Problem Solving in Data Structures and Algorithms

Coding Spoon

A Training Report

Submitted in partial fulfillment of the requirements for the award of degree of

Bachelor of Technology

(Computer Science and Engineering)

Submitted to

LOVELY PROFESSIONAL UNIVERSITY

PHAGWARA, PUNJAB

From 26th May 2024 to 31st July 2024

SUBMITTED BY

Name of Student: Aman Rawat

Registration Number: 12211066

To whom so ever it may concern

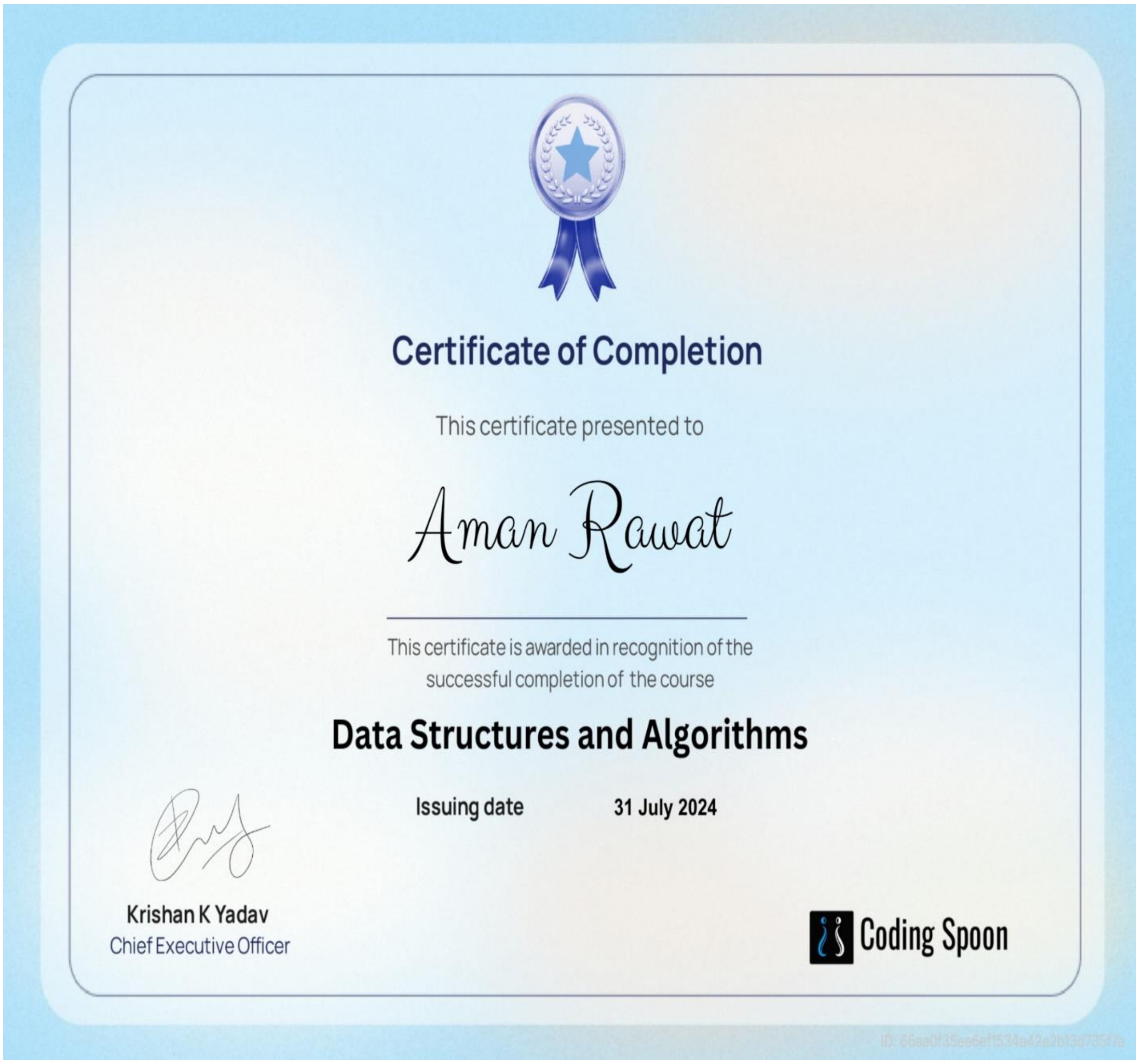
I, **Aman Rawat**, 12211066, hereby declare that the work done by me on **“Problem Solving in Data Structures and Algorithms”** from **26th May 2024 to 25th July 2024**, is a record of original work for the partial fulfilment of the requirements for the award of the degree, **Bachelor of Technology**.

Name of the Student: Aman Rawat

Registration Number: 12211066

Dated: 25th August 2024

SUMMER TRAINING CERTIFICATE



Certificate of Completion

This certificate presented to

Aman Rawat

This certificate is awarded in recognition of the
successful completion of the course

Data Structures and Algorithms

Issuing date 31 July 2024

[illegible]

Krishan K Yadav
Chief Executive Officer

Coding Spoon

ID: 66aa0f35ee6ef1534a42e2b13d735f7e

ACKNOWLEDGEMENT

First and foremost, I am deeply grateful for the opportunity to expand my knowledge and skills in a new technology. I would like to extend my sincere thanks to the instructor of the DSA course from Coding Spoon, whose guidance has been valuable in facilitating my learning journey from home.

I am also thankful to my college, Lovely Professional University, for providing access to such an enriching course that not only enhanced my programming abilities but also introduced me to new technologies.

I would like to express my heartfelt appreciation to my parents and friends for their unwavering support, valuable advice, and encouragement in choosing this course. Lastly, my gratitude extends to my classmates, whose collaboration and assistance have been greatly appreciated.

LIST OF CONTENTS

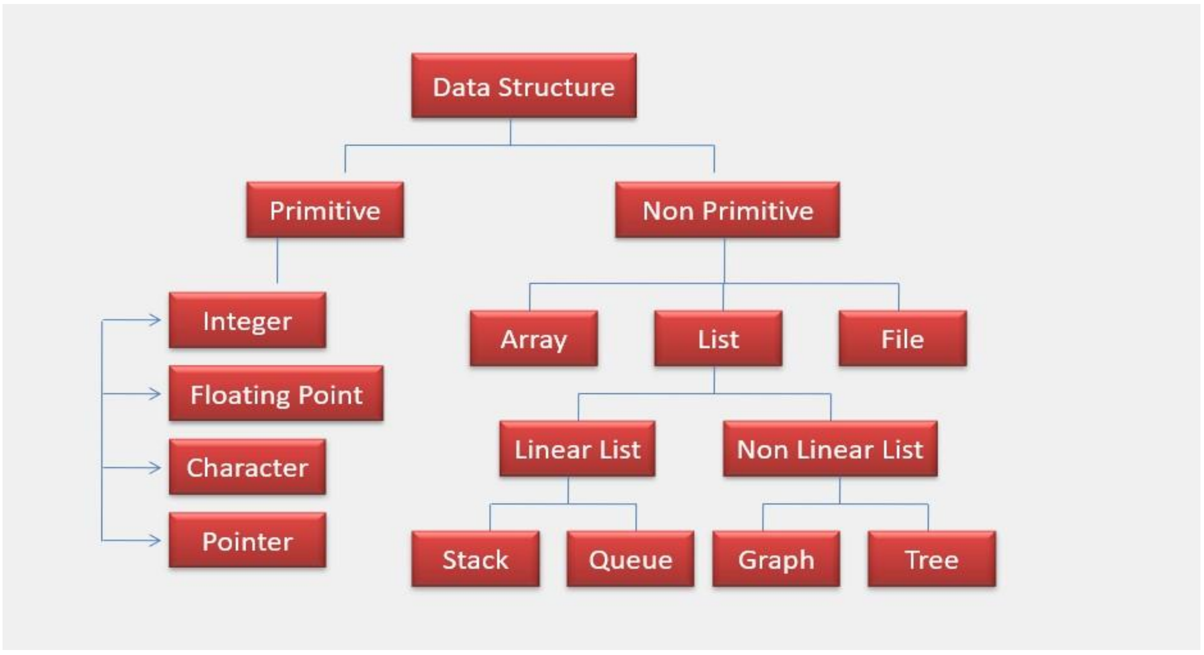
LIST OF ABBREVIATION

DSA – Data Structures and Algorithms

INTRODUCTION

➢ **What is a Data Structure?** A data structure is a method of organizing and managing data so that operations can be performed efficiently. It involves arranging data elements based on specific relationships to improve organization and storage. For instance, if we have data with a player's name "Virat" and age 26, "Virat" is a String type, while 26 is an Integer type.

➢ **Classification of Data Structure:**



➢ **What is an Algorithm?** An algorithm is a set of step-by-step instructions or logic, arranged in a specific order, to achieve a particular task. It is the fundamental solution to a problem, presented either as a high-level description in pseudocode or through a flowchart, but it is not the full code or program.

➢ This course provided a comprehensive learning experience, guiding me through Data Structures and Algorithms from the basics to advanced concepts. The curriculum is structured into 8 weeks, allowing participants to work through practice problems and assessments at their own pace. The course offers a variety of programming challenges that are invaluable in preparing for interviews with leading companies like Microsoft, Amazon, Adobe, and others.

