Trees & Tree Traversal

Lecture 5 - EECS 214

Introduction: Hi, I’m Ethan

- I’m your graduate TA
- I’m Ian’s PhD student
- I make video games
- I also research video games
  - My focus is on experimental gameplay and procedural content generation
- I play a lot of video games
  - I like Factorio, Dark Souls, & Persona 5
- I have a cat named Nina Jones ➜

Nina Jones (not me)

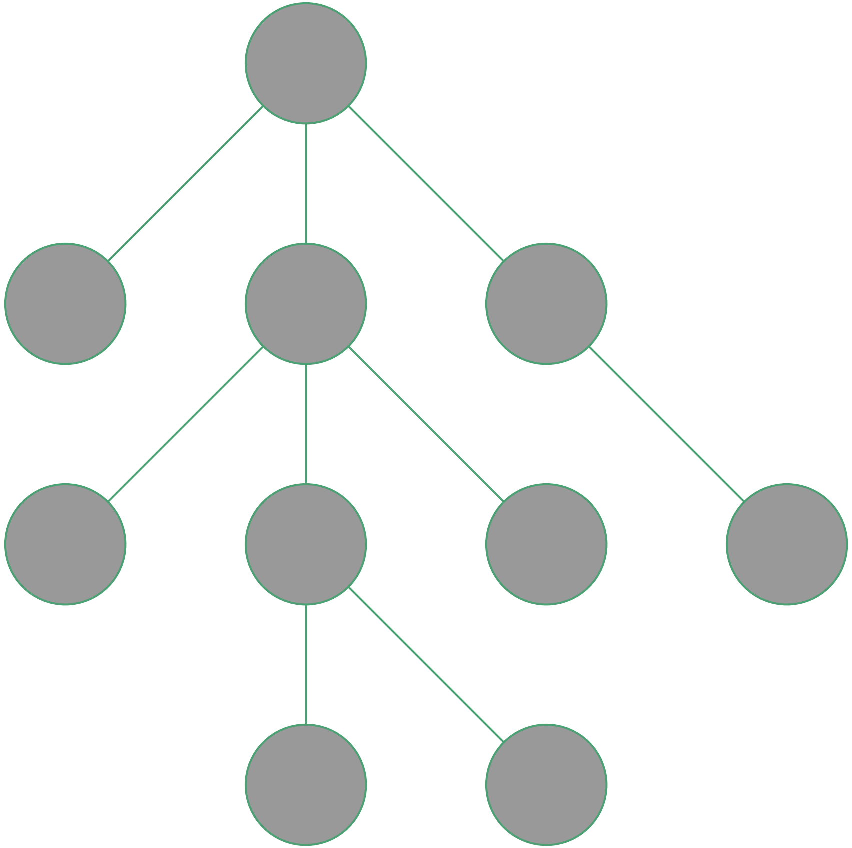
Trees

What are they?

- *Extremely common* data structure in representing information
- Most tree **algorithms** work by:
    - Starting at a **node** (usually the **root**)
    - Moving through the tree (i.e., from a given **node** to its **children**)

Today, we'll talk about:

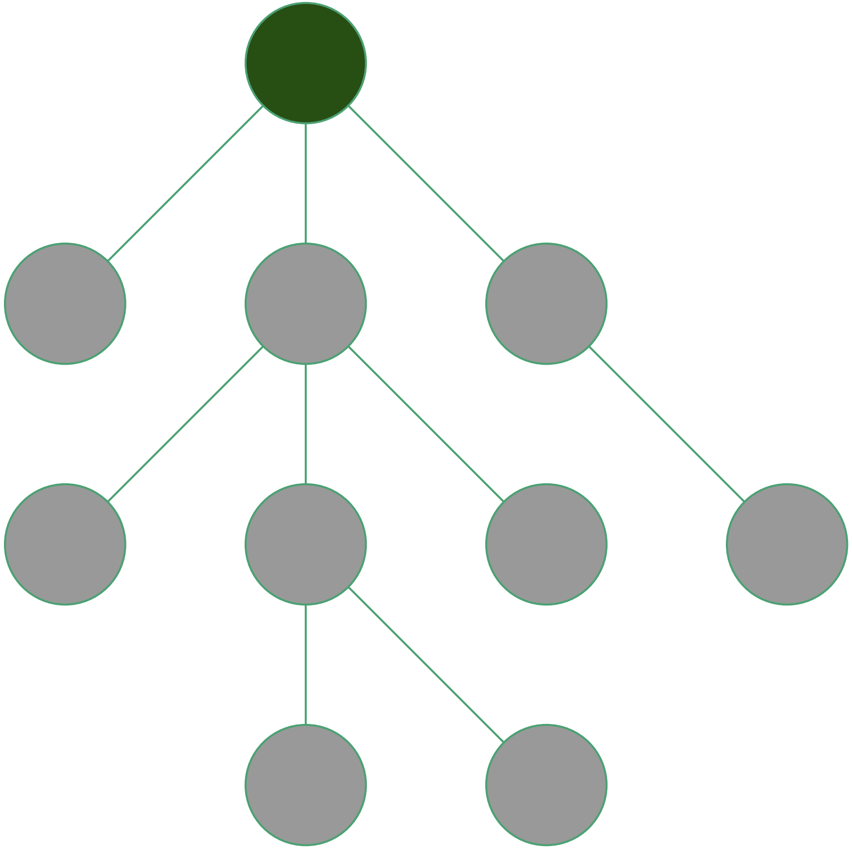
- Basic tree representations
- Tree traversals (a.k.a tree walks)
    - algorithms for moving through all of the nodes in a tree



