Practice Final
(Final: Wednesday 5/18, 2:00–5:00)

Like the midterms, the final is closed-book, closed technology.

The examination period is 180 minutes.

The final covers *everything* that we have covered in the course.

Note that the exam will only be 10–11 questions (*i.e.*, a 2 hour exam in a 3 hour slot). I have included extra problems for practice as well as to introduce some variety in question styles that I will ask on the exam.

There will roughly be 6 non-programming questions and 4 programming questions. All relevant Java APIs and documentation will be given to you for the exam.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>Total</td>
</tr>
</tbody>
</table>
Problem 1: Just the Truth (10 points)

For each of the following statements, determine whether it is true (T) or false (F).

1. An array list is a fixed-size, sequential data structure.
   
2. The time complexity of linked list deletion is $O(n)$ in the average case.
   
3. Given a hash function $h$ and values $v_1$ and $v_2$, if $h(v_1) = h(v_2)$ then $v_1 = v_2$.
   
4. The space complexity of merge sort is $O(n)$.  
   
5. For small arrays, insertion sort is preferable over quicksort.
   
6. Is-a relationships between classes are established through inheritance.
   
7. Non-static members of a class can access static members of the same class.
   
8. A heap is an appropriate data structure for modeling the relationships between family members.
   
9. A tree is an appropriate data structure for containing a set of elements (of the same type) that have no particular relationship between them.
   
10. A stream is a potentially infinite, sequential data structure.
Problem 2: The Memory Game (10 points)

Consider the following class declaration:

```java
public class C {
    public String s;
    public int x;
    public C(int x) { this.x = x; s = Integer.toString(x + 1); }
    public int f(int n) {
        // POINT B (first call to f)
        // POINT C (last call to f)
        if (n != 0) {
            D d = new D(n);
            return f(n - 1);
        }
        return x;
    }
    public static void start() {
        C c = new C(0);
        // POINT A
        int x = c.f(3);
        // POINT D
    }
}
```

Give complete stack-and-heap diagrams outlining the state of memory at each of the given points above, assuming that program execution begins with the `start` method. For objects on the heap, you should include the object’s class tag indicating the type of the object; you do not need to include their v-table.

**Point A:**

Stack

Heap

**Point B:**

Stack

Heap
Point C:

Stack  Heap

Point D:

Stack  Heap
Problem 3: Tracers (10 points)

Draw the step-by-step evolution of a binary search tree after each of the given insert and removal 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(1); t.insert(6);`

(d) `t.remove(5);`
Draw the step-by-step evolution of a priority queue after each of the given add and poll operations. The priority queue is a max priority queue—that is, poll returns the largest element in the queue. You do not need to distinguish between equivalent elements in your priority queue.

(e) PriorityQueue<Integer> pq = new PriorityQueue<>(); pq.add(5);

(f) pq.insert(3); pq.insert(2); pq.insert(1);

(g) pq.insert(4);

(h) pq.poll();
Problem 4: Silent Steeples (10 points)

Consider the following class hierarchy:

```java
public class A {
    public void f1() { System.out.println("A.f1"); }
    public void f2() { System.out.println("A.f2"); }
}
public class B extends A {
    public void f3() { System.out.println("B.f3"); }
}
public class C extends B {
    public void f2() { System.out.println("C.f2"); }
}
public class D extends A {
    public void f1() { System.out.println("D.f1"); }
}
```

For each of the following variable initialization statements, state (a) if it type checks and (b) if it does type check, what is the static and dynamic types of the variable.

(a) `B b = new B();`

(b) `D d = new C();`

(c) `A a = new D();`

(d) `A a = new C();`

For each of the combinations of variable initializations statements and method invocations, determine if (a) the method invocation type checks and (b) if so, the output of the method call.

<table>
<thead>
<tr>
<th></th>
<th>f1()</th>
<th>f2()</th>
<th>f3()</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>D d = new D();</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>A a = new C();</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 5: Abstract Abstractions (10 points)

Consider the following implementation of a fixed-size queue of integers in Java using an array.

```java
public class Queue {
    public int[] data;
    public int size;

    public Queue(int n) {
        data = new int[n];
        size = 0;
    }

    public boolean enqueue(int v) {
        if (size == data.length) {
            return false;
        } else {
            data[size++] = v;
        }
    }

    public int dequeue() {
        if (size == 0) {
            throw new IllegalStateException();
        } else {
            int ret = data[0];
            for (int i = 0; i < size - 2; i++) {
                data[i] = data[i+1];
            }
            size -= 1;
            return ret;
        }
    }
}
```