TECHNOLOGY LEARNT

Analysis of Algorithms:

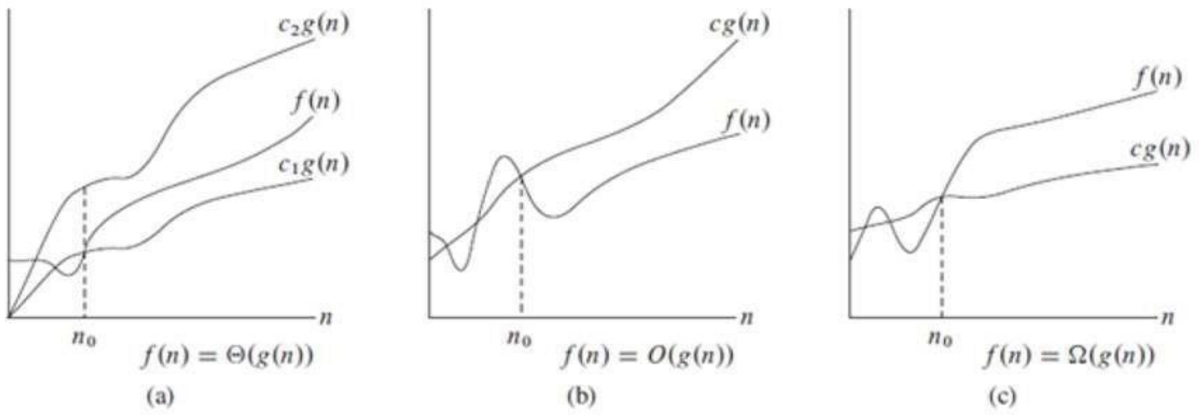
Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it

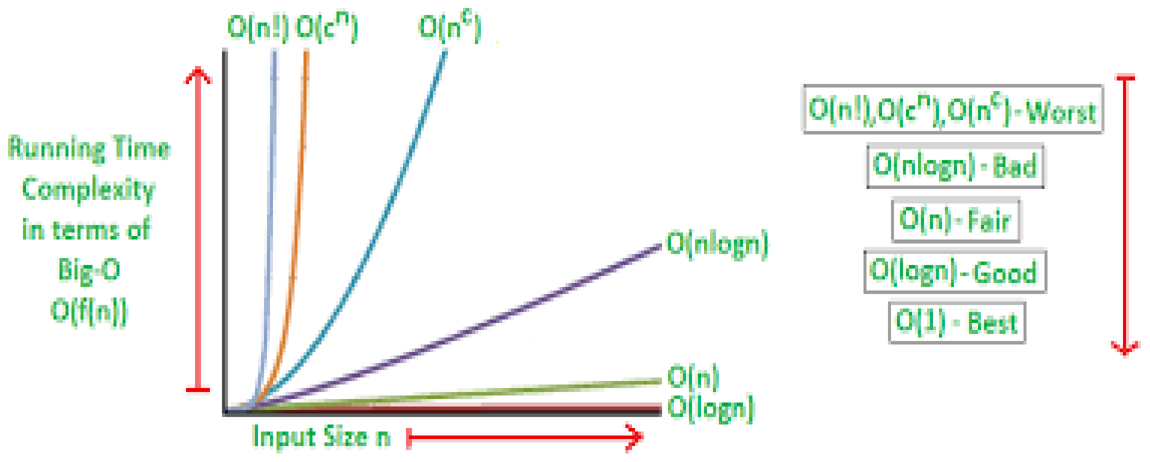
- **Analysis of Algorithm:** This section focused on understanding background analysis through programs and their functions.
- **Order of Growth:** I learned about the mathematical representation of growth analysis using limits and functions, including a straightforward approach to determining the order of growth.
- **Asymptotic Notations:** This covered the best, average, and worst-case scenarios, explained through example programs.
- **Big O Notation:** The concept was explained both graphically and mathematically, with calculations and applications demonstrated using Linear Search.
- **Omega Notation:** I explored this notation through graphical and mathematical explanations, along with calculations.
- **Theta Notation:** The Theta notation was explained with graphical and mathematical insights, including detailed calculations.
- **Recursion Analysis:** This included various calculations using the Recursion Tree method

- **Space Complexity:** I explored basic programs, auxiliary space, and analyzed the space complexity of recursive functions and the Fibonacci sequence.

Why Analysis of Algorithms is important?

To predict the behavior of an algorithm without implementing it on a specific computer. It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes. It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors. The analysis is thus only an approximation; it is not perfect. More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.





Mathematics

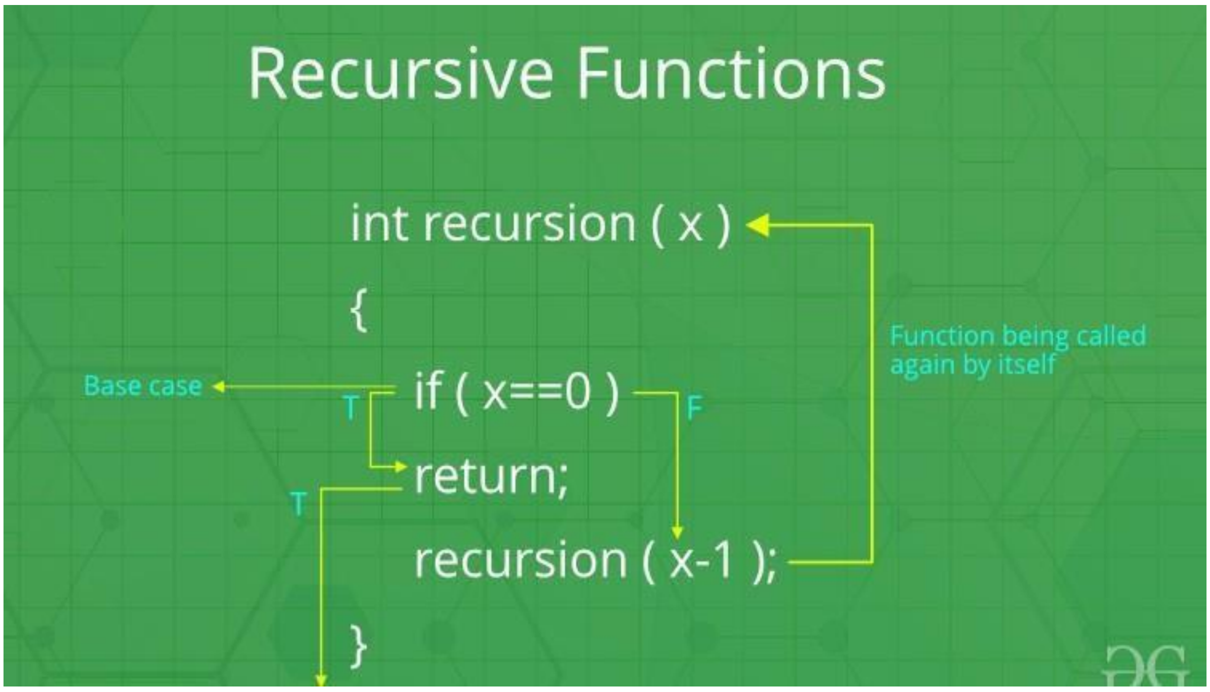
- **Digit Counting:** Techniques for finding the number of digits in a number.
- **Progressions:** Arithmetic and geometric progressions were covered.
- **Statistical Measures:** Concepts of mean and median.
- **Prime Numbers:** Understanding prime numbers.
- **LCM and HCF:** Learning about Least Common Multiple (LCM) and Highest Common Factor (HCF).
- **Factorials:** Calculating factorials.
- **Permutations and Combinations:** Exploring permutations and combinations.
- **Modular Arithmetic:** Learning the basics of modular arithmetic.

Bitwise Operations

- **Bitwise Operators in C++:** I explored the operations of AND, OR, XOR, Left Shift, Right Shift, and Bitwise NOT in C++.
- **Problem Solving:** I tackled problems like checking if the Kth bit is set, using methods such as Left Shift and Right Shift.

Recursion

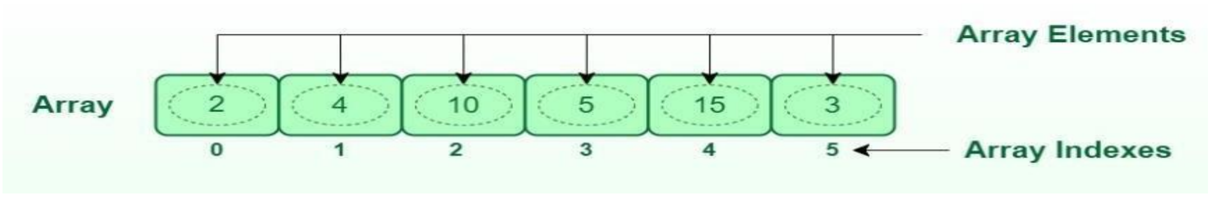
- **Introduction:** The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), In order/Preorder/Post order Tree Traversals, DFS of Graph, etc.
- **Writing Base Cases:** Learning to write base cases in recursion through examples like factorial and N-th Fibonacci number.

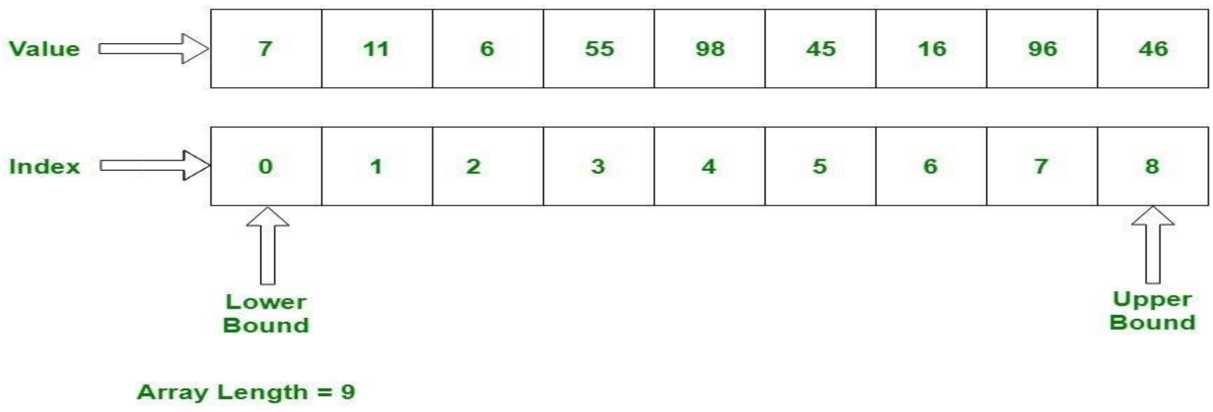


Arrays

- **Introduction and Applications:** An array is a collection of items of same data type stored at contiguous memory locations. It used in storing and accessing the data, searching, sorting the data, etc.
- **Types of Arrays:** Covered fixed-sized and dynamic-sized arrays.

