SOS-2023

FINAL-REPORT

Data Structures and Algorithms

Name - Sujeet Mehta

TOPICS LEARNT:

1) Brushing up C++ with OOPs and learnt basic data structure Arrays.

2) Linear Search, Binary Search, Selection sort, Insertion sort, and Bubble sort in an Array.

3) Pointers and Double Pointers, Recursion.

4) Merge and Quick Sort using recursion.

5) Linked Lists and its types, detecting and removing loop in a LL, Merging two LL.

6) Stacks, Queues and Priority Queues.

7) Binary Tree - representation, implementation, traversal.

8) Binary Search Tree(BST) - implementation

9) Heaps , max Heap, min Heap, Heapify Algorithm.

10) Hashmaps and Backtracking

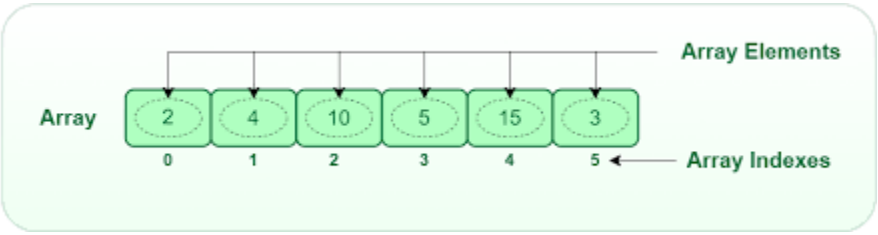
11) Graphs and Shortest path Algorithms

12) Dynamic Programming

Basic Data Structures

Arrays-

Arrays are data structures that allow you to store and manipulate a collection of elements of the same type.



In an array, each element is identified by its index, which represents its position within the array. The index typically starts at 0 for the first element and increments by 1 for each subsequent element. This allows for easy and efficient access to individual elements.

Searching in an Array

1) **Linear Search -**
   Linear search is a simple searching algorithm used to find the presence of an element in an array. It sequentially checks each element of the array until a match is found or the entire array has been traversed.

2) **Binary Search -**
   Binary search is an efficient searching algorithm used to find a specific element in a sorted array. It follows a divide-and-conquer approach, repeatedly dividing the search space in half until the target element is found or the search space is empty.

Sorting in an Array

1) **Selection Sort -**
   Selection sort is a simple comparison-based sorting algorithm. It works by dividing the input array into two portions: the sorted portion at the beginning and the unsorted portion at the end. In each iteration, it finds the smallest (or largest) element from the unsorted portion and swaps it with the first element of the unsorted portion, expanding the sorted portion by one element.

2) **Bubble Sort -**
   Bubble sort is a simple comparison-based sorting algorithm that repeatedly steps through the array, compares adjacent elements, and swaps them if they are in the wrong order. It repeatedly passes through the array until the entire array is sorted.

3) **Insertion Sort -**
   Insertion sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time. It works by iteratively inserting each element from the unsorted portion into its correct position in the sorted portion of the array.

Basic STL-

STL (Standard Template Library) is a library available in C++ that provides a collection of reusable algorithms and data structures.

1) **Vectors -**
   ```
   #include <vector>
   std::vector<int> numbers;  // Declaration of a vector
   numbers.push_back(10);  // Insert an element at the end
   numbers.pop_back();  // Remove the last element
   int size = numbers.size();  // Get the number of elements
   ```

2) **Lists -**

```
#include <list>
std::list<int> numbers;  // Declaration of a list
numbers.push_back(10);  // Insert an element at the end
numbers.push_front(20);  // Insert an element at the beginning
numbers.pop_back();  // Remove the last element
```

3) **Map -**

```
#include <map>
std::map<std::string, int> scores;  // Declaration of a map
scores["Alice"] = 85;  // Insert/Update a value with the key "Alice"
int score = scores["Alice"];  // Access the value with the key "Alice"
```

4) **Set -**

```
#include <set>

std::set<int> numbers; // Declaration of a set

numbers.insert(10); // Insert an element

numbers.erase(10); // Remove an element

bool exists = numbers.count(10); // Check if an element exists
```

5) **Iterators -**

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (std::vector<int>::iterator it = numbers.begin(); it !=
numbers.end(); ++it) {
    int element = *it;  // Access the current element
}
```

Linked List-

A linked list is a linear data structure in which elements are stored as separate objects called "nodes," and each node contains a value and a reference (or pointer) to the next node in the list. Unlike arrays, linked lists do not require contiguous memory allocation.

Detecting and Removing Loop in a Linked List -

Detecting and removing a loop in a linked list can be done using the Floyd's cycle-finding algorithm, also known as the "tortoise and hare" algorithm. It involves using two pointers, one moving at a slower pace (tortoise) and the other at a faster pace (hare), to detect if there is a loop in the linked list. Once a loop is detected, it can be removed by breaking the loop and making the last node point to nullptr.