Trees - Definition

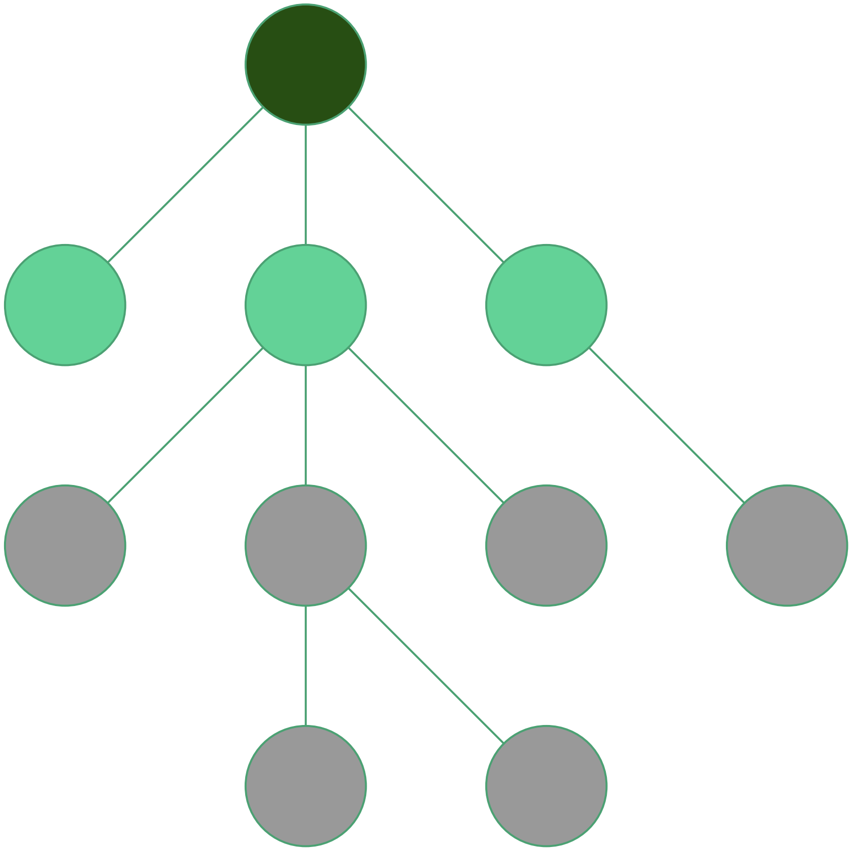
A ***tree*** is a ***graph*** in which any pair of ***nodes*** has exactly one path between them.

- In computer science, we like to distinguish one of these nodes as the ***root***
- And we draw the tree with the root on top, the opposite of real trees
    - we like to draw diagrams with the most important things on top



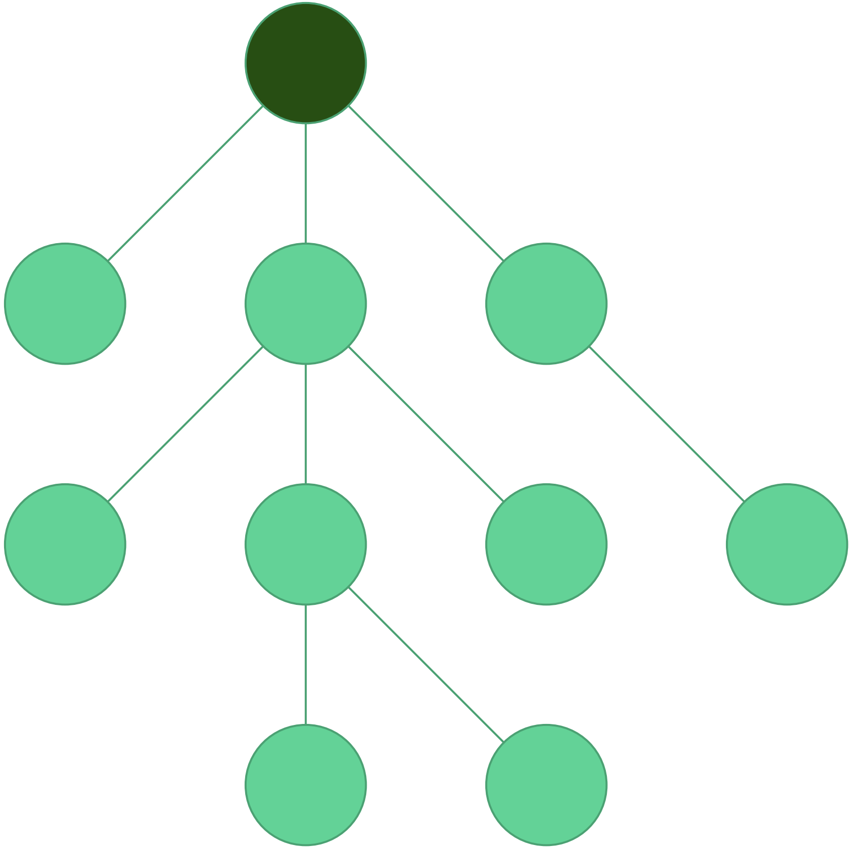
Trees - Children

- The nodes adjacent to a given node, but at the next level down, are called its ***children***



