Source Code:

```
# Assignment No. 1
class Telebook:
    def __init__(self):
        self.name = ""
        self.tel = 0

class Hashing:
    def __init__(self):
        self.data = [Telebook() for _ in range(100)]
        self.nm = ""
        self.t = 0
        self.index = 0
        self.size = 100

    def create_record(self):
        self.nm = input("\nEnter name of customer: ")
        self.t = int(input("Enter telephone number (10-digit): "))
        self.index = self.t % self.size

        for _ in range(self.size):
            if self.data[self.index].tel == 0:
                self.data[self.index].name = self.nm
                self.data[self.index].tel = self.t
                break
            else:
                self.index = (self.index + 1) % self.size

    def search_record(self):
        index1 = 0
        flag = 0
        key = int(input("\nTelephone number to search: "))
        index1 = key % self.size

        for _ in range(self.size):
            if self.data[index1].tel == key:
                flag = 1
                print("\nRecord found:")
                print("\tNAME \t\tTELEPHONE")
                print(f"{self.data[index1].name} \t{self.data[index1].tel}")
                break
            else:
                index1 = (index1 + 1) % self.size

        if flag == 0:
            print("\n........Record not found")

    def display_record(self):
        print("\n\tNAME \t\tTELEPHONE")

        for a in range(self.size):
            if self.data[a].tel != 0:
                print(f"\t{self.data[a].name} \t{self.data[a].tel}")



if __name__ == "__main__":
    s = Hashing()
    cho = 0

    while cho < 4:
        print("\nEnter 1: Create new Client record")
        print("Enter 2: Display all record")
        print("Enter 3: Search")
        print("Enter 4: Exit")
```

```
cho = int(input("Enter your choice: "))

if cho == 1:
    print("\n1. CREATE Record")
    s.create_record()
elif cho == 2:
    print("\n\n\n\n2. DISPLAY Record")
    s.display_record()
elif cho == 3:
    print("\n\n\n\n3. SEARCH Record")
    s.search_record()
```

Source Code:

```
# Assignment No. 2
class SetADT:
    def __init__(self):
        self.elements = []

    def add(self, element):
        if element not in self.elements:
            self.elements.append(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):
        result = SetADT()
        for element in self.elements:
            if other_set.contains(element):
                result.add(element)
        return result

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

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

    def is_subset(self, other_set):
        for element in self.elements:
            if not other_set.contains(element):
                return False
        return True


# Function to get user input for a set
def get_set_input():
    set_input = input("Enter the elements of the set (space-separated): ")
    elements = set_input.split()
    return elements


# Create set 1
print("Enter the elements of Set 1:")
set1 = SetADT()
elements = get_set_input()
for element in elements:
    set1.add(element)

# Create set 2
print("Enter the elements of Set 2:")
set2 = SetADT()
elements = get_set_input()
for element in elements:
    set2.add(element)

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

while True:
    print("\n--- Set Operations ---")
    print("1. Intersection")
    print("2. Union")
    print("3. Difference (Set 1 - Set 2)")
    print("4. Difference (Set 2 - Set 1)")
    print("5. Check if Set 2 is a subset of Set 1")
    print("6. Remove an element from Set 1")
    print("7. Remove an element from Set 2")
    print("8. Check if an element is in Set 1")
    print("9. Check if an element is in Set 2")
    print("10. Get the size of Set 1")
    print("11. Get the size of Set 2")
    print("12. Exit")

    choice = input("Enter your choice (1-12): ")
    if choice == '1':
        intersection_set = set1.intersection(set2)
        print("Intersection:", list(intersection_set.iterator()))
    elif choice == '2':
```

```
    union_set = set1.union(set2)
    print("Union:", list(union_set.iterator()))
elif choice == '3':
    difference_set = set1.difference(set2)
    print("Difference (Set 1 - Set 2):", list(difference_set.iterator()))
elif choice == '4':
    difference_set = set2.difference(set1)
    print("Difference (Set 2 - Set 1):", list(difference_set.iterator()))
elif choice == '5':
    is_subset = set2.is_subset(set1)
    if is_subset:
        print("Set 2 is a subset of Set 1")
    else:
        print("Set 2 is not a subset of Set 1")
elif choice == '6':
    element = input("Enter the element to remove from Set 1: ")
    set1.remove(element)
    print("Element removed from Set 1.")
elif choice == '7':
    element = input("Enter the element to remove from Set 2: ")
    set2.remove(element)
    print("Element removed from Set 2.")
elif choice == '8':
    element = input("Enter the element to check in Set 1: ")
    if set1.contains(element):
        print("Element is in Set 1")
    else:
        print("Element is not in Set 1")
elif choice == '9':
    element = input("Enter the element to check in Set 2: ")
    if set2.contains(element):
        print("Element is in Set 2")
    else:
        print("Element is not in Set 2")
elif choice == '10':
    print("Size of Set 1:", set1.size())
elif choice == '11':
    print("Size of Set 2:", set2.size())
elif choice == '12':
    print("Exiting...")
    break
else:
    print("Invalid choice. Please enter a valid option.")
