Practicle A1

Problem Statement:

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

Source Code:

```
class TelephoneBook:
    def __init__(self, size, collision_handling):
        self.size = size
        self.collision_handling = collision_handling
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)

        if self.collision_handling == 'chaining':
            if self.table[index] is None:
                self.table[index] = []
            self.table[index].append((key, value))
        elif self.collision_handling == 'linear_probing':
            while self.table[index] is not None:
                index = (index + 1) % self.size
            self.table[index] = (key, value)

    def search(self, key):
        index = self.hash_function(key)
```

```
        if self.collision_handling == 'chaining':
            if self.table[index] is None:
                return None
            for pair in self.table[index]:
                if pair[0] == key:
                    return pair[1]
            return None
        elif self.collision_handling == 'linear_probing':
            while self.table[index] is not None:
                if self.table[index][0] == key:
                    return self.table[index][1]
                index = (index + 1) % self.size
            return None

# Example usage
telephone_book_chaining = TelephoneBook(10, 'chaining')
telephone_book_chaining.insert(123456, 'John Doe')
telephone_book_chaining.insert(789012, 'Jane Smith')
telephone_book_chaining.insert(345678, 'Michael Johnson')

telephone_book_linear_probing = TelephoneBook(10, 'linear_probing')
telephone_book_linear_probing.insert(123456, 'John Doe')
telephone_book_linear_probing.insert(789012, 'Jane Smith')
telephone_book_linear_probing.insert(345678, 'Michael Johnson')

# Taking input from user with validation
while True:
    try:
        key = int(input("Enter the key you want to search for: "))
        break
    except ValueError:
        print("Please enter a valid integer.")
```

```
# Searching for a telephone number
chaining_result = telephone_book_chaining.search(key)
linear_probing_result = telephone_book_linear_probing.search(key)

if chaining_result is not None:
    print("Chaining - Telephone number for key", key, ":", chaining_result)
else:
    print("Chaining - Key", key, "not found.")

if linear_probing_result is not None:
    print("Linear Probing - Telephone number for key", key, ":", linear_probing_result)
else:
    print("Linear Probing - Key", key, "not found.")
```

Practicle A2

Problem Statement:

To create ADT that implement the "set" concept.

i. Add (newElement) -Place a value into the set

ii. Remove (element) Remove the value

iii. Contains (element) Return true if element is in collection

iv. Size () Return number of values in collection Iterator () Return an iterator used to loop over collection

v. Intersection of two sets

vi. Union of two sets

vii. Difference between two sets

viii. Subset

Source Code:

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

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

    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, otherSet):
    intersectionSet = Set()
    for element in self.elements:
        if otherSet.contains(element):
            intersectionSet.add(element)
    return intersectionSet

def union(self, otherSet):
    unionSet = Set()
    unionSet.elements = self.elements.copy()
    for element in otherSet.elements:
        unionSet.add(element)
    return unionSet

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

def subset(self, otherSet):
    for element in self.elements:
        if not otherSet.contains(element):
            return False
    return True
```

```
# Example usage
set1 = Set()
n = int(input("Enter the number of elements for set 1: "))
print("Enter elements for set 1: ")
for _ in range(n):
    element = input()
    set1.add(element)

set2 = Set()
m = int(input("Enter the number of elements for set 2: "))
print("Enter elements for set 2: ")
for _ in range(m):
    element = input()
    set2.add(element)

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

print("Intersection:", list(set1.intersection(set2).iterator()))
print("Union:", list(set1.union(set2).iterator()))
print("Difference (Set 1 - Set 2):", list(set1.difference(set2).iterator()))

print("Set 1 is a subset of Set 2:", set1.subset(set2))



