```
/* Assignment 1: Flight management: 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 takes to reach city B
from A, or the amount of fuel used for the journey. Write a menu driven C++
program to represent this as a graph using adjacency matrix and adjacency
list. The node can be represented by the airport name or name of the city.
Check whether cities are connected through flight or not. Compare the storage
representation.*/

#include<iostream>
using namespace std;

class Graph
{
    public:
        int i,j,s1,d1;
        string src,dest;
        int Adjmat[10][10];
        int Time[10][10];
        string city[10];
        int n;

    void data() //member function: Initializes the city names entered by the
user and sets up the adjacency matrix.
    {
        cout<<"Enter the Number of Cities: ";
        cin>>n;

        cout<<"Enter Name of Cities: ";
        for(i=0;i<n;i++)
        {
            cin>>city[i];
        }

        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
            {
                Adjmat[i][j]=0;
                Time[i][j]=0;
            }
        }

    }
    void addflight()
    {

        cout<<"Enter the Starting and Destination City: ";
```

```
        cin>>src>>dest;


        for(i=0;i<n;i++)
        {
            if(city[i]==src)
            {
                s1=i;
            }
        }

        for(i=0;i<n;i++)
        {
            if(city[i]==dest)
            {
                d1=i;
            }
        }

        cout<<"Enter Time: ";
        cin>>Time[s1][d1];
        Adjmat[s1][d1]=1;
        Adjmat[d1][s1]=1;

    }

    void displayArray()   //member function: Displays the flight data using an
adjacency matrix, including city names and flight times.
    {
        cout<<endl;

         for(j=0;j<n;j++)
         {  cout<<"\t"<<city[j];  }

         for(i=0;i<n;i++)
         {
            cout<<"\n "<<city[i];
            for(j=0;j<n;j++)
            {
                cout<<"\t"<<Time[i][j];
            }
                cout<<"\n";
         }
    }


    void displayList() //member function: Placeholder function to display
flight data using an adjacency list.
```

```
    {
        for (i = 0; i < n; i++)
        {
            cout << city[i] << " is connected to: ";

            for (j = 0; j < n; j++)
            {
                if (Adjmat[i][j] == 1)
                {
                    cout << city[j] << " ";
                }
            }

            cout << endl;
        }
    }

};

int main()
{
    Graph g;
    int choice;

    g.data();

    do
    {
        cout<<"\t\t---------------- Flight Management ----------------"<<endl;
        cout<<"\n1) Add Flight \n2) Display using Adjacency Matrix \n3)
Display using Adjacency List \n4)exit"<<endl;
        cout<<"Enter your choice: ";
        cin>>choice;
        switch(choice)
        {
            case 1:
                    g.addflight();
                    break;
            case 2:
                    g.displayArray();
                    break;
            case 3:
                    g.displayList();
                    break;

            case 4: cout<<"Thank you for visit......!!"<<endl;
                    break;
            default:
```

```
                    cout<<"Wrong Choice!! Try Again...";
                    break;
        }

    }while(choice!=4);

    return 0;
}
```

```cpp
/*Assignment 2:Graph traversal: The area around the college and the prominent
landmarks of it are represented using graphs.
Write a menu driven C++ program to represent this as a graph using
adjacency matrix /list and perform DFS and BFS.*/

#include <iostream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <unordered_set>
 #include <stack>
using namespace std;

class Graph {
    unordered_map<string, vector<string>> adj_list;

public:
    void add_edge(string u, string v) {
        adj_list[u].push_back(v);
        adj_list[v].push_back(u); // Assuming an undirected graph
    }

    void bfs(string start) {
        unordered_set<string> visited;
        queue<string> q;


        q.push(start);
        visited.insert(start);

        while (!q.empty()) {
            string node = q.front();
            q.pop();
            cout<<node<<" ";

            for (string neighbor : adj_list[node]) {
                if (visited.find(neighbor) == visited.end()) {
                    q.push(neighbor);
                    visited.insert(neighbor);
                }
            }
        }
        }
        void DFS(string s)
        { unordered_set<string>::iterator itr;
         unordered_set<string> visited;

         stack<string> stack;
```

```
        stack.push(s);

        while(!stack.empty())
        {
         s=stack.top();
         visited.insert(s);

          cout<<endl;
         cout<<s<<" ";
         stack.pop();
         for (string adjacent : adj_list[s]) {
                    if (visited.find(adjacent) == visited.end()) {
                   //cout<<"push"<<adjacent<<endl;

                       stack.push(adjacent);
                   }
               }
         }
        }
};

int main() {
    Graph graph;

    // Adding edges
    graph.add_edge("civil", "cs");
    graph.add_edge("civil", "IT");
    graph.add_edge("civil", "AIDS");
    graph.add_edge("cs", "AIDS");
    graph.add_edge("AIDS", "IT");
    graph.add_edge("cs", "electrical");

    // Performing BFS starting from node 'A'
    graph.bfs("civil");
    cout<<endl;
    graph.DFS("civil");
    // Printing BFS traversal order

    return 0;
}
```

```
/*Activity on vertex(AOV) network: Sandy is a well organized person.
Every day he makes a list of things which need to be done and
enumerates them from 1 to n. However, some things need to be done
