PRACTICAL ASSIGNMENT

(ATMA RAM SANATAN DHARMA COLLEGE)

(ADVANCED ALGORITHMS AND ANALYSIS)

- **NAME – TUSHAR**
- **ROLL NUMBER – 22/28087**
- **SUBMITTED TO – MRS ARCHANA GAHLAUT MA’AM**
- **SUBJECT- ADVANCED ALGORITHMS AND ANALYSIS**
- **COURSE – BSC(HONS) COMPUTER SCIENCE**

Q1. Write a program to sort the elements of an array using Randomized Quick Sort (the program should report the number of comparisons).

```cpp
#include <iostream>
#include <vector>
#include <cstdlib> // for rand()
#include <ctime>   // for time()

using namespace std;

// Function to perform randomized quick sort
pair<vector<int>, int> randomized_quick_sort(const vector<int>& arr) {
    if (arr.size() <= 1) {
        return {arr, 0}; // Base case: return the array and 0 comparisons
    }

    // Randomly select a pivot
    int pivot_index = rand() % arr.size();
    int pivot = arr[pivot_index];

    vector<int> less_than_pivot;
    vector<int> equal_to_pivot;
    vector<int> greater_than_pivot;

    int comparisons = 0;

    for (const int& element : arr) {
        comparisons++; // Count comparison
        if (element < pivot) {
            less_than_pivot.push_back(element);
        } else if (element > pivot) {
            greater_than_pivot.push_back(element);
        } else {
            equal_to_pivot.push_back(element);
        }
    }

    // Recursively sort the sub-arrays
    pair<vector<int>, int> sorted_less = randomized_quick_sort(less_than_pivot);
    pair<vector<int>, int> sorted_greater = randomized_quick_sort(greater_than_pivot);

    // Combine the sorted sub-arrays and the pivot
    vector<int> sorted_array = sorted_less.first;
    sorted_array.insert(sorted_array.end(), equal_to_pivot.begin(), equal_to_pivot.end());
    sorted_array.insert(sorted_array.end(), sorted_greater.first.begin(), sorted_greater.first.end());

    // Calculate the total number of comparisons
    int total_comparisons = comparisons + sorted_less.second + sorted_greater.second;

    return {sorted_array, total_comparisons};
}

int main() {
    srand(static_cast<unsigned int>(time(0))); // Seed for random number generation

    vector<int> array = {34, 7, 23, 32, 5, 62};
    pair<vector<int>, int> result = randomized_quick_sort(array);

    cout << "Sorted Array: ";
    for (const int& num : result.first) {
        cout << num << " ";
    }
    cout << endl;
    cout << "Number of Comparisons: " << result.second << endl;

    return 0;
}
```

OUTPUT TERMINAL:

```
Sorted Array: 5 7 23 32 34 62
Number of Comparisons: 13
```

Q2. Write a program to find the ith smallest element of an array using Randomized Select.

```cpp
#include <iostream>
#include <vector>
#include <cstdlib> // for rand()
#include <ctime>   // for time()
using namespace std;
// Function to partition the array around the pivot
int partition(vector<int>& arr, int low, int high, int pivotIndex) {
    int pivotValue = arr[pivotIndex];
    swap(arr[pivotIndex], arr[high]); // Move pivot to end
    int storeIndex = low;

    for (int i = low; i < high; i++) {
        if (arr[i] < pivotValue) {
            swap(arr[storeIndex], arr[i]);
            storeIndex++;
        }
    }
    swap(arr[storeIndex], arr[high]); // Move pivot to its final place
    return storeIndex; // Return the final position of the pivot
}

// Randomized Select function to find the i-th smallest element
int randomized_select(vector<int>& arr, int low, int high, int i) {
    if (low == high) {
        return arr[low]; // If the list contains only one element
    }

    // Select a random pivot index
    int pivotIndex = low + rand() % (high - low + 1);
    pivotIndex = partition(arr, low, high, pivotIndex);

    // The pivot is in its final sorted position
    if (i == pivotIndex) {
        return arr[i]; // Found the i-th smallest element
    } else if (i < pivotIndex) {
        return randomized_select(arr, low, pivotIndex - 1, i); // Search left
    } else {
        return randomized_select(arr, pivotIndex + 1, high, i); // Search right
    }
}
int main() {
    srand(static_cast<unsigned int>(time(0))); // Seed for random number generation

    vector<int> array = {34, 7, 23, 32, 5, 62};
    int i = 3; // For example, find the 3rd smallest element (0-based index)

    if (i < 0 || i >= array.size()) {
        cout << "Index out of bounds." << endl;
        return 1;
    }

    int result = randomized_select(array, 0, array.size() - 1, i);
    cout << "The " << i + 1 << "th smallest element is: " << result << endl;

    return 0;
}
```