##OUTPUT:
##Enter the number of elements for set 1: 3
##Enter elements for set 1:
##2
```

##4

##6

##Enter the number of elements for set 2: 4

##Enter elements for set 2:

##2

##4

##6

##8

##Set 1: [2, 4, 6]

##Set 2: [2, 4, 6, 8]

##Intersection: [2, 4, 6]

##Union: [2, 4, 6, 8]

##Difference (Set 1 - Set 2): []

##Set 1 is a subset of Set 2: True

Practicle B1

Problem Statement:

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

Source Code:

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

struct node // Node Declaration
{
   string label;
   //char label[10];
   int ch_count;
   struct node *child[10];
} * root;

class GT // Class Declaration
{
public:
   void create_tree();
   void display(node *r1);

   GT()
   {
      root = NULL;
   }
};

void GT::create_tree()
```

```
{
   int tbooks, tchapters, i, j, k;
   root = new node;
   cout << "Enter name of book : ";
   cin.get();
   getline(cin, root->label);
   cout << "Enter number of chapters in book : ";
   cin >> tchapters;
   root->ch_count = tchapters;
   for (i = 0; i < tchapters; i++)
   {
      root->child[i] = new node;
      cout << "Enter the name of Chapter " << i + 1 << " : ";
      cin.get();
      getline(cin, root->child[i]->label);
      cout << "Enter number of sections in Chapter : " << root->child[i]->label << " : ";
      cin >> root->child[i]->ch_count;
      for (j = 0; j < root->child[i]->ch_count; j++)
      {
         root->child[i]->child[j] = new node;
         cout << "Enter Name of Section " << j + 1 << " : ";
         cin.get();
         getline(cin, root->child[i]->child[j]->label);
      }
   }
}

void GT::display(node *r1)
{
   int i, j, k, tchapters;
```

```
    if (r1 != NULL)
    {
        cout << "\n-----Book Hierarchy---";
        cout << "\n Book title : " << r1->label;
        tchapters = r1->ch_count;
        for (i = 0; i < tchapters; i++)
        {

            cout << "\nChapter " << i + 1;
            cout << " : " << r1->child[i]->label;
            cout << "\nSections : ";
            for (j = 0; j < r1->child[i]->ch_count; j++)
            {
                cout << "\n"<< r1->child[i]->child[j]->label;
            }
        }
    }
    cout << endl;
}

int main()
{
    int choice;
    GT gt;
    while (1)
    {
        cout << "-----------------" << endl;
        cout << "Book Tree Creation" << endl;
        cout << "-----------------" << endl;
        cout << "1.Create" << endl;
```

```
        cout << "2.Display" << endl;
        cout << "3.Quit" << endl;
        cout << "Enter your choice : ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            gt.create_tree();
        case 2:
            gt.display(root);
            break;
        case 3:
            cout << "Thanks for using this program!!!";
            exit(1);
        default:
            cout << "Wrong choice!!!" << endl;
        }
    }
    return 0;
}
```

/*

OUTPUT:

```
-----------------
Book Tree Creation
-----------------
1.Create
2.Display
```

3.Quit

Enter your choice : 1

Enter name of book : Book

Enter number of chapters in book : 2

Enter the name of Chapter 1 : Chap1

Enter number of sections in Chapter : Chap1 : 2

Enter Name of Section 1 : Sec1

Enter Name of Section 2 : Sec2

Enter the name of Chapter 2 : Chap2

Enter number of sections in Chapter : Chap2 : 2

Enter Name of Section 1 : Sec3

Enter Name of Section 2 : Sec4

-----Book Hierarchy---

Book title : Book

Chapter 1 : Chap1

Sections :

Sec1

Sec2

Chapter 2 : Chap2

Sections :

Sec3

Sec4

Book Tree Creation

1.Create

2.Display

3.Quit

Enter your choice : 2

-----Book Hierarchy---

Book title : Book

Chapter 1 : Chap1

Sections :

Sec1

Sec2

Chapter 2 : Chap2

Sections :

Sec3

Sec4

Book Tree Creation

1.Create

2.Display

3.Quit

Enter your choice : 3

Thanks for using this program!!!

*/

Practicle B2

Problem Statement:

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree –

i. Insert new node

ii. Find number of nodes in longest path from root

iii. Minimum data value found in the tree

iv. Change a tree so that the roles of the left and right pointers are swapped at every node

v. Search a value

Source Code:

```
#include<bits/stdc++.h>
using namespace std;

struct Node {
   int data;
   Node *left;
   Node *right;
   Node(int data) {
      this->data = data;
      left = NULL;
      right = NULL;
   }
};

void traversal(Node* root) {
   if (root == NULL) {
      return;
   }
   else {
      cout << root->data << " ";
      traversal(root->left);
```

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

Node* insert(Node* &root, int x) {
    if (root == NULL) {
        root = new Node(x);
    }
    else if (x >= root->data) {
        root->right = insert(root->right, x);
    }
    else if (x < root->data) {
        root->left = insert(root->left, x);
    }
    return root;
}

int depth(Node* root) {
    if (root == NULL) {
        return 0;
    }
    int left_subtree = 1 + depth(root->left);
    int right_subtree = 1 + depth(root->right);
    return max(left_subtree, right_subtree);
}

int minimum(Node* root) {
    if (root->left == NULL) {
        return root->data;
    }
    return min(minimum(root->left), root->data);
}
```