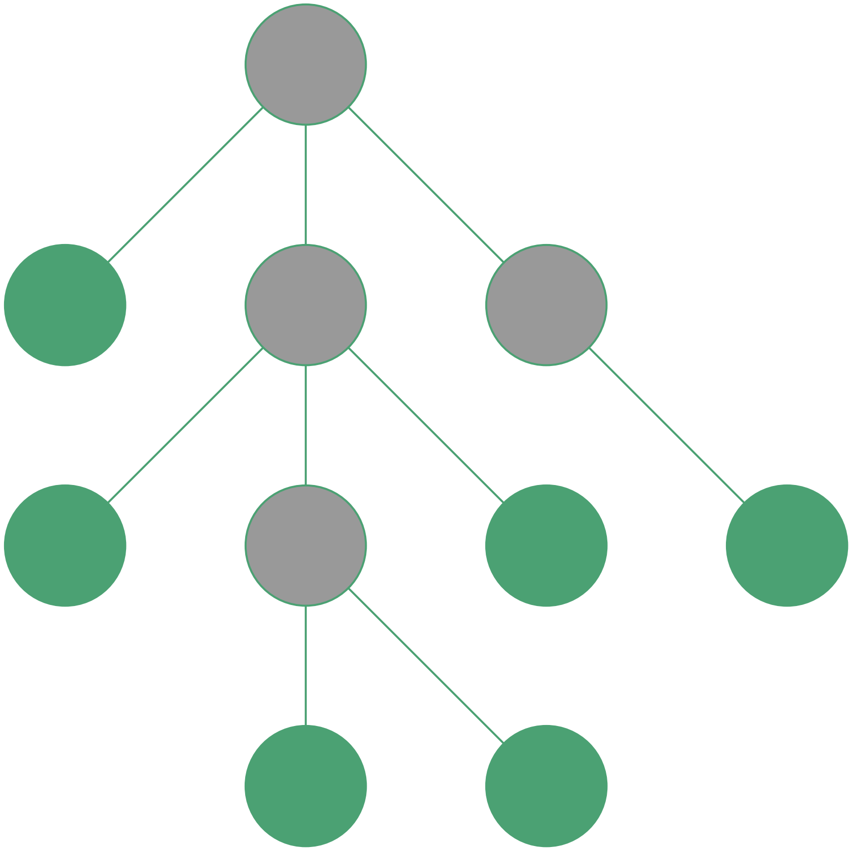
Trees - Children

- The nodes adjacent to a given node, but at the next level down, are called its **children**
- A node’s children, and its children’s children, and so on, are called its ***descendants***



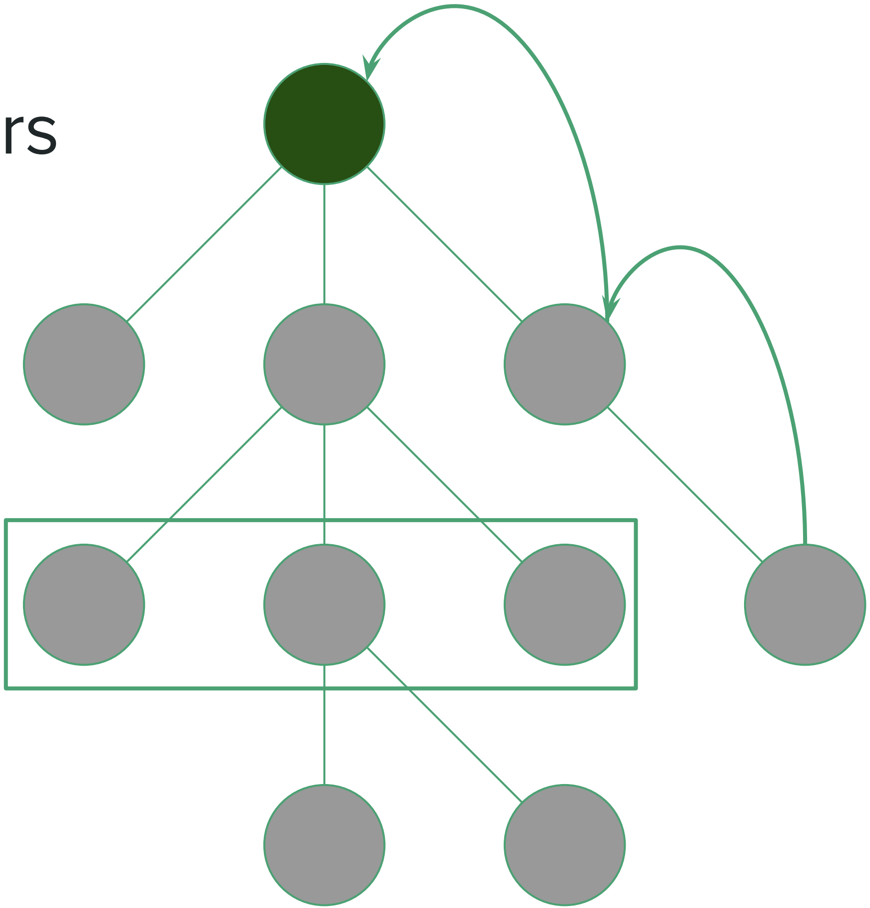
Trees - Children

- The nodes adjacent to a given node, but at the next level down, are called its **children**
- A node’s children, and its children’s children, and so on, are called its **descendants**
- A node with no **children** is called a ***leaf***



Trees - Parents & Ancestors

- The node immediately above a given node is called its ***parent***
    - All nodes have a parent except the root
- The nodes above a given node are called its ***ancestors***
- Nodes with the same parent are called ***siblings***



