CS61BL Midterm 2 Review Solution

1 Problem Solving with Abstract Data Types

Write `hasDuplicates` which returns true if the input collection of `elements` has duplicates and false otherwise.

```
public static <E> boolean hasDuplicates(Collection<E> elements) {
    Set<E> seen = new HashSet<>();
    for (E element : elements) {
        if (seen.contains(element)) {
            return true;
        }
        seen.add(element);
    }
    return false;
}
```

Write `isPermutation` which returns true if the first string, `s1`, is a permutation of the second string, `s2`. The permutations of `cat`, for example, are: `cat, cta, act, atc, tca, and tac`.

```
public static boolean isPermutation(String s1, String s2) {
    Map<Character, Integer> charCounts = new HashMap<>();
    for (char c : s1.toCharArray()) {
        int count = 0;
        if (charCounts.containsKey(c)) {
            count = charCounts.get(c);
        }
        charCounts.put(c, count + 1);
    }
    for (char c : s2.toCharArray()) {
        int count = 0;
        if (charCounts.containsKey(c)) {
            count = charCounts.get(c);
        }
        charCounts.put(c, count - 1);
    }
    for (Integer count : charCounts.values()) {
        if (count != 0) {
            return false;
        }
    }
    return true;
}
```

2 Generic Question

Consider the `NumericSet` class below.

```
class NumericSet<T extends Number> extends HashSet<T> {
    /** implementation goes here */
}
```

(a) What is the purpose of the `extends` keyword?

`<T extends Number>` requires that the generic type `T` is some subclass of `Number`.

(b) What types or classes of objects can you add to a `NumericSet`?

Some examples include `Integer, Long, Float, Double`.

(c) Why is the generic type for `HashSet <T>` and not `<T extends Number>`?

Since we already declared the generic type bound for `T` earlier, there's no need to redeclare it again the next time we use `T`.

Now, consider the class header for the `BinarySearchTree` we introduced in lab.

```
class BinarySearchTree<T extends Comparable<T>> implements Comparable<T> {
    /** implementation goes here */
}
```

(a) What type of elements does an instance of `BinarySearchTree` hold?

A `BinarySearchTree` contains any one class of elements that is `Comparable` to itself.

(b) What method do all elements stored in a `BinarySearchTree` instance have in common?

We can guarantee that each element in the `BinarySearchTree` has a `compareTo` method because it implements `Comparable`.

(c) Why not define the generic type as `<? extends Comparable>`?

Defining the generic type this way would mean that we would lose reference to the specific type `T`. This means that the best guarantee we can make about a `BinarySearchTree` is that its elements are of some unknown type of object that is comparable to other objects (without bound).

3 VoteIterator

Write an iterator that takes in an `Integer` array of vote counts and iterates over the votes. The input array contains the number of votes each selection received. For example, if the input array contained the following:

`[0, 2, 1, 0, 1, 0]`

then calls to `next()` would eventually return 1 twice, 2 once, and 4 once. After that, `hasNext()` would return false. Provide code for the `VoteIterator` class below. Make sure your iterator adheres to standard iterator rules.

```
public class VoteIterator implements Iterator<Integer> {
    private Integer[] votes;
    private int index, current;

    public VoteIterator(Integer[] votes) {
        this.votes = votes;
        this.index = this.current = 0;
    }

    public boolean hasNext(){
        if (current < votes[index]) {
            return true;
        }
        for (int i = index + 1; i < votes.length; i++) {
            if (votes[i] > 0) {
                return true;
            }
        }
        return false;
    }

    public Integer next() {
        while (current == votes[index]) {
            index += 1;
            current = 0;
        }
        current += 1;
        return index;
    }

    public void remove() {
        votes[index] -= 1;
        current -= 1;
    }
}
```

4 db61bl

Let's implement a relational database management system! In this system, we manage *tables* consisting of several organizing *columns* and *rows* containing the entries for those columns.

