ICSI 403
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 07 & 08 – Chapter 12 (Cormen), Binary Search Trees

J Marques de Carvalho

Dynamic Sets – Introduction

- See Cormen, pp. 229-231
- Dynamic sets are sets of data that can change over time.
  - Item(s) can be added to a set.
  - Item(s) can be deleted from a set.
  - Item(s) within a set can be modified (changed).

Dynamic Sets – Introduction

- Some dynamic sets are simple lookup mechanisms (dictionary).
  - Add / delete / lookup (test membership).
- Other dynamic sets are more complicated in terms of what they do.
- How we implement a dynamic set is highly dependent on what we will want it to do.
  - Stack / Queue / List.
  - Tree / Graph.

Dynamic Sets – Introduction

- Just as with the sorting problem, dynamic sets can store key values, which must remain associated with the corresponding satellite data.

Dynamic Sets – Introduction

- Some lists exist only to contain data, and there is no particular ordering of the data.
  - General linked lists.
- Some lists contain data to be maintained in a particular order (text, numeric, chronological).
  - Sorted linked lists (text/numeric).
  - Stacks / Queues (chronological).

Dynamic Sets – Introduction

- If a dynamic set is to contain ordered data, it needs to support "ordering" operations, such as:
  - Find the minimum (or maximum) value in the set.
  - From a given value, find the next (or previous) value in the set (if there *is* one, of course).

Dynamic Sets – Introduction

- Dynamic set operations can be divided into two broad categories:
  - *Queries* return information on the set.
  - *Modifying operations* change the set in some way.
- Our dynamic set will have to be able to handle a variety of different operators.
  - A given application may or may not need all the potential operations.
    - A set to which we only add items (like most dictionaries) does not need a DELETE operation.

Dynamic Sets – Introduction

- Dynamic Set Operations (1):
  - SEARCH(S, k) - A query that, given a set S and a key value k, returns a pointer x to an element in S such that $x.key = k$, or NIL if S contains no such element.
  - INSERT(S, x) A modifying operation that augments the set S with the element pointed to by x.
    - We usually assume that any fields in element x needed by the set implementation have already been initialized.

Dynamic Sets – Introduction

- Dynamic Set Operations (2):
  - DELETE(S, x) A modifying operation that, given a pointer x to an element in the set S, removes x from S.
    - Note: this operation uses a pointer to an element x, not a key value.
  - MINIMUM(S) A query on a totally ordered set S that returns the element of S with the smallest key.
  - MAXIMUM(S) A query on a totally ordered set S that returns the element of S with the largest key.

Dynamic Sets – Introduction

- Dynamic Set Operations (3):
  - SUCCESSOR(S, x) A query that, given an element x whose key is from a totally ordered set S, returns the next larger element in S, or NIL if x is the maximum element.
  - PREDECESSOR(S, x) A query that, given an element x whose key is from a totally ordered set S, returns the next smaller element in S, or NIL if x is the minimum element.

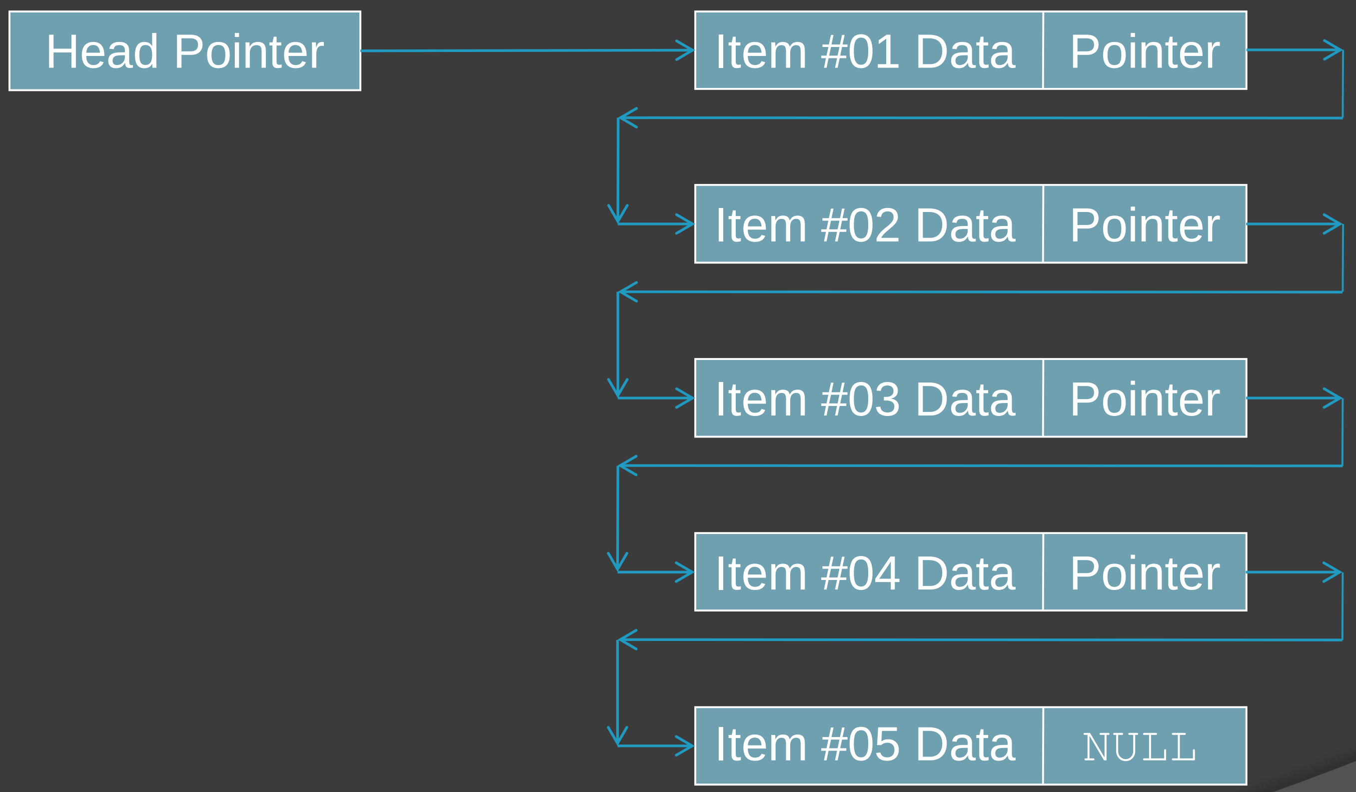
Dynamic Sets – Introduction

- Notes on Dynamic Set Operations:
  - The SUCCESSOR and PREDECESSOR queries are often extended to sets with nondistinctive keys. For a set of n keys, the normal presumption is that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in a sorted order.
  - The time taken to execute a set operation is usually measured in terms of the size of the set given as one of its arguments.

Binary Search Trees – Introduction

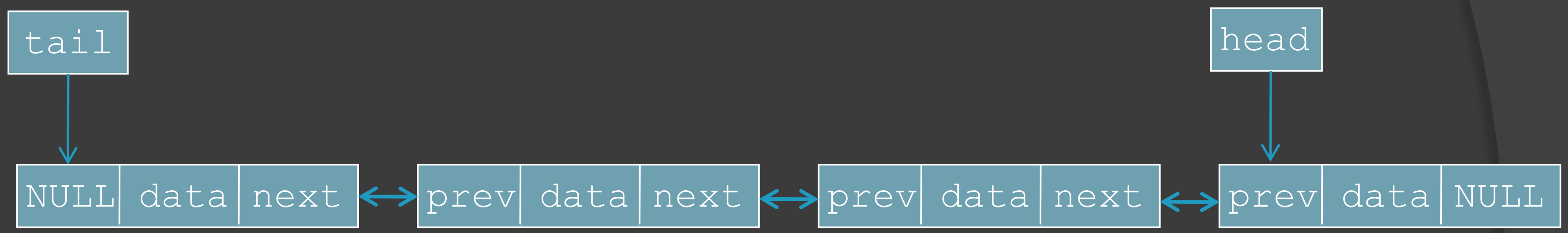
- Previous data structures (stacks, simple linked lists, and doubly-linked lists) implemented a chain of links we could follow forwards (singly-linked) and/or backwards (doubly-linked).
- These lists are one-dimensional (lines on which we can move) – *Linear Data Structures*

Linked Lists Form A Chain

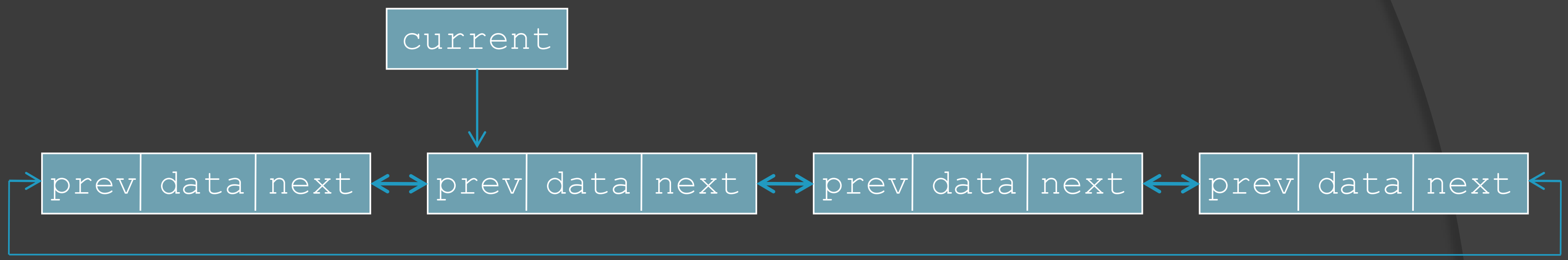


Doubly-Linked Lists

- Rather than `next` pointers, `prev` ***and*** `next`



Doubly-Linked Lists → Rings



Trees

- Trees are *<u>Non-linear Data Structures</u>*
- Examples of trees in everyday life:
  - Directory Structure on hard drive.
  - Family Tree.
    - We'll draw heavily from the family tree analogy as we talk about trees.
  - Decision structure in code.
- Just like a linked list, trees also consist of nodes, linked to each other with pointers (references) to other nodes.

Tree Node Structure

- Linked lists have, at a minimum, a data field and a pointer (to the next node) field.
- With the linear nature of the nodes in a chain, there is only *one* path through a linked list.
- Trees have, at a minimum, a data field and TWO pointers (to TWO 'next' nodes).
- With two possible successors to each node, we have multiple PATHS through a tree.
  - Multiple paths implies *branching.*

Tree Nodes

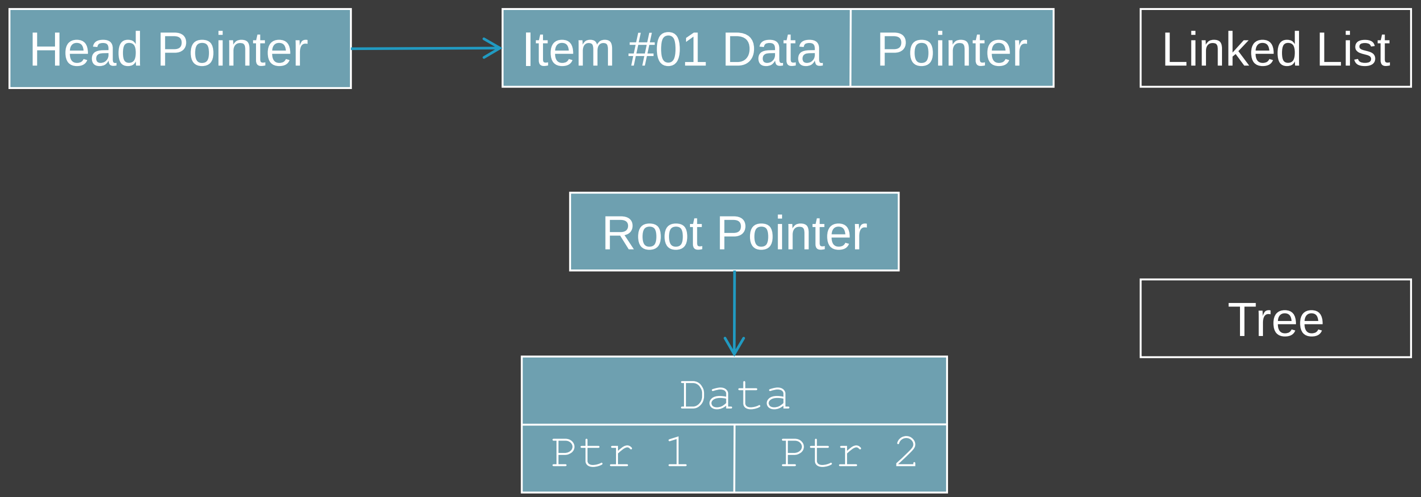
- Trees also introduce some new terms, so we'll start with some nomenclature:
- At its simplest:

Data	
Ptr 1	Ptr 2

- A tree with two successor pointers is called a *binary tree and* has a *branching factor* of 2.
- There *are* trees with more than two pointers (higher branching factors), but we will concentrate on binary trees for now.

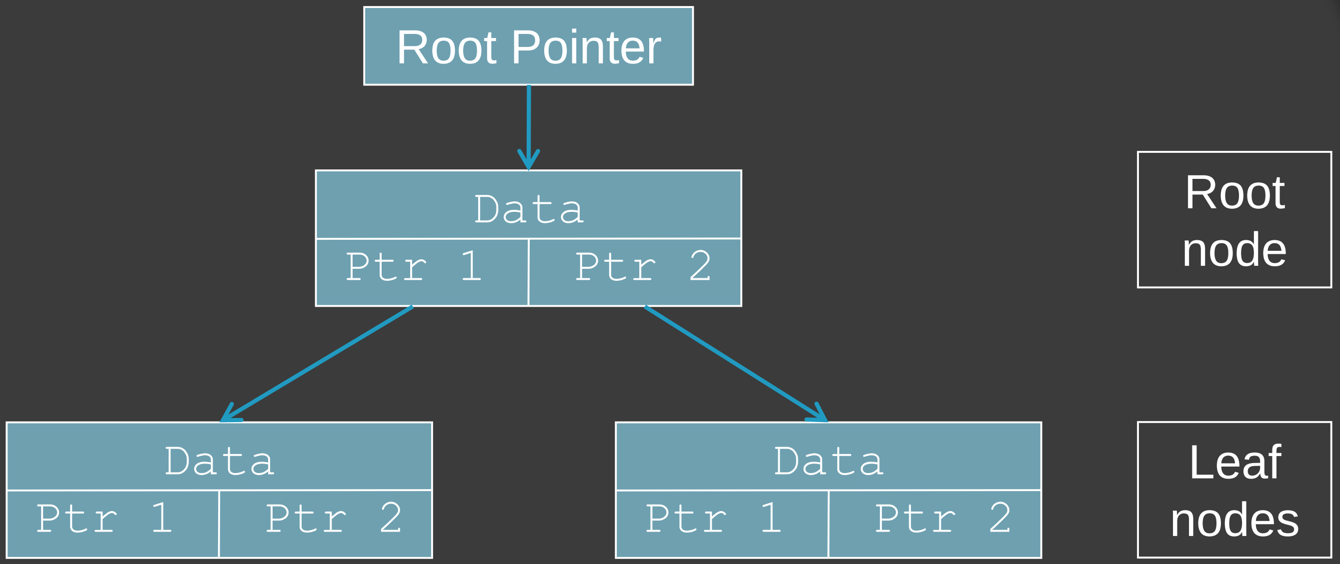
Starting a Tree

- Just as a linked list has a HEAD pointer, trees have a ROOT pointer.



