Exam 2 Practice
(October 10, 2017)

This exam is closed-book, notes, and technology.

Please do not open the test until the instructor says time has begun.
Please stop writing once the instructor has called time.
Failure to stop writing will result in a zero on the exam.

Remember you are here to learn.
Relax and think of this as yet another learning experience.

Good luck, have fun!

Your Name: Solutions

<table>
<thead>
<tr>
<th></th>
<th>/ 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Problem 1: Stupid Blues (20 points)

Consider each of the following claims about sorting algorithms. Indicate if the claim is true (T) or false (F).

(1) The worst case runtime of merge sort is \( O(n^2) \). \( \text{F} \)

(2) The best case runtime of quicksort is \( O(n \log n) \). \( \text{T} \)

(3) We may prefer insertion sort over quicksort if we care about preserving the relative order of equal elements. \( \text{T} \)

(4) We may prefer selection sort over insertion sort when we must sort an input sequence as it comes over the wire, e.g., over a network connection. \( \text{F} \)

(5) If I choose the pivot of quicksort to be the element in the middle index of my sequence, I am guaranteed \( O(n \log n) \) runtime. \( \text{F} \)

The Merge Operation. Consider the following array:

\[ [3, 9, 10, 12, 4, 7, 11, 15] \]

Give the step-by-step execution of the merge operation from merge sort, merging the two halves of the given array together into a sorted whole. Your diagram should also include the scratch array used to hold merged elements. At each step you should:

- Indicate the state of the algorithm after executing the step. The state of the algorithm includes the indicies of the two “fingers” into the input array and the current contents of the scratch array, e.g., \( i = 1, j = 4, [3, \ldots] \).

Please number your steps so that their order is clear.

Step 0: \( i = 0, j = 4, [(\text{empty})] \)

1) \( i = 1, j = 4, [3] \)

2) \( i = 1, j = 5, [3, 4] \)

3) \( i = 1, j = 6, [3, 4, 7] \)

4) \( i = 2, j = 6, [3, 4, 7, 9] \)

5) \( i = 3, j = 6, [3, 4, 7, 9, 10] \)

6) \( i = 3, j = 7, [3, 4, 7, 9, 10, 11] \)

7) \( i = 4, j = 7, [3, 4, 7, 9, 10, 11, 12] \)

8) \( i = 4, j = 8, [3, 4, 7, 9, 10, 11, 12, 15] \)
Problem 2: What’s Left for You? (20 points)

Draw the step-by-step evolution of a binary search tree after each of the given operations. Assume that removal chooses the next largest element in the in-order traversal of the tree as the value to rotate upwards.

(a) `Tree<Integer> t = new Tree<>();`

(b) `t.insert(5);`

(c) `t.insert(1); t.insert(2);`

(d) `t.insert(3); t.insert(6); t.insert(4);`

(e) `t.remove(1);`  

*Typo: 4 should be to the right of 3 rather than to the left.*
Given the following binary search tree, write the resulting sequences obtained by traversing the tree using each of the given strategies (a blank space denotes an empty leaf):

(f) Pre-order traversal:
\[ \otimes, \Pi, \equiv, \equiv, +, \emptyset, \oplus, \leftrightarrow, \top, \prec, \checkmark \]

(g) In-order traversal:
\[ \equiv, \equiv, \Pi, +, \emptyset, \leftrightarrow, \oplus, \otimes, \prec, \checkmark, \top \]

(h) Post-order traversal:
\[ \equiv, \equiv, \leftrightarrow, \oplus, \emptyset, +, \Pi, \checkmark, \prec, \top, \otimes \]
Problem 3: Afterglow (20 points)

Draw the step-by-step evolution of two hash tables after each of the given put operations. The first hash table is implemented with a linear probing strategy. When this table is full, the table proceeds by first (1) doubling the backing array size and (2) rehashing the current elements of the table from left-to-right. The second hash table is implemented with a separate chaining strategy. It does not rehash when its load factor is too high. Both hash tables initially start with a backing array of size 3. Make sure to write both the key and value in the table rather than just the key.