```
bool search(Node* root, int key) {
    if (root == NULL) {
        return false;
    }
    if (root->data == key) {
        return true;
    }
    else if (root->data > key) {
        return search(root->left, key);
    }
    else if (root->data <= key) {
        return search(root->right, key);
    }
    return false; // Return false if the element is not found
}

void mirror(Node* &root) {
    if (root == NULL) {
        return;
    }
    mirror(root->left);
    mirror(root->right);
    swap(root->left, root->right);
}

int main() {
    Node *root = NULL;
    int n;
    cout << "Enter the number of nodes to be inserted in the BST-\n";
    cin >> n;
    for (int i = 0; i < n; i++) {
```

```
        int element;
        cout << "Enter the data value to be inserted-\n";
        cin >> element;
        insert(root, element);
    }
    cout << "Displaying the elements of the BST-\n";
    traversal(root);
    cout << endl;
    cout << "Depth of the tree is " << depth(root) << endl;
    cout << "Minimum value in the BST is " << minimum(root) << endl;

    char choice;
    do {
        int key;
        cout << "Enter the data value to be searched in the BST-\n";
        cin >> key;
        if (search(root, key)) {
            cout << "Key is present in the BST!" << endl;
        }
        else {
            cout << "Key is not present in the BST!" << endl;
        }
        cout << "Do you want to search for another key? (y/n): ";
        cin >> choice;
    } while (choice == 'y' || choice == 'Y');

    return 0;
}

/*
OUTPUT:
```

```
Enter the number of nodes to be inserted in the BST-
12
Enter the data value to be inserted-
10
Enter the data value to be inserted-
8
Enter the data value to be inserted-
15
Enter the data value to be inserted-
12
Enter the data value to be inserted-
13
Enter the data value to be inserted-
7
Enter the data value to be inserted-
9
Enter the data value to be inserted-
17
Enter the data value to be inserted-
20
Enter the data value to be inserted-
18
Enter the data value to be inserted-
4
Enter the data value to be inserted-
5
Displaying the elements of the BST-
10 8 7 4 5 9 15 12 13 17 20 18
Depth of the tree is 5
Minimum value in the BST is 4
Enter the data value to be searched in the BST-
16
```

Key is not present in the BST!

Do you want to search for another key? (y/n): y

Enter the data value to be searched in the BST-

17

Key is present in the BST!

Do you want to search for another key? (y/n): y

Enter the data value to be searched in the BST-

9

Key is present in the BST!

Do you want to search for another key? (y/n): y

Enter the data value to be searched in the BST-

1

Key is not present in the BST!

Do you want to search for another key? (y/n): n

*/

Practicle B1

Problem Statement:

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

Source Code:

```
#include <bits/stdc++.h>
using namespace std;



struct Node {
   int key;
   Node *left, *right;
        bool isThreaded;
};


void populateQueue(Node* root, queue<Node*>* q)
{
   if (root == NULL)
      return;
   if (root->left)
      populateQueue(root->left, q);
   q->push(root);
   if (root->right)
      populateQueue(root->right, q);
}


void createThreadedUtil(Node* root, queue<Node*>* q)
{
   if (root == NULL)
      return;
```

```
    if (root->left)
        createThreadedUtil(root->left, q);
    q->pop();

    if (root->right)
        createThreadedUtil(root->right, q);

    else
            {
        root->right = q->front();
        root->isThreaded = true;
    }
}

void createThreaded(Node* root)
{

    queue<Node*> q;

    populateQueue(root, &q);

    createThreadedUtil(root, &q);
}

Node* leftMost(Node* root)
{
    while (root != NULL && root->left != NULL)
        root = root->left;
    return root;
}

void inOrder(Node* root)
```

```
{
    if (root == NULL)
        return;

    Node* cur = leftMost(root);

    while (cur != NULL) {
        cout << cur->key << " ";

        if (cur->isThreaded)
            cur = cur->right;

        else
            cur = leftMost(cur->right);
    }
}

Node* newNode(int key)
{
    Node* temp = new Node;
    temp->left = temp->right = NULL;
    temp->key = key;
    return temp;
}

int main()
{
    Node* root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(6);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
```

