

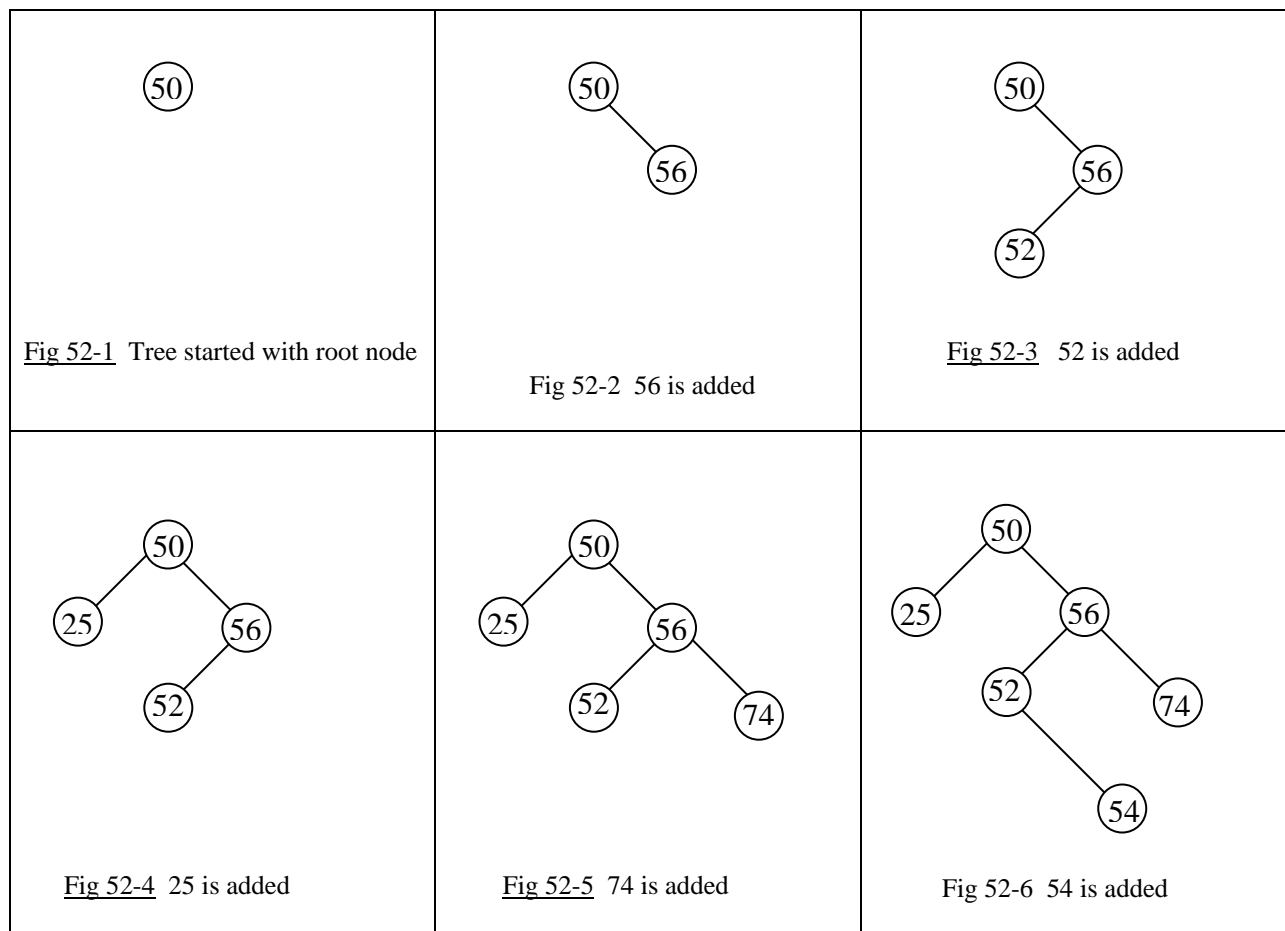
Lesson 52..... Binary Search Tree

We will begin by showing how a binary search tree is constructed. We will construct our tree from the following sequence of integers; 50, 56, 52, 25, 74, and 54. Each number will result in a “node” being constructed. The nodes in the series of figures below are depicted with circles, and the sequence of figures shows the sequence in which the nodes are added.

