1.Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client's telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers

```
def linear_probing():
    index=key% size
    if(l[index]==-1):
        l[index]=key
        return l
    else:
        for j in range(size):
            p=(index+j)% size
            if(l[p])==-1:
                l[p]=key
                return l
        print("Table is full")
        return l
def Quad_probing():
    index=key% size
    if(l[index]==-1):
        l[index]=key
        return l
    else:
        for j in range(1,size):
            p=(index*j*j)% size
            if(l[p]==-1):
                l[p]=key
                return l
        print("Table is full")
        return l

def search():
    count=1
    for i in range(size):
        index=(key+i)% size
        if(l[index]==key):
            print("Telephone number is found at index:",index)
            print("Record is:",l[index])
            print("No. of comparsions:",count)
            break
        elif l[index]==-1:
            print("Telephone number not found")
            break
        else:
            count+=1
    else:
        print("Telephone number not found")


while(True):
    ch=int(input("Enter choice :\n1.Linear probing \n2.Quadratic Probing \n3.search \n4Exit\n"))
    if ch==1:
        l=[]
        size=int(input("Enter table size:"))
        for i in range(size):
```

```
        l.append(-1)

    print(l)
    print("using Linear probing to strore Telephone numbers:")
    i=0
    while i<size:
        key=int(input("Enter Telephone number:"))
        temp=key
        cnt=0
        while temp>0:
            temp//=10
            cnt+=1
        if cnt>10 or cnt<10:
            print("Telephone number is invalid")
        else:
            print(linear_probing())
            i+=1
elif ch==2:
    l=[]
    size2=int(input("Enter table size:"))
    for i in range(size2):
        l.append(-1)

    print(l)

    print("using Quadratic probing to strore Telephone numbers:")
    i=0
    while i<size2:
        key=int(input("Enter Telephone number:"))
        temp=key
        cnt=0
        while temp>0:
            temp//=10
            cnt+=1
        if cnt>10 or cnt<10:
            print("Telephone number is invalid")
        else:
            print(Quad_probing())
            i+=1
elif ch ==3:
    key=int(input("enter Telephone number top search:"))
    temp=key
    cnt=0
    while temp>0:
        temp//=10
        cnt+=1
    if cnt>10 or cnt<10:
        print("Telephone number is invalid")
    else:
        search()
elif ch==4:
    print("Exiting program")
    break
else:
```

```
        print("invalid choice").
```

2. Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement.

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique. Standard Operations: Insert(key, value), Find(key), Delete(key)

```
def hash_function(key, size):
    return hash(key) % size


def insert(table, size, key, value):
    index = hash_function(key, size)
    for pair in table[index]:
        if pair[0] == key:
            pair[1] = value
            return
    table[index].append([key, value])


def find(table, size, key):
    index = hash_function(key, size)
    for pair in table[index]:
        if pair[0] == key:
            return pair[1]
    return None


def delete(table, size, key):
    index = hash_function(key, size)
    for i, pair in enumerate(table[index]):
        if pair[0] == key:
            del table[index][i]
            return True
    return False


def display(table):
    for i, valuess in enumerate(table):
        print(f"Index {i}: {valuess}")


size = int(input("Enter size of hash table: "))
hash_table = [[] for _ in range(size)]

while True:
    print("\nDictionary Operations:")
    print("1. Insert (key, value)")
    print("2. Find (key)")
    print("3. Delete (key)")
    print("4. Display Hash Table")
    print("5. Exit")

    choice = input("Enter your choice (1-5): ")
```

```
    if choice == "1":
        k = input("Enter key: ")
        v = input("Enter value: ")
        insert(hash_table, size, k, v)
    elif choice == "2":
        k = input("Enter key to find: ")
        result = find(hash_table, size, k)
        if result is not None:
            print(f"Value: {result}")
        else:
            print("Key not found")
    elif choice == "3":
        k = input("Enter key to delete: ")
        deleted = delete(hash_table, size, k)
        if deleted:
            print("Deleted")
        else:
            print("Key not found")
    elif choice == "4":
        display(hash_table)
    elif choice == "5":
        print("Exiting...")
        break
    else:
        print("Invalid choice! Please enter a number between 1 and 5.")
```

