CS 211 **Computers and Programming**

Fall 1999

Final Exam

December 16th, 1999

NAME:____

CU ID:____

Recitation instructor/time____

You have two and a half hours to do this exam.

All programs in this exam must be written in Java.

```
*******************************************************
   Problem           Score
     1                 8

     2                 6

     3                12

     4                10

     5                20

     6                10

     7                14

     8                12

     9                 8

   Total             100
```

1. (Recursion)[8 points]

   The following rules define postfix expressions (pfe):

   ```
   pfe -> integer
   pfe -> ( pfe pfe + )
   pfe -> ( pfe pfe - )
   pfe -> ( pfe pfe * )
   ```

   Therefore, a pfe can be an integer, or a fully parenthesized expression in which the operator follows the operands.

   The static recursive method **evalPFE** below reads in a pfe from a file, and returns the value of that pfe. This method assumes that the pfe is legal. Remember that the class CS211In defines the following instance methods: **int getInt();** and **char getOp()**.

   Complete the implementation of the method in the two places indicated by (a) and (b) in the code.

```
public class PFEevaluator {

   private static CS211In pfe;

   public static int evalPFE() {
       int result = 0;
       switch (pfe.peekAtKind()) {
           case pfe.INTEGER: // (a) Fill in with your code







           case pfe.OPERATOR: // (b) Fill in with your code
















       }
       return result;
   }

   public static void main(String[] args) {
       if (args.length == 1)
           pfe = new CS211In(args[0]); // Open the file
       else {
           System.out.println("Usage: PFEevaluator <filename>");
           return;
       }
       System.out.println(evalPFE());  // evaluate the pfe
   }
}
```

Answer:

```
public class PFEevaluator {

    private static CS211In pfe;

    public static int evalPFE() {
        int result =  0;
        switch (pfe.peekAtKind()) {
            case pfe.INTEGER: // (a) Fill in with your code
                result =  pfe.getInt();
                break;
            case pfe.OPERATOR:// (b) Fill in with your code
                pfe.getOp();                  // '('
                int operand1 =  evalPFE();     //  PFE
                int operand2 =  evalPFE();     //  PFE
                char operator =  pfe.getOp();  //  operator
                pfe.getOp();                  // ')'
                switch (operator) {
                     case '+': result =  operand1 + operand2; break;
                     case '-': result =  operand1 - operand2; break;
                     case '*': result =  operand1 * operand2; break;
                }
        }
        return result;
    }

    public static void main(String[] args) {
        if (args.length = =  1)
            pfe =  new CS211In(args[0]); // Open the file
        else {
            System.out.println("Usage: PFEevaluator <filename>");
            return;
        }
        System.out.println(evalPFE());  // evaluate the pfe
    }
}
Correct base case                  2
Correct recursive case             2
Correct parsing of pfe             2
Calculate the pfe value correctly  2
```

2. (Java basics) [6 points]

   Java Nagila, our novice programmer, has just been learning about fractions. Each fraction is written as a ratio of two integers: a numerator and a non-zero denominator. For example, 1/2 is a fraction, there 1 is the numerator and 2 is the denominator. This fraction can also be represented by the real number 0.5.

   Ms. Nagila has written some code which is shown below.

   Answer the questions from (a) to (f) in the space provided in the code.

```
1 point for each correct answer


1. public interface IRealNum { double getReal(); }
2.
3. public class Real implements IRealNum {
4.     protected double rVal;
5.
6.     public Real(double r) { rVal =  r; }
7.     public double getReal() { return rVal; }
8.     // (a) Write the equals() method which returns true if the
9.     // values in the two Reals being compared are equal.
10.     public boolean equals(Object obj) {
Answer:
         if ( !(obj instanceof Real) ) return false;
         return rVal == ((Real) obj).rVal;




11.     }
12. }
13.
14. final public class Fraction extends Real {
15. // The corresponding real value is stored in the super class=20
16.     private int numerator;
17.     private int denominator;
18.
19.     public Fraction(int n, int d) {
20.         super((double)n/d);
21.         numerator = n;
22.         denominator = d;
23.     }
24.
25.     int getNumerator() { return numerator; }
26.     int getDenominator() { return denominator; }
```

```
    public static Fraction makeCopyOfFraction(Fraction q) {
         return new Fraction(q.numerator, q.denominator);
    }
}
```

```
public class Client1 {

    public static void main(String args[]){

        IRealNum  num1 = new Real(0.75);
        IRealNum  num2 = new Real(0.75);
        IRealNum  num3 = new Fraction(3,4);

        //(b) State what you expect to be printed here:
        System.out.println(num1 == num2);
```