- Unlike real trees, we draw data trees upside-down, with the root at the top.

Trees With More Than One Node

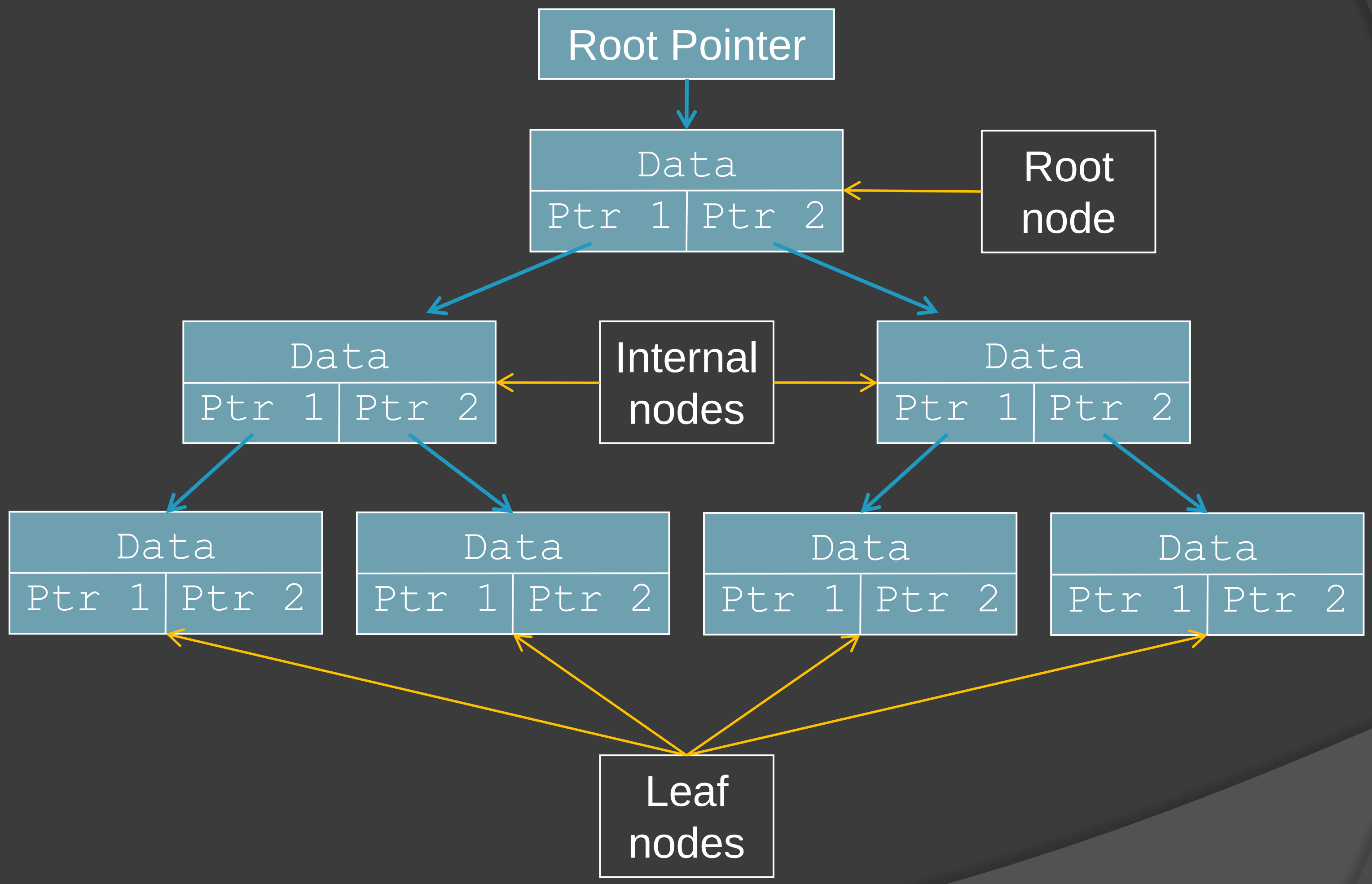


- If the ‘root’ is drawn at the top, then what we draw at the bottom must be ‘leaves’.

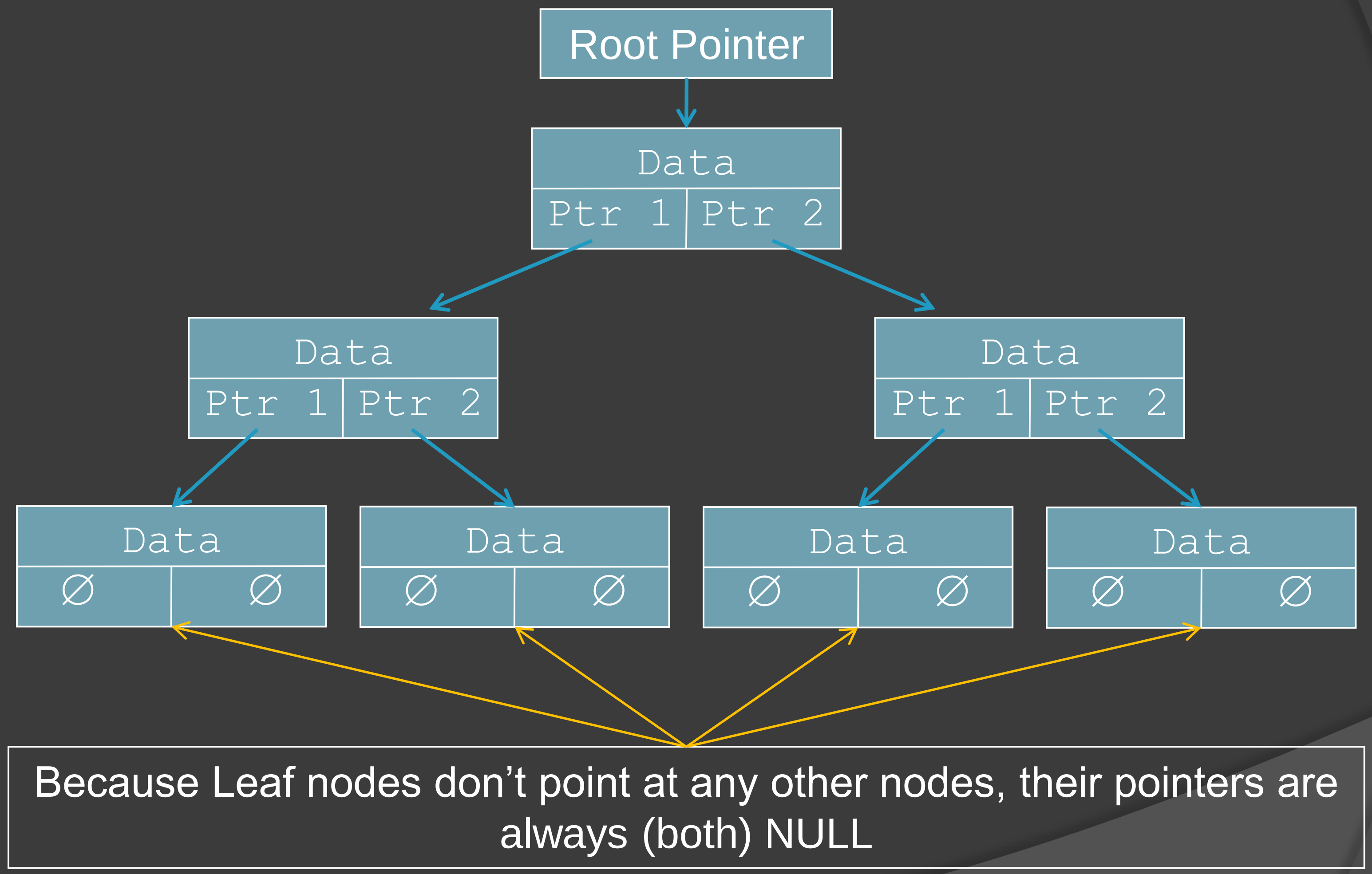
Parent Pointers

- The book gives the node structure as holding not only the key, satellite data, and left & right child pointers, but also a pointer (p) back to a node's <u>*parent.*</u>
- Only the root node doesn't have a parent (i.e., *root.p* is NIL).
  - The text uses NIL; we'll use NULL, too.
- Not all tree implementations require parent pointers; it simplifies coding some operations.