OUTPUT TERMINAL:

```
The 4th smallest element is: 32
```

Q3. Write a program to determine the minimum spanning tree of a graph using Kruskal's algorithm.

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Structure to represent an edge with weight
struct Edge {
    int u, v, weight;
};

// Compare edges based on their weights
bool compareEdge(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Class to represent a graph
class Graph {
    int V; // Number of vertices
    vector<Edge> edges; // List of edges

public:
    Graph(int V) {
        this->V = V;
    }

    // Add an edge to the graph
    void addEdge(int u, int v, int weight) {
        edges.push_back({u, v, weight});
    }

    // Find function with path compression
    int find(vector<int>& parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]);
        }
        return parent[i];
    }

    // Union function by rank
    void unionSets(vector<int>& parent, vector<int>& rank, int x, int y) {
        int rootX = find(parent, x);
        int rootY = find(parent, y);

        if (rank[rootX] < rank[rootY]) {
            parent[rootX] = rootY;
        } else if (rank[rootX] > rank[rootY]) {
            parent[rootY] = rootX;
        } else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }

```

```cpp
    // Function to perform Kruskal's algorithm
    void kruskalMST() {
        // Sort all edges based on their weights
        sort(edges.begin(), edges.end(), compareEdge);

        vector<int> parent(V);
        vector<int> rank(V, 0);

        // Initialize disjoint sets
        for (int i = 0; i < V; i++) {
            parent[i] = i;
        }

        vector<Edge> mst; // Store the edges included in the MST

        for (const auto& edge : edges) {
            int u = edge.u;
            int v = edge.v;

            int setU = find(parent, u);
            int setV = find(parent, v);

            // If u and v are in different sets, include this edge in MST
            if (setU != setV) {
                mst.push_back(edge);
                unionSets(parent, rank, setU, setV);
            }
        }

        // Print the edges in the MST
        cout << "Edges in the Minimum Spanning Tree:\n";
        for (const auto& edge : mst) {
            cout << edge.u << " - " << edge.v << " : " << edge.weight << endl;
        }
    }
};

int main() {
    int V = 4; // Number of vertices
    Graph g(V);

    g.addEdge(0, 1, 10);
    g.addEdge(0, 2, 6);
    g.addEdge(0, 3, 5);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);

    g.kruskalMST();

    return 0;
}
```

OUTPUT TERMINAL:

```
Edges in the Minimum Spanning Tree:
2 - 3 : 4
0 - 3 : 5
0 - 1 : 10
```

Q4. Write a program to implement the Bellman-Ford algorithm to find the shortest paths from a given source node to all other nodes in a graph.

```cpp
#include <iostream>
#include <vector>
#include <climits> // for INT_MAX

using namespace std;

// Edge structure to store information about a graph edge (u -> v with weight w)
struct Edge {
    int u, v, weight;
};

// Function to implement Bellman-Ford algorithm
bool bellmanFord(int vertices, int edges, int source, const vector<Edge>& graph, vector<int>& distances) {
    // Initialize distances from source to all vertices as infinite (INT_MAX)
    distances.assign(vertices, INT_MAX);
    distances[source] = 0;  // Distance from source to itself is always 0

