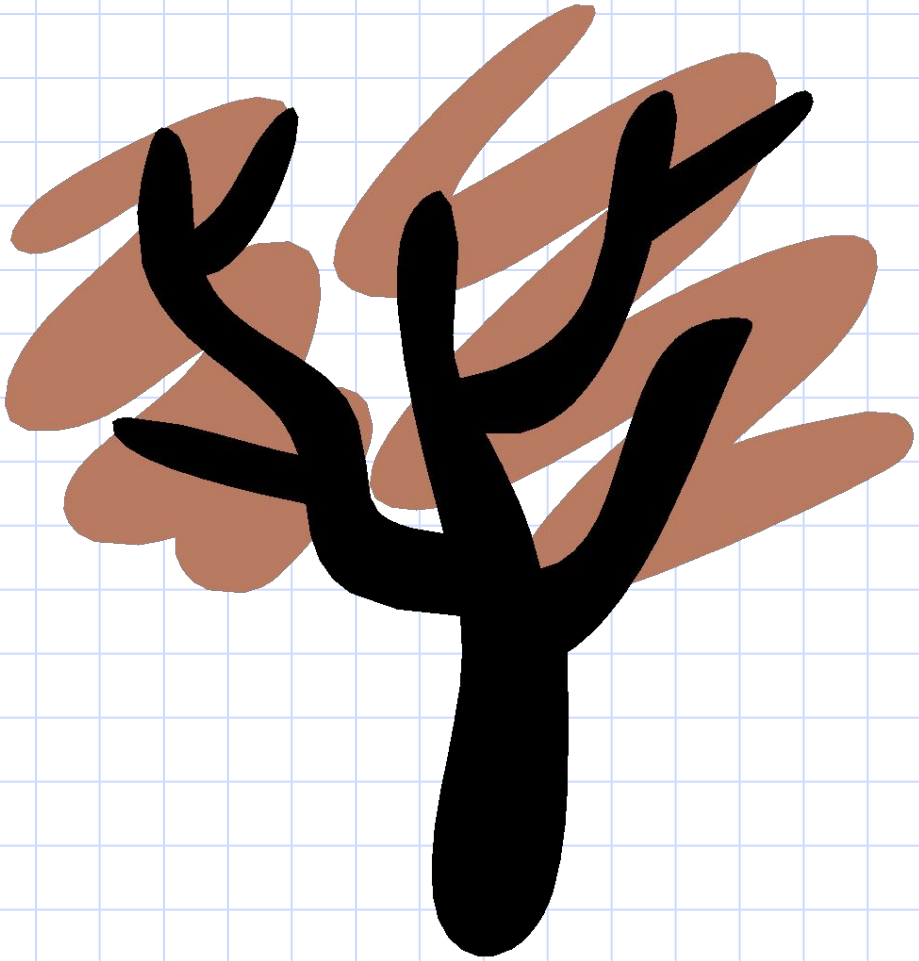
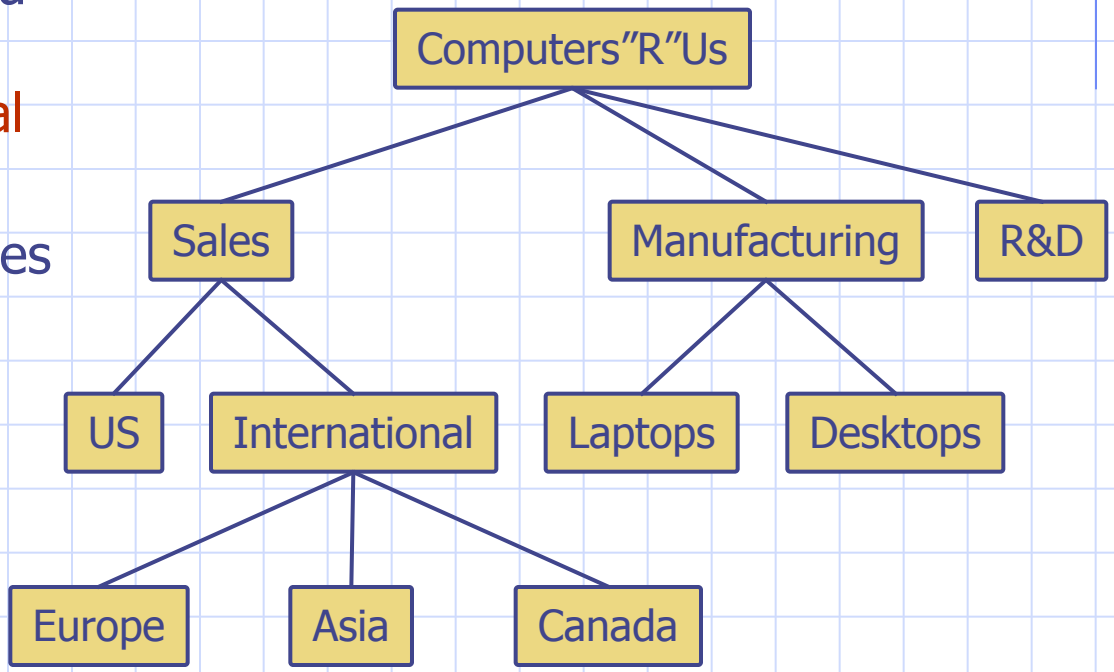
Tree



- Trees:
  - Definition of tree
  - Representation of tree
  - Types of tree
  - Binary tree traversal
  - Storage representation and manipulation of binary tree
  - Conversion of general tree to a binary tree
  - Other representation of tree, application to tree.
- Graphs :
  - Representation of graphs
  - Graph traversal and spanning forest.
  - Finding the shortest path (Warshall's Algorithm, Warshall's modified algorithm, Dijkstra's Technique)
  - Graph traversal (depth first search, breadth first search)

Trees

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



Tree: Recursive Definition

1. An empty structure is an empty tree
2. If $t_1, t_2, \ldots t_K$ are disjoint trees, then the structure whose root has, as its children, the roots of $t_1, t_2, \ldots t_K$ is also a tree.
3. Only structures generated by rules 1 and 2 are trees.

Terminology for a Tree

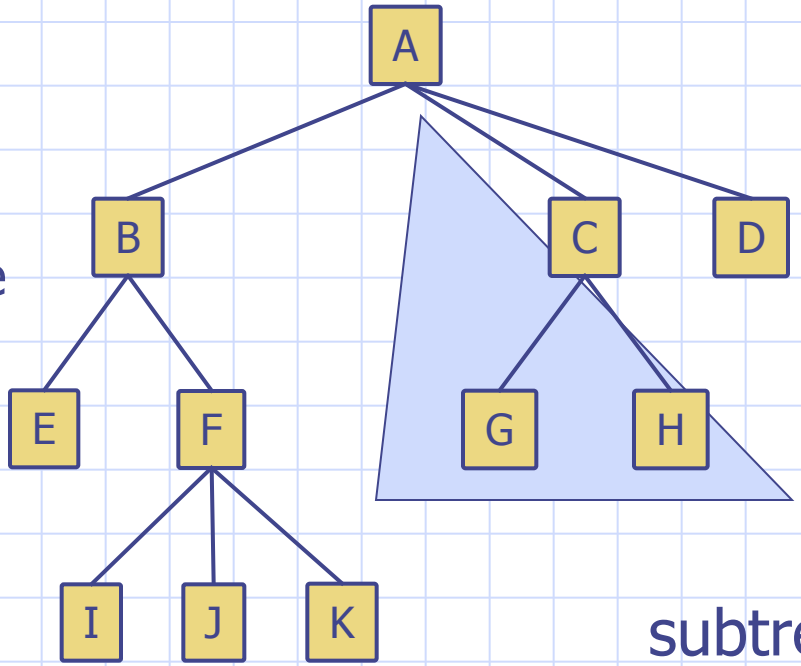
- **Tree:** A collection of data whose entries have a hierarchical organization
- **Node:** An entry in a tree
- **Root node:** The node at the top
- **Terminal** or **leaf node:** A node at the bottom
- **Parent:** The node immediately above a specified node
- **Child:** A node immediately below a specified node
- **Ancestor:** Parent, parent of parent, etc.
- **Descendent:** Child, child of child, etc.
- **Siblings:** Nodes sharing a common parent

Terminology for a Tree

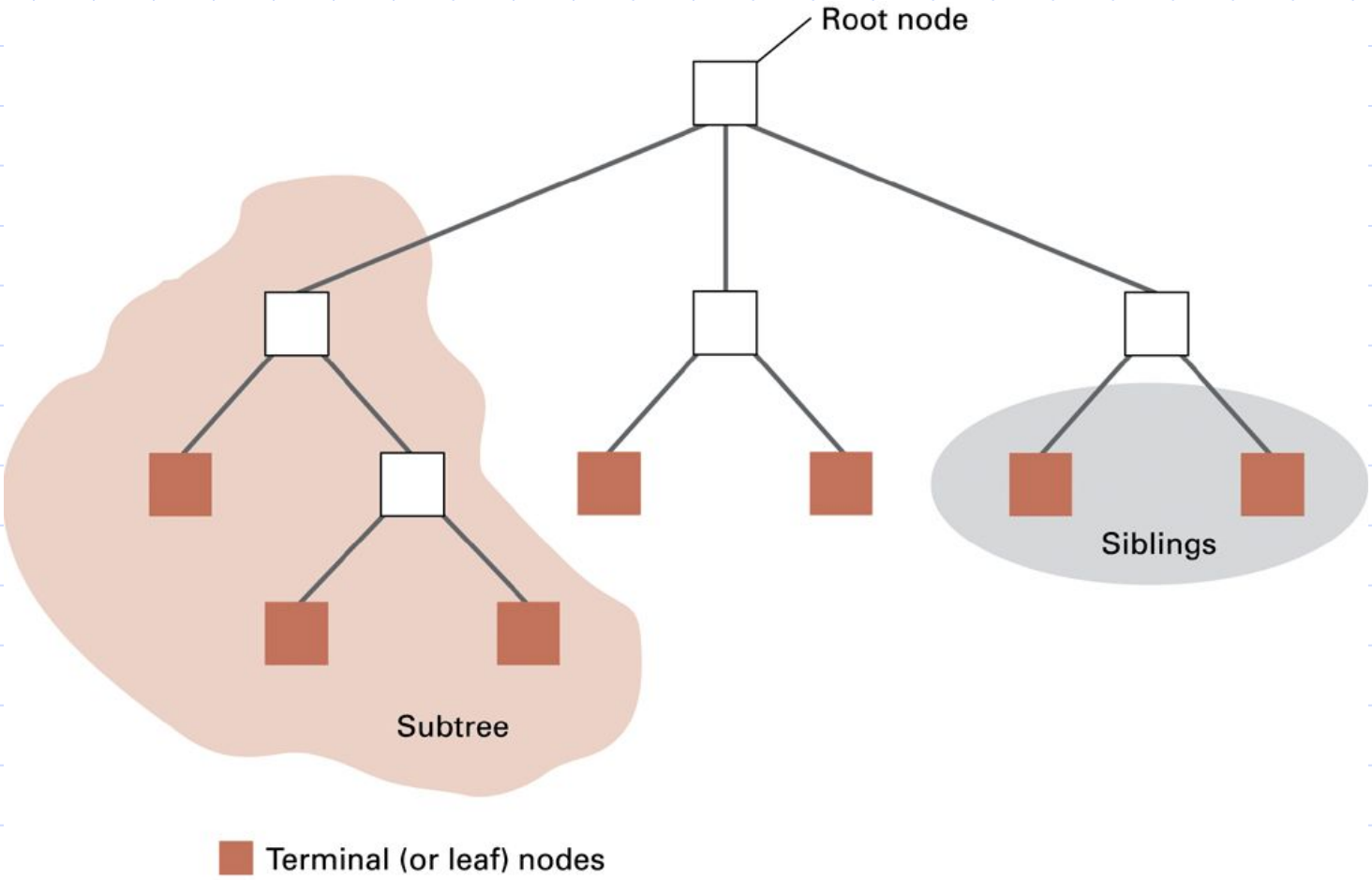
- **Binary tree:** A tree in which every node has at most two children
- **Depth:** The number of nodes in longest path from root to leaf
- **Depth of a node**: number of ancestors
- **Height of a tree**: maximum depth of any node
- **Subtree**: tree consisting of a node and its descendants

Tree Terminology

- Internal node: node with at least one child (A, B, C, F)
- External node (leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Subtree: tree consisting of a node and its descendants

