Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 1

A1: 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

Program:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash_function(self, key):
        return sum(ord(c) for c in key) % self.size

    def insert(self, key, value):
        index = self._hash_function(key)
        self.table[index].append((key, value))

    def separate_chaining_search(self, key):
        index = self._hash_function(key)
        comparisons = 0
        for item in self.table[index]:
            comparisons += 1
            if item[0] == key:
                return item[1], comparisons
        return None, comparisons

    def linear_probing_search(self, key):
        index = self._hash_function(key)
        comparisons = 0
```

```
        i = index
        while self.table[i]:
            comparisons += 1
            if self.table[i][0][0] == key:
                return self.table[i][0][1], comparisons
            i = (i + 1) % self.size
            if i == index:
                break
        return None, comparisons


# Test the implementation
telephone_book = HashTable(10)

# Insert telephone numbers
telephone_book.insert("John Smith", "1234567890")
telephone_book.insert("Alice Johnson", "9876543210")
telephone_book.insert("Bob Davis", "5678901234")
telephone_book.insert("Emma Wilson", "4321098765")
telephone_book.insert("Oliver Thompson", "9087654321")

# Perform separate chaining search
print("Separate Chaining Search:")
print(telephone_book.separate_chaining_search("John Smith"))
print(telephone_book.separate_chaining_search("Emma Wilson"))
print(telephone_book.separate_chaining_search("Unknown Client"))

# Perform linear probing search
print("\nLinear Probing Search:")
print(telephone_book.linear_probing_search("John Smith"))
print(telephone_book.linear_probing_search("Emma Wilson"))
```

```
print(telephone_book.linear_probing_search("Unknown Client"))
```

output:

Separate Chaining Search:

('1234567890', 1)

('4321098765', 1)

(None, 2)

Linear Probing Search:

('1234567890', 1)

('4321098765', 1)

(None, 5)

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 2

A2: To create ADT that implement the "set" concept. i. Add (newElement) -Place a value into the set ii. Remove (element) Remove the value iii. Contains (element) Return true if element is in collection iv. Size () Return number of values in collection Iterator () Return an iterator used to loop over collection v. Intersection of two sets vi. Union of two sets vii. Difference

Program:

```
class Set:
    def __init__(self):
        self.elements = []

    def add(self, new_element):
        if new_element not in self.elements:
            self.elements.append(new_element)

    def remove(self, element):
        if element in self.elements:
            self.elements.remove(element)

    def contains(self, element):
        return element in self.elements

    def size(self):
        return len(self.elements)

    def iterator(self):
        return iter(self.elements)

    def intersection(self, other_set):
        intersection_set = Set()
```

```
        for element in self.elements:
            if other_set.contains(element):
                intersection_set.add(element)
        return intersection_set

    def union(self, other_set):
        union_set = Set()
        for element in self.elements:
            union_set.add(element)
        for element in other_set.iterator():
            union_set.add(element)
        return union_set

    def difference(self, other_set):
        difference_set = Set()
        for element in self.elements:
            if not other_set.contains(element):
                difference_set.add(element)
        return difference_set


# Test the Set implementation
set1 = Set()
set1.add(1)
set1.add(2)
set1.add(3)

set2 = Set()
set2.add(2)
set2.add(3)
set2.add(4)
```

```
print("Set 1:", list(set1.iterator()))
print("Set 2:", list(set2.iterator()))

print("Intersection:", list(set1.intersection(set2).iterator()))
print("Union:", list(set1.union(set2).iterator()))
print("Difference:", list(set1.difference(set2).iterator()))
```

Output:

Set 1: [1, 2, 3]

Set 2: [2, 3, 4]

Intersection: [2, 3]

Union: [1, 2, 3, 4]

Difference: [1]

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 3

B1: A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method

Program :

```
#include <iostream>

#include <vector>

using namespace std;


class Node {

public:

   string name;

   vector<Node*> children;


   Node(const string& name) {

      this->name = name;

   }

};


void printNodes(Node* node) {

   if (node == nullptr) {

      return;

   }


   cout << node->name << endl;

   for (Node* child : node->children) {

      printNodes(child);

   }
```

```
}

int main() {
    // Constructing the tree
    Node* book = new Node("Book");

    // Add chapters
    Node* chapter1 = new Node("Chapter 1");
    Node* chapter2 = new Node("Chapter 2");
    book->children.push_back(chapter1);
    book->children.push_back(chapter2);

