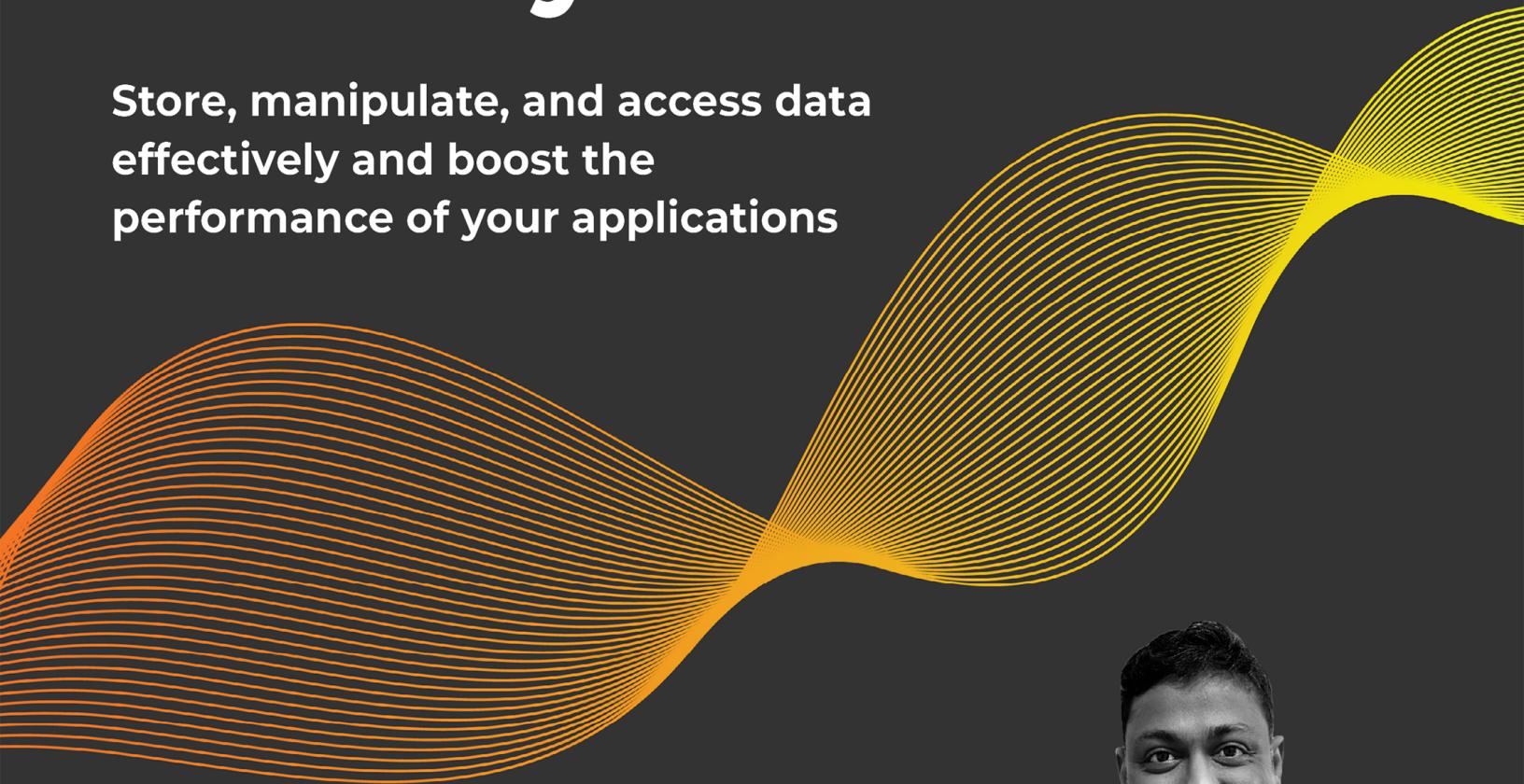


EXPERT INSIGHT



Hands-On Data Structures and Algorithms with Python

Store, manipulate, and access data
effectively and boost the
performance of your applications



Third Edition



Dr. Basant Agarwal

<packt>

Hands-On Data Structures and Algorithms with Python

Third Edition

Store, manipulate, and access data effectively and boost the performance of your applications

Dr. Basant Agarwal



BIRMINGHAM—MUMBAI

“Python” and the Python Logo are trademarks of the Python Software Foundation.

Hands-On Data Structures and Algorithms with Python

Third Edition

Copyright © 2022 Packt Publishing

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 Packt Publishing 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.

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

Senior Publishing Product Manager: Denim Pinto

Acquisition Editor – Technical Reviews: Saby Dsilva

Project Editor: Rianna Rodrigues

Content Development Editor: Rebecca Robinson

Copy Editor: Safis Editing

Technical Editor: Karan Sonawane

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Presentation Designer: Ganesh Bhadwalkar

First published: May 2017

Second edition: October 2018

Third edition: July 2022

Production reference: 1150722

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80107-344-8

www.packt.com

Contributors

About the author

Dr. Basant Agarwal is working as an Assistant Professor at the Department of Computer Science and Engineering, Indian Institute of Information Technology Kota (IIIT-Kota), India, which is an Institute of National Importance. He holds a Ph.D. and M.Tech. from the Department of Computer Science and Engineering, Malaviya National Institute of Technology Jaipur, India. He has more than 9 years of experience in research and teaching. He has worked as a Postdoc Research Fellow at the Norwegian University of Science and Technology (NTNU), Norway, under the prestigious **European Research Consortium for Informatics and Mathematics (ERCIM)** fellowship in 2016. He has also worked as a Research Scientist at Temasek Laboratories, **National University of Singapore (NUS)**, Singapore. His research interests are in artificial intelligence, cyber-physical systems, text mining, natural language processing, machine learning, deep learning, intelligent systems, expert systems, and related areas.

This book is dedicated to my family, and friends.

Thank you to Benjamin Baka for his hard work in the first edition.

– Dr. Basant Agarwal

About the reviewers

Patrick Arminio is a software engineer based in London. He's currently the chair of Python Italia, an association that organizes Python events in Italy.

He's been working with Python for more than 10 years, focusing on web development using Django. He's also the maintainer of Strawberry GraphQL, an open source Python library for creating GraphQL APIs.

Dong-hee Na is a software engineer and an open-source enthusiast. He works at Line Corporation as a backend engineer. He has professional experience in machine learning projects based on Python and C++. As for his open-source works, he focuses on the compiler and interpreter area, especially for Python-related projects. He has been a CPython core developer since 2020.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



Contents

Preface

Who this book is for

What this book covers

To get the most out of this book

Get in touch

1. Python Data Types and Structures

Introducing Python 3.10

Installing Python

Windows operating system

Linux-based operating systems

Mac operating system

Setting up a Python development environment

Setup via the command line

Setup via Jupyter Notebook

Overview of data types and objects

Basic data types

Numeric

Boolean

Sequences

Strings

Range

Lists

Membership, identity, and logical operations

Membership operators

Identity operators

Logical operators

Tuples

Complex data types

Dictionaries

Sets

Immutable sets

Python's collections module

Named tuples

Deque

[Ordered dictionaries](#)

[Default dictionary](#)

[ChainMap object](#)

[Counter objects](#)

[UserDict](#)

[UserList](#)

[UserString](#)

[Summary](#)

2. [Introduction to Algorithm Design](#)

[Introducing algorithms](#)

[Performance analysis of an algorithm](#)

[Time complexity](#)

[Space complexity](#)

[Asymptotic notation](#)

[Theta notation](#)

[Big O notation](#)

[Omega notation](#)

[Amortized analysis](#)

[Composing complexity classes](#)

[Computing the running time complexity of an algorithm](#)

[Summary](#)

[Exercises](#)

3. [Algorithm Design Techniques and Strategies](#)

[Algorithm design techniques](#)

[Recursion](#)

[Divide and conquer](#)

[Binary search](#)

[Merge sort](#)

[Dynamic programming](#)

[Calculating the Fibonacci series](#)

[Greedy algorithms](#)

[Shortest path problem](#)

[Summary](#)

[Exercises](#)

4. [Linked Lists](#)

[Arrays](#)

[Introducing linked lists](#)

Nodes and pointers

Singly linked lists

Creating and traversing

Improving list creation and traversal

Appending items

Appending items to the end of a list

Appending items at intermediate positions

Querying a list

Searching an element in a list

Getting the size of the list

Deleting items

Deleting the node at the beginning of the singly linked list

Deleting the node at the end in the singly linked list

Deleting any intermediate node in a singly linked list

Clearing a list

Doubly linked lists

Creating and traversing

Appending items

Inserting a node at beginning of the list

Inserting a node at the end of the list

Inserting a node at an intermediate position in the list

Querying a list

Deleting items

Circular lists

Creating and traversing

Appending items

Querying a list

Deleting an element in a circular list

Practical applications of linked lists

Summary

Exercise

5. Stacks and Queues

Stacks

Stack implementation using arrays

Stack implementation using linked lists

Push operation

[Pop operation](#)
[Peek operation](#)
[Applications of stacks](#)

[Queues](#)

[Python's list-based queues](#)
 [The enqueue operation](#)
 [The dequeue operation](#)

[Linked list based queues](#)
 [The enqueue operation](#)
 [The dequeue operation](#)

[Stack-based queues](#)
 [Approach 1: When the dequeue operation is costly](#)
 [Approach 2: When the enqueue operation is costly](#)
 [Enqueue operation](#)
 [Dequeue operation](#)
 [Applications of queues](#)

[Summary](#)

[Exercises](#)

6. [Trees](#)

[Terminology](#)

[Binary trees](#)

[Implementation of tree nodes](#)
[Tree traversal](#)
 [In-order traversal](#)
 [Pre-order traversal](#)
 [Post-order traversal](#)
 [Level-order traversal](#)
[Expression trees](#)
 [Parsing a reverse Polish expression](#)

[Binary search trees](#)

[Binary search tree operations](#)
 [Inserting nodes](#)
 [Searching the tree](#)
 [Deleting nodes](#)
 [Finding the minimum and maximum nodes](#)
 [Benefits of a binary search tree](#)
[Summary](#)

Exercises

7. Heaps and Priority Queues

Heaps

Insert operation

Delete operation

Deleting an element at a specific location from a heap

Heap sort

Priority queues

Summary

Exercises

8. Hash Tables

Introducing hash tables

Hashing functions

Perfect hashing functions

Resolving collisions

Open addressing

Linear probing

Implementing hash tables

Storing elements in a hash table

Growing a hash table

Retrieving elements from the hash table

Testing the hash table

Implementing a hash table as a dictionary

Quadratic probing

Double hashing

Separate chaining

Symbol tables

Summary

Exercise

9. Graphs and Algorithms

Graphs

Directed and undirected graphs

Directed acyclic graphs

Weighted graphs

Bipartite graphs

Graph representations

Adjacency lists

[Adjacency matrix](#)

[Graph traversals](#)

[Breadth-first traversal](#)

[Depth-first search](#)

[Other useful graph methods](#)

[Minimum Spanning Tree](#)

[Kruskal's Minimum Spanning Tree algorithm](#)

[Prim's Minimum Spanning Tree algorithm](#)

[Summary](#)

[Exercises](#)

10. [Searching](#)

[Introduction to searching](#)

[Linear search](#)

[Unordered linear search](#)

[Ordered linear search](#)

[Jump search](#)

[Binary search](#)

[Interpolation search](#)

[Exponential search](#)

[Choosing a search algorithm](#)

[Summary](#)

[Exercise](#)

11. [Sorting](#)

[Technical requirements](#)

[Sorting algorithms](#)

[Bubble sort algorithms](#)

[Insertion sort algorithm](#)

[Selection sort algorithm](#)

[Quicksort algorithm](#)

[Implementation of quicksort](#)

[Timsort algorithm](#)

[Summary](#)

[Exercise](#)

12. [Selection Algorithms](#)

[Technical requirements](#)

[Selection by sorting](#)

[Randomized selection](#)

[Quickselect](#)

[Deterministic selection](#)

[Implementation of the deterministic selection algorithm](#)

[Summary](#)

[Exercise](#)

13. [String Matching Algorithms](#)

[Technical requirements](#)

[String notations and concepts](#)

[Pattern matching algorithms](#)

[The brute force algorithm](#)

[The Rabin-Karp algorithm](#)

[Implementing the Rabin-Karp algorithm](#)

[The Knuth-Morris-Pratt algorithm](#)

[The prefix function](#)

[Understanding the KMP algorithm](#)

[Implementing the KMP algorithm](#)

[The Boyer-Moore algorithm](#)

[Understanding the Boyer-Moore algorithm](#)

[Bad character heuristic](#)

[Good suffix heuristic](#)

[Implementing the Boyer-Moore algorithm](#)

[Summary](#)

[Exercise](#)

[Appendix: Answers to the Questions](#)

[Chapter 2: Introduction to Algorithm Design](#)

[Chapter 3: Algorithm Design Techniques and Strategies](#)

[Chapter 4: Linked Lists](#)

[Chapter 5: Stacks and Queues](#)

[Chapter 6: Trees](#)

[Chapter 7: Heaps and Priority Queues](#)

[Chapter 8: Hash Tables](#)

[Chapter 9: Graphs and Algorithms](#)

[Chapter 10: Searching](#)

[Chapter 11: Sorting](#)

[Chapter 12: Selection Algorithm](#)

[Chapter 13: String Matching Algorithms](#)

[Other Books You May Enjoy](#)

[Index](#)

Preface

Data structures play a vital role in storing and organizing data within an application. It is important to choose the right data structure to significantly improve the application's performance, as it is highly desirable to be able to scale the application as the data quantity increases. This new edition teaches you essential Python data structures and the most common and important algorithms for building easy, maintainable applications. It also allows you to implement these algorithms with working examples and easy to follow step-by-step instructions.

In this book, you will learn the essential Python data structures and the most common algorithms. With this easy-to-read book, you will learn how to create complex data structures such as linked lists, stacks, heaps, queues, trees, and graphs as well as sorting algorithms including bubble sort, insertion sort, heapsort, and quicksort. We also describe various selection algorithms such as randomized and deterministic selection and provide a detailed discussion of various data structure algorithms and design paradigms such as greedy algorithms, divide-and-conquer, and dynamic programming. In addition, complex data structures such as trees and graphs are explained with easy pictorial examples to understand the concepts of these useful data structures. You will also learn various important string processing and pattern-matching algorithms such as KMP and

Boyer-Moore algorithms along with their easy implementation in Python.

Who this book is for

This book is intended for Python developers who are studying data structures and algorithms at a beginner or intermediate level, as chapters provide practical examples and an easy approach to complex algorithms. It may also be useful for engineering students on a course in data structures and algorithms, as it covers almost all the algorithms, concepts, and designs that are studied. This book is also designed for software developers who want to deploy various applications using a specific data structures, as this book provides efficient ways to store relevant data.

It is assumed that the reader has some basic knowledge of the Python; however, it is not necessary, as we provide a quick overview of Python and object-oriented concepts.

What this book covers

Chapter 1, Python Data Types and Structures, introduces the basic data types and structures in Python. It will provide an overview of several built-in data structures available in Python that are pivotal for understanding the internals of data structures.

Chapter 2, Introduction to Algorithm Design, provides details about algorithm design issues and techniques. This chapter will compare different analyzing algorithms via running time and computation complexity, which will tell us which ones perform better than others for a given problem.

Chapter 3, Algorithm Design Techniques and Strategies, covers various important data structure design paradigms such as greedy algorithms, dynamic programming, divide-and-conquer. We will learn to create data structures via a number of primary principles, such as robustness, adaptability and reusability, and learn to separate structure from a function.

Chapter 4, Linked Lists, covers linked lists, which are one of the most common data structures and are often used to implement other structures, such as stacks and queues. In this chapter, we describe linked lists, their operation, and implementation. We compare their behavior to arrays and discuss the relative advantages and disadvantages of each.

Chapter 5, Stacks and Queues, describes stack and queue data structures in detail. It also discusses the behavior and demonstrates

some implementations of these linear data structures. We give examples of typical real-life example applications.

Chapter 6, Trees, considers how trees form the basis of many of the most important advanced data structures. In this chapter we look at how to implement a binary tree. We will examine how to traverse trees and retrieve and insert values.

Chapter 7, Heaps and Priority Queues, looks into priority queues as important data structures and shows how to implement them using heap.

Chapter 8, Hash Tables, describes symbol tables, gives some typical implementations, and discusses various applications. We will look at the process of hashing, give an implementation of a hash table, and discuss the various design considerations.

Chapter 9, Graphs and Algorithms, looks at some of the more specialized structures, including graphs and spatial structures. We will learn to represent data through nodes and vertices and create structures such as directed and undirected graphs. We will also learn different algorithms for minimum spanning trees such as Prim's algorithm and Kruskal's algorithm.

Chapter 10, Searching, discusses the most common searching algorithms including, binary search and interpolation searching algorithms. We also give examples of their use for various data structures. Searching a data structure is a fundamental task and there are a number of approaches.

Chapter 11, Sorting, looks at the most common approaches to sorting. This will include bubble sort, insertion sort, selection sort, quick sort,

and heap sort algorithms with detailed explanations, along with their Python implementations.

Chapter 12, Selection Algorithms, discusses how selection algorithms are commonly used to find the i^{th} smallest element from the list. It is an important operation related to sorting algorithms, and broadly related to the data structures and algorithms.

Chapter 13, String Matching Algorithms, covers basic concepts and definitions related to strings. In this chapter, various string and pattern matching algorithms are discussed in detail such as the naïve approach, and the **Knuth-Morris-Pratt (KMP)** and Boyer-Moore pattern matching algorithms.

Appendix, Answers to the Questions, provides answers to the exercises at the end of each chapter. Please feel free to check the appendix at the end of the book.

There is also bonus content available online related to tree algorithms at https://static.packt-cdn.com/downloads/9781801073448_Bonus_Content.pdf.

To get the most out of this book

The code in this book needs to be run on Python 3.10 or higher. Python's interactive environment can also be used to run the code snippets. It is advised to learn the algorithms and concepts by executing the code provided in the book to better understand the algorithms. The book is aimed to give practical exposure to the readers, so it is recommended to do the programming for all the algorithms to get maximum out of this book.

Download the example code files

The code bundle for the book is hosted on GitHub at

<https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Third-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:

https://static.packt-cdn.com/downloads/9781801073448_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: “The ‘`not in`’ operator returns `True` if it does not find a variable in the specified sequence and `False` if it is found.”

A block of code is set as follows:

```
p = "Hello India"  
q = 10  
r = 10.2  
print(type(p))  
print(type(q))  
print(type(r))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
while self.slots[h] != None:  
    if self.slots[h].key == key:  
        return self.slots[h].value  
    h = (h + j * (self.prime_num - (self.h2(key) % self.prime_num)))  
    j = j + 1  
return None
```

Any command-line input or output is written as follows:

```
sudo apt-get install python3.10
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: “Each position in the hash table is often called a **slot** or **bucket** that can store an element.”



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details.

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 copyright@packtpub.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 authors.packtpub.com.

Share your thoughts

Once you've read *Hands-On Data Structures and Algorithms with Python - Third Edition*, we'd love to hear your thoughts! Please [click here to go straight to the Amazon review page](#) for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Python Data Types and Structures

Data structures and algorithms are important components in the development of any software system. An algorithm can be defined as a set of step-by-step instructions to solve any given problem; an algorithm processes the data and produces the output results based on the specific problem. The data used by the algorithm to solve the problem has to be stored and organized efficiently in the computer memory for the efficient implementation of the software. The performance of the system depends upon the efficient access and retrieval of the data, and that depends upon how well the data structures that store and organize the data in the system are chosen.

Data structures deal with how the data is stored and organized in the memory of the computer that is going to be used in a program. Computer scientists should understand how efficient an algorithm is and which data structure should be used in its implementation. The Python programming language is a robust, powerful, and widely used language to develop software-based systems. Python is a high-level, interpreted, and object-oriented language that is very convenient to learn and understand the concepts of data structures and algorithms.

In this chapter, we briefly review the Python programming language components that we will be using to implement the various data structures discussed in this book. For a more detailed discussion on the Python language in broader terms, take a look at the Python documentation:

- <https://docs.python.org/3/reference/index.html>
- <https://docs.python.org/3/tutorial/index.html>

In this chapter, we will look at the following topics:

- Introducing Python 3.10
- Installing Python
- Setting up a Python development environment
- Overview of data types and objects
- Basic data types
- Complex data types
- Python's collections module

Introducing Python 3.10

Python is an interpreted language: the statements are executed line by line. It follows the concepts of object-oriented programming. Python is dynamically typed, which makes it an ideal candidate among languages for scripting and fast-paced development on many platforms. Its source code is open source, and there is a very big community that is using and developing it continuously, at a very fast pace.

Python code can be written in any text editor and saved with the `.py` file extension. Python is easy to use and learn because of its compactness and elegant syntax.

Since the Python language will be used to write the algorithms, an explanation is provided of how to set up the environment to run the examples.

Installing Python

Python is preinstalled on Linux- and Mac-based operating systems. However, you will want to install the latest version of Python, which can be done on different operating systems as per the following instructions.

Windows operating system

For Windows, Python can be installed through an executable `.exe` file.

1. Go to <https://www.python.org/downloads/>.
2. Choose the latest version of Python—currently, it is 3.10.0—according to your architecture. If you have a 32-bit version of Windows, choose the 32-bit installer; otherwise, choose the 64-bit installer.
3. Download the `.exe` file.
4. Open the `python-3.10.0.exe` file.
5. Make sure to check **Add Python 3.10.0 to PATH**.
6. Click **Install Now** and then wait until the installation is complete; you can now use Python.
7. To verify that Python is installed correctly, open the Command Prompt and type the `python --version` command. It should output `Python 3.10.0`.

Linux-based operating systems

To install Python on a Linux machine, take the following steps:

1. Check whether you have Python preinstalled by entering the `python --version` command in the terminal.
2. If you do not have Python installed, then install it through the following command:

```
sudo apt-get install python3.10
```

3. Now, verify that you have installed Python correctly by typing the `python3.10 --version` command in the terminal. It should output `Python 3.10.0`.

Mac operating system

To install Python on a Mac, take the following steps:

1. Go to <https://www.python.org/downloads/>.
2. Download and open the installer file for `Python 3.10.0`.
3. Click **Install Now**.

4. To verify that Python is installed correctly, open the terminal and type `python -version`. It should output `Python 3.10.0`.

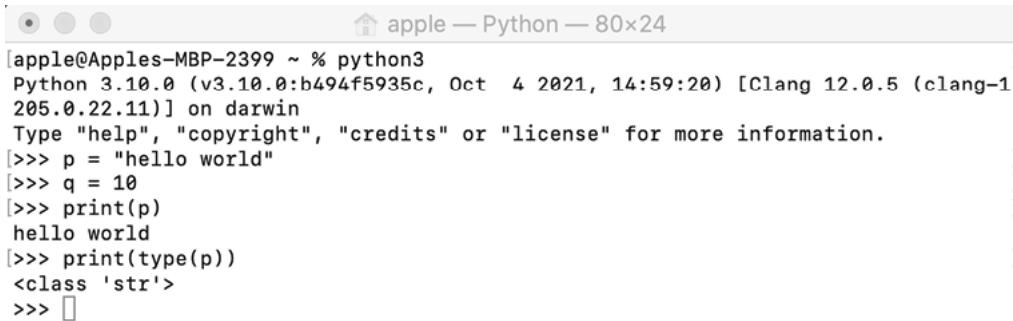
Setting up a Python development environment

Once you have installed Python successfully for your respective OS, you can start this hands-on approach with data structures and algorithms. There are two popular methods to set up the development environment.

Setup via the command line

The first method to set up the Python executing environment is via the command line, after installation of the Python package on your respective operating system. It can be set up using the following steps.

1. Open the terminal on Mac/Linux OS or Command Prompt on Windows.
2. Execute the Python 3 command to start Python, or simply type `py` to start Python in the Windows Command Prompt.
3. Commands can be executed on the terminal.



```
[apple@apples-MBP-2399 ~ % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021, 14:59:20) [Clang 12.0.5 (clang-1
205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> p = "hello world"
[>>> q = 10
[>>> print(p)
hello world
[>>> print(type(p))
<class 'str'>
>>> ]]
```

Figure 1.1: Screenshot of the command-line interface for Python

The User Interface for the command-line execution environment is shown in *Figure 1.1*.

Setup via Jupyter Notebook

The second method to run the Python program is through Jupyter Notebook, which is a browser-based interface where we can write the code. The User Interface of Jupyter Notebook is shown in *Figure 1.2*. The place where we can write the code is called a “cell.”

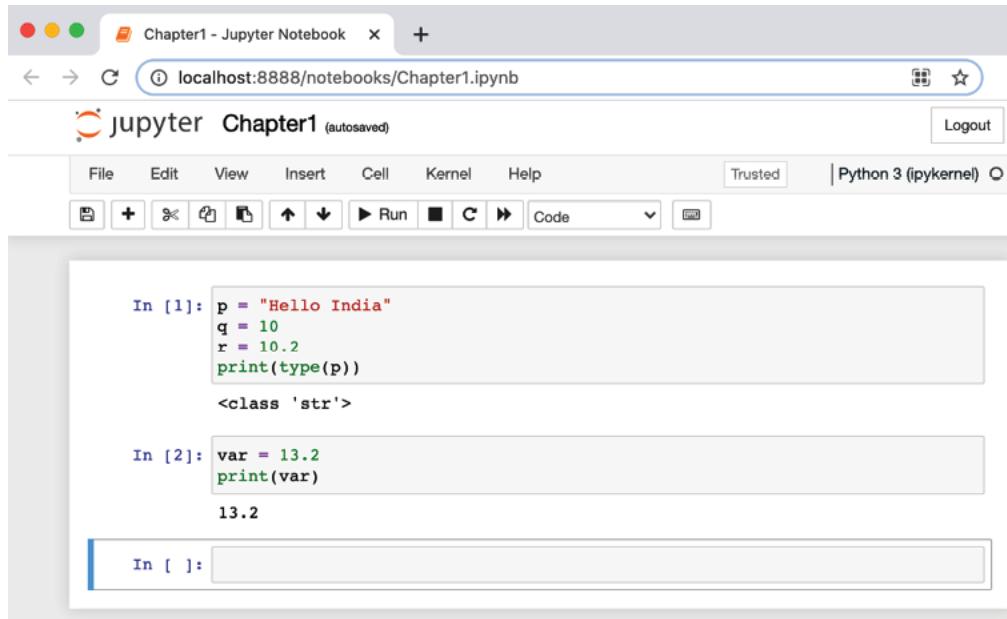


Figure 1.2: Screenshot of the Jupyter Notebook interface

Once Python is installed, on Windows, Jupyter Notebook can be easily installed and set up using a scientific Python distribution called Anaconda by taking the following steps.

1. Download the Anaconda distribution from <https://www.anaconda.com/products/individual>.
2. Install it according to the installation instructions.
3. Once installed, on Windows, we can run the notebook by executing the `jupyter notebook` command at the Command Prompt. Alternatively, following installation, the `Jupyter Notebook` app can be searched for and run from the taskbar.
4. On Linux/Mac operating systems, Jupyter Notebook can be installed using `pip3` by running the following code in the terminal:

```
pip3 install notebook
```

5. After installation of Jupyter Notebook, we can run it by executing the following command at the Terminal:

```
jupyter notebook
```



On some systems, this command does not work, depending upon the operating system or system configuration. In that case, Jupyter Notebook should start by executing the following command on the terminal.

```
python3 -m notebook
```

It is important to note that we will be using Jupyter Notebook to execute all the commands and programs throughout the book, but the code will also function in the command line if you'd prefer to use that.

Overview of data types and objects

Given a problem, we can plan to solve it by writing a computer program or software. The first step is to develop an algorithm, essentially a step-by-step set of instructions to be followed by a computer system, to solve the problem. An algorithm can be converted into computer software using any programming language. It is always desired that the computer software or program be as efficient and fast as possible; the performance or efficiency of the computer program also depends highly on how the data is stored in the memory of a computer, which is then going to be used in the algorithm.

The data to be used in an algorithm has to be stored in variables, which differ depending upon what kind of values are going to be stored in those variables. These are called *data types*: an integer variable can store only integer numbers, and a float variable can store real numbers, characters, and so on. The variables are containers that can store the values, and the values are the contents of different data types.

In most programming languages, variables and their data types must initially be declared, and then only that type of data can be statically stored in those variables. However, in Python, this is not the case. Python is a dynamically typed language; the data type of the variables is not required to be explicitly defined. The Python interpreter implicitly binds the value of the variable with its type at runtime. In Python, data types of the variable type can be checked using the function `type()`, which returns the type of variable passed. For example, if we enter the following code:

```
p = "Hello India"  
q = 10  
r = 10.2  
print(type(p))  
print(type(q))  
print(type(r))  
print(type(12+31j))
```

We will get an output like the following:

```
<class 'str'>  
<class 'int'>  
<class 'float'>  
<class 'complex'>
```

The following example, demonstrates a variable that has a `var` float value, which is substituted for a string value:

```
var = 13.2  
print(var)  
  
print(type(var))
```

```
var = "Now the type is string"
print(type(var))
```

The output of the code is:

```
13.2
<class 'float'>
<class 'str'>
```

In Python, every item of data is an object of a specific type. Consider the preceding example; here, when a variable `var` is assigned a value of `13.2`, the interpreter initially creates a float object having a value of `13.2`; a variable `var` then points to that object as shown in *Figure 1.3*:

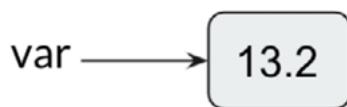


Figure 1.3: Variable assignment

Python is an easy-to-learn object-oriented language, with a rich set of built-in data types. The principal built-in types are as follows and will be discussed in more detail in the following sections:

- Numeric types: `Integer (int)`, `float`, `complex`
- Boolean types: `bool`
- Sequence types: `String (str)`, `range`, `list`, `tuple`
- Mapping types: `dictionary (dict)`
- Set types: `set`, `frozenset`

We will divide these into basic (numeric, Boolean, and sequence) and complex (mapping and set) data types. In subsequent sections, we will discuss them one by one in detail.

Basic data types

The most basic data types are numeric and Boolean types. We'll cover those first, followed by sequence data types.

Numeric

Numeric data type variables store numeric values. Integer, float, and complex values belong to this data type. Python supports three types of numeric types:

- **Integer (int)**: In Python, the interpreter takes a sequence of decimal digits as a decimal value, such as the integers `45`, `1000`, or `-25`.
- **Float**: Python considers a value having a floating-point value as a float type; it is specified with a decimal point. It is used to store floating-point numbers such as `2.5` and `100.98`. It is accurate up to `15` decimal points.

- **Complex:** A complex number is represented using two floating-point values. It contains an ordered pair, such as $a + ib$. Here, a and b denote real numbers and i denotes the imaginary component. The complex numbers take the form of `3.0 + 1.3i`, `4.0i`, and so on.

Boolean

This provides a value of either `True` or `False`, checking whether any statement is true or false. `True` can be represented by any non-zero value, whereas `False` can be represented by 0. For example:

```
print(type(bool(22)))
print(type(True))
print(type(False))
```

The output will be the following:

```
<class 'bool'>
<class 'bool'>
<class 'bool'>
```

In Python, the numeric values can be used as bool values using the built-in `bool()` function. Any number (integer, float, complex) having a value of zero is regarded as `False`, and a non-zero value is regarded as `True`. For example:

```
bool(False)
print(bool(False))
va1 = 0
print(bool(va1))
va2 = 11
print(bool(va2))
va3 = -2.3
print(bool(va3))
```

The output of the above code will be as follows.

```
False
False
True
True
```

Sequence data types are also a very basic and common data type, which we'll look at next.

Sequences

Sequence data types are used to store multiple values in a single variable in an organized and efficient way. There are four basic sequence types: string, range, lists, and tuples.

Strings

A string is an immutable sequence of characters represented in single, double, or triple quotes.



Immutable means that once a data type has been assigned some value, it can't be changed.

The string type in Python is called `str`. A triple quote string can span into multiple lines that include all the whitespace in the string. For example:

```
str1 = 'Hello how are you'
str2 = "Hello how are you"
str3 = """multiline
String"""
print(str1)
print(str2)
print(str3)
```

The output will be as follows:

```
Hello how are you
Hello how are you
multiline
String
```

The `+` operator concatenates strings, which returns a string after concatenating the operands, joining them together. For example:

```
f = 'data'
s = 'structure'
print(f + s)
print('Data ' + 'structure')
```

The output will be as follows:

```
datastructure
Data structure
```

The `*` operator can be used to create multiple copies of a string. When it is applied with an integer (n , let's say) and a string, the `*` operator returns a string consisting of n concatenated copies of the string. For example:

```
st = 'data.'
print(st * 3)
print(3 * st)
```

The output will be as follows.

```
data.data.data.
data.data.data.
```

Range

The `range` data type represents an immutable sequence of numbers. It is mainly used in `for` and `while` loops. It returns a sequence of numbers starting from a given number up to a number specified by the function argument. It is used as in the following command:

```
range(start, stop, step)
```

Here, the `start` argument specifies the start of the sequence, the `stop` argument specifies the end limit of the sequence, and the `step` argument specifies how the sequence should increase or decrease. This example Python code demonstrates the working of the range function:

```
print(list(range(10)))
print(range(10))
print(list(range(10)))
print(range(1,10,2))
print(list(range(1,10,2)))
print(list(range(20,10,-2)))
```

The output will be as follows.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(1, 10, 2)
[1, 3, 5, 7, 9]
[20, 18, 16, 14, 12]
```

Lists

Python lists are used to store multiple items in a single variable. Duplicate values are allowed in a list, and elements can be of different types: for example, you can have both numeric and string data in a Python list.

The items stored in the list are enclosed within square brackets, `[]`, and separated with a comma, as shown below:

```
a = ['food', 'bus', 'apple', 'queen']
print(a)
mylist = [10, "India", "world", 8]
# accessing elements in list.
print(mylist[1])
```

The output of the above code will be as follows.

```
['food', 'bus', 'apple', 'queen']
India
```

The data element of the list is shown in *Figure 1.4*, showing the index value of each of the list items:

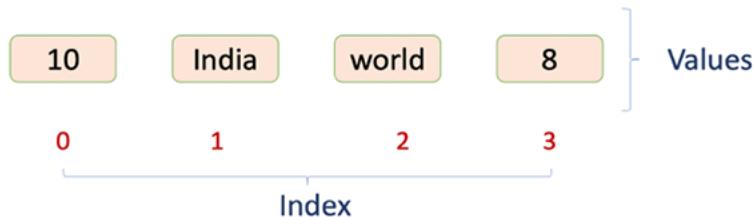


Figure 1.4: Data elements of a sample list

The characteristics of a list in Python are as follows. Firstly, the list elements can be accessed by its index, as shown in *Figure 1.4*. The list elements are ordered and dynamic. It can contain any arbitrary objects that are so desired. In addition, the `list` data structure is mutable, whereas most of the other data types, such as `integer` and `float` are immutable.



Seeing as a list is a mutable data type, once created, the list elements can be added, deleted, shifted, and moved within the list.

All the properties of lists are explained in *Table 1.1* below for greater clarity:

Property	Description	Example
Ordered	The list elements are ordered in a sequence in which they are specified in the list at the time of defining them. This order does not need to change and remains innate for its lifetime.	<pre>[10, 12, 31, 14] == [14, 10, 31, 12]</pre> <pre>False</pre>
Dynamic	The list is dynamic. It can grow or shrink as needed by	<pre>b = ['data', 'and', 'book', 'structure', 'hello', 'st']</pre> <pre>b += [32]</pre> <pre>print(b)</pre> <pre>b[2:3] = []</pre> <pre>print(b)</pre> <pre>del b[0]</pre> <pre>print(b)</pre>

	adding or removing list items.	<pre>['data', 'and', 'book', 'structure', 'hello', 'st', 32] ['data', 'and', 'structure', 'hello', 'st', 32] ['and', 'structure', 'hello', 'st', 32]</pre>
List elements can be any arbitrary set of objects	List elements can be of the same type or varying data types.	<pre>a = [2.2, 'python', 31, 14, 'data', False, 33.59] print(a)</pre> <pre>[2.2, 'python', 31, 14, 'data', False, 33.59]</pre>
List elements can be accessed through an index	<p>Elements can be accessed using zero-based indexing in square brackets, similar to a string.</p> <p>Accessing elements in a list is similar to strings; negative list indexing also works in lists.</p> <p>A negative list index counts from the end of the list.</p> <p>Lists also support slicing. If <code>abc</code> is a list, the expression <code>abc[x:y]</code> will return the portion of</p>	<pre>a = ['data', 'structures', 'using', 'python', 'happy', 'learning'] print(a[0]) print(a[2]) print(a[-1]) print(a[-5]) print(a[1:5]) print(a[-3:-1])</pre> <pre>data using learning structures ['structures', 'using', 'python', 'happy'] ['python', 'happy']</pre>

	elements from index <code>x</code> to index <code>y</code> (not including index <code>y</code>)	
Mutable	Single list value: Elements in a list can be updated through indexing and simple assignment. Modifying multiple list values is also possible through slicing.	<pre>a = ['data', 'and', 'book', 'structure', 'hello', 'st'] print(a) a[1] = 1 a[-1] = 120 print(a) a = ['data', 'and', 'book', 'structure', 'hello', 'st'] print(a[2:5]) a[2:5] = [1, 2, 3, 4, 5] print(a)</pre>
		<pre>['data', 'and', 'book', 'structure', 'hello', 'st'] ['data', 1, 'book', 'structure', 'hello', 120] ['book', 'structure', 'hello'] ['data', 'and', 1, 2, 3, 4, 5, 'st']</pre>
Other operators	Several operators and built-in functions can also be applied in lists, such as <code>in</code> , <code>not in</code> , concatenation (<code>+</code>), and replication (<code>*</code>) operators. Moreover, other built-in	<pre>a = ['data', 'structures', 'using', 'python', 'happy', 'learning'] print('data' in a) print(a) print(a + ['New', 'elements']) print(a) print(a *2) print(len(a)) print(min(a))</pre> <pre>['data', 'structures', 'using', 'python', 'happy', 'learning'] ['data', 'structures', 'using', 'python', 'happy', 'learning', 'New', 'elements'] ['data', 'structures', 'using', 'python', 'happy', 'learning'] ['data', 'structures', 'using', 'python', 'happy', 'learning', 'data', 'structures'] 6 data</pre>

	functions, such as <code>len()</code> , <code>min()</code> , and <code>max()</code> , are also available.
--	---

Table 1.1: Characteristics of list data structures with examples

Now, while discussing list data types, we should first understand different operators, such as membership, identity, and logical operators, before discussing them and how they can be used in list data types or any other data types. In the coming section, we discuss how these operators work and are used in various data types.

Membership, identity, and logical operations

Python supports membership, identity, and logical operators. Several data types in Python support them. In order to understand how these operators work, we'll discuss each of these operations in this section.

Membership operators

These operators are used to validate the membership of an item. Membership means we wish to test if a given value is stored in the sequence variable, such as a string, list, or tuple. Membership operators are to test for membership in a sequence; that is, a string, list, or tuple. Two common membership operators used in Python are `in` and `not in`.

The `in` operator is used to check whether a value exists in a sequence. It returns `True` if it finds the given variable in the specified sequence, and `False` if it does not:

```
# Python program to check if an item (say second
# item in the below example) of a List is present
# in another list (or not) using 'in' operator
mylist1 = [100, 20, 30, 40]
mylist2 = [10, 50, 60, 90]
if mylist1[1] in mylist2:
    print("elements are overlapping")
else:
    print("elements are not overlapping")
```

The output will be as follows:

```
elements are not overlapping
```

The '`not in`' operator returns to `True` if it does not find a variable in the specified sequence and `False` if it is found:

```
val = 104
mylist = [100, 210, 430, 840, 108]
if val not in mylist:
```

```
    print("Value is NOT present in mylist")
else:
    print("Value is present in mylist")
```

The output will be as follows:

```
Value is NOT present in mylist
```

Identity operators

Identity operators are used to compare objects. The different types of identity operators are `is` and `is not`, which are defined as follows.

The `is` operator is used to check whether two variables refer to the same object. This is different from the equality (`==`) operator. In the equality operator, we check whether two variables are equal. It returns `True` if both side variables point to the same object; if not, then it returns `False`:

```
Firstlist = []
Secondlist = []
if Firstlist == Secondlist:
    print("Both are equal")
else:
    print("Both are not equal")
if Firstlist is Secondlist:
    print("Both variables are pointing to the same object")
else:
    print("Both variables are not pointing to the same object")
thirdList = Firstlist
if thirdList is Secondlist:
    print("Both are pointing to the same object")
else:
    print("Both are not pointing to the same object")
```

The output will be as follows:

```
Both are equal
Both variables are not pointing to the same object
Both are not pointing to the same object
```

The `is not` operator is used to check whether two variables point to the same object or not. `True` is returned if both side variables point to different objects, otherwise, it returns `False`:

```
Firstlist = []
Secondlist = []
if Firstlist is not Secondlist:
    print("Both Firstlist and Secondlist variables are the same object")
else:
    print("Both Firstlist and Secondlist variables are not the same object")
```

The output will be as follows:

```
Both Firstlist and Secondlist variables are not the same object
```

This section was about identity operators. Next, let us discuss logical operators.

Logical operators

These operators are used to combine conditional statements (`True` or `False`). There are three types of logical operators: `AND`, `OR`, and `NOT`.

The logical `AND` operator returns True if both the statements are true, otherwise it returns False. It uses the following syntax: A and B:

```
a = 32
b = 132
if a > 0 and b > 0:
    print("Both a and b are greater than zero")
else:
    print("At least one variable is less than 0")
```

The output will be as follows.

```
Both a and b are greater than zero
```

The logical `OR` operator returns True if any of the statements are true, otherwise it returns False. It uses the following syntax: A or B:

```
a = 32
b = -32
if a > 0 or b > 0:
    print("At least one variable is greater than zero")
else:
    print("Both variables are less than 0")
```

The output will be as follows.

```
At least one variable is greater than zero
```

The logical `NOT` operator is a Boolean operator, which can be applied to any object. It returns `True` if the object/operand is false, otherwise it returns `False`. Here, the operand is the unary expression/statement on which the operator is applied. It uses the following syntax: `not A`:

```
a = 32
if not a:
    print("Boolean value of a is False")
else:
    print("Boolean value of a is True")
```

The output will be as follows.

```
Boolean value of a is True
```

In this section, we learned about different operators available in Python, and also saw how membership and identity operators can be applied to list data types. In the next section, we will continue discussing a final sequence data type: tuples.

Tuples

Tuples are used to store multiple items in a single variable. It is a read-only collection where data is ordered (zero-based indexing) and unchangeable/imutable (items cannot be added, modified, removed). Duplicate values are allowed in a tuple, and elements can be of different types, similar to lists. Tuples are used instead of lists when we wish to store the data that should not be changed in the program.

Tuples are written with round brackets and items are separated by a comma:

```
tuple_name = ("entry1", "entry2", "entry3")
```

For example:

```
my_tuple = ("Shyam", 23, True, "male")
```

Tuples support `+` (concatenation) and `*` (repetition) operations, similar to strings in Python. In addition, a membership operator and iteration operation are also available in a tuple. Different operations that tuples support are listed in *Table 1.2*:

Expression	Result	Description
<code>print(len((4,5, "hello")))</code>	3	Length
<code>print((4,5)+(10,20))</code>	(4,5,10,20)	Concatenation
<code>print((2,1)*3)</code>	(2,1,2,1,2,1)	Repetition
<code>print(3 in ('hi', 'xyz',3))</code>	True	Membership
<code>for p in (6,7,8): print(p)</code>	6,7,8	Iteration

Table 1.2: Example of tuple operations

Tuples in Python support zero-based indexing, negative indexing, and slicing. To understand it, let's take a sample tuple, as shown below:

```
x = ("hello", "world", "india")
```

We can see examples of zero-based indexing, negative indexing, and slicing operations in *Table 1.3*:

Expression	Result	Description
<code>print(x[1])</code>	<code>"world"</code>	Zero-based indexing means that indexing starts from 0 rather than 1, and hence in this example, the first index refers to the second member of the tuple.
<code>print(x[-2])</code>	<code>"world"</code>	Negative: counting from the right-hand side.
<code>print(x[1:])</code>	<code>("world", "india")</code>	Slicing fetches a section.

Table 1.3: Example of tuple indexing and slicing

Complex data types

We have discussed basic data types. Next, we discuss complex data types, which are mapping data types, in other words, dictionary, and set data types, namely, set and frozenset. We will discuss these data types in detail in this section.

Dictionaries

In Python, a dictionary is another of the important data types, similar to a list, in the sense that it is also a collection of objects. It stores the data in unordered {key-value} pairs; a key must be of a hashable and immutable data type, and value can be any arbitrary Python object. In this context, an object is hashable if it has a hash value that does not change during its lifetime in the program.

Items in the dictionary are enclosed in curly braces, {}, separated by a comma, and can be created using the {key:value} syntax, as shown below:

```
dict = {
    <key>: <value>,
    <key>: <value>,
    .
    .
    .
    <key>: <value>
}
```

Keys in dictionaries are case-sensitive, they should be unique, and cannot be duplicated; however, the values in the dictionary can be duplicated. For example, the following code can be used to create a dictionary:

```
my_dict = {'1': 'data',
           '2': 'structure',
           '3': 'python',
           '4': 'programming',
           '5': 'language'
         }
```

Figure 1.5 shows the {key-value} pairs created by the preceding piece of code:

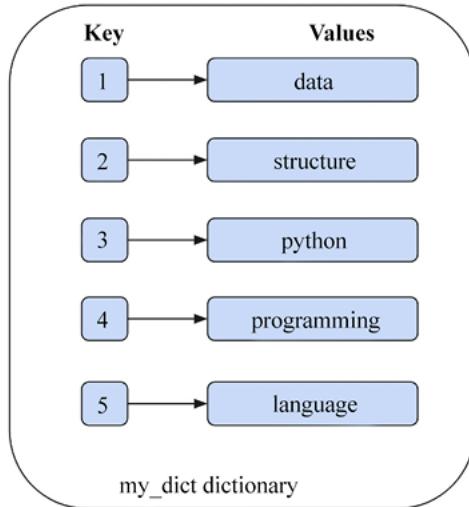


Figure 1.5: Example dictionary data structure

Values in a dictionary can be fetched based on the key. For example: `my_dict['1']` gives `data` as the output.

The `dictionary` data type is mutable and dynamic. It differs from lists in the sense that dictionary elements can be accessed using keys, whereas the list elements are accessed via indexing. *Table 1.4* shows different characteristics of the dictionary data structure with examples:

Item	Example
Creating a dictionary, and accessing elements from a dictionary	<pre>person = {} print(type(person)) person['name'] = 'ABC' person['lastname'] = 'XYZ' person['age'] = 31 person['address'] = ['Jaipur'] print(person) print(person['name'])</pre> <div style="background-color: black; color: white; padding: 5px; margin-top: 10px;"> <class 'dict'>{'name': 'ABC', 'lastname': 'XYZ', 'age': 31, 'address': ['Jaipur']}ABC </div>

<code>in</code> and <code>not in</code> operators	<pre>print('name' in person) print('fname' not in person)</pre> <p>True True</p>
Length of the dictionary	<pre>print(len(person))</pre> <p>4</p>

Table 1.4: Characteristics of dictionary data structures with examples

Python also includes the dictionary methods as shown in *Table 1.5*:

Function	Description	Example
<code>mydict.clear()</code>	Removes all elements from a dictionary.	<pre>mydict = {'a': 1, 'b': 2, 'c': 3} print(mydict) mydict.clear() print(mydict)</pre> <p>{'a': 1, 'b': 2, 'c': 3} {}</p>
<code>mydict.get(<key>)</code>	Searches the dictionary for a key and returns the corresponding value, if it is found; otherwise, it returns <code>None</code> .	<pre>mydict = {'a': 1, 'b': 2, 'c': 3} print(mydict.get('b')) print(mydict) print(mydict.get('z'))</pre> <p>2 {'a': 1, 'b': 2, 'c': 3} None</p>
<code>mydict.items()</code>	Returns a list of dictionary items in (key, value) pairs.	<pre>print(list(mydict.items()))</pre> <p>[('a', 1), ('b', 2), ('c', 3)]</p>
<code>mydict.keys()</code>	Returns a list of dictionary keys.	<pre>print(list(mydict.keys()))</pre>

<code>ys()</code>		<code>['a', 'b', 'c']</code>
<code>mydict.va lues()</code>	Returns a list of dictionary values.	<code>print(list(mydict.values()))</code> <code>[1, 2, 3]</code>
<code>mydict.po p()</code>	If a given key is present in the dictionary, then this function will remove the key and return the associated value.	<code>print(mydict.pop('b'))</code> <code>print(mydict)</code> <code>{'a': 1, 'c': 3}</code>
<code>mydict.po pitem()</code>	This method removes the last key-value pair added in the dictionary and returns it as a tuple.	<code>mydict = {'a': 1, 'b': 2, 'c': 3}</code> <code>print(mydict.popitem())</code> <code>print(mydict)</code> <code>('a': 1, 'b': 2)</code>
<code>mydict.up date(<obj >)</code>	Merges one dictionary with another. Firstly, it checks whether a key of the second dictionary is present in the first dictionary; the corresponding value is then updated. If the key is not present in the first dictionary, then the key-value pair is added.	<code>d1 = {'a': 10, 'b': 20, 'c': 30}</code> <code>d2 = {'b': 200, 'd': 400}</code> <code>print(d1.update(d2))</code> <code>print(d1)</code> <code>{'a': 10, 'b': 200, 'c': 30, 'd': 400}</code>

Table 1.5: List of methods of dictionary data structures

Sets

A set is an unordered collection of hashable objects. It is iterable, mutable, and has unique elements. The order of the elements is also not defined. While the addition and removal of items are allowed, the items themselves within the set must be immutable and hashable. Sets support membership testing operators (`in`, `not in`), and operations such as intersection, union, difference, and symmetric difference. Sets cannot contain duplicate items. They are created by using the built-in `set()` function or curly braces `{}`. A `set()` returns a set object from an iterable. For example:

```
x1 = set(['and', 'python', 'data', 'structure'])
print(x1)
print(type(x1))
x2 = {'and', 'python', 'data', 'structure'}
print(x2)
```

The output will be as follows:

```
{'python', 'structure', 'data', 'and'}  
<class 'set'>  
{'python', 'structure', 'data', 'and'}
```



It is important to note that sets are unordered data structures, and the order of items in sets is not preserved. Therefore, your outputs in this section may be slightly different than those displayed here. However, this does not affect the function of the operations we will be demonstrating in this section.

Sets are generally used to perform mathematical operations, such as intersection, union, difference, and complement. The `len()` method gives the number of items in a set, and the `in` and `not in` operators can be used in sets to test for membership:

```
x = {'data', 'structure', 'and', 'python'}  
print(len(x))  
print('structure' in x)
```

The output will be as follows:

```
4  
True
```

The most commonly used methods and operations that can be applied to `set` data structures are as follows. The union of the two sets, say, `x1` and `x2`, is a set that consists of all elements in either set:

```
x1 = {'data', 'structure'}  
x2 = {'python', 'java', 'c', 'data'}
```

Figure 1.6 shows a Venn diagram demonstrating the relationship between the two sets:

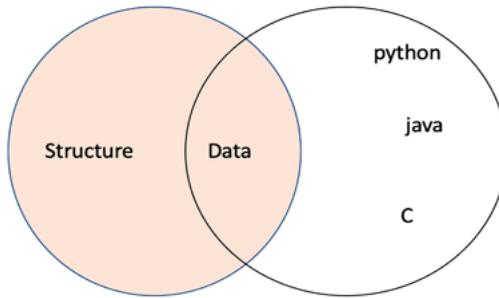


Figure 1.6: Venn diagram of sets

A description of the various operations that can be applied on set type variables is shown, with examples, in *Table 1.6*:

Description	Example sample code
Union of two sets, <code>x1</code> and <code>x2</code> . It can be done using two methods, (1) using the <code> </code> operator, (2) using the <code>union</code> method.	<pre>x1 = {'data', 'structure'} x2 = {'python', 'java', 'c', 'data'} x3 = x1 x2 print(x3) print(x1.union(x2))</pre> <div style="background-color: black; color: white; padding: 10px;"> {'structure', 'data', 'java', 'c', 'python'} {'structure', 'data', 'java', 'c', 'python'} </div>
Intersection of sets: to compute the intersection of two sets, an <code>&</code> operator and the <code>intersection()</code> method can be used, which returns a set of items common to both sets, <code>x1</code> and <code>x2</code> .	<pre>print(x1.intersection(x2)) print(x1 & x2)</pre> <div style="background-color: black; color: white; padding: 10px;"> {'data'} {'data'} </div>
The difference between sets can be obtained using <code>.difference()</code> and the subtraction operator, <code>-</code> , which returns a set of all elements that are in <code>x1</code> , but not in <code>x2</code> .	<pre>print(x1.difference(x2)) print(x1 - x2)</pre> <div style="background-color: black; color: white; padding: 10px;"> {'structure'} {'structure'} </div>
Symmetric difference can be obtained using <code>.symmetric_difference()</code> , while <code>^</code> returns a set of all data items that are present in either <code>x1</code> or <code>x2</code> , but not both.	<pre>print(x1.symmetric_difference(x2)) print(x1 ^ x2)</pre> <div style="background-color: black; color: white; padding: 10px;"> {'structure', 'python', 'c', 'java'} {'structure', 'python', 'c', 'java'} </div>
To test whether a set is a subset of another, use <code>.issubset()</code> and the operator <code><=</code> .	<pre>print(x1.issubset(x2)) print(x1 <= x2)</pre> <div style="background-color: black; color: white; padding: 10px;"> False False </div>

Table 1.6: Description of various operations applicable to set type variables

Immutable sets

In Python, `frozenset` is another built-in type data structure, which is, in all respects, exactly like a set, except that it is immutable, and so cannot be changed after creation. The order of the elements is also

undefined. A `frozenset` is created by using the built-in function `frozenset()`:

```
x = frozenset(['data', 'structure', 'and', 'python'])
print(x)
```

The output is:

```
frozenset({'python', 'structure', 'data', 'and'})
```

Frozensets are useful when we want to use a set but require the use of an immutable object. Moreover, it is not possible to use set elements in the set, since they must also be immutable. Consider an example:

```
a11 = set(['data'])
a21 = set(['structure'])
a31 = set(['python'])
x1 = {a11, a21, a31}
```

The output will be:

```
TypeError: unhashable type: 'set'
```

Now with `frozenset`:

```
a1 = frozenset(['data'])
a2 = frozenset(['structure'])
a3 = frozenset(['python'])
x = {a1, a2, a3}
print(x)
```

The output is:

```
{frozenset({'structure'}), frozenset({'python'}), frozenset({'data'})}
```

In the above example, we create a set `x` of frozensets (`a1`, `a2`, and `a3`), which is possible because the frozensets are immutable.

We have discussed the most important and popular data types available in Python. Python also provides a collection of other important methods and modules, which we will discuss in the next section.

Python's collections module

The `collections` module provides different types of containers, which are objects that are used to store different objects and provide a way to access them. Before accessing these, let's consider briefly the role and relationships between modules, packages, and scripts.

A module is a Python script with the `.py` extension that contains a collection of functions, classes, and variables. A package is a directory that contains collections of modules; it has an `__init__.py` file, which

lets the interpreter know that it is a package. A module can be called into a Python script, which can in turn make use of the module's functions and variables in its code. In Python, we can import these to a script using the `import` statement. Whenever the interpreter encounters the `import` statement, it imports the code of the specified module.

Table 1.7 provides the data types and operations of the collections module and their descriptions:

Container data type	Description
<code>namedtuple</code>	Creates a <code>tuple</code> with named fields similar to regular tuples.
<code>deque</code>	Doubly-linked lists that provide efficient adding and removing of items from both ends of the list.
<code>defaultdict</code>	A <code>dictionary</code> subclass that returns default values for missing keys.
<code>ChainMap</code>	A <code>dictionary</code> that merges multiple dictionaries.
<code>Counter</code>	A <code>dictionary</code> that returns the counts corresponding to their objects/key.
<code>UserDict</code> <code>UserList</code> <code>UserString</code>	These data types are used to add more functionalities to their base data structure, such as a <code>dictionary</code> , <code>list</code> , and <code>string</code> . And we can create subclasses from them for custom <code>dict/list/string</code> .

Table 1.7: Different container data type of the collections module

Let's consider these types in more detail.

Named tuples

The `namedtuple` of `collections` provides an extension of the built-in tuple data type. `namedtuple` objects are immutable, similar to standard tuples. Thus, we can't add new fields or modify existing ones after the `namedtuple` instance is created. They contain keys that are mapped to a particular value, and we can iterate through named tuples either by index or key. The `namedtuple` function is mainly useful when several tuples are used in an application and it is important to keep track of each of the tuples in terms of what they represent.

In this situation, `namedtuple` presents a more readable and self-documenting method. The syntax is as follows:

```
nt = namedtuple(typename , field_names)
```

Here is an example:

```

from collections import namedtuple
Book = namedtuple ('Book', ['name', 'ISBN', 'quantity'])
Book1 = Book('Hands on Data Structures', '9781788995573', '50')
#Accessing data items
print('Using index ISBN:' + Book1[1])
print('Using key ISBN:' + Book1.ISBN)

```

The output will be as follows.

```

Using index ISBN:9781788995573
Using key ISBN:9781788995573

```

Here, in the above code, we firstly imported `namedtuple` from the `collections` module. `Book` is a named tuples, “`class`,” and then, `Book1` is created, which is an instance of `Book`. We also see that the data elements can be accessed using index and key methods.

Deque

A `deque` is a double-ended queue (deque) that supports append and pop elements from both sides of the list. Deques are implemented as double-linked lists, which are very efficient for inserting and deleting elements in O(1) time complexity.

Consider an example:

```

from collections import deque
s = deque() # Creates an empty deque
print(s)
my_queue = deque([1, 2, 'Name'])
print(my_queue)

```

The output will be as follows.

```

deque([])
deque([1, 2, 'Name'])

```

You can also use some of the following predefined functions:

Function	Description
<code>my_queue.append('age')</code>	Insert <code>'age'</code> at the right end of the list.
<code>my_queue.appendleft('age')</code>	Insert <code>'age'</code> at the left end of the list.
<code>my_queue.pop()</code>	Delete the rightmost value.
<code>my_queue.popleft()</code>	Delete the leftmost value.

Table 1.8: Description of different queue functions

In this section, we showed the use of the `deque` method of the `collections` module, and how elements can be added and deleted from the `queue`.

Ordered dictionaries

An ordered dictionary is a dictionary that preserves the order of the keys that are inserted. If the key order is important for any application, then `OrderedDict` can be used:

```
od = OrderedDict([items])
```

An example could look like the following:

```
from collections import OrderedDict
od = OrderedDict({'my': 2, 'name': 4, 'is': 2, 'Mohan': 5})
od['hello'] = 4
print(od)
```

The output will be as follows.

```
OrderedDict([('my', 2), ('name', 4), ('is', 2), ('Mohan', 5), ('hello', 4)])
```

In the above code, we create a dictionary, `od`, using the `OrderedDict` module. We can observe that the order of the keys is the same as the order when we created the key.

Default dictionary

The default dictionary (`defaultdict`) is a subclass of the built-in dictionary class (`dict`) that has the same methods and operations as that of the `dictionary` class, with the only difference being that it never raises a `KeyError`, as a normal dictionary would. `defaultdict` is a convenient way to initialize dictionaries:

```
d = defaultdict(def_value)
```

An example could look like the following:

```
from collections import defaultdict
dd = defaultdict(int)
words = str.split('data python data data structure data python')
for word in words:
    dd[word] += 1
print(dd)
```

The output will be as follows.

```
defaultdict(<class 'int'>, {'data': 4, 'python': 2, 'structure': 1})
```

In the above example, if an ordinary dictionary had been used, then Python would have shown `KeyError` while the first key was added. `int`, which we supplied as an argument to `defaultdict`, is really the `int()` function, which simply returns a zero.

ChainMap object

`ChainMap` is used to create a list of dictionaries. The `collections.ChainMap` data structure combines several dictionaries into a single mapping. Whenever a key is searched in the `chainmap`, it looks through all the dictionaries one by one, until the key is not found:

```
class collections.ChainMap(dict1, dict2)
```

An example could look like the following:

```
from collections import ChainMap
dict1 = {"data": 1, "structure": 2}
dict2 = {"python": 3, "language": 4}
chain = ChainMap(dict1, dict2)
print(chain)
print(list(chain.keys()))
print(list(chain.values()))
print(chain["data"])
print(chain["language"])
```

The output will be:

```
ChainMap({'data': 1, 'structure': 2}, {'python': 3, 'language': 4})
['python', 'language', 'data', 'structure']
[3, 4, 1, 2]
1
4
```

In the above code, we create two dictionaries, namely, `dict1` and `dict2`, and then we can combine both of these dictionaries using the `ChainMap` method.

Counter objects

As we discussed earlier, a hashable object is one whose hash value will remain the same during its lifetime in the program. `counter` is used to count the number of hashable objects. Here, the dictionary key is a hashable object, while the corresponding value is the count of that object. In other words, `counter` objects create a hash table in which the elements and their count are stored as dictionary keys and value pairs.

`Dictionary` and `counter` objects are similar in the sense that data is stored in a `{key, value}` pair, but in `counter` objects, the value is the count of the key whereas it can be anything in the case of `dictionary`. Thus, when we only want to see how many times each unique word is occurring in a string, we use the `counter` object.

An example could look like the following:

```
from collections import Counter
inventory = Counter('hello')
print(inventory)
print(inventory['l'])
print(inventory['e'])
print(inventory['o'])
```

The output will be:

```
Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
2
1
1
```

In the above code, the `inventory` variable is created, which holds the counts of all the characters using the `counter` module. The count values of these characters can be accessed using dictionary-like key access (`[key]`).

UserDict

Python supports a container, `UserDict`, present in the `collections` module, that wraps the dictionary objects. We can add customized functions to the dictionary. This is very useful for applications where we want to add/update/modify the functionalities of the dictionary. Consider the example code below where pushing/adding a new data element is not allowed in the dictionary:

```
# we can not push to this user dictionary
from collections import UserDict
class MyDict(UserDict):
    def push(self, key, value):
        raise RuntimeError("Cannot insert")
d = MyDict({'ab':1, 'bc': 2, 'cd': 3})
d.push('b', 2)
```

The output is as follows:

```
RuntimeError: Cannot insert
```

In the above code, a customized `push` function in the `MyDict` class is created to add the customized functionality, which does not allow you to insert an element into the dictionary.

UserList

A `UserList` is a container that wraps list objects. It can be used to extend the functionality of the `list` data structure. Consider the example code below, where pushing/adding a new data element is not allowed in the `list` data structure:

```
# we can not push to this user list
from collections import UserList
class MyList(UserList):
```

```
def push(self, key):
    raise RuntimeError("Cannot insert in the list")
d = MyList([11, 12, 13])
d.push(2)
```

The output is as follows:

```
RuntimeError: Cannot insert in the list
```

In the above code, a customized `push` function in the `MyList` class is created to add the functionality to not allow you to insert an element into the `list` variable.

UserString

Strings can be considered as an array of characters. In Python, a character is a string of one length and acts as a container that wraps a string object. It can be used to create strings with customized functionalities. An example could look like the following:

```
#Create a custom append function for string
from collections import UserString
class MyString(UserString):
    def append(self, value):
        self.data += value
s1 = MyString("data")
print("Original:", s1)
s1.append('h')
print("After append: ", s1)
```

The output is:

```
Original: data
After append:  datah
```

In the above example code, a customized append function in the `MyString` class is created to add the functionality to append a string.

Summary

In this chapter, we have discussed different built-in data types supported by Python. We have also looked at a few basic Python functions, libraries, and modules, such as the `collections` module. The main objective of this chapter was to give an overview of Python and make a user acquainted with the language so that it is easy to implement the advanced algorithms of data structures.

Overall, this chapter has provided an overview of several data structures available in Python that are pivotal for understanding the internals of data structures. In the next chapter, we will introduce the basic concepts of algorithm design and analysis.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/MEvK4>



2

Introduction to Algorithm Design

The objective of this chapter is to understand the principles of designing algorithms, and the importance of analyzing algorithms in solving real-world problems. Given input data, an algorithm is a step-by-step set of instructions that should be executed in sequence to solve a given problem.

In this chapter, we will also learn how to compare different algorithms and determine the best algorithm for the given use-case. There can be many possible correct solutions for a given problem, for example, we can have several algorithms for the problem of sorting n numeric values. So, there is no one algorithm to solve any real-world problem.

In this chapter, we will look at the following topics:

- Introducing algorithms
- Performance analysis of an algorithm
- Asymptotic notation
- Amortized analysis
- Choosing complexity classes
- Computing the running time complexity of an algorithm

Introducing algorithms

An algorithm is a sequence of steps that should be followed in order to complete a given task/problem.

It is a well-defined procedure that takes input data, processes it, and produces the desired output. A representation of this is shown in *Figure 2.1*.

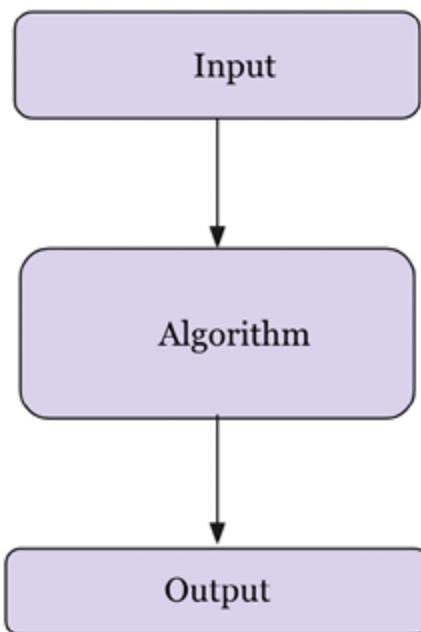


Figure 2.1: Introduction to algorithms

Summarized below are some important reasons for studying algorithms:

- Essential for computer science and engineering
- Important in many other domains (such as computational biology, economics, ecology, communications, physics, and so on)
- They play a role in technology innovation

- They improve problem-solving and analytical thinking

There are two aspects that are of prime importance in solving a given problem. Firstly, we need an efficient mechanism to store, manage, and retrieve data, which is required to solve a problem (this comes under data structures); secondly, we require an efficient algorithm that is a finite set of instructions to solve that problem. Thus, the study of data structures and algorithms is key to solving any problem using computer programs. An efficient algorithm should have the following characteristics:

- It should be as specific as possible
- It should have each instruction properly defined
- There should not be any ambiguous instructions
- All the instructions of the algorithm should be executable in a finite amount of time and in a finite number of steps
- It should have clear input and output to solve the problem
- Each instruction of the algorithm should be integral in solving the given problem

Consider an example of an algorithm (an analogy) to complete a task in our daily lives; let us take the example of preparing a cup of tea. The algorithm to prepare a cup of tea can include the following steps:

1. Pour water into the pan
2. Put the pan on the stove and light the stove
3. Add crushed ginger to the warming water
4. Add tea leaves to the pan
5. Add milk

6. When it starts boiling, add sugar to it
7. After 2-3 minutes, the tea can be served

The above procedure is one of the possible ways to prepare tea. In the same way, the solution to a real-world problem can be converted into an algorithm, which can be developed into computer software using a programming language. Since it is possible to have several solutions for a given problem, it should be as efficient as possible when it is to be implemented using software. Given a problem, there may be more than one correct algorithm, defined as the one that produces exactly the desired output for all valid input values. The costs of executing different algorithms may be different; it may be measured in terms of the time required to run the algorithm on a computer system and the memory space required for it.

There are primarily two things that one should keep in mind while designing an efficient algorithm:

1. The algorithm should be correct and should produce the results as expected for all valid input values
2. The algorithm should be optimal in the sense that it should be executed on the computer within the desired time limit, in line with an optimal memory space requirement

Performance analysis of the algorithm is very important for deciding the best solution for a given problem. If the performance of an algorithm is within the desired time and space requirements, it is optimal. One of the most popular and common methods of estimating the performance of an algorithm is through analyzing its complexity. Analysis of the algorithm helps us to determine which one is most efficient in terms of the time and space consumed.

Performance analysis of an algorithm

The performance of an algorithm is generally measured by the size of its input data, n , and the time and the memory space used by the algorithm. The time required is measured by the key operations to be performed by the algorithm (such as comparison operations), where key operations are instructions that take a significant amount of time during execution. Whereas the space requirement of an algorithm is measured by the memory needed to store the variables, constants, and instructions during the execution of the program.

Time complexity

The time complexity of the algorithm is the amount of time that an algorithm will take to execute on a computer system to produce the output. The aim of analyzing the time complexity of the algorithm is to determine, for a given problem and more than one algorithm, which one of the algorithms is the most efficient with respect to the time required to execute. The running time required by an algorithm depends on the input size; as the input size, n , increases, the runtime also increases. Input size is measured as the number of items in the input, for example, the input size for a sorting algorithm will be the number of items in the input. So, a sorting algorithm will have an increased runtime to sort a list of input size 5,000 than that of a list of input size 50.

The runtime of an algorithm for a specific input depends on the key operations to be executed in the algorithm. For example, the key

operation for a sorting algorithm is a comparison operation that will take up most of the runtime, compared to assignment or any other operation. Ideally, these key operations should not depend upon the hardware, the operating system, or the programming language being used to implement the algorithm.

A constant amount of time is required to execute each line of code; however, each line may take a different amount of time to execute. In order to understand the running time required for an algorithm, consider the below code as an example:

Code	Time required (Cost)
<pre>if n==0 n == 3 #constant time print("data") else: for i in range(#Loop run for n times print("structure")</pre>	c1 c2 c3 c4 c5

Here, in statement 1 of the above example, if the condition is true then "data" will be printed, and if the condition is not true then the `for` loop will execute `n` times. The time required by the algorithm depends on the time required for each statement, and how many times a statement is executed. The running time of the algorithm is the sum of time required by all the statements. For the above code, assume statement 1 takes `c1` amount of time, statement 2 takes `c2` amount of time, and so on. So, if the i^{th} statement takes a constant

amount of time c_i and if the i^{th} statement is executed n times, then it will take $c_i n$ time. The total running time $T(n)$ of the algorithm for a given value of n (assuming the value of n is not zero or three) will be as follows.

$$T(n) = c_1 + c_3 + c_4 \times n + c_5 \times n$$

If the value of n is equal to zero or three, then the time required by the algorithm will be as follows.

$$T(n) = c_1 + c_2$$

Therefore, the running time required for an algorithm also depends upon what input is given in addition to the size of the input given. For the given example, the best case will be when the input is either zero or three, and in that case, the running time of the algorithm will be constant. In the worst case, the value of n is not equal to zero or three, then, the running time of the algorithm can be represented as $a \times n + b$. Here, the values of a and b are constants that depend on the statement costs, and the constant times are not considered in the final time complexity. In the worst case, the runtime required by the algorithm is a linear function of n .

Let us consider another example, linear search:

```
def linear_search(input_list, element):
    for index, value in enumerate(input_list):
        if value == element:
            return index

    return -1
input_list = [3, 4, 1, 6, 14]
element = 4
print("Index position for the element x is:", linear_search(input_list))
```

The output in this instance will be as follows:

```
Index position for the element x is: 1
```

The **worst-case running time** of the algorithm is the upper-bound complexity; it is the maximum runtime required for an algorithm to execute for any given input. The worst-case time complexity is very useful in that it guarantees that for any input data, the runtime required will not take more time as compared to the worst-case running time. For example, in the linear search problem, the worst case occurs when the element to be searched is found in the last comparison or not found in the list. In this case, the running time required will linearly depend upon the length of the list, whereas, in the best case, the search element will be found in the first comparison.

The **average-case running time** is the average running time required for an algorithm to execute. In this analysis, we compute the average over the running time for all possible input values. Generally, probabilistic analysis is used to analyze the average-case running time of an algorithm, which is computed by averaging the cost over the distribution of all the possible inputs. For example, in the linear search, the number of comparisons at all positions would be 1 if the element to be searched was found at the 0th index; and similarly, the number of comparisons would be 2, 3, and so forth, up to n , respectively, for elements found at the $1, 2, 3, \dots (n-1)$ index positions. Thus, the average-case running time will be as follows.

$$T(n) = \frac{1 + 2 + 3 \dots n}{n} = \frac{n(n + 1)}{2n}$$

For average-case, the running time required is also linearly dependent upon the value of n . However, in most real-world applications, worst-case analysis is mostly used, since it gives a guarantee that the running time will not take any longer than the worst-case running time of the algorithm for any input value.

Best-case running time is the minimum time needed for an algorithm to run; it is the lower bound on the running time required for an algorithm; in the example above, the input data is organized in such a way that it takes its minimum running time to execute the given algorithm.

Space complexity

The space complexity of the algorithm estimates the memory requirement to execute it on a computer to produce the output as a function of input data. The memory space requirement of an algorithm is one of the criteria used to decide how efficient it is.

While executing the algorithm on the computer system, storage of the input is required, along with intermediate and temporary data in data structures, which are stored in the memory of the computer. In order to write a programming solution for any problem, some memory is required for storing variables, program instructions, and executing the program on the computer. The space complexity of an algorithm is the amount of memory required for executing and producing the result.

For computing the space complexity, consider the following example, in which, given a list of integer values, the function returns the square value of the corresponding integer number.

```
def squares(n):
    square_numbers = []
    for number in n:
        square_numbers.append(number * number)
    return square_numbers

nums = [2, 3, 5, 8]
print(squares(nums))
```

The output of the code is:

```
[4, 9, 25, 64]
```

In the above code, the algorithm will require allocating memory for the number of items in the input list. Say the number of elements in the input is n , then the space requirement increases with the input size, therefore, the space complexity of the algorithm becomes $O(n)$.

Given two algorithms to solve a given problem, with all other requirements being equal, then the algorithm that requires less memory can be considered more efficient. For example, suppose there are two search algorithms, one has $O(n)$ and another algorithm has $O(n \log n)$ space complexity. The first algorithm is the better algorithm as compared to the second with respect to the space requirements. Space complexity analysis is important to understand the efficiency of an algorithm, especially for applications where the memory space requirement is high.

When the input size becomes large enough, the order of growth also becomes important. In such situations, we study the asymptotic efficiency of algorithms. Generally, algorithms that are asymptotically efficient are considered to be better algorithms for large-size inputs. In the next section, we will study asymptotic notation.

Asymptotic notation

To analyze the time complexity of an algorithm, the rate of growth (order of growth) is very important when the input size is large. When the input size becomes large, we only consider the higher-order terms and ignore the insignificant terms. In asymptotic analysis, we analyze the efficiency of algorithms for large input sizes considering the higher order of growth and ignoring the multiplicative constants and lower-order terms.

We compare two algorithms with respect to input size rather than the actual runtime and measure how the time taken increases with an increased input size. The algorithm which is more efficient asymptotically is generally considered a better algorithm as compared to the other algorithm. The following asymptotic notations are commonly used to calculate the running time complexity of an algorithm:

- Θ notation: It denotes the worst-case running time complexity with a tight bound.
- O notation: It denotes the worst-case running time complexity with an upper bound, which ensures that the function never grows faster than the upper bound.

- Ω notation: It denotes the lower bound of an algorithm's running time. It measures the best amount of time to execute the algorithm.

Theta notation

The following function characterizes the worst-case running time for the first example discussed in the *Time complexity* section:

$$T(n) = c_1 + c_3 \times n + c_5 \times n$$

Here, for a large input size, the worst-case running time will be $\Theta(n)$ (pronounced as theta of n). We usually consider one algorithm to be more efficient than another if its worst-case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. For example, once the input size n becomes large enough, the merge sort algorithm performs better as compared to insertion sort with worst-case running times of $\Theta(\log n)$ and $\Theta(n^2)$ respectively.

Theta notation (Θ) denotes the worst-case running time for an algorithm with a tight bound. For a given function $F(n)$, the asymptotic worst-case running time complexity can be defined as follows.

$$T(n) = \Theta(F(n))$$

iff there exists constants n_0 , c_1 , and c_2 such that:

$$0 \leq c_1(F(n)) \leq T(n) \leq c_2(Fn) \text{ for all } n \geq n_0$$

The function $T(n)$ belongs to a set of functions $\Theta(F(n))$ if there exists positive constants c_1 and c_2 such that the value of $T(n)$ always lies in between $c_1F(n)$ and $c_2F(n)$ for all large values of n . If this condition is true, then we say $F(n)$ is asymptotically tight bound for $T(n)$.

Figure 2.2 shows the graphic example of the theta notation (Θ). It can be observed from the figure that the value of $T(n)$ always lies in between $c_1F(n)$ and $c_2F(n)$ for values of n greater than n_0 .

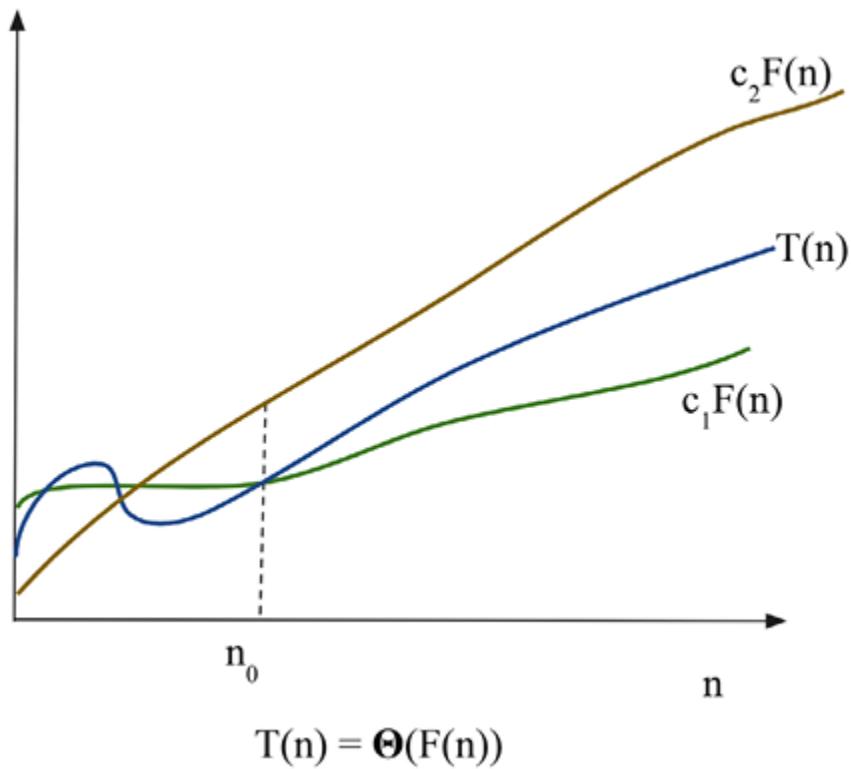


Figure 2.2: Graphical example of theta notation (Θ)

Let us consider an example to understand what should be the worst case running time complexity with the formal definition of theta notation for a given function:

$$f(n) = n^2 + n \text{ is } \Theta(n^2)$$

In order to determine the time complexity with the Θ notation definition, we have to first identify the constants c_1, c_2, n_0 such that

$$0 \leq c_1 * n^2 \leq n^2 + n \leq c_2 * n^2, \text{ for all } n \geq n_0$$

Dividing by n^2 will produce:

$$0 \leq c_1 \leq 1 + \frac{1}{n} \leq c_2, \text{ for all } n \geq n_0$$

By choosing $c_1 = 1$, $c_2 = 2$, and $n_0 = 1$, the following condition can satisfy the definition of theta notation.

$$0 \leq n^2 \leq n^2 + n \leq 2n^2, \text{ for all } n \geq 1$$

That gives:

$$f(n) = \Theta(g(n)), \text{ means } f(n) = \Theta(n^2)$$

Consider another example to find out the asymptotically tight bound (Θ) for another function:

$$f(n) = \frac{n^2}{2} + \frac{n}{2}$$

In order to identify the constants c_1, c_2 , and n_0 , such that they satisfy the condition:

$$0 \leq c_1 * n^2 \leq \frac{n^2}{2} \leq c_2 * n^2, \text{ for all } n \geq n_0$$

By choosing $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 1$, the following condition can satisfy the definition of theta notation:

$$0 \leq \frac{n^2}{5} \leq \frac{n^2}{2} + \frac{n}{2} \leq n^2 \text{ for all the values of } n \geq 1$$

$$\Rightarrow \frac{n^2}{2} + \frac{n}{2} = \Theta(n^2) \text{ with } c_1 = \frac{1}{5} \text{ and } c_2 = 1 \text{ and } n_0 = 1$$

So, the following is true:

$$f(n) = \frac{n^2}{2} + \frac{n}{2} = \Theta(n^2)$$

It shows that the given function has the complexity of $\Theta(n^2)$ as per the definition of theta notation.

So, the theta notation provides a tight bound for the time complexity of an algorithm. In the next section, we will discuss Big O notation.

Big O notation

We have seen that the theta notation is asymptotically bound from the upper and lower sides of the function whereas the Big O notation characterizes the worst-case running time complexity, which is only the asymptotic upper bound of the function. Big O notation is defined as follows. Given a function $F(n)$, the $T(n)$ is a Big O of function $F(n)$, and we define this as follows:

$$T(n) = O(F(n))$$

iff there exists constants n_0 and c such that:

$$T(n) \leq c(F(n)) \text{ for all } n \geq n_0$$

In Big O notation, a constant multiple of $F(n)$ is an asymptotic upper bound on $T(n)$, and the positive constants n_0 and c should be in such

a way that all values of n greater than n_0 always lie on or below function $c^*F(n)$.

Moreover, we only care what happens at higher values of n . The variable n_0 represents the threshold below which the rate of growth is not important. The plot shown in *Figure 2.3* shows a graphical representation of function $T(n)$ with a varying value of n . We can see that $T(n) = n^2 + 500 = O(n^2)$, with $c = 2$ and n_0 being approximately 23.

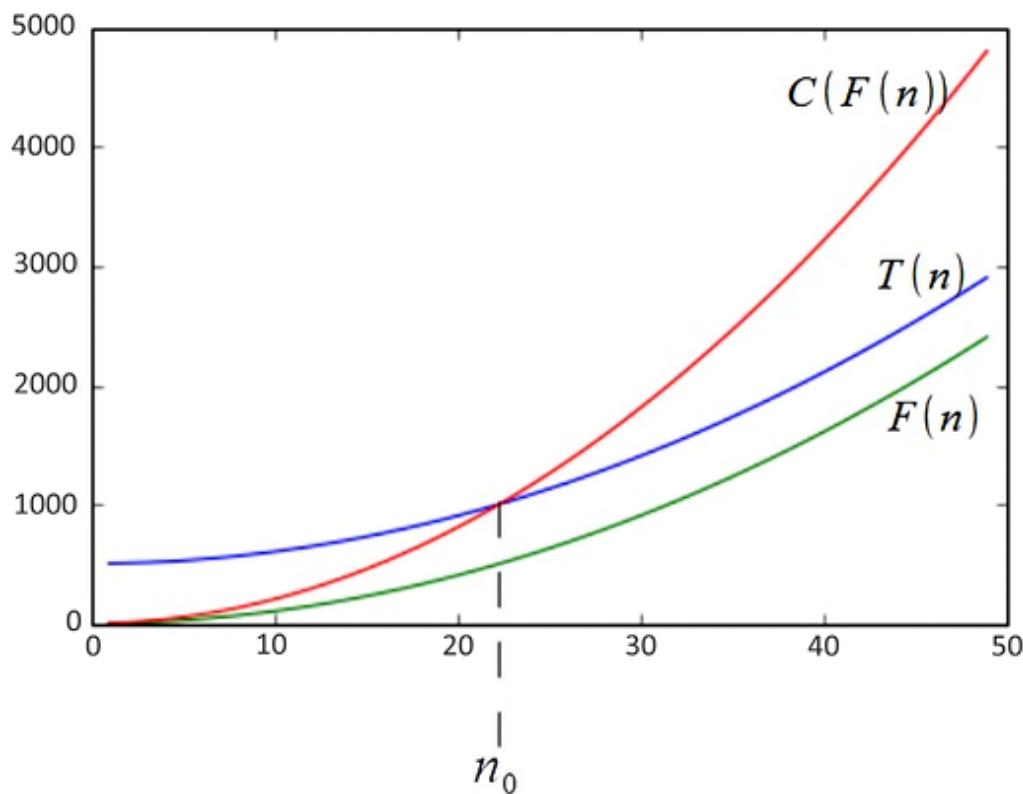


Figure 2.3: Graphical example of O notation

In O notation, $O(F(n))$ is really a set of functions that includes all functions with the same or smaller rates of growth than $F(n)$. For example, $O(n^2)$ also includes $O(n)$, $O(\log n)$, and so on. However, Big O notation should characterize a function as closely as possible, for

example, it is true that function $F(n) = 2n^3+2n^2+5$ is $O(n^4)$, however, it is more accurate that $F(n)$ is $O(n^3)$.

In the following table, we list the most common growth rates in order from lowest to highest.

Time Complexity	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Linear-logarithmic
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

Table 2.1: Runtime complexity of different functions

Using Big O notation, the running time of an algorithm can be computed by analyzing the structure of the algorithm. For example, a double nested loop in an algorithm will have an upper bound on the worst-case running time of $O(n^2)$, since the values of i and j will

be at most n , and both the loops will run n^2 times as shown in the below example code:

```
for i in range(n):
    for j in range(n):
        print("data")
```

Let us consider a few examples in order to compute the upper bound of a function using the O-notation:

1. Find the upper bound for the function:

$$T(n) = 2n + 7$$

Solution: Using O notation, the condition for the upper bound is:

$$T(n) \leq c * F(n)$$

This condition holds true for all values of $n > 7$ and $c=3$.

$$2n + 7 \leq 3n \text{ This is true for all values of } n, \text{ with } c=3, n_0=7$$

$$T(n) = 2n+7 = O(n)$$

2. Find $F(n)$ for functions $T(n) = 2n+5$ such that $T(n) = O(F(n))$.

Solution: Using O notation, the condition for the upper bound is $T(n) \leq c * F(n)$.

Since, $2n+5 \leq 3n$, for all $n \geq 5$.

The condition is true for $c=3, n_0=5$.

$$2n + 5 \leq O(n)$$

$$F(n) = n$$

3. Find $F(n)$ for the function $T(n) = n^2 + n$, such that $T(n) = O(F(n))$.

Solution: Using O notation, since, $n^2 + n \leq 2n^2$, for all $n \geq 1$ (with $c = 2$, $n_0=2$)

$$n^2 + n \leq O(n^2)$$

$$F(n) = n^2$$

4. Prove that $f(n) = 2n^3 - 6n \neq O(n^2)$.

Solution: Clearly, $2n^3 - 6n \geq n^2$, for $n \geq 2$. So it cannot be true that $2n^3 - 6n \neq O(n^2)$.

5. Prove that: $20n^2 + 2n + 5 = O(n^2)$.

Solution: It is clear that:

$$20n^2 + 2n + 5 \leq 21n^2 \text{ for all } n > 4 \text{ (let } c = 21 \text{ and } n_0 = 4\text{)}$$

$$n^2 > 2n + 5 \text{ for all } n > 4$$

So, the complexity is $O(n^2)$.

So, Big-O notation provides an upper bound on a function, which ensures that the function never grows faster than the upper-bounded function. In the next section, we will discuss Omega notation.

Omega notation

Omega notation (Ω) describes an asymptotic lower bound on algorithms, similar to the way in which Big O notation describes an upper bound. Omega notation computes the best-case runtime complexity of the algorithm. The Ω notation ($\Omega(F(n))$) is pronounced as omega of F of n , is a set of functions in such a way that there are

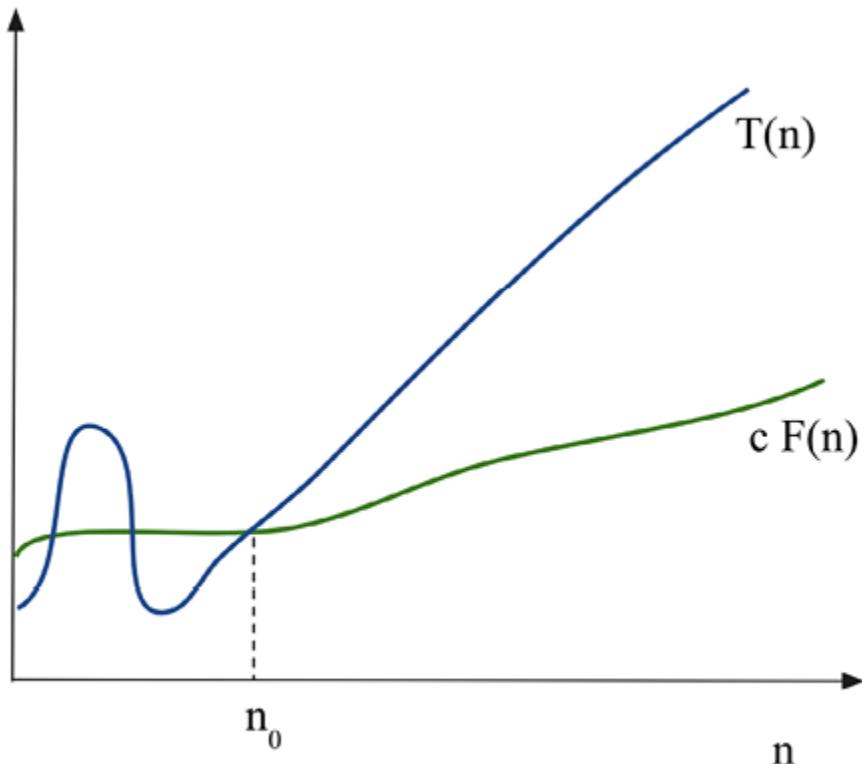
positive constants n_0 and c such that for all values of n greater than n_0 , $T(n)$ always lies on or above a function to $c^*F(n)$.

$$T(n) = \Omega(F(n))$$

Iff constants n_0 and c are present, then:

$$0 \leq c(F(n)) \leq T(n), \text{ for all } n \geq n_0$$

Figure 2.4 shows the graphical representation of the omega (Ω) notation. It can be observed from the figure that the value of $T(n)$ always lies above $cF(n)$ for values of n greater than n_0 .



$$T(n) = \Omega(F(n))$$

Figure 2.4: The graphical representation of Ω notation

If the running time of an algorithm is $\Omega(F(n))$, it means that the running time of the algorithm is at least a constant multiplier of $F(n)$ for sufficiently large values of input size (n). The Ω notation gives a lower bound on the best-case running time complexity of a given algorithm. It means that the running time for a given algorithm will be at least $F(n)$ without depending upon the input.

In order to understand the Ω notation and how to compute the lower bound on the best-case runtime complexity of an algorithm:

1. Find $F(n)$ for the function $T(n) = 2n^2 + 3$ such that $T(n) = \Omega(F(n))$.

Solution: Using the Ω notation, the condition for the lower bound is:

$$c^*F(n) \leq T(n)$$

This condition holds true for all values of n greater than 0, and $c=1$.

$$0 \leq cn^2 \leq 2n^2 + 3, \text{ for all } n \geq 0$$

$$2n^2 + 3 = \Omega(n^2)$$

$$F(n)=n^2$$

2. Find the lower bound for $T(n) = 3n^2$.

Solution: Using the Ω notation, the condition for the lower bound is:

$$c^*F(n) \leq T(n)$$

Consider $0 \leq cn^2 \leq 3n^2$. The condition for Ω notation holds true for all values of n greater than 1, and $c=2$.

$$cn^2 \leq 3n^2 \text{ (for } c = 2 \text{ and } n_0 = 1\text{)}$$

$$3n^2 = \Omega(n^2)$$

3. Prove that $3n = \Omega(n)$.

Solution: Using the Ω notation, the condition for the lower bound is:

$$c^*F(n) \leq T(n)$$

Consider $0 \leq c^*n \leq 3n$. The condition for Ω notation holds true for all values of n greater than 1, and $c=1$.

$$cn^2 \leq 3n^2 \text{ (for } c = 2 \text{ and } n_0 = 1\text{)}$$

$$3n = \Omega(n)$$

The Ω notation is used to describe that at least a certain amount of running time will be taken by an algorithm for a large input size. In the next section, we will discuss amortized analysis.

Amortized analysis

In the amortized analysis of an algorithm, we average the time required to execute a sequence of operations with all the operations of the algorithm. This is called amortized analysis. Amortized analysis is important when we are not interested in the time complexity of individual operations but we are interested in the average runtime of sequences of operations. In an algorithm, each operation requires a different amount of time to execute. Certain operations require significant amounts of time and resources while some operations are not costly at all. In amortized analysis, we analyze algorithms considering both the costly and less costly operations in order to analyze all the sequences of operations. So, an

amortized analysis is the average performance of each operation in the worst case considering the cost of the complete sequence of all the operations. Amortized analysis is different from average-case analysis since the distribution of the input values is not considered. An amortized analysis gives the average performance of each operation in the worst case.

There are three commonly used methods for amortized analysis:

- **Aggregate analysis.** In aggregate analysis, the amortized cost is the average cost of all the sequences of operations. For a given sequence of n operations, the amortized cost of each operation can be computed by dividing the upper bound on the total cost of n operations with n .
- **The accounting method.** In the accounting method, we assign an amortized cost to each operation, which may be different than their actual cost. In this, we impose an extra charge on early operations in the sequence and save “credit cost,” which is used to pay expensive operations later in the sequence.
- **The potential method.** The potential method is like the accounting method. We determine the amortized cost of each operation and impose an extra charge to early operations that may be used later in the sequence. Unlike the accounting method, the potential method accumulates the overcharged credit as “potential energy” of the data structure as a whole instead of storing credit for individual operations.

In this section, we had an overview of amortized analysis. Now we will discuss how to compute the complexity of different functions with examples in the next section.

Composing complexity classes

Normally, we need to find the total running time of complex operations and algorithms. It turns out that we can combine the complexity classes of simple operations to find the complexity class of more complex, combined operations. The goal is to analyze the combined statements in a function or method to understand the total time complexity of executing several operations. The simplest way to combine two complexity classes is to add them. This occurs when we have two sequential operations. For example, consider the two operations of inserting an element into a list and then sorting that list. Assuming that inserting an item occurs in $O(n)$ time, and sorting in $O(n \log n)$ time, then we can write the total time complexity as $O(n + n \log n)$; that is, we bring the two functions inside the $O(\dots)$, as per Big O computation. Considering only the highest-order term, the final worst-case complexity becomes $O(n \log n)$.

If we repeat an operation, for example in a `while` loop, then we multiply the complexity class by the number of times the operation is carried out. If an operation with time complexity $O(f(n))$ is repeated $O(n)$ times, then we multiply the two complexities: $O(f(n)) * O(n) = O(nf(n))$. For example, suppose the function $f(n)$ has a time complexity of $O(n^2)$ and it is executed n times in a `for` loop, as follows:

```
for i in range(n):
    f(...)
```

The time complexity of the above code then becomes:

$$O(n^2) \times O(n) = O(n \times n^2) = O(n^3)$$

Here, we are multiplying the time complexity of the inner function by the number of times this function executes. The runtime of a loop is at most the runtime of the statements inside the loop multiplied by the number of iterations. A single nested loop, that is, one loop nested inside another loop, will run n^2 times, such as in the following example:

```
for i in range(n):
    for j in range(n)
        #statements
```

If each execution of the statements takes constant time, c , i.e. $O(1)$, executed $n \times n$ times, we can express the running time as follows:

$$c \times n \times n = c \times n^2 = O(n^2)$$

For consecutive statements within nested loops, we add the time complexities of each statement and multiply by the number of times the statement is executed—as in the following code, for example:

```
def fun(n):
    for i in range(n):  #executes n times
        print(i)      #c1
    for i in range(n):
        for j in range(n):
            print(j)  #c2
```

This can be written as: $c^1n + c^2 * n^2 = O(n^2)$.

We can define (base 2) logarithmic complexity, reducing the size of the problem by half, in constant time. For example, consider the

following snippet of code:

```
i = 1
while i <= n:
    i = i*2
    print(i)
```

Notice that `i` is doubling in each iteration. If we run this code with $n = 10$, we see that it prints out four numbers: 2, 4, 8, and 16. If we double n , we see it prints out five numbers. With each subsequent doubling of n , the number of iterations is only increased by 1. If we assume that the loop has k iterations, then the value of n will be 2^k . We can write this as follows:

$$\log_2(2^k) = \log_2(n)$$

$$k\log_2(2) = \log_2(n)$$

$$k = \log(n)$$

From this, the worst-case runtime complexity of the above code is equal to $O(\log(n))$.

In this section, we have seen examples to compute the running time complexity of different functions. In the next section, we will take examples to understand how to compute the running time complexity of an algorithm.

Computing the running time complexity of an algorithm

To analyze an algorithm with respect to the best-, worst-, and average-case runtime of the algorithm, it is not always possible to compute these for every given function or algorithm. However, it is always important to know the upper-bound worst-case runtime complexity of an algorithm in practical situations; therefore, we focus on computing the upper-bound Big O notation to compute the worst-case runtime complexity of an algorithm:

1. Find the worst-case runtime complexity of the following Python snippet:

```
# Loop will run n times
for i in range(n):
    print("data") #constant time
```

Solution: The runtime for a loop, in general, takes the time taken by all statements in the loop, multiplied by the number of iterations. Here, total runtime is defined as follows:

$$T(n) = \text{constant time } (c) * n = c*n = O(n)$$

2. Find the time complexity of the following Python snippet:

```
for i in range(n):
    for j in range(n): # This Loop will also run for n times
        print("run")
```

Solution: $O(n^2)$. The `print` statement will be executed n^2 times, n times for the inner loop, and, for each iteration of the outer loop, the inner loop will be executed.

3. Find the time complexity of the following Python snippet:

```
for i in range(n):
    for j in range(n):
        print("run fun")
        break
```

Solution: The worst-case complexity will be $O(n)$ since the `print` statement will run n times because the inner loop executes only once due to a `break` statement.

4. Find the time complexity of the following Python snippet:

```
def fun(n):
    for i in range(n):
        print("data") #constant time
    #outer loop execute for n times
    for i in range(n):
        for j in range(n): #inner loop execute n times
            print("run fun") #constant time
```

Solution: Here, the `print` statements will execute n times in the first loop and n^2 times for the second nested loop. Here, the total time required is defined as the following:

$$T(n) = \text{constant time } (c_1) * n + c_2 * n * n$$

$$c_1 n + c_2 n^2 = O(n^2)$$

5. Find the time complexity of the following Python snippet:

```
if n == 0: #constant time
    print("data")
else:
    for i in range(n): #Loop run for n times
        print("structure")
```

Solution: $O(n)$. Here, the worst-case runtime complexity will be the time required for the execution of all the statements; that is, the time required for the execution of the `if-else` conditions, and the `for` loop. The time required is defined as the following:

$$T(n) = c_1 + c_2 n = O(n)$$

6. Find the time complexity of the following Python snippet:

```
i = 1
j = 0
while i*i < n:
    j = j +1
    i = i+1
    print("data")
```

Solution: $O(\sqrt{n})$. The loop will terminate based on the value of `i`; the loop will iterate based on the condition:

$$i^2 \leq n$$

$$T(n) = O(\sqrt{n})$$

7. Find the time complexity of the following Python snippet:

```
i = 0
for i in range(int(n/2), n):
    j = 1
    while j+n/2 <= n:
        k = 1
        while k < n:
            k *= 2
            print("data")
        j += 1
```

Solution: Here, the outer loop will execute $n/2$ times, the middle loop will also run $n/2$ times, and the innermost loop will run for $\log(n)$ time. So, the total running time complexity will be $O(n^*n^*\log n)$:

$O(n^2\log n)$

Summary

In this chapter, we have looked at an overview of algorithm design. The study of algorithms is important because it trains us to think very specifically about certain problems. It is conducive to increasing our problem-solving abilities by isolating the components of a problem and defining the relationships between them. In this chapter, we discussed different methods for analyzing algorithms and comparing algorithms. We also discussed asymptotic notations, namely: Big O, Ω , and θ notation.

In the next chapter, we will discuss algorithm design techniques and strategies.

Exercises

1. Find the time complexity of the following Python snippets:

a.

```
i=1
while(i<n):
    i*=2
    print("data")
```

b.

```
i =n
while(i>0):
    print('complexity')
    i/ = 2
```

c.

```
for i in range(1,n):
    j = i
    while(j<n):
        j*=2
```

d.

```
i=1
while(i<n):
    print('python')
    i = i**2
```

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



3

Algorithm Design Techniques and Strategies

In the field of computing, algorithm design is very important for IT professionals for improving their skills and enabling growth in the industry. The algorithm design process starts with a substantial number of real-world computing problems, which must be clearly formulated for efficiently building the solution using one of the possible techniques from the range of algorithm design techniques available. The world of algorithms contains a plethora of techniques and design principles, mastery of which is required to tackle more difficult problems in the field. Algorithm designs are important in computer science, in general, to efficiently design the solution for a precisely formulated problem since a very sophisticated and complex problem can easily be solved with an appropriate algorithm design technique.

In this chapter, we will discuss the ways in which different kinds of algorithms can be categorized. Design techniques will be described and illustrated, and we will further discuss the analysis of algorithms. Finally, we will provide detailed implementations for a few very important algorithms.

In this chapter, we will look at the following algorithm design techniques:

- Divide and conquer
- Dynamic programming
- Greedy algorithms

Algorithm design techniques

Algorithm design is a powerful tool for viewing and clearly understanding well-posed, real-world problems. A straightforward, or **brute-force**, approach is available that is very simple, yet effective, for many problems. The brute-force approach is trying all possible combinations of solutions in order to solve any problem. For example, suppose a salesperson has to visit 10 cities across the country. In which order should the cities be visited in order to minimize the total distance traveled? The brute-force approach to this problem will be to calculate the total distance for all possible combinations of routes, and then select the route that provides the smallest distance.

As you might guess, the brute-force algorithm is not efficient.

It can provide useful solutions for limited input sizes, but it becomes very inefficient when the input size becomes large. Therefore, we will break the process down into two fundamental components for finding the optimal solution for a computing problem:

1. Formulate the problem clearly
2. Identify the appropriate algorithm design technique based on the structure of the problem for an efficient solution

That is why the study of algorithm design becomes very important when developing scalable and robust systems. Design and analysis are important in the first instance because they assist in developing algorithms that are organized and easy to understand. Design technique guidelines also help in developing new algorithms easily for complex problems. Moreover, design techniques can also be used to categorize the algorithms and this also helps to understand them better. There are several algorithm paradigms as follows:

- Recursion
- Divide and conquer
- Dynamic programming
- Greedy algorithms

Since we will be using recursion many times while discussing different algorithm design techniques, let us first understand the concept of recursion, and thereafter, we will discuss different algorithm design techniques.

Recursion

A recursive algorithm calls itself repeatedly in order to solve the problem until a certain condition is fulfilled. Each recursive call itself spins off other recursive calls. A recursive function can be in an infinite loop; therefore, it is required that each recursive function adheres to certain properties. At the core of a recursive function are two types of cases:

1. **Base cases:** These tell the recursion when to terminate, meaning the recursion will be stopped once the base condition is met

2. Recursive cases: The function calls itself recursively, and we progress toward achieving the base criteria

A simple problem that naturally lends itself to a recursive solution is calculating factorials. The recursive factorial algorithm defines two cases: the base case when n is zero (the terminating condition) and the recursive case when n is greater than zero (the call of the function itself). A typical implementation is as follows:

```
def factorial(n):
    # test for a base case
    if n == 0:
        return 1
    else:
        # make a calculation and a recursive call
        return n*factorial(n-1)
print(factorial(4))
```

This produces the following output:

24

To calculate the factorial of 4, we require four recursive calls, plus the initial parent call, as can be seen in *Figure 3.1*. The details of how these recursive calls work is as follows. Initially, the number 4 is passed to the factorial function, which will return the value 4 multiplied by the factorial of (4-1=3). For this, the number 3 is again passed to the factorial function, which will return the value 3 multiplied by the factorial of (3-1=2). Similarly, in the next iteration, the value 2 is multiplied by the factorial of (2-1 =1).

This continues until we reach the factorial of `0`, which returns `1`. Now, each function returns the value to finally compute `1*1*2*3*4=24`, which is the final output of the function.

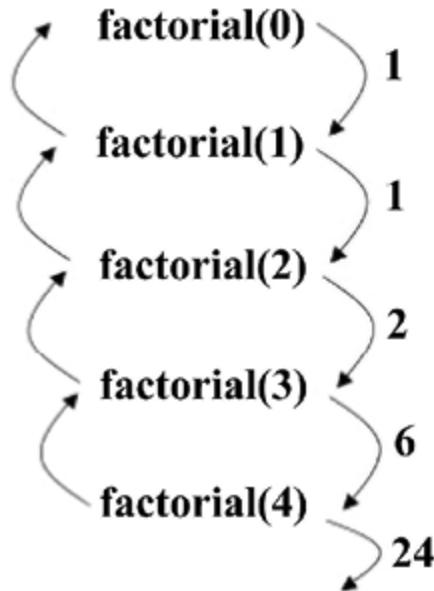


Figure 3.1: The flow of execution of the factorial 4

We discussed the concept of recursion, which will be very useful in understanding the implementation of different algorithm paradigms. So, now let us move on to the distinct algorithm design strategies in turn, starting with the divide-and-conquer technique in the next section.

Divide and conquer

One of the important and effective techniques for solving a complex problem is divide and conquer. The divide-and-conquer paradigm divides a problem into smaller sub-problems, and then solves these; finally, it combines the results to obtain a global, optimal solution.

More specifically, in divide-and-conquer design, the problem is divided into two smaller sub-problems, with each of them being solved recursively. The partial solutions are merged to obtain a final solution. This is a very common problem-solving technique, and is, arguably, the most commonly used approach in algorithm design.

Some examples of the divide-and-conquer design technique are as follows:

- Binary search
- Merge sort
- Quick sort
- Algorithm for fast multiplication
- Strassen's matrix multiplication
- Closest pair of points

Let's have a look at two examples, the binary search and merge sort algorithms, to understand how the divide-and-conquer design technique works.

Binary search

The binary search algorithm is based on the divide-and-conquer design technique. This algorithm is used to find a given element from a sorted list of elements. It first compares the search element with the middle element of the list; if the search element is smaller than the middle element, then the half of the list of elements greater than the middle element is discarded; the process repeats recursively until the search element is found or we reach the end of the list. It is important to note that in each iteration, half of the search space is

discarded, which improves the performance of the overall algorithm because there are fewer elements to search through.

Take the example shown in *Figure 3.2*; let's say we want to search for element 4 in the given sorted list of elements. The list is divided in half in each iteration; with the divide-and-conquer strategy, the element is searched $O(\log n)$ times.

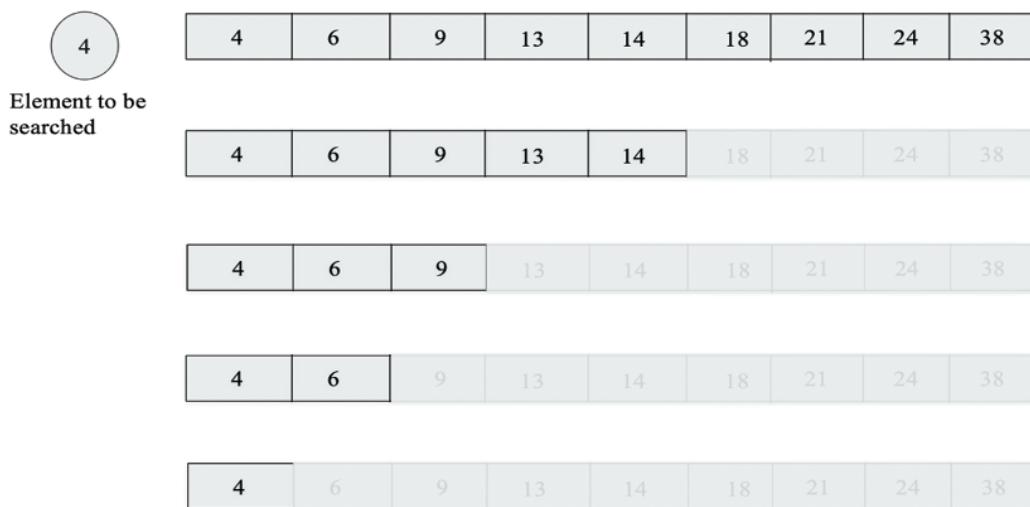


Figure 3.2: The process of searching for an element using a binary search algorithm

The Python code for searching for an element in a sorted list of elements is shown here:

```
def binary_search(arr, start, end, key):
    while start <= end:
        mid = start + (end - start)/2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            start = mid + 1
        else:
            end = mid - 1
    return -1
arr = [4, 6, 9, 13, 14, 18, 21, 24, 38]
```

```
x = 13
result = binary_search(arr, 0, len(arr)-1, x)
print(result)
```

When we search for 13 in the given list of elements, the output of the preceding code is 3, which is the position of the searched item.

In the code, initially, the start and end index give the position of the first and last index of the input array [4, 6, 9, 13, 14, 18, 21, 24, 38]. The item to be searched that is stored in the variable key is firstly matched with the mid element of the array, and then we discard half of the list and search for the item in another half of the list. The process is iterated until we find the item to be searched, or we reach the end of the list, and we don't find the element.

When analyzing the workings of the binary search algorithm in the worst case, we can see that for a given array of 8 elements, following the first unsuccessful attempt, the list is halved, and then again for an unsuccessful search attempt, the list is of length 2, and finally, only 1 element is left. So, the binary search requires 4 searches. If the size of the list is doubled, in other words, to 16, following the first unsuccessful search, we will have a list of size 8, and that will take a total of 4 searches. Therefore, the binary search algorithm will require 5 searches for a list of 16 items. Thus, we can observe that when we double the number of items in the list, the number of searches required also increments by 1. We can say this as when we have a list of length n, the total number of searches required will be the number of times we repeated halving the list until we are left with 1 element plus 1, which is mathematically equivalent to $(\log_2 n + 1)$. For example, if n=8, the output will be 3, meaning the number

of searches required will be 4. The list is divided in half in each iteration; with the divide-and-conquer strategy, the worst-case time complexity of the binary search algorithm is $O(\log n)$.

Merge sort is another popular algorithm that is based on the divide-and-conquer design strategy. We will be discussing merge sort in more detail in the next section.

Merge sort

Merge sort is an algorithm for sorting a list of n natural numbers in increasing order. Firstly, the given list of elements is divided iteratively into equal parts until each sublist contains one element, and then these sublists are combined to create a new list in a sorted order. This programming approach to problem-solving is based on the divide-and-conquer methodology and emphasizes the need to break down a problem into smaller sub-problems of the same type or form as the original problem. These sub-problems are solved separately and then results are combined to obtain the solution of the original problem.

In this case, given a list of unsorted elements, we split the list into two approximate halves. We continue to divide the list into halves recursively.

After a while, the sublist created as a result of the recursive call will contain only one element. At that point, we begin to merge the solutions in the conquer or merge step. This process is shown in *Figure 3.3:*

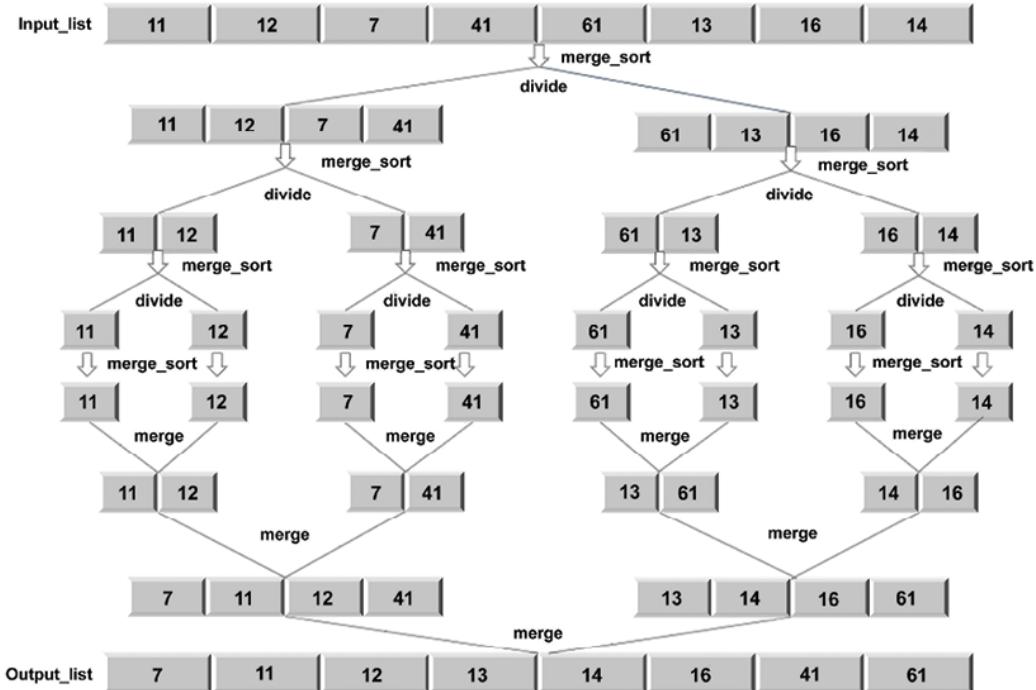


Figure 3.3: Overview of the merge sort algorithm

The implementation of the merge sort algorithm is implemented using primarily two methods, namely, the `merge_sort` method, which recursively divides the list. Afterward, we will introduce the `merge` method to combine the results:

```
def merge_sort(unsorted_list):
    if len(unsorted_list) == 1:
        return unsorted_list
    mid_point = int(len(unsorted_list)/2)
    first_half = unsorted_list[:mid_point]
    second_half = unsorted_list[mid_point:]
    half_a = merge_sort(first_half)
    half_b = merge_sort(second_half)
    return merge(half_a, half_b)
```

The implementation starts by accepting the list of unsorted elements into the `merge_sort` function. The `if` statement is used to establish the

base case, where, if there is only one element in the `unsorted_list`, we simply return that list again. If there is more than one element in the list, we find the approximate middle using `mid_point = len(unsorted_list)//2`.

