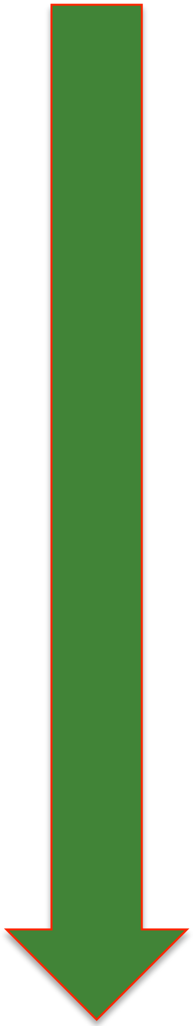
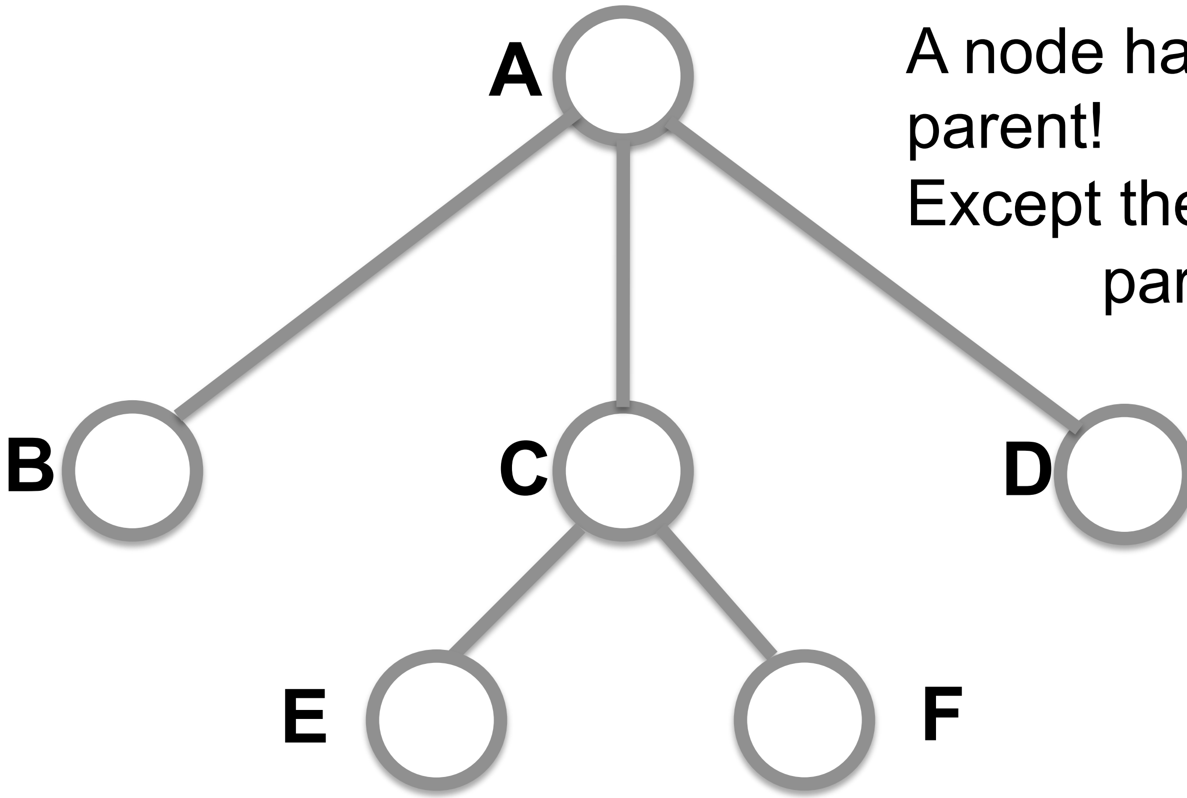
CS200: Trees

Rosen Ch. 11.1 & 11.3

Prichard Ch. 11

Trees

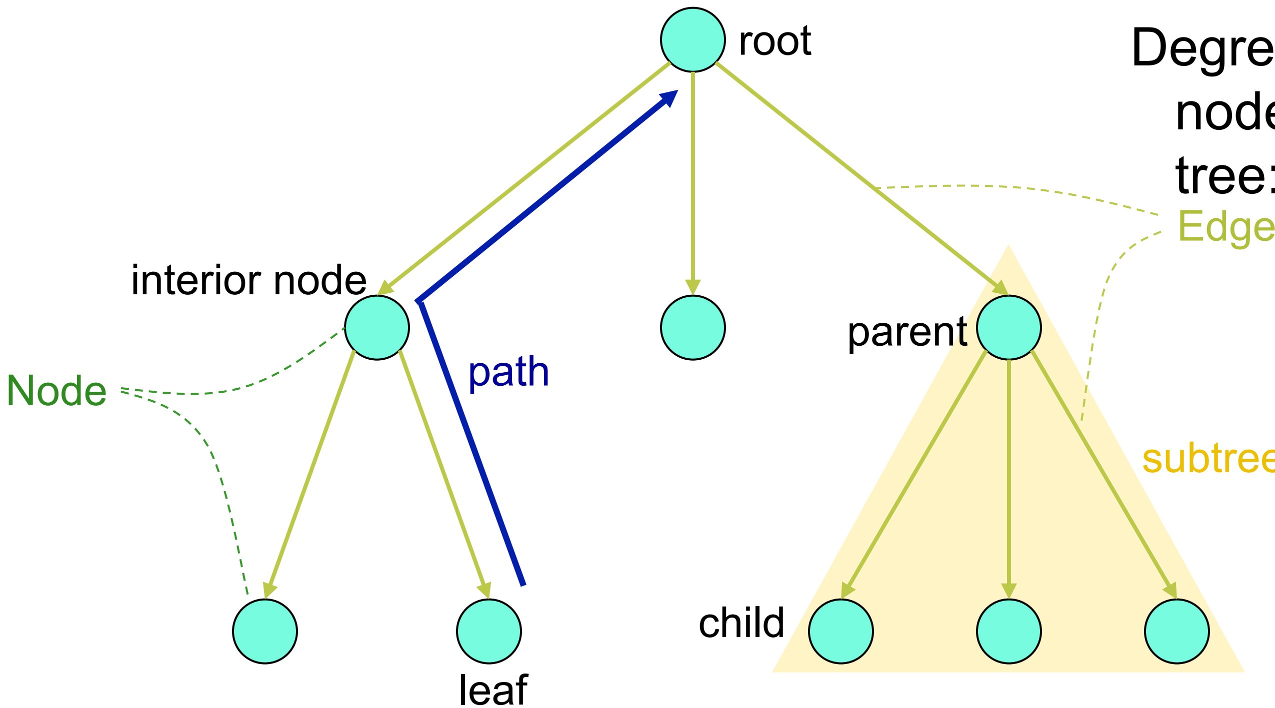




A node has only one parent!
Except the root: zero parents

Tree grows top to bottom!

Tree Terminology



Degree:
- node: # children
- tree: max node degree

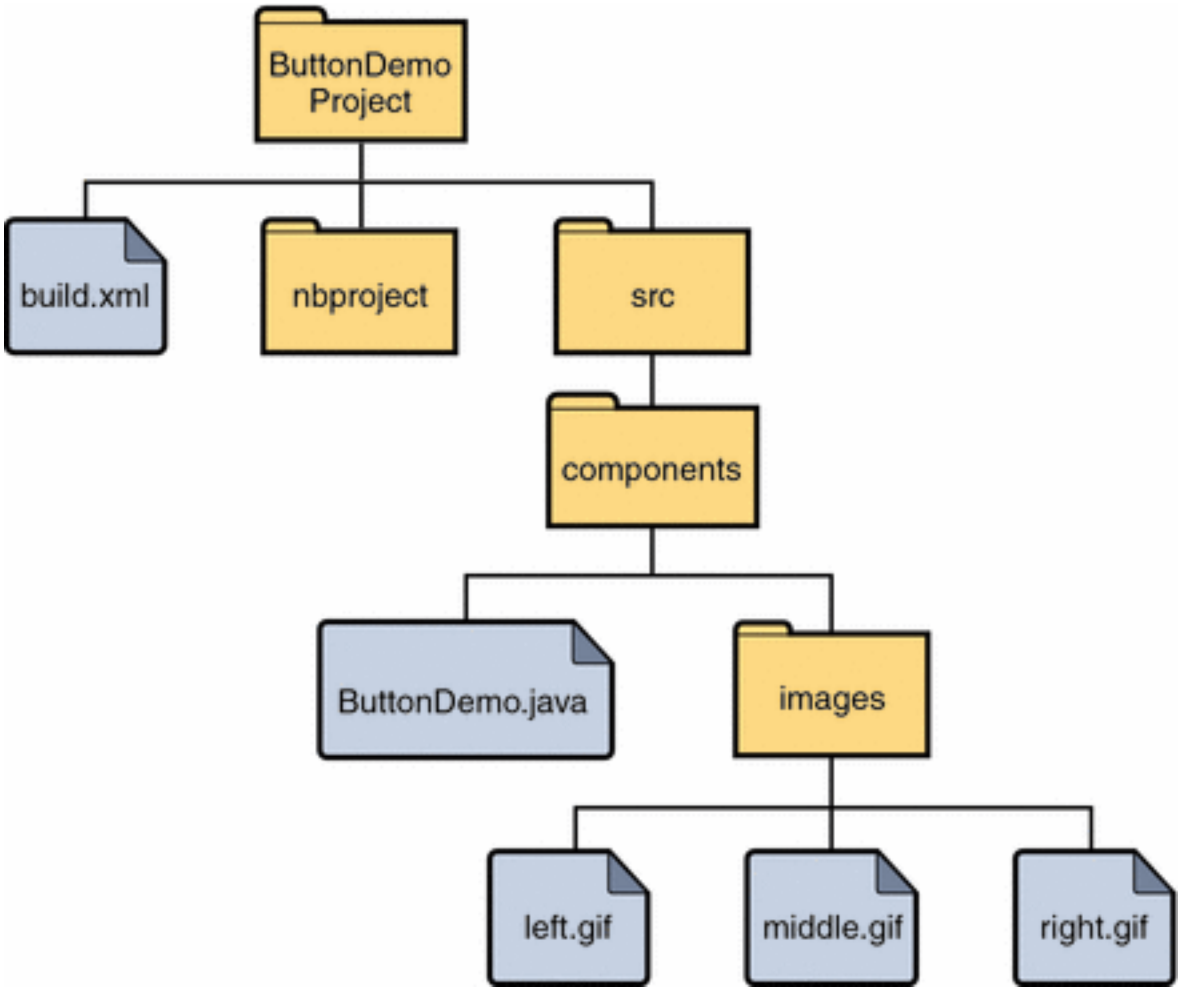
Depth/Level:
- root: 1
- child: level parent + 1

Height: max level

The parent child relationship is generalized to the relationship of ancestor and descendant

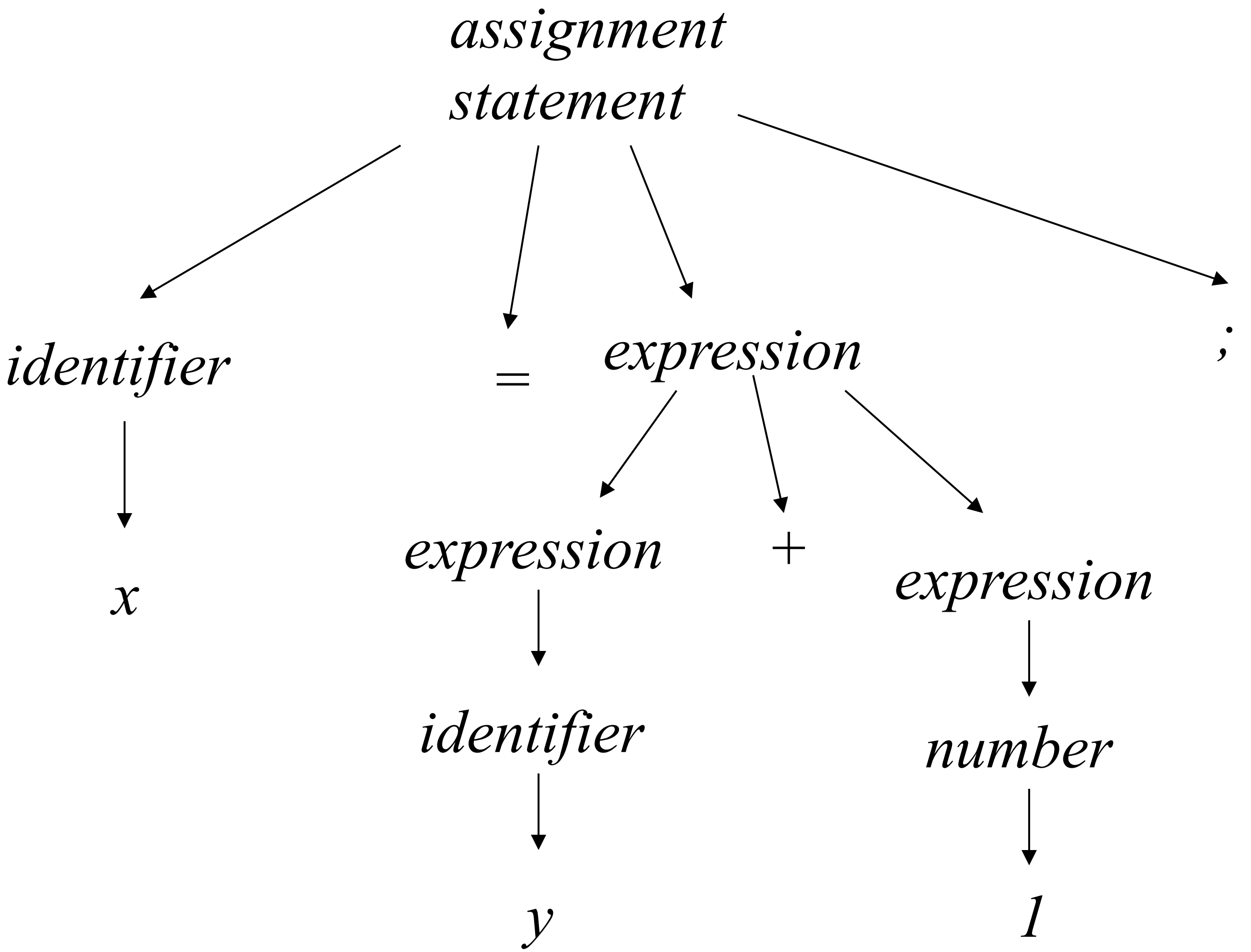
All defs are in Prichard

Applications – File System

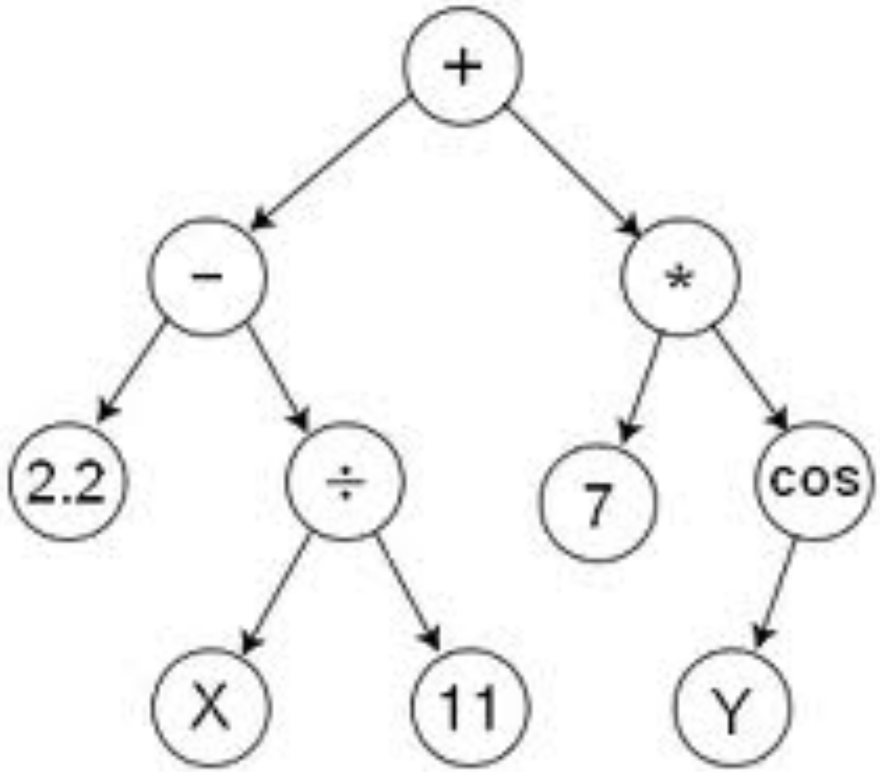


Applications - Parse Trees

Used in compilers to check syntax



Applications – Expression Tree



$$\left(2.2 - \left(\frac{X}{11}\right)\right) + \left(7 * \cos(Y)\right)$$

Predictively parsing expressions

expr = expr “+” term | term

term = term “*” factor | factor

factor = number | “(“ expr “)”

What’s the problem?

For each non-terminal (expr, term, factor) create a method recognizing that non-terminal.

That method implements the alternatives on the RHS of its production.

- When encountering a terminal token**, check whether it is on input, and read passed it (“consume it”).**
- When encountering a non-terminal, **call its method**.

