Algorithms & Data Structures

Lesson 5: Dictionary ADTs; Binary Trees

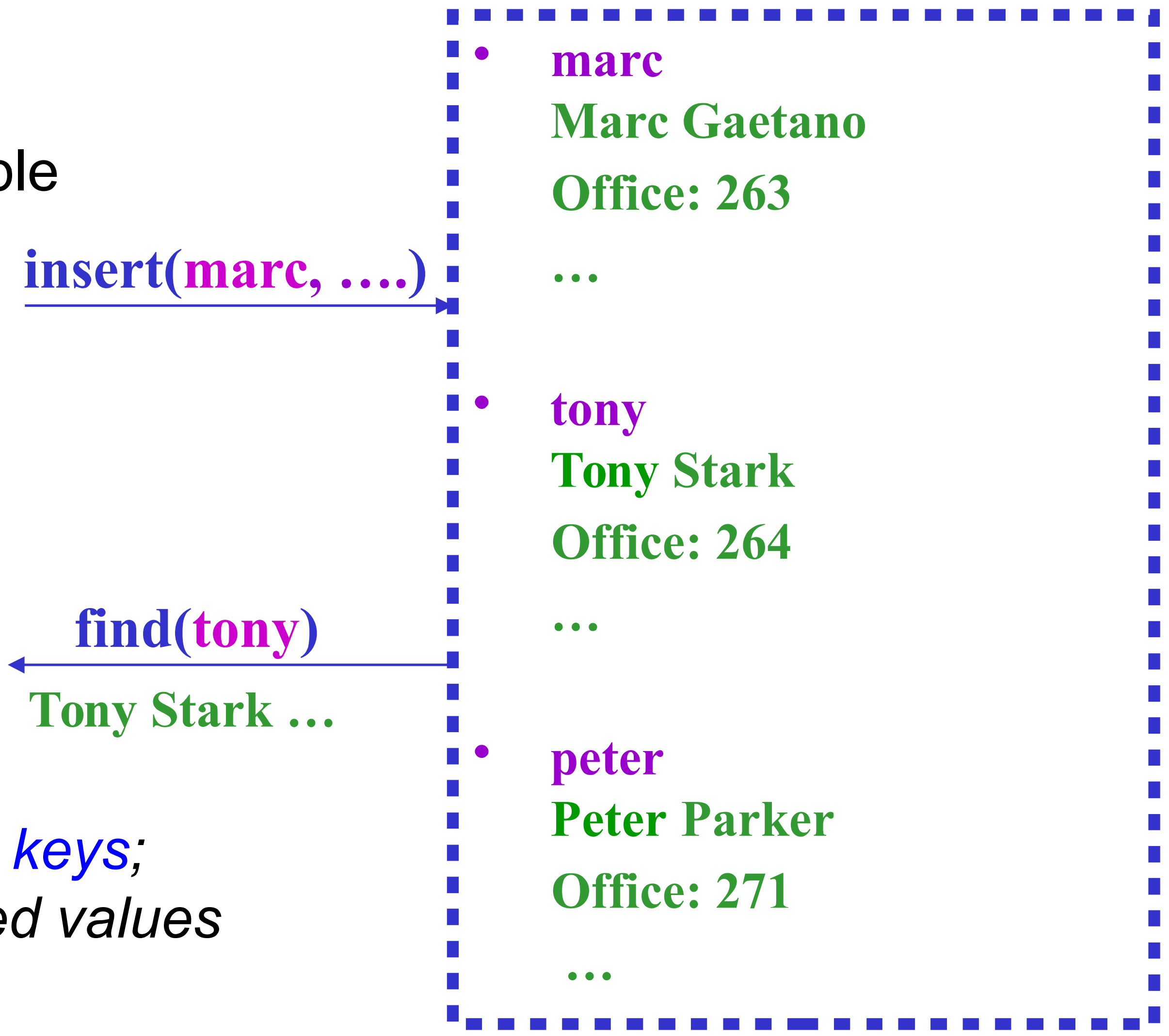
Marc Gaetano

Edition 2017-2018

The Dictionary (a.k.a. Map) ADT

- **Data:**
  - set of (key, value) pairs
  - keys must be comparable
- **Operations:**
  - **`insert(key,value)`**
  - **`find(key)`**
  - **`delete(key)`**
  - …

Will tend to emphasize the keys;
don't forget about the stored values



A “Modest” Few Uses

Any time you want to store information according to some key and be able to retrieve it efficiently

– Lots of programs do that!

