This homework is out of 10 points and graded for completion. Working together on a homework is okay, copying answers is not. Some homeworks will include challenge extra credit problems that will be worth 2 points each.

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

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?

If only tails are returned, then the skip list will behave just like a linked list because no elements will be propagated into higher levels. If only heads are returned the algorithm, will endlessly propagate the first element into higher and higher levels causing an infinite loop.

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.
After running `make_heap` A becomes [1, 5, 2, 8, 7, 3, 4, 9, 10, 13, 11, 14, 6, 5]. By scanning through we can compare each index \( i \) to indexes \( 2i + 1 \) and \( 2i + 2 \) to make sure the heap property holds.

(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.

The smallest element in the complete binary tree is either the root element or one of the roots of the left and right subtrees, because they obey the heap property no other element can be smaller. If the root of the tree is the smallest element, the tree is already a heap and `bubble_down` will perform no swaps and immediately terminate. When one of the subtree roots is smaller, `bubble_down` will swap with the smaller of the two. After that swap the only location in the tree where the heap property is being violated is wherever the root was swapped to. The new location of the root of the entire tree is at the root of some new subtree where the two subtrees below obey the heap property. So we are back in the situation we started, but one level lower in the tree. Either the `bubble_down` process will eventually choose not to swap or hit a leaf of the tree. In both cases it terminates with the tree now following the heap property.

Another way to see this is to observe that this is exactly the case we are in when `remove_min` is called in a binary heap.

(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.

A single element is a one element heap that is it is a complete binary tree where each parent element is smaller than its children. The calls of `bubble_down` start at the bottom of the tree and work towards the top. Because every element at the bottom level forms its own heap, the `bubble_down` calls at the second level of the tree will form heaps of height 2. After the `bubble_down` calls at each level the elements at that level are the roots of newly formed heaps and the final `bubble_down` call at the root of the entire tree will make the entire tree into a heap.

(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.

There are at most \( 2^i \) elements in level \( i \) of the tree. Therefore there are at most `bubble_down` calls made by `make_heap` at level \( i \) of the tree. The `bubble_down` calls can at worst swap in element all the way down to a leaf at the bottom level. Because the are \( \log n \) levels in the tree, a `bubble_down` call at level \( i \) can perform at most \( \log n - i \) swaps.

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

\[
= n \sum_{i=0}^{\log n} (\log n - i)^2 - \log n
\]

At this point if you know from calculus that \(\sum_{i=0}^{\infty} \frac{1}{2^i}\) is a convergent series, then you can conclude that the summation is upper bounded by \(n\) times a constant. We can be more exact by differentiating a geometric series:

\[
f(x) = \sum_{i=0}^{\infty} x^i = \frac{1}{1-x}
\]

\[
f'(x) = \sum_{i=1}^{\infty} ix^{i-1} = \frac{1}{(1-x)^2}
\]

\[
x f'(x) = \sum_{i=1}^{\infty} ix^i = \frac{x}{(1-x)^2}
\]

\[
\frac{1}{2} f'(\frac{1}{2}) = \sum_{i=1}^{\infty} \frac{1}{2} i^2 = \frac{1}{(1-\frac{1}{2})^2} = 2
\]

Therefore:

\[
\sum_{i=0}^{\log n} (\log n - i)^2 = n \sum_{i=0}^{\log n} i2^{-i}
\]

\[
\leq 2n = O(n)
\]

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

Because the run time for the bubble_down calls is at most a constant times the number of swaps they perform and they perform a linear number of swaps, the runtime for make_heap is \(O(n)\).

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?
Any priority queue can be used to sort by first inserting the \( n \) values to be sorted and then repeatedly extracting the minimum element to get the values out in sorted order. If the priority queue is comparison based, then this process must take \( \Omega(n \log n) \) time by our comparison sorting lower bound. So if \( \text{insert} \) is \( O(1) \) time, then \( \text{extract\_minimum} \) must be \( \Omega(\log n) \) time and vice versa if \( \text{extract\_minimum} \) is \( O(1) \) time, then \( \text{insert} \) must be \( \Omega(\log n) \).

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.

First we will verify whether or not each element is smaller than its children. This can be done with a recursive algorithm:

```python
def parent_child_verifier(node):
    if node == nullptr:
        return true;
    if node->left != nullptr and node->left->key < node->key:
        return false;
    if node->right != nullptr and node->right->key < node->key:
        return false;
    return parent_child_verifier(node->left) and parent_child_verifier(node->right)
```

If \( \text{parent\_child\_verifier(root)} \) is true, then we only need to make sure the tree is a complete tree. This can be attacked a couple of different ways. One way is to utilize our knowledge for how we can store a complete binary tree into an array. If there are \( n \) nodes in a tree, then we can attempt to copy the elements into an array of size \( n \) such that the left and right children for a node at index \( i \) are at \( 2i + 1 \) and \( 2i + 2 \). If a node would need to be placed outside the range of the array, then the tree is not complete.

If both of these checks pass, then the rooted binary tree is a heap.

6. Run the following code when \( x \) is a queue. Then run it when \( x \) is a stack.

```python
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(2);
cout<<x.pop();
cout<<x.pop();
cout<<x.pop();
```

When \( x \) is a queue:
463152

When $x$ is a stack:

613254