Data Structures II
Trees

Gianmaria Silvello

Department of Information Engineering
University of Padua
gianmaria.silvello@unipd.it
http://www.dei.unipd.it/~silvello/

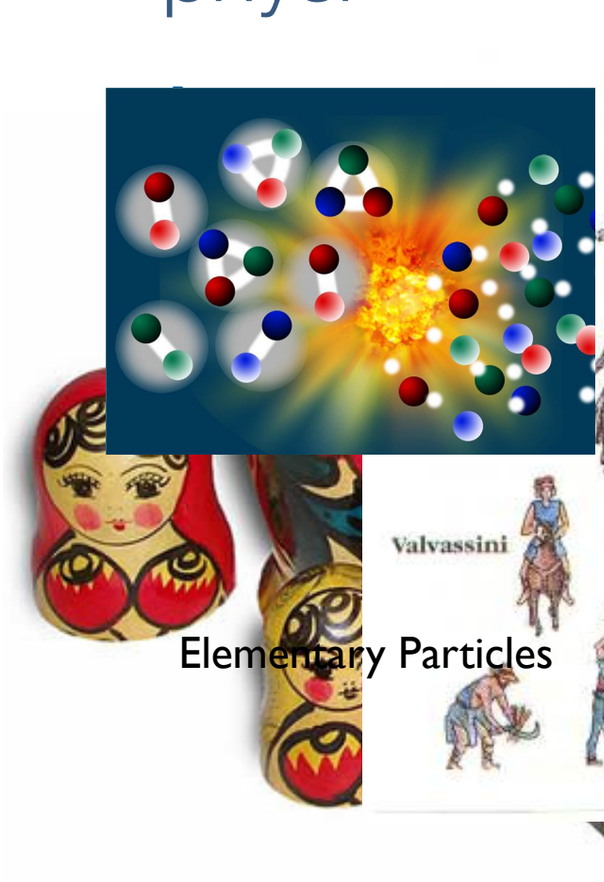
Outline

- Intuition
- Free trees
- Unordered and ordered trees
- Tree ADT
- Tree basic algorithms
- Binary trees
- Tree traversal algorithms
- Binary Search Trees

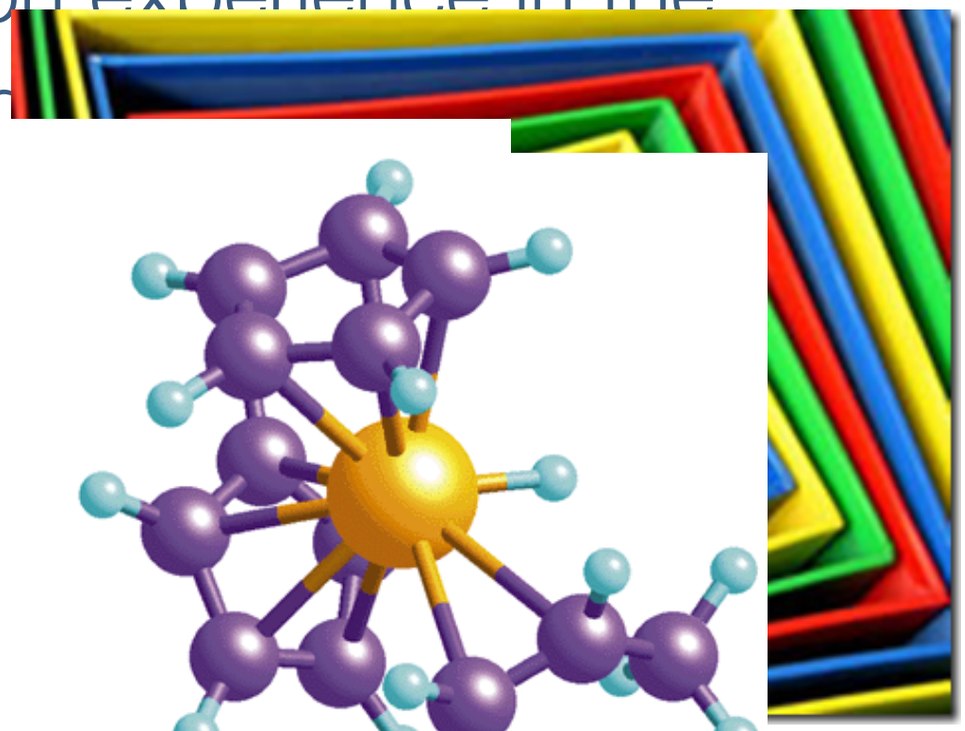
Reference: Chapter 8
of Goodrich, Tamassia and Goldwasser

Hierarchies

- Hierarchies are part of our common experience in the physical as well as in the living and



Elementary Particles



fu

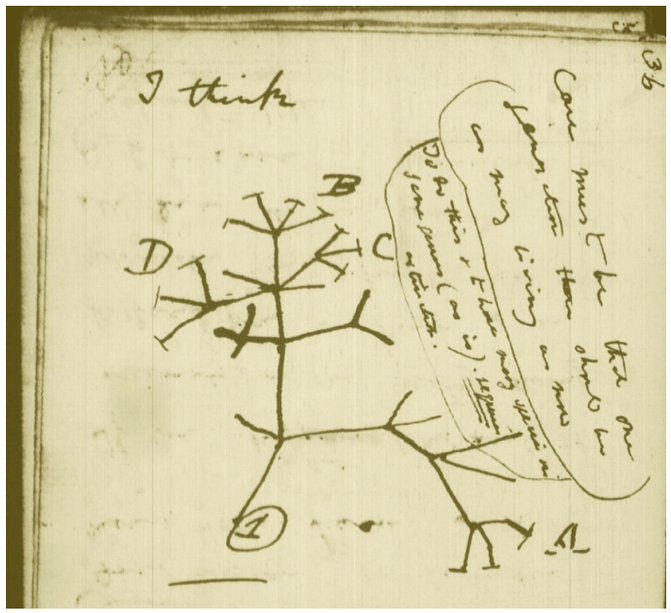
Atoms

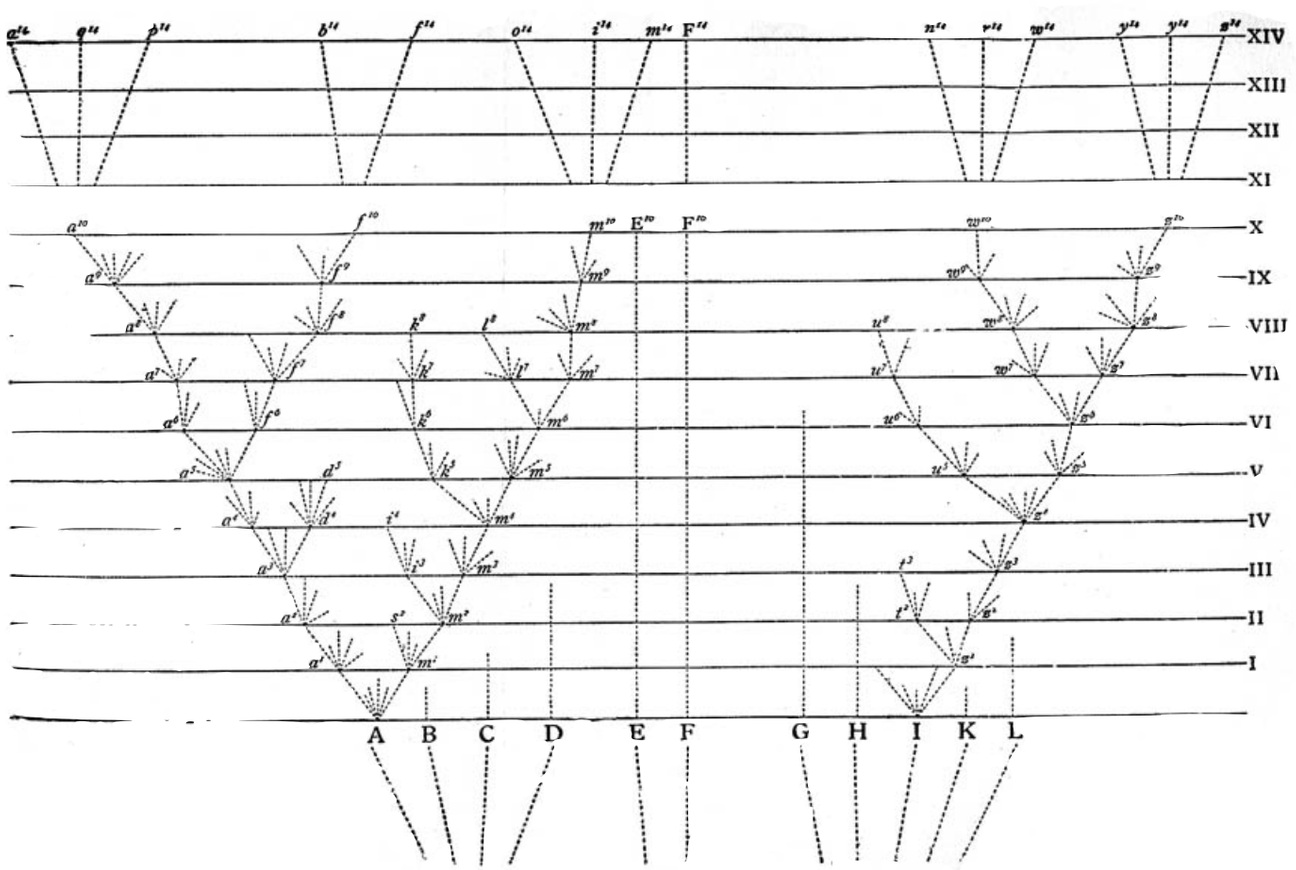
Molecules

Level hierarchy: Each level is characterized by a particular spatiotemporal scale for its associated entities and for the processes through which the entities at this level interact with one another.

