**Experiment No. – 3.2**

**Student Name: Deepak Saini UID: 20BCS4066**

**Branch: 20BCC1 Section/Group: A**

**Semester:**  **5th Date of Performance: 11/10/2022**

**Subject Name: ADVANCED PROGRAMMING LAB**

**Subject Code: 20CSP-334**

1. **Aim/Overview of the practical:**

Demonstrate insert ,delete and search in Treap.

1. **Task to be done:**

Demonstrate insert ,delete and search in Treap.

1. **Steps for practical**:

**Insert**

1) Create new node with key equals to x and value equals to a random value.

2) Perform standard BST insert.

3) A newly inserted node gets a random priority, so Max-Heap property may be violated.. Use rotations to make sure that inserted node’s priority follows max heap property.

During insertion, we recursively traverse all ancestors of the inserted node.

a) If new node is inserted in left subtree and root of left subtree has higher priority, perform right rotation.

b) If new node is inserted in right subtree and root of right subtree has higher priority, perform left rotation.

Delete:

The delete implementation here is slightly different from the steps discussed in previous post.

1) If node is a leaf, delete it.

2) If node has one child NULL and other as non-NULL, replace node with the non-empty child.

3) If node has both children as non-NULL, find max of left and right children.

….a) If priority of right child is greater, perform left rotation at node

….b) If priority of left child is greater, perform right rotation at node.

The idea of step 3 is to move the node to down so that we end up with either case 1 or case 2.

**4 . Code:**

// C++ program to demonstrate search, insert and delete in Treap

#include <bits/stdc++.h>

using namespace std;

// A Treap Node

struct TreapNode

{

int key, priority;

TreapNode \*left, \*right;

};

/\* T1, T2 and T3 are subtrees of the tree rooted with y

(on left side) or x (on right side)

y x

/ \ Right Rotation / \

x T3 – – – – – – – > T1 y

/ \ < - - - - - - - / \

T1 T2 Left Rotation T2 T3 \*/

// A utility function to right rotate subtree rooted with y

// See the diagram given above.

TreapNode \*rightRotate(TreapNode \*y)

{

TreapNode \*x = y->left, \*T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Return new root

return x;

}

// A utility function to left rotate subtree rooted with x

// See the diagram given above.

TreapNode \*leftRotate(TreapNode \*x)

{

TreapNode \*y = x->right, \*T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Return new root

return y;

}

/\* Utility function to add a new key \*/

TreapNode\* newNode(int key)

{

TreapNode\* temp = new TreapNode;

temp->key = key;

temp->priority = rand()%100;

temp->left = temp->right = NULL;

return temp;

}

// C function to search a given key in a given BST

TreapNode\* search(TreapNode\* root, int key)

{

// Base Cases: root is null or key is present at root

if (root == NULL || root->key == key)

return root;

// Key is greater than root's key

if (root->key < key)

return search(root->right, key);

// Key is smaller than root's key

return search(root->left, key);

}

/\* Recursive implementation of insertion in Treap \*/

TreapNode\* insert(TreapNode\* root, int key)

{

// If root is NULL, create a new node and return it

if (!root)

return newNode(key);

// If key is smaller than root

if (key <= root->key)

{

// Insert in left subtree

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

// Fix Heap property if it is violated

if (root->left->priority > root->priority)

root = rightRotate(root);

}

else // If key is greater

{

// Insert in right subtree

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

// Fix Heap property if it is violated

if (root->right->priority > root->priority)

root = leftRotate(root);

}

return root;

}

/\* Recursive implementation of Delete() \*/

TreapNode\* deleteNode(TreapNode\* root, int key)

{

if (root == NULL)

return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

// IF KEY IS AT ROOT

// If left is NULL

else if (root->left == NULL)

{

TreapNode \*temp = root->right;

delete(root);

root = temp; // Make right child as root

}

// If Right is NULL

else if (root->right == NULL)

{

TreapNode \*temp = root->left;

delete(root);

root = temp; // Make left child as root

}

// If key is at root and both left and right are not NULL

else if (root->left->priority < root->right->priority)

{

root = leftRotate(root);

root->left = deleteNode(root->left, key);

}

else

{

root = rightRotate(root);

root->right = deleteNode(root->right, key);

}

return root;

}

// A utility function to print tree

void inorder(TreapNode\* root)

{

if (root)

{

inorder(root->left);

cout << "key: "<< root->key << " | priority: %d "

<< root->priority;

if (root->left)

cout << " | left child: " << root->left->key;

if (root->right)

cout << " | right child: " << root->right->key;

cout << endl;

inorder(root->right);

}

}

// Driver Program to test above functions

int main()

{

srand(time(NULL));

struct TreapNode \*root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout << "Inorder traversal of the given tree \n";

inorder(root);

cout << "\nDelete 20\n";

root = deleteNode(root, 20);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 30\n";

root = deleteNode(root, 30);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

cout << "\nDelete 50\n";

root = deleteNode(root, 50);

cout << "Inorder traversal of the modified tree \n";

inorder(root);

