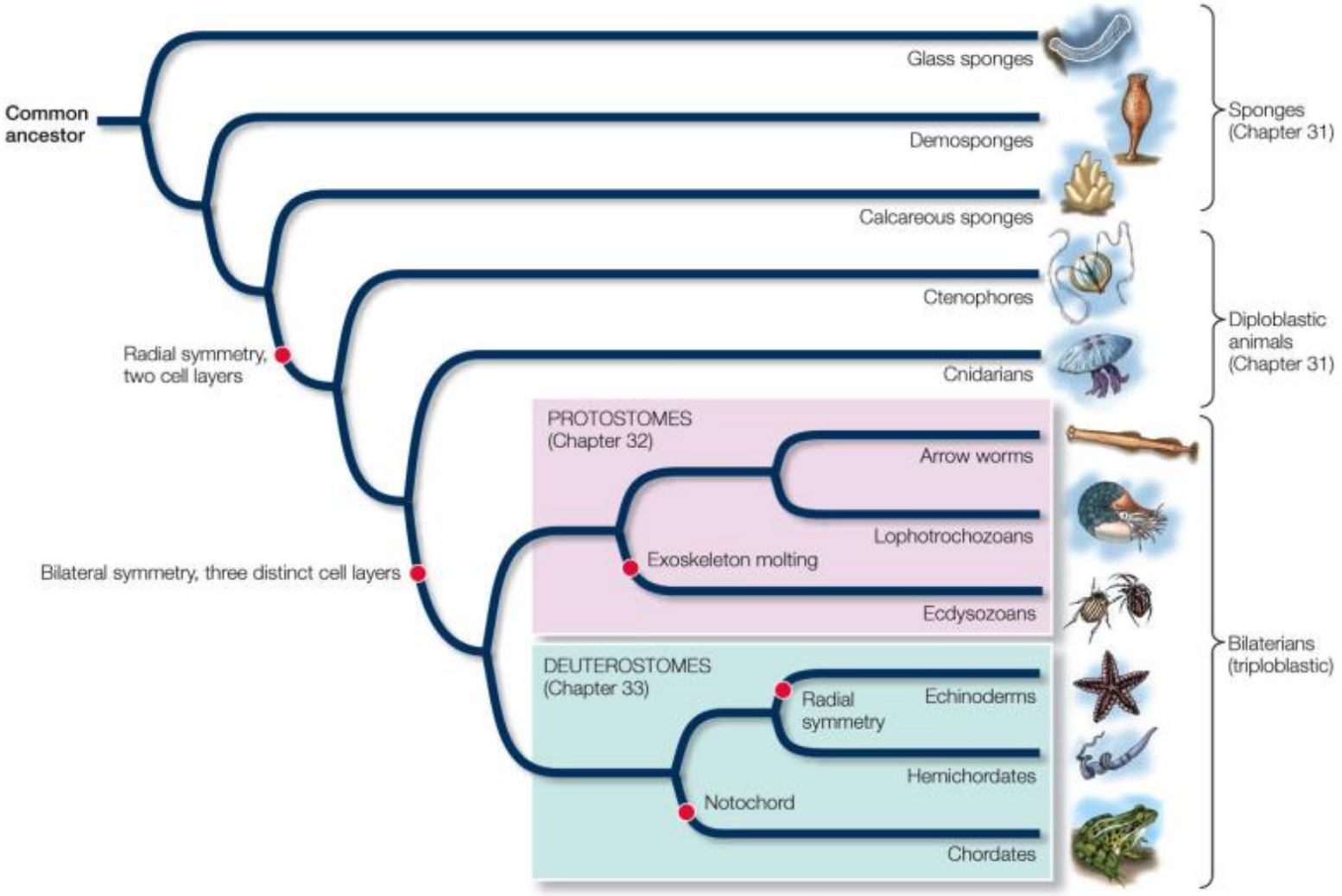
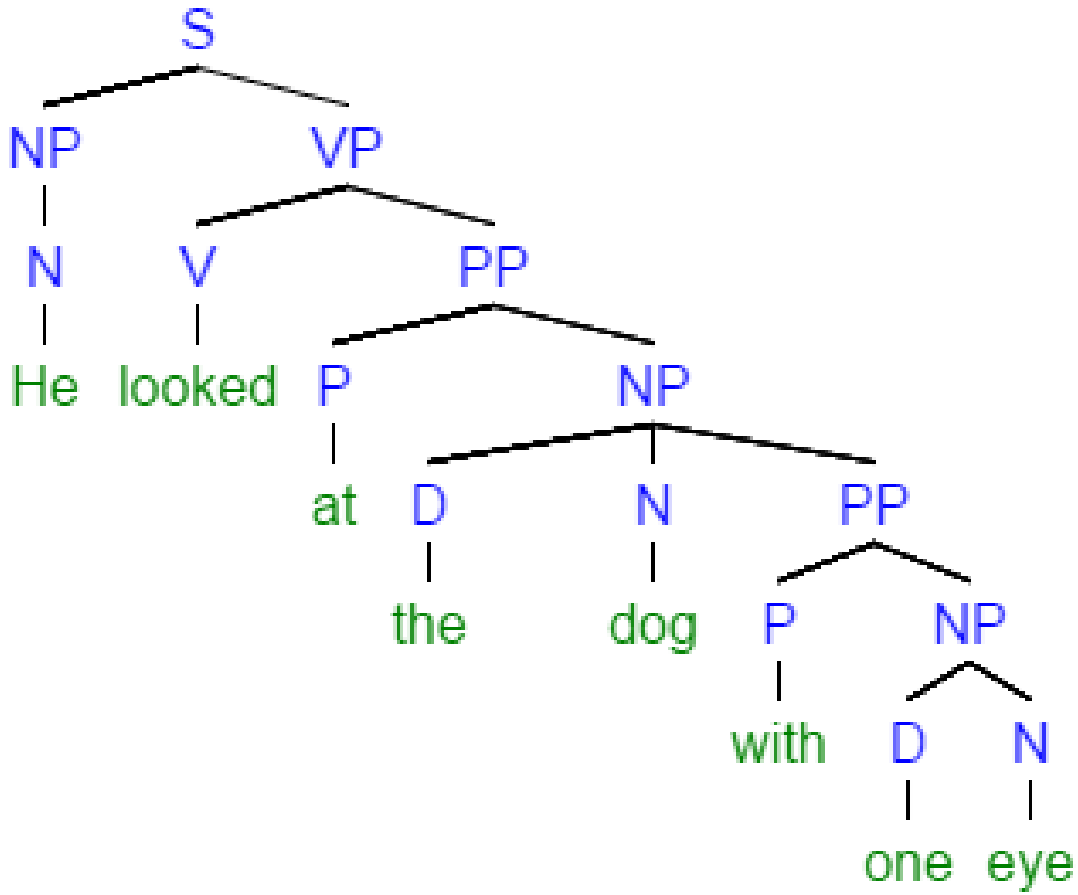
New Basic Data Structure: *Tree*

Lecture 02.10
by Marina Barsky

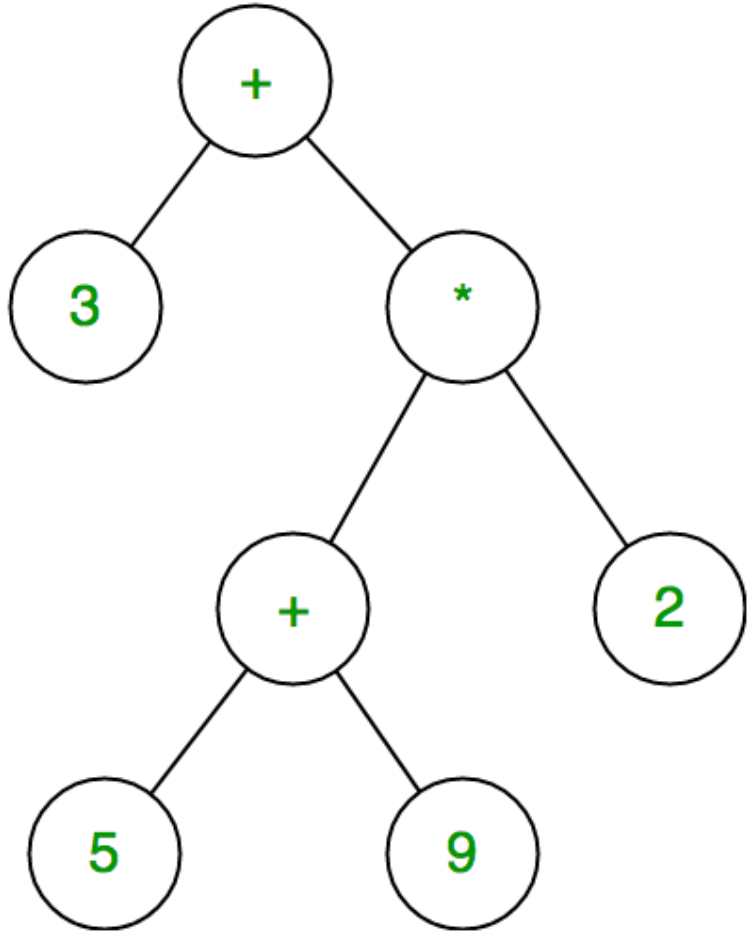
Phylogenetic tree of animals



Syntax Tree

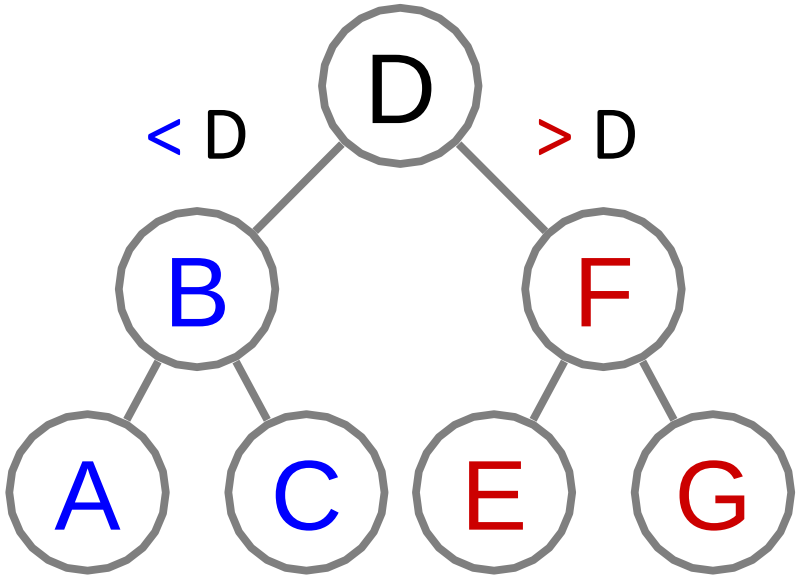


Expression tree



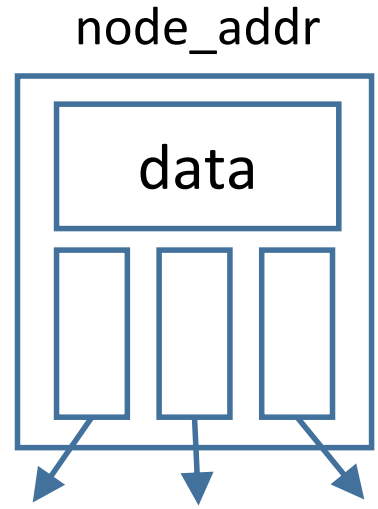
3 + ((5 + 9) * 2)

Binary Search Tree



Tree - new recursive data structure

- Main element of the tree: *node*
- Each node contains data and an **array** of links to the child nodes



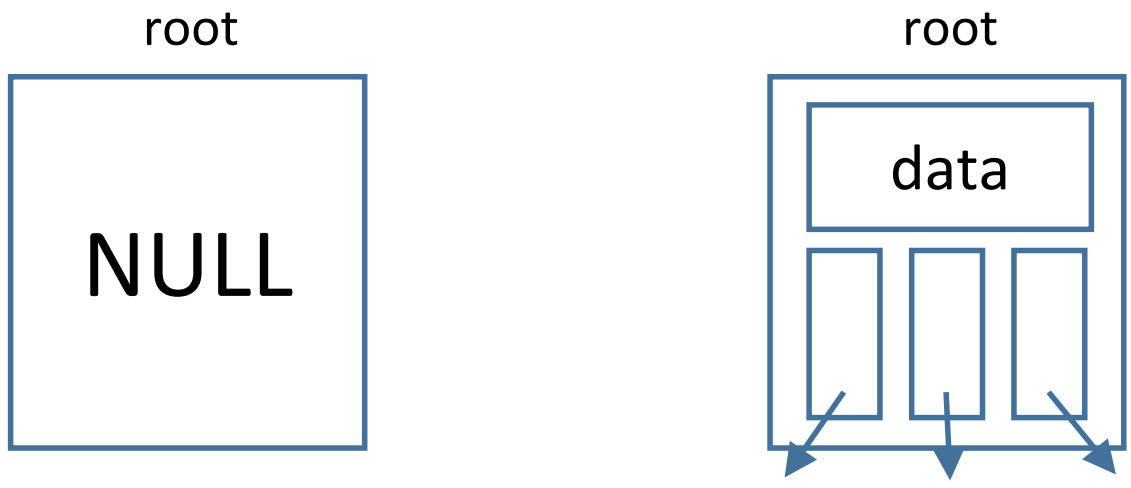

```
typedef struct node {
    int data;
    struct node ** children;
    [struct node * parent;]
} TreeNode;
```

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.children = []
        [self.parent = None]
```

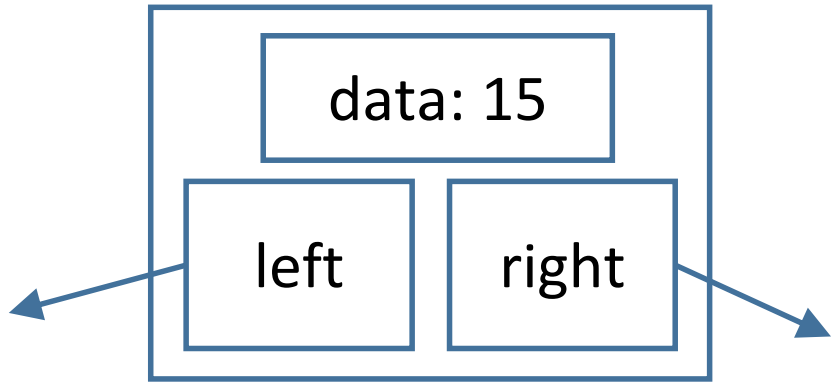
Tree is defined by a single node

Tree is either

- Null (empty tree)
- Root node which contains data and links to child nodes



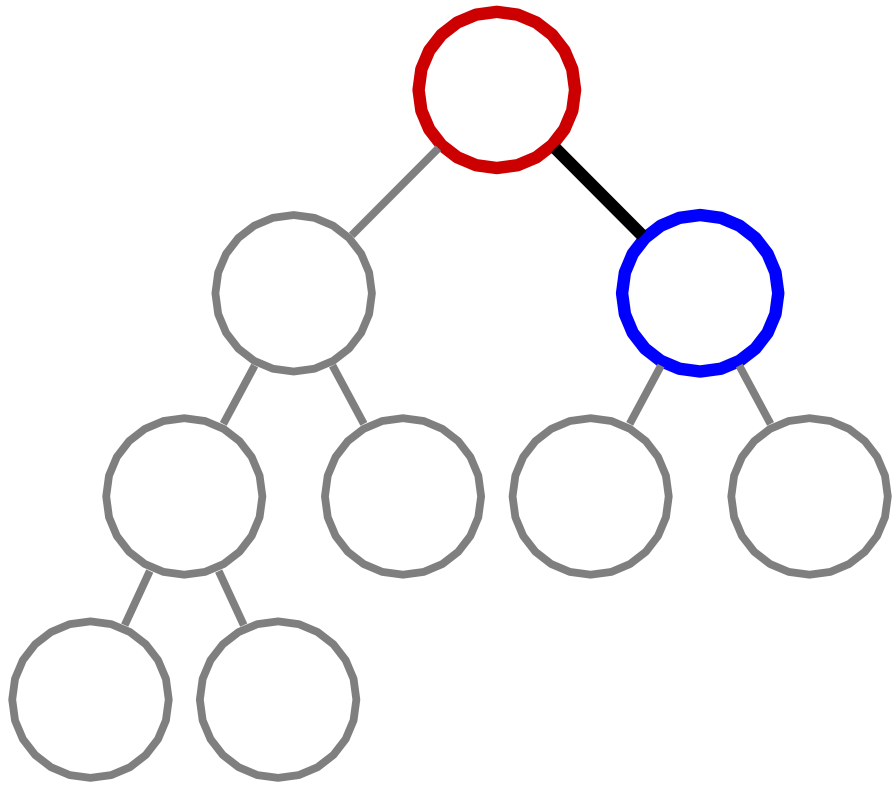
Binary tree: at most 2 children



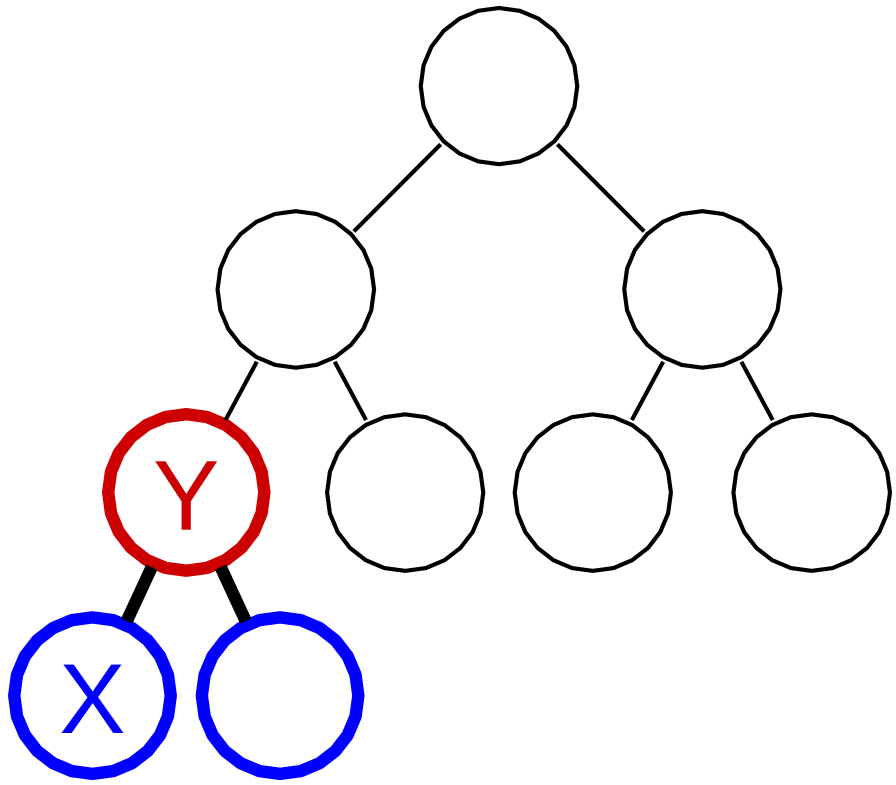

