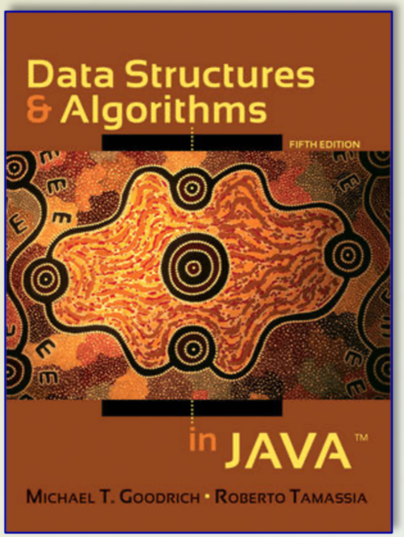
Data Structure & Algorithms in JAVA

5^{th} edition

Michael T. Goodrich
Roberto Tamassia



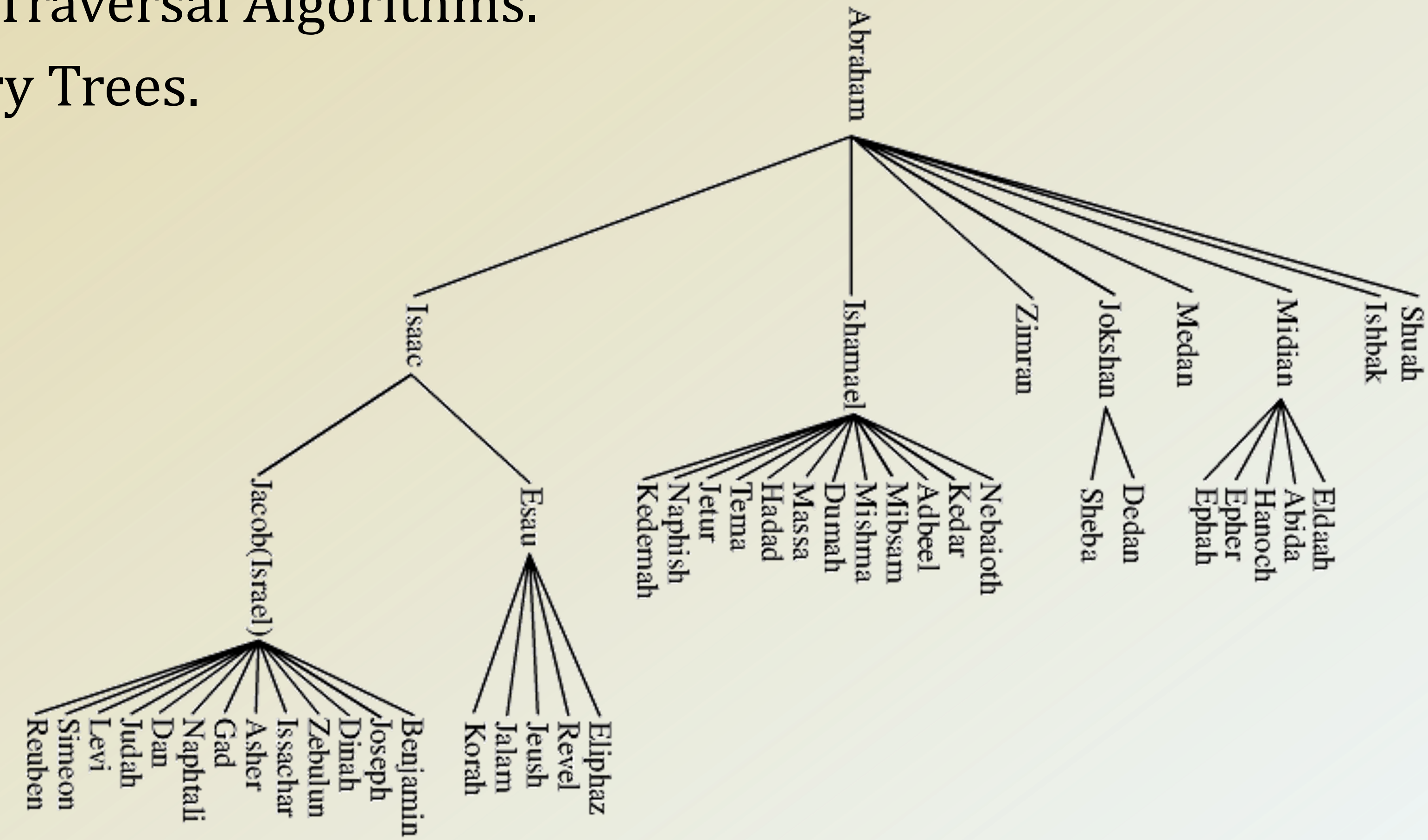
Chapter 7: Trees

CPSC 3200

Algorithm Analysis and Advanced Data Structure

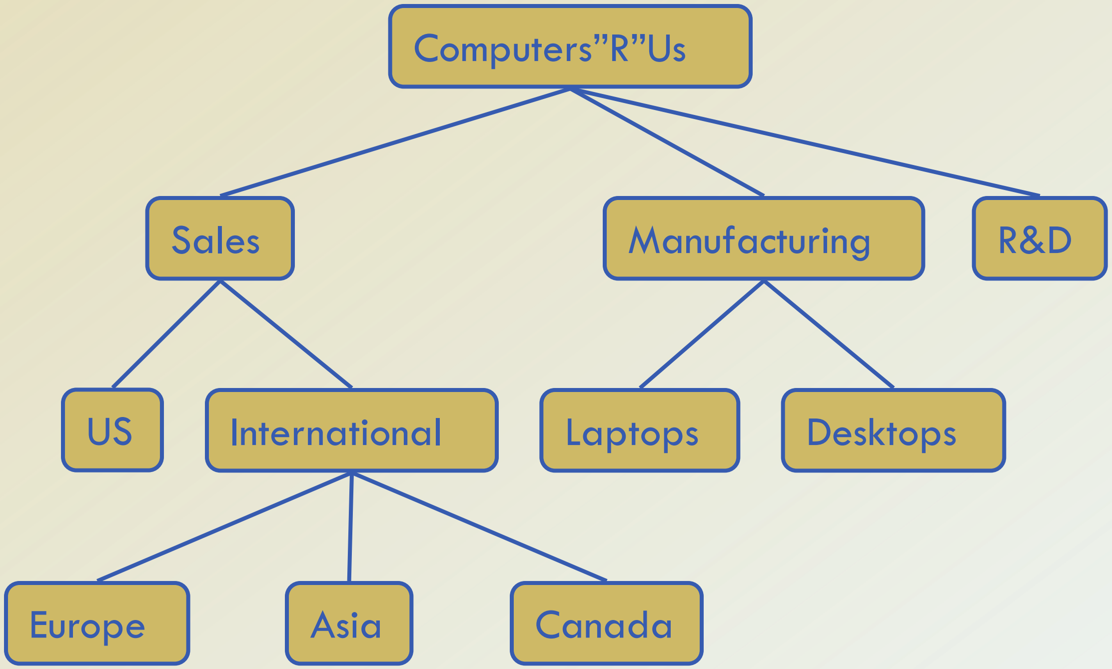
Chapter Topics

- General Trees.
- Tree Traversal Algorithms.
- Binary Trees.



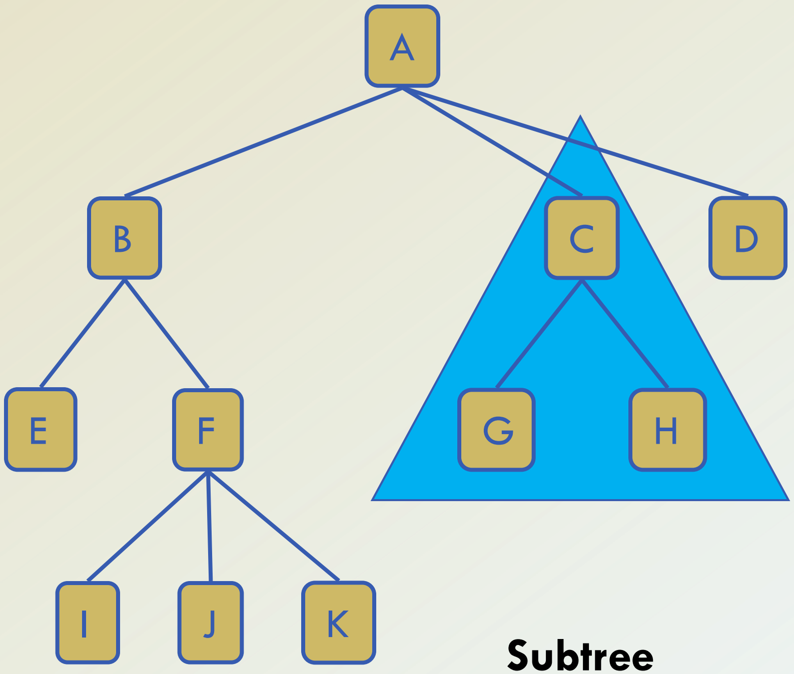
What is a Tree

- In computer science, a tree is an abstract model of a **hierarchical structure**.
- A tree consists of nodes with a **parent-child** relation.
- **Applications:**
  - Organization charts.
  - File systems.
  - Programming environments.



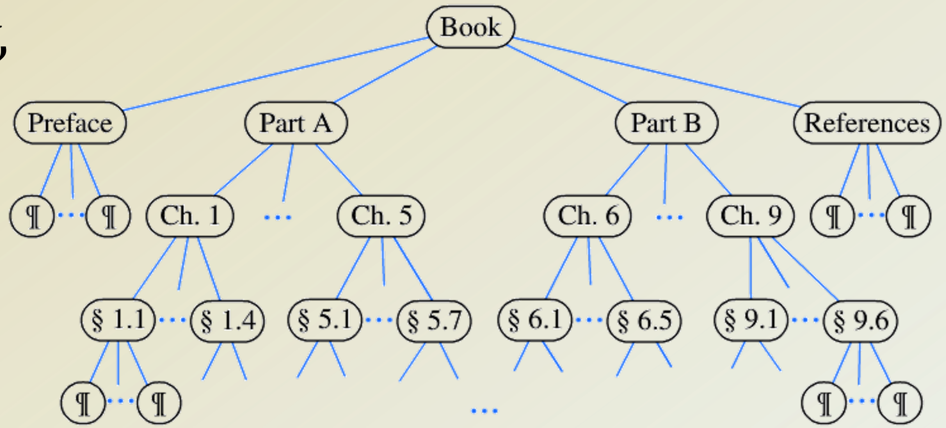
Tree Terminology

- **Root:** node without parent (A)
- **Internal node:** node with at least one child (A, B, C, F)
- **External node (a.k.a. leaf):** node without children (E, I, J, K, G, H, D)
- **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- **Depth of a node:** number of ancestors
- **Height of a tree:** maximum depth of any node (3)
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- **Subtree:** tree consisting of a node and its descendants.



Tree Terminology (Cont.)

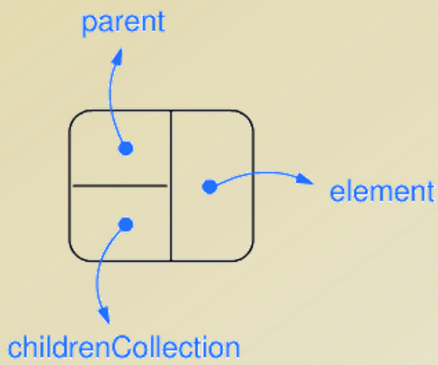
- **edge of tree T** is a pair of nodes (u,v) such that u is the parent of v, or vice versa.
- **Path of T** is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.
- A tree is **ordered** if there is a linear ordering defined for the children of each node

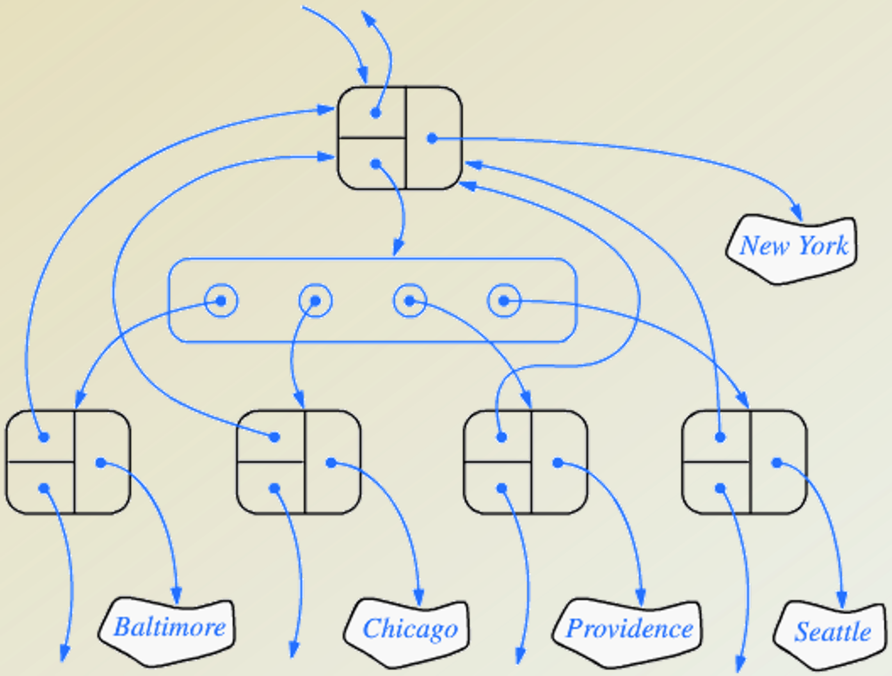


Tree ADT

- We use positions (nodes) to abstract nodes.
  - **getElement( )**: Return the object stored at this position.
- **Generic methods:**
  - integer **getSize( )**
  - boolean **isEmpty( )**
  - Iterator **iterator( )**
  - Iterable **positions( )**
- **Accessor methods:**
  - position **getRoot( )**
  - position **getThisParent(p)**
  - Iterable **children(p)**
- **Query methods:**
  - boolean **isInternal(p)**
  - boolean **isExternal(p)**
  - boolean **isRoot(p)**
- **Update method:**
  - element **replace (p, o)**
- Additional update methods may be defined by data structures implementing the Tree ADT.