```

Source Code:

```
//Assignment No. 3
#include <iostream>
using namespace std;
class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};
class BinarySearchTree {
private:
    TreeNode* root;

    TreeNode* insertNode(TreeNode* root, int value) {
        if (root == nullptr) {
            root = new TreeNode(value);
        }
        else if (value < root->data) {
            root->left = insertNode(root->left, value);
        }
        else {
            root->right = insertNode(root->right, value);
        }
        return root;
    }
```

```
int getLongestPath(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
    int leftPath = getLongestPath(root->left);
    int rightPath = getLongestPath(root->right);
    return 1 + max(leftPath, rightPath);
}

int findMinValue(TreeNode* root) {
    if (root == nullptr) {
        cout << "Tree is empty." << endl;
        return -1;
    }
    if (root->left == nullptr) {
        return root->data;
    }
    return findMinValue(root->left);
}

TreeNode* swapTreeNodes(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }
    TreeNode* temp = root->left;
    root->left = swapTreeNodes(root->right);
    root->right = swapTreeNodes(temp);
    return root;
}

TreeNode* searchValue(TreeNode* root, int value) {
        if (root == nullptr || root->data == value) {
            return root;
        }
        if (value < root->data) {
            return searchValue(root->left, value);
        }
        return searchValue(root->right, value);
    }

public:
    BinarySearchTree() {
        root = nullptr;
    }

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

    int longestPath() {
        return getLongestPath(root);
    }

    int minValue() {
        return findMinValue(root);
    }

    void swapNodes() {
        root = swapTreeNodes(root);
    }

    bool search(int value) {
        TreeNode* result = searchValue(root, value);
        return (result != nullptr);
    }
};

int main() {
    BinarySearchTree bst;

    // Construct binary search tree by inserting values
    int values[] = { 5, 3, 8, 2, 4, 7, 9 };
    int size = sizeof(values) / sizeof(values[0]);
    for (int i = 0; i < size; i++) {
        bst.insert(values[i]);
    }

    int choice;
    do {
        cout << "|-------------------|" << endl;
        cout << "| Menu              |" << endl;
        cout << "| 1. Insert         |" << endl;
        cout << "| 2. Longest Path   |" << endl;
        cout << "| 3. Minimum Value  |" << endl;
        cout << "| 4. Swap Nodes     |" << endl;
        cout << "| 5. Search         |" << endl;
        cout << "| 6. Exit           |" << endl;
        cout << "|-------------------|" << endl;

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

        switch (choice) {
            case 1: {
                int value;
            cout << "Enter a value to insert: ";
            cin >> value;
            bst.insert(value);
            cout << "Value inserted." << endl;
            break;
        }
        case 2: {
            int longestPath = bst.longestPath();
            cout << "Number of nodes in the longest path from root: " << longestPath << endl;
            break;
        }
        case 3: {
            int minValue = bst.minValue();
            cout << "Minimum value found in the tree: " << minValue << endl;
            break;
        }
        case 4: {
            bst.swapNodes();
            cout << "Tree nodes swapped." << endl;
            break;
        }
        case 5: {
            int value;
            cout << "Enter a value to search: ";
            cin >> value;
            bool isFound = bst.search(value);
            if (isFound) {
                cout << value << " found in the tree." << endl;
            }
            else {
                cout << value << " not found in the tree." << endl;
            }
```

```
                break;
            }
            case 6: {
                cout << "Exiting program..." << endl;
                break;
            }
            default: {
                cout << "Invalid choice. Please try again." << endl;
                break;
            }
        }
    } while (choice != 6);

    return 0;
}
```

Source Code:

```
// Assignment No. 4
#include <iostream>
using namespace std;

// Structure for a threaded binary tree node
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    bool isThreaded;

    TreeNode(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
        isThreaded = false;
    }
};

// Class for threaded binary tree
class ThreadedBinaryTree {
private:
    TreeNode* root;

    // Helper function to perform Morris Inorder Traversal
    void threadedInorderUtil(TreeNode* node, TreeNode*& prev) {
        if (node == nullptr)
            return;

        // Traverse left subtree
        threadedInorderUtil(node->left, prev);
        // Process current node
        if (node->left == nullptr) {
            node->left = prev;
            node->isThreaded = true;
        }

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

        prev = node;

        // Traverse right subtree
        threadedInorderUtil(node->right, prev);
    }

public:
    ThreadedBinaryTree() {
        root = nullptr;
    }

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

    TreeNode* insertNode(TreeNode* node, int value) {
        if (node == nullptr)
            return new TreeNode(value);

        if (value < node->data)
            node->left = insertNode(node->left, value);
        else
            node->right = insertNode(node->right, value);

        return node;
    }

    void convertToThreaded() {
        TreeNode* prev = nullptr;
        threadedInorderUtil(root, prev);
    }

    void inorderTraversal() {
