FINAL EXAMINATION KEY
MAY 2013
CSC 112 ♦ SECTION 01
INSTRUCTOR: NICHOLAS R. HOWE

YOU MAY USE TWO 8.5"x11" SHEETS OF NOTES ON THIS EXAM.
YOU MAY NOT USE THE TEXTBOOK, A COMPUTER, OR ANY OTHER INFORMATION SOURCE BESIDES YOUR TWO PAGES OF NOTES.

All work should be written in the exam booklet. Partial credit will be granted where appropriate if intermediate steps are shown.

1. **Vocabulary** (8 pts)

Identify the short term or phrase associated with the following definitions.

a.) A piece of code that specifies the type of a variable or field, possibly with one or more qualifiers. *Declaration*

b.) Creation of space on the heap to store the data for some reference type; typically requires use of the keyword new. Automatically triggers execution of a constructor to fill in the data. *Allocation (will accept Initialization)*

c.) The unique identifying pattern formed by the return type of a method, its name, and the types of its parameters/arguments. *Call signature*

d.) General term referring to fields, methods, or nested classes appearing within a class. *Members*

e.) In a subclass, replacing the definition of a method inherited from the parent. *Overriding*

f.) In a subclass, creating a new method with the same name as an inherited method but a different list of parameters. *Overloading*

g.) Word describing the eight types in java that are not stored by reference. *Primitive types*

h.) Qualifier used to indicate that a field or method does not depend on any particular instance of a class, and instead exists as part of the class as a whole. *Static*

2. **Sorting** (8 points)

Consider the array shown below. For each sorting algorithm indicated, show how the array would be sorted in place from smallest to largest by giving the state of the array after each swap.

33	65	81	23	50	91	58	85	74	59

a.) Insertion sort

33	*65*	*81*	*23*	*50*	*91*	*58*	*85*	*74*	*59*
33	*65*	$\boxed{23}$	$\boxed{81}$	*50*	*91*	*58*	*85*	*74*	*59*
33	$\boxed{23}$	$\boxed{65}$	*81*	*50*	*91*	*58*	*85*	*74*	*59*
$\boxed{23}$	$\boxed{33}$	*65*	*81*	*50*	*91*	*58*	*85*	*74*	*59*
23	*33*	*65*	$\boxed{50}$	$\boxed{81}$	*91*	*58*	*85*	*74*	*59*
23	*33*	$\boxed{50}$	$\boxed{65}$	*81*	*91*	*58*	*85*	*74*	*59*
23	*33*	*50*	*65*	*81*	$\boxed{58}$	$\boxed{91}$	*85*	*74*	*59*
23	*33*	*50*	*65*	$\boxed{58}$	$\boxed{81}$	*91*	*85*	*74*	*59*
23	*33*	*50*	$\boxed{58}$	$\boxed{65}$	*81*	*91*	*85*	*74*	*59*
23	*33*	*50*	*58*	*65*	*81*	$\boxed{85}$	$\boxed{91}$	*74*	*59*
23	*33*	*50*	*58*	*65*	*81*	*85*	$\boxed{74}$	$\boxed{91}$	*59*
23	*33*	*50*	*58*	*65*	*81*	$\boxed{74}$	$\boxed{85}$	*91*	*59*
23	*33*	*50*	*58*	*65*	$\boxed{74}$	$\boxed{81}$	*85*	*91*	*59*
23	*33*	*50*	*58*	*65*	*74*	*81*	*85*	$\boxed{59}$	$\boxed{91}$
23	*33*	*50*	*58*	*65*	*74*	*81*	$\boxed{59}$	$\boxed{85}$	*91*
23	*33*	*50*	*58*	*65*	*74*	$\boxed{59}$	$\boxed{81}$	*85*	*91*
23	*33*	*50*	*58*	*65*	$\boxed{59}$	$\boxed{74}$	*81*	*85*	*91*
23	*33*	*50*	*58*	$\boxed{59}$	$\boxed{65}$	*74*	*81*	*85*	*91*

