT and RIGHT are the left and right sub-trees of R, respectively. Node A is designated as the root node of the tree. Nodes B and C are the left and right child of A, respectively. Nodes B, D, E, and G

constitute the left sub-tree of the root. Similarly, nodes C, F, H, and I constitute the right sub-tree of the root.

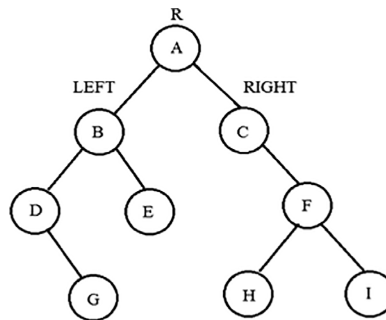


FIGURE 1.3 A binary tree

Advantages of a tree

1. The searching process is very fast in trees.
2. The insertion and deletion of the elements is easier compared to other data structures.

Frequently Asked Questions

5. Define the term "binary tree."

Answer:

A binary tree is a hierarchal data structure in which each node has at most two children, that is, the left and right child. In a binary tree, the degree of each node can be at most two. Binary trees are used to implement binary search trees, which are used for efficient searching and sorting. A variation of BST is an AVL tree, where the height of the left and right subtree differs by one. A binary tree is a popular subtype of a k-ary tree, where k is 2.

1.2.10 Graphs

A *graph* is a general tree with no parent-child relationship. It is a non-linear data structure that consists of vertices, also called nodes, and the edges that connect those vertices. In a graph, any complex relationship can exist. A graph G may be defined as a finite set of V vertices and E edges. Therefore, G = (V, E) where V is the set of vertices and E is the set of edges. Graphs are used in various applications of mathematics and computer science. Unlike a root node in trees, graphs don't have root nodes; rather, the nodes can be connected to any node in the graph. Two nodes are called *neighbors* when they are connected via an edge.

> **Practical Application:**
>
> A real-life example of a graph can be seen in workstations where several computers are joined to one another via network connections.

For example, consider a graph G with six vertices and eight edges. Here, Q and Z are neighbors of P. Similarly, R and T are neighbors of S.

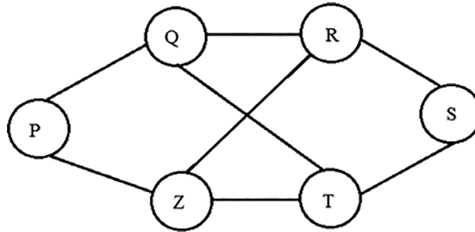


FIGURE 1.4 A graph

1.3 OPERATIONS ON DATA STRUCTURES

Here, we discuss various operations that are performed on data structures.

- **Creation** – This is the process of creating a data structure. The declaration and initialization of the data structure are done here. It is the first operation.
- **Insertion** – This is the process of adding new data elements in the data structure, for example, to add the details of an employee who has recently joined an organization.
- **Deletion** – This is the process of removing a particular data element from the given collection of data elements, for example, to remove the name of an employee who has left the company.
- **Updating** – This is the process of modifying the data elements of a data structure. For example, if the address of a student is changed, it should be updated.
- **Searching** – This is used to find the location of a particular data element or all the data elements with the help of a given key, for example, to find the names of people who live in New York.
- **Sorting** – This is the process of arranging the data elements in some order, that is, either ascending or descending order. An example is arranging the names of students of a class in alphabetical order.
- **Merging** – This is the process of combining the data elements of two different lists to form a single list of data elements.
- **Traversal** – This is the process of accessing each data element exactly once so that it can be processed. An example is to print the names of all the students of a class.
- **Destruction** – This is the process of deleting the entire data structure. It is the last operation in the data structure.

1.4 ALGORITHMS

An *algorithm* is a systematic set of instructions combined to solve a complex problem. It is a step-by-finite-step sequence of instructions, each of which has a clear meaning and can be executed in a minimum amount of effort in finite time. In general, an algorithm is a blueprint for writing a program to solve the problem. Once we have a blueprint of the solution, we can easily implement it in any high-level language like C, C++, or Python. It divides the problem into a finite number of steps. An algorithm written in a programming language is known as a *program*. A computer is a machine with no brain or intelligence. Therefore, the computer must be instructed to perform a given task in unambiguous steps. Hence, a programmer must define his problem in the form of an algorithm written in English. Thus, such an algorithm should have the following features:

1. An algorithm should be simple and concise.
2. It should be efficient and effective.
3. It should be free of ambiguity; that is, the logic must be clear.

Similarly, an algorithm must have the following characteristics:

- **Input** – It reads the data of the given problem.
- **Output** – The desired result must be produced.
- **Process/Definiteness** – Each step or instruction must be unambiguous.
- **Effectiveness** – Each step should be accurate and concise. The desired result should be produced within a finite time.
- **Finiteness** – The number of steps should be finite.

1.4.1 Developing an Algorithm

To develop an algorithm, some steps are necessary:

1. Defining or understanding the problem.
2. Identifying the result or output of the problem.
3. Identifying the inputs required by the problem and choosing the best input.
4. Designing the logic from the given inputs to get the desired output.
5. Testing the algorithm for different inputs.
6. Repeating the previous steps until it produces the desired result for all the inputs.

