

**SIX WEEKS SUMMER TRAINING REPORT**

**on**

**“Solving Competitive Problems Using Data Structures”**

Submitted by

**BONDALAPATI SAI VASISTA**

**Registration Number : 12109750**

**Program Name : B.Tech.(CSE)**

Under the guidance of

**Mr. Ravi Kant Sahu**

**Assistant Professor**

**School of Computer Science and Engineering**

**Lovely Professional University, Phagwara**(June-July,2023)

**DECLARATION**

I hereby declare that I have completed my six weeks summer training at **Human Resource Development Centre, Lovely Professional University, Phagwara (Punjab)** from **6th June 2023** to **17th July 2023** under the guidance of **Mr. Ravi Kant Sahu**. I declare that I have worked with full dedication during these six weeks of training and my learning outcomes fulfill the requirements of training for the award of degree of B.Tech.(Computer Science & Engineering) at Lovely Professional University, Phagwara.

B.SAI VASISTA

Registration No: 12109750

**ACKNOWLEDGEMENT**

To acknowledge all the persons who had helped in completing the training and project is not possible for any trainee. However, despite all that, it becomes a foremost responsibility of the trainee and the part of research ethics to acknowledge those who had played a significant role in the completion of the training and project.

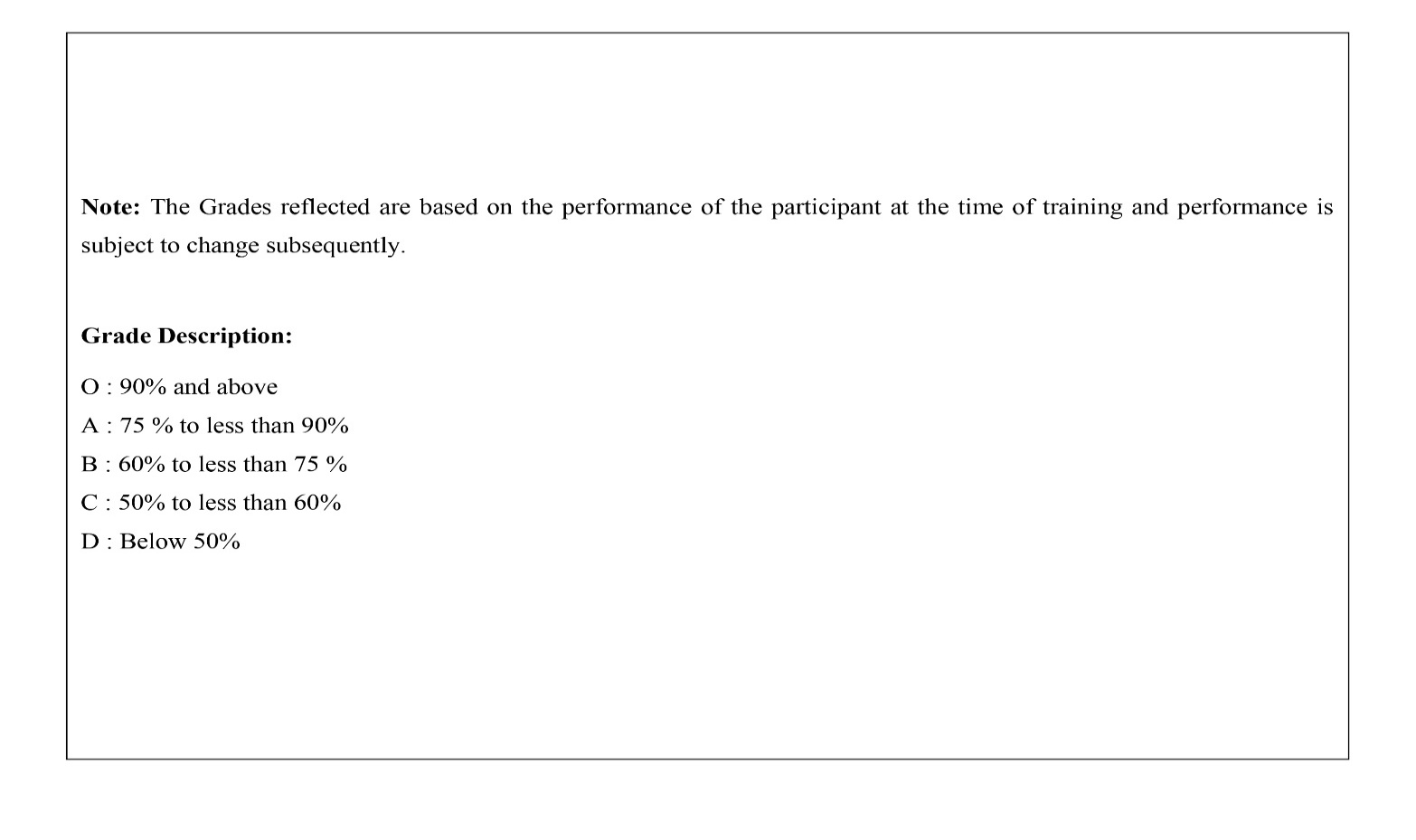
So, in the same sequence at very first, we would like to acknowledge Mr. Ravi Kant Sahu, the mentor of our Training. He has been instrumental in guiding and helping us while undergoing the planning phase of our project and helped clear the concepts related to the topics and project. Because the Training and Placement Cell of School of Computer Science & Engineering has allowed me to do the summer training, I would like to thank for the same. I would like to thank Human Resource Development Centre, LPU also for organizing such a wonderful summer training program.

Later, I would like to confer the flower of acknowledgment to our parents because of whom we got the existence in the world for the inception and the conception of this training and project. Rest all those people who helped us are not only a matter of acknowledgment but also authorized to share our success.

With regards

B.SAI VASISTA

**SUMMER TRAINING CERTIFICATE**

****

**TABLE OF CONTENTS**

|  |  |
| --- | --- |
| **Topics** | **Page Number** |
| Declaration | 2 |
| Acknowledgement | 3 |
| Table of content | 4 |
| Technology Learnt | 5-9 |
| Brief description of the Intern | 10-34 |
| Project Code | 35-36 |
| Snapshots | 37 |
| Conclusion | 38 |
| Reference | 39 |

**Technology Learned**

During the 6-week online summer training on " Data Structures with C++" participants gained a solid understanding of fundamental Data Structures and applied these concepts to solve standard competitive coding problems using C++.