Using this `mid_point`, we divide the list into two sublists, namely, `first_half` and `second_half`:

```
first_half = unsorted_list[:mid_point]
second_half = unsorted_list[mid_point:]
```

A recursive call is made by passing the two sublist to the `merge_sort` function again:

```
half_a = merge_sort(first_half)
half_b = merge_sort(second_half)
```

Now, for the merge step, `half_a` and `half_b` are sorted. When `half_a` and `half_b` have passed their values, we call the `merge` function, which will merge or combine the two solutions stored in `half_a` and `half_b`, which are lists:

```
def merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
```

```
i += 1
while j < len(second_sublist):
    merged_list.append(second_sublist[j])
    j += 1
return merged_list
```

The `merge` function takes the two lists we want to merge, `first_sublist` and `second_sublist`. The `i` and `j` variables are initialized to 0 and are used as pointers to tell us where we are in the two lists with respect to the merging process.

The final `merged_list` will contain the merged list.

The `while` loop starts comparing the elements in `first_sublist` and `second_sublist`:

```
while i < len(first_sublist) and j < len(second_sublist):
    if first_sublist[i] < second_sublist[j]:
        merged_list.append(first_sublist[i])
        i += 1
    else:
        merged_list.append(second_sublist[j])
        j += 1
```

The `if` statement selects the smaller of the two, `first_sublist[i]` or `second_sublist[j]`, and appends it to `merged_list`. The `i` or `j` index is incremented to reflect where we are with the merging step. The `while` loop stops when either sublist is empty.

There may be elements left behind in either `first_sublist` or `second_sublist`. The last two `while` loops make sure that those elements are added to `merged_list` before it is returned. The last call to `merge(half_a, half_b)` will return the sorted list. The following code shows how to pass an array to sort the elements using merge sort:

```
a= [11, 12, 7, 41, 61, 13, 16, 14]  
print(merge_sort(a))
```

The output will be:

```
[7, 11, 12, 14, 16, 41, 61]
```

Let's give the algorithm a dry run by merging the two sublists [4, 6, 8] and [5, 7, 11, 40], shown in *Table 3.1*. In this example, initially, the two sorted sublists are given, and then the first elements are matched, and since the first element of the first list is smaller, it is moved to `merge_list`. Next, in *step 2*, again, the starting elements from both of the lists are matched, and the smaller element, which is from the second list, is moved to `merge_list`. The same process is repeated until one of the lists becomes empty.

Step	first_sublist	second_sublist	merged_list
0	[4 6 8]	[5 7 11 40]	[]
1	[6 8]	[5 7 11 40]	[4]
2	[6 8]	[7 11 40]	[4 5]
3	[8]	[7 11 40]	[4 5 6]
4	[8]	[11 40]	[4 5 6 7]

5	[]	[11 40]	[4 5 6 7 8]
6	[]	[]	[4 5 6 7 8 11 40]

Table 3.1: Example of merging two lists

This process can also be seen in *Figure 3.4*:

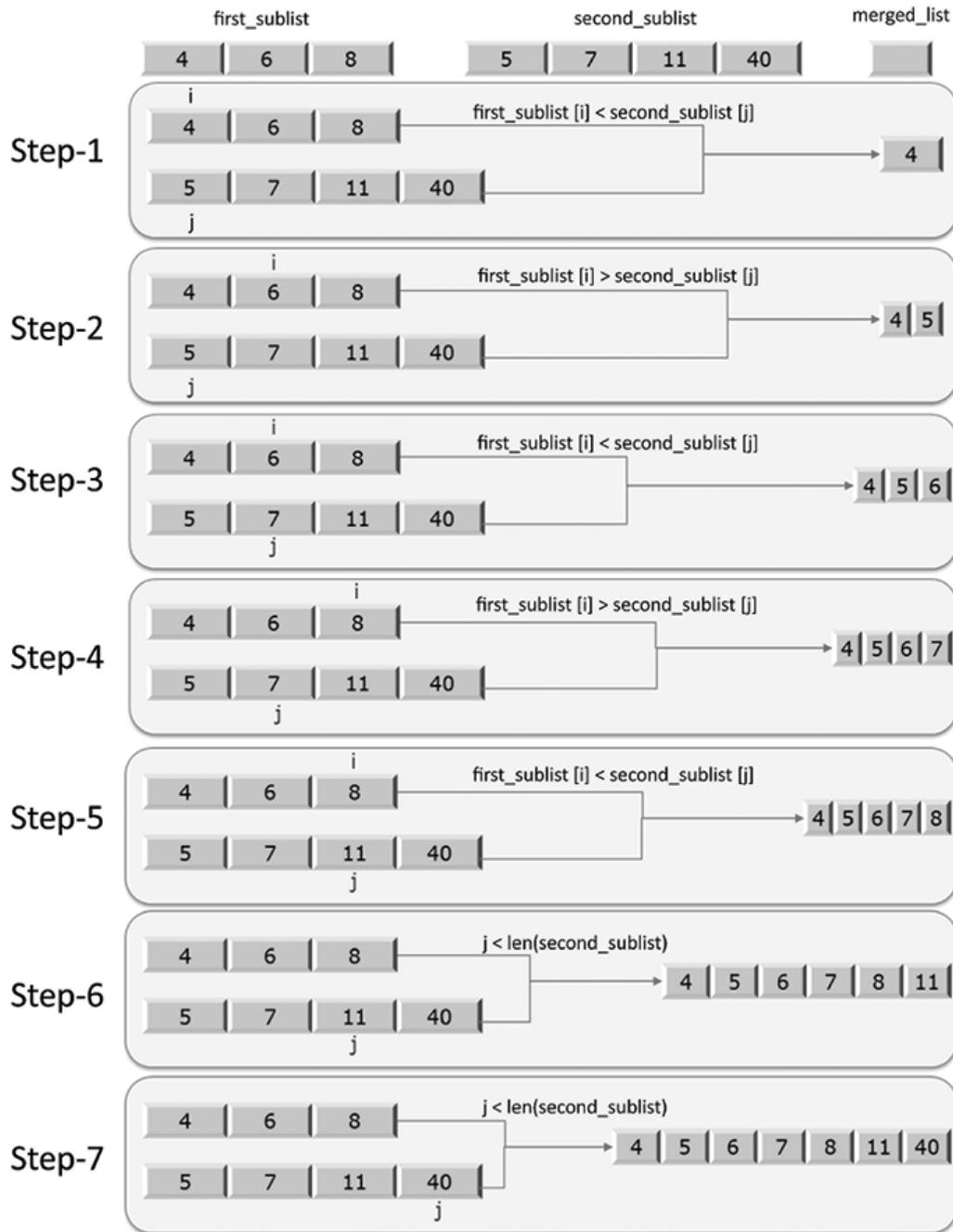


Figure 3.4: The process of merging the two sublists

After one of the lists becomes empty, like after *step 4* in this example, at this point in the execution, the third `while` loop in the `merge` function kicks in to move `11` and `40` into `merged_list`. The returned `merged_list` will contain the fully sorted list.

The worst-case running time complexity of the merge sort will depend on the following steps:

1. Firstly, the divide step will take a constant time since it just computes the midpoint, which can be done in $O(1)$ time
2. Then, in each iteration, we divide the list into half recursively, which will take $O(\log n)$, which is quite similar to what we have seen in the binary search algorithm
3. Further, the combine/merge step merges all the n elements into the original array, which will take (n) time.

Hence, the merge sort algorithm has a runtime complexity of $O(\log n)$ $T(n) = O(n) * O(\log n) = O(n \log n)$.

We have discussed the divide-and-conquer algorithm design technique with the help of a few examples. In the next section, we will discuss another algorithm design technique: dynamic programming.

Dynamic programming

Dynamic programming is the most powerful design technique for solving optimization problems. Such problems generally have many possible solutions. The basic idea of dynamic programming is based on the intuition of the divide-and-conquer technique. Here, essentially, we explore the space of all the possible solutions by decomposing the problem into a series of sub-problems and then combining the results to compute the correct solution for the large problem. The divide-and-conquer algorithm is used to solve a problem by combining the solutions of the non-overlapping

(disjoint) sub-problems, whereas dynamic programming is used when the sub-problems are overlapping, meaning that the sub-problems share sub-sub-problems. The dynamic programming technique is similar to divide and conquer in that a problem is broken down into smaller problems. However, in divide and conquer, each sub-problem has to be solved before its results can be used to solve bigger problems. In contrast, dynamic programming-based techniques solve each sub-sub-problems only once and do not recompute the solution to an already-encountered sub-problem. Rather, it uses a remembering technique to avoid the re-computation.

Dynamic programming problems have two important characteristics:

- **Optimal substructure:** Given any problem, if the solution can be obtained by combining the solutions of its sub-problems, then the problem is said to have an optimal substructure. In other words, an optimal substructure means that the optimal solution of the problem can be obtained from the optimal solution of its sub-problems. For example, the i^{th} Fibonacci number from its series can be computed from $(i-1)^{\text{th}}$ and $(i-2)^{\text{th}}$ Fibonacci numbers; for example, $\text{fib}(6)$ can be computed from $\text{fib}(5)$ and $\text{fib}(4)$.
- **Overlapping sub-problem:** If an algorithm has to repeatedly solve the same sub-problem again and again, then the problem has overlapping sub-problems. For example, $\text{fib}(5)$ will have multiple time computations for $\text{fib}(3)$ and $\text{fib}(2)$.

If a problem has these characteristics, then the dynamic programming approach is useful, since the implementation can be improved by reusing the same solution computed before. In a dynamic programming strategy, the problem is broken down into independent sub-problems, and the intermediate results are cached, which can then be used in subsequent operations.

In the dynamic approach, we divide a given problem into smaller sub-problems. In recursion also, we divide the problem into sub-problems. However, the difference between recursion and dynamic programming is that similar sub-problems can be solved any number of times, but in dynamic programming, we keep track of previously solved sub-problems, and care is taken not to recompute any of the previously encountered sub-problems. One property that makes a problem an ideal candidate for being solved with dynamic programming is that it has an **overlapping set of sub-problems**. Once we realize that the form of sub-problems has repeated itself during computation, we need not compute it again. Instead, we return a pre computed result for that previously encountered sub-problem.

Dynamic programming takes account of the fact that each sub-problem should be solved only once, and to ensure that we never re-evaluate a sub-problem, we need an efficient way to store the results of each sub-problem. The following two techniques are readily available:

- **Top-down with memoization:** This technique starts from the initial problem set and divides it into small sub-problems. After the solution to a sub-program has been determined, we store the

result of that particular sub-problem. In the future, when this sub-problem is encountered, we only return its pre computed result. Therefore, if the solution to a given problem can be formulated recursively using the solution of the sub-problems, then the solution of the overlapping sub-problems can easily be memoized.

Memoization means storing the solution of the sub-problem in an array or hash table. Whenever a solution to a sub-problem needs to be computed, it is first referred to the saved values if it is already computed, and if it is not stored, then it is computed in the usual manner. This procedure is called *memoized*, which means it “remembers” the results of the operation that has been computed before.

- **Bottom-up approach:** This approach depends upon the “size” of the sub-problems. We solve the smaller sub-problems first, and then while solving a particular sub-problem, we already have a solution of the smaller sub-problems on which it depends. Each sub-problem is solved only once, and whenever we try to solve any sub-problem, solutions to all the prerequisite smaller sub-problems are available, which can be used to solve it. In this approach, a given problem is solved by dividing it into sub-problems recursively, with the smallest possible sub-problems then being solved. Furthermore, the solutions to the sub-problems are combined in a bottom-up fashion to arrive at the solution to the bigger sub-problem in order to recursively reach the final solution.

Let's consider an example to understand how dynamic programming works. Let us solve the problem of the Fibonacci series using dynamic programming.

Calculating the Fibonacci series

The Fibonacci series can be demonstrated using a recurrence relation. Recurrence relations are recursive functions that are used to define mathematical functions or sequences. For example, the following recurrence relation defines the Fibonacci sequence [1, 1, 2, 3, 5, 8 ...]:

```
func(0) = 1
func(1) = 1
func(n) = func(n-1) + func(n-2) for n>1
```

Note that the Fibonacci sequence can be generated by putting the values of n in sequence [0, 1, 2, 3, 4, ...]. Let's take an example to generate the Fibonacci series to the fifth term:

```
1 1 2 3 5
```

A recursive-style program to generate the sequence would be as follows:

```
def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
for i in range(5):
    print(fib(i))
```

This will produce output like the following:

```
1  
1  
2  
3  
5
```

In this code, we can see that the recursive calls are being called in order to solve the problem. When the base case is met, the `fib()` function returns `1`. If n is equal to or less than 1, the base case is met. If the base case is not met, we call the `fib()` function again. The recursion tree to solve up to the fifth term in the Fibonacci sequence is shown in *Figure 3.5*:

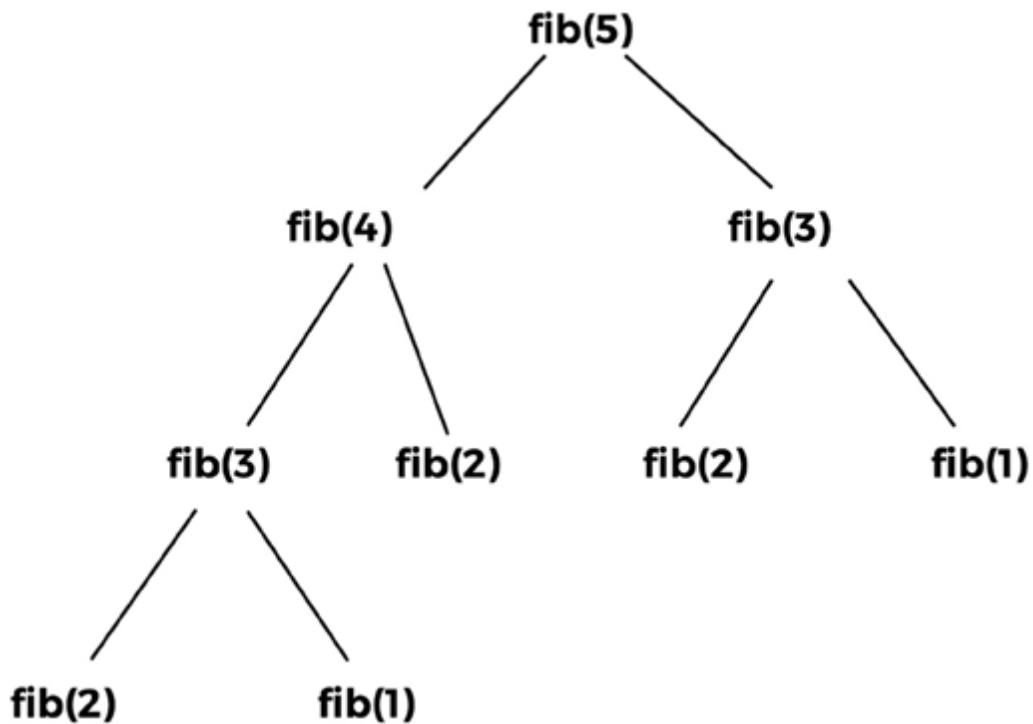


Figure 3.5: Recursion tree for `fib(5)`

We can observe from the overlapping sub-problems from the recursion tree as shown in *Figure 3.6* that the call to **fib(1)** happens twice, the call to **fib(2)** happens three times, and the call to **fib(3)** occurs twice. The return values of the same function call never change; for example, the return value for **fib(2)** will always be the same whenever we call it. Likewise, it will also be the same for **fib(1)** and **fib(3)**. So, they are overlapping problems, thus, computational time will be wasted if we compute the same function again whenever it is encountered. These repeated calls to a function with the same parameters and output suggest that there is an overlap. Certain computations reoccur down in the smaller sub-problem.

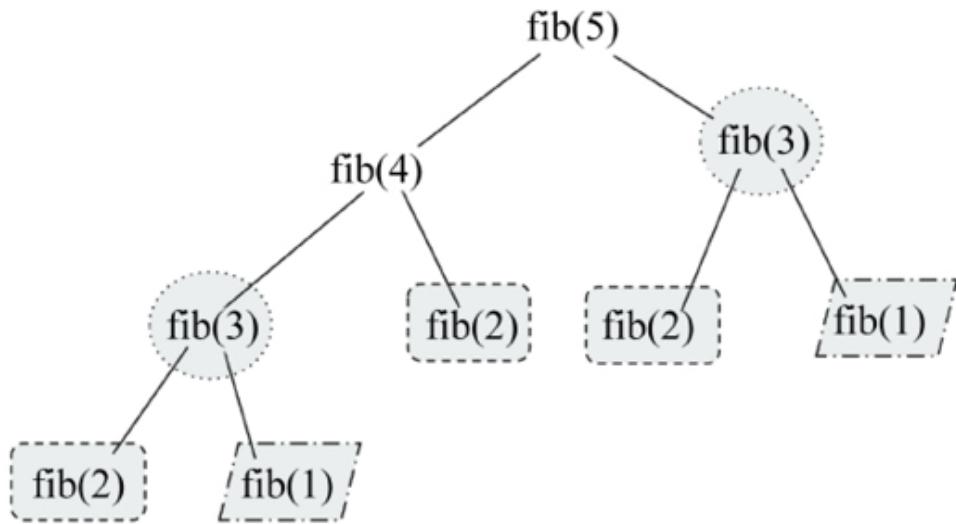
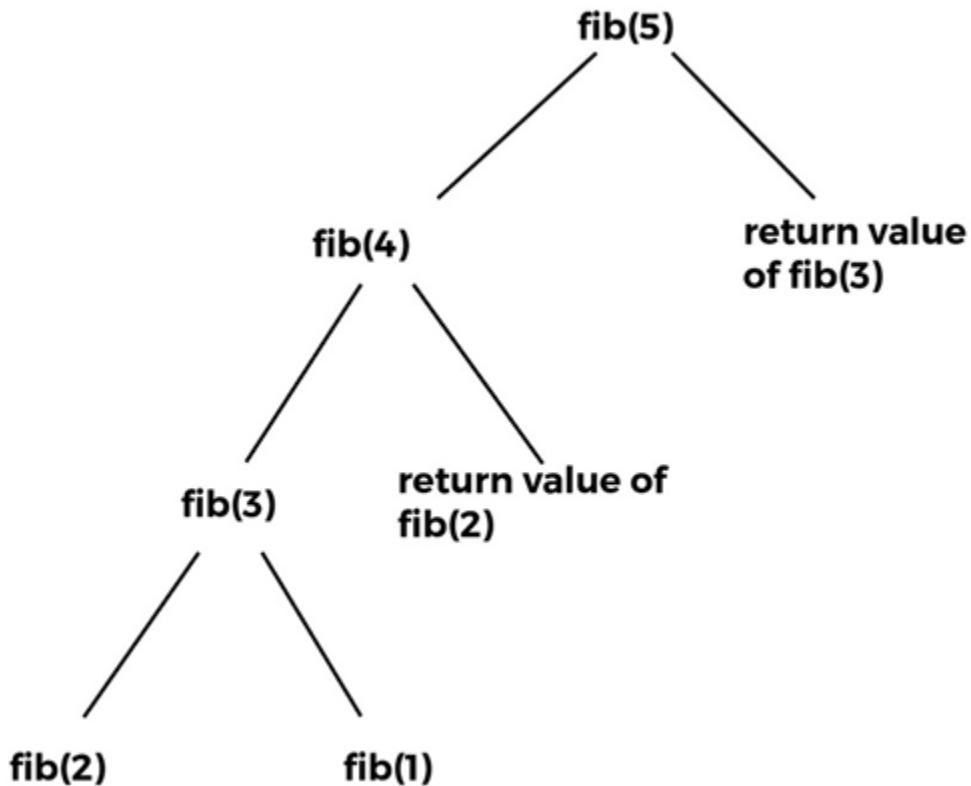


Figure 3.6: Overlapping sub-problems shown in the recursion tree for fib(5)

In dynamic programming using the memoization technique, we store the results of the computation of **fib(1)** the first time it is encountered. Similarly, we store return values for **fib(2)** and **fib(3)**. Later, whenever we encounter a call to **fib(1)**, **fib(2)**, or **fib(3)**, we

simply return their respective results. The recursive tree diagram is shown in *Figure 3.7*:



*Figure 3.7: Recursion tree for **fib(5)** showing re-use of the already computed values*

Thus, in dynamic programming, we have eliminated the need to compute **fib(3)**, **fib(2)**, and **fib(1)** if they are encountered multiple times. This is called the memoization technique, wherein there is no recomputation of overlapping calls to functions when breaking a problem down into its sub-problems.

Hence, the overlapping function calls in our Fibonacci example are **fib(1)**, **fib(2)**, and **fib(3)**. Below is the code for the dynamic programming-based implementation for the Fibonacci series.

```

def dyna_fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if lookup[n] is not None:
        return lookup[n]

    lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
    return lookup[n]
lookup = [None]*(1000)

for i in range(6):
    print(dyna_fib(i))

```

This will produce an output like the following:

```

0
1
1
2
3
5

```

In the dynamic implementation of the Fibonacci series, we store the results of previously solved sub-problems in a list (in other words, a `lookup` in this example code). We first check whether the Fibonacci of any number is already computed; if it is already computed, then we return the stored value from the `lookup[n]`. Otherwise, when we compute its value, it is done through the following code:

```

if lookup[n] is not None:
    return lookup[n]

```

After computing the solution of the sub-problem, it is again stored in the lookup list. The Fibonacci number of the given value is returned as shown in the following code snippet:

```
lookup[n] = dyna_fib(n-1) + dyna_fib(n-2)
```

Furthermore, in order to store a list of 1,000 elements, we create a list `lookup` using the `dyna_fib` function:

```
lookup = [None]*(1000)
```

So, in dynamic programming-based solutions, we use the precomputed solutions in order to compute the final results.

Dynamic programming improves the running time complexity of the algorithm. In the recursive approach, for every value, two functions are called; for example, `fib(5)` calls `fib(4)` and `fib(3)`, and then `fib(4)` calls `fib(3)` and `fib(2)`, and so on. Thus, the time complexity for the recursive approach is $O(2^n)$, whereas, in the dynamic programming approach, we do not recompute the sub-problems, so for `fib(n)`, we have n total values to be computed, in other words, `fib(0), fib(1), fib(2)... fib(n)`. Thus, we only solve these values once, so the total running time complexity is $O(n)$. Thus, dynamic programming in general improves performance.

In this section, we have discussed the dynamic programming design technique, and in the next section, we discuss the design techniques for greedy algorithms.

Greedy algorithms

Greedy algorithms often involve optimization and combinatorial problems. In greedy algorithms, the objective is to obtain the optimum solution from many possible solutions in each step. We try to get the local optimum solution, which may eventually lead us to obtain the global optimum solution. The greedy strategy does not always produce the optimal solution. However, the sequence of locally optimal solutions generally approximates the globally optimal solution.

For example, consider that you are given some random digits, say 1, 4, 2, 6, 9, and 5. Now you have to make the biggest number by using all the digits without repeating any digit. To create the biggest number from the given digits using the greedy strategy, we perform the following steps. Firstly, we select the largest digit from the given digits, and then append it to the number and remove the digit from the list until we have no digits left in the list. Once all the digits have been used, we get the largest number that can be formed by using these digits: 965421. The stepwise solution to this problem is shown in *Figure 3.8*:

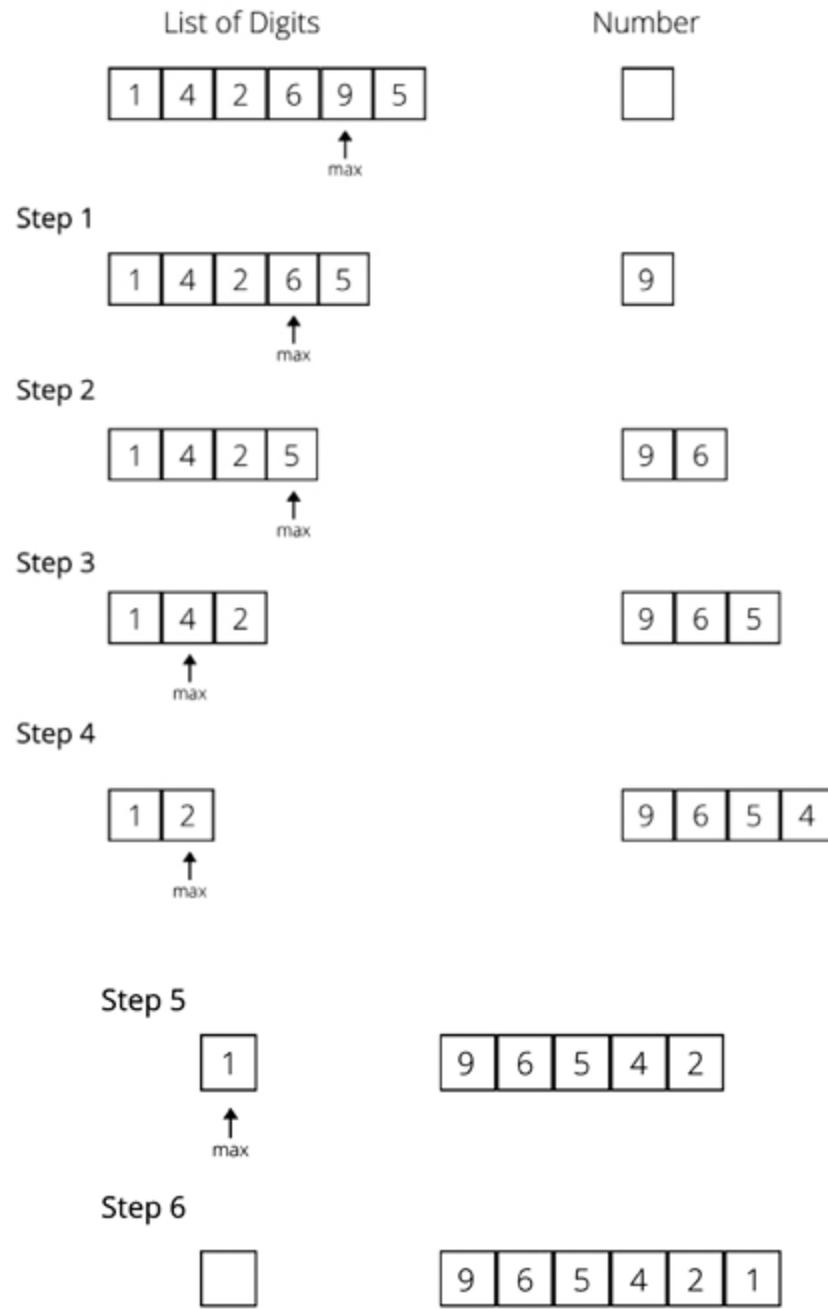


Figure 3.8: Example of a greedy algorithm

Let us consider another example to better understand the greedy approach. Say you have to give 29 Indian rupees to someone in the minimum number of notes, giving one note at a time, but never exceeding the owed amount. Assume that we have notes in

denominations of 1, 2, 5, 10, 20, and 50. To solve this using the greedy approach, we will start by handing over the 20-rupee note, then for the remaining 9 rupees, we will give a 5-rupee note; for the remaining 4 rupees, we will give the 2-rupee note, and then another 2-rupee note.

In this approach, at each step, we chose the best possible solution and gave the largest available note. Assume that, for this example, we have to use the notes of 1, 14, and 25. Then, using the greedy approach, we will pick the 25-rupee note, and then four 1-rupee notes, which makes a total of 5 notes. However, this is not the minimum number of notes possible . The better solution would be to give notes of 14, 14, and 1. Thus, it is also clear that the greedy approach does not always give the best solution, but a feasible and simple one.

The classic example is to apply the greedy algorithm to the traveling salesperson problem, where a greedy approach always chooses the closest destination first. In this problem, a greedy approach always chooses the closest unvisited city in relation to the current city; in this way, we are not sure that we will get the best solution, but we surely get an optimal solution. This shortest-path strategy involves finding the best solution to a local problem in the hope that this will lead to a global solution.

Listed here are many popular standard problems where we can use greedy algorithms to obtain the optimum solution:

- Kruskal's minimum spanning tree
- Dijkstra's shortest path problem
- The Knapsack problem

- Prim's minimal spanning tree algorithm
- The traveling salesperson problem

Let us discuss one of the popular problems, in other words, the shortest path problem, which can be solved using the greedy approach, in the next section.

Shortest path problem

The shortest path problem requires us to find out the shortest possible route between nodes on a graph. Dijkstra's algorithm is a very popular method for solving this using the greedy approach. The algorithm is used to find the shortest distance from a source to a destination node in a graph.

Dijkstra's algorithm works for weighted directed and undirected graphs. The algorithm produces the output of a list of the shortest path from a given source node, A, in a weighted graph. The algorithm works as follows:

1. Initially, mark all the nodes as unvisited, and set their distance from the given source node to infinity (the source node is set to zero).
2. Set the source node as the current one.
3. For the current node, look for all the unvisited adjacent nodes, and compute the distance to that node from the source node through the current node. Compare the newly computed distance to the currently assigned distance, and if it is smaller, set this as the new value.

Once we have considered all the unvisited adjacent nodes of the current node, we mark it as visited.

If the destination node has been marked visited, or if the list of unvisited nodes is empty, meaning we have considered all the unvisited nodes, then the algorithm is finished.

We next consider the next unvisited node that has the shortest distance from the source node. Repeat *steps 2 to 6*.

Consider the example in *Figure 3.9* of a weighted graph with six nodes [A, B, C, D, E, and F] to understand how Dijkstra's algorithm works.

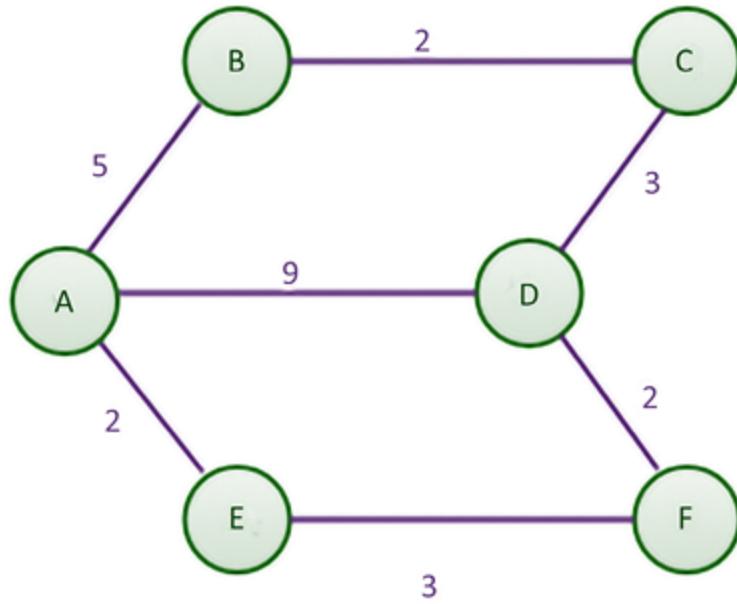


Figure 3.9: Example weighted graph with six nodes

By manual inspection, the shortest path between nodes **A** and **D**, at first glance, seems to be the direct line with a distance of 9. However, the shortest route means the lowest total distance, even if this comprises several parts. By comparison, traveling from node **A** to **E**,

then from **E** to **F**, and finally to **D** will incur a total distance of 7, making it a shorter route.

We would implement the shortest path algorithm with a single source. It would determine the shortest path from the origin, which in this case is **A**, to any other node in the graph. In *Chapter 9, Graphs and Other Algorithms*, we will discuss how to represent a graph with an adjacency list. We use an adjacency list along with the weight/cost/distance on every edge to represent the graph, as shown in the following Python code. The adjacency list for the diagram and table is as follows:

```
graph = dict()
graph[ 'A' ] = { 'B': 5, 'D': 9, 'E': 2}
graph[ 'B' ] = { 'A': 5, 'C': 2}
graph[ 'C' ] = { 'B': 2, 'D': 3}
graph[ 'D' ] = { 'A': 9, 'F': 2, 'C': 3}
graph[ 'E' ] = { 'A': 2, 'F': 3}
graph[ 'F' ] = { 'E': 3, 'D': 2}
```

We will return to the rest of the code after a visual demonstration, but don't forget to declare the graph to ensure the code runs correctly.

The nested dictionary holds the distance and adjacent nodes. A table is used to keep track of the shortest distance from the source in the graph to any other node. *Table 3.2* is the starting table:

Node	Shortest distance from source	Previous node
A	0	None

B	∞	None
C	∞	None
D	∞	None
E	∞	None
F	∞	None

Table 3.2: Initial table showing the shortest distance from the source

When the algorithm starts, the shortest distance from the given source node (**A**) to any of the nodes is unknown. Thus, we initially set the distance to all other nodes to infinity, with the exception of node **A**, as the distance from node **A** to node **A** is 0. No prior nodes have been visited when the algorithm begins. Therefore, we mark the previous node column of node **A** as **None**.

In *step 1* of the algorithm, we start by examining the adjacent nodes to node **A**. To find the shortest distance from node **A** to node **B**, we need to find the distance from the start node to the previous node of node **B**, which happens to be **A**, and add it to the distance from node **A** to node **B**. We do this for the other adjacent nodes of **A**, these being **B**, **E**, and **D**. This is shown in *Figure 3.10*:

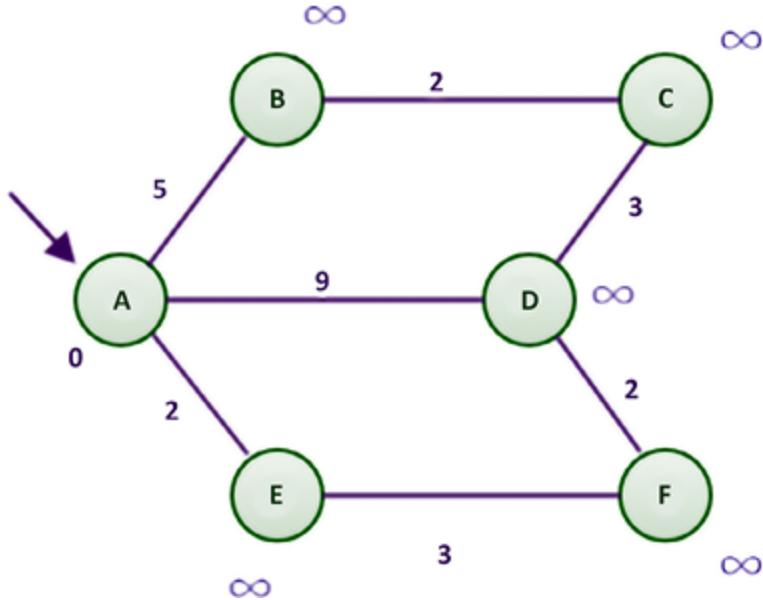


Figure 3.10: A sample graph for Dijkstra's algorithm

Firstly, we take the adjacent node **E** as its distance from node **A** is the minimum; the distance from the start node (**A**) to the previous node (**None**) is 0, and the distance from the previous node to the current node (**E**) is **2**.

This sum is compared with the data in the shortest distance column of node **E** (refer to *Table 3.3*). Since **2** is less than infinity (∞), we replace ∞ with the smaller of the two, in other words, **2**. Similarly, the distance from node **A** to nodes **B** and **D** is compared with the existing shortest distance to these nodes from node **A**. Any time the shortest distance of a node is replaced by a smaller value, we need to update the previous node column for all the adjacent nodes of the current node.

After this, we mark node **A** as visited (represented in blue in *Figure 3.11*):

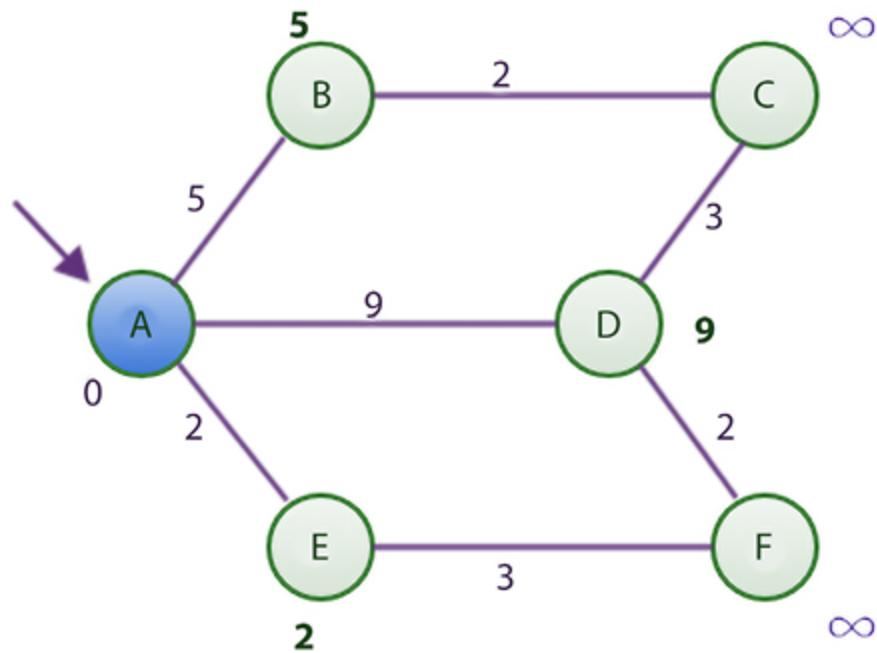


Figure 3.11: Shortest distance graph after visiting node A using Dijkstra's algorithm

At the end of step 1, the table looks like that shown in Table 3.3, in which the shortest distance from node A to nodes B, D, and E are updated.

Node	Shortest distance from source	Previous node
A*	0	None
B	5	A
C	∞	None
D	9	A
E	2	A

F	∞	None
---	----------	------

Table 3.3: Shortest distance table after visiting node A

At this point, node **A** is considered visited. As such, we add node **A** to the list of visited nodes. In the table, we show that node **A** has been visited by appending an asterisk sign to it.

In the second step, we find the node with the shortest distance using *Table 3.3* as a guide. Node **E**, with its value of 2, has the shortest distance. To reach node **E**, we must visit node **A** and cover a distance of **2**.

Now, the adjacent nodes of node **E** are nodes **A** and **F**. Since node **A** has already been visited, we will only consider node **F**. To find the shortest route or distance to node **F**, we must find the distance from the starting node to node **E** and add it to the distance between nodes **E** and **F**. We can find the distance from the starting node to node **E** by looking at the shortest distance column of node **E**, which has a value of **2**. The distance from nodes **E** to **F** can be obtained from the adjacency list, which is **3**. These two total 5, which is less than infinity. Remember that we are examining the adjacent node **F**. Since there are no more adjacent nodes to node **E**, we mark node **E** as visited. Our updated table and the figure will have the following values, shown in *Table 3.4* and *Figure 3.12*:

Node	Shortest distance from source	Previous node
A*	0	None

B	5	A
C	∞	None
D	9	A
E*	2	A
F	5	E

Table 3.4: Shortest distance table after visiting node E

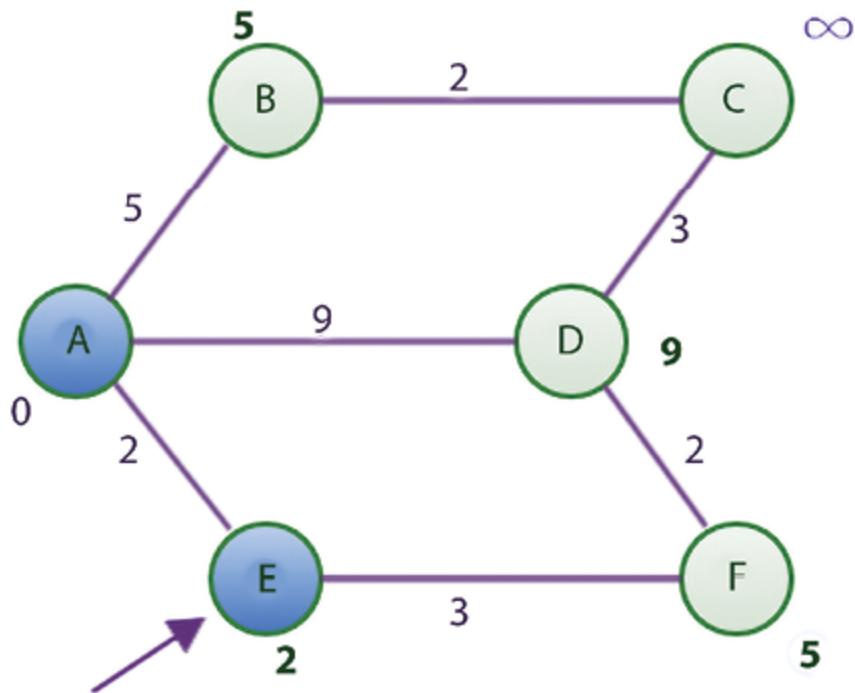


Figure 3.12: Shortest distance graph after visiting node E using Dijkstra's algorithm

After visiting node E, we find the smallest value in the Shortest distance column of Table 3.4, which is 5 for nodes B and F. Let us choose B instead of F for alphabetical reasons. The adjacent nodes of

B are nodes **A** and **C** since node **A** has already been visited. Using the rule we established earlier, the shortest distance from **A** to **C** is 7, which is computed as the distance from the starting node to node **B**, which is 5, while the distance from node **B** to **C** is 2. Since 7 is less than infinity, we update the shortest distance to 7 and update the previous node column with node **B** in *Table 3.4*.

Now, **B** is also marked as visited (represented in blue in *Figure 3.13*).

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C	7	B
D	9	A
E*	2	A
F	5	E

Table 3.5: Shortest distance table after visiting node B

The new state of the table is as follows, in *Table 3.5*:

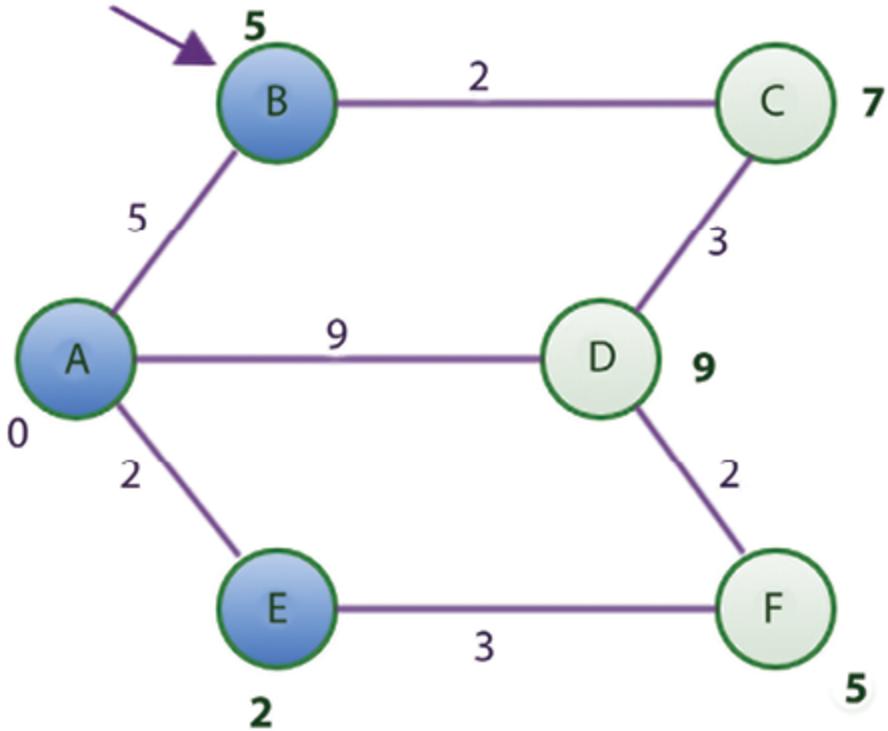


Figure 3.13: Shortest distance graph after visiting node B using Dijkstra's algorithm

The node with the shortest distance yet unvisited is node **F**. The adjacent nodes to **F** are nodes **D** and **E**. Since node **E** has already been visited, we will focus on node **D**. To find the shortest distance from the starting node to node **D**, we calculate this distance by adding the distance from nodes **A** to **F** to the distance from nodes **F** to **D**. This totals 7, which is less than 9. Thus, we update the 9 with 7 and replace **A** with **F** in node **D**'s previous node column of *Table 3.5*.

Node **F** is now marked as visited (represented in blue in *Figure 3.14*).

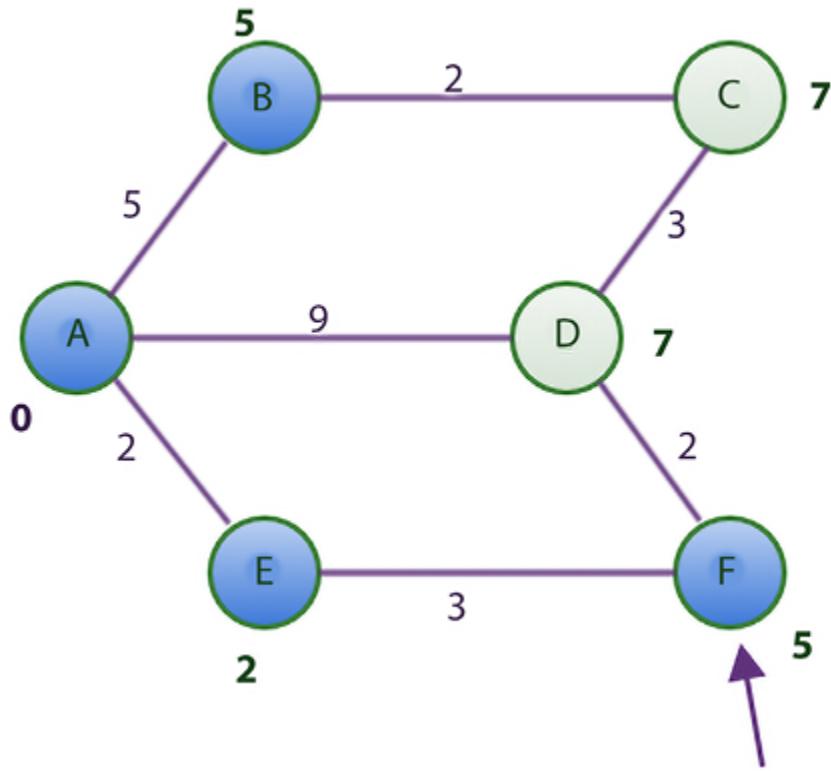


Figure 3.14: Shortest distance graph after visiting node F using Dijkstra's algorithm

Here is the updated table, as shown in Table 3.6:

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C	7	B
D	7	F
E*	2	A

F*	5	E
-----------	---	---

Table 3.6: Shortest distance table after visiting node F

Now, only two unvisited nodes are left, **C** and **D**, both with a distance cost of 7. In alphabetical order, we choose to consider node **C** because both nodes have the same shortest distance from the starting node **A**.

However, all the adjacent nodes to **C** have been visited (represented in blue in *Figure 3.15*). Thus, we have nothing to do but mark node **C** as visited. The table remains unchanged at this point.

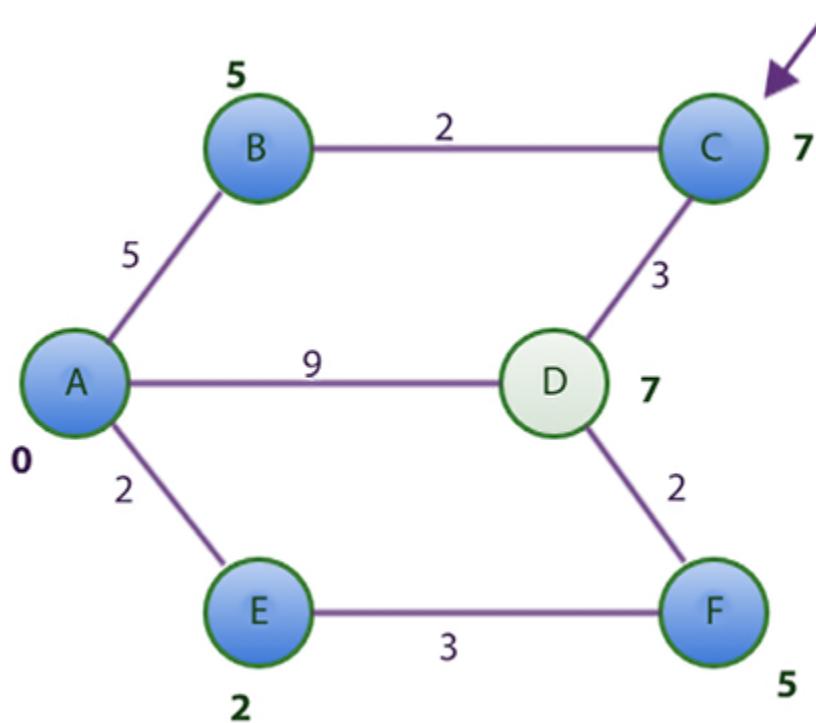


Figure 3.15: Shortest distance graph after visiting node C using Dijkstra's algorithm

Lastly, we take node **D** and find out that all its adjacent nodes have been visited too. We only mark it as visited (represented in blue in

Figure 3.16).

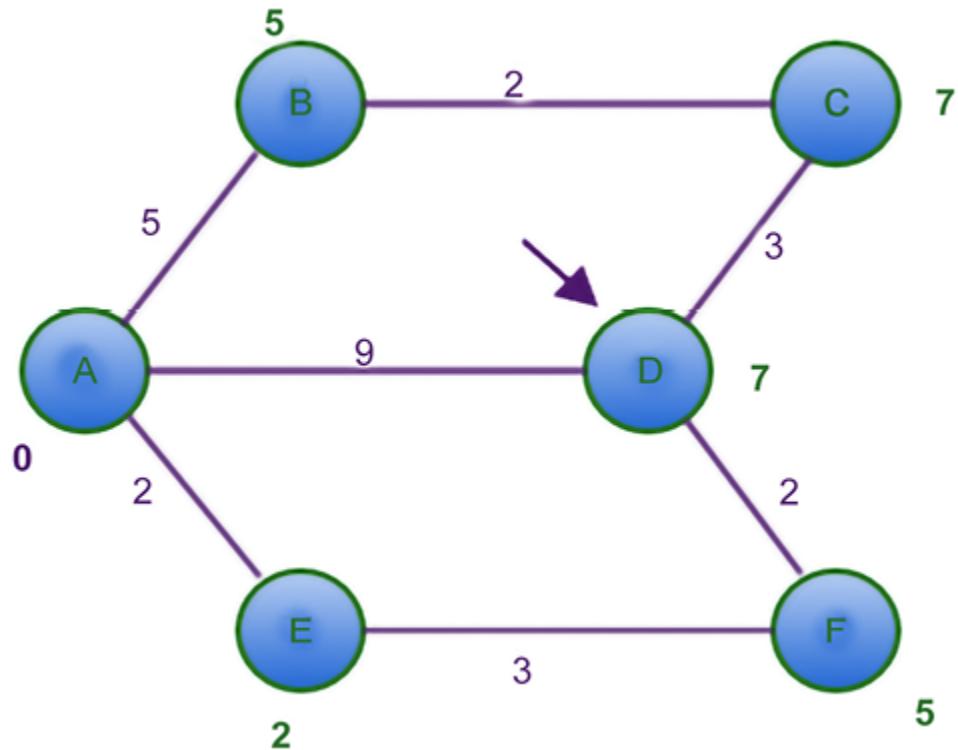


Figure 3.16: Shortest distance graph after visiting node D using Dijkstra's algorithm

The table remains unchanged, as shown in Table 3.7:

Node	Shortest distance from source	Previous node
A*	0	None
B*	5	A
C*	7	B
D*	7	F

E*	2	A
F*	5	E

Table 3.7: Shortest distance table after visiting node F

Let's verify *Table 3.7* with our initial graph. From the graph, we know that the shortest distance from **A** to **F** is **5**.

According to the table, the shortest distance from the source column for node **F** is **5**. This is true. It also tells us that to get to node **F**, we need to visit node **E**, and from **E** to node **A**, which is our starting node. This is actually the shortest path from node **A** to node **F**.

Now, we will discuss the Python implementation of Dijkstra's algorithm to find the shortest path. We begin the program for finding the shortest distance by representing the table that enables us to track the changes in the graph. For the initial *Figure 3.8* that we used, here is a dictionary representation of the table to accompany the graph representation we showed earlier in the section:

```
table = {
    'A': [0, None],
    'B': [float("inf"), None],
    'C': [float("inf"), None],
    'D': [float("inf"), None],
    'E': [float("inf"), None],
    'F': [float("inf"), None],
}
```

The initial state of the table uses `float("inf")` to represent infinity. Each key in the dictionary maps to a list. At the first index of the list,

the shortest distance from the source, node A is stored. At the second index, the previous node is stored:

```
DISTANCE = 0
PREVIOUS_NODE = 1
INFINITY = float('inf')
```

Here, the shortest path's column index is referenced by `DISTANCE`. The previous node column's index is referenced by `PREVIOUS_NODE`.

Firstly, we discuss the helper methods that we will be using while implementing the main function to find the shortest path, in other words, `find_shortest_path`. The first helper method is

`get_shortest_distance`, which returns the shortest distance of a node from the source node:

```
def get_shortest_distance(table, vertex):
    shortest_distance = table[vertex][DISTANCE]
    return shortest_distance
```

The `get_shortest_distance` function returns the value stored in index 0 of the table. At that index, we always store the shortest distance from the starting node up to `vertex`. The `set_shortest_distance` function only sets this value as follows:

```
def set_shortest_distance(table, vertex, new_distance):
    table[vertex][DISTANCE] = new_distance
```

When we update the shortest distance of a node, we update its previous node using the following method:

```
def set_previous_node(table, vertex, previous_node):
    table[vertex][PREVIOUS_NODE] = previous_node
```

Remember that the `PREVIOUS_NODE` constant equals 1. In the table, we store the value of `previous_node` at `table[vertex][PREVIOUS_NODE]`. To find the distance between any two nodes, we use the `get_distance` function:

```
def get_distance(graph, first_vertex, second_vertex):
    return graph[first_vertex][second_vertex]
```

The last helper method is the `get_next_node` function:

```
def get_next_node(table, visited_nodes):
    unvisited_nodes = list(set(table.keys()).difference(set(visit
assumed_min = table[unvisited_nodes[0]][DISTANCE]
min_vertex = unvisited_nodes[0]
for node in unvisited_nodes:
    if table[node][DISTANCE] < assumed_min:
        assumed_min = table[node][DISTANCE]
        min_vertex = node
return min_vertex
```

The `get_next_node` function resembles a function to find the smallest item in a list. The function starts off by finding the unvisited nodes in our table by using `visited_nodes` to obtain the difference between the two sets of lists. The very first item in the list of `unvisited_nodes` is assumed to be the smallest in the shortest distance column of `table`.

If a lesser value is found while the `for` loop runs, `min_vertex` will be updated. The function then returns `min_vertex` as the unvisited vertex or node with the smallest shortest distance from the source.

Now all is set up for the main function of the algorithm, in other words, `find_shortest_path`, as shown here:

```
def find_shortest_path(graph, table, origin):
    visited_nodes = []
    current_node = origin
    starting_node = origin
    while True:
        adjacent_nodes = graph[current_node]
        if set(adjacent_nodes).issubset(set(visited_nodes)):
            # Nothing here to do. All adjacent nodes have been visited
            pass
        else:
            unvisited_nodes =
                set(adjacent_nodes).difference(set(visited_nodes))
            for vertex in unvisited_nodes:
                distance_from_starting_node =
                    get_shortest_distance(table, vertex)
                if distance_from_starting_node == INFINITY and
                   current_node == starting_node:
                    total_distance = get_distance(graph, vertex,
                                                   current_node)
                else:
                    total_distance = get_shortest_distance (table,
                     current_node) + get_distance(graph, current_node,
                     vertex)
                if total_distance < distance_from_starting_node:
                    set_shortest_distance(table, vertex,
                                          total_distance)
                    set_previous_node(table, vertex, current_node)
            visited_nodes.append(current_node)
            #print(visited_nodes)
            if len(visited_nodes) == len(table.keys()):
                break
            current_node = get_next_node(table,visited_nodes)
    return (table)
```

In the preceding code, the function takes the graph, represented by the adjacency list, the table, and the starting node as input parameters. We keep the list of visited nodes in the `visited_nodes` list. The `current_node` and `starting_node` variables both point to the node in the graph that we choose to make our starting node. The `origin` value is the reference point for all other nodes with respect to finding the shortest path.

The main process of the function is implemented by the `while` loop. Let's break down what the `while` loop is doing. In the body of the `while` loop, we consider the current node in the graph that we want to investigate and initially get all the adjacent nodes of the current node with `adjacent_nodes = graph[current_node]`. The `if` statement is used to find out whether all the adjacent nodes of `current_node` have been visited.

When the `while` loop is executed for the first time, `current_node` will contain node A and `adjacent_nodes` will contain nodes B, D, and E. Furthermore, `visited_nodes` will be empty. If all nodes have been visited, we only move on to the statements further down the program, otherwise, we begin a whole new step.

The `set(adjacent_nodes).difference(set(visited_nodes))` statement returns the nodes that have not been visited. The loop iterates over this list of unvisited nodes:

```
distance_from_starting_node = get_shortest_distance(table, vertex
```

The `get_shortest_distance(table, vertex)` helper method will return the value stored in the shortest distance column of our table, using one

of the unvisited nodes referenced by `vertex`:

```
if distance_from_starting_node == INFINITY and current_node == starting_node:  
    total_distance = get_distance(graph, vertex, current_node)
```

When we are examining the adjacent nodes of the starting node, `distance_from_starting_node == INFINITY` and `current_node == starting_node` will evaluate to `True`, in which case we only have to find the distance between the starting node and `vertex` by referencing the graph:

```
total_distance = get_distance(graph, vertex, current_node)
```

The `get_distance` method is another helper method we use to obtain the value (distance) of the edge between `vertex` and `current_node`. If the condition fails, then we assign to `total_distance` the sum of the distance from the starting node to `current_node` and the distance between `current_node` and `vertex`.

Once we have our total distance, we need to check whether `total_distance` is less than the existing data in the shortest distance column of our table. If it is less, then we use the two helper methods to update that row:

```
if total_distance < distance_from_starting_node:  
    set_shortest_distance(table, vertex, total_distance)  
    set_previous_node(table, vertex, current_node)
```

At this point, we add `current_node` to the list of visited nodes:

```
visited_nodes.append(current_node)
```

If all nodes have been visited, then we must exit the `while` loop. To check whether this is the case, we compare the length of the `visited_nodes` list with the number of keys in our table. If they have become equal, we simply exit the `while` loop.

The `get_next_node` helper method is used to fetch the next node to visit. It is this method that helps us find the minimum value in the shortest distance column from the starting nodes using our table. The whole method ends by returning the updated table. To print the table, we use the following statements:

```
shortest_distance_table = find_shortest_path(graph, table, 'A')
for k in sorted(shortest_distance_table):
    print("{} - {}".format(k, shortest_distance_table[k]))
```

This is the output for the preceding code snippet:

```
A - [0, None]
B - [5, 'A']
C - [7, 'B']
D - [7, 'F']
E - [2, 'A']
F - [5, 'E']
```

The running time complexity of Dijkstra's algorithm depends on how the vertices are stored and retrieved. Generally, the min-priority queue is used to store the vertices of the graph, thus, the time complexity of Dijkstra's algorithm depends on how the min-priority queue is implemented.

In the first case, the vertices are stored numbered from 1 to $|V|$ in an array. Here, each operation for searching a vertex from the entire

array will take $O(V)$ time, making the total time complexity $O(V^2 V^2 + E) = O(V^2)$. Furthermore, if the min-priority queue is implemented using the Fibonacci heap, the time taken for each iteration of the loop and extracting the minimum node will take $O(|V|)$ time. Further, iterating over all the vertices' adjacent nodes and updating the shortest distance takes $O(|E|)$ time, and each priority value update takes $O(\log |V|)$ time, which makes $O(|E| + \log |V|)$. Thus, the total running time complexity of the algorithm becomes $O(|E| + |V| \log |V|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges.

Summary

Algorithm design techniques are very important in order to formulate, understand, and develop an optimal solution to a complex problem. In this chapter, we have discussed algorithm design techniques, which are very important in the field of computer science. Important categories of algorithm design, such as dynamic programming, greedy approach, and divide and conquer, we discussed in detail along with implementations of important algorithms.

The dynamic programming and divide-and-conquer techniques are quite similar in the sense that both solve a bigger problem by combining the solutions of the sub-problems. Here, the divide-and-conquer technique partitions the problem into disjointed sub-problems, solving them recursively, and then combines the solutions of the sub-problems to obtain the solution of the original problem, whereas, in dynamic programming, this technique is employed

when sub-problems overlap, and recomputation of the same subproblem is avoided. Furthermore, in the greedy approach-based algorithm design technique, at each step in the algorithm, the best choice is taken that looks likely to attain the solution.

In the next chapter, we will be discussing important data structures such as [Linked Lists](#) and [Pointer Structures](#).

Exercises

1. Which of the following options will be correct when a top-down approach of dynamic programming will be applied to solve a given problem related to the space and time complexity?
 - a. It will increase both time and space complexity.
 - b. It will increase the time complexity, and decrease the space complexity
 - c. It will increase the space complexity, and decrease the time complexity
 - d. It will decrease both time and space complexities.
2. Dijkstra's single shortest path algorithm is applied on edge weighted directed graph shown in Figure 3.17. What will be the order of the nodes for the shortest path distance path (Assume A as source) ?

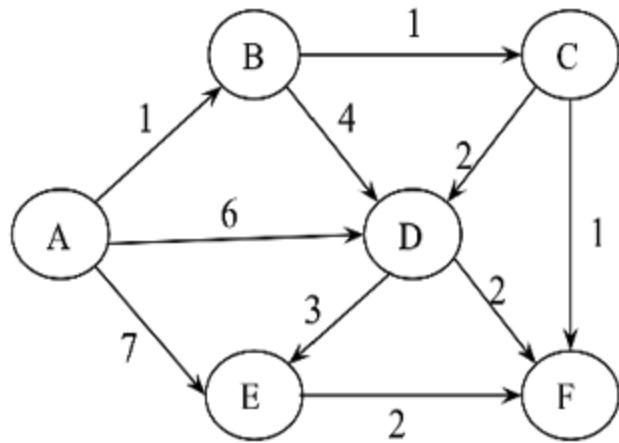


Figure 3.17: An edge-weighted directed graph

3. Consider the weights and values of the items listed in *Table 3.8*. Note that there is only one unit of each item.

Item	Weight	Value
A	2	10
B	10	8
C	4	5
D	7	6

Table 3.8: The weights and values of different items

We need to maximize the value; the maximum weight should be 11 kg. No item may be split. Establish the values of the items using a greedy approach.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



4

Linked Lists

Python's list implementation is quite powerful and can encompass several different use cases. We have discussed the built-in data structures of **lists** in Python in *Chapter 1, Python Data Types and Structures*. Most of the time, Python's built-in implementation of a list data structure is used to store data using a linked list. In this chapter, we will understand how linked lists work along with their internals.

A linked list is a data structure where the data elements are stored in a linear order. Linked lists provide efficient storage of data in linear order through pointer structures. Pointers are used to store the memory address of data items. They store the data and location, and the location stores the position of the next data item in the memory.

The focus of this chapter will be the following:

- Arrays
- Introducing linked lists
- Singly linked lists
- Doubly linked lists
- Circular lists
- Practical applications of linked lists

Before discussing linked lists, let us first discuss an array, which is one of the most elementary data structures.

Arrays

An array is a collection of data items of the same type, whereas a linked list is a collection of the same data type stored sequentially and connected through pointers. In the case of lists, the data elements are stored in different memory locations, whereas the array elements are stored in contiguous memory locations.

An array stores the data of the same data type and each data element in the array is stored in contiguous memory locations. Storing multiple data values of the same type makes it easier and faster to compute the position of any element in the array using **offset** and **base address**. The term *base address* refers to the address of memory location where the first element is stored, and offset refers to an integer indicating the displacement between the first element and a given element.

Figure 4.1 demonstrates an array holding a sequence of seven integer values that are stored sequentially in contiguous memory locations. The first element (data value 3) is stored at index 0, the second element at index position 1, and so on.

3	11	7	1	4	2	1
Indexes →	0	1	2	3	4	5

Figure 4.1: Representation of a one-dimensional array

To store, traverse, and access array elements is very fast as compared to lists since elements can be accessed randomly using their index positions, whereas in the case of a linked list, the elements are accessed sequentially. Therefore, if the data to be stored in the array is large and the system has low memory, the array data structure will not be a good choice to store the data because it is difficult to allot a large block of memory locations. The array data structure has further limitations in that it has a static size that has to be declared at the time of creation.

In addition, the insertion and deletion operations in array data structures are slow as compared to linked lists. This is because it is difficult to insert an element in an array at a given location since all data elements after that desired position must be shifted and then new elements inserted in between. Thus, array data structures are suitable when we want to do a lot of accessing of elements and fewer insertion and deletion operations, whereas linked lists are suitable in applications where the size of the list is not fixed, and a lot of insertion and deletion operations will be required.

Introducing linked lists

The linked list is an important and popular data structure with the following properties:

1. The data elements are stored in memory in different locations that are connected through pointers. A pointer is an object that can store the memory address of a variable, and each data element points to the next data element and so on until the last element, which points to `None`.

2. The length of the list can increase or decrease during the execution of the program.

Contrary to arrays, linked lists store data items sequentially in different locations in memory, wherein each data item is stored separately and linked to other data items using pointers. Each of these data items is called a node. More specifically, a node stores the actual data and a pointer. In *Figure 4.2*, nodes A and B store the data independently, and node A is connected to node B.

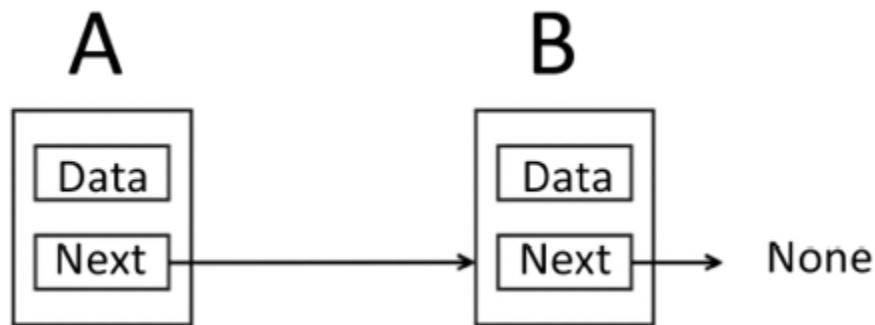


Figure 4.2: A linked list with two nodes

Moreover, the nodes can have links to other nodes based differently on how we want to store the data, and on which basis we will learn various kinds of data structures, such as singular linked lists, doubly linked lists, circular link lists, and trees.

Nodes and pointers

A node is a key component of several data structures such as linked lists. A node is a container of data, together with one or more links to other nodes where a link is a pointer.

To begin with, let us consider an example of creating a linked list of two nodes that contains data (for example, strings). For this, we first declare the variable that stores the data along with pointers that point to the next variable. Consider the example in the following *Figure 4.3*, in which there are two nodes. The first node has a pointer to the string (**eggs**), and another node pointing to the **ham** string.

Furthermore, the first node that points to the **eggs** string has a link to another node. Pointers are used to store the address of a variable, and since the string is not actually stored in the node, rather, the address of the string is stored in the node.

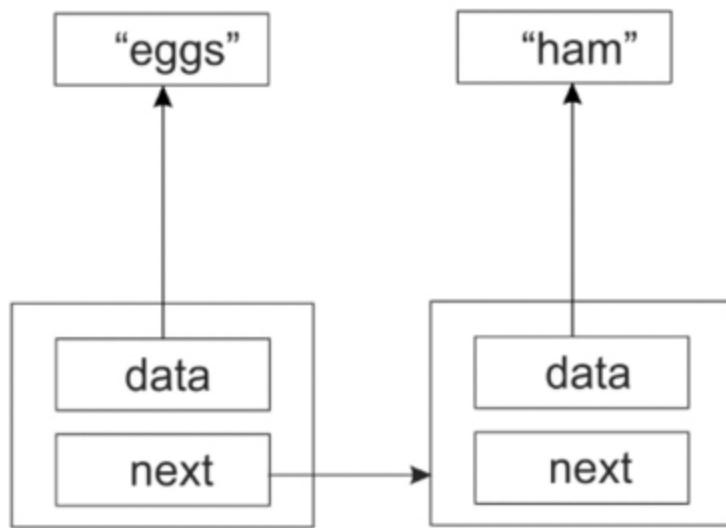


Figure 4.3: A sample linked list of two nodes

Furthermore, we can also add a new third node to this existing linked list that stores **spam** as a data value, while a second node points to the third node, as shown in *Figure 4.4*. Hence, *Figure 4.3* demonstrates the structure of three nodes having data strings, in other words, **eggs**, **ham**, and **spam**, which are stored sequentially in a linked list.

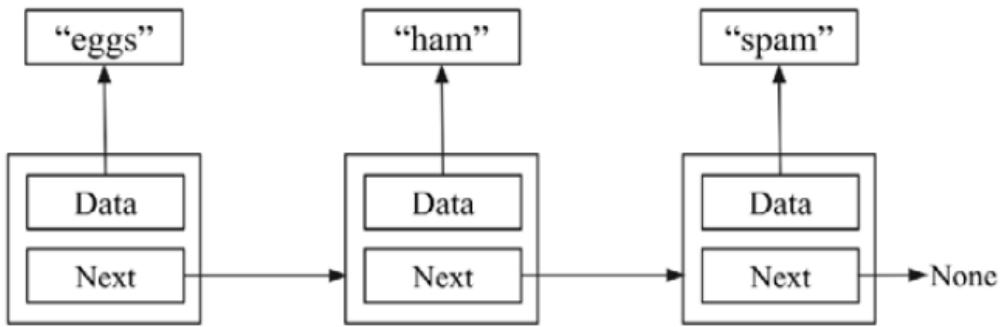


Figure 4.4: A sample linked list of three nodes

So, we have created three nodes—one containing **eggs**, one **ham**, and another **spam**. The **eggs** node points to the **ham** node, which in turn points to the **spam** node. But what does the **spam** node point to? Since this is the last element in the list, we need to make sure its next member has a value that makes this clear. If we make the last element point to nothing, then we make this fact clear. In Python, we will use the special value **None** to denote nothing. Consider *Figure 4.5*. Node **B** is the last element in the list, and thus it is pointing to **None**.

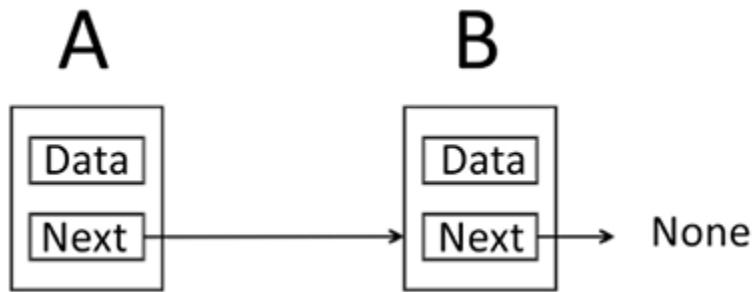


Figure 4.5: A linked list with two nodes

Let us first learn about the implementation of the node, as shown in the following code snippet:

```
class Node:  
    def __init__(self, data=None):  
        self.data = data  
        self.next = None
```

Here, the **next** pointer is initialized to `None`, meaning that unless we change the value of **next**, the node is going to be an endpoint, meaning that initially, any node that is attached to the list will be independent.

You can also add any other data items to the node class if required. If your node is going to contain customer data, then create a `Customer` class and place all the data there.

There are three kinds of list—a singly linked list, a doubly linked list, and a circular linked list. First of all, let's discuss singly linked lists.

We need to learn the following operations in order to use any linked lists in real-time applications.

- Traversing the list
- Inserting a data item in the list:
 - Inserting a new data item (node) at the beginning
 - Inserting a new data item (node) at the end of the list
 - Inserting a new data item (node) in the middle/or at any given position in the list
- Deleting an item from the list:
 - Deleting the first node
 - Deleting the last node
 - Deleting a node in the middle/or at any given position in the list

We will be discussing these important operations on different types of linked lists in subsequent subsections, along with their implementations, using Python. Let us start with singly linked lists.

Singly linked lists

A linked list (also called a singly linked list) contains a number of nodes in which each node contains data and a pointer that links to the next node. The link of the last node in the list is `None`, which indicates the end of the list. Refer to the following linked list in *Figure 4.6*, in which a sequence of integers is stored.

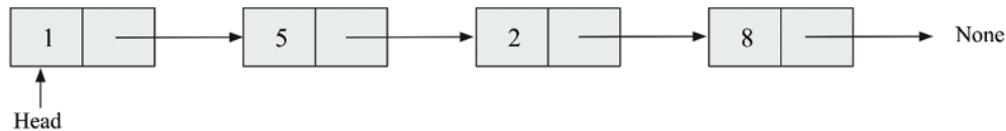


Figure 4.6: An example of a singly linked list

Next, we discuss how to create a singly linked list, and how to traverse it.

Creating and traversing

In order to implement the singly linked list, we can use the node class that we created in the previous section. For example, we create three nodes, `n1`, `n2`, and `n3`, that store three strings:

```
n1 = Node('eggs')
n2 = Node('ham')
n3 = Node('spam')
```

Next, we link the nodes sequentially to form the linked list. For example, in the following code, node `n1` is pointing to node `n2`, node `n2` is pointing to node `n3`, and node `n3` is the last node, and is pointing to `None`:

```
n1.next = n2  
n2.next = n3
```

Traversal of the linked lists means visiting all the nodes of the list, from the starting node to the last node. The process of traversing the singly linked list begins with the first node, displaying the data of the current node, following the pointers, and finally stopping when we reach the last node.

To implement the traversal of the linked list, we start by setting the `current` variable to the first item (starting node) in the list, and then we traverse the complete list through a loop, traversing each node as shown in the following code:

```
current = n1  
while current:  
    print(current.data)  
    current = current.next
```

In the loop, we print out the current element after which we set `current` to point to the next element in the list. We keep doing this until we reach the end of the list. The output of the preceding code for this example is:

```
eggs  
ham
```

```
spam
```

There are, however, several problems with this simplistic list implementation:

- It requires too much manual work by the programmer
- Too much of the inner workings of the list is exposed to the programmer

So, let us discuss a better and more efficient way of traversing the linked list.

Improving list creation and traversal

As you will notice in the earlier example of the list traversal, we are exposing the node class to the client/user. However, the client node should not interact with the node object. We need to use `node.data` to get the contents of the node, and `node.next` to get the next node. We can access the data by creating a method that returns a generator, which can be done using the `yield` keyword in Python. The updated code snippet for list traversal is as follows:

```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val
```

Here, the `yield` keyword is used to return from a function while saving the states of its local variables to enable the function to resume from where it left off. Whenever the function is called again,

the execution starts from the last yield statement. Any function that contains a yield keyword is termed a **generator**.

Now, list traversal is much simpler. We can completely ignore the fact that there is anything called a node outside of the list:

```
for word in words.iter():
    print(word)
```

Notice that since the `iter()` method yields the data member of the node, our client code doesn't need to worry about that at all.

A singly linked list can be created using a simple class to hold the list. We start with a constructor that holds a reference to the very first node in the list (that is `head` in the following code). Since this list is initially empty, we will start by setting this reference to `None`:

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None
```

In the preceding code, we start with an empty list that points to `None`. Now, new data elements can be appended/added to this list.

Appending items

The first operation that we need to perform is to `append` items to the list. This operation is also called an `insertion` operation. Here we get a chance to hide the `Node` class away. The user of the list class should never have to interact with `Node` objects.

Appending items to the end of a list

Let's have a look at the Python code for creating a linked list where we append new elements to the list using the `append()` method, as shown here:

The first shot at an `append()` method may look like this:

```
class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0
    def append(self, data):
        # Encapsulate the data in a Node
        node = Node(data)
        if self.head is None:
            self.head = node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = node
```

Here, in this code, we encapsulate data in a node so that it has the next pointer attribute. From here, we check if there are any existing nodes in the list (that is, whether `self.head` points to a `Node`). If there is `None`, this means that initially, the list is empty and the new node will be the first node. So, we make the new node the first node of the list; otherwise, we find the insertion point by traversing the list to the last node and updating the next pointer of the last node to the new node. This working is depicted in *Figure 4.7*.

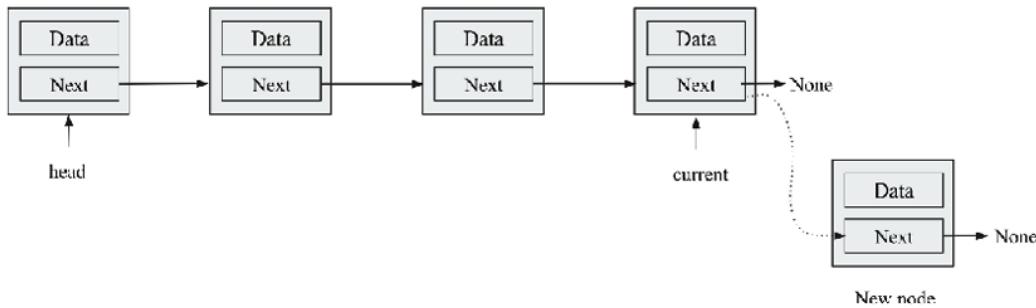


Figure 4.7: Inserting a node at the end of the list in a singly linked list

Consider the following example code to append three nodes:

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
```

List traversal will work as we discussed before. You will get the first element of the list from the list itself, and then traverse the list through the `next` pointer:

```
current = words.head
while current:
    print(current.data)
    current = current.next
```

Still, this implementation is not very efficient, and there is a drawback with the `append` method. In this, we have to traverse the entire list to find the insertion point. This may not be a problem when there are just a few items in the list, but it will be very inefficient when the list is long, as it will have to traverse the whole list to add an item every time. Let us discuss a better implementation of the `append` method.

For this, the idea is that we not only have a reference to the first node in the list but also have one more variable in the node that references the last node of the list. That way, we can quickly append a new node at the end of the list. The worst-case running time of the append operation can be reduced from $O(n)$ to $O(1)$ using this method. We must ensure that the previous last node points to the new node that is to be appended to the list.

Here is our updated code:

```
class SinglyLinkedList:
    def __init__(self):
        self.tail = None
        self.head = None
        self.size = 0
    def append(self, data):
        node = Node(data)
        if self.tail:
            self.tail.next = node
            self.tail = node
        else:
            self.head = node
            self.tail = node
```



Take note of the convention being used. The point at which we append new nodes is through `self.tail`. The `self.head` variable points to the first node in the list.

In this code, a new node can be appended in the end through a `tail` pointer by making a link from the last node to the new node. *Figure 4.8* shows the workings of the preceding code.

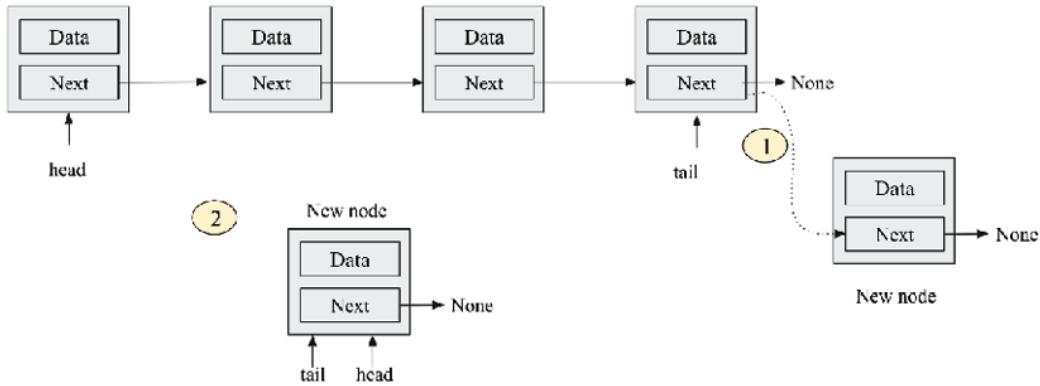


Figure 4.8: Illustrating the insertion of a node at the end of a linked list

In Figure 4.8, step 1 shows the addition of the new node at the end, and step 2 shows when the list is empty. In that case, `head` is made the new node, with `tail` pointing to that node.

Furthermore, the following code snippet shows the workings of the code:

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')

current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
eggs
ham
spam
```

Appending items at intermediate positions

To append or insert an element in an existing linked list at a given position, firstly, we have to traverse the list to reach the desired position where we want to insert an element. An element can be inserted in between two successive nodes using two pointers (`prev` and `current`).

A new node can easily be inserted in between two existing nodes by updating these links, as shown in *Figure 4.9*.

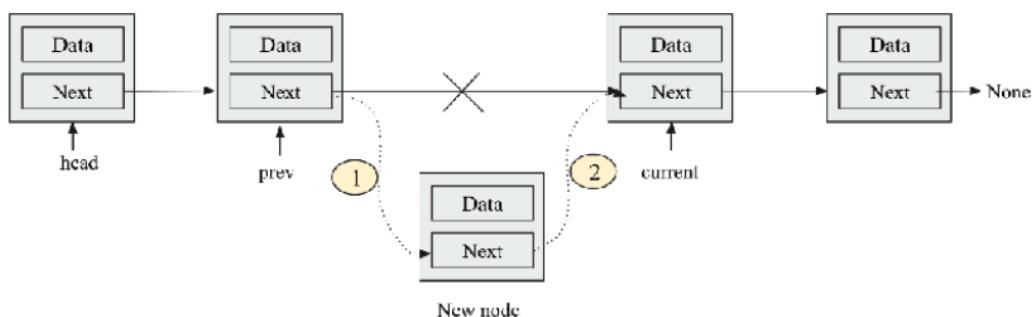


Figure 4.9: Insertion of a node between two successive nodes in a linked list

When we want to insert a node in between two existing nodes, all we have to do is update two links. The previous node points to the new node, and the new node should point to the successor of the previous node.

Let's look at the complete code below to add a new element at a given index position:

```
class SinglyLinkedList:
    def __init__(self):
        self.tail = None
        self.head = None
        self.size = 0
    def append_at_a_location(self, data, index):
```

```

current = self.head
prev = self.head
node = Node(data)
count = 1
while current:
    if count == 1:
        node.next = current
        self.head = node
        print(count)
        return
    elif index == index:
        node.next = current
        prev.next = node
        return
    count += 1
    prev = current
    current = current.next
if count < index:
    print("The list has less number of elements")

```

In the preceding code, we start from the first node and move the current pointer to reach the index position where we want to add a new element, and then we update the node pointers accordingly. In the `if` condition, firstly, we check whether the index position is `1`. In that case, we have to update the nodes as we are adding the new node at the start of the list. Therefore, we have to make the new node a `head` node. Next, in the `else` part, we check whether we have reached the required index position by comparing the value of `count` and `index`. If both values are equal, we add a new node in between nodes indicated by `prev` and `current` and update the pointers accordingly. Finally, we print an appropriate message if the required index position is greater than the length of the linked list.

The following code snippet uses the `append` method to add a “new” data element at an index position of `2` in the existing linked list:

```
words = SinglyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
current = words.head
while current:
    print(current.data)
    current = current.next
words.append_at_a_location('new', 2)
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
egg
new
ham
spam
```

It is important to note that the condition where we may want to insert a new element can change depending upon the requirement, so let's say we want to insert a new element just before an element that has the same data value. In that case, the code to `append_at_a_position` will be as follows:

```
def append_at_a_location(self, data):
    current = self.head
    prev = self.head
    node = Node(data)
    while current:
        if current.data == data:
            node.next = current
            prev.next = node
```

```
prev = current
current = current.next
```

We can now use the preceding code to insert a new node at an intermediate position:

```
words.append_at_a_location('ham')
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
egg
ham
ham
spam
```

The worst-case time complexity of the `insert` operation is $O(1)$ when we have an additional pointer that points to the last node.

Otherwise, when we do not have the link to the last node, the time complexity will be $O(n)$ since we have to traverse the list to reach the desired position and in the worst case, we may have to traverse all the n nodes in the list.

Querying a list

Once the list is created, we may require some quick information about the linked list, such as the size of the list, and occasionally to establish whether a given data item is present in the list.

Searching an element in a list

We may also need to check whether a list contains a given item. This can be implemented using the `iter()` method, which we have already seen in the previous section while traversing the linked list. Using that, we write the search method as follows:

```
def search(self, data):
    for node in self.iter():
        if data == node:
            return True
    return False
```

In the above code, each pass of the loop compares the data to be searched with each data item in the list one by one. If a match is found, `True` is returned, otherwise `False` is returned.

If we run the following code for searching a given data item:

```
print(words.search('ssspam'))
print(words.search('spam'))
```

The output of the preceding code is as follows:

```
False
True
```

Getting the size of the list

It is important to get the size of the list by counting the number of nodes. One way to do it is by traversing the entire list and increasing the counter as we go along:

```
def size(self):
    count = 0
    current = self.head
    while current:
        count += 1
        current = current.next
    return count
```

The above code is very similar to what we did while traversing the linked list. Similarly, in this code, we traverse the nodes of the list one by one and increase the `count` variable. However, list traversal is potentially an expensive operation that we should avoid wherever we can.

So instead, we can opt for another method in which we can add a `size` member to the `SinglyLinkedList` class, initializing it to `0` in the constructor, as shown in the following code snippet:

```
class SinglyLinkedList:
    def __init__(self):
        self.head = data
        self.size = 0
```

Because we are now only reading the `size` attribute of the node object, and not using a loop to count the number of nodes in the list, we reduce the worst-case running time from `O(n)` to `O(1)`.

Deleting items

Another common operation on a linked list is to delete nodes. There are three possibilities that we may encounter in order to delete a node from the singly linked list.

Deleting the node at the beginning of the singly linked list

Deleting a node from the beginning is quite easy. It involves updating the `head` pointer to the second node in the list. This can be done in two steps:

1. A temporary pointer (`current` pointer) is created that points to the first node (`head` node), as shown in *Figure 4.10*.

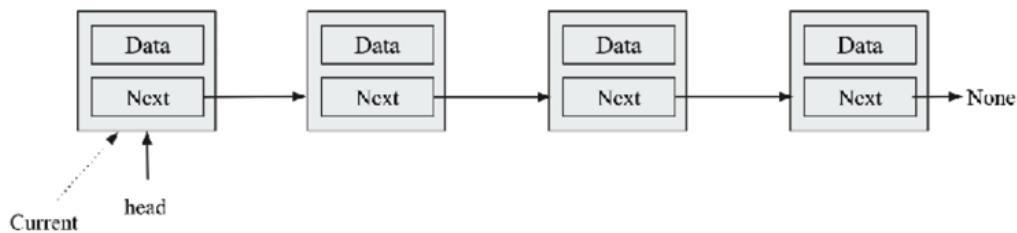


Figure 4.10: Illustration of the deletion of the first node from the linked list

2. Next, the `current` node pointer is moved to the next node and assigned to the `head` node. Now, the second node becomes the `head` node that is pointed to by the `head` pointer, as shown in *Figure 4.11*.

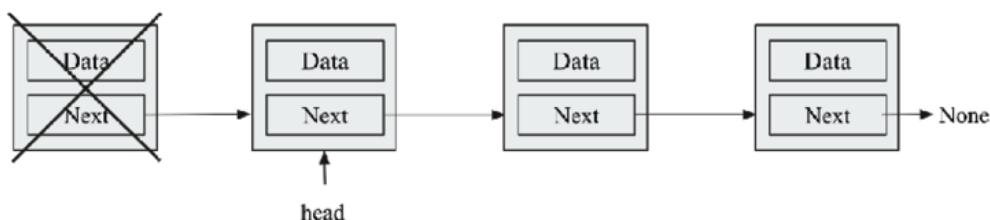


Figure 4.11: After deleting the first node, the head pointer now points to the new starting element

This can be implemented using the following Python code. In this code, initially, three data elements are added as we have done previously, and then the first node of the list is deleted:

```
def delete_first_node (self):  
    current = self.head  
    if self.head is None:  
        print("No data element to delete")  
    elif current == self.head:  
        self.head = current.next
```

In the above code, we initially check if there is no item to delete from the list, and we print the appropriate message. Next, if there is some data item in the list, we assign the `head` pointer to the temporary pointer `current` as per *step 1*, and then the `head` pointer is now pointing to the next node, assuming that we already have a linked list of three data items – “eggs”, “ham”, and “spam”:

```
words.delete_first_node()  
current = words.head  
while current:  
    print(current.data)  
    current = current.next
```

The output of the preceding code is as follows:

```
ham  
spam
```

Deleting the node at the end in the singly linked list

To delete the last node from the list, we have to first traverse the list to reach the last node. At that time, we also need an extra pointer that points to just one node before the last node, so that after

deleting the last node, the second last node can be marked as the last node. It can be implemented in the following three steps:

1. Firstly, we have two pointers, in other words, a `current` pointer that will point to the last node, and a `prev` pointer that will point to the node previous to the last node (second last node). Initially, we will have three pointers (`current`, `prev`, and `head`) pointing to the first node, as shown in *Figure 4.12*.

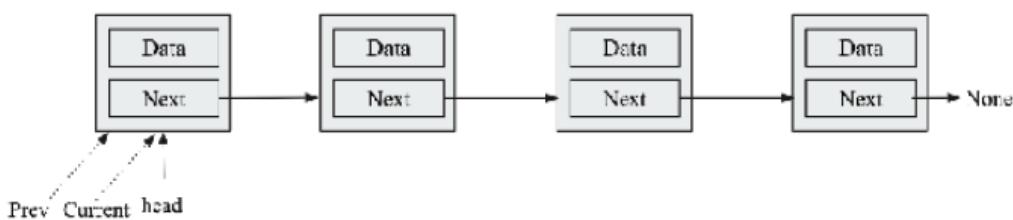


Figure 4.12: Illustration of the deletion of the end node from the linked list

2. To reach the last node, we move the `current` and `prev` pointers in such a way that the `current` pointer should point to the last node and the `prev` pointer should point to the second last node. So, we stop when the `current` pointer reaches the last node. This is shown in *Figure 4.13*.

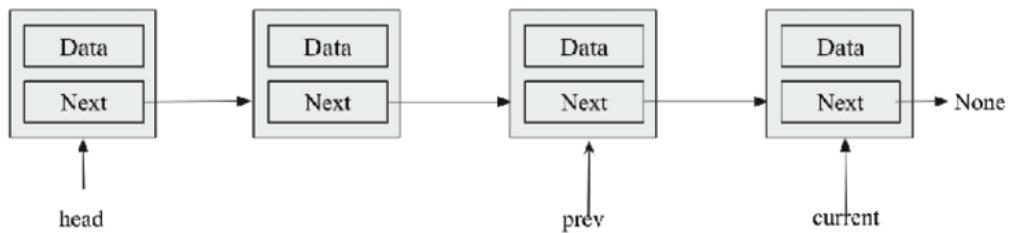


Figure 4.13: Traversal of the linked list to reach the end of the list

3. Finally, we mark the `prev` pointer to point to the second last node, which is rendered as the last node of the list by pointing

this node to `None`, as shown in *Figure 4.14*.

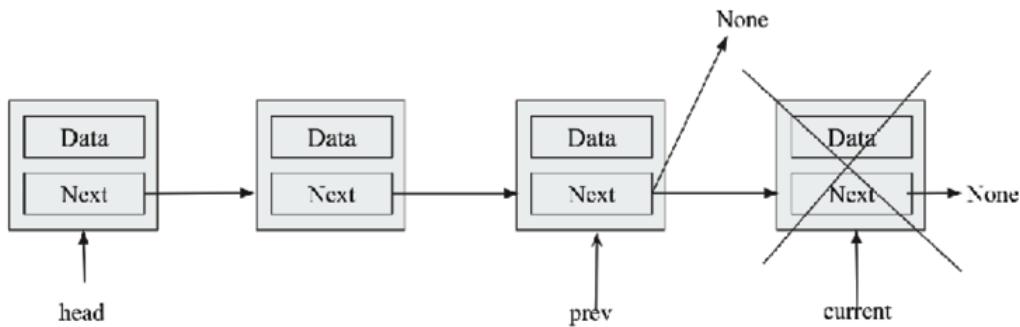


Figure 4.14: Deletion of the last node from the linked list

The implementation in Python for deleting a node from the end of the list is as follows:

```
def delete_last_node (self):
    current = self.head
    prev = self.head
    while current:
        if current.next is None:
            prev.next = current.next
            self.size -= 1
        prev = current
        current = current.next
```

In the preceding code, firstly, the `current` and `prev` pointers are assigned the `head` pointer as per *step 1*. Then, in the `while` loop, we check whether we reached the end of the list using the `current.next is None` condition. Once we reach the end of the list, we make the second last node, which is indicated by the `prev` pointer, the last node. We also decrement the size of the list. If we do not reach the end of the list, we increment the `prev` and `current` pointers in the

`while` loop in the last two lines of code. Next, let us discuss how to delete any intermediate node in a singly linked list.

Deleting any intermediate node in a singly linked list

We first have to decide how to select a node for deletion. Identifying the intermediate node to be deleted can be determined by the index number or by the data the node contains. Let us understand this concept by deleting a node depending on the data it contains.

To delete any intermediary node, we need two pointers similar to the case when we learned to delete the last node; in other words, the `current` pointer and the `prev` pointer. Once we reach the node that is to be deleted, the desired node can be deleted by making the previous node point to the next node of the node that is to be deleted. The process is provided in the following steps:

1. *Figure 4.15 shows when an intermediate node is deleted from the given linked list. In this, we can see that the initial pointers point to the first node.*

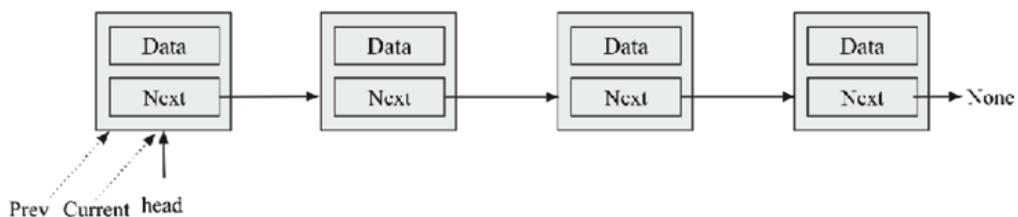


Figure 4.15: Illustration of the deletion of an intermediate node from the linked list

2. Once the node is identified, the `prev` pointer is updated to delete the node, as shown in *Figure 4.16*. The node to be deleted is

shown along with the link to those to be updated in *Figure 4.16*.

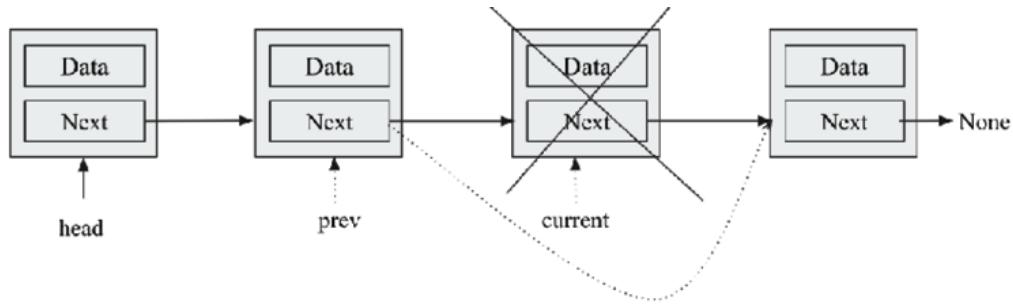


Figure 4.16: Traversing to reach the intermediate node that is to be deleted in the linked list

3. Finally, the list after deleting the node is shown in *Figure 4.17*.

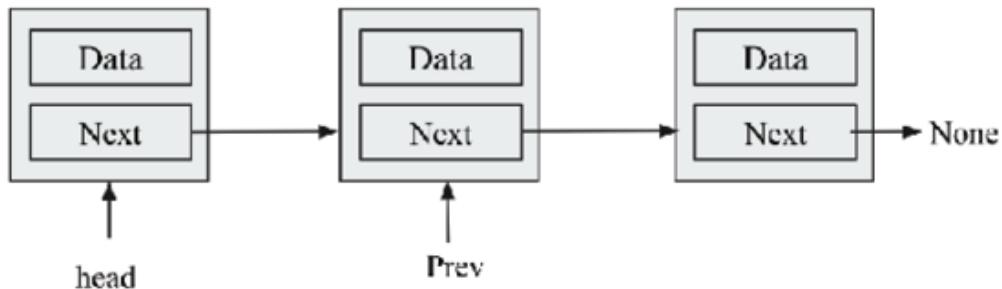


Figure 4.17: Deletion of an intermediate node from the linked list

Let's say we want to delete a data element that has the given value. For this given condition, we can first search the node to be deleted and then delete the node as per the steps discussed.

Here is what the implementation of the `delete()` method may look like:

```
def delete(self, data):
    current = self.head
    prev = self.head
    while current:
        if current.data == data:
            if current == self.head:
```

```
        self.head = current.next
    else:
        prev.next = current.next
    self.size -= 1
    return
prev = current
current = current.next
```

Assuming that we already have a linked list of three items – “eggs”, “ham”, and “spam”, the following code is for executing the delete operation, that is, deleting a data element with the value “ham” from the given linked list:

```
words.delete("ham")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the preceding code is as follows:

```
egg
spam
```

The worst-case time complexity of the `delete` operation is $O(n)$ since we have to traverse the list to reach the desired position and, in the worst-case scenario, we may have to traverse all the n nodes in the list.

Clearing a list

We may need to clear a list quickly, and there is a very simple way to do this. We can clear a list by simply clearing the pointer `head` and

tail by setting them to `None`:

```
def clear(self):
    # clear the entire list.
    self.tail = None
    self.head = None
```

In the above code, we can clear the list by assigning `None` to the `tail` and `head` pointers.

We have discussed different operations for a singly linked list, and now we will discuss the concept of doubly linked list and learn how different operations can be implemented in a doubly linked list in the next section.

Doubly linked lists

A doubly linked list is quite similar to the singly linked list in the sense that we use the same fundamental concept of nodes along with how we can store data and links together, as we did in a singly linked list. The only difference between a singly linked list and a doubly linked list is that in a singly linked list, there is only one link between each successive node, whereas, in a doubly linked list, we have two pointers—a pointer to the next node and a pointer to the previous node. See the following *Figure 4.18* of a node; there is a pointer to the next node and the previous node, which are set to `None` as there is no node attached to this node.

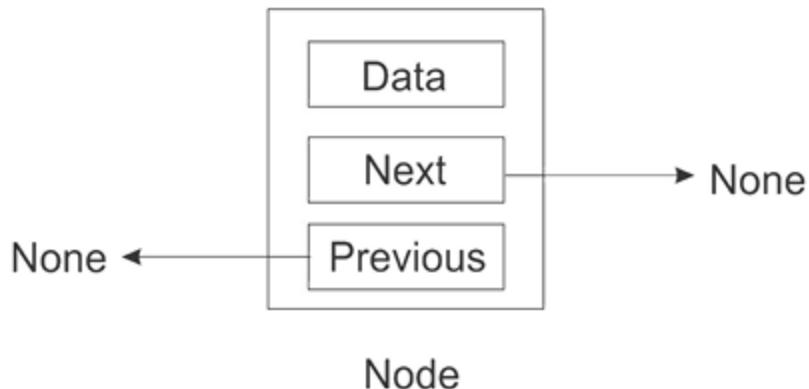


Figure 4.18: Represents a doubly linked list with a single node

A node in a singly linked list can only determine the next node associated with it. However, there is no link to go back from this referenced node. The direction of flow is only one way. In a doubly linked list, we solve this issue and include the ability not only to reference the next node, but also to reference the previous node. Consider the following *Figure 4.19* to understand the nature of the linkages between two successive nodes. Here, node **A** is referencing node **B**; in addition, there is also a link back to node **A**.

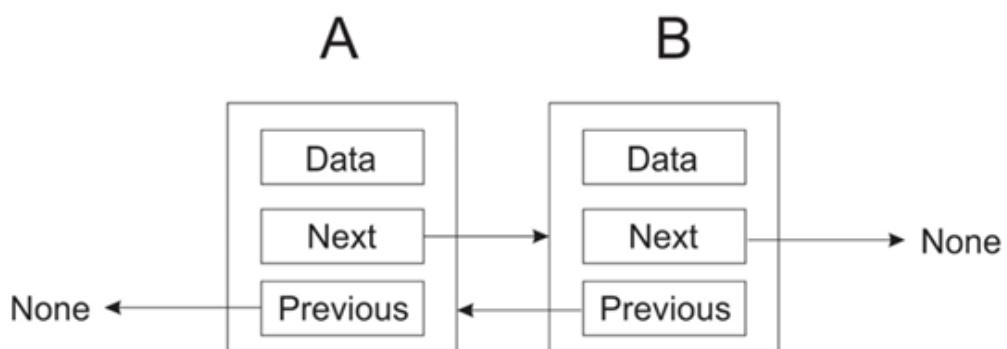


Figure 4.19: Doubly linked list with two nodes

Doubly linked lists can be traversed in any direction. A node in a doubly linked list can be easily referred to by its previous node

whenever required without having a variable to keep track of that node.

However, in a singly linked list, it may be difficult to move back to the start or beginning of the list to make some changes at the start of the list, which is very easy now in the case of a doubly linked list.

Creating and traversing

The Python code to create a doubly linked list node includes its initializing methods, the `prev` pointer, the `next` pointer, and the `data` instance variables. When a node is newly created, all these variables default to `None`:

```
class Node:  
    def __init__(self, data=None, next=None, prev=None):  
        self.data = data  
        self.next = next  
        self.prev = prev
```

The `prev` variable has a reference to the previous node, while the `next` variable keeps the reference to the next node, and the `data` variable stores the data.

Next, let's create a doubly linked list class.

The doubly linked list class has two pointers, `head` and `tail`, that will point to the start and end of the doubly linked list, respectively. In addition, for the size of the list, we set the `count` instance variable to `0`. It can be used to keep track of the number of items in the linked list. Consider the following Python code for creating a doubly linked list class:

```
class DoublyLinkedList:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
        self.count = 0
```



Here, `self.head` points to the beginner node of the list, and `self.tail` points to the last node. However, there are no fixed rules as to the naming of the head and tail node pointers.

Doubly linked lists also require functionalities that return the size of the list, insert items into the list, and delete nodes from the list. Next, we discuss different operations that can be applied to the doubly linked list. Let's start with the append operation.

Appending items

The `append` operation is used to add an element at the end of a list. An element can be appended or inserted into a doubly linked list in the following instances.

Inserting a node at beginning of the list

Firstly, it is important to check whether the `head` node of the list is `None`. If it is `None`, this means that the list is empty, otherwise the list has some nodes, and a new node can be appended to the list. If a new node is to be added to the empty list, it should have the `head` pointer pointing to the newly created node, and the tail of the list should also point to this newly created node.

The following *Figure 4.20* illustrates the `head` and `tail` pointers of the doubly linked list when a new node is added to an empty list.

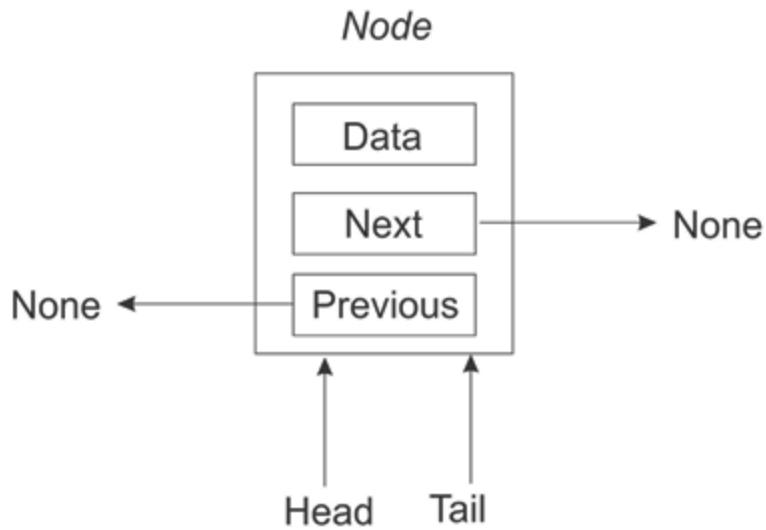


Figure 4.20: Illustration of inserting a node in an empty doubly linked list

Alternatively, we can insert or append a new node at the beginning of an existing doubly linked list, as shown in *Figure 4.21*.

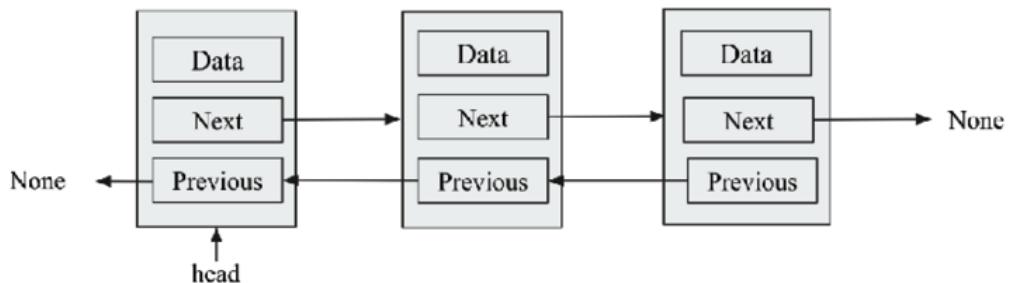


Figure 4.21: Illustration of inserting an element in a doubly linked list

The new node should be made as a new starting node of the list and that should now point to the previous `head` node.

It can be done by updating the three links, which are also shown with dotted lines in *Figure 4.22* and described as follows:

1. Firstly, the `next` pointer of a new node should point to the `head` node of the existing list
2. The `prev` pointer of the `head` node of the existing list should point to the new node
3. Finally, mark the new node as the `head` node in the list

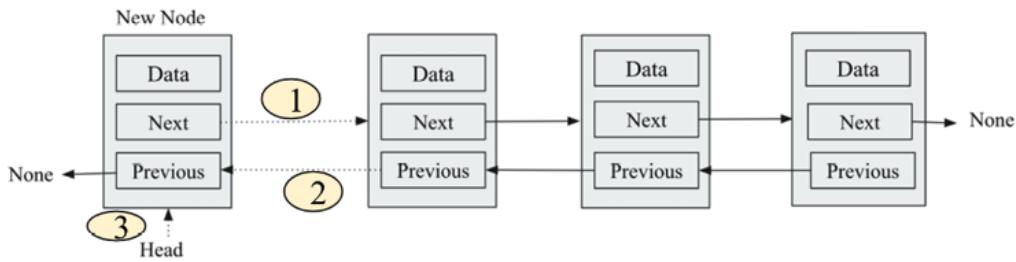


Figure 4.22: Inserting a node at the beginning of the doubly linked list

The following code is used to append/insert an item at the beginning when the list is initially empty and with an existing doubly linked list:

```
def append_at_start(self, data):
    #Append an item at beginning to the list.
    new_node = Node(data, None, None)
    if self.head is None:
        self.head = new_node
        self.tail = self.head
    else:
        new_node.next = self.head
        self.head.prev = new_node
        self.head = new_node
    self.count += 1
```

In the above code, firstly, the `self.head` condition is checked irrespective of whether the list is empty. If it is empty, then the head and tail pointers point to the newly created node. In this case, the

new node becomes the `head` node. Next, if the condition is not true, this means the list is not empty, and a new node has to be added at the beginning of the list. For this, three links are updated as shown in *Figure 4.22*, and also shown in the code in bold font. After updating these three links, finally, the size of the list is increased by 1. Furthermore, let us understand how to insert an element at the end of the doubly linked list.

Further, the following code snippet shows how we can create a double link list and append a new node at the starting of the list:

```
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
print("Items in doubly linked list before append:")
current = words.head
while current:
    print(current.data)
    current = current.next
words.append_at_start('book')
print("Items in doubly linked list after append:")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code is:

```
Items in doubly linked list before append:
egg
ham
spam
Items in doubly linked list after append:
book
egg
```

```
ham  
spam
```

In the output, we can see that the new data item “book” is added in the starting of the list.

Inserting a node at the end of the list

To append/insert a new element at the end of the doubly linked list, we will need to traverse through the list to reach the end of the list if we do not have a separate pointer pointing to the end of the list.

Here, we have a `tail` pointer that points to the end of the list.

A visual representation of the append operation to an existing list is shown in the following *Figure 4.23*.

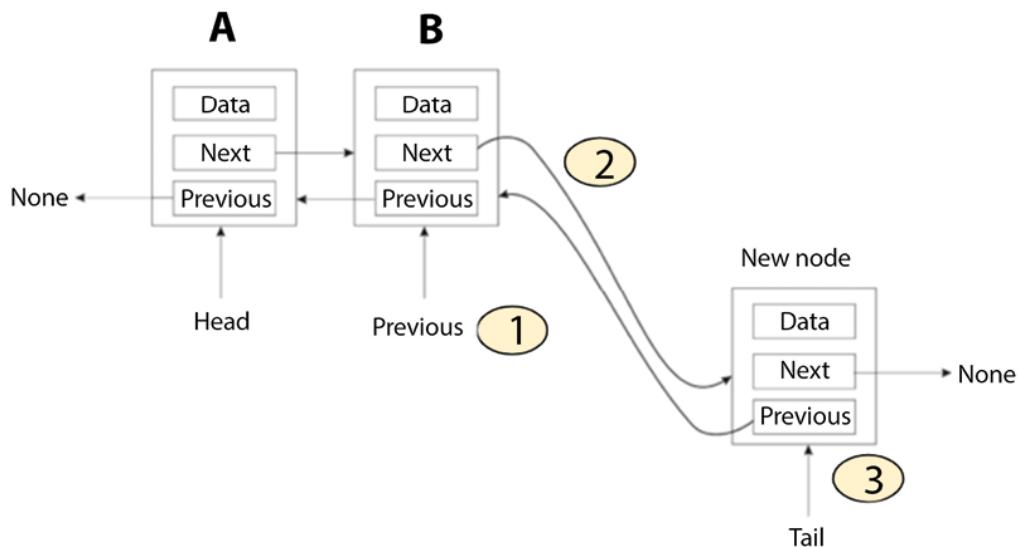


Figure 4.23: Inserting a node at the end of the list in a doubly linked list

To add a new node at the end, we update two links as follows:

1. Make the `prev` pointer of the new node point to the previous `tail` node

2. Make the previous `tail` node point to the new node
3. Finally, update the `tail` pointer so that the `tail` pointer now points to the new node

The following code is used to append an item at the end of the doubly linked list:

```
def append(self, data):  
    #Append an item at the end of the list.  
    new_node = Node(data, None, None)  
    if self.head is None:  
        self.head = new_node  
        self.tail = self.head  
    else:  
        new_node.prev = self.tail  
        self.tail.next = new_node  
        self.tail = new_node  
    self.count += 1
```

In the above code, the `if` part of the preceding program is for adding a node to the empty list; the `else` part of the preceding program will be executed if the list is not empty. If the new node is to be added to a list, the new node's previous variable is to be set to the tail of the list:

```
new_node.prev = self.tail
```

The tail's next pointer (or variable) has to be set to the new node:

```
self.tail.next = new_node
```

Lastly, we update the tail pointer to point to the new node:

```
    self.tail = new_node
```

Since an append operation increases the number of nodes by one, we increase the counter by one:

```
    self.count += 1
```

The following code snippet can be used to append a node at the end of the list:

```
print("Items in doubly linked list after append")
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
words.append('book')
print("Items in doubly linked list after adding element at end.")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code:

```
Items in doubly linked list after adding element at end.
egg
ham
spam
book
```

The worst-case time complexity of appending an element to the doubly linked list is $O(1)$ since we already have the tail pointer that points to the end of the list, and we can directly add a new element.

Next, we will discuss how to insert a node at an intermediate position of the doubly linked list.

Inserting a node at an intermediate position in the list

Inserting a new node at any given position in a doubly linked list is similar to what we discussed in a singly linked list. Let us take an example in which we insert a new element just before the element that has the same data value as the given data.

Firstly, we traverse to the position where we want to insert a new element in that situation. The `current` pointer points to the target node, while the `prev` pointer just points to the previous node of the target node, as shown in *Figure 4.24*.

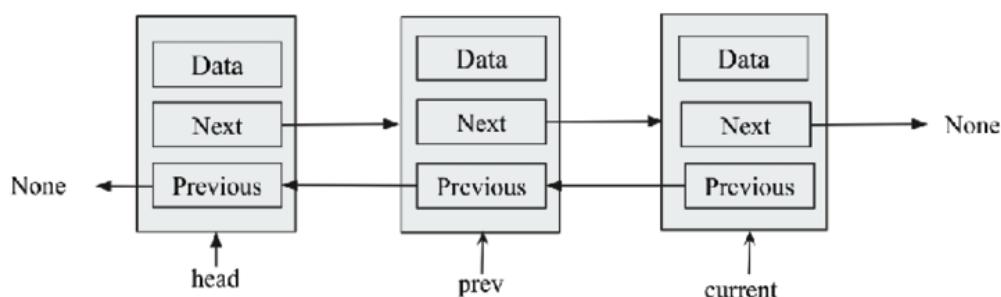


Figure 4.24: Illustration of pointers for inserting a node at an intermediate position in a doubly linked list

After reaching the correct position, a few pointers have to be added in order to add a new node. The details of these links that need to be updated (also shown in *Figure 4.25*) are as follows:

1. The `next` pointer of the new node points to the current node

2. The `prev` pointer of the new node should point to the previous node
3. The `next` pointer of the previous node should point to the new node
4. The `prev` pointer of the current node should point to the new node

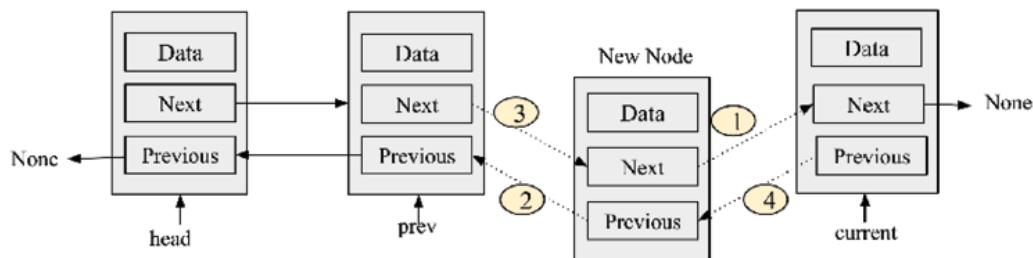


Figure 4.25: Demonstration of links that need to be updated in order to add a new node at any intermediate position in the list

Here is what the implementation of the `append_at_a_location()` method may look like:

```
def append_at_a_location(self, data):
    current = self.head
    prev = self.head
    new_node = Node(data, None, None)
    while current:
        if current.data == data:
            new_node.prev = prev
            new_node.next = current
            prev.next = new_node
            current.prev = new_node
            self.count += 1
        prev = current
        current = current.next
```

In the preceding code, firstly, the `current` and `prev` pointers are initialized by pointing to the `head` node. Then, in the `while` loop, we first reach the desired position by checking the condition. In this example, we check the data value of the current node against the data value provided by the user. Once we reach the desired position, we update four links as discussed, which are also shown in *Figure 4.25*.