        if (root == nullptr) {
            cout << "Tree is empty." << endl;
            return;
        }

        TreeNode* current = leftMost(root);
        while (current != nullptr) {
            // Process the node
            cout << current->data << " ";

            // If the right child is a thread, go to its inorder successor
            if (current->isThreaded) {
                current = current->right;
            }
            else {
                // Otherwise, go to the leftmost node of the right subtree
                current = leftMost(current->right);
            }
        }
```

```
        cout << endl;
    }

    TreeNode* leftMost(TreeNode* node) {
        if (node == nullptr)
            return nullptr;

        while (node->left != nullptr)
            node = node->left;

        return node;
    }
};

int main() {
    ThreadedBinaryTree tbtree;

    int choice;
    do {
        cout << "|-------------------|" << endl;
        cout << "| Menu              |" << endl;
        cout << "| 1. Insert Value    |" << endl;
        cout << "| 2. Inorder Traversal |" << endl;
        cout << "| 3. Convert to Threaded |" << endl;
        cout << "| 4. Exit           |" << endl;
        cout << "|-------------------|" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1: {
            int value;
            cout << "Enter a value to insert: ";
            cin >> value;
            tbtree.insert(value);
            cout << "Value inserted successfully." << endl;
            break;
        }
        case 2:
            cout << "Inorder Traversal: ";
            tbtree.inorderTraversal();
            break;
        case 3:
            tbtree.convertToThreaded();
            cout << "Binary tree converted to threaded binary tree." << endl;
            break;
        case 4:
            cout << "Exiting program..." << endl;
            break;
        default:
            cout << "Invalid choice. Please try again." << endl;
            break;
        }
    } while (choice != 4);

    return 0;
}
```

Source Code:

```
// Assignment No. 5
#include <iostream>

#include <string>

using namespace std;

// Node structure for BST
struct Node {
  string keyword;
  string meaning;
  Node * left;
  Node * right;
};

// Function to create a new node
Node * createNode(string keyword, string meaning) {
  Node * newNode = new Node;
  newNode -> keyword = keyword;
  newNode -> meaning = meaning;
  newNode -> left = nullptr;
  newNode -> right = nullptr;
  return newNode;
}

// Function to insert a node into BST
Node * insertNode(Node * root, string keyword, string meaning) {
  if (root == nullptr) {
    root = createNode(keyword, meaning);
    return root;
  }
  if (keyword < root -> keyword)
    root -> left = insertNode(root -> left, keyword, meaning);
  else if (keyword > root -> keyword)
    root -> right = insertNode(root -> right, keyword, meaning);
  else {
    // Keyword already exists, update the meaning
    root -> meaning = meaning;
  }

  return root;
}

// Function to search for a keyword in BST
Node * searchKeyword(Node * root, string keyword) {
  if (root == nullptr || root -> keyword == keyword)
    return root;

  if (keyword < root -> keyword)
    return searchKeyword(root -> left, keyword);

  return searchKeyword(root -> right, keyword);
}

// Function to find the node with the minimum value in BST
Node * findMinNode(Node * node) {
  Node * current = node;
  while (current && current -> left != nullptr)
    current = current -> left;
  return current;
}
```

```
// Function to delete a node from BST
Node * deleteNode(Node * root, string keyword) {
 if (root == nullptr)
  return root;

 if (keyword < root -> keyword)
  root -> left = deleteNode(root -> left, keyword);
 else if (keyword > root -> keyword)
  root -> right = deleteNode(root -> right, keyword);
 else {
  // Keyword found, delete the node
  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 has two children, find the inorder successor
  Node * temp = findMinNode(root -> right);

  // Copy the inorder successor's data to the current node
  root -> keyword = temp -> keyword;
  root -> meaning = temp -> meaning;

  // Delete the inorder successor
  root -> right = deleteNode(root -> right, temp -> keyword);
 }
```

```
  return root;
}

// Function to perform inorder traversal and display the BST
void inorderTraversal(Node * root) {
  if (root == nullptr)
    return;

  inorderTraversal(root -> left);
  cout << root -> keyword << ": " << root -> meaning << endl;
  inorderTraversal(root -> right);
}

// Function to perform reverse inorder traversal and display the BST
void reverseInorderTraversal(Node * root) {
  if (root == nullptr)
    return;

  reverseInorderTraversal(root -> right);
  cout << root -> keyword << ": " << root -> meaning << endl;
  reverseInorderTraversal(root -> left);
}

int main() {
  Node * root = nullptr;

  int choice;
  string keyword, meaning;

  do {
    cout << "|----------------------------|" << endl;
    cout << "|          Menu           |" << endl;
    cout << "|---------------------------|" << endl;
    cout << "| 1. Add keyword and meaning|" << endl;
    cout << "| 2. Delete keyword         |" << endl;
    cout << "| 3. Update meaning         |" << endl;
    cout << "| 4. Display in ascending order  |" << endl;
    cout << "| 5. Display in descending order |" << endl;
    cout << "| 6. Search keyword         |" << endl;
    cout << "| 7. Exit                   |" << endl;
    cout << "|---------------------------|" << endl;

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

    switch (choice) {
    case 1:
      cout << "Enter keyword: ";
      cin >> keyword;
      cout << "Enter meaning: ";
      cin.ignore();