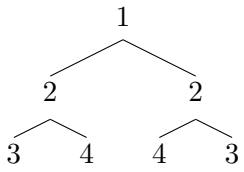
titles	SID	Lastname	Firstname	Major
rows	101	Yao	Alan	Food Systems
	102	Chen	Antares	Literature
	103	Nguyen	Daniel	Parrot Biology
	104	Lee	Maurice	Chemical Engineering
	105	Jian	Lisa	Cosmology
	106	Kim	Sarah	Parrot Biology

Fill out the implementations for the overloaded methods `select`, `export`, and `exportBy` on the next page. You may find it helpful to write the most abstract version of each method (the one that consumes the most arguments) first before writing the other convenience methods.

```
class Table {
    String name;
    String[] titles;
    Set<Row> rows;

    Table(String name, String... titles) {
        this.name = name;
        this.titles = titles;
        this.rows = new HashSet<>();
    }

    class Row {
        String[] data;

        Row(String... data) {
            this.data = data;
        }

        /** Return a new row with the entries specified by COLUMNS. */
        Row getRow(String... columns) {
            String[] newData = new String[columns.length];
            for (int i = 0; i < columns.length; i++) {
                int rowIndex = getColumnIndex(columns[i]);
                newData[i] = data[rowIndex];
            }
            return new Row(newData);
        }
    }

    /** Add multiple ROWSTOADD to this table and return this table. */
    Table addAll(Stream<Row> rowsToAdd) {
        rowsToAdd.forEach(rows::add);
        return this;
    }
```

```
    /** Return a table containing new rows where PRED is true. */
    Table select(Predicate<Row> pred) {
        return new Table(name, titles).addAll(select(pred, r -> r));
    }

    /** Return a table of rows consisting of COLUMNS from this table. */
    Table select(String... columns) {
        return select(r -> true, columns);
    }

    /** Return a table containing new rows from first applying PRED and
     *  then selecting the remaining COLUMNS. */
    Table select(Predicate<Row> pred, String... columns) {
        return new Table(name, columns).addAll(select(pred,
                                               r -> r.getRow(columns)));
    }

    /** Return a stream of rows applying PRED then SELECTOR to each. */
    Stream<Row> select(Predicate<Row> pred, Function<Row,Row> selector) {
        return rows.stream().filter(pred).map(selector);
    }

    /** Return a list of strings by applying GETTER to each row. */
    List<String> export(Function<Row,String> getter) {
        return export(r -> true, getter);
    }

    /** Return a list of strings from rows applying PRED then GETTER. */
    List<String> export(Predicate<Row> pred,Function<Row,String> getter) {
        return rows.stream().filter(pred).map(getter).collect(toList());
    }

    /** Return a map of strings to results grouped according to GROUPER
     *  and with values selected by GETTER. */
    Map<String, List<String>> exportBy(
            Function<Row,String> grouper, Function<Row,String> getter) {
        return exportBy(r -> true, grouper, getter);
    }

    /** Return a map of strings to results where the row satisfies PRED,
     *  grouped according to GROUPER, and values selected by GETTER. */
    Map<String, List<String>> exportBy(Predicate<Row> pred,
            Function<Row,String> grouper, Function<Row,String> getter) {
        return rows.stream().filter(pred).collect(groupingBy(grouper,
                                          mapping(getter, toList())));
    }
}
```

