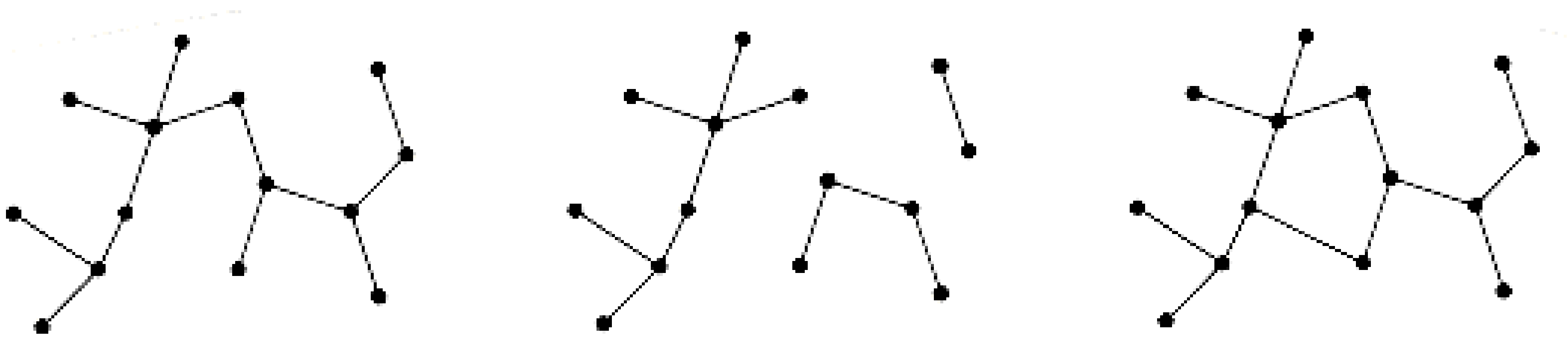
Tree



(a) (b) (c)

Tree

Free tree (graph theory)

- any two vertices are connected
- no cycles

Rooted tree

- ***+ root*** : one of the nodes is distinguished from the others

Ordered tree (most used in computer science)

- is a rooted tree in which the children of each node are ordered

  if a node has k children, then there is a first child, a second child, . . . , and a k-th child

Data Structure → rooted, ordered tree
(for us, by default)

Tree

recursive definition

Tree:

is empty

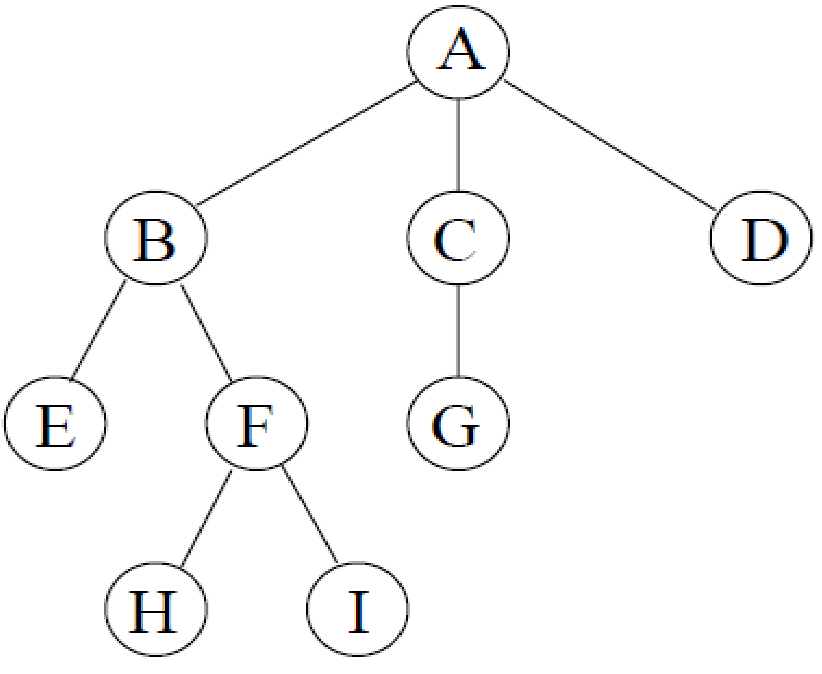
or it has a root **r** and 0 or more sub-trees

Properties:

- Each node has exactly one “*predecessor*” – its parent
- has exactly zero, one or more “*successors*” – its children

Trees

- root
- parent , children, sibling
- ancestor , descendants
- leaves , internal
- depth (level) , height
- degree



Tree

Node degree – the number of descendants

Node depth ***(level)***

- the length of the path to the root
- root – depth 0

Node height:

- the longest path from that node to a leaf (of the tree)
- (equivalent) the height of the subtree having that node as root

If the last edge on the path from the root *r* of a tree *T* to a node *x* is (*y, x*), then *y* is the ***parent*** of *x*, and *x* is a ***child*** of *y*.