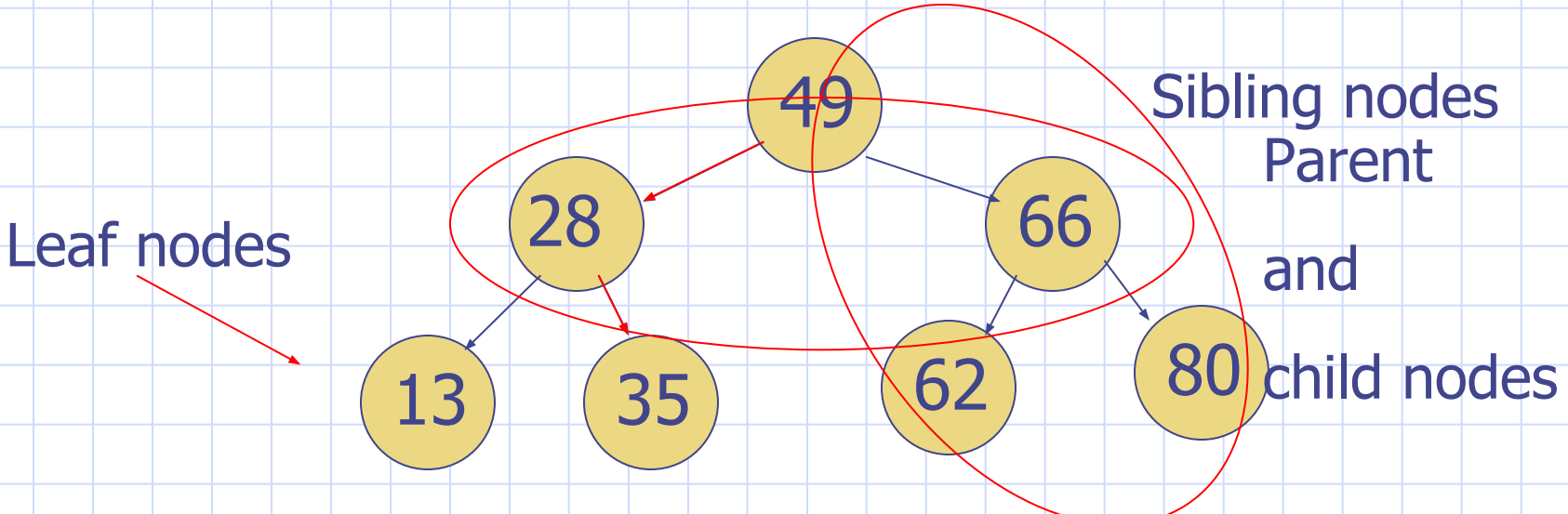


Tree terminology



Tree Terminology

- A tree consists of:
  - finite collection of nodes
  - non empty tree has a root node
  - root node has no incoming links
  - every other node in the tree can be reached from the root by unique sequence of links



Applications of Trees

- Genealogical tree
  - pictures a person's descendants and ancestors
- Game trees
  - shows configurations possible in a game such as the Towers of Hanoi problem
- Parse trees
  - used by compiler to check syntax and meaning of expressions such as 2 * (3 + 4)

General Tree

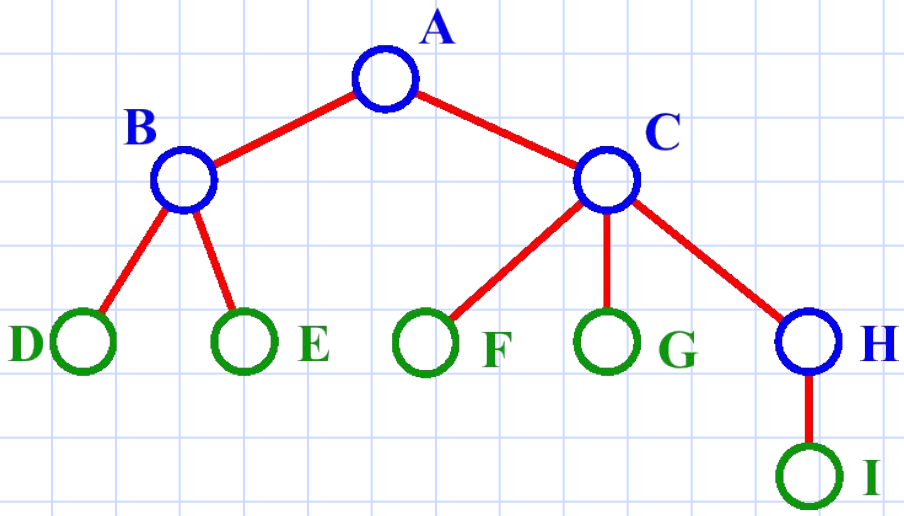
- Defnied as non empty finite set of elements called nodes such that
  - Tree contains the root lement
  - The remaining elements of the tree form an ordered collection of zero or more disjoint trees, t_1, t_2,..... t_m

t_1, t_2,..... t_m subtrees of root.

The roots of t_1, t_2,..... t_m are called successors of the root.

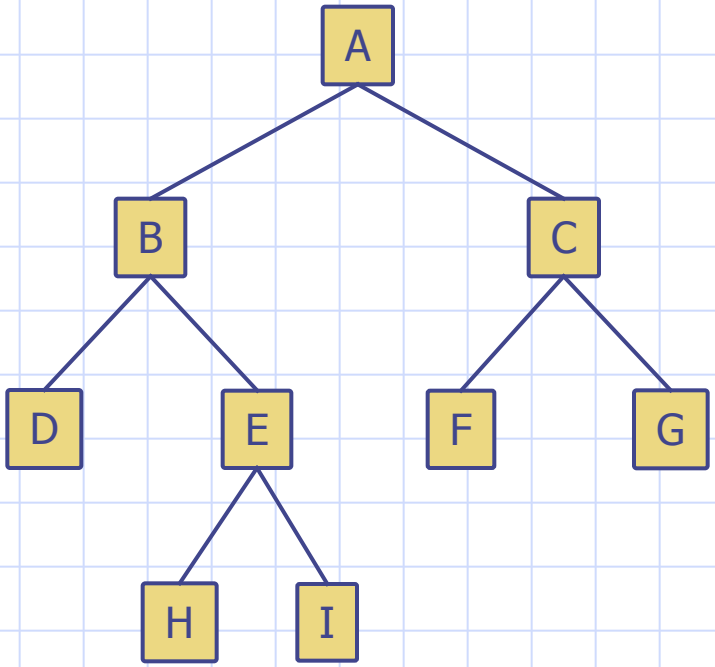
Trees

- *A, B, C, H* are *internal nodes*
- The *depth (level)* of *E* is *2*
- The *height* of the tree is *3*
- The *degree* of node *B* is *2*



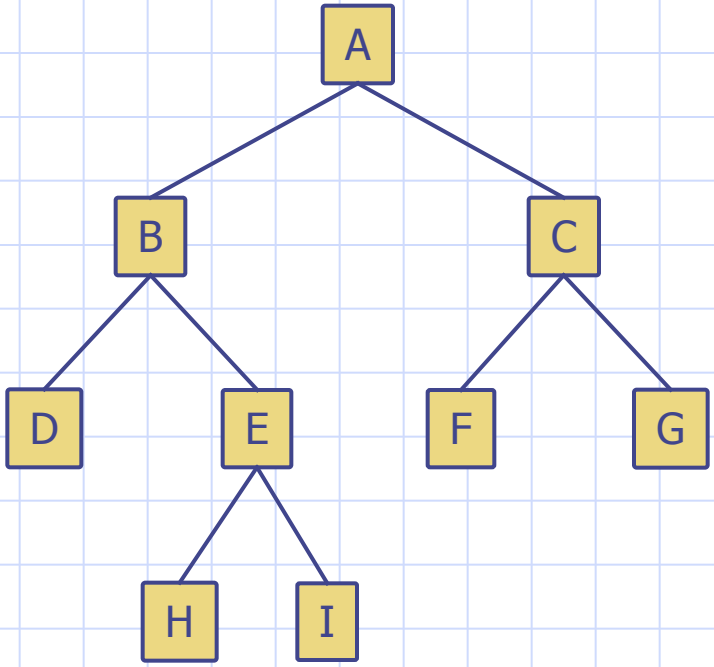
Binary Tree

- A binary tree is a tree with the following properties:
  - Each internal node has two children
  - 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



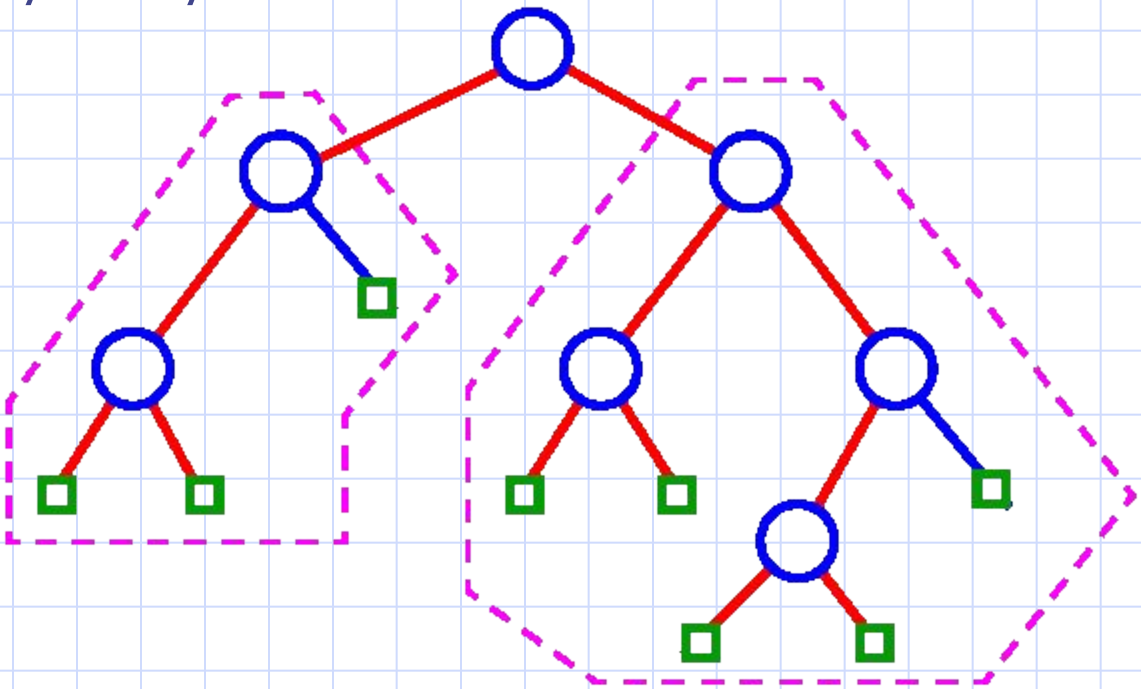
Extended Binary Tree

- A binary tree can be converted to an extended binary tree by adding new nodes that have only one child
- New nodes are added in such a way that all the nodes in resultant tree have either zero or 2 children.
- Also know as 2-Tree
- Original tree nodes are internal, added are external



Binary Tree

- If a tree has n nodes, number of branches is n-1
- Only one parent to each node except root node
- A single path connects any two nodes
- For a binary tree of height h, the maximum number of nodes can be $2^{n+1} - 1$
- Any binary tree with n internal nodes has n+1 external nodes



Applications of Binary Tree

- Searching algorithm
- arithmetic expressions
- decision processes
- Populating voluminous, relational, hierarchical data into memory
- Expression evaluation
- AI

Binary Tree Creation: Recursive Algorithm

Create_Tree (Info, Node)

Step 1: [Check whether the tree is empty]

If Node = NULL

Node = Create a node

Left_child[Node] = NULL

Right_child[Node] = NULL

Step 2 : [Test for the left child]

If Info[Node] >= Info

Left_child[Node] = Call Create_Tree (Info, Left_Child[Node])

Else

Right_child[Node] = Call Create_Tree (Info, Right_Child[Node])

Step 3:

Return (Node)

