Midterm 2 Review

Note: There are many tricky problems in CS61BL Summer 2016 Midterm 2!

Streams

Intermediate Operations to Know

1. `stream()`: Converts a collection into a stream
2. `sorted(Comparator c)`: Sorts the stream using `c`, which can also be a lambda
3. `map(Function f)()`: Applies `f` to each element in the stream
4. `filter(Predicate p)`: Tests each element with `p` to see if it should remain in the stream

Terminal Operations to Know

1. `reduce(BinaryOperator accumulator)`: Applies the accumulator and combines the elements in the stream, two at a time. Returns an Optional.
2. `collect(Collector c)`: Turns the stream into a collection.
   - To a List: `Collectors.toList()`
   - To a Set: `Collectors.toSet()`
   - To a Map: `Collectors.grooupingBy(Function f)`
3. `forEach(Consumer action)`

Practice Problems

Given the following classes, write methods to perform the desired operations. You may only use one semicolon per method.

```
public class Student {
   public int getLabSection() { ... }
   public int getAge() {...}
   public Group getGroup {...}
}

public class Group {
   public int size() {...}
   public Group merge(Group other) { ... }
}
```

1. Collect all the unique lab sections from a list of students. There should not be duplicates.

```
public Collection<Integer> uniqueSections(List<Student> lst) {




}
```

2. Each TA submits a list of their students. Write a method that will map each lab section to the students in that section whose group is larger than size 4. Assume there is always at least one student in each section whose group size is larger than 4.

```
public Map<Integer, List<Student>> grpMap(List<List<Student>> lst) {




}
```

3. Each TA submits a list of their students. Write a method that will get the number of students who are over age 20.

```
public int over20 (List<List<Student>> lst) {




}
```

4. Each TA submits a list of the groups in their section. Write a method that will keep only the sections with less than 4 groups in them. Then, merge all the groups in a section into one supergroup. If some section has 0 groups, then leave that section's supergroup as null instead. You should end up with a list of supergroups. (Use the `merge` method of the `Group` class)

```
public List<Group> groupmerge(List<List<Group>> lst) {




}
```

Higher Order Functions

- Function objects that can be passed around to other functions
- Pass method references using '::' (`User::getAge`, no parentheses!)
- `Function<T, R>` Interface:
  - `R apply(T item)`: will apply function on `item` of type `T`, returning a value of type `R`.
  - `User::getAge` is a `Function<User, Integer>` object; it takes in a User object and returns the integer age of that User object
- `Predicate<T>` Interface:
  - `boolean test(T item)`: will test the predicate on `item` of type `T`, returning `true` or `false`. (pred.test(o1))

Lambda Functions

- Can create anonymous functions, and can be used in place of objects that implement a functional interface (ex. `Comparator`)
- (arguments) -> (expression)
- `(u1, u2) -> (u1.getAge() - o2.getAge())`

Interfaces and Abstract Classes

Interfaces	Abstract classes
- `public interface` (interfaceName) - can `implement` many interfaces - cannot be instantiated - static and final variables only - all methods are abstract (unless noted as default) - access modifier is public	- `public abstract class` (className) - can `extend` only one class - cannot be instantiated - can have variables - all methods are default methods unless noted abstract - access modifier can be anything but private

Iterators/Generics

Generics

- Like in normal functions that take in function arguments, one can think of generics as type arguments. If one specifies a specific type as the generic `T`, any `T` that appears within the class will be replaced with the specified type.
- Generics can be specified with any word, though the convention is that a generic type is represented as a capital letter. (ex. `T`)

Interface Contract

1. `Iterator<T>`
   - `T hasNext()`: Returns if there are values left to return from this `Iterator`. Should not change the state of the iterator.
   - `T next()`: Calculates and returns the next value. If there are no elements left to return, should throw an `NoSuchElementException`.
   - `T remove()`: Default method, implemented to throw `UnsupportedOperationException`. Can override this method if desired.
   - Any class that implements this interface should contain state-saving variables to keep track of where the `Iterator` object is in the iteration. One analogy is that the data one is iterating through is a book and the iterator should be used as a bookmark. It may be useful to set up invariants (statements that should always be true) for the state-saving variables.
2. `Iterable`
   - `Iterator<T> iterator()`: This method returns an `Iterator<T>` object. The returned `Iterator` object will iterate over the specified type `T`.
   - Any class that implements this interface can be used in the enhanced for loop. The `for` loop will implicitly create an `Iterator` object from what is on the right-hand-side of the colon, and save what is outputted from `next` to the object defined on the left-hand-side, so long as `hasNext()` outputs `true`. The two iteration schemes shown below are identical.

```
ArrayList<Amoeba> amoebas = ...;
for (Amoeba a : amoebas) {
   do something with a
}
```

```
Iterator i = o.iterator();
while (i.hasNext()) {
      do something with i.next()
}
```