The rules:

As can be observed from the drawings, a new integer (n) is added by starting at the root node (top level node) as the first “comparison node” and then using the following rules:

1. If the number we are to insert (n) is greater than the “comparison node”, move down the tree and to the right; the first node encountered is the new “comparison node”.
2. If the number we are to insert (n) is less than or equal to the “comparison node”, move down the tree and to the left; the first node encountered is the new “comparison node”.
3. If, after comparing at a node position, a node does not exist in the direction in which we try to move, then insert a new node containing the integer n at that position.



Creation of the *BSTNode* class:

We will now create a class call *BST* (binary search tree) that will allow us to add nodes in the fashion described above. However, we must first have a class that creates the nodes themselves. What information must a node contain? For storing integers in the nodes, as we are doing in this example, each node should contain the following:

1. The actual data (an integer for the example above)
2. A reference to the right-hand node immediately beneath this node (*null* if it doesn't exist)
3. A reference to the left-hand node immediately beneath this node (*null* if it doesn't exist)

We are going to call this node-creating class, *BstNode*. Its implementation is shown below.

```
public class BstNode
{
    public BstNode(int i) //Constructor
    {
        leftNode = null;
        rightNode = null;
        intData = i;
    }

    public int intData;
    public BstNode leftNode;
    public BstNode rightNode;
}
```

Notice that the three state variables in this class correspond to the three numbered requirements mentioned earlier. Also, notice that the *leftNode* and *rightNode* fields are set to *null* since when a node is constructed, there are no other nodes “hanging off it” yet.

The *BST* class:

Next, we turn our attention to the *BST* class itself. In the constructor, we simply create the root node (the highest level node).

The reader is strongly urged to look over [Appendix W](#) (Tree Definitions). There, you will get a firm grounding in the tree-terms we have already used here and new ones we will be using soon.

The constructor and state variables are as follows for the *BST* class:

```
public class BST
{
    public BST(int i) //constructor
    {
        // Root node is instantiated at the time of creation of the tree object.
        rootNode = new BstNode(i); //create a node with the above class
    }
    ...more code to come...

    BstNode rootNode;
}
```

The *addNode* method:

Now comes the most important (and most complex) method of the *BST* class, the *addNode* method in which decisions are made as to the correct position for each new node to be added. Here are the rules for inserting a new node after first setting the root

node as the *currentNode*.

1. If the number we are to insert (n) is greater than the *currentNode*, move down the tree and to the right; the first node encountered is the new *currentNode*.
2. If the number we are to insert (n) is less than or equal to the *currentNode*, move down the tree and to the left; the first node encountered is the new *currentNode*.
3. Continuing in this same fashion, move down the tree. If, after comparing at a node position, a node does not exist in the direction in which we try to move, then insert a new node containing the integer n at that position.

If these rules seem familiar, they are essentially the same as those at the top of page 52-1. Here, in this latest rendition, we become more specific with regard to the variable names to be used in the method that implements the rules. The complete class (including the *addNode* method) now reads:

```
public class BST
{
    public BST(int i) //constructor: Root node added at the time of creation of the tree
    {
        rootNode = new BstNode(i);
    }

    public void addNode(int i)
    {
        BstNode currentNode = rootNode;
        boolean finished = false;
        do
        {
            BstNode curLeftNode = currentNode.leftNode;
            BstNode curRightNode = currentNode.rightNode;
            int curIntData = currentNode.intData;

            if(i > curIntData) //look down the right branch
            {
                if(curRightNode == null)
                { //create a new node referenced with currentNode.rightNode
                    currentNode.rightNode = new BstNode(i);
                    finished = true;
                }
                else //keep looking by assigning a new current node one level down
                { currentNode = currentNode.rightNode; }
            }
            else //look down the left branch
            {
                if(curLeftNode == null)
                { //create a new node referenced with currentNode.leftNode
                    currentNode.leftNode = new BstNode(i);
                    finished = true;
                }
                else
                { //keep looking by assigning a new current node one level down
```

```

        currentNode = currentNode.leftNode;
    }
}
}while(!finished);
}
BstNode rootNode;
}

```

It is left to the reader to examine the code in the *addNode* method and to convince himself that the three numbered rules are being implemented.