```
    root->right->left = newNode(5);
    root->right->right = newNode(7);

    createThreaded(root);

    cout << "Inorder traversal of created threaded tree "
         "is\n";
    inOrder(root);

    return 0;
}

/*
OUTPUT:
Inorder traversal of created threaded tree is
1 2 3 4 5 6 7
*/
```

Practicle C1

Problem Statement:

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as agraph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.

Source Code:

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

// Global variables to store adjacency matrix and visited status
int adj_mat[50][50] = {0};
int visited[50] = {0};

// Depth-First Search (DFS) traversal function
void dfs(int s, int n, string arr[]) {
    visited[s] = 1;
    cout << arr[s] << " ";
    for (int i = 0; i < n; i++) {
        if (adj_mat[s][i] && !visited[i])
            dfs(i, n, arr);
    }
}

// Breadth-First Search (BFS) traversal function
void bfs(int s, in#include <iostream>
#include <queue>
using namespace std;
```

```
int adj_mat[50][50] = {0, 0};
int visited[50] = {0};

void dfs(int s, int n, string arr[])
{
    visited[s] = 1;
    cout << arr[s] << " ";
    for (int i = 0; i < n; i++)
    {
        if (adj_mat[s][i] && !visited[i])
            dfs(i, n, arr);
    }
}

void bfs(int s, int n, string arr[])
{
    bool visited[n];
    for (int i = 0; i < n; i++)
        visited[i] = false;
    int v;
    queue<int> bfsq;
    if (!visited[s])
    {
        cout << arr[s] << " ";
        bfsq.push(s);
        visited[s] = true;
        while (!bfsq.empty())
        {
            v = bfsq.front();
            for (int i = 0; i < n; i++)
            {
                if (adj_mat[v][i] && !visited[i])
```

```
            {
                cout << arr[i] << " ";
                visited[i] = true;
                bfsq.push(i);
            }
        }
        bfsq.pop();
    }
  }
}

int main()
{
  cout << "Enter no. of cities: ";
  int n, u;
  cin >> n;
  string cities[n];
  for (int i = 0; i < n; i++)
  {
    cout << "Enter city #" << i << " (Airport Code): ";
    cin >> cities[i];
  }

  cout << "\nYour cities are: " << endl;
  for (int i = 0; i < n; i++)
    cout << "city #" << i << ": " << cities[i] << endl;
  for (int i = 0; i < n; i++)
  {
    for (int j = i + 1; j < n; j++)
    {
      cout << "Enter distance between " << cities[i] << " and " << cities[j] << " : ";
      cin >> adj_mat[i][j];
```

```
            adj_mat[j][i] = adj_mat[i][j];
        }
    }
    cout << endl;
    for (int i = 0; i < n; i++)
        cout << "\t" << cities[i] << "\t";
    for (int i = 0; i < n; i++)
    {
        cout << "\n"
             << cities[i];
        for (int j = 0; j < n; j++)
            cout << "\t" << adj_mat[i][j] << "\t";
        cout << endl;
    }
    cout << "Enter Starting Vertex: ";
    cin >> u;
    cout << "DFS: ";
    dfs(u, n, cities);
    cout << endl;
    cout << "BFS: ";
    bfs(u, n, cities);
    return 0;
}


/* OUTPUT:

Enter no. of cities: 4
Enter city #0 (Airport Code): 1
Enter city #1 (Airport Code): 2
Enter city #2 (Airport Code): 3
Enter city #3 (Airport Code): 4
```

Your cities are:

city #0: 1

city #1: 2

city #2: 3

city #3: 4

Enter distance between 1 and 2 : 12

Enter distance between 1 and 3 : 24

Enter distance between 1 and 4 : 36

Enter distance between 2 and 3 : 48

Enter distance between 2 and 4 : 60

Enter distance between 3 and 4 : 72

	1	2	3	4
1	0	12	24	36
2	12	0	48	60
3	24	48	0	72
4	36	60	72	0

Enter Starting Vertex: 12

DFS:

---

*/

Practicle C2

Problem Statement:

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

Source Code:

```
#include<iostream>

using namespace std;

class tree          {

        int a[20][20],l,u,w,i,j,v,e,visited[20];

public:

        void input();

        void display();

        void minimum();

};

void tree::input()

{

        cout<<"Enter the no. of branches: ";

        cin>>v;


        for(i=0;i<v;i++)

        {

                visited[i]=0;

                for(j=0;j<v;j++)

                {

                        a[i][j]=999;

                }

        }

        cout<<"\nEnter the no. of connections: ";

        cin>>e;

        for(i=0;i<e;i++)
```

```
        {
                cout<<"Enter the end branches of connections:  "<<endl;
                cin>>l>>u;
                cout<<"Enter the phone company charges for this connection:  ";
                cin>>w;
                a[l-1][u-1]=a[u-1][l-1]=w;
        }
}
void tree::display()
{
        cout<<"\nAdjacency matrix:";
        for(i=0;i<v;i++)
        {
                cout<<endl;
                for(j=0;j<v;j++)
                {
                        cout<<a[i][j]<<"   ";
                }
                cout<<endl;
        }
}
void tree::minimum()
{
        int p=0,q=0,total=0,min;
        visited[0]=1;
        for(int count=0;count<(v-1);count++)
        {
                min=999;
                for(i=0;i<v;i++)
                {
                        if(visited[i]==1)
                        {
```

```
                                for(j=0;j<v;j++)
                                {
                                        if(visited[j]!=1)
                                        {
                                                if(min > a[i][j])
                                                {
                                                        min=a[i][j];
                                                        p=i;
                                                        q=j;
                                                }
                                        }
                                }
                        }
                }
                visited[p]=1;
                visited[q]=1;
                total=total+min;
                cout<<"Minimum cost connection is"<<(p+1)<<" -> "<<(q+1)<<"  with charge :
"<<min<< endl;
        }
        cout<<"The minimum total cost of connections of all branches is: "<<total<<endl;
}
int main()
{
        int ch;
        tree t;
        do
        {
                cout<<"==========PRIM'S ALGORITHM================="<<endl;
                cout<<"\n1.INPUT\n \n2.DISPLAY\n \n3.MINIMUM\n"<<endl;
                cout<<"Enter your choice :"<<endl;
                cin>>ch;
```

```
        switch(ch)
        {
        case 1: cout<<"*******INPUT YOUR VALUES*******"<<endl;
                t.input();
                break;

        case 2: cout<<"*******DISPLAY THE CONTENTS********"<<endl;
                t.display();
                break;

        case 3: cout<<"*********MINIMUM***********"<<endl;
                t.minimum();
                break;
        }

        }while(ch!=4);
        return 0;
}



/* OUTPUT:

==========PRIM'S ALGORITHM================

1.INPUT

2.DISPLAY

3.MINIMUM

Enter your choice :
```

```
1
*******INPUT YOUR VALUES*******
Enter the no. of branches: 2

Enter the no. of connections: 1
Enter the end branches of connections:
4

9
Enter the phone company charges for this connection:  35
==========PRIM'S ALGORITHM=================

1.INPUT

2.DISPLAY

3.MINIMUM

Enter your choice :
3
*********MINIMUM***********
Minimum cost connection is1 -> 1  with charge : 999
The minimum total cost of connections of all branches is: 999
*/
```

Practicle D1

Problem Statement:

Given sequence k = k1 <k2 < ... <kn of n sorted keys, with a search probability pi for each key ki . Build the Binary search tree that has the least search cost given the access probability for each key?