before others. Write a C++ code to find out whether Sandy can solve all
his duties and if so, print the correct order*/


/*work to be done: 1. wake up 2. exercise 3. brush 4. take bath 5. worshipping
god
6. Cook Lunch 7. Go to Office 8. have Lunch 9. Come from Office 10.Cook Dinner
11.Have Dinner 12. Go to sleep*/
#include <iostream>
#include <stack>
#include <string>
using namespace std;


class Graph {
    int V;
    int adj[12][12];
    string tasks[12];
    int inDegree[12];

public:
    Graph(int V);
    void addEdge(int v, int w);
    void addTask(int v, string task);
    void topologicalSort();
};

Graph::Graph(int V) {
    this->V = V;
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            adj[i][j] = 0;
        }
        inDegree[i] = 0;
    }
}

void Graph::addTask(int v ,string task) {
    tasks[v] = task;
}

void Graph::addEdge(int v, int w) {
    adj[v][w] = 1;
    inDegree[w]++;
}
```

```
void Graph::topologicalSort() {
    stack<int> stack;

    for (int i = 0; i < V; ++i) {
        if (inDegree[i] == 0)
            stack.push(i);
    }

    int count = 0;
    int order[12];

    while (!stack.empty()) {
        int v = stack.top();
        stack.pop();
        order[count++] = v;

        for (int i = 0; i < V; ++i) {
            if (adj[v][i]) {
                if (--inDegree[i] == 0)
                    stack.push(i);
            }
        }
    }

    if (count != V) {
        cout <<endl<< "Sandy cannot solve all his duties." << endl;
        return;
    }

    cout << "Sandy can solve all his duties. Correct order is:" << endl;
    for (int i = 0; i < count; ++i) {
        cout <<endl<< i + 1 << ". " << tasks[order[i]] << endl;
    }
}

int main() {

    int ch;
    Graph g(5);
    string tasks[5];
    do{
    cout<<"Menu..."<<endl;
    cout<<"1.Enter the Tasks performed by Sandy"<<endl;
    cout<<"2.Display Taks"<<endl;
    cout<<"3.Enter the order of Tasks"<<endl;
    cout<<"4.Sort the Tasks"<<endl;
    cout<<"Enter your choice: "<<endl;
    cin>>ch;
```

```
    switch(ch){
    case 1:
    cout<<"Enter the tasks sandy is going to perform!"<<endl;
    for(int i=0; i<5; i++){
        cout<<"Task "<<i+1<<": ";
        cin>>tasks[i];
        g.addTask(i,tasks[i]);
    }
    cout<<endl<<endl;
    break;

    case 2:
    for(int i=0; i<5; i++){
        cout<<"Task "<<i+1<<": ";
        cout<<tasks[i]<<endl;
    }
    cout<<endl<<endl;
    break;
    case 3:
    cout<<"Add the Graph Edges: "<<endl;
    int v, w;
    for (int i = 0; i < 4; i++) {
        cout << "Enter dependency for task " << i + 1 << ": ";
        cin >> v >> w;
        g.addEdge(v - 1, w - 1); // Adjust indices to start from 0
    }
    break;

    case 4:
    g.topologicalSort();
    break;

    default:
    cout<<"Invalid choice!"<<endl;
    }}while(ch<=4);

    return 0;
}
```

```cpp
/*Assignment 4: Binary search tree: Write a menu driven C++ program to
construct a
binary search tree by inserting the values in the order give, considering at
the beginning with an empty binary search tree, 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. Search a value
v. Print values in ascending and descending order*/


#include<iostream>
#include<algorithm> // for std::max
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
};

class BST {
public:
    Node* root;

    BST() {
        root = nullptr;
    }

    Node* insert(Node* node, int value) { // Insert a new node with the given
value
        if (node == nullptr) {
            node = new Node;
            node->data = value;
            node->left = node->right = nullptr;
        } else if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    int getHeight(Node* node) { // Get the height of the tree
        if (node == nullptr)
            return 0;
        int leftHeight = getHeight(node->left);
        int rightHeight = getHeight(node->right);
```

```
        return 1 + max(leftHeight, rightHeight);
    }

    int findMinValue(Node* node) { // Find the minimum value in the tree
        while (node->left != nullptr) {
            node = node->left;
        }
        return node->data;
    }

    bool searchValue(Node* node, int value) { // Search for a value in the
tree
        if (node == nullptr)
            return false;
        if (node->data == value)
            return true;
        else if (value < node->data)
            return searchValue(node->left, value);
        else
            return searchValue(node->right, value);
    }

    void printAscending(Node* node) { // Print the values in ascending order
        if (node != nullptr) {
            printAscending(node->left);
            cout << node->data << " ";
            printAscending(node->right);
        }
    }

    void printDescending(Node* node) { // Print the values in descending order
        if (node != nullptr) {
            printDescending(node->right);
            cout << node->data << " ";
            printDescending(node->left);
        }
    }

    void insertNode() { // Insert a new node
        int value;
        cout << "Enter the value to insert: ";
        cin >> value;
        root = insert(root, value);
        cout << "Node inserted successfully.\n";
    }

    void findLongestPath() {
        int height = getHeight(root);
```

```
        cout << "Number of nodes in the longest path from the root: " <<
height << endl;
    }

    void findMinValue() {
        if (root == nullptr) {
            cout << "Tree is empty.\n";
        } else {
            int minValue = findMinValue(root);
            cout << "Minimum data value found in the tree: " << minValue <<
endl;
        }
    }

    void searchValue() {
        int value;
        cout << "Enter the value to search: ";
        cin >> value;