Fix this implementation so that it can hold any type by using generics. Rather than completely rewriting the new class, rewrite only changed lines of the queue in the empty space to the right of the code listing.
Problem 6: Complex Numbers (10 points)

(a) Consider the following functions:

```java
public int f(int[] arr, int i) { /* Write the counts in the comments below */
    int x = arr[i]; /* */
    arr[i] = arr[i+1]; /* */
    arr[i+1] = x; /* */
}
public int g(int[] arr) {
    for (int i = 1; i < arr.length - 1; i++) { /* */
        f(arr, i); /* */
        f(arr, i - 1); /* */
    }
}
```

For each statement contained in the functions above, write the exact number of array accesses performed by that statement (both reads and writes). For function calls, the number of array accesses should be given in terms of its arguments when appropriate. For loops, you should give the total number of array accesses in terms of the length of the input array `arr` (call the length `n`) when appropriate.

(b) Give the run time complexity of `f` and `g` in terms of the length `n` of the input array.

(c) Give a recurrence relation describing the time complexity of the following function in terms of the size of its input `n`. You do not need to solve this recurrence relation for an overall time complexity.

```java
public void foo(int n) { /* Give the recurrence here */
    if (n <= 0) {
        return 0;
    } else {
        return n + foo(n - 2) + foo(n - 3);
    }
}
```
Problem 7: Reasons (10 points)

Consider the following Java method:

```java
public static String calculate(List<Character> chs) {
    String s = "";
    if (chs.size() == 0) {
        return s;
    }
    // POINT A
    s = chs.get(0).toString();
    for (int i = 1; i < chs.size(); i++) {
        s += ", " + chs.get(i);
        // POINT B
    }
    // POINT C
    return s;
}
```

For each of the propositions, determine if the proposition never holds (X), sometimes holds (?), or always holds (√) at the given program points.

<table>
<thead>
<tr>
<th></th>
<th>ss.size() &gt; 0</th>
<th>s.length &lt; ss.size()</th>
<th>s.length == ss.size() * 2 - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a sentence or two, describe what `calculate` does in terms of its inputs and output.
**Problem 8: The Slipstream (15 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>Key</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>2</td>
</tr>
<tr>
<td>c2</td>
<td>5</td>
</tr>
<tr>
<td>c3</td>
<td>6</td>
</tr>
<tr>
<td>c4</td>
<td>1</td>
</tr>
</tbody>
</table>

(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 Chaining

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

Probing Chaining

On the following page, execute merge sort by hand on the given array. You should show your work using a binary tree style where the nodes of the tree are sorting sub-problems and the two children are the recursive invocations of the merge sort algorithm. At each node, you should write the sub-array (a) before sorting and (b) after merging. Write it in the format:

<before sorting> → <after sorting>.

Below is an example of such a tree for your reference.

```
[4, 2, 9, 1] → [1, 2, 4, 9]
```

```
[4, 2] → [2, 4]  [9, 1] → [1, 9]
```

```
```
Execute merge sort on the array below, showing your work in the provided space:

\[9, 8, 3, 7, 5, 2, 6, 1, 4, 0\]
Problem 9: Streamlined (20 points)

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

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

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

(a) Given a stream of integers s, produce a stream of booleans where an entry in the new stream is true iff the corresponding integer is odd.

(b) Given a stream of passwords s represented as strings, produce a stream of passwords that only contains passwords of at least length 6 (Hint: use String.length()).

(c) Given a stream of strings s from a file, produce the total number of letters found in the file (Hint: use String.length()).

(d) Given a stream of passwords s represented as strings, produce a boolean indicating if any of them contain the word "password" using any mixture of case (Hint: use String.contains(String) and String.toLowerCase()).
Problem 10: Classy (20 points)

Write a class Complex that represents a complex number. Recall that a complex number is composed of two parts, a real number and an imaginary number, represented by doubles. A complex number is typically written $r + mi$ where $r$ is the real part and $m$ is the imaginary part. $i$ here is the imaginary number $i = \sqrt{-1}$. Your class should support the following constructors and operations:

- **Complex(r, m)**: creates a new complex number of the form $r + mi$.
- **double re()**: returns the real part of the complex number.
- **double im()**: returns the imaginary part of the complex number.
- **Complex add(o)**: returns the result of adding this complex number to the other given complex number $o$. Addition over complex numbers is defined by adding their respective parts.
- **Complex conjugate()**: returns the complex conjugate of this conjugate. The conjugate of a complex number $r + mi$ is defined as $r - mi$.
- **String toString()**: returns the string representation of the complex number in the form "$r + mi$" or "$r - mi$" if the imaginary portion is positive or negative, respectively.

