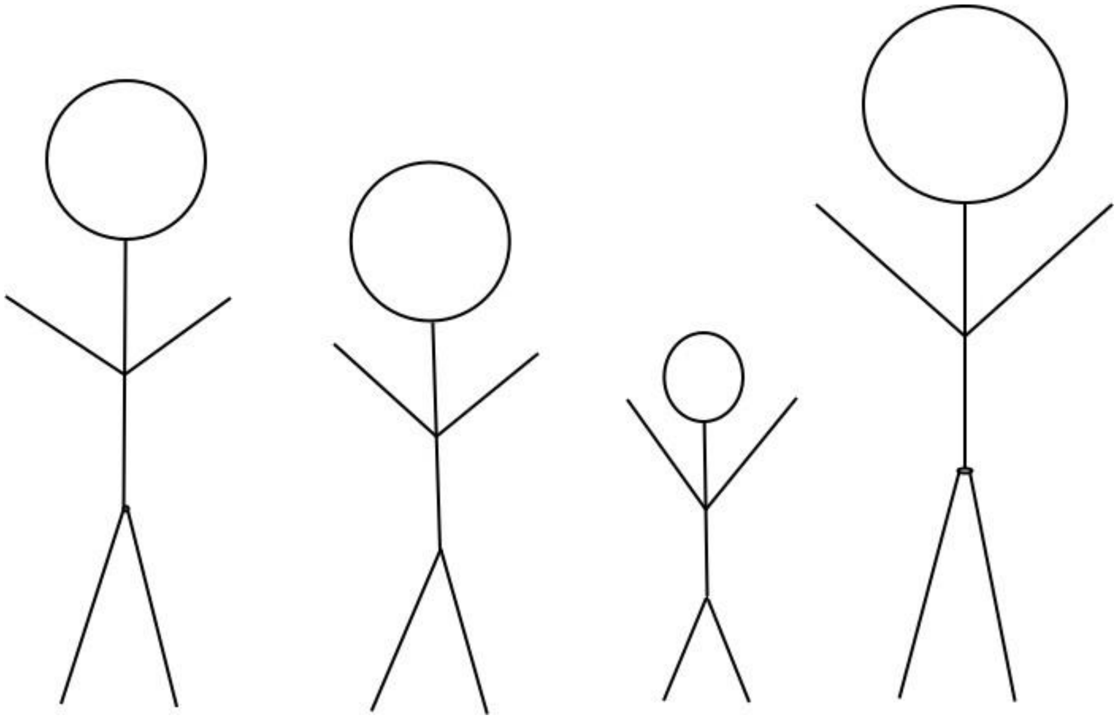
ADT and Data structures.
Local Range ADT

[Review 02.05]

by Marina Barsky

Example 1: Closest Height

Find 3 people in your class whose height is closest to yours.



Example 2: Date Ranges

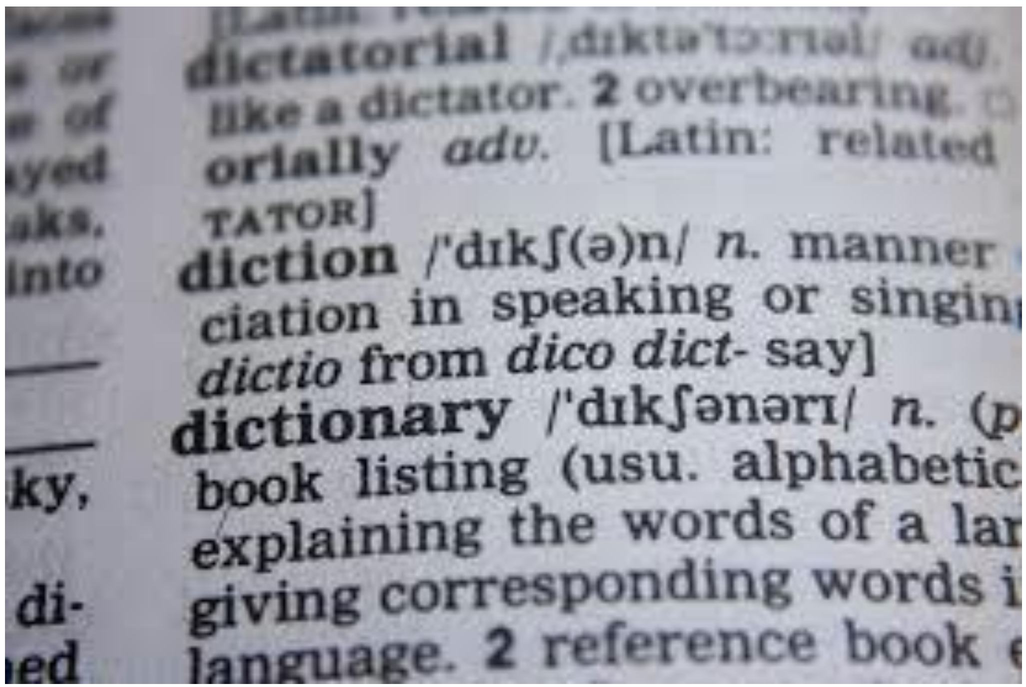
Find all emails received in a given period

Inbox

FROM	KNOW	TO	SUBJECT	SENT TIME ▾
"lawiki.i2p admin" <J5uF>		**Bote User <uhOd>**	**hi**	**Unknown**
anonymous		Bote User <uhOd>	Sanders 2016	Aug 30, 2015 3:27 PM
anonymous		Bote User <uhOd>	I2PCon 2016	Aug 30, 2015 3:25 PM
Anon Developer <gvbM>		**Bote User <uhOd>**	**Re: Bote changess**	**Aug 30, 2015 2:54 PM**
I2P User <uUUx>		**Bote User <uhOd>**	**Hello World!**	**Aug 30, 2015 2:51 PM**

Example 3: Partial Matching

Find all words that **start with** some given *prefix*



Abstract Data Type: Local Range

Specification

A ***Local Range*** **ADT** stores a number of elements each with a *key* and supports the following operations:

- ➔ ***RangeSearch****(lo, hi)*: returns all elements with keys between *lo* and *hi*

  Reduces to *find(x)* if *x=lo=hi*

- ➔ ***NearestNeighbors****(x, k)*: returns *k* elements with keys closest to *x*

  when k = 1:

  you want successor(x)

  or you want predecessor(x)

Sorted Keys

1	4	6	7	10	13	15

The best idea for these queries is to store the keys **in a sorted order**

Dynamic Data Structure

- Store keys in **sorted order**
- Also want to be able to add/remove keys efficiently:

Insert(x): Adds an element with key *x*

Delete(x): Removes the element with key *x*

Data Structures for Range ADT

1	4	6	7	10	13	15

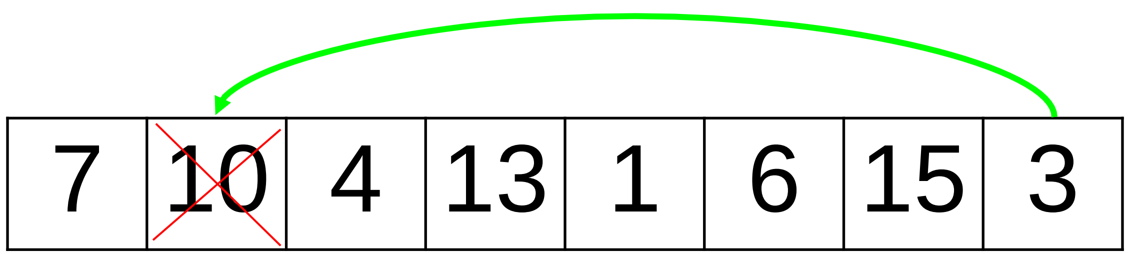
Let’s try known data structures:

- ➢ Unsorted Array
- ➢ Sorted Array
- ➢ Sorted Linked List
- ➢ Hash table

Unsorted Array

- ➔ Range Search: $O(n)$ ×
- ➔ Nearest Neighbors: $O(n)$ ×
- ➔ Insert: $O(1)$ V
- ➔ Delete: $O(1)$ V

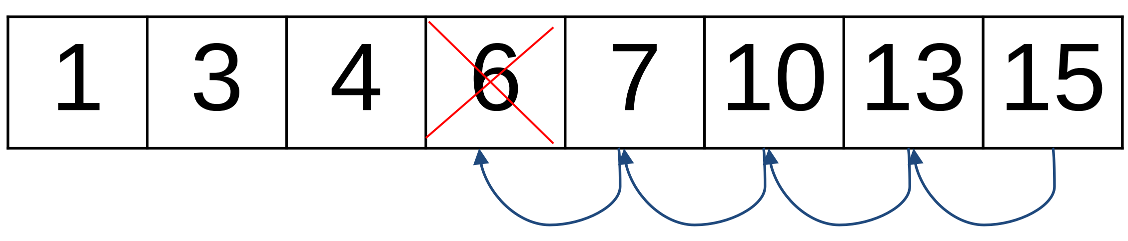
delete (10)



Sorted Array

- ➔ Range Search: $O(\log(n))$ V
- ➔ Nearest Neighbors: $O(\log(n))$ V
- ➔ Insert: $O(n)$ ×
- ➔ Delete: $O(n)$ ×

