Q1 Accept prefix expressions. and construct a binary tree and perform recursive and non-

recursive traversals.

#include <iostream>

#include <cstdlib>

#include <cstdio>

#include <cstring>

using namespace std;

class TreeN//node declaration {

   public: char d;

   TreeN \*l, \*r;

   TreeN(char d) {

      this->d = d;

      this->l = NULL;

      this->r = NULL;

   }

};

class StackNod// stack declaration {

   public: TreeN \*treeN;

   StackNod \*n;

   StackNod(TreeN\*treeN)//constructor {

      this->treeN = treeN;

      n = NULL;

   }

};

class ExpressionTree {

   private: StackNod \*top;

   public: ExpressionTree() {

      top = NULL;

   }

   void clear() {

      top = NULL;

   }

   void push(TreeN \*ptr) {

      if (top == NULL)

         top = new StackNod(ptr);

      else {

         StackNod \*nptr = new StackNod(ptr);

         nptr->n = top;

         top = nptr;

      }

   }

   TreeN \*pop() {

      if (top == NULL) {

         cout<<"Underflow"<<endl;

      } else {

         TreeN \*ptr = top->treeN;

         top = top->n;

         return ptr;

      }

   }

   TreeN \*peek() {

      return top->treeN;

   }

   void insert(char val) {

      if (isDigit(val)) {

         TreeN \*nptr = new TreeN(val);

         push(nptr);

      } else if (isOperator(val)) {

         TreeN \*nptr = new TreeN(val);

         nptr->l = pop();

         nptr->r= pop();

         push(nptr);

      } else {

         cout<<"Invalid Expression"<<endl;

         return;

      }

   }

   bool isDigit(char ch) {

      return ch >= '0' && ch <= '9';

   }

   bool isOperator(char ch) {

      return ch == '+' || ch == '-' || ch == '\*' || ch == '/';

   }

   int toDigit(char ch) {

      return ch - '0';

   }

   void buildTree(string eqn) {

      for (int i = eqn.length() - 1; i >= 0; i--)

         insert(eqn[i]);

   }

   void postfix() {

      postOrder(peek());

   }

   void postOrder(TreeN\*ptr) {

      if (ptr != NULL) {

         postOrder(ptr->l);

         postOrder(ptr->r);

         cout<<ptr->d;

      }

   }

   void infix() {

      inOrder(peek());

   }

   void inOrder(TreeN \*ptr) {

      if (ptr != NULL) {

         inOrder(ptr->l);

         cout<<ptr->d;

         inOrder(ptr->r);

      }

   }

   void prefix() {

      preOrder(peek());

   }

   void preOrder(TreeN \*ptr) {

      if (ptr != NULL) {

         cout<<ptr->d;

         preOrder(ptr->l);

         preOrder(ptr->r);

      }

   }

};

int main() {

   string s;

   ExpressionTree et;

   cout<<"\nEnter equation in Prefix form: ";

   cin>>s;

   et.buildTree(s);

   cout<<"\nPrefix : ";

   et.prefix();

   cout<<"\n\nInfix : ";

   et.infix();

   cout<<"\n\nPostfix : ";

   et.postfix();

}

Q2 A Dictionary stores keywords & its meanings.Provide facilty for adding new keywords. deleting keywords.updating values or any entry. Provide a faci ity to display whole data sortedin ascending/ Descending order. Also, find how many maximum comparisons may

require for finding any keyword. Use Binary Search Tree forimplementation.

#include"iostream"  
#include<string.h>  
using namespace std;  
  
typedef struct node  
{  
  
 char k[20];  
 char m[20];  
 class node  \*left;  
 class node \* right;  
}node;  
  
class dict  
{  
public:  
 node \*root;  
 void create();  
 void disp(node \*);  
 void insert(node \* root,node \*temp);  
 int search(node \*,char []);  
 int update(node \*,char []);  
 node\* del(node \*,char []);  
 node \* min(node \*);  
};  
  
void dict :: create()  
{  
 class node \*temp;  
 int ch;  
  
 do  
 {  
  temp = new node;  
  cout<<"\nEnter Keyword:";  
  cin>>temp->k;  
  cout<<"\nEnter Meaning:";  
  cin>>temp->m;  
  
  temp->left = NULL;  
  temp->right = NULL;  
  
  if(root == NULL)  
  {  
   root = temp;  
  }  
  else  
  {  
   insert(root, temp);  
  }  
  cout<<"\nDo u want to add more (y=1/n=0):";  
  cin>>ch;  
 }  
 while(ch == 1);  
  
