**Assignment no 10**

#include <iostream>

#include <vector>

#include <algorithm>

// Function to build a max heap

void buildMaxHeap(std::vector<int>& heap) {

int n = heap.size();

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

int parent = i;

int leftChild = 2 \* parent + 1;

int rightChild = 2 \* parent + 2;

int maxIndex = parent;

if (leftChild < n && heap[leftChild] > heap[maxIndex])

maxIndex = leftChild;

if (rightChild < n && heap[rightChild] > heap[maxIndex])

maxIndex = rightChild;

if (maxIndex != parent) {

std::swap(heap[parent], heap[maxIndex]);

buildMaxHeap(heap);

}

}

}

// Function to find the maximum and minimum marks

void findMinMaxMarks(const std::vector<int>& marks, int& maxMarks, int& minMarks) {

std::vector<int> heap = marks;

buildMaxHeap(heap);

maxMarks = heap[0];

minMarks = heap.back();

}

int main() {

int numStudents;

std::cout << "Enter the number of students: ";

std::cin >> numStudents;

std::vector<int> marks(numStudents);

std::cout << "Enter the marks obtained by students:\n";

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

std::cout << "Student " << i + 1 << ": ";

std::cin >> marks[i];

}

int maxMarks, minMarks;

findMinMaxMarks(marks, maxMarks, minMarks);

std::cout << "Maximum marks obtained: " << maxMarks << std::endl;

std::cout << "Minimum marks obtained: " << minMarks << std::endl;

return 0;

}

**Output**

[?2004l

Enter the number of students: 3

Enter the marks obtained by students:

Student 1: 80

Student 2: 90

Student 3: 98

Maximum marks obtained: 98

Minimum marks obtained: 80

[?2004h

**Assignment no 2**

class KeyValuePair:

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

self.key = key

self.value = value

class Dictionary:

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

self.size = size

self.table = [[] for \_ in range(size)]

def hash\_function(self, key):

return hash(key) % self.size

def insert(self, key, value):

index = self.hash\_function(key)

chain = self.table[index]

for pair in chain:

if pair.key == key:

pair.value = value # Update existing key-value pair

return

chain.append(KeyValuePair(key, value)) # Add new key-value pair

def find(self, key):

index = self.hash\_function(key)

chain = self.table[index]

for pair in chain:

if pair.key == key:

return pair.value

return None # Key not found

def delete(self, key):

index = self.hash\_function(key)

chain = self.table[index]

for pair in chain:

if pair.key == key:

chain.remove(pair) # Remove key-value pair

return

def display(self):

for i in range(self.size):

chain = self.table[i]

if chain:

for pair in chain:

print(f"Key: {pair.key}, Value: {pair.value}")

# Testing the dictionary implementation

dict\_obj = Dictionary(10)

dict\_obj.insert("apple", "A sweet fruit")

dict\_obj.insert("banana", "A tasty fruit")

dict\_obj.insert("orange", "A citrus fruit")

print("Dictionary after insertion:")

dict\_obj.display()

print("\nValue of 'apple':", dict\_obj.find("apple"))

print("Value of 'grape':", dict\_obj.find("grape"))

dict\_obj.delete("banana")

print("\nDictionary after deletion:")

dict\_obj.display()

**output**

[?2004l

Dictionary after insertion:

Key: banana, Value: A tasty fruit

Key: apple, Value: A sweet fruit

Key: orange, Value: A citrus fruit

Value of 'apple': A sweet fruit

Value of 'grape': None

Dictionary after deletion:

Key: apple, Value: A sweet fruit

Key: orange, Value: A citrus fruit

[?2004h

**Assignment no 3**

class Node:

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

self.data = data

self.children = []

def add\_child(self, child):

self.children.append(child)

def construct\_tree():

book = Node("Book")