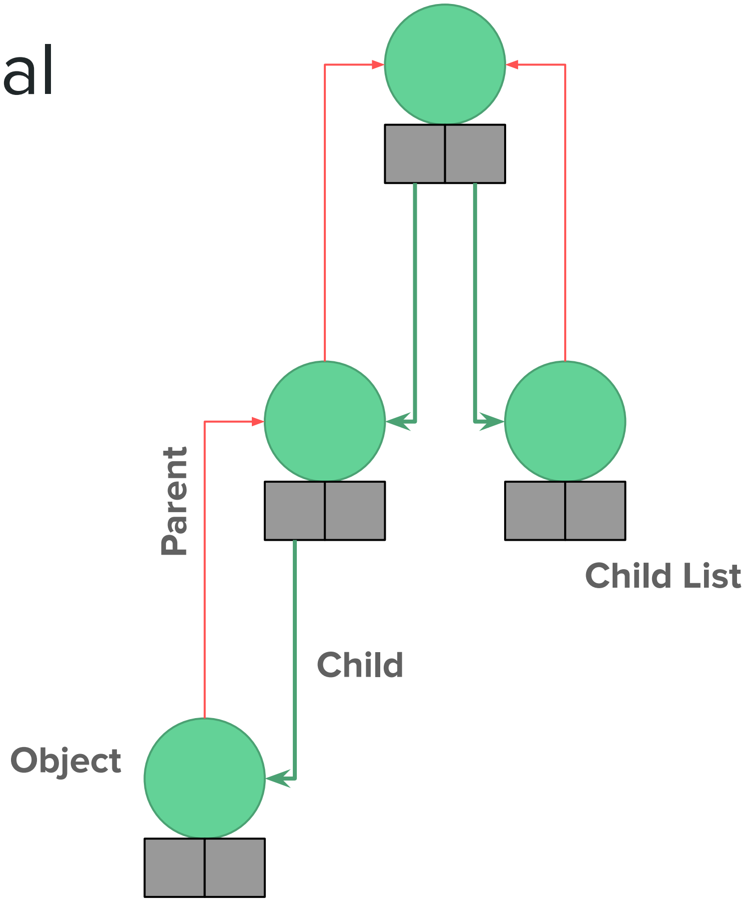
Interlude: Invariants

- In computer science, we refer to something that we know to be true about a program (or some subset of it) as an ***invariant***.
- ***Invariants*** are things that are constant (consistent, reliable, etc.) about a particular bit of a program.
- This can include things like:
  - `“(+ X Y) returns the arithmetic sum of the variables X and Y”`
  - `“Adding an element to the doubly linked list increases its size by 1”`
- ***Invariants*** are useful for reasoning about whether or not a program is running correctly because they also to know what a program is *supposed* to do.
  - If an invariant is broken, then the program is broken.
  - Hey, this sounds like a useful thing for thinking about unit tests…

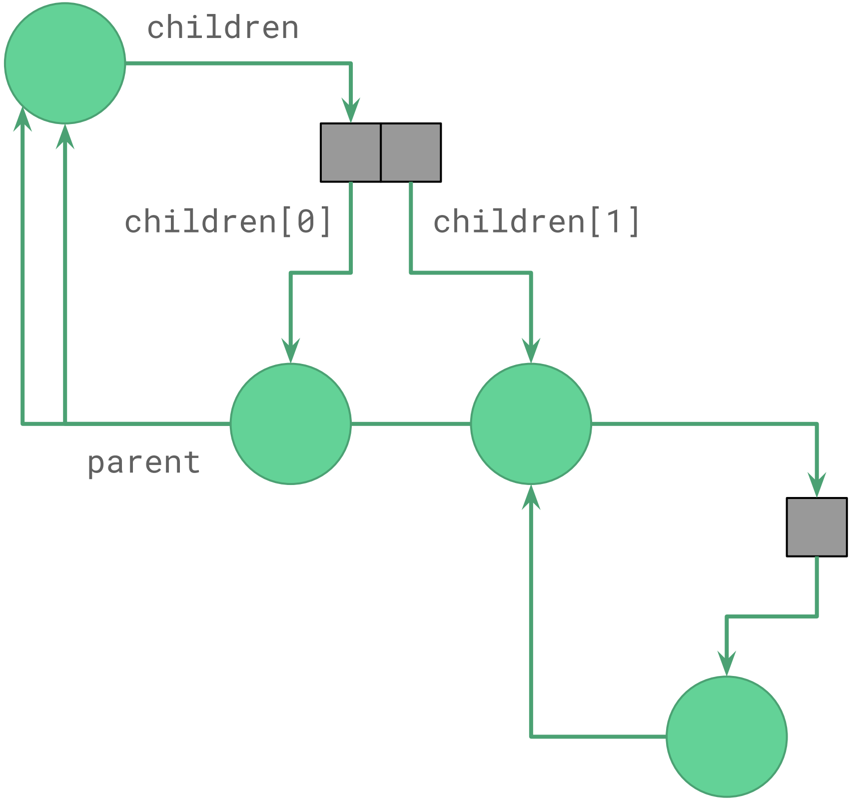
Tree Representations

Tree Representations - General

- Each tree node is an **object** (circles)
- Each node contains:
    - **parent** (red arrow)
    - list of **children** (gray boxes)
        - linked list, array, whatever
    - *anything else you care to remember about the node*
        - numbers, other trees, abstract representations of 3D space, whatever