Answer: false

```
        //(c) State what you expect to be printed here:
        System.out.println(num1.equals(num2));
```

Answer: true

```
        //(d) State what you expect to be printed here:
        System.out.println(num1.equals("0.75"));
```

Answer: false

```
      //(e) State what you expect to be printed here:
        System.out.println(num1 == num3);
```

Answer: false

```
      // (f) State what you expect to be printed here:
        System.out.println(num1.equals(num3));
```

Answer: true

```
    }
}
```

3. (Subtyping) [12 points]

   Java Nagila has written a class called Client2 that uses the IRealNum interface and the two classes Real and Fraction she defined earlier. Nagila expects the following result from Client2 when it is run:

   Fraction 1/2 has real value 0.5
   The copy also has the real value 0.5

   Unfortunately, her code contains errors. Identify all errors in her code, using the following style.

   Line 1: Missing keyword "new" in front of the constructor call.

   Your itemized list of errors and fixes should be written in the space provided at the end of this problem.

```
// See the code for IRealNum and Real and Fraction classes
// defined earlier.
public class Client2 {

    public static void main(String args[]){

1.     Fraction frac = Fraction(4,5);
2.     IRealNum numA = new Fraction(1,2);
3.     Fraction numB = numA;
4.     Real     numD = (Fraction) numA;
5.
6.     IRealNum numE = new Fraction(1,2);
7.     numE.getReal();
8.     ((Real) numE).getReal();
9.     ((Real) numE).getNumerator();
10.     int n = numE.getNumerator();
11.     int d = numE.getDenominator();
12.     double r = numE.rVal;
13.     System.out.println("Fraction " + n + "/" + d + " has real value " +=
 r);
14.
15.     Fraction numF = Fraction.makeCopyOfFraction(numE);
16.     System.out.println("The copy also has the real value " +=
 numF.getReal());
    }
}
```

```
Answer:
// See the code for IRealNum and Real and Fraction classes
// defined earlier.
public class Client2 {

public static void main(String args[]){

    Fraction frac = new Fraction(4,5);
    IRealNum numA = new Fraction(1,2);
    Fraction numB = (Fraction) numA;
    Real     numD = (Fraction) numA;

    IRealNum numE = new Fraction(1,2);
    numE.getReal();
    ((Real) numE).getReal();
    ((Fraction) numE).getNumerator();
    int n = ((Fraction)numE).getNumerator();
    int d = ((Fraction)numE).getDenominator();
    double r = ((Real)numE).rVal;//or cast to Fraction
    System.out.println("Fraction " + n + "/" + d + " has real value " + r);

    Fraction numF = Fraction.makeCopyOfFraction((Fraction)numE);
    System.out.println("The copy also has the real value " +=
 numF.getReal());
    }
}
2 points for each statement identified.

Line 3. Down casting requires an explicit cast to Fraction.
Line 9. Down casting to wrong type Real. Change Real to Fraction
Line 10. Requires an explicit cast to Fraction.
Line 11. Requires an explicit cast to Fraction.
Line 12. Requires an explicit cast to Real or Fraction.
Line 15. Incompatible parameter type. Parameter requires an explicit cast
to Fraction.

Deduct 1 point for each correct statement identified as wrong.
```

4. (Asymptotic complexity)[10 points]

(a) Order the following running times from fastest to slowest:
$O(n^{0.1})$, $O(log(n))$, $O(1)$, $O(2^n)$.
You do not have to give any explanation.