**Data Structures**

Data structures are essential tools for organizing and manipulating data efficiently. They provide specialized formats for storing and retrieving information on a computer system. In the course, participants delved into the following key aspects of data structures:

Classification of Data Structures: Participants were introduced to the various types of data structures available, including arrays, linked lists, stacks, queues, trees, graphs, and heaps. Understanding these classifications laid the groundwork for selecting the appropriate structure for different problem scenarios.

Complexity Analysis: Participants learned about asymptotic notations for analyzing the time and space complexity of algorithms. This knowledge is crucial for evaluating the efficiency of algorithms and making informed decisions about algorithm selection.

Array Operations: The course covered array basics, traversal techniques, and problems based on array traversal. Participants also delved into array insertion and deletion, understanding the underlying mechanisms for these operations.

Searching Algorithms: Linear and binary search operations were explored, including solving problems related to searching within arrays. Binary search, in particular, is a key technique for optimizing search operations.

Sorting Techniques: Bubble sort, insertion sort, and selection sort were discussed in detail. Participants learned how these techniques function and their relative advantages and disadvantages.

Linked Lists: The concept of linked lists was introduced, including their types and applications. Participants worked with linked lists to perform searching, insertion, and deletion operations.

Stacks and Queues: The basics of stacks and queues were covered, along with their applications. Participants learned how to solve competitive problems involving stacks and queues.

Recursion: The concept of recursion was explored, including solving standard problems using recursive techniques. Competitive problems that involved recursion and stack handling were also tackled.

Binary Trees: Participants gained insights into binary trees, their types, basic terminologies, traversal techniques, and searching strategies. They also learned how to solve competitive problems related to binary trees.

Heaps: Heap data structures were introduced, including building and maintaining heaps, along with the heap sort algorithm.

Graphs: The basics of graphs, terminologies, and traversal techniques (BFS and DFS) were covered, as well as algorithms like Prim's and Kruskal's for Minimum Spanning Trees.

**Classification of Data Structures**

Data structures are fundamental components in computer science, serving as building blocks for efficient storage, organization, and manipulation of data. They can be classified into various categories based on their characteristics and applications. This classification provides programmers with a toolbox of options, allowing them to choose the most suitable structure for a given problem. Let's explore the primary types of data structures along with associated algorithms:

1. Arrays:

Arrays are one of the simplest and most common data structures. They store elements of the same data type in contiguous memory locations. The key feature of arrays is their constant-time access to elements using indices. Algorithms associated with arrays include:

Traversal: Accessing each element in an array sequentially.

Search: Linear search and binary search algorithms to find an element's position.

Insertion and Deletion: Shifting elements to accommodate insertions or deletions.

2. Linked Lists:

Linked lists consist of nodes where each node holds data and a reference to the next node. They provide dynamic memory allocation and efficient insertions and deletions. Algorithms related to linked lists include:

Traversal: Iterating through nodes sequentially.

Insertion and Deletion: Adding or removing nodes at various positions.

Searching: Scanning the list to find a specific value.

3. Stacks and Queues:

Stacks and queues are linear data structures with specific insertion and removal policies:

Stacks: Follow the Last-In-First-Out (LIFO) principle.

Queues: Follow the First-In-First-Out (FIFO) principle.

Algorithms associated with stacks and queues include:

Push and Pop (Stacks): Adding and removing elements from the top of the stack.

Enqueue and Dequeue (Queues): Adding and removing elements from the front and rear, respectively.

4. Trees:

Trees are hierarchical structures with a root node and child nodes. They're used for representing hierarchical relationships. Algorithms related to trees include:

Binary Trees: Traversals like in-order, pre-order, and post-order.

Binary Search Trees (BST): Insertion, deletion, and searching operations while maintaining the BST property.

Balanced Trees: Algorithms like AVL and Red-Black trees for balancing.

5. Graphs:

Graphs consist of nodes and edges representing relationships between nodes. They can be used to model various real-world scenarios. Algorithms associated with graphs include:

Graph Traversal: Breadth-First Search (BFS) and Depth-First Search (DFS).

Minimum Spanning Tree: Algorithms like Prim's and Kruskal's.

Shortest Path: Dijkstra's and Bellman-Ford algorithms.

6. Heaps:

Heaps are specialized trees that satisfy the heap property. They're commonly used for implementing priority queues. Algorithms related to heaps include:

Heapify: Converting an array into a heap structure.

Heap Sort: Sorting elements using the heap data structure.

Understanding the classification of data structures empowers programmers to choose the appropriate structure for solving specific problems efficiently. Each type has its own strengths and weaknesses, making it crucial to analyze problem requirements before making a decision.

**Brief description of the Intern**

**Complexity Analysis**

In the realm of computer science, understanding the efficiency of algorithms is paramount. Complexity analysis provides a systematic approach to evaluate the performance of algorithms as the input size grows. It allows us to predict how algorithms will behave on larger inputs and aids in making informed choices when selecting algorithms for various tasks.

**Asymptotic Notations:**

Asymptotic notations are mathematical tools used to describe the upper and lower bounds of algorithm performance in terms of input size. The most commonly used asymptotic notations are:

* Big O Notation (O): Represents the upper bound of an algorithm's running time. It gives an upper limit on the growth rate of the algorithm.
* Omega Notation (Ω): Represents the lower bound of an algorithm's running time. It provides a lower limit on the growth rate.
* Theta Notation (Θ): Denotes both upper and lower bounds, indicating that the algorithm's running time grows at a specific rate.

**Importance of Complexity Analysis:**

1. Complexity analysis enables us to:

* Predict Performance: By analyzing the complexity of an algorithm, we can estimate how it will perform as the input size increases. This prediction is crucial for making informed choices in algorithm selection.
* Optimize Algorithms: Understanding the bottlenecks of an algorithm helps in identifying areas for optimization, leading to improved efficiency.
* Compare Algorithms: Complexity analysis provides a standard framework for comparing different algorithms solving the same problem.
* Scale for Large Inputs: Efficient algorithms are essential when dealing with large datasets or resource-constrained environments.