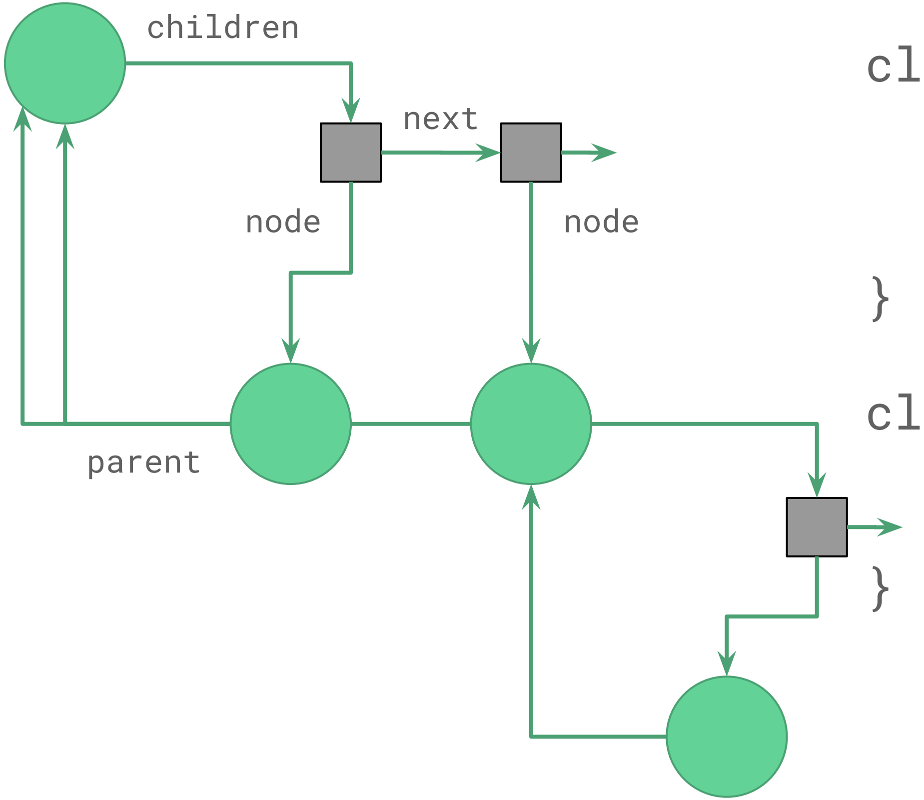


Tree Representations - Arrays



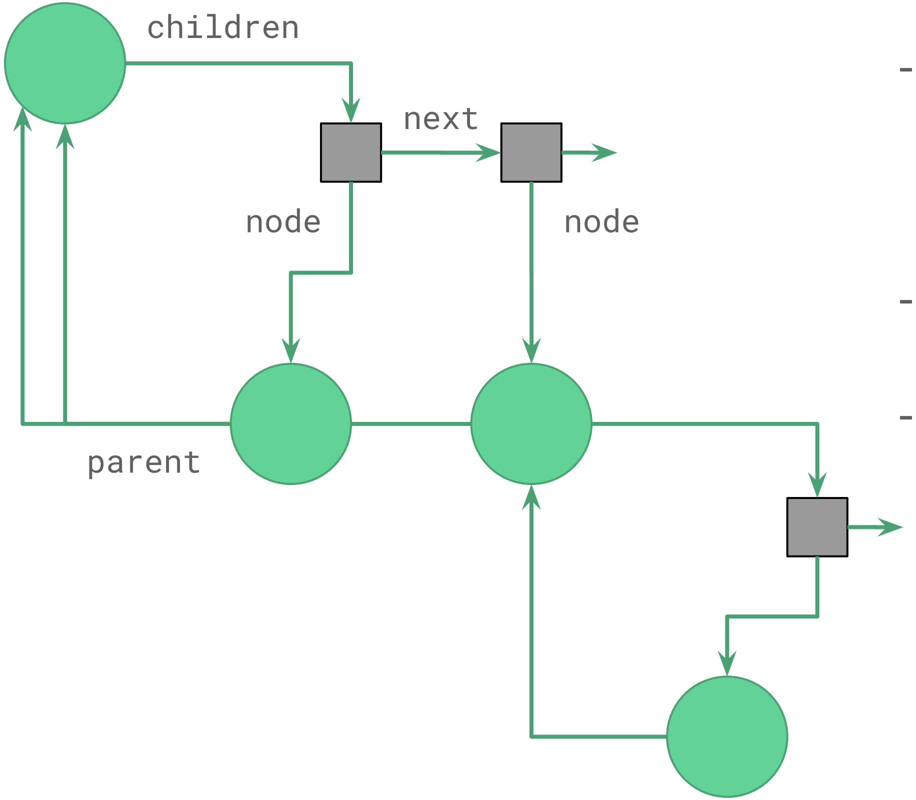

```
class TreeNode {
    TreeNode parent;
    TreeNode[] children;
    // ... other data ...
}
```

Tree Representations - Linked Lists



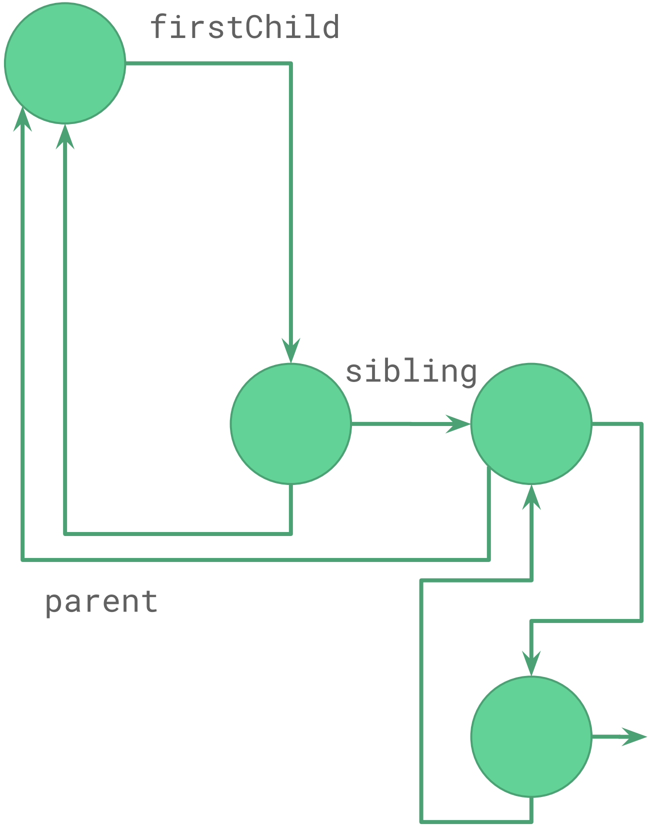

```
class TreeNode {
    TreeNode parent;
    TreeNodeList children;
    // ... other data ...
}

class TreeNodeList {
    TreeNode node;
    TreeNodeList next;
}
```

