CS61BL: Data Structures & Programming Methodology Summer 2015

Instructor: Joseph Moghadam

Midterm 2

July 31, 2015

Name:	Dexter Xucheng Guo
Student ID Number:	26958386
Section Time:	08
TA:	Alice
Course Login: cs61bl-??	JF
Person on Left: Possibly "Aisle" or "Wall"	Wall
Person on Right: Possibly "Aisle" or "Wall"	Aisle

- Fill out ALL sections on this page. (1 point)
- Do NOT turn this page until the beginning of the exam is announced.
- Do not remove any pages from your exam, or write on any paper not included with the exam.
- Once the exam starts, fill out the header of each page in the exam with your login.
- You may not use outside resources other than your 1 page, 2-sided cheat sheet.
- You have 110 minutes to complete this exam.
- Your exam should contain 7 problems over 18 pages.
- This exam comprises 15% of the points on which your final grade will be based (45 points).
- If you have a question, raise your hand and a staff member will come to help you.
- Make sure to check for corrections / clarifications that will be periodically added to the screen at the front of the room.

Potentially Useful Documentation

Below is documentation of classes/interfaces from `java.util` you might find helpful in this exam.

```
public interface Queue<E>

    /** Inserts the specified element into this queue if possible without violating
        capacity restrictions. */
    public boolean offer(E e);

    /** Retrieves, but does not remove, the head of this queue, or returns null if
        this queue is empty. */
    public E peek()

    /** Retrieves and removes the head of this queue, or returns null if this queue is
         empty. */
    public E poll()
```

```
public class PriorityQueue<E> implements Queue<E>

An unbounded priority queue based on a heap. Does not permit insertion of non-
    comparable objects. The elements of the priority queue are ordered according to
    their implementation of Comparable. A priority queue does not permit null elements.
```

```
public interface Comparable<T>

    /** Compares this object with the specified object for order. Returns a negative
        integer, zero, or a positive integer as this object is less than, equal to, or
        greater than the argument object respectively. */
    public int compareTo(T o);
```

1 Warm-Up (3 points)

Assume `java.util` is imported.

```
double decker = 3.0;
double down = Double.POSITIVE_INFINITY; // number that represents infinity
double standard = -10.0;
PriorityQueue<Double> pq = new PriorityQueue<Double>();
Queue<Double> q = pq;
q.offer(decker);
q.offer(standard);
q.offer(down);
System.out.println(q.poll());
```

(a) What would the above code block print?: - 10.0

Now assume we define the following class:

```
public class Person implements Comparable<Person> {
  public String myName;

  public Person(String name) {
    myName = name;
  }

  @Override
  public int compareTo(Person p) {
    return myName.compareTo(p.myName);
  }

  @Override
  public int hashCode() {
    return myName.hashCode();
  }
}
```

Samantha - Sam

Mantha

```
TreeMap<Person, String> nicknames = new TreeMap<Person, String>();
Person s = new Person("Samantha");
nicknames.put(s, "Sam");
s.myName = "Mantha";
System.out.println(nicknames.get(s));
```

(b) What would the above code block print?: Sam

```
HashMap<Person, String> nicknames = new HashMap<Person, String>();
Person t = new Person("Tammy");
nicknames.put(t, "Tamantha");
for (Person p : nicknames.keySet()) { // iterates over keys in the map
  Person clone = new Person(p.myName);
  System.out.println(nicknames.get(clone));
}
```

Tammy - Tamantha

(c) What would the above code block print?: null

2 The Right ADT for the Job (7 points)

For each of the following scenarios, pick one or two ADTs and describe very *briefly* (no more than two sentences will be graded) how you'd use it to solve the problem. You should select from the following: `Map`, `Set`, `Stack`, `List`, `array`, `PriorityQueue`, and `Tree`.

You should pick the ADT (or ADTs) that optimize for average case asymptotic runtime as well as ease-of-use, taking into account the specific types of inputs listed in each problem. State the specific impelementation of the ADT you're using, as well.

(a) Given a text file that contains the name and country of birth of every Berkeley student, count the number of different countries that are represented.

