Data Structures and Algorithms [1]

A Study Guide for Students of Sorsogon State University - Bulan Campus[2]

Jarrian Vince G. Gojar[3]

November 20, 2024

[1] A course in the Bachelor of Science in Computer Science/Information Technology/Information Systems program.

[2] This book is a study guide for students of Sorsogon State University - Bulan Campus taking up the course Data Structures and Algorithms.

[3] https://github.com/godkingjay

Sorsogon State University - Bulan Campus

Contents

Contents ii

List of Figures viii

List of Tables x

1 Introduction to Data Structures and Algorithms 2
- 1.1 Introduction . . . 2
- 1.2 Setup and Installation . . . 2
  - 1.2.1 C++ Compiler Installation . . . 2
    - 1.2.1.1 Windows . . . 2
  - 1.2.2 Visual Studio Code Installation . . . 3
  - 1.2.3 Testing the Installation . . . 3
- 1.3 What are Data Structures? . . . 4
- 1.4 What are Algorithms? . . . 4
- 1.5 Why Study Data Structures and Algorithms? . . . 4
- 1.6 Basic Terminologies . . . 4
  - 1.6.1 Data . . . 4
  - 1.6.2 Data Object . . . 4
  - 1.6.3 Data Type . . . 4
    - 1.6.3.1 Primitive Data Types . . . 5
      - 1.6.3.1.1 Integer (int) . . . 5
      - 1.6.3.1.2 Character (char) . . . 5
      - 1.6.3.1.3 Boolean (bool) . . . 5
      - 1.6.3.1.4 Floating-Point (float) . . . 5
      - 1.6.3.1.5 Double (double) . . . 6
    - 1.6.3.2 Non-primitive Data Types . . . 6
      - 1.6.3.2.1 Array (int, float, char, etc.) . . . 6
      - 1.6.3.2.2 String (char) . . . 6
      - 1.6.3.2.3 Structure . . . 6
      - 1.6.3.2.4 Class . . . 7
      - 1.6.3.2.5 Vector . . . 7
      - 1.6.3.2.6 List . . . 7
        - 1.6.3.2.6.1 Singly Linked List . . . 7
        - 1.6.3.2.6.2 Doubly Linked List . . . 8
        - 1.6.3.2.6.3 Circular Linked List . . . 8
        - 1.6.3.2.6.4 Circular Doubly Linked List . . . 8
      - 1.6.3.2.7 Stack . . . 9
      - 1.6.3.2.8 Queue . . . 9
        - 1.6.3.2.8.1 Circular Queue . . . 9

1.6.3.2.8.2 Priority Queue . . . 9
1.6.3.2.8.3 Double-Ended Queue (Deque) . . . 10
1.6.3.2.9 Tree . . . 10
1.6.3.2.10 Graph . . . 10
1.6.3.2.11 Hash Map or Hash Table . . . 11
1.6.3.2.12 Set . . . 11
1.6.4 Abstract Data Type . . . 12
1.6.5 Pointers . . . 12
1.6.5.1 Declaring Pointers . . . 13
1.6.5.2 Initializing Pointers . . . 13
1.6.5.3 Dereferencing Pointers . . . 13
1.6.5.4 Pointer Arithmetic . . . 14
1.6.5.5 Pointer to Pointer . . . 14
1.7 Asymptotic Notations . . . 15
1.7.1 Big-O Notation . . . 15
1.7.2 Omega Notation . . . 15
1.7.3 Theta Notation . . . 16
1.7.4 Complexity of an Algorithm . . . 16
1.7.4.1 Time Complexity . . . 16
1.7.4.1.1 Constant Time Complexity ($O(1)$) . . . 16
1.7.4.1.2 Logarithmic Time Complexity ($O(\log n)$) . . . 17
1.7.4.1.3 Linear Time Complexity ($O(n)$) . . . 17
1.7.4.1.4 Linearithmic Time Complexity ($O(n \log n)$) . . . 17
1.7.4.1.5 Quadratic Time Complexity ($O(n^2)$) . . . 18
1.7.4.1.6 Exponential Time Complexity ($O(2^n)$) . . . 19
1.7.4.1.7 Factorial Time Complexity ($O(n!)$) . . . 19
1.7.4.2 Space Complexity . . . 19
1.7.4.2.1 Constant Space Complexity ($O(1)$) . . . 19
1.7.4.2.2 Linear Space Complexity ($O(n)$) . . . 20
1.7.4.2.3 Quadratic Space Complexity ($O(n^2)$) . . . 20
1.7.4.2.4 Exponential Space Complexity ($O(2^n)$) . . . 21
1.7.4.2.5 Factorial Space Complexity ($O(n!)$) . . . 21
1.8 Summary . . . 21
1.9 Coding Exercises . . . 22

2 Arrays and Linked Lists 23
2.1 Introduction . . . 23
2.2 Arrays . . . 23
2.2.1 Types of Arrays . . . 25
2.2.1.1 One-dimensional Array . . . 25
2.2.1.2 Multi-dimensional Array . . . 26
2.2.2 Array Operations . . . 27
2.2.2.1 Insertion . . . 27
2.2.2.2 Deletion . . . 28
2.2.2.3 Searching . . . 29
2.3 Linked Lists . . . 30
2.3.1 Types of Linked Lists . . . 30
2.3.1.1 Singly Linked List . . . 30
2.3.1.2 Doubly Linked List . . . 31
2.3.1.3 Circular Linked List . . . 32
2.3.1.4 Doubly Circular Linked List . . . 34

2.3.2 Operations on Linked Lists . . . 35
2.3.2.1 Insertion . . . 35
2.3.2.2 Deletion . . . 36
2.3.2.3 Searching . . . 38
2.4 Comparison of Arrays and Linked Lists . . . 39
2.5 Summary . . . 39
2.6 Coding Exercises . . . 39

3 Stacks and Queues 42
3.1 Introduction . . . 42
3.2 Stack . . . 42
3.2.1 Operations on Stack . . . 43
3.2.1.1 Push . . . 43
3.2.1.2 Pop . . . 44
3.2.1.3 Top . . . 45
3.2.1.4 Size . . . 45
3.2.1.5 Empty . . . 46
3.3 Queue . . . 47
3.3.1 Operations on Queue . . . 48
3.3.1.1 Push . . . 48
3.3.1.2 Pop . . . 49
3.3.1.3 Front . . . 50
3.3.1.4 Back . . . 51
3.3.1.5 Size . . . 51
3.3.1.6 Empty . . . 52
3.4 Comparison of Stack and Queue . . . 53
3.5 Summary . . . 53

4 Introduction to Algorithms 54
4.1 Introduction . . . 54
4.2 How do Algorithms Work? . . . 54
4.3 Characteristics of an Algorithm . . . 55
4.4 What is the Need for Algorithms? . . . 55
4.5 Types of Algorithms . . . 55
4.6 Examples of Algorithms . . . 56
4.7 How to Write an Algorithm? . . . 56
4.8 How to Design an Algorithm? . . . 56
4.8.1 Example . . . 57
4.8.1.1 Step 1: Fulfilling the pre-requisites . . . 57
4.8.1.2 Step 2: Designing the Algorithm (Pseudo-code) . . . 57
4.8.1.3 Step 3: Testing the Algorithm . . . 57
4.9 Summary . . . 58

5 Searching Algorithms 59
5.1 Introduction . . . 59
5.2 Importance of Searching Algorithms . . . 59
5.3 Characteristics of Searching Algorithms . . . 59
5.4 Types of Searching Algorithms . . . 60
5.4.1 Linear Search . . . 60
5.4.1.1 Algorithm of Linear Search . . . 61
5.4.1.2 Pseudo-code of Linear Search . . . 61
5.4.1.3 Linear Search in C++ . . . 61

5.4.1.4 Application of Linear Search . . . 62
5.4.1.5 Advantages and Disadvantages of Linear Search . . . 62
5.4.1.6 Complexity Analysis of Linear Search . . . 62
5.4.2 Binary Search . . . 62
5.4.2.1 Algorithm of Binary Search . . . 63
5.4.2.2 Pseudo-code of Binary Search . . . 63
5.4.2.3 Binary Search in C++ . . . 64
5.4.2.4 Application of Binary Search . . . 65
5.4.2.5 Advantages and Disadvantages of Binary Search . . . 65
5.4.2.6 Complexity Analysis of Binary Search . . . 65
5.4.3 Other Searching Algorithms . . . 66
5.5 Summary . . . 66

6 Sorting Algorithms 67
6.1 Introduction . . . 67
6.2 Importance of Sorting Algorithms . . . 67
6.3 Types of Sorting Techniques . . . 68
6.4 Types of Comparison-Based Sorting Algorithms . . . 68
6.4.1 Selection Sort . . . 69
6.4.1.1 Algorithm of Selection Sort . . . 70
6.4.1.2 Pseudo-code of Selection Sort . . . 70
6.4.1.3 Selection Sort in C++ . . . 70
6.4.1.4 Advantages and Disadvantages of Selection Sort . . . 71
6.4.1.5 Complexity Analysis of Selection Sort . . . 72
6.4.2 Bubble Sort . . . 72
6.4.2.1 Algorithm of Bubble Sort . . . 73
6.4.2.2 Pseudo-code of Bubble Sort . . . 73
6.4.2.3 Bubble Sort in C++ . . . 73
6.4.2.4 Advantages and Disadvantages of Bubble Sort . . . 74
6.4.2.5 Complexity Analysis of Bubble Sort . . . 75
6.4.3 Insertion Sort . . . 75
6.4.3.1 Algorithm of Insertion Sort . . . 76
6.4.3.2 Pseudo-code of Insertion Sort . . . 76
6.4.3.3 Insertion Sort in C++ . . . 76
6.4.3.4 Applications of Insertion Sort . . . 77
6.4.3.5 Advantages and Disadvantages of Insertion Sort . . . 77
6.4.3.6 Complexity Analysis of Insertion Sort . . . 78
6.4.4 Merge Sort . . . 78
6.4.4.1 Algorithm of Merge Sort . . . 78
6.4.4.2 Pseudo-code of Merge Sort . . . 78
6.4.4.3 Merge Sort in C++ . . . 79
6.4.4.4 Applications of Merge Sort . . . 81
6.4.4.5 Advantages and Disadvantages of Merge Sort . . . 82
6.4.4.6 Complexity Analysis of Merge Sort . . . 82
6.4.5 Quick Sort . . . 82
6.4.5.1 Algorithm of Quick Sort . . . 83
6.4.5.2 Pseudo-code of Quick Sort . . . 83
6.4.5.3 Quick Sort in C++ . . . 84
6.4.5.4 Applications of Quick Sort . . . 85
6.4.5.5 Advantages and Disadvantages of Quick Sort . . . 85
6.4.5.6 Complexity Analysis of Quick Sort . . . 86

6.4.6 Heap Sort . . . 86
6.5 Types of Non-comparison-based Sorting Algorithms . . . 86
6.5.1 Counting Sort . . . 86
6.5.2 Radix Sort . . . 86
6.6 Summary . . . 86

7 Hashing **87**
7.1 Introduction . . . 87
7.2 Hash Table . . . 87
7.3 Hash Function . . . 87
7.4 Collision Resolution Techniques . . . 87
7.4.1 Separate Chaining . . . 87
7.4.2 Open Addressing . . . 87
7.4.2.1 Linear Probing . . . 87
7.4.2.2 Quadratic Probing . . . 87
7.4.2.3 Double Hashing . . . 87
7.5 Complexity Analysis of Hashing . . . 87
7.6 Summary . . . 87

8 Advanced Data Structures and Algorithms **88**
8.1 Introduction . . . 89
8.2 Advanced Data Structures . . . 89
8.2.1 Segment Tree . . . 89
8.2.2 Fenwick Tree . . . 89
8.2.3 Suffix Tree . . . 89
8.2.4 Suffix Array . . . 89
8.2.5 Trie . . . 89
8.2.6 Heap . . . 89
8.2.7 Disjoint Set . . . 89
8.2.8 Skip List . . . 89
8.2.9 Splay Tree . . . 89
8.2.10 Bloom Filter . . . 89
8.2.11 KD Tree . . . 89
8.2.12 Quad Tree . . . 89
8.2.13 Octree . . . 89
8.2.14 B-Tree . . . 89
8.2.15 B+ Tree . . . 89
8.2.16 R-Tree . . . 89
8.2.17 X-Tree . . . 89
8.2.18 Y-Tree . . . 89
8.2.19 Z-Tree . . . 89
8.3 Advanced Algorithms . . . 89
8.3.1 Dynamic Programming . . . 89
8.3.2 Greedy Algorithms . . . 89
8.3.3 Backtracking . . . 89
8.3.4 Divide and Conquer . . . 89
8.3.5 Branch and Bound . . . 89
8.3.6 Randomized Algorithms . . . 89
8.3.7 Approximation Algorithms . . . 89
8.3.8 String Matching Algorithms . . . 89
8.3.9 Pattern Searching Algorithms . . . 89
8.3.10 Cryptography Algorithms . . . 89

8.3.11 Geometric Algorithms . . . 89
8.3.12 Graph Algorithms . . . 89
8.3.13 Network Flow Algorithms . . . 89
8.3.14 Game Theory Algorithms . . . 89
8.3.15 Quantum Algorithms . . . 89
8.4 Summary . . . 89

9 Applications of Data Structures and Algorithms 90
9.1 Applications in Computer Science . . . 91
9.1.1 Operating Systems . . . 91
9.1.2 Database Management Systems . . . 91
9.1.3 Compiler Design . . . 91
9.1.4 Networking . . . 91
9.1.5 Artificial Intelligence . . . 91
9.1.6 Machine Learning . . . 91
9.1.7 Computer Graphics . . . 91
9.1.8 Computer Vision . . . 91
9.1.9 Robotics . . . 91
9.1.10 Web Development . . . 91
9.1.11 Mobile Development . . . 91
9.1.12 Game Development . . . 91
9.1.13 Cybersecurity . . . 91
9.1.14 Quantum Computing . . . 91
9.2 Applications in Real Life . . . 91
9.2.1 Social Media . . . 91
9.2.2 E-commerce . . . 91
9.2.3 Healthcare . . . 91
9.2.4 Finance . . . 91
9.2.5 Transportation . . . 91
9.2.6 Education . . . 91
9.2.7 Agriculture . . . 91
9.2.8 Manufacturing . . . 91
9.2.9 Entertainment . . . 91
9.2.10 Sports . . . 91
9.2.11 Travel . . . 91
9.2.12 Telecommunications . . . 91
9.2.13 Energy . . . 91
9.2.14 Environment . . . 91
9.2.15 Politics . . . 91
9.2.16 Military . . . 91
9.3 Summary . . . 91

10 References 92

List of Figures

1 Pointer Example . . . 12
2 Dereferencing Pointers . . . 13
3 Pointer Arithmetic . . . 14
4 Pointer to Pointer . . . 15
5 Asymptotic Notation . . . 16

6 Elements of an array in C++ . . . 23
7 Initializing Array Elements . . . 24
8 Initializing Array Elements with Empty Members . . . 25
9 One-dimensional Array in C++ . . . 26
10 Two-dimensional Array in C++ . . . 26
11 Array Insertion . . . 28
12 Array Deletion . . . 29
13 Array Searching . . . 29
14 Singly Linked List . . . 30
15 Doubly Linked List . . . 31
16 Circular Linked List . . . 33
17 Doubly Circular Linked List . . . 34
18 Linked List Insertion . . . 35
19 Linked List Deletion . . . 37
20 Linked List Searching . . . 38

21 Stack Data Structure . . . 42
22 Stack Push Operation . . . 43
23 Stack Pop Operation . . . 44
24 Stack Top Operation . . . 45
25 Stack Size Operation . . . 46
26 Stack Empty Operation . . . 47
27 Queue Data Structure . . . 47
28 Queue Push Operation . . . 48
29 Queue Pop Operation . . . 49
30 Queue Front Operation . . . 50
31 Queue Back Operation . . . 51
32 Queue Size Operation . . . 52
33 Queue Empty Operation . . . 52

34 Flowchart for an Algorithm . . . 54

35 Linear Search . . . 60
36 Binary Search . . . 63

37 Sorting Techniques 68
38 Selection Sort 69
39 Bubble Sort 73
40 Insertion Sort 75
41 Merge Sort 78
42 Quick Sort 83

List of Tables

List of Codes

1.1 Hello World Program . . . 3
1.2 Compiling the Program . . . 3
1.3 Running the Program . . . 3
1.4 Integer Data Type . . . 5
1.5 Character Data Type . . . 5
1.6 Boolean Data Type . . . 5
1.7 Floating-Point Data Type . . . 5
1.8 Double Data Type . . . 6
1.9 Array Data Type . . . 6
1.10 String Data Type . . . 6
1.11 Structure Data Type . . . 6
1.12 Class Data Type . . . 7
1.13 Vector Data Type . . . 7
1.14 List Data Type . . . 7
1.15 Singly Linked List Data Type . . . 8
1.16 Doubly Linked List Data Type . . . 8
1.17 Circular Linked List Data Type . . . 8
1.18 Circular Doubly Linked List Data Type . . . 8
1.19 Stack Data Type . . . 9
1.20 Queue Data Type . . . 9
1.21 Priority Queue Data Type . . . 10
1.22 Deque Data Type . . . 10
1.23 Tree Data Type . . . 10
1.24 Graph Data Type . . . 11
1.25 Ordered Map Data Type . . . 11
1.26 Unordered Map Data Type . . . 11
1.27 Ordered Set Data Type . . . 11
1.28 Unordered Set Data Type . . . 12
1.29 Pointer Data Type . . . 12
1.30 Declaring Pointers . . . 13
1.31 Initializing Pointers . . . 13
1.32 Dereferencing Pointers . . . 13
1.33 Pointer Arithmetic . . . 14
1.34 Pointer to Pointer Data Type . . . 15
1.35 Constant Time Complexity . . . 16
1.36 Logarithmic Time Complexity . . . 17
1.37 Linear Time Complexity . . . 17
1.38 Linearithmic Time Complexity . . . 17
1.39 Quadratic Time Complexity . . . 18
1.40 Exponential Time Complexity . . . 19
1.41 Factorial Time Complexity . . . 19

1.42 Constant Space Complexity . . . 19
1.43 Linear Space Complexity . . . 20
1.44 Quadratic Space Complexity . . . 20
1.45 Exponential Space Complexity . . . 21
1.46 Factorial Space Complexity . . . 21
2.1 Array Declaration . . . 23
2.2 Assigning Values to Array Elements . . . 24
2.3 Initializing Array Elements . . . 24
2.4 Initializing Array Elements with Unspecified Size . . . 24
2.5 Initializing Array Elements with Empty Members . . . 25
2.6 One-dimensional Array . . . 26
2.7 Two-dimensional Array . . . 26
2.8 Two-dimensional Array with Empty Members . . . 27
2.9 Array Insertion . . . 27
2.10 Array Deletion . . . 28
2.11 Array Searching . . . 29
2.12 Singly Linked List . . . 30
2.13 Doubly Linked List . . . 31
2.14 Circular Linked List . . . 33
2.15 Doubly Circular Linked List . . . 34
2.16 Linked List Insertion . . . 35
2.17 Linked List Deletion . . . 36
2.18 Linked List Searching . . . 38
3.1 Stack Push Operation . . . 43
3.2 Stack Pop Operation . . . 44
3.3 Stack Top Operation . . . 45
3.4 Stack Size Operation . . . 46
3.5 Stack Empty Operation . . . 46
3.6 Queue Push Operation . . . 48
3.7 Queue Pop Operation . . . 49
3.8 Queue Front Operation . . . 50
3.9 Queue Back Operation . . . 51
3.10 Queue Size Operation . . . 51
3.11 Queue Empty Operation . . . 52
4.1 Adding Three Numbers . . . 57
5.1 Linear Search Pseudo-code . . . 61
5.2 Linear Search in C++ . . . 61
5.3 Binary Search Pseudo-code . . . 63
5.4 Binary Search in C++ . . . 64
6.1 Selection Sort Pseudo-code . . . 70
6.2 Selection Sort in C++ . . . 70
6.3 Bubble Sort Pseudo-code . . . 73
6.4 Bubble Sort in C++ . . . 74
6.5 Insertion Sort Pseudo-code . . . 76
6.6 Insertion Sort in C++ . . . 76
6.7 Merge Sort Pseudo-code . . . 79
6.8 Merge Sort in C++ . . . 80
6.9 Quick Sort Pseudo-code . . . 83
6.10 Quick Sort in C++ . . . 84

Preface

> *"Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*
>
> – Linus Torvalds

Jarrian Vince G. Gojar
https://github.com/godkingjay

1

Introduction to Data Structures and Algorithms

1.1 Introduction

Data structures and algorithms are one of the fundamental components of computer science. They are essential for solving complex problems efficiently and effectively. Data structures are used to store and organize data in a computer so that it can be accessed and manipulated efficiently. Algorithms are step-by-step procedures or formulas for solving a problem. They are the instructions that tell a computer how to perform a task.

In this course, we will learn about the fundamental data structures and algorithms that are used in computers. We will study how to design, implement, and analyze data structures and algorithms to solve real-world problems. By the end of this course, you will have a solid foundation in data structures and algorithms that will help you become a better programmer and problem solver.

1.2 Setup and Installation

In this course, we will be using the C++ programming language to implement data structures and algorithms. C++ is a powerful and versatile programming language that is widely used in the field of computer science. To get started, you will need to install a C++ compiler and an integrated development environment (IDE) on your computer.

1.2.1 C++ Compiler Installation

The first step is to install a C++ compiler on your computer. A compiler is a program that translates source code written in a programming language into machine code that can be executed by a computer. There are several C++ compilers available, but we recommend using the GNU Compiler Collection (GCC) which is a free and open-source compiler that supports multiple programming languages including C++.

