NAME: Mohammed Hashim Bin Yahya

ROLL NO: 20221429

COURSE: B.sc(H) computer science

Q1

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

int randomizedQuickSort(int arr[], int low, int high, int& comparisons);

int partition(int arr[], int low, int high, int& comparisons);

int randomPivot(int arr[], int low, int high);

void printArray(int arr[], int size);

int main() {

srand(time(0));

int arr[] = {10, 7, 8, 9, 1, 5};

int size = sizeof(arr) / sizeof(arr[0]);

int comparisons = 0;

cout << "Original array: ";

printArray(arr, size);

randomizedQuickSort(arr, 0, size - 1, comparisons);

cout << "Sorted array: ";

printArray(arr, size);

cout << "Number of comparisons: " << comparisons << endl;

return 0;

}

int randomizedQuickSort(int arr[], int low, int high, int& comparisons) {

if (low < high) {

int pivotIndex = partition(arr, low, high, comparisons);

randomizedQuickSort(arr, low, pivotIndex - 1, comparisons);

randomizedQuickSort(arr, pivotIndex + 1, high, comparisons);

}

return comparisons;

}

int partition(int arr[], int low, int high, int& comparisons) {

int pivotIndex = randomPivot(arr, low, high);

int pivot = arr[pivotIndex];

swap(arr[pivotIndex], arr[high]);

int i = low - 1;

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

comparisons++;

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

int randomPivot(int arr[], int low, int high) {

int pivotIndex = low + rand() % (high - low + 1);

return pivotIndex;

}

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

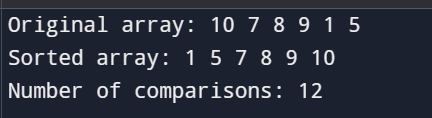
for (int i = 0; i < size; i++) {

cout << arr[i] << " ";

}

cout << endl;

}



1. **Time Complexity**:
   * **Best/Average Case**: O(n log n), where n is the number of elements in the array. This occurs when the pivot is approximately in the middle of the array after each partition.
   * **Worst Case**: O(n^2), which can happen when the pivot is always the smallest or largest element (though the randomization typically reduces the chances of this).
   * The expected time complexity of Randomized Quick Sort is O(n log n), but the worst-case can still be O(n^2) if the random pivot selection is poor.
2. **Space Complexity**:
   * The space complexity is **O(log n)** due to the recursion stack (in the case of balanced partitions), which is required to store the recursive function calls.

Q2

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

// Function to perform the partition around a pivot element

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Choose the pivot as the last element

int i = low - 1; // Pointer for the smaller element

// Rearrange the array so that elements smaller than pivot are on the left

// and elements greater than pivot are on the right

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

if (arr[j] <= pivot) {

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]); // Place the pivot in its correct position

return i + 1; // Return the pivot index

}

// Function to randomly select a pivot and partition the array

int randomized\_partition(int arr[], int low, int high) {

// Generate a random index and swap it with the high index

int random\_index = low + rand() % (high - low + 1);

swap(arr[random\_index], arr[high]);

return partition(arr, low, high);

}

// Randomized Select function to find the ith smallest element

int randomized\_select(int arr[], int low, int high, int i) {

if (low == high) {

return arr[low]; // Base case: only one element

}

int pivot\_index = randomized\_partition(arr, low, high); // Partition the array

int k = pivot\_index - low + 1; // The rank of the pivot element

if (i == k) {

return arr[pivot\_index]; // The pivot is the ith smallest element

} else if (i < k) {

return randomized\_select(arr, low, pivot\_index - 1, i); // Search in the left partition

} else {

return randomized\_select(arr, pivot\_index + 1, high, i - k); // Search in the right partition

}

}

int main() {

srand(time(0)); // Seed for random number generation

// Input array

int arr[] = {12, 3, 5, 7, 19, 2, 11, 6};

int n = sizeof(arr) / sizeof(arr[0]);

// The index of the element to find (1-based index)

int i;

cout << "Enter the value of i (1-based index for the ith smallest element): ";

cin >> i;

