Anusha Garg
16121
BSc (H) Computer Science
Sem V-A
ADS Practicals

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

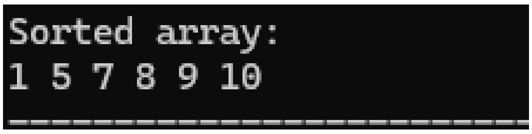
CODE:

```
#include <cstdlib>

#include <time.h>

#include <iostream>

using namespace std;

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

{

        int pivot = arr[high];

        int i = (low - 1);


        for (int j = low; j <= high - 1; j++)

        {

                if (arr[j] <= pivot) {

                        i++;

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

                }

        }

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

        return (i + 1);

}
```

```
int partition_r(int arr[], int low, int high)
{
	srand(time(NULL));
	int random = low + rand() % (high - low);
	swap(arr[random], arr[high]);

	return partition(arr, low, high);
}

void quickSort(int arr[], int low, int high)
{
	if (low < high) {
		int pi = partition_r(arr, low, high);
		quickSort(arr, low, pi - 1);
		quickSort(arr, pi + 1, high);
	}
}
void printArray(int arr[], int size)
{
	int i;
	for (i = 0; i < size; i++)
		cout<<arr[i]<<" ";
}
int main()
{
	int arr[] = { 10, 7, 8, 9, 1, 5 };
	int n = sizeof(arr) / sizeof(arr[0]);

	quickSort(arr, 0, n - 1);
	printf("Sorted array: \n");
	printArray(arr, n);
```

```
        return 0;
}
```

OUTPUT:



2. Write a program to find the i^{th} smallest element of an array using Randomized Select.

CODE:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
```

```
}
int partition_r(int arr[], int low, int high) {
   srand(time(NULL));
   int random = low + rand() % (high - low);
   swap(&arr[random], &arr[high]);
   return partition(arr, low, high);
}
int randomizedSelect(int arr[], int low, int high, int k) {
   if (low == high)
      return arr[low];

   int pi = partition_r(arr, low, high);
   int length = pi - low + 1;
   if (length == k)
      return arr[pi];
   else if (k < length)
      return randomizedSelect(arr, low, pi - 1, k);
   else
      return randomizedSelect(arr, pi + 1, high, k - length);
}
int main() {
   int n, k;
   cout << "Enter the size of the array: ";
   cin >> n;
   int arr[n];
   cout << "Enter the elements of the array: ";
   for (int i = 0; i < n; i++) {
      cin >> arr[i];
   }
   cout << "Enter the value of k: ";
   cin >> k;
```

```
    int result = randomizedSelect(arr, 0, n - 1, k);
    cout << "The " << k << "-th smallest element is: " << result << endl;

    return 0;
}
```

OUTPUT:

```
Enter the size of the array: 4
Enter the elements of the array: 3 1 5 7
Enter the value of k: 2
The 2-th smallest element is: 3
```

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

CODE:

```
#include <bits/stdc++.h>
using namespace std;
class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    int findUPar(int node) {
        if (node == parent[node])
```

```
        return node;
    return parent[node] = findUPar(parent[node]);
}

void unionByRank(int u, int v) {
    int ulp_u = findUPar(u);
    int ulp_v = findUPar(v);
    if (ulp_u == ulp_v) return;
    if (rank[ulp_u] < rank[ulp_v]) {
        parent[ulp_u] = ulp_v;
    }
    else if (rank[ulp_v] < rank[ulp_u]) {
        parent[ulp_v] = ulp_u;
    }
    else {
        parent[ulp_v] = ulp_u;
        rank[ulp_u]++;
    }
}

void unionBySize(int u, int v) {
    int ulp_u = findUPar(u);
    int ulp_v = findUPar(v);
    if (ulp_u == ulp_v) return;
    if (size[ulp_u] < size[ulp_v]) {
        parent[ulp_u] = ulp_v;
        size[ulp_v] += size[ulp_u];
    }
    else {
        parent[ulp_v] = ulp_u;
        size[ulp_u] += size[ulp_v];
```

```
        }
    }
};
class Solution
{
public:
    //Function to find sum of weights of edges of the Minimum Spanning Tree.
    int spanningTree(int V, vector<vector<int>> adj[])
    {
        vector<pair<int, pair<int, int>>> edges;
        for (int i = 0; i < V; i++) {
            for (auto it : adj[i]) {
                int adjNode = it[0];
                int wt = it[1];
                int node = i;

                edges.push_back({wt, {node, adjNode}});
            }
        }
        DisjointSet ds(V);
        sort(edges.begin(), edges.end());
        int mstWt = 0;
        for (auto it : edges) {
            int wt = it.first;
            int u = it.second.first;
            int v = it.second.second;

            if (ds.findUPar(u) != ds.findUPar(v)) {
                mstWt += wt;
                ds.unionBySize(u, v);
            }
```

```
        }

        return mstWt;
    }
};

int main() {

    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2, 3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<vector<int>> adj[V];
    for (auto it : edges) {
        vector<int> tmp(2);
        tmp[0] = it[1];
        tmp[1] = it[2];
        adj[it[0]].push_back(tmp);

        tmp[0] = it[0];
        tmp[1] = it[2];
        adj[it[1]].push_back(tmp);
    }

    Solution obj;
    int mstWt = obj.spanningTree(V, adj);
    cout << "The sum of all the edge weights: " << mstWt << endl;
    return 0;
}
```

OUTPUT:

```
The sum of all the edge weights: 5
```

4. 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.

CODE:

```
#include <bits/stdc++.h>

using namespace std;


class Solution {

public:

        vector<int> bellman_ford(int V, vector<vector<int>>& edges, int S) {

                vector<int> dist(V, 1e8);

                dist[S] = 0;

                for (int i = 0; i < V - 1; i++) {

                        for (auto it : edges) {

                                int u = it[0];

                                int v = it[1];

                                int wt = it[2];

                                if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

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

                                }

                        }

                }

                for (auto it : edges) {

                        int u = it[0];

                        int v = it[1];

                        int wt = it[2];

                        if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

                                return { -1};

                        }

                }
```

```
                return dist;
        }
};
int main() {

        int V = 6;
        vector<vector<int>> edges(7, vector<int>(3));
        edges[0] = {3, 2, 6};
        edges[1] = {5, 3, 1};
        edges[2] = {0, 1, 5};
        edges[3] = {1, 5, -3};
        edges[4] = {1, 2, -2};
        edges[5] = {3, 4, -2};
        edges[6] = {2, 4, 3};

        int S = 0;
        Solution obj;
        vector<int> dist = obj.bellman_ford(V, edges, S);
        cout<<"Distances after applying Bellman Ford:"<<endl;
        for (auto d : dist) {
                cout << d << " ";
        }
        cout << endl;

        return 0;
}
```

OUTPUT:

```
Distances after applying Bellman Ford:
0 5 3 3 1 2
```

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

CODE:

```
// C++ Program to Implement B-Tree
#include <iostream>
using namespace std;

// class for the node present in a B-Tree
template <typename T, int ORDER>
class BTreeNode {
public:
// Array of keys
   T keys[ORDER - 1];
   // Array of child pointers
   BTreeNode* children[ORDER];
    // Current number of keys
   int n;
   // True if leaf node, false otherwise
   bool leaf;

   BTreeNode(bool isLeaf = true) : n(0), leaf(isLeaf) {
      for (int i = 0; i < ORDER; i++)
         children[i] = nullptr;
   }
};

// class for B-Tree
template <typename T, int ORDER>
class BTree {
private:
   BTreeNode<T, ORDER>* root; // Pointer to root node
```

```
// Function to split a full child node
void splitChild(BTreeNode<T, ORDER>* x, int i) {
    BTreeNode<T, ORDER>* y = x->children[i];
    BTreeNode<T, ORDER>* z = new BTreeNode<T, ORDER>(y->leaf);
    z->n = ORDER / 2 - 1;

    for (int j = 0; j < ORDER / 2 - 1; j++)
        z->keys[j] = y->keys[j + ORDER / 2];

    if (!y->leaf) {
        for (int j = 0; j < ORDER / 2; j++)
            z->children[j] = y->children[j + ORDER / 2];
    }

    y->n = ORDER / 2 - 1;

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

    x->children[i + 1] = z;

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

    x->keys[i] = y->keys[ORDER / 2 - 1];
    x->n = x->n + 1;
}

// Function to insert a key in a non-full node
void insertNonFull(BTreeNode<T, ORDER>* x, T k) {
    int i = x->n - 1;
```

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

        x->keys[i + 1] = k;
        x->n = x->n + 1;
    } else {
        while (i >= 0 && k < x->keys[i])
            i--;

        i++;
        if (x->children[i]->n == ORDER - 1) {
            splitChild(x, i);

            if (k > x->keys[i])
                i++;
        }
        insertNonFull(x->children[i], k);
    }
}

// Function to traverse the tree
void traverse(BTreeNode<T, ORDER>* x) {
    int i;
    for (i = 0; i < x->n; i++) {
        if (!x->leaf)
            traverse(x->children[i]);
        cout << " " << x->keys[i];
```

```
    }

    if (!x->leaf)
        traverse(x->children[i]);
}

// Function to search a key in the tree
BTreeNode<T, ORDER>* search(BTreeNode<T, ORDER>* x, T k) {
    int i = 0;
    while (i < x->n && k > x->keys[i])
        i++;

    if (i < x->n && k == x->keys[i])
        return x;

    if (x->leaf)
        return nullptr;

    return search(x->children[i], k);
}

// Function to find the predecessor
T getPredecessor(BTreeNode<T, ORDER>* node, int idx) {
    BTreeNode<T, ORDER>* current = node->children[idx];
    while (!current->leaf)
        current = current->children[current->n];
    return current->keys[current->n - 1];
}

// Function to find the successor
T getSuccessor(BTreeNode<T, ORDER>* node, int idx) {
```

```
    BTreeNode<T, ORDER>* current = node->children[idx + 1];
    while (!current->leaf)
        current = current->children[0];
    return current->keys[0];
}

// Function to fill child node
void fill(BTreeNode<T, ORDER>* node, int idx) {
    if (idx != 0 && node->children[idx - 1]->n >= ORDER / 2)
        borrowFromPrev(node, idx);
    else if (idx != node->n && node->children[idx + 1]->n >= ORDER / 2)
        borrowFromNext(node, idx);
    else {
        if (idx != node->n)
            merge(node, idx);
        else
            merge(node, idx - 1);
    }
}

// Function to borrow from previous sibling
void borrowFromPrev(BTreeNode<T, ORDER>* node, int idx) {
    BTreeNode<T, ORDER>* child = node->children[idx];
    BTreeNode<T, ORDER>* sibling = node->children[idx - 1];

    for (int i = child->n - 1; i >= 0; --i)
        child->keys[i + 1] = child->keys[i];

    if (!child->leaf) {
        for (int i = child->n; i >= 0; --i)
            child->children[i + 1] = child->children[i];
```

```
    }

    child->keys[0] = node->keys[idx - 1];

    if (!child->leaf)
        child->children[0] = sibling->children[sibling->n];

    node->keys[idx - 1] = sibling->keys[sibling->n - 1];

    child->n += 1;
    sibling->n -= 1;
}

// Function to borrow from next sibling
void borrowFromNext(BTreeNode<T, ORDER>* node, int idx) {
    BTreeNode<T, ORDER>* child = node->children[idx];
    BTreeNode<T, ORDER>* sibling = node->children[idx + 1];

    child->keys[child->n] = node->keys[idx];

    if (!child->leaf)
        child->children[child->n + 1] = sibling->children[0];

    node->keys[idx] = sibling->keys[0];

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

    if (!sibling->leaf) {
        for (int i = 1; i <= sibling->n; ++i)
            sibling->children[i - 1] = sibling->children[i];
```

```
    }

    child->n += 1;
    sibling->n -= 1;
}

// Function to merge two nodes
void merge(BTreeNode<T, ORDER>* node, int idx) {
    BTreeNode<T, ORDER>* child = node->children[idx];
    BTreeNode<T, ORDER>* sibling = node->children[idx + 1];

    child->keys[ORDER / 2 - 1] = node->keys[idx];

    for (int i = 0; i < sibling->n; ++i)
        child->keys[i + ORDER / 2] = sibling->keys[i];