The Tree

- The concept of hierarchy is at the highest level of abstraction.
- The common way to represent a hierarchy in many different scientific and non-scientific fields is by means of the tree structure.





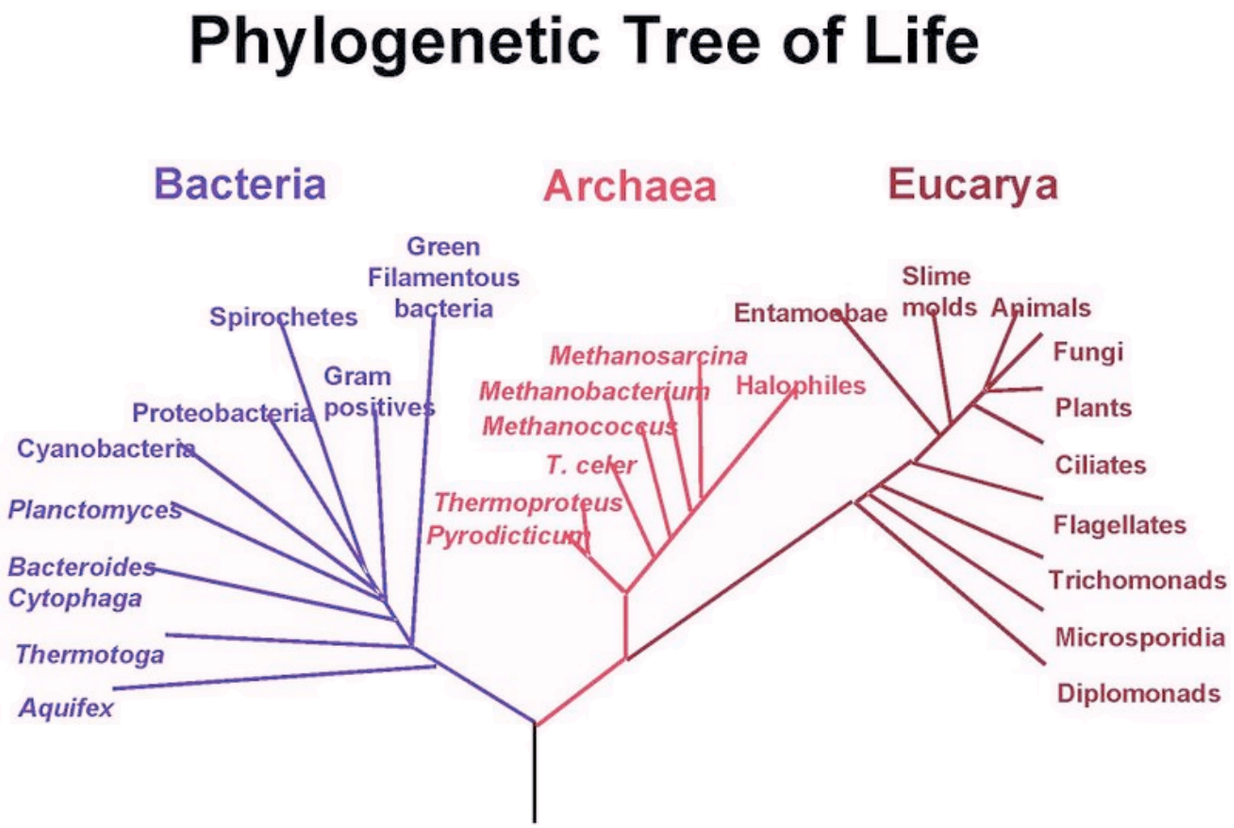
Darwin's Evolutionary Tree [Darwin1859].

"It is an odd looking affair, but *indispensable* to show the nature of the very complex affinities of past and present animals" [Moretti05].

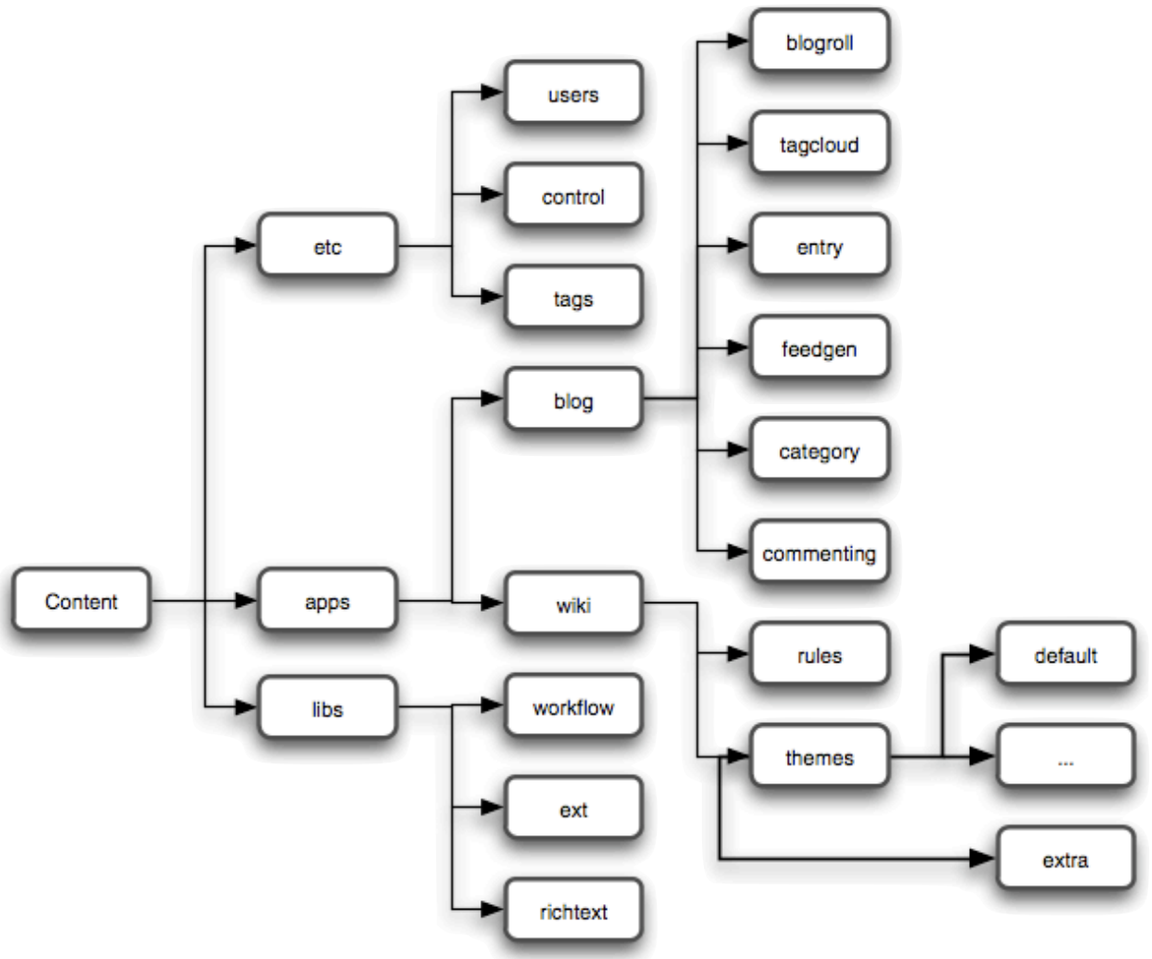
[Darwin1859] Darwin, C., "The Origin of Species". Barnes & Noble Classics, New York, NY,USA, 1859.

[Moretti05] Moretti, F., "Graphs, Maps, Trees – Abstract Models for Literary History". Verso, New York, NY, USA, 2005.