The keys of the hash table are objects of type C. The following table describes the hash values of these objects:

| c1 | 0 |
| c2 | 3 |
| c3 | 1 |
| c4 | 5 |

(a) Map<C, Character> m = new HashMap<>(); m.put(c1, '+');

Probing

Chaining

(b) m.put(c2, '-')

Probing

Chaining

(c) m.put(c3, '*')

Probing

Chaining
Consider the following weighted, undirected graph. Clearly highlight in the graph a minimum spanning tree (MST) for the graph. You may use any method to compute the MST of this graph. In addition, give the total weight of the MST.

**Kruskal's Algorithm**

1) HC (1)
2) ED (2)
3) AB (3)
4) DF (3)
5) HB (5)
6) AE (5)
7) HG (8)
Problem 4: High and Dry (20 points)

Write a class, Skiperator<T> that implements the Iterator<T> interface and iterates through every nth element of an underlying list through its provided iterator, starting with the first element. For example, if the list contains the elements [1,2,3,4,5,6,7,8,9], then a skiperator with \( n = 2 \) produces the sequence 1,3,5,7,9 whereas a skiperator with \( n = 3 \) produces the sequence 1,4,7. If \( n = 1 \), then the skiperator behaves like a regular iterator through the list.

In your solution, assume the existence of the standard Iterator<T> interface that exposes boolean hasNext() and T next() methods.

Your Skiperator<T> class should have the following constructor and operations:

- Skiperator(Iterator<T> iter, int n): constructs a new Skiperator that moves through \( n \) elements in the sequence denoted by the given iterator. Assume that iter is non-null.
- boolean hasNext(): returns true iff the iterator still possesses elements.
- T next(): returns the current element the iterator points at and advances the iterator \( n \) elements; throws an IllegalStateException if hasNext() returns false.

```java
public class Skiperator<T> implements Iterator<T> {
    private int n;
    private Iterator<T> src;
    // pre: n > 0
    public Skiperator(Iterator<T> src, int n) {
        this.src = src;
        this.n = n;
    }

    public boolean hasNext() {
        return src.hasNext();
    }

    public T next() {
        if (!src.hasNext()) {
            throw new IllegalStateException();
        }
        T ret = src.next();
        for (int i = 0; i < n - 1 && src.hasNext(); i++) {
            src.next();
        }
        return ret;
    }
}
```
Problem 5: Orville (The Secret of Learning to Fly Is Forgetting to Hit the Ground) (20 points)

Consider the problem of determining friendship networks. Given a collection of people and friendship relationships between people, we would like to build a data structure that captures the different social networks between these people. For example, if Alice is friends with Bob and Bob is friends with Cynthia, then we say that Alice, Bob, and Cynthia form a social network because are connected via friendship relations (namely Bob’s mutual friendship with Alice and Cynthia).

With this data structure, we’d like to be able to record whether two people are friends and then determine if two people are in a social network together via a connection of mutual friendships.

To simplify the problem, let’s presuppose that there are only \( n \) people under consideration and they are labeled as numbers in the range \([0, n]\). Suppose that we start with seven people:

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

Initially, we don’t know anything about their relationships. We can think of each of the seven people as sitting in their own seven buckets where people that are in the same bucket are connected via some set of friendships. If we discover that 0 is friends with 1, we can represent this as follows:

\[
\begin{array}{ccccccc}
0 & 2 & 3 & 4 & 5 & 6 \\
\mid \\
1 & \\
\end{array}
\]

where we record that 0 and 1 are friends by making 1 a child of 0. In effect, our buckets are a collection of trees, or a forest of connected components. If we discover that 2, 5, and 6 are friends, we update our diagram as follows:

\[
\begin{array}{cccc}
0 & 2 & 3 & 4 \\
\mid & \\
1 & 5 \\
\mid \\
6 \\
\end{array}
\]

Where 2 is connected to 5 whom in turn is connected to 6. It might be the case that 2 is friends with 6 or 2 is friends with 5—we don’t care about the specific friendships in favor of efficiently storing which sets of people sit in the same social network.

