**Group 24**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Faculty of Information and Communication Technology** | | | | | | | | | |
| **I declare that I am familiar with, and will abide by the Assignment rules of the Tshwane University of Technology**  **Signature** | **COURSE NAME: DATA STRUCTURES AND ALGORITHMS V**  **COURSE CODE: DSA417B/DTD117V** | | | | | | | | |
| **Assignment 3**  **Duration**: 7 Days  **Due Date**: 31 March 2024  **Total Marks**: 40  **Total Pages 2** | | | | **Examiner**: Dr. T Chiyangwa **Moderator**: Dr. R Mushininga | | | | |
| **Student number:** | | | | | | | | |
|  |  |  |  |  |  |  |  |  |
| **Surname**: | | | | **Initials**: | | |  | % |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **QUESTION** | **1** | **2** | **3** | **4** | **5** |  |  | **MAX** | **Signatures** |
| **TOTAL MARK** | **5** | **5** | **10** | **10** | **10** |  |  | **40** |  |
| **EXAMINER MARK** |  |  |  |  |  |  |  |  |  |
| **STUDENT(S) MARK** |  |  |  |  |  |  |  |  |  |

**GROUP MEMBERS**:

|  |  |
| --- | --- |
| **NAMES** | **STUDENT NO.** |
| Khoza NP | 215738787 |
| Khumalo NN | - |
| Koenaite SL | 215734684 |
| Mudau M | 217313619 |
| Ramalatsoa HE | 217593158 |

**Question 1.**

**Explain the fundamental concepts of tree-like data structures. Discuss the advantages of tree structures over linear data structures and provide examples of real-world applications where tree structures are preferred.**

**Tree** - data structures are continuous information structures composed of hubs associated by edges. The elemental concepts of tree structures incorporate:

**Hub**- is a centralized system for data storage, each component in a tree is called hub, it contains information references for its own children.

**Root** -is the top node in a tree data structure, is also known as the beginning point of getting the tree information.

**Edge** – is one of the two primary units used to form graphs. It speaks to the relationship between hubs.

**Parent**: is a node that is a predecessor of any node, it has branch from it to any other node

**Child**: the node descendant of another node is called its children.

**Leaf**: Hubs that don't have any children are called leaf hubs or terminal nodes.

**Subtree**: each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node

A tree structure consists of nodes connected by edges, trees give effective looking and addition operations, particularly for sorted information which is easier and quicker access to data and permit for logarithmic time complexity for these operations.

**Proficient Operations**: Different tree operations like finding the minimum/maximum component, traversals (pre-order, in-order, post-order), and erasing components can be effectively executed.

**Adjusted Trees for Optimization**: Adjusted trees, like AVL trees and red-black trees, keep up adjust and guarantee productive operations indeed with energetic information.

**Example of real-world applications of tree structures:**

* **Trees find application in various fields including**:

File systems, databases, network routing, artificial intelligence and syntax parsing.

* **Record Frameworks**: Record frameworks on computers are regularly organized in a tree structure, with catalogs (envelopes) speaking to hubs and records speaking to leaf hubs.
* **Database Ordering**: In databases, tree structures such as B-trees and B+ trees are utilized for ordering, which permits proficient looking and recovery of information.

**Question 2.**

**Discuss the characteristics and applications of binary search trees (BSTs). Explain how the properties of BSTs, such as the left subtree containing nodes with values less than the root and the right subtree containing nodes with values greater than the root, facilitate efficient searching and insertion operations.**

Binary search trees (BSTs) - They are a type of binary tree data structure where each node has two children, referred to as the left child and the right child.

key characteristic of a BST for any given node:

* All nodes in the left subtree have values less than the node's value.
* All nodes in the right subtree have values greater than the node's value.