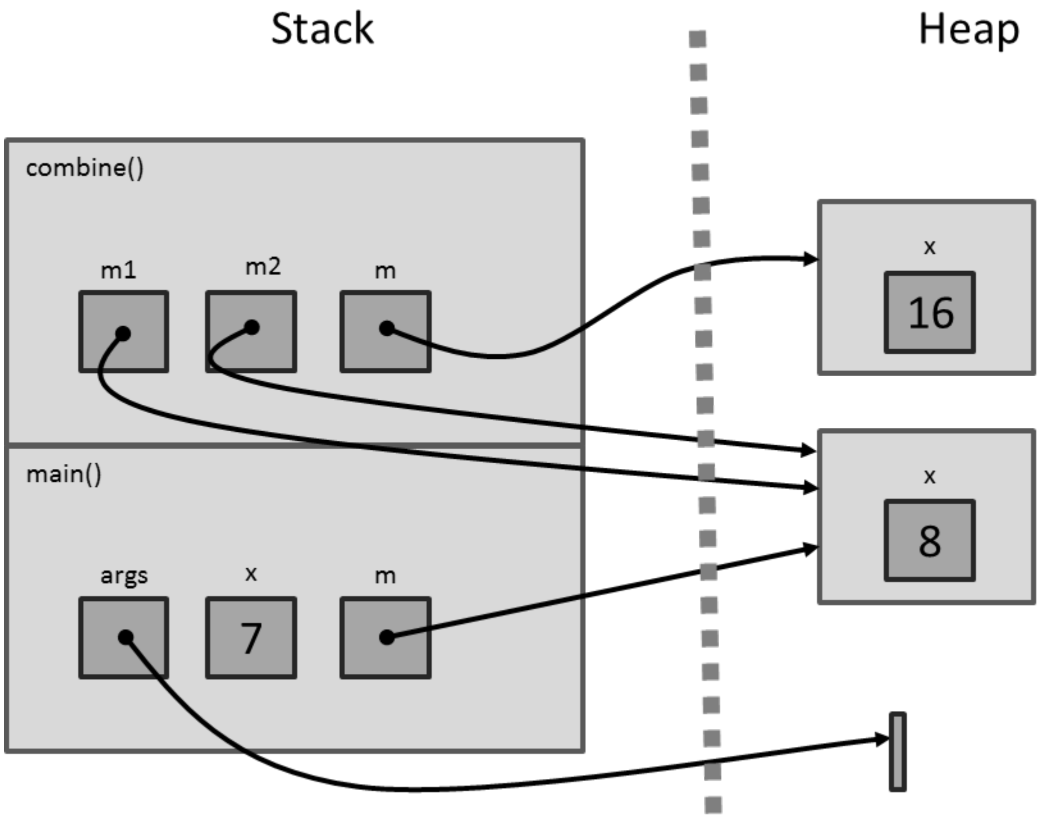
b.) Heap sort

33	*65*	*81*	*23*	*50*	*91*	*58*	*85*	*74*	*59*	
65	*33*	*81*	*23*	*50*	*91*	*58*	*85*	*74*	*59*	
81	*33*	*65*	*23*	*50*	*91*	*58*	*85*	*74*	*59*	
81	*50*	*65*	*23*	*33*	*91*	*58*	*85*	*74*	*59*	
81	*50*	*91*	*23*	*33*	*65*	*58*	*85*	*74*	*59*	
91	*50*	*81*	*23*	*33*	*65*	*58*	*85*	*74*	*59*	
91	*50*	*81*	*85*	*33*	*65*	*58*	*23*	*74*	*59*	
91	*85*	*81*	*50*	*33*	*65*	*58*	*23*	*74*	*59*	
91	*85*	*81*	*74*	*33*	*65*	*58*	*23*	*50*	*59*	
91	*85*	*81*	*74*	*59*	*65*	*58*	*23*	*50*	*33*	*// heap*
33	*85*	*81*	*74*	*59*	*65*	*58*	*23*	*50*	*91*	
85	*33*	*81*	*74*	*59*	*65*	*58*	*23*	*50*	*91*	
85	*74*	*81*	*33*	*59*	*65*	*58*	*23*	*50*	*91*	
85	*74*	*81*	*50*	*59*	*65*	*58*	*23*	*33*	*91*	
33	*74*	*81*	*50*	*59*	*65*	*58*	*23*	*85*	*91*	
81	*74*	*33*	*50*	*59*	*65*	*58*	*23*	*85*	*91*	
81	*74*	*65*	*50*	*59*	*33*	*58*	*23*	*85*	*91*	
23	*74*	*65*	*50*	*59*	*33*	*58*	*81*	*85*	*91*	
74	*23*	*65*	*50*	*59*	*33*	*58*	*81*	*85*	*91*	
74	*59*	*65*	*50*	*23*	*33*	*58*	*81*	*85*	*91*	
58	*59*	*65*	*50*	*23*	*33*	*74*	*81*	*85*	*91*	
65	*59*	*58*	*50*	*23*	*33*	*74*	*81*	*85*	*91*	
33	*59*	*58*	*50*	*23*	*65*	*74*	*81*	*85*	*91*	
59	*33*	*58*	*50*	*23*	*65*	*74*	*81*	*85*	*91*	
59	*50*	*58*	*33*	*23*	*65*	*74*	*81*	*85*	*91*	
23	*50*	*58*	*33*	*59*	*65*	*74*	*81*	*85*	*91*	
58	*50*	*23*	*33*	*59*	*65*	*74*	*81*	*85*	*91*	
33	*50*	*23*	*58*	*59*	*65*	*74*	*81*	*85*	*91*	
50	*33*	*23*	*58*	*59*	*65*	*74*	*81*	*85*	*91*	
23	*33*	*50*	*58*	*59*	*65*	*74*	*81*	*85*	*91*	
33	*23*	*50*	*58*	*59*	*65*	*74*	*81*	*85*	*91*	
23	*33*	*50*	*58*	*59*	*65*	*74*	*81*	*85*	*91*	

3. **Memory Models** (12 points)

Consider the short program below. Draw a diagram showing the state of memory when the execution reaches the line with the comment CHECKPOINT. Clearly show which variables are on the stack, and which are on the heap. Include the names of all variables in your diagram. Assume that there are no command line arguments.

```
public class Memory {
    private int x;
    public Memory(int x) {
        this.x = x;
    }
    public static Memory combine(Memory m1, Memory m2) {