(b) A result called Stirling's formula tells us that $n! \geq (n/e)^n$. Use this result to argue that $2^n = O(n!)$. You will not get any credit unless you provide a witness pair (c, n_0) to support your answer.

(c) Java de Hutt has written the following program outline in which $|A|$ represents the number of items in an array.

```
method Gargle (array A) {
  if |A| > 1 {
        //some preprocessing work
        Compute for O(|A|log(|A|)) time;
        //Split is constant time operation
        Split A into two equal size pieces called B and C;
        Gargle(B);
        Gargle(C);
        Update A using O(|A|) time;
    }
  else Update A using some constant time operation;
}
```

Write down a recurrence equation that describes the time taken by de Hutt's program.

Answer:

(a) (4 points) $O(1), O(log(n)), O(n^{0.1}), O(2^n)$.

Give 1 point for every function in the right place in this sequence.

(b) (4 points) Since $n! \geq (n/e)^n$, it is sufficient to find a witness pair (c, n_0) such that

$2^n \leq c * (n/e)^n$ for $n \geq n_0$

It is easy to see that $(1, 6)$ serves as a witness pair because

$2^n \leq (n/e)^n$ means $n > 2 * e = 5.5...$ which is consistent with $n \geq 6$.

There are other witness pairs of course, so check the answer carefully.

(c) (2 points) T(n) = n*log(n) + 2*T(n/2)
T(1) = O(1)

Also accepted: T(n) = n*log(n) + 2*T(n/2) + n

5. (Sorting and Binary Search Trees)[20 points]

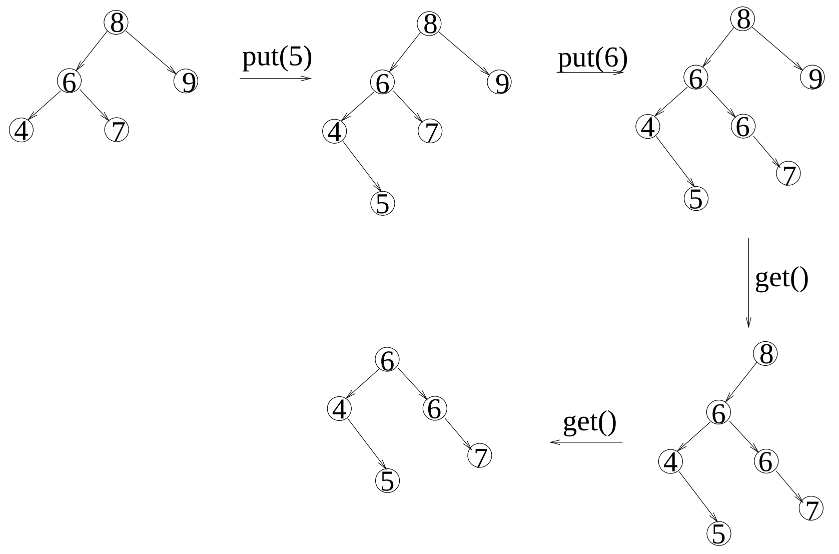
   Gill Bates, emperor of the Dork Side (DOS), wants you to implement a priority queue class using a Binary Search Tree (BST) according to the following specifications:

   - It implements the SequenceStructure interface:

     ```
     interface SeqStructure {
       void put(Object o); //stick into sequence structure
       Object get(); //extract from sequence structure
       boolean isEmpty();
       int size();}
     ```

   - The objects in the data structure must be maintained as a binary search tree. The *TreeCell* class discussed in lecture is given at the end of this exam for your use.
   - The *put* method takes an object of type Comparable and inserts it into the right place in the BST. Note that the object is stuck into the data structure regardless of whether or not there is another object equal to it already present in the data structure.
   - The *get* method should extract the largest object from the BST (an object o1 such that $o1.compareTo(o2) \geq 0$ for all other object o2 in the BST), and return it (Hint: think about the extractMax method we discussed in lecture).

   The following series of pictures shows some of the steps in building the BST when the elements in the sequence $8, 6, 9, 4, 7, 5, 6$ are *put* into the data structure, and when two *get*'s are done to the resulting data structure.