3.Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree - i. Insert new node ii.Find number of nodes in longest path iii. Minimum data value found in the tree iv.
Change a tree so that the roles of the left and right pointers are swapped at every node v. Search a value

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

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int key) {
        data = key;
        left = nullptr;
        right = nullptr;
    }
};

class BinarySearchTree {
public:
    Node* insert(Node* root, int key) {
        if (root == nullptr) {
            return new Node(key);
        }

        if (key < root->data) {
```

```
        root->left = insert(root->left, key);
    } else {
        root->right = insert(root->right, key);
    }

    return root;
}

Node* insert_new_node(BinarySearchTree* bst, Node* root, int key) {
    return bst->insert(root, key);
}

int longest_path(Node* root) {
    if (root == nullptr) {
        return 0;
    } else {
        int left_height = longest_path(root->left);
        int right_height = longest_path(root->right);
        return max(left_height, right_height) + 1;
    }
}

int min_value(Node* root) {
    Node* current = root;
    while (current->left != nullptr) {
        current = current->left;
    }
    return current->data;
}

void swap_children(Node* root) {
    if (root == nullptr) {
        return;
    }
    swap(root->left, root->right);
    swap_children(root->left);
    swap_children(root->right);
}

Node* search(Node* root, int key) {
    if (root == nullptr || root->data == key) {
        return root;
    }

    if (key < root->data) {
        return search(root->left, key);
    }
    return search(root->right, key);
}

void print_inorder(Node* root) {
    if (root != nullptr) {
        print_inorder(root->left);
        cout << root->data << " ";
```

```
            print_inorder(root->right);
        }
    }
};

int main() {
    BinarySearchTree bst;
    Node* root = nullptr;

    int n;
    cout << "Enter the number of nodes in the tree: ";
    cin >> n;

    cout << "Enter the values to insert into the BST: ";
    for (int i = 0; i < n; ++i) {
        int value;
        cin >> value;
        root = bst.insert_new_node(&bst, root, value);
    }

    int new_value;
    cout << "Enter a new value to insert into the tree: ";
    cin >> new_value;
    root = bst.insert_new_node(&bst, root, new_value);

    cout << "Number of nodes in the longest path (height of the tree): " << bst.longest_path(root)
<< endl;

    cout << "Minimum data value in the tree: " << bst.min_value(root) << endl;

    bst.swap_children(root);
    cout << "Tree after swapping left and right children at each node (Inorder Traversal): ";
    bst.print_inorder(root);
    cout << endl;

    int search_value;
    cout << "Enter a value to search in the tree: ";
    cin >> search_value;
    Node* result = bst.search(root, search_value);
    if (result) {
        cout << "Value " << search_value << " found in the tree." << endl;
    } else {
        cout << "Value " << search_value << " not found in the tree." << endl;
    }

    return 0;
}
```

4.A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Binary Search Tree for implementation.

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

```
struct Node
{
    string key;
    string value;
    Node* left;
    Node* right;

    Node(string key1, string value1)
    {
        key = key1;
        value = value1;
        left = nullptr;
        right = nullptr;
    }
};

// insert node
Node* insert(Node* root, string key1, string value1) {
    if (root == nullptr) {
        return new Node(key1, value1);
    }
    if (key1< root->key) {
        root->left = insert(root->left, key1, value1);
    }
    else if (key1> root->key) {
        root->right = insert(root->right, key1, value1);
    }
    else {
        cout << "Keyword already exists\n";
    }
    return root;
}

// find node
Node* search(Node* root, string find) {
    if (root == nullptr || root->key == find) {
        return root;
    }
    if (find < root->key) {
        return search(root->left, find);
    }
    else {
        return search(root->right, find);
    }
}

// minimum value
Node* findmin(Node* root) {
    while (root && root->left != nullptr) {
        root = root->left;
    }
    return root;
}
```