```
void detectAndRemoveLoop(Node* head) {
    if (head == nullptr || head->next == nullptr)
        return;  // No loop exists

    Node* slow = head;
    Node* fast = head;

    // Move the slow pointer by one step and the fast pointer by two steps
    // until they meet or the fast pointer reaches the end of the list
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        // Loop detected
        if (slow == fast)
            break;
    }
```

```
    // No loop exists
    if (slow != fast)
        return;

    // Move the slow pointer back to the head and move both pointers
    // at the same pace until they meet at the start of the loop
    slow = head;
    while (slow->next != fast->next) {
        slow = slow->next;
        fast = fast->next;
    }

    // Break the loop by making the last node point to nullptr
    fast->next = nullptr;
}
```

Merging Two Linked Lists -

The mergeLists function takes two sorted linked lists as input and merges them into a single sorted linked list. It uses a dummy node (dummyNode) as the head of the merged list and a tail pointer to keep track of the last node in the merged list. The function traverses both lists simultaneously, comparing the values of the nodes at each step. The smaller value is appended to the merged list, and the respective pointer (head1 or head2) is moved to the next node. After the traversal, any remaining nodes in either list are attached to the merged list.

Stacks-

A stack is a Last-In-First-Out (LIFO) data structure, where the last element inserted is the first one to be removed. Think of it as a stack of plates, where you can only access the topmost plate. The key operations of a stack are:

Push: Insert an element onto the top of the stack.
Pop: Remove the topmost element from the stack.
Peek or Top: Retrieve the value of the topmost element without removing it.
IsEmpty: Check if the stack is empty.

Queue -

A queue is a First-In-First-Out (FIFO) data structure, where the first element inserted is the first one to be removed. It resembles a queue of people waiting in line, where the person who arrived first is served first. The primary operations of a queue are:

Enqueue: Add an element to the end of the queue.
Dequeue: Remove the element from the front of the queue.
Front: Retrieve the value of the element at the front without removing it.
IsEmpty: Check if the queue is empty.

Priority Queue -

A priority queue is an abstract data type that is similar to a queue but assigns a priority value to each element. Elements with higher priority are dequeued (removed) before elements with lower priority. Priority queues are often implemented using binary heaps or other heap-based data structures.

Here are the main operations of a priority queue:

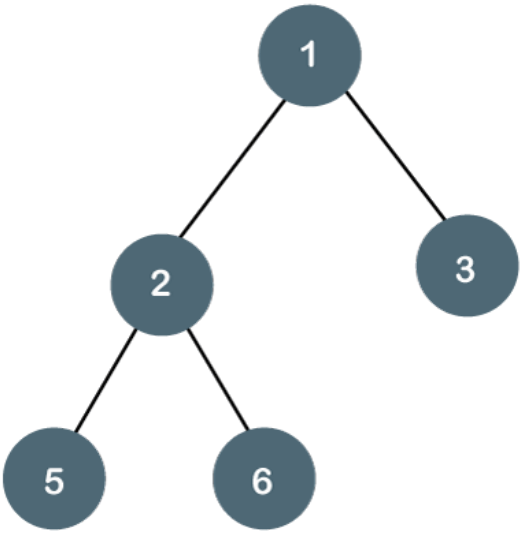
Insert: Add an element to the priority queue with an associated priority value.
Delete-Max (or Delete-Min): Remove and return the element with the highest (or lowest) priority from the priority queue.
Peek-Max (or Peek-Min): Return the element with the highest (or lowest) priority without removing it.
IsEmpty: Check if the priority queue is empty.

Binary Tree -

A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is a type of tree structure where nodes are connected through edges.

Properties of a binary tree:

- Each node in a binary tree can have at most two children, often called the left child and the right child.
- The left child is positioned to the left of the parent node, and the right child is positioned to the right.
- The order of the children is significant, meaning that the left child is considered before the right child.
- Each node in a binary tree can have zero, one, or two children.
- The binary tree can be empty, meaning it does not contain any nodes.

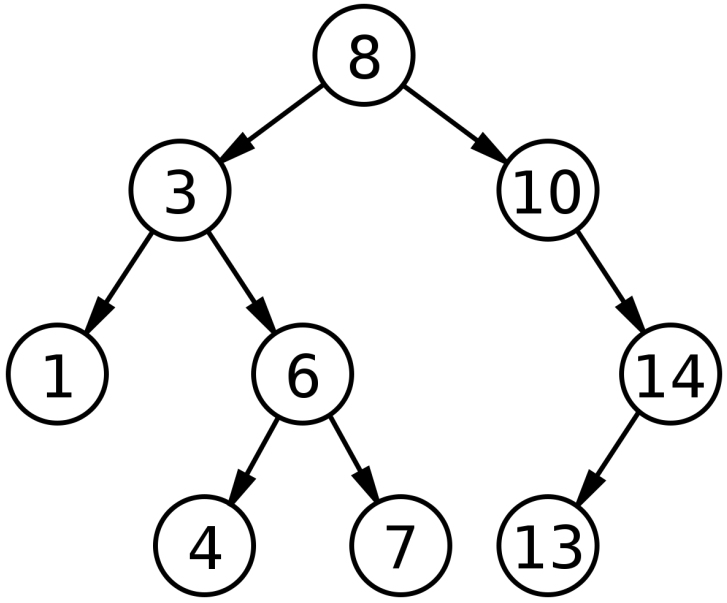


Binary Search Tree -

A binary search tree (BST) is a binary tree data structure that follows a specific ordering property. In a BST, the left subtree of a node contains values smaller than the node, and the right subtree contains values greater than the node. This property enables efficient searching, insertion, and deletion operations.

Properties of a binary search tree:

- For any node in the tree, the values in its left subtree are smaller than the node's value, and the values in its right subtree are greater.
- No duplicate values are allowed in the tree (although some variants of BSTs may allow duplicates in different variations).



Heaps -

A heap is a specialized tree-based data structure that satisfies the heap property. Heaps are commonly used to implement priority queues, where the element with the highest (or lowest) priority can be efficiently accessed.

There are two main types of heaps:

Min Heap: In a min heap, for any given node, the value of the node is smaller than or equal to the values of its children. Therefore, the minimum element is always at the root.

Max Heap: In a max heap, for any given node, the value of the node is greater than or equal to the values of its children. Thus, the maximum element is always at the root.

Heapify Algorithm:

The heapify algorithm is used to convert an array into a valid heap structure. It is an important operation in heap-related data structures and algorithms. Here's an overview of the heapify algorithm:

- Start with an unstructured array of elements.
- Identify the last non-leaf node in the array. The non-leaf nodes are the ones that have at least one child.
- For an array-based representation, the last non-leaf node can be calculated as (array.length / 2) - 1.
- Iterate from the last non-leaf node to the first element of the array.
- For each node, perform the "heapify-down" operation to establish the heap property.
- Compare the value of the current node with its children.
- If the node violates the heap property (e.g., in a max heap, the node's value is smaller than one or both of its children), swap the node with the larger child (in a max heap) or smaller child (in a min heap).

- Repeat this process with the swapped child until the heap property is satisfied.

Hash Mapping -

Hashing: The hash function takes a key as input and calculates its hash code. The hash code is typically an integer that represents the index in the underlying array where the value associated with the key will be stored.

Indexing: The hash code is then converted into a valid index within the array by performing a modulo operation with the array size. This ensures that the index falls within the valid range of array indices.

Collision Handling: If two or more keys produce the same hash code (collision), a collision resolution mechanism is employed to store multiple values associated with the same index. Common techniques for collision resolution include chaining (using linked lists or other data structures to store multiple values at the same index) or open addressing (probing for the next available index).

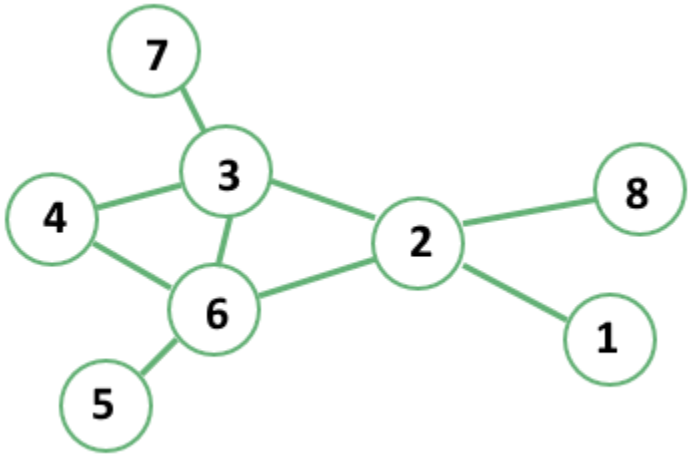
Storage and Retrieval: The key-value pair is stored at the computed index in the underlying array. When retrieving a value, the key is hashed again to compute the index, and the value is retrieved from that index.

Deletion: To delete a key-value pair, the key is used to compute the index, and the corresponding value is removed from that index. Collision resolution mechanisms may need to be applied to handle the removal of values in a chained structure.

Graphs -

Graphs consist of a set of vertices (also known as nodes) and a set of edges (also known as links or connections) that connect pairs of vertices.

Undirected Graphs : In undirected graphs, the edges have no direction and simply represent a connection between two vertices. If there is an edge between vertex A and vertex B, it means that A is connected to B and B is connected to A. The relationship is symmetric. For example, think of a network of friends where edges represent friendships



Directed Graphs (Digraphs): In directed graphs, the edges have a direction, indicating that there is a one-way relationship between two vertices. If there is a directed edge from vertex A to vertex B, it means there is a connection from A to B, but not necessarily from B to A. The relationship is asymmetric. For example, a network of roads where edges represent one-way streets is a directed graph.

Cycle Detection in Directed Graphs — Khan's Algorithm