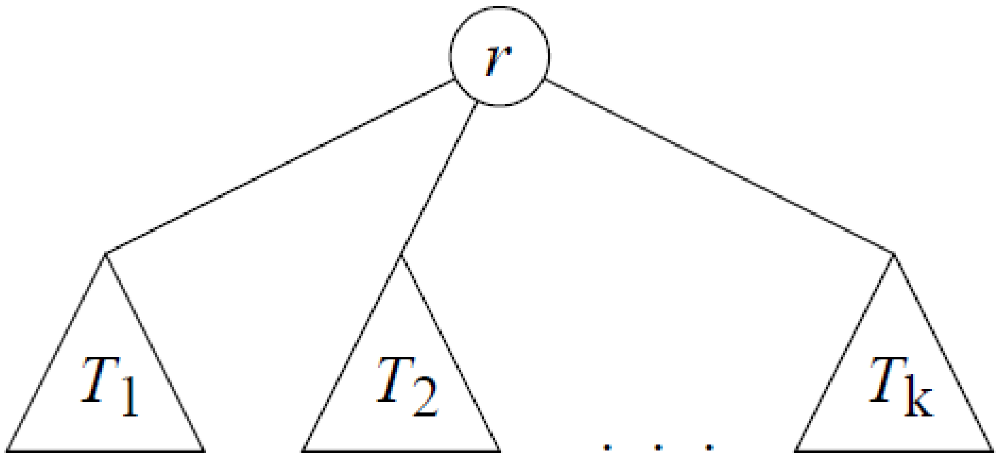
If two nodes have the same parent, they are ***siblings***.

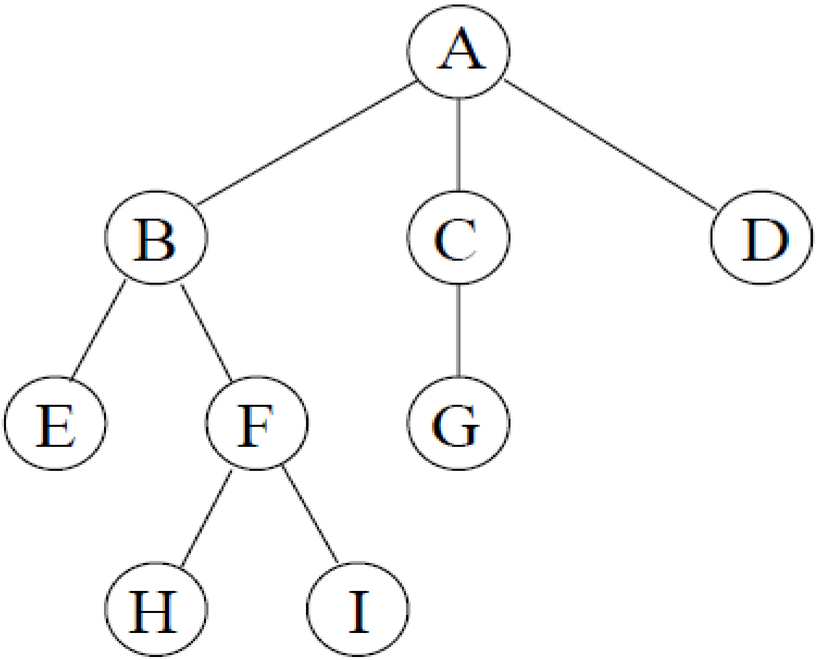
The root is the only node in *T* with no parent.

A node with no children is a ***leaf***. A non-leaf node is an ***internal node.***