Predictively parsing expressions

expr = expr "+" term | term

term = term "*" factor | factor

factor = number | "(" expr ")"

What's the solution?

For each non-terminal (expr, term, factor) create a method recognizing that non-terminal. That method implements the alternatives on the RHS of its production. When encountering a terminal token, check whether it is on input, and read passed it. When encountering a non-terminal, call its method.

The grammar is left recursive: expr will call expr will call expr etc. without ever reading any tokens

Alternative, iterative grammar for expressions

expr = term (“+” term)*

term = factor (“*” factor)*

factor = number | “(“ expr “)”

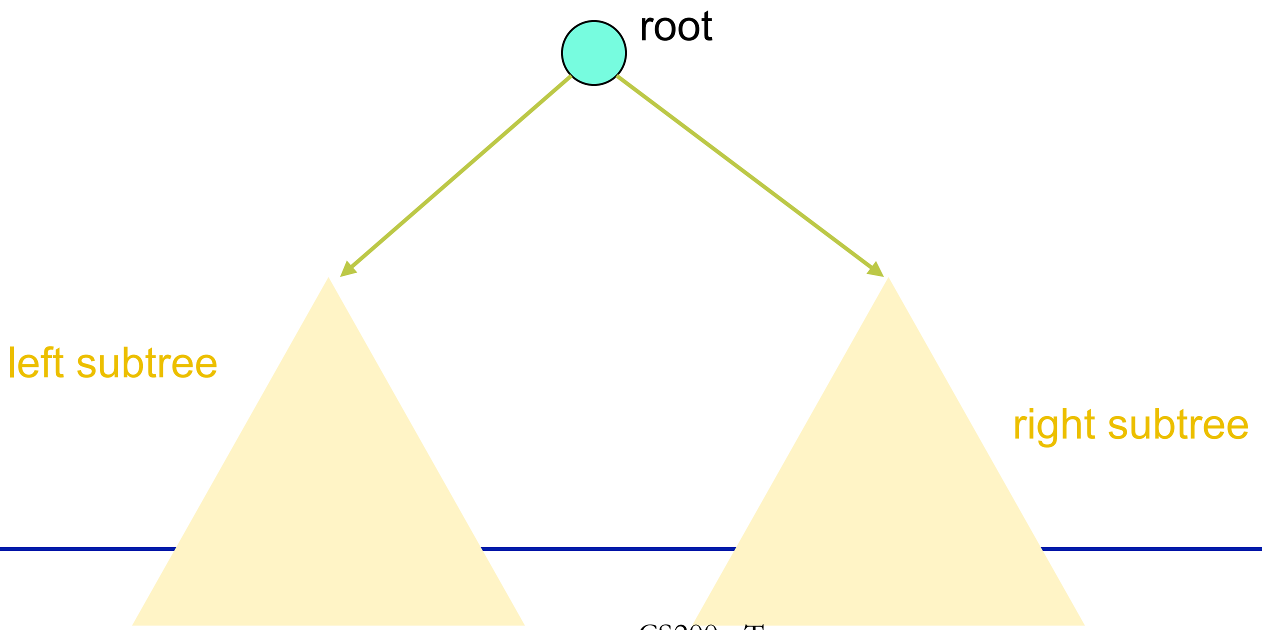
(…)* is implemented with a while loop

Let’s go check out some code:

Parsing infix expressions

Binary Trees

- A binary tree is a set T of nodes such that either
  - T is empty, or
  - T is partitioned into three disjoint subsets:
    - A single node r, the root
    - Two binary trees, the left and right subtrees of r



Tree Terminology

- Level/depth of a node n in a tree T
  - If n is the root of T, it is at level 1
  - If n is not the root of T, its level is 1 greater than the level of its parent

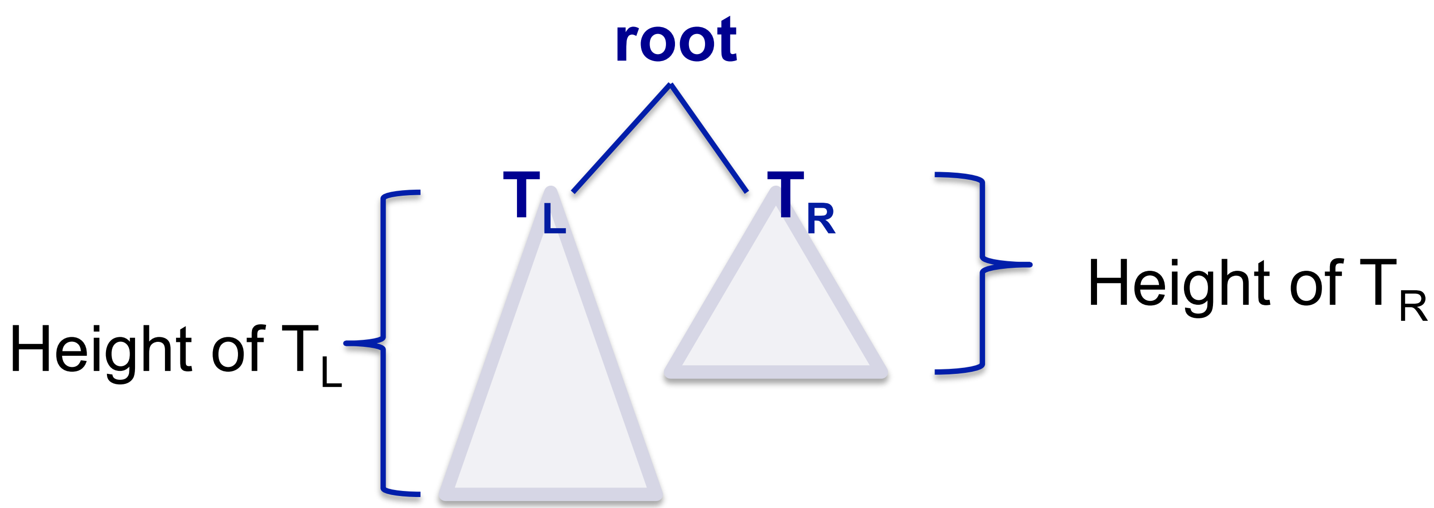
- Height: max level

Starting at level 1 and counting nodes for path length is the Prichard style (Rosen starts at 0)

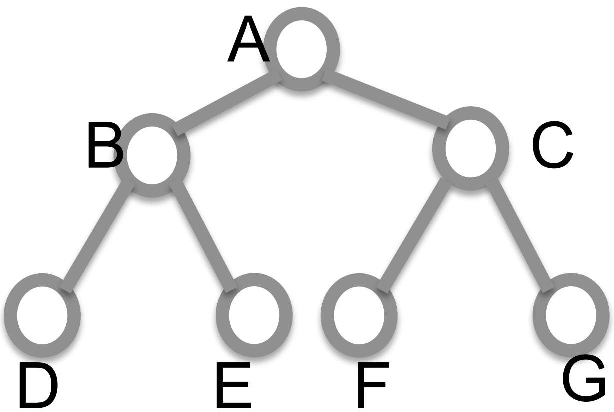
Height of a Binary Tree

- If T is empty, its height is 0.
- If T is a non empty binary tree,

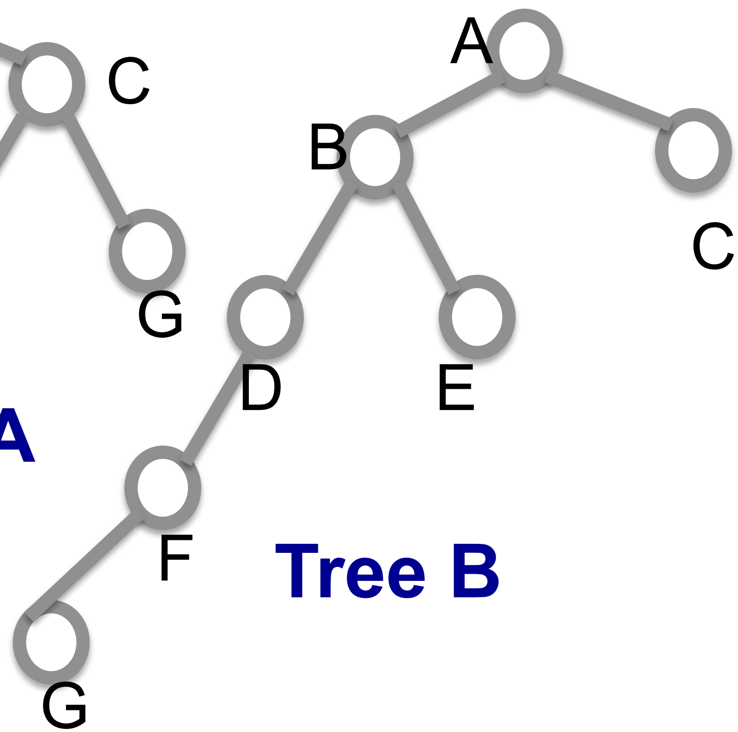
$$height(T) = 1 + max\{height(TL), height(TR)\}$$



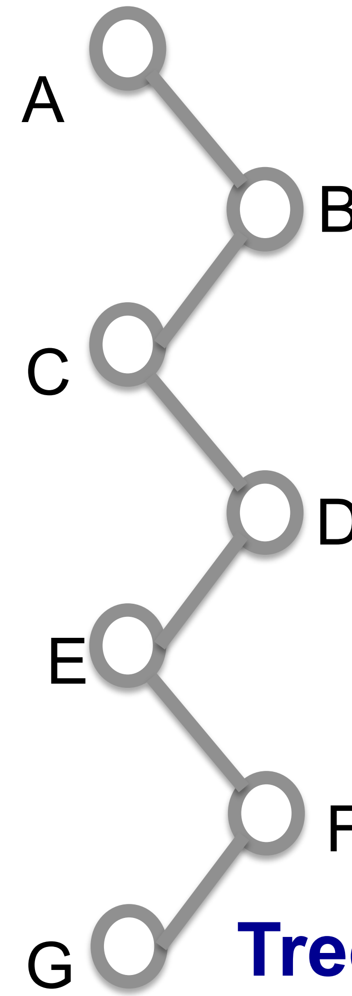
Binary trees with same nodes but different heights



Tree A



Tree B



Tree D

Operations of the Binary Tree

- Create Tree consisting of a Leaf Node
- Create Tree with one or two existing subtrees

- Add and remove node and subtrees
- Retrieve or set the data in the root
- Determine whether the tree is empty

Possible operations

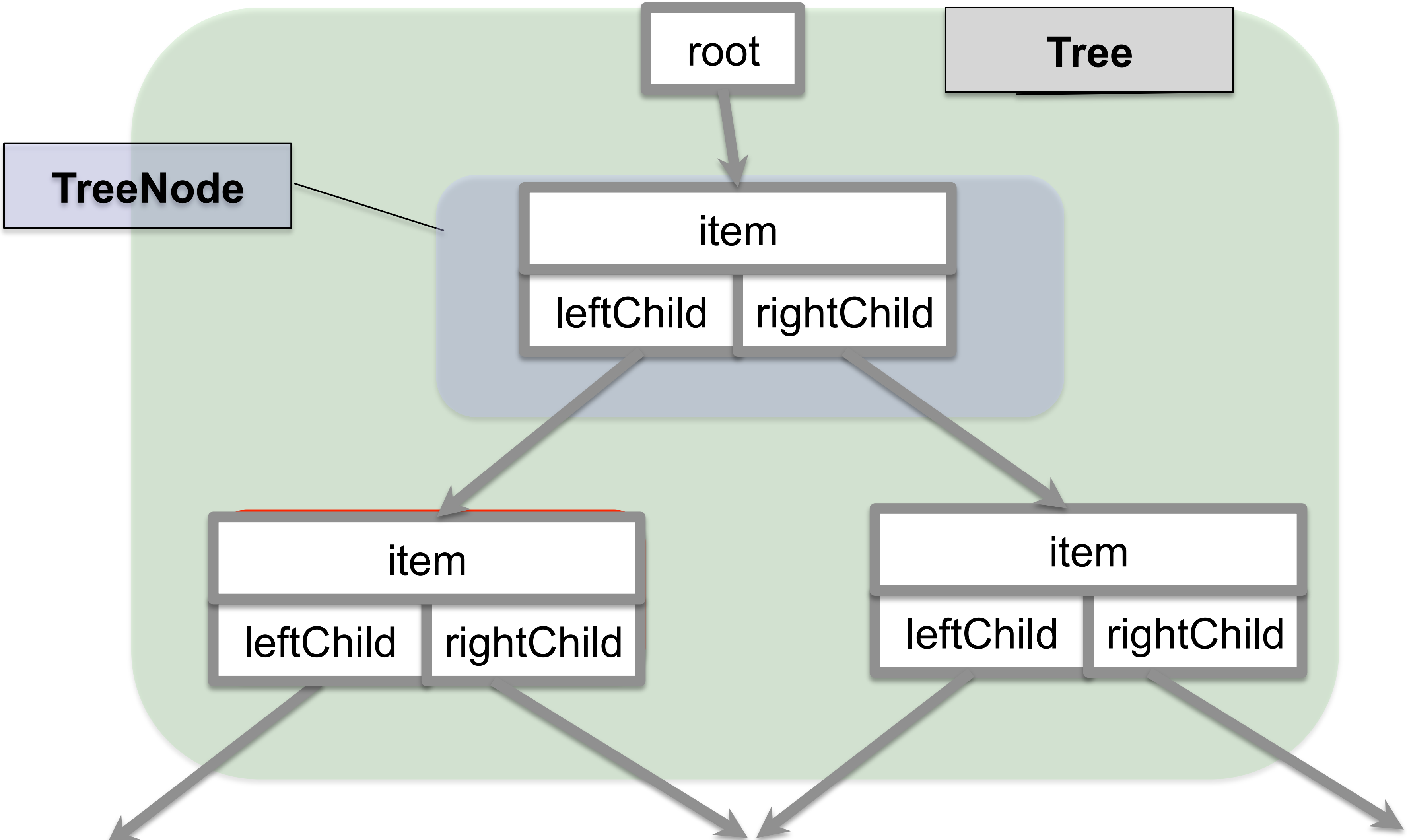
```
Root
Left subtree
Right subtree
```

```
createBinaryTree()
makeEmpty()
isEmpty()
getRootItem()
setRootItem()
attachLeft()
attachRight()
attachLeftSubtree()
attachRightSubtree()
detachLeftSubtree()
detachRightSubtree()
getLeftSubtree()
getRightSubtree()
```

Example

```
// Draw these trees

tree1.setRootItem(“F”)
tree1.attachLeft(“G”)

tree2.setRootItem(“D”)
tree2.attachLeftSubtree(tree1)

tree3.setRootItem(“B”)
tree3.attachLeftSubtree(tree2)
tree3.attachRight(“E”)

tree4.setRootItem(“C”)

binTree.createBinaryTree(“A”, tree3, tree4)
```

A reference-based representation



Tree Node

```
public TreeNode<T> {
     T item;
     TreeNode<T> leftChild;
     TreeNode<T> rightChild;

     public TreeNode(T newItem){
        item = newItem;
      leftChild = null;
      rightChild = null;
     }

     public TreeNode(T newItem, TreeNode<T> left, TreeNode<T>
                         right){
         item = newItem;
         leftChild = left;
         rightChild = right;
     }
}
```

1: Binary Tree Node

Tree

2: Binary Tree

```
// A Binary Tree
public class BinaryTree<T> {
  private TreeNode root;
  // empty tree
   public BinaryTree(){
      this.root = null;
  }
  // rootItem
   public BinaryTree(treeNode node){
     this.root = node;
   }
  . . . .
  // methods that manipulate the whole binary tree
}
```

Building a tree bottom up

- Using a TreeNode constructor:

```
public TreeNode(T item, TreeNode left, TreeNode right){
    this.item = item;
    this.left = left;
    this.right = right;
}
```

```
TreeNode tn1 = new TreeNode("abc");
TreeNode tn2 = new TreeNode("stu");
TreeNode root = new TreeNode("pqr",tn1,tn2);
```

Let’s go check out some more code:

parsing infix expressions and building their expression trees.

Traversal Algorithms

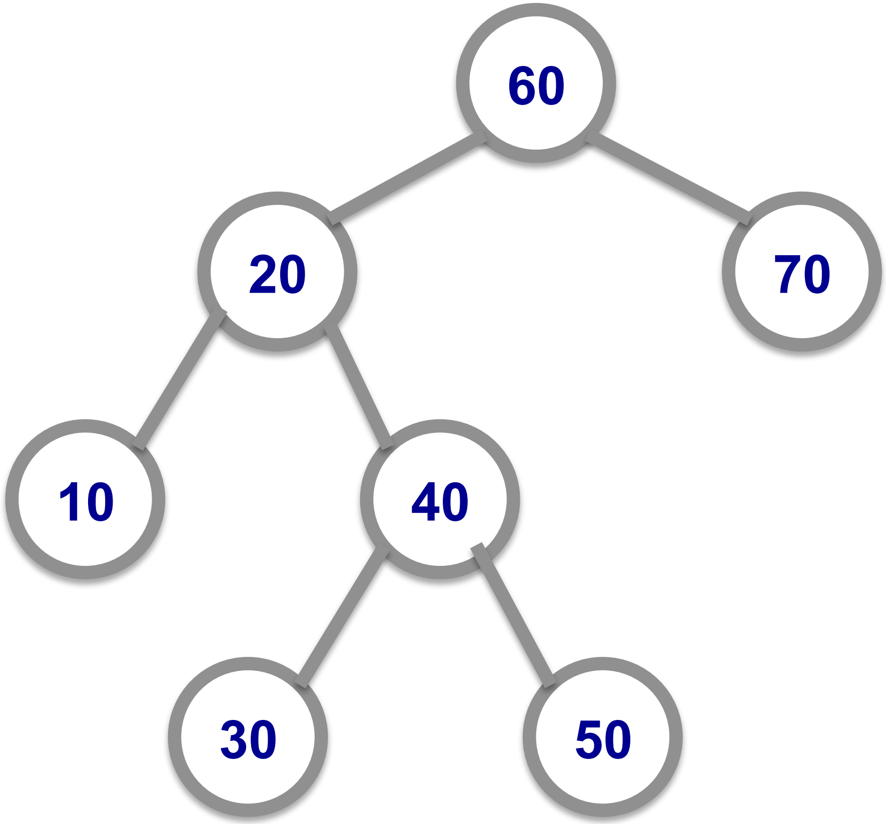
- The traversal of a tree is the process of "visiting" every node of the tree
  - Display a portion of the data in the node.
  - Process the data in the node

- Because a tree is not linear, there are many ways that this can be done.

