**Raahat Arora (Week 6)**

**230957216**

**Roll no – 72**

**Q1-**

#include <iostream>

#include <algorithm>

using namespace std;

// Definition for a binary tree node

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

// Function to calculate the height of a tree

int height(TreeNode\* root, int &diameter) {

if (root == nullptr) {

return 0;

}

// Recursively find the height of the left and right subtrees

int leftHeight = height(root->left, diameter);

int rightHeight = height(root->right, diameter);

// Update the diameter (longest path between any two nodes)

diameter = max(diameter, leftHeight + rightHeight);

// Return the height of the current subtree

return 1 + max(leftHeight, rightHeight);

}

// Function to calculate the diameter of the binary tree

int diameterOfBinaryTree(TreeNode\* root) {

int diameter = 0;

height(root, diameter); // Call the helper function

return diameter;

}

int main() {

// Constructing a binary tree

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(3);

root->left->left = new TreeNode(4);

root->left->right = new TreeNode(5);

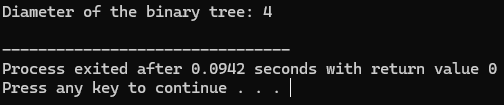
root->left->left->left = new TreeNode(6);

cout << "Diameter of the binary tree: " << diameterOfBinaryTree(root) << endl;

return 0;

}

**Output-**



**Q2-**

#include <iostream>

#include <algorithm>

using namespace std;

// Definition for a binary tree node

struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

// Function to count the total number of nodes in the tree

int countNodes(TreeNode\* root) {

if (root == nullptr) {

return 0;

}

// Recursively count the nodes in the left and right subtrees

return 1 + countNodes(root->left) + countNodes(root->right);

}

int main() {

// Constructing a binary tree

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(3);

root->left->left = new TreeNode(4);

root->left->right = new TreeNode(5);

root->right->left = new TreeNode(6);

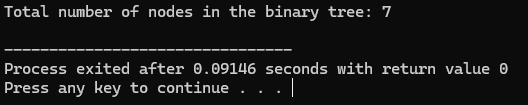
root->right->right = new TreeNode(7);

cout << "Total number of nodes in the binary tree: " << countNodes(root) << endl;

return 0;

}

**Output-**

****

**Q3-**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to heapify a subtree rooted at index i

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

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

if (left < n && heap[left] > heap[largest]) {

largest = left;

}

if (right < n && heap[right] > heap[largest]) {

largest = right;

}

if (largest != i) {

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

heapify(heap, n, largest);

}

}

// Function to delete an element from the heap

void deleteElement(vector<int>& heap, int n, int value) {

// Step 1: Find the element to delete

int index = -1;

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

if (heap[i] == value) {

index = i;

break;

}

}

// If the element is not found

if (index == -1) {

cout << "Element not found in the heap." << endl;

return;

}

// Step 2: Replace it with the last element in the heap

heap[index] = heap[n - 1];

heap.pop\_back(); // Remove the last element

// Step 3: Heapify to restore the heap property

heapify(heap, n - 1, index);

}

// Function to insert an element into the heap

void insertElement(vector<int>& heap, int value) {

heap.push\_back(value);

int i = heap.size() - 1;

// Fix the heap property if it's violated

while (i > 0 && heap[(i - 1) / 2] < heap[i]) {

swap(heap[i], heap[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

// Function to print the heap

void printHeap(const vector<int>& heap) {

for (int i : heap) {

cout << i << " ";

}

cout << endl;

}

int main() {

vector<int> heap = {100, 50, 30, 20, 15, 10, 5, 3, 2};

cout << "Original Heap: ";

printHeap(heap);

// Deleting an element (example: 20)

int valueToDelete = 20;

cout << "Deleting element: " << valueToDelete << endl;

deleteElement(heap, heap.size(), valueToDelete);

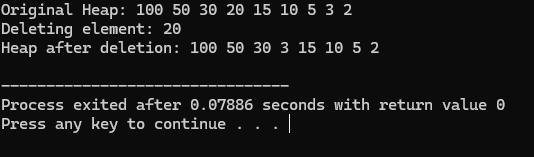
cout << "Heap after deletion: ";

printHeap(heap);

return 0;

}

**Output-**

****

**Q4-**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to heapify a subtree rooted at index i

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

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // Left child index

int right = 2 \* i + 2; // Right child index

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root, swap and continue heapifying

if (largest != i) {

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

heapify(arr, n, largest);

}

}

// Function to build the max heap (bottom-up)

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

// Start from the last non-leaf node and heapify each node

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

heapify(arr, n, i);

}

}

// Function to perform heap sort

void heapSort(vector<int>& arr) {

int n = arr.size();

// Step 1: Build the max heap

buildMaxHeap(arr, n);

// Step 2: Extract elements from the heap one by one

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

// Move the current root to the end

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

// Heapify the root element again to restore the heap property

heapify(arr, i, 0);

}

}

// Function to print the array

void printArray(const vector<int>& arr) {

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

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

}

cout << endl;

}

int main() {

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

cout << "Original array: ";

printArray(arr);

heapSort(arr);

cout << "Sorted array: ";

printArray(arr);

return 0;

}

**Output-**

