Solutions

Name: ______________________ NetID: ____________

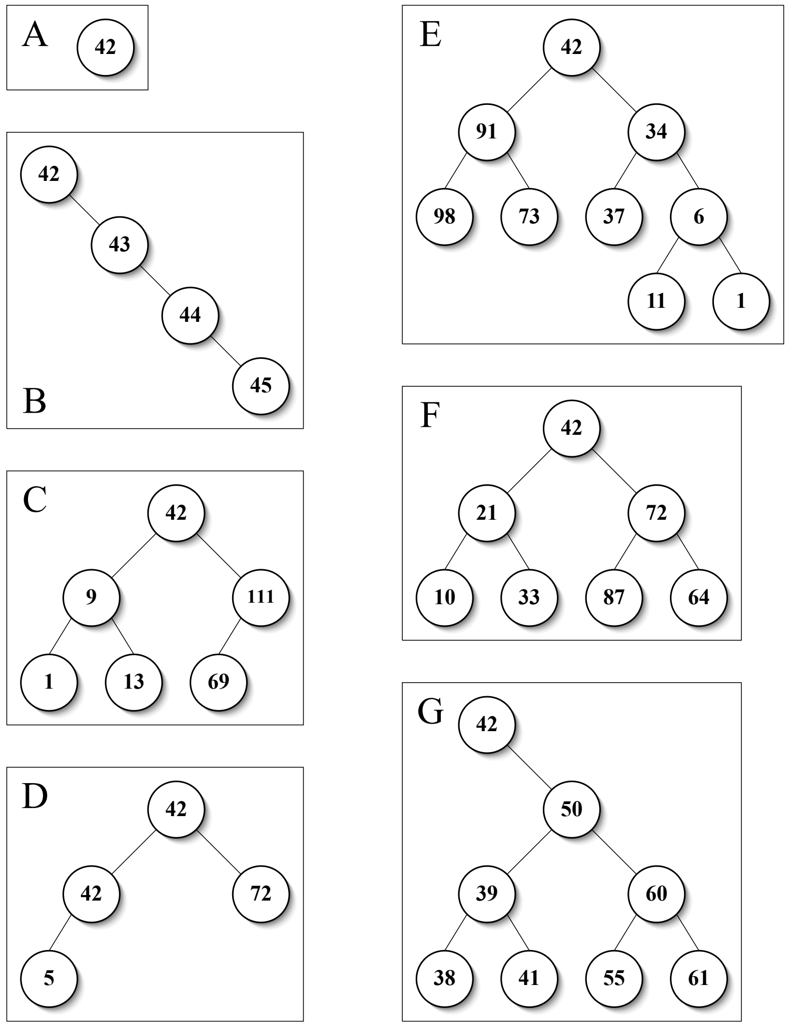
This exam has 8 pages — check now to make sure that you have them all. There are 5 questions worth 100 points total. You will have one hour and fifteen minutes to complete the exam.

This exam is closed book and closed notes. Calculators are not allowed (though I doubt they'd be helpful). For partial credit, you must show your work. Don't spend too much time on any one problem. Abide by Cornell's Code of Academic Integrity.

Problem	Points	Score
1	25	
2	25	
3	25	
4	10	
5	15	
Total	100	

1. Binary Search Trees [25 pts]

(a) [10 pts] Which of A-F are valid binary search trees using the standard compareTo ordering? What is wrong with the trees that are not BSTs?



Answer:
Binary search trees: Nodes in BSTs have at most two children. For a node x with value val, nodes below and to the left of x must have value less than val. Nodes below and to the right of x must have value greater than val. Thus, tree's A-C are definately BSTs and tree's F and G are definately not BSTs. Tree D has a duplicate value, typically not allowed in BSTs. Tree E is a BST under a reverse ordering, but not under the standard ordering. Either answer was accepted for D and E as long as it was justified.

(b) [5 pts] Which of the following traversals will visit the nodes of a binary tree in sorted order?

i. preorder

ii. inorder

iii. postorder

iv. mailorder

Answer:

Inorder (visit left child, then self, then right child) will visit the nodes of a BST in sorted order.

