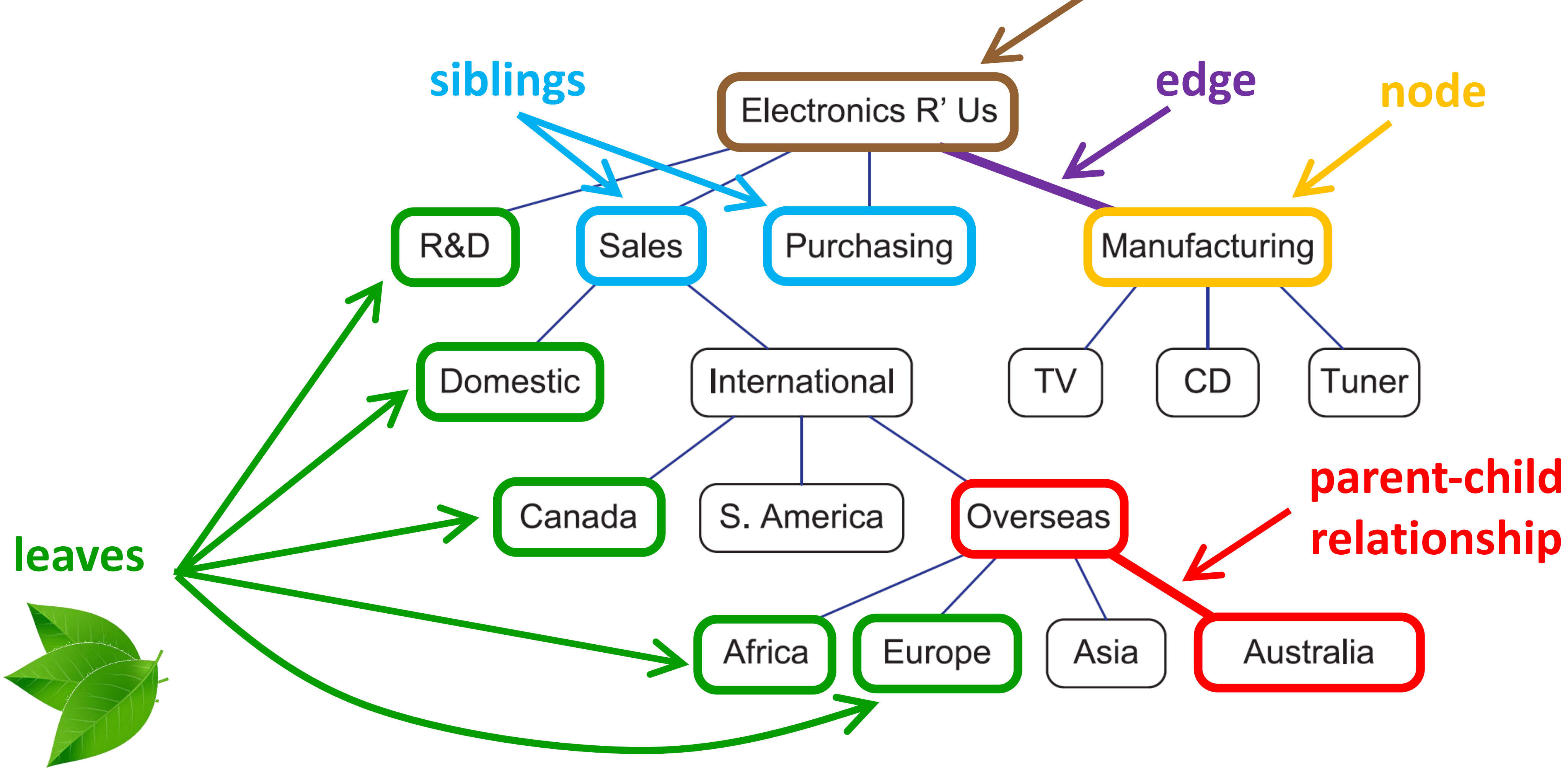
CHAPTER 7:

TREES

TREE EXAMPLE

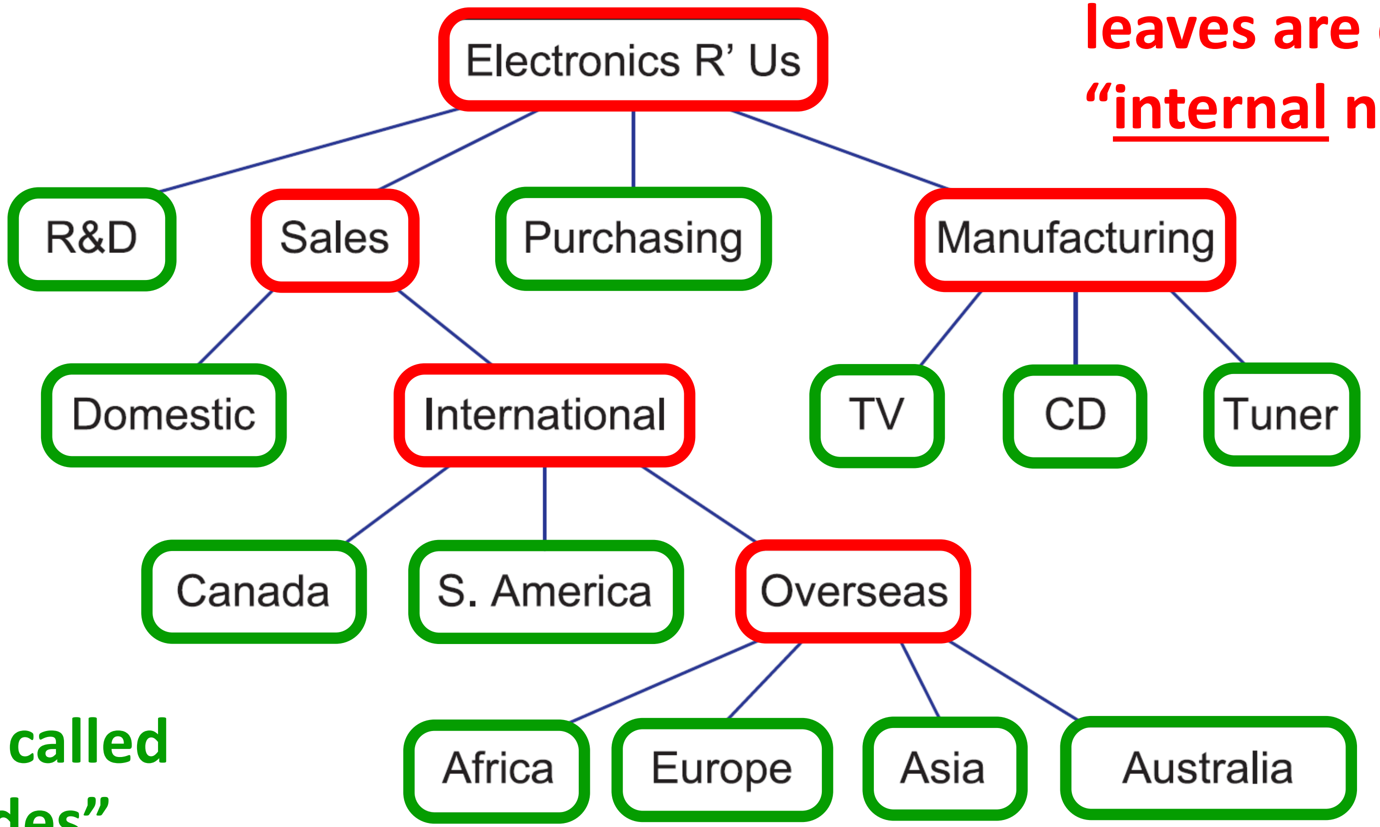
- A tree can represent a hierarchical structure



