CS 3410 – Ch 18 – Trees

Sections	Pages
18.1-18.4	651-680

All code in these notes can be found in the HW 3 download.

18.1 Trees

1. A non-recursive definition of a tree is that it consists of a set of *nodes* which are connected by a set of edges.

2. In this course, we will consider a *rooted tree* which has the following properties:

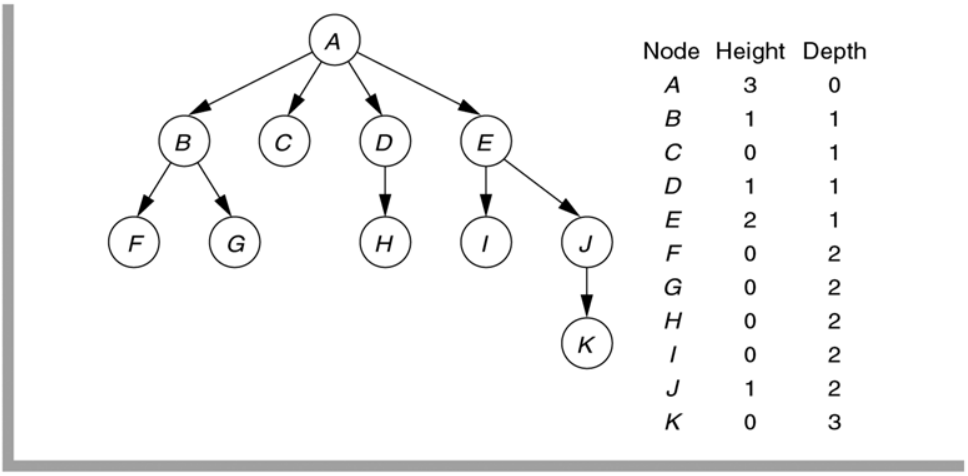
   a. One node is distinguished as the *root*.
   b. Every node, c, except the root, is connected by an *edge* from exactly one other node, p. Node p is $c's$ *parent* and c is one of $p's$ *children. c* is *p's descendant*, and *p* is *c's ancestor*.
   c. A unique path traverses from the root node to each node. The number of edges that must be followed to that node is the *path length*.

3. Other definitions:

   a. *leaf* – A node that has no children
   b. *depth of a node* – the length of the path from the root to the node.
   c. *height of a node* – the length of the path from the node to the deepest leaf.
   d. *height of a tree* – the height of the root
   e. *size of a node* – the number of descendants a node has, including itself.
   f. *size of a tree* – the size of the root.

figure 18.1

A tree, with height and depth information



Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

4. A recursive definition of a tree is: either a tree is empty or it consists of a root node with zero or more subtrees, each of whose roots are connected by an edge from the root.

