GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework #5 Report

Hasan Mutlu
1801042673

1. SYSTEM REQUIREMENTS

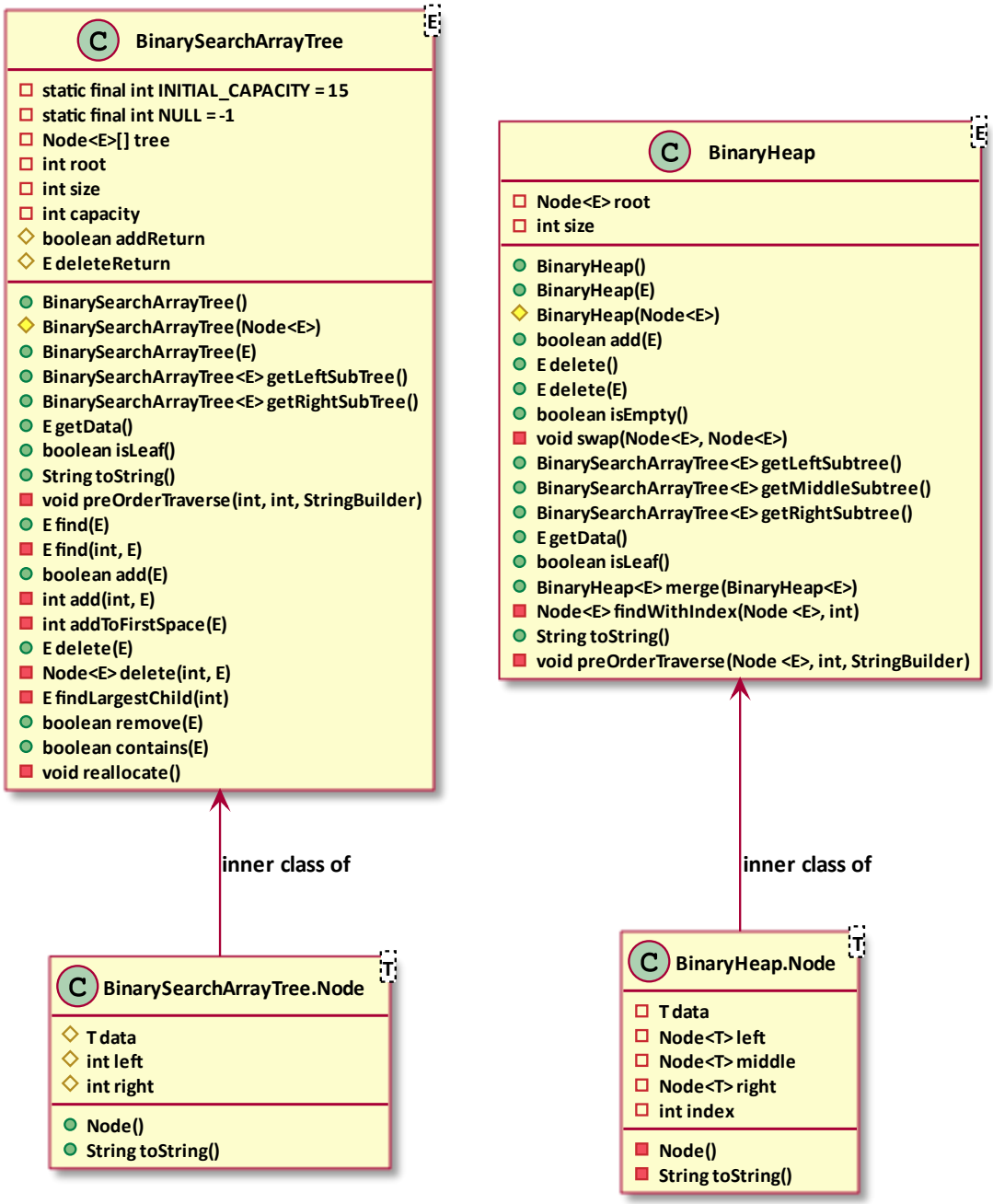
Functional Requirements:

➔ User should be able to see the result of hard-coded test cases.