```
// function to delete
Node* deleteNode(Node* root, string delete_key) {
    if (root == nullptr)
        return root;

    if (delete_key < root->key) {
        root->left = deleteNode(root->left, delete_key);
    }
    else if (delete_key > root->key) {
        root->right = deleteNode(root->right, delete_key);
    }
    else {
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = findmin(root->right);
        root->key = temp->key;
        root->value = temp->value;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

// update function
Node* update(Node* root, string key, string updated_value) {
    Node* node = search(root, key);
    if (node) {
        node->value = updated_value;
        cout << "Value updated successfully\n";
    }
    else {
        cout << "Invalid key\n";
    }
    return root;
}

// inorder traversal
void inorder(Node* root) {
    if (root == nullptr) return;

    inorder(root->left);
    cout << root->key << ":" << root->value << endl;
    inorder(root->right);
}

// reverse inorder
```

```
void reverseInorder(Node* root) {
    if (root == nullptr) return;

    reverseInorder(root->right);
    cout << root->key << ":" << root->value << endl;
    reverseInorder(root->left);
}

// find max depth
int maxDepth(Node* root) {
    if (root == nullptr) {
        return 0;
    }
    return 1+ max(maxDepth(root->left), maxDepth(root->right));
}

int main() {
    Node* root = nullptr;
    int choice;
    string key, value;

    do {
        cout << "Menu\n";
        cout << "1. Add keyword\n";
        cout << "2. Delete key\n";
        cout << "3. Update key\n";
        cout << "4. Display in ascending order\n";
        cout << "5. Display in descending order\n";
        cout << "6. Find maximum depth\n";
        cout << "7. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        cin.ignore();

        switch (choice) {
        case 1:
            cout << "Enter key: ";
            getline(cin, key);
            cout << "Enter value: ";
            getline(cin, value);

            root = insert(root, key, value);
            break;

        case 2:
            cout << "Enter key to delete: ";
            getline(cin, key);
            root = deleteNode(root, key);
            break;

        case 3:
            cout << "Enter key to update: ";
            getline(cin, key);
            cout << "Enter new value: ";
```

```
            getline(cin, value);
            root = update(root, key, value);
            break;

        case 4:
            cout << "Dictionary in ascending order:\n";
            inorder(root);
            break;

        case 5:
            cout << "Dictionary in descending order:\n";
            reverseInorder(root);
            break;

        case 6:
            cout << "Maximum depth: " << maxDepth(root) << endl;
            break;

        case 7:
            cout << "Exiting...\n";
            break;

        default:
            cout << "Invalid choice\n";
            break;
        }
    } while (choice != 7);

    return 0;
}
```

5.Convert given binary tree into threaded binary tree. Analyze time and space complexity of the algorithm.

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

// Structure for a Node of Binary Tree
struct Node {
    int data;
    Node* left;
    Node* right;
    bool leftThread;
    bool rightThread;

    // Constructor to create a new node
    Node(int val) {
        data = val;
        left = right = nullptr;
        leftThread = false;
        rightThread = false;
    }
};

// A global pointer for the previous node during traversal
Node* prev = nullptr;
```

```
// Function to convert Binary Tree to Threaded Binary Tree
void convertToThreaded(Node* root) {
    // Base case: if root is null
    if (root == nullptr)
        return;

    // Recur for the left subtree
    convertToThreaded(root->left);

    // If the left child of the current node is NULL, we link it to the predecessor
    if (root->left == nullptr && prev != nullptr) {
        root->left = prev;
        root->leftThread = true; // Mark as a thread
    }

    // If the right child of the previous node is NULL, we link it to the current node
    if (prev != nullptr && prev->right == nullptr) {
        prev->right = root;
        prev->rightThread = true; // Mark as a thread
    }

    // Mark the current node as the previous node for the next iteration
    prev = root;

    // Recur for the right subtree
    convertToThreaded(root->right);
}

// Function to perform In-Order Traversal using Threaded Binary Tree
void inOrderTraversal(Node* root) {
    Node* curr = root;
    while (curr != nullptr) {
        // Traverse to the leftmost node
        while (curr != nullptr && curr->leftThread == false)
            curr = curr->left;

        // Process the current node
        cout << curr->data << " ";

        // If there's a thread to the next node, follow it
        while (curr->rightThread == true) {
            curr = curr->right;
            cout << curr->data << " ";
        }

        // Move to the right child
        curr = curr->right;
    }
}

// Helper function to create a new node
Node* createNode(int data) {
    return new Node(data);
```

```
}

int main() {
    // Create a simple Binary Tree
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->right = createNode(6);

    // Convert Binary Tree to Threaded Binary Tree
    convertToThreaded(root);

    // Perform in-order traversal of the threaded binary tree
    cout << "In-Order Traversal of Threaded Binary Tree: ";
    inOrderTraversal(root);

    return 0;
}
```