        if (searchValue(root, value)) {
            cout << "Value found in the tree.\n";
        } else {
            cout << "Value not found in the tree.\n";
        }
    }

    void printValues() {
        cout << "Ascending Order: ";
        printAscending(root);
        cout << "\nDescending Order: ";
        printDescending(root);
        cout << endl;
    }
};

int main() {
    BST bst;
    int choice;

    do {
        cout << "\nMenu:\n";
        cout << "1. Insert new node\n";
        cout << "2. Find number of nodes in longest path from root\n";
        cout << "3. Minimum data value found in the tree\n";
        cout << "4. Search a value\n";
        cout << "5. Print values in ascending and descending order\n";
        cout << "6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
```

```
        switch (choice) {
            case 1:
                bst.insertNode();
                break;
            case 2:
                bst.findLongestPath();
                break;
            case 3:
                bst.findMinValue();
                break;
            case 4:
                bst.searchValue();
                break;
            case 5:
                bst.printValues();
                break;
            case 6:
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
        }

    } while (choice != 6);

    return 0;
}
```

```
/*Expression tree: Write a menu driven C++ program to construct an
expression tree from the given prefix expression eg. +--a*bc/def and
perform following operations:
1. Traverse it using post order traversal (non recursive)
2. Delete the entire tree
3. Change a tree so that the roles of the left and right pointers are
swapped at every node*/

#include <iostream>
#include <stack>
#include <cstring>
using namespace std;

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

};

Node* create_node(char val){
        Node* new_node = new Node();
        new_node->data = val;
        new_node->left = nullptr;
        new_node->right = nullptr;
        return new_node;
    }

    Node* Expression_tree(char prefix[]){
        stack<Node*> s;
        int i = strlen(prefix)-1;
        while(i>=0){
            char ch = prefix[i];
            if(ch>='a' && ch<='z'){
            Node* new_node = create_node(ch);
            s.push(new_node);
            }
            else{
            Node* new_node = create_node(ch);
            new_node->left = s.top();
            s.pop();
            new_node->right = s.top();
            s.pop();
            s.push(new_node);
            }
            i--;
        }
```

```
        return s.top();
    }

    void postOrderTraversal(Node* root)
    {
        if (root == NULL)
            return;

        // define two stacks - stack1 and stack2
        stack<Node *> stack1, stack2;

        // start from root node and push to first stack
        stack1.push(root);
        Node* node;

        // Repeat untill first stack is not empty
        while (!stack1.empty()) {
            // pop node from stack1 and push to stack2
            node = stack1.top();
            stack1.pop();
            stack2.push(node);

            // push left and right node of popped stack
            if (node->left)
                stack1.push(node->left);
            if (node->right)
                stack1.push(node->right);
        }

        // if stack is empty print the nodes in stack2
        while (!stack2.empty()) {
            node = stack2.top();
            stack2.pop();
            cout << node->data << " ";
        }
    }

int main(){

int ch;
char exp[] = "+--a*bc/def";
Node* root = Expression_tree(exp);

    do{
    cout<<"Enter Your Choice: "<<endl;
    cin>>ch;
        switch(ch){
            case 1:
```

```
            Expression_tree(exp);
            cout<<"Expression Tree created!"<<endl;
            break;

            case 2:
            cout<<"Post Order Traversal: "<<endl;
            postOrderTraversal(root);
            cout<<endl;
            break;

            default:
            cout<<"Invalid Choice"<<endl;
            break;
        }
    }while(true);

}
```

```cpp
/* Assignment 6: A Dictionary using BST: A Dictionary stores key and value
pairs
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)
Write a menu driven C++ program to provide above standard operations
on dictionaries and provide a 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>
#include <string>
using namespace std;

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

class BST {
private:
    Node* root;

    Node* createNode(string key, string value) {
        Node* newNode = new Node();
        if (!newNode) {
            cout << "Memory error\n";
            return NULL;
        }
        newNode->key = key;
        newNode->value = value;
        newNode->left = newNode->right = NULL;
        return newNode;
    }

    Node* insertNode(Node* root, string key, string value) {
        if (root == NULL) {
            root = createNode(key, value);
        } else if (key < root->key) {
            root->left = insertNode(root->left, key, value);
        } else {
            root->right = insertNode(root->right, key, value);
        }
        return root;
    }
```

```
void printInOrder(Node* root) {
    if (root == NULL)
        return;
    printInOrder(root->left);
    cout << root->key << " : " << root->value << "\n";
    printInOrder(root->right);
}

void printReverseOrder(Node* root) {
    if (root == NULL)
        return;
    printReverseOrder(root->right);
    cout << root->key << " : " << root->value << "\n";
    printReverseOrder(root->left);
}

int searchKey(Node* root, string key, int count = 0) {
    if (root == NULL) {
        return -1;
    } else if (root->key == key) {
        return count;
    } else if (key < root->key) {
        return searchKey(root->left, key, count + 1);
    } else {
        return searchKey(root->right, key, count + 1);
    }
}

Node* deleteNode(Node* root, string key) {
    if (root == NULL)
        return root;
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        if (root->left == NULL) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == NULL) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = minValueNode(root->right);
        root->key = temp->key;
```

```
            root->value = temp->value;
            root->right = deleteNode(root->right, temp->key);
        }
        return root;
    }

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

public:
    BST() {
        root = NULL;
    }

    void insert(string key, string value) {
        root = insertNode(root, key, value);
    }

    void remove(string key) {
        root = deleteNode(root, key);
    }

    int search(string key) {
        return searchKey(root, key);
    }

    void printAscending() {
        printInOrder(root);
    }

    void printDescending() {
        printReverseOrder(root);
    }
};

int main() {
    BST bst;
    int choice;
    string key, value;
    while (true) {
        cout << "1. Insert\n"
             << "2. Delete\n"
             << "3. Search\n"
             << "4. Print (ascending)\n"
```

```
             << "5. Print (descending)\n"
             << "6. Exit\n";
        cin >> choice;
        switch (choice) {
        case 1:
            cout << "Enter key and value to insert: ";
            cin >> key >> value;
            bst.insert(key, value);
            break;
        case 2:
            cout << "Enter key to delete: ";
            cin >> key;
            bst.remove(key);
            break;
        case 3:
            cout << "Enter key to search: ";
            cin >> key;
            int result;
            result = bst.search(key);
            if (result != -1)
                cout << "Found with " << result << " comparisons\n";
            else
                cout << "Not found\n";
            break;
        case 4:
            bst.printAscending();
            break;
        case 5:
            bst.printDescending();
            break;
        case 6:
            exit(0);
        default:
            cout << "Invalid choice\n";
        }
    }
    return 0;
}
```

```
/*
Tree using traversal sequence: Write a C++ program to construct the
binary tree with a given preorder and inorder sequence and Test your tree
with all traversals
*/

#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{

   public:
    int findIndex(int inorder[], int val, int size){
        int index = -1;
        for(int i =0; i<size; ++i){
            if(inorder[i]==val){
                return index = i;
            }
        }

            return index;
    }


    Node *buildTree(int inorder[], int preorder[], int startIndex, int
endIndex, int *preIndex, int size){

    if(startIndex>endIndex || (*preIndex) >=size){
        return nullptr;
    }

    int rootIndex = findIndex(inorder, preorder[*preIndex],size);
    Node *root = new Node(inorder[rootIndex]);
```

```
    *preIndex = *preIndex + 1;

    root->left = buildTree(inorder,preorder,startIndex,rootIndex-
1,preIndex,size);
    root->right =
buildTree(inorder,preorder,rootIndex+1,endIndex,preIndex,size);

    return root;

    }

    void inorderTraversal(Node *root){
        if(root == nullptr){
           return;
           }
        inorderTraversal(root->left);
        cout<<root->data<<" ";
        inorderTraversal(root->right);

    }

    void preorderTraversal(Node *root){
        if(root == nullptr){
           return;
           }
        cout<<root->data<<" ";
        preorderTraversal(root->left);
        preorderTraversal(root->right);

    }

    void postorderTraversal(Node *root){
        if(root == nullptr){
           return;
           }
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        cout<<root->data<<" ";

    }



};

int main(){
```

```
  BST b1;
  int size = 6;
  int inorder[] = {4, 2, 5, 1, 6, 3};
  int preorder[] = {1, 2, 4, 5, 3, 6};
  int preindex = 0;
  int ch;
  Node *root = b1.buildTree(inorder,preorder,0, size-1,&preindex,size);

  do{
  cout<<endl<<endl<<"Menu..."<<endl;
  cout<<"1. Inorder Traversal"<<endl;
  cout<<"2. Preorder Traversal"<<endl;
  cout<<"3. Postorder Traversal"<<endl<<endl;
  cout<<"Enter Your Choice:"<<endl;
  cin>>ch;
      switch(ch){
        case 1:
            cout<<"Inorder Traversal: "<<endl;
            b1.inorderTraversal(root);
            break;

        case 2:
            cout<<"Preorder Traversal: "<<endl;
            b1.preorderTraversal(root);
            break;

        case 3:
            cout<<"Postorder Traversal: "<<endl;
            b1.postorderTraversal(root);
            break;

        default:
            cout<<"Invalid Choice!"<<endl;
      }
  }while(ch<=3);

}
```

```
/*A Dictionary using AVL: A Dictionary stores key and value pairs
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)
Write a menu driven C++ program to provide above standard operations
on dictionaries and provide a facility to display whole data sorted in
ascending/ Descending order. Also find how many maximum
comparisons may require for finding any keyword.
Use Height balanced tree(AVL) and find the complexity for finding a
keyword*/

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

class AVLNode {
public:
    string key;
    string value;
    AVLNode* left;
    AVLNode* right;
    int height;

    AVLNode(string k, string v) {
        key = k;
        value = v;
        left = nullptr;
        right = nullptr;
        height = 1;
    }
};

class AVLTree {
private:
    AVLNode* root;

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

    int balanceFactor(AVLNode* node) {
        if (node == nullptr)
            return 0;
        return height(node->left) - height(node->right);
    }
```

```
    AVLNode* rotateRight(AVLNode* y) {
        AVLNode* x = y->left;
        AVLNode* T2 = x->right;

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

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

        return x;
    }

    AVLNode* rotateLeft(AVLNode* x) {
        AVLNode* y = x->right;
        AVLNode* T2 = y->left;

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

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

        return y;
    }

    AVLNode* insert(AVLNode* node, string key, string value) {
        if (node == nullptr)
            return new AVLNode(key, value);

        if (key < node->key)
            node->left = insert(node->left, key, value);
        else if (key > node->key)
            node->right = insert(node->right, key, value);
        else {
            // Duplicate keys not allowed, you can modify as per your
requirement
            return node;
        }

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

        int balance = balanceFactor(node);

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

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

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

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

        return node;
    }

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

    AVLNode* deleteNode(AVLNode* root, string key) {
        if (root == nullptr)
            return root;

        if (key < root->key)
            root->left = deleteNode(root->left, key);
        else if (key > root->key)
            root->right = deleteNode(root->right, key);
        else {
            if (root->left == nullptr || root->right == nullptr) {
                AVLNode* temp = root->left ? root->left : root->right;

                if (temp == nullptr) {
                    temp = root;
                    root = nullptr;
                } else
                    *root = *temp;
                delete temp;
            } else {
                AVLNode* temp = minValueNode(root->right);
                root->key = temp->key;
                root->value = temp->value;
```

```
            root->right = deleteNode(root->right, temp->key);
        }
    }

    if (root == nullptr)
        return root;

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

    int balance = balanceFactor(root);

    // Left Left Case
    if (balance > 1 && balanceFactor(root->left) >= 0)
        return rotateRight(root);

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

    // Right Right Case
    if (balance < -1 && balanceFactor(root->right) <= 0)
        return rotateLeft(root);

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

    return root;
}

void inOrder(AVLNode* root) {
    if (root != nullptr) {
        inOrder(root->left);
        cout << "(" << root->key << ", " << root->value << ") ";
        inOrder(root->right);
    }
}

void reverseInOrder(AVLNode* root) {
    if (root != nullptr) {
        reverseInOrder(root->right);
        cout << "(" << root->key << ", " << root->value << ") ";
        reverseInOrder(root->left);
    }
```

```
    }

    AVLNode* findNode(AVLNode* root, string key, int& comparisons) {
        comparisons++;
        if (root == nullptr || root->key == key)
            return root;

        if (root->key < key)
            return findNode(root->right, key, comparisons);

        return findNode(root->left, key, comparisons);
    }

    int maxComparisons(AVLNode* root, string key) {
        int comparisons = 0;
        findNode(root, key, comparisons);
        return comparisons;
    }

public:
    AVLTree() {
        root = nullptr;
    }

    void insert(string key, string value) {
        root = insert(root, key, value);
    }

    void deleteKey(string key) {
        root = deleteNode(root, key);
    }

    void displayAscending() {
        cout << "Ascending Order: ";
        inOrder(root);
        cout << endl;
    }

    void displayDescending() {
        cout << "Descending Order: ";
        reverseInOrder(root);
        cout << endl;
    }

    int getMaxComparisons(string key) {
        return maxComparisons(root, key);
    }
};
```

```cpp
int main() {
    AVLTree dictionary;
    int choice;
    string key, value;

    do {
        cout << "1. Insert\n";
        cout << "2. Find\n";
        cout << "3. Delete\n";
        cout << "4. Display Ascending\n";
        cout << "5. Display Descending\n";
        cout << "6. Maximum Comparisons\n";
        cout << "7. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter key and value to insert: ";
                cin >> key >> value;
                dictionary.insert(key, value);
                break;
            case 2:
                cout << "Enter key to find: ";
                cin >> key;
                if (dictionary.getMaxComparisons(key) > 0)
                    cout << "Key found\n";
                else
                    cout << "Key not found\n";
                break;
            case 3:
                cout << "Enter key to delete: ";
                cin >> key;
                dictionary.deleteKey(key);
                break;
            case 4:
                dictionary.displayAscending();
                break;
            case 5:
                dictionary.displayDescending();
                break;
            case 6:
                cout << "Enter key to find maximum comparisons: ";
                cin >> key;
                cout << "Maximum comparisons required: " <<
