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

(b) Consider an array based min-heap with N elements. What is the worst case running time of each of the following operations if we ignore resizing? What is the worst case running time if we take into account resizing? What are the advantages of using an array based heap vs. using a node-based heap?

```
Insert θ(log N)
Find Min θ(1)
Remove Min θ(log N)


Accounting for possible resizing:
Insert θ(N)
```

```
Find Min θ(1)
Remove Min θ(logN) (Java data structures in general do not size down.
    Suppose you did have a data structure that resized down, perhaps
    after reaching half capacity, you would have to recreate a new
    smaller array and copy the elements into that array, thus running in
    θ(N))

Using a tree/node representation is not as space-efficient. For an array
    based heap, you simply need to keep a cell for each element. For a
    tree, you need to have pointers to your children in addition to a
    field for you own value.
```

(c) 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. When you retrieve the value from the map, the changes made to the value will be reflected.

3 Hash Codes

(a) Suppose that we represent Tic-Tac-Toe boards as 3 by 3 arrays of integers (each of which is in the range 0 to 2). Describe a good hash function for Tic-Tac-Toe boards that are represented in this manner. Try to come up with one such that boards that are not equal will never have the same hash code.
We can interpret the Tic-Tac-Toe board as a nine digit base 3 number, and use this as the hash code. More concretely, if the array used to store the Tic-Tac-Toe board was called board, then we could compute the hash code as follows:

$$\text{board}[0][0] + 3 \cdot \text{board}[0][1] + 3^2 \cdot \text{board}[0][2] + 3^3 \cdot \text{board}[1][0] + \ldots + 3^8 \cdot \text{board}[2][2]$$

This hash code actually guarantees that any two distinct Tic-Tac-Toe boards will always have distinct hash codes (in most situations this property is not feasible). Another thing to note is that if we used this same idea on boards of size $N \times N$ then it would take $\Theta(N^2)$ time to compute.

(b) Is it possible to add arbitrarily many Strings to a Java HashSet with no collisions? If not, what is the minimum number of distinct Strings you need to add to a HashSet to guarantee a collision?

No, it is not possible. Ideally, we should be able to make arbitrarily large hash codes and keep resizing the HashSet's underlying array as many times as necessary (which would mean we could add arbitrarily many Strings to a HashSet without collisions). However, in Java this is not possible. There are several reasons for this:

1) In Java, the hashCode method must return an **int**, which must have a value between -2^{31} and $2^{31}-1$. This means that there are only 2^{32} possible distinct hashCodes, so if we add $2^{32}+1$ distinct Strings then we are guaranteed that two of them will have the same hashCode.

2) In Java, arrays have a maximum size of $2^{31}-1$. So we cannot resize the HashSet's underlying array past this point. So if we add 2^{31} Strings then we are guaranteed that two of them will be put in the same bucket (though they might not have the same hashCode).

3) In Java's implementation of HashSet, the size of the underlying array is always a power of two. Thus the maximum size of the underlying array is 2^{30}, so if we add $2^{30}+1$ Strings then we are guaranteed that two of them will be put in the same bucket.

So the final answer is that $2^{30}+1$ is the minimum number of Strings required to guarantee a collision. You aren't expected to be able to come up with this exact number yourself, since it depends on the specific implementation details of Java's HashSet. Understanding the basic reasoning is enough (for instance, (1) is a good answer, though not technically correct).

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.

See Java's LinkedHashMap data structure to see how this might be implemented.