6.Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph.Identify the prominent land marks as nodes and perform DFS and BFS on that.

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

// Graph class using adjacency list
class Graph {
public:
    unordered_map<string, list<string>> adjList;

    // Add an edge between two nodes (landmarks)
    void addEdge(string u, string v) {
        adjList[u].push_back(v);
        adjList[v].push_back(u); // Since this is an undirected graph
    }

    // Depth-First Search (DFS)
    void DFS(string start) {
        unordered_map<string, bool> visited;
        DFSUtil(start, visited);
    }

    void DFSUtil(string node, unordered_map<string, bool>& visited) {
        visited[node] = true;
        cout << node << " ";

        // Recur for all the vertices adjacent to this vertex
        for (auto neighbor : adjList[node]) {
            if (!visited[neighbor]) {
                DFSUtil(neighbor, visited);
            }
```

```
        }
    }

    // Breadth-First Search (BFS)
    void BFS(string start) {
        unordered_map<string, bool> visited;
        list<string> queue;

        visited[start] = true;
        queue.push_back(start);

        while (!queue.empty()) {
            string node = queue.front();
            queue.pop_front();
            cout << node << " ";

            for (auto neighbor : adjList[node]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.push_back(neighbor);
                }
            }
        }
    }
};

int main() {
    Graph g;

    // Adding edges based on the landmarks' connections
    g.addEdge("Main Gate", "Library");
    g.addEdge("Library", "Cafeteria");
    g.addEdge("Cafeteria", "Auditorium");
    g.addEdge("Auditorium", "Sports Complex");
    g.addEdge("Sports Complex", "Hostel");
    g.addEdge("Hostel", "Gym");
    g.addEdge("Gym", "Main Gate");

    cout << "DFS Traversal starting from Main Gate: ";
    g.DFS("Main Gate");
    cout << endl;

    cout << "BFS Traversal starting from Main Gate: ";
    g.BFS("Main Gate");
    cout << endl;

    return 0;
}
```

LA7

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.

Check whether the graph is connected or not. Justify the storage representation used.

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

class FlightGraph {
    int V; // Number of cities
    vector<list<pair<int, int>>> adjList; // city -> (connected city, cost)
    unordered_map<string, int> cityToIndex;
    vector<string> indexToCity;

public:
    FlightGraph(int v) {
        V = v;
        adjList.resize(V);
        indexToCity.resize(V);
    }

    void addCity(int index, const string& name) {
        cityToIndex[name] = index;
        indexToCity[index] = name;
    }

    void addFlight(const string& from, const string& to, int cost) {
        int u = cityToIndex[from];
        int v = cityToIndex[to];
        adjList[u].push_back({v, cost});
        adjList[v].push_back({u, cost}); // Since flights are bi-directional
    }

    void displayGraph() {
        cout << "\nAdjacency List Representation:\n";
        for (int i = 0; i < V; i++) {
            cout << indexToCity[i] << " -> ";
            for (auto& edge : adjList[i]) {
                cout << indexToCity[edge.first] << "(" << edge.second << ") ";
            }
            cout << endl;
        }
    }

    void DFS(int v, vector<bool>& visited) {
        visited[v] = true;
        for (auto& neighbor : adjList[v]) {
            if (!visited[neighbor.first])
                DFS(neighbor.first, visited);
        }
    }

    bool isConnected() {
```

```
        vector<bool> visited(V, false);
        DFS(0, visited);
        for (bool flag : visited) {
            if (!flag) return false;
        }
        return true;
    }
};

// Main Function
int main() {
    int numCities, numFlights;
    cout << "Enter number of cities: ";
    cin >> numCities;

    FlightGraph graph(numCities);

    cout << "Enter city names:\n";
    for (int i = 0; i < numCities; i++) {
        string city;
        cout << "City " << i + 1<< ": ";
        cin >> city;
        graph.addCity(i, city);
    }

    cout << "Enter number of flight connections: ";
    cin >> numFlights;

    cout << "Enter flight details (from to cost):\n";
    for (int i = 0; i < numFlights; i++) {
        string from, to;
        int cost;
        cout << "Flight " << i + 1<< ": ";
        cin >> from >> to >> cost;
        graph.addFlight(from, to, cost);
    }

    graph.displayGraph();

    if (graph.isConnected()) {
        cout << "\nThe flight graph is CONNECTED.\n";
    } else {
        cout << "\nThe flight graph is NOT CONNECTED.\n";
    }

    return 0;
}
```

