COMP232 – Data Structures & Problem Solving
Fall 2020

Homework #5
Binary Tree Applications

1. Describe an algorithm for using a Priority Queue to sort a list of integers. You do not have to write any code, just describe the algorithm in sufficient detail to convince me that the code could be written fairly easily.

For this algorithm you would most likely want to start by sorting the array of integers if it is not already sorted. I know there is a method that can sort an array in descending order which would make the largest integer first. This would make the building of a priority queue similar because we could just iterate over the array and build our max heap, which is a binary tree where the root node holds the highest value, and each parent node should have a higher value than the children. You could define the given list of integers pre-sorted or not and then define an empty binary tree and as we iterate over the array we create new nodes, verifying their value is less than their parent's value, until we have iterated over the entire array.

2. Give and briefly justify asymptotic bounds for the running time of the algorithm you described in #1, assuming that the backing store for the priority queue is:

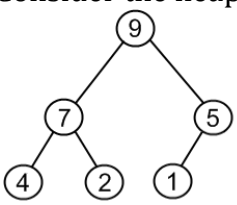
a. an unsorted array-based list.

For an unsorted array backing this would mean the add function would have an average runtime of n^2 because every time you add an element you would also have to swap n times until the heap is correctly organized

b. a heap.

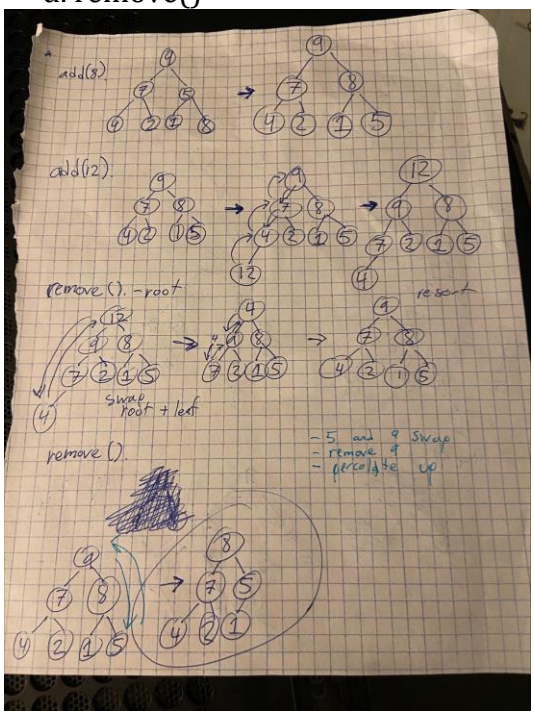
With a heap as our backing store when we add the integers from the list the best case for every iteration, or value comparison would be if the value is already in the correct spot and doesn't need to be swapped. In this case, the big-omega would be constant. The more realistic runtime is big-0 of nlgn because most likely as we add values we will need to make n swaps so that each parent is greater than both children nodes.

3. Consider the heap shown below:



Apply each of the following operations, one after the other, to the above max heap. Redraw the heap after each operation has been performed.

a. add(8)
b. add(12)
c. remove()
d. remove()



4. The `GenericBox` class in the `linear.generic` package of the COMP232-SampleCode project is a generic class with the type parameter `T`. How would you change the definition of the `GenericBox` class such that it can only store types that are numeric (i.e. of type `Integer`, `Float`, `Short`, `Byte`, `Double`, `Float`, etc.). You need only give the first line of the class definition (e.g. `public class GenericBox`…) as your solution for this question. Hint: These classes share a common super class.

You would change the T to an N signify that the type should be numeric. So the first line of the class would be public class GenericBox<T extends Number> {}.

5. The `CS232ArrayHeap` class is designed to maintain a max heap with respect to the `compareTo` method implemented by the key type. The `compareTo` method in the `String` class places items into alphabetical order, thus strings later in a dictionary would come out of the heap before words earlier in a dictionary. This may be backwards for many applications. This can be addressed by defining a new type for the key that provides a `compareTo` method that orders the keys appropriately. This is (almost) done in the `StringMinHeapKey` class in the `hw05` package.

a. Run the `main` method in the `StringMinHeapKey` class in the `hw05` package. What happens? Why? Hint: Look at the constructor being used, the keys being passed in and the `compareTo` method implementation.