(c) [10 pts] Complete the following *recursive* method:

```
class BSTNode
{
   public BSTNode left;
   public BSTNode right;
   public int     value;
}

/** Inserts a value into a binary search tree, and returns the root of the
 *  resulting tree.
 *  @param root     the tree to be updated
 *  @param toInsert the integer to be inserted
 *  @returns        the resulting tree
 */
BSTNode BST-insert( BSTNode root, int toInsert )
{
```

Answer:

```
/* if root.val = val, do nothing (thereby avoiding duplicates) */
if (root == null) /* base case, avoids null ptr exception */
   return new BSTNode(val,null,null);
else if (root.val < val) /* recurse to left descendents */
   root.left = BSTInsert(root.left,val);
else if (root.val > val) /* recurse to right descendents */
   root.right = BSTInsert(root.right,val);
else throw new Exception( "cannot insert duplicate values" );
return root;
```

```
}
```

2. Induction [25 pts]

Consider the following algorithm:

```
/** Removes the odd-indexed elements from a singly-linked list
 *  @param head the head of the list to be filtered
 */
static void dropOdds( ListCell head ) {
   if( head == null || head.next == null )
      return;
   head.next = head.next.next
   dropOdds( head.next )
}
```

Use induction to show that if ℓ has n elements, then after calling `dropOdds`(ℓ), ℓ will contain $\lceil \frac{n}{2} \rceil$ elements.

Answer:

Some notes: If $|x|$ denotes the length of list x, then:

- $|x.next| = |x| - 1$
- $|x| = |x.next| + 1$
- $|x.next.next| = |x| - 2$.

base case *if head has 0 or 1 elements In this case, either:*

- *head == null (0 elements), or*
- *head.next == null (1 element)*

In both these cases, dropOdds does nothing, leaving the list with 0 or 1 elements, respectively. Conveniently, $\lceil 0/2 \rceil = 0$ and $\lceil 1/2 \rceil = 1$ as desired.

(strong) inductive hypothesis *Assume that if any list ℓ of length $k \leq n$ is passed into* `dropOdds`*, then when* `dropOdds` *returns, ℓ will contain $\lceil k/2 \rceil$ elements.*

inductive step *Suppose $|\ell| = n+1$. We want to show that after* `dropOdds` *returns, that $|\ell| = \lceil \frac{n+1}{2} \rceil$. Because $n > 0$ ($n = 0$ is handled as a base case), $n + 1 > 1$, and thus* `head != null` *and* `head.next != null`*. In this case,* `dropOdds(head)` *assigns* `head.next` *to* `head.next.next` *and recursively calls* `dropOdds(head.next)`*.*

Before the assignment:

- $|\texttt{head}| = n + 1$
- $|\texttt{head.next.next}| = n - 1$

After the assignment:

- $|\texttt{head.next}| = n - 1$
- $|\texttt{head}| = |\texttt{head.next}| + 1 = n$

After recursive call (by inductive hypothesis ... as $|\texttt{head.next}| \leq n$):

- $|\texttt{head.next}| = \lceil \frac{n-1}{2} \rceil$
- $|\texttt{head}| = \lceil \frac{n-1}{2} \rceil + 1$.

Conveniently, $\lceil \frac{n-1}{2} \rceil + 1 = \lceil \frac{n+1}{2} \rceil$, which is what we set out to prove.

3. Inheritance [25 pts] This question refers to the class hierarchy on page 6. You may detach that page for reference.

Examine the following lines of code, and do the following:

(a) Cross out any lines that wouldn't compile.

(b) Circle any lines that would throw an exception.

(c) Write the output for the remaining lines.

```
A a = new B();                                    // compiles and runs
System.out.println( a.x );                        // prints A.x
System.out.println( a.y );                        // prints A.y
System.out.println( a.contents() );               // prints B.x A.y

B b = (B) a;                                      // compiles and runs
System.out.println( b.x );                        // prints B.x
System.out.println( b.y );                        // prints A.y
System.out.println( b.contents() );               // prints B.x A.y

C c = new C();                                    // compiles and runs
System.out.println( c.x );                        // prints C.x
System.out.println( c.y );                        // prints A.y
System.out.println( c.contents() );               // prints A.x C.y

a = new C() { public String getX(){ return x; } };  // compiles and runs
System.out.println( a.contents() );               // prints C.x C.y

List<A> alist1 = null;                            // compiles and runs
alist1 = new ArrayList<A>();                      // compiles and runs
alist1.add( new A() );                            // does not compile
alist1.add( new C() );                            // compiles and runs

List<A> alist2 = null;                            // compiles and runs
alist2 = new ArrayList<B>();                      // does not compile
alist2.add( new B() );                            // null pointer exception
alist2.add( new C() );                            // null pointer exception

List<? extends A> wlist = null;                   // compiles and runs
wlist = new ArrayList<B>();                       // compiles and runs
wlist.add( new B() );                             // does not compile
wlist.add( new A() );                             // does not compile
A a = wlist.get( 0 );                             // index out of bounds exception
B b = wlist.get( 0 );                             // does not compile
```

