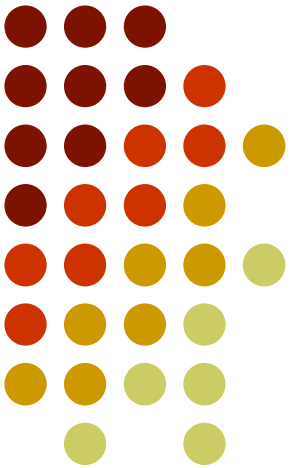
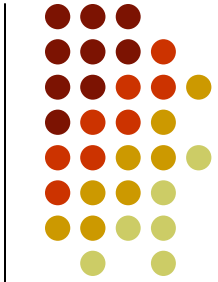
CMPT 225

Binary Search Trees

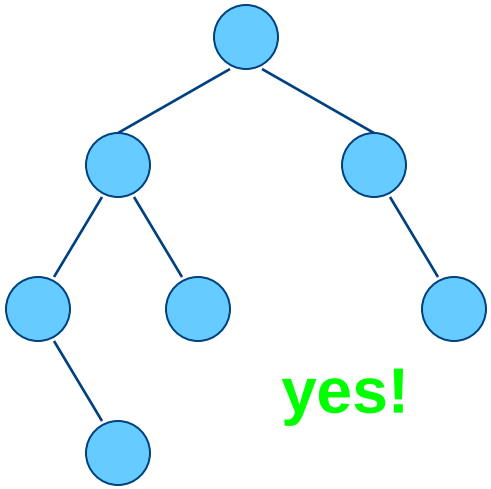


Trees

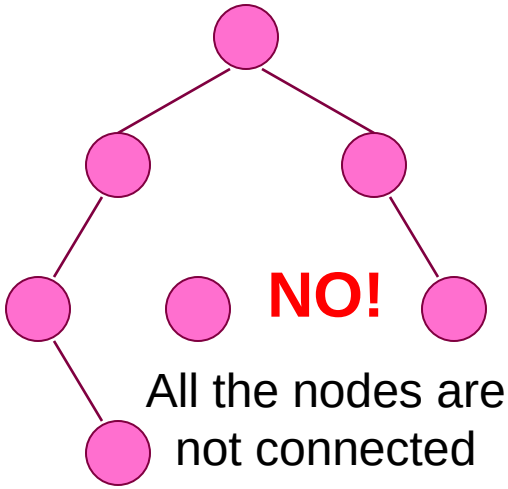


- A set of nodes with a single starting point
  - called the **root**
- Each node is connected by an edge to some other node
- A tree is a connected graph
  - There is a path to every node in the tree
- There are no cycles in the tree.
  - It can be proved by MI that a tree has one less edge than the number of nodes.
- Usually depicted with the root at the top

Is it a Tree?

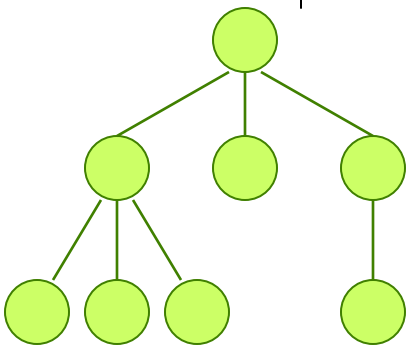


yes!

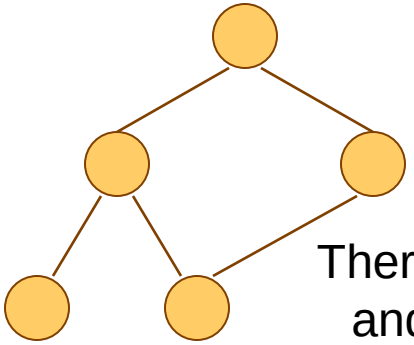


NO!

All the nodes are not connected

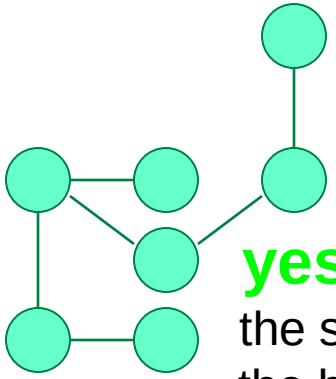


yes! (but not a binary tree)



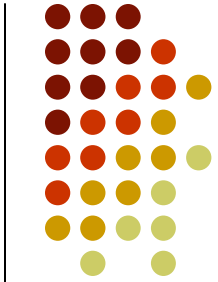
NO!