    // Add sections to Chapter 1
    Node* section1_1 = new Node("Section 1.1");
    Node* section1_2 = new Node("Section 1.2");
    chapter1->children.push_back(section1_1);
    chapter1->children.push_back(section1_2);

    // Add subsections to Section 1.1
    Node* subsection1_1_1 = new Node("Subsection 1.1.1");
    Node* subsection1_1_2 = new Node("Subsection 1.1.2");
    section1_1->children.push_back(subsection1_1_1);
    section1_1->children.push_back(subsection1_1_2);

    // Add sections to Chapter 2
    Node* section2_1 = new Node("Section 2.1");
    chapter2->children.push_back(section2_1);

    // Print all nodes
    cout << "Nodes in the tree:" << endl;
    printNodes(book);
```

```
    // Free memory
    delete subsection1_1_1;
    delete subsection1_1_2;
    delete section1_1;
    delete section1_2;
    delete section2_1;
    delete chapter1;
    delete chapter2;
    delete book;

    return 0;
}
```

Output:

```
Nodes in the tree:
Book
Chapter 1
Section 1.1
Subsection 1.1.1
Subsection 1.1.2
Section 1.2
Chapter 2
Section 2.1
```

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 4

B2: 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 from root 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

Program:

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

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

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

class BST {
private:
    Node* root;

    Node* insertRecursive(Node* root, int value) {
        if (root == nullptr) {
```

```
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insertRecursive(root->left, value);
    } else if (value > root->data) {
        root->right = insertRecursive(root->right, value);
    }

    return root;
}

int findHeightRecursive(Node* root) {
    if (root == nullptr) {
        return 0;
    }

    int leftHeight = findHeightRecursive(root->left);
    int rightHeight = findHeightRecursive(root->right);

    return max(leftHeight, rightHeight) + 1;
}

Node* findMinimum(Node* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}

Node* swapPointers(Node* root) {
```

```
        if (root == nullptr) {
            return nullptr;
        }

        Node* temp = root->left;
        root->left = root->right;
        root->right = temp;

        swapPointers(root->left);
        swapPointers(root->right);

        return root;
    }

    bool searchRecursive(Node* root, int value) {
        if (root == nullptr) {
            return false;
        }

        if (root->data == value) {
            return true;
        } else if (value < root->data) {
            return searchRecursive(root->left, value);
        } else {
            return searchRecursive(root->right, value);
        }
    }

public:
    BST() {
        root = nullptr;
```

```
    }

    void insert(int value) {
        root = insertRecursive(root, value);
    }

    int findHeight() {
        return findHeightRecursive(root);
    }

    int findMinimumValue() {
        Node* minNode = findMinimum(root);
        return minNode->data;
    }

    void swapTreePointers() {
        root = swapPointers(root);
    }

    bool search(int value) {
        return searchRecursive(root, value);
    }
};

int main() {
    BST bst;

    // Construct the binary search tree
    bst.insert(8);
    bst.insert(3);
    bst.insert(10);
```

```
bst.insert(1);
bst.insert(6);
bst.insert(14);
bst.insert(4);
bst.insert(7);
bst.insert(13);

cout << "Binary Search Tree Construction Successful!" << endl;

// Insert a new node
bst.insert(9);
cout << "New node (9) inserted successfully!" << endl;

// Find the number of nodes in the longest path from the root
int longestPath = bst.findHeight();
cout << "Number of nodes in the longest path from the root: " << longestPath << endl;

// Find the minimum data value in the tree
int minValue = bst.findMinimumValue();
cout << "Minimum data value in the tree: " << minValue << endl;

// Swap the left and right pointers at every node
bst.swapTreePointers();
cout << "Tree pointers swapped successfully!" << endl;

// Search for a value
int valueToSearch = 4;
bool found = bst.search(valueToSearch);
if (found) {
    cout << valueToSearch << " found in the tree!" << endl;
} else {
```

```
        cout << valueToSearch << " not found in the tree!" << endl;
    }

    return 0;
}
```

Output:

Binary Search Tree Construction Successful!

New node (9) inserted successfully!

Number of nodes in the longest path from the root: 4

Minimum data value in the tree: 1

Tree pointers swapped successfully!

4 found in the tree!