Here is the class hierarchy for question 3:

```
abstract class A
{
   public String x = "A.x";
   public String y = "A.y";

   public String getX() {
      return x;
   }

   public abstract String getY();

   public String contents() {
      return getX() + ", " + getY();
   }
}

class B extends A
{
   public String x = "B.x";

   public String getX() { return x; }
   public String getY() { return y; }
}


class C extends A
{
   private String y = "C.y";
   public  String x = "C.x";

   public String getY() { return y; }
}
```

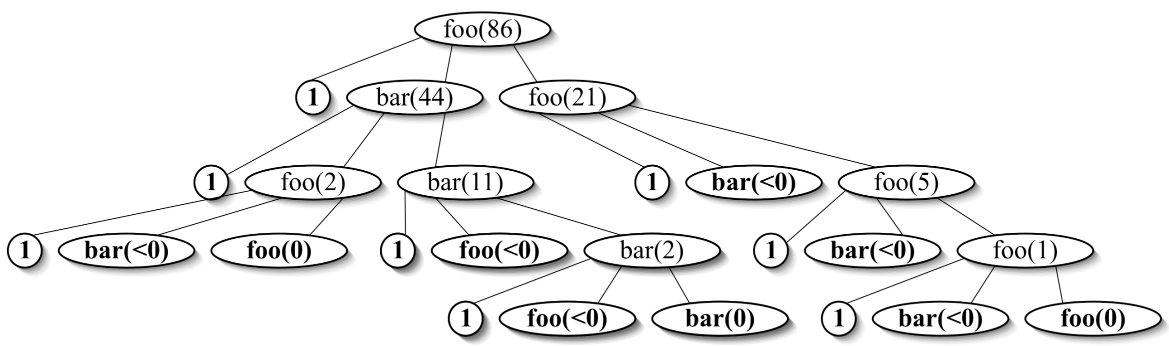
Just for reference, here are the relevant methods of the `List` and `ArrayList` types:

```
public interface List<E> extends Collection<E>
{
   public void add( E o );
   public E    get( int i );
}

public class ArrayList<E> implements List<E>
{
   // ...
}
```

4. Recursion [10 pts] What text is output during the method call `foobar(86)`?

```
public void foobar(int x) {
   System.out.println( "result: (" + foo(x) + ")" );
}

public int foo(int x) {
   if( x <= 0 ) return 1;

   System.out.println( "foo( " + x + " )" );
   return 1 + bar(x-42) + foo(x/4);
}

public int bar(int y) {
   if( y <= 0 ) return 1;

   System.out.println( "bar( " + y + " )" );
   return 1 + foo(y-42) + bar(y/4);
}
```

Answer:
Here is the call tree:



This yields the following output:

```
foo( 86 )
bar( 44 )
foo( 2 )
bar( 11 )
bar( 2 )
foo( 21 )
foo( 5 )
foo( 1 )
result: (17)
```

5. Grammars and Parsing [15 pts]

Recall the grammar for the InfoStructure language:

value	→	*struct* | *array* | NUMBER | `STRING`
struct	→	`NAME OPEN_PAR` *attr_list* `CLOSE_PAR`
attr_list	→	*attr attr_tail* | ϵ
attr_tail	→	`COMMA` *attr attr_tail* | ϵ
attr	→	`NAME EQUALS` *value*
array	→	`OPEN_BRACE` *value_list* `CLOSE_BRACE`
value_list	→	*value value_tail* | ϵ
value_tail	→	`COMMA` *value value_tail* | ϵ

Draw the parse tree that shows that the following is a valid InfoStructure document:

```
State( name = "New York", cities = {"New York", "Ithaca"} )
```

You may abbreviate the single-character tokens (e.g. write **{** instead of `CLOSE_BRACE`).

Answer: