

Data Structures and Algorithms¹

A Study Guide for Students of Sorsogon State University - Bulan Campus²

JARRIAN VINCE G. GOJAR³

September 25, 2024

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

²This 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	ix
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	43
3.2	Stacks	43
3.2.1	Operations on Stacks	43
3.2.1.1	Push	43
3.2.1.2	Pop	43
3.2.1.3	Peek	43
3.2.1.4	isEmpty	43
3.2.1.5	isFull	43
3.2.2	Complexity Analysis of Stacks	43
3.2.3	Implementation of Stacks Using Arrays	43
3.2.4	Implementation of Stacks Using Linked Lists	43
3.3	Queues	43
3.3.1	Types of Queues	43
3.3.1.1	Linear Queue	43
3.3.1.2	Circular Queue	43
3.3.1.3	Priority Queue	43
3.3.1.4	Double-ended Queue (Deque)	43
3.3.2	Operations on Queues	43
3.3.2.1	Enqueue	43
3.3.2.2	Dequeue	43
3.3.2.3	Front	43
3.3.2.4	Rear	43
3.3.3	Complexity Analysis of Queues	43
3.3.4	Implementation of Queues Using Arrays	43
3.3.5	Implementation of Queues Using Linked Lists	43
3.4	Comparison of Stacks and Queues	43
3.5	Summary	43
4	Trees	44
4.1	Introduction	45
4.2	Properties of Trees	45
4.2.1	Root Node	45
4.2.2	Parent Node	45
4.2.3	Child Node	45
4.2.4	Leaf Node	45
4.2.5	Ancestors	45
4.2.6	Siblings	45
4.2.7	Descendants	45
4.2.8	Height of a Tree	45
4.2.9	Depth of a Node	45
4.2.10	Degree of a Node	45
4.2.11	Level of a Node	45
4.2.12	Subtree	45

4.3	Types of Trees	45
4.3.1	Binary Tree	45
4.3.1.1	Types of Binary Trees	45
4.3.1.1.1	Left-skewed Binary Tree	45
4.3.1.1.2	Right-skewed Binary Tree	45
4.3.1.1.3	Complete Binary Tree	45
4.3.2	Ternary Tree	45
4.3.3	N-ary Tree	45
4.3.4	Binary Search Tree	45
4.3.5	AVL Tree	45
4.3.6	Red-Black Tree	45
4.4	Basic Operations on Trees	45
4.4.1	Creation of a Tree	45
4.4.2	Insertion	45
4.4.3	Deletion	45
4.4.4	Searching	45
4.4.5	Traversal	45
4.4.5.1	Preorder Traversal	45
4.4.5.2	Inorder Traversal	45
4.4.5.3	Postorder Traversal	45
4.4.5.4	Level-order Traversal	45
4.5	Complexity Analysis of Trees	45
4.6	Summary	45
5	Graphs	46
5.1	Introduction	47
5.2	Properties of Graphs	47
5.2.1	Vertex	47
5.2.2	Edge	47
5.2.3	Degree of a Vertex	47
5.2.4	Path	47
5.3	Types of Graphs	47
5.3.1	Finite Graph	47
5.3.2	Infinite Graph	47
5.3.3	Trivial Graph	47
5.3.4	Simple Graph	47
5.3.5	Multi Graph	47
5.3.6	Null Graph	47
5.3.7	Complete Graph	47
5.3.8	Pseudo Graph	47
5.3.9	Regular Graph	47
5.3.10	Bipartite Graph	47
5.3.11	Labelled Graph	47
5.3.12	Weighted Graph	47
5.3.13	Directed Graph	47
5.3.14	Undirected Graph	47
5.3.15	Connected Graph	47
5.3.16	Disconnected Graph	47
5.3.17	Cyclic Graph	47
5.3.18	Acyclic Graph	47
5.3.19	Directed Acyclic Graph (DAG)	47

