Test 3: CompSci 6

75 Minute Exam

April 6, 2009

Name (print): _____

Honor Acknowledgment (signature): _____

DO NOT SPEND MORE THAN 15 MINUTES ON ANY OF THE QUESTIONS! If you do not see the solution to a problem right away, move on to another problem and come back to it later.

Before starting, make sure your test contains 12 pages. The last page is blank and can be used as scratch paper, but **all pages must be turned in**.

None of the Java programs or program segments should have syntax errors unless stated. If you think there is a syntax error or you do not understand what a question is asking, then please ask. Import statements may not always be shown.

Do **not** discuss this test **with anyone** until the test is handed back.

	value	grade
Problem 1	14 points	
Problem 2	15 points	
Problem 3	31 points	
TOTAL:	60 points	

PROBLEM 1 : (*How to choose:* (14 pts))

For each of the following problems, state the type of collection that best solves it and justify your decision. If you are using a map, you must clearly state what are the keys and values.

Part A: (6 points)

Consider the problem of building a web browser. How would you represent each of the following collections:

- a history of the URLs you have visited

- the URLs you have marked as your favorite

- the search results returned by the Google search engine

Part B: (8 points)

For each of the following problems, in addition to choosing the best type of collection as above, give psuedocode to solve the problem using that collection (do *not* write Java code).

Your answers to these questions should *not* change your answers above, instead, you should focus on what might be needed to solve this described problem.

- a "speed dial" for your browser that represents the ten most visited URLs

- a repository that allows people to view URLs that are categorized according to a collection of keywords known as tags and ranked according to the average value of all people that have ranked this URL

PROBLEM 2 : (*Out of order:* (15 points))

Part A: (7 points)

Write the method `indexOutofOrder` that is given a list of comparable values that are supposed to be sorted in increasing order and an index to start with and returns the index of the first value that is out of order. If none of the values are out of order, it returns -1.

For example, if the list `b` is shown below. Then the call `indexOutOfOrder(b, 0)` would return `3`, since `7`, which is in position `3` is the first value out of increasing order. While the call `indexOutOfOrder(b, 3)` would return `5`, since `11`, which is in position `5` is the first value out of increasing order starting from index `3`.

3	4	9	7	12	11	18	10

You will not receive full credit if you cast the values in the given list to any type more specific than `Comparable`. In other words, you may *not* assume that the values are numbers.

Complete the method below.

```
public int indexOutOfOrder (List<Comparable> values)
{









}
```

Part B: (5 points)

Write the method `sortByRemoval` that is given a list of values and "sorts" them by removing the ones that are out of order. Its algorithm is: find the first value out of sorted order and remove it; then, starting from that index, find the next value out of order and remove it; and so on until it reaches the end of the list. Thus it never actually verifies the entire list is in sorted order, assuming that removing the ones not in order will leave the entire list sorted.

Given the list `b` shown in the previous part, it would remove the values `7`, `11`, and `10`, because those three values are at the indices returned by the three successive calls: `indexOutOfOrder(b, 0)`, `indexOutOfOrder(b, 3)`, `indexOutOfOrder(b, 4)` (note the last index is different from the one given in the previous example because the values in the list would have shifted right when the value at the index `3` was removed).

You will not receive full credit for this part unless you call the method `indexOutOfOrder` that you wrote in Part A at least once and use its result in determining the result of this method. Assume `indexOutOfOrder` works as specified regardless of what you wrote in Part A.

Complete the method below.

```
public void sortByRemoval (List<Comparable> values)
{









}
```

Part C: (3 points)

Does the algorithm described in Part B guarantee to leave the given list in sorted order? If yes, justify why you believe it is *guaranteed* to work. If no, explain why not and provide an example list for which it fails.

PROBLEM 3 : (*Happy Birthday:* (31 points))

Part A: (9 points)

Write a several methods for a class that represents a date as three integer values, one for the day, one for the month, and one for the year:

1. a constructor that takes a string representing the month, day, and year, each value separated by a forward slash, "/", e.g., "7/27/2004"
2. a method that compares the current date to another by first comparing their year, then, if those were equal, their month, and finally, if they were equal, their day; returning 1 if the current date is greater (comes after) the given date, 0 if they represent the same day, or -1 if it is less (comes before) the given date
3. a method that determines if the current date is the same as another by verifying that both have the same month, day, and year

If you need to, you may assume that this class also has the methods `getDay()`, `getMonth()`, and `getYear()` that return their respective int values.

Complete the class methods below.

```
public class Date implements Comparable<Date>
{
    private int myDay;
    private int myMonth;
    private int myYear;

    public Date (String formattedInput)
    {









    }
```

Problem continued on next page ...

```
    /**
     * Returns
     *    positive value if this date is chronologically later than the given date,
     *    0 if this date represents the same day as the given date, and
     *    negative value if this date is chronologically earlier than the given date.
     */
    public int compareTo (Date other)
    {









    }


    /**
     * Returns
     *    true if this date represents the same day as the given date,
     *    false otherwise
     */
    public boolean equals (Object o)
    {
        if (o instanceof Date)
        {
            Date other = (Date)o;









        }
        return false;
    }
}
```

Part B: (10 points)

Given a file of dates and todo items, one date and todo item per line, write a method `readTodos` that, given a `Scanner` for reading that file, constructs and returns a `Map<Date, Set<String>>` of all the todo items occurring on each date.

Each line of the file has a date in the format of three integer values representing the month, day, year, respectively, each separated by a forward slash, "/" followed by text describing a single todo item for that day. The dates given are in no particular order.