LA8

Given sequence k = k1<="" ...="" <="" div="" style="box-sizing: border-box;">

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

```
const int MAX = 100;

// Function to construct the optimal BST
void optimalBST(vector<double>& p, int n) {
    double cost[MAX][MAX] = {0};
    double sum[MAX][MAX] = {0};
    int root[MAX][MAX] = {0};

    // Initialization
    for (int i = 0; i < n; i++) {
        cost[i][i] = p[i];
        sum[i][i] = p[i];
        root[i][i] = i;
    }

    // l is chain length
    for (int l = 2; l <= n; l++) {
        for (int i = 0; i <= n - l; i++) {
            int j = i + l - 1;
            cost[i][j] = 1e9; // Infinity

            // Sum of probabilities from i to j
            sum[i][j] = sum[i][j - 1] + p[j];

            // Try all roots from i to j
            for (int r = i; r <= j; r++) {
                double left = (r > i) ? cost[i][r - 1] : 0;
                double right = (r < j) ? cost[r + 1][j] : 0;
                double temp = left + right + sum[i][j];

                if (temp < cost[i][j]) {
                    cost[i][j] = temp;
                    root[i][j] = r;
                }
            }
        }
    }

    // Print final cost
    cout << fixed << setprecision(3);
    cout << "Minimum cost of optimal BST: " << cost[0][n - 1] << endl;

    // Print root table (optional)
    cout << "\nRoot Table:\n";
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            cout << root[i][j] + 1 << " ";  // +1 for 1-based key numbering
        }
        cout << endl;
    }
}

// Driver function
```

```
int main() {
    int n;
    cout << "Enter number of keys: ";
    cin >> n;

    vector<double> p(n);
    cout << "Enter probabilities for each key:\n";
    for (int i = 0; i < n; i++) {
        cout << "p[" << i + 1<< "]: ";
        cin >> p[i];
    }

    optimalBST(p, n);
    return 0;
}
```

LA9

A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword

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

// Node structure for AVL Tree
struct Node {
    string keyword;
    string meaning;
    Node* left;
    Node* right;
    int height;

    Node(string k, string m) {
        keyword = k;
        meaning = m;
        left = right = nullptr;
        height = 1;
    }
};

// Utility functions
int height(Node* n) {
    return n ? n->height : 0;
}

int getBalance(Node* n) {
    return n ? height(n->left) - height(n->right) : 0;
}