If you run the main method, there is an error that says the heap is not valid. This is mainly because the ArrayHeap class is built for a max heap, where the parents are higher than the children. However, in our main function, we are attempting to create a min heap.

b. Modify the `compareTo` method so that the given keys form a valid heap with "A" having higher priority than "B", and "B" higher priority than "C", etc. Give the body of your `compareTo` method as your solution for this exercise.

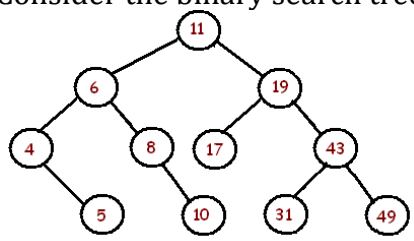
```
return -key.compareTo(o.key);
```

c. Explain why your solution in part b works.
This would work because the compare-to method either returns 0,-1, or 1. 0 if the objects are equal. 1 is returned if the first object (key) is greater than the second(o.key) and -1 if the second object is great than the first. Since we know we want to swap these -1 and 1 outputs we can put a negative in front to reverse the output.

6. Complete the `add` method in the `CS232ArrayHeap` class in your `hw05` package. The `No6Tests` class contains tests that you can use to check your implementation of this method.

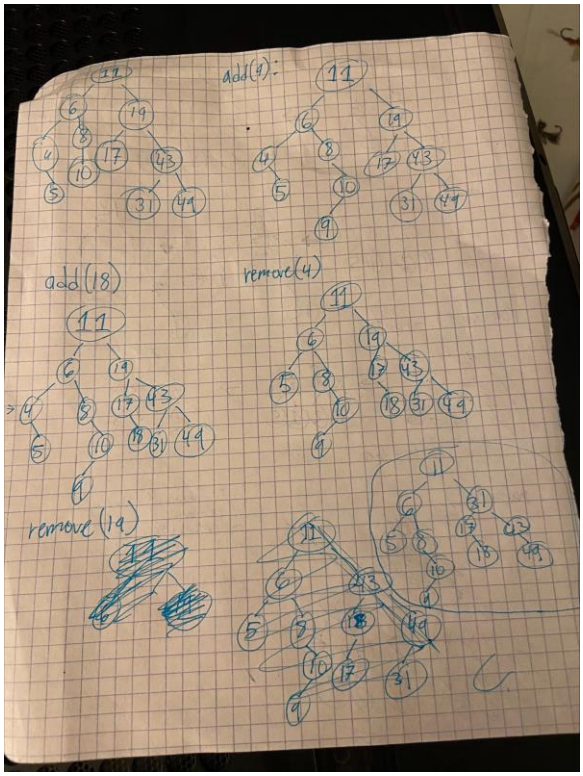
7. Complete the `adjustPriority` method in the `CS232ArrayHeap` class in your `hw05` package. The `No7Tests` class contains tests that you can use to check your implementation of this method.

8. Consider the binary search tree shown below:



Apply each of the following operations, one after another, to the above binary search tree. Redraw the binary search tree after each operation has been performed.

a. add(9)
b. add(18)
c. remove(4)
d. remove(19)



9. Complete the `get` method in the `CS232LinkedBinarySearchTree` class in your `hw05` package. The `No9Tests` class contains tests that you can use to check your implementation of this method. Suggestion: Writing a helper method here like we did with `get` in `LinkedBinaryTree` will help you with the next two questions.

10. Complete the `set` method in the `CS232LinkedBinarySearchTree` class in your `hw05` package. The `No10Tests` class contains tests that you can use to check your implementation of this method.

11. Complete the `remove` method in the `CS232LinkedBinarySearchTree` class in your `hw05` package. The `No11Tests` class contains tests that you can use to check your implementation of this method. Suggestion: Handle each of the 3 cases for remove in turn. Implement one, when that passes the appropriate tests; add the next case and so on. Suggestion 2: When you write the 3rd case, add a helper method to find the node with the smallest key in a subtree. Hint: The smallest key will always be in the leftmost node in the subtree.