5.3.20	Digraph	47
5.3.21	Subgraph	47
5.4	Operations on Graphs	47
5.4.1	Creation of a Graph	47
5.4.2	Insertion	47
5.4.2.1	Insertion of a Vertex	47
5.4.2.2	Insertion of an Edge	47
5.4.3	Deletion	47
5.4.3.1	Deletion of a Vertex	47
5.4.3.2	Deletion of an Edge	47
5.4.4	Traversal	47
5.4.4.1	Depth First Search (DFS)	47
5.4.4.2	Breadth First Search (BFS)	47
5.4.5	Shortest Path	47
5.4.6	Minimum Spanning Tree	47
5.5	Complexity Analysis of Graphs	47
5.6	Summary	47
6	Sorting and Searching	48
6.1	Introduction	49
6.2	Sorting	49
6.2.1	Types of Sorting Algorithms	49
6.2.1.1	Bubble Sort	49
6.2.1.2	Selection Sort	49
6.2.1.3	Insertion Sort	49
6.2.1.4	Merge Sort	49
6.2.1.5	Quick Sort	49
6.2.1.6	Heap Sort	49
6.2.1.7	Radix Sort	49
6.2.1.8	Counting Sort	49
6.2.1.9	Bucket Sort	49
6.2.2	Comparison of Sorting Algorithms	49
6.3	Searching	49
6.3.1	Types of Searching Algorithms	49
6.3.1.1	Linear Search	49
6.3.1.2	Binary Search	49
6.3.1.3	Jump Search	49
6.3.1.4	Interpolation Search	49
6.3.1.5	Exponential Search	49
6.3.1.6	Fibonacci Search	49
6.3.1.7	Ternary Search	49
6.3.2	Comparison of Searching Algorithms	49
6.4	Summary	49
7	Hashing	50
7.1	Introduction	50
7.2	Hash Table	50
7.3	Hash Function	50
7.4	Collision Resolution Techniques	50
7.4.1	Separate Chaining	50
7.4.2	Open Addressing	50
7.4.2.1	Linear Probing	50

7.4.2.2	Quadratic Probing	50
7.4.2.3	Double Hashing	50
7.5	Complexity Analysis of Hashing	50
7.6	Summary	50
8	Advanced Data Structures and Algorithms	51
8.1	Introduction	52
8.2	Advanced Data Structures	52
8.2.1	Segment Tree	52
8.2.2	Fenwick Tree	52
8.2.3	Suffix Tree	52
8.2.4	Suffix Array	52
8.2.5	Trie	52
8.2.6	Heap	52
8.2.7	Disjoint Set	52
8.2.8	Skip List	52
8.2.9	Splay Tree	52
8.2.10	Bloom Filter	52
8.2.11	KD Tree	52
8.2.12	Quad Tree	52
8.2.13	Octree	52
8.2.14	B-Tree	52
8.2.15	B+ Tree	52
8.2.16	R-Tree	52
8.2.17	X-Tree	52
8.2.18	Y-Tree	52
8.2.19	Z-Tree	52
8.3	Advanced Algorithms	52
8.3.1	Dynamic Programming	52
8.3.2	Greedy Algorithms	52
8.3.3	Backtracking	52
8.3.4	Divide and Conquer	52
8.3.5	Branch and Bound	52
8.3.6	Randomized Algorithms	52
8.3.7	Approximation Algorithms	52
8.3.8	String Matching Algorithms	52
8.3.9	Pattern Searching Algorithms	52
8.3.10	Cryptography Algorithms	52
8.3.11	Geometric Algorithms	52
8.3.12	Graph Algorithms	52
8.3.13	Network Flow Algorithms	52
8.3.14	Game Theory Algorithms	52
8.3.15	Quantum Algorithms	52
8.4	Summary	52
9	Applications of Data Structures and Algorithms	53
9.1	Applications in Computer Science	54
9.1.1	Operating Systems	54
9.1.2	Database Management Systems	54
9.1.3	Compiler Design	54
9.1.4	Networking	54
9.1.5	Artificial Intelligence	54

9.1.6	Machine Learning	54
9.1.7	Computer Graphics	54
9.1.8	Computer Vision	54
9.1.9	Robotics	54
9.1.10	Web Development	54
9.1.11	Mobile Development	54
9.1.12	Game Development	54
9.1.13	Cybersecurity	54
9.1.14	Quantum Computing	54
9.2	Applications in Real Life	54
9.2.1	Social Media	54
9.2.2	E-commerce	54
9.2.3	Healthcare	54
9.2.4	Finance	54
9.2.5	Transportation	54
9.2.6	Education	54
9.2.7	Agriculture	54
9.2.8	Manufacturing	54
9.2.9	Entertainment	54
9.2.10	Sports	54
9.2.11	Travel	54
9.2.12	Telecommunications	54
9.2.13	Energy	54
9.2.14	Environment	54
9.2.15	Politics	54
9.2.16	Military	54
9.3	Summary	54
10	References	55

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	Reversing an Array	40

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

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:

```
1 #include <iostream>
2 namespace std;
3
4 int main() {
5     cout << "Hello, World!" << endl;
6     return 0;
7 }
```

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:

```
1 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:

```
1 ./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.

```
1 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.

```
1 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.

```
1 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.

```
1 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.

```
1 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++.

```
1 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++.

```
1 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.

```
1 struct Person {  
2     char name[50];  
3     int age;  
4     float height;  
5 };
```

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.

```
1 class Circle {  
2     private:  
3         float radius;  
4     public:  
5         float getArea() {  
6             return 3.14 * radius * radius;  
7         }  
8 };
```

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.

