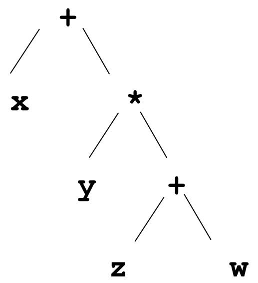
Trees

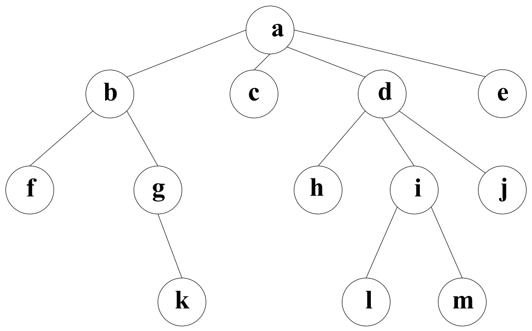
- Family Tree
- Parse Trees
  - e.g. for **x+(y*(z+w))**



- Trees to organize data bases / file systems
  - e.g. The **UNIX** file system
- Search Trees

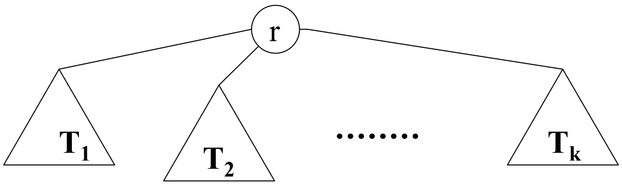
Rooted Trees
Basic Terminology

- nodes (vertices)
- root
- parent
- children
- siblings
- degree



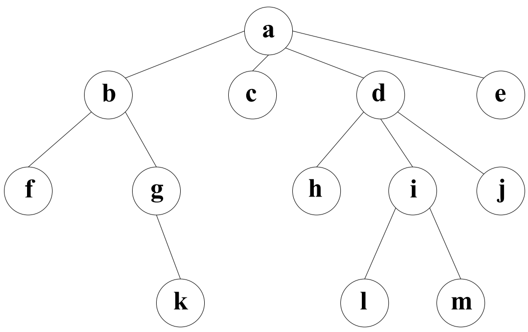
Trees - Recursive Definition

- The <u>empty tree</u> Λ has no nodes
- Given trees $T_1, T_2, \ldots, T_k$
  with roots $r_1, r_2, \ldots, r_k$ respectively,
  and a node r,
  we can form the tree T by making r the root,
  and making $r_1, r_2, \ldots, r_k$ the children of r



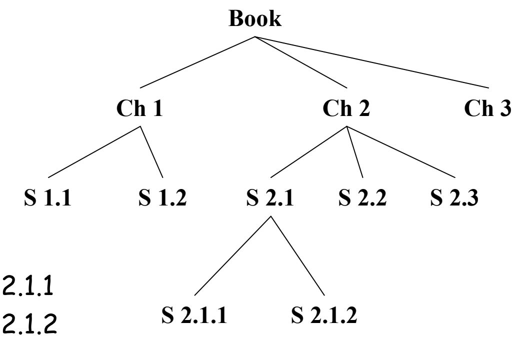
Trees - More Terminology

- path
- ancestor
- descendent
- subtree
- leaf
- height
- depth



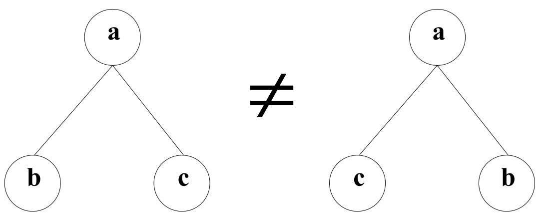
Example - Book

- Book
  - Chapter 1
    - Section 1.1
    - Section 1.2
  - Chapter 2
    - Section 2.1
      - Subsection 2.1.1
      - Subsection 2.1.2
    - Section 2.2
    - Section 2.3
  - Chapter 3



Ordered Trees

The children of each node are ordered



`Leftmost-Child`

`Right-Sibling`

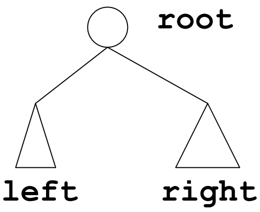
Binary Trees

Recursive Definition:

