CC-213L

Data Structures and Algorithms

Laboratory 12

Graphs, Sorting and Hashing

Version: 1.0.0

Release Date: 03-01-2025

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- Pointers and Dynamic Memory Allocation
- Non-Linear Data Structure
- Graph Data Structure

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, not even allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the following.

Teachers:		
Course Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistants	Tahir Mustafvi	bitf20m018@pucit.edu.pk
	Maryam Rasool	bcsf21m055@pucit.edu.pk

Background and Overview

Non-Linear Data structures

Non-linear data structures are data structures in which elements are not arranged in a sequential, linear manner. Unlike linear data structures (e.g., arrays, linked lists) where elements are stored in a linear order, non-linear data structures allow for more complex relationships among elements

Graphs

A graph is a data structure that consists of a set of nodes (or vertices) and a set of edges connecting these nodes. Graphs are widely used to represent relationships and connections between different entities. The nodes in a graph can represent entities (such as people, cities, or web pages), and the edges represent relationships or connections between these entities.

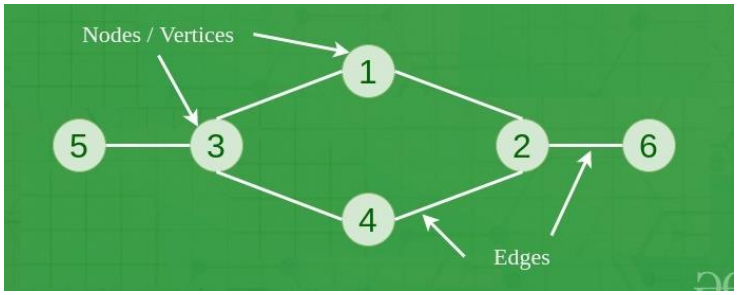


Figure 1(Graph)

Representation of Graphs

Adjacency Matrix:

In an adjacency matrix, a graph with n vertices is represented by an n x n matrix. The entry matrix[i][j] indicates whether there is an edge between vertices i and j. For an undirected graph, the matrix is symmetric.

Example:

Consider the following undirected graph:

A

/ \

B --- C

The adjacency matrix for this graph would be:

	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

In this matrix:

- The entry matrix [0][1] (or matrix [1][0]) is 1, indicating there is an edge between vertex A and vertex B.

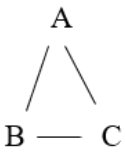
- The entry matrix [0][2] (or matrix [2][0]) is 1, indicating there is an edge between vertex A and vertex C.
- The entry matrix [1][2] (or matrix [2][1]) is 1, indicating there is an edge between vertex B and vertex C.

Adjacency List:

In an adjacency list representation, each vertex has a list of its neighboring vertices. This is an array of lists (or a HashMap), where each index represents a vertex, and the corresponding list contains the vertices adjacent to that vertex.

Example:

Using the same example graph:



The adjacency list for this graph would be:

A: [B, C]

B: [A, C]

C: [A, B]

In this list:

Vertex A is adjacent to vertices B and C.

Vertex B is adjacent to vertices A and C.

Vertex C is adjacent to vertices A and B.

Comparison:

1. Space Complexity:

Adjacency Matrix: Takes O(V^2) space for V vertices. It's more space-efficient for dense graphs.

Adjacency List: Takes O (V + E) space for V vertices and E edges. It's more space-efficient for sparse graphs.

2. Edge Existence Check:

Adjacency Matrix: Checking if an edge exists takes constant time (O (1)).

Adjacency List: Checking if an edge exists may take time proportional to the degree of the vertex (O(degree)).

3. Traversal:

Adjacency Matrix: Traversing all neighbors of a vertex takes O(V) time.

Adjacency List: Traversing all neighbors of a vertex takes O(degree) time, which is generally faster for sparse graphs.

Sorting

Sorting is the process of arranging elements in a specific order, often based on some criteria or key. In computer science and programming, sorting is a fundamental operation that involves rearranging a collection of items into a specific order. The order can be ascending or descending, and it is typically determined by the values or properties of the elements.