A class for testing:

To test the *BST* class, use the following *Tester* class:

```

public class Tester
{
    public static void main(String args[])
    {
        //the first integer in the tree is used to create the object
        BST bstObj = new BST(50);
        bstObj.addNode(56);
        bstObj.addNode(52);
        bstObj.addNode(25);
        bstObj.addNode(74);
        bstObj.addNode(54);
    }
}

```

Prove that it really works:

The integers mentioned in the example at the beginning of this lesson are added to the tree with this test code. But how do we really know that it is working? What we need is some type of printout. If we add the following *traverseAndPrint* method to the *BST* class we will see that our class does, indeed, perform as advertised.

```

public void traverseAndPrint(BstNode currentNode )
{
    System.out.print("data = " + currentNode.intData);
    //To aid in your understanding, you may want to just ignore this
    //indented portion and just print the integer. In that case, change the
    //line above to a println instead of a print.
    if(currentNode.leftNode == null)
    {   System.out.print(" left = null");   }
    else
    {   System.out.print(" left = " + (currentNode.leftNode).intData);   }

    if(currentNode.rightNode == null)
    {   System.out.print(" right = null");   }
    else
    {   System.out.print(" right = " + (currentNode.rightNode).intData);}
    System.out.println("");

    if(currentNode.leftNode != null)
        traverseAndPrint(currentNode.leftNode);
}

```

```

        if(currentNode.rightNode != null)
            traverseAndPrint(currentNode.rightNode);
    }

```

This method has recursive calls to itself and will print every node in the tree. In addition to the data stored in each node (an integer), it also prints the contents of the two nodes “hanging off” this node.

Test this new method by adding the following code at the bottom of the *main* method in the *Tester* class.

```

//print all the nodes
bstObj.traverseAndPrint(bstObj.rootNode);

```

For the data suggested in the examples on page 52-1 the output will appear as shown below when the *main* method in the *Tester* class is executed:

```

data = 50      left = 25      right = 56
data = 25      left = null    right = null
data = 56      left = 52      right = 74
data = 52      left = null    right = 54
data = 54      left = null    right = null
data = 74      left = null    right = null

```

Project... *BST find* Method

Now we come to what a binary search tree is all about, the search. You are to create a method of the *BST* class called *find*. Its signature is as follows:

```

public boolean find(int i)

```

It returns a *true* if *i* is found in the binary tree and *false* if it's not found. This method will use essentially the same rules as those for the *addNode* method except when we come to the place where we formerly added a node; we will exit the method and say that the search was unsuccessful. Likewise, there is more to the comparisons. We can no longer just test to see if the data we are searching for is greater than or less than that of the *currentNode*. We must now also test for equality.

To test the *find* method, add the following code to the bottom of the *main* method in *Tester*.

```

System.out.println(bstObj.find(74)); //This is one it will find...prints a true
System.out.println(bstObj.find(13)); //This is one it won't find...prints a false

```

Why use a Binary Search Tree?

What can searching a Binary Search Tree (BST) do that we could not accomplish searching a linear array? The BST can do it faster, much faster. The Big O value for a

reasonably balanced BST is $O(\log n)$. For an unordered array it's $O(n)$; however, for an ordered array, a binary search is also of the order $O(\log n)$. So, what are the advantages of a binary search tree over searching an ordered array (using a binary search) since their Big O values are the same? The advantages are:

1. Using a binary search on an array, **ordering is necessary** after the insertion of **each new element**. An alternative to this is **inserting** the new element in the correct position. In either case, the time required to do this is typically considerably more than the time required to insert a new node in a *BST*.
2. In an array, we must pre-dimension the array.
 - a. If we dimension to small, we run the risk of running out of space if more nodes need to be added than were originally anticipated.
 - b. If we dimension to large, we waste memory and may degrade the performance of the computer.

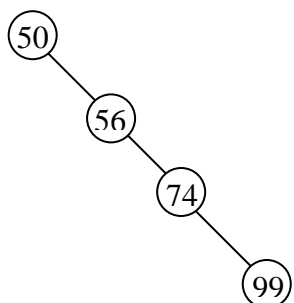
With the *BST* object, we dynamically create nodes as we need them in dynamic memory. There is no need to know ahead of time how many there will eventually be.