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 5

B3: Convert given binary tree into threaded binary tree. Analyze time and space complexity of the algorithm

Program:

```
#include <iostream>

using namespace std;


// Node structure for binary tree

struct Node {

    int data;

    Node* left;

    Node* right;

    bool isThreaded;


    Node(int value) {

        data = value;

        left = nullptr;

        right = nullptr;

        isThreaded = false;

    }

};


// Function to perform the threaded binary tree conversion using Morris Traversal

void convertToThreaded(Node* root) {

    if (root == nullptr) {

        return;

    }
```

```
    Node* current = root;
    Node* prev = nullptr;

    while (current != nullptr) {
        if (current->left == nullptr) {
            if (prev != nullptr && prev->right == nullptr) {
                prev->right = current;
                prev->isThreaded = true;
            }

            prev = current;
            current = current->right;
        } else {
            Node* predecessor = current->left;

            while (predecessor->right != nullptr && predecessor->right != current) {
                predecessor = predecessor->right;
            }

            if (predecessor->right == nullptr) {
                predecessor->right = current;
                predecessor->isThreaded = true;
                current = current->left;
            } else {
                predecessor->right = nullptr;
                current = current->right;
            }
        }
    }
}
```

```
// Function to perform an inorder traversal of the threaded binary tree
void inorderTraversal(Node* root) {
  if (root == nullptr) {
    return;
  }

  Node* current = root;

  while (current != nullptr) {
    // Find the leftmost threaded node
    while (current->left != nullptr && !current->isThreaded) {
      current = current->left;
    }

    // Print the node value
    cout << current->data << " ";

    // If the right child is a thread, move to the threaded node
    if (current->isThreaded) {
      current = current->right;
    } else {
      // Otherwise, move to the right child
      current = current->right;
      while (current != nullptr && !current->isThreaded) {
        current = current->left;
      }
    }
  }
}
```

```
// Function to create a sample binary tree
Node* createBinaryTree() {
   Node* root = new Node(1);
   root->left = new Node(2);
   root->right = new Node(3);
   root->left->left = new Node(4);
   root->left->right = new Node(5);
   root->right->left = new Node(6);
   root->right->right = new Node(7);

   return root;
}

int main() {
   Node* root = createBinaryTree();

   cout << "Inorder Traversal of Binary Tree: ";
   inorderTraversal(root);
   cout << endl;

   convertToThreaded(root);

   cout << "Inorder Traversal of Threaded Binary Tree: ";
   inorderTraversal(root);
   cout << endl;
    return 0;
}
```

Output:

Inorder Traversal of Binary Tree: 4 2 5 1 6 3 7

Inorder Traversal of Threaded Binary Tree: 4 2 5 1 6 3 7

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 6

C1 : 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

Program:

```
#include <iostream>

#include <list>

#include <vector>

using namespace std;


class Graph {

private:

   int numCities;

   vector<list<int>> adjList;


public:

   Graph(int cities) {

      numCities = cities;

      adjList.resize(numCities);

   }


   void addFlightPath(int source, int destination) {

      adjList[source].push_back(destination);

      adjList[destination].push_back(source);

   }
```

```
    bool isConnected() {
        vector<bool> visited(numCities, false);

        // Perform DFS traversal
        dfs(0, visited);

        // Check if all cities are visited
        for (bool v : visited) {
            if (!v) {
                return false;
            }
        }

        return true;
    }

    void dfs(int city, vector<bool>& visited) {
        visited[city] = true;

        for (int neighbor : adjList[city]) {
            if (!visited[neighbor]) {
                dfs(neighbor, visited);
            }
        }
    }
};

int main() {
    // Create a graph
    Graph graph(6);
```

```
    // Add flight paths between cities
    graph.addFlightPath(0, 1);
    graph.addFlightPath(1, 2);
    graph.addFlightPath(2, 3);
    graph.addFlightPath(3, 4);
    graph.addFlightPath(4, 5);

    // Check if the graph is connected
    bool connected = graph.isConnected();

    if (connected) {
        cout << "The graph is connected." << endl;
    } else {
        cout << "The graph is not connected." << endl;
    }

    return 0;
}
```

Output:

The graph is connected.