• Search:	inverted indexes, phone directories, …
• Networks:	router tables
• OS:	page tables
• Compilers:	symbol tables
• Databases:	dictionaries with other nice properties
• Biology:	genome maps
• …	

Possibly the most widely used ADT!!

Simple implementations

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
• Unsorted linked-list	$O(1)$*	$O(n)$	$O(n)$
• Unsorted array	$O(1)$*	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$

*** Unless we need to check for duplicates**

We'll see a Binary Search Tree (BST) probably does better
 but not in the worst case (unless we keep it balanced)

Better dictionary data structures

There are many good data structures for (large) dictionaries

1. **Binary search trees**
2. **AVL trees**
   - Binary search trees with *guaranteed balancing*

3. **B-Trees**
   - Also always balanced, but different and shallower
   - B-Trees are not the same as Binary Trees

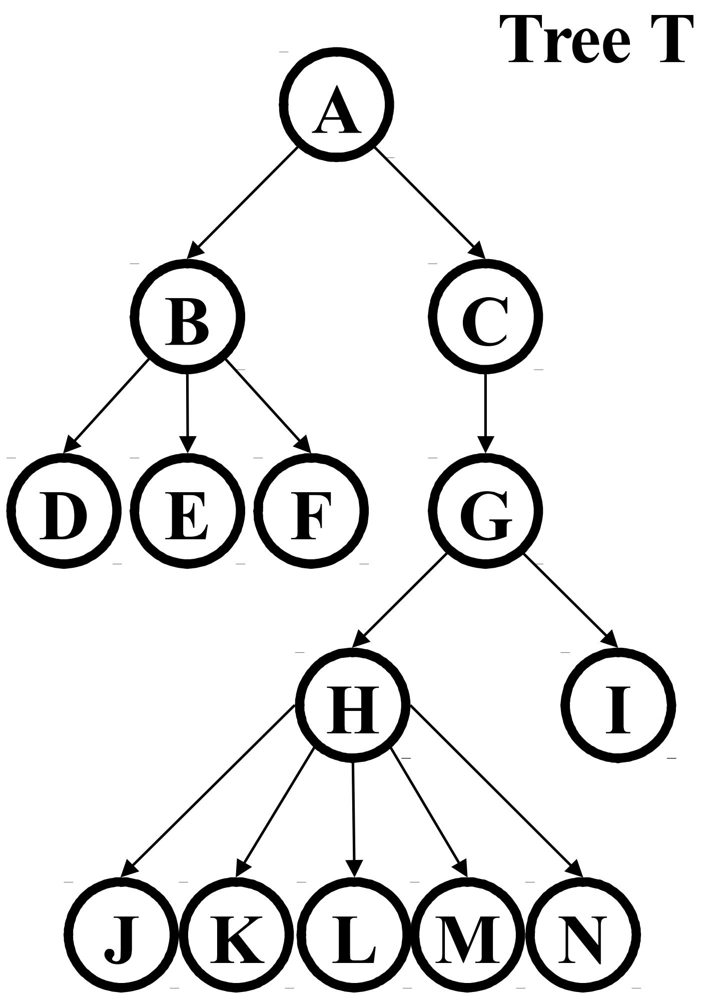
     B-Trees generally have large branching factor

4. **Hashtables**
   - Not tree-like at all

Tree terminology

Root (tree)
Leaves (tree)
Children (node)
Parent (node)
Siblings (node)
Ancestors (node)
Descendents (node)
Subtree (node)

Depth (node)
Height (tree)
Degree (node)
Branching factor (tree)



More tree terminology

- **There are many kinds of trees**
  - Every binary tree is a tree
  - Every list is kind of a tree (think of “next” as the one child)

