Question-1.

Assignment on Implementation of Trie Data Structure with Hindi Data.

Objectives

The objective of this assignment is to:

- Understand the concept of Trie data structure
- Implement a Trie data structure in C++ to store and search Hindi words

Requirements

Your task is to implement a Trie data structure in C++ to store and search Hindi words. Your solution should meet the following requirements:

- Use a node-based implementation through to represent the Trie
- Store Hindi words in the Trie using Unicode encoding
- Support the following operations:
  - Insert a Hindi word into the Trie
  - Search for a Hindi word in the Trie
  - Print all the Hindi words in the Trie

Implementation

Your implementation should include the following:

- Download the the hindi text from the link below and download the hindi text and read the file to take the hindi words

  https://drive.google.com/file/d/1Lt91ZOnJRrYfoMY7nqX60QFYFHRyMjNc/view?usp=sharing

- Define a TrieNode struct to represent a node of the Trie, which should include the following members:(10 marks)
  - children: a structure to store the child nodes of the current node
  - isEndOfWord: a boolean flag to indicate if the current node represents the end of a word
- Define a Trie class to represent the Trie, which should include the following members:(20 marks)
  - root: a pointer to the root node of the Trie

- **insert():** a function to insert a Hindi word into the Trie, which should take the following parameter:
  - word: a string representing the Hindi word to insert
- **search():** a function to search for a Hindi word in the Trie, which should take the following parameter:
  - word: a string representing the Hindi word to search for
  - Returns true if the word is found in the Trie, false otherwise
- **printAllWords():** a function to print all the Hindi words in the Trie

Your implementation should ensure that:

- Read the words from the file (10 marks)
- Hindi words are inserted correctly into the Trie (20 marks)
- Searching for a Hindi word returns true if the word in Trie, false otherwise (20 marks)
- All the Hindi words in the Trie are printed correctly (20 marks)

Testing

To test your implementation, you can:

- Insert Hindi words into the Trie and check that they are inserted correctly
- Search for Hindi words in the Trie and check that the search returns true for words that are in the Trie, and false for words that are not in the Trie
- Print all the Hindi words in the Trie and check that all the words are printed correctly

Question-2.

Reader-Writer Problem Using Semaphores and Threads.

Objectives

The objective of this assignment is to:

- Understand the Reader-Writer problem and its solution using semaphores
- Implement the solution to the Reader-Writer problem using threads and semaphores in C++

Requirements

Your task is to implement a solution to the Reader-Writer problem using semaphores and threads in C++. Your solution should meet the following requirements:

- Use semaphores to synchronize the access to the shared resource
- Use two types of threads: reader threads and writer threads
- Reader threads should only read the shared resource
- Writer threads should modify the shared resource
- Ensure that only one thread can access the shared resource at a time
- Implement the solution in C++ using the standard library's threading facilities

Implementation

Your implementation should include the following:

- Define a shared resource, such as an array or a data structure(5 marks)
- Define a semaphore for controlling access to the shared resource(10 marks)
- Define two functions for reader threads: one to read from the shared resource and another to release the semaphore(10 marks)
- Define two functions for writer threads: one to write to the shared resource and another to release the semaphore(10 marks)
- Create a main function that creates multiple reader and writer threads, and joins them once they have completed(15marks)

Your implementation should ensure that:

- Multiple reader threads can access the shared resource simultaneously(15 marks)
- Only one writer thread can access the shared resource at a time(15 marks)
- Readers do not interfere with writers, and vice versa(10 marks)
- The order of execution is deterministic and does not cause any deadlocks or race conditions(10 marks)

Testing

To test your implementation, you can:

- Create multiple reader threads that read from the shared resource
- Create multiple writer threads that write to the shared resource
- Observe that only one writer thread can access the shared resource at a time
- Observe that multiple reader threads can access the shared resource simultaneously
- Observe that readers do not interfere with writers, and vice versa
- Observe that the order of execution is deterministic and does not cause any deadlocks or race conditions

Question-3

Making Three Data Structures Persistent.

