Practice Exam 2 (Exam 2 date: Friday 4/22)

Logistics:
- In-class, written exam.
- Closed-book, notes, and technology.
- All relevant Java API documentation will be provided for you.
- Exam is out of 100 points (10% of your overall grade).
- This practice exam mimics the layout and types of problems on the actual exam.
- All topics covered through Wednesday 4/20 are fair game.

Relevant Java API documentation:

- Stream<T>:
  - 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.
**Problem 1: Sort Sort Sort**

Consider the following array \([3, 8, 9, 1, 2]\). 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, insertion evolves the above array as follows:

1. \([3 \mid 8, 9, 1, 2]\)
2. \([3, 8 \mid 9, 1, 2]\)
3. \([3, 8, 9 \mid 1, 2]\)
4. \([3, 8, 1 \mid 9, 2]\)
5. \([3, 1, 8 \mid 9, 2]\)
6. \([1, 3, 8, 9 \mid 2]\)
7. \([1, 3, 8, 2 \mid 9]\)
8. \([1, 3, 2, 8 \mid 9]\)
9. \([1, 2, 3, 8, 9]\)

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

(a) **Selection Sort** (Note the current element and sorted region of array.)

(b) **The Merge Operation** (Assume you are merging the first two elements and the last three elements. Show the auxiliary array that contains the merge results. Note the current elements being considered.)

(c) In a sentence or two, describe why we might prefer *merge sort* over *quicksort*. 
Problem 2: TREETembedded

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(2); t.insert(9);`

(d) `t.insert(3); t.insert(7); t.insert(10);`

(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:

(f) Pre-order traversal:

(g) In-order traversal:

(h) Post-order traversal:
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:

<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) m.put(c4, 'd')
Probing Chaining

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

For each of the problems involving Java’s Stream<T> class 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 strings s, produce the total number of characters found in all those strings.
Problem 4: Iter-hater (20 points)

Write a class, Skiperator<T> that implements the Iterator<T> interface and iterates through every other element of an underlying linked list, starting with the first. Assume the existence of a standard Node<T> class with public value and next fields.

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

- Skiperator(Node<T> first): constructs a new Skiperator starting at the given node of some linked 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 forward.

Augment the following LinkedList<T> class with a method skiperator() that produces a Skiperator<T> that initially starts on the first element of the list.

```java
public class LinkedList<T> {
    private Node<T> first;
    // Your definition of skiperator() goes here:
}
```
Problem 5: A Blooming Future (20 points)

(Note: Note this problem is significantly longer than the one that will appear on the second midterm, but I wanted to include because it covers a lot of what we’ll discuss in the final week leading up the exam. Irrespective of its length, this problem captures the type of question I’ll ask you on the exam: design a custom data structure that builds upon what you have learned in the course so far.)

Recall that a set is a collection of elements of the same type with the property that only one instance of a given element exists in the set. For example, the result of adding 5 to the set \{2, 5, 3\} is the original set. The critical operation of sets is membership or lookup where we check to see if an element is contained within a set.

We can efficiently implement a set using a hash map where the keys of the hash map are the elements of the set and the values are irrelevant. With such an implementation, it is sufficient to simply store these keys inside the map, irrespective of whether we use a probing or chaining scheme for resolving collisions. Lookup with this structure amounts to looking up a key which is very fast under appropriate conditions. We call such a data structure a hash set.

(a) While a hash set is very efficient, there are some limitations to the structure that are worth exploring. First, consider the runtime of hash set lookup. Give the best-case and worst-case time complexity for lookup. In at most two sentences, describe the situations in which the best- and worst-cases arise. In particular, what aspect of the hash set determines which situation we fall under?

(b) Now consider space efficiency. The space efficiency of the hash set is $O(n)$ where $n$ is the number of elements in the set. In a sentence or two, describe what we must store in the backing array of our hash set to support add and lookup and why we must do so. In another sentence, describe when storing these things may be infeasible. (Hint: a hash map stores a key-value pair, but we don’t need to store the value. What purpose did storing the key have?)
(c) Hash functions are critical for enabling fast lookup of values. However, because two distinct values are allowed to hash to the same value, we must spend considerable effort resolving collisions to ensure correctness. What if we didn’t need to be completely correct? Can we improve on the problems raised in parts (a) and (b)? This is the motivation behind an efficient probabilistic data structure for sets called a Bloom filter.

A Bloom filter over values of type $T$ consists of:

1. An array of booleans of some fixed size $n$.
2. A collection of hash functions $h_1, \ldots, h_k$ of type $T \to \text{int}$.

The implementations of the key set operations—add and contains—have simple descriptions:

- To add an element $v$ to a Bloom filter, we run $v$ through each of the $k$ hash functions (modulo $n$) to generate $k$ indices into the array of booleans and set those indices to true.
- To check to see if an element $v$ is in a Bloom filter, we run $v$ through each of the $k$ hash functions (modulo $n$) to generate $k$ indices into the array of booleans and check to see if all of those indices are true.

As an example, consider a Bloom filter with a backing array of size ten that contains elements of type $C$ with three hash functions $h_1$, $h_2$, and $h_3$. Here is a table of example values of type $C$ and their associated output using each hash function:

<table>
<thead>
<tr>
<th>c</th>
<th>$h_1$</th>
<th>$h_2$</th>
<th>$h_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>c_2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>c_3</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Initially, the backing array consists of ten entries, all of which are false:


When we add $c_1$ to the set, we flip the booleans of indices 1, 3, and 5 because $h_1(c_1) = 1$, $h_2(c_1) = 3$, and $h_3(c_1) = 5$.


Checking to see if $c_1$ is in the set amounts to ensuring that indices 1, 3, and 5 are all marked true.

Below, draw the updated Bloom filter after adding $c_2$.

With this updated Bloom filter, is $c_3$ considered in the set? ________________
(d) Now, let’s implement the Bloom filter. Write a class called `BloomFilter<T>` that holds elements of type `T`. Your class should have the following constructor and methods:

- `BloomFilter(List<Function<T, Boolean>> hs, int n)`: constructs a new Bloom filter that contains elements of type `T`. The filter uses all the hash functions present in `hs` and possesses a backing array of size `n`.
- `void add(T v)` and `boolean contains(T v)`: as described in part (c).

Recall that the `Function<T, Boolean>` class is a functional interface for lambda expressions of type `T → Boolean`. In particular, to apply such a function, you invoke its `apply` method, e.g., `h.apply(c)` where `h` is the object of type `Function<T, Boolean>`. *(Hint: don’t forget to consider the size of the array when accessing indices!)*
(e) Finally, let’s briefly analyze the Bloom filter:

First, describe the primary drawback of Bloom filters with respect to the behavior of their `contains` operation:

Now the upsides! What is the runtime of `contains` of the Bloom filter? Let $n$ be the number of elements in the filter and $k$ be the number of hash functions.

While the asymptotic space complexity of a Bloom filter is the same as a hash map, $O(n)$, a Bloom filter is more space efficient. Keeping in mind your answer to part (b), how is a Bloom filter more space efficient than a hash map?