At the county fair, prizes are awarded to the five heaviest cows. More than 2000 cows are entered, and their records are stored in an array. Which of the following algorithms provides the most efficient way of finding the records of the five heaviest cows?

(A) Selection Sort
(B) Selection Sort terminated after the first five interations
(C) Insertion Sort
(D) Insertion Sort terminated after the first five iterations
(E) Mergesort

(B) Selection Sort terminated after the first five interations
Selection sort finds the largest element in the first iteration, then the second largest on the second iteration, and so forth.

A thought about yesterday. . .

```
void insert (TreeNode node, Object data)
// Will insert data into an ordered binary tree.
// The solution is recursive.
```

IN ACTUAL PRACTICE . . .

```
public void insert(Object data)
{
    ???  insertHelper(myRoot, data);
}

private TreeNode insertHelper(TreeNode node, Object data)
{
}
```

OUR RECURSIVE METHOD WHICH WILL PASS BACK NEW NODES AS A REFERENCE

STUDENT OUTLINE

Lesson 34 – Binary Tree Algorithms

INTRODUCTION: Having built a binary tree in the memory of your computer, we need the algorithms to verify that the tree is indeed ordered. The recursive inorder algorithm is elegant, clean, but difficult to follow. You must understand this algorithm, for it provides a template for many binary tree routines. Once convinced that the tree is correct we will code a recursive search algorithm.

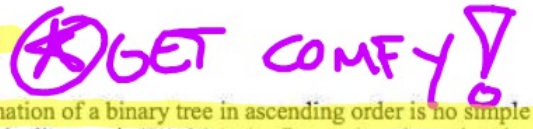
The key topics for this lesson are:
A. Inorder Tree Traversal
B. Preorder and Postorder Tree Traversals
C. Counting the Nodes in a Tree
D. Searching a Binary Tree

VOCABULARY:

TREE TRAVERSAL	VISITING A NODE
INORDER	PREORDER
POSTORDER	

DISCUSSION:

A. Inorder Tree Traversal



See Transparency T.A. 34.1, *Printing A Binary Tree.*

1. Printing out the information of a binary tree in ascending order is no simple task. Using the example diagram in T.A.34.1, the first node value printed should be 9, and getting there is fairly simple. The next value is 14, then 21, then comes a big problem - how do we get back to the root node whose value is 26? This is a backtracking problem that is best solved with recursion.

2. As a review, assume the following class definition applies in this student outline.

```
public class TreeNode
{
  private Object value;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(Object initValue, TreeNode initLeft,
                  TreeNode initRight)
  {
    value = initValue;
    left = initLeft;
    right = initRight;
  }

  public Object getValue()
  {
    return value;
  }

  public TreeNode getLeft()
  {
    return left;
  }
```

```
  public TreeNode getRight()
  {
    return right;
  }

  public void setValue(Object theNewValue)
  {
    value = theNewValue;
  }

  public void setLeft(TreeNode theNewLeft)
  {
    left = theNewLeft;
  }

  public void setRight(TreeNode theNewRight)
  {
    right = theNewRight;
  }
}
```

3. A tree traversal is an algorithm that visits every node in the tree. To visit a node means to process something regarding the data stored in the node. For now, visiting the node will involve printing the *value* object field.

4. An inorder tree traversal visits every node in a certain order. Each node is processed in the following sequence:

DISPLAY DATA

RECURSIVE CALLS

Solve left subtree inorder
Visit node
Solve right subtree inorder

Notice that visiting the node is placed between the two recursive calls.

5. Here is the code for the `inorder` method:

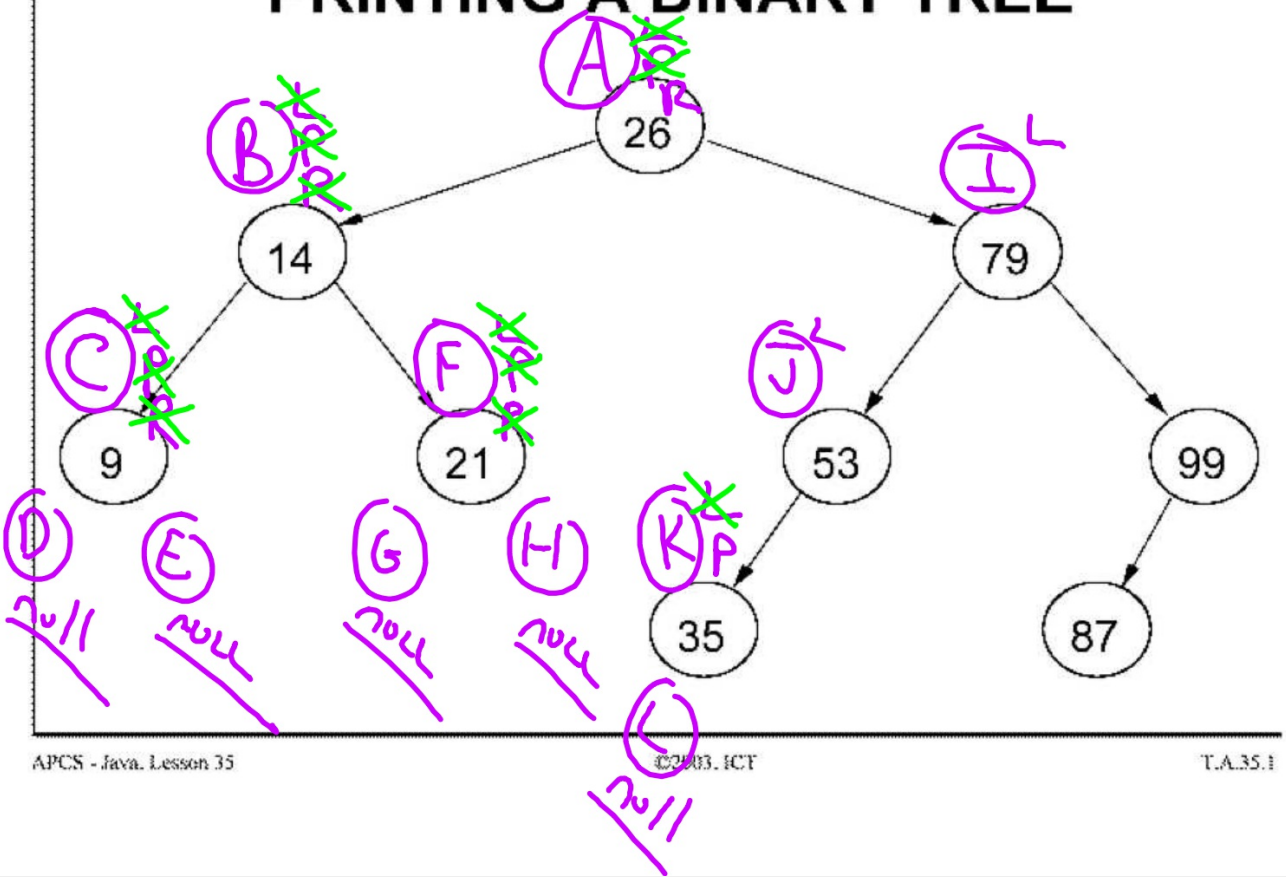
```
void inorder (TreeNode temp)
{
  if (temp != null)
  {
    inorder (temp.getLeft());
    System.out.println(temp.getValue());
    inorder (temp.getRight());
  }
}
```