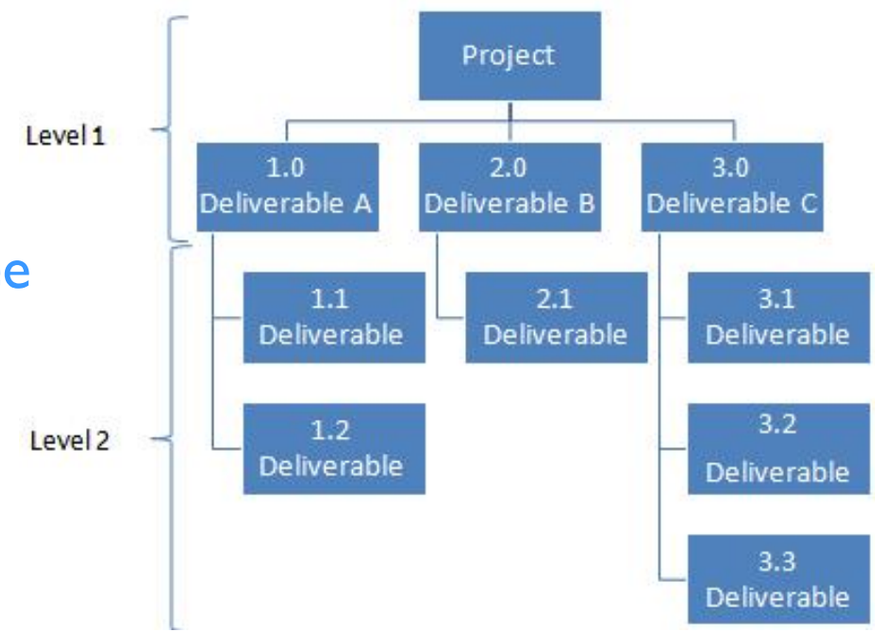
The Tree



File System Tree



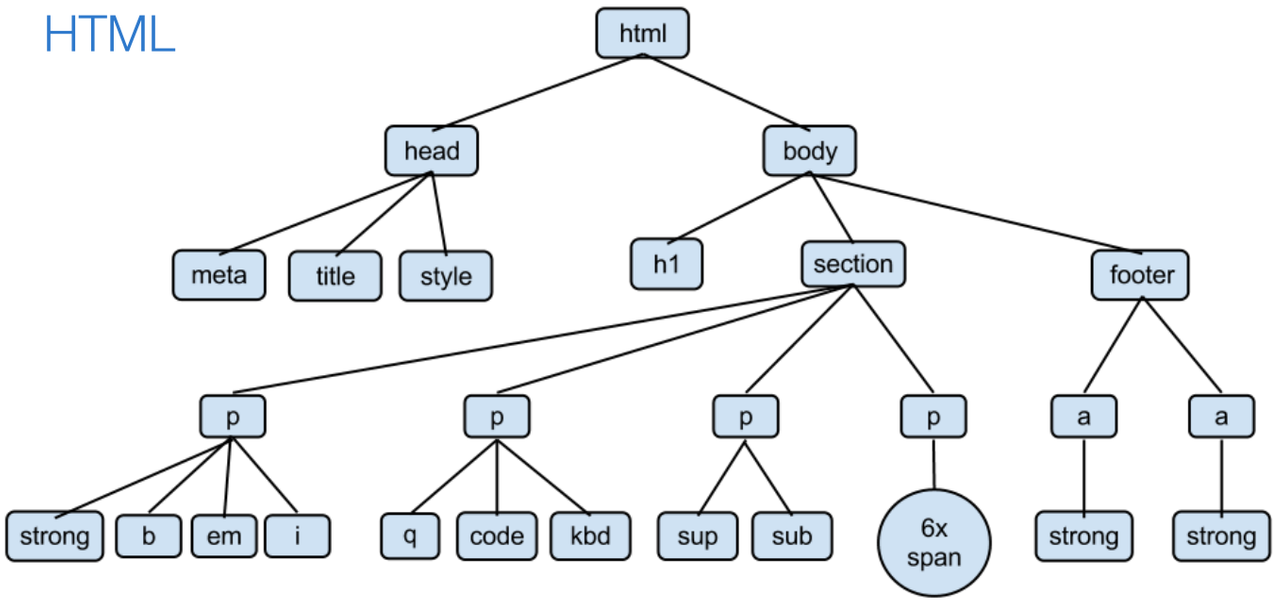
Organisation Tree



The Tree

- A tree is a collection of nodes connected by edges

HTML

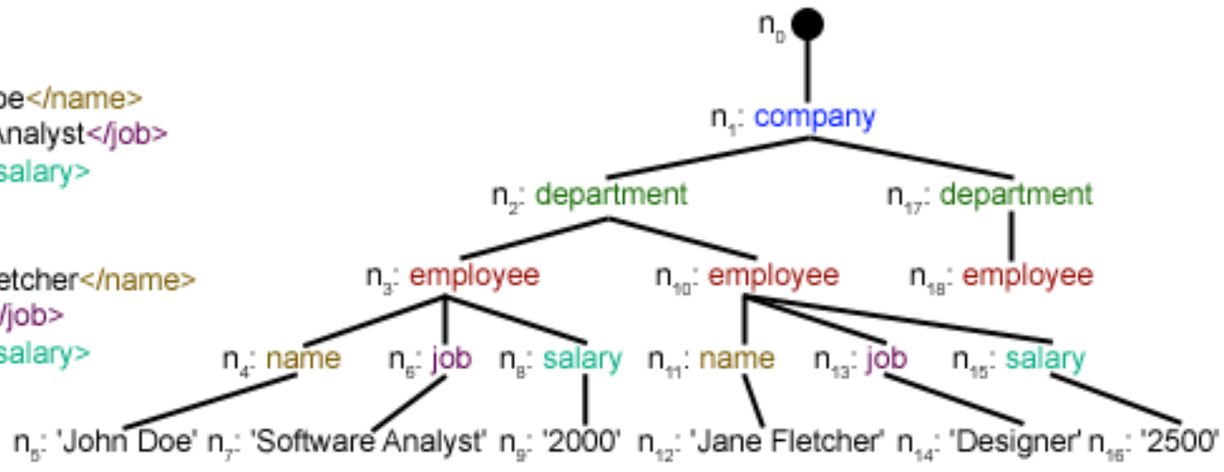


JSON

```
[
    {
        "name": "Top Level",
        "parent": "null",
        "children": [
            {
                "name": "Level 2: A",
                "parent": "Top Level",
                "children": [
                    {
                        "name": "Son of A",
                        "parent": "Level 2: A"
                    },
                    {
                        "name": "Daughter of A",
                        "parent": "Level 2: A"
                    }
                ]
            },
            {
                "name": "Level 2: B",
                "parent": "Top Level"
            }
        ]
    }
]
```

XML

```
<company>
 <department>
  <employee>
   <name>John Doe</name>
   <job>Software Analyst</job>
   <salary>2000</salary>
  </employee>
  <employee>
   <name>Jane Fletcher</name>
   <job>Designer</job>
   <salary>2500</salary>
  </employee>
 </department>
 <department>
  <employee>
  </employee>
 </department>
</company>
```



The Tree Data Structure

- The tree representation is formally defined as a proper data structure and it is one of the most important data structure in computer science [Knuth97].

Formal Definition in Graph Theory [Christofides75]

Let $V = \{v_1, \ldots, v_n\}$ be a set of nodes and the set E is a mapping of the set V in V, $E : V \rightarrow V$, thus $T(V, E) = T(V, V \times V)$; E is defined as a set of couples $\{v_i, v_j\}$ where $v_i, v_j \in V$ such that v_i is connected to v_j and thus v_i is the parent of v_j. If $T(V, E)$ is:

(i) A connected graph of n verteces and $(n - 1)$ links,

or (ii) A connected graph without a circuit,

or (iii) A graph in which every pair of verteces is connected with one and only one elementary path,

Then $T(V, E)$ is a **tree**.

[Knuth97] Knuth, D., "The Art of Computer Programming", Volume 1, Addison Wesley, 1997.

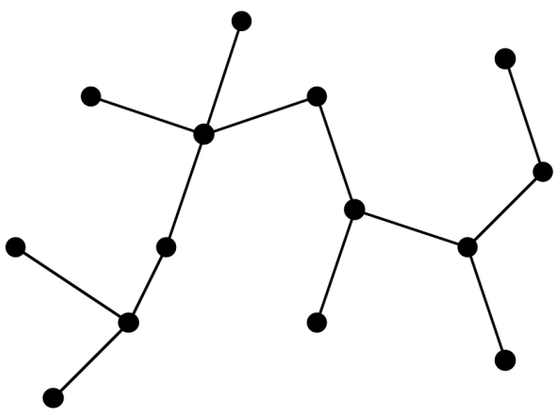
[Christofides75] Christofides, N., "Graph Theory", Academic Press, Imperial College, London, 1975.

Free trees

A graph G is a pair (V, E), where V is a finite set of vertices (nodes) and E is a binary relation on V (the set of edges).

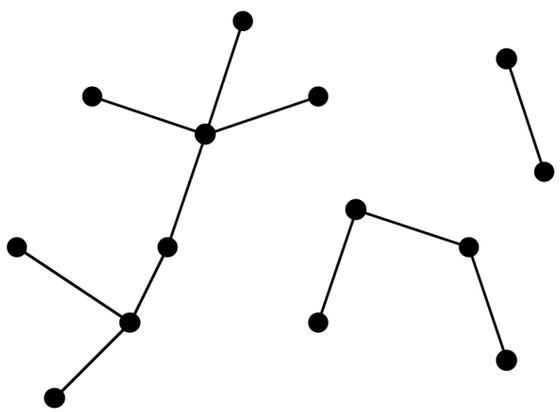
A free tree is a connected, acyclic, undirected graph.

If a graph is acyclic but possibly disconnected, it is a **forest**.



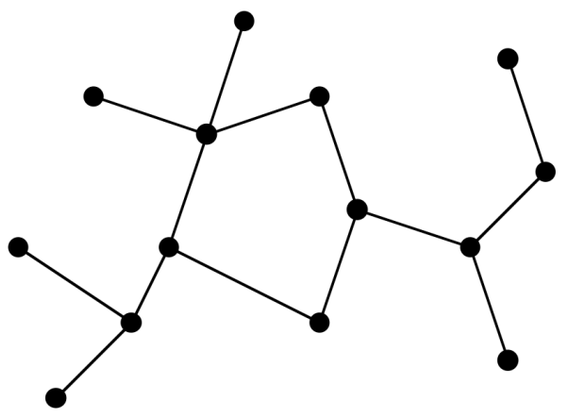
(a)

Free tree



(b)

Forest



(c)

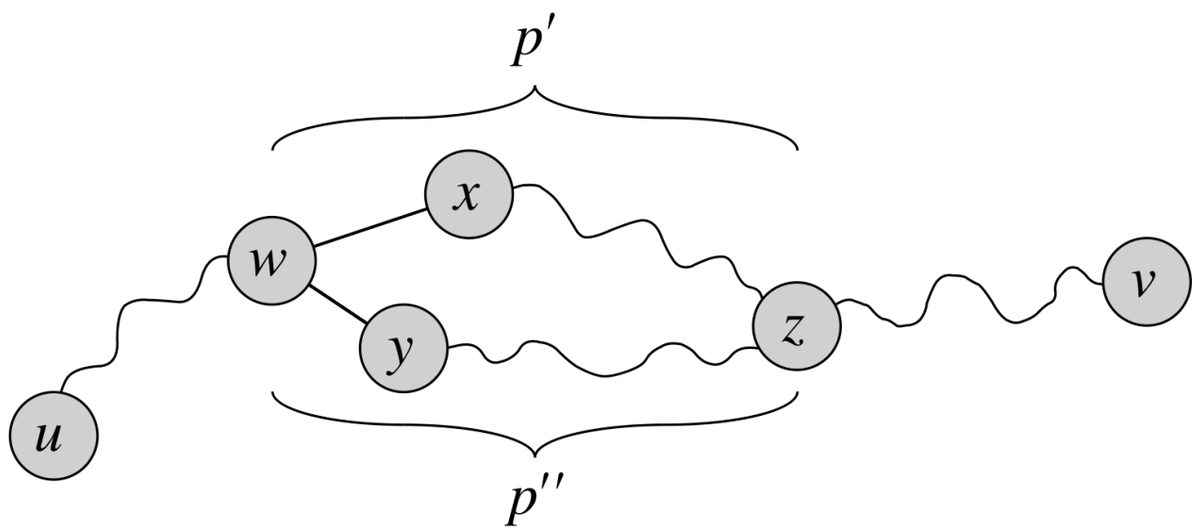
Not a tree

Properties of free trees

Let $G = (V,E)$ be an undirected graph. The following statements are equivalent:

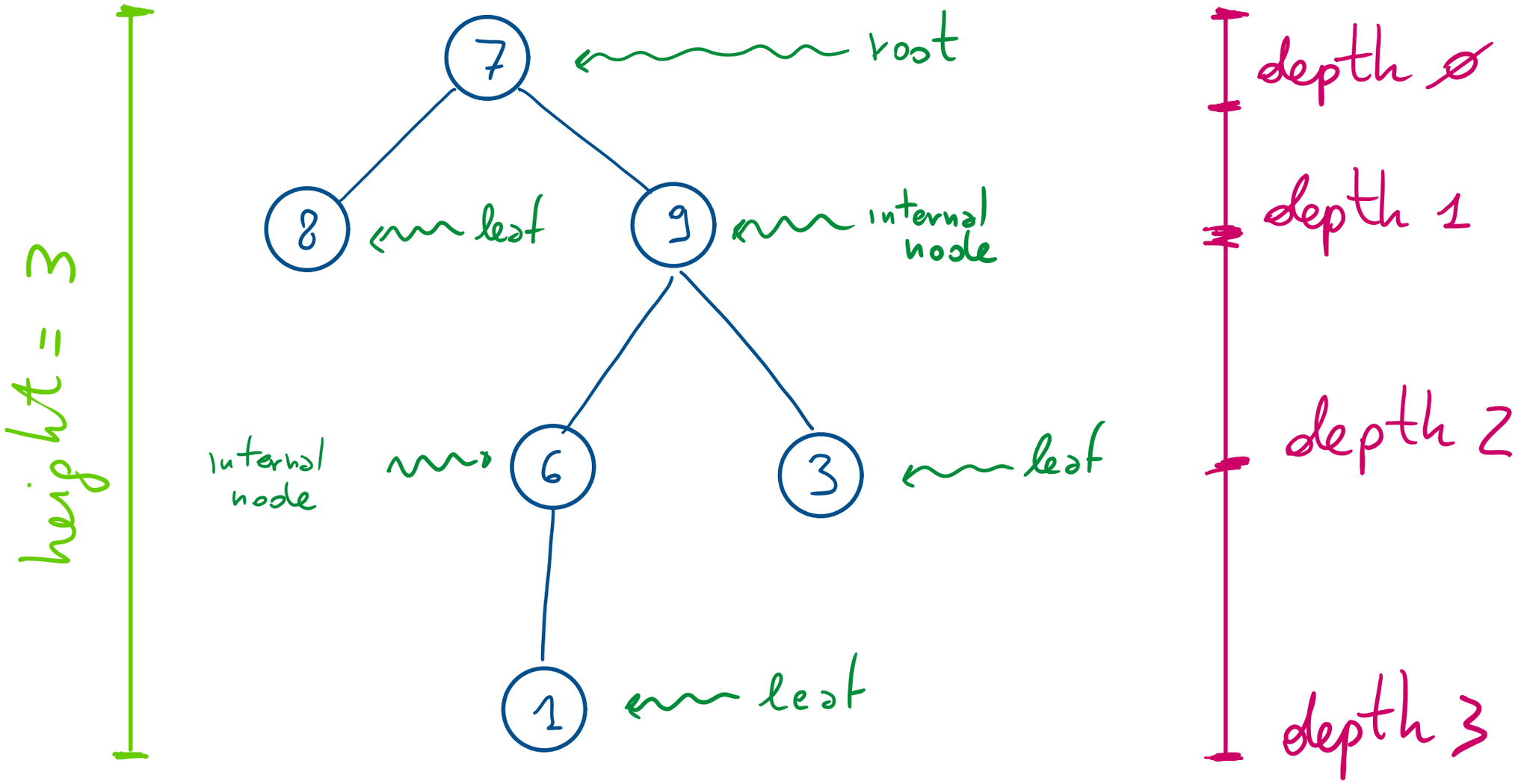
1. G is a free tree.
2. Any two vertices in G are connected by a unique simple path.
3. G is connected, but if any edge is removed from E, the resulting graph is disconnected
4. G is connected and $|E| = |V| - 1$
5. G is acyclic and $|E| = |V| - 1$
6. G is acyclic, but if any edge is added to E, the resulting graph contains a cycle

Prove that 1->2, 2->3, 3->4 and so on

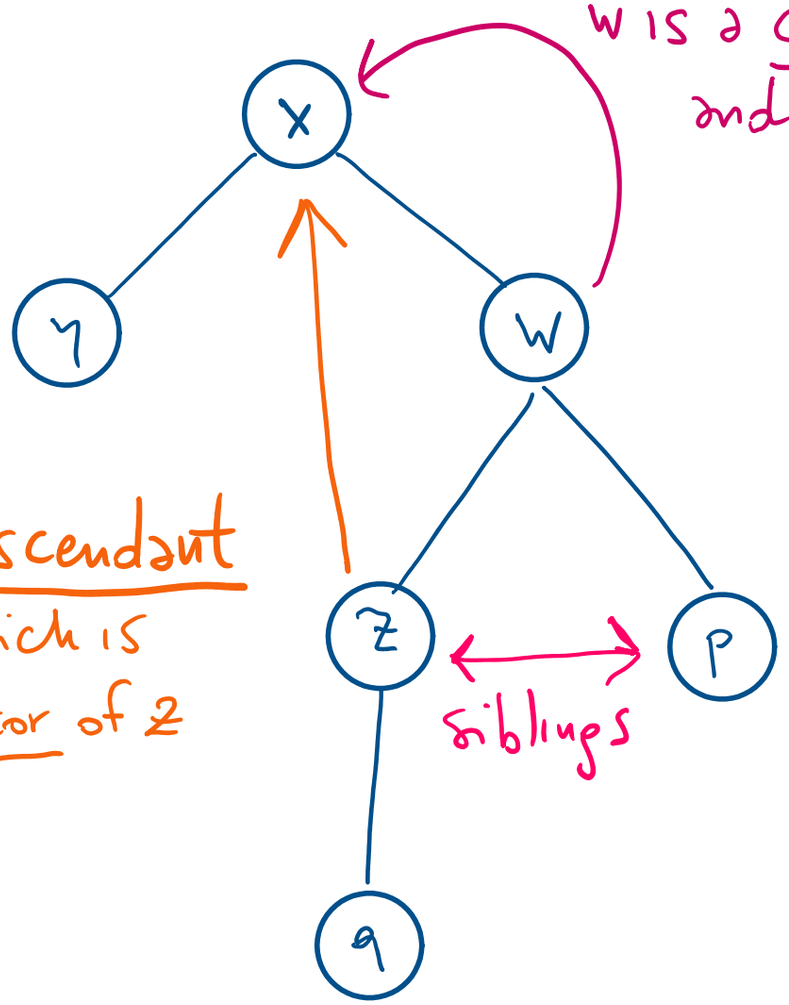


Rooted tree

- A rooted tree is a free tree where one of the nodes is called root



Rooted trees



w is a child of x
and x is the parent of w

z is a descendant of x which is an ancestor of z

Descendants $(x) = \{y, w, z, p, q\}$

Ancestors $(x) = \{\emptyset\}$

Ancestors $(q) = \{z, w, x\}$

Parent $(x) = \{\emptyset\}$

Children $(q) = \{\emptyset\}$ Children $(w) = \{z, p\}$

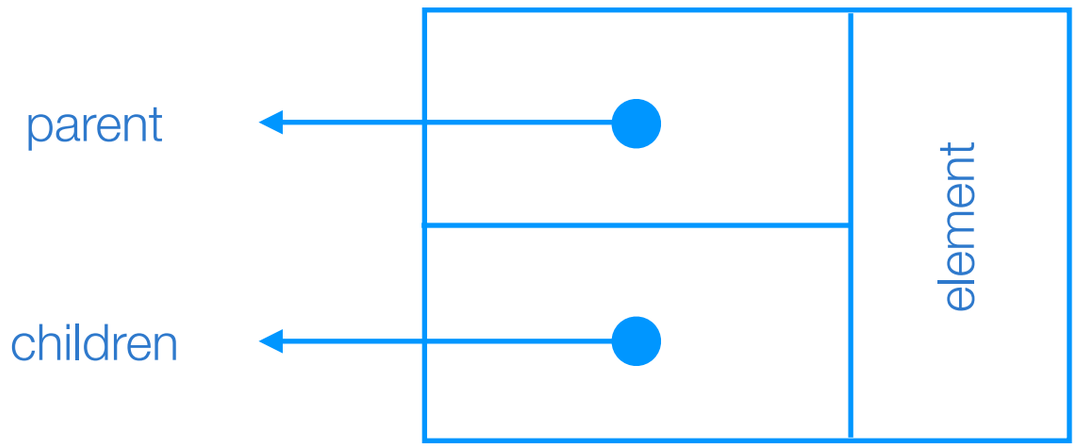
Rooted trees

- The *subtree* rooted at *x* is the tree induced by the descendants of *x* and *x* itself.

