CS20B: Homework 3 (100 points)

prof. dr. Irma Ravkic

This homework covers Recursion (Chapter 3), Queues (Chapter 4) and Collections (Chapter 5).

1 Questions (48 points)

In your answer document please include the question text as well so I can easily match questions to your answers.

1. (8 points) Given the following method:

```
int exer(int num)
{
    if (num == 0)
        return 0;
    else
        return num + exer(num + 1);
}
```

   (a) Is there a constraint on the value that can be passed as an argument for this method to pass the smaller-caller test?

   (b) Is exer(7) a valid call? If so, what is returned from the method?

   (c) Is exer(0) a valid call? If so, what is returned from the method?

   (d) Is exer(-5) a valid call? If so, what is returned from the method?

2. (16 points) You must assign the grades for a programming class. Right now the class is studying recursion, and students have been given this assignment: Write a recursive method *sumValues* that is passed an index into an array values of int (this array is accessible from within the *sumValues* method) and returns the sum of the values in the array between the location indicated by index and the end of the array. The argument will be between 0 and values.length inclusive. For example, given this configuration:

   then *sumValues(4)* returns 12 and *sumValues(0)* returns 34 (the sum of all the values in the array). You have received quite a variety of solutions. Grade the methods below. If the solution is incorrect, explain what the code actually does in lieu of returning the correct result. You can assume the array is "full." You can assume a "friendly" user-the method is invoked with an argument between 0 and values.length.

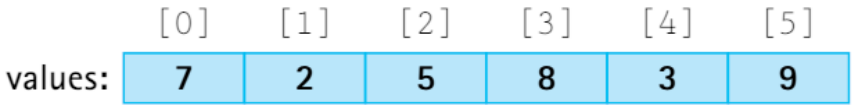

Figure 1: Example

(a)
```
1  int sumValues(int index)
2
3  {
4      if (index == values.length)
5          return 0;
6      else
7          return index + sumValues(index + 1);
8  }
```

(b)
```
1  int sumValues(int index)
2  {
3      int sum = 0;
4      for (int i = index; i < values.length; i++)
5          sum = sum + values[i];
6      return sum;
7  }
```

(c)
```
1  int sumValues(int index)
2  {
3      if (index == values.length)
4          return 0;
5      else
6          return (1 + sumValues(index + 1);
7  }
```

(d)
```
1  int sumValues(int index)
2  {
3      return values[index] + sumValues(index + 1);
4  }
```

(e)
```
1  int sumValues(int index)
2  {
3  if (index > values.length)
4          return 0;
5  else
6      return values[index] + sumValues(index + 1);
7  }
```

Table 5.2 Comparison of Collection Implementations

Storage Structure	`ArrayCollection` Unsorted array	`SortedArrayCollection` Sorted array	`LinkedCollection` Unsorted linked list
Space	Bounded by original capacity	Unbounded—invokes `enlarge` method as needed	Unbounded—grows and shrinks
Class constructor	$O(N)$	$O(N)$	$O(1)$
`size` `isEmpty` `isFull`	$O(1)$	$O(1)$	$O(1)$
`contains`	$O(N)$	$O(\log_2 N)$	$O(N)$
`get`	$O(N)$	$O(\log_2 N)$	$O(N)$
`add`	$O(1)$	$O(N)$	$O(1)$
`remove`	$O(N)$	$O(N)$	$O(N)$

Figure 2: Comparison of Collection Implementations.

(f)

```
1  int sumValues(int index)
2  {
3      if (index >= values.length)
4         return 0;
5      else
6        return values[index]+sumValues(index + 1);
7  }
```

(g)

```
1  int sumValues(int index)
2
3  {
4      if (index < 0)
5        return 0;
6      else
7        return values[index] + sumValues(index - 1);
8  }
```

3. (6 points) Read Chapter 5, view all the code examples there: even those we didn't see in the class (the complete code for classes from this Chapter is provided for you, you can also find partial code in the book). Look at the table provided in Chapter 5 (see Figure 2 in this file). Explain the following:

   (a) (2 points) Why does *SortedArrayCollection* exhibit logarithmic time for *contains* and *get* methods?

   (b) (2 points) Why do *ArrayCollection* and *LinkedCollection* exhibit $O(1)$ time for *add* (adding an element method), while *SortedArrayCollection* exhibits a worse $O(N)$ time for the same method?

   (c) (2 points) How is $O(1)$ accomplished for *size* method for all of these collections in the table?

4. (4 points) a) Which method do we need to implement to check if the contents of our objects are the same? b) Write its signature. c) Is it inherited from some class? d) If yes, which one?