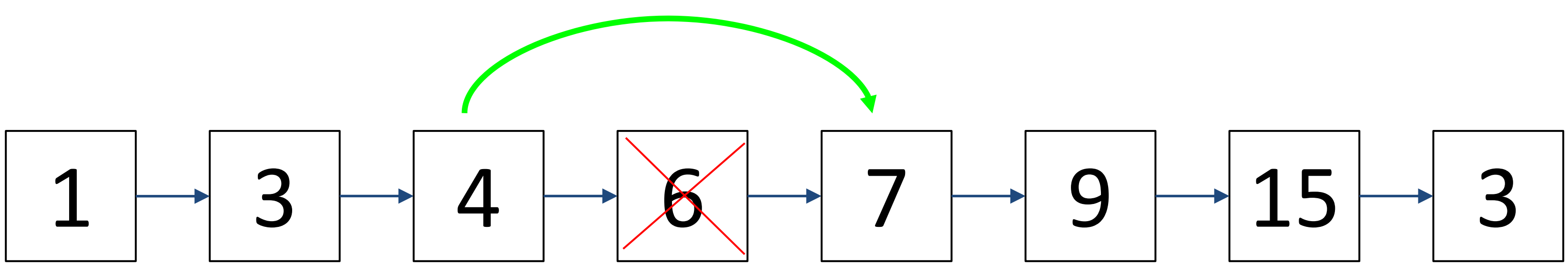
delete (6)



Sorted Linked List

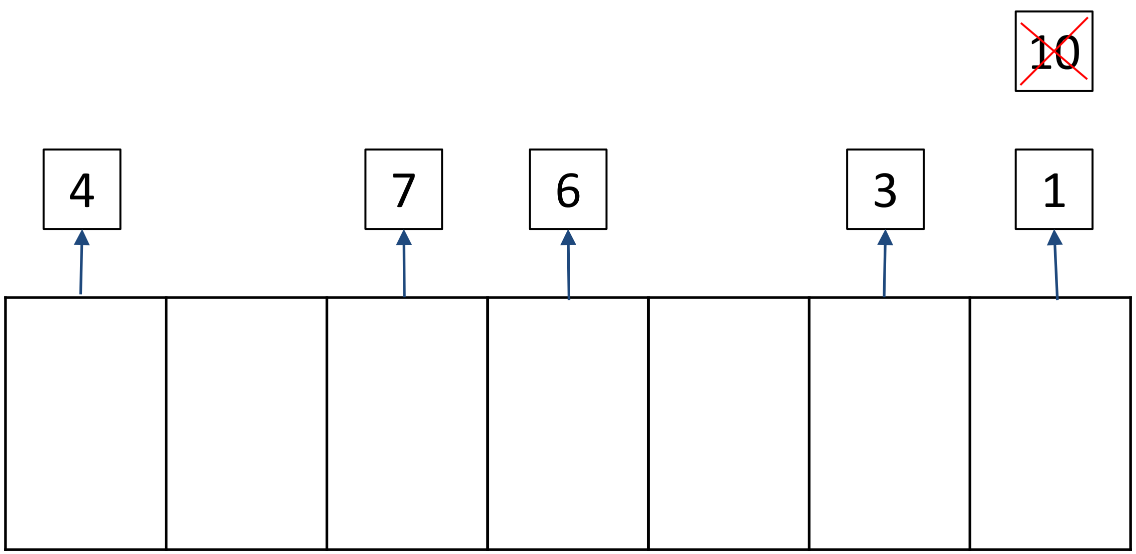
- ➔ Range Search: $O(n)$ ×
- ➔ Nearest Neighbors: $O(n)$ ×
- ➔ Insert: $O(n)$ ×
- ➔ Delete: $O(n)$ ×

delete (6)



Hash Table

- ➔ Range Search: Impossible ×
- ➔ Nearest Neighbors: Impossible ×
- ➔ Insert: $O(1)$ V
- ➔ Delete: $O(1)$ V

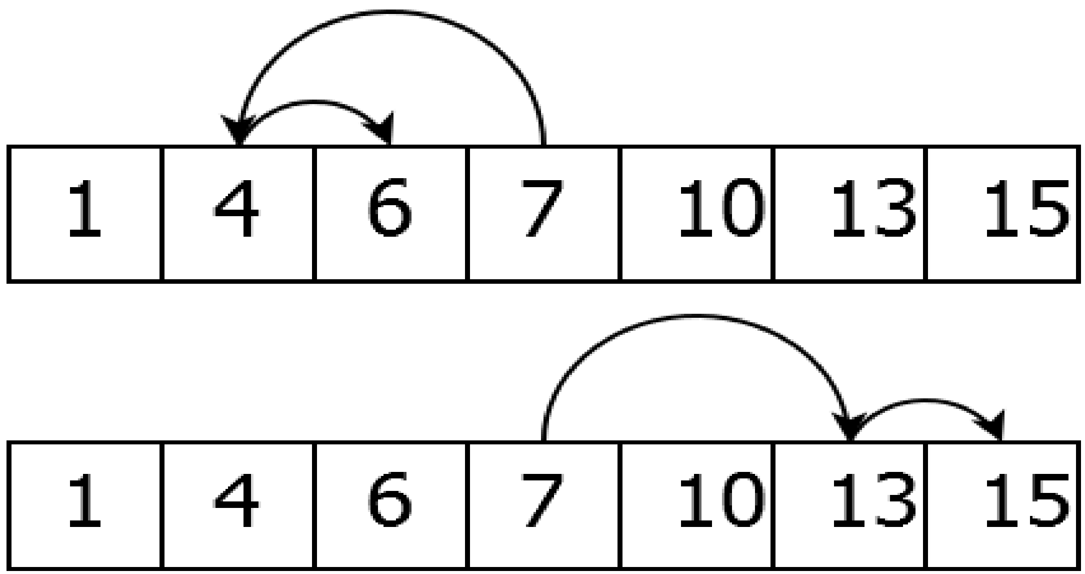


Nothing works

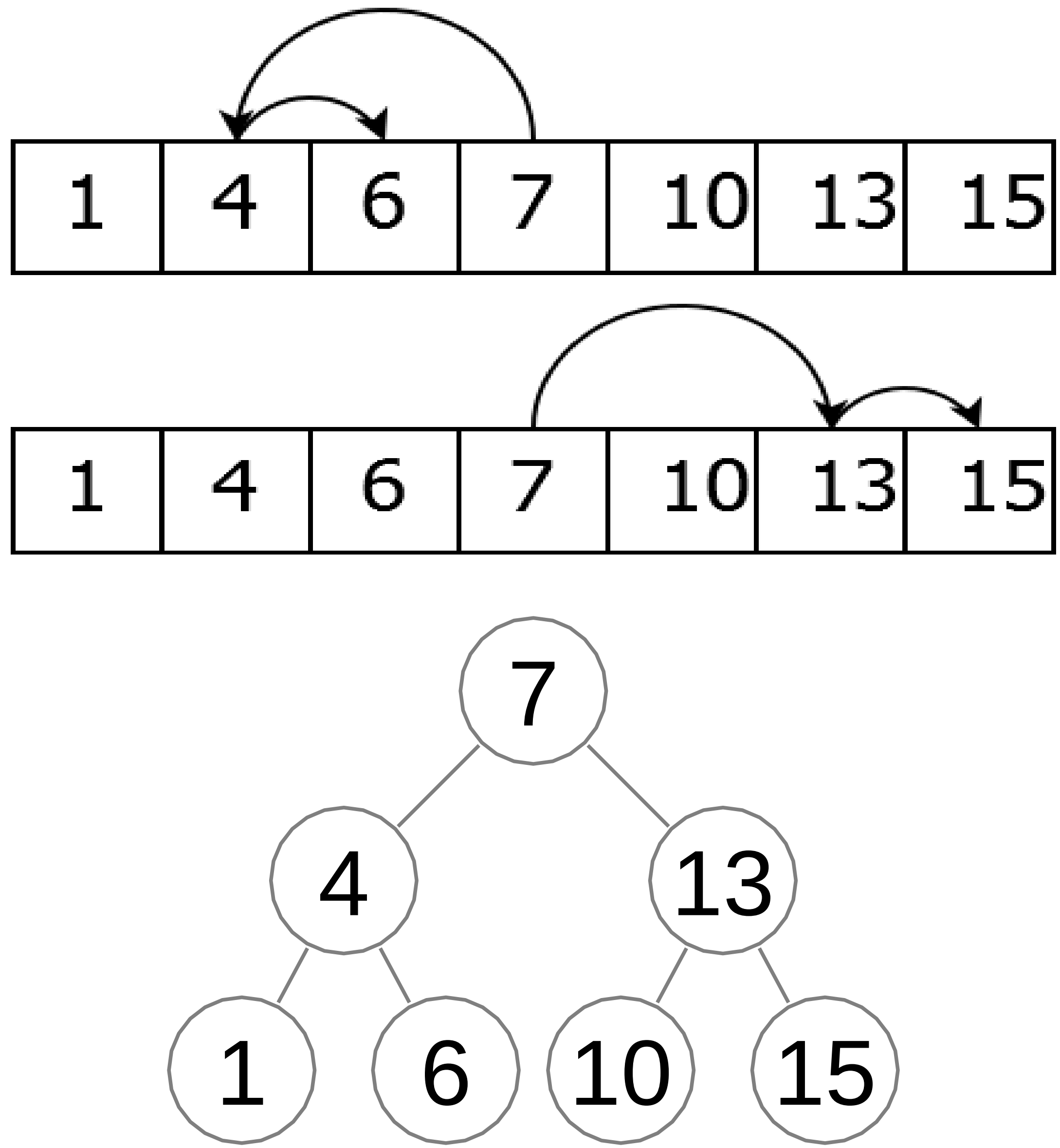
- We want efficient data structure for Local Range ADT
- None of the existing data structures work
- Sorted arrays are good for search but not for update