TreapNode \*res = search(root, 50);

(res == NULL)? cout << "\n50 Not Found ":

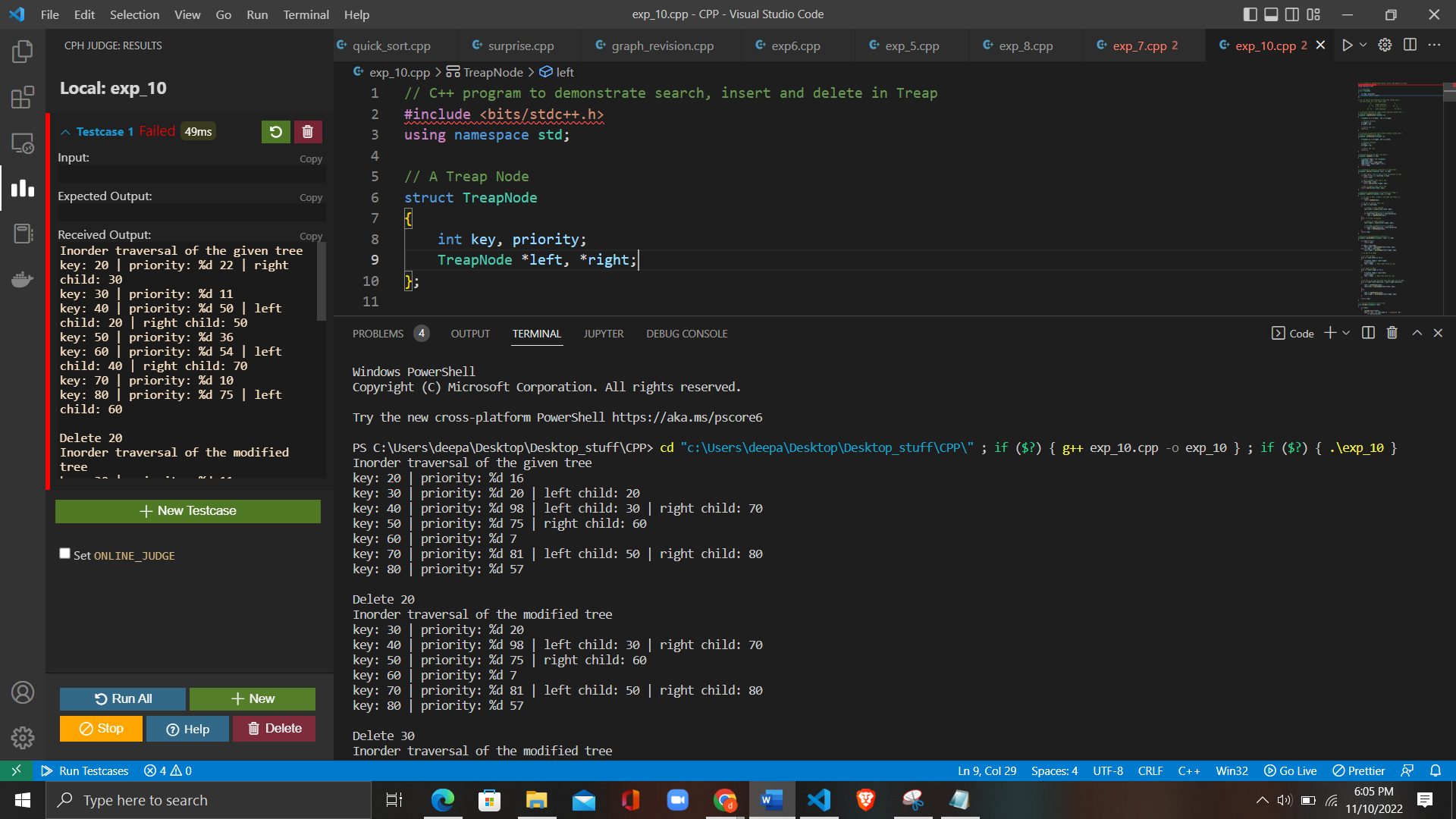
cout << "\n50 found";