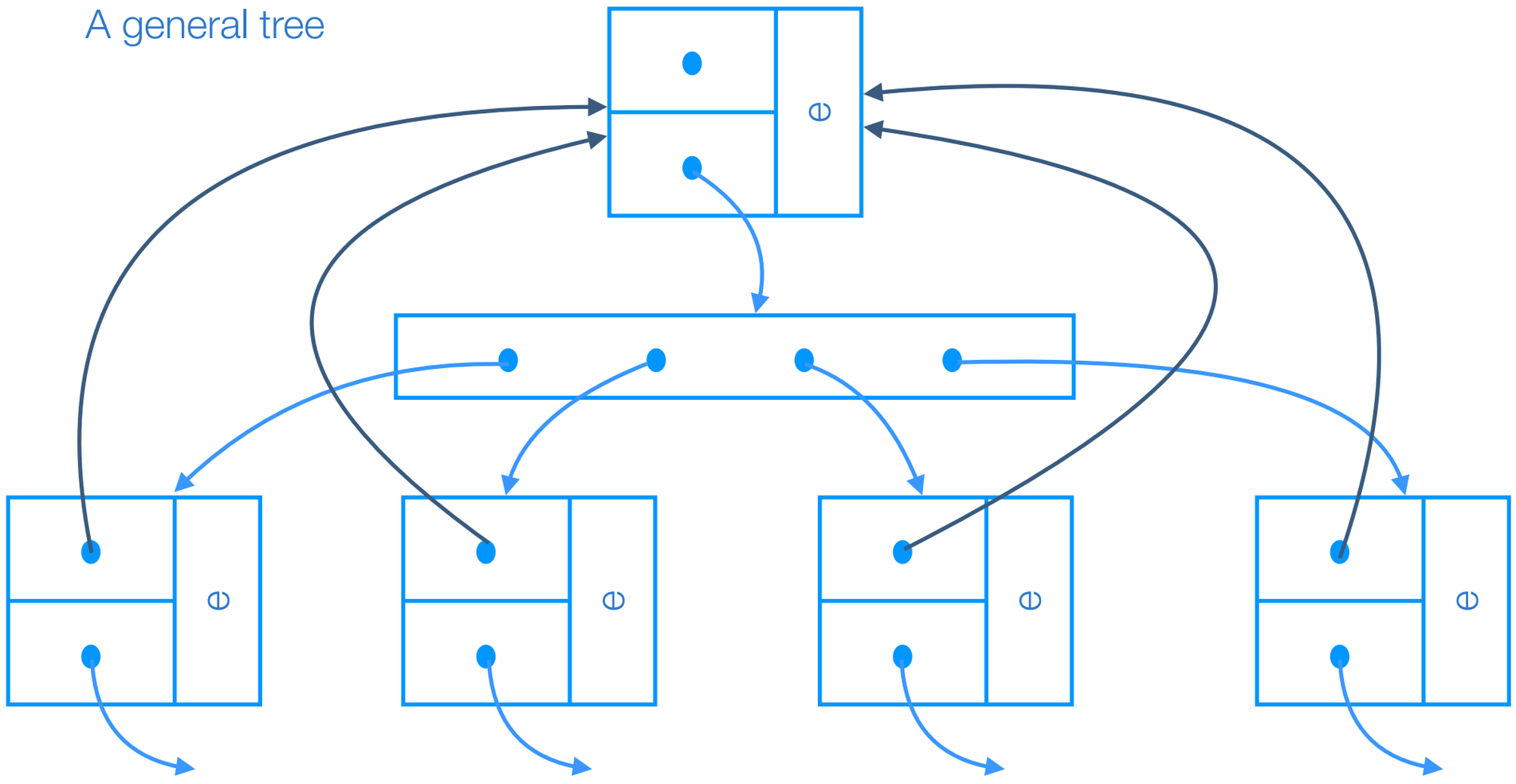
- The number of children of a node x is defined as the *degree* of x

- An ordered tree is a rooted tree in which the children of each node are ordered; so, if x has 3 children, say {y, w, z} we can say that y is the first child, w is the second child…

A tree as a linked structure



A general tree



General Tree ADT

- `p.element()`
  - Return the element stored at position *p*
- `T.root()`
  - Return the position of the root of the tree or *None* if it's empty
- `T.is_root(p)`
  - Return true if the node stored at position *p* is the root
- `T.parent(p)`
  - Return the position of the parent of the node stored in p
- `T.num_children(p)`

General Tree ADT

- `T.children(p)`
  - Generate an iteration of the children of position p
- `T.is_leaf(p)` or `T.isExternal(p)` (specular method: `T.isInternal(p)`)
  - Return true if p is a leaf
- `len(T)`
  - Return the number of nodes in T (the number of positions)
- `T.is_empty()`
  - Return true is the tree is empty
- `T.positions()` or `T.iterator()`

Depth of a tree

```
depth(T,p)
Input: p∈T (position of the node)
Output: depth of p in T

if(T.is_root(p)) then
    return 0
else
    return 1 + depth(T, T.parent(p))
```

```
depth(T,p)
Input: p∈T
Output: depth of p in T

d = 0
while (p = T.parent(p) is not none) do
    d = d + 1
return d
```

Worst-case

Tree with *n* nodes
structured as a “chain”

Running time: $\Theta(n)$

Height of a tree

Proposition: The height of a tree is equal to the maximum of the depths of its leaves

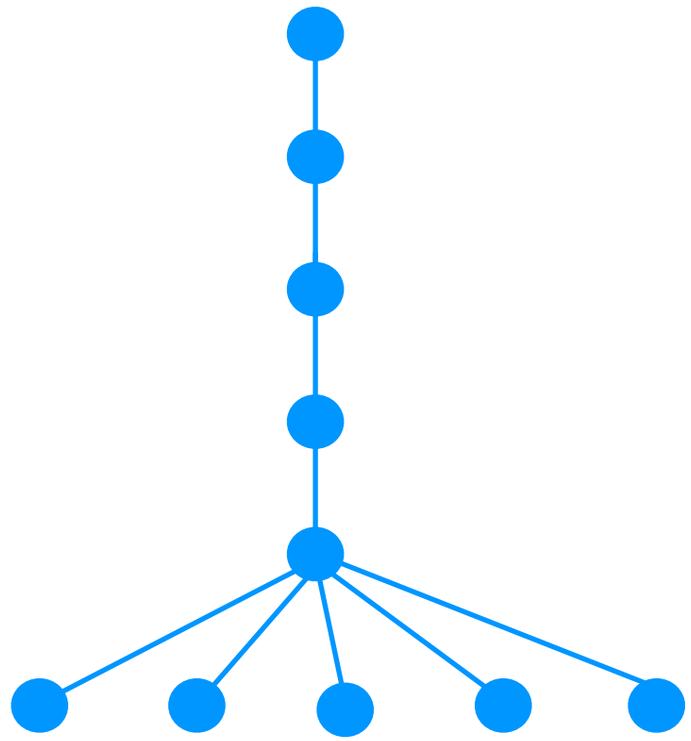
```
height(T)
Input: T (the tree)
Output: height of T

h = 0
for each v∈T.positions() do
    if (T.isLeaf(v)) then
        h = max(h, depth(T,v))
return h
```

`T.positions()` can be implemented in *O*(n)

Worst-case

This is called $O(\sum_{p \in L}(d_p + 1))$ times



Running time: we iterate over *n* nodes so O(n) + O(`depth`) for each leaf

n/2 leaves each one with depth n/2

So -> $\Theta(n^2)$

Height of a tree

Proposition: The height of a tree is equal to the maximum of the depths of its leaves

```
height2(T,p)
Input: p∈T (position of the node)
Output: height of p in T

if(T.isLeaf(p)) then
    return 0
else
    h = 0
    for each w∈T.children(p) do
        h = max(h, height2(T,w))
return h+1
```

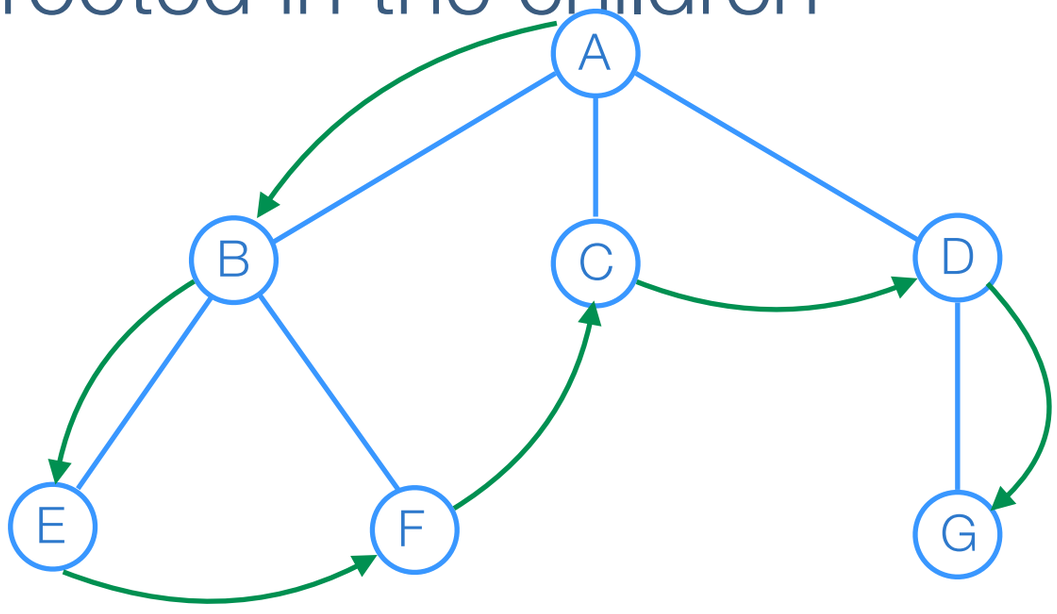
Worst-case

A chain with *n* nodes

Running time: $\Theta(n)$

Tree traversal: Pre-order

- Tree traversal is a systematic method to visit the nodes of a tree executing an operation (e.g. the visit) in each node
- **Pre-order**: you visit the parent and then the sub-trees rooted in the children




```
preorder(T,p)
    visit(p)
    for each c in T.children(p) do
        preorder(T,c)
```