Set -> HashSet

use HashSet to store the name of the country. Count the number of countries encountered in the Set

(b) You're designing a text editor, and you want to save the changes the user makes to their document, so they can undo them. The user should be able to undo several times in a row. They should also be able to redo and undo in series. How would you save the changes the user makes?

List -> ArrayList

store "changes" as a chain of ~~nodes~~ items of ArrayList. Move the index pointer forwards and backwards. if necessary

(c) Given a list of strings, partition the strings into groups based on which ones are anagrams of each other. e.g. given `["cat", "love", "act", "bat", "tab", "tac"]`, you should return `[["cat", "act", "tac"], ["love"], ["bat", "tab"]]`. Hint: assume you have a helper method that takes in a string and returns the same string but with its characters sorted.

~~HashS~~

~~Set → HashSet~~

Map → HashMap

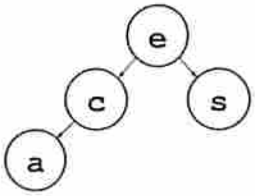
for map, key is the Sorted string and value is a list of strings, that have same character sequence.

for example, key "act" → { "cat", "act", "tac" }.

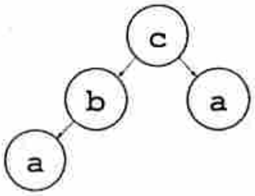
3 Forestry Class (4 points)

Consider the beautiful forest of trees below. Mark whether each one could be a subtree of a binary search tree, an AVL tree, a min or max heap, or a trie. A tree could be one of those, none, or multiple. By subtree, we mean there are no further nodes below, but there could be more nodes above. Assume the items are Strings, which are comparable by lexicographic order (abcdefghijklmnopqrstuvwxyz).

(a)



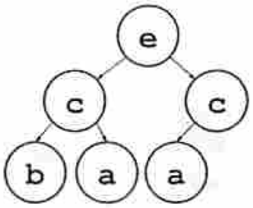
(b)



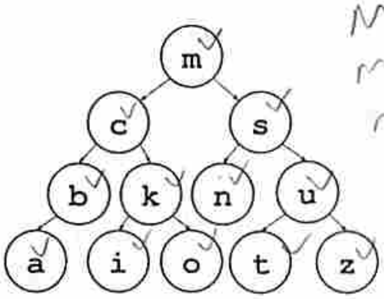
may heap

AVL.

(c)

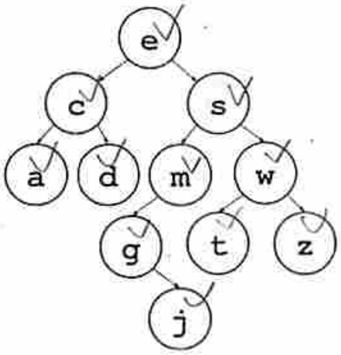


(d)



MCba
mcki
mcko
msn
msut
msuz.

(e)



	BST	AVL	Heap	Trie
(a)	✓	✓		✓
(b)		✓	✓	✓
(c)		✓	✓	
(d)	✓	✓		✓
(e)	✓			✓

eca
ecd
esmgj
eswt
eswz.

4 Hash Something! (4 points)

Remember your first project? For this problem, we want you to make the `Board` class hashable, so you could use it in a `HashMap`. The hash code method you write must satisfy the requirements to be a legal hash code. We'll also grade your code based on how well it hashes.

When you made the project, there was variation as to how you could have implemented your board class. Here, we'll make the decision for you. `Board` has a single instance variable, a `Piece[][]` named `pieces`. You can assume `Board` has an `equals` method that checks the pieces in each position are `equals`.

Piece[] Pieces.

As a recap, here are the relevant methods of the `Piece`.

```
/** Returns 0 if the piece is a fire piece, 1 if water */
public int side() {...}

/** Returns true if the piece has been crowned, false otherwise */
public boolean isKing() {...}
```

Also recall that `Piece` has two subclasses: `ShieldPiece` and `BombPiece`. Assume these classes have `equals` methods that check if the pieces are the same type, the same side, and the same kingship.