5. (8 points) Based on the equals method for Circle objects defined in Section 5.4, "Comparing Objects Revisited," what is the output of the following code sequence and **why**! (you need to provide 8 answers for each printout, and a very short explanation of why it is so)?

```
Circle c1 = new Circle(5);

Circle c2 = new Circle(5);

Circle c3 = new Circle(15);

Circle c4 = null;

System.out.println(c1 == c1);

System.out.println(c1 == c2);

System.out.println(c1 == c3);

System.out.println(c1 == c4);

System.out.println(c1.equals(c1));

System.out.println(c1.equals(c2));

System.out.println(c1.equals(c3));

System.out.println(c1.equals(c4));
```

6. (6 points) Consider the following interfaces:

```
public interface Event{
   public void RSVP();
}

public interface PublicEvent extends Event{
   public void doSecurityCheck();
}

public interface PrivateEvent extends Event{
   public void checkIdentification();
}

public interface PartneredEvent extends Event{
   public void findPartner();
}
```

Which method(s) should implement a class

(a) (3 point) called *Hockey* that implements *PublicEvent*?

(b) (3 point) called *Prom* that implements *PrivateEvent* and *PartneredEvent*.

2 Programming exercises (52 points)

For the programming exercises, please follow the provided instructions in the instructions.pdf. Each of the exercises below should be a separate zip of a Java project.

1. Project 1: (10 points) Attached to this homework is a .java file called *LinkedListReverse.java*. Part of it is implemented for you and there is a main method already written that will test your code. A recursive printing of a linked list is also provided for you in method called *recRevPrintList*. You need to implement *iterRevPrintList* (method signature is given to you) which **iteratively**, not recursively, does the same - prints a linked list in reverse. Make sure you include and import all necessary support classes for this code to work.

2. (15 points) Add the following methods to the *LinkedQueue* class (from ch04.zip), and create a test driver for each to show that they work correctly. In order to practice your linked list coding skills, code each of these methods by accessing the internal variables of the LinkedQueue, not by calling the previously defined public methods of the class.

   (a) (5 points) *String toString()* creates and returns a string that correctly represents the current queue. Such a method could prove useful for testing and debugging the class and for testing and debugging applications that use the class. Assume each queued element already provides its own reasonable toString method.

   (b) (10 points) *void remove(int N)* removes the front N elements from the queue; throws *QueueUnderflowException* if less than N elements are in the queue.

3. (20 points) Add the following methods to the *ArrayCollection* class, and create a test driver for each to show that they work correctly. **Note** that you can find the implementation of the classes mentioned here in *ch05.zip* or check the book for the implementation details. Some classes require imports from other packages. All the packages you need and classes can be found in the code I shared with you. In order to practice your array coding skills, code each of these methods by accessing the internal variables of the ArrayCollection, not by calling the previously defined public methods of the class. Make sure you have a driver class that will put your methods to use (and also serve to check whether your methods are correct).

   (a) (2 points) *String toString()* creates and returns a string that correctly represents the current collection. Such a method could prove useful for testing and debugging the class and for testing and debugging applications that use the class. Assume each stored element already provides its own reasonable *toString()* method.

(b) (4 points) *int count(T target)* returns a count of the number of elements **e** in the collection such that **e.equals(target)** is true.

(c) (5 points) *void removeAll(T target)* removes all elements **e** from the collection such that **e.equals(target)** is true.

(d) (9 points) ArrayCollection<T> combine(ArrayCollection<T> other) creates and returns a new ArrayCollection object that is a combination of this object and the argument object. Choose any way to combine these collections and make sure you include comments to explain what the combination is. **(Note: add the signature of this method to *CollectionInterface***

4. (7 points) Create a *FacebookUser* class that includes *numFriends* and *username* attributes of type int and String, respectively, both set by a constructor. The attribute *numFriends* represents the number of friends a user with *username* has. Implement an *equals* method that such that two FacebookUsers are equal if they have the same username.

   In the homework package I gave you a FacebookUser.java code. Use this code to test whether your *equals* works well.

   Next are two bonus exercises for the adventurous ones. These can make up for the lost points on some other exercises, or in general can make me decide in favor of the higher grade when borderlining, but above all, it's a great exercise for you. To receive credit you have to write a driver class I can run and see your solution work, and around 90% of your code should work.

5. (Bonus Exercise) *This one is not as hard as the next Bonus Exercise.*

   An *Advanced Set* includes all the operations of a *Basic Set* plus operations for the union, intersection, and difference of sets.

   a. Define an Advanced Set interface.

   b. Implement the Advanced Set using an unsorted array; include a test driver that demonstrates your implementation works correctly.

   Hint: you can find the implementation of Basic Set in ch05.zip code, and also you can check the book for more details.

6. (Bonus exercise) *This is a rather challenging but very interesting exercise from Chapter 4.*

   Design and code the Queue ADT operations using a circular linked implementation. With the circular linked list we don't need to maintain two pointers for front and rear. We can maintain a pointer to the last inserted node and front can always be obtained as next of last. The book gives you some hints on how to do this in the end of Section 4.5.