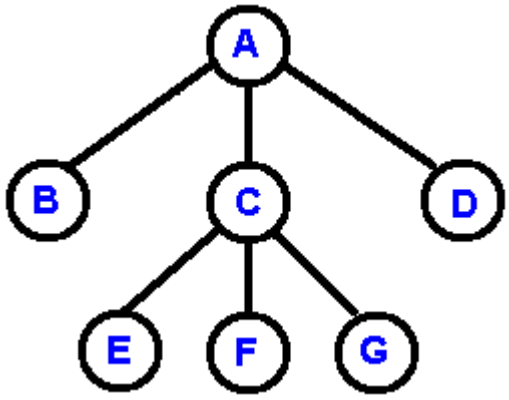
Binary Trees

Introduction
We extend the concept of linked data structures to structure containing nodes with more than one self-referenced field.

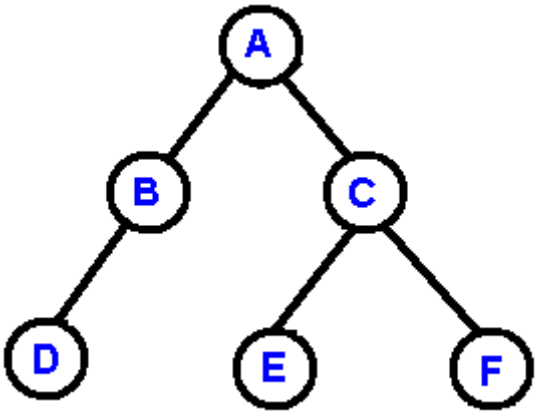
Definition
A tree is either empty or consists of one node called the root and zero or more subtrees.

Every node (exclude a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings. In the following picture

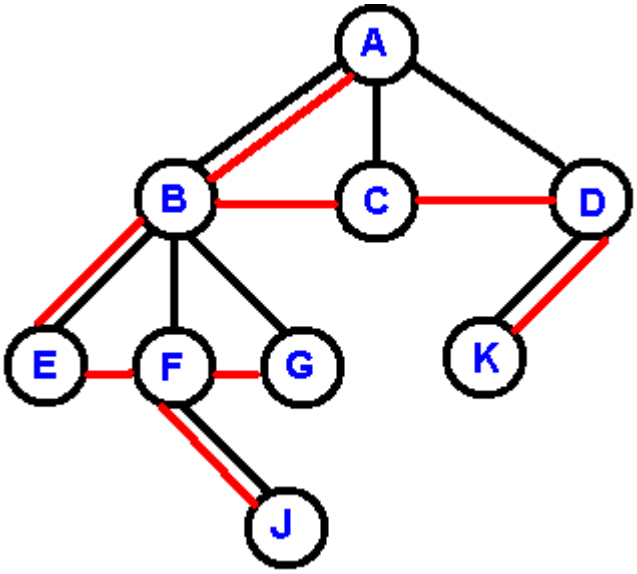


A is the root;
B, C and D are children of A;
C is the parent of E, F and G;
B, D, E, F and G are leaves;
B, C and D are siblings

Definition
A binary tree is either empty or consists of a root, a left subtree and a right subtree.



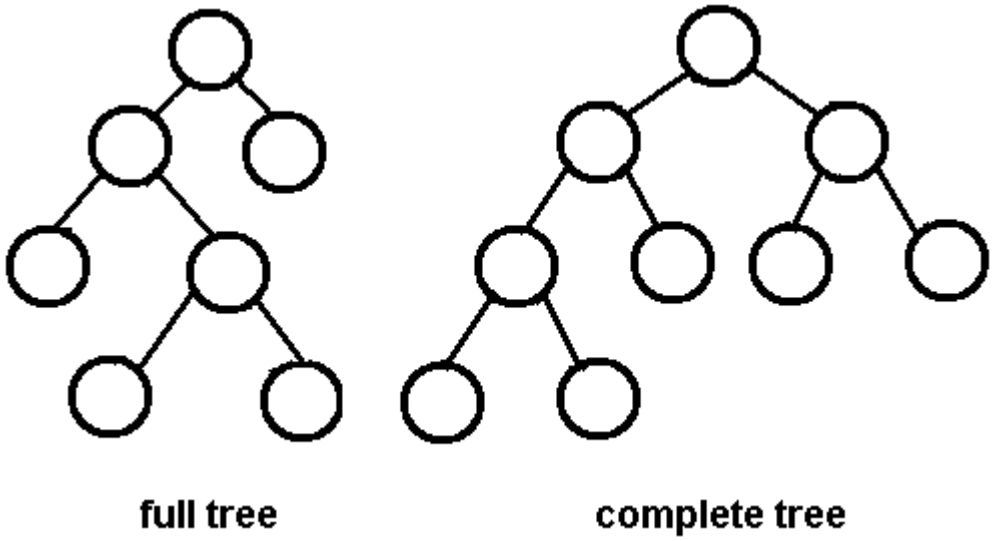
Definition
The depth of a node is the number of edges from the root to the node.
The height of a node is the number of edges from the node to the deepest leaf. The height of a tree is a height of the root.