dictionary.getMaxComparisons(key) << endl;
                break;
```

```
            case 7:
                cout << "Exiting program...\n";
                break;
            default:
                cout << "Invalid choice! Please try again.\n";
        }
    } while (choice != 7);

    return 0;
}
```

```
/*
Telephone book management: Consider the telephone book database of
N clients. Write a menu driven C++ program to make use of a hash table
implementation to quickly look up a client's telephone number. Use of
two collision handling techniques and compare them using number of
comparisons required to find a set of telephone numbers
*/

#include <iostream>
using namespace std;

// Store details : Node-> Key Name Telephone
class node{
    private:
        string name;
        string telephone;
        int key;
    public:
        node(){
            key=0;
        }
        friend class hashing;   // To access the private members of class node
};

// Hashng Fuction that generates different key value
// Sum of ascii value of each character in string
int ascii_generator(string s){
    int sum=0;
    for (int i = 0; s[i] != '\0'; i++)
        sum = sum + s[i];
    return sum%100;
}

// Class -> Hashing
class hashing{
    private:
        node data[100]; // Size of directory -> 100
        string n;
        string tele;
        int k, index;
        int size=100;
    public:
        hashing(){
            k=0;
        }

        // Function to create record
        void create_record(string n,string tele){
```

```
        k=ascii_generator(n);   //using ascii value of string as key
        index=k%size;
        for (int j=0;j<size;j++){
            if(data[index].key==0){
                data[index].key=k;
                data[index].name=n;
                data[index].telephone=tele;
                break;
            }
            else
                index=(index+1)%size;
        }
    }

    // Function to search for record based on name input
    void search_record(string name){
        int index1,k,flag=0;
        k=ascii_generator(name);
        index1=k%size;

        for(int a=0;a<size;a++){
            if(data[index1].key==k){
                flag=1;
                cout<<"\nRecord found\n";
                cout<<"Name :: "<<data[index1].name<<endl;
                cout<<"Telephone :: "<<data[index1].telephone<<endl;
                break;
            }
            else
                index1=(index1+1)%size;
        }
        if(flag==0)
            cout<<"Record not found";
    }

    // Function to delete existing record
    void delete_record(string name){
        int index1,key,flag=0;
        key=ascii_generator(name);
        index1=key%size;

        for(int a=0;a<size;a++){
            if(data[index1].key==key){
                flag=1;
                data[index1].key=0;
                data[index1].name=" ";
                data[index1].telephone=" ";
                cout<<"\nRecord Deleted successfully"<<endl;
```

```
                    break;
                }
                else
                    index1=(index1+1)%size;
            }
            if(flag==0)
                cout<<"\nRecord not found";
        }

        // Function to update existing record
        void update_record(string name){
            int index1,key,flag=0;
            key=ascii_generator(name);
            index1=key%size;

            for(int a=0;a<size;a++){
                if(data[index1].key==key){
                    flag=1;
                    break;
                }
                else
                    index1=(index1+1)%size;
            }

            if(flag==1){
                cout<<"Enter the new telephone number :: ";
                cin>>tele;
                data[index1].telephone=tele;
                cout<<"\nRecord Updated successfully";
            }
        }

        // Function to display the directory
        void display_record(){
            cout<<"\t  Name  \t\t Telephone";
            for (int a = 0; a < size; a++) {
                if(data[a].key!=0){
                    cout<<"\n\t"<<data[a].name<<" \t\t\t "<<data[a].telephone;
                }
            }
        }
};

// Main Function
int main(){
    hashing s;
    string name;
    string telephone;
```

```
    int choice,x;
    bool loop=1;
    // Menu driven code
    while(loop){
        cout<<"\n-------------------------"<<endl
            <<" Telephone book Database "<<endl
            <<"-------------------------"<<endl
            <<"1. Create Record"<<endl
            <<"2. Display Record"<<endl
            <<"3. Search Record"<<endl
            <<"4. Update Record"<<endl
            <<"5. Delete Record"<<endl
            <<"6. Exit"<<endl
            <<"Enter choice :: ";
        cin>>choice;
        switch (choice)
        {
        case 1:
            cout<<"\nEnter name :: ";
            cin>>name;
            cout<<"Enter Telephone number :: ";
            cin>>telephone;
            s.create_record(name,telephone);
            break;

        case 2:
            s.display_record();
            break;

        case 3:
            cout<<"\nEnter the name :: ";
            cin>>name;
            s.search_record(name);
            break;

        case 4:
            cout<<"\nEnter the name :: ";
            cin>>name;
            s.update_record(name);
            break;

        case 5:
            cout<<"\nEnter name to Delete :: ";
            cin>>name;
            s.delete_record(name);
            break;

        case 6:
```

```
            loop=0;
            break;

        default:
            cout<<"\nYou Entered something wrong!";
            break;
        }
    }
    return 0;
}
```

