```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

// Function to swap two elements
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Function to partition the array around a pivot
int partition(int arr[], int low, int high, int &comparisons) {
    int pivot = arr[high]; // Pivot element
    int i = low - 1;

    for (int j = low; j < high; j++) {
        comparisons++; // Increment comparison count
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Function to choose a random pivot and partition the array
int randomizedPartition(int arr[], int low, int high, int &comparisons) {
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[randomIndex], arr[high]);
    return partition(arr, low, high, comparisons);
}

// Function implementing Randomized Quick Sort
void randomizedQuickSort(int arr[], int low, int high, int &comparisons) {
    if (low < high) {
        int pivotIndex = randomizedPartition(arr, low, high, comparisons);
        randomizedQuickSort(arr, low, pivotIndex - 1, comparisons);
        randomizedQuickSort(arr, pivotIndex + 1, high, comparisons);
    }
}

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

    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;

    int arr[n];
    cout << "Enter the elements of the array:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int comparisons = 0; // Variable to track the number of comparisons
    randomizedQuickSort(arr, 0, n - 1, comparisons);

    cout << "Sorted array:" << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

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

    return 0;
}
```

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

// Function to swap two elements
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

// Partition the array around a pivot
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return i + 1;
}

// Randomized partition: Chooses a random pivot and partitions the array
int randomizedPartition(int arr[], int low, int high) {
    int randomIndex = low + rand() % (high - low + 1);
    swap(arr[randomIndex], arr[high]);
    return partition(arr, low, high);
}

// Randomized Select algorithm
int randomizedSelect(int arr[], int low, int high, int i) {
    if (low == high) {
        return arr[low];
    }

    int pivotIndex = randomizedPartition(arr, low, high);
    int k = pivotIndex - low + 1; // Number of elements in the left partition

    if (i == k) {
        return arr[pivotIndex]; // The pivot is the i-th smallest element
    } else if (i < k) {
        return randomizedSelect(arr, low, pivotIndex - 1, i); // Look in the left
partition
    } else {
        return randomizedSelect(arr, pivotIndex + 1, high, i - k); // Look in the
right partition
    }
}

// Main function
int main() {
    srand(static_cast<unsigned>(time(0))); // Seed the random number generator

    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;

    int arr[n];
    cout << "Enter the elements of the array:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int i;
    cout << "Enter the value of i (1-based index) to find the i-th smallest
element: ";
    cin >> i;

    if (i < 1 || i > n) {
        cout << "Invalid value of i. Please enter a value between 1 and " << n <<
"." << endl;
        return 1;
    }

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

    return 0;
}
```

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

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Structure to represent a graph
struct Graph {
    int V, E;
    vector<Edge> edges;
};

// Structure to represent a subset for Union-Find
struct Subset {
    int parent;
    int rank;
};

// Function to find the subset of an element `u` using path compression
int find(Subset subsets[], int u) {
    if (subsets[u].parent != u) {
        subsets[u].parent = find(subsets, subsets[u].parent);
    }
    return subsets[u].parent;
}

// Function to perform the union of two subsets using rank
void Union(Subset subsets[], int u, int v) {
    int rootU = find(subsets, u);
    int rootV = find(subsets, v);

    if (subsets[rootU].rank < subsets[rootV].rank) {
        subsets[rootU].parent = rootV;
    } else if (subsets[rootU].rank > subsets[rootV].rank) {
        subsets[rootV].parent = rootU;
    } else {
        subsets[rootV].parent = rootU;
        subsets[rootU].rank++;
    }
}

// Comparator function to sort edges by weight
bool compareEdges(Edge a, Edge b) {
    return a.weight < b.weight;
}

// Function to compute the MST using Kruskal's Algorithm
void KruskalMST(Graph &graph) {
    int V = graph.V;
    vector<Edge> result; // Store the edges of the MST
    int e = 0;           // Counter for edges in the MST
    int i = 0;           // Counter for sorted edges

    // Step 1: Sort all edges by weight
    sort(graph.edges.begin(), graph.edges.end(), compareEdges);
```

```cpp
    // Allocate memory for subsets
    Subset *subsets = new Subset[V];
    for (int v = 0; v < V; ++v) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Step 2: Pick the smallest edge and add it to the MST
    while (e < V - 1 && i < graph.E) {
        Edge nextEdge = graph.edges[i++];
        int x = find(subsets, nextEdge.src);
        int y = find(subsets, nextEdge.dest);

        // If including this edge doesn't form a cycle
        if (x != y) {
            result.push_back(nextEdge);
            Union(subsets, x, y);
            e++;
        }
    }

    // Print the MST
    cout << "Edges in the Minimum Spanning Tree:" << endl;
    int totalWeight = 0;
    for (auto edge : result) {
        cout << edge.src << " -- " << edge.dest << " == " << edge.weight << endl;
        totalWeight += edge.weight;
    }
    cout << "Total weight of the Minimum Spanning Tree: " << totalWeight << endl;

    delete[] subsets;
}

// Main function
int main() {
    int V, E;
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;

    Graph graph;
    graph.V = V;
    graph.E = E;

    cout << "Enter the edges (source, destination, weight):" << endl;
    for (int i = 0; i < E; i++) {
        Edge edge;
        cin >> edge.src >> edge.dest >> edge.weight;
        graph.edges.push_back(edge);
    }

    KruskalMST(graph);

    return 0;
}
```

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

// Structure to represent an edge
struct Edge {
    int src, dest, weight;
};

// Function to implement the Bellman-Ford algorithm
void bellmanFord(int V, int E, vector<Edge> &edges, int src) {
    // Initialize distances from the source to all vertices as infinite
    vector<int> distance(V, INT_MAX);
    distance[src] = 0;

    // Step 1: 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;

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

    // Step 2: Check for negative weight cycles
    for (int j = 0; j < E; j++) {
        int u = edges[j].src;
        int v = edges[j].dest;
        int weight = edges[j].weight;

        if (distance[u] != INT_MAX && distance[u] + weight < distance[v]) {
            cout << "Graph contains a negative weight cycle" << endl;
            return;
        }
    }

    // Print the distances from the source
    cout << "Vertex\tDistance from Source" << endl;
    for (int i = 0; i < V; i++) {
        cout << i << "\t" << (distance[i] == INT_MAX ? "INF" :
to_string(distance[i])) << endl;
    }
}

// Main function
int main() {
    int V, E;
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;

    vector<Edge> edges(E);

    cout << "Enter the edges (source, destination, weight):" << endl;
    for (int i = 0; i < E; i++) {
        cin >> edges[i].src >> edges[i].dest >> edges[i].weight;
    }

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

    bellmanFord(V, E, edges, src);