1. **Common Complexity Classes and Associated Algorithms:**

* Constant Time (O(1)): Algorithms with constant time complexity have a consistent runtime, regardless of the input size. Example: accessing an element in an array using an index.
* Linear Time (O(n)): Algorithms with linear time complexity have a runtime proportional to the input size. Example: traversing an array or a linked list.
* Logarithmic Time (O(log n)): Algorithms with logarithmic time complexity often divide the input size in half at each step. Example: binary search in a sorted array.
* Quadratic Time (O(n^2)): Algorithms with quadratic time complexity have runtimes proportional to the square of the input size. Example: nested loops.
* Linearithmic Time (O(n log n)): Algorithms with linearithmic time complexity often arise in divide-and-conquer algorithms like merge sort and heap sort.
* Polynomial Time (O(n^k)): Polynomial time complexity arises in algorithms with nested loops where the number of loops is constant (k). Example: matrix multiplication using the Strassen algorithm.
* Exponential Time (O(2^n)): Algorithms with exponential time complexity have runtimes that grow rapidly with input size. Example: naive recursive solutions to certain problems.
* Factorial Time (O(n!)): Algorithms with factorial time complexity have runtimes that grow even faster than exponential algorithms. Example: generating all permutations of a set.

1. **Algorithms for Complexity Analysis:**

* Counting Operations: By analyzing the number of basic operations executed by an algorithm, we can estimate its complexity.
* Recurrence Relations: For recursive algorithms, recurrence relations help express the runtime in terms of smaller inputs.
* Master Theorem: A formula for solving recurrence relations that arise in divide-and-conquer algorithms.
* Best, Worst, and Average Cases: Analyzing an algorithm's performance in different scenarios helps provide a comprehensive view of its complexity.
* Complexity analysis plays a pivotal role in designing efficient algorithms and predicting their behavior. It empowers programmers to make informed choices that optimize performance and resource utilization, a vital skill in the world of software development.

**Array Operations**

Arrays are fundamental data structures that store a collection of elements in contiguous memory locations. They provide a basis for various algorithms and operations. Understanding array operations is essential for efficient data manipulation and problem-solving. Let's delve into the key operations associated with arrays and the algorithms behind them.

**1. Array Traversal:**

Array traversal involves visiting each element of an array to perform a specific task. Traversal is the foundation of many algorithms and is used extensively in data processing and analysis.

Algorithms:

Linear Traversal: Iterating through the array sequentially from the first element to the last.

Reverse Traversal: Iterating through the array in reverse order, from the last element to the first.

Skipping Elements: Traversing every alternate or specific subset of elements.

2**. Searching in Arrays:**

Searching is the process of locating a specific element within an array. Various algorithms are employed to efficiently find the desired element.

Algorithms:

Linear Search: Iterating through the array element by element until the target element is found.

Binary Search: Applicable only to sorted arrays, this algorithm compares the target element with the middle element and eliminates half of the remaining elements in each step.

3. **Insertion in Arrays:**

Inserting an element into an array involves adding a new element at a specific position. This operation may require shifting elements to accommodate the new entry.

Algorithms:

Insertion at End: Adding an element to the end of the array is straightforward and does not require shifting.

Insertion at Specific Position: Inserting an element at a particular index involves shifting subsequent elements to make space.

4. **Deletion from Arrays:**

Removing an element from an array necessitates shifting subsequent elements to fill the gap left by the removed element.

Algorithms:

Deletion by Value: Searching for the element to be deleted, removing it, and shifting remaining elements.

Deletion by Index: Directly removing the element at the specified index and shifting subsequent elements.

5. **Dynamic Arrays:**

Dynamic arrays allow for dynamic resizing, accommodating varying numbers of elements. When the array reaches capacity, a new larger array is created, and elements are copied.

Algorithms:

Resize Strategy: Deciding how much larger the new array should be.

Copying Elements: Moving elements from the old array to the new array during resizing.

6. **Array Manipulation Techniques:**

Arrays serve as a foundation for various problem-solving techniques:

Algorithms:

Two-Pointer Technique: Using two pointers to traverse or manipulate an array simultaneously.

Sliding Window Technique: Utilizing a fixed-size window to process or analyze subarrays.

Prefix Sum: Precomputing cumulative sums to quickly calculate subarray sums.

Array operations are pivotal in algorithm design and problem-solving. A solid grasp of these techniques empowers programmers to efficiently manipulate data, solve complex problems, and optimize performance.

**Searching Algorithms**

Searching algorithms are essential tools in computer science that allow us to find specific elements within a dataset quickly and effectively. In this section, we will delve into two primary searching algorithms: Linear Search and Binary Search. These algorithms play a crucial role in various applications, from information retrieval to problem-solving.

**Part 1: Linear Search**

Linear Search is a straightforward approach to finding a target element in an array by sequentially examining each element until a match is found.

Algorithm:

* Start from the first element of the array.
* Compare the target element with the current element.
* If a match is found, return the index of the element.
* If no match is found, move to the next element and repeat the comparison.
* Continue until the end of the array is reached or a match is found.

Complexity:

* Time Complexity: O(n) - In the worst case, linear search may require examining all n elements in the array.
* Space Complexity: O(1) - Linear search only requires a constant amount of extra space.

Code :

#include <iostream>

int linearSearch(int arr[], int size, int target) {

for (int i = 0; i < size; ++i) {

if (arr[i] == target) {

return i; // Return the index where the target element was found

}

}

return -1; // Return -1 if the target element was not found in the array

}

**Part 2: Binary Search**

Binary Search is a more efficient searching algorithm that works specifically with sorted arrays. It employs a divide-and-conquer strategy to quickly narrow down the search space.

Algorithm:

* Initialize pointers at the beginning and end of the sorted array.
* Calculate the middle index.
* Compare the middle element with the target element.
* If the middle element is equal to the target, return its index.
* If the middle element is less than the target, narrow the search space to the right half of the array.
* If the middle element is greater than the target, narrow the search space to the left half of the array.
* Repeat steps 2-6 until the target element is found or the search space is exhausted.

Complexity:

* Time Complexity: O(log n) - In each step, binary search reduces the search space by half.
* Space Complexity: O(1) - Binary search requires a constant amount of extra space.