1.5 APPROACHES FOR DESIGNING AN ALGORITHM

A complicated algorithm is divided into smaller units called *modules*. These modules are further divided into sub-modules. Thus, in this way, a complex algorithm can easily be solved. The process of dividing an algorithm into modules is called *modularization*. There are two popular approaches for designing an algorithm:

- Top-Down Approach
- Bottom-Up Approach

Now let us understand both approaches.

1. **Top-Down Approach**–A *top-down approach* states that the complex/complicated problem/algorithm should be divided into a smaller number of one or more modules. These smaller modules are further divided into one or more sub-modules. This process of decomposition is repeated until we achieve the desired output of module complexity. A top-down approach starts from the topmost module, and the modules are incremented accordingly until a level is reached where we don't require any more sub-modules, that is, the desired level of complexity is achieved.

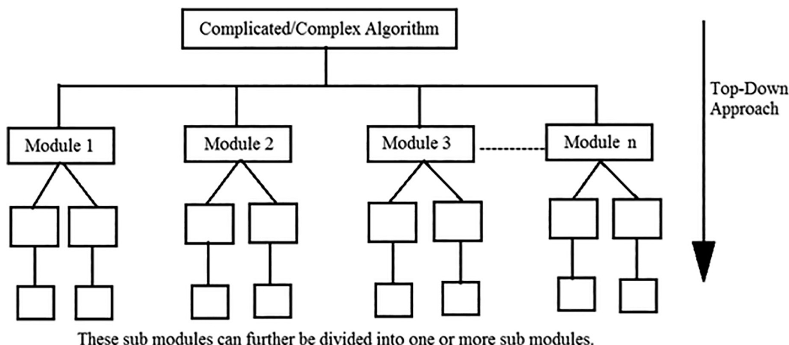


FIGURE 1.5 Top-down approach

2. **Bottom-Up Approach**–A bottom-up algorithm design approach is the opposite of a top-down approach. In this kind of approach, we first start with designing the basic modules and proceed further toward designing the high-level modules. The sub-modules are grouped to form a module of a higher level. Similarly, all high-level modules are grouped to form more high-level modules. Thus, this process of combining the sub-modules is repeated until we obtain the desired output of the algorithm.

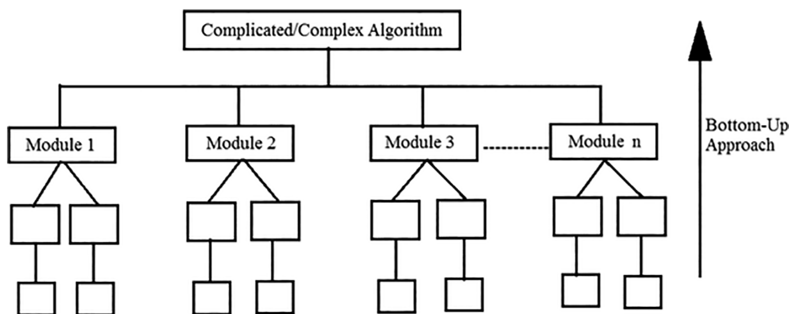


FIGURE 1.6 Bottom-up approach

1.6 ANALYZING AN ALGORITHM

An algorithm can be analyzed by two factors: space and time. We should develop an algorithm that makes the best use of both these resources. Analyzing an algorithm measures the

Data Structures and Program Design Using Python

About the Book

This book emphasized the Fundamentals of Data Structures as an introductory subject and also designed for beginners, who would like to learn the basics of data structures and their implementation using the Python programming language. This book present various fundamentals of the subject, well supported with real-world analogies to enable a quick understanding of the technical concepts and to help in identifying appropriate data structures to solve specific practical problems. This book will serve the purpose of a text, immense help especially to undergraduate or graduate students of various courses in Information Technology, Computer Applications, and Information Sciences.

About the Authors

Dr. Dheeraj Malhotra is a Senior Faculty in the Information Technology, Department of VIPS, GGSIPU. He is a Google Certified Educator (Level 1) and recipient of the Best Doctoral Thesis and Best Research Paper Award from the University of Kota and the Association of Indian Management Schools respectively. He has published numerous research papers with reputed International Publishers like ACM, Springer, Elsevier, Taylor-Francis, and IEEE. His teaching and research interests include Data Structures, Computer Graphics, Machine Learning, and Big Data Analytics.

Dr. Neha Malhotra is a Senior Faculty to students of Computer Applications in VIPS, GGSIPU. She is conferred with Best Research Paper Award in AIMS-2019. She has research publications with International publishers of repute like Emerald, Taylor-Francis, IEEE, and Springer. Her teaching and research interests include Data Structures, OOP with C++, Advanced Java, and Big Data Analytics.

KHANNA PUBLISHERS®
ISO 9001:2015
4575/15, Onkar House, Opp. Happy School,
Ground Floor, Daryaganj, New Delhi-110002
Phones: 011-45033819, 9811541460
E-mail: contactus@khannapublishers.in

Website:
www.khannapublishers.in