**CSE 222**

**HOMEWORK 5 REPORT**

**----------------------------------------------**

**ERSEL CELAL EREN**

**1901042673**

**1 - System Requirements**

In this task, we are asked to implement Binary Heap Tree, Binary Search Tree, to research quadtree and calculate some heigth, level, node calculations according to complete binary tree. Heap is implemented by using node structure which I extended from Node class in Binary Tree that given in our textbook. For Binary Search Tree, array implementation was required. There was a relation that we should see in array implementation of Binary Search Tree for left and right childs.

**3 – Problem Solution Approach**

***Q3-)*** For this question, I inherited MyHeap class from BinaryTree and implemented Comparable interface as generic. This heap class is implemented as min-heap class, so root of the tree is the smallest element. Elements are sorted according to their data by using compareTo method that should be implemented by generic class. MyHeap class has inner generic HeapNode class which is extended from Node class in BinaryTree class and it has additional fields such as parent and depth. Every node has depth value and parent reference that indicates its parent. These two member fields are used in methods like offer() and delete(). Also MyHeap class has helper fields to handle this methods. It has merge function that takes another heap instance as parameter and merge them while preserves original heaps. MyHeap class considers properties of Binary Heap. It is always complete tree and parent node is always smaller than left and right child.

Q4-) In this question, I used array to represent Binary Search Tree. It starts with 3 sized array. There is relation in array implementation of BST. If index of parent is

***i***, then index of left child is (2\****i***)+1, and index of left child is (2\****i***)+2 . By using that relation between array indexes and Binary Search Tree, I implemented necessary methods like add, contains, find, delete with the help of that arithmetic. Class has both recursive and iterative methods mixed. Printing the tree as preorder is recursive, on the contrary searching element has iterative solution with the help of array. It would be a bad way if it had node structure.

Both classes using exceptions.

**4 – TEST CASES**

*Q3 Test Cases-)*

MyHeap**<**Integer**>** heap1 **=** **new** MyHeap**<>**();

//offered elements

heap1.offer(80);

heap1.offer(120);

heap1.offer(33);

heap1.offer(51);

heap1.offer(0);

heap1.offer(49);

heap1.offer(190);

heap1.offer(10); //offered element that will be smallest

heap1.poll(); //polling root

//creating new heap and adding elements

MyHeap**<**Integer**>** heap2 **=** **new** MyHeap**<>**();

heap2.offer(235);

heap2.offer(112);

heap2.offer(319);

heap2.offer(13);

heap2.offer(77);

heap2.offer(88);

heap2.offer(139);

//merging two heap

MyHeap**<**Integer**>** merged **=** heap1.merge(heap2);

//polling until end of the tree

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

System.*out*.println("Removed Element: " **+** merged.poll());

//printing heap1 and heap2 to see that they are preserved

System.*out*.printf("\nHeap1 after merging, it is protected\n");

System.*out*.println(heap1);

System.*out*.printf("\nHeap2 after merging, it is protected\n");

System.*out*.println(heap2);

Q4 Test Cases-)

//creating BinarySearchTree and adding elements

ArrayBasedBinarySearchTree**<**Integer**>** BST1 **=** **new** ArrayBasedBinarySearchTree**<>**();

BST1.add(107);

BST1.add(53);

BST1.add(148);

BST1.add(**-**1);

BST1.add(99);

BST1.add(78);

BST1.add(65);

BST1.add(85);

BST1.add(100);

BST1.add(173);

BST1.add(107); //adding same element

BST1.find(99)); //search existing element

BST1.find(8765)); //search non-existing element

BST1.contains(99); //search existing element

BST1.contains(8765); //search non-existing element

//deleting items

BST1.delete(99);

BST1.delete(107);

**5 – RESULTS**

Q4 Test Results-)

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Q3 Test Results -)

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**2-Class Diagrams**

metin, siyah, skorbord, levha içeren bir resim

Açıklama otomatik olarak oluşturuldu

**COMPLEXITY ANALYZE**

Q4)

ArrayBasedBinarySearchTree() 🡪🡪 θ(n)

PreOrderPrint(…) 🡪🡪 θ(n) because traverse all nodes

add(E **item**) 🡪🡪 This method calls findPlace and reallocate, so complexity is O(n)

findPlace(E **item**) 🡪🡪 T(n) = theta(n) + theta(logn) , so complexity is O(n)

reallocateArray() 🡪🡪 θ(n)

findIndex(E **item**) 🡪🡪 Best case θ(1) , worst case θ(n), general case O(n)

contains(E **target**) 🡪🡪 This method call ‘findIndex’, so complexity is O(n)

find(E **target**) 🡪🡪 Calls findIndex, so complexity is O(n)

delete(E **target**) 🡪🡪 Best case θ(1), worst case θ(n), complexity is O(n)

findMostLeft(*int* **index**) 🡪🡪 Best case θ(1), worst case θ(logn), so complexity is O(logn)

preOrderTraverse(…) 🡪🡪 θ(nlogn)

toString() 🡪🡪Calls preOrderTraverse method, so complexity is θ(nlogn)

remove(E **target**) 🡪🡪 Calls delete method, so complexity is O(n)

Q3)

MyHeap(MyHeap**<**E**>** **other**) 🡪🡪 O(n^2) because it calls offer and poll methods in for loop

MyHeap(E **data**) 🡪🡪 θ(1)

MyHeap() 🡪🡪 θ(1)

merge(MyHeap**<**E**>** **other**) 🡪🡪 O(n^2) because it calls offer and poll methods in for loop

set\_max\_depth() 🡪🡪 θ(logn)

offer(…)🡪🡪Bestcase θ(1), worstcaseθ(n),so complexity isO(n) because fixtree and findavailableNode

fixTree(…)🡪🡪Because it is complete tree, best case θ(1), worst case θ(logn), general caseO(logn)

findAvailableNode(HeapNode**<**E**>** **node**) 🡪🡪 Best case θ(1), worst case θ(n), , general case O(n)

findRemovedNode(HeapNode**<**E**>** **root**) 🡪🡪 Best case θ(1), worst case θ(n), , general case O(n)

remove() 🡪🡪 Complexity is O(n),, because it calls poll() method

poll() 🡪🡪 Best case θ(1), worst case O(n), , general case O(n),

peek() 🡪🡪 θ(n)

element() 🡪🡪 θ(n)

isEmpty() 🡪🡪 θ(n)

contains(E **item**) 🡪🡪 Best case θ(1), worst case θ (n), , general case O(n),

toString() 🡪🡪 O(n^2)

preOrderPrint(…) 🡪🡪 O(n^2)





