Name: _____

USC loginid (e.g., `ttrojan`): _____

Midterm Exam 2
CS 455, Spring 2014

Monday, April 7, 2014

There are 6 problems on the exam, with 58 points total available. There are 7 pages to the exam, including this one; make sure you have all of them. There is also a separate double-sided one-page code handout. If you need additional space to write any answers, you may use the backs of exam pages (just direct us to look there).

Note: if you give multiple answers for a problem, we will only grade the first one. Avoid this issue by labeling and **circling** your final answers and crossing out any other answers you changed your mind about (though it's fine if you show your work).

Put your name and loginid at the top of the exam. Please read over the whole test before beginning. Good luck!

	value	score
Problem 1	6 pts.	
Problem 2	6 pts.	
Problem 3	8 pts.	
Problem 4	10 pts.	
Problem 5	8 pts.	
Problem 6	20 pts.	
TOTAL	58 pts.	

Problem 1 [6 pts. total]

Assume you have an array of names with the following contents (numbers above the line are indices):

0	1	2	3	4	5	6	7	8
Flynn	Gus	Hank	Jesse	Marie	Mike	Saul	Skyler	Walter

For each of the targets given below, list the **names** that the target would be compared with in a **binary search** on the array. List them in the order that the comparisons would be done.

1. target = **Mike**

2. target = **Lydia**

3. target = **Walter**

Problem 2 [6 pts]

Here's the a slightly modified version of the `Names` class we discussed in lecture earlier in the semester, with some of its implementation shown (this version uses an ArrayList representation):

```
//   Stores a list of unique names in alphabetical order.  Allows
//   look-up, insert, and removal of elements in the list.
public class Names {

    //  Creates an empty names object
    public Names() { namesArr = new ArrayList<String>(); }

    //  Returns the number of names in the list
    public int numNames() { return namesArr.size(); }

    //  Returns true iff target is present in names
    public boolean lookup(String target) { . . . }

    //  Removes target from names, and returns true.  If target wasn't present
    //         in names, returns false and no change made to names.
    public boolean remove(String target) { . . . }

    //  Inserts newName into alphabetical names list, and returns true.
    //   Returns false and no change is made to names if newName is already
    //   present in names.
    public boolean insert(String newName) { . . . }

    //  Returns arraylist of all the names in alphabetical order
    public ArrayList<String> getNames() { return namesArr; }

    private ArrayList<String> namesArr;
     // representation invariant:
     //     a. namesArr stores numNames() names
     //     b. names are in alphabetical order, with smallest in namesArr.get(0),
     //           and largest in namesArr.get(names.Arr.size()-1).
     //     c. names are unique
}
```

All the methods above work correctly (i.e., as described in the comments) on an object that satisfies the representation invariant, but there is a problem with the code because a *client* can invalidate an object.

Part A [4]. Give concrete client code that invalidates the object `theNames`; that is, after your code runs, the representation invariant will no longer be true. We've created a `Names` object for you below to work with:

```
Names theNames = new Names();
theNames.insert("Joe");
theNames.insert("Bob");
theNames.insert("Sally");
```

Part B [2]. For the code you wrote in part A say what part(s) of the invariant was violated (you can use the labels a, b, c used in the invariant).

Problem 3 [8 pts. total]

The `ArrayList` and `LinkedList` classes are both implementations of a Java interface called `List`. Some of the methods that we have seen that come from this interface are the `add`, `get`, and `size` methods, among others. Below are two static methods written in terms of this interface, and a main method with two calls to each of the methods, one time using an array list, another time using a linked list.

A reminder about how interfaces and implementations work: for example, when we call `addElmts`, below, with an `ArrayList`, `ArrayList`'s version of `add` gets called, and when we call it with a `LinkedList`, `LinkedList`'s version of `add` gets called.