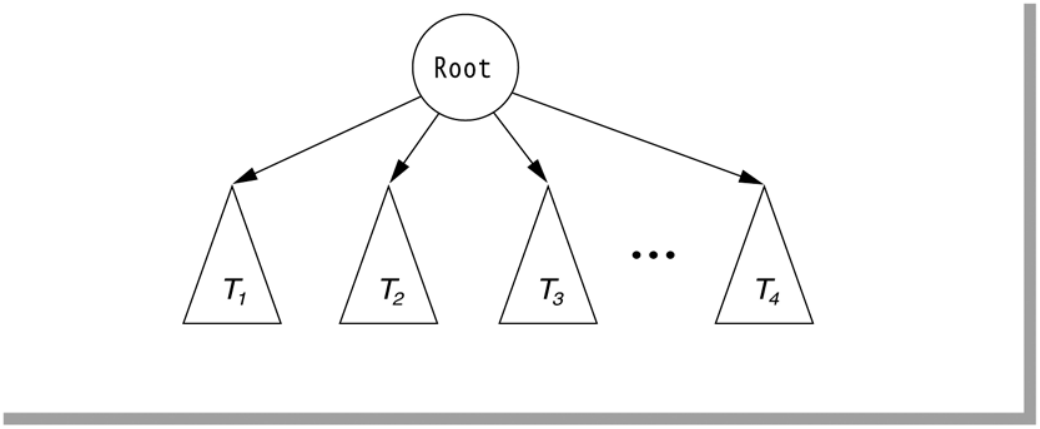


figure 18.2
A tree viewed recursively

5. A UML representation of a tree:

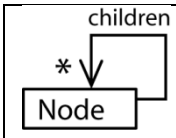
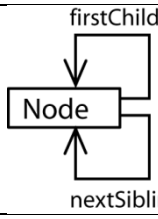
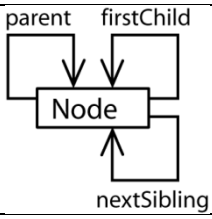
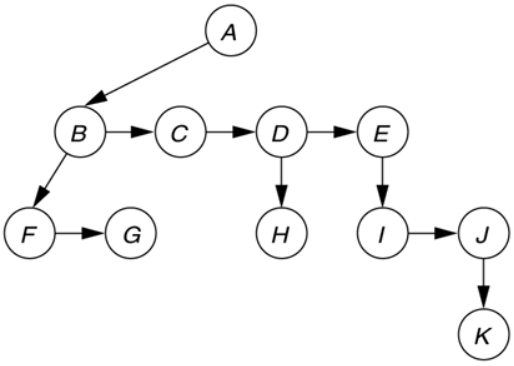
		
A node with a collection of children. As an example, see Figure 18.1 above.	A node with a link to the *firstChild* and a link to the *nextSibling*. See Figure 18.3 below.	Shows a link back to the parent.

figure 18.3
First child/next sibling representation of the tree in Figure 18.1



Homework 18.1

1. Define a *tree* as used in computer science recursively and non-recursively.
2. Define the terms: leaf, depth of a node, height of a node, height of a tree, size of a node, size of a tree.
3. Draw and explain a UML representation of a tree.
4. Problem 18.1, p.682
5. Problem 18.3, p.682
6. Problem 18.3, p.682

18.2 Binary Trees

1. A binary tree is a tree with no more than two children. We call these children, left and right. A binary tree has many uses in computer science.

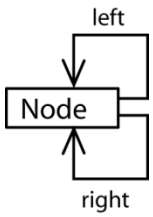
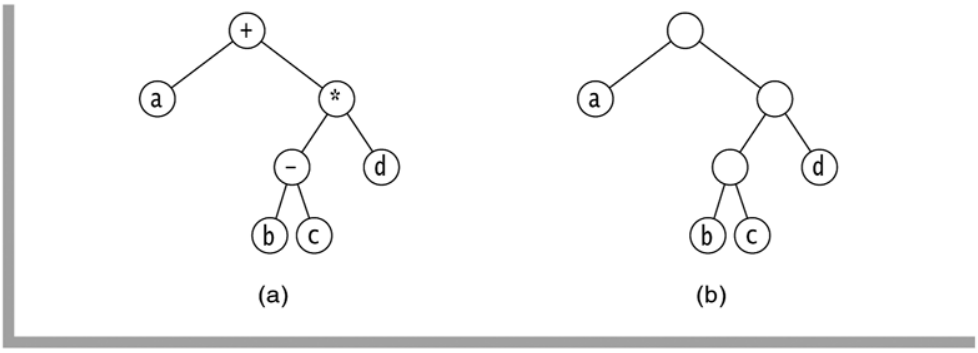