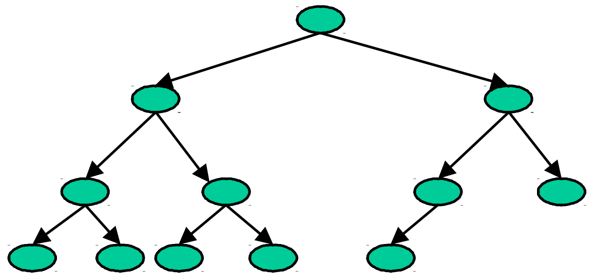
- **There are many kinds of binary trees**
  - Every binary search tree is a binary tree
  - Later: A binary heap is a different kind of binary tree

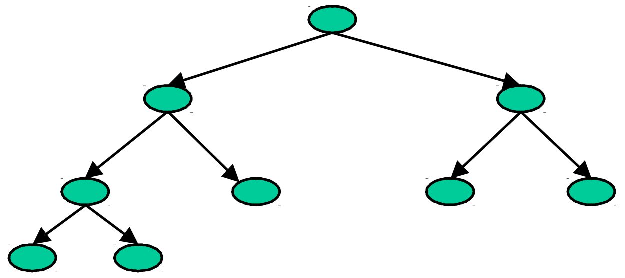
- **A tree can be balanced or not**
  - A balanced tree with *n* nodes has a height of *O*(**`log`** *n*)
  - Different tree data structures have different “balance conditions” to achieve this

Kinds of trees

Certain terms define trees with specific structure

- Binary tree: Each node has at most 2 children (branching factor 2)
- *n*-ary tree: Each node has at most *n* children (branching factor *n*)
- Perfect tree: Each row completely full
- Complete tree: Each row completely full except maybe the bottom row, which is filled from left to right





What is the height of a perfect binary tree with n nodes?

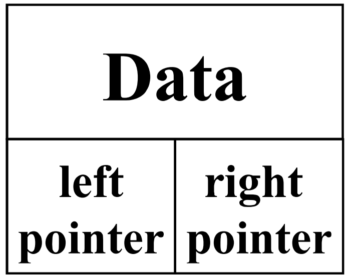
A complete binary tree?

Binary Trees

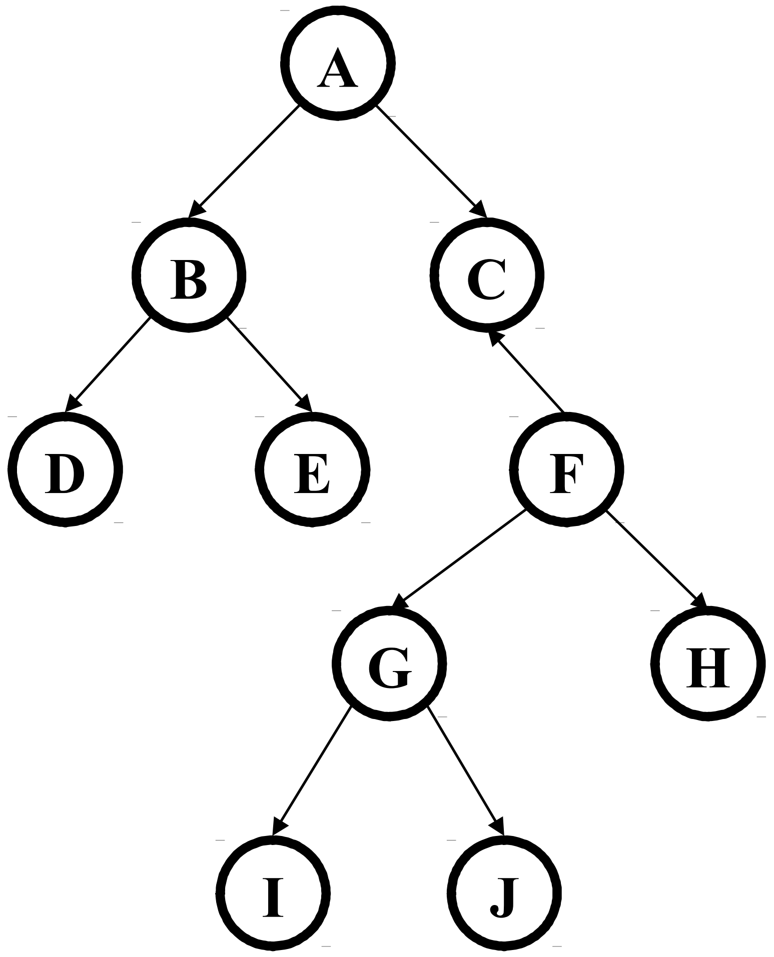
- Binary tree: Each node has at most 2 children (branching factor 2)