Anonymous objects:

Have you noticed that with the *BST* class, the **node objects that contain our data are not named** (except for the root node)? We have to traverse the tree and each node we encounter gives references to the two nodes “hanging off” it with *leftNode* and *rightNode*. Recall that we had a similar situation in [Lesson 49](#) with the singly linked list in which we had a “chain” of nodes, each with a reference to the next. Here, each node has references to **two** nodes that follow it.

Balanced Tree:

Above it was mentioned that the Big O value for searching a Binary Search Tree was $O(\log n)$ if the tree was reasonably balanced. What do we mean by a balanced tree? Refer back to [Fig 52-6](#) and it can be seen that this tree is not balanced. There are more nodes to the right of the root (50) than to the left. An extreme case of this is shown below.



[Fig 52-7](#) A totally unbalanced “tree”

Consider the following sequence of numbers to be added to a binary tree.

{50, 56, 74, 99}

The resulting “tree” to the left is totally unbalanced. Every new node to be added lies to the right of the previous node. In this case (which is clearly the worst case) the Big O value for searching the tree is $O(n)$. If there are n items we might very well have to do n comparisons before finding the desired one.

If we are **very** unlucky, just such a tree **might** result when we add our nodes. With random data, it is not very likely to be as bad as [Fig 52-7](#); however, what is more likely, is that that tree be somewhat out of balance which would, of course, reduce the efficiency of the search. What can we do to prevent this unbalance? It is beyond the scope of this

book, however, there are algorithms that can detect this and “rebalance the tree”. Nothing comes free, and this rebalancing adds complexity to the code as well as additional processing time.

Generalizing, Using Objects for Data:

It is possible to modify our class so that instead of just storing primitive integers we could store objects. To do this we would replace the code everywhere we pass an *int* variable as an argument, with *Comparable obj*.

The only catch is that the *obj* object that we pass in, **must implement the *compareTo* method**. The other requirement is that the former state variable, *int intData* be replaced with *Comparable data*. Rather than modify the *BST* class that we have already done, we are going to present another class that adds *Comparable* type objects to nodes in a Binary Search Tree. This class is just about the ultimate as far as brevity of code is concerned; however it is more difficult to understand because it uses recursion.

```
public class BsTree
{
    public BsTree(Comparable d)
    {
        theData = d;
        leftNode = null;    //This and next line could be omitted,
        rightNode = null;  //they are automatically null.
    }

    public BsTree addNode(Comparable d)
    {
        if(d.compareTo(theData) > 0)
        { //d should be inserted somewhere in the branch to the right
            if(rightNode != null)
                //right node exists, go down that branch, look for place to put it
                rightNode.addNode(d);
            else
                rightNode = new BsTree(d); //Create new rightNode, store d in it
        }
        else
        { //d should be inserted somewhere in the branch to the left
            if(leftNode != null)
                //left node exists, go down that branch, look for place to put it
                leftNode.addNode(d);
            else
                leftNode = new BsTree(d); //Create a new leftNode, store d in it
        }
        return this;
    }
    private Comparable theData;
    private BsTree leftNode, rightNode;
}
```

It is left to the reader to create a *find* method comparable to those of the *BST* class earlier in this lesson. We also need a *traverseAndPrint* method for this class. Three different versions of *traverseAndPrint* will be offered below as the various types of traversals are discussed.

Traversal types:

There are four traversal types. They are preorder, inorder, postorder, and level order traversals. Each visits **all** of the nodes in the tree, but each in a different order.