Tree Representations - Linked Lists



- Note that **exactly one linked list cell points to each node**
    - (Except for the root, which doesn't have a parent)
- So we can move the **`next`** pointer into the node itself
- And remove the linked list cells

Tree Representations - Left Child, Right Sibling

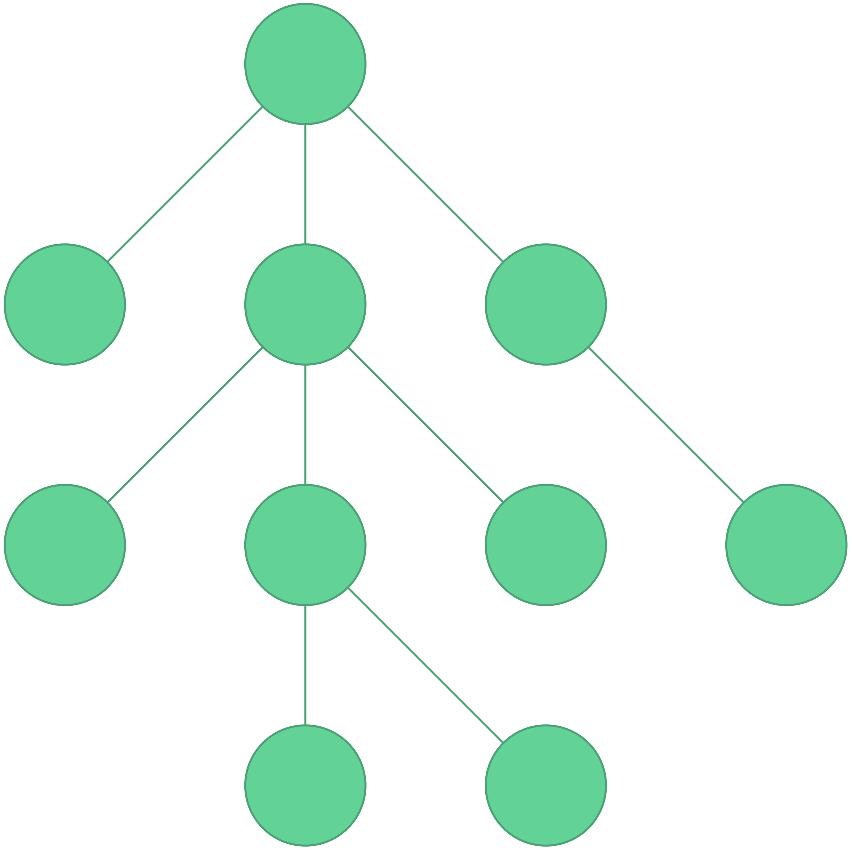



```
class TreeNode {
    TreeNode parent;
    TreeNode firstChild;
    TreeNode sibling;
    // ... other data ...
}
```

- This is called ***left child, right sibling*** representation
- It's pretty elegant, and also very useful in certain applications
- However, it's also a *little* weird, so you might not find it in the wild too often

Tree Traversals (Walks)

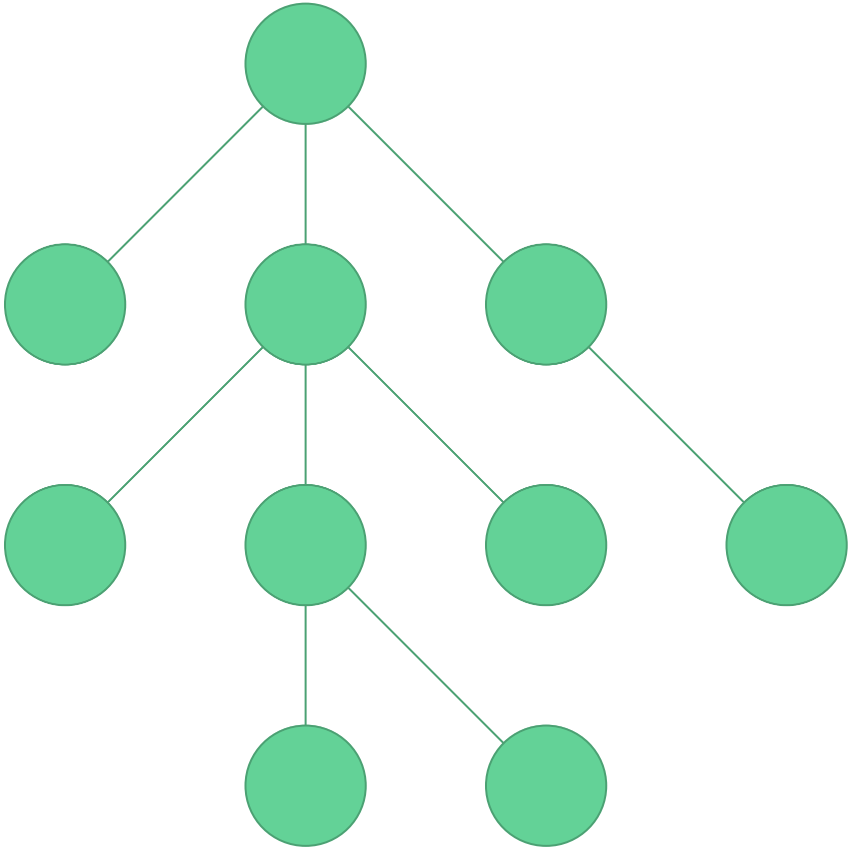
- We generally don't talk about **iterating** over the nodes in a tree
- Instead, we talk about ***walking*** or ***traversing*** over them
- This is because tree walks are generally **recursions**
- It's also because there are many different ways and **orders** in which to **traverse** a tree



