Exam 2
(November 18, 2016)

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!

UID (Not Your Name): ____________________________

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/ 15</td>
</tr>
<tr>
<td>2</td>
<td>/ 15</td>
</tr>
<tr>
<td>3</td>
<td>/ 15</td>
</tr>
<tr>
<td>4</td>
<td>/ 15</td>
</tr>
<tr>
<td>5</td>
<td>/ 20</td>
</tr>
<tr>
<td>6</td>
<td>/ 20</td>
</tr>
<tr>
<td>Total</td>
<td>/ 100</td>
</tr>
</tbody>
</table>
Problem 1: The Melbourne Shuffle

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

1. The best case runtime of insertion sort is \( O(n^2) \). \( O(n) \)
   - F

2. The space complexity of merge sort is \( O(\log n) \). \( O(n) \)
   - F

3. Selection sort is a stable sorting algorithm. Swaps break stability
   - F or -T

4. We may prefer quicksort over merge sort in situations where we are concerned about space complexity.
   - T

5. We may prefer insertion sort over merge sort in cases where we do not have all the input to sort immediately available.
   - T

6. We may prefer insertion sort over quick sort for large input arrays.
   - F

The Partition Operation. Consider the following array:

\[ [2, 8, 4, 0, 7, 1] \]

Give the step-by-step execution of the partition operation from quicksort as discussed in class using the element at index 2 (value 4) as the pivot. Your diagram should include the initial swap of the pivot with the end of the array as well as the final swap of the pivot back into its final position in the array. At each step you should:

- Indicate via arrows the location of your two "fingers" into the array.
- Move at most one pointer one position per step.

Finally, you should advance the pointers by first advancing the left pointer as far as it can go and then the right pointer as far as it can go rather than alternating between the two.

\[
\begin{align*}
\llbracket 2, 8, 1, 0, 7, 4 \rrbracket \\
\llbracket 2, 8, 1, 0, 7, 4 \rrbracket \\
\llbracket 2, 8, 1, 0, 7, 4 \rrbracket \\
\llbracket 2, 8, 1, 0, 7, 4 \rrbracket \\
\llbracket 2, 0, 1, 8, 7, 4 \rrbracket \\
\llbracket 2, 0, 1, 8, 7, 4 \rrbracket \\
\llbracket 2, 0, 1, 8, 7, 4 \rrbracket \\
\llbracket 2, 0, 1, 8, 7, 4 \rrbracket \\
\llbracket 2, 0, 1, 8, 7, 4 \rrbracket \\
\llbracket 2, 0, 1, 8, 7, 4 \rrbracket
\end{align*}
\]
Problem 2: TREEsemme

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(4);`

(c) `t.insert(8); t.insert(10);`

(d) `t.insert(6); t.insert(7); t.insert(2);`

(e) `t.remove(4);`
Given the following binary search tree, write the resulting sequences obtained by traversing the tree using each of the given strategies. (A dot (·) indicates that there is no node at that position in the tree).

(f) Pre-order traversal:
\[ \alpha, \beta, \gamma, \epsilon, \delta, \theta, \eta, \lambda, \kappa, \mu \]

(g) In-order traversal:
\[ \gamma, \delta, \epsilon, \beta, \alpha, \eta, \theta, \kappa, \lambda, \mu \]

(h) Post-order traversal:
\[ \gamma, \delta, \epsilon, \beta, \eta, \kappa, \mu, \lambda, \theta, \alpha \]
Problem 3: Purely Therapeutic

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 | 2 |
| c4 | 5 |

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

Probing

Chaining

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

Probing

Chaining

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

Probing

Chaining
(d) `m.put(c4, 'd')`

Probing: \[(c_1,'a'), (c_3,'c'), (c_2,'b'), (c_4,d')\]  
Chaining:

(e) `m.put(c1, 'e')`

Probing: \[(c_1,e'), (c_3,'c'), (c_2,'b'), (c_4,y')\]  
Chaining:
Problem 4: Mapping Mystery

Recall that the Java Stream<T> class exposes the following methods:

- `stream.map(Function<T, U> f)`: creates a stream of Us using function f.
- `stream.filter(Function<T, Boolean> f)`: filters the stream according to the predicate function f.
- `stream.reduce(U init, BiFunction<U, T, U> f)`: reduces the stream to a single value U using the initial value and f.

For each of the problems below, write the method chain of map, filter, and reduce calls required to produce the desired result. You may only use these methods in your solution.

(a) Given a stream s of strings, produce a stream of characters where the characters are drawn from the first letters of each of the strings (Hint: use the charAt(i) method of the String class).

\[ s, map (s \rightarrow s, charAt(0)) \]

(b) Given a stream s of strings, produce a stream of characters where the characters are drawn from the first letters of each of the strings, keeping only the characters that are digits (Hint: use the static isDigit(ch) method of the Character class).

\[ s, map (s \rightarrow s, charAt(0)) \]
\[ \cdot filter (c \rightarrow Character, isDigit (c)) \]

(c) Given a stream s of strings, produce true if the first letters of any of the strings is a digit.

\[ s, map (s \rightarrow s, charAt(0)) \]
\[ \cdot filter (c \rightarrow Character, isDigit (c)) \]
\[ \cdot reduce (false, (acc, c) \rightarrow acc + 1) \geq 1 \]