Objectives

The objective of this assignment is to:

- Understand the concept of persistent data structures
- Implement three common data structures to be persistent in C++:
  - Binary Search Tree (BST) :insert(10 marks),delete(10 marks), search(10 marks).
  - Heap:insert(10 marks),delete(10 marks), search(10 marks).
  - Linked-List: insert(10 marks),delete(10 marks), search(10 marks).
- Visualize the BST and Heap using Graphviz(10 marks).

Requirements

Your task is to implement the above-mentioned data structures to be persistent in C++. Your solution should meet the following requirements:

- Use a node-based implementation to represent each data structure
- Store the nodes in each data structure in a way that allows the structures to be easily traversed to create a new version of the structure.
- Support the following operations for each data structure:
  - Insert a new element into the structure
  - Delete an existing element from the structure
  - Search for an element in the structure

Implementation

Your implementation should include the following:

- Define a node struct to represent a node of each data structure, which should include the following members:
  - `data`: the element to be stored in the node
  - `left`: a pointer to the left child of the node (if any)
  - `right`: a pointer to the right child of the node (if any)
- Define a class for each data structure, which should include the following members:
  - `root`: a pointer to the root node of the data structure
  - `insert()`: a function to insert a new element into the structure, which should take the following parameter:
    - `data`: the element to be inserted

        - Returns a new instance of the data structure with the inserted element.
    - remove(): a function to remove an existing element from the structure, which should take the following parameter:
        - data: the element to be removed
        - Returns a new instance of the data structure without the removed element
    - search(): a function to search for an element in the structure, which should take the following parameter:
        - data: the element to search for
        - Returns true if the element is in the structure, false otherwise

Your implementation should ensure that:

- The data structures are persistent, meaning that any changes made to a structure result in a new instance of the structure without modifying the original structure.
- Inserting and deleting elements from the structures results in a new instance of the structure with the changes applied correctly.
- Searching for an element in the structures returns true if the element is in the structure, false otherwise.

Testing

To test your implementation, you can:

- Insert elements into the structures and check that they are inserted correctly.
- Delete elements from the structures and check that they are deleted correctly.
- Search for elements in the structures and check that the search returns true for elements that are in the structure, and false for elements that are not in the structure.

Question-4

Implementation of VEB Tree with Insertion, Deletion, Successor, Predecessor

Objectives

The objective of this assignment is to:

- Understand the concept of Van Emde Boas (VEB) trees
- Implement a VEB tree with insertion, deletion, successor and predecessor operations in C++

Requirements

Your task is to implement a VEB tree with the following requirements:

- Use an array to represent the VEB tree.
- Implement the following operations:
  - insert(): Insert an element into the VEB tree.
  - delete(): Delete an element from the VEB tree .
  - successor(): Find the successor of an element in the VEB tree.
  - predecessor(): Find the predecessor of an element in the VEB tree.

Implementation

Your implementation should include the following:

- Define a class for the VEB tree, which should include the following members:
  - u: the size of the universe of the VEB tree
  - min: the minimum element in the VEB tree.[2 marks]
  - max: the maximum element in the VEB tree.[3marks]
  - summary: a pointer to the summary VEB tree.[4 marks]
  - clusters: an array of pointers to the sub-VEB trees.[2 marks]
  - high(): a function that returns the high order bits of a given element in the VEB tree.[3 marks]
  - low(): a function that returns the low order bits of a given element in the VEB tree.[3 marks]
  - index(): a function that returns the index of a given element in the VEB tree.[6 marks].
  - minimum(): a function that returns the minimum element in the VEB tree.[7 marks]

- maximum(): a function that returns the maximum element in the VEB tree.[7 marks]
- empty(): a function that returns true if the VEB tree is empty, false otherwise.[3 marks]
- insert(): a function that inserts an element into the VEB tree.[10 marks]
- delete(): a function that deletes an element from the VEB tree.[10 marks]
- successor(): a function that finds the successor of a given element in the VEB tree.[20 marks]
- predecessor(): a function that finds the predecessor of a given element in the VEB tree.[20 marks]

Your implementation should ensure that:

- Inserting and deleting elements from the VEB tree result in the correct structure of the tree.
- Finding the successor and predecessor of elements in the VEB tree is performed correctly.