5 Trees

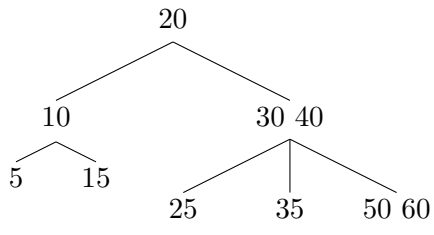


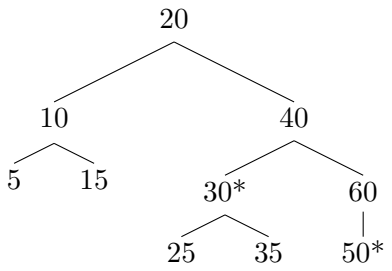
Define `isSymmetric` which checks whether the binary tree is a mirror of itself.

```
public class BinaryTree<T> {
    TreeNode root;

    public class TreeNode {
        T item;
        TreeNode left;
        TreeNode right;
    }

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

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

(a) Insert the following numbers into a 2-3 tree: `[20, 10, 35, 40, 50, 5, 25, 15, 30, 60]`

(b) Draw the corresponding left-leaning red-black tree.





6 Asymptotic Analysis

Provide the tightest asymptotic runtime bounds for the following methods: give a $\Theta(\cdot)$ bound if possible, otherwise give both the $\Omega(\cdot)$ and $O(\cdot)$ bound.

(a) Insertion of N **distinct** elements in sequence into the following data structures.

- `ArrayList` $\Theta(N)$
- `LinkedList` $\Omega(N), O(N^2)$
- `TreeSet` $\Theta(N \log N)$
- `HashSet` $\Omega(N), O(N^2)$

(b) Insertion of N **identical** elements in sequence into the following data structures.

- `ArrayList` $\Theta(N)$
- `LinkedList` $\Omega(N), O(N^2)$
- `TreeSet` $\Theta(N)$
- `HashSet` $\Theta(N)$

(c) Check that a single element is contained in a collection of N distinct elements.

- `ArrayList` $\Omega(1), O(N)$
- `LinkedList` $\Omega(1), O(N)$
- `TreeSet` $\Omega(1), O(\log N)$
- `HashSet` $\Omega(1), O(N)$

(d) Suppose we have a `HashMap` implementation that has an initial capacity of 1, and a load factor of 1. In this hash map implementation, each resizing of the hash map increases the number of buckets by 1. Assume we are using a good hash function, one that (miraculously) always divides our data sets evenly between the buckets. Give a $\Theta(\cdot)$ worst-case bound for the running times of the following operations.

- Inserting N distinct key-value pairs. $\Theta(N^2)$
- Looking up a single key after the N insertions above. $\Theta(1)$

(e) Now, suppose we have the same `HashMap` implementation, except resizing doubles the capacity, and we use a hash function that maps all keys to 0. Give a $\Theta(\cdot)$ bound for the average runtime of the following operations.

- Inserting N distinct key-value pairs. $\Theta(N^2)$
- Looking up a single key after the N insertions above. $\Theta(N)$

7 HashBrowns & HashMaps

Describe what would happen in each of the following scenarios.

(a) Define `String.hashCode` to return the length of the string.

This is a valid `hashCode`. However, the worst case runtime may approach $O(N)$ on certain inputs where the length of each string is the same.

(b) Define `String.hashCode` to return a random, uniformly-distributed number each time.

This is an invalid `hashCode` because elements inserted into the `HashMap` earlier based on an older `hashCode` might not be found under a different random number.

(c) Override the `equals` method without overriding the `hashCode` method.

This is invalid. `Object.equals` returns true only if the two objects are identical (pointer equality) and `Object.hashCode` maintains the relationship by returning the memory address of the object. By overriding the `equals` method, however, two different objects in memory may be considered equal. But because the `hashCode` method was not changed, those two objects may be assigned to different hash buckets.

(d) Override the `hashCode` method without overriding the `equals` method.

This causes the same problem as above.

(e) Modify the key of an entry inserted into a `HashMap`.

This is invalid as the key might hash to a different value and may no longer be found in the `HashMap`.

(f) Modify the value of an entry inserted into a `HashMap`.

This is valid as only the key is involved in the hash lookup process.

(g) Instead of a linked list, we use an array, BST, or hash table as the external chain.

In the best-case scenario with a uniformly-distributed `hashCode`, the length of an external chaining linked list will be, on average, about the same as the load factor. Because the load factor is usually a small constant, the length of the external chaining linked list will also be some constant as well due to amortized resizing.

In the worst-case scenario, however, all elements may hash to the same bucket causing search over the linked list in $O(N)$ time where N is the number of elements. This is where a balanced search tree might be advantageous if our elements are `Comparable` because of its guaranteed $O(\log N)$ search time. However, in practice, the overhead of setting up additional pointers and extra logic becomes more expensive than traversing a short linked list.

An array does not provide any additional advantages over a linked list because random access is not necessary: we need to iterate over all the elements in the chain. The drawback of arrays is in increased instantiation time: repeated resizing of the external chaining array can get expensive over time especially as arrays need to be reallocated as the hash table resizes.

It's possible to use another hash table as the external chaining mechanism. Although this may resolve hash collisions caused by the length of the table, for poor hash functions, this merely puts off the hash collision. See cuckoo hashing for an alternative scheme.