TREE EXAMPLE

- A tree can represent a hierarchical structure

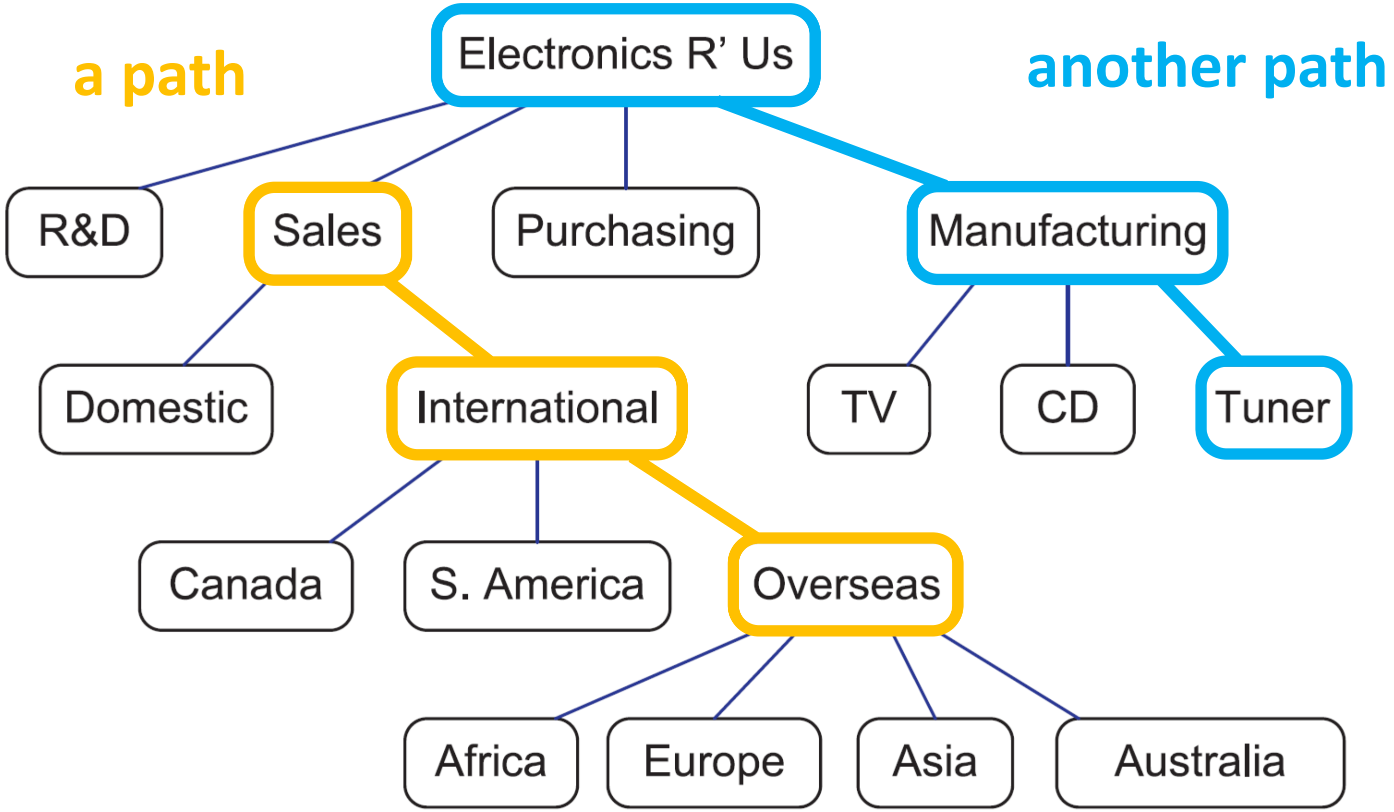
Nodes that are not leaves are called "<u>internal</u> nodes"



leaves are also called "<u>external</u> nodes"