Preorder traversal of a Binary Search Tree:

Order of visitation of nodes: **50, 25, 18, 7, 19, 35, 30, 37, 76, 61, 56, 68, 80, 78, 85**

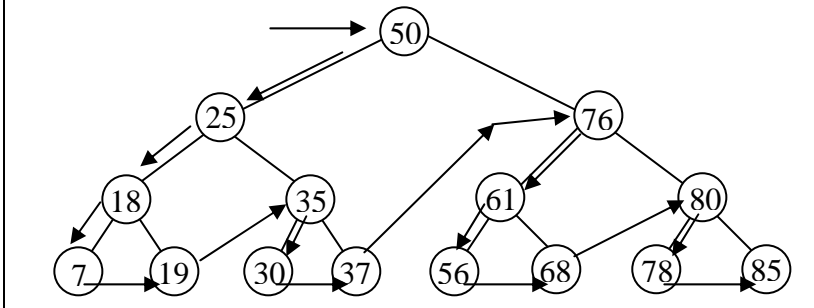


Fig. 52-8

Preorder traversal follows the sequence of arrows. **Rule: A node is visited before its descendants.**

The following code implements a preorder traversal of a tree as depicted in Fig. 52-8. An easy way to remember this code is to note the printing for this preorder traversal comes before the two recursive calls.

```
public void traverseAndPrint() //Use with BsTree class on previous page.
{
    System.out.println(theData);
    if( leftNode != null ) leftNode.traverseAndPrint();
    if( rightNode != null ) rightNode.traverseAndPrint();
}
```

Inorder traversal of a Binary Search Tree:

Order of visitation of nodes: **7, 18, 19, 25, 30, 35, 37, 50, 56, 61, 68, 76, 78, 80, 85**

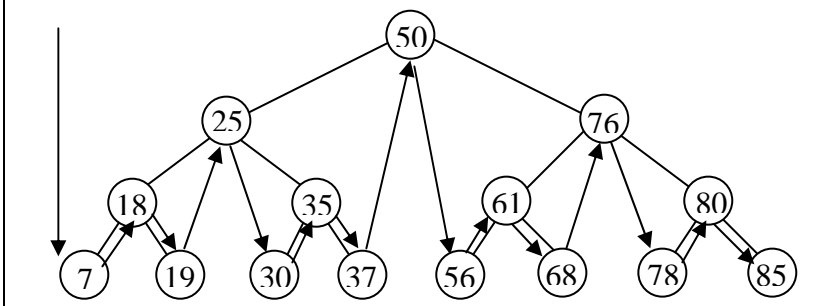


Fig. 52-9

Inorder traversal follows the sequence of arrows. The order is the ascending order of a sorted list. **Rule: A node is visited after its left subtree and before its right subtree.**

The following code implements an inorder traversal of a tree as depicted in Fig. 52-9. This technique is important since it **visits the nodes in a “sorted order.”** An easy way to remember this code is to note the printing for this inorder traversal comes in-between the two recursive calls.

```
public void traverseAndPrint()
{
```



```

    if( leftNode != null ) leftNode.traverseAndPrint( );
    System.out.println(theData);
    if( rightNode != null ) rightNode.traverseAndPrint( );
}
//Exchanging the first and last lines of this method results in a reverse-order traversal.

```

Postorder traversal of a Binary Search Tree:

Order of visitation of nodes: **7, 19, 18, 30, 37, 35, 25, 56, 68, 61, 78, 85, 80, 76, 50**

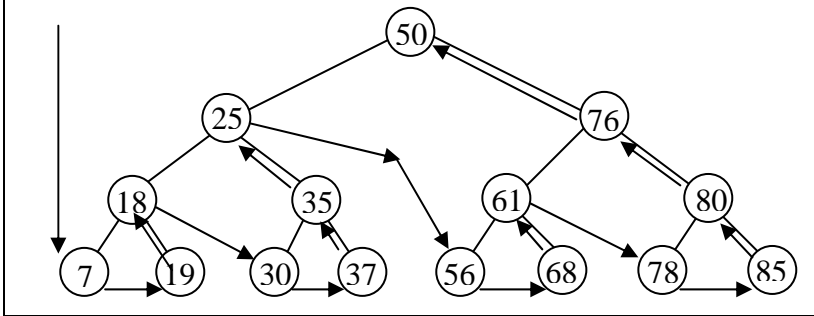


Fig. 52-10

Postorder traversal follows the sequence of arrows. **Rule: A node is visited after its descendants.**