Testing

To test your implementation, you can:

- Insert elements into the VEB tree and check that they are inserted correctly
- Delete elements from the VEB tree and check that they are deleted correctly
- Find the successor and predecessor of elements in the VEB tree and check that the results are correct

Question-5

AVL Tree Implementation in C++ without Calculating Height

The goal of this assignment is to implement an AVL tree in C++ without calculating the height of the tree for adjusting balance factors during insertion and deletion. Your task is to create a program that allows a user to interact with an AVL tree by inserting and deleting nodes, and finding the successor or predecessor of a given node.

Your program should use the following components:

- A node struct to represent each node in the AVL tree.
- An AVL tree class to manage the nodes and provide insertion, deletion, successor, and predecessor functions.
- A user interface that allows the user to input commands to interact with the AVL tree.

The node struct should have the following features:

- A key value to store the value of the node.
- Left and right pointers to reference the left and right child nodes.
- A balance factor to track the balance of the subtree rooted at this node.

The AVL tree class should have the following features:

- An insert function to add a new node to the tree.
- A delete function to remove a node from the tree.
- A successor function to find the node with the smallest key greater than a given key.
- A predecessor function to find the node with the largest key smaller than a given key.

The user interface should allow the user to enter commands to interact with the AVL tree. The following commands should be supported:

- insert: insert a new node into the AVL tree.(10 marks)
- delete: remove a node from the AVL tree.(10 marks)
- search: search a node in the AVL tree.(10 marks)
- successor: find the successor of a given node.(20 marks)
- predecessor: find the predecessor of a given node.(20 marks)
- print: print the contents of the AVL tree in ascending order.(10 marks)
- visualize_tree: visualize the AVL tree using the GRAPHVIZ tool(20 marks)

To complete this assignment, you should write a C++ program that implements an AVL tree without calculating the height of the tree for adjusting balance factors during insertion and deletion. Your program should have a user interface that allows the user to input commands to interact with the AVL tree. You should test your program with various inputs to ensure that it functions correctly.

Question 6:
Double Threaded Binary Search Tree Implementation in C++

Task Description:

In this assignment, you will be implementing a double threaded binary search tree in C++. The threaded binary search tree is a modified version of the binary search tree, where each node has an extra pointer, called a thread, which points to either its in-order predecessor or successor. These threads allow for efficient in-order traversal of the tree without requiring recursion or a stack.

Requirements:

Your program should use the following components:

- A node class to represent each node in the binary search tree.
- A binary search tree class to manage the nodes and provide insertion, deletion, successor, and predecessor functions.
- A user interface that allows the user to input commands to interact with the binary search tree.

The node class should have the following features:

- A key value to store the value of the node.
- Left and right pointers to reference the left and right child nodes.
- Left_Thread pointing to the inorder predecessor.
- Right_Thread pointing to inorder successor.

Implementation:

- insert (x) -- insert an element x (if not present) into the BST. If x is present, throw an exception. (5 marks)
- search(x) -- search an element x, if found return its reference, otherwise return NULL (5 marks)
- delete(x) -- delete an element x, if the element x is not present, throw an exception(10 marks)
- reverseInorder() -- returns a singly linked list containing the elements of the BST in max to min order. You should not use any extra stack and use threading for non-recursive implementation.(15 marks)
- successor(ptr) -- returns the key value of the node which is the inorder successor of the x, where x is the key value of the node pointed by ptr. (10 marks)
- split(k) -- split the BST into two BSTs T1 and T2, where keys(T1) <= k and keys(T2) > k. Note that, k may / may not be an element of the tree. Your code should run in O(h) time, where h is the height of the tree. Print the inorder of both the BSTs T1 and T2 using recursive/non-recursive implementation within this function. (15 marks)
- allElementsBetween(k1, k2) -- returns a singly linked list (write your own class) that contains all the elements (k) between k1 and k2, i.e., k1 <= k <= k2. Your code should run in O(h + N) time, where N is the number of elements that appears between k1 and k2 in the BST. (15 marks)

