**Tree**

The ADTs in this chapter organize data in a nonlinear, hierarchical form.  
The ADTs list, stack, and queue are all position oriented, and their operations have the form:  
1. Insert a data item into the ith position of a data collection  
2. Delete a data item from the ith position of a data collection  
3. Ask a question about the data item into the ith position of a data collection

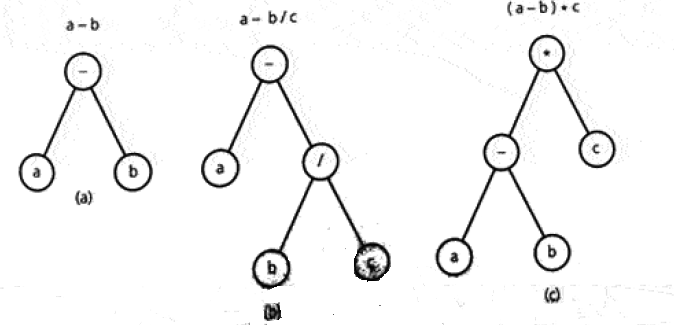
As you have seen, the ADT list places no restriction on the value of i, while the ADTs stack and queue do impose some restrictions. Thus, although they differ with respect to the flexibility of their operations, lists, stacks, and queues manage an association between data items and positions.

This chapter discussed two major ADTs: the binary tree and binary search tree. As you will see, the binary tree is a position-oriented ADT, but it is not linear as are lists, stacks, and queues. Thus, you will not reference items in a binary tree by using a position number. Our discussion of the ADT binary tree provides an important background for the more useful binary search tree, which is a value-oriented ADT. Although a binary search tree is also not linear, it has operations similar to those of a sorted list, which is linear.

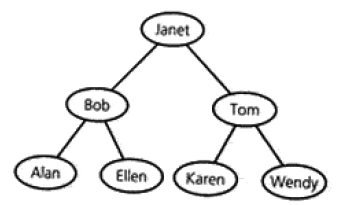
**1. Terminology**

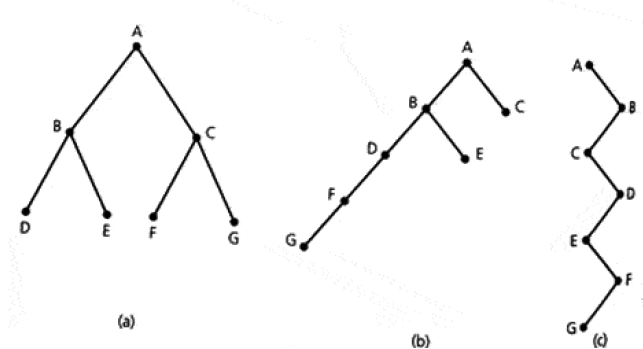
All trees are hierarchical in nature. Intuitively, hierarchical means that a “parent-child” relationship exists between node in the tree. There are parent node and child node. Children of the same parent are called *sibling*. Each node in a tree has at most one parent, and exactly one node - called *root* of the tree -has no parent. A node that has no children is called a leaf of the tree.  
The parent-child relationship between the nodes is generalized to the relationships *ancestor* and *descendant.* Not all nodes are related by the ancestor or descendant relationship. However, the root of any tree is an ancestor of every node in the tree. A *subtree* in a tree is any node in the tree together with all of its descendant.  
  
Because trees are hierarchical in nature, you can use them to represent information that itself is hierarchical in nature.  
  
Binary tree is a set T of nodes such that either  
1. T is empty  
2. T is partitioned into three disjoint subsets:  
 a. A single node r, the root  
 b. Two possible empty sets that are binary trees, called left and right subtrees of r.

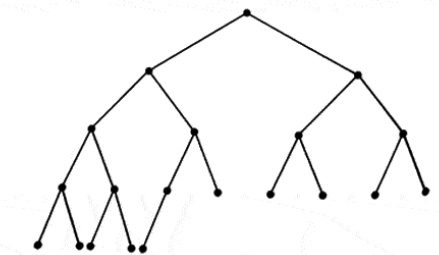
Intuitive definition of a binary tree:  
T is binary tree if either:  
1. T has no nodes, or  
2. T is of the form  
  
where r is a node and TL and TR are both binary trees.

As an example of how you can use a binary tree to represent data in a hierarchical form, consider the following example:  


The binary trees in this figure represent algebraic expressions that involve the binary operations +, -, \*, and /. To represent an expression such as *a-b,* you place the operator in the root node and the operands into the left and right children, respectively, of the root.  
Parentheses do not appear in these trees. The binary tree provides a hierarchy for the operations- that is, the tree specifies an unambiguous order for evaluating an expression.

A binary search tree is a binary tree that is in a sense sorted according to the values in its nodes. For each node *n*, a binary search tree satisfies the following three properties:  
1. N’s value is greater than all values in its left subtree TL  
2. N’s value is less than all values in its right subtree TR  
3. Both TL and TR are binary search trees.  
  