(a) Write a class named BatesMotel that meets this specification by filling in the shell shown below. Make sure your *get* method works correctly even if the largest value is at the root of the BST.

(b) What is the asymptotic complexity of the *put* and *get* methods? Justify your answer briefly.

(c) If this priority queue implementation is used to sort n values by doing n *put*'s followed by n *get*'s like in HeapSort, what is the asymptotic complexity of this sorting algorithm?

(d) Will Gill Bates be happy with this sorting method? If not, name one sorting algorithm which has better asymptotic complexity than your algorithm.

(e) What is the asymptotic complexity of quicksort? Describe an input array for which it exhibits this worst-case complexity.

(a)
```
class BatesMotel implements SeqStructure {
     protected TreeCell t;
     protected TreeCell finger;//a cursor into the data structure
     protected TreeCell previous;//previous is one step behind finger
     protected int size;

     public BatesMotel() {
       t = null;
       size = 0;
     }

     public String toString() {
       return "Tree has " + size + " elements:" + t;
     }

     public boolean isEmpty() {
          return (size == 0);
     }

     public int size() {
       return size;
     }
     //helper method
     protected boolean search(Object o){
       finger = t;
       previous = null;

       while (finger != null) {
            int test = ((Comparable)finger.getDatum()).compareTo(o);
            if (test == 0) return true;
            //need to explore one of the subtrees
            //set up the cursors correctly
            previous = finger;
            if (test < 0)
                //explore right subtree
                finger = finger.getRight();
            else
                //explore left subtree
                finger = finger.getLeft();
        }
        return false;
     }
```

```
  public void put (Object o)  {


















  }

  public Object get(){

















   }
}
```

Answer:

```
class BatesMotel implements SeqStructure {
  ......
  public void put (Object o)  {
    if (t == null){//empty tree
      t = new TreeCell(o);
      size = size + 1;
      return;
    }
    //non-empty tree:look for object first, setting up cursors
    boolean found = search(o);
    if (! found) {
        //put o into a child of previous
        int test = ((Comparable)previous.getDatum()).compareTo(o);
        if (test < 0)
            previous.setRight(new TreeCell(o));
        else
            previous.setLeft(new TreeCell(o));
        size = size + 1;
    }
    else {//make duplicate new root of right subtree
          //you can also make it new root of left subtree
        finger.setRight(new TreeCell(o,null,finger.getRight()));
        size = size + 1;
    }
  }

  1 points for checking for empty tree
  2 point for updating t correctly when tree is empty
  2 points for handling (! found) case correctly
  3 points for handling (found) case correctly


  public Object get(){
    if (t == null) {
        System.out.println("Duh: attempt to get from empty PQ");
        return null;
        }
    //set up cursors
    finger = t;
    previous = null;
    while (finger.getRight() != null)  {
        previous = finger;
        finger = finger.getRight();
        }
      if (previous != null)
          previous.setRight(finger.getLeft());
      else
          t = finger.getLeft();
      size = size - 1;
      return finger.getDatum();
     }

    1 point for empty tree handled correctly
    1 points for locating max value correctly
    2 points for removing it correctly
    1 point for updating size and returning the right value




}
```

2 points	Both methods are $O(n)$.
2 points	Sorting will take $O(n^2)$ time.
1 point	You can do better with heapsort or mergesort.
2 points	Quicksort takes $O(n^2)$ time. Bad cases are sorted arrays.

6. (10 points) Franz Liszt wants you to write a *recursive* class method called *intersection* which takes two lists L1 and L2 as parameters, and returns a new list containing only the elements that occur in both lists. This new list must not share any list cells with L1 and L2. Assume that both lists contain objects of type Comparable sorted in decreasing order.

   You do not have to clone the data objects of the Comparable type. The class ListCell discussed in lecture in given at the end of this exam.

   (a) What is the base case for your recursion? What is the output for this case?

   (b) Write the specified method.

   (c) What is the asymptotic complexity of your method?

