Practice Final
(Final: Friday 12/16, 9:00–12:00)

- 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.
- The final will have 12 questions total—think two midterms stapled together (that you have three hours to complete).
- There will roughly be 8 non-programming questions and 4 programming questions.
- Like the midterms, 60% of the points are in the non-programming questions and 40% of the points are in the programming questions.
- The questions on this practice final are more difficult than those you will find on the final.
- All relevant Java APIs and documentation will be given to you for the exam.
**Problem 1: Regurgitate**

Below is a table of various data structures and operations over those data structures. Fill in the blanks in the table with the average-case time complexity of each of these combinations. If a time complexity has certain assumptions about it—it is amortized or requires balancing—state so in your answer.

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>get</th>
<th>remove</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Singly-) Linked List</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Red-Black Tree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hash Map</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 2: Just the Truth

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. [T/F]  
(2) The time complexity of linked list deletion is $O(n)$ in the average case. [T/F]  
(3) Given a hash function $h$ and values $v_1$ and $v_2$, if $h(v_1) = h(v_2)$ then $v_1 = v_2$. [T/F]  
(4) The space complexity of merge sort is $O(n)$. [T/F]  
(5) For small arrays, insertion sort is preferable over quicksort. [T/F]  
(6) Is-a relationships between classes are established through inheritance. [T/F]  
(7) Non-static members of a class can access static members of the same class. [T/F]  
(8) A heap is an appropriate data structure for modeling the relationships between family members. [T/F]  
(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. [T/F]  
(10) A stream is a potentially infinite, sequential data structure. [T/F]  
(11) The best-case time complexity of insertion sort is $O(n^2)$. [T/F]  
(12) The time complexity of tree deletion is $O(\log n)$. [T/F]  
(13) We use parametric polymorphism whenever we implement two methods with the same name but different arguments. [T/F]  
(14) We can use a tree-like structure to efficiently store a sorted set of data. [T/F]  
(15) The semantic heap invariant states that the children of any given node is less than the value found at the node. [T/F]
Problem 3: The Memory Game

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) {
            C c = new C(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 4: Tracers

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();
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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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) \texttt{m.put(c4, 'd')} \\
\hspace{1cm} \text{Probing} \hspace{3cm} \text{Chaining}

(e) \texttt{m.put(c1, 'e')} \\
\hspace{1cm} \text{Probing} \hspace{3cm} \text{Chaining}
Problem 5: Silent Steeples

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 initialization 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 6: Abstract Abstractions

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 7: Complex Numbers

For each of the following methods:

1. Give a mathematical function or recurrence that models the time complexity of the method. State explicitly what operations your function tracks as well as what the input to the function represents. If your model is a recurrence relation, solve that relation for an explicit mathematical function.

2. Give a tight upper-bound for your function using Big-O notation. You can simply state the upper-bound rather than formally proving it correct.

```java
// pre: arr1, arr2 != null
public static int f1(int[] arr1, int[] arr2) {
    int[] ret = new int[arr1.length * 2 + arr2.length];
    for (int i = 0; i < arr1.length; i++) {
        ret[i] = arr1[i];
        ret[ret.length - 1 - i] = arr1[i];
    }
    for (int i = 0; i < arr2.length; i++) {
        ret[arr1.length + i] = arr2[i];
    }
    return ret;
}

// pre: arr != null
// assume f2 is called initially with lo = 0, hi = arr.length
public static boolean f2(int v, int[] arr, int lo, int hi) {
    if (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (v < arr[mid]) {
            return f2(v, arr, lo, mid);
        } else if (v > arr[mid]) {
            return f2(v, arr, mid + 1, hi);
        } else {
            return true;
        }
    } else {
        return false;
    }
}
```
Problem 8: Reasons