```
typedef struct node {
    int data;
    struct node * left;
    struct node * right;
    [struct node * parent;]
} TreeNode;
```

```
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        [self.parent = None]
```

Parent and child

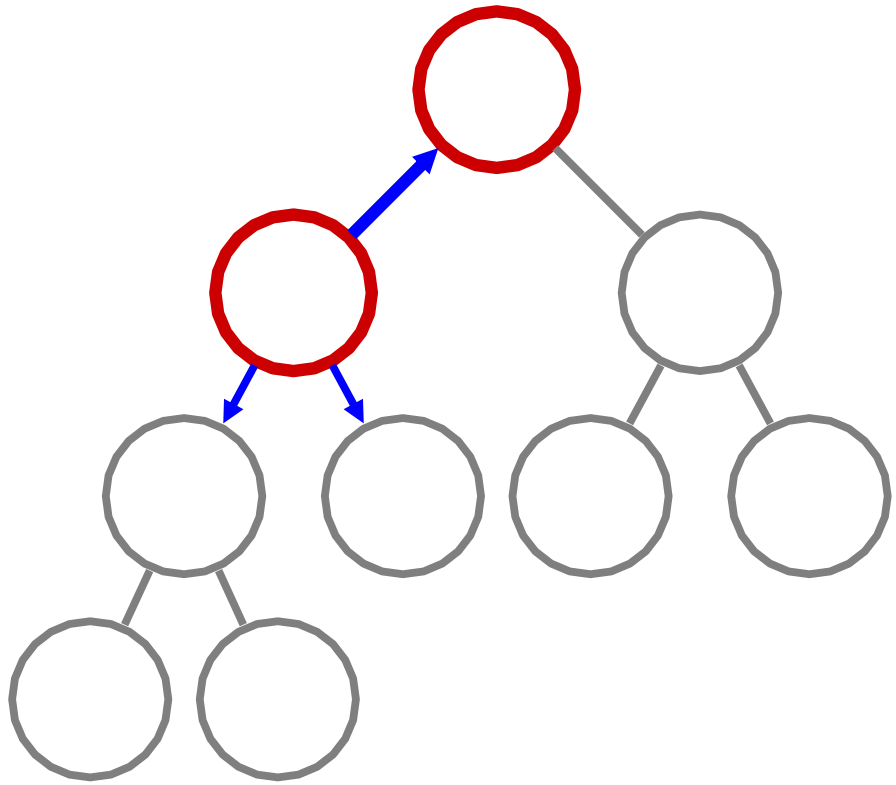


Parent and child



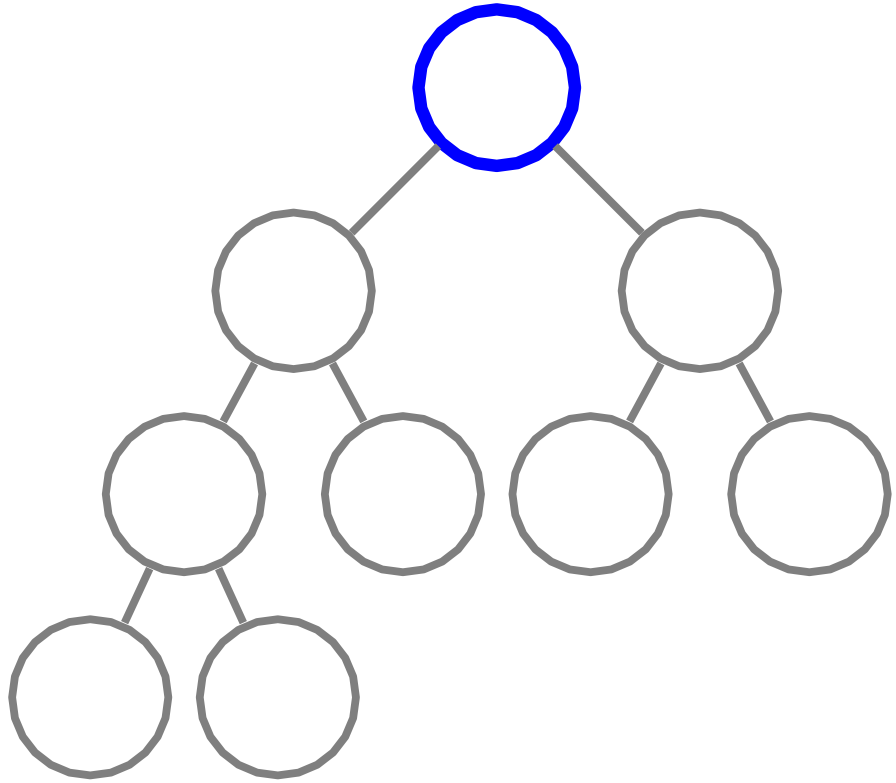
Have direct relationship

Node and edge



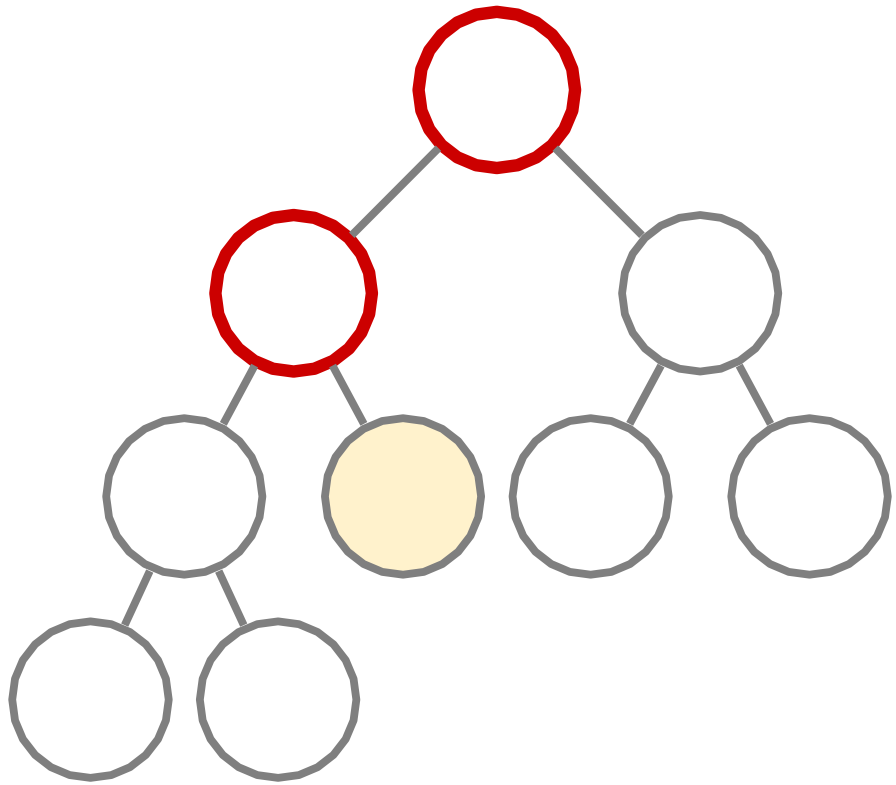
An edge connects nodes:
parent-child or child-parent relationships

Root



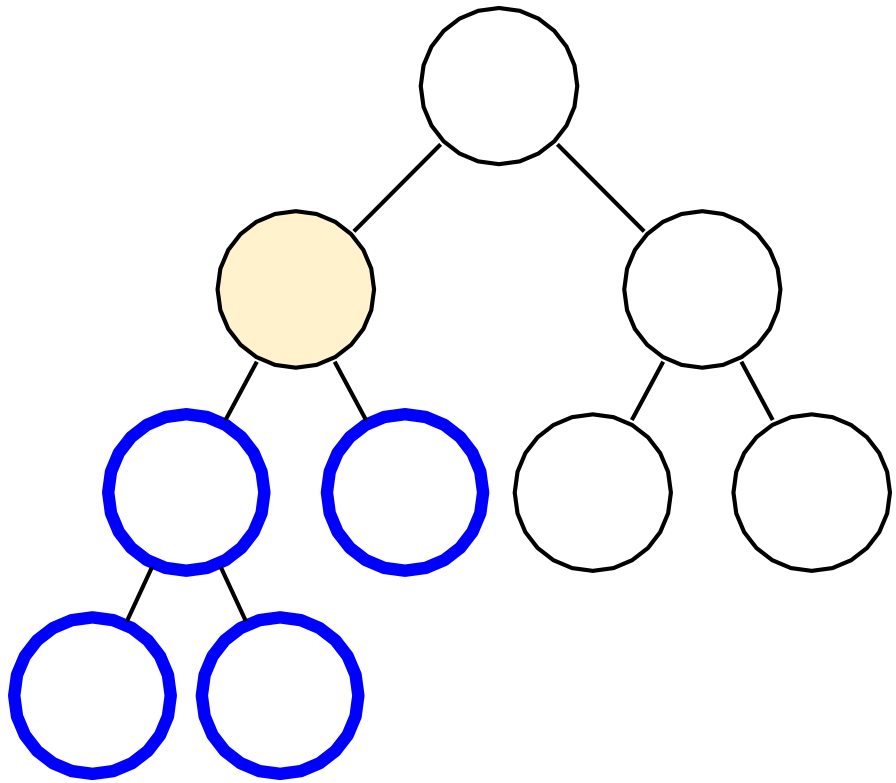
The parent of all nodes, the starting point

Ancestor and descendant



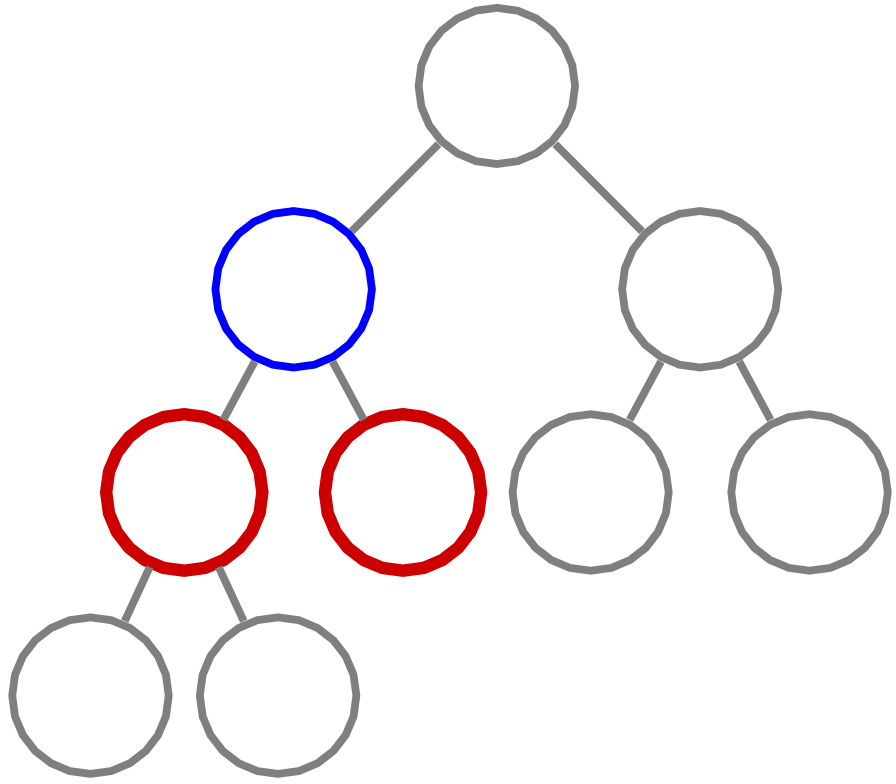
Parent, or parent of parent, etc.

Ancestor and descendant



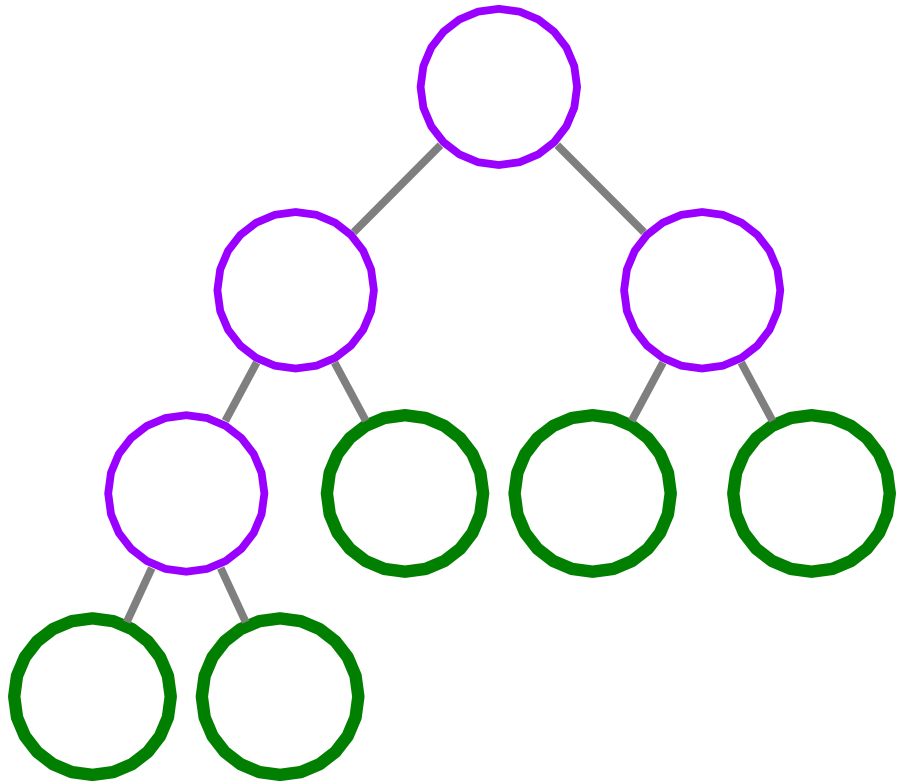
Child, or child of child, etc.

Siblings

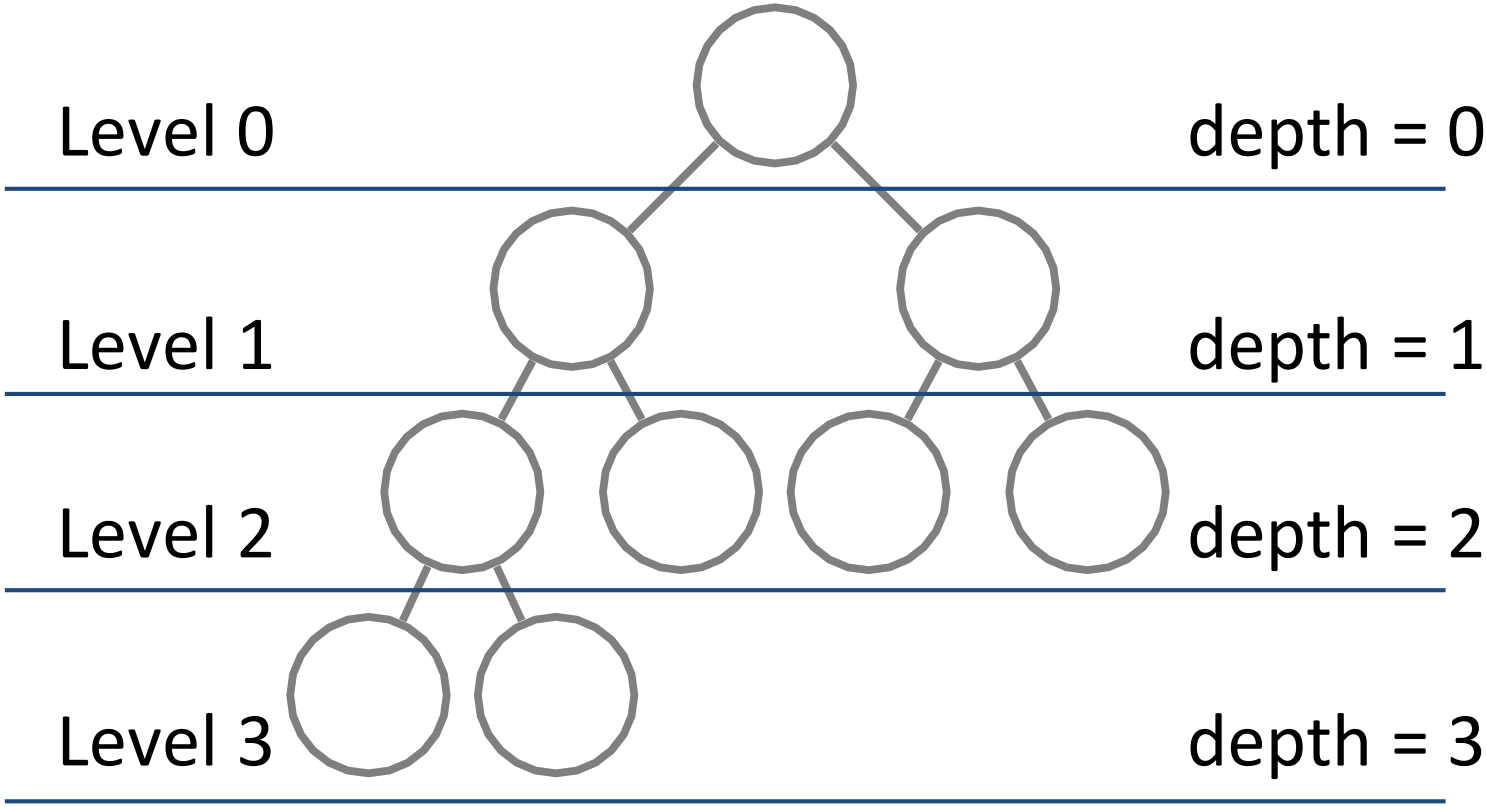


Sharing the same parent

Leafs and interior (internal) nodes

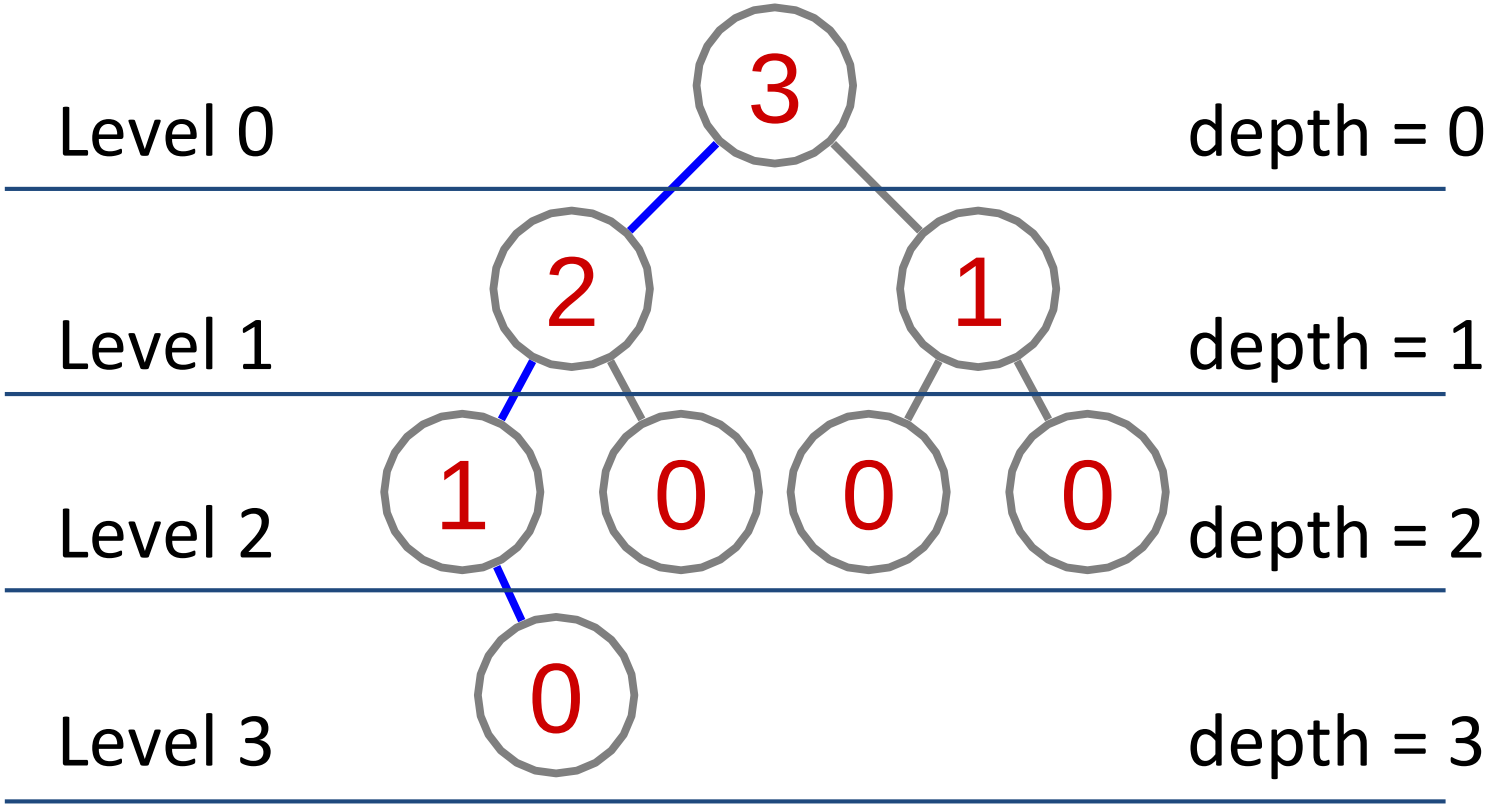


Tree levels and node depth



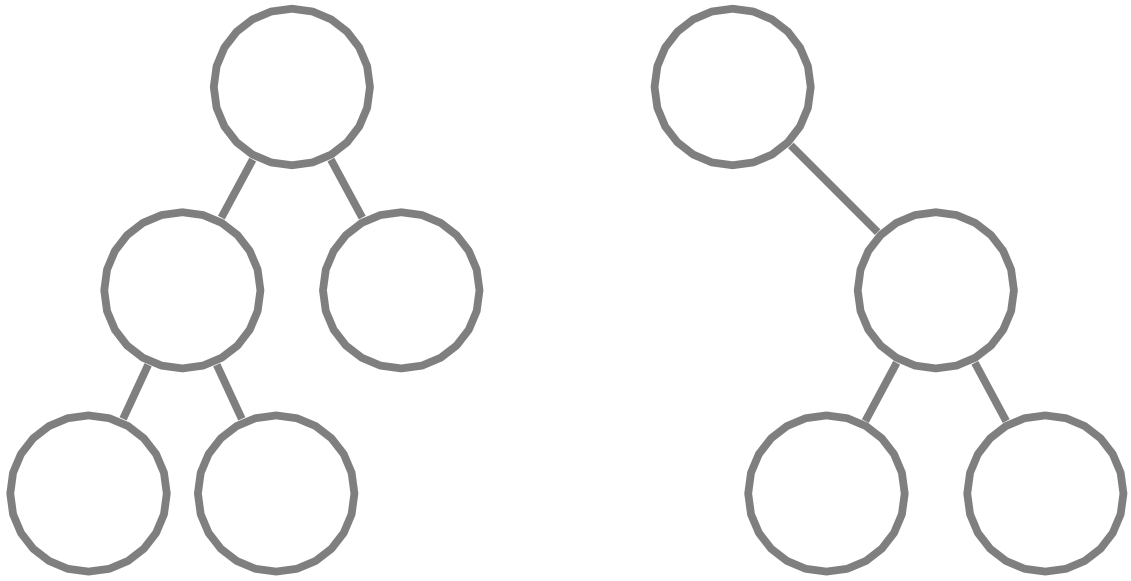
Distance from the root:
how many edges to go from a node to the root

Node height



Distance from the node to the bottom: how many edges to go to the furthest leaf

Forest



Collection of trees

Algorithm *Height* (*node*)

