Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with without replacement. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique. Standard Operations: Insert(key, value), Find(key), Delete(key)
Code:

```
class Hashtable:
    def __init__(self):
        self.size = 7
        self.h = [[None, None] for _ in range(self.size)]
        self.count = 0

    def input(self):
        key = input("\nEnter the key: ")
        value = input("Enter the value: ")
        self.hashf(key, value)

    def hashf(self, key, value):
        sum = 0
        for char in key:
            sum += ord(char)

        ch = sum % self.size
        self.linearp(key, value, ch)

    def linearp(self, key, value, ch):
        if self.count == self.size:
            print("Table is Full")
        else:
            while self.h[ch][0] is not None and self.count != self.size:
                ch = (ch + 1) % self.size

            self.h[ch][0] = key
            self.h[ch][1] = value
            self.count += 1

    def display(self):
        print("\n\t[Key]\t[Value]")
        for i in range(self.size):
            for j in range(2):
                print("\t", self.h[i][j], end="")

            print()

    def search(self):
        k = input("\nEnter the string to search: ")
        sum = 0
        for char in k:
            sum += ord(char)

        ch = sum % self.size
        if self.count == self.size:
            print("\nSearch is not found")
        else:
```

```
            while self.h[ch][0] != k and self.count != self.size:
                ch = (ch + 1) % self.size

            self.count += 1
            if self.h[ch][0] == k:
                print(f"String '{k}' found at index {ch}")

    def delete(self):
        k = input("\nEnter the string to delete: ")
        sum = 0
        for char in k:
            sum += ord(char)

        ch = sum % self.size
        if self.count == self.size:
            print("Search is not found")
        else:
            while self.h[ch][0] != k and self.count != self.size:
                ch = (ch + 1) % self.size

        self.h[ch][0] = None
        self.h[ch][1] = None
        print(f"String '{k}' deleted from index {ch}")


def main():

    h1 = Hashtable()

    while True:
        print("\nEnter choice\n1. input  2. display  3. search  4. delete\nChoice [1/2/3/4]: ")
        ch = int(input())
        if ch == 1:
            h1.input()
        elif ch == 2:
            h1.display()
        elif ch == 3:
            h1.search()
        elif ch == 4:
            h1.delete()
        else:
            break


if __name__ == "__main__":
    main()
```

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

Code:

```
class HashTable:
def __init__(self):
    self.hash_table = [-1] * 10
    self.count = 0
    self.prob = 0

def linear(self, key):
    address = key % 10
    while True:
        if self.hash_table[address] == -1:
            self.hash_table[address] = key
            self.count += 1
            break
        else:
            address += 1
            if address == 10:
                address = 0
        if self.count == 10:
            print("Hash Table is Full")
            break

def quadratic(self, key):
    address = key % 10

    while True:

        if self.hash_table[address] == -1:
            self.hash_table[address] = key
            self.count += 1
            break
        else:
            for i in range(10):
                address1 = (address + (i* i)) % 10
                if address1 == 10:
                    address1 = 0
                if self.hash_table[address1] == -1:
                    self.hash_table[address1] = key
                    self.count += 1
                    break
        break
        if self.count == 10:
            print("Hash Table is Full")
            break

def linear_search(self, search):
    address = search % 10
```

```
        while self.prob < 10:
            if self.hash_table[address] == search:
                print("Found")
                self.prob += 1
                break
            else:
                address += 1
                self.prob += 1
                if address == 10:
                    address = 0
        else:
            print("Not found")

    def quadratic_search(self, search):
        address = search % 10
        self.prob = 0
        while self.prob < 10:
            if self.hash_table[address] == search:
                print("Found")
                self.prob += 1
                break
            else:
                    for i in range(10):
                        address1 = (address + (i* i)) % 10
                        self.prob += 1
                        if address1 == 10:
                            address1=0
                        if self.hash_table[address1] == search:
                            print("Found")
                            break
            break
        else:
            print("Not found")


    def display(self):
        print("Hash Table:", self.hash_table)



if __name__ == "__main__":
    obj1 = HashTable()
    obj2 = HashTable()

    while True:
        print("\n1) Linear")
        print("2) Quadratic")
        ch = int(input("Enter your choice: "))

        if ch == 1:
            obj1.hash_table = [-1] * 10
            obj1.count = 0
            obj1.prob = 0
            n = int(input("Enter Total number of records: "))
```

```
        if n<=10:
            for i in range(n):
                key = int(input("Enter number: "))
                obj1.linear(key)
            obj1.display()
            search = int(input("Enter the mobile number to be searched: "))
            obj1.linear_search(search)
            print("The comparison required is:", obj1.prob)
    elif ch == 2:
        obj2.hash_table = [-1] * 10
        obj2.count = 0
        obj1.prob = 0
        n = int(input("Enter Total number of records: "))
        if n<=10:
            for i in range(n):
                key = int(input("Enter number: "))
                obj2.quadratic(key)
            obj2.display()
            search = int(input("Enter the mobile number to be searched: "))
            obj2.quadratic_search(search)
            print("The comparison required is:", obj2.prob)
    else:
        print("Invalid choice")
```