Code :

int binarySearch(int arr[], int size, int target) {

int left = 0;

int right = size - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

return mid; // Return the index where the target element was found

} else if (arr[mid] < target) {

left = mid + 1; // Adjust left pointer if target is on the right side

} else {

right = mid - 1; // Adjust right pointer if target is on the left side

}

}

return -1; // Return -1 if the target element was not found in the array

}

Applications and Use Cases:

Linear Search: Suitable for small unsorted arrays or when the element's position is not known.

Binary Search: Highly efficient for large sorted arrays, as it significantly reduces the search space in each step.

Considerations:

Sorted Array: Binary search requires the array to be sorted in ascending or descending order.

Algorithm Choice: Choose between linear and binary search based on the size, ordering, and nature of the dataset.

Searching algorithms are foundational concepts in computer science, influencing the efficiency and performance of various applications. By understanding these algorithms and their intricacies, programmers can make informed decisions to optimize data retrieval operations.

**Sorting Techniques**

Sorting is a fundamental operation in computer science that involves arranging elements in a specific order. This section will delve into three primary sorting techniques: Bubble Sort, Insertion Sort, and Selection Sort. Understanding these techniques is essential for efficient data organization and manipulation.

**Part 1: Bubble Sort**

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

Algorithm:

* Start from the beginning of the array.
* Compare the current element with the next element.
* If the current element is greater than the next element, swap them.
* Move to the next pair of elements and repeat steps 2 and 3.
* Continue iterating through the array until no swaps are needed.

Complexity:

Time Complexity: O(n^2) - In the worst and average cases, bubble sort requires comparing and swapping elements for each pair.

Space Complexity: O(1) - Bubble sort requires a constant amount of extra space.

Code :

void bubbleSort(int arr[], int size) {

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

for (int j = 0; j < size - i - 1; ++j) {

if (arr[j] > arr[j + 1]) {

// Swap arr[j] and arr[j+1]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

**Part 2: Insertion Sort**

Insertion Sort is an efficient sorting algorithm that builds a sorted array one element at a time by inserting each element into its correct position.

Algorithm:

* Start from the second element and assume the first element is already sorted.
* Compare the current element with the previous elements in the sorted section.
* If the current element is smaller, shift the larger elements to the right.
* Insert the current element into the correct position.
* Repeat steps 2-4 for all remaining unsorted elements.

Complexity:

Time Complexity: O(n^2) - In the worst case, each element may need to be compared with all previous elements.

Space Complexity: O(1) - Insertion sort requires a constant amount of extra space.

Code :

void insertionSort(int arr[], int size) {

for (int i = 1; i < size; ++i) {

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1], that are greater than key,

// to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

**Part 3: Selection Sort**

Selection Sort is a sorting algorithm that repeatedly finds the minimum element from the unsorted part of the array and places it at the beginning.

Algorithm:

* Start from the beginning of the array.
* Find the minimum element in the unsorted section.
* Swap the minimum element with the first element in the unsorted section.
* Move the boundary between the sorted and unsorted sections one position to the right.
* Repeat steps 2-4 until the entire array is sorted.

Complexity:

* Time Complexity: O(n^2) - Selection sort requires finding the minimum for each element.
* Space Complexity: O(1) - Selection sort requires a constant amount of extra space.

Code :

void selectionSort(int arr[], int size) {

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

int minIndex = i;

// Find the index of the minimum element in the unsorted portion

for (int j = i + 1; j < size; ++j) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// Swap the minimum element with the first element in the unsorted portion

int temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

}

Applications and Use Cases:

* Bubble Sort: Mainly used for educational purposes due to its simplicity.
* Insertion Sort: Suitable for small datasets and almost sorted arrays.
* Selection Sort: Useful when the number of swaps needs to be minimized.

Sorting techniques are essential tools for programmers to organize data efficiently and optimize various operations. By grasping these techniques and their intricacies, programmers can enhance their problem-solving skills and algorithmic thinking.

**Linked Lists**

Linked lists are dynamic data structures that allow efficient insertion and deletion of elements. In this section, we will delve into various aspects of linked lists, including their introduction, types, traversal, insertion, and deletion operations. Understanding linked lists is crucial for mastering data organization and manipulation.

**Part 1: Introduction to Linked Lists**

A linked list is a linear data structure that consists of a sequence of elements, each connected to the next using pointers. Linked lists provide flexibility in memory allocation and are used when the size of the dataset is not fixed.

* Singly Linked List:

A singly linked list comprises nodes, each containing data and a reference to the next node.

Code :

struct Node {

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class LinkedList {

private:

Node\* head;

public:

LinkedList() : head(nullptr) {}

// Function to insert a new node at the end of the linked list

void append(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

return;

}

Node\* current = head;

while (current->next) {

current = current->next;

}

current->next = newNode;

}

// Function to display the linked list

void display() {

Node\* current = head;

while (current) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

}

};

* Doubly Linked List:

A doubly linked list adds a reference to the previous node in addition to the next node.

Code :

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int value) : data(value), prev(nullptr), next(nullptr) {}

};

class DoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

// Function to insert a new node at the end of the doubly linked list

void append(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

// Function to display the doubly linked list from head to tail

void displayForward() {

Node\* current = head;

while (current) {

std::cout << current->data << " ";

current = current->next;

}

std::cout << std::endl;

}

// Function to display the doubly linked list from tail to head

void displayBackward() {

Node\* current = tail;

while (current) {

std::cout << current->data << " ";

current = current->prev;

}

std::cout << std::endl;

}

};

* Circular Linked List:

In a circular linked list, the last node points back to the first node.

Code :

struct Node {

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class CircularLinkedList {

private:

Node\* head;

public:

CircularLinkedList() : head(nullptr) {}

// Function to insert a new node at the end of the circular linked list

void append(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* current = head;

while (current->next != head) {

current = current->next;

}

current->next = newNode;

newNode->next = head;

}

}

// Function to display the circular linked list

void display() {

if (!head) {

std::cout << "Circular Linked List is empty." << std::endl;

return;

}

Node\* current = head;

do {

std::cout << current->data << " ";

current = current->next;

} while (current != head);

std::cout << std::endl;

}

};

**Part 2: Linked List Traversal**

Linked list traversal involves visiting each node to perform a specific task. Traversal is fundamental for various linked list operations.