```
if node is Null :
   return 0
if node.left is Null and node.right is Null:
   return 0
return 1 + Max(Height(node.left),
                Height(node.right))
```

Recursive algorithms are common

Algorithm *Size* (*tree*)

```
if tree is Null
   return 0
return 1 + Size(tree.left) +
           Size(tree.right)
```

Tree traversals

- How do we list all the nodes in the tree?

- Two types of traversals:
  - *Depth-first*: we completely traverse one sub-tree before exploring a sibling sub-tree
  - *Breadth-first*: We traverse all nodes at one level before progressing to the next level

Depth-first tree traversals

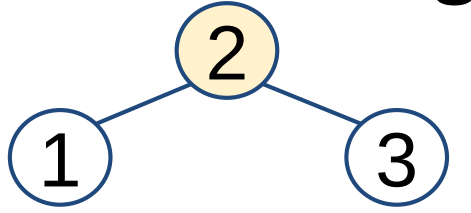
- In-order
- Pre-order
- Post-order

Depth-first

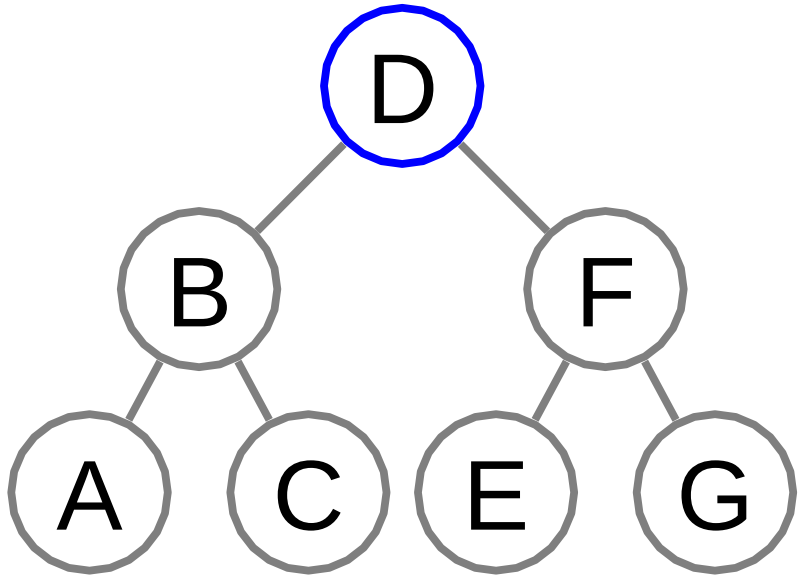
Algorithm ***InOrder****Traversal*(*tree*)