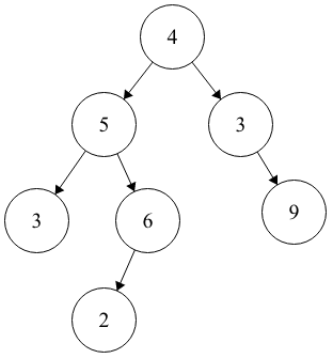
Practice Problems

1. We have a `FunkyRestaurant` that serves `LambSkewer` and `Tsingtao` in alternating turns. If the restaurant runs out of either one, it should just serve the other until it completely runs out of food. **(This is slightly different from quiz 10)**

   `LambSkewer` and `Tsingtao` are subtypes of `Food`. Fill out the code as necessary.

```
public class FunkyRestaurant implements Iterable<Food> {
    Iterable<LambSkewer> lambSkewers = ...;
    Iterable<Tsingtao> tsingtao = ...;

    @Override
    public Iterator<_________> iterator() {
        Iterator<__________> lambIter = lambSkewers.iterator();
        Iterator<__________> tsingtaoIter = tsingtao.iterator();
        return new FoodIterator<_____________>(lambIter, tsingtaoIter);
    }
}

public class FoodIterator<T1 extends _______, T2 extends ________>
    implements Iterator<_____________> {
    Iterator<T1> iter1;
    Iterator<T2> iter2;
    ------------------------------------------

    public FoodIterator(Iterator<T1> iter1, Iterator<T2> iter2) {
        this.iter1 = iter1;
        this.iter2 = iter2;
      ------------------------------------------
    }

    @Override
    public boolean hasNext() {
      ------------------------------------------------------------------
    }

    @Override
    public ________ next() {









    }
}
```

2. Given a `BinaryTree`, `MaxIterator` should repeatedly iterate through the child with the largest item. For example, for the following BinaryTree.



Repeatedly calling `next()` should return 4, 5, 6, 2.

```
public class MaxIterator implements Iterator<Integer> {
    ____________________________;
    public MaxIterator(BinaryTree tree) {
        ____________________________;
    }
    public boolean hasNext() {
        ____________________________;
    }

    public ________ next() {
        TreeNode temp = ___________;
        if(____________________) {
           ________________________;
        } else if(____________________) {
           ________________________;
        } else {
            if(____________________________) {
                ______________________;
            } else {
                ______________________;
            }
        }
        return ___________;
    }
}
```

Comparable/Comparator

- `Comparable<T>`
  - `int compareTo(T o)`: If this object is less than the given `o`, returns a negative value. Else if this object is greater than the give `o`, returns a positive values. Else, returns 0.
- `Comparator<T>`
  - Any object that implements `Comparator<T>` will take two objects and compare them. Think about this object as a third-party that will compare the objects.
  - `int compare(T o1, T o2)`: If this object is less than the given `o`, returns a negative value. Else if this object is greater than the give `o`, returns a positive values. Else, returns 0.
  - This is a functional interface (only has one method in it). Thus, we can use higher order functions (lambda functions, method references, etc.) in place of a `Comparator` object.

Asymptotics

Practice Problems

1. Asymptotic Analysis: for the following methods, give the runtime in terms of N

```
public void f1 (int N) {
   if (N > 0) {
      for (int i = 0; i < N; i++) {
         System.out.println("hello");
        }
        f1(N/2);
    }
}
```

```
public int f2 (int N) [
   if (N <= 1) {
      return N;
    }
    return f2(N-1) + f2(N-1);
]
```

3. For each statement, answer true or false. If true, explain why and if false provide a counterexample.

   - If $f(n) \in O(n^2)$ and $g(n) \in O(n)$ are positive-valued functions (that is for all n, $f(n), g(n) > 0$), then $\frac{f(n)}{g(n)} \in O(n)$.