1.2.1.1 Windows

To install GCC on Windows, you can use the MinGW (Minimalist GNU for Windows) project which provides a port of GCC to Windows. You can download the MinGW installer from the MinGW website and follow the installation instructions. You can install MinGW

by following the instructions here: https://code.visualstudio.com/docs/languages/cpp#_example-install-mingwx64-on-windows

1.2.2 Visual Studio Code Installation

The next step is to install an integrated development environment (IDE) on your computer. An IDE is a software application that provides comprehensive facilities to computer programmers for software development. We recommend using Visual Studio Code which is a free and open-source IDE developed by Microsoft. You can download Visual Studio Code from the official website and follow the installation instructions: https://code.visualstudio.com/Download

Other than Visual Studio Code, you also need to install the C/C++ extension for Visual Studio Code. You can install the C/C++ extension by following the instructions here: https://code.visualstudio.com/docs/languages/cpp

1.2.3 Testing the Installation

To test if the installation was successful, you can create a simple C++ program and compile it using the C++ compiler. Open Visual Studio Code and create a new file with the following C++ code:

```cpp
#include <iostream>
namespace std;

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

Code 1.1: Hello World Program

Save the file with a .cpp extension (e.g., hello.cpp) and open a terminal window in Visual Studio Code. Compile the program using the following command:

```
g++ hello.cpp -o hello
```

Code 1.2: Compiling the Program

If there are no errors, you can run the program by executing the following command:

```
./hello
```

Code 1.3: Running the Program

If everything is set up correctly, you should see the output "Hello, World!" printed on the screen.

1.3 What are Data Structures?

A ***data structure*** is a way of organizing and storing data in a computer so that it can be accessed and manipulated efficiently. Data structures provide a way to manage large amounts of data effectively for various applications. They define the relationship between the data, and the operations that can be performed on the data. There are many different types of data structures that are used in computer science, each with its own strengths and weaknesses. The use of the right data structure can significantly improve the performance of an algorithm and make it more efficient.

1.4 What are Algorithms?

An ***algorithm*** is a step-by-step procedure or formula for solving a problem. It is a sequence of well-defined instructions that take some input and produce an output. Algorithms are used to solve complex problems and perform various tasks efficiently. They are the instructions that tell a computer how to perform a task. Algorithms are essential for writing computer programs and developing software applications. The efficiency of an algorithm is measured by its time complexity and space complexity.

1.5 Why Study Data Structures and Algorithms?

Data structures and algorithms are essential topics in computer science and software engineering. They are one of the fundamental components of computer science and are used in various applications such as operating systems, database management systems, networking, artificial intelligence, and many others. A good understanding of data structures and algorithms will help you become a better programmer and problem solver. In addition, many companies use data structures and algorithms as part of their technical interviews to assess the problem-solving skills of candidates. Therefore, studying data structures and algorithms is essential for anyone pursuing a career in software engineering or software development.

1.6 Basic Terminologies

Before we dive into the details of data structures and algorithms, let's understand some basic terminologies that might be helpful in understanding the concepts better.

1.6.1 Data

Data is a collection of facts, figures, or information that can be used for analysis or reference. It can be in the form of numbers, text, images, audio, video, or any other format. Data is the raw material that is processed by a computer to produce meaningful information.

1.6.2 Data Object

A ***data object*** is an instance of a data structure that contains data along with the operations that can be performed on the data. It is an abstraction of a real-world entity that is represented in a computer program.

1.6.3 Data Type

A ***data type*** is a classification of data that tells the compiler or interpreter how the programmer intends to use the data. It defines the operations that can be performed on the data, the values that can be stored in the data, and the memory space required to store the data.

1.6.3.1 Primitive Data Types

Primitive data types are the basic data types that are built into the programming language. They are used to store simple values such as integers, floating-point numbers, characters, and booleans. Examples of primitive data types include int, float, char, and bool. The following are the common primitive data types used in programming:

1.6.3.1.1 Integer (int)

The ***integer*** data type is used to store whole numbers without any decimal points. It can be either signed or unsigned, depending on whether it can store negative values or not. An integer's value can range from -2,147,483,648 to 2,147,483,647 and takes 4 bytes of memory.

```
int x = 10;
```

Code 1.4: Integer Data Type

1.6.3.1.2 Character (char)

The ***character*** data type is used to store a single character such as a letter, digit, or special symbol. It is represented by a single byte of memory. A char value can range from -128 to 127 or 0 to 255, depending on whether it is signed or unsigned. These values are represented using ASCII codes.

```
char c = 'A';
```

Code 1.5: Character Data Type

1.6.3.1.3 Boolean (bool)

The ***boolean*** data type is used to store true or false values. It is represented by a single byte of memory. A bool value can be either true or false.

```
bool flag = true;
```

Code 1.6: Boolean Data Type

1.6.3.1.4 Floating-Point (float)

The ***floating-point*** data type is used to store real numbers with decimal points. It can represent both integer and fractional parts of a number. It can be either single precision or double precision, depending on the number of bits used to store the value. A float value can range from 1.2E-38 to 3.4E+38 and takes 4 bytes of memory.

```
float y = 3.14;
```

Code 1.7: Floating-Point Data Type

1.6.3.1.5 Double (double)

The ***double*** data type is used to store real numbers with double precision. It can represent both integer and fractional parts of a number with higher precision than the float data type. A double value can range from 2.3E-308 to 1.7E+308 and takes 8 bytes of memory.

```
double z = 3.14159;
```

Code 1.8: Double Data Type

1.6.3.2 Non-primitive Data Types

Non-primitive data types are more complex data types that are derived from primitive data types. They are used to store collections of values or objects. Examples of non-primitive data types include arrays, strings, structures, classes, and pointers.

1.6.3.2.1 Array (int, float, char, etc.)

An ***array*** is a collection of elements of the same data type that are stored in contiguous memory locations. It is used to store multiple values of the same type under a single name. The elements of an array can be accessed using an index value. In C++, arrays are zero-indexed, which means the first element is at index 0. Arrays also have a fixed size that is specified at the time of declaration. If you need a dynamic size array, you can use a vector in C++.

```
int arr[5] = {1, 2, 3, 4, 5};
```

Code 1.9: Array Data Type

1.6.3.2.2 String (char)

A ***string*** is a collection of characters that are stored as a sequence of characters terminated by a null character '\0'. It is used to represent text in a computer program. Strings are treated as arrays of characters in C++.

```
char str[] = "Hello, World!";
```

Code 1.10: String Data Type

1.6.3.2.3 Structure

A ***structure*** is a user-defined data type that is used to store a collection of different data types under a single name. It is used to represent a record that contains multiple fields or members. Each field in a structure can have a different data type.

```
struct Person {
    char name[50];
    int age;
    float height;
};
```

Code 1.11: Structure Data Type

1.6.3.2.4 Class

A ***class*** is a user-defined data type that is used to define objects that contain data members and member functions. It is used to implement object-oriented programming concepts such as encapsulation, inheritance, and polymorphism.

```
class Circle {
    private:
        float radius;
    public:
        float getArea() {
            return 3.14 * radius * radius;
        }
};
```

Code 1.12: Class Data Type

1.6.3.2.5 Vector

A ***vector*** is a dynamic array that can grow or shrink in size dynamically. It is a part of the Standard Template Library (STL) in C++ and provides a more flexible alternative to fixed-size arrays. Vectors are used to store a collection of elements of the same data type.

```
vector<int> vec = {1, 2, 3, 4, 5};
```

Code 1.13: Vector Data Type

1.6.3.2.6 List

A ***list*** is a linear data structure that is used to store a collection of elements in a sequential order. It is a part of the Standard Template Library (STL) in C++ and provides operations to insert, delete, and access elements in the list. Lists are used to implement linked lists in C++.

```
list<int> lst = {1, 2, 3, 4, 5};
```

Code 1.14: List Data Type

There are different types of lists in C++, such as singly linked list, doubly linked list, circular linked list, and circular doubly linked list, that provide different operations and performance characteristics.

1.6.3.2.6.1 Singly Linked List

A ***singly linked list*** is a linear data structure that is used to store a collection of elements in a sequential order. Each element in the list is stored in a node that contains the data and a

pointer to the next node in the list.

```
struct Node {
    int data;
    Node *next;
};
```

Code 1.15: Singly Linked List Data Type

1.6.3.2.6.2 Doubly Linked List

A ***doubly linked list*** is a linear data structure that is used to store a collection of elements in a sequential order. Each element in the list is stored in a node that contains the data, a pointer to the next node, and a pointer to the previous node in the list.

```
struct Node {
    int data;
    Node *next;
    Node *prev;
};
```

Code 1.16: Doubly Linked List Data Type

1.6.3.2.6.3 Circular Linked List

A ***circular linked list*** is a linear data structure that is used to store a collection of elements in a circular order. Each element in the list is stored in a node that contains the data and a pointer to the next node in the list. The last node in the list points back to the first node, creating a circular structure.

```
struct Node {
    int data;
    Node *next;
};
```

Code 1.17: Circular Linked List Data Type

1.6.3.2.6.4 Circular Doubly Linked List

A ***circular doubly linked list*** is a linear data structure that is used to store a collection of elements in a circular order. Each element in the list is stored in a node that contains the data, a pointer to the next node, and a pointer to the previous node in the list. The last node in the list points back to the first node, creating a circular structure.

```
struct Node {
    int data;
    Node *next;
    Node *prev;
```

```
};
```

Code 1.18: Circular Doubly Linked List Data Type

1.6.3.2.7 Stack

A ***stack*** is a linear data structure that follows the Last In First Out (LIFO) principle. It is used to store a collection of elements in a sequential order. The main operations on a stack are push (to insert an element) and pop (to remove an element).

```
stack<int> stk;
stk.push(1);
stk.push(2);
stk.push(3);
stk.push(4);
stk.pop();
```

Code 1.19: Stack Data Type

1.6.3.2.8 Queue

A ***queue*** is a linear data structure that follows the First In First Out (FIFO) principle. It is used to store a collection of elements in a sequential order. The main operations on a queue are enqueue (to insert an element) and dequeue (to remove an element).

```
queue<int> que;
que.push(1);
que.push(2);
que.push(3);
que.push(4);
que.pop();
```

Code 1.20: Queue Data Type

There are different types of queues in C++, such as linear queue, circular queue, priority queue, and double-ended queue (deque), that provide different operations and performance characteristics.

1.6.3.2.8.1 Circular Queue

A ***circular queue*** is a type of queue that uses a circular structure to store elements. Unlike a linear queue, a circular queue does not have a fixed front and rear end. Instead, the front and rear ends wrap around the ends of the queue. This allows the queue to reuse the space freed up by dequeued elements. In C++, there is no built-in circular queue data type, but you can implement one using an array and a few pointers.

1.6.3.2.8.2 Priority Queue

A ***priority queue*** is a type of queue that stores elements based on their priority. The element with the highest priority is dequeued first. Priority queues are typically implemented using

heaps, which are a type of binary tree data structure.

```
priority_queue<int> pq;
pq.push(1);
pq.push(4);
pq.push(2);
pq.push(3);
pq.pop();
```

Code 1.21: Priority Queue Data Type

1.6.3.2.8.3 Double-Ended Queue (Deque)

A ***double-ended queue*** or ***deque*** is a type of queue that allows elements to be inserted and removed from both ends. It is a generalization of both stacks and queues and provides more flexibility in manipulating elements.

```
deque<int> dq;
dq.push_front(1);
dq.push_back(2);
dq.push_front(3);
dq.push_back(4);
dq.pop_front();
```

Code 1.22: Deque Data Type

1.6.3.2.9 Tree

A ***tree*** is a non-linear data structure that is used to store a collection of elements in a hierarchical order. It consists of nodes that are connected by edges. The topmost node in a tree is called the root node, and the nodes below it are called child nodes. Trees are used to represent hierarchical relationships between elements.

```
struct Node {
    int data;
    Node *left;
    Node *right;
};
```

Code 1.23: Tree Data Type

There are different types of trees in computer science, such as binary trees, binary search trees, AVL trees, red-black trees, and many others, that provide different operations and performance characteristics.

1.6.3.2.10 Graph

A ***graph*** is a non-linear data structure that is used to store a collection of elements and the relationships between them. It consists of nodes (vertices) that are connected by edges.

Graphs are used to represent networks, social relationships, maps, and many other real-world applications.

```
struct Graph {
    int V;
    list<int> *adj;
};
```

Code 1.24: Graph Data Type

There are different types of graphs in computer science, such as directed graphs, undirected graphs, weighted graphs, and many others, each with its own set of advantages and disadvantages.

Another important thing to note is that "Every tree is a graph, but not every graph is a tree."

1.6.3.2.11 Hash Map or Hash Table

A ***Hash Map*** is a data structure that is used to store a collection of key-value pairs. It uses a hash function to map keys to values and stores them in an array. Hash maps provide fast access to elements and are used to implement associative arrays, sets, and dictionaries.

There are two ways to implement a hash map in C++: the ordered map using the map class or using the unordered_map class.

```
map<string, int> mp;
mp["one"] = 1;
mp["two"] = 2;
mp["three"] = 3;
```

Code 1.25: Ordered Map Data Type

```
unordered_map<string, int> ump;
ump["one"] = 1;
ump["two"] = 2;
ump["three"] = 3;
```

Code 1.26: Unordered Map Data Type

1.6.3.2.12 Set

A ***set*** is a data structure that is used to store a collection of unique elements. It is used to implement the mathematical set abstraction and provides operations to insert, delete, and search for elements.

There are two ways to implement a set in C++: the ordered set using the set class or using the unordered_set class.

```
set<int> st;
st.insert(1);
```

```
st.insert(2);
st.insert(3);
```

Code 1.27: Ordered Set Data Type

```
unordered_set<int> ust;
ust.insert(1);
ust.insert(2);
ust.insert(3);
```

Code 1.28: Unordered Set Data Type

1.6.4 Abstract Data Type

An ***abstract data type (ADT)*** is a mathematical model that defines a set of data values and operations that can be performed on those values. It is an abstraction of a data structure that specifies the operations that can be performed on the data without specifying how they are implemented. *Abstraction* refers to the process of hiding the implementation details of a data structure and exposing only the essential features. An ADT is defined by its interface, which includes the data values and operations that can be performed on those values.

1.6.5 Pointers

A ***pointer*** is a special type of data type that stores the memory address of another data type. An "address" is a unique number that identifies a location in memory. The memory address of a variable is the location in memory where the variable is stored. Pointers are used to store the address of a variable or object in memory. Pointers are used to implement dynamic memory allocation and to pass parameters by reference.

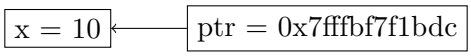


Figure 1: Pointer Example

Figure 1 shows an example of a pointer in C++. The variable `x` stores the value 10, and the pointer `ptr` stores the memory address of the variable `x`. The memory address is represented as a hexadecimal number `0x7fffbf7f1bdc`.

```
int main() {
    int x = 10;
    int *ptr = &x;

    cout << *ptr; // Output: 10

    return 0;
}
```

Code 1.29: Pointer Data Type

In the above example, the pointer `ptr` stores the memory address of the variable `x`. The `*` operator is used to dereference the pointer and access the value stored at the memory address.

An example of an address of a variable is `0x7fffbf7f1bdc`. When you dereference the pointer, you get the value stored at that address.

1.6.5.1 Declaring Pointers

To declare a pointer, you need to specify the data type of the variable or object it points to. You can declare a pointer using the following syntax:

```
int *ptr;
```

Code 1.30: Declaring Pointers

In the above example, the pointer `ptr` is declared to point to an integer variable. You can also declare a pointer to a structure, class, or any other data type.

1.6.5.2 Initializing Pointers

To initialize a pointer, you need to assign it the memory address of a variable or object. You can initialize a pointer using the address-of operator `&`. You can also initialize a pointer to `NULL` or `nullptr` to indicate that it does not point to any memory location.

```
int x = 10;
int *ptr = &x;
```

Code 1.31: Initializing Pointers

In the above example, the pointer `ptr` is initialized with the memory address of the variable `x`.

1.6.5.3 Dereferencing Pointers

To access the value stored at the memory address pointed to by a pointer, you need to dereference the pointer using the dereference operator `*`. The dereference operator is used to access the value stored at the memory address.

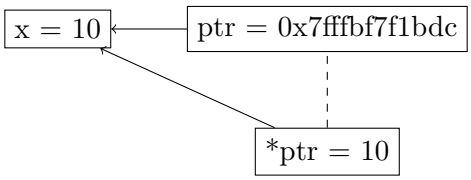


Figure 2: Dereferencing Pointers

Figure 2 shows an example of dereferencing a pointer in C++. The pointer `ptr` points to the variable `x`, and the dereference operator `*` is used to access the value stored at the memory address.

```
int x = 10;
int *ptr = &x;

cout << *ptr; // Output: 10

```

```
*ptr = 20;

cout << x; // Output: 20
```

Code 1.32: Dereferencing Pointers

In the above example, the pointer `ptr` points to the variable `x`. You can use the dereference operator `*` to access the value stored at the memory address. You can also use the dereference operator to modify the value stored at the memory address.

1.6.5.4 Pointer Arithmetic

Pointer arithmetic is a feature of pointers that allows you to perform arithmetic operations on pointers. You can add or subtract an integer value from a pointer to move it to a different memory location. Pointer arithmetic is used to access elements in an array or to iterate over a data structure.

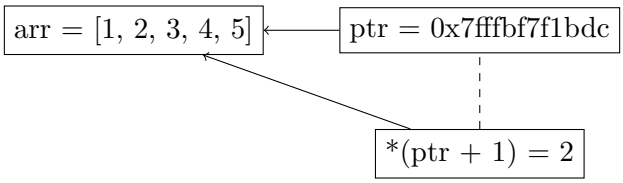


Figure 3: Pointer Arithmetic

Figure 3 shows an example of pointer arithmetic in C++. The pointer `ptr` points to the first element of the array `arr`, and the pointer arithmetic operation `*(ptr + 1)` is used to access the second element of the array.

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int *ptr = arr;

    cout << *ptr; // Output: 1
    cout << *(ptr + 1); // Output: 2

    return 0;
}
```

Code 1.33: Pointer Arithmetic

In the above example, the pointer `ptr` points to the first element of the array `arr`. You can use pointer arithmetic to access the elements of the array by adding an integer value to the pointer.

1.6.5.5 Pointer to Pointer

A ***pointer to pointer*** is a special type of pointer that stores the memory address of another pointer. It is used to store the address of a pointer variable in memory. Pointer to pointer is used to implement multiple levels of indirection and to create dynamic data structures such as linked lists and trees.

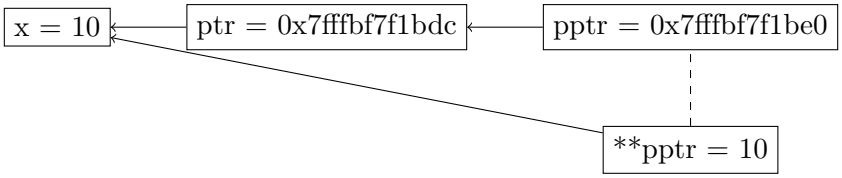


Figure 4: Pointer to Pointer

Figure 4 shows an example of a pointer to pointer in C++. The pointer `ptr` stores the memory address of the variable `x`, and the pointer `pptr` stores the memory address of the pointer `ptr`. The double dereference operator `**` is used to access the value stored at the memory address.

```
int main() {
    int x = 10;
    int *ptr = &x;
    int **pptr = &ptr;

    cout << **pptr; // Output: 10

    return 0;
}
```

Code 1.34: Pointer to Pointer Data Type

In the above example, the pointer `ptr` stores the memory address of the variable `x`, and the pointer `pptr` stores the memory address of the pointer `ptr`. You can use the double dereference operator `**` to access the value stored at the memory address.

1.7 Asymptotic Notations

Asymptotic notations are mathematical notations used to describe the limiting behavior of a function as the input size approaches infinity. They are used to analyze the complexity of algorithms and to compare the performance of different algorithms. The three most common asymptotic notations used in computer science are big-O notation, omega notation, and theta notation.

1.7.1 Big-O Notation

The ***big-O notation*** is used to describe the upper bound on the growth rate of an algorithm as the input size approaches infinity. It provides an upper limit on the worst-case time complexity of an algorithm. The big-O notation is used to analyze the efficiency of an algorithm in terms of the number of basic operations it performs.

1.7.2 Omega Notation

The ***omega notation*** or ***big-omega notation*** is used to describe the lower bound on the growth rate of an algorithm as the input size approaches infinity. It provides a lower limit on the best-case time complexity of an algorithm. The omega notation is used to analyze the efficiency of an algorithm in terms of the minimum number of basic operations it performs.

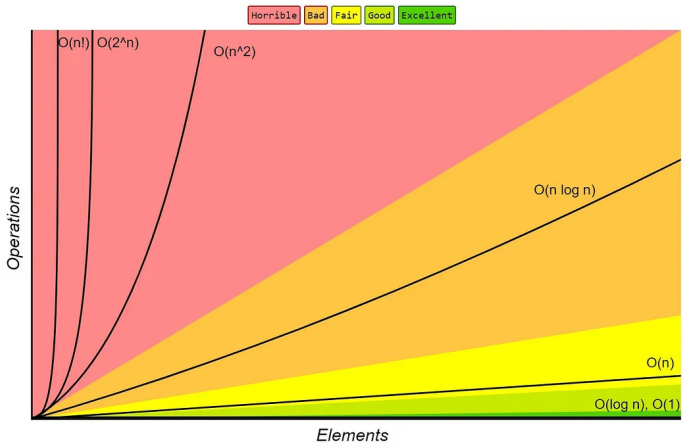


Figure 5: Asymptotic Notation

Big-O Complexity Analysis Chart from freeCodeCamp

1.7.3 Theta Notation

The ***theta notation*** or ***big-theta notation*** is used to describe the tight bound on the growth rate of an algorithm as the input size approaches infinity. It provides an upper and lower limit on the time complexity of an algorithm. The theta notation is used to analyze the efficiency of an algorithm in terms of the average number of basic operations it performs.