The following code snippet can be used to insert an data element “`ham`” after the first occurrence of the word “`ham`” in the doubly linked list:

```
words = DoublyLinkedList()
words.append('egg')
words.append('ham')
words.append('spam')
words.append_at_a_location('ham')
print("Doubly linked list after adding an element after word \"ham\"")
current = words.head
while current:
    print(current.data)
    current = current.next
```

The output of the above code:

```
Doubly linked list after adding an element after word "ham" in the list
egg
ham
ham
spam
```

Appending at the start and end positions in a doubly linked list will have a worst-case running time complexity of $O(1)$ since we can directly append the new node, and the worst-case time complexity for appending a new node at any intermediate position will be $O(n)$ since we may have to traverse the list of n items.

Next, let us learn how to search a given item if that is present in the doubly linked list or not.

Querying a list

The search for an item in a doubly linked list is similar to the way we did it in the singly linked list. We use the `iter()` method to check the data in all the nodes. As we run a loop through all the data in the list, each node is matched with the data passed in the `contains` method. If we find the item in the list, `True` is returned, denoting that the item is found, otherwise `False` is returned, which means the item was not found in the list. The Python code for this is as follows:

```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val

def contains(self, data):
    for node_data in self.iter():
        if data == node_data:
            print("Data item is present in the list.")
            return
    print("Data item is not present in the list.")
    return
```

The following code can be used to search if a data item is present in the existing doubly linked list:

```
words = DoublyLinkedList()

words.append('egg')
words.append('ham')
words.append('spam')
words.contains("ham")
words.contains("ham2")
```

The output of the above code is as follows:

```
Data item is present in the list.
Data item is not present in the list.
```

The search operation in a doubly linked list has a running time complexity of $O(n)$ since we have to traverse the list in order to reach the desired element and, in the worst case, we may have to traverse the whole list of n items.

Deleting items

The deletion operation is easier in the doubly linked list compared to the singly linked list. Unlike in a singly linked list, where we need to traverse the linked list to reach the desired position, and we also need one more pointer to keep track of the previous node of the target node, in a doubly linked list, we don't have to do that because we can traverse in both directions.

The `delete` operation in a doubly linked list can have four scenarios, which are discussed as follows:

1. The item to be deleted is located at the start of the list
2. The item to be deleted is found at the tail end of the list
3. The item to be deleted is located anywhere at an intermediate position in the list
4. The item to be deleted is not found in the list

The node to be deleted is identified by matching the data instance variable with the data that is passed to the method. If the data matches the data variable of a node, that matching node will be deleted:

1. For the first scenario, when we have found the item to be deleted at the first position, we will have to simply update the `head` pointer to the next node. It is shown in *Figure 4.26*.

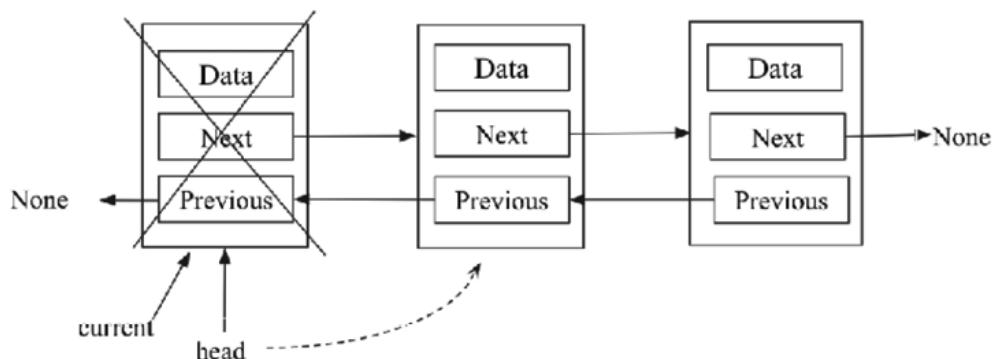


Figure 4.26: Illustration of the deletion of the first node in a doubly linked list

2. For the second scenario, when we found the item to be deleted at the last position in the list, we will have to simply update the `tail` pointer to the second last node. It is shown in *Figure 4.27*.

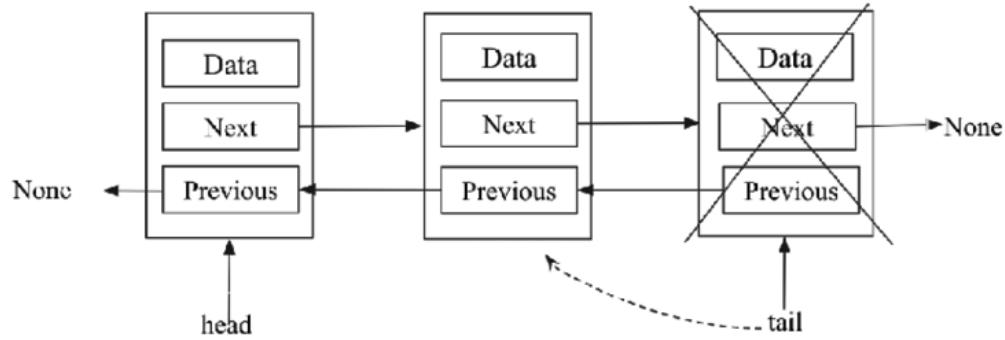


Figure 4.27: Illustration of the deletion of the last node in a doubly linked list

3. For the third scenario, we found the data item to be deleted at any intermediate position. To better understand this, consider the example shown in *Figure 4.28*. In this, there are three nodes, **A**, **B**, and **C**. To delete node **B** in the middle of the list, we will essentially make **A** point to node **C** as its next node, while making **C** point to **A** as its previous node.

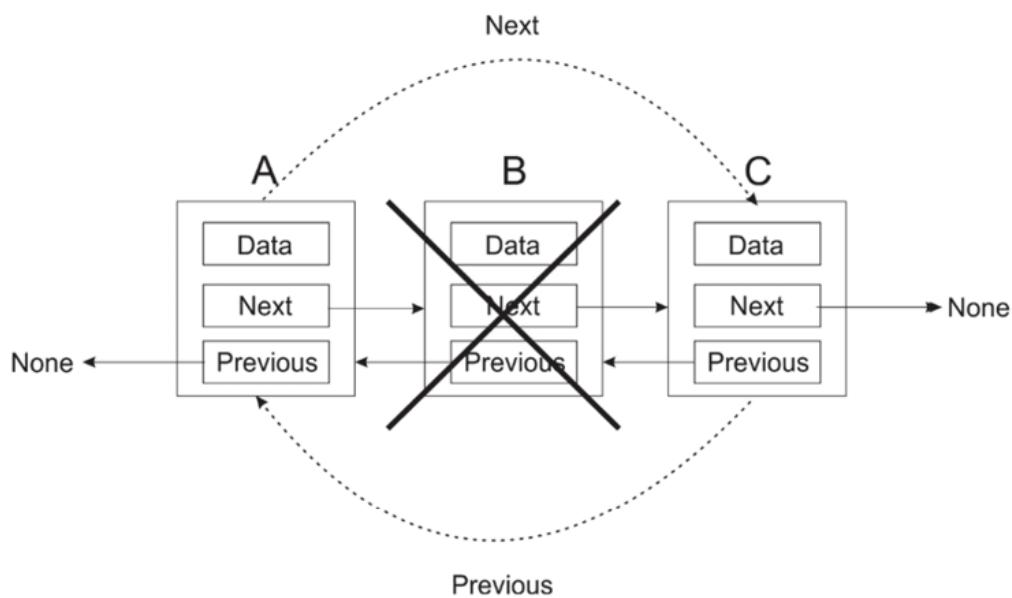


Figure 4.28: Illustration of the deletion of the intermediate node B from the doubly linked list

The complete implementation to delete a node from the doubly linked list in Python is as follows. We'll discuss each part of this code step by step:

```
def delete(self, data):
    # Delete a node from the List.
    current = self.head
    node_deleted = False
    if current is None:
        #List is empty
        print("List is empty")
    elif current.data == data:
        #Item to be deleted is found at starting of the List
        self.head.prev = None
        node_deleted = True
        self.head = current.next
    elif self.tail.data == data:
        #Item to be deleted is found at the end of list
        self.tail = self.tail.prev
        self.tail.next = None
        node_deleted = True
    else:
        while current:
            #search item to be deleted, and delete that node
            if current.data == data:
                current.prev.next = current.next
                current.next.prev = current.prev
                node_deleted = True
                current = current.next
            if node_deleted == False:
                # Item to be deleted is not found in the list
                print("Item not found")
        if node_deleted:
            self.count -= 1
```

Initially, we create a `node_deleted` variable to denote the deleted node in the list and this is initialized to `False`. The `node_deleted` variable is set to `True` if a matching node is found and subsequently removed.

In the `delete` method, the `current` variable is initially set to the `head` node of the list (that is, it points to the `self.head` node of the list). This is shown in the following code fragment:

```
def delete(self, data):
    current = self.head
    node_deleted = False
```

Next, we use a set of `if...else` statements to search various parts of the list to ascertain the node with the specified data that is to be deleted.

First of all, we search for the data to be deleted at the `head` node, and if the data is matched at the `head` node, this node would be deleted. Since `current` is pointing at `head`, if `current` is `None`, this means that the list is empty and has no nodes to find the node to be deleted. The following is its code fragment:

```
if current is None:
    node_deleted = False
```

However, if `current` (which now points to `head`) contains the data being searched for, this means that we found the data to be deleted at the `head` node, and `self.head` is then marked to point to the `current.next` node. Since there is now no node behind `head`, `self.head.prev` is set to `None`. Consider the following code snippet for this:

```
elif current.data == data:
    self.head.prev = None
```

```
    node_deleted = True
    self.head = current.next
```

Similarly, if the node that is to be deleted is found at the `tail` end of the list, we delete the last node by setting its previous node pointing to `None`. `self.tail` is set to point to `self.tail.prev`, and `self.tail.next` is set to `None` as there is no node afterward. Consider the following code fragment for this:

```
elif self.tail.data == data:
    self.tail = self.tail.prev
    self.tail.next = None
    node_deleted = True
```

Lastly, we search for the node to be deleted by looping through the entire list of nodes. If the data that is to be deleted is matched with a node, that node will be deleted.

To delete a node, we make the previous node of the `current` node point to the next node using the `current.prev.next = current.next` code. After that step, we make the current's next node point to the previous node of the `current` node using `current.next.prev = current.prev`. Furthermore, if we traverse the complete list, and the desired item is not found, we print the appropriate message. Consider the following code snippet for this:

```
else:
    while current:
        if current.data == data:
            current.prev.next = current.next
            current.next.prev = current.prev
            node_deleted = True
        current = current.next
```

```
    if node_deleted == False:  
        # Item to be deleted is not found in the List  
        print("Item not found")
```

Finally, the `node_delete` variable is then checked to ascertain whether a node is actually deleted. If any node is deleted, then we decrease the count variable by `1`, and this keeps track of the total number of nodes in the list. See the following code fragment:

```
if node_deleted:  
    self.count -= 1
```

This decrements the count variable by `1` in case any node is deleted.

Let's take an example to see how the delete operation works with the same example of adding three strings – “egg”, “ham”, and “spam”, and then a node with the value “ham” is deleted from the list. The code is as follows:

```
#Code to create for a doubly Linked List  
words = DoublyLinkedList()  
words.append('egg')  
words.append('ham')  
words.append('spam')  
words.delete('ham')  
current = words.head  
while current:  
    print(current.data)  
    current = current.next
```

The output of the preceding code is as follows:

```
egg  
spam
```

The worst-case running time complexity of the delete operation is $O(n)$ since we may have to traverse the list of n items to search for the item to be deleted.

In the next section, we will learn different operations on a circular linked list.

Circular lists

A circular linked list is a special case of a linked list. In a circular linked list, the endpoints are connected, which means that the last node in the list points back to the first node. In other words, we can say that in circular linked lists, all the nodes point to the next node (and the previous node in the case of a doubly linked list) and there is no end node, meaning no node will point to `None`.

The circular linked lists can be based on both singly and doubly linked lists. Consider *Figure 4.29* for the circular linked list based on a singly linked list where the last node, **C**, is again connected to the first node **A**, thus making a circular list.

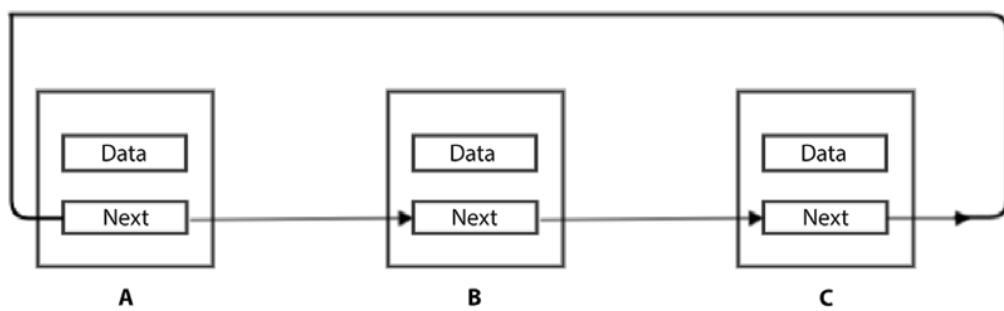


Figure 4.29: Example of a circular list based on a singly linked list

In the case of a doubly linked circular list, the first node points to the last node, and the last node points back to the first node. *Figure 4.30* shows the concept of the circular linked list based on a doubly linked list where the last node **C** is again connected to the first node **A** through the `next` pointer. Node **A** is also connected to node **C** through the `previous` pointer, thus making a circular list.

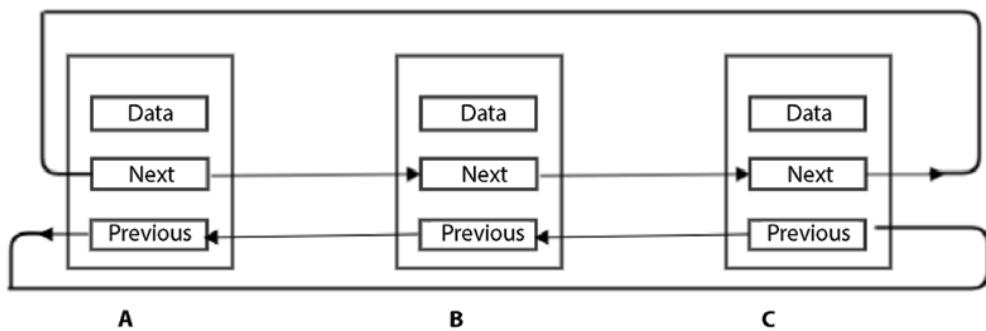


Figure 4.30: Example of a circular list based on a doubly linked list

Now, we are going to look at an implementation of a singly linked circular list. It is very straightforward to implement a doubly linked circular list once we understand the basic concepts of singly and doubly linked lists.

Almost everything is similar except that we should be careful in managing the link of the last node to the first node.

We can reuse the node class that we created in the singly linked lists subsection. We can reuse most parts of the `SinglyLinkedList` class as well. So, we are going to focus on where the circular list implementation differs from the normal singly linked list.

Creating and traversing

The circular linked list class can be created using the following code:

```
class CircularList:  
    def __init__(self):  
        self.tail = None  
        self.head = None  
        self.size = 0
```

In the above code, initially in the circular linked list class, we have two pointers; `self.tail` is used to point to the last node, and `self.head` is used to point to the first node of the list.

Appending items

Here, we want to add a node at the end of a circular linked list, as shown in *Figure 4.31*, in which we have four nodes, wherein the head is pointing to the starting node and the tail is pointing to the last node.

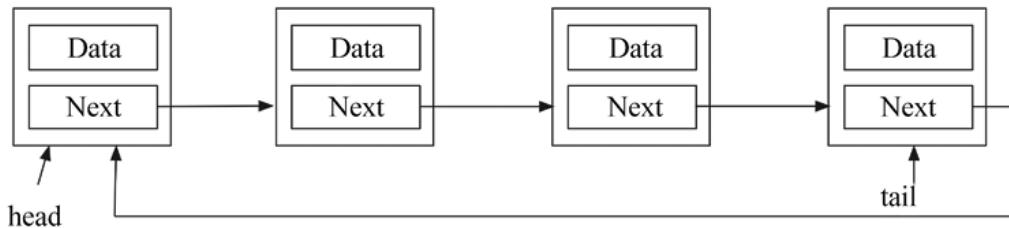


Figure 4.31: Example of a circular linked list for adding a node at the end

Figure 4.32 shows how a node is added to a circular linked list.

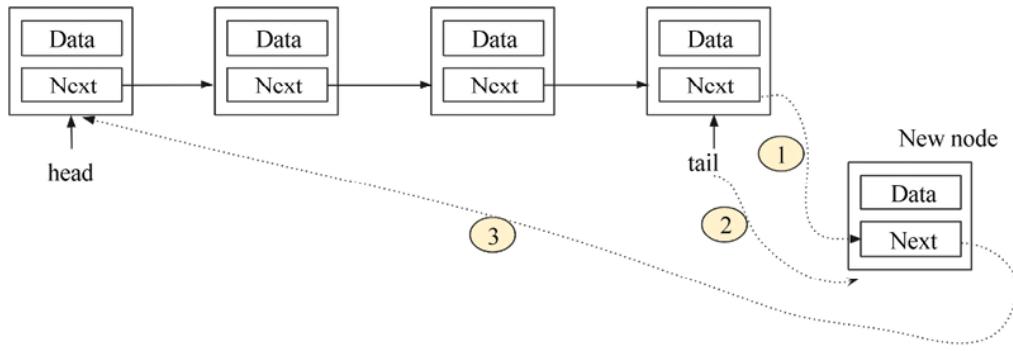


Figure 4.32: Inserting a node at the end of the singly circular list

To add a node at the end, we will update three links:

1. The `next` pointer of the last node to point to a new node
2. The `next` pointer of a new node to point to the `head` node
3. Update the `tail` pointer to point to the new node

The implementation of the circular linked list to append an element at the end of the circular list based on a singly linked list is as follows:

```
def append(self, data):
    node = Node(data)
    if self.tail:
        self.tail.next = node
        self.tail = node
        node.next = self.head
    else:
        self.head = node
        self.tail = node
        self.tail.next = self.tail
    self.size += 1
```

In the above code, firstly, we check whether the list is empty. If the list is empty, we go to the `else` part of the above code. In this case, the new node will be the first node of the list, and both the `head` and

`tail` pointers will point to the new node, while the `next` pointer of the new node will again point to the new node.

Otherwise, if the list is not empty, we go to the `if` part of the preceding code. In this case, we update the three pointers as shown in *Figure 4.32*. This is similar to what we did in the case of the single linked list. Only one link is additionally added in this case, which is shown in bold font in the preceding code.

Further, we can use `iter()` method traverse all the elements of the list, The `iter()` method described below should be defined in `CircularList` class:

```
def iter(self):
    current = self.head
    while current:
        val = current.data
        current = current.next
        yield val
```

The below code can be used to create a singly circular linked list, and then print all the data elements of the list, and then we stop when the counter becomes 3 which is the length of the list.

```
words = CircularList()
words.append('eggs')
words.append('ham')
words.append('spam')
```

```
counter = 0
for word in words.iter():
    print(word)
    counter += 1
```

```
if counter > 2:  
    break
```

The output of the preceding code is as follows:

```
eggs  
ham  
spam
```

Appending any element at an intermediate position in a circular list is exactly to its implementation in a singly linked list.

Querying a list

Traversing a circular linked list is very convenient as we don't need to look for the starting point. We can start anywhere, and we just need to carefully stop traversing when we reach the same node again. We can use the same `iter()` method, which we discussed at the start of this chapter. This will also be the case for the circular list; the only difference is that we have to mention an exit condition when we are iterating through the circular list, otherwise the program will get stuck in a loop, and it will run indefinitely. We can make any exit condition dependent upon our requirements; for example, we can use a counter variable. Consider the following example code:

```
words = CircularList()  
words.append('eggs')  
words.append('ham')  
words.append('spam')  
counter = 0  
for word in words.iter():
```

```
print(word)
counter += 1
if counter > 100:
    break
```

In the above code, we add three strings of data to the circular linked list, and then we print the data values iterating through the list 100 times.

In the next section, let us understand how the `delete` operation works in a circular linked list.

Deleting an element in a circular list

To delete a node in a circular list, it looks like we can do it similarly to how we did in the case of the append operation—simply make sure that the last node through the `tail` pointer points back to the starting node of the list through the `head` pointer. We have the following three scenarios:

1. When the item to be deleted is the `head` node:

In this scenario, we have to ensure that we make the second node of the list the new `head` node (shown as *step 1* in *Figure 4.33*), and the last node should be pointing back to the new head (shown as *step 2* in *Figure 4.33*).

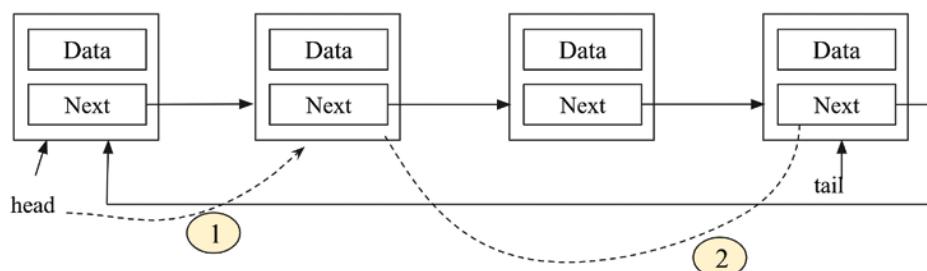


Figure 4.33: Deletion of a starting node in a singly circular list

2. When the item to be deleted is the `last` node:

In this scenario, we have to ensure that we make the second last node the new tail node (shown as *step 1* in *Figure 4.34*), while the new tail node should be pointing back to the new head (shown as *step 2* in *Figure 4.34*).

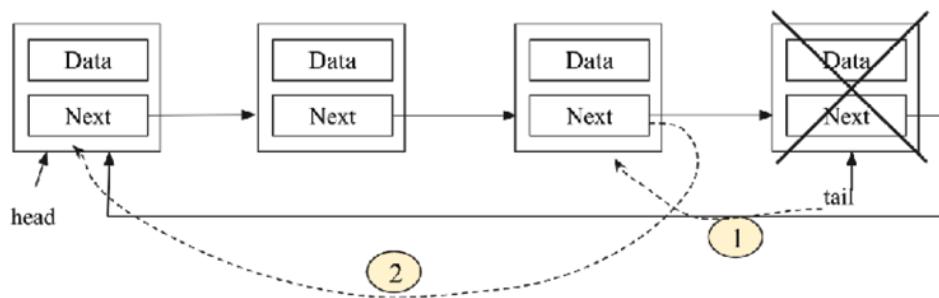


Figure 4.34: Deletion of the last node in a singly circular list

3. When the item to be deleted is an intermediate node:

This is very similar to what we did in the singly linked list. We have to make a link from the previous node of the target node to the next node of the target node, as shown in *Figure 4.35*.

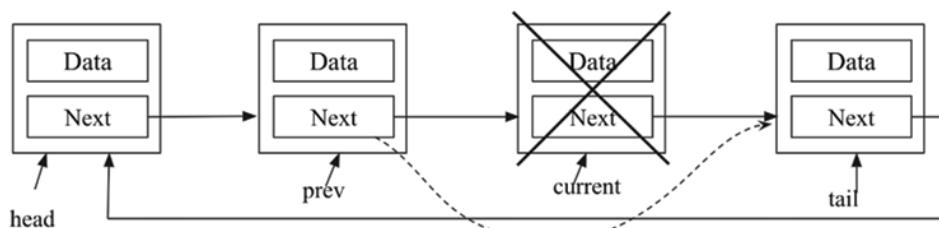


Figure 4.35: Deletion of any intermediate node in a singly circular list

The implementation of the `delete` operation is as follows:

```
def delete(self, data):  
    current = self.head
```

```

prev = self.head
while prev == current or prev != self.tail:
    if current.data == data:
        if current == self.head:
            #item to be deleted is head node
            self.head = current.next
            self.tail.next = self.head
        elif current == self.tail:
            #item to be deleted is tail node
            self.tail = prev
            prev.next = self.head
        else:
            #item to be deleted is an intermediate node
            prev.next = current.next
        self.size -= 1
    return
prev = current
current = current.next
if flag is False:
    print("Item not present in the list")

```

In the preceding code, firstly, iterate over all the elements to search the desired element to be deleted. Here, it is important to note the stopping condition. If we simply check the `current` pointer to be equal to `None` (which we did in the singly linked list), the program will go into an indefinite loop since the current node will never point to `None` in the case of circular linked lists.

For this, we cannot check whether `current` has reached `tail` because then it will never check the last node. So, the stopping criterion in the circular list is the fact that the `prev` and `current` pointers point to the same node. It will work fine except on one occasion when the first loop iteration, at that time, `current` and `prev`, will point to the same node, in other words, the `head` node.

Once, we enter the loop, we check the data value of the current pointer with the given data value to get the node to be deleted. We check whether the node to be deleted is the `head` node, tail node, or intermediate node, and then update the appropriate links shown in *Figures 4.33, 4.34, and 4.35*.

So, we have discussed the different scenarios while deleting any node in singly circular linked list, similarly, the doubly linked list based circular linked list can be implemented.

The following code can be used to create a circular linked list, and apply different delete operations:

```
words = CircularList()
words.append('eggs')
words.append('ham')
words.append('spam')
words.append('foo')
words.append('bar')

print("Let us try to delete something that isn't in the list.")
words.delete('socks')
counter = 0
for item in words.iter():
    print(item)
    counter += 1
    if counter > 4:
        break

print("Let us delete something that is there.")
words.delete('foo')
counter = 0
for item in words.iter():
    print(item)
    counter += 1
    if counter > 3:
        break
```

The output of the above code is as follows:

```
Let us try to delete something that isn't in the list.  
Item not present in the list  
eggs  
ham  
spam  
foo  
bar  
Let us delete something that is there.  
eggs  
ham  
spam  
bar
```

The worst-case time complexity of inserting an element at a given location in the circular linked list is $O(n)$ since we have to traverse the list to the desired location. The complexity of insertion at the first and last locations of the circular list will be $O(1)$. Similarly, the worst-case time complexity to delete an element at a given location is $O(n)$.

So far, we have discussed the different scenarios while deleting any node in a singly circular linked list. Similarly, the doubly linked list can be implemented based on a circular linked list.

In a singly linked list, the traversal of nodes can be done in one direction, whereas, in a doubly linked list, it is possible to traverse in both directions (forward and backward). In both cases, the complexity of the insertion and deletion operations at a given location is $O(n)$ whenever we have to traverse the list in order to reach the desired location where we want to insert or delete any element. Similarly, the worst-case time complexity of the insertion or deletion of a node for a given desired location is $O(n)$. Whenever we

need to save memory space, we should use a singly linked list since it only needs one pointer, whereas a doubly linked list takes more memory space to store double pointers. When a search operation is important, we should use a doubly linked list since it is possible to search in both directions. Furthermore, the circular linked list should be used when we have an application when we need to iterate over the nodes in the list. Let us now see more real-world applications of linked lists.

Practical applications of linked lists

As of now, we have discussed singly linked lists, circular linked lists, and doubly linked lists. Depending upon what kind of operations (insertion, deletion, updating, and so on) will be required in different applications, these data structures are used accordingly. Let's see a few real-time applications where these data structures are being used.

Singly linked lists can be used to represent any sparse matrix. Another important application is to represent and manipulate polynomials by accumulating constants in the node of linked lists.

It can also be used in implementing a dynamic memory management scheme that allows the user to allocate and deallocate the memory as per requirements during the execution of programs.

On the other hand, doubly linked lists are used by the thread scheduler in the operating system to maintain the list of processes running at that time. These lists are also used in the implementation

of **MRU** (most recently used) and **LRU** (least recently used) cache in the operating system.

Doubly linked lists can also be used by various applications to implement **Undo** and **Redo** functionality. The browsers can use these lists to implement backward and forward navigation of the web pages visited.

A circular linked list can be used by operating systems to implement a round-robin scheduling mechanism. Another application of circular linked lists is to implement **Undo** functionality in Photoshop or Word software and use it in implementing a browser cache that allows you to hit the **BACK** button. Besides that, it is also used to implement advanced data structures such as the Fibonacci heap. Multiplayer games also use a circular linked list to swap between players in a loop.

Summary

In this chapter, we studied the concepts that underlie lists, such as nodes and pointers to other nodes. We have discussed singly linked lists, doubly linked lists, and circular linked lists. We have seen various operations that can be applied to these data structures and their implementations using Python.

These types of data structures have certain advantages over arrays. In the case of arrays, insertion and deletion are quite time-consuming as these operations require the shifting of elements downward and upward, respectively, due to contiguous memory allocations. On the other hand, in the case of linked lists, these

operations require only changes in pointers. Another advantage of linked lists over arrays is the allowance of a dynamic memory management scheme that allocates memory during the runtime as and when needed, while the array is based on a static memory allocation scheme.

The singly linked list can traverse in a forward direction only, while traversal in doubly linked lists is bidirectional, hence the reason why the deletion of a node in a doubly linked list is easy compared to a singly linked list. Similarly, circular linked lists save time while accessing the first node from the last node as compared to the singly linked list. Thus, each list has its advantages and disadvantages. We should use them as per the requirements of the application.

In the next chapter, we are going to look at two other data structures that are usually implemented using lists—stacks and queues.

Exercise

1. What will be the time complexity when inserting a data element after an element that is being pointed to by a pointer in a linked list?
2. What will be the time complexity when ascertaining the length of the given linked list?
3. What will be the worst-case time complexity for searching a given element in a singly linked list of length n ?
4. For a given linked list, assuming it has only one `head` pointer that points to the starting point of the list, what will be the time complexity for the following operations?

- a. Insertion at the front of the linked list
 - b. Insertion at the end of the linked list
 - c. Deletion of the front node of the linked list
 - d. Deletion of the last node of the linked list
5. Find the n^{th} node from the end of a linked list.
6. How can you establish whether there is a loop (or circle) in a given linked list?
7. How can you ascertain the middle element of the linked list?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



5

Stacks and Queues

In this chapter, we will discuss two very important data structures: stacks and queues. Stacks and queues have many important applications, such as form operating system architecture, arithmetic expression evaluation, load balancing, managing printing jobs, and traversing data. In stack and queue data structures, the data is stored sequentially, like arrays and linked lists, but unlike arrays and linked lists, the data is handled in a specific order with certain constraints, which we will be discussing in detail in this chapter. Moreover, we will also examine how we can implement stacks and queues using linked lists and arrays.

In this chapter, we will discuss constraints and methods to handle the data in stacks and queues. We will also implement these data structures and learn how to apply different operations to these data structures in Python.

In this chapter, we will cover the following:

- How to implement stacks and queues using various methods
- Some real-life example applications of stacks and queues

Stacks

A stack is a data structure that stores data, similar to a stack of plates in a kitchen. You can put a plate on the top of the stack, and when you need a plate, you take it from the top of the stack.

The last plate that was added to the stack will be the first to be picked up from the stack:

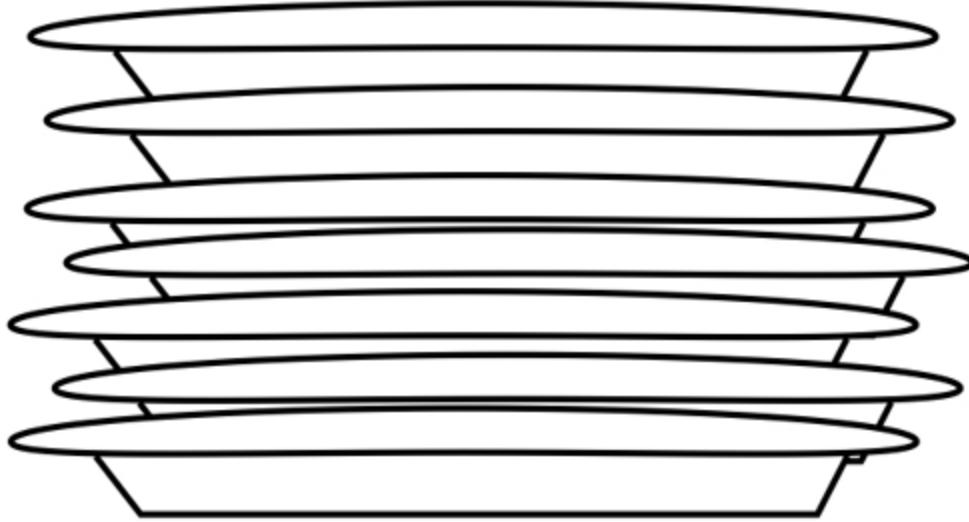


Figure 5.1: Example of a stack

The preceding diagram depicts a stack of plates. Adding a plate to the pile is only possible by leaving that plate on top of the pile. To remove a plate from the pile of plates means to remove the plate that is on top of the pile.

A stack is a data structure that stores the data in a specific order similar to arrays and linked lists, with several constraints:

- Data elements in a stack can only be inserted at the end (`push` operation)
- Data elements in a stack can only be deleted from the end (`pop` operation)

- Only the last data element can be read from the stack (`peek` operation)

A stack data structure allows us to store and read data from one end, and the element which is added last is picked up first. Thus, a stack is a **last in first out (LIFO)** structure, or **last in last out (LILO)**.

There are two primary operations performed on stacks – `push` and `pop`. When an element is added to the top of the stack, it is called a `push` operation, and when an element is to be picked up (that is, removed) from the top of the stack, it is called a `pop` operation.

Another operation is `peek`, in which the top element of the stack can be viewed without removing it from the stack. All the operations in the stack are performed through a pointer, which is generally named `top`. All these operations are shown in *Figure 5.2*:

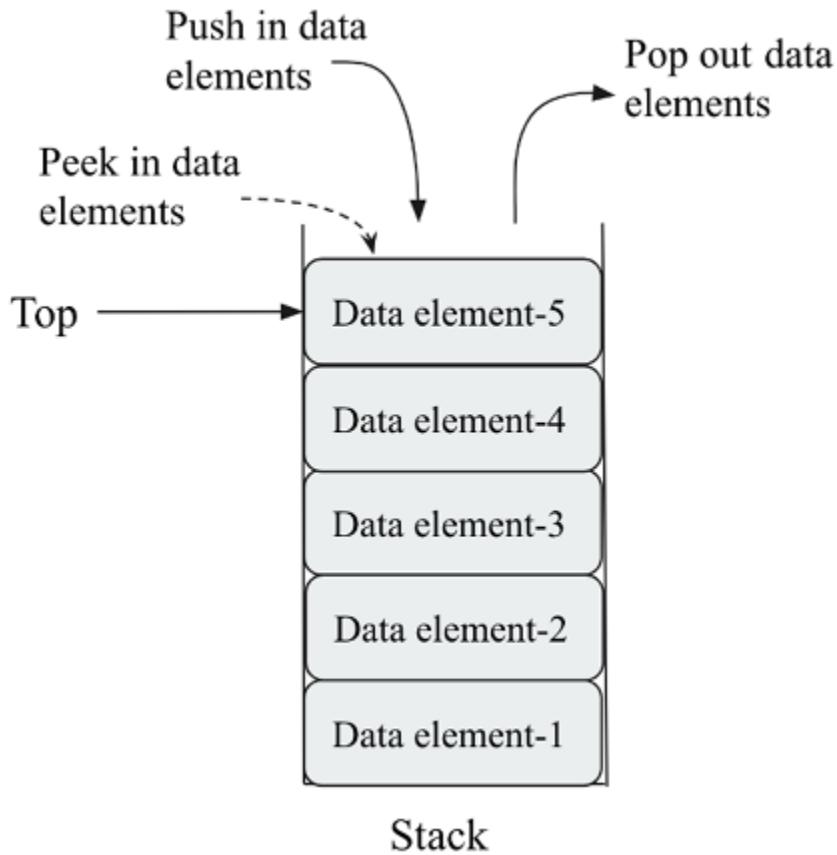


Figure 5.2: Demonstration of push and pop operations in a stack

The following table demonstrates the use of two important stack operations (`push` and `pop`) in the stack:

Stack operation	Size	Contents	Operation results
<code>stack()</code>	0	<code>[]</code>	Stack object created, which is empty.
<code>push "egg"</code>	1	<code>['egg']</code>	One item <code>egg</code> is added to the stack.

<code>push "ham"</code>	2	<code>['egg', 'ham']</code>	One more item, <code>ham</code> , is added to the stack.
<code>peek()</code>	2	<code>['egg', 'ham']</code>	The top element, <code>ham</code> , is returned.
<code>pop()</code>	1	<code>['egg']</code>	The <code>ham</code> item is popped off and returned. (This item was added last, so it is removed first.)
<code>pop()</code>	0	<code>[]</code>	The <code>egg</code> item is popped off and returned. (This is the first item added, so it is returned last.)

Table 5.1: Illustration of different operations in a stack with examples

Stacks are used for a number of things. One common usage for stacks is to keep track of the return address during function calls. Let's imagine that we have the following program:

```
def b():
    print('b')
def a():
    b()
a()
print("done")
```

When the program execution gets to the call to `a()`, a sequence of events will be followed in order to complete the execution of this program. A visualization of all these steps is shown in *Figure 5.3*:

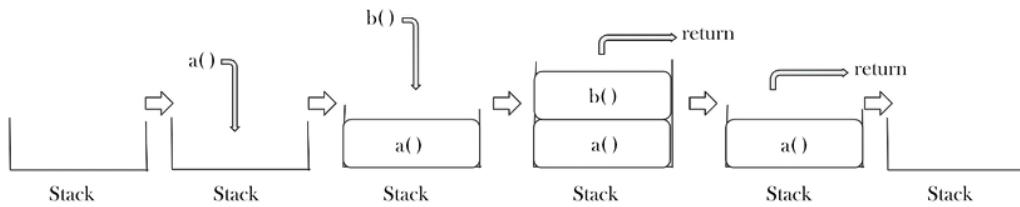


Figure 5.3: Steps for a sequence of events during function calls in our sample program

The sequence of events is as follows:

1. First, the address of the current instruction is pushed onto the stack, and then execution jumps to the definition of `a`
2. Inside function `a()`, function `b()` is called
3. The return address of function `b()` is pushed onto the stack.
Once the execution of the instructions and functions in `b()` are complete, the return address is popped off the stack, which takes us back to function `a()`
4. When all the instructions in function `a()` are completed, the return address is again popped off the stack, which takes us back to the main program and the `print` statement

The output of the above program is as follows:

```
b
done
```

We have now discussed the concept of the stack data structure. Now, let us understand its implementation in Python using array and linked list data structures.

Stack implementation using arrays

Stacks store data in sequential order like arrays and linked lists, with a specific constraint that the data can only be stored and read from one end of the stack following the **last in first out (LIFO)** principle. In general, stacks can be implemented using arrays and linked lists. Array-based implementations will have fixed lengths for the stack, whereas linked list-based implementations can have stacks of variable lengths.

In the case of the array-based implementation of a stack (where the stack has a fixed size), it is important to check whether the stack is full or not, since trying to push an element into a full stack will generate an error, called an overflow. Likewise, trying to apply a `pop` operation to an empty stack causes an error known as an underflow.

Let us understand the implementation of a stack using an array with an example in which we wish to push three data elements, “egg”, “ham”, and “spam”, into the stack. Firstly, to insert new elements into a stack using the `push` operation, we check the overflow condition, which is when the `top` pointer is pointing to the end index of the array. The `top` pointer is the index position of the top element in the stack. If the top element is equal to the overflow condition, the new element cannot be added. This is a stack overflow condition. If there is free space in the array to insert new elements, new data is pushed into the stack. An overview of the `push` operation on a stack using an array is shown in *Figure 5.4*:

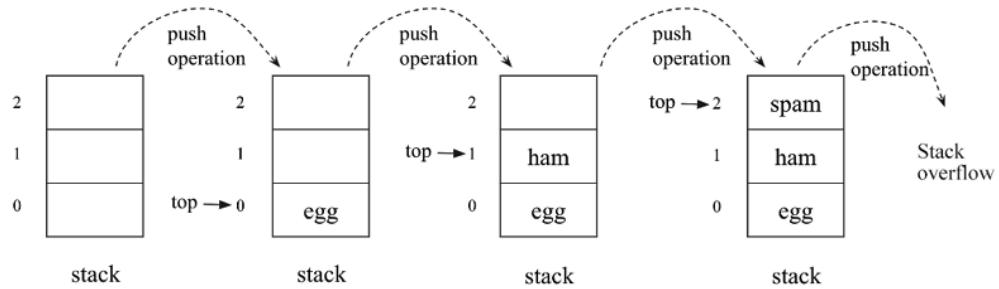


Figure 5.4: Sequence of push operations in a stack implementation using an array

The Python code for the `push` operation is as follows:

```
size = 3
data = [0] * (size)    #Initialize the stack
top = -1
def push(x):
    global top
    if top >= size - 1:
        print("Stack Overflow")
    else:
        top = top + 1
        data[top] = x
```

In the above code, we initialize the stack with a fixed size (say, 3 in this example), and also the `top` pointer to `-1`, which indicates that the stack is empty. Further, in the `push` method, the `top` pointer is compared with the size of the stack to check the overflow condition and, if the stack is full, the stack overflow message is printed. If the stack is not full, the `top` pointer is incremented by 1, and the new data element is added to the top of the stack. The following code is used to insert data elements into the stack:

```
push('egg')
push('ham')
push('spam')
```

```

print(data[0 : top + 1] )
push('new')
push('new2')

```

In the above code, when we try to insert the first three elements, they are added since there was enough space, but when we try to add the data elements `new` and `new2`, the stack is already full, hence these two elements cannot be added to the stack. The output of this code is as follows:

```

['egg', 'ham', 'spam']
Stack Overflow
Stack Overflow

```

Next, the `pop` operation returns the value of the top element of the stack and removes it from the stack. Firstly, we check if the stack is empty or not. If the stack is already empty, a stack underflow message is printed. Otherwise, the top is removed from the stack. An overview of the `pop` operation is shown in *Figure 5.5*:

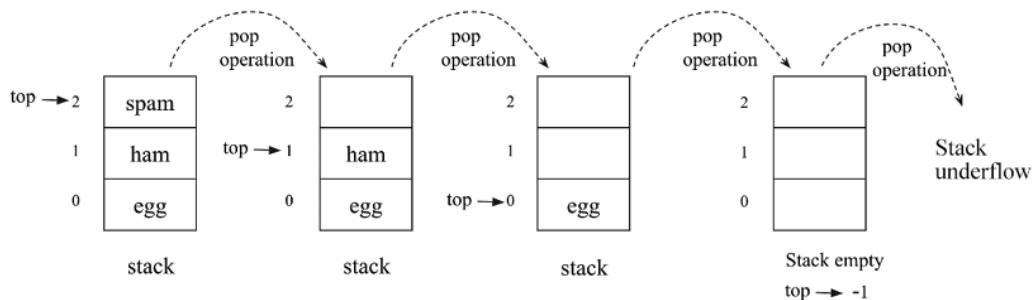


Figure 5.5: Sequence of the pop operation in a stack implementation using an array

The Python code for the `pop` operation is as follows:

```
def pop():
    global top
    if top == -1:
        print("Stack Underflow")
    else:
        top = top - 1
        data[top] = 0
    return data[top+1]
```

In the above code, we first check the underflow condition by checking whether the stack is empty or not. If the `top` pointer has a value of `-1`, it means the stack is empty. Otherwise, the data elements in the stack are removed by decrementing the `top` pointer by 1, and the top data element is returned to the main function.

Let's assume we already added three data elements to the stack, and then we call the `pop` function four times. Since there are only three elements in the stack, the initial three data elements are removed, and when we try to call the `pop` operation a fourth time, the stack underflow message is printed. This is shown in the following code snippet:

```
print(data[0 : top + 1])
pop()
pop()
pop()
pop()
print(data[0 : top + 1])
```

The output of the above code is as follows:

```
['egg', 'ham', 'spam']
Stack Underflow
[]
```

Next, let us see an implementation of the `peek` operation in which we return the value of the top element of the stack. The Python code for this is as follows:

```
def peek():
    global top
    if top == -1:
        print("Stack is empty")
    else:
        print(data[top])
```

In the above code, firstly, we check the position of the `top` pointer in the stack. If the value of the `top` pointer is -1 , it means that the stack is empty, otherwise, we print the value of the top element of the stack.

We have discussed the Python implementation of a stack using an array, so next let us discuss stack implementation using linked lists.

Stack implementation using linked lists

In order to implement the stacks using linked lists, we will write the `stack` class in which all the methods will be declared; however, we will also use the `node` class similar to what we discussed in the previous chapter:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
```

As we know, a `node` in a linked list holds data and a reference to the next item in the linked list. Implementing the stack data structure using a linked list can be treated as a standard linked list with some constraints, including that elements can be added or removed from the end of the list (`push` and `pop` operations) through the `top` pointer. This is shown in *Figure 5.6*:

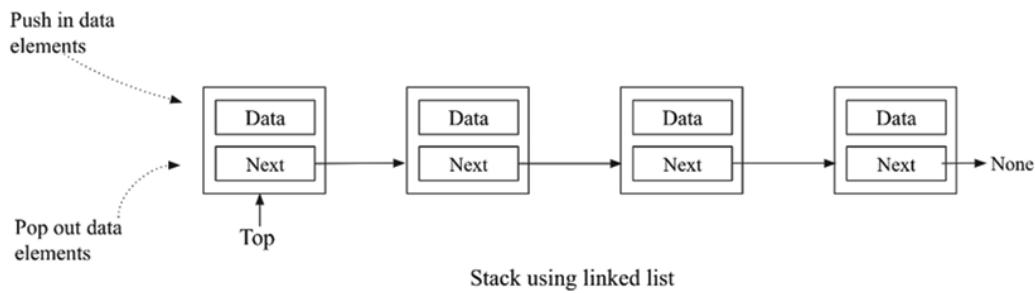


Figure 5.6: Representation of the stack using a linked list

Now let us look at the `stack` class. Its implementation is quite similar to a singly linked list. In addition, we need two things to implement a stack:

1. We first need to know which node is at the top of the stack so that we can apply the `push` and `pop` operations through this node
2. We would also like to keep track of the number of nodes in the stack, so we add a `size` variable to the `stack` class

Consider the following code snippet for the `stack` class:

```

class Stack:
    def __init__(self):
        self.top = None
        self.size = 0

```

In the above code, we have declared the `top` and `size` variables, which are initialized to `None` and `0`. Once we have initialized the `Stack` class, next, we will implement different operations in the `Stack` class. First, let us start with a discussion of the `push` operation.

Push operation

The `push` operation is an important operation on a stack; it is used to add an element at the top of the stack. In order to add a new node to the stack, firstly, we check if the stack already has some items in it or if it is empty. We are not required here to check the overflow condition because we are not required to fix the length of the stack, unlike the stack implementation using arrays.

If the stack already has some elements, then we have to do two things:

1. The new node must have its next pointer pointing to the node that was at the top earlier
2. We put this new node at the top of the stack by pointing `self.top` to the newly added node

See the two instructions in the following *Figure 5.7*:

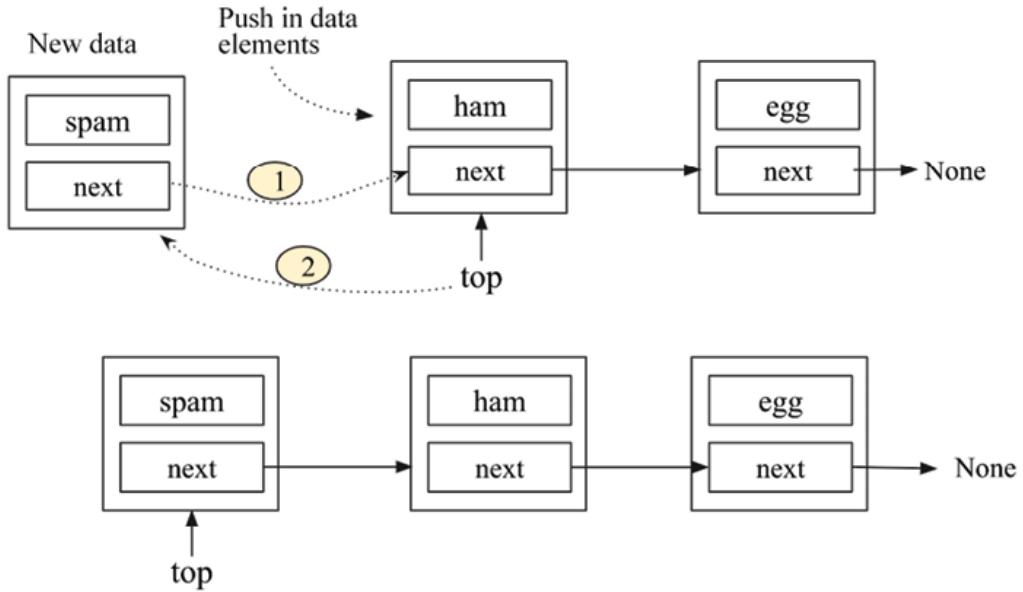


Figure 5.7: Workings of the push operation on the stack

If the existing stack is empty, and the new node to be added is the first element, we need to make this node the top node of the element. Thus, `self.top` will point to this new node. See the following *Figure 5.8*:

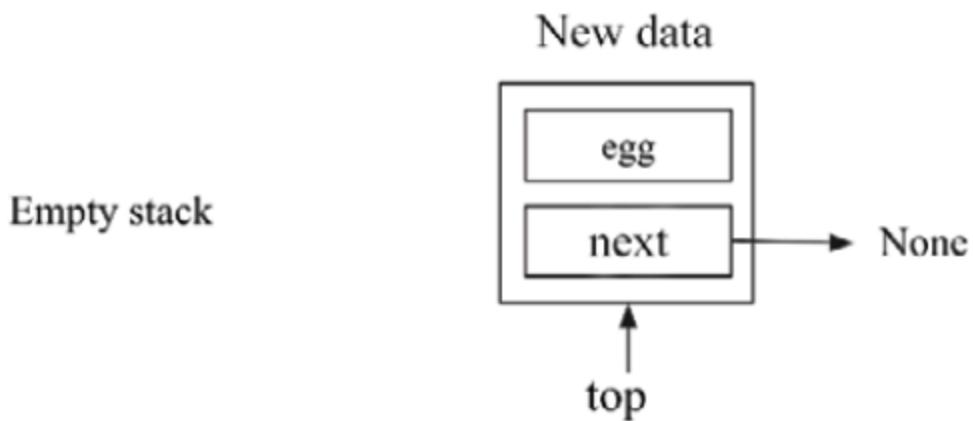


Figure 5.8: Insertion of the data element "egg" into an empty stack

The following is the complete implementation of the `push` operation, which should be defined in the `stack` class:

```
def push(self, data):
    # create a new node
    node = Node(data)
    if self.top:
        node.next = self.top
        self.top = node
    else:
        self.top = node
    self.size += 1
```

In the above code, we create a new node and store the data in that. Then we check the position of the `top` pointer. If it is not null, that means the stack is not empty, and we add the new node, updating two pointers as shown in *Figure 5.7*. In the `else` part, we make the `top` pointer point to the new node. Finally, we increase the size of the stack by incrementing the `self.size` variable.

To create a stack of three data elements, we use the following code:

```
words = Stack()
words.push('egg')
words.push('ham')
words.push('spam')
#print the stack elements.
current = words.top
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
spam
ham
egg
```

In the above code, we created a stack of three elements – egg, ham, and spam. Next, we will discuss the `pop` operation in stack data structures.

Pop operation

Another important operation that is applied to the stack is the `pop` operation. In this operation, the topmost element of the stack is read, and then removed from the stack. The `pop` method returns the topmost element of the stack and returns `None` if the stack is empty.

To implement the `pop` operation on a stack, we do following:

1. First, check if the stack is empty. The `pop` operation is not allowed on an empty stack.
2. If the stack is not empty, check whether the top node has its next attribute pointing to some other node. If so, it means the stack contains elements, and the topmost node is pointing to the next node in the stack. To apply the `pop` operation, we have to change the top pointer. The next node should be at the top. We do this by pointing `self.top` to `self.top.next`. See the following *Figure 5.9* to understand this:

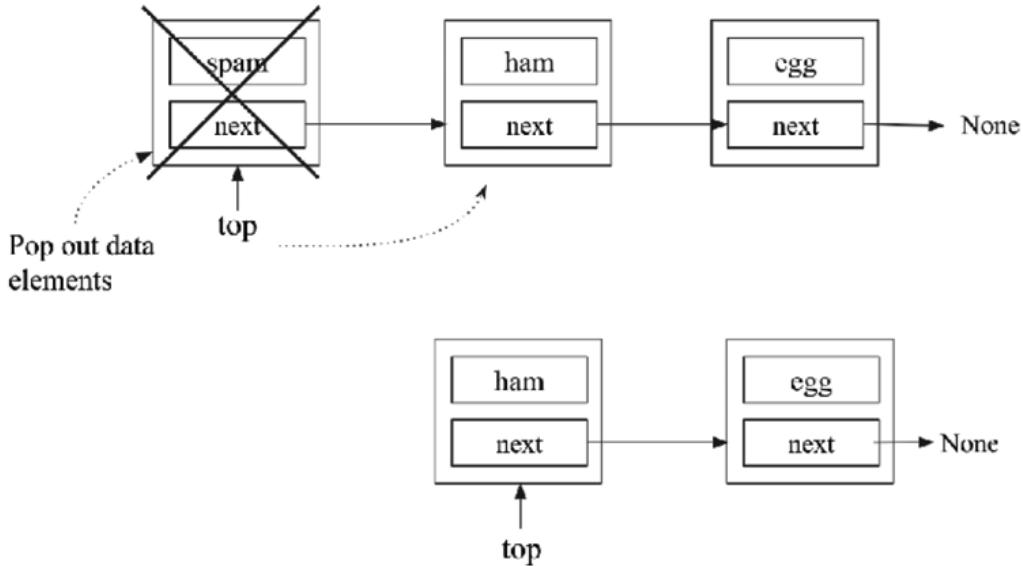


Figure 5.9: Workings of the pop operation on the stack

3. When there is only one node in the stack, the stack will be empty after the `pop` operation. We have to change the `top` pointer to `None`. See the following *Figure 5.10*:

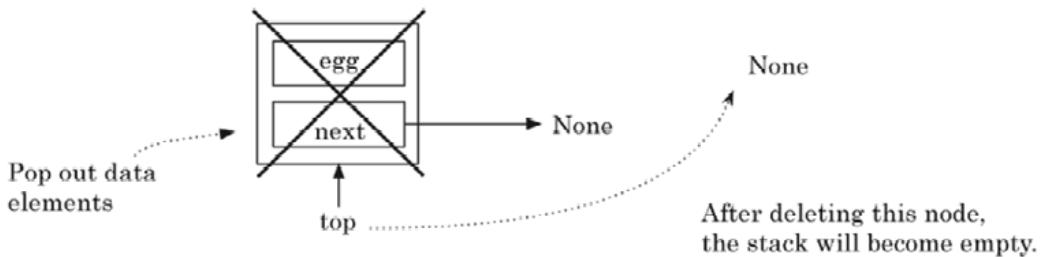


Figure 5.10: The pop operation on a stack with one element

4. Removing this node results in `self.top` pointing to `None`, as shown in *Figure 5.10*.
5. We also decrement the size of the stack by `1` if the stack is not empty.

Here is the code for the `pop` operation for the stack in Python, which should be defined in the `stack` class:

```
def pop(self):
    if self.top:
        data = self.top.data
        self.size -= 1
        if self.top.next: #check if there is more than one node.
            self.top = self.top.next
        else:
            self.top = None
    return data
else:
    print("Stack is empty")
```

In the above code, firstly, we check the position of the `top` pointer. If it is not null, it means the stack is not empty, and we can apply the `pop` operation such that if there is more than one data element in the stack, we move the `top` pointer to point to the next node (see *Figure 5.9*), and if that is the last node, we make the `top` pointer point to `None` (see *Figure 5.10*). We also decrease the size of the stack by decrementing the `self.size` variable.

Let's say we have three data elements in a stack. We can use the following code to apply the `pop` operation to the stack:

```
words.pop()
current = words.top
while current:
    print(current.data)
    current = current.next
```

The output of the above code is as follows:

```
ham
egg
```

In the above code, we popped off the top element from the stack of three elements – `egg`, `ham`, `spam`.

Next, we will discuss the `peek` operation used on stack data structures.

Peek operation

There is another important operation that can be applied to stacks—the `peek` method. This method returns the top element from the stack without deleting it from the stack. The only difference between `peek` and `pop` is that the `peek` method just returns the topmost element; however, in the case of a `pop` method, the topmost element is returned, and that element is also deleted from the stack.

The `peek` operation allows us to look at the top element without changing the stack. This operation is very straightforward. If there is a top element, return its data; otherwise, return `None` (thus, the behavior of `peek` matches that of `pop`). The implementation of the `peek` method is as follows (this should be defined in the `Stack` class):

```
def peek(self):
    if self.top:
        return self.top.data
    else:
        print("Stack is empty")
```

In the above code, we first check the position of the `top` pointer using `self.top`. If it is not null, this means the stack is not empty, and we return the data value of the topmost node, otherwise, we print the

message that the stack is empty. We can use the `peek` method to get the top element of the stack through the following code:

```
words.peek()
```

The output of the above code is:

```
spam
```

As per our original example of the three data elements being added to the stack, if we use the `peek` method, we get the top element, `spam`, as an output.

Stacks are an important data structure with several real-world applications. To better understand the concept of the stack, we will discuss one of these applications: bracket matching utilizing stacks.

Applications of stacks

As we know, array and linked list data structures can do whatever the stack or queue data structures (that we will discuss shortly) can do.

Despite this, these data structures are important because of their many applications. For example, in any application, it may be required to add or delete any element in a particular order. stack and queues can be used for this to avoid any potential bug in the program, perhaps accessing/deleting an element from the middle of the list (which can happen in the cases of arrays and linked lists).

Now let us look at an example bracket-matching application and see how we can use our stack to implement it.

Let us write a function `check_brackets` that will verify whether a given expression containing brackets—`()`, `[]`, or `{ }`—is balanced or not, that is, whether the number of closing brackets matches the number of opening brackets. Stacks can be used for traversing a list of items in reverse order since they follow the **LIFO** rule, which makes them a good choice for this problem.

The following code is for a separate `check_brackets` method defined outside the `Stack` class. This method will use the `Stack` class that we discussed in the previous section. The method takes an expression consisting of alphabetical characters and brackets as input and produces `True` or `False` as output for whether the given expression is valid or not, respectively. The code for the `check_brackets` method is as follows:

```
def check_brackets(expression):
    brackets_stack = Stack()      #The stack class, we defined in prev
    last = ''
    for ch in expression:
        if ch in ('{', '[', '('):
            brackets_stack.push(ch)
        if ch in ('}', ']', ')'):
            last = brackets_stack.pop()
            if last == '{' and ch == '}':
                continue
            elif last == '[' and ch == ']':
                continue
            elif last == '(' and ch == ')':
                continue
            else:
                return False
    if brackets_stack.size > 0:
```

```
        return False
else:
    return True
```

The above function parses each character in the expression passed to it. If it gets an open bracket, it pushes it onto the stack. If it gets a closing bracket, it pops the top element off the stack and compares the two brackets to make sure their types match-`(` should match `)`, `[` should match `]`, and `{` should match `}`. If they don't, we return `False`; otherwise, we continue parsing.

Once we reach the end of the expression, we need to do one last check. If the stack is empty, then it is fine and we can return `True`. But if the stack is not empty, then we have an opening bracket that does not have a matching closing bracket and we will return `False`.

We can test the bracket-matcher with the following code:

```
sl = (
    "({foo}{bar})[hello](((this)is)a)test",
    "({foo}{bar})[hello](((this)is)atest",
    "({foo}{bar})[hello](((this)is)a)test))"
for s in sl:
    m = check_brackets(s)
    print("{}: {}".format(s, m))
```

Only the first of the three statements should match. When we run the code, we get the following output:

```
{(foo)(bar)}[hello](((this)is)a)test: True
{(foo)(bar)}[hello](((this)is)atest: False
{(foo)(bar)}[hello](((this)is)a)test)): False
```

In the above sample three expressions, we can see that the first expression is valid, while the other two are not valid expressions. Hence, the output of the preceding code is `True`, `False`, and `False`.

In summary, the `push`, `pop`, and `peek` operations of the stack data structure have a time complexity of $O(1)$ since the addition and deletion operations can be directly performed in constant time through the `top` pointer. The stack data structure is simple; however, it is used to implement many functionalities in real-world applications. For example, the back and forward buttons in web browsers are implemented using stacks. Stacks are also used to implement the undo and redo functionalities in word processors.

We have discussed the stack data structure and its implementations using arrays and linked lists. In the next section, we will discuss the queue data structure and the different operations that can be applied to queues.

Queues

Another important data structure is the queue, which is used to store data similarly to stacks and linked lists, with some constraints and in a specific order. The queue data structure is very similar to the regular queue you are accustomed to in real life. It is just like a line of people waiting to be served in sequential order at a shop. Queues are a fundamentally important concept to grasp since many other data structures are built on them.

A queue works as follows. The first person to join the queue usually gets served first, and everyone will be served in the order in which

they joined the queue. The acronym **FIFO** best explains the concept of a queue. **FIFO** stands for **first in, first out**. When people are standing in a queue waiting for their turn to be served, service is only rendered at the front of the queue. Therefore, people are dequeued from the front of the queue and enqueued from the back where they wait their turn. The only time people exit the queue is when they have been served, which only occurs at the very front of the queue. Refer to the following diagram, where people are standing in the queue, and the person at the front will be served first:

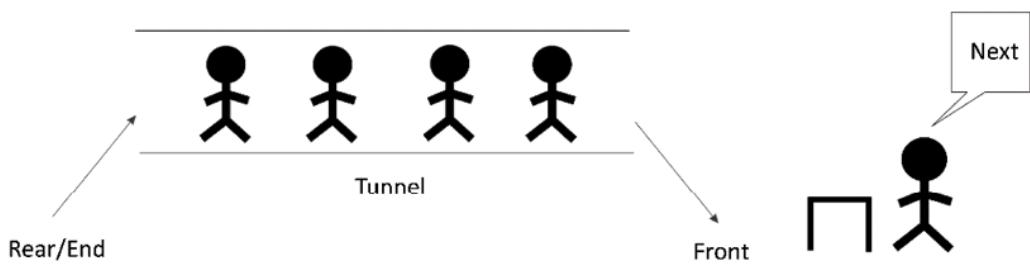


Figure 5.11: Illustration of a queue

To join the queue, participants must stand behind the last person in the queue. This is the only legal or permitted way the queue accepts new entrants. The length of the queue does not matter.

A queue is a list of elements stored in sequence with the following constraints:

1. Data elements can only be inserted from one end, the rear end/tail of the queue.
2. Data elements can only be deleted from the other end, the front/head of the queue.
3. Only data elements from the front of the queue can be read.

The operation to add an element to the queue is known as `enqueue`. Deleting an element from the queue uses the `dequeue` operation. Whenever an element is enqueued, the length or size of the queue increments by 1, and dequeuing an item reduces the number of elements in the queue by 1.

We can see this concept in the doubly linked list shown in *Figure 5.12*, in which we can add new elements to the tail/rear end and elements can only be deleted from the head/front end of the queue:

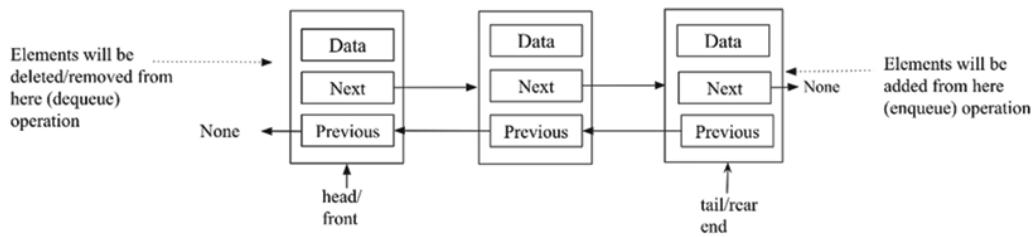


Figure 5.12: Queue implementation using the stack data structure

The reader is advised to not confuse the notation: the `enqueue` operation will be performed only at the **tail/rear** end and the `dequeue` operation will be performed from the **head/front** end. It should be fixed that one end will be used for `enqueue` operations and the other end will be used for `dequeue` operations; however, either end can be used for each of these operations. It is good in general practice to fix that we perform `enqueue` operations from the **rear** end and `dequeue` operations from the **front** end. To demonstrate these two operations, the following table shows the effects of adding and removing elements from a queue:

Queue operation	Size	Contents	Operation results

<code>queue()</code>	0	<code>[]</code>	Queue object created, which is empty.
<code>enqueue-</code> <code>"packt"</code>	1	<code>['packt']</code>	One item, <code>packt</code> , is added to the queue.
<code>enqueue</code> <code>"publishin</code> <code>g"</code>	2	<code>[</code> <code>'packt',</code> <code>'publishi</code> <code>ng']</code>	One more item, <code>publishing</code> , is added to the queue.
<code>Size()</code>	2	<code>[</code> <code>'packt',</code> <code>'publishi</code> <code>ng']</code>	Return the number of items in the queue, which is 2 in this example.
<code>dequeue()</code>	1	<code>['publish</code> <code>ing']</code>	The <code>packt</code> item is dequeued and returned. (This item was added first, so it is removed first.)
<code>dequeue()</code>	0	<code>[]</code>	The <code>publishing</code> item is dequeued and returned. (This is the last item added, so it is returned last.)

Table 5.2: Illustration of different operations on an example queue

Queue data structures in Python have a built-in implementation, `queue.Queue`, and can also be implemented using the `deque` class from

the `collections` module. Queue data structures can be implemented using various methods in Python, namely, (1) Python's built-in list, (2) stacks, and (3) node-based linked lists. We will discuss them one by one in detail.

Python's list-based queues

Firstly, in order to implement a queue based on Python's `list` data structure, we create a `ListQueue` class, in which we declare and define the different functionalities of queue. In this method, we store the actual data in Python's `list` data structure. The `ListQueue` class is defined as follows:

```
class ListQueue:  
    def __init__(self):  
        self.items = []  
        self.front = self.rear = 0  
        self.size = 3      # maximum capacity of the queue
```

In the `__init__` initialization method, the `items` instance variable is set to `[]`, which means the queue is empty when created. The size of the queue is also set to `4` (as an example in this code), which is the maximum capacity for the number of elements that can be stored in the queue. Moreover, the initial position of the rear and front indices are also set to `0`. `enqueue` and `dequeue` are important methods in queues, and we will discuss them next.

The enqueue operation

The `enqueue` operation adds an item at the end of the queue. Consider the example of adding elements to the queue to understand the

concept shown in *Figure 5.13*. We start with an empty list. Initially, we add an item 3 at index 0.

Next, we add an item 11 at index 1, and move the rear pointer every time we add an element:

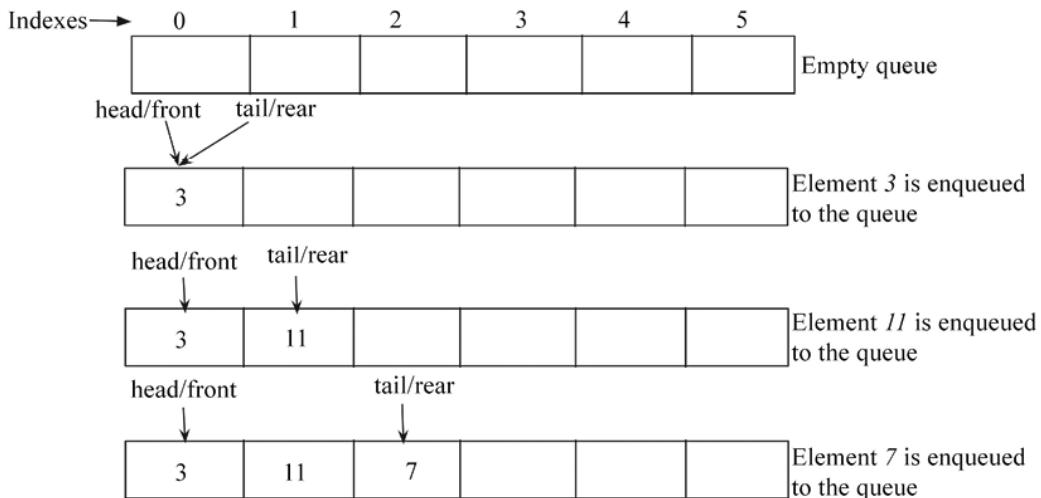


Figure 5.13: Example of an enqueue operation on the queue

In order to implement the enqueue operation, we use the `append` method of the `List` class to append items (or data) to the end of the queue. See the following code for the implementation of the `enqueue` method. This should be defined in the `ListQueue` class:

```
def enqueue(self, data):
    if self.size == self.rear:
        print("\n Queue is full")
    else:
        self.items.append(data)
        self.rear += 1
```

Here, we first check whether the queue is full by comparing the maximum capacity of the queue with the position of the `rear` index.

Further, if there is space in the queue, we use the `append` method of the `List` class to add the data at the end of the queue and increase the rear pointer by `1`.

To create a queue using the `ListQueue` class, we use the following code:

```
q= ListQueue()  
q.enqueue(20)  
q.enqueue(30)  
q.enqueue(40)  
q.enqueue(50)  
print(q.items)
```

The output of the above code is as follows:

```
Queue is full  
[20, 30, 40]
```

In the above code, we can add a maximum of three data elements since we have set the maximum capacity of the queue to be `3`. After adding three elements, when we try to add another new element, we get a message that the queue is full.

The `dequeue` operation

The `dequeue` operation is used to read and delete items from the queue. This method returns the front item from the queue and deletes it. Consider the example of dequeuing elements from the queue shown in *Figure 5.14*. Here, we have a queue containing elements `{3, 11, 7, 1, 4, 2}`. In order to dequeue any element from this queue, the element inserted first will be removed first, so the

item 3 is removed. When we dequeue any element from the queue, we also decrease the rear pointer by 1:

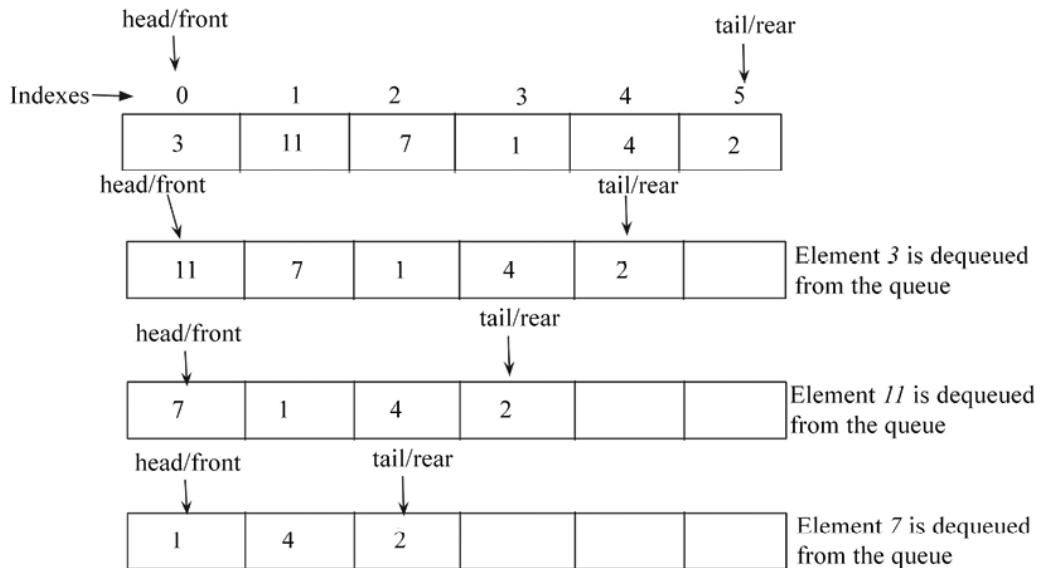


Figure 5.14. Example of a dequeue operation on a queue

The following is the implementation of the `dequeue` method, which should be defined in the `ListQueue` class:

```
def dequeue(self):  
    if self.front == self.rear:  
        print("Queue is empty")  
    else:  
        data = self.items.pop(0)      # delete the item from front  
        self.rear -= 1  
    return data
```

In the above code, we firstly check whether the queue is already empty by comparing the front and rear pointers. If both rear and front pointers are same, it means the queue is empty. If there are some elements in the queue, we use the `pop` method to dequeue an

element. The Python `List` class has a method called `pop()`. The `pop` method does the following:

1. Deletes the last item from the list
2. Returns the deleted item from the list back to the user or code that called it

The item at the first position pointed to by the `front` variable is popped and saved in the `data` variable. We also decrease the `rear` variable by `1`, since one data item has been deleted from the queue. Finally, in the last line of the method, the data is returned.

To dequeue any element from an existing queue (say items `{20, 30, 40}`), we use the following code:

```
data = q.dequeue()
print(data)
print(q.items)
```

The output of the above code is as follows:

```
20
[30, 40]
```

In the above code, when we dequeue an element from the queue, we get the element `20`, which was the first added.

The limitation of this approach to queue implementation is that the length of the queue is fixed, which may be not desirable for an efficient implementation of a queue. Now, let's discuss the linked list-based implementation of queues.

Linked list based queues

A queue data structure can also be implemented using any linked list, such as singly-linked or doubly-linked lists. We already discussed the implementation of singly or doubly linked lists in the previous *Chapter 4, Linked Lists*. We implement queues using linked lists that follow the **FIFO** property of the queue data structure.

Let us discuss the implementation of a queue using a doubly-linked list. For this, we start with the implementation of the `node` class the same as the `node` we defined when we discussed doubly-linked lists in the previous *Chapter 4, Linked Lists*. Moreover, the `Queue` class is very similar to that of the doubly-linked list class. Here, we have `head` and `tail` pointers, where `tail` points to the end of the queue (the rear end) that will be used for adding new elements, and the `head` pointer points to the start of the queue (the front end) that will be used for dequeuing the elements from the queue. The implementation of the `Queue` class is shown in the following code:

```
class Node(object):
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.count = 0
```

Initially, the `self.head` and `self.tail` pointers are set to `None` upon creation of an instance of the `Queue` class. To keep a count of the

number of nodes in `Queue`, the `count` instance variable is also maintained here and initially set to `0`.

The enqueue operation

Elements are added to a `Queue` object via the `enqueue` method. The data elements are added through nodes. The `enqueue` method code is very similar to the `append` operation of the doubly-linked list that we discussed in *Chapter 4, Linked Lists*.

The enqueue operation creates a node from the data passed to it and appends it to the `tail` of the queue.

Firstly, we check if the new node to be enqueued is the first node, and whether the queue is empty or not. If it is empty, the new node becomes the first node of the queue, as shown in *Figure 5.15*:

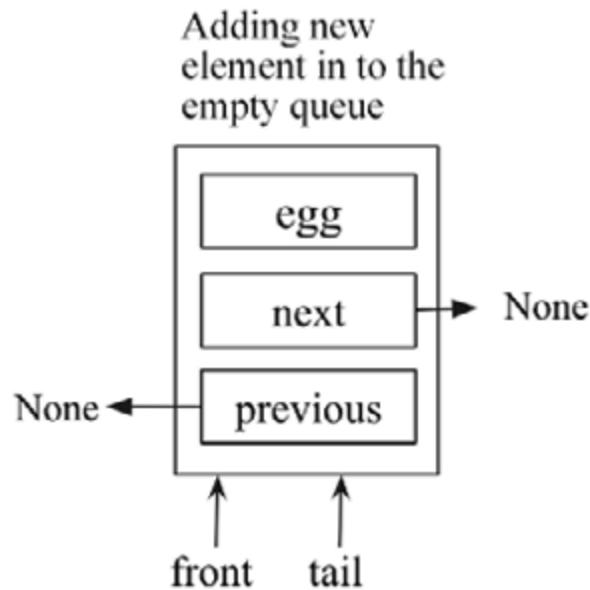


Figure 5.15: Illustration of enqueueing a new node in an empty queue

If the queue is not empty, the new node is appended to the rear end of the queue. In order to do this and enqueue an element to an existing queue, we append the node by updating three links: (1) the previous pointer of the new node should point to the tail of the queue, (2) the next pointer of the tail node should point to the new node, and (3) the tail pointer should be updated to the new node. All these links are shown in *Figure 5.16*:

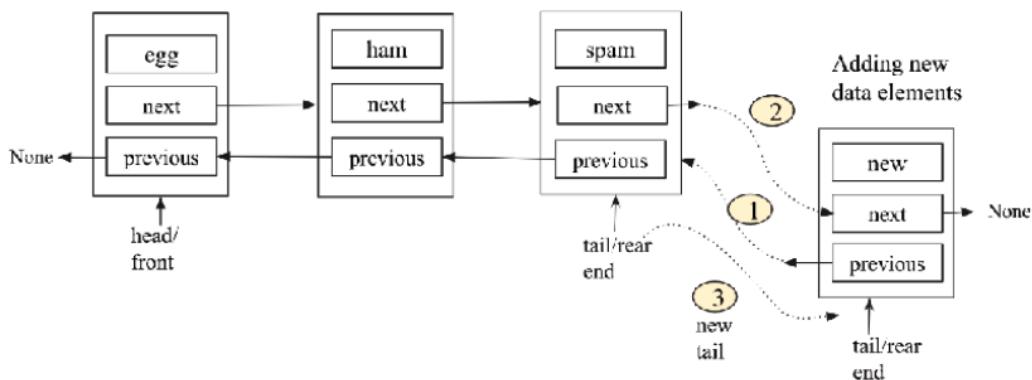


Figure 5.16: Illustration of links to be updated for an enqueue operation in a queue

The `enqueue` operation is implemented in the `Queue` class, as shown in the following code:

```
def enqueue(self, data):
    new_node = Node(data, None, None)
    if self.head == None:
        self.head = new_node
        self.tail = self.head
    else:
        new_node.prev = self.tail
        self.tail.next = new_node
        self.tail = new_node
    self.count += 1
```

In the above code, we first check whether the queue is empty or not. If `head` points to `None`, this means the queue is empty. If it is empty, the new node is made the first node of the queue, and we make both `self.head` and `self.tail` point to the newly created node. If the queue is not empty, we append the new node to the rear of the queue by updating the three links shown in *Figure 5.16*. Finally, the total count of elements in the queue is increased by the line `self.count += 1`.

The worst-case time complexity of an `enqueue` operation on the queue is $O(1)$, since any item can be appended directly through the `tail` pointer in constant time.

The dequeue operation

The other operation that makes a doubly-linked list behave like a queue is the `dequeue` method. This method removes the node at the front of the queue, as shown in *Figure 5.17*. Here, we first check whether the dequeuing element is the last node in the queue, and if so, we will make the queue empty after the `dequeue` operation. If this is not the case, we dequeue the first element by updating the front/head pointer to the next node and the previous pointer of the new head to `None`, as shown in *Figure 5.17*:

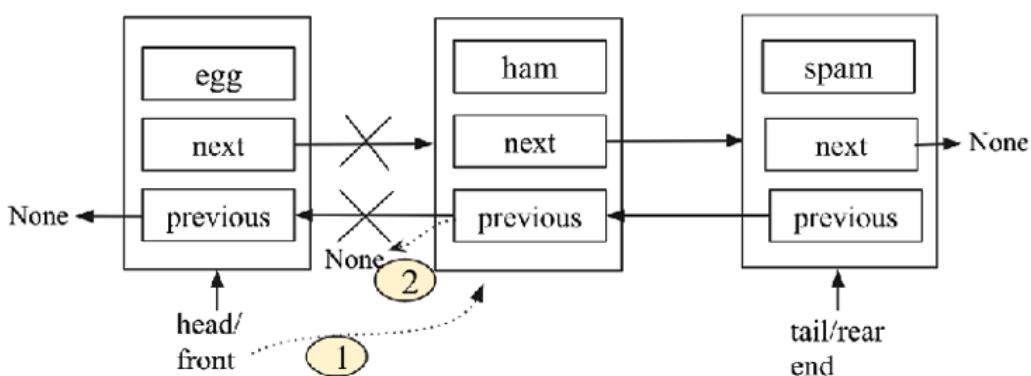


Figure 5.17: Illustration of the dequeue operation on a queue

The implementation of the `dequeue` operation on a queue is very similar to deleting the first element from the given doubly-linked list, as the following code for the `dequeue` operation shows:

```
def dequeue(self):
    if self.count == 1:
        self.count -= 1
        self.head = None
        self.tail = None
    elif self.count > 1:
        self.head = self.head.next
        self.head.prev = None
    elif self.count < 1:
        print("Queue is empty")
    self.count -= 1
```

In order to dequeue any element from the queue, we firstly check the number of items in the queue using the `self.count` variable. If the `self.count` variable is equal to `1`, it means the dequeuing element is the last element, and we update the head and tail pointers to `None`.

If the queue has many nodes, then the head pointer is shifted to point to the next node after `self.head` by updating the two links shown in *Figure 5.17*. We also check whether there is an item left in the queue, and if not, then a message is printed that the queue is empty. Finally, the `self.count` variable is decremented by `1`.

The worst-case time complexity of a dequeue operation in the queue is `O(1)`, since any item can be directly removed via the `head` pointer in constant time.

Stack-based queues

A queue is a linear data structure in which enqueue operations are performed from one end and deletion (dequeue) operations are performed from the other end following the **FIFO** principle. There are two methods to implement queues using stacks:

- When the dequeue operation is costly
- When the enqueue operation is costly

Approach 1: When the dequeue operation is costly

We use two stacks for the implementation of the queue. In this approach, the enqueue operation is straightforward. A new element can be enqueued in the queue using the push operation on the first of the two stacks (in other words, Stack-1) used for the implementation of the queue.

The enqueue operation is depicted in *Figure 5.18*, showing an example of enqueueing elements {23, 13, 11} to the queue:

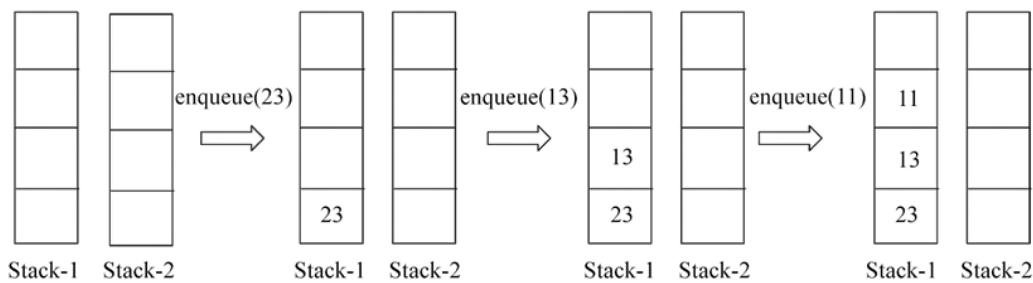


Figure 5.18: Illustration of an enqueue operation in the queue using approach 1

Further more, the dequeue operation can be implemented with two stacks (Stack-1 and Stack-2) using the following steps:

1. Firstly, the elements are removed (popped off) from Stack-1, and then one by one all the elements are added (pushed) to Stack-2.
2. The topmost data element will be popped off Stack-2 and will be returned as the desired element.
3. Finally, the remaining elements are popped off Stack-2 one by one and then pushed again to Stack-1.

Let's look at an example to help understand this concept. Let's say we have three elements stored in the queue `{23, 13, 11}`, and now we want to dequeue an element from this queue. The complete process is shown in *Figure 5.19* following the above three steps. As you might notice, this implementation follows the **FIFO** property of queues and hence 23 is returned, as it was added first:

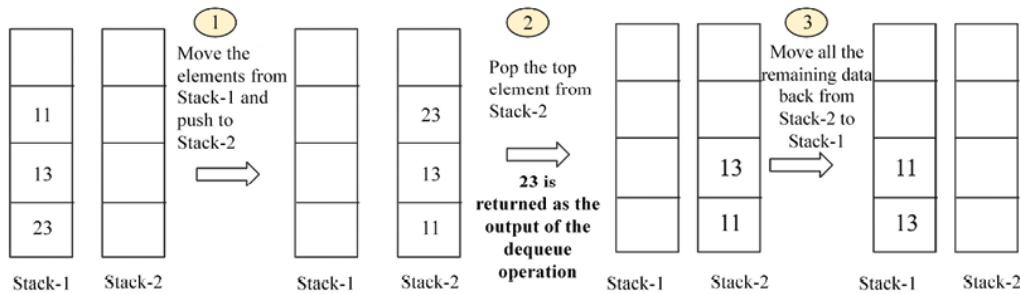


Figure 5.19: Illustration of a dequeue operation in the queue using approach 1

The worst-case time complexity of enqueue operations is $O(1)$, since any element can be added directly to the first stack, and the time complexity of the dequeue operation is $O(n)$, since all elements are accessed and transferred from Stack-1 to Stack-2.

Approach 2: When the enqueue operation is costly

In this method, the enqueue operation is quite similar to the dequeue operation of the previous approach we just discussed, and the dequeue operation is likewise similar to the previous enqueue operation.

In order to implement the enqueue operation, we follow the steps:

1. Move all the elements from Stack-1 to Stack-2.
2. Push the element we want to enqueue to Stack-2.

Move all the elements from Stack-2 to Stack-1 one by one. Pop the elements from Stack-2 and push them to Stack-1.

Let's take an example to understand this concept. Let's say we want to enqueue three elements $\{23, 13, 11\}$ in the queue one by one. We do this by following the above three steps, as shown in *Figure 5.20*, *Figure 5.21*, and *Figure 5.22*:

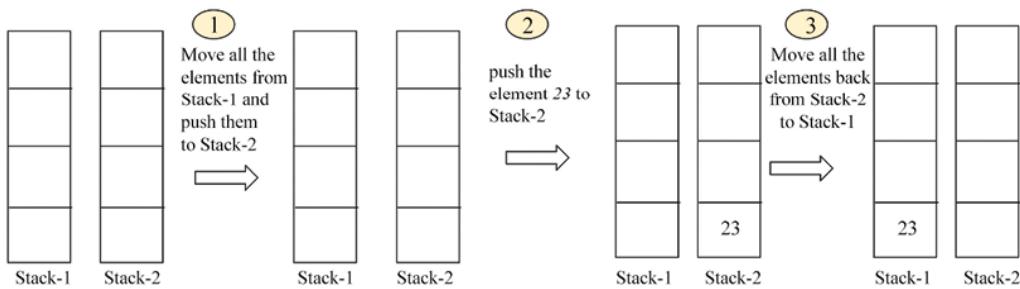


Figure 5.20: Enqueueing element 23 to an empty queue using approach 2

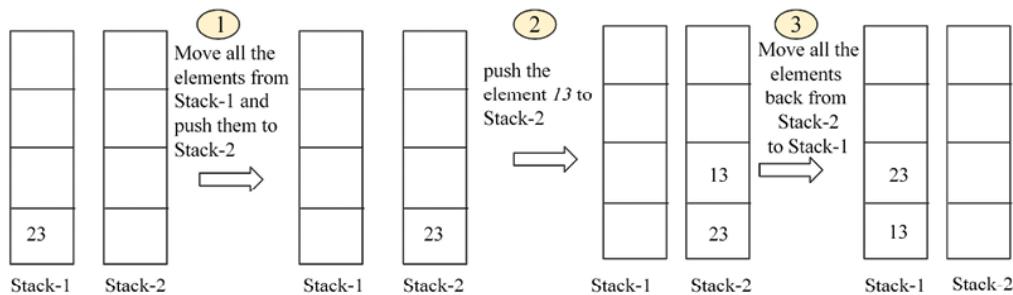


Figure 5.21: Enqueueing element 13 to the existing queue using approach 2

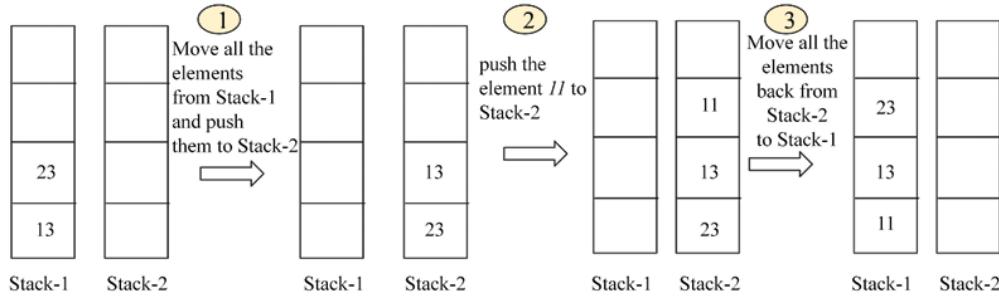


Figure 5.22: Enqueueing element 11 to the queue using approach 2

The dequeue operation can be directly implemented by applying a pop operation to Stack-1. Let's take an example to understand this. Assuming we have already enqueued three elements, and we want to apply the dequeue operation, we can simply pop the top element off the stack, as shown in *Figure 5.23*:

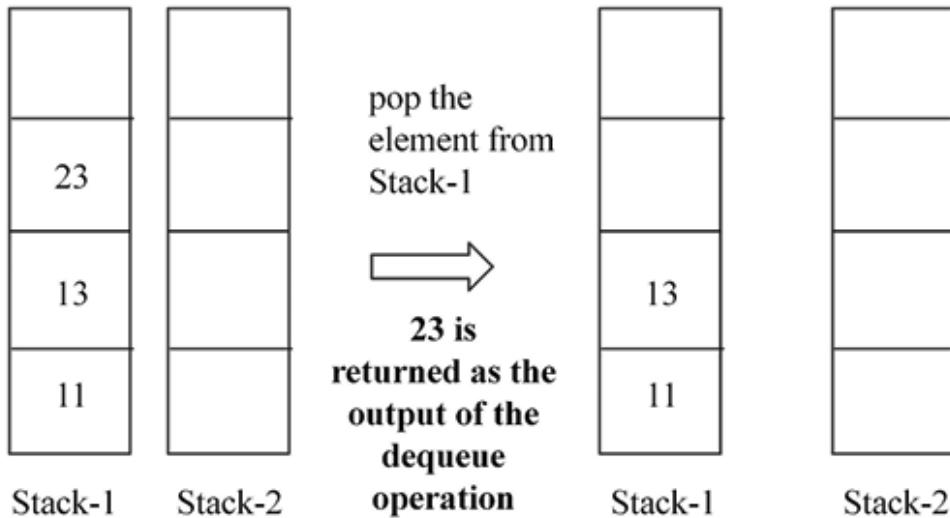


Figure 5.23: Illustration of a dequeue operation on a queue using approach 2

In this second approach, the time complexity for the enqueue operation is $O(n)$, and for the dequeue operation, it is $O(1)$.

Next, we discuss the implementation of a queue using two stacks using approach-1, in which the dequeue operation is costly. In order to implement queues using two stacks, we initially set two stack instance variables to create an empty queue upon initialization. The stacks, in this case, are simply Python lists that allow us to call the `push` and `pop` methods on them, which allow us to get the functionality of the `enqueue` and `dequeue` operations. Here is the `Queue` class:

```
class Queue:  
    def __init__(self):  
        self.Stack1 = []  
        self.Stack2 = []
```

`Stack1` is only used to store elements that are added to the queue. No other operation can be performed on this stack.

Enqueue operation

The `enqueue` method is used to add items to the queue. This method only receives the `data` that is to be appended to the queue. This data is then passed to the `append` method of `Stack1` in the `Queue` class. Further, the `append` method is used to mimic the `push` operation, which pushes elements to the top of the stack. The following code is the implementation of `enqueue` using the stack in Python, which should be defined in the `Queue` class:

```
def enqueue(self, data):  
    self.Stack1.append(data)
```

To enqueue data onto `Stack1`, the following code does the job:

```
queue = Queue()
queue.enqueue(23)
queue.enqueue(13)
queue.enqueue(11)
print(queue.Stack1)
```

The output of `Stack1` on the queue is as follows:

```
[23, 13, 11]
```

Next, we will examine the implementation of the `dequeue` operation.

Dequeue operation

The `dequeue` operation is used to delete the elements from the queue in the same order in which the items were added, according to the **FIFO** principle. New elements are added to the queue in `stack1`. Further, we use another stack, that is, `stack2`, to delete the elements from the queue. The delete (`dequeue`) operation will only be performed through `stack2`. To better understand how `stack2` can be used to delete the items from the queue, let us consider the following example.

Initially, assume that `stack2` was filled with the elements 5, 6, and 7, as shown in *Figure 5.24*:

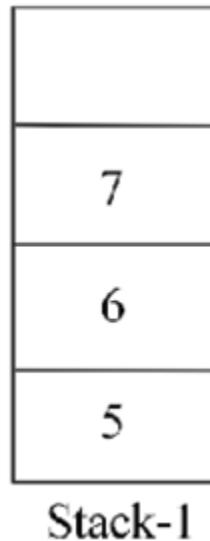


Figure 5.24. Example of Stack1 in a queue

Next, we check if the `stack2` is empty or not. As it is empty at the start, we move all the elements delete from `stack1` to `stack2` using the `pop` operation on `stack1` for all the element and then push them to `stack2`. Now, `stack1` becomes empty and `stack2` has all the elements. We show this in *Figure 5.25* for more clarity:

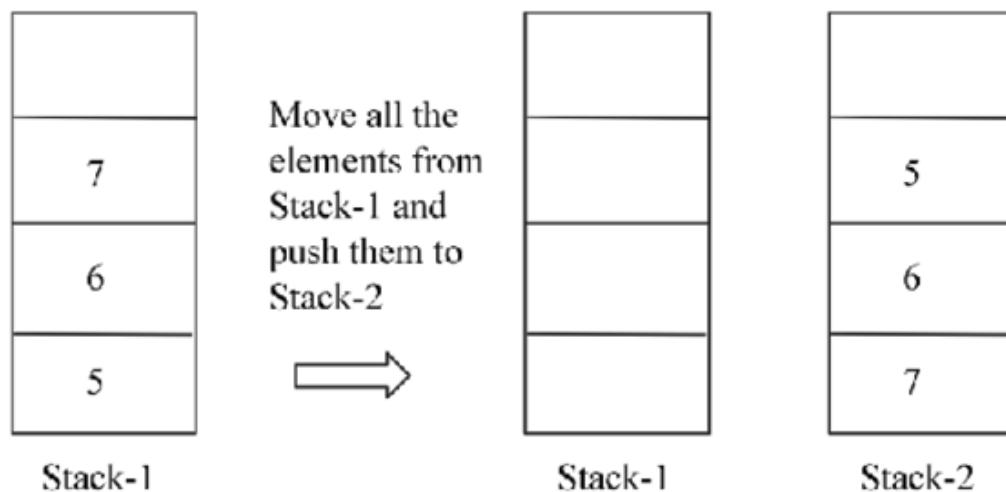


Figure 5.25. Demonstration of Stack1 and Stack2 in a queue

Now, if the `stack` is not empty, in order to pop an element from this queue, we apply the `pop` operation to `stack2`, and we get the element `5`, which is correct as it was added first and should be the first element to be popped off from the queue.

Here is the implementation of the `dequeue` method for the queue, which should be defined in the `Queue` class:

```
def dequeue(self):
    if not self.Stack2:
        while self.Stack1:
            self.Stack2.append(self.Stack1.pop())
    if not self.Stack2:
        print("No element to dequeue")
        return
    return self.Stack2.pop()
```

The `if` statement first checks whether `stack2` is empty. If it is not empty, we proceed to remove the element at the front of the queue using the `pop` method, as follows:

```
return self.Stack2.pop()
```

If `stack2` is empty, all the elements of `stack1` are moved to `stack2`:

```
while self.Stack1:
    self.Stack2.append(self.Stack1.pop())
```

The `while` loop will continue to be executed as long as there are elements in `stack1`.

The `self.Stack1.pop()` statement will remove the last element added to `stack1` and immediately pass the popped data to the

`self.Stack2.append()` method.

Let us consider some example code to understand the operations on the queue. We firstly use the `Queue` implementation to add three items to the queue, that is, `5`, `6`, and `7`. Next, we apply `dequeue` operations to remove items from the queue using the following code:

```
queue = Queue()
queue.enqueue(23)
queue.enqueue(13)
queue.enqueue(11)
print(queue.Stack1)

queue.dequeue()
print(queue.Stack2)
```

The output for the preceding code is as follows:

```
[23, 13, 11]
[13, 11]
```

The preceding code snippet firstly adds elements to a queue and prints out the elements within the queue. Next, the `dequeue` method is called, after which a change in the number of elements is observed when the queue is printed out again.

The enqueue and dequeue operations on the queue data structure using a stack with approach 1 have time complexities of `O(1)`, and `O(n)` respectively. The reason for this is that the enqueue operation is straightforward as a new element can be appended directly, whereas in the dequeue operation, all the n elements need to be accessed and moved to the other stack.

Overall, the linked list-based implementation is the most efficient since both the enqueue and dequeue operations can be performed in $O(1)$ time and there is no constraint on the size of the queue. In the stack-based implementation of queues, one of the operations is costly, be it enqueue or dequeue.

Applications of queues

Queues can be used to implement a variety of functionalities in many real computer-based applications. For instance, instead of providing each computer on a network with its own printer, a network of computers can be made to share one printer by queuing what each computer wants to print. When the printer is ready to print, it will pick one of the items (usually called jobs) in the queue to print out. It will print the command from the computer that has given the command first and will choose the following jobs in the order in which they were submitted by the different computers.

Operating systems also queue processes to be executed by the CPU. Let's create an application that makes use of a queue to create a bare-bones media player.

Most music player software allows users to add songs to a playlist. Upon hitting the play button, all the songs in the main playlist are played one after the other. Sequential playing of the songs can be implemented with queues because the first song to be queued is the first song that is to be played. This aligns with the **FIFO** acronym. We will implement our own playlist queue to play songs in the **FIFO** manner.

Our media player queue will only allow for the addition of tracks and a way to play all the tracks in the queue. In a full-blown music player, threads would be used to improve how the queue is interacted with, while the music player continues to be used to select the next song to be played, paused, or even stopped.

The `track` class will simulate a musical track:

```
from random import randint
class Track:
    def __init__(self, title=None):
        self.title = title
        self.length = randint(5, 10)
```

Each track holds a reference to the title of the song and also the length of the song. The length of the song is a random number between `5` and `10`. The `random` module in Python provides the `randint` function to enable us to generate random numbers. The class represents any MP3 track or file that contains music. The random length of a track is used to simulate the number of seconds it takes to play a track.

To create a few tracks and print out their lengths, we do the following:

```
track1 = Track("white whistle")
track2 = Track("butter butter")
print(track1.length)
print(track2.length)
```

The output of the preceding code is as follows:

Your output may be different depending on the random length generated for the two tracks.

Now, let's create our queue using inheritance. We simply inherit from the `Queue` class:

```
import time
class MediaPlayerQueue(Queue):
```

To add tracks to the queue, an `add_track` method is created in the `MediaPlayerQueue` class:

```
def add_track(self, track):
    self.enqueue(track)
```

The method passes a `track` object to the `enqueue` method of the queue `super` class. This will, in effect, create a `Node` using the `track` object (as the node's data) and point either the tail if the queue is not empty, or both the head and tail if the queue is empty, to this new node.

Assuming the tracks in the queue are played sequentially, from the first track added to the last (**FIFO**), then the `play` function has to loop through the elements in the queue:

```
def play(self):
    while self.count > 0:
        current_track_node = self.dequeue()
        print("Now playing {}".format(current_track_node.data.title))
        time.sleep(current_track_node.data.length)
```

`self.count` keeps count of when a track is added to our queue and when tracks have been dequeued. If the queue is not empty, a call to the `dequeue` method will return the node (which houses the `track` object) at the front of the queue. The `print` statement then accesses the title of the track through the `data` attribute of the node. To further simulate the playing of a track, the `time.sleep()` method halts program execution till the number of seconds for the track has elapsed:

```
time.sleep(current_track_node.data.length)
```

The media player queue is made up of nodes. When a track is added to the queue, the track is hidden in a newly created node and associated with the `data` attribute of the node. That explains why we access a node's `track` object through the `data` property of the node returned by the call to `dequeue`.

You can see that, instead of our `node` object just storing any data, it stores tracks in this case.

Let's take our music player for a spin:

```
track1 = Track("white whistle")
track2 = Track("butter butter")
track3 = Track("Oh black star")
track4 = Track("Watch that chicken")
track5 = Track("Don't go")
```

We create five track objects with random words as titles, as follows:

```
print(track1.length)
print(track2.length)
```

The output is as follows:

```
8
9
```

The output may be different from what you get on your machine due to the random length.

Next, an instance of the `MediaPlayerQueue` class is created using the following code snippet:

```
media_player = MediaPlayerQueue()
```

The tracks will be added, and the output of the `play` function should print out the tracks being played in the same order in which we queued them:

```
media_player.add_track(track1)
media_player.add_track(track2)
media_player.add_track(track3)
media_player.add_track(track4)
media_player.add_track(track5)
media_player.play()
```

The output of the preceding code is as follows:

```
Now playing white whistle
Now playing butter butter
Now playing Oh black star
```

```
Now playing Watch that chicken  
Now playing Don't go
```

Upon execution of the program, it can be seen that the tracks are played in the order in which they were queued. When playing each track, the system also pauses for the number of seconds equal to the length of the track.

Summary

In this chapter, we discussed two important data structures, namely, stacks and queues. We have seen how these data structures closely mimic stacks and queues in the real world. Concrete implementations, together with their varying types, were explored. We later applied the concepts of stacks and queues to write real-life programs.

We will consider trees in the next chapter. The major operations on trees will be discussed, along with the different spheres of application of this data structure.

Exercises

1. Which of the following options is a true queue implementation using linked lists?
 - a. If, in the enqueue operation, new data elements are added at the start of the list, then the dequeue operation must be performed from the end.

- b. If, in the enqueue operation, new data elements are added to the end of the list, then the enqueue operation must be performed from the start of the list.
 - c. Both of the above.
 - d. None of the above.
2. Assume a queue is implemented using a singly-linked list that has head and tail pointers. The enqueue operation is implemented at the head, and the dequeue operation is implemented at the tail of the queue. What will be the time complexity of the enqueue and dequeue operations?
3. What is the minimum number of stacks required to implement a queue?
4. The enqueue and dequeue operations in a queue are implemented efficiently using an array. What will be the time complexity for both of these operations?
5. How can we print the data elements of a queue data structure in reverse order?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



6

Trees

A **tree** is a hierarchical form of data structure. Data structures such as lists, queues, and stacks are linear in that the items are stored in a sequential way. However, a tree is a non-linear data structure, as there is a **parent-child relationship** between the items. The top of the tree's data structure is known as a **root node**. This is the ancestor of all other nodes in the tree.

Tree data structures are very important, owing to their use in various applications, such as parsing expressions, efficient searches, and priority queues. Certain document types, such as `XML` and `HTML`, can also be represented in a tree.

In this chapter, we will cover the following topics:

- Terms and definitions of trees
- Binary trees and binary search trees
- Tree traversal
- Binary search trees

Terminology

Let's consider some of the terminology associated with tree data structures.

To understand trees, we need to first understand the basic concepts related to them. A tree is a data structure in which data is organized in a hierarchical form.

Figure 6.1 contains a typical tree consisting of character nodes lettered **A** through to **M**:

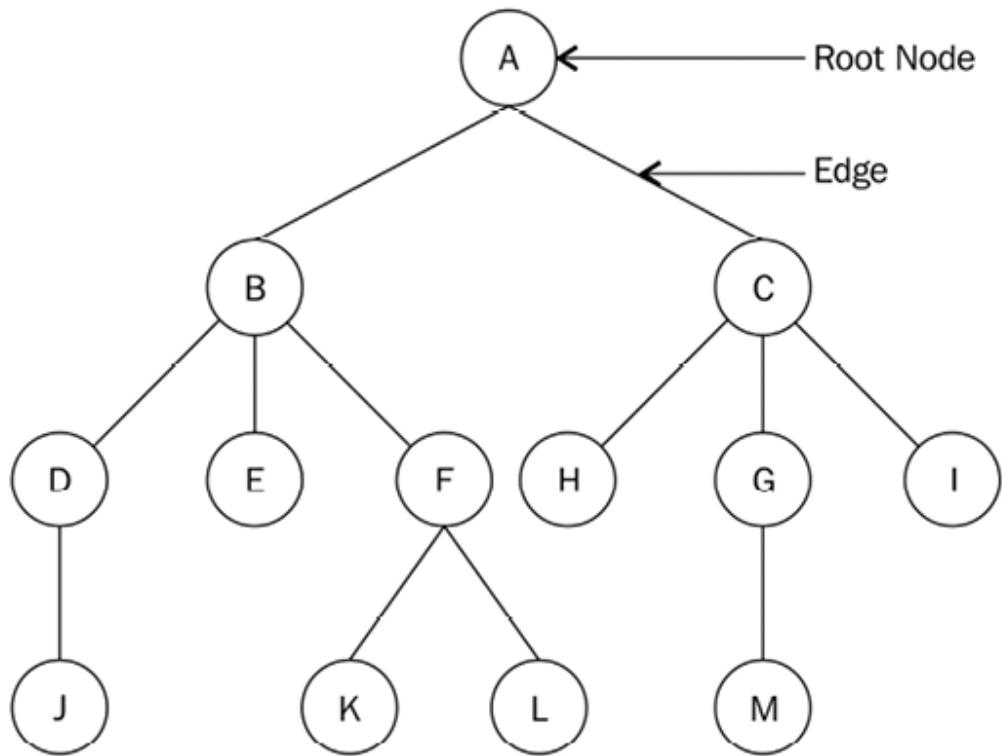


Figure 6.1: Example tree data structure

Here is a list of terms associated with a tree:

- **Node:** Each circled letter in the preceding diagram represents a node. A node is any data structure that stores data.
- **Root node:** The root node is the first node from which all other nodes in the tree descend from. In other words, a root node is a node that does not have a parent node. In every tree, there is

always one unique root node. The root node is node **A** in the above example tree.

- **Subtree:** A subtree is a tree whose nodes descend from some other tree. For example, nodes **F**, **K**, and **L** form a subtree of the original tree.
- **Degree:** The total number of children of a given node is called the **degree of the node**. A tree consisting of only one node has a degree of 0. The degree of node **A** in the preceding diagram is 2, the degree of node **B** is 3, the degree of node **C** is 3, and, the degree of node **G** is 1.
- **Leaf node:** The leaf node does not have any children and is the terminal node of the given tree. The degree of the leaf node is always 0. In the preceding diagram, the nodes **J**, **E**, **K**, **L**, **H**, **M**, and **I** are all leaf nodes.
- **Edge:** The connection among any given two nodes in the tree is called an edge. The total number of edges in a given tree will be a maximum of one less than the total nodes in the tree. An example edge is shown in *Figure 6.1*.
- **Parent:** A node that has a subtree is the parent node of that subtree. For example, node **B** is the parent of nodes **D**, **E**, and **F**, and node **F** is the parent of nodes **K** and **L**.
- **Child:** This is a node that is descendant from a parent node. For example, nodes **B** and **C** are children of parent node **A**, while nodes **H**, **G**, and **I** are the children of parent node **C**.
- **Sibling:** All nodes with the same parent node are siblings. For example, node **B** is the sibling of node **C**, and, similarly, nodes **D**, **E**, and **F** are also siblings.

- **Level:** The root node of the tree is considered to be at level 0. The children of the root node are considered to be at level 1, and the children of the nodes at level 1 are considered to be at level 2, and so on. For example, in *Figure 6.1*, root node **A** is at level 0, nodes **B** and **C** are at level 1, and nodes **D**, **E**, **F**, **H**, **G**, and **I** are at level 2.
- **Height of a tree:** The total number of nodes in the longest path of the tree is the height of the tree. For example, in *Figure 6.1*, the height of the tree is 4, as the longest paths, **A-B-D-J**, **A-C-G-M**, and **A-B-F-K**, all have a total number of four nodes each.
- **Depth:** The depth of a node is the number of edges from the root of the tree to that node. In the preceding tree example, the depth of node **H** is 2.

In linear data structures, data items are stored in sequential order, whereas non-linear data structures store data items in a non-linear order, where a data item can be connected to more than one other data item. All of the data items in linear data structures, such as *arrays*, *lists*, *stacks*, and *queues*, can be traversed in one pass, whereas this is not possible in the case of non-linear data structures such as trees; they store the data differently from other linear data structures.

In a tree data structure, the nodes are arranged in a parent-child relationship. There should not be any cycle among the nodes in trees. The tree structure has nodes to form a hierarchy, and a tree that has no nodes is called an **empty tree**.

First, we'll discuss one of the most important kind of trees, that is, the **binary tree**.

Binary trees

A binary tree is a collection of nodes, where the nodes in the tree can have zero, one, or two child nodes. A simple binary tree has a maximum of two children, that is, the left child and the right child.

For example, in the binary tree shown in *Figure 6.2*, there is a root node that has two children (a left child, a right child):

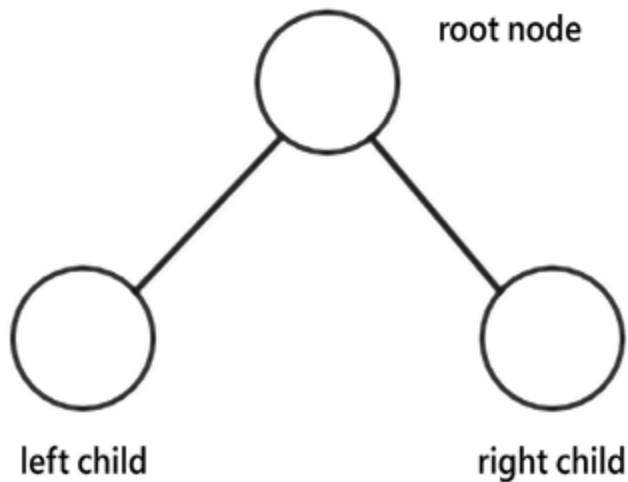


Figure 6.2: Example of a binary tree

The nodes in the binary tree are organized in the form of the left subtree and right subtree. For example, a tree of five nodes is shown in *Figure 6.3* that has a root node, `R`, and two subtrees, i.e. left subtree, `T1`, and right subtree, `T2`:

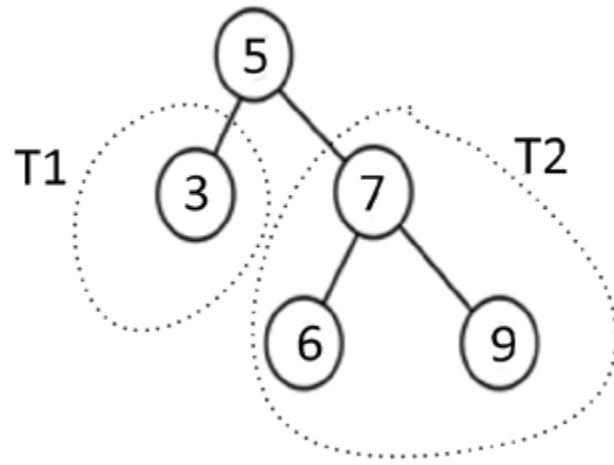


Figure 6.3: An example binary tree of five nodes

A regular binary tree has no other rules as to how elements are arranged in the tree. It should only satisfy the condition that each node should have a maximum of two children.

A tree is called a **full binary** tree if all the nodes of a binary tree have either zero or two children, and if there is no node that has one child. An example of a full binary tree is shown in *Figure 6.4*:

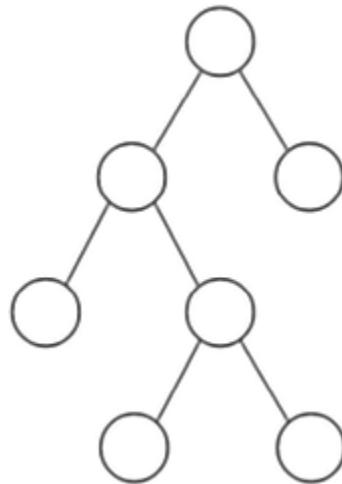


Figure 6.4: An example of a full binary tree

A perfect binary tree has all the nodes in the binary tree filled, and it doesn't have space vacant for any new nodes; if we add new nodes, they can only be added by increasing the tree's height. A sample perfect binary tree is shown in *Figure 6.5*:

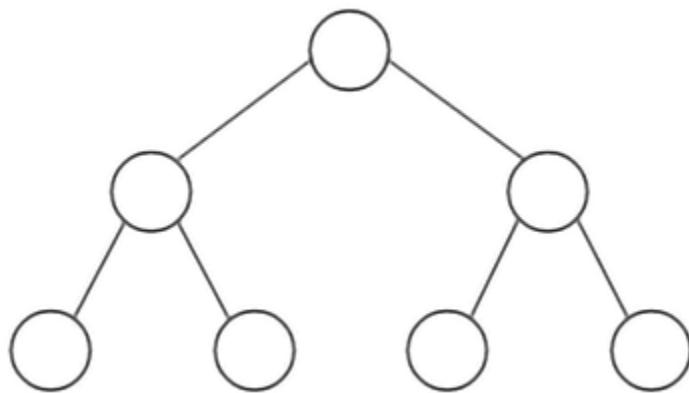


Figure 6.5: An example of a perfect binary tree

A **complete binary tree** is filled with all possible nodes except with a possible exception at the lowest level of the tree. All nodes are also filled on the left side. A complete binary tree is shown in *Figure 6.6*:

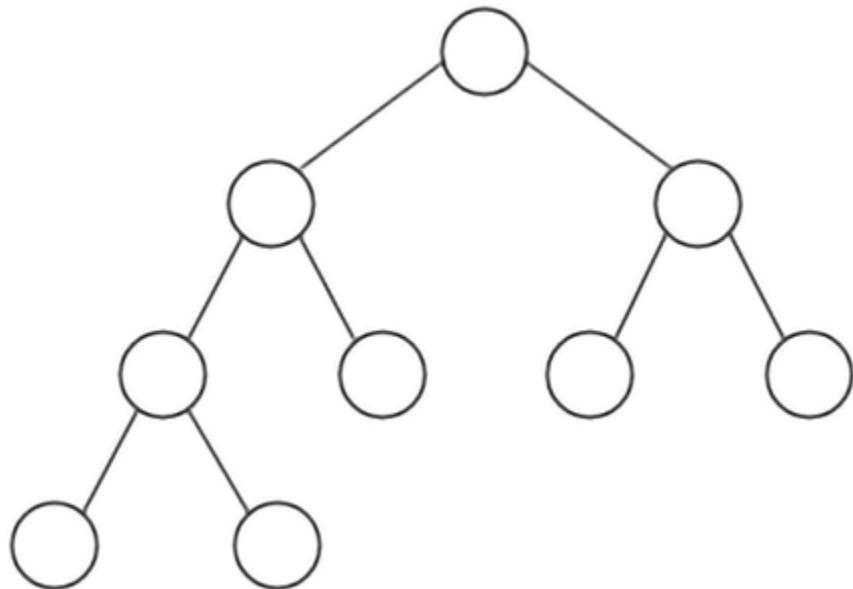
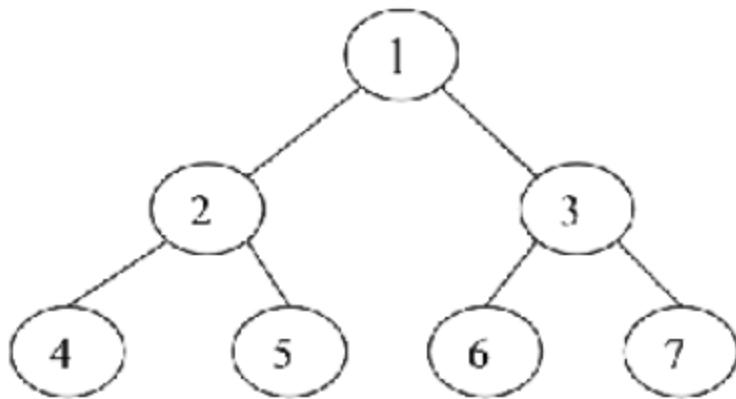


Figure 6.6: An example of a complete binary tree

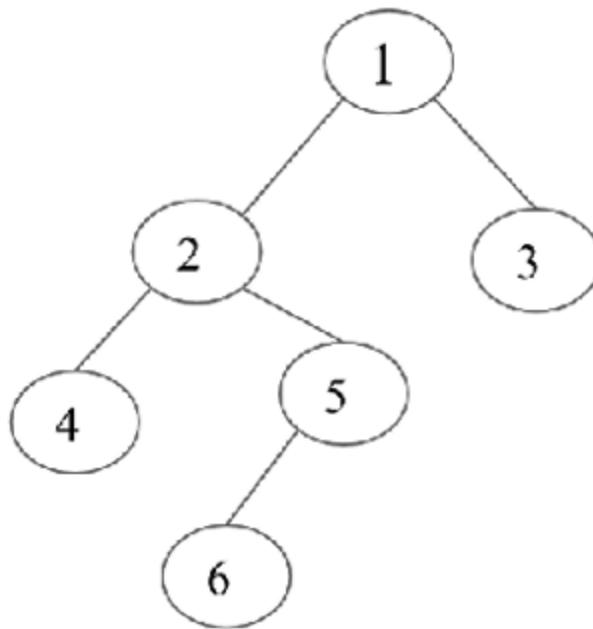
A binary tree can be balanced or unbalanced. In a balanced binary tree, the difference in height of the left and right subtrees for every node in the tree is no more than 1. A balanced tree is shown in *Figure 6.7*:



Balanced tree

Figure 6.7: An example of a balanced tree

An unbalanced binary tree is a binary tree that has a difference of more than 1 between the right subtree and left subtree. An example of an unbalanced tree is shown in *Figure 6.8*:



Unbalanced tree

Figure 6.8: An example of an unbalanced tree

Next, we'll discuss the details of the implementation of a simple binary tree.

Implementation of tree nodes

As we have already discussed in previous chapters, a node consists of data items and references to other nodes.

In a binary tree node, each node will contain data items and two references that will point to their left and right children, respectively. Let's look at the following code for building a binary tree `Node` class in Python:

