Binary and rooted trees

Binary trees

- A generalization of numbers.
- Axioms
  - There exists a binary tree called *empty*.
  - If T_1 and T_2 are binary *trees* then so is $root(T_1,T_2)$.
  - If a property holds for the empty tree, and assuming it holds for T_1,T_2 , it holds for $root(T_1,T_2)$, then it holds for all binary trees.
  - $root(T_1,T_2) \neq empty$ and $root(T_1,T_2) = root(T_3,T_4)$ iff $T_1 = T_3$ and $T_2 = T_4$.

Examples

- Objects generated from a fixed object by combining two objects of the same type.
- The combining operation may be different.
- All such objects correspond to binary trees.
- Balanced parenthesis strings, triangulations of a convex polygon, permutations without the pattern $p_i < p_k < p_j$ for $i < j < k$ (1,3,2).

Nodes

- A binary tree can also be considered as a set of *nodes*.
- A node is just an object that can hold some data.
- A tree $T = root(T_1, T_2)$ is considered to be obtained by attaching trees T_1, T_2 to a node, called *root node*.
- T_1 is the *left* and T_2 the *right* subtree of the root node.
- Every node in a binary tree has a left and right subtree.
- The nodes in T are the root along with nodes in T_1, T_2, which are considered to be disjoint sets.

Subtrees

- A binary tree can also be thought of as a collection of subtrees.
- The *empty* tree has only itself as a subtree.
- If $T = root(T_1,T_2)$, the subtrees of *T* are *T* itself along with the subtrees of *T1* and *T2*.
- Every non-empty subtree of *T* has a root node in *T*.
- Every node in *T* is the root of a subtree of *T*.

Terminology

- If $T = root(T_1,T_2)$ is a tree, and T_1 is not empty, the root of T_1 is called the *left* child of the root of T, and root of T_2 the right child, if it exists.
- The root of T is the parent of the roots of T_1, T_2, if they exist.
- A *path* in a tree is a sequence of nodes, $v_1, v_2, \ldots, v_l$ such that v_{i+1} is a child (left or right) of v_i for $1 <= i < l$.
- The length of the path is $l-1$ and it is from v_1 *to* v_l.

Terminology

- A node *b* is a *descendant* of node *a*, and *a* an *ancestor* of *b*, if there exists a path from *a* to *b* in the tree.
- There exists at most one path from *a* to *b*, for any nodes *a, b.* (Prove it).
- Every node is a descendant of the root.
- The *depth* of a node is the length of the path from root to the node.
- The *height* of a node is 1 + length of longest path starting from the node (the extra 1 may not be used in some books).

Labeled Trees

- A sequence is similar to numbers, except that instead of *next(n)* we have *push(S,x)*, where *x* can be any object of some type.
- Similarly, a labeled binary tree is obtained by *root(T1,T2,x)* where *T1,T2* are labeled binary trees and *x* is any object of some type.
- The root node is assumed to be labeled *x*.
- Every node has a label, not necessarily distinct.

Traversals

- A tree traversal converts a labeled tree into a sequence.
- *traverse(empty) = empty.*
- $traverse(root(T_1, T_2, x))$ = (+ is concatenation)
  - $x + traverse(T_1) + traverse(T_2)$ (preorder)
  - $traverse(T_1) + x + traverse(T_2)$ (inorder)
  - $traverse(T_1) + traverse(T_2) + x$ (postorder)

Rooted trees

- A rooted tree is obtained by *root(S)* where *S* is a sequence of rooted trees (possibly empty).
- Any tree in *S* is called a subtree of *root(S)*.
- A node can have any number of subtrees, which are ordered in a sequence.
- The simplest rooted tree is *root(empty)*.
- Every rooted tree is non-empty.

Rooted and binary trees

- Rooted trees are essentially binary trees.
- There is a bijection *f* between them, defined by
- $f(empty) = root(empty)$
- $f(root(T_1,T_2)) = root(push(S_1, f(T_2)))$ where $f(T_1) = root(S_1)$.
- A rooted tree has one more node than the corresponding binary tree.
- Terminology remains same, except there is no left or right child, and only the first child, second child etc.

Implementation

- No standard implementation available in C++.
- Usually represented by a pointer to a node.
- 0 represents empty tree.

```
struct node {
    node *left, *right;
    T  label;
};
node  *root;
```

Implementation

- Algorithms easy to define and also implement recursively.
- Rooted trees can be implemented using the bijection to binary trees.
- A simpler implementation is to number the nodes and store a parent array.
- Convenient for bottom-up algorithms.

Applications

- Trees can be studied as abstract objects.
- Problems on numbers, sequences can be generalized to trees.
- More useful for representing other objects and implementing other data structures.
- More efficient than sequences in some cases.
- Sets, maps, tries, heaps are some data structures with trees as the underlying data structure.