    return 0;
}
```

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

// A B-Tree Node
class BTreeNode {
    int *keys;           // Array of keys
    int t;               // Minimum degree (defines the range for number of keys)
    BTreeNode **children; // Array of child pointers
    int n;               // Current number of keys
    bool leaf;           // True if leaf node

public:
    BTreeNode(int _t, bool _leaf);

    // Traverse all nodes in a subtree rooted with this node
    void traverse();

    // Search for a key in the subtree rooted with this node
    BTreeNode *search(int k);

    // Insert a key in a non-full node
    void insertNonFull(int k);

    // Split the child `i` of this node
    void splitChild(int i, BTreeNode *y);

    friend class BTree;
};

// A B-Tree
class BTree {
    BTreeNode *root; // Pointer to the root node
    int t;           // Minimum degree

public:
    BTree(int _t) {
        root = nullptr;
        t = _t;
    }

    // Traverse the B-Tree
    void traverse() {
        if (root != nullptr)
            root->traverse();
    }

    // Search for a key in the B-Tree
    BTreeNode *search(int k) {
        return (root == nullptr) ? nullptr : root->search(k);
    }

    // Insert a new key in the B-Tree
    void insert(int k);
};

// Constructor for a BTreeNode
BTreeNode::BTreeNode(int t1, bool leaf1) {
    t = t1;
    leaf = leaf1;
    keys = new int[2 * t - 1];
    children = new BTreeNode *[2 * t];

    n = 0;
}

// Function to traverse the B-Tree
void BTreeNode::traverse() {
    int i;
    for (i = 0; i < n; i++) {
        if (!leaf)
            children[i]->traverse();
        cout << keys[i] << " ";
    }

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

// Function to search for a key in the subtree rooted with this node
BTreeNode *BTreeNode::search(int k) {
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    if (keys[i] == k)
        return this;

    if (leaf)
        return nullptr;

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

// Function to insert a new key in the B-Tree
void BTree::insert(int k) {
    if (root == nullptr) {
        root = new BTreeNode(t, true);
        root->keys[0] = k;
        root->n = 1;
    } else {
        if (root->n == 2 * t - 1) {
            BTreeNode *s = new BTreeNode(t, false);

            s->children[0] = root;

            s->splitChild(0, root);

            int i = 0;
            if (s->keys[0] < k)
                i++;
            s->children[i]->insertNonFull(k);

            root = s;
        } else
            root->insertNonFull(k);
    }
}
```

```
// Function to insert a key in a non-full node
void BTreeNode::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++;
    } else {
        while (i >= 0 && keys[i] > k)
            i--;

        if (children[i + 1]->n == 2 * t - 1) {
            splitChild(i + 1, children[i + 1]);

            if (keys[i + 1] < k)
                i++;
        }
        children[i + 1]->insertNonFull(k);
    }
}

// Function to split a child
void BTreeNode::splitChild(int i, BTreeNode *y) {
    BTreeNode *z = new BTreeNode(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->children[j] = y->children[j + t];
    }

    y->n = t - 1;

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

    children[i + 1] = z;

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

    keys[i] = y->keys[t - 1];
    n++;
}

// Main function
int main() {
    int t;
    cout << "Enter the minimum degree of the B-Tree: ";
    cin >> t;
    BTree btree(t);

    int choice, key;
    while (true) {
        cout << "\n1. Insert key\n2. Traverse B-Tree\n3. Search key\n4. 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 << "Enter the key to search: ";
            cin >> key;
            if (btree.search(key) != nullptr)
                cout << "Key " << key << " is found in the B-Tree.\n";
            else
                cout << "Key " << key << " is not found in the B-Tree.\n";
            break;
        case 4:
            return 0;
        default:
            cout << "Invalid choice! Please try again.\n";
        }
    }
}
```

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

// Trie Node structure
class TrieNode {
public:
    TrieNode *children[26]; // Array of pointers to children nodes (26 letters for
lowercase English)
    bool isEndOfWord;       // True if the node represents the end of a word

    TrieNode() {
        isEndOfWord = false;
        for (int i = 0; i < 26; i++)
            children[i] = nullptr;
    }
};

// Trie class
class Trie {
private:
    TrieNode *root;

public:
    Trie() {
        root = new TrieNode();
    }

    // Function to insert a word into the Trie
    void insert(string word) {
        TrieNode *node = root;

        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                node->children[index] = new TrieNode();
            }
            node = node->children[index];
        }

        node->isEndOfWord = true;
        cout << "Inserted \"" << word << "\" into the Trie.\n";
    }

    // Function to search for a word in the Trie
    bool search(string word) {
        TrieNode *node = root;

        for (char c : word) {
            int index = c - 'a';
            if (!node->children[index]) {
                return false;
            }
            node = node->children[index];
        }

        return node->isEndOfWord;
    }
};
```

```
// Main function
int main() {
    Trie trie;

    int choice;
    string word;

    while (true) {
        cout << "\n1. Insert word\n2. Search word\n3. Exit\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter the word to insert: ";
            cin >> word;
            trie.insert(word);
            break;

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

        case 3:
            return 0;

        default:
            cout << "Invalid choice! Please try again.\n";
        }
    }
}
```

```
//  Knuth-Morris-Pratt (KMP) Algorithm

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

// Function to compute the LPS (Longest Prefix Suffix) array
void computeLPSArray(const string &pattern, vector<int> &lps) {
    int m = pattern.size();
    int length = 0; // Length of the previous longest prefix suffix
    lps[0] = 0;     // lps[0] is always 0

    int i = 1;
    while (i < m) {
        if (pattern[i] == pattern[length]) {
            length++;
            lps[i] = length;
            i++;
        } else {
            if (length != 0) {
                length = lps[length - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
}

// KMP Algorithm to search for a pattern in a given text
void KMPSearch(const string &text, const string &pattern) {
    int n = text.size();
    int m = pattern.size();

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

    int i = 0; // Index for text
    int j = 0; // Index for pattern
    bool found = false;

    while (i < n) {
        if (pattern[j] == text[i]) {
            i++;
            j++;
        }

        if (j == m) {
            cout << "Pattern found at index " << i - j << endl;
            found = true;
            j = lps[j - 1];
        } else if (i < n && pattern[j] != text[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }
    }

    if (!found)
        cout << "Pattern not found in the text.\n";
}

// Main function
int main() {
    string text, pattern;

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

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

    KMPSearch(text, pattern);

    return 0;
}
```

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

// Structure for a Suffix Tree Node
struct SuffixTreeNode {
    unordered_map<char, SuffixTreeNode*> children; // Map of child nodes
    int start;                                   // Start index of the substring
    int *end;                                    // End index of the substring
    SuffixTreeNode* suffixLink;                  // Suffix link for Ukkonen's
algorithm

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

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

// Suffix Tree class
class SuffixTree {
private:
    string text;                    // The input text
    SuffixTreeNode *root;           // Root node of the suffix tree
    SuffixTreeNode *activeNode;     // Active node during construction
    int activeEdge;                 // Active edge during construction
    int activeLength;               // Active length during construction
    int remainingSuffixCount;       // Remaining suffixes to insert
    int *leafEnd;                   // End of the current leaf
    int size;                       // Size of the input text

    // Function to extend the suffix tree by one character
    void extendSuffixTree(int pos);

    // Helper function to traverse the tree for pattern search
    bool traverseTree(SuffixTreeNode *node, const string &pattern, int &index);

    // Function to delete the suffix tree and free memory
    void deleteTree(SuffixTreeNode *node);

public:
    SuffixTree(const string &text);
    ~SuffixTree();

    // Function to search for a pattern in the text
    bool search(const string &pattern);
};

// Constructor
SuffixTree::SuffixTree(const string &text) {
    this->text = text;
    size = text.size();
    root = new SuffixTreeNode(-1, new int(-1));
    activeNode = root;
    activeEdge = -1;
    activeLength = 0;
    remainingSuffixCount = 0;
    leafEnd = new int(-1);

    // Build the suffix tree
    for (int i = 0; i < size; i++) {
        extendSuffixTree(i);
    }
}

// Destructor
SuffixTree::~SuffixTree() {
    deleteTree(root);
    delete leafEnd;
}

// Function to extend the suffix tree by one character
void SuffixTree::extendSuffixTree(int pos) {
    (*leafEnd) = pos;
    remainingSuffixCount++;

    SuffixTreeNode *lastNewNode = nullptr;

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

        char currentChar = text[activeEdge];
        if (activeNode->children.find(currentChar) == activeNode->children.end()) {
            activeNode->children[currentChar] = new SuffixTreeNode(pos, leafEnd);

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

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

            split->children[text[pos]] = new SuffixTreeNode(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 delete the suffix tree and free memory
void SuffixTree::deleteTree(SuffixTreeNode *node) {
    for (auto &child : node->children) {
        deleteTree(child.second);
    }
    delete node->end;
    delete node;
}

// Function to search for a pattern in the text
bool SuffixTree::search(const string &pattern) {
    int index = 0;
    return traverseTree(root, pattern, index);
}

// Helper function to traverse the tree for pattern search
bool SuffixTree::traverseTree(SuffixTreeNode *node, const string &pattern, int
&index) {
    if (index == pattern.size()) return true;

    if (node->children.find(pattern[index]) == node->children.end()) return false;

    SuffixTreeNode *next = node->children[pattern[index]];
    int edgeLength = next->edgeLength();

    for (int i = 0; i < edgeLength && index < pattern.size(); i++) {
        if (text[next->start + i] != pattern[index]) return false;
        index++;
    }

    return traverseTree(next, pattern, index);
}

// Main function
int main() {
    string text, pattern;

    cout << "Enter the text: ";
    cin >> text;
    SuffixTree suffixTree(text);

    cout << "Enter the pattern to search: ";
    cin >> pattern;

    if (suffixTree.search(pattern)) {
        cout << "Pattern found in the text.\n";
    } else {
        cout << "Pattern not found in the text.\n";
    }

    return 0;
}
```