```
if tree = Null :
   return
InOrderTraversal(tree.left)
print (tree.key )
InOrderTraversal(tree.right)
```

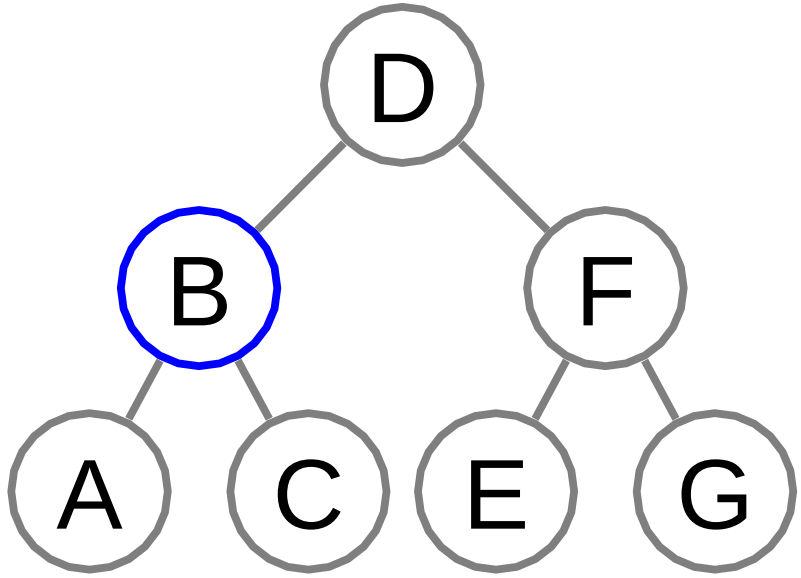
left - me - right



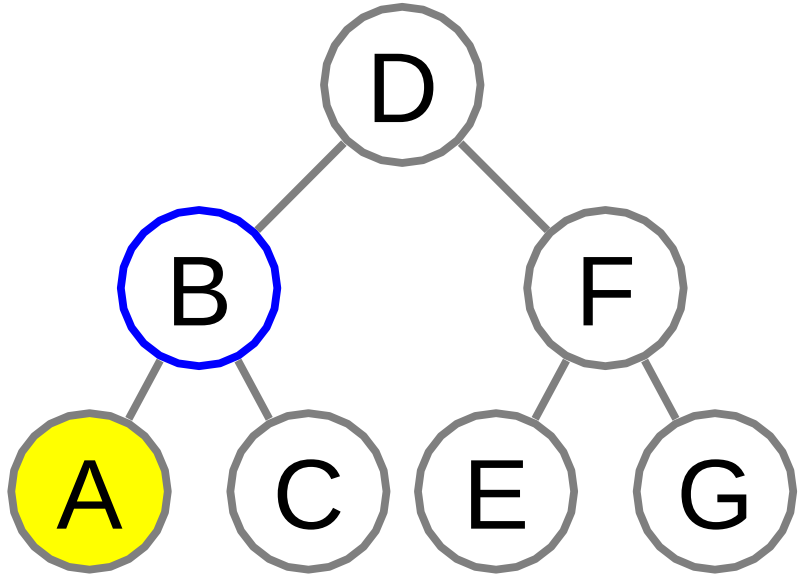
In-order



In-order

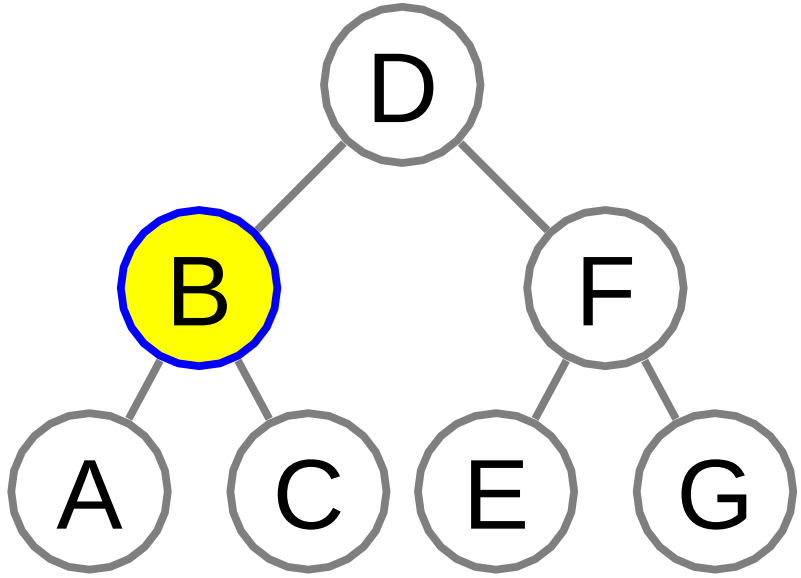


In-order



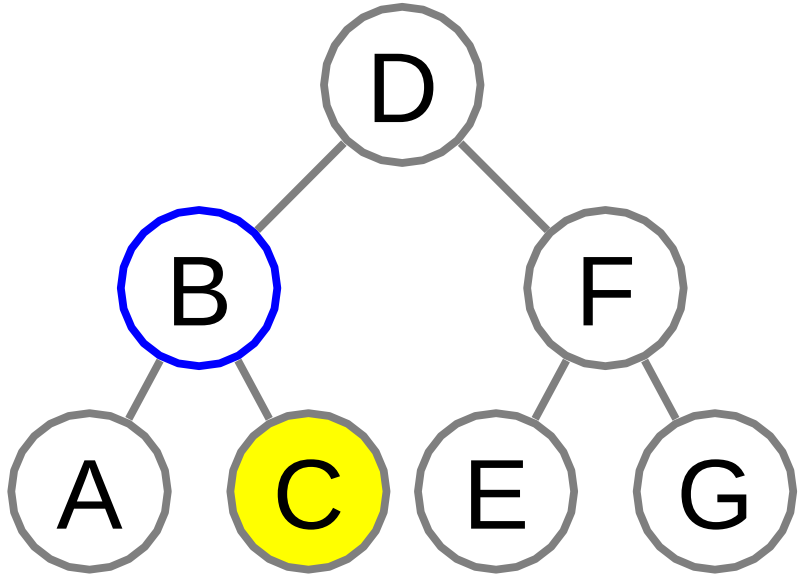
A

In-order



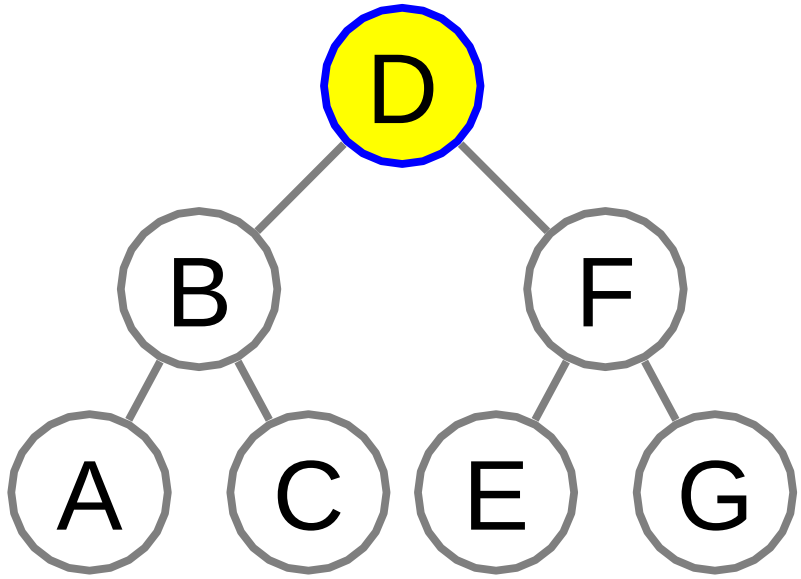
A B

In-order



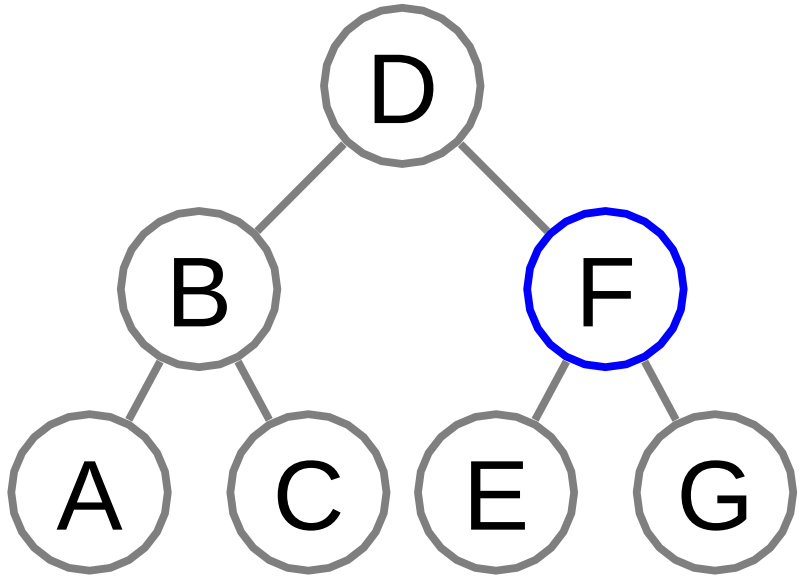
A B C

In-order



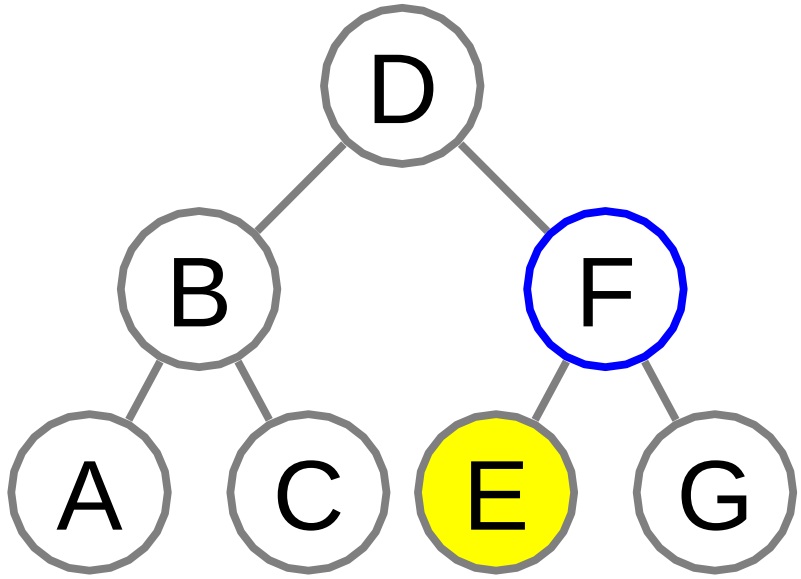
A B C D

In-order



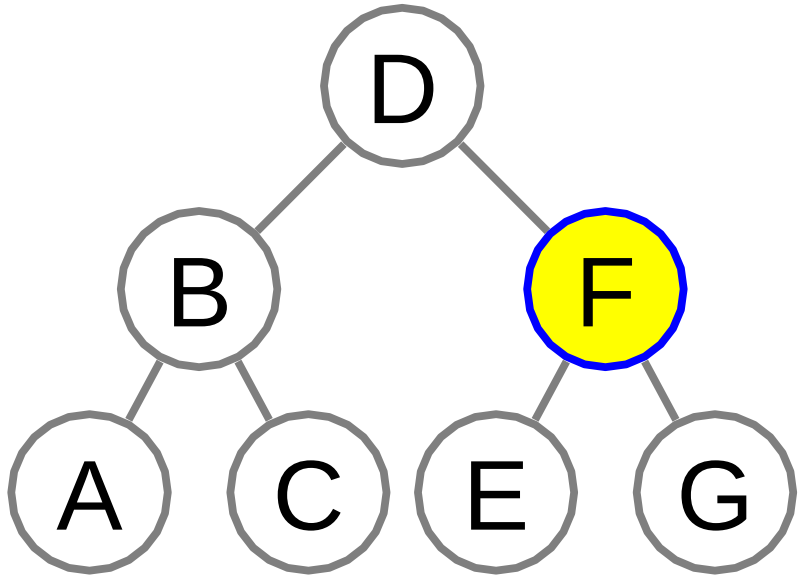
A B C D

In-order



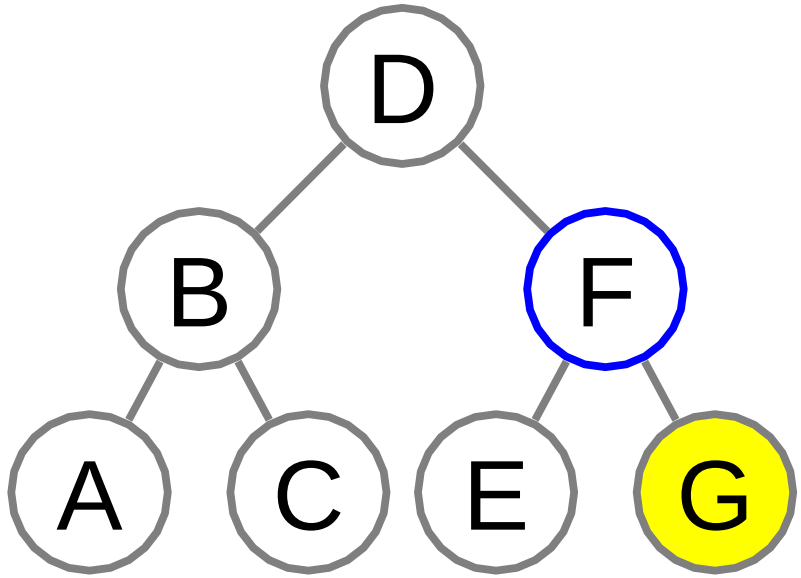
A B C D E

In-order



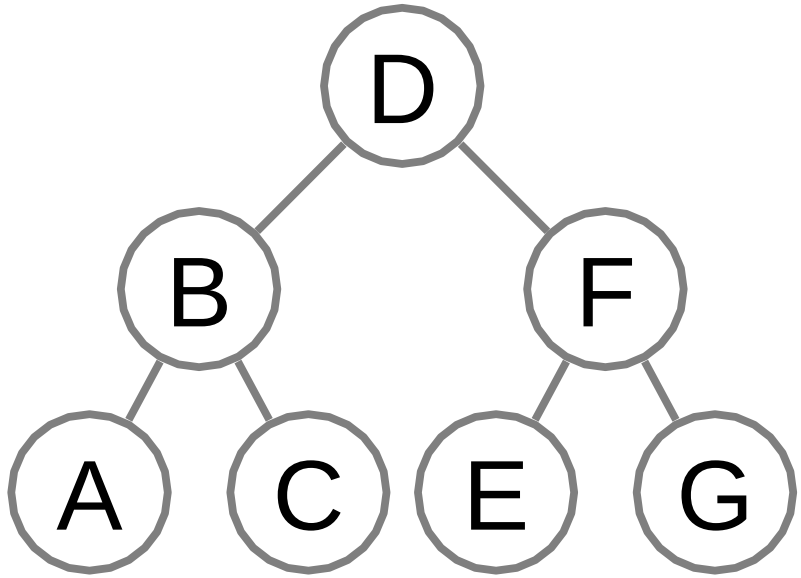
A B C D E F

In-order



A B C D E F G

In-order



me, node D

A B C D

left subtree of D E F right subtree of D G

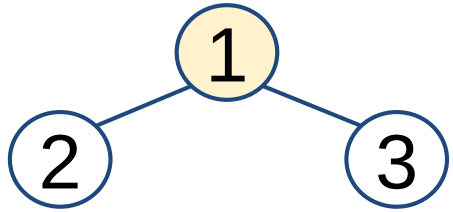
Depth-first

Algorithm ***PreOrder****Traversal*(*tree*)

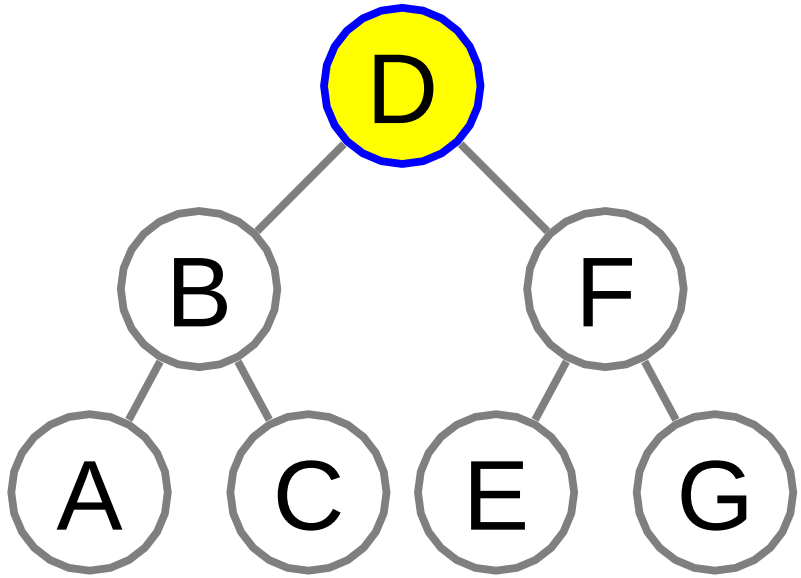
```
if tree is null:
    return
print (tree.key )
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

