Common Binary Tree Traversals:

Preorder

//as with General Trees

Postorder

//as with General Trees

Inorder

//Specific to Binary Trees

Euler Tour

// Specific to Binary Trees

Preorder Binary Tree Traversal Algorithms

The preorder traversal algorithm used with the general Tree class can be used with a binary tree. However, a preorder traversal algorithm specific to a binary tree is mostly used, employing such methods as **left(v)** and **right(v).**

The following algorithm prints all the nodes, one per line, of the subtree of the node at **v** in the binary tree **bt1.**

```
public static void preorderPrintLines (BinaryTree   bt1,
                                       BNode        v) {
    String s = bt1.elementAt(v).toString();
                //the class for the element in the node
                //needs to have a toString() method
    System.out.println (s);   // subtree root element printed

    if (bt1.isInternal(v))   {
        BNode       p;
        if (bt1.hasLeft(v)) {
                p  = bt1.left(v);
                preorderPrintLines (bt1, p ); }
            //We have just traversed tree of left child
        if (bt1.hasRight(v)) {
                p  = bt1.right(v);
                preorderPrintLines (bt1, p );  }
          //We have just traversed tree of right child
        } //end if
    }
```

Postorder Binary Tree Traversal Algorithms

The postorder traversal algorithm used with the general Tree class can be used with a binary tree. However, a postorder traversal algorithm specific to a binary tree is mostly used, with such methods as **left(v)** and **right(v).**

The following algorithm prints all the nodes, one per line, of the subtree of the node at **v** in the binary tree **bt1.**

```
public static void postorderPrintLines (BinaryTree  bt1,
                                        BNode       v) {
    String      s;
    if (bt1.isInternal(v))   {
        BNode      p;

        if (bt1.hasLeft(v)) {
                p  = bt1.left(v);
                postorderPrintLines (bt1, p);    }
        if (bt1.hasRight(v)) {
                p  = bt1.right(v);
                postorderPrintLines (bt1, p);  }
         // all children of node at v processed
        }  //endif

      //now print the parent of the children processed
      s = bt1.elementAt(v).toString();
                  //the class for the element in the node
                  //needs to have a toString() method
     System.out.println (s); //subtree root element output
  }
```

Expression trees -- Binary Production Trees

Consider the following expression tree for:

((5 * 10) / 2) - ((2 + 3) + 6)

```
                      -

               /                       +

          *          2           +          6

       5     10              2     3
```

The **postorder** traversal is:

5 10 * 2 / 2 3 + 6 + -

Putting in parenthesis, we get a **postfix** expression:

(((5 10) * 2) / ((2 3) + 6)+)-

which evaluates left to right to:

((50 2)/ (5 6)+)-

which evaluates left to right to:

(25 11)- = 14

Binary Expression Tree Evaluation Program

*[Assumes **object** values of **Double** or **Character** in nodes]*

```
public static double expressEval (BinaryTree      bt1,
                                  BNode           v) {
    double      result,  x = 0, y = 0;
    BNode      p;
     if (bt1.internal(v)) {
          p  =  bt1.left(v);
          x = /*double*/expressEval(bt1, p);
                    // left child pyramid evaluated
          p  =  bt1.right(v);
          y = /*double*/expressEval(bt1, p);
                    // right child pyramid evaluated
   //now operate on x and y using the operator found
   // in the parent node.
         Character  cC;
         cC = (Character) bt1.elementAt(v);  //cast
         char c = cC.charValue();   //unwrapping char
         result = arithmeticOp(c, x, y) ;
              //uses c value  to carry out +,-,*, or / on x, y
     } //end if-then
    else  {
  //if the parent was a leaf, it must hold an operand
 //so return that instead  but watch out for object type
      Double     resultD;      //type Double, not type double
      resultD = (Double) bt1.elementAt(v);  //cast
      result = resultD.doubleValue();  //unwrapping    }
  //end if-then-else
  if (t1.isRoot(v) { //i.e entire tree has now been processed
         System.out.println ( "Evaluates to :   "  + result);  }
   return result;  }
```

Essentials of static method arithmeticOp()

```
public static double arithmeticOp (char      c,
                                   double    x, y) {
        double      r;

        if (c == '+')  r = x + y;
        if (c == '-')   r = x - y;
        if (c == '*')  r = x * y;
        if (c == '/')   r = x / y;

         return r;
}
```