- **Binary tree is**
  - A root *(with data)*
  - A left subtree *(may be empty)*
  - A right subtree *(may be empty)*

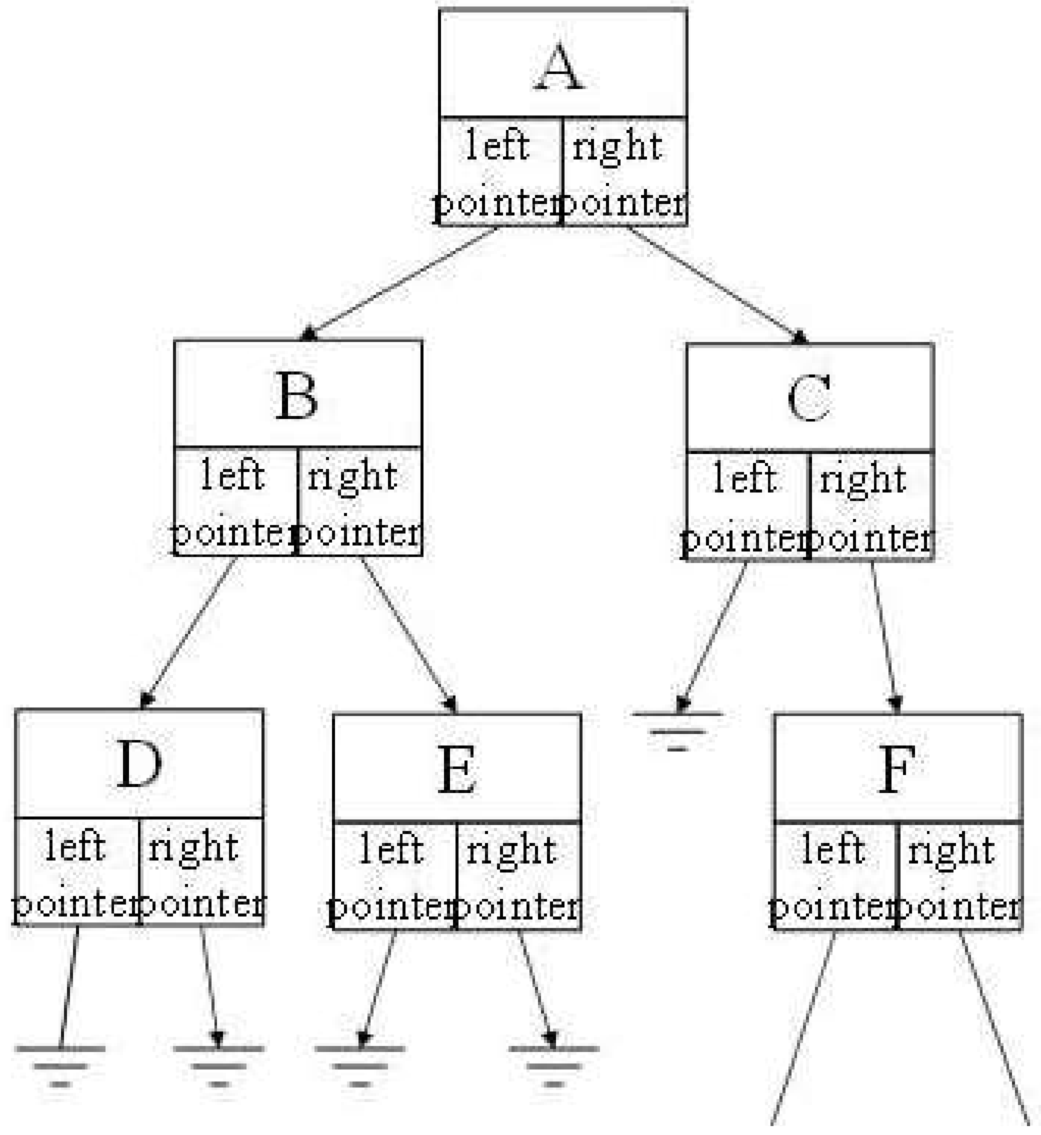
- **Representation:**

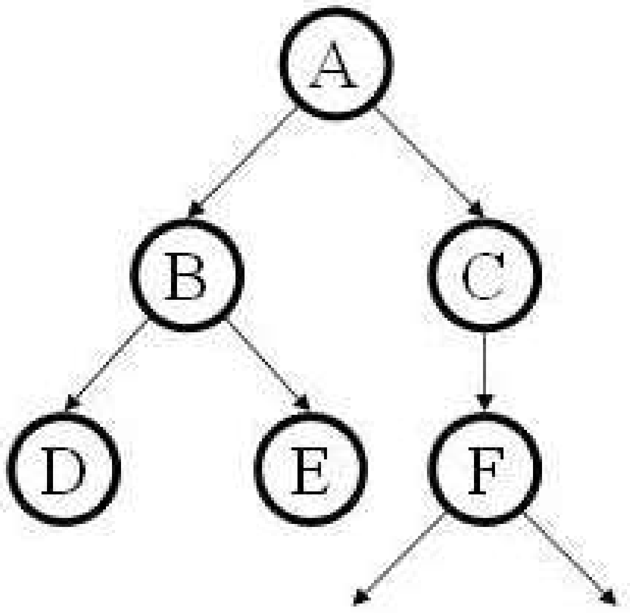


- For a dictionary, data will include a key and a value



Binary Tree Representation



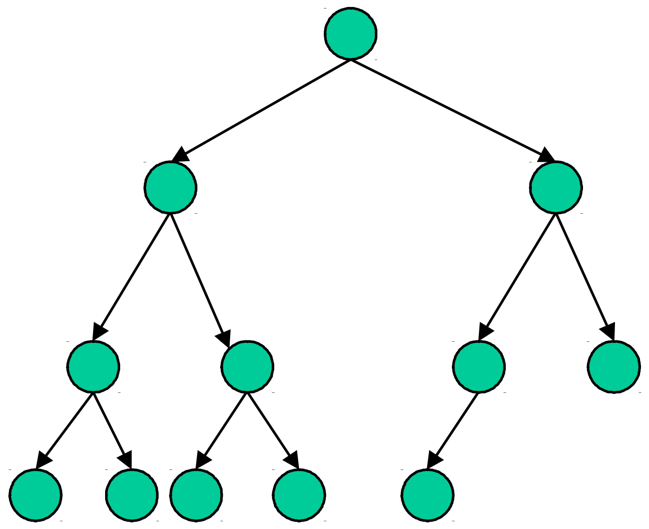


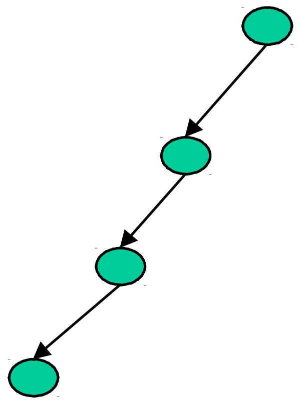
Binary Trees: Some Numbers

height of a tree = longest path from root to leaf (count edges)

For binary tree of height *h*:

- max # of leaves: 2^h
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h + 1$





For n nodes, we cannot do better than O(`log` n) height and we want to avoid O(n) height

Calculating height

What is the height of a tree with root **`root`**?

```
int treeHeight(Node root) {

        ???


}
```

Calculating height

What is the height of a tree with root **`root`**?

```
int treeHeight(Node root) {
  if(root == null)
    return -1;
  return 1 + max(treeHeight(root.left),
                 treeHeight(root.right));
}
```