There is a cycle and an extra edge (5 nodes and 5 edges)



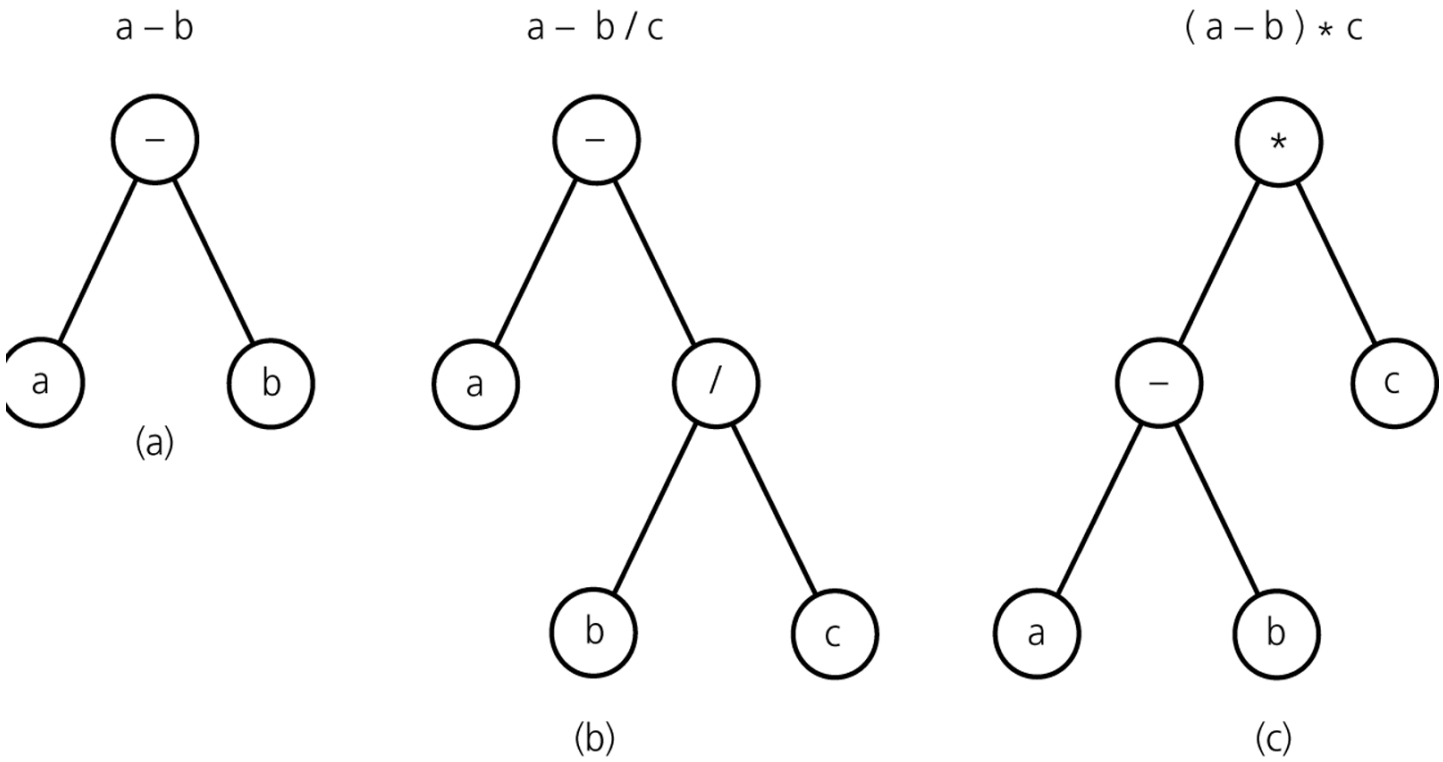
yes! (it's actually the same graph as the blue one) – but usually we draw tree by its "levels"

Examples of trees



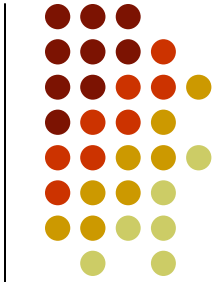
- directory structure
- family trees:
  - all descendants of a particular person
  - all ancestors born after year 1800 of a particular person
- evolutionary tress (also called phylogenetic trees)
- albegraic expressions