Algorithm:

* Start from the head (first node) of the linked list.
* Process the data in the current node.
* Move to the next node using the next pointer.
* Repeat steps 2-3 until the end of the linked list is reached.

**Part 3: Insertion in Linked Lists**

Insertion in linked lists involves adding a new node at a specific position. This operation adjusts the references of neighboring nodes.

Algorithm:

* Create a new node with the desired data.
* Update the next pointers to connect the new node appropriately.
* Adjust the previous and next pointers of neighboring nodes.

Types of Insertion:

* Insertion at the Beginning: Update the head to point to the new node.
* Insertion at the End: Traverse to the last node and update its next pointer.
* Insertion at a Specific Position: Traverse to the desired position and adjust pointers.

**Part 4: Deletion from Linked Lists**

Deletion in linked lists involves removing a node from a specific position. This operation adjusts the references of neighboring nodes.

Algorithm:

* Traverse to the node to be deleted.
* Update the next and previous pointers of neighboring nodes to bypass the node to be deleted.
* Delete the node.

Types of Deletion:

* Deletion at the Beginning: Update the head to point to the next node.
* Deletion at the End: Traverse to the second-to-last node and update its next pointer.
* Deletion at a Specific Position: Traverse to the node and adjust pointers.

Applications and Use Cases:

* Singly Linked List: Useful for applications where traversal in one direction is sufficient.
* Doubly Linked List: Beneficial when bidirectional traversal is required.
* Circular Linked List: Suitable for circular navigation or looping scenarios.
* Linked lists offer dynamic memory allocation and efficient insertion and deletion. By mastering linked list operations, programmers can enhance their ability to manipulate data structures effectively.

**Stacks and Queues**

Stacks and queues are linear data structures that play pivotal roles in managing data flow and maintaining order. In this section, we will delve into various aspects of stacks and queues, including their introduction, operations, and algorithms. Understanding stacks and queues is crucial for mastering data management and efficient processing.

**Part 1: Introduction to Stacks**

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It operates like a stack of objects, where the last item added is the first one to be removed.

Implementation:

* Array-based Stack: Using an array to store elements, managing indices to simulate the stack behavior.
* Linked List-based Stack: Using a linked list to manage stack operations.

**Part 2: Stack Operations and Algorithms**

Stacks support fundamental operations that maintain their LIFO nature.

Operations:

Push: Adding an element to the top of the stack.

Pop: Removing the top element from the stack.

Peek (Top): Viewing the top element without removing it.

isEmpty: Checking if the stack is empty.

Algorithm: Balancing Parentheses

* Initialize an empty stack.
* Traverse the expression:
* If an opening parenthesis is encountered, push it onto the stack.
* If a closing parenthesis is encountered, pop from the stack if a matching opening parenthesis is found.
* If the stack is empty at the end, the expression is balanced.

**Part 3: Introduction to Queues**

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. It operates like a line of people waiting, where the first person to join the line is the first to be served.

Implementation:

* Array-based Queue: Using an array to store elements, managing indices for enqueue and dequeue operations.
* Linked List-based Queue: Using a linked list to manage queue operations.

**Part 4: Queue Operations and Algorithms**

Queues support fundamental operations that maintain their FIFO nature.

Operations:

* Enqueue: Adding an element to the rear (end) of the queue.
* Dequeue: Removing the front (first) element from the queue.
* Front (Peek): Viewing the front element without removing it.
* Rear: Viewing the rear element without removing it.
* isEmpty: Checking if the queue is empty.

Algorithm: Breadth-First Search (BFS)

* Initialize a queue and enqueue the starting node.
* While the queue is not empty:
* Dequeue a node.
* Process the node's data.
* Enqueue unvisited neighbors.
* BFS explores nodes level by level, making it ideal for finding shortest paths.

Applications and Use Cases:

Stacks: Used for undo-redo mechanisms, function calls in programming, and parsing expressions.

Queues: Applied in scheduling tasks, breadth-first traversal, and implementing buffering systems.

Stacks and queues provide essential tools for managing data flow and organizing processes. By mastering these structures and their operations, programmers can efficiently handle data and optimize various operations.

**Recursion**

Recursion is a powerful concept in computer science that allows a function to call itself as part of its execution. In this section, we will delve into various aspects of recursion, including its introduction, the mechanics behind recursion, and algorithms that leverage this technique. Understanding recursion is crucial for mastering complex problem-solving and algorithmic thinking.

**Part 1: Introduction to Recursion**

Recursion is a programming technique where a function calls itself to solve a smaller instance of the problem at hand. It enables elegant solutions to problems that can be divided into smaller subproblems.

Key Elements:

* Base Case: A condition that determines when the recursion stops.
* Recursive Case: The part of the function that calls itself with a modified input.

**Part 2: Basics of Recursion**

Understanding the mechanics of recursion is essential to utilize this technique effectively.

Mechanics:

* Divide the Problem: Break the problem into smaller, similar subproblems.
* Solve the Base Case: Provide a solution for the smallest subproblem.
* Combine Solutions: Use the solutions of smaller subproblems to solve the original problem.

Pros and Cons:

* Pros: Elegant solutions for complex problems, often mimicking the problem's structure.
* Cons: Can lead to excessive memory usage and performance issues if not implemented carefully.

**Part 3: Recursive Algorithms**

Recursion is employed in various algorithms to solve problems in a concise and efficient manner.

Algorithm: Factorial Calculation

* Base Case: If the input is 0 or 1, return 1.
* Recursive Case: Multiply the input with the factorial of (input - 1).

Algorithm: Fibonacci Sequence

* Base Case: If the input is 0 or 1, return the input.
* Recursive Case: Sum the results of the function for inputs (input - 1) and (input - 2).

**Part 4: Applying Recursion in Problem Solving**

Recursion finds applications in algorithms for various problem domains.

Algorithm: Tower of Hanoi

* Base Case: If there's only one disc, move it to the destination peg.
* Recursive Case: Move n-1 discs from the source peg to the auxiliary peg.
* Move the remaining disc to the destination peg.
* Move the n-1 discs from the auxiliary peg to the destination peg.

Algorithm: Binary Search using Recursion

* Base Case: If the array has one element, check if it's the target.
* Recursive Case: Divide the array and recursively search the appropriate subarray.