Breadth-first traversal (BFS)

- Breadth-first processes the tree **level by level** starting at the root and handling all the nodes at a particular level from **left to right**.

Breadth-first traversal



60 – 20 – 70 – 10 – 40 – 30 – 50

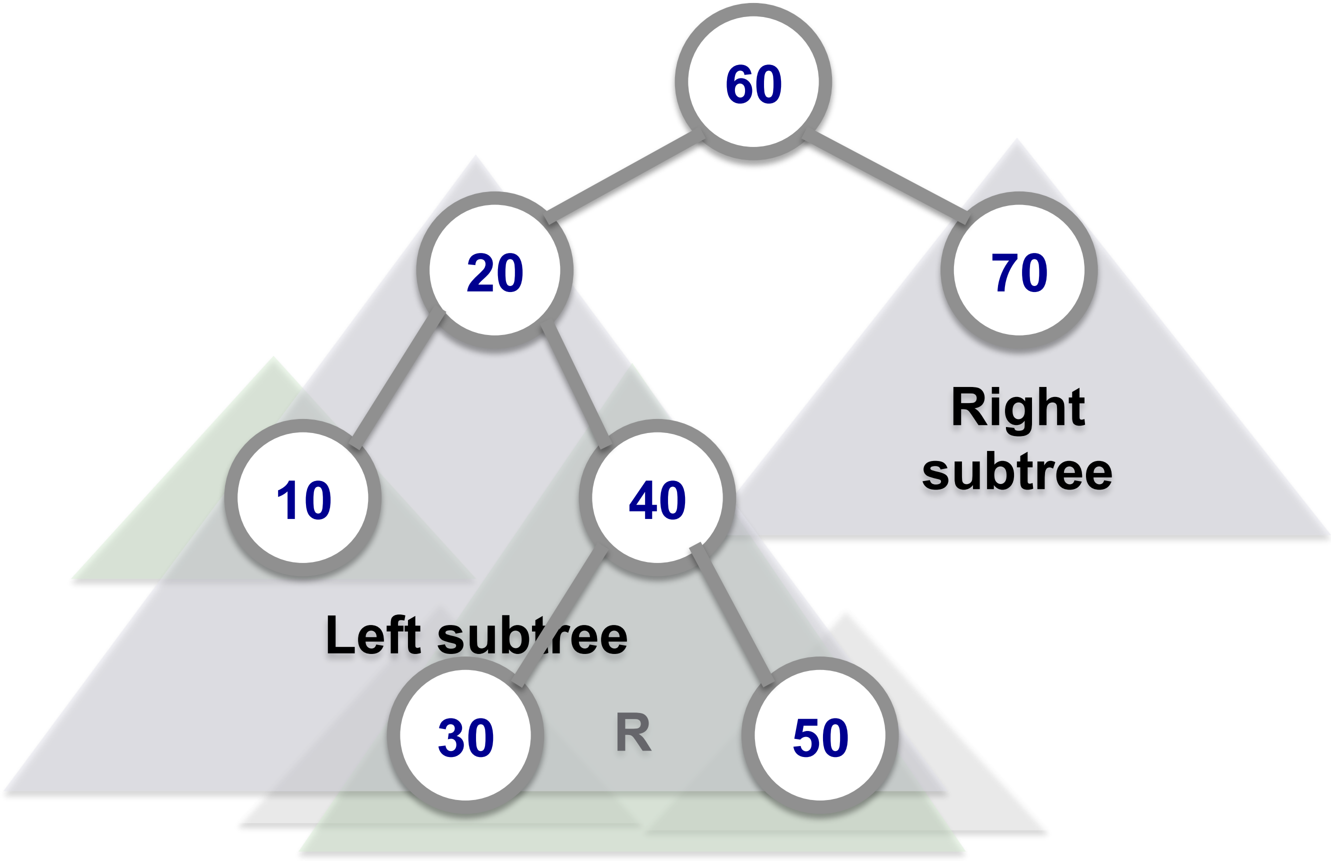
Depth-first traversals (DFS)

- DFS recursively follows the parent-child links
- Three choices of when to visit the root *r*.
  1. **Before** it traverses both of *r*'s subtrees
  2. After it has traversed *r*'s **left** subtree (before it traverses *r*'s right subtree)
  3. After it has traversed **both** of *r*'s subtrees
- visiting = displaying information (e.g. the item)
- **Preorder, inorder,** and **postorder**

Depth First: Preorder traversal

- ***Preorder traversal*** processes the information at the root, followed by the entire left subtree and concluding with the entire right subtree.

Depth First: Preorder traversal

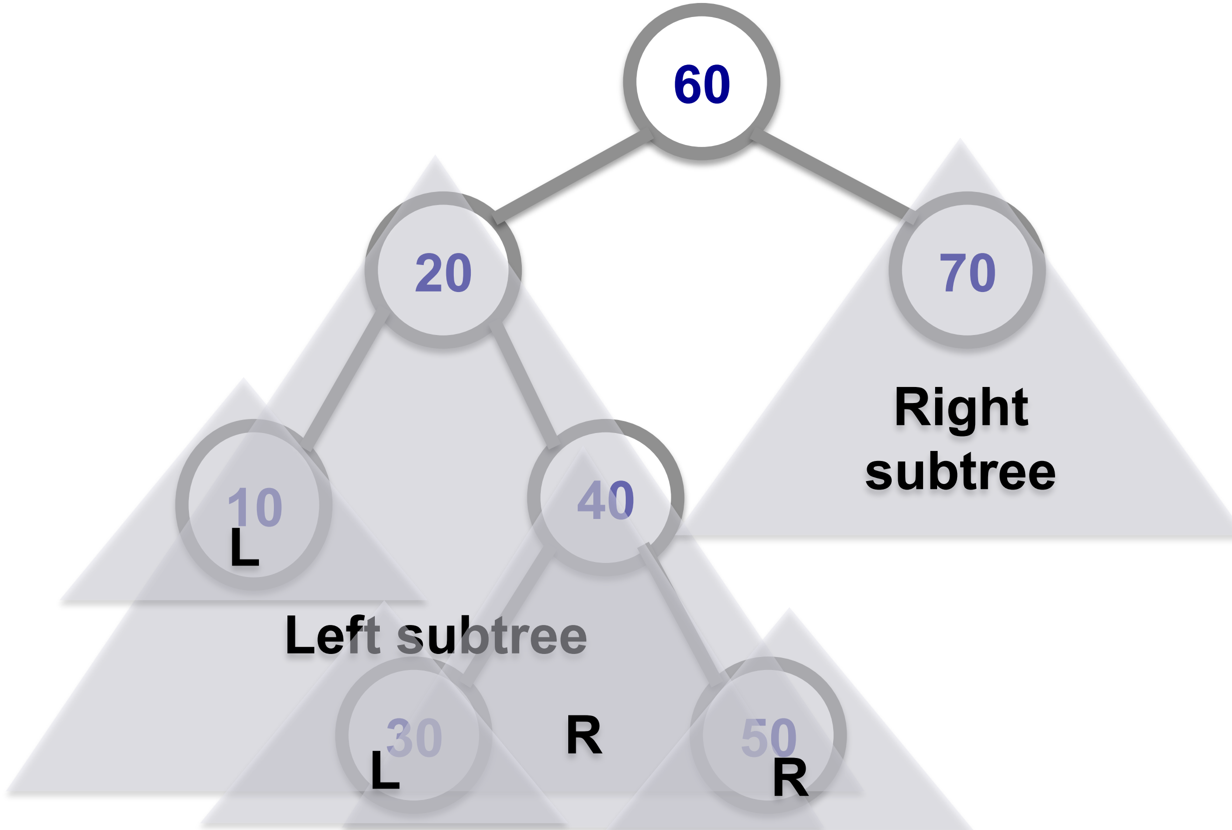


60 – 20 – 10 – 40 – 30 – 50 – 70

Depth First: Inorder traversal

- ***Inorder traversal*** processes all the information in the left subtree before processing the root.
- It finishes by processing all the information in the right subtree.

Depth First: Inorder traversal

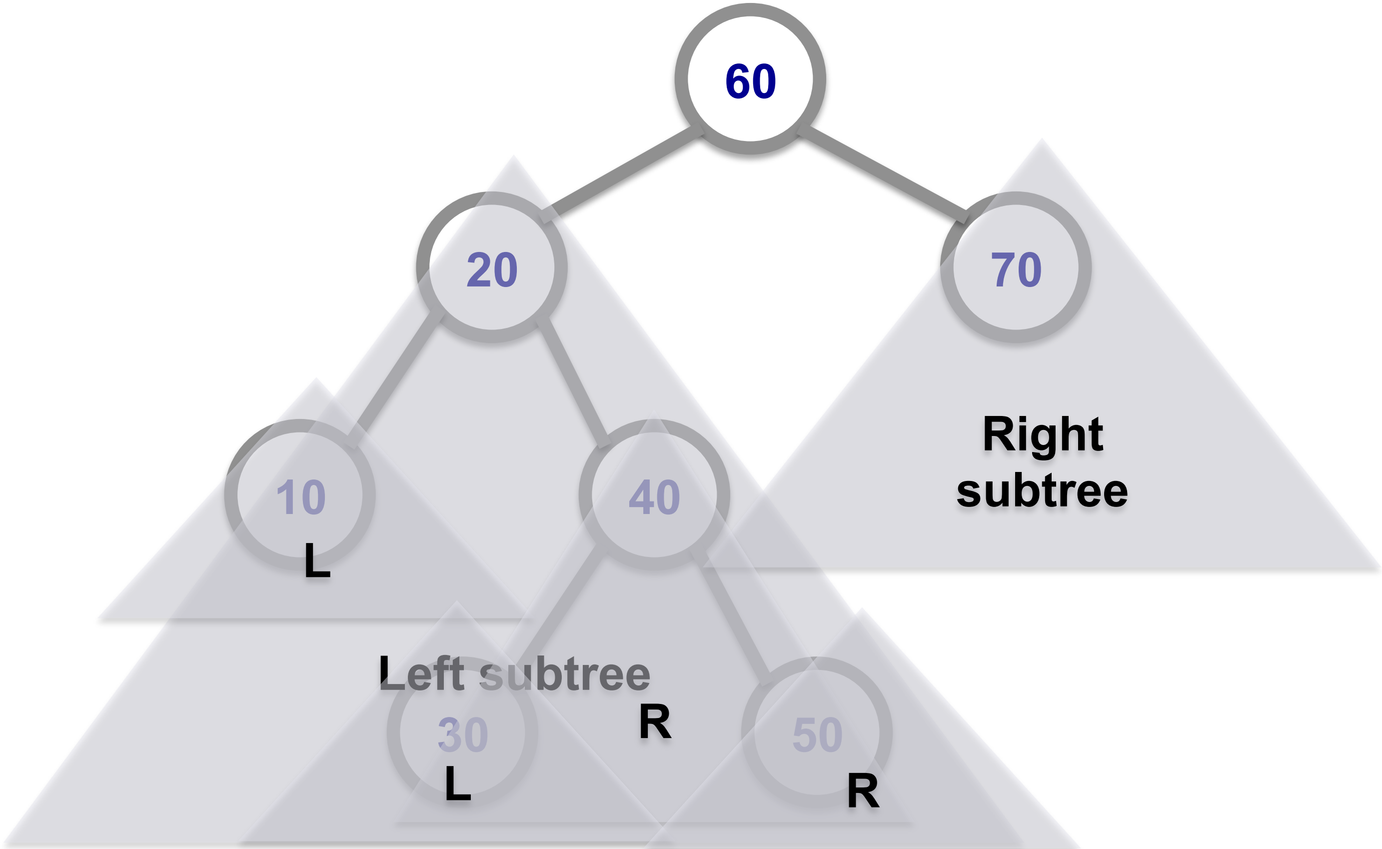


10 – 20 – 30 – 40 – 50 – 60 – 70

Depth First: Postorder traversal

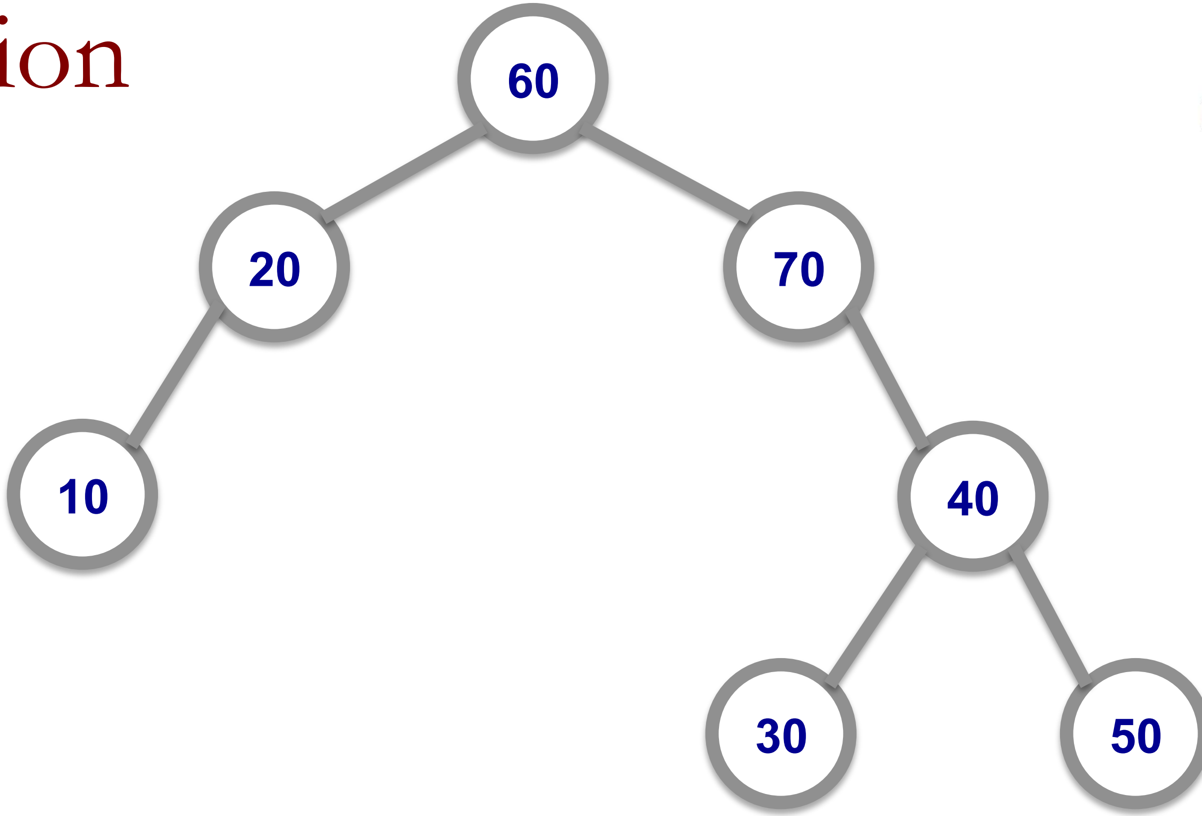
- **Postorder traversal** processes the left subtree, then the right subtree and finishes by processing the root.