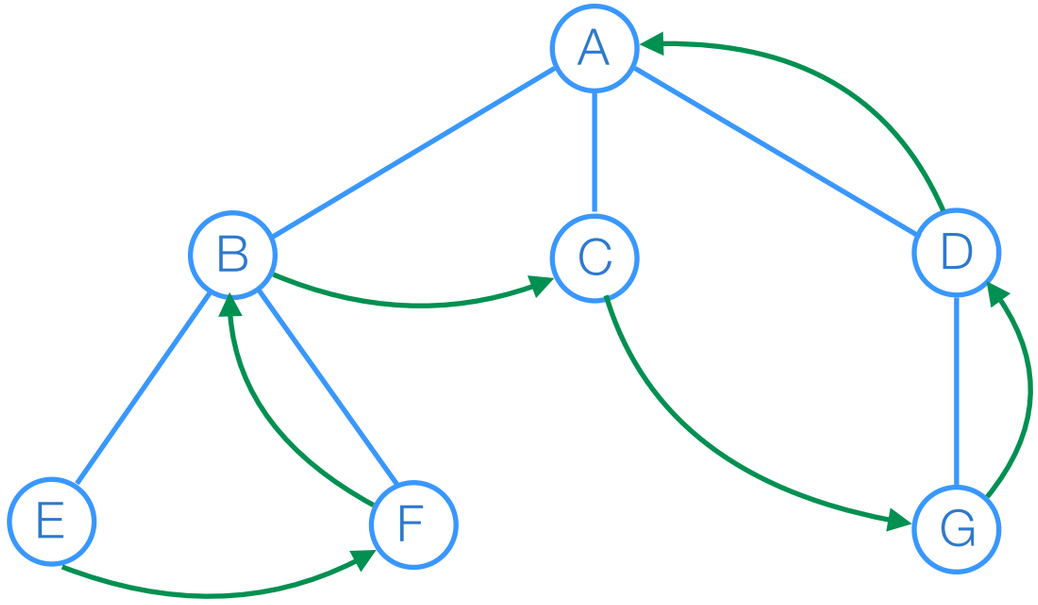
Pre-order: A, B, E, F, C, D, G

Running time: O(n)

Tree-traversal: Post-order

- **Post-order**: you recursively visit the sub-trees rooted in the children and then the parent

```
postorder(T,p)
    for each c in T.children(p) do
        postorder(T,c)
    visit(p)
```

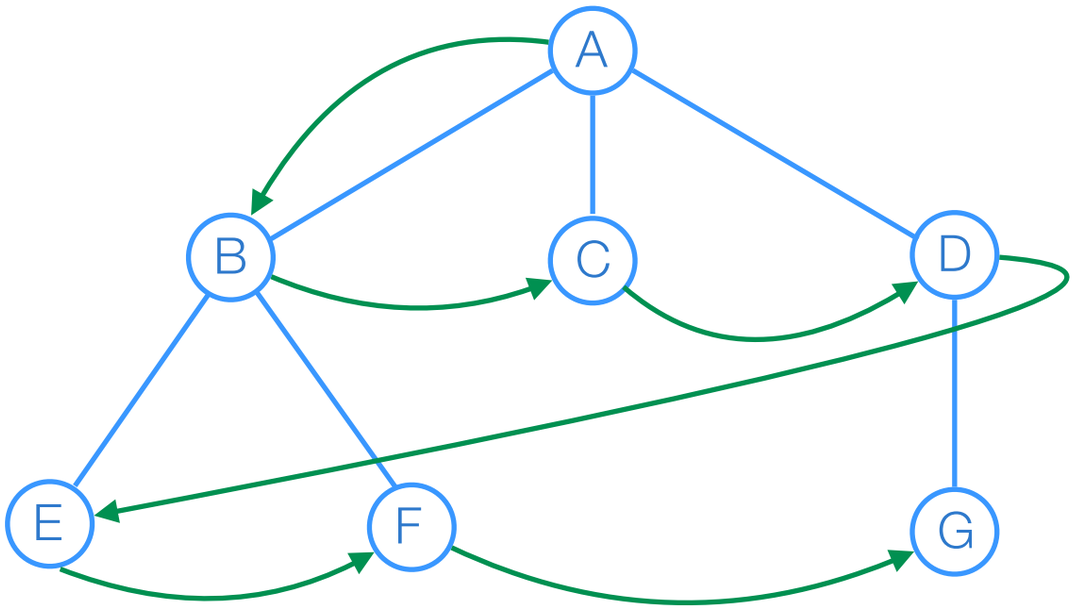


Post-order: E, F, B, C, G, D, A

Running time: O(n)

Tree traversal: Breadth First Traversal

- You visit all the nodes at depth *d* before visiting all the nodes at depth *d+1*




```
BFT(T,p)
    Q.enqueue(T.root())
    while !Q.isEmpty() do
        p = Q.dequeue()
        visit(p)
        for each c in T.children(p) do
            Q.enqueue(c)
```

Exercises

- Ex.1: Preorder visit

Ex. 1

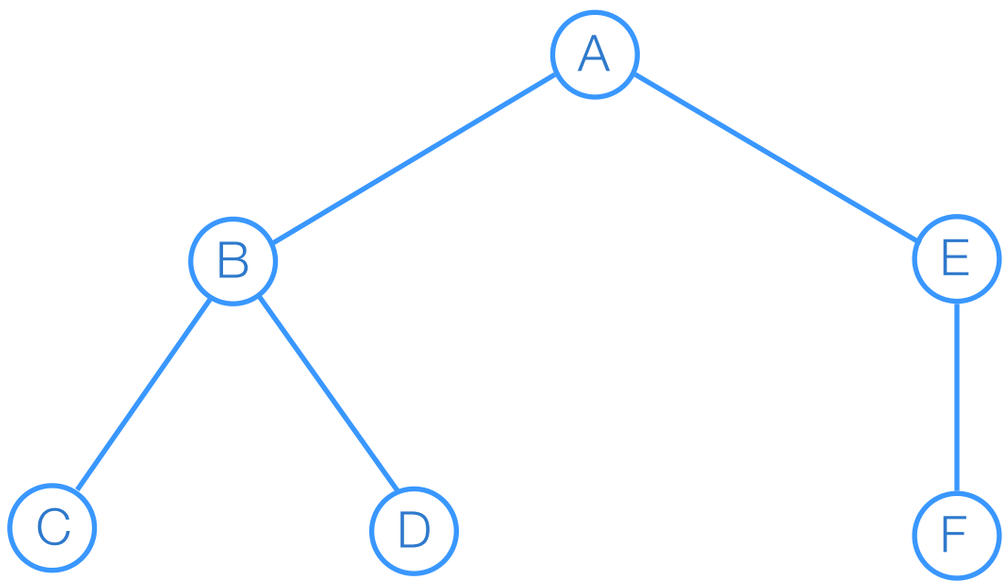
Preorder visit of a tree: A,B,C,D,E,F

Which of the followings can be a postorder visit of the same tree?

1. A,C,D,E,F,B
2. C,D,E,A,F,B
3. C,D,E,B,F,A ← The root is the last visited node

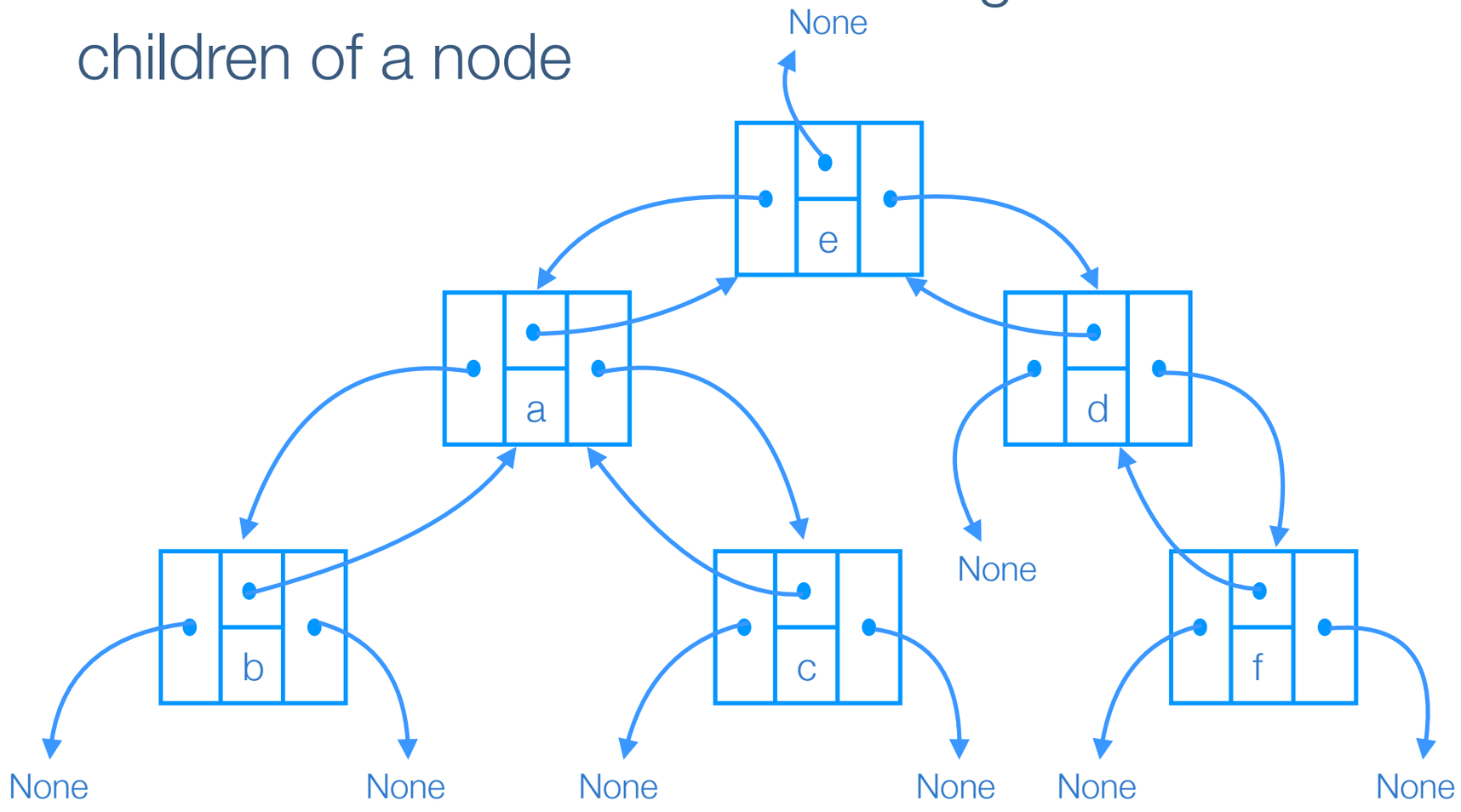
Ex. 2

Draw the tree given the preorder visit (A B C D E F) and the postorder visit (C D B F E A)