We need something new

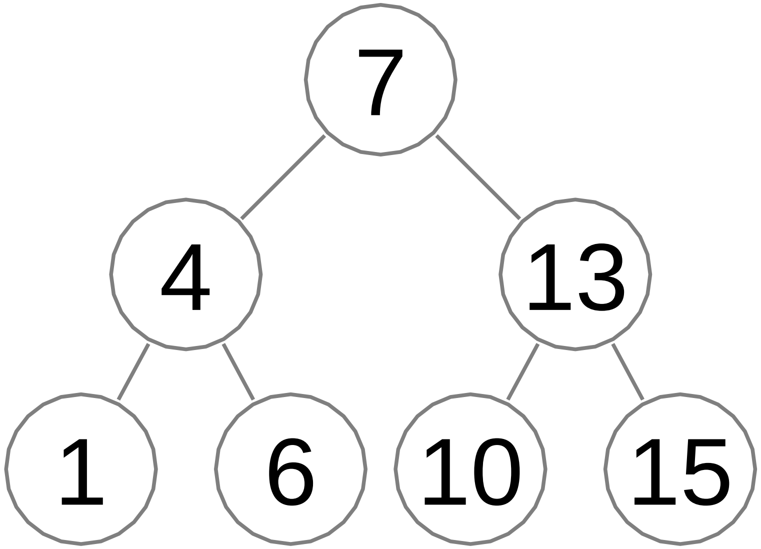
Binary Search



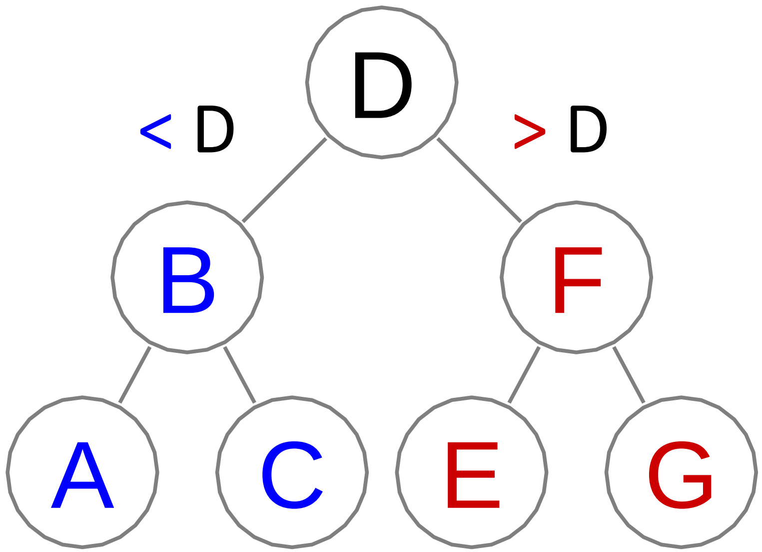
Record search questions



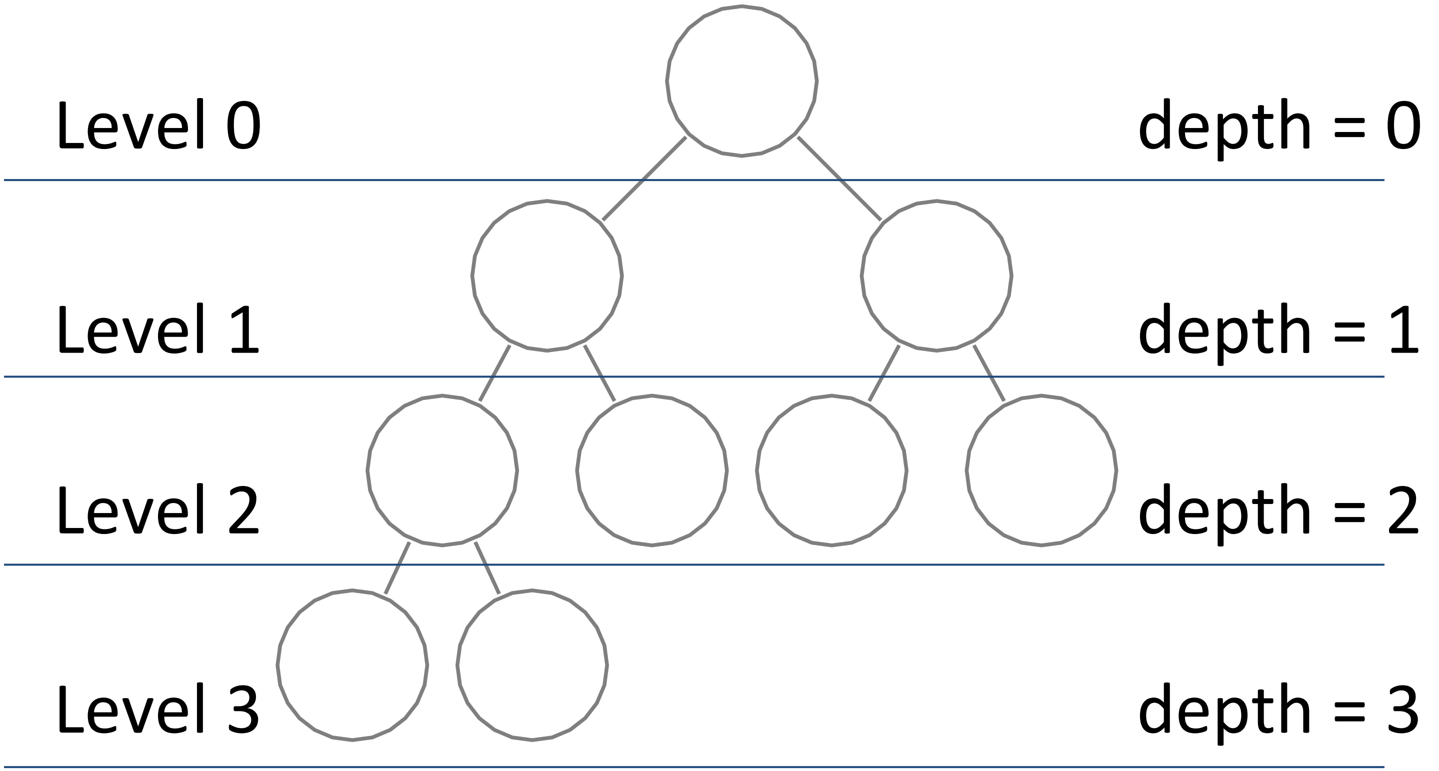
We get a tree



Binary Search Tree

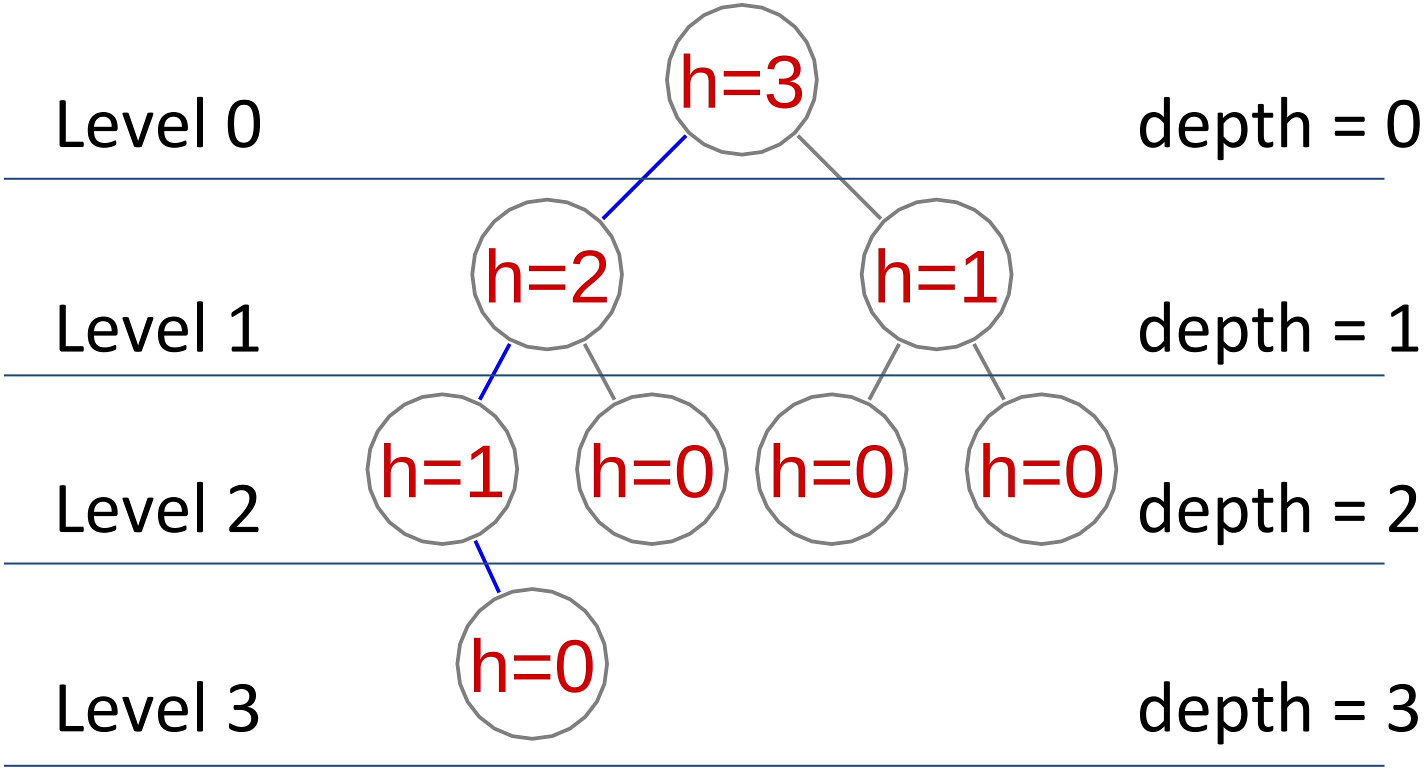


Tree levels and node depth



***Depth*: distance from the root -**
how many edges to go from the root to a

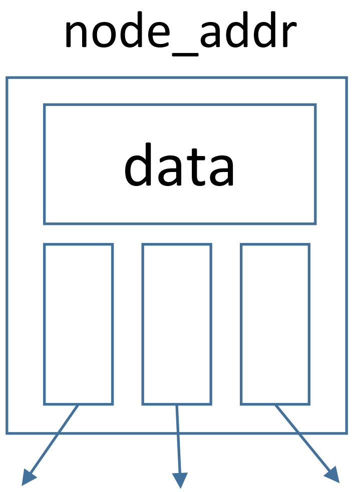
Node height



***Height*: distance from the node to the bottom: how many edges to go to the furthest leaf**

Tree – recursive data structure

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



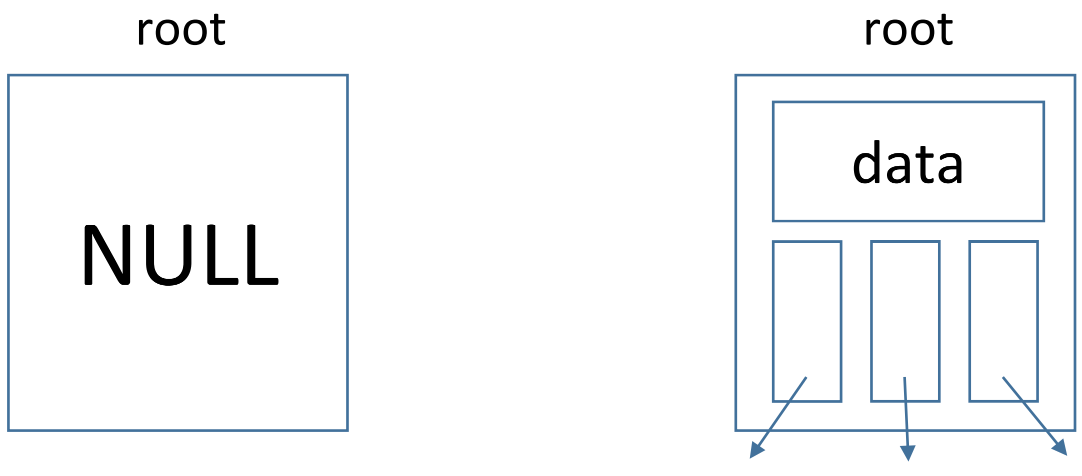

```c
typedef struct node {
    int data;
    struct node ** children;
    [struct node * parent;]

} TreeNode;
```

