CS211, Fall 2004

Prelim 2 Solutions

PART A. Inheritance (14 points total)

Consider the following class definitions.

```
public class Pair {
    public String value;
    public String name;
}

public class A {

    private Pair x;

    public A (String value) {
        x = new Pair();
        x.value = value;
    }

    public A () {
        x = new Pair();
        x.name = "A";
        x.value = "A";
    }

    final public String getValue () {
        return getx().value;
    }

    final public void setValue (String value) {
        getx().value = value;
    }

    final public String getName () {
        return x.name;
    }


    final public void setName (String name) {
        x.name = name;
    }

    protected Pair getx () {
        return x;
    }
}

public class B extends A {
```

```
    private Pair x;

    public B (String value) {
        super(value);
    }

    protected Pair getx () {
        return x;
    }
}

String test (A a) {
    a.setValue("V1");
    a.setName("N1");
    String result = "a.x.name␣␣==␣" + a.getName() + ",␣" +
                    "a.x.value␣==␣" + a.getValue();
    return result;
}
```

A.1) What is the return value of the following function invocation: test(**new** B("foo")) ? (7 points)

A.1) Solution: The execution will halt with a null pointer exception in the call to a.setValue, because getx() will return B.x which is null.

A.2) Write a class C such that the result of invoking the function test(**new** C()) is the string: "a.x.name == N1, a.x.value == V2". Recall that you cannot override **final** methods in sub-classes. (7 points)

A.2) Solution:
A full credit solution:

```
class C extends A
{
    Pair x;

    C()
    {
        super();
        x = new Pair();
    }

    protected Pair getx()
    {
        x.value = "V2";
        return x;
    }
}
```

If the student overloads the getValue or setValue methods, they can receive at most 2 points.

PART B: Searching and Sorting (30 points total)

Consider an array of colored objects, where each object can be one of three colors (red, blue, or green) and there is at least one object of each color. Write down a function to sort the array of objects by color, with the red objects appearing before blue objects appearing before green objects. Your function should have an asymptotic time complexity of O(N), where N is the number of colored objects in the input array. Your sort should be "in-place" (like Quicksort), and should not require an extra array besides the input array. Hint: Consider extending the first phase of the Quicksort algorithm, where you partition the array based on the pivot value (What would be a good pivot value in this particular case? Is it necessary to recursively sort both partitions of the array?) When writing the code for your sorting function, you should use the classes given below.

```
/**
 * Class representing colored object
 */
public class ColoredObject {

        public static final int RED = 0;
        public static final int BLUE = 1;
        public static final in GREEN = 2;

        private int color;
        private Object value;

        public ColoredObject (int color, Object value) {
                this.color = color;
                this.value = value;
        }

        public int getColor () {
                return this.color;
        }

        public Object getValue () {
                return this.value;
        }
}
```

B.1) Write a function to sort an array of colored objects as per the specifications above. The function should have the following signature: **void** sort (ColoredObject arr[]). Recall that the algorithm should have time complexity O(N), where N is the number of colored objects in the input array. (30 points)

B.1) Solution: There are many possible solutions to this problem. Below are two major approaches, each of which has variations. For example, in the quick-sort like one, one may pivot around the smallest element, in which case the recursive call will shift the beginning, not the end. Similarly, there is a symmetric variant of the second algorithm that fills stuff in from

the end. It is also possible to do something closer in spirit to counting sort, but then one would have to be careful not to undo any work.

Both routines below use the following helper:

```
void swap(ColoredObject[] arr, int a, int b)
{
    ColoredObject tmp = arr[a];
    arr[a] = arr[b];
    arr[b] = tmp;
}
```