6. The first call of `inorder` is asked to process the root node. The first step of the method is a recursive call of `inorder` to process the left pointer of the root node. This recursive call to solve the left pointer will take place before the `System.out.println` statement.

7. Solving for the node with value 14 results in another recursive call to solve the left pointer to the node having value 9. The `inorder` call to process the node with value 9 in turn calls `inorder`, which hits a **`null`**. This recursive call results in nothing executed.

9 14 21 26 35 ETCETERA

PRINTING A BINARY TREE



DESCRIBED IN PICTURE ON PREVIOUS PAGE

8. The recursion backtracks to the inorder processing of the node with value 9. Our first output occurs and the value 9 is printed. We recursively visit the node to the right and since nothing is there, we return to the node with value 9.

9. For the node with value 9, we have now done all three steps. We checked left, printed the 9, and checked right. This method call is done and we backtrack to where we left off, which is processing the node with value 14.

10. From here, the recursion will continue working its way through the tree. Inorder calls which are postponed are placed on the stack. When a call of `inorder` is completed, the program will go to the stack (if necessary) to backtrack through the tree.

B. Preorder and Postorder Tree Traversals

1. A preorder tree traversal processes each node in a different order.

   Visit the node
   Process the left subtree preorder
   Process the right subtree preorder

   The only difference is that we will visit first, then go left, then right. The preorder output of the same binary tree will be:

   ```
   26  14  9  21  79  53  35  99  87
   ```

2. A postorder tree traversal has this order:

   Process left subtree postorder
   Process right subtree postorder
   Visit the node

   The prefix "post" refers to after, hence the location of visiting the node after the recursive calls. The printout of the same tree will be as follows:

   ```
   9  21  14  35  53  87  99  79  26
   ```

C. Counting the Nodes in a Tree

1. A standard binary tree algorithm is to count the number of nodes in the tree. Here is a pseudocode version.

   AN INORDER TRAVERSAL

   ```
   Count left subtree recursively
   Count the current node as one
   Count right subtree recursively
   ```

2. As you develop the code, consider what base case will terminate the recursion.

D. Searching a Binary Tree

1. Searching an ordered binary tree can be solved iteratively or recursively. Here is the iterative version.

```
TreeNode find(TreeNode root, Comparable valueToFind)
{
  TreeNode node = root;

  while (node != null)
  {
    int result = valueToFind.compareTo(node.getValue());
    if (result == 0)
      return node;
    else if (result < 0)
      node = node.getLeft();
    else  // if (result > 0)
      node = node.getRight();
  }
  return null;
}
```

2. If the value is not in the tree, the node pointer will eventually hit a `null`.

3. Notice the type of the argument, `valueToFind`, in the `find` method is designated as `Comparable`. This means that `valueToFind` belongs to a class that implements the library interface `Comparable`. `find`'s code calls the `compareTo` method of the `valueToFind` object to determine the ordering relationship – that's why `valueToFind` must be a `Comparable`, not just an `Object`.

4. A recursive version is left for you to solve as part of the lab exercise.

5. The order of searching an ordered binary tree is $O(\log_2 N)$ for the best case situation. For a perfectly balanced tree, the capacity of each level is $2^{\text{level \#}}$.

Level #	Capacity of Level	Capacity of Tree
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
5	32	63
etc.		

6. So starting at the root node, a tree of 63 nodes would require a maximum of 5 left or right moves to find a value. The number of steps in searching an ordered binary tree is approximately $O(\log_2 N)$.

SUMMARY/ REVIEW: The most important topic of this lesson is recursive algorithms required to process binary trees. Study the examples, draw pictures, do whatever it takes to understand recursive tree traversals. Many of the algorithms in future lessons will require recursion.

ASSIGNMENT: Lab Exercise L.A.34.1, *BSTree (Part 2)*