1.7.4 Complexity of an Algorithm

The ***complexity of an algorithm*** is a measure of the amount of time and space required to execute the algorithm as a function of the input size. It is used to analyze the efficiency of an algorithm and to compare different algorithms for the same problem. The complexity of an algorithm is usually expressed using big-O notation, which provides an upper bound on the growth rate of the algorithm as the input size increases.

1.7.4.1 Time Complexity

The ***time complexity*** of an algorithm is a measure of the amount of time required to execute the algorithm as a function of the input size. It is used to analyze the efficiency of an algorithm in terms of the number of basic operations it performs. The time complexity of an algorithm is usually expressed using big-O notation, which provides an upper bound on the growth rate of the algorithm as the input size increases.

1.7.4.1.1 Constant Time Complexity ($O(1)$)

An algorithm is said to have a ***constant time complexity*** if the execution time of the algorithm does not depend on the input size. It means that the algorithm takes the same amount of time to execute regardless of the input size. An example of an algorithm with constant time complexity is accessing an element in an array using its index.

```
1 int arr[5] = {1, 2, 3, 4, 5};
2 int x = arr[2]; // Accessing the element at index 2
```

Code 1.35: Constant Time Complexity

1.7.4.1.2 Logarithmic Time Complexity ($O(\log n)$)

An algorithm is said to have a ***logarithmic time complexity*** if the execution time of the algorithm grows logarithmically as the input size increases. An example of an algorithm with logarithmic time complexity is binary search, where the input size is halved at each step.

```
int binarySearch(int arr[], int n, int x) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x) return mid;
        else if (arr[mid] < x) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

Code 1.36: Logarithmic Time Complexity

1.7.4.1.3 Linear Time Complexity ($O(n)$)

An algorithm is said to have a ***linear time complexity*** if the execution time of the algorithm grows linearly as the input size increases. It means that the algorithm takes a constant amount of time to process each element in the input. An example of an algorithm with linear time complexity is traversing an array to find the maximum element.

```
int findMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) max = arr[i];
    }
    return max;
}
```

Code 1.37: Linear Time Complexity

1.7.4.1.4 Linearithmic Time Complexity ($O(n \log n)$)

An algorithm is said to have a ***linearithmic time complexity*** if the execution time of the algorithm grows linearithmically as the input size increases. An example of an algorithm with linearithmic time complexity is sorting an array using the merge sort algorithm.

```
void merge(int arr[], int l, int m, int r) {
    // Merge two subarrays of arr[]
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = new int[n1];
    int *R = new int[n2];
```

```
9
10      for (i = 0; i < n1; i++) L[i] = arr[l + i];
11      for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
12
13      i = 0; j = 0; k = l;
14      while (i < n1 && j < n2) {
15          if (L[i] <= R[j]) arr[k++] = L[i++];
16          else arr[k++] = R[j++];
17      }
18
19      while (i < n1) arr[k++] = L[i++];
20      while (j < n2) arr[k++] = R[j++];
21  }
22
23  void mergeSort(int arr[], int l, int r) {
24      if (l < r) {
25          int m = l + (r - l) / 2;
26          mergeSort(arr, l, m);
27          mergeSort(arr, m + 1, r);
28          merge(arr, l, m, r);
29      }
30  }
```

Code 1.38: Linearithmic Time Complexity

1.7.4.1.5 Quadratic Time Complexity ($O(n^2)$)

An algorithm is said to have a ***quadratic time complexity*** if the execution time of the algorithm grows quadratically as the input size increases. It means that the time taken by the algorithm to process each element in the input is proportional to the square of the input size. An example of an algorithm with quadratic time complexity is the bubble sort algorithm.

```
1   void bubbleSort(int arr[], int n) {
2       for (int i = 0; i < n - 1; i++) {
3           for (int j = 0; j < n - i - 1; j++) {
4               if (arr[j] > arr[j + 1]) {
5                   int temp = arr[j];
6                   arr[j] = arr[j + 1];
7                   arr[j + 1] = temp;
8               }
9           }
10      }
11  }
```

Code 1.39: Quadratic Time Complexity

Another common example of an algorithm with quadratic time complexity is a nested loop that iterates over all pairs of elements in an array.

1.7.4.1.6 Exponential Time Complexity ($O(2^n)$)

An algorithm is said to have an ***exponential time complexity*** if the execution time of the algorithm grows exponentially as the input size increases. It means that the time taken by the algorithm increases exponentially with each additional element in the input. An example of an algorithm with exponential time complexity is the recursive Fibonacci sequence algorithm.

```
int fibonacci(int n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Code 1.40: Exponential Time Complexity

1.7.4.1.7 Factorial Time Complexity ($O(n!)$)

An algorithm is said to have a ***factorial time complexity*** if the execution time of the algorithm grows factorially as the input size increases. It means that the time taken by the algorithm increases a factorial number of times with each additional element in the input. An example of an algorithm with factorial time complexity is the permutation algorithm that generates all possible permutations of a set of elements.

```
void permute(string str, int l, int r) {
    if (l == r) cout << str << endl;
    else {
        for (int i = l; i <= r; i++) {
            swap(str[l], str[i]);
            permute(str, l + 1, r);
            swap(str[l], str[i]);
        }
    }
}
```

Code 1.41: Factorial Time Complexity

1.7.4.2 Space Complexity

The ***space complexity*** of an algorithm is a measure of the amount of memory required to execute the algorithm as a function of the input size. It is used to analyze the efficiency of an algorithm in terms of the amount of memory it uses. The space complexity of an algorithm is usually expressed using big-O notation, which provides an upper bound on the amount of memory the algorithm uses as the input size increases.

1.7.4.2.1 Constant Space Complexity ($O(1)$)

An algorithm is said to have a ***constant space complexity*** if the amount of memory required to execute the algorithm does not depend on the input size. It means that the algorithm uses a fixed amount of memory to process the input. An example of an algorithm with constant space complexity is swapping two variables without using a temporary variable.

```
void swap(int &a, int &b) {
    a = a + b;
    b = a - b;
    a = a - b;
}
```

Code 1.42: Constant Space Complexity

1.7.4.2.2 Linear Space Complexity ($O(n)$)

An algorithm is said to have a ***linear space complexity*** if the amount of memory required to execute the algorithm grows linearly as the input size increases. It means that the algorithm uses a memory space that is proportional to the input size. An example of an algorithm with linear space complexity is storing the elements of an array in a separate array in reverse order.

```
void reverseArray(int arr[], int n) {
    int start = 0, end = n - 1;
    while (start < end) {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}
```

Code 1.43: Linear Space Complexity

1.7.4.2.3 Quadratic Space Complexity ($O(n^2)$)

An algorithm is said to have a ***quadratic space complexity*** if the amount of memory required to execute the algorithm grows quadratically as the input size increases. It means that the algorithm uses a memory space that is proportional to the square of the input size. An example of an algorithm with quadratic space complexity is storing all pairs of elements in an array in a separate array.

```
void allPairs(int arr[], int n) {
    vector<int> pairs(n * n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            pairs[i * n + j] = arr[i] + arr[j];
        }
    }
}
```

Code 1.44: Quadratic Space Complexity

1.7.4.2.4 Exponential Space Complexity ($O(2^n)$)

An algorithm is said to have an ***exponential space complexity*** if the amount of memory required to execute the algorithm grows exponentially as the input size increases. An example of an algorithm with exponential space complexity is generating all subsets of a set of elements.

```
void generateSubsets(int arr[], int n) {
    for (int i = 0; i < (1 << n); i++) {
        for (int j = 0; j < n; j++) {
            if (i & (1 << j)) cout << arr[j] << " ";
        }
        cout << endl;
    }
}
```

Code 1.45: Exponential Space Complexity

1.7.4.2.5 Factorial Space Complexity ($O(n!)$)

An algorithm is said to have a ***factorial space complexity*** if the amount of memory required to execute the algorithm grows factorially as the input size increases. An example of an algorithm with factorial space complexity is generating all permutations of a set of elements.

```
void permute(string str, int l, int r) {
    if (l == r) cout << str << endl;
    else {
        for (int i = l; i <= r; i++) {
            swap(str[l], str[i]);
            permute(str, l + 1, r);
            swap(str[l], str[i]);
        }
    }
}
```

Code 1.46: Factorial Space Complexity

1.8 Summary

In this chapter, we introduced the fundamental concepts of data structures and algorithms. We discussed the importance of data structures and algorithms in computer science and software engineering. We also covered some basic terminologies related to data structures and algorithms, such as data, data object, data type, abstract data type, and complexity of an algorithm. We introduced the concept of asymptotic notations, such as big-O notation, omega notation, and theta notation, and discussed the time complexity of algorithms in terms of big-O notation. We covered common time complexity ranges from best to worst performance, such as constant time complexity, logarithmic time complexity, linear time complexity, linearithmic time complexity, quadratic time complexity, exponential time complexity, and factorial time complexity.

1.9 Coding Exercises

1. Implement a C++ program that demonstrates the primitive data types.
   (a) Declare and initialize variables of the following different data types.
      i. Integer
      ii. Float
      iii. Double
      iv. Character
      v. Boolean
   (b) Print the values of the variables to the console.
2. Implement a C++ program to find the maximum element in an array using linear time complexity.
   (a) Declare an array of integers.

   ```
   int arr[6];
   ```

   (b) Initialize the array with random values.

   ```
   arr[6] = {19, 10, 8, 17, 9, 15};
   ```

   (c) Find the maximum element in the array.
   (d) Print the maximum element to the console.

   Output: 19

   (e) Determine the **time complexity** and **space complexity** of the program.
3. Implement a C++ program to find the sum of all elements in an array using linear time complexity.
   (a) Declare an array of integers.

   ```
   int arr[6];
   ```

   (b) Initialize the array with random values.

   ```
   arr[6] = {19, 10, 8, 17, 9, 15};
   ```

   (c) Find the sum of all elements in the array.
   (d) Print the sum to the console.

   Output: 78

   (e) Determine the **time complexity** and **space complexity** of the program.

2

Arrays and Linked Lists

2.1 Introduction

Some of the most basic and fundamental data structures in computer science are arrays and linked lists. These data structures are used to store and manipulate collections of elements in a computer program. In this chapter, we will discuss the properties, operations, and complexity analysis of arrays and linked lists.

2.2 Arrays

An ***array*** is a collection of elements of the same data type that are stored in contiguous memory locations. It is used to store multiple values of the same type under a single name. The elements of an array can be accessed using an index value. In C++, arrays are zero-indexed, which means the first element is at index 0. Arrays also have a fixed size that is specified at the time of declaration.

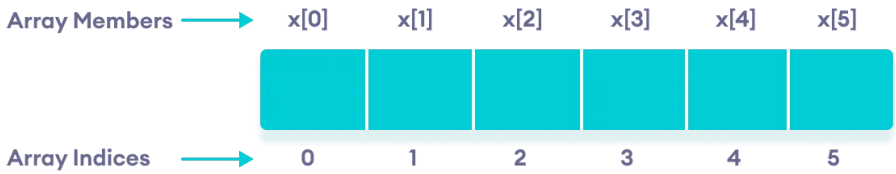


Figure 6: Elements of an array in C++

Elements of an array in C++ from Programiz

Figure 6 shows the the visual representation of the elements of an array in C++. It shows the array members and indices.

```cpp
// array.cpp
int main() {
    int arr[6];
    return 0;
}
```

Code 2.1: Array Declaration

The above code snippet declares an array named `arr` of size 6 that can store 6 integer values. The elements of the array are accessed using index values from 0 to 5 as shown in Figure 6.

```cpp
// array_assign.cpp
int main() {
    int arr[6];
    arr[0] = 19;
    arr[1] = 10;
    arr[2] = 8;
    return 0;
}
```

Code 2.2: Assigning Values to Array Elements

The above code snippet assigns values to the elements of the array `arr` at index 0, 1, and 2. The elements of the array can be accessed and modified using their index values. Array elements that are not explicitly initialized are assigned default values based on their data type. For example, integer elements are initialized to 0.

Figure 7: Initializing Array Elements

Initializing Array Elements from Programiz

Figure 7 shows the code for initializing array elements in C++. The array elements are initialized using curly braces `{}` with the values separated by commas. When the size of the array is specified, the number of elements in the initialization list must match the size of the array. If the size of the array is not specified, the size is automatically determined based on the number of elements in the array during initialization.

```cpp
int main() {
    int arr[6] = {19, 10, 8, 17, 9, 15};
    return 0;
}
```

Code 2.3: Initializing Array Elements

The above code snippet initializes the elements of the array `arr` with the values 19, 10, 8, 17, 9, and 15. The size of the array is specified as 6, and the number of elements in the initialization list matches the size of the array.

```cpp
int main() {
    int arr[] = {19, 10, 8, 17, 9, 15};
```

```
    return 0;
}
```

Code 2.4: Initializing Array Elements with Unspecified Size

The above code snippet initializes the elements of the array `arr` with the values 19, 10, 8, 17, 9, and 15. The size of the array is not specified and is automatically determined based on the number of elements in the initialization list.

Figure 8: Initializing Array Elements with Empty Members

Initializing Array Elements with Empty Members from Programiz

Figure 8 shows the code for initializing array elements with empty members in C++. The array elements are initialized using curly braces `{}` with empty members. Empty members only appear at the end of the initialization list and are assigned default values based on their data type. For example, integer elements are initialized to 0.

```
int main() {
    int arr[6] = {19, 10, 8};
    return 0;
}
```

Code 2.5: Initializing Array Elements with Empty Members

The above code snippet initializes the first three elements of the array `arr` with the values 19, 10, and 8. The remaining elements of the array are initialized to 0, which is the default value for integer elements.

2.2.1 Types of Arrays

There are two main types of arrays in C++: one-dimensional arrays and multi-dimensional arrays.

2.2.1.1 One-dimensional Array

A ***one-dimensional array*** is a collection of elements of the same data type that are stored in a single row. It is the most common type of array used in computer programming. The elements of a one-dimensional array are accessed using a single index value.

Figure 9 shows the visual representation of a one-dimensional array in C++. As shown in the figure, the elements of the array are stored in a single row, and each element is accessed using a single index value.

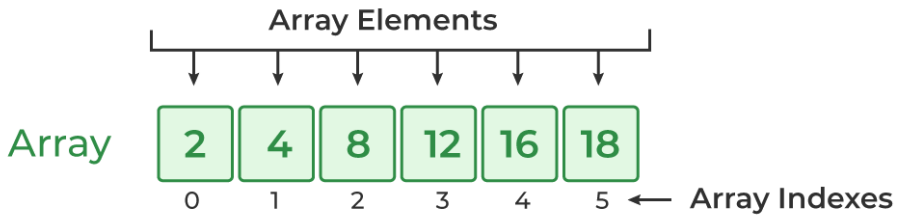


Figure 9: One-dimensional Array in C++

One-dimensional Array in C++ from GeekforGeeks

```
int main() {
    int arr[6] = {2, 4, 8, 12, 16, 18};
    return 0;
}
```

Code 2.6: One-dimensional Array

The above code snippet declares and initializes a one-dimensional array named `arr` with 6 integer elements. A one-dimensional array only has one set of square brackets `[]`. One set of square brackets signifies that the array is one-dimensional.

2.2.1.2 Multi-dimensional Array

A ***multi-dimensional array*** is a collection of elements of the same data type that are stored in multiple rows and columns. It is used to store data in a tabular format. The elements of a multi-dimensional array are accessed using multiple index values.

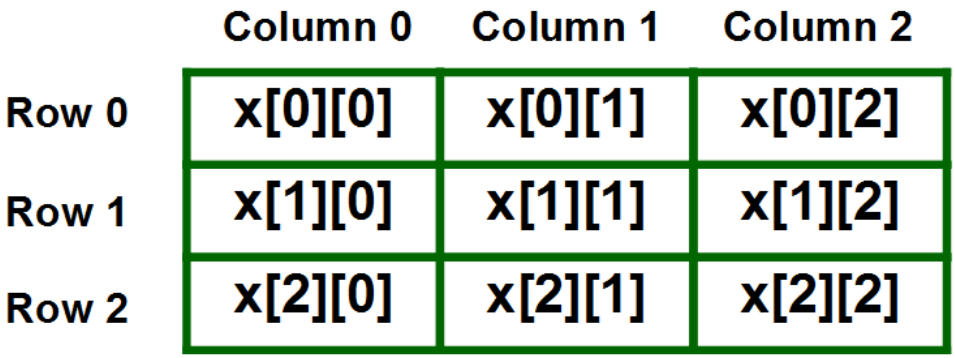


Figure 10: Two-dimensional Array in C++

Two-dimensional Array in C++ from GeekforGeeks

Figure 10 shows the visual representation of a two-dimensional array in C++. As shown in the figure, the elements of the array are stored in multiple rows and columns, and each element is accessed using two index values. The number of elements in a two-dimensional array is determined by the number of rows and columns.

```
int main() {
    int arr[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    return 0;
}
```

Code 2.7: Two-dimensional Array

The above code snippet declares and initializes a two-dimensional array named `arr` with 3 rows and 3 columns. A two-dimensional array has two sets of square brackets `[][]`. Two sets of square brackets signify that the array is two-dimensional. The number of rows and columns in a two-dimensional array is specified within the square brackets. The first set of square brackets specifies the number of rows, and the second set of square brackets specifies the number of columns. Thus, in the above example, the array `arr` has 3 rows and 4 columns.

```
int main() {
    int arr[3][4] = {
        {1, 2},
        {5, 6, 7},
        {9}
    };
    return 0;
}
```

Code 2.8: Two-dimensional Array with Empty Members

The above code snippet initializes the elements of the two-dimensional array `arr` with empty members. The first row of the array has 2 elements, the second row has 3 elements, and the third row has 1 element. The remaining elements of the array are initialized to 0, which is the default value for integer elements.

2.2.2 Array Operations

Arrays support various operations such as insertion, deletion, and searching of elements. These operations are essential for manipulating the elements of an array and performing various tasks in a computer program.

2.2.2.1 Insertion

The ***insertion*** operation is used to add an element to an array at a specific position. The element is inserted at the specified index, and the existing elements are shifted to accommodate the new element.

Figure 11 shows the visual representation of the insertion operation in an array. The element 12 is inserted at index 2 of the array, and the existing elements are shifted to accommodate the new element.

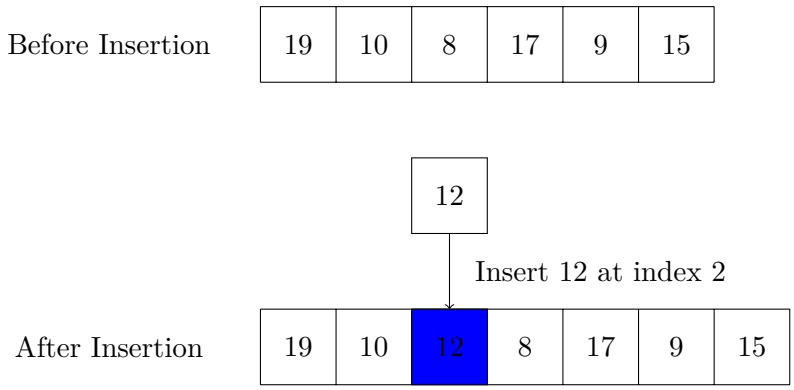


Figure 11: Array Insertion

```cpp
int main() {
    int arr[100] = {19, 10, 8, 17, 9, 15};
    int n = 6;
    int index = 2;
    int value = 12;

    for (int i = n - 1; i >= index; i--) {
        arr[i + 1] = arr[i];
    }
    arr[index] = value;
    n++;

    return 0;
}
```

Code 2.9: Array Insertion

Code 2.9 shows the code for inserting an element into an array in C++. The code snippet declares an array `arr` of size 100 and initializes it with 6 elements. It then inserts the element 12 at index 2 of the array and shifts the existing elements to accommodate the new element.

2.2.2.2 Deletion

The ***deletion*** operation is used to remove an element from an array at a specific position. The element is deleted from the specified index, and the remaining elements are shifted to fill the gap created by the deletion.

Figure 12 shows the visual representation of the deletion operation in an array. The element 8 is deleted from index 2 of the array, and the remaining elements are shifted to fill the gap created by the deletion.

```cpp
int main() {
    int arr[100] = {19, 10, 8, 17, 9, 15};
    int n = 6;
    int index = 2;

    for (int i = index; i < n - 1; i++) {
```

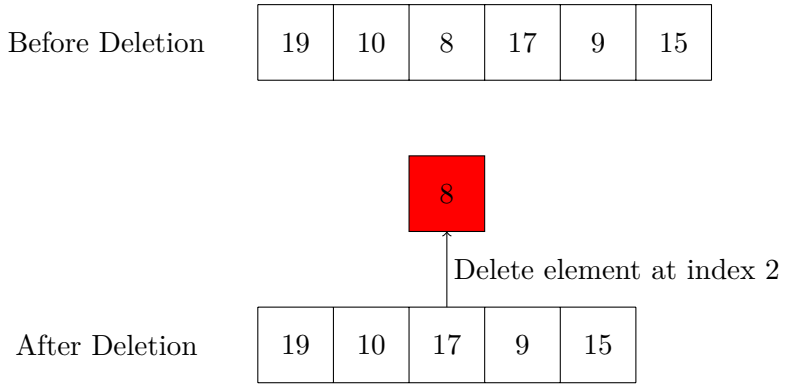


Figure 12: Array Deletion

```
        arr[i] = arr[i + 1];
    }
    n--;

    return 0;
}
```

Code 2.10: Array Deletion

Code 2.10 shows the code for deleting an element from an array in C++. The code snippet declares an array `arr` of size 100 and initializes it with 6 elements. It then deletes the element at index 2 of the array and shifts the remaining elements to fill the gap created by the deletion.