}  
  
void dict ::  insert(node \* root,node \*temp)  
{  
 if(strcmp (temp->k, root->k) < 0 )  
 {  
  if(root->left == NULL)  
   root->left = temp;  
  else  
   insert(root->left,temp);  
 }  
 else  
 { if(root->right == NULL)  
   root->right = temp;  
  else  
   insert(root->right,temp);  
 }  
  
}  
  
void dict:: disp(node \* root)  
{  
 if( root != NULL)  
 {  
  disp(root->left);  
  cout<<"\n Key Word :"<<root->k;  
  cout<<"\t Meaning :"<<root->m;  
  disp(root->right);  
 }  
}  
  
int dict :: search(node \* root,char k[20])  
{  
 int c=0;  
 while(root != NULL)  
 {  
  c++;  
  if(strcmp (k,root->k) == 0)  
  {  
   cout<<"\nNo of Comparisons:"<<c;  
   return 1;  
  }  
  if(strcmp (k, root->k) < 0)  
   root = root->left;  
  if(strcmp (k, root->k) > 0)  
   root = root->right;  
 }  
  
 return -1;  
}  
int dict :: update(node \* root,char k[20])  
{  
 while(root != NULL)  
 {  
  if(strcmp (k,root->k) == 0)  
  {  
   cout<<"\nEnter New Meaning ofKeyword"<<root->k;  
   cin>>root->m;  
   return 1;  
  }  
  if(strcmp (k, root->k) < 0)  
   root = root->left;  
  if(strcmp (k, root->k) > 0)  
   root = root->right;  
 }  
 return -1;  
}  
node\* dict :: del(node \* root,char k[20])  
{  
 node \*temp;  
  
 if(root == NULL)  
 {  
  cout<<"\nElement No Found";  
  return root;  
 }  
  
 if (strcmp(k,root->k) < 0)  
 {  
  root->left = del(root->left, k);  
  return root;  
 }  
 if (strcmp(k,root->k) > 0)  
 {  
   root->right = del(root->right, k);  
   return root;  
 }  
  
 if (root->right==NULL&&root->left==NULL)  
 {  
  temp = root;  
  delete temp;  
  return NULL;  
  }  
  if(root->right==NULL)  
  {  
  temp = root;  
  root = root->left;  
  delete temp;  
  return root;  
  }  
  else if(root->left==NULL)  
  {  
  temp = root;  
  root = root->right;  
  delete temp;  
  return root;  
  }  
  temp = min(root->right);  
  strcpy(root->k,temp->k);  
  root->right = del(root->right, temp->k);  
  return root;  
  
}  
  
node \* dict :: min(node \*q)  
{  
 while(q->left != NULL)  
 {  
  q = q->left;  
 }  
 return q;  
}  
  
  
  
int main()  
{  
 int ch;  
 dict d;  
 d.root = NULL;  
  
  
 do  
 {  
  cout<<"\nMenu\n1.Create\n2.Disp\n3.Search\n4.Update\n5.Delete\nEnter Ur CH:";  
  cin>>ch;  
  
  switch(ch)  
  {  
case 1: d.create();  
  break;  
case 2: if(d.root == NULL)  
  {  
  cout<<"\nNo any Keyword";  
  }  
  else  
  {  
  d.disp(d.root);  
  }  
  break;  
case 3: if(d.root == NULL)  
 {  
  cout<<"\nDictionary is Empty. First add keywords then try again ";  
 }  
  else  
 {  
  
        cout<<"\nEnter Keyword which u want to search:";  
  char k[20];  
  cin>>k;  
  