There are various sorting algorithms, each with its own advantages and disadvantages in terms of time complexity, space complexity, and stability. Some common sorting algorithms include:

1. **Bubble Sort:** A simple algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

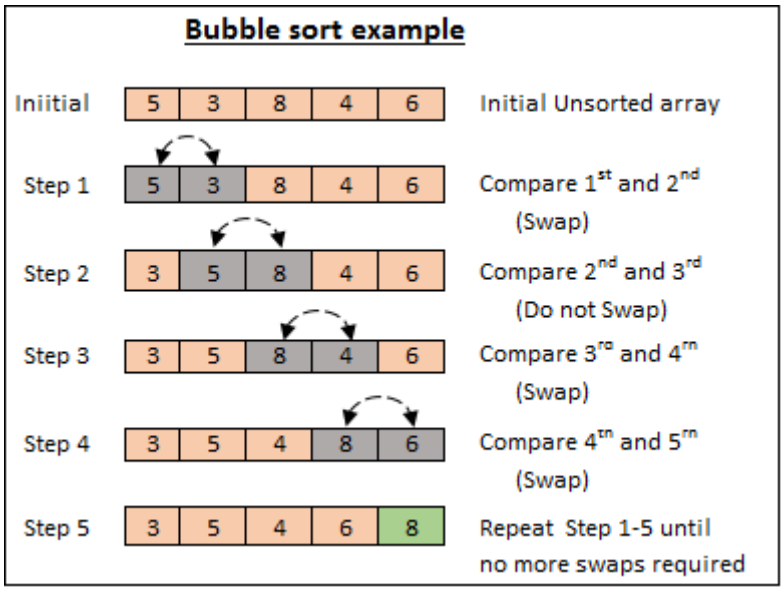


Figure 2(Bubble Sort)

2. Selection Sort: This algorithm divides the input list into a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and swaps it with the first element of the unsorted region.

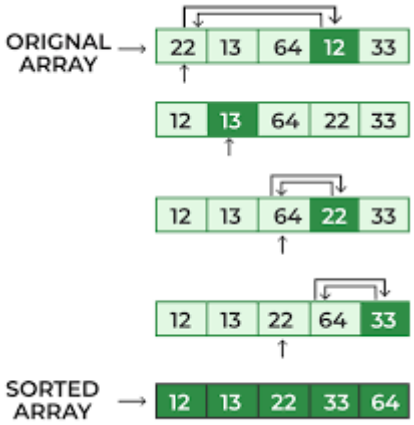


Figure 3(Selection Sort)

3. Insertion Sort: Elements are iteratively taken from the unsorted part of the array and inserted into their correct position in the sorted part.

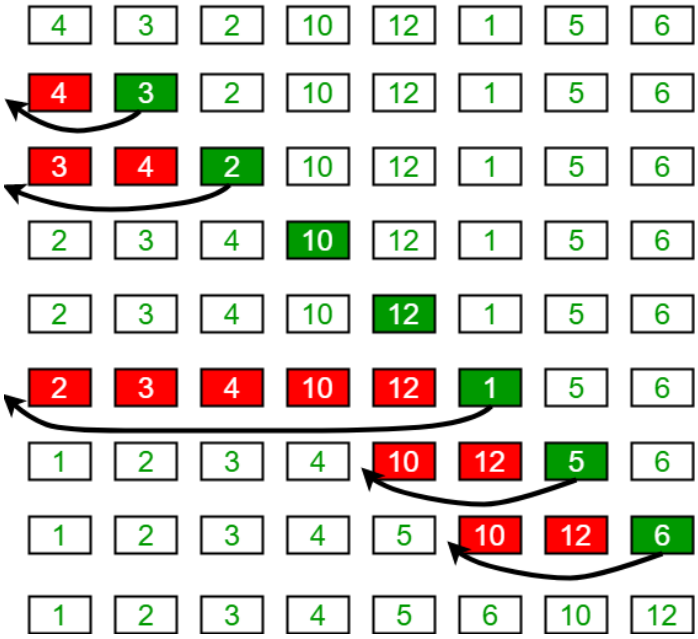


Figure 4(Insertion Sort)

4. Merge Sort: A divide-and-conquer algorithm that divides the input into smaller parts, sorts them, and then merges them back together.