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Right rotate
```

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}

// Left rotate
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}

// Insert node in AVL
Node* insert(Node* node, string keyword, string meaning) {
    if (!node) return new Node(keyword, meaning);

    if (keyword < node->keyword)
        node->left = insert(node->left, keyword, meaning);
    else if (keyword > node->keyword)
        node->right = insert(node->right, keyword, meaning);
    else {
        cout << "Keyword already exists. Use update option.\n";
        return node;
    }

    node->height = 1+ max(height(node->left), height(node->right));

    int balance = getBalance(node);

    // Balancing the tree
    if (balance > 1&& keyword < node->left->keyword)
        return rightRotate(node);

    if (balance < -1&& keyword > node->right->keyword)
        return leftRotate(node);

    if (balance > 1&& keyword > node->left->keyword) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
```

```
    }

    if (balance < -1&& keyword < node->right->keyword) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// Find minimum value node
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != nullptr)
        current = current->left;
    return current;
}

// Delete node
Node* deleteNode(Node* root, string keyword) {
    if (!root)
        return root;

    if (keyword < root->keyword)
        root->left = deleteNode(root->left, keyword);
    else if (keyword > root->keyword)
        root->right = deleteNode(root->right, keyword);
    else {
        if (!root->left || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            if (!temp) {
                temp = root;
                root = nullptr;
            } else
                *root = *temp;
            delete temp;
        } else {
            Node* temp = minValueNode(root->right);
            root->keyword = temp->keyword;
            root->meaning = temp->meaning;
            root->right = deleteNode(root->right, temp->keyword);
        }
    }

    if (!root) return root;

    root->height = 1+ max(height(root->left), height(root->right));
    int balance = getBalance(root);

    // Rebalance
    if (balance > 1&& getBalance(root->left) >= 0)
        return rightRotate(root);

    if (balance > 1&& getBalance(root->left) < 0) {
```

```
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (balance < -1&& getBalance(root->right) <= 0)
        return leftRotate(root);

    if (balance < -1&& getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Update keyword meaning
void updateMeaning(Node* root, string keyword, string newMeaning) {
    while (root) {
        if (keyword == root->keyword) {
            root->meaning = newMeaning;
            cout << "Meaning updated.\n";
            return;
        } else if (keyword < root->keyword)
            root = root->left;
        else
            root = root->right;
    }
    cout << "Keyword not found.\n";
}

// Inorder (ascending)
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
        cout << root->keyword << " : " << root->meaning << endl;
        inorder(root->right);
    }
}

// Reverse inorder (descending)
void reverseInorder(Node* root) {
    if (root) {
        reverseInorder(root->right);
        cout << root->keyword << " : " << root->meaning << endl;
        reverseInorder(root->left);
    }
}

// Search keyword and return number of comparisons
int search(Node* root, string keyword, int& comparisons) {
    while (root) {
        comparisons++;
        if (keyword == root->keyword) {
            cout << "Meaning: " << root->meaning << endl;
```

```
            return comparisons;
        } else if (keyword < root->keyword)
            root = root->left;
        else
            root = root->right;
    }
    cout << "Keyword not found.\n";
    return comparisons;
}

int main() {
    Node* root = nullptr;
    int choice;
    string keyword, meaning;

    do {
        cout << "\n--- Dictionary Menu ---\n";
        cout << "1. Add Keyword\n2. Delete Keyword\n3. Update Meaning\n4. Display Ascending\n";
        cout << "5. Display Descending\n6. Search Keyword\n7. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter keyword: ";
            cin >> keyword;
            cout << "Enter meaning: ";
            cin.ignore();
            getline(cin, meaning);
            root = insert(root, keyword, meaning);
            break;
        case 2:
            cout << "Enter keyword to delete: ";
            cin >> keyword;
            root = deleteNode(root, keyword);
            break;
        case 3:
            cout << "Enter keyword to update: ";
            cin >> keyword;
            cout << "Enter new meaning: ";
            cin.ignore();
            getline(cin, meaning);
            updateMeaning(root, keyword, meaning);
            break;
        case 4:
            cout << "\nDictionary in Ascending Order:\n";
            inorder(root);
            break;
        case 5:
            cout << "\nDictionary in Descending Order:\n";
            reverseInorder(root);
            break;
        case 6: {
            cout << "Enter keyword to search: ";
```

```
            cin >> keyword;
            int comparisons = 0;
            int total = search(root, keyword, comparisons);
            cout << "Comparisons made: " << total << endl;
            break;
        }
        case 7:
            cout << "Exiting...\n";
            break;
        default:
            cout << "Invalid choice!\n";
        }
    } while (choice != 7);

    return 0;
}
```