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.
Code:

```
#include<iostream>

using namespace std;

struct node
{
        string label;
        int chno;
        struct node* child[10];
}*root;

class tree
{
        public:

                tree()
                {
                        root = NULL;
                }
                void create()
                {
                        int books, chapters;
                        root = new node;
                        cout << "Enter the name of Book :";
```

```
                cin.get();
                getline(cin, root->label);
                cout << "Enter the number of chapters: ";
                cin >> chapters;
                root->chno = chapters;
                for (int i = 0; i < chapters; i++)
                {
                        root->child[i] = new node;
                        cout << "Name of the chapter " << i + 1 << ":";
                        cin.get();
                        getline(cin, root->child[i]->label);
                        cout << "Enter no of sctions in chapter" << root->child[i]->label << ":";
                        cin >> root->child[i]->chno;
                        for (int j = 0; j < root->child[i]->chno; j++)
                        {
                                root->child[i]->child[j] = new node;
                                cout << "Enter the name of Section " << j + 1 << ":";
                                cin.get();
                                getline(cin, root->child[i]->child[j]->label);
                        }
                }

        }
        void display()
        {
                int tchapters;
                if(root!=NULL)
                {
                        cout<<"\nBook list";
                        cout<<"\nBook Title: "<<root->label;
                        tchapters=root->chno;
                        for (int i = 0; i < tchapters; i++)
                        {
                                cout<<"\nChapter "<<i+1;
                                cout<<" :"<<root->child[i]->label;
                                cout<<"\nSections :";
                                for (int j = 0; j < root->child[i]->chno; j++)
                                {
                                        cout<<"\n"<< root->child[i]->child[j]->label << "\n";
                                }


                        }

                }
        }
};


int main()
{
        tree obj;
        obj.create();
   obj.display();
}
```

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

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

struct Node {
   int data;
   Node* left;
   Node* right;
};
Node* createNode(int value) {
   Node* newNode = new Node;
   newNode->data = value;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}
Node* insertNode(Node* root, int value) {
   if(root == NULL) root = createNode(value);
   else if(value <= root->data) root->left = insertNode(root->left, value);
   else root->right = insertNode(root->right, value);
   return root;
}
int findHeight(Node* root) {
   if(root == NULL) return -1;
   else {
      int leftHeight = findHeight(root->left);
      int rightHeight = findHeight(root->right);
      return max(leftHeight, rightHeight) + 1;
   }
}
int findMin(Node* root) {
   if(root == NULL) {
      cout << "Error: Tree is empty!" << endl;
      return -1;
   }
   else if(root->left == NULL) return root->data;
   else return findMin(root->left);
}
Node* swapNodes(Node* root) {
   if(root == NULL) return NULL;
   else {
      Node* temp = root->left;
      root->left = swapNodes(root->right);
      root->right = swapNodes(temp);
```