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 7

C2: You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures

Program:

```
#include <iostream>

#include <vector>

#include <queue>

using namespace std;


// Structure to represent an edge in the graph

struct Edge {

    int source;

    int destination;

    int cost;


    Edge(int src, int dest, int cst) {

        source = src;

        destination = dest;

        cost = cst;

    }

};


// Structure to represent a disjoint set

struct DisjointSet {

    vector<int> parent;
```

```
vector<int> rank;

DisjointSet(int n) {
    parent.resize(n);
    rank.resize(n, 0);

    for (int i = 0; i < n; i++) {
        parent[i] = i;
    }
}

int find(int x) {
    if (parent[x] != x) {
        parent[x] = find(parent[x]);
    }
    return parent[x];
}

void unionSets(int x, int y) {
    int xRoot = find(x);
    int yRoot = find(y);

    if (rank[xRoot] < rank[yRoot]) {
        parent[xRoot] = yRoot;
    } else if (rank[xRoot] > rank[yRoot]) {
        parent[yRoot] = xRoot;
    } else {
        parent[yRoot] = xRoot;
        rank[xRoot]++;
    }
}
```

```
};

// Function to find the minimum cost of leasing phone lines to connect all offices
int findMinimumCost(vector<Edge>& edges, int numCities) {
    // Sort edges in non-decreasing order of cost
    sort(edges.begin(), edges.end(), [](const Edge& a, const Edge& b) {
        return a.cost < b.cost;
    });

    DisjointSet disjointSet(numCities);
    int minCost = 0;

    for (const Edge& edge : edges) {
        int sourceRoot = disjointSet.find(edge.source);
        int destinationRoot = disjointSet.find(edge.destination);

        // Add the edge to the minimum spanning tree if it doesn't create a cycle
        if (sourceRoot != destinationRoot) {
            disjointSet.unionSets(sourceRoot, destinationRoot);
            minCost += edge.cost;
        }
    }

    return minCost;
}

int main() {
    // Number of cities (offices)
    int numCities = 5;

    // Create a vector of edges representing the costs between pairs of cities
```

```
    vector<Edge> edges;
    edges.push_back(Edge(0, 1, 2));
    edges.push_back(Edge(0, 2, 3));
    edges.push_back(Edge(1, 2, 1));
    edges.push_back(Edge(1, 3, 4));
    edges.push_back(Edge(2, 3, 2));
    edges.push_back(Edge(2, 4, 3));
    edges.push_back(Edge(3, 4, 1));

    // Find the minimum cost of leasing phone lines to connect all offices
    int minCost = findMinimumCost(edges, numCities);

    cout << "Minimum cost of leasing phone lines to connect all offices: " << minCost << endl;

    return 0;
}
```

Output:

Minimum cost of leasing phone lines to connect all offices: 8

Name: Abhishek Patil

Div -B Roll no.84

Practical no : 8

D1: Given sequence k = k1 < ...

Program:

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

// Structure to represent a node in the binary search tree
struct Node {
    int key;
    Node* left;
    Node* right;

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

// Function to build the optimal binary search tree
Node* buildOptimalBST(vector<int>& keys, vector<double>& probabilities, int start, int end) {
    // Base case: no keys in the range
    if (start > end) {
        return nullptr;
    }

    // Find the index of the root with minimum average search cost
```

```
    int minRootIndex = start;
    double minCost = probabilities[start];
    for (int i = start + 1; i <= end; i++) {
        if (probabilities[i] < minCost) {
            minCost = probabilities[i];
            minRootIndex = i;
        }
    }

    // Create the root node
    Node* root = new Node(keys[minRootIndex]);

    // Recursively build the left and right subtrees
    root->left = buildOptimalBST(keys, probabilities, start, minRootIndex - 1);
    root->right = buildOptimalBST(keys, probabilities, minRootIndex + 1, end);

    return root;
}

// Function to print the keys of the binary search tree in inorder traversal
void inorderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }

    inorderTraversal(root->left);
    cout << root->key << " ";
    inorderTraversal(root->right);
}

int main() {
```

```
    // Sorted keys
    vector<int> keys = {10, 20, 30, 40, 50};

    // Access probabilities for each key
    vector<double> probabilities = {0.1, 0.2, 0.05, 0.1, 0.15};

    // Build the optimal binary search tree
    Node* root = buildOptimalBST(keys, probabilities, 0, keys.size() - 1);

    // Print the keys of the binary search tree in inorder traversal
    cout << "Keys in the optimal binary search tree (inorder traversal): ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:

Keys in the optimal binary search tree (inorder traversal): 30 10 40 20 50

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 9

D2: 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 Height balance tree and find the complexity for finding a keyword