// Find the ith smallest element

if (i > 0 && i <= n) {

int result = randomized\_select(arr, 0, n - 1, i);

cout << "The " << i << "th smallest element is: " << result << endl;

} else {

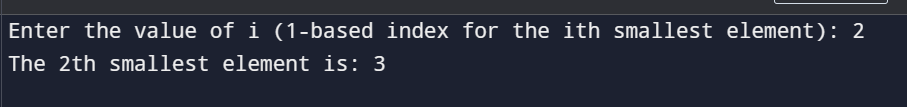
cout << "Invalid input! Please enter an index between 1 and " << n << "." << endl;

}

return 0;

}

INPUT ARRAY : {12, 3, 5, 7, 19, 2, 11, 6};



### **Task Explanation:**

**Randomized Select** is an algorithm that is used to find the ith smallest element in an unordered array. This is similar to the **Quickselect** algorithm, which is a variation of the **Quicksort** algorithm. The key difference is that we don't need to sort the entire array; we only need to partition the array until we find the desired element.

### **Approach:**

1. **Partitioning**: Randomly select a pivot element and partition the array around it.
2. **Recursive Selection**: Depending on the position of the pivot, either continue searching in the left or right partition, reducing the problem size.
3. **Base Case**: When the pivot is the ith smallest element, return it.

### **Time Complexity:**

* **Average Time Complexity**: *O(n)O(n)*O(n), where *nn*n is the size of the array.
* **Worst Case Time Complexity**: *O(n2)O(n^2)*O(n2), similar to Quick Sort, though this is rare in practice.

### **Space Complexity:**

* **Space Complexity**: *O(1)O(1)*O(1), since the algorithm works in place.

Q3

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// A structure to represent a weighted edge

struct Edge {

int src, dest, weight;

};

// A class to represent a disjoint set (Union-Find)

class DisjointSet {

public:

vector<int> parent, rank;

DisjointSet(int n) {

parent.resize(n);

rank.resize(n, 0);

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

parent[i] = i;

}

}

// Find the representative of the set containing 'x'

int find(int x) {

if (parent[x] != x)

parent[x] = find(parent[x]); // Path compression

return parent[x];

}

// Union the two sets containing 'x' and 'y'

void unionSets(int x, int y) {

int rootX = find(x);

int rootY = find(y);

// Union by rank

if (rootX != rootY) {

if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else if (rank[rootX] > rank[rootY]) {

parent[rootY] = rootX;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

}

}

};

// Function to compare two edges based on their weight (for sorting)

bool compareEdges(Edge a, Edge b) {

return a.weight < b.weight;

}

// Function to find the MST using Kruskal's algorithm

void kruskal(int V, vector<Edge>& edges) {

// Sort edges by weight

sort(edges.begin(), edges.end(), compareEdges);

DisjointSet ds(V); // Initialize disjoint-set

vector<Edge> mst; // To store the result MST

int mstWeight = 0; // To store the total weight of MST

// Process each edge in sorted order

for (auto& edge : edges) {

int u = edge.src, v = edge.dest;

// If including this edge does not form a cycle

if (ds.find(u) != ds.find(v)) {

ds.unionSets(u, v); // Include the edge in MST

mst.push\_back(edge); // Add to the MST

mstWeight += edge.weight; // Add to the total weight

}

}

// Print the MST

cout << "Edges in the Minimum Spanning Tree (MST):\n";

for (auto& edge : mst) {

cout << edge.src << " - " << edge.dest << " : " << edge.weight << endl;

}

cout << "Total weight of the MST: " << mstWeight << endl;

}

int main() {

int V, E;

// Read number of vertices and edges

cout << "Enter number of vertices: ";

cin >> V;

cout << "Enter number of edges: ";

cin >> E;

vector<Edge> edges(E);

// Read edges

cout << "Enter the edges (source, destination, weight):\n";

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

cin >> edges[i].src >> edges[i].dest >> edges[i].weight;