```
        return root;
    }
}
bool search(Node* root, int value) {
    if(root == NULL) return false;
    else if(root->data == value) return true;
    else if(value <= root->data) return search(root->left, value);
    else return search(root->right, value);
}
void printMenu() {
    cout << "Menu:" << endl;
    cout << "1. Insert a new node" << endl;
    cout << "2. Find the number of nodes in longest path from root" << endl;
    cout << "3. Find the minimum data value in the tree" << endl;
    cout << "4. Swap the left and right pointers at every node" << endl;
    cout << "5. Search a value" << endl;
    cout << "6. Exit" << endl;
}

int main() {
    Node* root = NULL;
    int arr[] = { };
    int n = sizeof(arr)/sizeof(arr[0]);
    for(int i = 0; i < n; i++) { root = insertNode(root, arr[i]); }
    int choice = 0;
    int value;
    while(choice != 6) {
        printMenu();
        cout << "Enter your choice: ";
        cin >> choice;
        switch(choice) {
            case 1:
                cout << "Enter the value to be inserted: ";
                cin >> value;
                root = insertNode(root, value);
                cout << "Node inserted successfully!" << endl;
                break;
            case 2:
                cout << "The number of nodes in the longest path from root is: " << findHeight(root) << endl;
                break;
            case 3:
                cout << "The minimum data value in the tree is: " << findMin(root) << endl;
                break;
            case 4:
                root = swapNodes(root);
                cout << "The tree has been swapped successfully!" << endl;
                break;
            case 5:
                cout << "Enter the value to be searched: ";
                cin >> value;
                if(search(root, value)) cout << "The value " << value << " is present in the tree." << endl;
                else cout << "The value " << value << " is not present in the tree." << endl;
                break;
            case 6:
                cout << "Exiting..." << endl;
```

```
            break;
        default:
            cout << "Error: Invalid choice! Please enter a valid choice." << endl;
            break;
      }
      cout<<endl;

   }
   return 0;
}
```

Construct an expression tree from the given prefix expression eg. +--a*bc/def and traverse it using postorder traversal (non recursive) and then delete the entire tree.
Code:

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

struct TreeNode {
   char data;
   TreeNode* left;
   TreeNode* right;
};
TreeNode* newNode(char data) {
   TreeNode* node = new TreeNode;
   node->data = data;
   node->left = node->right = NULL;
   return node;
}
TreeNode* constructExpressionTree(string prefix) {
   stack<TreeNode*> s;
   for (int i = prefix.length() - 1; i >= 0; i--) {
      char c = prefix[i];
      if (isdigit(c) || isalpha(c)) {
         TreeNode* node = newNode(c);
         s.push(node);
      }
      else {
         TreeNode* node = newNode(c);
```

```
            node->left = s.top();
            s.pop();
            node->right = s.top();
            s.pop();
            s.push(node);
        }
    }
    return s.top();
}
void postorderTraversal(TreeNode* root) {
    stack<TreeNode*> s;
    TreeNode* lastNodeVisited = NULL;
    while (!s.empty() || root != NULL) {
        if (root != NULL) {
            s.push(root);
            root = root->left;
        }
        else {
            TreeNode* peekNode = s.top();
            if (peekNode->right != NULL && lastNodeVisited != peekNode->right) root = peekNode->right;
            else {
                cout << peekNode->data << " ";
                lastNodeVisited = peekNode;
                s.pop();
            }
        }
    }
}
void deleteTree(TreeNode* root) {
    if (root == NULL) return;
    deleteTree(root->left);
    deleteTree(root->right);
    delete root;
}

int main() {
```

```
int choice;
string prefix;
TreeNode* root = NULL;
do {
   cout << "Menu" << endl;
   cout << "1. Enter prefix expression" << endl;
   cout << "2. Postorder traversal" << endl;
   cout << "3. Delete tree" << endl;
   cout << "4. Exit" << endl;
   cout << "Enter your choice: ";
   cin >> choice;
   switch (choice) {
      case 1:
         cout << "Enter prefix expression: ";
         cin.ignore();
         getline(cin, prefix);
         root = constructExpressionTree(prefix);
         break;
      case 2:
         if (root == NULL) cout << "Tree not constructed yet" << endl;
         else {
            cout << "Postorder traversal of expression tree: ";
            postorderTraversal(root);
            cout << endl;
         }
         break;
      case 3:
         if (root == NULL) cout << "Tree not constructed yet" << endl;
         else {
            deleteTree(root);
            root = NULL;
            cout << "Tree deleted" << endl;
         }
         break;
      case 4:
         cout << "Exiting program" << endl;
```

```
            break;
        default:
            cout << "Invalid choice" << endl;
            break;
    }
    cout << endl;
  } while (choice != 4);
  return 0;
}
```