      getline(cin, meaning);
      root = insertNode(root, keyword, meaning);
      cout << "Keyword and meaning added to the BST." << endl;
      break;
    case 2:
      cout << "Enter keyword to delete: ";
      cin >> keyword;
      root = deleteNode(root, keyword);
      cout << "Keyword deleted from the BST." << endl;
      break;
    case 3:
      cout << "Enter keyword to update: ";
      cin >> keyword;
      cout << "Enter new meaning: ";
      cin.ignore();
      getline(cin, meaning); {
        Node * searchNode = searchKeyword(root, keyword);
        if (searchNode != nullptr) {
          searchNode -> meaning = meaning;
          cout << "Meaning updated successfully." << endl;
        } else {
          cout << "Keyword not found in the BST." << endl;
        }
      }
      break;
    case 4:
      cout << "BST in ascending order (keyword: meaning):" << endl;
      inorderTraversal(root);
      break;
    case 5:
      cout << "BST in descending order (keyword: meaning):" << endl;
      reverseInorderTraversal(root);
      break;
    case 6:
      cout << "Enter keyword to search: ";
      cin >> keyword; {
        Node * searchResult = searchKeyword(root, keyword);
        if (searchResult != nullptr) {
          cout << "Keyword found! Meaning: " << searchResult -> meaning << endl;
        } else {
          cout << "Keyword not found in the BST." << endl;
        }
      }
      break;
    case 7:
            cout << "Exiting program..." << endl;
            break;
          default:
            cout << "Invalid choice. Please try again." << endl;
            break;
          }
        } while (choice != 7);

        return 0;
    }
```

Source Code:

```
// Assignment No. 6
#include <iostream>
#include <vector>
#include <list>

using namespace std;

// Structure to represent an edge between cities
struct Edge {
    int destination;  // Index of the destination city
    int cost;         // Cost of the edge (time or fuel)
};

// Graph class
class Graph {
private:
    int numCities;                  // Number of cities
    vector<string> cityNames;       // Names of the cities
    vector<list<Edge>> adjacencyList; // Adjacency list representation

public:
    // Constructor
    Graph(int cities) {
        numCities = cities;
        adjacencyList.resize(cities + 1);
        cityNames.resize(cities + 1);
    }

    // Function to add an edge between cities
    void addEdge(int source, int destination, int cost) {
        Edge edge;
    edge.destination = destination;
    edge.cost = cost;

    adjacencyList[source].push_back(edge);
}

// Function to set the name of a city
void setCityName(int index, const string& name) {
    if (index >= 0 && index <= numCities) {
        cityNames[index] = name;
    }
}

// Function to check if the graph is connected
bool isConnected() {
    vector<bool> visited(numCities + 1, false);
    dfs(1, visited); // Start from vertex 1

    for (int i = 1; i <= numCities; i++) {
        if (!visited[i])
            return false;
    }

    return true;
}

// Depth-first search traversal
void dfs(int vertex, vector<bool>& visited) {
    visited[vertex] = true;

    for (const Edge& edge : adjacencyList[vertex]) {
        if (!visited[edge.destination]) {
                dfs(edge.destination, visited);
            }
        }
    }
};

int main() {
    int numCities, numEdges;
    cout << "Enter the number of cities: ";
    cin >> numCities;
    cout << "Enter the number of edges: ";
    cin >> numEdges;

    Graph graph(numCities);

    for (int i = 1; i <= numCities; i++) {
        string cityName;
        cout << "Enter the name of city " << i << ": ";
        cin >> cityName;
        graph.setCityName(i, cityName);
    }

    for (int i = 0; i < numEdges; i++) {
        int source, destination, cost;
        cout << "Enter the source city index, destination city index, and cost: ";
        cin >> source >> destination >> cost;
        graph.addEdge(source, destination, cost);
    }

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

    return 0;
}
```

Source Code:

```
// Assignment No. 7
#include<iostream>

using namespace std;
int main() {
 int n, i, j, k, row, col, mincost = 0, min;
 char op;
 cout << "Enter no. of vertices: ";
 cin >> n;
 int cost[n][n];
 int visit[n];
 for (i = 0; i < n; i++)
  visit[i] = 0;
 for (i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
   cost[i][j] = -1;
 for (i = 0; i < n; i++) {
  for (j = i + 1; j < n; j++) {
   cout << "Do you want an edge between " << i + 1 << " and "<<j+1<<"(y-yes, n-no): ";
   //use 'i' & 'j' if your vertices start from 0
   cin >> op;
   if (op == 'y' || op == 'Y') {
    cout << "Enter weight: ";
    cin >> cost[i][j];
    cost[j][i] = cost[i][j];
   }
  }
 }
 visit[0] = 1;
 for (k = 0; k < n - 1; k++) {
  min = 999;
  for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
    if (visit[i] == 1 && visit[j] == 0) {
     if (cost[i][j] != -1 && min > cost[i][j]) {
      min = cost[i][j];
      row = i;
      col = j;
     }
    }
   }
  }
  mincost += min;
  visit[col] = 1;
  cost[row][col] = cost[col][row] = -1;
  cout << row + 1 << "->" << col + 1 << endl;
  //use 'row' & 'col' if your vertices start from 0
 }
 cout << "\nMin. Cost: " << mincost;
 return 0;
}
```

Source Code:

```
// Assignment No. 8
#include<iostream>
#include<string>
using namespace std;
class dictionary;
class avlnode
{
        string keyword;
        string meaning;
        avlnode *left,*right;
        int bf;
        public:
        avlnode()
        {
                keyword='\0';
                meaning='\0';
                left=right=NULL;
                bf=0;
        }
        avlnode(string k,string m)
        {
                keyword=k;