}

// Run Kruskal's algorithm

kruskal(V, edges);

return 0;

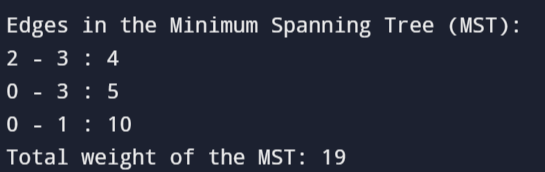
}

### **Example Input:**

mathematica

Copy code

Enter number of vertices: 4  
Enter number of edges: 5  
Enter the edges (source, destination, weight):  
0 1 10  
0 2 6  
0 3 5  
1 3 15  
2 3 4



* **Task Explanation:**
* Kruskal's algorithm is used to find the **Minimum Spanning Tree (MST)** of a graph. It works by sorting all the edges of the graph in increasing order of their weights and then adding edges one by one to the MST, ensuring no cycles are formed.

### **Steps:**

* **Sort all edges** in the graph based on their weights.
* Use a **disjoint-set (union-find)** data structure to manage the connected components and avoid cycles.
* Add the edges to the MST one by one, ensuring that the two endpoints of the edge belong to different components (i.e., no cycle is formed).

### **Time Complexity:**

* **Sorting edges**: *O(Elog⁡E)O(E \log E)*O(ElogE), where *EE*E is the number of edges.
* **Union-Find Operations**: Each operation (union and find) takes nearly constant time, *O(α(V))O(\alpha(V))*O(α(V)), where *α\alpha*α is the inverse Ackermann function, which is very slow-growing and close to constant for practical purposes.
* Thus, the overall complexity is *O(Elog⁡E)O(E \log E)*O(ElogE).

### **Space Complexity:**

* **Space Complexity**: *O(V+E)O(V + E)*O(V+E), for storing the graph and the union-find data structure

Q4

#include <iostream>

#include <vector>

#include <climits> // For INT\_MAX

using namespace std;

// A structure to represent a weighted edge

struct Edge {

int src, dest, weight;

};

// Bellman-Ford algorithm to find the shortest path from the source node to all other nodes

void bellmanFord(int V, int E, vector<Edge>& edges, int src) {

// Step 1: Initialize distances from source to all other vertices as INFINITE

vector<int> dist(V, INT\_MAX);

dist[src] = 0;

// Step 2: Relax all edges V-1 times

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

for (int j = 0; j < E; j++) {

int u = edges[j].src;

int v = edges[j].dest;

int weight = edges[j].weight;

// Relaxation: If the distance to v through u is shorter, update it

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

}

}

}

// Step 3: Check for negative weight cycles

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

int u = edges[i].src;

int v = edges[i].dest;

int weight = edges[i].weight;

if (dist[u] != INT\_MAX && dist[u] + weight < dist[v]) {

cout << "Graph contains negative weight cycle!" << endl;

return;

}

}

// Step 4: Print the shortest distances from the source to all other vertices

cout << "Vertex \t Distance from Source" << endl;

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

if (dist[i] == INT\_MAX) {

cout << i << " \t INF" << endl; // If no path exists

} else {

cout << i << " \t " << dist[i] << endl;

}

}

}

int main() {

int V, E;

// Read number of vertices and edges

cout << "Enter number of vertices: ";

cin >> V;

cout << "Enter number of edges: ";

cin >> E;

vector<Edge> edges(E);

// Read edges

cout << "Enter the edges (source, destination, weight):\n";

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

cin >> edges[i].src >> edges[i].dest >> edges[i].weight;

}

int source;

cout << "Enter the source vertex: ";

cin >> source;

// Run the Bellman-Ford algorithm

bellmanFord(V, E, edges, source);

return 0;