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

Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;
// Structure to represent an edge
struct Edge {
char destination;
int cost;
};
// Function to implement Dijkstra's algorithm
void dijkstra(vector<vector<Edge>>& graph, char source, char destination) {
int numCities = graph.size();
// Create a vector to store the shortest distance from source to each city
vector<int> distance(numCities, INT_MAX);
// Create a min heap to store the vertices and their distances
priority_queue<pair<int, char>, vector<pair<int, char>>, greater<pair<int, char>>> pq;
// Initialize the distance of source city as 0 and push it into the min heap
distance[source - 'A'] = 0;
pq.push({0, source});
// Process vertices until the min heap is empty
while (!pq.empty()) {
char currentCity = pq.top().second;
int currentDistance = pq.top().first;
pq.pop();
// If the current distance is greater than the shortest distance, skip
if (currentDistance > distance[currentCity - 'A']) {
continue;
}
// Iterate through all the neighboring cities of the current city
for (Edge& edge : graph[currentCity - 'A']) {
int newDistance = distance[currentCity - 'A'] + edge.cost;
```

```
// If the new distance is shorter, update the shortest distance
if (newDistance < distance[edge.destination - 'A']) {
distance[edge.destination - 'A'] = newDistance;
pq.push({newDistance, edge.destination});
}

}
}
// Print the shortest distance from source to destination
cout << "Shortest distance from city " << source << " to city " << destination << ": ";
if (distance[destination - 'A'] == INT_MAX) {
cout << "No path found." << endl;
} else {
cout << distance[destination - 'A'] << endl;
}
}
int main() {
int numCities, numFlights;
cout << "Enter the number of cities: ";
cin >> numCities;
cout << "Enter the number of flights: ";
cin >> numFlights;
// Create a graph to represent the flight paths between cities
vector<vector<Edge>> graph(numCities);
// Take input for flight paths and their costs
for (int i = 0; i < numFlights; i++) {
char source, destination;
int cost;
cout << "Enter the source city, destination city, and cost of flight " << i + 1 << ": ";
cin >> source >> destination >> cost;
// Add the flight path to the graph
graph[source - 'A'].push_back({destination, cost});
}
char source, destination;
cout << "Enter the source city: ";
cin >> source;
cout << "Enter the destination city: ";
cin >> destination;
// Call the Dijkstra's algorithm function
dijkstra(graph, source, destination);
return 0;
}
/*OUTPUT :
Enter the number of cities: 5
Enter the number of flights: 7
Enter the source city, destination city, and cost of flight 1: A B 5
Enter the source city, destination city, and cost of flight 2: A C 3
Enter the source city, destination city, and cost of flight 3: B D 2
Enter the source city, destination city, and cost of flight 4: C D 4
Enter the source city, destination city, and cost of flight 5: C E 6
Enter the source city, destination city, and cost of flight 6: D E 1
Enter the source city, destination city, and cost of flight 7: E B 2
Enter the source city: A
Enter the destination city: E
Shortest distance from city A to city E:*/
```

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