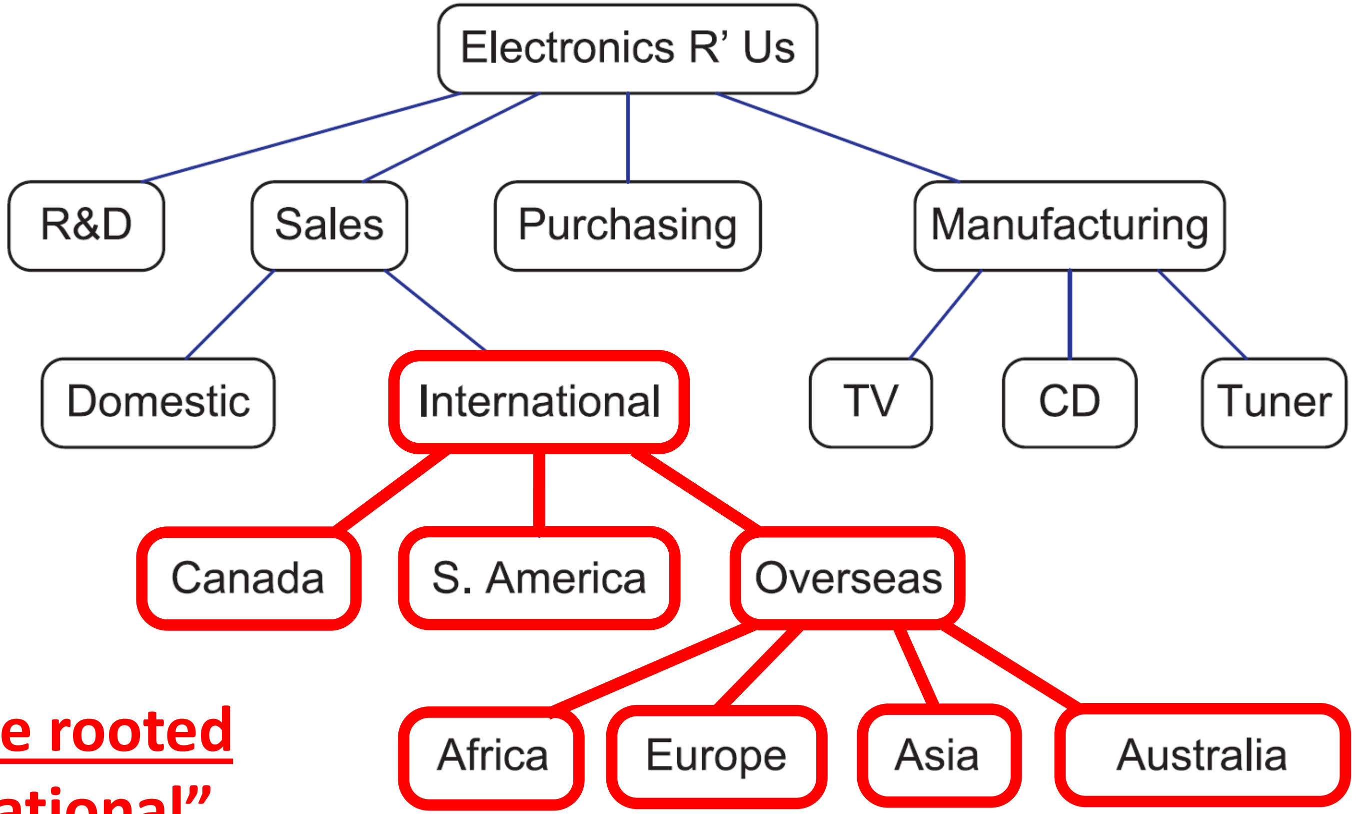
TREE EXAMPLE

- A tree can represent a hierarchical structure



TREE EXAMPLE

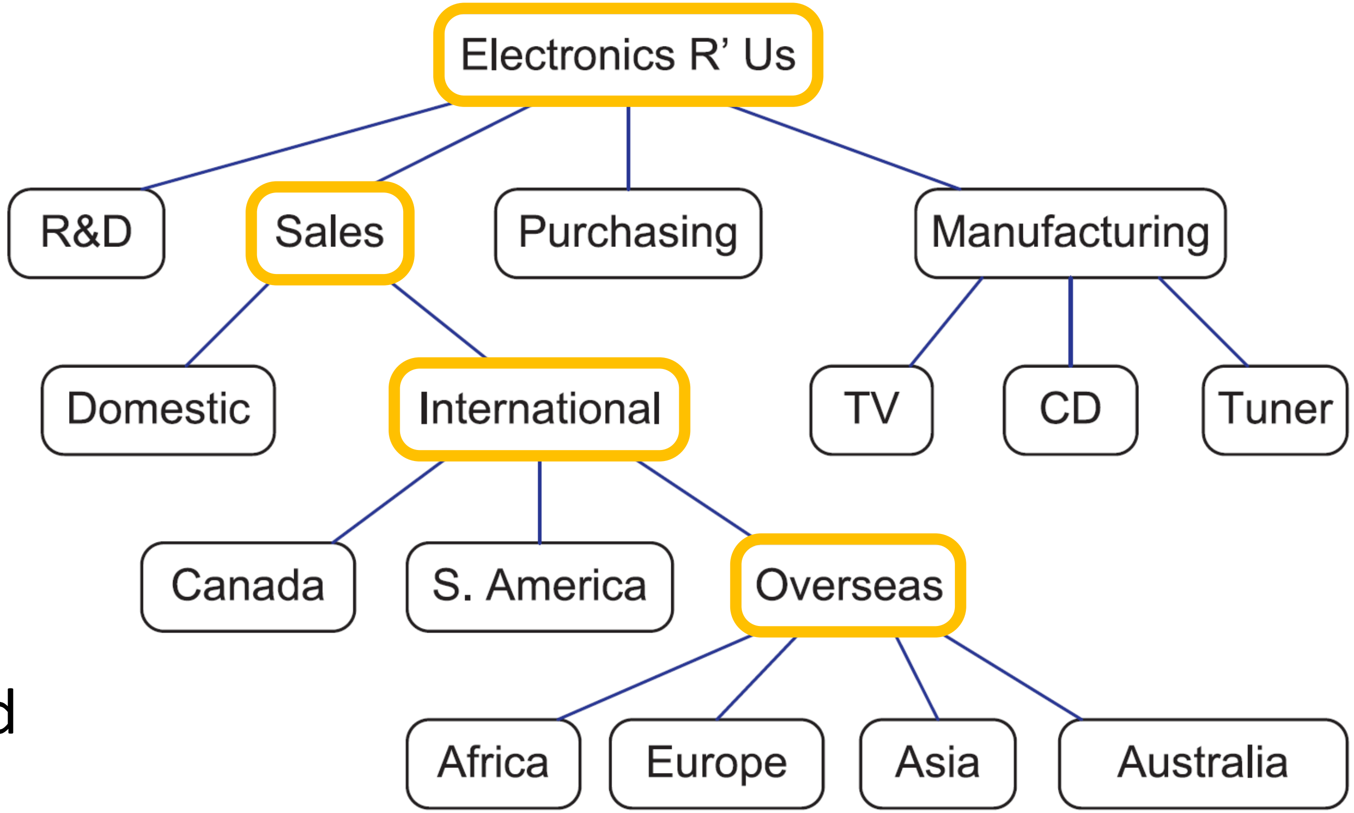
- A tree can represent a hierarchical structure



The <u>subtree rooted</u> at "International"