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

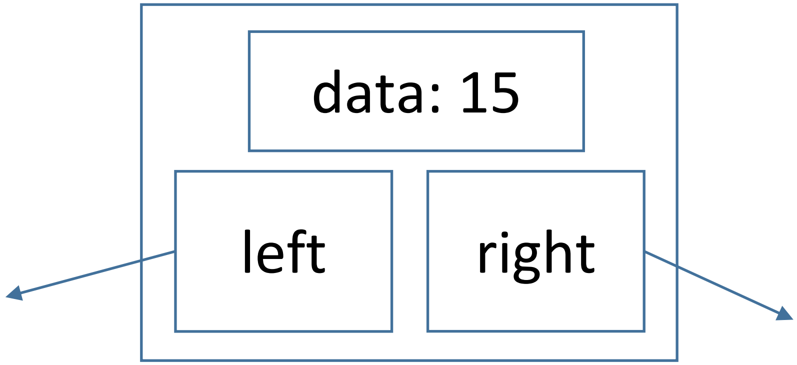
Tree is defined by a single node

Tree is either

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



Binary tree: at most 2 children




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

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

Recursive algorithms are common

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))
```

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

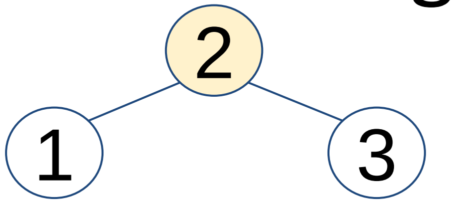
- In-order
- Pre-order
- Post-order

Depth-first: in-order

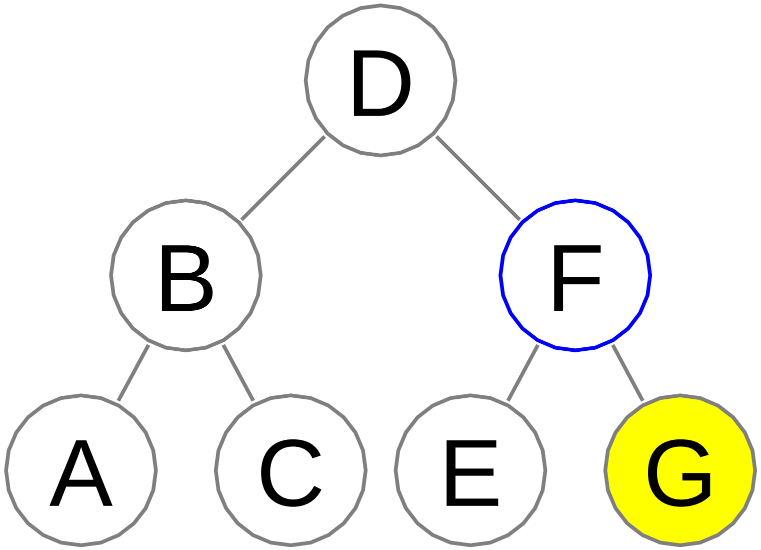
Algorithm *InOrderTraversal*(*tree*)

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

left - me - right

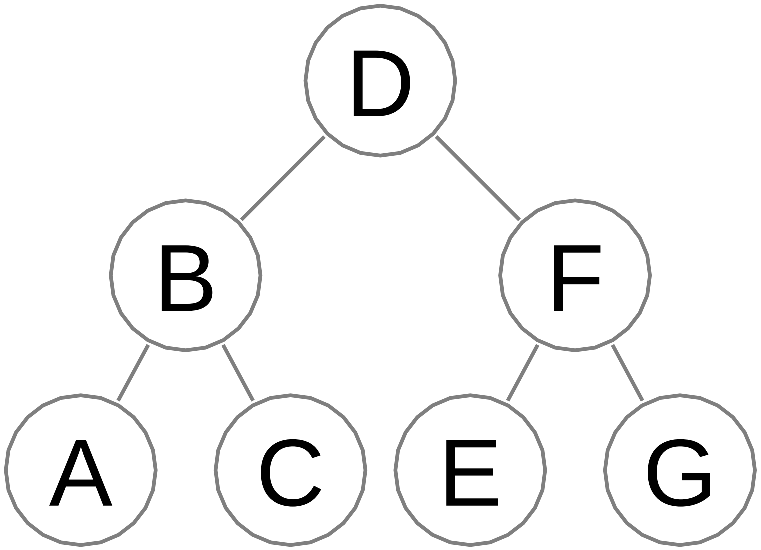


In-order



A B C D E F G

In-order



me, node D

A B C D E F G

left subtree of D

right subtree of D

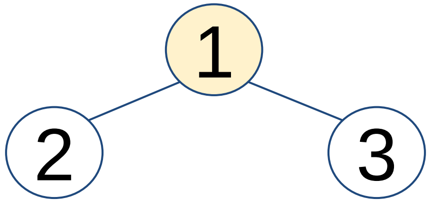
Depth-first: pre-order

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

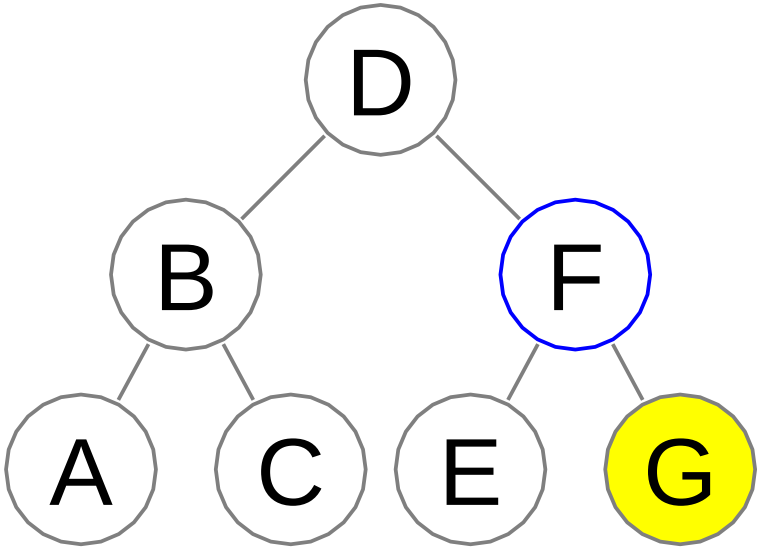
```
if tree is null:
      return
print (tree.key)
PreOrderTraversal(tree.left)
PreOrderTraversal(tree.right)
```

me first

me - left -right

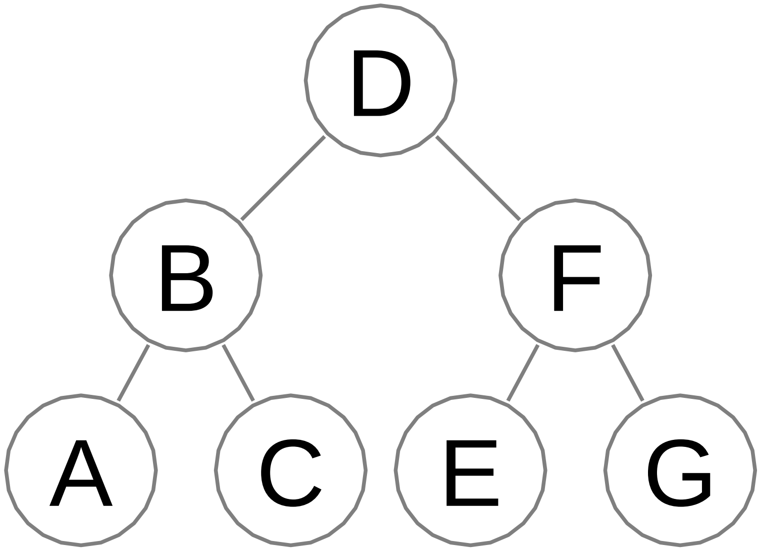


Pre-order



D 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

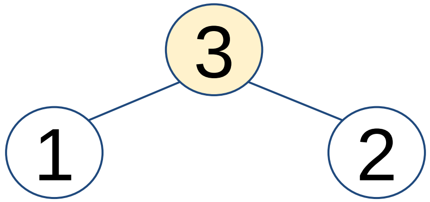
Depth-first: postorder

Algorithm *PostOrderTraversal*(*tree*)

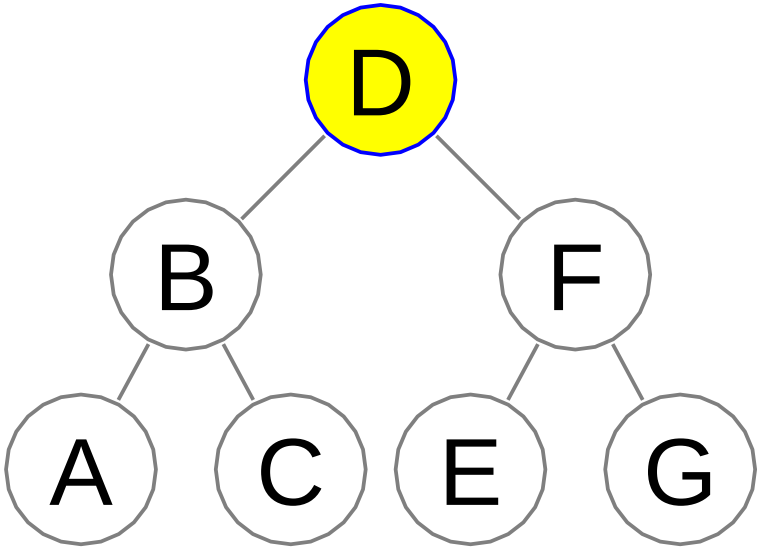
```
if tree is null:
  return
PostOrderTraversal(tree.left)
PostOrderTraversal(tree.right)
print(tree.key )
```

children first

left-right-me

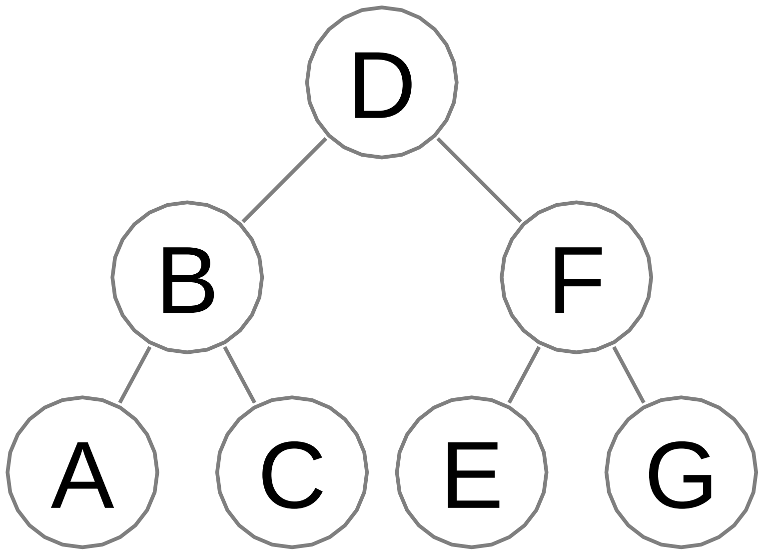


Post-order



A C B E G F D

Post-order



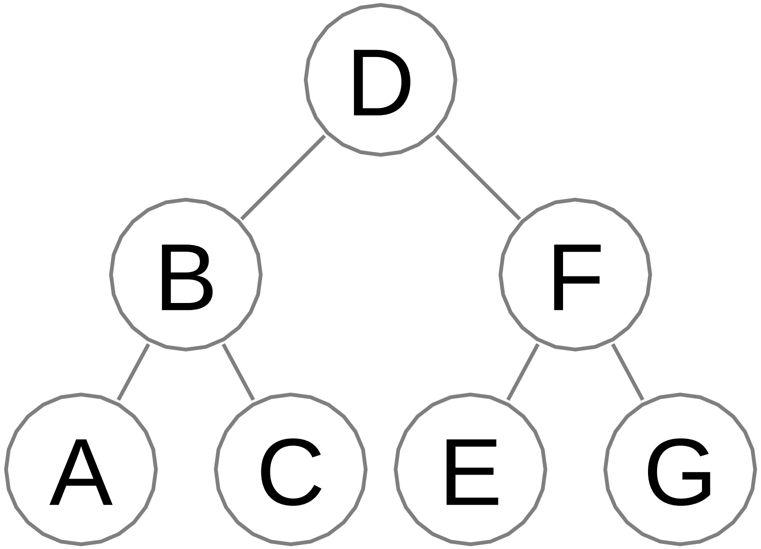
me, node D

A C B E G F D

left subtree of D

right subtree of D

Breadth first



Level traversal:

D

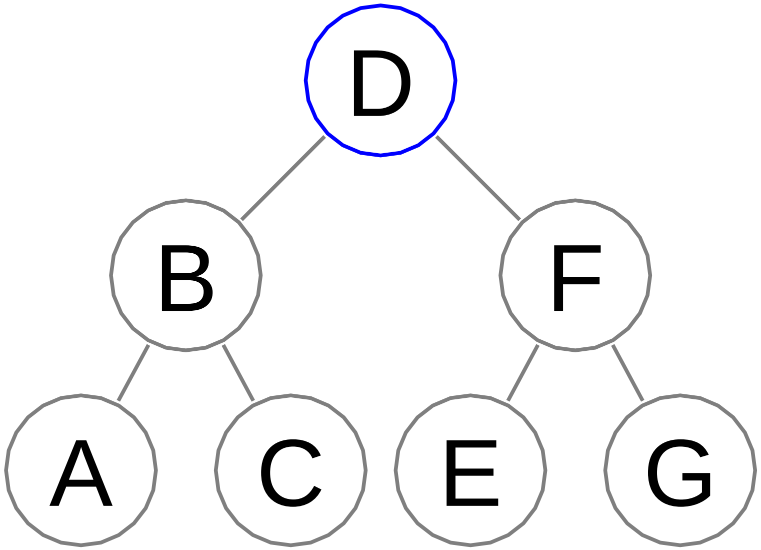
B F

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)
```

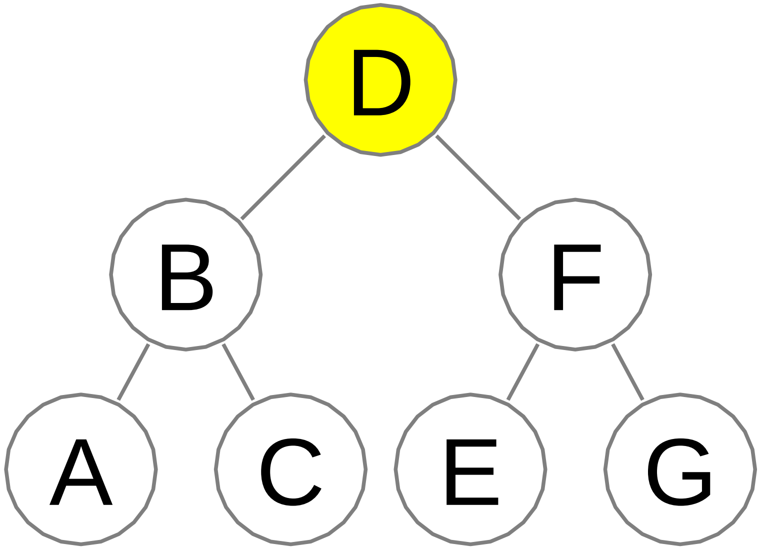
Breadth first: level traversal



Queue: D

Output:

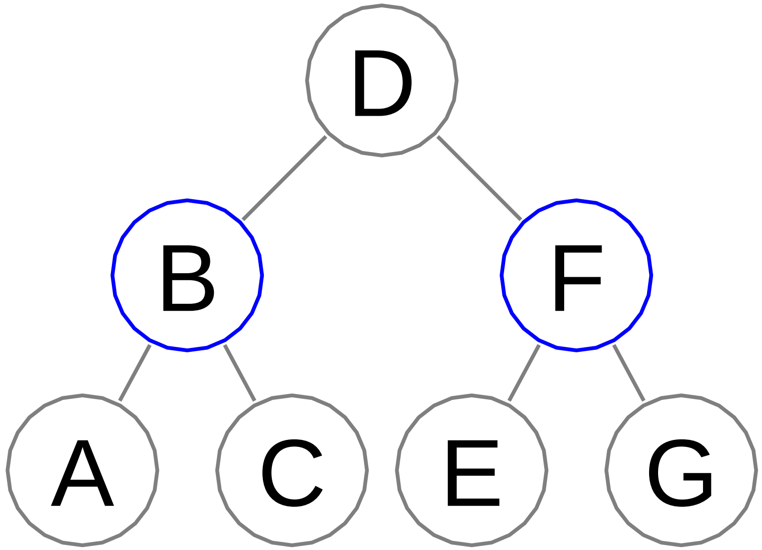
Breadth first: level traversal



Queue:

Output: D

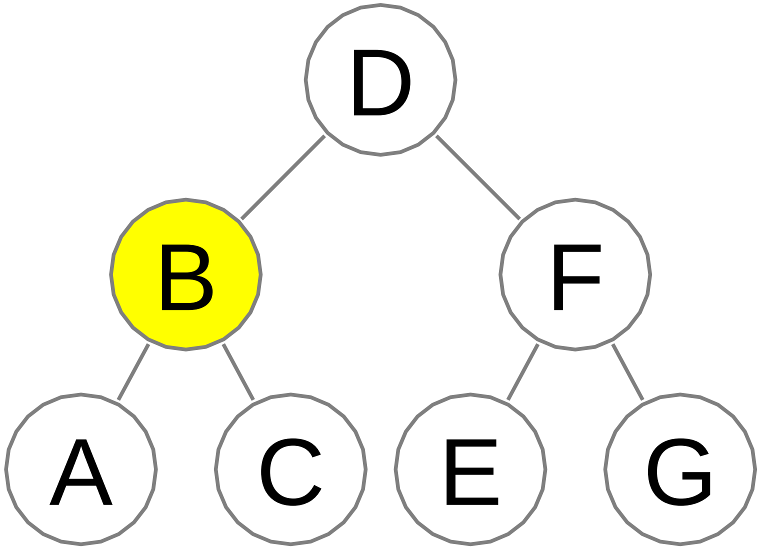
Breadth first: level traversal



Queue: B F

Output: D

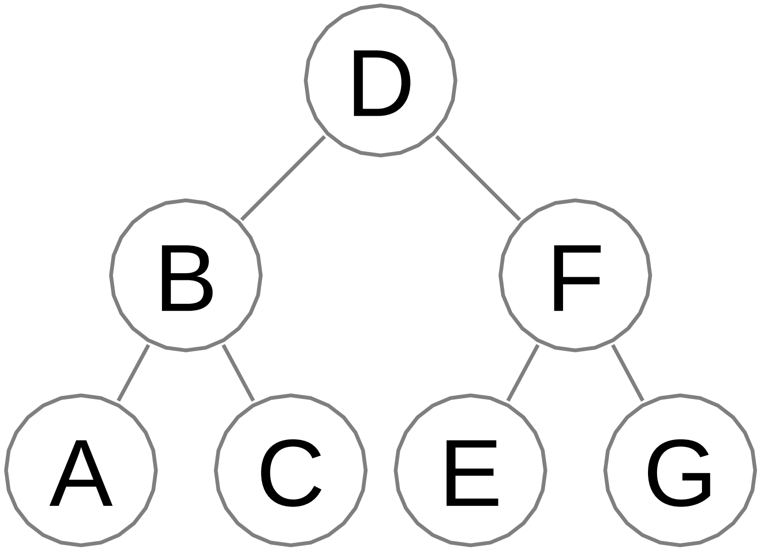
Breadth first: level traversal



Queue: B F

Output: D

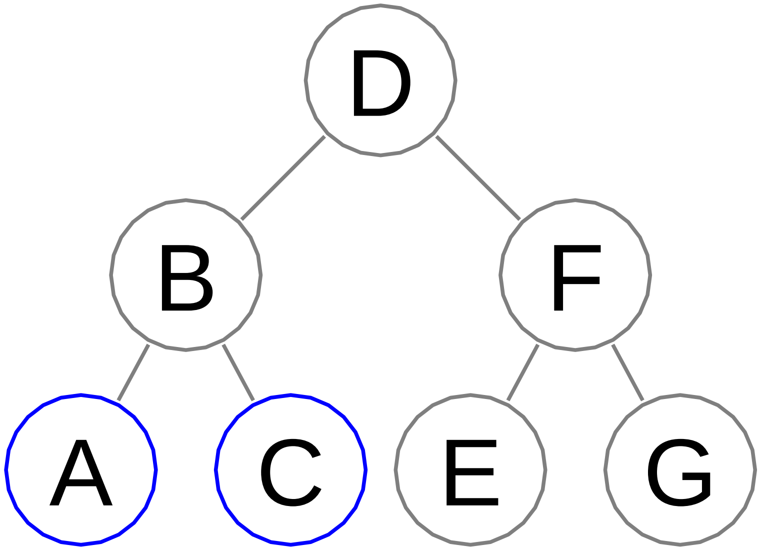
Breadth first: level traversal



Queue: F

Output: D <u>B</u>

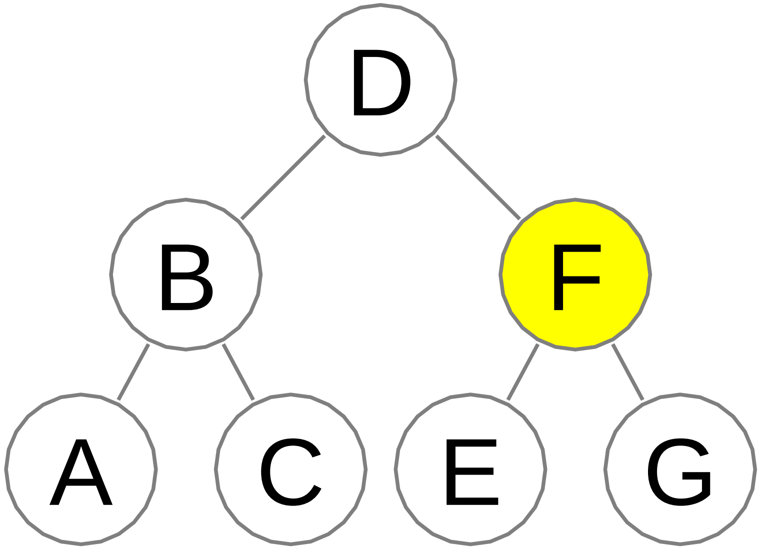
Breadth first: level traversal



Queue: F A C

Output: D B

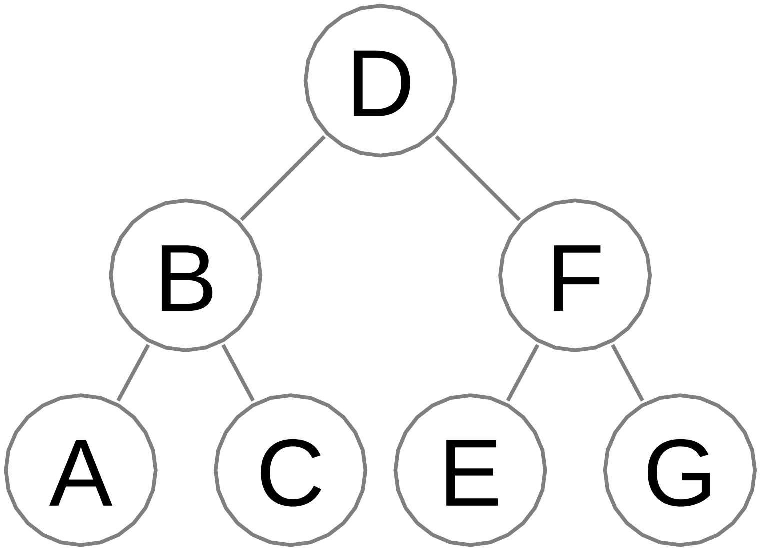
Breadth first: level traversal



Queue: F A C

Output: D B

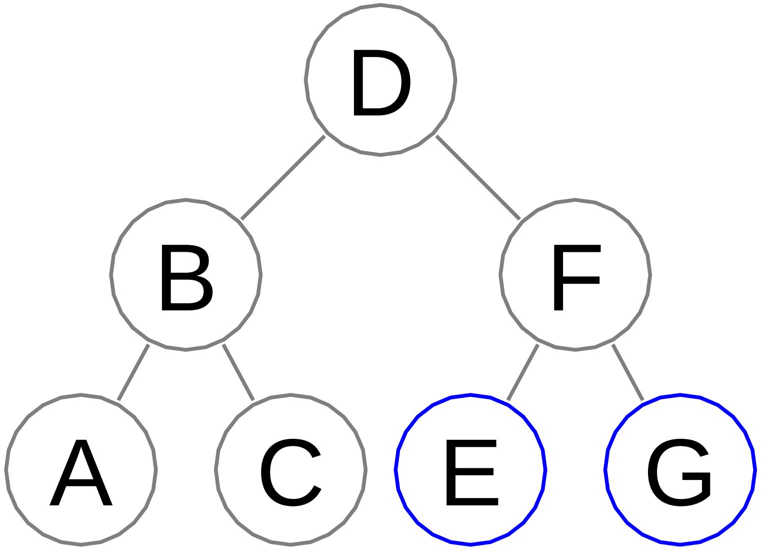
Breadth first: level traversal



Queue: A C

Output: D B F

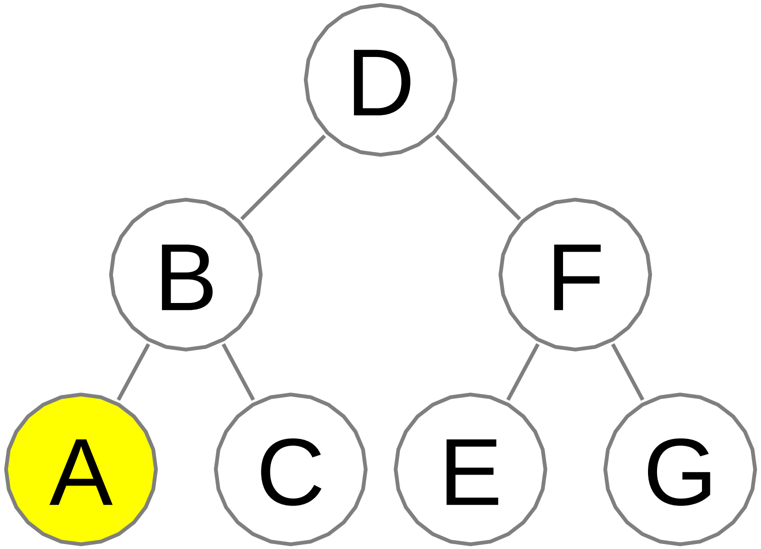
Breadth first: level traversal