```
#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <stack>
using namespace std;
// Function to perform DFS on the graph
void dfs(vector<vector<int>>& graph, int startNode) {
	stack<int> s;
	vector<bool> visited(graph.size(), false);

	s.push(startNode);
	visited[startNode] = true;

 while(!s.empty()) {
	int currNode = s.top();
	s.pop();
	cout << "Visited Node: " << currNode << endl;

	for (int neighbor : graph[currNode]) {
	if (!visited[neighbor]) {
		s.push(neighbor);
	visited[neighbor] = true;
	}
}
}
}
// Function to perform BFS on the graph
void bfs(vector<vector<int>>& graph, int startNode) {
	queue<int> q;
	vector<bool> visited(graph.size(), false);

	q.push(startNode);
	visited[startNode] = true;

	while (!q.empty()) {
		int currNode = q.front();
		q.pop();
		cout << "Visited Node: " << currNode << endl;

		for (int neighbor : graph[currNode]) {
			if (!visited[neighbor]) {
			q.push(neighbor);
			visited[neighbor] = true;
		}
	}
```

```
 }
}
int main() {
	int numNodes;
	cout << "Enter the number of nodes in the graph: ";
	cin >> numNodes;
	map<int, string> nodeNames;
	cout << "Enter the name of each node, separated by spaces:\n";
	for (int i = 1; i <= numNodes; i++) {
		string nodeName;
		cin >> nodeName;
		nodeNames[i] = nodeName;
}

vector<vector<int>> graph(numNodes + 1); // Since we are using 1-indexed nodes
cout << "Enter the connections between the nodes (Enter 0 0 to stop):\n";  while (true) {
	int node1, node2;
	cin >> node1 >> node2;
	if (node1 == 0 && node2 == 0) {
		break;
	}
	graph[node1].push_back(node2);
	graph[node2].push_back(node1);
}
int startNode;
cout << "Enter the starting node for DFS and BFS: ";
cin >> startNode;



cout << "DFS traversal starting from Node " << startNode << ":\n";  dfs(graph, startNode);

cout << "\nBFS traversal starting from Node " << startNode << ":\n";  bfs(graph, startNode);

return 0;
}
/*/Output:
cnsl@cnsl-ThinkCentre-M70s:~$ g++ college.cpp
^[[Acnsl@cnsl-ThinkCentre-M70s:~$ ./a.out
Enter the number of nodes in the graph: 5
Enter the name of each node, separated by spaces:
A B C D E
Enter the connections between the nodes (Enter 0 0 to stop):
1 2
1 3
2 4
3 4
3 5
0 0
Enter the starting node for DFS and BFS: 1
DFS traversal starting from Node 1:
Visited Node: 1
Visited Node: 3
Visited Node: 5
Visited Node: 4
```

```
Visited Node: 2
BFS traversal starting from Node 1:
Visited Node: 1
Visited Node: 2
Visited Node: 3
Visited Node: 4
Visited Node: 5
/*/
```

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?
Code:

```
#include <iostream>
using namespace std;
//Number of Keys
int Keys = 3;
//.....Class to create an Array of Objects of (key, freq)
class OBST
{
int key;
int freq;
public:
void init_Keys();
void bubble_Sort();
void create_OBST();
void show_Preorder(struct BSTNode *root);
}
obj[3];
//Member Function to Accept and Display Object's (key, freq)
void OBST::init_Keys()
{
int i;

for(i=0; i<Keys; i++)
{
cout<<"\n\t Enter Key: ";
cin>>obj[i].key;

cout<<"\n\t Enter Frequency: ";
cin>>obj[i].freq;
}

cout<<"\n\t Key Frequency";
for(i=0; i<Keys; i++)
{
cout<<"\n\t "<<obj[i].key;

cout<<"\t"<<obj[i].freq;
}
}
//Member Function to Sort Keys in Decreasing Order of their Frequencies
```