Source Code:

```
#include <iostream>
using namespace std;
void con_obst(void);
void print(int, int);
float a[20], b[20], wt[20][20], c[20][20];
int r[20][20], n;
int main()
{
   int i;
   cout << "\n****** PROGRAM FOR OBST ******\n";
   cout << "\nEnter the no. of nodes : ";
   cin >> n;
   cout << "\nEnter the probability for successful search :: ";
   cout << "\nâ€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”\n";
   for (i = 1; i <= n; i++)
   {
      cout << "p[" << i << "]";
      cin >> a[i];
   }
   cout << "\nEnter the probability for unsuccessful search :: ";
   cout << "\nâ€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€”â€“\n";
   for (i = 0; i <= n; i++)
   {
      cout << "q[" << i << "]";
      cin >> b[i];
   }
```

```
    con_obst();
    print(0, n);
    cout << endl;
}
void con_obst(void)
{
    int i, j, k, l, min;
    for (i = 0; i < n; i++)
    { // Initialisation
        c[i][i] = 0.0;
        r[i][i] = 0;
        wt[i][i] = b[i];
        // for j-i=1 can be j=i+1
        wt[i][i + 1] = b[i] + b[i + 1] + a[i + 1];
        c[i][i + 1] = b[i] + b[i + 1] + a[i + 1];
        r[i][i + 1] = i + 1;
    }
    c[n][n] = 0.0;
    r[n][n] = 0;
    wt[n][n] = b[n];
    // for j-i=2,3,4....,n
    for (i = 2; i <= n; i++)
    {
        for (j = 0; j <= n - i; j++)
        {
            wt[j][j + i] = b[j + i] + a[j + i] + wt[j][j + i - 1];
            c[j][j + i] = 9999;
            for (l = j + 1; l <= j + i; l++)
            {
                if (c[j][j + i] > (c[j][l - 1] + c[l][j + i]))
                {
                    c[j][j + i] = c[j][l - 1] + c[l][j + i];
```

```
                r[j][j + i] = l;
            }
        }
        c[j][j + i] += wt[j][j + i];
    }
    cout << endl;
  }
  cout << "\n\nOptimal BST is :: ";
  cout << "\nw[0][" << n << "] :: " << wt[0][n];
  cout << "\nc[0][" << n << "] :: " << c[0][n];
  cout << "\nr[0][" << n << "] :: " << r[0][n];
}
void print(int l1, int r1)
{
  if (l1 >= r1)
    return;
  if (r[l1][r[l1][r1] - 1] != 0)
    cout << "\n Left child of " << r[l1][r1] << " :: " << r[l1][r[l1][r1] - 1];
  if (r[r[l1][r1]][r1] != 0)
    cout << "\n Right child of " << r[l1][r1] << " :: " << r[r[l1][r1]][r1];
  print(l1, r[l1][r1] - 1);
  print(r[l1][r1], r1);
  return;
}



/*
OUTPUT:

****** PROGRAM FOR OBST ******
```

Enter the no. of nodes : 3

Enter the probability for successful search ::

GÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇö

p[1]25

p[2]10

p[3]30

Enter the probability for unsuccessful search ::

GÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇöGÇô

q[0]0

q[1]1

q[2]2

q[3]3

Optimal BST is ::

w[0][3] :: 71

c[0][3] :: 122

r[0][3] :: 3

Left child of 3 :: 1

Right child of 1 :: 2

*/

Practicle D2

Problem Statement:

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

Source Code:

```
#include <iostream>

using namespace std;


// Structure for a dictionary entry

struct Entry {

    string keyword;

    string meaning;

    Entry* left;

    Entry* right;

    int height;

};


// Function to get the height of a node

int getHeight(Entry* node) {

    if (node == NULL)

        return 0;

    return node->height;

}


// Function to get the maximum of two integers

int max(int a, int b) {

    return (a > b) ? a : b;

}


// Function to create a new dictionary entry
```

```
Entry* createEntry(string keyword, string meaning) {
    Entry* entry = new Entry;
    entry->keyword = keyword;
    entry->meaning = meaning;
    entry->left = NULL;
    entry->right = NULL;
    entry->height = 1;
    return entry;
}

// Function to perform a right rotation
Entry* rightRotate(Entry* y) {
    Entry* x = y->left;
    Entry* T2 = x->right;

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

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

    return x;
}

// Function to perform a left rotation
Entry* leftRotate(Entry* x) {
    Entry* y = x->right;
    Entry* T2 = y->left;

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

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

    return y;
}

// Function to get the balance factor of a node
int getBalance(Entry* node) {
    if (node == NULL)
        return 0;
    return getHeight(node->left) - getHeight(node->right);
}

// Function to insert a new entry into the dictionary
Entry* insertEntry(Entry* root, string keyword, string meaning) {
    if (root == NULL)
        return createEntry(keyword, meaning);

    if (keyword < root->keyword)
        root->left = insertEntry(root->left, keyword, meaning);
    else if (keyword > root->keyword)
        root->right = insertEntry(root->right, keyword, meaning);
    else {
        cout << "Keyword already exists in the dictionary." << endl;
        return root;
    }

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

    int balance = getBalance(root);

    if (balance > 1 && keyword < root->left->keyword)
```

```
        return rightRotate(root);

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

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

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

    return root;
}

// Function to traverse and print the AVL tree in inorder
void printInorder(Entry* root) {
    if (root == NULL)
        return;

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

int main() {
    Entry* root = NULL;

    // Inserting entries into the AVL tree
```

```
    root = insertEntry(root, "apple", "A fruit");
    root = insertEntry(root, "banana", "Another fruit");
    root = insertEntry(root, "cat", "An animal");

    // Printing the contents of the AVL tree
    cout << "Contents of the AVL tree:" << endl;
    printInorder(root);

    return 0;
}
```

Practicle E1

Problem Statement:

Consider a scenario for Hospitals to cater services to different kinds of patients as a)Serious (top priority),b) non-serious (medium priority), and c) General Checkup (Least priority). Implement the priority queue to cater services to the patients.