```
1 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++.

```
1 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.

```
1 struct Node {  
2     int data;  
3     Node *next;  
4 };
```

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.

```
1 struct Node {  
2     int data;  
3     Node *next;  
4     Node *prev;  
5 };
```

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.

```
1 struct Node {  
2     int data;  
3     Node *next;  
4 };
```

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.

```
1 struct Node {  
2     int data;  
3     Node *next;  
4     Node *prev;
```

```
5 };
```

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).

```
1 stack<int> stk;  
2 stk.push(1);  
3 stk.push(2);  
4 stk.push(3);  
5 stk.push(4);  
6 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).

```
1 queue<int> que;  
2 que.push(1);  
3 que.push(2);  
4 que.push(3);  
5 que.push(4);  
6 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.

```
1 priority_queue<int> pq;
2 pq.push(1);
3 pq.push(4);
4 pq.push(2);
5 pq.push(3);
6 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.

```
1 deque<int> dq;
2 dq.push_front(1);
3 dq.push_back(2);
4 dq.push_front(3);
5 dq.push_back(4);
6 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.

```
1 struct Node {
2     int data;
3     Node *left;
4     Node *right;
5 };
```

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.

```
1 struct Graph {  
2     int V;  
3     list<int> *adj;  
4 };
```

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.

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

Code 1.25: Ordered Map Data Type

```
1 unordered_map<string, int> ump;  
2 ump["one"] = 1;  
3 ump["two"] = 2;  
4 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.

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



```

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

```

Code 1.27: Ordered Set Data Type

```

1 unordered_set<int> ust;
2 ust.insert(1);
3 ust.insert(2);
4 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.

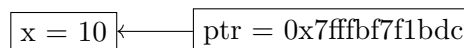


Figure 1: Pointer Example

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 `0x7ffbf7f1bdc`.

```

1 int main() {
2     int x = 10;
3     int *ptr = &x;
4
5     cout << *ptr; // Output: 10
6
7     return 0;
8 }

```

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 `0x7ffbf7f1bdc`. 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:

```
1 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.

```
1 int x = 10;
2 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.

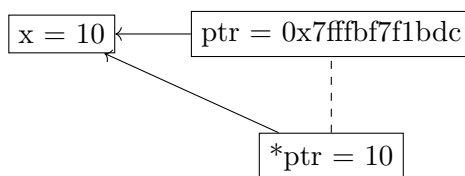


Figure 2: Dereferencing Pointers

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.

```
1 int x = 10;
2 int *ptr = &x;
3
4 cout << *ptr; // Output: 10
5
```

```

6 *ptr = 20;
7
8 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.

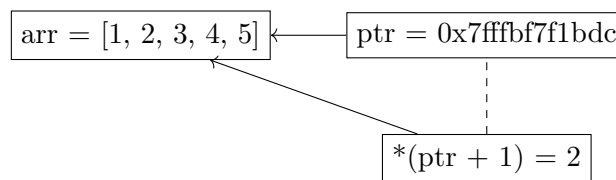


Figure 3: Pointer Arithmetic

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.

```

1 int main() {
2     int arr[5] = {1, 2, 3, 4, 5};
3     int *ptr = arr;
4
5     cout << *ptr; // Output: 1
6     cout << *(ptr + 1); // Output: 2
7
8     return 0;
9 }

```

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.

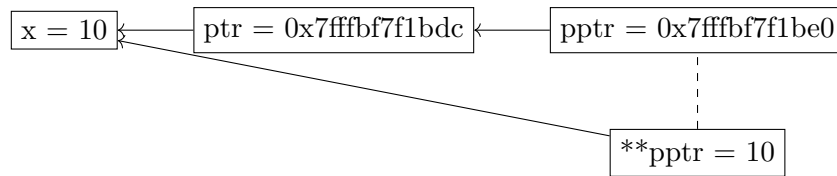


Figure 4: Pointer to Pointer

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.

```

1 int main() {
2     int x = 10;
3     int *ptr = &x;
4     int **pptr = &ptr;
5
6     cout << **pptr; // Output: 10
7
8     return 0;
9 }

```

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.