This page intentionally left blank.

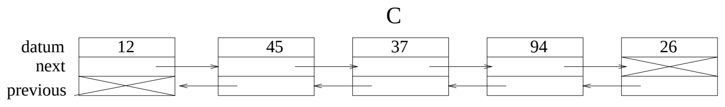
(a) Base case: one or both lists is empty. Output: empty list.

(b)
```
public static ListCell intersection(ListCell l1, ListCell  l2){
    if ((l1 == null) || (l2 == null)) return null;
    int t = l1.getDatum().compareTo(l2.getDatum());
    if (t == 0) return new ListCell(l1.getDatum(),
                                intersection(l1.getNext(),
                                             l2.getNext()
                                            )
                                );
    else
        if (t > 0) return intersection(l1.getNext(), l2);
        else return intersection(l1, l2.getNext());
}

1 point: correct base case
1 point: correct output for base case
2 points: if first objects of l1 and l2 are equal,
          that object is in output list
2 points: correct recursive call when first objects of l1 and l2 are equal
2 points: if first objects are not equal,
          correct recursive call
```

(c) (2 points) The asymptotic complexity is $O(|l1| + |l2|)$.

7. (14 points) A *doubly-linked list* (DLL) is like a linked list except that each cell has three fields called *datum*, *next* and *previous*, where *datum* contains reference to an object of type *Object*, and *next* and *previous* point to the succeeding and preceding cells in the DLL.



C.next() : returns reference to cell containing 94
C.previous(): returns reference to cell containing 45
C.first(): returns reference to cell containing 12
C.last(): returns reference to cell containing 26

(a) Write a class for implementing a DLL by filling in the following shell.

```
class DLL{

 protected Object datum;
 protected DLL next;
 protected DLL previous;

 //constructor: n and p are references to DLL cells that will
 //become the next and previous cells respectively for this cell
 //Hint: your constructor must also modify fields of n and p to
 //chain all cells up correctly.
 public DLL(Object o, DLL n, DLL p){
  ......
 }

 public Object getDatum() {
  ....
 }

 public DLL getNext() {
  ....
 }

 public DLL getPrevious() {
  ...
 }

 //return reference to first cell of DLL
 public DLL getFirst() {
   ...
  }

  //return reference to last cell of DLL
  public DLL getLast() {
   ...
  }
 }
```

(b) Write a client class that uses your DLL class to build and print a doubly-linked list containing the Integers (6,5,4) in that order.

```
class DLL {
 protected Object datum;
 protected DLL next;
 protected DLL previous;

  //4 points: one for each statement in constructor other than the first
  public DLL(Object o, DLL n, DLL p){
    datum = o;
    next = n;
    previous = p;
    if (p != null) p.next = this;
    if (n != null) n.previous = this;
  }
  //1 point
  public Object getDatum() {
    return datum;
  }
  //1 point
  public DLL getNext(){
    return next;
  }
  //1 point
  public DLL getPrevious(){
    return previous;
  }
  //3 points
  public DLL getFirst() {
    DLL finger = this;
    while (finger.previous != null)
       finger = finger.previous;
    return finger;
  }
 //3 points
 public DLL getLast() {
    DLL finger = this;
    while (finger.next != null)
        finger = finger.next;
    return finger;
 }

}
class testDLL {
 //1 point
 public static void main(String[] args) {
  DLL d = new DLL(new Integer(4), null, null);
  d = new DLL(new Integer(5), d, null);
  d = new DLL(new Integer(6), d, null);
  System.out.println(d);
 }
}
```

8. (Heaps and Binary Search Trees) (12 points)

   (a) Describe briefly what it means for a tree to be a *binary heap*.

   (b) The famous French priest Père Tree wants you to write a class method that takes a TreeCell as a parameter, and returns true if that TreeCell is the root node of a heap, and false otherwise. Write this method for him. The class TreeCell discussed in lecture is given at the end of the exam. You may assume that the heap contains objects of type Comparable.