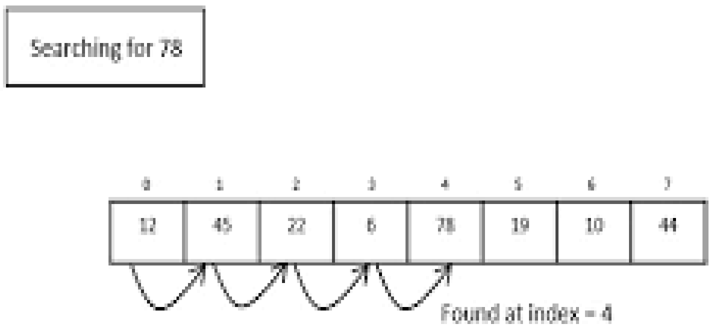
- **Array Operations:** Studied operations like searching, insertion, deletion, comparison with other data structures, and reversing arrays with complexity analysis.





Searching

- **Introduction:** I learned both iterative and recursive approaches to binary search, along with associated problems.
- **Two Pointer Approach:** Explored problems that utilize the two-pointer approach.

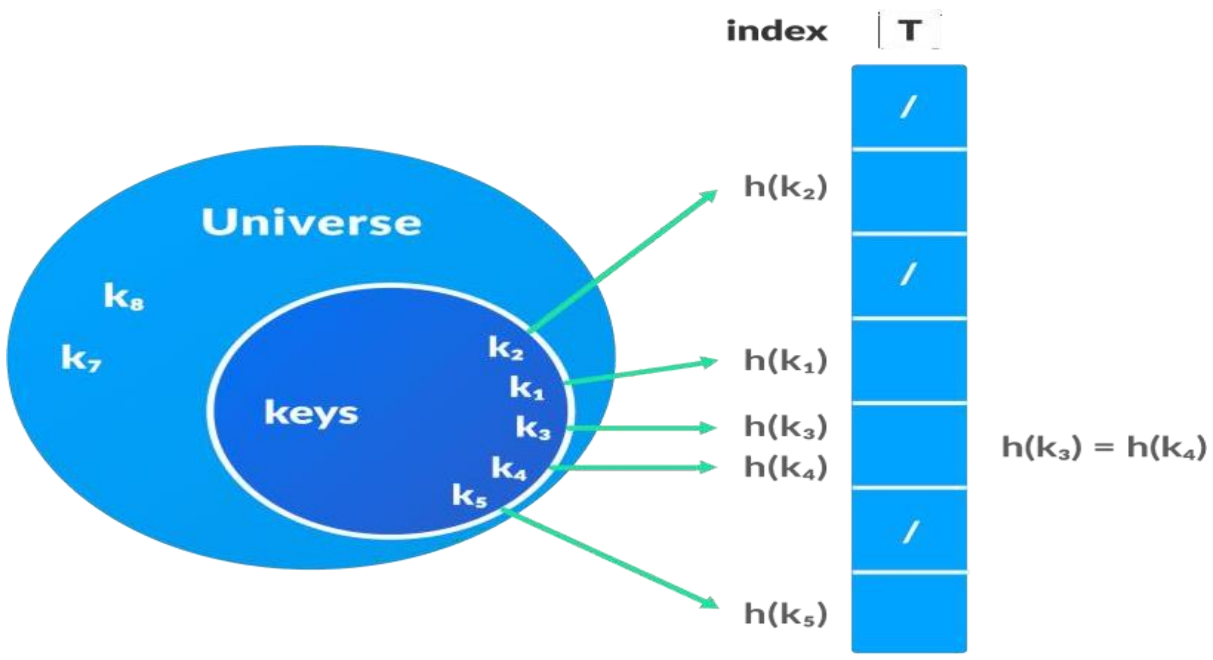


Sorting

- **STL Sort in C++:** Implementing the **sort()** function in arrays and vectors in C++ with a focus on time complexities.
- **Stability in Sorting Algorithms:** Analyzed stable and unstable sorting algorithms with examples.
- **Sorting Algorithms:** I implemented and analyzed sorting algorithms like Insertion Sort, Merge Sort, and Quick Sort (including Lomuto and Hoare partitioning methods), along with their time and space complexities, pivot selection, and worst-case scenarios.
- **Sorting Algorithms Overview:** A broad overview of sorting algorithms.

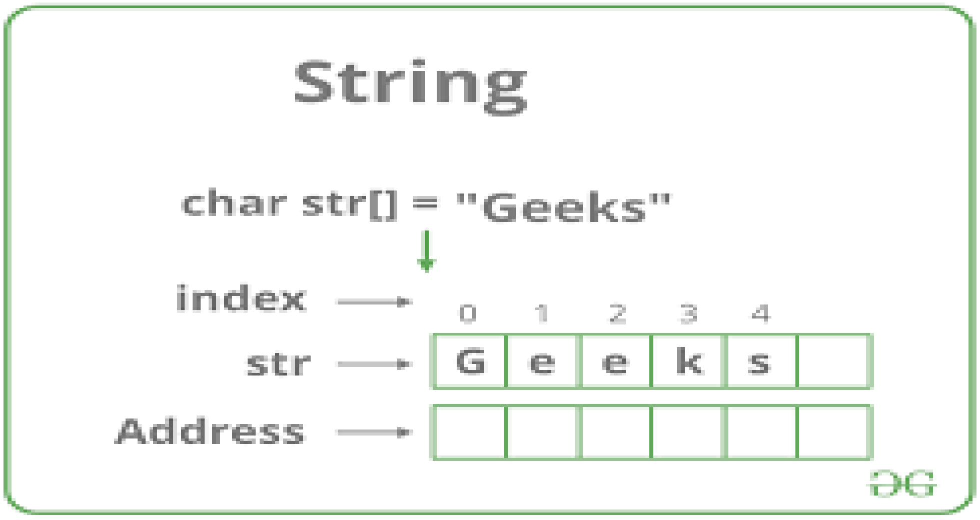
Hashing

- **Introduction to Hashing:** Hashing is a method of storing and retrieving data from a database efficiently, suppose that we want to design a system for storing employee records keyed using phone numbers.
  Time complexity analysis and applications of hashing were introduced.
- **Applications**: Password Verification, Compiler Operations, Rabin-Karp Algorithm, Cryptography, etc.
- **Hash Functions:** I learned about Direct Address Tables and various hashing functions.
- **Collision Handling:** Discussed different collision handling techniques, including chaining and open addressing, with implementations.
- **Hashing in C++:** Explored **unordered_set** and **unordered_map** in C++, and **HashSet** and **HashMap** in Java.



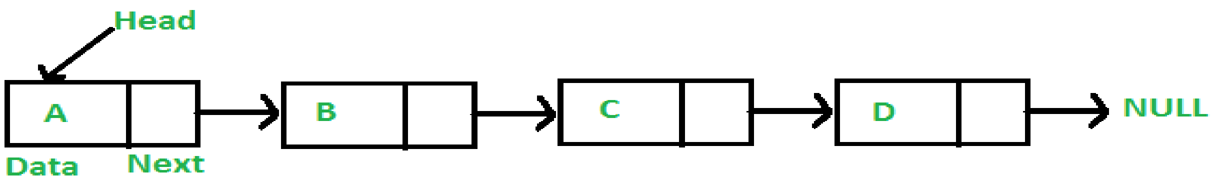
Strings

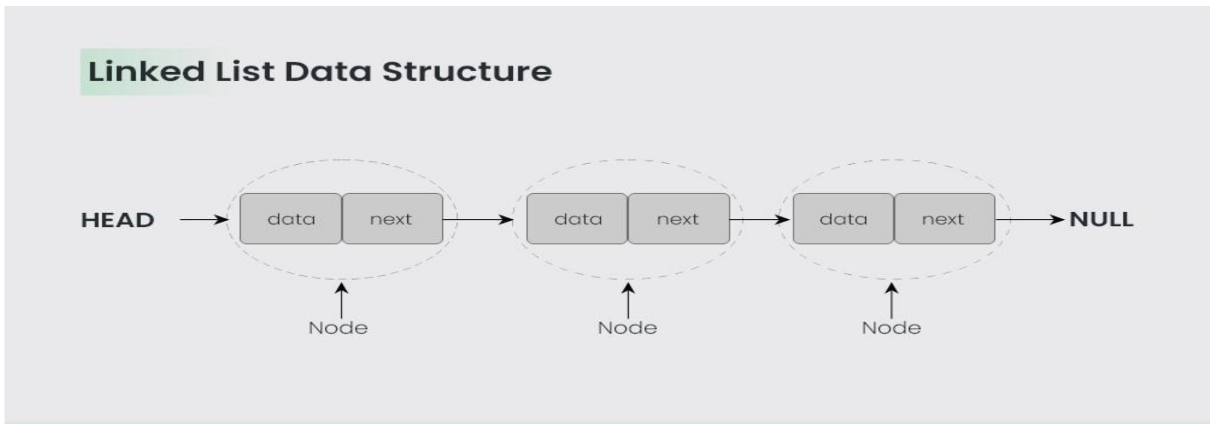
- **String Data Structure:** Studied string data structures in C++. A string is a sequence of characters used to represent text.
- **String Algorithms:** Implemented string algorithms like Rabin Karp and KMP.



