1 Heaps of fun®

(a) Assume that we have a binary min-heap (smallest value on top) data structue called Heap that stores integers and has properly implemented insert and removeMin methods. Draw the heap and its corresponding array representation after each of the operations below:

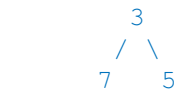
```
Heap h = new Heap(5); //Creates a min-heap with 5 as the root
[5]
```



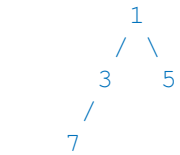
```
h.insert(7);
[5,7]
```



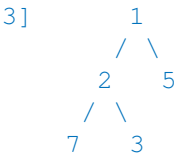
```
h.insert(3);
[3,7,5]
```



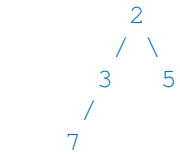

```
h.insert(1);
[1,3,5,7]
```



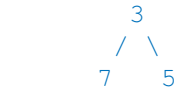

```
h.insert(2);
[1,2,5,7,3]
```




```
h.removeMin();
[2,3,5,7]
```




```
h.removeMin();
[3,7,5]
```



(b) Your friend Alyssa P. Hacker challenges you to quickly implement a max-heap data structure - "Hah! I'll just use my min-heap implementation as a template", you think to yourself. Unfortunately, your arch-nemesis Malicious Mallory deletes your min-heap.java file. You notice that you still have the min-heap.class file; could you use it to complete the challenge?
Yes. For every insert operation negate the number and add it to the min-heap. For a removeMax operation call removeMin on the min-heap and negate the number returned.

2 HashMap Modification (from 61BL SU2010 MT2)

(a) When you modify a key that has been inserted into a HashMap will you be able to retrieve that entry again? Explain?

☐ Always ☒ Sometimes ☐ Never

It is possible that the new Key will end up colliding with the old Key. Only in this rare situation will we be able to retrieve the value. It is very bad to modify the Key in a Map because we cannot guarantee that the data structure will be able to find the object for us if we change the Key.

(b) When you modify a value that has been inserted into a HashMap will you be able to retrieve that entry again? Explain?

☒ Always ☐ Sometimes ☐ Never

You can safely modify the value without any trouble. If you reference the value that you put in the tree, the changes will be reflected.

3 Sum Paths

```
void printSumPaths(Node t, int k) {
    if (t != null) {
        sumPathsHelper(t, 0, "", k);
    }
}

void sumPathsHelper(Node curNode, int curSum, String curPath, int k) {
    curSum += curNode.value;
    curPath += curNode.value + " ";

    if (curNode.left == null && curNode.right == null) {
        if (curSum == k) {
            System.out.println(curPath);
        }
        return;
    }

    if (curNode.left != null) {
        sumPathsHelper(curNode.left, curSum, curPath, k);
    }

    if (curNode.right != null) {
        sumPathsHelper(curNode.right, curSum, curPath, k);
    }
}
Bonus question solutions: In the worst case, the tree height is N. At level
   h, the code performs a concatenation of strings of length k1 * h + k2,
   e.g.

"5"
"5 " + "33"
```

```
"5 33 " + "91"
"5 33 91 " + "10"

Since String concatenation takes linear time, this results in a runtime of
```
$1+2+3+...+N = \Theta(N^2)$.

4 Bonus Question

Describe a way to implement a linked list of Strings so that removing a String from the list takes constant time. You may assume that the list will never contain duplicates.

Use a doubly linked list and a HashMap whose keys are the Strings in the list and whose values are pointers to the nodes of the list. Then when removing a String, look up the corresponding node in the HashMap and delink that node from the list.