Non-Functional Requirements:

➔ Implementation: The programs shall be implemented using VSCode, Ubuntu 18.04 WSL and Java 11.
➔ Compiling and Running: The programs should be compiled and run with following commands:
➔ -$ javac *.java PartX.java
➔ -$ java PartX
➔ Efficiency: The algorithms must run as efficiently as possible.
➔ Relaibility: The algorithms and the programs must run reliably, should handle every possible valid usage.

2. CLASS DIAGRAM



3. PROBLEM SOLUTION APPROACH

In part 1, I firstly researched online sources about the problems. Then I calculated them mathematically.

I couldn't do part 2 due to tight schedule.

In part 3, I firstly studied what is heap. I researched the book and online sources. Nearly all of them implemented this data structure using ArrayList. I decided to make my implementation similar to them, using index in nodes to traverse around the tree. While implementing some of the methods, I am influenced by the heap algorithms and PriorityQueue codes from the book. Heap being ternary didn't change much in the algorithms.

In part 4, I created a node class similar to linked structures. But in this one, nodes held index information of their children rather than references. I created an array of nodes and stored the data in them. While implementing the algortihms, I am influenced by the BinarySearchTree codes from the book.

4. TEST CASES

Test Case #	Test Case Description	Test Data	Expected Result	Actual Result	Pass/Fail
Part3 Test1	Create Heap1	Add: 50, 40, 60, 70, 80, 30, 20, 10, 0	Heap created correctly	As Expected	Pass
Part3 Test2	Delete Elements from Heap1	Delete: 0, 30, 60	Elements deleted correctly	As Expected	Pass
Part3 Test3	Create Heap2	Add: 55, 45, 65, 85, 75, 25, 35, 5, 15	Heap created correctly	As Expected	Pass
Part3 Test4	Delete Elements from Heap2	Delete: 45, 25, 85	Elements deleted correctly	As Expected	Pass
Part3 Test5	Merge heaps	Heap1 and Heap2	Heaps merged correctly	As Expected	Pass
Part4 Test1	Create Tree1	Add: 50, 75, 25, 87, 95, 12, 35, 45, 65, 60, 55, 90, 100	Tree created correctly	As Expected	Pass
Part4 Test2	Create Tree2 Linearly	Add: 1, 2, 3, 4, 5, 6	Tree created correctly	As Expected	Pass

Part4 Test3	Test Contains Method on Tree1	Contains: 55 and 48	True for 55, False for 48	As Expected	Pass

5. RUNNING AND RESULTS

Part3:

```
Homework 5 Part 3 Driver Program
Create Heap1 and add 50, 40, 60, 70, 80, 30, 20, 10, 0
Heap 1:
0
-10
--20
---70
---50
--80
-30
--60
--40
```

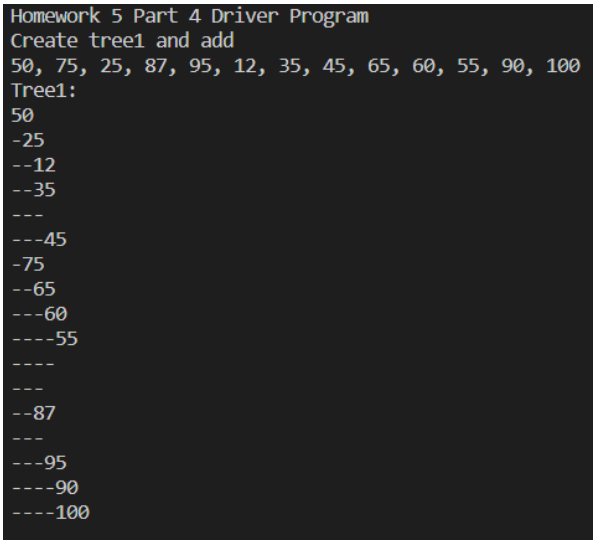
```
Delete 0, 30 and 60
Heap1 after deletion:
10
-20
--40
--70
-50
--80
```