2.2.2.3 Searching

The ***searching*** operation is used to find the position of an element in an array. The element is searched for in the array, and the index of the element is returned if it is found. If the element is not found, a special value such as -1 is returned to indicate that the element is not present in the array.

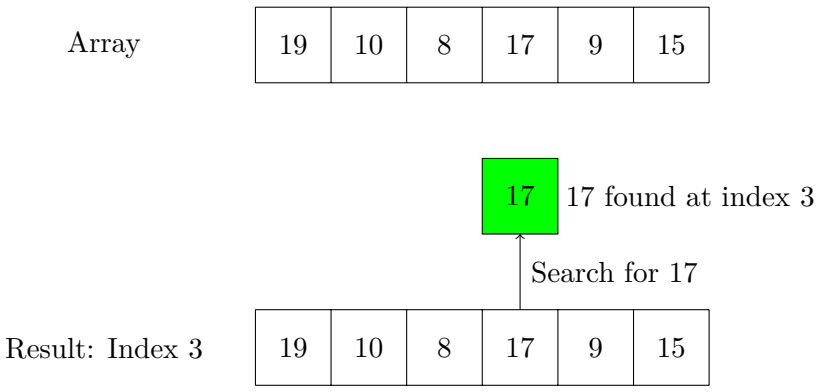


Figure 13: Array Searching

Figure 13 shows the visual representation of the searching operation in an array. The element 17 is searched for in the array, and the index of the element is returned if it is found. In this case, the element 17 is found at index 3 of the array.

```cpp
int main() {
    int arr[100] = {19, 10, 8, 17, 9, 15};
    int n = 6;
    int value = 17;
    int index = -1;

    for (int i = 0; i < n; i++) {
        if (arr[i] == value) {
            index = i;
            break;
        }
    }

    return 0;
}
```

Code 2.11: Array Searching

Code 2.11 shows the code for searching an element in an array in C++. The code snippet declares an array `arr` of size 100 and initializes it with 6 elements. It then searches for the element 17 in the array and returns the index of the element if it is found. In this case, the element 17 is found at index 3 of the array. The searching algorithm used in the example is a linear search algorithm.

2.3 Linked Lists

A ***linked list*** is a data structure that consists of a sequence of elements called nodes. A node contains a data part and a reference part that points to the next or previous node in the sequence. Linked lists are used to store and manipulate collections of elements in a computer program. Unlike arrays, linked lists do not require contiguous memory locations, and the size of a linked list can grow or shrink dynamically.

2.3.1 Types of Linked Lists

2.3.1.1 Singly Linked List

A ***singly linked list*** is a type of linked list in which each node contains a data part and a next part that points to the next node in the sequence. The last node in the list points to a special value called NULL to indicate the end of the list.

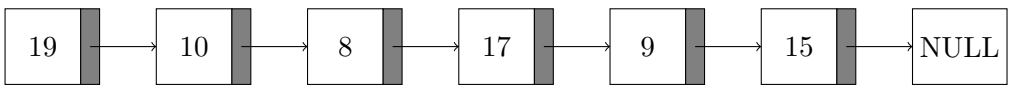


Figure 14: Singly Linked List

Figure 14 shows the visual representation of a singly linked list. The linked list contains nodes with data values 19, 10, 8, 17, 9, and 15. Each node points to the next node in the sequence, and the last node points to NULL to indicate the end of the list.

```cpp
struct Node {
    int data;
```

```
    Node *next;

    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};

int main() {
    Node *first = new Node(19);
    Node *second = new Node(10);
    Node *third = new Node(8);
    Node *fourth = new Node(17);
    Node *fifth = new Node(9);
    Node *sixth = new Node(15);

    Node *head = first;

    first->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = sixth;

    return 0;
}
```

Code 2.12: Singly Linked List

Code 2.12 shows the code for creating a singly linked list in C++. The code snippet defines a structure `Node` that contains a data part and a next part that points to the next node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. Each node points to the next node in the sequence, and the last node points to NULL to indicate the end of the list.

2.3.1.2 Doubly Linked List

A ***doubly linked list*** is a type of linked list in which each node contains a data part, a next part that points to the next node in the sequence, and a previous part that points to the previous node in the sequence. The first node in the list points to NULL in the previous part, and the last node in the list points to NULL in the next part.

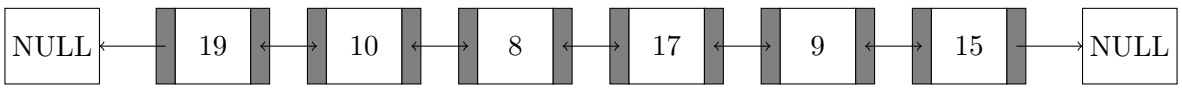


Figure 15: Doubly Linked List

Figure 15 shows the visual representation of a doubly linked list. The linked list contains nodes with data values 19, 10, 8, 17, 9, and 15. Each node points to the next and previous nodes in the sequence, and the first and last nodes point to NULL to indicate the end of the list.

```
struct Node {
    int data;
    Node *next;
    Node *prev;

    Node(int data) {
        this->data = data;
        this->next = NULL;
        this->prev = NULL;
    }
};

int main() {
    Node *first = new Node(19);
    Node *second = new Node(10);
    Node *third = new Node(8);
    Node *fourth = new Node(17);
    Node *fifth = new Node(9);
    Node *sixth = new Node(15);

    Node *head = first;

    first->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = sixth;

    second->prev = first;
    third->prev = second;
    fourth->prev = third;
    fifth->prev = fourth;
    sixth->prev = fifth;

    return 0;
}
```

Code 2.13: Doubly Linked List

Code 2.13 shows the code for creating a doubly linked list in C++. The code snippet defines a structure `Node` that contains a data part, a next part that points to the next node in the sequence, and a previous part that points to the previous node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. Each node points to the next and previous nodes in the sequence, and the first and last nodes point to NULL to indicate the end of the list.

2.3.1.3 Circular Linked List

A ***circular linked list*** is a type of linked list in which the last node in the list points back to the first node, forming a circular loop. This allows traversal of the list in a circular manner, starting from any node in the list.

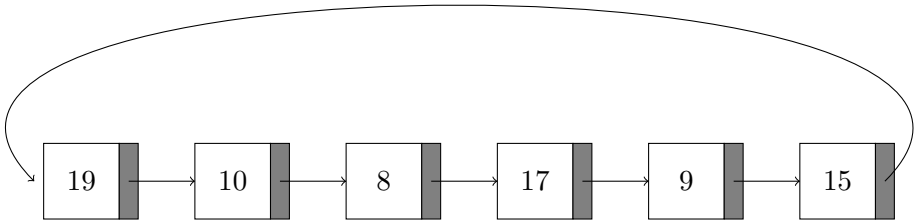


Figure 16: Circular Linked List

Figure 16 shows the visual representation of a circular linked list. The linked list contains nodes with data values 19, 10, 8, 17, 9, and 15. Each node points to the next node in the sequence, and the last node points back to the first node, forming a circular loop.

```
struct Node {
    int data;
    Node *next;

    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};

int main() {
    Node *first = new Node(19);
    Node *second = new Node(10);
    Node *third = new Node(8);
    Node *fourth = new Node(17);
    Node *fifth = new Node(9);
    Node *sixth = new Node(15);

    Node *head = first;

    first->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = sixth;
    sixth->next = head;

    return 0;
}
```

Code 2.14: Circular Linked List

Code 2.14 shows the code for creating a circular linked list in C++. The code snippet defines a structure `Node` that contains a data part and a next part that points to the next node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. Each node points to the next node in the sequence, and the last node points back to the first

node, forming a circular loop.

2.3.1.4 Doubly Circular Linked List

A ***doubly circular linked list*** is a type of linked list in which each node contains a data part, a next part that points to the next node in the sequence, and a previous part that points to the previous node in the sequence. The first node in the list points to the last node in the previous part, and the last node in the list points to the first node in the next part, forming a circular loop.

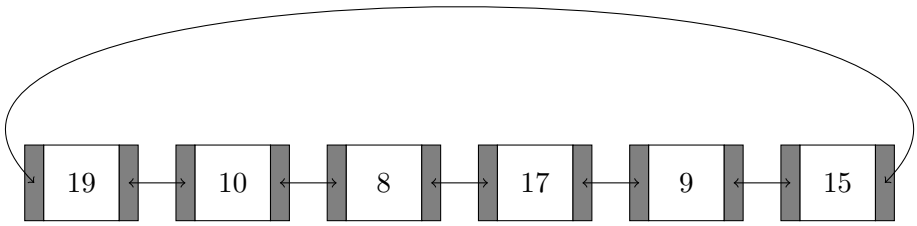


Figure 17: Doubly Circular Linked List

Figure 17 shows the visual representation of a doubly circular linked list. The linked list contains nodes with data values 19, 10, 8, 17, 9, and 15. Each node points to the next and previous nodes in the sequence, and the first and last nodes point to each other, forming a circular loop.

```
struct Node {
    int data;
    Node *next;
    Node *prev;

    Node(int data) {
        this->data = data;
        this->next = NULL;
        this->prev = NULL;
    }
};

int main() {
    Node *first = new Node(19);
    Node *second = new Node(10);
    Node *third = new Node(8);
    Node *fourth = new Node(17);
    Node *fifth = new Node(9);
    Node *sixth = new Node(15);

    Node *head = first;

    first->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = sixth;
```

```
    sixth->next = first;

    first->prev = sixth;
    second->prev = first;
    third->prev = second;
    fourth->prev = third;
    fifth->prev = fourth;
    sixth->prev = fifth;

    return 0;
}
```

Code 2.15: Doubly Circular Linked List

Code 2.15 shows the code for creating a doubly circular linked list in C++. The code snippet defines a structure `Node` that contains a data part, a next part that points to the next node in the sequence, and a previous part that points to the previous node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. Each node points to the next and previous nodes in the sequence, and the first and last nodes point to each other, forming a circular loop.

2.3.2 Operations on Linked Lists

2.3.2.1 Insertion

The ***insertion*** operation is used to add a new node to a linked list at a specific position. The new node is inserted at the specified position, and the existing nodes are adjusted to accommodate the new node.

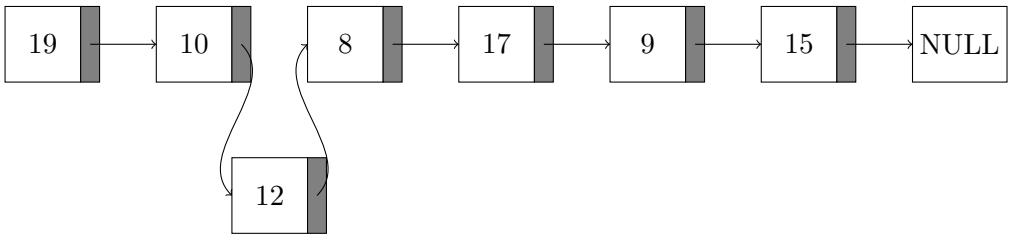


Figure 18: Linked List Insertion

Figure 18 shows the visual representation of the insertion operation in a linked list. The element 12 is inserted at index 2 of the linked list, and the existing nodes are adjusted to accommodate the new node.

```
struct Node {
    int data;
    Node *next;

    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};
```

```
10
11 int main() {
12     Node *first = new Node(19);
13     Node *second = new Node(10);
14     Node *third = new Node(8);
15     Node *fourth = new Node(17);
16     Node *fifth = new Node(9);
17     Node *sixth = new Node(15);
18
19     Node *head = first;
20
21     first->next = second;
22     second->next = third;
23     third->next = fourth;
24     fourth->next = fifth;
25     fifth->next = sixth;
26
27     Node *newNode = new Node(12);
28     Node *temp = head;
29     int index = 2;
30
31     for (int i = 0; i < index - 1; i++) {
32         temp = temp->next;
33     }
34
35     newNode->next = temp->next;
36     temp->next = newNode;
37
38     return 0;
39 }
```

Code 2.16: Linked List Insertion

Code 2.16 shows the code for inserting an element in a linked list in C++. The code snippet defines a structure `Node` that contains a data part and a next part that points to the next node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. The element 12 is inserted at index 2 of the linked list, and the existing nodes are adjusted to accommodate the new node.

2.3.2.2 Deletion

The ***deletion*** operation is used to remove a node from a linked list at a specific position. The node at the specified position is removed, and the existing nodes are adjusted to maintain the integrity of the list.

Figure 19 shows the visual representation of the deletion operation in a linked list. The element 10 is deleted from index 2 of the linked list, and the existing nodes are adjusted to maintain the integrity of the list.

```
1 struct Node {
2     int data;
```

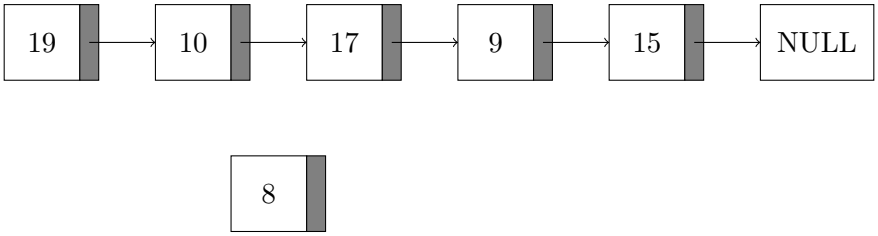


Figure 19: Linked List Deletion

```
    Node *next;

    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};

int main() {
    Node *first = new Node(19);
    Node *second = new Node(10);
    Node *third = new Node(8);
    Node *fourth = new Node(17);
    Node *fifth = new Node(9);
    Node *sixth = new Node(15);

    Node *head = first;

    first->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = sixth;

    Node *temp = head;
    int index = 2;

    for (int i = 0; i < index - 1; i++) {
        temp = temp->next;
    }

    Node *deletedNode = temp->next;
    temp->next = temp->next->next;
    delete deletedNode;

    return 0;
}
```

Code 2.17: Linked List Deletion

Code 2.17 shows the code for deleting an element from a linked list in C++. The code snippet

defines a structure `Node` that contains a data part and a next part that points to the next node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. The element 10 is deleted from index 2 of the linked list, and the existing nodes are adjusted to maintain the integrity of the list.

2.3.2.3 Searching

The ***searching*** operation is used to find a specific element in a linked list. The list is traversed from the head node to the last node to find the element with the specified value.

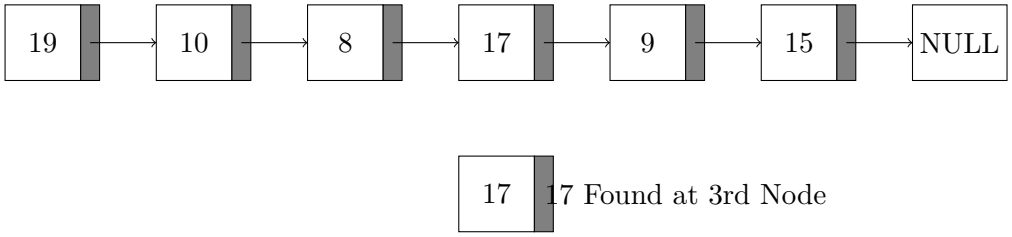


Figure 20: Linked List Searching

Figure 20 shows the visual representation of the searching operation in a linked list. The element 17 is searched for in the linked list, and the list is traversed from the head node to the last node to find the element with the specified value.

```
struct Node {
    int data;
    Node *next;

    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};

int main() {
    Node *first = new Node(19);
    Node *second = new Node(10);
    Node *third = new Node(8);
    Node *fourth = new Node(17);
    Node *fifth = new Node(9);
    Node *sixth = new Node(15);

    Node *head = first;

    first->next = second;
    second->next = third;
    third->next = fourth;
    fourth->next = fifth;
    fifth->next = sixth;

    int searchValue = 17;
    Node *temp = head;
    int index = 0;
```

```
30
31      while (temp != NULL) {
32          if (temp->data == searchValue) {
33              break;
34          }
35          temp = temp->next;
36          index++;
37      }
38
39      return 0;
40  }
```

Code 2.18: Linked List Searching

Code 2.18 shows the code for searching an element in a linked list in C++. The code snippet defines a structure `Node` that contains a data part and a next part that points to the next node in the sequence. It then creates a linked list with nodes containing data values 19, 10, 8, 17, 9, and 15. The element 17 is searched for in the linked list, and the list is traversed from the head node to the last node to find the element with the specified value.

2.4 Comparison of Arrays and Linked Lists

Unlike arrays, linked lists do not have a fixed size and can grow dynamically by adding new nodes. This makes linked lists more flexible and efficient for insertion and deletion operations, as they do not require shifting elements to accommodate new elements. However, linked lists have higher memory overhead due to the additional pointers in each node.

Arrays are more efficient for random access operations, as elements can be accessed directly using their index. In contrast, linked lists require traversing the list from the head node to the desired node, which can be slower for large lists. Arrays also have better cache locality, as elements are stored contiguously in memory, leading to faster access times.

Thus, if the application requires frequent insertion and deletion operations with a dynamic size, linked lists are a better choice. On the other hand, if the application requires frequent random access operations and has a fixed size, arrays are more suitable.

2.5 Summary

In this chapter, we discussed arrays and linked lists as fundamental data structures in computer science. We covered the basic concepts, operations, and implementations of arrays and linked lists, along with their comparison in terms of performance and memory usage. We also explored different types of linked lists, such as singly linked lists, doubly linked lists, circular linked lists, and doubly circular linked lists, and discussed the operations on linked lists, such as insertion, deletion, and searching. Finally, we compared arrays and linked lists based on their characteristics and use cases.

2.6 Coding Exercises

1. **Reverse an Array:** Write a C++ program to reverse an array of integers. The program should take the size of the array and the elements of the array as input and output the

reversed array.

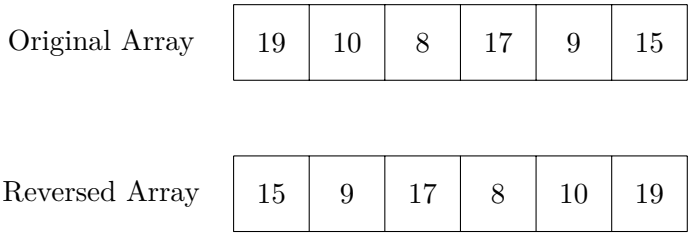
(a) Declare an array of integers with a fixed size.

int arr[6];

(b) Initialize the array with the input elements.

arr = {19, 10, 8, 17, 9, 15};

(c) Reverse the array using a loop.



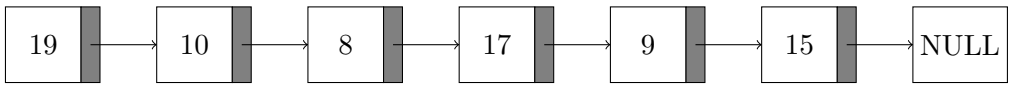
(d) Print the reversed array to the console.

Reversed Array: 15, 9, 17, 8, 10, 19

(e) Determine the **time complexity** and **space complexity** of the program.

2. **Reverse a Linked List:** Write a C++ program to reverse a singly linked list. The program should take the elements of the linked list as input and output the reversed linked list.

(a) Create a singly linked list with nodes containing the elements



(b) Reverse the linked list using a loop.

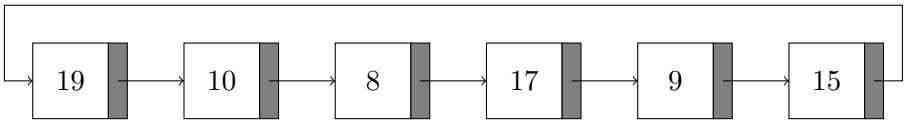
(c) Print the reversed linked list to the console.

Reversed Linked List: 15, 9, 17, 8, 10, 19

(d) Determine the **time complexity** and **space complexity** of the program.

3. **Search a Circular Linked List:** Write a C++ program to search for an element in a circular linked list. The program should take the elements of the circular linked list as input and output the position of the element in the list. If the current node's next pointer points to the head node, the list is circular. If the element is not found in the list, output "Element not found." Else, output the position of the element in the list.

(a) Create a circular linked list with nodes containing the elements



(b) Using *cin*, take the input element to search for

Search Element: 21

(c) Search for the element in the circular linked list

Output: Element not found.

(d) Determine the **time complexity** and **space complexity** of the program.

3

Stacks and Queues

3.1 Introduction

'Stack' and 'Queue' are two fundamental data structures in computer science that are used to store and manage data in a specific order. They are widely used in various applications, such as operating systems, compilers, and network protocols, to manage data efficiently.

3.2 Stack

A 'Stack' is a linear data structure that follows the Last In First Out (LIFO) principle, where the last element inserted is the first one to be removed. The operations on a stack are 'Push' (to insert an element), 'Pop' (to remove an element), 'Top' (to get the top element), 'Size' (to get the number of elements), and 'Empty' (to check if the stack is empty).

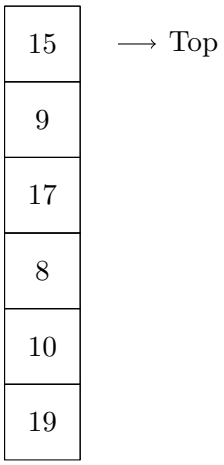


Figure 21: Stack Data Structure

Figure 21 shows the visual representation of a stack data structure. The stack contains elements 19, 10, 8, 17, 9, and 15, where 15 is the top element of the stack.

3.2.1 Operations on Stack

3.2.1.1 Push

The 'Push' operation is used to insert an element into the stack. The new element is added to the top of the stack.

