Family Name: . . . . **Other Names:** . . . .

ID Number: . . . .

COMP103 Test

19 August, 2010

| ************ WITH SOLUTIONS ************

Instructions

- Time: **45 minutes**.
- Answer **all** the questions.
- There are 45 marks in total.
- Write your answers in the boxes in this test paper and hand in all sheets.
- Every box with a heavy outline requires an answer.
- If you do not understand a question, ask for clarification.
- There is java documentation at the end of the exam paper that you may find useful.

		Marks		
1.	Various topics	[10]	1	
2.	Using Collections	[15]	2	
3.	Implementing Collections	[14]	3	
4.	Recursion and Sorting	[6]	4	
			Total:	

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 1. Various topics [10 marks]

(a) [2 marks] You can read the contents of a file with a Scanner and push the lines onto a Stack. When you pop the items from the Stack and print them, they appear in the reverse order. Why does this happen?

| The Stack is FILO, so the first item will be popped off the last.

(b) [3 marks] Consider SortedArraySet. Explain why in this implementation the asymptotic ("big O") cost for contains() is $\mathcal{O}(\log n)$ but is $\mathcal{O}(n)$ for remove().

| contains can use binary search (O(log n)), since it's a sorted array. But remove requires shifting a large portion of the array down (to keep it sorted), which is O(n).

(c) [3 marks] Explain the differences between Set, Map and Bag.

| Set: no duplicates
Bag: allows duplicates
Map: has key-value pairs (the keys are unique)

(d) [2 marks] In Java, you can have two methods with the same name. What conditions must be satisfied for this to happen, and what is the concept called?

| The number, or type, of the parameters (or the return type) must differ. This is called "overloading".

Question 2. Using Collections [15 marks]

The for each loop is a shorthand for an iterator. Here is an example of a for each loop:

```
for (Fish f : basket) {
    f.wriggle();
}
```

(a) [4 marks] In the box below, expand the above for each loop using an iterator explicitly.

```
Iterator itr = basket.iterator();
while ( itr.hasNext() )
    itr.next().wriggle();
```

.

(b) [2 marks] What type does basket have to be for the for each loop to work?

Iterable (1 mark only for "Collection"! and half of THAT (recursively :)) for something explicit like "List" etc.)

(c) [2 marks] In the box below, fix the error(s) in the following declaration:

```
List <String> myList = new List <String> ;
```

corrected version:

```
List <String> myList = new ArrayList <String> ();
```

.

(Question 2 continued on next page)

(Question 2 continued)

Consider code that stores the lines of text from a file using a Stack, and then reverses that Stack:

```
Stack <Line> mystack = new Stack <Line> ();

// the stack then has various Lines added to it by the scanner.
:
:

mystack = reverseStack(mystack);
```

(d) [7 marks] In the box below, complete the reverseStack method, that takes a Stack of Line objects as a parameter, and returns a Stack that has the same lines but in the *reversed order*.
You may assume the Line class exists.

```
public ...


 public Stack <Line> reverseStack (Stack <Line> mys) {

   // make a second stack
   Stack <Line> st = new Stack <Line> ();

   // pop off items from the original stack into the new one
   while (!mys.isEmpty()) {
     st.push( mys.pop() );
   }

   return st;

 }









}
```

Question 3. Implementing Collections [14 marks]

A Priority Queue is a special queue that is **not** FIFO (first-in-first-out). Instead, items are added in the queue based on their priority, and the item returned by poll or peek is always the one with highest priority.

A common internal data structure to implement different Collections is an Array. For example, you have used an Array to implement an ArrayList, or an ArrayQueue.

Now consider using an Array to implement a ArrayPriorityQueue, as shown below:

```
public ArrayPriorityQueue <E> extends AbstractQueue <E> {

  private static  int  INITIALCAPACITY = 10;
  private  int  count = 0;
  private E[]  data;

  public ArrayPriorityQueue(){ data = (E[])  new Object[INITIALCAPACITY];}
  public boolean isEmpty(){return count == 0;}
  public int  size  ()  {  return count;}

  /** Ensure data  array  has   sufficient   number of elements
   * to  add  a  new element */
  private void ensureCapacity () {...}      //assume done

  /** Find  and  returns  the  index  (between 0 and count)  of  where an element  is  in  the  dataarray ,
   * based  on  the  priority    (or  where  it  ought  to  be .)  */
  private  int  findIndex(Object itm ){...}  //assume done
  :
  public E poll(){}                          //not  done  yet .
  public boolean offer(E item){}             //  not  done  yet .

}
```

(Question 3 continued on next page)

(Question 3 continued)

(a) [8 marks] Complete the offer method below, for the ArrayPriorityQueue implementation. You may assume the findIndex method is available, which returns the index of where an item should be in the ArrayPriorityQueue.

Note: The priority is automatically handled by the findIndex method. Also assume ensureCapacity is available.

```
public boolean offer (E item) {

    if (item==null)
      return false;
    ensureCapacity();
    int index = findIndex(item);
    for( int i=count; i>index; i−−)
      data[i] = data[i−1];
    data[index] = item;
    count++;
    return true;

}
```

(b) [6 marks] Complete the poll method for ArrayPriorityQueue.