Pseudocode Version of expressEval (bt1, v)

Algorithm expressEval (bt1, v);   **returns numeric value**;

if (bt1.isInternal(v)) **then** {

x < -- expressEval(bt1, bt1.left(v));

//left child visited

y < -- expressEval(bt1, bt1.right(v));

//right child visited

c <-- *character value of:* bt1.elementAt(v);

//parent visited

r < -- (x <op c> y) ; }

else

r < -- double value of: bt1.elementAt(v);

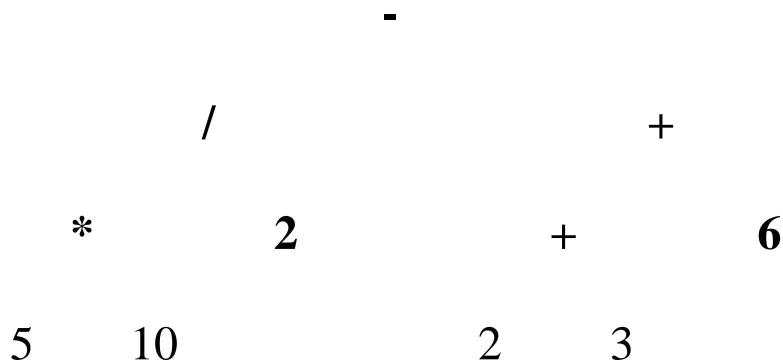
if (bt1.isRoot(v)) print out: "Result: ", r;

return r }

Inorder Traversals

With an inorder traversal, you visit the left child, then the parent, and then the right child, etc.

Consider the following expression tree:



An inorder traversal traces out the conventional algebraic presentation of an expression in an expression tree:

5 * 10 / 2   - 2 + 3 + 6

If we put in parentheses:

((5 * 10) / 2) - ((2 + 3) + 6)

Inorder traversal to return and print an arithmetic expression

```
public static  String inorderPrint (BinaryTree  bt1,
                                     BNode        v) {
     String        s  = null;

     BNode       p;

     //concat left child to string s
     if (bt1.hasLeft(v)) {
             p  = bt1.left(v);
             s  += inorderPrint (bt1, p);    }

     //now concat parent to string s
     s += bt1.elementAt(v).toString();
                   //the class for the element in the node
                   //needs to have a toString() method

    //now concat  right child to string s
    if (bt1.hasRight(v)) {
             p  = bt1.right(v);
             s += inorderPrint (bt1, p);  }
      // all children of node at v processed

  if ( bt1.isRoot(v) )
       System.out.println ("Expression:  "  + s);

  // Note: no parentheses added
  return s;  }
```

Binary search trees and inorder traversals

Suppose we have a set of elements in a definite order, for example:

B D F K M N R T W Y *// alphabetical*

06 17 21 44 55 80 87 92 *//arithmetic*

H He Li Be B C N O F Ne Na //Atomic number

We can place such elements in a ***binary search tree*** as follows.

One element, preferably from the middle of the set, but not necessarily from the middle, is in the root node.

The values in the left subtree are all less than or equal to the value in the root node.

The values in the right subtree are all greater than or equal to the value in the root node.

More generally, for any node v:
The values in the left subtree are all less than or equal to the value in the parent node v.

The values in the right subtree are all greater than or equal to the value in the parent node.

The leaf nodes, by definition, do **NOT** store an element

Examples of binary search trees, based on the set

06 17 21 44 55 80 87 92

```
Example_1            44

       17                          87

   06       21             55             92

 -    -    -    -        -    80        -    -

                              -   -
```

```
Example_2                55

        17                               92

    06         44                 80               -

  -    -     21    -          -        87

              -   -                  -    -
```

Notice that an inorder traversal of either tree will generate the original set in correct order:

06 17 21 44 55 80 87 92

Searching and the Binary Search Tree

The binary tree search algorithm is simple in principle.

Suppose we are using the top tree above and are searching for the element with target value 80.

1. Start at the root node.

2. Investigate node

If node value = target, search successful, stop

If node is a leaf, search unsuccessful, stop
//note: in this case you must go to
//the bottom empty leaf

If node value > target,
investigate left child node (repeat step 2)

If node value < target,
investigate right child node (repeat step 2)