Linked structure for General Tree





Operation	Time
size, isEmpty	$O(1)$
iterator, positions	$O(n)$
replace	$O(1)$
root, parent	$O(1)$
children(v)	$O(c_v)$
isInternal, isExternal, isRoot	$O(1)$

Depth and Height

- Let ***v*** be a node of a tree **T**. The **depth** of ***v*** is the number of ancestors of ***v***, excluding ***v*** itself.
  - If ***v*** is the root, then the depth of ***v*** is 0
  - Otherwise, the depth of ***v*** is one plus the depth of the parent of ***v***.

Algorithm depth(T, v):
if *v* is the root of T **then**
  return 0
else
  return 1+depth(T, *w*), where w is the parent of *v* in T

- The running time of algorithm depth(T, v) is $\mathbf{O(d_v)}$, where $\mathbf{d_v}$ denotes the depth of the node ***v*** in the tree T.

Data Structure (Tree)

- A tree is a data structure which stores elements in **parent-child** relationship.

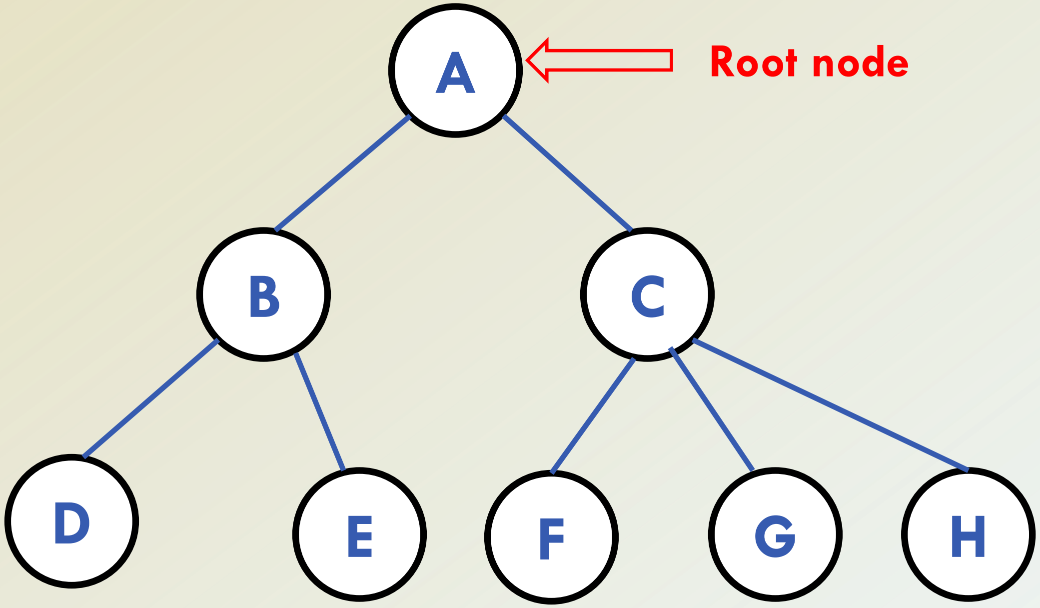
Internal nodes

Leaf nodes (External nodes)

Siblings

Siblings

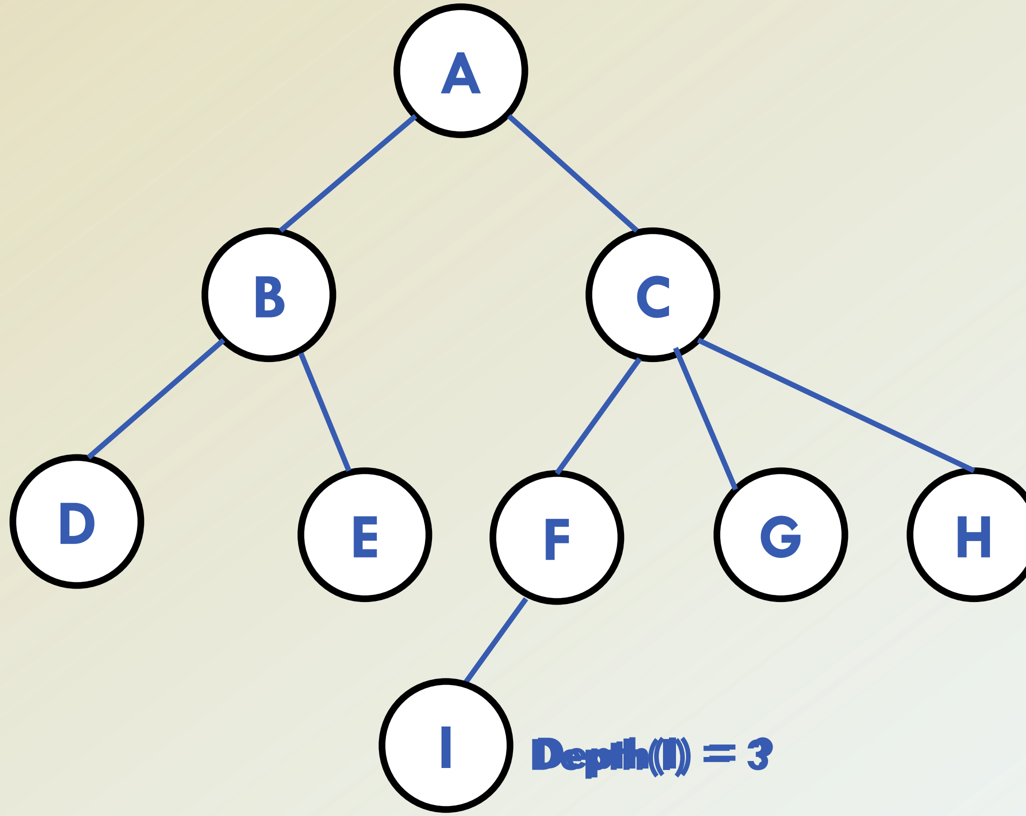
Siblings



Attributes of a tree

- **Depth:** the number of ancestors of that node (excluding itself).
- **Height:** the maximum depth of an external node of the tree/subtree.

Depth(D) = 2



Height = MAX[Depth(A), Depth(B), Depth(C), Depth(D), Depth(E), Depth(F), Depth(G), Depth(H), Depth(I)]

Height = MAX[0, 1, 1, 2, 2, 2, 2, 2, 3] = 3

Depth and Height (Cont.)

- The **height** of a node ***v*** in a tree T is can be calculated using the **depth** algorithm.

> **Algorithm height1(T):**
> $h \leftarrow 0$
> **for** each vertex v in T **do**
>   **if** v is an external node in T **then**
> $h \leftarrow \max(h, \mathrm{depth}(T, v))$
> **return** h

- algorithm **height1** runs in $O(\mathbf{n}^2)$ time

Depth and Height (Cont.)

- The **height** of a node ***v*** in a tree T is also defined recursively:
  - If ***v*** is an external node, then the height of ***v*** is 0
  - Otherwise, the height of ***v*** is one plus the maximum height of a child of ***v***.

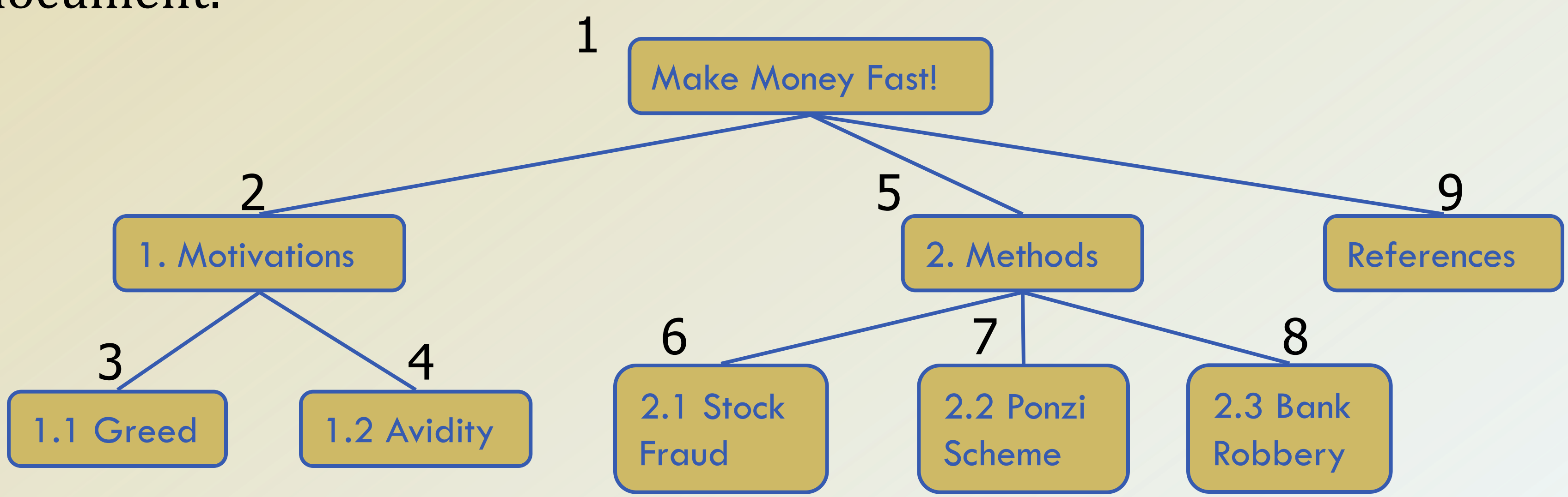
```
Algorithm height2(T, v):
if v is an external node in T then
    return 0
else
    h ← 0
for each child w of v in T do
    h ← max(h, height2(T, w))
return 1+h
```

- algorithm **height1** runs in O(**n**) time

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner.
- In a **preorder** traversal, a node is visited before its descendants.
- Application: print a structured document.

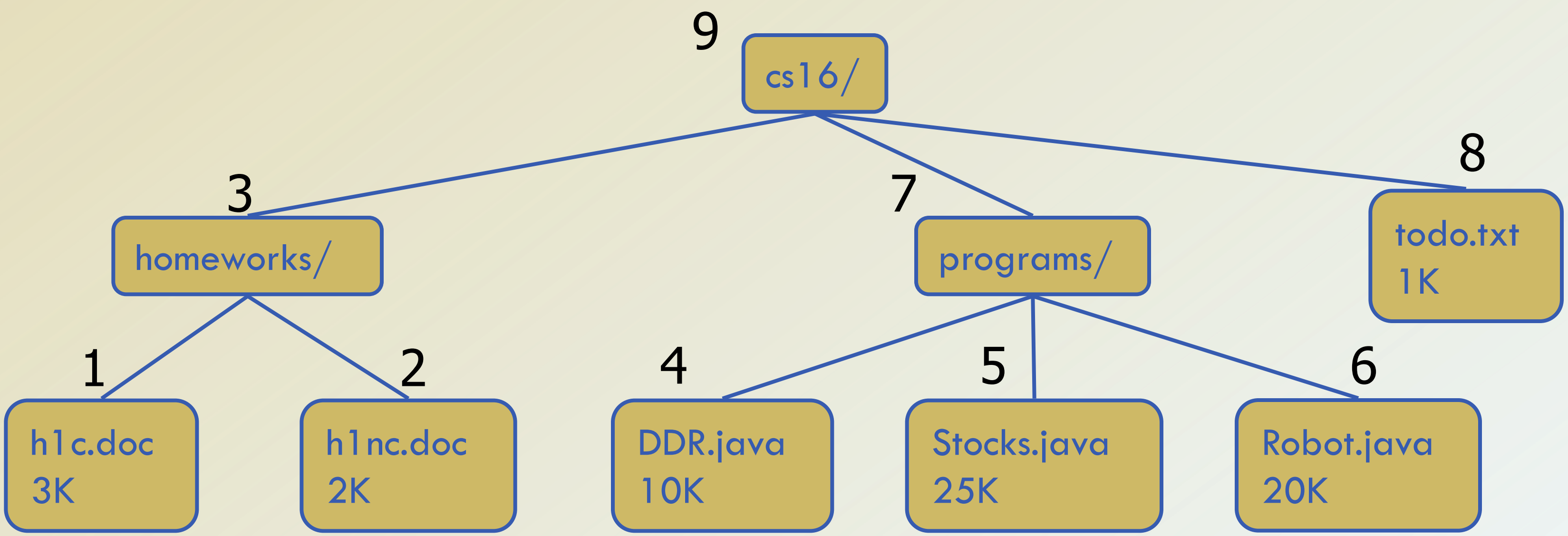
Algorithm ***preOrder***(*v*)
    visit(*v*)
    for each child ***w*** of ***v***
        preorder (***w***)



Postorder Traversal

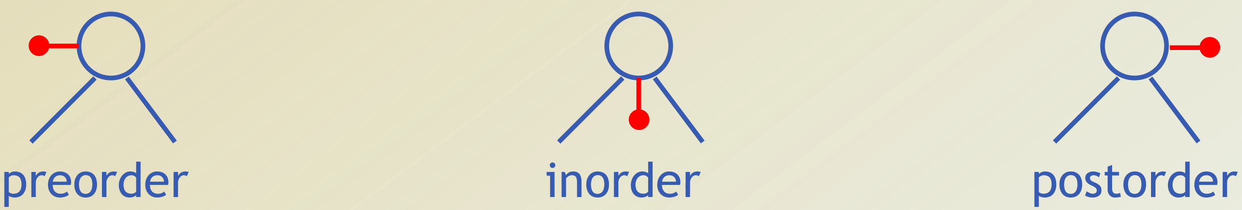
- In a **postorder** traversal, a node is visited after its descendants.
- Application: compute space used by files in a directory and its subdirectories.