Depth First: Postorder traversal



10 – 30 – 50 – 40 – 20 – 70 – 60

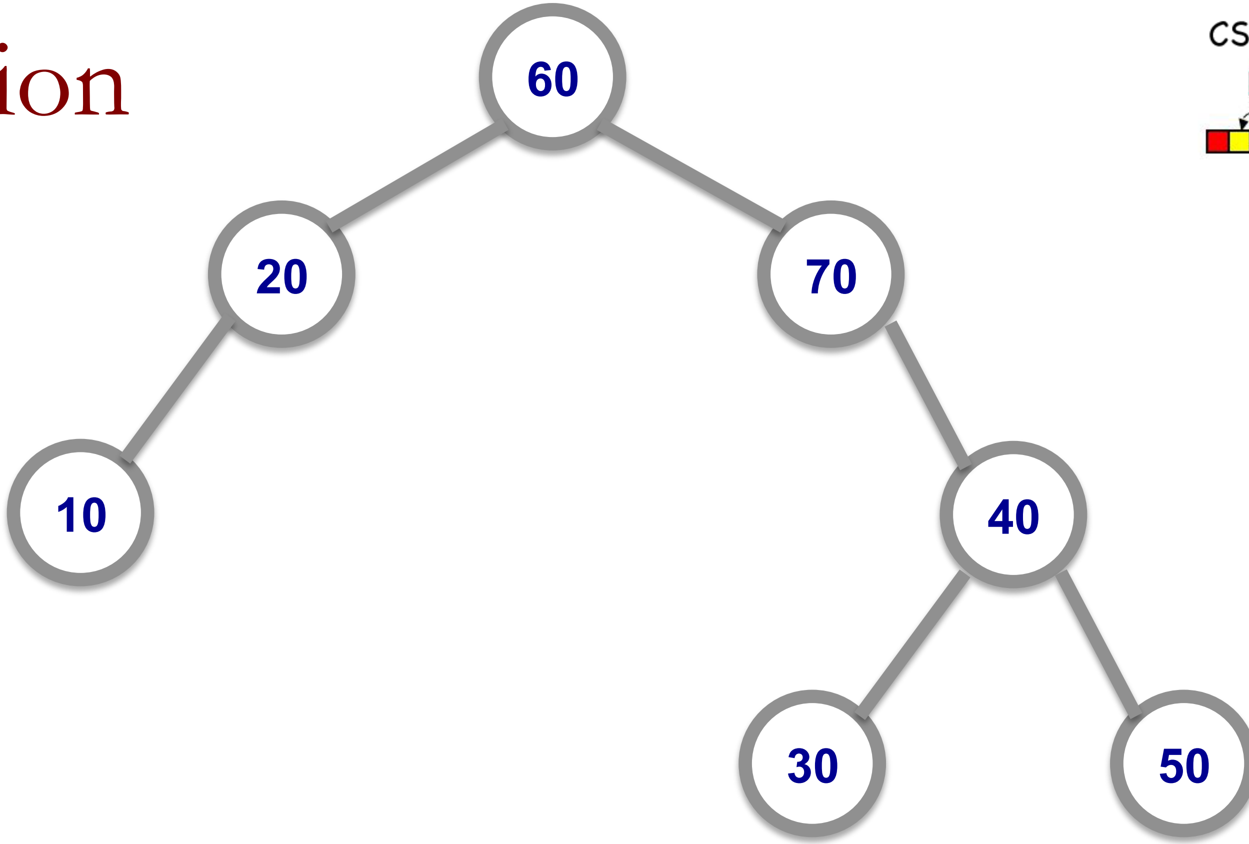
Question



What is the preorder traversal of this tree?

A. 60-20-10-70-40-30-50
B. 10-20-60-70-30-40-50
C. 10-20-30-50-40-70-60

Question



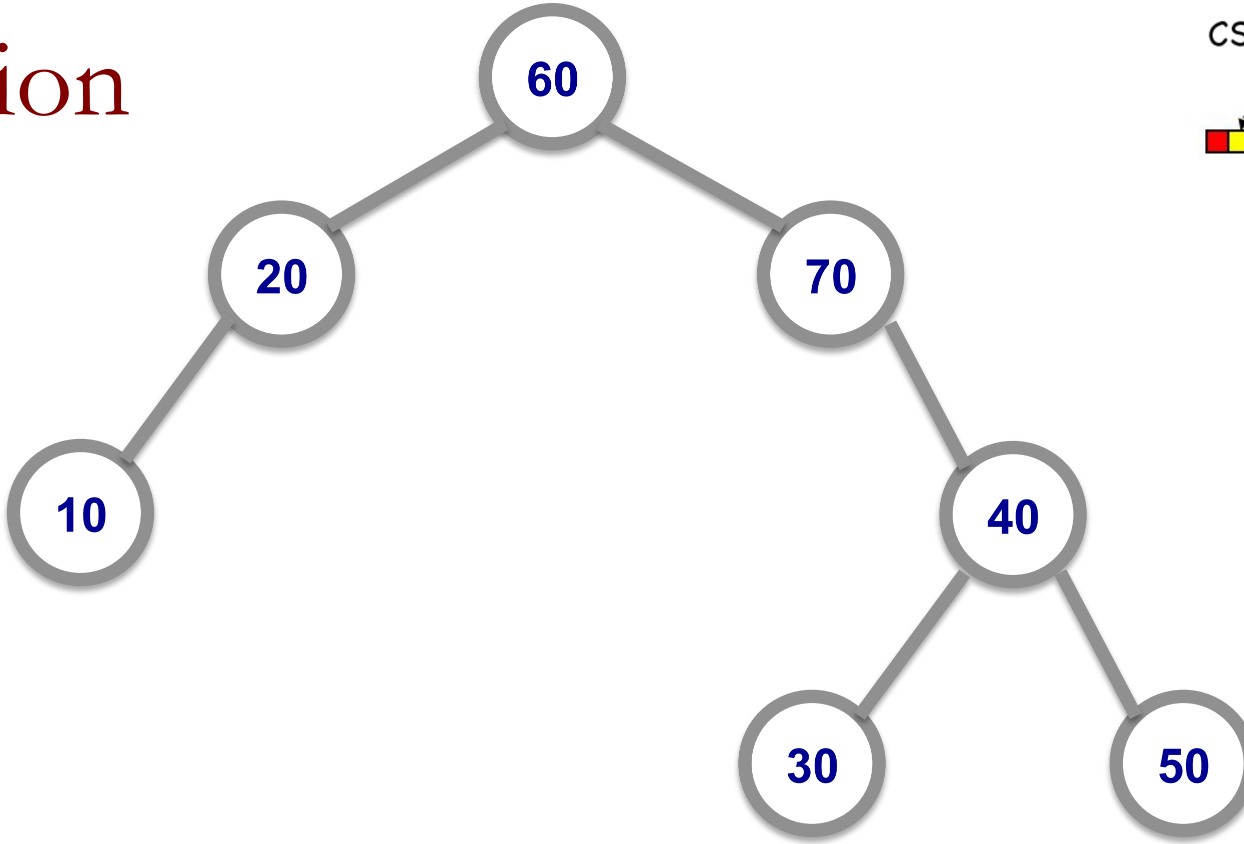
What is the postorder traversal of this tree?

A. 60-20-10-70-40-30-50

B. 10-20-60-70-30-40-50

C. 10-20-30-50-40-70-60

Question



What is the inorder traversal of this tree?

A. 60-20-10-70-40-30-50
B. 10-20-60-70-30-40-50
C. 10-20-30-50-40-70-60

Preorder algorithm

```
public void preorderTraverse(){
        if(debug)
                System.out.println("Pre Order Traversal");
        if (!isEmpty())
                preorderTraverse(root,"");
        else
                System.out.println("root is null");
 }

 public void preorderTraverse(TreeNode node, String indent){
        System.out.println(indent+node.getItem());
        if(node.getLeft()!=null) preorderTraverse(node.getLeft(),indent+" ");
        if(node.getRight()!=null) preorderTraverse(node.getRight(),indent+" ");
}
```

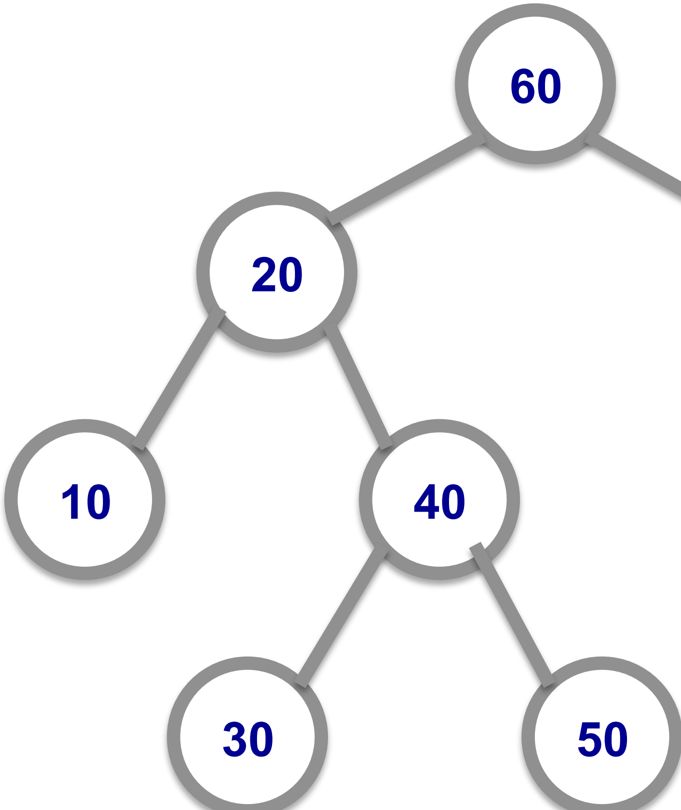
Question

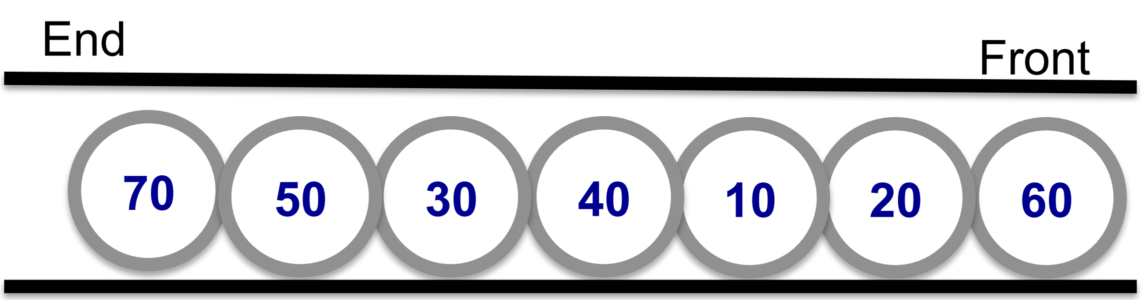
- What does the inorder algorithm look like?
  A. Put “display” at beginning
  B. Put “display” in middle
  C. Put “display” at end

Implementing Traversal with Iterators

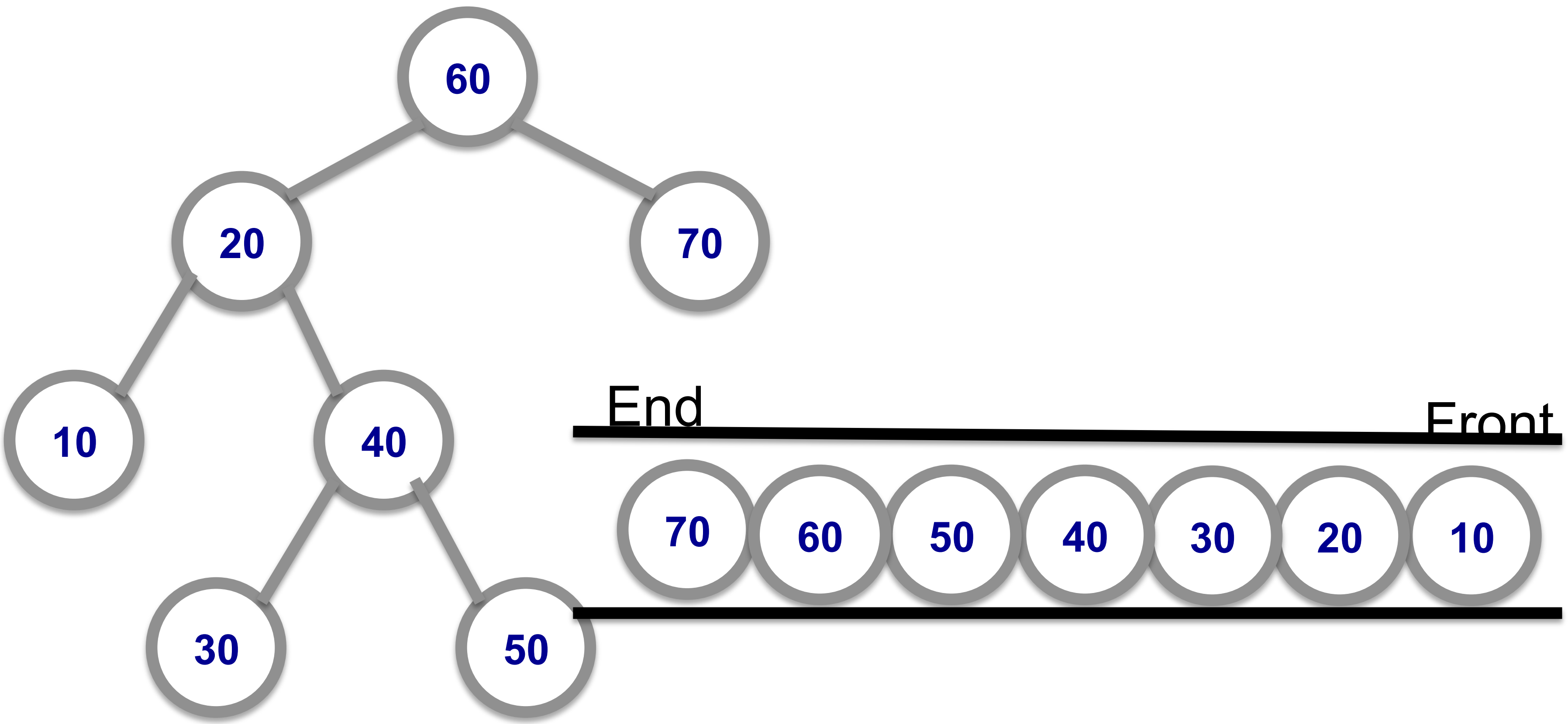
- Use a queue to order the nodes according to the type of traversal.
- Initialize iterator by type (pre, post or in) and enqueue all nodes in order necessary for traversal
- dequeue in next operation

Using TreeIterator for Preorder

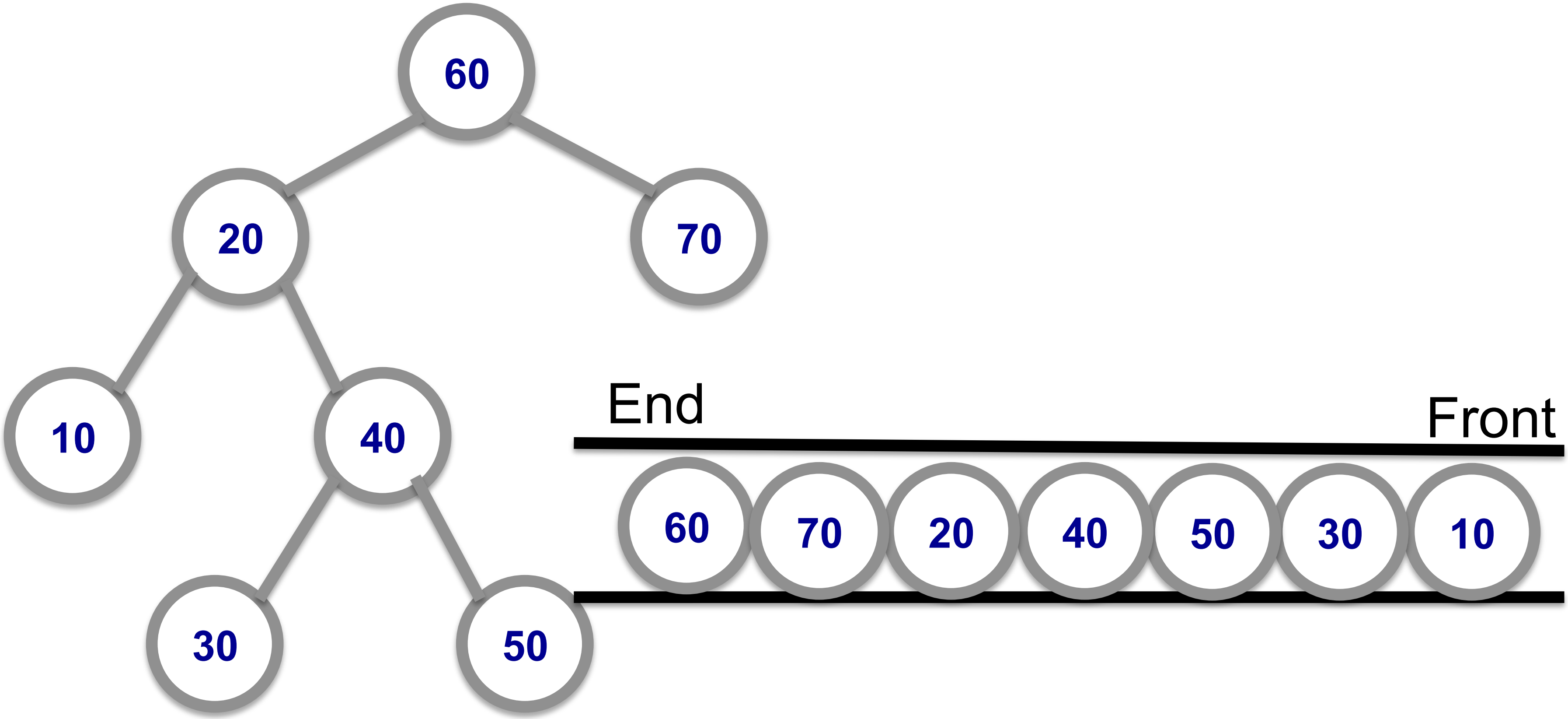




Using TreeIterator for Inorder



Using TreeIterator for Postorder

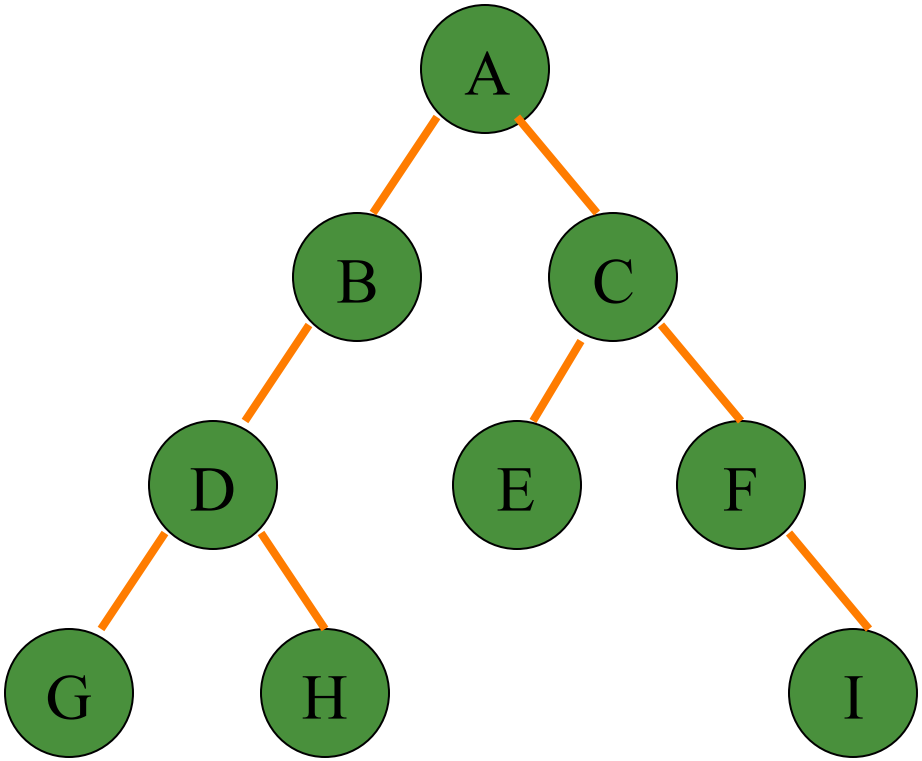


BFS: Level Order Algorithm

- Use a *queue* to track unvisited nodes

- For each node that is dequeued,
  - enqueue each of its children
  - until queue empty