These properties empower productive looking and addition operations by leveraging the structure of the tree:

1. Productive Looking:

* To search for a value in a BST, you begin at the root hub and compare the target esteem with the current node's esteem.
* In the event that the target esteem is break even with to the current node's esteem, the look is effective.
* In the event that the target esteem is less than the current node's esteem, you move to the cleared out subtree.
* In the event that the target esteem is greater than the current node's esteem, you move to the correct subtree.
* This prepare proceeds recursively until the target esteem is found or until a leaf hub (invalid) is come to, showing that the esteem isn't present within the tree.
* Since of the BST property, at each step of the look, you'll kill half of the remaining hubs, coming about in an normal time complexity of O(log n) for looking, where n is the number of hubs within the tree.

1. Productive Addition:

* To embed a unused esteem into a BST, you begin at the root hub and compare the esteem with the current node's esteem.
* In the event that the esteem is less than the current node's esteem and the cleared out child is invalid, you insert the modern hub as the cleared out child.
* In case the esteem is more prominent than the current node's value and the correct child is invalid, you embed the modern hub as the correct child.
* If the esteem is less than the current node's value but the cleared out child isn't invalid, you recursively embed the esteem into the cleared out subtree.
* In case the esteem is more prominent than the current node's esteem but the correct child isn't null, you recursively embed the esteem into the correct subtree.
* This handle guarantees that the BST property is kept up after inclusion.
* Addition in a BST too has an normal time complexity of O(log n) since, comparative to looking, it includes navigating the height of the tree, which is logarithmic within the number of hubs.

1. Applications of Parallel Look Trees:

* Image Tables: BSTs are broadly utilized to execute image tables in programming dialects and compilers, where proficient looking, inclusion, and erasure of key-value sets are required.
* Database Ordering: BSTs are utilized in database frameworks to make files on columns for fast recovery of information. Cases incorporate B-trees and B+ trees.
* Lexicon Executions: BSTs can be utilized to actualize word references and acquainted clusters, where keys are put away in sorted arrange, permitting for proficient lookup, inclusion, and cancellation operations.
* Auto-Complete and Spell Check: BSTs can be utilized in content preparing applications for actualizing auto-complete and spell-check functionalities by putting away a word reference of words in sorted arrange.
* Extend Inquiries: BSTs can productively bolster run inquiries, where you need to recover all values inside a given run, by performing an in-order traversal of the tree whereas checking the extend condition.

**Question 3**

**Implement a binary search tree (BST) in Java, C++ or Python and define methods to insert elements, search for a specific element, and perform in-order traversal to print the elements in sorted order.**

#include <iostream>

class TreeNode {

public:

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) {

this->val = val;

this->left = nullptr;

this->right = nullptr;

}

};

class BinarySearchTree {

private:

TreeNode\* root;

public:

BinarySearchTree() {

this->root = nullptr;

}

void insert(int val) {

this->root = insertRecursive(this->root, val);

}

TreeNode\* insertRecursive(TreeNode\* root, int val) {

if (root == nullptr) {

return new TreeNode(val);

}

if (val < root->val) {

root->left = insertRecursive(root->left, val);

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

root->right = insertRecursive(root->right, val);

}

return root;

}

bool search(int val) {

return searchRecursive(this->root, val);

}

bool searchRecursive(TreeNode\* root, int val) {

if (root == nullptr) {

return false;

}

if (root->val == val) {

return true;

} else if (val < root->val) {

return searchRecursive(root->left, val);

} else {

return searchRecursive(root->right, val);

}

}

void inOrderTraversal() {

inOrderTraversalRecursive(this->root);

}

void inOrderTraversalRecursive(TreeNode\* root) {

if (root != nullptr) {

inOrderTraversalRecursive(root->left);

std::cout << root->val << " ";

inOrderTraversalRecursive(root->right);

}

}

};

int main() {

BinarySearchTree bst;

bst.insert(5);

bst.insert(3);

bst.insert(7);

bst.insert(1);

bst.insert(4);

std::cout << "Is 4 present in the BST? " << (bst.search(4) ? "true" : "false") << std::endl;

std::cout << "Is 6 present in the BST? " << (bst.search(6) ? "true" : "false") << std::endl;

std::cout << "Elements of the BST in sorted order:" << std::endl;

bst.inOrderTraversal();

return 0;