This is an approach using pivoting similar to quicksort, as in the hint, with some comments on possible variations:

```
//Sort the [0, end) portion;
//note that if [0, end] is used, all the comparisons
//with end and the recursive calls must be adjusted
void sortAux(ColoredObject[] arr, int end)
{
    if (end == 0) //alternatively, one can do an emptiness check when trying
        return;     //to move the largest one to the end

    //Find the largest object: to be used as a pivot
    ColoredObject largest      = null;
    int           largestIndex = -1;
    for (int i = 0; i < end; ++i)
    {
        //All sorts of optimizations are possible here to stop when the
        //maximum color is found, but they're not that important
        if (largest == null || arr[i].getColor() > largest.getColor())
        {
            largest      = arr[i];
            largestIndex = i;
        }
    }

    //Possible optimization here: return if only one color.

    //alternative emptiness check here
    //if (largest == null)
    //  return;

    //Put the pivot in place
    swap(arr, largestIndex, end - 1);

    //Now, do the "collapsing walls" portion of the algorithm,
    //except we extend the right-hand portion to include
    //everything that equals to the pivot.
    int l = 0;
```

```
    int r = end - 1;
    do
    {
        //Extend left..
        while (l < end && arr[l].getColor() < largest.getColor())
            ++l;

        //Extend right..
        while (r >= 0 && arr[r].getColor() == largest.getColor())
            --r;

        //Swap if safe.
        if (l < r)
            swap(arr, l, r);
    }
    while (l < r);
    //note: needs a post-check to handle case of
    //1-element range consistently

    //Now r points to the element before the right partition:
    //so need to execute a recursive call with endpoint one right of that,
    //as it is non-inclusive
    sortAux(arr, r + 1);
}

void sort(ColoredObject[] arr)
{
    sortAux(arr, arr.length);
}
```

And the following is based on selection sort: it simply selects all the elements on the given color, and makes 2 passes to select out red and blue nodes.

```
//Sorts by moving entries of color = color to
//the front of the [begin, arr.length) portion of arr.
void sortAux(ColoredObject[] arr, int color, int begin)
{
    if (color >= ColoredObject.GREEN)
        return;

    //Write out all the entries of the given color into the array
    //by swapping them into a proper position,
    //kept track of by the out index
    int out = begin;
    for (int i = begin; i < arr.length; ++i)
    {
        //Note the invariant here: out <= i;
        //and the only way arr[out].getColor() == color
        //is if out = i, since otherwise this entry was already
```

```
        //passed by the i loop.
        if (arr[i].getColor() == color)
        {
            swap(arr, i, out);
            ++out;
        }
    }

    //out now points one past the region of the color color:
    //that's where the next color may start.
    sortAux(arr, color + 1, out);
}

void sort(ColoredObject[] arr)
{
    sortAux(arr, ColoredObject.RED, 0);
}
```

B.1) Grading:

- +4 pts: for comments that explain the approach taken, even if incorrect.
- Implementation that are not linear in time should receive no correctness credit (but may receive comment credit), unless the complexity appears to be due to a bug and not flawed design.
- +10 pts: if code or comments suggest something that's akin to a working algorithm, but which would not actually come close to working; partial fraction of this should be given if the student seems to have understood the ideas underlying a solution, but not put them together into an overall shape of solution.
- +26 pts: essentially a working algorithm, but may have bugs, which will have deductions as below (but this should not be deducted to below +10, as that's the baseline for getting it but not implementing properly)
- -10 pts: major bugs: portions of implementation mostly incorrect
- -5 pts: substantial bugs that affect alogithm correctness
- -2 pts: each off-by-one style error, including loop bounds (but not the pivoting and similar comparisons)
- -2 pts: for each missing bound checks in the partitioning loop, if any.
- -1 pts: if any minor syntax errors that could be considered typos (total, only one of these deductions should be made)

PART C: Asymptotic Complexity (20 points total)

C.1) Consider four functions $f(n)$, $g(n)$, $d(n)$, $h(n)$, which only return non-negative values. If $f(n)$ is $O(g(n))$ and $d(n)$ is $O(h(n))$, then $f(n) \cdot d(n)$ is $O(g(n) \cdot h(n))$. Justify this by finding an appropriate witness pair. (7 points)

C.1) Solution: Since $f(n)$ is $O(g(n))$, there is some witness pair (n_f, c_f) such that for all $n > n_f$, $f(n) \leq c_f g(n)$. Likewise there is some witness pair (n_d, c_d) such that for all (n_d, c_d), it is the case that $d(n) \leq c_d h(n)$.

Thus, if $n > \max(n_f, n_d)$, then $n > n_f$ so $f(n) \leq c_f g(n)$, and n is also greater than n_d, so $d(n) \leq c_d h(n)$. Thus $f(n) \cdot d(n) \geq c_f c_d g(n) h(n)$, so $(\max(n_f, n_d), c_f c_d)$ is an appropriate witness pair.

There are other possible witness pairs, for example $(n_f + n_d, c_f c_d)$, and students should get full credit for *any* valid witness pair with justification. However a student who does not define n_f, n_d and so on shall receive no more that 2 points. A student who defines these, but guesses something other than $(\max(n_f, n_d), c_f c_d)$ without any justification shall receive no more than 3 points.