    if (!child->leaf) {
        for (int i = 0; i <= sibling->n; ++i)
            child->children[i + ORDER / 2] = sibling->children[i];
    }

    for (int i = idx + 1; i < node->n; ++i)
        node->keys[i - 1] = node->keys[i];

    for (int i = idx + 2; i <= node->n; ++i)
        node->children[i - 1] = node->children[i];

    child->n += sibling->n + 1;
    node->n--;

    delete sibling;
```

```
}

// Function to remove a key from a non-leaf node
void removeFromNonLeaf(BTreeNode<T, ORDER>* node, int idx) {
    T k = node->keys[idx];

    if (node->children[idx]->n >= ORDER / 2) {
        T pred = getPredecessor(node, idx);
        node->keys[idx] = pred;
        remove(node->children[idx], pred);
    } else if (node->children[idx + 1]->n >= ORDER / 2) {
        T succ = getSuccessor(node, idx);
        node->keys[idx] = succ;
        remove(node->children[idx + 1], succ);
    } else {
        merge(node, idx);
        remove(node->children[idx], k);
    }
}

// Function to remove a key from a leaf node
void removeFromLeaf(BTreeNode<T, ORDER>* node, int idx) {
    for (int i = idx + 1; i < node->n; ++i)
        node->keys[i - 1] = node->keys[i];

    node->n--;
}

// Function to remove a key from the tree
void remove(BTreeNode<T, ORDER>* node, T k) {
    int idx = 0;
```

```
        while (idx < node->n && node->keys[idx] < k)
            ++idx;

        if (idx < node->n && node->keys[idx] == k) {
            if (node->leaf)
                removeFromLeaf(node, idx);
            else
                removeFromNonLeaf(node, idx);
        } else {
            if (node->leaf) {
                cout << "The key " << k << " is not present in the tree\n";
                return;
            }

            bool flag = ((idx == node->n) ? true : false);

            if (node->children[idx]->n < ORDER / 2)
                fill(node, idx);

            if (flag && idx > node->n)
                remove(node->children[idx - 1], k);
            else
                remove(node->children[idx], k);
        }
    }

public:
    BTree() { root = new BTreeNode<T, ORDER>(true); }

    // Function to insert a key in the tree
    void insert(T k) {
```

```
    if (root->n == ORDER - 1) {
        BTreeNode<T, ORDER>* s = new BTreeNode<T, ORDER>(false);
        s->children[0] = root;
        root = s;
        splitChild(s, 0);
        insertNonFull(s, k);
    } else
        insertNonFull(root, k);
}

// Function to traverse the tree
void traverse() {
    if (root != nullptr)
        traverse(root);
}

// Function to search a key in the tree
BTreeNode<T, ORDER>* search(T k) {
    return (root == nullptr) ? nullptr : search(root, k);
}

// Function to remove a key from the tree
void remove(T k) {
    if (!root) {
        cout << "The tree is empty\n";
        return;
    }

    remove(root, k);

    if (root->n == 0) {
```

```
            BTreeNode<T, ORDER>* tmp = root;
            if (root->leaf)
                root = nullptr;
            else
                root = root->children[0];

            delete tmp;
        }
    }
};

int main() {
    BTree<int, 3> t;

    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
    t.insert(7);
    t.insert(17);

    cout << "Traversal of the constructed tree is: ";
    t.traverse();
    cout << endl;

    int k = 6;
    (t.search(k) != nullptr) ? cout << k << " is found" << endl
                        : cout << k << " is not found" << endl;
```

```
    k = 15;
    (t.search(k) != nullptr) ? cout << k << " is found" << endl
                             : cout << k << " is not found" << endl;

    t.remove(6);
    cout << "Traversal of the tree after removing 6: ";
    t.traverse();
    cout << endl;

    t.remove(13);
    cout << "Traversal of the tree after removing 13: ";
    t.traverse();
    cout << endl;
    return 0;
}
```