Applications and Use Cases:

* Mathematics: Factorials, Fibonacci sequence, and mathematical series.
* Algorithms: Searching (binary search), sorting (merge sort), and graph traversal (depth-first search).
* Recursion is a powerful technique that simplifies complex problems by breaking them down into smaller components. By mastering recursion, programmers can enhance their problem-solving skills and tackle a wide range of computational challenges.

**Binary Trees**

Binary trees are hierarchical data structures that play a significant role in various algorithms and applications. In this section, we will delve into the introduction, operations, and algorithms related to binary trees. Understanding binary trees is crucial for mastering data organization and efficient traversal.

**Part 1: Introduction to Binary Trees**

A binary tree is a hierarchical structure composed of nodes, each containing a value and references to its left and right child nodes. Binary trees provide efficient storage and retrieval of data, and they form the basis for many other tree structures.

Properties:

* Root: The top node of the tree.
* Parent Node: A node with children.
* Child Nodes: Nodes connected to a parent node.
* Leaf Nodes: Nodes with no children.
* Depth: The level of a node in the tree.

**Part 2: Binary Tree Traversal and Algorithms**

Binary tree traversal involves visiting nodes in a specific order to process their data.

In-Order Traversal:

* Traverse the left subtree.
* Process the current node's data.
* Traverse the right subtree.

Pre-Order Traversal:

* Process the current node's data.
* Traverse the left subtree.
* Traverse the right subtree.

Post-Order Traversal:

* Traverse the left subtree.
* Traverse the right subtree.
* Process the current node's data.

Algorithm: Binary Tree Height

* Base Case: If the node is null, return -1.
* Recursive Case: Return the maximum height of the left and right subtrees + 1.

**Part 3: Introduction to Heaps**

A heap is a specialized tree-based data structure that satisfies the heap property. It is commonly used to implement priority queues.

Min-Heap Property:

The value of each node is greater than or equal to the values of its children.

Max-Heap Property:

The value of each node is less than or equal to the values of its children.

Part 4: Heap Operations and Algorithms

Heaps support various operations, with the primary focus on maintaining the heap property.

Operations:

* Insertion: Adding a new element to the heap and ensuring the heap property is maintained.
* Extraction: Removing the root element (minimum or maximum) and adjusting the heap.

Algorithm: Heap Sort

* Build a max-heap from the input array.
* Extract the maximum element (root) and place it at the end of the sorted array.
* Repeat step 2 until the heap is empty.

**Heaps**

Heaps are specialized tree-based data structures that enable efficient management of priority-related operations. In this section, we will delve into various aspects of heaps, including their introduction, types, operations, and algorithms. Understanding heaps is crucial for mastering data organization and optimizing priority-based tasks.

**Part 1: Introduction to Heaps**

A heap is a binary tree-based data structure that satisfies the heap property. The heap property dictates the relationship between parent and child nodes, making heaps suitable for priority queues and sorting algorithms.

Types of Heaps:

Max-Heap: Each parent node has a value greater than or equal to its children.

Min-Heap: Each parent node has a value less than or equal to its children.

**Part 2: Heap Operations**

Heaps support various operations that maintain the heap property.

Operations:

* Insertion: Adding an element to the heap and arranging it to satisfy the heap property.
* Extraction: Removing the root element (maximum in a max-heap or minimum in a min-heap) and restructuring the heap.
* Peek: Viewing the root element without removing it.
* Heapify: Rearranging the elements of an array to satisfy the heap property.

**Part 3: Heap Algorithms**

Heaps find applications in algorithms for tasks like sorting and efficient priority management.

Algorithm: Heap Sort

* Build Heap: Convert the input array into a max-heap.
* Sorting Phase: Repeatedly extract the maximum element (root) and place it at the end of the sorted array.
* Continue until the heap is empty.

Algorithm: Priority Queue

* Insertion: Add a new element to the heap.
* Extraction: Remove the root element (highest priority) and restructure the heap.
* Peek: View the root element without removal.

**Part 4: Applications of Heaps**

Heaps find applications in various scenarios where priority management is crucial.

Priority Queues:

* Job Scheduling: Manage tasks based on their priority.
* Dijkstra's Algorithm: Find the shortest path in a graph.

Heap Sort:

* In-Place Sorting: Efficiently sort large datasets.
* Selection of Kth Largest/Smallest: Find kth largest or smallest element in an array.

Considerations:

* Time Complexity: Heap operations take O(log n) time due to the balanced structure.
* Space Complexity: Heaps typically require O(n) space to store elements.

Heaps provide a structured approach to managing priority-based tasks and optimizing certain algorithms. By mastering heap operations and algorithms, programmers can enhance their ability to efficiently handle priority-related challenges.

**Graphs**

Graphs are versatile data structures used to model relationships between entities. In this section, we will delve into various aspects of graphs, including their introduction, types, operations, and algorithms. Understanding graphs is crucial for mastering complex network modeling and algorithmic design.

**Part 1: Introduction to Graphs**

A graph is a collection of nodes (vertices) and edges that connect pairs of nodes. Graphs are used to represent a wide range of relationships, from social networks to road networks.

Terminology:

* Vertex: A node in the graph.
* Edge: A connection between two vertices.
* Directed Edge: An edge with a direction.
* Weighted Edge: An edge with a numerical value (weight).

**Part 2: Types of Graphs**

Graphs can be categorized into various types based on their properties and relationships.

Undirected Graph:

Edges have no direction, and they simply connect two vertices.

Directed Graph (Digraph):

Edges have a direction, moving from one vertex to another.

Weighted Graph:

Edges have associated weights or costs.

**Part 3: Graph Operations**

Graphs support various operations that allow us to explore and manipulate their structure.

Traversal:

* Depth-First Search (DFS): Explore as far as possible along each branch before backtracking.
* Breadth-First Search (BFS): Explore level by level, visiting neighbors before their children.

Connectivity:

* Connected Graph: A graph where there is a path between any two vertices.
* Strongly Connected Digraph: Every vertex is reachable from any other vertex.

Minimum Spanning Tree:

* Prim's Algorithm: Build a tree by selecting edges with the smallest weight.
* Kruskal's Algorithm: Build a tree by selecting edges in increasing order of weight.

**Part 4: Graph Algorithms**

Graph algorithms solve various problems related to paths, connectivity, and more.

