**Roll No…………….. Total No. of Pages:……**

**ST-3 (SET-IV)**

**6th SEMESTER 2023-24**

**CS192- Advanced Data Structures**

**Time allowed: 90 Minutes Max. Marks: 40**

**General Instructions:**

* **Follow the instructions given in each section.**
* **Make sure that you attempt the questions in order.**

**SECTION-A (10\*1 mark=10 marks)**

***(All questions are compulsory)***

* 1. Which of the following is a valid postfix expression for the infix expression "3 + 4 \* (2 - 1)"?
     1. **"3 4 2 1 - \* +"**
     2. "3 4 2 1 - + \*"
     3. "3 4 + 2 1 - \*"
     4. "3 4 + 2 1 \* -"
  2. The stack-based approach is commonly used to solve problems related to which type of graph traversal?
     1. Breadth-first traversal
     2. **Depth-first traversal**
     3. Pre-order traversal
     4. Post-order traversal
  3. A binary tree with all leaf nodes at the same level is known as:
     1. **Perfect Binary Tree**
     2. Complete Binary Tree
     3. Full Binary Tree
     4. Skewed Binary Tree
  4. What is the output of a pre-order traversal of the binary tree below?

5

/ \

3 8

**a) 5 3 8**

b) 3 5 8

c) 8 5 3

d) 3 8 5

* 1. When inserting an element into a binary min-heap, the element is placed in such a way that it is smaller than or equal to its \_\_\_\_\_\_.
     + 1. **Parent**
       2. Left child
       3. Right child
       4. Both Parent and Children
  2. Which hashing technique requires a secondary hash function to resolve collisions?
  3. Linear Probing
  4. Quadratic Probing
  5. Separate Chaining
  6. **Double Hashing**
  7. Two balanced binary trees are given with m and n elements respectively. They can be merged into a balanced binary search tree in \_\_\_\_ time.

**a) O(m+n)**

b) O(mn)

c) O(m)

d) O(mlog n)

* 1. When deleting a node in an AVL tree, how many recursive calls are made to reach the target node?
     1. One recursive call
     2. Two recursive calls
     3. Three recursive calls
     4. **It depends on the tree's height**
  2. In a binary tree, a node with no children is called:
     1. Root node
     2. Internal node
     3. **Leaf node**
     4. Sibling node
  3. When deleting a node in an AVL tree, in which order are the rotations performed to restore balance?
     1. **First rotate, then update height balance**
     2. First update height balance, then rotate
     3. Rotations are not performed during deletion
     4. It depends on the node's value being deleted

**SECTION-B (5\*2 mark=10 marks)**

***(All questions are compulsory)***

11) Following is an incorrect pseudocode for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

declare a character stack

while ( more input is available)

{

read a character

if ( the character is a '(' )

push it on the stack

else if ( the character is a ')' and the stack is not empty )

pop a character off the stack

else

print "unbalanced" and exit

}

print "balanced"

Which of these unbalanced sequences does the above code think is balanced?

* + 1. **((())**
    2. ())(()
    3. (()()))
    4. (()))()

12) What will be the output of the following code?

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

bool isLeaf(Node\* node) {

return node != nullptr && node->left == nullptr && node->right == nullptr;

}

int countLeafNodes(Node\* root) {

if (root == nullptr) return 0;

if (isLeaf(root)) return 1;

return countLeafNodes(root->left) + countLeafNodes(root->right);

}

int main() {

Node\* root = new Node(1);

root->left = new Node(2);

root->right = new Node(3);

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

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

root->right->right = new Node(6);

cout << countLeafNodes(root);

return 0;

}

**a) 3**

b) 4

c) 5

d) 6

13) What is the role of give snippet?

Node\* fun(Node\* root, int key) {

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

return root;

if (key < root->data)

return fun(root->left, key);

return fun(root->right, key);

}

a) Print inorder

b) print postorder

**c) search key**

d) fin maximum key

14) AVL\_delete(Node\* root, int key):

if root is NULL:

return NULL

else if key < root->data:

root->left = AVL\_delete(root->left, key)

else if key > root->data:

root->right = AVL\_delete(root->right, key)

else:

if root->left is NULL or root->right is NULL:

temp = root->left ? root->left : root->right

root = temp

else:

temp = find\_min\_node(root->right)

root->data = temp->data

root->right = AVL\_delete(root->right, temp->data)

return root

What does the function find\_min\_node() do in this pseudo-code?

a) Finds the maximum node in the AVL tree

**b) Finds the node with the minimum value in the AVL tree**

c) Finds the node with the maximum value in the AVL tree

d) Finds the parent node of the node with the minimum value

15) What the function of give code snippet?

bool func(int arr1[], int m, int arr2[], int n)

{

set<int> hashset;

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

hashset.insert(arr1[i]);

}

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

if (hashset.find(arr2[i]) == hashset.end())

return false;

}

return true;

}

a) Check if two arrays are equal

**b) Check if arr2 is subset of arr1**

c) Check if arr1 is subset of arr2

d) Merge two arrays

**SECTION-C(Coding Question) (2x5 marks=5 marks)**

Q16) Given a stack S, the task is to copy the content of the given stack S to another stack T maintaining the same order.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | Source:- 5 4 3 2 1 | Source:- 14 16 24 57 98 | Source:- 11 22 33 44 55 |
| **Output** | Destination:- 5 4 3 2 1 | Destination:- 14 16 24 57 98 | Destination:- 11 22 33 44 55 |