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

class Node {
public:
    int key;
    string value;
    Node* next;

    Node(int key, string value) {
        this->key = key;
        this->value = value;
        this->next = nullptr;
    }
};

class SeparateChaining {
public:
    static const int n = 10;
    Node* hashTable[n];

    SeparateChaining() {
        for (int i = 0; i < n; ++i)
            hashTable[i] = nullptr;
    }

    int hashFunction(int key) {
        return key % n;
    }

    void hashing(int key, string value) {
        int index = hashFunction(key);
        Node* newNode = new Node(key, value);

        if (hashTable[index] == nullptr) {
            hashTable[index] = newNode;
        } else {
            newNode->next = hashTable[index];
            hashTable[index] = newNode;
        }
    }

    Node* search(int key) {
        int index = hashFunction(key);
        Node* temp = hashTable[index];

        while (temp != nullptr) {
            if (temp->key == key)
```

```
                return temp;
            temp = temp->next;
        }
        return nullptr;
    }

    void display() {
        for (int i = 0; i < n; ++i) {
            cout << "Hash Table[ " << i << "] -> ";
            Node* temp = hashTable[i];
            while (temp != nullptr) {
                cout << "->(" << temp->key << ", " << temp->value << ")";
                temp = temp->next;
            }
            cout <<"->NULL"<<endl;
        }
    }
};


int main(){
    SeparateChaining sc;
    int ch;
    int key;
    string value;
    Node* found;
    do{
    cout<<"Menu.. "<<endl;
    cout<<"1. Insert"<<endl;
    cout<<"2. Search"<<endl;
    cout<<"3. Display"<<endl;
    cout<<"Enter Your Choice: "<<endl;
    cin>>ch;
    switch(ch){
        case 1:
          cout<<"Enter Key:"<<endl;
          cin>>key;
          cout<<"Enter Value: "<<endl;
          cin>>value;
          sc.hashing(key, value);
         break;
        case 2:
         cout<<"Enter Key to search:"<<endl;
         cin>>key;
         found = sc.search(key);
         if(found!=nullptr){
            cout<<"Key: "<<key<<"Value: "<<found->value<<" found!"<<endl;
         }
         else{
```

```
            cout<<key<<" not found!"<<endl;
         }
         break;
        case 3:
        sc.display();
         break;
        default:
         cout<<"Invalid Choice"<<endl;
    }
    }while(ch<=3);
}
```

```cpp
/*Assignment 11: Sequential File: The students' club members (MemberID, name,
phone,
email) list is to be maintained. The common operations performed
include these: add member, search member, delete member, and update
the information. Write a menu driven C++ program that uses file
operation to implement the same and perform all operations*/

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cstring>

using namespace std;

struct Member {
    int MemberID;
    char name[50];
    char phone[15];
    char email[50];
};

void addMember() {
    ofstream file("members.dat", ios::binary | ios::app);
    Member member;

    cout << "Enter Member ID: ";
    cin >> member.MemberID;
    cout << "Enter Name: ";
    cin.ignore();
    cin.getline(member.name, 50);
    cout << "Enter Phone: ";
    cin.getline(member.phone, 15);
    cout << "Enter Email: ";
    cin.getline(member.email, 50);

    file.write(reinterpret_cast<char*>(&member), sizeof(Member));
    file.close();
    cout << "Member added successfully." << endl;
}

void searchMember() {
    ifstream file("members.dat", ios::binary);
    if (!file) {
        cout << "File not found." << endl;
        return;
    }

    int searchID;
```

```
    cout << "Enter Member ID to search: ";
    cin >> searchID;

    Member member;
    bool found = false;
    while (file.read(reinterpret_cast<char*>(&member), sizeof(Member))) {
        if (member.MemberID == searchID) {
            found = true;
            cout << "Member found:" << endl;
            cout << "Member ID: " << member.MemberID << endl;
            cout << "Name: " << member.name << endl;
            cout << "Phone: " << member.phone << endl;
            cout << "Email: " << member.email << endl;
            break;
        }
    }

    if (!found) {
        cout << "Member not found." << endl;
    }

    file.close();
}

void deleteMember() {
    ifstream inFile("members.txt", ios::binary);
    ofstream outFile("temp.txt", ios::binary);

    int deleteID;
    cout << "Enter Member ID to delete: ";
    cin >> deleteID;

    Member member;
    bool found = false;
    while (inFile.read(reinterpret_cast<char*>(&member), sizeof(Member))) {
        if (member.MemberID != deleteID) {
            outFile.write(reinterpret_cast<char*>(&member), sizeof(Member));
        } else {
            found = true;
        }
    }

    inFile.close();
    outFile.close();

    remove("members.dat");
    rename("temp.dat", "members.dat");
```

```
    if (found) {
        cout << "Member deleted successfully." << endl;
    } else {
        cout << "Member not found." << endl;
    }
}

void updateMember() {
    fstream file("members.dat", ios::binary | ios::in | ios::out);

    int updateID;
    cout << "Enter Member ID to update: ";
    cin >> updateID;

    Member member;
    bool found = false;
    while (file.read(reinterpret_cast<char*>(&member), sizeof(Member))) {
        if (member.MemberID == updateID) {
            found = true;

            cout << "Enter updated Name: ";
            cin.ignore();
            cin.getline(member.name, 50);
            cout << "Enter updated Phone: ";
            cin.getline(member.phone, 15);
            cout << "Enter updated Email: ";
            cin.getline(member.email, 50);

            file.seekp(file.tellg() - sizeof(Member));
            file.write(reinterpret_cast<char*>(&member), sizeof(Member));

            cout << "Member updated successfully." << endl;
            break;
        }
    }

    if (!found) {
        cout << "Member not found." << endl;
    }

    file.close();
}

int main() {
    char choice;
    do {
        cout << "\n1. Add Member\n";
        cout << "2. Search Member\n";
```