figure 18.11

Uses of binary trees: (a) an expression tree and (b) a Huffman coding tree



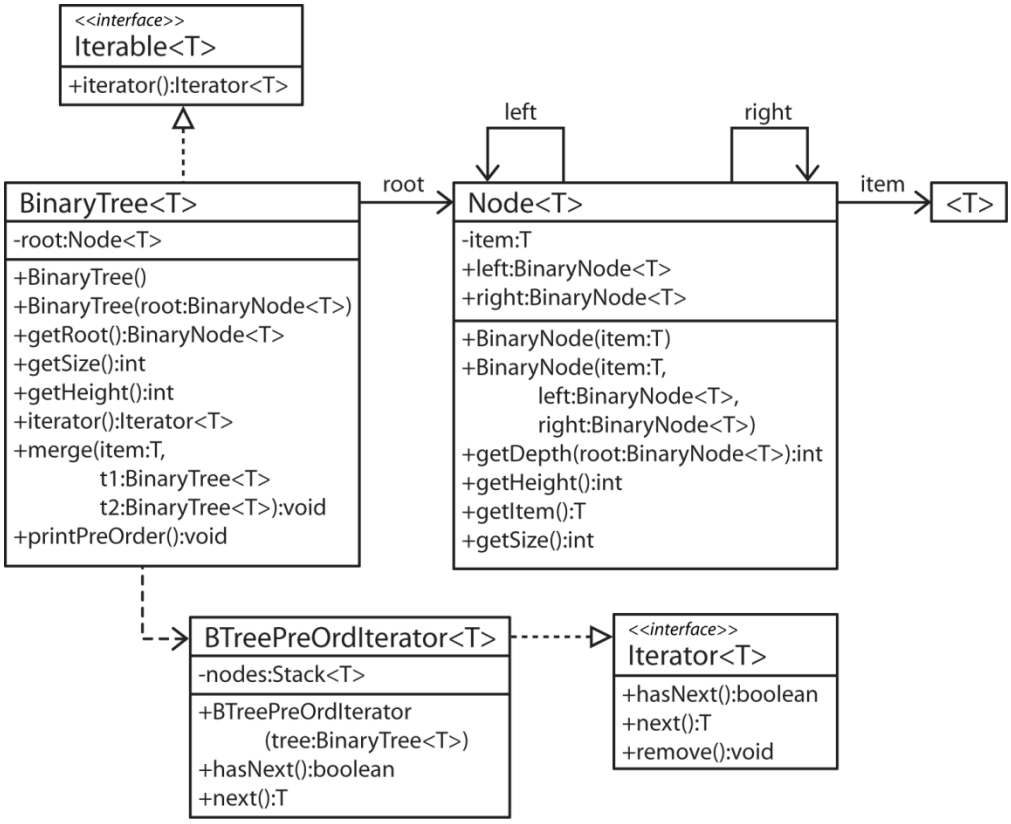
Homework 18.2

1. Problem 18.5, p.683
2. Problem 18.6, p.683
3. Problem 18.7a, p.683
4. Define a *binary tree*.
5. Draw and explain a UML representation of a simple binary tree.

18.2 Binary Tree Data Structure

The text defines a binary tree data structure and shows the implementation of it. Here, we will design our own, somewhat different, and implement it. We do this to emphasize that there are different ways to implement certain operations and that a binary tree implementation is not completely rigid. In other words, in practice we design classes to support the need at hand.

Here, we provide a BinaryTree class that contains a reference to the root node of the tree. This class manages the binary tree. We also make our BinaryTree *Iterable* by implementing a pre-order iterator.

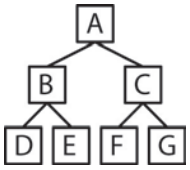


In practice, we would provide scope information for the classes. For instance, the BTreePreOrdIterator would be package level and perhaps an inner class. In the configuration above, the Node class must be known by any client that utilizes BinaryTree because a constructor and a method use the Node class. For instance, to create a tree (again, using the configuration above) you must essentially construct a "binary tree" using the BinaryNode(item) and BinaryNode(item, nodeLeft, nodeRight) constructors. Then, pass the "root" to the BinaryTree(root) constructor. The situation in the text is the same.

There are two major goals here: *Use*/*Create* a BinaryTree instance and *Write* a BinaryTree class (and supporting classes).

Homework 18.3

1. Draw and explain a UML representation of a binary tree data structure. Provide the common methods.
2. Write a snippet of code to produce a binary tree that looks as shown below. Use the API provided in the notes above.



18.2-18.4 – Size and Height of a Node

1. Consider finding the size of a node. Thus, we want the *Node* class to have a *getSize* method. The size of a node is 1 (for the node itself) plus the size of the left subtree plus the size of the right subtree. The public (probably protected in practice) method is:

```
getSize() { getSize( this ) }
```

   Notice that the public *getSize* calls a private (recursive) helper method, *getSize* and passes a reference to the node itself (*this*).