                meaning=m;
                left=right=NULL;
                bf=0;
        }
friend class dictionary;
};

class dictionary
{
        avlnode *par,*loc;
        public:
        avlnode *root;
        dictionary()
        {
                root=NULL;
                par=loc=NULL;
        }
        void accept();
        void insert(string key,string mean);
        void LLrotation(avlnode*,avlnode*);

        void RRrotation(avlnode*,avlnode*);

        void inorder(avlnode *root);
        void deletekey(string key);
        void descending(avlnode *);
        void search(string);
        void update(string,string);

};
void dictionary::descending(avlnode *root)
{
        if(root)
        {
                descending(root->right);
                cout<<root->keyword<<" "<<root->meaning<<endl;
                descending(root->left);
        }
}

void dictionary::accept()
```

```
{
    string key,mean;
    cout<<"Enter keyword "<<endl;
    cin>>key;
    cout<<"Enter meaning "<<endl;
    cin>>mean;
    insert(key,mean);
}
void dictionary::LLrotation(avlnode *a,avlnode *b)
{
    cout<<"LL rotation"<<endl;
    a->left=b->right;
    b->right=a;
    a->bf=b->bf=0;
}

void dictionary::RRrotation(avlnode *a,avlnode *b)
{
    cout<<"RR rotation"<<endl;
    a->right=b->left;
    b->left=a;
    a->bf=b->bf=0;
}
void dictionary::insert(string key,string mean)
{
    //cout<<"IN Insert \n";
    if(!root)
    {
        //create new root
        root=new avlnode(key,mean);
        cout<<"ROOT CREATED \n";
        return;
    }
//  else
//  {
        avlnode *a,*pa,*p,*pp;
        //a=NULL;
        pa=NULL;
        p=a=root;
        pp=NULL;

        while(p)
        {
            cout<<"In first while \n";
            if(p->bf)
            {
            a=p;
            pa=pp;
            }
            if(key<p->keyword){pp=p;p=p->left;}   //takes the left branch
            else if(key>p->keyword){pp=p;p=p->right;} //right branch
            else
            {
                //p->meaning=mean;
                cout<<"Already exist \n";
                return;
            }
        }
        cout<<"Outside while \n";
        avlnode *y=new avlnode(key,mean);
        if(key<pp->keyword)
        {
            pp->left=y;
        }
```

```
else
        pp->right=y;
cout<<"KEY INSERTED \n";

int d;
avlnode *b,*c;
//a=pp;
b=c=NULL;
if(key>a->keyword)
{
        cout<<"KEY >A->KEYWORD \n";
        b=p=a->right;
        d=-1;
        cout<<" RIGHT HEAVY \n";
}
else
{
        cout<<"KEY < A->KEYWORD \n";
        b=p=a->left;
        d=1;
        cout<<" LEFT HEAVY \n";
}

while(p!=y)
{
        if(key>p->keyword)
        {
                p->bf=-1;
                p=p->right;

        }
        else
```

```
		{
			p->bf=1;
			p=p->left;
		}

	}
	cout<<" DONE ADJUSTING INTERMEDIATE NODES \n";
	if(!(a->bf)||!(a->bf+d))
	{
		a->bf+=d;
		return;
	}
	//else
	//{
	if(d==1)
	{
		//left heavy
		if(b->bf==1)
		{
			LLrotation(a,b);
			/*a->left=b->right;
			b->right=a;
			a->bf=0;
			b->bf=0;*/
		}
		else //if(b->bf==-1)
		{

			cout<<"LR rotation"<<endl;
			c=b->right;
			b->right=c->left;
			a->left=c->right;
```

```
          c->left=b;
          c->right=a;
          switch(c->bf)
          {
                    case 1:
                    {
                              a->bf=-1;
                              b->bf=0;
                              break;
                    }
                    case -1:
                    {
                              a->bf=0;
                              b->bf=1;
                              break;
                    }

                    case 0:
                    {
                              a->bf=0;
                              b->bf=0;
                              break;
                    }

          }
          c->bf=0;
          b=c;   //b is new root


}
//else
//        cout<<"Balanced \n";
```

```
}

if(d==-1)
{
    if(b->bf==-1)
    {
//          cout<<"RR rotation"<<endl;
        /*a->right=b->left;
        b->left=a;
        a->bf=b->bf=0;*/
        RRrotation(a,b);
    }
    else// if(b->bf==1)
    {
        c=b->left;
//          cout<<"RL rotation"<<endl;
        a->right=c->left;
        b->left=c->right;
        c->left=a;
        c->right=b;
        switch(c->bf)
        {
            case 1:
            {
                a->bf=0;
                b->bf=-1;
                break;
            }
            case -1:
            {
                a->bf=1;
                b->bf=0;
                break;
            }

            case 0:
            {
                a->bf=0;
                b->bf=0;
                break;
            }

        }
        c->bf=0;
        b=c;  //b is new root

    }

    //else
        //cout<<"Balanced \n";
}
//}
if(!pa)
    root=b;
else if(a==pa->left)
    pa->left=b;
else
    pa->right=b;
cout<<"AVL tree created!! \n";
//cout<<"AVL \n";
//inorder(root);
```

```
}
void dictionary::search(string key)
{
        cout<<"ENTER SEARCH \n";
        loc=NULL;
        par=NULL;
        if(root==NULL)
        {
                cout<<"Tree not created  "<<endl;
                //      root=key;
                loc=NULL;
                par=NULL;
        }

        //par=NULL;loc=NULL;
        avlnode *ptr;
        ptr=root;
        while(ptr!=NULL)
        {
                if(ptr->keyword==key)
                {

                        //flag=1;
                        loc=ptr;
                        break;                          //imp for delete1 else it doesnt exit while
loop
                }
                else if(key<ptr->keyword)
                {
                        par=ptr;
                        ptr=ptr->left;
                }

                else