- kthElement(k) -- finds the k-th largest element in the BST and prints the key value. Your code should run in O(h) time.(10 marks)
- printTree() -- Your program should print the BST (in a tree like structure) [use graphviz](15 marks)

Question-7.

Assignment: In a connected, undirected graph, a vertex is called an articulation point if removing it and all the edges associated with it disconnects G.

This assignment is on identifying all articulation points in a given connected and undirected graph.

Problem 1:

Create an **adjacency list** representation of a connected and undirected graph given of the form:

- Line 1 contains n, the number of nodes in the graph. (nodes are numbered with 0, 1, 2, …, n-1).
- Line 2 contains r, the number of edges.
- Next r lines give the edges of the form (u, v).

6

7

1 0

0 5

1 2

1 3

1 2

2 3

2 4

3 4

Remember the graph is undirected. A template code is given. You need to complete the ***add_edge*** *function.*

Problem 2:

Develop a **naïve algorithm to find all articulation point** in a connected and undirected graph. The complexity of your algorithm can be O (V x (V + E)), where V is the number of vertices and E is the number of edges in the Graph. You are also allowed to use additional space equivalent to

adjacency list. You need to update **find_AP_basic** function for the same. A basic dfs code is given. You may utilize (update if required) this function for your purpose.

Output: Space separated articulation points; NONE in case no articulation point is found

Example: For the above graph 1 0

Problem 3:

Here, the objective is to develop an efficient algorithm for finding articulation points. Let us consider the depth-first tree (DFT) GT of G. In DFT, a vertex v is an articulation point if one of the following two conditions is true.

(1) The vertex v is the root of GT and it has at least two children.

(2) The vertex v is non-root vertex in GT and v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v in GT.

The first case is simple to detect. For every vertex, count children. If currently visited vertex v is root (parent[v] is NIL) and has more than two children, print it. For second case, for every node v, we need to find out the earliest visited vertex (the vertex with minimum discovery time) that can be reached from subtree rooted with u. So, you need to maintain the array low[v] which is defined as follows:

low[v] = min (d[v], d[w]), where (u, w) is a back edge for some descendant u of v to some an ancestor w of v. d[v] denotes the discovery time of vertex v during DFS.

Develop an efficient algorithm by modifying DFS to calculate low[v] and use it determine if a vertex is satisfying the condition. It may be noted that identifying condition 1 is relatively easy as discussed above. The complexity of your algorithm must be O (V + E) time and O (1) space. You need to update **find_AP_efficient** function for the same.

Output: Space separated articulation points; NONE in case no articulation point is found

Example: For the above graph

1 0

Evaluation Criteria:

The template of the code is given. You need to write the three functions in the code.