Queue: A C E G

Output: D B F

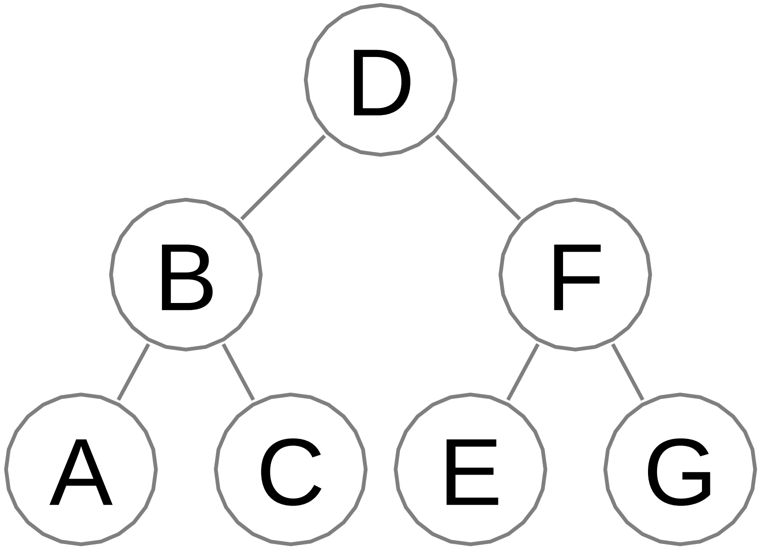
Breadth first: level traversal



Queue: <u>A</u> C E G

Output: D B F

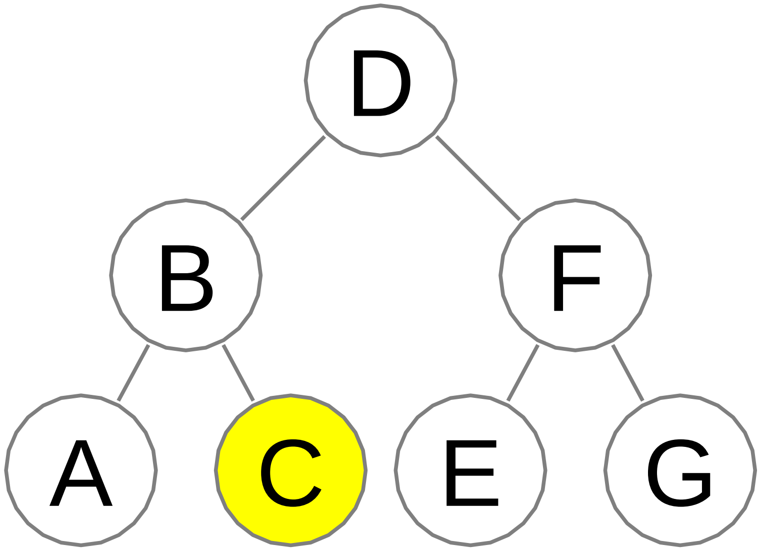
Breadth first: level traversal



Queue: C E G

Output: D B F <u>A</u>

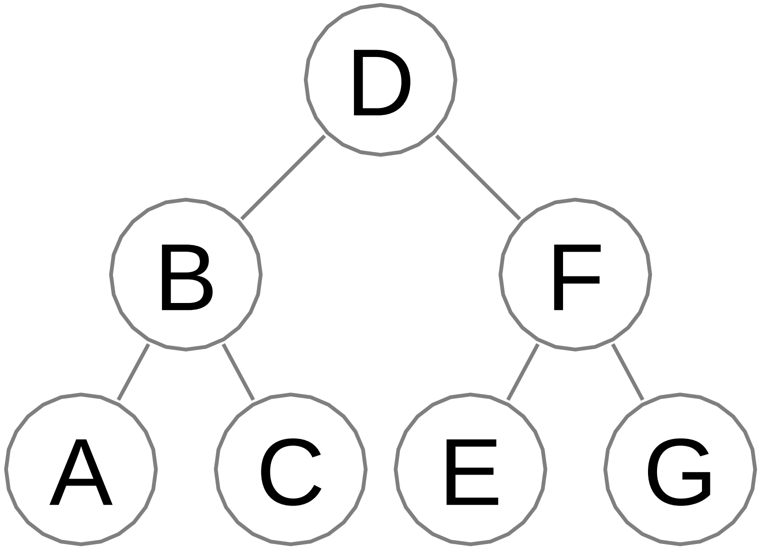
Breadth first: level traversal



Queue: <u>C</u> E G

Output: D B F A

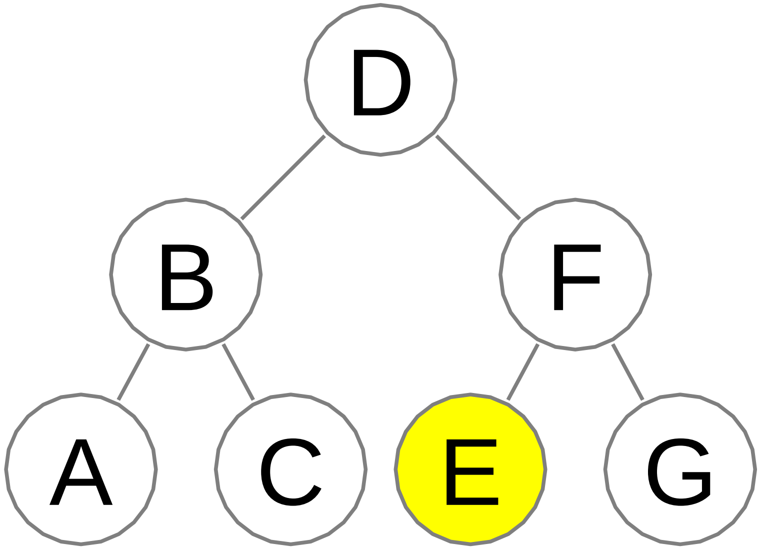
Breadth first: level traversal



Queue: E G

Output: D B F A <u>C</u>

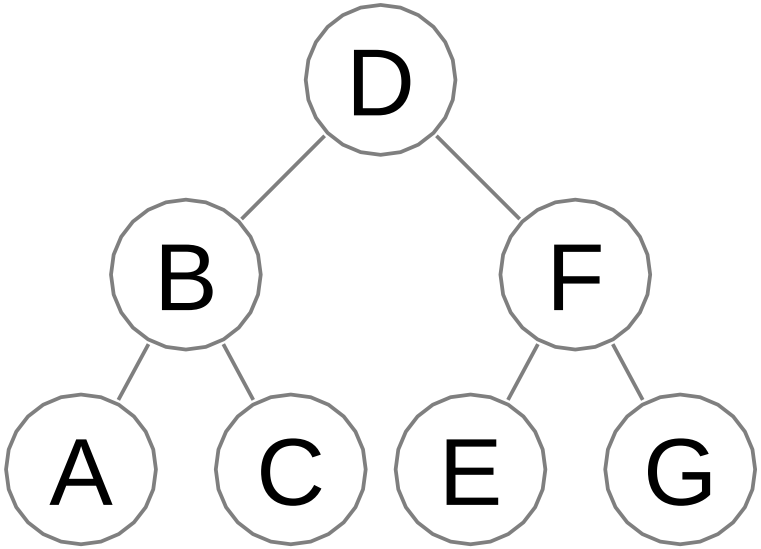
Breadth first: level traversal



Queue: E G

Output: D B F A C

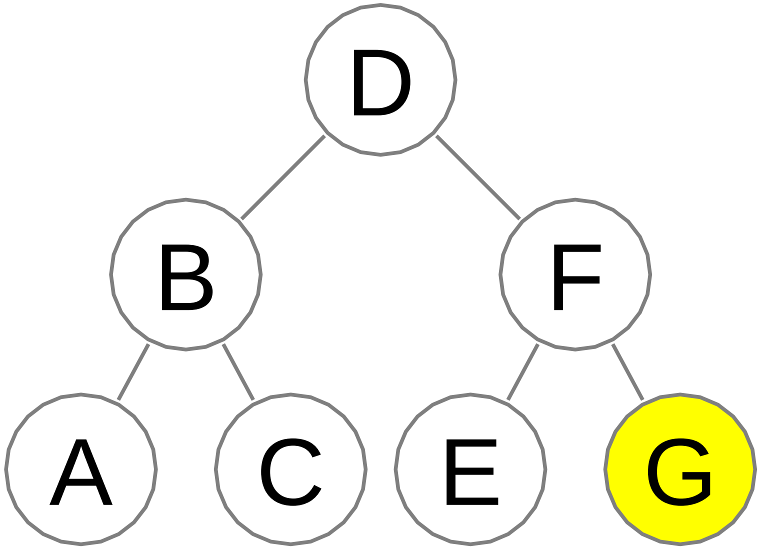
Breadth first: level traversal