The following code implements a postorder traversal of a tree as depicted in [Fig. 52-10](#). An easy way to remember this code is to note the printing for this postorder traversal comes after the two recursive calls.

```

public void traverseAndPrint( )
{
    if( leftNode != null ) leftNode.traverseAndPrint( );
    if( rightNode != null ) rightNode.traverseAndPrint( );
    System.out.println(theData);
}

```

Level order traversal of a Binary Search Tree:

Order of visitation of nodes: **50, 25, 76, 18, 35, 61, 80, 7, 19, 30, 37, 56, 68, 78, 85**

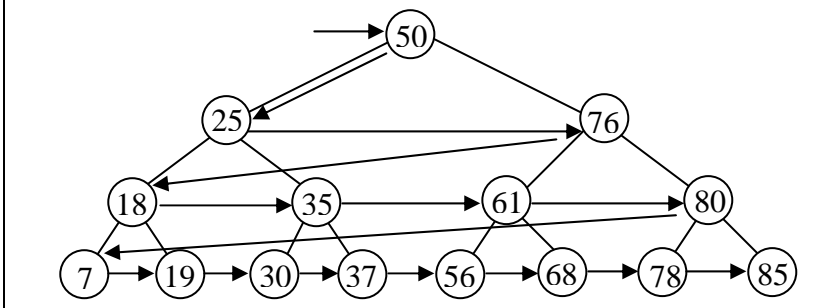


Fig. 52-11

Level order traversal follows the sequence of arrows.

The code that would implement this is a bit more involved than the others. One way to do it is to have counters that keep up with how deep we are in the tree.

An Application of Binary Trees... Binary Expression Trees

Consider the infix expressions $(6 + 8) * 2$ and $5 + (3 * 4)$. The expression trees to the right are a result of parsing these expressions. As can be inferred from the drawings, the following rules apply for an expression tree:

- Each leaf node contains a single operand.
- Each interior node contains an operator.
- The left and right subtrees of an operator node represent subexpressions that must be evaluated **before** applying the operator at the operator node.
 - The levels of the nodes in the tree indicate their relative precedence of evaluation.
 - Operations at the lower levels must be done **before** those above them.
 - The operation at the root of the tree will be the last to be done.

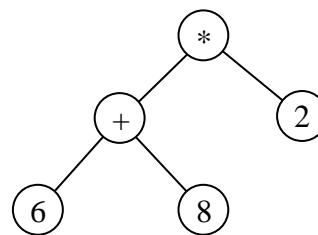


Fig. 52-12 $(6 + 8) * 2$

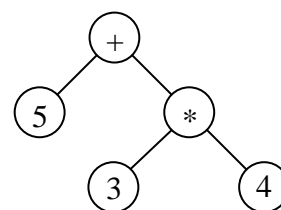


Fig 52-13 $5 + (3 * 4)$

We will now look at a larger expression tree and see how the inorder, preorder, and postorder traversals of the tree have special meanings with regard to the mathematics of an expression.

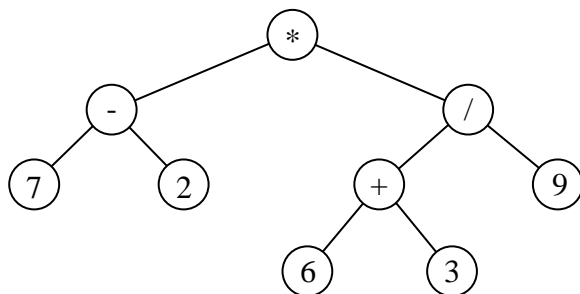


Fig. 52-14 A binary expression tree for the infix expression $(7 - 2) * ((6 + 3) / 9)$

An **Inorder Traversal** of the above expression tree yields the **infix** form: $(7 - 2) * ((6 + 3) / 9)$

A **Preorder Traversal** of the above expression tree yields the **prefix** form: $* - 7 2 / + 6 3 9$

A **Postorder Traversal** of the above expression tree yields the **postfix** form: $7 2 - 6 3 + 9 / *$

Notice that the postfix form is Reverse Polish Notation (RPN), the form that was used for the stack calculator of [Lesson 50](#).