Linked Lists

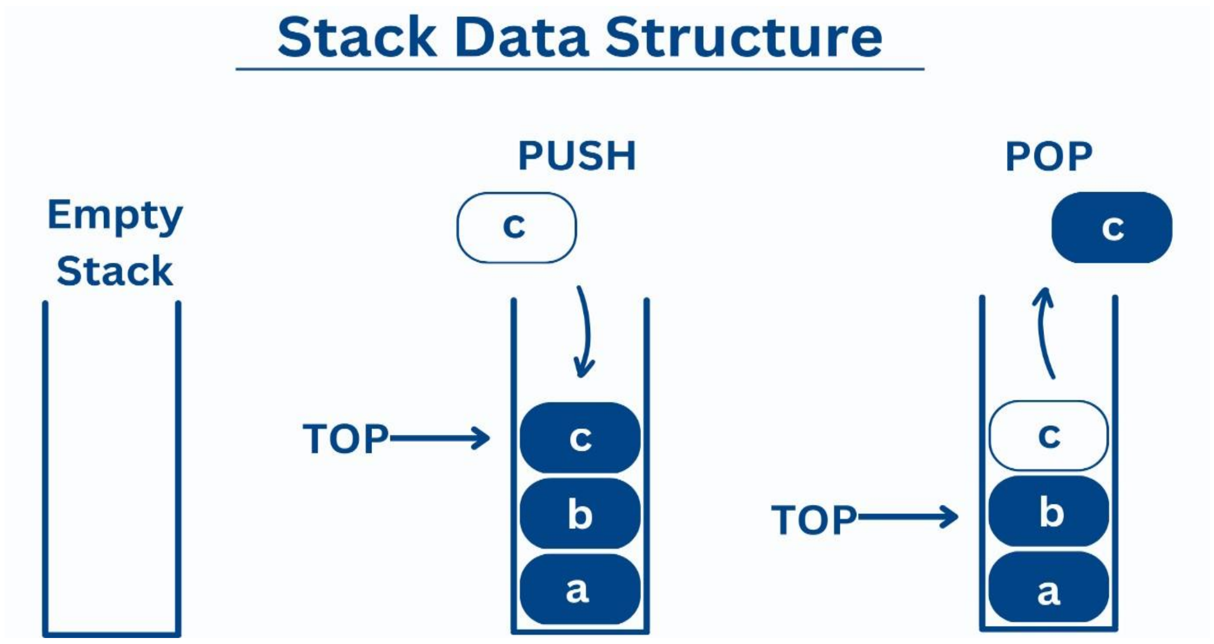
- **Introduction:** A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- **Types of Linked Lists:** Explored doubly linked lists and circular linked lists.
- **Loop Problems:** Covered problems like detecting loops using Floyd's cycle detection algorithm and detecting and removing loops in linked lists.





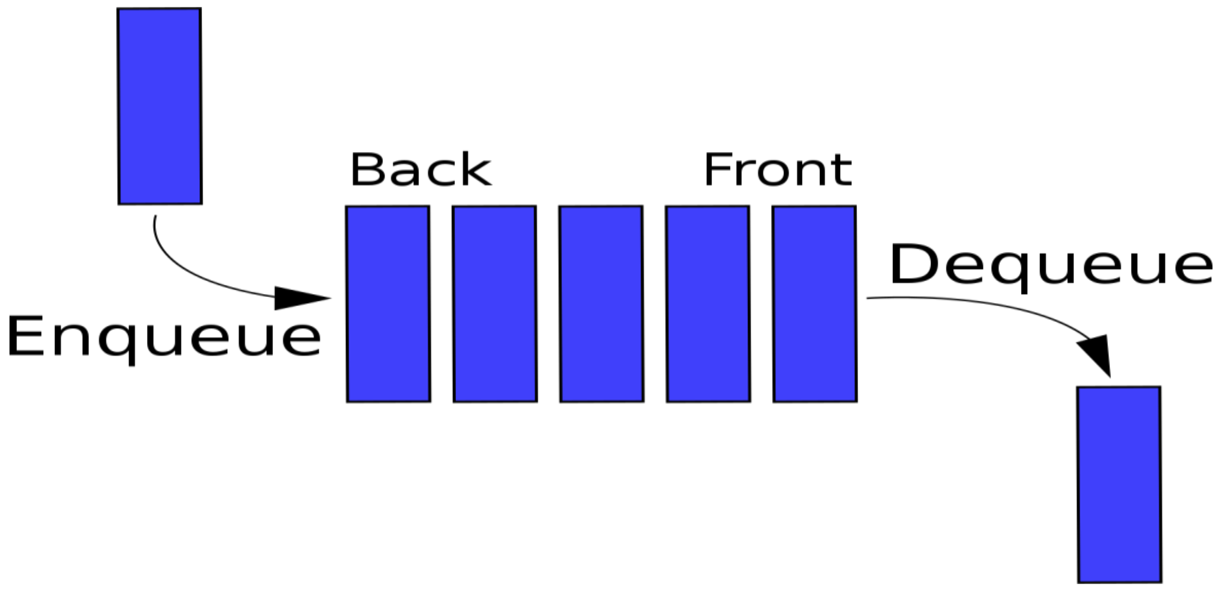
Stacks

- **Understanding Stacks:** A Stack is a linear data structure which follows LIFO (Last in First Out) order that allow us to store and retrieve data sequentially.
- **Applications:** It is used to perform various operation like insertion (PUSH) and deletion (POP), in algorithm problems like Tower of Hanoi, Infix to Postfix, etc.
- **Stack Implementation:** Implementation of stacks using arrays and linked lists in C++.



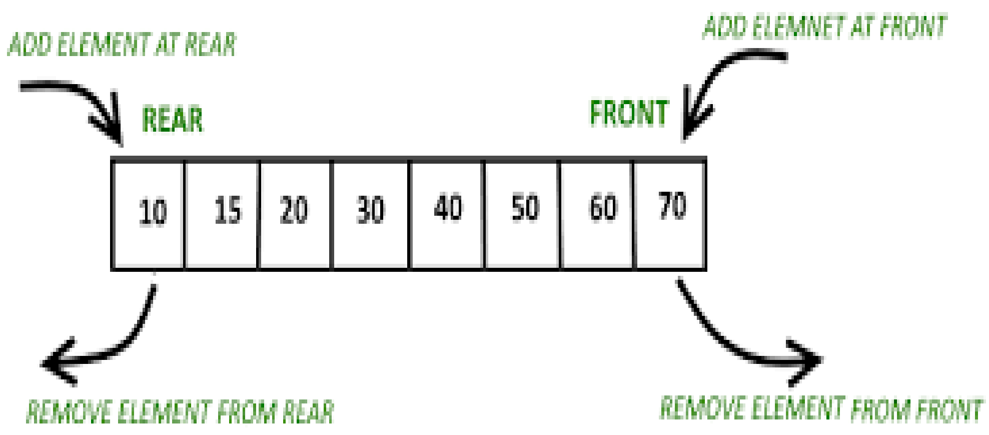
Queues

- **Introduction to Queues:**A queue is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. It operates like a line where elements are added at one end (**rear**) and removed from the other end (**front**).
- **Applications:** It is used in Task Scheduling, Resource Allocation, Traffic Management, etc.
- **Queue Implementation:** Implemented queues using arrays and linked lists in C++ STL, including stack-based queue implementation.



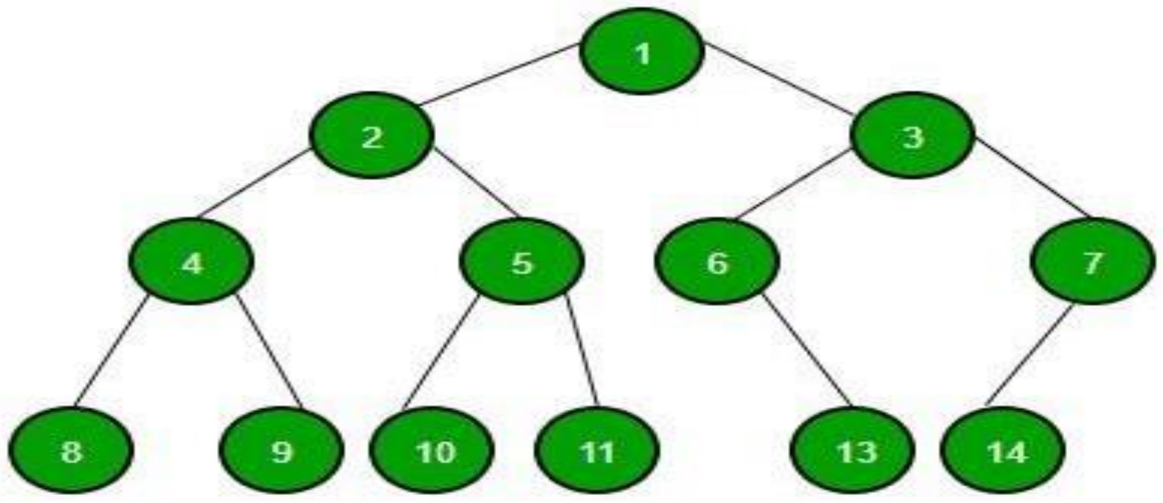
Deques

- **Introduction to Deques:** A Queue is linear data structure which follows LIFO (Last in First Out) and FIFO (First in First Out) order that allow insertion are done at one end and deletion are done at another end.
- **Applications:** It is used in Job Scheduling algorithms, BFS, undo operations, etc.
- **Problem Solving:** Tackled problems like finding the maximum of all subarrays of size k and designing a data structure with min-max operations.



