7.3. Binary Trees

7.3.1. Binary Trees. A *binary tree* is a rooted tree in which each vertex has at most two children, designated as *left child* and *right child*. If a vertex has one child, that child is designated as either a left child or a right child, but not both. A *full binary tree* is a binary tree in which each vertex has exactly two children or none. The following are a few results about binary trees:

1. If T is a full binary tree with i internal vertices, then T has $i+1$ terminal vertices and $2i+1$ total vertices.
2. If a binary tree of height h has t terminal vertices, then $t \leq 2^h$.

7.3.2. Binary Search Trees. Assume S is a set in which elements (which we will call "data") are ordered; e.g., the elements of S can be numbers in their natural order, or strings of alphabetic characters in lexicographic order. A *binary search tree* associated to S is a binary tree T in which data from S are associate with the vertices of T so that, for each vertex v in T, each data item in the left subtree of v is less than the data item in v, and each data item in the right subtree of v is greater than the data item in v.

Example: Figure 7.10 contains a binary search tree for the set $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$. In order to find a element we start at the root and compare it to the data in the current vertex (initially the root). If the element is greater we continue through the right child, if it is smaller we continue through the left child, if it is equal we have found it. If we reach a terminal vertex without founding the element, then that element is not present in S.

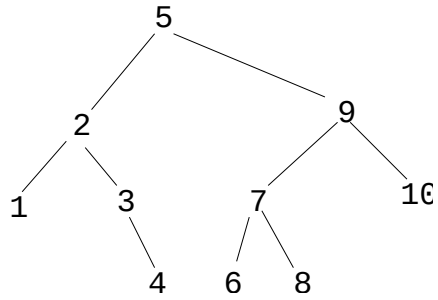

FIGURE 7.10. Binary Search Tree.

7.3.3. Making a Binary Search Tree. We can store data in a binary search tree by randomly choosing data from S and placing it in the tree in the following way: The first data chosen will be the root of the tree. Then for each subsequent data item, starting at the root we compare it to the data in the current vertex v. If the new data item is greater than the data in the current vertex then we move to the right child, if it is less we move to the left child. If there is no such child then we create one and put the new data in it. For instance, the tree in figure 7.11 has been made from the following list of words choosing them in the order they occur: "IN A PLACE OF LA MANCHA WHOSE NAME I DO NOT WANT TO REMEMBER".

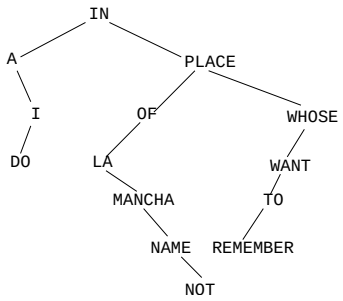

FIGURE 7.11. Another binary Search Tree.

7.3.4. Tree Transversal's. In order to motivate this subject, we introduce the concept of Polish notation. Given a (not necessarily commutative) binary operation $\circ$, it is customary to represent the result of applying the operation to two elements a, b by placing the operation symbol in the middle:

$$a \circ b\,.$$

This is called *infix* notation. The *Polish* notation consists of placing the symbol to the left:

$$\circ\, a\, b\,.$$

The *reverse Polish* notation consists of placing the symbol to the right:

$$a\, b\, \circ\ .$$

The advantage of Polish notation is that it allows us to write expressions without need for parenthesis. For instance, the expression $a*(b+c)$ in Polish notation would be $*\,a+b\,c$, while $a*b+c$ is $+*a\,b\,c$. Also, Polish notation is easier to evaluate in a computer.

In order to evaluate an expression in Polish notation, we scan the expression from right to left, placing the elements in a stack.[1] Each time we find an operator, we replace the two top symbols of the stack by the result of applying the operator to those elements. For instance, the expression $*+2\,3\,4$ (which in infix notation is "$(2+3)*4$") would be evaluated like this:

expression	stack
$*+2\,3\,4$	
$*+2\,3$	4
$*+2$	3 4
$*+$	2 3 4
$*$	5 4
	20

An algebraic expression can be represented by a binary rooted tree obtained recursively in the following way. The tree for a constant or variable a has a as its only vertex. If the algebraic expression S is of the form $S_L \circ S_R$, where S_L and S_R are subexpressions with trees T_L and T_R respectively, and $\circ$ is an operator, then the tree T for S consists of $\circ$ as root, and the subtrees T_L and T_R (fig. 7.12).

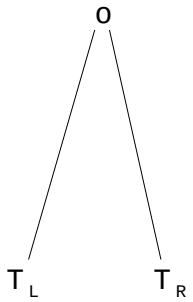

FIGURE 7.12. Tree of $S_1 \circ S_2$.

For instance, consider the following algebraic expression:

$$a + b * c + d \uparrow e * (f + h)\,,$$

where $+$ denotes addition, $*$ denotes multiplication and $\uparrow$ denotes exponentiation. The binary tree for this expression is given in figure 7.13.

[1] A *stack* or *last-in first-out* (LIFO) system, is a linear list of elements in which insertions and deletions take place only at one end, called *top* of the list. A *queue* or *first-in first-out* (FIFO) system, is a linear list of elements in which deletions take place only at one end, called *front* of the list, and insertions take place only at the other end, called *rear* of the list.

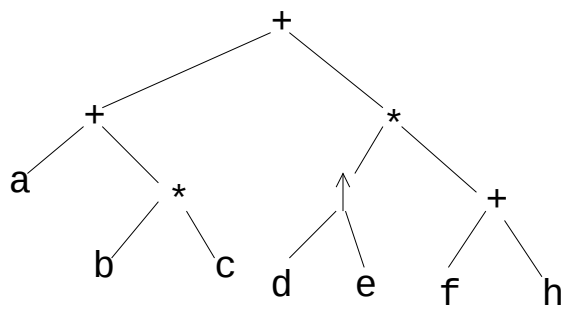

FIGURE 7.13. Tree for $a + b * c + d \uparrow e * (f + h)$.

Given the binary tree of an algebraic expression, its Polish, reverse Polish and infix representation are different ways of ordering the vertices of the tree, namely in *preorder*, *postorder* and *inorder* respectively.

The following are recursive definitions of several orderings of the vertices of a rooted tree $T = (V, E)$ with root r. If T has only one vertex r, then r by itself constitutes the *preorder*, *postorder* and *inorder* transversal of T. Otherwise, let $T_1, \ldots, T_k$ the subtrees of T from left to right (fig. 7.14). Then:

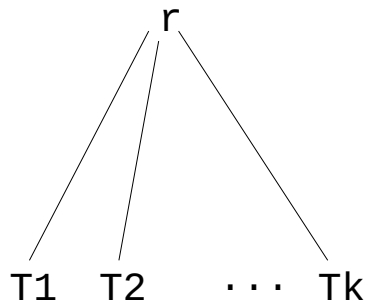

FIGURE 7.14. Ordering of trees.

1. *Preorder Transversal*: $\text{Pre}(T) = r, \text{Pre}(T_1), \ldots, \text{Pre}(T_k)$.
2. *Postorder Transversal*: $\text{Post}(T) = \text{Post}(T_1), \ldots, \text{Post}(T_k), r$.
3. *Inorder Transversal*. If T is a binary tree with root r, left subtree T_L and right subtree T_R, then: $\text{In}(T) = \text{In}(T_L), r, \text{In}(T_R)$.