  if( d.search(d.root,k) == 1)  
  cout<<"\nKeyword Found";  
  else  
  cout<<"\nKeyword Not Found";  
 }  
  break;  
case 4:  
  if(d.root == NULL)  
  {  
  cout<<"\nDictionary is Empty. First add keywords then try again ";  
 }  
  else  
  {  
  cout<<"\nEnter Keyword which meaning  want to update:";  
  char k[20];  
  cin>>k;  
  if(d.update(d.root,k) == 1)  
  cout<<"\nMeaning Updated";  
  else  
  cout<<"\nMeaning Not Found";  
  }  
  break;  
case 5:  
  if(d.root == NULL)  
  {  
  cout<<"\nDictionary is Empty. First add keywords then try again ";  
  }  
  else  
  {  
  cout<<"\nEnter Keyword which u want to delete:";  
  char k[20];  
  cin>>k;  
  if(d.root == NULL)  
  {  
  cout<<"\nNo any Keyword";  
  }  
  else  
  {  
  d.root = d.del(d.root,k);  
    }  
   }  
  }  
 }  
 while(ch<=5);  
 return 0;  
  
}

Q3 Create a Binary Search tree and find its mirrorImage.Print original& new tree levelwise.

Find height & printleaf nodes.

#include <iostream>

#include <queue>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

};

Node\* createBST(Node\* root, int data) {

if (root == nullptr) {

Node\* node = new Node;

node->data = data;

node->left = node->right = nullptr;

return node;

}

if (data < root->data) {

root->left = createBST(root->left, data);

} else if (data > root->data) {

root->right = createBST(root->right, data);

}

return root;

}

void printLevelOrder(Node\* root) {

if (root == nullptr) {

return;

}

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

int levelSize = q.size();

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

Node\* node = q.front();

q.pop();

cout << node->data << " ";

if (node->left != nullptr) {

q.push(node->left);

}

if (node->right != nullptr) {

q.push(node->right);

}

}

cout << endl;

}

}

Node\* mirrorTree(Node\* root) {

if (root == nullptr) {

return nullptr;

}

Node\* left = mirrorTree(root->left);

Node\* right = mirrorTree(root->right);

root->left = right;

root->right = left;

return root;

}

int height(Node\* root) {

if (root == nullptr) {

return 0;

}

int leftHeight = height(root->left);

int rightHeight = height(root->right);

return max(leftHeight, rightHeight) + 1;

}

void printLeaves(Node\* root) {

if (root == nullptr) {

return;

}

if (root->left == nullptr && root->right == nullptr) {

cout << root->data << " ";

}

printLeaves(root->left);

printLeaves(root->right);

}

int main() {

Node\* root = nullptr;

root = createBST(root, 50);

createBST(root, 30);

createBST(root, 20);

createBST(root, 40);

createBST(root, 70);

createBST(root, 60);

createBST(root, 80);

cout << "Original Tree Level Order: " << endl;

printLevelOrder(root);

Node\* mirroredRoot = mirrorTree(root);

cout << "Mirrored Tree Level Order: " << endl;

printLevelOrder(mirroredRoot);

cout << "Height of the Tree: " << height(root) << endl;

cout << "Leaf Nodes: ";

printLeaves(root);

cout << endl;

return 0;

}

Q4 Create anin-order threaded binary search tree and perform the traversals

#include <iostream>

using namespace std;

struct Node {

int data;

bool leftThread;

bool rightThread;

Node\* left;

Node\* right;

};

Node\* createNode(int data) {

Node\* node = new Node;

node->data = data;

node->leftThread = node->rightThread = true;

node->left = node->right = nullptr;

return node;

}

Node\* insert(Node\* root, int data) {

if (root == nullptr) {

return createNode(data);

}

if (data < root->data) {

Node\* left = insert(root->left, data);

root->left = left;

if (left != nullptr) {

root->leftThread = false;

}

} else if (data > root->data) {

Node\* right = insert(root->right, data);

root->right = right;

if (right != nullptr) {

root->rightThread = false;

}

}

return root;

}

Node\* leftMostNode(Node\* node) {

while (node->left != nullptr) {

node = node->left;

}

return node;

}

void inOrderTraversal(Node\* root) {

Node\* current = leftMostNode(root);

while (current != nullptr) {

cout << current->data << " ";

if (current->rightThread) {

current = current->right;

} else {

current = leftMostNode(current->right);