Algorithm ***postOrder***(***v***)
    for each child ***w*** of ***v***
        postOrder (***w***)
    visit(***v***)

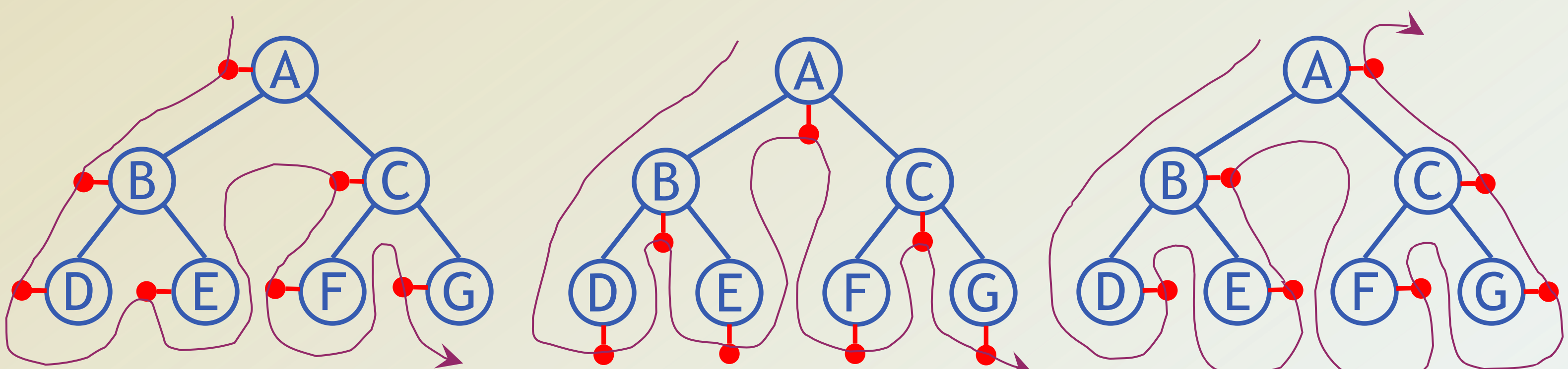


Tree traversals using "flags"

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:



- To traverse the tree, collect the flags:



A B D E C F G   D B E A F C G   D E B F G C A

Other traversals

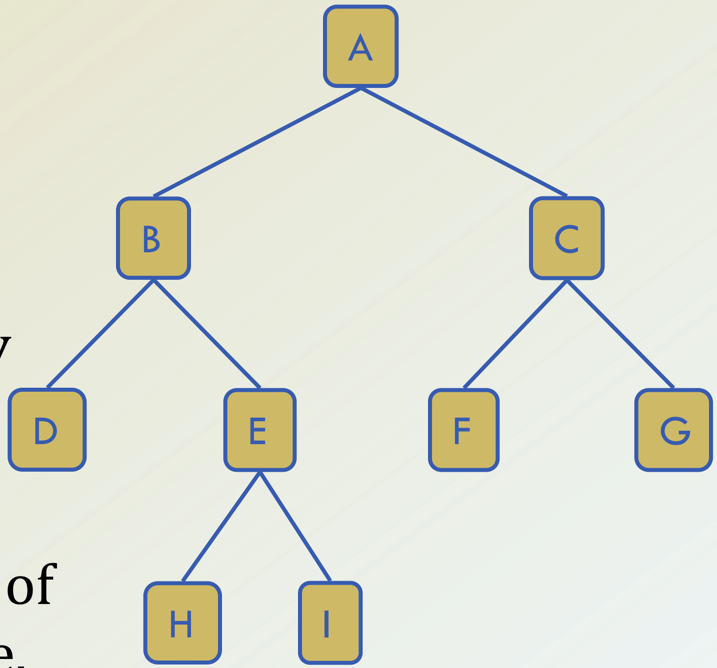
- The other traversals are the reverse of these three standard ones
  - That is, the right subtree is traversed before the left subtree is traversed
- **Reverse preorder:** root, right subtree, left subtree.
- **Reverse inorder:** right subtree, root, left subtree.
- **Reverse postorder:** right subtree, left subtree, root.

Binary Trees

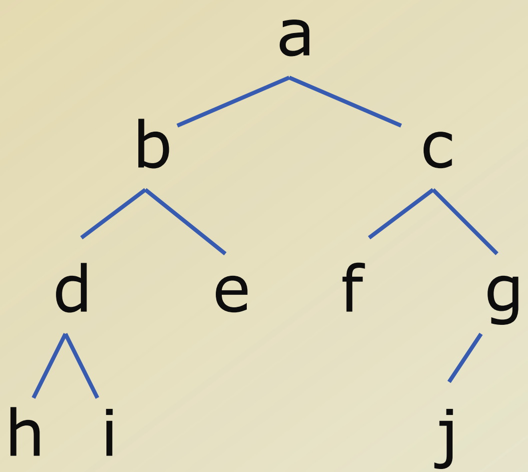
- A **binary tree** is a tree with the following properties:
  - Each internal node has **at most two children** (**exactly two for proper binary trees**).
  - The children of a node are an ordered pair.
- We call the children of an internal node left child and right child.
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree.

Applications:

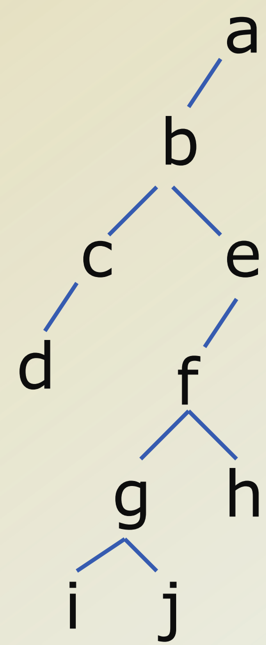
- arithmetic expressions.
- decision processes.
- searching.



Tree Balance



A balanced binary tree

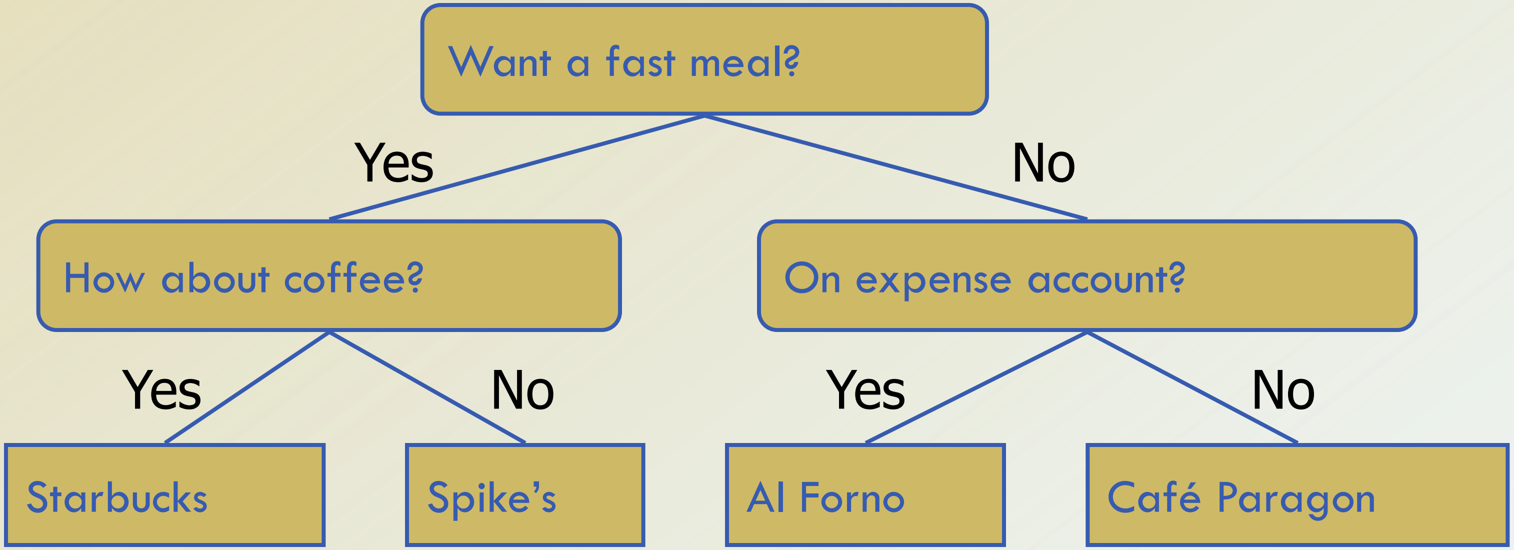


An unbalanced binary tree

- A binary tree is balanced if every level above the lowest is “full” (contains 2^h nodes)
- In most applications, a reasonably balanced binary tree is desirable.

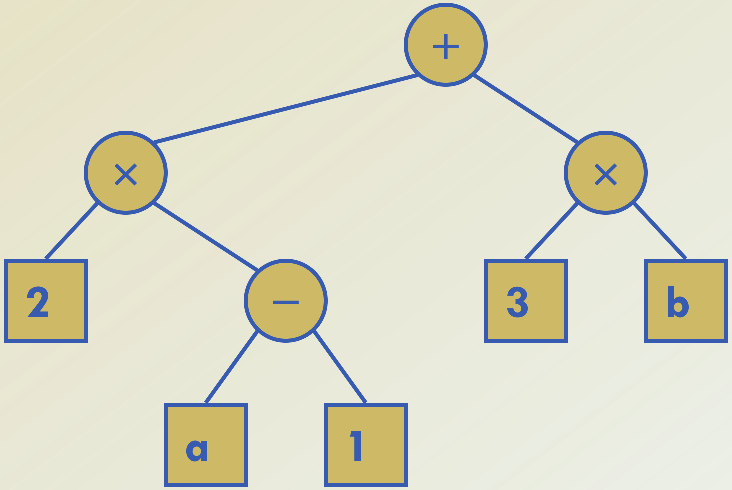
Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- **Example:** dining decision



Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- **Example:** arithmetic expression tree for the expression
  $(2 \times (a - 1) + (3 \times b))$



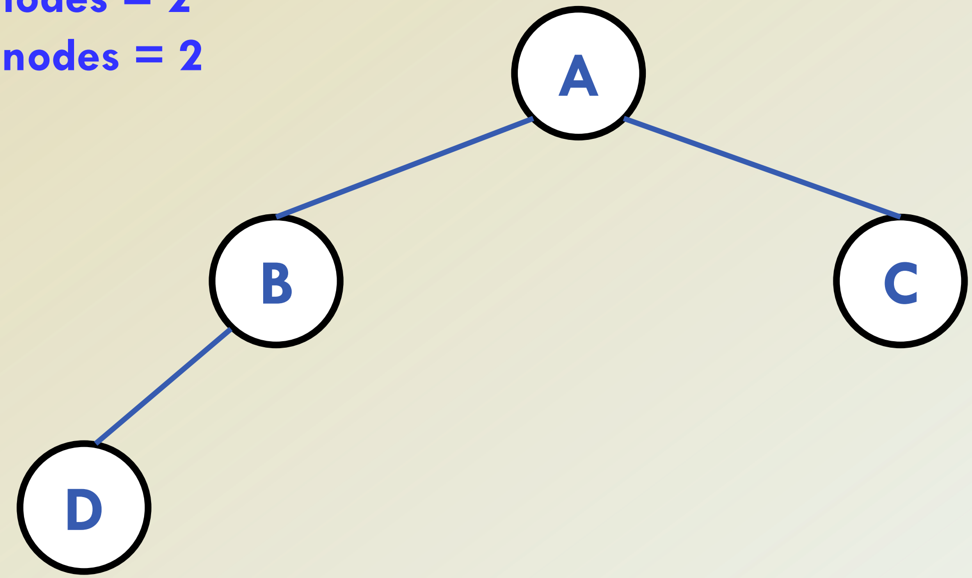
Proper Binary Tree

- Is a binary tree where the number of external nodes is 1 more than the number of internal nodes.

Proper Binary Tree

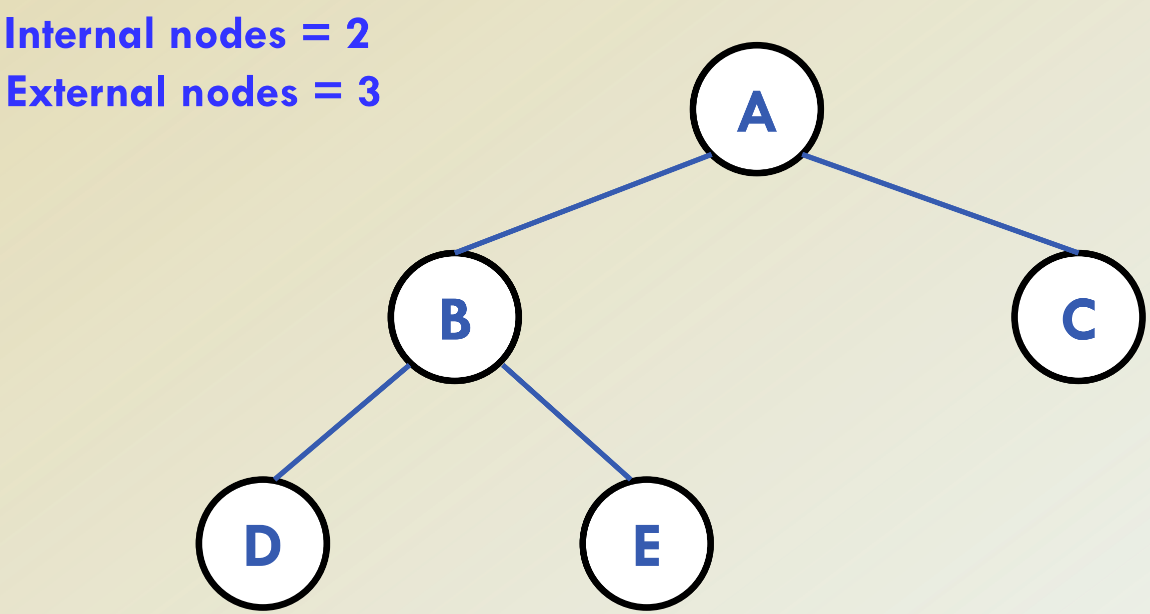
- Is a binary tree where the number of external nodes is 1 more than the number of internal nodes.

Internal nodes = 2
External nodes = 2



Proper Binary Tree

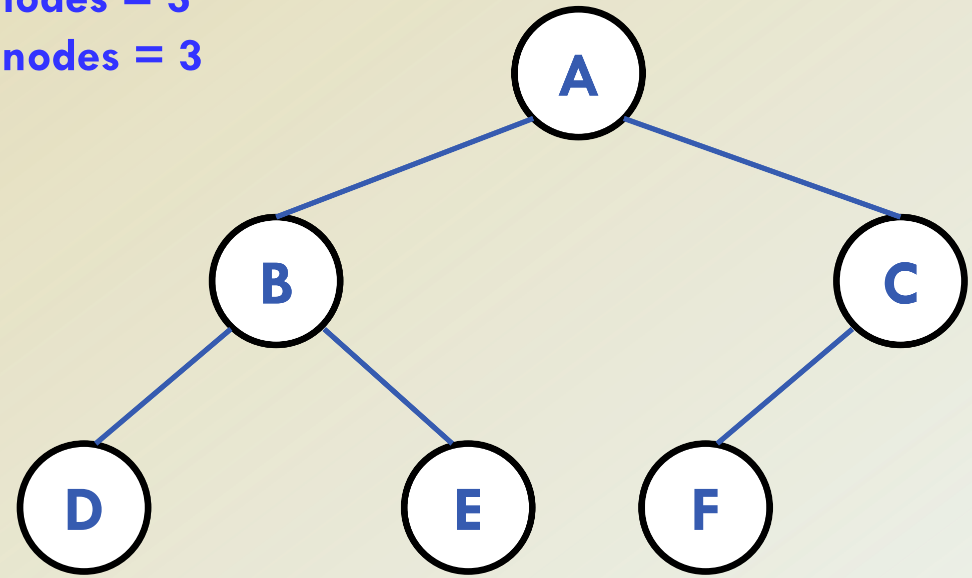
- Is a binary tree where the number of external nodes is 1 more than the number of internal nodes.



Proper Binary Tree

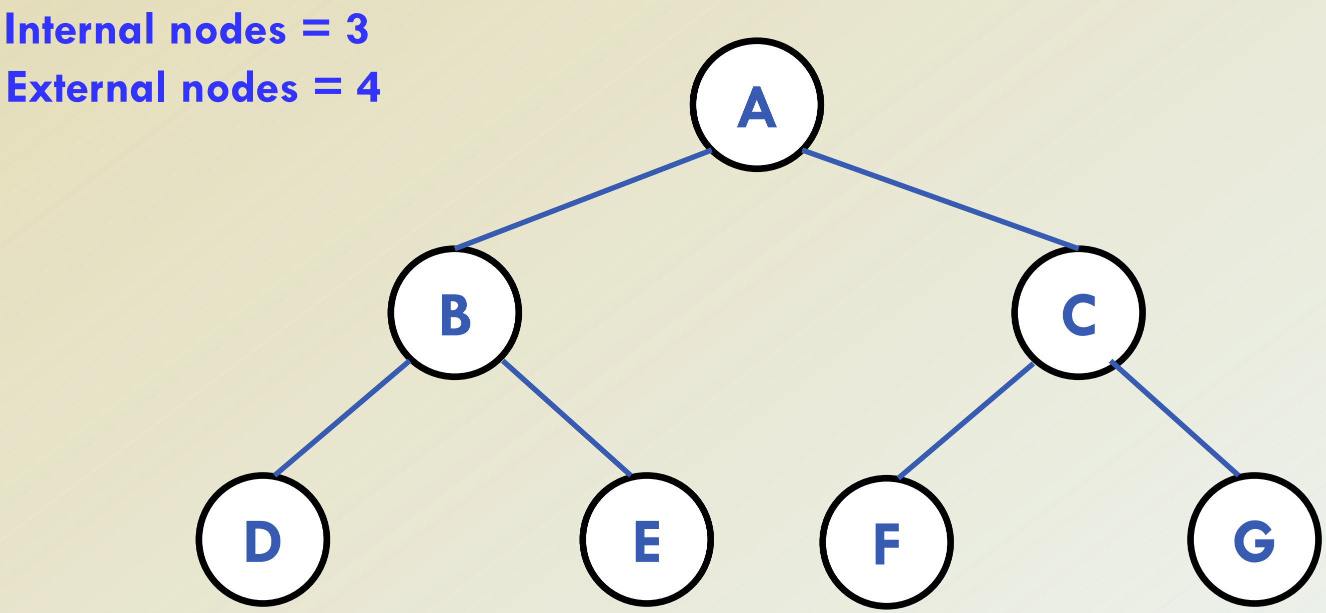
- Is a binary tree where the number of external nodes is 1 more than the number of internal nodes.

Internal nodes = 3
External nodes = 3



Proper Binary Tree

- Is a binary tree where the number of external nodes is 1 more than the number of internal nodes.

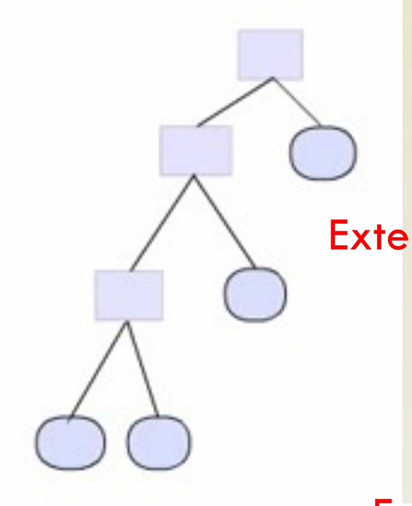


Properties of a Proper Binary Tree

1. The number of external nodes is at least h+1 and at most 2^h

Ex: h = 3

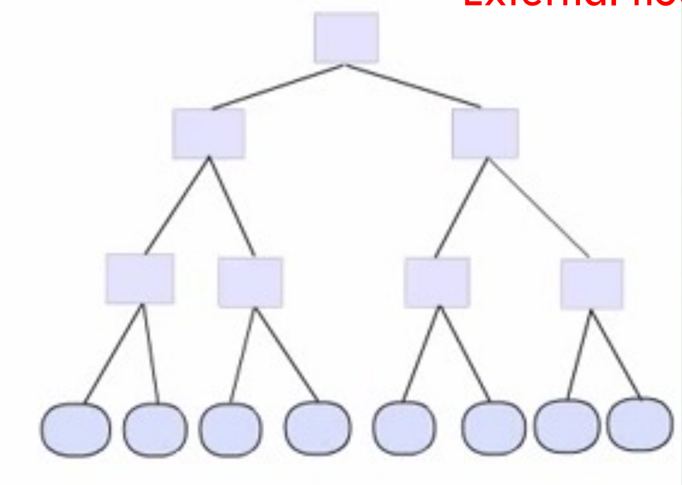
Worst case: The tree having the minimum number of external and internal nodes.



External nodes = 3+1 = 4

Best case: The tree having the maximum number of external and internal nodes.

External nodes = 2^3 = 8

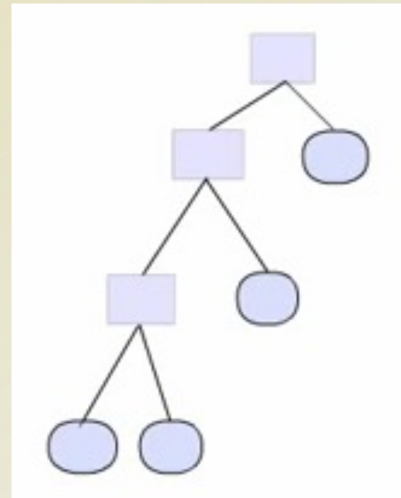


Properties of a Proper Binary Tree

2. The number of internal nodes is at least h and at most 2^h-1

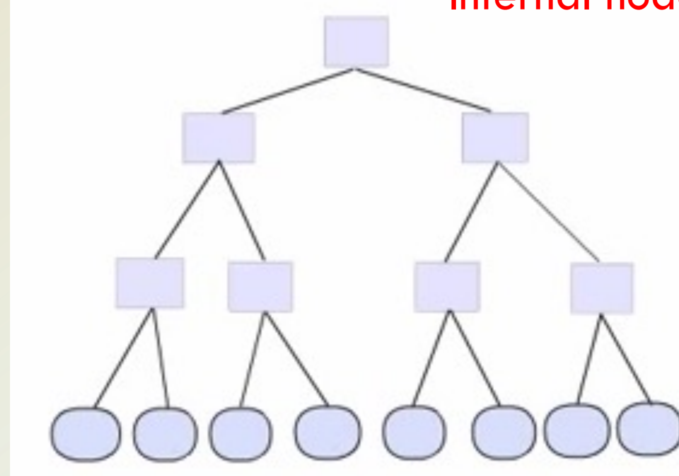
Ex: h = 3

Worst case: The tree having the minimum number of external and internal nodes.



Internal nodes = 3

Internal nodes = 2^3 -1=7



Best case: The tree having the maximum number of external and internal nodes.

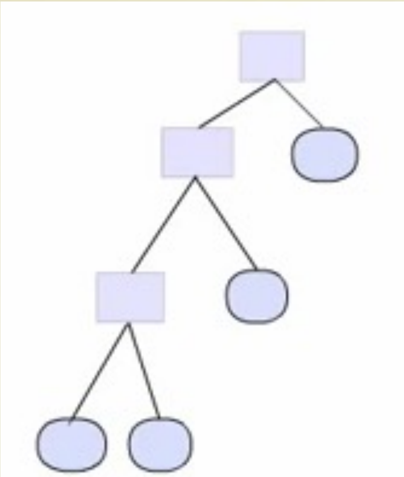
Properties of a Proper Binary Tree

3. The number of nodes is at least 2h+1 and at most 2^{h+1} -1

Ex: h = 3

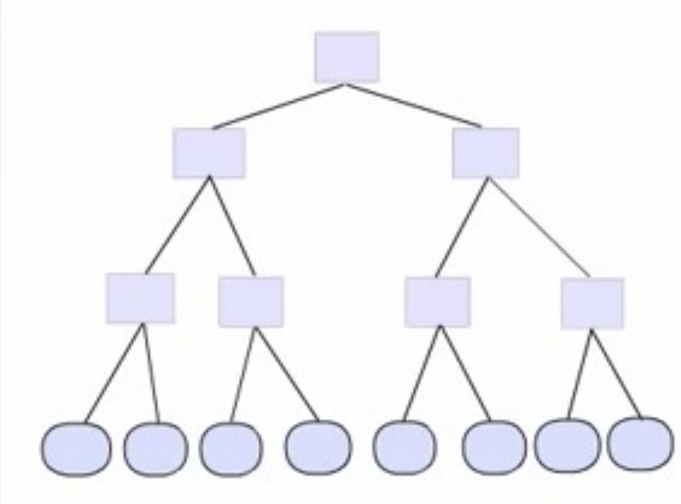
Internal nodes = 3
External nodes = 4

Internal + External = 2*3 +1 = 7



Internal nodes = 7
External nodes = 8

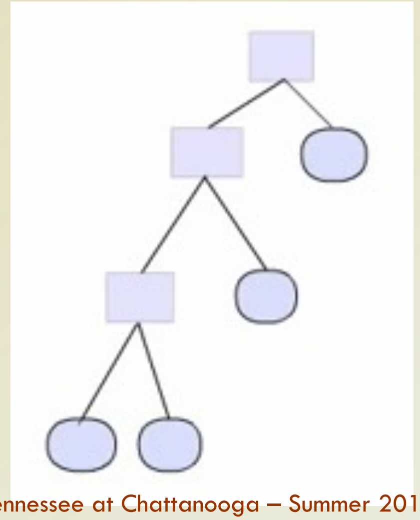
Internal + External = $2^{3+1} - 1 = 15$