Examples of trees

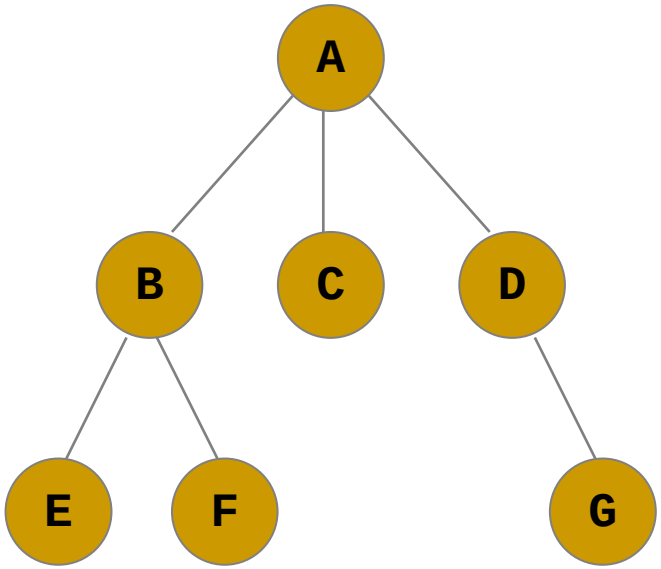


Binary trees that represent algebraic expressions

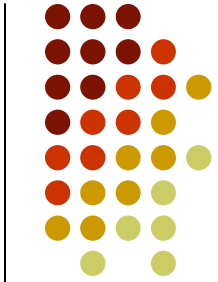
Tree Relationships



- If there is an edge between two nodes *u* and *v*, and *u* is “above” *v* in the tree (closer to the root), then *v* is said to be a **child** of *u,* and *u* the **parent** of *v*
  - A is the parent of B, C and D
- This relationship can be generalized (transitively)
  - E and F are **descendants** of A
  - D and A are **ancestors** of G
  - B, C and D are **siblings**