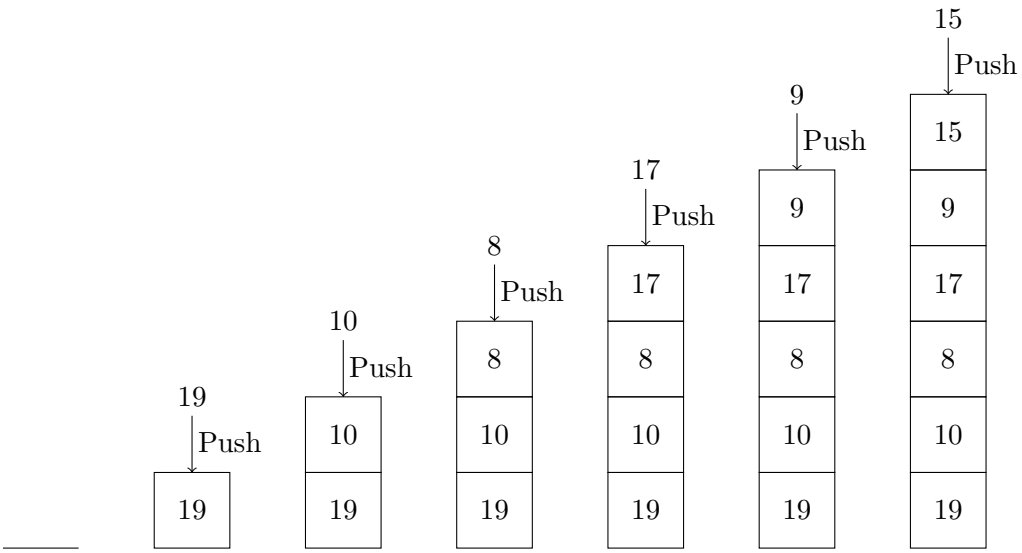


Figure 22: Stack Push Operation

Figure 22 shows the visual representation of the 'Push' operation on a stack. The stack is initially empty, and elements 19, 10, 8, 17, 9, and 15 are pushed onto the stack one by one.

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s;

    s.push(19);
    s.push(10);
    s.push(8);
    s.push(17);
    s.push(9);
    s.push(15);

    return 0;
}
```

Code 3.1: Stack Push Operation

Code 3.1 shows the code for the 'Push' operation on a stack in C++. The code snippet creates a stack using the `stack` class from the Standard Template Library (STL) and pushes elements 19, 10, 8, 17, 9, and 15 onto the stack. To 'push' an element onto the stack, the `push()` method is called with the element to be inserted as an argument.

3.2.1.2 Pop

The 'Pop' operation is used to remove the top element from the stack. The element is removed from the top of the stack, and the stack is adjusted accordingly.

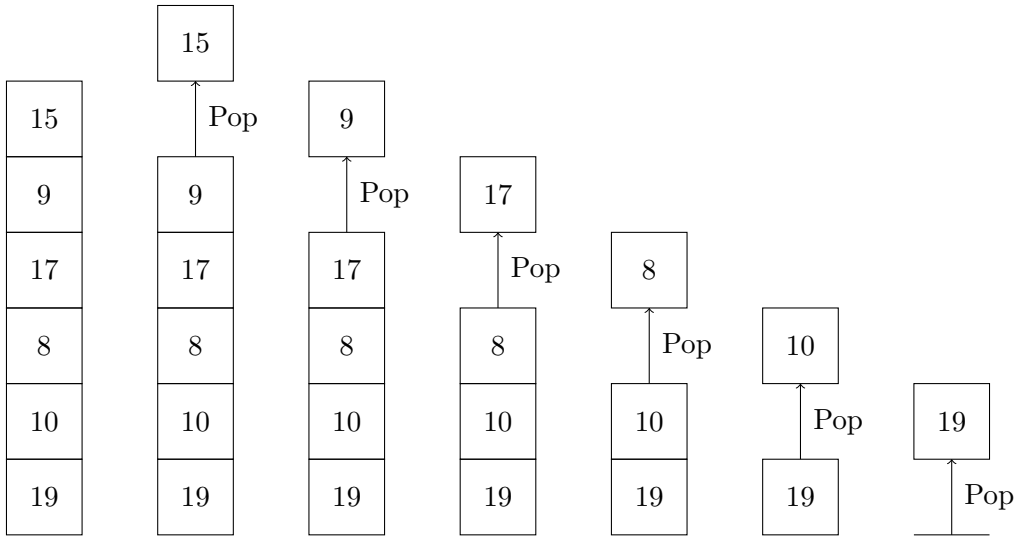


Figure 23: Stack Pop Operation

Figure 23 shows the visual representation of the 'Pop' operation on a stack. The stack contains elements 19, 10, 8, 17, 9, and 15, and each element is popped from the stack one by one.

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s = stack<int>({19, 10, 8, 17, 9, 15});

    s.pop(); // Remove 15
    s.pop(); // Remove 9
    s.pop(); // Remove 17
    s.pop(); // Remove 8
    s.pop(); // Remove 10
    s.pop(); // Remove 19

    return 0;
}
```

Code 3.2: Stack Pop Operation

Code 3.2 shows the code for the 'Pop' operation on a stack in C++. The code snippet creates a stack using the `stack` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. The elements are then popped from the stack one by one using the `pop()` method.

3.2.1.3 Top

The 'Top' operation is used to get the top element of the stack without removing it from the stack.

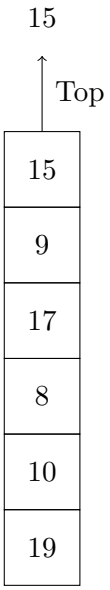


Figure 24: Stack Top Operation

Figure 24 shows the visual representation of the 'Top' operation on a stack. The stack contains elements 19, 10, 8, 17, 9, and 15, and the top element of the stack is 15.

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s = stack<int>({19, 10, 8, 17, 9, 15});

    cout << "Top Element: " << s.top() << endl;

    return 0;
}
```

Code 3.3: Stack Top Operation

Code 3.3 shows the code for the 'Top' operation on a stack in C++. The code snippet creates a stack using the `stack` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To get the top element of the stack, the `top()` method is called.

3.2.1.4 Size

The 'Size' operation is used to get the number of elements in the stack.

Figure 25 shows the visual representation of the 'Size' operation on a stack. The stack contains elements 19, 10, 8, 17, 9, and 15, and the size of the stack is 6.

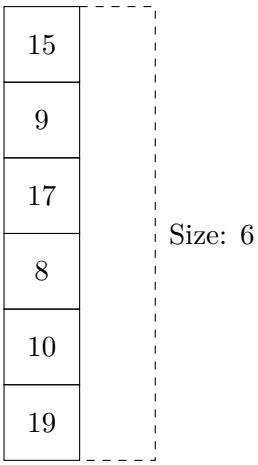


Figure 25: Stack Size Operation

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s = stack<int>({19, 10, 8, 17, 9, 15});

    cout << "Size of Stack: " << s.size() << endl;

    return 0;
}
```

Code 3.4: Stack Size Operation

Code 3.4 shows the code for the 'Size' operation on a stack in C++. The code snippet creates a stack using the `stack` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To get the size of the stack, the `size()` method is called.

3.2.1.5 Empty

The 'Empty' operation is used to check if the stack is empty or not. It returns `true` if the stack is empty and `false` otherwise.

Figure 26 shows the visual representation of the 'Empty' operation on a stack. The stack contains elements 19, 10, 8, 17, 9, and 15, and the stack is not empty.

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s = stack<int>({19, 10, 8, 17, 9, 15});

    cout << "Is Stack Empty: " << (s.empty() ? "true" : "false") << endl;
```

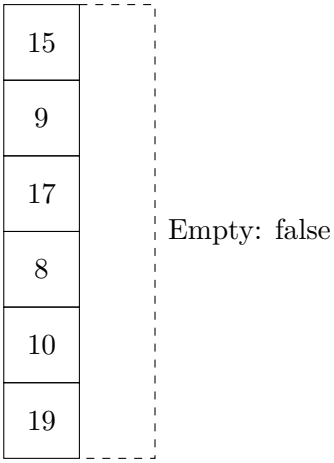


Figure 26: Stack Empty Operation

```
9
10      return 0;
11  }
```

Code 3.5: Stack Empty Operation

Code 3.5 shows the code for the 'Empty' operation on a stack in C++. The code snippet creates a stack using the `stack` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To check if the stack is empty, the `empty()` method is called.

3.3 Queue

A 'Queue' is a linear data structure that follows the First In First Out (FIFO) principle, where the first element inserted is the first one to be removed. The operations on a queue are 'Push' (to insert an element), 'Pop' (to remove an element), 'Front' (to get the front element), 'Back' (to get the back element), 'Size' (to get the number of elements), and 'Empty' (to check if the queue is empty).

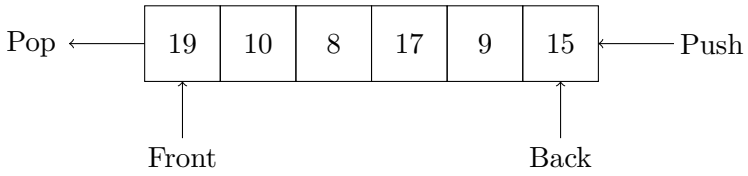


Figure 27: Queue Data Structure

Figure 27 shows the visual representation of a queue data structure. The queue contains elements 19, 10, 8, 17, 9, and 15, where 19 is the front element and 15 is the back element of the queue.

3.3.1 Operations on Queue

3.3.1.1 Push

The 'Push' operation is used to insert an element into the queue. The new element is added to the back of the queue.

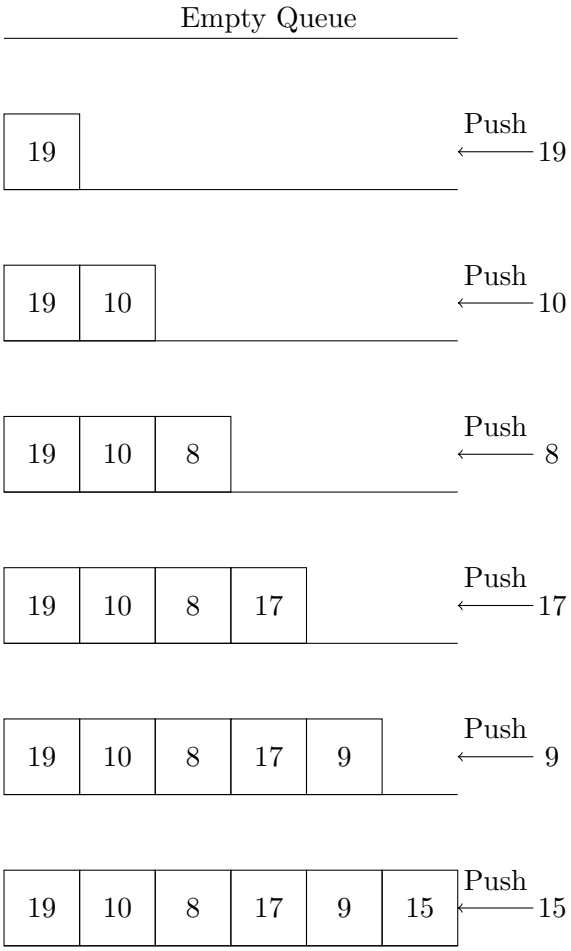


Figure 28: Queue Push Operation

Figure 28 shows the visual representation of the 'Push' operation on a queue. The queue is initially empty, and elements 19, 10, 8, 17, 9, and 15 are pushed into the queue one by one.

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;

    q.push(19);
    q.push(10);
    q.push(8);
    q.push(17);
    q.push(9);
    q.push(15);
```

```
14
15      return 0;
16  }
```

Code 3.6: Queue Push Operation

Code 3.6 shows the code for the 'Push' operation on a queue in C++. The code snippet creates a queue using the `queue` class from the Standard Template Library (STL) and pushes elements 19, 10, 8, 17, 9, and 15 into the queue. To 'push' an element into the queue, the `push()` method is called with the element to be inserted as an argument.

3.3.1.2 Pop

The 'Pop' operation is used to remove the front element from the queue. The element is removed from the front of the queue, and the queue is adjusted accordingly.

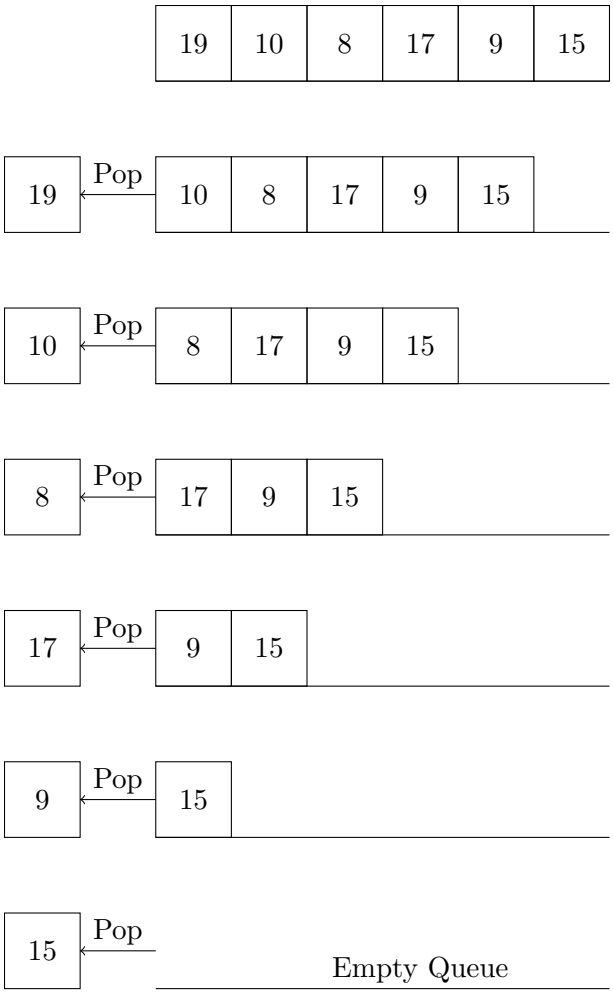


Figure 29: Queue Pop Operation

Figure 29 shows the visual representation of the 'Pop' operation on a queue. The queue contains elements 19, 10, 8, 17, 9, and 15, and each element is popped from the queue one by one.

```
1  #include <iostream>
```

```cpp
#include <queue>
using namespace std;

int main() {
    queue<int> q = queue<int>({19, 10, 8, 17, 9, 15});

    q.pop(); // Remove 19
    q.pop(); // Remove 10
    q.pop(); // Remove 8
    q.pop(); // Remove 17
    q.pop(); // Remove 9
    q.pop(); // Remove 15

    return 0;
}
```

Code 3.7: Queue Pop Operation

Code 3.7 shows the code for the 'Pop' operation on a queue in C++. The code snippet creates a queue using the `queue` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. The elements are then popped from the queue one by one using the `pop()` method.

3.3.1.3 Front

The 'Front' operation is used to get the front element of the queue without removing it from the queue.

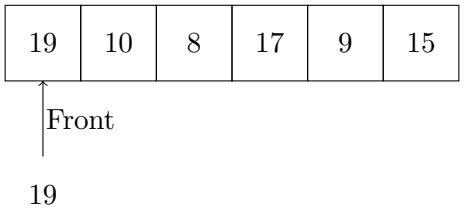


Figure 30: Queue Front Operation

Figure 30 shows the visual representation of the 'Front' operation on a queue. The queue contains elements 19, 10, 8, 17, 9, and 15, and the front element of the queue is 19.

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q = queue<int>({19, 10, 8, 17, 9, 15});

    cout << "Front Element: " << q.front() << endl;

    return 0;
}
```

Code 3.8: Queue Front Operation

Code 3.8 shows the code for the 'Front' operation on a queue in C++. The code snippet creates a queue using the `queue` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To get the front element of the queue, the `front()` method is called.

3.3.1.4 Back

The 'Back' operation is used to get the back element of the queue without removing it from the queue.

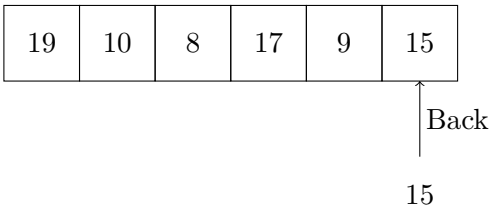


Figure 31: Queue Back Operation

Figure 31 shows the visual representation of the 'Back' operation on a queue. The queue contains elements 19, 10, 8, 17, 9, and 15, and the back element of the queue is 15.

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q = queue<int>({19, 10, 8, 17, 9, 15});

    cout << "Back Element: " << q.back() << endl;

    return 0;
}
```

Code 3.9: Queue Back Operation

Code 3.9 shows the code for the 'Back' operation on a queue in C++. The code snippet creates a queue using the `queue` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To get the back element of the queue, the `back()` method is called.

3.3.1.5 Size

The 'Size' operation is used to get the number of elements in the queue.

Figure 32 shows the visual representation of the 'Size' operation on a queue. The queue contains elements 19, 10, 8, 17, 9, and 15, and the size of the queue is 6.

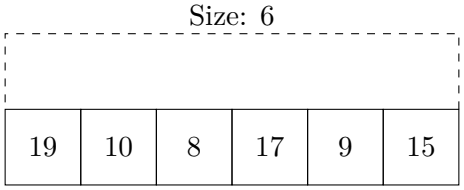


Figure 32: Queue Size Operation

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q = queue<int>({19, 10, 8, 17, 9, 15});

    cout << "Size of Queue: " << q.size() << endl;

    return 0;
}
```

Code 3.10: Queue Size Operation

Code 3.10 shows the code for the 'Size' operation on a queue in C++. The code snippet creates a queue using the `queue` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To get the size of the queue, the `size()` method is called.

3.3.1.6 Empty

The 'Empty' operation is used to check if the queue is empty or not. It returns `true` if the queue is empty and `false` otherwise.

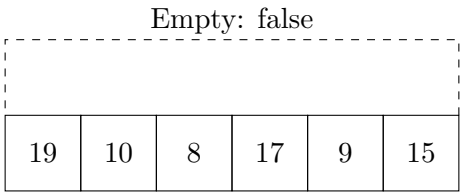


Figure 33: Queue Empty Operation

Figure 33 shows the visual representation of the 'Empty' operation on a queue. The queue contains elements 19, 10, 8, 17, 9, and 15, and the queue is not empty.

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q = queue<int>({19, 10, 8, 17, 9, 15});

```

```
8     cout << "Is Queue Empty: " << (q.empty() ? "true" : "false") << endl;
9
10    return 0;
11 }
```

Code 3.11: Queue Empty Operation

Code 3.11 shows the code for the 'Empty' operation on a queue in C++. The code snippet creates a queue using the `queue` class from the Standard Template Library (STL) and initializes it with elements 19, 10, 8, 17, 9, and 15. To check if the queue is empty, the `empty()` method is called.

3.4 Comparison of Stack and Queue

The key differences between a 'Stack' and a 'Queue' is that a stack follows the Last In First Out (LIFO) principle, while a queue follows the First In First Out (FIFO) principle. You can visualize a stack as a stack of plates, where the last plate added is the first one to be removed. On the other hand, you can visualize a queue as a line of people waiting for a bus, where the first person in line is the first one to board the bus.

In 'stack' you can only access the top element, while in 'queue' you can access both the front and back elements. In 'stack' you can only push and pop elements at the top, while in 'queue' you can push elements at the back and pop elements from the front. The time complexity of the 'push', 'pop', 'front', 'back', 'size', and 'empty' operations is the same for both stack and queue. Both also have the same space complexity. In 'stack' you can only access the top element while in 'queue' you can access both the front and back elements.

3.5 Summary

In this chapter, we discussed the 'Queue' data structure. A queue is a linear data structure that follows the First In First Out (FIFO) principle. We learned about the basic operations on a queue, such as 'Push', 'Pop', 'Front', 'Back', 'Size', and 'Empty'. We also compared the 'Stack' and 'Queue' data structures.

4

Introduction to Algorithms

4.1 Introduction

Algorithms are a fundamental concept in computer science and play a crucial role in solving computational problems efficiently. An algorithm is a step-by-step procedure or a set of rules to solve a specific problem. It is a sequence of instructions that takes an input, performs some computation, and produces an output. Algorithms are used in various applications, such as search engines, social media platforms, e-commerce websites, navigation systems, and many more.

4.2 How do Algorithms Work?

Algorithms work by taking an input, processing it through a series of steps, and producing an output. The input to an algorithm can be any data, such as numbers, strings, arrays, graphs, or any other data structure. The algorithm processes the input data using a set of rules or instructions to perform a specific task. The output of the algorithm is the result obtained after processing the input data. The output can be a single value, a set of values, or any other form of data, depending on the problem being solved.

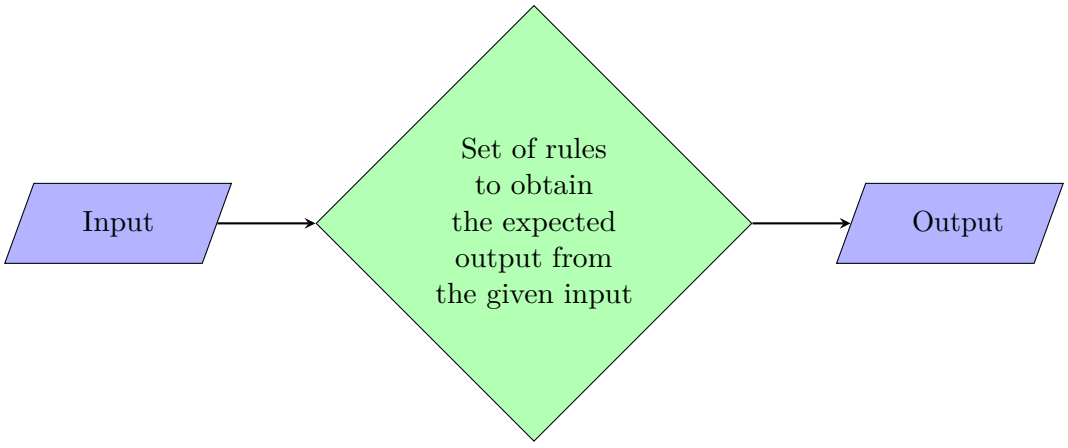


Figure 34: Flowchart for an Algorithm

Figure 34 shows a flowchart for an algorithm. The algorithm takes an input, processes it

using an algorithm, and produces an output.