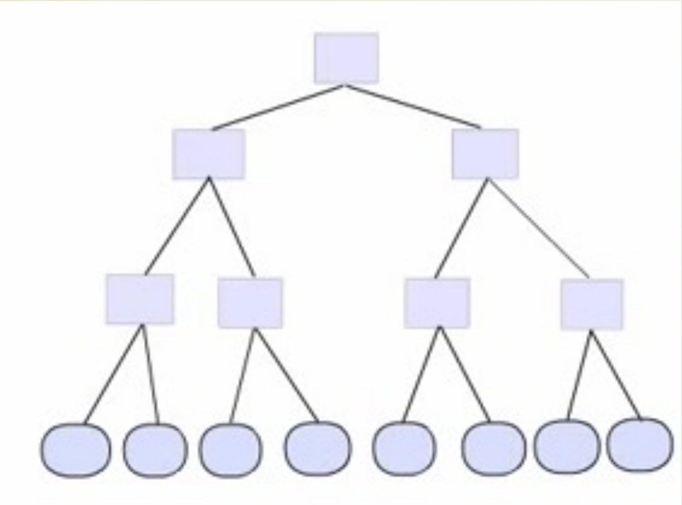
Properties of a Proper Binary Tree

4. The height is at least log(n+1)-1 and at most (n-1)/2

Number of nodes = 7
h = 3



Number of nodes = 15
h = 3

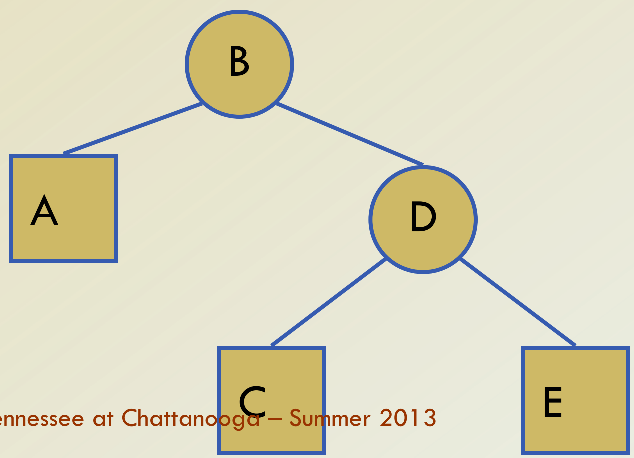


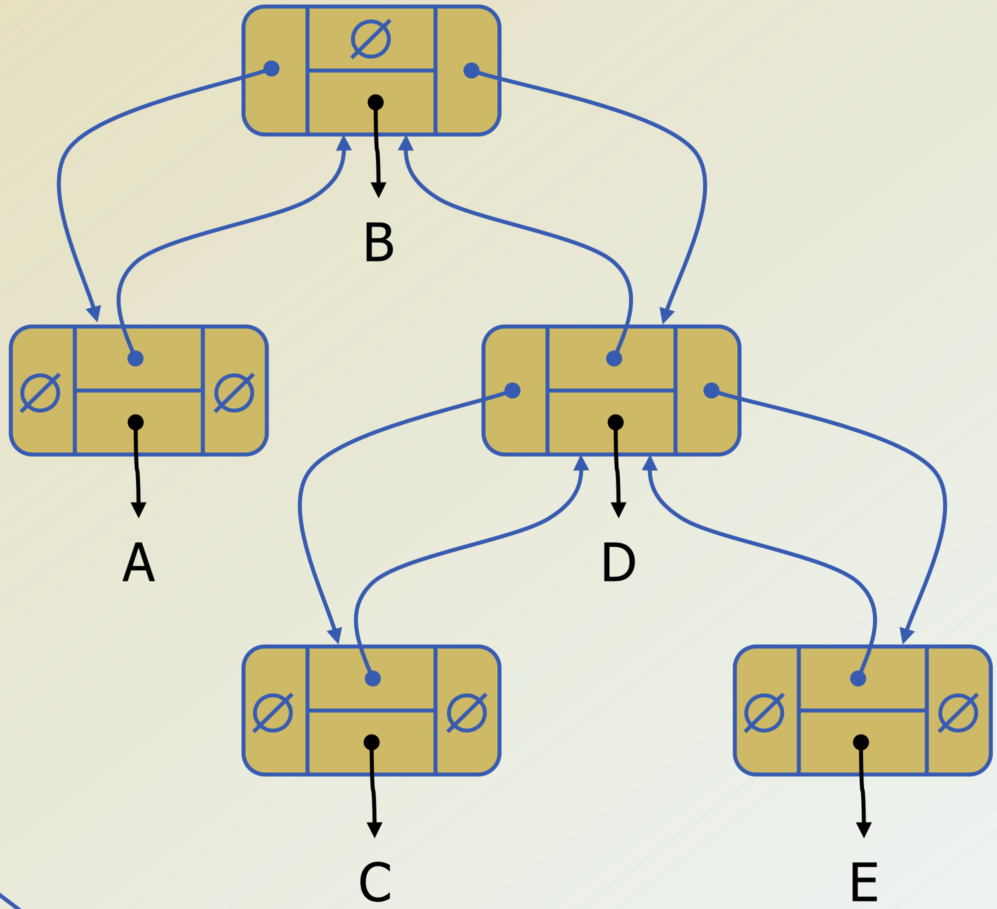
BinaryTree ADT

- The BinaryTree ADT **extends** the Tree ADT, i.e., it inherits all the methods of the Tree ADT.
- **Additional methods:**
  - position **getThisLeft(p)**
  - position **getThisRightight(p)**
  - boolean **hasLeft(p)**
  - boolean **hasRight(p)**
- Update methods may be defined by data structures implementing the BinaryTree ADT.

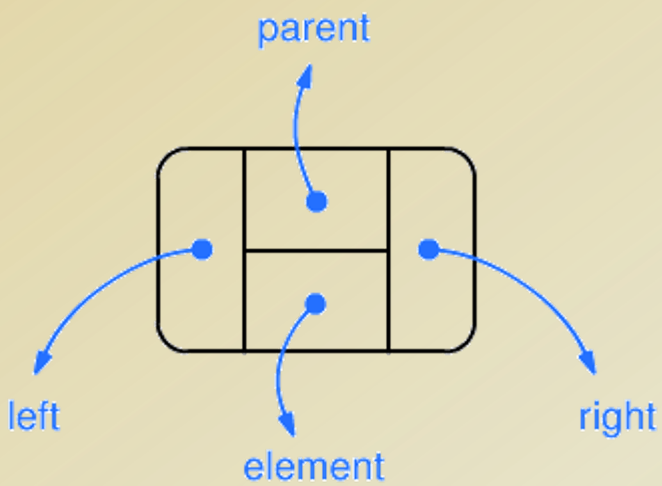
Linked Structure for Binary Trees

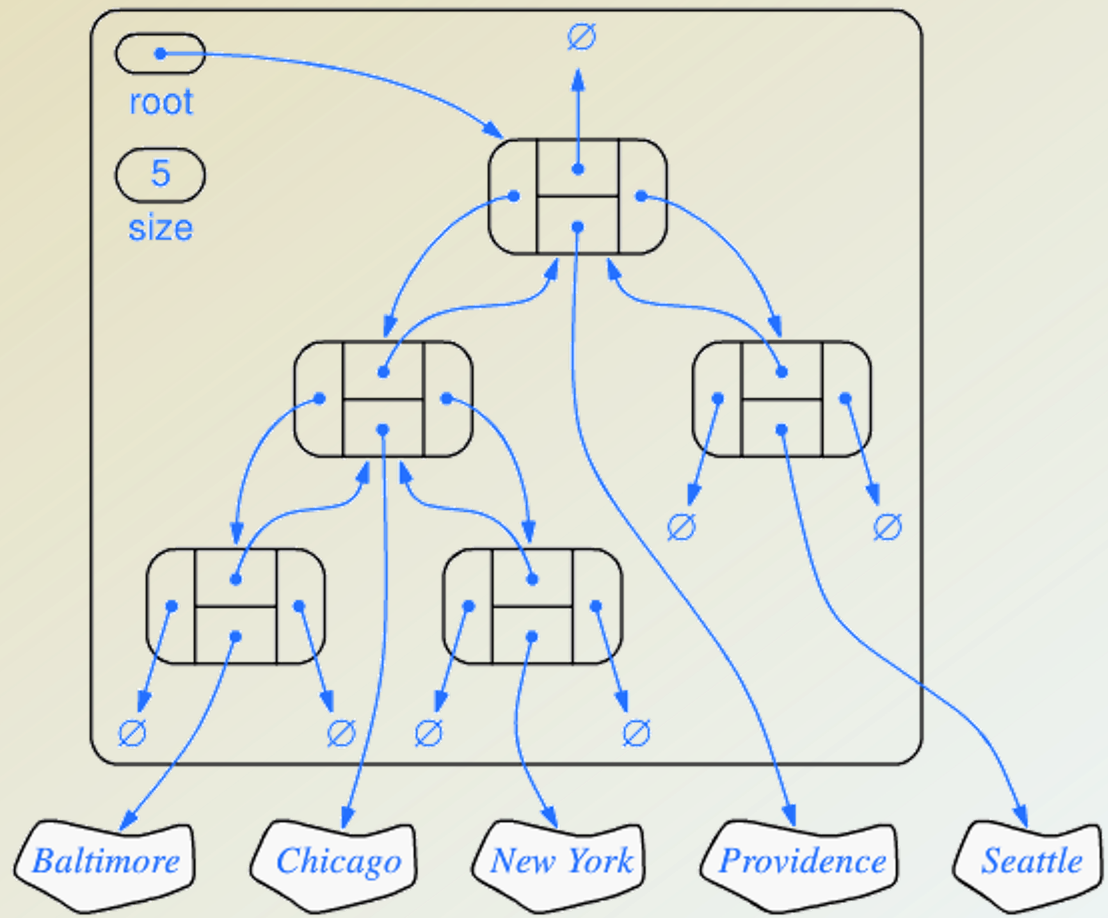
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT





Binary Tree - Example





Implementation of the Linked Binary Tree Structure

- **addRoot(e):** Create and return a new node r storing element e and make r the root of the tree; an error occurs if the tree is not empty.
- **insertLeft(v, e):** Create and return a new node w storing element e, add w as the the left child of v and return w; an error occurs if v already has a left child.
- **insertRight(v ,e):** Create and return a new node z storing element e, add z as the the right child of v and return z; an error occurs if v already has a right child.
- **remove(v):** Remove node v, replace it with its child, if any, and return the element stored at v; an error occurs if v has two children.
- **attach(v, T1, T2):** Attach T1 and T2, respectively, as the left and right subtrees of the external node v; an error condition occurs ifv is not external.

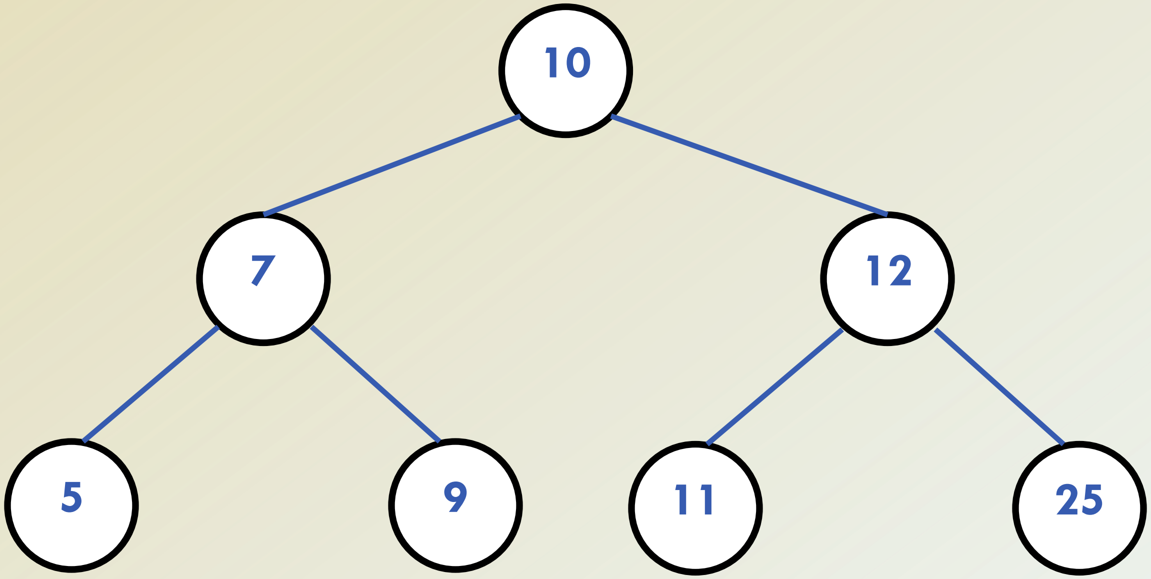
Binary Search Tree (BST)

- Binary trees are excellent data structures for searching large amounts of information.

- When used to facilitate searches, a binary tree is called a ***binary search tree***.

Binary Search Tree (BST)

- A binary search tree (BST) is a binary tree in which:
  - Elements in left subtree are smaller than the current node.
  - Elements in right subtree are greater than the current node.