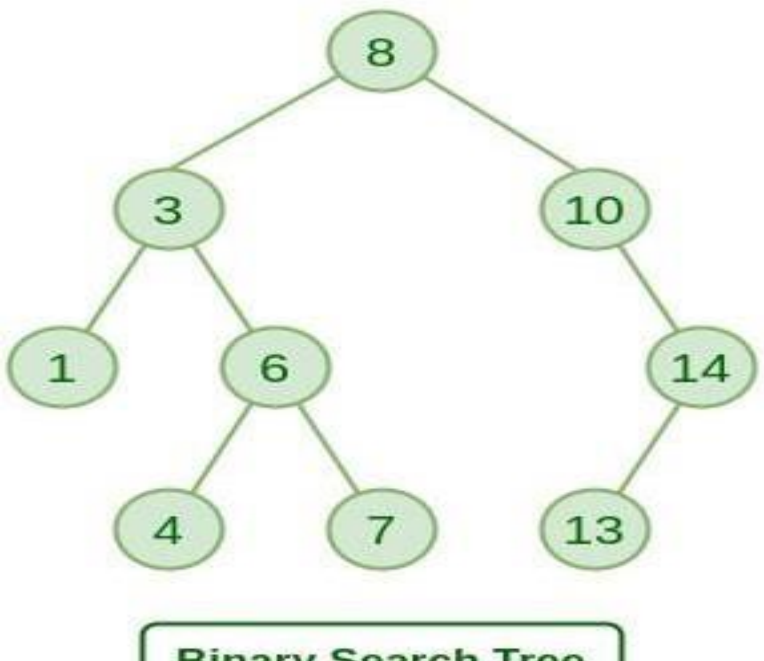
Trees

- **Trees:** A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links.
- **Binary Tree:** A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child.
- **Applications:** It is used in File Explorer, Database Systems, Routing Algorithm, etc.
- **Tree Traversals:** Implemented various tree traversal techniques like Inorder, Preorder, Postorder, Level Order (line by line).



Binary Search Trees (BST)

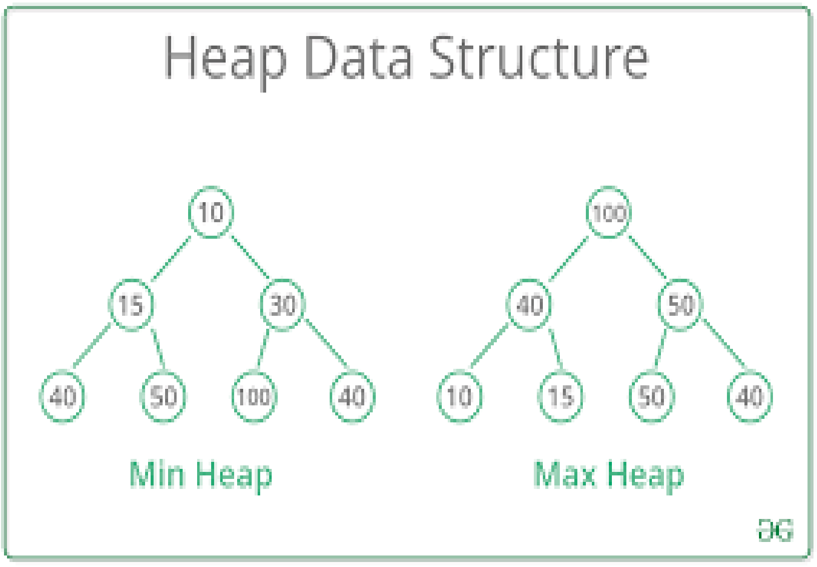
- **Introduction to BST:** A BST is a data structure used to storing data in a sorted manner. Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node.
- **Applications:** It is used in File Explorer, Database Systems, Routing Algorithm, etc.
- **BST Operations:** Implemented search, insertion, deletion, and floor operations in BST, along with exploring self-balancing BSTs.



Heaps

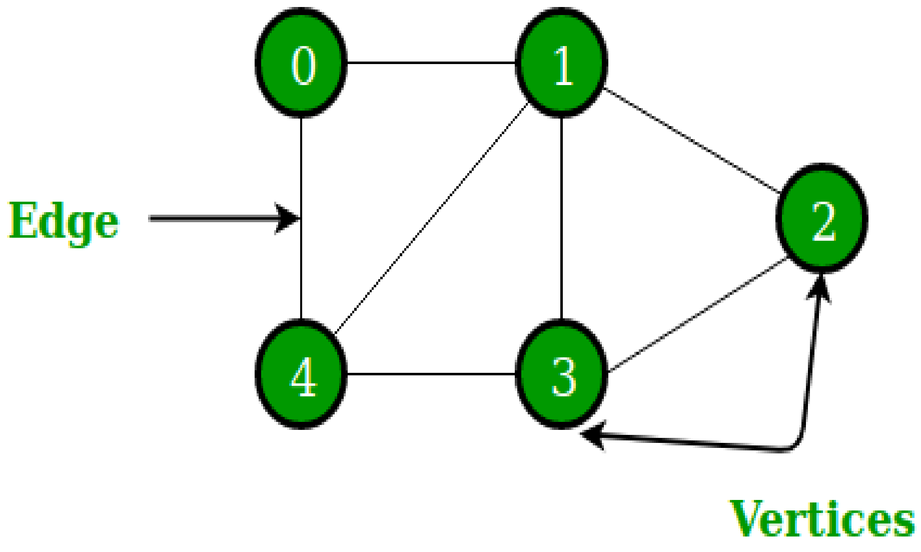
- **Introduction to Heaps:** Heap is a special tree-based data structure where the tree is always a complete binary tree. Heaps are of two types: Max heap and Min heap.
- **Applications:** It is used in medical applications, Priority Queues, Resource Allocation, etc.
- **Binary Heap Operations:** Implemented binary heap operations like insertion, heapify, extract, decrease key, delete, and building a heap.

- **Heap Sort:** Studied heap sort, and priority queues in C++.



Graphs

- **Introduction to Graphs:** A graph is a non-linear data structure, which consists of vertices (or nodes) connected by edges (or arcs) where edges may be directed or undirected.
- **Applications:** It is used in Google Maps, WWW, Facebook, etc.
- **Graph Traversal:** Implemented Breadth-First Search (BFS) and Depth-First Search (DFS) along with their applications.
- **Shortest Path Algorithms:** Studied algorithms for finding the shortest path in Directed Acyclic Graphs, Prim's Algorithm for Minimum Spanning Tree, Dijkstra's Shortest Path Algorithm, Bellman-Ford Algorithm, Kosaraju's Algorithm.



Greedy Algorithms

- **Introduction to Greedy Algorithms:** A greedy Algorithm is defined as a method for solving optimization problems by making decisions that result in the most evident and immediate benefit irrespective of the outcome. It works for cases where minimization or maximization leads to the required solution.
- **Greedy Problems:** Implemented and analyzed problems like Activity Selection, Fractional Knapsack, and Job Sequencing.

Backtracking

- **Backtracking Concepts:** Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**. It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.
- **Applications:** Introduction to backtracking with examples like the Rat in a Maze and N-Queen problem.

Dynamic Programming

- **Introduction to Dynamic Programming:** This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.
- **Dynamic Programming Techniques:** Learned and implemented memoization and tabulation methods.

➢ Enhanced my problem-solving abilities by tackling a variety of challenges to become a more proficient developer.

- **Practice Problems:** This track offers numerous essential practice problems that are crucial for mastering Data Structures and Algorithms.

➢ Sharpened my analytical skills in Data Structures, enabling me to utilize them more effectively.

➢ Successfully tackled problems commonly encountered in interviews with product-based companies.

➢ Competed in contests that mirror the coding rounds for Software Development Engineer (SDE) roles.

PROJECT

CONTACT MANAGEMENT SYSTEM

Overview

A Contact Management System is a software application designed to store, manage, and retrieve contact information efficiently. This system implements a Binary Search Tree (BST) to organize contacts based on names, ensuring quick insertion, search, and deletion operations. Additionally, a hash map is utilized to enable rapid lookup of contacts by phone numbers, while a priority queue (min-heap) helps in managing the smallest name lexicographically. The system allows users to add, view, search, and delete contacts through a user-friendly console interface. In this project, we'll create a basic address book application that allows users to add, view, search for, and delete contacts. This project showcases data structures and algorithms in C++ to build an effective and responsive address book.