OUTPUT:

```
Traversal of the constructed tree is:  5 7 17
6 is not found
15 is not found
The key 6 is not present in the tree
Traversal of the tree after removing 6:  5 7 17
The key 13 is not present in the tree
Traversal of the tree after removing 13:  5 7 17

--------------------------------
Process exited after 13.4 seconds with return value 0
Press any key to continue . . .
```

6. Write a program to implement the Tree Data structure, which supports the following operations:

a. Insert

b. Search

CODE:

```
#include <iostream>
```

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

// Define the TreeNode structure
class TreeNode {
public:
    int value;                  // Value of the node
    vector<TreeNode*> children;   // Children of the node

    // Constructor
    TreeNode(int val) : value(val) {}
};

// Define the Tree structure
class Tree {
private:
    TreeNode* root;

public:
    // Constructor
    Tree() : root(nullptr) {}

    // Insert function to add a node to the tree
    void insert(int parentValue, int newValue) {
        if (!root) {
            root = new TreeNode(newValue);
            cout << "Root node inserted with value: " << newValue << endl;
            return;
        }

        TreeNode* parent = search(root, parentValue);
```

```
        if (parent) {
            TreeNode* newNode = new TreeNode(newValue);
            parent->children.push_back(newNode);
            cout << "Node inserted with value: " << newValue << " under parent: " << parentValue <<
endl;
        } else {
            cout << "Parent with value " << parentValue << " not found!" << endl;
        }
    }

    // Search function to find a node by its value
    bool search(int value) {
        return search(root, value) != nullptr;
    }

private:
    // Helper function to search for a node in the tree
    TreeNode* search(TreeNode* node, int value) {
        if (!node) return nullptr;

        if (node->value == value) return node;

        for (TreeNode* child : node->children) {
            TreeNode* result = search(child, value);
            if (result) return result;
        }

        return nullptr;
    }

public:
```

```
    // Display the tree using pre-order traversal
    void display(TreeNode* node = nullptr, int level = 0) {
        if (!node) {
            if (!root) {
                cout << "Tree is empty!" << endl;
                return;
            }
            node = root;
        }

        // Print the current node
        for (int i = 0; i < level; i++) cout << "--";
        cout << node->value << endl;

        // Print the children recursively
        for (TreeNode* child : node->children) {
            display(child, level + 1);
        }
    }
};

int main() {
    Tree tree;

    // Insert operations
    tree.insert(0, 10);  // Inserting root node
    tree.insert(10, 20);
    tree.insert(10, 30);
    tree.insert(20, 40);
    tree.insert(30, 50);
```

```
    // Display the tree
    cout << "Tree structure:" << endl;
    tree.display();

    // Search operations
    int searchValue = 40;
    if (tree.search(searchValue)) {
        cout << "Value " << searchValue << " found in the tree." << endl;
    } else {
        cout << "Value " << searchValue << " not found in the tree." << endl;
    }

    return 0;
}
```

OUTPUT:

```
Root node inserted with value: 10
Node inserted with value: 20 under parent: 10
Node inserted with value: 30 under parent: 10
Node inserted with value: 40 under parent: 20
Node inserted with value: 50 under parent: 30
Tree structure:
10
--20
----40
--30
----50
Value 40 found in the tree.

