1   Heaps of fun®

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

```
Heap h = new Heap(5); //Creates a min-heap with 5 as the root
[0, 5]                  5
h.insert(7);
[0, 5,7]                  5
                         /
                        7
h.insert(3);
[0, 3,7,5]                3
                         / \
                        7   5
h.insert(1);
[0, 1,3,5,7]              1
                        / \
                       3   5
                      /
                     7
h.insert(2);
[0, 1,2,5,7,3]            1
                        / \
                       2   5
                      / \
                     7   3
h.removeMin();
[0, 2,3,5,7]             2
                       / \
                      3   5
                     /
                    7
h.removeMin();
[0, 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) You are tasked to implement a max-heap data structure of integers using only a min-heap of integers. Could you complete the task? If so, describe your approach. If not, explain why it's impossible.

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. Otherwise, the new key will hash to a different hash code, causing us to look in the wrong bucket inside our `HashMap` for our entry. 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. We use the key to determine where to look for our value inside our `HashMap`, and because the key hasn't been changed, we are still able to find the entry we are looking for.

3 Hash Codes

(a) Here are four potential implementations of the `Integer`'s `hashCode()` function. Categorize each as either a valid or an invalid hash function. If it is invalid, explain why. If it is valid, point out a flaw or disadvantage.

A few notes: A "valid" `hashCode()` means that any two `Integers` that are `.equals()` to each other should also return the same hash code value. In additon, the `Integer` class

extends the `Number` class, a direct subclass of `Object`. The `Number` class' `hashCode()` method directly calls the `Object` class' `hashCode()` method.

```
(1) public int hashCode() {
        return -1;
    }
```

Valid. As required, this hash function returns the same hash code for `Integers` that are `.equals()` to each other. However, this is a terrible hash code because collisions are extremely frequent (collisions occur 100% of the time).

```
(2) public int hashCode() {
        return intValue() * intValue();
    }
```

Valid. Similar to (a), this hash function returns the same hash code for `Integers` that are `.equals()`. However, `Integers` that share the same absolute values will collide (for example, `x = 5` and `x = -5` will have the same hash code). A better hash function would be to just return the `intValue()` itself.

```
(3) public int hashCode() {
        Random rand = new Random();
        return rand.nextInt();
    }
```

Invalid. This is not a valid hash function because the `hashCode` method will return different integers whenever it's called. If we have an integer that we call `hashCode` on multiple times, an integer is `.equals()` with itself, but different hash codes will be returned.

```
(4) public int hashCode() {
        return super.hashCode();
    }
```

Invalid. This is not a valid hash function because `Integers` that are `.equals()` to each other will not have the same hash code. Instead, this hash function returns some integer corresponding to the `Integer` object's location in memory.

(b) Suppose that we represent Tic-Tac-Toe boards as 3 by 3 arrays of integers (with each integer in the range 0 to 2 to represent blank, 'X', and 'O' respectively). Describe a hash function for Tic-Tac-Toe boards that are represented in this way 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.

(c) 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?

(A few useful hints: Java arrays have a maximum size of $2^{31} - 1$, and Java HashSet's underlying array's size is always a power of 2.)

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 hash codes, so if we add $2^{32} + 1$ distinct Strings then we are guaranteed that two of them will have the same hash code.

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 hash code).

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 without the hints, 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).