4.3 Characteristics of an Algorithm

An algorithm has the following characteristics:

- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-defined Outputs**: The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finiteness**: The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible**: The algorithm must be simple, generic, and practical, such that it can be executed using reasonable constraints and resources.
- **Language Independent**: Algorithm must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

4.4 What is the Need for Algorithms?

Algorithms are essential for solving complex computational problems efficiently and effectively. They can be used to solve a wide range of problems, such as searching, sorting, optimization, and many more. They provide a systematic approach to:

- **Solving Problems**: Algorithms break down problems into smaller, manageable steps, making it easier to solve complex problems.
- **Optimizing Solutions**: Algorithms find the best or near-optimal solutions to problems, helping to optimize the performance of systems.
- **Automating Tasks**: Algorithms can automate repetitive or complex tasks, saving time and effort in performing these tasks manually.

4.5 Types of Algorithms

There are various types of algorithms based on their design and functionality. Some common types of algorithms include:

- **Brute Force Algorithm**: It is the simplest approach to a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.
- **Recursive Algorithm**: A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.
- **Backtracking**: The backtracking algorithm builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails, we trace back to the failure point build on the next solution and continue this process till we find the solution or all possible solutions are looked after.
- **Searching Algorithm**: Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.
- **Sorting Algorithm**: Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an

increasing or decreasing manner.

- **Hashing Algorithm**: Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.
- **Divide and Conquer Algorithm**: This algorithm breaks a problem into sub-problems, solves a single sub-problem, and merges the solutions to get the final solution. It consists of the following three steps: Divide, Solve, and Combine.
- **Greedy Algorithm**: In this type of algorithm, the solution is built part by part. The solution for the next part is built based on the immediate benefit of the next part. The one solution that gives the most benefit will be chosen as the solution for the next part.
- **Dynamic Programming Algorithm**: This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping sub-problems and solves them.
- **Randomized Algorithm**: In the randomized algorithm, we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

4.6 Examples of Algorithms

Below are some examples of algorithms:

- **Sorting Algorithms**: Merge sort, Quick sort, Heap sort
- **Searching Algorithms**: Linear search, Binary search, Hashing
- **Graph Algorithms**: Dijkstra's algorithm, Prim's algorithm, Floyd-Warshall algorithm
- **String Matching Algorithms**: Knuth-Morris-Pratt algorithm, Boyer-Moore algorithm

4.7 How to Write an Algorithm?

To write an algorithm, follow these steps:

- **Define the problem**: Clearly state the problem to be solved.
- **Design the algorithm**: Choose an appropriate algorithm design paradigm and develop a step-by-step procedure.
- **Implement the algorithm**: Translate the algorithm into a programming language.
- **Test and debug**: Execute the algorithm with various inputs to ensure its correctness and efficiency.
- **Analyze the algorithm**: Determine its time and space complexity and compare it to alternative algorithms.

4.8 How to Design an Algorithm?

To write an algorithm, the following things are needed as a pre-requisite:

1. The problem that is to be solved by this algorithm i.e. clear problem definition.
2. The constraints of the problem must be considered while solving the problem.
3. The input to be taken to solve the problem.
4. The output to be expected when the problem is solved.
5. The solution to this problem within the given constraints.

Then the algorithm is written with the help of the above parameters such that it solves the problem.

4.8.1 Example

Consider the example to add three numbers and print the sum.

4.8.1.1 Step 1: Fulfilling the pre-requisites

1. The problem that is to be solved by this algorithm: Add 3 numbers and print their sum.
2. The constraints of the problem that must be considered while solving the problem: The numbers must contain only digits and no other characters.
3. The input to be taken to solve the problem: The three numbers to be added.
4. The output to be expected when the problem is solved: The sum of the three numbers taken as the input i.e. a single integer value.
5. The solution to this problem, in the given constraints: The solution consists of adding the 3 numbers. It can be done with the help of the '+' operator, or bit-wise, or any other method.

4.8.1.2 Step 2: Designing the Algorithm (Pseudo-code)

1. START
2. Declare 3 integer variables num1, num2, and num3.
3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
5. Add the 3 numbers and store the result in the variable sum.
6. Print the value of the variable sum
7. END

4.8.1.3 Step 3: Testing the Algorithm

```
#include <iostream>
using namespace std;

int main() {
    int num1, num2, num3;

    int sum;

    cout << "Enter 1st number: ";
    cin >> num1;
    cout << " " << num1 << endl;

    cout << "Enter 2nd number: ";
    cin >> num2;
    cout << " " << num2 << endl;

```

```
    cout << "Enter 3rd number: ";
    cin >> num3;
    cout << " " << num3 << endl;

    sum = num1 + num2 + num3;

    cout << "Sum of the three numbers: " << sum << endl;

    return 0;
}
```

Code 4.1: Adding Three Numbers

Problem 1:

- Input: num1 = 10, num2 = 20, num3 = 30
- Output: sum = 60

Problem 2:

- Input: num1 = 5, num2 = 15, num3 = 25
- Output: sum = 45

Code 4.1 shows the code for adding three numbers in C++. The code snippet takes three numbers as input from the user, adds them, and prints the sum. Here's the step-by-step algorithm of the code:

1. Declare 3 integer variables "num1", "num2", and "num3" to store the three numbers to be added.
2. Declare an integer variable "sum" to store the sum of the three numbers.
3. Take the three numbers as input from the user and store them in the variables "num1", "num2", and "num3".
4. Add the three numbers and store the result in the variable "sum".
5. Print the sum of the three numbers.

4.9 Summary

In this chapter, we discussed the concept of algorithms and their importance in computer science. Algorithms are step-by-step procedures or a set of rules to solve a specific problem. We learned about the characteristics of an algorithm, the need for algorithms, types of algorithms, and how to write and design an algorithm. We also saw an example of adding three numbers using an algorithm.

5

Searching Algorithms

5.1 Introduction

Searching is the process of finding a specific element in a collection of elements. Searching algorithms are used to search for an element in a data structure, such as an array, list, tree, or graph. There are various searching algorithms, each with its own approach and complexity. Searching algorithms play a crucial role in various applications, such as search engines, databases, and information retrieval systems.

5.2 Importance of Searching Algorithms

Searching algorithms are essential for finding specific information in a large collection of data. They help in quickly locating the desired element without the need to scan the entire data structure. Searching algorithms are used in various applications, such as:

- **Search Engines**: Search engines use searching algorithms to retrieve relevant information from the web.
- **Databases**: Databases use searching algorithms to search for specific records or data.
- **Information Retrieval Systems**: Information retrieval systems use searching algorithms to retrieve relevant information from a large collection of documents.
- **E-commerce Websites**: E-commerce websites use searching algorithms to search for products based on user queries.
- **Social Media Platforms**: Social media platforms use searching algorithms to search for users, posts, or content.
- **Navigation Systems**: Navigation systems use searching algorithms to find the shortest path or route between two locations.

5.3 Characteristics of Searching Algorithms

Understanding the characteristics of searching in data structures and algorithms is crucial for designing efficient algorithms and making informed decisions about which searching technique to employ. Here, we explore key aspects and characteristics associated with searching:

- **Target Element**: In searching, there is always a specific target element or item that you want to find within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

- **Search Space**: The search space refers to the entire collection of data within which you are looking for the target element. Depending on the data structure used, the search space may vary in size and organization.
- **Search Order**: The search order determines the sequence in which elements are examined during the search process. The order can be linear, binary, or any other specific pattern based on the algorithm used.
- **Search Result**: The search result is the outcome of the search process. It indicates whether the target element was found or not, and may include additional information such as the location of the element in the data structure.
- **Complexity**: Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.

5.4 Types of Searching Algorithms

There are several types of searching algorithms, each with its own approach and complexity. These searching algorithms have advantages and disadvantages based on the type of data structure and the size of the data. Deter

5.4.1 Linear Search

Linear search is a simple searching algorithm that searches for an element in a collection of elements by checking each element one by one until the desired element is found or the end of the collection is reached. It is also known as “sequential search”.

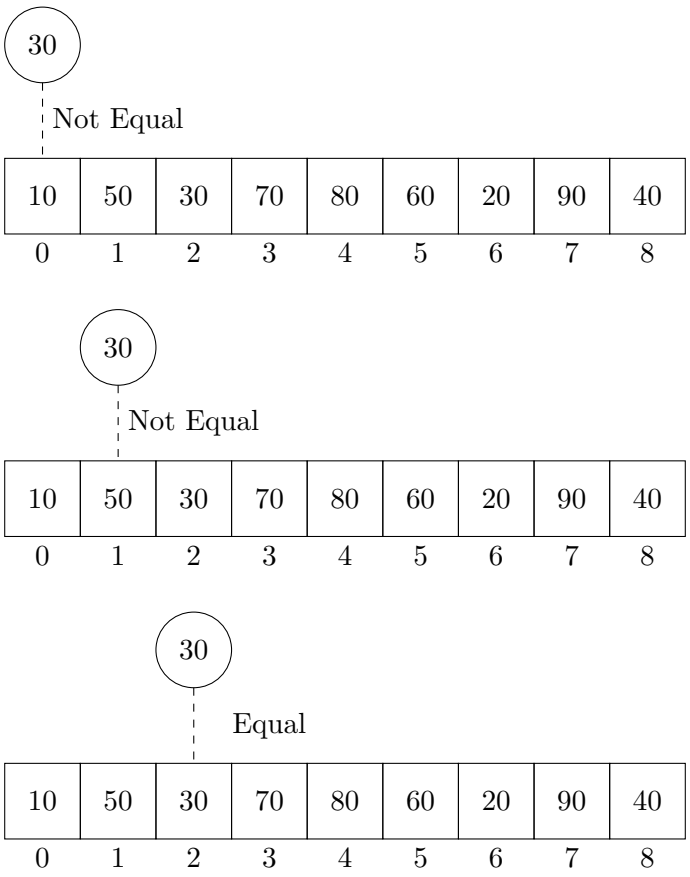


Figure 35: Linear Search

Figure 35 shows the linear search algorithm in action. The algorithm searches for the element 30 in an array of elements. It checks each element one by one until it finds the desired element or reaches the end of the array. In this case, the element 30 is found at index 2.

5.4.1.1 Algorithm of Linear Search

- The Algorithm examines each element, one by one, in the collection, treating each element as a potential match for the key you're searching for.
- The Algorithm examines each element, one by one, in the collection, treating each element as a potential match for the key you're searching for.
- If it goes through all the elements and none of them matches the key, then that means "No match is Found".

5.4.1.2 Pseudo-code of Linear Search

```
LinearSearch(arr, key)
    for i = 0 to arr.length - 1
        if arr[i] == key
            return i
    return -1
```

Code 5.1: Linear Search Pseudo-code

Pseudo-code for Linear Search is shown in the code snippet. The algorithm takes an array and a key as input and searches for the key in the array. If the key is found, it returns the index of the key in the array; otherwise, it returns -1.

5.4.1.3 Linear Search in C++

```
#include <iostream>
#include <vector>
using namespace std;

int linearSearch(vector<int> &arr, int key) {
    int n = arr.size();
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}

int main() {
    vector<int> arr = {10, 50, 30, 70, 80, 60, 20, 90, 40};
    int key = 30;

    int index = linearSearch(arr, key);

    if (index != -1) {
        cout << "Element found at index: " << index << endl;
```

```
    } else {
        cout << "Element not found" << endl;
    }

    return 0;
}
```

Code 5.2: Linear Search in C++

Code 5.2 shows the implementation of the linear search algorithm in C++. The code snippet defines a function `linearSearch` that takes a vector of integers and a key as input and searches for the key in the vector. If the key is found, it returns the index of the key in the vector; otherwise, it returns -1. In the example, the linear search algorithm is used to search for the element 30 in an array of elements. And the element is found at index 2.

5.4.1.4 Application of Linear Search

Linear search can be used in various applications, such as:

- **Unsorted Lists**: Linear search is suitable for searching in unsorted lists where the elements are not in any particular order.
- **Small Data Sets**: Linear search is efficient for small data sets where the overhead of sorting the data is not justified.
- **Linear Data Structures**: Linear search is commonly used in linear data structures like arrays and linked lists as it traverses the elements sequentially.
- **Simple Search Operations**: Linear search is useful for simple search operations where the data is not too large and the search is infrequent.

5.4.1.5 Advantages and Disadvantages of Linear Search

Advantages:

- Suitable for small data sets.
- Works well for unsorted lists.
- Does not require any additional memory.

Disadvantages:

- Inefficient for large data sets as it has a time complexity of O(n) in the worst case.
- Not suitable for sorted lists.
- Slower than other searching algorithms like binary search.

5.4.1.6 Complexity Analysis of Linear Search

The time complexity of linear search is `O(n)`, where `n` is the number of elements in the collection. In the worst-case scenario, the algorithm may have to examine all elements in the collection to find the desired element. The space complexity of linear search is `O(1)`, as it does not require any additional memory.

5.4.2 Binary Search

Binary search is a searching algorithm that finds the position of a target value within a sorted array. It compares the target value to the middle element of the array and eliminates half of the

remaining elements each time. Binary search is more efficient than linear search for large data sets.

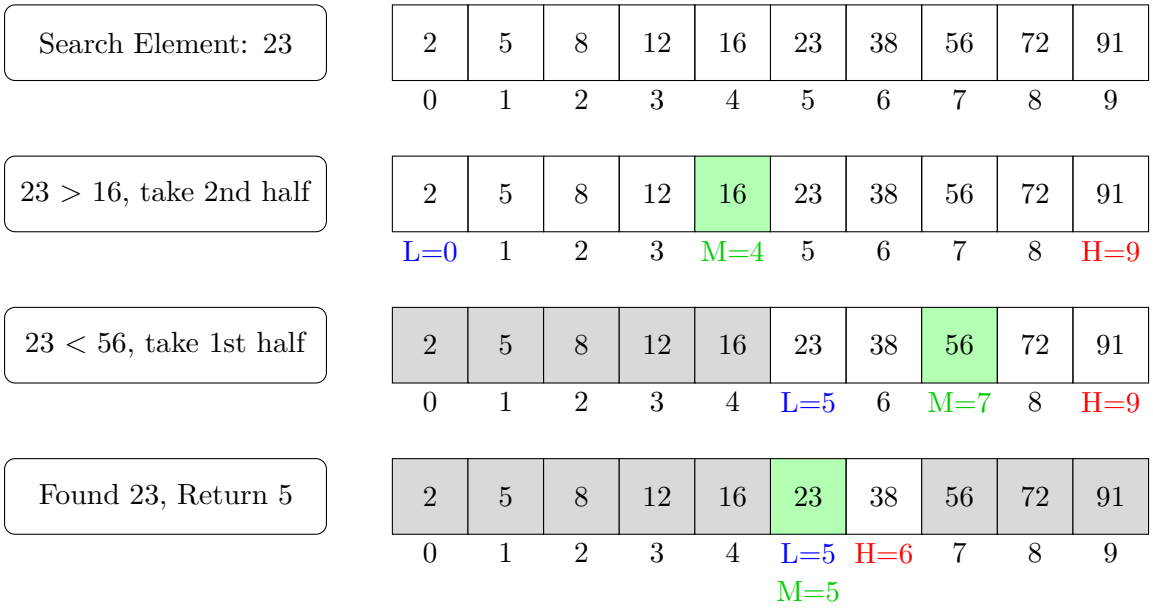


Figure 36: Binary Search

Figure 36 shows the binary search algorithm in action. The algorithm searches for the element 23 in a sorted array of elements. It compares the target value to the middle element of the array and eliminates half of the remaining elements each time. In the 1st iteration, the algorithm compares 23 with 16 and decides to take the 2nd half since $23 > 16$. In the 2nd iteration, the algorithm compares 23 with 56 and decides to take the 1st half since $23 < 56$. In the 3rd iteration, the algorithm finds 23 at index 5 and returns the index.

5.4.2.1 Algorithm of Binary Search

- Divide the search space into two halves by finding the middle index "mid".
- Compare the middle element of the search space with the key.
- Compare the middle element of the search space with the key.
- If the key is not found at middle element, choose which half will be used as the next search space.
  - If the key is not found at middle element, choose which half will be used as the next search space.
  - If the key is not found at middle element, choose which half will be used as the next search space.
- If the key is not found at middle element, choose which half will be used as the next search space.

5.4.2.2 Pseudo-code of Binary Search

```
BinarySearch(arr, key)
    low = 0
    high = arr.length - 1
    while low <= high
        mid = (low + high) / 2
        if arr[mid] == key
```

```
            return mid
        else if arr[mid] < key
            low = mid + 1
        else
            high = mid - 1
    return -1
```

Code 5.3: Binary Search Pseudo-code

Pseudo-code for Binary Search is shown in the code snippet. The algorithm takes a sorted array and a key as input and searches for the key in the array using the binary search technique. If the key is found, it returns the index of the key in the array; otherwise, it returns -1.

5.4.2.3 Binary Search in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(vector<int> &arr, int key) {
    int low = 0;
    int high = arr.size() - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1;
}

int main() {
    vector<int> arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int key = 23;

    int index = binarySearch(arr, key);

    if (index != -1) {
        cout << "Element found at index: " << index << endl;
    } else {
        cout << "Element not found" << endl;
    }

```