--------------------------------
Process exited after 12.92 seconds with return value 0
Press any key to continue . . .
```

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

CODE:

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

vector<int> computeFailFunction(const string& pattern){
	vector<int> fail(pattern.size());
	fail[0]=0;
	int m=pattern.size();
	int j=0;
	int i=1;
	while(i<m){
		if(pattern[j]==pattern[i]){
			fail[i]=j+1;
			i++;
			j++;
		}
		else if(j>0){
			j=fail[j-1];
		}
		else{
			fail[i]=0;
			i++;
		}
	}
	return fail;
}
int KMPmatch(const string& text,const string& pattern){
	int n=text.size();
	int m=pattern.size();
	vector<int> fail=computeFailFunction(pattern);
```

```
	int i=0;
	int j=0;
	while(i<n){
		if(pattern[j]==text[i]){
			if(j==m-1)
				return i-m+1;
			i++;
			j++;
		}
		else if (j>0) j=fail[j-1];
		else i++;
	}
	return -1;
}

int main(){
	int ans;
	string st;
	string sub;
	cout<<"Enter the String: ";
	cin>>st;
	cout<<"Enter the Substring: ";
	cin>>sub;
	ans=KMPmatch(st,sub);
	cout<<"The Substring is found at index: "<<ans;
}
```

OUTPUT:

```
Enter the String: ababcabcd
Enter the Substring: abcd
The Substring is found at index: 5
--------------------------------
Process exited after 108.5 seconds with return value 0
Press any key to continue . . .
```

8. Write a program to implement a Suffix tree.

CODE:

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

class SuffixTreeNode {
public:
    unordered_map<char, SuffixTreeNode*> children;
    int start;
    int* end;
    int suffixLink;

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

class SuffixTree {
private:
    string text;
    SuffixTreeNode* root;
    int* leafEnd;
    SuffixTreeNode* activeNode;
    int activeEdge;
    int activeLength;
```

```
int remainingSuffixCount;
int size;

SuffixTreeNode* createNode(int start, int* end) {
    return new SuffixTreeNode(start, end);
}

void extendSuffixTree(int pos) {
    leafEnd = new int(pos);
    remainingSuffixCount++;
    SuffixTreeNode* lastCreatedInternalNode = nullptr;

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

        if (activeNode->children.find(text[activeEdge]) == activeNode->children.end()) {
            activeNode->children[text[activeEdge]] = createNode(pos, leafEnd);
            if (lastCreatedInternalNode) {
                lastCreatedInternalNode->suffixLink = activeNode;
            }
            lastCreatedInternalNode = nullptr;
        } else {
            SuffixTreeNode* next = activeNode->children[text[activeEdge]];
            if (walkDown(next)) {
                continue;
            }
            if (text[next->start + activeLength] == text[pos]) {
                if (lastCreatedInternalNode && activeNode != root) {
                    lastCreatedInternalNode->suffixLink = activeNode;
                }
```

```
            activeLength++;
            break;
        }

        int* splitEnd = new int(next->start + activeLength - 1);
        SuffixTreeNode* split = createNode(next->start, splitEnd);
        activeNode->children[text[activeEdge]] = split;

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

        if (lastCreatedInternalNode) {
            lastCreatedInternalNode->suffixLink = split;
        }

        lastCreatedInternalNode = split;
    }

    remainingSuffixCount--;
    if (activeNode == root) {
        if (activeLength > 0) {
            activeEdge = pos - remainingSuffixCount + 1;
            activeLength--;
        }
    } else {
        activeNode = activeNode->suffixLink ? activeNode->suffixLink : root;
    }
  }
}
```

```
    bool walkDown(SuffixTreeNode* next) {
        int edgeLength = *next->end - next->start + 1;
        if (activeLength >= edgeLength) {
            activeEdge += edgeLength;
            activeLength -= edgeLength;
            activeNode = next;
            return true;
        }
        return false;
    }

    void display(SuffixTreeNode* node, int level) {
        if (node) {
            for (auto& child : node->children) {
                for (int i = 0; i < level; i++) cout << " ";
                cout << text.substr(child.second->start, *child.second->end - child.second->start + 1) <<
endl;
                display(child.second, level + 1);
            }
        }
    }

public:
    SuffixTree(string s) : text(s) {
        size = text.size();
        root = createNode(-1, new int(-1));
        activeNode = root;
        activeEdge = -1;
        activeLength = 0;
        remainingSuffixCount = 0;
```

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

    void display() {
        display(root, 0);
    }
};

int main() {
    string str = "banana";
    SuffixTree st(str);
    st.display();
    return 0;
}
```

OUTPUT:

banana

anana

nana

ana

na

a