LevelOrder



	Queue	Output
Init	[A]	-
Step 1	[B,C]	A
Step 2	[C,D]	A B
Step 3	[D,E,F]	A B C
Step 4	[E,F,G,H]	A B C D
Step 5	[F,G,H]	A B C D E
Step 6	[G,H,I]	A B C D E F
Step 7	[H,I]	A B C D E F G
Step 8	[I]	A B C D E F G H
Step 9	[ ]	A B C D E F G H I

Categories of Data Structures

- Position-oriented data structures:
  access is by position/index (get(i))
- Value-oriented structures:
  access is by value (get(Value))

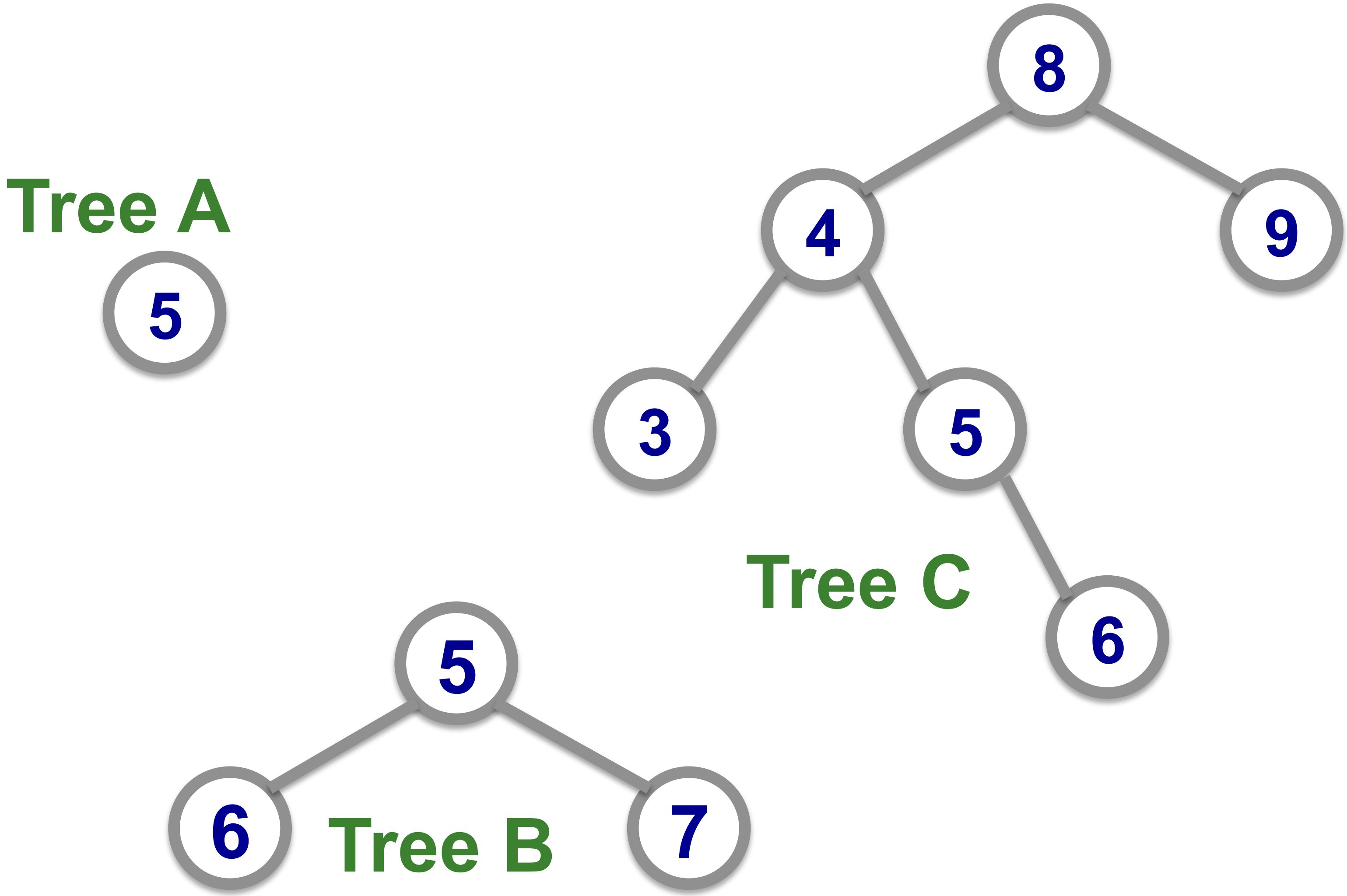
- Whether a data structure is index or value oriented depends often on the way they are used.
- Examples?

Binary Search Trees (BST)

- A binary tree (BST) T is a binary search tree if for every node *n* in T:
  - *n*'s value is greater than all values in its left subtree T_L
  - *n*'s value is less than all values in its right subtree T_R
  - T_R and T_L are binary search trees

- The Items in BST Nodes must be **Comparable**!

Which are BST?



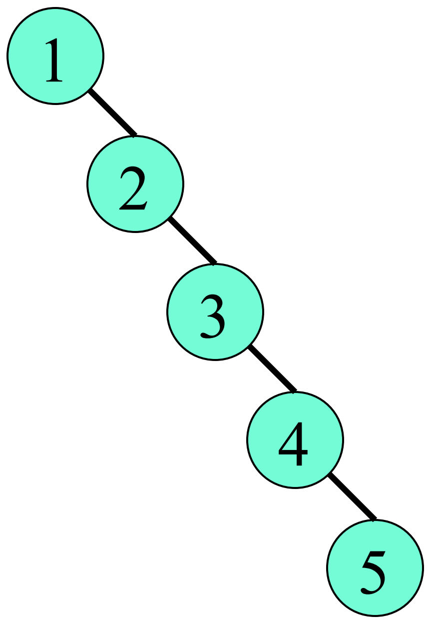
BST

- Organization
  - the sequence of adding and removing influences the shape of the tree
- Search / Retrieval
  - Using *inorder traversal*

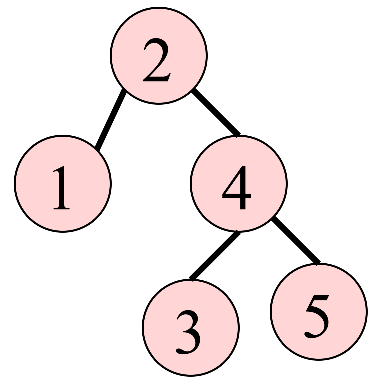
    WHY inorder?

    on the search key

1, 2, 3 ,4 ,5



2, 1, 4, 5, 3



BST Methods

insert(in newIterm:TreeItemType)

- inserts newItem into a BST whose items have distinct search keys that differ from newItem's

delete(in searchKey: KeyType) throws TreeException

- Deletes the item whose search key equals searchKey. If none exists, the operation fails.

retrieve(in searchKey:KeyType):TreeItemType

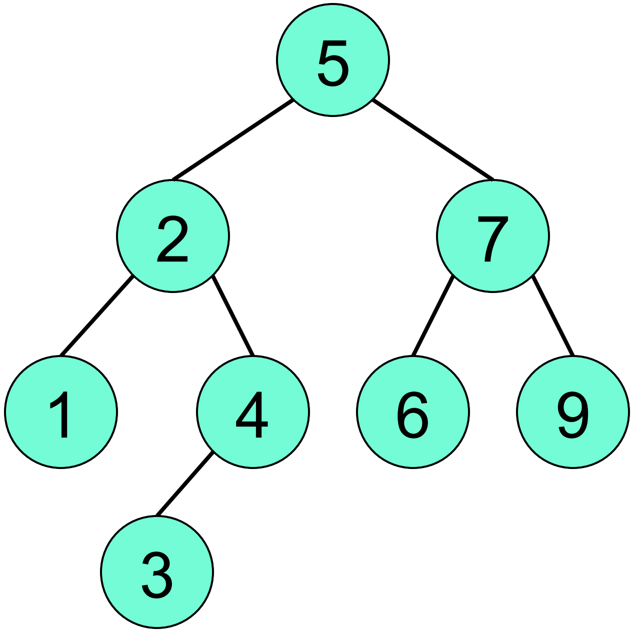
- Returns the item whose search key equals searchKey. Returns null if not found.

In P4 we build a symbol table: a search tree of BST nodes.

BST - Search

compare value with node

- null: not found
- == : found
- <   : search in the left sub-tree
- >   : search in the right sub-tree



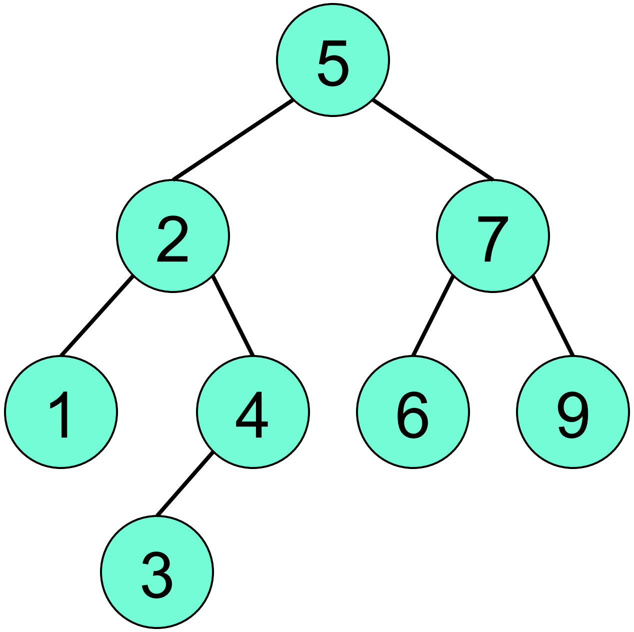
Locate 4 in the BST !

Insert: question

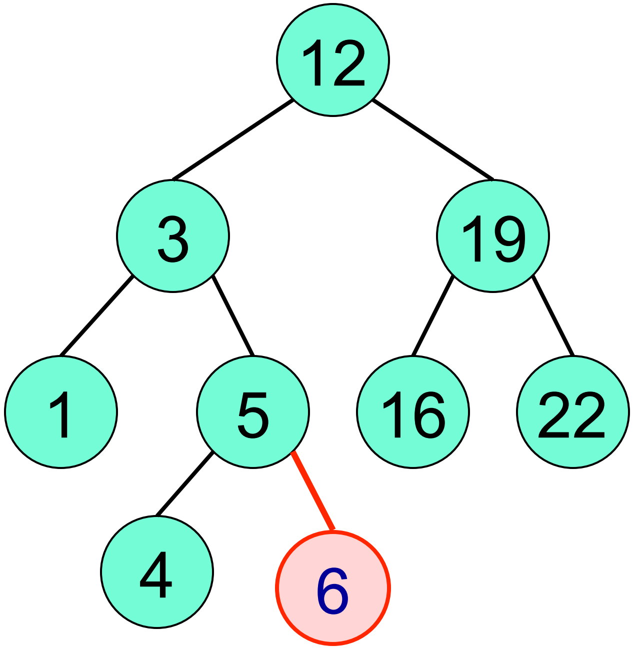
Where will "8" be added?

Where the search would have looked for it:

Left child of 9



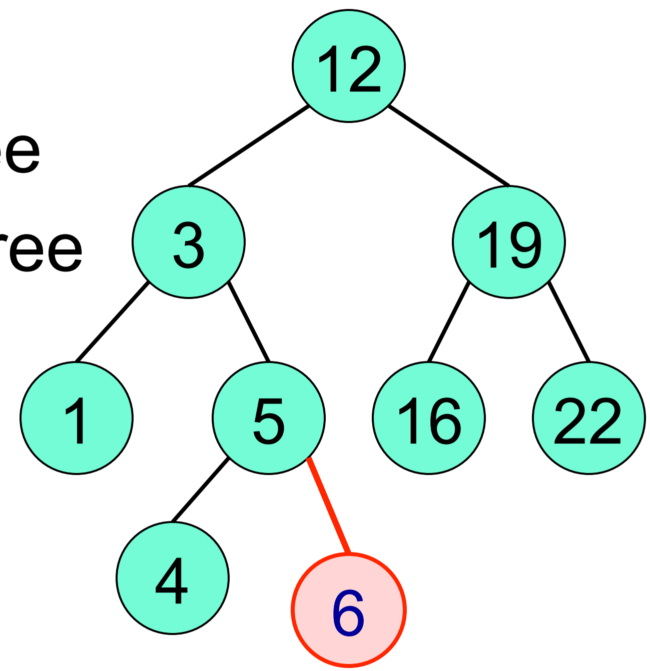
BST – Insert 6



Add 6

BST – Insert

- Always add as a leaf – in the position where the search method would look for it
- Find leaf location
  - < root : add to the left sub-tree
  - > root : add to the right sub-tree
- Special Cases:
  - already there
  - empty tree



Inserting an item

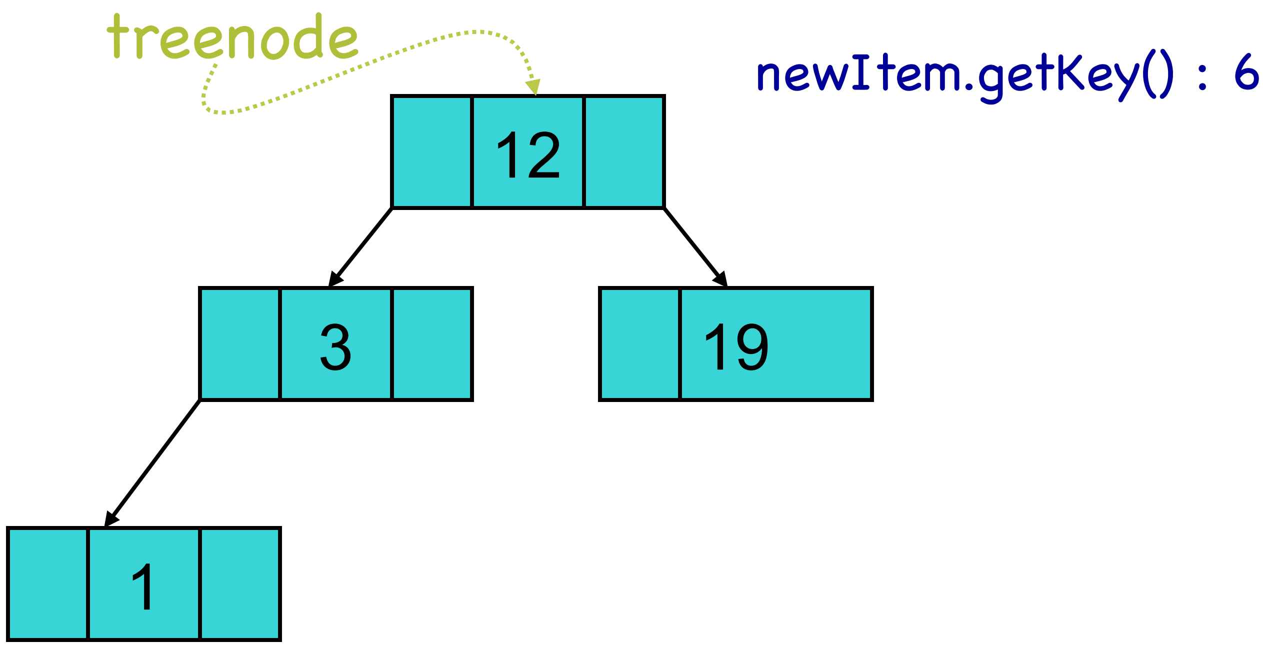
```
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
    // Inserts newItem into the binary search tree of which
    //treeNode is the root

    Let parentNode be the parent of the empty subtree at which
    search terminates when it seeks newItem's search key

    if (search terminated at parentNode's left subtree) {
        set leftChild of parentNode to reference newItem
    }
    else {
        set rightChild of parentNode to reference newItem
    }
```