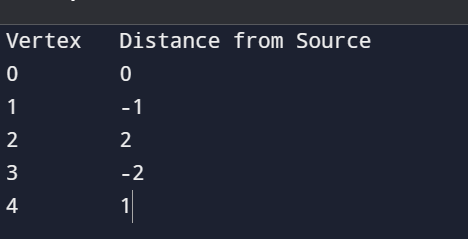
}

### **Example Input:**

mathematica

Copy code

Enter number of vertices: 4  
Enter number of edges: 5  
Enter the edges (source, destination, weight):  
0 1 10  
0 2 6  
0 3 5  
1 3 15  
2 3 4



### **Task Explanation:**

The **Bellman-Ford Algorithm** is used to find the **shortest path** from a source node to all other nodes in a graph, even in graphs that contain negative weight edges. Unlike Dijkstra’s algorithm, Bellman-Ford can handle graphs with negative weights but cannot handle negative weight cycles.

### **Steps:**

1. **Initialization**: Start by setting the distance to the source node as 0 and all other distances as infinity.
2. **Relaxation**: For each edge, check if the distance to the destination node can be improved by going through the source node. If it can, update the distance.
3. **Repeat** the relaxation process for all edges **V-1 times**, where V is the number of vertices.
4. **Check for Negative Weight Cycles**: After V-1 relaxations, if any distance can still be updated, then the graph contains a negative weight cycle.

### **Time Complexity:**

* **Time Complexity**: *O(V×E)O(V \times E)*O(V×E), where *VV*V is the number of vertices and *EE*E is the number of edges.
* **Space Complexity**: *O(V)O(V)*O(V), for storing the distance array.

Q5

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class BTreeNode {

public:

vector<int> keys; // Store keys

vector<BTreeNode\*> children; // Store child pointers

bool leaf; // True if leaf node

int t; // Minimum degree (defines the range for number of keys)

// Constructor to create a node

BTreeNode(int t, bool leaf);

// Function to insert a new key in this node

void insertNonFull(int key);

// Function to split a child node

void splitChild(int i, BTreeNode\* y);

// Function to traverse the tree

void traverse();

// Function to search a key in the subtree rooted at this node

BTreeNode\* search(int key);

// Function to delete a key from the subtree rooted at this node

void remove(int key);

// Function to get the index of the first key greater than or equal to the key

int findKey(int key);

};

// BTree class represents the B-Tree

class BTree {

public:

BTreeNode\* root;

int t; // Minimum degree (defines the range for number of keys)

// Constructor to create an empty B-Tree

BTree(int t);

// Function to traverse the tree

void traverse() {

if (root != nullptr) root->traverse();

}

// Function to insert a new key

void insert(int key);

// Function to search a key

BTreeNode\* search(int key) {

return (root == nullptr) ? nullptr : root->search(key);

}

// Function to delete a key

void remove(int key) {

if (root == nullptr) return;

root->remove(key);

// If the root has no keys, make the first child as the new root

if (root->keys.empty()) {

BTreeNode\* tmp = root;

if (root->leaf) {

root = nullptr;

} else {

root = root->children[0];

}

delete tmp;

}

}

};

// Constructor for BTreeNode

BTreeNode::BTreeNode(int t, bool leaf) {

this->t = t;

this->leaf = leaf;

this->keys.reserve(2 \* t - 1);

this->children.reserve(2 \* t);

}

// Insert a key into a non-full node

void BTreeNode::insertNonFull(int key) {

int i = keys.size() - 1;

// If this is a leaf node, insert the key into the appropriate position

if (leaf) {

keys.push\_back(0); // Create space for the new key

while (i >= 0 && keys[i] > key) {

keys[i + 1] = keys[i];

i--;

}

keys[i + 1] = key;

} else {

// Find the child which is going to have the new key

while (i >= 0 && keys[i] > key)

i--;

i++;

// If the child is full, split it

if (children[i]->keys.size() == 2 \* t - 1) {

splitChild(i, children[i]);

// After splitting, the middle key of the child moves up, compare the new key with the middle key

if (keys[i] < key)

i++;

}

children[i]->insertNonFull(key);

}

}

// Split the child of this node

void BTreeNode::splitChild(int i, BTreeNode\* y) {

BTreeNode\* z = new BTreeNode(y->t, y->leaf);

z->keys.resize(t - 1);