    // Relax all edges (vertices-1) times
    for (int i = 0; i < vertices - 1; i++) {
        for (int j = 0; j < edges; j++) {
            int u = graph[j].u;
            int v = graph[j].v;
            int weight = graph[j].weight;

            if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
                distances[v] = distances[u] + weight;
            }
        }
    }

    // Check for negative weight cycles by relaxing edges one more time
    for (int j = 0; j < edges; j++) {
        int u = graph[j].u;
        int v = graph[j].v;
        int weight = graph[j].weight;

        if (distances[u] != INT_MAX && distances[u] + weight < distances[v]) {
            // Negative weight cycle detected
            return false;
        }
    }

    return true;
}

int main() {
    int vertices, edges;
    cout << "Enter number of vertices and edges: ";
    cin >> vertices >> edges;

    vector<Edge> graph(edges);

    cout << "Enter edges (u v weight):" << endl;
    for (int i = 0; i < edges; i++) {
        cin >> graph[i].u >> graph[i].v >> graph[i].weight;
    }

    int source;
    cout << "Enter source vertex: ";
    cin >> source;

    vector<int> distances(vertices);

    // Run Bellman-Ford algorithm
    if (bellmanFord(vertices, edges, source, graph, distances)) {
```

```
        cout << "Shortest distances from source " << source << " are:" << endl;
        for (int i = 0; i < vertices; i++) {
            if (distances[i] == INT_MAX) {
                cout << "Vertex " << i << ": No path" << endl;
            } else {
                cout << "Vertex " << i << ": " << distances[i] << endl;
            }
        }
    } else {
        cout << "Graph contains a negative weight cycle!" << endl;
    }

    return 0;
}
```

OUTPUT TERMINAL:

```
Enter number of vertices and edges: 6 9
Enter edges (u v weight):
0 1 6
0 2 4
0 3 5
3 2 -2
2 1 -2
3 5 -1
1 4 -1
2 4 3
4 5 3
Enter source vertex: 0
Shortest distances from source 0 are:
Vertex 0: 0
Vertex 1: 1
Vertex 2: 3
Vertex 3: 5
Vertex 4: 0
Vertex 5: 3
```

Q5. Write a program to implement a B-Tree.

```
#include <iostream>
using namespace std;

class Node {
    int *keys; // Array of keys
    int t; // Minimum degree
    Node **C; // Array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when the node is a leaf

public:
    Node(int _t, bool _leaf); // Constructor

    void insertNonFull(int k);
    void splitChild(int i, Node *y);
    void traverse(); // Function to traverse the nodes

    friend class BTree;
};

class BTree {
    Node *root; // Pointer to root node
    int t; // Minimum degree

```

```cpp
public:
    BTree(int _t) {
        root = NULL;
        t = _t;
    }

    void traverse() {
        if (root != NULL)
            root->traverse();
    }

    void insert(int k);
};

Node::Node(int t1, bool leaf1) {
    t = t1;
    leaf = leaf1;
    keys = new int[2 * t - 1];
    C = new Node *[2 * t];
    n = 0;
}

// Traverse the nodes in the B-Tree
void Node::traverse() {
    int i;
    for (i = 0; i < n; i++) {
        if (!leaf)
            C[i]->traverse();
        cout << " " << keys[i];
    }
    if (!leaf)
        C[i]->traverse();
}

// Insert a key into the B-Tree
void BTree::insert(int k) {
    if (root == NULL) {
        root = new Node(t, true);
        root->keys[0] = k;
        root->n = 1;
    } else {
        if (root->n == 2 * t - 1) {
            Node *s = new Node(t, false);
            s->C[0] = root;
            s->splitChild(0, root);
            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->C[i]->insertNonFull(k);
            root = s;
        } else
            root->insertNonFull(k);
    }
}

// Insert key in a non-full node
void Node::insertNonFull(int k) {
    int i = n - 1;

    if (leaf) {
        while (i >= 0 && keys[i] > k) {
            keys[i + 1] = keys[i];
            i--;
        }
        keys[i + 1] = k;
        n = n + 1;
```