k-ary tree

- A ***k-ary tree*** – each node have at most ***k*** descendants





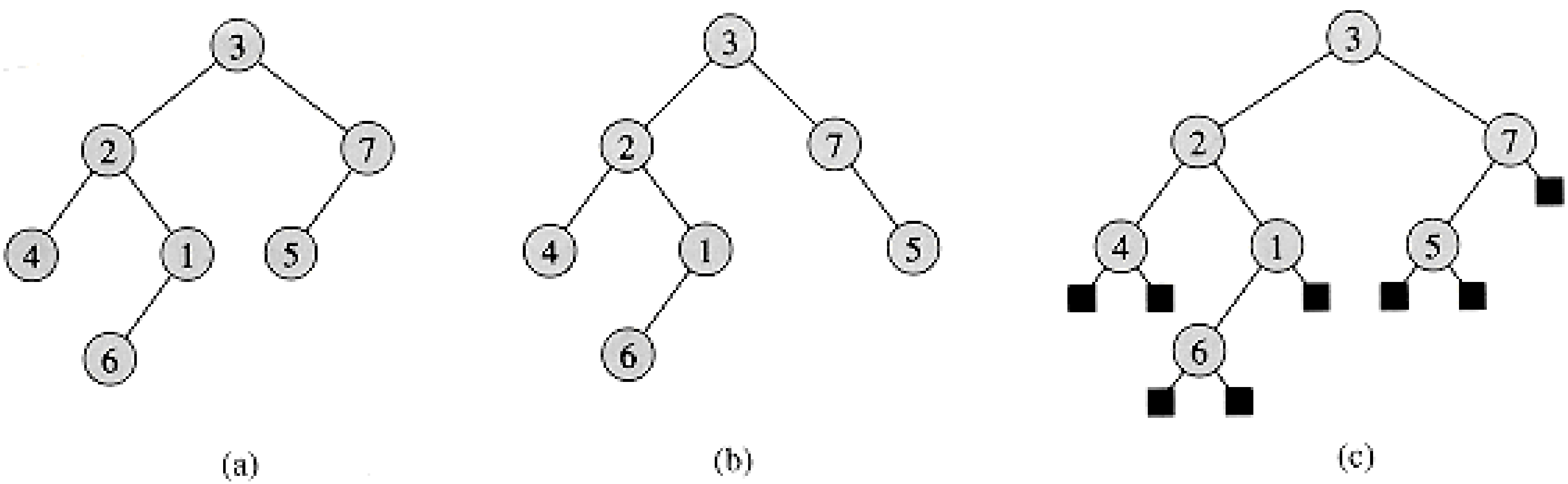
Binary trees

Rooted trees

each node have at most two descendants.

- first descendant is the left descendant
- second descendant is the right descendant

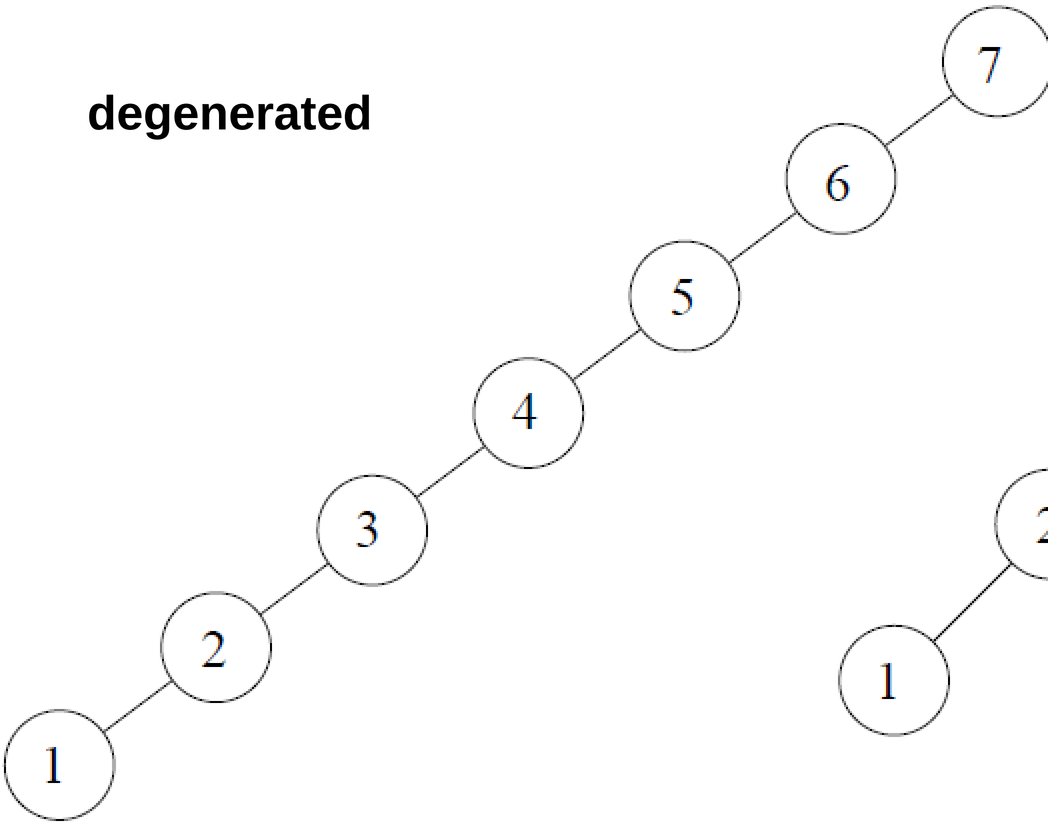
A tree with N nodes has N-1 edges



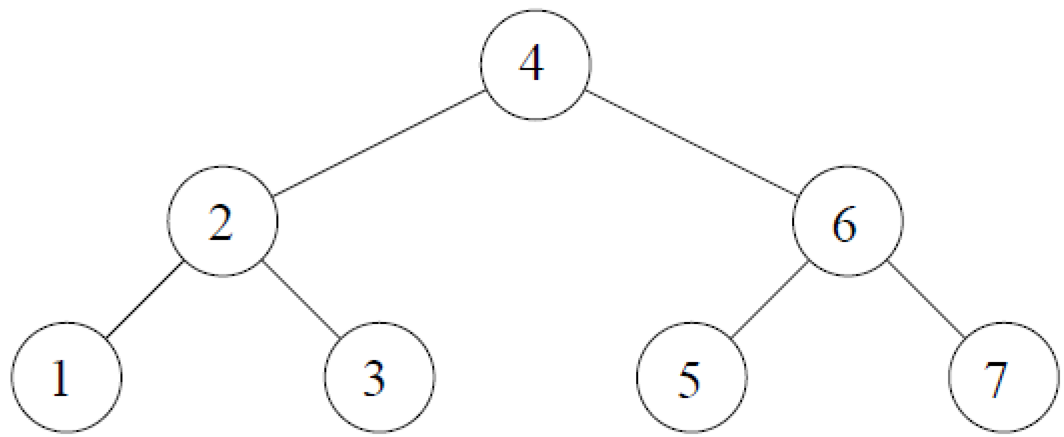
(a) (b) (c)