A <u>binary tree</u>

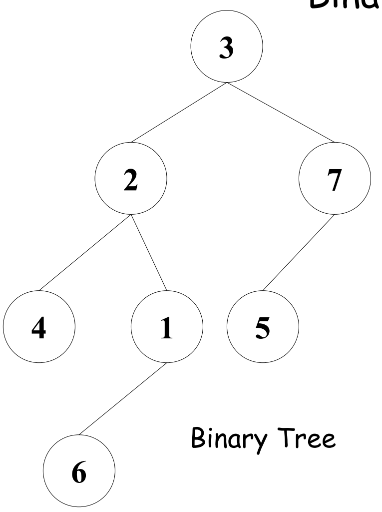
- contains no nodes (Λ),

  or,
- has 3 disjoint sets of nodes:
  - a <u>root</u>
  - a binary subtree called its <u>left subtree</u>
  - a binary subtree called its <u>right subtree</u>



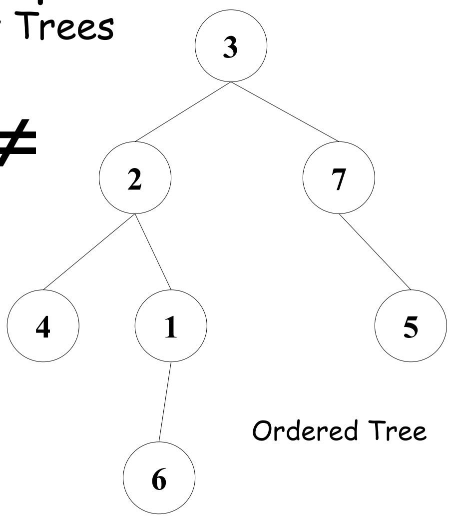
Example

Binary Trees



Binary Tree

$\neq$



Ordered Tree

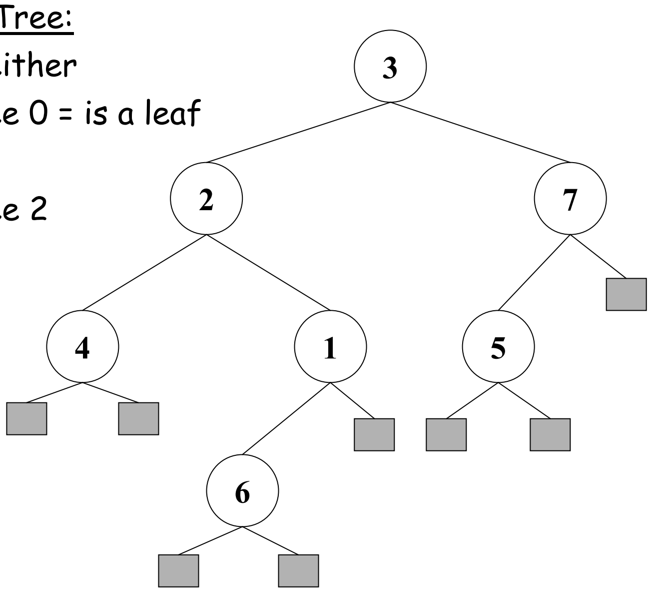
Full Binary Tree:

Each node either

- has degree 0 = is a leaf

or

- has degree 2



Positional Trees

Recursive Definition:

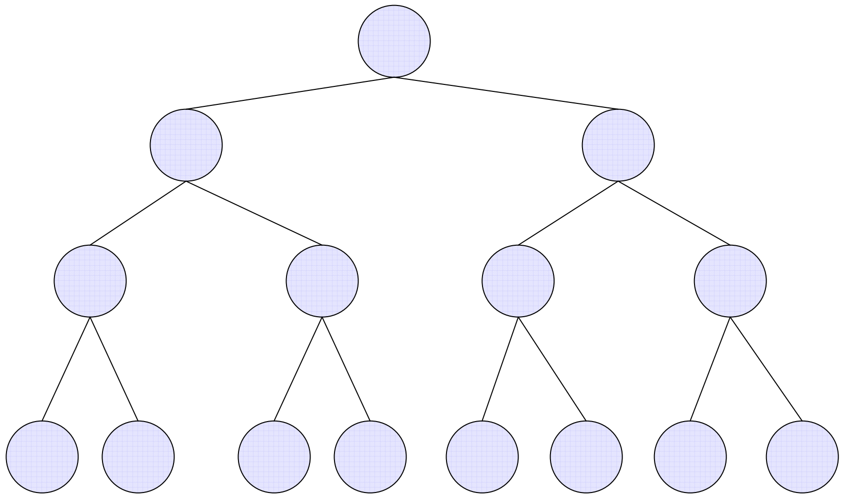
A **k-ary** tree

- contains no nodes (Λ),

  or,

- has **k+1** disjoint sets of nodes:
  - a root
  - **k** **k**-ary sub-trees

Complete Binary Tree



Complete k-ary Tree

- All leaves have same depth
- All internal nodes have degree **k**

- **k**-ary tree of height **h** has:
  - **k** nodes at depth 1
  - $k*k = k^2$ nodes at depth 2