```cpp
    } else {
        while (i >= 0 && keys[i] > k)
            i--;

        if (C[i + 1]->n == 2 * t - 1) {
            splitChild(i + 1, C[i + 1]);
            if (keys[i + 1] < k)
                i++;
        }
        C[i + 1]->insertNonFull(k);
    }
}
// Split a full child
void Node::splitChild(int i, Node *y) {
    Node *z = new Node(y->t, y->leaf);
    z->n = t - 1;

    for (int j = 0; j < t - 1; j++)
        z->keys[j] = y->keys[j + t];

    if (!y->leaf) {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j + t];
    }

    y->n = t - 1;
    for (int j = n; j >= i + 1; j--)
        C[j + 1] = C[j];

    C[i + 1] = z;

    for (int j = n - 1; j >= i; j--)
        keys[j + 1] = keys[j];

    keys[i] = y->keys[t - 1];
    n = n + 1;
}
int main() {
    int t;
    cout << "Enter the minimum degree of the B-Tree: ";
    cin >> t;
    BTree btree(t);

    int choice, key;
    do {
        cout << "\n1. Insert key\n2. Traverse B-Tree\n3. Exit\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter the key to insert: ";
                cin >> key;
                btree.insert(key);
                break;
            case 2:
                cout << "B-Tree traversal: ";
                btree.traverse();
                cout << endl;
                break;
            case 3:
                cout << "Exiting..." << endl;
                break;
            default:
                cout << "Invalid choice, please try again." << endl;
        }
    } while (choice != 3);
    return 0;
}
```

OUTPUT TERMINAL:

```
Enter the minimum degree of the B-Tree: 2

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 80

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 82

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 86

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 78

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 2

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 3

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 4
```

```
Enter your choice: 1
Enter the key to insert: 4

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 5

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 6

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 7

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 8

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 9

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 1
Enter the key to insert: 100

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 2
B-Tree traversal:  2 3 4 5 6 7 8 9 78 80 82 86 100

1. Insert key
2. Traverse B-Tree
3. Exit
Enter your choice: 3
Exiting...
PS C:\Users\tusha\OneDrive\Desktop\CODING\c++\Advanced_Analysis>
```

Q6. Write a program to implement the Trie Data structure, which supports the following operations:

a. Insertion

b. Deletion

```
#include <iostream>
#include <string>
using namespace std;

const int ALPHABET_SIZE = 26;

struct TrieNode {
    TrieNode* children[ALPHABET_SIZE];
    bool isEndOfWord;

    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < ALPHABET_SIZE; i++)
            children[i] = nullptr;
    }
};

void insert(TrieNode* root, const string& key) {
    TrieNode* currentNode = root;
    for (char c : key) {
        int index = c - 'a';
        if (!currentNode->children[index]) {
            currentNode->children[index] = new TrieNode();
        }
        currentNode = currentNode->children[index];
    }
    currentNode->isEndOfWord = true;
}

bool hasChildren(TrieNode* node) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (node->children[i])
            return true;
    }
    return false;
}

bool deleteWord(TrieNode* root, const string& key, int depth = 0) {
    if (!root)
        return false;

```

```cpp
    if (depth == key.size()) {
        if (root->isEndOfWord) {
            root->isEndOfWord = false;  // Unmark the end of word
            return !hasChildren(root);  // If no children, it can be deleted
        }
        return false; // Word does not exist
    }

    int index = key[depth] - 'a';
    if (deleteWord(root->children[index], key, depth + 1)) {
        delete root->children[index];
        root->children[index] = nullptr;

        return !root->isEndOfWord && !hasChildren(root);
    }

    return false;
}

bool search(TrieNode* root, const string& key) {
    TrieNode* currentNode = root;
    for (char c : key) {
        int index = c - 'a';
        if (!currentNode->children[index])
            return false;
        currentNode = currentNode->children[index];
    }
    return currentNode != nullptr && currentNode->isEndOfWord;
}

int main() {
    TrieNode* root = new TrieNode();
    int choice;
    string word;