Example: Expression Trees

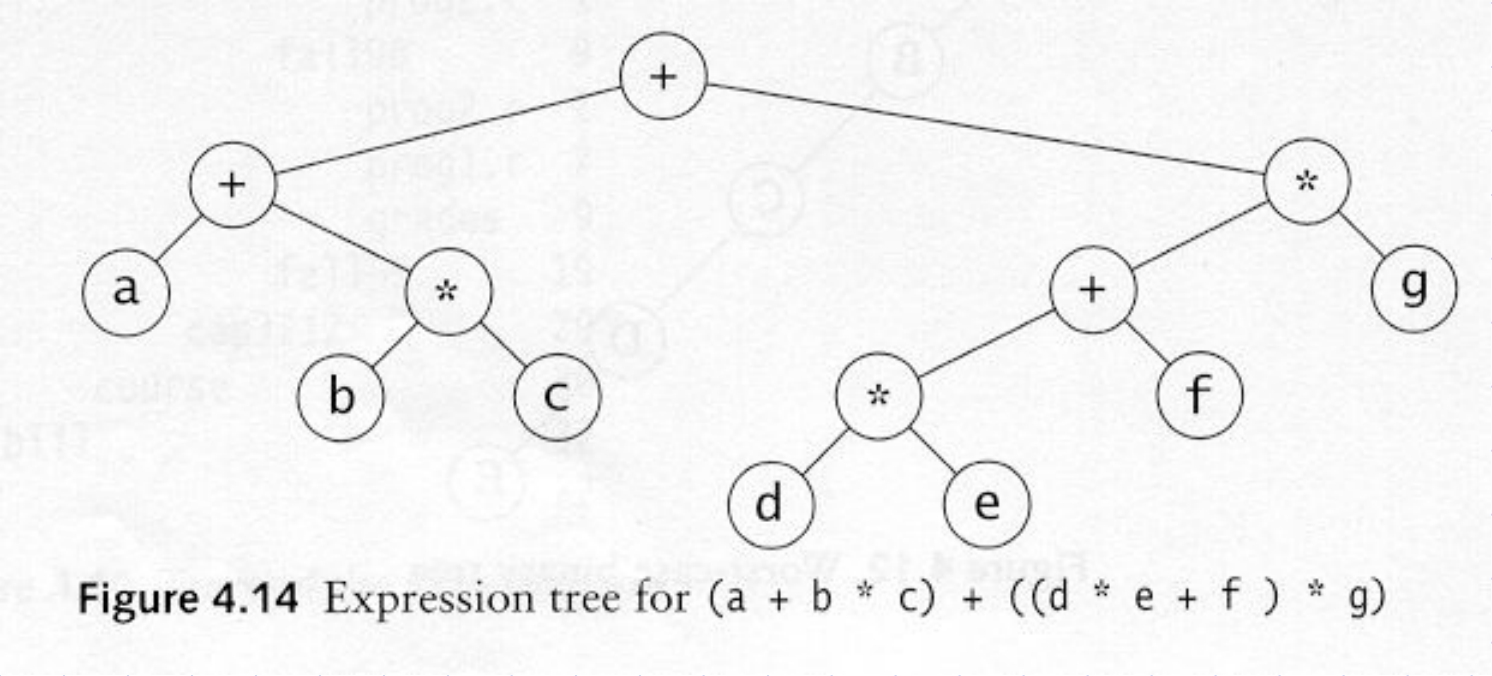


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The internal nodes contain operators
- Will not be a binary tree if some operators are not binary

Binary Tree Traversal

- Traversal is the process of visiting every node once
- Visiting a node entails doing some processing at that node, but when describing a traversal strategy, we need not concern ourselves with what that processing is

Binary Tree Traversal Techniques

- Three recursive techniques for binary tree traversal
- In each technique, the left subtree is traversed recursively, the right subtree is traversed recursively, and the root is visited
- What distinguishes the techniques from one another is the order of those 3 tasks

Tree Traversal

- Used to print out the data in a tree in a certain order
- Pre-order traversal
  - Print the data at the root
  - Recursively print out all data in the leftmost subtree
  - …
  - Recursively print out all data in the rightmost subtree

Preoder, Inorder, Postorder

- In Preorder, the root is visited before (pre) the subtrees traversals
- In Inorder, the root is visited in-between left and right subtree traversa
- In Preorder, the root is visited after (pre) the subtrees traversals

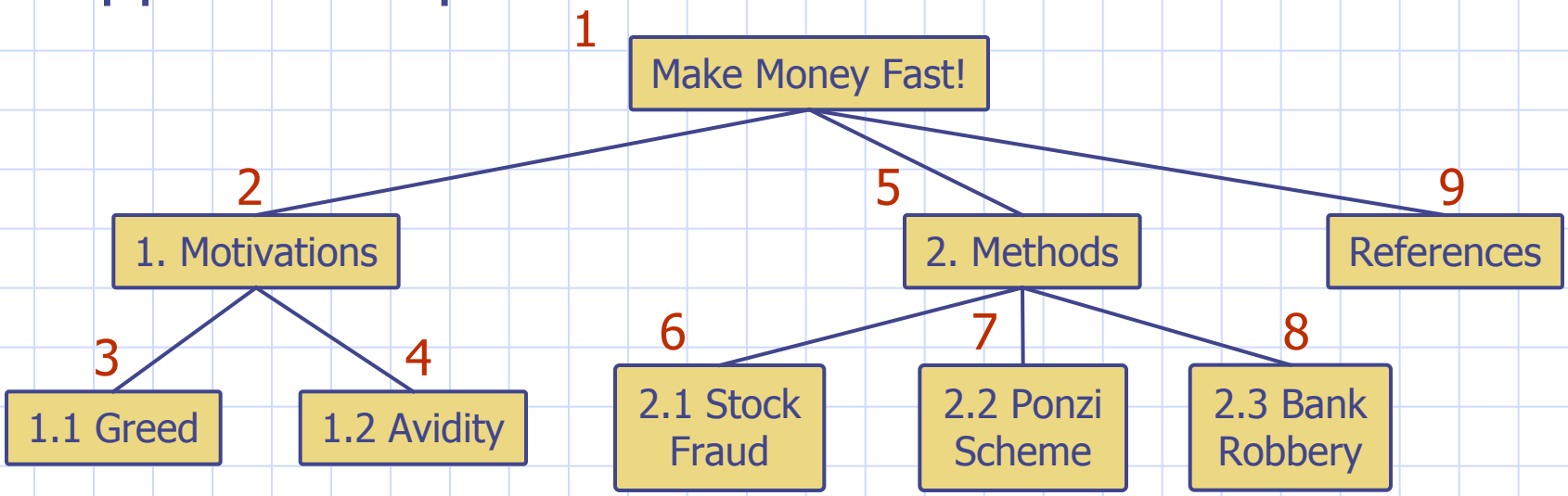
Preorder Traversal:
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

Inorder Traversal:
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

Postorder Traversal:
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

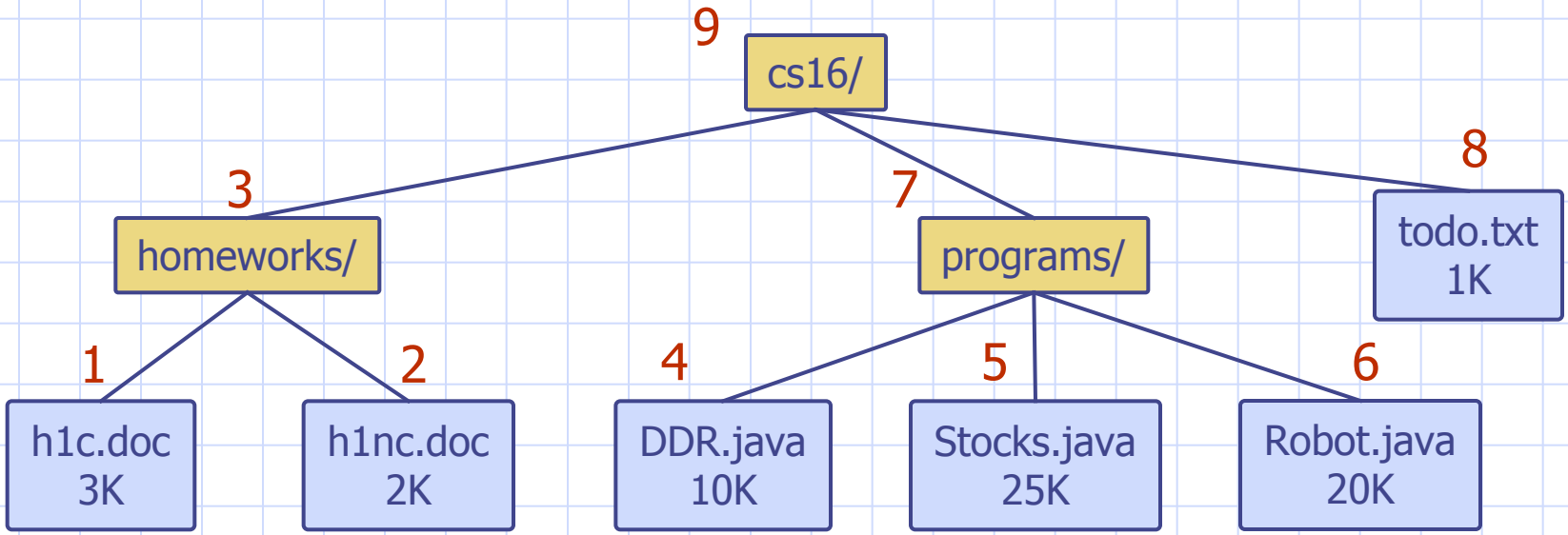
Preorder Traversal

- 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



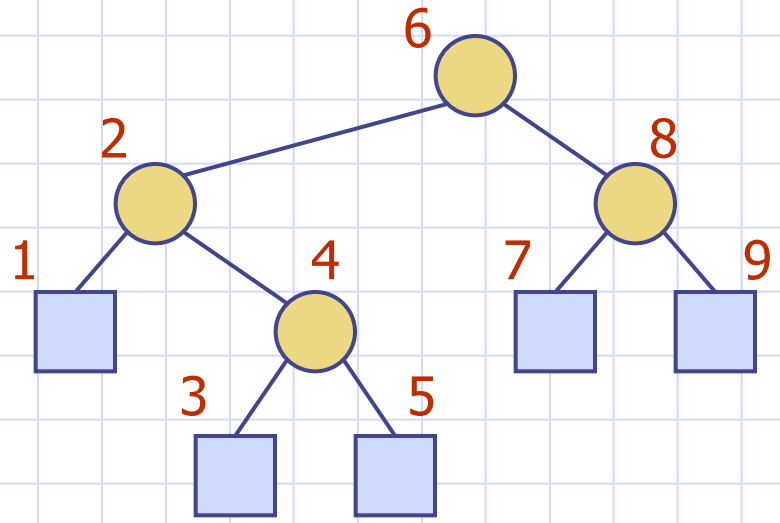
Postorder Traversal

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



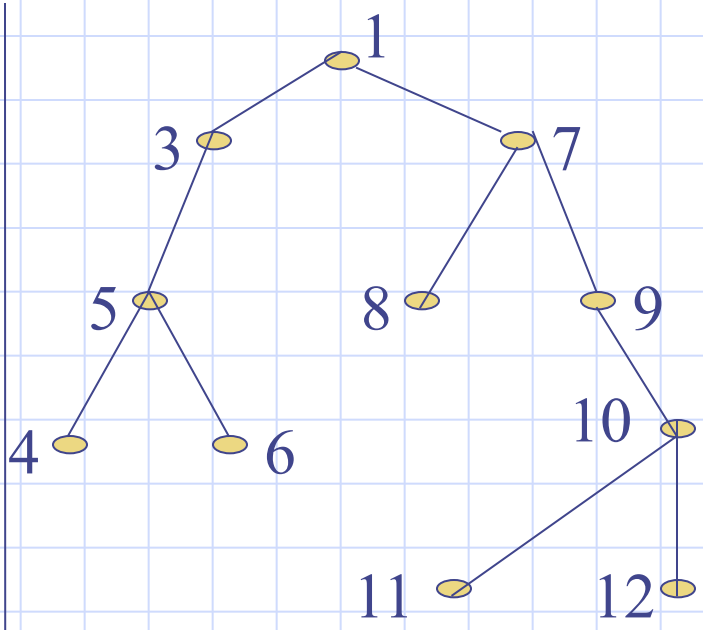
Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree



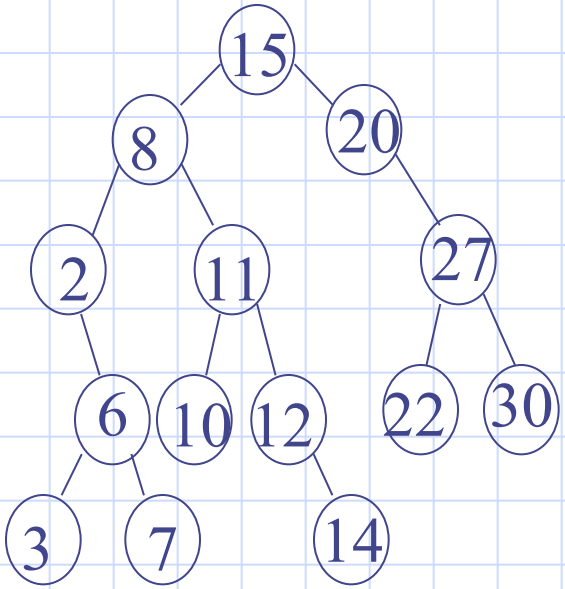
Illustrations for Traversals

- Assume: visiting a node is printing its label
- Preorder:
  1 3 5 4 6 7 8 9 10 11 12
- Inorder:
  4 5 6 3 1 8 7 9 11 10 12
- Postorder:
  4 6 5 3 8 11 12 10 9 7 1



Illustrations for Traversals (Contd.)

- Assume: visiting a node is printing its data
- Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30
- Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30
- Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15



Preorder Traversing: Algorithm

Preorder(Node)

Step 1: [Do through step 3]

If Node # NULL

Output Info[Node]

Step 2: Call Preorder(Left_Child[Node])

Step 3: Call Preorder(Right_Child[Node])

Step 4: Exit

Inorder Traversing: Algorithm

Inorder(Node)

Step 1: [Do through step 4]

If Node # NULL

Step 2: Call Inorder(Left_Child[Node])

Step 3: Output Info[Node]

Step 4: Call Inorder(Right_Child[Node])

Step 5: Exit

Postorder Traversing: Algorithm

Postorder(Node)

Step 1: [Do through step 4]

If Node # NULL

Step 2: Call Postorder(Left_Child[Node])

Step 3: Call Postorder(Right_Child[Node])

Step 4: Output Info[Node]

Step 5: Exit

Code for the Traversal Techniques

- The code for visit is up to you to provide, depending on the application
- A typical example for visit(...) is to print out the data part of its input node

```
void preOrder(Tree *tree){
    if (tree->isEmpty( ))       return;
    visit(tree->getRoot( ));
    preOrder(tree->getLeftSubtree());

preOrder(tree->getRightSubtree());
}
void inOrder(Tree *tree){
    if (tree->isEmpty( ))       return;
    inOrder(tree->getLeftSubtree( ));
    visit(tree->getRoot( ));
    inOrder(tree->getRightSubtree( ));
}
void postOrder(Tree *tree){
    if (tree->isEmpty( ))       return;
       postOrder(tree->getLeftSubtree(
));
     postOrder(tree->getRightSubtree(
));
    visit(tree->getRoot( ));
```

Preorder

- Preorder traversal
  - node, left, right
  - prefix expression
    - ++a*bc*+*defg

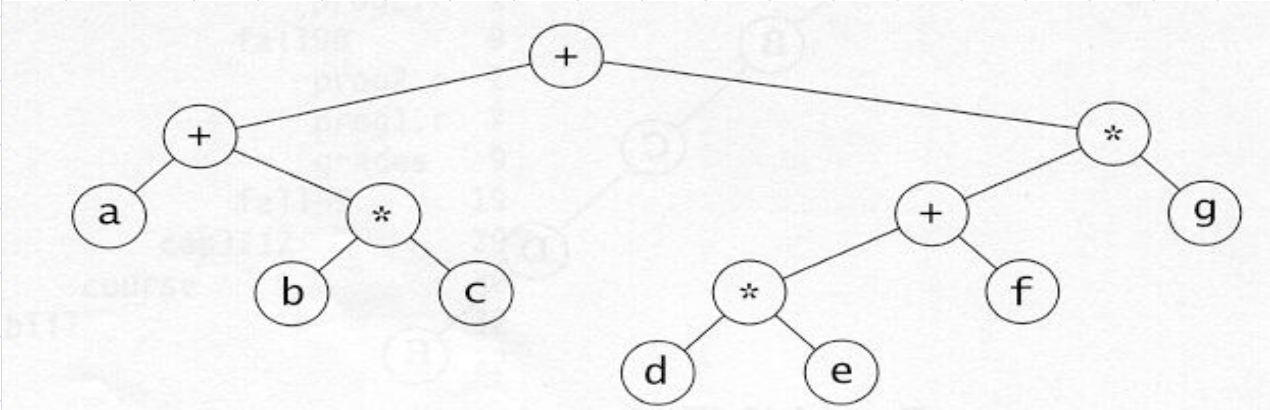


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder and Inorder

- Postorder traversal
  - left, right, node
  - postfix expression
    - abc*+de*f+g*+
- Inorder traversal
  - left, node, right
  - infix expression
    - a+b*c+d*e+f*g

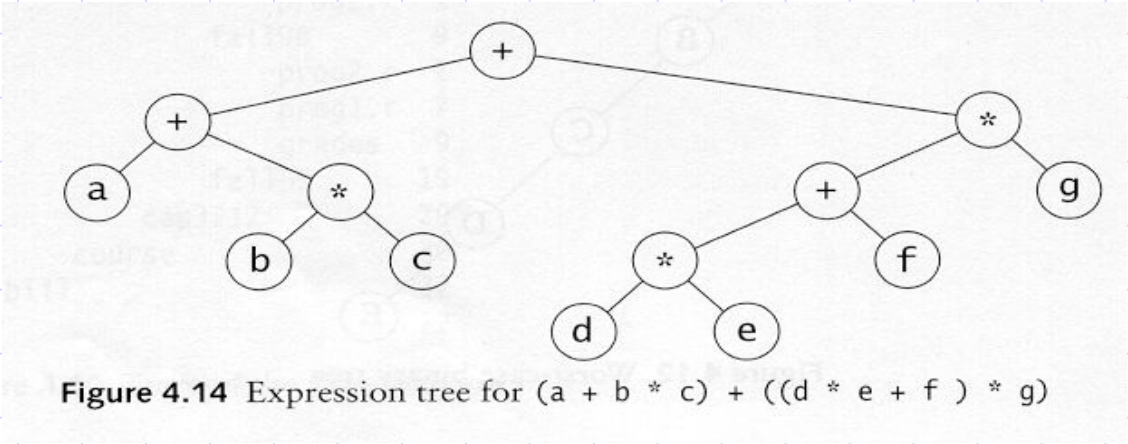
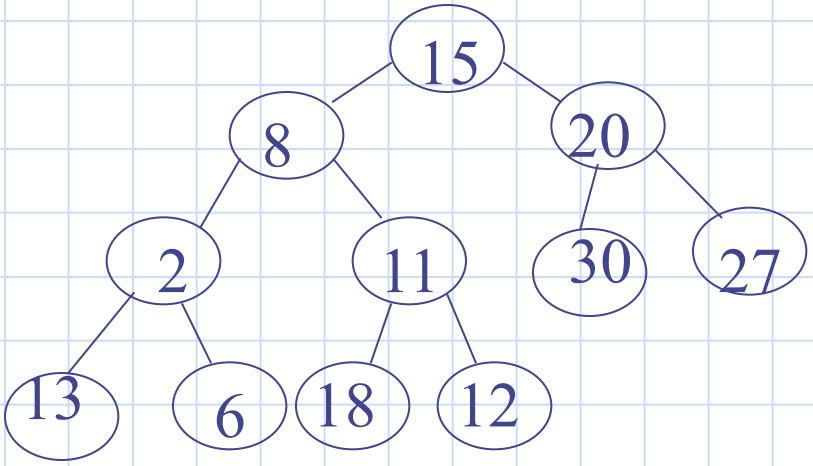


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Array Representation of Full Trees and Almost Complete Trees

- A canonically label-able tree, like full binary trees and almost complete binary trees, can be represented by an array A of the same length as the number of nodes
- A[k] is identified with node of label k
- That is, A[k] holds the data of node k
- Advantage:
  - no need to store left and right pointers in the nodes save memory
  - Direct access to nodes: to get to node k, access A[k]

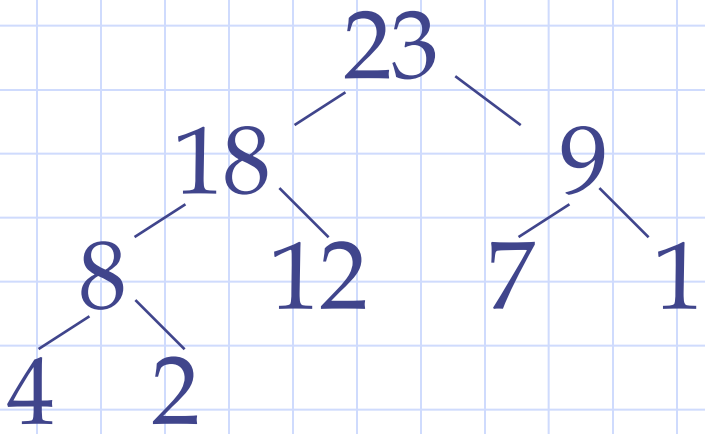
Illustration of Array Representation



15	8	20	2	11	30	27	13	6	18	12
1	2	3	4	5	6	7	8	9	10	11

- Notice: Left child of A[5] (of data 11) is A[2*5]=A[10] (of data 18), and its right child is A[2*5+1]=A[11] (of data 12).
- Parent of A[4] is A[4/2]=A[2], and parent of A[5]=A[5/2]=A[2]

Array Representation [For index 0]



for the item in A[i]:
leftChild is in A[2i+1]
rightChild is in A[2i+2]
parent is in A[(i-1)/2]

A	23	18	9	8	12	7	1	4	2
	0	1	2	3	4	5	6	7	8

Size
9

Tree Structure

A node of a binary tree can be represented in C as follows:

```
struct BTreeNode {
    int data;
    BTreeNode *left;
    BTreeNode *right;
};
```