                {
                        par=ptr;                //edit this in previous code
                        ptr=ptr->right;

                }
        }

        if(loc==NULL)
        {
                cout<<"Not found "<<endl;
        }

}

void dictionary::update(string oldkey,string newmean)
{
        search(oldkey);
        loc->meaning=newmean;
        cout<<"UPDATE SUCCESSFUL \n";
}
void dictionary::deletekey(string key)
{

}
void dictionary::inorder(avlnode *root)
{
        if(root)
        {
                inorder(root->left);
        cout<<root->keyword<<" "<<root->meaning<<endl;
        inorder(root->right);
    }
}

int main()
{
    string k,m;
    dictionary d;
    int ch;
    string key,mean;

    do
    {
    cout<<"1.Insert \n2.Update \n3.Ascending \n4.Descending \n5.Display \n6.Quit
\n";
    cin>>ch;
    switch(ch)
    {
        case 1:
        {
            d.accept();
            break;
        }
        case 2:
        {
            cout<<"Enter key whose meaning to update \n";
            cin>>key;
            cout<<"Enter new meaning\n";
            cin>>mean;
            d.update(key,mean);
            break;
```

```
                }
                case 3:
                        d.inorder(d.root);
                        break;


                case 4:
                        cout<<"Descending \n";
                        d.descending(d.root);
                        break;


                case 5:
                        d.inorder(d.root);
                        break;
                default:
                        break;
        }
        }while(ch!=6);  /*cout<<"Enter word and its meaning"<<endl;
                cin>>k>>m;
                d.insert(k,m);*/
//              d.accept();
                //cout<<"Enter another word and its meaning \n";
//              cin>>k>>m;
//              d.insert(k,m);



                //cout<<"MAIN \n";




return 0;
}
```

Source Code:

```
// Assignment No. 9
// Heap Sort Class using T data type as a placeholder and extends Comparable Class to
compare two generic classes
public class HeapSort<T extends Comparable<T>> {


    private T data[];
    private int length;


    HeapSort(T data[]) {
        this.data = data;
        this.length = this.data.length;
    }


    public T[] buildMaxHeap(T[] tempData) {
        for (int i = (int) Math.floor(length / 2); i >= 0; i--) {
            tempData = heapify(tempData, i);
        }
        return tempData;
    }


    public T[] heapify(T[] tempData, int node) {
        int leftNode = node * 2 + 1;
        int rightNode = node * 2 + 2;


        int maxNode = node;


        if (leftNode < length) {
            if (tempData[leftNode].compareTo(tempData[maxNode]) > 0) {
                maxNode = leftNode;
            }
        }
        if (rightNode < length) {
            if ((tempData[rightNode].compareTo(tempData[maxNode])) > 0) {
                maxNode = rightNode;
            }
        }

        if (maxNode != node) {
            T temp = tempData[node];
            tempData[node] = tempData[maxNode];
            tempData[maxNode] = temp;
            tempData = heapify(tempData, maxNode);
        }
        return tempData;
    }

    public void sort() {
        this.data = buildMaxHeap(this.data);
        while (length > 0) {
            this.length--;

            T temp = this.data[0];
            this.data[0] = this.data[length];
            this.data[length] = temp;

            this.data = heapify(this.data, 0);
        }
    }

    public void printData() {
        for (T i : this.data) {
            System.out.print(i + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        /* Sorting Integer Data using Heap Sort */
//      Integer[] dataToBeSorted = {2, 8, 5, 3, 9, 1};
//      HeapSort heapSort = new HeapSort<Integer>(dataToBeSorted);

        /* Sorting Double Data using Heap Sort */
//      Double[] dataToBeSorted = {1.2, 4.3, 6.7, 7.1, 3.9};
//      HeapSort heapSort = new HeapSort<Double>(dataToBeSorted);

        /* Sorting Character Data using Heap Sort */
//      Character[] dataToBeSorted = {'b', 'a', 'z', 'v', 'T'};
//      HeapSort heapSort = new HeapSort<Character>(dataToBeSorted);

        /* Sorting String Data using Heap Sort */
        String[] dataToBeSorted = {"lalu", "vivek", "kia", "priya", "jui"};
        HeapSort heapSort = new HeapSort<String>(dataToBeSorted);

        System.out.println("Given Data - ");
        heapSort.printData();

        heapSort.sort();

        System.out.println("Sorted Data - ");
        heapSort.printData();
    }
}
```

Source Code:

```
// Assignment No. 10

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// Function to build a min heap
void buildMinHeap(vector<int>& arr) {
   int n = arr.size();
   for (int i = (n/2) - 1; i >= 0; i--) {
      int parent = i;
      while (true) {
         int leftChild = 2 * parent + 1;
         int rightChild = 2 * parent + 2;
         int smallest = parent;

         if (leftChild < n && arr[leftChild] < arr[smallest]) {
            smallest = leftChild;
         }
         if (rightChild < n && arr[rightChild] < arr[smallest]) {
            smallest = rightChild;
         }

         if (smallest != parent) {
            swap(arr[parent], arr[smallest]);
            parent = smallest;
         }
         else {
            break;
         }
      }
   }
}

// Function to find the maximum and minimum marks
void findMinMaxMarks(const vector<int>& marks, int& maxMarks, int& minMarks) {
   vector<int> minHeap = marks;
   vector<int> maxHeap = marks;

   // Build min heap
   buildMinHeap(minHeap);

   // Build max heap
   make_heap(maxHeap.begin(), maxHeap.end());

   // Get minimum marks