```
36     return 0;
37 }
```

Code 5.4: Binary Search in C++

Code 5.4 shows the implementation of the binary search algorithm in C++. The code snippet defines a function `binarySearch` that takes a vector of integers and a key as input and searches for the key in the vector using the binary search technique. If the key is found, it returns the index of the key in the vector; otherwise, it returns -1. In the example, the binary search algorithm is used to search for the element 23 in a sorted array of elements. And the element is found at index 5. During the search process, the algorithm compares the target value to the middle element of the array and eliminates half of the remaining elements each time. If the middle element is greater than the target value, the algorithm chooses the 1st half as the next search space; otherwise, it chooses the 2nd half. For each iteration, the algorithm updates the low and high indices to narrow down the search space until the target value is found or the search space is empty. If the search space is empty, the algorithm returns -1 to indicate that the target value is not present in the array.

5.4.2.4 Application of Binary Search

Binary search can be used in various applications, such as:

- **Building Block**: Binary search can be used as a building block for more complex algorithms used in machine learning, such as algorithms for training neural networks or finding the optimal hyperparameters for a model.
- **Computer Graphics**: It can be used for searching in computer graphics such as algorithms for ray tracing or texture mapping.
- **Database**: It can be used for searching a database.

5.4.2.5 Advantages and Disadvantages of Binary Search

Advantages:

- More efficient than linear search for large data sets.
- More efficient than other searching algorithms with a similar time complexity, such as interpolation search or exponential search.
- Binary search is well-suited for searching large datasets that are stored in external memory, such as on a hard drive or in the cloud.

Disadvantages:

- The array should be sorted.
- Binary search requires that the data structure being searched be stored in contiguous memory locations.
- Binary search requires that the elements of the array be comparable, meaning that they must be able to be ordered.

5.4.2.6 Complexity Analysis of Binary Search

The time complexity of binary search is `O(log n)`, where `n` is the number of elements in the collection. In each iteration, the search space is divided in half, resulting in a time complexity of `O(log n)`. The space complexity of binary search is `O(1)`, as it does not require any additional memory.

5.4.3 Other Searching Algorithms

Apart from linear search and binary search, there are several other searching algorithms that are commonly used in practice. Some of the popular searching algorithms are:

- **Jump Search**: Jump search is an algorithm for finding the position of a target value within a sorted array. It works by jumping ahead by fixed steps and then performing a linear search to find the target value.
- **Interpolation Search**: Interpolation search is an algorithm for finding the position of a target value within a sorted array. It works by estimating the position of the target value based on the values at the ends of the array and then performing a binary search to find the target value.
- **Exponential Search**: Exponential search is an algorithm for finding the position of a target value within a sorted array. It works by doubling the size of the search space until the target value is found, and then performing a binary search to find the target value.
- **Fibonacci Search**: Fibonacci search is an algorithm for finding the position of a target value within a sorted array. It works by dividing the search space into two parts using the Fibonacci numbers and then performing a binary search to find the target value.
- **Hash Search**: Hash search is an algorithm for finding the position of a target value within a hash table. It works by using a hash function to map the target value to a key and then looking up the key in the hash table to find the target value.

5.5 Summary

In this chapter, we discussed the searching algorithms, especially the linear search and binary search algorithms. We explained the working principles of these algorithms, provided pseudo-code examples, and implemented them in C++. We also discussed the applications, advantages, disadvantages, and complexity analysis of these algorithms. Additionally, we introduced other searching algorithms such as jump search, interpolation search, exponential search, Fibonacci search, and hash search. These searching algorithms play a crucial role in various applications and are essential building blocks for more complex algorithms used in computer science and related fields.

6

Sorting Algorithms

6.1 Introduction

Imagine you have a messy pile of socks, and you want to organize them by color. That's exactly what a sorting algorithm does, but with data! Sorting algorithms are fundamental techniques used to arrange the elements of a data structure in a specific order, such as ascending or descending. These algorithms are crucial in computer science because they optimize the efficiency of other algorithms that require sorted data, like search algorithms. Whether it's Bubble Sort gently swapping elements like a careful librarian, or Quick Sort slicing through data like a ninja chef, these algorithms make sure everything is in the right place. And the best part? Most programming languages come with these sorting Algorithms built-in, so you don't have to reinvent the wheel every time you need to sort something.

6.2 Importance of Sorting Algorithms

Sorting algorithms are essential in computer science and programming as they are used to arrange data in a specific order. There is a wide range of applications for these algorithms, including searching algorithms, database algorithms, divide and conquer methods, and data structure algorithms. Here are some of the key reasons why sorting algorithms are important:

- **Efficient Searching**: Sorting algorithms are used to arrange data in a specific order, making it easier and faster to search for elements in the data structure. For example, binary search requires the data to be sorted before it can be applied.
- **Data Analysis**: Sorting algorithms are used in data analysis to organize and analyze large datasets. By sorting the data, patterns and trends can be identified more easily.
- **Database Management**: Sorting algorithms are used in database management systems to sort records and improve the efficiency of queries and data retrieval.
- **Optimization**: Sorting algorithms are used in optimization problems to arrange elements in a specific order that minimizes or maximizes a certain criterion.
- **Data Structure Operations**: Sorting algorithms are used in various data structure operations, such as merging two sorted arrays, finding the median of a dataset, and removing duplicates from a list.
- **Machine Learning**: Sorting algorithms are used in machine learning algorithms to preprocess and organize data before training models. Sorting data can improve the performance of machine learning algorithms and make them more efficient.
- **Operating Systems**: Sorting algorithms are used in operating systems to manage and organize files and directories. By sorting files and directories, the user can easily locate

and access the required information.

6.3 Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based Sorting Algorithms**: In comparison-based sorting algorithms, elements are compared with each other to determine their relative order. The most common comparison-based sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, and Shell Sort.

2. **Non-comparison-based Sorting Algorithms**: In non-comparison-based sorting algorithms, elements are not compared with each other directly. Instead, these algorithms use specific properties of the elements to determine their order. The most common non-comparison-based sorting

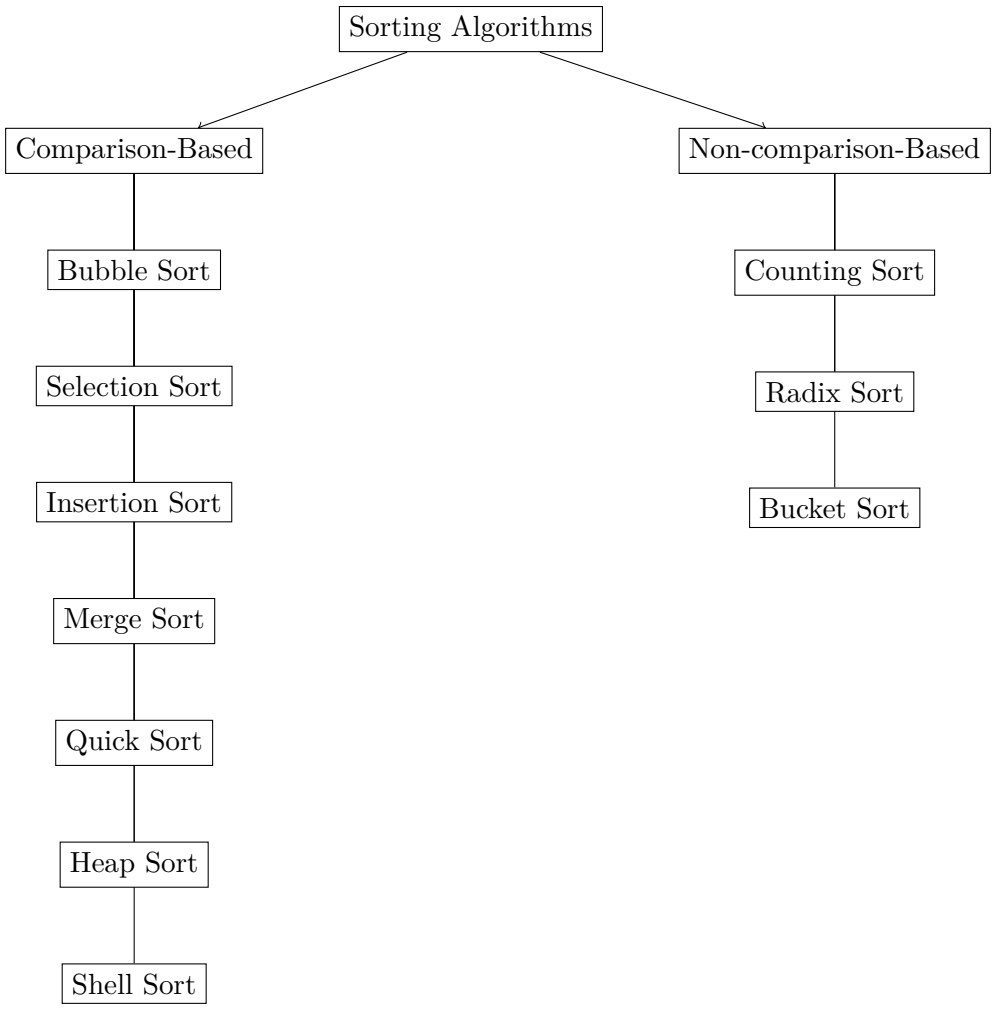


Figure 37: Sorting Techniques

6.4 Types of Comparison-Based Sorting Algorithms

There are various types of sorting algorithms used in computer science and programming. These sorting algorithms can be broadly classified into two categories: comparison-based sorting

algorithms and non-comparison-based sorting algorithms.

6.4.1 Selection Sort

Selection Sort is a simple sorting algorithm that works by repeatedly selecting the minimum element from the unsorted portion of the array and swapping it with the first unsorted element. The algorithm divides the array into two parts: the sorted part at the beginning and the unsorted part at the end. In each iteration, the algorithm finds the minimum element in the unsorted part and swaps it with the first unsorted element. This process continues until the entire array is sorted.

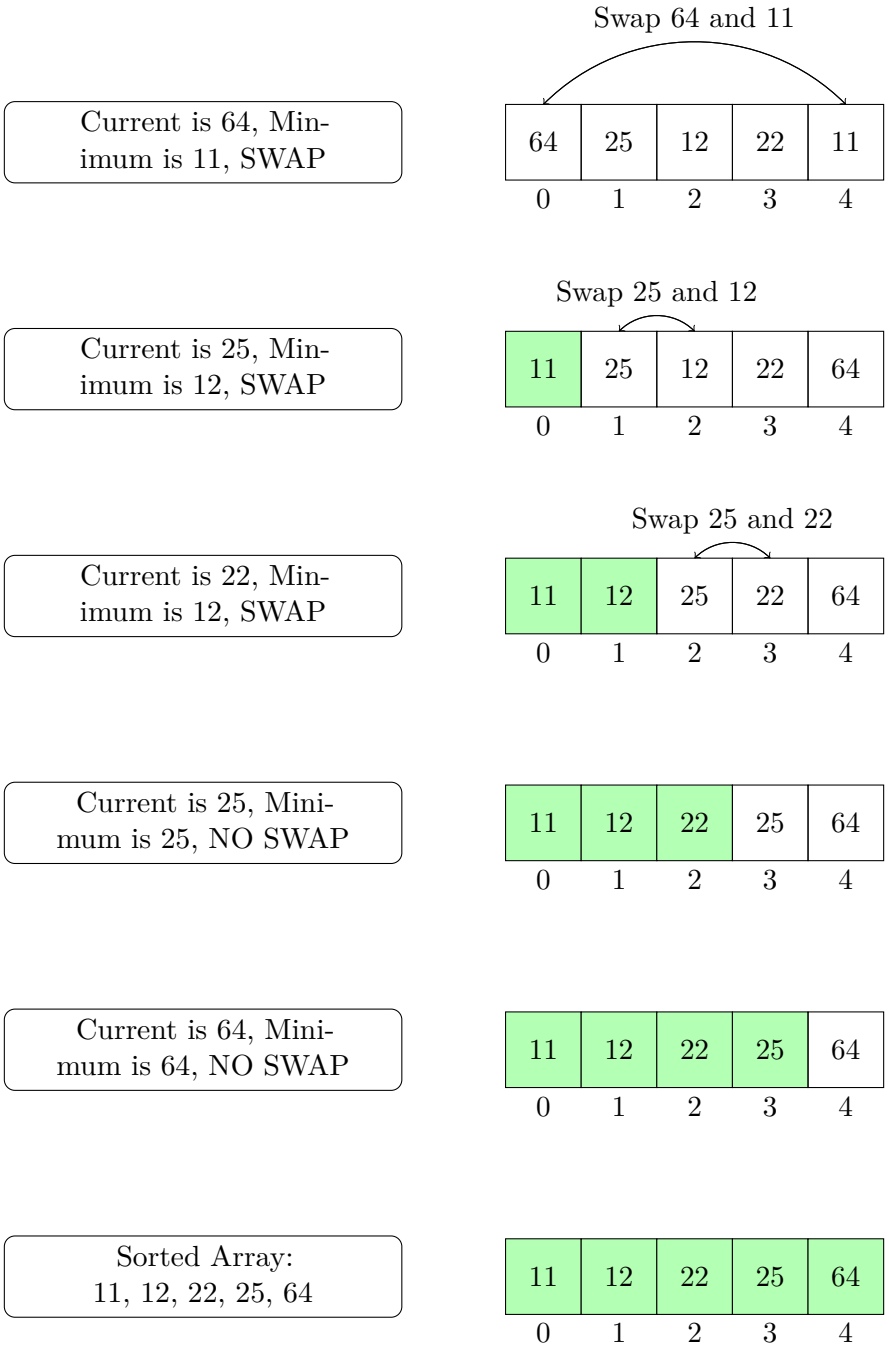


Figure 38: Selection Sort

Figure 38 shows the Selection Sort algorithm in action. The algorithm sorts an array of

elements by repeatedly selecting the minimum element from the unsorted portion of the array and swapping it with the first unsorted element. In each iteration, the algorithm finds the minimum element in the unsorted part and swaps it with the first unsorted element. This process continues until the entire array is sorted.

6.4.1.1 Algorithm of Selection Sort

- Divide the array into two parts: the sorted part at the beginning and the unsorted part at the end.
- Find the minimum element in the unsorted part of the array.
- Swap the minimum element with the first unsorted element.
- Repeat the process for the remaining unsorted elements until the entire array is sorted.

6.4.1.2 Pseudo-code of Selection Sort

```
SelectionSort(arr)
    n = arr.length
    for i = 0 to n-1
        minIndex = i
        for j = i+1 to n
            if arr[j] < arr[minIndex]
                minIndex = j
        swap(arr[i], arr[minIndex])
```

Code 6.1: Selection Sort Pseudo-code

Pseudo-code for Selection Sort is shown in the code snippet. The algorithm takes an array of elements as input and sorts the array using the Selection Sort technique. The algorithm iterates through the array, finds the minimum element in the unsorted part, and swaps it with the first unsorted element. This process continues until the entire array is sorted.

6.4.1.3 Selection Sort in C++

```
#include <iostream>
#include <vector>
using namespace std;

void selectionSort(vector<int> &arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        swap(arr[i], arr[minIndex]);
    }
```

```
}

void printArray(vector<int> &arr) {
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {64, 25, 12, 22, 11};

    cout << "Original Array: ";
    printArray(arr);

    selectionSort(arr);

    cout << "Sorted Array: ";
    printArray(arr);

    return 0;
}
```

Code 6.2: Selection Sort in C++

Code 6.2 shows the implementation of the Selection Sort algorithm in C++. The code snippet defines a function `selectionSort` that takes a vector of integers as input and sorts the array using the Selection Sort technique. The algorithm iterates through the array, finds the minimum element in the unsorted part, and swaps it with the first unsorted element. This process continues until the entire array is sorted. In the example, the Selection Sort algorithm is used to sort an array of elements, and the sorted array is printed to the console.

6.4.1.4 Advantages and Disadvantages of Selection Sort

Advantages:

- Easy to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant $O(1)$ extra memory space.
- Requires less number of swaps (or memory writes) compared to many other standard algorithms. Only cycle sort beats it in terms of memory writes. Therefore, it can be a simple algorithm choice when memory writes are costly.

Disadvantages:

- Selection sort has a time complexity of $O(n^2)$ makes it slower compared to algorithms like Quick Sort or Merge Sort.
- Does not maintain the relative order of equal elements.
- Does not preserve the relative order of items with equal keys which means it is not stable.

6.4.1.5 Complexity Analysis of Selection Sort

The time complexity of Selection Sort is $O(n^2)$, where n is the number of elements in the array. The space complexity of Selection Sort is $O(1)$, as it does not require any additional memory.

6.4.2 Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. For each pass through the list, the largest element goes to the end of the list. If a larger element is found, it is swapped with the next element, and the process is repeated until the list is sorted.

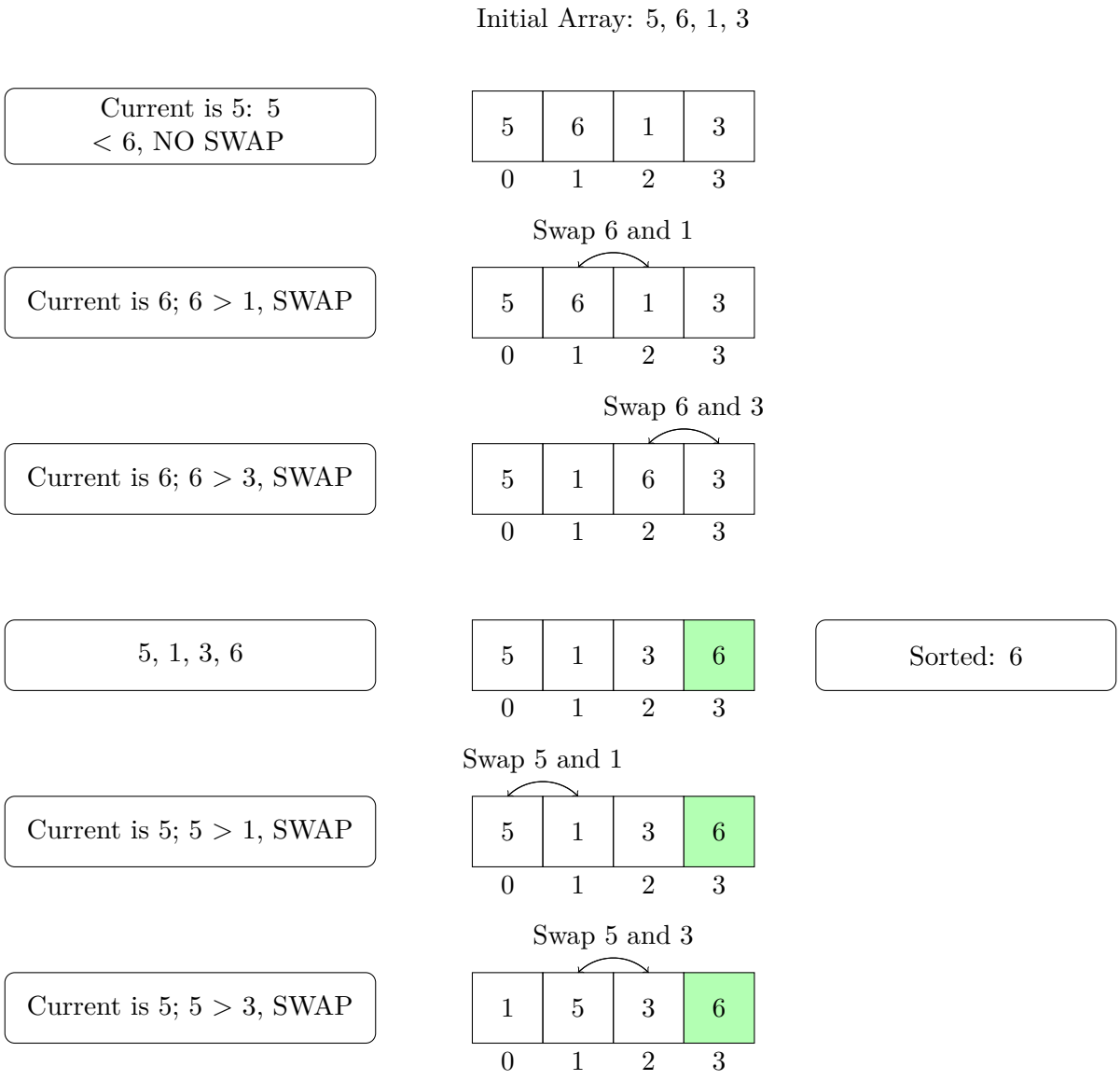


Figure 39 shows the Bubble Sort algorithm in action. The algorithm sorts an array of elements by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. In each pass through the list, the largest element goes to the end of the list. The process is repeated until the entire array is sorted.

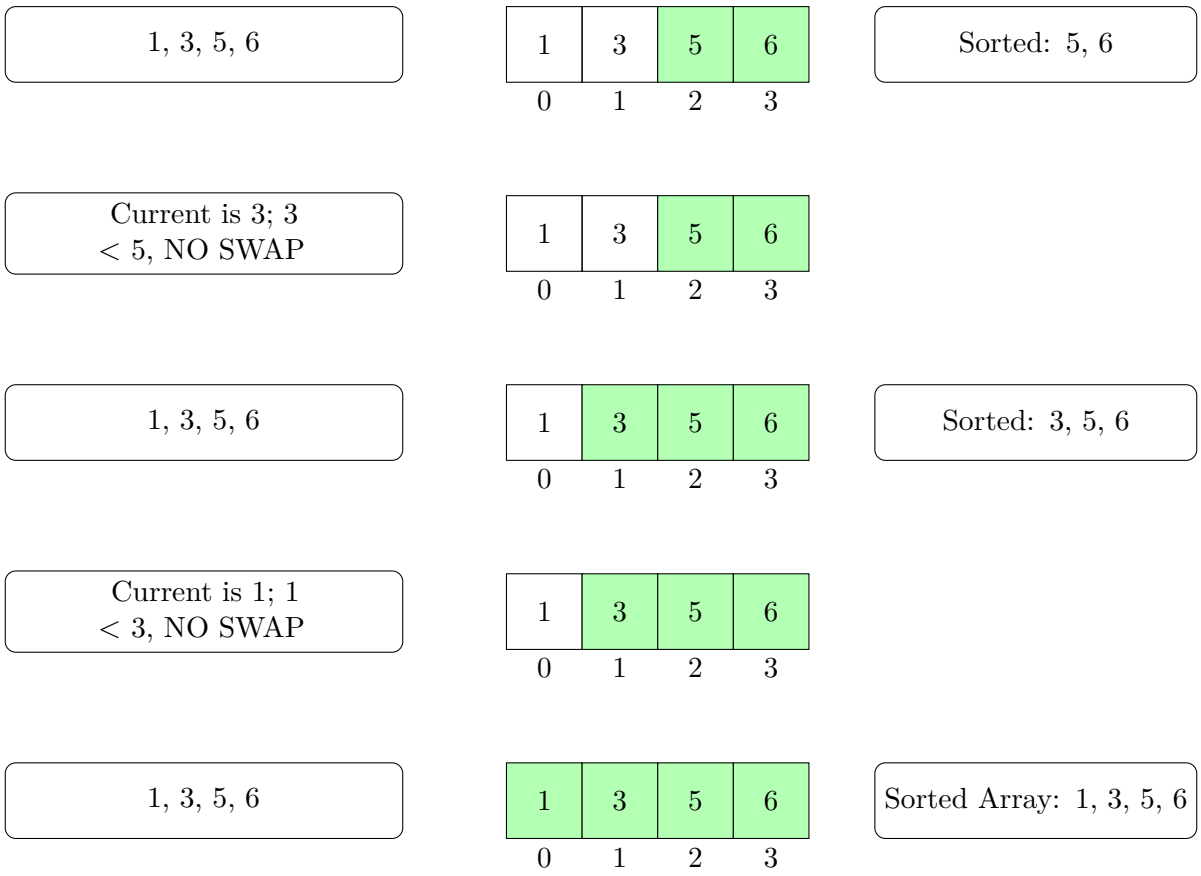


Figure 39: Bubble Sort

6.4.2.1 Algorithm of Bubble Sort

- Start at the beginning of the list and compare adjacent elements.
- If the elements are in the wrong order, swap them.
- Repeat the process for each pair of adjacent elements until the entire list is sorted.
- Continue the process until no more swaps are needed.

6.4.2.2 Pseudo-code of Bubble Sort

```
BubbleSort(arr)
    n = arr.length
    for i = 0 to n-1
        for j = 0 to n-i-1
            if arr[j] > arr[j+1]
                swap(arr[j], arr[j+1])
```

Code 6.3: Bubble Sort Pseudo-code

Pseudo-code for Bubble Sort is shown in the code snippet. The algorithm takes an array of elements as input and sorts the array using the Bubble Sort technique. The algorithm iterates through the array, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the entire array is sorted.

6.4.2.3 Bubble Sort in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

void bubbleSort(vector<int> &arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
}

void printArray(vector<int> &arr) {
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {5, 6, 1, 3};

    cout << "Original Array: ";
    printArray(arr);

    bubbleSort(arr);

    cout << "Sorted Array: ";
    printArray(arr);

    return 0;
}
```

Code 6.4: Bubble Sort in C++

Code 6.4 shows the implementation of the Bubble Sort algorithm in C++. The code snippet defines a function `bubbleSort` that takes a vector of integers as input and sorts the array using the Bubble Sort technique. The algorithm iterates through the array, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the entire array is sorted. In the example, the Bubble Sort algorithm is used to sort an array of elements, and the sorted array is printed to the console.

6.4.2.4 Advantages and Disadvantages of Bubble Sort

Advantages:

- Simple to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant $O(1)$ extra memory space.
- Stable sorting algorithm that preserves the relative order of equal elements.

Disadvantages:

- Bubble Sort has a time complexity of $O(n^2)$, making it slower compared to more efficient algorithms like Quick Sort or Merge Sort.
- Inefficient for large lists due to its quadratic time complexity.
- Not suitable for large datasets or real-world applications where performance is critical.