C.2) Consider the following function:

```
public void test (int n) {
    int a = 0;
    for (int i=0; i <n*n; i++)
        for (int j = 0 ; j <= i; j++)
            a = a + i * j;
}
```

What is the asymptotic time complexity of this function? Briefly justify your answer. You will not receive credit without the proper justification. (7 points).

C.2) Solution: The running time of the inner loop is $O(i)$. The running time for the outer loop is

$$\sum_{i=0}^{n^2} O(i) = O\left(\frac{n^2(n^2+1)}{2}\right) = O(n^4)$$

If the student just writes $O(n^4)$ without some justification, no points shall be awarded. If the student sees that the inner loop runs for n^2 iterations and that the inner loop takes $O(i)$ time, but gets the analysis wrong, then you may award up to 4 points.

C.3) Consider the following pseudocode:

```
public void test2 (array A) {
   if size of A is 0, 1 or 2
        do some fixed amount of work;
   otherwise
        divide the array A into 3 equal parts, and call test2 recursively on each piece;
        do work proportional to the size of the array A;
}
```

Write down the recurrence equation for the running time of the following pseudo-code. (6 points)

C.3) Solution:

$$\begin{aligned} T(0) &= O(1) \\ T(1) &= O(1) \\ T(2) &= O(1) \quad \text{1 point for these 3 above} \\ T(n) &= 3T(n/3) + O(n) \quad \text{6 points} \end{aligned}$$

If the student uses constants instead of big-O notation, deduct two points, unless they use the same constants, in which case they should lose 4 points (i.e. for $T(0) = c$, $T(n) = 3T(n/3) + cn + c$ or something).

PART D. Sequence Structures, Generic Programming, and Exceptions (36 points total)

D.1) Consider the complete heap in Figure 1 (i.e., a heap that is a complete binary tree).

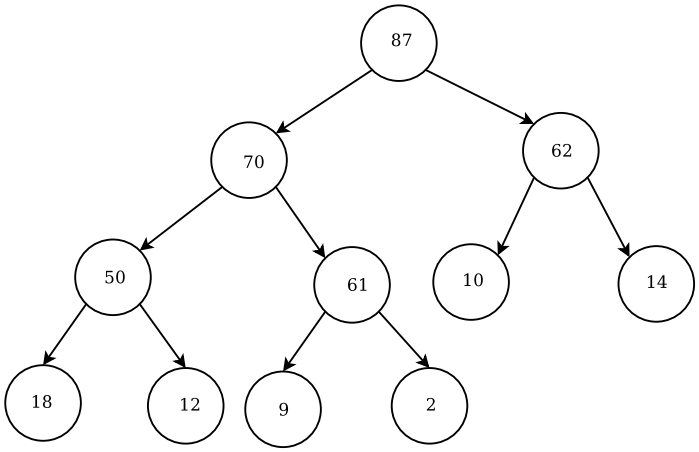

Figure 1: Initial heap

D.1) Draw the state of the heap after a get operation has been performed and the heap has been reorganized to again be a complete heap (you only need to draw the final state of the new complete heap). You should use the reorganization algorithm discussed in class. (6 points)

D.1) Solution: The solution is given in Figure 2. 4 points are given for a correct heap and 2 points for a complete tree.

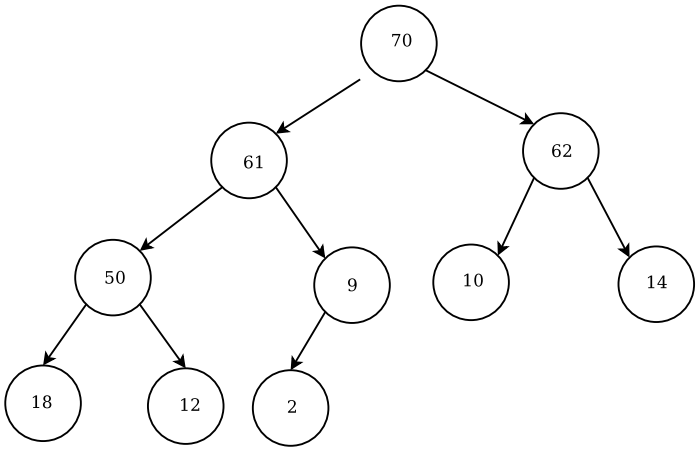

Figure 2: Final heap

D.2) Create an iterator that traverses a binary tree in preorder. The constructor of the iterator should have time complexity $O(1)$. Each invocation of the `next()` operation in the iterator should have time complexity $O(h)$, where h is the height of the binary tree.