}

}

}

void preOrderTraversal(Node\* root) {

while (root != nullptr) {

cout << root->data << " ";

if (!root->leftThread) {

root = root->left;

} else if (!root->rightThread) {

root = root->right;

} else {

while (root != nullptr && root->rightThread) {

root = root->right;

}

if (root != nullptr) {

root = root->right;

}

}

}

}

void postOrderTraversal(Node\* root) {

Node\* current = leftMostNode(root);

while (current != nullptr) {

if (current->rightThread) {

current = current->right;

} else {

current = leftMostNode(current->right);

}

while (current != nullptr && (current->leftThread || current->right == root)) {

cout << current->data << " ";

root = current;

if (current->rightThread) {

current = current->right;

} else {

current = leftMostNode(current->right);

}

}

}

}

int main() {

Node\* root = nullptr;

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

cout << "In-Order Traversal: ";

inOrderTraversal(root);

cout << endl;

cout << "Pre-Order Traversal: ";

preOrderTraversal(root);

cout << endl;

cout << "Post-Order Traversal: ";

postOrderTraversal(root);

cout << endl;

return 0;

}

Q5 Represent a given graph using an adjacency lst and perform DFS or BFS.

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void dfs(vector<int> adj[], int u, bool visited[]) {

visited[u] = true;

cout << u << " ";

for (int v : adj[u]) {

if (!visited[v]) {

dfs(adj, v, visited);

}

}

}

void bfs(vector<int> adj[], int u, bool visited[]) {

queue<int> q;

visited[u] = true;

q.push(u);

while (!q.empty()) {

int u = q.front();

q.pop();

cout << u << " ";

for (int v : adj[u]) {

if (!visited[v]) {

visited[v] = true;

q.push(v);

}

}

}

}

int main() {

int V = 5;

vector<int> adj[V];

adj[0].push\_back(1);

adj[0].push\_back(4);

adj[1].push\_back(0);

adj[1].push\_back(2);

adj[1].push\_back(3);

adj[1].push\_back(4);

adj[2].push\_back(1);

adj[2].push\_back(3);

adj[3].push\_back(1);

adj[3].push\_back(2);

adj[3].push\_back(4);

adj[4].push\_back(0);

adj[4].push\_back(1);

adj[4].push\_back(3);

bool visited[V] = { false };

cout << "DFS: ";

dfs(adj, 0, visited);

cout << endl;

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

visited[i] = false;

}

cout << "BFS: ";

bfs(adj, 0, visited);

cout << endl;

return 0;

}

Q6 represent a given graph using an adjacency list or array and find the shortest path using dijkstra's algorithm.

#include <iostream>

#include <vector>

#include <queue>

#include <limits.h>

using namespace std;

#define V 6 // number of vertices

void dijkstra(vector<pair<int, int>> adj[], int src) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

vector<int> dist(V, INT\_MAX);

pq.push(make\_pair(0, src));

dist[src] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto i : adj[u]) {

int v = i.first;

int weight = i.second;

if (dist[v] > dist[u] + weight) {

dist[v] = dist[u] + weight;

pq.push(make\_pair(dist[v], v));

}

}

}

cout << "Vertex \t Distance from Source\n";

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

cout << i << "\t\t" << dist[i] << endl;

}

}

int main() {

vector<pair<int, int>> adj[V];

adj[0].push\_back(make\_pair(1, 4));

adj[0].push\_back(make\_pair(2, 2));

adj[1].push\_back(make\_pair(2, 3));

adj[1].push\_back(make\_pair(3, 2));

adj[1].push\_back(make\_pair(4, 3));

adj[2].push\_back(make\_pair(1, 1));

adj[2].push\_back(make\_pair(3, 4));

adj[3].push\_back(make\_pair(4, 2));

adj[4].push\_back(make\_pair(5, 3));

dijkstra(adj, 0);

return 0;

}

Q7 Represent a given graph using an adjacency list or array and generate a minimum spanning tree using Kruskal's and algorithm in python

# Python program for Kruskal's algorithm to find

# Minimum Spanning Tree of a given connected,

# undirected and weighted graph

# Class to represent a graph

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

# Function to add an edge to graph

def addEdge(self, u, v, w):

self.graph.append([u, v, w])