```
class Node:  
    def __init__(self, data):  
        self.data = data
```

```
self.right_child = None  
self.left_child = None
```

To better understand the working of this class, let's first create a binary tree of four nodes—`n1`, `n2`, `n3`, and `n4`—as shown in *Figure 6.9*:

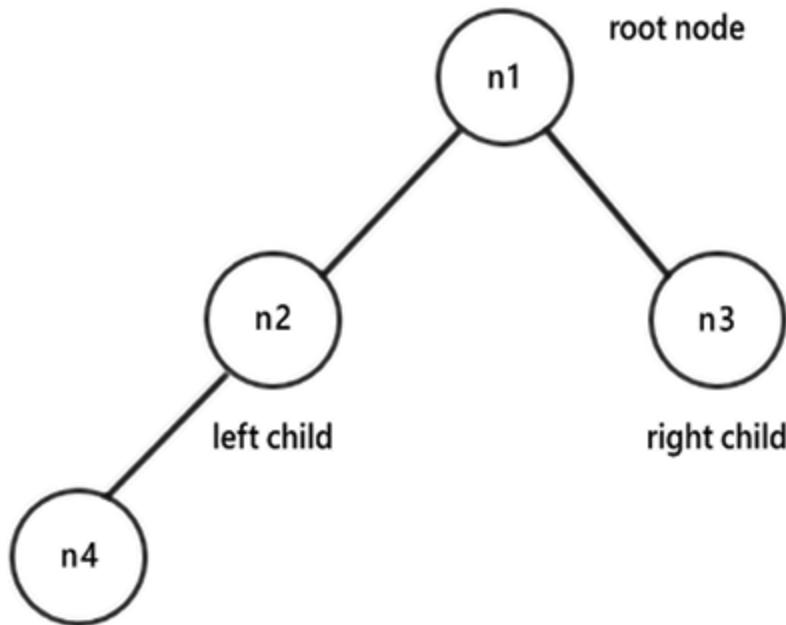


Figure 6.9: An example binary tree of four nodes

For this, we firstly create four nodes—`n1`, `n2`, `n3`, and `n4`:

```
n1 = Node("root node")  
n2 = Node("left child node")  
n3 = Node("right child node")  
n4 = Node("left grandchild node")
```

Next, we connect the nodes to each other according to the previously discussed properties of a binary tree. `n1` will be the root node, with `n2` and `n3` as its children. Furthermore, `n4` will be the left child of `n2`.

The next code snippet shows the connections among different nodes of the tree according to the desired tree, as shown in *Figure 6.9*:

```
n1.left_child = n2  
n1.right_child = n3  
n2.left_child = n4
```

Here, we have created a very simple tree structure of four nodes. After creating a tree, one of the most important operations that is to be applied to trees is **traversal**. Next, we'll understand how we can traverse the tree.

Tree traversal

The method to visit all the nodes in a tree is called **tree traversal**. In the case of a linear data structure, data element traversal is straightforward since all the items are stored in a sequential manner, so each data item is visited only once. However, in the case of non-linear data structures, such as trees and graphs, traversal algorithms are important. To understand traversing, let's traverse the left subtree of the binary tree we created in the previous section. For this, we start from the root node, print out the node, and move down the tree to the next left node. We keep doing this until we have reached the end of the left subtree, like so:

```
current = n1  
while current:  
    print(current.data)  
    current = current.left_child
```

The output of traversing the preceding code block is as follows:

```
root node  
left child node  
left grandchild node
```

There are multiple ways to process and traverse the tree that depend upon the sequence of visiting the root node, left subtree, or right subtree. Mainly, there are two kinds of approaches, firstly, one in which we start from a node and traverse every available child node, and then continue to traverse to the next sibling. There are three possible variations of this method, namely, **in-order**, **pre-order**, and **post-order**. Another approach to traverse the tree is to start from the root node and then visit all the nodes on each level, and process the nodes level by level. We will discuss each approach in the following sections.

In-order traversal

In-order tree traversal works as follows: we start traversing the left subtree recursively, and once the left subtree is visited, the root node is visited, and then finally the right subtree is visited recursively. It has the following three steps:

- We start traversing the left subtree and call an ordering function recursively
- Next, we visit the root node
- Finally, we traverse the right subtree and call an ordering function recursively

So, in a nutshell, for in-order tree traversal, we visit the nodes in the tree in the order of left subtree, root, then the right subtree.

Let's consider an example tree shown in *Figure 6.10* to understand in-order tree traversal:

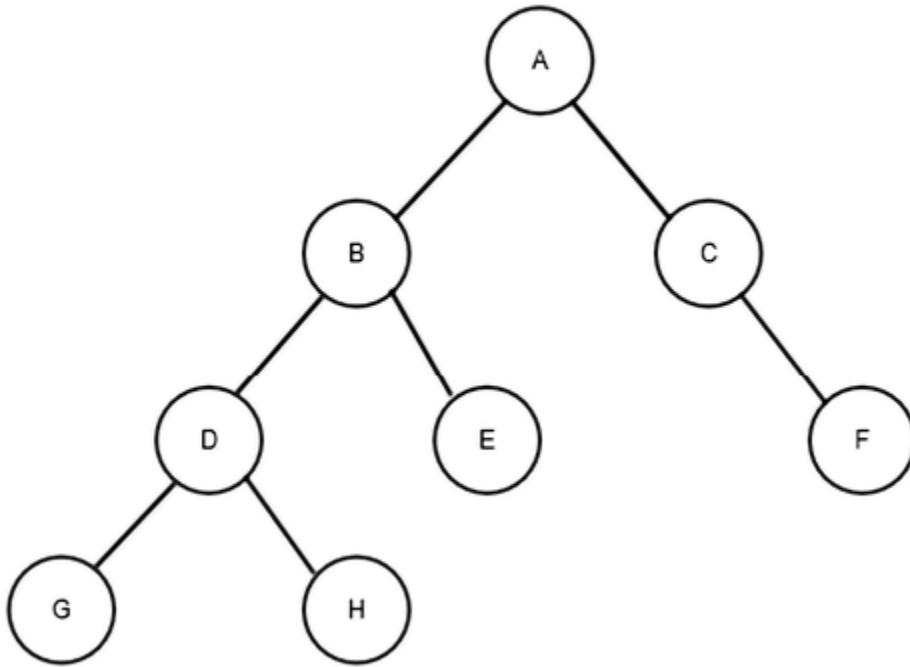


Figure 6.10: An example binary tree for in-order tree traversal

In the binary tree shown in *Figure 6.10*, the working of the in-order traversal is as follows: first, we recursively visit the left subtree of the root node **A**. The left subtree of node **A** has node **B** as the root node, so we again go to the left subtree of root node **B**, that is, node **D**. We recursively go to the left subtree of root node **D** so that we get the left child of root node **D**. We visit the left child, **G**, then visit the root node, **D**, and then visit the right child, **H**.

Next, we visit node **B** and then visit node **E**. In this manner, we have visited the left subtree of root node **A**. Next, we visit root node **A**. After that, we visit the right subtree of root node **A**. Here, we first go to the left subtree of root node **C**, which is null, so next, we visit node **C**, and then we visit the right child of node **C**, that is, node **F**.

Therefore, the in-order traversal for this example tree is G-D-H-B-E-A-C-F.

The Python implementation of a recursive function to return an in-order listing of nodes in a tree is as follows:

```
def inorder(root_node):
    current = root_node
    if current is None:
        return
    inorder(current.left_child)
    print(current.data)
    inorder(current.right_child)
inorder(n1)
```

Firstly, we check if the current node is null or empty. If it is not empty, we traverse the tree. We visit the node by printing the visited node. In this case, we first recursively call the `inorder` function with `current.left_child`, then we visit the root node, and finally, we recursively call the `inorder` function with `current.right_child`.

Finally, when we apply the above in-order traversal algorithm on the above sample tree of four nodes. With `n1` as the root node, we get the following output:

```
left grandchild node
left child node
root node
right child node
```

Next, we will discuss pre-order traversal.

Pre-order traversal

Pre-order tree traversal traverses the tree in the order of the root node, the left subtree, and then the right subtree. It works as follows:

1. We start traversing with the root node
2. Next, we traverse the left subtree and call an ordering function with the left subtree recursively
3. Next, we visit the right subtree and call an ordering function with the right subtree recursively

Consider the example tree shown in *Figure 6.11* to understand pre-order traversal:

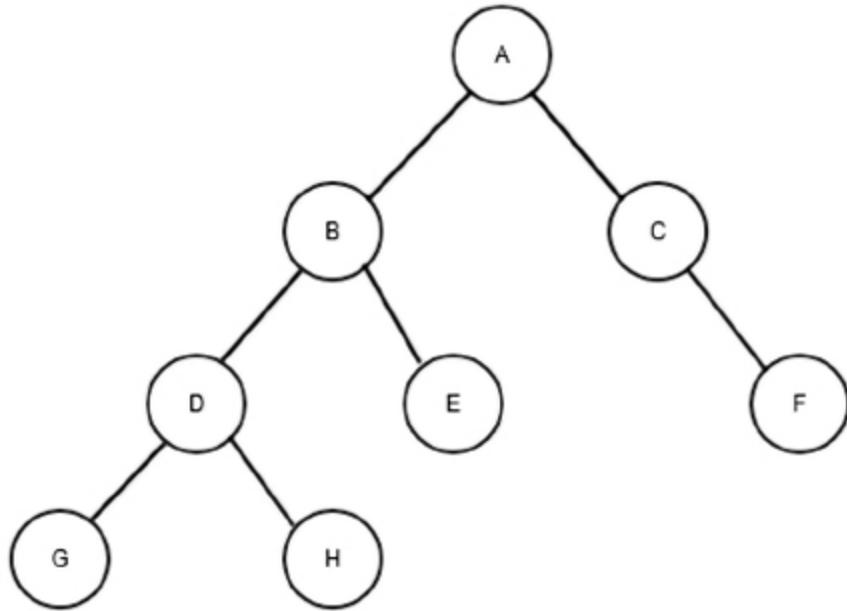


Figure 6.11: An example tree to understand pre-order traversal

The pre-order traversal for the example binary tree shown in *Figure 6.11* works as follows: first, we visit root node **A**. Next, we go to the left subtree of root node **A**. The left subtree of node **A** has node **B** as the root, so we visit this root node, and then go to the left subtree of root node **B**, node **D**. We visit node **D** and then the left subtree of root

node **D**, and then we visit the left child, **G**, which is the subtree of root node **D**. Since there is no child of node **G**, we visit the right subtree. We visit the right child of the subtree of root node **D**, node **H**. Next, we visit the right child of the subtree of root node **B**, node **E**.

In this manner, we have visited root node **A** and the left subtree of root node **A**. Next, we visit the right subtree of root node **A**. Here, we visit root node **C**, and then we go to the left subtree of root node **C**, which is null, so we visit the right child of node **C**, node **F**.

The pre-order traversal for this example tree would be **A-B-D-G-H-E-C-F**.

The recursive function for the pre-order tree traversal is as follows:

```
def preorder(root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    preorder(current.left_child)
    preorder(current.right_child)
preorder(n1)
```

First, we check if the current node is null or empty. If it is empty, it means the tree is an empty tree, and if the current node is not empty, then we traverse the tree using the pre-order algorithm. The pre-order traversal algorithm traverses the tree in the order of root, left subtree, and right subtree recursively, as shown in the above code. Finally, when we apply the above pre-order traversal algorithm on the above sample tree of four nodes with **n1** node as the root node, we get the following output:

```
root node  
left child node  
left grandchild node  
right child node
```

Next, we will discuss post-order traversal.

Post-order traversal

Post-order tree traversal works as follows:

1. We start traversing the left subtree and call an ordering function recursively
2. Next, we traverse the right subtree and call an ordering function recursively
3. Finally, we visit the root node

So, in a nutshell, for post-order tree traversal, we visit the nodes in the tree in the order of the left subtree, the right subtree, and finally the root node.

Consider the following example tree shown in *Figure 6.12* to understand post-order tree traversal:

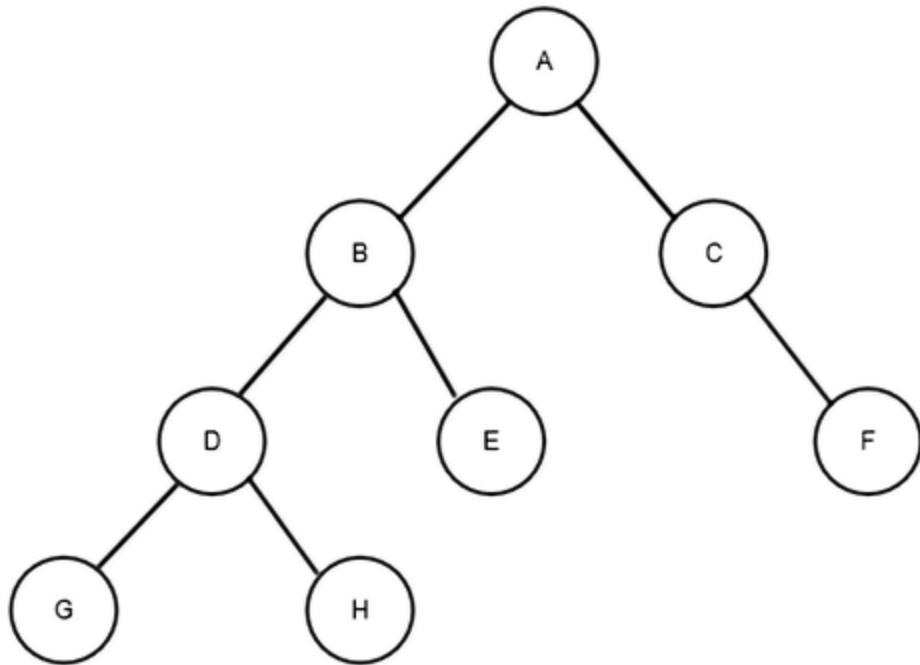


Figure 6.12: An example tree to understand pre-order traversal

In the preceding figure, *Figure 6.12*, we first visit the left subtree of root node **A** recursively. We get to the last left subtree, that is, root node **D**, and then we visit the node to the left of it, which is node **G**. We visit the right child, **H**, after this, and then we visit root node **D**. Following the same rule, we next visit the right child of node **B**, node **E**. Then, we visit node **B**. Following on from this, we traverse the right subtree of node **A**. Here, we first reach the last right subtree and visit node **F**, and then we visit node **C**. Finally, we visit root node **A**.

The post-order traversal for this example tree would be **G-H-D-E-B-F-C-A**.

The implementation of the post-order method for tree traversal is as follows:

```

def postorder( root_node):
    current = root_node
  
```

```
if current is None:  
    return  
postorder(current.left_child)  
postorder(current.right_child)  
print(current.data)  
postorder(n1)
```

First, we check if the current node is null or empty. If it is not empty, we traverse the tree using the post-order algorithm as discussed, and finally, when we apply the above post-order traversal algorithm on the above sample tree of four nodes with `n1` as the root node. We get the following output:

```
left grandchild node  
left child node  
right child node  
root node
```

Next, we will discuss level-order traversal.

Level-order traversal

In this traversal method, we start by visiting the root of the tree before visiting every node on the next level of the tree. Then, we move on to the next level in the tree, and so on. This kind of tree traversal is how breadth-first traversal in a graph works, as it broadens the tree by traversing all the nodes in a level before going deeper into the tree.

Let's consider the following example tree and traverse it:

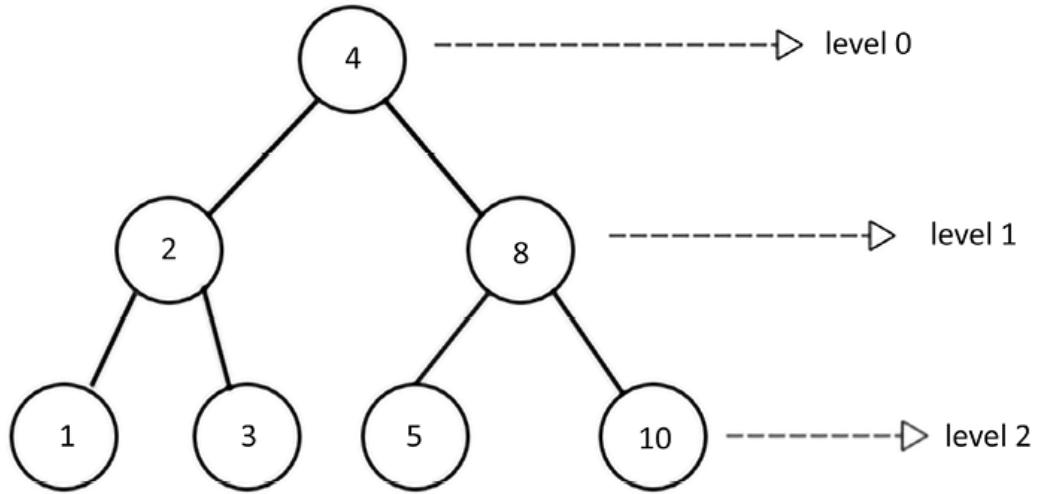


Figure 6.13: An example tree to understand level-order traversal

In *Figure 6.13*, we start by visiting the root node at level 0, which is the node with a value of 4. We visit this node by printing out its value. Next, we move to level 1 and visit all the nodes at this level, which are the nodes with the values 2 and 8. Finally, we move to the next level in the tree, that is, level 3, and we visit all the nodes at this level, which are 1, 3, 5, and 10. Thus, the level-order tree traversal for this tree is as follows: 4, 2, 8, 1, 3, 5, and 10.

This level-order tree traversal is implemented using a queue data structure. We start by visiting the root node, and we push it into a queue. The node at the front of the queue is accessed (dequeued), which can then be either printed or stored for later use. After adding the root node, the left child node is added to the queue, followed by the right node. Thus, when traversing at any given level of the tree, all the data items of that level are firstly inserted in the queue from left to right. After that, all the nodes are visited from the queue one by one. This process is repeated for all the levels of the tree.

The traversal of the preceding tree using this algorithm will enqueue root node 4, dequeue it, and visit the node. Next, nodes 2 and 8 are enqueued, as they are the left and right nodes at the next level. Node 2 is dequeued so that it can be visited. Next, its left and right nodes, nodes 1 and 3, are enqueued. At this point, the node at the front of the queue is node 8. We dequeue and visit node 8, after which we enqueue its left and right nodes. This process continues until the queue is empty.

The Python implementation of breadth-first traversal is as follows. We enqueue the root node and keep a list of the visited nodes in the `list_of_nodes` list. The `deque` class is used to maintain a queue:

```
from collections import deque
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4

def level_order_traversal(root_node):
    list_of_nodes = []
    traversal_queue = deque([root_node])
    while len(traversal_queue) > 0:
        node = traversal_queue.popleft()
        list_of_nodes.append(node.data)
        if node.left_child:
            traversal_queue.append(node.left_child)
        if node.right_child:
```

```
        traversal_queue.append(node.right_child)
    return list_of_nodes
print(level_order_traversal(n1))
```

If the number of elements in `traversal_queue` is greater than zero, the body of the loop is executed. The node at the front of the queue is popped off and added to the `list_of_nodes` list. The first `if` statement will `enqueue` the left child node if the `node` provided with a left node exists. The second `if` statement does the same for the right child node. Further, the `list_of_nodes` list is returned in the last statement.

The output of the above code is as follows:

```
[ 'root node', 'left child node', 'right child node', 'left grandchild'
```

We have discussed different tree traversal algorithms; we can use any of these algorithms depending upon the application. In-order traversal is very useful when we need sorted contents from a tree. This also applies if we need items in descending order, which we can do by reversing the order, such as right subtree, root, and then left subtree. This is known as reverse in-order traversal. And, if we need to inspect the root before any leaves, we use pre-order traversal. Likewise, if we need to inspect the leaf nodes before the root nodes.

The following are some important applications of binary trees:

1. Binary trees as expression trees are used in compilers
2. It is also used in Huffman coding in data compression
3. Binary search trees are used for efficient searching, insertion, and deletion of a list of items

4. **Priority Queue (PQ)**, which is used for finding and deleting minimum or maximum items in a collection of elements in logarithm time in the worst case

Next, let us discuss expression trees.

Expression trees

An expression tree is a special kind of binary tree that can be used to represent arithmetic expressions. An arithmetic expression is represented by a combination of operators and operands, where the operators can be unary or binary. Here, the operator shows which operation we want to perform, and the operator tells us what data items we want to apply those operations to. If the operator is applied to one operand, then it is called a unary operator, and if it is applied to two operands, it is called a binary operator.

An arithmetic expression can also be represented using a binary tree, which is also known as an expression tree. The **infix** notation is a commonly used notation to express arithmetic expressions where the operators are placed in between the operands. It is a commonly used method of representing an arithmetic expression. In an expression tree, all the leaf nodes contain operands and non-leaf nodes contain the operators. It is also worth noting that an expression tree will have one of its subtrees (right or left) empty in the case of a unary operator.

The arithmetic expression is shown using three notations: **infix**, **postfix**, or **prefix**. The in-order traversal of an expression tree

produces the infix notation. For example, the expression tree for $3 + 4$ would look as shown in *Figure 6.14*:

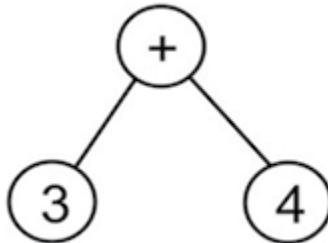
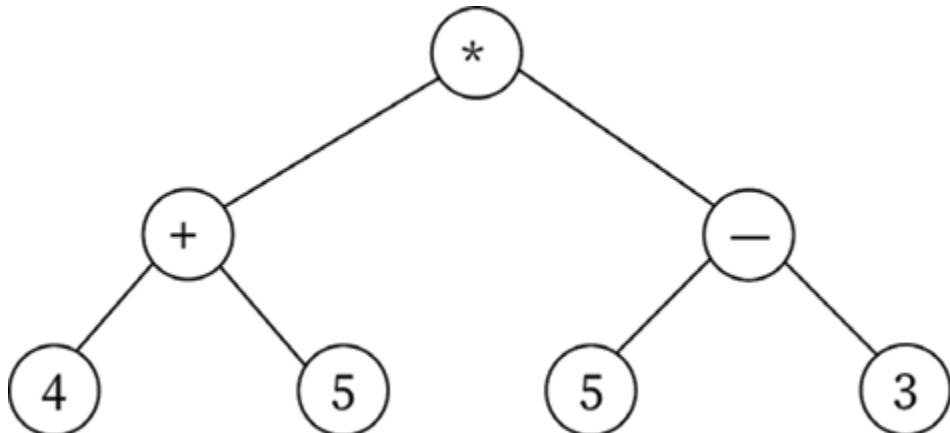


Figure 6.14: An expression tree for the expression $3 + 4$

In this example, the operator is inserted (infix) between the operands, as $3 + 4$. When necessary, parentheses can be used to build a more complex expression. For example, for $(4 + 5) * (5 - 3)$, we would get the following:



*Figure 6.15: An expression tree for the expression $(4 + 5) * (5 - 3)$*

Prefix notation is commonly referred to as *Polish* notation. In this notation, the operator comes before its operands. For example, the arithmetic expression to add two numbers, 3 and 4, would be shown as $+ 3 4$. Let's consider another example, $(3 + 4) * 5$. This can also be represented as $* (+ 3 4) 5$ in prefix notation. The pre-order traversal

of an expression tree results in the prefix notation of the arithmetic expression. For example, consider the expression tree shown in *Figure 6.16*:

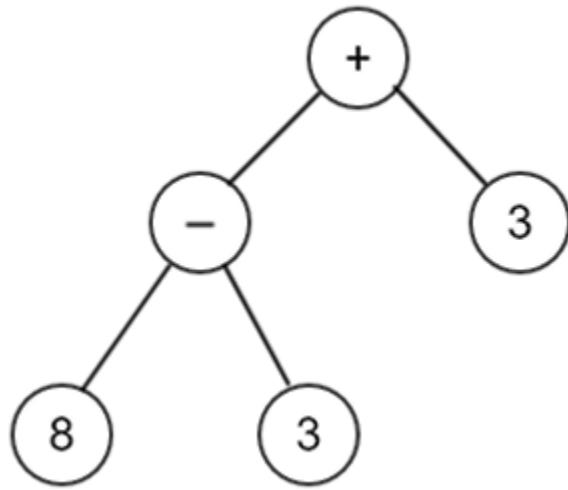


Figure 6.16: An example expression tree to understand pre-order traversal

The pre-order traversal of the expression tree shown in *Figure 6.16* will give the expression in prefix notation as `+ - 8 3 3`.

Postfix, or **reverse Polish notation (RPN)**, places the operator after its operands, such as `3 4 +`. The post-order traversal of the expression tree shown in *Figure 6.17* gives the postfix notation of the arithmetic expression.

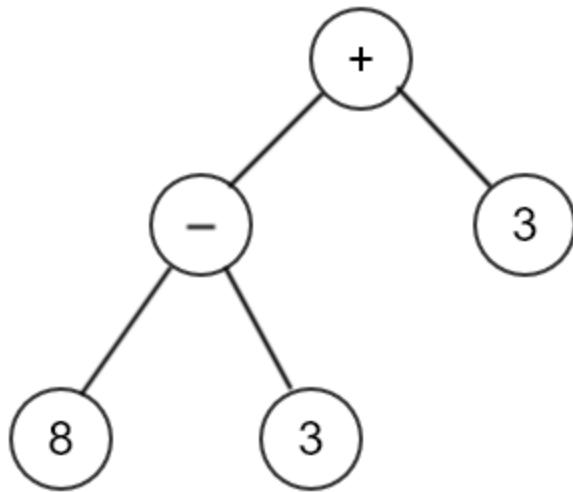


Figure 6.17: An example expression tree to understand post-order traversal

The postfix notation for the preceding expression tree is `8 3 -3 +`. We have now discussed expression trees. It is easy to evaluate an expression tree for the given arithmetic expression using the reverse Polish notation since it provides faster calculations.

Parsing a reverse Polish expression

To create an expression tree from the postfix notation, a stack is used. In this, we process one symbol at a time; if the symbol is an operand, then its references are pushed in to the stack, and if the symbol is an operator, then we pop two pointers from the stack and form a new subtree, whose root is the operator. The first reference popped from the stack is the right child of the subtree, and the second reference becomes the left child of the subtree. Further, a reference to this new subtree is pushed into the stack. In this manner, all the symbols of the postfix notation are processed to create the expression tree.

Let's take an example of `4 5 + 5 3 - *`.

Firstly, we push symbols `4` and `5` onto the stack, and then we process the next symbol `+` as shown in *Figure 6.18*:

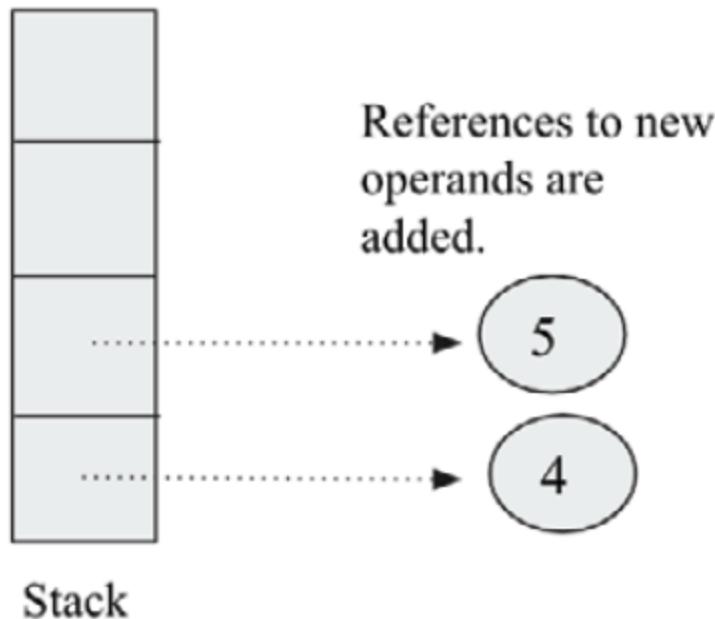


Figure 6.18: Operands 4 and 5 are pushed onto the stack

When the new symbol `+` is read, it is made into a root node of a new subtree, and then two references are popped from the stack, and the topmost reference is added as the right of the root node, and the next popped reference is added as the left child of the subtree, as shown in *Figure 6.19*:

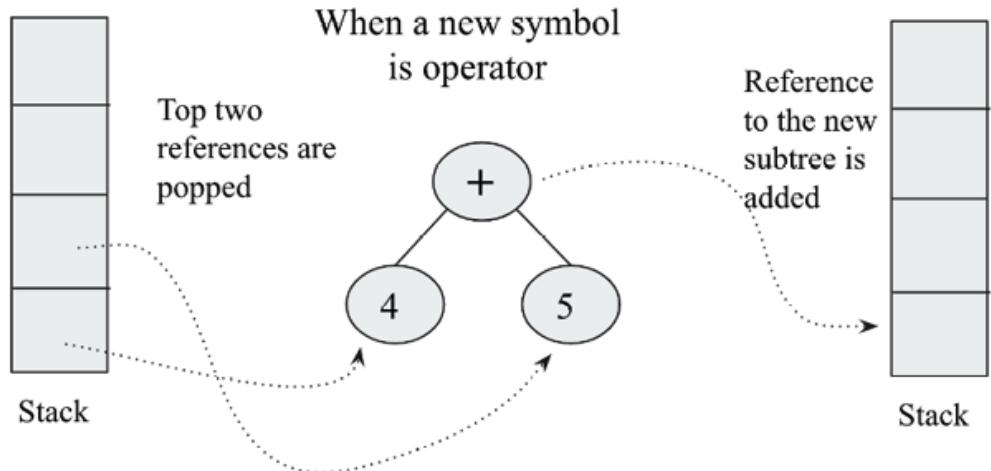


Figure 6.19: Operator + is processed in creating an expression tree

The next symbols are 5 and 3, and they are pushed into the stack. Next, when a new symbol is an operator (-), it is created as the root of the new subtree, and two top references are popped and added to the right and left child of this root respectively, as shown in *Figure 6.20*. Then, the reference to this subtree is pushed to the stack:

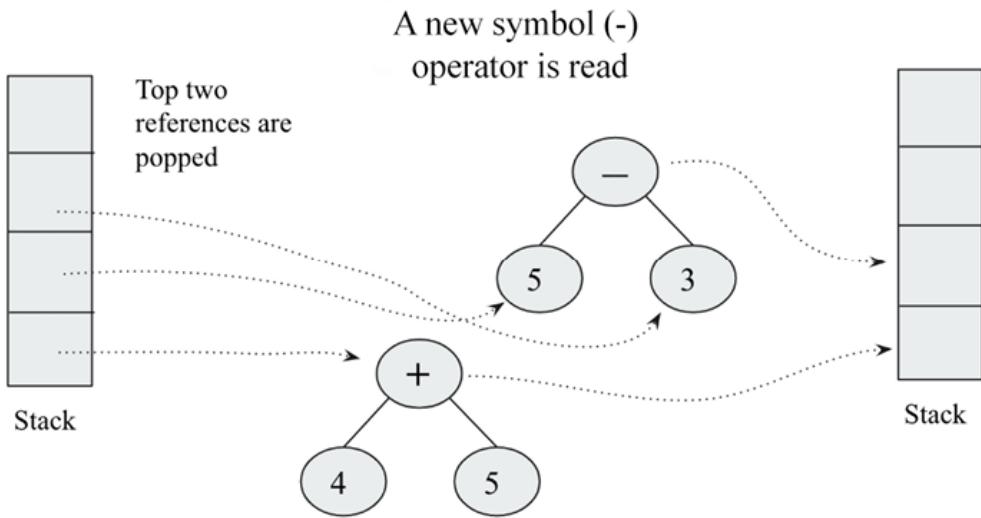


Figure 6.20: Operator (-) is processed in creating an expression tree

The next symbol is the operator `*`; as we have done so far, this will be created as the root, and then two references will be popped from the stack, as shown in *Figure 6.21*. The final tree is then shown in *Figure 6.21*:

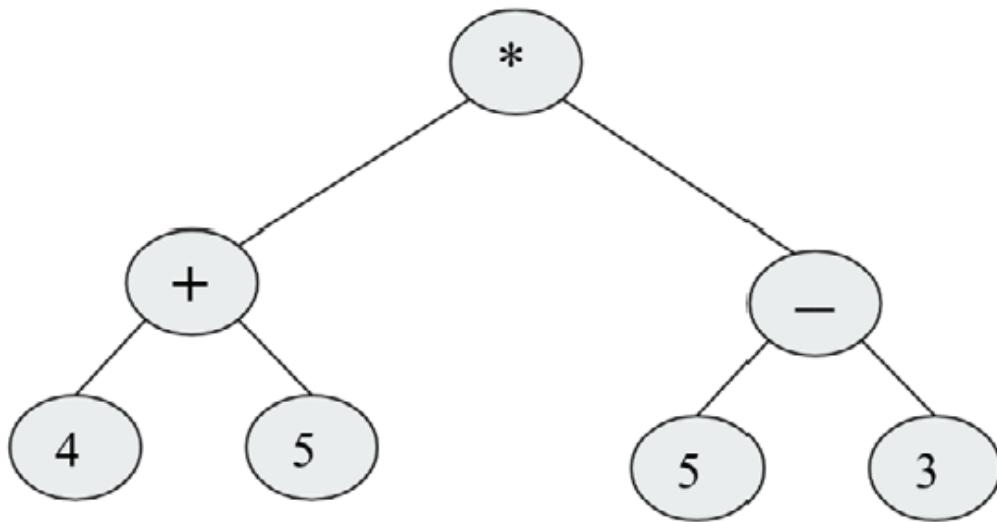


Figure 6.21: Operator () is processed in creating an expression tree*

To learn how to implement this algorithm in Python, we will look at building a tree for an expression written in postfix notation. For this, we need a tree node implementation; it can be defined as follows:

```
class TreeNode:  
    def __init__(self, data=None):  
        self.data = data  
        self.right = None  
        self.left = None
```

The following is the code for the implementation of the stack class that we will be using:

```
class Stack:  
    def __init__(self):
```

```
    self.elements = []

    def push(self, item):
        self.elements.append(item)

    def pop(self):
        return self.elements.pop()
```

In order to build the tree, we are going to enlist the items with the help of a stack. Let's take an example of an arithmetic expression and set up our stack:

```
expr = "4 5 + 5 3 - *".split()
stack = Stack()
```

In the first statement, the `split()` method splits on whitespace by default. The `expr` is a list with the values `4`, `5`, `+`, `5`, `3`, `-`, and `*`.

Each element of the `expr` list is going to be either an operator or an operand. If we get an operand, then we embed it in a tree node and push it onto the stack. If we get an operator, we embed the operator into a tree node and pop its two operands into the node's right and left children. Here, we have to take care to ensure that the first `pop` reference goes into the right child.

In continuation of the previous code snippet, the below code is a loop to build the tree:

```
for term in expr:
    if term in "+-*":
        node = TreeNode(term)
        node.right = stack.pop()
        node.left = stack.pop()
    else:
```

```
    node = TreeNode(int(term))
    stack.push(node)
```

Notice that we perform a conversion from `string` to `int` in the case of an operand. You could use `float()` instead, if you wish to support floating-point operands.

At the end of this operation, we should have one single element in the stack, and that holds the full tree.

If we want to evaluate the expression, we can use the following function:

```
def calc(node):
    if node.data == "+":
        return calc(node.left) + calc(node.right)
    elif node.data == "-":
        return calc(node.left) - calc(node.right)
    elif node.data == "*":
        return calc(node.left) * calc(node.right)
    elif node.data == "/":
        return calc(node.left) / calc(node.right)
    else:
        return node.data
```

In the preceding code, we pass a node to the function. If the node contains an operand, then we simply return that value. If we get an operator, then we perform the operation that the operator represents on the node's two children. However, since one or more of the children could also contain either operators or operands, we call the `calc()` function recursively on the two child nodes (bearing in mind that all the children of every node are also nodes).

Now, we just need to pop the root node off the stack and pass it onto the `calc()` function. Then, we should have the result of the calculation:

```
root = stack.pop()
result = calc(root)
print(result)
```

Running this program should yield the result `18`, which is the result of `(4 + 5) * (5 - 3)`.

Expression trees are very useful in representing and evaluating complex expressions easily. It is also useful to evaluate the postfix, prefix, and infix expression. It can be used to find out the associativity of the operators in the given expression.

In the next section, we will discuss the binary search tree, which is a special kind of binary tree.

Binary search trees

A **binary search tree (BST)** is a special kind of binary tree. It is one of the most important and commonly used data structures in computer science applications. A binary search tree is a tree that is structurally a binary tree, and stores data in its nodes very efficiently. It provides very fast search, insertion, and deletion operations.

A binary tree is called a binary search tree if the value at any node in the tree is greater than the values in all the nodes of its left subtree, and less than (or equal to) the values of all the nodes of the right

subtree. For example, if k_1 , k_2 , and k_3 are key values in a tree of three nodes (as shown in *Figure 6.22*), then it should satisfy the following conditions:

- The key values $K_2 \leq K_1$
- The key values $K_3 > K_1$

The following figure depicts the above condition of the binary search tree:

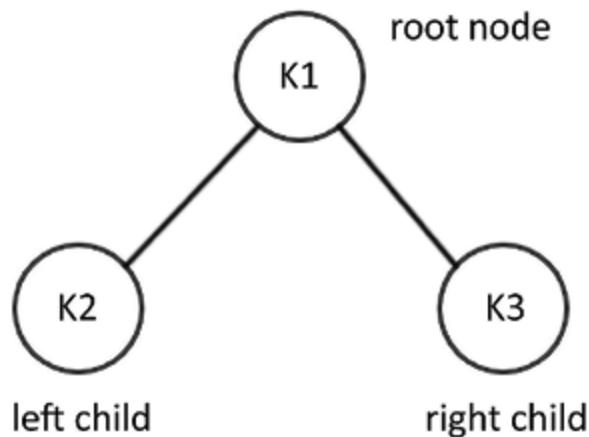


Figure 6.22: An example of a binary search tree

Let's consider another example so that we have a better understanding of binary search trees. Consider the binary search tree shown in *Figure 6.23*:

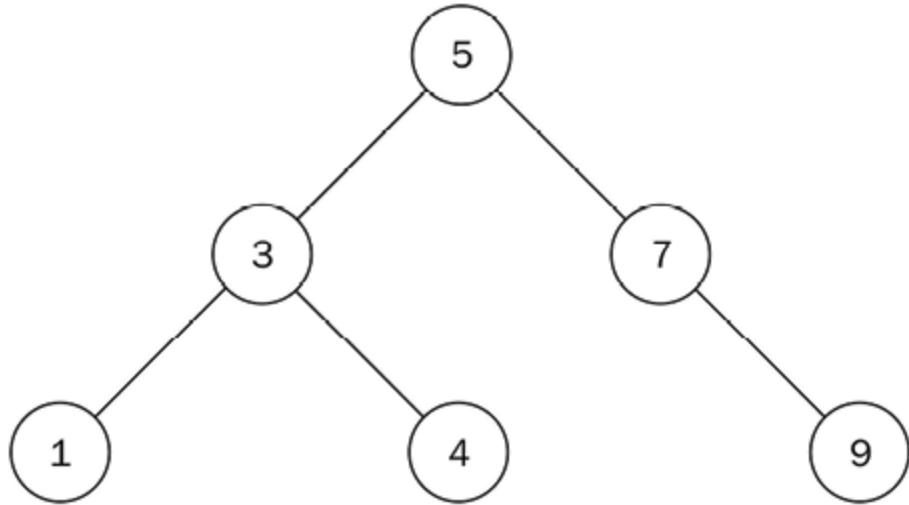


Figure 6.23: Binary search tree of six nodes

In this tree, all the nodes in the left subtree are less than (or equal to) the value of the parent node. All the nodes in the right subtree of this node are also greater than that of the parent node.

To see if the above example tree fulfills the properties of a binary search tree, we see that all the nodes in the left subtree of the root node have a value less than 5. Likewise, all the nodes in the right subtree have a value that is greater than 5. This property applies to all the nodes in the tree with no exceptions. For example, if we take another node with the value 3, we can see that the values for all the left subtree nodes are less than the value 3 and the values for all the right subtree nodes are greater than 3.

Considering another example of a binary tree. Let's check to see if it is a binary search tree. Despite the fact that the following diagram, *Figure 6.24*, looks similar to the previous diagram, it does not qualify as a binary search tree, as node 7 is greater than the root node 5; even though it is located in the left subtree of the root node. Node 4 is to the right subtree of its parent node 7, which is also violating a

rule of binary search trees. Thus, the following figure, *Figure 6.24*, is not a binary search tree:

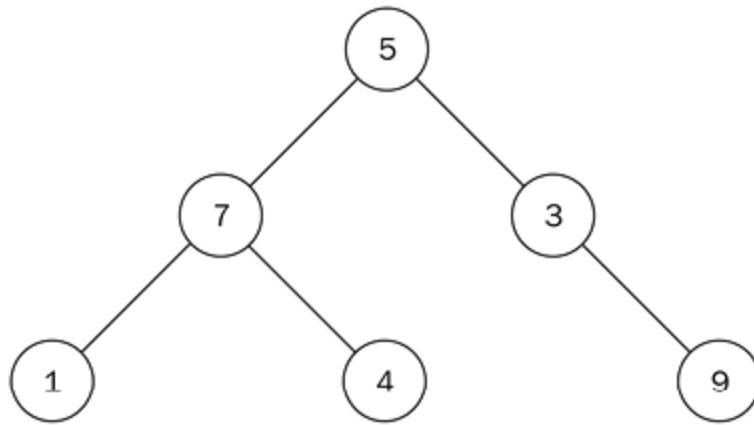


Figure 6.24: An example of a binary tree that is not a binary search tree

Let's begin the implementation of a binary search tree in Python. Since we need to keep track of the root node of the tree, we start by creating a `Tree` class that holds a reference to the root node:

```
class Tree:  
    def __init__(self):  
        self.root_node = None
```

That's all it takes to maintain the state of a tree. Now, let's examine the main operations used within the binary search tree.

Binary search tree operations

The operations that can be performed on a binary search tree are `insert`, `delete`, `finding min`, `finding max`, and `searching`. We discuss them in detail one by one in the following sections.

Inserting nodes

One of the most important operations to implement on a binary search tree is to insert data items in the tree. In order to insert a new element into a binary search tree, we have to ensure that the properties of the binary search tree are not violated after adding the new element.

In order to insert a new element, we start by comparing the value of the new node with the root node: if the value is less than the root value, then the new element will be inserted into the left subtree; otherwise, it will be inserted into the right subtree. In this manner, we go to the end of the tree to insert the new element.

Let's create a binary search tree by inserting data items 5, 3, 7, and 1 in the tree. Consider the following:

1. **Insert 5:** We start with the first data item, 5. To do this, we will create a node with its data attribute set to 5, since it is the first node.
2. **Insert 3:** Now, we want to add the second node with a value of 3 so that the data value of 3 is compared with the existing node value, 5, of the root node. Since the node value 3 is less than 5, it will be placed in the left subtree of node 5. The binary search tree will look as shown in *Figure 6.25*:

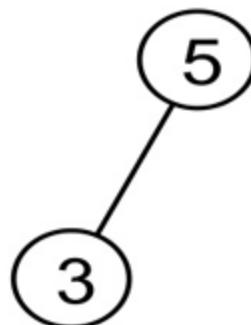


Figure 6.25: Step 2 of the insertion operation in an example binary search tree

Here, the tree satisfies the binary search tree rule, where all the nodes in the left subtree are less than the parent.

3. **Insert 7:** To add another node with a value of 7 to the tree, we start from the root node with value 5 and make a comparison, as shown in *Figure 6.26*. Since 7 is greater than 5, the node with a value of 7 is placed to the right of this root:



Figure 6.26: Step 3 of the insertion operation in an example binary search tree

4. **Insert 1:** Next, we add another node with the value 1. Starting from the root of the tree, we make a comparison between 1 and 5, as shown in *Figure 6.27*:

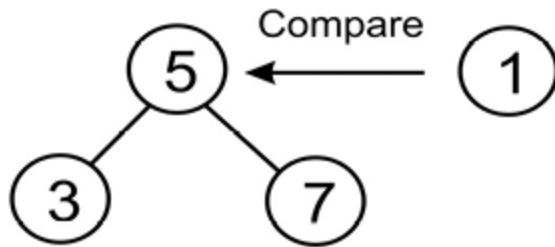


Figure 6.27: Step 4 of the insertion operation in an example binary search tree

This comparison shows that 1 is less than 5, so we go to the left subtree of 5, which has a node with a value of 3, as shown in *Figure 6.28*:

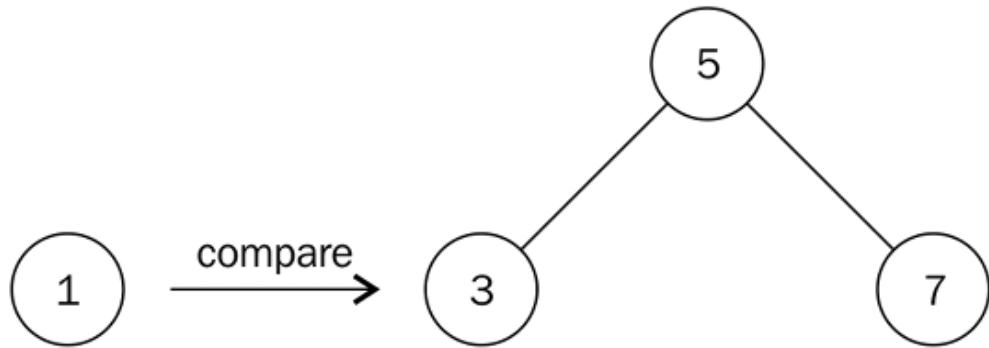


Figure 6.28: Comparison of node 1 and node 3 in an example binary search tree

When we compare 1 against 3, 1 is less than 3, so we move a level below node 3 and to its left, as shown in *Figure 6.28*. However, there is no node there. Therefore, we create a node with a value of 1 and associate it with the left pointer of node 3 to obtain the final tree. Here, we have the final binary search tree of 4 nodes, as shown in *Figure 6.29*:

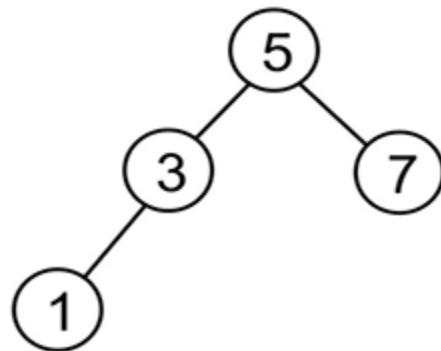


Figure 6.29: Final step of the insertion operation in an example binary search tree

We can see that this example contains only integers or numbers. So, if we need to store string data in a binary search tree, the strings would be compared alphabetically.

If we wanted to store any custom data types inside a binary search tree, we would have to make sure that the binary search tree class supports ordering.

The Python implementation of the `insert` method to add the nodes in the binary search tree is given as follows:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

class Tree:
    def __init__(self):
        self.root_node = None
    def insert(self, data):
        node = Node(data)
        if self.root_node is None:
            self.root_node = node
            return self.root_node
        else:
            current = self.root_node
            parent = None
            while True:
                parent = current
                if node.data < parent.data:
                    current = current.left_child
                    if current is None:
                        parent.left_child = node
                        return self.root_node
                else:
                    current = current.right_child
                    if current is None:
                        parent.right_child = node
                        return self.root_node
```

In the above code, we first declare the `Node` class with the `Tree` class. All the operations that can be applied to the tree are defined in the

`Tree` class. Let's understand the steps of the `insert` method. We begin with a function declaration:

```
def insert(self, data):
```

Next, we encapsulate the data in a node using the `Node` class. We check whether we have a root node or not. If we don't have a root node in the tree, the new node becomes the root node and then root node is returned:

```
node = Node(data)
if self.root_node is None:
    self.root_node = node
    return self.root_node
else:
```

Further, in order to insert a new element, we have to traverse the tree and reach the correct position where we can insert the new element in a way that the properties of the binary search tree are not violated. For this, we keep track of the current node while traversing the tree as well as its parent. The `current` variable is always used to track where a new node will be inserted:

```
current = self.root_node
parent = None
while True:
    parent = current
```

Here, we must perform a comparison. If the data held in the new node is less than the data held in the current node, then we check

whether the current node has a left child node. If it doesn't, this is where we insert the new node. Otherwise, we keep traversing:

```
if node.data < parent.data:  
    current = current.left_child  
    if current is None:  
        parent.left_child = node  
        return self.root_node
```

After this, we need to take care of the greater than (or equal to) case. If the current node doesn't have a right child node, then the new node is inserted as the right child node. Otherwise, we move down and continue looking for an insertion point:

```
else:  
    current = current.right_child  
    if current is None:  
        parent.right_child = node  
        return self.root_node
```

Now, in order to see what we have inserted in the binary search tree, we can use any of the existing tree traversal algorithms. Let's implement the in-order traversal, which should be defined in the `Tree` class. The code is as follows:

```
def inorder(self, root_node):  
    current = root_node  
    if current is None:  
        return  
    self.inorder(current.left_child)  
    print(current.data)  
    self.inorder(current.right_child)
```

Now, let us take an example to insert a few elements (e.g. elements 5, 2, 7, 9, and 1) in a binary search tree, as shown in *Figure 6.24*, and then we can use the in-order traversal algorithm to see what we have inserted in the tree:

```
tree = Tree()
r = tree.insert(5)
r = tree.insert(2)
r = tree.insert(7)
r = tree.insert(9)
r = tree.insert(1)

tree.inorder(r)
```

The output of the above code is as follows:

```
1
2
5
7
9
```

Insertion of a node in a binary search tree takes $O(h)$, where h is the height of the tree.

Searching the tree

A binary search tree is a tree data structure in which all the nodes in the left subtree of a node have lower key values and the right subtree has greater key values. Thus, searching for an element with a given key value is quite easy. Let's consider an example binary search tree that has nodes 1, 2, 3, 4, 8, 5, and 10, as shown in *Figure 6.30*:

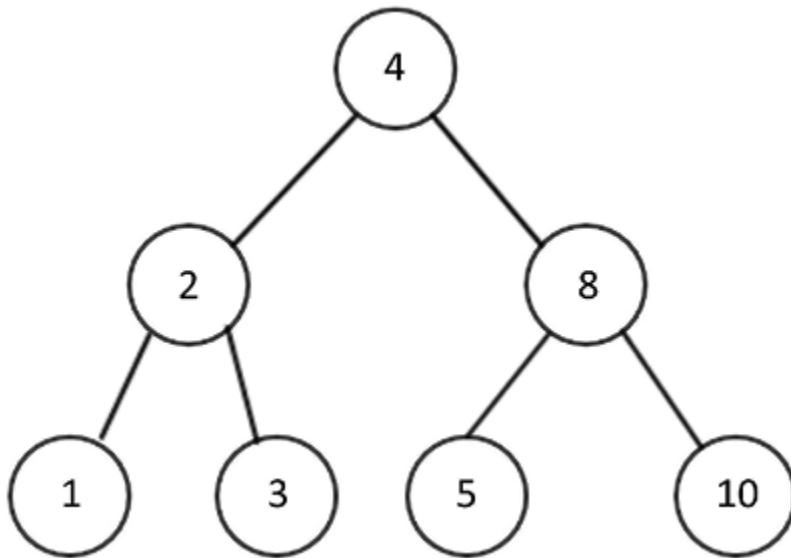


Figure 6.30: An example binary search tree with seven nodes

In the preceding tree shown in *Figure 6.30*, if we wish to search for a node with a value of `5`, for example, then we start from the root node and compare the root with our desired value. As node `5` is a greater value than the root node's value of `4`, we move to the right subtree. In the right subtree, we have node `8` as the root node, so we compare node `5` with node `8`. As the node to be searched has a smaller value than node `8`, we move it to the left subtree. When we move to the left subtree, we compare the left subtree node `5` with the required node value of `5`. This is a match, so we return `"item found"`.

Here is the implementation of the searching method in a binary search tree, which is being defined in the `Tree` class:

```

def search(self, data):
    current = self.root_node
    while True:
        if current is None:
            print("Item not found")
            return None

```

```
        elif current.data is data:
            print("Item found", data)
            return data
        elif current.data > data:
            current = current.left_child
        else:
            current = current.right_child
```

In the preceding code, we return the data if it was found, or `None` if the data wasn't found. We start searching from the root node. Next, if the data item to be searched for doesn't exist in the tree, we return `None`. If we find the data, it is returned.

If the data that we are searching for is less than that of the current node, we go down the tree to the left. Furthermore, in the `else` part of the code, we check if the data we are looking for is greater than the data held in the current node, which means that we go down the tree to the right.

Finally, the below code can be used to create an example binary search tree with some values between 1 and 10. Then, we search for a data item with the value `9`, and also all the numbers in that range. The ones that exist in the tree get printed:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)
tree.search(9)
```

The output of the above code is as follows:

Item found 9

In the above code, we see the items that were present in the tree have been correctly found; the rest of the items could not be found in the range 1 to 10. In the next section, we discuss the deletion of a node in binary search tree.

Deleting nodes

Another important operation on a binary search tree is the deletion or removal of nodes. There are three possible scenarios that we need to take care of during this process. The node that we want to remove might have the following:

- **No children:** If there is no leaf node, directly remove the node
- **One child:** In this case, we swap the value of that node with its child, and then delete the node
- **Two children:** In this case, we first find the in-order successor or predecessor, swap their values, and then delete that node

The first scenario is the easiest to handle. If the node about to be removed has no children, we can simply remove it from its parent. In *Figure 6.31*, suppose we want to delete node **A**, which has no children. In this case, we can simply delete it from its parent (node **Z**):



Figure 6.31: Deletion operation when deleting a node with no children

In the second scenario, when the node we want to remove has one child, the parent of that node is made to point to the child of that particular node. Let's take a look at the following diagram, where we want to delete node **6**, which has one child, node **5**, as shown in *Figure 6.32*:

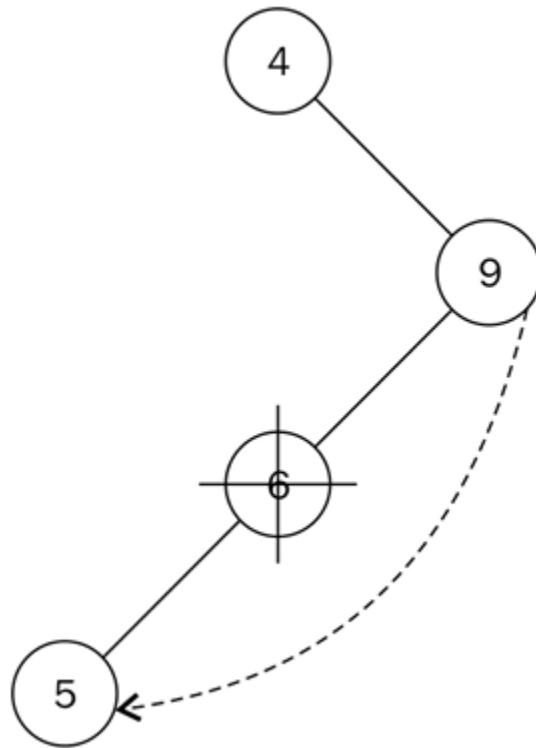


Figure 6.32: Deletion operation when deleting a node with one child

In order to delete node **6**, which has node **5** as its only child, we point the left pointer of node **9** to node **5**. Here, we need to ensure that the child and parent relationship follows the properties of a binary search tree.

In the third scenario, when the node we want to delete has two children, in order to delete it, we first find a successor node, then

move the content of the successor node into the node to be deleted. The successor node is the node that has the minimum value in the right subtree of the node to be deleted; it will be the first element when we apply the in-order traversal on the right subtree of the node to be deleted.

Let's understand it with the example tree shown in *Figure 6.33*, where we want to delete node `9`, which has two children:

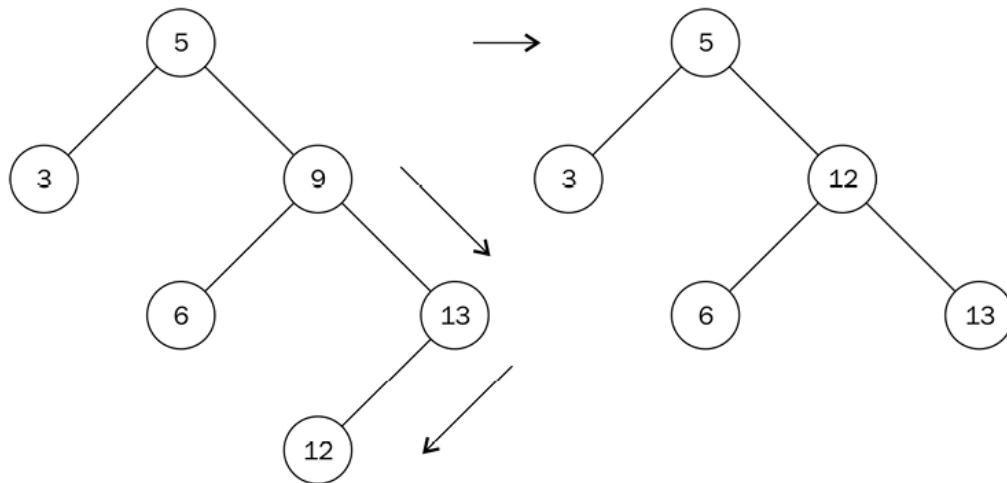


Figure 6.33: Deletion operation when deleting a node with two children

In the example tree shown in *Figure 6.33*, we find the smallest element in the right subtree of the node (i.e. the first element in the in-order traversal in the right subtree) which is node `12`. After that, we replace the value of node `9` with the value `12` and remove node `12`. Node `12` has no children, so we apply the rule for removing nodes without children accordingly.

To implement the above algorithm using Python, we need to write a helper method to get the node that we want to delete along with the reference to its parent node. We have to write a separate method

because we do not have any reference to the parent in the `Node` class. This helper method `get_node_with_parent` is similar to the `search` method, which finds the node to be deleted, and returns that node with its parent node:

```
def get_node_with_parent(self, data):
    parent = None
    current = self.root_node
    if current is None:
        return (parent, None)
    while True:
        if current.data == data:
            return (parent, current)
        elif current.data > data:
            parent = current
            current = current.left_child
        else:
            parent = current
            current = current.right_child
    return (parent, current)
```

The only difference is that before we update the `current` variable inside the loop, we store its parent with `parent = current`. The method to do the actual removal of a node begins with this search:

```
def remove(self, data):
    parent, node = self.get_node_with_parent(data)
    if parent is None and node is None:
        return False
    # Get children count
    children_count = 0
    if node.left_child and node.right_child:
        children_count = 2
    elif (node.left_child is None) and (node.right_child is None):
        children_count = 0
    else:
        children_count = 1
```

We pass the parent and the found nodes to `parent` and `node`, respectively, with the `parent, node = self.get_node_with_parent(data)` line. It is important to know the number of children that the node has that we want to delete, and we do so in the `if` statement.

Once we know the number of children a node has that we want to delete, we need to handle various conditions in which a node can be deleted. The first part of the `if` statement handles the case where the node has no children:

```
if children_count == 0:  
    if parent:  
        if parent.right_child is node:  
            parent.right_child = None  
        else:  
            parent.left_child = None  
    else:  
        self.root_node = None
```

In cases where the node to be deleted has only one child, the `elif` part of the `if` statement does the following:

```
elif children_count == 1:  
    next_node = None  
    if node.left_child:  
        next_node = node.left_child  
    else:  
        next_node = node.right_child  
    if parent:  
        if parent.left_child is node:  
            parent.left_child = next_node  
        else:  
            parent.right_child = next_node
```

```
    else:  
        self.root_node = next_node
```

`next_node` is used to keep track of that single node, which is the child of the node that is to be deleted. We then connect `parent.left_child` or `parent.right_child` to `next_node`.

Lastly, we handle the condition where the node we want to delete has two children:

```
else:  
    parent_of_leftmost_node = node  
    leftmost_node = node.right_child  
    while leftmost_node.left_child:  
        parent_of_leftmost_node = leftmost_node  
        leftmost_node = leftmost_node.left_child  
    node.data = leftmost_node.data
```

In finding the in-order successor, we move to the right node with `leftmost_node = node.right_child`. As long as a left node exists, `leftmost_node.left_child` will be `True` and the `while` loop will run. When we get to the leftmost node, it will either be a leaf node (meaning that it will have no child node) or have a right child.

We update the node that's about to be removed with the value of the in-order successor with `node.data = leftmost_node.data`:

```
if parent_of_leftmost_node.left_child == leftmost_node:  
    parent_of_leftmost_node.left_child = leftmost_node.r  
else:  
    parent_of_leftmost_node.right_child = leftmost_node.r
```

The preceding statement allows us to properly attach the parent of the leftmost node with any child node. Observe how the right-hand side of the equals sign stays unchanged. This is because the in-order successor can only have a right child as its only child.

The following code demonstrates how to use the remove method in the `Tree` class:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)
tree.search(9)
tree.remove(9)
tree.search(9)
```

The output of the above code is:

```
Item found 9
Item not found
```

In the above code, when we search for item `9`, it is available in the tree, and after the remove method, item `9` is not present in the tree. In the worst-case scenario, the `remove` operation takes $O(h)$, where `h` is the height of the tree.

Finding the minimum and maximum nodes

The structure of the binary search tree makes searching a node that has a maximum or a minimum value very easy. To find a node that has the smallest value in the tree, we start traversal from the root of

the tree and visit the left node each time until we reach the end of the tree. Similarly, we traverse the right subtree recursively until we reach the end to find the node with the biggest value in the tree.

For example, consider *Figure 6.34*, in order to search for the minimum and maximum elements.

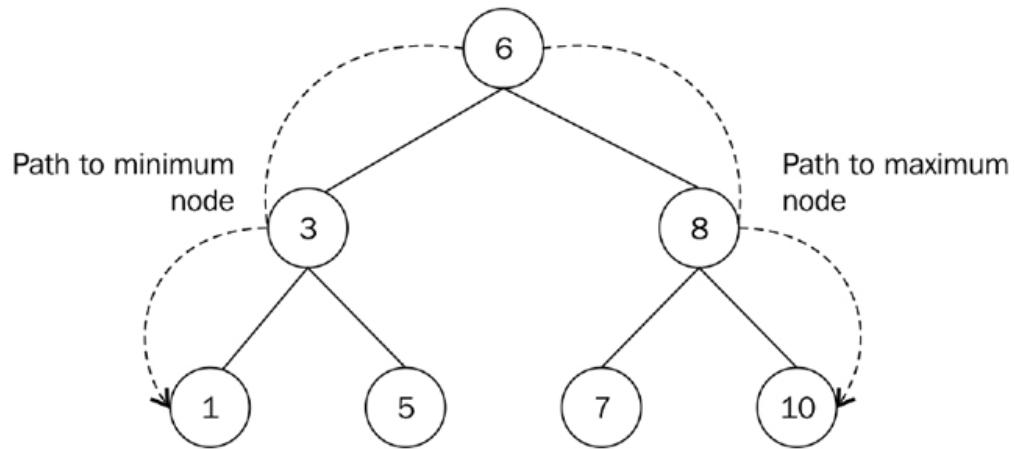


Figure 6.34: Finding the minimum and maximum nodes in a binary search tree

Here, we start by moving down the tree from root node 6 to 3, and then from node 3 to 1 to find the node with the smallest value. Similarly, to find the maximum value node from the tree, we go down from the root along the right-hand side of the tree, so we go from node 6 to node 8 and then node 8 to node 10 to find the node with the largest value.

The Python implementation of the method that returns the minimum value of any node is as follows:

```
def find_min(self):  
    current = self.root_node  
    while current.left_child:
```

```
    current = current.left_child
    return current.data
```

The `while` loop continues to get the left node and visits it until the last left node points to `None`. It is a very simple method.

Similarly, the following is the code of the method that returns the maximum node:

```
def find_max(self):
    current = self.root_node
    while current.right_child:
        current = current.right_child
    return current.data
```

The following code demonstrates how to use the `find_min` and `find_max` methods in the `Tree` class:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)
print(tree.find_min())
print(tree.find_max())
```

The output of the above code is as shown below:

```
1
9
```

The output of the above code, `1` and `9`, are the minimum and maximum values. The minimum value in the tree is `1` and the

maximum is 9. The running time complexity to find the minimum or maximum value in a binary search tree is $O(h)$, where h is the height of the tree.

Benefits of a binary search tree

A binary search tree is, in general, a better choice compared to arrays and linked lists when we are mostly interested in accessing the elements frequently in any application. A binary search tree is fast for most operations, such as searching, insertion, and deletion, whereas arrays provide fast searching, but are comparatively slow regarding insertion and deletion operations. In a similar fashion, linked lists are efficient in performing insertion and deletion operations, but are slower when performing the search operation. The best-case running time complexity for searching an element from a binary search tree is $O(\log n)$, and the worst-case time complexity is $O(n)$, whereas both best-case and worst-case time complexity for searching in lists is $O(n)$.

The following table provides a comparison of the array, linked list, and binary search tree data structures:

Properties	Array	Linked list	BST
Data structure	Linear.	Linear.	Non-linear.
Ease of use	Easy to create and use.	Insertion and deletion are	Access of elements,

	Average-case complexity for search, insert, and delete is $O(n)$.	fast, especially with the doubly linked list.	insertion, and deletion is fast with the average-case complexity of $O(\log n)$.
Access complexity	Easy to access elements. Complexity is $O(1)$.	Only sequential access is possible, so slow. Average- and worst-case complexity are $O(n)$.	Access is fast, but slow when the tree is unbalanced, with a worst-case complexity of $O(n)$.
Search complexity	Average- and worst-case complexity are $O(n)$.	It is slow due to sequential searching. Average- and worst-case complexity are $O(n)$.	Worst-case complexity for searching is $O(n)$.
Insertion complexity	Insertion is slow. Average- and worst-case complexity are $O(n)$.	Average- and worst-case complexity are $O(1)$.	The worst-case complexity for insertion is $O(n)$.

Deletion complexity	Deletion is slow. Average- and worst-case complexity are $O(n)$.	Average- and worst-case complexity are $O(1)$.	The worst-case complexity for deletion is $O(n)$.
---------------------	---	---	--

Let's consider an example to understand when the binary search tree is a good choice to store the data. Let's assume that we have the following data nodes—5, 3, 7, 1, 4, 6, and 9, as shown in *Figure 6.35*. If we use a list to store this data, the worst-case scenario will require us to search through the entire list of seven elements to find the item. So, it will require six comparisons to search for item 9 in this data node, as shown in *Figure 6.35*:

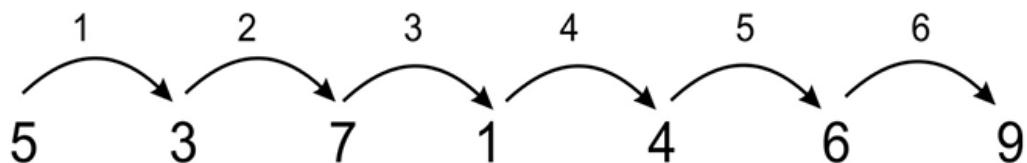


Figure 6.35: An example list of seven elements requires six comparisons if stored in a list

However, if we use a binary search tree to store these values, as shown in the following diagram, in the worst-case scenario, we will require two comparisons to search for item 9, as shown in *Figure 6.36*:

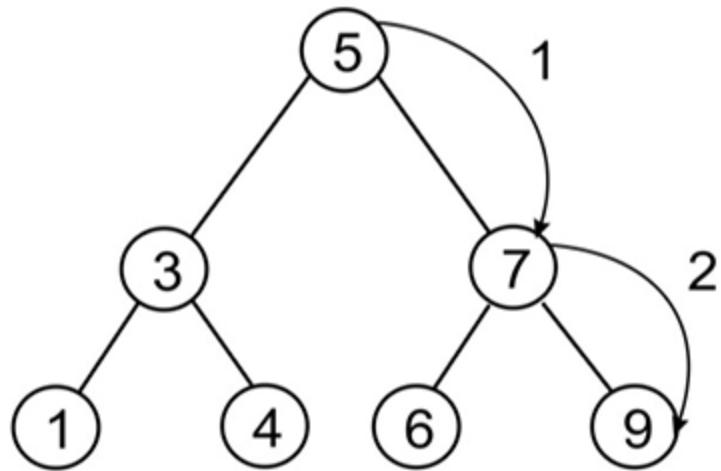


Figure 6.36: An example list of seven elements requires three comparisons if stored in a binary search tree

However, it is important to note that the efficiency of searching also depends on how we built the binary search tree. If the tree hasn't been constructed properly, it can be slow. For example, if we had inserted the elements into the tree in the order 1, 3, 4, 5, 6, 7, 9, as shown in *Figure 6.37*, then the tree would not be more efficient than the list:

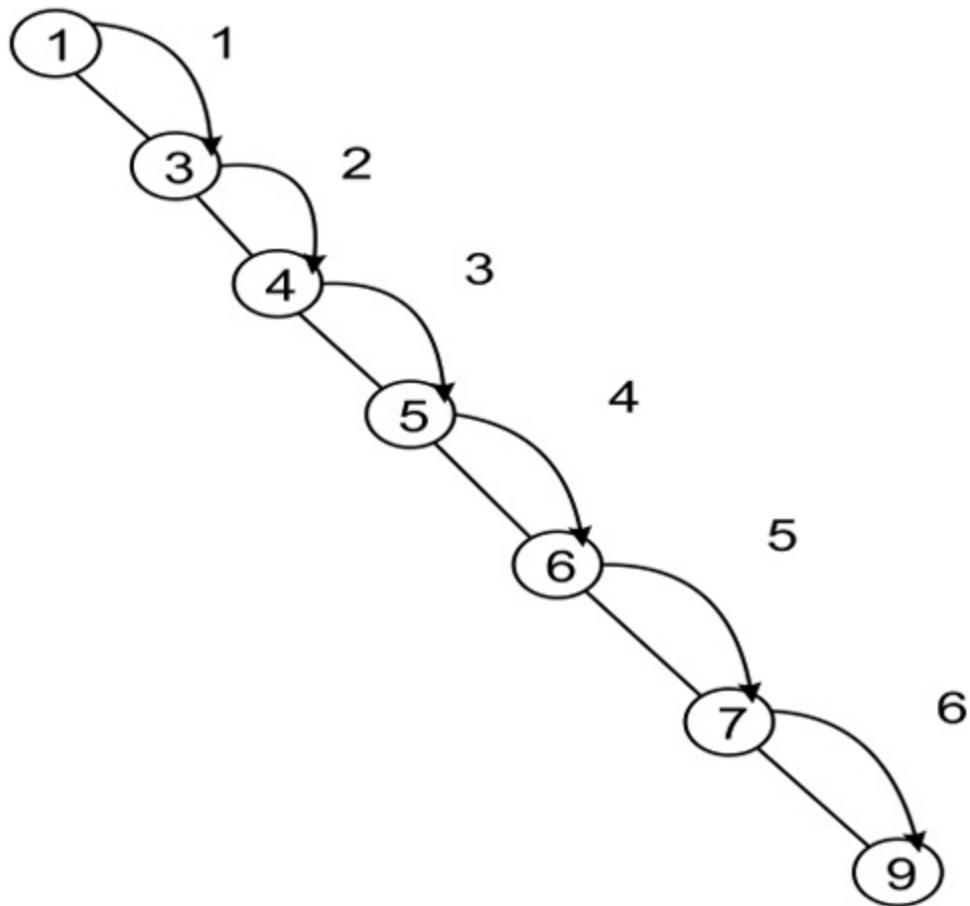


Figure 6.37: A binary search tree constructed with elements in the order 1, 3, 4, 5, 6, 7,9

Depending upon the sequence of the nodes added to the tree, it is possible that we may have a binary tree that is unbalanced. Thus, it is important to use a method that can make the tree a self-balancing tree, which in turn will improve the `search` operation. Therefore, we should note that a binary search tree is a good choice if the binary tree is balanced.

Summary

In this chapter, we discussed an important data structure, i.e. tree data structures. Tree data structures in general provide better performance compared to linear data structures in `search`, `insert`, and `deletion` operations. We have also discussed how to apply various operations to tree data structures. We studied binary trees, which can have a maximum of two children for each node. Further, we learned about binary search trees and discussed how we can apply different operations to them. Binary search trees are very useful when we want to develop a real-world application in which the retrieval or searching of data elements is an important operation. We need to ensure that the tree is balanced for the good performance of binary search tree. We will discuss priority queues and heaps in the next chapter.

Exercises

1. Which of the following is a true about binary trees:
 - a. Every binary tree is either complete or full
 - b. Every complete binary tree is also a full binary tree
 - c. Every full binary tree is also a complete binary tree
 - d. No binary tree is both complete and full
 - e. None of the above
2. Which of the tree traversal algorithms visit the root node last?

Consider this binary search tree:

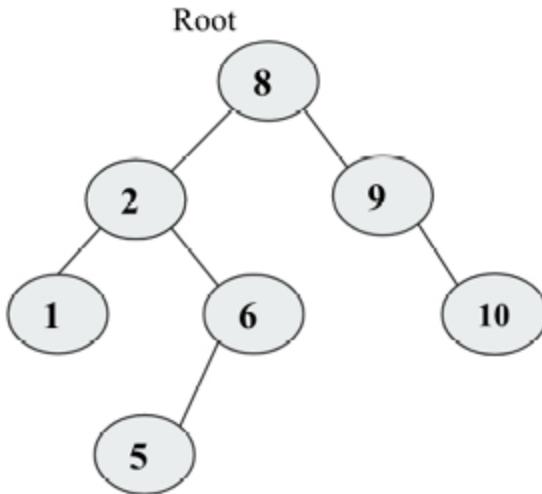


Figure 6.38: Sample binary search tree

3. Suppose we remove the root node 8, and we wish to replace it with any node from the left subtree, then what will be the new root?
4. What will be the `inorder`, `postorder` and `preorder` traversal of the following tree?

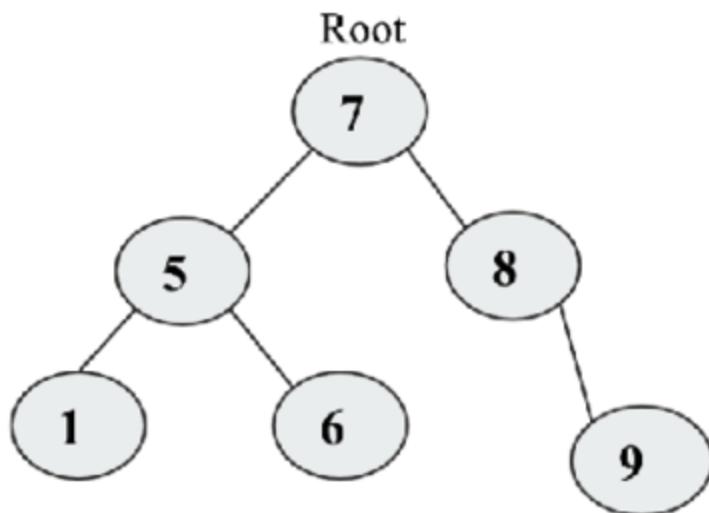


Figure 6.39: Example tree

5. How do you find out if two trees are identical?
6. How many leaves are there in the tree mentioned in *question number 4*?
7. What is the relation between a perfect binary tree's height and the number of nodes in that tree?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



7

Heaps and Priority Queues

A heap data structure is a tree-based data structure in which each node of the tree has a specific relationship with other nodes, and they are stored in a specific order. Depending upon the specific order of the nodes in the tree, heaps can be of different types, such as a `min` heap and a `max` heap.

A priority queue is an important data structure that is similar to the queue and stack data structures that stores data along with the priority associated with them. In this, the data is served according to the priority. Priority queues can be implemented using an array, linked list, and trees; however, they are often implemented using a heap as it is very efficient.

In this chapter, we will learn the following:

- The concept of the heap data structure and different operations on it
- Understanding the concept of the priority queue and its implementation using Python

Heaps

A heap data structure is a specialization of a tree in which the nodes are ordered in a specific way. A heap is a data structure where each data elements satisfies a `heap` property, and the `heap` property states that there must be a certain relationship between a parent node and its child nodes. According to this certain relationship in the tree, the heaps can be of two types, in other words, `max` heaps and `min` heaps. In a `max` heap, each parent node value must always be greater than or equal to all its children. In this kind of tree, the `root` node must be the greatest value in the tree. For example, see *Figure 7.1* showing the `max` heap in which all the nodes have greater values compared to their children:

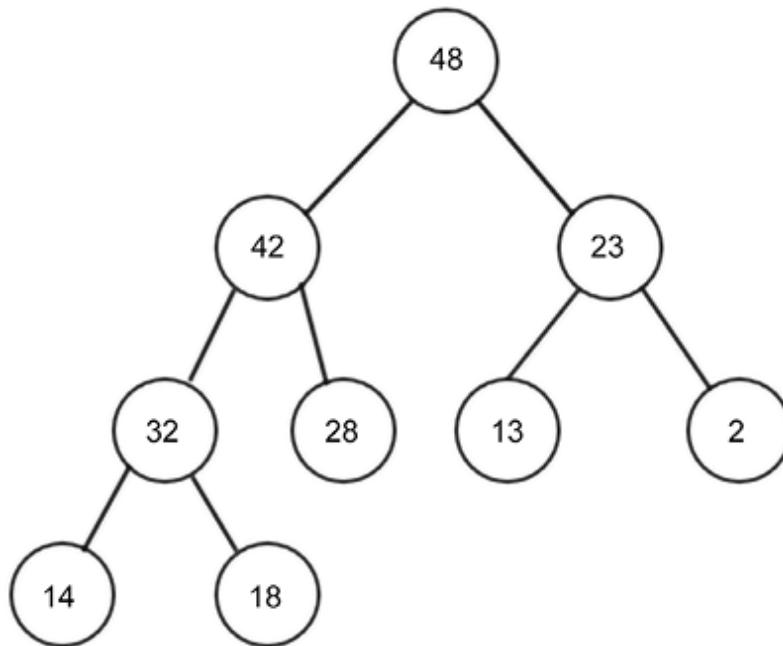


Figure 7.1: An example of a max heap

In a `min` heap, the relationship between parent and children is that the value of the parent node must always be less than or equal to its children. This rule should be followed by all the nodes in the tree. In the `min` heap, the `root` node holds the lowest value. For example, see

Figure 7.2 showing the `min` heap in which all the nodes have smaller values compared to their children:

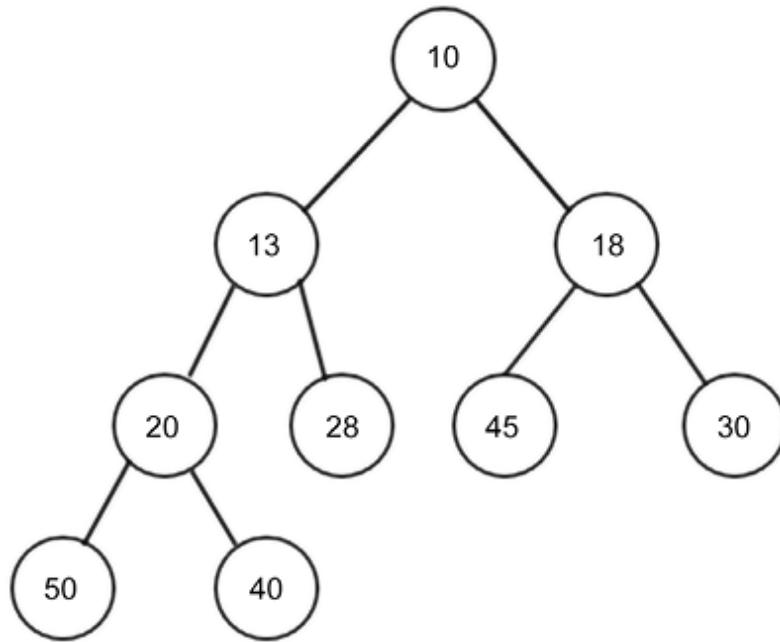


Figure 7.2: An example of a min heap

The heap is an important data structure due to its several applications and uses in implementing heap sort algorithms and priority queues. We will be discussing these in detail later in the chapter. The heap can be any kind of tree; however the most common type of heap is a binary heap in which each node has at most two children.

If the binary heap is a **complete binary tree** with n nodes, then it will have a minimum height of $\log_2 n$.

A complete binary tree is one in which each row must be fully filled before starting to fill the next row, as shown in the following *Figure 7.3*:

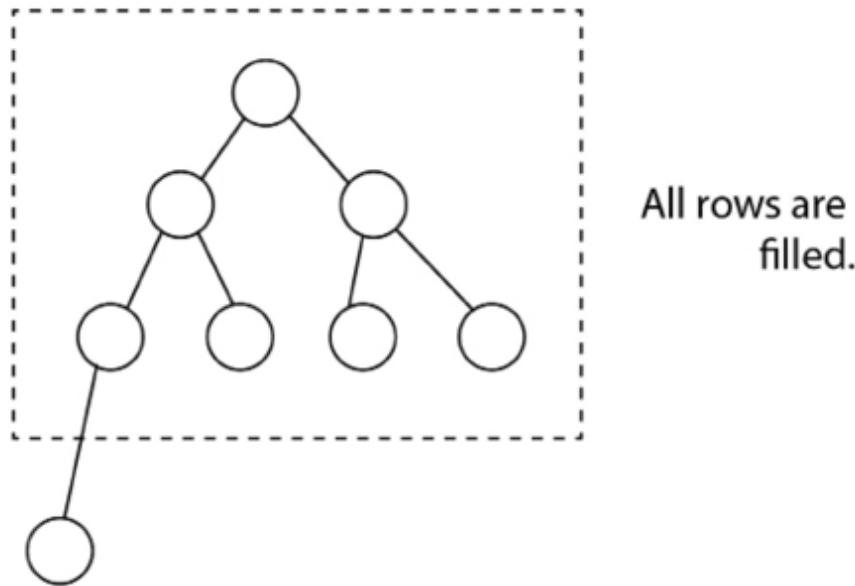


Figure 7.3: An example of a complete binary tree

In order to implement the heap, we can derive a relationship between parent and child nodes in `index` values. The relationship is that the children of any node at the `n` index can be retrieved easily, in other words, the left child will be located at `2n`, and the right child will be located at `2n + 1`. For example, the node `c` would be at the index of `3`, since node `c` is a right child of the node `a`, which is at index `1`, so it becomes $2n+1 = 2*1 + 1 = 3$. This relationship always holds true. Let's say we have a list of elements `{A, B, C, D, E}` as shown in *Figure 7.4*. If we store any element at an index of `i`, then its parent can be stored at index `i/2`, for example, if the index of the node `D` is `4`, then its parent would be at $4/2 = 2$, index `2`. The index of root has to be starting from `1` in the array. See *Figure 7.4* to understand the concept:

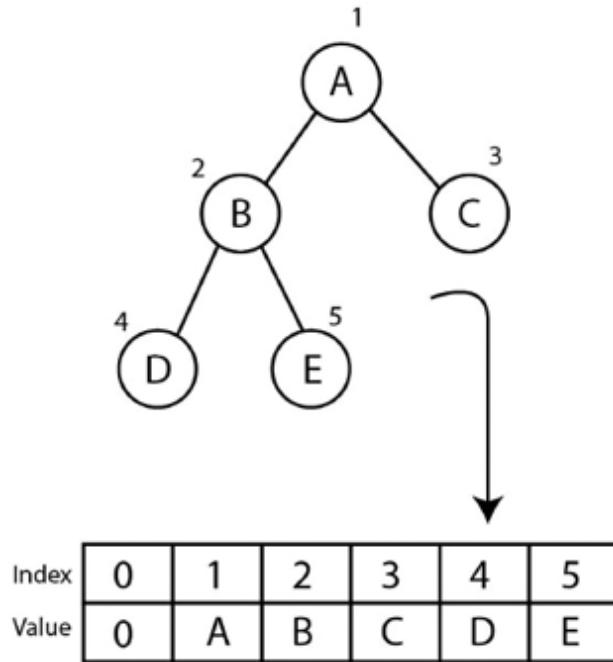


Figure 7.4: Binary tree and index positions of all the nodes

This relation between parent and child is a complete binary tree. In respect of indexing values, it is very important in order to efficiently retrieve, search, and store the data elements in the heap. Due to this property, it is very easy to implement the heap. The only constraint is that we should have indexing starting from 1, and if we implement the heap using an array, then we have to add one dummy element at index 0 in the array. Next, let's understand the implementation of the heap. It is important to note that we will be discussing all the concepts with respect to the `min` heap, and the implementation for the `max` heap will be very similar to it, with the only difference being the `heap` property.

Let's discuss the implementation of the `min` heap using Python. We start with the `heap` class, as follows:

```
class MinHeap:  
    def __init__(self):  
        self.heap = [0]  
        self.size = 0
```

We initialize the heap list with a zero to represent the dummy first element, and we are adding a dummy element just to start the indexing of data items from `1` since if we start indexing from `1`, accessing of the elements becomes very easy due to the parent-child relationship. We also create a variable to hold the size of the heap. We will further discuss different operations, such as `insert`, `delete`, and `delete` at a specific location in the heap. Let's start with the insertion operation in the heap.

Insert operation

The insertion of an item into a `min` heap works in two steps. First, we add the new element to the end of the list (which we understand to be the bottom of the tree), and we increment the size of the heap by one. Secondly, after each insertion operation, we need to arrange the new element up in the heap tree, to organize all the nodes in such a way that satisfies the `heap` property, which in this case is that each node must be larger than its parent. In other words, the value of the parent node must always be less than or equal to its children, and the lowest element in the `min-heap` needs to be the root element.

Therefore, we first insert an element into the last heap of the tree; however, after inserting an element into the heap, it is possible that the `heap` property is violated. In that case, the nodes have to be rearranged so that all the nodes satisfy the `heap` property. This process is called heapifying. To heapify the `min` heap, we need to find

the minimum of its children and swap it with the current element, and this process has to be repeated until the `heap` property is satisfied for all the nodes.

Let's consider an example of adding an element in the `min` heap, such as inserting a new node with a value of `2` in *Figure 7.5*:

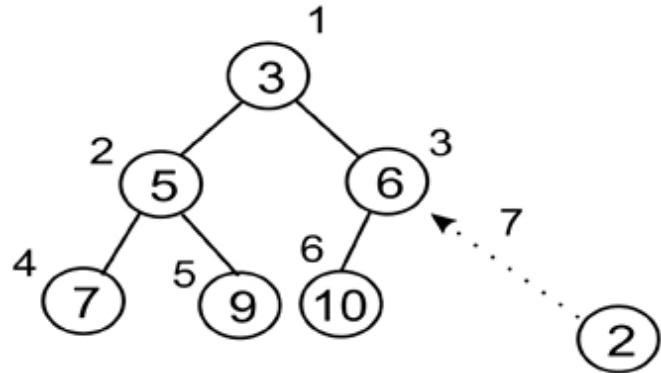


Figure 7.5: Insertion of a new node 2 in the existing heap

The new element will be added to the last position in the third row or level. Its index value is `7`. We compare that value with its parent. The parent is at index $7/2 = 3$ (integer division). The parent node holds value `6`, which is higher than the new node value (in other words, `2`), so according to the property of the `min` heap, we swap these values, as shown in *Figure 7.6*:

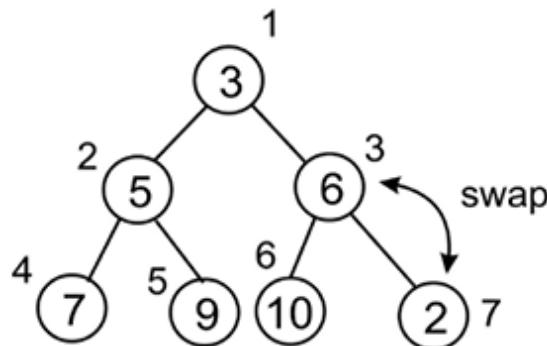


Figure 7.6: Swapping nodes 2 and 6 to maintain the heap property

The new data element has been swapped and moved up to index 3. Since, we have to check all the nodes up to the root, we check the index of its parent node which is $3/2 = 1$ (*integer division*), so we continue the process to heapify.

So, we compare both of these elements, and swap again, as shown in *Figure 7.7*:

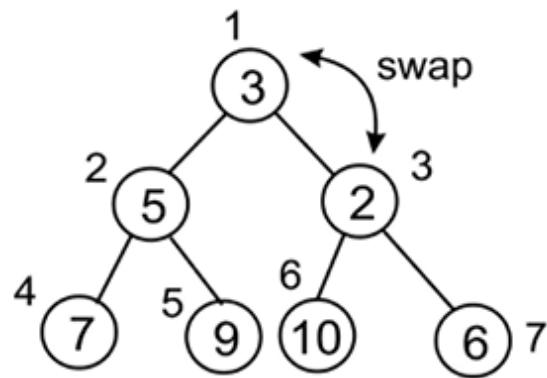


Figure 7.7: Swapping nodes 2 and 3 to maintain the heap property

After the final swap, we reach the root. Here, we can notice that this heap adheres to the definition of the `min` heap, as shown in *Figure 7.8*:

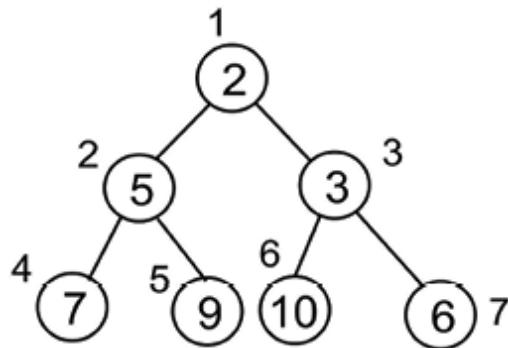


Figure 7.8: Final heap after insertion of a new node 2

Now, let's take another example to see how to create and insert elements in a heap. We start with the construction of a heap by

inserting 10 elements, one by one. The elements are {4, 8, 7, 2, 9, 10, 5, 1, 3, 6}. We can see a step-by-step process to insert elements into the heap in *Figure 7.9*:

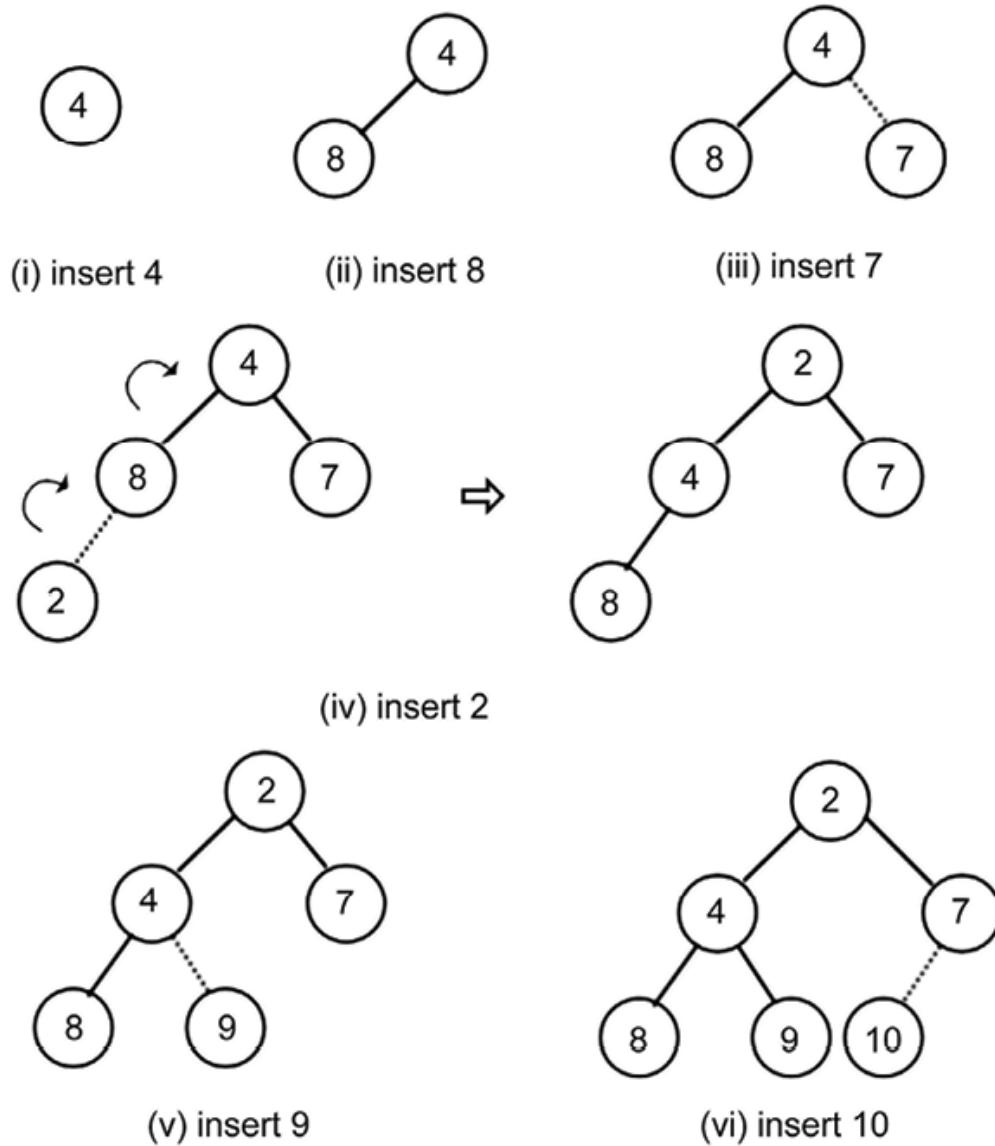


Figure 7.9: The step-by-step procedure to create a heap

We can see, in the preceding diagram, a step-by-step process to insert elements into the heap. Here, we continue adding elements, as shown in *Figure 7.10*:

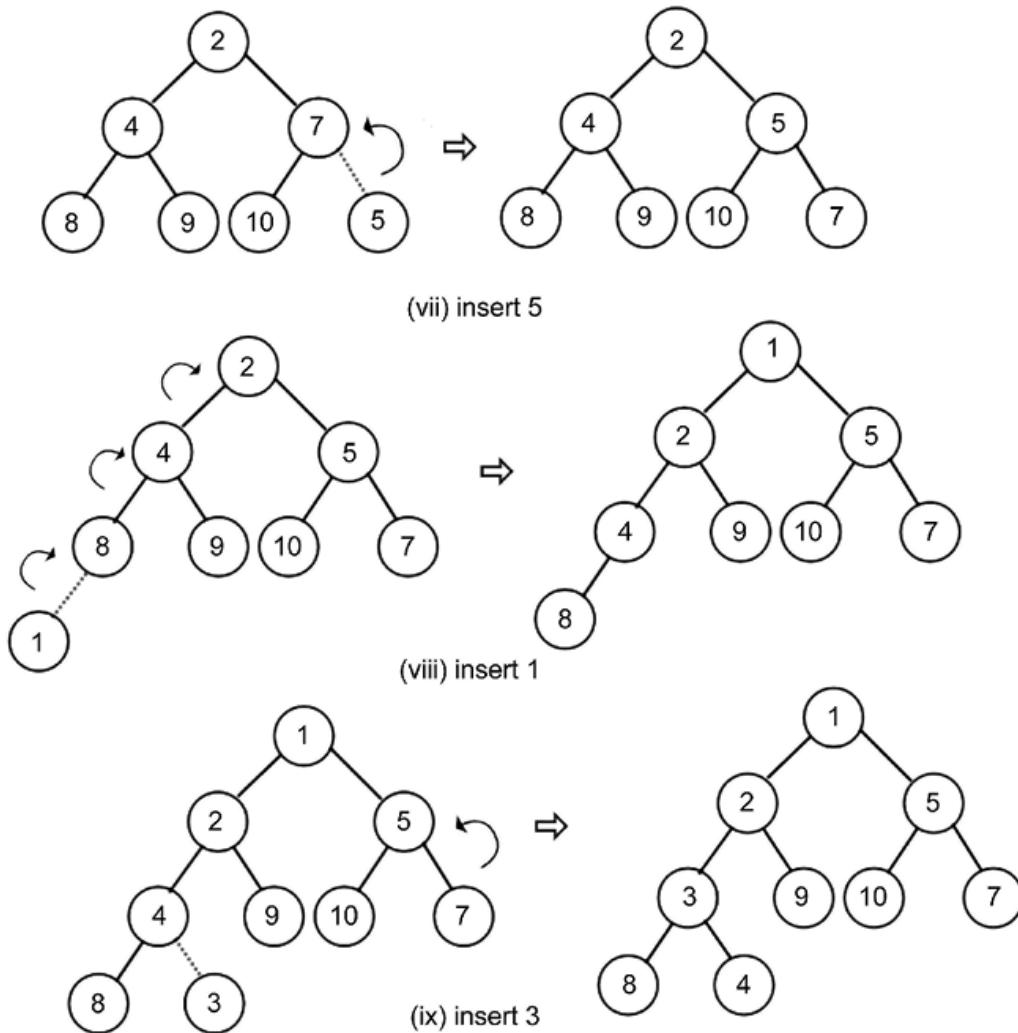


Figure 7.10: Steps 7 to 9 in creating the heap

Finally, we insert an element, 6, into the heap, as shown in Figure 7.11:

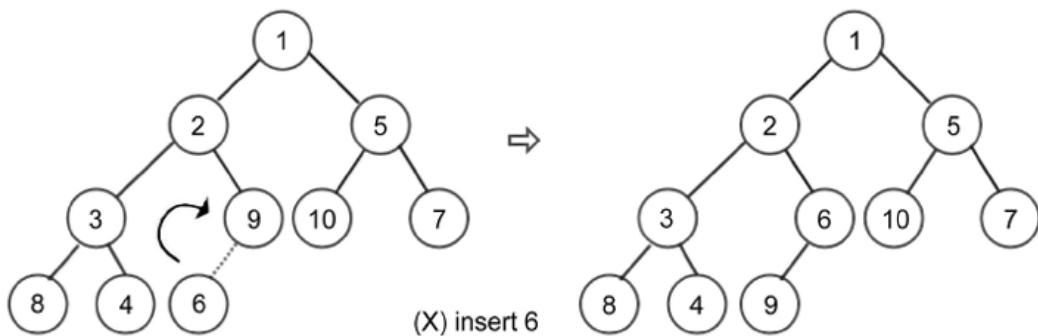


Figure 7.11: Last step and construction of the final heap

The implementation of the insertion operation in the heap is discussed as follows. Firstly, we create a helper method, called the `arrange`, that takes care of arrangements of all the nodes after insertion of a new node. Here is the implementation of the `arrange()` method, which should be defined in the `MinHeap` class:

```
def arrange(self, k):
    while k // 2 > 0:
        if self.heap[k] < self.heap[k//2]:
            self.heap[k], self.heap[k//2] = self.heap[k//2], self.
        k ///= 2
```

We execute the loop until we reach up to the `root` node; until then, we can keep arranging the element. Here, we are using integer division. The loop will break out after the following condition:

```
while k // 2 > 0:
```

After that, we compare the values between the parent and child node. If the parent is greater than the child, swap the two values:

```
if self.heap[k] < self.heap[k//2]:
    self.heap[k], self.heap[k//2] = self.heap[k//2], self.heap
```

Finally, after each iteration, we move up in the tree:

```
k ///= 2
```

This method ensures that the elements are ordered properly.

Now, for adding new elements in the heap, we need to use the following `insert` method, which should be defined in the `MinHeap` class:

```
def insert(self, item):
    self.heap.append(item)
    self.size += 1
    self.arrange(self.size)
```

In the above code, we can insert an element using the `append` method; then we increase the size of the heap. Then, in the last line of the `insert` method, we call the `arrange()` method to reorganize the heap (heapify it) to ensure that all the nodes in the heap satisfy the `heap` property.

Now, let's create the heap and insert that data `{4, 8, 7, 2, 9, 10, 5, 1, 3, 6}` using the `insert()` method, which is defined in the `MinHeap` class, as shown in the following code:

```
h = MinHeap()
for i in (4, 8, 7, 2, 9, 10, 5, 1, 3, 6):
    h.insert(i)
```

We can print the heap list, just to inspect how the elements are ordered. If you redraw this as a tree structure, you will notice that it meets the required properties of a heap, similar to what we created manually:

```
print(h.heap)
```

The output of the above code is as follows:

```
[0, 1, 2, 5, 3, 6, 10, 7, 8, 4, 9]
```

We can see in the output that all the data items of the heap in the array are as in the index position as per *Figure 7.11*. Next, we will discuss the delete operation in the heap.

Delete operation

The `delete` operation removes an element from the heap. To delete any element from the heap, let's first discuss how we can delete the root element since it is mostly used for several use cases, such as finding the minimum or maximum element in a heap. Remember, in a `min-heap`, the root element denotes the minimum value of the list, and the root of the `max-heap` gives the maximum value of the list of elements.

Once we delete the root element from the heap, we make the last element of the heap the new root of the heap. In that case, the `heap` property will not be satisfied by the tree. So, we have to reorganize the nodes of the tree such that all the nodes of the tree satisfy the `heap` property. The delete operation in `min-heap` works as follows.

1. Once we delete the `root` node, we need a new `root` node. For this, we take the last item from the list and make it the new root.
2. Since the selected last node might not be the lowest element of the heap, we have to reorganize the nodes of the heap.
3. We reorganize the nodes from the `root` node to the last node (which is made into a new root); this process is called `heapify`. Since we move from top to bottom (which means from the `root`

node down to the last element) of the heap, this process is called percolate down.

Let's consider an example to help us understand this concept in the following heap. First, we delete the `root` node that has value `2`, as shown in *Figure 7.12*:

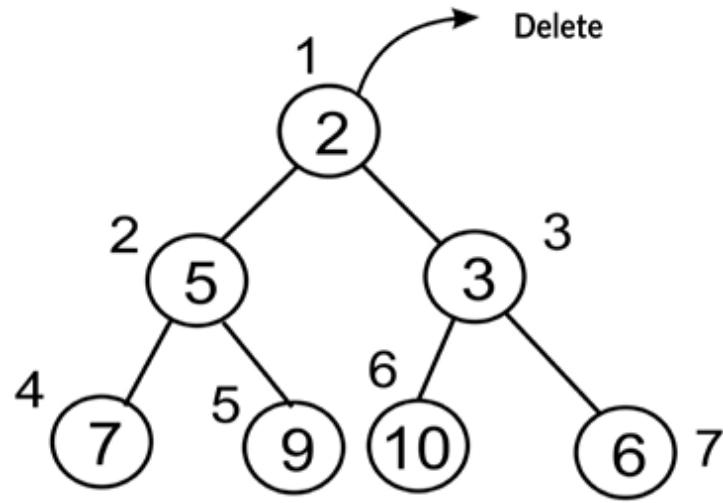


Figure 7.12: Deletion of a node with value 2 at the root in the existing heap

Once we delete the root, next we need to choose a node that can be the new root; commonly, we choose to take the last node, in other words, node `6` at index `7`. So, the last element, `6`, is placed at the root position, as shown in *Figure 7.13*:

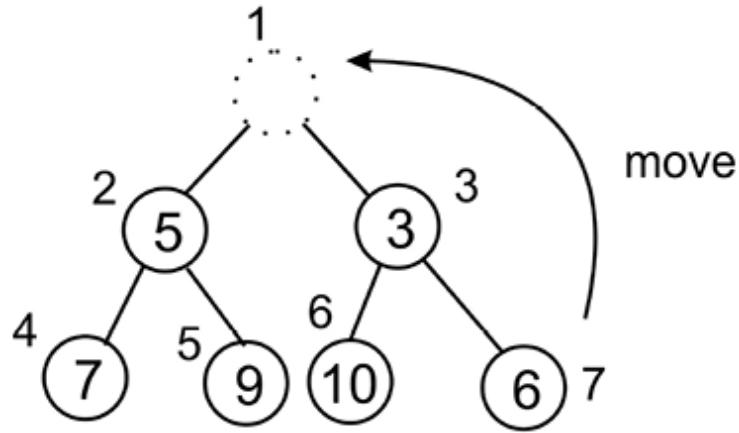


Figure 7.13: Moving the last element, in other words, node 6 to the root position

After moving the last element to the new root, clearly this tree is now not satisfying the `min-heap` property. So, we have to reorganize the nodes of the heap, hence we move down from the root to the nodes in the heap, that is, heapify the tree. So, we compare the value of the newly replaced node with all its children nodes in the tree. In this example, we compare the two children of the root, that is, `5` and `3`. Since the right child is smaller, its index is `3`, which is represented as $(\text{root index} * 2 + 1)$. We will go ahead with this node and compare the new `root` node with the value at this index, as shown in *Figure 7.14*:

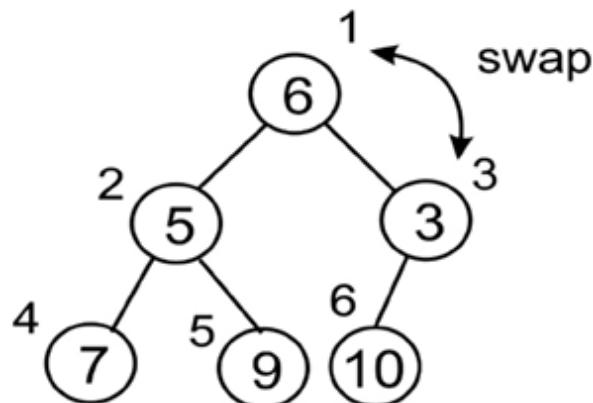


Figure 7.14: Swapping of the root node with the node 3

Now, the node with value 6 should be moved down to index 3 as per the `min heap` property. Next, we need to compare it to its children down to the heap. Here, we only have one child, so we don't need to worry about which child to compare it against (for a `min` heap, it is always the lesser child), as shown in *Figure 7.15*:

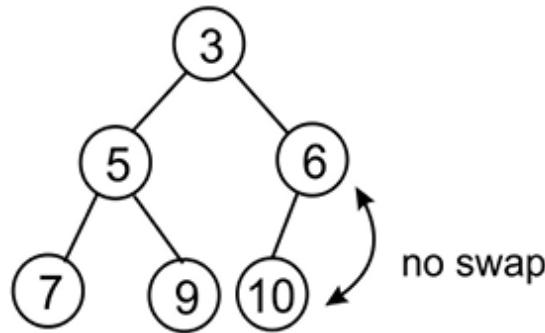


Figure 7.15: Swapping of node 6 and node 10

There is no need to swap here since it is following the `min-heap` property. After reaching the last one, the final heap adheres to the `min-heap` property.

In order to implement the deletion of the `root` node from the heap using Python, firstly, we implement the percolate-down process, in other words, the `sink()` method. Before we implement the `sink()` method, we implement a `helper` method for finding out which of the children to compare against the parent node. This `helper` method is `minchild()`, which should be defined in the `MinHeap` class:

```
def minchild(self, k):
    if k * 2 + 1 > self.size:
        return k * 2
    elif self.heap[k*2] < self.heap[k*2+1]:
        return k * 2
    else:
        return k * 2 + 1
```

In this method, firstly, we check if we get beyond the end of the list— if we do, then we return the index of the left child:

```
if k * 2 + 1 > self.size:  
    return k * 2
```

Otherwise, we simply return the index of the lesser of the two children:

```
elif self.heap[k*2] < self.heap[k*2+1]:  
    return k * 2  
else:  
    return k * 2 + 1
```

Now we can create the `sink()` method. The `sink()` method should be defined in the `MinHeap` class:

```
def sink(self, k):  
    while k * 2 <= self.size:  
        mc = self.minchild(k)  
        if self.heap[k] > self.heap[mc]:  
            self.heap[k], self.heap[mc] = self.heap[mc], self.heap  
        k = mc
```

In the above code, we first run the loop until the end of the tree so that we can sink (move down) our element down as far as is needed; this is shown in the following code snippet:

```
def sink(self, k):  
    while k*2 <= self.size:
```

Next, we need to know which of the left or right children to compare against. This is where we make use of the `minindex()` function, as shown in the following code snippet:

```
mi = self.minchild(k)
```

Next, we compare parent and child to see whether we need to make the swap, as we did in the `arrange()` method during the insertion operation:

```
if self.heap[k] > self.heap[mc]:  
    self.heap[k], self.heap[mc] = self.heap[mc], self.heap
```

Finally, we need to make sure that we move down the tree in each iteration so that we don't get stuck in a loop, as follows:

```
k = mc
```

Now, we can implement the main `delete_at_root()` method itself, which should be defined in the `MinHeap` class:

```
def delete_at_root(self):  
    item = self.heap[1]  
    self.heap[1] = self.heap[self.size]  
    self.size -= 1  
    self.heap.pop()  
    self.sink(1)  
    return item
```

In the above code for deletion of the `root` node, we first copy the root element in a variable `item`, and then the last element is moved to the

`root` node in the following statement:

```
self.heap[1] = self.heap[self.size]
```

Further, we reduce the size of the heap, and remove the element from the heap, and then we use the `sink()` method to reorganize the heap element so that all the elements of the heap follow the `heap` property.

We can now use the following code to delete the `root` node from the heap. Let's first insert some data items `{2, 3, 5, 7, 9, 10, 6}` in the heap and then remove the `root` node:

```
h = MinHeap()
for i in (2, 3, 5, 7, 9, 10, 6):
    h.insert(i)
print(h.heap)
n = h.delete_at_root()
print(n)
print(h.heap)
```

The output of the above code is as follows:

```
[0, 2, 3, 5, 7, 9, 10, 6]
2
[0, 3, 6, 5, 7, 9, 10]
```

We can see in the output that the root element 2 is returned in the new heap, and that the data elements are rearranged so that all the nodes of the heap are following the `heap` property (indexes of the nodes can checked as shown in *Figure 7.16*). Next, we will discuss if we want to delete any node with the given index position.

Deleting an element at a specific location from a heap

Generally, we delete an element at the root, however, an element can be deleted at a specific location from the heap. Let us understand it with an example. Given the following heap, let's assume that we want to delete a node with value 3 at index 2. After deleting the node with value 3, we move the last node to the deleted node, in other words, the node with value 15, as shown in *Figure 7.16*:

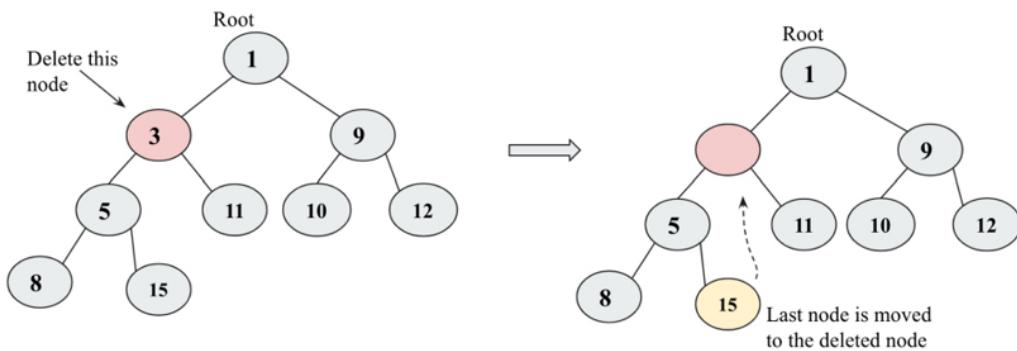


Figure 7.16: The deletion of node 3 from the heap

After shifting the last element to the deleted node, we compare this with its root element since it is already greater than the root element, so we do not swap. Next, we compare this element with all of its children, and since the left child is smaller, it is swapped with the left child, as shown in *Figure 7.17*:

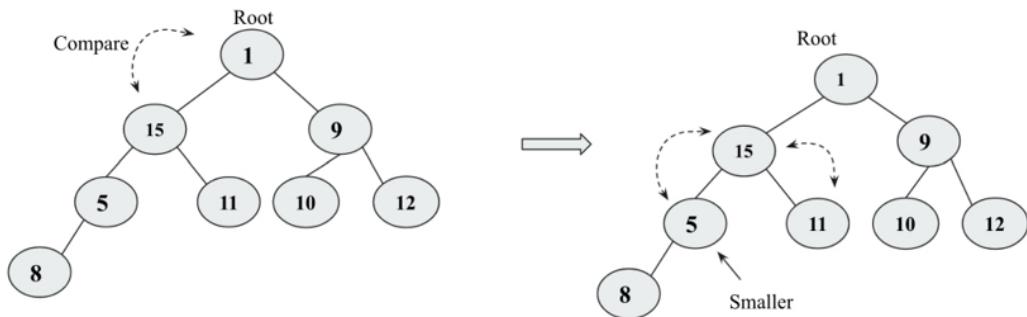


Figure 7.17: A comparison of node 15 with 5 and 11, and swapping node 15 and node 5

After swapping node 15 with node 5, we move down in the heap.

Next, we compare node 15 with its child, node 8. Finally, node 8 and node 15 are swapped. Now, the final tree follows the `heap` property, as shown in *Figure 7.18*:

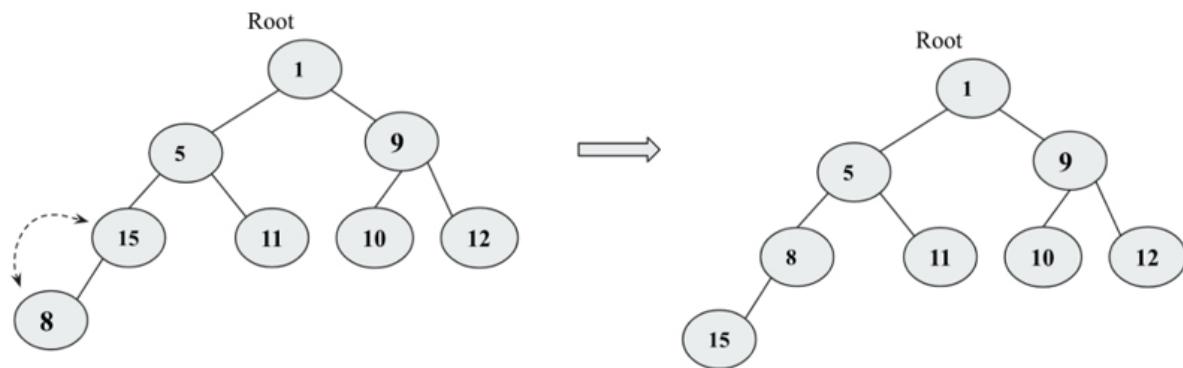


Figure 7.18: The final heap after swapping node 8 and node 15

The implementation of the delete operation for removing a data item at any given index location is given below, which should be defined in the `MinHeap` class:

```

def delete_at_location(self, location):
    item = self.heap[location]
    self.heap[location] = self.heap[self.size]
    self.size -= 1
    self.heap.pop()

```

```
    self.sink(location)
    return item
```

This implementation is very similar to what we have seen in the previous section for deleting the root element. The only difference is that in this code, we have specified the index location that has to be deleted. The following code snippet demonstrates the deletion of a node at a specific location 2 from the heap created from data elements

{4, 8, 7, 2, 9, 10, 5, 1, 3, 6}:

```
h = MinHeap()
for i in (4, 8, 7, 2, 9, 10, 5, 1, 3, 6):
    h.insert(i)
print(h.heap)

n = h.delete_at_location(2)
print(n)
print(h.heap)
```

The output of the preceding code is as follows:

```
[0, 1, 2, 5, 3, 6, 10, 7, 8, 4, 9]
2
[0, 1, 3, 5, 4, 6, 10, 7, 8, 9]
```

In the above output, we see that, before and after, the heap nodes are placed according to their index positions. We have discussed the concepts and implementation using examples of `min-heap`; all these operations and concepts can be easily implemented for a `max-heap` by simply reversing the logic in conditions where we ensured that the parent node should have smaller values compared to the children in `min-heap`. Now in the case of `max-heap`, we have to make the larger

value in the parent. Heaps are used in various applications such as to implement heap sort and priority queues, which we will discuss in subsequent sections.

Heap sort

Heap is an important data structure for sorting a list of elements since it is very suitable for a large number of elements. If we want to sort a list of elements, say in ascending order, we can use `min-heap` for this purpose; we first create a `min-heap` of all the given data elements, and as per the `heap` property, the smallest data value will be stored at the root of the heap. With the help of the `heap` property, it is straightforward to sort the elements. The process is as follows:

1. Create a `min-heap` using all the given data elements.
2. Read and delete the root element, which is the minimum value.
After that, copy the last element of the tree to the new root, and further reorganize the tree to maintain the `heap` property.
3. Now, we repeat step 2 until we get all the elements.
4. Finally, we get the sorted list of elements.

The data elements are stored in the heap adhering to the `heap` property; whenever a new element is added or deleted, the `heap` property is maintained using the `arrange()` and `sink()` helper methods, respectively, as discussed in previous sections.

In order to implement heap sort using the heap data structure, first we create a heap with the data items `{4, 8, 7, 2, 9, 10, 5, 1, 3, 6}` using the below code (details of the creation of the heap are given in previous sections):

```
h = MinHeap()
unsorted_list = [4, 8, 7, 2, 9, 10, 5, 1, 3, 6]
for i in unsorted_list:
    h.insert(i)
print("Unsorted list: {}".format(unsorted_list))
```

In the above code, the `min-heap`, `h`, is created and the elements in `unsorted_list` are inserted. After each call to the `insert()` method, the heap order property is restored by the subsequent call to the `sink` method.

After creation of the heap, next, we read and delete the root element. In each iteration, we get the minimum value, and thus the data items in ascending order. The implementation of the `heap_sort()` method should be defined in the `minHeap` class (it uses the `delete_at_root()` method discussed in previous sections):

```
def heap_sort(self):
    sorted_list = []
    for node in range(self.size):
        n = self.delete_at_root()
        sorted_list.append(n)
    return sorted_list
```

In the above code, we create an empty array, `sorted_list`, which stores all the data elements in sorted order. Then we run the loop for the number of items in the list. In each iteration, we call the `delete_at_root()` method to get the minimum value, which is appended to `sorted_list`.

Now we can use the heap sort algorithm using the following code:

```
print("Unsorted list: {}".format(unsorted_list))
print("Sorted list: {}".format(h.heap_sort()))
```

The output of the above code is as follows:

```
Unsorted list: [4, 8, 7, 2, 9, 10, 5, 1, 3, 6]
Sorted list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The time complexity to build the heap using the insert method takes $O(n)$ times. Further, to reorganize the tree after deleting the root element takes $O(\log n)$ since we go from top to bottom in the heap tree, and the height of the heap is $\log_2(n)$, hence the complexity of rearranging the tree is $O(\log n)$. So, overall, the worst-case time complexity of the heap sort is $O(n \log n)$. Heapsort is very efficient in general, giving a worst-case, average-case and best-case complexity of $O(n \log n)$.