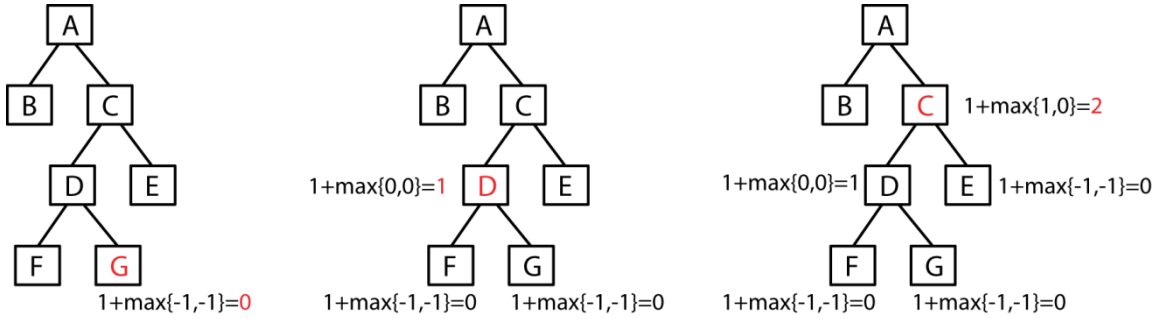
```
getSize( node )
{
      if( node == null )
            return 0;
      else
            return 1 + getSize( node.left ) + getSize( node.right );
}
```

2. Now, consider a public *getSize* method in the *BinaryTree* class. It can simply get the root node and call it's *getSize* method.

```
getSize() { root.getSize() }
```

3. We also want the *Node* class to have a method to get the height of a node. The height of a node is calculated by incrementing a counter each time we recurse, negating the count when we have gone too far. In other words, we count in advance of a recursive call, which may lead to a null node, which will require the negation of that count.

```
getHeight() { getHeight( this ) }

getHeight( node )
{
    if( node == null )
        return -1;
    else
        return 1 + Math.max( getHeight( node.left  ),
                             getHeight( node.right )   );
}
```



4. Similarly, we might want a public *getHeight* method in the *BinaryTree* class. It can simply get the root node and call it's *getHeight* method.

```
getHeight() { root.getHeight() }
```

Homework 18.4

1. Write methods for a BinaryNode class that return the size of a node.
2. Write methods for a BinaryTree class that finds the size of the tree.
3. Write methods for a BinaryNode class that return the height of a node.
4. Write methods for a BinaryTree class that finds the height of the tree.
5. Write a method for the BinaryTree class with signature:

   public int getDepth(T item)

   which will return the depth of the node that contains *item* or -1 if it doesn't exist. Hints: (a) What do you need to find the depth of an arbitrary node? (b) How do you find the node where *item* is? Answer: search left, search right, checking to see if you have hit *null* or found the *item*. (c) It is similar to the getHeight method, only you need to get the left and right heights before doing the max. If both left and right are -1, then you want to return -1 to signify that the item is not in that subtree. If left and right are not both -1, then you do the max. (d) when you find the item, return 0.

6. Problem 18.9, p.683
7. Problem 18.10ab, p.683

18.2-18.4 – BinaryTree Class Method – Merging Trees

1. The BinaryNode class has a constructor that creates a new node by joining an *item* with two existing nodes:

```
public BinaryNode( T item, BinaryNode<T> left, BinaryNode<T> right )
{
   this.item = item;
   this.left = left;
   this.right = right;
}
```

2. The BinaryTree class has a merge method that combines two binary trees with an item, where the item will be the root node of the tree as shown below. Essentially, it is a one-liner. However, we must take care.

```
public void merge( T item, BinaryTree<T> t1, BinaryTree<T> t2 )
{

   if( t1.root == t2.root && t1.root != null )  // Comment A
      throw new IllegalArgumentException();

   root = new BinaryNode<T>( item, t1.root, t2.root );

   if( this != t1 ) // Comment B
      t1.root = null;
   if( this != t2 )
      t2.root = null;
}
```

Comment A: Consider a call to merge: t.merge(x,t1,t1). We must prevent this (unless t1 is null) so that nodes are not shared between the left and right subtrees.

Comment B: Suppose we want to use: t.merge(x,t1,t2). At the conclusion of the method, nodes from t1 will be in two places: t and t1. In other words, the nodes are shared between two trees. To prevent this, we set: t1.root = null. The same situation exists for t2. However, we only do this when the reference object (this) is not t1. For instance, consider: t1.merge(x,t1,t2). If we set t1.root=null, then we would remove the root node from the tree.

18.2-18.4 – Binary Tree Traversals

1. Given a node, to *visit* (*process*) a node means to do something with that node (*e.g.* print the node's state, invoke a method on it, *etc.*).