# A utility function to find set of an element i

# (truly uses path compression technique)

def find(self, parent, i):

if parent[i] != i:

# Reassignment of node's parent

# to root node as

# path compression requires

parent[i] = self.find(parent, parent[i])

return parent[i]

# A function that does union of two sets of x and y

# (uses union by rank)

def union(self, parent, rank, x, y):

# Attach smaller rank tree under root of

# high rank tree (Union by Rank)

if rank[x] < rank[y]:

parent[x] = y

elif rank[x] > rank[y]:

parent[y] = x

# If ranks are same, then make one as root

# and increment its rank by one

else:

parent[y] = x

rank[x] += 1

# The main function to construct MST

# using Kruskal's algorithm

def KruskalMST(self):

# This will store the resultant MST

result = []

# An index variable, used for sorted edges

i = 0

# An index variable, used for result[]

e = 0

# Sort all the edges in

# non-decreasing order of their

# weight

self.graph = sorted(self.graph,

key=lambda item: item[2])

parent = []

rank = []

# Create V subsets with single elements

for node in range(self.V):

parent.append(node)

rank.append(0)

# Number of edges to be taken is less than to V-1

while e < self.V - 1:

# Pick the smallest edge and increment

# the index for next iteration

u, v, w = self.graph[i]

i = i + 1

x = self.find(parent, u)

y = self.find(parent, v)

# If including this edge doesn't

# cause cycle, then include it in result

# and increment the index of result

# for next edge

if x != y:

e = e + 1

result.append([u, v, w])

self.union(parent, rank, x, y)

# Else discard the edge

minimumCost = 0

print("Edges in the constructed MST")

for u, v, weight in result:

minimumCost += weight

print("%d -- %d == %d" % (u, v, weight))

print("Minimum Spanning Tree", minimumCost)

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

g = Graph(4)

g.addEdge(0, 1, 10)

g.addEdge(0, 2, 6)

g.addEdge(0, 3, 5)

g.addEdge(1, 3, 15)

g.addEdge(2, 3, 4)

# Function call

g.KruskalMST()

Q7 B Represent a given graph using an adjacency list or array and generate a minimum spanning tree using Prim's algorithm in python

# A Python program for Prims's MST for

# adjacency list representation of graph

from collections import defaultdict

import sys

class Heap():

def \_\_init\_\_(self):

self.array = []

self.size = 0

self.pos = []

def newMinHeapNode(self, v, dist):

minHeapNode = [v, dist]

return minHeapNode

# A utility function to swap two nodes of

# min heap. Needed for min heapify

def swapMinHeapNode(self, a, b):

t = self.array[a]

self.array[a] = self.array[b]

self.array[b] = t

# A standard function to heapify at given idx

# This function also updates position of nodes

# when they are swapped. Position is needed

# for decreaseKey()

def minHeapify(self, idx):

smallest = idx

left = 2 \* idx + 1

right = 2 \* idx + 2

if left < self.size and self.array[left][1] < \

self.array[smallest][1]:

smallest = left

if right < self.size and self.array[right][1] < \

self.array[smallest][1]:

smallest = right

# The nodes to be swapped in min heap

# if idx is not smallest

if smallest != idx:

# Swap positions

self.pos[self.array[smallest][0]] = idx

self.pos[self.array[idx][0]] = smallest

# Swap nodes

self.swapMinHeapNode(smallest, idx)

self.minHeapify(smallest)

# Standard function to extract minimum node from heap

def extractMin(self):

# Return NULL wif heap is empty

if self.isEmpty() == True:

return

# Store the root node

root = self.array[0]

# Replace root node with last node

lastNode = self.array[self.size - 1]

self.array[0] = lastNode

# Update position of last node

self.pos[lastNode[0]] = 0

self.pos[root[0]] = self.size - 1

# Reduce heap size and heapify root

self.size -= 1

self.minHeapify(0)

return root

def isEmpty(self):

return True if self.size == 0 else False

def decreaseKey(self, v, dist):

# Get the index of v in heap array

i = self.pos[v]