Program:

```
#include <iostream>

#include <string>

using namespace std;


// Structure to represent a node in the AVL tree

struct Node {

   string keyword;

   string meaning;

   int height;

   Node* left;

   Node* right;


   Node(string key, string val) {

      keyword = key;

      meaning = val;

      height = 1;

      left = nullptr;

      right = nullptr;

   }

};


// Class for the AVL Tree
```

```
class AVLTree {
private:
    Node* root;

    int height(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        return node->height;
    }

    int getBalance(Node* node) {
        if (node == nullptr) {
            return 0;
        }
        return height(node->left) - height(node->right);
    }

    Node* rotateRight(Node* node) {
        Node* leftChild = node->left;
        Node* leftRightChild = leftChild->right;

        leftChild->right = node;
        node->left = leftRightChild;

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

        return leftChild;
    }
```

```
Node* rotateLeft(Node* node) {
   Node* rightChild = node->right;
   Node* rightLeftChild = rightChild->left;

   rightChild->left = node;
   node->right = rightLeftChild;

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

   return rightChild;
}

Node* insertNode(Node* node, string key, string value) {
   if (node == nullptr) {
      return new Node(key, value);
   }

   if (key < node->keyword) {
      node->left = insertNode(node->left, key, value);
   } else if (key > node->keyword) {
      node->right = insertNode(node->right, key, value);
   } else {
      node->meaning = value;
      return node;
   }

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

   int balance = getBalance(node);
```

```
    // Left Left Case
    if (balance > 1 && key < node->left->keyword) {
        return rotateRight(node);
    }

    // Right Right Case
    if (balance < -1 && key > node->right->keyword) {
        return rotateLeft(node);
    }

    // Left Right Case
    if (balance > 1 && key > node->left->keyword) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->keyword) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != nullptr) {
        current = current->left;
    }
    return current;
```

```
}

Node* deleteNode(Node* node, string key) {
    if (node == nullptr) {
        return node;
    }

    if (key < node->keyword) {
        node->left = deleteNode(node->left, key);
    } else if (key > node->keyword) {
        node->right = deleteNode(node->right, key);
    } else {
        if (node->left == nullptr || node->right == nullptr) {
            Node* temp = node->left ? node->left : node->right;

            if (temp == nullptr) {
                temp = node;
                node = nullptr;
            } else {
                *node = *temp;
            }

            delete temp;
        } else {
            Node* temp = minValueNode(node->right);

            node->keyword = temp->keyword;
            node->meaning = temp->meaning;

            node->right = deleteNode(node->right, temp->keyword);
        }
```

```
}

if (node == nullptr) {
    return node;
}

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

int balance = getBalance(node);

// Left Left Case
if (balance > 1 && getBalance(node->left) >= 0) {
    return rotateRight(node);
}

// Left Right Case
if (balance > 1 && getBalance(node->left) < 0) {
    node->left = rotateLeft(node->left);
    return rotateRight(node);
}

// Right Right Case
if (balance < -1 && getBalance(node->right) <= 0) {
    return rotateLeft(node);
}

// Right Left Case
if (balance < -1 && getBalance(node->right) > 0) {
    node->right = rotateRight(node->right);
    return rotateLeft(node);
}
```

```
    return node;
}

void inorderTraversal(Node* node) {
    if (node == nullptr) {
        return;
    }

    inorderTraversal(node->left);
    cout << "Keyword: " << node->keyword << ", Meaning: " << node->meaning << endl;
    inorderTraversal(node->right);
}

void reverseInorderTraversal(Node* node) {
    if (node == nullptr) {
        return;
    }

    reverseInorderTraversal(node->right);
    cout << "Keyword: " << node->keyword << ", Meaning: " << node->meaning << endl;
    reverseInorderTraversal(node->left);
}

Node* searchNode(Node* node, string key, int& comparisons) {
    if (node == nullptr || node->keyword == key) {
        return node;
    }

    if (key < node->keyword) {
        comparisons++;
```

```
            return searchNode(node->left, key, comparisons);
        }

        comparisons++;
        return searchNode(node->right, key, comparisons);
    }

public:
    AVLTree() {
        root = nullptr;
    }

    void addKeyword(string key, string value) {
        root = insertNode(root, key, value);
        cout << "Keyword added: " << key << endl;
    }

    void deleteKeyword(string key) {
        root = deleteNode(root, key);
        cout << "Keyword deleted: " << key << endl;
    }

    void updateKeyword(string key, string value) {
        Node* node = searchNode(root, key);
        if (node != nullptr) {
            node->meaning = value;
            cout << "Keyword updated: " << key << endl;
        } else {
            cout << "Keyword not found: " << key << endl;
        }
    }
```

```
    void displayAscending() {
        cout << "Dictionary (sorted in ascending order):" << endl;
        inorderTraversal(root);
    }

    void displayDescending() {
        cout << "Dictionary (sorted in descending order):" << endl;
        reverseInorderTraversal(root);
    }

    int findKeyword(string key) {
        int comparisons = 0;
        Node* node = searchNode(root, key, comparisons);
        if (node != nullptr) {
            cout << "Keyword found: " << key << endl;
        } else {
            cout << "Keyword not found: " << key << endl;
        }
        return comparisons;
    }
};

int main() {
    AVLTree dictionary;

    dictionary.addKeyword("Apple", "A fruit");
    dictionary.addKeyword("Banana", "A tropical fruit");
    dictionary.addKeyword("Cat", "A domestic animal");
    dictionary.addKeyword("Dog", "A domestic animal");
```

```
    dictionary.displayAscending();

    dictionary.deleteKeyword("Banana");
    dictionary.updateKeyword("Apple", "A red fruit");
    dictionary.updateKeyword("Grapes", "A fruit");

    dictionary.displayDescending();

    int comparisons = dictionary.findKeyword("Dog");
    cout << "Number of comparisons: " << comparisons << endl;

    comparisons = dictionary.findKeyword("Elephant");
    cout << "Number of comparisons: " << comparisons << endl;

    return 0;
}
```