    do {
        cout << "\nTrie Operations Menu:\n";
        cout << "1. Insert a word\n";
        cout << "2. Search for a word\n";
        cout << "3. Delete a word\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter word to insert: ";
                cin >> word;
                insert(root, word);
                cout << "'" << word << "' has been inserted into the Trie.\n";
                break;

            case 2:
                cout << "Enter word to search: ";
                cin >> word;
                if (search(root, word)) {
                    cout << "'" << word << "' is found in the Trie.\n";
                } else {
                    cout << "'" << word << "' is not found in the Trie.\n";
                }
                break;

            case 3:
                cout << "Enter word to delete: ";
                cin >> word;
                if (deleteWord(root, word)) {
                    cout << "'" << word << "' has been deleted from the Trie.\n";
                } else {
                    cout << "'" << word << "' was not found in the Trie or could not be deleted.\n";
                }
                break;

```

```cpp
            case 4:
                cout << "Exiting program.\n";
                break;

            default:
                cout << "Invalid choice! Please try again.\n";
                break;
        }
    } while (choice != 4);

    return 0;
}
```

OUTPUT TERMINAL:

```
Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 1
Enter word to insert: tushar
'tushar' has been inserted into the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 1
Enter word to insert: tree
'tree' has been inserted into the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 1
Enter word to insert: trie
'trie' has been inserted into the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 2
Enter word to search: tushar
'tushar' is found in the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 2
Enter word to search: trip
'trip' is not found in the Trie.
```

```
Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 3
Enter word to delete: tushar
'tushar' has been deleted from the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 2
Enter word to search: tushar
'tushar' is not found in the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 3
Enter word to delete: hello
'hello' was not found in the Trie or could not be deleted.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 3
Enter word to delete: tree
'tree' has been deleted from the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 2
Enter word to search: trie
'trie' is found in the Trie.

Trie Operations Menu:
1. Insert a word
2. Search for a word
3. Delete a word
4. Exit
Enter your choice: 4
Exiting program.
```

Q7. Write a program to search a pattern in a given text using the KMP algorithm.

```cpp
#include <iostream>
#include <vector>
using namespace std;

// Function to compute the LPS (Longest Prefix Suffix) array
void computeLPSArray(string pattern, vector<int>& lps) {
    int length = 0;
    int i = 1;
    lps[0] = 0; // LPS for the first character is always 0

    // Build the LPS array
    while (i < pattern.length()) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// Function to perform KMP search
void KMPSearch(string text, string pattern) {
    int m = pattern.length();
    int n = text.length();

    // Create the LPS array for the pattern
    vector<int> lps(m);
    computeLPSArray(pattern, lps);

    int i = 0; // index for text
    int j = 0; // index for pattern

    // Iterate through the text to search for the pattern
    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }

        if (j == m) {
            // Pattern found at index (i - j)
            cout << "Pattern found at index: " << i - j << endl;
            j = lps[j - 1];
        }

        // Mismatch after j matches
        else if (i < n && pattern[j] != text[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
}

int main() {
    string text, pattern;
```

```

    // Input text and pattern from the user
    cout << "Enter the text: ";
    getline(cin, text);

    cout << "Enter the pattern: ";
    getline(cin, pattern);

    // Perform KMP search
    KMPSearch(text, pattern);

    return 0;
}
```

OUTPUT TERMINAL:

```
Enter the text: ababcabbacbbcbababacbacbabcbababcbabcba
Enter the pattern: bcab
Pattern found at index: 3
```

Q8. Write a program to implement a Suffix tree.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;

// Suffix Tree Node
struct SuffixTreeNode {
    map<char, SuffixTreeNode*> children;  // Map for storing children nodes
    int start;  // Starting index of the substring represented by the node
    int* end;  // Pointer to the end index of the substring
    SuffixTreeNode* suffixLink;  // Link to the node representing the largest suffix

    SuffixTreeNode(int start, int* end) {
        this->start = start;
        this->end = end;
        suffixLink = nullptr;
    }

    int edgeLength() {
        return *end - start + 1;
    }
};

// Global variables for the tree construction
const int MAX_CHAR = 256;
string text;  // The input text for the suffix tree
SuffixTreeNode* root = nullptr;
SuffixTreeNode* lastNewNode = nullptr;
SuffixTreeNode* activeNode = nullptr;

int activeEdge = -1;
int activeLength = 0;
int remainingSuffixCount = 0;
int leafEnd = -1;
int* rootEnd = nullptr;
int* splitEnd = nullptr;
int size = -1; // Length of the input text

// Function to create a new node in the suffix tree
SuffixTreeNode* newNode(int start, int* end) {
    SuffixTreeNode* node = new SuffixTreeNode(start, end);
    return node;
}