    . . .
  - $k*k*...*k = k^h$ leaves
  - $1 + k + k^2 + ... + k^{h-1} = \frac{k^h-1}{k-1}$ internal nodes

Tree ADT

`Make-Tree(r,T1,…,Tk)` - make a tree `T` with root `r` and $T_1, T_2, \ldots, T_k$ as its sub-trees

`Root(T)` - return the root of tree `T`

`Parent(n,T)` - return the parent of node `n`
Λ if `n` is the root

`Leftmost-Child(n,T)` - return the first child of `n`
Λ if `n` is a leaf

`Right-Sibling(n,T)` - return the right sibling of `n`
Λ if `n` is rightmost

Binary Tree ADT

`Make-Tree(r,Tl,Tr)` - make a tree `T` with root `r` and T_l, T_r as its subtrees

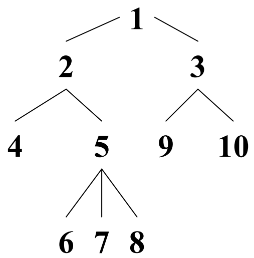
`Root(T)` - return the root of tree `T`

`Parent(n,T)` - return the parent of node `n`
Λ if `n` is the root

`Left-Child(n,T)` - return the left child of `n`
Λ if `n` is a leaf

`Right-Child(n,T)` - return the right child of `n`
Λ if `n` is rightmost

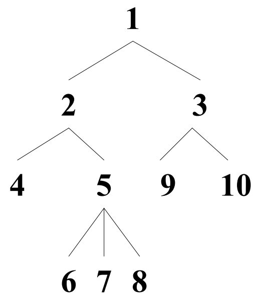
Trees - Implementation

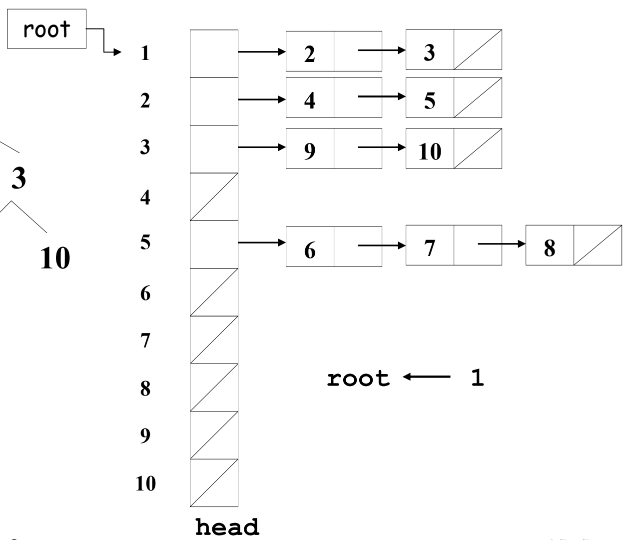


1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	5	5	5	3	3

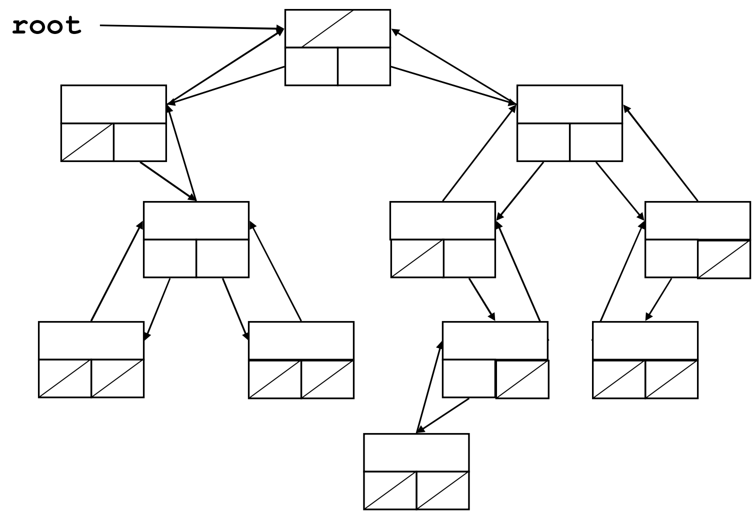
Parent representation

Trees - Implementation

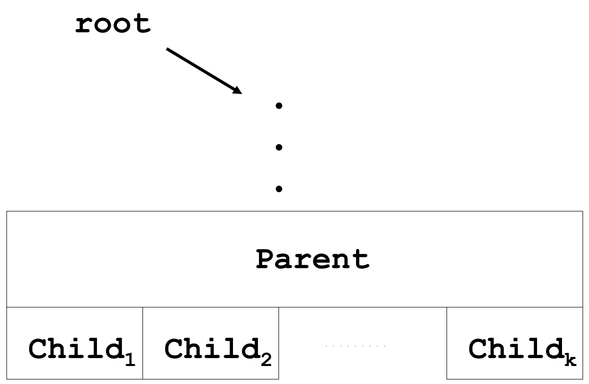