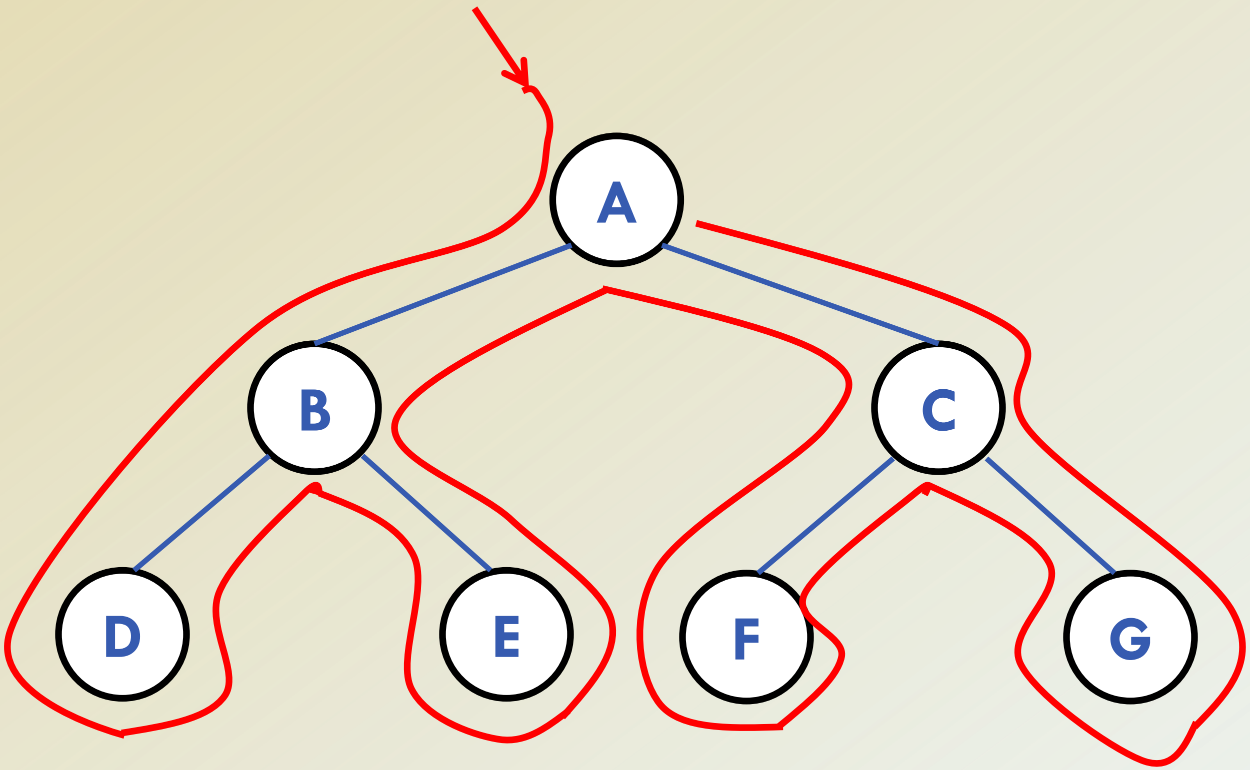
Traversing the tree

- There are three common methods for traversing a binary tree and processing the value of each node:
  - ***Pre-order***
  - ***In-order***
  - ***Post-order***

- Each of these methods is best implemented as a recursive function.

Tree Traversal (Pre-order)

- Pre-order: Node ⇨ Left ⇨ Right



A B D E C F G

Exercise: Pre-order traversal

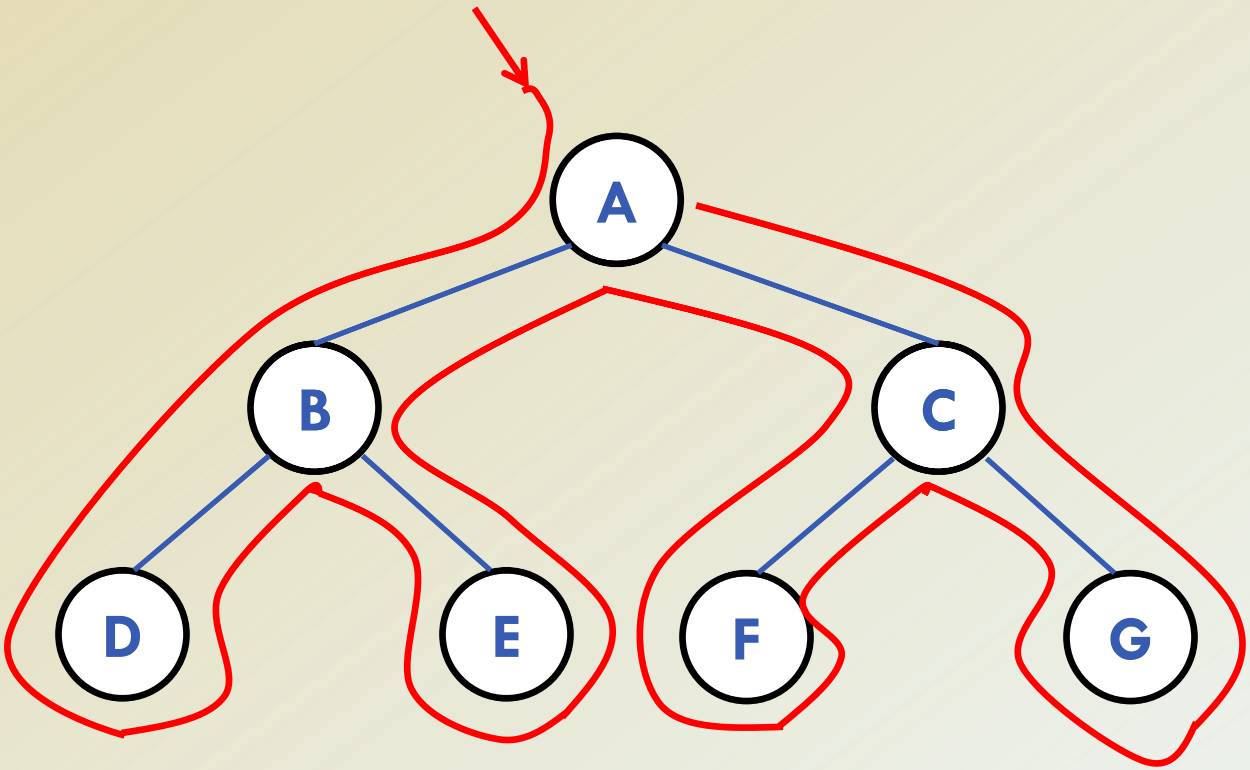
- Insert the following items into a binary search tree.

  50, 25, 75, 12, 30, 67, 88, 6, 13, 65, 68

- Draw the binary tree and print the items using **Pre-order** traversal.

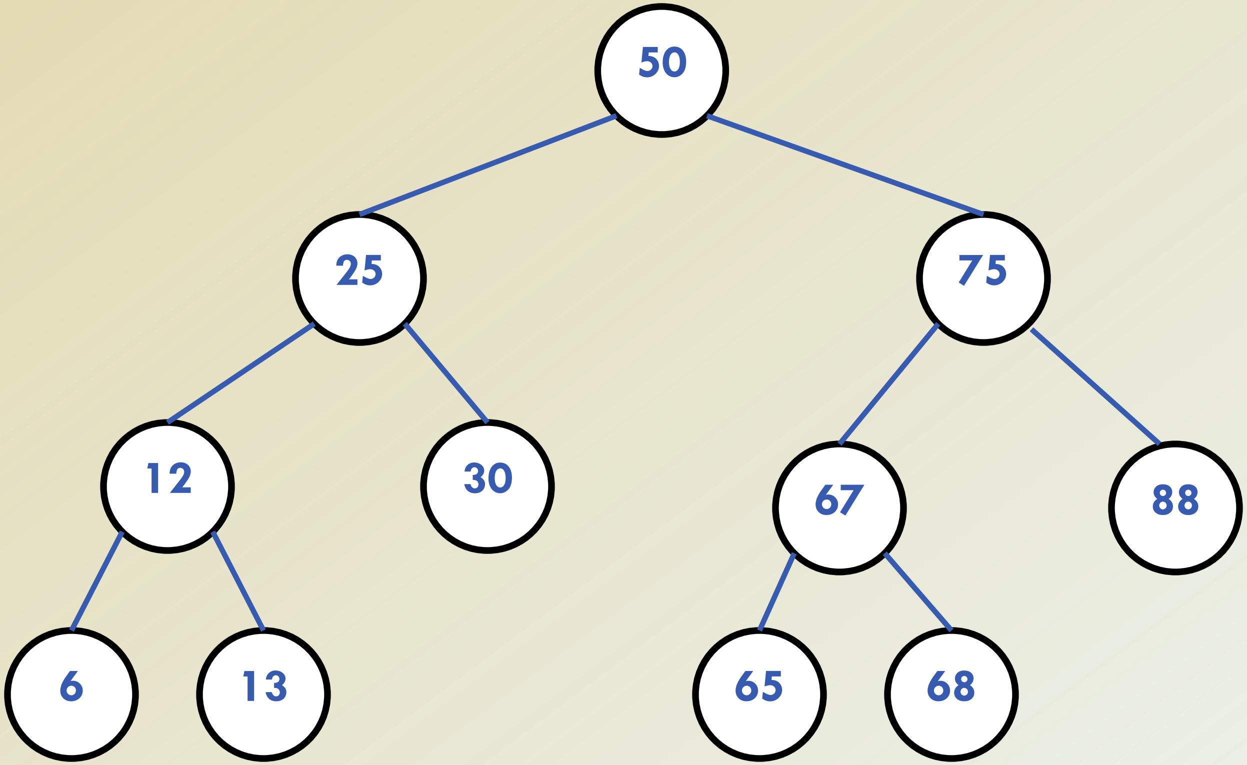
Tree Traversal (In-order)

- In-order: Left ⇨ Node ⇨ Right



D B E A F C G

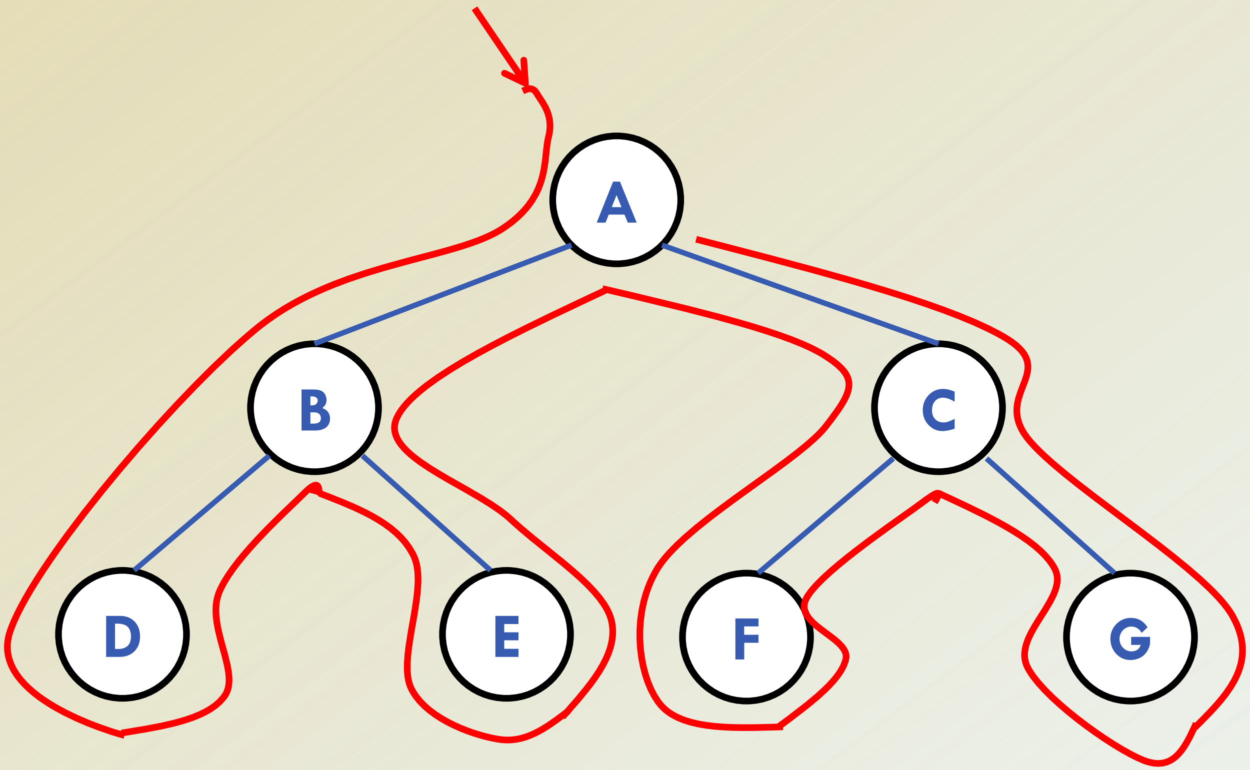
Exercise: In-order traversal



- From the previous exercise, print the tree's nodes using In-order traversal.