6.4.2.5 Complexity Analysis of Bubble Sort

The time complexity of Bubble Sort is $O(n^2)$, where n is the number of elements in the array. The space complexity of Bubble Sort is $O(1)$, as it does not require any additional memory.

6.4.3 Insertion Sort

Insertion Sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

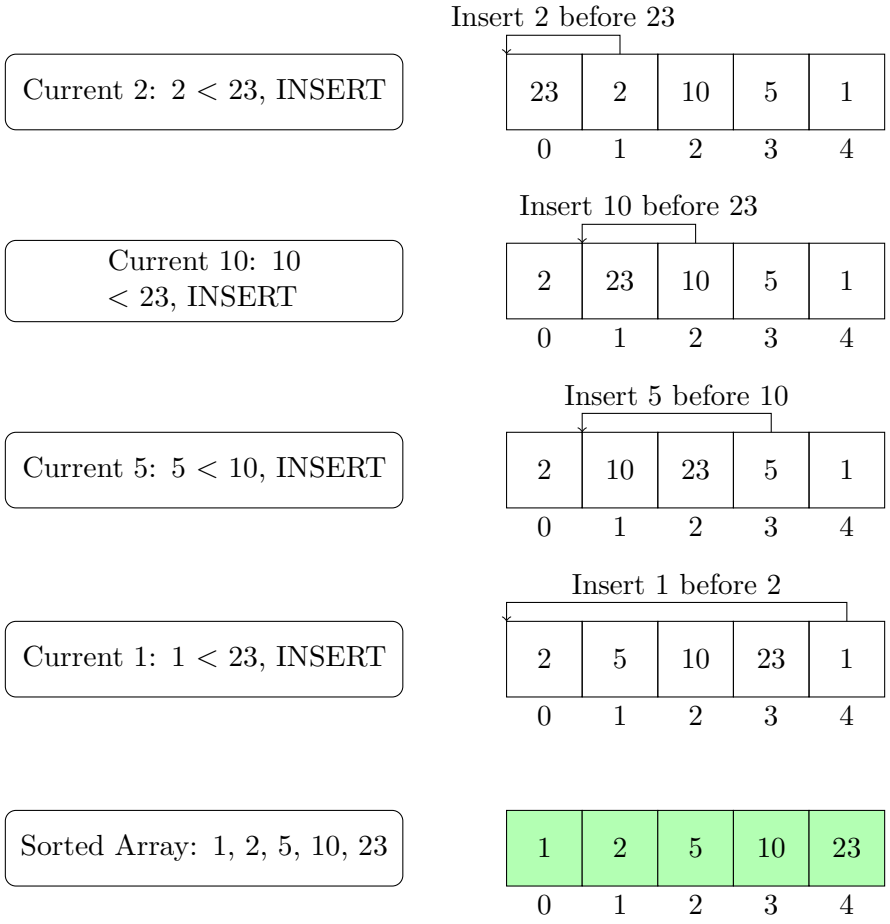


Figure 40: Insertion Sort

Figure 40 shows the Insertion Sort algorithm in action. The algorithm sorts an array of

elements by iteratively inserting each element into its correct position in a sorted portion of the list. In each iteration, the algorithm picks an element from the unsorted group and inserts it into the correct position in the sorted group. This process continues until the entire array is sorted.

6.4.3.1 Algorithm of Insertion Sort

- Start with the first element of the list as the sorted group.
- Pick the next element from the unsorted group and insert it into the correct position in the sorted group.
- Repeat the process for each element in the unsorted group until the entire list is sorted.

6.4.3.2 Pseudo-code of Insertion Sort

```
InsertionSort(arr)
    n = arr.length
    for i = 1 to n-1
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key
            arr[j+1] = arr[j]
            j = j - 1
        arr[j+1] = key
```

Code 6.5: Insertion Sort Pseudo-code

Pseudo-code for Insertion Sort is shown in the code snippet. The algorithm takes an array of elements as input and sorts the array using the Insertion Sort technique. The algorithm iterates through the array, picks an element from the unsorted group, and inserts it into the correct position in the sorted group. This process continues until the entire array is sorted.

6.4.3.3 Insertion Sort in C++

```
#include <iostream>
#include <vector>
using namespace std;

void insertionSort(vector<int> &arr) {
    int n = arr.size();

    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key;
    }
}
```

```
20
21 void printArray(vector<int> &arr) {
22     for (int i = 0; i < arr.size(); i++) {
23         cout << arr[i] << " ";
24     }
25     cout << endl;
26 }
27
28 int main() {
29     vector<int> arr = {23, 2, 10, 5, 1};
30
31     cout << "Original Array: ";
32     printArray(arr);
33
34     insertionSort(arr);
35
36     cout << "Sorted Array: ";
37     printArray(arr);
38
39     return 0;
40 }
```

Code 6.6: Insertion Sort in C++

Code 6.6 shows the implementation of the Insertion Sort algorithm in C++. The code snippet defines a function `insertionSort` that takes a vector of integers as input and sorts the array using the Insertion Sort technique. The algorithm iterates through the array, picks an element from the unsorted group, and inserts it into the correct position in the sorted group. This process continues until the entire array is sorted. In the example, the Insertion Sort algorithm is used to sort an array of elements, and the sorted array is printed to the console.

6.4.3.4 Applications of Insertion Sort

- Insertion Sort is suitable for small lists or nearly sorted lists where simplicity and stability are important.
- It is used as a subroutine in other sorting algorithms like Bucket Sort.
- Insertion Sort can be useful when the array is already almost sorted with very few inversions.
- It is used in hybrid sorting algorithms along with other efficient algorithms like Quick Sort and Merge Sort. When the subarray size becomes small, we switch to Insertion Sort in these recursive algorithms. For example, IntroSort and TimSort use Insertion Sort.

6.4.3.5 Advantages and Disadvantages of Insertion Sort

Advantages:

- Simple and easy to implement.
- Stable sorting algorithm.
- Efficient for small lists and nearly sorted lists.
- Space-efficient as it is an in-place algorithm.
- Adoptive. The number of inversions is directly proportional to the number of swaps. For example, no swapping happens for a sorted array, and it takes $O(n)$ time only.

Disadvantages:

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., Merge Sort, Quick Sort) for most cases.

6.4.3.6 Complexity Analysis of Insertion Sort

The time complexity of Insertion Sort is $O(n^2)$ in the worst-case scenario, where n is the number of elements in the array. The space complexity of Insertion Sort is $O(1)$, as it does not require any additional memory.

6.4.4 Merge Sort

Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts the two halves, and then merges the sorted halves. It is a stable sorting algorithm that is efficient for large datasets. Merge Sort is a comparison-based sorting algorithm that has a time complexity of $O(n \log n)$.

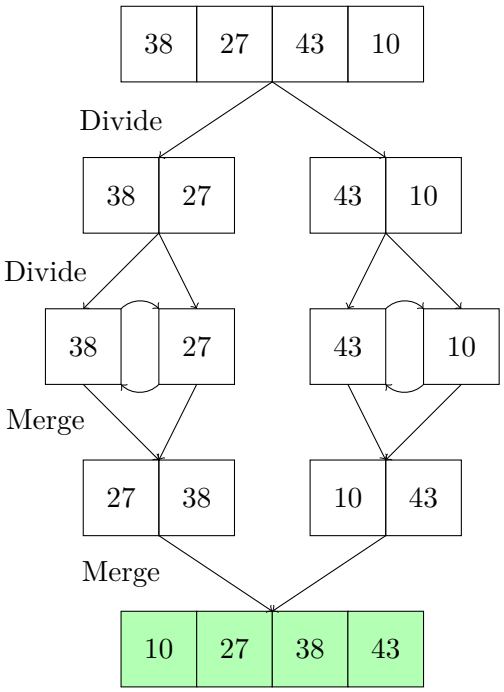


Figure 41: Merge Sort

Figure 41 shows the Merge Sort algorithm in action. The algorithm sorts an array of elements by dividing the array into two halves, recursively sorting the two halves, and then merging the sorted halves. In each step, the algorithm divides the array into smaller subarrays until each subarray contains only one element. It then merges the subarrays in a sorted order to produce the final sorted array.

6.4.4.1 Algorithm of Merge Sort

- Divide the input array into two halves.
- Recursively sort the two halves.
- Merge the sorted halves to produce the final sorted array.

6.4.4.2 Pseudo-code of Merge Sort

```
Merge(arr, left, mid, right)
    n1 = mid - left + 1
    n2 = right - mid

    L[1..n1], R[1..n2]

    for i = 0 to n1
        L[i] = arr[left + i]

    for j = 0 to n2
        R[j] = arr[mid + 1 + j]

    i = 0, j = 0, k = left

    while i < n1 and j < n2
        if L[i] <= R[j]
            arr[k] = L[i]
            i++
        else
            arr[k] = R[j]
            j++
        k++

    while i < n1
        arr[k] = L[i]
        i++
        k++

    while j < n2
        arr[k] = R[j]
        j++
        k++

MergeSort(arr, left, right)
    if left < right
        mid = left + (right - left) / 2

        MergeSort(arr, left, mid)
        MergeSort(arr, mid + 1, right)

        Merge(arr, left, mid, right)
```

Code 6.7: Merge Sort Pseudo-code

Pseudo-code for Merge Sort is shown in the code snippet. The algorithm takes an array of elements as input and sorts the array using the Merge Sort technique. The algorithm divides the array into two halves, recursively sorts the two halves, and then merges the sorted halves to produce the final sorted array.

6.4.4.3 Merge Sort in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

void merge(vector<int> &arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++) {
        L[i] = arr[left + i];
    }

    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
```

```
51
52          merge(arr, left, mid, right);
53      }
54  }
55
56  void printArray(vector<int> &arr) {
57      for (int i = 0; i < arr.size(); i++) {
58          cout << arr[i] << " ";
59      }
60      cout << endl;
61  }
62
63  int main() {
64      vector<int> arr = {38, 27, 43, 10};
65
66      cout << "Original Array: ";
67      printArray(arr);
68
69      mergeSort(arr, 0, arr.size() - 1);
70
71      cout << "Sorted Array: ";
72      printArray(arr);
73
74      return 0;
75  }
```

Code 6.8: Merge Sort in C++

Code 6.8 shows the implementation of the Merge Sort algorithm in C++. The code snippet defines a function `mergeSort` that takes a vector of integers as input and sorts the array using the Merge Sort technique. The algorithm divides the array into two halves, recursively sorts the two halves, and then merges the sorted halves to produce the final sorted array. In the example, the Merge Sort algorithm is used to sort an array of elements, and the sorted array is printed to the console.

6.4.4.4 Applications of Merge Sort

- Merge Sort is suitable for sorting large datasets efficiently.
- It is used in external sorting when the dataset is too large to fit in memory.
- Merge Sort is used in inversion counting, where the number of inversions in an array is calculated.
- Merge Sort and its variations are used in library methods of programming languages. For example, its variation TimSort is used in Python, Java, Android, and Swift.
- It is a preferred algorithm for sorting linked lists due to its efficient use of memory.
- Merge Sort can be easily parallelized as subarrays can be independently sorted and then merged.
- The merge function of Merge Sort can be used to efficiently solve problems like the union and intersection of two sorted arrays.

6.4.4.5 Advantages and Disadvantages of Merge Sort

Advantages:

- Stable sorting algorithm that maintains the relative order of equal elements.
- Guaranteed worst-case performance of $O(n \log n)$, making it efficient for large datasets.
- Simple to implement using the divide-and-conquer approach.
- Naturally parallelizable, as subarrays can be independently sorted and then merged.

Disadvantages:

- Requires additional memory for storing the merged subarrays, leading to a space complexity of $O(n)$.
- Not an in-place sorting algorithm, which can be a disadvantage in memory-constrained applications.
- Slower than Quick Sort in general due to additional memory requirements and slower cache performance.

6.4.4.6 Complexity Analysis of Merge Sort

The time complexity of Merge Sort is $O(n \log n)$ in the worst-case scenario, where n is the number of elements in the array. The space complexity of Merge Sort is $O(n)$ due to the additional memory required for merging the subarrays.

6.4.5 Quick Sort

Quick Sort is a comparison-based sorting algorithm that uses a divide-and-conquer strategy to sort an array of elements. It is one of the most efficient sorting algorithms with an average-case time complexity of $O(n \log n)$. Quick Sort works by selecting a pivot element from the array, partitioning the array into two subarrays based on the pivot, and recursively sorting the subarrays.

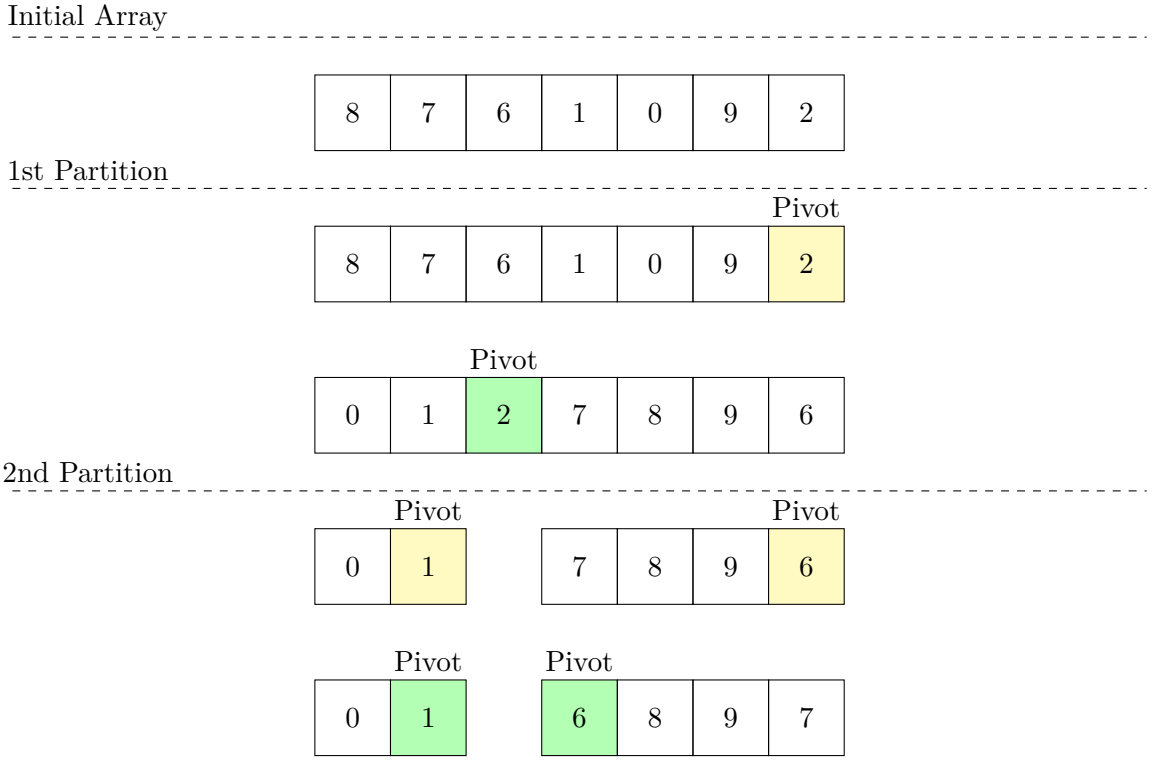


Figure 42: Quick Sort

Figure 42 shows the Quick Sort algorithm in action. The algorithm sorts an array of elements by selecting a pivot element, partitioning the array into two subarrays based on the pivot, and recursively sorting the subarrays. In each step, the algorithm selects a pivot element, partitions the array into smaller subarrays, and then repeats the process until the entire array is sorted.

6.4.5.1 Algorithm of Quick Sort

- Select a pivot element from the array.
- Partition the array into two subarrays: elements less than the pivot and elements greater than the pivot.
- Recursively sort the two subarrays.

6.4.5.2 Pseudo-code of Quick Sort

```
Partition(arr, low, high)
    pivot = arr[high]
    i = low - 1

    for j = low to high - 1
```

```
        if arr[j] < pivot
            i++
            swap arr[i] and arr[j]

    swap arr[i + 1] and arr[high]
    return i + 1

QuickSort(arr, low, high)
    if low < high
        pi = Partition(arr, low, high)

        QuickSort(arr, low, pi - 1)
        QuickSort(arr, pi + 1, high)
```

Code 6.9: Quick Sort Pseudo-code

Pseudo-code for Quick Sort is shown in the code snippet. The algorithm takes an array of elements as input and sorts the array using the Quick Sort technique. The algorithm selects a pivot element from the array, partitions the array into two subarrays based on the pivot, and recursively sorts the subarrays.

6.4.5.3 Quick Sort in C++

```
#include <iostream>
#include <vector>
using namespace std;

int partition(vector<int> &arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quickSort(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

```cpp

void printArray(vector<int> &arr) {
    for (int i = 0; i < arr.size(); i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    vector<int> arr = {8, 7, 6, 1, 0, 9, 2};

    cout << "Original Array: ";
    printArray(arr);

    quickSort(arr, 0, arr.size() - 1);

    cout << "Sorted Array: ";
    printArray(arr);

    return 0;
}
```

Code 6.10: Quick Sort in C++

Code 6.10 shows the implementation of the Quick Sort algorithm in C++. The code snippet defines a function `quickSort` that takes a vector of integers as input and sorts the array using the Quick Sort technique. The algorithm selects a pivot element from the array, partitions the array into two subarrays based on the pivot, and recursively sorts the subarrays. In the example, the Quick Sort algorithm is used to sort an array of elements, and the sorted array is printed to the console.

6.4.5.4 Applications of Quick Sort

- Quick Sort is efficient for sorting large datasets with an average-case time complexity of $O(n \log n)$.
- It is used in partitioning problems like finding the kth smallest element or dividing arrays by a pivot.
- Quick Sort is integral to randomized algorithms, offering better performance than deterministic approaches.
- It is applied in cryptography for generating random permutations and unpredictable encryption keys.
- The partitioning step of Quick Sort can be parallelized for improved performance in multi-core or distributed systems.
- Quick Sort is important in theoretical computer science for analyzing average-case complexity and developing new techniques.

6.4.5.5 Advantages and Disadvantages of Quick Sort

Advantages:

- Divide-and-conquer algorithm that simplifies problem-solving.
- Efficient on large datasets with an average-case time complexity of $O(n \log n)$.

- Low overhead and memory requirements, making it suitable for memory-constrained applications.
- Cache-friendly algorithm that operates on the same array to sort without copying data to auxiliary arrays.
- Fastest general-purpose algorithm for large datasets when stability is not required.
- Tail-recursive, allowing tail call optimization for improved performance.

Disadvantages:

- Worst-case time complexity of $O(n^2)$ when the pivot is chosen poorly.
- Not suitable for small datasets due to overhead and partitioning costs.
- Not a stable sort, meaning that the relative order of equal elements may not be preserved in the sorted output.

6.4.5.6 Complexity Analysis of Quick Sort

The worst-case time complexity of Quick Sort is $O(n^2)$, which occurs when the pivot element is chosen poorly. However, the average-case time complexity of Quick Sort is $O(n \log n)$, making it efficient for large datasets. The space complexity of Quick Sort is $O(\log n)$ due to the recursive calls and partitioning steps.

6.4.6 Heap Sort

6.5 Types of Non-comparison-based Sorting Algorithms

6.5.1 Counting Sort

6.5.2 Radix Sort

6.6 Summary

7

Hashing

7.1 Introduction

7.2 Hash Table

7.3 Hash Function

7.4 Collision Resolution Techniques

7.4.1 Separate Chaining

7.4.2 Open Addressing

7.4.2.1 Linear Probing

7.4.2.2 Quadratic Probing

7.4.2.3 Double Hashing

7.5 Complexity Analysis of Hashing

7.6 Summary

8

Advanced Data Structures and Algorithms

8.1 Introduction

8.2 Advanced Data Structures

8.2.1 Segment Tree

8.2.2 Fenwick Tree

8.2.3 Suffix Tree

8.2.4 Suffix Array

8.2.5 Trie

8.2.6 Heap

8.2.7 Disjoint Set

8.2.8 Skip List

8.2.9 Splay Tree

8.2.10 Bloom Filter

8.2.11 KD Tree

8.2.12 Quad Tree

8.2.13 Octree

8.2.14 B-Tree

8.2.15 B+ Tree

8.2.16 R-Tree

8.2.17 X-Tree

8.2.18 Y-Tree

8.2.19 Z-Tree

8.3 Advanced Algorithms

8.3.1 Dynamic Programming

8.3.2 Greedy Algorithms

9

Applications of Data Structures and Algorithms

9.1 Applications in Computer Science

9.1.1 Operating Systems

9.1.2 Database Management Systems

9.1.3 Compiler Design

9.1.4 Networking

9.1.5 Artificial Intelligence

9.1.6 Machine Learning

9.1.7 Computer Graphics

9.1.8 Computer Vision

9.1.9 Robotics

9.1.10 Web Development

9.1.11 Mobile Development

9.1.12 Game Development

9.1.13 Cybersecurity

9.1.14 Quantum Computing

9.2 Applications in Real Life

9.2.1 Social Media

9.2.2 E-commerce

9.2.3 Healthcare

9.2.4 Finance

9.2.5 Transportation

9.2.6 Education

9.2.7 Agriculture

9.2.8 Manufacturing

9.2.9 Entertainment

10

References

A. Books

- Vishwas R. (2023). Data Structure Handbook. Dr. Vishwas Raval. ISBN: 978-9359063591
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. ISBN: 978-0262046305
- Erickson, J. (2019). Algorithms. ISBN: 978-1792644832

B. Other Sources

- Tutorialspoint. (n.d.). Data Structures Basics. Data Structure Basics. https://www.tutorialspoint.com/data_structures_algorithms/data_structures_basics.htm
- Algorithm Archive · Arcane Algorithm Archive. (n.d.). https://www.algorithm-archive.org/