Algorithm: Dijkstra's Shortest Path

* Initialization: Set distances to all vertices to infinity except the start vertex (set distance to 0).
* Relaxation: For each vertex, update its distance if a shorter path is found through the current vertex.
* Priority Queue: Use a priority queue to select the vertex with the smallest distance.

Algorithm: Depth-First Search (DFS)

* Start at a vertex and mark it as visited.
* Visit an unvisited neighbor and repeat step 1.
* Backtrack if all neighbors are visited.

**Part 5: Applications of Graphs**

Graphs find applications in various domains, facilitating data representation and problem-solving.

* Social Networks: Modeling friendships and connections.
* Transportation Networks: Analyzing road, rail, or flight connections.
* Web Pages: Representing hyperlinks between web pages.
* Considerations:
* Complexity: Graph algorithms have varying time complexities, depending on the specific problem and algorithm used.
* Storage: Graphs can be represented using adjacency matrices or adjacency lists.

Graphs provide a powerful framework for representing complex relationships and solving intricate problems. By mastering graph operations and algorithms, programmers can navigate and analyze networks effectively.

**PROJECT CODE**

// Function to print the array or string

template<typename T>

void printData(const T& data) {

  for (const auto& element : data) {

    std::cout << element << " ";

  }

  std::cout << std::endl;

}

// Merge Sort

template<typename T>

void merge(T& data, int left, int mid, int right) {

  int i, j, k;

  int n1 = mid - left + 1;

  int n2 = right - mid;

  std::vector<typename T::value\_type> L(n1), R(n2);

  for (i = 0; i < n1; i++)

    L[i] = data[left + i];

  for (j = 0; j < n2; j++)

    R[j] = data[mid + 1 + j];

  i = 0;

  j = 0;

  k = left;

  while (i < n1 && j < n2) {

    if (L[i] <= R[j]) {

      data[k] = L[i];

      i++;

    } else {

      data[k] = R[j];

      j++;

    }

    k++;

  }

  while (i < n1) {

    data[k] = L[i];

    i++;

    k++;

  }

  while (j < n2) {

    data[k] = R[j];

    j++;

    k++;

  }

}

template<typename T>

void mergeSort(T& data, int left, int right) {

  if (left < right) {

    int mid = left + (right - left) / 2;

    mergeSort(data, left, mid);

    mergeSort(data, mid + 1, right);

    merge(data, left, mid, right);

  }

}

// Quick Sort

template<typename T>

int partition(T& data, int low, int high) {

  typename T::value\_type pivot = data[high];

  int i = low - 1;

  for (int j = low; j <= high - 1; j++) {

    if (data[j] < pivot) {

      i++;

      std::swap(data[i], data[j]);

    }

  }

  std::swap(data[i + 1], data[high]);

  return i + 1;

}

template<typename T>

void mergeSort(T& data, int left, int right) {

  if (left < right) {

    int mid = left + (right - left) / 2;

    mergeSort(data, left, mid);

    mergeSort(data, mid + 1, right);

    merge(data, left, mid, right);

  }

}

// Quick Sort

template<typename T>

int partition(T& data, int low, int high) {

  typename T::value\_type pivot = data[high];

  int i = low - 1;

  for (int j = low; j <= high - 1; j++) {

    if (data[j] < pivot) {

      i++;

      std::swap(data[i], data[j]);

    }

  }

  std::swap(data[i + 1], data[high]);

  return i + 1;

}

template<typename T>

void quickSort(T& data, int low, int high) {

  if (low < high) {

    int pi = partition(data, low, high);

    quickSort(data, low, pi - 1);

    quickSort(data, pi + 1, high);

  }

}

// Insertion Sort

template<typename T>

void insertionSort(T& data) {

  int n = data.size();

  for (int i = 1; i < n; i++) {

    typename T::value\_type key = data[i];

    int j = i - 1;

    while (j >= 0 && data[j] > key) {

      data[j + 1] = data[j];

      j--;  }

    data[j + 1] = key;

  }

}

// Deletion Sort

template<typename T>

void deletionSort(T& data) {

  int n = data.size();

  for (int i = 0; i < n - 1; i++) {

    int min\_idx = i;

    for (int j = i + 1; j < n; j++) {

      if (data[j] < data[min\_idx]) {

        min\_idx = j;

      }

    }

    std::swap(data[min\_idx], data[i]);

  }

}

// Bubble Sort

template<typename T>

void bubbleSort(T& data) {

  int n = data.size();

  for (int i = 0; i < n - 1; i++) {

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

      if (data[j] > data[j + 1]) {

        std::swap(data[j], data[j + 1]);

      }

    }

  }

}

// Selection Sort

template<typename T>

void selectionSort(T& data) {

  int n = data.size();

  for (int i = 0; i < n - 1; i++) {

    int min\_idx = i;

    for (int j = i + 1; j < n; j++) {

      if (data[j] < data[min\_idx]) {

        min\_idx = j;

      }

    }

    std::swap(data[min\_idx], data[i]);

  }

}

// Deletion Sort

template<typename T>

void deletionSort(T& data) {

  int n = data.size();

  for (int i = 0; i < n - 1; i++) {

    int min\_idx = i;

    for (int j = i + 1; j < n; j++) {

      if (data[j] < data[min\_idx]) {

        min\_idx = j;

      }

    }

    std::swap(data[min\_idx], data[i]);

  }

}

// Heap Sort

template<typename T>

void heapify(T& data, int n, int i) {

  int largest = i;

  int left = 2 \* i + 1;

  int right = 2 \* i + 2;

  if (left < n && data[left] > data[largest]) {

    largest = left;

  }

  if (right < n && data[right] > data[largest]) {

    largest = right;

  }

  if (largest != i) {

    std::swap(data[i], data[largest]);

    heapify(data, n, largest);

  }

}

template<typename T>

void heapSort(T& data) {

  int n = data.size();

  for (int i = n / 2 - 1; i >= 0; i--) {

    heapify(data, n, i);