```
void OBST::bubble_Sort()
{
int i,j;
int Key;
int Freq;

for(i=0; i<Keys-1; i++)
{
for(j=0; j<Keys-1; j++)
{
if(obj[j].freq < obj[j+1].freq)
{
Key = obj[j].key;
Freq = obj[j].freq;

obj[j].key = obj[j+1].key;

obj[j].freq = obj[j+1].freq;

obj[j+1].key = Key;
obj[j+1].freq = Freq;
}
}
}

cout<<"\n\t Key Frequency";
for(i=0; i<Keys; i++)
{
cout<<"\n\t "<<obj[i].key;

cout<<"\t"<<obj[i].freq;
}
}
//Structure of Node of BST
struct BSTNode
{
int key;
int freq;
struct BSTNode *left;
struct BSTNode *right;
}*Root;
//Member Function to create an OBST
void OBST::create_OBST()
{
int i;
int done;

struct BSTNode *Newnode, *current;

i = 0;
while(i < Keys)
{
done = 0;

Newnode = new struct BSTNode;
```

```
Newnode->key = obj[i].key;
Newnode->freq = obj[i].freq;
Newnode->left = NULL;
Newnode->right = NULL;

if(Root == NULL)
{
Root = Newnode;
}
else
{
current = Root;
while(!done)
{
if(Newnode->key < current->key)
{
if(current->left == NULL)
{
current->left = Newnode;
done = 1;

}
else
current = current->left;
}
else
{
if(current->right == NULL)
{
current->right = Newnode;
done = 1;
}
else
current = current->right;
}
}
}
i++;
}
}
//Member Function to display OBST in Preorder
int level = 1;
int cost = 0;
void OBST::show_Preorder(struct BSTNode *root)
{
if(root)
{
cout<<" "<<root->key;
cost = cost + level*root->freq;
level++;

this->show_Preorder(root->left);
this->show_Preorder(root->right);
}
```

```
}
//Main Function
int main()
{
cout<<"\n -----*** A C++ Program to create an Optimal BST ***------ \n";
OBST Obj1;

cout<<"\n 1. Accept and Display Keys and Frequencies.....\n";
Obj1.init_Keys();

cout<<"\n 2. Sort Keys as per Frequencies.....\n";
Obj1.bubble_Sort();

cout<<"\n 3. Create OBST and Display OBST in Preorder.....\n";
Obj1.create_OBST();

cout<<"\n Preorder(OBST): ";
Obj1.show_Preorder(Root);
cout<<"\n Total Searching Cost: "<<cost;
return 0;
}
```

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.
Code:

```
#include <iostream> //Header Files
using namespace std;
struct HBTNode //Structure for HBTre Node
{
int Key;
char Mean[10];
HBTNode *left;
HBTNode *right;
} *Root;

void create_HBT() //Function to store keywords and meanings in HBT
{
int i;
int nodes;
int done;
struct HBTNode *Newnode, *current;
cout << "\n\n Enter the no of nodes to insert in HBT.? : ";
cin >> nodes;
for (i = 0; i < nodes; i++)
{
Newnode = new struct HBTNode; //memory allocation
cout << "\n\t Enter Keyword: "; //store keys

cin >> Newnode->Key;
cout << "\n\t Enter Meaning: "; //store meanings
cin >> Newnode->Mean;
```

```
Newnode->left = NULL; //left and right pointers initiall null

Newnode->right = NULL;
if (Root == NULL)
{
Root = Newnode;
}
else
{
done = 0;
current = Root;
while (!done)
{
if (Newnode->Key < current->Key)
{
if (current->left == NULL)
{
current->left = Newnode;
done = 1;
}
else
current = current->left;
}
else
{
if (current->right == NULL)

{
current->right = Newnode;
done = 1;
}
else
current = current->right;
}
} //while
} //else
} //for
} //function end
//Function to display keywords and meanings in HBT
void display_HBT(struct HBTNode *root)
{
if (root) //pre-order display
{
cout << "\n\t" << root->Key << " - " << root->Mean; //...Data
display_HBT(root->left); //...Left
display_HBT(root->right); //...Right
}
}

void Sorted_List(struct HBTNode *root)
{
if (root)
{
```

```
Sorted_List(root->left); //...Left
cout << "\n\t" << root->Key << " - " << root->Mean; //...Data