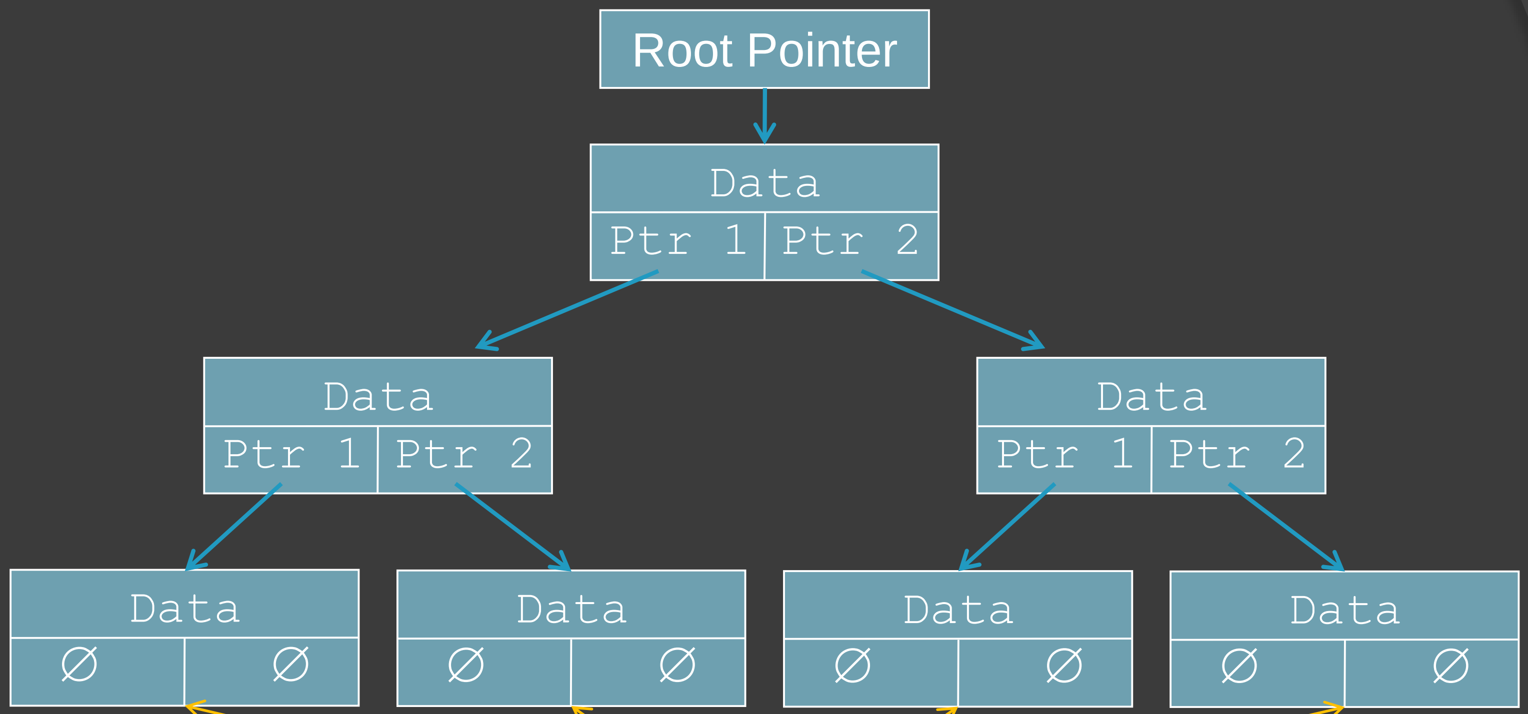
Trees With More Than One Node



Trees With More Than One Node

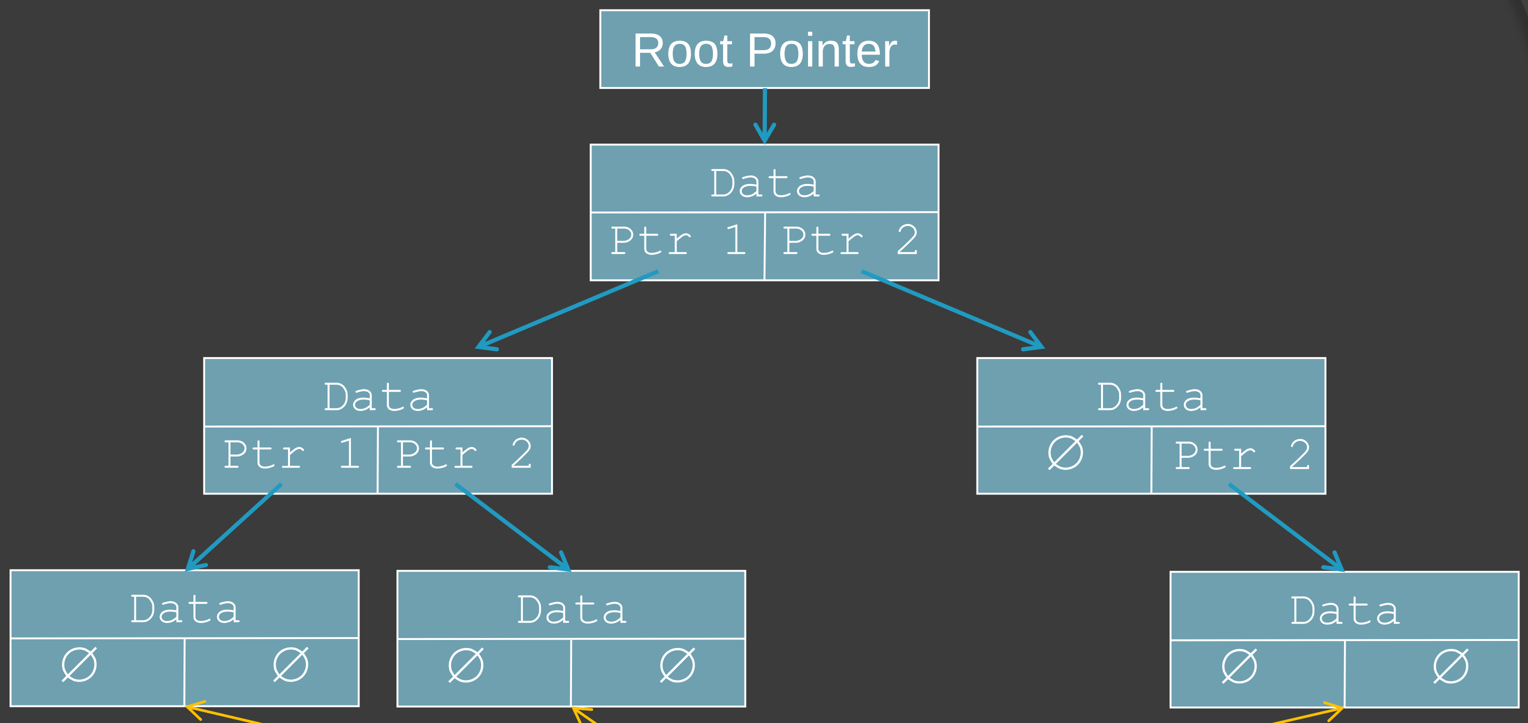


Trees With More Than One Node



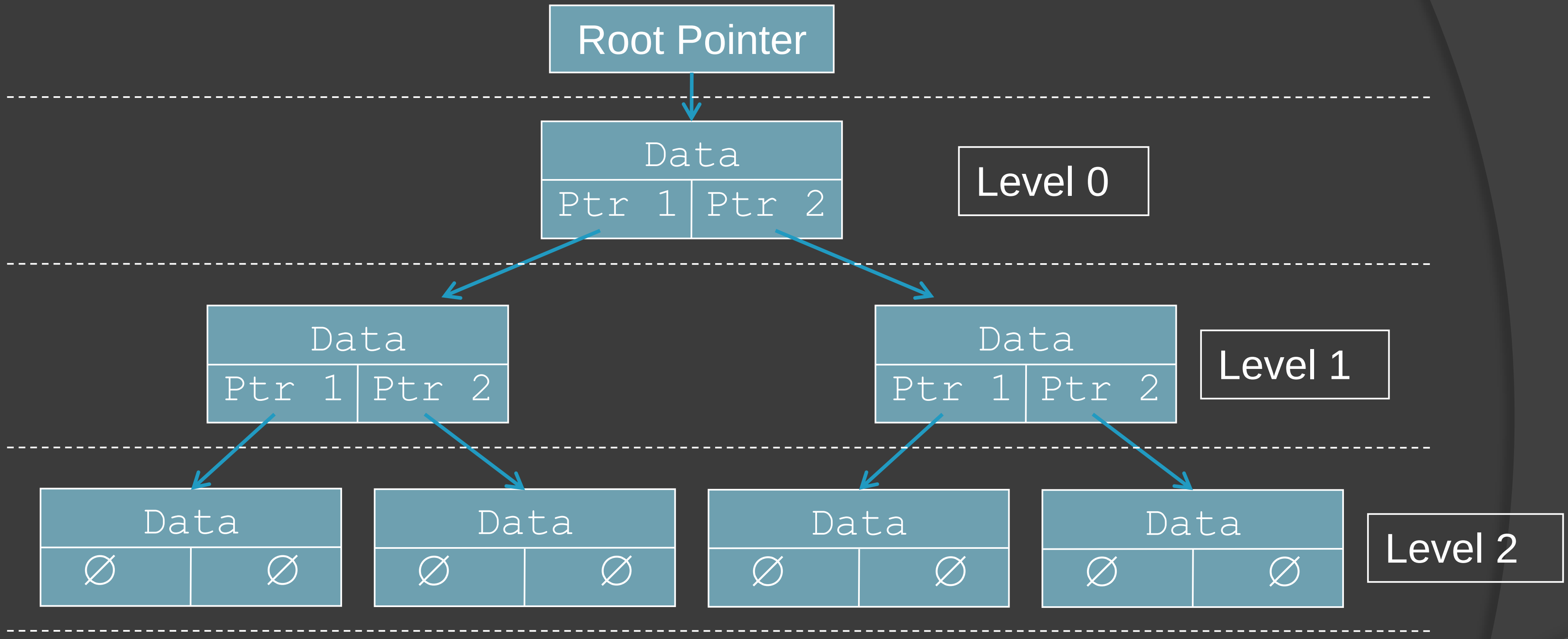
A tree in which only the leaf nodes have NULL pointers is called a *complete* binary tree

Trees With More Than One Node



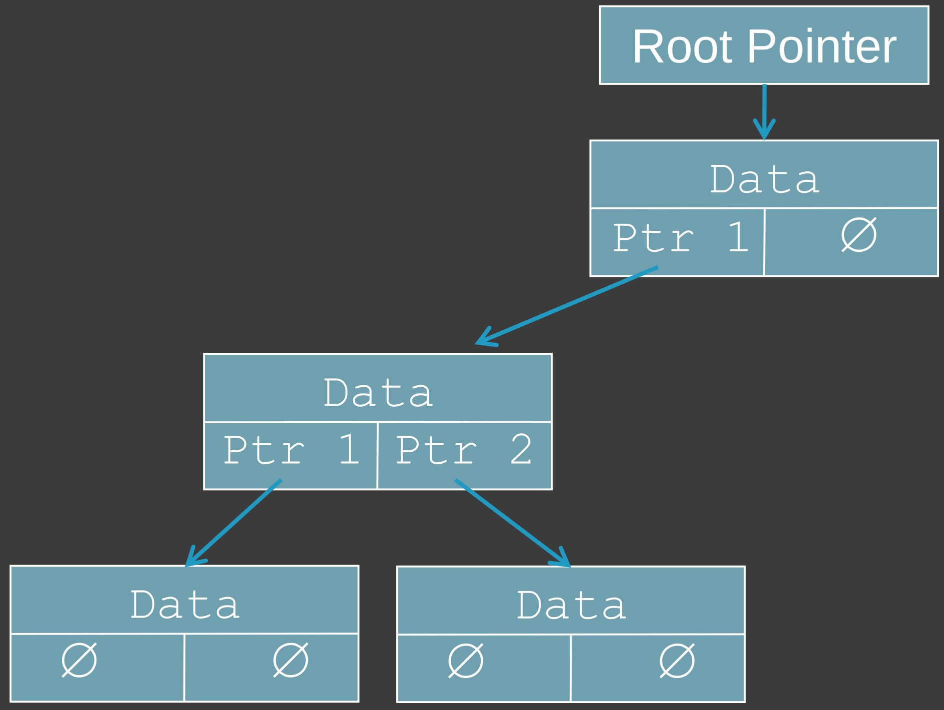
This Tree IS valid, but is NOT complete (there’s a NULL pointer in an internal [non-leaf] node)

Trees Have *Levels*, *Height*, or *Depth*



In general, a binary tree with n levels will have, at most, 2^n leaf nodes (in the case of a complete binary tree), and $2^{(n+1)} - 1$ nodes total. A complete binary tree with n nodes will have approximately $log_2(n)$ levels.

Trees With More Than One Node



Trees are *imblanced* if the height of the left and right subtrees differ

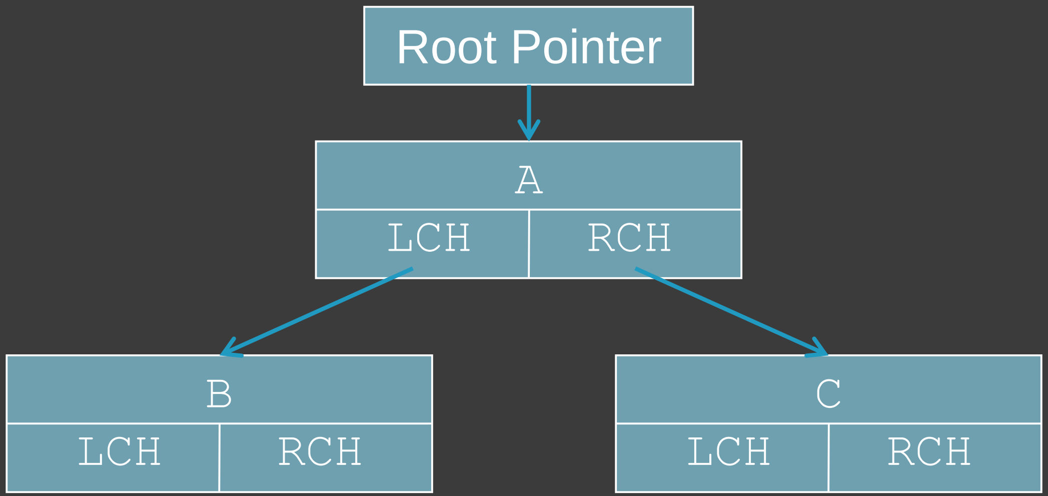
Child Nodes

- In keeping with the family tree analogy, the pointers in a binary tree's nodes are typically called the left and right *child* pointers.

Data	
LCH	RCH

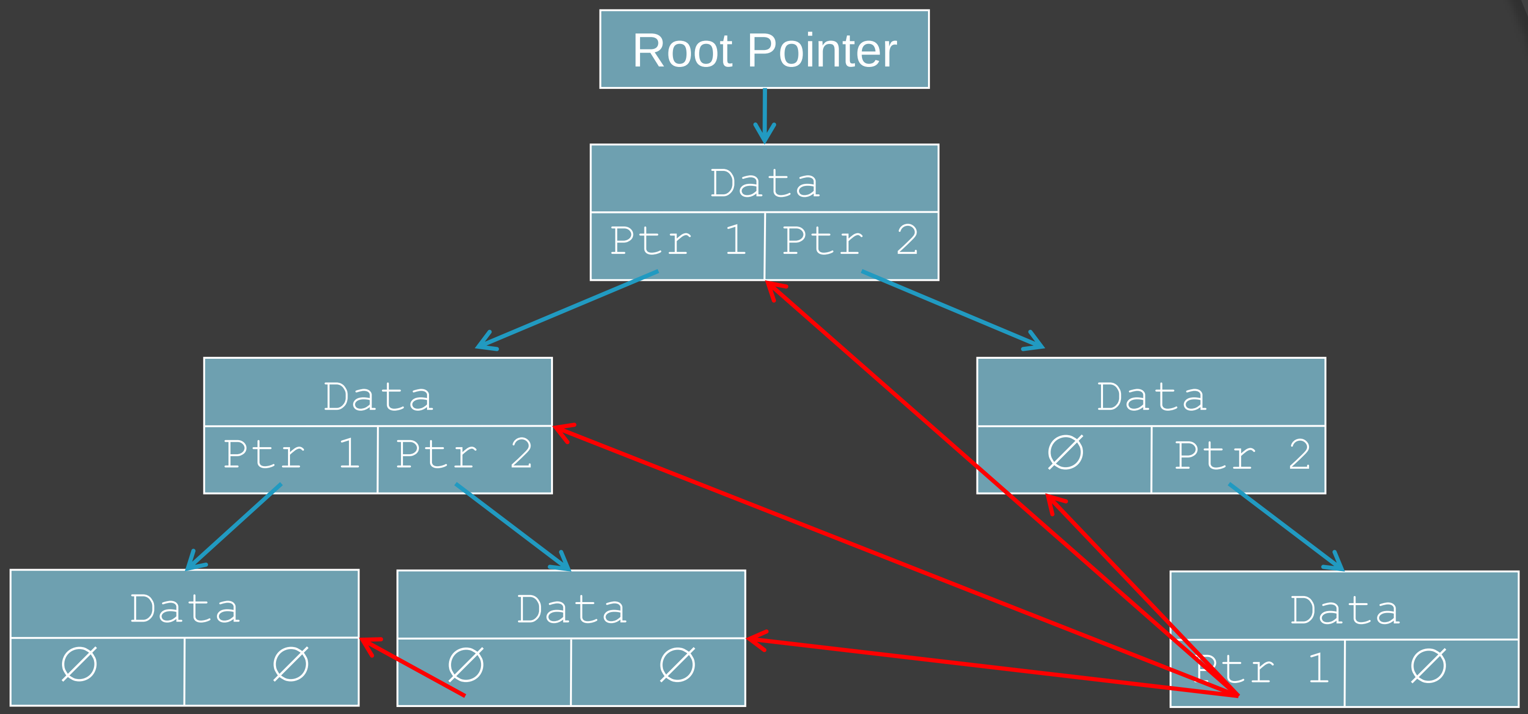
- Leaf nodes have no child nodes (both NULL)
- The root node and all internal nodes may have one or two child nodes

Continuing the Familial Analogy



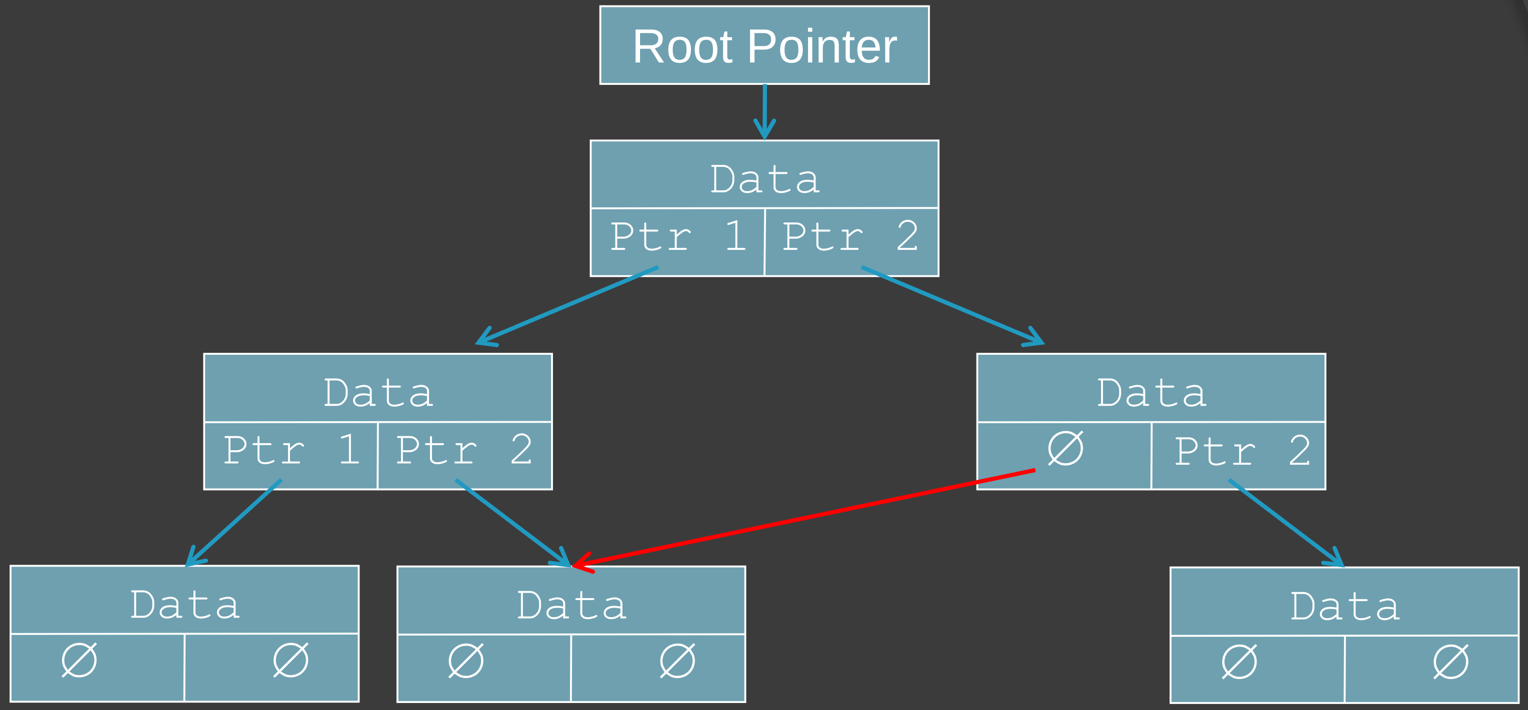
- Nodes `B` and `C` are the ***children*** of `A`
- `A` is the ***parent*** node of both `B` and `C`
- `B` and `C` are ***siblings***