For example, the following file representes five items.

```
4/5/2009 Study for big CompSci exam
4/6/2009 Take exam
4/12/2009 Study
4/10/2009 Party
4/12/2009 Eat brunch
4/7/2009 Eat lunch
```

Complete the method below.

```
public Map<Date, Set<String>> readTodos (Scanner input)
{









}
```

Part C: (10 points)

Write a method `getTodos` that, given a `Map<Date, Set<String>>` and a single `Date`, returns a `List<String>` of all the todo items that are due *before* the given date sorted alphabetically (rather than based on what date they occur on). If there are no items before the target date, then an empty list should be returned.

For example, given the date 4/9/2009 and the items from the previous part, your method should return the following:

```
Eat Lunch
Study for big CompSci exam
Take exam
```

Note, you **cannot** assume that the given `Map` is a `TreeMap` in completing the method below.

```
public List<String> getTodos (Map<Date, Set<String>> dates, Date target)
{









}
```

Part D: (2 points)

Describe how your code would change if you knew the given `Map` was a `TreeMap`.

Throughout this test, assume that the following classes and public methods are available. These classes are taken directly from the material used in class. There should be no methods you have never seen before here.

String

```
class String
{
  // Returns string's length
  int length ()

  // Returns a substring of this string that
  // begins at the given beginIndex and extends
  // to the character at index endIndex - 1
  String substring (int beginIndex, int endIndex)

  // Returns a substring of this string that
  // begins at the given beginIndex and extends
  // to the end of the string
  String substring (int beginIndex)

  // Returns position of the first occurrence
  // of str, -1 if not found
  int indexOf (String str)

  // Returns position of the first occurrence
  // of str after index start, -1 if not found
  int indexOf (String str, int start)

  // returns -1 if less than str, 0 if equal,
  // +1 if greater
  int compareTo (String str)

  // returns true if this string starts with
  // the characters in str
  boolean startsWith (String str)

  // returns true if this string ends with
  // the characters in str
  boolean endsWith (String str)
}
```

Object

Methods common to all Java objects

```
class Object
{
  // Returns true if this object's values are the
  // same as the given object's values
  boolean equals (Object other)

  // Returns string representation of this object
  String toString ()
}
```

Collection

Methods common to all collections

```
class Collection<ItemType>
{
  // Returns the number of elements in this collection
  int size ()

  // Returns true if there are no elements in this collection
  boolean isEmpty ()

  // Removes all elements from this collection
  void clear ()

  // Returns true if the given element is in this collection
  // (uses equals to determine same-ness)
  boolean contains (ItemType element)

  // Adds given element to this collection
  boolean add (ItemType o)

  // Removes specified element from this collection
  boolean remove (ItemType o)
}
```

List

Additional methods for a list

```
class ArrayList<ItemType>
{
  // Constructs an empty list
  ArrayList ()

  // Constructs a list containing the given elements
  ArrayList (Collection<ItemType> elements)

  // Returns element at position index in this list
  ItemType get (int index)

  // Returns the index of the given element in this list
  int indexOf (ItemType element)
}
```

Set

Additional methods for a set

```
class TreeSet<ItemType>
{
  // Constructs an empty set
  TreeSet ()

  // Constructs a set containing the given elements
  TreeSet (Collection<ItemType> elements)
}
```

Map

Additional methods for a map

```
class TreeMap<KeyType, ValueType> {
  // Constructs an empty map
  TreeMap ()

  // Returns value associated with given key,
  // null if not in map
  ValueType get (KeyType key)

  // Returns a set of this map's keys
  Set<KeyType> keySet ()

  // Returns a collection of this map's values
  Collection<ValueType> values ()

  // Adds a mapping from key to value to this map
  ValueType put (KeyType key, ValueType value)
}
```

Scanner

```
class Scanner {
  // Constructs Scanner that reads from given string
  Scanner (String str)

  // Check if more items are available
  boolean hasNext ()

  // Get next delimited item as a string
  String next ()

  // Get next line as a string
  String nextLine ()

  // Get next delimited item as an integer value
  int nextInt ()

  // Get next delimited item as a double value
  double nextDouble ()
}
```

Comparable

```
// defines a natural ordering for the object ItemType;
// this interface should be implemented by a class
// to describe how it is compared to another of the
// same type
interface Comparable<ItemType> {
  // Returns negative if this object is less than other,
  // 0 if equal, positive if greater
  int compareTo (ItemType other)
}
```

Comparator

```
// defines an ordering on objects of ItemType;
// this interface should be implemented by a class
// that is passed to a collection to describe how
// to compare to objects of the same type
interface Comparator<ItemType> {
  // Returns negative if lhs is less than rhs,
  // 0 if equal, positive if greater
  int compare (ItemType lhs, ItemType rhs)
}
```

Collections

```
class Collections {
  // Returns largest value in the given collection
  // according to their natural ordering
  ItemType max (Collection<ItemType> elements)

  // Returns largest value in the given collection
  // according to the given ordering
  ItemType max (Collection<ItemType> elements,
                Comparator<ItemType> comp)

  // Returns smalles value in the given collection
  // according to their natural ordering
  ItemType min (Collection<ItemType> elements)

  // Returns smalles value in the given collection
  // according to the given ordering
  ItemType min (Collection<ItemType> elements,
                Comparator<ItemType> comp)

  // Sorts the elements in the given collection
  // by their natural ordering
  void sort (Collection<ItemType> elements)

  // Sorts the elements in the given collection
  // by the given ordering
  void sort (Collection<ItemType> elements,
             Comparator<ItemType> comp)
}
```