TREE EXAMPLE

- A tree can represent a hierarchical structure

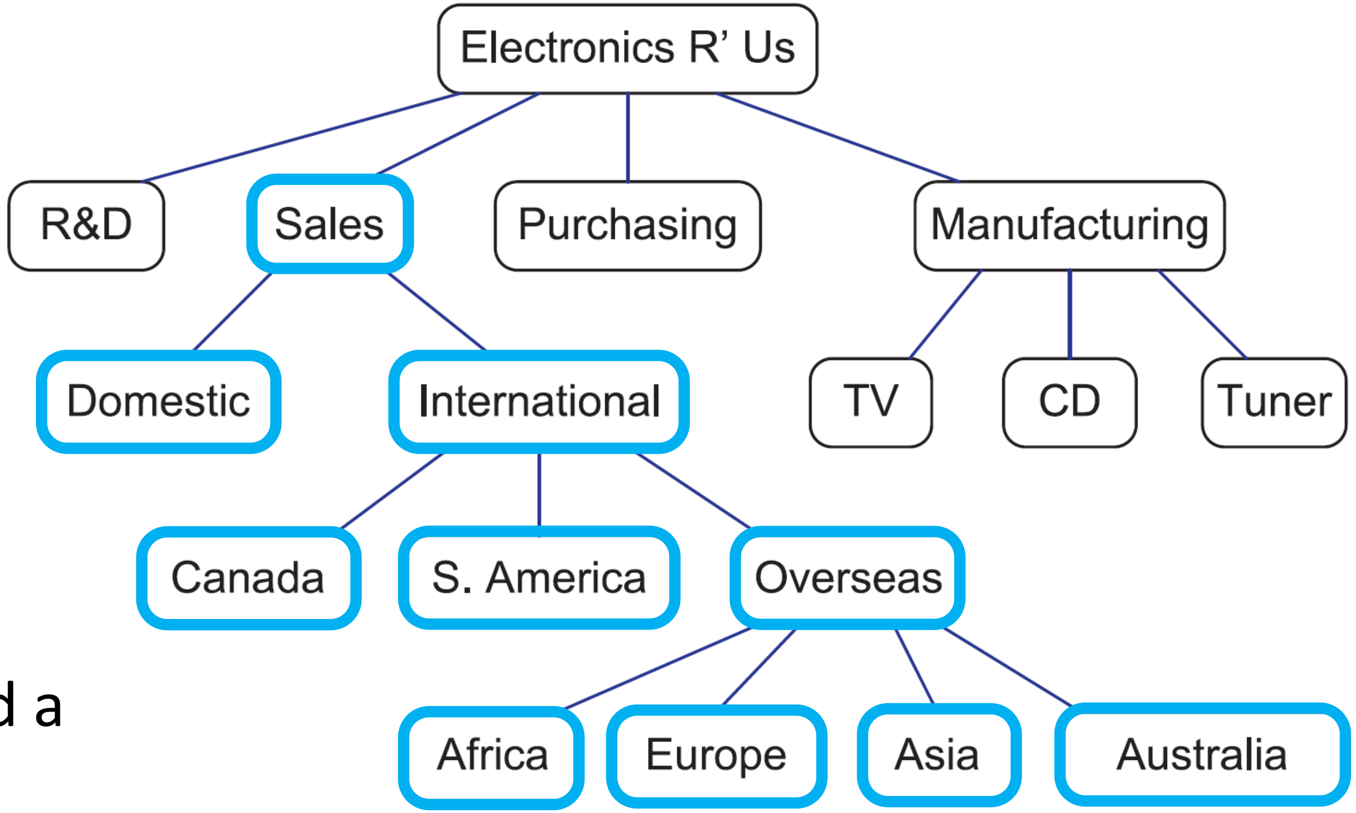


The ancestors of "Overseas"

A node is considered an ancestor of itself

TREE EXAMPLE

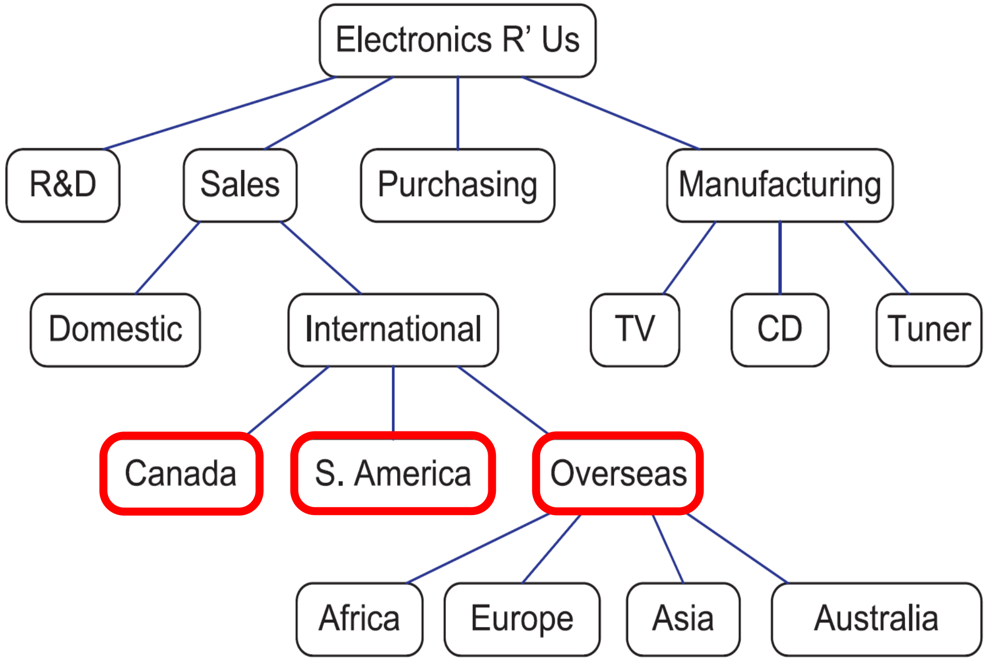
- A tree can represent a hierarchical structure



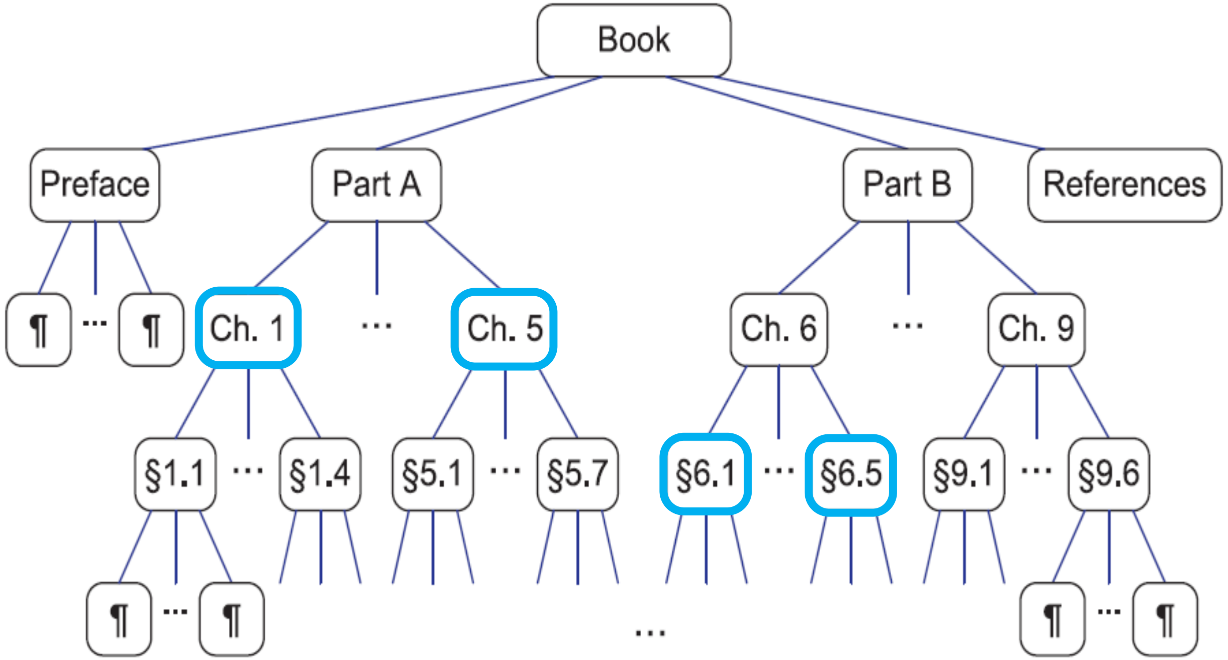
The descendants of "Sales"

A node is considered a descendant of itself

TREE EXAMPLE



This is **not an ordered tree**, because there is no order between siblings



This tree is **ordered**, because the book parts are ordered, and in each part the chapters are ordered, etc.