Binary trees

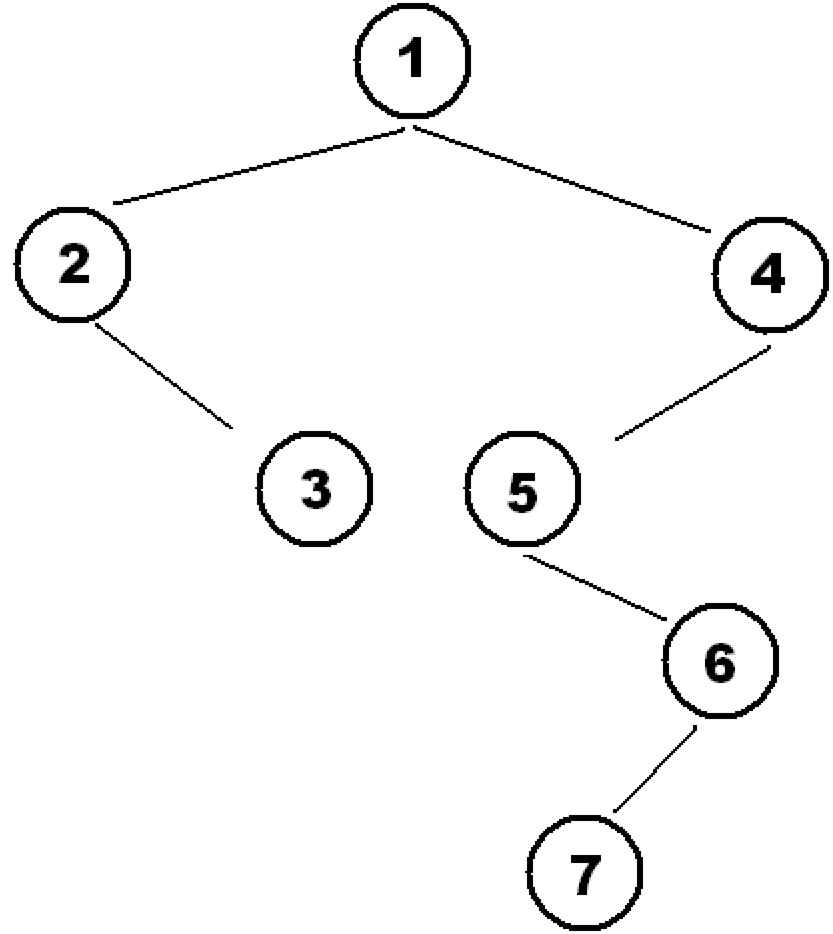
degenerated

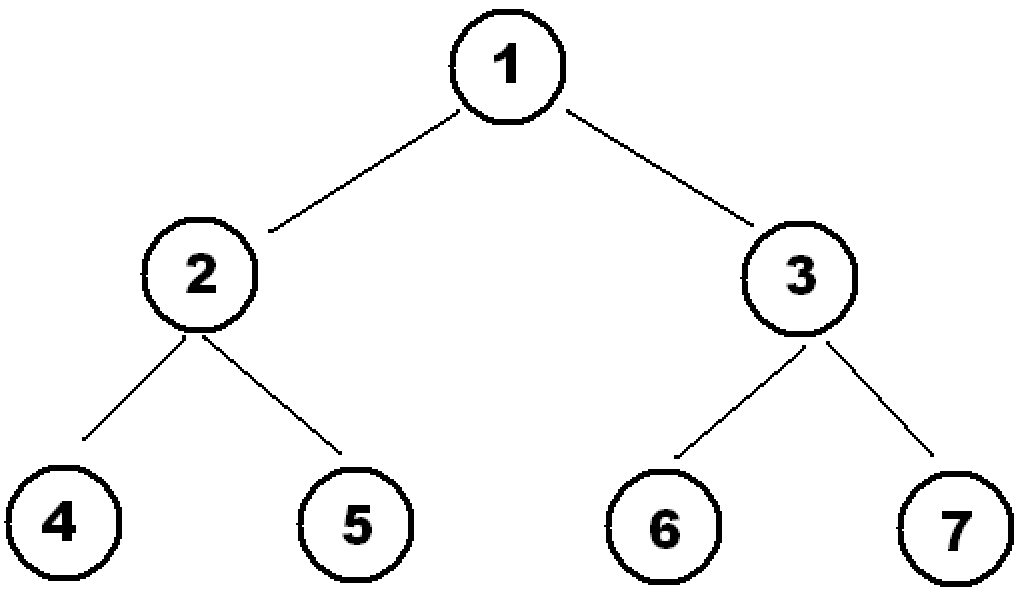


perfect



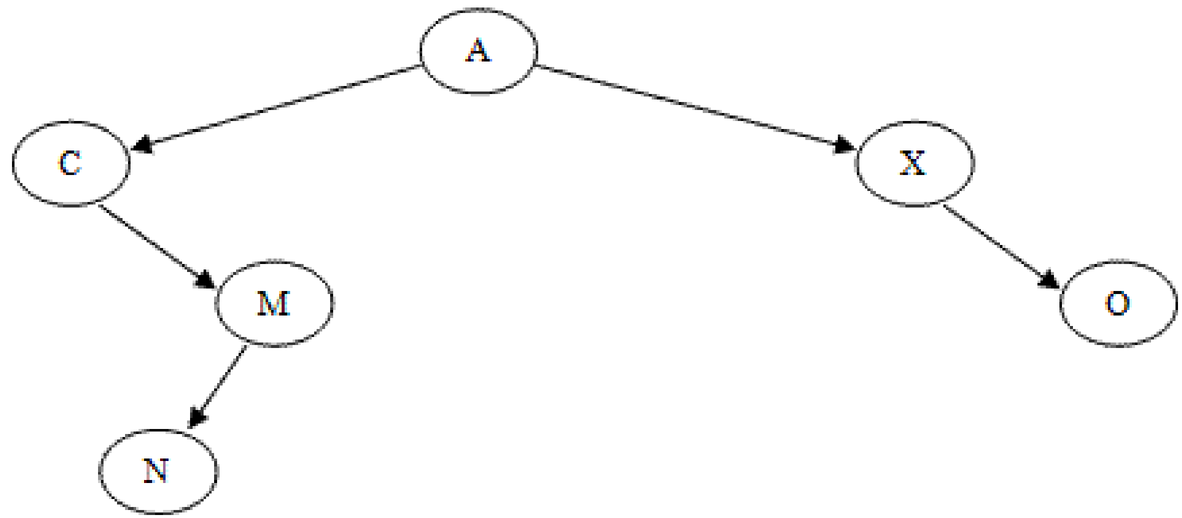
Binary trees



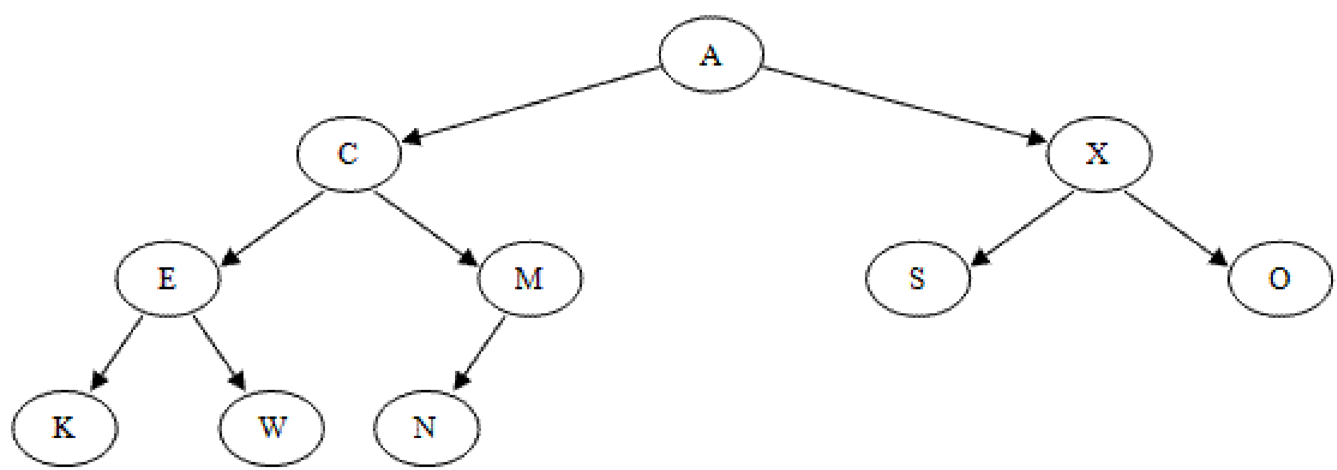


Binary trees

degenerated



(almost) complete



Binary tree types

Perfect tree:

- all leaves have the same depth
- and all internal nodes have two children

(Almost) **complete** tree:

- for each level , except possibly the deepest, the nodes have 2 children
- in the deepest level, all nodes are as far left as possible

A **degenerate** tree

- each parent node has only one child

➔ the tree will essentially behave like a linked list data structure

A **balanced binary tree**

- no leaf is much farther away from the root than any other leaf
  - different balancing schemes allow different definitions of "much farther

Binary tree types

(true or false ?)