# Chapter 1

chapter1 = Node("Chapter 1")

section1\_1 = Node("Section 1.1")

section1\_2 = Node("Section 1.2")

section1\_3 = Node("Section 1.3")

chapter1.add\_child(section1\_1)

chapter1.add\_child(section1\_2)

chapter1.add\_child(section1\_3)

# Chapter 2

chapter2 = Node("Chapter 2")

section2\_1 = Node("Section 2.1")

subsection2\_1\_1 = Node("Subsection 2.1.1")

subsection2\_1\_2 = Node("Subsection 2.1.2")

section2\_2 = Node("Section 2.2")

section2\_1.add\_child(subsection2\_1\_1)

section2\_1.add\_child(subsection2\_1\_2)

chapter2.add\_child(section2\_1)

chapter2.add\_child(section2\_2)

# Chapter 3

chapter3 = Node("Chapter 3")

section3\_1 = Node("Section 3.1")

chapter3.add\_child(section3\_1)

# Add chapters to the book

book.add\_child(chapter1)

book.add\_child(chapter2)

book.add\_child(chapter3)

return book

def print\_nodes(node, level=0):

print(" " \* level + node.data)

for child in node.children:

print\_nodes(child, level + 1)

def calculate\_requirements(node):

# Base case: empty tree

if node is None:

return 0, 0

# Calculate time and space requirements recursively

time = 1 # Counting the current node

space = 1

for child in node.children:

child\_time, child\_space = calculate\_requirements(child)

time += child\_time

space = max(space, child\_space)

return time, space

# Construct the tree

book\_tree = construct\_tree()

# Print the nodes of the tree

print("Nodes of the tree:")

print\_nodes(book\_tree)

# Calculate time and space requirements

time\_req, space\_req = calculate\_requirements(book\_tree)

print("Time requirement:", time\_req)

print("Space requirement:", space\_req)

**output**

[?2004l

Nodes of the tree:

Book

Chapter 1

Section 1.1

Section 1.2

Section 1.3

Chapter 2

Section 2.1

Subsection 2.1.1

Subsection 2.1.2

Section 2.2

Chapter 3

Section 3.1

Time requirement: 12

Space requirement: 1

[?2004h

**Assignment no 8**

#include <iostream>

using namespace std;

struct Node {

int key;

double searchProbability;

Node\* left;

Node\* right;

Node(int k, double p) {

key = k;

searchProbability = p;

left = right = nullptr;

}

};

Node\* insert(Node\* root, int key, double probability) {

if (root == nullptr)

return new Node(key, probability);

if (key < root->key)

root->left = insert(root->left, key, probability);

else if (key > root->key)

root->right = insert(root->right, key, probability);

return root;

}

double calculateSearchCost(Node\* root, double cumulativeProbability) {

if (root == nullptr)

return 0.0;

double leftCost = calculateSearchCost(root->left, cumulativeProbability + root->searchProbability);

double rightCost = calculateSearchCost(root->right, cumulativeProbability + root->searchProbability);

return cumulativeProbability + root->searchProbability + leftCost + rightCost;

}

int main() {

int keys[] = { 2, 4, 5, 6, 8, 10 };

double probabilities[] = { 0.2, 0.1, 0.15, 0.2, 0.05, 0.2 };

int size = sizeof(keys) / sizeof(keys[0]);

Node\* root = nullptr;

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

root = insert(root, keys[i], probabilities[i]);

}

double searchCost = calculateSearchCost(root, 0.0);

cout << "The binary search tree has been built." << endl;

cout << "The least search cost is: " << searchCost << endl;

return 0;

}

**Output**

[?2004l

The binary search tree has been built.

The least search cost is: 3.2

[?2004h

**Assignment no 7**

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

**Output**

[?2004l

Enter no. of vertices: 3

Do you want an edge between 1 and 2: y

Enter weight: 33

Do you want an edge between 1 and 3: y

Enter weight: 22

Do you want an edge between 2 and 3: y

Enter weight: 4

1->3

3->2

Min. Cost: 26[?2004h