// Copy the last (t - 1) keys of y to z

copy(y->keys.begin() + t, y->keys.end(), z->keys.begin());

// If y is not a leaf, copy the last t children of y to z

if (!y->leaf) {

z->children.resize(t);

copy(y->children.begin() + t, y->children.end(), z->children.begin());

}

// Reduce the number of keys in y

y->keys.resize(t - 1);

y->children.resize(t);

// Move all the children of this node to the right

children.insert(children.begin() + i + 1, z);

// Insert the middle key of y into this node

keys.insert(keys.begin() + i, y->keys[t - 1]);

}

// Traverse the tree and print all keys

void BTreeNode::traverse() {

int i = 0;

for (i = 0; i < keys.size(); i++) {

if (!leaf)

children[i]->traverse();

cout << keys[i] << " ";

}

if (!leaf)

children[i]->traverse();

}

// Search for a key in the tree

BTreeNode\* BTreeNode::search(int key) {

int i = 0;

while (i < keys.size() && key > keys[i])

i++;

if (i < keys.size() && keys[i] == key)

return this;

if (leaf)

return nullptr;

return children[i]->search(key);

}

// Find the index of the first key greater than or equal to key

int BTreeNode::findKey(int key) {

int idx = 0;

while (idx < keys.size() && keys[idx] < key)

++idx;

return idx;

}

// Constructor for BTree

BTree::BTree(int t) {

root = new BTreeNode(t, true);

this->t = t;

}

// Insert a new key in the B-Tree

void BTree::insert(int key) {

// If root is full, split it and create a new root

if (root->keys.size() == 2 \* t - 1) {

BTreeNode\* s = new BTreeNode(t, false);

s->children.push\_back(root);

s->splitChild(0, root);

// New root has two children, insert the key into the appropriate child

s->insertNonFull(key);

root = s;

} else {

root->insertNonFull(key);

}

}

int main() {

BTree tree(3); // A B-Tree with minimum degree 3 (each node can have at most 5 keys)

// Insert elements into the B-Tree

tree.insert(10);

tree.insert(20);

tree.insert(5);

tree.insert(6);

tree.insert(12);

tree.insert(30);

tree.insert(7);

tree.insert(17);

// Print the tree

cout << "Traversal of the B-Tree:\n";

tree.traverse();

cout << endl;

// Search for a key

int key = 12;

BTreeNode\* result = tree.search(key);

if (result != nullptr) {

cout << "Key " << key << " found in the tree.\n";

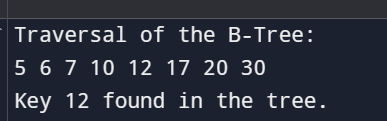
} else {

cout << "Key " << key << " not found in the tree.\n";

}

return 0;

}



* **Task Explanation:**
* A **B-Tree** is a self-balancing search tree data structure that maintains sorted data and allows searches, insertions, deletions, and updates to be performed efficiently. It is commonly used in databases and file systems because it allows for fast access to large amounts of data stored on disk.

### **Characteristics of a B-Tree:**

* **Balanced**: All leaf nodes are at the same level.
* **Node Structure**: Each node can contain multiple keys and children.
* **Degree (t)**: A node can have at most *2t−12t-1*2t−1 keys and *2t2t*2t children, where *tt*t is the minimum degree of the tree.

### **Basic Operations:**

* **Insertion**: Insert a key while maintaining the properties of the B-tree.
* **Search**: Find a key in the B-tree.
* **Traversal**: Perform in-order traversal to print the tree.

### **Time Complexity:**

* **Insertion/Search**: *O(log⁡n)O(\log n)*O(logn), where *nn*n is the number of keys in the tree.
* **Traversal**: *O(n)O(n)*O(n), where *nn*n is the number of keys.