Inserting an item

```
insertItem(in treeNode:TreeNode, in newItem:TreeItemType)
    // Inserts newItem into the binary search tree of which
    // treeNode is the root
    if (treeNode is null) {
            create new node with newItem as data
            return new node }
    else if (newItem.getKey() < treeNode.getItem().getKey()) {
            treeNode.setLeft(insertItem(treeNode.getLeft(), newItem))
          return treeNode}
    else {
            treeNode.setRight(insertItem(treeNode.getRight(),newItem))
            return treeNode }
```

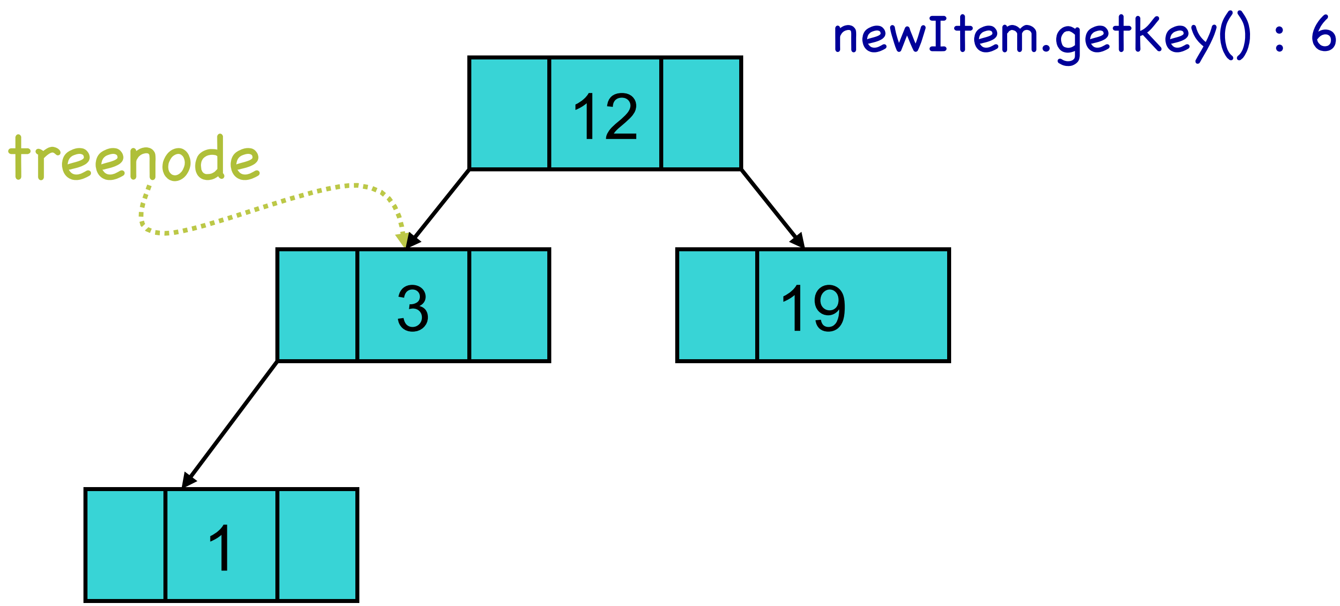
Let's go check out some code

BST – Insert



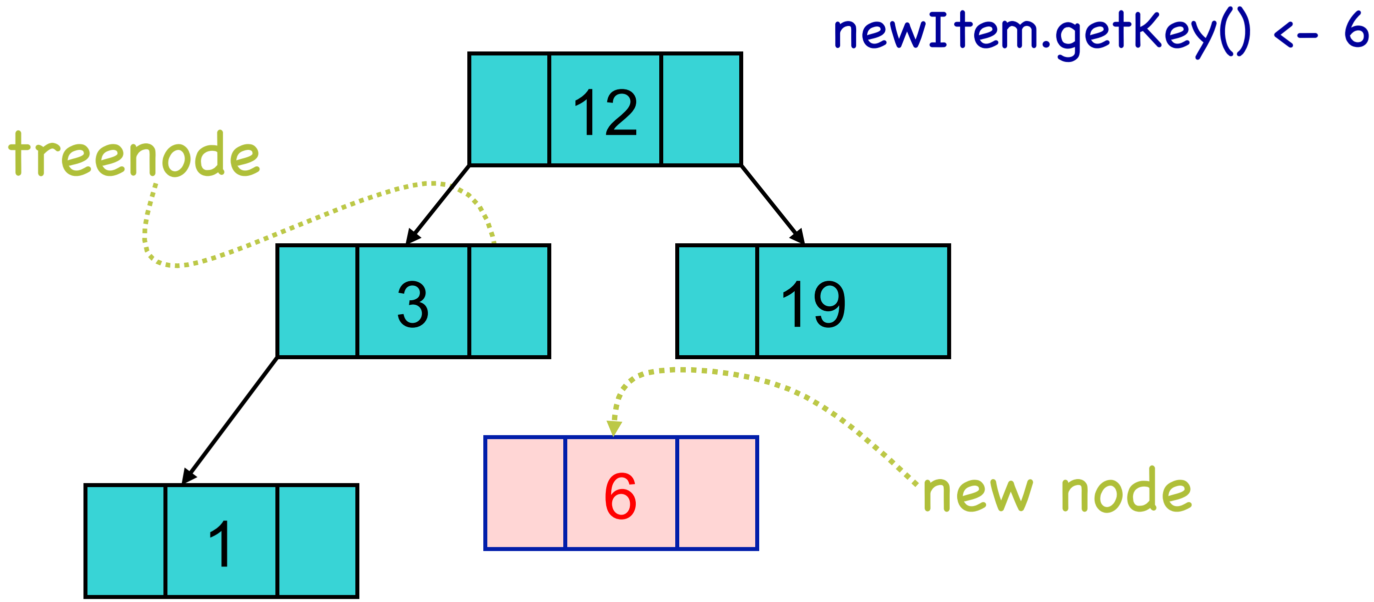

```
if (newItem.getKey() < treeNode.getItem().getKey()) {
                    treeNode.setLeft(insertItem(treeNode.getLeft(), newItem))
```

BST – Insert




```
else {
        treeNode.setRight(insertItem(treeNode.getRight(),newItem))
```

BST – Insert

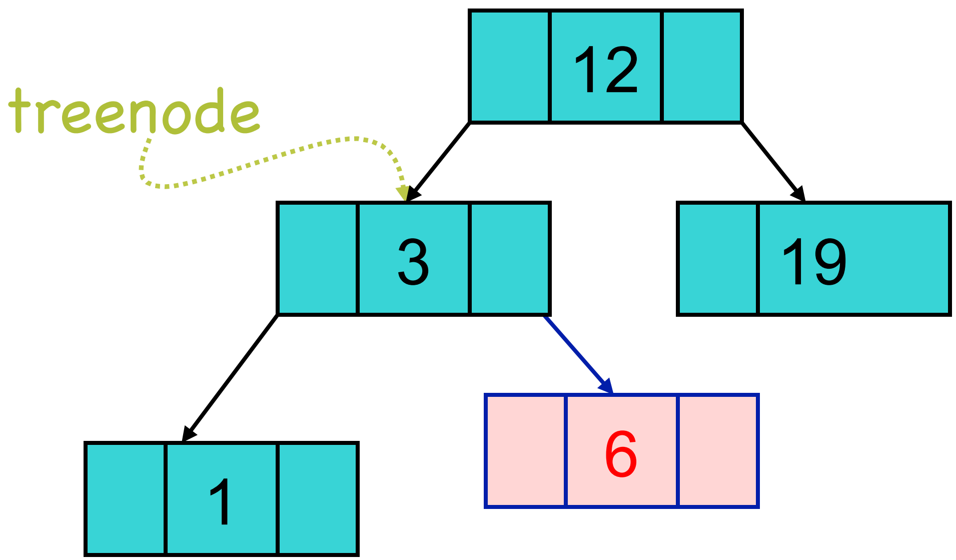


if (treeNode is null) {

create new node with newItem as data

return new node

BST – Insert



treeNode.setRight(insertItem(treeNode.getRight(),newItem))
return treeNode

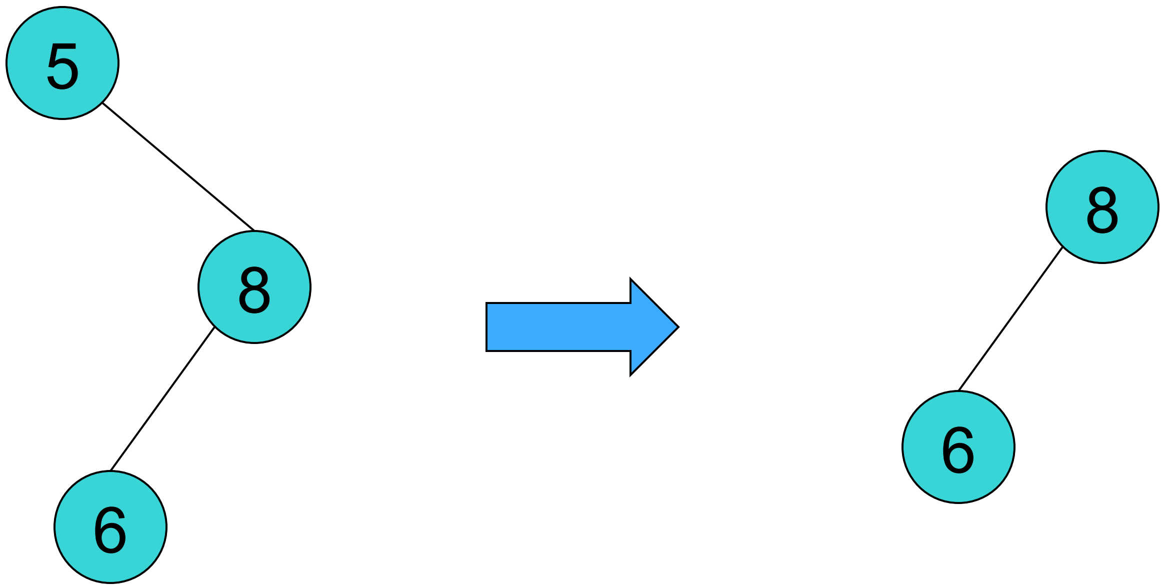
Delete: Cases to Consider

- Delete something that is not there
  - Throw exception
- Delete a leaf
  - Easy, just set link from parent to null
- Delete a node with one child
- Delete a node with two children

Delete

Case 1: one child

delete(5)



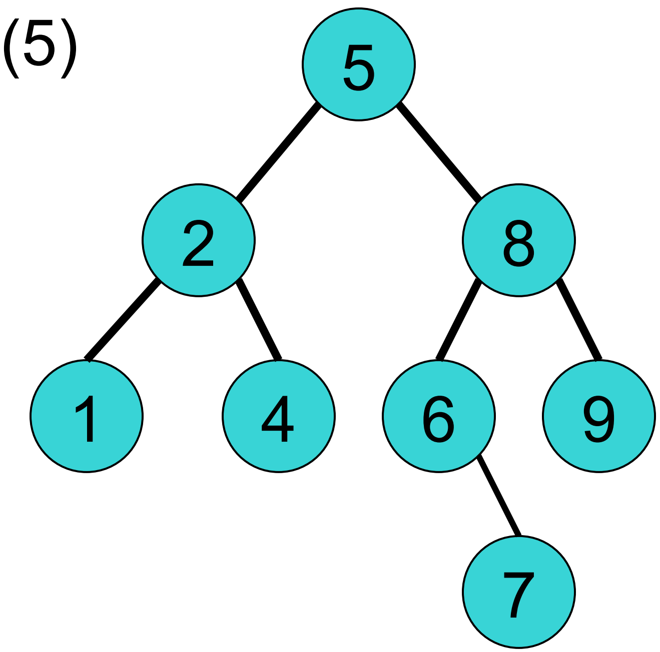
Child becomes root

Delete

Case 2: two children

Which are valid replacement nodes?

delete(5)



4 and 6, WHY?

max of left, min of right

what would be a good one here?

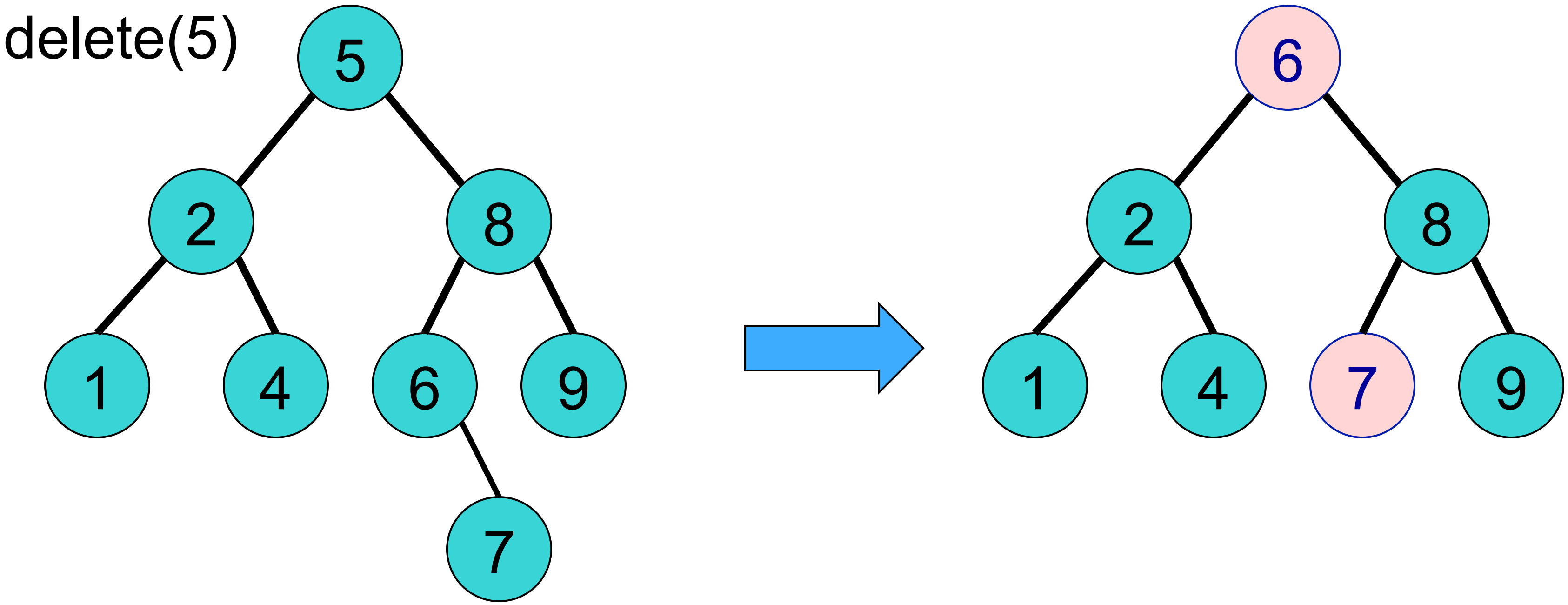
6, WHY?

Digression: inorder traversal of BST

- In order:
  - go left
  - visit the node
  - go right
- The keys of an inorder traversal of a BST are in sorted order!

