Data Structures and Algorithms [1]

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

Jarrian Vince G. Gojar[3]

September 3, 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

List of Figures

List of Tables

List of Codes

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 Pointer

A ***pointer*** is a special type of data type that stores the memory address of another data type. It is 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.

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

cout << *ptr; // Output: 10
```

Code 1.13: Pointer 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.

1.7 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.

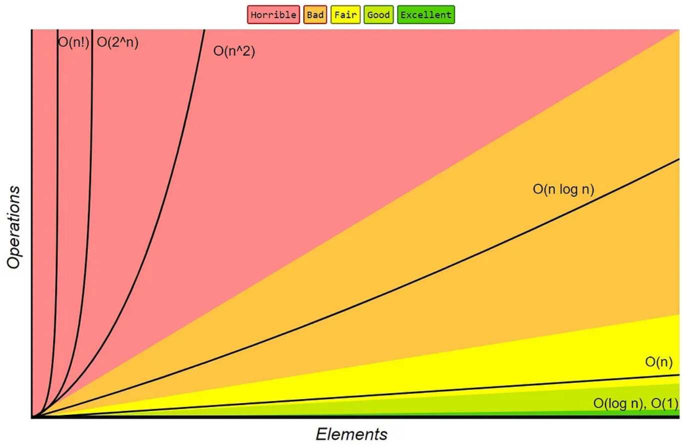

Figure 1: Asymptotic Notation

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.

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.

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

Code 1.14: 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 1.15: 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 1.16: 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];

    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    i = 0; j = 0; k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

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

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Code 1.17: 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.

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
```

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

Code 1.18: 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 1.19: 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 1.20: 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 1.21: 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 1.22: 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 1.23: 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 1.24: 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 1.25: 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.

2

Arrays and Linked Lists

2.1 Introduction

2.2 Arrays

2.2.1 Types of Arrays

2.2.1.1 One-dimensional Array

2.2.1.2 Multi-dimensional Array

2.2.2 Array Operations

2.2.2.1 Insertion

2.2.2.2 Deletion

2.2.2.3 Searching

2.2.3 Complexity Analysis of Arrays

2.3 Linked Lists

2.3.1 Types of Linked Lists

2.3.1.1 Singly Linked List

2.3.1.2 Doubly Linked List

2.3.1.3 Circular Linked List

2.3.2 Operations on Linked Lists

2.3.2.1 Insertion

2.3.2.2 Deletion

2.3.2.3 Searching

2.3.3 Complexity Analysis of Linked Lists

2.4 Comparison of Arrays and Linked Lists

2.5 Summary

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

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/