# Get the node and update its dist value

self.array[i][1] = dist

# Travel up while the complete tree is not

# heapified. This is a O(Logn) loop

while i > 0 and self.array[i][1] < \

self.array[(i - 1) // 2][1]:

# Swap this node with its parent

self.pos[self.array[i][0]] = (i-1)/2

self.pos[self.array[(i-1)//2][0]] = i

self.swapMinHeapNode(i, (i - 1)//2)

# move to parent index

i = (i - 1) // 2

# A utility function to check if a given vertex

# 'v' is in min heap or not

def isInMinHeap(self, v):

if self.pos[v] < self.size:

return True

return False

def printArr(parent, n):

for i in range(1, n):

print("% d - % d" % (parent[i], i))

class Graph():

def \_\_init\_\_(self, V):

self.V = V

self.graph = defaultdict(list)

# Adds an edge to an undirected graph

def addEdge(self, src, dest, weight):

# Add an edge from src to dest. A new node is

# added to the adjacency list of src. The node

# is added at the beginning. The first element of

# the node has the destination and the second

# elements has the weight

newNode = [dest, weight]

self.graph[src].insert(0, newNode)

# Since graph is undirected, add an edge from

# dest to src also

newNode = [src, weight]

self.graph[dest].insert(0, newNode)

# The main function that prints the Minimum

# Spanning Tree(MST) using the Prim's Algorithm.

# It is a O(ELogV) function

def PrimMST(self):

# Get the number of vertices in graph

V = self.V

# key values used to pick minimum weight edge in cut

key = []

# List to store constructed MST

parent = []

# minHeap represents set E

minHeap = Heap()

# Initialize min heap with all vertices. Key values of all

# vertices (except the 0th vertex) is initially infinite

for v in range(V):

parent.append(-1)

key.append(1e7)

minHeap.array.append(minHeap.newMinHeapNode(v, key[v]))

minHeap.pos.append(v)

# Make key value of 0th vertex as 0 so

# that it is extracted first

minHeap.pos[0] = 0

key[0] = 0

minHeap.decreaseKey(0, key[0])

# Initially size of min heap is equal to V

minHeap.size = V

# In the following loop, min heap contains all nodes

# not yet added in the MST.

while minHeap.isEmpty() == False:

# Extract the vertex with minimum distance value

newHeapNode = minHeap.extractMin()

u = newHeapNode[0]

# Traverse through all adjacent vertices of u

# (the extracted vertex) and update their

# distance values

for pCrawl in self.graph[u]:

v = pCrawl[0]

# If shortest distance to v is not finalized

# yet, and distance to v through u is less than

# its previously calculated distance

if minHeap.isInMinHeap(v) and pCrawl[1] < key[v]:

key[v] = pCrawl[1]

parent[v] = u

# update distance value in min heap also

minHeap.decreaseKey(v, key[v])

printArr(parent, V)

# Driver program to test the above functions

graph = Graph(9)

graph.addEdge(0, 1, 4)

graph.addEdge(0, 7, 8)

graph.addEdge(1, 2, 8)

graph.addEdge(1, 7, 11)

graph.addEdge(2, 3, 7)

graph.addEdge(2, 8, 2)

graph.addEdge(2, 5, 4)

graph.addEdge(3, 4, 9)

graph.addEdge(3, 5, 14)

graph.addEdge(4, 5, 10)

graph.addEdge(5, 6, 2)

graph.addEdge(6, 7, 1)

graph.addEdge(6, 8, 6)

graph.addEdge(7, 8, 7)

graph.PrimMST()

Q8 Create a hash table and handle the collisions using linear probing with or without

replacement.