Convert a Generic Tree to a Binary Tree

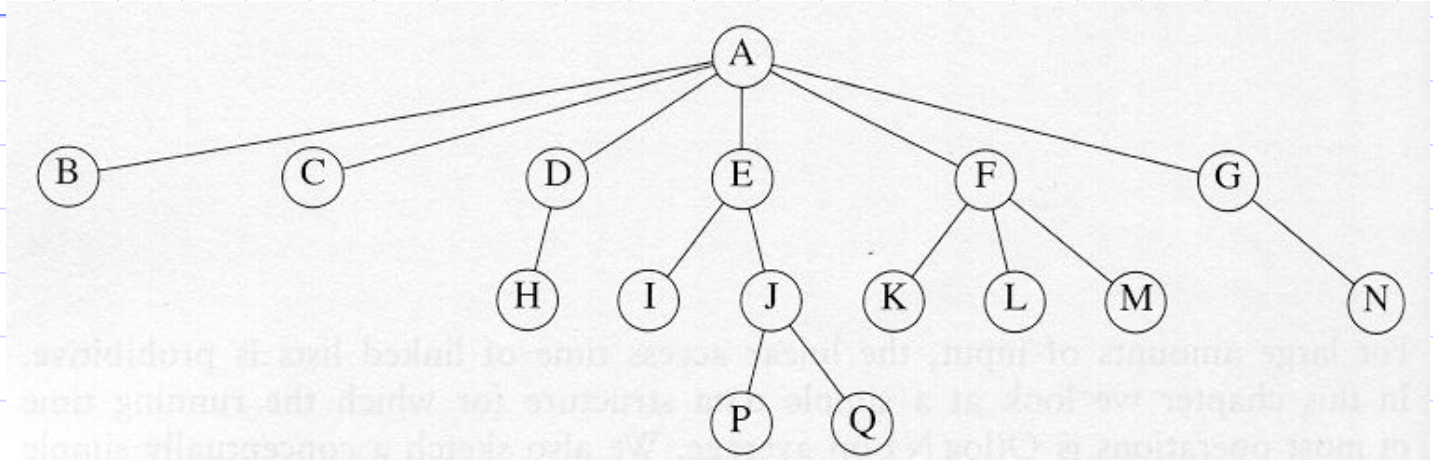


Figure 4.2 A tree

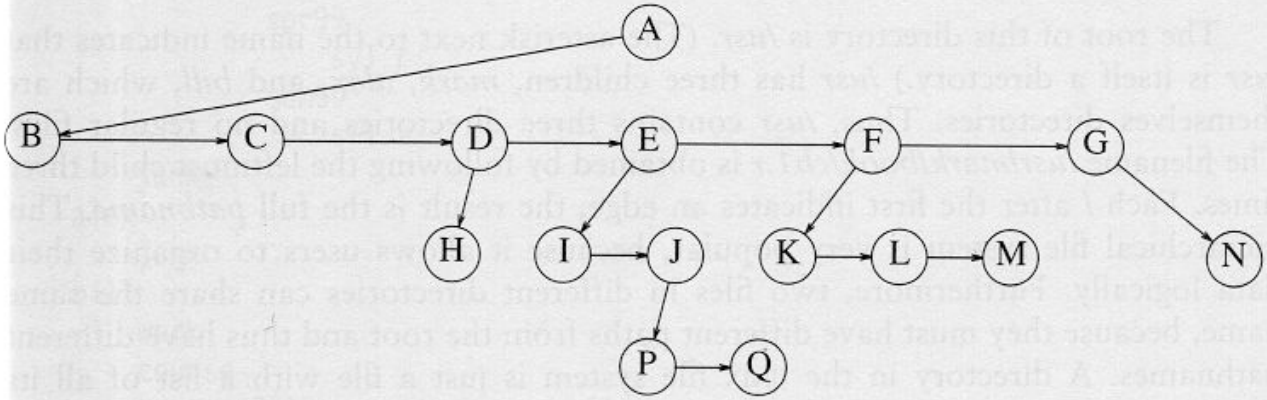


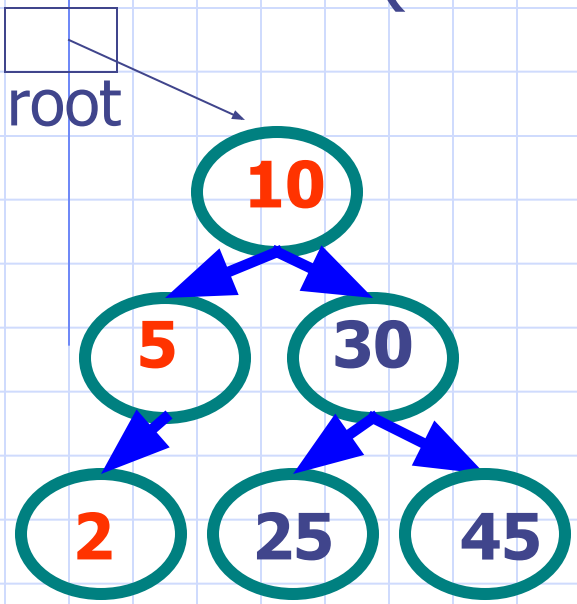
Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Binary Search Tree

- Binary search tree definition
  - A set of nodes T is a binary search tree if either of the following is true
    - T is empty
    - Its root has two subtrees such that each is a binary search tree and the value in the root is greater than all values of the left subtree but less than all values in the right subtree
- A data structure for efficient searching, insertion and deletion

Example Binary Searches

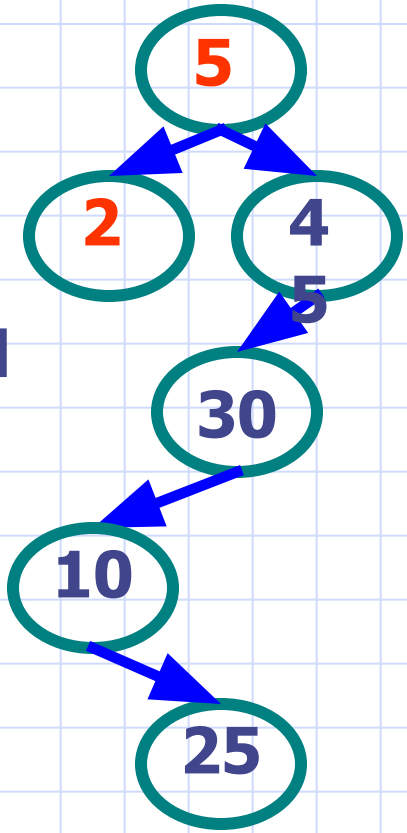
- Find (root, 2)



10 > 2, left

5 > 2, left

2 = 2, found

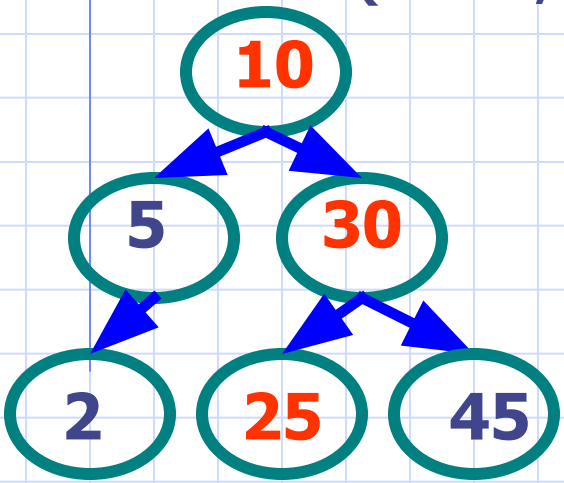


5 > 2, left

2 = 2, found

Example Binary Searches

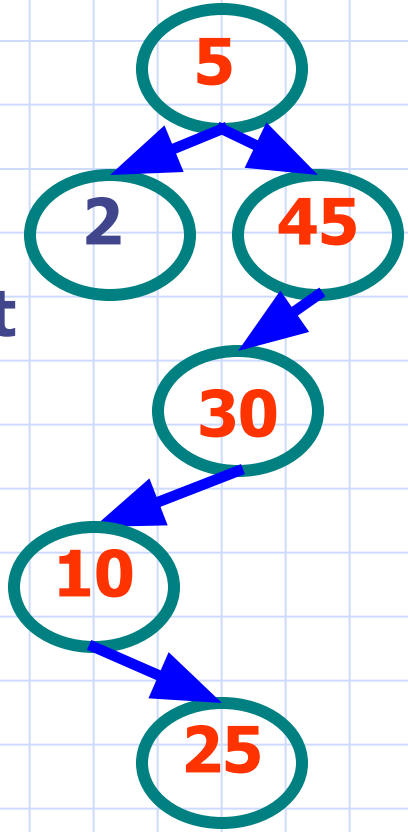
- Find (root, 25)



10 < 25, right

30 > 25, left

25 = 25, found



5 < 25, right

45 > 25, left

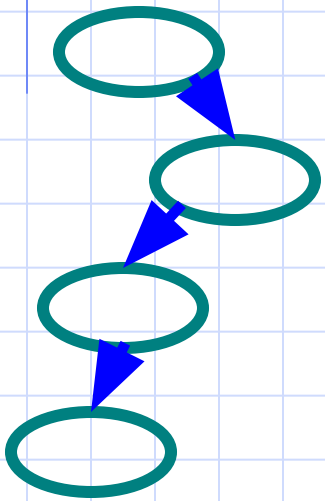
30 > 25, left

10 < 25, right

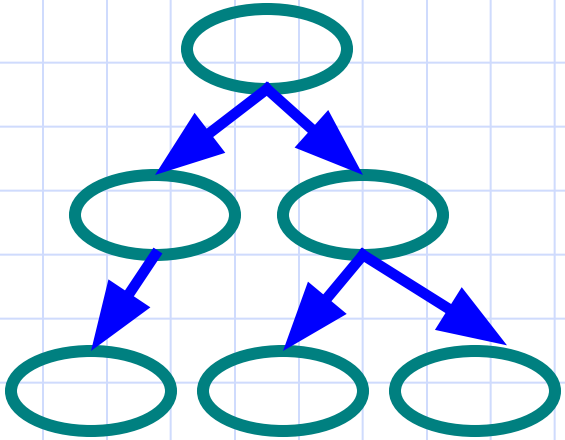
25 = 25, found

Types of Binary Trees

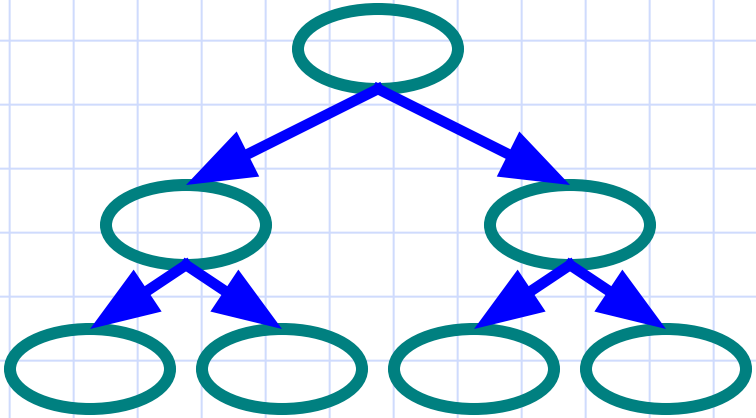
- Degenerate – only one child
- Complete – always two children
- Balanced – "mostly" two children



Degenerate binary tree



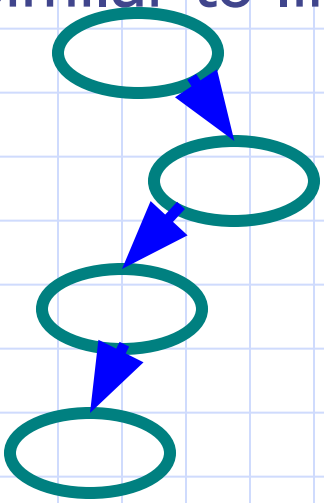
Balanced binary tree



Complete binary tree

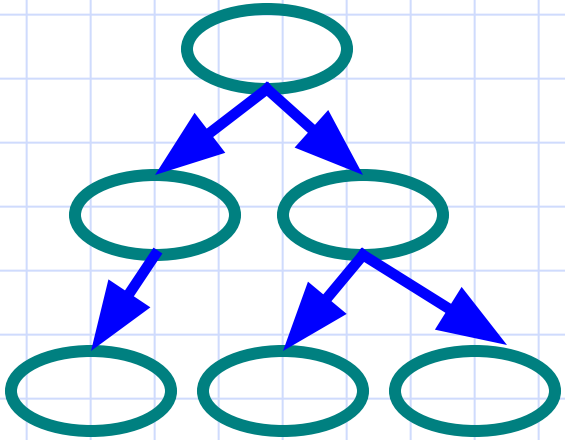
Binary Trees Properties

- Degenerate
  - Height = O(n) for n nodes
  - Similar to linked list



Degenerate binary tree

- Balanced
  - Height = O(log(n)) for n nodes
  - Useful for searches



Balanced binary tree

Binary Search Properties

- Time of search
  - Proportional to height of tree
  - Balanced binary tree
    - O(log(n)) time
  - Degenerate tree
    - O(n) time
    - Like searching linked list / unsorted array

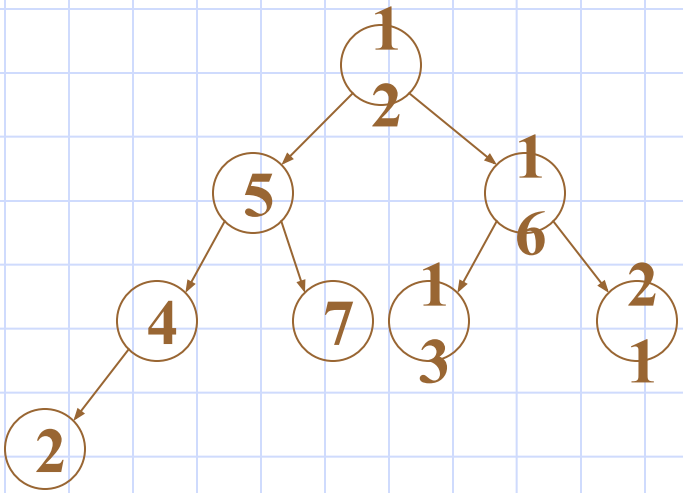
Binary Search Tree Construction

- How to build & maintain binary trees?
  - Insertion
  - Deletion
- Maintain key property (invariant)
  - Smaller values in left subtree
  - Larger values in right subtree

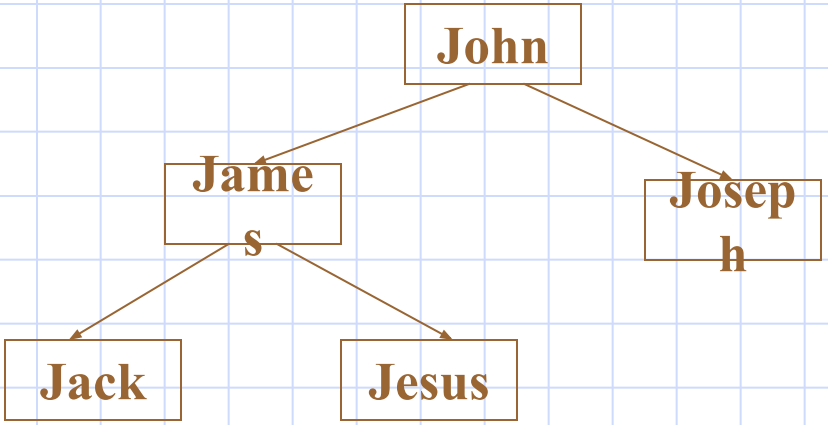
Binary Search Tree – Insertion

- Algorithm
  1. Perform search for value X
  2. Search will end at node Y (if X not in tree)
  3. If X < Y, insert new leaf X as new left subtree for Y
  4. If X > Y, insert new leaf X as new right subtree for Y
- Observations
  - O(log(n)) operation for balanced tree
  - Insertions may unbalance tree