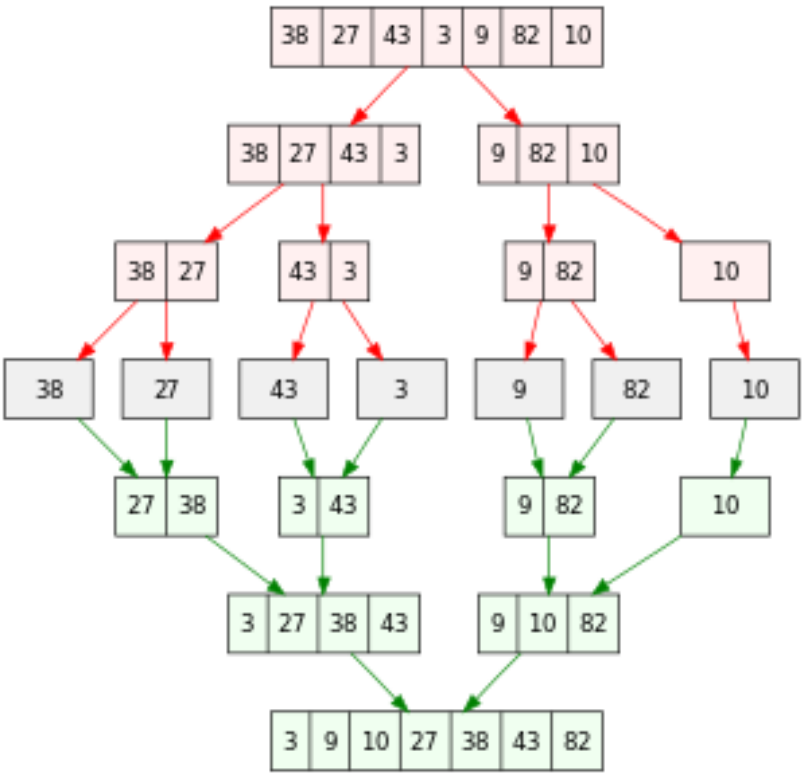


Figure 5(Merge Sort)

5. Quick Sort: Another divide-and-conquer algorithm that partitions the array into smaller segments, and then recursively sorts each segment.

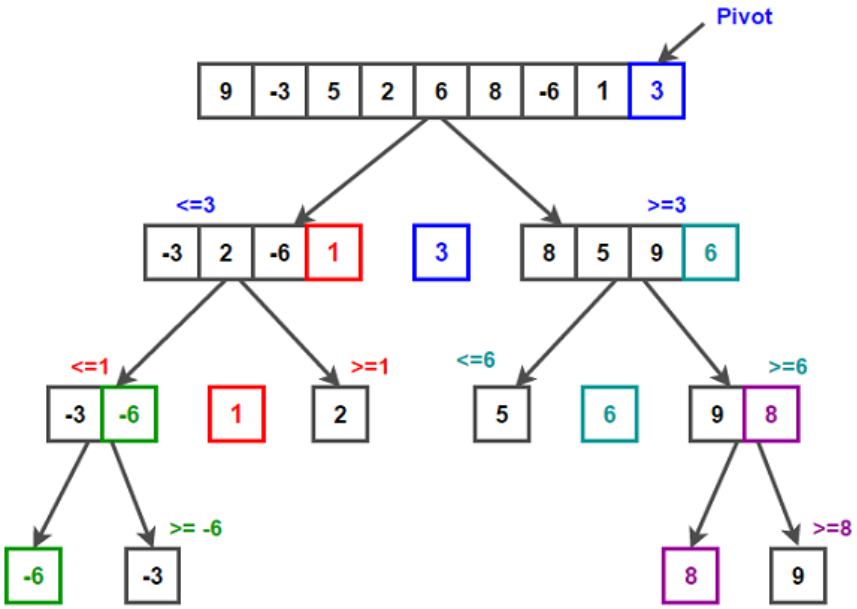


Figure 6(Quick Sort)

6. Heap Sort: Involves building a max-heap (for ascending order) or min-heap (for descending order) and repeatedly extracting the root element until the heap is empty.