   - If $f(n) \in \Theta(n^2)$ and $g(n) \in \Theta(n)$ are positive-valued functions, then $\frac{f(n)}{g(n)} \in \Theta(n)$

4. Data Structure Runtimes
   For each of the following data structures, fill in the runtime for each of the listed operations. Specify what the best and worst case runtime scenarios are as well

   - `ArrayList`
     `add`:
     `add` to front:
     `get`:
   - `LinkedList`
     `add`:
     `add` to front:
     `get`:
   - `HashMap`
     With no assumptions made:
     `put`:
     `get`:

     Assuming a good hash code:
     `put`:
     `get`:

     Assuming no resizing:
     `put`:
     `get`:
   - `BinarySearchTree`
     With no assumptions made:
     `insert`:
     `contains`:

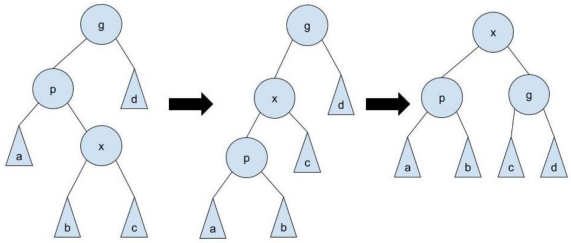
     Assuming a balanced tree:
     `insert`:
     `contains`:
   - `2-3-4/RB Tree`
     `insert`:
     `contains`:
   - `SplayTree`
     With no assumptions made:
     `insert`:
     `contains`:
     Assuming we are accessing recently used items:
     `insert`:
     `contains`:

Search Trees

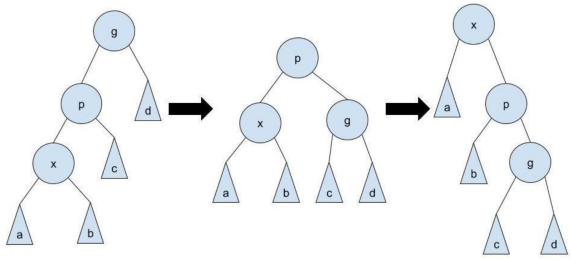
- For each node `n`, all values in the left subtree of `n` are less than the values at `n`. All values in the right subtree of `n` are greater than the values of `n`.
- Binary Search Trees
  - Each node can have 0, 1, or 2 children.
  - Methods to know: `insert(T item)`, `remove(T item)`, `contains(T item)`
- Balanced Search Trees
  - **2-3-4 Trees**
    - **Motivation:** Sometimes, our trees can become too tall so we instead stuff many items per node to decrease the height of our tree.
    - Put up to three items per node. Each node can have **2, 3, or 4** children. Increasing the height will take, on average, many insertions. Not responsible for removal.
    - To insert:
      - Traverse down the tree from root to leaf to find the correct place to put n
      - If you ever encounter a node with 3 items as you traverse down the tree, split that node by kicking up the middle item and creating two new nodes from the left and right items. Then reconnect the pointers of the nodes, maintaining the search tree property.
      - Once you are at a leaf, insert n in its correct spot in the leaf node.
  - **Red-Black Trees**
    - **Motivation:** 2-3-4 trees are hard to implement. Instead, have the same idea as a 2-3-4 tree but represent it in a binary tree with red and black nodes.
    - A red node means that it is a B-tree node with the parent node. Won't be responsible for inserting and removing from a red-black tree. Can translate any red-black tree into a 2-3-4 tree, insert into the 2-3-4 tree, and then translate back the the red-black tree.
  - **Splay Trees**
    - **Main Idea:** We have a binary search tree, but any time we do some `get`, `put`, or `remove` operation on some node n, we **splay** the node n (or the parent of n in the case of `remove`). If the node n is not found, the node that the search was terminated on will be splayed.
    - Splaying a node n means we do a series of rotations to make n the root of the tree. Although this isn't the same kind of balancing that 2-3-4 trees do, it allows for a special feature: **locality of reference**.
    - Locality of reference: items that are accessed the most are going to be near the top of the tree, allowing for faster accesses!
    - Rotations
      - `rotateLeft(x)`: rotate x left through its parent

- `rotateRight(x)`: rotate x right through its parent
- Analogy: Pick up the x node and allow other nodes to fall, reassign the middle pointer to the correct place.