```

```cpp
// Function to extend the suffix tree by adding new characters
void extendSuffixTree(int pos) {
    leafEnd = pos;
    remainingSuffixCount++;
    lastNewNode = nullptr;

    while (remainingSuffixCount > 0) {
        if (activeLength == 0) {
            activeEdge = pos;
        }

        if (activeNode->children.find(text[activeEdge]) == activeNode->children.end()) {
            activeNode->children[text[activeEdge]] = newNode(pos, &leafEnd);

            if (lastNewNode != nullptr) {
                lastNewNode->suffixLink = activeNode;
                lastNewNode = nullptr;
            }
        } else {
            SuffixTreeNode* next = activeNode->children[text[activeEdge]];

            if (activeLength >= next->edgeLength()) {
                activeEdge += next->edgeLength();
                activeLength -= next->edgeLength();
                activeNode = next;
                continue;
            }

            if (text[next->start + activeLength] == text[pos]) {
                if (lastNewNode != nullptr && activeNode != root) {
                    lastNewNode->suffixLink = activeNode;
                    lastNewNode = nullptr;
                }

                activeLength++;
                break;
            }

            splitEnd = new int;
            *splitEnd = next->start + activeLength - 1;

            SuffixTreeNode* split = newNode(next->start, splitEnd);
            activeNode->children[text[activeEdge]] = split;

            split->children[text[pos]] = newNode(pos, &leafEnd);
            next->start += activeLength;
            split->children[text[next->start]] = next;

            if (lastNewNode != nullptr) {
                lastNewNode->suffixLink = split;
            }

            lastNewNode = split;
        }

        remainingSuffixCount--;

        if (activeNode == root && activeLength > 0) {
            activeLength--;
            activeEdge = pos - remainingSuffixCount + 1;
        } else if (activeNode != root) {
            activeNode = activeNode->suffixLink;
        }
    }
}
// Function to print the suffix tree
void print(int i, int j) {
    for (int k = i; k <= j; k++) {
        cout << text[k];
    }
}
void setSuffixIndexByDFS(SuffixTreeNode* node, int labelHeight) {
    if (node == nullptr) return;

```

```
    if (node->children.empty()) {
        cout << "Suffix : ";
        print(node->start, *(node->end));
        cout << " [Index = " << size - labelHeight << "]" << endl;
    } else {
        for (auto it : node->children) {
            setSuffixIndexByDFS(it.second, labelHeight + it.second->edgeLength());
        }
    }
}
// Function to build the suffix tree
void buildSuffixTree() {
    size = text.length();
    rootEnd = new int(-1);
    root = newNode(-1, rootEnd);
    activeNode = root;

    for (int i = 0; i < size; i++) {
        extendSuffixTree(i);
    }
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeight);
}
// Main function
int main() {
    cout << "Enter the text: ";
    cin >> text;
    buildSuffixTree();
    return 0;
}
```

OUTPUT TERMINAL:

```
Enter the text: TusharDixit
Suffix : Dixit [Index = 6]
Suffix : TusharDixit [Index = 0]
Suffix : arDixit [Index = 4]
Suffix : harDixit [Index = 3]
Suffix : t [Index = 9]
Suffix : xit [Index = 7]
Suffix : rDixit [Index = 5]
Suffix : sharDixit [Index = 2]
Suffix : t [Index = 10]
Suffix : usharDixit [Index = 1]
Suffix : xit [Index = 8]
```