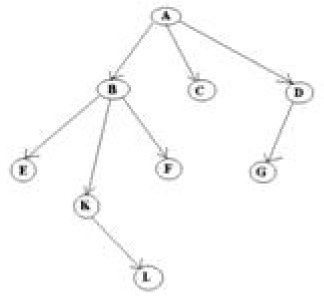
Tree Data Structure

There are many basic data structures that can be used to solve application problems. Array is a good static data structure that can be accessed randomly and is fairly easy to implement. Linked Lists on the other hand is dynamic and is ideal for application that requires frequent operations such as add, delete, and update. One drawback of linked list is that data access is sequential. Then there are other specialized data structures like, stacks and queues that allows us to solve complicated problems (eg: Maze traversal) using these restricted data structures. One other data structure is the hash table that allows users to program applications that require frequent search and updates. They can be done in O(1) in a hash table.

One of the disadvantages of using an array or linked list to store data is the time necessary to search for an item. Since both the arrays and Linked Lists are **linear structures** the time required to search a “linear” list is proportional to the size of the data set. For example, if the size of the data set is n, then the number of comparisons needed to find (or not find) an item may be as bad as some multiple of n. So imagine doing the search on a linked list (or array) with $n = 10^6$ nodes. Even on a machine that can do million comparisons per second, searching for m items will take roughly m seconds. This not acceptable in today’s world where speed at which we complete operations is extremely important. Time is money. Therefore it seems that better (more efficient) data structures are needed to store and search data.

In this chapter, we can extend the concept of linked data structure (linked list, stack, queue) to a structure that may have multiple relations among its nodes. Such a structure is called a **tree**. A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a *nonlinear* data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more subtrees. A tree has following general properties:

- One node is distinguished as a **root**;
- Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent -> children*



A is a parent of B, C, D,
B is called a child of A.
on the other hand, B is a parent of E, F, K

In the above picture, the root has 3 subtrees.

Each node can have *arbitrary* number of children. Nodes with no children are called **leaves**, or **external** nodes. In the above picture, C, E, F, L, G are leaves. Nodes, which are not leaves, are called **internal** nodes. Internal nodes have at least one child.

Nodes with the same parent are called **siblings**. In the picture, B, C, D are called siblings. The **depth of a node** is the number of edges from the root to the node. The depth of K is 2. The **height of a node** is the number of edges from the node to the deepest leaf. The height of B is 2. The **height of a tree** is a height of a root.

A General Tree

A general tree is a tree where each node may have zero or more children (a binary tree is a specialized case of a general tree). General trees are used to model applications such as file systems.

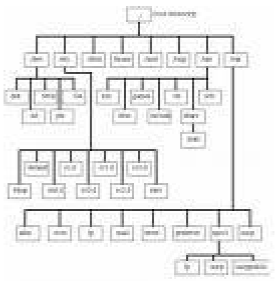


Figure courtesy of **www.washington.edu**

Implementation

Since each node in a tree can have an arbitrary number of children, and that number is not known in advance, the *general* tree can be implemented using a **first child/next sibling** method. Each node will have **TWO** pointers: one to the leftmost child, and one to the rightmost sibling. The following picture illustrates this

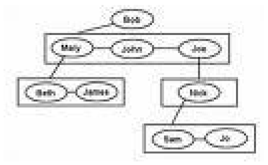


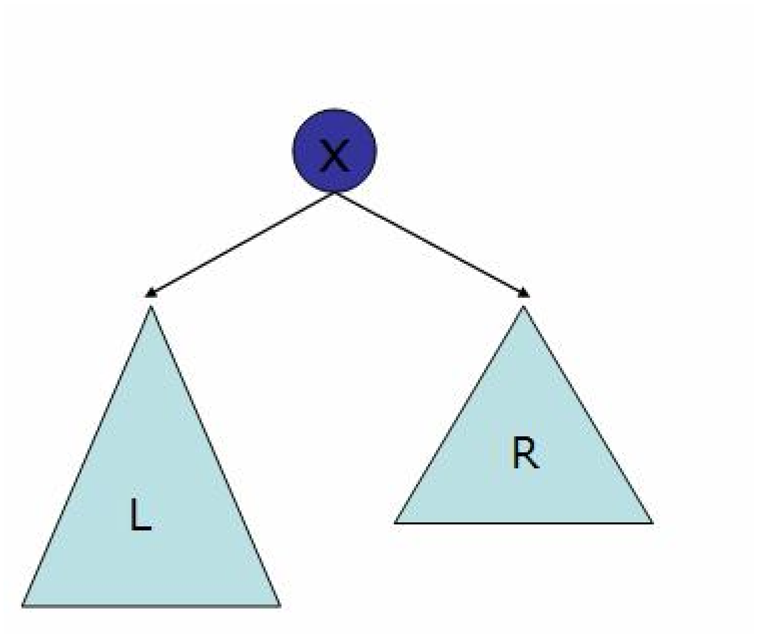
Figure courtesy of www.gamedev.net

The following Java code may be used to define a general tree node.

```
public class TNode {
    private Object  data;
    private MyLinkedList siblings;
    private TNode myLeftChild;
    public TNode(Object n){data=n; siblings=NULL;myLeftChild=NULL;}
}
```

Binary Trees

We will see that dealing with **binary** trees, a tree where each node can have no more than two children is a good way to understand trees.



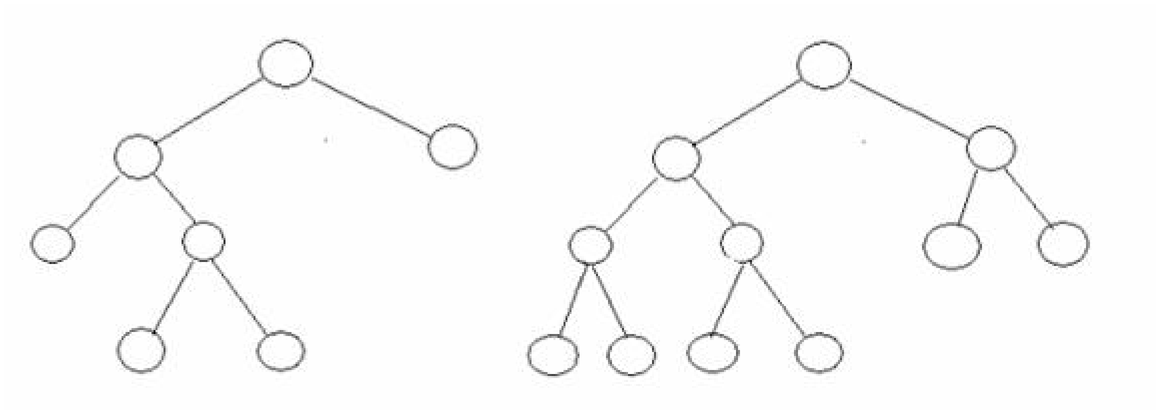
Here is a Java prototype for a tree node:

```
public class BNode
{
   private Object data;
   private BNode left, right;
   public BNode()
   {
      data=left=right=null;
   }
   public BNode(Object data)
   {
      this.data=data;
      left=right=null;
   }
}
```

A binary tree in which each node has exactly zero or two children is called **a full binary tree**. In a full tree, there are no nodes with exactly one child.

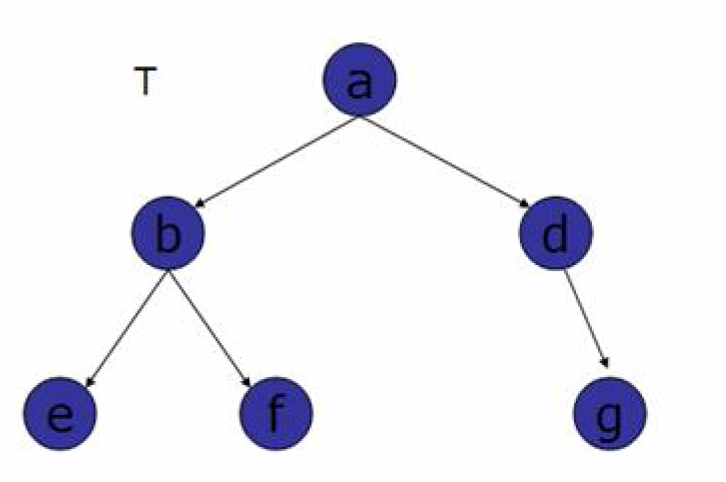
A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2^h and $2^{(h+1)}$-1 nodes. Here are some examples:

full tree **complete tree**



Binary Search Trees

Given a binary tree, suppose we visit each node (recursively) as follows. We visit left child, then root and then the right child. For example, visiting the following tree



In the order defined above will produce the sequence {e, b,f,a,d,g} which we call flat(T). A binary search tree (BST) is a tree, where flat(T) is an ordered sequence. In other words, a binary search tree can be "searched" efficiently using this ordering property. A "balanced" binary search tree can be searched in O(log n) time, where n is the number of nodes in the tree.

In the next lesson we will learn some of the operations on BST's.