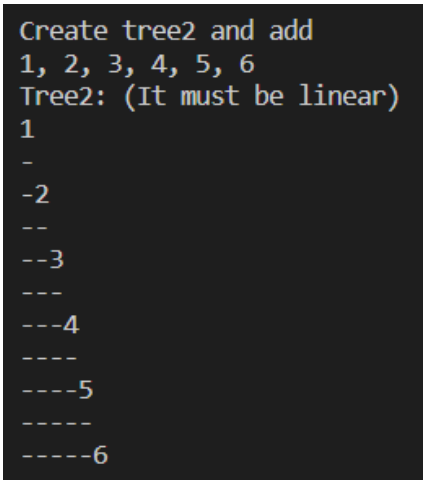
```
Create Heap2 and add 55, 45, 65, 85, 75, 25, 35, 5, 15
Heap2:
5
-15
--25
---85
---55
--75
-35
--65
--45
```

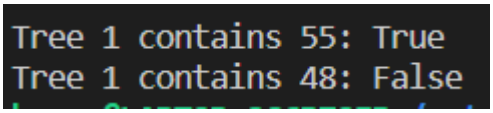
```
Delete 45, 25 and 85
Heap2 after deletion:
5
-15
--75
--65
-35
--55
```

```
Merge two heaps:
5
-15
--20
---40
---35
--65
---75
---70
-10
--55
---80
--50
```

Part4:







6. EXPLANATIONS AND ANALYZES

Part1:

a.) Let the total depth of nodes function be f(h) where h is height

-> f(1) = 1 (Root is 1)

-> f(2) = 2 + 2 + 1 = 2 + 2 + f(1) (There are 2 nodes with depth 2 and there is also previous depth's sum.)

-> f(3) = 3 + 3 + 3 + 3 + f(2) (There are 4 nodes with depth 3 and there is also previous depths' sum.)

-> f(4) = 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + f(3) (There are 8 nodes with depth 3 plus previous depths' sum.)

A slight pattern can be seen now. Take functions into height degree paranthesis'.

-> f(2) = 2*2 + f(1)

-> f(3) = 3*4 + f(2)

-> f(4) = 4*8 + f(3)

Pattern is now clear. Result is:

-> f(h) = h*2^(h-1) + f(h-1)

-> f(1) = 1

b.)

In a complete binary search tree, number of remaining elements to be searched decreases by half after each comparison. So, in worst case, number of comparisons required would be in ceiling(log2(n)). Number of nodes in a complete binary tree is 2^height – 1. Number of nodes in a heigh h is 2^(h-1). So, to get an average,

- ➔ n = 2^h -1
- ➔ so, h = log2(n+1)
- ➔ Number of searches required to find an element at a height(in other words, depth): ceiling(log2(2^h -1))
- ➔ Number of nodes at a height: 2^(h-1)

So, if we sum number of searches required to find an element at a height for all elements and divide it by number of elements, we get the average. To achive this, we need two nested summations (∑). Outer is for iterating height, inner is for iterating elements at a height. The formula would be like this:

$$\sum_{h=1}^{\log_2(n)} \left(\sum_{i=1}^{2^{h-1}} ceiling\left(\log_2\left(2^i - 1 \right) \right) \right)$$

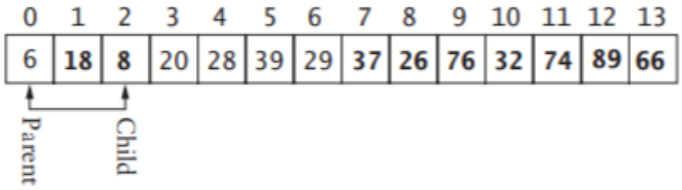
Actually, the upper part of the equation is equal to the total depth of nodes in a tree of height h, which is what we found in previous question. We can find average number of comparisons needed by using that recursive formula and dividing the result by log2(h+1) too.

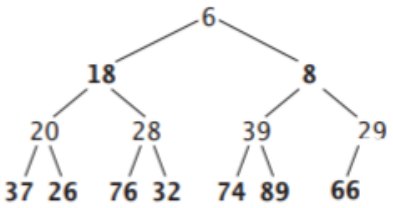
c.)