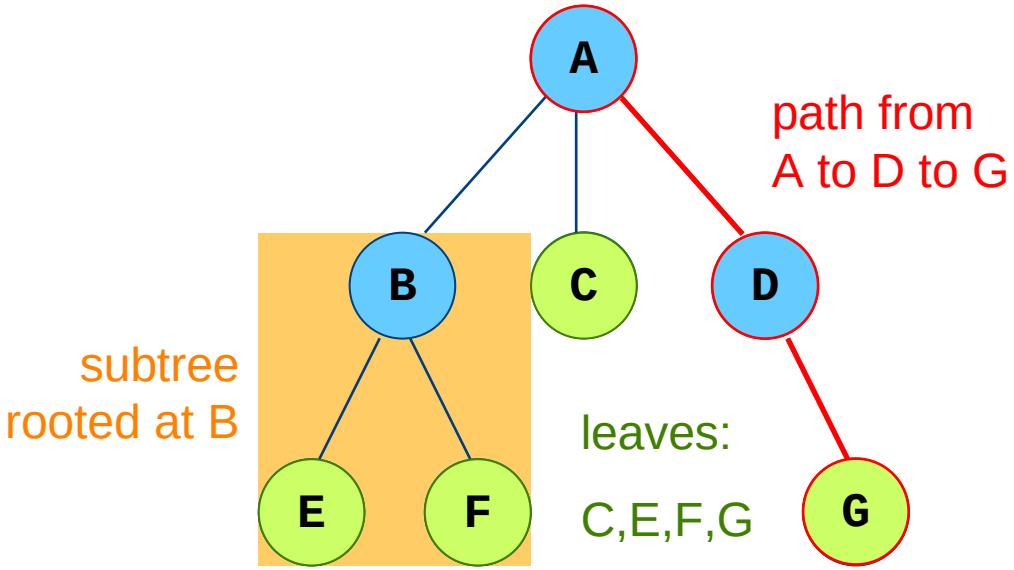


More Tree Terminology

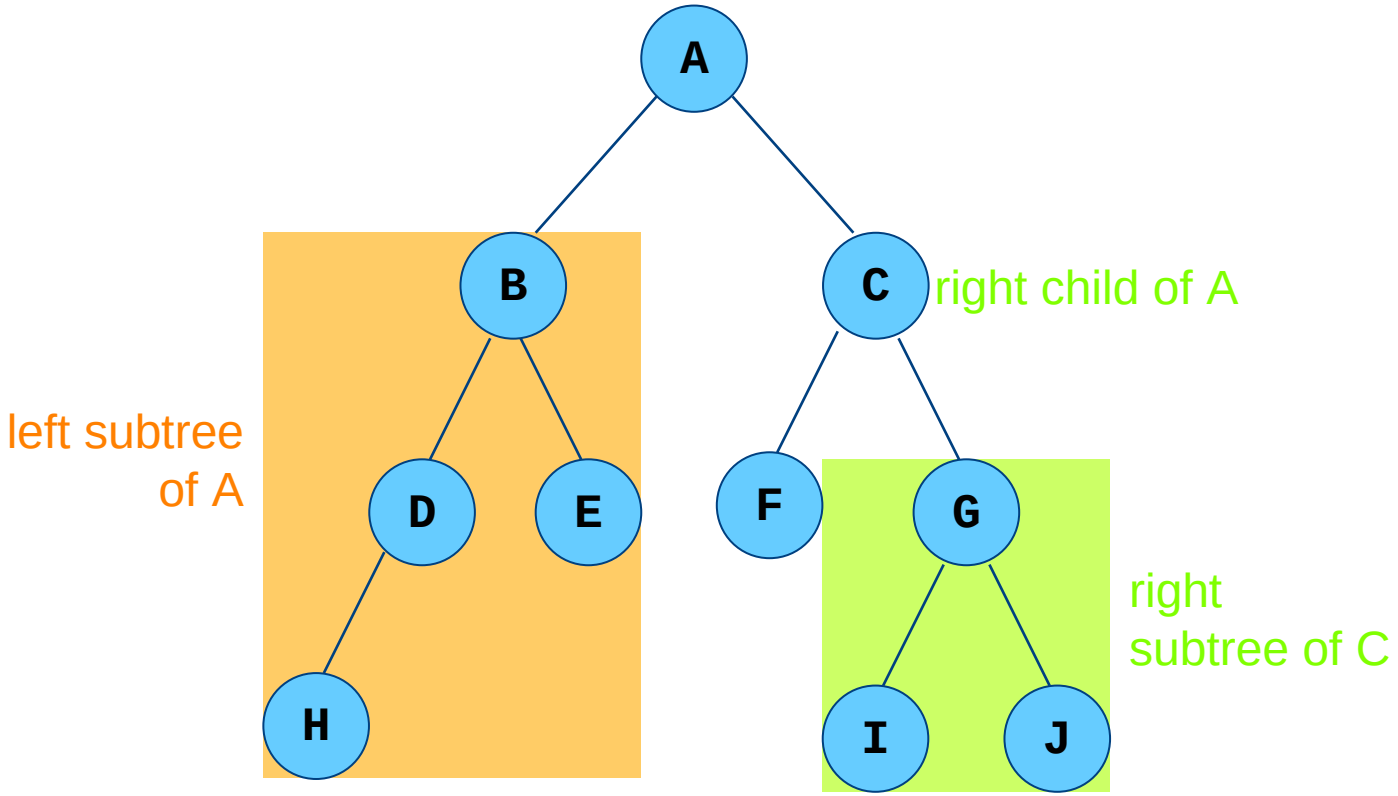


- A **leaf** is a node with no children
- A **path** [a **branch**] is a sequence of nodes $v_1 \ldots v_n$
  - where v_i is a parent of v_{i+1} ($1 \leq i \leq n\text{-}1$) [and v_1 is a root and v_n is a leaf]
- A **subtree** is any node in the tree along with all of its descendants
- A **binary** tree is a tree with at most two children per node
  - The children are referred to as left and right (i.e., children are usually ordered)
  - We can also refer to left and right subtrees of a node