}

**Question 4:**

**Implement a heap data structure in Java, C++ or Python and define methods to insert elements and perform heapify operations. Demonstrate the usage of these methods by creating a heap and inserting elements into it.**

#include <iostream>

using namespace std;

class MinHeap {

private:

int\* heap;

int size;

int capacity;

public:

MinHeap(int capacity) {

this->capacity = capacity;

this->size = 0;

this->heap = new int[capacity];

}

void insert(int value) {

if (size == capacity) {

cout << "Heap is full. Cannot insert more elements." << endl;

return;

}

heap[size] = value;

size++;

heapifyUp(size - 1);

}

void heapifyUp(int index) {

int parentIndex = (index - 1) / 2;

while (index > 0 && heap[index] < heap[parentIndex]) {

swap(index, parentIndex);

index = parentIndex;

parentIndex = (index - 1) / 2;

}

}

void swap(int index1, int index2) {

int temp = heap[index1];

heap[index1] = heap[index2];

heap[index2] = temp;

}

void printHeap() {

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

cout << heap[i] << " ";

}

cout << endl;

}

};

int main() {

MinHeap minHeap(10);

minHeap.insert(5);

minHeap.insert(3);

minHeap.insert(8);

minHeap.insert(1);

minHeap.insert(10);

cout << "Heap elements after insertion:" << endl;

minHeap.printHeap();

return 0;

}

**Question 5:**

**Implement a queue data structure in Java, C++, or Python using an array and demonstrate its usage by performing enqueue and dequeue operations. Provide a brief explanation of each operation.**

#include <iostream>

using namespace std;

class Queue {

private:

int capacity;

int\* queue;

int front;

int rear;

int size;

public:

Queue(int capacity) {

this->capacity = capacity;

queue = new int[capacity];

front = 0;

rear = -1;

size = 0;

}

~Queue() {

delete[] queue;

}

bool is\_empty() {

return size == 0;

}

bool is\_full() {

return size == capacity;

}

void enqueue(int item) {

if (is\_full()) {

cout << "Queue is full. Cannot enqueue item." << endl;

return;

}

rear = (rear + 1) % capacity;

queue[rear] = item;

size++;

cout << "Enqueued item: " << item << endl;

}

int dequeue() {

if (is\_empty()) {

cout << "Queue is empty. Cannot dequeue item." << endl;

return -1;

}

int item = queue[front];

front = (front + 1) % capacity;

size--;

return item;

}

void display() {

if (is\_empty()) {

cout << "Queue is empty." << endl;

return;

}

cout << "Queue contents: ";

int i = front;

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

cout << queue[i] << " ";

i = (i + 1) % capacity;

}

cout << endl;

}

};

int main() {

Queue queue(5); // Create a queue with capacity 5

// To add an item to a queue

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.enqueue(40);

// Display the contents of the queue

queue.display();

// remove an element

int dequeued\_item = queue.dequeue();

cout << "Dequeued item: " << dequeued\_item << endl;

// Display the contents of the queue after dequeue

queue.display();

return 0;

}

**Explanation of operations:**

1. **enqueue( item)** - Adds an item to the reverse of the line. It checks if the line is full before adding the item. However, it prints a communication indicating that the enqueue operation cannot be performed, If the line is full.
2. **dequeue()** - Removes and returns the frontal item from the line. It checks if the line is empty before dequeuing an item. However, it prints a communication indicating that the dequeue operation cannot be performed, If the line is empty.
3. **display()** - Displays the contents of the queue. However, it prints a communication indicating that the line is empty, If the line is empty. else, it iterates through the line and prints each element.

////////////////////////////////////////////////////END!!!!!!!!!!!!!!!!!!!!!!!!!!////////////////////////////////////////////