  }

  for (int i = n - 1; i >= 0; i--) {

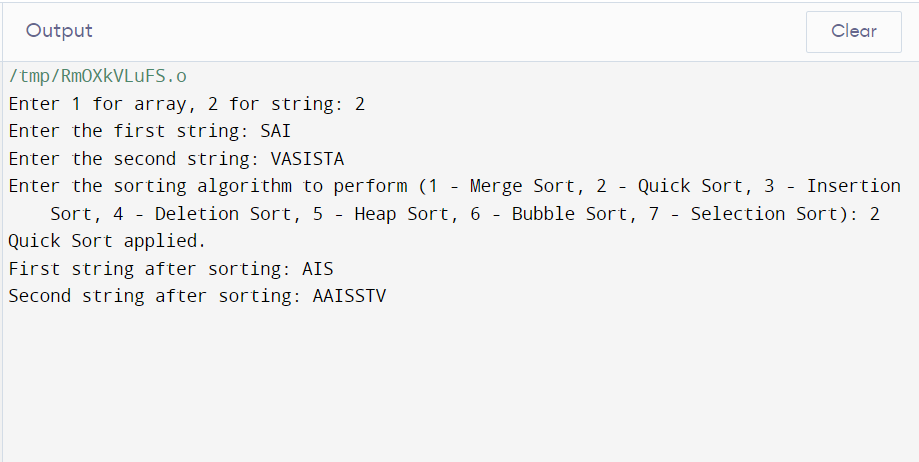
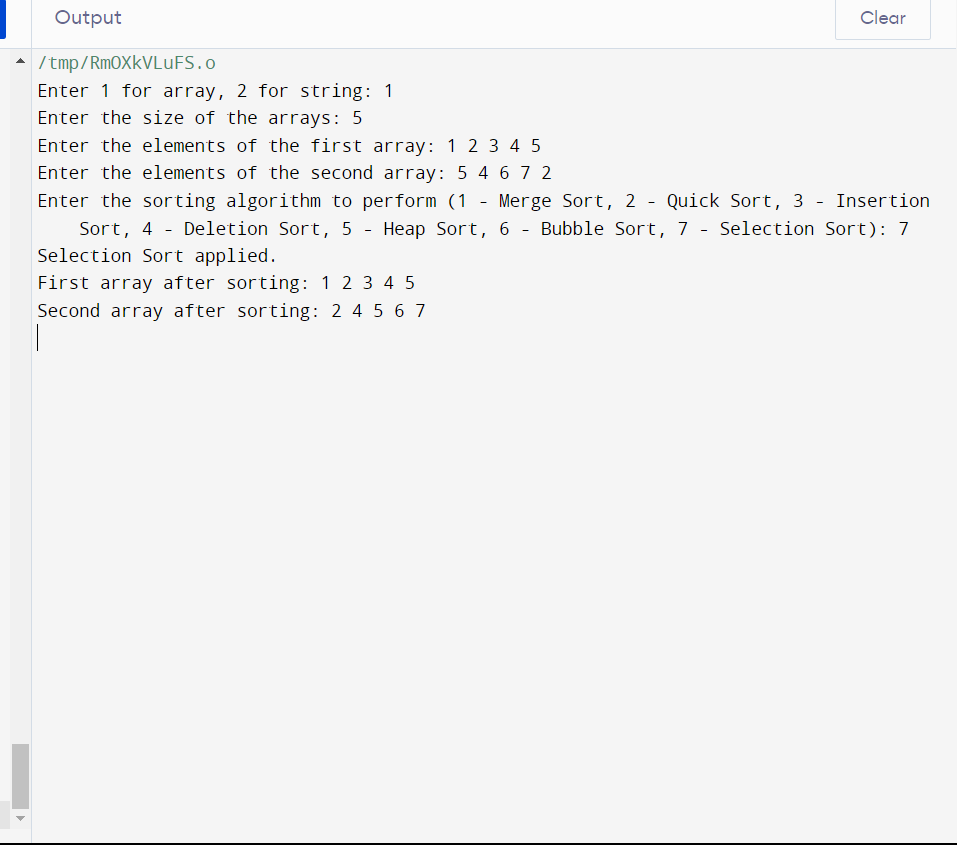
    std::swap(data[0], data[i]);

    heapify(data, i, 0);

  }

}

**SNAPSHOTS**

****

**Conclusion**

The C++ Programming and Data Structures & Algorithms summer course at LPU has been an enriching journey into the world of programming and computational problem-solving. Throughout the course, participants have been exposed to a comprehensive curriculum that covered a wide spectrum of topics, from the foundational concepts of data structures to advanced algorithms for efficient data manipulation and traversal.

The journey began with a solid foundation in C++ programming, setting the stage for a deeper exploration of data structures. Participants grasped the significance of selecting the right data structure for different problem domains, which is paramount for optimizing memory usage and execution time. The classification of data structures provided a clear roadmap for navigating through arrays, linked lists, stacks, queues, trees, graphs, and heaps. This classification serves as a mental toolkit that empowers programmers to dissect complex problems and devise effective solutions.

Furthermore, the understanding of complexity analysis empowered participants to gauge the efficiency of algorithms by analyzing their time and space complexities. This skill is indispensable in selecting the most appropriate algorithm for a given scenario, contributing to optimized program performance.

The immersive study of array operations, searching algorithms, sorting techniques, linked lists, stacks, queues, recursion, binary trees, heaps, and graphs provided a panoramic view of data manipulation methods. Through algorithms and hands-on problem-solving, participants not only acquired theoretical knowledge but also honed their practical skills in solving competitive coding challenges.

As participants move forward in their programming journey, the knowledge gained from this course will serve as a cornerstone. The ability to select the right data structure, optimize algorithms, and efficiently traverse complex data sets will undoubtedly be invaluable in real-world programming scenarios. The journey undertaken during this course is a stepping stone toward becoming adept programmers capable of tackling intricate challenges and contributing to the technological landscape.

In conclusion, the C++ Programming and Data Structures & Algorithms summer course has equipped participants with a strong foundation in programming principles and a deep understanding of data manipulation techniques. This knowledge will undoubtedly serve as a catalyst for their growth as programmers and problem solvers in the ever-evolving world of technology.

**References**

* "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
* "Data Structures and Algorithm Analysis in C++" by Mark Allen Weiss.
* GeeksforGeeks (https://www.geeksforgeeks.org/): An online platform with a plethora of articles, tutorials, and practice problems on various data structures and algorithms topics.
* Lecture notes, presentations, and study materials provided by your course instructors can be referenced for accurate and comprehensive information.