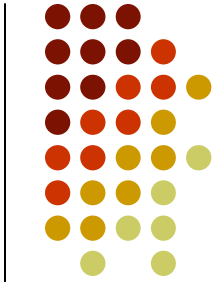
Tree Terminology Example



Binary Tree

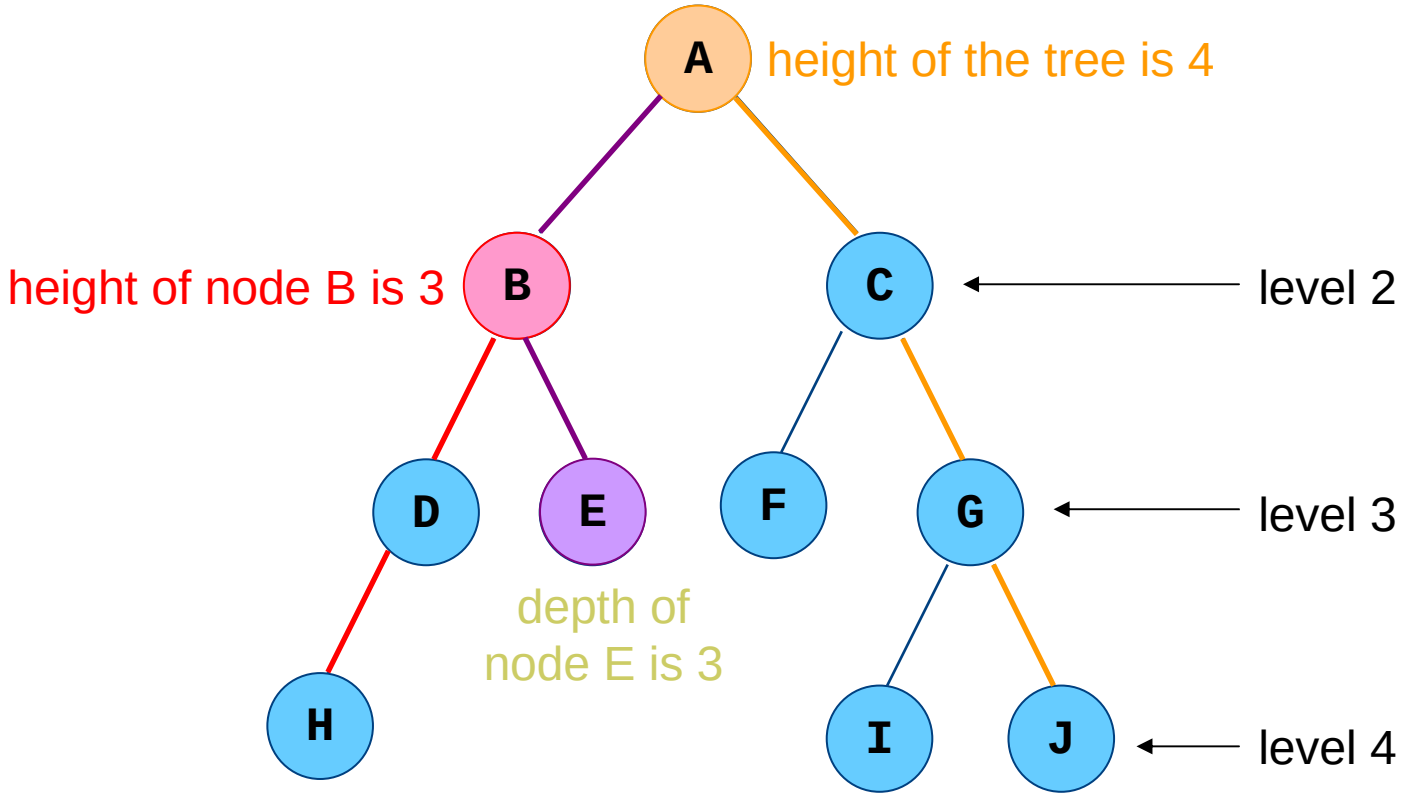

A
B
C
D
E
F
G
H
I
J
left subtree
of A
right child of A
right
subtree of C

Measuring Trees

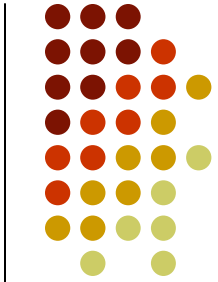


- The **height** of a node *v* is the number of nodes on the longest path from *v* to a leaf
  - The height of the tree is the height of the root, which is the number of nodes on the longest path from the root to a leaf
- The **depth** of a node *v* is the number of nodes on the path from the root to *v*
  - This is also referred to as the **level** of a node
- Note that there are slightly different formulations of the height of a tree
  - Where the height of a tree is said to be the length (the number of edges) on the longest path from node to a leaf

Height of a Binary Tree



Representation of binary trees

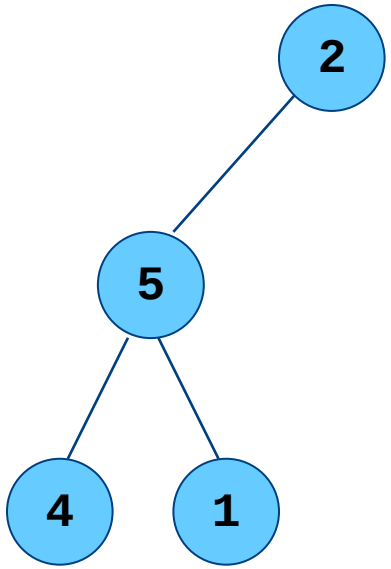


```
public class TreeNode<T> {
  private T item;
  private TreeNode<T> leftChild;
  private TreeNode<T> rightChild;

  public TreeNode(T newItem) {
  // Initializes tree node with item and no children (a leaf).
    item = newItem;
    leftChild  = null;
    rightChild = null;
  }  // end constructor

  public TreeNode(T newItem,
                  TreeNode<T> left, TreeNode<T> right) {
  // Initializes tree node with item and
  // the left and right children references.
    item = newItem;
    leftChild  = left;
    rightChild = right;
  }  // end constructor

  public T getItem() {
  // Returns the item field.
    return item;
  }  // end getItem
```

```
  public void setItem(T newItem) {
  // Sets the item field to the new value newItem.
    item  = newItem;
  }  // end setItem

  public TreeNode<T> getLeft() {
  // Returns the reference to the left child.
    return leftChild;
  }  // end getLeft

  public void setLeft(TreeNode<T> left) {
  // Sets the left child reference to left.
    leftChild  = left;
  }  // end setLeft

  public TreeNode<T> getRight() {
  // Returns the reference to the right child.
    return rightChild;
  }  // end getRight

  public void setRight(TreeNode<T> right) {
  // Sets the right child reference to right.
    rightChild  = right;
  }  // end setRight
}  // end TreeNode
```