Queue: G

Output: D B F A C <u>E</u>

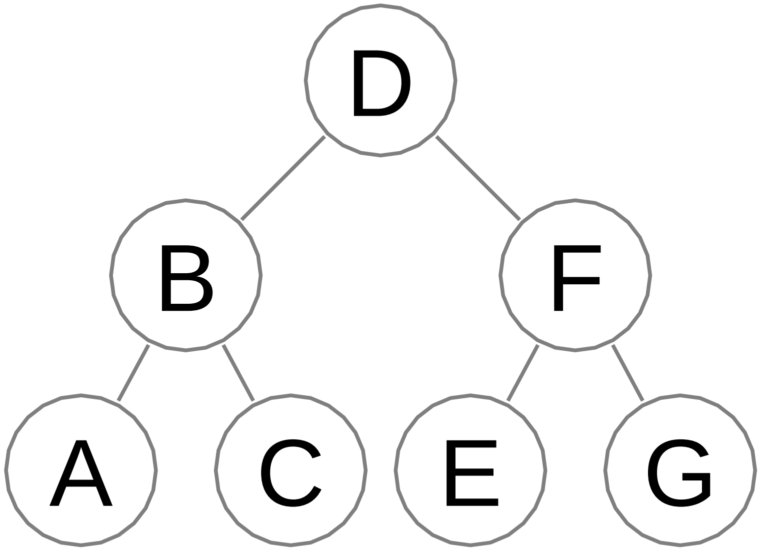
Breadth first: level traversal



Queue: G

Output: D B F A C E

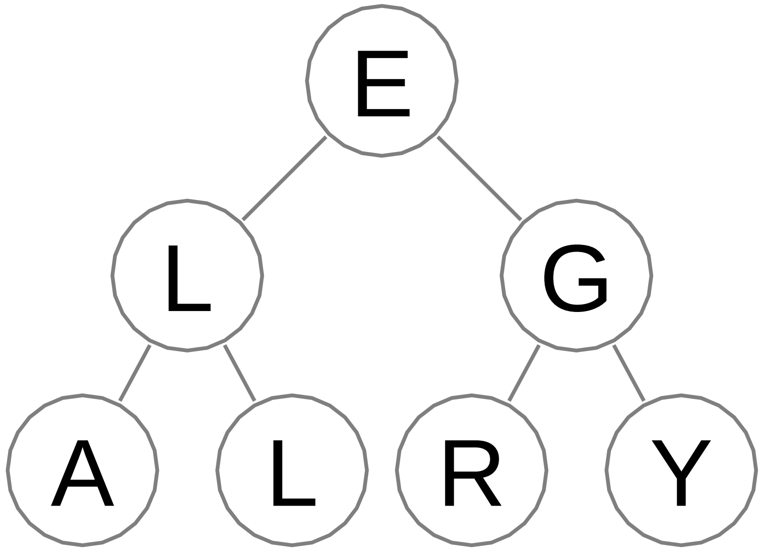
Breadth first: level traversal



Queue: empty

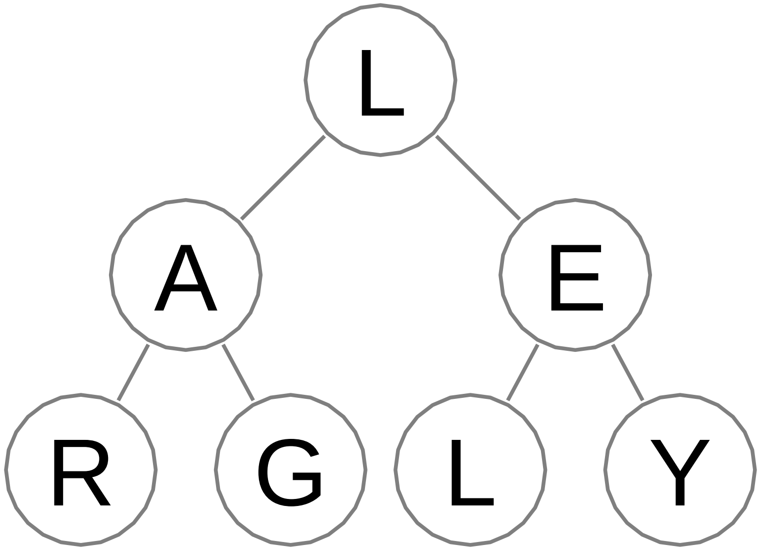
Output: D B F A C E <u>G</u>

Tree-traversal Puzzle 1



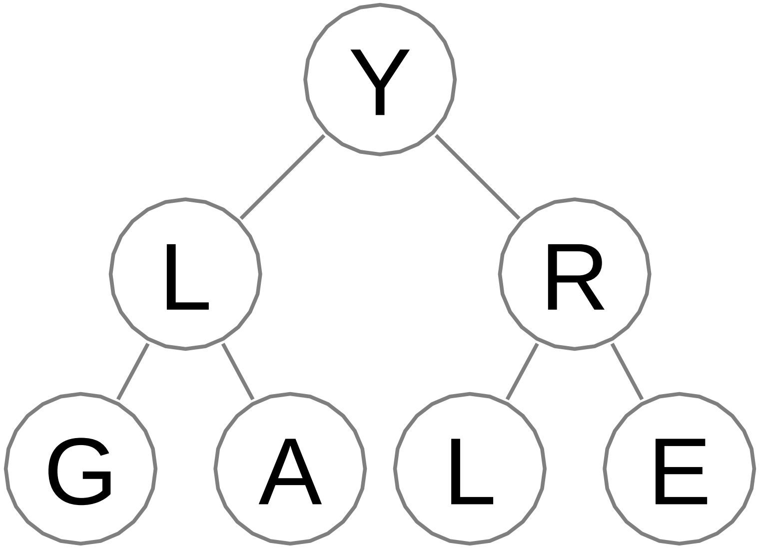
Guess the word: **in-order** traversal

Tree-traversal Puzzle 2



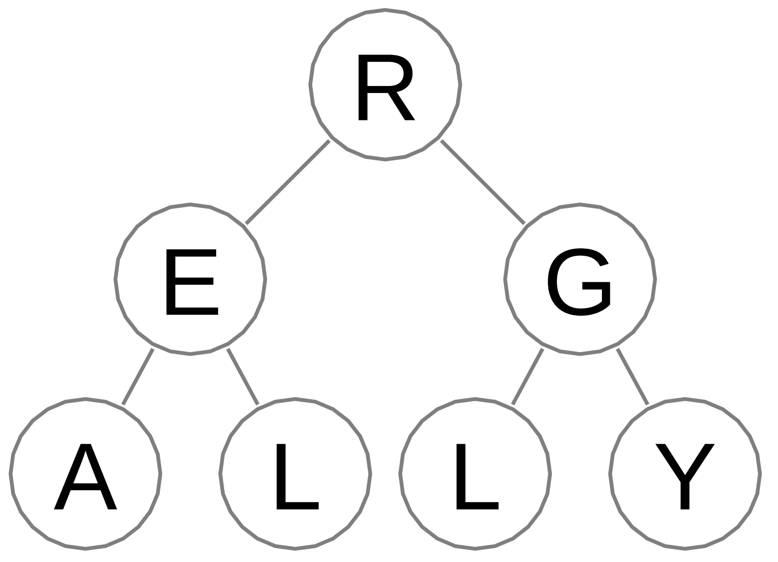
Guess the word: **pre-order** traversal

Tree-traversal Puzzle 3



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

Tree-traversal Puzzle 4



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