NODE-RELATED METHODS

- Each node can be associated with a **position**. Given a position *p*, its node is accessed by **p* to get the element. We will often use the terms “position” and “node” interchangeably

- The position, *p*, **is an object** that not only points to a node, but also describes its position in the tree via its methods:

  - parent(): returns the parent of *p* (error, if *p* is the root)
  - children(): Returns a list of positions of all *p*‘s children. If *p* is external, the returned list will be empty
  - isRoot(): Returns true, if *p* is the root
  - isExternal(): Returns true, if *p* is external

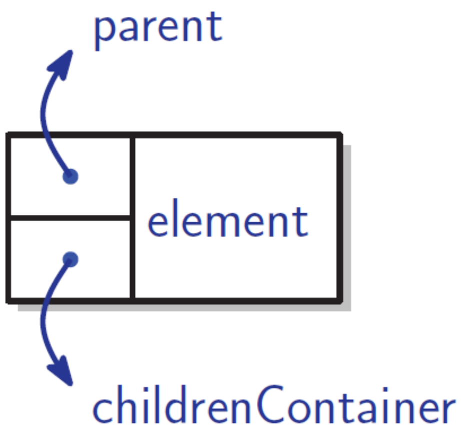
TREE-RELATED METHODS

- The tree itself mainly supports, but not limited to, the following functionalities:

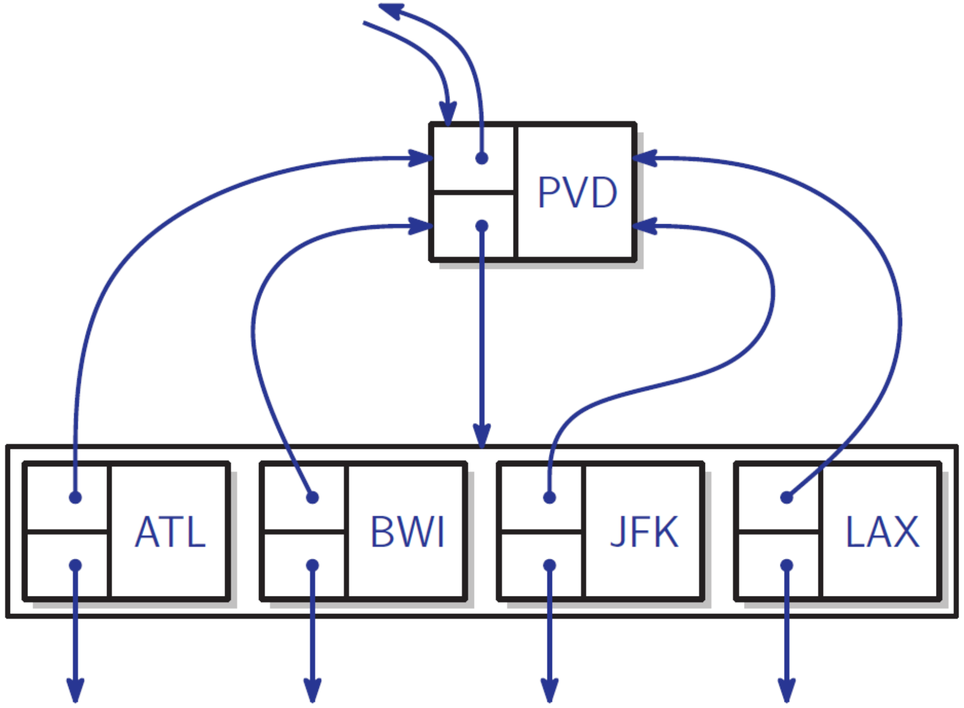
  - size(): Returns the number of nodes in the tree
  - empty(): Returns true, if the tree is empty
  - root(): Returns a position for the root (error, if tree is empty)
  - positions(): Returns a list of positions of all the nodes of the tree

TREE – LINKED STRUCTURE

- Here is an illustration of a node, which has an **element**, a **parent** (which is NULL if the node is the root) and a container of all its **children**



- Here is an illustration of the tree data structure



TREE – COMPLEXITY

- The running time, where c_p is the number of children of p:

Operation	*Time*
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

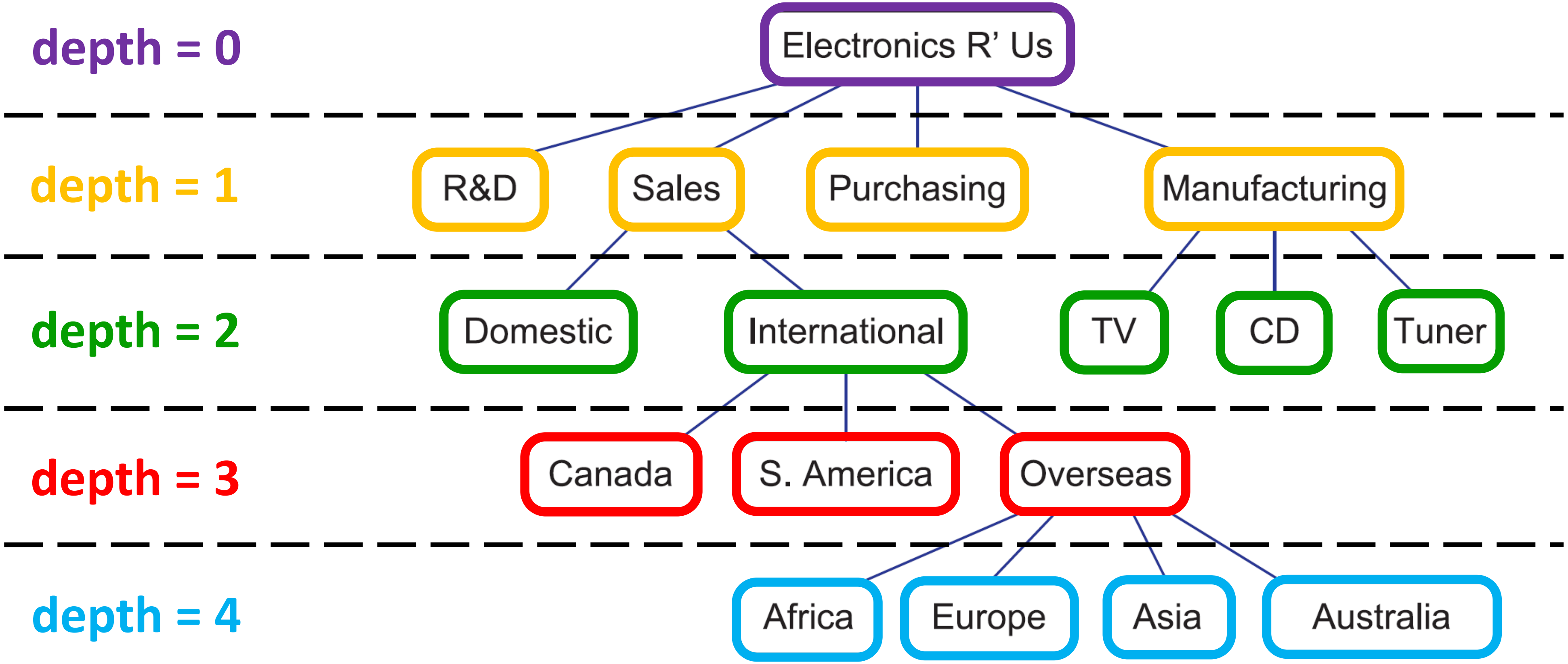
Node-related functions

Tree-related functions

- children(p) **iterates through the container** containing the children, which takes $O(c_p)$
- With a similar reasoning, "positions" runs in O(n) time

NODE DEPTH

- The "**depth**" of a node represents its **distance from the root**



NODE DEPTH

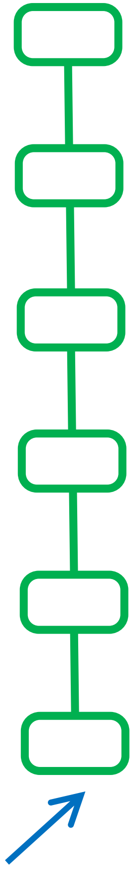
- The "**depth**" of a node represents its **distance from the root**

  How would you compute the depth of a node, p, **recursively**?

  Algorithm depth(T, p):
  Input: Tree T and a position p
  Output: The depth of the node referred to by p
  if p.isRoot() **then**
      return 0
  else return 1 + depth(T, p.parent())

- The running time is O(d_p), where d_p is the depth of node p in tree T.
- In the worst case, the depth algorithm runs in O(n) time. What does the tree look like in the worst case? Like a chain!
- Still, it's more accurate to characterize the running time in terms of d_p rather than n, because d_p is often much smaller than n

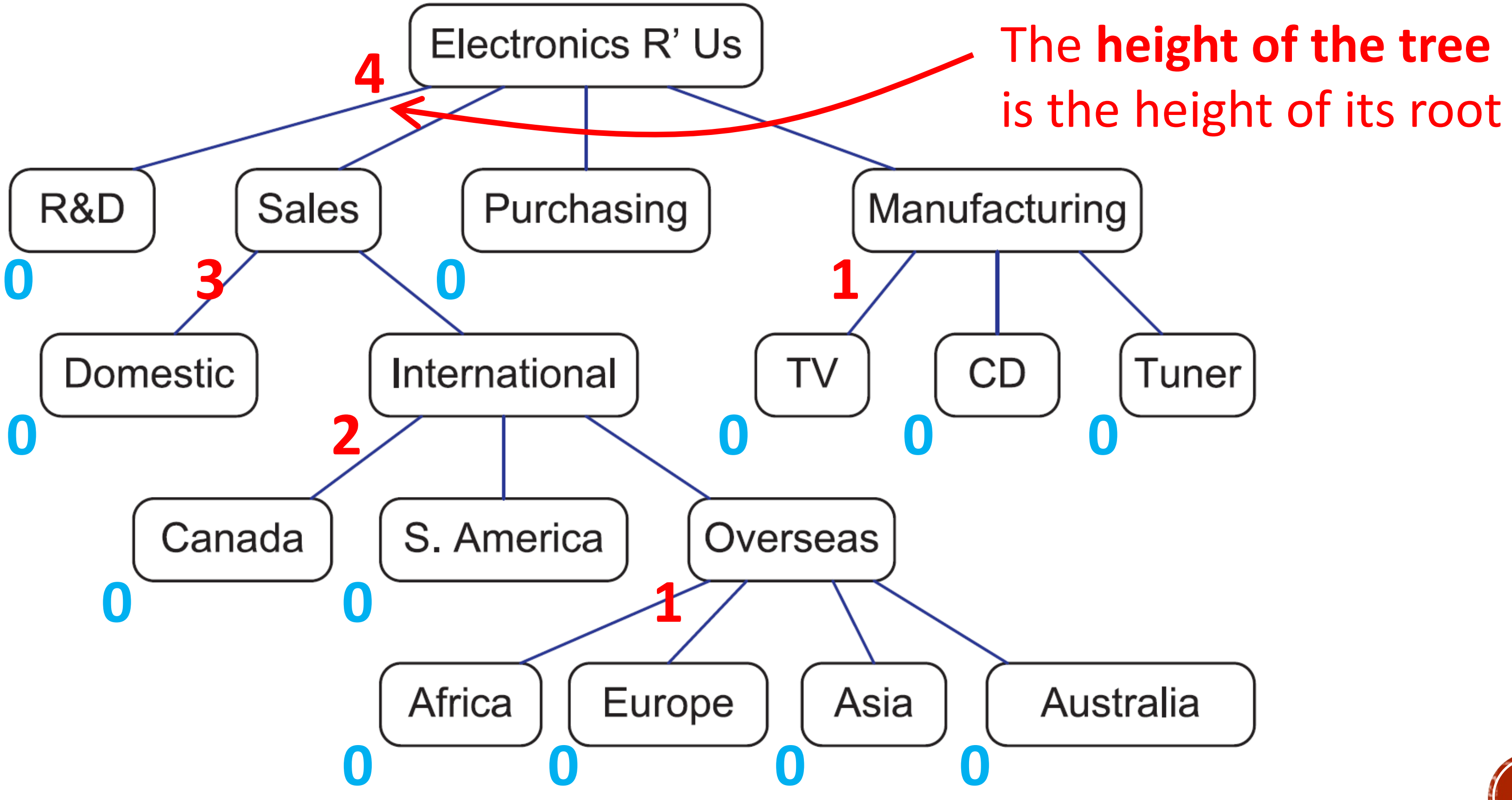
root



For this node,
$d_p = n$

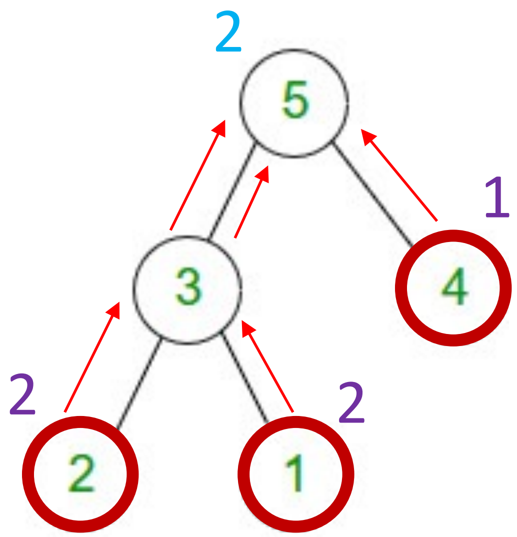
NODE HEIGHT

- The "**height**" of a node represents its **distance from the farthest descendant**
- **Proposition:** The height of a tree is equal to **the maximum depth of its leaves**!