        Memory m = new Memory(m1.x+m2.x);
        // CHECKPOINT
        return m;
    }
    public static void main(String[] args) {
        int x = 7;
        Memory m = new Memory(8);
        m = combine(m,m);
    }
}
```



4. **Recursion** (10 points)

Below are several attempts to write recursive programs. For each, determine whether or not it will work as desired. If it won't work, determine what is wrong and how it can be fixed. Assume that all methods compile without errors; this question is about their runtime behavior.

```
    private static int sum = 0;
    /** compute the sum of all the elements in a binary tree */
a.  public static int sumTree(BinaryTree<Integer> tree) {
        if (tree==null) {
            sum = 0;
        } else {
            sum = tree.getData();
            sum += sumTree(tree.getLeft());
            sum += sumTree(tree.getRight());
        }
        return sum;
    }
```

This one doesn't work because it uses a global field for sum, which gets overwritten.

```
    /** compute the sum of an array */
b.  public int sum(int[] a, int n) {
        return sum(a,n-1)+a[n-1];
    }
```

This one doesn't work because it has no stop condition.

```
    /** compute the nth fibonacci number */
c.  public static int fib(int n) {
        if (n==1) {
            return 1;
        } else {
            return fib(n-1)+fib(n-2);
        }
    }
```

This one doesn't work because it won't always hit the stop condition.

```
    /** compute the gcd of m and n */
d.  public static int gcd(int m, int n) {
        if (m < n) {
            return gcd(m,n-m);
        } else if (m > n) {
            return gcd(m-n,n);
        } else {
            return m;
        }
    }
```

This one should work.

```
    /** draw a fractal dragon */
e.  private static void drawDragon(int rank, Point pA, Point pD, Graphics g) {
       if (rank <= 0) {
           g.drawLine(pA.x,pA.y,pD.x,pD.y);
       } else {
           int dx = (pD.x-pA.x)/4;
           int dy = (pD.y-pA.y)/4;
           Point pB = new Point(pA.x-dy+dx,pA.y+dx+dy);
           Point pC = new Point(pD.x+dy-dx,pD.y-dx-dy);
```

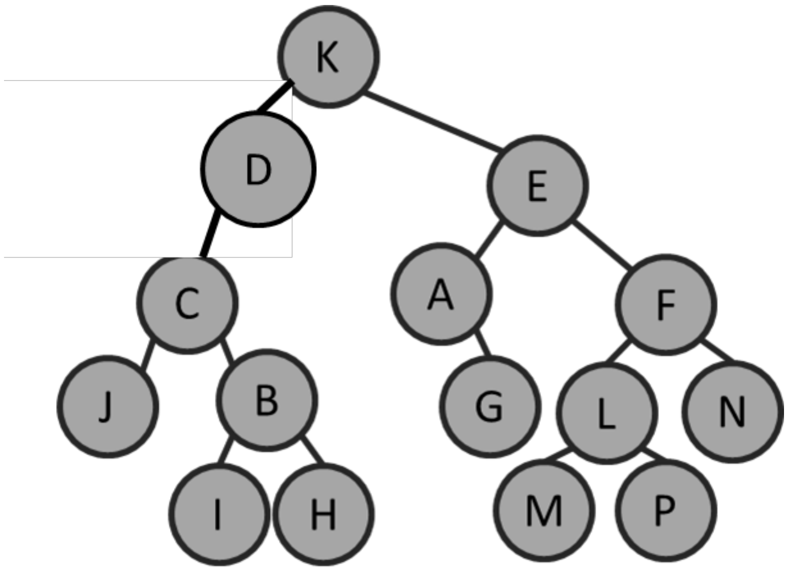
```
            drawDragon(rank-1,pA,pB,g);
            drawDragon(rank,pB,pC,g);
            drawDragon(rank-1,pC,pD,g);
        }
    }
```

This one doesn't work because it doesn't make progress towards the stop condition (rank of the central dragon doesn't go down).

5. **Trees** (12 points)

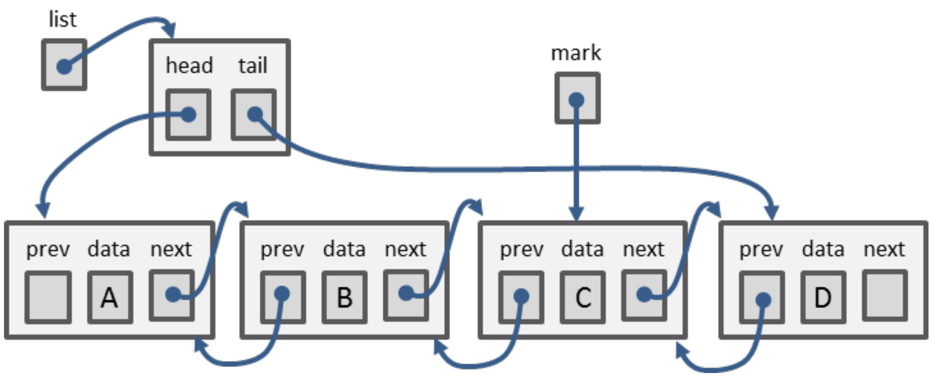
A particular tree generates the following output when the nodes are traversed in particular orders. Reproduce the tree.

Preorder: K D C J B I H E A G F L M P N
Inorder: J C I B H D K A G E M L P F N
Postorder: J I H B C D G A M P L N F E K



6. **Lists** (16 points)

Consider the diagram below, and write code to accomplish the tasks listed.



a.) Using `mark`, change the data whose value is “C” to “E”

mark.data = “E”;

b.) Using `list`, change the data whose value is “B” to “F”

list.head.next.data = “F”; // or

list.tail.prev.prev.data = “F”;

c.) Assuming that this list is implementing a queue data structure, update the date as necessary for an `out` operation.

head = head.next; head.prev = null;

d.) Using `mark`, remove the element whose value is “C” from the list.

mark.prev.next = mark.next;

mark.next.prev = mark.prev;

mark.next = mark.prev = null; // optional

7. **Data Structure Analysis** (12 points)