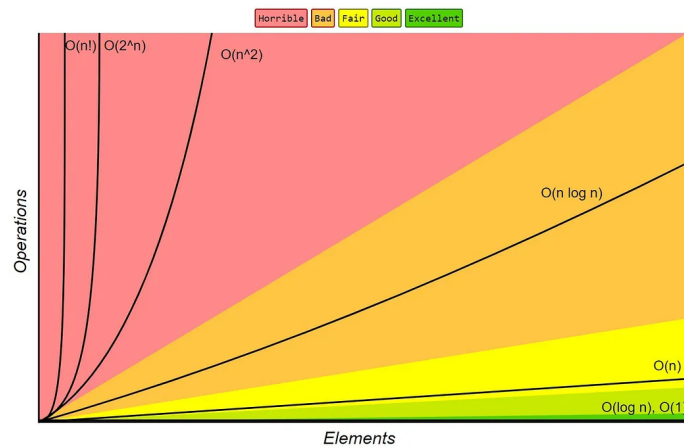


Figure 5: Asymptotic Notation

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.

```
1 int binarySearch(int arr[], int n, int x) {
2     int low = 0, high = n - 1;
3     while (low <= high) {
4         int mid = low + (high - low) / 2;
5         if (arr[mid] == x) return mid;
6         else if (arr[mid] < x) low = mid + 1;
7         else high = mid - 1;
8     }
9     return -1;
10 }
```

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.

```
1 int findMax(int arr[], int n) {
2     int max = arr[0];
3     for (int i = 1; i < n; i++) {
4         if (arr[i] > max) max = arr[i];
5     }
6     return max;
7 }
```

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.

```
1 void merge(int arr[], int l, int m, int r) {
2     // Merge two subarrays of arr[]
3     int i, j, k;
4     int n1 = m - l + 1;
5     int n2 = r - m;
6
7     int *L = new int[n1];
8     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.

```
1 int fibonacci(int n) {  
2     if (n <= 1) return n;  
3     return fibonacci(n - 1) + fibonacci(n - 2);  
4 }
```

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.

```
1 void permute(string str, int l, int r) {  
2     if (l == r) cout << str << endl;  
3     else {  
4         for (int i = l; i <= r; i++) {  
5             swap(str[l], str[i]);  
6             permute(str, l + 1, r);  
7             swap(str[l], str[i]);  
8         }  
9     }  
10 }
```

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.


```
1 void swap(int &a, int &b) {  
2     a = a + b;  
3     b = a - b;  
4     a = a - b;  
5 }
```

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.

```
1 void reverseArray(int arr[], int n) {  
2     int start = 0, end = n - 1;  
3     while (start < end) {  
4         int temp = arr[start];  
5         arr[start] = arr[end];  
6         arr[end] = temp;  
7         start++;  
8         end--;  
9     }  
10 }
```

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.

```
1 void allPairs(int arr[], int n) {  
2     vector<int> pairs(n * n);  
3     for (int i = 0; i < n; i++) {  
4         for (int j = 0; j < n; j++) {  
5             pairs[i * n + j] = arr[i] + arr[j];  
6         }  
7     }  
8 }
```

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.

```
1 void generateSubsets(int arr[], int n) {  
2     for (int i = 0; i < (1 << n); i++) {  
3         for (int j = 0; j < n; j++) {  
4             if (i & (1 << j)) cout << arr[j] << " ";  
5         }  
6         cout << endl;  
7     }  
8 }
```

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.

```
1 void permute(string str, int l, int r) {  
2     if (l == r) cout << str << endl;  
3     else {  
4         for (int i = l; i <= r; i++) {  
5             swap(str[l], str[i]);  
6             permute(str, l + 1, r);  
7             swap(str[l], str[i]);  
8         }  
9     }  
10 }
```

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.

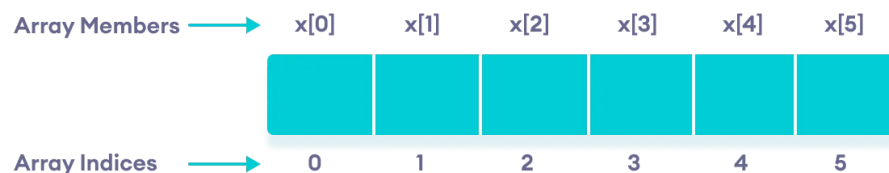


Figure 6: Elements of an array in C++

Elements of an array in C++ from Programiz

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

```
1 // array.cpp
2 int main() {
3     int arr[6];
4     return 0;
5 }
```

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.

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

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.

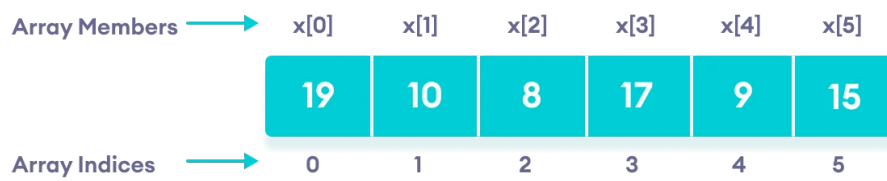


Figure 7: Initializing Array Elements

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.

```
1 int main() {
2     int arr[6] = {19, 10, 8, 17, 9, 15};
3     return 0;
4 }
```

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.

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

```

3   return 0;
4 }

```

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.