   minMarks = minHeap[0];

   // Get maximum marks
   maxMarks = maxHeap.front();
}

int main() {
   vector<int> marks;
   int numStudents;

   // Read the number of students
   cout << "Enter the number of students: ";
   cin >> numStudents;

   // Read the marks obtained by students
   cout << "Enter the marks obtained by students:\n";
   for (int i = 0; i < numStudents; i++) {
      int mark;
      cin >> mark;
      marks.push_back(mark);
   }

   // Find the maximum and minimum marks
   int maxMarks, minMarks;
   findMinMaxMarks(marks, maxMarks, minMarks);

   // Display the maximum and minimum marks
   cout << "Maximum marks: " << maxMarks << endl;
   cout << "Minimum marks: " << minMarks << endl;

   return 0;
}
```

<u>Source Code:</u>

```
// Assignment No. 11
#include <iostream>
#include <fstream>
#include <string>
#include <sstr1eam>

using namespace std;

// Structure to hold student information
struct Student {
    int rollNumber;
    string name;
    string division;
    string address;
};

// Function to add a new student record
void addStudentRecord() {
    ofstream outfile("student_data.txt", ios::app);
    if (!outfile) {
        cout << "Error opening file!";
        return;
    }

    Student student;

    cout << "Enter roll number: ";
    cin >> student.rollNumber;
    cout << "Enter name: ";
    cin.ignore();
    getline(cin, student.name);
    cout << "Enter division: ";
    getline(cin, student.division);
    cout << "Enter address: ";
    getline(cin, student.address);

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

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

    outfile.close();
}

// Function to delete a student record
void deleteStudentRecord() {
    int rollNumber;
    cout << "Enter roll number of the student to delete: ";
    cin >> rollNumber;

    ifstream infile("student_data.txt");
    if (!infile) {
        cout << "Error opening file!";
        return;
    }

    ofstream tempFile("temp_data.txt");
    if (!tempFile) {
        cout << "Error creating temporary file!";
        return;
    }

    bool found = false;
    string line;
    while (getline(infile, line)) {
        size_t pos = line.find(",");
        int currentRollNumber = stoi(line.substr(0, pos));
        if (currentRollNumber == rollNumber) {
            found = true;
            continue; // Skip the line to delete the record
        }
        tempFile << line << endl;
    }

    infile.close();
    tempFile.close();

    remove("student_data.txt");
    rename("temp_data.txt", "student_data.txt");

    if (found)
        cout << "Student record deleted successfully!" << endl;
    else
        cout << "Student record not found!" << endl;
}

// Function to display student information
void displayStudentRecord() {
    int rollNumber;
    cout << "Enter roll number of the student to display: ";
    cin >> rollNumber;

    ifstream infile("student_data.txt");
    if (!infile) {
        cout << "Error opening file!";
        return;
    }

    bool found = false;
    string line;
    while (getline(infile, line)) {
        size_t pos = line.find(",");
        int currentRollNumber = stoi(line.substr(0, pos));
        if (currentRollNumber == rollNumber) {
            found = true;
            cout << "Student details:" << endl;
            cout << "Roll number: " << currentRollNumber << endl;

            size_t nextPos = line.find(",", pos + 1);
            string name = line.substr(pos + 1, nextPos - pos - 1);
            cout << "Name: " << name << endl;

            pos = nextPos;
            nextPos = line.find(",", pos + 1);
            string division = line.substr(pos + 1, nextPos - pos - 1);
            cout << "Division: " << division << endl;

            pos = nextPos;
            string address = line.substr(pos + 1);
            cout << "Address: " << address << endl;

            break;
        }
    }

    infile.close();
```

```
    if (!found)
        cout << "Student record not found!" << endl;
}

// Function to search for a student record by name
// Function to search for a student record by name
void searchStudentRecord() {
    string searchName;
    cout << "Enter the name of the student to search: ";
    cin.ignore();
    getline(cin, searchName);

    ifstream infile("student_data.txt");
    if (!infile) {
        cout << "Error opening file!";
        return;
    }

    bool found = false;
    string line;
    while (getline(infile, line)) {
        stringstream ss(line);
        string rollNumberStr;
        getline(ss, rollNumberStr, ',');
        int rollNumber = stoi(rollNumberStr);
        string name;
        getline(ss, name, ',');
        string division;
        getline(ss, division, ',');
        string address;
        getline(ss, address);
        if (name == searchName) {
            found = true;
            cout << "Student details:" << endl;
            cout << "Roll number: " << rollNumber << endl;
            cout << "Name: " << name << endl;
            cout << "Division: " << division << endl;
            cout << "Address: " << address << endl;
            break;
        }
    }

    infile.close();

    if (!found)
        cout << "Student record not found!" << endl;
}


// Main function
int main() {
    int choice;

    do {
        cout << "1. Add student record" << endl;
        cout << "2. Delete student record" << endl;
        cout << "3. Display student record" << endl;
        cout << "4. Search student record" << endl;
        cout << "5. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                addStudentRecord();
                break;
            case 2:
                deleteStudentRecord();
                break;
            case 3:
                displayStudentRecord();