(d) Given a stream s of strings, produce true if the first letters of all of the strings is a digit.

\[ s, map (s \rightarrow s, charAt(0)) \]
\[ \cdot reduce (true , (acc, c) \rightarrow acc \&\& Character, isDigit (c)) \]
Problem 5: Riding Single File

Write a class, Zipator<T> that implements the Iterator<T> interface and iterates between two input iterators of the same carrier type, alternating between the two. For example, if the zipator is constructed with iterators to the following lists:

\[ [0, 1, 2], [3, 4, 5] \]

Then the zipator produces the sequence \[0, 3, 1, 4, 2, 5\]. Note that the zipator produces the first element of the first iterator, the first element of the second iterator, then their second elements, and so forth. Once one iterator is exhausted, the zipator consumes the remaining iterator like normal.

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

- Zipator(Iterator<T> fst, Iterator<T> snd): constructs a new Zipator that "zips" between the two given iterators. Assume that fst and snd are 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 forward.

```java
public class Zipator<T> implements Iterator<T> {
    private Iterator<T> fst;
    private Iterator<T> snd;
    private boolean drawFromFst;

    public Zipator(Iterator<T> fst, Iterator<T> snd) {
        this.fst = fst;
        this.snd = snd;
        drawFromFst = true;
    }

    public boolean hasNext() { return fst.hasNext() & snd.hasNext(); }

    public T next() {
        boolean fstNext = fst.hasNext();
        boolean sndNext = snd.hasNext();
        if ( !fstNext && !sndNext ) throw new IllegalStateException();
        if ( fstNext && !sndNext ) return fst.next();
        if ( !fstNext && sndNext ) return snd.next();
        T ret = null;
        if ( drawFromFst ) ret = fst.next(); else ret = snd.next();
        drawFromFst = !drawFromFst;
        return ret;
    }
}
```
Problem 6: Hogtied

(a) Consider the following two methods:

```java
public static String f1(String s1, String s2) {
    return s1 + s2;
}

public static String f2(String s1, String s2) {
    String ret = "";
    for (int i = 0; i < s1.length(); i++) { ret = ret + s1.charAt(i); }
    for (int i = 0; i < s2.length(); i++) { ret = ret + s2.charAt(i); }
    return ret;
}
```

If $n$ is the combined size of the input strings $s1$ and $s2$, what are the time complexities of each of the methods $f1$ and $f2$? You may simply state the run times without justification. (Hint: they aren’t the same! What is the runtime of a single concatenation ($+$) operation wrt to strings of length $n$?)

\[
\begin{align*}
    f1 & : O(n) \\
    f2 & : O(n^2)
\end{align*}
\]

Repeated string appends, especially with small strings, are extremely inefficient. In this problem, we build a data structure, called a rope, to alleviate these problems. A rope is a binary tree whose nodes correspond to string concatenation operations and leaves correspond to strings being concatenated. For example, the following rope:

```
    +
   /|
  +  "\text{"}
 /|
"hello"  "world"
```

represents the string concatenations "hello" + "world" + "!".
(b) To represent this in Java, we will follow the pattern of defining a tree's components in terms of an interface for its nodes. Define the Rope interface as follows:

```java
public interface Rope {
    /** @return the length of the rope---the number of characters it contains */
    public int length();
    /** Performs the concatenations contained in this rope and stores them
     * in the given char array starting at index i---assumes that the array
     * is large enough to store everything
     * @return the first index after the added characters to chs */
    public int concat(int i, char[] chs);
}
```

concat "collapses" a rope into a single string stored in a char array. For example:

- If you concat a leaf containing "hello" into an array starting at index 3, then the array contains [\ldots, 'h', 'e', 'l', 'o', \ldots] starting at index 3 and concat returns 8, the first index after the added characters.

- If you concat a node containing the concatenations "hello" + "world" into an array starting at index 5, then the array contains the characters "helloworld" starting at index 5 and concat returns 15.

Write a class RLeaf that implements the Rope interface and represents a leaf in a rope. In addition to the methods of the Rope interface, RLeaf defines a single constructor:

- `RLeaf(String s):` creates a rope leaf from the given string

*(Hint: The `length()` and `charAt(int index)` methods of the `String` class will be useful here.)*
(c) Write a class RNode that implements the Rope interface and represents a node in a rope. In addition to the methods of the Rope interface, RNode defines a single constructor:

Typo: Rope

- RNode(String s1, String s2): creates a rope node concatenating the two strings together.

```java
public class RNode implements Rope {
    private Rope left, right;

    public RNode(Rope left, Rope right) {
        this.left = left;
        this.right = right;
    }

    public int length() { return left.length() + right.length(); }

    public int concat(int i, char[] chs) {
        i = left.concat(i, chs);
        return right.concat(i, chs);
    }
}
```

*(if the args are strings, you should make Rleafs from them and store those as Ropes)*

(d) Finally, write a static method String concat(Rope r) that performs the concatenations found in the given rope and returns the resulting string. State the runtime of concat in terms of the length of the rope, n. *(Hint: The String class has a single-argument constructor that constructs a string from an array of characters. How big does this array need to be?)*

```java
public static String concat(Rope r) {
    char[] chs = new char[r.length()];
    r.concat(0, chs);
    return new String(chs);
}
```

\[ O(n) \]