CSCI 151
Exam 1
March 5, 2021

The exam has 5 numbered questions that are worth 20 points each.

This is a closed-book, closed-notes, closed-Internet exam. You can type your answers into Microsoft Word or your favorite text editor, or if you prefer you can take with pen and paper, and scan your answers into a PDF file. Either way, email your answers back to me by noon on Saturday, March 6 (Oberlin time). Time yourself. If you do not have extended time on exams you have one hour for this exam.

After you have finished the exam please indicate whether you followed the Honor Code on the exam. Also give the time you started working on this exam, and the time when you finished it.

I □ did □ did not follow the Honor Code while taking this exam.

Start time: _____________________

Finish time: _____________________

__________________________________
Signature
1. Suppose S is a stack that starts off empty and we do the following sequence of operations:

```
S.push(1);   S: 1
S.push(2);   S: 1 2
S.push(3);   S: 1 2 3
S.pop();     S: 1 2
S.push(4);   S: 1 2 4
S.pop();     S: 1 2
```

What will S.top() return?  

Now we continue with this same stack and do

```
S.pop();     S: 1
S.push(5);   S: 1 5
S.push(6);   S: 1 5 6
S.pop();     S: 1 5
```

What will S.pop() return now?  

What will another S.pop() return?  

What will another S.pop() return? It throws an exception

Next, we do this with a queue. Suppose Q is an empty queue and we do

```
Q.enqueue(1);  Q: 1
Q.enqueue(2);  Q: 1 2
Q.dequeue();   Q: 2
Q.enqueue(3);  Q: 2 3
Q.dequeue();   Q: 3
```

What will Q.front() return?  

Now we do two enqueues:

```
Q.enqueue(4);  Q: 3 4
Q.enqueue(5);  Q: 3 4 5
```

What will Q.dequeue() return?  

What will another Q.dequeue() return?  

What will another Q.dequeue() return?
2. Consider the following abstract class that holds two objects of class E:

```java
public abstract class Pair<E> {
    abstract E getFirst(); // return the first element of pair
    abstract E getSecond(); // returns the second element
    abstract E setFirst(E item);
    abstract E setSecond(E item);
    void isTwin() {
        return getFirst() == getSecond();
    }
    void switcheroo() {
        E temp = getFirst();
        setFirst(setSecond());
        setSecond(temp);
    }
}
```

Explain what is involved in making a non-abstract version of Pair<E>. You don’t need to write any code; just say what needs to be done. Alternatively, if you prefer coding write the following class:

```java
public class MyPair<E> extends Pair<E>{
    ...
}
```

You need to make all of those abstract methods concrete. To do that, you need some way to represent the data from a pair. Here’s an easy way to start:

```java
public class MyPair<E> extends Pair<E>{
    E first;
    E second;
    public MyPair( E first, E second) {
        this.first  = first;
        this.second = second;
    }
    ...
}
```

What I have done here is to add data fields `first` and `second` and a constructor to give them initial values. It is then an easy matter to write concrete getter and setter methods.
3. Remember your code for MyArrayList from Llab 2. It probably started

```java
public class MyArrayList<E> extends AbstractList<E> {
    private int size;
    private E[] data;
    ...
}
```

Give code for a new MyArrayList<E> method

```java
public void reverse()
```

that reverses the order of the data in a list. This modifies the list; it does NOT make a new list. So if L starts as the MyArrayList<Integer> with data {1, 2, 3} then after we do L.reverse() L will have data {3, 2, 1}

There are lots of ways to reverse a list. I was a bit surprised that no one pushed all of the element onto a stack and then popped them back into the list. That’s not a very good solution, but it we used it in Lab 3. Here is a straightforward solution:

```java
public void reverse() {
    int leftIndex = 0;
    int rightIndex = size-1;
    while (leftIndex < rightIndex) {
        E left = data[leftIndex];
        E right = data[rightIndex];
        data[leftIndex] = right;
        data[rightIndex] = left;
        leftIndex += 1;
        rightIndex -= 1;
    }
}
```

This walks through half of the list, swapping opposing values (first and last, second and next-to-last, etc). By using the while loop it avoids computing where the first half of the list ends.

But if you really like for-loops, you could call the leftIndex i, the rightIndex size-1-i, and have the loop bounds be for(int i=0; i<size/2; i++) ....
4. Here is a new operation with lists. Method `increment(L)` works with lists of integers by adding 1 to the value of each element. Here is the way it is implemented:

```java
public void increment( LIST L) {
    for (int i = 0; i < L.size(); i++) {
        int oldValue = L.get(i);
        L.set(i, oldValue+1);
    }
}
```

In this code the “type” `LIST` is a placeholder for either `ArrayList<Integer>` or `LinkedList<Integer>`. If `L` is a list with values such as `{4, 8, 10}`, `increment(L)` adds 1 to each of those values, producing the list `{5, 9, 11}`.

a) **Give a Big-Oh estimate for the number of steps it takes to run `increment(L)` on an ArrayList of size `n`**. Also give a 1-sentence explanation of your estimate.

   For an ArrayList the `get` and `set` methods are both constant-time. We do each of them `size` times. For a list of size `n` this is $O(2*n) = O( n )$.

b) **What is your estimate if `L` is a LinkedList?** If your estimate and explanation are the same as in part (a) just write “Same”. If something is different for LinkedLists please explain.

   It is quite different for a LinkedList. Getting or setting the `ith` element requires `i + 1` steps, because we have to walk from the start to the `ith` element. We have values of `i` running from 0 to `n-1`, where `n` is the size of the list. So just for calls to `get` we do $1+2+3+\ldots+n = \frac{n(n+1)}{2}$ steps.

   This is $O(n^2)$. We do the same number of steps for calls to `set`. The whole algorithm is $O(2*n^2) = O( n^2 )$
5. In this question we will make a new data structure that I call BobStruct. I have a
copyright on that name, so don’t try to steal it. BobStructs hold int values and only have
two methods:

```java
public void add( int x ) // this adds x to the structure
public int removeBiggest() // removes the largest value from the
   // structure and returns it
```

Here is the start of an implementation based on ArrayLists:

```java
public class BobStruct {
    ArrayList<Integer> data;
    public BobStruct () {
        data = new ArrayList<Integer>();
    }
    ...
}
```

Give code for the `add(int x)` and `removeBiggest()` methods. Also give a Big-Oh
estimate of the running time for each method on a BobStruct that contains `n` ints. You
do not need to explain your Big Oh estimates.

In some situations it might pay to make add more complicated to make removeBiggest()
faster, but for an exam I would go with what is easy. Just have add append onto the end
of the ArrayList:

```java
public void add( int x) {
    data.add(x);                    This is O(1) unless the array needs to be expanded
}
```

`removeBiggest()` does a search. It should throw an exception if the list is empty, but I’m
going to ignore that to keep it simple. The code is on the next page:
public Integer removeBiggest() {
    int big = data.get(0);
    int bigIndex = 0;
    for (int i=1; i < data.size(); i++) {
        int temp = data.get(i);
        if (temp > big) {
            big = temp;
            bigIndex = i;
        }
    }
    data.remove(bigIndex);
    return big;
}

Since the remove method returns the value it removes, we could have combined the last two lines into return data.remove(bigIndex);

The for-loop is O(n) and remove() (which is not inside the loop) is O(n), so this is all O(n).

Note that in Lab 7 we will see a much more efficient way to implement a “BobStruct”.