Consider the following Java method:

```java
// pre: chs != null, chs does not contain any '\0' chars
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>Proposition</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>ss.size() &gt; 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.length() &lt; ss.size()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.length() == ss.size() * 2 - 1</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.

The `calculate` method takes a list of characters as input and returns a string containing the characters, separated by commas. If the list is empty, it returns an empty string. Otherwise, it concatenates the first character with commas and all subsequent characters.
Problem 9: Classy

Write a class `Point3D` that represents a point in three-dimensional space. A point is composed of three integers: the point’s `x`, `y`, and `z` coordinates. Your class should support the following constructors and operations:

- `Point3D(x, y, z)`: creates a new `Point3D` of the form `(x,y,z).
- `int getX(), int getY(), int getZ()`: returns the appropriate component of this `Point3D`.
- `Point3D add(Point3D other)`: adds this `Point3D` to its argument, returning a `Point3D` containing the result.

In addition, you should also override the three standard methods from the `Object` class:

- `String toString()` which returns a string of the form "(x, y, z)".
- `boolean equals(Object o)`. 
- `int hashCode()`.

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 10: Twin Peaks

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.
Problem 11: Going Higher

In class, we implemented the higher-order functions, map, filter, and fold over our list abstract data type. It turns out that we can implement these functions over virtually any data structure! In this problem, we’ll implement these higher order functions over a binary search tree.

Recall that in Java 8, we work with anonymous functions (lambda expressions) as follows:

- The type of a lambda is dictated by a number of functional interfaces defined in the java.util.function. For our purposes, we only care about two: Function<T, R>—the type of functions that take a T and produce an R and BiFunction<T, U, R>—the type of functions that take a T and a U and produce an R.

- To invoke an anonymous function, we call the method specified by its functional interface. For the Java standard library types, this method is called apply, e.g., f.apply(5) calls the lambda stored in f with the value 5.

- The syntax of a lambda itself is (parameters) -> <expression body>, e.g., (int x) -> x + 1 is a lambda that returns its argument, an integer, plus one.

For these methods, assume a standard definition of a binary search tree as presented in class:

```java
public class Node<T> {
    public T value;
    public Node<T> left;
    public Node<T> right;
    public Node(T value, Node<T> left, Node<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}

public class BinarySearchTree<T extends Comparable<T>> {
    private Node<T> root;
    // ...
}
```
(a) Complete the definition of the `map` method over the `BinarySearchTree` class. Recall that `map` takes a transformation function over elements and produces a new tree whose elements are the result of applying this function to the elements of the old tree. For example, we may map the following input tree:

```
2
/ \
1   3
```

To the following output tree:

```
true
/ \
false false
```

By passing a transformation function that returns true if the input is even. *(Hint: recall that you will need to write a static helper method that operates over the nodes of the tree.)*

```java
public <U> BinarySearchTree<U> map(Function<T, U> f) {
```

(b) Complete the definition of the `fold` method over the `BinarySearchTree` class. `fold` takes as input an initial value and a binary transformation function. This function takes the accumulated value so far as well as the current element of the tree and produces a new updated value. For example, if we fold the addition function over the following tree with an initial value of 0:

```
      2
     / \
    1   3
```

Then the result of the fold is $0 + 2 + 1 + 3 = 6$.
Assume that `fold` applies its argument function to the tree using an *in-order* traversal.

```java
public <R> R treeFold(BiFunction<R, T, R> f, R initial) {
```
(c) Complete the definition of the `filter` method over the `BinarySearchTree` class. `filter` takes as input an initial value and a function from elements of the tree to booleans. `filter` then keeps all the elements of the tree where the function (or `predicate`) returns `true` for that element. For example, we may map the following input tree:

```
      2
     / \
    1   3
```

To the following output tree:

```
      1
     / \
    -   3
```

If our filter returned `true` for any integer that was odd.

To implement `filter`, assume the existence of a method `Node remove(Node cur, T t)` in the `Node` class that removes the first occurrence of `t` from the tree, preserving the order of the binary search tree by promoting the next element in the in-order traversal of the tree to the removed element’s position. `remove` returns the updated version of `Node` passed to the method.
Problem 12: Connected

Consider the problem of network connectivity. Given a collection of computers, we would like to build a data structure that captures the collection of computers and the network connections between them. For example, in the diagram below:

(A) -- (B) -- (C) -- (D)

Computer A is connected to computer B which is connected to computer C. We would therefore like to conclude that A has a connection to C (through B). In contrast, A is not connected to D because there is no path of connections that takes us to D from A.

When analyzing a fixed set of computers, we would like to be able to record whether two computers are connected to each other. After analyzing these computers, we would also like to determine if two computers are connected. Note that the exact chain of computers that establish this connection are not important for our purposes, just whether a connection exists. For example, suppose that we start with seven computers labeled A–G.

(A) (B) (C) (D) (E) (F) (G)

Initially, we don’t know anything about their connection structure so they are all unconnected. We can think of each of the seven computers as sitting in their own seven buckets where computers that are in the same bucket are connected. If we discover that A is connected to B, we can represent this as follows:

(A) (C) (D) (E) (F) (G)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>(B)</td>
</tr>
</tbody>
</table>

where we record that A and B are connected by making B a child of A. In effect, our buckets are a collection of trees, or a forest of connected components. If we discover that C, F, and G are connected, we update our diagram as follows:

(A) (C) (D) (E)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>(B) (F)</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>(G)</td>
</tr>
</tbody>
</table>

Where C is connected to F whom in turn is connected to G. We call C the representative element of the bucket containing C, F, and G because it is the root.

Now, if we discover that B is connected to G, we can record this as follows:

---(C) (D) (E)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) (F)</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(B) (G)</td>
</tr>
</tbody>
</table>
by joining A and C, the representative elements of the two buckets. Now A, B, C, F, and G are now all connected by forming a *tree* of connectivity. We can tell if, e.g., B and G are connected, by noting that if we follow the links from leaf to root, that they share the same representative element, C (their common root).

(a) First, let’s make sure we understand how our data structure will work. Draw a diagram representing 5 computers labeled A–E, initially unconnected.

Update the diagram so that A and D are connected.

Update the diagram so that B and E are connected.

Update the diagram so that D and B are connected.

What are the representative elements of A and E? Are they connected?

What are the representative elements of C and D? Are they connected?
(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 computer in the data structure. If we assign an integer to each computer, then an array suffices to remember these parents. Using our example from before,

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

We may represent it using the following array:

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

The $i$th element of the array records the parent of this computer in the data structure. When that computer is a root, then the element is its own index. For example, the parent of computer 1 is computer 0 ($\text{arr}[1] == 0$). In contrast, computer 3 is the root of its connectivity tree because index 3 of the array is 3.

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

```java
public class ComputerNetwork {
    // Write your fields here

    /** Constructs a new network of n computers (labeled 0 to n-1),
     * all initially disconnected. */
    public ComputerNetwork(int n) {
    }
}
```
(c) Write the `connect` method of the `ComputerNetwork` class—`void connect(int c1, int c2)`—that records a connection between two computers `c1` and `c2`. The method throws an `IllegalArgumentException` if either `c1` or `c2` are invalid computers for this network. What is the time complexity of `connect`?

(d) Write the `representative` method of the `ComputerNetwork` class—`int representative(int c)`—that returns the representative element of `c` as described above. The method throws an `IllegalArgumentException` if `c` is an invalid computer for this network. What is the `worst-case` time complexity of `representative`?
(e) Finally, write the connected method of the ComputerNetwork class—
`boolean connected(int c1, int c2)`—that returns `true` if and only if the two input computers are connected according to the network. The method throws an `IllegalArgumentException` if either `c1` or `c2` are invalid computers for this network. What is the worst-case time complexity of `connected`?

(f) Analyze your implementation of the ComputerNetwork. Describe (but do not implement) one optimization you could implement to make the `connected` method more efficient.