There exists a unique mapping from any (ordered) tree to a binary tree. A general tree can be implemented using only two references for each node: one to the leftmost child, and one to the sibling. The following picture illustrates that mapping (nodes of the general tree are in black, and edges of a binary tree are in red)

Definition
A binary tree in which each node has exactly zero or two children is called a full binary tree.

Definition
A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.



full tree complete tree

Exercise. What is the maximum height of a binary tree with N nodes?

Exercise. What is the minimum height of a binary tree with N nodes?

The minimum height is provided by a tree that is completely filled with nodes. Let us assume that a tree height is h. Then the maximum number of nodes is given by (we count nodes by levels)

$$n = 1 + 2 + 4 + ... + 2h-1 + 2h = 2h+1 - 1$$

Exercise. What is the maximum number of external nodes (or leaves) in a binary tree of height h?

Tree Traversals

There are four standard methods of traversing trees:

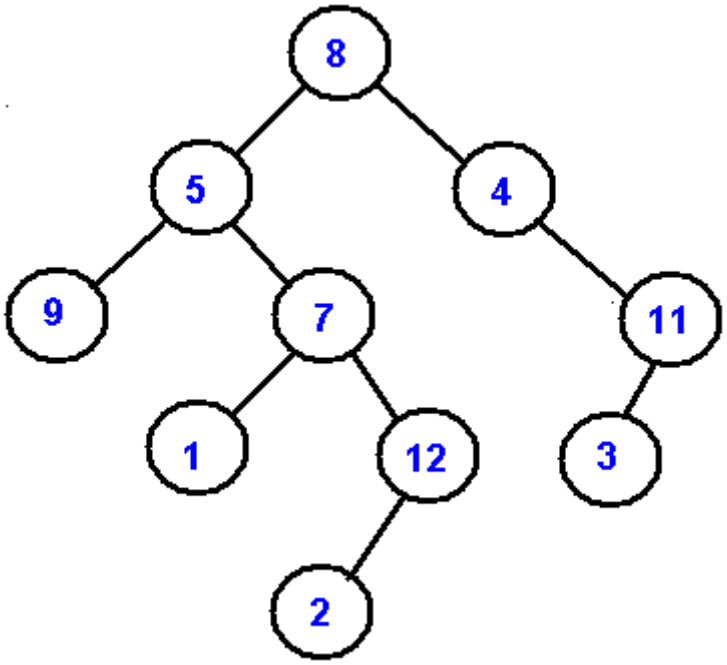
PreOrder traversal -visit the parent first and then left and right children;

InOrder traversal -visit the left child, then the parent and the right child;

PostOrder traversal -visit left child, then the right child and then the parent;

LevelOrder traversal -visit nodes by levels from top to bottom and from left to right.

Consider the following tree:



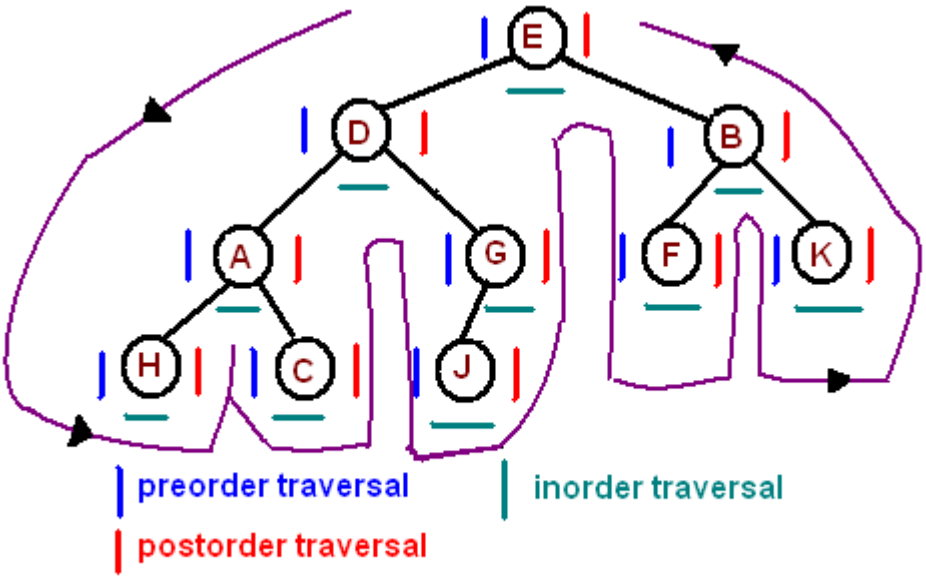
PreOrder Traversal -8, 5, 9, 7, 1, 12, 2, 4, 11, 3

InOrder Traversal -9, 5, 1, 7, 2, 12, 8, 4, 3, 11

PostOrder Traversal -9, 1, 2, 12, 7, 5, 3, 11, 4, 8

LevelOrder Traversal -8, 5, 4, 9, 7, 11, 1, 12, 3, 2

These common traversals can be represented as a single algorithm by assuming that we visit each node three times. An Euler tour is a walk around the binary tree where each edge is treated as a wall, which you cannot cross. In this walk each node will be visited either on the left, or under the below, or on the right. The Euler tour in which we visit nodes on the left produces a *preorder* traversal. When we visit nodes from the below, we get an *inorder* traversal. And when we visit nodes on the right, we get a *postorder* traversal. Traversals can be easily implemented recursively.



Exercise. Draw a binary tree T such that all the following conditions are satisfied:

- each node stores a single number and
- a preorder traversal of T yields 6,4,2,1,3,5,10,8,7,9 and
- an inorder traversal of T yields 1,2,3,4,5,6,7,8,9,10

Complexity of traversal

Assume that T(n) is the function that describes complexity of traversing a binary tree with n nodes. T(n) accumulates complexity for visiting the root -it takes constant time -and for visiting the left subtree T(m) and the right subtree T(k), where m+k = n-1. Therefore,

$$T(n) = c + T(m) + T(k)$$

We solve this recurrence equation asymptotically, by guessing that
T(n) < a*n. It follows that

$$T(n) = c + T(m) + T(k) < a1*m + a2*k = \max(a1,a2) * (m + k) = O(n)$$

Tree iterator

We implement a non-recursive preorder traversal by adding a new method iterator. This method returns an iterator over the nodes of a binary tree in pre-order:

```
public Iterator iterator()
{
     return new PreOrderIterator();
}
```

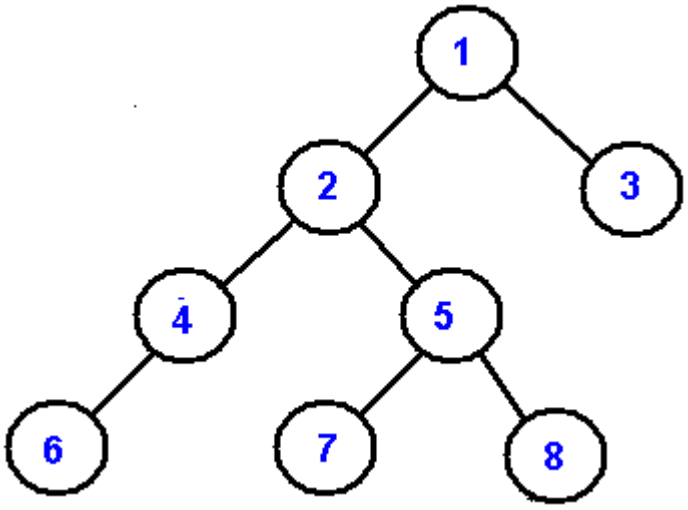
The PreOrderIterator class is implemented as an inner private class of the BTree class

```
private class PreOrderIterator implements Iterator
{
     ...
}
```

The algorithm uses the Stack as an intermediate storage. We start with the root and push it on a stack. When a user calls for next(), we check if the top element has a left child. If it has a left child, we push the child on a stack and return a parent node. If there is no left child, we check for a right child. If it has a right child, we push the right child on a stack and return a parent node. If there is no right child, we move back up the tree (in a while-loop by popping elements from a stack) until we find a node with a right child.

```
public Object next() {
     BNode cur = stk.peek();
     if(cur.left != null){
          stk.push(cur.left);
     }
     else {
          BNode tmp = stk.pop();
          while(tmp.right == null){
               if (stk.isEmpty())
                    return cur;
               tmp = stk.pop();
          }
          stk.push(tmp.right);
     }
     return cur;
}
```

Let us demonstrate the above code on the following example.



Here is a table showing the output and the stack during each call to next()

Output		1	2	4	6	5	7	8	3
				6					
			4	4		7			
		2	2	2	5	5	8		
Stack	1	1	1	1	1	1	1	3	

Expression Trees

Typically we deal with mathematical expressions in *infix* notation where the operators (e.g. +, *) are written between the operands:

2 + 5

Here, 2 and 5 are called operands, and the '+' is operator. The above arithmetic expression is called *infix*, since the operator is in between operands. Writing the operators after the operands gives a *postfix* form.

2 5 +

In a *prefix* form, operands follow operator:

+ 2 5

The *infix* form has a disadvantage that parentheses must be used to indicate the order of evaluation. For example, the following expression is ambiguous

7 + 10 * 15

Evaluation becomes much easier if we evaluate the operators from left to right. This leads to a *postfix* form expression. *Postfix* form has a nice property that parentheses are unnecessary.

7 10 15 * +