To implement the iterator, you have to use the classes given below and implement the methods in the `PreorderIterator` inner class of the `Tree` class. Make sure that exceptions are handled and thrown correctly, as per the function signatures.

Hint: Consider using the stack sequence structure in `PreorderIterator` to keep track of the current state of the iteration. Implement the stack using the `ListCell` class provided below.

D.2) Implement the PreorderIterator class as per the above specifications. Recall that the constructor should have time complexity O(1) and the next() operation should have time complexity O(h), where h is the height of the binary tree. Also, make sure that you handle and throw exceptions correctly. (30 points).

```
public class Tree {

        private BinaryTreeNode root;

        public Tree (BinaryTreeNode root, int height) {
                this.root = root;
                this.height = height;
        }

        public Iterator  preorderIterator () throws IteratorException {
                return new PreorderIterator();
        }

        /**
          * The inner iterator  class
          */
        private class PreorderIterator {

              // add variables  as  required

                public PreorderIterator() throws IteratorException {
                        // implement this.
                }

                public boolean hasNext() {
                       // implement this.
                }

                public BinaryTreeNode next() throws IteratorException {
                      // implement this
                }
        }
}
```

```
/**
  * The binary tree node interface.
  */
public interface BinaryTreeNode {
        public Object getData() throws TreeException;
        public BinaryTreeNode getLeft() throws TreeException;
        public BinaryTreeNode getRight() throws TreeException;
}

/**
  * The iterator exception class
  */
public class IteratorException extends Exception {

        private String errorMessage;

        IteratorException (String errorMessage) {
                this.errorMessage = errorMessage;
        }

        String toString () {
                return this.errorMessage;
        }
}

/**
  * The tree exception class
  */
public class TreeException extends Exception {

        private String errorMessage;

        TreeException (String errorMessage) {
                this.errorMessage = errorMessage;
        }

        String toString () {
                return this.errorMessage;
        }
}

/**
  * The ListCell class
  */
public class ListCell {

      private Object datum;
      private ListCell next;
```

```
    public ListCell (Object datum, ListCell next) {
        this.datum = datum;
        this.next = next;
    }

    public Object getDatum () {
        return this.datum;
    }

    public ListCell getNext () {
        return this.next;
    }
}
```

D.2) Solution: The solution follows. The points are assigned in the following way:

For a solution that does not demonstrate adequate understanding of problem 5-10pts (max 10) with points given for:

- +5: some stack operations
- upto +5: hasNext() works, etc.

Solution that demonstrates understanding:

- +5 for correct complexity of constructor and next()
- +5 for correct constructor
- +3 for correct hasNext()
- +4 for try block within next()
- +3 for catch of TreeException + throw IteratorException within next()
- +5 for stack class or implicit stack within PreorderIterator
- +5 for correct order of getRight(), getLeft() in next()
- -2 for not checking for (!hasNext()) in next()
- -1 for no return node; in next()
- -1 for no error message in catch block

```
/**
 * The stack class
 */
public class Stack
{
    private ListCell tos;

    public Stack()
    {
    }
```

```
    public boolean empty()
    {
        return tos == null;
    }

    public void push(Object datum)
    {
        ListCell newCell = new ListCell(datum, tos);
        // reset top of the stack pointer
        tos = newCell;
    }

    public Object pop()
    {
        if (empty())
            return null;

        Object retVal = tos.getDatum();
        tos = tos.getNext();

        return retVal;
    }
}

/**
* The tree class
*/
public class Tree
{
    BinaryTreeNode root;

    public Tree(BinaryTreeNode root)
    {
        this.root = root;
    }

    public PreorderIterator preorderIterator() throws IteratorException
    {
        return new PreorderIterator();
    }

    /**
    * The inner iterator class
    */
    protected class PreorderIterator
    {
        private Stack stack;
```

```
        public PreorderIterator() throws IteratorException
        {
            stack = new Stack();
            stack.push(root);
        }

        public boolean hasNext()
        {
            return !stack.empty();
        }

        public BinaryTreeNode next() throws IteratorException
        {
            if (!hasNext())
                return null;

            // pop the current node
            BinaryTreeNode node = (BinaryTreeNode) stack.pop();
            // push it's children on the stack if any, right first to respect the order
            try
            {
                if (node.getRight() != null)
                {
                    stack.push(node.getRight());
                }
                if (node.getLeft() != null)
                {
                    stack.push(node.getLeft());
                }

                return node;
            }
            catch (TreeException e)
            {
                throw new IteratorException(e.getMessage());
            }
        }

    }
}
```