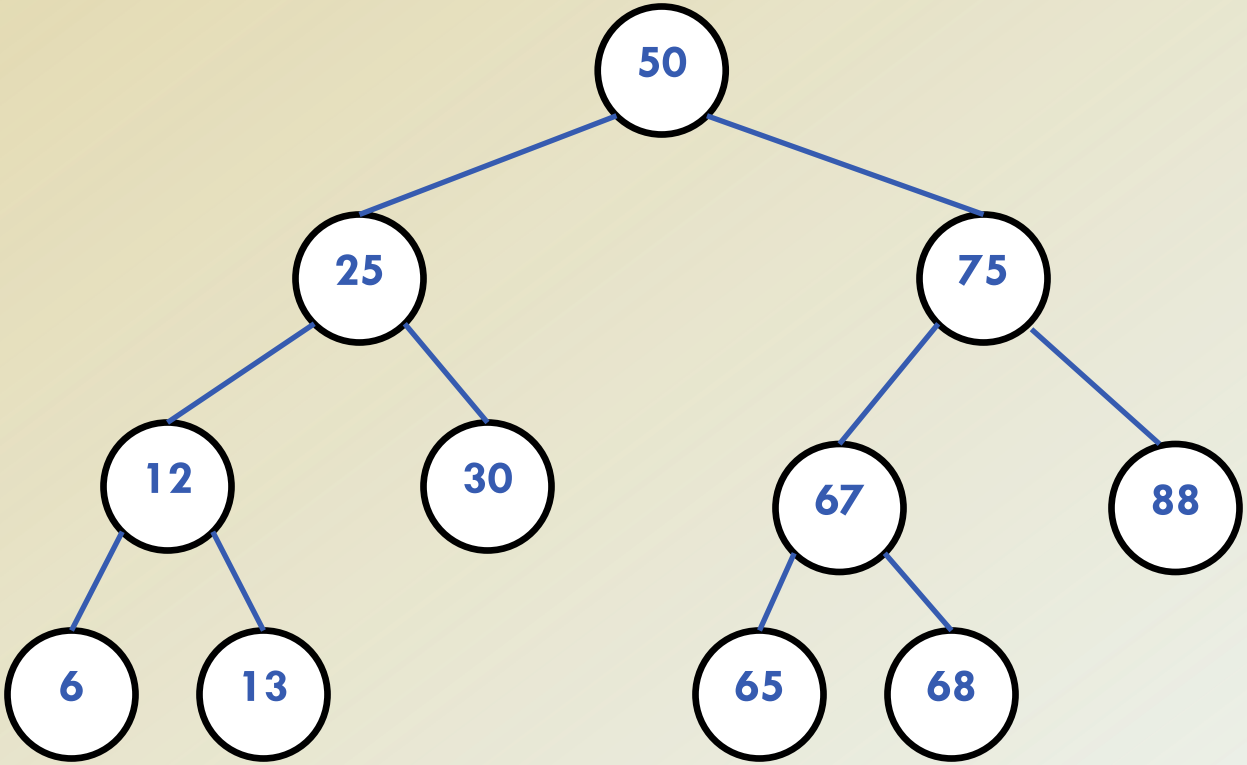
Tree Traversal (Post-order)

- Post-order: Left ⇨ Right ⇨ Node



D E B F G C A

Exercise: Post-order traversal

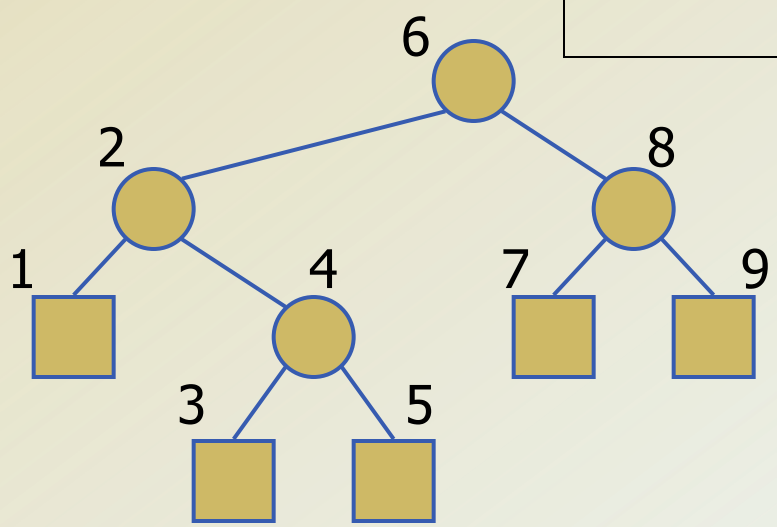


- From the previous exercise, print the tree's nodes using Post-order traversal.

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

```
Algorithm inOrder(v)
    if hasLeft (v)
        inOrder (left (v))
    visit(v)
    if hasRight (v)
        inOrder (right (v))
```



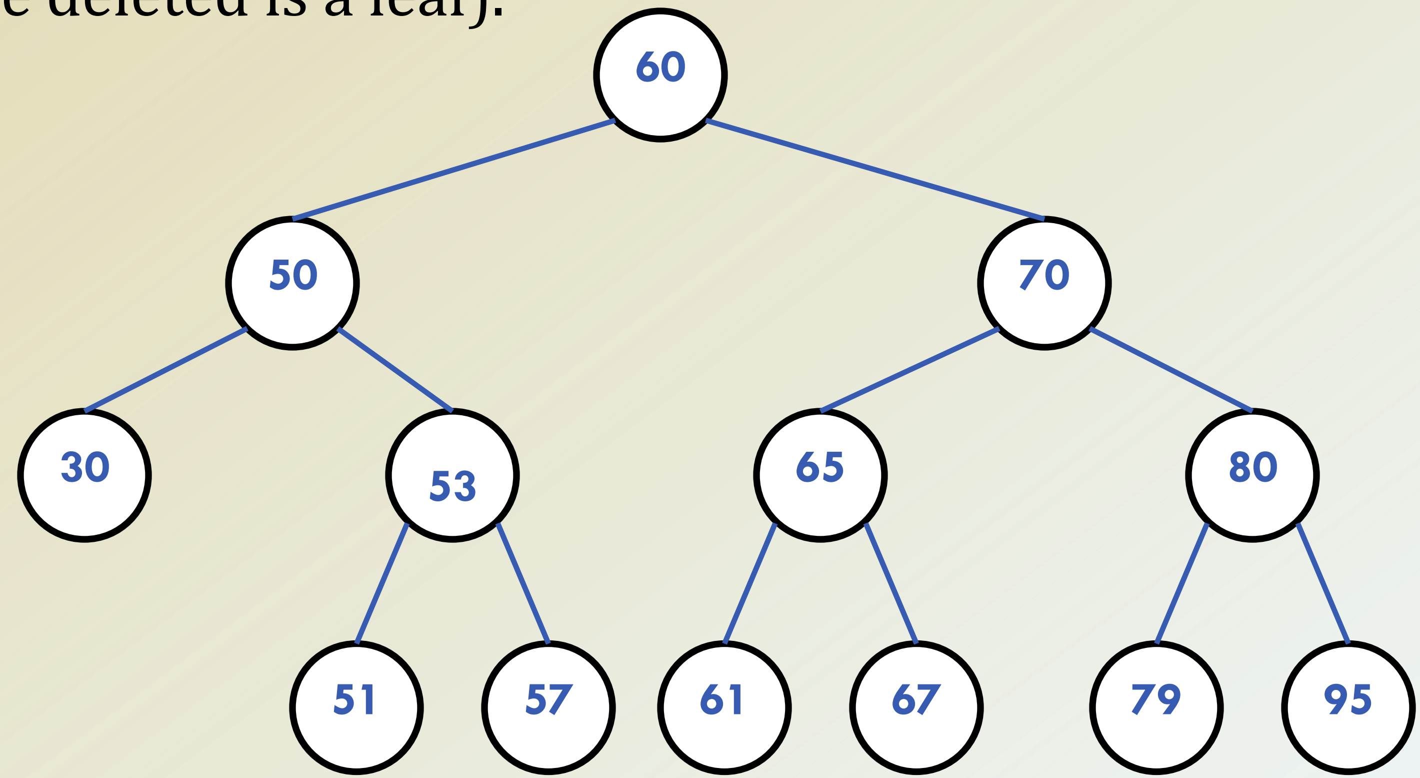
Delete a node

- After deleting an item, the resulting binary tree must be a binary search tree.
  1. Find the node to be deleted.
  2. Delete the node from the tree.

Delete (Case 1)

- The node to be deleted has no left and right subtree (the node to be deleted is a leaf).

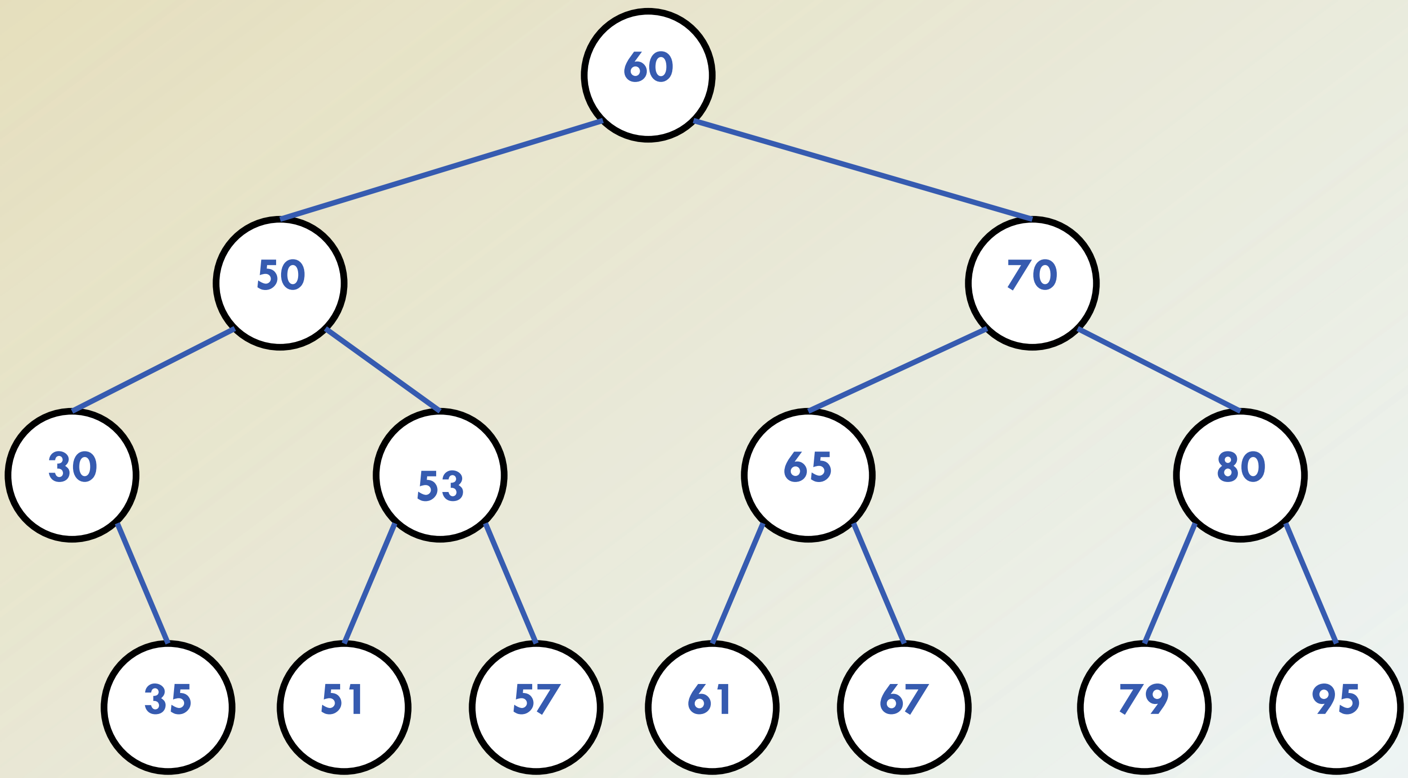
delete(30)



Delete (Case 2)

- The node to be deleted has no left subtree (the left subtree is empty but it has a nonempty right subtree).

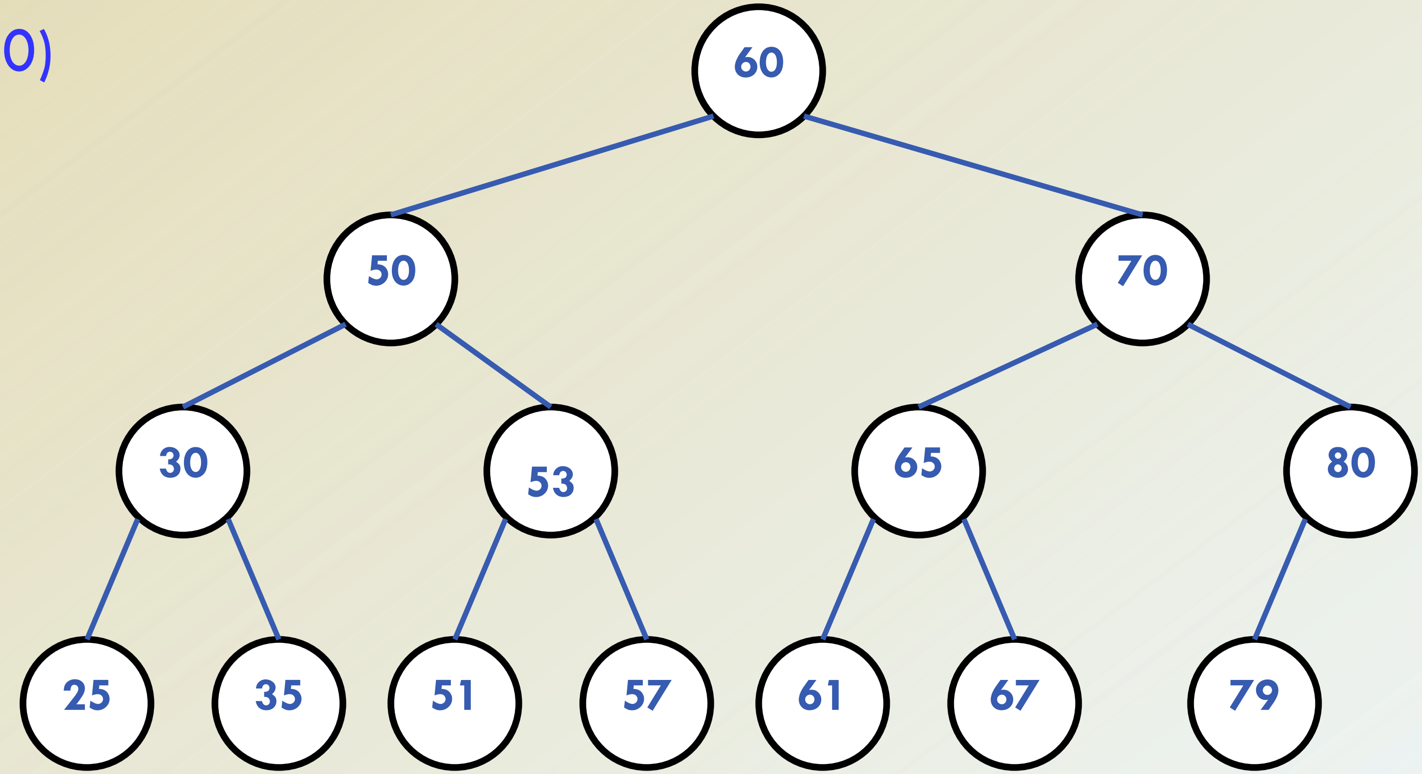
delete(30)



Delete (Case 3)

- The node to be deleted has no right subtree (the right subtree is empty but it has a nonempty left subtree).