   (c) Père Tree tells you that a binary tree is a binary search tree (BST) if the object stored at each node N in the tree is (i) greater than or equal to the object stored in the left child of N (if a left child exists), and (ii) less than or equal to the object stored in its right child of N (if a right child exists). Is he right? If not, give a counter-example to Père Tree's claim.

Answer:

(i) (1 point) A tree is a binary heap if every node has at most two children, and the value contained in a node is greater than or equal to the values contained in its children.

(ii)

```
   //This method tests if a TreeCell t is the root of a heap
public static boolean isHeap(TreeCell t) {
  //1 point for empty tree
  if (t == null) return true;
  //2 points for checking left and right subtrees for Heap property
  boolean OK = isHeap(t.getLeft()) && isHeap(t.getRight());
  Comparable datum = (Comparable)(t.getDatum());
  //2 points for checking left child and right child vs root
  //2 points for making sure that left or right child is not null
  if (t.getLeft() != null)
      OK = (datum.compareTo(t.getLeft().getDatum()) >= 0) && OK;
  if (t.getRight() != null)
      OK = (datum.compareTo(t.getRight().getDatum()) >= 0) && OK;
  return OK;
}
```

(iii) (4 points) Père Tree is wrong. The following tree satisfies his criterion but it is not a BST.

```
     7
    /
   3
  / \
 2   8
```

9. (8 points) Hashley Wilkes has a hash-table of size 10 which uses the hash function $H(X) = X \ mod \ 10$. Scarlett O'Java, whose role in life is to test Hashley, gives him input values in the following order:

   $4371, 1323, 6173, 4199, 4344, 9679, 1989$.

   (a) Draw a picture of Hashley Wilkes's hash-table after these values have been inserted into his hash-table. You may assume any order you wish for the values in a single bucket.

   (b) In the context of hash-tables, what is meant by a "collision"?

   (c) For what values in Scarlett's input are there collisions?

   Answer:

   (a)
```
(4 points total)

0.5 points for spine
0.5 points per value stored correctly

  0    1    2    3    4    5    6    7    8    9
 ------------------------------------------------
|____|____|____|____|____|____|____|____|____|____|
      |         |    |                        |
      V         V    V                        V
     4371      1323 4344                     4199
               6173                          9679
                                             1989
```

   (b) (1 point) A collision occurs when the hash function maps two or more objects in the input to the same bucket.

   (c) (3 points) The values (1323,6173), and (4199,9679,1989) have collisions.

Appendix

These classes were discussed in lecture and are provided for your use during the exam.

```
class ListCell {

 protected Object datum;
 protected ListCell next;

  public ListCell(Object o, ListCell n){
    datum = o;
    next = n;
  }

  //this is sometimes called the "car" method
  public Object getDatum() {
    return datum;
  }

  //this is sometimes called the "cdr" method
  public ListCell getNext(){
    return next;
  }

  //this is sometimes called the "rplaca" method
  public void setDatum(Object o) {
    datum = o;
  }

  //this is sometimes called the "rplacd" method
  public void setNext(ListCell l){
    next = l;
  }

 public String toString(){
   String rString = datum.toString();
   if (next == null) return rString;
   else return rString + " " + next.toString();
 }
}
```

```
class TreeCell {
  protected Object datum;
  protected TreeCell left;
  protected TreeCell right;

  public TreeCell(Object i) {
    datum = i; //left and right are null by default
  }

  public TreeCell (Object i, TreeCell l, TreeCell r) {
    datum = i;
    left = l;
    right = r;
  }

  public void setDatum(Object o) {
    this.datum = o;
  }

  public Object getDatum() {
    return datum;
  }

  public void setLeft(TreeCell t) {
    this.left = t;
  }

  public TreeCell getLeft() {
    return left;
  }

  public void setRight(TreeCell t) {
    this.right = t;
  }

  public TreeCell getRight() {
    return right;
  }

  public String toString() {
    String lString,rString;
    if (left == null)
      lString = "()";
    else
      lString = left.toString();
    if (right == null)
      rString = "()";
    else
      rString = right.toString();
    return "(" + lString + " " + datum + " " + rString + ")";
  }
}
```