Your work will be graded on correctness as well as design. In particular, you should only expose members that the interface defined above says you should expose.
Problem 11: Twin Peaks (25 points)

Write a class, `ZipIterator<T1, T2>`, that iterates through two lists in a pairwise fashion. For example, if \(l_1 = [1, 2, 3]\) and \(l_2 = [4, 5, 6]\) then a ZipIterator should produce the sequence of values \((1, 4), (2, 5), \text{and} (3, 6)\), respectively. A ZipIterator may only be constructed with lists that have the same size.

Assume the existence of a standard `Pair<T1, T2>` class:

```java
public class Pair<T1, T2> {
    private T1 t1; private T2 t2;
    public Pair(T1 t1, T2 t2) { this.t1 = t2; this.t1 = t2; }
    public T1 fst() { return t1; }
    public T2 snd() { return t2; }
}
```

Your ZipIterator class should have the following constructor and operations:

- **ZipIterator(l1, l2):** constructs a new ZipIterator over the given Lists \(l_1\) and \(l_2\) of types `List<T1>` and `List<T2>` respectively. If \(l_1\) and \(l_2\) do not have the same sizes, then the constructor throws an `IllegalArgumentException`. (Hint: use `List.size()` to retrieve the sizes of \(l_1\) and \(l_2\).)
- **boolean hasNext():** returns `true` iff the ZipIterator still possesses elements.
- **Pair<T1, T2> next():** returns the pair of elements from the two lists the iterator currently points at (packaged in a `Pair` object) and advances the iterator.
Problem 12: Trie’ing Hard (25 points)

A trie is a tree-like data structure for efficiently storing collections of strings. We can think of a trie as either a node that represents a position in a string with branches correspond to the possible letters in that position or a leaf that contains nothing. For example, the strings “tri”, “trip”, “trie”, and “try” can be represented by the following trie:

```
  o
  t
  r
  i/
y
    ■
  e/
p
    ■
```

Nodes marked with black squares (■) denote nodes that terminate words. We elide the branches for the other characters that do not appear in our set of words.

(a) First, let’s design the Node class of our trie. The nodes of a trie contain the following state:

- An array of nodes of size 26 which correspond to the 26 possible characters at each position. If an entry in this array is null, then there is no branch for that particular letter.
- A boolean flag indicating if the path of letters traversed to reach this node (from the root) constitute a word.

Create a Node class with these fields below. The fields may be marked public for convenience. The Node class has a single constructor:

- Node(boolean): constructs a new Node with the given boolean flag indicating if this node ends in a word of our trie.
(b) Using your Node class, write a Trie class that represents a trie data structure. Your Trie class must support the following constructors and operations:

- **Trie()**: creates a new, empty Trie.
- **void add(String)**: adds the given string to the trie.
- **List<String> fetchAll(String)**: returns a list of strings that are all the strings contained in the trie that start with the given prefix (i.e., the string argument).
Problem 13:  Mind the Gap (25 points)

Imagine that you are writing a text editor program. The key piece of data in such a program is the **text buffer**. Because text is linear in nature, our first inclination is to store the text in an array, e.g., an array list of characters:

```
['h', 'e', 'l', 'l', 'o', '
', 'w', 'o', 'r', 'l', 'd']
```

(Recall that character literals in Java are written with single quotes and '\n' is the character literal for a newline. Note that newlines are not treated specially—they are simply additional characters in the buffer).

The caret above represents the position of the cursor in the buffer. The cursor is an essential part of a text buffer because we can only edit text (add or remove characters) at the cursor.

(a) There is a big performance problem with this choice of representation, however, when we try to insert text into the buffer. What is it? Demonstrate it on the sample array list of characters give above (Hint: consider what array list operation we need to perform to insert text, and what happens if the text buffer is very large).

(b) How can we perform insertion at the cursor more efficiently? Describe an alternative way of representing text so that insertion is $O(1)$ (potentially amortized over a fraction of the text). Demonstrate your scheme on the example list of characters above. (Hint: split the text up...)
(c) Design a data structure that allows us to perform insertion into the text at the cursor more efficiently than the naive scheme described above. Implement this in a class called TextBuffer on the following page. Your TextBuffer class should contain the following constructor and methods:

- `TextBuffer()`: creates a new, empty text buffer.
- `void move(n)`: moves the cursor `n` characters forward in the buffer (an integer); a negative `n` causes the cursor to move backwards
- `void insert(ch)`: inserts the given character at the cursor and advances the cursor (operates in O(1) amortized time)

The following methods of the ArrayList class will be helpful in your implementation.

- `void add(v)`: adds `v` to the end of the array list.
- `T remove(index)`: removes the element at the given index from the array list.