2. Given a node (often the root), you can *Visit (Process)* all descendants by recursively branching.

3. A *traversal* of a BinaryTree means to *visit (process)* all the nodes. There are a number of orders in which we can visit the nodes:

4. Four common traversal methods:

   a. Pre-Order Traversal – visit node before branching

      ```
      traverse( n )
            visit( n )
            traverse( n.L )
            traverse( n.R )
      ```

   b. Post-Order Traversal – visit node after branching

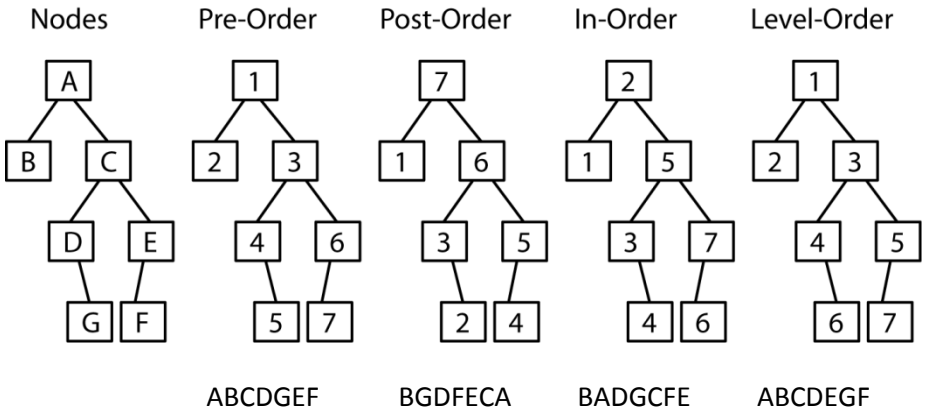
      ```
      traverse( n )
            traverse( n.L )
            traverse( n.R )
            visit( n )
      ```

   c. In-Order Traversal - visit node in between the two branchings

      ```
      traverse( n )
            traverse( n.L )
            visit( n )
            traverse( n.R )
      ```

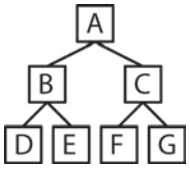
   d. Level-Order Traversal – Visit nodes in order of their depth, top-to-bottom, left-to-right. This will be discussed later.

5. Shown below are 4 common ways to traverse the nodes in binary tree:



Homework 18.5

1. State the order the nodes will be visited with the traversal is:
   a. pre-order
   b. in-order
   c. post-order
   d. level-order



18.2-18.4 – BinaryTree Class Methods – Traversals

Although not shown in the class diagram in for the BinaryTree class on page 4, these methods could exist there.

1. Pre-Order, Post-Order, In-Order implementations.

Pre-Order – A node is processed, then it's two children are processed, recursively. A binary tree might have a method like this:

```
preOrderTraversal()
{
   preOrderTraversal( root )
}

preOrderTraversal( node )
{
   process( node );
   if( node.left != null )
         preOrderTraversal( node.left );
   if( node.right != null )
         preOrderTraversal( node.right );
}
```

Post-Order – A nodes is processed after it's children are processed recursively

```
postOrderTraversal()
{
   postOrderTraversal( root )
}

postOrderTraversal( node )
{
   if( node.left != null )
      postOrderTraversal( node.left );
   if( node.right != null )
      postOrderTraversal( node.right );

   process( node );
}
```

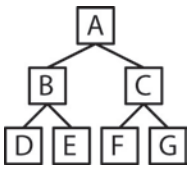
In-Order – The left child is recursively processed, then the current node is processed, and then the right child is recursively processed

```
inOrderTraversal()
{
   inOrderTraversal( root )
}

inOrderTraversal( node )
{
   if( node.left != null )
      inOrderTraversal( node.left );

   process( node );

   if( node.right != null )
      inOrderTraversal( node.right );
}
```

2. Level-Order – Nodes are visited top to bottom, left to right. In other words, the depth 0 node (root) is visited, then the depth 1 nodes, then the depth 2 nodes, *etc*. A queue is used to implement a level-order processing. The queue contains nodes that have not been visited yet. At each iteration, the node at the front of the queue is removed and then, its children are placed, left, then right, at the end of the queue.

```
levelOrderTraversal()
{
      LinkedList queue = new LinkedList();
      queue.offer( root );
      levelOrderTraversal( queue )
}

levelOrderTraversal( queue )
{
      if( !queue.isEmpty() )
      {
            node = queue.poll();
            process( node );

            if( node.left != null )
                  nodes.offer( node.left );

            if( node.right != null )
                  nodes.offer( node.right );

            if( !queue.isEmpty() )
                  levelOrderTraversal( queue );
      }
}
```