                break;
            case 4:
                searchStudentRecord();
                break;
            case 5:
                cout << "Exiting program..." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }

        cout << endl;
    } while (choice != 5);

    return 0;
}
```

Source Code:

```
// Assignment No. 12
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>

using namespace std;

const int MAX_RECORDS = 20;
const string EMPLOYEE_FILE = "employee_data.txt";

// Structure to hold employee information
struct Employee {
    int emp_id;
    string name;
    string designation;
    double salary;
};

// Function to add a new employee record
void addEmployeeRecord() {
    ofstream outfile(EMPLOYEE_FILE, ios::app);
    if (!outfile) {
        cout << "Error opening file!";
        return;
    }

    Employee employee;

    cout << "Enter employee ID: ";
    cin >> employee.emp_id;
    cout << "Enter name: ";
    cin.ignore();
    getline(cin, employee.name);
    cout << "Enter designation: ";
    getline(cin, employee.designation);
    cout << "Enter salary: ";
    cin >> employee.salary;

    outfile << employee.emp_id << "," << employee.name << "," << employee.designation << "," << employee.salary << endl;

    cout << "Employee record added successfully!" << endl;

    outfile.close();
}

// Function to delete an employee record
void deleteEmployeeRecord() {
    int emp_id;
    cout << "Enter employee ID to delete record: ";
    cin >> emp_id;

    ifstream infile(EMPLOYEE_FILE);
    if (!infile) {
        cout << "Error opening file!";
        return;
    }

    ofstream tempFile("temp_data.txt");
    if (!tempFile) {
        cout << "Error creating temporary file!";
```

```
 return;
}

bool found = false;
string line;
while (getline(infile, line)) {
 stringstream ss(line);
 string field;
 vector<string> fields;
 while (getline(ss, field, ',')) {
  fields.push_back(field);
 }

 if (fields.size() == 4) {
  int currentEmpID = stoi(fields[0]);
  if (currentEmpID == emp_id) {
   found = true;
   continue; // Skip the line to delete the record
  }
 }
 tempFile << line << endl;
}

infile.close();
tempFile.close();

remove(EMPLOYEE_FILE.c_str());
rename("temp_data.txt", EMPLOYEE_FILE.c_str());

if (found)
 cout << "Employee record deleted successfully!" << endl;
else
```

```
        cout << "Employee record not found!" << endl;
}

// Function to display employee information
void displayEmployeeRecord() {
    int emp_id;
    cout << "Enter employee ID to display record: ";
    cin >> emp_id;

    ifstream infile(EMPLOYEE_FILE);
    if (!infile) {
        cout << "Error opening file!";
        return;
    }

    bool found = false;
    string line;
    while (getline(infile, line)) {
        stringstream ss(line);
        string field;
        vector<string> fields;
        while (getline(ss, field, ',')) {
            fields.push_back(field);
        }

        if (fields.size() == 4) {
            int currentEmpID = stoi(fields[0]);
            if (currentEmpID == emp_id) {
                found = true;
                cout << "Employee details:" << endl;
                cout << "Employee ID: " << currentEmpID << endl;
                cout << "Name: " << fields[1] << endl;
                cout << "Designation: " << fields[2] << endl;
                double salary = stod(fields[3]);
                cout << "Salary: " << salary << endl;
                break;
            }
        }
    }

    infile.close();

    if (!found)
        cout << "Employee record not found!" << endl;
}

// Function to search for an employee record
void searchEmployeeRecord() {
    int emp_id;
    cout << "Enter employee ID to search record: ";
    cin >> emp_id;

    ifstream infile(EMPLOYEE_FILE);
    if (!infile) {
        cout << "Error opening file!";
        return;
    }

    bool found = false;
    string line;
    while (getline(infile, line)) {
        stringstream ss(line);
        string field;
        vector<string> fields;
        while (getline(ss, field, ',')) {
            fields.push_back(field);
        }

        if (fields.size() == 4) {
            int currentEmpID = stoi(fields[0]);
            if (currentEmpID == emp_id) {
                found = true;
                cout << "Employee details:" << endl;
                cout << "Employee ID: " << currentEmpID << endl;
                cout << "Name: " << fields[1] << endl;
                cout << "Designation: " << fields[2] << endl;
                double salary = stod(fields[3]);
                cout << "Salary: " << salary << endl;
                break;
            }
        }
    }

    infile.close();

    if (!found)
        cout << "Employee record not found!" << endl;
}

// Function to display the menu
void showMenu() {
    cout << "------------------------- Employee Management System -------------------------" << endl;
    cout << "1. Add employee record" << endl;
    cout << "2. Delete employee record" << endl;
    cout << "3. Display employee record" << endl;
    cout << "4. Search employee record" << endl;
    cout << "5. Exit" << endl;
    cout << "Enter your choice: ";
}

int main() {
    int choice;

    do {
        showMenu();
        cin >> choice;

        switch (choice) {
            case 1:
                addEmployeeRecord();
                break;
            case 2:
                deleteEmployeeRecord();
                break;
            case 3:
                displayEmployeeRecord();
                break;
            case 4:
                searchEmployeeRecord();
                break;
            case 5:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }

        cout << endl;
    } while (choice != 5);

    return 0;
}
```