Source Code:

```
#include <iostream>
#include <queue>
#include <string>

using namespace std;

// Structure to represent a patient
struct Patient {
    string name;
    int priority; // Priority of the patient, lower value means higher priority

    // Constructor
    Patient(const string& name, int priority) : name(name), priority(priority) {}

    // Overloading the < operator to compare patients based on priority
    bool operator<(const Patient& other) const {
        // Higher priority patients should come before lower priority ones
        return priority > other.priority;
    }
};

// Function to display the menu options
void displayMenu() {
    cout << "Hospital Management System\n";
    cout << "1. Add Patient\n";
    cout << "2. Serve Patient\n";
    cout << "3. Display Patients\n";
```

```
    cout << "4. Exit\n";
}

int main() {
    priority_queue<Patient> patients; // Priority queue to store patients

    int choice;
    do {
        displayMenu();
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1: {
                string name;
                int priority;
                cout << "Enter patient name: ";
                cin >> name;
                cout << "Enter patient priority: ";
                cin >> priority;
                patients.push(Patient(name, priority));
                cout << "Patient added successfully.\n";
                break;
            }
            case 2: {
                if (patients.empty()) {
                    cout << "No patients to serve.\n";
                } else {
                    cout << "Serving patient: " << patients.top().name << endl;
                    patients.pop();
                }
                break;
            }
            case 3: {
```

```
            if (patients.empty()) {
                cout << "No patients in the queue.\n";
            } else {
                cout << "Patients in the queue:\n";
                priority_queue<Patient> temp = patients; // Create a copy of the priority queue
                while (!temp.empty()) {
                    cout << "Name: " << temp.top().name << ", Priority: " << temp.top().priority << endl;
                    temp.pop();
                }
            }
            break;
        }
        case 4: {
            cout << "Exiting program.\n";
            break;
        }
        default: {
            cout << "Invalid choice. Please try again.\n";
            break;
        }
      }
    } while (choice != 4);

    return 0;
}
```

Practicle F1

Problem Statement:

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

Source Code:

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;
struct Student {
    int rollNumber;
    char name[50];
    char division;
    char address[100];
};

void addStudent() {
    ofstream outFile("students.dat", ios::binary | ios::app);
    Student student;
    cout << "Enter Roll Number: ";
    cin >> student.rollNumber;
    cin.ignore();
    cout << "Enter Name: ";
    cin.getline(student.name, 50);
    cout << "Enter Division: ";
    cin >> student.division;
    cin.ignore();
    cout << "Enter Address: ";
    cin.getline(student.address, 100);
```

```
    outFile.write(reinterpret_cast<char*>(&student), sizeof(Student));

    outFile.close();
}

void deleteStudent(int rollNumber) {
    ifstream inFile("students.dat", ios::binary);
    ofstream outFile("temp.dat", ios::binary);

    Student student;
    bool found = false;

    while (inFile.read(reinterpret_cast<char*>(&student), sizeof(Student))) {
        if (student.rollNumber != rollNumber) {
            outFile.write(reinterpret_cast<char*>(&student), sizeof(Student));
        }
        else {
            found = true;
        }
    }

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

    if (found) {
        remove("students.dat");
        rename("temp.dat", "students.dat");
        cout << "Student record deleted successfully." << endl;
    }
    else {
        remove("temp.dat");
```

```
        cout << "Student record not found." << endl;
    }
}

void displayStudent(int rollNumber) {
    ifstream inFile("students.dat", ios::binary);