Now, if we discover that 0 is friends with 2, we can record this as follows:

\[
\begin{array}{cccc}
2 & 3 & 4 \\
\mid \\
5 \\
\mid \\
6 \\
\mid \\
0 \\
\mid \\
1 \\
\end{array}
\]
Now 0, 1, 2, 5, and 6 are now all in the same social network through 0 and 2's friendship. We can tell if, for example, 1 and 6 are in the same social network by noting that if we follow the links from leaf to root, that they share the same representative element, 2 (their common root). We say that two friends share the same social network if they share the same representative element in our data structure.

(a) First, let’s make sure we understand how our data structure will work. First draw a diagram representing 6 people labeled 0 through 5, initially friendless.

```
0 1 2 3 4 5
```

Update the diagram so that 0 and 3 are friends.

```
0 1 2 4 5
   
3  
```

Update the diagram so that 1 and 5 are friends.

```
0 1 2 4
   
3 5
```

Update the diagram so that 3 and 1 are friends.

```
0 
1 2 4
3 
 
5
```

What are the representative elements of 0 and 4? Are 0 and 4 in the same social network?

Representative of 0: 0  
Not in the same network  
Representative of 4: 4  
(0 \neq 4)
What are the representative elements of 0 and 5? Are 2 and 8 in the same social network?

Representative of 0 = Representative of 5 = 0
0 and 5 are in the same network (same reps)

(b) Now let's implement this data structure. Note that we do not need to implement a full tree data structure to keep track of connectivity! It is sufficient to simply record the parents of each person in the data structure. Keeping in mind that each person is identified as an integer, we can use an array to remember these parents. Using our example from before,

```
(2)   (3)   (4)
|     |     |
(5)   |     |
(6)   |     |
(0)   |     |
(1)   |
```

We may represent it using the following array:

```
[6, 0, 2, 3, 4, 2, 5]
```

The ith element of the array records the parent of this person in the data structure. When that person is a root, then the element is its own index. For example, the parent of 1 is 0 (arr[1] == 0). In contrast, 3 is the root of its connectivity tree because index 3 of the array is 3.

With this representation in mind, write a class `FriendNetwork` that tracks the connectivity between a finite set of people labeled 0 to n - 1 (for some n) as defined above. First, write the class definition and any relevant fields you need. Then write the constructor for this class:

- `FriendNetwork(int n)`: constructs a new friend network of n people labeled 0 to n - 1 where there are no friendships.

```java
public class FriendNetwork {
    private int[] data;
    public FriendNetwork(int n) {
        data = new int[n];
        for (int i=0; i<n; i++) {
            data[i] = i;
            3
        }
    }
    // rest of impl...
}
```
(c) Write the `representative` method of the `FriendNetwork` class—`int representative(int p)`—that returns the representative friend of `p`'s social network as described above. What is the worst-case time complexity of `representative`?

```java
public int representative(int p) {
    if (data[p] == p) {
        return p;
    } else {
        return representative(data[p]);
    }
}

Time complexity: \(O(n)\) (\(n = \# \text{ of people in structure}\))
```

(d) Write the `friend` method of the `FriendNetwork` class—`void friend(int p1, int p2)`—that records a friendship between two friends `p1` and `p2`. What is the time complexity of `connect`?

```java
public void friend(int p1, int p2) {
    int rl = representative(p1);
    int r2 = representative(p2);
    data[rl] = r2;
}

Time complexity: \(O(n)\)
```

// Note: you could also connect `p1` to `p2` or vice versa.

(e) Finally, write the `sharesNetwork` method of the `FriendNetwork` class that returns `true` if and only if the two input people share the same social network. What is the worst-case time complexity of `sharesNetwork`?

```java
public boolean sharesNetwork(int p1, int p2) {
    return representative(p1) == representative(p2);
}

Time complexity: \(O(n)\)
(Scratch page. Put extra work here. Please label it if it is important!)