1. You need to write one function each problem.

2. You will get ZERO for a function which is NOT implemented in the complexity mentioned.

3. You will be awarded ZERO if you touch any other part of the code.

template code:

```
#include<iostream>
#include<stdlib.h>
using namespace std;

typedef struct node{
	int dest;
	struct node* next;
}node;

typedef struct list{
	node* head;
}list;

typedef struct graph{
	list* adj_list; // Adjacency List of graph
	int n; // No. of nodes in graph
}Graph;

void add_edge(Graph* g, int src, int dest){

	//UPDATE ME for Problem 1.
}
```

```
void read_graph(Graph* g){

	cin >> g->n;
	g->adj_list = (list*)malloc((g->n)*sizeof(list));

	// Nullify all vertex lists in adjacency list
	for(int i=0;i<g->n;i++)
		g->adj_list[i].head = NULL;

	int e; // No. of edges
	cin >> e;

	for(int i=0;i<e;i++){
		int src,dest;
		cin >> src >> dest;
		add_edge(g,src,dest);
	}

}


void dfs(Graph* g, int i, bool* vis, int* c){
	vis[i]=true;
	*c += 1;
```

```
        node* t = g->adj_list[i].head;
        while(t!=NULL){
                if(!vis[t->dest]){
                        dfs(g, t->dest, vis, c);
                }
                t=t->next;
        }
}


void find_AP_basic(Graph* g, bool* ap){

        //UPDATE me for Problem 2
}


void find_AP_efficient(Graph* g, bool* ap){

        //UPADTE me for Problem 3

}

void print_ap(Graph* g, bool* ap){
        // Prints all the Articulation Points
        bool flg = false;
```

```
        for(int i=0;i<g->n;i++){
                if(ap[i]){
                        cout << i<<" ";
                        flg = true;
                }
        }
        if(!flg){
                // No Articulation Points
                cout << "NONE";
        }
        cout << endl;
}



int main(){
        Graph* g = (graph*)malloc(sizeof(graph));

        //Problem 1
        read_graph(g); //update the add_edge function calling from read_graph

        //Problem 2
        bool ap[g->n]; //ap[i] = true if vertex i is an articulation point; ap[i] = false otherwise.
        find_AP_basic(g, ap); //This function assign true/false to the array ap for each vertex.
        print_ap(g,ap); //print the articulation points
```

```
        //Problem 3
        bool ap2[g->n]; //similar to array ap above
        find_AP_efficient(g,ap2); //populate ap2 array
        print_ap(g,ap2);
}
```

Question-8

Problem statement:

Given an array of N integers and Q queries in the form L, R, and K. Find the number of elements in range L, R which are strictly smaller than K. Time Complexity should be strictly below N^2.

Mergesort Tree:

Mergesort Tree is an efficient data-structure that builds a tree out of array of elements and solves range queries better as compared to the Brute Force implementation. The key idea is to build a Segment Tree with a vector at every node and the vector contains all the elements of the sub-range in a sorted order. And if we observe this segment tree structure this is somewhat similar to the tree formed during the merge sort algorithm(that is why it is called merge sort tree).

Implementation:

The tree structure is made as follows:-

Start in bottom up manner. Let's say we have an array

Array = [2,1,3,5,7,4,9,5].

The Pictorial representation of the process is as follows. At each index of tree we store a vector:-