Perfect tree:

A binary tree with all leaf nodes at the same depth.

All internal nodes have degree 2.

(Almost) **complete** tree:

A binary tree in which every level, except possibly the deepest, is completely filled.

At depth n, the height of the tree, all nodes must be as far left as possible.

http://xlinux.nist.gov/dads

(equivalent definitions)

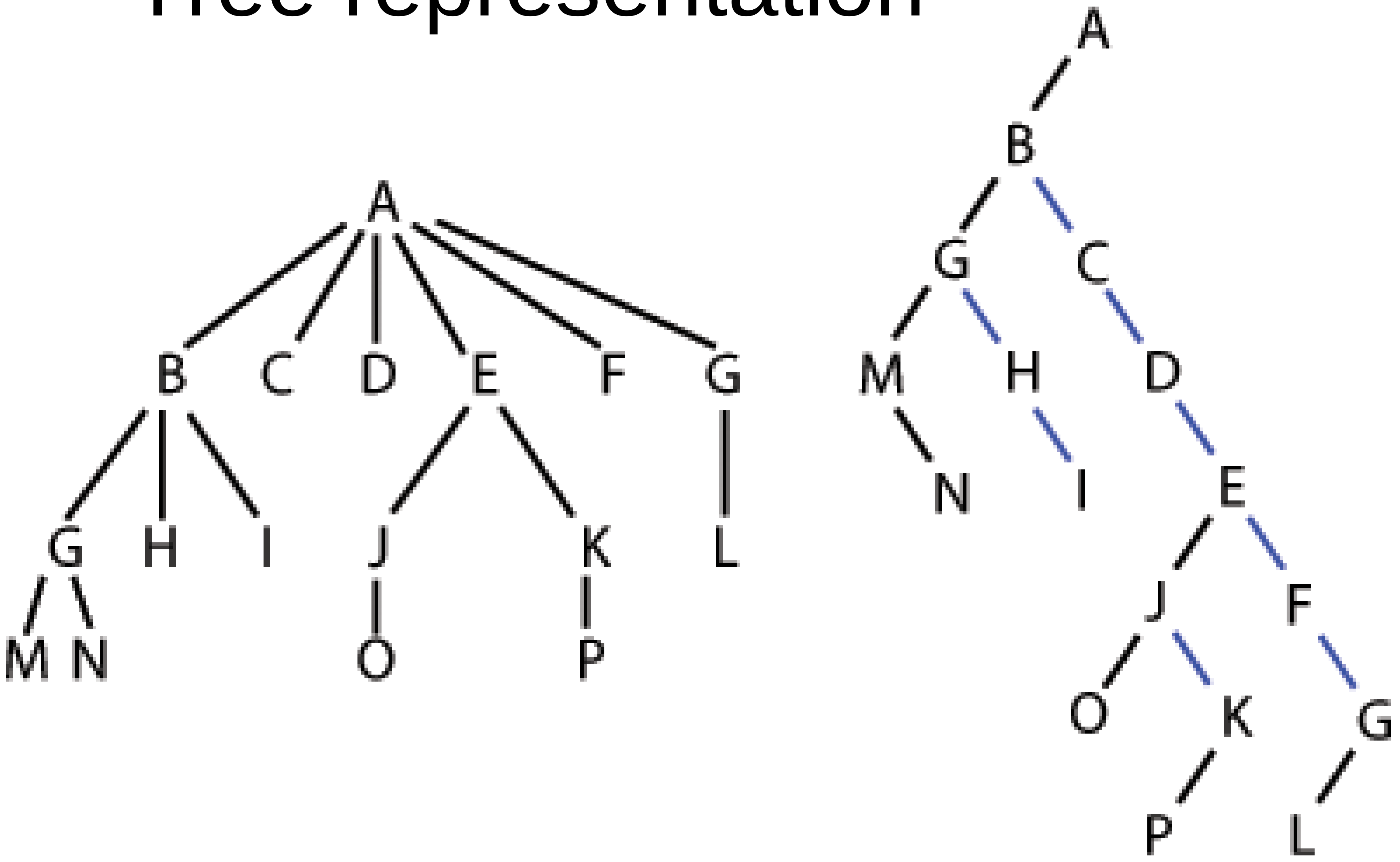
Binary tree properties

1. A tree with N nodes has N-1 edges

   (true for any tree)

2. No of nodes in a perfect binary tree with height h is $2^{h+1}-1$
3. Maximum no of nodes in a binary tree with height h is $2^{h+1}-1$
4. A binary tree with n nodes has height at least $[\log_2 n]$

Tree representation



Tree representation

(A (B (E (K, L) , F) , C (G) , D (H, I, J)))