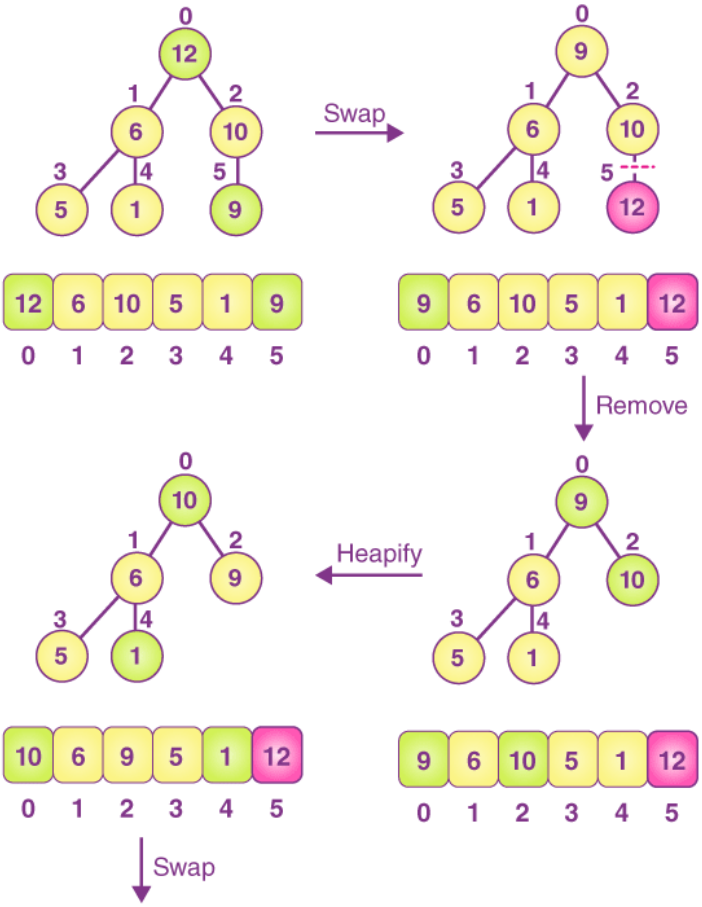


Figure 7(Heap Sort)

7. Counting Sort: Counting Sort is a non-comparison-based sorting algorithm that works well for integers with a limited range. It operates by counting the number of occurrences of each element and using that information to place the elements in their correct order. The basic idea involves creating an auxiliary array (often called a count array) to store the count of each distinct element. After counting, the algorithm places the elements in their sorted order. Counting Sort has a linear time complexity, making it efficient for certain types of data sets, particularly when the range of input values is not significantly larger than the number of elements.

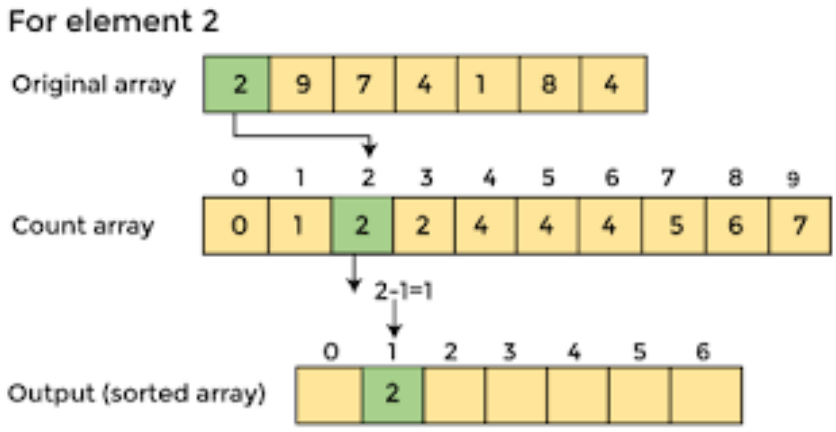


Figure 8(Counting Sort)

8. Radix Sort: Radix Sort is another non-comparison-based sorting algorithm that works well for integers or strings. It sorts elements by processing individual digits or characters, starting from the least significant digit and moving towards the most significant digit. Radix Sort can be implemented using the LSD (Least Significant Digit) or MSD (Most Significant Digit) approach. The algorithm repeatedly distributes the elements into buckets based on the values of their digits and then concatenates the buckets to obtain the sorted order. Radix Sort is known for its linear time complexity under certain conditions and is often used when the size of the data set is relatively small compared to the range of values.

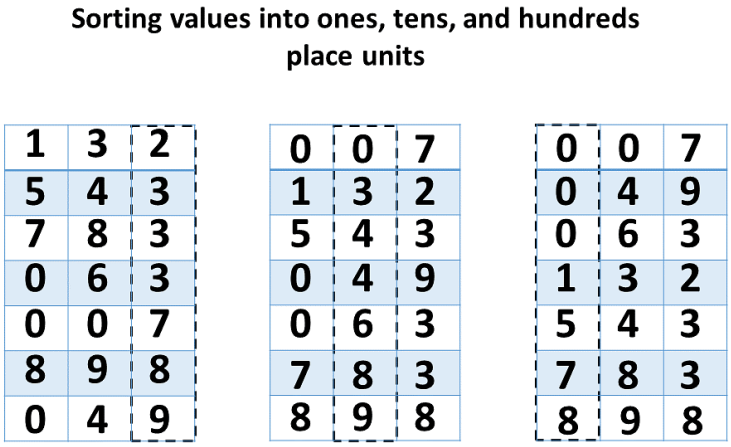


Figure 9(Radix Sort)

Sorting is a critical operation in many applications, ranging from data processing, database management, and search algorithms to various fields of science and engineering. The choice of sorting algorithm depends on factors such as the size of the data set, the nature of the data, and the specific requirements of the application.