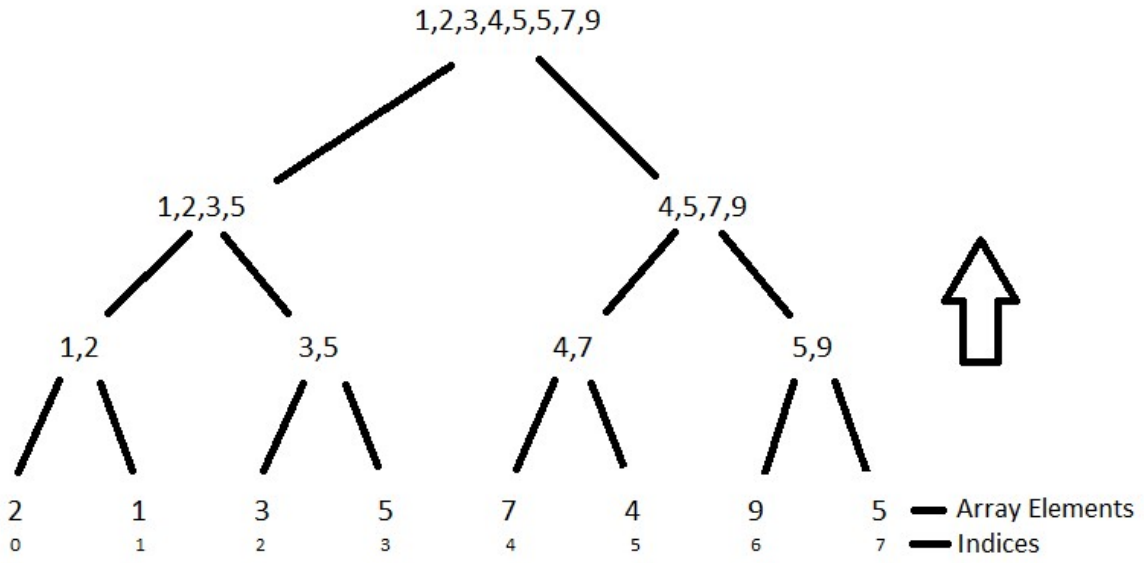

Fig 1

Step 1:- Initialise two array one for input array and one for a *tree-vector* say *arr, tree*. The dimension of which should be N and 4*N respectively. Where N is number of elements in array.

Step 2:- Write a *build tree* function. This function builds the tree in bottomup fashion. It should take 3 parameters: *index, start* and *end*. *index* represents index of segment tree, *start and end* are for indices of array which we call **segment**. Initally *index* will start at 0(the root node), it's child at 2**index* and 2**index* +1. And so on.

void buildtree(index, start, end) // Builds a tree in form of segment

> **Step 2.1**:- Base case is reached when we have only 1 element in a segment. Then that element would be inserted to index of *tree-vector*.
>
> *if start = end*
> *tree[index].insert(arr[start] or arr[end]).*
>
> *// insert function pushes the element in vector present at the index.*
>
> The base case yields the leaves of the Mergesort tree.

Step 2.2:- Find out the mid of a segment and recursively call *buidtree* for building subsequent segments.

mid = (start + end) / 2 // Assign mid to middle of array.

*buildtree(2*index, start, mid) // Left Subtree*
*buildtree(2*index+1, mid+1, end) // Right Subtree*

Step 2.3:- Recall Mergesort Algorithm where we do exactly this, we first divide and then conquer. Now we have 2 subtrees which are already built recursively.

Follow the Merge Procedure of Mergesort. The 2 vectors to be merged are *tree[2*index]* and *tree[2*index + 1]*.

As an example, after base case, we got [2] & [1] as elements who are merged to form [1,2]

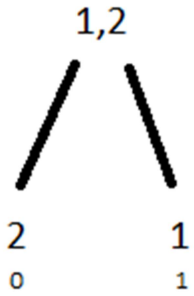

Fig 2

Follow standard Merge procedure of Mergesort Algorithm for the same.

This continues until the tree looks like Fig 1.

Step 3:- Write a *query* function

int query(index, start, end, L, R, K)

It is clear that now every node of segment tree is represented by an index, the root represent index [0 to N], it's left child [0 to N/2] and right one [N/2+1 to N] and so on.