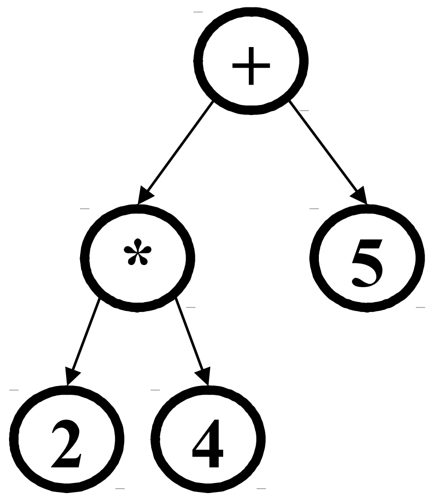
Running time for tree with *n* nodes: *O*(*n*) – single pass over tree

Note: non-recursive is painful – need your own stack of pending nodes; much easier to use recursion's call stack

Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

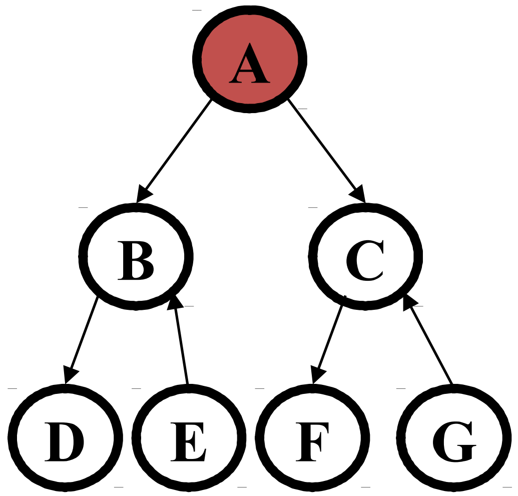
- ***Pre-order***: root, left subtree, right subtree
- ***In-order***: left subtree, root, right subtree
- ***Post-order***: left subtree, right subtree, root



(an expression tree)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```

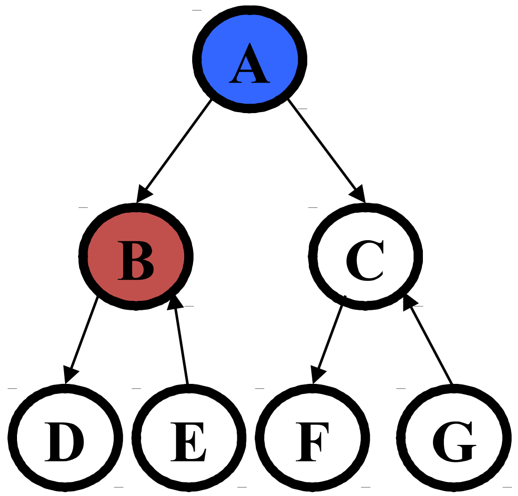


🔴 = current node 🔵 = processing (on the call stack)

⚫ = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```

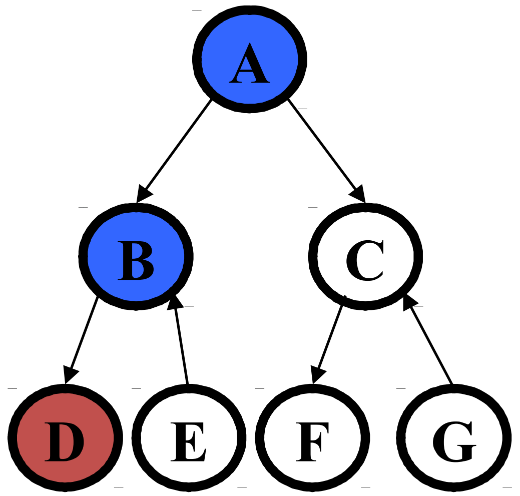


🔴 A = current node 🔵 A = processing (on the call stack)