# Our own Hashnode class

class HashNode:

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

# Our own Hashmap class

class HashMap:

# hash element array

def \_\_init\_\_(self):

self.capacity = 20

self.size = 0

self.arr = [None] \* self.capacity

# dummy node

self.dummy = HashNode(-1, -1)

# This implements hash function to find index for a key

def hashCode(self, key):

return key % self.capacity

# Function to add key value pair

def insertNode(self, key, value):

temp = HashNode(key, value)

# Apply hash function to find index for given key

hashIndex = self.hashCode(key)

# find next free space

while self.arr[hashIndex] is not None and self.arr[hashIndex].key != key and self.arr[hashIndex].key != -1:

hashIndex += 1

hashIndex %= self.capacity

# if new node to be inserted, increase the current size

if self.arr[hashIndex] is None or self.arr[hashIndex].key == -1:

self.size += 1

self.arr[hashIndex] = temp

# Function to delete a key value pair

def deleteNode(self, key):

# Apply hash function to find index for given key

hashIndex = self.hashCode(key)

# finding the node with given key

while self.arr[hashIndex] is not None:

# if node found

if self.arr[hashIndex].key == key:

temp = self.arr[hashIndex]

# Insert dummy node here for further use

self.arr[hashIndex] = self.dummy

# Reduce size

self.size -= 1

return temp.value

hashIndex += 1

hashIndex %= self.capacity

# If not found return None

return None

# Function to search the value for a given key

def get(self, key):

# Apply hash function to find index for given key

hashIndex = self.hashCode(key)

counter = 0

# finding the node with given key

while self.arr[hashIndex] is not None:

# If counter is greater than capacity to avoid infinite loop

if counter > self.capacity:

return None

# if node found return its value

if self.arr[hashIndex].key == key:

return self.arr[hashIndex].value

hashIndex += 1

hashIndex %= self.capacity

counter += 1

# If not found return None

return 0

# Return current size

def sizeofMap(self):

return self.size

# Return true if size is 0

def isEmpty(self):

return self.size == 0

# Function to display the stored key value pairs

def display(self):

for i in range(self.capacity):

if self.arr[i] is not None and self.arr[i].key != -1:

print("key = ", self.arr[i].key, " value = ", self.arr[i].value)

# Driver method to test map class

if \_\_name\_\_ == "\_\_main\_\_":

h = HashMap()

h.insertNode(1, 1)

h.insertNode(2, 2)

h.insertNode(2, 3)

h.display()

print(h.sizeofMap())

print(h.deleteNode(2))

print(h.sizeofMap())

print(h.isEmpty())

print(h.get(2))

Q9 Implementation of simple index file.

# Python program for Indexed

# Sequential Search

def indexedSequentialSearch(arr, n, k):

elements = [0] \* 20

indices = [0] \* 20

j, ind, start, end = 0, 0, 0, 0

set\_flag = 0

for i in range(0, n, 3):

# Storing element

elements[ind] = arr[i]

# Storing the index

indices[ind] = i

ind += 1

if k < elements[0]:

print("Not found")

exit(0)

else:

for i in range(1, ind + 1):

if k <= elements[i]:

start = indices[i - 1]

end = indices[i]

set\_flag = 1

break

if set\_flag == 0:

start = indices[i-1]

end = n

for i in range(start, end + 1):

if k == arr[i]:

j = 1

break

if j == 1:

print("Found at index", i)

else:

print("Not found")

# Driver code

if \_\_name\_\_ == "\_\_main\_\_":

arr = [6, 7, 8, 9, 10]

n = len(arr)

# Element to search

k = 8

# Function call

indexedSequentialSearch(arr, n, k)

Q 10 Company maintains employee information such as employee ID, name.designation and salary. Allow users to add,deleteinformation about employees.Displayinformation of a particular employee. If an employee does not exist.an appropriate message is displayed.If

itis, then the system displays the employee details.Use a sequential file to maintain the

data.

import csv

# Function to add employee information to the file

def add\_employee():

emp\_id = input("Enter employee ID: ")

name = input("Enter employee name: ")

designation = input("Enter employee designation: ")

salary = input("Enter employee salary: ")

with open('employee\_data.csv', mode='a', newline='') as file:

writer = csv.writer(file)

writer.writerow([emp\_id, name, designation, salary])

print("Employee added successfully.")

# Function to delete employee information from the file

def delete\_employee():

emp\_id = input("Enter employee ID to delete: ")

with open('employee\_data.csv', mode='r') as file:

reader = csv.reader(file)

rows = list(reader)

found = False

with open('employee\_data.csv', mode='w', newline='') as file:

writer = csv.writer(file)

for row in rows:

if row[0] == emp\_id:

found = True

continue

writer.writerow(row)

if found:

print("Employee deleted successfully.")

else:

print("Employee not found.")

