Exam 2
(April 22, 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): 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:  Sort Sort Sort (20 points)

Consider the following array [3, 1, 5, 4, 2]. For each of the following sorts, demonstrate how the sorting algorithms operates over the array. To do this, write down the series of swaps that the algorithm performs as it sorts the array. For example, selection sort evolves the above array as follows:

1. [3 | 1, 5, 4, 2]  5. [1, 2, 3, 4 | 5]
2. [1 | 3, 5, 4, 2]  6. [1, 2, 3, 4, 5]
3. [1, 2 | 3, 5, 4, 3]
4. [1, 2, 3 | 4, 5]

In addition, you should note relevant pointers and invariants in your diagrams. For example above, the current index under consideration is marked with a caret (^) and the invariant split in the array is marked with a pipe (|).

(a) Insertion Sort (Note the current element to be inserted and sorted region of array.)

1. [3 | 1, 5, 4, 2]  5. [1, 3, 4, 2 | 5]
2. [1, 3 | 5, 4, 2]  6. [1, 3, 2, 4 | 5]
3. [1, 3, 5 | 4, 2]  7. [1, 2, 3, 4, 5]
4. [1, 3, 4, 5 | 2]

(b) The Partition Operation (Assume you are pivoting against the value 3. Note the position of the two pointers that perform the swaps.)

1. [3, 1, 5, 4, 2]  4. [2, 1, 5, 4 | 3]
2. [2, 1, 5, 4, 3]  5. [2, 1, 5, 4 | 3]
3. [2, 1, 5, 4, 3]  6. [2, 1 | 3, 4, 5]

(c) In a sentence or two, describe why we might prefer insertion sort over merge sort.

For small arrays, insertion sort has less constant-time overhead than mergesort. Furthermore, insertion sort has \( O(n) \) time on sorted arrays whereas mergesort is consistently \( O(n \log n) \).
Problem 2: TREEsemmé (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(7); t.insert(9);`

(d) `t.insert(3); t.insert(6); t.insert(8);`

(e) `t.remove(5);`
Given the following binary search tree, write the resulting sequences obtained by traversing the tree using each of the given strategies (○ denotes empty leaves):

```
  E
 /  \
B   G
 / \
A  C  F  H
    /\   \
   D  ○   
```

(f) Pre-order traversal:

```
E, B, A, C, D, G, F, H
```

(g) In-order traversal:

```
A, B, C, D, E, F, G, H
```

(h) Post-order traversal:

```
A, D, C, B, F, H, G, E
```
Problem 3: Hashers (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:

<table>
<thead>
<tr>
<th>c1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c2</td>
<td>6</td>
</tr>
<tr>
<td>c3</td>
<td>1</td>
</tr>
<tr>
<td>c4</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) `Map<C, Character> m = new HashMap<>(); m.put(c1, 1);`

Probing

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(c1, 1)</td>
</tr>
</tbody>
</table>
```

Chaining

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(c1, 1)</td>
</tr>
</tbody>
</table>
```

(b) `m.put(c2, 2)`

Probing

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>(c2, 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>(c1, 1)</td>
</tr>
</tbody>
</table>
```

Chaining

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>(c2, 2)</td>
<td>(c1, 1)</td>
</tr>
</tbody>
</table>
```

(c) `m.put(c3, 3)`

Probing

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(c2, 2)</td>
<td>(c3, 3)</td>
<td>(c1, 1)</td>
<td></td>
</tr>
</tbody>
</table>
```

Chaining

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(c2, 2)</td>
<td>(c3, 3)</td>
<td>(c1, 1)</td>
<td></td>
</tr>
</tbody>
</table>
```
(d) \texttt{m.put(c4, 4)}

Probing

Chaining

(e) \texttt{m.put(c1, 5)}

Probing

Chaining
**Problem 4: Purple Rain (20 points)**

Write a class, `Backerator<T>` that implements the `Iterator<T>` interface and iterates through every element of an underlying array list from *last to first* Assume the existence of a standard `ArrayList<T>` class that implements the `List<T>` interface with public methods `int size()`, `T get(int index)`, and `T remove(int index)`.

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

- `Backerator(ArrayList<T> list)`: constructs a new `Backerator` starting at the last element of the given list.
- `boolean hasNext()`: returns `true` iff the iterator still possesses elements.
- `T next()`: returns the current element the iterator points at and advances the iterator; throws an `IllegalStateException` if `hasNext()` returns `false`.
- `T remove()`: removes the current element the iterator points at, advancing the iterator, and returning that element; throws an `IllegalStateException` if `hasNext()` returns `false`.

```java
public class Backerator<T> implements Iterator<T> {
    private int index;
    private ArrayList<T> list;

    public Backerator(ArrayList<T> l) {
        list = l;
        index = list.size() - 1;
    }

    public boolean hasNext() {
        return index >= 0;
    }

    public T next() {
        if (!hasNext()) {
            throw new IllegalStateException();
        }
        return list.remove(index--);
    }
}
```
Problem 5: The Ninth Wonder (20 points)

(Note: this question has four parts.)

(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 respectively, what are the time complexities of f1 and f2 respectively? 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?)

\[ f1 : O(n) \]

\[ f2 : O(n^2) \]

Repeated string appends, especially with small strings, is 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:

```
+  
  + "!"
  "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 result of appending two ropes r1 and r2 (forming r1 + r2) */
    public Rope append(Rope other);
    // 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.)*

```java
public class RLeaf implements Rope {
    private String s;
    public RLeaf(String s) { this.s = s; }
    public Rope append(Rope other) {  
        RNode n = new RNode(null, null);
        n.left = this;
        n.right = other;
        return n;
    }
    public int concat(int i, char[] chs) {  
        for (int j = 0; j < s.length(); j++) {  
            chs[i++] = s.charAt(j);
        }
        return i;
    }
    public int length() {  
        return s.length();
    }
}
```
(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:

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

```java
public class RNode implements Rope {
    public Rope left, right;
    public RNode (String s1, String s2) {
        left = new RLeaf(s1);
        right = new RLeaf(s2);
    }

    public Rope append (Rope other) {
        RNode n = new RNode (null, null);
        n.left = this;
        n.right = other;
        return n;
    }

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

(d) Finally, write a static method \texttt{String concat(Rope r)} that performs the concatenations found
in the given rope and returns the resulting string. State the runtime of \texttt{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) {
    int sz = r.size();
    char [] chs = new char [sz];
    r.concat (0, chs);
    return new String(chs);
}
```

\( O(n) \)