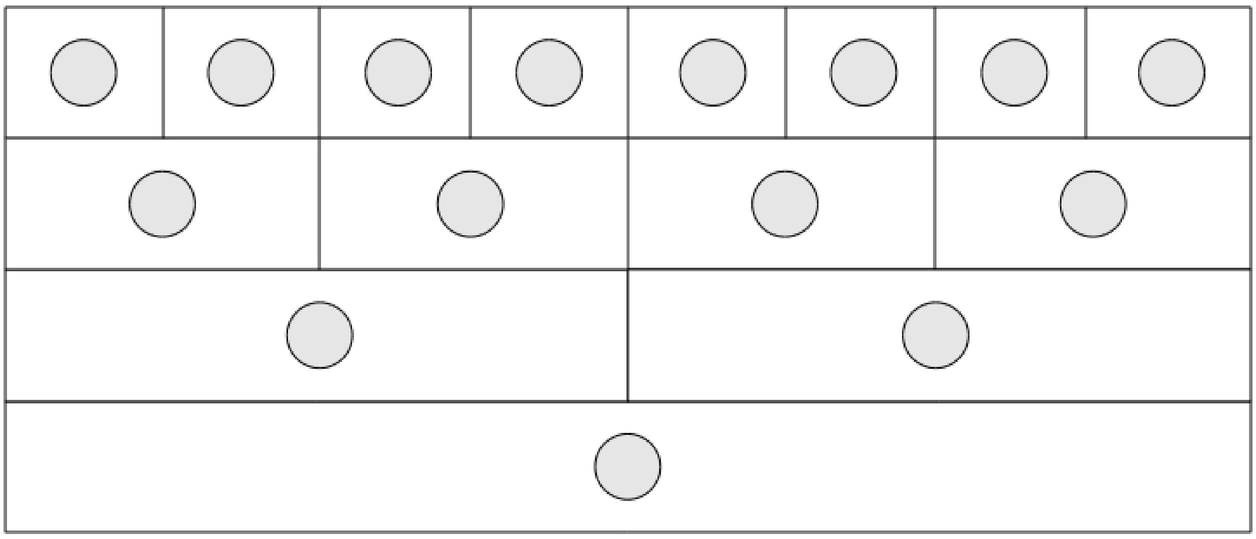
```
public E poll(){

    if (isEmpty())
      return null;

    E item = data[count−1];
    data[count−1] = null;
    count−−;
    return item;

}
```

Question 4. Recursion and Sorting [6 marks]

(a) [2 marks] A "brick-wall" consists of a brick with two half size brick-walls on top. The following code draws such a wall, using recursion:

```
public void brickWall ( int x, int y, int wd){
    drawBrick(x, y, wd);
    if ( wd > 10 ) {
        int w = wd/2;                    // width of next smaller brick
        brickWall(x,    y−10, w);        // left half
        brickWall(x+w,  y−10, w);        // right half
    }
}
```

In the brick-wall image below, write a digit inside the circle on each brick, showing the order in which they're drawn by the above method.



(Question 4 continued on next page)

(Question 4 continued)

(b) [2 marks] Consider the array shown in the first row below. In the next two rows, show the array after the first two iterations of the outer loop of SelectionSort.

S	C	N	P	Q	A	M	R	B

(c) [2 marks] The cost of QuickSort is $\mathcal{O}(n \log n)$ provided a sensible choice of pivot is made. But if QuickSort is implemented by simply using the *first* point in the array as the pivot, its cost increases from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n^2)$, in one particular scenario. What is that scenario?

A list that is already almost sorted.

****

Student ID: ....

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Appendix (may be removed)

Brief (and simplified) specifications of some relevant interfaces and classes.

```
public interface Iterator <E>
   public boolean hasNext();
   public E next();
   public void remove();

public interface Iterable <E>                  // Can use in the "for each" loop
   public Iterator <E> iterator();

public interface Comparable<E>                 // Can compare this to another E
   public int compareTo(E o);

public interface Comparator<E>                 // Can use this to compare two E's
   public int compare(E o1, E o2);
public interface Collection<E>
   public boolean isEmpty();
   public int size ();
   public boolean add();
   public Iterator <E> iterator();

public interface List<E> extends Collection<E>
   // Implementations: ArrayList
   public E get(int index);
   public void set(int index, E element);
   public void add(E element);
   public void add(int index, E element);
   public void remove(int index);
   public void remove(Object element);

public interface Set extends Collection<E>
   // Implementations: ArraySet, SortedArraySet, HashSet
   public boolean contains(Object element);
   public boolean add(E element);
   public boolean remove(Object element);

public interface Queue<E> extends Collection<E>
   // Implementations: ArrayQueue, LinkedList
   public E peek ();                        // returns null if queue is empty
   public E poll  ();                       // returns null if queue is empty
   public boolean offer  (E element);

public class Stack<E> implements Collection<E>
   public E peek ();                        // returns null if stack is empty
   public E pop ();                         // returns null if stack is empty
   public E push (E element);

public interface Map<K, V>
   // Implementations: HashMap, TreeMap, ArrayMap
   public V get(K key);                     // returns null if no such key
   public void put(K key, V value);
   public void remove(K key);
   public Set<Map.Entry<K, V>> entrySet();
```