Performance Analysis of Sorting Algorithms

1. Bubble Sort:

Time Complexity:

- Worst Case: O(n^2)
- Best Case (optimized version): O(n) when the list is already sorted

Space Complexity: O(1)

2. Selection Sort:

Time Complexity:

- Worst Case: O(n^2)
- Best Case: O(n^2)

Space Complexity: O(1)

3. Insertion Sort:

Time Complexity:

- Worst Case: O(n^2)
- Best Case: O(n) when the list is already sorted

Space Complexity: O(1)

4. Merge Sort:

Time Complexity: O(n log n) in all cases (worst, average, and best)

Space Complexity: O(n) auxiliary space for merging

5. Quick Sort:

Time Complexity:

- Worst Case: O(n^2)
- Average Case: O(n log n)
- Best Case: O(n log n)

Space Complexity: O(log n) auxiliary space for the recursive call stack

6. Heap Sort:

Time Complexity: O(n log n) in all cases (worst, average, and best)

-Space Complexity: O(1) (in-place sorting)

7. Counting Sort:

Time Complexity: O(n + k), where n is the number of elements and k is the range of input

Space Complexity: O(k), where k is the range of input

8. Radix Sort:

Time Complexity: O (d × (n + k)), where d is the number of digits in the input, n is the number of elements, and k is the radix (size of the digit set)

Space Complexity: O (n + k), where n is the number of elements and k is the radix (size of the digit set)

Hashing

Hashing is a technique used in computer science and cryptography to map data of arbitrary size to fixed-size values, typically for the purpose of fast data retrieval. The result of the mapping is called a hash code or hash value. Hashing is commonly used in hash tables, data structures that provide efficient data retrieval based on key-value pairs.

1. Hash Function:

- A hash function is a mathematical algorithm that takes an input (or "message") and produces a fixed-size string of characters, which is typically a hash code.
- The output, or hash code, is ideally unique to the input data. However, due to the fixed size of the output (hash space), collisions (two different inputs producing the same hash code) can occur.

2. Hash Table:

- A hash table is a data structure that uses a hash function to map keys to indices of an array where the corresponding values are stored.
- It provides efficient data retrieval because, in an ideal scenario, the hash function allows for direct access to the location where a particular value is stored.

3. Collisions:

- Collisions occur when two different inputs produce the same hash code. Hash functions aim to minimize collisions, but they are inevitable due to the finite size of the hash space compared to the potentially infinite set of inputs.
- Collision resolution techniques include chaining (using linked lists to handle multiple values at the same hash index) and open addressing (finding an alternative location within the hash table for the colliding value).

4. Applications:

Hashing is widely used in various applications, including:

- Hash tables for efficient data retrieval in databases and associative arrays.
- Cryptographic hash functions for secure data integrity verification (e.g., SHA-256, MD5).
- Password hashing for secure storage and validation.
- Digital signatures and checksums.

5. Cryptographic Hash Functions:

Cryptographic hash functions have additional properties beyond regular hash functions. They should be resistant to various attacks, including collision resistance (it should be computationally infeasible to find two different inputs producing the same hash) and pre-image resistance (it should be difficult to reverse the hash to find the original input).

Overall, hashing is a fundamental concept in computer science with widespread applications in data structures, databases, cryptography, and more. The choice of a hash function depends on the specific requirements of the application, such as speed, collision resistance, and security.

Activities

Pre-Lab Activities:

Task 01: Dijkstra’s Algo

Objective:
Design and implement a program to find the shortest path from a given source node to all other nodes in a weighted, directed graph using Dijkstra's algorithm.

Problem Description:
You are tasked with implementing a shortest path algorithm for a transportation network represented as a weighted, directed graph. The goal is to determine the minimum distance from a given source node S to every other node in the network.

The graph consists of ‘n’ nodes and ‘m’ edges, where each edge has a non-negative weight representing the cost to traverse it. Your program should output the shortest distances in ascending order of node numbers. If a node is unreachable from the source, indicate it by outputting "INF" for that node.

To achieve this, use **Dijkstra's algorithm**, which efficiently finds the shortest paths using a greedy approach. The algorithm relies on a priority queue (min-heap) to repeatedly select the node with the smallest tentative distance, updating the distances of its neighbours.