Trees Can Not Contain *Cycles*



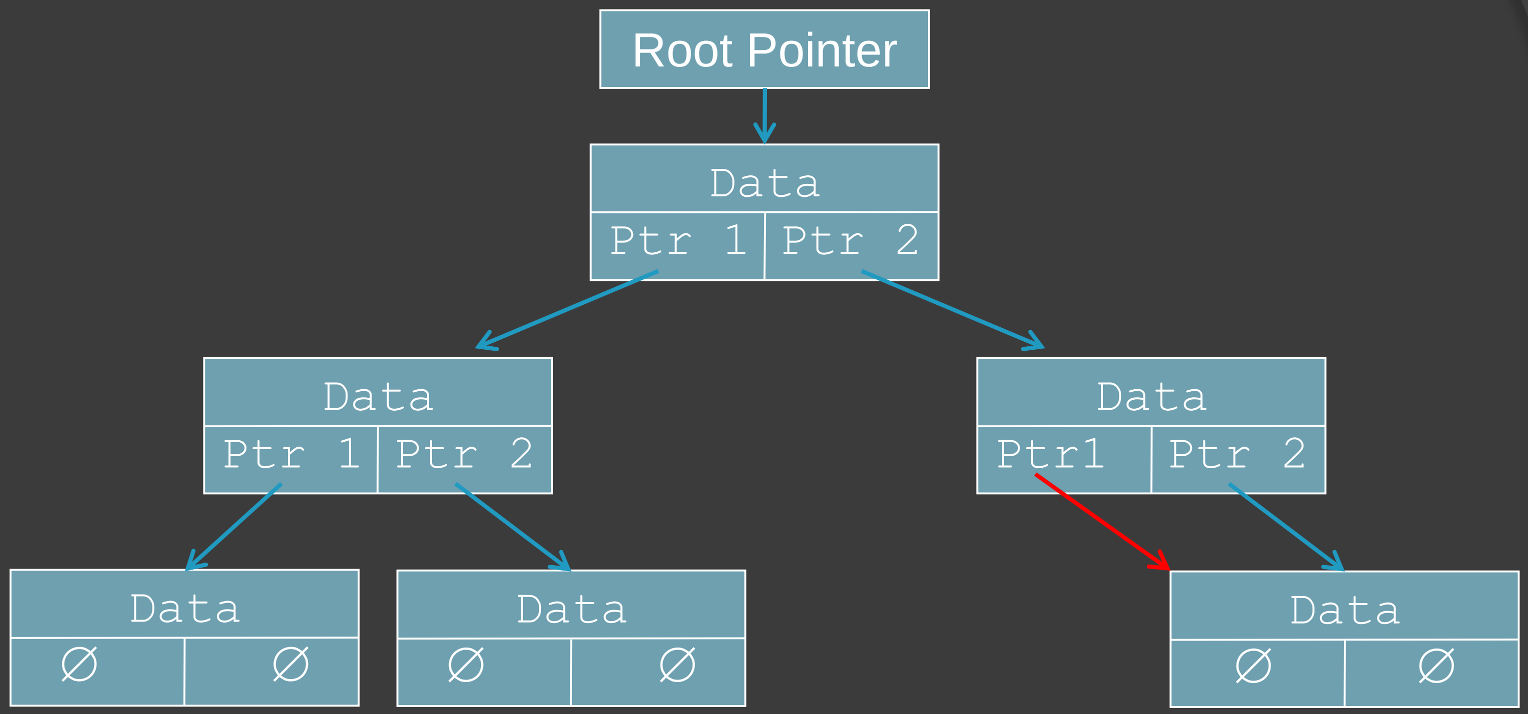
In general pointers can not point at any ancestor; nor can they go up to previous levels at all. These are all invalid links. Nodes may not point at siblings or cousins. These are invalid links as well. However, some algorithms have pointer to ancestor nodes.

Nodes Must Have Single Parents



A node must have exactly one parent. Two nodes can not point at the same child node. This is an invalid link. *__Links must point at a single child or at nothing at all.__*

Child Pointers Must Be Different



Unless we're considering a leaf node (both child pointers NULL), the two child pointers of any node must be different (i.e., point at different nodes, or one can be NULL, but they can not both point at the same node. This is an invalid link.

So Why Use TWO Successor Pointers?

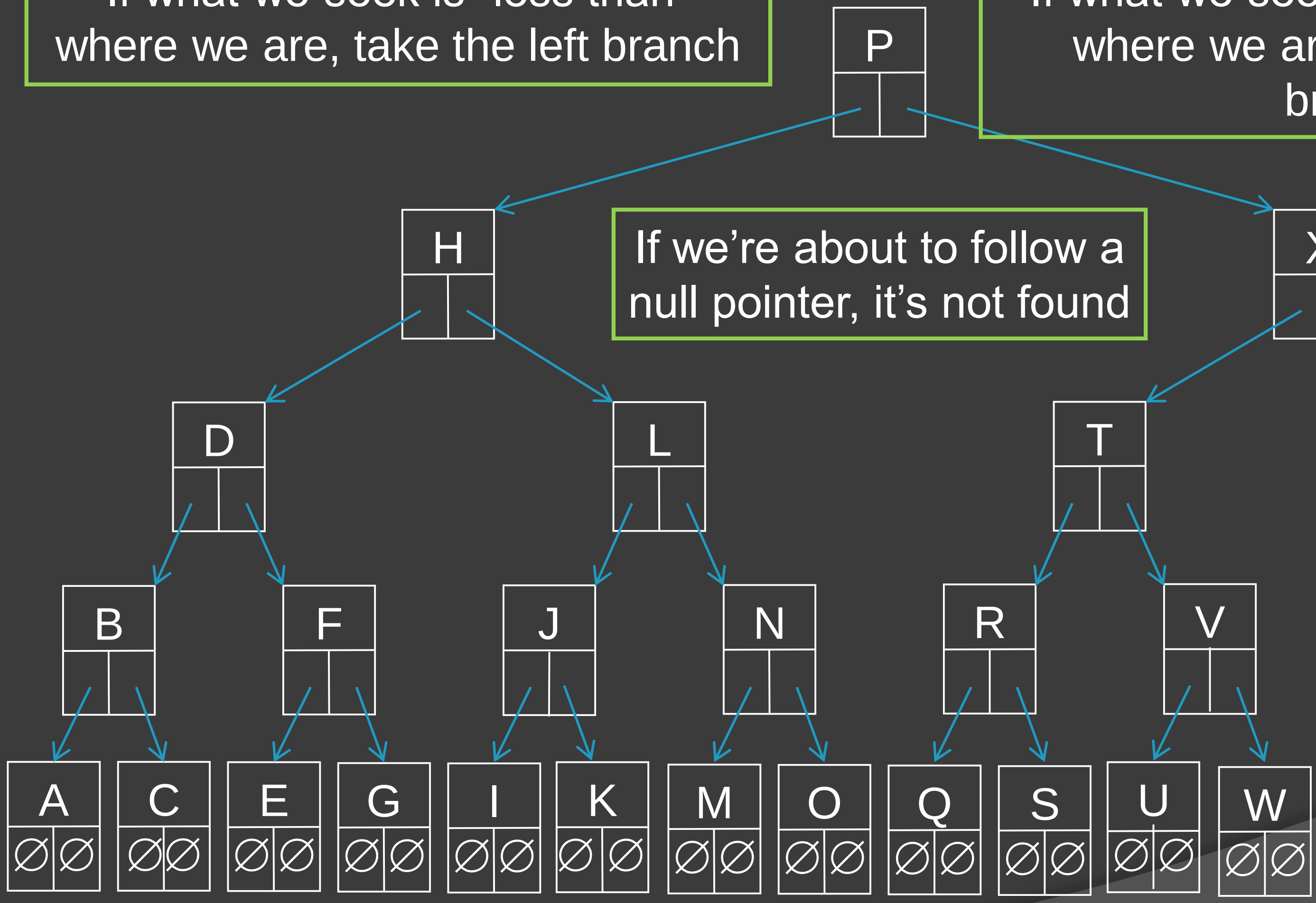
- Trees typically are used to maintain *ordered* data.
- At the root, we can follow either the branch for the left child or the right child
- If the tree is more-or-less balanced, we eliminate more-or-less half of the nodes as soon as we take the first branch.
- Sound familiar?
- Binary Search

Binary Search Tree

If what we seek is “less than” where we are, take the left branch

If what we seek is “greater than” where we are, take the right branch

If we’re about to follow a null pointer, it’s not found



So, to Recap…(1)

- Introduction to Trees.
  - Linked lists are linear structures with *one path.*
  - Trees have *multiple paths* through them.
- Trees consist of *nodes.*
  - Nodes have a *data* portion and *two* pointers:
    - *Left Child* Pointer.
    - *Right Child* Pointer.
    - *Parent* pointer (may or may not be implemented).
  - Nodes follow the family tree metaphor, with:
    - *Ancestors*, *descendants*, *children*, *parents*, *siblings.*

So, to Recap...(2)

- Trees are drawn upside-down.
  - The *<u>root node</u>* is at the top.
  - The *<u>leaf nodes</u>* (those without children) are at the bottom.
  - Everything else is an *<u>internal node.</u>*
- One way we measure trees is by *<u>height.</u>*
  - The root is *<u>level</u>* 0.
  - The child(ren) of the root are at *<u>level</u>* 1.
  - The grandchild(ren) of the root are at *<u>level</u>* 2…
  - Longest path through the tree (root→leaf): its *<u>height.</u>*

So, to Recap…(3)

- Trees of height n have (at most):
  - 2^n leaves .
  - $2^n - 1$ non-leaf nodes.
  - $2^{n+1} - 1$ nodes total.
- <u>*Complete*</u> binary trees have no internal nodes with null child pointers – the only NULL pointers are at the leaves.
- Trees are <u>*balanced*</u> if the height of the left and right subtrees of all nodes are equal.

How Do We Navigate Through A Tree?

- Trees typically are used to maintain *ordered* data.
- At the root, we can follow either the branch for the left child or the right child.
- If the tree is more-or-less balanced, we eliminate more-or-less half of the nodes as soon as we take the first branch.
- Binary Search!

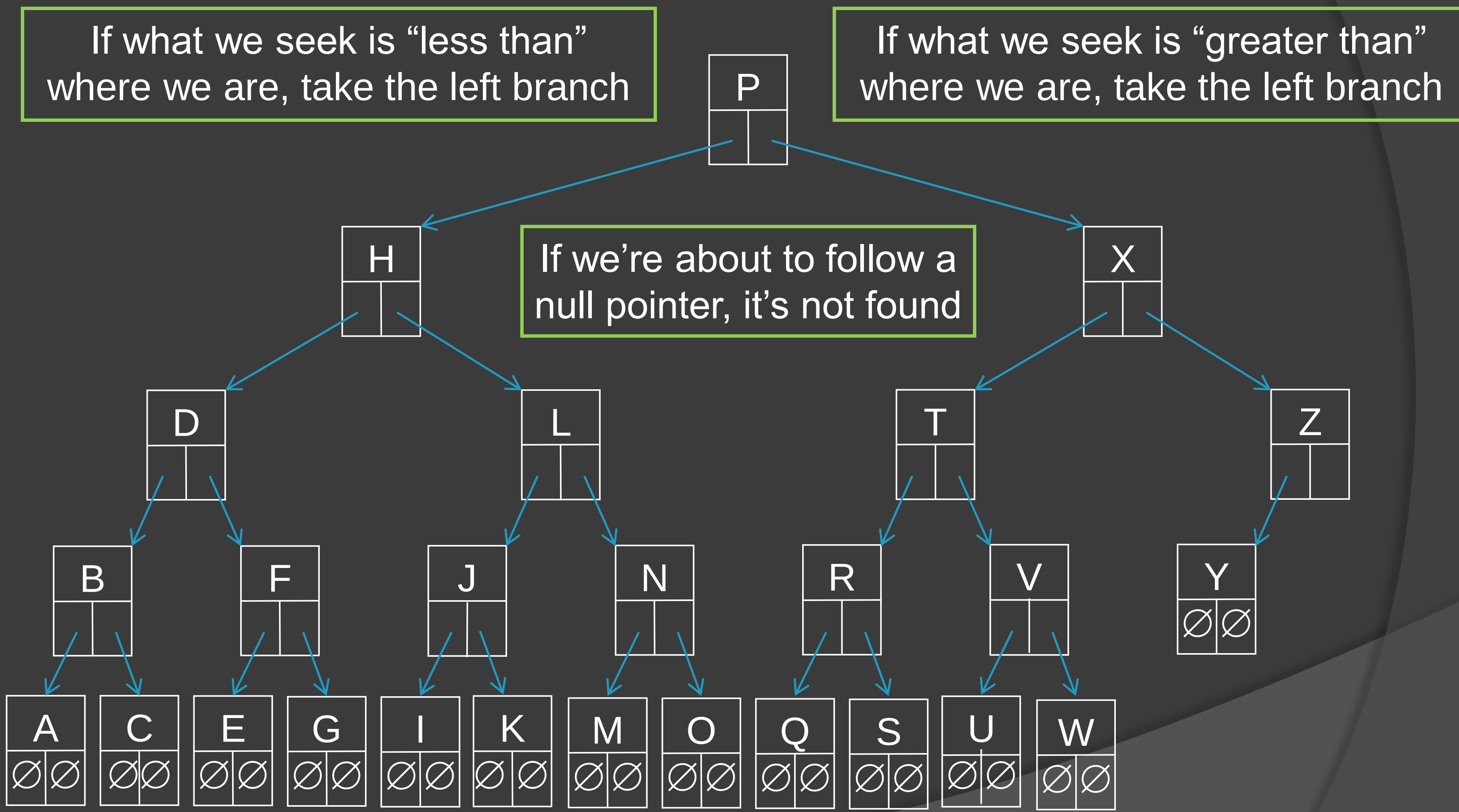
The Binary Search Tree Property

- Trees typically are used to maintain <u>*ordered*</u> data.
- There are other uses for trees, as we will see, that do NOT maintain ordered data. Trees with ordered data have the ***Binary-Search Tree Property:***

  If `y` is in the left subtree of `x`, then `key[y]` ≤ `key[x]`

  If `y` is in the right subtree of `x`, then `key[y]` ≥ `key[x]`

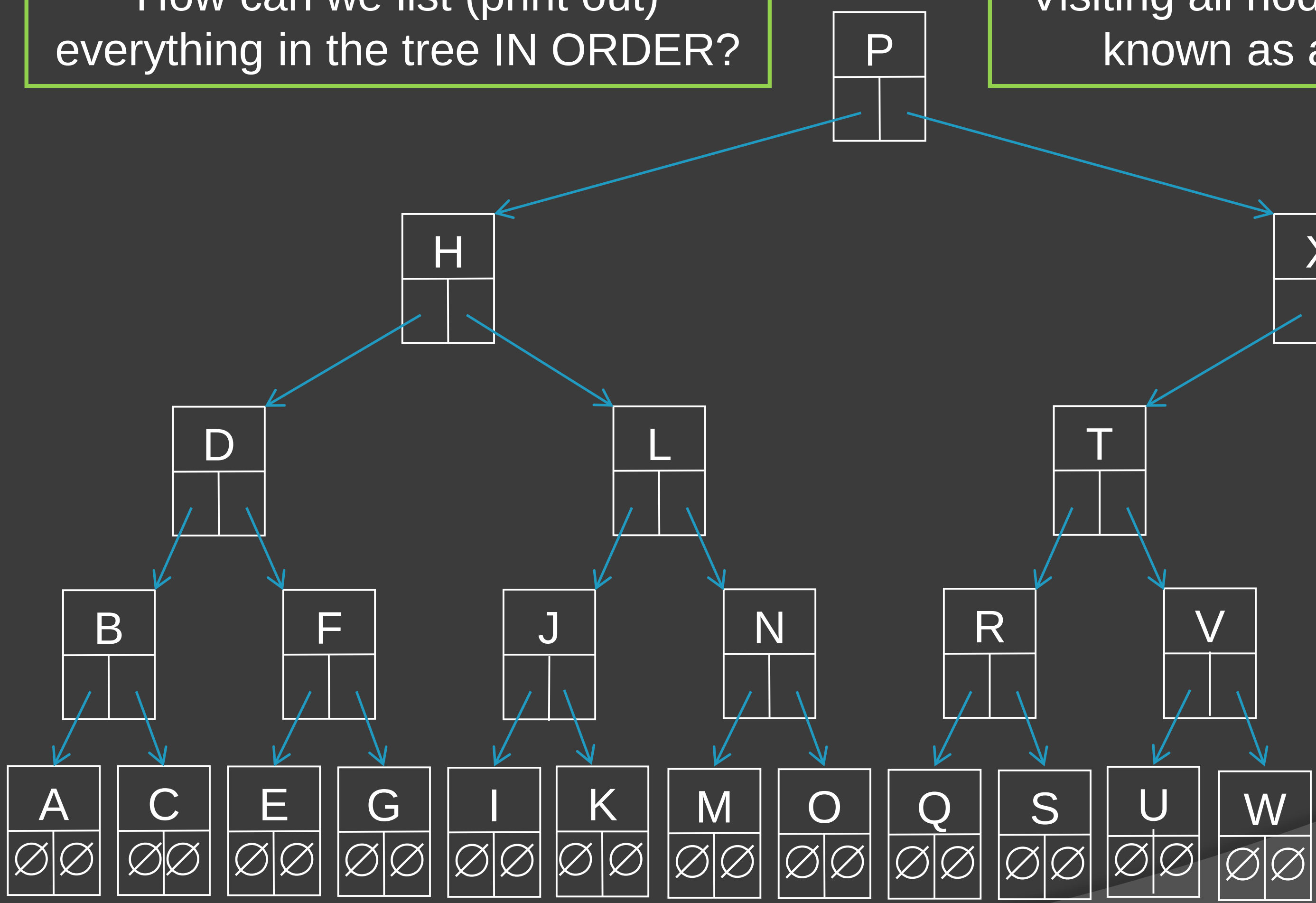
Binary Search Tree



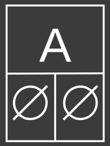
Listing the Contents of a Binary Search Tree

How can we list (print out) everything in the tree IN ORDER?