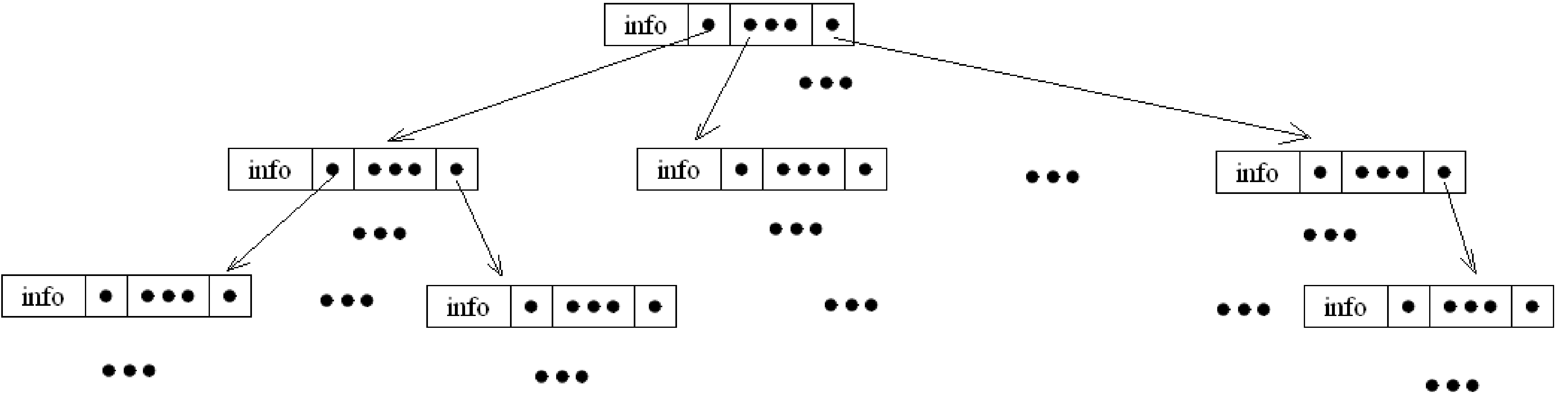
Tree representation (1)

Based on recursive definition

collection?

- Node   root information
  list of subTrees

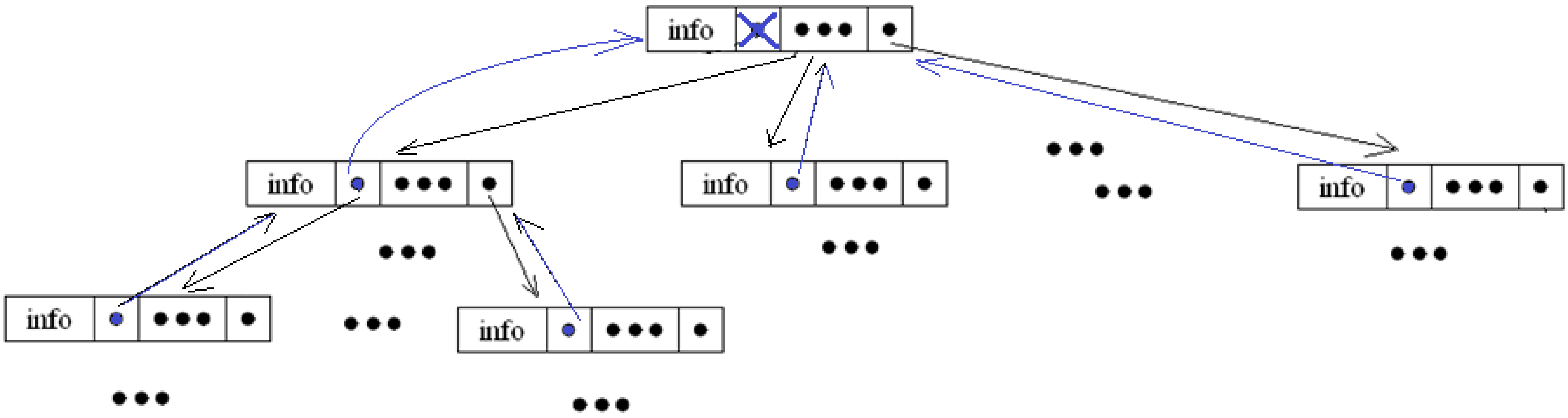
<u>remark</u>: a tree is known by knowing its root (links to subtrees)



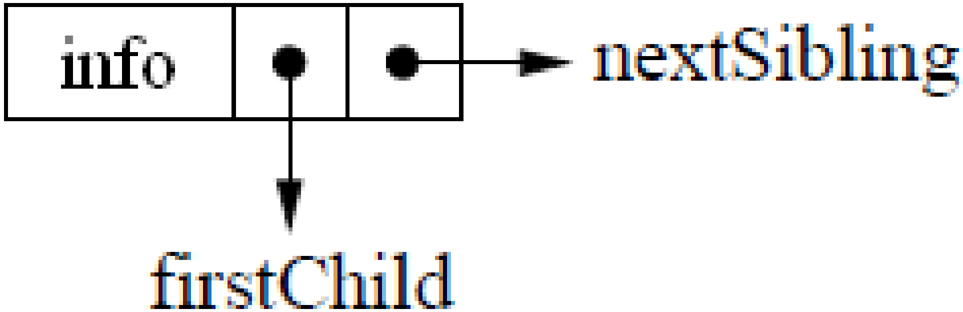
Linked representation (1)