Examples

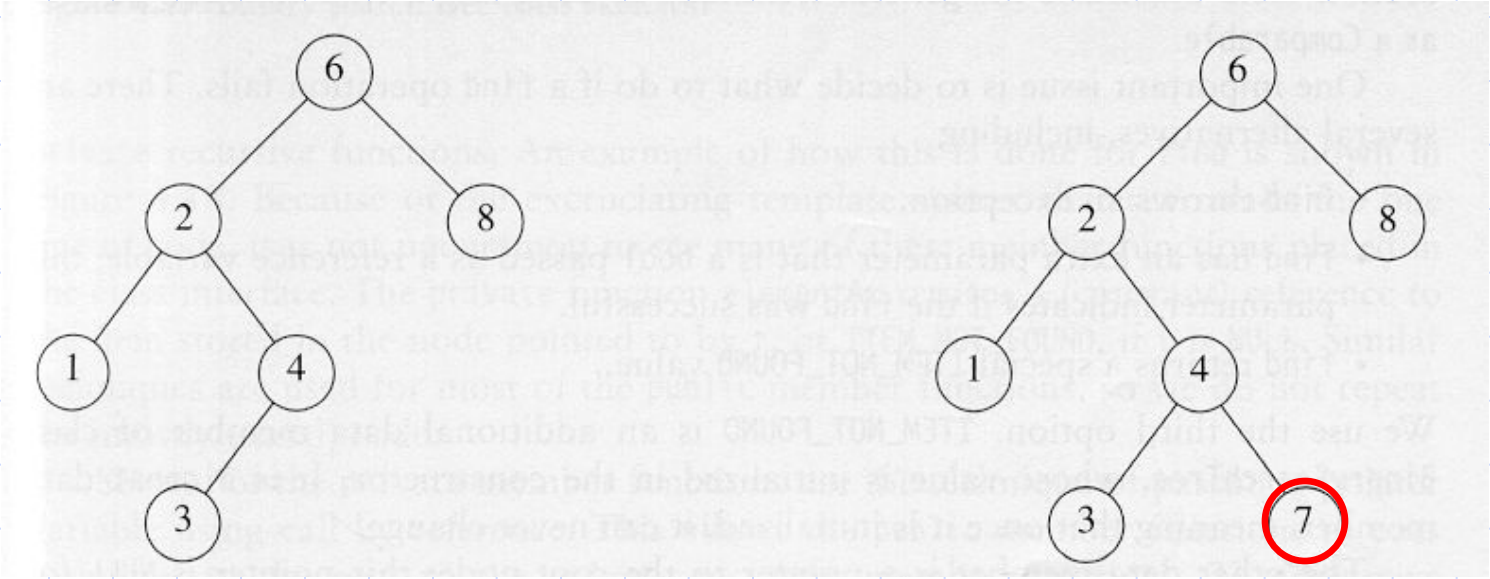


BST for numbers



BST for names

Binary Search Trees

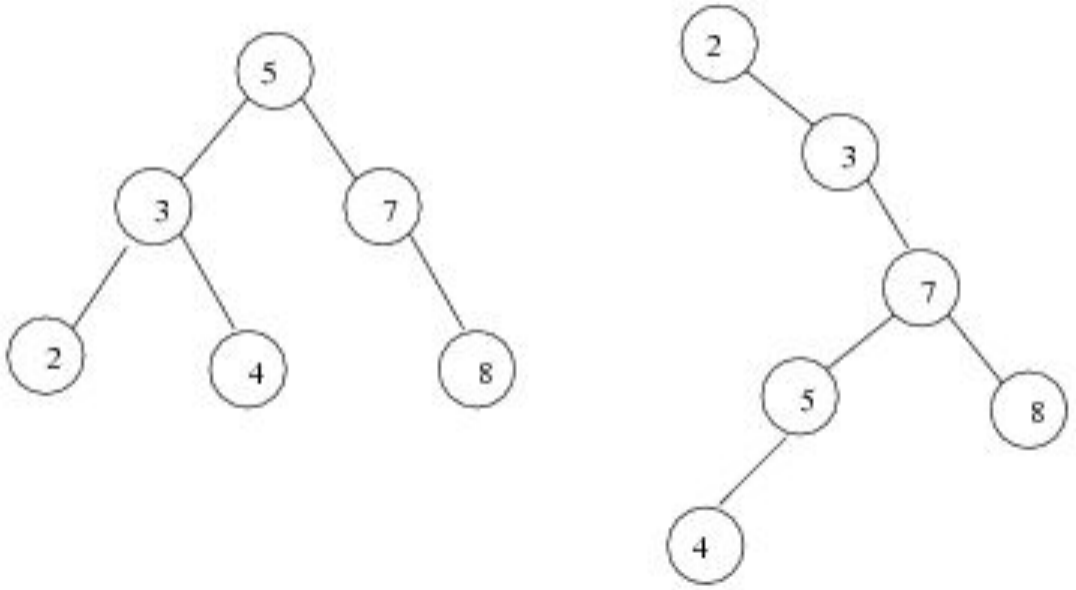


A binary search tree

Not a binary search tree

Binary Search Trees

The same set of keys may have different BSTs



- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

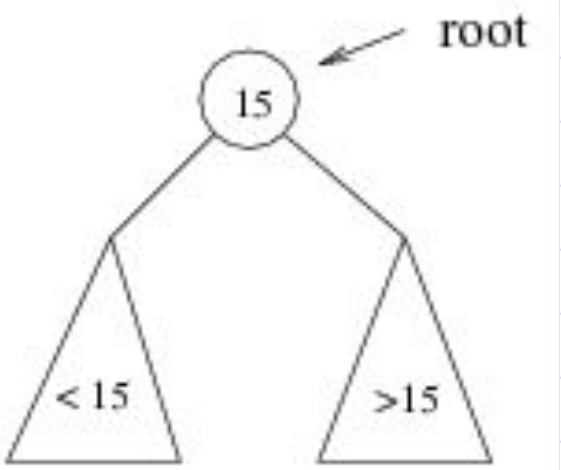
Constructing a BST

- **Base case:**
  if tree is empty, create new node for item

- **Recursive case:**
  if key < root's value, add to the left subtree, otherwise to the right subtree

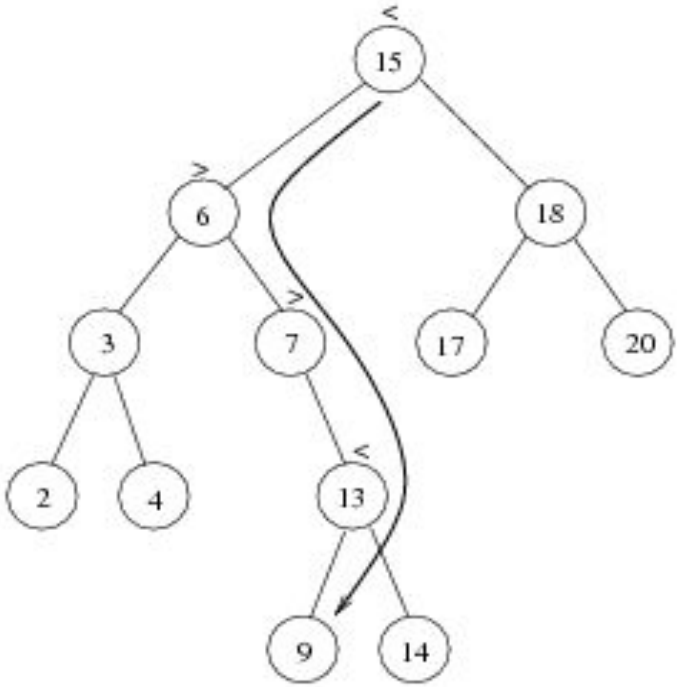
Searching BST

- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Searching BST

Example: Search for 9 ...

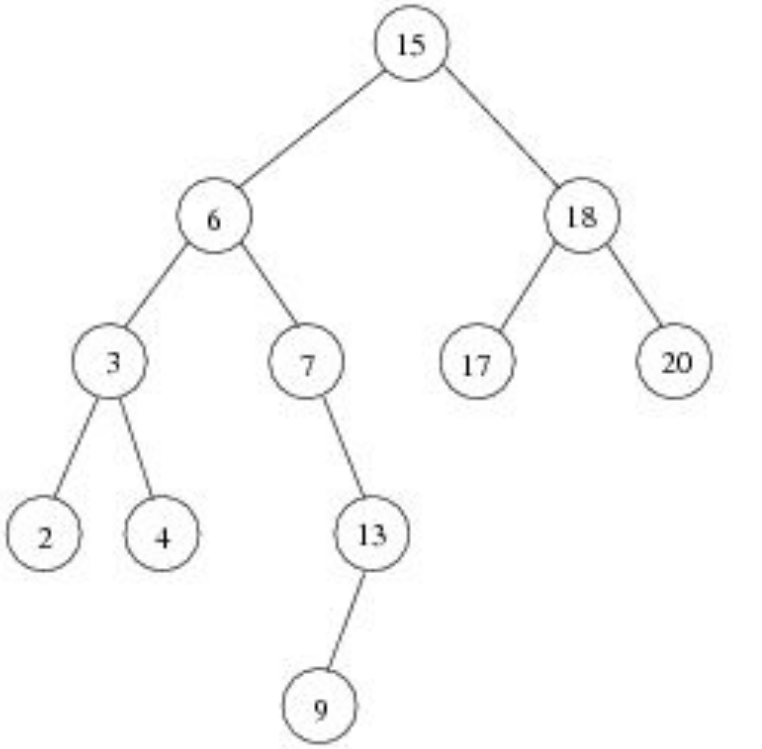


Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Inorder Traversal of BST

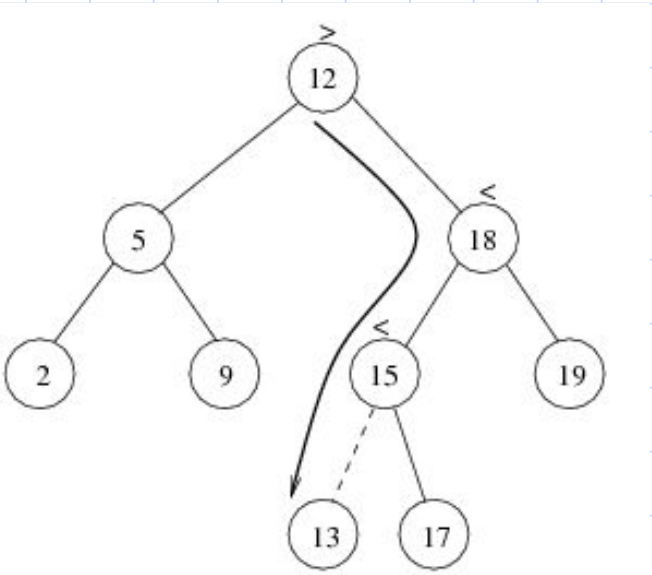
- Inorder traversal of BST prints out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Insertion

- Proceed down the tree as you would with a find
- If X is found, do nothing (or update something)
- Otherwise, insert X at the last spot on the path traversed



- Time complexity = O(height of the tree)

Inserting an Element in a Binary Search Tree

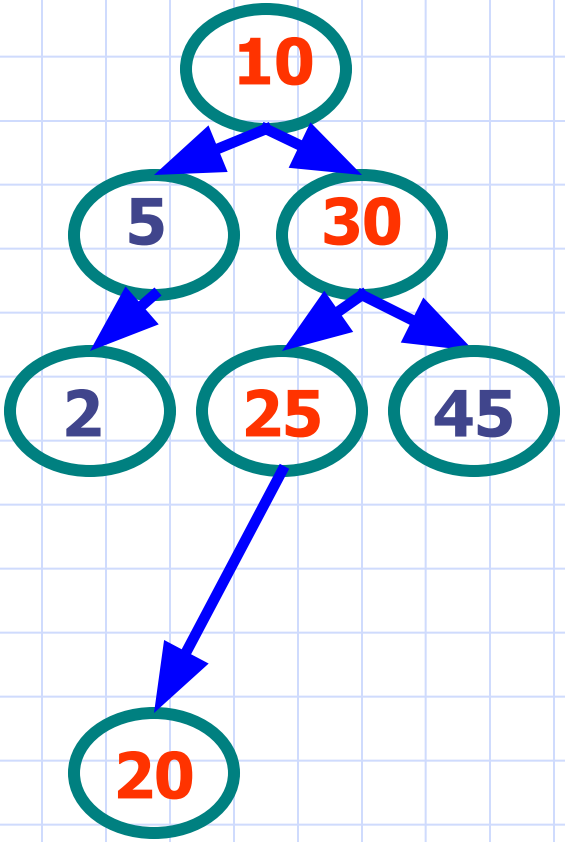
- Inserting an element in a binary search tree involves creating a new node and re-organizing.
- The parent nodes link to accommodate the new node.
- Typically insertions will always be on a terminal node.
- We will never be inserting a node in the middle of a binary search tree.