Visiting all nodes in a tree is known as a *traversal*

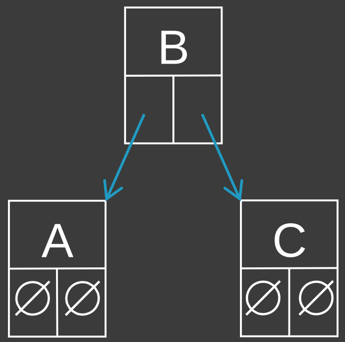


Binary Tree Traversal



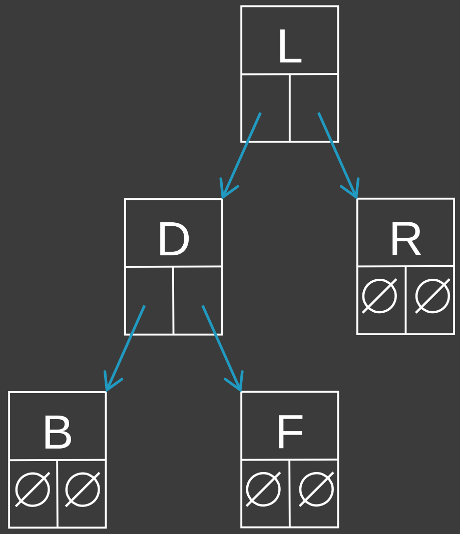
- One node
  - Print out the contents of the node.

Binary Tree Traversal



- Three nodes
  - Start at the root (the only place we *can* start).
  - Print what's in the left-child node (A).
  - Print what's in the root node (B).
  - Print what's in the right-child node (C).

Binary Tree Traversal

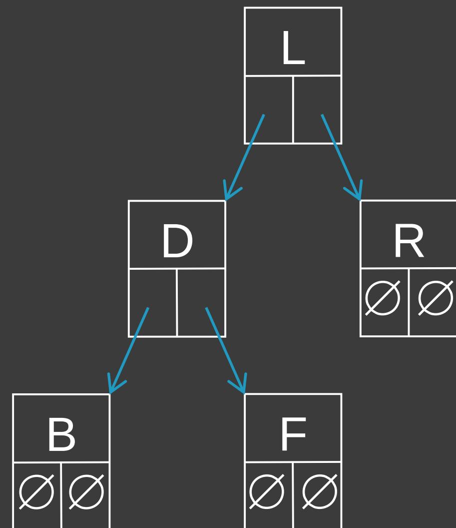


- Five nodes.
  - Start at the root (the only place we <u>*can*</u> start).
  - Print the contents of the left sub-tree (B D F).
  - Print the contents of the root node (L).
  - Print the contents of the right sub-tree (R).

Binary Tree Traversal and Recursion

Do we process node D any differently than how we process node L?

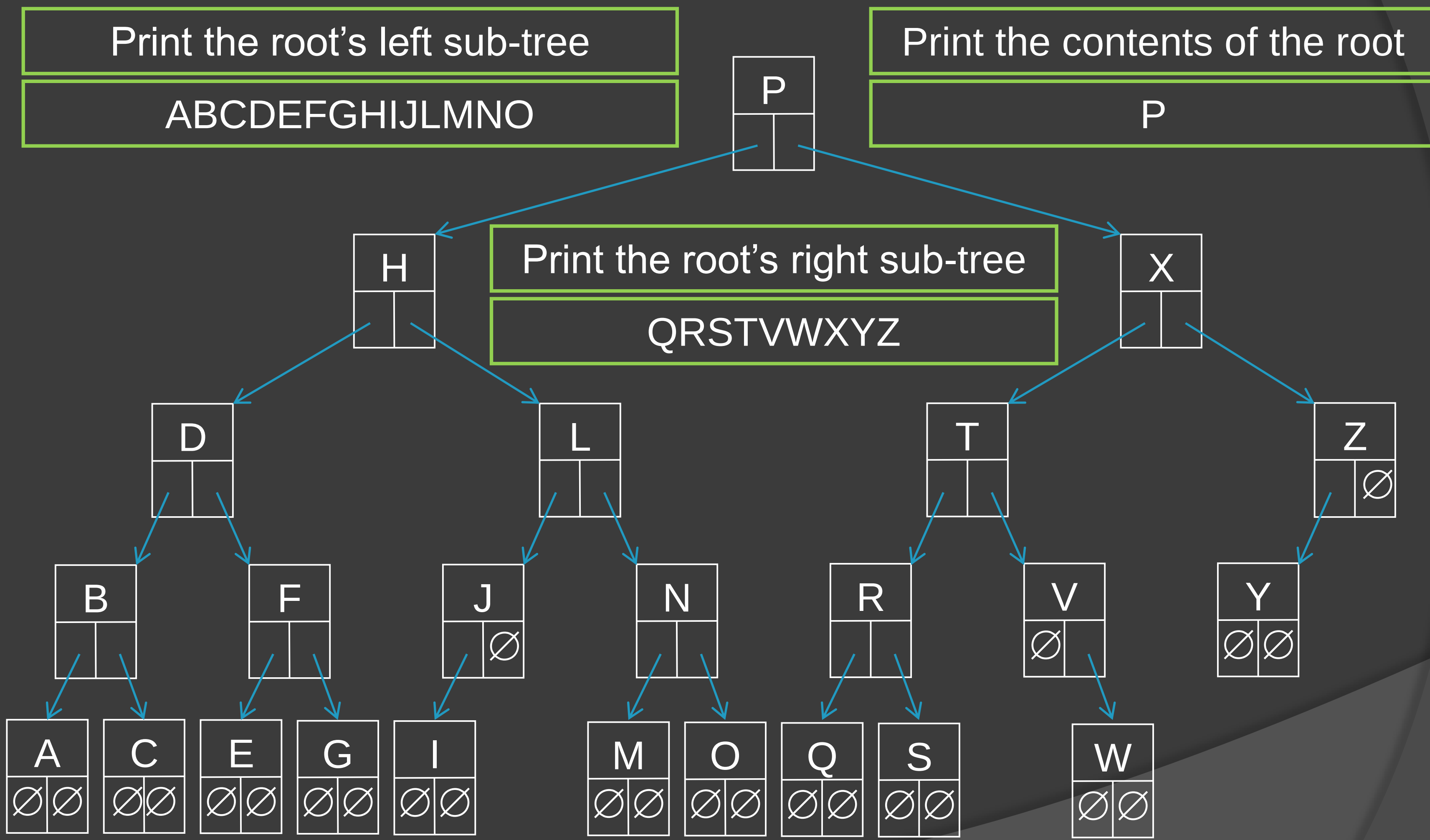
D is the root of a sub-tree, just as L is the root of the whole tree.



B, F, and R are *also* roots of sub-trees.

- Start at the root (the only place we *can* start).
- Print the contents of the left sub-tree (B D F).
- Print the contents of the root node (L).
- Print the contents of the right sub-tree (R).

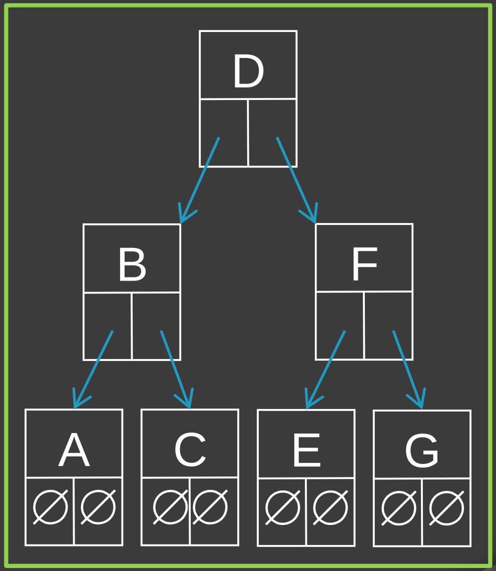
Binary Tree Traversal



Recursive Binary Tree Traversal

- Algorithm (call with `Traverse(root)`):

```
Traverse(node) {
  if (node==NULL) return;
  Traverse(node->LChild);
  print node->data;
  Traverse(node->RChild);
}
```



Recursive Binary Tree Traversal

```
Traverse(node) {
   if (node==NULL) return;
   Traverse(node->LChild);
   print node->data;
   Traverse(node->RChild);
}
```

Slightly more efficient (why?):

```
Traverse(node) {
   if (node->LChild !=NULL) Traverse(node->LChild);
   print node->data;
   if (node->RChild !=NULL) Traverse(node->RChild);
}
```

Recursive Binary Tree Traversal

```
Traverse(node) {
   if (node==NULL) return;
   Traverse(node->LChild);
   print node->data;
   Traverse(node->RChild);
}
```

Consider the order:

Left sub-tree, root, right sub-tree.

This is known as an *in-order* traversal.

Root, left sub-tree, right sub-tree: *pre-order.*

Left sub-tree, right sub-tree, root: *post-order.*

Recursive Binary Tree Traversal

- The book calls a traversal a "walk"

INORDER-TREE-WALK(x)

if x ≠ NIL

    INORDER-TREE-WALK($x.left$)

    print $x.key$

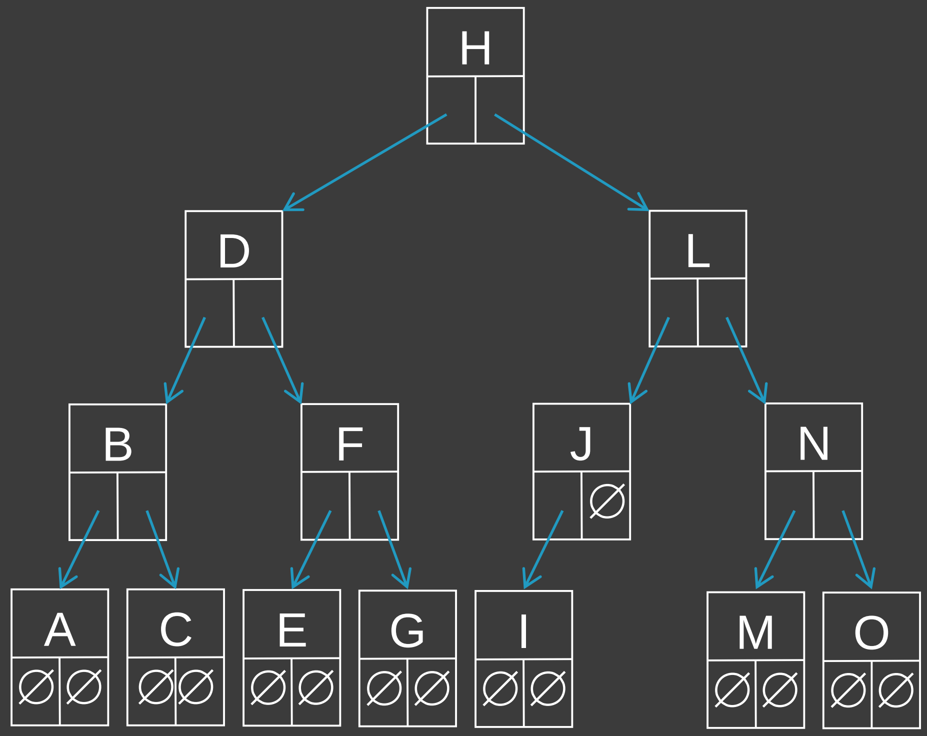
    INORDER-TREE-WALK($x.right$)

Recursive Binary Tree Traversal

PREORDER-TREE-WALK(*x*)
if *x* ≠ NIL
    print *x.key*
    PREORDER-TREE-WALK(*x.left*)
    PREORDER-TREE-WALK(*x.right*)

POSTORDER-TREE-WALK(*x*)
if x ≠ NIL
    POSTORDER-TREE-WALK(*x.left*)
    POSTORDER-TREE-WALK(*x.right*)
    print *x.key*

Binary Tree Traversals



In-order Traversal:

ABCDEFGHIJLMNO

Pre-order Traversal:

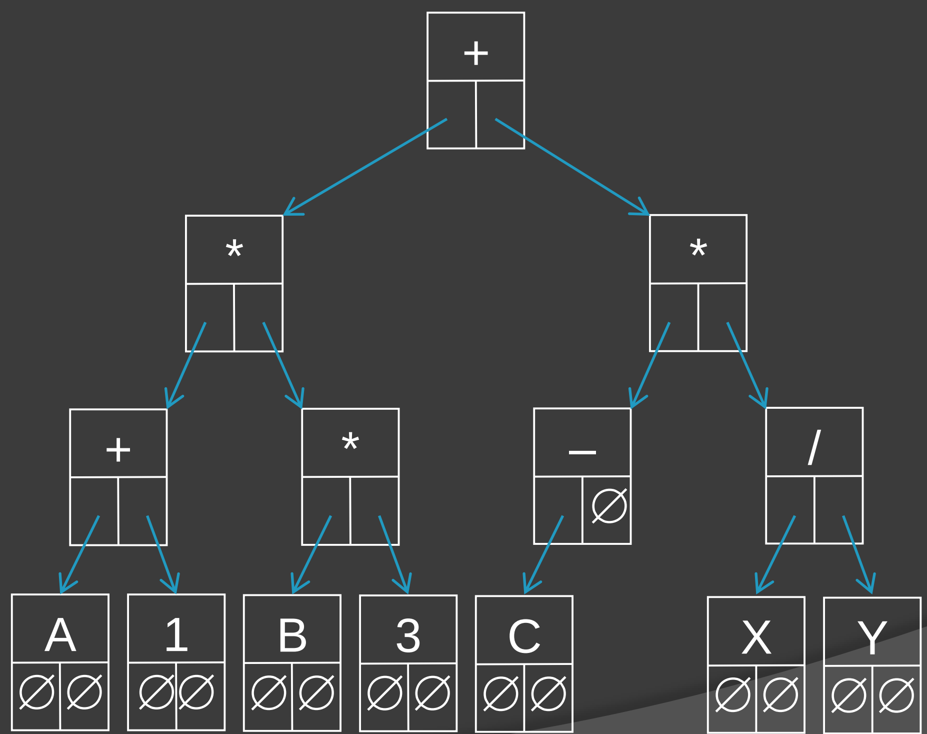
HDBACFEGLJINMO

Post-order Traversal:

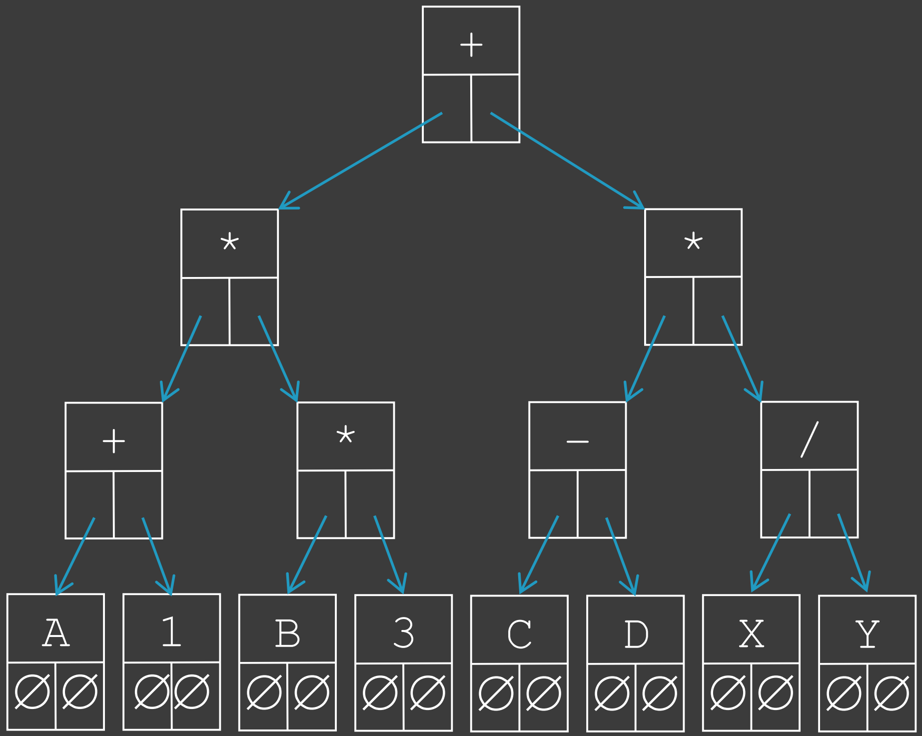
ACBEGFDIJMONLH

So, What Can We DO With Trees?

- Suppose we have a binary tree where:
  - Leaves contain variables or constants
  - Non-leaf nodes contain operators



So, What Can We DO With Trees?

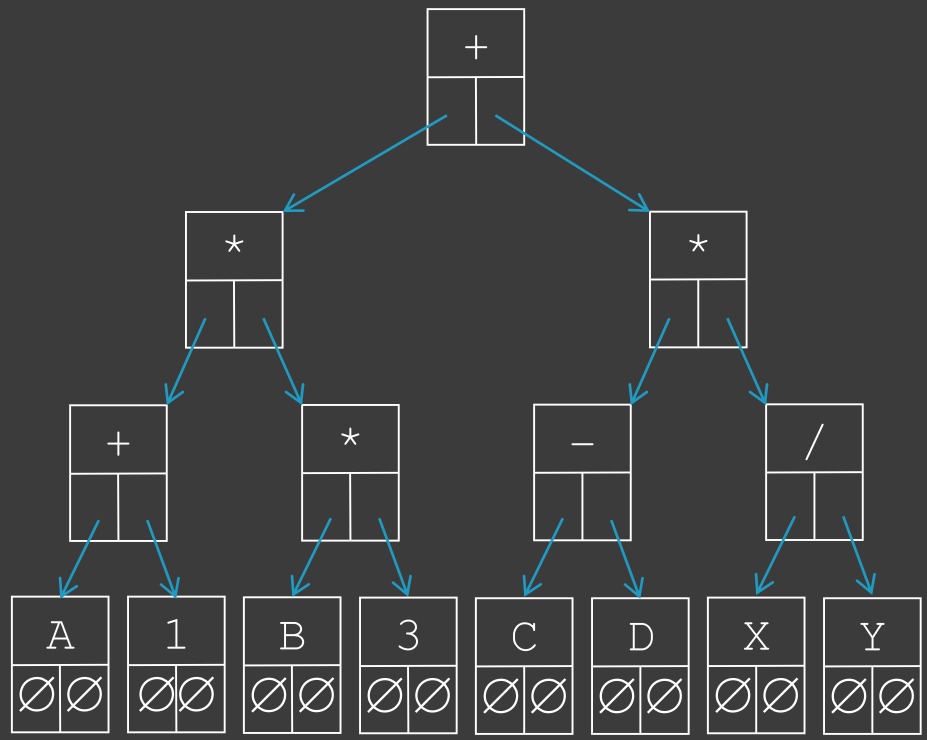


- Post-order traversal:
  - A 1 + B 3 * * C D – X Y / * +

Slightly Revised In-Order Traversal

```
Traverse(node) {
  print "(";
  if (node->Lchild!=NULL) {
     print "(";
     Traverse(node->LChild);
  }
  print node->data;
  if (node->Rchild!=NULL) {
     print "(";
     Traverse(node->RChild);
  }
  print ")";
}
```

So, What Do We DO With Trees?



- Modified In-order traversal:
  - (((A+1)*((B*3)))+(((C-D))*((X/Y)))

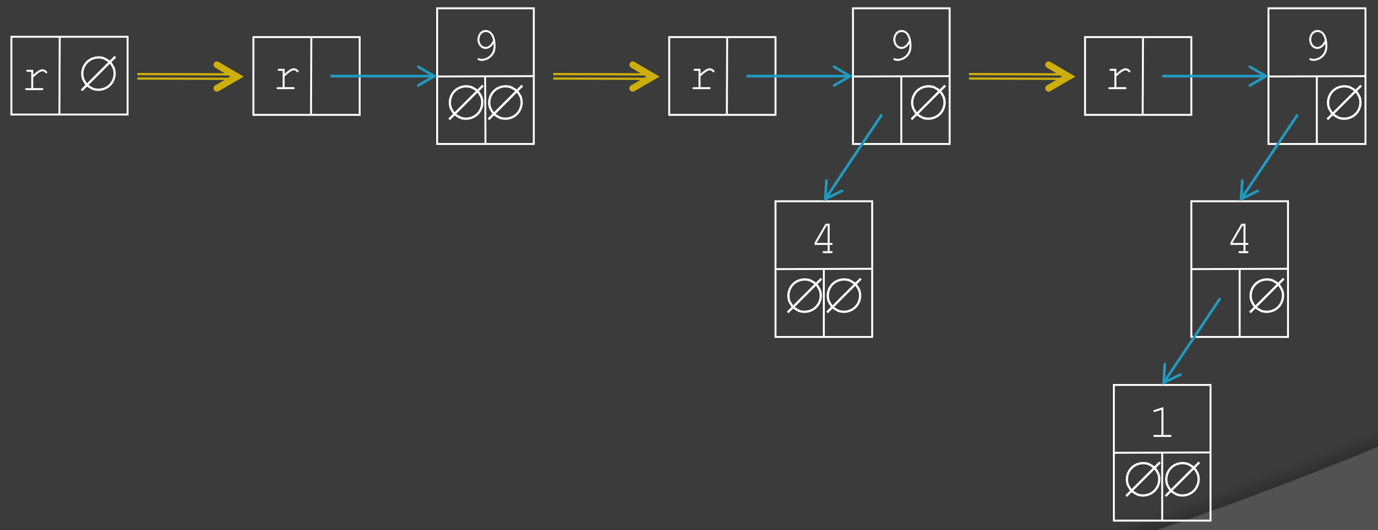
Building a Binary Tree

- We've seen how to search a binary tree.
- We've seen how to traverse a binary tree.
- How do we BUILD a binary tree?
- Just like any other linked data structure.
  - If the root pointer is NULL, allocate a new node and make the root pointer point to it.
  - Otherwise, determine where it SHOULD go, and make it a child of wherever it belongs.
    - Details to follow.

Let's Create A Binary Tree

- Assume we're starting with an empty tree.
- Insert the following values into a binary tree:

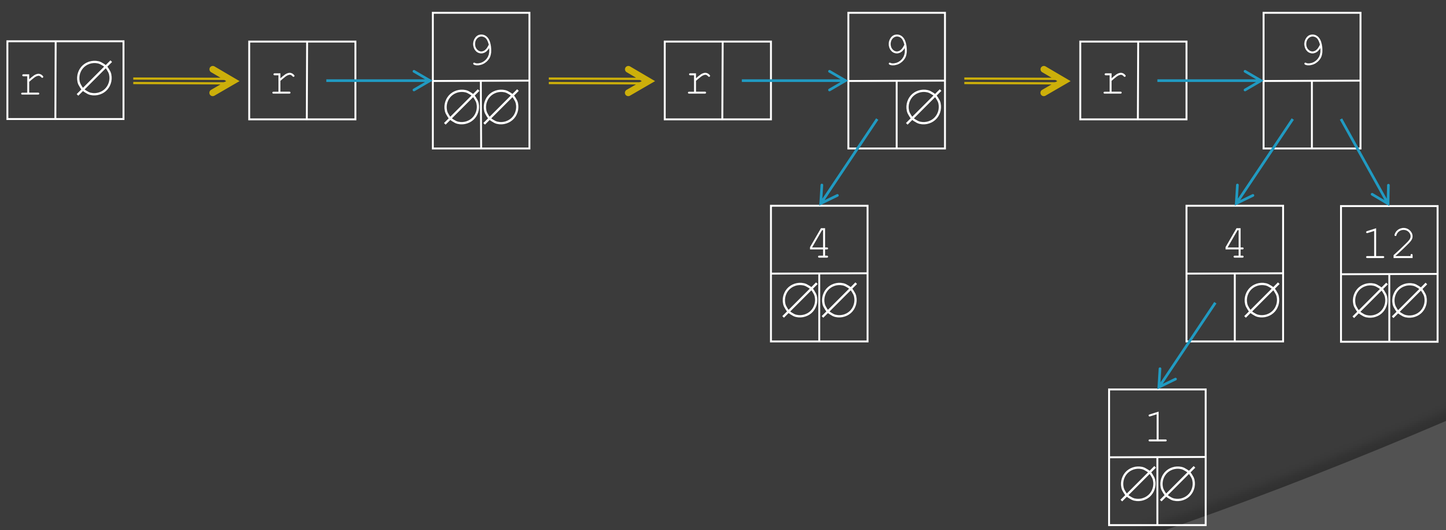
  9, 4, 1, 12, 16, 13



Let’s Create A Binary Tree

- Assume we’re starting with an empty tree.
- Insert the following values into a binary tree:

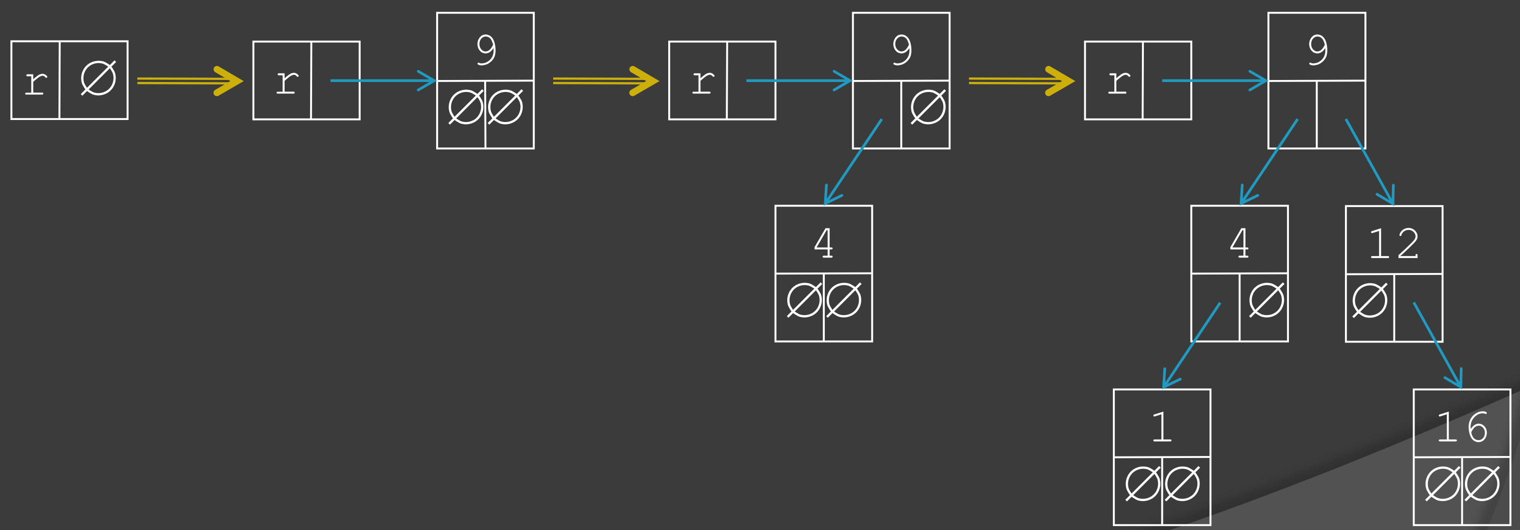
  9, 4, 1, 12, 16, 13



Let’s Create A Binary Tree

- Assume we’re starting with an empty tree.
- Insert the following values into a binary tree:

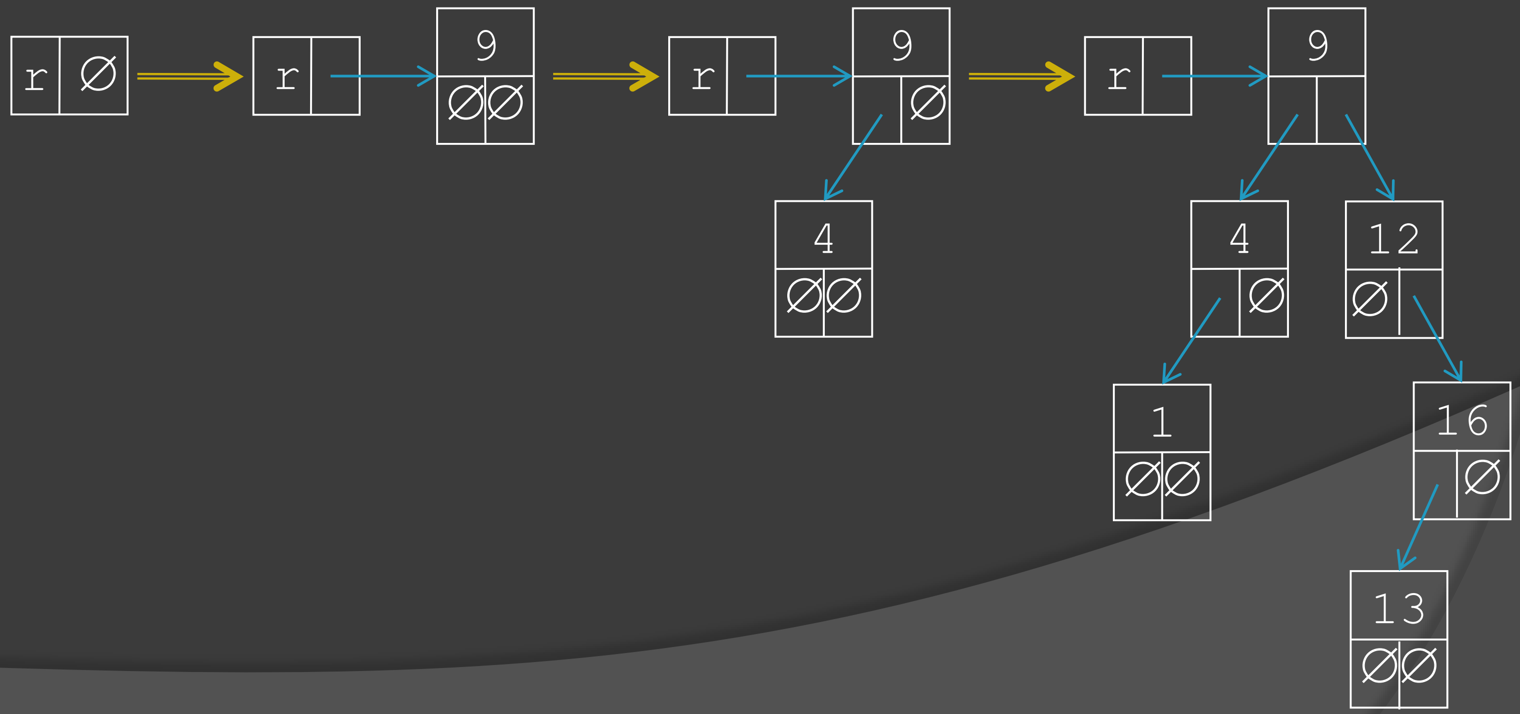
  9, 4, 1, 12, 16, 13



Let's Create A Binary Tree

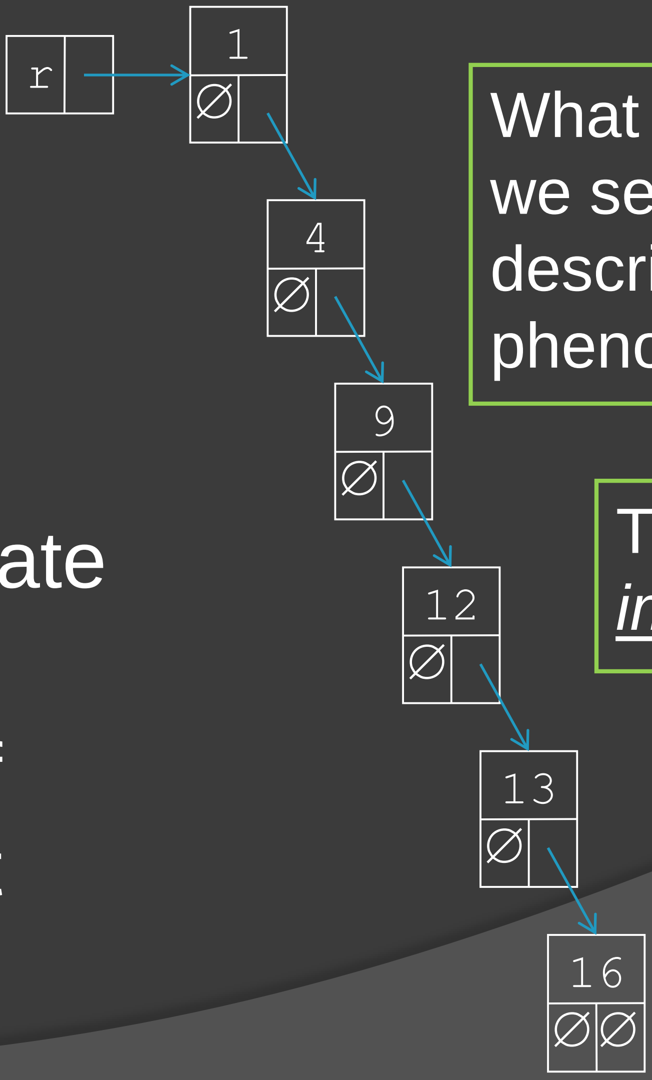
- Assume we're starting with an empty tree.
- Insert the following values into a binary tree:

  9, 4, 1, 12, 16, 13



Order Becomes Important!

- Insert the same values into a binary tree in a new order: 1, 4, 9, 12, 13, 16
- Trees <u>*can*</u> degenerate into linked lists!
- Now the niceties of binary search don't work!



What term have we seen that describes this phenomenon?

The tree is <u>*imbalanced.*</u>

Handling Imbalanced Trees

- The preceding walk-through was to give you an idea of the process; we'll come back to precisely how to do insert (and delete) shortly.
- Later on, we will see a couple of methods for handling the imbalance problem, such that we can insert nodes *<u>in any order</u>* and still maintain an almost-balanced binary tree.

Those Dynamic Set Operators Again

- ~~Search~~
- Insert
- Delete
- Minimum
- Maximum
- Predecessor
- Successor

Minimum and Maximum (1)

- Thanks to the Binary Search Tree property, the minimum (smallest) value in a tree will be the left child of the left child of the left child…

TREE-MINIMUM(x)

1 **while** $x.left \neq$ NIL

2 $\quad x = x.left$

3 **return** x

Minimum and Maximum (2)

- Thanks to symmetry (symmetree?), finding the maximum value in a binary tree is the same, except that we use right children.

TREE-MAXIMUM(x)

1 **while** $x.right \neq$ NIL

2 $\quad x = x.right$

3 **return** x

Those Dynamic Set Operators Again

- ~~Search~~
- Insert
- Delete
- ~~Minimum~~
- ~~Maximum~~
- Predecessor
- Successor

Predecessors and Successors

- Assuming all of the keys in the tree are distinct (no duplicates), then the successor of node `x` is the node `y` such that: `key[y]` is the smallest key > `key[x]`
- This is based on the structure of the tree, rather than the values in the nodes themselves (the `insert` process will build the tree with this property).
  - i.e., we don't have to compare any keys; the tree's <u>*structure*</u> tells us where to look.

Predecessors and Successors

- It should be obvious that the successor of the maximum value in the tree (and predecessor of the minimum value in the tree) is NULL.
  - The largest value in a Binary Search Tree doesn't *<u>have</u>* a right child, because it *<u>is</u>* the right-most child.

Finding the Successor of a Node `x`

- Two cases:
  - If the node HAS a right child, then the successor is the left-most descendant of this right child.
  - If node `x` DOESN'T have a right child, then `x`'s successor is `x`'s most recent (lowest) ancestor whose left child is also an ancestor of `x`.

Finding the Successor of a Node x

- First case:
  - If node x HAS a right child, then x's successor is the left-most descendant of this right child.

if $x.right \neq$ NIL

then return TREE-MINIMUM($x.right$)

Finding the Successor of a Node

- Second case:
  - If node X DOESN'T have a right child, then X's successor is X's most recent (lowest) ancestor whose left child is also an ancestor of X.

$y = x.parent$

while $y \neq \text{NIL}$ and $x == y.right$

  do $x = y$

  $y = y.parent$

return y

Putting it All Together

```
TREE-SUCCESSOR[x]
if x.right ≠ NIL
    then return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x == y.right
    x = y
    y = y.parent
return y
```

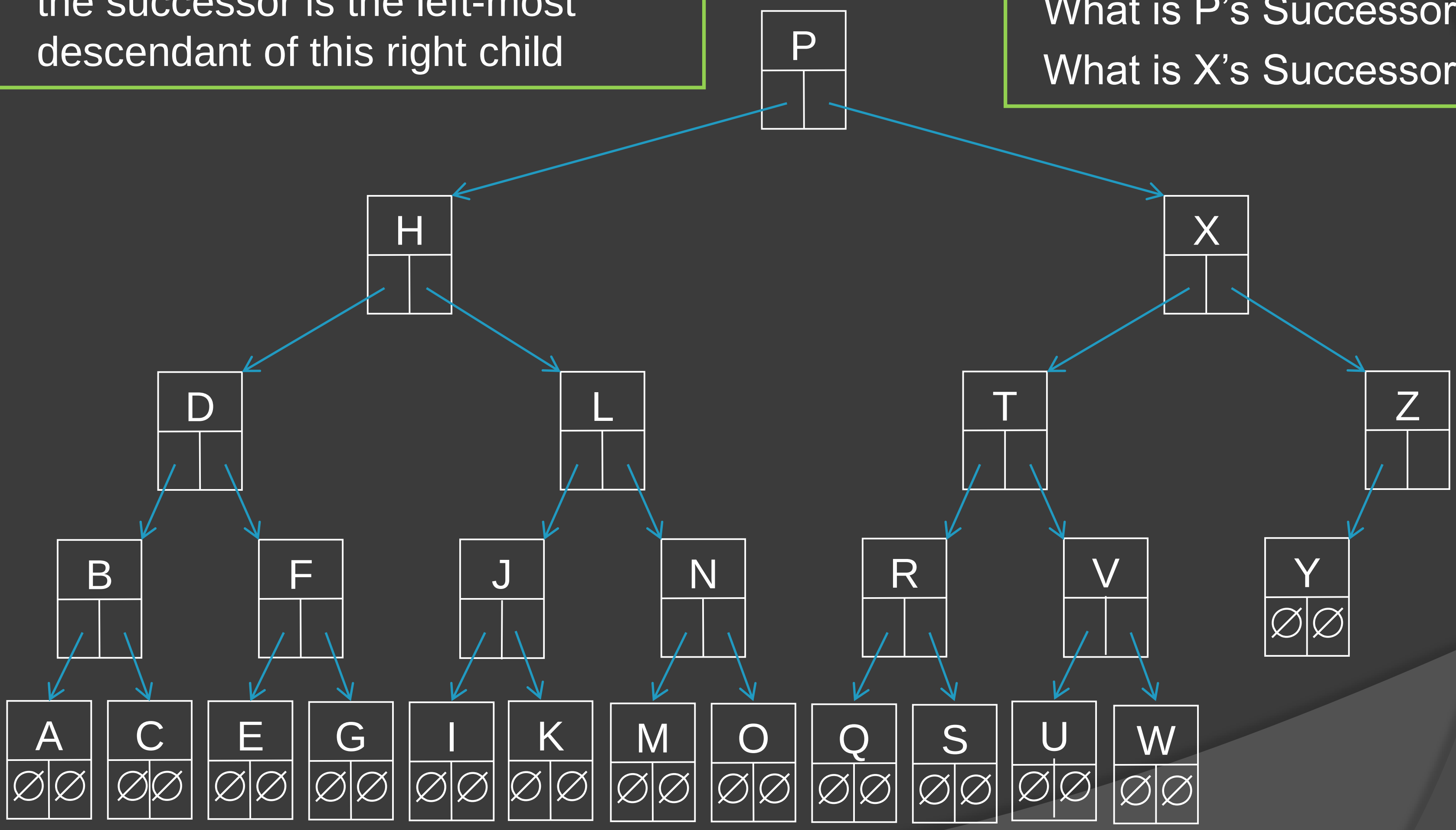
Examples

Let’s take a couple of examples (see board)

Successor Examples

If the node HAS a right child, then the successor is the left-most descendant of this right child

What is D's Successor?
What is P's Successor?
What is X's Successor?

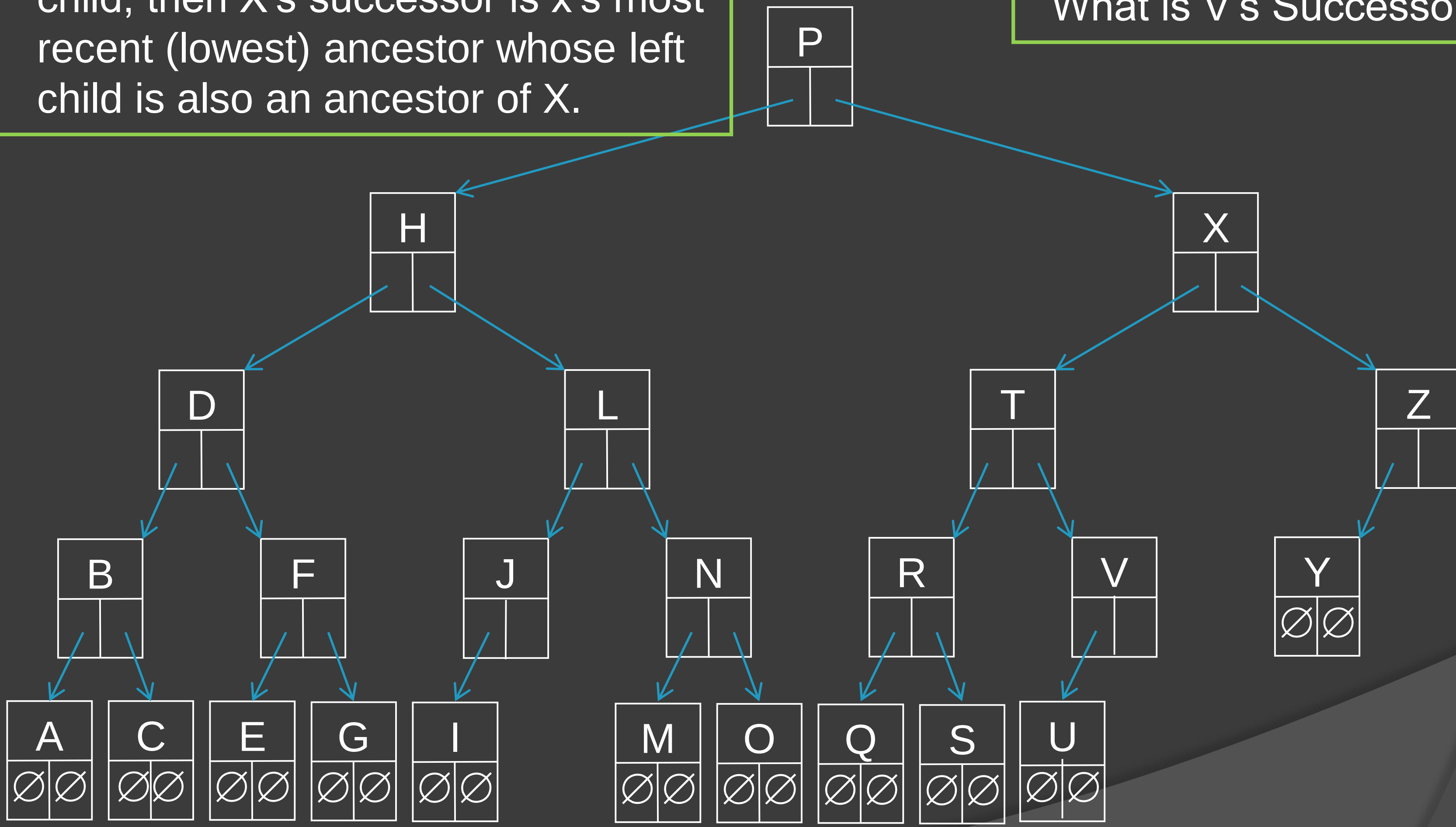


Successor Examples

If node X DOESN'T have a right child, then X's successor is x's most recent (lowest) ancestor whose left child is also an ancestor of X.