Sorted_List(root->right); //...Right
}
}

void Find_Keyword(int key)
{
int comp = 0;
int level = 0;
int done;
struct HBTNode *current;
done = 0;
current = Root;
while (!done)
{
if (key < current->Key)
{
current = current->left;
level++;
comp++;
}
else if (key > current->Key)
{
current = current->right;

level++;
comp++;
}
else
{
done = 1;

comp++;
cout << "\n\t Key : " << key;
cout << "\n\t Found at Level: " << level;
cout << "\n\t No. of Comparisons: " << comp;
}
}
}

int main()
{
cout << "\n -------***A C++ Program to Implement Dictionary using Height-Balanced Tree.***-------";
cout<< "\n 1. Store Keywords and Meanings in Height-Balanced Tree.";
Root = NULL;
create_HBT();
cout << "\n 2. Display Keywords and Meanings in Height-Balanced Tree.";
cout << "\n Keyword - Meaning";
display_HBT(Root);
cout << "\n 3. Display a Sorted List of Keywords and Meanings.";
cout << "\n Keyword - Meaning";
Sorted_List(Root);
cout << "\n 4. Display Number of Comparisons required to find a keyword.";
```

```
Find_Keyword(1);
return 0;
}
```

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

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

class Heap{
public:
   void maxHeapify (int [], int, int);
   void buildMaxHeap (int [], int);
   void heapsort (int [], int);
   void accept ();
   void display (int [],int);
};

void Heap::maxHeapify (int marks[], int i, int n) {      //reheapdown - deleting element from top location
   int l, r, largest;
   l = 2 * i;
   r = (2 * i + 1);

   largest = ((l <= n) && marks[l] > marks[i]) ? l : i;

   if ((r <= n) && (marks[r] > marks[largest]))
      largest=r;

   if (largest != i) {
      swap(marks[largest], marks[i]);
      maxHeapify (marks, largest,n);
   }
}

void Heap::buildMaxHeap (int marks[], int n) {
   for (int k = n / 2; k >= 1; k--)
      maxHeapify (marks, k, n);
}

void Heap::heapsort (int marks[], int n) {
   buildMaxHeap (marks,n);
   for (int i = n; i >= 2; i--) {
      swap (marks[i], marks[1]);
      maxHeapify (marks, 1, i - 1);
   }
}

void Heap::accept (){
   int n;
   cout << "Enter the number of students: ";
   cin >> n;
   int marks[n];
```

```
   cout << "\nEnter the marks of the students: ";
   for (int i = 1; i <= n; i++)
      cin >> marks[i];

   heapsort (marks, n);
   display (marks, n);
}

void Heap::display (int marks[],int n) {
   cout << "\n:::::::SORTED MARKS::::::\n\n";

   for (int i = 1; i <= n; i++)
      cout << marks[i] << endl;

   cout << "\nMinimum marks obtained: " << marks[1];
   cout << "\nMaximum marks obtained: " << marks[n];
}

int main () {
     Heap h;
   h.accept ();
   return 0;
}

/*
Enter the number of students: 8

Enter the marks of the students: 12 74 84 19 52 41 22 93

:::::::SORTED MARKS::::::

12
19
22
41
52
74
84
93

Minimum marks obtained: 12
Maximum marks obtained: 93
*/
```