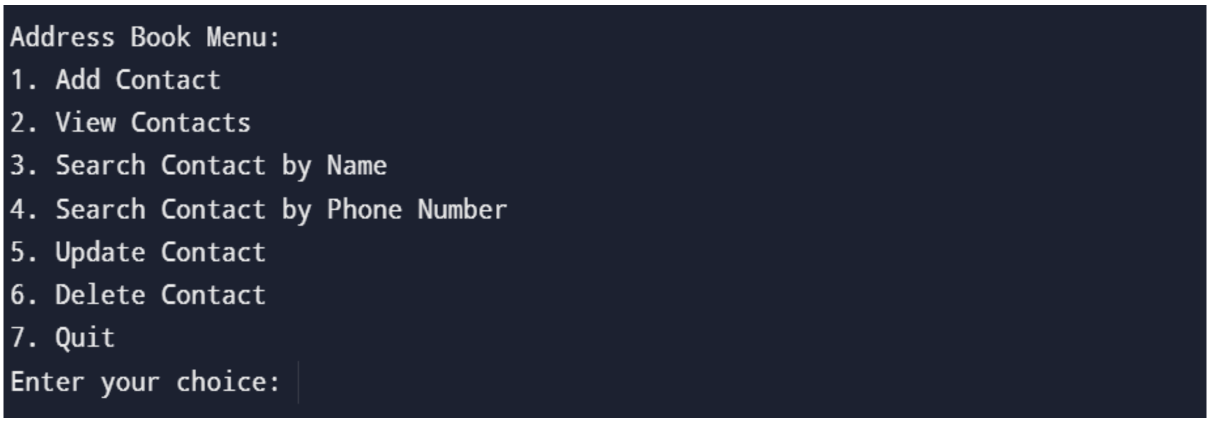
CODE:

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

class Node {
public:
    string name;
    string phoneNumber;
    Node* left;
    Node* right;

    Node(string name, string phoneNumber) {
        this->name = name;
        this->phoneNumber = phoneNumber;
        left = right = nullptr;
    }
};

struct CompareNames {
    bool operator()(const Node* a, const Node* b) {
        return a->name > b->name;
    }
};

class AddressBook {
private:
    Node* root;
    map<string, string> phoneToNameMap;
    priority_queue<Node*, vector<Node*>, CompareNames>
minHeap;

    Node* insert(Node* root, string name, string
phoneNumber) {
        if (root == nullptr) {
            Node* newNode = new Node(name, phoneNumber);
            minHeap.push(newNode);
            return newNode;
        }

        if (name < root->name) {
            root->left = insert(root->left, name, phoneNumber);
        } else if (name > root->name) {
            root->right = insert(root->right, name, phoneNumber);
        } else {
            root->phoneNumber = phoneNumber;
        }

        return root;
    }

    void inOrder(Node* root) {
        if (root != nullptr) {
            inOrder(root->left);
            cout << "Name: " << root->name << ", Phone: " <<
root->phoneNumber << endl;
            inOrder(root->right);
        }
    }

    Node* searchByName(Node* root, string name) {
        if (root == nullptr || root->name == name) {
            return root;
        }

        if (name < root->name) {
            return searchByName(root->left, name);
        } else {
            return searchByName(root->right, name);
        }
    }

    Node* minValueNode(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr) {
            current = current->left;
        }
        return current;
    }

    Node* deleteNode(Node* root, string name) {
        if (root == nullptr) return root;

        if (name < root->name) {
            root->left = deleteNode(root->left, name);
        } else if (name > root->name) {
            root->right = deleteNode(root->right, name);
        } else {
            if (root->left == nullptr) {
                Node* temp = root->right;
                delete root;
                return temp;
            } else if (root->right == nullptr) {
                Node* temp = root->left;
                delete root;
                return temp;
            }

            Node* temp = minValueNode(root->right);
            root->name = temp->name;
            root->phoneNumber = temp->phoneNumber;
            root->right = deleteNode(root->right, temp->name);
        }

        return root;
    }

public:
    AddressBook() {
        root = nullptr;
    }

    void addContact(string name, string phoneNumber) {
        root = insert(root, name, phoneNumber);
        phoneToNameMap[phoneNumber] = name;
        cout << "Contact added successfully." << endl;
    }

    void updateContact(string oldName, string newName,
string newPhoneNumber) {
        Node* result = searchByName(root, oldName);
        if (result != nullptr) {
            phoneToNameMap.erase(result->phoneNumber);
            root = deleteNode(root, oldName);
            root = insert(root, newName, newPhoneNumber);
            phoneToNameMap[newPhoneNumber] = newName;
            cout << "Contact updated successfully." << endl;
        } else {
            cout << "Contact not found." << endl;
        }
    }

    void viewContacts() {
        if (root == nullptr) {
            cout << "No contacts in the address book." << endl;
        } else {
            cout << "Contacts in the address book (sorted by
name):" << endl;
            inOrder(root);
        }
    }

    void searchContactByName(string name) {
        Node* result = searchByName(root, name);
        if (result != nullptr) {
            cout << "Found: Name: " << result->name << ",
Phone: " << result->phoneNumber << endl;
        } else {
            cout << "Contact not found." << endl;
        }
    }

    void searchContactByPhone(string phoneNumber) {
        if (phoneToNameMap.find(phoneNumber) !=
phoneToNameMap.end()) {
```

```
            cout << "Found: Name: " <<
phoneToNameMap[phoneNumber] << ", Phone: " <<
phoneNumber << endl;
        } else {
            cout << "Contact not found." << endl;
        }
    }

    void deleteContact(string name) {
        root = deleteNode(root, name);
        cout << "Contact '" << name << "' deleted (if it existed)."
<< endl;
    }
};

int main() {
    AddressBook addressBook;
    while (true) {
        cout << "\nAddress Book Menu:\n";
        cout << "1. Add Contact\n";
        cout << "2. View Contacts\n";
        cout << "3. Search Contact by Name\n";
        cout << "4. Search Contact by Phone Number\n";
        cout << "5. Update Contact\n";
        cout << "6. Delete Contact\n";
        cout << "7. Quit\n";
        cout << "Enter your choice: ";
        int choice;
        cin >> choice;
        cin.ignore();