me first

me - left -right

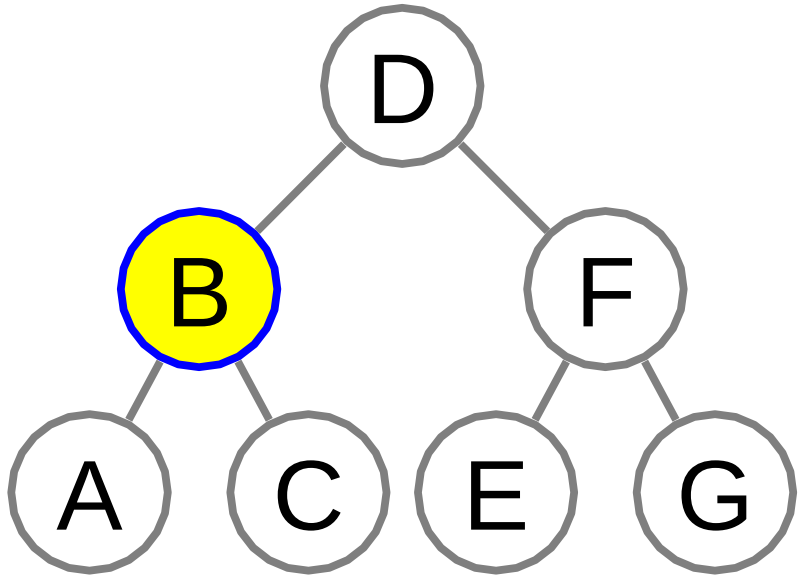


Pre-order



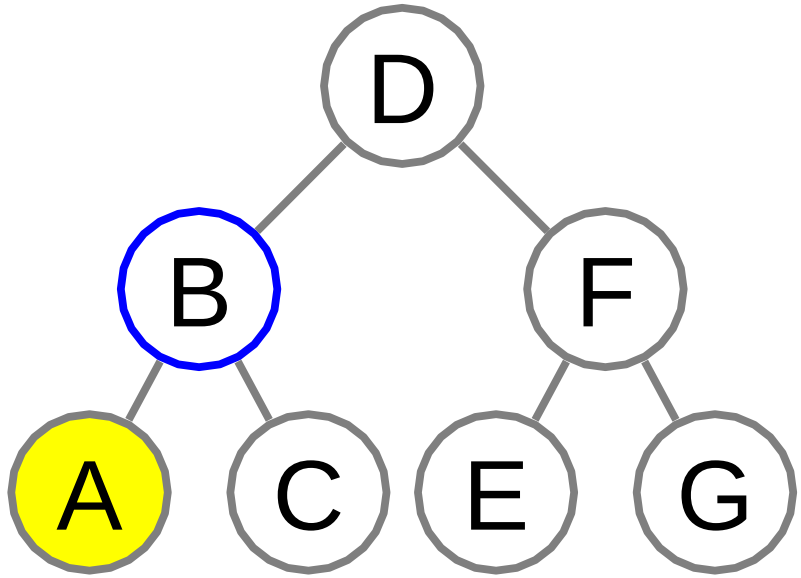
D

Pre-order



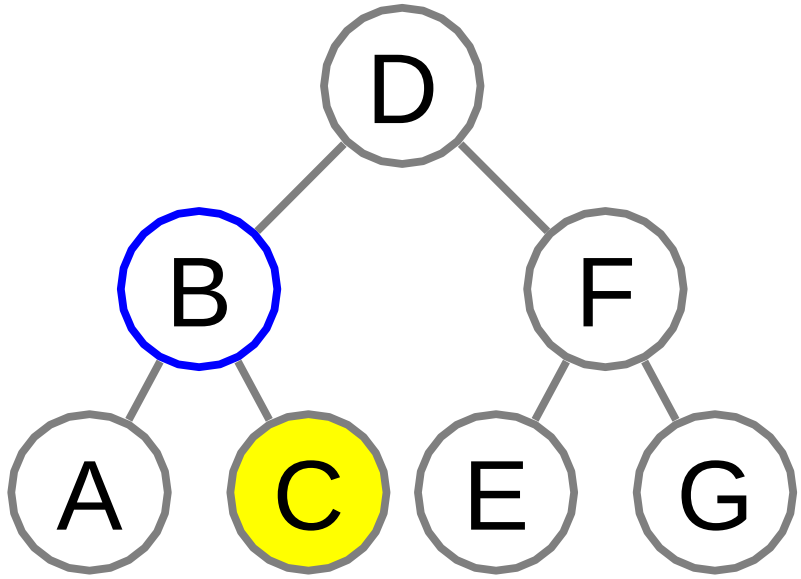
D B

Pre-order



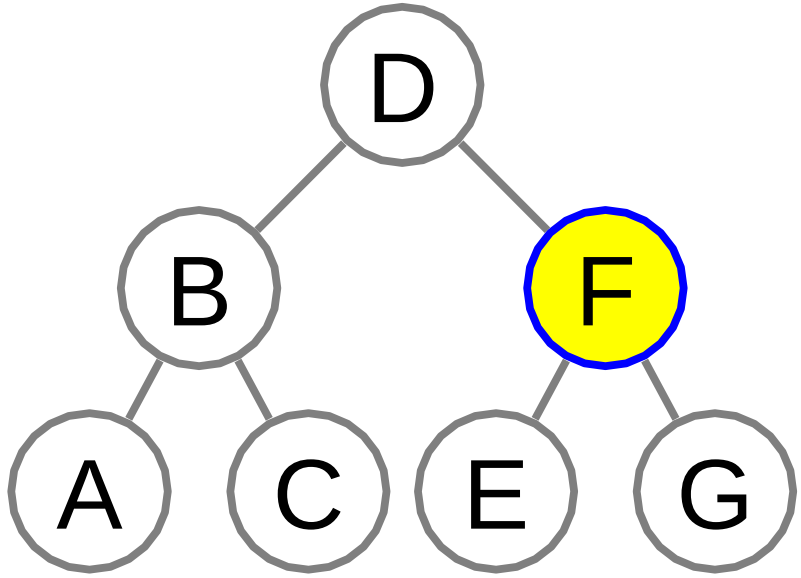
D B A

Pre-order



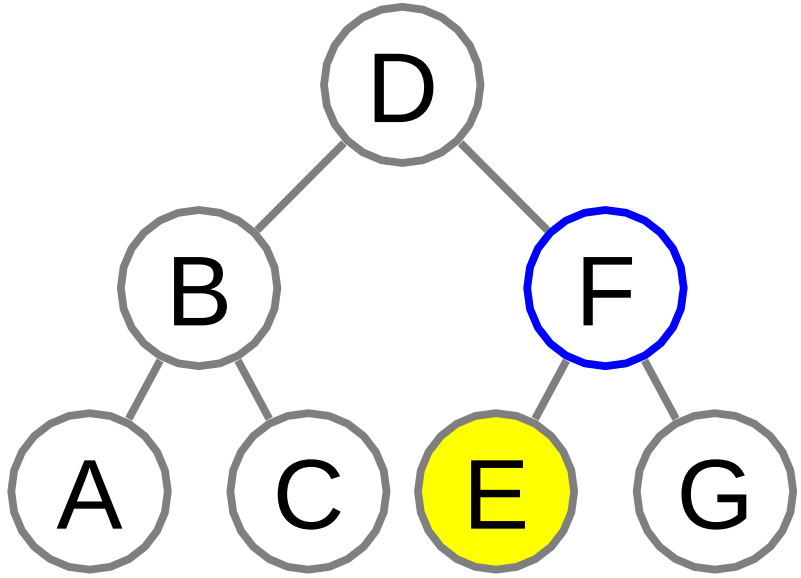
D B A C

Pre-order



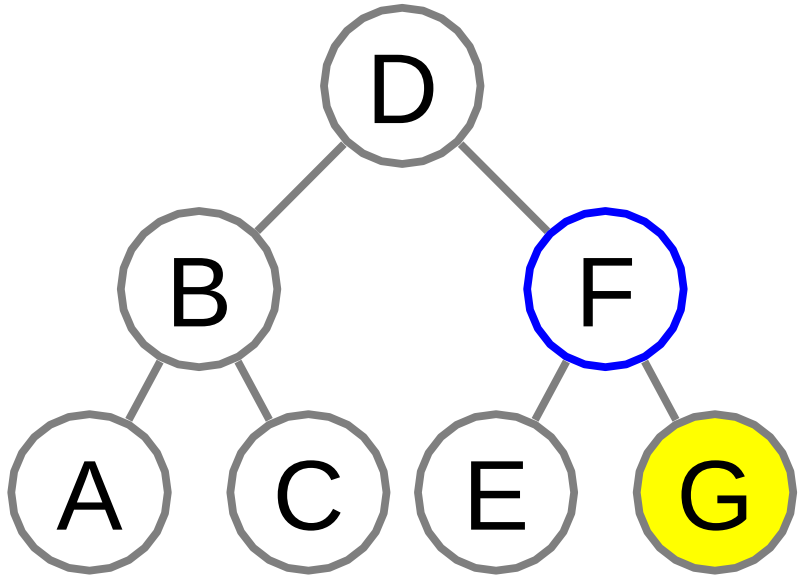
D B A C F

Pre-order



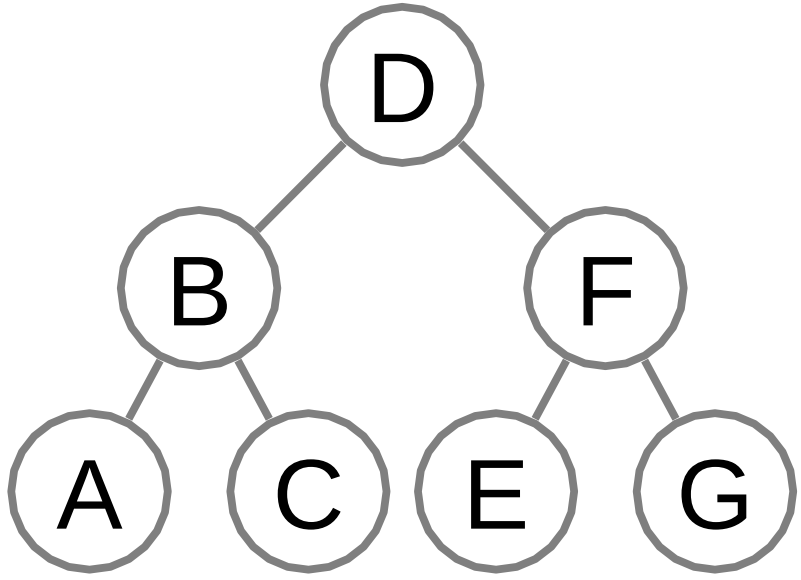
D B A C F E

Pre-order



D B A C F E G

Pre-order



me, node D

D B A C F E G

left subtree of D

right subtree of D

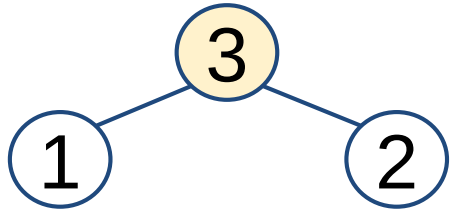
Depth-first

Algorithm *PostOrderTraversal*(*tree*)

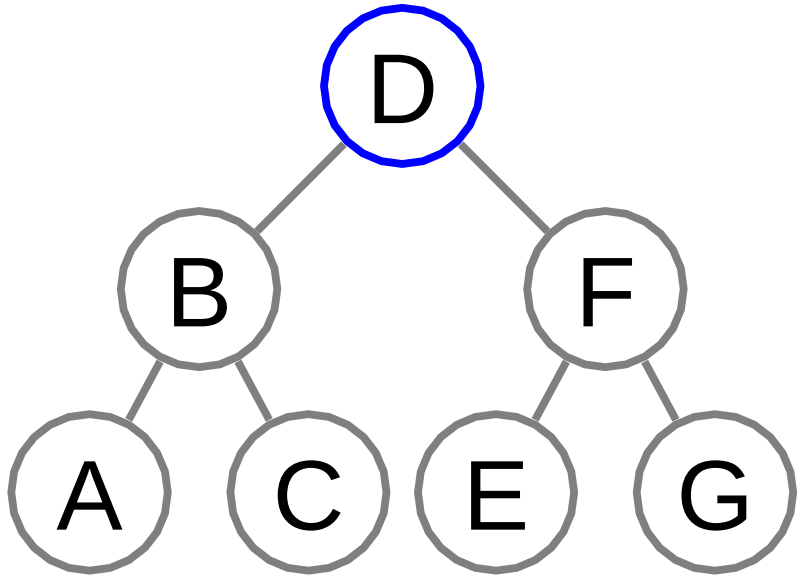
```
if tree is null:
   return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
print(tree.key )
```

