Name: NetID:

CS2110 Final Exam SOLUTION

Sunday, 6 December 2017, 9:00–11:30

	1	2	3	4	5	6	Total
Question	Short Answer	Sorting	Object Oriented	Data Structures	Graphs	Concurrency	
Max	27	10	22	21	10	10	100
Score							
Grader							

1. Short Questions (27 points)

(a) Hashing (4 points)

Consider hashing with quadratic probing using an array $b[0..4]$ of size 5 to maintain a set of integers. Do not be concerned with the load factor. The hash function to be used is just the integer itself: $hashCode(i) = i$. Draw array b after these values have been added: 8, 13, 3, 7.

b		7	3	8	13

(b) Exceptions (6 points)

Consider the code below. On the right are three questions concerning 3 calls on method `m`. Below each, write what that call prints to the console.

```
public static void m(int x){
    try {
        m2(x);
        System.out.println(1);
    } catch (ArithmeticException e) {
        System.out.println(2);
    } catch (Exception e) {
        System.out.println(3);
    }
}

public static void m2(int x) throws IOException {
    System.out.println(4);
    if (x==1)
        throw new IOException();
    if (x==0)
        throw new ArithmeticException();
    System.out.println(5);
}
```

(i) 2 points Write what is printed to the console by `m(0)`.

4
2

(ii) 2 points Write what is printed to the console by `m(1)`.

4
3

(iii) 2 points Write what is printed to the console by `m(2)`.

4
5
1

(c) New-expression (3 points)

Write down the steps used to evaluate the new-expression `new D(5, 3)`.

1. Create (draw) an object of class `D`.
2. Execute the constructor call `D(5, 3)`.
3. Use as value of the new-expression the name of (or pointer to) the new object.

(d) Complexity (2 points)

What is the tightest worst-case time complexity (in terms of the length n of parameter s) for the following method for compressing strings? Circle one: $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, or $O(2^n)$. $O(n^2)$

```
/** Return a string in which each sub-sequence of equal characters in s is replaced
  * by that character followed by the number of times it occurs ---except that
  * for a sequence of length 1, the number 1 is omitted.
  * Example: For s = "aaaabbbadd" return "a4b3ad2".
  * Precondition: No sequence of equal chars in s is longer than 9 chars. */
public static String compress(String s) {
        if (s.length() == 0) return s;
        String compressed= "";
        int n= 1;  int i;
        // part of invariant: s[i-n..i-1] are equal and different from s[i-n-1]
        for (i= 1; i <= s.length(); i++) {
            if (i < s.length() && s.charAt(i-1) == s.charAt(i)) n= n+1;
            else {
```

```
            compressed= compressed + s.charAt(i-1) + (n == 1 ? "" : n);
            n= 1;
        }
    }
    return compressed;
}
```

(e) Recursion (4 points)

Write a recursive method to decompress strings compressed with the function of 2(d). Hint: you may use the three static methods given below. The use of function `repeat` eliminates the need for any loop.

`boolean Character.isDigit(char c)`
`char Character.getNumericValue(char c)`
`String repeat(char c, int n).` –return a string of n copies of c

```
/** Return s, decompressed. Precondition: s was produced using method compress. */
  * E.g. for s = "a4b3ad2" return  "aaaabbbadd". */
public static String decompress(String s) {
    if (s.length() <= 1) return s;
    char c= s.charAt(1);
    if (!Character.isDigit(c)) return s.charAt(0) + decompress(s.substring(1));
    return repeat(s.charAt(0), Character.getNumericValue(c)) +
              decompress(s.substring(2));

}
```

(f) Testing (2 points)

State two facets of structural testing (also known as white-box testing).

From the JavaHyperText entry for "testing":

1. Test each statement of a unit
2. Test each branch of a unit
3. Test each expression thoroughly
4. Test extreme or corner cases.

(g) Generics (6 points)

Assume you have a **non-empty** list `ls` of type `List<RuntimeException>`. For each of the following, indicate whether the statement (1) would result in a compile-time error, (2) might result in a runtime error, or (3) neither.