Delete

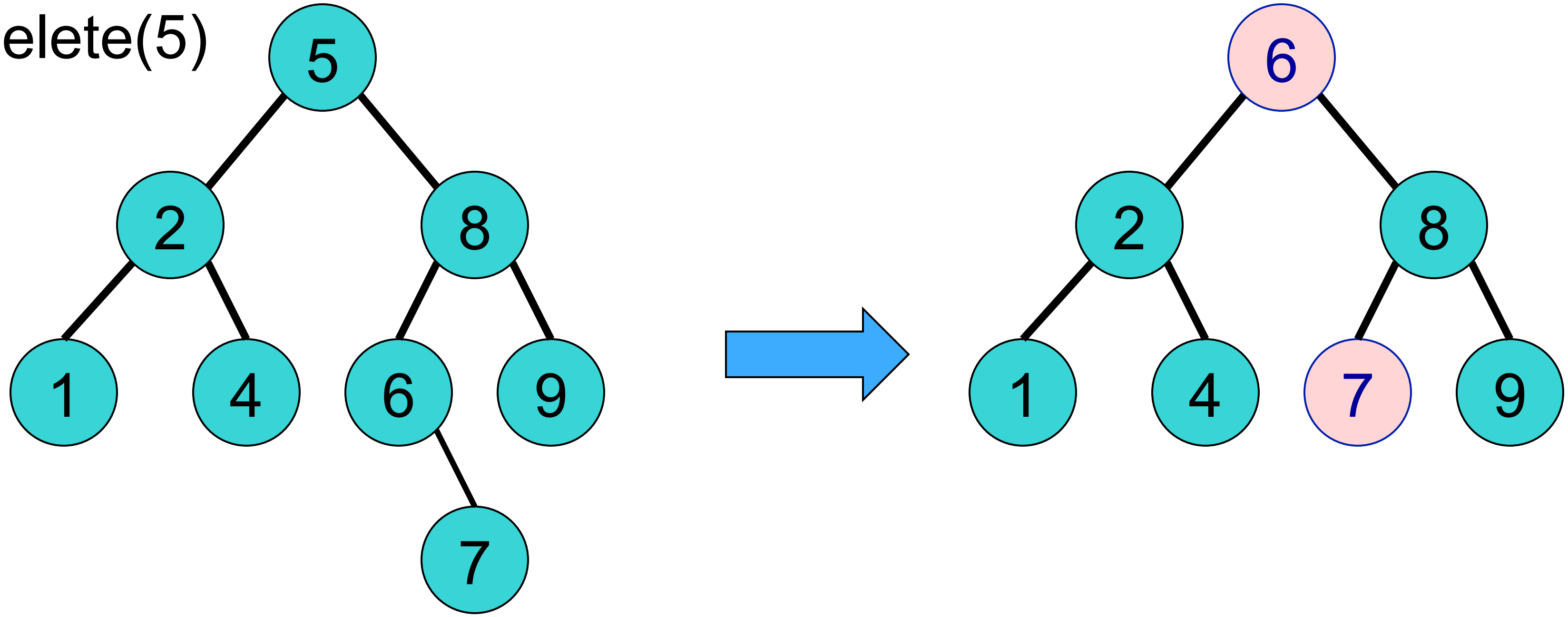
Case 2: two children



Replace root with its **leftmost right descendant** and replace that node with its right child, if necessary (an easy delete case). That node is the inorder successor of the root

Delete Case 2: two children

delete(5)



Replace root with its **leftmost right descendant** and replace that node **with its right child,** if necessary (an easy delete case). That node is the inorder successor of the root.

Can that node have two children? A left child?

Delete

Case 2: two children

1. Find the ***inorder successor*** of N’s search key.
   - The node whose search key comes immediately after N’s search key
   - The inorder successor is in the leftmost node in N’s right subtree.
2. Copy the item of the inorder successor, M, to the deleting node N.
3. Remove the node M from the tree.

Delete Pseudo Code I

```
deleteItem(in rootNode:TreeNode, in searchKey:KeyType): TreeNode
    if (rootNode is null){ throw TreeException}
    else if (searchKey equals key in rootNode item) {      //found it
        newRoot = deleteNode(rootNode)
        return newRoot }
    else if (searchKey < key in rootNode item) {           //search left
        newLeft = deleteItem(rootNode.getLeft(), searchKey)
        rootNode.setLeft(newLeft)
        return rootNode }
    else {              // search right
        newRight = deleteItem(rootNode.getRight(), searchKey)
        rootNode.setRight(newRight)
        return rootNode }
```

remove it

repair links to child nodes

Delete Pseudo Code II

```
deleteNode(in treeNode:TreeNode):TreeNode
    // deletes the item in the node referenced by treeNode
    // returns root of resulting subtree
    if (treeNode is leaf) { return null }
    else if (treeNode has only 1 child c) {
        if (c is left child) { return treeNode.getLeft() }
        else { return treeNode.getRight() }
    }
    else { // find and delete leftmost child on right
      treeNode.setItem(findLeftMostItem(treeNode.getRight()))
      treeNode.setRight(deleteLeftMostNode(treeNode.getRight()));
      return treeNode;
    }
```

Case 1: replace root w/child

Case 2: replace rootItem w/leftmost childItem on right; delete leftMost child on right

Why two methods (not one)?

Delete Pseudo Code III

```
deleteLeftMostNode(in treeNode:TreeNode):TreeNode
    // Deletes the node that is the leftmost descendant of the tree rooted at treeNode
    // Returns subtree of deleted node
    if (treeNode.getLeft() is null)    // found the node to delete
            { return treeNode.getRight() }
    else { // still replacing left nodes
            treeNode.setLeft(deleteLeftMostNode(treeNode.getLeft())
            return treeNode
    }
```

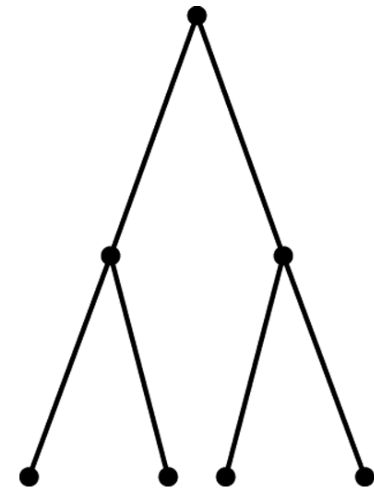
Complexity of BST Operations

	Average	Worst
search	O(log n)	O(n)
insert	O(log n)	O(n)
delete	O(log n)	O(n)

When does worst in BST happen?

Trees - more definitions

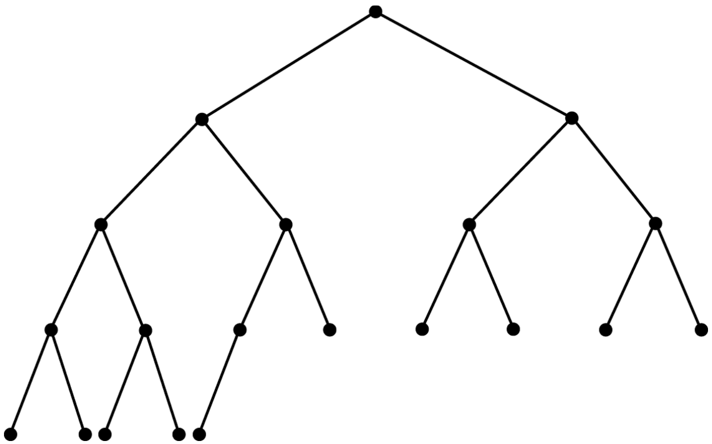
- m-ary tree
  - Every internal vertex has no more than m children.
  - Our main focus will be binary trees
- Full m-ary tree
  - all interior nodes have m children
- Perfect m-ary tree
  - Full m-ary tree where all leaves are at the same level



- Perfect binary tree
  - number of leaf nodes: 2^{h-1}
  - total number of nodes: $2^h - 1$

More definitions

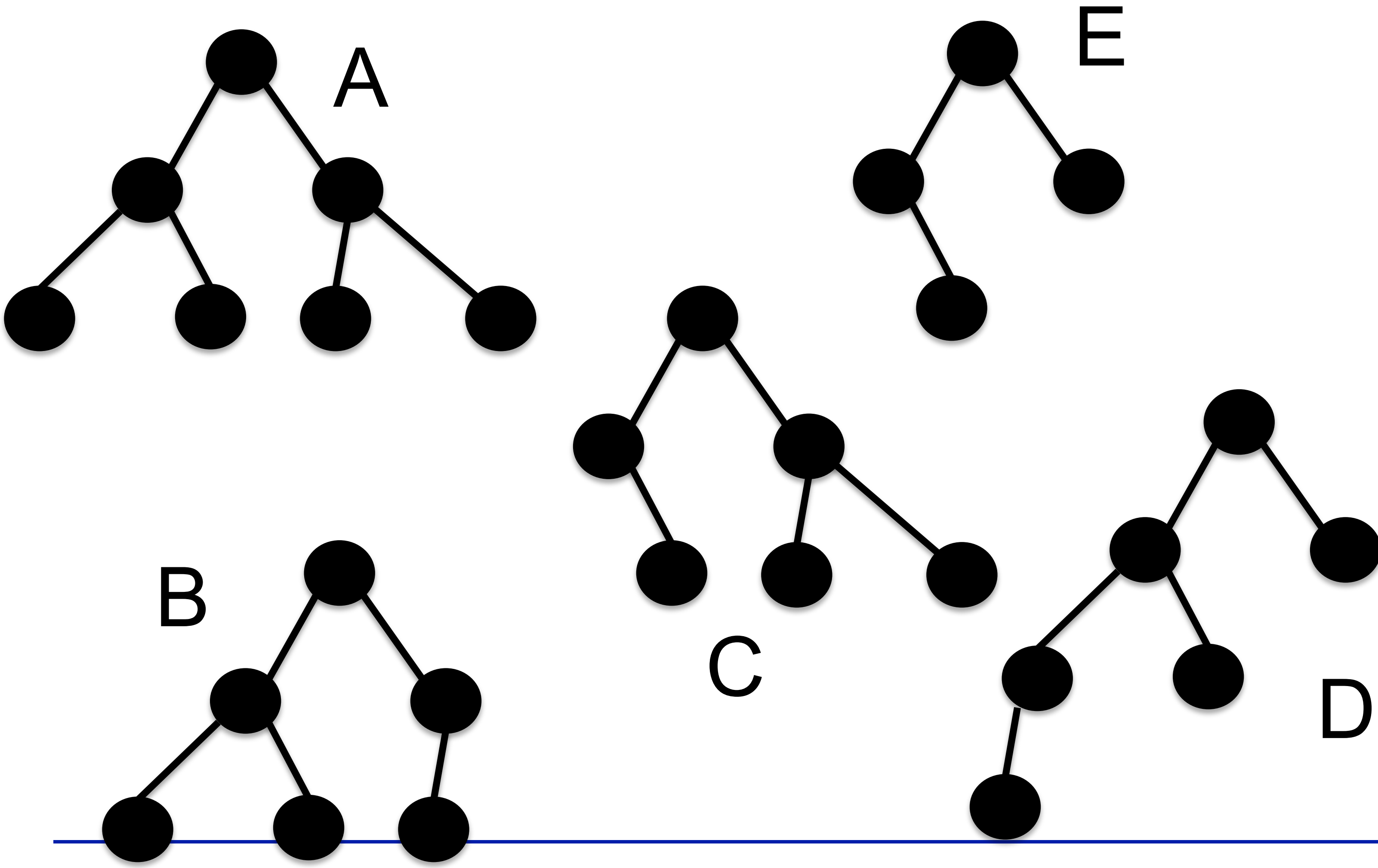
- Complete binary tree of height h
  - zero or more rightmost leaves not present at level h
- A binary tree T of height h is complete if
  - All nodes at level h – 1 and above have two children each, and
  - When a node at level h has children, all nodes to its left at the same level have two children each, and
  - When a node at level h has one child, it is a left child
  - So the leaves at level h go from left to right



More definitions

- balanced tree
  - Height of any node's right subtree differs from left subtree by 0 or 1
- A complete tree is balanced

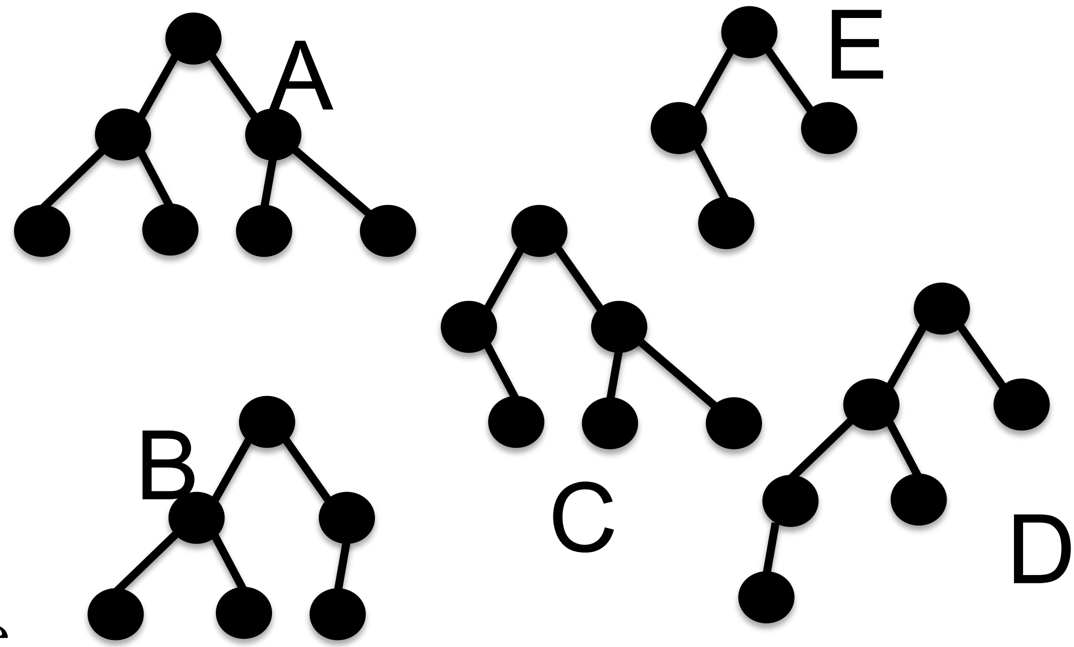
Full? Complete? Balanced?



Question

Full trees are:

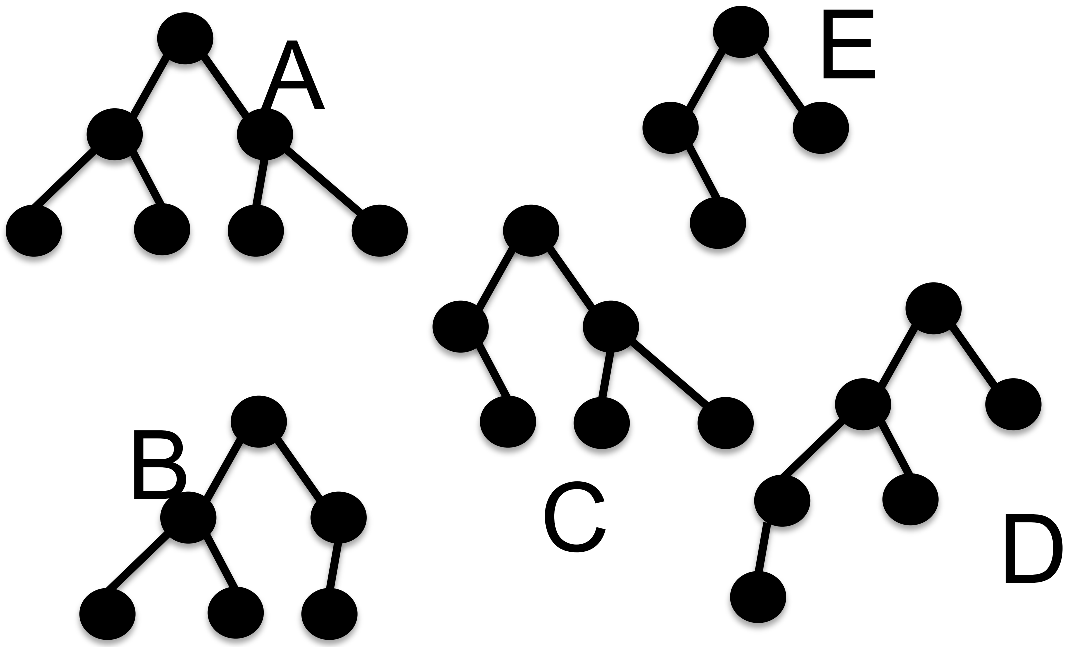
A. {}
B. {A}
C. {A,B}
D. {A,B,C}
E. None of the above



Question

Complete trees are:

A. {}
B. {A}
C. {A,B}
D. {A,B,C}
E. None of the above



Question

Balanced trees are:

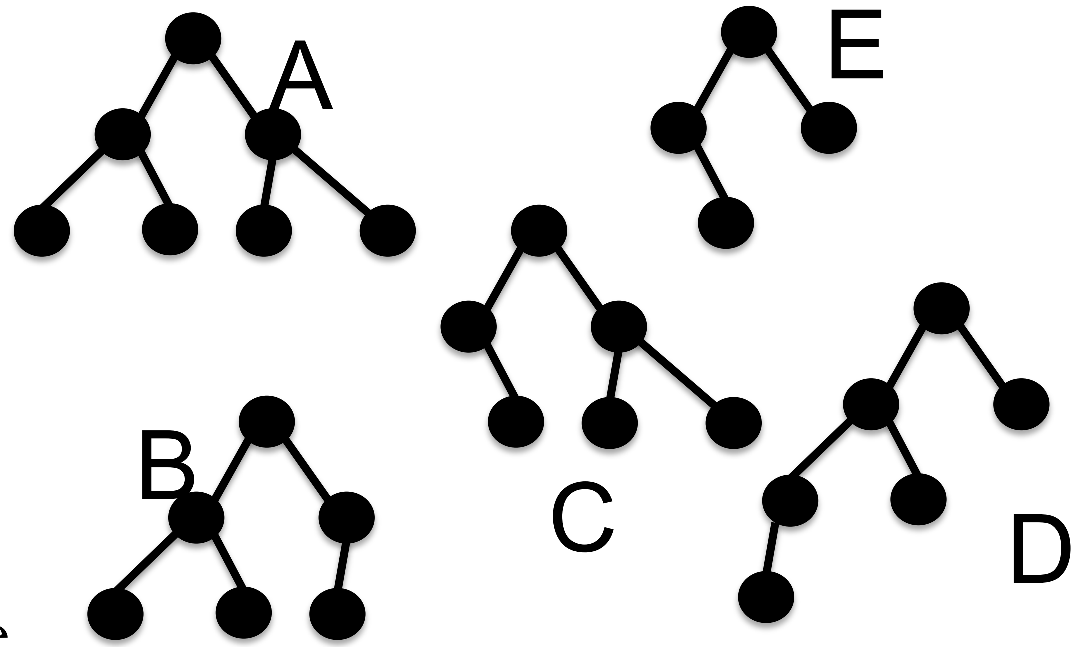
A. {}

B. {A}

C. {A,B}

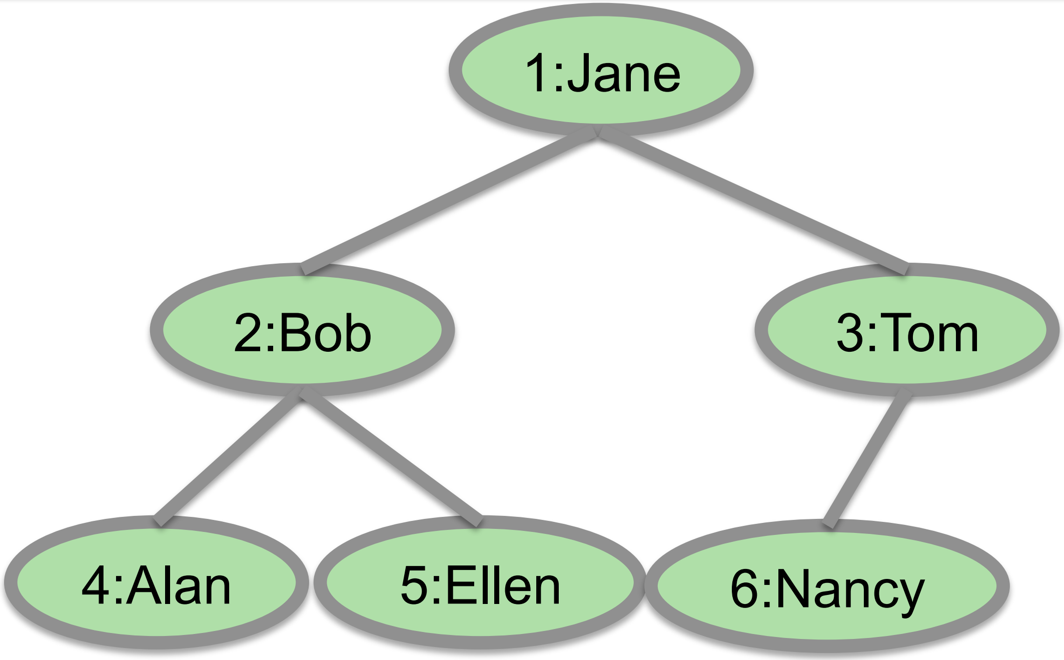
D. {A,B,C}

E. None of the above



Complete Binary Tree

Level-by-level numbering of a complete binary tree



What is the parent child index relationship?.

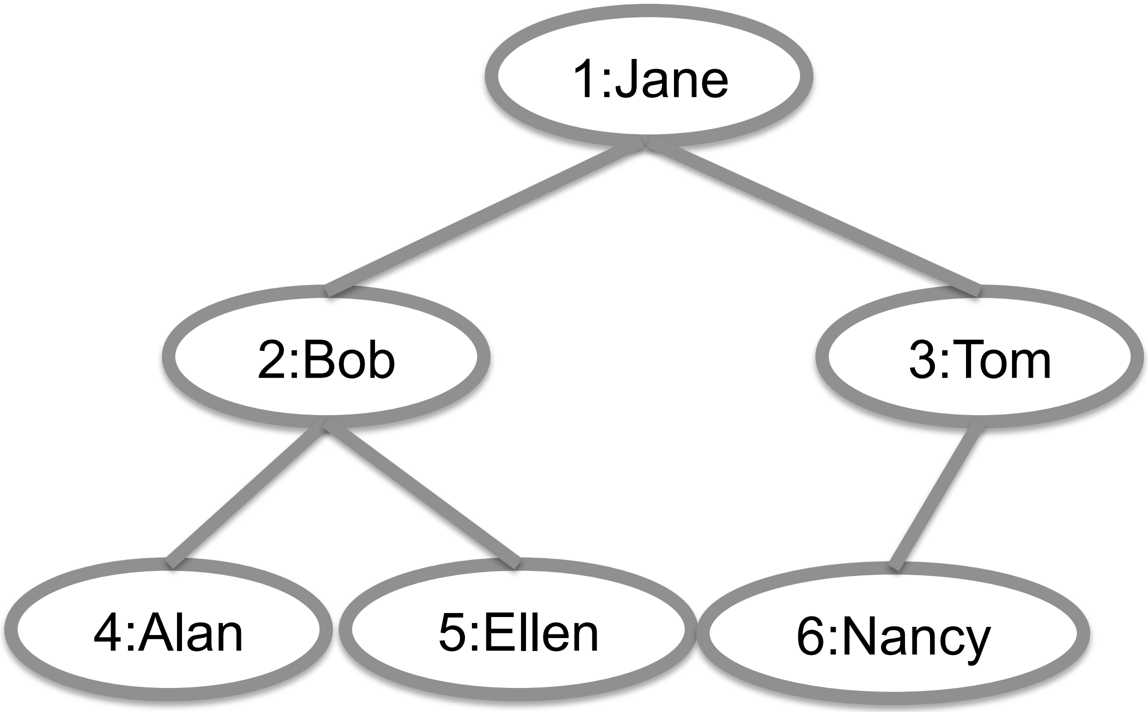
*left child i: at 2*i.*

*right child i: at 2*i+1.*

lparent i: at i/2.

Question

What is the maximum number of nodes in a complete binary tree with Prichard height h?



Properties of Trees (Rosen)

1. A tree has a unique path between any two of its vertices.

2. A tree with n vertices has n-1 edges.

3. A full *bin*ary tree with n internal nodes n+1 leaves.

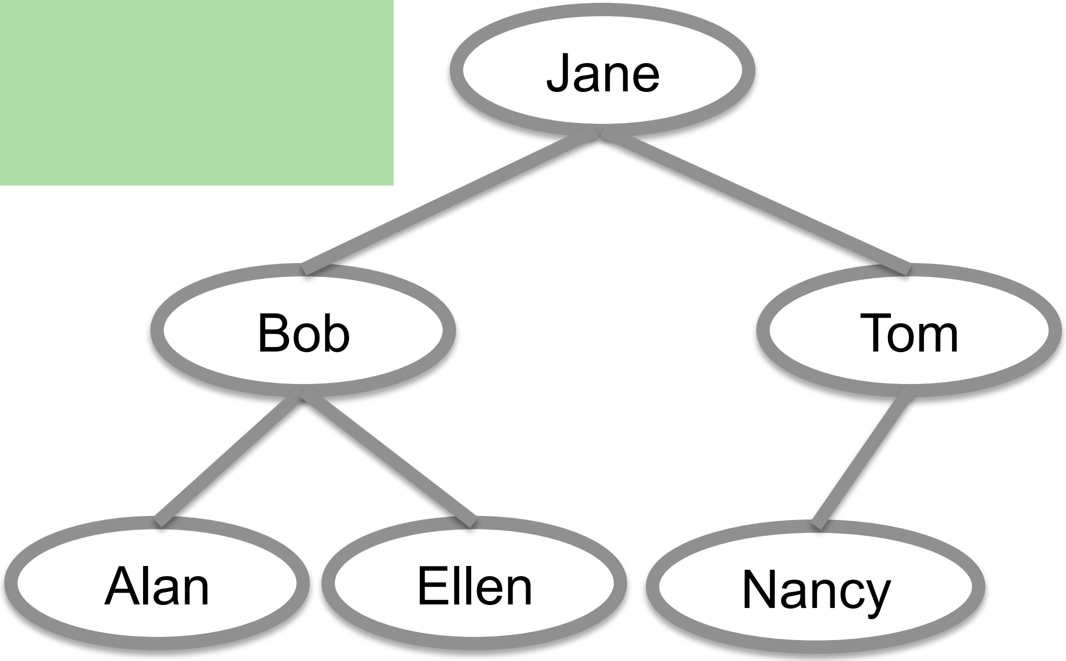
Question

Question : What is the maximum number of nodes at level m (root at level 1) in a binary tree?

A. 2^m

B. 2^{m-1}

C. 2^{m+1}



Sorting with a Tree

- Uses the binary search tree ADT to sort an array of records according to search-key
- Efficiency
  - Average case: O(n * log n)
  - Worst case: $O(n^2)$

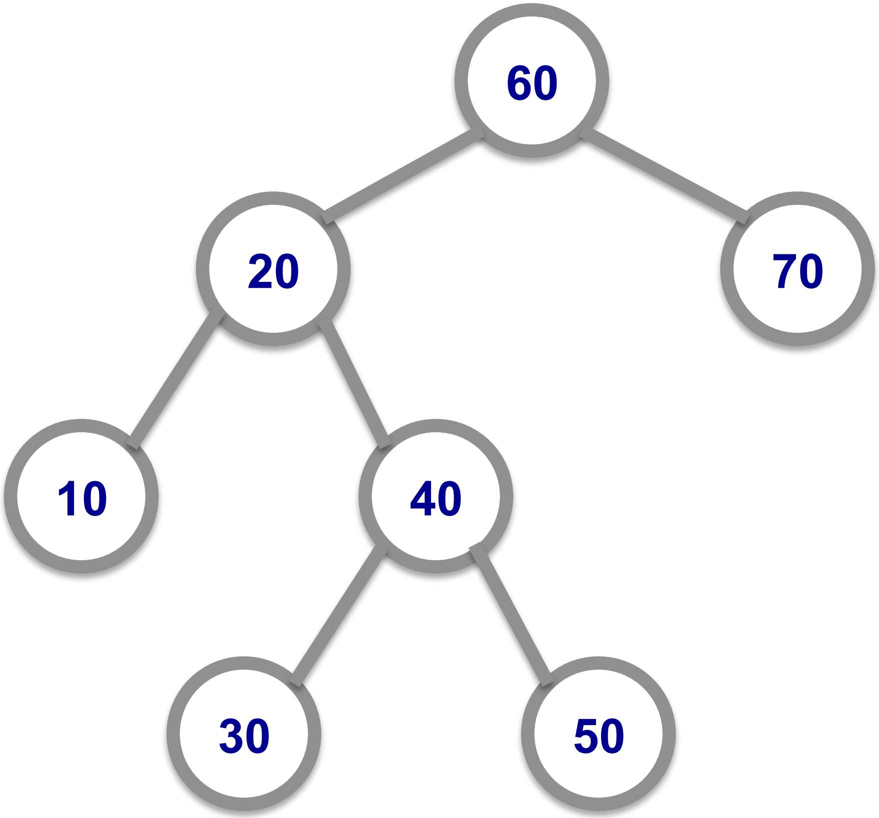
Example of Binary sorting

Create Tree

60 20 10 40 70 50 30

Inorder traverse Tree

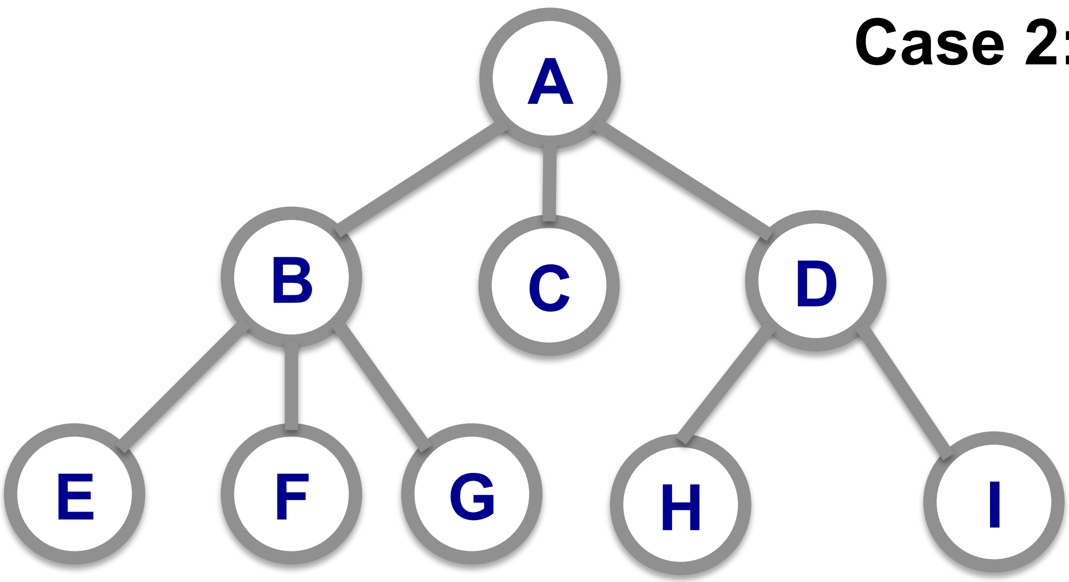
10 20 30 40 50 60 70



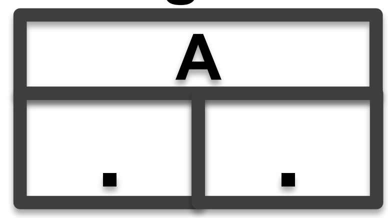
n-ary General tree

- Tree with nodes that have no more than *n* children.
- How can we implement it?

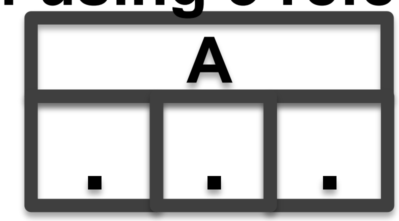
$n = 3$



Case 1: using 2 references

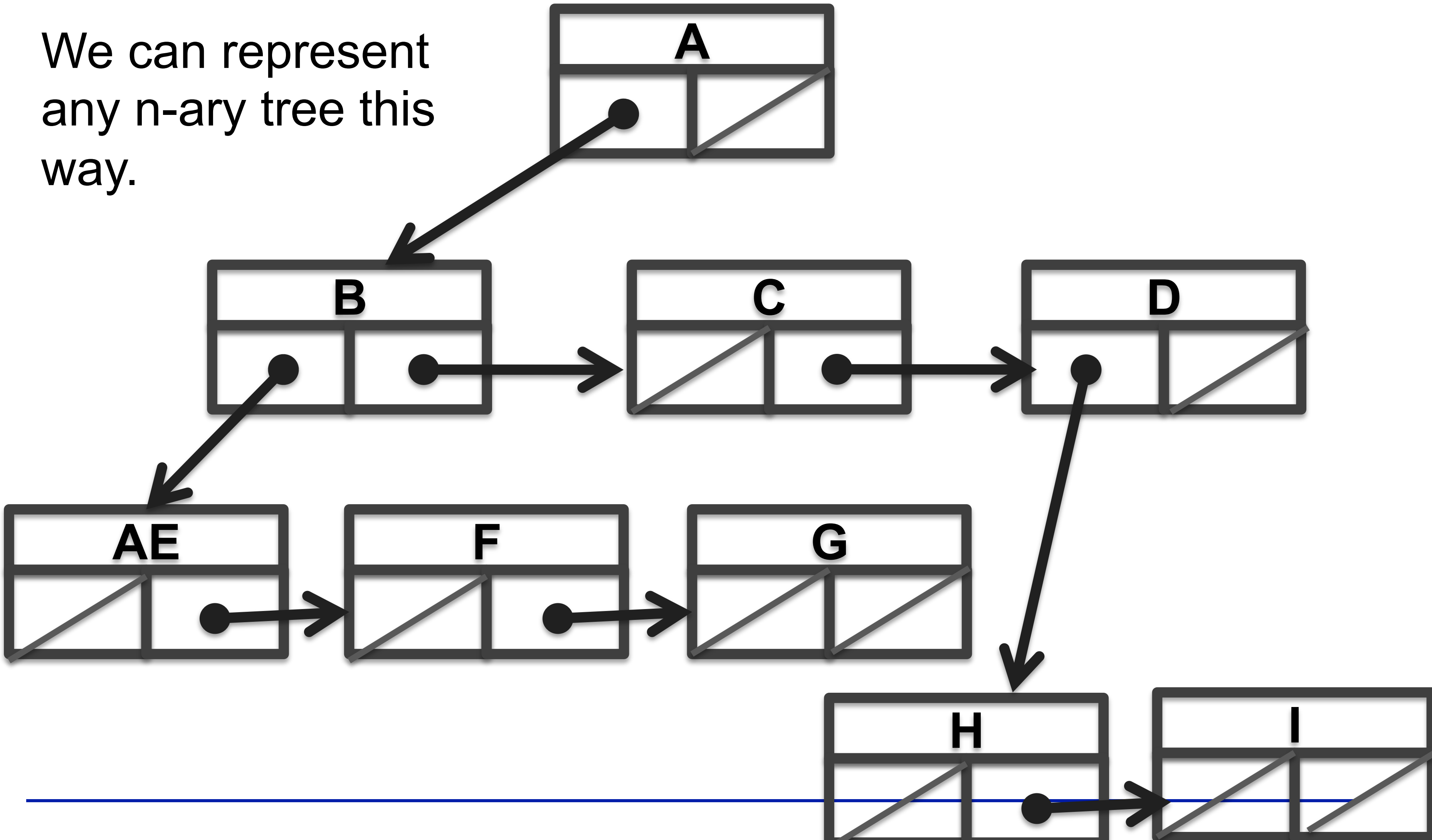


Case 2: using 3 references



Case 1: Using 2 references

We can represent any n-ary tree this way.



Case 2: Using 3 references

more direct, used in search trees, and parse trees