Representing a tree

```
TreeNode<Integer> root=new
  TreeNode<Integer>(new Integer(2));
TreeNode<Integer> root.setLeft(
  new TreeNode<Integer>(new Integer(5),
    new TreeNode<Integer>(new Integer(4)),
    new TreeNode<Integer>(new Integer(1))));
```

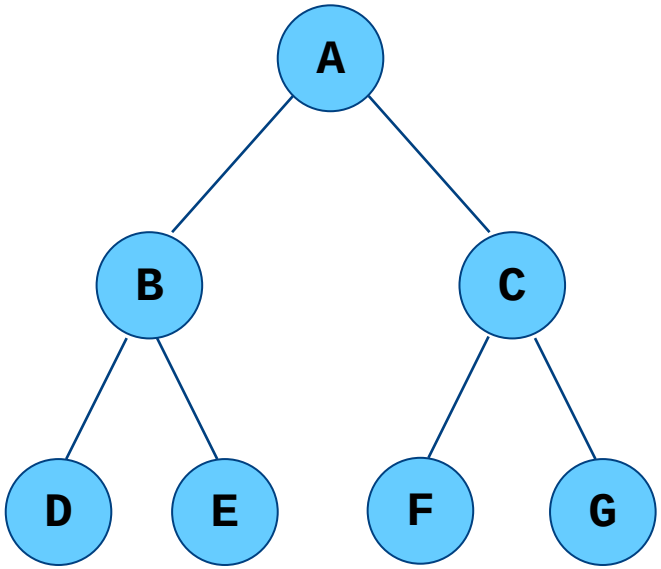


- How to compute height of a node?
  - if tree is empty, height is 0 (no levels at all)
  - if tree has just a root, height is 1
  - ***height****(T) = 1* + max(***height****(left-subtree(T),*
    height*(right-subtree(T))*

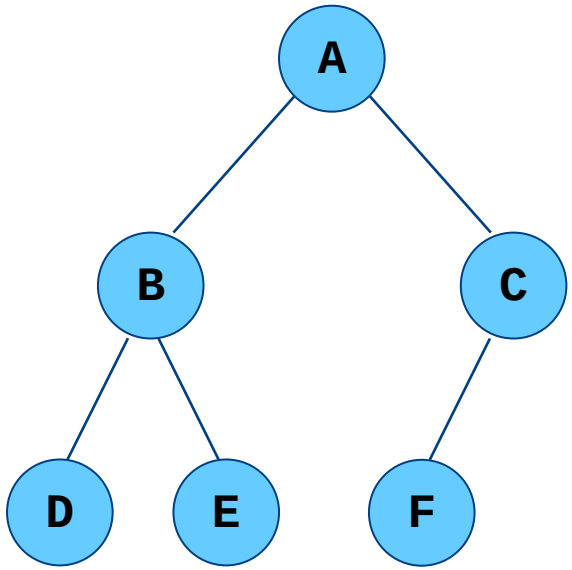
Special types of Binary Trees

- A binary tree is **full** if no node has only one child and if all the leaves have the same depth
  - A full binary tree of height *h* has $(2^h - 1)$ nodes, of which 2^{h-1} are leaves
- A **complete** binary tree is one where
  - The leaves are on at most two different levels,
  - The second to bottom level is filled in and
  - The leaves on the bottom level are as far to the left as possible.
- A **balanced** binary tree is one where
  - No leaf is more than a certain amount farther from the root than any other leaf, this is sometimes stated more specifically as:
    - The height of any node's right subtree is at most one different from the height of its left subtree
  - A balanced tree's height is sometime specified in terms of its relation to the number of nodes in the tree (see red-black trees)