⚫ A = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```

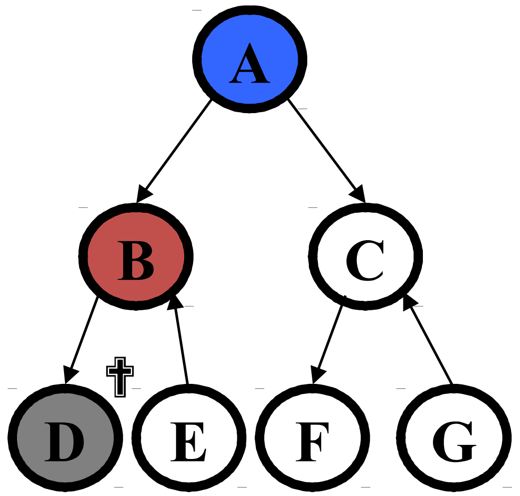


🔴 A = current node 🔵 A = processing (on the call stack)

⚫ A = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
     inOrderTraversal(t.left);
     process(t.element);
     inOrderTraversal(t.right);
   }
}
```

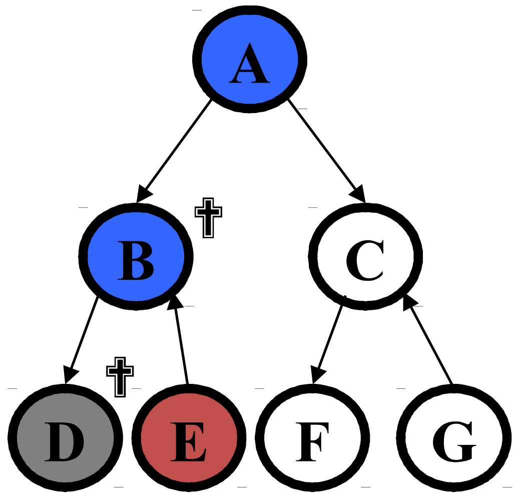


🟥 A = current node 🟦 A = processing (on the call stack)

⬛ A = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
  if(t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}
```

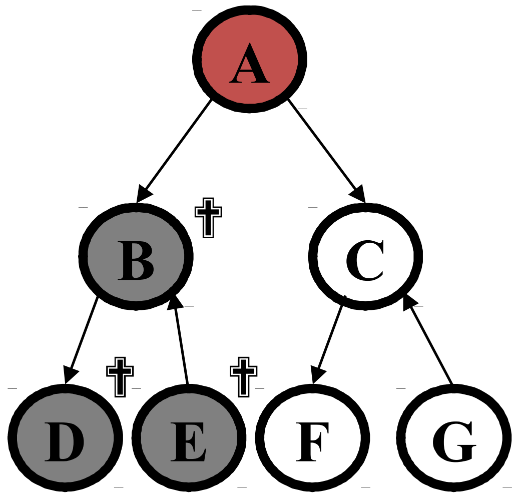


🟥 = current node   🟦 = processing (on the call stack)

⬛ = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```

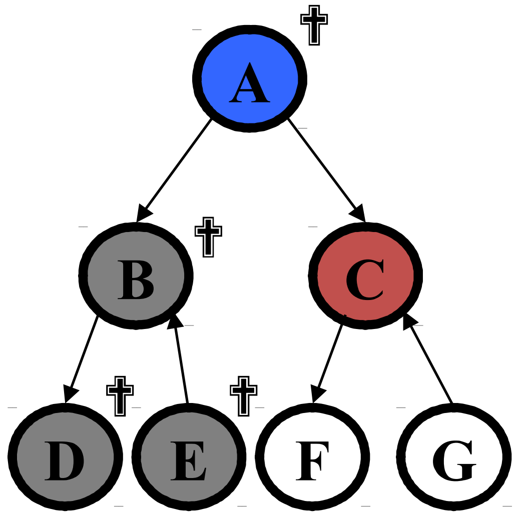


🔴 = current node 🔵 = processing (on the call stack)