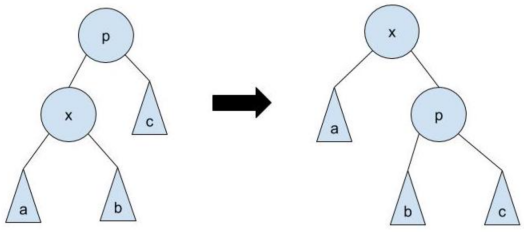
* `splayNode(Node n)`: To splay some node n, we must do as many rotations on n needed until it becomes the root. For any node n, we consider the position of its parent node p, and its grandparent node g to figure out what kind of rotation we need to do. Here are three cases you'll run into when you rotate n:
  - Zig-Zag: can only occur when n is the right child of p which is the left child of g. Will consist of `rotateLeft(n)` and `rotateRight(n)`.



  - Zig-Zig: can only occur when n is the right child of p which is a right child of g. Will consist of `rotateLeft(p)` and `rotateLeft(n)`. We must rotate the parent first



  - Zig: can only occur when n is the right child of p, the root of the tree. Will consist of one `rotateLeft(n)`.



  - Note: All these operations have a symmetric operation by swapping all 'left's with 'right's and vice versa.

Practice Problems

1. **Binary Search Trees: Snip**
   Assume that `BST` is as defined in lab. Fill in the function below so that any nodes with `value` greater than `limit` are destructively removed from the tree. Your implementation must *not* create any new nodes. Do not change the value of the nodes themselves.

```
public static BSTNode<Integer> trimHigh(BSTNode<Integer> node, int limit) {
   if (__________________) {

     ------------------------------------------------

    } else if (__________________) {

     ------------------------------------------------
    } else {

     ------------------------------------------------

       ------------------------------------------------
    }
}
```

2. **Binary Tree Traversal**
   write a method in the `BinaryTree` class that returns the items of the tree in reverse BFS order. The method should return an `ArrayList` of items. Assume root is not null.

```
public Arraylist<Object> reverseBFS() {









}
```

3. **Binary Trees: Symme-tree**
Given a binary tree, write a method `isSymmetric` that will check whether the tree is a mirror of itself (i.e. symmetric around its center). You may not need all the lines.

```
public class BinaryTree<T> {
    protected TreeNode root;
    protected class TreeNode {
        public T value;
        public TreeNode left;
        public TreeNode right;
    }

    public boolean isSymmetric() {
      ____________________________________________________

      ____________________________________________________

        ____________________________________________________

        ____________________________________________________
    }

    private boolean isSymmetric(TreeNode left, TreeNode right) {

      if (_____________________________) {

         ____________________________________________________

        } else if (_____________________________) {

         ____________________________________________________

        } else if (_____________________________) {

         ____________________________________________________
        } else {

            ____________________________________________________

            ____________________________________________________

            ____________________________________________________
        }
    }
}
```

Hashing

- **Main Idea:**
  - Use an array to keep track of values.
  - The index at which we put values will be based on `hashCode() % array.length`.
  - A tactic called external chaining will be used to keep track of multiple values in a particular bucket.
  - Load factor is defined as $\frac{\texttt{num items in array}}{\texttt{size of array}}$, and each `HashMap` will keep track of its maximum load factor. Once the load factor of the `HashMap` exceeds the maximum load factor, it will resize (usually by doubling the size of the array).
  - Most operations are constant time, though the resize operation will take linear time due to needing to re-hash all the elements of the data structure.
- `hashCode()`: An integer that represents a particular value. Can be used as an index within a hash map when modded by the length of the internal array. Should not be confused with being the actual index of the item in the `HashMap` (remember items can return negative values or super large values).
  - Validity:
    - If two objects are `.equals()`, then their `hashCode`s are the same.
    - If `hashCode()` is called multiple times on a single object, the `hashCode()` shouldn't change.
  - Goodness:
    - Must be valid before it can be considered good.
    - Evenly distribute values.
    - Shouldn't be too expensive to compute relative to the size of the given key.
    - Shouldn't rely on values that can change, based on your implementation of the object.

Data Structure Choices

Which data structure is best for the following:

1. A text editor history function that should has undo and redo functionality.

2. Given a file with a list of people and their corresponding ages, count the number of unique ages.

3. Given a list of reservations ordered by time, retrieve the ith earliest reservation.

4. A cache. Caches should save all added items and be very fast for frequent queries.