

Midterm 2 Review Solutions

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.groupingBy(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) {  
    return lst.stream()  
        .map(Student::getLabSection)  
        .collect(Collectors.toSet());  
}
```

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) {  
    return lst.stream()  
        .map(o -> o.stream()  
            .filter(getGroup.size() > 4)  
            .collect(Collectors.toList()))  
        .collect(Collectors.groupingBy(  
            o -> o.get(0).getLabSection()));  
}
```

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) {  
    return lst.stream()  
        .map(o-> o.stream()  
            .filter(s->s.getAge()>20)  
            .collect(Collectors.toList())  
            .size())  
        .reduce((o1, o2)-> o1+o2)  
        .orElse(0);  
}
```

```
//Or some other solution with flatMap  
public int over20 (List<List<Student>> lst) {  
    return lst.stream()  
        .flatMap(o-> o.stream())  
        .filter(s->s.getAge()>20)  
        .collect(Collectors.toList())  
        .size();  
}
```

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.

```
public List<Group> groupMerge(List<List<Group>> lst) {
    return lst.stream()
        .filter(o -> o.size() <4)
        .map(o -> o.stream()
            .reduce((o1, o2)->o1.merge(o2))
            .orElse(null))
        .collect(Collectors.toList())
}
```

Iterators/Generics

Interface Contract

1. Iterator

- `hasNext()`: This method should not change the state of the iterator.
- `next()`: This method should calculate and return the next value. If there are no elements left to return, should throw an `NoSuchElementException`.
- `remove()`: This method is, by default, implemented to throw `UnsupportedOperationException`.

2. Iterable

- `iterator()`: This method should return an `Iterator` object.
- Any class that implements this interface can be used in the enhanced for loop (shown below). The `for` loop will implicitly create an `Iterator` object from the right-hand-side of the colon,

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

Practice Problems

1. We have a `FunkyRestaurant` that emphasizes that it serve `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.

`LambSkewer` and `Tsingtao` are subtypes of `Food`. Fill out the generic type arguments, provide the implementation for `FoodIterator` so that the code works as desired.

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

    @Override
    public Iterator<Food> iterator() {
        Iterator<LambSkewer> lambIter = lambSkewers.iterator();
        Iterator<Tsingtao> tsingtaoIter = tsingtao.iterator();
        return new FoodIterator<LambSkewer, Tsingtao> (lambIter, tsingtaoIter);
    }
}

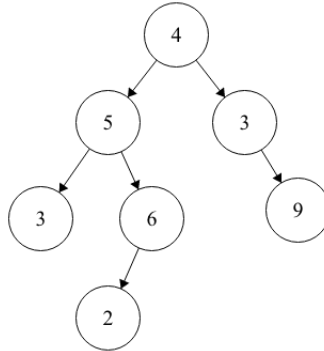
public class FoodIterator<T1 extends Food, T2 extends Food>
    implements Iterator<Food> {
    Iterator<T1> iter1;
    Iterator<T2> iter2;
    boolean first;

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

    @Override
    public boolean hasNext() {
        return iter1.hasNext() || iter2.hasNext();
    }

    @Override
    public Food next() {
        if (!iter1.hasNext()) {
            return iter2.next();
        } else if (!iter2.hasNext()) {
            return iter1.next();
        } else if (first) {
            first = !first;
            return iter1.next();
        } else {
            first = !first;
            return iter2.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> {
    TreeNode current;
    public MaxIterator(BinaryTree tree) {
        current = tree.root;
    }
    public boolean hasNext() {
        return current == null;
    }

    public Integer next() {
        TreeNode temp = current;
        if(current.left == null) {
            current = current.right;
        } else if(current.right == null) {
            current = current.left;
        } else {
            if(current.left.item > current.right.item) {
                current = current.left;
            } else {
                current = current.right;
            }
        }
        return temp.item;
    }
}
```