    Student student;
    bool found = false;

    while (inFile.read(reinterpret_cast<char*>(&student), sizeof(Student))) {
        if (student.rollNumber == rollNumber) {
            cout << "Roll Number: " << student.rollNumber << endl;
            cout << "Name: " << student.name << endl;
            cout << "Division: " << student.division << endl;
            cout << "Address: " << student.address << endl;
            found = true;
            break;
        }
    }

    inFile.close();
    if (!found) {
        cout << "Student record not found." << endl;
    }
}

int main() {
    int choice;
    int rollNumber;

    do {
```

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

    switch (choice) {
        case 1:
            addStudent();
            break;
        case 2:
            cout << "Enter Roll Number of student to delete: ";
            cin >> rollNumber;
            deleteStudent(rollNumber);
            break;
        case 3:
            cout << "Enter Roll Number of student to display: ";
            cin >> rollNumber;
            displayStudent(rollNumber);
            break;
        case 4:
         cout<<"Thanks for using the program !!!";
            break;
        default:
            cout << "Invalid choice. Please try again." << endl;
            break;
    }

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

```
    return 0;
}
```

Practicle F2

Problem Statement:

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

Source Code:

```
#include <bits/stdc++.h>


#define max 20
using namespace std;


struct employee {
   string name;
   long int code;
   string designation;
   int exp;
   int age;
};


int num;
void showMenu();


employee emp[max], tempemp[max],
   sortemp[max], sortemp1[max];


// void build()
// {
//    cout << "Maximum Entries can be "<< max << "\n";


//    cout << "Enter the number of Entries required: ";
//    cin >> num;
```

```
//     if (num > 20) {
//         cout << "Maximum number of Entries are 20.\n";
//         num = 20;
//     }
//     cout << "Enter the following data:\n";4

//     for (int i = 0; i < num; i++) {
//         cout << "Name: ";
//         cin >> emp[i].name;

//         cout << "Employee ID: ";
//         cin >> emp[i].code;

//         cout << "Designation: ";
//         cin >> emp[i].designation;

//         cout << "Experience: ";
//         cin >> emp[i].exp;

//         cout << "Age: ";
//         cin >> emp[i].age;
//     }
//     showMenu();
// }

void insert()
{
    if (num < max) {
        int i = num;
        num++;
```

```
        cout << "Enter the information of the Employees.\n";
        cout << "Name: ";
        cin >> emp[i].name;

        cout << "Employee ID: ";
        cin >> emp[i].code;

        cout << "Designation: ";
        cin >> emp[i].designation;

        cout << "Experience: ";
        cin >> emp[i].exp;

        cout << "Age: ";
        cin >> emp[i].age;
    }
    else {
        cout << "Employee Size Full\n";
    }
    showMenu();
}

// Function to delete record at index i
void deleteIndex(int i)
{
    for (int j = i; j < num - 1; j++) {
        emp[j].name = emp[j + 1].name;
        emp[j].code = emp[j + 1].code;
        emp[j].designation = emp[j + 1].designation;
        emp[j].exp = emp[j + 1].exp;
        emp[j].age = emp[j + 1].age;
    }
```

```
    return;
}

// Function to delete record
void deleteRecord()
{
    cout << "Enter the Employee ID to Delete Record: ";
    int code;
    cin >> code;
    for (int i = 0; i < num; i++) {
        if (emp[i].code == code) {
            deleteIndex(i);
            num--;
            break;
        }
    }
    showMenu();
}

void searchRecord()
{
    cout << "Enter the Employee ID to Search Record: ";

    int code;
    cin >> code;

    for (int i = 0; i < num; i++) {

        // If the data is found
        if (emp[i].code == code) {
            cout << "Name: "
                 << emp[i].name << "\n";
```

```
            cout << "Employee ID: "
                 << emp[i].code << "\n";

            cout << "Designation: "
                 << emp[i].designation << "\n";

            cout << "Experience: "
                 << emp[i].exp << "\n";

            cout << "Age: "
                 << emp[i].age << "\n";
            break;
        }
    }

    showMenu();
}

void showMenu()
{
    cout << "------------------------- Employee Management System -------------------------\n\n";
    cout << "Available Options:\n\n";
    cout << "Insert New Entry    (1)\n";
    cout << "Delete Entry        (2)\n";
    cout << "Search a Record     (3)\n";
    cout << "Exit                (4)\n";

    int option;
    cin >> option;

    if (option == 1) {
```

```
        insert();
    }
    else if (option == 2) {
        deleteRecord();
    }
    else if (option == 3) {
        searchRecord();
    }
    else if (option == 4) {
        return;
    }
    else {
        cout << "Expected Options are 1 to 4";
        showMenu();
    }
}

int main()
{
    showMenu();
    return 0;
}
```