Perfect and Complete Binary Trees

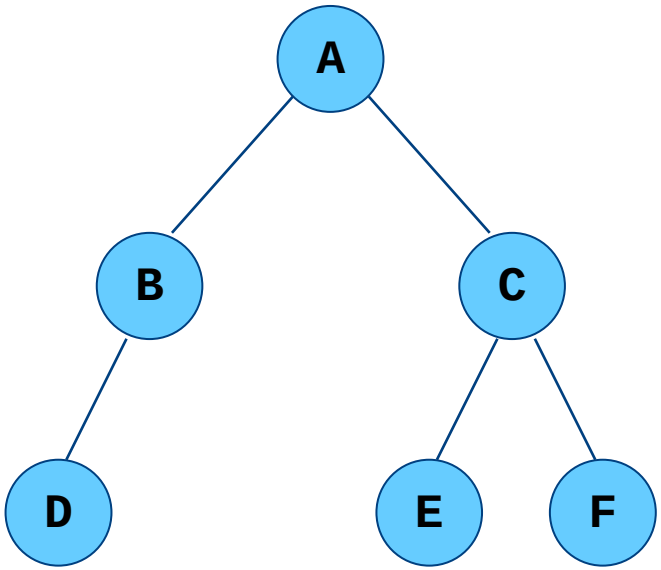


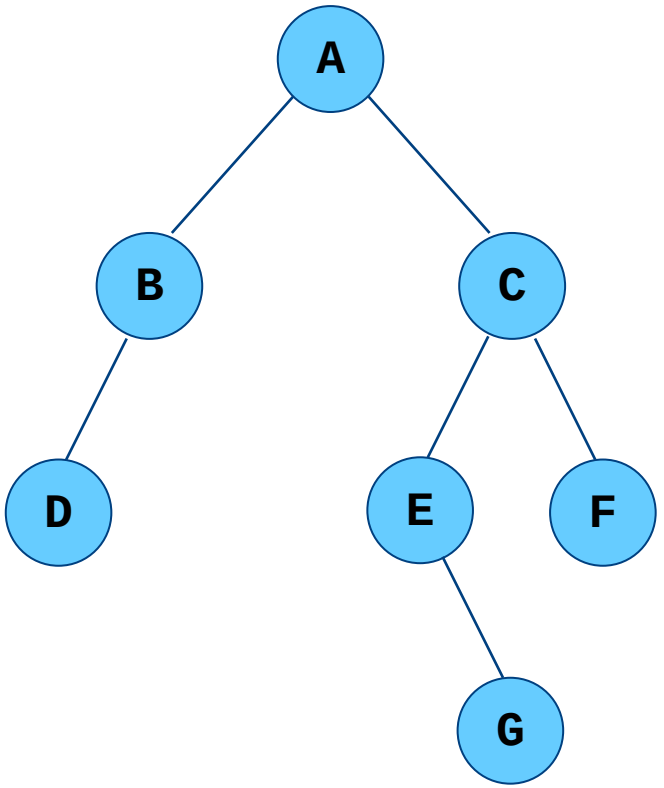
Full binary tree



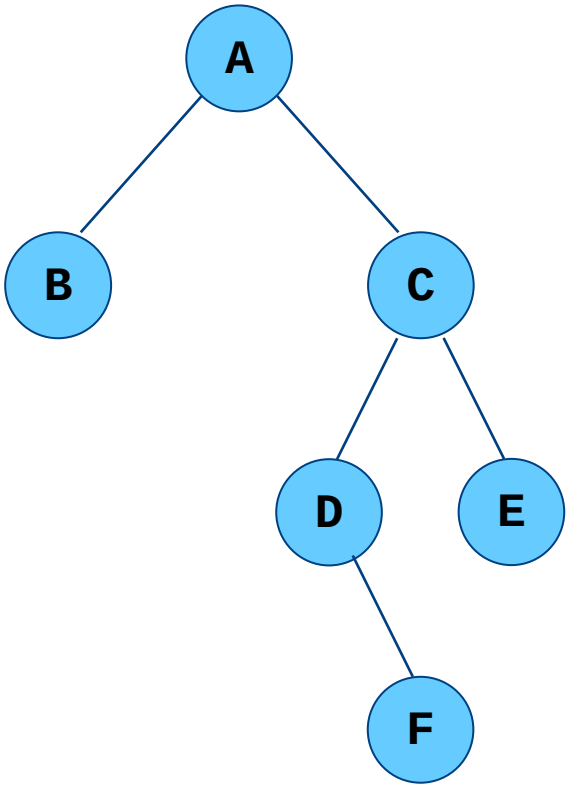
Complete binary tree

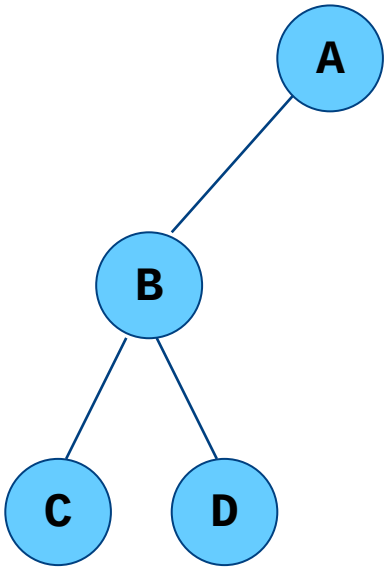
Balanced Binary Trees



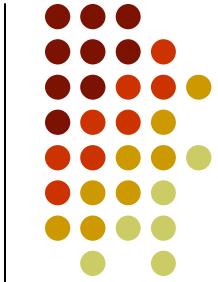


Unbalanced Binary Trees



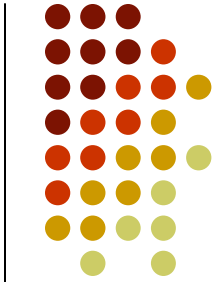


Binary Tree Traversals



- A traversal algorithm for a binary tree visits each node in the tree
  - and, typically, does something while visiting each node!
- Traversal algorithms are naturally recursive
- There are three traversal methods
  - Inorder
  - Preorder
  - Postorder