```

1 int main() {
2     int arr[6] = {19, 10, 8};
3     return 0;
4 }

```

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.

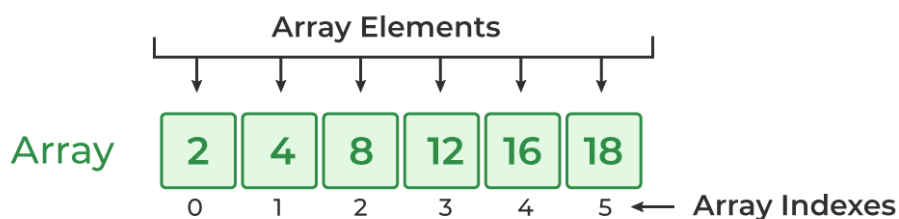


Figure 9: One-dimensional Array in C++

One-dimensional Array in C++ from GeekforGeeks

```

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

```

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.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

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.

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

Code 2.7: Two-dimensional Array

The above code snippet declares and initializes a two-dimensional array named `arr` with 3 rows and 4 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.

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

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.

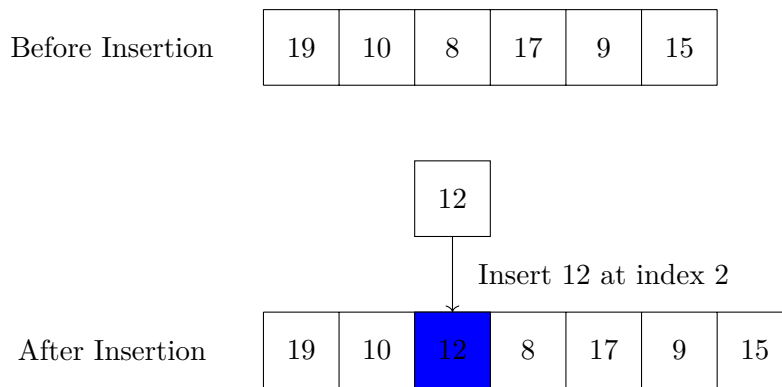


Figure 11: Array Insertion

```

1 int main() {
2     int arr[100] = {19, 10, 8, 17, 9, 15};
3     int n = 6;
4     int index = 2;
5     int value = 12;
6
7     for (int i = n - 1; i >= index; i--) {
8         arr[i + 1] = arr[i];
9     }
10    arr[index] = value;
11    n++;
12
13    return 0;
14 }

```

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.

```

1 int main() {
2     int arr[100] = {19, 10, 8, 17, 9, 15};
3     int n = 6;
4     int index = 2;
5
6     for (int i = index; i < n - 1; i++) {

```

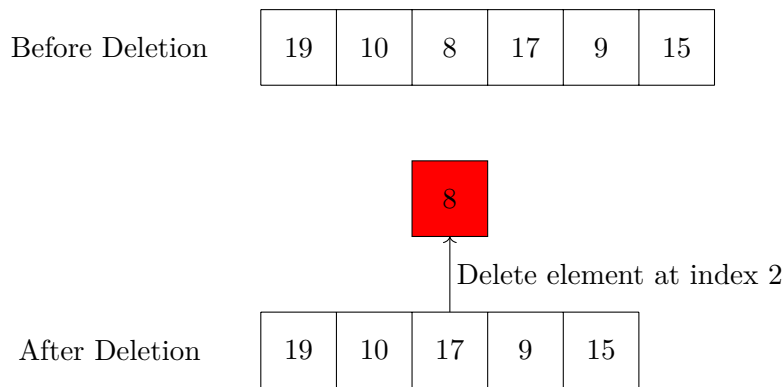


Figure 12: Array Deletion

```

7      arr[i] = arr[i + 1];
8  }
9  n--;
10
11  return 0;
12 }
```

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.

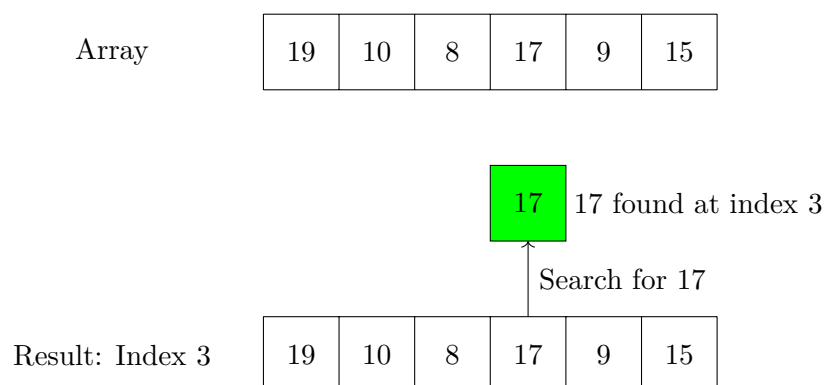


Figure 13: Array Searching

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.