        switch (choice) {
        case 1: {
            string name, phoneNumber;
            cout << "Enter Name: ";
            getline(cin, name);
            cout << "Enter Phone Number: ";
            getline(cin, phoneNumber);
            while (phoneNumber.length() != 10) {
                cout << "Not valid. \nPlease enter a valid 10-digit
number: ";
                getline(cin, phoneNumber);
            }
            addressBook.addContact(name, phoneNumber);
            break;
        }
        case 2:
            addressBook.viewContacts();
            break;
        case 3: {
            string name;
            cout << "Enter Name to Search: ";
            getline(cin, name);
            addressBook.searchContactByName(name);
            break;
        }
        case 4: {
            string phoneNumber;
            cout << "Enter Phone Number to Search: ";
            getline(cin, phoneNumber);
            while (phoneNumber.length() != 10) {
                cout << "Not valid. \nPlease enter a valid 10-digit
number: ";
                getline(cin, phoneNumber);
            }
            addressBook.searchContactByPhone(phoneNumber);
            break;
        }
        case 5: {
            string oldName, newName, newPhoneNumber;
            cout << "Enter Name to Update: ";
            getline(cin, oldName);
            cout << "Enter New Name: ";
            getline(cin, newName);
            cout << "Enter New Phone Number: ";
            getline(cin, newPhoneNumber);
            while (newPhoneNumber.length() != 10) {
                cout << "Not valid. \nPlease enter a valid 10-digit
number: ";
                getline(cin, newPhoneNumber);
            }
            addressBook.updateContact(oldName, newName,
newPhoneNumber);
            break;
        }
        case 6: {
            string name;
            cout << "Enter Name to Delete: ";
            getline(cin, name);
            addressBook.deleteContact(name);
            break;
        }
        case 7:
            cout << "Goodbye!" << endl;
            return 0;
        default:
            cout << "Invalid choice. Please enter a valid option."
<< endl;
        }
    }
    return 0;
}
```

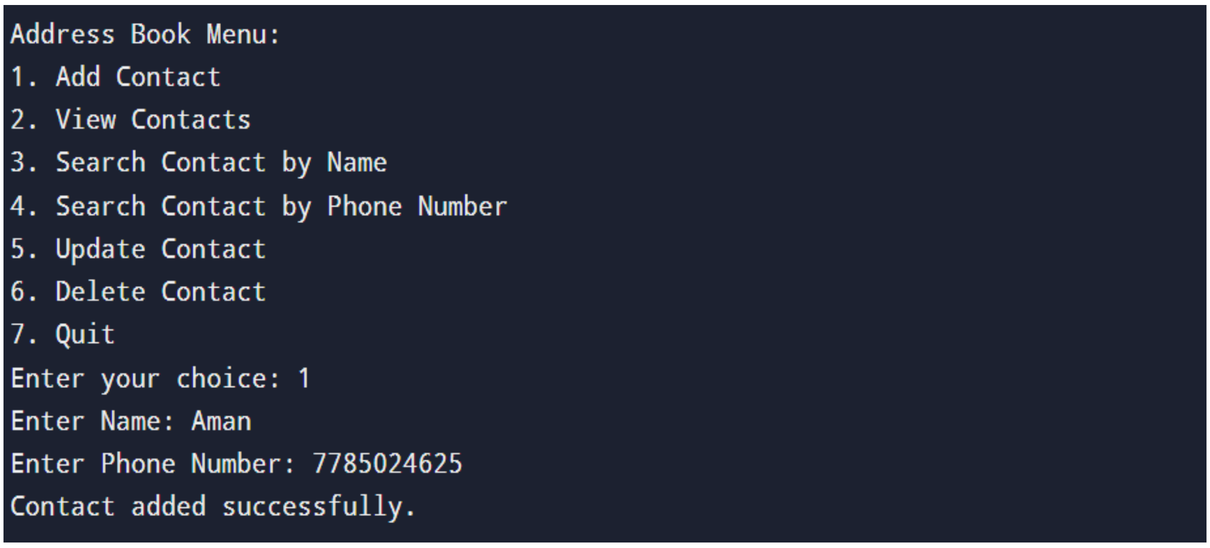
OUTPUT:

You will get different option for starting your contact manager.



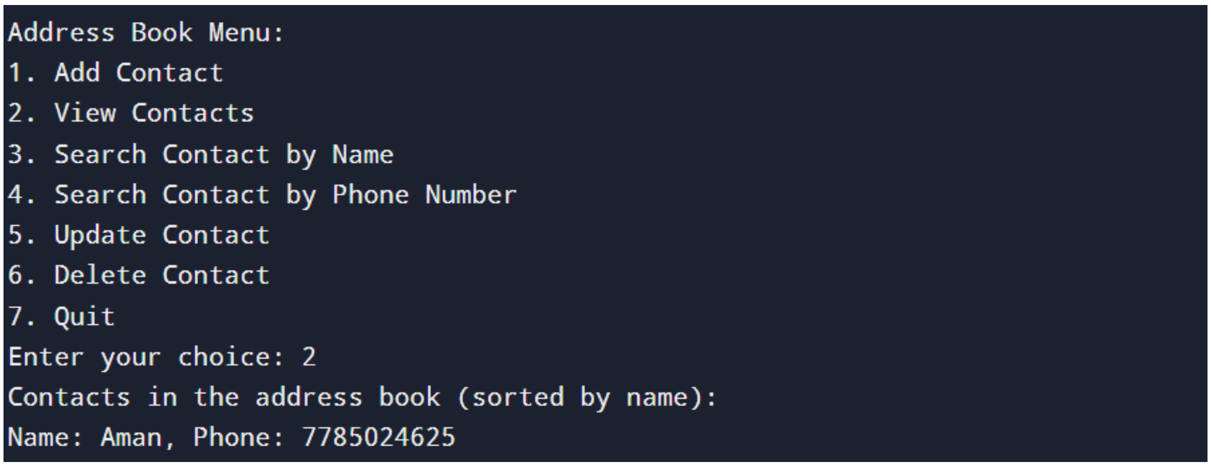
```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice:
```

You can add contact by giving name and mobile number by selecting option 1.



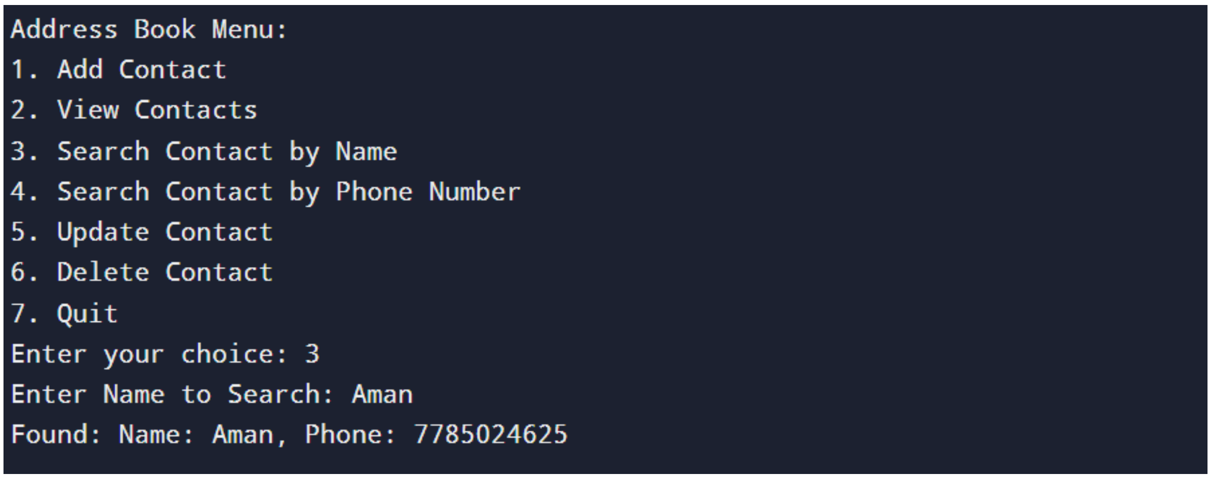
```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 1
Enter Name: Aman
Enter Phone Number: 7785024625
Contact added successfully.
```

You can see your contact list which you have saved before by selecting option 1, by selecting option 2.



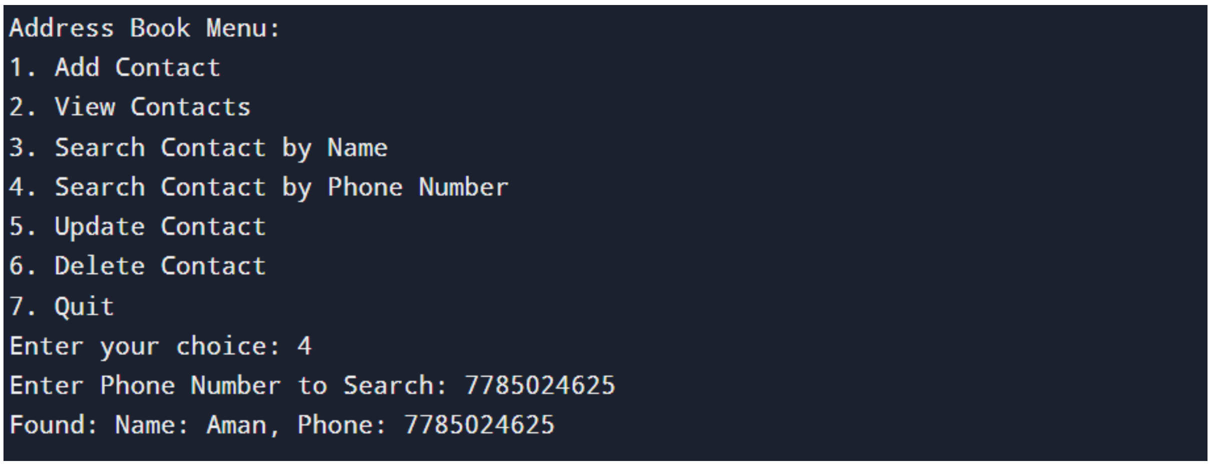
```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 2
Contacts in the address book (sorted by name):
Name: Aman, Phone: 7785024625
```

If you want to search any contact list by name, you can select option 3.



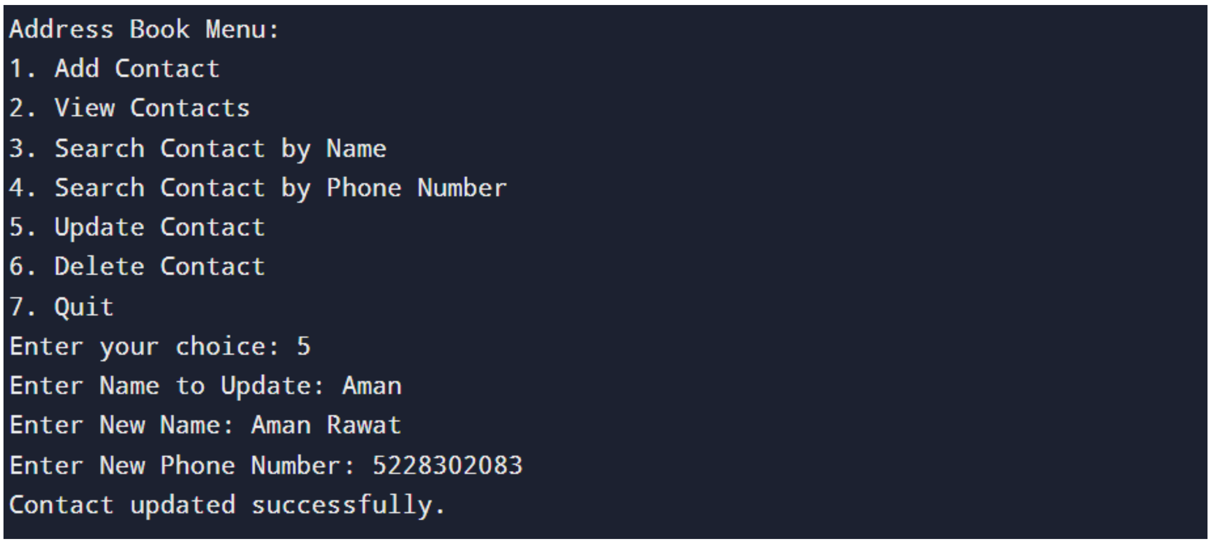
```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 3
Enter Name to Search: Aman
Found: Name: Aman, Phone: 7785024625
```

If you want to search any contact list by phone number, you can select option 4.



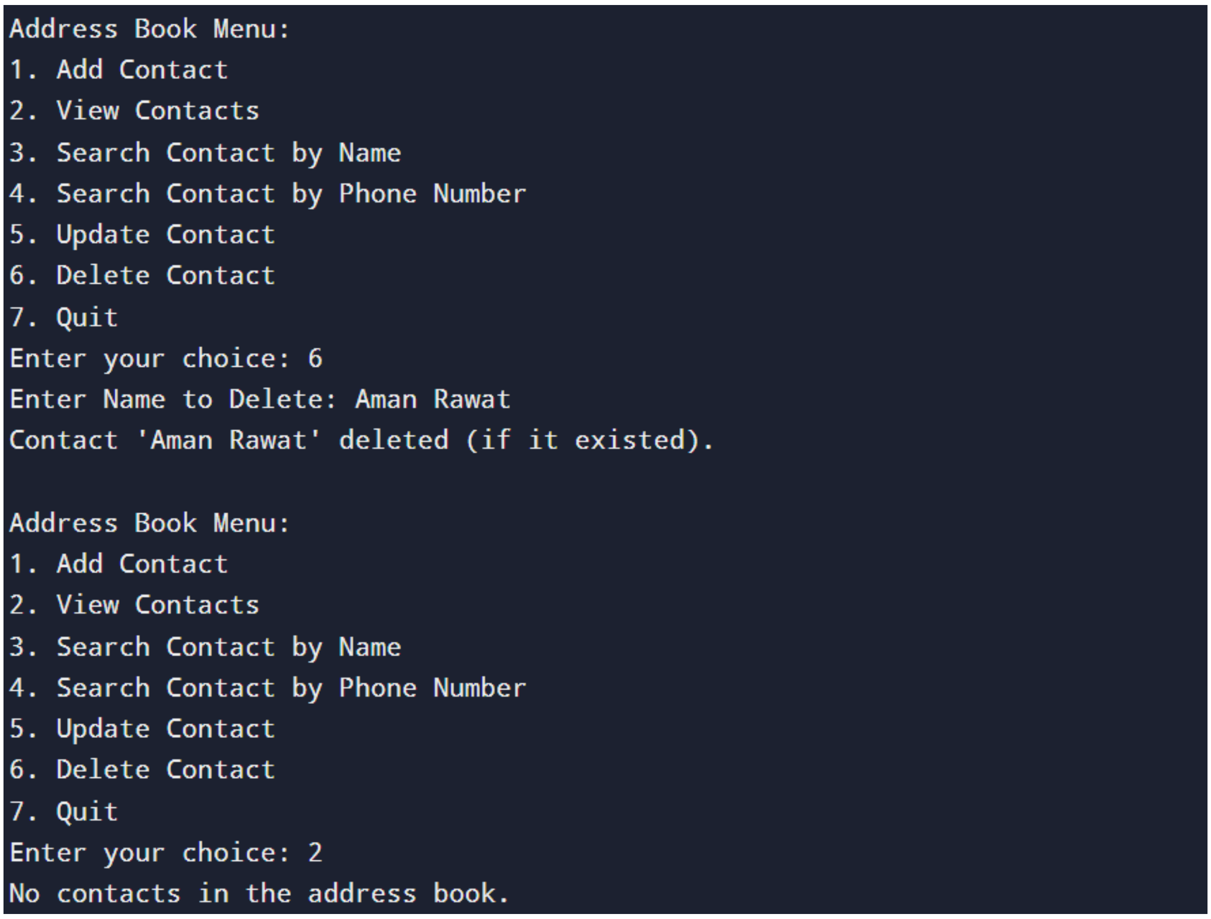
```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 4
Enter Phone Number to Search: 7785024625
Found: Name: Aman, Phone: 7785024625
```

You can update name and phone number in the contact list by selecting option 5.



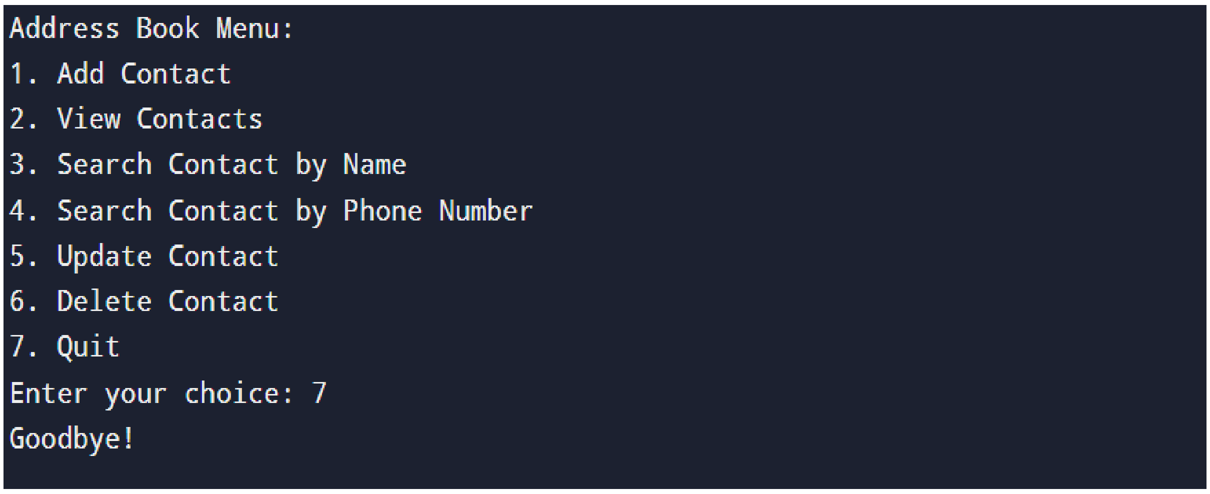
```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 5
Enter Name to Update: Aman
Enter New Name: Aman Rawat
Enter New Phone Number: 5228302083
Contact updated successfully.
```

If you want to delete any contact list you can select option 6.