Priority queues

A priority queue is a data structure that is similar to a queue in which data is retrieved based on the **First In, First Out (FIFO)** policy, but in the priority queue, priority is attached with the data. In the priority queue, the data is retrieved based on the priority associated with the data elements, the data elements with the highest priority are retrieved before the lower priority data elements, and if two data elements have the same priority, they are retrieved according to the **FIFO** policy.

We can assign the priority of the data depending upon the application. It is used in many applications, such as CPU scheduling,

and many algorithms also rely on priority queues, such as Dijkstra's shortest-path, A* search, and Huffman codes for data compression.

So, in the priority queue, the item with the highest priority is served first. The priority queue stores the data according to the priority associated with the data, so insertion of an element will be at a specific position in the priority queue. Priority queues can be considered as modified queues that return the items in the order of highest priority instead of returning the items in the **FIFO** order. A priority queue can be implemented by modifying an enqueue position by inserting the item according to the priority. It is demonstrated in *Figure 7.19*, in which given the queue, a new item **5** is added to the queue at a specific index (here assuming that the data items having higher values have higher priority):

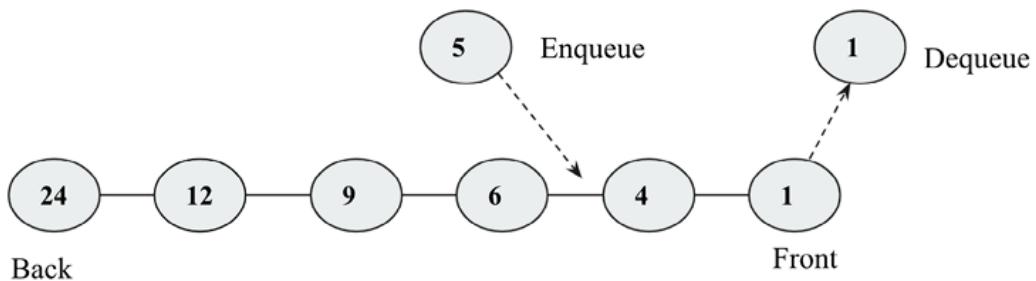


Figure 7.19: A demonstration of a priority queue

Let's understand the priority queue with an example. When we receive data elements in an order, the elements are enqueued in the priority queue in the order of priority (assuming that the higher data value is of higher importance). Firstly, the priority queue is empty, so **3** is added initially in the queue; the next data element is **8**, which will be enqueued at the start since it is greater than **3**. Next, the data item is **2**, then **6**, and finally, **10**, which are enqueued in the priority queue as per their priority, and when the dequeue operation is

applied, the high priority item will be dequeued first. All the steps are represented in *Figure 7.20*:

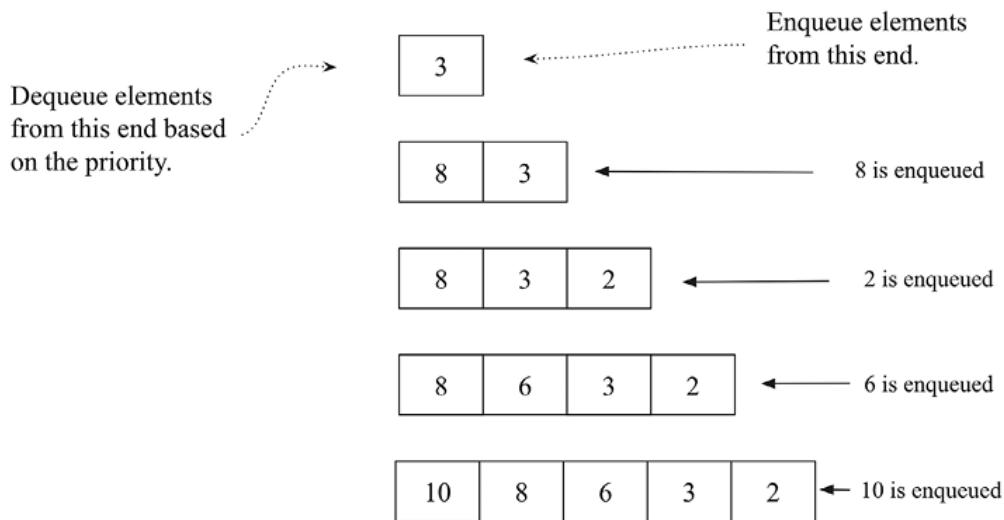


Figure 7.20: A step-by-step procedure to create a priority queue

Let us discuss the implementation of a priority queue in Python. We first define the node class. A node class will have the data elements along with the priority associated with the data in the priority queue:

```
# class for Node with data and priority
class Node:
    def __init__(self, info, priority):
        self.info = info
        self.priority = priority
```

Next, we define the `PriorityQueue` class and initialize the queue:

```
# class for Priority queue
class PriorityQueue:
    def __init__(self):
        self.queue = []
```

Next, let us discuss the implementation of the insertion operation for adding a new data element to the priority queue. In the implementation, we assume that the data element has high priority if it has a smaller priority value (for example, a data element with the priority value `1` has higher priority compared to the data element that has a priority value `4`). The following are cases of insertion of elements in a priority queue:

1. Insertion of a data element to the priority queue when the queue is initially empty.
2. If the queue is not empty, we perform the traversal of the queue and reach the appropriate index position in the queue according to the associated priorities by comparing the priorities of the existing node with the new node. We add the new node before the node that has a priority greater than the new node.
3. If the new node has a lower priority than the high priority value, then the node will be added to the start of the queue.

The implementation of the `insert()` method is as follows, which should be defined in the `PriorityQueue` class:

```
def insert(self, node):
    if len(self.queue) == 0:
        # add the new node
        self.queue.append(node)
    else:
        # traverse the queue to find the right place for new node
        for x in range(0, len(self.queue)):
            # if the priority of new node is greater
            if node.priority >= self.queue[x].priority:
                # if we have traversed the complete queue
                if x == (len(self.queue)-1):
                    # add new node at the end
                    self.queue.insert(x+1, node)
```

```
        else:
            continue
    else:
        self.queue.insert(x, node)
    return True
```

In the above code, we first append a new data element when the queue is empty, and then we iteratively reach the appropriate position by comparing the priorities associated with the data elements.

Next, when we apply the delete operation in the priority queue, the highest priority data element is returned and removed from the queue. It should be defined in the `PriorityQueue` class as follows:

```
def delete(self):
    # remove the first node from the queue
    x = self.queue.pop(0)
    print("Deleted data with the given priority-", x.info, x.priority)
    return x
```

In the preceding code, we get the top element with the highest priority value. Further, the implementation of the `show()` method that prints all the data elements of the priority queue in the order of the priorities should be defined in the `PriorityQueue` class:

```
def show(self):
    for x in self.queue:
        print(str(x.info) + " - " + str(x.priority))
```

Now, let's consider an example to see how to use the priority queue in which we firstly add data elements ("Cat", "Bat", "Rat", "Ant", and "Lion") with associated priorities 13, 2, 1, 26, and 25, respectively:

```
p = PriorityQueue()
p.insert(Node("Cat", 13))
p.insert(Node("Bat", 2))
p.insert(Node("Rat", 1))
p.insert(Node("Ant", 26))
p.insert(Node("Lion", 25))
p.show()
p.delete()
```

The output of the above code is as follows:

```
Rat - 1
Bat - 2
Cat - 13
Lion - 25
Ant - 26
Deleted data with the given priority- Rat 1
```

Priority queues can be implemented using several data structures; in the above example, we saw its implementation using a list of tuples where the tuple contains the priority as the first element and the value data item as the next element. However, the priority queues are mostly implemented using a heap, since it is efficient with the worst-case time complexity of $O(\log n)$ in insertion and deletion operations.

The implementation of the priority queue using heap is very similar to what we have discussed in the `min-heap` implementation. The only difference is that now we store the priorities associated with the data elements, and we create a `min-heap` tree considering the priority values

using a list of tuples in Python. For completeness, the code for the priority queue using heaps is as follows:

```
class PriorityQueueHeap:
    def __init__(self):
        self.heap = []
        self.size = 0

    def arrange(self, k):
        while k // 2 > 0:
            if self.heap[k][0] < self.heap[k//2][0]:
                self.heap[k], self.heap[k//2] = self.heap[k//2], self.heap[k]
            k //= 2

    def insert(self, priority, item):
        self.heap.append((priority, item))
        self.size += 1
        self.arrange(self.size)

    def sink(self, k):
        while k * 2 <= self.size:
            mc = self.minchild(k)
            if self.heap[k][0] > self.heap[mc][0]:
                self.heap[k], self.heap[mc] = self.heap[mc], self.heap[k]
            k = mc

    def minchild(self, k):
        if k * 2 + 1 > self.size:
            return k * 2
        elif self.heap[k*2][0] < self.heap[k*2+1][0]:
            return k * 2
        else:
            return k * 2 + 1

    def delete_at_root(self):
        item = self.heap[1][1]
        self.heap[1] = self.heap[self.size]
        self.size -= 1
        self.heap.pop()
        self.sink(1)
        return item
```

We use the code below to create a priority queue with data elements "Bat", "Cat", "Rat", "Ant", "Lion", and "Bear" with the associated priority values 2, 13, 18, 26, 3, and 4, respectively:

```
h = PriorityQueueHeap()
h.insert(2, "Bat")
h.insert(13, "Cat")
h.insert(18, "Rat")
h.insert(26, "Ant")
h.insert(3, "Lion")
h.insert(4, "Bear")
h.heap
```

The output of the above code is as follows:

```
[(), (2, 'Bat'), (3, 'Lion'), (4, 'Bear'), (26, 'Ant'), (13, 'Cat')]
```

In the above output, we can see that it shows a `min-heap` tree that adheres to the `min-heap` property. Now we can use the code below to remove the data elements:

```
for i in range(h.size):
    n = h.delete_at_root()
    print(n)
    print(h.heap)
```

The output of the preceding code is as follows:

```
'Bat
[(), (3, 'Lion'), (13, 'Cat'), (4, 'Bear'), (26, 'Ant'), (18, 'Rat')]
Lion
[(), (4, 'Bear'), (13, 'Cat'), (18, 'Rat'), (26, 'Ant')]
```

```
Bear
[(), (13, 'Cat'), (26, 'Ant'), (18, 'Rat')]
Cat
[(), (18, 'Rat'), (26, 'Ant')]
Rat
[(), (26, 'Ant')]
Ant
[()]
```

In the above output, we can see that the data items are produced according to the priorities associated with the data elements.

Summary

In this chapter, we have discussed an important data structure, in other words, the heap data structure. We also discussed heap properties for `min-heap` and `max-heap`. We have seen the implementation of several operations that can be applied to the heap data structure, such as heapifying, and the insertion and deletion of a data element from the heap. We have also discussed two of the important applications of the heap—heap sort and a priority queue. The heap is an important data structure since it has many applications, such as sorting, selecting minimum and maximum values in a list, graph algorithms, and priority queues. Moreover, the heap can also be useful when we have to repeatedly remove a data object with the highest or lowest priority values.

In the next chapter, we will discuss the concepts of **Hashing** and **Symbol Tables**.

Exercises

- What will be the time complexity for deleting an arbitrary element from the `min-heap`?
- What will be the time complexity for finding the k^{th} smallest element from the `min-heap`?
- What will be the worst-case time complexity for ascertaining the smallest element from a binary `max-heap` and binary `min-heap`?
- What will be the time complexity to make a `max-heap` that combines two `max-heap` each of size n ?
- The level order traversal of `max-heap` is 12, 9, 7, 4, and 2. After inserting new elements 1 and 8, what will be the final `max-heap` and the level order traversal of the final `max-heap`?
- Which of the following is a binary `max-heap`?

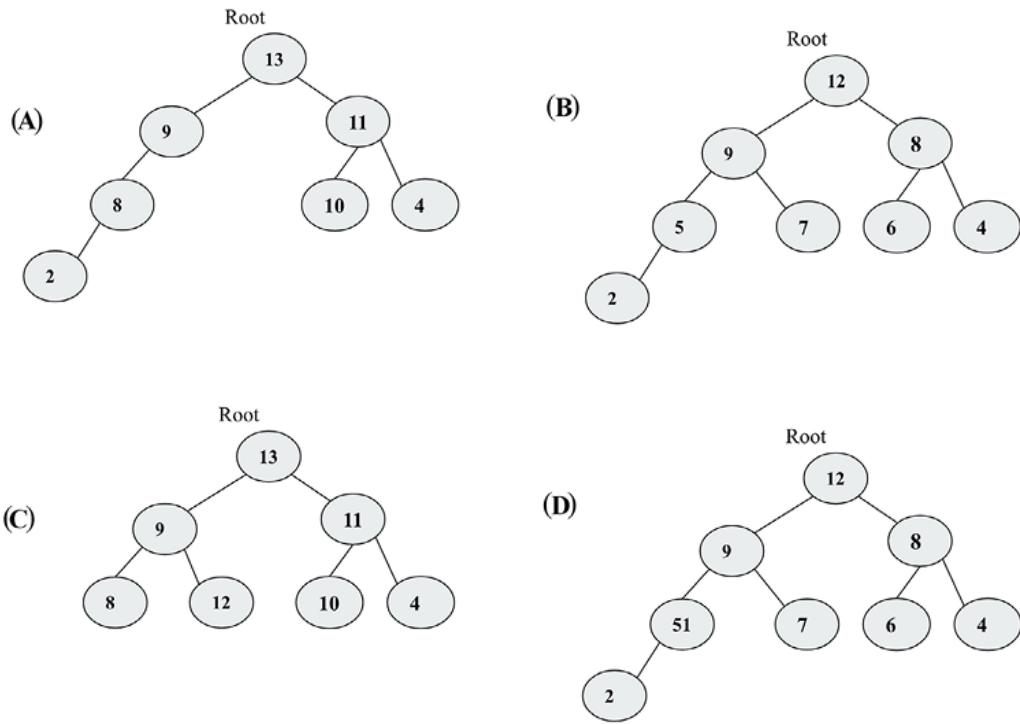


Figure 7.21: Example trees

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



8

Hash Tables

A hash table is a data structure that implements an associative array in which the data is stored by mapping the keys to the values as `key-value` pairs. In many applications, we mostly require different operations such as insert, search, and delete in a dictionary data structure. For example, a symbol table is a data structure based on a hash table that is used by the compiler. A compiler that translates a programming language maintains a symbol table in which keys are character strings that are mapped to the identifiers. In such situations, a hash table is an effective data structure since we can directly compute the index of the required record by applying a hash function to the key. So, instead of using the key as an array index directly, the array index is computed by applying the hash function to the key. It makes it very fast to access an element from any index from the hash table. The hash table uses the hashing function to compute the index of where the data item should be stored in the hash table.

While looking up an element in the hash table, hashing of the key gives the index of the corresponding record in the table. Ideally, the hash function assigns a unique value to each of the keys; however, in practice, we may get hash collisions where the hash function

generates the same index for more than one key. In this chapter, we will be discussing different techniques that deal with such collisions.

In this chapter, we will discuss all the concepts related to these, including:

- Hashing methods and hash table techniques
- Different collision resolution techniques in hash tables

Introducing hash tables

As we know, **arrays** and **lists** store the data elements in sequence. As in an array, the data items are accessed by an index number.

Accessing array elements using index numbers is fast. However, they are very inconvenient to use when it is required to access any element when we can't remember the index number. For example, if we wish to extract the phone number for a person from the address book at index 56, there is nothing to link a particular contact with number 56. It is difficult to retrieve an entry from the list using the index value.

Hash tables are a data structure better suited to this kind of problem. A **hash table** is a data structure where elements are accessed by a keyword rather than an index number, unlike in **lists** and **arrays**. In this data structure, the data items are stored in key-value pairs similar to dictionaries. A hash table uses a hashing function in order to find an index position where an element should be stored and retrieved. This gives us fast lookups since we are using an index number that corresponds to the hash value of the key.

An overview of how the hash table stores the data is shown in *Figure 8.1*, in which key values are hashed using any hash function to obtain the index position of the record in the hash table.

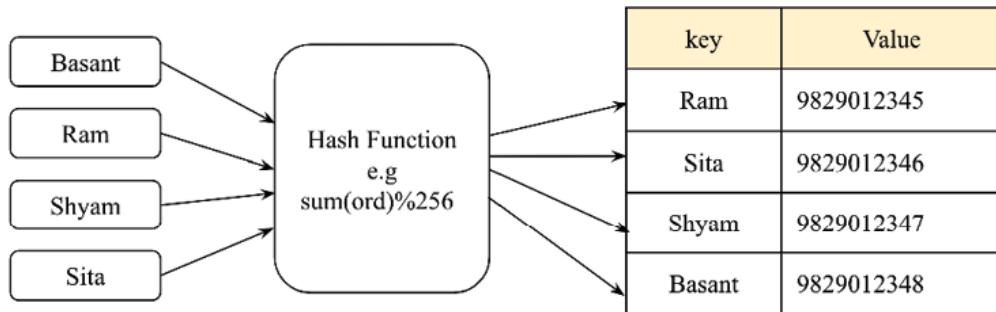


Figure 8.1: An example of a hash table

Dictionaries are a widely used data structure, often built using hash tables. A dictionary uses a keyword instead of an index number, and it stores data in (`key`, `value`) pairs. That is, instead of accessing the contact with the index value, we use the *key* value in the dictionary data structure.

The following code demonstrate the working of dictionaries that store the data in (`key`, `value`) pairs:

```
my_dict={"Basant" : "9829012345", "Ram": "9829012346", "Shyam": "9829012347"
print("All keys and values")
for x,y in my_dict.items():
    print(x, ":" , y)      #prints keys and values
my_dict["Ram"]
```

The output of the preceding code is as follows:

```
Basant : 9829012345
Ram : 9829012346
```

```
Shyam : 9829012347
Sita : 9829012348
'9829012346'
```

Hash tables stores the data in a very efficient way so that retrieval can be very fast. Hash tables are based on a concept called hashing.

Hashing functions

Hashing is a technique in which, when we provide data of arbitrary size to a function, we get a small, simplified value. This function is called a hash function. Hashing uses a hash function to map the keys to another range of data in a way that a new range of keys can be used as an index in the hash table; in other words, hashing is used to convert the key values to integer values, which can be used as an index in the hash table.

In our discussions in this chapter, we are using hashing to convert strings into integers. We could have used any other data type in place of strings that can be converted into integers. Let's take an example. Say, we want to hash the expression `hello world`, that is, we want to get a numeric value corresponding to this string that can be used as an index in the hash table.

In Python, the `ord()` function returns a unique integer value (known as ordinal values) that is mapped to a character in the Unicode encoding system. The ordinal values map the Unicode character to a unique numerical representation provided the character is Unicode-compatible, for example, numbers 0-127 are mapped to ASCII characters, which also correspond to the ordinal values within Unicode systems. However, the range of Unicode encoding may be

larger. So, Unicode encoding is a superset of ASCII. For example, in Python, we get a unique ordinal value `102` for character '`f`' by using `ord('f')`. Further, to get the hash of the whole string, we could just sum the ordinal numbers of each character in the string. See the following code snippet:

```
sum(map(ord, 'hello world'))
```

The output of the above is as follows:

```
1116
```

In the above output, we obtain a numeric value, `1116`, for the `hello world` string, which is the **hash** of the given string. Consider the following *Figure 8.2* to see the ordinal values of each character in the string that results in the hash value `1116`:

<code>h</code>	<code>e</code>	<code>l</code>	<code>l</code>	<code>o</code>		<code>w</code>	<code>o</code>	<code>r</code>	<code>l</code>	<code>d</code>
104	101	108	108	111	32	119	111	114	108	100

`= 1116`

Figure 8.2: Ordinal values of each character for the hello world string

The preceding approach used to obtain the hash value for a given string has the problem that more than one string can have the same hash value; for example, when we change the order of the characters in the string and we have the same hash value. See the following code snippet where we get the same hash value for the `'world hello'` string:

```
sum(map(ord, 'world hello'))
```

The output of the above is as follows:

```
1116
```

Again, there would be the same hash value for the `'gello xorld'` string, as the sum of the ordinal values of the characters for this string would be the same since `g` has an ordinal value that is one less than that of `h`, and `x` has an ordinal value that is one greater than that of `w`. See the following code snippet:

```
sum(map(ord, 'gello xorld'))
```

The output of the above is as follows:

```
1116
```

Look at the following *Figure 8.3*, where we can see that the hash value for this `'gello xorld'` string is again `1116`:

g	e	l	l	o		x	o	r	l	d
103	101	108	108	111	32	120	111	114	108	100

-1 +1

`= 1116`

Figure 8.3: Ordinal values of each character for the gello xorld string

In practice, most of the hashing functions are imperfect and face collisions. This means that a hash function gives the same hash value

to more than one string. Such collisions are undesirable for implementing the hash table.

Perfect hashing functions

A perfect **hashing function** is one by which we get a unique hash value for a given string (it can be any data type; here, we are using a string data type as an example). Our aim is to create a hash function that minimizes the number of collisions, is fast, easy to compute, and distributes the data items equally in the hash table. But, normally, creating an efficient hash function that is fast as well as providing a unique hash value for each string is very difficult. If we try to develop a hash function that avoids collisions, this becomes very slow, and a slow hash function does not serve the purpose of the hash table. So, we use a fast hash function and try to find a strategy to resolve the collisions rather than trying to find a perfect hash function.

To avoid the collisions in the hash function discussed in the previous section, we can add a multiplier to the ordinal value of each character that continuously increases as we progress in the string. Furthermore, the hash value of the string can be obtained by adding the multiplied ordinal value of each character. To better understand the concept, refer to the following *Figure 8.4*:

h	e	l	l	o		w	o	r	l	d	
104	101	108	108	111	32	119	111	114	108	100	= 1116
1	2	3	4	5	6	7	8	9	10	11	
104	202	324	432	555	192	833	888	1026	1080	1100	= 6736

Figure 8.4: Ordinal values multiplied by numeric values for each character of the hello world string

In the preceding *Figure 8.4*, the ordinal value of each character is progressively multiplied by a number. Note that row two has the ordinal values of each character; row three shows the multiplier value; and, in row four, we get values by multiplying the values of rows two and three so that `104 x 1` equals `104`. Finally, we add all of these multiplied values to get the hash value of the `hello world` string, that is, `6736`.

The implementation of this concept is shown in the following function:

```
def myhash(s):
    mult = 1
    hv = 0
    for ch in s:
        hv += mult * ord(ch)
        mult *= 1
    return hv
```

We can test this function on the strings that we used earlier, shown as follows:

```
for item in ('hello world', 'world hello', 'gello xorld'):
    print("{}: {}".format(item, myhash(item)))
```

When we execute the preceding code, we get the following output:

```
hello world: 6736  
world hello: 6616  
gello xorld: 6742
```

We can see that this time, we get different hash values for these three strings. Still, this is not a perfect hash. Let's now try the strings `ad` and `ga`:

```
for item in ('ad', 'ga'):  
    print("{}: {}".format(item, myhash(item)))
```

The output of the preceding code snippet is as follows:

```
ad: 297  
ga: 297
```

So, we still do not have a perfect hash function since we get the same hash values for these two different strings. Therefore, we need to devise a strategy for resolving such collisions. We will discuss more strategies to resolve collisions in the next sections.

Resolving collisions

Each position in the hash table is often called a **slot** or **bucket** that can store an element. Each data item in the form of a `(key, value)` pair is stored in the hash table at a position that is decided by the hash value of the key. Let's take an example in which firstly we use the hashing function that computes the hash value by summing up the

ordinal values of all the characters. Then, we compute the final hash value (in other words, the index position) by computing the total ordinal values of module 256. Here, we use 256 slots/buckets as an example. We can use any number of slots depending upon how many records we require in the hash table. We show a sample hash in *Figure 8.5*, which has key strings corresponding to data values, for example, the `eggs` key string has the corresponding data value `123456789`.

This hash table uses a hashing function that maps the input string `hello world` to a hash value of `92`, which finds a slot position in the hash table:

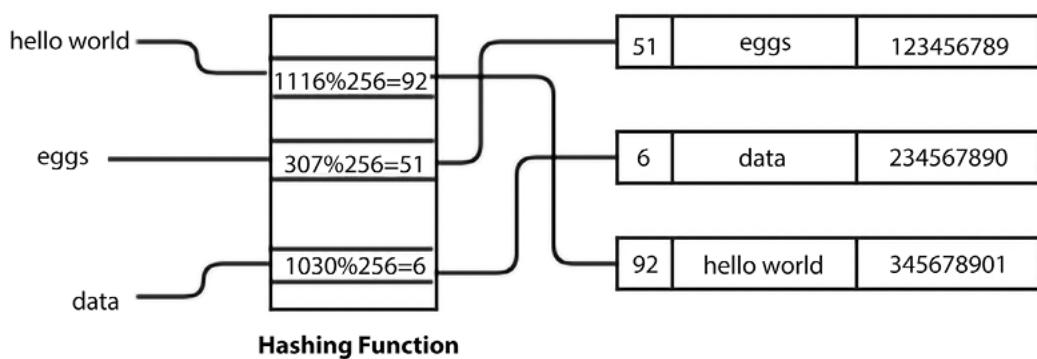


Figure 8.5: A sample hash table

Once we know the hash value of the key, it will be used to find the position where the element should be stored in the hash table. So, we need to find an empty slot. We start at the slot that corresponds to the hash value of the key. If that slot is empty, we insert the data item there. And, if the slot is not empty, that means we have a collision. It means that we have a hash value for the item that is the same as a previously stored item in the table. We need to ascertain a strategy to avoid such collisions or conflicts.

For example, in the following diagram, the key string `hello world` is already stored in the table at index position `92`, and with a new key string, for example, `world hello`, we get the same hash value of `92`. This means that there is a collision. Refer to the following *Figure 8.6* depicting this concept:

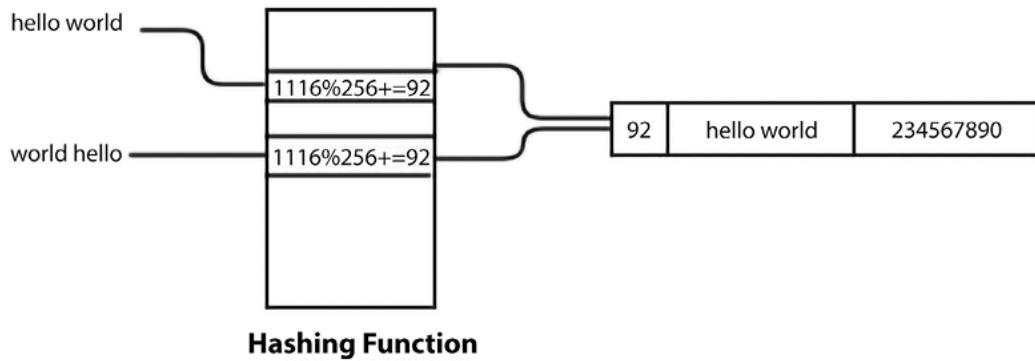


Figure 8.6: Hash values of two strings are the same

One way of resolving this kind of collision is to find another free slot from the position of the collision. This collision resolution process is called **open addressing**.

Open addressing

In open addressing, the key values are stored in the hash table, and collisions are resolved using the probing technique. Open addressing is a collision resolution technique used in hash tables. The collision is resolved by searching (also called probing) an alternate position until we get an unused slot in the hash table to store the data item.

There are three popular approaches for an open addressing-based collision resolution technique:

1. Linear probing
2. Quadratic probing
3. Double hashing

Linear probing

The systematic way of visiting each slot is a linear way of resolving collisions, in which we linearly look for the next available slot by adding 1 to the previous hash value where we get the collision. This is known as linear probing. We can resolve the conflict by adding 1 to the sum of the ordinal values of each character in the key string, which is further used to compute the final hash value by taking its modulo according to the size of the hash table.

Let's consider an example. First, compute the hash value of the key. If the position is occupied, we check the hash table sequentially for the next free slot. Let's use this to resolve a collision, as shown in the following *Figure 8.7*, wherein, for the key string egg, the sum of ordinal values comes to 307, and then we compute the hash value by taking the module 256, which gives the hash value for the egg key string as 51. However, data is already stored at this position, so this means a collision. Therefore, we add 1 to the hash value that is computed by the sum of the ordinal values of each character of the string. In this way, we obtain a new hash value, 52, for this key string to store the data. Refer to the following *Figure 8.7*, which depicts the above process:

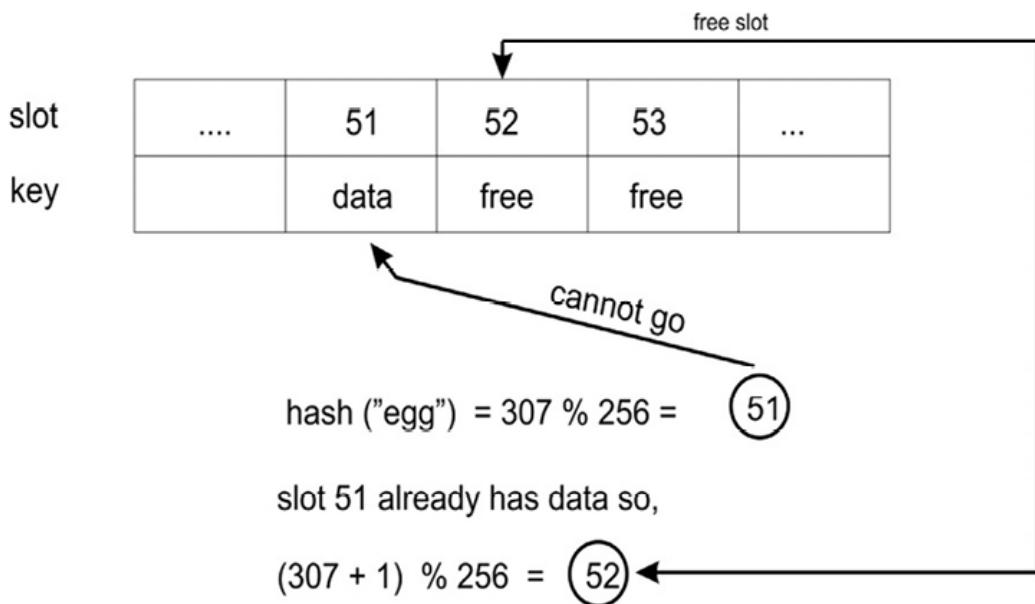


Figure 8.7: An example of collision resolution

In order to find the next free slot in the hash table, we increment the hashing value, and this increment is fixed in the case of linear probing. Due to a fixed increment in the hashing value when we get collisions, the new data element is always stored at the next available index position given by the hash function. This creates a continuous cluster of occupied index positions, with this cluster growing whenever we get another data element that has a hash value anywhere within the cluster.

So, one major drawback of this approach is that the hash table can have consecutive occupied positions that are called clusters of items. In this case, one portion of the hash table may become dense, with the other part of the table remaining empty. Because of these limitations, we may prefer to use a different strategy to resolve collisions such as quadrant probing or double hashing, which we will discuss in forthcoming sections. Let us first discuss the implementation of the hash table with linear probing as a collision

resolution technique, and after understanding the concept of linear probing, we will discuss other collision resolution techniques.

Implementing hash tables

To implement the hash table, we start by creating a class to hold hash table items. These need to have a key and a value since the hash table is a `{key-value}` store:

```
class HashItem:  
    def __init__(self, key, value):  
        self.key = key  
        self.value = value
```

Next, we start working on the hash table class itself. As usual, we start with a constructor:

```
class HashTable:  
    def __init__(self):  
        self.size = 256  
        self.slots = [None for i in range(self.size)]  
        self.count = 0
```

Standard Python lists can be used to store data elements in a hash table. Let's set the size of the hash table to `256` elements to start with. Later, we will look at strategies for how to grow the hash table as we begin filling it up. We will now initialize a list containing `256` elements in the code. These are the positions where the elements are to be stored—the slots or buckets. So, we have `256` slots to store elements in the hash table. It is important to note the difference between the size and count of a table. The size of a table refers to the

total number of slots in the table (used or unused). The count of the table refers to the number of slots that are filled, meaning the number of actual `(key-value)` pairs that have been added to the table.

Now, we have to decide on a hashing function for the table. We can use any hash function. Let's take the same hash function that returns the sum of ordinal values for each character in the strings with a slight modification. Since this hash table has `256` slots, that means we need a hashing function that returns a value in the range of `0` to `255` (the size of the table). A good way of doing it is to return the remainder of dividing the hash value by the size of the table since the remainder would surely be an integer value between `0` and `255`.

Since the hashing function is only meant to be used internally by the class, we put an underscore (`_`) at the beginning of the name to indicate this. This is a Python convention for indicating that something is intended for internal use. Here is the implementation of the `hash` function, which should be defined in the `HashTable` class:

```
def __hash__(self, key):
    mult = 1
    hv = 0
    for ch in key:
        hv += mult * ord(ch)
        mult *= 1
    return hv % self.size
```

For the time being, we are assuming that keys are strings. We will discuss how non-string keys can be used later. For now, the `_hash()` function is going to generate the hash value for a string.

Storing elements in a hash table

To store the elements in the hash table, we add them to the table with the `put()` function and retrieve them with the `get()` function. First, we will look at the implementation of the `put()` function. We start by adding the key and the value to the `HashItem` class and then compute the hash value of the key. The `put()` method should be defined in the `HashTable` class:

```
def put(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + 1) % self.size
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()
```

After obtaining the hash value of the key and if the slot is not empty, the next free slot is checked by adding `1` to the previous hash value by applying the linear probing technique. Consider the following code:

```
while self.slots[h] != None:
    if self.slots[h].key == key:
        break
    h = (h + 1) % self.size
```

If the slot is empty, then we increase the count by one and store the new element (meaning the slot contained `None` previously) in the list

at the required position. Refer to the following code:

```
if self.slots[h] is None:  
    self.count += 1  
self.slots[h] = item  
self.check_growth()
```

In the above code, we have created a hash table and discussed the `put()` method for storing the data element in the hash table with the linear probing technique at the time of the collision.

In the last line of the preceding code, we call a `check_growth()` method, which is used to expand the size of the hash table when we have a very limited number of empty slots remaining in the hash table. We will discuss this in more detail in the next section.

Growing a hash table

In the example that we have discussed, we have fixed the hash table size at `256`. It is obvious that, when we add the elements to the hash table, the hash table starts filling up, and at some point, all of the slots would be filled up and the hash table will be full. To avoid such a situation, we can grow the size of the table when it is starting to get full.

To grow the size of the hash table, we compare the size and the count in the table. `size` is the total number of slots, and `count` denotes the number of slots that contain elements. So, if `count` is equal to `size`, this means we have filled up the table. The load factor of the hash table is generally used to expand the size of the table; that gives us an indication of how many available slots of the table have been

used. The load factor of the hash table is computed by dividing the number of **used** slots by the **total** number of slots in the table. It is defined as follows:

$$\text{Load factor} = n/k$$

Here, n is the number of used slots, and k is the total number of slots. As the load factor value approaches 1, this means that the table is going to be filled, and we need to grow the size of the table. It is better to grow the size of the table before it gets almost full, as the retrieval of elements from the table becomes slow when the table fills up. A value of 0.75 for the load factor may be a good value to grow the size of the table. Another question is how much we should increase the size of the table. One strategy would be to simply double its size.

The problem of linear probing is that as the load factor increases, it takes a long time to find the insertion point for the new element. Moreover, in the case of the open addressing collision resolution technique, we should grow the size of the hash table depending upon the load factor to reduce the number of collisions.

The implementation of growing the hash table when the load factor increases more than the threshold is as follows. First, we redefine the `HashTable` class that includes one more variable, `MAXLOADFACTOR`, that is used to ensure that the load factor of the hash table is always below the predefined maximum load factor. The `HashTable` class is defined as follows:

```
class HashTable:  
    def __init__(self):  
        self.size = 256
```

```
    self.slots = [None for i in range(self.size)]
    self.count = 0
    self.MAXLOADFACTOR = 0.65
```

Next, we check the load factor of the hash table after adding any record to the hash table using the following `check_growth()` method, which should be defined in the `HashTable` class:

```
def check_growth(self):
    loadfactor = self.count / self.size
    if loadfactor > self.MAXLOADFACTOR:
        print("Load factor before growing the hash table", self.c
        self.growth()
        print("Load factor after growing the hash table", self.co
```

In the preceding code, we compute the load factor of the table, and then we check if it is more than the set threshold (in other words, `MAXLOADFACTOR` is a variable that we initialize at the time of creating a hash table). In that case, we call the `growth()` method that increases the hash table size (in this example, we are doubling the hash table size). The `growth()` method, which should be defined in the `HashTable` class, is implemented as follows:

```
def growth(self):
    New_Hash_Table = HashTable()
    New_Hash_Table.size = 2 * self.size
    New_Hash_Table.slots = [None for i in range(New_Hash_Table.si

    for i in range(self.size):
        if self.slots[i] != None:
            New_Hash_Table.put(self.slots[i].key, self.slots[i].v

    self.size = New_Hash_Table.size
    self.slots = New_Hash_Table.slots
```

In the preceding code, we firstly create a new hash table double the size of the original hash table and then we initialize all of its slots to be `None`. Next, we check all the filled slots in the original hash table where we have the data, since we have to insert all these existing records into the new hash table, hence, we call the `put()` method with all the key-value pairs of the existing hash table. Once we copy all the records to the new hash table, we replace the size and slots of the existing table with the new hash table.

Let's create a hash table with a maximum capacity of 10 records and a threshold load factor of 65% by defining `self.size = 10` in the `__init__` method in the `HashTable` class, meaning whenever a seventh record is added to the hash table, we call a `check_growth()` method:

```
ht = HashTable()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")
ht.put("awd", "do not")
ht.put("add", "do not")
ht.checkGrow()
```

In the above code, we add seven records using the `put()` method. The output of the preceding code is as follows:

```
Load factor before growing the hash table 0.7
Load factor after growing the hash table 0.35
```

In the above output, we can see that the load factor before and after adding the seventh record became half of the load factor before growing the hash table.

In the next section, we will discuss the `get()` method for retrieving the data element that we have stored in the hash table.

Retrieving elements from the hash table

To retrieve the elements from the hash table, the value stored corresponding to the key would be returned. Here, we discuss the implementation of the retrieval method—the `get()` method. This method returns the value stored in the table corresponding to the given key.

Firstly, we compute the hash of the given key corresponding to the value that is to be retrieved. Once we have the hash value of the key, we look up the hash table at the position of the hash value. If the key item is matched with the stored key value at that location, the corresponding value is retrieved.

If that does not match, then we add 1 to the sum of the ordinal values of all the characters in the string, similar to what we did at the time of storing the data, and we look at the newly obtained hash value. We keep searching until we get the key element, or we check all the slots in the hash table.

Here, we used the linear probing technique to resolve the collision, and hence we use the same technique when retrieving the data element from the hash table. Hence, if we were to use a different

technique, let's say double hashing or quadratic probing at the time of storing the data element, we should use the same method to retrieve the data element. Consider an example to understand the concept in *Figure 8.8*, and in the following four steps:

1. We compute the hash value for the given key string, `egg`, which turns out to be `51`. Then, we compare this key with the stored key value at location `51`, but it does not match.
2. As the key does not match, we compute a new hash value.
3. We look up the key at the location of the newly created hash value, which is `52`; we compare the key string with the stored key value and, here, it matches, as shown in the following diagram.
4. The stored value is returned corresponding to this key value in the hash table. See the following *Figure 8.8*:

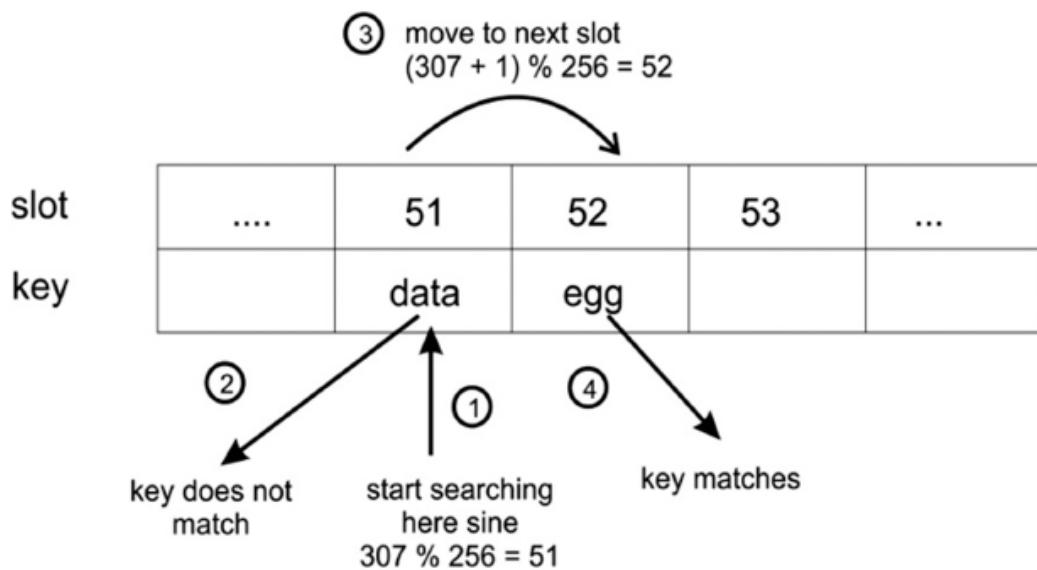


Figure 8.8: Four steps are demonstrated for retrieving an element from the hash table

To implement this retrieval method, that is, the `get()` method, we start by calculating the hash of the key. Next, we look up the

computed hash value in the table. If there is a match, we return the corresponding stored value. Otherwise, we keep looking at the new hash value location computed as described. Here is the implementation of the `get()` method, which should be defined in the `HashTable` class:

```
def get(self, key):
    h = self._hash(key)      # computed hash for the given key
    while self.slots[h] != None:
        if self.slots[h].key == key:
            return self.slots[h].value
        h = (h+ 1) % self.size
    return None
```

Finally, we return `None` if the key was not found in the table; we could have printed the message that the key is not found in the hash table.

Testing the hash table

To test the hash table, we create `HashTable` and store a few elements in it, and then try to retrieve them. We can use `get()` method to find out if a record exists for a given key. We also use the two strings, `ad` and `ga`, that had the collision and returned the same hash value with our hashing function. To evaluate the work of the hash table, we throw this collision as well, just to see that the collision is properly resolved. Refer to the example code, as follows:

```
ht = HashTable()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
```

```
ht.put("ad", "do not")
ht.put("ga", "collide")
for key in ("good", "better", "best", "worst", "ad", "ga"):
    v = ht.get(key)
    print(v)
```

After executing the above code, we get the following output:

```
eggs
ham
spam
none
do not
collide
```

As you can see, looking up the `worst` key returns `None`, since the key does not exist. The `ad` and `ga` keys also return their corresponding values, showing that the collision between them is handled properly.

Implementing a hash table as a dictionary

Using the `put()` and `get()` methods to store and retrieve elements in the hash table may look slightly inconvenient. However, we can also use the hash table as a dictionary, as it would be easier to use. For example, we would like to use `ht["good"]` instead of `ht.get("good")` to retrieve elements from the table.

This can easily be done with the special methods, `__setitem__()` and `__getitem__()`, which should be defined in the `HashTable` class.

See the following code for this:

```
def __setitem__(self, key, value):
    self.put(key, value)
def __getitem__(self, key):
    return self.get(key)
```

Now, our test code would be like the following:

```
ht = HashTable()
ht["good"] = "eggs"
ht["better"] = "ham"
ht["best"] = "spam"
ht["ad"] = "do not"
ht["ga"] = "collide"
for key in ("good", "better", "best", "worst", "ad", "ga"):
    v = ht[key]
    print(v)
print("The number of elements is: {}".format(ht.count))
```

The output of the preceding code is as follows:

```
eggs
ham
spam
none
do not
collide
The number of elements is: 5
```

Notice that we also print the number of elements already stored in the hash table using the `count` variable. The above code does the same thing as we did in the previous section, but it is just more convenient to use.

In the next section, we discuss the quadratic probing technique for collision resolution.

Quadratic probing

This is also an open addressing scheme for resolving collisions in hash tables. It resolves the collision by computing the hash value of the key and adding successive values of a quadratic polynomial; the new hash is iteratively computed until an empty slot is found. If a collision occurs, the next free slots are checked at the locations $h + 1^2$, $h + 2^2$, $h + 3^2$, $h + 4^2$, and so on. Hence, the new hash value is computed as follows:

```
new-hash(key) = (old-hash-value + i2)
```

Here, `hash-value` = `key mod table_size`

When we have a key as strings, we compute the hash value using the sum of the ordinal values multiplied by numeric values for each character, and then we pass it the hash function to finally obtain the hash of the key string. However, in the case of non-string key elements, we can use the hash function directly to compute the hash of the key.

Let us take a simple example of a hash table in which we have seven slots and assume that the hash function is `h(key) = key mod 7`. To understand the concept of quadratic probing, let's assume that we have key element values that are the hash of the given key strings.

So, whenever we use the quadratic probing technique to ascertain the next index positions to store a data element when we have a collision, we should perform the following steps to resolve the collision:

1. Initially, since we have an empty table, when we get a key element of 15 (assuming it is a hash of the given string), we compute the hash value using our given hash function, in other words, $15 \bmod 7 = 1$. So, the data element is stored at index position 1.
2. Then, let's say we get a key element of 22 (assuming it is a hash of the next given string), we use the hash function to compute the hash value, in other words, $22 \bmod 7 = 1$, it gives the index position 1. Since index position 1 is already occupied, there is a collision, so we compute a new hash value using quadratic probing, which is $(1+1^2=2)$. The new index position is 2. Therefore, the data element is stored at index position 2.
3. Next, assuming that we get a data element of 29 (assuming it is a hash of the given string), we compute the hash value $29 \bmod 7 = 1$. Since we have a collision here, we compute the hash value again as in step 2, but we get another collision here, so we have to recompute the hash value once more, in other words $(1+2^2=5)$, so the data is stored at that location.

The above example of resolving the process using the quadratic probing technique is shown in *Figure 8.9*:

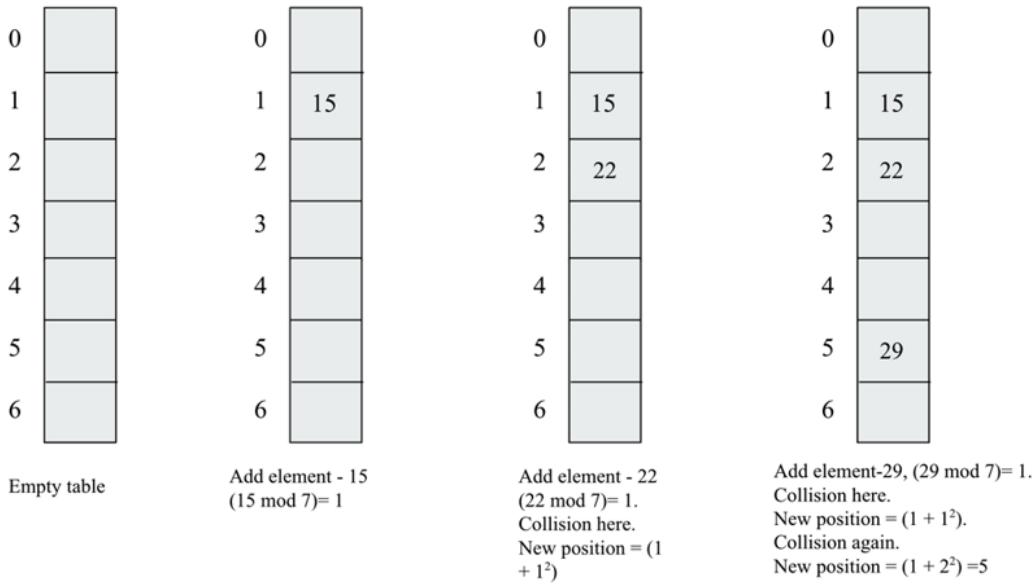


Figure 8.9: Example of collision resolution using quadratic probing

The quadratic probing technique for collision avoidance does not suffer from the formation of clusters of items in the same way as linear probing; however, it does suffer from secondary clustering. Secondary clustering creates a long run of filled slots since the data elements that have the same hash value will also have the same probe sequence.

We discussed the implementation of a hash table in the previous section with the addition and retrieval of data elements, and we used the linear probing technique to resolve the collision. Now, we can update the implementation of the hash table if we want to use any other collision resolution technique, such as the quadratic probing technique. All the methods will be the same in the `HashTable` class except the following two methods, which should be defined in the `HashTable` class:

```
def get_quadratic(self, key):
    h = self._hash(key)
```

```

j = 1
while self.slots[h] != None:
    if self.slots[h].key == key:
        return self.slots[h].value
    h = (h+ j*j) % self.size
    j = j + 1
return None
def put_quadratic(self, key, value):
    item = HashItem(key, value)
    h = self._hash(key)
    j = 1
    while self.slots[h] != None:
        if self.slots[h].key == key:
            break
        h = (h + j*j) % self.size
        j = j+1
    if self.slots[h] == None:
        self.count += 1
    self.slots[h] = item
    self.check_growth()

```

The above code of the `get_quadratic()` and `put_quadratic()` methods are similar to the implementation of the `get()` and `put()` methods that we discussed earlier, except for the fact that the code statements are in bold in the preceding codes. The bold statements are indicating that at the time of the collision, we check the next empty slot using the quadratic probing formula:

```

ht = HashTable()
ht.put_quadratic("good", "eggs")
ht.put_quadratic("ad", "packt")
ht.put_quadratic("ga", "books")

v = ht.get_quadratic("ga")
print(v)

```

In the above code, we first add three data elements along with their associated values, and then we search for a data item with the key "ga" in the hash table. The output of the preceding code is as follows:

```
books
```

The above output corresponds to the key string "ga", which is correct as per the input data stored in the hash table. Next, we will discuss another collision resolution technique – double hashing.

Double hashing

In the double hashing collision resolution technique, we use two hashing functions. This technique works as follows. Firstly, the primary hash function is used to compute the index position in the hash table, and whenever we get a collision, we use another hash function to decide the next free slot to store the data by incrementing the hashing value.

In order to find the next free slot in the hash table, we increment the hashing value, and this increment is fixed in the case of linear probing and quadratic probing. Due to a fixed increment in the hashing value when we get collisions, the record is always moved to the next available index position given by the hash function. It creates a continuous cluster of occupied index positions. This cluster grows whenever we get another record that has a hash value anywhere within the cluster.

However, in the case of the double hashing technique, the probing interval depends on the key data itself, meaning that we always map

to the different index positions in the hash table whenever we get a collision, which, in turn, helps in avoiding the formation of clusters.

The probing sequence for this collision resolving technique is as follows:

```
(h1(key)+i*h2(key))mod table_size  
h1(key) = key mod table_size
```

It is important to note here that the second hash function should be fast, easy to compute, should not evaluate to 0, and should be different from the first hash function.

One choice for the second hash function can be defined as follows:

```
h2(key) = prime_number - (key mod prime_number)
```

In the above hash function, the prime number should be less than the table size.

For example, let's say we have a hash table that can have a maximum of seven slots when we add data elements {15, 22, 29} to this table in sequence. The following steps are performed to store these data elements in the hash table using the double hashing technique when we get a collision:

1. Firstly, we have data element 15, and we compute the hash value using the primary hash function, in other words, $(15 \text{ mod } 7 = 1)$. Since the table is empty initially, we store the data at index position 1.

2. Next, the data element is 22, and we compute the hash value using the primary hash function, in other words, $(22 \bmod 7 = 1)$. Since the index position 1 is already filled, this means there is a collision. Next, we use the secondary hashing function defined above as $h^2(key) = prime_number - (key \bmod prime_number)$ to ascertain the next index positions in the hash table. Here, we assume that the prime number less than the table size is 5. This means that the next index position in the hash table will be $(1 + 1*(5 - (22 \bmod 5))) \bmod 7$, which is equivalent to 4. So, we store this data element at index position 4.
3. Next, we have data element 29, so we compute the hash value using the primary hashing function, in other words, $(29 \bmod 7 = 1)$. We get a collision, and now we use the secondary hash function to establish the next index position for storing the data element, in other words, $(1 + 1*(5 - (29 \bmod 5))) \bmod 7$, which turns out to be 2, so we store this data element at location 2.

The above example of the process of resolving the collision using double hashing is shown in *Figure 8.10*:

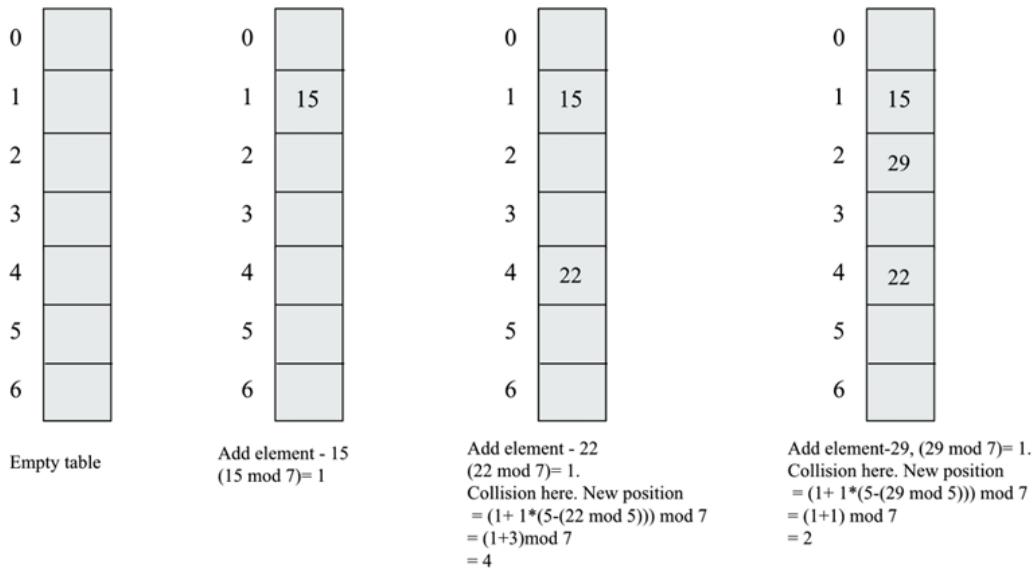


Figure 8.10: Example of collision resolution using double hashing

Let us now see how we can implement the hash table with the double hashing technique to resolve the collision. The `put_double_hashing()` and `get_double_hashing()` methods are given as follows, which should be defined in the `HashTable` class.

The following `h2()` method is used to compute the sum of the ordinal values since, in our examples, we have strings as a key element:

```
def h2(self, key):
    mult = 1
    hv = 0
    for ch in key:
        hv += mult * ord(ch)
        mult += 1
    return hv
```

Furthermore, we should redefine the hash table to include a prime number as a variable that will be used in computing the secondary

hash function:

```
class HashTable:  
    def __init__(self):  
        self.size = 256  
        self.slots = [None for i in range(self.size)]  
        self.count = 0  
        self.MAXLOADFACTOR = 0.65  
        self.prime_num = 5
```

The following code is designed to insert a data element and associated value in the hash table and use the double hashing technique at the time of collision:

```
def put_double_hashing(self, key, value):  
    item = HashItem(key, value)  
    h = self._hash(key)  
    j = 1  
    while self.slots[h] != None:  
        if self.slots[h].key == key:  
            break  
        h = (h + j * (self.prime_num - (self.h2(key) % self.prime_num)))  
        j = j+1  
    if self.slots[h] == None:  
        self.count += 1  
    self.slots[h] = item  
    self.check_growth()  
  
def get_double_hashing(self, key):  
    h = self._hash(key)  
    j = 1  
    while self.slots[h] != None:  
        if self.slots[h].key == key:  
            return self.slots[h].value  
        h = (h + j * (self.prime_num - (self.h2(key) % self.prime_num)))  
        j = j + 1  
    return None
```

The above code of the `get_doubleHashing()` and `put_doubleHashing()` methods are very similar to the implementation of the `get()` and `put()` methods that we discussed earlier, except for the statements that are in bold in the preceding codes. The statements in bold are showing that at the time of the collision, we use the double hashing technique formula to get the next empty slot in the hash table:

```
ht = HashTable()
ht.put_doubleHashing("good", "eggs")
ht.put_doubleHashing("better", "spam")
ht.put_doubleHashing("best", "cool")
ht.put_doubleHashing("ad", "donot")
ht.put_doubleHashing("ga", "collide")
ht.put_doubleHashing("awd", "hello")
ht.put_doubleHashing("addition", "ok")

for key in ("good", "better", "best", "worst", "ad", "ga"):
    v = ht.get_doubleHashing(key)
    print(v)
print("The number of elements is: {}".format(ht.count))
```

In the above code, we first insert seven different data elements along with their associated values, and then we search and check a few random data items in the hash table. The output of the preceding code is as follows:

```
eggs
spam
cool
none
donot
collide
The number of elements is: 7
```

In the above output, we can observe that the key string `worst` is not present in the hash table, meaning the output corresponding to this is `None`.

Linear probing leads to primary clustering, while quadratic probing may lead to secondary clustering, whereas the double hashing technique is one of the most effective methods for collision resolution since it does not yield any clusters. The advantage of this technique is that it produces a uniform distribution of records in the hash table.

In open addressing collision resolution techniques, we search for another empty slot within the hash table, as we did in linear probing, quadratic probing, and double hashing. “closed” in “closed hashing” refers to the fact that we do not leave the hash table, and every record is stored at an index position given by the hash function, hence “closed hashing” and “open addressing” are synonyms.

On the other hand, when a record is always stored at an index position given by the hash function, this is known as the “closed addressing,” or “open hashing,” technique. Here, “open” in “open hashing” refers to the fact that we are open to leaving the hash table through a separate list where the data elements can be stored; for example, separate chaining is a closed addressing technique.

In the next section, we will discuss another collision resolution technique – the chaining technique.

Separate chaining

Separate chaining is another method to handle the problem of collision in hash tables. It solves this problem by allowing each slot in the hash table to store a reference to many items at the position of a collision. So, at the index of a collision, we are allowed to store multiple items in the hash table.

In chaining, the slots in the hash table are initialized with empty lists. When a data element is inserted, it is appended to the list that corresponds to that element's hash value. For example, in the following *Figure 8.11*, there is a collision for the key strings `hello world` and `world hello`. In the case of chaining, both data elements are stored using a list at the index position given by the hash function, in other words, `92` in the example shown in *Figure 8.11*. Here is an example to show collision resolution using chaining:

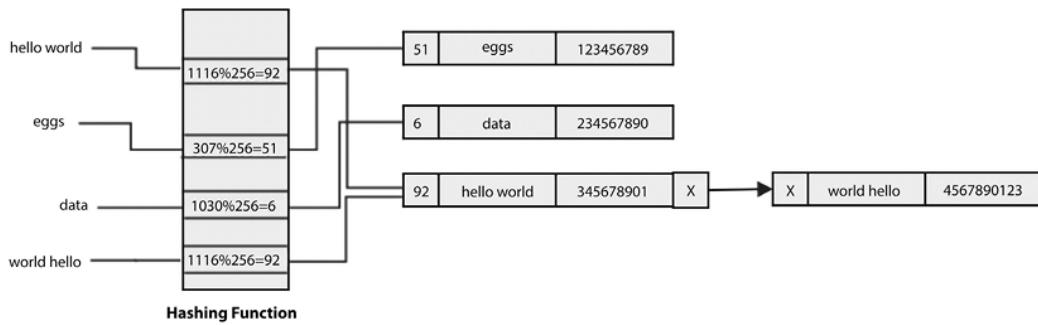


Figure 8.11: Example of collision resolution using chaining

One more example is shown in *Figure 8.12*, wherein if we have many data elements that have a hash value of `51`, all of these elements would be added to the list that exists in the same slot of the hash table:



Figure 8.12: More than one element having the same hash value stored in a list

Chaining then avoids conflict by allowing multiple elements to have the same hash value. Hence, there is no limit in terms of the number of elements that can be stored in a hash table, whereas, in the case of open addressing collision resolution techniques, we had to fix the size of the table, which we need to later grow when the table is filled up. Moreover, the hash table can hold more values than the number of available slots, since each slot holds a list that can grow.

However, there is a problem with chaining—it becomes inefficient when a list grows at a particular hash value location. As a particular slot has many items, searching them can become very slow since we have to do a linear search through the list until we find the element that has the key we want. This can slow down retrieval, which is not good since hash tables are meant to be efficient. Hence, the worst-case time complexity for searching in a separate chaining algorithm using linked lists is $O(n)$, because in the worst case, all the items will be added to only one index position in the hash table, and searching an item will work just similar to a linked list. The following *Figure 8.13* demonstrates a linear search through list items until we find a match:

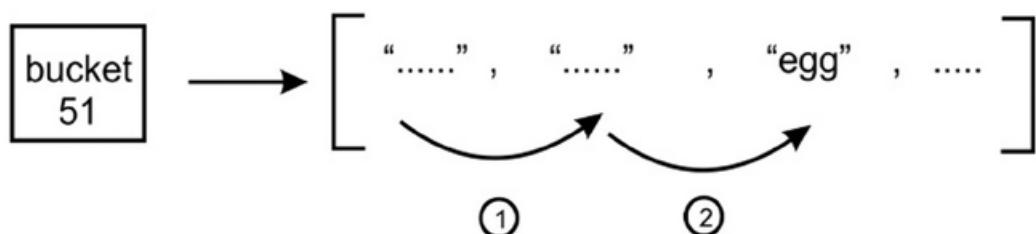


Figure 8.13: Demonstration of a linear search for the hash value of 51

So, there is a problem with the slow retrieval of items when a particular position in a hash table has many entries. This problem can be resolved using another data structure in place of using a list that can perform fast searching and retrieval. There is a nice choice of using **binary search trees (BSTs)**, which provide fast retrieval, as we discussed in the previous chapter.

We could simply insert an (initially empty) BST into each slot, as shown in the following *Figure 8.14*:

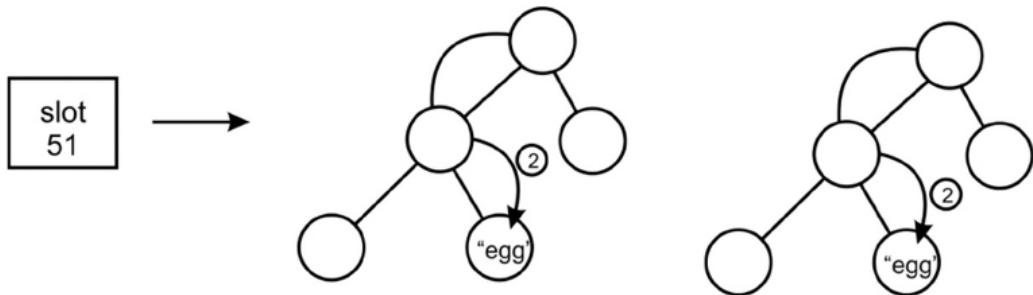


Figure 8.14: BST for a bucket for the hash value of 51

In the preceding diagram, the `51` slot holds a BST, which we use to store and retrieve the data items. However, we would still have a potential problem—depending on the order in which the items were added to the BST, we could end up with a search tree that is as inefficient as a list. That is, each node in the tree has exactly one child. To avoid this, we would need to ensure that our BST is self-balancing.

Here is the implementation of the hash table with separate chaining. Firstly, we create a `Node` class to store the key-value pairs and one pointer for pointing to the next node in the linked list:

```
class Node:  
    def __init__(self, key=None, value=None):  
        self.key = key  
        self.value = value  
        self.next = None
```

Next, we define the singly linked list, the details of which are provided in *Chapter 4, Linked Lists*. Here, we have defined the `append()` method for adding a new data record to the linked list:

```
class SinglyLinkedList:  
    def __init__(self):  
        self.tail = None  
        self.head = None  
  
    def append(self, key, value):  
        node = Node(key, value)  
        if self.tail:  
            self.tail.next = node  
            self.tail = node  
        else:  
            self.head = node  
            self.tail = node
```

Next, we define the `traverse()` method, which prints all the data records with `key-value` pairs. The `traverse()` method should be defined in the `SinglyLinkedList` class. We start from the head node, and move the next nodes while iterating through the `while` loop:

```
def traverse(self):  
    current = self.head  
    while current:  
        print("\\"", current.key, "--", current.value, "\")  
        current = current.next
```

Next, we define a `search()` method that matches the key that we want to search in the linked list. If the key matches any of the nodes, the corresponding key-value pair is printed. The `search()` method should be defined in the `SinglyLinkedList` class:

```
def search(self, key):
    current = self.head
    while current:
        if current.key == key:
            print("\\"Record found:", current.key, "-", current.value)
            return True
        current = current.next
    return False
```

Once, we have defined the linked list and all the required methods, we define the `HashTableChaining` class, in which we initialize the hash table with its size and all the slots with an empty linked list:

```
class HashTableChaining:
    def __init__(self):
        self.size = 6
        self.slots = [None for i in range(self.size)]
        for x in range(self.size) :
            self.slots[x] = SinglyLinkedList()
```

Next, we define the hash function, in other words, `_hash()`, similar to what we have discussed in previous sections:

```
def _hash(self, key):
    mult = 1
    hv = 0
    for ch in key:
        hv += mult * ord(ch)
```

```
        mult += 1
    return hv % self.size
```

Then, we define the `put()` method to insert a new data record in the hash table. Firstly, we create a node with key-pair pairs and then compute the index position based on the hash function. Then, we append the node at the end of the linked list associated with the given index position. The `put()` method should be defined in the `HashTableChaining` class:

```
def put(self, key, value):
    node = Node(key, value)
    h = self._hash(key)
    self.slots[h].append(key, value)
```

Next, we define the `get()` method to retrieve the data elements given the key value from the hash table. Firstly, we compute the index position using the same hash function that we used at the time of adding the records to the hash table, and then we search the required data record in the linked list associated with the given index position computed. The `get()` method should be defined in the `HashTableChaining` class:

```
def get(self, key):
    h = self._hash(key)
    v = self.slots[h].search(key)
```

Finally, we can define the `printHashTable()` method, which prints the complete hash table showing all the records of the hash table:

```
def printHashTable(self) :
    print("Hash table is :- \n")
    print("Index \t\tValues\n")
    for x in range(self.size) :
        print(x,end="\t\n")
        self.slots[x].traverse()
```

We can use the following code to insert a few sample data records in the hash table and we use the chaining technique to store the data. Then, we search a data record with the key string `best`, and we also print the complete hash table:

```
ht = HashTableChaining()
ht.put("good", "eggs")
ht.put("better", "ham")
ht.put("best", "spam")
ht.put("ad", "do not")
ht.put("ga", "collide")
ht.put("awd", "do not")

ht.printHashTable()
```

The output of the preceding code is as follows:

```
Hash table is :-
Index          Values
0
1
2
" good - eggs "
3
" better - ham "
" ad - do not "
" ga - collide "
4
5
```

```
" best - spam "
" awd - do not "
```

The above output shows how all the data records are stored at each index position in the hash table. We can observe that multiple data records are stored at the same index position given by the hash function.

Hash tables are important data structures for storing data in key-value pairs, and we can use any of the collision resolution techniques, in other words, open addressing or separate chaining. Open addressing techniques are very fast when the keys are uniformly distributed in the hash table, but there is a possible complication of cluster formation.

The separate chaining technique does not have the problem of clustering, but it may become slower when all the data records are hashed to a very few index positions in the hash table.

Symbol tables

Symbol tables are used by compilers and interpreters to keep track of the symbols and different entities, such as objects, classes, variables, and function names, that have been declared in a program. Symbol tables are often built using hash tables since it is important to efficiently retrieve a symbol from the table.

Let's look at an example. Suppose we have the following Python code in the `symb.py` file:

```
name = "Joe"  
age = 27
```

Here, we have two symbols, `name` and `age`. Each symbol has a value; for example, the `name` symbol has the value `Joe`, and the `age` symbol has the value `27`. A symbol table allows the compiler or the interpreter to look up these values. So, the `name` and `age` symbols become keys in the hash table. All of the other information associated with them becomes the `value` of the symbol table entry.

In compilers, symbol tables can have other symbols as well, such as functions and class names. For example, the `greet()` function and two variables, in other words, `name` and `age`, are stored in the symbol table as shown in *Figure 8.15*:

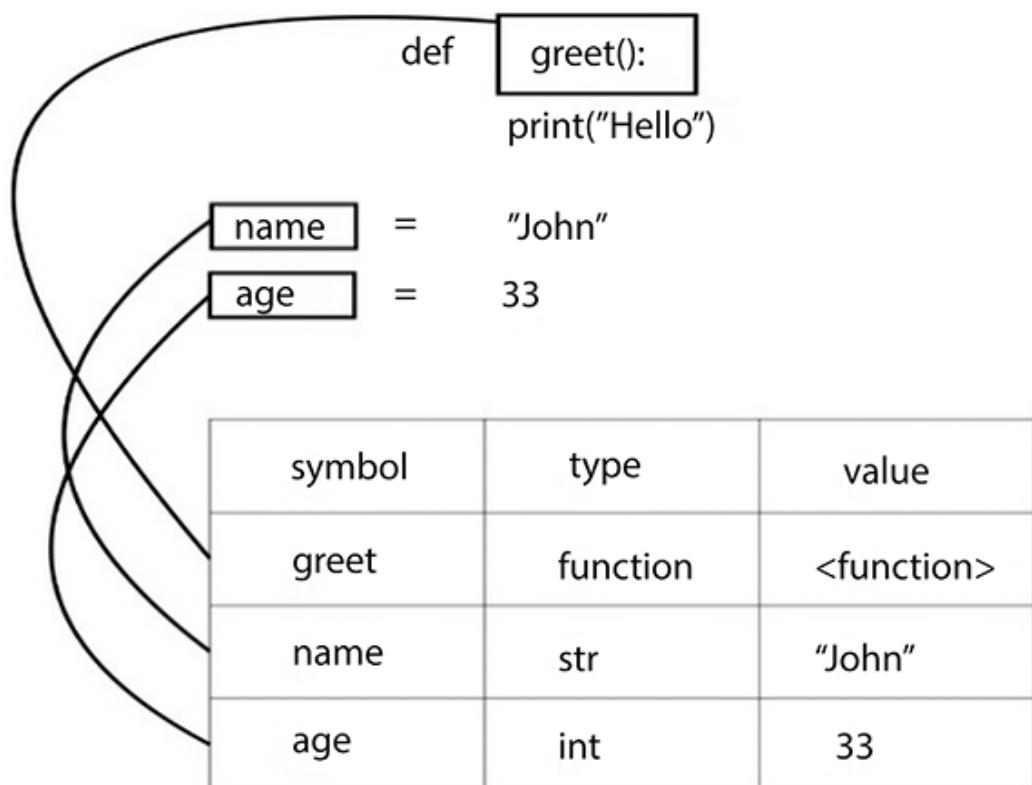


Figure 8.15: Example of a symbol table

The compiler creates a symbol table for each of its modules that are loaded in memory at the time of its execution. Symbol tables are one of the important applications of hash tables, which are mostly used in the compilers and interpreters to efficiently store and retrieve the symbols and associated values.

Summary

In this chapter, we discussed hashing techniques and the data structure of hash tables. We learned about the implementation and concepts of different operations performed on hash tables. We also discussed several collision resolution techniques, including open addressing techniques, namely, linear probing, quadratic probing, and double hashing. Furthermore, we discussed another kind of collision resolution method – separate chaining. Finally, we looked at symbol tables, which are often built using hash tables. Symbol tables allow a compiler or an interpreter to look up a symbol (such as a variable, function, or class) that has been defined and retrieve all the information about it. In the next chapter, we will discuss graph algorithms in detail.

Exercise

1. There is a hash table with 40 slots and there are 200 elements stored in the table. What will be the load factor of the hash table?

2. What is the worst-case search time of hashing using a separate chaining algorithm?
3. Assume a uniform distribution of keys in the hash table. What will be the time complexities for the Search/Insert/Delete operations?
4. What will be the worst-case complexity for removing duplicate characters from an array of characters?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



9

Graphs and Algorithms

Graphs are a non-linear data structure, in which the problem is represented as a network by connecting a set of nodes with edges, like a telephone network or social network. For example, in a graph, nodes can represent different cities while the links between them represent edges. Graphs are one of the most important data structures; they are used to solve many computing problems, especially when the problem is represented in the form of objects and their connection, e.g. to find out the shortest path from one city to another city. Graphs are useful data structures for solving real-world problems in which the problem can be represented as a network-like structure. In this chapter, we will be discussing the most important and popular concepts related to graphs.

In this chapter, we will learn about the following concepts:

- The concept of the graph data structure
- How to represent a graph and traverse it
- Different operations and their implementation on graphs

First, we will be looking into the different types of graphs.

Graphs

A graph is a set of a finite number of vertices (also known as nodes) and edges, in which the edges are the links between vertices, and each edge in a graph joins two distinct nodes. Moreover, a graph is a formal mathematical representation of a network, i.e. a graph G is an ordered pair of a set V of vertices and a set E of edges, given as $G = (V, E)$ in formal mathematical notation.

An example of a graph is shown in *Figure 9.1*:

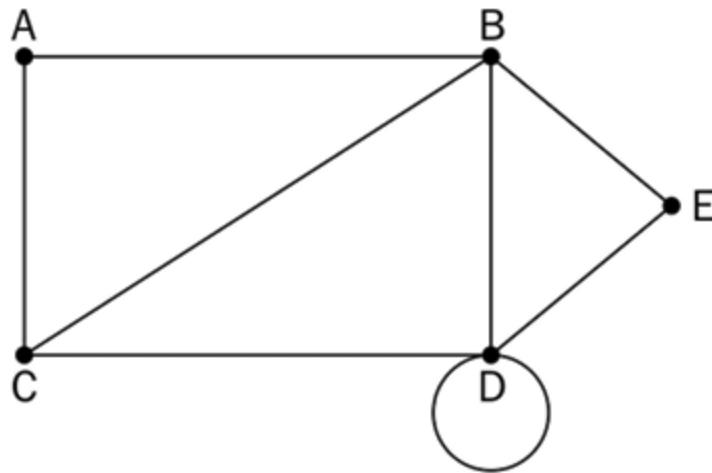


Figure 9.1: An example of a graph

The graph $G = (V, E)$ in *Figure 9.1* can be described as below:

- $V = \{A, B, C, D, E\}$
- $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}, \{C, D\}, \{D, D\}, \{B, E\}, \{D, E\}\}$
- $G = (V, E)$

Let's discuss some of the important definitions of a graph:

- **Node or vertex:** A point or node in a graph is called a vertex. In the preceding diagram, the vertices or nodes are **A, B, C, D**, and **E** and are denoted by a dot.

- **Edge:** This is a connection between two vertices. The line connecting **A** and **B** is an example of an edge.
- **Loop:** When an edge from a node is returned to itself , that edge forms a loop, e.g. **D** node.
- **Degree of a vertex/node:** The total number of edges that are incidental on a given vertex is called the degree of that vertex. For example, the degree of the **B** vertex in the previous diagram is 4.
- **Adjacency:** This refers to the connection(s) between any two nodes; thus, if there is a connection between any two vertices or nodes, then they are said to be adjacent to each other. For example, the **C** node is adjacent to the **A** node because there is an edge between them.
- **Path:** A sequence of vertices and edges between any two nodes represents a path. For example, **CABE** represents a path from the **C** node to the **E** node.
- **Leaf vertex** (also called *pendant vertex*): A vertex or node is called a leaf vertex or pendant vertex if it has exactly one degree.

Now, we shall take a look at the different types of graphs.

Directed and undirected graphs

Graphs are represented by the edges between the nodes. The connecting edges can be considered directed or undirected. If the connecting edges in a graph are undirected, then the graph is called an undirected graph, and if the connecting edges in a graph are directed, then it is called a directed graph. An undirected graph

simply represents edges as lines between the nodes. There is no additional information about the relationship between the nodes, other than the fact that they are connected. For example, in *Figure 9.2*, we demonstrate an undirected graph of four nodes, **A**, **B**, **C**, and **D**, which are connected using edges:

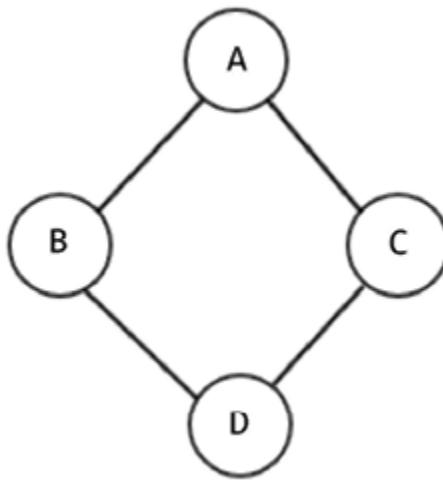


Figure 9.2: An example of an undirected graph

In a directed graph, the edges provide information on the direction of connection between any two nodes in a graph. If an edge from **A** node to **B** is said to be directed, then the edge (**A**, **B**) would not be equal to the edge (**B**, **A**). The directed edges are drawn as lines with arrows, which will point in whichever direction the edge connects the two nodes.

For example, in *Figure 9.3*, we show a directed graph where many nodes are connected using directed edges:

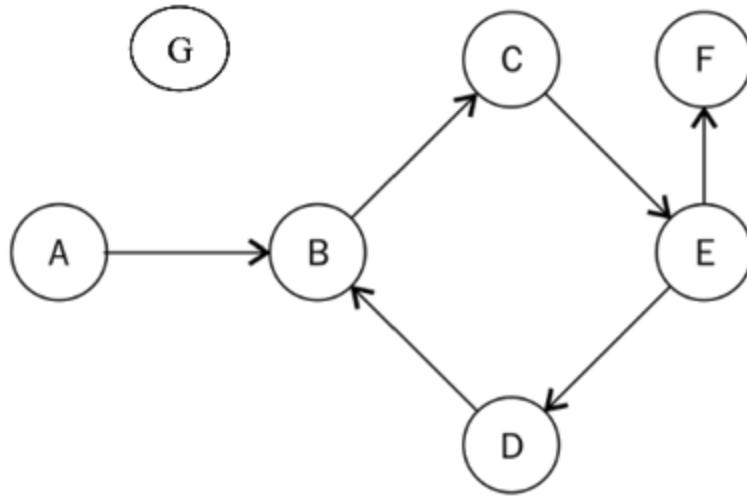


Figure 9.3: An example of a directed graph

The arrow of an edge determines the flow of direction. One can only move from **A** to **B**, as shown in the preceding diagram—not **B** to **A**. In a directed graph, each node (or vertex) has an indegree and an outdegree. Let's have a look at what these are:

- **Indegree:** The total number of edges that come into a vertex in the graph is called the indegree of that vertex. For example, in the previous diagram, the **E** node has 1 indegree, due to edge **CE** coming into the **E** node.
- **Outdegree:** The total number of edges that go out from a vertex in the graph is called the outdegree of that vertex. For example, the **E** node in the previous diagram has an outdegree of 2, as it has two edges, **EF** and **ED**, going out of that node.
- **Isolated vertex:** A node or vertex is called an isolated vertex when it has a degree of zero, as shown as **G** node in *Figure 9.3*.
- **Source vertex:** A vertex is called a source vertex if it has an indegree of zero. For example, in the previous diagram, the **A**

node is the source vertex.

- **Sink vertex:** A vertex is a sink vertex if it has an outdegree of zero. For example, in the previous diagram, the F node is the sink vertex.

Now that we understand how directed graphs work, we can look into directed acyclic graphs.

Directed acyclic graphs

A **directed acyclic graph (DAG)** is a directed graph with no cycles; in a DAG all the edges are directed from one node to another node so that the sequence of edges never forms a closed loop. A cycle in a graph is formed when the starting node of the first edge is equal to the ending node of the last edge in a sequence.

A DAG is shown in *Figure 9.4* in which all the edges in the graph are directed and the graph does not have any cycles:

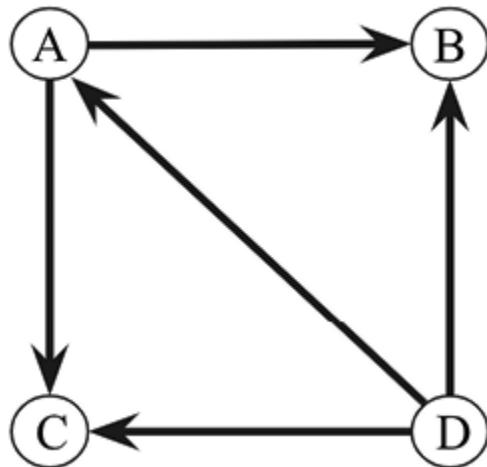


Figure 9.4: An example of a directed acyclic graph

So, in a directed acyclic graph, if we start on any path from a given node, we never find a path that ends on the same node. A DAG has many applications, such as in job scheduling, citation graphs, and data compression.

Next, we will discuss weighted graphs.

Weighted graphs

A weighted graph is a graph that has a numeric weight associated with the edges in the graph. A weighted graph can be either a directed or an undirected graph. The numeric weight can be used to indicate distance or cost, depending on the purpose of the graph:

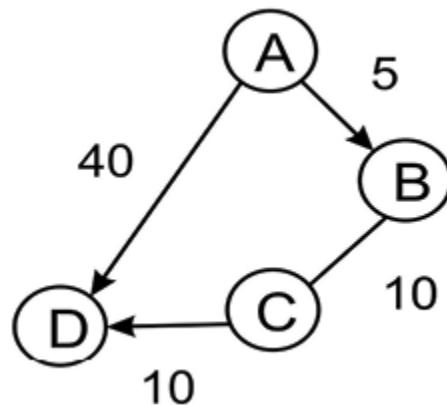


Figure 9.5: An example of a weighted graph

Let's consider an example – *Figure 9.5* indicates different ways to reach from **A** node to **D** node. There are two possible paths, such as from **A** node to **D** node, or it can be nodes **A-B-C-D** through **B** node and **C** node. Now, depending on the weights associated with the edges, any one of the paths can be considered better than the others for the journey – e.g. assume the weights in this graph represent the

distance between two nodes, and we want to find out the shortest path between **A-D** nodes; then one possible path **A-D** has an associated cost of 40, and another possible path **A-B-C-D** has an associated cost of 25. In this case, the better path is **A-B-C-D**, which has a lower distance.

Next, we will discuss bipartite graphs.

Bipartite graphs

A bipartite graph (also known as a bigraph) is a special graph in which all the nodes of the graph can be divided into two sets in such a way that edges connect the nodes from one set to the nodes of another set. See *Figure 9.6* for a sample bipartite graph; all the nodes of the graphs are divided into two independent sets, i.e., set U and set V, so that each edge in the graph has one end in set U and another end in set V (e.g. in edge (A, B) , one end or one vertex is from set U, and another end or another vertex is from set V).

In bipartite graphs, no edge will connect to the nodes of the same set:

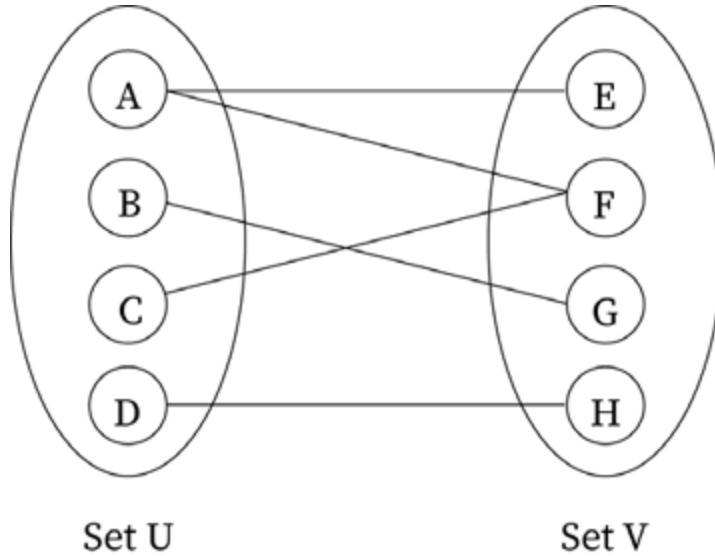


Figure 9.6: An example of a bipartite graph

Bipartite graphs are useful when we need to model a relationship between two different classes of objects, for example, a graph of applicants and jobs, in which we may need to model the relationship between these two different groups; another example may be a bipartite graph of football players and clubs in which we may need to model if a player has played for a particular club or not.

Next, we will discuss different graph representation techniques.

Graph representations

A graph representation technique means how we store the graph in memory, i.e., how we store the vertices, edges, and weights (if the graph is a weighted graph). Graphs can be represented with two methods, i.e. (1) an adjacency list, and (2) an adjacency matrix.

An adjacency list representation is based on a linked list. In this, we represent the graph by maintaining a list of neighbors (also called an

adjacent node) for every vertex (or node) of the graph. In an adjacency matrix representation of a graph, we maintain a matrix that represents which node is adjacent to which other node in the graph; i.e., the adjacency matrix has the information of every edge in the graph, which is represented by cells of the matrix.

Either of these two representations can be used; however, our choice depends on the application where we will be using the graph representation. An adjacency list is preferable when we expect that the graph is going to be sparse and we will have a smaller number of edges; e.g. if a graph of 200 nodes has say 100 edges, it is better to store this kind of graph in an adjacency list, because if we use an adjacency matrix, the size of the matrix will be 200x200 with a lot of zero values. The adjacency matrix is preferable when we expect the graph to have a lot of edges, and the matrix will be dense. In the adjacency matrix, the lookup and check for the presence or absence of an edge are very easy compared to adjacency list representation.

We will be discussing adjacency matrices in detail in subsequent sections. First, we will take a look at adjacency lists.

Adjacency lists

In this representation, all the nodes directly connected to a node x are listed in its adjacent list of nodes. The graph is represented by displaying the adjacent list for all the nodes of the graph.

Two nodes, **A** and **B**, in the graph shown in *Figure 9.7*, are said to be adjacent if there is a direct connection between them:

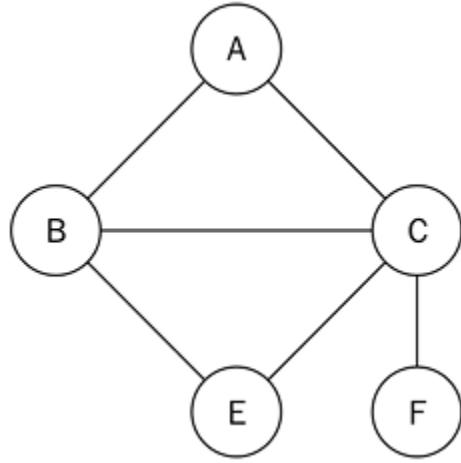


Figure 9.7: A sample graph of five nodes

A linked list can be used to implement the adjacency list. In order to represent the graph, we need the number of linked lists equal to the total number of nodes in the graph. At each index, the adjacent nodes to that vertex are stored. For example, consider the adjacency list shown in *Figure 9.8* corresponding to the sample graph shown in *Figure 9.7*:

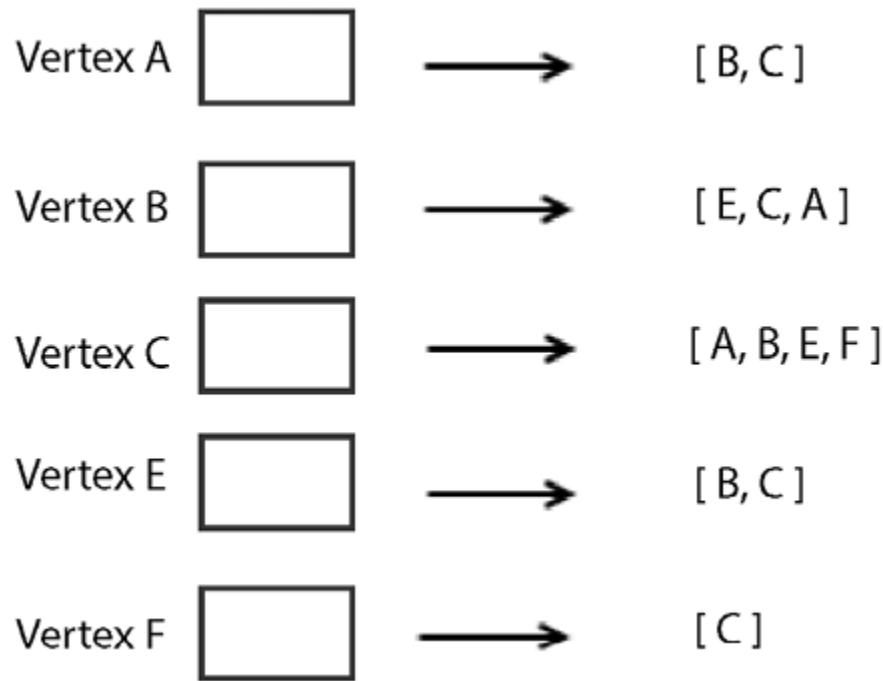


Figure 9.8: Adjacency list for the graph shown in Figure 9.7

Here, the first node represents the `A` vertex of the graph, with its adjacent nodes being `B` and `C`. The second node represents the `B` vertex of the graph, with its adjacent nodes of `E`, `C`, and `A`. Similarly, the other vertices, `C`, `E`, and `F`, of the graph are represented with their adjacent nodes, as shown in the previous *Figure 9.8*.

Using a `list` for the representation is quite restrictive, because we lack the ability to directly use the vertex labels. So, to implement a graph efficiently using Python, a `dictionary` data structure is used since it is more suitable to represent the graph. To implement the same graph using a dictionary data structure, we can use the following code snippet:

```
graph = dict()
graph['A'] = ['B', 'C']
graph['B'] = ['E', 'C', 'A']
```

```
graph['C'] = ['A', 'B', 'E', 'F']
graph['E'] = ['B', 'C']
graph['F'] = ['C']
```

Now we can easily establish that the `A` vertex has the adjacent vertices of `B` and `C`. The `F` vertex has the `C` vertex as its only neighbor. Similarly, the `B` vertex has adjacent vertices of `E`, `C`, and `A`.

The adjacency list is a preferable graph representation technique when the graph is going to be sparse and we may need to add or delete the nodes in the graph frequently. However, it is very difficult to check whether a given edge is present in the graph or not using this technique.

Next, we will discuss another method of graph representation, i.e., the adjacency matrix.

Adjacency matrix

Another approach to representing a graph is to use an adjacency matrix. In this, the graph is represented by showing the nodes and their interconnections through edges. Using this method, the dimensions ($v \times v$) of a matrix are used to represent the graph, where each cell denotes an edge in the graph. A matrix is a two-dimensional array. So, the idea here is to represent the cells of the matrix with a `1` or a `0`, depending on whether two nodes are connected by an edge or not. We show an example graph, along with its corresponding adjacency matrix, in *Figure 9.9*:

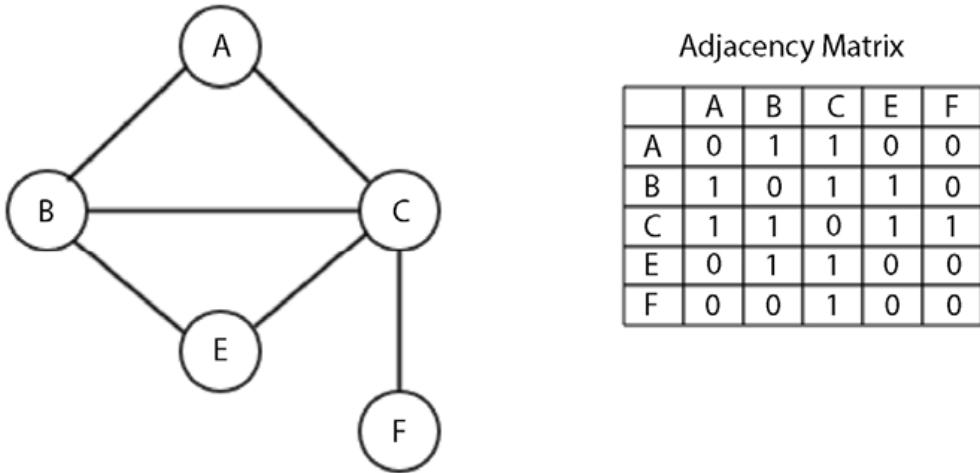


Figure 9.9: Adjacency matrix for a given graph

An adjacency matrix can be implemented using the given adjacency list. To implement the adjacency matrix, let's take the previous dictionary-based implementation of the graph. Firstly, we have to obtain the key elements of the adjacency matrix. It is important to note that these matrix elements are the vertices of the graph. We can get the key elements by sorting the keys of the graph. The code snippet for this is as follows:

```
matrix_elements = sorted(graph.keys())
cols = rows = len(matrix_elements)
```

Next, the length of the keys of the graph will be the dimensions of the adjacency matrix, which are stored in `cols` and `rows`. The values of the `cols` and `rows` are equal.

So, now, we create an empty adjacency matrix of the dimensions `cols` by `rows`, initially filling all the values with zeros. The code snippet to initialize an empty adjacency matrix is as follows:

```
adjacency_matrix = [[0 for x in range(rows)] for y in range(cols)]
edges_list = []
```

The `edges_list` variable will store the tuples that form the edges in the graph. For example, an edge between the A and B nodes will be stored as (A, B). The multidimensional array is filled using a nested `for` loop:

```
for key in matrix_elements:
    for neighbor in graph[key]:
        edges_list.append((key, neighbor))
print(edges_list)
```

The neighbors of a vertex are obtained by `graph[key]`. The key, in combination with the `neighbor`, is then used to create the tuple stored in `edges_list`.

The output of the preceding Python code for storing the edges of the graph is as follows:

```
[('A', 'B'), ('A', 'C'), ('B', 'E'), ('B', 'C'), ('B', 'A'), ('C', 'A')]
```

The next step in implementing the adjacency matrix is to fill it, using 1 to denote the presence of an edge in the graph. This can be done with the `adjacency_matrix[index_of_first_vertex][index_of_second_vertex] = 1` statement. The full code snippet that marks the presence of edges of the graph is as follows:

```
for edge in edges_list:
    index_of_first_vertex = matrix_elements.index(edge[0])
    index_of_second_vertex = matrix_elements.index(edge[1])
```

```
adjacency_matrix[index_of_first_vertex][index_of_second_vertex] =  
print(adjacency_matrix)
```

The `matrix_elements` array has its `rows` and `cols`, starting from `A` to all other vertices with indices of `0` to `5`. The `for` loop iterates through the list of tuples and uses the `index` method to get the corresponding index where an edge is to be stored.

The output of the preceding code is the adjacency matrix for the sample graph shown previously in *Figure 9.9*. The adjacency matrix produced looks like the following:

```
[0, 1, 1, 0, 0]  
[1, 0, 0, 1, 0]  
[1, 1, 0, 1, 1]  
[0, 1, 1, 0, 0]  
[0, 0, 1, 0, 0]
```

At row `1` and column `1`, `0` represents the absence of an edge between `A` and `A`. Similarly, at row `3` and column `2` there is a value of `1` that denotes the edge between the `C` and `B` vertices in the graph.

The use of the adjacency matrix for graph representation is suitable when we have to frequently look up and check the presence or absence of an edge between two nodes in the graph, e.g. in creating routing tables in networks, searching routes in public transport applications and navigation systems, etc. Adjacency matrices are not suitable when nodes are frequently added or deleted within a graph, in those situations, the adjacency list is a better technique.

Next, let us discuss different graph traversal methods in which we visit all the nodes of the given graph.

Graph traversals

A graph traversal means to visit all the vertices of the graph while keeping track of which nodes or vertices have already been visited and which ones have not. A graph traversal algorithm is efficient if it traverses all the nodes of the graph in the minimum possible time.

Graph traversal, also known as a graph search algorithm, is quite similar to the tree traversal algorithms like `preorder`, `inorder`, `postorder`, and level order algorithms; similar to them, in a graph search algorithm we start with a node and traverse through edges to all other nodes in the graph.

A common strategy of graph traversal is to follow a path until a dead end is reached, then traverse back up until there is a point where we meet an alternative path. We can also iteratively move from one node to another in order to traverse the full graph or part of it. Graph traversal algorithms are very important in answering many fundamental problems—they can be useful to determine how to get from one vertex to another in a graph, and which path from **A** node to **B** node in a graph is better than other paths. For example, graph traversal algorithms can be useful in finding out the shortest route from one city to another in a network of cities.

In the next section, we will discuss two important graph traversal algorithms: **breadth-first search (BFS)** and **depth-first search (DFS)**.

Breadth-first traversal

Breadth-first search (BFS) works very similarly to how a level order traversal algorithm works in a tree data structure. The BFS algorithm

also works level by level; it starts by visiting the root node at level 0, and then all the nodes at the first level directly connected to the root node are visited at level 1. The level 1 node has a distance of 1 from the root node. After visiting all the nodes at level 1, the level 2 nodes are visited next. Likewise, all the nodes in the graph are traversed level by level until all the nodes are visited. So, breadth-first traversal algorithms work breadthwise in the graph.

A queue data structure is used to store the information of vertices that are to be visited in a graph. We begin with the starting node. Firstly, we visit that node, and then we look up all of its neighboring, or adjacent, vertices. We first visit these adjacent vertices one by one, while adding their neighbors to the list of vertices that are to be visited. We follow this process until we have visited all the vertices of the graph, ensuring that no vertex is visited twice.

Let's consider an example to better understand the working of the breadth-first traversal for graphs, using the sample shown in *Figure 9.10*:

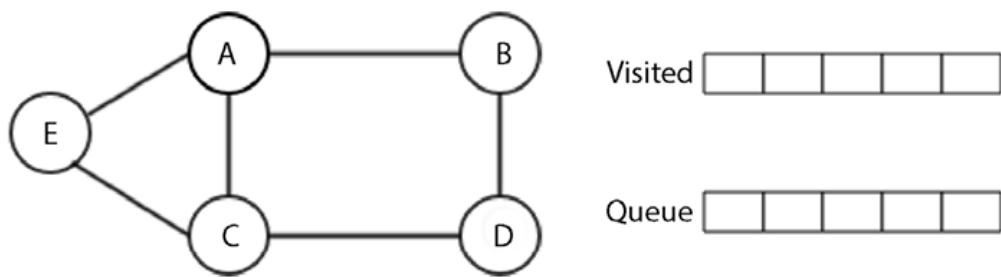


Figure 9.10: A sample graph

In *Figure 9.10*, we have a graph of five nodes on the left, and on the right, a queue data structure to store the vertices to be visited. We start visiting the first node, i.e., A node, and then we add all its

adjacent vertices, **B**, **C**, and **E**, to the queue. Here, it is important to note that there are multiple ways of adding the adjacent nodes to the queue since there are three nodes, **B**, **C**, and **E**, that can be added to the queue as either **BCE**, **CEB**, **CBE**, **BEC**, or **ECB**, each of which would give us different tree traversal results.

All of these possible solutions to the graph traversal are correct, but in this example, we add the nodes in alphabetical order just to keep things simple in the queue, i.e., **BCE**. The **A** node is visited as shown in *Figure 9.11*:

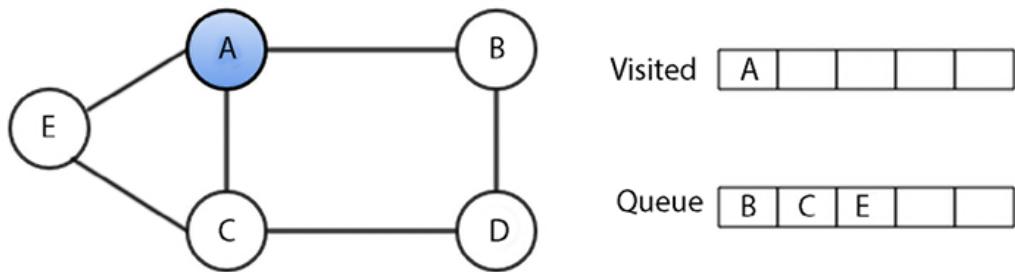


Figure 9.11: Node A is visited in breadth-first traversal

Once we have visited the **A** vertex, next, we visit its first adjacent vertex, **B**, and add those adjacent vertices of vertex **B** that are not already added in the queue or not visited. In this case, we have to add the **D** vertex (since it has two vertices, **A** and **D** nodes, out of which **A** is already visited) to the queue, as shown in *Figure 9.12*:

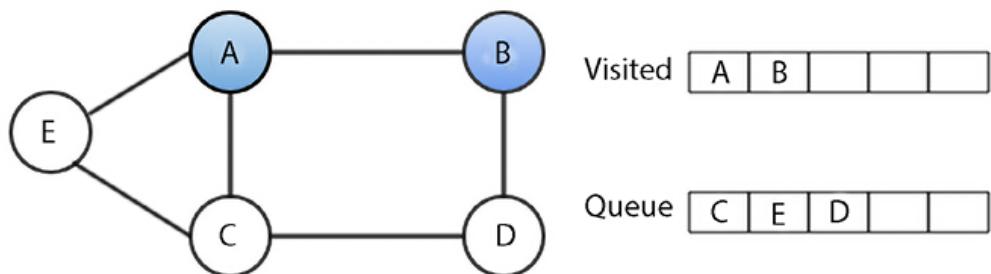


Figure 9.12: Node B is visited in breadth-first traversal

Now, after visiting the **B** vertex, we visit the next vertex from the queue—the **C** vertex. And again, add those adjacent vertices that have not already been added to the queue. In this case, there are no unrecorded vertices left, as shown in *Figure 9.13*:

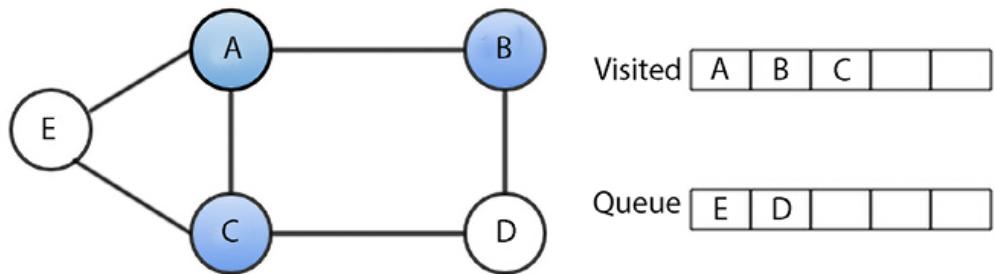


Figure 9.13: Node C is visited in breadth-first traversal

After visiting the **C** vertex, we visit the next vertex from the queue, the **E** vertex, as shown in *Figure 9.14*:

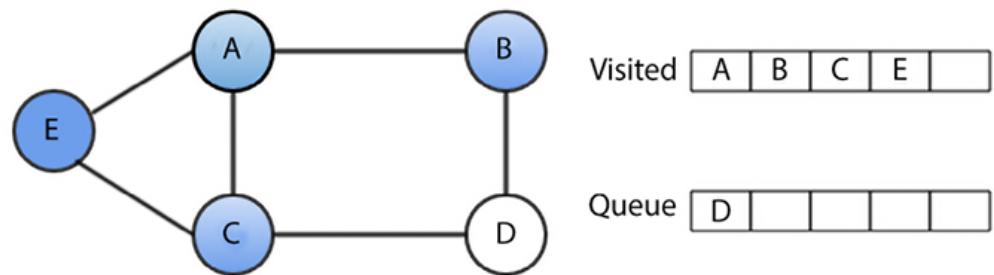


Figure 9.14: Node E is visited in breadth-first traversal

Similarly, after visiting the **E** vertex, we visit the **D** vertex in the last step, as shown in *Figure 9.15*:

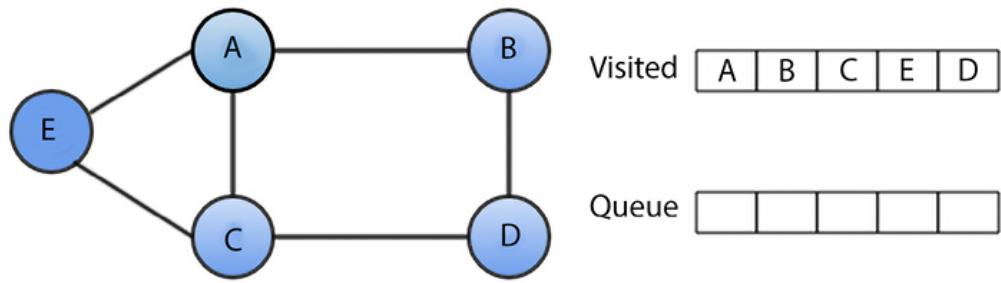


Figure 9.15: D node is visited in breadth-first traversal

Therefore, the BFS algorithm for traversing the preceding graph visits the vertices in the order of **A-B-C-E-D**. This is one of the possible solutions to the BFS traversal for the preceding graph, but we can get many possible solutions, depending on how we add the adjacent nodes to the queue.

To understand the implementation of this algorithm in Python, we will use another example of an undirected graph, as shown in *Figure 9.16*:

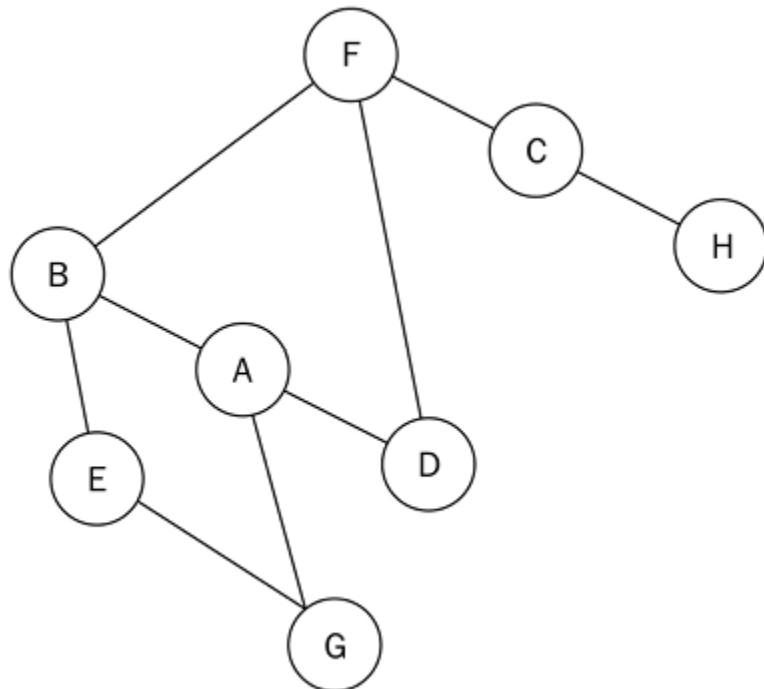


Figure 9.16: An undirected sample graph

The adjacency list for the graph shown in *Figure 9.16* is as follows:

```
graph = dict()
graph['A'] = ['B', 'G', 'D']
graph['B'] = ['A', 'F', 'E']
graph['C'] = ['F', 'H']
graph['D'] = ['F', 'A']
graph['E'] = ['B', 'G']
graph['F'] = ['B', 'D', 'C']
graph['G'] = ['A', 'E']
graph['H'] = ['C']
```

After storing the graph using the adjacency list, the implementation of the BFS algorithm is as follows, which we will discuss with an example in detail:

```
from collections import deque
def breadth_first_search(graph, root):
    visited_vertices = []
    graph_queue = deque([root])
    visited_vertices.append(root)
    node = root
    while len(graph_queue) > 0:
        node = graph_queue.popleft()
        adj_nodes = graph[node]
        remaining_elements = set(adj_nodes).difference(set(visited_vertices))
        if len(remaining_elements) > 0:
            for elem in sorted(remaining_elements):
                visited_vertices.append(elem)
                graph_queue.append(elem)
    return visited_vertices
```

To traverse this graph using the breadth-first algorithm, we first initialize the queue and the source node. We start traversal from A

node. Firstly, **A** node is queued and added to the list of visited nodes. Afterward, we use a `while` loop to affect the traversal of the graph. In the first iteration of the `while` loop, node **A** is dequeued.