(i) 1 point `ls= new ArrayList<ArithmeticException>();` compile-time error

(ii) 1 point `ls= new ArrayList<Exception>();`compile-time error

(iii) 1 point `ls.add(new Exception());` compile-time error

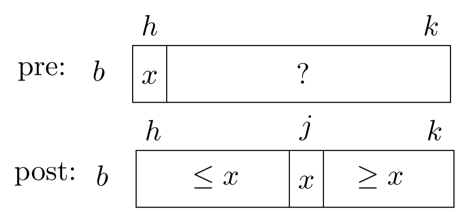
(iv) 1 point `ls.add(new ArithmeticException());` neither

(v) 1 point `ArithmeticException ae= (ArithmeticException) ls.get(0);`possible runtime error

(vi) 1 point `Exception e= (Exception) ld.get(0);`neither

2. Sorting (10 points)

Assume we have written the partition algorithm of quicksort as this function:




```
/** Let x be the value in b[h], as shown in
  * the precondition to the right.
  * Swap the values of b[h..k] so that the post-
  * condition to the right is true and return j. */
public static int partition(int[] b, int h, int k) { ... }
```

Complete the following version of quicksort so that it requires only log n space (for b[h..k] of size n). Do this by sorting recursively only the smaller of the two segments to be sorted, so the depth of recursion is O(log n) because the smaller is less than half the size of the original.

This question relies on your knowledge of (1) the original quicksort algorithm and (2) the four loopy questions used to develop a loop.

```
/** Sort b[h..k] */
public static void QS(int[] b, int h, int k) {

    int h1=   h;      int k1=     k;

    // invariant P: b[h..k] will be sorted when b[h1..k1] is sorted
    while (   k1+1-h1 >= 2            ) {
         int j= partition(b, h1, k1);
         if (j-h1 < k1-j) { QS(b, h1, j-1); h1= j+1; }
         else             { QS(b, j+1, k1); k1= j-1; }

    }
    // Postcondition: P and b[h1..k1] contains < 2 elements
}
```

3. Object-Oriented Programming (22 points)

Interface `Cloneable()`, shown to the left below, has one abstract method: `clone()`. The purpose of `clone()` in any class is to make and return a copy of the object. *That copy should have **no** objects in common with the original; all fields in it should be copied as well.*

To help implement this, class `Object` defines `clone()`; it makes and returns a copy of the object with no fields changed. Method `clone()` in a class that implements `Cloneable()` should call `clone()` in its superclass and then change the fields of the returned object so that they are different from the original.

Interface `Cloneable` and an example class `Animal` that implements `Cloneable` are given below. Note that in this question, we do not write specs for methods whose specs are obvious.

```
public interface Cloneable {
    /** Make and return a copy of this
    /*  object. That copy should have
    /* no objects in common with the
    /* original; all fields in it should
    /* be copied as well.            **/
    public Object clone()
      throws CloneNotSupportedException;
}
```

```
public class Animal implements Cloneable {
    private Date date; // date of birth

    public Animal(Date d) { ... }

    public Date getDate() { ... }

    public Object clone() { ... }
}
```

Your cat-loving friend hears about `clone()` and decides to write the following code for cloning (i.e. making copies of) cats.

```
/** An instance represents a
  *  pet's name */
public class PetName {
    private String name;

    public PetName(String name) {
        name= name;
    }

    public String getName() {
        return name;
    }
}
```

```
/** An instance represents a cat with a name.  */
public class Cat
         extends Animal implements Cloneable {
    private PetName name;

    public Cat(Date d, PetName pn) {
         date= d;
         name= pn;
     }

    public Object clone() {
        Cat c= new Cat();
        c.name= name;
        return c;
    }
}
```

(a) (2 points) Explain what it means for field `name` to be private and why this is good programming practice.

Private fields can be accessed only from within that class. This is good for abstraction and information hiding.

(b) (8 points) Identify and explain four reasons why this code might not make exact copies of cats as intended. Hint: you can ignore exceptions.