# Function to display employee information

def display\_employee():

emp\_id = input("Enter employee ID to display: ")

with open('employee\_data.csv', mode='r') as file:

reader = csv.reader(file)

for row in reader:

if row[0] == emp\_id:

print(f"Employee ID: {row[0]}")

print(f"Name: {row[1]}")

print(f"Designation: {row[2]}")

print(f"Salary: {row[3]}")

break

else:

print("Employee not found.")

# Main program loop

while True:

print("\nEmployee Management System")

print("1. Add Employee")

print("2. Delete Employee")

print("3. Display Employee")

print("4. Exit")

choice = input("Enter your choice: ")

if choice == '1':

add\_employee()

elif choice == '2':

delete\_employee()

elif choice == '3':

display\_employee()

elif choice == '4':

break

else:

print("Invalid choice.")

Q 11 A Dictionary stores keywords &its meanings. Provide faciity for adding new keywords, deleting keywords. updating values of any entry. Provide a facility to display whole data sortedin ascending/ Descending order. Also, find how many maximum comparisons may require for finding any keyword.Use Height balanced tree andfind the complexity for finding a keyword in python

from collections import OrderedDict

# Function to add a new keyword and its meaning

def add\_keyword(dictionary):

keyword = input("Enter the keyword: ")

meaning = input("Enter the meaning: ")

dictionary[keyword] = meaning

print("Keyword added successfully.")

# Function to delete a keyword

def delete\_keyword(dictionary):

keyword = input("Enter the keyword to delete: ")

if keyword in dictionary:

del dictionary[keyword]

print("Keyword deleted successfully.")

else:

print("Keyword not found.")

# Function to update the meaning of a keyword

def update\_keyword(dictionary):

keyword = input("Enter the keyword to update: ")

if keyword in dictionary:

meaning = input("Enter the new meaning: ")

dictionary[keyword] = meaning

print("Keyword updated successfully.")

else:

print("Keyword not found.")

# Function to display the keywords and their meanings

def display\_dictionary(dictionary, reverse\_order):

sorted\_dict = OrderedDict(sorted(dictionary.items(), key=lambda x: x[0], reverse=reverse\_order))

print("\nKeywords and their meanings:")

for keyword, meaning in sorted\_dict.items():

print(keyword, ":", meaning)

# Main program loop

dictionary = {}

while True:

print("\nKeyword Dictionary Management System")

print("1. Add Keyword")

print("2. Delete Keyword")

print("3. Update Keyword")

print("4. Display Dictionary in Ascending Order")

print("5. Display Dictionary in Descending Order")

print("6. Exit")

choice = input("Enter your choice: ")

if choice == '1':

add\_keyword(dictionary)

elif choice == '2':

delete\_keyword(dictionary)

elif choice == '3':

update\_keyword(dictionary)

elif choice == '4':

display\_dictionary(dictionary, False)

elif choice == '5':

display\_dictionary(dictionary, True)

elif choice == '6':

break

else:

print("Invalid choice.")

Q12 Implement Heap sort in c++

#include <iostream>

#include <vector>

using namespace std;

void heapify(vector<int>& arr, int n, int i) {

int largest = i; // Initialize largest as root

int l = 2 \* i + 1; // left = 2\*i + 1

int r = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (l < n && arr[l] > arr[largest])

largest = l;

// If right child is larger than largest so far

if (r < n && arr[r] > arr[largest])

largest = r;

// If largest is not root

if (largest != i) {

swap(arr[i], arr[largest]);

// Recursively heapify the affected sub-tree

heapify(arr, n, largest);

}

}

void heap\_sort(vector<int>& arr) {

int n = arr.size();

// Build heap (rearrange array)

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

heapify(arr, n, i);

// One by one extract an element from heap

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

// Move current root to end

swap(arr[0], arr[i]);

// call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

int main() {

vector<int> arr = {12, 11, 13, 5, 6, 7};

heap\_sort(arr);

cout << "Sorted array is: ";

for (int i = 0; i < arr.size(); ++i) {

cout << arr[i] << " ";

}

cout << endl;

return 0;

}