Next, all the unvisited adjacent nodes of **A** node, which are **B**, **D**, and **G**, are sorted in alphabetical order and queued up. The queue now contains nodes **B**, **D**, and **G**. This is shown in *Figure 9.17*:

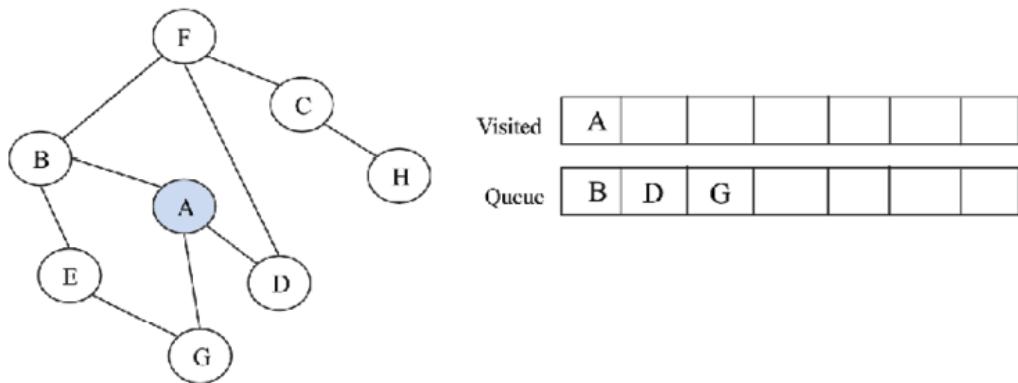


Figure 9.17: Node A is visited using the BFS algorithm

For implementation, we add all these nodes (**B**, **D**, **G**) to the list of visited nodes, and then we add the adjacent/neighboring nodes of these nodes. At this point, we start another iteration of the `while` loop. After visiting **A** node, **B** node is dequeued. Out of its adjacent nodes (**A**, **E**, and **F**), **A** node has already been visited. Therefore, we only queue the **E** and **F** nodes in alphabetical order, as shown in *Figure 9.18*.

When we want to find out whether a set of nodes is in the list of visited nodes, we use the `remaining_elements = set(adj_nodes).difference(set(visited_vertices))` statement. This uses the `set` object's `difference` method to find the nodes that are in `adj_nodes`, but not in `visited_vertices`:

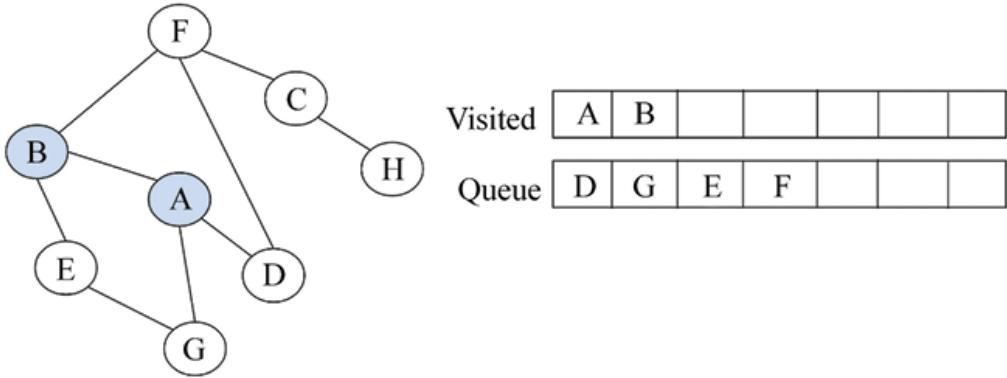


Figure 9.18: Node B is visited using the BFS algorithm

The queue now holds the following nodes at this point—D, G, E, and F. The D node is dequeued, but all of its adjacent nodes have been visited, so we simply dequeue it. The next node at the front of the queue is G. We dequeue the G node, but we also find out that all its adjacent nodes have been visited because they are in the list of visited nodes. So, the G node is also dequeued. We dequeue the E node too because all of its adjacent nodes have also been visited. The only node in the queue now is the F node; this is shown in *Figure 9.19*:

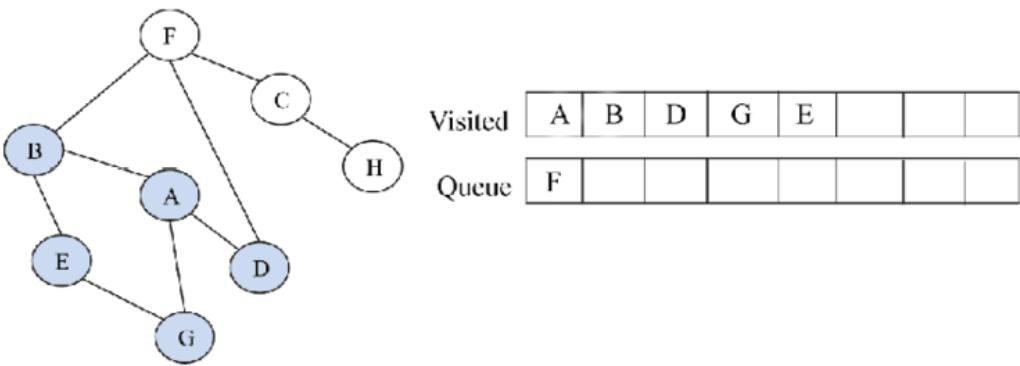


Figure 9.19: Node E is visited using the BFS algorithm

The F node is dequeued, and we see that out of its adjacent nodes, B, D, and C, only C has not been visited. We then enqueue the C node

and add it to the list of visited nodes, as shown in *Figure 9.20*:

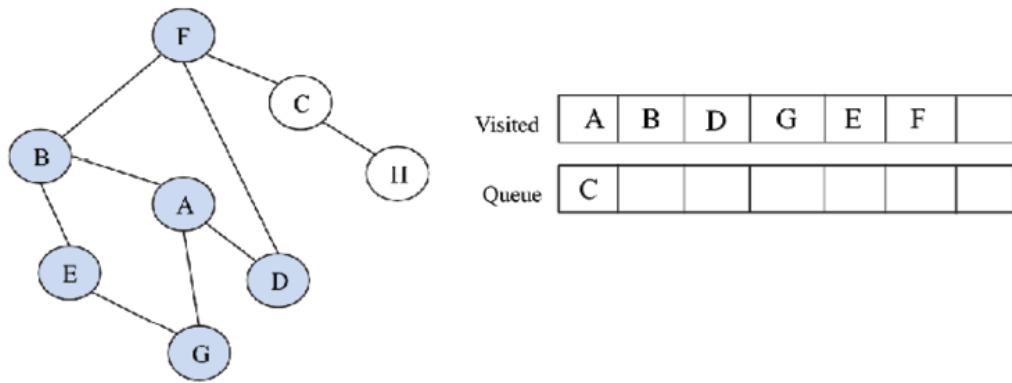


Figure 9.20: Node E is visited using the BFS algorithm

Then, the **C** node is dequeued. **C** has the adjacent nodes of **F** and **H**, but **F** has already been visited, leaving the **H** node. The **H** node is enqueueued and added to the list of visited nodes. Finally, the last iteration of the `while` loop will lead to the **H** node being dequeued.

Its only adjacent node, **C**, has already been visited. Once the queue is empty, the loop breaks. This is shown in *Figure 9.21*:

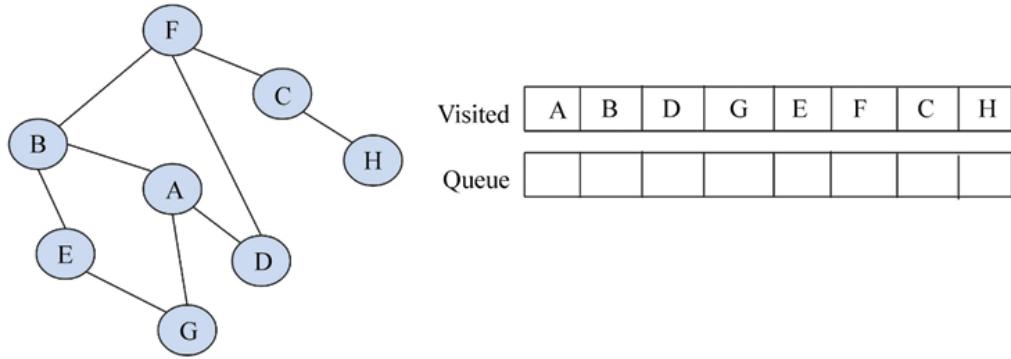


Figure 9.21: Final node H is visited using the BFS algorithm

The output of the traversal of the given graph using the BFS algorithm is **A, B, D, G, E, F, C, and H**.

When we run the above BFS code on the graph shown in *Figure 9.16* using the following code:

```
print(breadth_first_search(graph, 'A'))
```

We get the following sequence of nodes when we traverse the graph shown in *Figure 9.16*:

```
['A', 'B', 'D', 'G', 'E', 'F', 'C', 'H']
```

In the worst-case scenario, each node and the edge will need to be traversed, and hence each node will be enqueued and dequeued at least once. The time taken for each enqueue and dequeue operation is $O(1)$, so the total time for this is $O(V)$. Further, the time spent scanning the adjacency list for every vertex is $O(E)$. So, the total time complexity of the BFS algorithm is $O(|V| + |E|)$, where $|V|$ is the number of vertices or nodes, while $|E|$ is the number of edges in the graph.

The BFS algorithm is very useful for constructing the shortest path traversal in a graph with minimal iterations. As for some of the real-world applications of BFS, it can be used to create an efficient web crawler in which multiple levels of indexes can be maintained for search engines, and it can maintain a list of closed web pages from a source web page. BFS can also be useful for navigation systems in which neighboring locations can be easily retrieved from a graph of different locations.

Next, we will discuss another graph traversal algorithm, i.e., the depth-first search algorithm.

Depth-first search

As the name suggests, the **depth-first search (DFS)** or traversal algorithm traverses the graph similar to how the `preorder` traversal algorithm works in trees. In the DFS algorithm, we traverse the tree in the depth of any particular path in the graph. As such, child nodes are visited first before sibling nodes.

In this, we start with the root node; firstly we visit it, and then we see all the adjacent vertices of the current node. We start visiting one of the adjacent nodes. If the edge leads to a visited node, we backtrack to the current node. And, if the edge leads to an unvisited node, then we go to that node and continue processing from that node. We continue the same process until we reach a dead end when there is no unvisited node; in that case, we backtrack to previous nodes, and we stop when we reach the root node while backtracking.

Let's take an example to understand the working of the DFS algorithm using the graph shown in *Figure 9.22*:

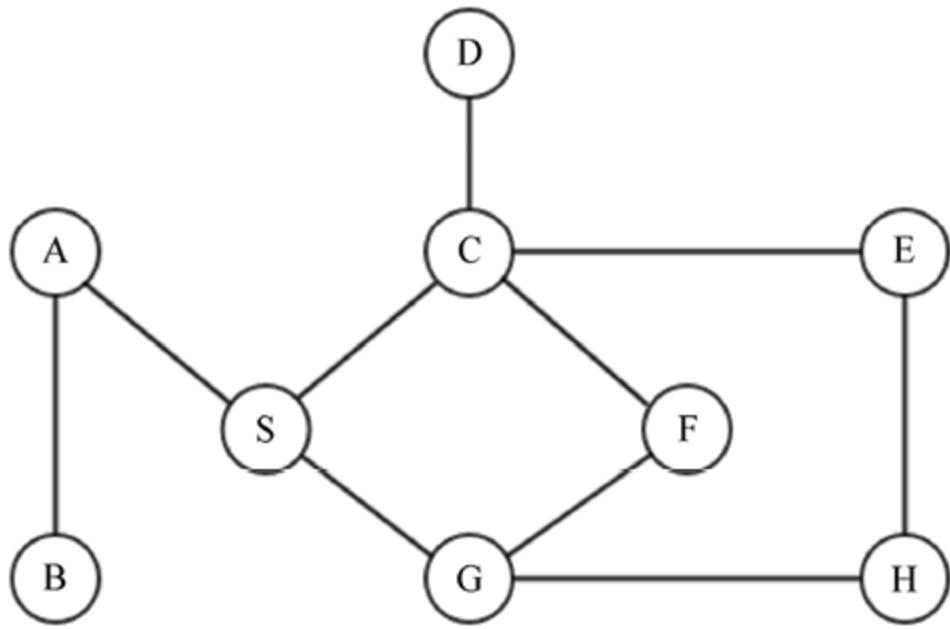


Figure 9.22: An example graph for understanding the DFS algorithm

We start by visiting the **A** node, and then we look at the neighbors of the **A** vertex, then a neighbor of that neighbor, and so on. After visiting the **A** vertex, we visit one of its neighbors, **B** (in our example, we sort alphabetically; however, any neighbor can be added), as shown in *Figure 9.23*:

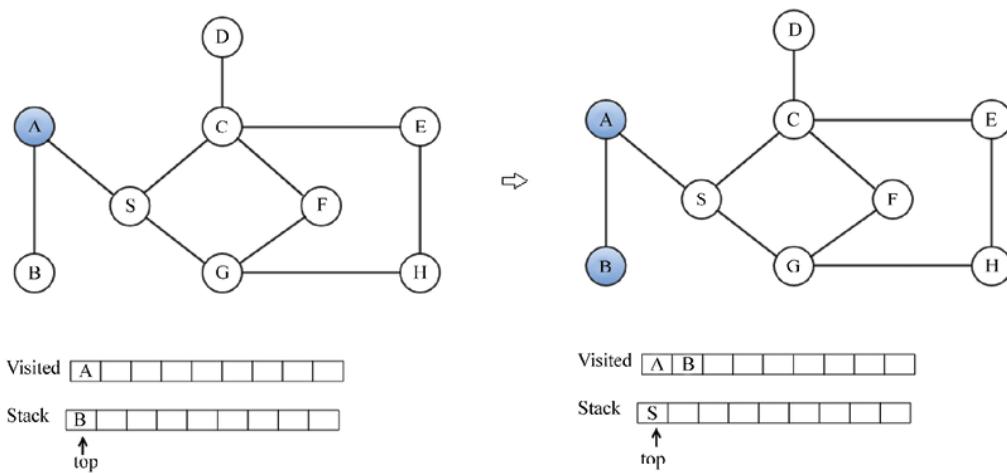


Figure 9.23: Nodes **A** and **B** are visited in depth-first traversal

After visiting the **B** vertex, we look at another neighbor of **A**, that is, **S**, as there is no vertex connected to **B** that can be visited. Next, we look for the neighbors of the **S** vertex, which are the **C** and **G** vertices. We visit **C** as shown in *Figure 9.24*:

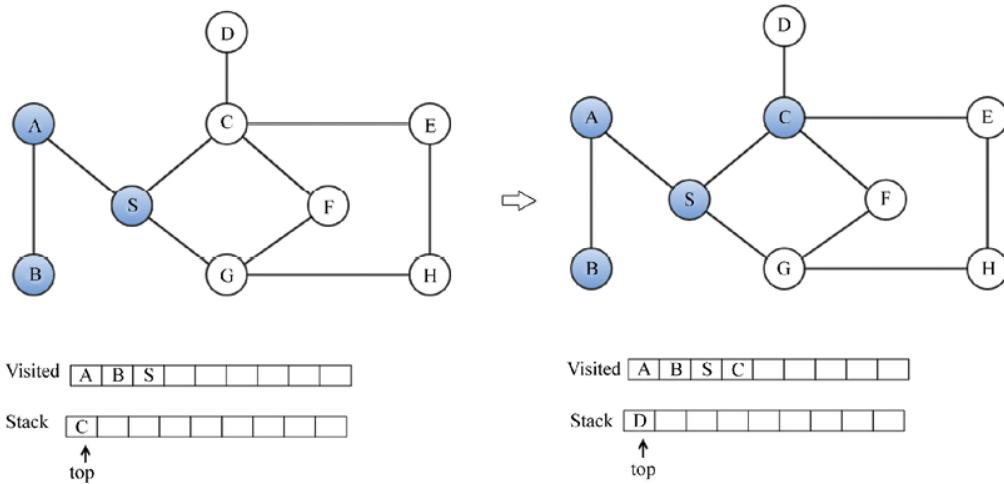


Figure 9.24: Node C is visited in depth-first traversal

After visiting the **C** node, we visit its neighboring vertices, **D** and **E**, as shown in *Figure 9.25*:

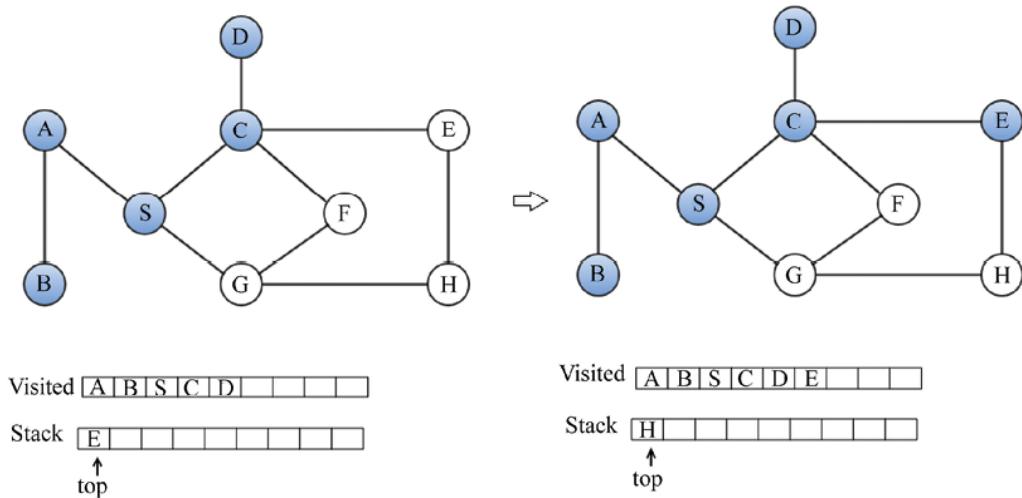


Figure 9.25: D and E nodes are visited in depth-first traversal

Similarly, after visiting the **E** vertex, we visit the **H** and **G** vertices, as shown in *Figure 9.26*:

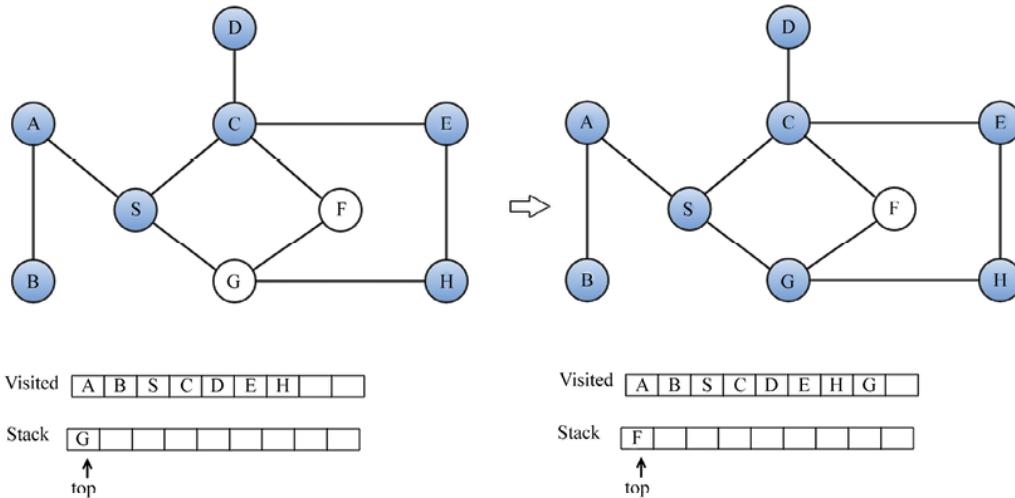
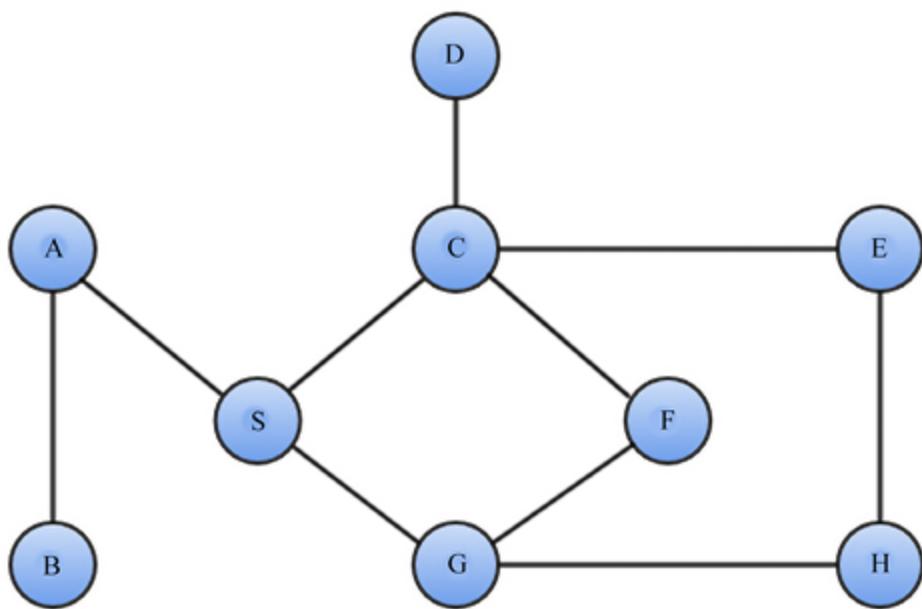


Figure 9.26: H and F nodes are visited in depth-first traversal

Finally, we visit the **F** node, as shown in *Figure 9.27*:



Visited

A	B	S	C	D	E	H	G	F
---	---	---	---	---	---	---	---	---

Stack

--	--	--	--	--	--	--	--	--

 ↑
 top

Figure 9.27: F node is visited in depth-first traversal

The output of the DFS traversal is **A-B-S-C-D-E-H-G-F**.

To implement DFS, we start with the adjacency list of the given graph. Here is the adjacency list of the preceding graph:

```
graph = dict()
graph['A'] = ['B', 'S']
graph['B'] = ['A']
graph['S'] = ['A', 'G', 'C']
graph['D'] = ['C']
graph['G'] = ['S', 'F', 'H']
graph['H'] = ['G', 'E']
graph['E'] = ['C', 'H']
```

```
graph['F'] = ['C', 'G']
graph['C'] = ['D', 'S', 'E', 'F']
```

The implementation of the DFS algorithm begins with creating a list to store the visited nodes. The `graph_stack` stack variable is used to aid the traversal process. We are using a Python list as a stack.

The starting node, called `root`, is passed with the graph's adjacency matrix, `graph`. Firstly, the `root` is pushed onto the stack. The statement `node = root` is for holding the first node in the stack:

```
def depth_first_search(graph, root):
    visited_vertices = list()
    graph_stack = list()
    graph_stack.append(root)
    node = root
    while graph_stack:
        if node not in visited_vertices:
            visited_vertices.append(node)
            adj_nodes = graph[node]
            if set(adj_nodes).issubset(set(visited_vertices)):
                graph_stack.pop()
                if len(graph_stack) > 0:
                    node = graph_stack[-1]
                    continue
            else:
                remaining_elements = set(adj_nodes).difference(set(visited_vertices))
                first_adj_node = sorted(remaining_elements)[0]
                graph_stack.append(first_adj_node)
                node = first_adj_node
    return visited_vertices
```

The body of the `while` loop will be executed, provided the stack is not empty. If the `node` under consideration is not in the list of visited nodes, we add it. All adjacent nodes of `node` are collected by `adj_nodes`

`= graph[node]`. If all the adjacent nodes have been visited, we pop the top node from the stack and set `node` to `graph_stack[-1]`. Here, `graph_stack[-1]` is the top node on the stack. The `continue` statement jumps back to the beginning of the `while` loop's test condition.

If, on the other hand, not all the adjacent nodes have been visited, then the nodes that are yet to be visited are obtained by finding the difference between the `adj_nodes` and `visited_vertices` with the `remaining_elements = set(adj_nodes).difference(set(visited_vertices))` statement.

The first item within `sorted(remaining_elements)` is assigned to `first_adj_node`, and pushed onto the stack. We then point the top of the stack to this node.

When the `while` loop exits, we will return `visited_vertices`.

We will now explain the working of the source code by relating it to the previous example. The **A** node is chosen as our starting node. **A** is pushed onto the stack and added to the `visited_vertices` list. In doing so, we mark it as having been visited. The `graph_stack` stack is implemented with a simple Python list. Our stack now has **A** as its only element. We examine the **A** node's adjacent nodes, **B** and **S**. To test whether all the adjacent nodes of **A** have been visited, we use the `if` statement:

```
if set(adj_nodes).issubset(set(visited_vertices)):
    graph_stack.pop()
    if len(graph_stack) > 0:
        node = graph_stack[-1]
    continue
```

If all the nodes have been visited, we pop the top of the stack. If the `graph_stack` stack is not empty, we assign the node on top of the stack to `node`, and start the beginning of another execution of the body of the `while` loop. The `set(adj_nodes).issubset(set(visited_vertices))` statement will evaluate to `True` if all the nodes in `adj_nodes` are a subset of `visited_vertices`. If the `if` statement fails, it means that some nodes remain to be visited. We obtain that list of nodes with

```
remaining_elements = set(adj_nodes).difference(set(visited_vertices)).
```

Referring to the diagram, the **B** and **S** nodes will be stored in `remaining_elements`. We will access the list in alphabetical order as follows:

```
first_adj_node = sorted(remaining_elements)[0]
graph_stack.append(first_adj_node)
node = first_adj_node
```

We sort `remaining_elements` and return the first node to `first_adj_node`. This will return **B**. We push the **B** node onto the stack by appending it to the `graph_stack`. We prepare the **B** node for access by assigning it to `node`.

On the next iteration of the `while` loop, we add the **B** node to the list of `visited_nodes`. We discover that the only adjacent node to **B**, which is **A**, has already been visited. Because all the adjacent nodes of **B** have been visited, we pop it off the stack, leaving **A** as the only element on the stack. We return to **A** and examine whether all of its adjacent nodes have been visited. The **A** node now has **S** as the only unvisited node. We push **S** to the stack and begin the whole process again.

The output of the traversal is A-B-S-C-D-E-H-G-F.

The time complexity of DFS is $O(V+E)$ when we use an adjacency list, and $O(V^2)$ when we use an adjacency matrix for graph representation. The time complexity of DFS with the adjacency list is lower because getting the adjacent nodes is easier, whereas it is not efficient with the adjacency matrix.

DFS can be applied to solving maze problems, finding connected components, cycle detection in graphs, and finding the bridges of a graph, among other use cases.

We have discussed very important graph traversal algorithms; now let us discuss some more useful graph-related algorithms for finding the spanning tree from the given graph. Spanning trees are useful for several real-world problems such as the traveling salesman problem.

Other useful graph methods

It is very often that we need to use graphs for finding a path between two nodes. Sometimes, it is necessary to find all the paths between nodes, and in some situations, we might need to find the shortest path between nodes. For example, in routing applications, we generally use various algorithms to determine the shortest path from the source node to the destination node. For an unweighted graph, we would simply determine the path with the lowest number of edges between them. If a weighted graph is given, we have to calculate the total weight of passing through a set of edges.

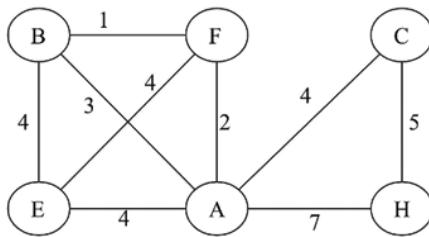
Thus, in a different situation, we may have to find the longest or shortest path using different algorithms, such as a **Minimum**

Spanning Tree, which we look into in the next section.

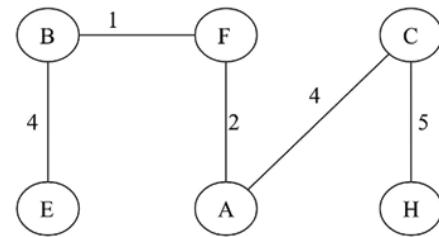
Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of the edges of the connected graph with an edge-weighted graph that connects all the nodes of the graph, with the lowest possible total edge weights and no cycle. More formally, given a connected graph G , where $G = (V, E)$ with real-valued edge weights, an MST is a subgraph with a subset of the edges $T \subseteq E$ so that the sum of edge weights is minimum and there is no cycle. There are many possible spanning trees that can connect all the nodes of the graph without any cycle, but the the minimum weight spanning tree is a spanning tree that has the lowest total edge weight (also called cost) among all other possible spanning trees. An example graph is shown in *Figure 9.28* along with its corresponding MST (on the right) in which we can observe that all the nodes are connected and have a subset of edges taken from the original graph (on the left).

The MST has the lowest total weight of all the edges, i.e. $(1+4+2+4+5 = 16)$ among all the other possible spanning trees:



A graph



A minimum spanning tree

Figure 9.28: A sample graph with the corresponding Minimum Spanning Tree

The MST has diverse real-world applications. They are mainly used in network design for road congestion, hydraulic cables, electric cable networks, and even cluster analysis.

First, let us discuss Kruskal's minimum spanning tree algorithm.

Kruskal's Minimum Spanning Tree algorithm

Kruskal's algorithm is a widely used algorithm for finding the spanning tree from a given weighted, connected, and undirected graph. It is based on the greedy approach, as we firstly find the edge with the lowest weight and add it to the tree, and then in each iteration, we add the edge with the lowest weight to the spanning tree so that we do not form a cycle. In this algorithm, initially, we treat all the vertices of the graph as a separate tree, and then in each iteration we select edge with the lowest weight in such a way that it does not form a cycle. These separate trees are combined, and it grows to form a spanning tree. We repeat this process until all the nodes are processed. The algorithm works as follows:

1. Initialize an empty MST (M) with zero edges
2. Sort all the edges according to their weights
3. For each edge from the sorted list, we add them one by one to the MST (M) in such a way that it does not form a cycle

Let's consider an example.

We start by selecting the edge with the lowest weight (weight 1), as represented by the dotted line shown in *Figure 9.29*:

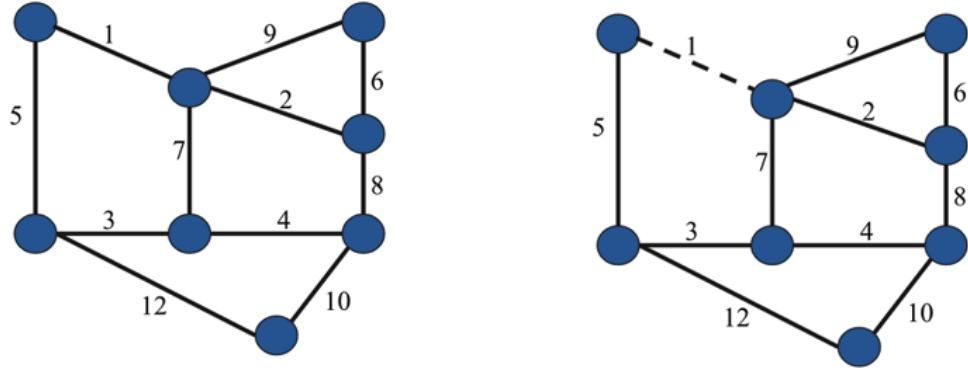


Figure 9.29: Selecting the first edge with the lowest weight in the spanning tree

After selecting the edge with weight 1, we select the edge with weight 2 and then the edge with weight 3, since these are the next lowest weights, as shown in *Figure 9.30*:

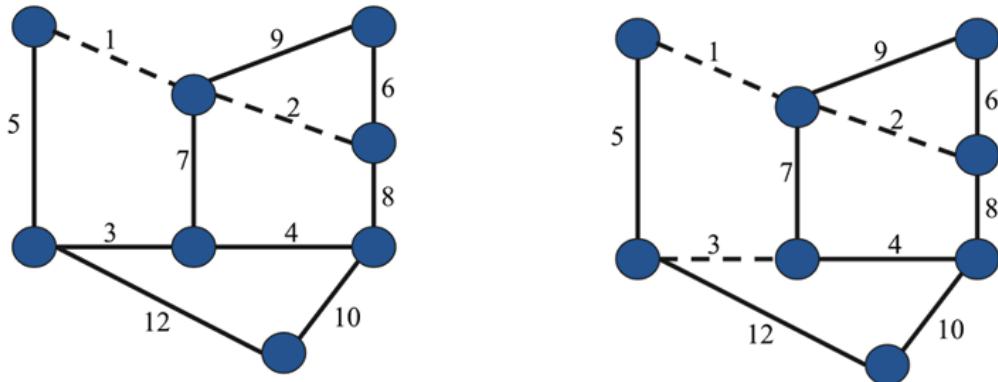


Figure 9.30: Selecting edges with wieghts 2 and 3 in the spanning tree

Similarly, we select the next edges with weights 4 and 5 respectively as shown in *Figure 9.31*:

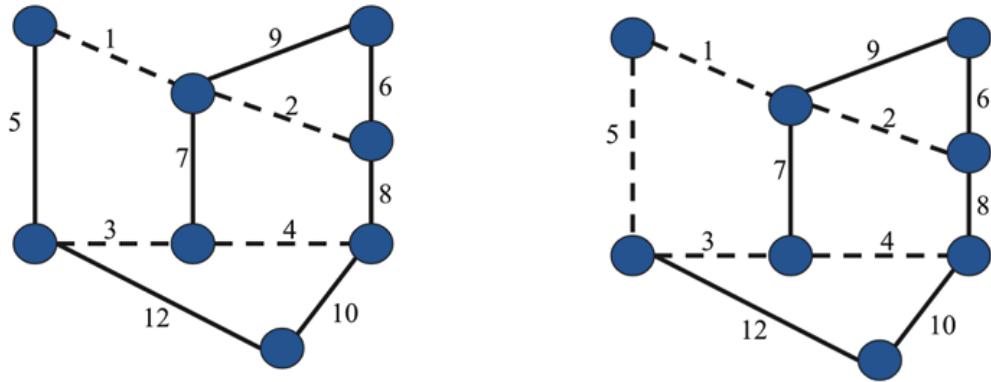


Figure 9.31: Selecting edges with weights 4 and 5 in the spanning tree

Next, we select the next edge with weight 6 and make it a dotted line. After that, we see that the lowest weight is 7 but if we select it, it makes a cycle, so we ignore it. Next, we check the edge with weight 8, and then 9, which are also ignored because they will also form a cycle. So, the next edge with the lowest weight, 10, is selected. This is shown in *Figure 9.32*:

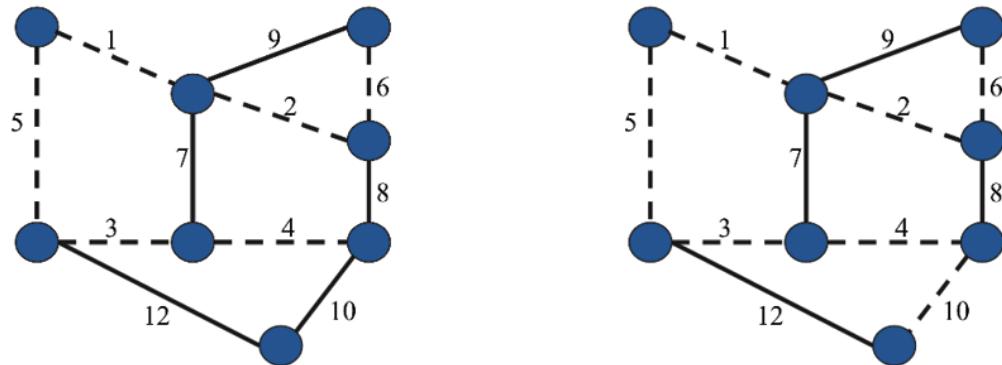


Figure 9.32: Selecting edges with weights 6 and 10 in the spanning tree

Finally, we see the following spanning tree using Kruskal's algorithm, as shown in *Figure 9.33*:

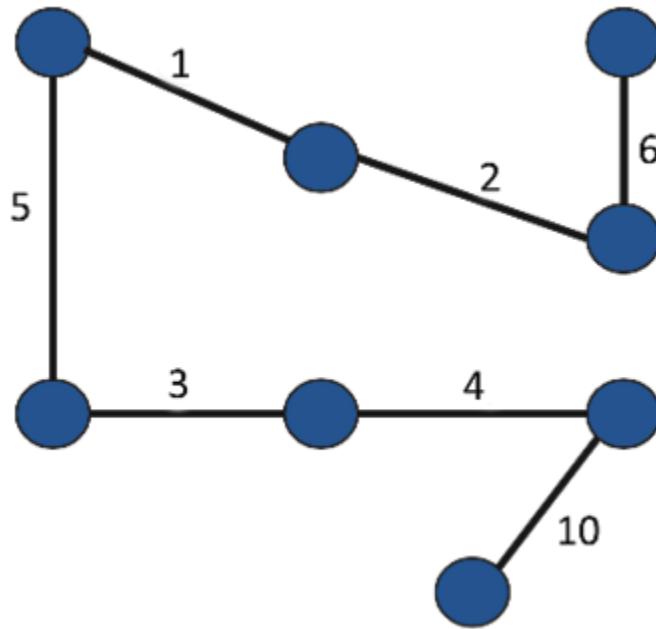


Figure 9.33: The final spanning tree created using Kruskal's algorithm

Kruskal's algorithm has many real-world applications, such as solving the traveling salesman problem (TSP), in which starting from one city, we have to visit all the different cities in a network with the minimum total cost and without visiting the same city twice. There are many other applications, such as TV networks, tour operations, LAN networks, and electric grids.

The time complexity of Kruskal's algorithm is $O(E \log(E))$ or $O(E \log(V))$, where E is the number of edges and V is the number of vertices.

Now, let us discuss one more popular MST algorithm in the next section.

Prim's Minimum Spanning Tree algorithm

Prim's algorithm is also based on a greedy approach to find the minimum cost spanning tree. Prim's algorithm is very similar to the Dijkstra algorithm for finding the shortest path in a graph. In this algorithm, we start with an arbitrary node as a starting point, and then we check the outgoing edges from the selected nodes and traverse through the edge that has the lowest cost (or weights). The terms cost and weight are used interchangeably in this algorithm. So, after starting from the selected node, we grow the tree by selecting the edges, one by one, that have the lowest weight and do not form a cycle. The algorithm works as follows:

1. Create a dictionary that holds all the edges and their weights
2. Get the edges, one by one, that have the lowest cost from the dictionary and grow the tree in such a way that the cycle is not formed
3. Repeat step 2 until all the vertices are visited

Let us consider an example to understand the working of Prim's algorithm. Assuming that we arbitrarily select **A** node, we then check all the outgoing edges from **A**. Here, we have two options, **AB** and **AC**; we select edge **AC** since it has less cost/weight (weight 1), as shown in *Figure 9.34*:

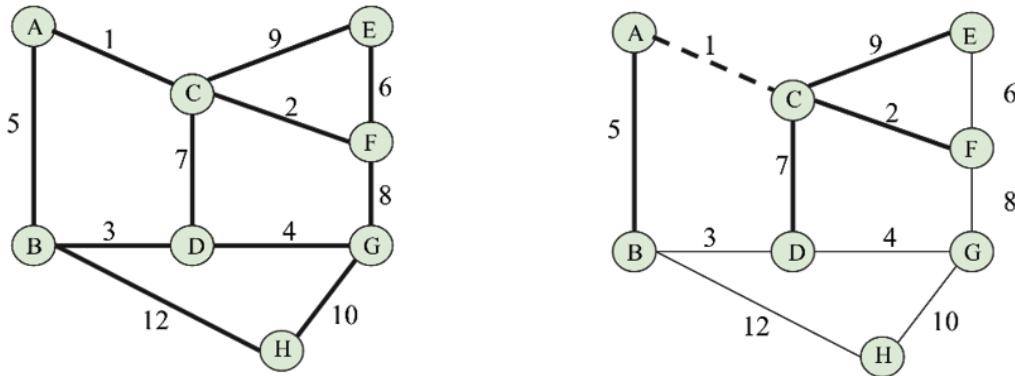


Figure 9.34: Selecting edge **AC** in constructing the spanning tree using Prim's algorithm

Next, we check the lowest outgoing edges from edge **AC**. We have options **AB**, **CD**, **CE**, **CF**, out of which we select edge **CF**, which has the lowest weight of 2. Likewise, we grow the tree, and next we select the lowest weighted edge, i.e., **AB**, as shown in *Figure 9.35*:

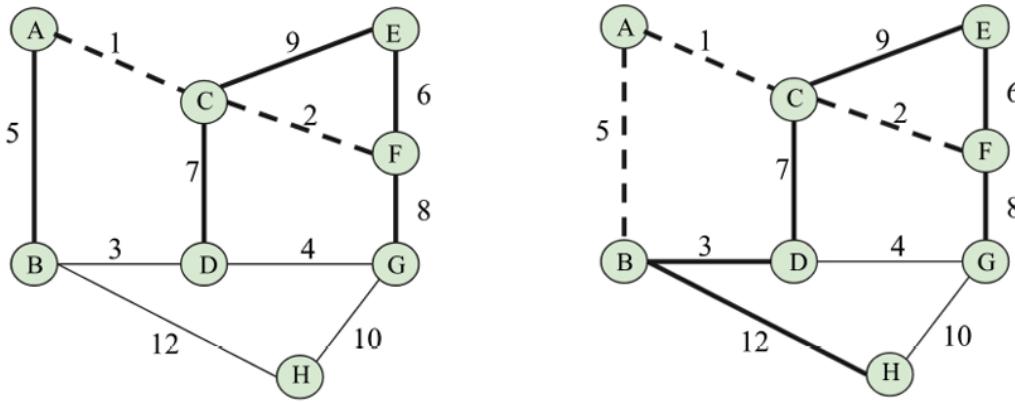


Figure 9.35: Selecting edge **AB** in constructing the spanning tree using Prim's algorithm

Afterward, we select edge **BD**, which has a weight of 3, and similarly, next, we select edge **DG**, which has the lowest weight of 4. This is shown in *Figure 9.36*:

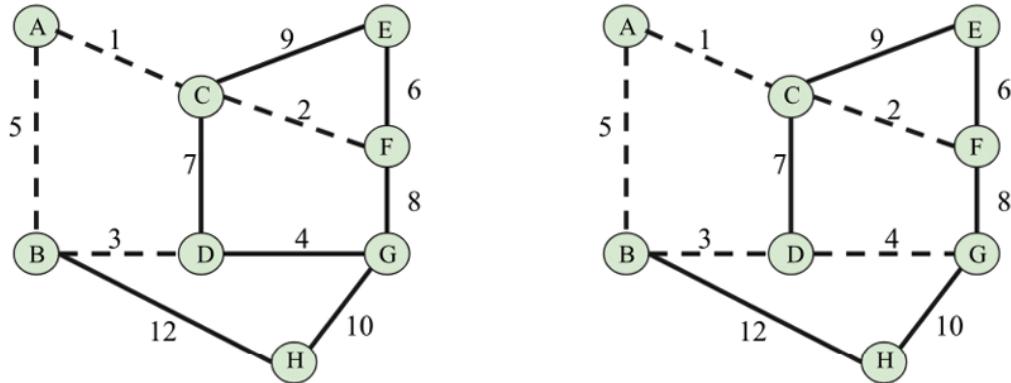


Figure 9.36: Selecting edges BD and DG in constructing the spanning tree using Prim's algorithm

Next, we select edges **FE** and **GH**, which have weights of 6 and 10 respectively, as shown in *Figure 9.37*:

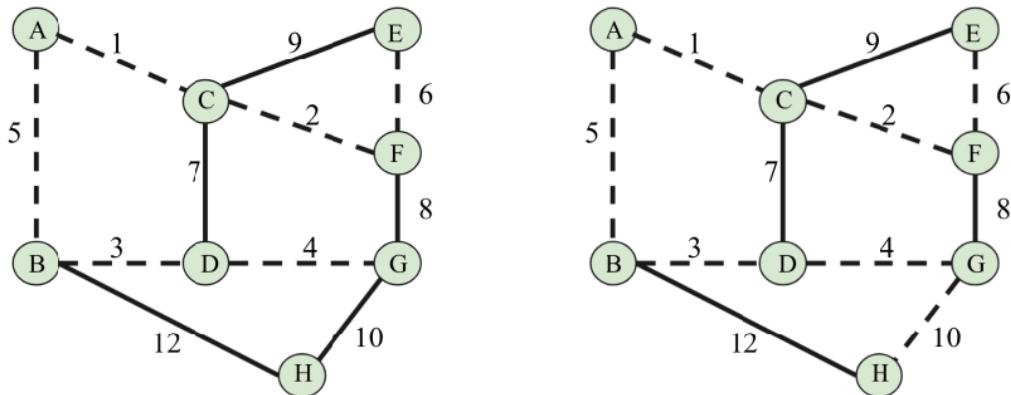


Figure 9.37: Selecting edges FE and GH in constructing the spanning tree using Prim's algorithm

Next, whenever we try to include any more edges, a cycle is formed, so we ignore those edges. Finally, we obtain the spanning tree, which is shown below in *Figure 9.38*:

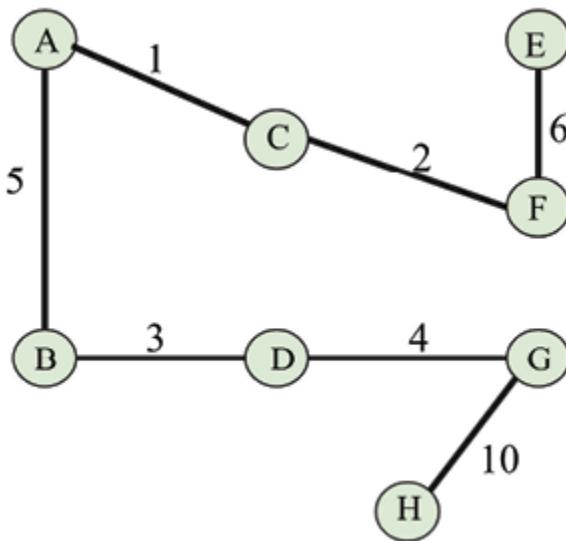


Figure 9.38: The final spanning tree using Prim's algorithm

Prim's algorithm also has many real-world applications. For all the applications where we can use Kruskal's algorithm, we can also use Prim's algorithm. Other applications include road networks, game development, etc.

Since both Kruskal's and Prim's MST algorithms are used for the same purpose, which one should be used? In general, it depends on the structure of the graph. For a graph with C vertices and E edges, Kruskal's algorithm's worst-case time complexity is $O(E \log V)$, and Prim's algorithm has a time complexity of $O(E + V \log V)$. So, we can observe that Prim's algorithm works better when we have a dense graph, whereas Kruskal's algorithm is better when we have a sparse graph.

Summary

A graph is a non-linear data structure, which is very important due to the large number of real-world applications it has. In this chapter, we have discussed different ways to represent a graph in Python, using lists and dictionaries. Further, we learned two very important graph traversal algorithms, i.e., depth-first search (DFS) and breadth-first search (BFS). Moreover, we also discussed two very important algorithms for finding an MST, i.e. Kruskal's algorithm and Prim's algorithm.

In the next chapter, we will discuss searching algorithms and the various methods using which we can efficiently search for items in lists.

Exercises

1. What is the maximum number of edges (without self-loops) possible in an undirected simple graph with five nodes?
2. What do we call a graph in which all the nodes have equal degrees?
3. Explain what cut vertices are and identify the cut vertices in the given graph:

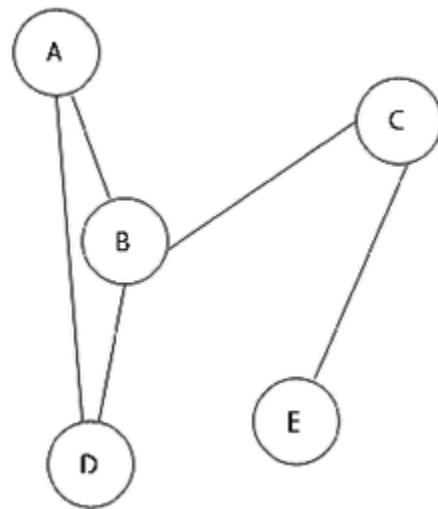


Figure 9.39: A sample graph

4. Assuming a graph G of order n , what will be the maximum number of cut vertices possible in graph G ?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



10

Searching

An important operation for all data structures is searching for elements from a collection of data. There are various methods to search for an element in data structures; in this chapter, we shall explore the different strategies that can be used to find elements in a collection of items.

Data elements can be stored in any kind of data structure, such as an array, link list, tree, or graph; the search operation is very important for many applications, mostly whenever we want to know if a particular data element is present in an existing list of data items. In order to retrieve the information efficiently, we require an efficient search algorithm.

In this chapter, we will learn about the following:

- Various search algorithms
- Linear search algorithm
- Jump search algorithm
- Binary search algorithm
- Interpolation search algorithm
- Exponential search algorithm

Let us start with an introduction to searching and a definition and then look at the linear search algorithm.

Introduction to searching

A search operation is carried out to find the location of the desired data item from a collection of data items. The search algorithm returns the location of the searched value where it is present in the list of items and if the data item is not present, it returns `None`.

Efficient searching is important to efficiently retrieve the location of the desired data item from a list of stored data items. For example, we have a long list of data values, such as `{1, 45, 65, 23, 65, 75, 23}`, and we want to see if `75` is present in the list or not. It becomes important to have an efficient search algorithm when the list of data items becomes large.

There are two different ways in which data can be organized, which can affect how a search algorithm works:

- First, the search algorithm is applied to a list of items that is already sorted; that is, it is applied to an ordered set of items. For example, `[1, 3, 5, 7, 9, 11, 13, 15, 17]`.
- The search algorithm is applied to an unordered set of items, which is not sorted. For example, `[11, 3, 45, 76, 99, 11, 13, 35, 37]`.

We will first take a look at linear searching.

Linear search

The search operation is used to find out the index position of a given data item in a list of data items. If the searched item is available in the given list of data items, then the search algorithm returns the index position where it is located; otherwise, it returns that the item is not found. Here, the index position is the location of the desired item in the given list.

The simplest approach to search for an item in a list is to search linearly, in which we look for items one by one in the whole list. Let's take an example of six list items {60, 1, 88, 10, 11, 100} to understand the linear search algorithm, as shown in *Figure 10.1*:

60	1	88	10	11	100
[0]	[1]	[2]	[3]	[4]	[5]

Figure 10.1: An example of linear search

The preceding list has elements that can be accessed through the index. To find an element in the list, we can search for the given element linearly one by one. This technique traverses the list of elements by using the index to move from the beginning of the list to the end. Each element is checked, and if it does not match the search item, the next item is examined. By hopping from one item to the next, the list is traversed sequentially. We use list items with integer values in this chapter to help you understand the concept, since integers can be compared easily; however, a list item can hold any other data type as well.

The linear search approach depends on how the list items are stored in memory—whether they are already sorted in order or they are not

sorted. Let's first see how the linear search algorithm works if the given list of items is not sorted.

Unordered linear search

The unordered linear search is a linear search algorithm in which the given list of date items is not sorted. We linearly match the desired data item with the data items of the list one by one till the end of the list or until the desired data item is found. Consider an example list that contains the elements `60`, `1`, `88`, `10`, and `100`—an unordered list. To perform a `search` operation on such a list, one proceeds with the first item and compares that with the search item. If the search item is not matched, then the next element in the list is checked. This continues till we reach the last element in the list or until a match is found.

In an unordered list of items, the search for the term `10` starts from the first element and moves to the next element in the list. Thus, firstly `60` is compared with `10`, and since it is not equal, we compare `66` with the next element `1`, then `88`, and so on till we find the search term in the list. Once the item is found, we return the index position of where we have found the desired item. This process is shown in *Figure 10.2*:

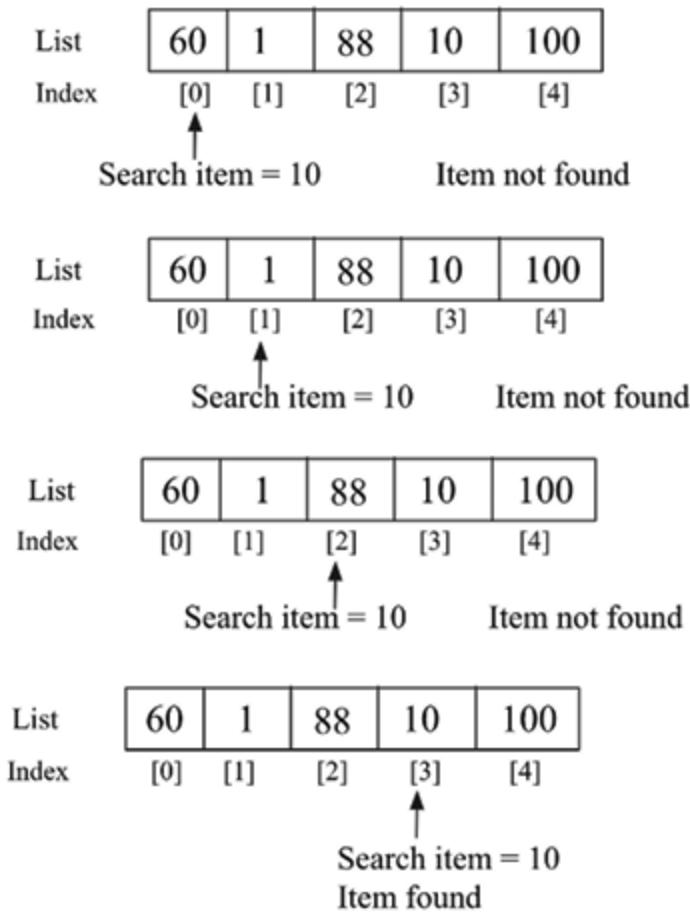


Figure 10.2: Unordered linear search

Here is the implementation in Python for the linear search on an unordered list of items:

```
def search(unordered_list, term):
    for i, item in enumerate(unordered_list):
        if term == unordered_list[i]:
            return i
    return None
```

The `search` function takes two parameters; the first is the list that holds the data, and the second parameter is the item that we are looking for, called the **search term**. On every pass of the `for` loop, we

check if the search term is equal to the indexed item. If this is true, then there is a match, and there is no need to proceed further with the search. We return the index position where the searched item is found in the list. If the loops run to the end of the list with no match found, then `None` is returned to signify that there is no such item in the list.

We can use the following code snippet to check if a desired data element is present in the given list of data items:

```
list1 = [60, 1, 88, 10, 11, 600]

search_term = 10
index_position = search(list1, search_term)
print(index_position)

list2 = ['packt', 'publish', 'data']
search_term2 = 'data'
Index_position2 = search(list2, search_term2)
print(Index_position2)
```

The output of the above code is as follows:

```
3
2
```

In the output of the above code, firstly, the index position `3` is returned when we search for data element `10` in `list1`. And secondly, index position `2` is returned when data item `'data'` is searched for in `list2`. We can use the same algorithm for searching a non-numeric data item from a list of non-numeric data items in Python, since string elements can also be compared similarly to numeric data in Python.

When searching for any element from an unordered list of items, in the worst case the desired item may be in the last position or may not be present in the list. In this situation we will have to compare the search item with all the elements of the list, i.e. n times if the total number of data items in the list is n . Thus, the unordered linear search has a worst-case running time of $O(n)$. All the elements may need to be visited before finding the search term. The worst-case scenario will be when the search term is located at the last position of the list.

Next, we discuss how the linear search algorithm works if the given list of data items is already sorted.

Ordered linear search

If the data elements are already arranged in a sorted order, then the linear search algorithm can be improved. The linear search algorithm in a sorted list of elements has the following steps:

1. Move through the list sequentially
2. If the value of a search item is greater than the object or item currently under inspection in the loop, then quit and return `None`

In the process of iterating through the list, if the value of the search term is less than the current item in the list, then there is no need to continue with the search. Let's consider an example to see how this works. Let's say we have a list of items $\{2, 3, 4, 6, 7\}$ as shown in *Figure 10.3*, and we want to search for term 5 :

List	2	3	4	6	7
Index	[0]	[1]	[2]	[3]	[4]
	↑				

Search item = 5 Item not found

List	2	3	4	6	7
Index	[0]	[1]	[2]	[3]	[4]
	↑				

Search item = 5 Item not found

List	2	3	4	6	7
Index	[0]	[1]	[2]	[3]	[4]
	↑				

Search item = 5 Item not found

List	2	3	4	6	7
Index	[0]	[1]	[2]	[3]	[4]
	↑				

Search item = 5
Since $5 < 6$, we stop searching.

Figure 10.3: Example of ordered linear search

We start the `search` operation by comparing the desired search element `5` with the first element; no match is found. We continue on to compare the search element with the next element, i.e. `3`, in the list. Since it also does not match, we move on to examine the next element, i.e. `4`, and since it also does not match, we continue searching in the list, and we compare the search element with the fourth element, i.e. `6`. This also does not match the search term. Since the given list is already sorted in ascending order and the value of the search item is less than the fourth element, the search item

cannot be found in any later position in the list. In other words, if the current item in the list is greater than the search term, then it means there is no need to further search the list, and we stop searching for the element in the list.

Here is the implementation of the linear search when the list is already sorted:

```
def search_ordered(ordered_list, term):
    ordered_list_size = len(ordered_list)
    for i in range(ordered_list_size):
        if term == ordered_list[i]:
            return i
        elif ordered_list[i] > term:
            return None
    return None
```

In the preceding code, the `if` statement now caters to checking if the search item is found in the list or not. Then, `elif` tests the condition where `ordered_list[i] > term`. We stop searching if the comparison evaluates to `True`, which means the current item in the list is greater than the search element. The last line in the method returns `None` because the loop may go through the list and still the search item is not matched in the list.

We use the following code snippet to use the search algorithm:

```
list1 = [2, 3, 4, 6, 7]

search_term = 5
index_position1 = search_ordered(list1, search_term)

if index_position1 is None:
    print("{} not found".format(search_term))
```

```
        else:
            print("{} found at position {}".format(search_term, index_positio

list2 = ['book', 'data', 'packt', 'structure']

search_term2 = 'structure'
index_position2 = search_ordered(list2, search_term2)
if index_position2 is None:
    print("{} not found".format(search_term2))
else:
    print("{} found at position {}".format(search_term2, index_positi
```

The output of the above code is as follows:

```
5 not found
structure found at position 3
```

In the output of the above code, firstly, the search item `5` is not matched in the given list. And for the second list of non-numeric data elements, the string `structure` is matched at index position `3`. Hence, we can use the same linear search algorithm for searching a non-numeric data item from an ordered list of data items, so the given list of data items should be sorted similarly to a contact list on a phone.

In the worst-case scenario, the desired search item will be present in the last position of the list or will not be present at all. In this situation, we will have to trace the complete list (say `n` elements). Thus, the worst-case time complexity of an ordered linear search is $O(n)$.

Next, we will discuss the jump search algorithm.

Jump search

The **jump search** algorithm is an improvement over linear search for searching for a given element from an ordered (or sorted) list of elements. This uses the divide-and-conquer strategy in order to search for the required element. In linear search, we compare the search value with each element of the list, whereas in jump search, we compare the search value at different intervals in the list, which reduces the number of comparisons.

In this algorithm, firstly, we divide the sorted list of data into subsets of data elements called blocks. Within each block, the highest value will lie within the last element, as the array is sorted. Next, in this algorithm, we start comparing the search value with the last element of each block. There can be three conditions:

1. If the search value is less than the last element of the block, we compare it with the next block.
2. If the search value is greater than the last element of the block, it means the desired search value must be present in the current block. So, we apply linear search in this block and return the index position.
3. If the search value is the same as the compared element of the block, we return the index position of the element and we return the candidate.

Generally, the size of the block is taken as \sqrt{n} , since it gives the best performance for a given array of length n .

In the worst-case situation, we will have to make n/m number of jumps (here, n is the total number of elements, and m is the block

size) if the last element of the last block is greater than the item to be searched, and we will need $m - 1$ comparisons for linear search in the last block. Therefore, the total number of comparisons will be $((n/m) + m - 1)$, which will minimize when $m = \sqrt{n}$. So the size of the block is taken as \sqrt{n} since it gives the best performance.

Let's take an example list `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}` to search for a given element (say `10`):

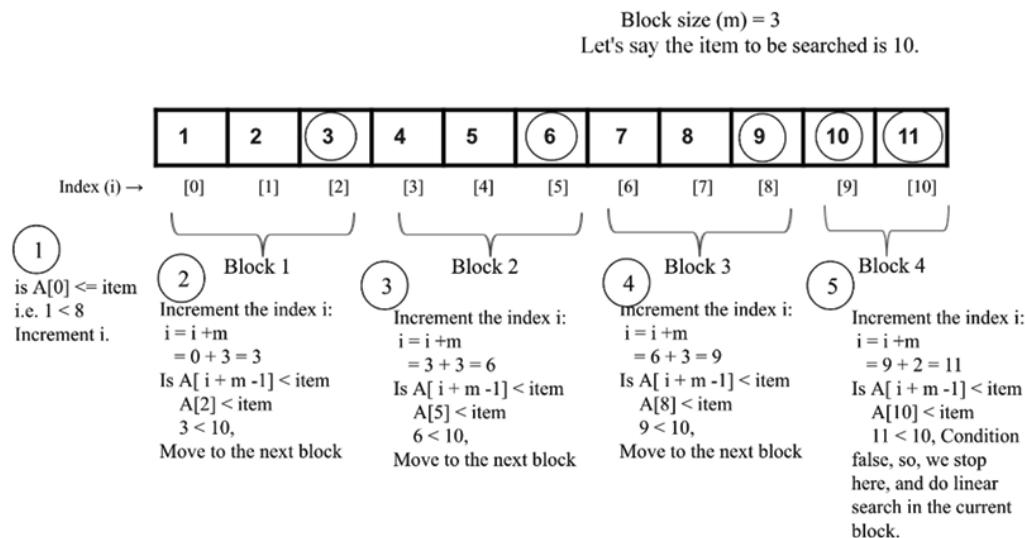


Figure 10.4: Illustration of the jump search algorithm

In the above example, we find the desired element `10` in `5` comparisons. Firstly, we compare the first value of the array with the desired item `A[0] <= item`; if it is true, then we increase the index by the block size (this is shown in *step 1* in *Figure 10.4*). Next, we compare the desired item with the last element of each block. If it is greater, then we move to the next block, such as from block 1 to block 3 (this is shown in *steps 2, 3, and 4* in *Figure 10.4*).

Further, when the desired search element becomes smaller than the last element of a block, we stop incrementing the index position and then we do the linear search in the current block. Now, let us discuss the implementation of the jump searching algorithms. Firstly, we implement the linear search algorithm, which is similar to what we discussed in the previous section.

It is given again here for the sake of completeness as follows:

```
def search_ordered(ordered_list, term):
    print("Entering Linear Search")
    ordered_list_size = len(ordered_list)
    for i in range(ordered_list_size):
        if term == ordered_list[i]:
            return i
        elif ordered_list[i] > term:
            return -1
    return -1
```

In the above code, given an ordered list of elements, it returns the index of the location where a given data element is found in the list. It returns `-1` if the desired element is not found in the list. Next, we implement the `jump_search()` method as follows:

```
def jump_search(ordered_list, item):
    import math
    print("Entering Jump Search")
    list_size = len(ordered_list)
    block_size = int(math.sqrt(list_size))
    i = 0
    while i != len(ordered_list)-1 and ordered_list[i] <= item:
        print("Block under consideration - {}".format(ordered_list[i]))
        if i+block_size > len(ordered_list):
            block_size = len(ordered_list) - i
            block_list = ordered_list[i: i+block_size]
            . . . . .
```

```
j = search_ordered(block_list, item)
if j == -1:
    print("Element not found")
    return
return i + j
if ordered_list[i + block_size - 1] == item:
    return i+block_size-1
elif ordered_list[i + block_size - 1] > item:
    block_array = ordered_list[i: i + block_size - 1]
    j = search_ordered(block_array, item)
    if j == -1:
        print("Element not found")
        return
    return i + j
i += block_size
```

In the above code, firstly we assign the length of the list to the variable `n`, and then we compute the block size as \sqrt{n} . Next, we start with the first element, index 0, and then continue searching until we reach the end of the list.

We start with the starting index `i = 0` with a block of size m , and we continue incrementing until the window reaches the end of the list. We compare whether `ordered_list [I + block_size -1] == item`. If they match, it returns the index position `(i+ block_size -1)`. The code snippet for this is as follows:

```
if ordered_list[i+ block_size -1] == item:
    return i+ block_size -1
```

If `ordered_list [i+ block_size -1] > item`, we proceed to carry out the linear search algorithm inside the current block `block_array = ordered_list [i : i+ block_size-1]`, as follows:

```

    elif ordered_list[i + block_size - 1] > item:
        block_array = ordered_list[i: i + block_size - 1]
        j = search_ordered(block_array, item)
        if j == -1:
            print("Element not found")
            return
        return i + j

```

In the above code, we use the linear search algorithm in the subarray. It returns `-1` if the desired element is not found in the list; otherwise, the index position of `(i + j)` is returned. Here, `i` is the index position until the previous block where we may find the desired element and `j` is the position of the data element within the block where the desired element is matched. This process is also depicted in *Figure 10.5*.

In this figure, we can see that `i` is in index position 5, and then `j` is the number of elements within the final block where we find the desired element, i.e. `2`, so the final returned index will be `5 + 2 = 7`:

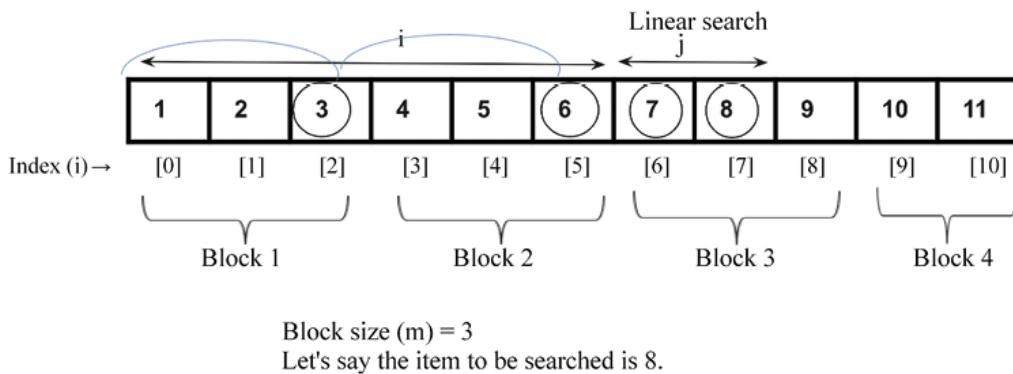


Figure 10.5: Demonstration of index position i and j for the search value 8

Further, we need to check for the length of the last block since it may have a number of elements less than the block size. For example, if

the total number of elements is 11, then in the last block we will have 2 elements. So, we check if the desired search element is present in the last block, and if so we should update the starting and ending index as follows:

```
if i+ block_size > len(ordered_list):
    block_size = len(ordered_list) - i
    block_list = ordered_list[i: i+block_size]
    j = search_ordered(block_list, item)
    if j == -1:
        print("Element not found")
        return
    return i + j
```

In the above code, we search for the desired element using the linear search algorithm.

Finally, if `ordered_list[i+m-1] < item`, then we move to the next iteration, and update the index by adding the block size to the index as `i += block_size`.

```
print(jump_search([1,2,3,4,5,6,7,8,9, 10, 11], 8))
```

The output of the above code snippet is:

```
Entering Jump Search
Block under consideration - [1, 2, 3]
Block under consideration - [4, 5, 6]
Block under consideration - [7, 8, 9]
Entering Linear Search
7
```

In the above output, we can see the steps for how we searched for element `10` in the given list of elements.

Thus, jump search performs linear search on a block, so first it finds the block in which the element is present and then applies linear search within that block. The size of the block depends on the size of the array. If the size of the array is `n`, then the block size may be \sqrt{n} . If it does not find the element in that block, it moves to the next block. The jump search first finds out in which block the desired element may be present. For a list of `n` elements, and a block size of m , the total number of jumps possible will be n/m jumps. Let's say the size of the block is \sqrt{n} ; thus, the worst-case time complexity will be $O(\sqrt{n})$.

Next, we will discuss the binary search algorithm.

Binary search

The **binary search** algorithm finds a given item from the given sorted list of items. It is a fast and efficient algorithm to search for an element; however, one drawback of this algorithm is that we need a sorted list. The worst-case running time complexity of a binary search algorithm is $O(\log n)$ whereas for linear search it is $O(n)$.

The binary search algorithm works as follows. It starts searching for the item by dividing the given list in half. If the search item is smaller than the middle value then it will look for the searched item only in the first half of the list, and if the search item is greater than the middle value it will only look at the second half of the list. We repeat the same process every time until we find the search item, or

we have checked the whole list. In the case of a non-numeric list of data items, for example, if we have string data items, then we should sort the data items in alphabetical order (similar to how a contact list is stored on a phone).

Let's understand the binary search algorithm with an example. Suppose we have a book with 1,000 pages, and we want to reach page number 250. We know that every book has its pages numbered sequentially from 1 upward. So, according to the binary search analogy, we first check for search item 250, which is less than the midpoint, which is 500. Thus, we search for the required page only in the first half of the book.

We again find the midpoint of the first half of the book, using page 500 as a reference we find the midpoint, 250. That brings us closer to finding the 250th page. Then we find the required page in the book.

Let's take another example to understand the workings of binary search. We want to search for item 43 from a list of 12 items, as shown in *Figure 10.6*:

If we want to search 43 in the given list

1	4	11	25	32	37	40	43	47	49	53	55
---	---	----	----	----	----	----	----	----	----	----	----

since $43 > 37$

We look only at the second half

1	4	11	25	32	37	40	43	47	49	53	55
---	---	----	----	----	----	----	----	----	----	----	----

since $43 > 37$

We now look only at the first half of the list

1	4	11	25	32	37	40	43	47	49	53	55
---	---	----	----	----	----	----	----	----	----	----	----

Search item 43 is found.

The function will return the index position

1	4	11	25	32	37	40	43	47	49	53	55
---	---	----	----	----	----	----	----	----	----	----	----

Figure 10.6: Working of binary search

We start searching for the item by comparing it to the middle item of the list, which is 37 in the example. If the value of the search item is less than the middle value, we only look at the first half of the list; otherwise, we will look at the other half. So, we only need to search for the item in the second half. We follow the same procedure until we find search item 43 in the list. This process is shown in the Figure 10.6.

The following is an implementation of the binary search algorithm on an ordered list of items:

```
def binary_search_iterative(ordered_list, term):
    size_of_list = len(ordered_list) - 1
    index_of_first_element = 0
```

```

index_of_last_element = size_of_list
while index_of_first_element <= index_of_last_element:
    mid_point = (index_of_first_element + index_of_last_element)/
    if ordered_list[mid_point] == term:
        return mid_point
    if term > ordered_list[mid_point]:
        index_of_first_element = mid_point + 1
    else:
        index_of_last_element = mid_point - 1
if index_of_first_element > index_of_last_element:
    return None

```

We'll explain the above code using a list of sorted elements `{10, 30, 100, 120, 500}`. Now let's assume we have to find the position where item `10` is located in the list shown in *Figure 10.7*:

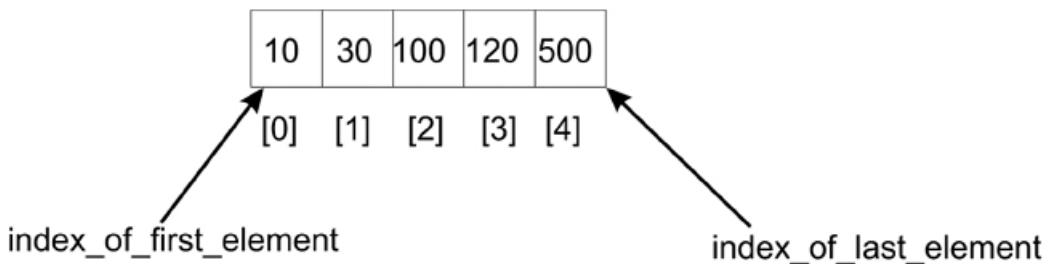


Figure 10.7: Sample list of five items

Firstly, we declare two variables, i.e. `index_of_first_element` and `index_of_last_element`, which denote the starting and ending index positions in the given list. Next, the algorithm uses a `while` loop to iteratively adjust the limits in the list within which we have to find a search item. The terminating condition to stop the `while` loop is that the difference between the starting index, `index_of_first_element`, and the `index_of_last_element` index should be positive.

The algorithm first finds the midpoint of the list by adding the index of the first element (i.e. `0` in this case) to the index of the last element

(which is 4 in this example) and dividing it by 2. We get the middle index, `mid_point`:

```
mid_point = (index_of_first_element + index_of_last_element)/2
```

In this case, the index of the midpoint is 2, and the data item stored at this position is 100. We compare the midpoint element with the search item 10.

Since these do not match, and the search item 10 is less than the midpoint, the desired search item should lie in the first half of the list, thus, we adjust the index range of `index_of_first_element` to `mid_point-1`, which means the new search range becomes 0 to 1, as shown in *Figure 10.8*:

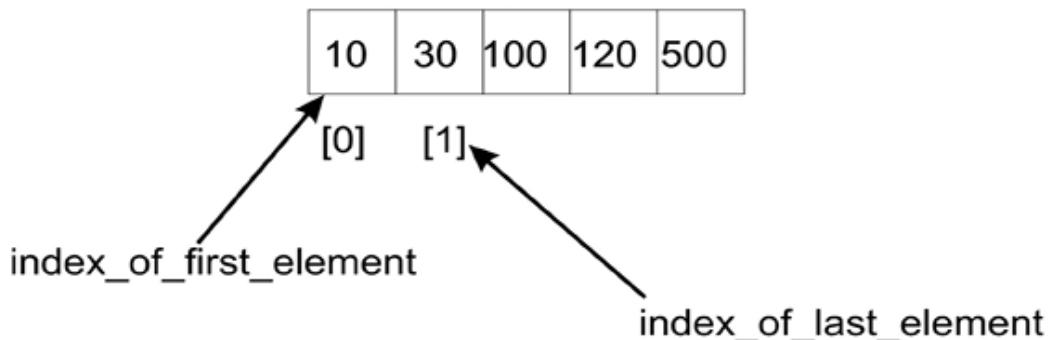


Figure 10.8: Index of first and last elements for the first half of the list

However, if we had been searching for 120, as 120 would have been greater than the middle value (100), we would have searched for the item in the second half of the list, and as a result, we would have needed to change the list index range to be `mid_point +1` to `index_of_last_element`. In that case the new range would have been (3, 4).

So, with the new indexes of the first and last elements, i.e.

`index_of_first_element` and `index_of_last_element`, now being `0` and `1` respectively, we compute the midpoint $(0 + 1)/2$, which equals `0`. The new midpoint is `0`, so we find the middle item and compare it with the search item, which yields the value `10`. Now, our search item is found, and the index position is returned.

Finally, we check if `index_of_first_element` is less than `index_of_last_element` or not. If this condition fails, it means that the search term is not in the list.

We can use the below code snippet to search for a term/item in the given list:

```
list1 = [10, 30, 100, 120, 500]

search_term = 10
index_position1 = binary_search_iterative(list1, search_term)
if index_position1 is None:
    print("The data item {} is not found".format(search_term))
else:
    print("The data item {} is found at position {}".format(search_te

list2 = ['book', 'data', 'packt', 'structure']

search_term2 = 'structure'
index_position2 = binary_search_iterative(list2, search_term2)
if index_position2 is None:
    print("The data item {} is not found".format(search_term2))
else:
    print("The data item {} is found at position {}".format(search_te
```

The output of the above code is as follows:

```
The data item 10 is found at position 0  
The data item structure is found at position 3
```

In the above code, firstly we check the search term `10` in the list, and we get the correct location, i.e. index position `0`. Further, we check the index position of the string structure in the given sorted list of data items, and we get the index position `3`.

The implementation that we have discussed is based on an iterative process. However, we can also implement it using the recursive method, in which we recursively shift the pointers that point to the beginning (or starting) and end of the search list. See the following code for an example of a recursive implementation of the binary search algorithm:

```
def binary_search_recursive(ordered_list, first_element_index, last_e
    if (last_element_index < first_element_index):
        return None
    else:
        mid_point = first_element_index + ((last_element_index - firs
            if ordered_list[mid_point] > term:
                return binary_search_recursive (ordered_list, first_eleme
            elif ordered_list[mid_point] < term:
                return binary_search_recursive (ordered_list, mid_point+1
            else:
                return mid_point
```

A call to this recursive implementation of the binary search algorithm and its output is as follows:

```
list1 = [10, 30, 100, 120, 500]
```

```
search_term = 10
```

```

index_position1 = binary_search_recursive(list1, 0, len(list1)-1, search_term)
if index_position1 is None:
    print("The data item {} is not found".format(search_term))
else:
    print("The data item {} is found at position {}".format(search_term))

list2 = ['book', 'data', 'packt', 'structure']

search_term2 = 'data'
index_position2 = binary_search_recursive(list2, 0, len(list1)-1, search_term2)
if index_position2 is None:
    print("The data item {} is not found".format(search_term2))
else:
    print("The data item {} is found at position {}".format(search_term2))

```

The output of the above code is as follows:

```

The data item 10 is found at position 0
The data item data is found at position 1

```

Here, the only distinction between the recursive binary search and the iterative binary search is the function definition and also the way in which `mid_point` is calculated. The calculation for `mid_point` after the `((last_element_index - first_element_index)//2)` operation must add its result to `first_element_index`. That way, we define the portion of the list to attempt the search.

In binary search, we repeatedly divide the search space (i.e. the list in which the desired item may lie) in half. We start with the complete list, and in each iteration, we compute the middle point; we only consider half the list to search for the item and the other half of the list is ignored. We repeatedly check until the value is found, or the interval is empty. Therefore, at each iteration, the size of the array

reduces by half; for example, at iteration 1, the size of the list is n , in iteration 2, the size of the list becomes $n/2$, in iteration 3 the size of the list becomes $n/2^2$, and after k iterations the size of the list becomes $n/2^k$. At that time the size of the list will be equal to 1. That means:

$$\Rightarrow n/2^k = 1$$

Applying the \log function on both sides:

$$\begin{aligned}\Rightarrow \log_2(n) &= \log_2(2^k) \\ \Rightarrow \log_2(n) &= k \log_2(2) \\ \Rightarrow k &= \log_2(n)\end{aligned}$$

Hence, the binary search algorithm has the worst-case time complexity of $O(\log n)$.

Next, we will discuss the interpolation search algorithm.

Interpolation search

The binary search algorithm is an efficient algorithm for searching. It always reduces the search space by half by discarding one half of the search space depending on the value of the search item. If the search item is smaller than the value in the middle of the list, the second half of the list is discarded from the search space. In the case of binary search, we always reduce the search space by a fixed value of half, whereas the interpolation search algorithm is an improved version of the binary search algorithm in which we use a more

efficient method that reduces the search space by more than half after each iteration.

The interpolation search algorithm works efficiently when there are uniformly distributed elements in the sorted list. In a binary search, we always start searching from the middle of the list, whereas in the interpolation search we compute the starting search position depending on the item to be searched. In the interpolation search algorithm, the starting search position is most likely to be close to the start or end of the list; if the search item is near the first element in the list, then the starting search position is likely to be near the start of the list and if the search item is near the end of the list, then the starting search position is likely to be near the end of the list.

It is quite similar to how humans perform a search on any list of items. It is based on trying to make a good guess of the index position where a search item is likely to be found in a sorted list of items.

It works in a similar way to the binary search algorithm except for the method to determine the splitting criteria to divide the data in order to reduce the number of comparisons. In the case of a binary search, we divide the data into equal halves and in the case of an interpolation search, we divide the data using the following formula:

$$mid = low_index + \frac{(upper_index - low_index)}{(list[upper_index] - list[low_index])} * (search_term - list[low_index])$$

In the preceding formula, `low_index` is the lower-bound index of the list, which is the index of the smallest value, and `upper_index` denotes the index position of the highest value in the list. The `list[low_index]` and `list[upper_index]` are the lowest and highest values respectively

in the list. The `search_value` variable contains the value of the item that is to be searched.

Let's consider an example to understand how the interpolation search algorithm works using the following list of seven items:

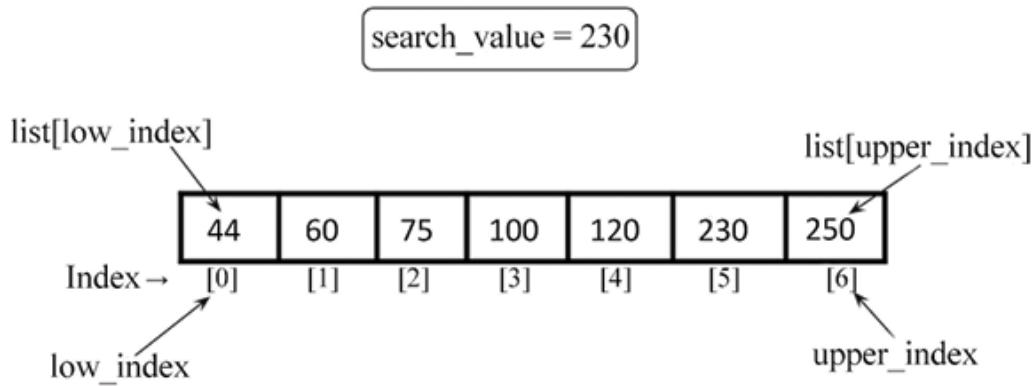


Figure 10.9: Example of interpolation search

Given the list of seven items, 44, 60, 75, 100, 120, 230, and 250, the `mid` point can be computed using the above mentioned formula with the following values:

```
list1 = [4, 60, 75, 100, 120, 230, 250]
low_index = 0
upper_index = 6
list1[upper_index] = 250
list1[low_index] = 44
search_value = 230
```

Putting the values of all the variables in the formula, we get:

```
mid = low_index + ((upper_index - low_index)/ (list1[upper_index] -
=> 0 + [(6-0)/(250-44)] * (230-44)
=> 5.41
=> 5
```

The `mid` index is `5`, in the case of an interpolation search, so the algorithm starts searching from the index position `5`. So, this is how we compute the midpoint from which we start searching for the given element.

The interpolation search algorithm works as follows:

1. We start searching for the given search value from the midpoint (we have just seen how to compute it).
2. If the search value matches the value stored at the index of the midpoint, we return this index position.
3. If the search value does not match the value stored at the midpoint, we divide the list into two sublists, i.e. a higher sublist and lower sublist. The higher sublist has all the elements with higher index values than the midpoint, and the lower sublist has all the elements with lower index values.
4. If the search value is greater than the value of the midpoint, we search the given search value in the higher sublist and ignore the lower sublist.
5. If the search value is lower than the value of the midpoint, we search the given search value in the lower sublist and ignore the higher sublist.
6. We repeat the process until the size of the sublists is reduced to zero.

Let us understand the implementation of the interpolation search algorithm. Firstly, we define the `nearest_mid()` method, which computes the midpoint as follows:

```
def nearest_mid(input_list, low_index, upper_index, search_value):
    mid = low_index + ((upper_index - low_index)/(input_list[upper_index] - input_list[low_index]))
    return int(mid)
```

The `nearest_mid` function takes, as arguments, the lists on which to perform the search. The `low_index` and `upper_index` parameters represent the bounds in the list within which we are hoping to find the search term. Furthermore, `search_value` represents the value being searched for.

In an interpolation search, the midpoint is generally more to the left or right. This is caused by the effect of the multiplier being used when dividing to obtain the midpoint. The implementation of the interpolation algorithm remains the same as that of the binary search except for the way we compute the midpoint.

In the following code, we provide the implementation of the interpolation search algorithm:

```
def interpolation_search(ordered_list, search_value):
    low_index = 0
    upper_index = len(ordered_list) - 1
    while low_index <= upper_index:
        mid_point = nearest_mid(ordered_list, low_index, upper_index, search_value)
        if mid_point > upper_index or mid_point < low_index:
            return None
        if ordered_list[mid_point] == search_value:
            return mid_point
        if search_value > ordered_list[mid_point]:
            low_index = mid_point + 1
        else:
            upper_index = mid_point - 1
    if low_index > upper_index:
        return None
```

In the above code, we initialize the `low_index` and `upper_index` variables for the given sorted list. We firstly compute the midpoint using the `nearest_mid()` method.

The computed midpoint using the `nearest_mid` function may produce values that are greater than `upper_bound_index` or lower than `lower_bound_index`. When this occurs, it means the search term, `term`, is not in the list. `None` is, therefore, returned to represent this.

Next, we match the search value with the value stored at the midpoint, i.e. `ordered_list[mid_point]`. If that matches, the index of the midpoint is returned; if it does not match, then we divide the lists into higher and lower sublists, and we readjust `low_index` and `upper_index` so that the algorithm will focus on the sublist that is likely to contain the search term similar to what we did in the binary search:

```
if search_value > ordered_list[mid_point]:  
    low_index = mid_point + 1  
else:  
    upper_index = mid_point - 1
```

In the above code, we check if the search value is greater than the value stored at `ordered_list[mid_point]`, then we only adjust the `low_index` variable to point to the `mid_point + 1` index.

Let's see how this adjustment occurs. Suppose we want to search for `190` in the given list in *Figure 10.10*, then the midpoint will be `4` as per the above formula. Then we compare the search value (i.e. `190`) with the value stored at the midpoint (i.e. `120`). Since the search value is

greater, we search for the element in the higher sublist, and readjust the `low_index` value. This is shown in *Figure 10.10*:

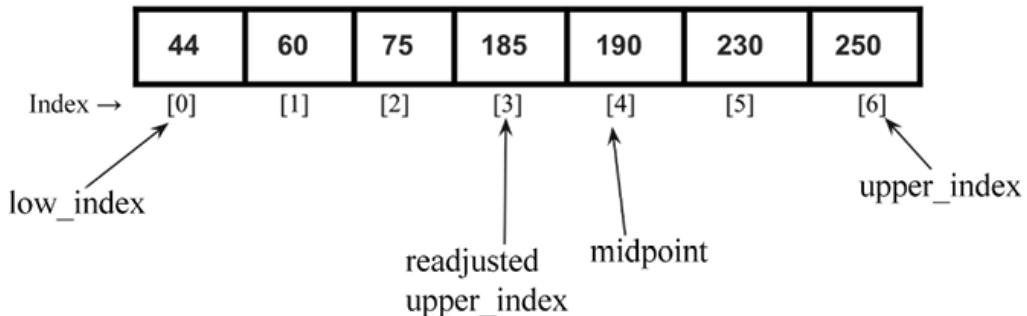


Figure 10.10: Readjustment of the low_index when the value of the search item is greater than the value at the midpoint

On the other hand, if the value of the search term is less than the value stored at `ordered_list[mid_point]`, then we only adjust the `upper_index` variable to point to the index `mid_point - 1`. For example, if we have the list shown in *Figure 10.11*, and we want to search for 185, then the midpoint will be 4 as per the formula.

Next, we compare the search value (i.e. 185) with the value stored at the midpoint (i.e. 190). Since the search value is less as compared to `ordered_list[mid_point]`, we search for the element in the lower sublist, and readjust the `upper_index` value. This is shown in *Figure 10.11*:

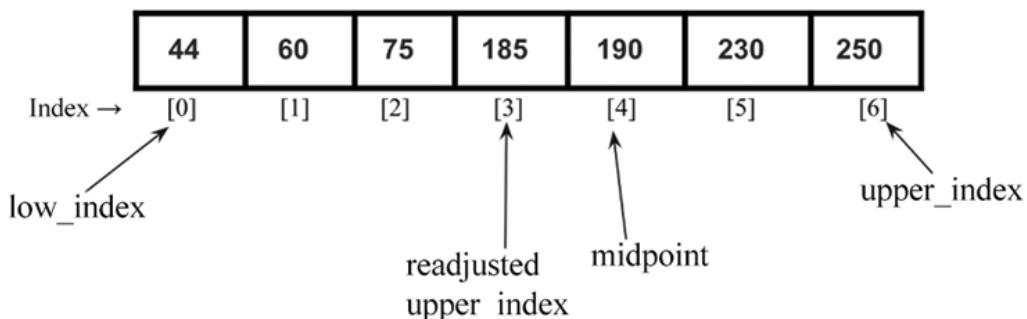


Figure 10.11: Readjustment of the upper_index when the search item is less than the value at the midpoint

The following code snippet can be used to create a list of elements `{44, 60, 75, 100, 120, 230, 250}`, in which we want to search for `120` using the interpolation search algorithm.

```
list1 = [44, 60, 75, 100, 120, 230, 250]
a = interpolation_search(list1, 120)
print("Index position of value 2 is ", a)
```

The output of the above code is as follows:

```
Index position of value 2 is 4
```

Let's use a more practical example to understand the inner workings of both the binary search and interpolation algorithms.

Consider for example the following list of elements:

```
[ 2, 4, 5, 12, 43, 54, 60, 77]
```

At index 0, the value `2` is stored, and at index `7`, the value `77` is stored. Now, assume that we want to find element `2` in the list. How will the two different algorithms go about it?

If we pass this list to the `interpolation search` function, then the `nearest_mid` function will return a value equal to `0` using the formula of `mid_point` computation, which is as follows:

```
mid_point = 0 + [(7-0)/(77-2)] * (2-2)
            = 0
```

As we get the `mid_point` value `0`, we start the interpolation search with the value at index `0`. Just with one comparison, we have found the search term.

On the other hand, the binary search algorithm needs three comparisons to arrive at the search term, as illustrated in *Figure 10.12*:

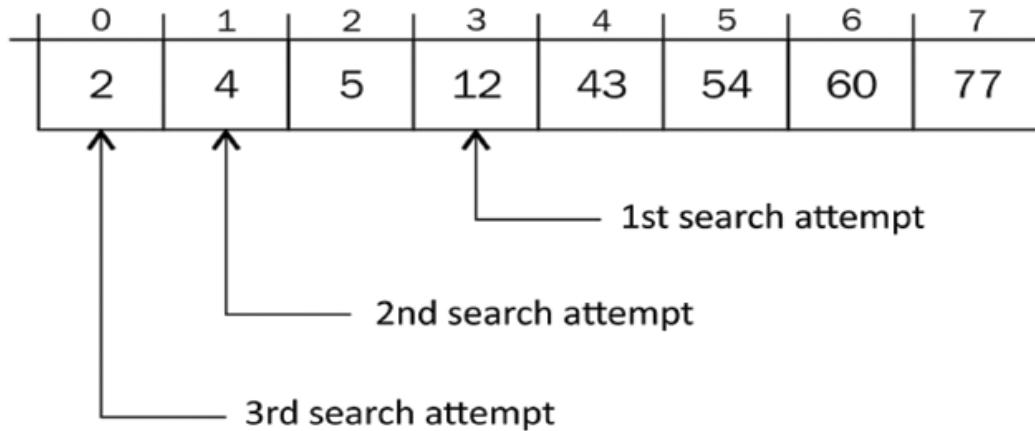


Figure 10.12: Three comparisons are required to search for the item using the binary search algorithm

The first `mid_point` value calculated is `3`. The second `mid_point` value is `1` and the last `mid_point` value where the search term is found is `0`. So, we reach the desired search item in three comparisons whereas in interpolation search we find the desired item on the first attempt.

The interpolation search algorithm works well when the data set is sorted, and uniformly distributed. In this case, the average case time complexity is $O(\log(\log n))$ in which `n` is the length of the array. Moreover, if the dataset is randomized, in that case, the worst-case time complexity of the interpolation search algorithm will be $O(n)$. So, interpolation search may work better than binary search if the given data is uniformly distributed.

Exponential search

Exponential search is another search algorithm that is mostly used when we have large numbers of elements in a list. Exponential search is also known as galloping search and doubling search. The exponential search algorithm works in the following two steps:

1. Given a sorted array of n data elements, we first determine the subrange in the original list where the desired search item may be present
2. Next, we use the binary search algorithm to find out the search value within the subrange of data elements identified in *step 1*

Firstly, in order to find out the subrange of data elements, we start searching for the desired item in the given sorted array by jumping 2^i elements every iteration. Here, i is the value of the index of the array. After each jump, we check if the search item is present between the last jump and the current jump. If the search item is present then we use the binary search algorithm within this subarray, and if it is not present, we move the index to the next location. Therefore, we first find the first occurrence of an exponent i such that the value at index 2^i is greater than the search value.

Then, the 2^i becomes the lower bound and 2^i-1 becomes the upper bound for this range of data elements in which the search value will be present. The exponential search algorithm is defined as follows:

1. First, we check the first element $A[0]$ with the search element.
2. Initialize the index position $i=1$.
3. We check two conditions: (1) if it is the end of the array or not (i.e. $2^i < \text{len}(A)$), and (2) if $A[i] \leq \text{search_value}$). In the first

condition, we check if we have searched the complete list, and we stop if we have reached the end of the list. In the second condition, we stop searching when we reach an element whose value is greater than the search value, because it means the desired element will be present before this index position (since the list is sorted).

4. If either of the above two conditions is true, we move to the next index position by incrementing i in powers of 2.
5. We stop when either of the two conditions of step 3 is satisfied.
6. We apply the binary search algorithm on the range $2^i//2$ to $\min(2^i, \text{len}(A))$.

Let's take an example of a sorted array of elements $A = \{3, 5, 8, 10, 15, 26, 35, 45, 56, 80, 120, 125, 138\}$ in which we want to search for the element 125.

We start with comparing the first element at index $i = 0$, i.e. $A[0]$ with the search element. Since $A[0] < \text{search_value}$, we jump to the next location 2^i with $i = 0$, since $A[2^0] < \text{search_value}$, the condition is true, hence we jump to the next location with $i = 1$ i.e. $A[2^{2^1}] < \text{search_value}$. We again jump to the next location 2^i with $i = 2$, since $A[2^2] < \text{search_value}$, the condition is true. We iteratively jump to the next location until we complete searching the list or the search value is greater than the value at that location, i.e. $A[2^i] < \text{len}(A)$ or $A[2^i] \leq \text{search_value}$. Then we apply the binary search algorithm on the range of the subarray. The complete process for searching a given element in the sorted array using the exponential search algorithm is depicted in *Figure 10.13*:

Assume the item to be searched is 125

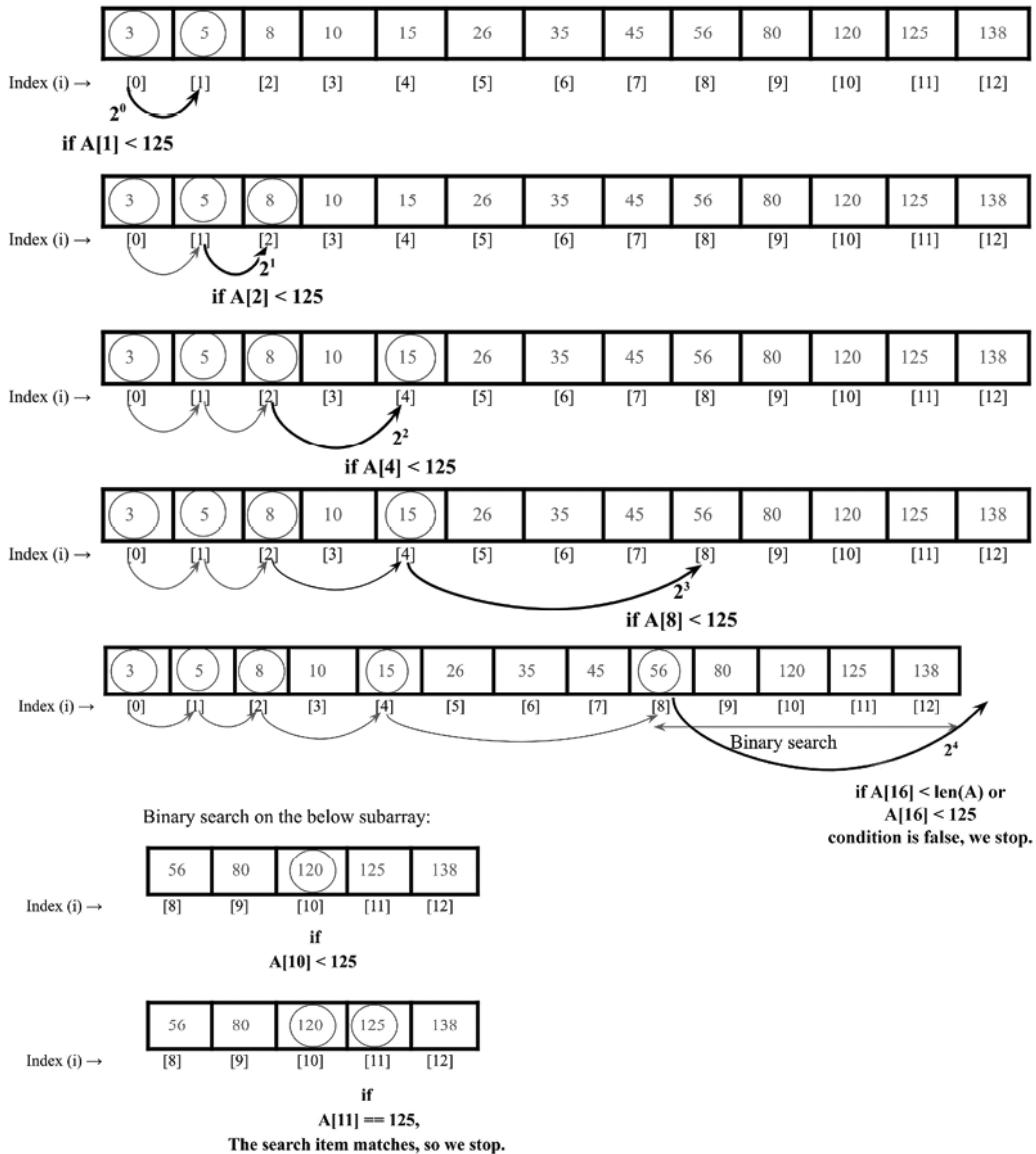


Figure 10.13: Illustration of the exponential search algorithm

Now, let us discuss the implementation of the exponential search algorithm. Firstly, we implement the binary search algorithm, which we have already discussed in the previous section, but for the completeness of this algorithm it is given again as follows:

```
def binary_search_recursive(ordered_list, first_element_index, last_element_index):
    if (last_element_index < first_element_index):
        return None
    else:
        mid_point = first_element_index + ((last_element_index - first_element_index) // 2)
        if ordered_list[mid_point] > term:
            return binary_search_recursive(ordered_list, first_element_index, mid_point)
        elif ordered_list[mid_point] < term:
            return binary_search_recursive(ordered_list, mid_point+1, last_element_index)
        else:
            return mid_point
```

In the above code, given the ordered list of elements, it returns the index of the location where the given data element is found in the list. It returns `None` if the desired element is not found in the list.

Next, we implement the `exponential_search()` method as follows:

```
def exponential_search(A, search_value):
    if (A[0] == search_value):
        return 0
    index = 1
    while index < len(A) and A[index] < search_value:
        index *= 2
    return binary_search_recursive(A, index // 2, min(index, len(A)) - 1)
```

In the above code, firstly, we compare the first element `A[0]` with the search value. If it matches then the index position `0` is returned. If that does not match, we increase the index position to 2^0 , i.e. 1. We check `A[1] < search_value`. Since the condition is true, we jump to the next location 2^1 , i.e. we compare `A[2] < search_value`. Since the condition is true, we move to the next location.

We iteratively increase the index position in the power of 2 until the stop condition is satisfied:

```
while index < len(A) and A[index] < search_value:  
    index *= 2
```

Finally, when the stopping criteria are met, we use the binary search algorithm to search for the desired search value within the subrange as follows:

```
return binary_search_recursive(A, index // 2, min(index, len(A)) -
```

Finally, the `exponential_search()` method returns the index position if the search value is found in the given array; otherwise, `None` is returned.

```
print(exponential_search([1,2,3,4,5,6,7,8,9, 10, 11, 12, 34, 40], 34)
```

The output of the above code snippet is:

```
12
```

In the above output, we get index position `12` for the search item `34` in the given array.

The exponential search is useful for very large-sized arrays. This is better than binary search because instead of performing a binary search on the complete array, we find a subarray in which the

element may be present and then apply binary search, so it reduces the number of comparisons.

The worst-case time complexity of exponential search is $O(\log_2 i)$, where i is the index where the element to be searched is present.

The exponential search algorithm can outperform binary search when the desired search element is present at the beginning of the array.

We can also use exponential search to search in bounded arrays. It can even out-perform binary search when the target is near the beginning of the array, since exponential search takes $O(\log(i))$ time whereas the binary search takes $O(\log n)$ time, where n is the total number of elements. The best-case complexity of exponential search is $O(1)$, when the element is present at the first location of the array.

Next, let us discuss how to decide which search algorithm we should choose for a given situation.

Choosing a search algorithm

Now that we've covered the different types of search algorithms, we can look into which ones work better and in what situations. The binary search and interpolation search algorithms are better in performance compared to both ordered and unordered linear search functions. The linear search algorithm is slower because of the sequential probing of elements in the list to find the search term.

Linear search has a time complexity of $O(n)$. The linear search algorithm does not perform well when the given list of data elements is large.

The binary search operation, on the other hand, slices the list in two anytime a search is attempted. On each iteration, we approach the search term much faster than in a linear strategy. The time complexity yields $O(\log n)$. The binary search algorithm performs well but the drawback of it is that it requires a sorted list of elements. So, if the given data elements are short and unsorted then it is better to use the linear search algorithm.

Interpolation search discards more than half of the list of items from the search space, and this gives it the ability to get to the portion of the list that holds a search term more efficiently. In the interpolation search algorithm, the midpoint is computed in such a way that it gives a higher probability of obtaining the search term faster. The average-case time complexity of interpolation search is $O(\log(\log n))$, whereas the worst-case time complexity of the interpolation search algorithm is $O(n)$. This shows that interpolation search is better than binary search and provides faster searching in most cases.

Therefore, if the list is short and unsorted, then the linear search algorithm is suitable, and if the list is sorted and not very big then the binary search algorithm can be used. Further, the interpolation search algorithm is good to use if the data elements in the list are uniformly distributed. If the list is very large, then the exponential search algorithm and jump search algorithm can be used.

Summary

In this chapter, we discussed the concept of searching for a given element from a list of data elements. We discussed several important search algorithms, such as linear search, binary search, jump search,

interpolation search, and exponential search. The implementations of these algorithms were discussed using Python in detail. We will be discussing sorting algorithms in the next chapter.

Exercise

1. On average, how many comparisons are required in a linear search of n elements?
2. Assume there are eight elements in a sorted array. What is the average number of comparisons that will be required if all the searches are successful and if the binary search algorithm is used?
3. What is the worst-case time complexity of the binary search algorithm?
4. When should the interpolation search algorithm perform better than the binary search algorithm?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



11

Sorting

Sorting means reorganizing data in such a way that it is in ascending or descending order. Sorting is one of the most important algorithms in computer science and is widely used in database-related algorithms. For several applications, if the data is sorted, it can efficiently be retrieved, for example, if it is a collection of names, telephone numbers, or items on a simple to-do list.

In this chapter, we'll study some of the most important and popular sorting techniques, including the following:

- Bubble sort
- Insertion sort
- Selection sort
- Quicksort
- Timsort

Technical requirements

All source code used to explain the concepts of this chapter is provided in the GitHub repository at the following link:

[https://github.com/PacktPublishing/Hands-On-Data-
Structures-and-Algorithms-with-Python-Third-](https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Third-)

Sorting algorithms

Sorting means arranging all the items in a list in ascending or descending order. We can compare different sorting algorithms by how much time and memory space is required to use them.

The time taken by an algorithm changes depending on the input size. Moreover, some algorithms are relatively easy to implement, but may perform poorly with respect to time and space complexity, whereas other algorithms are slightly more complex to implement, but can perform well when sorting longer lists of data. One of the sorting algorithm, merge sort, we have already discussed in *Chapter 3, Algorithm Design Techniques and Strategies*. We will discuss several more sorting algorithms one by one in detail along with their implementation details, starting with the bubble sort algorithm.

Bubble sort algorithms

The idea behind the bubble sort algorithm is very simple. Given an unordered list, we compare adjacent elements in the list, and after each comparison, we place them in the right order according to their values. So, we swap the adjacent items if they are not in the correct order. This process is repeated $n-1$ times for a list of n items.

In each iteration, the largest element of the list is moved to the end of the list. After the second iteration, the second largest element will be placed at the second-to-last position in the list. The same process is repeated until the list is sorted.

Let's take a list with only two elements, {5, 2}, to understand the concept of bubble sort, as shown in *Figure 11.1*:

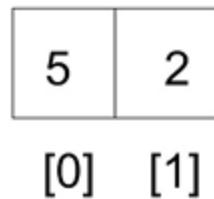


Figure 11.1: Example of bubble sort

To sort this list of two elements, first, we compare 5 and 2; since 5 is greater than 2, it means they are not in the correct order, so we swap these values to put them in the correct order. To swap these two numbers, first, we move the element stored at index 0 in a temporary variable (*step 1* of *Figure 11.2*), then the element stored at index 1 is copied to index 0 (*step 2* of *Figure 11.2*), and finally the first element stored in the temporary variable is stored back at index 1 (*step 3* of *Figure 11.2*). So, first, element 5 is copied to a temporary variable, `temp`. Then, element 2 is moved to index 0. Finally, 5 is moved from `temp` to index 1. The list will now contain the elements as [2, 5]:

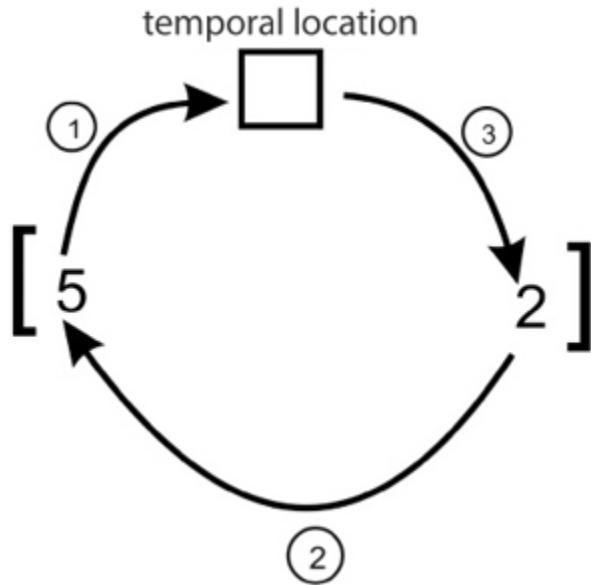


Figure 11.2: Swapping of two elements in bubble sort

The following code will swap the elements of `unordered_list[0]` with `unordered_list[1]` if they are not in the right order:

```
unordered_list = [5, 2]
temp = unordered_list[0]
unordered_list[0] = unordered_list[1]
unordered_list[1] = temp
print(unordered_list)
```

The output of the above code is:

```
[2, 5]
```

Now that we have been able to swap a two-element array, it should be simple to use this same idea to sort a whole list using bubble sort.

Let's consider another example to understand the working of the bubble sort algorithm and sort an unordered list of six elements, such as {45, 23, 87, 12, 32, 4}. In the first iteration, we start comparing

the first two elements, 45 and 23, and we swap them, as 45 should be placed after 23. Then, we compare the next adjacent values, 45 and 87, to see whether they are in the correct order. As 87 is a higher value than 45, we do not need to swap them. We swap two elements if they are not in the correct order.

We can see, in *Figure 11.3*, that after the first iteration of the bubble sort, the largest element, 87, is placed in the last position of the list:

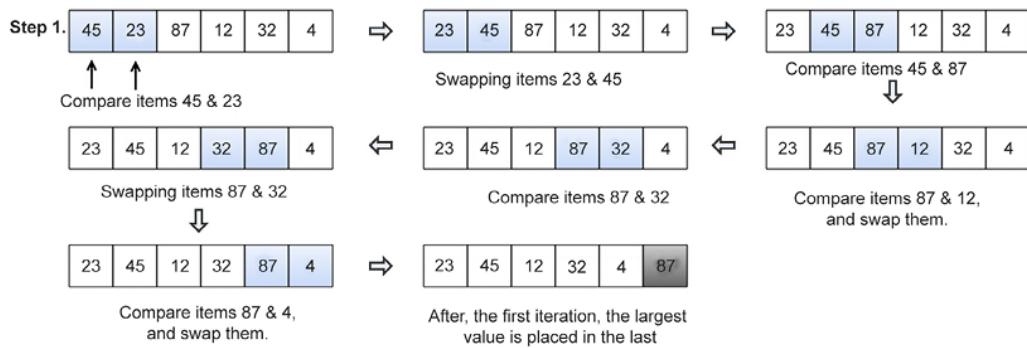


Figure 11.3: Steps of the first iteration to sort an example array using bubble sort

After the first iteration, we just need to arrange the remaining $(n-1)$ elements; we repeat the same process by comparing the adjacent elements for the remaining five elements. After the second iteration, the second largest element, 45, is placed at the second-to-last position in the list, as shown in *Figure 11.4*:

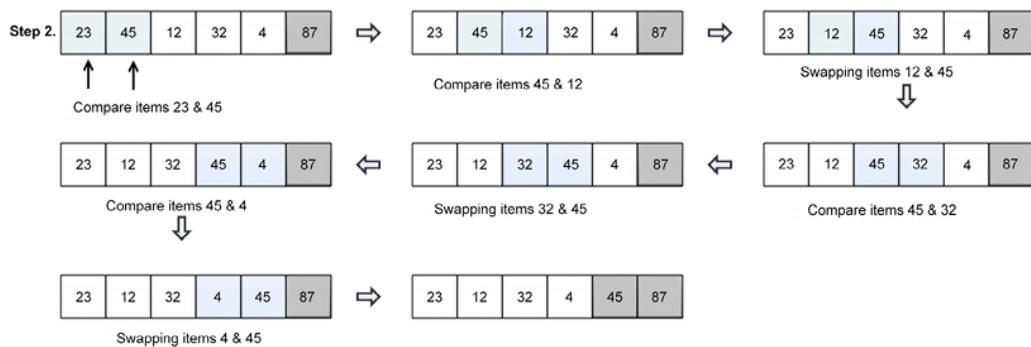


Figure 11.4: Steps of the second iteration to sort an example array using bubble sort

Next, we have to compare the remaining $(n-2)$ elements to arrange them as shown in *Figure 11.5*:

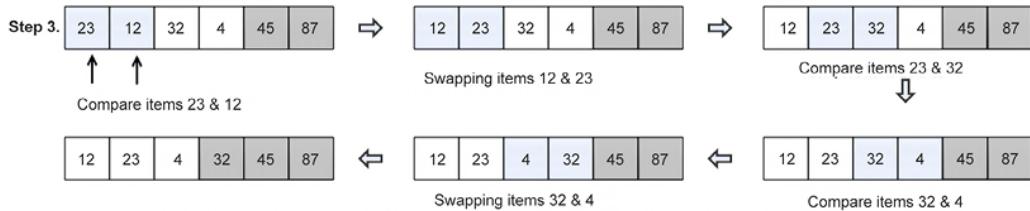


Figure 11.5: Steps of the third iteration to sort an example array using bubble sort

Similarly, we compare the remaining elements to sort them, as well, as shown in *Figure 11.6*:

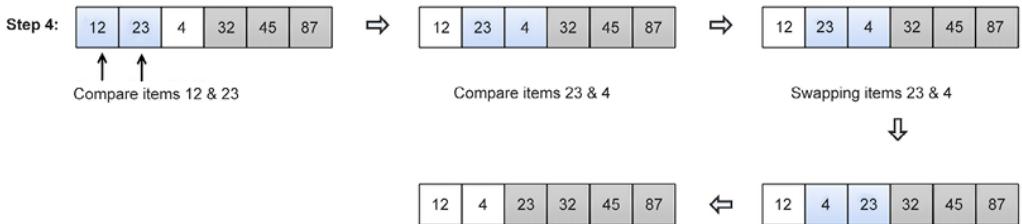


Figure 11.6: Steps of the fourth iteration to sort an example array using bubble sort

Finally, for the last two remaining elements, we place them in the correct order to obtain the final sorted list, as shown in *Figure 11.7*:



Figure 11.7: Steps of the fifth iteration to sort an example array using bubble sort

The complete Python code of the bubble sort algorithm is shown below, and afterward, each step is explained in detail:

```
def bubble_sort(unordered_list):
    iteration_number = len(unordered_list)-1
    for i in range(iteration_number,0,-1):
        for j in range(i):
            if unordered_list[j] > unordered_list[j+1]:
                temp = unordered_list[j]
                unordered_list[j] = unordered_list[j+1]
                unordered_list[j+1] = temp
```

Bubble sort is implemented using a double-nested loop, wherein one loop is inside another loop. In bubble sort, the inner loop repeatedly compares and swaps the adjacent elements in each iteration for a given list, and the outer loop keeps track of how many times the inner loop should be repeated.

Firstly, in the above code, we compute how many times the loop should run to complete all swaps; this is equal to the length of the list minus 1 and could be written as `iteration_number = len(unordered_list)-1`. Here, the `len` function will give the length of the list. We subtract 1 because it gives us exactly the maximum number of iterations to run. The outer loop ensures this and executes for one minus the size of the list.