Solution :

**#include <bits/stdc++.h>**

**using namespace std;**

**// Auxiliary function to copy elements**

**// of the source stack to the destination stack**

**void RecursiveCloneStack(stack<int>& S, stack<int>& Des) {**

**// Base case for Recursion**

**if (S.size() == 0)**

**return;**

**// Stores the top element of the source stack**

**int val = S.top();**

**// Removes the top element of the source stack**

**S.pop();**

**// Recursive call to the function with the remaining stack**

**RecursiveCloneStack(S, Des);**

**// Push the top element of the source stack into the Destination stack**

**Des.push(val);**

**}**

**// Function to copy the elements of the source stack to the destination stack**

**void cloneStack(stack<int>& S) {**

**// Stores the destination stack**

**stack<int> Des;**

**// Recursive function call to copy the source stack to the destination stack**

**RecursiveCloneStack(S, Des);**

**cout << "Destination:- ";**

**int f = 0;**

**// Iterate until stack Des is non-empty**

**while (!Des.empty()) {**

**if (f == 0) {**

**cout << Des.top();**

**f = 1;**

**} else**

**cout << " " << Des.top();**

**Des.pop();**

**cout << '\n';**

**}**

**}**

**int main() {**

**stack<int> S;**

**S.push(1);**

**S.push(2);**

**S.push(3);**

**S.push(4);**

**S.push(5);**

**cloneStack(S);**

**return 0;**

**}**

Q17) Write a C++ program for deleting root element in MAX-Heaps

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | heap={ 10, 5, 3, 2, 4 } | heap={ 12, 6, 4, 3, 5} | heap={ 22, 16, 14 } |
| **Output** | 5 4 3 2 | 6 5 4 3 | 16 14 |

Solution :

**#include <iostream>**

**using namespace std;**

**// To heapify a subtree rooted with node i which is**

**// an index of arr[] and n is the size of heap**

**void heapify(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);**

**}**

**}**

**// Function to delete the root from Heap**

**void deleteRoot(int arr[], int& n)**

**{**

**// Get the last element**

**int lastElement = arr[n - 1];**

**// Replace root with last element**

**arr[0] = lastElement;**

**// Decrease size of heap by 1**

**n = n - 1;**

**// heapify the root node**

**heapify(arr, n, 0);**

**}**

**/\* A utility function to print array of size n \*/**

**void printArray(int arr[], int n)**

**{**

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

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

**cout << "\n";**

**}**

**int main()**

**{**

**// 10**

**// / \**

**// 5 3**

**// / \**

**// 2 4**

**int arr[] = { 10, 5, 3, 2, 4 };**

**int n = sizeof(arr) / sizeof(arr[0]);**

**deleteRoot(arr, n);**

**printArray(arr, n);**

**return 0;**

**}**

**SECTION-D (Coding Question)(1x10 mark=10 mark)**

Q18) Given a Preorder traversal of a Binary Search Tree. The task is to print leaf nodes of the Binary Search Tree from the given preorder.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Test Case 1** | **Test Case 2** | **Test Case 3** |
| **Input** | preorder[] = {890, 325, 290, 530, 965} | preorder[] = {20,40,60} | preorder[] = {60,50,90,70} |
| **Output** | 290 530 965 | 60 | 50 70 |

Solution :

**#include<bits/stdc++.h>**

**using namespace std;**

**// Binary Search**

**int binarySearch(int inorder[], int l, int r, int d)**

**{**

**int mid = (l + r)>>1;**

**if (inorder[mid] == d)**

**return mid;**

**else if (inorder[mid] > d)**

**return binarySearch(inorder, l, mid - 1, d);**

**else**

**return binarySearch(inorder, mid + 1, r, d);**

**}**

**// Function to print Leaf Nodes by doing preorder traversal of tree using preorder and inorder arrays.**

**void leafNodesRec(int preorder[], int inorder[],**

**int l, int r, int \*ind, int n)**

**{**

**// If l == r, therefore no right or left subtree.**

**// So, it must be leaf Node, print it.**

**if(l == r)**

**{**

**printf("%d ", inorder[l]);**

**\*ind = \*ind + 1;**

**return;**

**}**

**// If array is out of bound, return.**

**if (l < 0 || l > r || r >= n)**

**return;**

**// Finding the index of preorder element**

**// in inorder array using binary search.**

**int loc = binarySearch(inorder, l, r, preorder[\*ind]);**

**// Incrementing the index.**

**\*ind = \*ind + 1;**

**// Finding on the left subtree.**

**leafNodesRec(preorder, inorder, l, loc - 1, ind, n);**

**// Finding on the right subtree.**

**leafNodesRec(preorder, inorder, loc + 1, r, ind, n);**

**}**

**// Finds leaf nodes from given preorder traversal.**

**void leafNodes(int preorder[], int n)**

**{**

**int inorder[n]; // To store inorder traversal**

**// Copy the preorder into another array.**

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

**inorder[i] = preorder[i];**

**// Finding the inorder by sorting the array.**

**sort(inorder, inorder + n);**

**// Point to the index in preorder.**

**int ind = 0;**

**// Print the Leaf Nodes.**

**leafNodesRec(preorder, inorder, 0, n - 1, &ind, n);**

**}**

**int main()**

**{**

**int preorder[] = { 890, 325, 290, 530, 965 };**

**int n = sizeof(preorder)/sizeof(preorder[0]);**

**leafNodes(preorder, n);**

**return 0;**

**}**