Your task is to add code to the classes so that you can properly hash `Board` objects. Please do not add extra instance variables. You are not allowed to use `instanceof` either.

Your code added to `Board` here:

```
public int hashcode() {
    int code = 0;
  boolean ~~int~~ negative = false;
    for (int i=0; i<pieces.length; i++){
        for( int j=0; j<pieces[i].length; i++){
            int temp=0;
            if (pieces[i][j] == null){
                temp += 7;
            } else {
                temp += pieces[i][j].side() + 1;
                temp += pieces[i][j].isKing()? 4 : 8;
                temp += Pieces[i][j].type() + 3;
                negative = !negative;
            }
            code += (i * pieces.length + j) * temp;
    }
    return negative? -code : code;
}
```

Your code added to `Piece` here:

```
public int type() {
    return 0;

}
```

Your code added to `ShieldPiece` here:

```
public int type() {
        return 1;
 }
```

Your code added to `BombPiece` here:

```
public int type() {

        return 2;

 }
```

5 I <3 Tree Processing (10 points)

In your quest for internet fame, you've decided to analyze comments on your favorite forum to see what makes a popular comment.

Your plan: For each comment thread, first observe all the posts there. For each post, count up the number of times each word in the post appears (i.e. create a histogram of the words in the post). Then see if you can determine any trends: are there certain words that commonly appear in heavily liked posts?

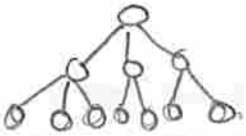
Specifically, here's the set up. You have a class `CommentThread`. A comment thread starts with a single post that contains text. The words of this post are counted up and stored in `myWordCounts`. This post may be liked by users of the forum, and the `CommentThread` stores these likes in `myLikes`. Further, other users on the forum may respond to the post with their own post – this becomes a new `CommentThread` object. These responses are stored in `myReplies`.

Here's the start of the class, with a constructor. The words in the post are passed in as an argument to the constructor.

```
import java.util.*;

public class CommentThread {
  private Map<String, Integer> myWordCounts;  Count of words appeared
  private int myLikes;  count of likes
  private List<CommentThread> myReplies;  ~~count of~~ all replies

  public CommentThread(List<String> comment) {
        // Store count of each word from comment in myWordCounts
    myReplies = new ArrayList<>();
    myLikes = 0;
  }
   // other methods here
}
```



You can imagine this class has other methods for adding new replies, getting liked, etc. You won't need to use these.

You'll add the following method:

```
public List<String> topTenWords() {...}
```

This should explore the *whole* comment thread (this post, its replies, its replies' replies, etc.) and return the ten most liked words (if there are less than ten existing words, just return all of them). Specifically, for each post, the number of times a word is considered liked is the number of times the word appears in the post times the number of likes on the post. Then add up this number for all posts. Here's an example. Suppose someone posts the following comment:

`I love love love wugs`   X

I × 10 + 100 × I

And it gets 10 likes.

Then someone replies to this comment with:

`I know right wuglets are so cute`

And it gets 100 likes.

Then the total number of likes associated with `I` is $1 * 10 + 1 * 100 = 110$. The total number of likes associated with `love` is $3*10+0*100=30$. In this case, `I` happens to be the most liked word (how narcissistic!). You need to return the top 10.

Your task is to write the method `topTenWords`. **You may also add any other code to the class that you want. Helper methods, etc.** If your code doesn't fully work, more partial credit will be given if your code is divided into smaller helper methods that *do* work.