Tree representation (1b)

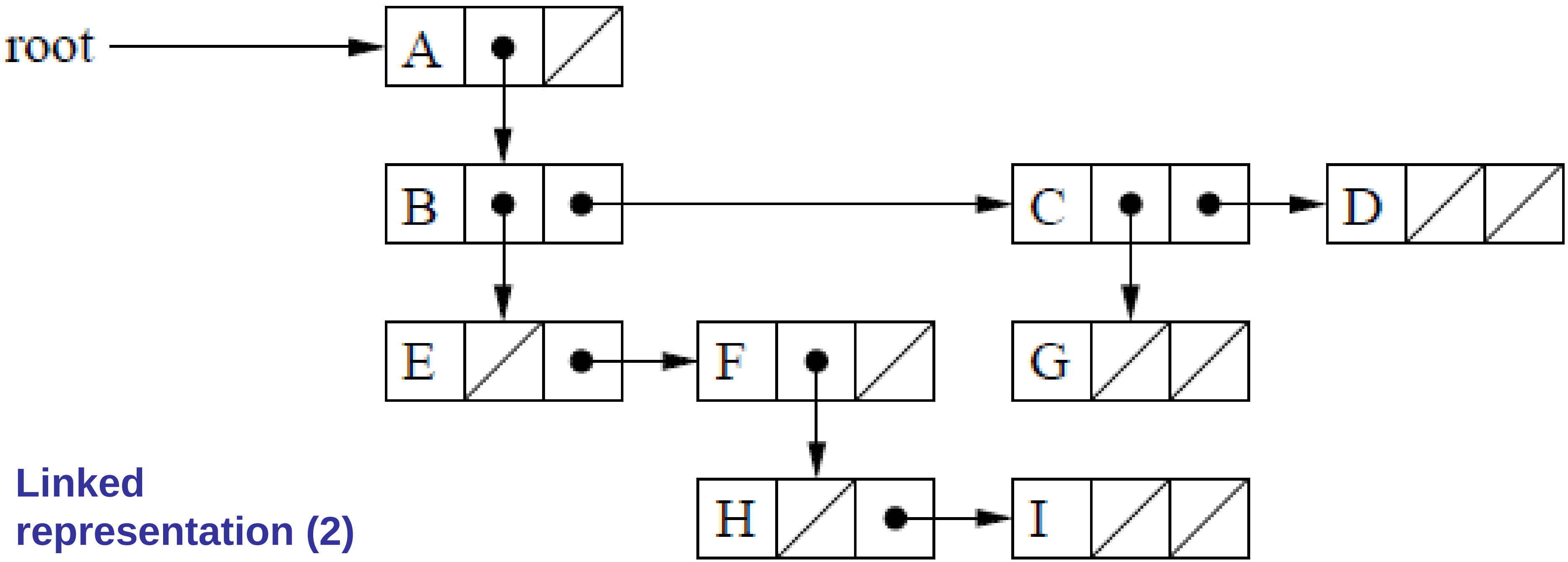
- Sometimes, a link to the parent node is also kept



Tree Representation (2)



=> sibling list



Linked representation (2)

Tree traversal

can be traversed in many ways

- depth-first traversal
- breadth-first traversal

  on levels

Representation (1) & dynamic allocation

```
TreeNode: record
            info: TElement
            left: Position
            right:Position
      end
Position: ^TreeNode
Tree: record
            root: Position
      end
```

Representation (1) & dynamic allocation

```
TreeNode: record
        info: TElement
        left: ^TreeNode
        right: ^TreeNode
    end

Tree: record
        root: ^TreeNode
    end
```

```
class TreeNode {
private:
    TElement info;
    TreeNode* left;
    TreeNode* right;
public:
    TreeNode(TElement value) {
        this->info = value;
        left = NULL;
        right = NULL;
    }
        …
};
class BinaryTree {
private:
    TreeNode* root;
public:
        …
```

Representation (1b) & dynamic allocation

```
TreeNode: record
        info: TElement
        left: ^TreeNode
        right:^TreeNode
    end
Tree: ^TreeNode
```

This representation fits the recursive definition of binary tree.

For some recursive algorithms, we are going to use this representation.

Representation (1) & over arrays

```
TreeNode: record
        info: TElement
        left: Integer
        right: Integer
   end

Tree: record
      root: Integer
      nodes: array [1..MAX] of TreeNode
         // …  information needed for freespace management
   end
```

Variations:

- using 3 arrays: Infos, Lefts, Rights
- over a dynamic vector