Consider the data structure below. (Names of methods and variables have been deliberately obscured to avoid giving away information.)

```
import java.util.*;
public class DataStructure<A,B> {
    private LinkedList<A> alist = new LinkedList<A>();
    private LinkedList<B> blist = new LinkedList<B>();
    public void op1(A a,B b) {
        if (alist.contains(a)) {
            blist.set(alist.indexOf(a),b);
        } else {
            alist.add(a);
            blist.add(b);
        }
    }
    public B op2(A a) {
        if (alist.contains(a)) {
            return blist.get(alist.indexOf(a));
        } else {
            return null;
        }
    }
}
```

a.) This class implements an abstract data type that we have studied. What is it? (Full credit for the general name; partial credit for any related data structure.)

This is an associative map. Op1 is store, op2 is lookup.

b.) Determine the running time of method op1 (big-O notation) in terms of **n**, the number of times it has been called in the past.

Because of the call to contains, this is O(n).

c.) Determine the running time of method op2 (big-O notation) in terms of **n**, the number of times op1 has been called in the past.

Also O(n)

8. **Graphs** (12 points)

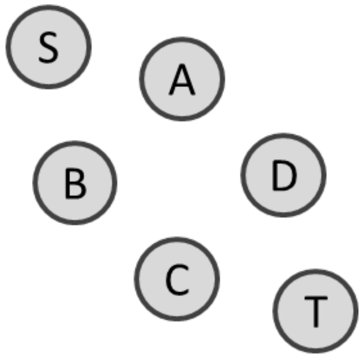
A graph has vertices {S,A,B,C,D,T} arranged in a 3x3 grid as shown at right. The capacities of different edges in this graph are shown in the table below, along with the current flow along certain edges.

a.) Draw the graph, showing all edges with nonzero capacity. You don't need to show any flow for this part.

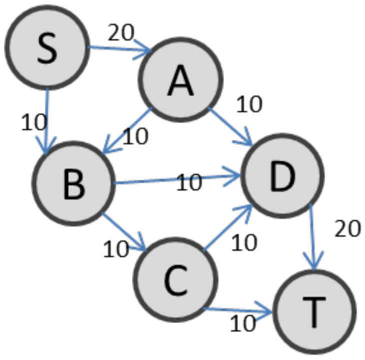
b.) Draw the residual graph.

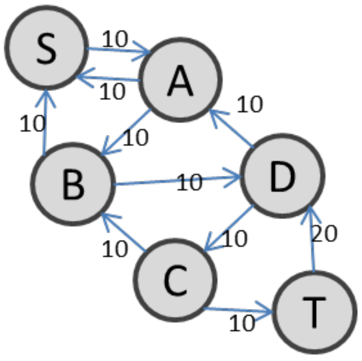
c.) Identify an augmenting path through which more flow could pass from S to T.

S-A-B-D-C-T, 10 units



Edge	Capacity	Flow
S to A	20	10
A to D	10	10
D to T	20	20
S to B	10	10
B to C	10	10
C to T	10	0
A to B	10	0
B to D	10	0
C to D	10	10





9. **Programming Style** (10 points)

We have focused on the elimination of duplication as one principle of good programming style. Explain the motivation behind this goal, and give at least five examples of ways you can reduce duplication in your own programs.

Duplication wastes resources: programs take longer to develop, are harder to maintain, and take up more space than necessary. Near-duplication where the differences are not apparent can be a source of bugs that are very hard to detect. Furthermore, identifying forms of duplication and eliminating them can lead to a better understanding of program behavior and a more abstract level of thinking.

The standard way to handle duplication is to write once anything that duplicate pieces of code have in common, and parameterize over the differences. This is done in many ways within programs written in Java (or in any other language). Loops repeat the same code but differ in the values of the loop control variables. Methods execute the same code but differ in the values of the arguments. Classes define objects that use the same set of data and behavior but may exist in multiple different instances. Generics define the behavior of either a class or method but allow the type of one or more variables to change depending on the context. Inheritance allows the definition of a child class that duplicates all the fields and methods of the parent, except for those explicitly overridden.