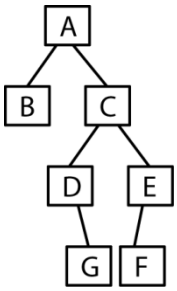
Consider this example:



	Queue	Print
Initial Contents	A	
poll A & process	B	
offer B & C	C	A
poll B & process	C	
offer D & E	D	
	E	B
poll C & process	D	
offer F & G	E	
	F	
	G	C
poll D & process	E	
(no children to offer)	F	
	G	D
poll E & process	F	
(no children to offer)	G	E
poll F & process	G	
(no children to offer)		F
poll G & process		
(no children to offer)		G

Homework 18.6

1. Write BinaryTree class methods to do pre-order, post-order, in-order, and level-order traversals
2. Consider the tree below. Show the contents of the queue at each step as the nodes are processed in Level order.



	Stack	Return
Initial Contents	A	
next()	B	
pop A, push C and B	C	
return A		A
next()	C	
pop B (no children to push)		
return B		B
next()	D	
pop C, push E and D	E	
return C		C
next()	G	
pop D, push G (no left child)	E	
return D		D
next()	E	
pop G (no children to push)		
return G		G
next()	F	
pop E, push F (no right child to push)		
return E		E
hasNext()		
pop F (no children to push)		
return F		F

18.4.3 – BinaryTree Pre-Order Iterator

1. The BinaryTree class shown previously implements the *Iterable* interface with the *iterator* method:

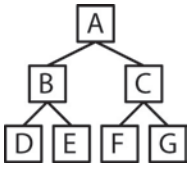
```
public Iterator<T> iterator()
{
   return new BTreePreOrdIterator<T>( this );
}
```

The iterator we define is a pre-order iterator. Remember that an iterator must keep track of the (pre) order of the nodes, so that when *next* is called, it can return the correct element. Remember, pre-order is: process n, recurse to the left, recurse to the right. In this case *process n* means *return n*. In other words, return *n* before returning any of its children. The mechanism we use to implement the pre-order *next* method is a Stack. We initialize the stack with the root node. Each time *next* is called, the stack is popped and the node is returned. However, before the return, we push the right child, then the left child.

```
next()
    pop n
    push n.right
    push n.left
    return n
```

We push the children, right then left, so that when the stack is popped, the left child comes out first.

2. Consider this example:



	Stack	Return
Initial Contents	A	
next()	B	
pop A, push C and B	C	
return A		A
next()	D	
pop B, push E and D	E	
return B	C	B
next()	E	
pop D (no children to push)	C	
return D		D
next()	C	
pop E (no children to push)		
return E		E
next()	F	
pop C, push G and F	G	
return C		C
next()	G	
pop F (no children to push)		
return F		F
hasNext()		
pop G (no children to push)		
return G		G

3. An implementation

```
public class BTreePreOrdIterator<T> implements Iterator<T>
{
   private Stack<BinaryNode<T>> nodes;

   public BTreePreOrdIterator( BinaryTree<T> tree )
   {
      nodes = new Stack<BinaryNode<T>>();
      nodes.push( tree.getRoot() );
   }

   public boolean hasNext() { return nodes.size()>0; }

   public T next()
   {
     BinaryNode<T> n;

      if( hasNext() )
      {
         n = nodes.pop();
         if( n.right != null )
            nodes.push( n.right );
         if( n.left != null )
            nodes.push( n.left );
         return n.getItem();
      }
      return null;
   }
   public void remove()
   {
      throw new UnsupportedOperationException();
   }
}
```

18.4.2 – BinaryTree In-Order Iterator

1. The in-order iterator is similar to the pre-order iterator, except that when a node is popped for the first time, it is not it's turn, it must wait until after his left child (afterall, this is in-order). So, when a node is popped for the first time, we push it back and then push his left child on top of him. When a node is popped for the second time, then we know his left child has been previously returned. Thus we return the node, but before doing so, we push the right child.

```
next()
    Loop
        pop n
        if count=1  // not your turn, left child goes first
            push n
            push n.left
        else if count=2  // now its your turn, but take care of your right child
            push n.right
            return n
```