Output:

```
Keyword added: Apple
Keyword added: Banana
Keyword added: Cat
Keyword added: Dog
Dictionary (sorted in ascending order):
Keyword: Apple, Meaning: A fruit
Keyword: Banana, Meaning: A tropical fruit
Keyword: Cat, Meaning: A domestic animal
Keyword: Dog, Meaning: A domestic animal
Keyword deleted: Banana
Keyword updated: Apple
Keyword not found: Grapes
Dictionary (sorted in descending order):
```

Keyword: Dog, Meaning: A domestic animal

Keyword: Cat, Meaning: A domestic animal

Keyword: Apple, Meaning: A red fruit

Number of comparisons: 2

Keyword not found: Elephant

Number of comparisons: 3

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 10

E1: Implement Heap/ Shell Sort algorithm in Java demonstrating data structure with modularity of programming Lanauage.

Program:

```
import java.util.Arrays;


public class SortingAlgorithms {


  // Heap Sort

  public static void heapSort(int[] arr) {

    int n = arr.length;


    // Build max heap

    for (int i = n / 2 - 1; i >= 0; i--)

      heapify(arr, n, i);


    // Heap sort

    for (int i = n - 1; i > 0; i--) {

      // Swap root (max element) with the last element

      int temp = arr[0];

      arr[0] = arr[i];

      arr[i] = temp;


      // Heapify the reduced heap

      heapify(arr, i, 0);

    }

  }


  private static void heapify(int[] arr, int n, int i) {
```

```
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        // Swap root with the largest element
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Shell Sort
public static void shellSort(int[] arr) {
    int n = arr.length;

    // Start with a large gap and reduce it
    for (int gap = n / 2; gap > 0; gap /= 2) {
        // Perform insertion sort on elements gapped by 'gap'
```

```
            for (int i = gap; i < n; i++) {
                int temp = arr[i];
                int j;
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                    arr[j] = arr[j - gap];
                }
                arr[j] = temp;
            }
        }
    }

    public static void main(String[] args) {
        int[] arrHeapSort = {9, 5, 7, 2, 8, 3, 1, 6, 4};
        int[] arrShellSort = {9, 5, 7, 2, 8, 3, 1, 6, 4};

        System.out.println("Array before Heap Sort: " + Arrays.toString(arrHeapSort));
        heapSort(arrHeapSort);
        System.out.println("Array after Heap Sort: " + Arrays.toString(arrHeapSort));

        System.out.println("Array before Shell Sort: " + Arrays.toString(arrShellSort));
        shellSort(arrShellSort);
        System.out.println("Array after Shell Sort: " + Arrays.toString(arrShellSort));
    }
}
```

Output:

```
Array before Heap Sort: [9, 5, 7, 2, 8, 3, 1, 6, 4]
Array after Heap Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Array before Shell Sort: [9, 5, 7, 2, 8, 3, 1, 6, 4]
Array after Shell Sort: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Name: Abhishek Patil

Div -B Roll no.84

Practical no : 11

F1: The department maintains student information. The file contains the roll number, name, division, and address. Allow user to add, delete information of students. 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’s details. Use the sequential file to main the data

Program:

```
#include <iostream>

#include <fstream>

#include <string>

using namespace std;


struct Student {

   int rollNumber;

   string name;

   string division;

   string address;

};


void addStudent(const Student& student) {

   ofstream file("student_records.txt", ios::app);

   if (file.is_open()) {

      file << student.rollNumber << "," << student.name << "," << student.division << "," <<
student.address << endl;

      file.close();

      cout << "Student record added successfully." << endl;

   } else {

      cout << "Failed to open the file." << endl;

   }

}
```

```
void deleteStudent(int rollNumber) {
    ifstream inputFile("student_records.txt");
    ofstream tempFile("temp.txt");
    if (inputFile.is_open() && tempFile.is_open()) {
        string line;
        bool found = false;
        while (getline(inputFile, line)) {
            size_t pos = line.find(',');
            int currentRollNumber = stoi(line.substr(0, pos));
            if (currentRollNumber != rollNumber) {
                tempFile << line << endl;
            } else {
                found = true;
            }
        }
        inputFile.close();
        tempFile.close();
        remove("student_records.txt");
        rename("temp.txt", "student_records.txt");
        if (found) {
            cout << "Student record deleted successfully." << endl;
        } else {
            cout << "Student record not found." << endl;
        }
    } else {
        cout << "Failed to open the file." << endl;
    }
}

void displayStudent(int rollNumber) {
```

```
    ifstream file("student_records.txt");
    if (file.is_open()) {
        string line;
        bool found = false;
        while (getline(file, line)) {
            size_t pos = line.find(',');
            int currentRollNumber = stoi(line.substr(0, pos));
            if (currentRollNumber == rollNumber) {
                found = true;
                cout << "Roll Number: " << currentRollNumber << endl;
                cout << "Name: " << line.substr(pos + 1, line.find(',', pos + 1) - pos - 1) << endl;
                pos = line.find(',', pos + 1);
                cout << "Division: " << line.substr(pos + 1, line.find(',', pos + 1) - pos - 1) << endl;
                pos = line.find(',', pos + 1);
                cout << "Address: " << line.substr(pos + 1) << endl;
                break;
            }
        }
        file.close();
        if (!found) {
            cout << "Student record not found." << endl;
        }
    } else {
        cout << "Failed to open the file." << endl;
    }
}

int main() {
    int choice;
    do {
        cout << "---------- Student Information System ----------" << endl;
```

```
cout << "1. Add Student" << endl;
cout << "2. Delete Student" << endl;
cout << "3. Display Student" << endl;
cout << "4. Exit" << endl;
cout << "Enter your choice: ";
cin >> choice;

switch (choice) {
  case 1: {
    Student student;
    cout << "Enter Roll Number: ";
    cin >> student.rollNumber;
    cin.ignore();
    cout << "Enter Name: ";
    getline(cin, student.name);
    cout << "Enter Division: ";
    getline(cin, student.division);
    cout << "Enter Address: ";
    getline(cin, student.address);
    addStudent(student);
    break;
  }
  case 2: {
    int rollNumber;
    cout << "Enter Roll Number to delete: ";
    cin >> rollNumber;
    deleteStudent(rollNumber);
    break;
  }
  case 3: {
    int rollNumber;
```

```
            cout << "Enter Roll Number to display: ";
            cin >> rollNumber;
            displayStudent(rollNumber);
            break;
          }
          case 4:
            cout << "Exiting the program." << endl;
            break;
          default:
            cout << "Invalid choice. Please try again." << endl;
            break;
        }

        cout << endl;
    } while (choice != 4);

    return 0;
}
```

Output:

```
---------- Student Information System ----------
1. Add Student
2. Delete Student
3. Display Student
4. Exit
Enter your choice: 1
Enter Roll Number: 101
Enter Name: John Doe
Enter Division: A
Enter Address: 123 Main Street
Student record added successfully.
```

---------- Student Information System ----------

1. Add Student

2. Delete Student

3. Display Student

4. Exit

Enter your choice: 1

Enter Roll Number: 102

Enter Name: Jane Smith

Enter Division: B

Enter Address: 456 Park Avenue

Student record added successfully.

---------- Student Information System ----------

1. Add Student

2. Delete Student

3. Display Student

4. Exit

Enter your choice: 3

Enter Roll Number to display: 102

Roll Number: 102

Name: Jane Smith

Division: B

Address: 456 Park Avenue

---------- Student Information System ----------

1. Add Student

2. Delete Student

3. Display Student

4. Exit

Enter your choice: 2

Enter Roll Number to delete: 101

Student record deleted successfully.

Exiting the program.

Name: Abhishek Patil

Div -B   Roll no.84

Practical no : 12

F2: Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.

Program:

```
#include <iostream>

#include <fstream>

#include <string>


using namespace std;


struct Employee {

   int employeeId;

   string name;

   string designation;

   double salary;

};


// Function to add an employee record

void addEmployee() {

   Employee employee;


   cout << "Enter Employee ID: ";

   cin >> employee.employeeId;

   cin.ignore();


   cout << "Enter Employee Name: ";
```

```
    getline(cin, employee.name);

    cout << "Enter Employee Designation: ";
    getline(cin, employee.designation);

    cout << "Enter Employee Salary: ";
    cin >> employee.salary;

    // Open the file in append mode
    ofstream file("employee_data.bin", ios::binary | ios::app);

    // Write the employee record to the file
    file.write(reinterpret_cast<const char*>(&employee), sizeof(Employee));

    file.close();
}

// Function to delete an employee record
void deleteEmployee(int employeeId) {
    // Open the file in read/write mode
    fstream file("employee_data.bin", ios::binary | ios::in | ios::out);

    Employee employee;
    bool found = false;

    // Read records one by one and check for the matching employee ID
    while (file.read(reinterpret_cast<char*>(&employee), sizeof(Employee))) {
        if (employee.employeeId == employeeId) {
            // Mark the record as deleted by setting the employee ID to -1
            employee.employeeId = -1;
            file.seekp(-static_cast<int>(sizeof(Employee)), ios::cur);
```

```
            file.write(reinterpret_cast<const char*>(&employee), sizeof(Employee));
            found = true;
            break;
        }
    }

    file.close();

    if (found) {
        cout << "Employee with ID " << employeeId << " deleted successfully." << endl;
    } else {
        cout << "Employee with ID " << employeeId << " not found." << endl;
    }
}

// Function to display information of a particular employee
void displayEmployee(int employeeId) {
    // Open the file in read mode
    ifstream file("employee_data.bin", ios::binary);

    Employee employee;
    bool found = false;

    // Read records one by one and check for the matching employee ID
    while (file.read(reinterpret_cast<char*>(&employee), sizeof(Employee))) {
        if (employee.employeeId == employeeId) {
            found = true;
            break;
        }
    }
```

```
    file.close();

    if (found) {
        cout << "Employee ID: " << employee.employeeId << endl;
        cout << "Employee Name: " << employee.name << endl;
        cout << "Employee Designation: " << employee.designation << endl;
        cout << "Employee Salary: " << employee.salary << endl;
    } else {
        cout << "Employee with ID " << employeeId << " not found." << endl;
    }
}

int main() {
    int choice;
    int employeeId;

    do {
        cout << "1. Add Employee" << endl;
        cout << "2. Delete Employee" << endl;
        cout << "3. Display Employee" << endl;
        cout << "4. Quit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                addEmployee();
                break;
            case 2:
                cout << "Enter Employee ID to delete: ";
                cin >> employeeId;
```

```
            deleteEmployee(employeeId);
            break;
        case 3:
            cout << "Enter Employee ID to display: ";
            cin >> employeeId;
            displayEmployee(employeeId);
            break;
        case 4:
            cout << "Exiting..." << endl;
            break;
        default:
            cout << "Invalid choice. Please try again." << endl;
      }

      cout << endl;

   } while (choice != 4);

   return 0;
}
```

Output:

```
1. Add Employee
2. Delete Employee
3. Display Employee
4. Quit
Enter your choice: 1
Enter Employee ID: 1001
Enter Employee Name: John Doe
Enter Employee Designation: Manager
```

Enter Employee Salary: 5000

1. Add Employee

2. Delete Employee

3. Display Employee

4. Quit

Enter your choice: 3

Enter Employee ID to display: 1001

Employee ID: 1001

Employee Name: John Doe

Employee Designation: Manager

Employee Salary: 5000

1. Add Employee

2. Delete Employee

3. Display Employee

4. Quit

Enter your choice: 2

Enter Employee ID to delete: 1001

Employee with ID 1001 deleted successfully.

1. Add Employee

2. Delete Employee

3. Display Employee

4. Quit

Enter your choice: 3

Enter Employee ID to display: 1001

Employee with ID 1001 not found.

1. Add Employee

2. Delete Employee

3. Display Employee

4. Quit

Enter your choice: 4

Exiting...