

**Reading:** GT 2.3.0-2.3.4, p. 75–93 [up to 100 minutes]

You may skip the *Unified tree traversal framework* and *Euler Tours* (p. 87–88). It is extremely important to understand how trees are implemented using pointers (references) pp. 92–93.

**Ingredients:** Trees in Math, trees as ADTs, traversals, vector implementation

## Trees in Mathematics

Trees look like this:

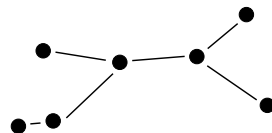


Figure 1: A tree

Trees are often *rooted*

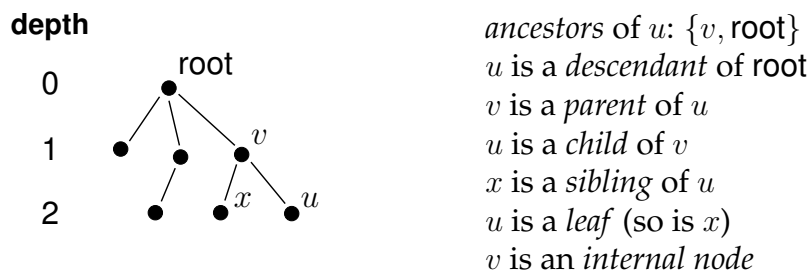


Figure 2: A rooted tree (the same as before, but with a designated root node)

**Definition 1.** A rooted tree such that each of its nodes has at most two children is called a *binary tree*. A binary tree is a *complete binary tree of depth  $n$*  if all its leaves have depth  $n$  and all its internal nodes have exactly two children.

We often denote a complete binary tree of depth  $n$  as  $T_n$ .

**Q.** How many leaves does  $T_n$  have?

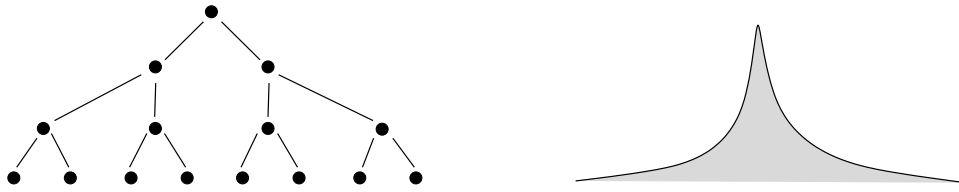


Figure 3:  $T_3$  on the left, a shape of  $T_n$  for large  $n$  on the right.

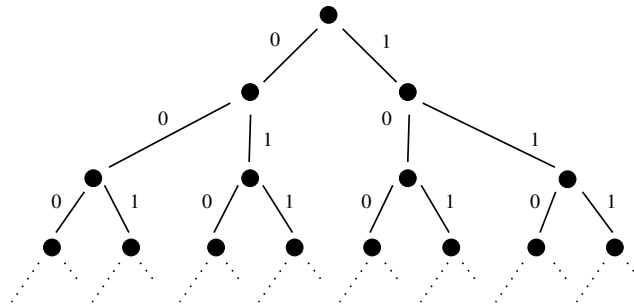


Figure 4: Any leaf is uniquely determined by a sequence of  $n$  bits.

So there is a bijection between the set of leaves of  $T_n$  and the set of all binary strings of length  $n$  (or in other words there is equally many of each).

There are  $2 \cdot 2 \cdot 2 \cdots 2 = 2^n$  different binary sequences of length  $n$ .

**Corollary:** The height  $h$  of any tree of  $n$  nodes is  $\Omega(\log n)$  and  $O(n)$ .

## Trees as Data Structure

Operations:

`ROOT()` — return the root node of the tree

`PARENT( $v$ )` — return the parent node of node  $v$  (error if  $v$  is root)

`CHILDREN( $v$ )` — return an iterator over children of  $v$ , a stuck one if  $v$  external (we use `LEFT( $v$ )` and `RIGHT( $v$ )` instead for binary trees).

Additional: `IS-INTERNAL`, `IS-EXTERNAL`, `IS-ROOT`, `SIZE`, more in the book.

Examples: file hierarchy, GUI tree views, many of the collections classes, AI in games.

## Tree Traversals

**Definition 2.** The depth of a node  $v$  is the number of ancestors of  $v$ , excluding  $v$ .

Algorithms on trees are often very conveniently described using recursion:

```
DEPTH( $T, v$ )
1  if  $T$ .IS-ROOT( $v$ )
2    then return 0
3  return 1 + DEPTH( $T, T$ .PARENT( $v$ ))
```

DEPTH runs in  $O(d_v)$  time, if  $d_v$  denotes the depth of  $v$ ;  $O(n)$  in the worst-case.

**Homework:** study a similar HEIGHT algorithm yourself (in the textbook).

For binary trees we distinguish three standard traversals: PRE-ORDER, IN-ORDER, POST-ORDER.

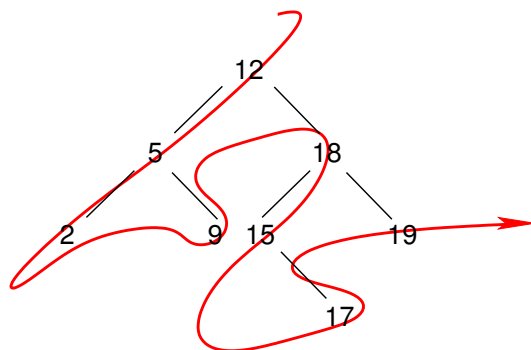


Figure 5: An example of the preorder traversal: 12 5 2 9 18 15 17 9

PREORDER( $T, v$ )	INORDER( $T, v$ )	POSTORDER( $T, v$ )
1 visit $v$	1 visit LEFT( $v$ )	1 visit LEFT( $v$ )
2 visit LEFT( $v$ )	2 visit $v$	2 visit RIGHT( $v$ )
3 visit RIGHT( $v$ )	3 visit RIGHT( $v$ )	3 visit $v$

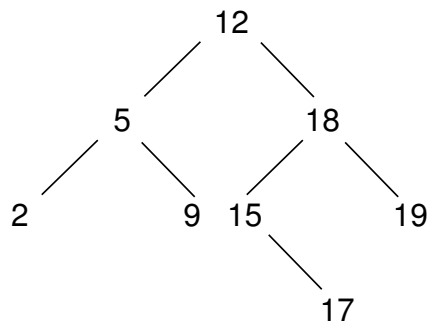
## Implementation

A simple implementation uses objects for nodes, and references for pointers (similarly to lists). **This is by far the most popular way to implement trees!**

We discuss a more exotic vector based implementation.

A *vector based implementation* ranks the nodes by their placement in the tree:

- If  $v$  is the root of  $T$  then  $p(v) = 1$
- If  $v$  is the left child of  $u$ , then  $p(v) = 2p(u)$
- If  $v$  is the right child of  $u$ , then  $p(v) = 2p(u) + 1$



12 5 18 2 9 15 19 - - - - - 17  
 0 1 2 3 4 5 6 7 8 9 10 11 12

Figure 6: An example of vector representation for a binary tree.

**Now you know all the basic data structures:  
 stacks, queues, arrays/vectors, lists, and trees**