The remainder of this page is intended as scratch paper. Nothing you write on this page will be graded. Put your complete answer over the following two pages.

```
public List<String> topTenWords ( ) {

        List <String> $ist = new ArrayList <String > ( );
          List <~~String~~ Integer> Count = new ArrayList <Integer> ( );


            public Map<String, Integer> wordScore() {
                    ~~for~~ Map<String, Integer> score = new Map<String, Integer>();
                       for (CommentThread t : myReplies) {
  }                            ~~for (String~~
                            <> replyScore = t. wordScore();
                             for (~~int i=0~~ String i : replyScore keySet()) {
            }
~~Public~~
public   addTo List(~~)~~ List <String> List <Integer> Count
                                 if (Score . get (i) != null) {
                }                     score (add (i, 0));
                                 }

                                 Score .add( i , score get (i) + t. scor
//helper method 1 it's really inefficient !
public HashMap<String,Integer> wordScore() {
  HashMap<String,Integer> score = new HashMap<String,Integer>();
    for (CommentThread t : myReplies) { // recursive
        HashMap<String,Integer> replyScore = t.wordScore();
        for (String i : replyScore.keySet()) { //iterate
            if(Score.get(i) == null) { // check null
                Score.add(i, 0); // initialize key with value 0
            }
            Score.add(i, Score.get(i) + replyScore.get(i));
            // increment the value, count
        } //finished adding reply score
    }

    for (String word:myWordCounts.keySet()) { //iterate
        if (Score.get(i) == null) { // check null again
            score.add(i,0);
        } // count * likes
        score.add(i, Score.get(i) + myWordCounts.get(i)*myLikes);
    }
    return Score;
}.
```

6. 6

```
//main method to use
public List<String> topTen Words () {
  ArrayList<String> words = new ArrayList<String>();
  ArrayList<Integer> count = new ArrayList <Integer>();
  HashMap<String, Integer> myScores = this.wordScore();
  for (~~myScores~~ String word: myScores. keySet ) {
      int ~~count~~ score = myScores. get(word);
      int index ~~=~~ -1;
      for ( int i=0; i< count.size() ; i++) {
          if ( count.get(i) ≮ score ) {
              index = i;
              break;
          }

      }

      if ( index != -1) {
          words. add ( ~~word);~~ index, word)
          ~~to~~ count add( index, score);
      }
      if ( count .size() >10 ) {
          words. remove( 10);
          count .remove (10);
      }
  }
  return words;
}
```

6 Big O Set (5 points)

We've had a lot of fun working with Big O this summer, haven't we? And we've had a lot of fun working with sets, too, right? So of course what's most beautiful is when we think about how the concepts overlap. In big O notation, O is a set, after all!

First, consider the following `Polynomial` class.

```java
public class Polynomial {
    private double[] myCoefficients;

    public Polynomial(double[] coefficients) {
        myCoefficients = coefficients;
    }

    public double[] coefficients() {
        return myCoefficients;
    }
}
```

A polynomial is represented by an array of its coefficients. For example, if I passed in `[3.4, 7.6, 0, 1.2]` into the constructor of a polynomial, I would get something that represents $f(x) = 3.4 + 7.6x + 1.2x^3$ You can assume the array that's passed into the constructor will not have any trailing zeroes.

Now consider the following `BigO` class:

```java
public class BigO {
    public BigO(Polynomial p) {...}
    private double size() {...}
    public boolean contains(Polynomial p) {...}
    public boolean isSubsetOf(BigO otherBigO) {...}
}
```

The big O class can be initialized by passing in a polynomial, just like the regular big O. For example, the expression

$O(N^2 + 2N + 4)$

is generated by the Java code

`new BigO(new Polynomial(4, 2, 1));`

Cool, right? Note that for simplicity our BigO class can only take in polynomial arguments; it can't take in other types of expressions, like logs or exponentials.

A big O is just a set, so like all sets in Java we've come across, it has a `contains` method and a `size` method. These methods report whether the big O set contains the argument polynomial passed in, and how many total polynomials the big O set contains, respectively. For fun, we also threw in an `isSubsetOf` method, which returns whether this big O set is a subset of the argument big O set (if A is a subset of B, it means that for every item A contains, B also contains).