Further, in the above code, for each iteration, in the inner loop, we compare the adjacent elements using the `if` statement, and we check if the adjacent elements are in the correct order or not. For the first iteration, the inner loop should run for `n` times, for the second iteration, the inner loop should run `n-1` times, and so on. For example, to sort a list of three numbers say `[3, 2, 1]`, the inner loop runs two times, and we need to swap the elements a maximum of two times as shown in *Figure 11.8*:

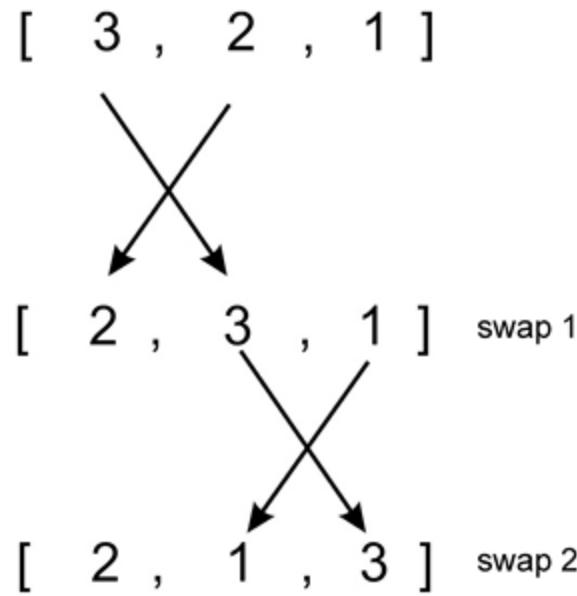


Figure 11.8: Number of swaps in iteration 1 for an example list $[3, 2, 1]$

Further, after the first iteration, in the second iteration, we execute the inner loop once as shown in *Figure 11.9*:

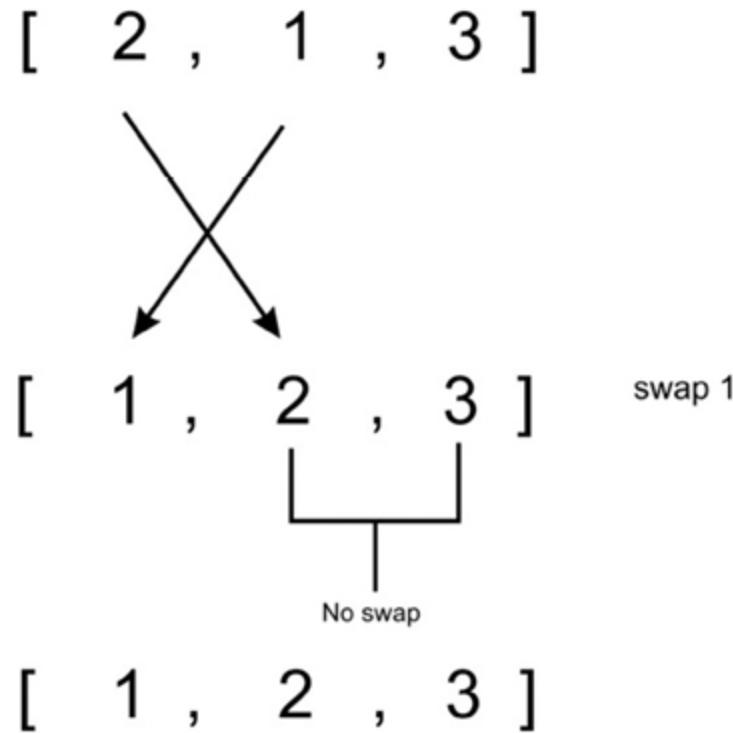


Figure 11.9: Number of swaps in iteration 2 for an example list [3, 2, 1]

The following code snippet can be used to deploy the bubble sort algorithm:

```
my_list = [4,3,2,1]
bubble_sort(my_list)
print(my_list)

my_list = [1,12,3,4]
bubble_sort(my_list)
print(my_list)
```

The output is as follows:

```
[1, 2, 3, 4]
[1, 3, 4, 12]
```

In the worst case, the number of comparisons required in the first iteration will be ($n-1$), in the second, the number of comparisons will be ($n-2$), and in the third iteration it will be ($n-3$), and so on. Therefore, the total number of comparisons required in the bubble sort will be as follows:

$$(n-1) + (n-2) + (n-3) + \dots + 1 = n(n-1)/2$$
$$n(n+1)/2$$
$$O(n^2)$$

The bubble sort algorithm is not an efficient sorting algorithm, as it provides a worst-case runtime complexity of $O(n^2)$, and a best-case complexity of $O(n)$. The worst-case situation occurs when we want to sort the given list in ascending order and the given list is in

descending order, and the best case occurs when the given list is already sorted; in that case, there will not be any need for swapping.

Generally, the bubble sort algorithm should not be used to sort large lists. The bubble sort algorithm is suitable for applications where performance is not important or the length of the given list is short, and moreover, short and simple code is preferred. The bubble sort algorithm performs well on relatively small lists.

Now we shall look into the insertion sort algorithm.

Insertion sort algorithm

The idea of insertion sort is that we maintain two sublists (a sublist is a part of the original larger list), one that is sorted and one that is not sorted, in which elements are added one by one from the unsorted sublist to the sorted sublist. So, we take elements from the unsorted sublist and insert them in the correct position in the sorted sublist, in such a way that this sublist remains sorted.

In the insertion sort algorithm, we always start with one element, taking it to be sorted, and then take elements one by one from the unsorted sublist and place them at the correct positions (in relation to the first element) in the sorted sublist. So, after taking one element from the unsorted sublist and adding it to the sorted sublist, now we have two elements in the sorted sublist. Then, we again take another element from the unsorted sublist, and place it in the correct position (in relation to the two already sorted elements) in the sorted sublist. We repeatedly follow this process to insert all the elements one by one from the unsorted sublist into the sorted sublist. The shaded

elements denote the ordered sublists in *Figure 11.10*, and in each iteration, an element from the unordered sublist is inserted at the correct position in the sorted sublist.

Let's consider an example to understand the working of the insertion sorting algorithm. Let's say; we have to sort a list of six elements: {45, 23, 87, 12, 32, 4}. Firstly, we start with one element, assuming it to be sorted, then take the next element, 23, from the unsorted sublist and insert it at the correct position in the sorted sublist. In the next iteration, we take the third element, 87, from the unsorted sublist, and again insert it into the sorted sublist at the correct position. We follow the same process until all elements are in the sorted sub-list. This whole process is shown in *Figure 11.10*:

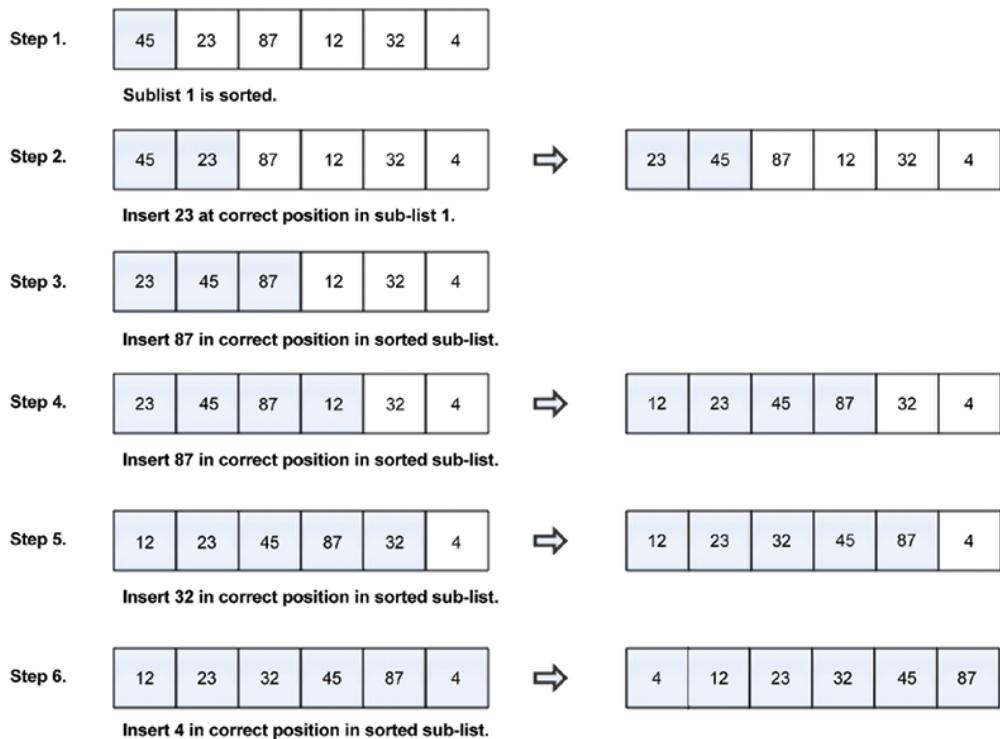


Figure 11.10: Steps to sort example array elements using the insertion sort algorithm

The complete Python code for insertion sort is given below; each statement of the algorithm is further explained in detail with an example:

```
def insertion_sort(unsorted_list):
    for index in range(1, len(unsorted_list)):
        search_index = index
        insert_value = unsorted_list[index]
        while search_index > 0 and unsorted_list[search_index-1] > in
            unsorted_list[search_index] = unsorted_list[search_index-
                search_index -= 1
        unsorted_list[search_index] = insert_value
```

To understand the implementation of the insertion sort algorithm, let's take another example of five elements, {5, 1, 100, 2, 10}, and examine the process with a detailed explanation. Let's consider the following array, as shown in *Figure 11.11*:

5	1	100	2	10
0	1	2	3	4

Figure 11.11: An example array with index positions

The algorithm starts by using a `for` loop to run between the 1 and 4 indices. We start from index 1 because we take the element stored at index 0 to be in the sorted subarray and elements between index 1 to 4 are of the unsorted sublist, as shown in *Figure 11.12*:

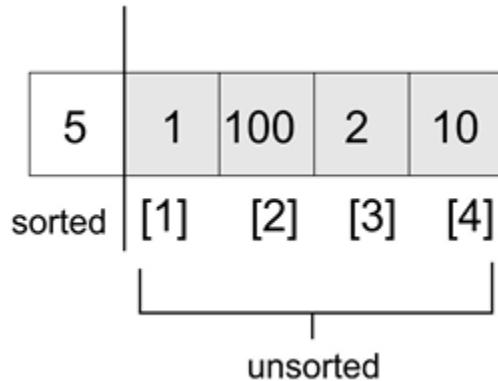


Figure 11.12: Demonstration of sorted and unsorted sublists in insertion sorting

At the start of the execution of the loop, we have the following code snippet:

```
for index in range(1, len(unsorted_list)):
    search_index = index
    insert_value = unsorted_list[index]
```

At the beginning of the execution of each run of the `for` loop, the element at `unsorted_list[index]` is stored in the `insert_value` variable. Later, when we find the appropriate position in the sorted portion of the sublist, `insert_value` will be stored at that index in the sorted sublist. The next code snippet is shown below:

```
while search_index > 0 and unsorted_list[search_index-1] > insert_value:
    unsorted_list[search_index] = unsorted_list[search_index-1]
    search_index -= 1
    unsorted_list[search_index] = insert_value
```

`search_index` is used to provide information to the `while` loop, that is, exactly where to find the next element that needs to be inserted into the sorted sublist.

The `while` loop traverses the list backward, guided by two conditions. First, if `search_index > 0`, then it means that there are more elements in the sorted portion of the list; second, for the `while` loop to run, `unsorted_list[search_index-1]` must be greater than the `insert_value` variable. The `unsorted_list[search_index-1]` array will do either of the following things:

- Point to the element, just before `unsorted_list[search_index]`, before the `while` loop is executed the first time
- Point to one element before `unsorted_list[search_index-1]`, after the `while` loop has been run the first time

In the example list, the `while` loop will be executed because `5 > 1`. In the body of the `while` loop, the element at `unsorted_list[search_index-1]` is stored at `unsorted_list[search_index]`. And, `search_index -= 1` moves the list traversal backward until it holds a value of `0`.

After the `while` loop exits, the last known position of `search_index` (which, in this case, is `0`) now helps us to know where to insert `insert_value`. *Figure 11.13* shows the position of elements after the first iteration:

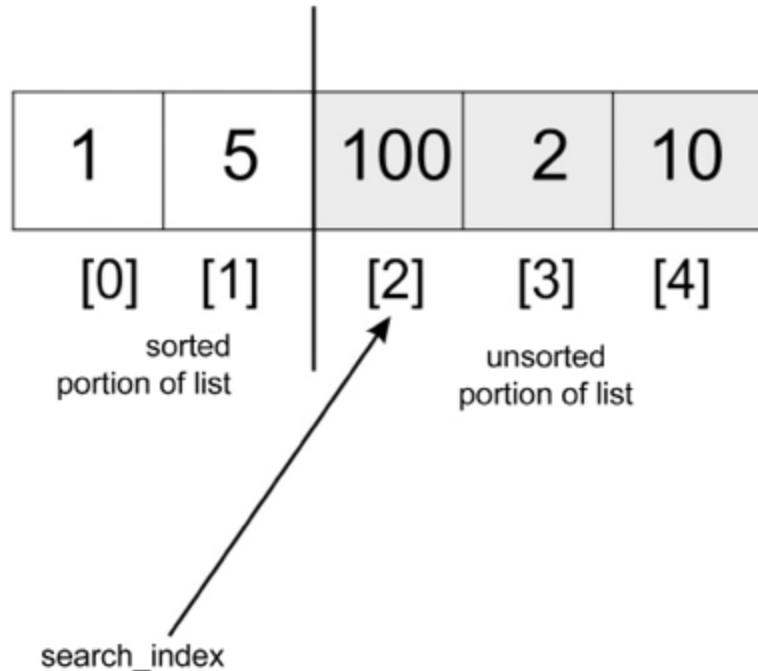


Figure 11.13: Example list position after 1st iteration

On the second iteration of the `for` loop, `search_index` will have a value of `2`, which is the index of the third element in the array. At this point, we start our comparison in the leftward direction (toward index `0`). `100` will be compared with `5`, but because `100` is greater than `5`, the `while` loop will not be executed. `100` will be replaced by itself, because the `search_index` variable never got decremented. As such, `unsorted_list[search_index] = insert_value` will have no effect.

When `search_index` is pointing at index `3`, we compare `2` with `100`, and move `100` to where `2` is stored. We then compare `2` with `5` and move `5` to where `100` was initially stored. At this point, the `while` loop will break and `2` will be stored in index `1`. The array will be partially sorted with the values `[1, 2, 5, 100, 10]`. The preceding step will occur one last time for the list to be sorted.

The following code can be used to create a list of elements, which we can sort using the defined `insertion_sort()` method:

```
my_list = [5, 1, 100, 2, 10]
print("Original list", my_list)
insertion_sort(my_list)
print("Sorted list", my_list)
```

The output of the above code is as follows:

```
Original list [5, 1, 100, 2, 10]
Sorted list [1, 2, 5, 10, 100]
```

The worst-case time complexity of insertion sort is when the given list of elements is sorted in reverse order. In that case, each element will have to be compared with each of the other elements. So, we will need one comparison in the first iteration, two comparisons in the second iteration, and three comparisons in the third iteration, and $(n-1)$ comparisons in the $(n-1)^{\text{th}}$ iteration. Thus, the total number of comparisons are:

```
1 + 2 + 3 .. (n-1)
n(n-1)/2
```

Hence, the insertion sort algorithm gives a worst-case runtime complexity of $O(n^2)$. Furthermore, the best-case complexity of the insertion sort algorithm is $O(n)$, in the situation when the given input list is already sorted in which each element from the unsorted sublist is compared to only the right-most element of the sorted sublist in each iteration. The insertion sort algorithm is good to use

when the given list has a small number of elements, and it is best suited when the input data arrives one by one, and we need to keep the list sorted. Now we are going to take a look at the selection sort algorithm.

Selection sort algorithm

Another popular sorting algorithm is selection sort. The selection sort algorithm begins by finding the smallest element in the list and interchanges it with the data stored at the first position in the list. Thus, it sorts the sublist sorted up to the first element. This process is repeated for $(n-1)$ times to sort n items.

Next, the second smallest element, which is the smallest element in the remaining list, is identified and interchanged with the second position in the list. This makes the initial two elements sorted. The process is repeated, and the smallest element remaining in the list is swapped with the element in the third index on the list. This means that the first three elements are now sorted.

Let's look at an example to understand how the algorithm works. We'll sort the following list of four elements $\{15, 12, 65, 10, 7\}$, as shown in *Figure 11.14*, along with their index positions using the selection sort algorithm:

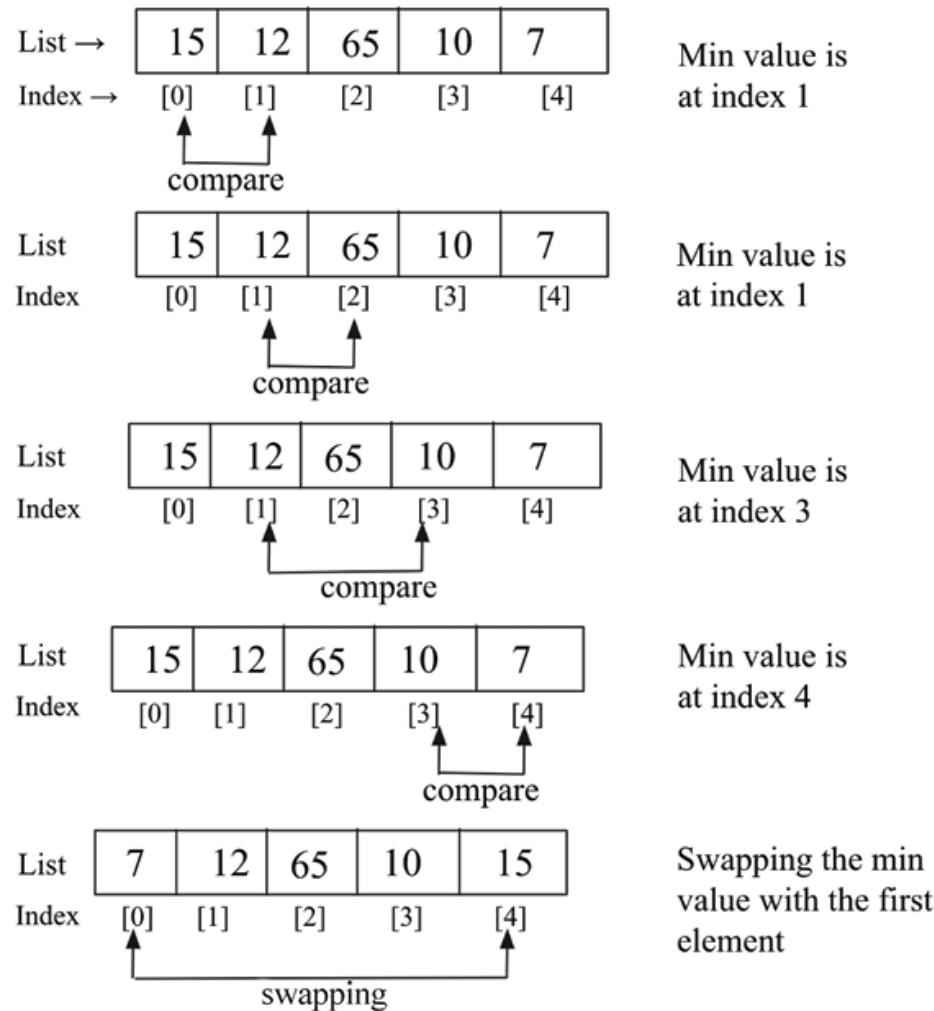


Figure 11.14: Demonstration of the first iteration of the selection sort

In the first iteration of the selection sort, we start at index 0, we search for the smallest item in the list, and when the smallest element is found, it is exchanged with the first data element of the list at index 0. We simply repeat this process until the list is fully sorted. After the first iteration, the smallest element will be placed in the first position in the list.

Next, we start from the second element of the list at index position 1 and search the smallest element in the data list from index position 1 to the length of the list. Once we find the smallest element from this

remaining list of elements, we swap this element with the second element of the list. The step-by-step process of the second iteration of the selection sort is shown in *Figure 11.15*:

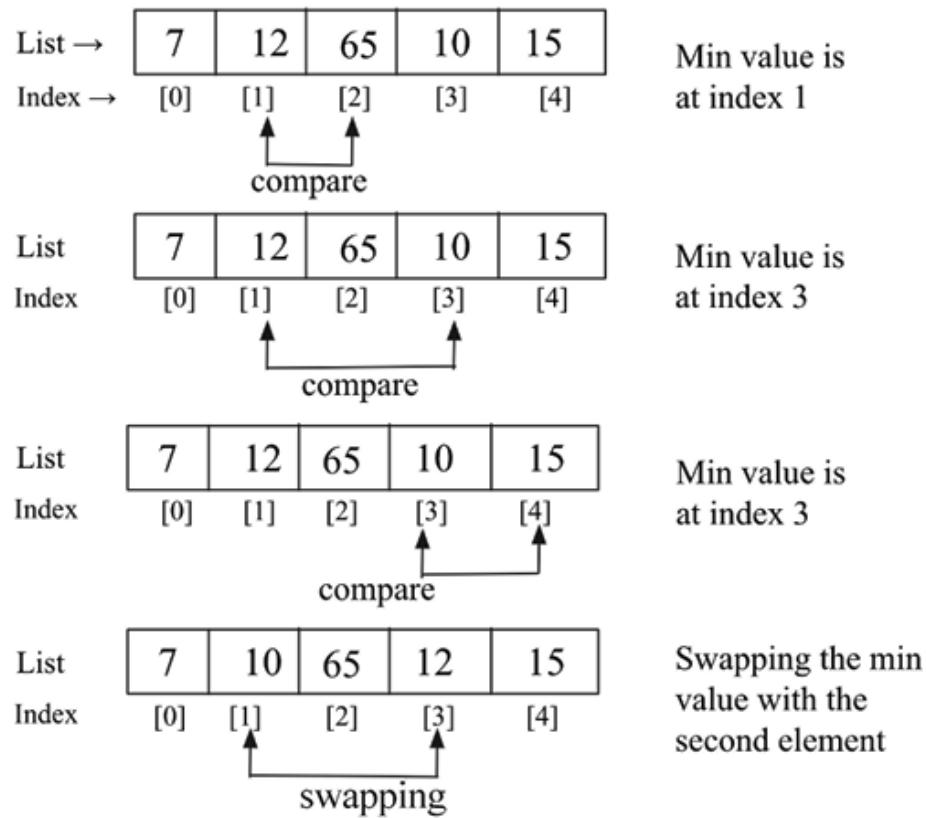


Figure 11.15: Demonstration of the second iteration of the selection sort

In the next iteration, we find out the smallest element in the remaining list in index position 2 to 4 and swap the smallest data element with the data element at index 2 in the second iteration. We follow the same process until we sort the complete list.

The following is an implementation of the selection sort algorithm. The argument to the function is the unsorted list of items we want to put in ascending order of their values:

```
def selection_sort(unsorted_list):
    size_of_list = len(unsorted_list)
    for i in range(size_of_list):
        small = i
        for j in range(i+1, size_of_list):
            if unsorted_list[j] < unsorted_list[small]:
                small = j
        temp = unsorted_list[i]
        unsorted_list[i] = unsorted_list[small]
        unsorted_list[small] = temp
```

In the above code of selection sort, the algorithm begins with the outer `for` loop to go through the list, starting from index `0` to `size_of_list`. Because we pass `size_of_list` to the `range` method, it'll produce a sequence from `0` through to `size_of_list-1`.

Next, we declare a variable `small`, which stores the index of the smallest element. Further, the inner loop is responsible for going through the list and we keep track of the index of the smallest value of the list. Once the index of the smallest element is found, then we swap this element with the correct position in the list.

The following code can be used to create a list of elements and we use the selection sort algorithm to sort the list:

```
a_list = [3, 2, 35, 4, 32, 94, 5, 7]
print("List before sorting", a_list)
selection_sort(a_list)
print("List after sorting", a_list)
```

The output of the above code is as follows:

```
List before sorting [3, 2, 35, 4, 32, 94, 5, 7]
List after sorting [2, 3, 4, 5, 7, 32, 35, 94]
```

In the selection sort, $(n-1)$ comparisons are required in the first iteration, and $(n-2)$ comparisons are required in the second iteration, and $(n-3)$ comparisons are required in the third iteration, and so on. So, the total number of comparisons required is: $(n-1) + (n-2) + (n-3) + \dots + 1 = n(n-1) / 2$, which nearly equals to n^2 . Thus, the worst-case time complexity of the selection sort is $O(n^2)$. The worst-case situation is when the given list of elements is reverse ordered. The selection sorting algorithm gives the best-case runtime complexity of $O(n^2)$. The selection sorting algorithm can be used when we have a small list of elements.

Next, we will discuss the quicksort algorithm.

Quicksort algorithm

Quicksort is an efficient sorting algorithm. The quicksort algorithm is based on the divide-and-conquer class of algorithms, similar to the merge sort algorithm, where we break (divide) a problem into smaller chunks that are much simpler to solve, and further, the final results are obtained by combining the outputs of smaller problems (conquer).

The concept behind quicksorting is partitioning a given list or array. To partition the list, we first select a data element from the given list, which is called a pivot element.

We can choose any element as a pivot element in the list. However, for the sake of simplicity, we'll take the first element in the array as the pivot element. Next, all the elements in the list are compared with this pivot element. At the end of first iteration, all the elements

of the list are arranged in such a way that the elements which are less than the pivot element are arranged to the left of the pivot, that the elements that are greater than the pivot element are arranged to the right of the pivot.

Now, let's understand the working of the quicksort algorithm with an example.

In this algorithm, firstly we partition the given list of unsorted data elements into two sublists in such a way that all the elements on the left side of that partition point (also called a pivot) should be smaller than the pivot, and all the elements on the right side of the pivot should be greater. This means that elements of the left sublist and the right sublist will be unsorted, but the pivot element will be at its correct position in the complete list. This is shown in *Figure 11.16*.

Therefore, after the first iteration of the quicksort algorithm, the chosen pivot point is placed in the list at its correct position, and after the first iteration, we obtain two unordered sublists and follow the same process again on these two sublists. Thus, the quicksort algorithm partitions the list into two parts and recursively applies the quicksort algorithm to these two sublists to sort the whole list:

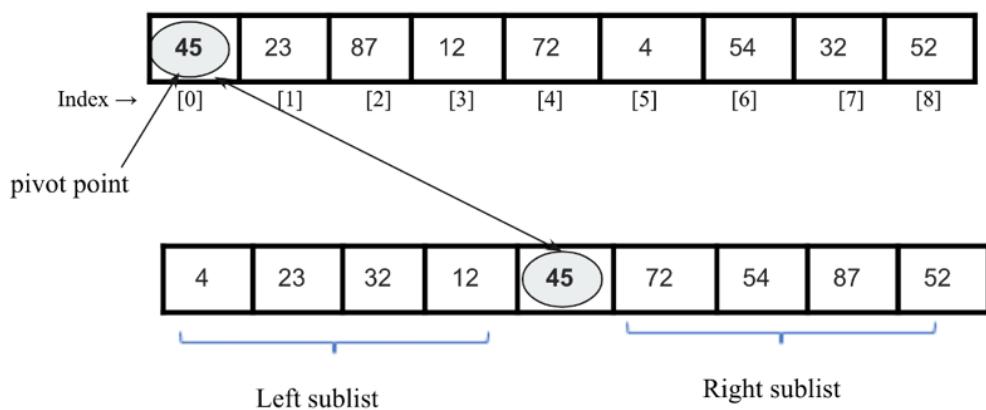


Figure 11.16: Illustration of sublists in quicksort

The quicksort algorithm works as follows:

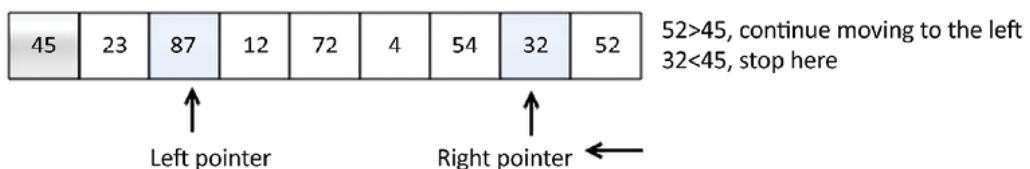
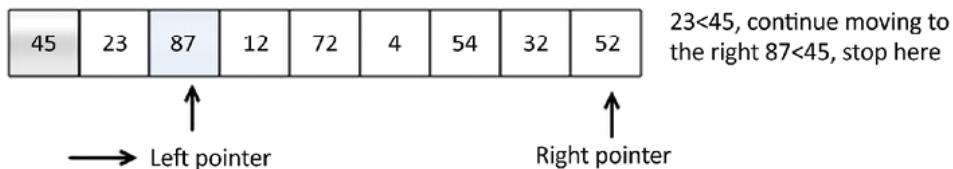
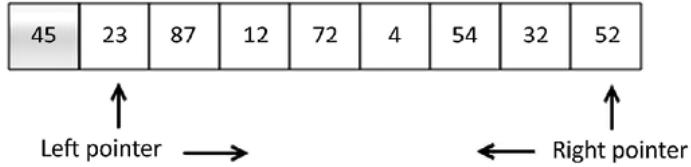
1. We start by choosing a pivot element with which all the data elements are to be compared, and at the end of the first iteration, this pivot element will be placed in its correct position in the list. In order to place the pivot element in its correct position, we use two pointers, a left pointer, and a right pointer. This process is as follows:
 - a. The left pointer initially points to the value at index 1, and the right pointer points to the value at the last index. The main idea here is to move the data items that are on the wrong side of the pivot element. So, we start with the left pointer, moving in a left-to-right direction until we reach a position where the data item in the list has a greater value than the pivot element.
 - b. Similarly, we move the right pointer toward the left until we find a data item less than the pivot element.
 - c. Next, we swap these two values indicated by the left and right pointers.
 - d. We repeat the same process until both pointers cross each other, in other words, until the right pointer index indicates a value less than that of the left pointer index.
2. After each iteration described in *step 1*, the pivot element will be placed at its correct position in the list, and the original list will be divided into two unordered sublists, left and right. We follow the same process (as described in *step 1*) for both these left and right sublists until each of the sublists contains a single element.

- Finally, all the elements will be placed at their correct positions, which will give the sorted list as an output.

Let's take an example of a list of numbers, {45, 23, 87, 12, 72, 4, 54, 32, 52}, to understand how the quicksort algorithm works. Let's assume that the pivot element (also called the pivot point) in our list is the first element, 45. We move the left pointer from index 1 in a rightward direction, and stop when we reach the value 87, because (87>45). Next, we move the right pointer toward the left and stop when we find the value 32, because (32<45). Now, we swap these two values. This process is shown in *Figure 11.17*:

45	23	87	12	72	4	54	32	52
----	----	----	----	----	---	----	----	----

Assume 45 is a pivot point.



Swap 87 and 32

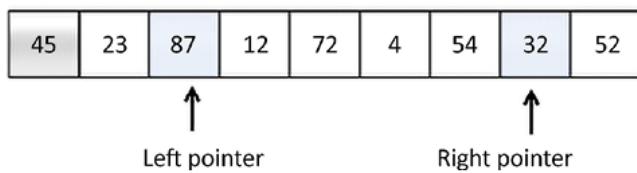


Figure 11.17: An illustrative example of the quicksort algorithm

After that, we repeat the same process and move the left pointer toward the right, and stop when we find the value 72, because $(72 > 45)$. Next, we move the right pointer toward the left and stop when we reach the value 4, because $(4 < 45)$. Now, we swap these two values, because they are on the wrong sides of the pivot value. We repeat the same process and stop once the right pointer index value becomes less than the left pointer index. Here, we find 4 as the

splitting point, and swap it with the pivot value. This is shown in *Figure 11.18*:

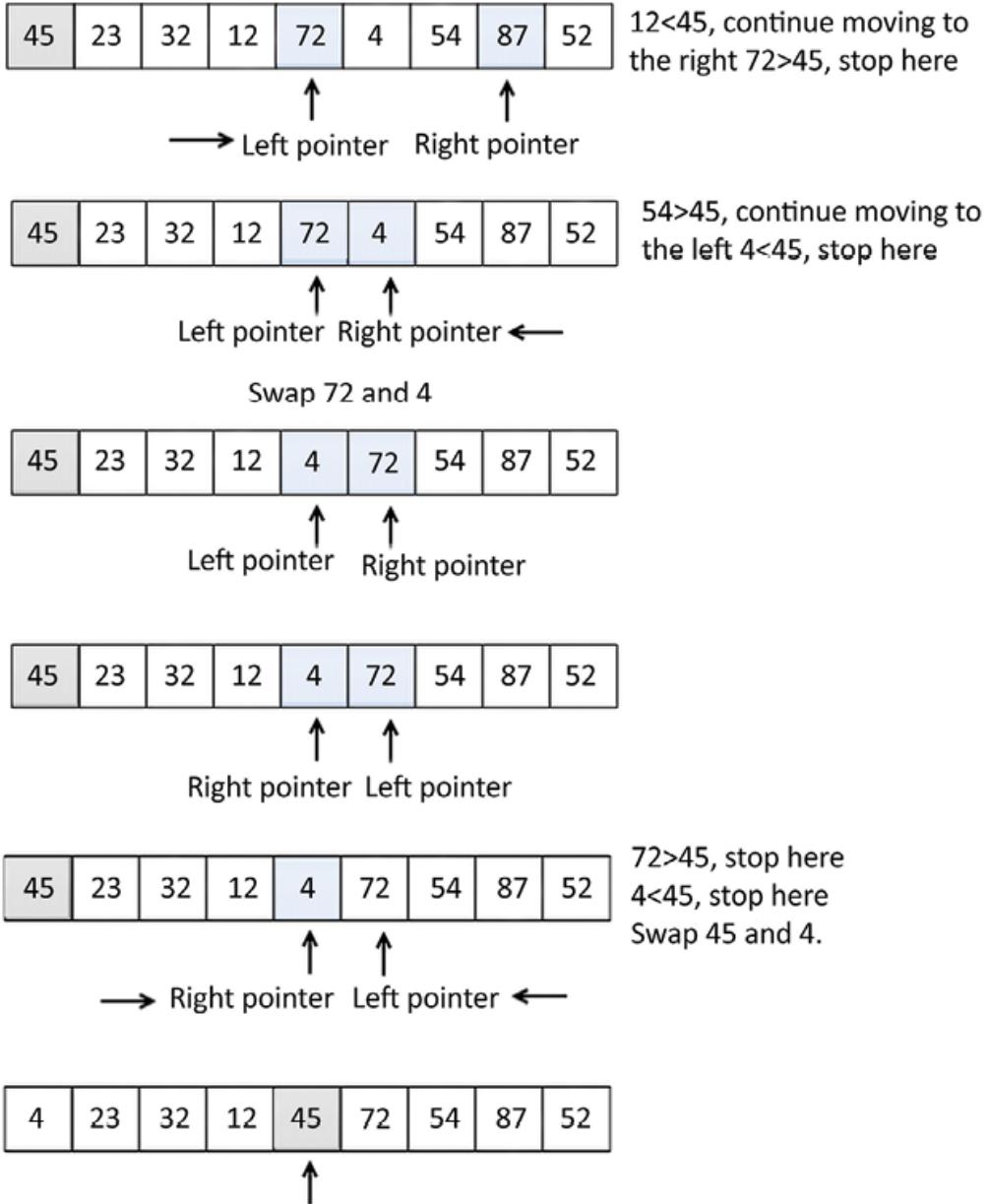


Figure 11.18: An example of the quicksort algorithm (continued)

It can be observed that after the first iteration of the quicksort algorithm, the pivot value 45 is placed at its correct position in the

list.

Now we have two sublists:

1. The sublist to the left of the pivot value, 45, has values less than 45.
2. Another sublist to the right of the pivot value contains values greater than 45. We will apply the quicksort algorithm recursively on these two sublists, and repeat it until the whole list is sorted, as shown in *Figure 11.19*:

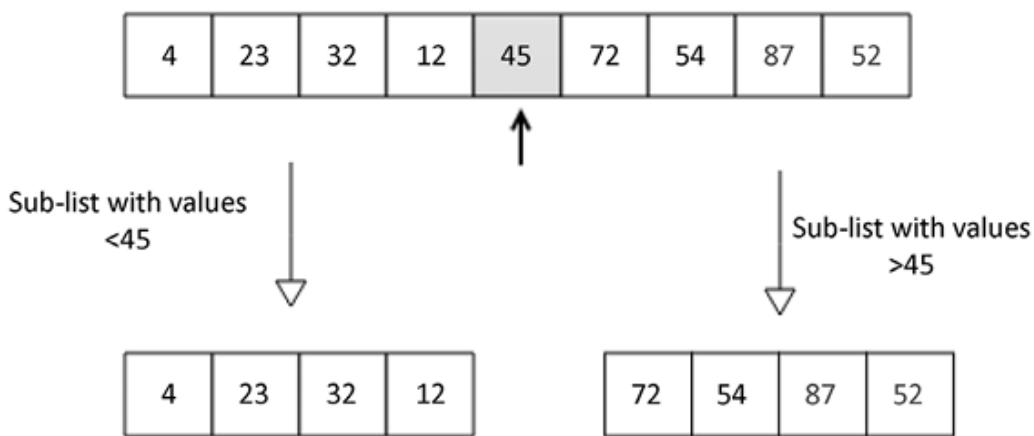


Figure 11.19: After the first iteration of the quicksort algorithm on an example list of elements

We will take a look at the implementation of the quicksort algorithm in the next section.

Implementation of quicksort

The main task of the quicksort algorithm is to first place the pivot element in its correct position so that we divide the given unsorted list into two sublists (left and right sublists); this process is called the partitioning step. The partitioning step is very important in

understanding the implementation of the quicksort algorithm, so we will understand the implementation of the partitioning step first with an example. In this, given a list of elements, all the elements will be arranged in such a way that elements smaller than the pivot element will be on the left side of it, and elements greater than the pivot will be arranged to the right of the pivot element.

Let's look at an example to understand the implementation.

Consider the following list of integers. [43, 3, 20, 89, 4, 77]. We shall partition this list using the partition function:

```
[43, 3, 20, 89, 4, 77]
```

Consider the code of the partition function below; we will discuss each line of this in detail:

```
def partition(unsorted_array, first_index, last_index):
    pivot = unsorted_array[first_index]
    pivot_index = first_index
    index_of_last_element = last_index
    less_than_pivot_index = index_of_last_element
    greater_than_pivot_index = first_index + 1
    while True:
        while unsorted_array[greater_than_pivot_index] < pivot and gr
            greater_than_pivot_index += 1
        while unsorted_array[less_than_pivot_index] > pivot and less_
            less_than_pivot_index -= 1
        if greater_than_pivot_index < less_than_pivot_index:
            temp = unsorted_array[greater_than_pivot_index]
            unsorted_array[greater_than_pivot_index] = unsorted_array
            unsorted_array[less_than_pivot_index] = temp
        else:
            break
    unsorted_array[pivot_index] = unsorted_array[less_than_pivot_inde
    unsorted_array[less_than_pivot_index] = pivot
    return less_than_pivot_index
```

The partition function receives, as its parameters, the indices of the first and last elements of the array that we need to partition.

The value of the pivot is stored in the `pivot` variable, while its index is stored in `pivot_index`. We are not using `unsorted_array[0]`, because when the unsorted array parameter is called with a segment of an array, index `0` will not necessarily point to the first element in that array. The index of the element next to the pivot, that is, the **left pointer**, `first_index + 1`, marks the position where we begin to look for an element in the array. This array is greater than the `pivot` as `greater_than_pivot_index = first_index + 1` suggests. The **right pointer less_than_pivot_index** variable points to the position of the last element in the `less_than_pivot_index = index_of_last_element` list, where we begin the search for the element that is less than the pivot.

Further, at the beginning of the execution of the main `while` loop, the array looks as shown in *Figure 11.20*:

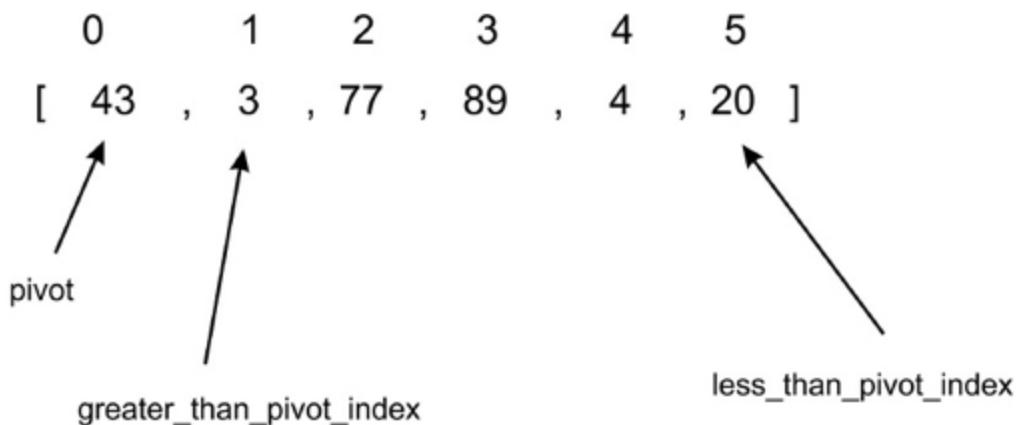


Figure 11.20: Illustration 1 of an example array for the quicksort algorithm

The first inner `while` loop moves one index to the right until it lands on index `2` because the value at that index is greater than `43`. At this

point, the first `while` loop breaks and does not continue. At each test of the condition in the first `while` loop, `greater_than_pivot_index += 1` is evaluated only if the `while` loop's test condition evaluates to `True`. This makes the search for an element, greater than the pivot, progress to the next element on the right.

The second inner `while` loop moves one index at a time to the left, until it lands on index `5`, whose value, `20`, is less than `43`, as shown in *Figure 11.21*:

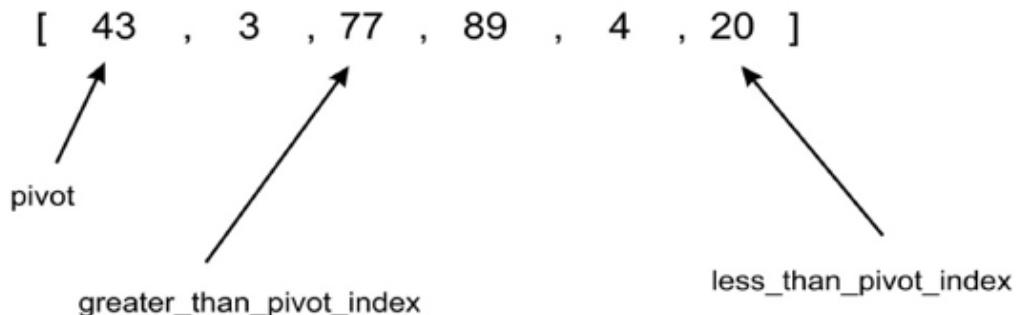


Figure 11.21 Illustration 2 of example array for quicksort algorithm

Next, at this point, neither of the inner `while` loops can be executed any further, and the next code snippet is as shown below:

```
if greater_than_pivot_index < less_than_pivot_index:  
    temp = unsorted_array[greater_than_pivot_index]  
    unsorted_array[greater_than_pivot_index] =  
        unsorted_array[less_than_pivot_index]  
    unsorted_array[less_than_pivot_index] = temp  
else:  
    break
```

Here, since `greater_than_pivot_index < less_than_pivot_index`, the body of the `if` statement swaps the element at those indexes. The `else`

condition breaks the infinite loop any time that `greater_than_pivot_index` becomes greater than `less_than_pivot_index`. In such a condition, it means that `greater_than_pivot_index` and `less_than_pivot_index` have crossed over each other.

The array now looks as shown in *Figure 11.22*:

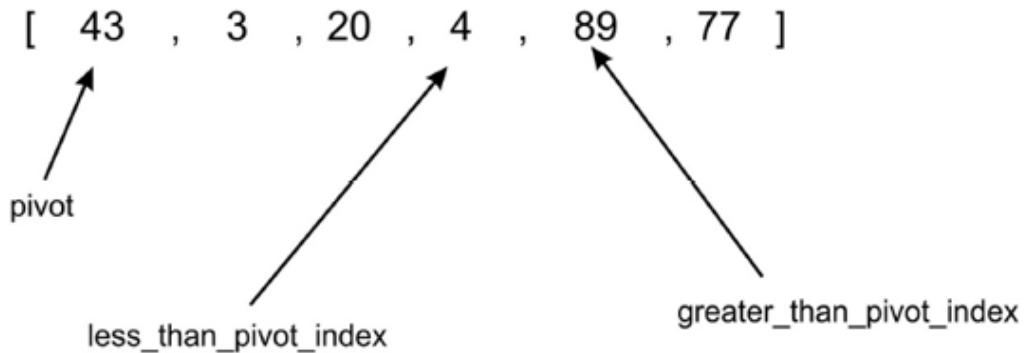


Figure 11.22: Illustration 3 of an example array for the quicksort algorithm

The `break` statement is executed when `less_than_pivot_index` is equal to `3` and `greater_than_pivot_index` is equal to `4`.

As soon as we exit the `while` loop, we interchange the element at `unsorted_array[less_than_pivot_index]` with that of `less_than_pivot_index`, which is returned as the index of the pivot:

```
unsorted_array[pivot_index]=unsorted_array[less_than_pivot_index]
unsorted_array[less_than_pivot_index]=pivot
return less_than_pivot_index
```

Figure 11.23 shows how the code interchanges `4` with `43` as the last step in the partitioning process:

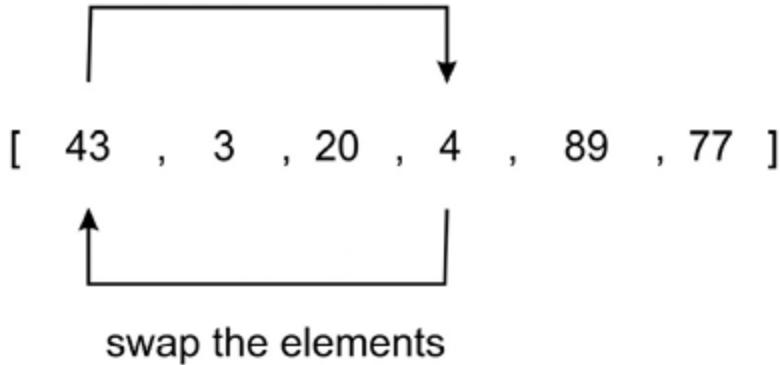


Figure 11.23: Illustration 4 of an example array for the quicksort algorithm

To recap, the first time the `quick_sort` function was called, it was partitioned at the element at index `0`. After the return of the partitioning function, we obtain the array in the order of `[4, 3, 20, 43, 89, 77]`.

As you can see, all elements to the right of element `43` are greater than `43`, while those to the left are smaller. Thus, the partitioning is complete.

Using the split point `43` with index `3`, we will recursively sort the two subarrays, `[4, 30, 20]` and `[89, 77]`, using the same process we just went through.

The body of the main `quick_sort` function is as follows:

```
def quick_sort(unsorted_array, first, last):
    if last - first <= 0:
        return
    else:
        partition_point = partition(unsorted_array, first, last)
        quick_sort(unsorted_array, first, partition_point-1)
        quick_sort(unsorted_array, partition_point+1, last)
```

The `quick_sort` function is quite simple; initially, the `partition` method is called, which returns the partition point. This partition point is in the `unsorted_array` array where all elements to the left are less than the pivot value, and all elements to the right are greater. We print the state of `unsorted_array` immediately after the partition progress to see the status of the array after every call.

After the first partition, the first subarray `[4, 3, 20]` will be done; the partition of this subarray will stop when `greater_than_pivot_index` is at index `2` and `less_than_pivot_index` is at index `1`. At that point, the two markers are said to have crossed. Because `greater_than_pivot_index` is greater than `less_than_pivot_index`, further execution of the `while` loop will cease. Pivot `4` will be exchanged with `3`, while index `1` is returned as the partition point.

We can use the below code snippet to create a list of elements, and use the quicksort algorithm to sort it:

```
my_array = [43, 3, 77, 89, 4, 20]
print(my_array)
quick_sort(my_array, 0, 5)
print(my_array)
```

The output of the above code is as follows:

```
[43, 3, 77, 89, 4, 20]
[3, 4, 20, 43, 77, 89]
```

In the quicksort algorithm, the partition algorithm takes $O(n)$ time. As the quicksort algorithm follows the **divide and conquer** paradigm, it takes $O(\log n)$ time; therefore, the overall average-case

runtime complexity of the quicksort algorithm is $O(n) * O(\log n) = O(n \log n)$. The quicksort algorithm gives a worst-case runtime complexity of $O(n^2)$. The worst-case complexity for the quicksort algorithm would be when it selects the worst pivot point every time, and one of the partitions always has a single element. For example, if the list is already sorted, the worst-case complexity would occur if the partition picks the smallest element as a pivot point. When worst-case complexity does occur, the quicksort algorithm can be improved by using the randomized quicksort. The quicksort algorithm is efficient when the given list of elements is very long; it works better compared to the other aforementioned algorithms for sorting in such situations.

Timsort algorithm

Timsort is used as the default standard sorting algorithm in all Python versions ≥ 2.3 . The Timsort algorithm is an optimal algorithm for real-world long lists that is based on a combination of the merge sort and insertion sort algorithms. The Timsort algorithm utilizes the best of both algorithms; insertion sort works best when the array is sorted partially and its size is small, and the merge method of the merge sort algorithm works fast when we have to combine small, sorted lists.

The main concept of the Timsort algorithm is that it uses the insertion sort algorithm to sort small blocks (also known as chunks) of data elements, and then it uses the merge sort algorithm to merge all the sorted chunks. The main characteristic of the Timsort algorithm is that it takes advantage of already-sorted data elements

known as “natural runs,” which occur very frequently in real-world data.

The Timsort algorithm works as follows:

1. Firstly, we divide the given array of data elements into a number of blocks which are also known as a run.
2. We generally use 32 or 64 as the size of the run as it is suitable for Timsort; however, we can use any other size that can be computed from the length of the given array (say N). The `minrun` is the minimum length of each run. The size of the `minrun` can be computed by following the given principles:
 - a. The `minrun` size should not be too long as we use the insertion sort algorithm to sort these small blocks, which performs well for short lists of elements.
 - b. The length of the run should not be very short; in that case, it will result in a greater number of runs, which will make the merging algorithm slow.
 - c. Since merge sort works best when we have the number of runs as a power of 2, it would be good if the number of runs that compute as N/minrun are a power of 2.
3. For example, if we take a run size of 32, then the number of runs will be `(size_of_array/32)`; if this is a power of 2, then the merge process will be very efficient.
4. Sort each of the runs one by one using the insertion sort algorithm.
5. Merge all the sorted runs one by one using the merge method of the merge sort algorithm.
6. After each iteration, we double the size of the merged subarray.

Let's take an example to understand the working of the Timsort algorithm. Let's say we have the array [4, 6, 3, 9, 2, 8, 7, 5]. We sort it using the Timsort algorithm; here, for simplicity, we take the size of the run as 4. So, we divide the given array into two runs, run 1 and run 2. Next, we sort run 1 using the insertion sort algorithm, and then we sort run 2 using the insertion sort algorithm. Once we have all the runs sorted, we use the merge method of the merge sort algorithm to obtain the final complete sorted list. The complete process is shown in *Figure 11.24*:

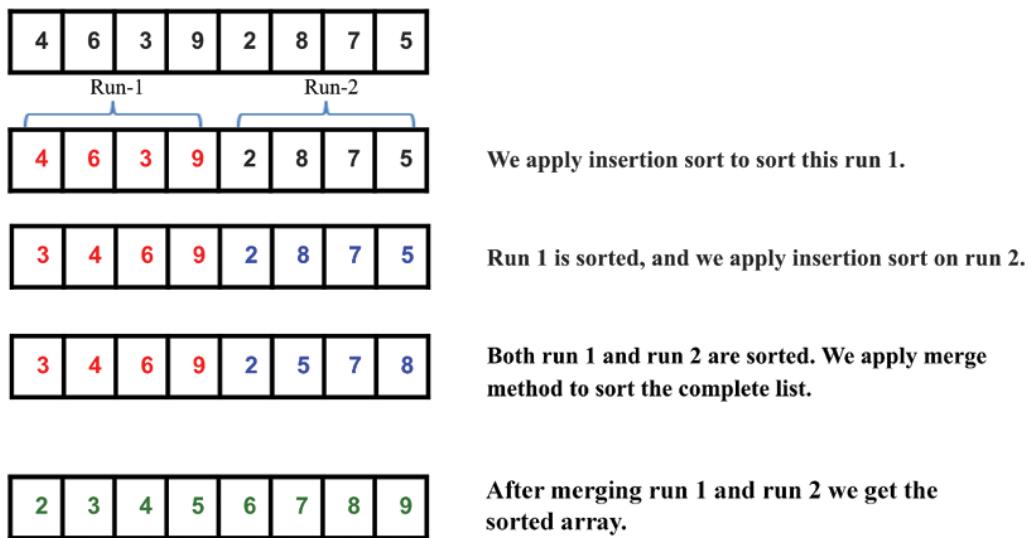


Figure 11.24: Illustration of an example array for the Timsort algorithm

Next, let's discuss the implementation of the Timsort algorithm. Firstly, we implement the insertion sort algorithm and the merge method of the merge sort algorithm. The insertion sort algorithm has already been discussed in detail in previous sections. For completeness, it is given below again:

```
def Insertion_Sort(unsorted_list):
    for index in range(1, len(unsorted_list)):
```

```
    search_index = index
    insert_value = unsorted_list[index]
    while search_index > 0 and unsorted_list[search_index-1] > in-
        unsorted_list[search_index] = unsorted_list[search_index-
            search_index -= 1
        unsorted_list[search_index] = insert_value
    return unsorted_list
```

In the above, the insertion sort method is responsible in sorting the run. Next, we present the merge method of the merge sort algorithm; this has been discussed in detail in *Chapter 3, Algorithm Design Techniques and Strategies*. This `Merge()` function is used to merge the sorted runs, and it is defined as follows:

```
def Merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list
```

Next, let's discuss the Timsort algorithm. Its implementation is given below. Let's understand it bit by bit:

```
def Tim_Sort(arr, run):
    for x in range(0, len(arr), run):
        arr[x : x + run] = Insertion_Sort(arr[x : x + run])
    runSize = run
    while runSize < len(arr):
        for x in range(0, len(arr), 2 * runSize):
            arr[x : x + 2 * runSize] = Merge(arr[x : x + runSize], ar
                runSize = runSize * 2
```

In the above implementation, we firstly pass two parameters, the array that is to be sorted and the size of the run. Next, we use insertion sort to sort the individual subarrays by run size in the below code snippet:

```
for x in range(0, len(arr), run):
    arr[x : x + run] = Insertion_Sort(arr[x : x + run])
```

In the above code for the example list [4, 6, 3, 9, 2, 8, 7, 5], let's say run size is 2, so we will have a total of four blocks/chunks/runs, and after exiting this loop, the array will be like this: [4, 6, 3, 9, 2, 8, 5, 7], indicating that all runs of size 2 are sorted. After that we initialize `runSize` and we iterate until `runSize` becomes equal to the array length. So, we use the merge method for combining the sorted small lists:

```
runSize = run
while runSize < len(arr):
    for x in range(0, len(arr), 2 * runSize):
        arr[x : x + 2 * runSize] = Merge(arr[x : x + runSize], ar
            runSize = runSize * 2
```

In the above code, the `for` loop is using the `Merge` function for merging the runs of size `runSize`. For the example above, the `runSize` is `2`. In the first iteration, it will merge the left run from index `(0 to 1)` and right run from index `(2 to 3)` to form a sorted array from index `(0 to 3)`, and the array will become `[3, 4, 6, 9, 2, 8, 5, 7]`.

Further, in the second iteration, it will merge the left run from index `(4 to 5)` and the right run from index `(6 to 7)` to form a sorted run from index `(4 to 7)`. After the second iteration the `for` loop will terminate and the array will become `[3, 4, 6, 9, 2, 5, 7, 8]`, which indicates the array has been sorted from index `(0 to 3)` and `(4 to 7)`.

Now we update the size of the run as `2*runSize` and we repeat the same process for the updated `runSize`. So now, `runSize` is `4`. Now, in the first iteration, it will merge the left run (index `0 to 3`) and right run (index `4 to 7`) to form a sorted array from index `(0 to 7)` and after this the `for` loop will terminate and the array will become `[2, 3, 4, 5, 6, 7, 8, 9]`, which indicates the array has been sorted.

Now, `runSize` will become equal to the array length so the `while` loop will terminate, and at last, we will be left with the sorted array.

We can use the below code snippet to create a list, and then sort the list using the Timsort algorithm:

```
arr = [4, 6, 3, 9, 2, 8, 7, 5]
run = 2
Tim_Sort(arr, run)
print(arr)
```

The output of the above code is as follows:

[2,3,4,5,6,7,8,9]

Timsort is very efficient for real-world applications since it has a worst-case complexity of $O(n \log n)$. Timsort is the best choice for sorting, even if the length of the given list is short. In that case, it uses the insertion sort algorithm, which is very fast for smaller lists, and the Timsort algorithm works fast for long lists due to the merge method; hence, the Timsort algorithm is a good choice for sorting due to its adaptability for sorting arrays of any length in real-world usage.

A comparison of the complexities of different sorting algorithms is given in the following table:

Algorithm	worst-case	average-case	best-case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Timsort	$O(n \log n)$	$O(n \log n)$	$O(n)$

Table 11.1: Comparing the complexity of different sorting algorithms

Summary

In this chapter, we have explored important and popular sorting algorithms that are very useful for many real-world applications. We have discussed the bubble sort, insertion sort, selection sort, quicksort, and Timsort algorithms, along with explaining their implementation in Python. In general, the quicksort algorithm performs better than the other sorting algorithms, and the Timsort algorithm is the best choice to use in real-world applications.

In the next chapter, we will discuss selection algorithms.

Exercise

1. If an array `arr = [55, 42, 4, 31]` is given and bubble sort is used to sort the array elements, then how many iterations will be required to sort the array?
 - a. 3
 - b. 2
 - c. 1
 - d. 0
2. What is the worst-case complexity of bubble sort?
 - a. $O(n \log n)$
 - b. $O(\log n)$
 - c. $O(n)$
 - d. $O(n^2)$
3. Apply quicksort to the sequence (56, 89, 23, 99, 45, 12, 66, 78, 34). What is the sequence after the first phase, and what pivot is

the first element?

- a. 45, 23, 12, 34, 56, 99, 66, 78, 89
- b. 34, 12, 23, 45, 56, 99, 66, 78, 89
- c. 12, 45, 23, 34, 56, 89, 78, 66, 99
- d. 34, 12, 23, 45, 99, 66, 89, 78, 56

4. Quicksort is a _____

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

5. Consider a situation where a swap operation is very costly.
Which of the following sorting algorithms should be used so
that the number of swap operations is minimized?

- a. Heap sort
- b. Selection sort
- c. Insertion sort
- d. Merge sort

6. If the input array `A = {15, 9, 33, 35, 100, 95, 13, 11, 2, 13}` is given, using selection sort, what would the order of the array be after the fifth swap? (Note: it counts regardless of whether they exchange places or remain in the same position.)

- a. 2, 9, 11, 13, 13, 95, 35, 33, 15, 100
- b. 2, 9, 11, 13, 13, 15, 35, 33, 95, 100
- c. 35, 100, 95, 2, 9, 11, 13, 33, 15, 13
- d. 11, 13, 9, 2, 100, 95, 35, 33, 13, 13

7. What will be the number of iterations to sort the elements {44,
21, 61, 6, 13, 1} using insertion sort?

- a. 6
- b. 5
- c. 7
- d. 1

8. How will the array elements A= [35, 7, 64, 52, 32, 22] look after the second iteration, if the elements are sorted using insertion sort?

- a. 7, 22, 32, 35, 52, 64
- b. 7, 32, 35, 52, 64, 22
- c. 7, 35, 52, 64, 32, 22
- d. 7, 35, 64, 52, 32, 22

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



12

Selection Algorithms

One interesting set of algorithms related to finding elements in an unordered list of items is selection algorithms. Given a list of elements, selection algorithms are used to find the k^{th} smallest or largest element from the list. So given a list of data elements and a number (k), the aim is to find the k^{th} smallest or largest element. The simplest case of selection algorithms is to find the minimum or maximum data element from the list. However, sometimes, we may need to find the k^{th} smallest or largest element in the list. The simplest way is to first sort the list using any sorting algorithm, and then we can easily obtain the k^{th} smallest (or largest) element. However, when the list is very large, then it is not efficient to sort the list to get the k^{th} smallest or largest element. In that case, we can use different selection algorithms that can efficiently produce the k^{th} smallest or largest element.

In this chapter, we will cover the following topics:

- Selection by sorting
- Randomized selection
- Deterministic selection

We will start with the technical requirements, and then we will discuss selection by sorting.

Technical requirements

All of the source code that's used in this chapter is provided in the given GitHub link:

<https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Third-Edition/tree/main/Chapter12>.

Selection by sorting

Items in a list may undergo statistical inquiries such as finding the mean, median, and mode values. Finding the mean and mode values does not require the list to be ordered. However, to find the median in a list of numbers, the list must first be ordered. Finding the median requires you to find the element in the middle position of the ordered list. In addition, this can be used when we want to find the k^{th} smallest item in the list. To find the k^{th} smallest number in an unordered list of items, an obvious method is to first sort the list, and after sorting, you can rest assured that the element at index 0 will hold the smallest element in the list. Likewise, the last element in the list will hold the largest element in the list.

For more information on how to order data items within a list, see *Chapter 11, Sorting*. However, in order to obtain a k^{th} smallest element from the list, it is not a good solution to apply a sorting algorithm to a long list of elements to obtain the minimum or maximum or k^{th} smallest or largest value from the list since sorting is quite an expensive operation. Thus, if we need to find out the k^{th} smallest or largest element from a given list, there is no need to sort

the complete list as we have other methods that we can use for this purpose. Let's discuss better techniques to find the k^{th} smallest element without having to sort the list in the first place, starting with randomized selection.

Randomized selection

The randomized selection algorithm is used to obtain the k^{th} smallest number that is based on the quicksort algorithm; the randomized selection algorithm is also known as quickselect. In *Chapter 11, Sorting*, we discussed the quicksort algorithm. The quicksort algorithm is an efficient algorithm to sort an unordered list of items. To summarize, the quicksort algorithm works as follows:

1. It selects a pivot.
2. It partitions the unsorted list around the pivot.
3. It recursively sorts the two halves of the partitioned list using steps 1 and 2.

One important fact about quicksort is that after every partitioning step, the index of the pivot does not change, even after the list becomes sorted. This means that after each iteration, the selected pivot value will be placed in its correct position in the list. This property of quicksort enables us to obtain the k^{th} smallest number without sorting the complete list. Let's discuss the randomized selection method, which is also known as the quickselect algorithm, to obtain the k^{th} smallest element from a list of n data items.

Quickselect

The quickselect algorithm is used to obtain the k^{th} smallest element in an unordered list of items. It is based on the quicksort algorithm, in which we recursively sort the elements of both the sublists from the pivot point. In each iteration, the pivot value reaches the correct position in the list, which divides the list into two unordered sublists (left and right sublists), where the left sublist has smaller values as compared to the pivot value, and the right sublist has greater values compared to the pivot value. Now, in the case of the quickselect algorithm, we recursively call the function only for the sublist that has the k^{th} smallest element.

In the quickselect algorithm, we compare the index of the pivot point with the k value to obtain the k^{th} smallest element from the given unordered list. There will be three cases in the quickselect algorithm, as follows:

1. If the index of the pivot point is smaller than k , then we are sure that the k^{th} smallest value will be present on the right-hand sublist of the pivot point. So we only recursively call the quickselect function for the right sublist.
2. If the index of the pivot point is greater than k , then it is obvious that the k^{th} smallest element will be present on the left-hand side of the pivot point. So we only recursively look for the i^{th} element in the left sublist.
3. If the index of the pivot point is equal to k , then it means that we have found out the k^{th} smallest value, and we return it.

Let's understand the working of the quickselect algorithm with an example. Consider a list of elements, {45, 23, 87, 12, 72, 4, 54, 32},

`52}`. We can use the quickselect algorithm to find the third smallest element in this list.

We start the algorithm by selecting a pivot value, that is, `45`. Here we are choosing the first element as the pivot element for simplicity; however, any other element can be chosen as a pivot element. After the first iteration of the algorithm, the pivot value moves to its correct position in the list, which in this example is at index 4 (the index is starting from 0). Next, we check the condition `k<pivot` point (that is, `2<4`). Case- 2 is applicable, so we only consider the left sublist, and recursively call the function. Here, we compare the index of the pivot value (that is, `4`) with the value of `k` (that is, the 3rd position or at index 2).

Next, we take the left sublist and select the pivot point (that is, `4`). After the run, the `4` is placed in its correct position (that is, the 0th index). As the index of the pivot is less than the value of `k`, we consider the right sublist.

Similarly, we take `23` as the pivot point, which is also placed in its correct position. Now, when we compare the index of the pivot point and the value of `k`, they are equal, which means we have found the 3rd smallest element, and it will be returned. The complete step-by-step process to find the 3rd smallest element is shown in *Figure 12.1*:

45	23	87	12	72	4	54	32	52
----	----	----	----	----	---	----	----	----

Assume, 45 is the pivot point

4	23	32	12	45	72	54	87	52
---	----	----	----	----	----	----	----	----

After first iteration, 45 is placed at its correct position.

Sub-list with values >45



Sub-list with values >45

4	23	32	12
---	----	----	----

72	54	87	52
----	----	----	----

Now, consider only the left sublist as the value of $k <$ index of the split point, i.e. ($2 < 4$)

4	23	32	12
---	----	----	----

Assuming 4 as the pivot point.



23	32	12
----	----	----

Now, 4 is placed as its correct position, in other words, at first place.
Now consider the right sublist.



12	23	32
----	----	----

Now considering 23 as the pivot point, after the partitioning, it is placed at its correct position, i.e. at 3rd position, which is required, so it will be returned.

Figure 12.1: Step-by-step demonstration of the quickselect algorithm

Let's discuss the implementation of the `quick_select` method. It is defined as follows:

```
def quick_select(array_list, start, end, k):
    split = partition(array_list, start, end)
    if split == k:
        return array_list[split]
    elif split < k:
        return quick_select(array_list, split + 1, end, k)
```

```
    else:  
        return quick_select(array_list, start, split-1, k)
```

In the above code, the `quick_select` function takes the complete array, the index of the first element of the list, the index of the last element, and the k^{th} element specified by value `k` as parameters. The value of `k` maps with the index that the user is searching for, meaning the k^{th} smallest number in the list.

Initially, we use the `partition()` method (which is defined and discussed in detail in *Chapter 11, Sorting*) to place the selected pivot point in such a way that it divides the given list of elements in the left sublist and the right sublist, in which the left sublist has data elements that are smaller than the pivot value, and right subtree has data elements that are greater than the pivot value. The `partition()` method is called `split = partition(array_list, start, end)` and returns the `split` index. Here, the `split` index is the position where the pivot element is placed in the array, and `(start, end)` is the starting and ending indices of the list. Once we get the split point, we compare the `split` index with the required value of `k` to find out whether we have reached the position of the k^{th} smallest data item or whether the required k^{th} smallest element will be on the left sublist or the right sublist. These three conditions are as follows:

1. If the `split` is equal to the value of `k`, then it means that we have reached the k^{th} smallest data item in the list.
2. If the `split` is less than `k`, then it means that the k^{th} smallest item should exist or be found between `split+1` and `right`.
3. If the `split` is greater than `k`, then it means that the k^{th} smallest item should exist or be found between `left` and `split-1`.

In the preceding example, a split point occurs at index 4 (index starting from 0). If we are searching for the 3rd smallest number, then since $4 < 2$ yields `false`, a recursive call to the right sublist is made using `quick_select(array_list, left, split-1, k)`.

Here, for the completeness of this algorithm, the `partition()` method is given as follows:

```
def partition(unsorted_array, first_index, last_index):
    pivot = unsorted_array[first_index]
    pivot_index = first_index
    index_of_last_element = last_index
    less_than_pivot_index = index_of_last_element
    greater_than_pivot_index = first_index + 1
    while True:
        while unsorted_array[greater_than_pivot_index] < pivot and gr
            greater_than_pivot_index += 1
        while unsorted_array[less_than_pivot_index] > pivot and less_
            less_than_pivot_index -= 1
        if greater_than_pivot_index < less_than_pivot_index:
            temp = unsorted_array[greater_than_pivot_index]
            unsorted_array[greater_than_pivot_index] = unsorted_array
            unsorted_array[less_than_pivot_index] = temp
        else:
            break
    unsorted_array[pivot_index] = unsorted_array[less_than_pivot_index]
    unsorted_array[less_than_pivot_index] = pivot
    return less_than_pivot_index
```

We can use the below code snippet to find out the k^{th} smallest element using the `quicksort` algorithm for a given array.

```
list1 = [3,1,10, 4, 6, 5]
print("The 2nd smallest element is", quick_select(list1, 0, 5, 1))
print("The 3rd smallest element is", quick_select(list1, 0, 5, 2))
```

The output of the above code is as follows:

```
The 2nd smallest element is 3  
The 3rd smallest element is 4
```

In the above code, we get the 2nd and 3rd smallest elements from the given list of elements. The worst-case performance of a randomized selection-based `quick-select` algorithm is $O(n^2)$.

In the above implementation of the `partition()` method, we use the first element of the list as the pivot element for simplicity, but any element can be chosen from the list as the pivot element. A good pivot element is one that divides the list into almost equal halves. Therefore, it is possible to improve the performance of the quickselect algorithm by selecting the split point more efficiently in linear time with the worst-case complexity of $O(n)$. We discuss how to do this in the next section using deterministic selection.

Deterministic selection

Deterministic selection is an algorithm for finding out the k^{th} item in an unordered list of elements. As we have seen in the `quickselect` algorithm, we select a random “pivot” element that partitions the list into two sublists and calls itself recursively for one of the two sublists. In a deterministic selection algorithm, we choose a pivot element more efficiently instead of taking any random pivot element.

The main concept of the deterministic algorithm is to select a pivot element that produces a good split of the list, and a good split is one

that divides the list into two halves. For instance, a good way to select a pivot element would be to choose the median of all the values. But we will need to sort the elements in order to find out the median element, which is not efficient, so instead, we try to find a way to select a pivot element that divides the list roughly in the middle.

The median of medians is a method that provides us with the approximate median value, that is, a value close to the actual median for a given unsorted list of elements. It divides the given list of elements in such a way that in the worst case, at least 3 out of 10 ($3/10$) of the list will be below the pivot element, and at least 3 out of 10 of the elements will be above the list.

Let's take an example to understand this. Let's say we have a list of 15 elements: `{11, 13, 12, 111, 110, 15, 14, 16, 113, 112, 19, 18, 17, 114, 115}.`

Next, we divide it into groups of 5 elements and sort them as follows: `{{11, 12, 13, 110, 111}, {14, 15, 16, 112, 113}, {17, 18, 19, 114, 115}}.`

Next, we compute the median of each of these groups, and they are `13, 16, and 19`, respectively. Further, the median of these median values `{13, 16, 19}` is `16`. This is the median of medians for the given list. Here, we can see that 5 elements are smaller, and `9` elements are greater than the pivot element. When we select this median of the median as a pivot element, the list of `n` elements is divided in such a way that at least `3n/10` elements are smaller than the pivot element.

The deterministic algorithm to select the k^{th} smallest element works as follows:

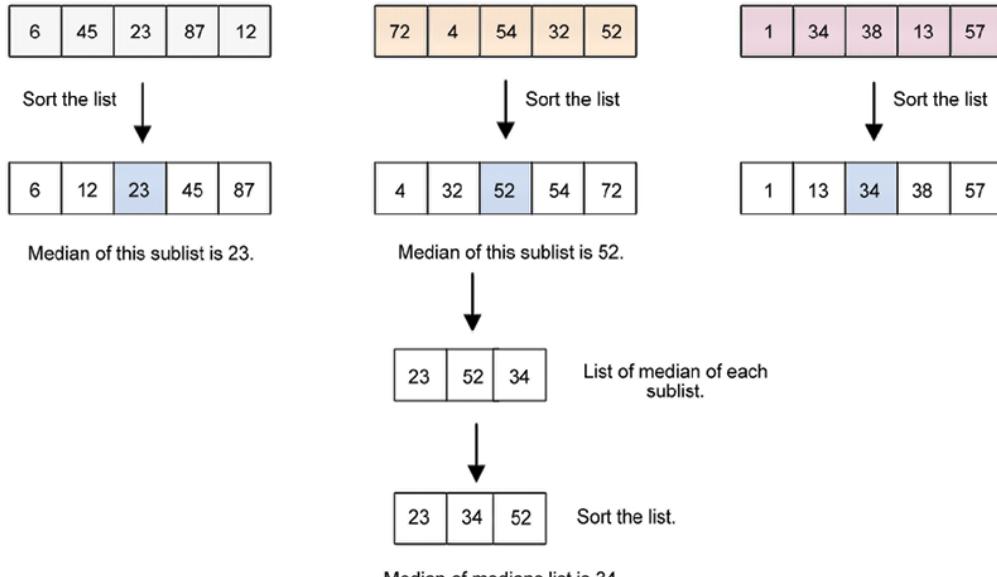
1. Split the list of unordered items into groups of five elements each (the number 5 is not mandatory; it can be changed to any other number, for example, 8)
2. Sort these groups (in general, we use insertion sort for this purpose) and find the median of all these groups
3. Recursively, find the median of the medians obtained from these groups; let's say that is point **p**
4. Using this point **p** as the pivot element, recursively call the partition algorithm similar to quickselect to find out the k^{th} smallest element

Let's consider an example list of 15 elements to understand the working of the deterministic algorithm to find out the 3rd smallest element from the list, as shown in *Figure 12.2*. First, we divide the list into groups of 5 elements each, and then we sort these groups/sublists. Once we have sorted the lists, we find out the median of the sublists. For this example, items **23**, **52**, and **34** are the medians of these three sublists, as shown in *Figure 12.2*.

Next, we sort the list of medians for all the sublists. Further, we find out the median of this list, that is, the median of the median, which is **34**. This median of medians is used to select the partition/pivot point for the whole list. Further, we divide the given list using this pivot element to partition the list into 2 sublists, placing the given pivot element at its correct position in the list. For this example, the index of the pivot element is 7 (index starting from 0; this is shown in *Figure 12.2*).).

6	45	23	87	12	72	4	54	32	52	1	34	38	13	57
---	----	----	----	----	----	---	----	----	----	---	----	----	----	----

break the whole list into sublists of 5 elements each.



Use 34 as the pivot value, and apply the partition algorithm.
Now, we obtain the following list where 34 is placed at its correct position in the list.

6	13	23	1	12	32	4	34	72	52	87	54	38	45	57
---	----	----	---	----	----	---	----	----	----	----	----	----	----	----

Since we wish to obtain the 3rd smallest element, the index of the pivot value is 7 ($2 < 7$), so we recursively run the algorithm on the left sublist.

Figure 12.2: Step-by-step procedure for the deterministic selection algorithm

The index of the pivot element is greater than the k^{th} value, and hence, we recursively call the algorithm on the left sublist to obtain the required k^{th} smallest element.

Next, we will discuss the implementation of the deterministic selection algorithm.

Implementation of the deterministic selection algorithm

To implement the deterministic algorithm for determining the k^{th} smallest value from the list, we start implementing the updated `partition()` method, which divides the list where we select the pivot element using the median of medians method. Let's now understand the code for the `partition` function:

```
def partition(unsorted_array, first_index, last_index):
    if first_index == last_index:
        return first_index
    else:
        nearest_median = median_of_medians(unsorted_array[first_index])
        index_of_nearest_median = get_index_of_nearest_median(unsorted_array)
        swap(unsorted_array, first_index, index_of_nearest_median)

        pivot = unsorted_array[first_index]
        pivot_index = first_index
        index_of_last_element = last_index
        less_than_pivot_index = index_of_last_element
        greater_than_pivot_index = first_index + 1

        ## This while Loop is used to correctly place pivot element at its correct position
        while 1:
            while unsorted_array[greater_than_pivot_index] < pivot and greater_than_pivot_index < index_of_last_element:
                greater_than_pivot_index += 1
            while unsorted_array[less_than_pivot_index] > pivot and less_than_pivot_index > first_index:
                less_than_pivot_index -= 1

            if greater_than_pivot_index < less_than_pivot_index:
                temp = unsorted_array[greater_than_pivot_index]
                unsorted_array[greater_than_pivot_index] = unsorted_array[less_than_pivot_index]
                unsorted_array[less_than_pivot_index] = temp
            else:
                break

        unsorted_array[pivot_index]=unsorted_array[less_than_pivot_index]
        unsorted_array[less_than_pivot_index]=pivot
    return less_than_pivot_index
```

In the above code, we implement the partition method, which is very similar to what we did in the quickselect algorithm. In the quickselect algorithm, we used a random pivot element (for simplicity, the first element of the list), but in the deterministic selection algorithm, we select the pivot element using the median of medians. The partition method divides the list into two sublists – the left and right sublists, in which the left sublist has elements that are smaller than the pivot element, and the right sublist has elements that are greater than the pivot element. The main benefit of using the pivot element with the median of medians is that it, in general, divides the list into almost two halves.

At the start of the code, firstly, in the `if-else` condition, we check the length of the given list of elements. If the length of the list is 1, then we return the index of that element, so if the `unsorted_array` has only one element, `first_index` and `last_index` will be equal. Therefore, `first_index` is returned. And, if the length is greater than 1, then we call the `median_of_medians()` method to compute the median of medians of the list passed to this method with the starting and ending indices as `first_index` and `last_index`. The return median of medians value is stored in the `nearest_median` variable.

Now, let's understand the code of the `median_of_medians()` method. It is given as follows:

```
def median_of_medians(elems):
    sublists = [elems[j:j+5] for j in range(0, len(elems), 5)]
    medians = []
    for sublist in sublists:
        medians.append(sorted(sublist)[int(len(sublist)/2)])
    if len(medians) <= 5:
        return sorted(medians)[int(len(medians)/2)]
```

```
    else:  
        return median_of_medians(medians)
```

In the above code of the `median_of_medians` function, recursion is used to compute the median of medians for the given list. The function begins by splitting the given list, `elems`, into groups of five elements each. As discussed earlier in the deterministic algorithm, we divide the given list into groups of 5 elements. Here, we choose 5 elements since it mostly performs well. However, we could have used any other number as well. This means that if `elems` contains 100 items, there will be 20 groups that are created by the `sublists = [elems[j:j+5] for j in range(0, len(elems), 5)]` statement, with each containing a maximum of five elements.

After creating sublists of five elements each, we create an empty array, `medians`, that stores the medians of each of the five-element arrays (i.e., `sublists`). Further, the `for` loop iterates over the list of lists inside `sublists`. Each sublist is sorted, the median is found, and it is stored in the `medians` list. The `medians.append(sorted(sublist)[len(sublist)//2])` statement will sort the list and obtain the element stored in its middle index. The `medians` variable becomes the median list of all the sublists of which there are five elements in each sublist. In this implementation, we use an existing sorting function of Python; it will not impact the performance of the algorithm due to the list's small size.

Thereafter, the next step is to recursively compute the median of medians, which we will use as a pivot element. It is important to note here that the length of the median array can itself be a large array because if the original length of the array is n , then the length

of the median array will be $n/5$, and sorting this may be time-consuming in itself. Hence, we check the length of the `medians` array, and if it is less than 5, we sort the `medians` list and return the element located in its middle index. If, on the other hand, the size of the list is greater than five, we recursively call the `median_of_medians` function again, supplying it with the list of the medians stored in `medians`. Finally, the function returns the median of medians of the given list of elements.

Let's take another example to better understand the concept of the median of medians with the following list of numbers:

```
[2, 3, 5, 4, 1, 12, 11, 13, 16, 7, 8, 6, 10, 9, 17, 15, 19, 20, 18, 2]
```

We can break this list down into groups of five elements, each with the `sublists = [elems[j:j+5] for j in range(0, len(elems), 5)]` code statement, in order to obtain the following list:

```
[[2, 3, 5, 4, 1], [12, 11, 13, 16, 7], [8, 6, 10, 9, 17], [15, 19, 20]]
```

Each of the five-element lists will be sorted as follows:

```
[[1, 2, 3, 5, 5], [7, 11, 12, 13, 16], [6, 8, 9, 10, 17], [15, 18, 19]]
```

Next, we obtain their medians to produce the following list:

```
[3, 12, 9, 19, 22]
```

We sort the above list:

```
[3, 9, 12, 19, 22]
```

Since the list is five elements in size, we only return the median of the sorted list, which is 12 in this case. Otherwise, if the length of this array had been greater than 5, we would have made another call to the `median_of_median` function.

Once we have the median of the median value, we need to find out its index in the given list. We write the `get_index_of_nearest_median` function for this purpose. This function takes the starting and ending indices of the list indicated by the `first` and `last` parameters:

```
def get_index_of_nearest_median(array_list, first, last, median):
    if first == last:
        return first
    else:
        return array_list.index(median)
```

Next in the partition method, we swap the median of medians value with the first element of the list, that is, we swap

`index_of_nearest_median` with `first_index` of the `unsorted_array` using the `swap` function:

```
swap(unsorted_array, first_index, index_of_nearest_median)
```

The `utility` function to swap two array elements is shown here:

```
def swap(array_list, first, index_of_nearest_median):
    temp = array_list[first]
```

```
array_list[first] = array_list[index_of_nearest_median]
array_list[index_of_nearest_median] = temp
```

We swap these two elements. The rest of the implementation is quite similar to what we discussed in the `quick_select` algorithm. Now, we have the median of the median for the given list, which is stored in `first_index` of the unsorted list.

Now, the rest of the implementation is similar to the partition method of the `quick_select` algorithm and also the quicksort algorithm, which is discussed in detail in *Chapter 11, Sorting*. For the completeness of the algorithm here, we discuss this again.

We consider the first element as a pivot element, and we take two pointers, that is, left and right. The left pointer moves from the left to the right direction in the list to keep elements that are smaller than the pivot element on the left hand side of the pivot element. It is initialized with the second element of the list, that is, `first_index+1`, whereas the right pointer moved from the right to the left direction, which maintains the list in a way that elements greater than the pivot element are on the right-hand side of the pivot element in the list. It is initialized with the last element of the list. So we have two variables `less_than_pivot_index` (the right pointer) and `greater_than_pivot_index` (the left pointer) in which `less_than_pivot_index` is initialized with `index_of_last_element` and `greater_than_pivot_index` with `first_index + 1`:

```
less_than_pivot_index = index_of_last_element
greater_than_pivot_index = first_index + 1
```

Next, we move the left and right pointers in such a way that after one iteration, the pivot element is placed in its correct position in the list. That means it divides the list into two sublists such that the left sublist has all the elements that are smaller than the pivot element, and the right sublist has elements greater than the pivot element. We do this with these three steps given below:

```
## This while loop is used to correctly place pivot element at its correct position
while 1:
    while unsorted_array[greater_than_pivot_index] < pivot and greater_than_pivot_index < last_index:
        greater_than_pivot_index += 1
    while unsorted_array[less_than_pivot_index] > pivot and less_than_pivot_index >= 0:
        less_than_pivot_index -= 1
    if greater_than_pivot_index < less_than_pivot_index:
        temp = unsorted_array[greater_than_pivot_index]
        unsorted_array[greater_than_pivot_index] = unsorted_array[less_than_pivot_index]
        unsorted_array[less_than_pivot_index] = temp
    else:
        break
```

1. The first while loop will move `greater_than_pivot_index` to the right side of the array until the element pointed out by `greater_than_pivot_index` is less than the pivot element and `greater_than_pivot_index` is less than `last_index`:

```
while unsorted_array[greater_than_pivot_index] < pivot and greater_than_pivot_index < last_index:
    greater_than_pivot_index += 1
```

2. In the second `while` loop, we'll be doing the same thing but for the `less_than_pivot_index` in the array. We'll move `less_than_pivot_index` to the left direction until the element pointed out by `less_than_pivot_index` is greater than the pivot

element and `less_than_pivot_index` is greater than or equal to `first_index`:

```
while unsorted_array[less_than_pivot_index] > pivot and less_thar
```

3. Now, we check if `greater_than_pivot_index` and `less_than_pivot_index` have crossed each other or not. If `greater_than_pivot_index` is still less than `less_than_pivot_index` (that is, we have not found the correct position for the pivot element yet), we swap the elements indicated by `greater_than_pivot_index` and `less_than_pivot_index`, and then we will repeat the same three steps again. If they have crossed each other, that means we have found the correct position for the pivot element, and we will break from the loop:

```
if greater_than_pivot_index < less_than_pivot_index:  
    temp = unsorted_array[greater_than_pivot_index]  
    unsorted_array[greater_than_pivot_index] = unsorted_array[les  
    unsorted_array[less_than_pivot_index] = temp  
else:  
    break
```

After exiting the loop, the variable `less_than_pivot_index` will point to the correct index of the pivot, so we will just swap the values that are present at `less_than_pivot_index` and `pivot_index`:

```
unsorted_array[pivot_index]=unsorted_array[less_than_pivot_index]  
unsorted_array[less_than_pivot_index]=pivot
```

Finally, we will simply return the pivot index, which is stored in the variable `less_than_pivot_index`.

After the partition method, the pivot element reaches its correct position in the list. Thereafter, we recursively call the partition method to one of the sublists (the left sublist or the right sublist) depending on the required value of `k` and the pivot element position to find out the k^{th} smallest element. This process is the same as the quickselect algorithm.

The implementation of the deterministic select algorithm is given as follows:

```
def deterministic_select(array_list, start, end, k):
    split = partition(array_list, start, end)
    if split == k:
        return array_list[split]
    elif split < k:
        return deterministic_select(array_list, split + 1, end, k)
    else:
        return deterministic_select(array_list, start, split-1, k)
```

As you may have observed, the implementation of the deterministic selection algorithm looks exactly the same as the quickselect algorithm. The only difference between the two is how we select the pivot element; apart from that, everything is the same.

After the initial `array_list` has been partitioned by the selected pivot element, which is the median of medians of the list, a comparison with the k^{th} element is made:

1. If the index of the split point, that is, `split`, is equal to the required value of `k`, it means that we have found the required

k^{th} smallest element.

2. If the index of the split point, the, `split` is less than the required value of k , then a recursive call to the right sublist is made as `deterministic_select(array_list, split + 1, right, k)`. This will look for the k^{th} element on the right-hand side of the array.

3. Otherwise, if the split index is greater than the value of k , then the function call to the left sublist is made as

`deterministic_select(array_list, left, split-1, k).`

The following code snippet can be used to create a list and further use the deterministic algorithm to find out the k^{th} smallest element from the list:

```
list1= [2, 3, 5, 4, 1, 12, 11, 13, 16, 7, 8, 6, 10, 9, 17, 15, 19, 20
print("The 6th smallest element is", deterministic_select(list1, 0, 1
```

The output of the above code is as follows.

```
The 6th smallest element is 6
```

In the output of the above code, we have the 6^{th} smallest element from a given list of 25 elements. The deterministic selection algorithm improves the quickselect algorithm by using the median of medians element as a pivot point for selecting the k^{th} smallest element from a list. It improves performance because the median of medians method finds out the estimated median in linear time, and when this estimated median is used as a pivot point in the quickselect algorithm, the worst-case running time's complexity improves from $O(n^2)$ to the linear $O(n)$.

The median of medians algorithm can also be used to choose a pivot point in the quicksort algorithm for sorting a list of elements. This significantly improves the worst-case performance of the quicksort algorithm from $O(n^2)$ to a complexity of $O(n\log n)$.

Summary

In this chapter, we discussed two important methods to find the k^{th} smallest element in a list, randomized selection and deterministic selection algorithms. The simple solution of merely sorting a list to perform the operation of finding the k^{th} smallest element is not optimal as we can use better methods to determine the k^{th} smallest element. The quickselect algorithm, which is the random selection algorithm, divides the list into two sublists. One list has smaller values, and the other list has greater values as compared to the selected pivot element. We recursively use one of the sublists to find the location of the k^{th} smallest element, which can be further improved by selecting the pivot point using the median of medians method in the deterministic selection algorithm.

In the next chapter, we will discuss several important string matching algorithms.

Exercise

1. What will be the output if the quickselect algorithm is applied to the given array

`arr = [3, 1, 10, 4, 6, 5]` with `k` given as 2?

2. Can quickselect find the smallest element in an array with duplicate values?
3. What is the difference between the quicksort algorithm and the quickselect algorithm?
4. What is the main difference between the deterministic selection algorithm and the quickselect algorithm?
5. What triggers the worst-case behavior of the selection algorithm?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



13

String Matching Algorithms

There are many popular string matching algorithms. String matching algorithms have very important applications, such as searching for an element in a text document, plagiarism detection, text editing programs, and so on. In this chapter, we will study the pattern matching algorithms that find the locations of a given pattern or substring in any given text. We will discuss the **brute force algorithm**, along with the **Rabin-Karp**, **Knuth-Morris-Pratt (KMP)**, and **Boyer-Moore pattern matching algorithms**. This chapter aims to discuss algorithms that are related to strings. The following topics will be covered in this chapter:

- Learning pattern matching algorithms and their implementation
- Understanding and implementing the **Rabin-Karp pattern matching algorithm**
- Understanding and implementing the **Knuth-Morris-Pratt (KMP) algorithm**
- Understanding and implementing the **Boyer-Moore pattern matching algorithm**

Technical requirements

All of the programs based on the concepts and algorithms discussed in this chapter are provided in the book as well as in the GitHub repository at the following link:

<https://github.com/PacktPublishing/Hands-On-Data-Structures-and-Algorithms-with-Python-Third-Edition/tree/main/Chapter13>.

String notations and concepts

Strings are sequences of characters. Python provides a rich set of operations and functions that can be applied to the string data type. Strings are textual data and are handled very efficiently in Python.

The following is an example of a string (`s`) — "packt publishing".

A substring is a sequence of characters that's part of the given string, i.e., specified indices in the string in a continuous order. For example, "packt" is a substring of the string "packt publishing". On the other hand, a subsequence is also a sequence of characters that can be obtained from the given string by removing some of the characters from the string by keeping the order of occurrence of the characters. For example, "pct pblishing" is a valid subsequence for the string "packt publishing" that is obtained by removing the characters `a`, `k`, and `u`. However, this is not a substring since "pct pblishing" is not a continuous sequence of characters. Hence, a subsequence is different from a substring, and it can be considered a generalization of substrings.

The prefix (`p`) is a substring of the string (`s`) in that it is present at the start of the string. There is also another string (`u`) that exists in the string (`s`) after the prefix. For example, the substring "pack" is a

prefix for the string (`s`) = "packt publishing" as it is the starting substring and there is another substring (`u`) = "publishing" after it. Thus, the prefix plus string (`u`) makes "packt publishing", which is the whole string.

The suffix (`d`) is a substring that is present at the end of the string (`s`). For example, the substring "shing" is one of the many possible suffixes for the string "packt publishing". Python has built-in functions to check whether a string starts or ends with a specific string, as shown in the following code snippet:

```
string = "this is data structures book by packt publisher"
suffix = "publisher"
prefix = "this"
print(string.endswith(suffix)) #Check if string contains given suffi
print(string.startswith(prefix)) #Check if string starts with given p
```

The output of the above code is as follows:

```
True
True
```

In the above example of the given string, we can see that the given text string ends with another substring "publisher", which is a valid suffix, and that also has another substring "this", which is a substring of the string start and is also a valid prefix.

Note that the pattern matching algorithms discussed here are not to be confused with the matching statements of Python 3.10.

Pattern matching algorithms are the most important string processing algorithms and we will discuss them in the subsequent

sections, starting with pattern matching algorithms.

Pattern matching algorithms

A pattern matching algorithm is used to determine the index positions where a given pattern string (p) is matched in a text string (T). Thus, the pattern matching algorithm finds and returns the index where a given string pattern appears in a text string. It returns "pattern not found" if the pattern does not have a match in the text string.

For example, for the given text string (s) = "packt publisher" and the pattern string (p) = "publisher", the pattern-matching algorithm returns the index position where the pattern string is matched in the text string. An example of a string matching problem is shown in *Figure 13.1*:

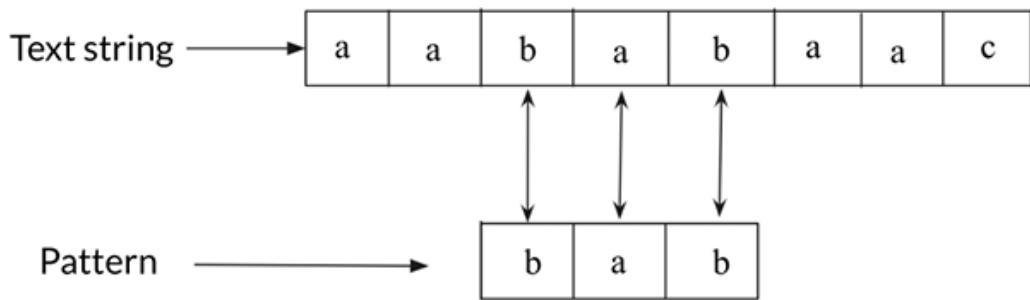


Figure 13.1: An example of a string matching problem

We will discuss four pattern matching algorithms, that is, the brute force method, Rabin-Karp algorithm, and the **Knuth-Morris-Pratt (KMP)** and Boyer-Moore pattern-matching algorithms. We start with the brute force pattern matching algorithm.

The brute force algorithm

The brute force algorithm is also called the naive approach to pattern matching algorithms. Naive approach means that it is a very basic and simple algorithm. In this approach, we match all the possible combinations of the input pattern in the given text string to find the position of the occurrence of the pattern. This algorithm is very naive and is not suitable if the text is very long.

In this algorithm, we start by comparing the characters of the pattern string and the text string one by one, and if all the characters of the pattern are matched with the text, we return the index position of the text where the first character of the pattern is located. If any character of the pattern is mismatched with the text string, we shift the pattern by one position to check if the pattern appears at the next index position. We continue comparing the pattern and text string by shifting the pattern by one index position.

To better understand how the brute force algorithm works, let's look at an example. Suppose we have a text string (T) = "acbcabccababcaacbcac", and the pattern string (P) is "acbcac".

Now, the objective of the pattern matching algorithm is to determine the index position of the pattern string in the given text, T , as shown in *Figure 13.2*:

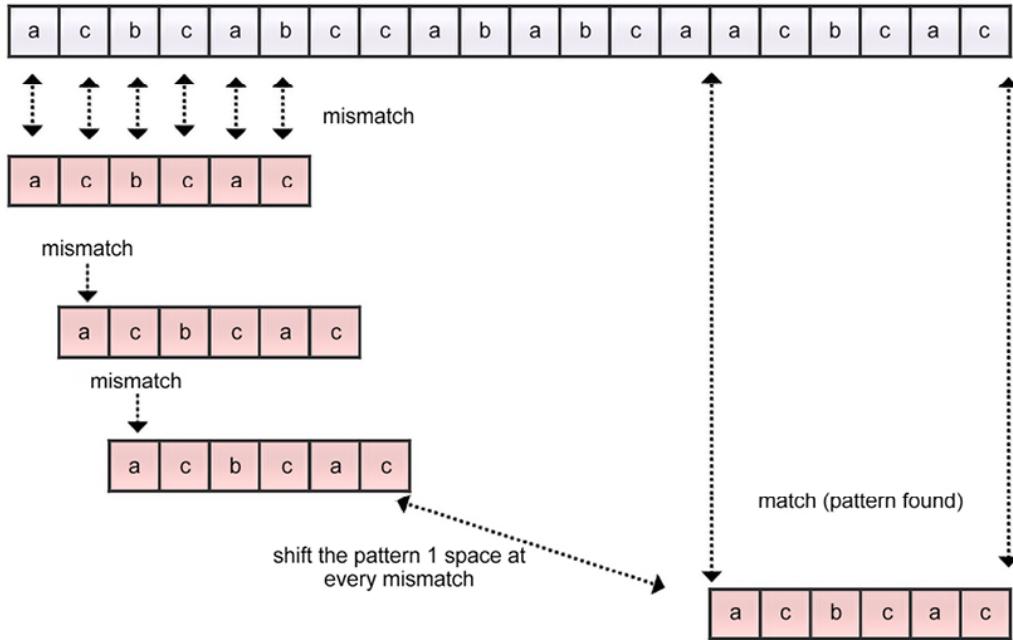


Figure 13.2: An example of the brute force algorithm for string matching

We start by comparing the first character of the text, that is, **a**, and the first character of the pattern. Here, the initial five characters of the pattern are matched, and then there is a mismatch in the last character of the pattern. This is a mismatch, so we shift the pattern by one place. We again start comparing the first character of the pattern and the second character of the text string one by one. Here, character **c** of the text string does not match with the character **a** of the pattern. So, this is also a mismatch, and we shift the pattern by one space, as shown in *Figure 13.2*. We continue comparing the characters of the pattern and the text string until we traverse the whole text string. In this example, we find a match at index position **14**, which is shown in *Figure 13.2*.

Let's consider the Python implementation of the brute force algorithm for pattern matching:

```

def brute_force(text, pattern):
    l1 = len(text)      # The Length of the text string
    l2 = len(pattern)   # The Length of the pattern
    i = 0
    j = 0              # Looping variables are set to 0
    flag = False        # If the pattern doesn't appear at all, then
    while i < l1:      # iterating from the 0th index of text
        j = 0
        count = 0
        # Count stores the length upto which the pattern and the text
        while j < l2:
            if i+j < l1 and text[i+j] == pattern[j]:
                # statement to check if a match has occurred or not
                count += 1      # Count is incremented if a character
                j += 1
            if count == l2:    # it shows a matching of pattern in the tex
                print("\nPattern occurs at index", i)
                # print the starting index of the successful match
                flag = True
                # flag is True as we wish to continue looking for mor
                i += 1
    if not flag:
        # If the pattern doesn't occur at all, means no match of patt
        print('\nPattern is not at all present in the array')

```

The following code snippet can be used to call the function to search the pattern 'acbcac' in the given string:

```
brute_force('acbcabccababcaacbcac', 'acbcac')      # function call
```

The output of the above function call is as follows:

```
Pattern occurs at index 14
```

In the preceding code for the brute force approach, we start by computing the length of the given text strings and pattern. We also initialize the looping variables with `0` and set the flag to `False`.

This variable is used to continue searching for a match of the pattern in the string. If the `flag` variable is `False` by the end of the text string, it means that there is no match for the pattern at all in the text string.

Next, we start the searching loop from the `0th` index to the end of the text string. In this loop, we have a `count` variable that is used to keep track of the length up to which the pattern and the text have been matched. Next, we have another nested loop that runs from the `0th` index to the length of the pattern. Here, the variable `i` keeps track of the index position in the text string and the variable `j` keeps track of the characters in the pattern. Next, we compare the characters of the patterns and the text string using the following code fragment:

```
if i+j<l1 and text[i+j] == pattern[j]:
```

Furthermore, we increment the `count` variable after every match of the character of the pattern in the text string. Then, we continue matching the characters of the pattern and text string. If the length of the pattern becomes equal to the `count` variable, it means there is a match.

We print the index position of the text string if there is a match for the pattern string in the text string and keep the `flag` variable as to `True` as we wish to continue searching for more matches of the patterns in the text string. Finally, if the value of the variable `flag` is

`False`, it means that there was not a match for the pattern in the text string at all.

The best-case and worst-case time complexities for the naive string matching algorithms are $O(n)$ and $O(m*(n-m+1))$, respectively. The best-case scenario occurs when the pattern is not found in the text and the first character of the pattern is not present in the text at all, for example, if the text string is `ABAACEBCCDAAEE`, and the pattern is `FAA`. Here, as the first character of the pattern will not find a match anywhere in the text, it will have comparisons equal to the length of the text (n).

The worst-case scenario occurs when all characters of the text string and the pattern are the same and we want to find out all the occurrences of the given pattern string in the text string, for example, if the text string is `AAAAAAAAAAAAAAA`, and the pattern string is `AAAA`. Another worst-case scenario occurs when only the last character is different, for example, if the text string is `AAAAAAAAAAAAAF` and the pattern is `AAAAF`. Thus, the total number of comparisons will be $m*(n-m+1)$ and the worst-case time complexity will be $O(m*(n-m+1))$.

Next, we discuss the Rabin-Karp pattern matching algorithm.

The Rabin-Karp algorithm

The Rabin-Karp pattern matching algorithm is an improved version of the brute force approach to find the location of the given pattern in the text string. The performance of the Rabin-Karp algorithm is improved by reducing the number of comparisons with the help of hashing. We discussed the concept of hashing in *Chapter 8, Hash*

Tables. The hashing function returns a unique numeric value for a given string.

This algorithm is faster than the brute force approach as it avoids unnecessary comparisons. In this algorithm, we compare the hash value of the pattern with the hash value of the substring of the text string. If the hash values are not matched, the pattern is shifted forward one position. This is a better algorithm as compared to the brute-force algorithm since there is no need to compare all the characters of the pattern one by one.

This algorithm is based on the concept that if the hash values of the two strings are equal, then it is assumed that both the strings are also equal. However, it is also possible that there can be two different strings whose hash values are equal. In that case, the algorithm will not work; this situation is known as a spurious hit and happens due to a collision in hashing. To avoid this with the Rabin-Karp algorithm, after matching the hash values of the pattern and the substring, we ensure that the pattern is actually matched in the string by comparing the pattern and the substring character by character.

The Rabin-Karp pattern matching algorithm works as follows:

1. First, we preprocess the pattern before starting the search, that is, we compute the hash value of the pattern of length m and the hash values of all the possible substrings of the text of length m . The total number of possible substrings would be $(n-m+1)$. Here, n is the length of the text.
2. We compare the hash value of the pattern with the hash value of the substrings of the text one by one.

3. If the hash values are not matched, then we shift the pattern by one position.
4. If the hash value of the pattern and the hash value of the substring of the text match, then we compare the pattern and substring character by character to ensure that the pattern is actually matched in the text.
5. We continue the process of steps 2-5 until we reach the end of the given text string.

In this algorithm, we compute the numerical hash values using Horner's rule (any other hashing function can also be used) that returns a unique value for the given string. We also compute the hash value using the sum of the ordinal values of all the characters of the string.

Let's consider an example to understand the **Rabin-Karp algorithm**.

Let's say we have a text string (T) = "publisher paakt packt", and the pattern (P) = "packt". First, we compute the hash values of the pattern (length m) and all the substrings (of length m) of the text string. The functionality of the **Rabin-Karp algorithm** is shown in *Figure 13.3:*

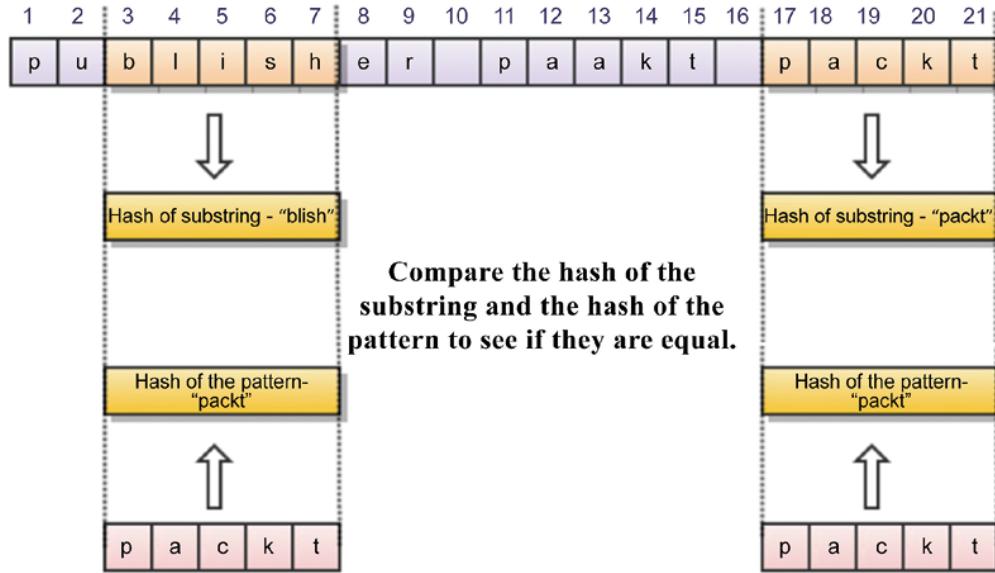


Figure 13.3: An example of the Rabin-Karp algorithm for string matching

We start comparing the hash value of the pattern "packt" with the first substring "publi". Since the hash values do not match, we shift the pattern by one position, and then we compare the hash value of pattern with the hash value of the next substring of the text, i.e.

"ublis". As these hash values also do not match, we again shift the pattern by one position. We shift the pattern by one position at a time if the hash values do not match. And, if the hash value of the pattern and the hash value of the substring match, we compare the pattern and substring character by character and we return the location of the text string if they match.

In the example shown in *Figure 13.3*, hash values of the pattern and the substring of the text are matched at location 17.

It is important to note that there can be a different string whose hash value can match with the hash of the pattern, i.e. a spurious hit.

Next, let us discuss the implementation of the **Rabin-Karp pattern matching algorithm**.

Implementing the Rabin-Karp algorithm

The implementation of the **Rabin-Karp algorithm** is done in two steps:

1. We implement the `generate_hash()` method, which is used to compute the hash value of the pattern and all the possible combinations of the substrings of length equal to the length of the pattern.
2. We implement the **Rabin-Karp algorithm**, which uses the `generate_hash()` method to identify the substring whose hash value matches the hash value of the pattern. Finally, we match them character by character to ensure we have correctly found the pattern.

Let us first discuss the implementation of generating hash values for the patterns and substrings of the text. For this, we need to first decide on the hash function. Here, we use the sum of all the ordinal values of all the characters of the string as the hashing function.

The complete Python implementation to compute the hashing values is given below:

```
def generate_hash(text, pattern):
    ord_text = [ord(i) for i in text]          # stores unicode value of
    ord_pattern = [ord(j) for j in pattern]    # stores unicode value of
    len_text = len(text)                      # stores length of the te
```

```
len_pattern = len(pattern)           # stores length of the pa
len_hash_array = len_text - len_pattern + 1 # stores the length o
hash_text = [0]*(len_hash_array)       # Initialize all the valu
hash_pattern = sum(ord_pattern)
for i in range(0,len_hash_array):      # step size of the loop w
    if i == 0:                         # Base condition
        hash_text[i] = sum(ord_text[:len_pattern])  # initial va
    else:
        hash_text[i] = ((hash_text[i-1] - ord_text[i-1]) + ord
[i+len_pattern-1])   # calculating next hash value using previous val
return [hash_text, hash_pattern]          # return the has
```

In the above code, we start by storing the ordinal values of all the characters of the text and the pattern in the `ord_text` and `ord_pattern` variables. Next, we store the length of the text and the pattern in the `len_text` and `len_pattern` variables.

Next, we create a variable called `len_hash_array` that stores the number of all the possible substrings of length (equal to the length of the pattern) using `len_text - len_pattern + 1`, and we create an array called `hash_text` that stores the hash value for all the possible substrings. This is shown in the following code snippet:

```
len_hash_array = len_text - len_pattern + 1
hash_text = [0]*(len_hash_array)
```

Next, we compute the hash value for the pattern by summing up the ordinal values of all the characters in the pattern using the following code snippet:

```
hash_pattern = sum(ord_pattern)
```

Next, we start a loop that executes for all the possible substrings of the text. For this, initially, we compute the hash value for the first substring by summing the ordinal values of all of its characters using `sum(ord_text[:len_pattern])`. Further, the hash values for all of the substrings are computed using the hash value of the previous substrings as shown in the following code snippet:

```
hash_text[i] = ((hash_text[i-1] - ord_text[i-1]) + ord_text[i+len
```

So, we have precomputed the hash values for the pattern and all the substrings of the text that we will use for comparing the pattern and the text in the implementation of the **Rabin-Karp algorithm**. The **Rabin-Karp algorithm** works as follows. Firstly, we compare the hash values of the pattern and substrings of the text. Next, we take the substring for which the hash matches with the hash of the pattern and compare them both character by character.

The complete Python implementation of the **Rabin-Karp algorithm** is as follows:

```
def Rabin_Karp_Matcher(text, pattern):
    text = str(text)                                # convert text into s
    pattern = str(pattern)                          # convert pattern into str
    hash_text, hash_pattern = generate_hash(text, pattern) # generate
    len_text = len(text)                            # Length of text
    len_pattern = len(pattern)                      # Length of pattern
    flag = False                                    # checks if pattern is present atle
    for i in range(len(hash_text)):
        if hash_text[i] == hash_pattern:          # if the hash value matche
            count = 0                           # count the total characte
            for j in range(len_pattern):
                if pattern[j] == text[i+j]: # checking equality for e
                    count += 1             # if value is equal, then
```

```
        else:
            break
        if count == len_pattern:           # if count is equal to length of pattern
            flag = True                  # update flag accordingly
            print('Pattern occurs at index',i)
    if not flag:                      # if pattern doesn't match
        print('Pattern is not at all present in the text')
```

In the above code, firstly, we convert the given text and pattern into string format as the ordinal values can only be computed for strings. Next, we use the `generate_hash` function to compute the hash values of patterns and texts. We store the length of the text and patterns in the `len_text` and `len_pattern` variables. We also initialize the `flag` variable to `False` so that it keeps track of whether the pattern is present in the text at least once.

Next, we start a loop that implements the main concept of the algorithm. This loop executes for the length of `hash_text`, which is the total number of possible substrings. Initially, we compare the hash value of the first substring with the hash of the pattern by using `if hash_text[i] == hash_pattern`. If they do not match; we move one index position and look for another substring. We iteratively move further until we get a match.

If we find a match, we compare the substring and the pattern character by character through a loop by using `if pattern[j] == text[i+j]`.

We then create a `count` variable to keep track of how many characters match in the pattern and the substring. If the length of the count and the length of the pattern are equal, this means that all of the characters match, and the index location where the pattern was

found is returned. Finally, if the `flag` variable remains `False`, this means that the pattern does not match at all with the text. The following code snippets can be used to execute the **Rabin-Karp matching algorithm**:

```
Rabin_Karp_Matcher("101110000011010010101101", "1011")
Rabin_Karp_Matcher("ABBACCADABBACCEDF", "ACCE")
```

The output of the above code is as follows:

```
Pattern occurs at index 0
Pattern occurs at index 18
Pattern occurs at index 11
```

In the above code, we first check whether the pattern “`1011`” appears in the given text string “`101110000011010010101101`”. The output shows that the given pattern occurs at index position `0` and `18`. Next, the pattern “`ACCE`” occurs at index position `11` in the text string “`ABBACCADABBACCEDF`”.

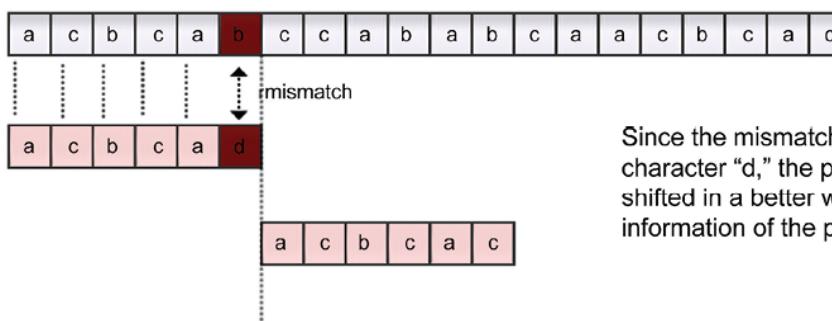
The Rabin-Karp pattern matching algorithm preprocesses the pattern before the searching; that is, it computes the hash value for the pattern that has the complexity of $O(m)$. Also, the worst-case running time complexity of the Rabin-Karp algorithm is $O(m * (n - m + 1))$. The worst-case scenario is when the pattern does not occur in the text at all. The average-case scenario is when the pattern occurs at least once.

Next, we will discuss the KMP string matching algorithm.

The Knuth-Morris-Pratt algorithm

The KMP algorithm is a pattern matching algorithm based on the idea that the overlapping text in the pattern itself can be used to immediately know at the time of any mismatch how much the pattern should be shifted to skip unnecessary comparisons. In this algorithm, we will precompute the `prefix` function that indicates the required number of shifts of the pattern whenever we get a mismatch. The KMP algorithm preprocesses the pattern to avoid unnecessary comparisons using the `prefix` function. So, the algorithm utilizes the `prefix` function to estimate how much the pattern should be shifted to search the pattern in the text string whenever we get a mismatch. The **KMP algorithm** is efficient as it minimizes the number of comparisons of the given patterns with respect to the text string.

The motivation behind the **KMP algorithm** can be observed in *Figure 13.4*. In this example, it can be seen that the mismatch occurred at the 6th position with the last character “d” after matching the initial 5 characters. It is also known from the `prefix` function that the character “d” did not appear before in the pattern, and utilizing this information, the pattern can be shifted by six places:



Since the mismatch occurred at character “d,” the pattern can be shifted in a better way utilizing the information of the pattern itself.

Figure 13.4: Example of the KMP algorithm

So, in this example, the pattern has shifted six positions instead of one. Let us discuss another example to understand the concept of the **KMP algorithm**, as shown in *Figure 13.5*:

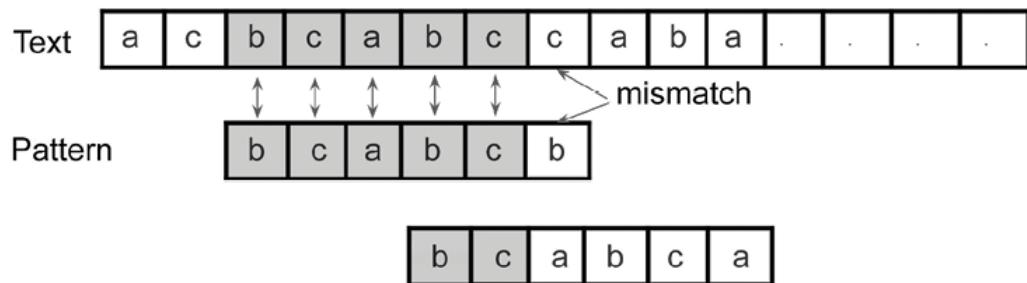


Figure 13.5: Second example of the KMP algorithm

In the above example, the mismatch occurs at the last character of the pattern. Since the pattern at the location of the mismatch has a partial match of the prefix **bc**, this information is given by the `prefix` function. Here, the pattern can be shifted to align with the other occurrence of the matched prefix **bc** in the pattern.

We will look into the `prefix` function next for a better understanding of how we use it to know by how much we should shift the pattern.

The `prefix` function

The `prefix` function (also known as the failure function) finds the pattern within the pattern. It finds out how much the previous comparisons can be reused due to repetition in the pattern itself when there is a mismatch. The `prefix` function returns a value for each position wherever we get a mismatch, which tells us by how much the pattern should be shifted.