children first

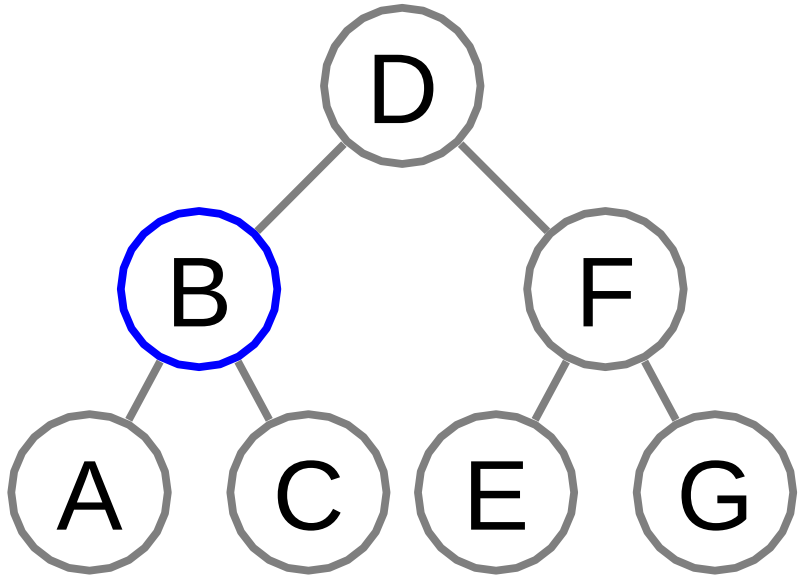
left-right-me



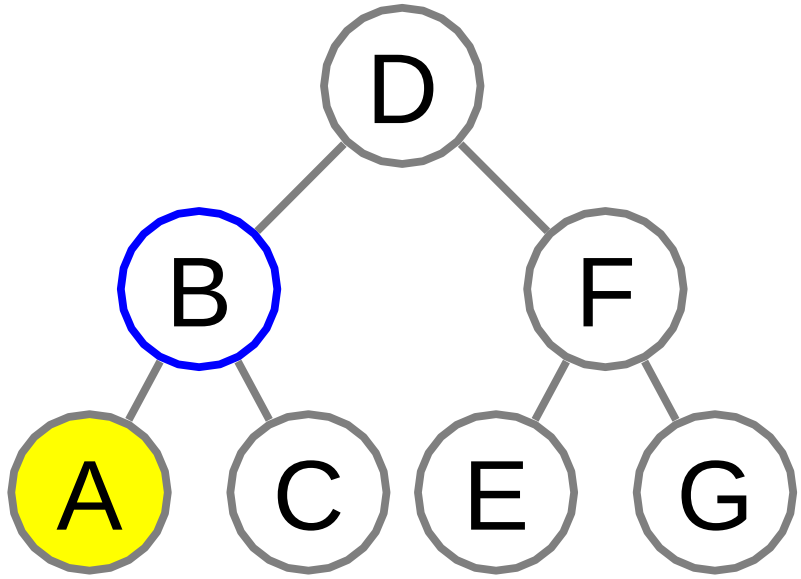
Post-order



Post-order

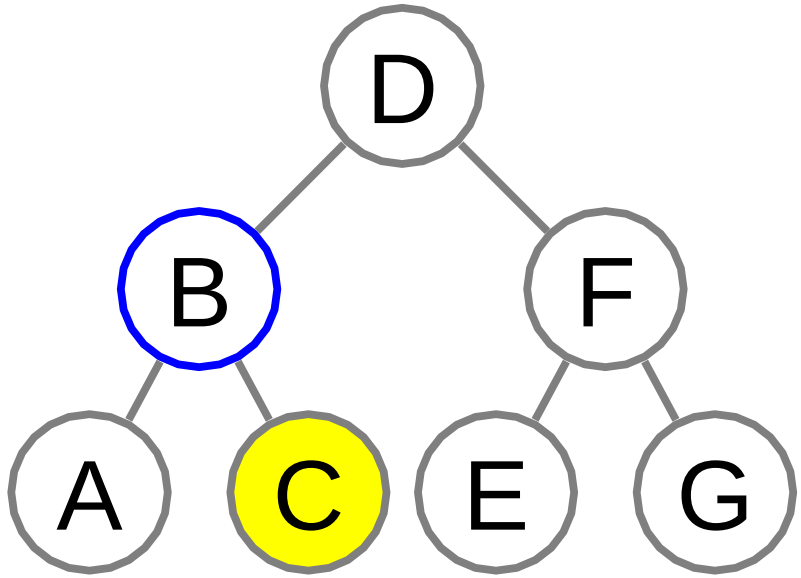


Post-order



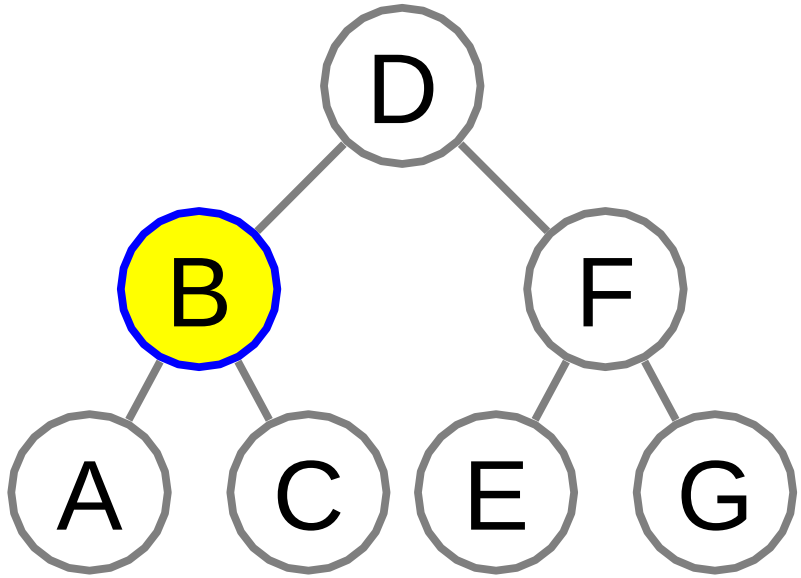
A

Post-order



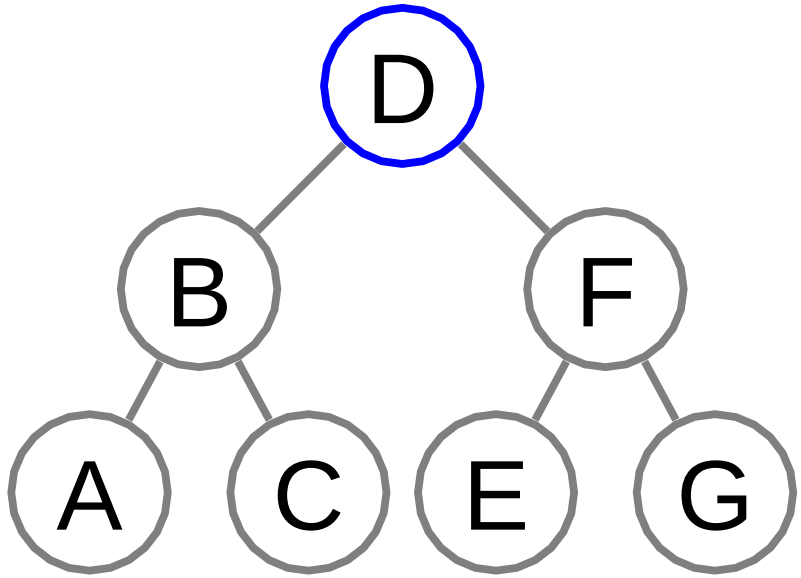
A C

Post-order



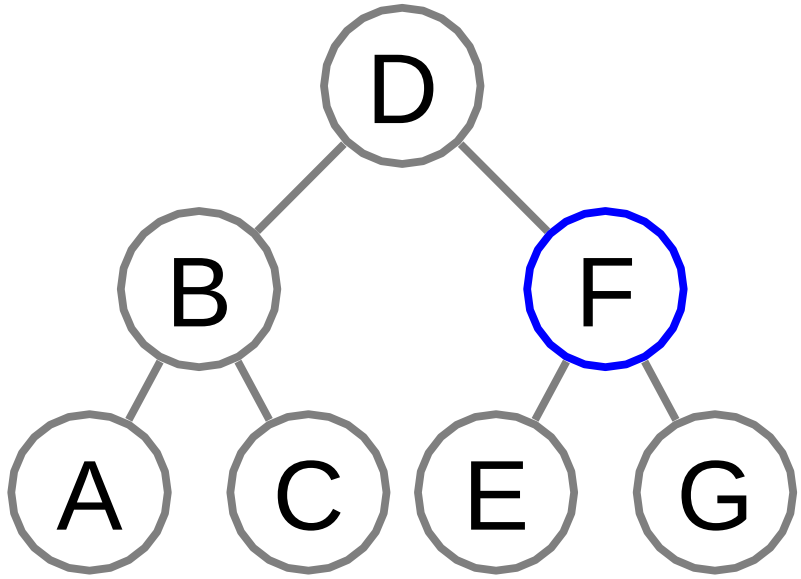
A C B

Post-order



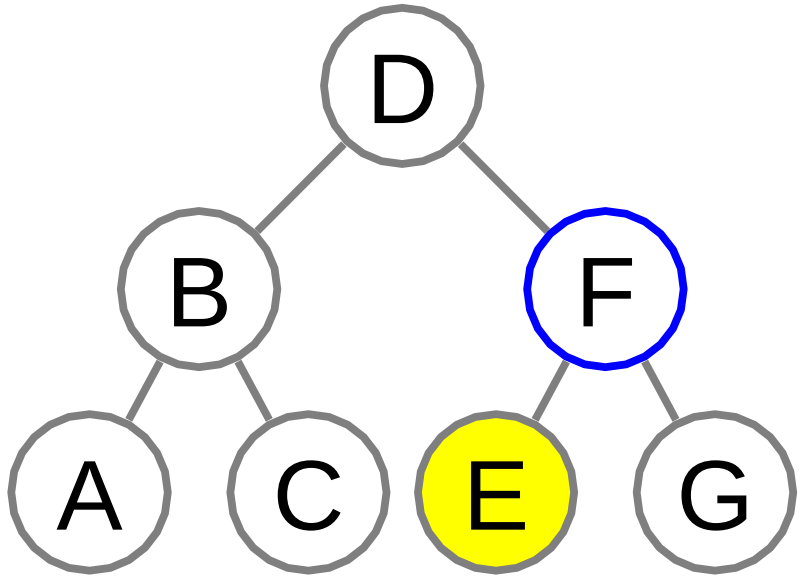
A C B

Post-order



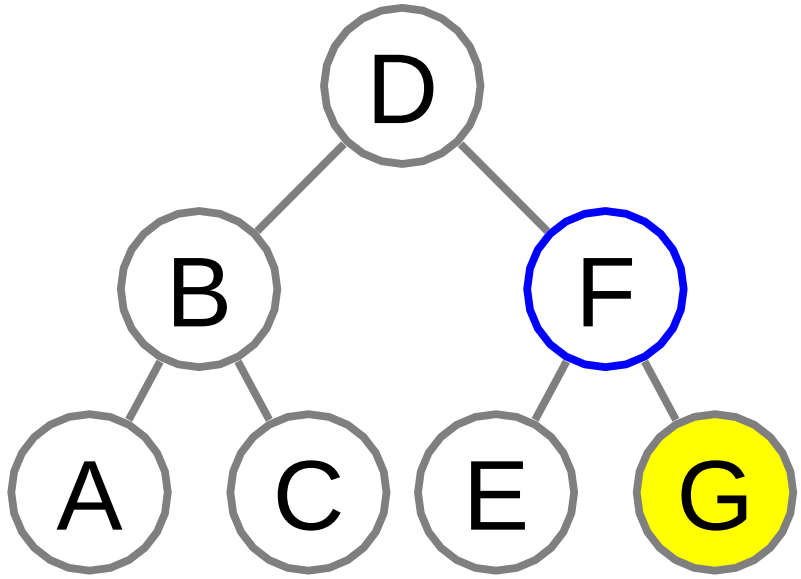
A C B

Post-order



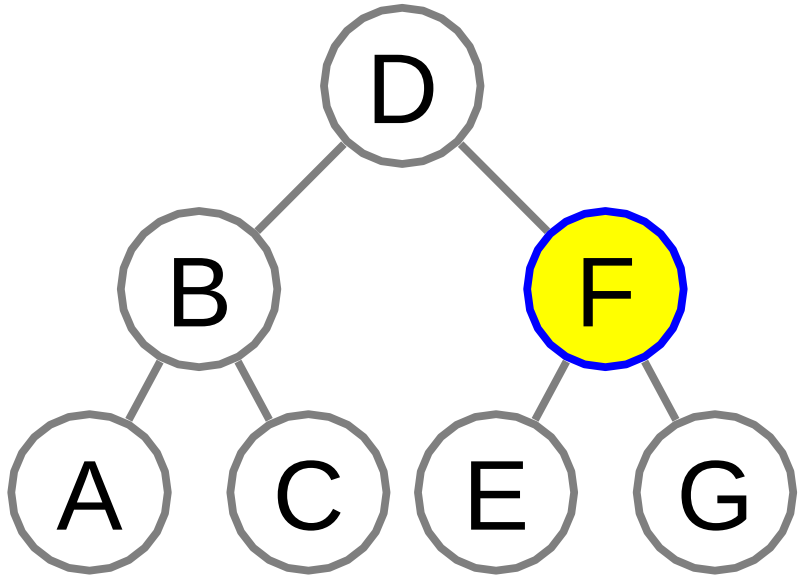
A C B E

Post-order



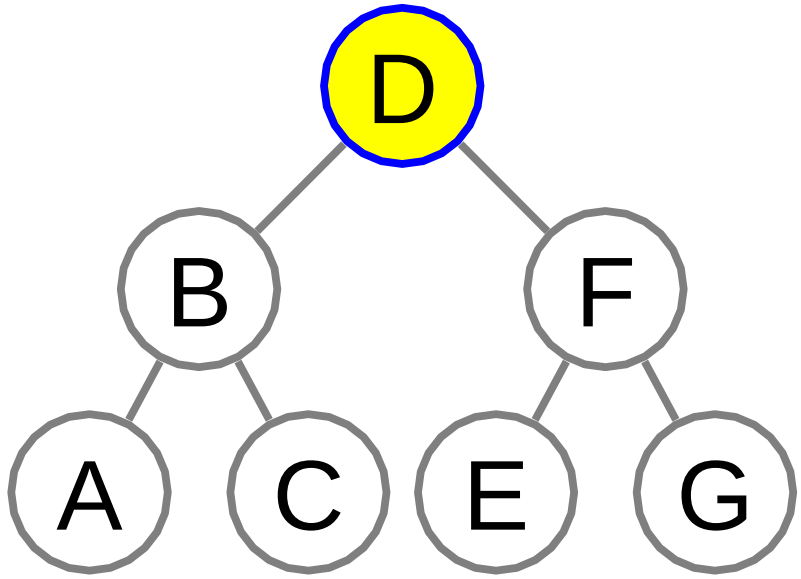
A C B E G

Post-order



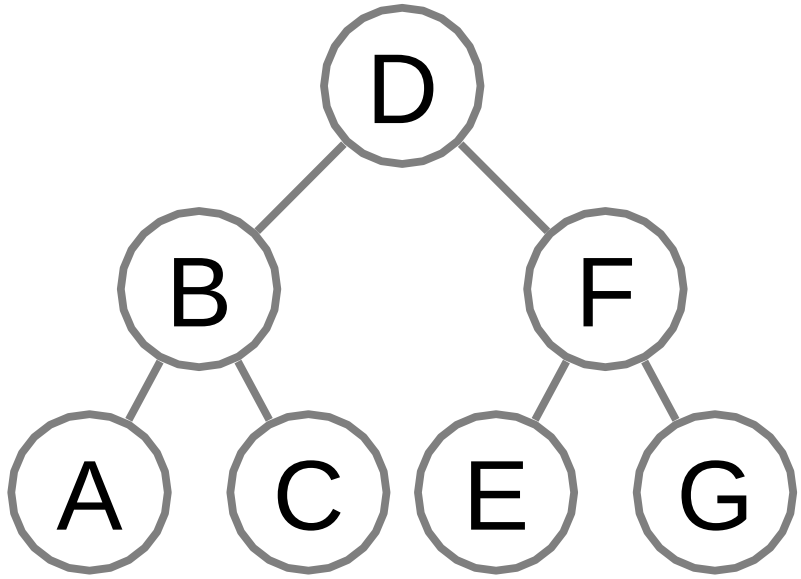
A C B E G F

Post-order



A C B E G F D

Post-order



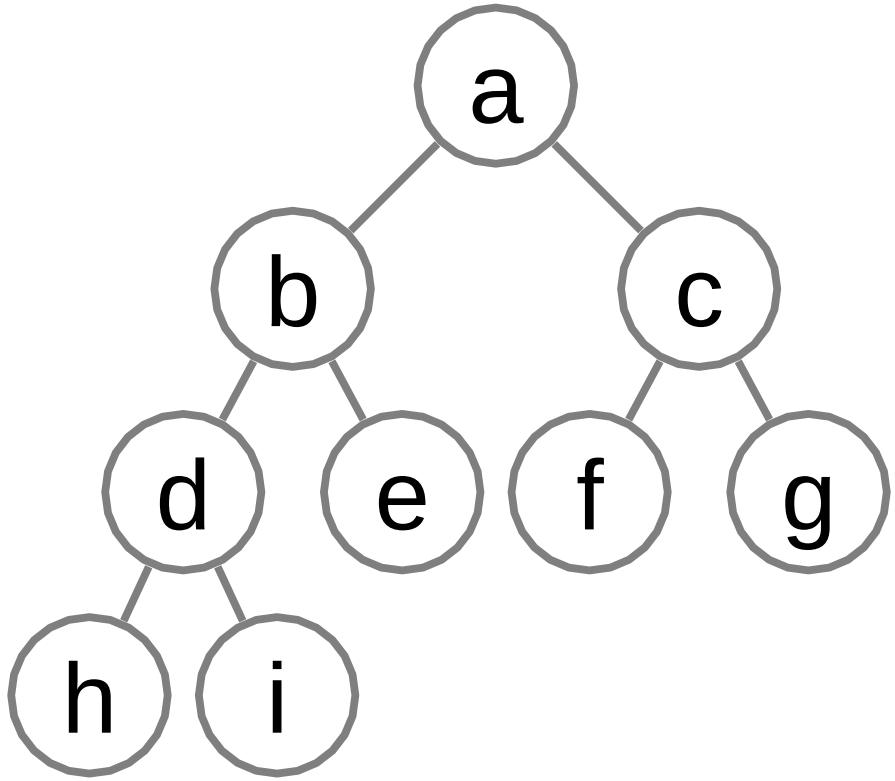
me, node D

A C B E G F D

left subtree of D

right subtree of D

Try it out

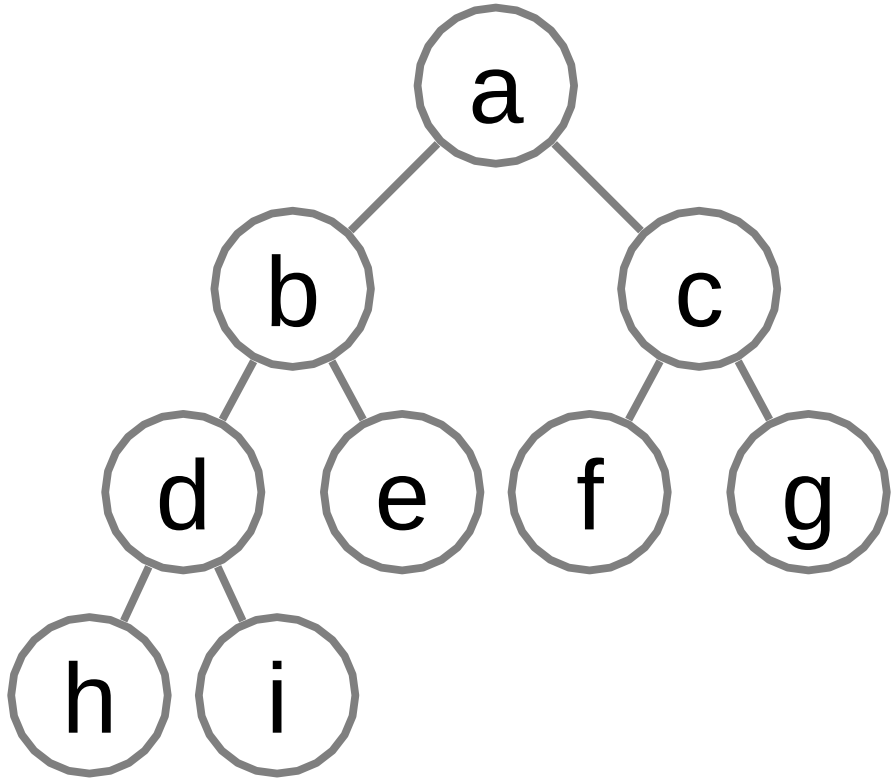


In-order:

Pre-order:

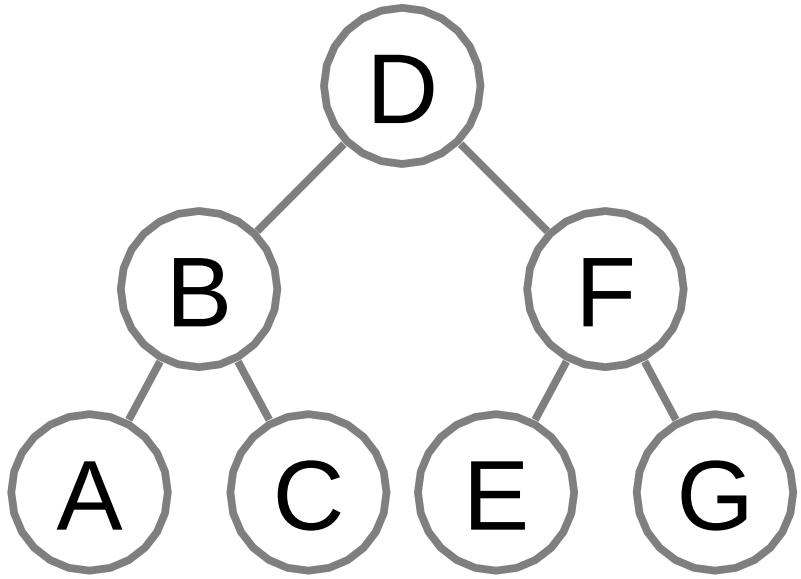
Post-order:

Solution



In-order: h d i b e a f c g
Pre-order: a b d h i e c f g
Post-order: h i d e b f g c a

Breadth first



Level traversal:

D

B F

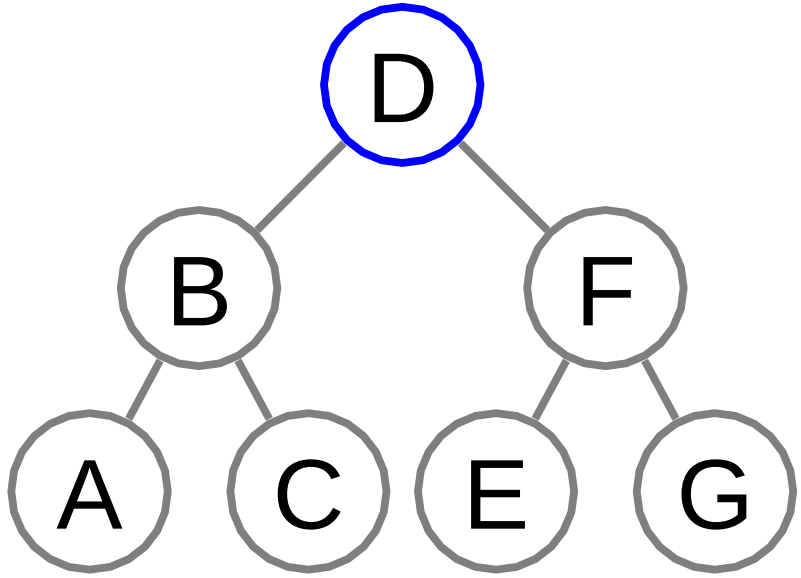
A C E G

Algorithm ***BreadthFirst****Traversal*(*tree*)

```
if tree is null:
   return

Queue q
q.Enqueue(tree)
while not q.Empty() :
   node ← q.Dequeue()
   Print(node)
   if node.left != null:
      q.Enqueue(node.left)
   if node.right != null:
      q.Enqueue(node.right)
```