On the next page, complete the Big O class. Be sure to write implementations for the constructor and three methods above. You may add instance variables if you want. Hint: The answer for `size` is not simply `return 1;`.

```
public class BigO {
    // Add any instance variables you want
        ~~Polynomial p;~~
        List<Polynomial> list; ~~= new~~


    public BigO(Polynomial p) {
        // Add as many lines as you want
        ~~this.p = p;~~
        list = new List<Polynomial>();
        for(int i=0; i< p coefficients().length; i++){
            double[] clist = new double[p.coefficients().length-i];
            for(int j=p.coefficients().length-1; j>=0; j--){
                clist[i] = p.coefficients()[j];
    }
    private double size() {
        // Please add only one line
        }   list.add(clist);
        return list.size();


    }

    public boolean contains(Polynomial p) {
        // Please add only one line

        return list.contains(p);

    }

    public boolean isSubsetOf(BigO otherBigO) {
        // Please add only one line
        return otherBigO.contains(p);


    }
}
```

7 Bookstore (11 points)

book $\xrightarrow{map}$ author. map
author → [books]

in

You're managing an electronic bookstore, because you love books. One function of your bookstore is that visitors can type in the title of a book, and you will show them the name of the author of the book. Another function is that a visitor can type in the name of an author, and you will show them all the books written by that author in your store. Visitors can also browse by recent releases, so you should be able to show them a list of all books in the store ordered from most recently added to least recently added. Assume there are no duplicate book titles or author names.

stack

Your bookstore receives new books from time to time, and you want to keep adding them to your collection. Sometimes a book goes out of stock, and you'd like to remove it from your collection.

You'd like to make this bookstore function with Java code. Specifically, the class `Bookstore` would have the following methods:

```java
/* Prints the name of the author who wrote the book title. */
public void findAuthor(String bookTitle) {...}

/* Prints the names of all books written by the author. */
public void findAllBooks(String authorName) {...}

/* Prints the names of all books in order of most recent to least. */
public void browseRecentBooks() {...}

/* Adds a new book to the store. */
public void addNewBook(String bookTitle, String authorName) {...}

/* Removes a book from the store. After this is called,
 * it should be as if this book was never added in the first place. */
public void removeBook(String bookTitle) {...}
```

Your task it to *design* a Java class to support this functionality, **without actually writing the qode.** Specifically:

(a) First, list out the names and (dynamic) types of all instance variables you would put in the `Bookstore` class. You may uses classes from `java.util`. You *may* invent private inner classes. If you do, list out the names and types of their instance variables, too.

(b) Next, list out the tighest and simplest big O bounds you can give for the average-case number of operations needed to complete each of the different methods. Write your bounds in terms of B, the total number of books in your bookstore, and A, the total number of authors. For `findAllBooks`, you may also write your answer in terms of b, the number of books written by the argument author. **Your goal is to make all of the operations have as low average-case asymptotic runtimes as possible.**

(c) Finally, write a brief, high-level description of how each method would work. You do not need to write code, just descriptions. Your descriptions need to be precise enough that the asymptotic runtime of each method is unambiguous.

This page is intended as scratch paper. Nothing you put on this page will be graded. Please put your complete answer on the following two pages.

first we, need two maps (maybe HashMap), (Book class is described later)

- HashMap<~~Book~~ String, String> | ~~Book~~ Title → Author
- HashMap<String, ArrayList<Book>> | ~~Title~~ ~~Author~~ Author → Books.

We also need a stack to store ALL the books.

- Stack<Book> | A stack of books.

There is an class (inner class) Book.

- class Book
  - String title | title of the book
  - String Author | Author of the book

for findAuthor: O(1)
findAllBooks: O(b) | print one by one
~~public~~ browseRecentBooks(): O(B) | take all out, add all back
addNewBook. O(1) | modify two hashmaps, add to Stack
removeBook: O(B+b) | take from stack, remove from list

○ for findAuthor:

because there is hashmap store "title" → "author", it's just a matter of take the value from hash map

○ find all books:

there is hashmap store "author" → "all books by the author". take the list out and print one by one...

○ browse RecentBooks:

the books are stored in Stacks. take elements out, print them (show to the Customer) and put back it's a stack of books that Customer can browse.

○ add new book:

adding a book takes a little bit of indexing. add it to the Stack, add it to the title → author map (index) and Author → Books as an element of that book list.

○ remove a book

removing takes time you search through the Stack of books until finding one, remove and put others back also remove the entry at author → all books map (from the books list) and title → author index.