```
        cout << "3. Delete Member\n";
        cout << "4. Update Member\n";
        cout << "5. Exit\n";
        cout << "-------------------------------------\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch(choice) {
            case '1':
                addMember();
                break;
            case '2':
                searchMember();
                break;
            case '3':
                deleteMember();
                break;
            case '4':
                updateMember();
                break;
            case '5':
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
        }
    } while (choice != '5');

    return 0;
}
```

```
/*Assignment 12: Min/max Heaps: Marks obtained by students of second year in
an
online examination of a particular subject are stored by the teacher.
Teacher wants to find the minimum and maximum marks of the subject.
Write a menu driven C++ program to find out maximum and minimum
marks obtained in that subject using heap data structure. Analyze the
algorithm*/


#include <iostream>
#include <algorithm> // Include the <algorithm> header to access the std::swap
function
#include <climits>
using namespace std;

void buildMaxHeap(int arr[], int n) {
    int i;
    for (i = (n / 2) - 1; i >= 0; i--) {
        int largest = i; // Declare and initialize the variable largest with
the value of i
        int left = 2 * i + 1; // Initialize the variable left
        int right = 2 * i + 2; // Initialize the variable right
        if (left < n && arr[left] > arr[largest])
            largest = left;
        if (right < n && arr[right] > arr[largest])
            largest = right;
        if (largest != i) {
            std::swap(arr[i], arr[largest]); // Use std::swap instead of swap
            buildMaxHeap(arr, n);
        }
    }
}

void buildMinHeap(int arr[], int n) {
    int i;
    for (i = (n / 2) - 1; i >= 0; i--) {
        int smallest = i;
        int left = 2 * i + 1; // Specify the namespace for the "left" variable
        int right = 2 * i + 2;
        if (left < n && arr[left] < arr[smallest]) // Specify the namespace
for the "left" variable
            smallest = left; // Specify the namespace for the "left" variable
        if (right < n && arr[right] < arr[smallest])
            smallest = right;
        if (smallest != i) {
            std::swap(arr[i], arr[smallest]);
            buildMaxHeap(arr, n);
        }
    }
```

```
}

int findMax(int arr[], int n) {
    if (n == 0)
        return INT_MIN;
    buildMaxHeap(arr, n);
    return arr[0];
}

int findMin(int arr[], int n) {
    if (n == 0)
        return INT_MAX;
    buildMinHeap(arr, n);
    return arr[0];
}

int main() {
    int max_size = 100; // Declare and initialize the variable max_size with
the desired maximum size value
    int a[max_size];
    int size, choice;

    cout << "Enter the size of array:";
    cin >> size;

    int i; // Declare the variable "i"
    if (size <= 0 || size > max_size) {
        cout << "Invalid Size";
        return 1;
    }

    cout << "marks:";
    for (i = 0; i < size; i++) {
        cin >> a[i]; // Input marks into array 'a'
    }

    cout << "menu:";
    cout << "1.Max marks:";
    cout << "2.Min marks:";
    cout << "3.Exiting";

    do {
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
```

```
                cout << "Maximum marks: " << findMax(a, size) << endl; // Pass
the array 'a' instead of 'marks'
                break;
            case 2:
                cout << "Minimum marks: " << findMin(a, size) << endl; // Pass
the array 'a' instead of 'marks'
                break;
            case 3:
                cout << "Exiting" << endl;
                break;
            default:
                cout << "Invalid choice" << endl;
                break;
        }
    } while (choice != 3);

    return 0;
}
```

```cpp
/*Assignment 13: A Dictionary using STL map and Hashmap: Implement Dictionary
(key and value pairs) using using STL map in C++ and Hashmap in
Java and compare all dictionary implementation
1. BST
2. AVL
3. User defined Hash table
4. STL Map
5. Hashmap in Java
Use Visual C++*/

#include <iostream>
#include <map>
#include <string>

using namespace std;

void searchDictionary(const map<string, string>& dict, const string& key) {
    auto it = dict.find(key);
    if (it != dict.end()) {
        cout << "Meaning of '" << key << "': " << it->second << endl;
    } else {
        cout << "Key not found in dictionary." << endl;
    }
}

void displayDictionary(const map<string, string>& dict) {
    if (dict.empty()) {
        cout << "Dictionary is empty." << endl;
        return;
    }

    cout << "Existing Dictionary:" << endl;
    for (const auto& pair : dict) {
        cout << "Meaning of '" << pair.first << "': " << pair.second << endl;
    }
}

int main() {
    map<string, string> dict;

    int choice;

    do {
        cout << "\n---------- Dictionary Management System ----------\n";
        cout << "1. Add Word\n";
        cout << "2. Search Word\n";
        cout << "3. Display Dictionary\n";
        cout << "4. Exit\n";
```

```
        cout << "--------------------------------------------------\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: {
                string key, meaning;
                cout << "Enter a word to add: ";
                cin >> key;
                cout << "Enter the meaning of '" << key << "': ";
                cin.ignore();
                getline(cin, meaning);
                dict[key] = meaning;
                cout << "Word added successfully!" << endl;
                break;
            }
            case 2: {
                string key;
                cout << "Enter the word to search: ";
                cin >> key;
                searchDictionary(dict, key);
                break;
            }
            case 3: {
                displayDictionary(dict);
                break;
            }
            case 4: {
                cout << "Exiting..." << endl;
                break;
            }
            default: {
                cout << "Invalid choice!" << endl;
            }
        }
    } while (choice != 4);

    return 0;
}
```