Steps for inserting an element in a binary search tree

1. If the tree is empty, then make the root node point to this node
2. If the tree contains nodes then compare the data with node's data and take appropriate path till the terminal node is reached
3. If data < node's data then take the left sub tree otherwise take the right sub tree
4. When there are no more nodes add the node to the appropriate place of the parent node.

Example Insertion

- Insert (20)



10 < 20, right

30 > 20, left

25 > 20, left

Insert 20 on left

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass

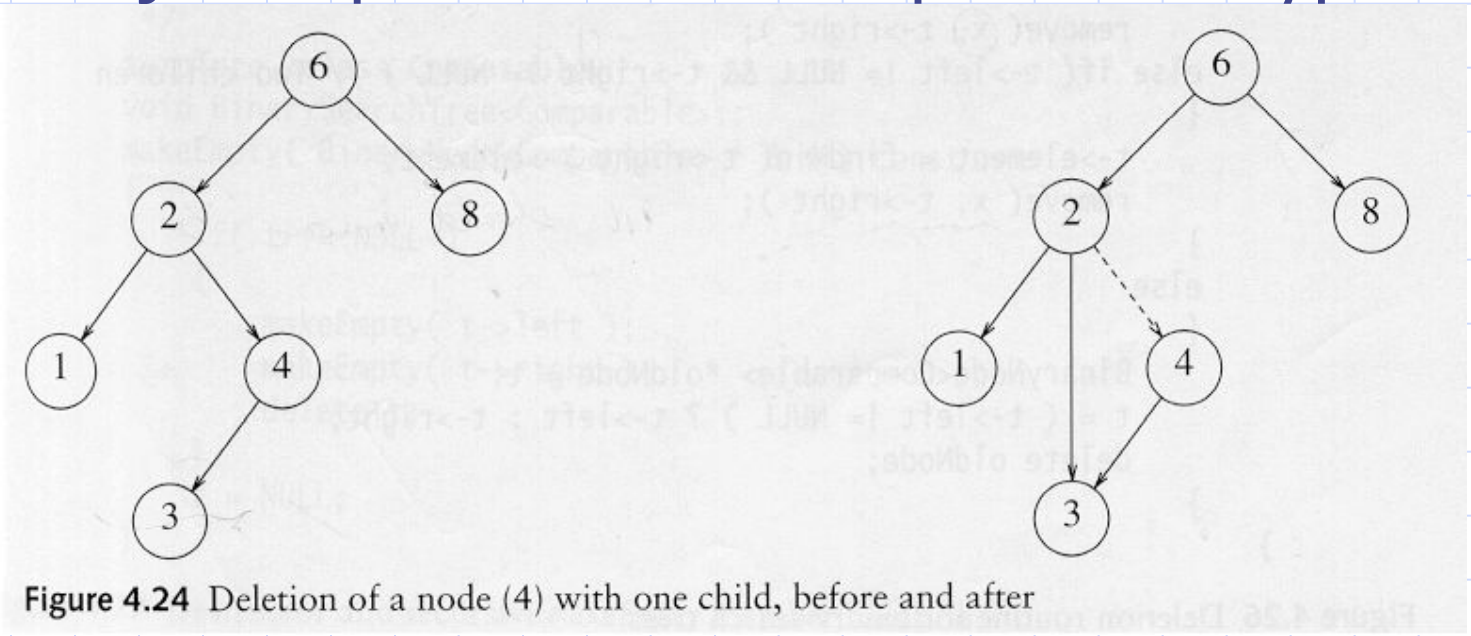


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

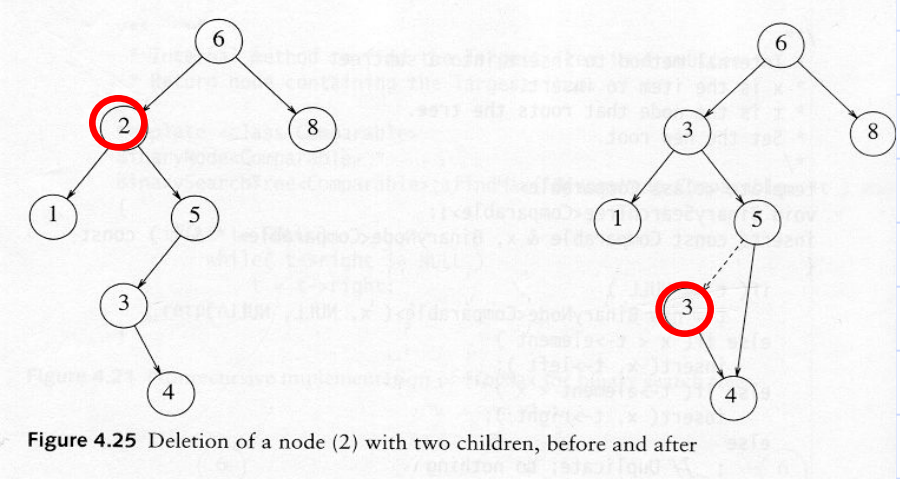


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Deleting an element from a binary tree

a) If the node being deleted is a terminal node then deletion is a straightforward procedure of just making the parent node point to

NULL.

b) If the node being deleted has only one child then the parent node will have to point to the child node of the node being deleted.

c) If the node being deleted has both the child's then we first need to find the inorder successor of the node being deleted and replace the node being deleted with this node. (The inorder successor of a node can be obtained by taking the right node of the current node and traversing in the left till we reach the left most node.)

Binary Search Tree – Deletion

- Algorithm
  1. Perform search for value X
  2. If X is a leaf, delete X
  3. Else // must delete internal node

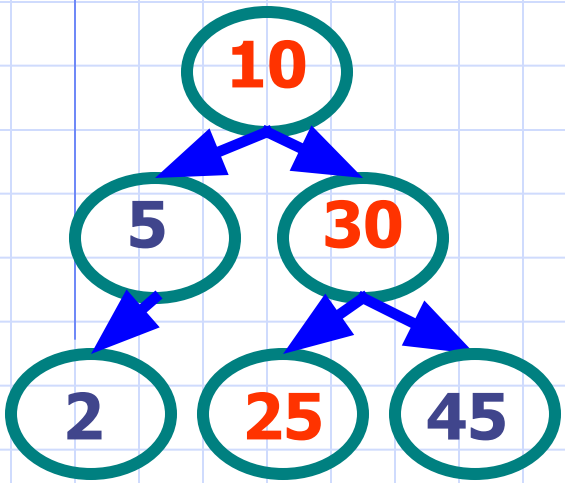
     a) Replace with largest value Y on left subtree

     OR smallest value Z on right subtree

     b) Delete replacement value (Y or Z) from subtree
- Observation
  - O(log(n)) operation for balanced tree
  - Deletions may unbalance tree

Example Deletion (Leaf)

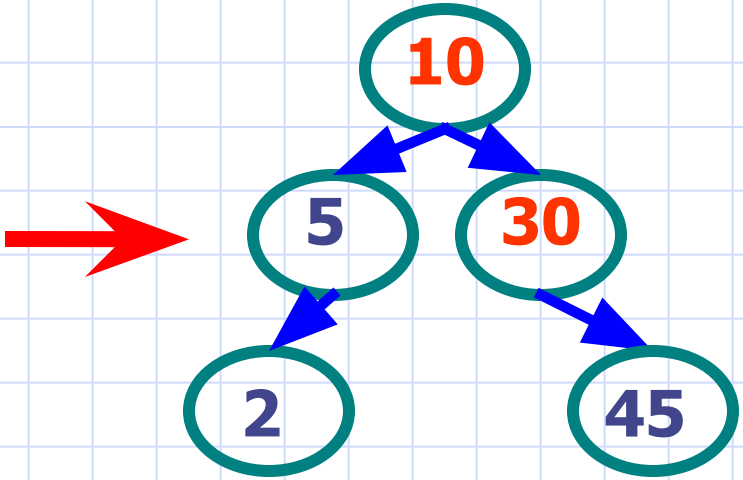
- Delete (25)



10 < 25, right

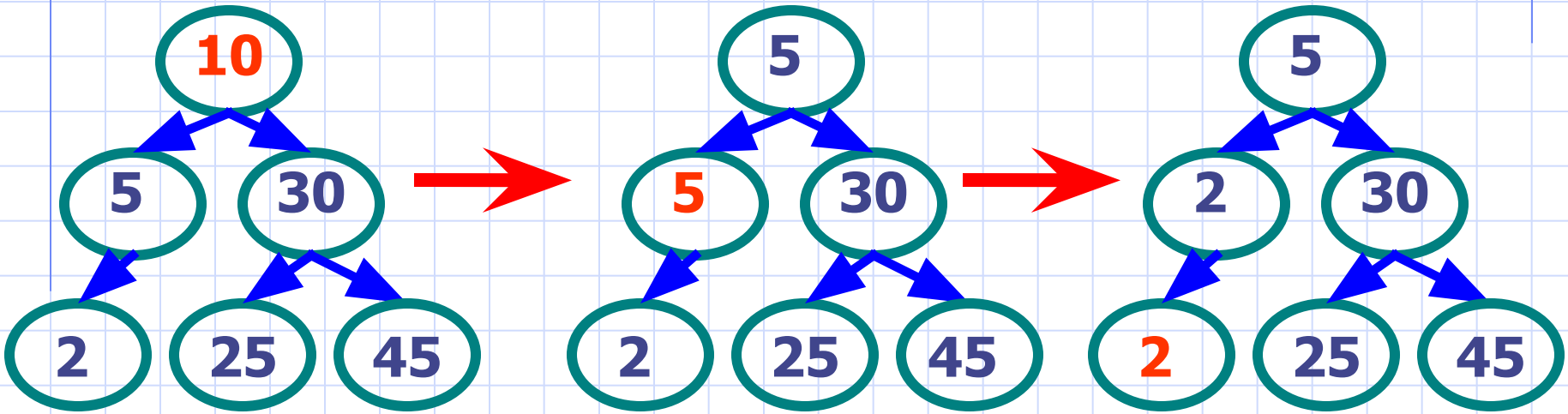
30 > 25, left

25 = 25, delete



Example Deletion (Internal Node)

- Delete (10)



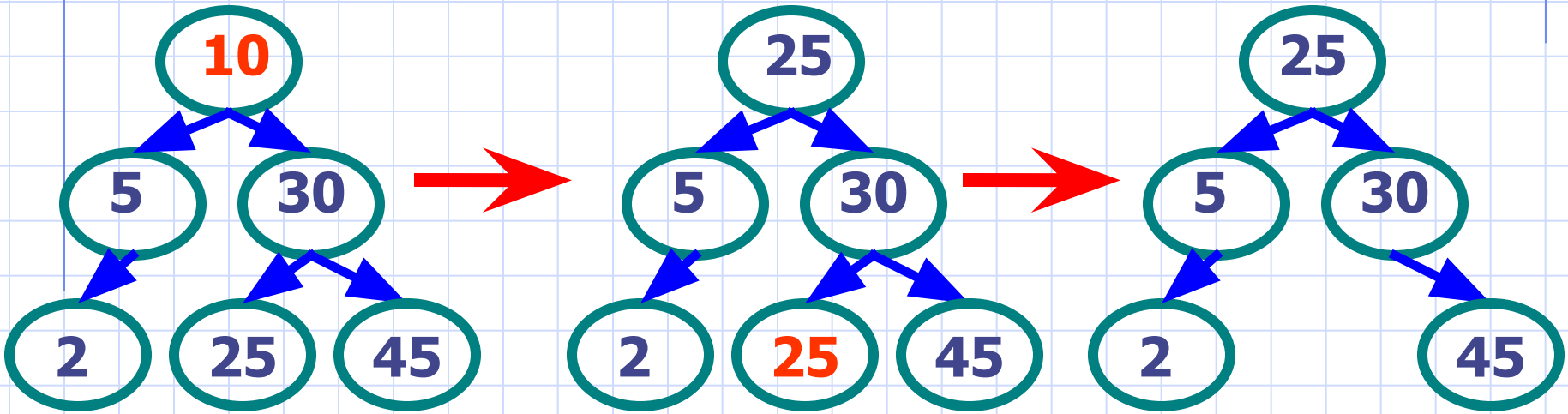
Replacing 10 with largest value in left subtree