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

Code:

```
#include <bits/stdc++.h>
#define max 20
using namespace std;
struct employee {
string name;
long int code;
string designation;
int exp;
int age, salary;
};
int num;
void showMenu();
employee emp[max], tempemp [max], sortemp[max], sortemp1[max];
void build() {
cout << "Build The Table\n";
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";
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;
cout << "Salary: ";
cin >> emp[i].salary;
cout<<"\n\n";
}
showMenu();
}
void insert() {
if (num < max) {
int i = num;
num++;
cout << "Enter the information "
<< "of the Employee\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;
cout << "Salary ";
cin >> emp[i].salary;
} else {
cout << "Employee Table Full\n";
}
showMenu();
}
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;
emp[j].salary = emp[j + 1].salary;
}
return;
}
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 (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";
cout << "Salary "<<emp[i].salary << "\n";
break;
}
}
showMenu();
```

```
}
void showMenu() {
cout <<"----------" << "Employee Management System" <<"----------\n\n";
cout << "Available Options:\n\n";
cout << "Build Table (1)\n";
cout << "Insert New Entry (2)\n";
cout << "Delete Entry (3)\n";
cout << "Search a Record (4)\n";
cout << "Exit (5)\n";
int option;
cout<<"Enter Selected Option:";
cin >> option;
if (option == 1) {
build();
} else if (option == 2) {
insert();
} else if (option == 3) {
deleteRecord();
} else if (option == 4) {
searchRecord();
} else if (option ==5) {
return;
}
else {
cout << "Expected Options are 1/2/3/4/5";
showMenu();
}
cout<<"Thank You!";
}
int main()
{
showMenu();
return 0;
}

/*Output:'
----------Employee Management System----------
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
Enter Selected Option:1
Build The Table
Maximum Entries can be 20
Enter the number of Entries required: 2
Enter the following data:
Name: xyz
Employee ID: 1
Designation: abc
Experience: 2
Age: 30
Salary: 600000
Name: pqr
```

Employee ID: 2
Designation: asd
Experience: 1
Age: 25
Salary: 500000
----------Employee Management System----------
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
Enter Selected Option:4
Enter the Employee ID to Search Record 1
Name xyz
Employee ID 1
Designation abc
Experience2
Age30
Salary 600000*/.

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

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
void addstudent()
{
   ofstream f("db.txt",ios::app);

   string rn,div,name,add;
   cout<<"\n enter the roll no of student:";
   cin>>rn;
   cout<<"\n enter the division of student:";
   cin>>div;
   cout<<"\n enter the name of student:";
   cin>>name;
   cout<<"\n enter the address of student:";
   cin>>add;
   f<<setw(20)<<rn<<setw(20)<<setw(20)<<div<<setw(20)<<name<<setw(20)<<add<<endl;
   f.close();
   cout<<"\n student added successfully!!!";


}
void printstudent()
{
   ifstream f("db.txt");
   string line;
```

```
   cout<<"\n printing data of student:";
   while(getline(f,line))
   {
      cout<<line<<endl;
   }
   cout<<"printing is completed!!";
   f.close();
}
void searchstudent()
{
   ifstream f("db.txt");
   string line,rn;
   cout<<"\n enter the roll to be search:";
   cin>>rn;
   bool found=false;
   while(getline(f,line))
   {
      if(line.find (rn)!=string::npos)
      {
         cout<<"\n student found "<<endl;
         cout<<"student details!!!"<<endl;
         cout<<line<<endl;
         found=true;
         break;
      }
   }
   f.close();
   if(!found)
   {
      cout<<"\n student not found"<<endl;
   }
}
void deletestudent()
{
   ifstream f("db.txt");
   string line,rn;
   string filedata;
   cout<<"\n enter the roll to be deleted:";
   cin>>rn;

   while(getline(f,line))
   {
      if(line.find(rn)==string::npos)
      {

         filedata += line;
         filedata += "\n";
      }
   }
   f.close();
   ofstream f1("db.txt", ios::out);
   f1<<filedata;
   f1.close();
}
int main()
```

```
{
   ofstream f("db.txt",ios::out);
  f<<left<<setw(20)<<"ROLL NO"<<setw(20)<<"DIV"<<setw(20)<<"NAME"<<setw(20)<<"ADDRESS"<<endl;
  f.close();
  int ip;
  while(ip!=-1)
  {
     cout<<"\n enter your choice \n 1.add student\n 2.delete student\n 3.search student\n 4.print
data\n.5.Exit";
     cin>>ip;
     switch(ip)
     {
     case 1:
      addstudent();
      break;
     case 2:
      deletestudent();
      break;
     case 3:
      searchstudent();
      break;
     case 4:
      printstudent();
      break;
     case 5:
      return 0;
      break;
      default:
      cout<<"\n please ReEnter your choice";
      break;
     }


  }

}
```