NODE HEIGHT

- The "**height**" of a node represents its **distance from the farthest descendant**
- **Proposition:** The height of a tree is equal to **the maximum depth of its leaves**!
- Based on this proposition, how can we the function "`depth(T,q)`" to compute the height of a tree?



Algorithm treeHeight(T):
  Input: Tree T
  Output: The height of the tree T
    $h \leftarrow 0$
    $nodes \leftarrow T$.positions()
    for $q \leftarrow nodes$.begin() to $nodes$.end() **do**
      if q.isExternal() **then**
        $h \leftarrow \max(h, \text{depth}(T, q))$
  return h

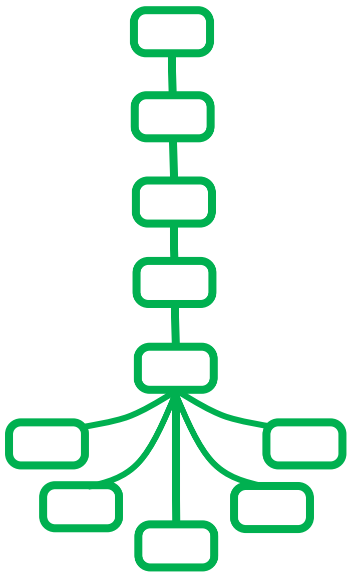
NODE HEIGHT

What is the running time, knowing that depth(T,*p) runs in O(d_p) time?

Algorithm treeHeight(T):
  Input: Tree T
  Output: The height of the tree T
    $h \leftarrow 0$
    $nodes \leftarrow T$.positions()
    for $q \leftarrow nodes$.begin() to $nodes$.end() **do**
      if q.isExternal() **then**
        $h \leftarrow \max(h, \text{depth}(T, q))$
    return h

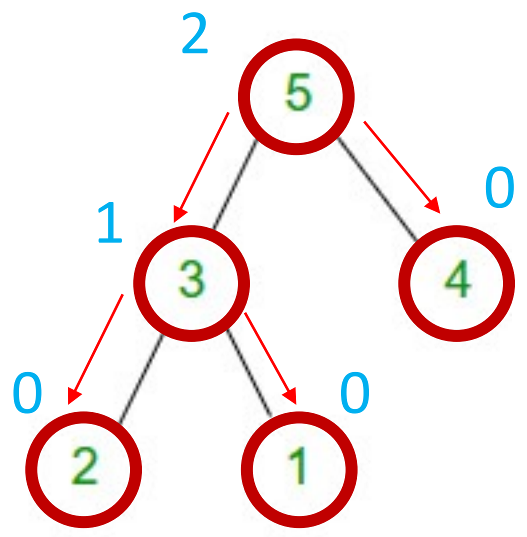
- For each **internal node**, we need to do some constant time operations, i.e., O(1)
- For each **external node**, q, we need to call depth(T, q), which runs in O(d_p)
- The total is then ≈ O($n + \sum_{p \,\in\, \text{ExternalNodes}} d_p$)
- The **worst case** is when:
  - $n/2$ nodes form a chain from the root
  - The remaining $n/2$ nodes are the leaves

This way, runtime is $O\left(n + \sum_{i=1}^{n/2} n/2\right)$ which is O(n^2)



NODE HEIGHT

- Here is another recursive algorithm, which computes the height of any given node, p, in a tree, T (to compute the height of T, set p to be the root of T).



Algorithm height(T, p):
  Input: Tree T and position p
  Output: The height of the node referred to by p
    if q.isExternal() **then**
      return 0
    $h \leftarrow 0$
    $childrenList \leftarrow p$.children()
    for $q \leftarrow childrenList$.begin() to $childrenList$.end() **do**
      $h \leftarrow \max(h, \text{height}(T, q))$
    return $h + 1$

NODE HEIGHT

What is the running time of this algorithm?

```
if q.isExternal() then
      return 0
   h ← 0
   childrenList ← p.children()
   for q ← childrenList.begin() to childrenList.end() do
      h ← max(h, height(T, q))
   return h + 1
```

- The algorithm is **recursive**, and, if it is initially called on the root of T, it will eventually be called on each node of T.
- Thus, we can determine the running time by summing, over all the nodes, the amount of time spent at each node (on the non-recursive part).
- For each node, p, this amount is ≈ $O(c_p)$ because of iterating over p's children
- Thus, the algorithm takes ≈ $O\left(\sum_p c_p\right)$ time.
- Finally, note that $\sum_p c_p = n - 1$. This because each node of T (except the root) is a child of another node, and thus contributes one unit to the sum.
- Thus, this algorithm runs in $O(n)$

NODE HEIGHT

```
Algorithm treeHeight(T):
  Input: Tree T
  Output: The height of the tree T
    h ← 0
    nodes ← T.positions()
    for q ← nodes.begin() to nodes.end() do
        if q.isExternal() then
            h ← max(h, depth(T, q))
    return h
```

$O(n^2)$

```
Algorithm height(T, p):
  Input: Tree T and position p
  Output: The height of the node referred to by p
    if q.isExternal() then
        return 0
    h ← 0
    childrenList ← p.children()
    for q ← childrenList.begin() to childrenList.end() do
        h ← max(h, height(T, q))
    return h + 1
```