Replacing 5 with largest value in left subtree

Deleting leaf

Example Deletion (Internal Node)

- Delete (10)



Replacing 10 with smallest value in right subtree

Deleting leaf

Resulting tree

Binary Search Algorithm

Search_Node(Node, Info)

Step1: [Initialize]

Flag=0

Step2: Repeat through step 3 while Node # NULL

Step3: If Info[Node]=Info

Flag=1

Return(Flag)

else if Info< Info[Node]

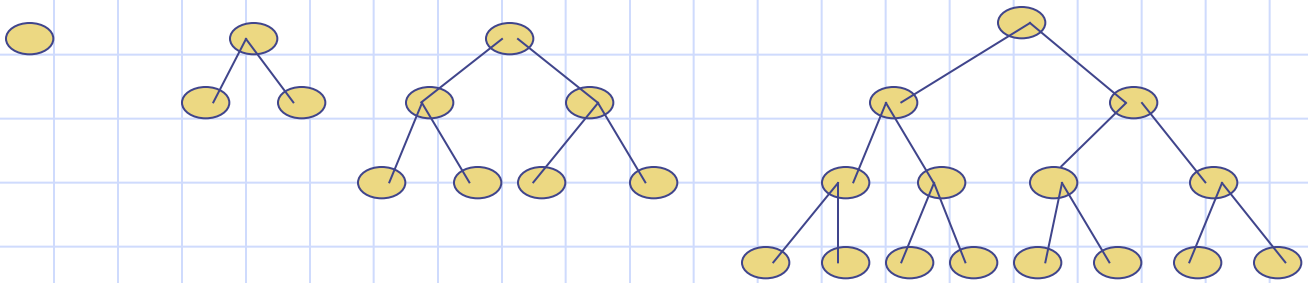
Node=Left_Child[Node]

else Node=Right_Child[Node]

Step4: Return (Flag)

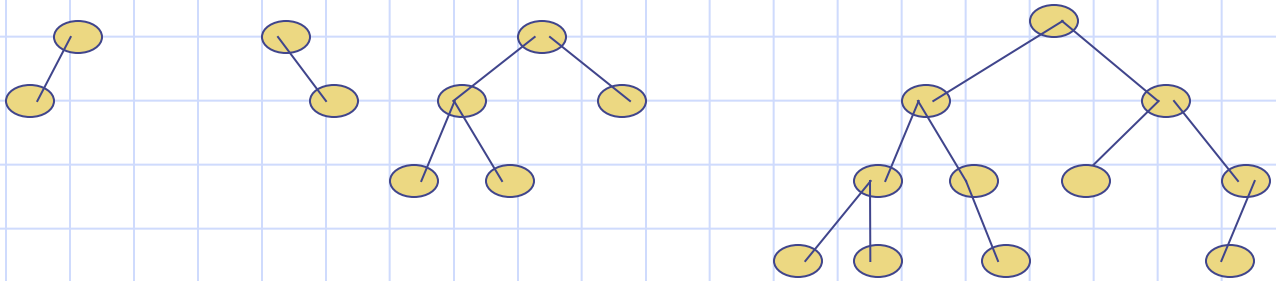
Other Kinds of Binary Trees (Full Binary Trees)

- **Full Binary Tree**: A full binary tree is a binary tree where all the leaves are on the same level and every non-leaf has two children
- The first four full binary trees are:



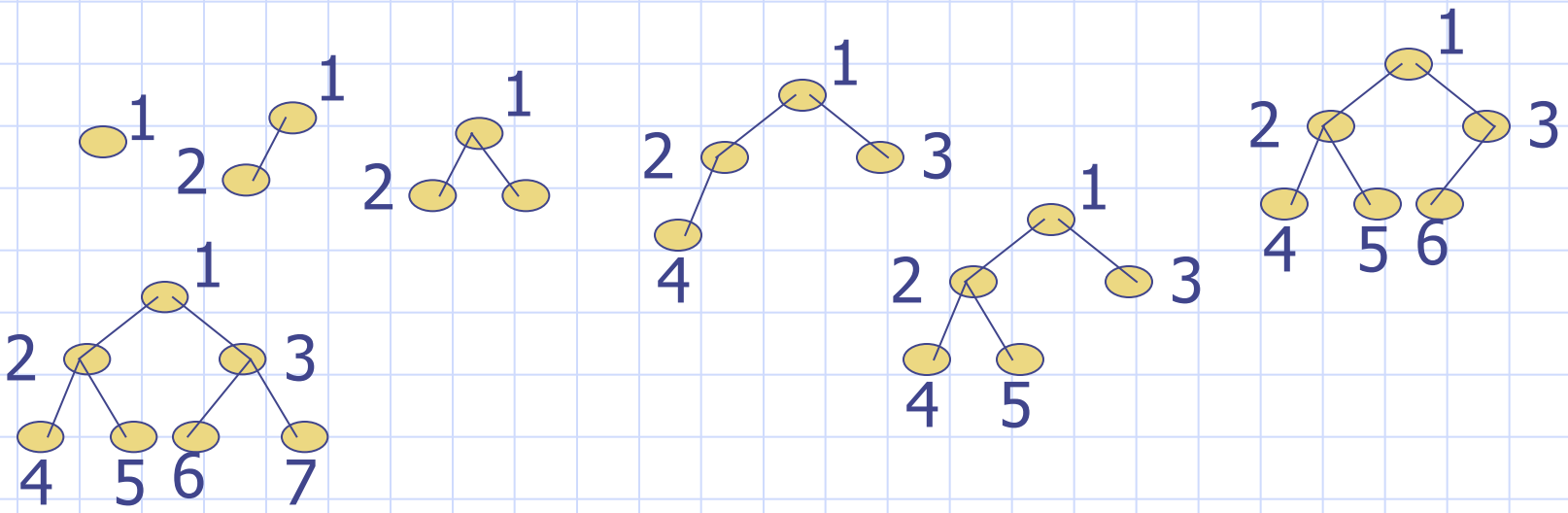
Examples of Non-Full Binary Trees

- These trees are NOT full binary trees:



Other Kinds of Binary Trees (Almost Complete Binary trees)

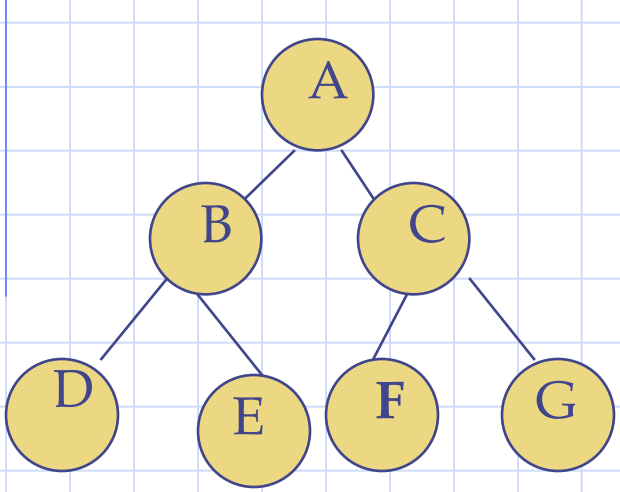
- **Almost Complete <u>Binary Tree</u>**: An almost complete binary tree of n nodes, for any arbitrary nonnegative integer n, is the binary tree made up of the first n nodes of a canonically labeled full binary

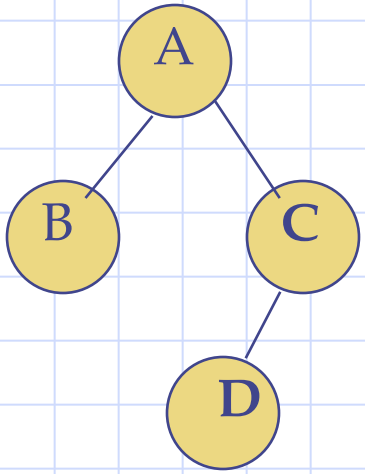


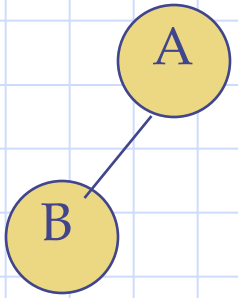
AVL Tree

- If height of both L and R subtrees are given then searching is efficient
- Frequent insertions and deletions make it unbalanced
- Efficiency is ideal if difference between L and R subtrees is 1
- It is called AVL tree or height balanced tree

Some AVL Trees







Balance at a node is height(left subtree) - height(right subtree)

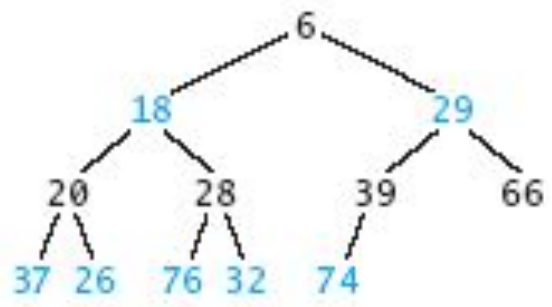
Heap

- is a binary tree that
  - is **complete (but not necessarily FULL),**
    - **except LAST level**
  - has the **heap-order property**
    - **max heap** - item stored in each node has a key/priority that is >= the priority of the items stored in each of its children
    - **min heap** - item stored in each node has a key/priority that is <= the priority of the items stored in each of its children
- efficient data structure for PriorityQueue ADT
  - requires the ability to compare items based on their priorities
- basis for the heapsort algorithm

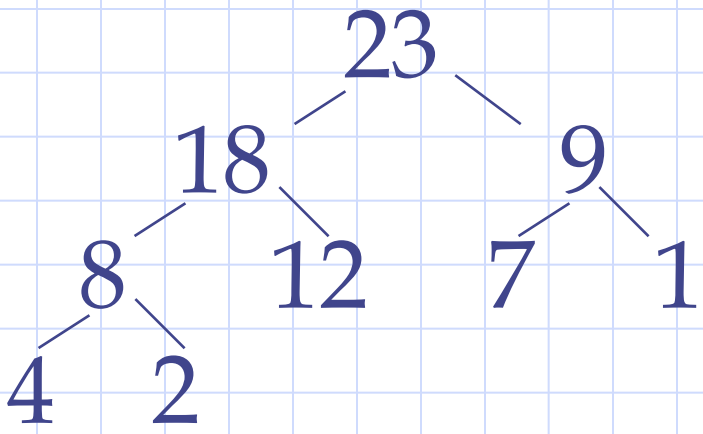
Heap

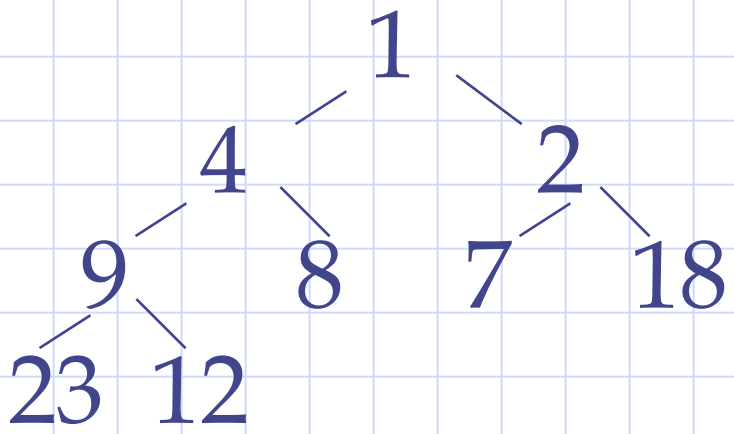
- In a heap, the value in a node is less than all values in its two sub-trees
- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every sub-tree is a heap

FIGURE 8.20
Example of a Heap



Two heaps





A heap is **always** a complete binary tree

Operations on Heaps

- Delete the minimum value and return it. This operation is called deleteMin.
- Insert a new data value

Applications of Heaps:

- A heap implements a priority queue, which is a queue that orders entities not a on first-come first-serve basis, but on a priority basis: the item of highest priority is at the head, and the item of the lowest priority is at the tail
- Another application: sorting, which will be seen later

Other (Non-Search) Trees

- Parse trees
  - Convert from textual representation to tree representation
  - Textual program to tree
    - Used extensively in compilers
  - Tree representation of data
    - E.g. HTML data can be represented as a tree
      - called DOM (Document Object Model) tree
    - XML
      - Like HTML, but used to represent data
      - Tree structured