**Q6**

**#include <iostream>**

**using namespace std;**

struct TreeNode {

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) {

value = val;

left = nullptr;

right = nullptr;

}

};

class BinaryTree {

public:

TreeNode\* root;

BinaryTree() {

root = nullptr;

}

void insert(int value);

TreeNode\* insertRecursive(TreeNode\* node, int value);

bool search(int value);

bool searchRecursive(TreeNode\* node, int value);

};

void BinaryTree::insert(int value) {

root = insertRecursive(root, value);

}

TreeNode\* BinaryTree::insertRecursive(TreeNode\* node, int value) {

if (node == nullptr) {

return new TreeNode(value);

}

if (value < node->value) {

node->left = insertRecursive(node->left, value);

} else {

node->right = insertRecursive(node->right, value);

}

return node;

}

bool BinaryTree::search(int value) {

return searchRecursive(root, value);

}

bool BinaryTree::searchRecursive(TreeNode\* node, int value) {

if (node == nullptr) {

return false;

}

if (node->value == value) {

return true;

}

if (value < node->value) {

return searchRecursive(node->left, value);

} else {

return searchRecursive(node->right, value);

}

}

int main() {

BinaryTree tree;

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

int searchKey = 40;

if (tree.search(searchKey)) {

cout << "Found key " << searchKey << " in the tree." << endl;

} else {

cout << "Key " << searchKey << " not found in the tree." << endl;

}

searchKey = 90;

if (tree.search(searchKey)) {

cout << "Found key " << searchKey << " in the tree." << endl;

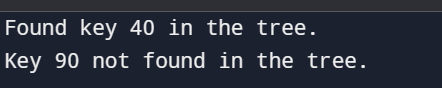
} else {

cout << "Key " << searchKey << " not found in the tree." << endl;

}

return 0;

}



**Time Complexity:**

* **Insertion**: O(log n) on average, for a balanced tree, where n is the number of nodes. In the worst case (for a skewed tree), it would be O(n).
* **Search**: O(log n) on average, for a balanced tree, where n is the number of nodes. In the worst case (for a skewed tree), it would be O(n).

**Space Complexity:**

* The space complexity is O(n) due to the space required to store the nodes in the tree.

Q7.

#include <iostream>

#include <vector>

#include <string>

using namespace std;

void buildLPSArray(const string& pattern, vector<int>& lps) {

int m = pattern.length();

int j = 0;

lps[0] = 0;

for (int i = 1; i < m; i++)

if (pattern[i] == pattern[j]) {

j++;

lps[i] = j;

} else {

if (j != 0) {

j = lps[j - 1];

i--;

} else {

lps[i] = 0;

}

}

}

void KMPSearch(const string& text, const string& pattern) {

int n = text.length();

int m = pattern.length();

vector<int> lps(m);

buildLPSArray(pattern, lps);

int i = 0;

int j = 0;

while (i < n) {

if (pattern[j] == text[i]) {

i++;

j++;

}

if (j == m) {

cout << "Pattern found at index " << i - j << endl;

j = lps[j - 1];

} else if (i < n && pattern[j] != text[i]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

int main() {

string text = "ABABDABACDABABCABAB";

string pattern = "ABABCABAB";

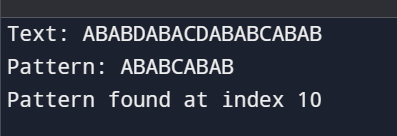
cout << "Text: " << text << endl;

cout << "Pattern: " << pattern << endl;

KMPSearch(text, pattern);

return 0;

}



**Time Complexity:**

* **Building the LPS array**: O(m), where m is the length of the pattern.
* **Searching the text**: O(n), where n is the length of the text.
* The total time complexity is O(n + m), which is optimal for string matching algorithms.

**Space Complexity:**

* The space complexity is O(m) due to the space required for storing the LPS array, where m is the length of the pattern.