$O(n)$

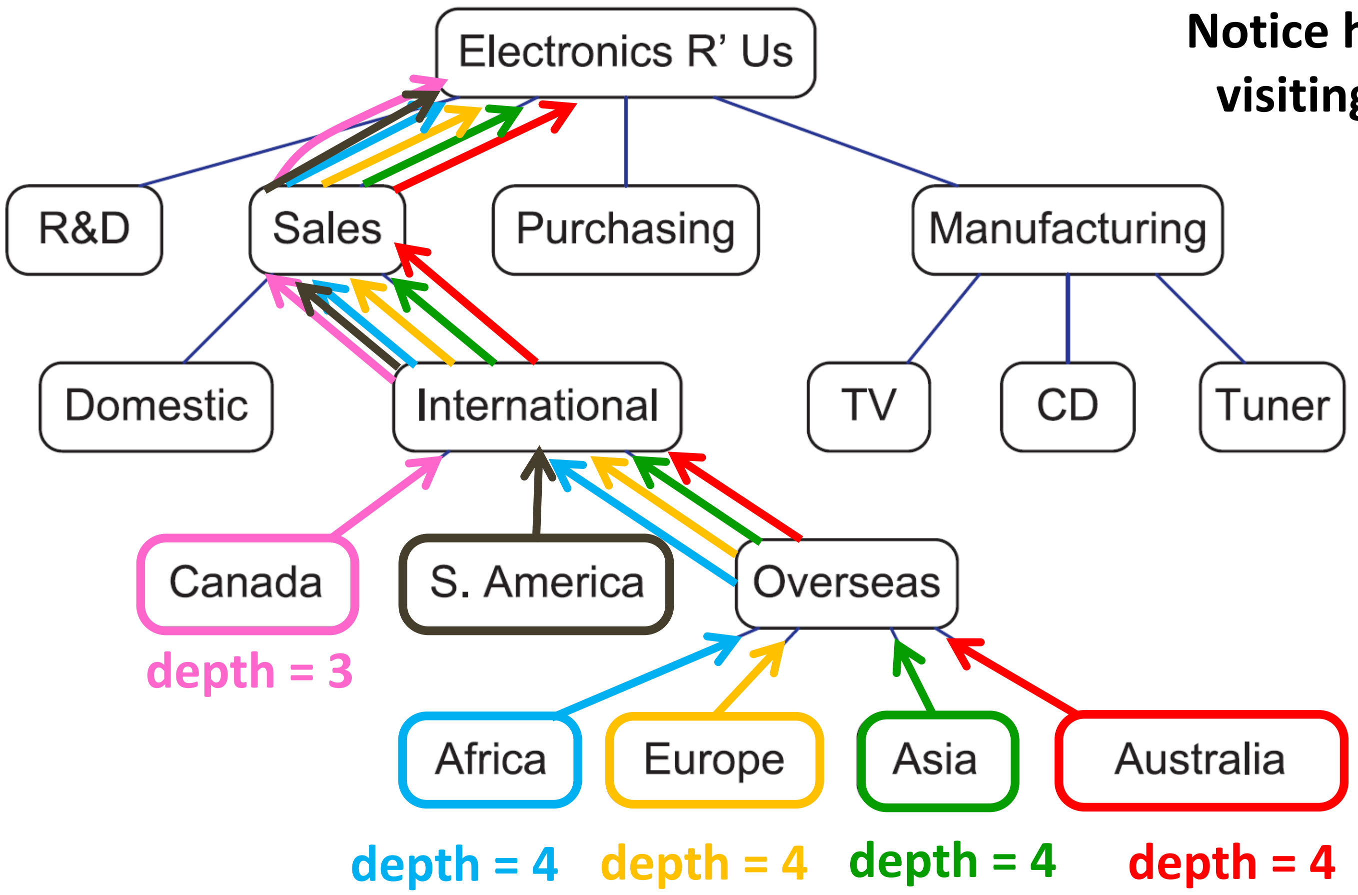
What happened?

What went wrong with the left algorithm?

Can you spot the repetition here?

WHAT WENT WRONG

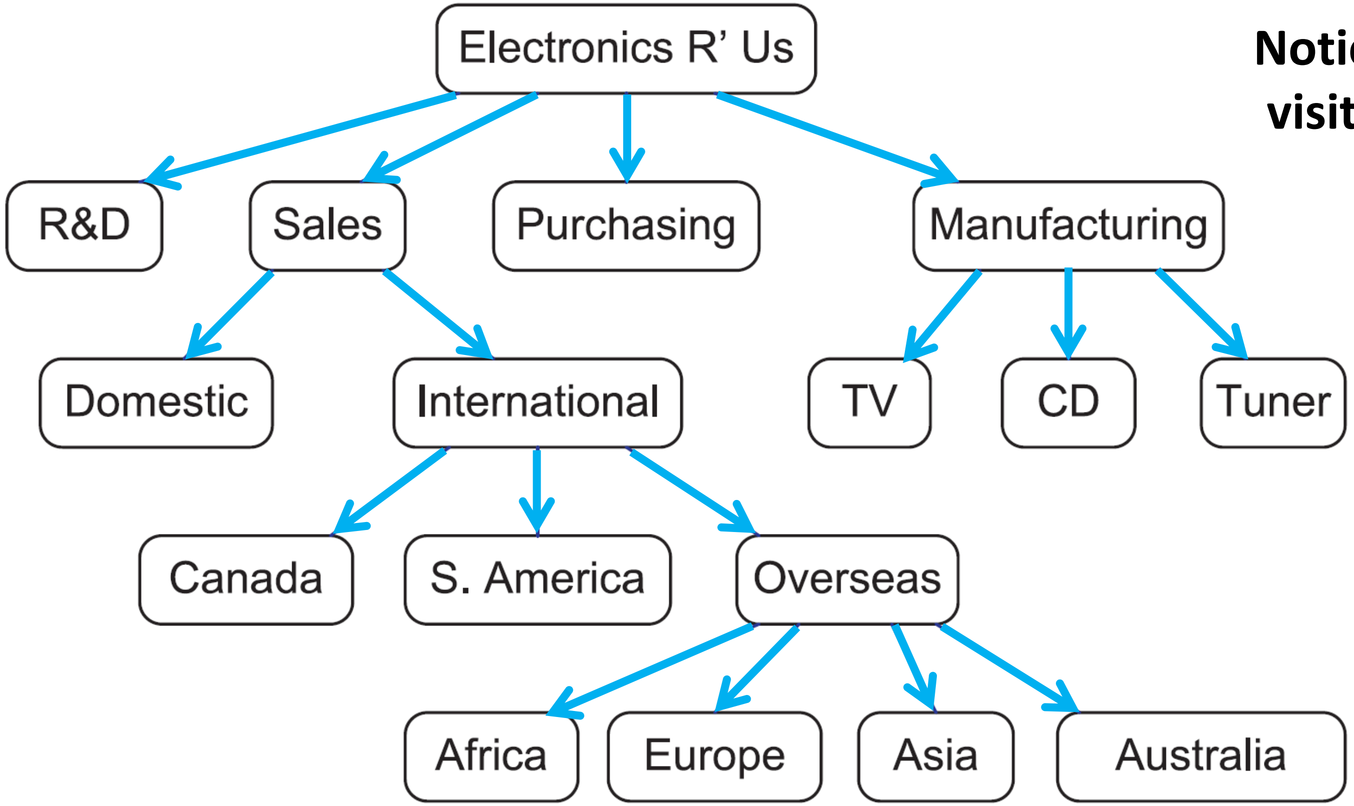
- The inefficient algorithm calculates the **depth of leaves**. Thus it starts from the **bottom** and traverses the tree **upward**!



Notice how the algorithm keeps visiting the same edges again and again!

WHAT WENT WRONG

- In contrast, the efficient algorithm calculates the height of the nodes, by starting from the **top** and traversing the tree **downward**!



Notice how the algorithm never visits an edge more than once!