Binary tree

- A binary tree is an ordered tree such that:
  - Every node has at most 2 children;
  - Every child node is labeled as **right node** or **left node**
  - The left child comes before the right child in the order of children of a node



Binary tree: properties

- A binary tree is called **proper** (or *full*) if every node has zero or two children

Let T be a proper binary tree with n nodes (where n_E are external nodes and n_I are internal nodes), then:

1. $2h + 1 \leq n \leq 2^{h+1} - 1$
2. $h + 1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $lg(n + 1) - 1 \leq h \leq (n - 1)/2$
5. $n_E = n_I + 1$

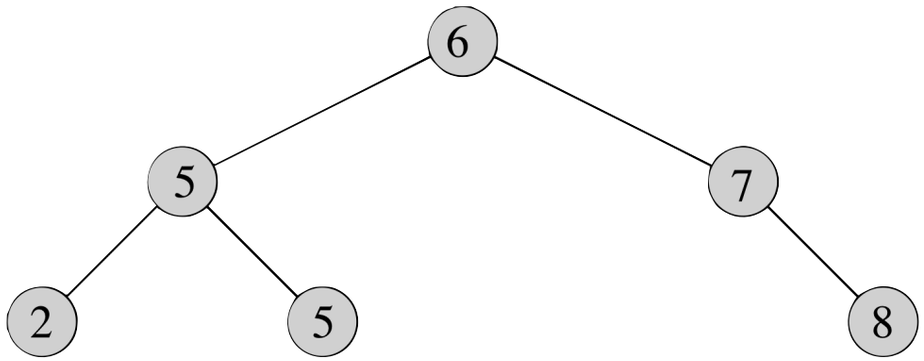
Binary Search Tree

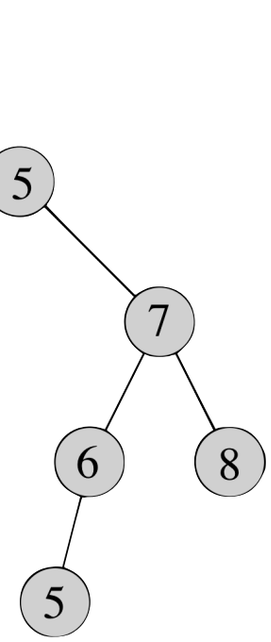
Binary Search Tree

- A binary search tree `T` is a binary tree where if `y` is a node in the left subtree of `x` then `y.key`$\leq$`x.key` and if `y` is a key in the right subtree of `x` then `y.key`$\geq$`x.key`
  - `x.key` is the same as `x.element()`
  - `x.left` (`x.right`) returns the position of the left (right) child of `x`
- In other words: is a rooted binary tree data structure whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree

Binary tree traversal: in order visit

- Can be seen as visit of the tree from left to right





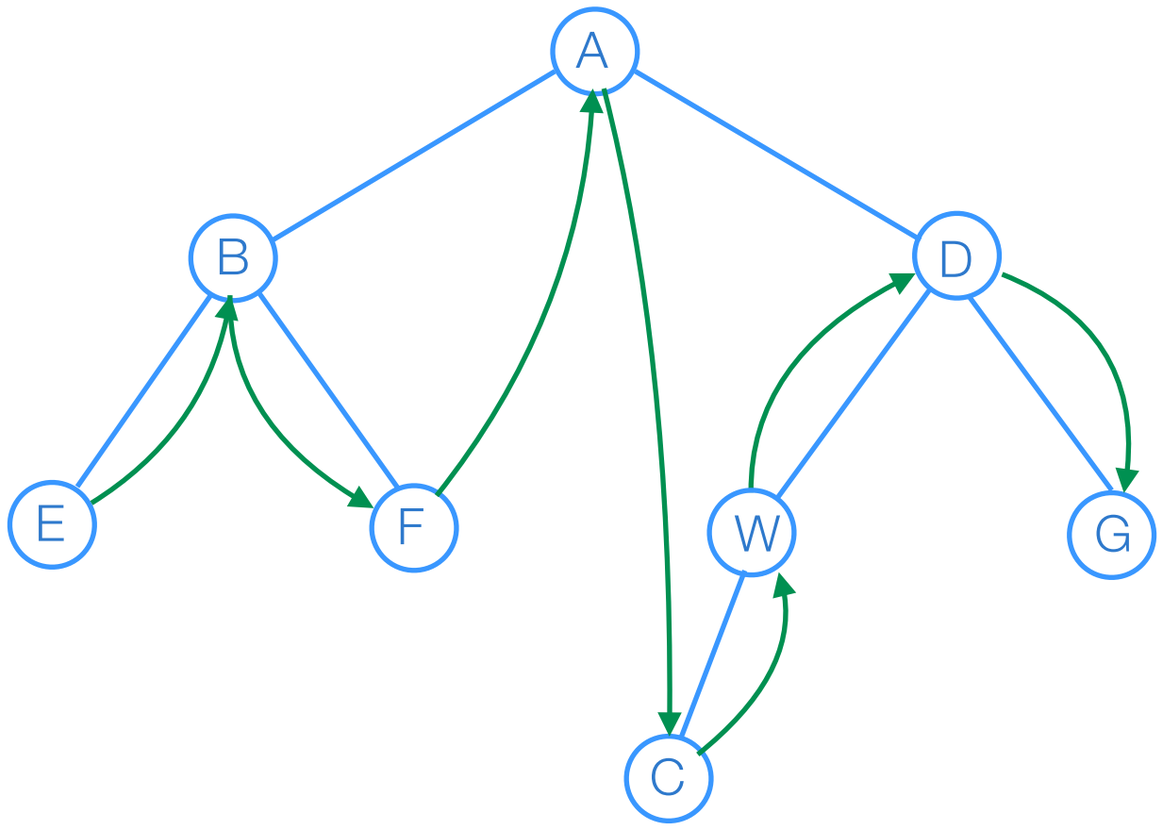
(b)

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

By calling Inorder-Tree-Walk(T.root()) we get 2, 5, 5, 6, 7, 8

Binary tree traversal: in order visit

- Can be seen as visit of the tree from left to right



Querying a binary search tree

- The core operation on a binary search tree is **search**
- But also:
  - **Minimum**: get the minimum element is a search binary tree
  - **Maximum**: get the maximum element is a search binary tree
  - **Successor**: Given a number, find the successor in the sorted order
  - **Predecessor**: Given a number, find the predecessor in the sorted order

Searching

We search for a node with a given key in a binary search tree.

TREE-SEARCH(*x*,*k*)

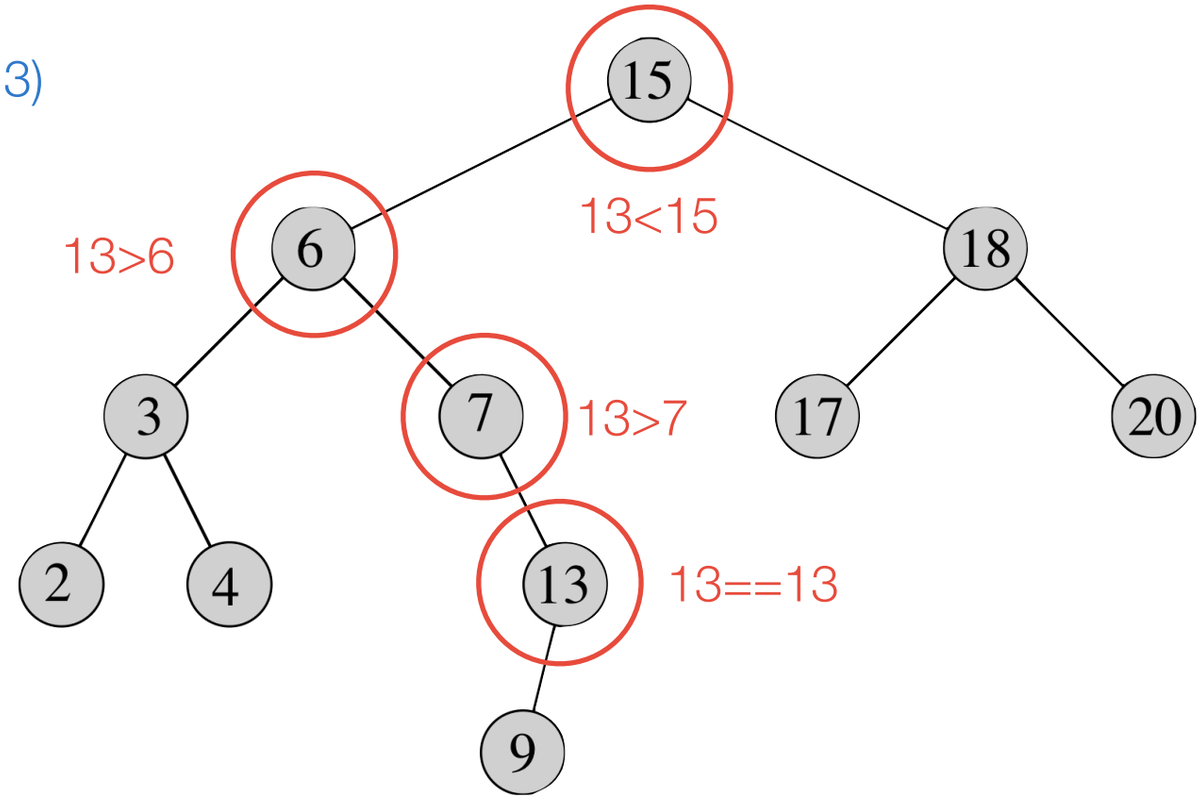
Inputs: *x* is a given node where the search starts (usually the root) and *k* is the key to be searched in the tree.
Outputs: a pointer to a node with key *k* or NIL if the key is not in the tree.

```
TREE-SEARCH(x,k)
if x==NIL or k==x.key then
    return x
if k<x.key
    return TREE-SEARCH(x.left,k)
else
    return TREE-SEARCH(x.right,k)
```

```
ITERATIVE-TREE-SEARCH(x,k)
while x!=NIL and k!=x.key do
    if k<x.key
        x = x.left
    else
        x = x.right
return x
```