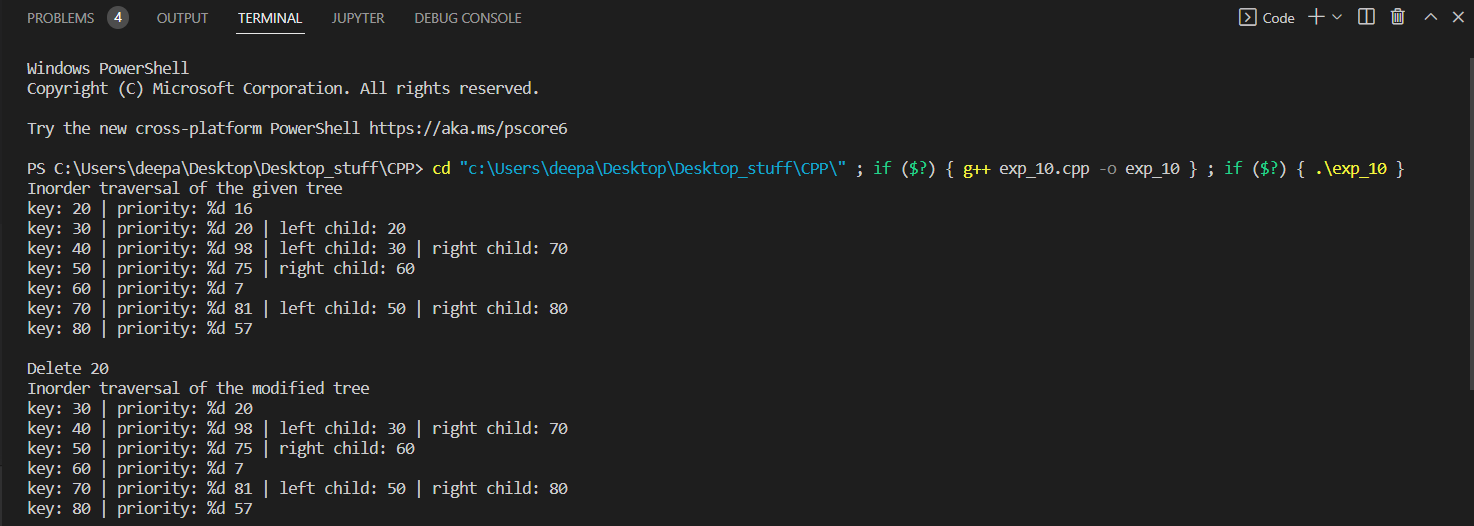
return 0;

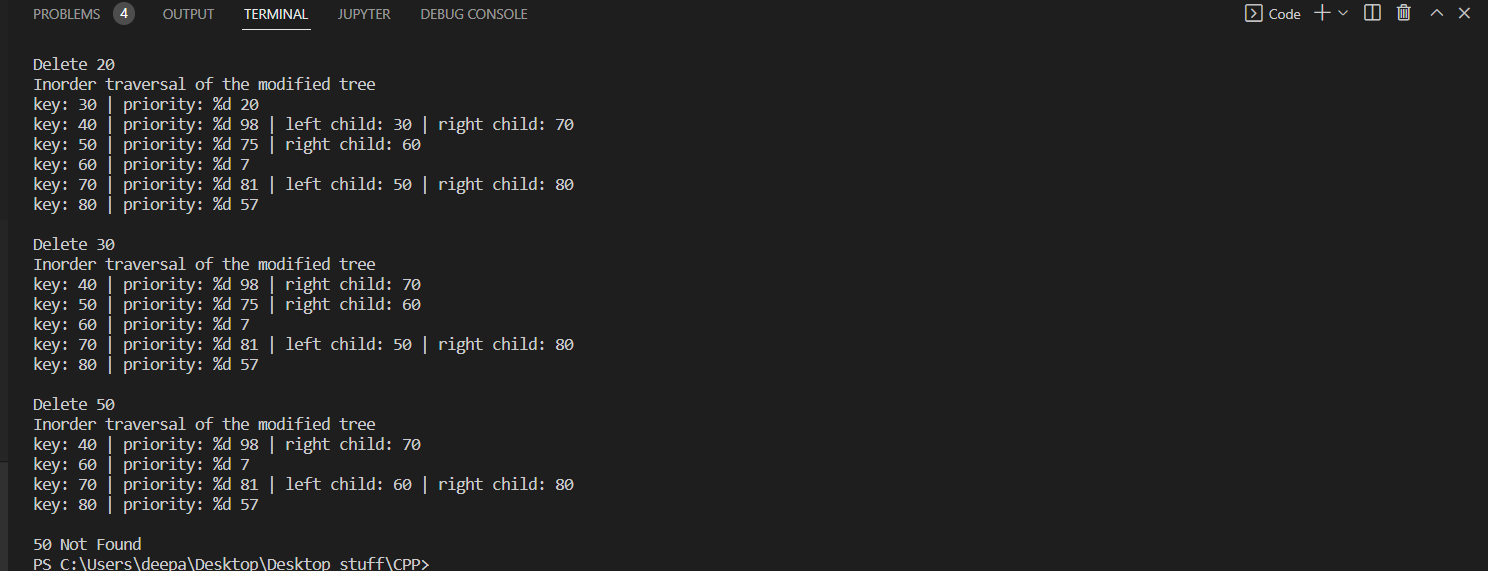
}

**5.** **Output:**

**a)**







**6. Learning Outcomes:**

* To learn the basics of Graph to how to take inputs.
* To learn the approach to how to solve problems related to graph.
* To learn about how to use stack data structure.
* To solve the problems using Dynamic Programming.

**Evaluation Grid (To be created as per the SOP and Assessment guidelines by the faculty):**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. No. | Parameters | Marks Obtained | Maximum Marks |
| 1. |  |  |  |
| 2. |  |  |  |
| 3. |  |  |  |