LA10

Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure. Analyze the algorithm.

```
#include <iostream>
#include <vector>
#include <queue> // For priority_queue
using namespace std;

int main() {
    int n;
    cout << "Enter number of students: ";
    cin >> n;

    vector<int> marks(n);
    cout << "Enter marks of " << n << " students:\n";
    for (int i = 0; i < n; ++i) {
        cin >> marks[i];
    }

    // Max-heap (default)
    priority_queue<int> maxHeap;

    // Min-heap (using greater comparator)
    priority_queue<int, vector<int>, greater<int>> minHeap;

    // Insert all marks into both heaps
    for (int mark : marks) {
        maxHeap.push(mark);
        minHeap.push(mark);
    }

    // The top of max-heap is the maximum mark
    cout << "\nMaximum Marks: " << maxHeap.top() << endl;

    // The top of min-heap is the minimum mark
```

```
    cout << "Minimum Marks: " << minHeap.top() << endl;

    return 0;
}
```

LA11

Assume we have two input and two output tapes to perform the sorting. The internal Memory can hold and sort m records at a time. Write a program in java for external sorting. Find out time complexity.

```
import java.io.*;
import java.util.*;

public class Main {

    // Method to sort a chunk of data and store it in an output file
    public static void sortChunk(List<String> chunk, File outputFile) throws IOException {
        // Sort the chunk in memory
        Collections.sort(chunk);

        // Write the sorted chunk back to the output file
        BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile));
        for (String record : chunk) {
            writer.write(record);
            writer.newLine();
        }
        writer.close();
    }

    // Method to merge two sorted files into a single sorted output file
    public static void mergeSortedFiles(File file1, File file2, File outputFile) throws IOException {
        BufferedReader reader1 = new BufferedReader(new FileReader(file1));
        BufferedReader reader2 = new BufferedReader(new FileReader(file2));
        BufferedWriter writer = new BufferedWriter(new FileWriter(outputFile));

        String line1 = reader1.readLine();
        String line2 = reader2.readLine();

        while (line1 != null && line2 != null) {
            if (line1.compareTo(line2) < 0) {
                writer.write(line1);
                writer.newLine();
                line1 = reader1.readLine();
            } else {
                writer.write(line2);
                writer.newLine();
                line2 = reader2.readLine();
            }
        }

        // Write any remaining lines from file1 or file2
        while (line1 != null) {
            writer.write(line1);
            writer.newLine();
            line1 = reader1.readLine();
```

```
    }
    while (line2 != null) {
        writer.write(line2);
        writer.newLine();
        line2 = reader2.readLine();
    }

    reader1.close();
    reader2.close();
    writer.close();
}

// Main method for external sorting simulation
public static void externalSort(File inputFile, int m) throws IOException {
    List<File> sortedChunks = new ArrayList<>();
    BufferedReader reader = new BufferedReader(new FileReader(inputFile));

    // Read and sort chunks
    List<String> chunk = new ArrayList<>();
    String line;
    while ((line = reader.readLine()) != null) {
        chunk.add(line);
        if (chunk.size() == m) {
            // Sort this chunk and write to a file
            File chunkFile = File.createTempFile("sorted_chunk_", ".txt");
            sortChunk(chunk, chunkFile);
            sortedChunks.add(chunkFile);
            chunk.clear();
        }
    }

    // Sort and save the last chunk if it has any remaining data
    if (!chunk.isEmpty()) {
        File chunkFile = File.createTempFile("sorted_chunk_", ".txt");
        sortChunk(chunk, chunkFile);
        sortedChunks.add(chunkFile);
    }
    reader.close();

    // Merge sorted chunks
    while (sortedChunks.size() > 1) {
        List<File> nextRound = new ArrayList<>();
        for (int i = 0; i < sortedChunks.size(); i += 2) {
            if (i + 1< sortedChunks.size()) {
                File mergedFile = File.createTempFile("merged_", ".txt");
                mergeSortedFiles(sortedChunks.get(i), sortedChunks.get(i + 1), mergedFile);
                nextRound.add(mergedFile);
            } else {
                nextRound.add(sortedChunks.get(i));
            }
        }
        sortedChunks = nextRound;
    }
```

```
        // The last remaining file is the fully sorted file
        File sortedFile = sortedChunks.get(0);
        System.out.println("Sorted data written to: " + sortedFile.getAbsolutePath());
    }

    public static void main(String[] args) {
        try {
            File inputFile = new File("input.txt");
            int m = 5;  // Number of records that fit into memory

            // Example: Creating an input file (can be skipped if input file is already available)
            BufferedWriter writer = new BufferedWriter(new FileWriter(inputFile));
            writer.write("10\n20\n5\n15\n30\n40\n25\n35\n60\n50\n");
            writer.close();

            // Perform the external sorting
            externalSort(inputFile, m);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

LA12

Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.