(1) Parameter name shadows field name in the PetName constructor.
(2) Field name of cloned cat points to same PetName as field name of original cat instead of creating a new copy (so updating one later will also update the other).
(3) In the Cat constructor, date cannot be referenced because it is private.
(4) Method clone() does not copy field values inherited from superclass Animal
(5) In method clone(), the object might actually be a subclass of Cat, like SiameseCat, so the copy is not copying the whole object. [Note: this answer isn't what we are looking for, since this should be handled by overriding clone() in the subclass, but should still get credit]

(c) (10 points) Write the method bodies below in classes `PetName` and `Cat` that correctly make identical copies of cats.

```
/** An instance represents a
  *  pet's name */
public class PetName {
  String name;

  public PetName(String name) {
      this.name= name;
  }

  public String getName() {
      return name;
  }
}
```

```
/** An instance represents a cat with a name. */
public class Cat
      extends Animal implements Cloneable {
  private PetName name;

  public Cat(Date d, PetName pn) {
     super(d);
     name= pn;
   }

  public Object clone()  {
     Cat c= (Cat) super.clone();
     c.name= new PetName(name.getName());
     return c;
           }
}
```

(d) (2 points) List two differences between interfaces and abstract classes. Note that one of these differences should be relevant to this example.

(1) All method declarations in an interface must be "abstract," in the sense that they do not have implementations (except for Java 8's default methods).
(2) Interfaces cannot have fields.
(3) A class can implement multiple interfaces but extend only one class.

4. Data Structures (21 points)

(a) LinkedList (6 points)

Java Collection class `LinkedList` is an implementation of doubly linked list. We provide part of the declaration of this class:

```
public class LinkedList<E> extends java.util.AbstractList<E> {
    protected int size;  // Number of nodes in the linked list.
    protected Node head; // first node of linked list (null if none)
    protected Node tail; // last node of linked list (null if none)

    protected E removeNode(Node node) {...}

    protected Node(Node pred, E element, Node succ) {
        this.pred= pred;
        this.succ= succ;
        this.data= element;
    }
```

(i) 2 points What advantage does a singly linked list have over a doubly linked list? Save space.

(ii) 4 points Complete function `removeNode`, below, by giving in the space to the right the code for STATEMENT1 and STATEMENT2. It may help to draw a few elements of the list.

```
/** Remove node from this list and return its data.
  * Precondition: node is a Node of this list; it may not be null.
protected E removeNode(Node node) {
    size--;
    if (node.pred == null)
        head= node.succ;
    else
        STATEMENT1                    STATEMENT1:
    if (node.succ == null)                   node.pred.succ= node.succ;
        tail= node.pred;
    else
        STATEMENT2                    STATEMENT2:
    E data= node.data;                       node.succ.pred= node.pred;
    return data;
}
```

(b) TreeMap (8 points)

Several classes implement Java interface `Map`. You already know about `HashMap`. Class `TreeMap` maintains a tree, where each node contains a key-value pair. The tree satisfies the following properties: (1) The tree is a binary search tree ordered by keys (note: keys must have a type that implements `Comparable`); and (2) The tree is highly balanced.

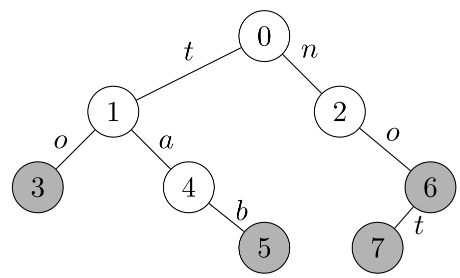
(i) 6 points Give the expected time complexity (in a `TreeMap` of size n) to:

- Find a node with a specific key. $O(log\, n)$
- Output all the keys in ascending order. $O(n)$ (inorder traversal)
- Output all the values in ascending order. $O(n\ log\ n)$ (you need to get all the values and sort them)

(ii) points Which of `HashMap` and `TreeMap` generally consumes more space? (Assume the same number of key-value pairs.) `HashMap`

(c) Trie (7 points)

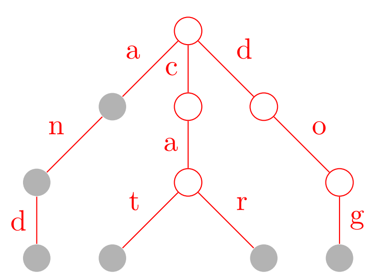
Trie (or "Prefix Tree") is a neat data structure for maintaining a set of words in the English language. Assume all letters are in lower case.
A Trie is implemented as a tree in which each node has a boolean `end` which indicates whether it is the end of a word (in the diagram, nodes are gray if `end` is true) and a 26-element array `children[0..25]`, where each element represents a lower case letter and is either null or a reference to a child node.
A word contained in a Trie is found by reading off the characters on a path from the root to a gray node.



(i) 2 points How many words does this trie contain? 4

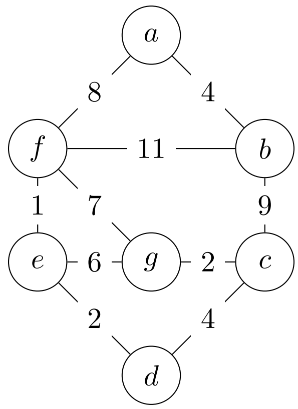
(ii) 2 points In the array of pointers for node 1, how many pointers are not null? 2

(iii) 3 points Draw the Trie for `word set = {a, an, and, cat, car, dog}`. You do not need to index the node, but you need to gray the node to indicate the end of word.



5. Graphs (10 points)

Answer questions based on the following graph.



(a) Shortest Path (2 points)

Below is a list of nodes that will be settled by Dijkstra's shortest-path algorithm, starting from node a. Which node will be added to the settled set after node f? what is the distance of this node to a?

settled nodes	a	b	f	
distance to a	0	4	8	

e, 9

(b) Planarity (2 points)

If we add edge (a, g) to the graph, is the graph still a planar graph?
Yes

(c) Depth-First Search (2 points)

Run the recursive DFS algorithm from node c, listing all the nodes in visit order. When there are several valid options, visit them in dictionary order.
c, b, a, f, e, d, g

(d) Spanning Trees (4 points)

The basic idea of Kruskal's minimal spanning tree algorithm is: "Starting from the empty tree, keep adding to the tree an edge with minimum length that does not introduce a cycle." Draw the final minimal spanning tree by Kruskal's algorithm.
The final tree is a path of g-c-d-e-f-a-b

6. Concurrency (10 points)

(a) Race Conditions (4 points)

(i) 2 points Assume a program has two threads, which execute the code in the table below.

Thread A	Thread B
x= x + 1;	x= x - 1;
	x= x - 1;

If `x` is defined by `public Integer x= 0`, what are the possible values of `x` after both threads terminate?

-2, -1, 0, 1

(ii) 2 points Which keyword in Java is used to address race conditions?

Synchronized

(b) Bounded Buffer (6 points)

Two thread—a producer and a consumer—share a common queue implemented with a fixed-size buffer. The producer generates data and puts it into the buffer. The consumer consumes the data, removing it from the buffer. Key point: the producer won't try to add to a full buffer and the consumer won't try remove data from an empty buffer.

This problem can be formulated as the following class. We have omitted the constructor. The two methods to the left have preconditions; they are private. The two methods to the right are the public ones.

```
class BoundedBuffer<E> {
  private E buffer[];
  private int h;
  private int n; // number of elements in buffer

  /** Put v into buffer.
   * Precon.: buffer is not full */
  private void put(E v) {
    buffer[(h+n) % buffer.length]= v;
    n= n+1;
  }

  /** Remove, return a value from buffer.
    * Precond.: buffer is not empty */
  private E get() { ... }

  /** If buffer is not full, put v into it. */
  public void produce(E v) {
     if (n != buffer.length) { buffer.put(v); };
  }

  /** If buffer is empty, return null;
    * else remove, return its first value.*/
  public E consume() {
     if (n == 0) return null;
     return buffer.get();
  }

}
```

(i) 2 points We haven't defined field `h`. Given the implementation of `put` in the above class, is the following statement correct? `h` is the index of the latest element that was produced. False. If $n > 0$, `h` is the index of the first element. If $n = 0$, `h` can be any value in 0..buffer.length-1.

(ii) 4 points We want to change method `consume` so that it works better in a multiprocessor environment. Instead of simply returning, it should wait until the buffer is not empty and then consume a value. Complete the correct implementation of `consume`, below, to do this. You need to fill in the **four blank lines** marked with -> on the left. Be sure to avoid race conditions. You do not need to modify function `produce`.

```
   /** Remove and return the first value from the buffer,
     * waiting if necessary until the buffer is not empty.
   public E consume() {
 ->        synchronized(this) {
               while (n == 0) {
                   try {
 ->                   wait();
                   }
                   catch (InterruptedException){}
               }
               E data= buffer.get();
 ->            notifyAll();
               return data;
 ->        }
   }
```