Yes, there is a restriction. In the light of what we found in previous two questions,

- ➔ Number of nodes: 2^h – 1, where h is the height of the tree.
- ➔ Number of leaves: 2^(h-1)
- ➔ Number of internal nodes: n – 2^(h-1)
- ➔ n = 2^h -1, so
- ➔ n – 2^(h-1) = 2^h - 1 – 2^(h-1)
- ➔ = 2^h(1 – 2^(-1)) – 1 = (2^h)/2 – 1

Part3:





In part 3, of course I used nodes and links. But inside nodes, I carried index information too, like the implementations in the book. When new data is inserted, it is linked to its parent by finding the parent with (size – 1) / 2 index. And when swapping operations are done, the nodes aren't swapped completely. Instead, the data in nodes are swapped and nodes' positions stayed the same.

I firstly implemented this structure as ternary but later I learned that it was a typo so I changed it to be binary.

Method Analyzes:

➔ add(E item)

In this method, firstly, a new node is created and added to the bottom. This is done in Teta(logn) time. After insertion, element is swapped with its parents, if necessary. Swapping can be at worst O(logn) time, all the way up to the root. So overall, addition is done in O(logn) time.

➔ delete(E item)

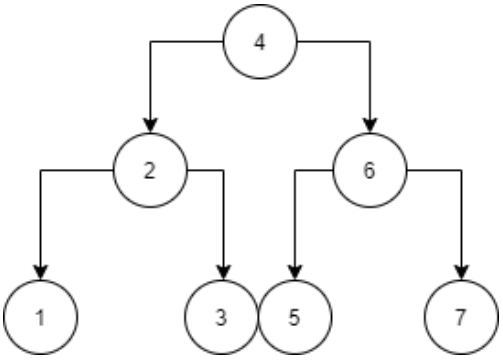
In this method, deletion is done by copying all elements except the deleted element to a new BinaryHeap and setting it to be the root of current heap. Elements are accessed and added to new heap in O(logn) time. This is done n-1 times. So overall, deletion is done in O(nlogn) time.

➔ merge(BinaryHeap<E> otherHeap)

In this method, merging is done by creating a new BinaryHeap and adding all elements of both heaps to it. Elements are accessed and added to new heap in O(logn) time. This is done n-1 times. So overall, merging is done in O(nlogn) time.

Part4:

0	1	2	3	4	5	6
Node	Node	Node	Node	Node	Node	Node
E data = 4 int left = 1 int right = 2	E data = 2 int left = 3 int right = 4	E data = 6 int left = 5 int right = 6	E data = 1 int left = null int right = null	E data = 3 int left = null int right = null	E data = 5 int left = null int right = null	E data = 7 int left = null int right = null



In part 4, an example tree like this is represented as an array of nodes as above. Nodes carry data and their children's index informations. I firstly thought of making it like array based binary heap implementations. But since the Binary Search Tree doesn't grow deterministic like heap does, I came up with this solution.

Method Analyzes:

➔ find(E item)

It works same as Binary Search Tree. In worst case, in a completely linear tree, a search takes place in Teta(n) time but it is an odd scenario. In regular basis, in a more complete tree, search is done in O(logn) time. It is not Teta because tree might not be fully complete.

➔ boolean add(E obj)

This wrapper function calls the recursive function. It may take Teta(n) time if memory reallocation is needed. In recursive function, in worst case, an insertion takes place in Teta(n) time, which is again when the tree is completely linear. But this is an odd scenario. In regular basis, in a more complete tree, insertion is done in O(logn) time at worst. It is not Teta because tree might not be fully complete. Reallocation happens really, so it has amortized O(logn) time complexity.

➔ E delete(E target)

This is wrapper cuntion for recursive call. In recursive function, deletion takes place in O(logn) time in worst case, which is the element to be deleted is a leaf. Again, it is not Teta because tree might not be fully complete. Even when element to be deleted is not a leaf, if it has two children, it is replaced by its largest child, which will definitely be a leaf, so it takes O(logn) time as well.