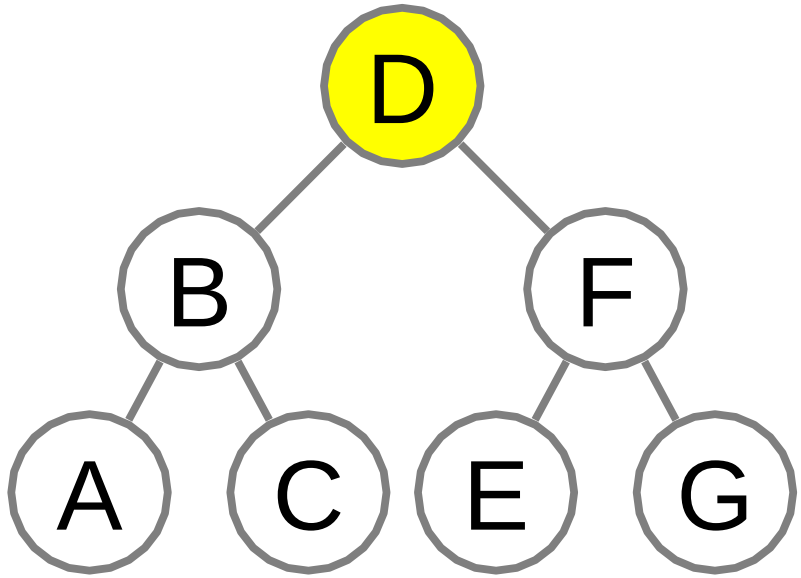
Breadth first: level traversal



Queue: D

Output:

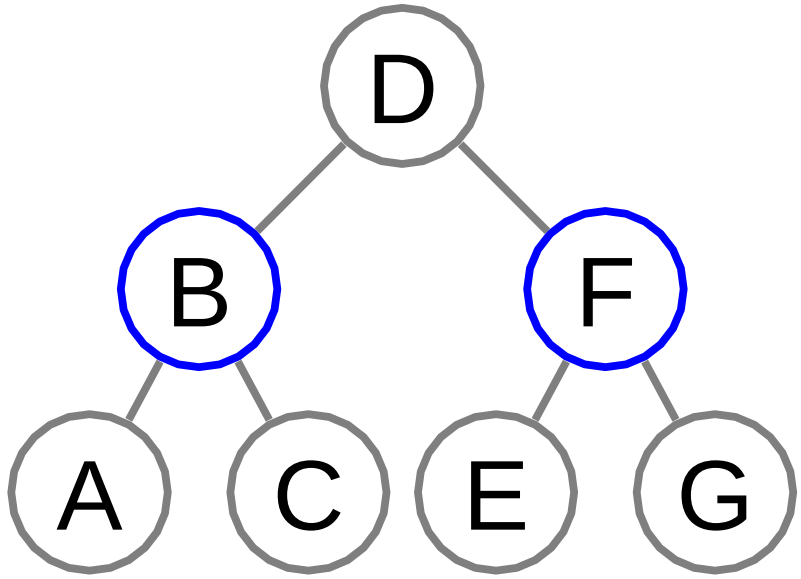
Breadth first: level traversal



Queue:

Output: D

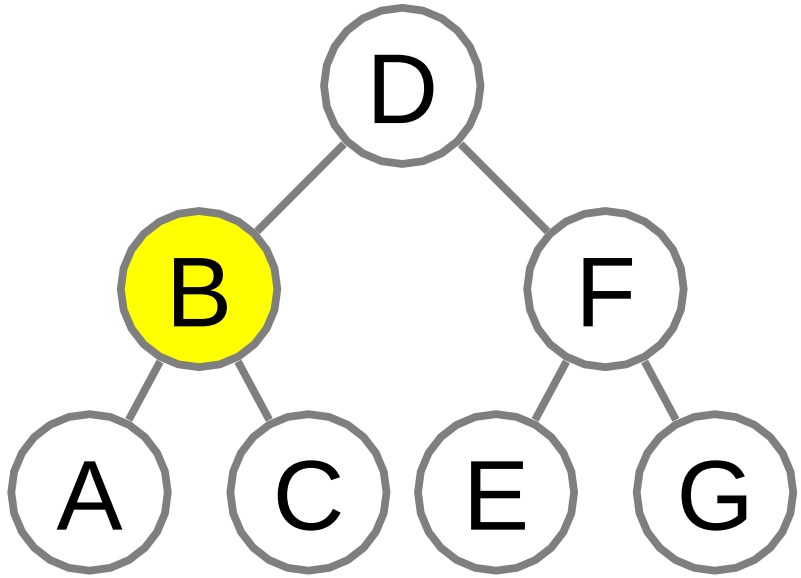
Breadth first: level traversal



Queue: B F

Output: D

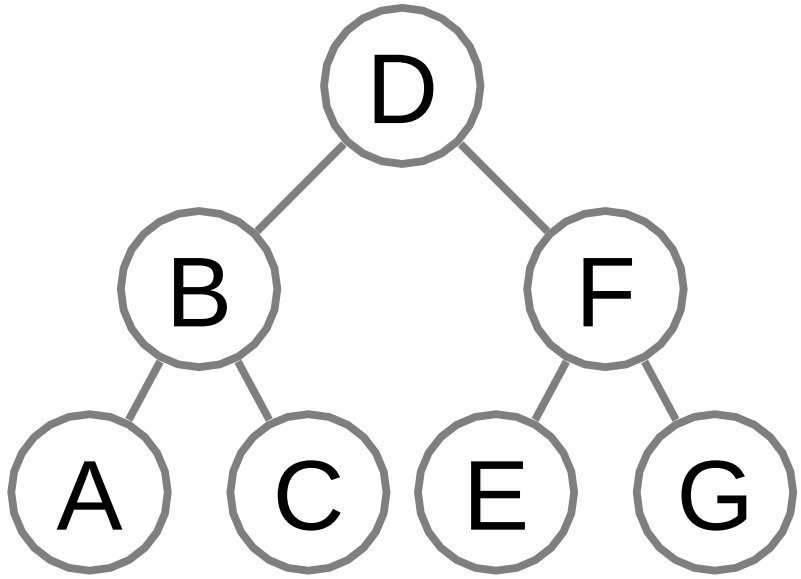
Breadth first: level traversal



Queue: B F

Output: D

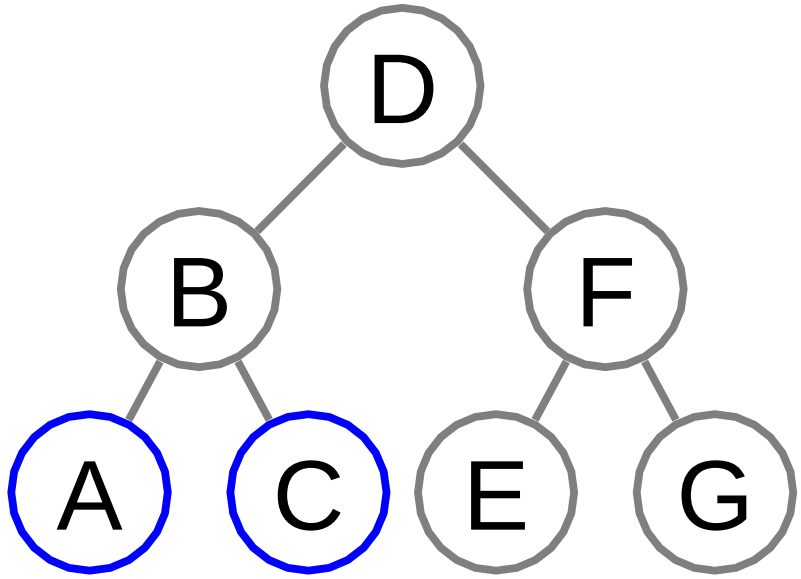
Breadth first: level traversal



Queue: F

Output: D B

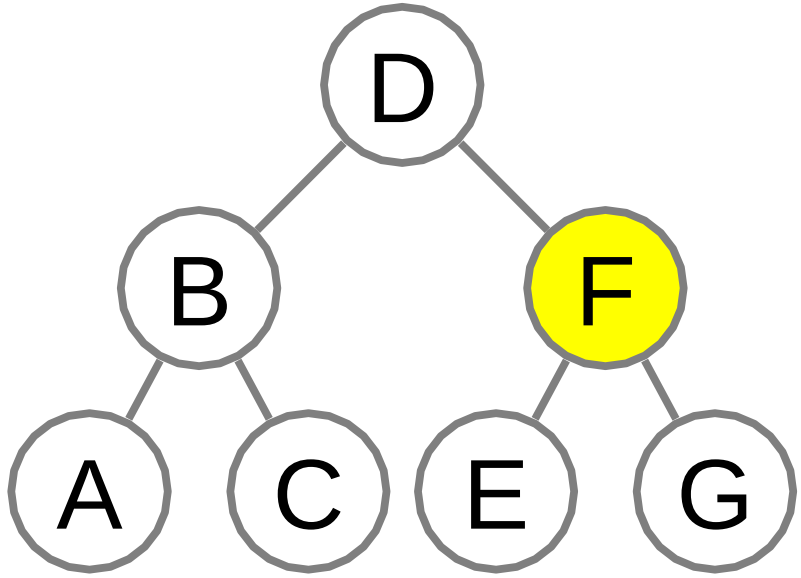
Breadth first: level traversal



Queue: F A C

Output: D B

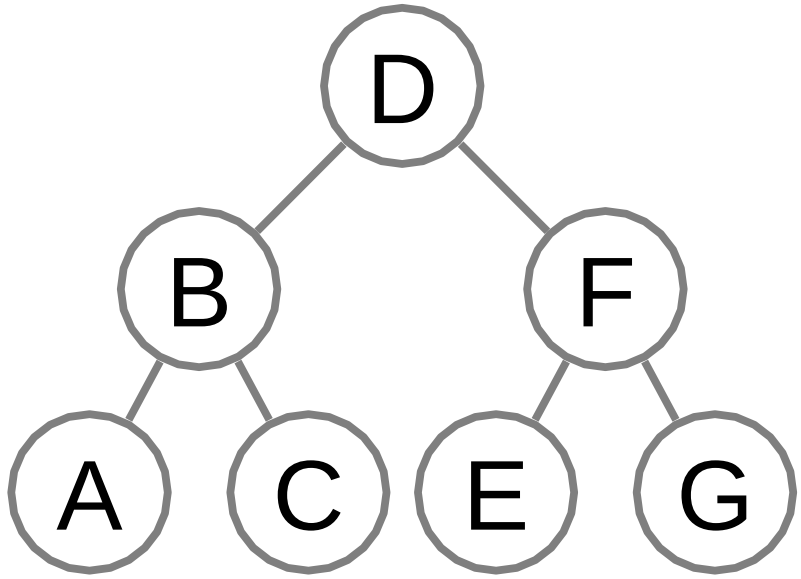
Breadth first: level traversal



Queue: F A C

Output: D B

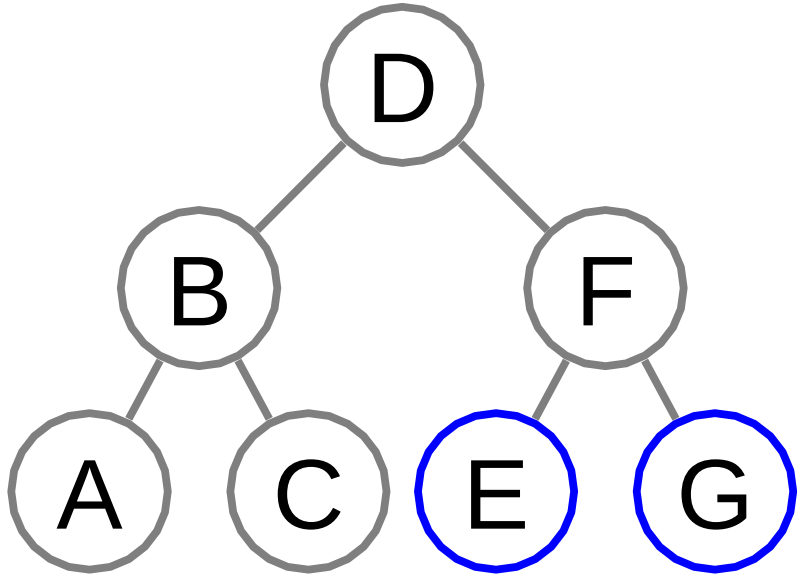
Breadth first: level traversal



Queue: A C

Output: D B F

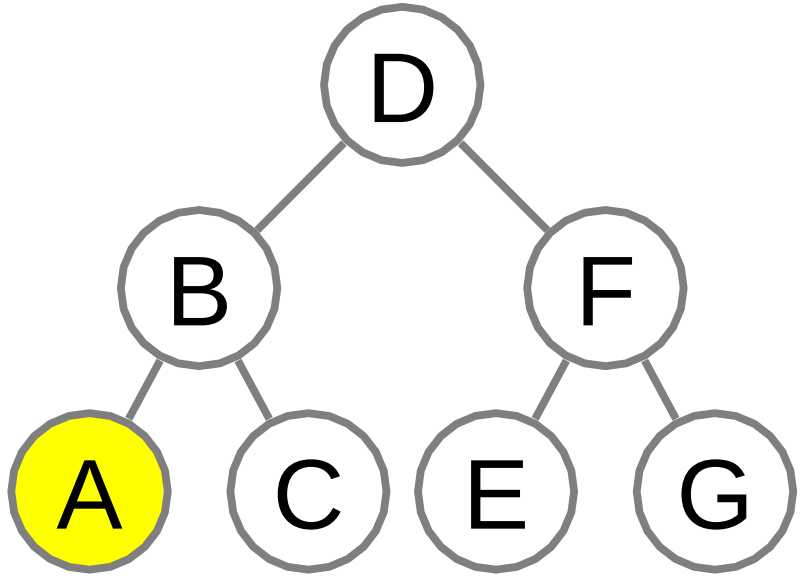
Breadth first: level traversal



Queue: A C E G

Output: D B F

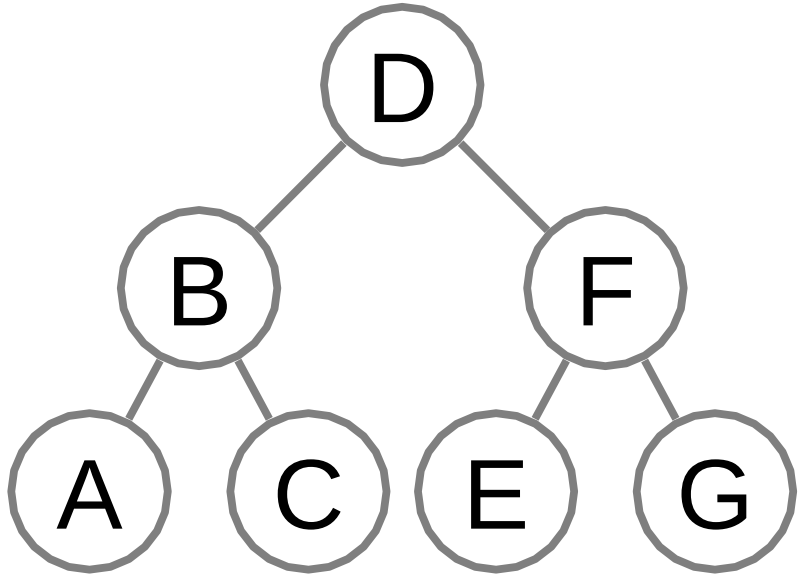
Breadth first: level traversal



Queue: A C E G

Output: D B F

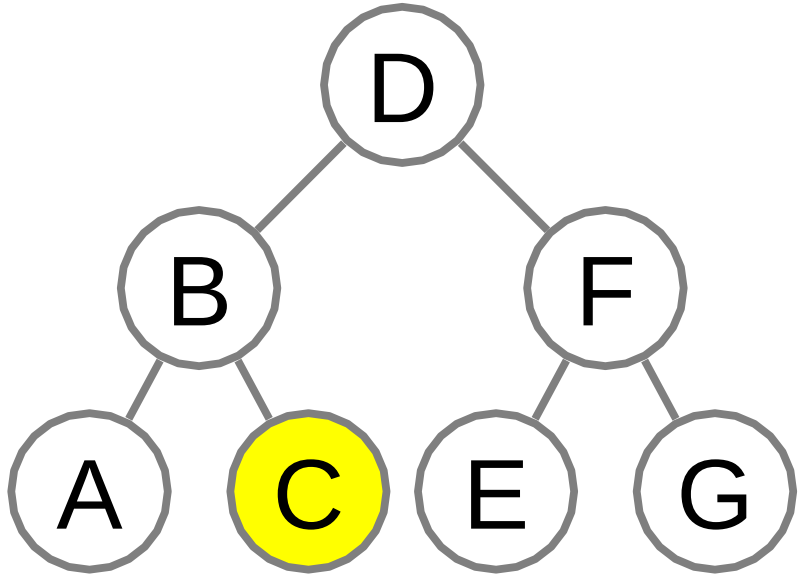
Breadth first: level traversal



Queue: C E G

Output: D B F A

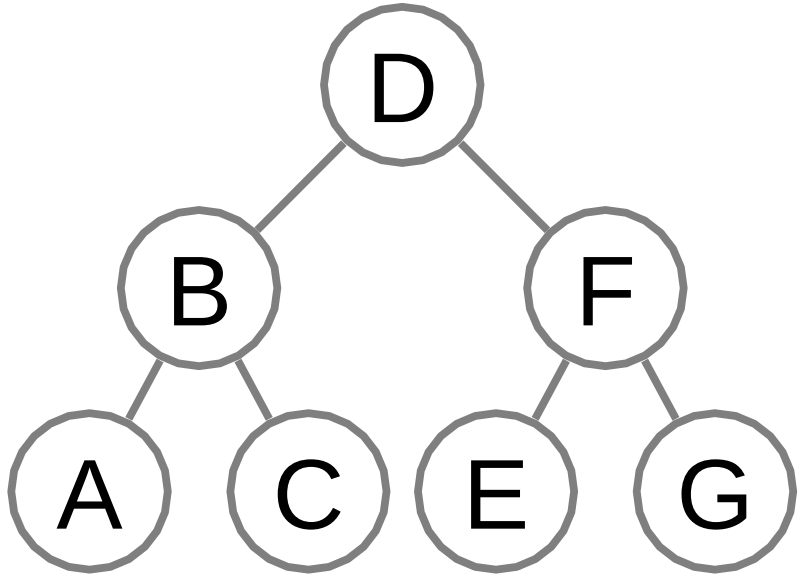
Breadth first: level traversal



Queue: C E G

Output: D B F A

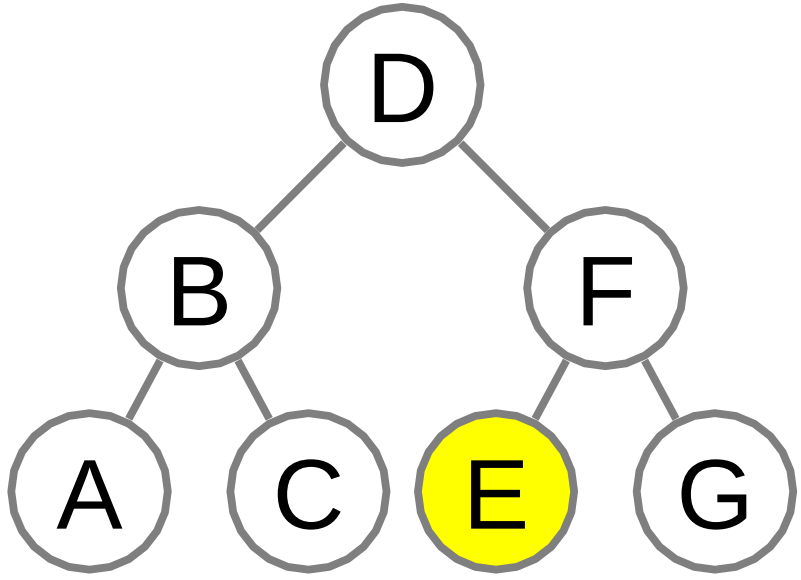
Breadth first: level traversal



Queue: E G

Output: D B F A C

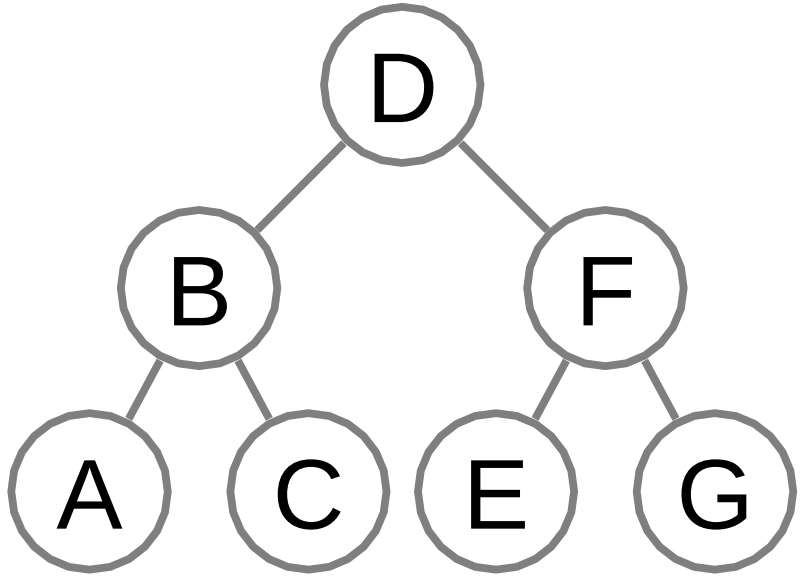
Breadth first: level traversal



Queue: E G

Output: D B F A C

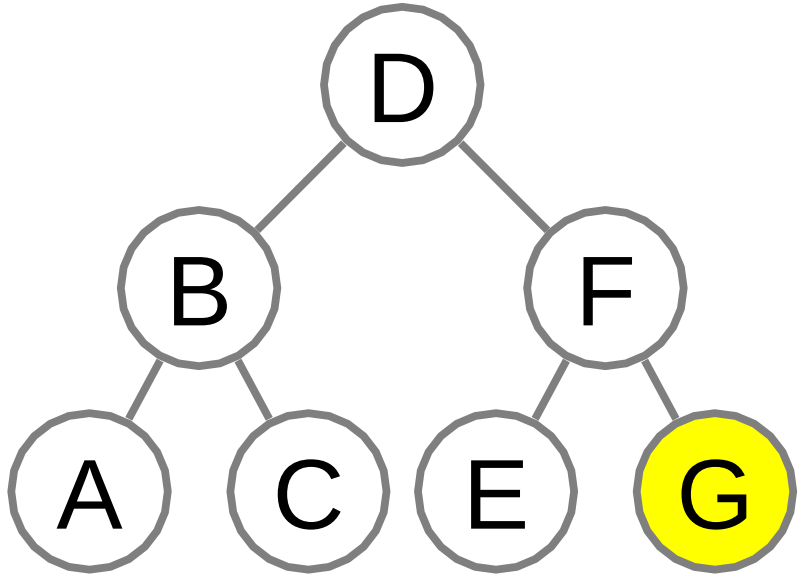
Breadth first: level traversal



Queue: G

Output: D B F A C E

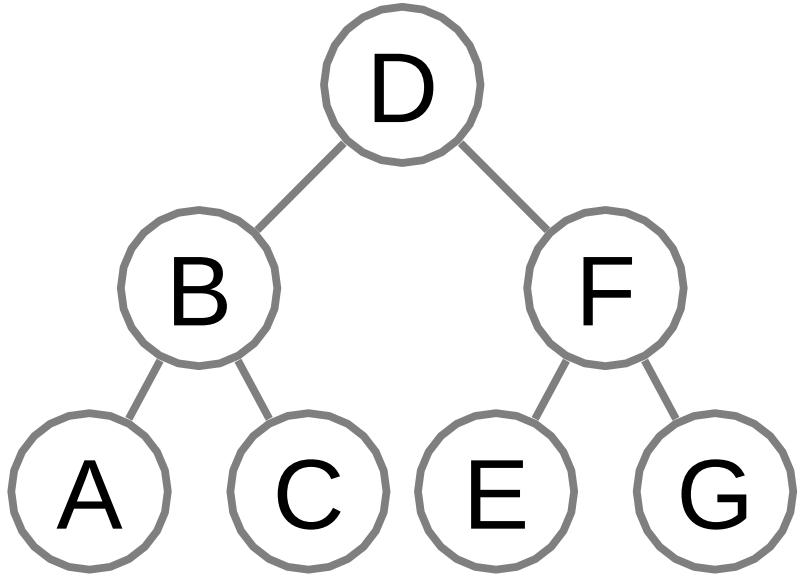
Breadth first: level traversal



Queue: G

Output: D B F A C E

Breadth first: level traversal



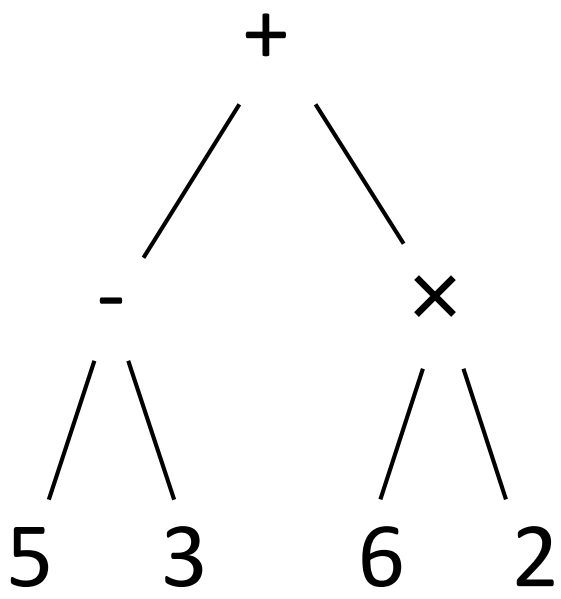
Queue: empty

Output: D B F A C E G

Summary

- Tree is defined by its root node
- Each node has (at least) a key and links to children
- Tree traversals:
  - DFS
    - pre-order
    - in-order
    - post-order
  - BFS
- When working with a tree, recursive algorithms are common
- In Computer Science, trees grow down!

Tree-traversal quiz

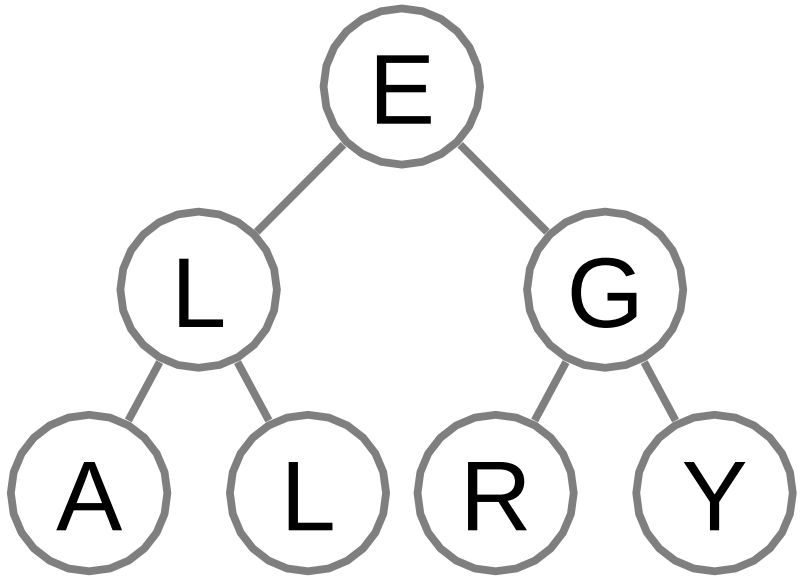


Print expression:

- ➔ Depth-first:
  - in-order
  - pre-order
  - post-order
- ➔ Breadth-first

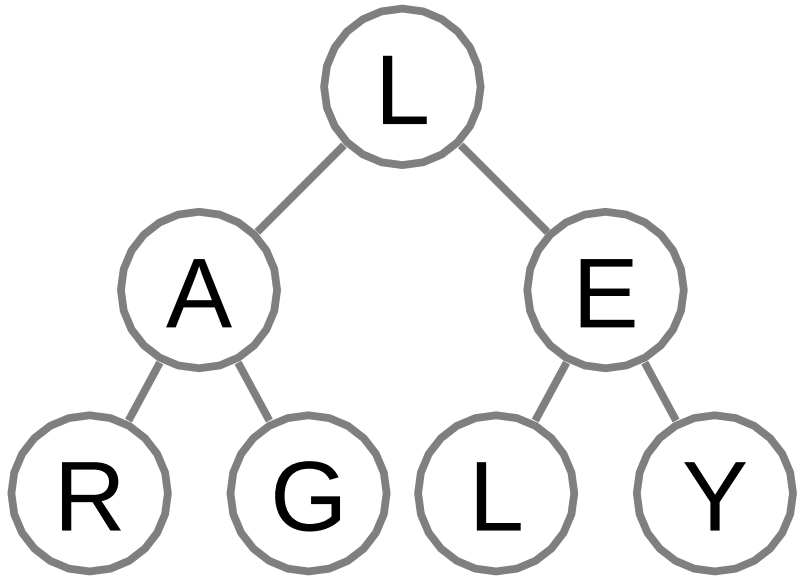
What do you think:
Which traversal is used for parsing expression trees?

Tree-traversal Puzzle 1



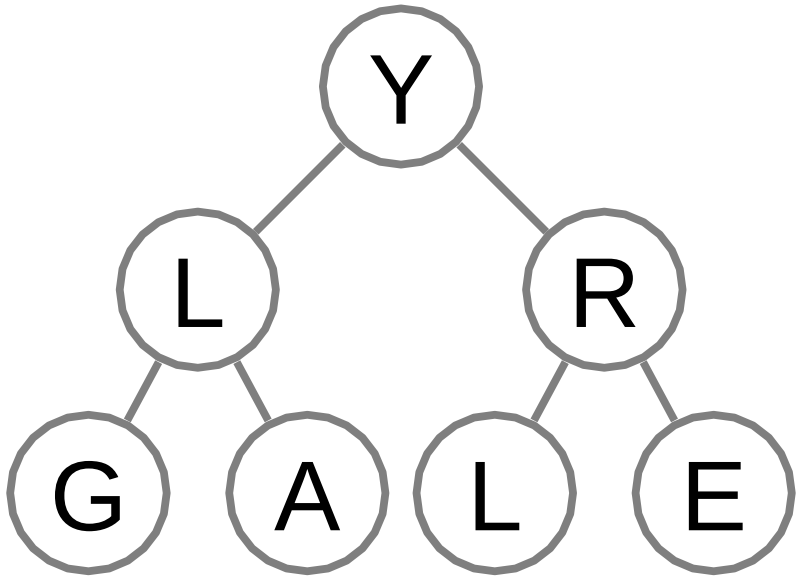
Guess the word: **in-order** traversal

Tree-traversal Puzzle 2



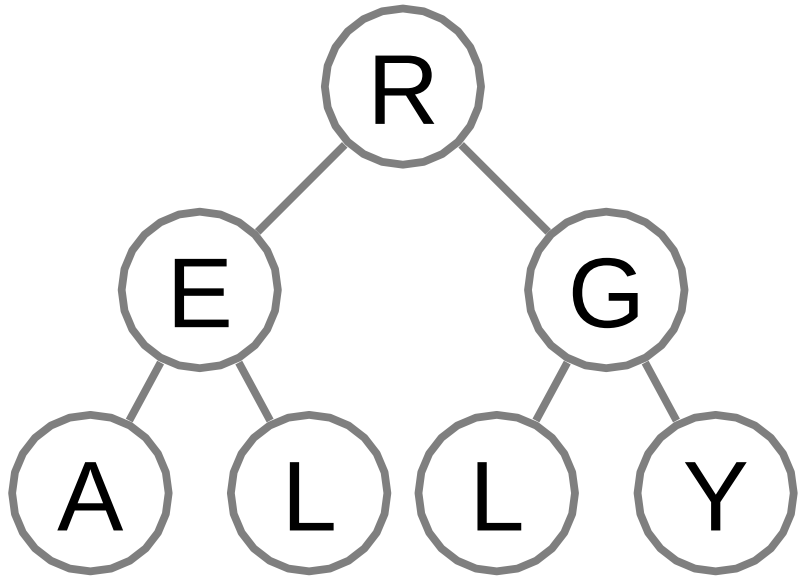
Guess the word: **pre-order** traversal

Tree-traversal Puzzle 3



Guess the word: **post-order** traversal

Tree-traversal Puzzle 4



Guess the word: **breadth-first** traversal