Next to each of the method calls below labeled `A`, `B`, `C`, and `D`, give the worst case big-O time to execute that call:

```
// adds the sequence of numbers 1 through n to the end of a list
static void addElmts(List<Integer> list, int n) {
   for (int num = 1; num <= n; num++) {
      list.add(num);
   }
}


// prints out all the elements in a list
static void printList(List<Integer> list) {
   for (int i = 0; i < list.size(); i++) {
      System.out.println(list.get(i));
   }
}


static void main(String[] args) {
   Scanner in = new Scanner(System.in);
   int numAL = in.nextInt();  // assume user enters two positive integers
   int numLL = in.nextInt();
   ArrayList<Integer> al = new ArrayList<Integer>();
   LinkedList<Integer> ll = new LinkedList<Integer>();

   addElmts(al, numAL);      // A. big-O as a function of numAL:


   printList(al);            // B. big-O as a function of numAL:


   addElmts(ll, numLL);      // C. big-O as a function of numLL:


   printList(ll);            // D. big-O as a function of numLL:
}
```

Problem 4 [10 points]

Assume we make a call to the following method, `prob4Func`, with a queue with the following value (in this diagram the front of the queue is at the ***left*** side, and is labeled as such):

value of `q` before call: front→ `5 8 2 0 1 7 4`

Show the contents of each of the stacks and queues listed at each of the points in the code, labeled A, B, and C. ***For each one clearly label where the top of your stack or the front of the queue is in your diagram.***

```
public static Stack<Integer> prob4Func(Queue<Integer> q) {
   Stack<Integer> first = new Stack<Integer>();
   Stack<Integer> second = new Stack<Integer>();

   boolean found = false;
   while (!q.isEmpty() && !found) {
         int curr = q.peek();
         first.push(curr);
         q.remove();
         if (curr == 0) {
            found = true;
         }
   }

   // A

   while (!q.isEmpty()) {
      second.push(q.remove());
   }

   // B

   while (!first.empty()) {
      second.push(first.pop());
   }

   // C

   return second;
}
```

at point **A**:

q:

first:

at point **B**:

q:

second:

at point **C**:

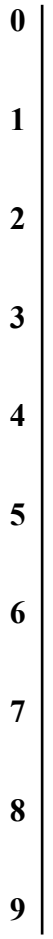
first:

second:

Problem 5 [8 pts. total]

Part A [4]. Show the results of inserting the following six keys that have the following hash values into the hash table below that starts out empty, has 10 buckets (indices 0 through 9), and uses chaining. Do the insertions in the order the keys are shown (a-f).

Key	Hash value	Key	Hash value
a. Tony	3	**d.** Silvio	3
b. Christopher	7	**e.** Paul	7
c. Carmela	3	**f.** Janice	4



For each of the following lookups from the table above, after the insertions are done, give the sequence of keys that the target key would have to be compared with to do the lookup

Part B [2]. lookup *Adriana*; Adriana's hash value is 7

Part C [2]. lookup *Sal*; Sal's hash value is 8

Problem 6 [20 pts]

Implement the static `boolean` method `isSubset` which returns true if its second argument, `sub`, is a subset of its first argument, `list`. Both arguments are Java linked list objects whose elements are in increasing order with no duplicates. ***For full credit your code must work in O(m+n) time, where m and n are the lengths of the two given lists.*** Hint: you can do this by taking advantage of the fact that the lists are ordered. Here are some examples:

`list`	**`sub`**	return value of **`isSubset(list, sub)`**	
(2 3 5 10)	*(3 6)*	*false*	
(2 3 5 10 12)	*(3 10)*	*true*	
(3 5 10)	*(3 10 17)*	*false*	
(1 7)	*()*	*true*	*[empty set is subset of every set]*
()	*(3)*	*false*	
()	*()*	*true*	*[empty set is subset of itself]*
(2 3 5 10)	*(2 3 5 10)*	*true*	*[any set is a subset of itself]*

```
public static boolean isSubset(LinkedList<Integer> list,
                               LinkedList<Integer> sub) {
```