```
import java.io.IOException;
import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StudentInfoSystem.run();
    }
}

class Student {
    int roll;
    String name, division, address;

    public Student(int roll, String name, String division, String address) {
        this.roll = roll;
        this.name = name;
        this.division = division;
        this.address = address;
    }

    public String toFileString() {
        return roll + "|" + name + "|" + division + "|" + address;
    }

    public static Student fromFileString(String line) {
        String[] parts = line.split("\\|");
```

```
        if (parts.length < 4) return null;
        return new Student(Integer.parseInt(parts[0]), parts[1], parts[2], parts[3]);
    }
}

class StudentInfoSystem {
    static final String FILENAME = "students.txt";
    static Scanner sc = new Scanner(System.in);

    public static void addStudent() throws IOException {
        System.out.print("Enter Roll No: ");
        int roll = sc.nextInt();
        sc.nextLine(); // flush newline

        System.out.print("Enter Name: ");
        String name = sc.nextLine();

        System.out.print("Enter Division: ");
        String division = sc.nextLine();

        System.out.print("Enter Address: ");
        String address = sc.nextLine();

        Student s = new Student(roll, name, division, address);

        BufferedWriter writer = new BufferedWriter(new FileWriter(FILENAME, true));
        writer.write(s.toFileString());
        writer.newLine();
        writer.close();

        System.out.println("Student added successfully!");
    }

    public static void displayStudent() throws IOException {
        System.out.print("Enter Roll No to search: ");
        int roll = sc.nextInt();

        BufferedReader reader = new BufferedReader(new FileReader(FILENAME));
        String line;
        boolean found = false;

        while ((line = reader.readLine()) != null) {
            Student s = Student.fromFileString(line);
            if (s != null && s.roll == roll) {
                System.out.println("\n--- Student Details ---");
                System.out.println("Roll No   : " + s.roll);
                System.out.println("Name      : " + s.name);
                System.out.println("Division  : " + s.division);
                System.out.println("Address   : " + s.address);
                found = true;
                break;
            }
        }
        reader.close();
```

```
        if (!found) {
            System.out.println("Student with Roll No " + roll + " not found.");
        }
    }

    public static void deleteStudent() throws IOException {
        System.out.print("Enter Roll No to delete: ");
        int roll = sc.nextInt();

        File inputFile = new File(FILENAME);
        File tempFile = new File("temp.txt");

        BufferedReader reader = new BufferedReader(new FileReader(inputFile));
        BufferedWriter writer = new BufferedWriter(new FileWriter(tempFile));

        String line;
        boolean deleted = false;

        while ((line = reader.readLine()) != null) {
            Student s = Student.fromFileString(line);
            if (s != null && s.roll == roll) {
                deleted = true;
                continue; // skip writing
            }
            writer.write(line);
            writer.newLine();
        }

        reader.close();
        writer.close();

        if (inputFile.delete()) {
            tempFile.renameTo(inputFile);
        }

        if (deleted) {
            System.out.println("Student deleted successfully.");
        } else {
            System.out.println("Student with Roll No " + roll + " not found.");
        }
    }

    public static void run() throws IOException {
        int choice;
        do {
            System.out.println("\n--- Student Information System ---");
            System.out.println("1. Add Student");
            System.out.println("2. Display Student by Roll Number");
            System.out.println("3. Delete Student");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            choice = sc.nextInt();
```