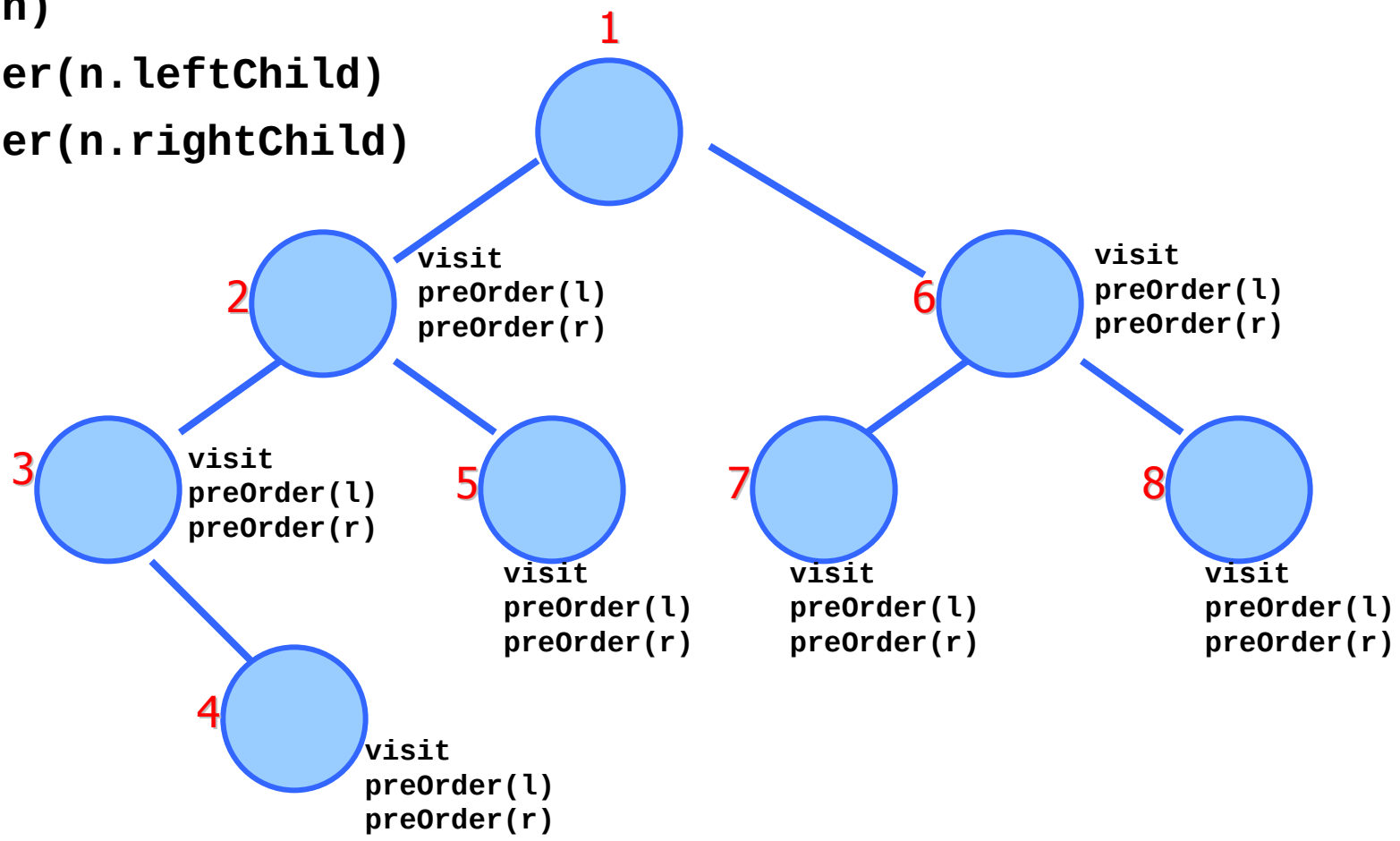
InOrder Traversal Algorithm



```
// InOrder traversal algorithm
inOrder(TreeNode<T> n) {
    if (n != null) {
        inOrder(n.getLeft());
        visit(n)
        inOrder(n.getRight());
    }
}
```

PreOrder Traversal

```
visit(n)
preOrder(n.leftChild)
preOrder(n.rightChild)
```



PostOrder Traversal

```
postOrder(n.leftChild)
postOrder(n.rightChild)
visit(n)
```