Binary Search Tree... Contest Type Problems

<p>1. Which of the following replaces <*1> in the code to the right to make the <i>traverseAndPrint</i> method visit and print every node in a “Postorder” fashion?</p> <p>A. <code>if(leftNd != null) leftNd.traverseAndPrnt(); System.out.print(info); if(rightNd!=null) rightNd.traverseAndPrnt();</code></p> <p>B. <code>if(leftNd != null) leftNd.traverseAndPrnt(); if(rightNd!=null) rightNd.traverseAndPrnt(); System.out.print(info);</code></p> <p>C. <code>System.out.print(info); if(leftNd != null) leftNd.traverseAndPrnt(); if(rightNd!=null)rightNd.traverseAndPrnt();</code></p> <p>D. <code>leftNd.traverseAndPrnt(); rightNd.traverseandPrnt();</code></p> <p>E. None of these</p>	<pre>//Binary Search Tree public class Bst { public Bst(Comparable addValue) { info = addValue; } public Bst addNd(Comparable addValue) { int cmp = info.compareTo(addValue); if(cmp<0) { if(rightNd!=null) rightNd.addNd(addValue); else rightNd=new Bst(addValue); } else if(cmp>0) { if(leftNd!=null) leftNd.addNd(addValue); else leftNd=new Bst(addValue); } return this; } public void traverseAndPrnt() { <*1> } private Comparable info; private Bst leftNd; private Bst rightNd; }</pre>
<p>2. Assume <*1> has been filled in correctly. Which of the following creates a <i>Bst</i> object <i>obj</i> and adds 55 as a wrapper class <i>Integer</i>?</p> <p>A. <code>Integer J; J = 55; Bst obj = Bst(J);</code></p> <p>B. <code>Bst obj = new Bst(new Integer(55));</code></p> <p>C. <code>Bst obj; obj.addNd(55);</code></p> <p>D. <code>Bst obj; obj.addNd(new Integer(55));</code></p> <p>E. None of these</p>	
<p>3. Assume <*1> has been filled in correctly and that <i>n</i> objects are added to an object of type <i>Bst</i> in order from largest to smallest. What is the Big O value for searching this tree?</p> <p>A. $O(n \log n)$</p> <p>B. $O(\log n)$</p> <p>C. $O(n)$</p> <p>D. $O(n^2)$</p> <p>E. None of these</p>	

4. When a *Bst* object is constructed, to what value will *leftNd* and *rightNd* be initialized?

- A. this
- B. 0
- C. null
- D. *Bst* object
- E. None of these

5. After executing the code below, what does the resulting tree look like?

```
Bst obj = new Bst(new Integer(11));
obj.add(new Integer(6))
obj.add(new Integer(13));
```

A. ArithmeticException



E. None of these

6. What replaces `<*1>` in the code to the right so that a “Preorder” traversal is done?

- A. `if(leftNd != null) leftNd.traverseAndPrint();`
`System.out.print(info);`
`if(rightNd!=null)rightNd.traverseAndPrint();`
- B. `if(leftNd != null) leftNd.traverseAndPrint();`
`if(rightNd!=null)rightNd.traverseAndPrint();`
`System.out.print(info);`
- C. `System.out.print(info);`
`if(leftNd != null) leftNd.traverseAndPrint();`
`if(rightNd!=null)rightNd.traverseAndPrint();`
- D. `leftNd.traverseAndPrint();`
`rightNd.traverseandPrint();`
- E. None of these

//Binary Search Tree

```
public class Bst
{
    public Bst(Comparable addValue)
    {
        info = addValue;
    }

    public Bst addNd(Comparable addValue)
    {
        int cmp = info.compareTo(addValue);

        if(cmp<0)
        {
            if(rightNd!=null)
                rightNd.addNd(addValue);
            else
                rightNd=new Bst(addValue);
        }
        else if(cmp>0)
        {
            if(leftNd!=null)
                leftNd.addNd(addValue);
            else
                leftNd=new Bst(addValue);
        }
        return this;
    }

    public void transverseAndPrint( )
    {
        <*1>
    }
}
```

```
private Comparable info;
private Bst leftNd;
private Bst rightNd;
```

```
}
```

7. What is a disadvantage of an unbalanced Binary Search Tree?

- A. No disadvantage
 B. Uses excessive memory
 C. Limited accuracy
 D. Reduced search efficiency
 E. None of these