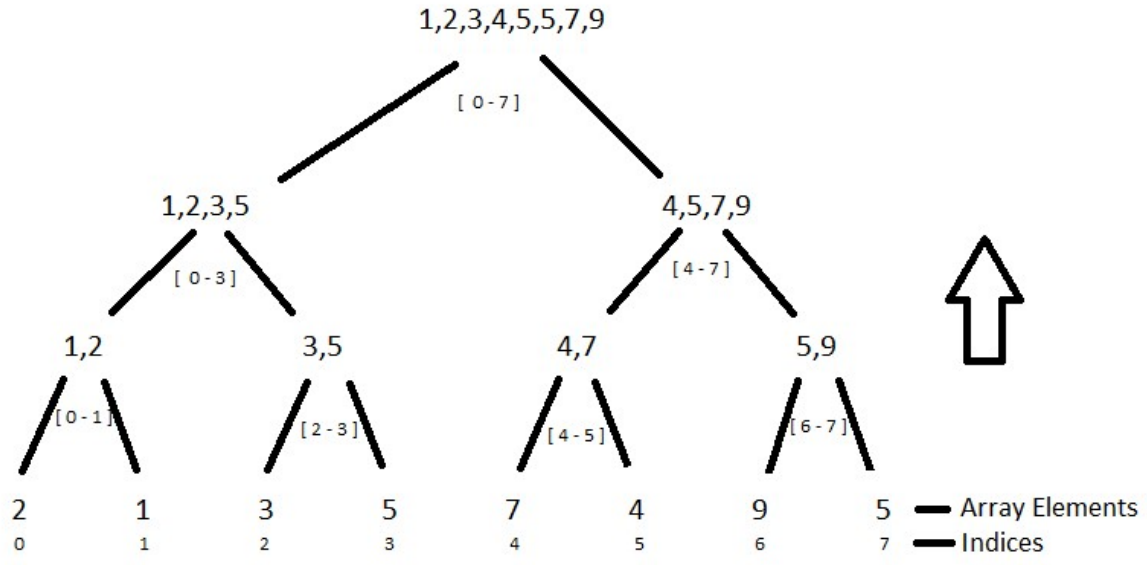

Fig 3

If segment of the node is a part of query range(L,R), then return
The number obtained by **Core logic** defined below**.**

If segment of the node is **outside the given range**(L,R) then return 0.

Finally, if a part of the segment **overlaps** with the given range(L,R) then traverse both Left and Right Sub-tree.

Core Logic:- Since the array in a segment is stored in sorted order, Binary Search can help find the element smaller than 'K' given in query. Return it's number.

Now what to do with Left and right calls numbers? Simply add and return!

Step 4:- Display the answer of the query function.

Evaluation Criteria:

Marking will be done based on the correctness of the whole procedure.
Plagiarism will be heavily penalized.

Implementation as per above steps only, 4 steps. [**40 marks**]

At time of evaluation, it will be tested against **5 Test cases**. [**20 marks**]

Analyze in terms of time and space complexity. [**10 marks**]

Viva during evaluation. Students are expected to know and explain the code and answer questions regarding the same. [**30 marks**]

Total: 100 marks

Important Note: NO marks will be awarded for N^2 time complexity solution. Implementation and evaluation of code/report **WILL NOT** be considered in Brute force(N^2) solution, rather will yield **0 mark**.

Example:-

Array: 3 5 3 6 8 9 3 2 1 4 3 6 7 8 9 10

Query: 1 5 4
Ans: 2

Query 7 10 4
Ans: 3

Question-9

a) Sketch a Binomial Heap data structure containing the four values {2, 4, 6, 8}. (20 marks)

(b) Sketch a Binomial Heap storing the thirteen values {1, 3, . . . , 25}(odd numbers). (25 marks)

Note that in parts (a) and (b) the values can be arranged in the heaps in several different ways, while still satisfying all the conditions required of a Binomial Heap. Your heaps will of course store the smallest values at the top, but you should explain how much flexibility there was beyond that and what policy you adopted in placing values.

(c) Form the union of the above two heaps, explaining the steps used and showing where the stored values end up. You do not need to display all the pointers in your data structures, and need not include any elaborate discussions of other operations on or applications of binomial trees or heaps. (30 marks)

Visualize a) and b) using Graphviz(25 marks).

Submission

To submit your assignment, you should:

- Create a C++ file that contains your implementation
- Test your implementation to ensure that it meets the requirements
- Write a short report that explains your implementation.
- Submit your C++ file and report.

Grading

Your assignment will be graded based on:

- Correctness: Your implementation should meet the requirements and pass the testing phase
- Readability: Your code should be well-structured, commented, and easy to read
- Efficiency: Your code should be efficient and avoid unnecessary overheads
- Report: Your report should explain your implementation clearly and concisely, and provide insights into your design decisions and challenges