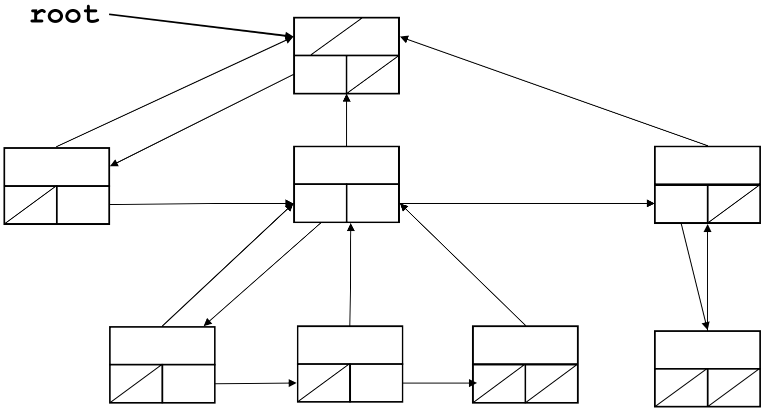
Binary Tree Data structure



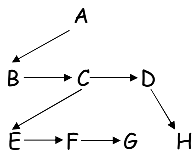
K-ary Tree Representaion



Tree Data structure



Example



	element	parent	leftmost child	right sibling
0	A	-1	1	-1
1	B	0	-1	2
2	C	0	4	3
3	D	0	7	-1
4	E	2	-1	5
5	F	2	-1	6
6	G	2	-1	-1
7	H	3	-1	-1

Example

	element	parent	leftmost child	right sibling
0	A	-1	1	-1
1	B	0	-1	2
2	C	0	3	-1
3	D	2	-1	4
4	E	2	-1	-1
5	F	-1	6	-1
6	G	5	-1	-1

Tree Walks

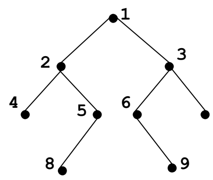
```
InOrder-Tree-Walk(x)
if  x ≠ NIL
    InOrder-Tree-Walk(left[x])
    print  key[x]
    InOrder-Tree-Walk(right[x])
```

```
PreOrder-Tree-Walk(x)
if  x ≠ NIL
    print  key[x]
    PreOrder-Tree-Walk(left[x])
    PreOrder-Tree-Walk(right[x])
```

```
PostOrder-Tree-Walk(x)
if  x ≠ NIL
    PostOrder-Tree-Walk(left[x])
    PostOrder-Tree-Walk(right[x])
    print  key[x]
```

Running Time: O(n)

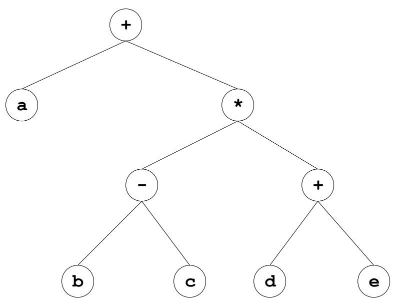
Example




```
PreOrder:    1 2 4 5 8 3 6 9 7
InOrder:     4 2 8 5 1 6 9 3 7
PostOrder:   4 8 5 2 9 6 7 3 1
```


Example:

Arithmetic Tree:




```
PreOrder:   + a * - b c + d e
InOrder:    ( (a) + ( (b-c) * (d+e) ) )
PostOrder:  a b c - d e + * +
```