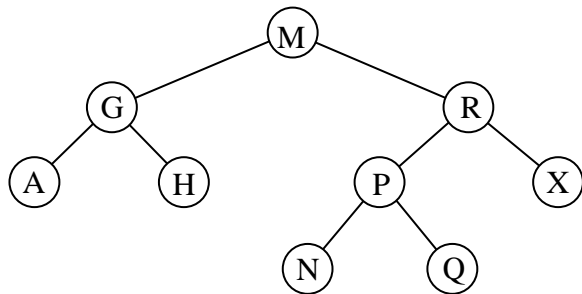
8. Average case search time for a Binary Search Tree that is reasonably balanced is of what order?

- A. $O(n \log n)$
 B. $O(n^2)$
 C. $O(n)$
 D. $O(1)$
 E. None of these

9. What positive thing(s) can be said about a completely unbalanced tree that results from adding the following integers to a tree in the sequence shown?

{ 5, 6, 7, ... 999, 1000 }

- A. The items are automatically in numerical order along the long sequential strand.
 B. The smallest number is automatically the root node.
 C. The largest number is the root node.
 D. Both A and B
 E. Both A and C



10. In what order are the nodes visited in the tree to the left if a preorder traversal is done?

- A. A, G, H, M, N, P, Q, R, X
 B. M, G, A, H, R, P, N, Q, X
 C. A, H, G, N, Q, P, X, R, M
 D. M, G, R, A, D, P, X, N, Q
 E. None of these

11. In what order are the nodes visited in the tree to the left if a postorder traversal is done?

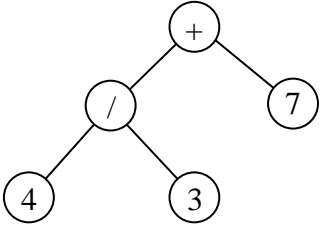
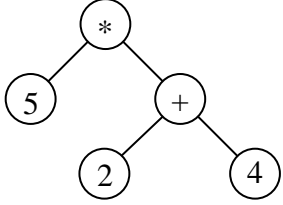
- A. A, G, H, M, N, P, Q, R, X
 B. M, G, A, H, R, P, N, Q, X
 C. A, H, G, N, Q, P, X, R, M
 D. M, G, R, A, H, P, X, N, Q
 E. None of these

12. In what order are the nodes visited in the tree to the left if an inorder traversal is done?

- A. A, G, H, M, N, P, Q, R, X
 B. M, G, A, H, R, P, N, Q, X
 C. A, H, G, N, Q, P, X, R, M
 D. M, G, R, A, H, P, X, N, Q
 E. None of these

13. For the tree above, which of the following is a possible order in which the nodes were originally added to the binary search tree?

- A. M, G, R, A, H, X, P, N, Q
 B. M, G, R, A, H, Q, N, P, X
 C. M, R, A, G, H, X, P, N, Q
 D. A, G, H, M, N, P, Q, R, X
 E. None of these

<p>14. What mathematical infix expression is represented by the binary expression tree to the right?</p> <p>A. $(4 + 3) / 7$ B. $4 / (3 + 7)$ C. $7 / 4 / 3 + 7$ D. $(4 / 3) + 7$ E. None of these</p>	
<p>15. What mathematical infix expression is represented by the binary expression tree to the right?</p> <p>A. $5 * 2 + 4$ B. $5 * (2 + 4)$ C. $(2 * 4) + 5$ D. $5 * 2 * (+4)$ E. None of these</p>	
<p>16. Which of the following is a postfix version of the following mathematical expression?</p> $(37 - 59) * ((4 + 1) / 6)$ <p>A. $* - 37 59 / + 4 1 6$ B. $(37 - 59) * ((4 + 1) / 6)$ C. $37 59 - 4 1 + 6 / *$ D. $37 - 59 * 4 + 1 / 6$ E. None of these</p>	
<p>17. What is the minimum number of levels for a binary tree with 20 nodes?</p> <p>A. 20 B. 7 C. 6 D. 5 E. None of these</p>	
<p>18. What is the maximum number of levels for a binary tree with 20 nodes?</p> <p>A. 20 B. 7 C. 6 D. 5 E. None of these</p>	