2. To implement this iterator, each node is associated with a *CountingNode*, which keeps track of the number of times a node has been popped. Thus, a *CountingNode* wraps a *Node* and provides an attribute that counts the number of times it has been popped. We use a nested class to implement *CountingNode*. Finally, we use a stack of *Counting Nodes*, initialized with the *CountingNode* that wraps the root node.

```
public class BTreeInOrdIterator<T> implements Iterator<T>
{
   protected Stack<CountingNode<T>> nodes;

   private static class CountingNode<T>
   {
      protected CountingNode( BinaryNode<T> n )
      {
         node = n;
         timesPopped = 0;
      }
      protected BinaryNode<T> node;
      protected int timesPopped;
   }

   public BTreeInOrdIterator( BinaryTree<T> tree )
   {
      // Create stack.
      nodes = new Stack<CountingNode<T>>();
      // Push root node onto stack.
      if( tree.getRoot() != null )
         nodes.push( new CountingNode<T>( tree.getRoot() ) );
   }
```

```
public T next()
{
   BinaryNode<T> n;
   CountingNode<T> cn;

   if( hasNext() )
   {
      while( true )
      {
         // Pop stack.
         cn = nodes.pop();
         // If node popped for first time...
         if( ++cn.timesPopped == 1 )
         {
           // Push node back onto stack.
            nodes.push( cn );

            // If left child exists, push it onto stack.
            if( cn.node.left != null )
            {
               nodes.push( new CountingNode<T>( cn.node.left ) );
            }
         }
         // If node popped for second time...
         else if( cn.timesPopped == 2 )
         {
            // If right child exists, push it onto stack.
            if( cn.node.right != null )
            {
               nodes.push( new CountingNode<T>( cn.node.right ) );
            }
            // Return node.
            return cn.node.getItem();
         }
      }
   }
   return null;
}
```

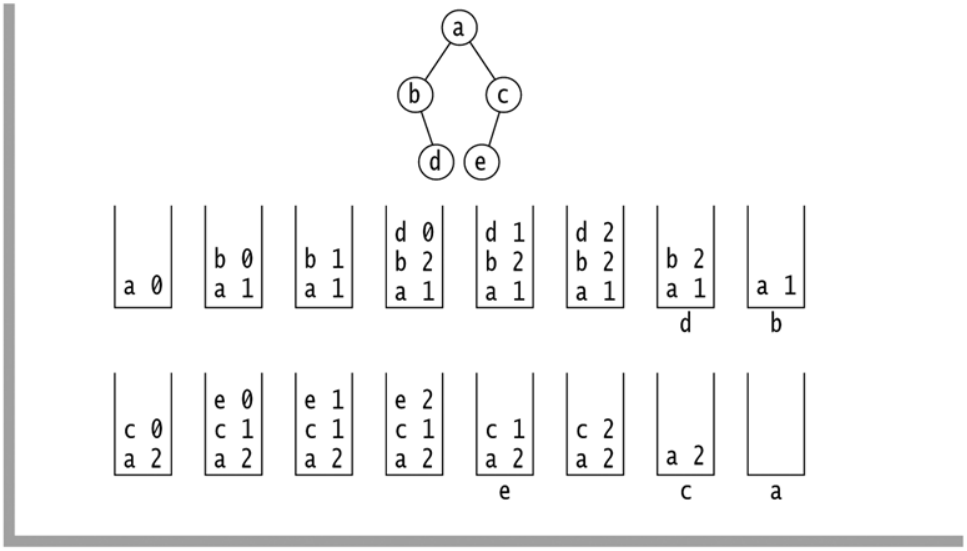
18.4.1 – BinaryTree Post-Order Iterator

1. Now, we consider a post-order iterator, which is similiar to the in-order iterator. However, a node will not be returned until it has been popped three times. The first time a node is popped, it is pushed back, along with his left child. The second time a node is popped, it is popped back with its right child.

```
next()
    Loop
        pop n
        if count=1  // not your turn, left child goes first
            push n
            push n.left
        else if count=2  // still not your turn, right child goes next
            push n
            push n.right
        else if count=3  // OK, you can go
            return n
```