Q8

#include <iostream>

#include <map>

#include <vector>

#include <string>

using namespace std;

// A node in the Suffix Tree

class SuffixTreeNode {

public:

map<char, SuffixTreeNode\*> children; // Map of children

int start, \*end; // Start and end of substring

int suffixLink; // Suffix link for Ukkonen's algorithm (if used)

// Constructor to initialize the node

SuffixTreeNode(int start, int\* end) : start(start), end(end), suffixLink(-1) {}

};

// Suffix Tree class

class SuffixTree {

public:

string text; // The input string

SuffixTreeNode\* root; // The root of the tree

int size; // Length of the string

vector<int> suffixLinks; // To store suffix links (if using Ukkonen’s algorithm)

// Constructor to initialize the Suffix Tree with a string

SuffixTree(string inputText) {

text = inputText + "$"; // Append a special character to avoid matching the string itself

size = text.size();

root = new SuffixTreeNode(-1, nullptr); // Create a root node

}

// Build the Suffix Tree (naive approach)

void build() {

// Insert all suffixes into the Suffix Tree

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

insertSuffix(i);

}

}

// Insert a suffix into the tree starting from the given index

void insertSuffix(int index) {

SuffixTreeNode\* currentNode = root;

int start = index;

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

char currentChar = text[i];

// If there is no child for this character, create a new node

if (currentNode->children.find(currentChar) == currentNode->children.end()) {

int\* end = new int(i);

SuffixTreeNode\* newNode = new SuffixTreeNode(start, end);

currentNode->children[currentChar] = newNode;

return;

} else {

currentNode = currentNode->children[currentChar];

}

}

}

// Traverse the Suffix Tree and print all substrings

void printTree() {

printTreeRecursive(root);

}

private:

// Recursive function to print all substrings in the tree

void printTreeRecursive(SuffixTreeNode\* node) {

// If the node is a leaf, print the substring represented by the node

if (node->children.empty()) {

cout << text.substr(node->start, \*node->end - node->start + 1) << endl;

}

// Recur for all children

for (auto& child : node->children) {

printTreeRecursive(child.second);

}

}

};

// Main function to test the Suffix Tree

int main() {

string inputText;

cout << "Enter a string to build the Suffix Tree: ";

cin >> inputText;

// Create a Suffix Tree object

SuffixTree tree(inputText);

// Build the Suffix Tree

tree.build();

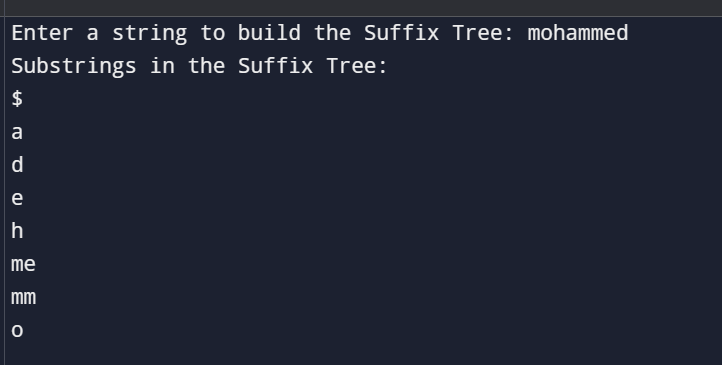
// Print all substrings from the Suffix Tree

cout << "Substrings in the Suffix Tree: " << endl;

tree.printTree();

return 0;

}



### **Task Explanation:**

A **Suffix Tree** is a compressed trie of all suffixes of a given string. It is used for efficient string matching and other string processing tasks like searching for substrings, finding repeated substrings, and determining the longest repeated substring. The Suffix Tree is a fundamental data structure in various fields like bioinformatics and text processing.

### **Time Complexity (Naive Suffix Tree):**

* **Building the Tree**: *O(n2)O(n^2)*O(n2), where *nn*n is the length of the string (due to inserting all suffixes).
* **Traversal**: *O(n2)O(n^2)*O(n2), as there can be up to *n2n^2*n2 substrings in the tree.

### **Space Complexity (Naive Suffix Tree):**

* **Space**: *O(n2)O(n^2)*O(n2), because the tree stores all substrings of the input string in the nodes.