```

1  int main() {
2      int arr[100] = {19, 10, 8, 17, 9, 15};
3      int n = 6;
4      int value = 17;
5      int index = -1;
6
7      for (int i = 0; i < n; i++) {
8          if (arr[i] == value) {
9              index = i;
10             break;
11         }
12     }
13
14     return 0;
15 }

```

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.

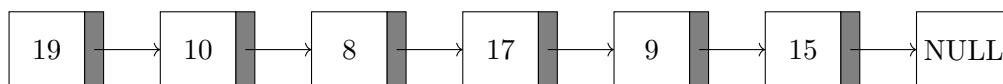


Figure 14: Singly Linked List

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.

```

1  struct Node {
2      int data;

```

```

3   Node *next;
4
5   Node(int data) {
6       this->data = data;
7       this->next = NULL;
8   }
9 };
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     return 0;
28 }

```

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.

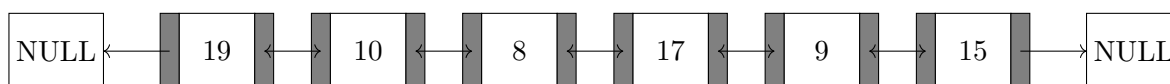


Figure 15: Doubly Linked List

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.

```
1 struct Node {
2     int data;
3     Node *next;
4     Node *prev;
5
6     Node(int data) {
7         this->data = data;
8         this->next = NULL;
9         this->prev = NULL;
10    }
11 };
12
13 int main() {
14     Node *first = new Node(19);
15     Node *second = new Node(10);
16     Node *third = new Node(8);
17     Node *fourth = new Node(17);
18     Node *fifth = new Node(9);
19     Node *sixth = new Node(15);
20
21     Node *head = first;
22
23     first->next = second;
24     second->next = third;
25     third->next = fourth;
26     fourth->next = fifth;
27     fifth->next = sixth;
28
29     second->prev = first;
30     third->prev = second;
31     fourth->prev = third;
32     fifth->prev = fourth;
33     sixth->prev = fifth;
34
35     return 0;
36 }
```

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.

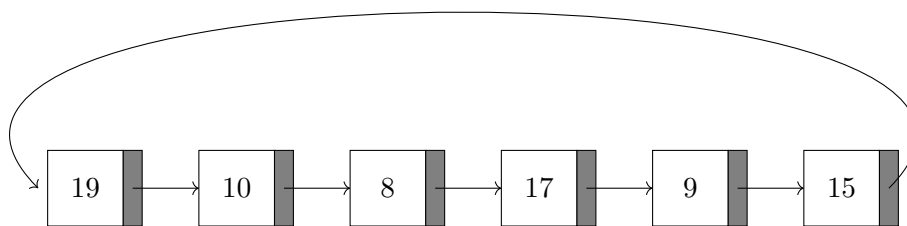


Figure 16: Circular Linked List

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.

```

1 struct Node {
2     int data;
3     Node *next;
4
5     Node(int data) {
6         this->data = data;
7         this->next = NULL;
8     }
9 };
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     sixth->next = head;
27
28     return 0;
29 }

```

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.

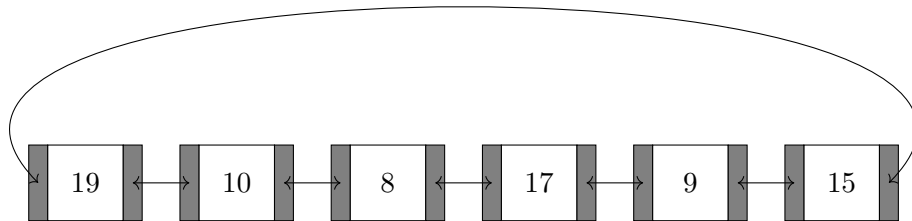


Figure 17: Doubly Circular Linked List

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.

```

1 struct Node {
2     int data;
3     Node *next;
4     Node *prev;
5
6     Node(int data) {
7         this->data = data;
8         this->next = NULL;
9         this->prev = NULL;
10    }
11 };
12
13 int main() {
14     Node *first = new Node(19);
15     Node *second = new Node(10);
16     Node *third = new Node(8);
17     Node *fourth = new Node(17);
18     Node *fifth = new Node(9);
19     Node *sixth = new Node(15);
20
21     Node *head = first;
22
23     first->next = second;
24     second->next = third;
25     third->next = fourth;
26     fourth->next = fifth;
27     fifth->next = sixth;

```