Let us understand how we use the `prefix` function to find the required shift amount with the following examples. Consider the first example: if we had a `prefix` function for a pattern where all of the characters are different, the `prefix` function would have a value of `0`. This means that if we find any mismatch, the pattern will be shifted by the number of characters compared up to that position in the pattern.

Consider an example with the pattern **abcde**, which contains all different characters. We start comparing the first character of the pattern with the first character of the text string, as shown in *Figure 13.6*. As shown in the figure, the mismatch occurs at the 4th character in the pattern. Since the prefix function has the value 0, it means that there is no overlap in the pattern and no previous comparisons would be reused, so the pattern will be shifted to the number of characters compared up until that point:

Index	1	2	3	4	5
Pattern	a	b	c	d	e
Prefix_function	0	0	0	0	0

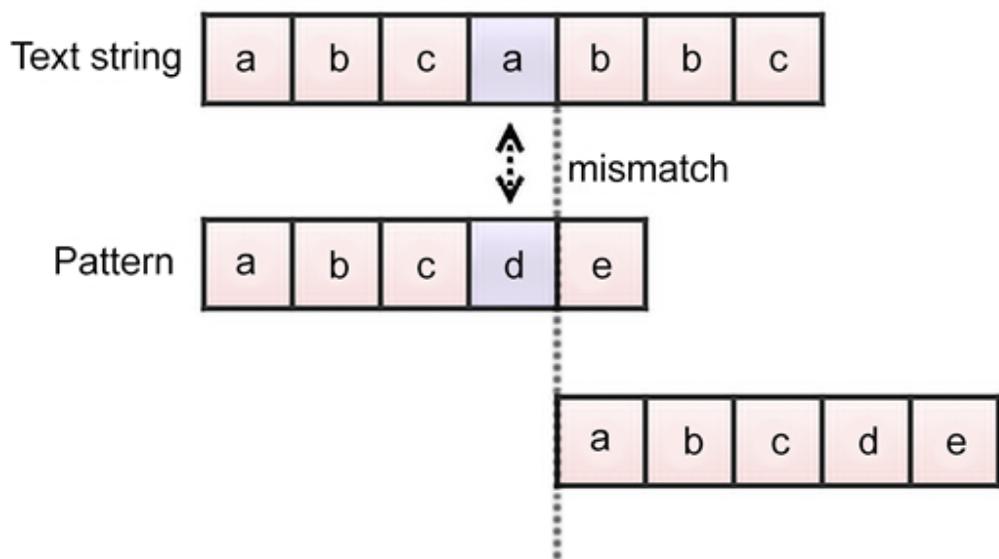


Figure 13.6: Prefix function in the KMP algorithm

Let's consider another example to better understand how the `prefix` function works for the pattern (P) **abcabbcab** as shown in Figure 13.7:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0						

Figure 13.7: Example of the prefix function in the KMP algorithm

In *Figure 13.7*, we start calculating the values of the `prefix` function starting from index 1. We assign the value 0 if there is no repetition of the characters in the pattern. So, in this example, we assign 0 to the `prefix` function for index positions 1 to 3. Next, at index position 4, we can see that there is a character, **a**, which is a repetition of the first character of the pattern itself, so we assign the value 1 here, as shown in *Figure 13.8*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1					

Figure 13.8: Value of the prefix function at index 4 in the KMP algorithm

Next, we look at the next character at position 5. It has the longest suffix pattern, **ab**, and so it would have a value of 2, as shown in *Figure 13.9*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2				

Figure 13.9: Value of the prefix function at index 5 in the KMP algorithm

Similarly, we look at the next index position of 6. Here, the character is **b**. This character does not have the longest suffix in the pattern, so

it has the value **0**. Next, we assign value **0** at index position **7**. Then, we look at the index position **8**, and we assign the value **1** as it has the longest suffix of length **1**.

Finally, at the index position of **9**, we have the longest suffix of **2**. This is shown in *Figure 13.10*:

Index	1	2	3	4	5	6	7	8	9
Pattern	a	b	c	a	b	b	c	a	b
Prefix_function	0	0	0	1	2	0	0	1	2

Figure 13.10: Value of the prefix function at index 6 to 9 in the KMP algorithm

The value of the `prefix` function shows how much of the start of the string can be reused if there is a mismatch. For example, if the comparison fails at index position **5**, the `prefix` function value is **2**, which means that the two starting characters don't need to be compared, and the pattern can be shifted accordingly.

Next, we discuss the details of the **KMP algorithm**.

Understanding the KMP algorithm

The **KMP pattern matching algorithm** detects overlaps in the pattern itself so that it avoids unnecessary comparisons. The main idea behind the **KMP algorithm** is to detect how much the pattern should be shifted, based on the overlaps in the patterns. The algorithm works as follows:

1. First, we precompute the `prefix` function for the given pattern and initialize a counter `q` that represents the number of characters that matched.
2. We start by comparing the first character of the pattern with the first character of the text string, and if this matches, then we increment the counter `q` for the pattern and the counter for the text string, and we compare the next character.
3. If there is a mismatch, then we assign the value of the precomputed `prefix` function for `q` to the index value of `q`.
4. We continue searching the pattern in the text string until we reach the end of the text, that is, if we do not find any matches. If all of the characters in the pattern are matched in the text string, we return the position where the pattern is matched in the text and continue to search for another match.

Let's consider the following example to understand the working of the **KMP algorithm**. We have a pattern `acacac` along with index positions from `1` to `6` (just for simplicity, we have index positions starting from 1 instead of 0), shown in *Figure 13.11*. The `prefix` function for the given pattern is constructed as shown in *Figure 13.11*:

Index	1	2	3	4	5	6
Pattern	a	c	a	c	a	c
Prefix_function	0	0	1	2	1	2

Figure 13.11: The prefix function for pattern "acacac"

Let us take an example to understand how we use the `prefix` function to shift the pattern according to the **KMP algorithm** for the text string and pattern given in *Figure 13.12*. We start comparing the pattern and the text character by character. When we mismatch at index position 6, we see the prefix value for this position is 2. Then we shift the pattern according to the return value of the `prefix` function. Next, we start comparing the pattern and text string from the index position of 2 on the pattern (character **c**), and the character **b** of the text string. Since this is a mismatch, the pattern will be shifted according to the value of the `prefix` function at this position. This description is depicted in *Figure 13.12*:

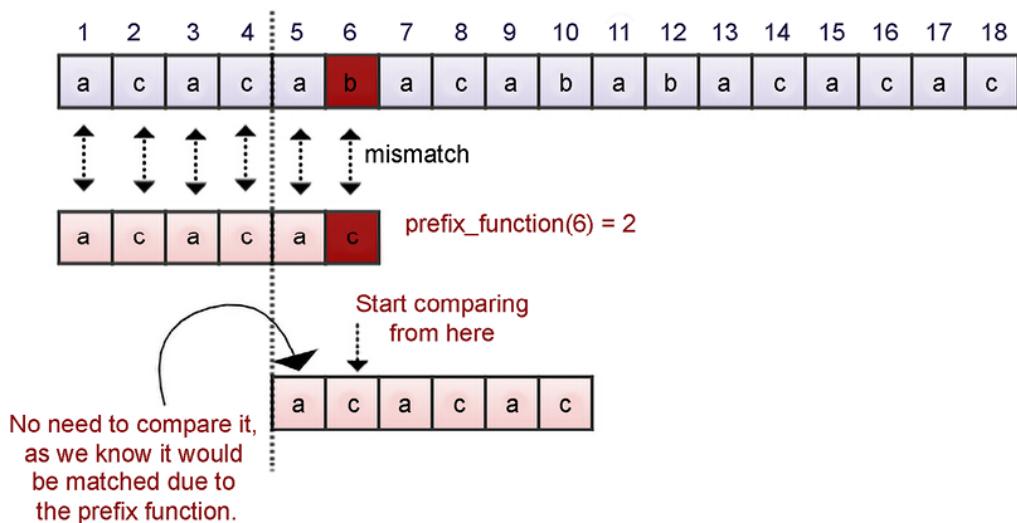


Figure 13.12: The pattern is shifted according to the return value of the prefix function

Now let's take another example shown in *Figure 13.13* where the position of the pattern over the text is shown. When we start comparing the characters **b** and **a**, these do not match, and we see the `prefix` function for index position 1 shows a value of 0, meaning no overlapping of text in the pattern has occurred. Therefore, we shift the pattern by 1 place as shown in *Figure 13.12*. Next, we

compare the pattern and text string character by character, and we find a mismatch at index position 10 in the text between characters **b** and **c**.

Here, we use the precomputed `prefix` function to shift the pattern – as the `prefix_function(4)` is **2**, we shift the pattern to align over the text at index position **2** of the pattern. After that, we compare characters **b** and **c** at index position **10**, and since they do not match, we shift the pattern by one place. This process is shown in *Figure 13.13*:

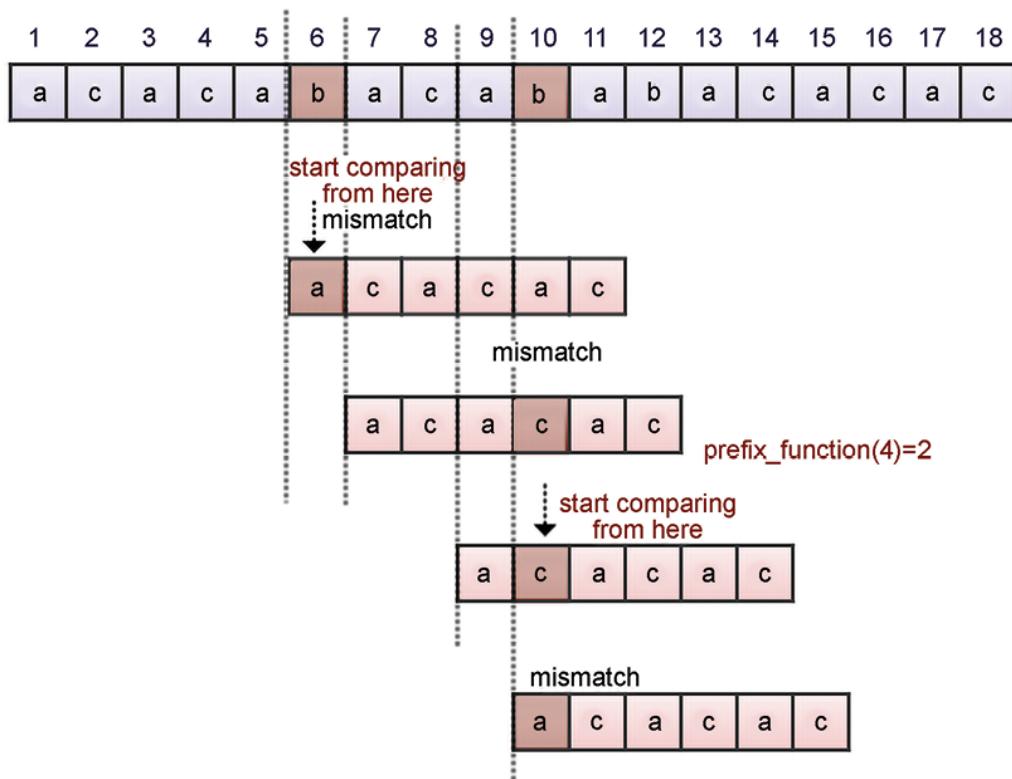


Figure 13.13: Shifting of the pattern according to the return value of the prefix function

Let us continue our searching from index position **11**, as shown in *Figure 13.14*. Next, we compare the characters at index **11** in the text and continue until a mismatch is found. We find a mismatch

between characters **b** and **c** at index position **12**, as shown in *Figure 13.14*. We shift the pattern and move it next to the mismatched character since the `prefix_function(2)` is **0**. We repeat the same process until we reach the end of the string. We find a match of the pattern in the text string at index location **13** in the text string, as in *Figure 13.14*:

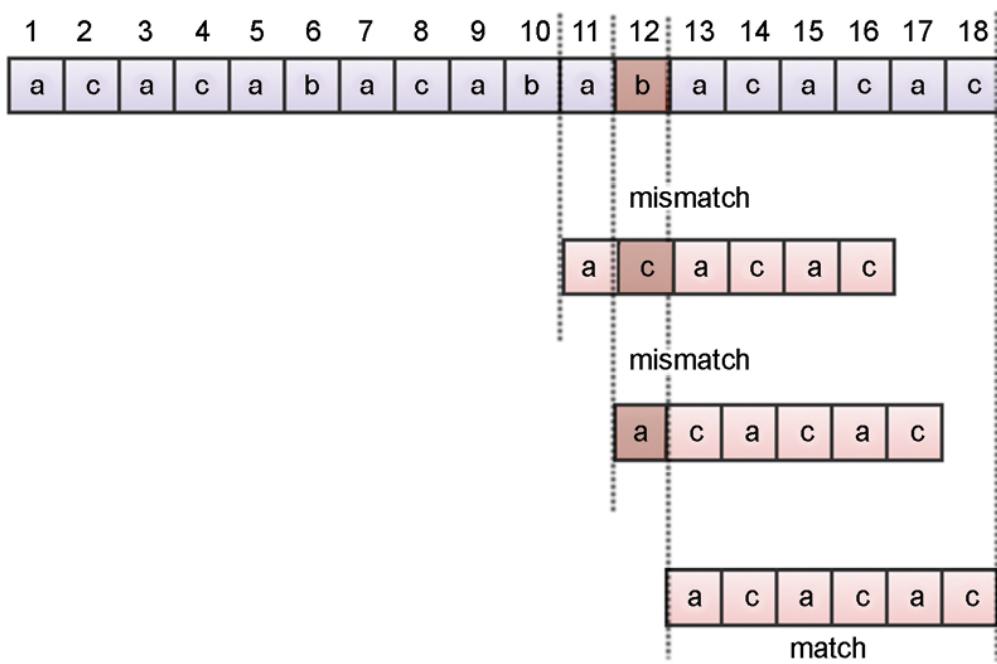


Figure 13.14: Shifting of the pattern for index positions of 11 to 18

The **KMP algorithm** has two phases: first, the preprocessing phase, which is where we compute the `prefix` function, which has the space and time complexity of $O(m)$. Further, the second phase involves searching, for which the **KMP algorithm** has a time complexity of $O(n)$. So, the worst-case time complexity of the **KMP algorithm** is $O(m + n)$.

Now, we will discuss the implementation of the **KMP algorithm** using Python.

Implementing the KMP algorithm

The Python implementation of the KMP algorithm is explained here.

We start by implementing the `prefix` function for the given pattern.

The code for the `prefix` function is as follows:

```
def pfun(pattern):                # function to generate prefix function
    n = len(pattern)              # Length of the pattern
    prefix_fun = [0]*(n)           # initialize all elements of the list
    k = 0
    for q in range(2,n):
        while k>0 and pattern[k+1] != pattern[q]:
            k = prefix_fun[k]
        if pattern[k+1] == pattern[q]:      # If the kth element of the
            k += 1                         # update k accordingly
        prefix_fun[q] = k
    return prefix_fun               # return the prefix function
```

In the above code, we first compute the length of the pattern using the `len()` function, and then we initialize a list to store the values computed by the `prefix` function.

Next, we start the loop that executes from 2 to the length of the pattern. Then, we have a nested loop that is executed until we have processed the whole pattern. The variable `k` is initialized to `0`, which is the `prefix` function for the first element of the pattern. If the k^{th} element of the pattern is equal to the q^{th} element, then we increment the value of `k` by `1`. The value of `k` is the value computed by the `prefix` function, and so we assign it at the index position of `q` in the pattern. Finally, we return the list of the `prefix` function that has the computed value for each character of the pattern.

Once we have created the `prefix` function, we implement the main **KMP matching algorithm**. The following code shows this in detail:

```
def KMP_Matcher(text,pattern):    # KMP matcher function
    m = len(text)
    n = len(pattern)
    flag = False
    text = '-' + text    # append dummy character to make it 1-based i
    pattern = '-' + pattern # append dummy character to the pattern a
    prefix_fun = pfun(pattern) # generate prefix function for the pat
    q = 0
    for i in range(1,m+1):
        while q>0 and pattern[q+1] != text[i]: # while pattern and te
            q = prefix_fun[q]
        if pattern[q+1] == text[i]:                # if pattern and
            q += 1
        if q == n:                                # if q is equ
            print("Pattern occurs at positions ",i-n)    # print the
            flag = True
            q = prefix_fun[q]
    if not flag:
        print('\nNo match found')
```

In the above code, we start by computing the length of the text string and the pattern, which are stored in the variables `m` and `n`, respectively. Next, we define a variable `flag` to indicate whether the pattern has found a match or not. Further, we add a dummy character `-` in the text and pattern to make the indexing start from index `1` instead of index `0`. Next, we call the `pfun()` method to construct the array containing the prefix values for all the positions of the pattern using `prefix_fun = pfun(pattern)`. Next, we execute a loop starting from `1` to `m+1`, where `m` is the length of the pattern. Further, for each iteration of the `for` loop, we compare the pattern and text in a `while` loop until we finish searching the pattern.

If we get a mismatch, we use the value of the `prefix` function at index `q` (here, `q` is the index where the mismatch occurs) to find out by how much we have to shift the pattern. If the pattern and text are equal, then the value of `1` and `n` will be equal, and we can return the index where the pattern was matched in the text. Further, we update the `flag` variable to `True` when the pattern is found in the text. If we finished searching the whole text string and still the variable `flag` was `False`, it would mean the pattern was not present in the given text.

The following code snippet can be used to execute the KMP algorithm for string matching:

```
KMP_Matcher('aabaaacaadaabaaba', 'aabaa')    # function call, with t
```

The output of the above code is as follows:

```
Pattern occurs at positions 0
Pattern occurs at positions 9
```

In the above output, we see that the pattern is present at index positions 0 and 9 in the given text string.

Next, we will discuss another pattern matching algorithm, the Boyer-Moore algorithm.

The Boyer-Moore algorithm

As we have already discussed, the main objective of the string pattern matching algorithm is to find ways of skipping comparisons

as much as possible by avoiding unnecessary comparisons.

The Boyer-Moore pattern matching algorithm is another such algorithm (along with the KMP algorithm) that further improves the performance of pattern matching by skipping comparisons using different methods. We have to understand the following concepts in order to understand the Boyer-Moore algorithm:

1. In this algorithm, we shift the pattern in the direction from left to right, similar to the KMP algorithm.
2. We compare the characters of the pattern and the text string from right to left, which is the opposite of what we do in the case of the KMP algorithm.
3. The algorithm skips the unnecessary comparisons by using the good suffix and bad character shift heuristics. These heuristics themselves find the possible number of comparisons that can be skipped. We slide the pattern over the given text with the greatest offsets suggested by both of these heuristics.

Let us understand all about these heuristics and the details of how the Boyer-Moore pattern matching algorithm works.

Understanding the Boyer-Moore algorithm

The Boyer-Moore algorithm compares the pattern with the text from right to left, meaning that in this algorithm if the end of the pattern does not match with the text, the pattern can be shifted rather than checking every character of the text. The key idea is that the pattern is aligned with the text and the last character of the pattern is

compared with the text, and if they do not match, then it is not required to continue comparing each character and we can rather shift the pattern.

Here, how much we shift the pattern depends upon the mismatched character. If the mismatched character of the text does not appear in the pattern, it means we can shift the pattern by the whole length of the pattern, whereas if the mismatched character appears in the pattern somewhere, then we partially shift the pattern in such a way that the mismatched character is aligned with the other occurrence of that character in the pattern.

In addition, in this algorithm, we can also see what portion of the pattern has matched (with the matched suffix), so we utilize this information and align the text and pattern by skipping any unnecessary comparisons. Making the pattern jump along the text to reduce the number of comparisons rather than checking every character of the pattern with the text is the main idea of an efficient string matching algorithm.

The concept behind the Boyer-Moore algorithm is demonstrated in *Figure 13.15*:

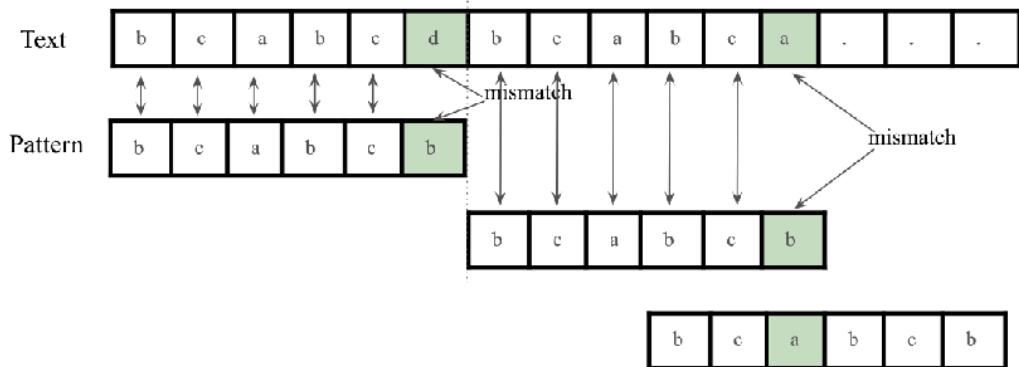


Figure 13.15: A example to demonstrate the concept of the Boyer-Moore algorithm

In the example shown in *Figure 13.15*, where character **b** of the pattern mismatches with character **d** of the text, we can shift the entire pattern since the mismatched character **d** is not present in the pattern anywhere. In the second mismatch, we can see that the mismatched character **a** in the text is present in the pattern, so we shift the pattern to align with that character. This example shows how we can skip unnecessary comparisons. Next, we will discuss further the details of the algorithm.

The Boyer-Moore algorithm has two heuristics to determine the maximum shift possible for the pattern when we find a mismatch:

- Bad character heuristic
- Good suffix heuristic

At the time of a mismatch, each of these heuristics suggests possible shifts, and the Boyer-Moore algorithm shifts the pattern over the text string by a longer distance considering the maximum shift given by bad character and good suffix heuristics. The details of the bad character and good suffix heuristics are explained in detail with examples in the following subsections.

Bad character heuristic

The Boyer-Moore algorithm compares the pattern and the text string in the direction of right to left. It uses the bad character heuristic to shift the pattern, where we start comparing character by character from the end of the pattern, and if they match then we compare the second to-last character, and if that also matches, then the process is repeated until the entire pattern is matched or we get a mismatch.

The mismatched character of the text is also known as a bad character. If we get any mismatch in this process, we shift the pattern according to one of the following conditions:

1. If the mismatched character of the text does not occur in the pattern, then we shift the pattern next to the mismatched character.
2. If the mismatched character has one occurrence in the pattern, then we shift the pattern in such a way that we align with the mismatched character.
3. If the mismatched character has more than one occurrence in the pattern, then we make the most minimal shift possible to align the pattern with that character.

Let us understand these three cases with examples. Consider a text string (T) and the pattern = {acacac}. We start by comparing the characters from right to left, that is, character **c** of the pattern and character **b** of the text string. Since they do not match, we look for the mismatched character of the text string (that is **b**) in the pattern. Since the bad character **b** does not appear in the pattern, we shift the pattern next to the mismatched character, as shown in *Figure 13.16*:

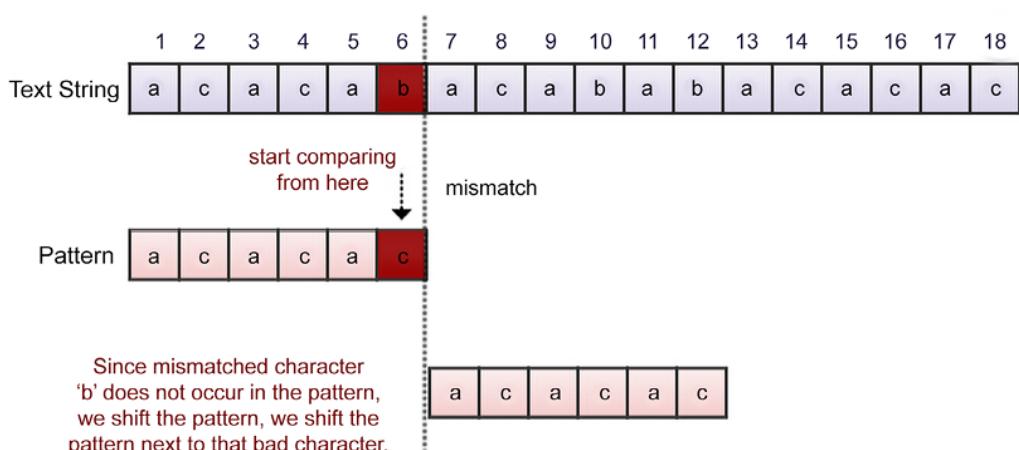


Figure 13.16: Example of the bad character heuristic in the Boyer-Moore algorithm

Let's take another example with a given text string and the pattern = **{acacac}** as shown in *Figure 13.17*. For the given example, we compare the characters of the text string and the pattern from right to left, and we get a mismatch for the character **d** of the text. Here, the suffix **ac** is matched, but the characters **d** and **c** do not match, and the mismatched character **d** does not appear in the pattern. Therefore, we shift the pattern next to the mismatched character, as shown in *Figure 13.17*:

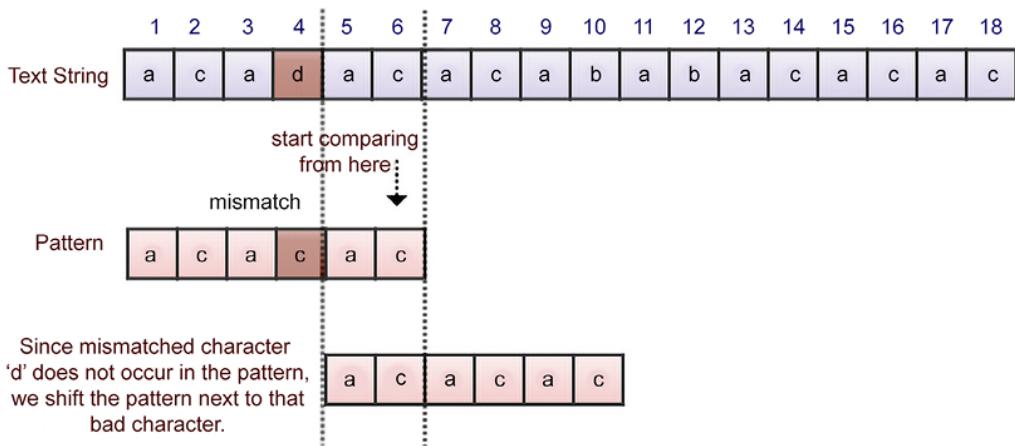


Figure 13.17: Second example of the bad character heuristic in the Boyer-Moore algorithm

Let's consider an example to understand the second and third cases of the bad character heuristic for the given text string and the pattern as shown in *Figure 13.18*. Here, the suffix **ac** is matched, but the next characters, **a** and **c**, do not match, so we search for the occurrences of the mismatched character **a** in the pattern. Since it has two occurrences in the pattern, we have two options for shifting the pattern to align it with the mismatched character. Both of these options are shown in *Figure 13.18*:

In such situations where we have more than one option to shift the pattern, we apply the least possible number of shifts to prevent missing any possible match. If on the other hand we have only one occurrence of the mismatched character in the pattern, we can easily shift the pattern in such a way that the mismatched character is aligned. So, in this example, we would prefer option 1 to shift the pattern as shown in *Figure 13.18*:

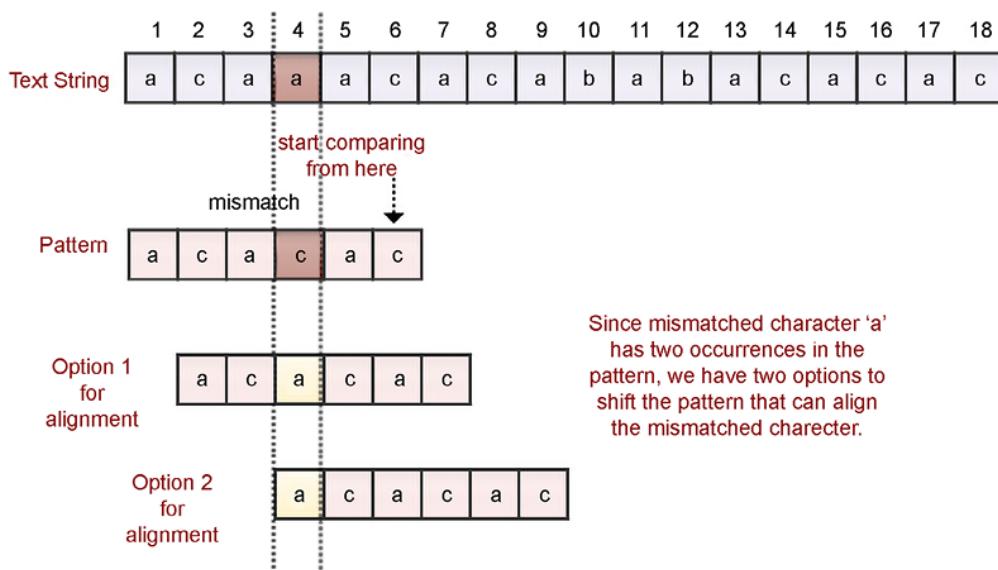


Figure 13.18: Third example of the bad character heuristic in the Boyer-Moore algorithm

We have discussed the bad character heuristic so far, and we consider the good suffix heuristic in the next section.

Good suffix heuristic

The bad character heuristic does not always provide good suggestions for shifting the pattern. The Boyer-Moore algorithm also uses the good suffix heuristic to shift the pattern over the text string, which is based on the matched suffix. In this method, we shift the pattern to the right in such a way that the matched suffix of the

pattern is aligned with another occurrence of the same suffix in the pattern.

It works like this: we start by comparing the pattern and the text string from right to left, and if we find any mismatch, then we check the occurrence of the suffix in the pattern that has been matched so far, which is known as a good suffix.

In such situations, we shift the pattern in such a way that we align another occurrence of the good suffix to the text. The good suffix heuristic has two main cases:

1. The matching suffix has one or more occurrences in the pattern
2. Some part of the matching suffix is present at the start of the pattern (this means that the suffix of the matched suffix exists as the prefix of the pattern)

Let's understand these cases with the following examples. Suppose we have a given text string and the pattern **acabac** as shown in *Figure 13.19*. We start comparing the characters from right to left, and we get a mismatch with the character **a** of the text string and **b** of the pattern. By the point of this mismatch, we have already matched the suffix **ac**, which is called the "good suffix." Now, we search for another occurrence of the good suffix **ac** in the pattern (which is present at the starting position of the pattern in this example) and we shift the pattern to align it with that suffix, as shown in *Figure 13.19*:

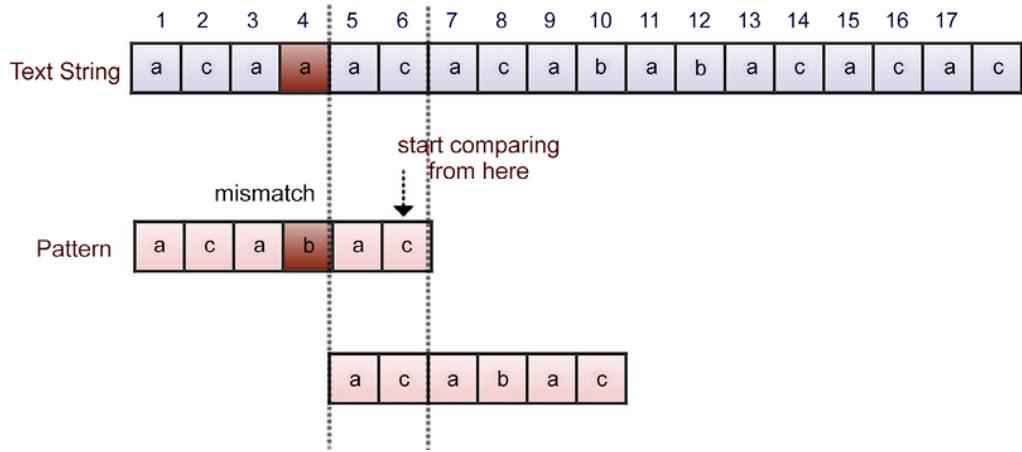


Figure 13.19: Example of the good suffix heuristic in the Boyer-Moore algorithm

Let's take another example to understand the good suffix heuristic. Consider the text string and pattern given in *Figure 13.18*. Here, we get a mismatch between characters **a** and **c**, and we get a good suffix **ac**. Here, we have two options for shifting the pattern to align it with the good suffix string.

In a situation where we have more than one option to shift the pattern, we take the option with the lower number of shifts. For this reason, we take option 1 in this example, as shown in *Figure 13.20*:

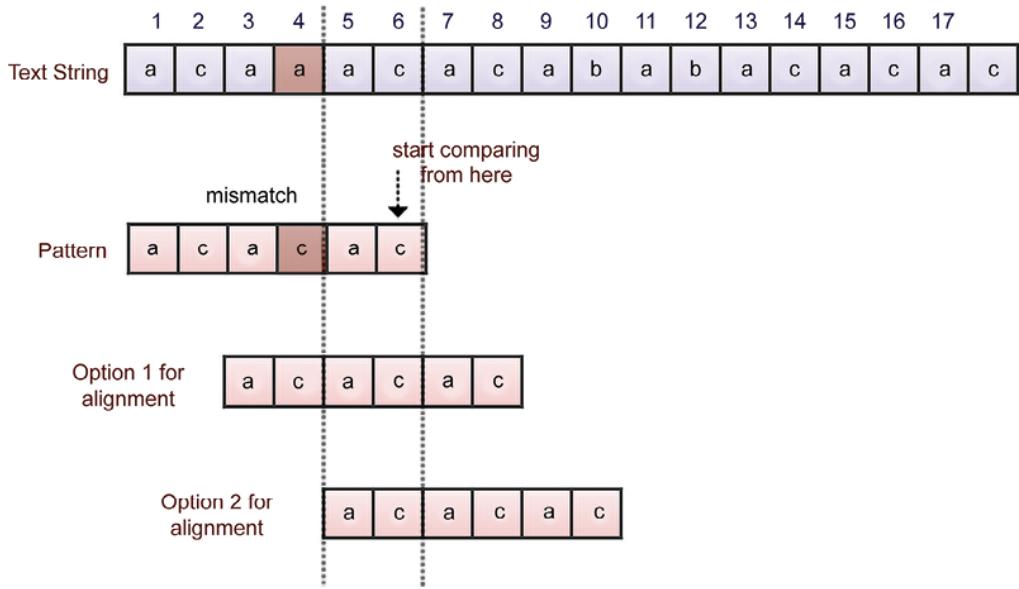


Figure 13.20: Second example of the good suffix heuristic in the Boyer-Moore algorithm

Let's take a look at another example of the text string and pattern shown in *Figure 13.19*. In this example, we get a good suffix string **aac**, and we get a mismatch for the characters **b** of the text string and **a** of the pattern. Now, we search for the good suffix **aac** in the pattern, but we do not find another occurrence of it. When this happens, we check whether the prefix of the pattern matches the suffix of the good suffix, and if so, we shift the pattern to align with it.

For this example, we find that the prefix **ac** at the start of the pattern does not match with the full good suffix, but does match the suffix **ac** of the good suffix **aac**. In such a situation, we shift the pattern by aligning with the suffix of **aac** that is also a prefix of the pattern as shown in *Figure 13.21*:

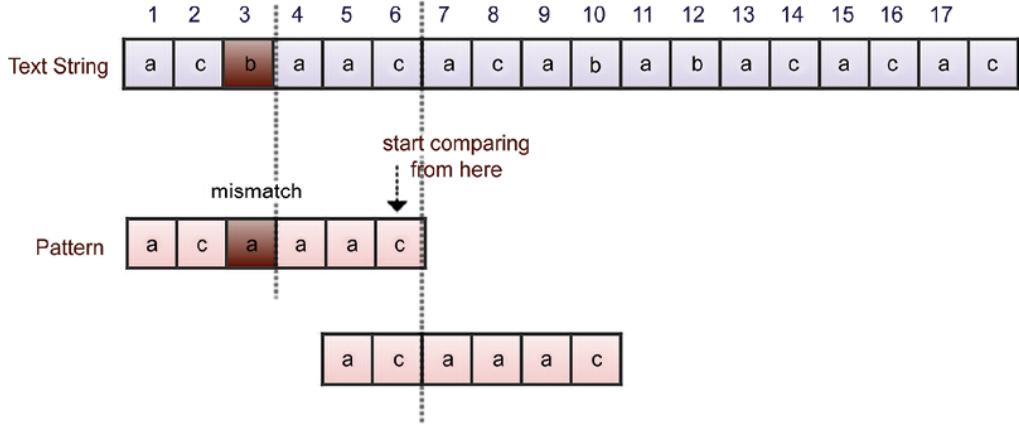


Figure 13.21: Third example of the good suffix heuristic in the Boyer-Moore algorithm

Another case for the good suffix heuristic for the given text string and pattern is shown in *Figure 13.22*. In this example, we compare the text and pattern and find the good suffix **aac**, and we get a mismatch with character **b** of the text and **a** of the pattern.

Next, we search for the matched good suffix in the pattern, but there is no occurrence of the suffix in the pattern, nor does any prefix of the pattern match the suffix of the good suffix. So, in this kind of situation, we shift the pattern after the matched good suffix as shown in *Figure 13.22*:

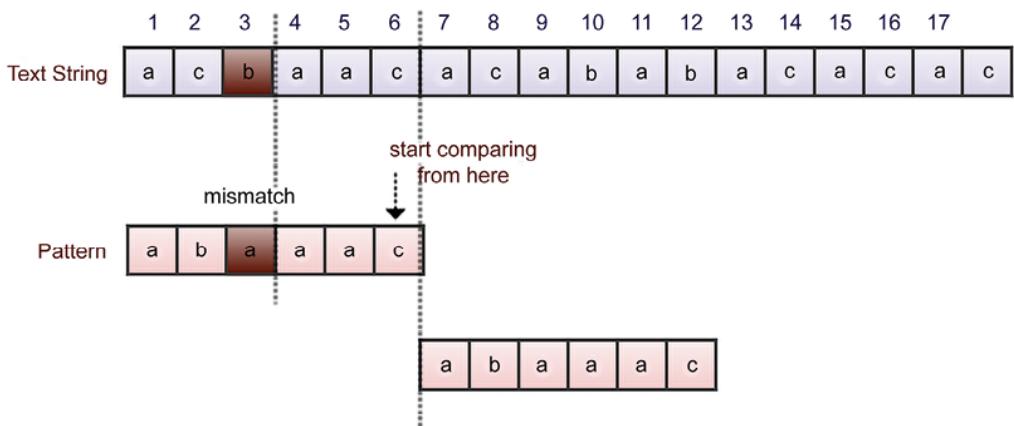


Figure 13.22: Fourth example of the good suffix heuristic in the Boyer-Moore algorithm

In the Boyer-Moore algorithm, we compute the shifts given by the bad character and good suffix heuristics. Further, we shift the pattern by the longer of the distances given by the bad character and good suffix heuristics.

The Boyer-Moore algorithm has a time complexity of $O(mn)$ for the preprocessing of the pattern, and the searching has a time complexity of $O(mn)$, where m is the length of the pattern and n is the length of the text.

Next, let us discuss the implementation of the Boyer-Moore algorithm.

Implementing the Boyer-Moore algorithm

Let's understand the implementation of the Boyer-Moore algorithm. The complete implementation of the Boyer-Moore algorithm is as follows:

```
text = "acbaaacacababacacac"
pattern = "acacac"

matched_indexes = []

i=0
flag = True
while i<=len(text)-len(pattern):
    for j in range(len(pattern)-1, -1, -1):      #reverse searching
        if pattern[j] != text[i+j]:
            flag = False      #indicates there is a mismatch
        if j == len(pattern)-1:      #if good-suffix is not present
            if text[i+j] in pattern[0:j]:
                i=i+j-pattern[0:j].rfind(text[i+j])
                #i+j is index of bad character, this line is used
        else:
            . . .
```

```

        i=i+j+1      #if bad character is not present, jump
    else:
        k=1
        while text[i+j+k:i+len(pattern)] not in pattern[0:len]
            #used for finding sub part of a good-suffix
            k=k+1
        if len(text[i+j+k:i+len(pattern)]) != 1:      #good-suffix
            gsshift=i+j+k-pattern[0:len(pattern)-1].rfind(text)
            #jumps pattern to a position where good-suffix of
        else:
            gsshift=i+len(pattern)
            gsshift=0  #when good-suffix heuristic is not appropriate
                        #we prefer bad character heuristic
        if text[i+j] in pattern[0:j]:
            bcshift=i+j-pattern[0:j].rfind(text[i+j])
            #i+j is index of bad character, this line is used
        else:
            bcshift=i+j+1
            i=max((bcshift, gsshift))
        break
    if flag:      #if pattern is found then normal iteration
        matched_indexes.append(i)
        i = i+1
    else:      #again set flag to True so new string in text can be examined
        flag = True

print ("Pattern found at", matched_indexes)

```

An explanation of each of the statements of the preceding code is presented here. Initially, we have the text string and the pattern. After initializing the variables, we start with a `while` loop that starts by comparing the last character of the pattern with the corresponding character of the text.

Then, the characters are compared from right to left by the use of the nested loop from the last index of the pattern to the first character of the pattern. This uses `range(len(pattern)-1, -1, -1)`.

The outer `while` loop keeps track of the index in the text string while the inner `for` loop keeps track of the index position in the pattern.

Next, we start comparing the characters by using `pattern[j] != text[i+j]`. If they are mismatched, we make the `flag` variable `False`, denoting that there is a mismatch.

Now, we check whether the good suffix is present using the condition `j == len(pattern)-1`. If this condition is true, it means that there is no good suffix possible, so we check for the bad character heuristics, that is, whether a mismatched character is present in the pattern using the condition `text[i+j] in pattern[0:j]`, and if the condition is true, then it means that the bad character is present in the pattern. In this case, we move the pattern to align this bad character to the other occurrence of this character in the pattern by using `i=i+j-pattern[0:j].rfind(text[i+j])`. Here, `(i+j)` is the index of the bad character.

If the bad character is not present in the pattern (it isn't in the `else` part of it), we move the whole pattern next to the mismatched character by using the index `i=i+j+1`.

Next, we go into the `else` part of the condition to check the good suffix. When we find the mismatch, we further test to see whether we have any subpart of a good suffix present in the prefix of the pattern. We do this using the following condition:

```
text[i+j+k:i+len(pattern)] not in pattern[0:len(pattern)-1]
```

Furthermore, we check whether the length of the good suffix is `1` or not. If the length of the good suffix is `1`, we do not consider this shift.

If the good suffix is more than 1, we find out the number of shifts by using the good suffix heuristics and store this in the `gsshift` variable. This is the pattern, which leads to a position where the good suffix of the pattern matches the good suffix in the text using the instruction

```
gsshift=i+j+k-
```

```
pattern[0:len(pattern)-1].rfind(text[i+j+k:i+len(pattern)]).
```

Furthermore, we computed the number of shifts possible due to the bad character heuristic and stored this in the `bcshift` variable. The number of shifts possible is `i+j-pattern[0:j].rfind(text[i+j])` when the bad character is present in the pattern, and the number of shifts possible would be `i+j+1` in the case of the bad character not being present in the pattern.

Next, we shift the pattern on the text string by the maximum number of moves given by the bad character and good suffix heuristics by using the instruction `i=max((bcshift, gsshift))`. Finally, we check whether the `flag` variable is `True` or not. If it is `True`, this means that the pattern has been found and that the matched index has been stored in the `matched_indexes` variable.

We have discussed the concept of the Boyer-Moore pattern matching algorithm, which is an efficient algorithm that skips unnecessary comparisons using the bad character and good suffix heuristics.

Summary

In this chapter, we have discussed the most popular and important string matching algorithms that have a wide range of applications in real-time scenarios. We discussed the brute force, Rabin-Karp, KMP, and Boyer-Moore pattern matching algorithms. In string matching

algorithms, we try to uncover ways to skip unnecessary comparisons and move the pattern over the text as fast as possible. The **KMP algorithm** detects unnecessary comparisons by looking at the overlapping substrings in the pattern itself to avoid redundant comparisons. Furthermore, we discussed the **Boyer-Moore algorithm**, which is very efficient when the text and pattern are long. It is the most popular algorithm used for string matching in practice.

Exercise

1. Show the KMP `prefix` function for the pattern "aabaaabcab".
2. If the expected number of valid shifts is small and the modulus is larger than the length of the pattern, then what is the matching time of the Rabin-Karp algorithm?
 - a. Theta (m)
 - b. Big O (n+m)
 - c. Theta (n-m)
 - d. Big O (n)
3. How many spurious hits does the Rabin-Karp string matching algorithm encounter in the text $T = "3141512653849792"$ when looking for all occurrences of the pattern $P = "26"$, working modulo $q = 11$, and over the alphabet set $\Sigma = \{0, 1, 2, \dots, 9\}$?
4. What is the basic formula applied in the Rabin-Karp algorithm to get the computation time as Theta (m)?
 - a. Halving rule
 - b. Horner's rule
 - c. Summation lemma

- d. Cancellation lemma
- 5. The Rabin-Karp algorithm can be used for discovering plagiarism in text documents.
 - a. True
 - b. False

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>



Appendix

Answers to the Questions

Chapter 2: Introduction to Algorithm Design

Question 1

Find the time complexity of the following Python snippets:

a.

```
i=1
while(i<n):
    i*=2
    print("data")
```

b.

```
i =n
while(i>0):
    print("complexity")
    i/ = 2
```

c.

```
for i in range(1,n):
    j = i
    while(j<n):
        j*=2
```

d.

```
i=1
while(i<n):
    print("python")
    i = i**2
```

Solution

a. The complexity will be $O(\log(n))$.

As we are multiplying the integer `i` by `2` in each step there will be exactly $\log(n)$ steps. (`1, 2, 4, till n`).

b. The complexity will be $O(\log(n))$.

As we are dividing the integer `i` by `2` in each step there will be exactly $\log(n)$ steps. (`n, n/2, n/4, till 1`).

c. The outer loop will run `n` times for each `i` in the outer loop, while the inner `while` loop will run $\log(i)$ times because we are multiplying each of the `j` values by `2` until it is less than `n`. Hence, there will be a maximum of $\log(n)$ steps in the inner loop. Therefore, the overall complexity will be $O(n\log(n))$.

In this code snippet, the `while` loop will execute based on the value of `i` until the condition becomes `false`. The value of `i` is incrementing in the following series:

`2, 4, 16, 256, ... n`

We can see that the number of times the loop is executing is $\log_2(\log_2(n))$ for a given value of `n`. So, for this series there will be exactly $\log_2(\log_2(n))$ executions of the loop. Hence the time complexity will be $O(\log_2(\log_2(n)))$.

Chapter 3: Algorithm Design Techniques and Strategies

Question 1

Which of the following options will be correct when a top-down approach of dynamic programming is applied to solve a given problem related to the space and time complexity?

- a. It will increase both time and space complexity
- b. It will increase the time complexity, and decrease the space complexity
- c. It will increase the space complexity, and decrease the time complexity
- d. It will decrease both time and space complexities

Solution

Option c is correct.

Since the top-down approach of dynamic programming uses the memoization technique, which stores the pre-calculated solution of a subproblem. It avoids recalculation of the same subproblem that decreases the time complexity, but at the same time, the space complexity will increase because of storing the extra solutions of the subproblems.

Question 2

What will be the sequence of nodes in the following edge-weighted directed graph using the greedy approach (assume node A as the source)?

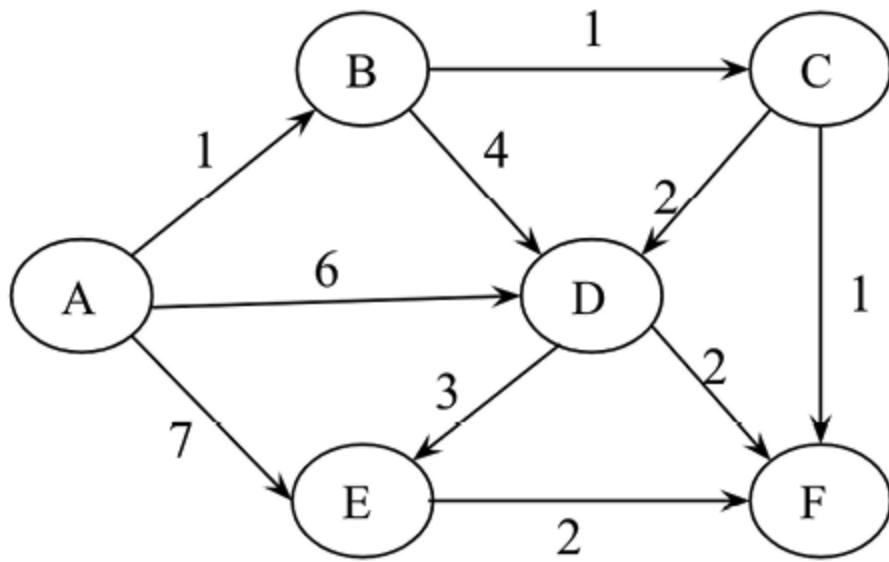


Figure A.1: A weighted directed graph

Solution

A, B, C, F, E, D

In Dijkstra's algorithm, at each point we choose the smallest weight edge, which starts from any one of the vertices in the shortest path found so far, and add it to the shortest path.

Question 3

Consider the weights and values of the items in *Table 3.8*. Note that there is only one unit of each item.

Item	Weight	Value
A	2	10
B	10	8
C	4	5
D	7	6

Table A.1: The weights and values of different items

We need to maximize the value; the maximum weight should be 11 kg. No item may be split. Establish the values of the items using a greedy approach.

Solution

Firstly, we picked item A (weight 2 kg) as the value is the maximum (10). The second highest value is for item B, but as the total weight becomes 12 kg, this violates the given condition, so we cannot pick it. The next highest value is item D, and now the total weight becomes $2+7 = 9$ kg (item A + item D). The next remaining item, C, cannot be picked because after adding it, the total weight condition will be violated.

So, the total value of the items picked up using the greedy approach
 $= 10 + 6 = 16$

Chapter 4: Linked Lists

Question 1

What will be the time complexity when inserting a data element after an element that is being pointed to by a pointer in a linked list?

Solution

It will be $O(1)$, since there is no need to traverse the list to reach the desired location where a new element is to be added. A pointer is pointing to the current location, and a new element can be directly added by linking it.

Question 2

What will be the time complexity when ascertaining the length of the given linked list?

Solution

$O(n)$.

In order to find out the length, each node of the list has to be traversed, which will take $O(n)$.

Question 3

What will be the worst-case time complexity for searching a given element in a singly linked list of length n ?

Solution

$O(n)$.

In the worst case, the data element to be searched will be at the end of the list, or will not be present in the list. In that case, there will be a total n number of comparisons, thus making the worst-case time complexity $O(n)$.

Question 4

For a given linked list, assuming it has only one `head` pointer that points to the starting point of the list, what will be the time complexity for the following operations?

- a. Insertion at the front of the linked list
- b. Insertion at the end of the linked list
- c. Deletion of the front node of the linked list
- d. Deletion of the last node of the linked list

Solution

- a. $O(1)$. This operation can be performed directly through the `head` node.
- b. $O(n)$. It will require traversing the list to reach the end of the list.
- c. $O(1)$. This operation can be performed directly through the `head` node.

d. $O(n)$. It will require traversing the list to reach the end of the list.

Question 5

Find the n^{th} node from the end of a linked list.

Solution

In order to find out the n^{th} node from the end of the linked list, we can use two pointers – `first` and `second`. Firstly, move the second pointer to `n` nodes from the starting point. Then, move both the pointers one step at a time until the second pointer reaches the end of the list. At that time, the first pointer will point to the n^{th} node from the end of the list.

Question 6

How can you establish whether there is a loop (or circle) in a given linked list?

Solution

To find out the loop in a linked list, it is most efficient to use **Floyd's cycle-finding algorithm**. In this approach, two pointers are used to detect the loop – let's say the first and second pointers. We start moving both the pointers from the starting point of the list.

We move the first and second pointers by one and two nodes at a time. If these two pointers meet at the same node, that indicates that there is a loop, otherwise, there is no loop in the given linked list.

The process is shown in the below figure with an example:

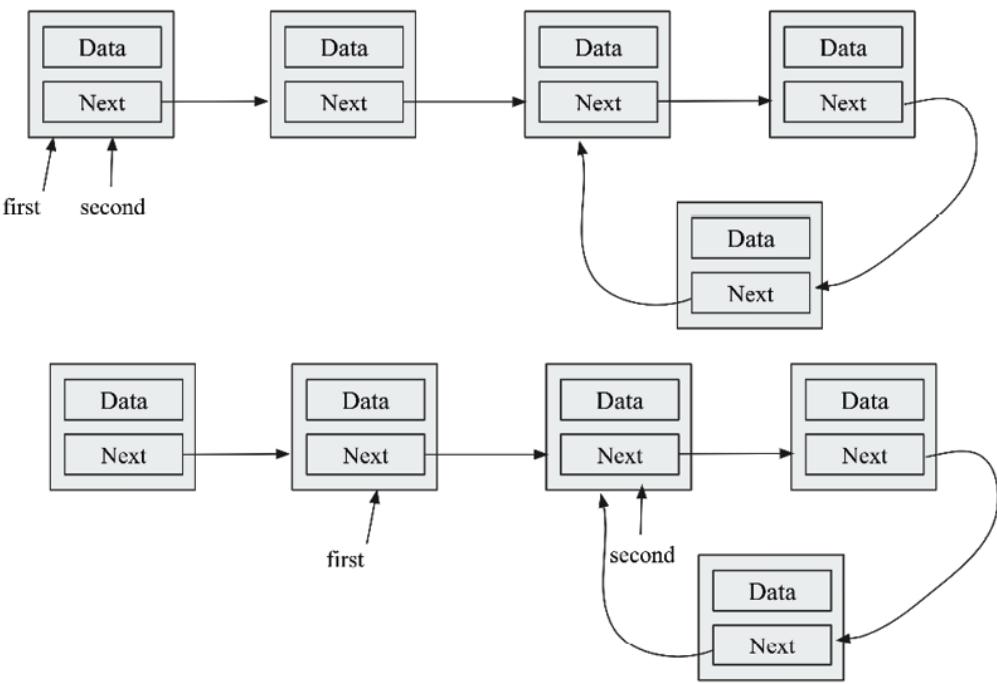


Figure A.2: Loop in a singly linked list

Question 7

How can you ascertain the middle element of the linked list?

Solution

It can be done with two pointers, say, the first and second pointers. Start moving these two pointers from the starting node. The first and second pointers should move one and two nodes at a time, respectively. When the second node reaches the end of the list, the first node will point to the middle element of the singly linked list.

Chapter 5: Stacks and Queues

Question 1

Which of the following options is a true queue implementation using linked lists?

- a. If, in the enqueue operation, new data elements are added at the start of the list, then the dequeue operation must be performed from the end.
- b. If, in the enqueue operation, new data elements are added to the end of the list, then the enqueue operation must be performed from the start of the list.
- c. Both of the above.
- d. None of the above.

Solution

B is correct. The queue data structure follows a FIFO order, hence data elements must be added to the end of the list, and then removed from the front.

Question 2

Assume a queue is implemented using a singly linked list that has `head` and `tail` pointers. The enqueue operation is implemented at `head`, and the dequeue operation is implemented at the `tail` of the

queue. What will be the time complexity of the enqueue and dequeue operations?

Solution

The time complexity of the enqueue operation will be $O(1)$ and $O(n)$ for the dequeue operation. As for the enqueue operation, we only need to delete the `head` node, which can be achieved in $O(1)$ for a singly linked list. For the dequeue operation, to delete the `tail`, we need to traverse the whole list first to the `tail`, and then we can delete it. For this we need linear, $O(n)$, time.

Question 3

What is the minimum number of stacks required to implement a queue?

Solution

Two stacks.

Using two stacks and the enqueue operation, the new element is entered at the top of `stack1`. In the dequeue process, if `stack2` is empty, all the elements are moved to `stack2`, and finally, the top of `stack2` is returned.

Question 4

The enqueue and dequeue operations in a queue are implemented efficiently using an array. What will be the time complexity for both

of these operations?

Solution

$O(1)$ for both operations.

If we use a circular array for the implementation of a queue, then we do not need to shift the elements, just the pointers, so we can implement both the enqueue and dequeue operations in $O(1)$ time.

Question 5

How can we print the data elements of a queue data structure in reverse order?

Solution

Make an empty stack, then enqueue each of the elements from the queue and push them into the stack. After the queue is empty, start popping out the elements from the stack and then printing them one by one.

Chapter 6: Trees

Question 1

Which of the following is true about binary trees:

- a. Every binary tree is either complete or full
- b. Every complete binary tree is also a full binary tree
- c. Every full binary tree is also a complete binary tree
- d. No binary tree is both complete and full
- e. None of the above

Solution

Option A is incorrect since it is not compulsory that a binary tree should be complete or full.

Option B is incorrect since a complete binary tree can have some nodes that are not filled in the last level, so a complete binary tree will not always be a full binary tree.

Option C is incorrect, as it is not always true, the following figure is a full binary tree, but not a complete binary tree:

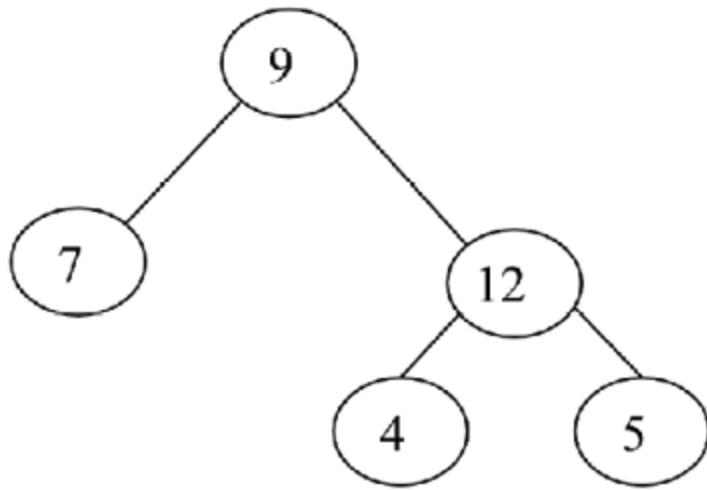


Figure A.3: A binary tree that is full, but not complete

Option D is incorrect, as it is not always true. The following tree is both a complete and full binary tree:

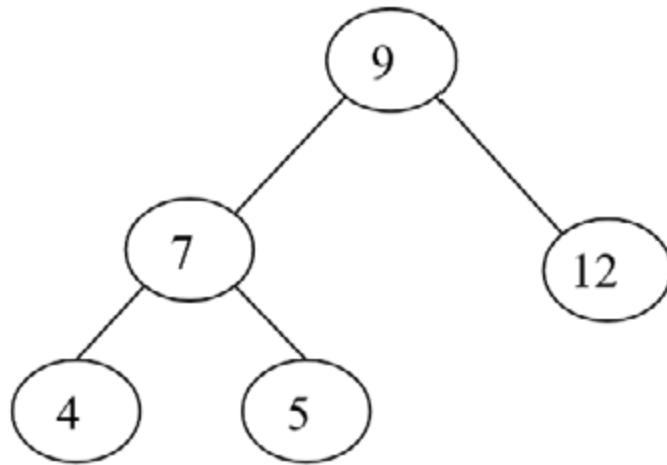


Figure A.4: A binary tree, that is full and complete

Question 2

Which of the tree traversal algorithms visit the root node last?

Solution

`postorder` traversal.

Using `postorder` traversal, we first visit the left subtree, then the right subtree, and finally we visit the `root` node.

Question 3

Consider this binary search tree:

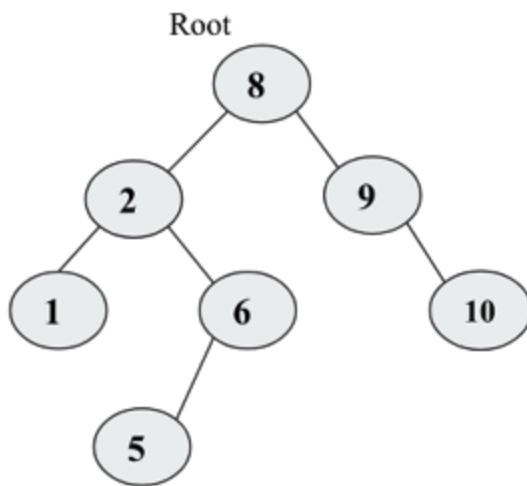


Figure A.5: Sample binary search tree

Suppose we remove the root node `8`, and we wish to replace it with any node from the left subtree then what will be the new root?

Solution

The new node will be node `6`. To maintain the properties of the binary search tree, the maximum value from the left subtree should be the new root.

Question 4

What will be the `inorder`, `postorder`, and `preorder` traversal of the following tree?

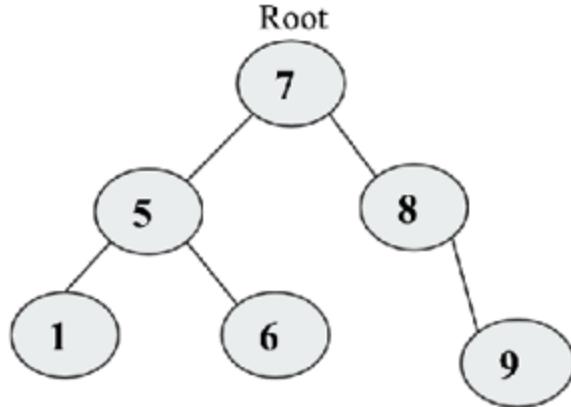


Figure A.6: Example tree

Solution

The `preorder` traversal will be `7-5-1-6-8-9`.

The `inorder` traversal will be `1-5-6-7-8-9`.

The `postorder` traversal will be `1-6-5-9-8-7`.

Question 5

How do you find out if two trees are identical?

Solution

In order to find out if two binary trees are identical or not, both of the trees should have exactly the same data and element

arrangement. This can be done by traversing both of the trees with any of the traversal algorithms (it should be the same for both trees) and matching them element by element. If all the elements are the same in traversing both of the trees, then the trees are identical.

Question 6

How many leaves are there in the tree mentioned in *question 4*?

Solution

Three, nodes 1, 6, and 9.

Question 7

What is the relation between a perfect binary tree's height and the number of nodes in that tree?

Solution

$$\log_2(n+1) = h.$$

The number of nodes in each level:

Level 0: $2^0 = 1$ nodes

Level 1: $2^1 = 2$ nodes

Level 2: $2^2 = 4$ nodes

Level 3: $2^3 = 8$ nodes

The total nodes at level h can be computed by adding all nodes in each level:

$$n = 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{h-1} = 2^h - 1$$

So, the relationship between n and h is: $n = 2^h - 1$

$$= \log(n+1) = \log 2^h$$

$$= \log_2(n+1) = h$$

Chapter 7: Heaps and Priority Queues

Question 1

What will be the time complexity for deleting an arbitrary element from the `min-heap`?

Solution

To delete any element from the `heap`, we first have to search the element that is to be deleted, and then we delete the element.

Total time complexity = Time for searching the element + Deleting the element

$$= O(n) + O(\log n)$$

$$= O(n)$$

Question 2

What will be the time complexity for finding the k^{th} smallest element from the `min-heap`?

Solution

The k^{th} element can be found out from the `min-heap` by performing `delete` operations k times. For each `delete` operation, the time

complexity is $O(\log n)$. So, the total time complexity for finding out the k^{th} smallest element will be $O(k \log n)$.

Question 3

What will be the time complexity to make a **max-heap** that combines two **max-heap** each of size n ?

Solution

$O(n)$.

Since the time complexity of creating a **heap** from n elements is $O(n)$, creating a **heap** of $2n$ elements will also be $O(n)$.

Question 4

What will be the worst-case time complexity for ascertaining the smallest element from a binary max-heap and binary min-heap?

Solution

In a max-heap, the smallest element will always be present at a leaf node. So, in order to find out the smallest element, we have to search all the leaf nodes. So, the worst-case complexity will be $O(n)$.

The worst-case time complexity to find out the smallest element in the min-heap will be $O(1)$ since it will always be present at the root node.

Question 5

The level order traversal of **max-heap** is 12, 9, 7, 4, 2. After inserting new elements 1 and 8, what will be the final **max-heap** and level order traversal of the final **max-heap**?

Solution

The **max-heap** after the insertion of element 1 is shown in the below figure:

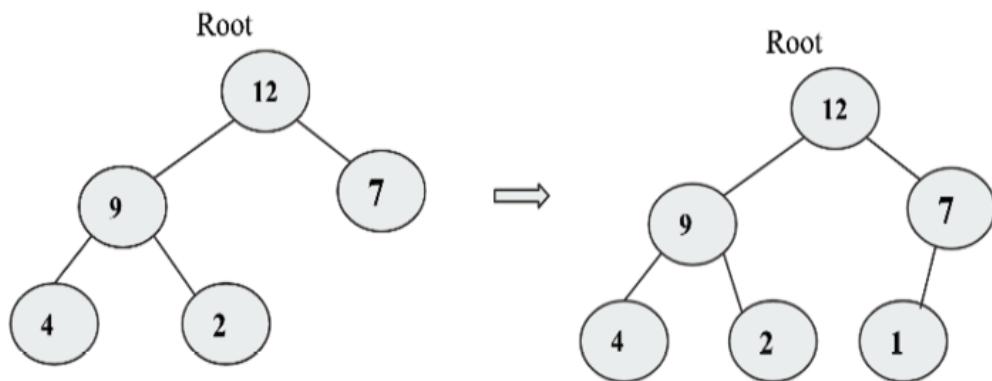


Figure A.7: The max-heap before insertion of element 8

The final **max-heap** after the insertion of element 8 is shown in the below figure:

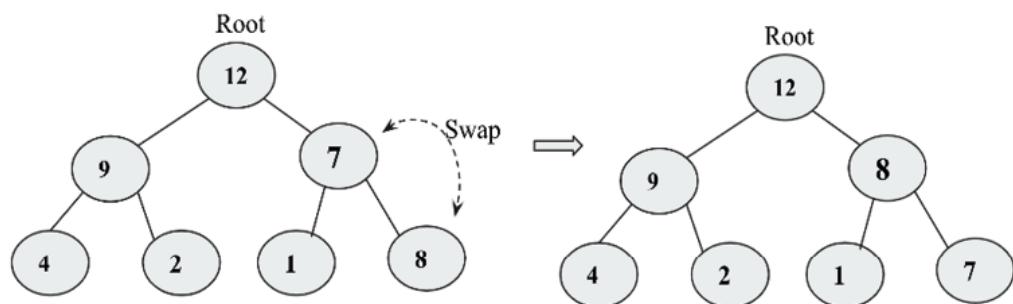


Figure A.8: The max-heap after the insertion of elements 1 and 8

The level order traversal of the final max-heap will be 12, 9, 8, 4, 2, 1, 7.

Question 6

Which of the following is a binary max-heap?

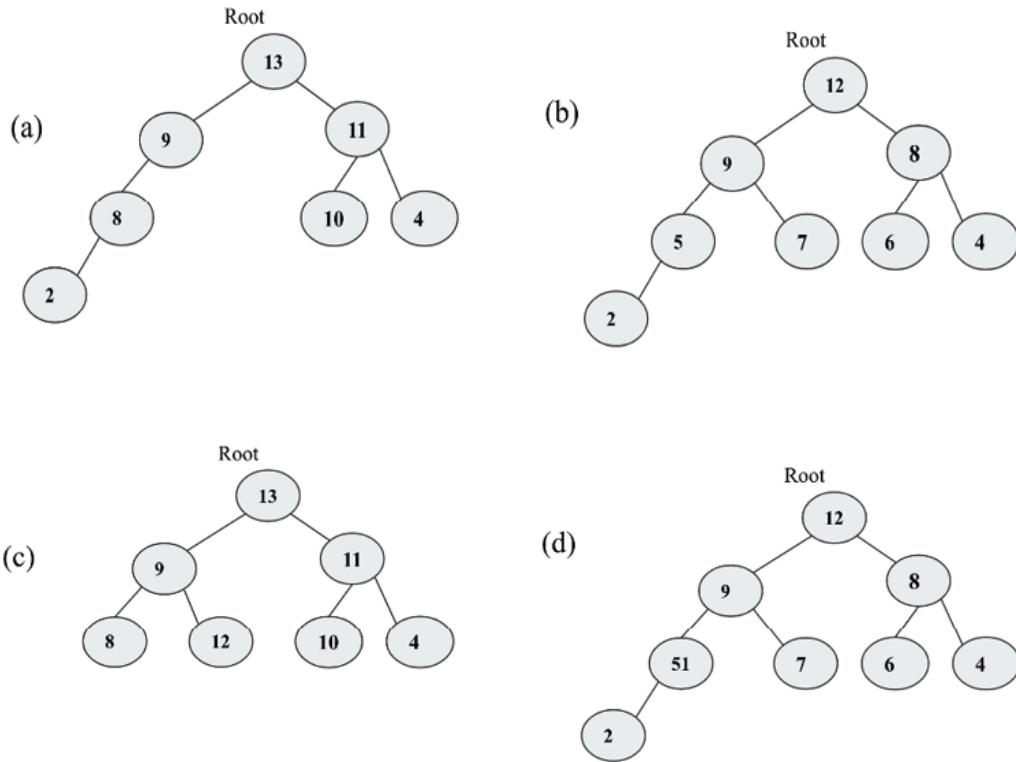


Figure A.9: Example trees

Solution

B.

A binary max-heap should be a complete binary tree and all the levels should be filled, except the last level. The value of the parent should

be greater or equal to the values of its children.

Option A is not correct because it is not a complete binary tree.

Options C and D are not correct because they are not fulfilling the `heap` property. Option B is correct because it is complete and fulfills the `heap` property.

Chapter 8: Hash Tables

Question 1

There is a hash table with 40 slots and there are 200 elements stored in the table. What will be the load factor of the hash table?

Solution

The load factor of the hash table = (no. of elements) / (no. of table slots) = $200/40 = 5$.

Question 2

What is the worst-case search time of hashing using a separate chaining algorithm?

Solution

The worst-case time complexity for searching in a separate chaining algorithm using linked lists is $O(n)$, because in the worst case, all the items will be added to `index 1` in a linked list, searching an item will work similarly to a linked list.

Question 3

Assume a uniform distribution of keys in the hash table. What will be the time complexities for the `search/insert/delete` operations?

Solution

The index of the hash table is computed from the key in $O(1)$ time when the keys are uniformly distributed in the hash table. The creation of the table will take $O(n)$ time, and other operations such as `search`, `insert`, and `delete` operations will take $O(1)$ time because all the elements are uniformly distributed, hence, we directly get the required element.

Question 4

What will be the worst-case complexity for removing the duplicate characters from an array of characters?

Solution

The brute force algorithm starts with the first character and searches linearly with all the characters of the array. If a duplicate character is found, then that character should be swapped with the last character and then the length of the string should be decremented by one. The same process is repeated until all characters are processed. The time complexity of this process is $O(n^2)$.

It can be implemented more efficiently using a hash table in $O(n)$ time.

Using this method, we start with the first character of the array and store it in the hash table according to the hash value. We do it for all the characters. If there is any collision, then that character can be ignored, otherwise, the character is stored in the hash table.

Chapter 9: Graphs and Algorithms

Question 1

What is the maximum number of edges (without self-loops) possible in an undirected simple graph with five nodes?

Solution

Each node can be connected to every other node in the graph. So, the first node can be connected to $n-1$ nodes, the second node can be connected to $n-2$ nodes, the third node can be connected to $n-3$ nodes, and so on. The total number of nodes will be:

$$[(n-1)+(n-2)+ \dots +3+2+1] = n(n-1)/2.$$

Question 2

What do we call a graph in which all the nodes have an equal degree?

Solution

A complete graph.

Question 3

Explain what cut vertices are and identify the cut vertices in the given graph?

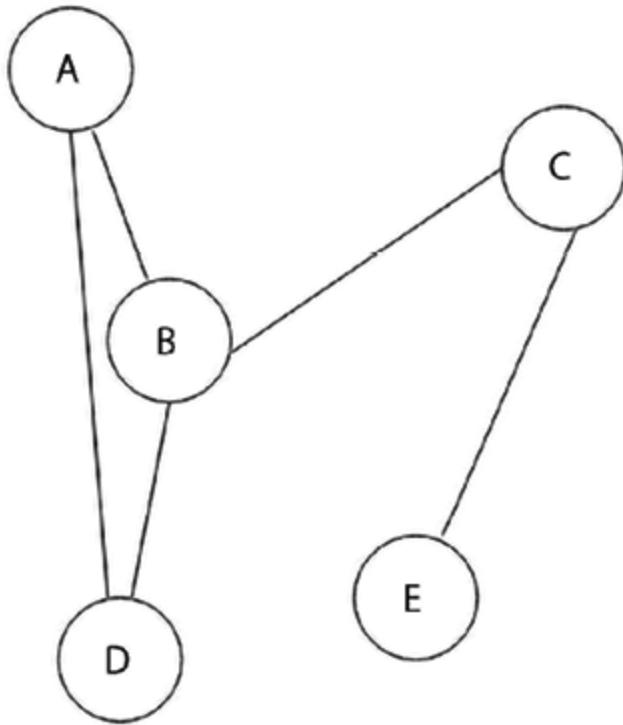


Figure A.10: Sample graph

Solution

Cut vertices also known as articulation points. These are the vertices in the graph, after removal of which, the graph splits in two disconnected components. In the given graph, the vertices **B**, and **C** are cut vertices since after removal of node **B**, the graph will split into **{A, D}**, **{C, E}** vertices. And, after removal of node **C**, the graph will split into **{A, B, D}**, **{E}** vertices.

Question 4

Assuming a graph **G** of order **n**, what will be the maximum number of cut vertices possible in graph **G**?

Solution

It will be $n-2$, since the first and last vertices will not be cut vertices, except those two nodes, all nodes can split the graph into two disconnected graphs. See the below graph:

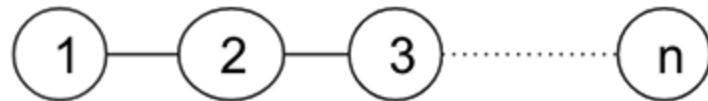


Figure A.11: A graph G

Chapter 10: Searching

Question 1

On average, how many comparisons are required in a linear search of n elements?

Solution

The average number of comparisons in linear search will be as follows. When a search element is found at the 1st position, 2nd position, 3rd position, and similarly at the n th position, correspondingly, it will require 1, 2, 3... n number of comparisons.

Total average number of comparisons

$$\begin{aligned} &= \frac{(1 + 2 + 3 + \dots + n)}{n} \\ &= \frac{\frac{n(n + 1)}{2}}{n} \\ &= \frac{(n + 1)}{2} \end{aligned}$$

Question 2

Assume there are eight elements in a sorted array. What is the average number of comparisons that will be required if all the searches are successful and if the binary search algorithm is used?

Solution

$$\text{Average number of comparisons} = (1+2+2+3+3+3+4)/8$$

$$= 21/8$$

$$= 2.625$$

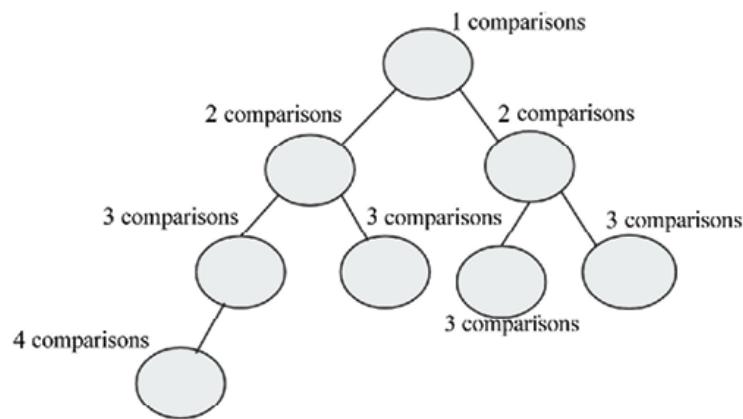
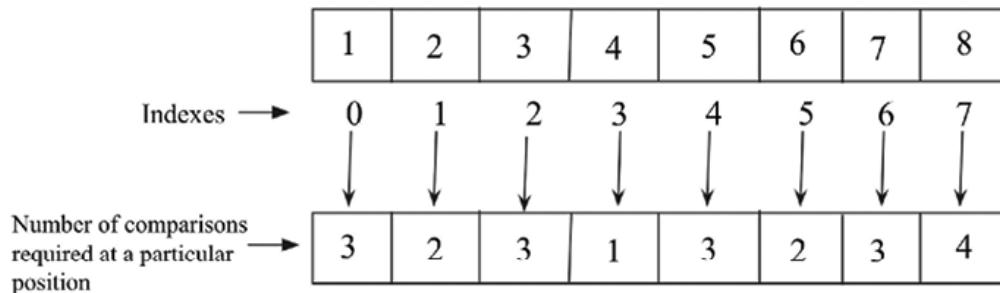


Figure A.12: Demonstration of number of the comparisons in the given array

Question 3

What is the worst-case time complexity of the binary search algorithm?

Solution

$O(\log n)$.

The worst-case scenario of the binary search algorithm will occur when the desired element is present in the first position or at the last position. In that case, $\log(n)$ comparisons will be required. Hence the worst-case complexity will be $O(\log n)$.

Question 4

When should the interpolation search algorithm perform better than the binary search algorithm?

Solution

The interpolation search algorithm performs better than the binary search algorithm when the data items in the array are uniformly distributed.

Chapter 11: Sorting

Question 1

If an array `arr = {55, 42, 4, 31}` is given and bubble sort is used to sort the array elements, then how many passes will be required to sort the array?

- a. 3
- b. 2
- c. 1
- d. 0

Solution

The answer is a. To sort n elements, the bubble sort algorithm requires $(n-1)$ iterations (passes), where n is the number of elements in the given array. Here in the question, the value of $n = 4$, so $4-1 = 3$ iterations will be required to sort the given array.

Question 2

What is the worst-case complexity of bubble sort?

- a. $O(n \log n)$
- b. $O(\log n)$
- c. $O(n)$
- d. $O(n^2)$

Solution

The answer is d. The worst case appears when the given array is in reverse order. In that case, the time complexity of bubble sort would be $O(n^2)$.

Question 3

Apply quicksort to the sequence (56, 89, 23, 99, 45, 12, 66, 78, 34).

What is the sequence after the first phase, and what pivot is the first element?

- a. 45, 23, 12, 34, 56, 99, 66, 78, 89
- b. 34, 12, 23, 45, 56, 99, 66, 78, 89
- c. 12, 45, 23, 34, 56, 89, 78, 66, 99
- d. 34, 12, 23, 45, 99, 66, 89, 78, 56

Solution

b.

After the first phase, 56 would be in the right position so that all the elements smaller than 56 will be on the left side of it, and elements bigger than 56 will be on the right side of it. Further, quicksort is applied recursively to the left subarray and right subarray. The process of the quicksort for the given sequence, as shown in the below figure.

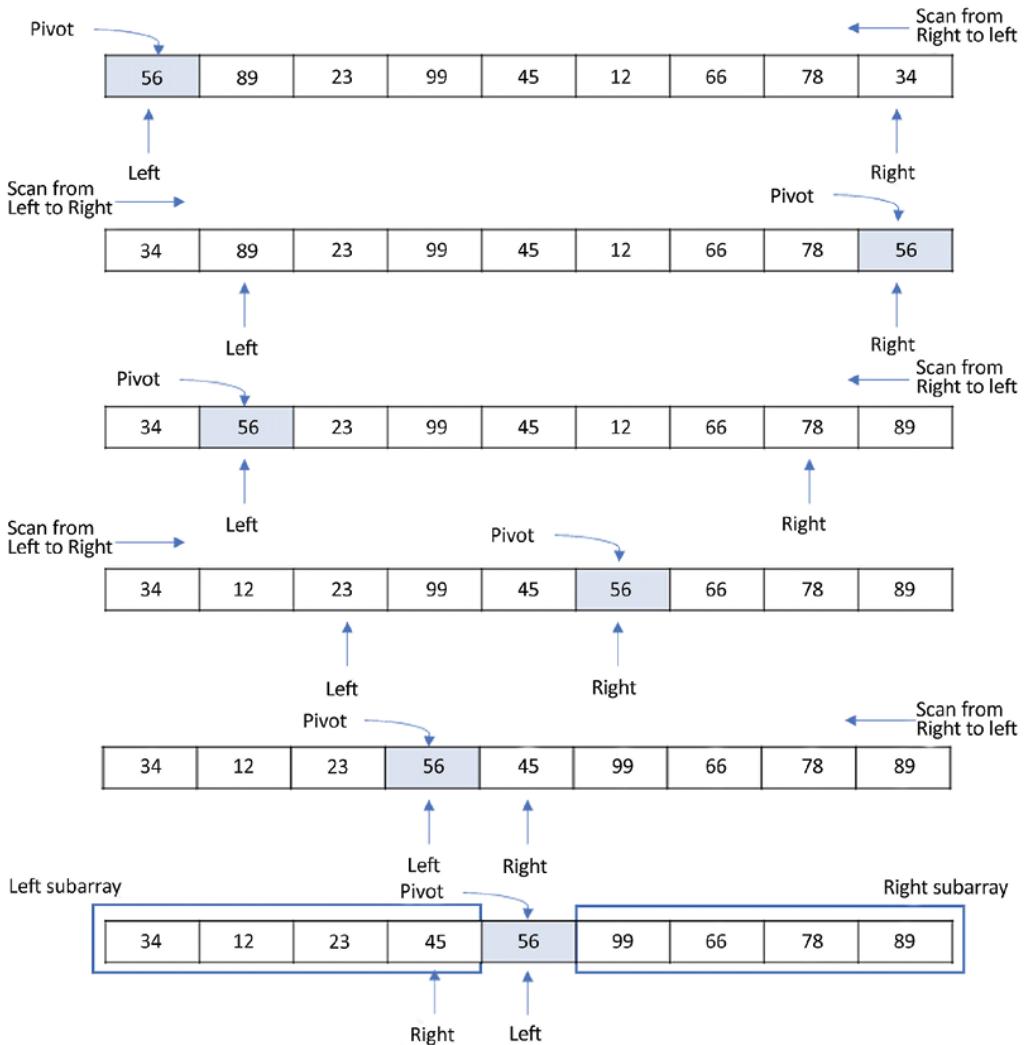


Figure A.13: Demonstration of the quicksort algorithm

Question 4

Quicksort is a _____

- Greedy algorithm
- Divide-and-conquer algorithm
- Dynamic programming algorithm
- Backtracking algorithm

Solution

The answer is b. Quicksort is a divide-and-conquer algorithm. Quick sort first partitions a large array into two smaller sub arrays and then recursively sorts the sub-arrays. Here, we find the pivot element such that all elements to the left side of the pivot element would be smaller than the pivot element and create the first subarray. The elements to the right side of the pivot element are greater than the pivot element and create the second subarray. Thus, the given problem is reduced into two smaller sets. Now, sort these two subarrays again, finding the pivot element in each subarray, i.e. apply quicksort on each subarray.

Question 5

Consider a situation where a `swap` operation is very costly. Which of the following sorting algorithms should be used so that the number of `swap` operations is minimized?

- a. Heap sort
- b. Selection sort
- c. Insertion sort
- d. Merge sort

Solution

b. In the selection sort algorithm, in general, we identify the largest element, and then swap it with the last element so that in each iteration, only one `swap` is required. For n elements, the total $(n-1)$

swaps will be required, which is the lowest in comparison to all other algorithms.

Question 6

If the input array `A = {15, 9, 33, 35, 100, 95, 13, 11, 2, 13}` is given, using selection sort, what would be the order of the array after the fifth swap? (Note: it counts regardless of whether they exchange or remain in the same position.)

- a. 2, 9, 11, 13, 13, 95, 35, 33, 15, 100
- b. 2, 9, 11, 13, 13, 15, 35, 33, 95, 100
- c. 35, 100, 95, 2, 9, 11, 13, 33, 15, 13
- d. 11, 13, 9, 2, 100, 95, 35, 33, 13, 13

Solution

The answer is a. In selection sort, select the smallest element. Start the comparison from the beginning of the array and swap the smallest element with the first greatest element. Now, exclude the previous element that was chosen as the smallest element, as it has been put in the right place.

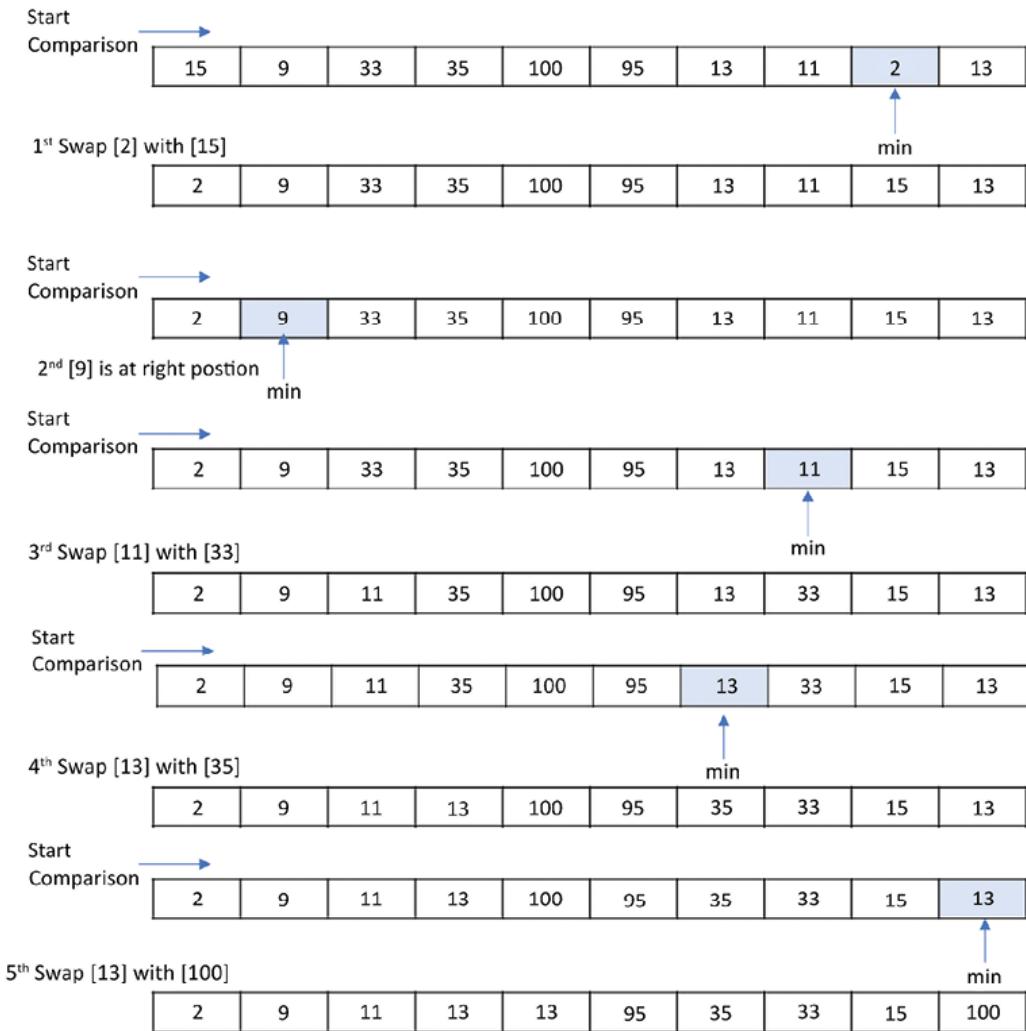


Figure A.14: Demonstration of insertion sort on the given sequence

Question 7

What will be the number of iterations to sort the elements {44, 21, 61, 6, 13, 1} using insertion sort?

- a. 6
- b. 5
- c. 7
- d. 1

Solution

The answer is a. Suppose there are N keys in an input list, then it requires N iterations to sort the entire list using insertion sort.

Question 8

How will the array elements $A = [35, 7, 64, 52, 32, 22]$ look after the second iteration, if the elements are sorted using insertion sort?

- a. 7, 22, 32, 35, 52, 64
- b. 7, 32, 35, 52, 64, 22
- c. 7, 35, 52, 64, 32, 22
- d. 7, 35, 64, 52, 32, 22

Solutions

d. Here $N = 6$. In the first iteration, the first element, that is, $A[1] = 35$, is inserted into array B , which is initially empty. In the second iteration, $A[2] = 7$ is compared with the elements in B starting from the rightmost element of B to find its place. So, after the second iteration, the input array would be $A = [7, 35, 64, 52, 32, 22]$.

Chapter 12: Selection Algorithm

Question 1

What will be the output if the quickselect algorithm is applied to the given array `arr=[3, 1, 10, 4, 6, 5]` with `k` given as 2?

Solution

1. Given the initial array: `[3, 1, 10, 4, 6, 5]`, we can find the median of medians: `4` (index = `3`).
2. We swap the pivot element with the first element: `[4, 1, 3, 10, 6, 5]`.
3. We will move the pivot element to its correct position: `[1, 3, 4, 10, 6, 5]`.
4. Now we get a split index equal to `2` but the value of `k` is also equal to `2`, hence the value at index `2` will be our output. Hence the output will be `4`.

Question 2

Can quickselect find the smallest element in an array with duplicate values?

Solution

Yes, it works. By the end of every iteration, we have all elements less than the current pivot stored to the left of the pivot. Let's consider

when all elements are the same. In this case, every iteration ends up putting a pivot element to the left of the array. And the next iteration will continue with one element shorter in the array.

Question 3

What is the difference between the quicksort algorithm and the quickselect algorithm?

Solution

In quickselect, we do not sort the array, and it is specifically for finding the k^{th} smallest element in the array. The algorithm repeatedly divides the array into two sections based on the value of the pivot element. As we know, the pivot element will be placed such that all the elements to its left are smaller than the pivot element, and all the elements to the right are larger than the pivot element. Thus, we can select any one of the segments of the array based on the target value. This way, the size of the operable range of our array keeps on reducing. This reduces the complexity from $O(n \log_2(n))$ to $O(n)$.

Question 4

What is the main difference between the deterministic selection algorithm and the quickselect algorithm?

Solution

In the `quickselect` algorithm, we find the k^{th} smallest element in an unordered list based on picking up the pivot element randomly. Whereas, in the deterministic selection algorithm, which is also used for finding the k^{th} smallest element from an unordered list, but in this algorithm, we choose a pivot element by using median of medians, instead of taking any random pivot element.

Question 5

What triggers the worst-case behavior of the selection algorithm?

Solution

Continuously picking the largest or smallest element on each iteration triggers the worst-case behavior of the selection algorithm.

Chapter 13: String Matching Algorithms

Question 1

Show the KMP `prefix` function for the pattern "aabaaabcab".

Solution

The `prefix` function values are given below:

pattern	a	a	b	a	a	b	c	a	b
prefix_function π	0	1	0	1	2	3	0	1	0

Table A.2: Prefix function for the given pattern

Question 2

If the expected number of valid shifts is small and the modulus is larger than the length of the pattern, then what is the matching time of the Rabin-Karp algorithm?

- a. Theta (m)
- b. Big O ($n+m$)
- c. Theta ($n-m$)
- d. Big O (n)

Solution

Big O ($n+m$)

Question 3

How many spurious hits does the Rabin-Karp string matching algorithm encounter in the text `T = "3141512653849792"` when looking for all occurrences of the pattern `P = "26"`, working modulo `q = 11` and over the alphabet set `$\Sigma = \{0, 1, 2, \dots, 9\}$` ?

Solution

2.

Question 4

What is the basic formula applied in the Rabin-Karp algorithm to get the computation time as Theta (m)?

- a. Halving rule
- b. Horner's rule
- c. Summation lemma
- d. Cancellation lemma

Solution

Horner's rule.

Question 5

The Rabin-Karp algorithm can be used for discovering plagiarism in text documents.

- a. True
- b. False

Solution

True, the Rabin-Karp algorithm is a string matching algorithm, and it can be used for detecting plagiarism in text documents.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers: <https://packt.link/MEvK4>





packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

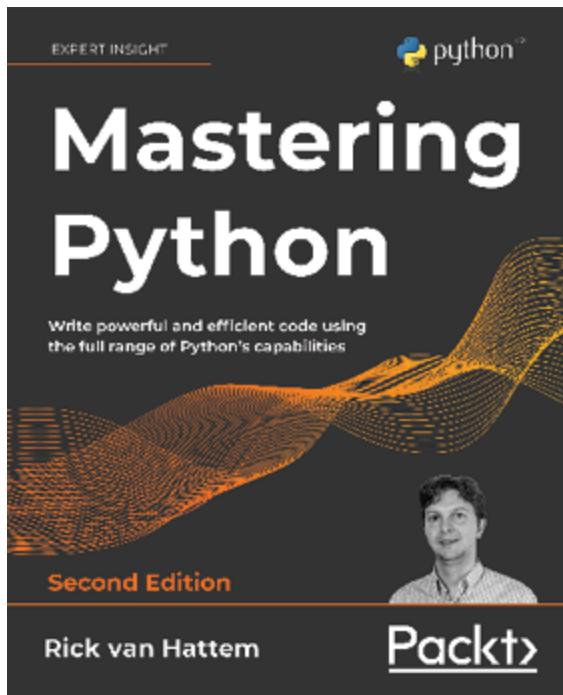
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.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 Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



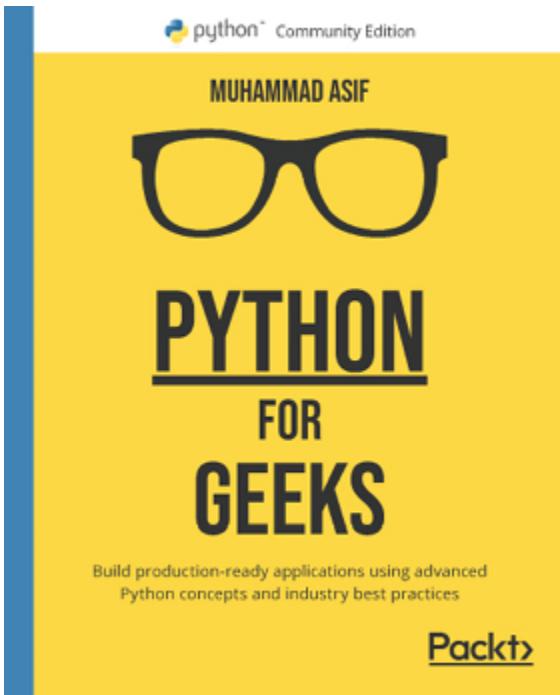
Mastering Python 2E

Rick van Hattem

ISBN: 978-1-80020-772-1

- Write beautiful Pythonic code and avoid common Python coding mistakes
- Apply the power of decorators, generators, coroutines, and metaclasses
- Use different testing systems like pytest, unittest, and doctest

- Track and optimize application performance for both memory and CPU usage
- Debug your applications with PDB, Werkzeug, and faulthandler
- Improve your performance through asyncio, multiprocessing, and distributed computing
- Explore popular libraries like Dask, NumPy, SciPy, pandas, TensorFlow, and scikit-learn
- Extend Python's capabilities with C/C++ libraries and system calls



Python for Geeks 2E

Muhammad Asif

ISBN: 978-1-80107-011-9

- Understand how to design and manage complex Python projects
- Strategize test-driven development (TDD) in Python
- Explore multithreading and multiprogramming in Python
- Use Python for data processing with Apache Spark and Google Cloud Platform (GCP)
- Deploy serverless programs on public clouds such as GCP
- Use Python to build web applications and application programming interfaces
- Apply Python for network automation and serverless functions
- Get to grips with Python for data analysis and machine learning

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight 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.

Share your thoughts

Now you've finished *Hands-On Data Structures and Algorithms with Python - Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here to go straight to the Amazon review page](#) for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

adjacency [282](#)

adjacency lists [287](#), [288](#)

adjacency matrix [288](#), [289](#), [290](#)

algorithm [35](#)

 benefits [36](#)

 criteria [36](#), [37](#)

 example [37](#)

 performance analysis [37](#), [38](#)

 running time complexity, computing [52](#), [53](#), [54](#)

algorithm design techniques [57](#), [58](#)

 divide-and-conquer [60](#), [61](#)

 dynamic programming [68](#), [69](#)

 greedy algorithms [74](#), [75](#), [76](#)

 recursion [59](#), [60](#)

algorithms [1](#)

amortized analysis [49](#)

 accounting method [50](#)

 aggregate analysis [50](#)

potential method [50](#)

Anaconda distribution

download link [5](#)

AND operator [17](#)

arithmetic expression [194](#)

infix notation [194](#)

postfix notation [194](#), [196](#)

prefix notation [194](#), [195](#)

arrays [94](#), [248](#)

used, for implementing stacks [145](#), [146](#), [147](#)

asymptotic notation [41](#)

Big O notation [44](#), [45](#), [46](#), [47](#)

for calculating running time complexity [42](#)

omega notation [47](#), [48](#), [49](#)

theta notation [42](#), [43](#), [44](#)

B

balanced binary tree [184](#)

base address [94](#)

base cases [59](#)

basic data types [7](#)

Boolean [8](#)

numeric [7](#), [8](#)

sequences [9](#)

tuples [18](#)

Big O notation [44](#), [45](#), [46](#), [47](#)

binary heap [222](#)

implementing [223](#)

binary search [61](#), [62](#)

binary search algorithm [325](#)

example [325](#)

implementation [326](#), [327](#), [328](#), [329](#), [330](#), [331](#)

binary search tree (BST) [201](#)

benefits [216](#), [217](#), [218](#), [219](#)

example [201](#), [202](#)

maximum node, finding [215](#), [216](#)

minimum node, finding [215](#), [216](#)

nodes, deleting [209](#), [210](#), [211](#), [212](#), [213](#), [214](#)

nodes, inserting [203](#), [204](#), [205](#), [206](#), [207](#)

operations [202](#)

tree, searching [208](#), [209](#)

binary search trees (BSTs) [273](#)

binary tree [181](#)

balanced binary tree [184](#)

complete binary tree [183](#)

example [182](#)

full binary tree [182](#)

nodes, implementing [184](#), [185](#), [186](#)

perfect binary tree [183](#)
regular binary tree [182](#)
simple binary tree [181](#)
unbalanced binary tree [184](#)

binary trees

applications [194](#)
expression trees [194](#)

bipartite graph [285](#)

Boolean [8](#)

Boyer-Moore algorithm [415](#)
bad character heuristic [417](#), [419](#), [420](#)
good character heuristic [420](#), [421](#), [422](#), [423](#), [424](#)
implementing [424](#), [425](#), [426](#)
working [416](#), [417](#)

breadth-first search (BFS) [291](#), [292](#), [293](#), [294](#), [295](#), [296](#), [297](#), [298](#)

brute force algorithm [397](#), [398](#), [399](#), [400](#)

brute-force approach [58](#)

bubble sort algorithms [346](#), [347](#), [349](#), [350](#), [352](#)

bucket [252](#)

C

ChainMap object [30](#)

child node [181](#)

circular linked list

element, deleting [134](#), [135](#), [136](#), [138](#)

querying [134](#)

circular linked lists [129](#), [130](#)

creating [131](#)

items, appending [131](#), [132](#), [133](#)

traversing [131](#)

collections module [27](#)

data types [27](#)

operations [27](#)

collisions

resolving [252](#), [253](#)

command line

Python development environment, setting up via [3](#), [4](#)

complete binary tree [183](#), [222](#)

complex data types [19](#)

dictionary [19](#), [20](#)

set [23](#), [24](#)

complexity classes, algorithm

composing [50](#), [52](#)

complex number [8](#)

container datatypes, collections module

ChainMap object [30](#)

counter objects [31](#)

default dictionary [29](#), [30](#)

deque [28](#), [29](#)

named tuples [27](#), [28](#)

ordered dictionary [29](#)

UserDict [32](#)

UserList [32](#)

UserString [33](#)

counter objects [31](#)

D

data structures [1](#)

data types

overview [6](#)

default dictionary [29](#), [30](#)

degree

of vertex/node [282](#)

delete operation

implementing, in heap [229](#), [230](#), [231](#), [232](#), [233](#)

depth-first search (DFS) [299](#), [300](#), [301](#), [302](#), [303](#), [304](#), [305](#)

deque [28](#)

functions [29](#)

deterministic selection [383](#)

implementation [386](#), [387](#), [388](#), [389](#), [390](#), [391](#), [392](#), [393](#)

working [384](#), [385](#)

dictionaries [248](#)

dictionary [19](#), [20](#)

characteristics [21](#)

hash table, implementing as [263](#), [264](#)

methods [22](#), [23](#)

directed acyclic graph (DAG) [284](#), [285](#)

directed graph [283](#)

indegree [284](#)

isolated vertex [284](#)

outdegree [284](#)

sink vertex [284](#)

source vertex [284](#)

divide-and-conquer design technique [60](#), [61](#)

binary search [61](#), [62](#)

merge sort [63](#), [64](#), [65](#), [66](#), [68](#)

double hashing technique

for collision resolution [267](#), [269](#), [270](#), [271](#)

doubly linked lists [114](#)

creating [115](#)

items, appending [116](#)

items, deleting [124](#), [125](#), [126](#), [127](#), [128](#), [129](#)

node, inserting at beginning [116](#), [117](#)

node, inserting at end [119](#), [120](#)

node, inserting at intermediate position [121](#), [123](#)

querying [123](#), [124](#)

traversing [115](#)

dynamic programming [68](#)

bottom-up approach [70](#)

top-down with memoization [69](#)

dynamic programming problems

characteristics [69](#)

E

edge [181](#), [282](#)

elements

retrieving, from hash table [260](#), [262](#)

storing, in hash tables [257](#), [258](#)

empty tree [181](#)

exponential search algorithm [337](#)

definition [338](#)

example [338](#)

implementation [340](#), [341](#)

working [337](#)

expression trees [194](#)

reverse Polish expression, parsing [196](#), [197](#), [198](#), [200](#)

F

factorial [59](#)

Fibonacci series

calculating [70](#), [71](#), [73](#), [74](#)

first in first out (FIFO) [237](#)

first in, first out (FIFO) [157](#)

float type [8](#)

frozenset [26](#)

full binary tree [182](#)

example [182](#)

G

generator [99](#)

graph methods [305](#)

Kruskal's Minimum Spanning Tree [306](#), [307](#), [308](#), [309](#)

Minimum Spanning Tree (MST) [305](#), [306](#)

Prim's Minimum Spanning Tree [309](#), [310](#), [311](#)

graph representations [286](#)

adjacency lists [287](#), [288](#)

adjacency matrix [288](#), [289](#), [290](#)

graphs [281](#)

adjacency [282](#)

bipartite graphs [285](#)

degree of vertex/node [282](#)

directed acyclic graph (DAG) [284](#), [285](#)

directed graphs [283](#)

edge [282](#)

example [282](#)

leaf vertex [282](#)

loop [282](#)

node [282](#)

path [282](#)

undirected graphs [283](#)

vertex [282](#)

weighted graphs [285](#)

graph traversals [291](#)

breadth-first search (BFS) [291](#), [292](#), [293](#), [294](#), [295](#), [296](#), [297](#), [298](#)

depth-first search (DFS) [299](#), [300](#), [301](#), [302](#), [303](#), [304](#), [305](#)

greedy algorithms [74](#)

examples [75](#), [76](#)

shortest path problem [76](#), [77](#), [78](#), [79](#), [80](#), [81](#), [82](#), [84](#), [85](#), [86](#), [87](#), [88](#), [89](#)

H

hashing functions [249](#), [250](#)

perfect hashing functions [251](#), [252](#)

hash tables [247](#), [248](#)

elements, retrieving from [260](#), [262](#)

elements, storing [257](#), [258](#)

example [248](#)

growing [258](#), [259](#), [260](#)
implementing [256](#), [257](#)
implementing, as dictionary [263](#), [264](#)
testing [262](#), [263](#)

heap data structure [221](#)

binary heap [222](#)
binary heap example [223](#)
delete operation [229](#), [230](#), [231](#), [232](#), [233](#)
element, deleting at specific location [234](#), [235](#)
heap sort [236](#), [237](#)
insert operation [224](#), [225](#), [226](#), [227](#), [228](#)
max heap [221](#)
max heap example [222](#)
min heap [222](#)
min heap example [222](#)

|

identity operators [16](#), [17](#)
immutable sets [26](#)
indegree [284](#)
infix notation [194](#)
in operator [15](#)
in-order tree traversal [186](#), [187](#), [188](#)
insertion sort algorithm [352](#), [353](#), [354](#)

insert operation

implementing, in heap [224](#), [225](#), [226](#), [227](#), [228](#)

integer data type [7](#)

interpolation search algorithm [331](#)

example [332](#)

implementation [333](#), [334](#), [335](#), [336](#), [337](#)

working [331](#)

is not operator [17](#)

isolated vertex [284](#)

is operator [16](#)

J

jump search algorithm [320](#)

example [321](#)

implementation [322](#), [323](#), [324](#), [325](#)

Jupyter Notebook

Python development environment, setting up via [4](#), [5](#)

K

Knuth-Morris-Pratt (KMP) algorithm [406](#), [407](#)

implementing [413](#), [414](#), [415](#)

prefix function [408](#), [409](#), [410](#)

working [410](#), [411](#), [412](#), [413](#)

Kruskal's Minimum Spanning Tree [306](#), [307](#), [308](#), [309](#)

L

last in first out (LIFO) [142](#), [145](#)

last in last out (LILO) [142](#)

leaf node [180](#)

leaf vertex [282](#)

level-order tree traversal [191](#), [192](#), [193](#)

linear probing [254](#), [255](#)

linear search [314](#), [315](#)

 ordered linear search [317](#), [318](#), [319](#), [320](#)

 unordered linear search [315](#), [316](#), [317](#)

linked list [287](#)

linked list-based queues [163](#)

 dequeue operation [165](#), [166](#)

 enqueue operation [163](#), [164](#), [165](#)

linked lists [95](#)

 circular linked lists [129](#), [130](#)

 doubly linked lists [114](#)

 nodes [95](#), [96](#), [97](#), [98](#)

 pointers [95](#), [96](#), [97](#), [98](#)

 practical applications [138](#), [139](#)

 properties [95](#)

 singly linked lists [98](#)

 used, for implementing stacks [148](#), [149](#)

Linux-based operating system

Python, installing for [3](#)

lists [11](#), [12](#), [248](#)

properties [12](#), [13](#), [14](#)

log2n [222](#)

logical operators [17](#), [18](#)

loop [282](#)

M

Mac operating system

Python, installing for [3](#)

matrix [288](#)

max heap [221](#)

example [222](#)

membership operators [15](#)

merge sort [63](#), [64](#), [65](#), [66](#), [68](#)

min heap [222](#)

example [222](#)

implementing [224](#)

Minimum Spanning Tree (MST) [305](#), [306](#)

module [27](#)

N

named tuples [27](#)

negative indexing [19](#)

node [95](#), [180](#), [282](#)

nodes [95](#), [96](#), [97](#), [98](#)

not in operator [15](#)

NOT operator [18](#)

numeric types [7](#)

complex [8](#)

float [8](#)

integer [7](#)

O

objects

overview [7](#)

offset address [94](#)

omega notation [47](#), [48](#), [49](#)

open addressing [254](#)

ordered dictionary [29](#)

ordered linear search [317](#), [318](#)

implementation [319](#), [320](#)

OR operator [17](#)

outdegree [284](#)

P

parent-child relationship [179](#)

parent node [181](#)

path [282](#)

pattern matching algorithms [397](#)

Boyer-Moore algorithm [415](#)

brute force algorithm [397](#), [398](#), [399](#), [400](#)

Knuth-Morris-Pratt (KMP) algorithm [406](#), [407](#)

Rabin-Karp algorithm [401](#), [402](#)

peek operation [154](#)

pendant vertex [282](#)

perfect binary tree [183](#)

perfect hashing functions [251](#), [252](#)

performance analysis, algorithm

space complexity [40](#), [41](#)

time complexity [38](#), [39](#)

pointers [95](#), [96](#), [97](#), [98](#)

pop operation [151](#)

implementing, on stack [151](#), [152](#), [153](#)

postfix notation [194](#), [196](#)

post-order tree traversal [190](#), [191](#)

prefix notation [194](#), [195](#)

pre-order tree traversal [188](#), [189](#), [190](#)

Prim♦s Minimum Spanning Tree [309](#), [310](#), [311](#)

priority queue [221](#), [237](#)

delete operation, implementing [241](#)

demonstration [238](#)

implementation [242](#), [243](#), [244](#)
implementation, in Python [239](#)
insertion operation, implementing [240](#)
usage example [241](#)

Priority Queue (PQ) [194](#)

push operation [149](#), [150](#), [151](#)

Python [1](#), [2](#)

installing [2](#)
installing, for Linux-based operating system [3](#)
installing, for Mac operating system [3](#)
installing, for Windows operating system [2](#), [3](#)
references [1](#)

Python 3.10 [2](#)

Python development environment

setting up [3](#)
setting up, via command line [3](#), [4](#)
setting up, via Jupyter Notebook [4](#), [5](#)

Python's list-based queues [159](#)

dequeue operation [161](#), [162](#)
enqueue operation [159](#), [160](#), [161](#)

Q

quadratic probing technique

for collision resolution [264](#), [265](#), [266](#)

queues [157](#)

- applications [173](#), [174](#), [175](#), [176](#)
- linked list-based queues [163](#)
- operations [158](#), [159](#)
- Python’s list-based queues [159](#)
- stack-based queues [166](#)

quickselect algorithm [379](#)

- working [379](#), [381](#), [382](#)

quicksort algorithm [359](#), [361](#), [362](#), [363](#), [364](#)

- implementing [364](#), [365](#), [366](#), [367](#), [368](#), [369](#)
- working [378](#)

R

Rabin-Karp algorithm [401](#)

- implementing [403](#), [404](#), [405](#), [406](#)
- working [401](#), [402](#)

randomized selection algorithm [378](#)

range data type [10](#), [11](#)

recursion [59](#), [60](#)

recursive cases [59](#)

regular binary tree [182](#)

reverse Polish expression

- parsing [196](#), [197](#), [198](#), [200](#)

reverse Polish notation (RPN) [196](#)

root node [179](#), [180](#)

running time complexity, algorithm

computing [52](#), [53](#), [54](#)

S

searching algorithms [313](#), [314](#)

binary search [325](#), [326](#), [327](#), [328](#), [329](#), [330](#), [331](#)

exponential search [337](#), [338](#), [340](#), [341](#)

interpolation search [331](#), [332](#), [333](#), [334](#), [335](#), [336](#), [337](#)

jump search [320](#), [321](#), [322](#), [323](#), [324](#), [325](#)

linear search [314](#), [315](#)

selecting [341](#)

search term [316](#)

selection algorithms [377](#)

selection by sorting [378](#)

selection sort algorithm [356](#), [357](#), [358](#), [359](#)

separate chaining [272](#), [273](#), [274](#), [276](#), [277](#)

sequence data types [9](#)

lists [11](#), [12](#)

range [10](#), [11](#)

string [9](#), [10](#)

set [23](#), [24](#)

immutable sets [26](#)

operations [25](#)

Venn diagram [24](#)

shortest path problem [76](#), [77](#), [78](#), [79](#), [80](#), [81](#), [82](#), [83](#), [84](#), [85](#), [86](#), [87](#), [88](#), [89](#)

siblings [181](#)

simple binary tree [181](#)

singly linked lists [98](#)

 clearing [113](#)

 creating [98](#)

 creation, improving [99](#)

 element, searching in [107](#)

 intermediate node, deleting [111](#), [112](#), [113](#)

 items, appending [100](#)

 items, appending at intermediate positions [103](#), [104](#), [105](#), [106](#)

 items, appending to end of list [100](#), [101](#), [102](#), [103](#)

 items, deleting [108](#)

 node, deleting at end [109](#), [110](#), [111](#)

 node, deleting from beginning [108](#), [109](#)

 querying [106](#)

 size, obtaining [107](#), [108](#)

 traversal, improving [99](#)

 traversing [98](#)

sink vertex [284](#)

slicing operations [19](#)

slot [252](#)

sorting algorithms [345](#)

bubble sort algorithms [346](#), [347](#), [349](#), [350](#), [352](#)
insertion sort algorithm [352](#), [353](#), [354](#), [355](#), [356](#)
quicksort algorithm [359](#), [361](#), [362](#), [363](#), [364](#)
selection sort algorithm [356](#), [357](#), [358](#), [359](#)
Timsort algorithm [369](#), [371](#), [372](#), [373](#)

source vertex [284](#)

space complexity, algorithm [40](#), [41](#)

stack-based queues [166](#)

approaches [166](#), [167](#), [168](#), [169](#)
dequeue operation [170](#), [171](#), [172](#), [173](#)
enqueue operation [170](#)

stacks [141](#)

applications [155](#), [156](#)
example [142](#), [144](#)
implementing, with arrays [145](#), [146](#), [147](#)
implementing, with linked lists [148](#), [149](#)
operations [143](#)
peek operation [154](#)
pop operation [142](#), [143](#), [151](#), [153](#)
push operation [142](#), [143](#), [149](#), [150](#), [151](#)

string [9](#)

+ operator [10](#)
* operator [10](#)

string matching algorithms [395](#)

strings [395](#)

prefix [396](#)

suffix [396](#)

sublist [352](#)

substring [396](#)

subtree [180](#)

symbol tables [247](#), [278](#)

example [278](#)

T

theta notation [42](#), [43](#), [44](#)

time complexity, algorithm [38](#)

average-case running time [40](#)

best-case running time [40](#)

constant amount of time [38](#)

running time [38](#), [39](#)

worst-case running time [40](#)

Timsort algorithm [369](#), [371](#), [372](#), [373](#)

trees [179](#)

binary search tree (BST) [201](#), [202](#)

binary tree [181](#)

child node [181](#)

degree of node [180](#)

depth of node [181](#)

edge [181](#)

height [181](#)

leaf node [180](#)

level of root node [181](#)

node [180](#)

parent node [181](#)

root node [180](#)

siblings [181](#)

subtree [180](#)

tree traversal [186](#)

in-order tree traversal [186](#), [187](#), [188](#)

level-order traversal [191](#), [192](#), [193](#)

post-order tree traversal [190](#), [191](#)

pre-order tree traversal [188](#), [189](#), [190](#)

tuples [18](#)

operations [18](#)

U

unbalanced binary tree [184](#)

undirected graph [283](#)

unordered linear search [315](#)

implementation [316](#), [317](#)

UserDict [32](#)

UserList [32](#)

UserString [33](#)

V

vertex [282](#)

W

weighted graph [285](#)

Windows operating system

Python, installing for [2](#), [3](#)

Z

zero-based indexing [19](#)