figure 18.25

Stack states during postorder traversal



2. Here, we just show the *next* method. The *CountingNode* class was described above.

```
public T next()
{
   CountingNode<T> cn;

   if( hasNext() )
   {
      while( true )
      {
         // Pop stack.
         cn = nodes.pop();

         // If node popped for first time...
         if( ++cn.timesPopped == 1 )
         {
            // Push node back onto stack.
            nodes.push( cn );

            // If a left child exists, push it onto stack.
            if( cn.node.left != null )
            {
               nodes.push( new CountingNode<T>( cn.node.left ) );
            }
         }
         // If node is popped for second time...
         else if( cn.timesPopped == 2 )
         {
            // Push node back onto stack.
            nodes.push( cn );

            // If a right child exists, push it onto stack.
            if( cn.node.right != null )
            {
               nodes.push( new CountingNode<T>( cn.node.right ) );
            }
         }
         // If node popped for third time, return the node/item.
         else if( cn.timesPopped == 3 )
         {
            return cn.node.getItem();
         }
      }
   }
   return null;
}
```

18.4.4 – BinaryTree Level-Order Iterator

1. Level-order processing was described earlier. The same approach is used to implement an iterator. A queue is used to contain nodes that have not been visited yet. At each iteration, the node at the front of the queue is removed and then, its children are placed, left, then right, at the end of the queue.

2. Here, we show only the *next()* method.

```
public class BTreeLevelOrdIterator<T> implements Iterator<T>
{
   private Queue<BinaryNode<T>> qNodes;

   public BTreeLevelOrdIterator( BinaryTree<T> tree )
   {
      // Create queue.
      qNodes = new LinkedList<BinaryNode<T>>();
      // Put root node into queue.
      if( tree.getRoot() != null )
         qNodes.offer( tree.getRoot() );
   }

   public boolean hasNext() { return qNodes.size()>0; }

   public T next()
   {
      BinaryNode<T> n;

      if( hasNext() )
      {
         // Remove node from queue.
         n = qNodes.poll();
         if( n.left != null )
            // Put left child onto queue.
            qNodes.offer( n.left );

         if( n.right != null )
            // Put right child onto queue.
            qNodes.offer( n.right );

         // Return the item removed from the queue.
         return n.getItem();
      }
      return null;
   }
```

Example

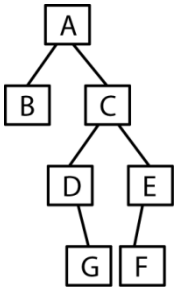
1. An example of these four iterators is provided in the math/cs public drive: dgibson/cs3410/18 (and in HW 3 download code). There, the BinaryTree class is outfitted with a property that determines which iterator is used. A portion of the BinaryTree code is shown below.

```
public enum IteratorType
{
   PreOrder, PostOrder, InOrder, LevelOrder;
}

private IteratorType iterType = IteratorType.PreOrder;


public void setIteratorType( IteratorType t )
{
   iterType = t;
}

public Iterator<T> iterator()
{
   switch( iterType )
   {
      case PostOrder:
         return new BTreePostOrdIterator<T>( this );
      case LevelOrder:
         return new BTreeLevelOrdIterator<T>( this );
      case PreOrder:
         return new BTreePreOrdIterator<T>( this );
      case InOrder:
         return new BTreeInOrdIterator<T>( this );
   }
   return new BTreePreOrdIterator<T>( this );
}
```

Homework 18.7

1. Write a pre-order iterator for the BinaryTree class.
2. Write pseudo-code for a post-order iterator.
3. Write pseudo-code for an in-order iterator.
4. Problem 18.4, p.683
5. Consider the tree below. Show the contents of the...
   a. Stack each time *next* is called as an iterator traverses the nodes in pre-order.
   b. Stack each time *next* is called as an iterator traverses the nodes in pin-order.
   c. Stack each time *next* is called as an iterator traverses the nodes in post-order.
   d. Queue each time *next* is called as an iterator traverses the nodes in level-order.

   Tree and solution for (a) follow.



	Stack	Return
Initial Contents	A	
next()	B	
pop A, push C and B	C	
return A		A
next()	C	
pop B (no children to push)		
return B		B
next()	D	
pop C, push E and D	E	
return C		C
next()	G	
pop D, push G (no left child)	E	
return D		D
next()	E	
pop G (no children to push)		
return G		G
next()	F	
pop E, push F (no right child to push)		
return E		E
hasNext()		
pop F (no children to push)		
return F		F