As its name suggest, a binary search tree organized data in a way that facilitates searching it for a particular data item  


*The height of trees:*   
  
Binary trees with the same nodes but different heights.

Although each of these trees has seven nodes, some are “taller” than others. The height of s tree is the number of nodes on the longest path from the root to a leaf.  
For binary trees, it is often convenient to use recursive definition of height:  
1. If T is empty, its height is 0.  
2. If T is nonempty binary tree, then the height of T is 1 greater than the height of its root’s taller subtree. That is:  
height(T) = 1 + max( height(TL), height(TR))  
  
*Full, complete, and balanced binary trees.*  
In a full binary tree of height *h*, all nodes that are at a level less than *h* have two children each. Each node in full binary tree has left and right subtrees of the same height. A full binary tree has as many leaves as possible, and they all are at level *h*. Intuitively, a full binary tree has no missing nodes.  
A complete binary tree of height *h* is a binary tree that is full down to level *h* with level h filled in from left to right:  


Finally, a binary tree is height balanced, or simply balanced, if the height of any node’s right subtree differs from the height of the node’s left subtree by no more than 1.

**1.2 The ADT Binary Tree**

As an abstract data type, the binary tree has operations that add and remove nodes and subtrees.

Basic Operations of the ADT Binary Tree  
1. Create an empty binary tree  
2. Create a one-node binary tree, given an item.  
3. Remove all nodes from a binary tree, given an item.  
4. Determine whether a binary tree is empty  
5. Determine what data is the binary tree’s root  
6. Set the data in the binary tree’s root (may not be implemented by all binary trees).  
  
*Pseudocode for the Basic Operations of the ADT Binary Tree*

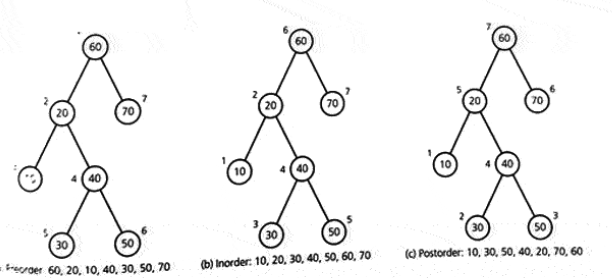
1. createBinaryTree()  
 // Create an empty binary tree  
2. createBinaryTree(in rootItem:TreeItemType)  
 // Create a one-node binary tree whose root contains rootItem  
3. makeEmpty()  
 // Remove all nodes from a binary tree, leaving an empty tree.  
4. isEmpty()  
 // Determine whether a binary tree is empty  
5. getRootItem():TreeItemType throws TreeException  
 // Retrieves the data item in the root of a nonempty binary tree. Throws TreeException if the tree is empty  
6. setRootItem(in rootItem:TreeItemType) throws UnsupportedOperationException  
 // Sets the data item in the root of a binary tree. Throws UnsupportedOperationException if the method is not implemented.  
   
As you can see, we must still specify other operations for building the tree. One possible set of operations is presented next.

*General Operations of the ADT Binary Tree*

The particular operations provided for an ADT binary tree depend on the kind of binary tree we are designing. A UML diagram for a binary tree is shown:  
A stack of dishes in a cafeteria makes a very good analogy of the abstract data type stack.

|  |
| --- |
| BinaryTree |
| root |
| left subtree right subtree |
| createBinaryTree() |
| makeEmpty() |
| isEmpty() |
| getRootItem() |
| setRootItem()  attachLeft()  attachRight()  attachLeftSubtree() attachRightSubtree()  detachLeftSubtree()  detachRightSubtree()  getLeftSubTree()  getRightSubtree() |

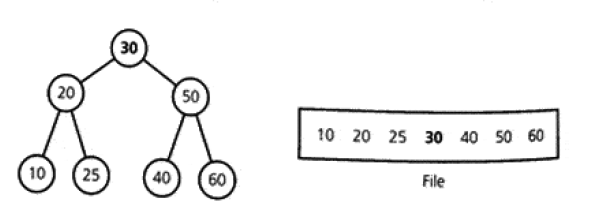
*Traversals of a Binary Tree*

  
Traversals of a binary tree: a) preorder, b) inorder, c) postorder

Each of these traversals visits every node in a binary tree exactly once. Thus, n visits occur for a tree of n nodes. Each visit performs the same operations on each node, independently of n, so it must be O(1). Thus, traversal is O(n).

Possible Representations of a Binary Tree

There are 3 different approaches in implementations of binary tree and 2 of these approaches use arrays, but the typical implementation uses references.

3.   
Reading the Binary Search Tree from File (array).  


References   
[1] Carrano Prichard Data Abstraction and Problem Solving with Java 3rd ed. 573-