```

28     sixth->next = first;
29
30     first->prev = sixth;
31     second->prev = first;
32     third->prev = second;
33     fourth->prev = third;
34     fifth->prev = fourth;
35     sixth->prev = fifth;
36
37     return 0;
38 }

```

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.

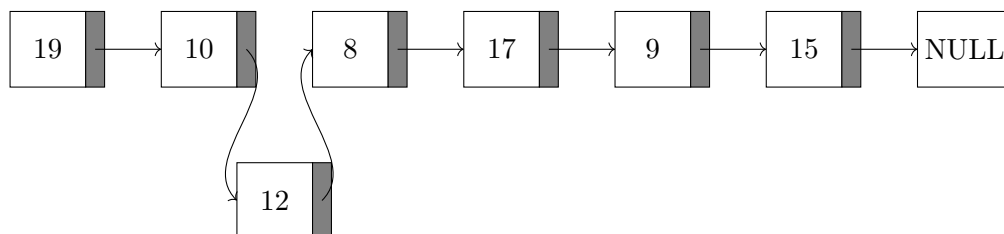


Figure 18: Linked List Insertion

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.

```

1 struct Node {
2     int data;
3     Node *next;
4
5     Node(int data) {
6         this->data = data;
7         this->next = NULL;
8     }
9 };

```



```

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;

```

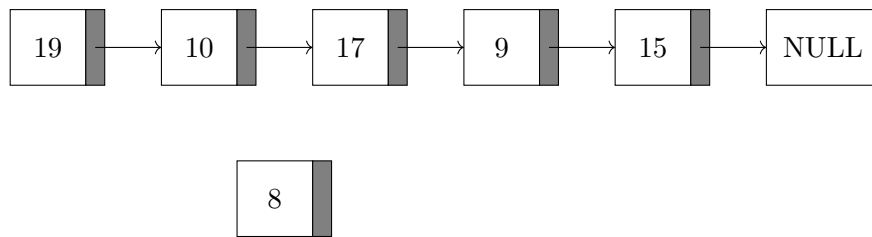


Figure 19: Linked List Deletion

```

3   Node *next;
4
5   Node(int data) {
6       this->data = data;
7       this->next = NULL;
8   }
9 };
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 *temp = head;
28     int index = 2;
29
30     for (int i = 0; i < index - 1; i++) {
31         temp = temp->next;
32     }
33
34     Node *deletedNode = temp->next;
35     temp->next = temp->next->next;
36     delete deletedNode;
37
38     return 0;
39 }

```

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.

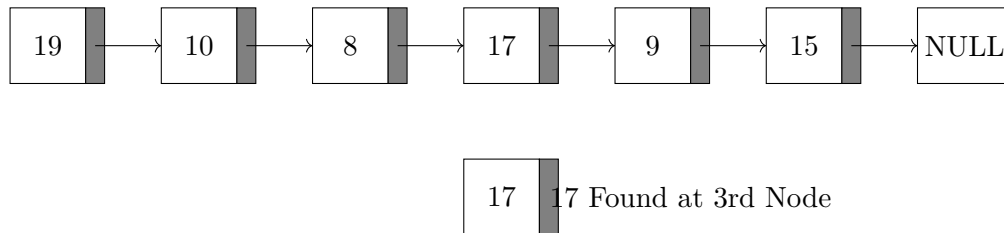


Figure 20: Linked List Searching

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.

```

1 struct Node {
2     int data;
3     Node *next;
4
5     Node(int data) {
6         this->data = data;
7         this->next = NULL;
8     }
9 };
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     int searchValue = 17;
28     Node *temp = head;
29     int position = 1;

```

```
30
31     while (temp != NULL) {
32         if (temp->data == searchValue) {
33             break;
34         }
35         temp = temp->next;
36         position++;
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.

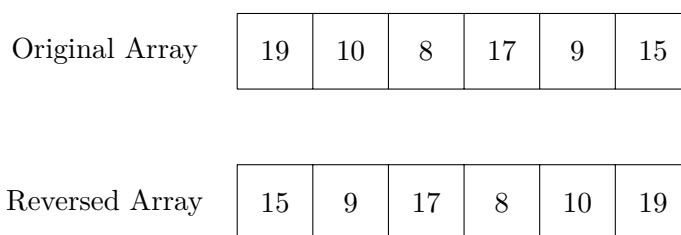


Figure 21: Reversing an Array

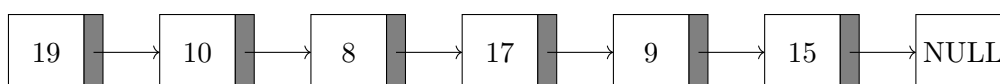
- (d) Print the reversed array to the console.