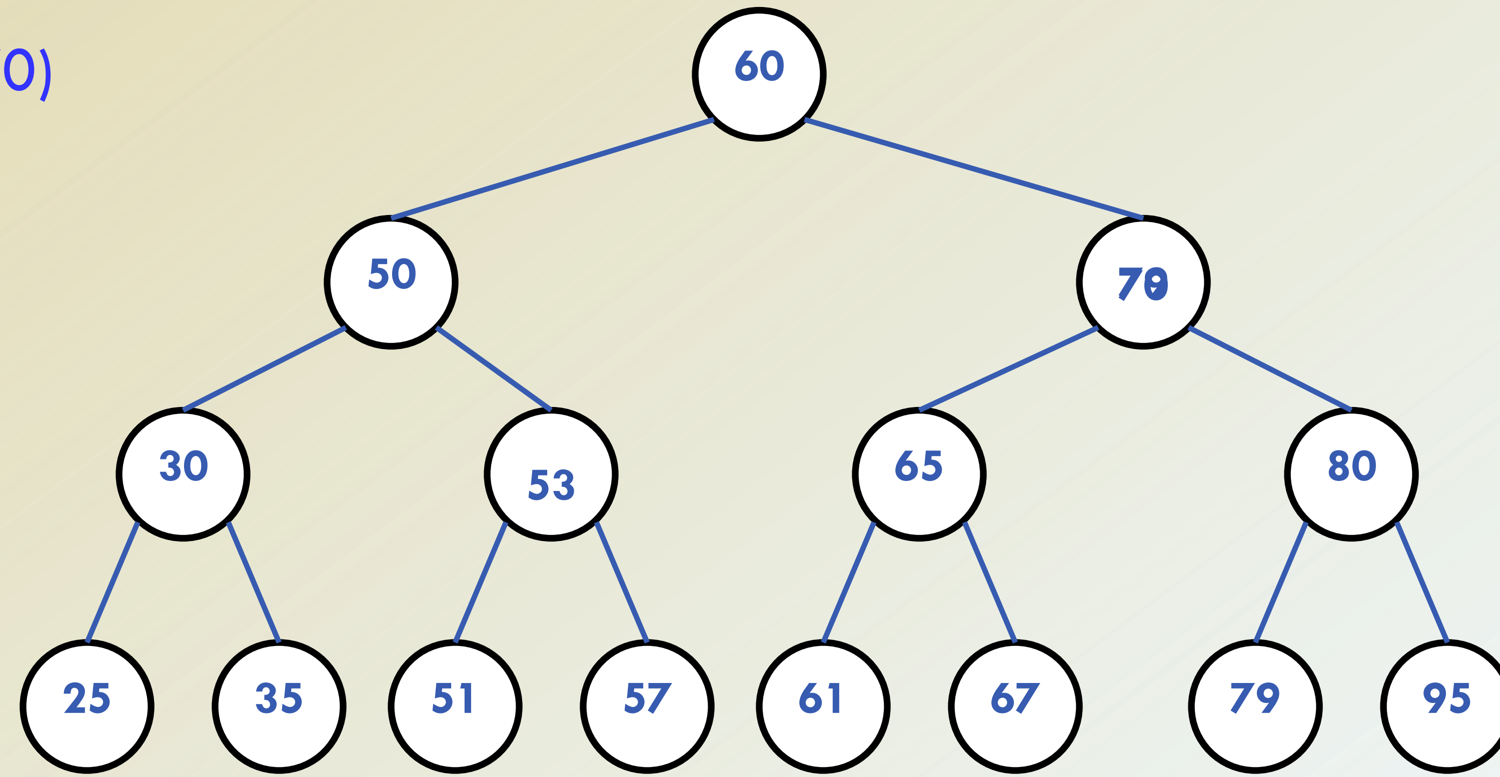
delete(80)



Delete (Case 4)

- The node to be deleted has nonempty left and right subtree.

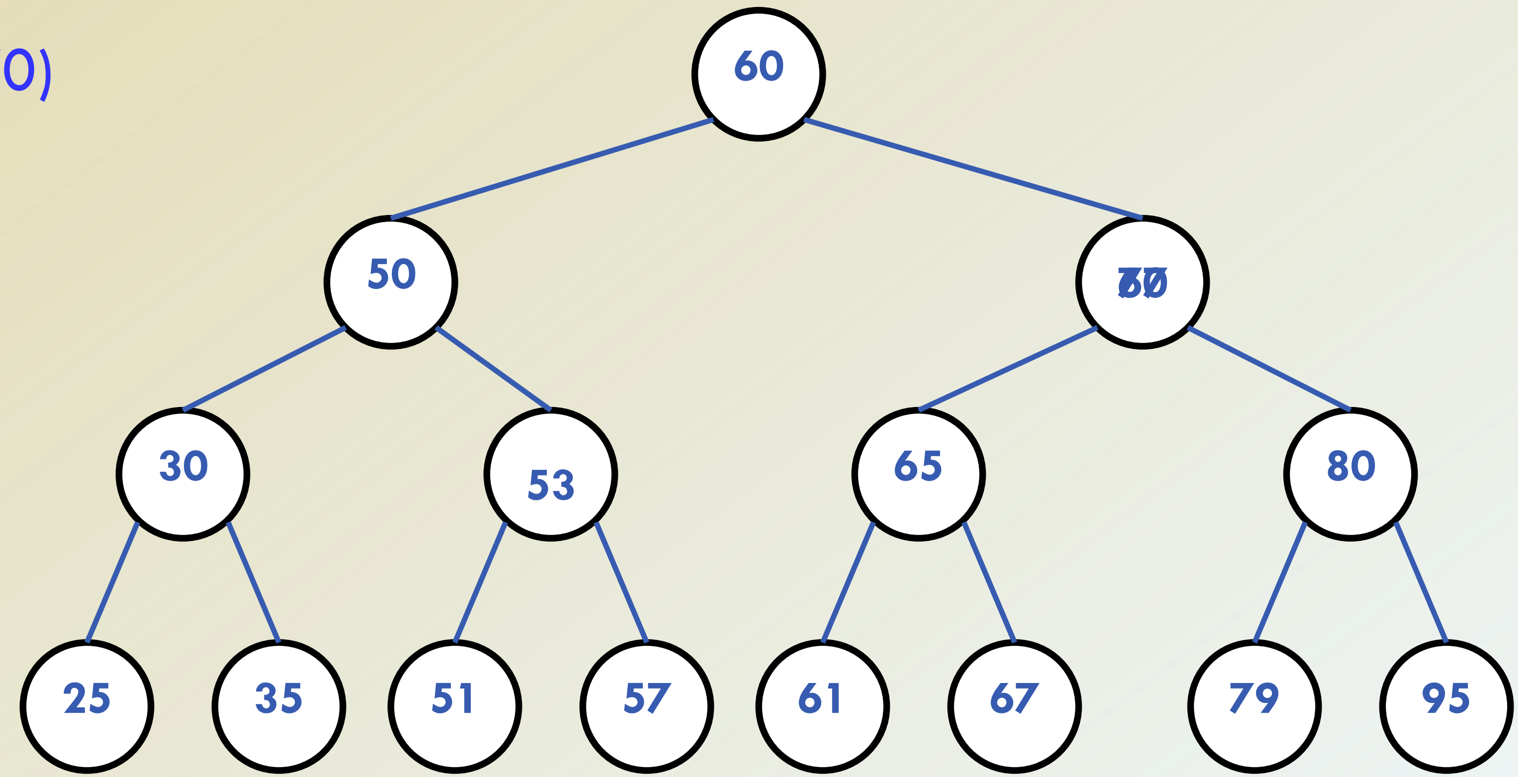
delete(70)



Delete (Case 4)

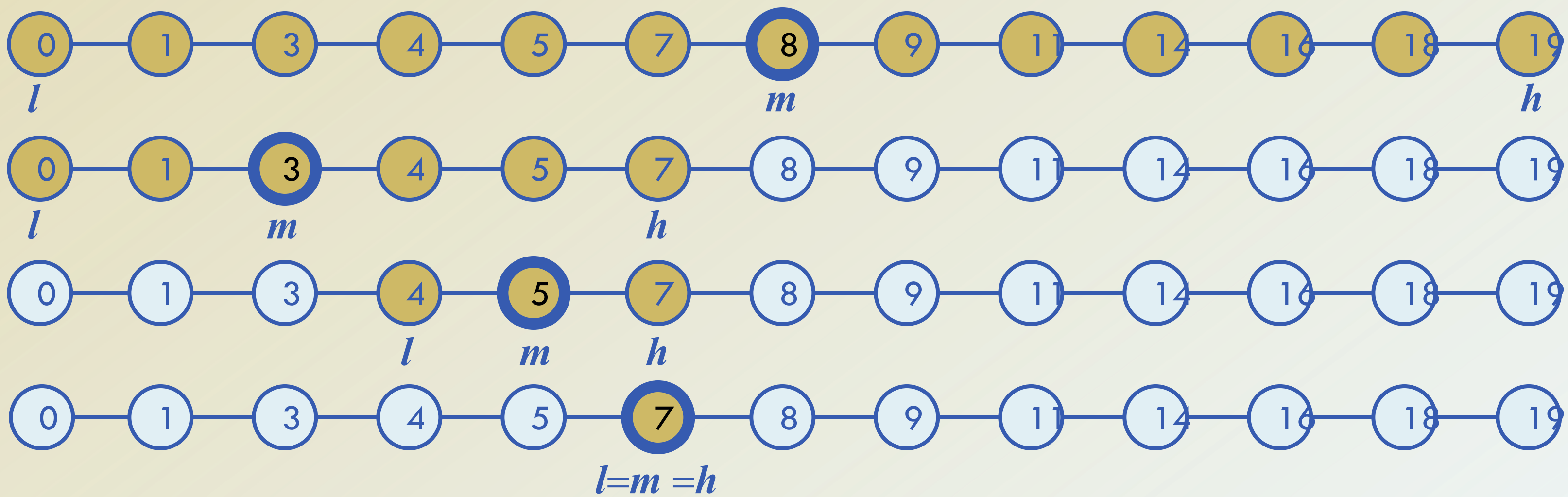
- The node to be deleted has nonempty left and right subtree.

delete(70)



Binary Search

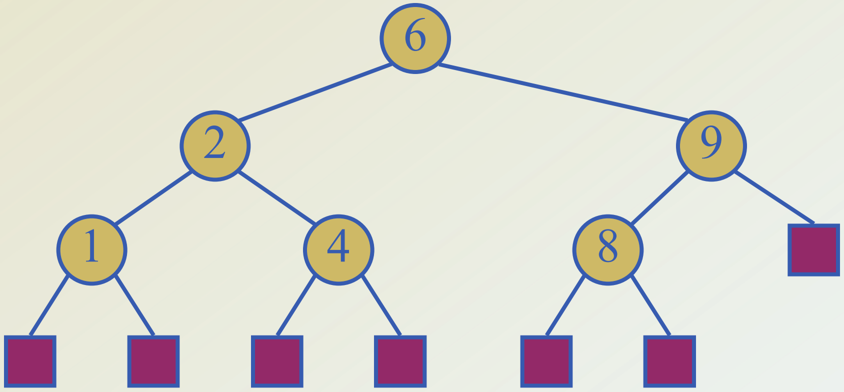
- Binary search can perform operations get, floorEntry and ceilingEntry on an ordered map implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- **Example:** find(7)



Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***. We have $\mathit{key}(u) \leq \mathit{key}(v) \leq \mathit{key}(w)$
- External nodes do not store items.

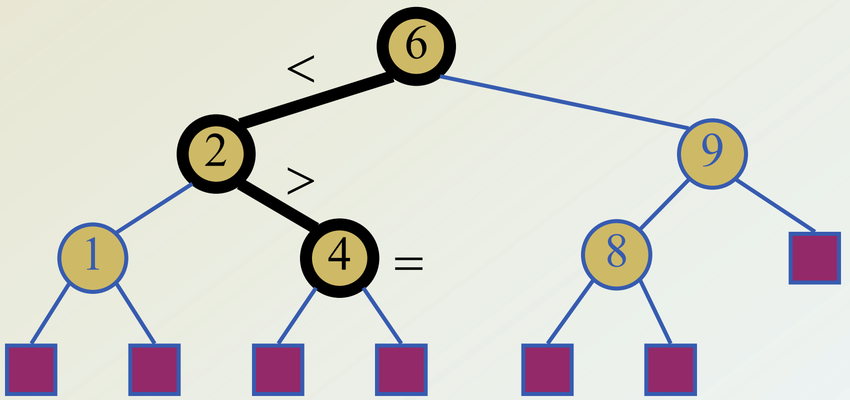
- An inorder traversal of a binary search trees visits the keys in increasing order.



Search

- To search for a key ***k***, we trace a downward path starting at the root.
- The next node visited depends on the comparison of ***k*** with the key of the current node.
- If we reach a leaf, the key is not found.
- **Example:** get(4):
  - Call **TreeSearch(4,root)**

Algorithm ***TreeSearch***(***k***, ***v***)
    if *T.isExternal* (*v*)
        return *v*
    if *k* < *key*(*v*)
        return *TreeSearch*(*k*, *T.left*(*v*))
    else if *k* = *key*(*v*)
        return *v*
    else { *k* > *key*(*v*) }
        return *TreeSearch*(*k*, *T.right*(*v*))



End of Chapter 7