⚫ = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```

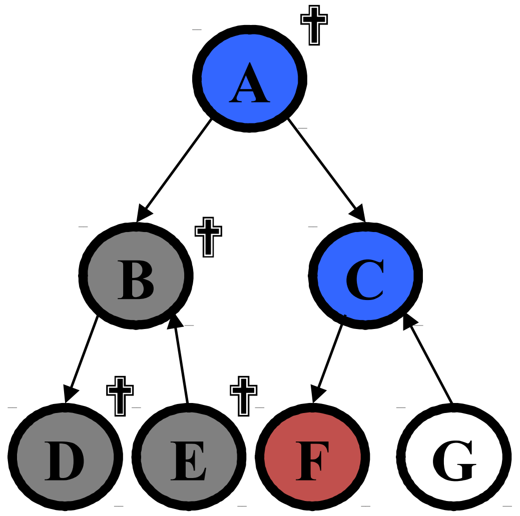


🔴 = current node 🔵 = processing (on the call stack)

⚫ = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```



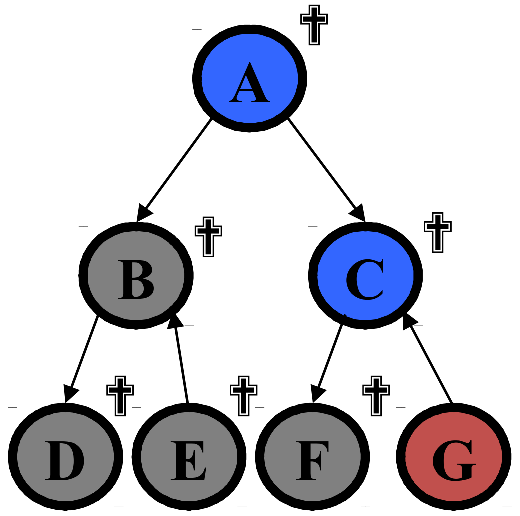
🔴 A = current node

🔵 A = processing (on the call stack)

⚫ A = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
  if(t != null) {
    inOrderTraversal(t.left);
    process(t.element);
    inOrderTraversal(t.right);
  }
}
```

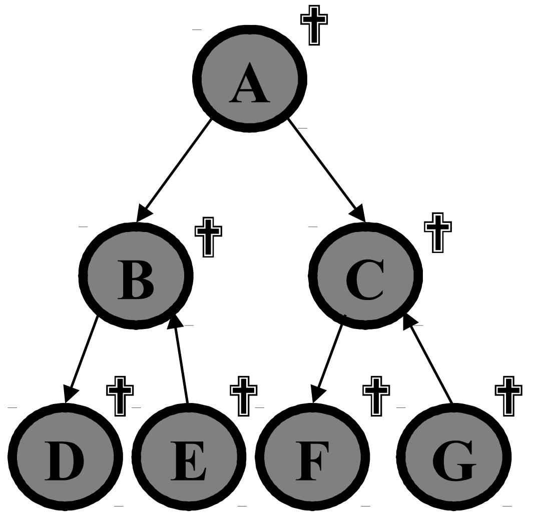


🔴 A = current node 🔵 A = processing (on the call stack)

⚫ A = completed node (element has been processed)

More on traversals

```
void inOrderTraversal(Node t){
   if(t != null) {
      inOrderTraversal(t.left);
      process(t.element);
      inOrderTraversal(t.right);
   }
}
```



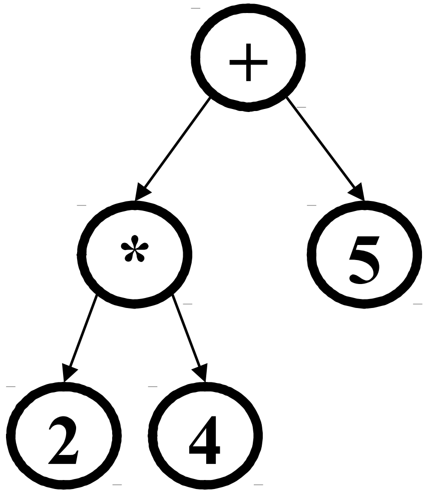
🔴 = current node 🔵 = processing (on the call stack)

⚫ = completed node (element has been processed)

Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

- ***Pre-order***: root, left subtree, right subtree
  + * 2 4 5
- ***In-order***: left subtree, root, right subtree
  2 * 4 + 5
- ***Post-order***: left subtree, right subtree, root
  2 4 * 5 +



(an expression tree)