Example:

Adjacency List Representation:

1: [(2, 4), (3, 2)]

2: [(3, 5), (4, 10)]

3: [(5, 3), (6, 9)]

4: [(6, 11)]

5: [(4, 4), (6, 6)]

6: []

Expected Output:

0 4 2 9 5 11

Explanation:

- Distance to Node 1 (source): 0
- Distance to Node 2: 4 (via 1→2)
- Distance to Node 3: 2 (via 1→3)
- Distance to Node 4: 9 (via 1→3→5→4)
- Distance to Node 5: 5 (via 1→3→5)
- Distance to Node 6: 11 (via 1→3→5→6)

In-Lab Activities:

Task 01: Sort Colorful Balls

You are given a collection of balls, each of which can be one of three colours: red, green, or blue. These balls are stored in a single row, unsorted. Your task is to rearrange the balls such that:

1. All red balls come before all green balls.
2. All green balls come before all blue balls.

You are required to perform this sorting *in-place* within the same row, without using extra storage for sorting. The number of swaps and comparisons should be minimized to ensure optimal performance.

Write a function that takes an array of integers where:

- 0 represents a red ball,
- 1 represents a green ball, and
- 2 represents a blue ball,

and rearranges the array according to the rules mentioned above.

```
void SortBalls(vector<int>& arr);
```

Task 02: Merge Sort

Given an array sort the array efficiently using merge sort.

```
void MergeSort(vector<int>& arr);
```

Task 03: Isomorphic Strings

You are given two strings, s1 and s2, of equal length. Your task is to determine if the strings are isomorphic.

Two strings are isomorphic if there is a one-to-one mapping between the characters of s1 and s2, such that:

1. Every character in s1 maps to a unique character in s2.
2. The order of characters in s1 is preserved in s2.

The strings may contain any printable ASCII characters. The mapping should be consistent; if a character in s1 maps to a character in s2, no other character in s1 can map to the same character in s2.

Example:

Input 1:

s1 = "egg"
s2 = "add"

Output 1:

true
(Explanation: 'e' → 'a', 'g' → 'd', mapping is consistent.)

Input 2:

s1 = "foo"
s2 = "bar"

Output 2:

false
(Explanation: 'o' cannot map to both 'a' and 'r'.)

Post-Lab Activities:

Task 01: Hash Table Implementation

Class Definition: Implement a HashTable class to store names using a dynamic array of strings. The class includes:

- **Attributes**:
  - table (dynamic array of strings),
  - S (total slots),
  - n (current elements).
- **Constructor**: Initializes an empty hash table of a given size.
- **Destructor**: Cleans up memory.
- **Utility Functions**:
  - isEmpty(), isFull(), loadFactor().

Hash Function: Implement a helper function getHashValue that computes the sum of ASCII values of characters in a string and applies MOD by the table size.

Member Functions: Implement the following:

- **insert(string name)**:
  - Insert a name using the hash function and resolve collisions via linear probing.
  - Display indices traversed during insertion.
  - Return true if inserted successfully; false if the table is full.
- **search(string name)**:
  - Search for a name using the hash function and linear probing.
  - Display indices traversed during the search.
  - Return true if found; false otherwise.
- **display()**: Display the hash table contents with indices, marking empty slots as "EMPTY".
- **remove(string name)**:
  - Remove a name from the hash table.
  - Return true if removed; false if not found.

→ This task requires a menu-based interactive program demonstrating the functionality of the HashTable class

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[10 marks]**
  - Task 01 Dijkstra Algo [10 marks]
- **Division of In-Lab marks:** **[30 marks]**
  - Task 01:Sort Colorful Balls [10 marks]
  - Task 02: Merge Sort [10 marks]
  - Task 03: Isomorphic Strings [10 marks]
- **Division of Post-Lab marks:** **[10 marks]**
  - Task 01: Hash Table Implementation [10 marks]

References and Additional Material:

Sorting

https://www.geeksforgeeks.org/sorting-algorithms/

Lab Time Activity Simulation Log:

- Slot – 01 – 02:15 – 02:30: Class Settlement
- Slot – 02 – 02:30 – 03:00: In-Lab Task 01
- Slot – 03 – 03:00 – 03:30: In-Lab Task 02
- Slot – 04 – 03:30 – 04:15: In-Lab Task 03
- Slot – 05 – 04:15 – 04:30: Discussion on Post-Lab