Searching

Tree-search(15,13)




```
ITERATIVE-TREE-SEARCH(x,k)
while x!=NIL and k!=key[x] do
    if k<x.key
        x = x.left
    else
        x = x.right
return x
```

The nodes encountered during the process form a path downward from the root, thus we visit a number of nodes which equals the height of the node k in the tree

Complexity: O(*h*) where *h* is the height

What is the worst case?

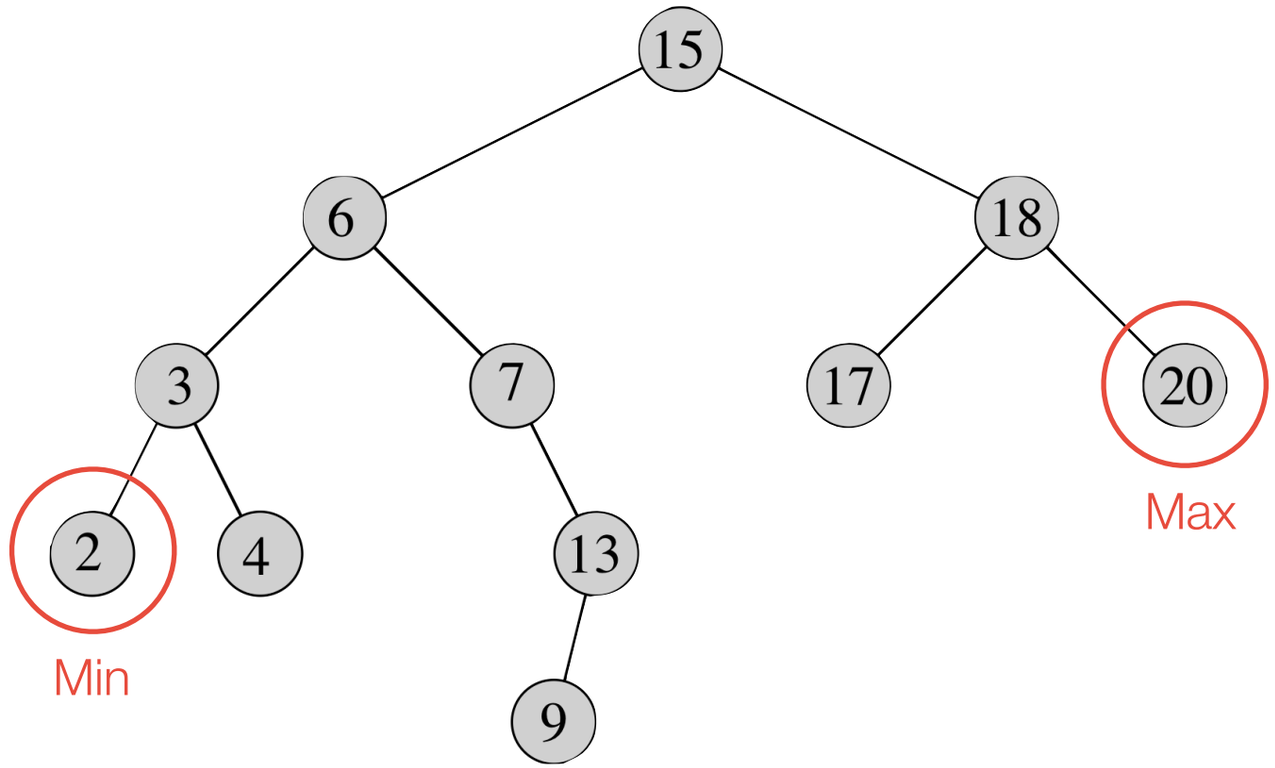
Searching exercise

- Suppose we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 364. Which of the following sequences cannot be the sequence of nodes examined?
  - 1,252,401,398,330,344,397,364
  - 924,220,911,244,898,258,362,364
  - 925,202,911,240,912,245,364 ← 912 cannot belong to the left subtree of 911

Minimum and Maximum

```
TREE-MINIMUM(x)
while x.left!=NIL do
    x = x.left
return x
```

You keep going down the left branch




```
TREE-MAXIMUM(x)
while x.right!=NIL do
    x = x.right
return x
```

You keep going down the right branch

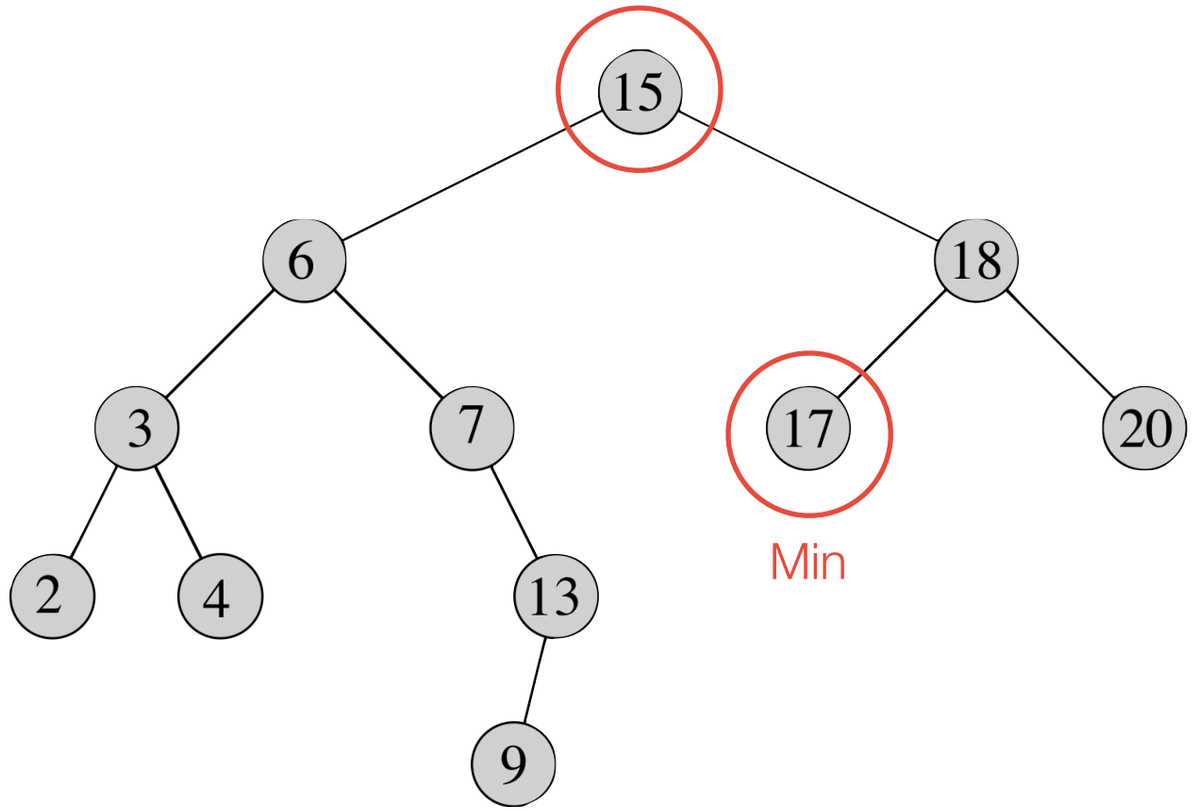
Successor: Property

- Given a binary search tree `T` and `x,y` in `T`, then `y` is a successor of `x` if `y.key>x.key` and there not exist a `z` in `T` such that `y.key>z.key>x.key`
- **Property**: Consider a binary search tree `T` whose keys are distinct. If the right subtree of a node `x` in `T` is empty and `x` has a successor `y`, then `y` is the lowest ancestor of `x` whose left child is also an ancestor-or-self of `x`.
- Note: in this case we consider a node to be an ancestor of itself.

Successor

```
TREE-SUCCESSOR(x)
if x.right!=NIL
    return TREE-MINIMUM(x.right)
else
    // go up the tree from x until we encounter a
    //node that is the left child of its parent
    y = T.parent(x)
    while y!=NIL and x==y.right
        x=y
        y = T.parent(y)
    return y
```

TREE-SUCCESSOR(15)



The running time is O(h) since we either follow a simple path downwards or a simple path upwards