```
class Solution {
public:
    vector<int> findOrder(int numCourses, vector<vector<int>>&
prerequisites) {
        //topological sort in directed graphs
        //using Kahn's Algorithm
        //make an adjacency list
        int n = prerequisites.size();
        unordered_map<int, list<int>> adj;
        vector<int> ans;
        vector<int> indegree(numCourses, 0);
        for(int i = 0; i < n; i++){
            int u = prerequisites[i][0];
            int v = prerequisites[i][1];
            adj[v].push_back(u);
            indegree[u]++;
        }


        queue<int> q;
        for(int i = 0; i < numCourses; i++){
            if(indegree[i] == 0){
                q.push(i);
```

```
            }
        }
        while(!q.empty()){
            int front = q.front();
            q.pop();
            ans.push_back(front);
            for(auto j : adj[front]){
                indegree[j]--;
                if(indegree[j] == 0){
                    q.push(j);
                }
            }
        }
        if(ans.size() != numCourses){
            return {};
        }
        return ans;
    }
};
```

Shortest Path — Dijkstra's Algorithm

```
#include <unordered_map>
#include<list>
#include<set>
vector<int> dijkstra(vector<vector<int>> &edge, int vertices, int edges, int
source) {
    // make the adjacency list
    unordered_map<int, list<pair<int, int>>> adj;
    for (int i = 0; i < edges; i++) {
        int u = edge[i][0];
        int v = edge[i][1];
        int w = edge[i][2];

        adj[u].push_back({ v, w });
        adj[v].push_back({ u, w });
```

```
    }

    // now making a distance vector
    vector<int> dist(vertices, INT_MAX);
    dist[source] = 0;
    set<pair<int, int>> q;
    q.insert(make_pair(0, source));
    while (!q.empty()) {
        auto front = *(q.begin());
        q.erase(q.begin());
        int index = front.second;
        int weight = front.first;
        for(auto j : adj[index]){
            if(weight + j.second < dist[j.first]){
                dist[j.first] = j.second + weight;
                q.insert(make_pair(dist[j.first], j.first));
            }
        }
    }

    return dist;
}
```

Dynamic Programming

In Dynamic Programmng three ways are applied to make the code effective in terms of reducing Time Complexity and reducing space Complexity.

1) Recursion + Memoisation
2) Tabulation
3) Space Optimization

Finding minimum cost to climb at the top of the ladder –

```
class Solution {
public:
    int find(vector<int>& cost, int stair,vector<int>& dp) {
        if(stair == -1)
            return min(find(cost, 0,dp), find(cost, 1,dp));
        else if (stair == cost.size() - 1 || stair == cost.size() - 2)
            return cost[stair];

        if(dp[stair]!=-1)
            return dp[stair] ;

        dp[stair]  = cost[stair] + min(find(cost, stair + 1,dp), find(cost, stair + 2,dp));
        return dp[stair] ;
    }
    int minCostClimbingStairs(vector<int>& cost) {
        vector<int> dp(cost.size()+1,-1) ;
        return find(cost,-1,dp);
    }
};
```

References -

1) Love Babbar (DSA playlist Youtube)
2) Coding Ninjas – 342 problems solved – https://www.codingninjas.com/codestudio/profile/UKR10
3) LeetCode – 100+ problem solved – https://leetcode.com/UmeshKr10/
4) GeeksForGeeks – 120+ problems solved https://auth.geeksforgeeks.org/user/umeshkumboor/practice