1. Insert the numbers 44, 32, 59, 38, 12, and 25 into a skiplist. For the coin flips, use this sequence of coin flips: HTHHTHHHTTHTHHHTTHTHTHTHTHTHTHTH.

2. If your random number generator is broken and only returns tails over and over again, what will happen to a skip list when items are inserted? What if it only returns heads over and over again?

3. In class we saw that a heap of $n$ items can be built by inserting each item one by one. Because insertion is $O(\log n)$ time this process takes $O(n \log n)$ time. In this problem we will develop an $O(n)$ time method to build a heap from a list of numbers.

```python
def make_heap(A):
    for (i=A.size(); i >= 0; i--)
        bubble_down(A[i])
    return A
```

(a) Run `make_heap` on $A = [11, 10, 4, 8, 13, 3, 2, 9, 1, 5, 7, 14, 6, 15]$. Verify the result is a heap.

(b) Given a complete binary tree, argue that if the left and right subtrees of the root have the heap property, then after running `bubble_down` on the root, the entire tree will have the heap property.

(c) Use part (b) to argue `make_heap` will turn the array into a binary heap. Hint: all of the “leaves” of the input array can be thought of as heaps of size 1.

(d) We now need to figure out how long `make_heap` takes to run. Number the levels of the tree from top to bottom e.g. the root is at level 0, the root’s children are at level 1 etc. Show that at most $2^i$ `bubble_down` calls are made at level $i$ and that each call at level $i$ performs at most $(\log n) - i$ swaps.

(e) Extra credit: Show that $\sum_{i=0}^{\log n} (\log n - i)2^i = O(n)$.

(f) Using parts (d) and (e), argue that `make_heap` takes linear time.

4. Suppose you are given a comparison based priority queue. If that priority queue’s `insert` operation is $O(1)$ time, how fast could `extract_minimum` run? Similarly if `extract_minimum` is $O(1)$ time, how fast could `insert` run?

5. Give pseudocode or describe an algorithm to decide if a binary rooted tree has the heap property. The heap property requires that the key in every node is smaller than the keys in its children AND that the tree is complete.
6. Run the following code when $x$ is a queue. Then run it when $x$ is a stack.

```cpp
x.push(4);
x.push(6);
cout<<x.pop();
x.push(3);
x.push(1);
cout<<x.pop();
cout<<x.pop();
x.push(5);
x.push(5);
cout<<x.pop();
cout<<x.pop();
cout<<x.pop();
cout<<x.pop();
```