What is J's Successor?

What is V's Successor?



What About Predecessors ?

- The PREDECESSOR case is symmetric to the SUCCESSOR case.
  - MINIMUM and MAXIMUM are symmetric, too.
- So, what do we change to come up with the "symmetric" version of the pseudocode?
- We will see other cases related to trees where there are operations whose sub-cases are symmetric.

Those Dynamic Set Operators Again

- ~~Search~~
- Insert
- Delete
- ~~Minimum~~
- ~~Maximum~~
- ~~Predecessor~~
- ~~Successor~~

INSERT and DELETE

- INSERT and DELETE remain.
- INSERT is the simpler of the two.
- When we do an INSERT, we must make sure that the Binary Search Tree property is preserved.

TREE-INSERT: Overview

- Inserts a new value v into Binary Search Tree T, create a new node z to hold v (NULL child pointers).

$$z = \text{new node}$$

$$z.key = v;$$

$$z.left = \text{NIL};$$

$$z.right = \text{NIL};$$

- Now we need to insert z at the right place in the tree and adjust pointers as needed.

TREE-INSERT: Pseudocode

```
TREE-INSERT(T, z)        // insert node z into tree T (as leaf)
y = NIL                  // x traverses tree, looking for insert point
x = T.root               // y lags one step behind x
while x ≠ NIL                    // loop until we hit NULL at leaf
    y = x                        // save current location in y
    if z.key < x.key             // should z be left of x?
            x = x.left           // if so, go left
        else x = x.right         // otherwise, go right
z.p = y                  // x is now NULL, and y points at leaf
if y == NIL              // if y is still NULL, then the tree is empty
    T.root = z           // tree T was empty, so make a 1-node tree
elseif z.key < y.key     // otherwise, z will either be y's new left
        y.left = z       // child, or y's new right child
else  y.right = z        // to keep with the book's notation, add z.p=y
```

TREE-INSERT: Summary (1)

- To insert a value into the tree, we first create a node (z) to hold the new value.
- The new node will become a leaf, so set its child pointers to both be NULL.
- Start at the root, use the Binary Search Tree property to navigate to where z belongs.
- Let x be a pointer that starts at the root, and let y be a pointer that lags (trails) x by one step (i.e., y always points at x's parent).

TREE-INSERT: Summary (2)

- As soon as x becomes NULL, we know we have tried to follow a child pointer at a leaf (which gave us the NULL value).
- x is NULL, and y is the node that contains the NULL child pointer we tried to follow.
- y's NULL child pointer should point to z instead.
- This means that z will either become y's left child, or y's right child.
- y will be z's parent.

Deleting a Node From a Binary Tree

- Before we start:
  - As with many tree-related procedures, not only the steps, but the ***order*** in which we take them, are important.
  - Just as `new` creates a new instance of an item, `delete` destroys one (frees the memory it occupied).
  - Make sure you have "fixed" the pointers before you use `delete` on a node. Once a node is `delete`d, don't count on accessing it, even if we still have a pointer TO it.

Deleting Node z From a Tree

- As with many tree-related procedures, there are different ***cases*** to consider.
  - If the root is a leaf (i.e., the tree contains one node), then node z won't have a parent → empty tree.
  - If node z is a leaf (i.e., has ***no*** children), all we have to do insert a NULL pointer in the node that USED to be to z's parent.
  - If node z has only ***one*** child, we just "splice out" or bypass node z.
  - If node z has ***two*** children, then we splice out its successor y, and replace z's data with y's data (and get rid of y).

Deletion Examples - 1

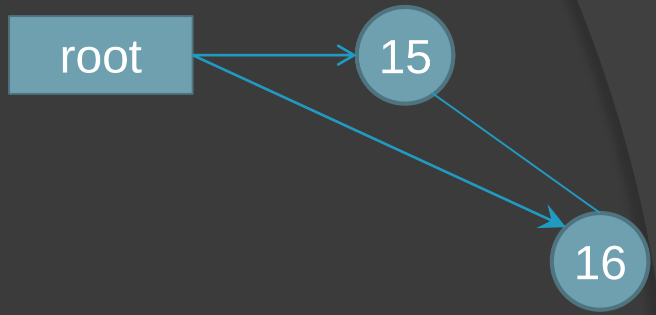
- If the root is a leaf (i.e., the tree contains one node), then node z won't have a parent → empty tree.

- Let's delete 15.

1. Make the root `NULL`.
2. `delete` z.

Deletion Examples – 1a

- If we're deleting the root (z), and it has one child, make the child the new root.
- Let's delete 15.

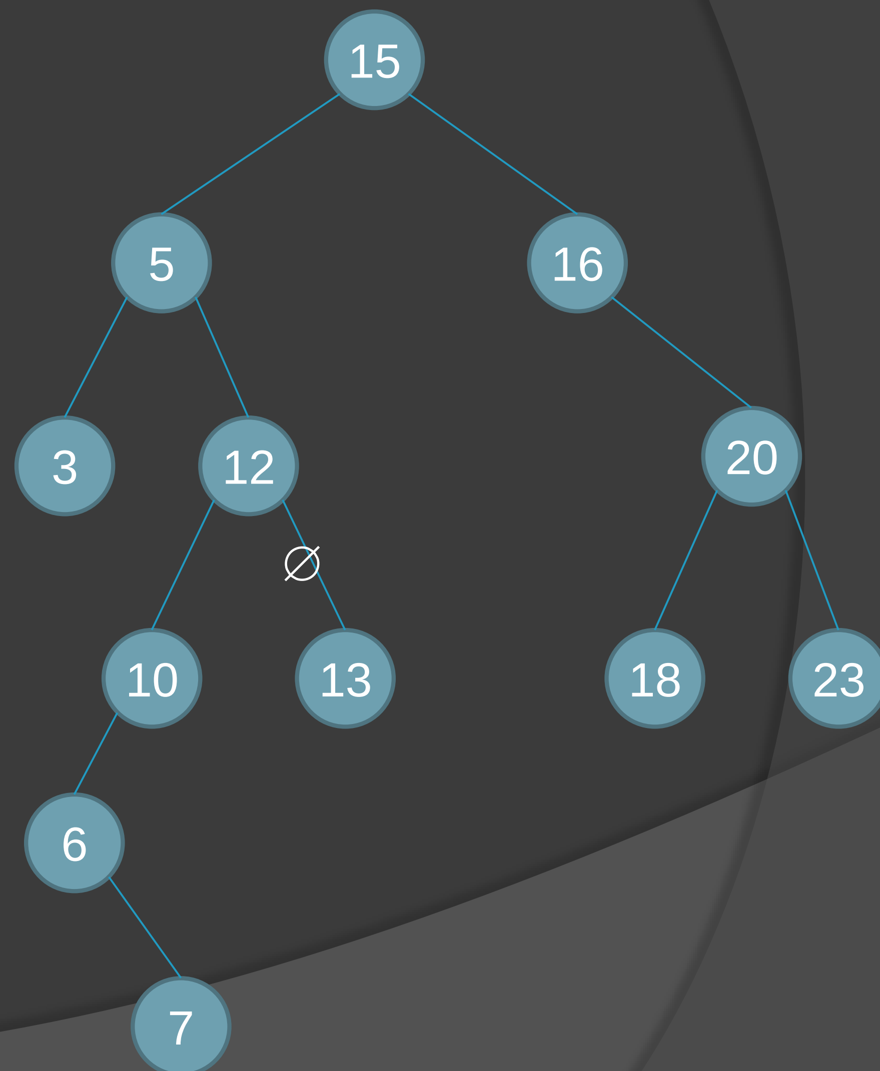


1. Make the root point to the root's (one) child.
2. `delete` z.

Deletion Examples - 2

- If node z is a leaf (i.e., has ***<u>no</u>*** children), all we have to do insert a NULL pointer in the node that USED to be to z's parent.
- Let's delete 13.

1. Make the appropriate child pointer of z's parent `NULL`.
2. `delete` z.

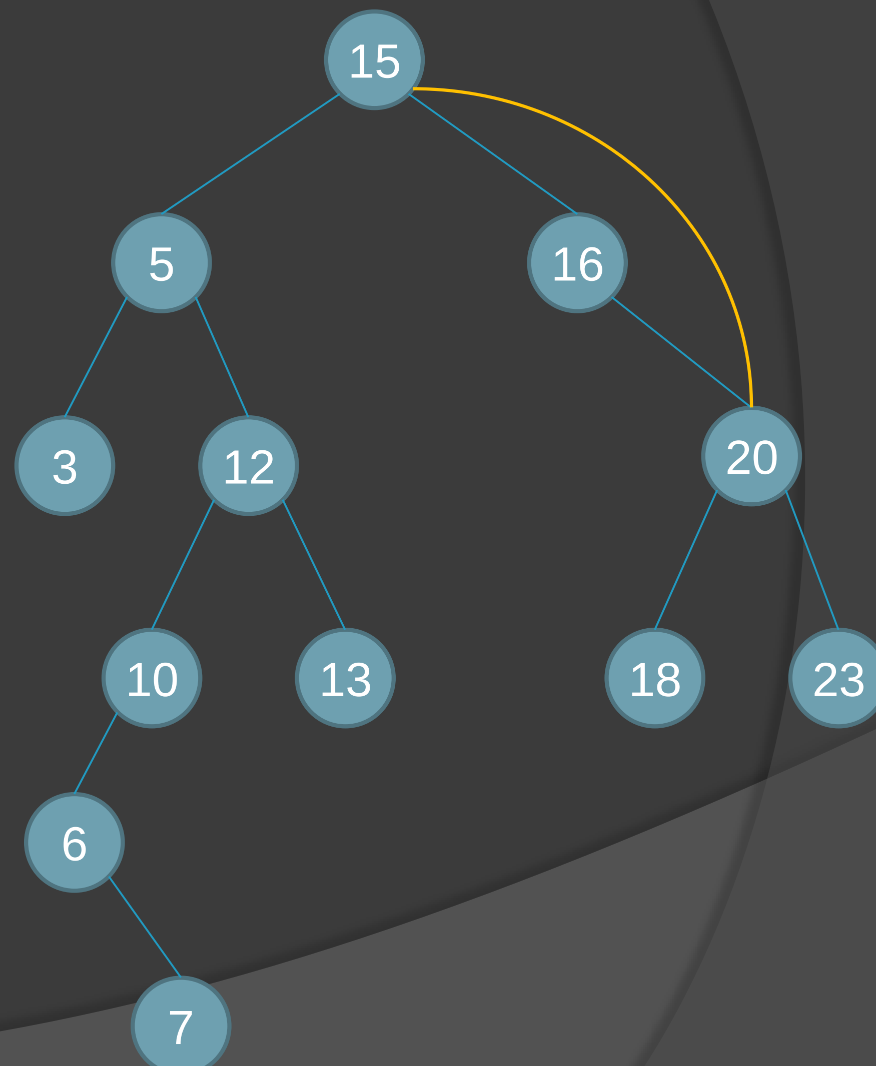


Deletion Examples - 3

- If an ***internal*** node z has only ***one*** child, we just "splice out" or bypass node z.

- Let's delete 16.

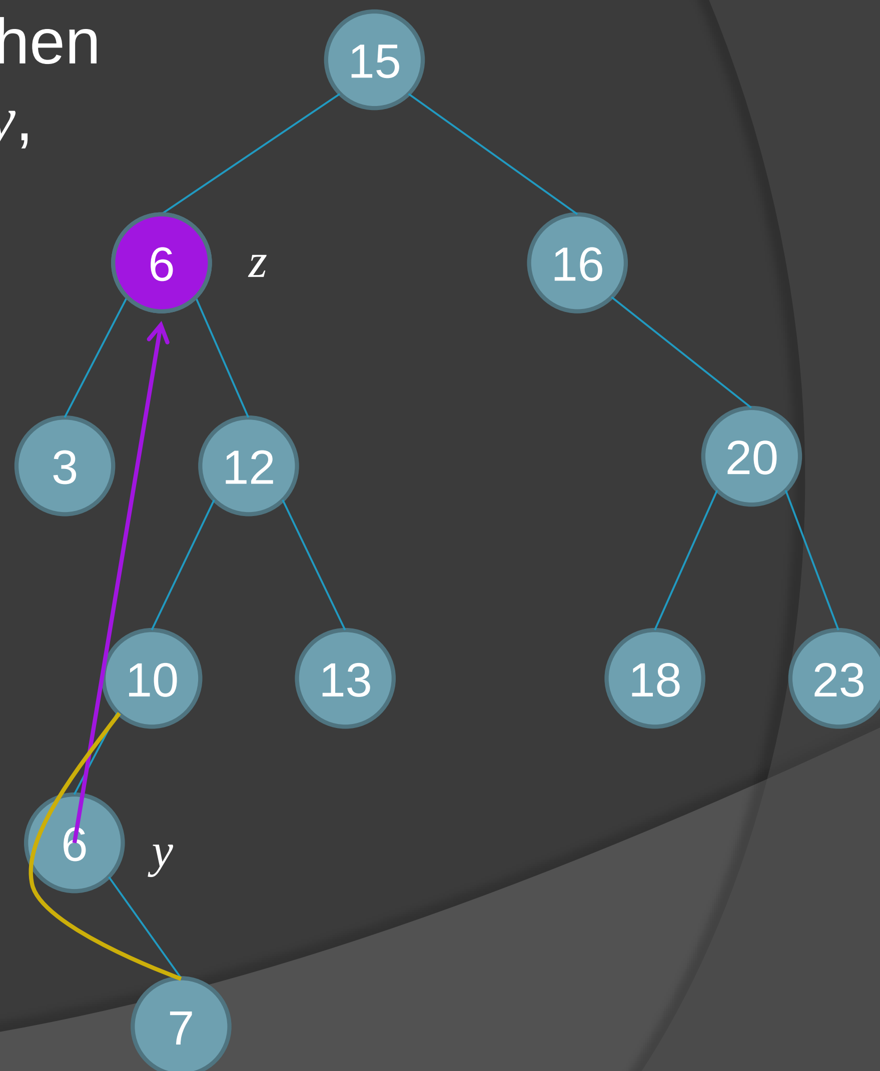
1. Make the appropriate child pointer of z's parent point to z's (only) child.
2. `delete` z.



Deletion Examples – 4

- If node *z* has ***two*** children, then we splice out its successor *y*, and replace *z*'s data with *y*'s data (and get rid of *y*).
- Let's delete 5.

1. Find *z*'s successor, *y* (6) (why do we know *y* can't have two children?).
2. Copy *y*'s data to ***z*.**
3. Splice Out (bypass) *y*.
4. `delete` *y*.



Deleting Node z – Pseudocode

```
TREE-DELETE(T, z)
if z.left == NIL or z.right == NIL
    y = z
    else y = TREE-SUCCESSOR(z)
if y.left ≠ NIL then x = y.left else x = y.right
if x ≠ NIL x.p = y.p
if y.p == NIL
     T.root = x
else if y == y.p.left
            y.p.left   = x
       else y.p.right = x
if y ≠ z then z.key = y.key  // (copy any other data, too)
```

Deleting Node z – Pseudocode (2)

- The pseudocode on the preceding page ***works***, but is pretty unintuitive.
- I suggest that, rather trying to turn this pseudocode into C++, you take the example slides above and turn THOSE into your code.
  - That way, you'll better understand what the code actually ***does***, or at least how it does what it does.
  - You'll write more LOC (Lines Of Code), but that's fine.

End of Chapter 12

Questions?