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

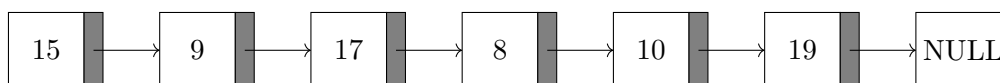
- (e) Determine the **time complexity** and **space complexity** of the program.

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.



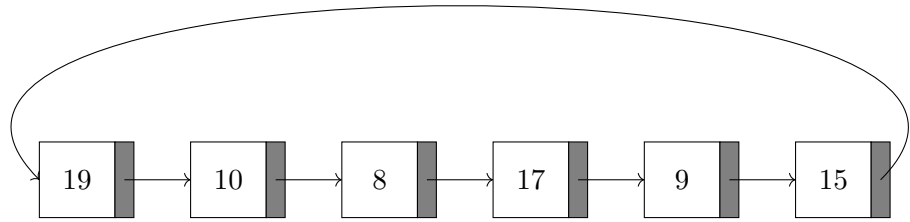
- (c) Print the reversed linked list to the console.

```
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

Element not found.

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

3

Stacks and Queues

3.1 Introduction

3.2 Stacks

3.2.1 Operations on Stacks

3.2.1.1 Push

3.2.1.2 Pop

3.2.1.3 Peek

3.2.1.4 isEmpty

3.2.1.5 isFull

3.2.2 Complexity Analysis of Stacks

3.2.3 Implementation of Stacks Using Arrays

3.2.4 Implementation of Stacks Using Linked Lists

3.3 Queues

3.3.1 Types of Queues

3.3.1.1 Linear Queue

3.3.1.2 Circular Queue

3.3.1.3 Priority Queue

3.3.1.4 Double-ended Queue (Deque)

3.3.2 Operations on Queues

3.3.2.1 Enqueue

3.3.2.2 Dequeue

3.3.2.3 Front

3.3.2.4 Rear

3.3.3 Complexity Analysis of Queues

3.3.4 Implementation of Queues Using Arrays

3.3.5 Implementation of Queues Using Linked Lists

3.4 Comparison of Stacks and Queues

4

Trees

4.1 Introduction

4.2 Properties of Trees

4.2.1 Root Node

4.2.2 Parent Node

4.2.3 Child Node

4.2.4 Leaf Node

4.2.5 Ancestors

4.2.6 Siblings

4.2.7 Descendants

4.2.8 Height of a Tree

4.2.9 Depth of a Node

4.2.10 Degree of a Node

4.2.11 Level of a Node

4.2.12 Subtree

4.3 Types of Trees

4.3.1 Binary Tree

4.3.1.1 Types of Binary Trees

4.3.1.1.1 Left-skewed Binary Tree

4.3.1.1.2 Right-skewed Binary Tree

4.3.1.1.3 Complete Binary Tree

4.3.2 Ternary Tree

4.3.3 N-ary Tree

4.3.4 Binary Search Tree

4.3.5 AVL Tree

4.3.6 Red-Black Tree

4.4 Basic Operations on Trees

5

Graphs

5.1 Introduction

5.2 Properties of Graphs

5.2.1 Vertex

5.2.2 Edge

5.2.3 Degree of a Vertex

5.2.4 Path

5.3 Types of Graphs

5.3.1 Finite Graph

5.3.2 Infinite Graph

5.3.3 Trivial Graph

5.3.4 Simple Graph

5.3.5 Multi Graph

5.3.6 Null Graph

5.3.7 Complete Graph

5.3.8 Pseudo Graph

5.3.9 Regular Graph

5.3.10 Bipartite Graph

5.3.11 Labelled Graph

5.3.12 Weighted Graph

5.3.13 Directed Graph

5.3.14 Undirected Graph

5.3.15 Connected Graph

5.3.16 Disconnected Graph

5.3.17 Cyclic Graph

5.3.18 Acyclic Graph

5.3.19 Directed Acyclic Graph (DAG)

6

Sorting and Searching

6.1 Introduction

6.2 Sorting

6.2.1 Types of Sorting Algorithms

6.2.1.1 Bubble Sort

6.2.1.2 Selection Sort

6.2.1.3 Insertion Sort

6.2.1.4 Merge Sort

6.2.1.5 Quick Sort

6.2.1.6 Heap Sort

6.2.1.7 Radix Sort

6.2.1.8 Counting Sort

6.2.1.9 Bucket Sort

6.2.2 Comparison of Sorting Algorithms

6.3 Searching

6.3.1 Types of Searching Algorithms

6.3.1.1 Linear Search

6.3.1.2 Binary Search

6.3.1.3 Jump Search

6.3.1.4 Interpolation Search

6.3.1.5 Exponential Search

6.3.1.6 Fibonacci Search

6.3.1.7 Ternary Search

6.3.2 Comparison of Searching Algorithms

6.4 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

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/>