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);
 }
}
```

---

Solution:  $\Theta(N)$

---

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

Solution: $\Theta(2^N)$

2. 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 $\frac{f(n)}{g(n)} \in O(n)$.

Nope this does not hold in general! Consider $f(n) = n^2$ and $g(n) = \frac{1}{n}$

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

True! See more justification at <http://datastructur.es/sp17/materials/discussion/discussion8sol.pdf>

3. 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**: Best case $O(1)$, Worst case $O(n)$

Remember that an **ArrayList** is backed by an array. Best case, we add to the back of the array. Worst case, we add to a point towards the front of the array and need to shift all the elements one over. Or, we may have to resize the array if there is not enough space to add.

- **add to front**: Best case $O(n)$, Worst case $O(n)$

Adding to the front means we always have to shift all of the elements over by one index. This takes linear time with respect to the number of elements.

- **get**: Best case $O(1)$, Worst case $O(1)$

We can immediately index and find an element from an array in constant time.

- **LinkedList**

- **add**: Best case $O(1)$, Worst case $O(n)$

In the best case, we add towards the front or back of the list, which can be done in constant time. In the worst case, we have to add somewhere towards the middle of the list, and must traverse through most of the list to get to that index.

- **add to front**: Best case $O(1)$, Worst case $O(1)$

We can add to the front of a linked list in constant time.

- **get**: Best case $O(1)$, Worst case $O(n)$

In the best case, we get something from the front or end of the list. In the worst case, we must traverse through most of the list to arrive at the correct index.

- **HashMap**

With no assumptions on the hash code made:

- **put**: Best case $O(1)$, Worst case $O(n)$

Best case, the hash code has a uniform distribution. In the worst case, we have to resize the whole array, or the hash code is bad and maps everything to the same bucket,

- **get**: Best case $O(1)$, Worst case $O(n)$

Best case, the hash code has a uniform distribution, or the item we are trying to get is at the front of the linked list in its bucket. Worst case, the hash code maps everything to the same bucket.

Assuming a good hash code:

- **put**: Best case $O(1)$, Worst case $O(n)$

Best case, we put to the map without needing to resize. In the worst case, we must resize the array and rehash/reput all elements

- **get**: Best case $O(1)$, Worst case $O(1)$

In both cases we get from the map as normal

Assuming no resizing:

- **put**: Best case $O(n)$, Worst case $O(n)$
Both cases are linear time since the number of items in the map grows asymptotically very large. So, since the size of our map's array is constant, the size of each linked list grows asymptotically very large, regardless of how good our hash code is.
- **get**: Best case $O(n)$, Worst case $O(n)$
Same reasoning as above

- **BinarySearchTree**

With no assumptions made:

- **insert**: Best case $O(1)$, Worst case $O(n)$
Best case, we have a situation like this: all the nodes are in the root's right subtree, and we insert on the left side of the node. In the worst case, we have the same situation, but we must now insert into the end of the right subtree.
- **contains**: Best case $O(1)$, Worst case $O(n)$
Best case, we run contains on a node toward the top of the BST. Worst case, we have a spindly BST and run contains on a node towards the bottom of the BST.

Assuming a balanced tree:

- **insert**: Best case $O(1)$, Worst case $O(\log n)$
Best case is similar as above, but now the worst case is that the height of the tree is logarithmically proportional to the number of items in the tree
- **contains**: Best case $O(1)$, Worst case $O(\log n)$
Similar logic to above

- **2-3-4/RB Tree**

insert: Best case $O(1)$, Worst case $O(\log n)$
contains: Best case $O(1)$, Worst case $O(\log n)$

- **SplayTree**

With no assumptions made:

insert: Best case $O(1)$, Worst case $O(n)$
contains: Best case $O(1)$, Worst case $O(n)$
NOTE: Amortized $O(\log n)$

Assuming we are accessing recently used items:

insert: Best case $O(1)$, Worst case $O(1)$
contains: Best case $O(1)$, Worst case $O(1)$

Search Trees

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 (node == null) {
        return null;
    } else if (node.value > limit) {
        return trimHigh(node.left, limit);
    } else {
        node.right = trimHigh(node.right, limit);
        return node;
    }
}
```

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() {
    LinkedList<TreeNode> queue = new LinkedList<TreeNode>();
    ArrayList<Object> returnList = new ArrayList<Object>();

    queue.add(root)

    // Add the items to a stack in BFS order
    while(queue.size() > 0) {
        TreeNode currentNode = queue.getFirst();

        //Add to front of the ArrayList
        returnList.push(currentNode.item);

        queue.add(currentNode.left);
        queue.add(currentNode.right);
    }

    return returnList;
}
```

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).

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

    public boolean isSymmetric() {
        if (root == null) {
            return true;
        }
        return isSymmetric(root.left, root.right);
    }

    private boolean isSymmetric(Node left, Node right) {
        if (left == null) {
            return right == null;
        } else if (right == null) {
            return false;
        } else if (!left.value.equals(right.value)) {
            return false;
        } else {
            return isSymmetric(left.right, right.left) &&
                isSymmetric(left.left, right.right)
        }
    }
}
```

Data Structure Choices

Which data structure is best for the following:

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

Solution: DoublyLinkedList. Add to history by appending to front of DoublyLinkedList. Undo and redo by traversing through the DoublyLinkedList

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

Solution: HashSet. Add each person's age to the HashSet and find the size at the end.

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

Solution: Array. Copy reservations into the array and find i th earliest reservation by checking i th index.

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

Solution: Splay tree. Splay trees have amortized $\theta(\log(n))$ insert/find but $\theta(1)$ for frequent finds.