```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 6
Enter Name to Delete: Aman Rawat
Contact 'Aman Rawat' deleted (if it existed).

Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 2
No contacts in the address book.
```

When you are done with your project, you can come out from the code by selecting option 7.



```
Address Book Menu:
1. Add Contact
2. View Contacts
3. Search Contact by Name
4. Search Contact by Phone Number
5. Update Contact
6. Delete Contact
7. Quit
Enter your choice: 7
Goodbye!
```

Functionality

The Contact Management System project is an efficient application designed to simplify the management of personal and professional contacts. This system provides users with the ability to store, search, and manage contact information seamlessly. The main features include:

- **Add Contact:** Users can add new contacts with details such as name and phone number. The system ensures that phone numbers are validated and correctly stored, preventing incorrect entries. If a contact with the same name already exists, the system updates the phone number.
- **View Contacts:** The system organizes and displays contacts in alphabetical order, making it easy for users to browse through their contact list. This is achieved using in-order traversal of a Binary Search Tree (BST).
- **Search by Name and Phone Number:** Users can search for contacts either by name or by phone number. The system uses the BST to locate contacts by name efficiently and a hash map for quick phone number searches, ensuring fast and accurate results.
- **Update Contact:** This feature enables users to update both the name and phone number of an existing contact in the address book. The system first searches for the contact by its current name. If found, the contact's name and phone number are updated in both the Binary Search Tree (BST) and the hash map002E
- **Delete Contact:** The system allows users to delete contacts by name, automatically adjusting the BST to maintain its structure. This feature ensures that contact lists remain up-to-date and free from unnecessary entries.
- **Contact Data Validation:** During both contact addition and search operations, the system ensures that the entered phone number is valid, specifically checking for a 10-digit format, thus maintaining data integrity.

Overall, this system improves the organization of contact information, automates retrieval, and simplifies contact management, offering a user-friendly interface for easy interaction.

REASON FOR CHOOSING DSA

- With rapid advancements in technology, programming has become an essential and highly sought-after skill for Software Developers. Everything from smart devices like TVs and ACs to traffic lights relies on programming to execute user commands.
- **Efficiency is Key:** To remain irreplaceable, one must be efficient. Mastery of Data Structures and Algorithms is a hallmark of a skilled Software Developer. Technical interviews at leading tech companies such as Google, Facebook, Amazon, and Flipkart often focus heavily on a candidate's proficiency in these areas. This focus is due to the significant impact that Data Structures and Algorithms have on enhancing a developer's problem-solving capabilities.
- **Learning Method:** This course offered comprehensive video lectures on all topics, which suited my preferred learning style. While books and notes have their importance, video lectures facilitate faster understanding through practical involvement.
- **Extensive Practice:** The course included over 200 algorithmic coding problems with video explanations, providing a rich resource for hands-on learning.
- **Structured Learning:** The course was organized with track-based learning and weekly assessments to continually test and improve my skills.

- **Productive Use of Time:** During the Covid-19 pandemic, this course was an excellent way for me to invest my time in learning, rather than wasting it on unproductive activities.
- **Lifetime Access:** The course offers lifetime access, allowing me to revisit the material whenever I need to refresh my knowledge, even after completing the initial training.

LEARNING OUTCOMES

Programming fundamentally revolves around the use of data structures and algorithms. Data structures manage and store data, while algorithms solve problems by processing that data.

Data Structures and Algorithms (DSA) explore solutions to common problems in depth, providing insights into the efficiency of different approaches. Understanding DSA empowers you to select the most effective methods for any given task. For example, if you need to search for a specific record in a database of 30,000 entries, you could choose between methods like Linear Search and Binary Search. In this scenario, Binary Search would be the more efficient choice due to its ability to quickly narrow down the possibilities.

Knowing DSA allows you to tackle problems with greater efficiency, making your code more scalable, which is crucial because:

- Time is valuable.
- Memory is costly.

DSA is also essential for excelling in technical interviews at product-based companies. Just as we prefer someone who can complete a task quickly and efficiently in our daily lives, companies look for developers who can solve complex problems effectively and with minimal resources.

For instance, if you're asked to calculate the sum of the first N natural numbers during an interview:

- One candidate might use a loop:

```
int sum = 0;
for (int n = 1; n <= N; n++) {
    sum += n;
}
```

- Another candidate might use the formula: Sum=N×(N+1)/2

  The latter approach is more efficient and would likely be favored by the interviewer.

Familiarity with data structures like Hash Maps, Trees, Graphs, and algorithms equips you to solve problems effectively, much like a mechanic uses the right tools to fix a car. Interviewers seek candidates who can select and apply the best tools to address the challenges they are presented with. Understanding the characteristics of different data structures helps you make informed decisions in problem-solving.

Practical Use of DSA: If you enjoy solving real-world problems, DSA is invaluable.

Consider these examples:

- **Organizing Books in a Library:** Suppose you need to find a book on Data Science. You'd first go to the technology section, then look for the Data Science subsection. If the books were not organized in this way, it would be much more challenging to find the specific book. Data structures help organize information in a computer system, enabling efficient processing based on the provided input.

- **Managing a Playlist:** If you have a playlist of your favorite songs, you might arrange them in a particular order, such as by genre or mood. A **Queue** could be used to manage the order in which the songs play, ensuring that they play in the sequence you prefer.
- **Navigating a City:** When finding the shortest route between two locations in a city, you could use a **Graph** data structure to represent the roads and intersections, with algorithms like Dijkstra’s or A* to find the optimal path.

These examples illustrate the importance of choosing the right data structure and algorithm for real-world problems, helping to solve them more efficiently.

Data Structures and Algorithms deepen your understanding of problems, allowing you to approach them more effectively and gain a better grasp of the world around you.

BIBLIOGRAPHY

- Coding Spoon Course
- Coding Spoon Website
- Google