Tree Traversals

One can walk a tree in two broad categories of traversal:

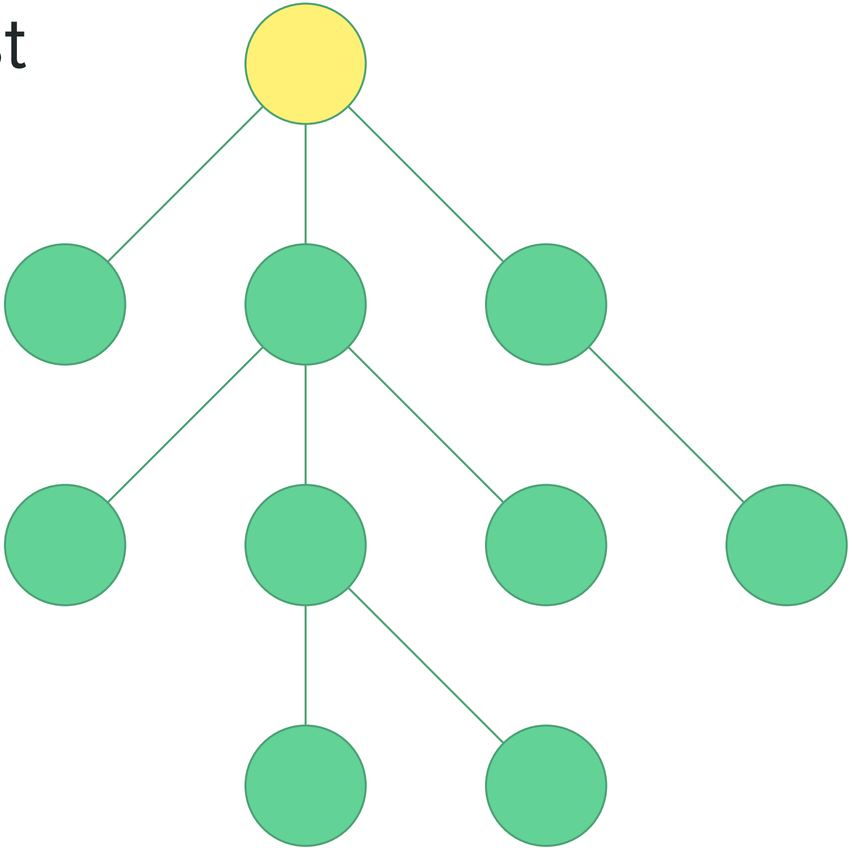
- ***Depth-first*** walk
    - Goes child to child (subtree to subtree)
- ***Breadth-first*** walk
    - Goes level to level



Tree Traversals - Depth-first

Depth-first walk:

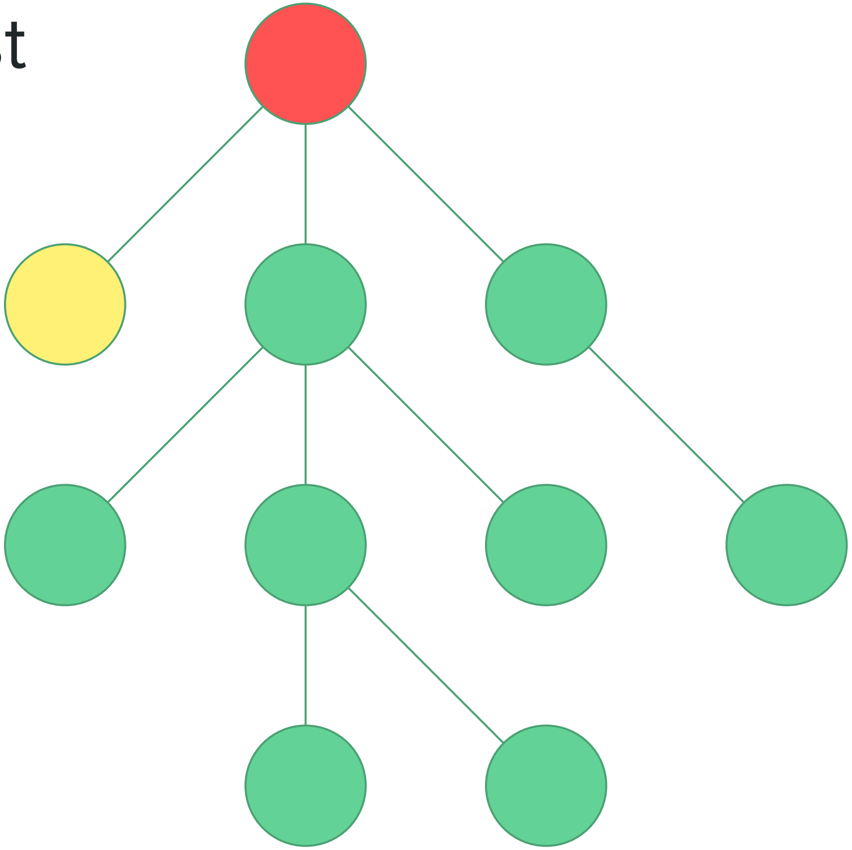
- Start at the root



Tree Traversals - Depth-first

Depth-first walk:

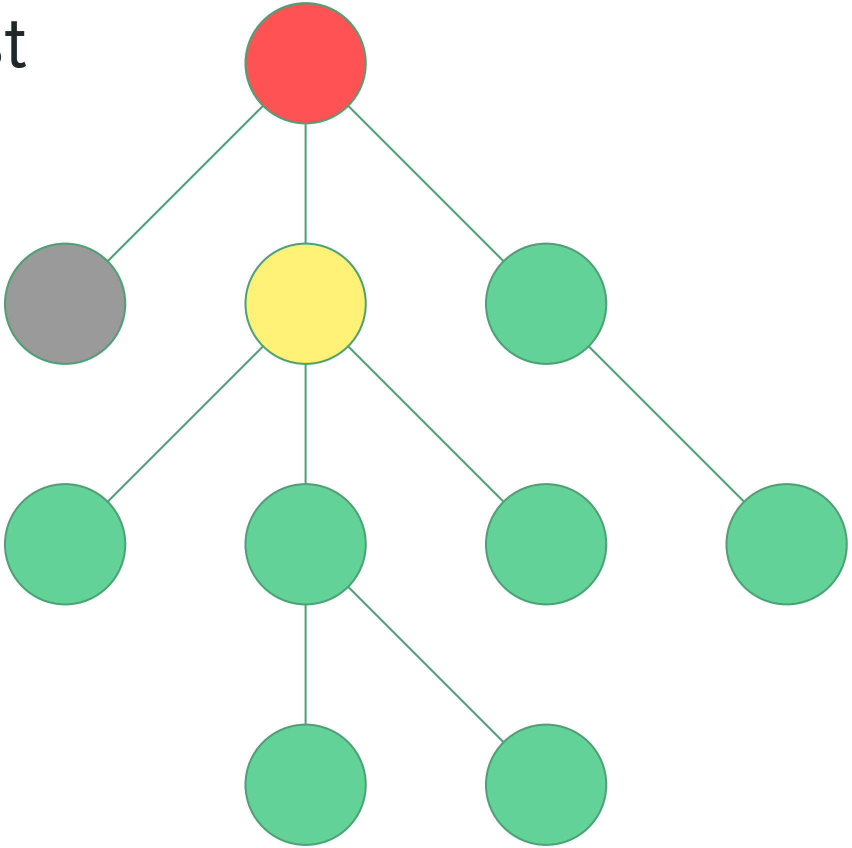
- Start at the root
- Move to its first child



Tree Traversals - Depth-first

Depth-first walk:

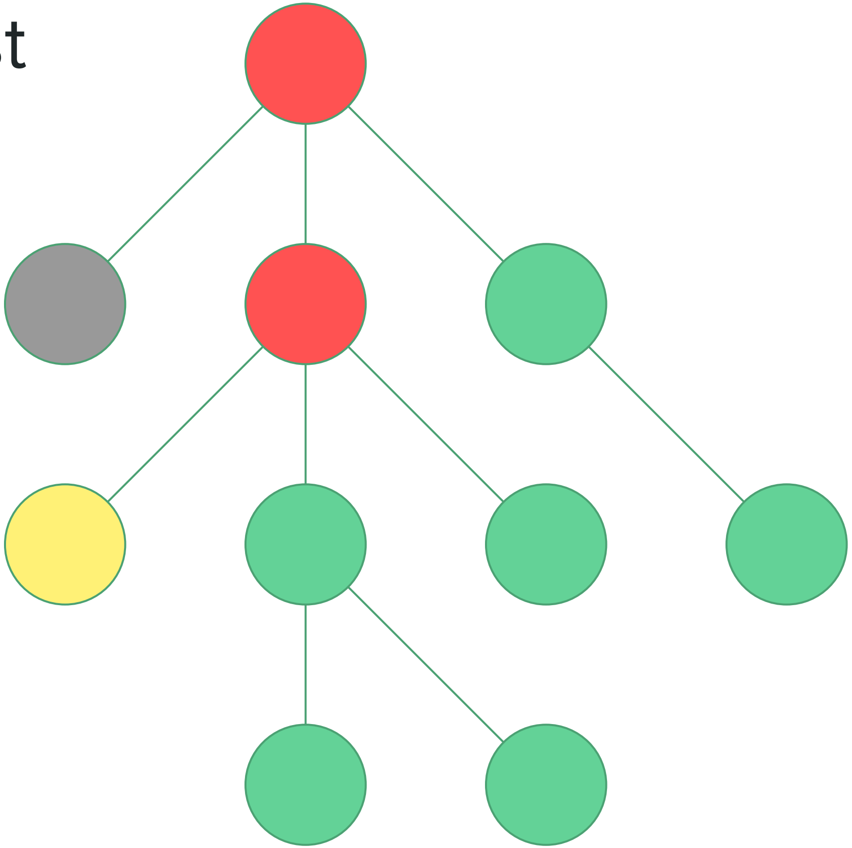
- Start at the root
- Move to its first child
- Which was a leaf, so we move on to its sibling



Tree Traversals - Depth-first

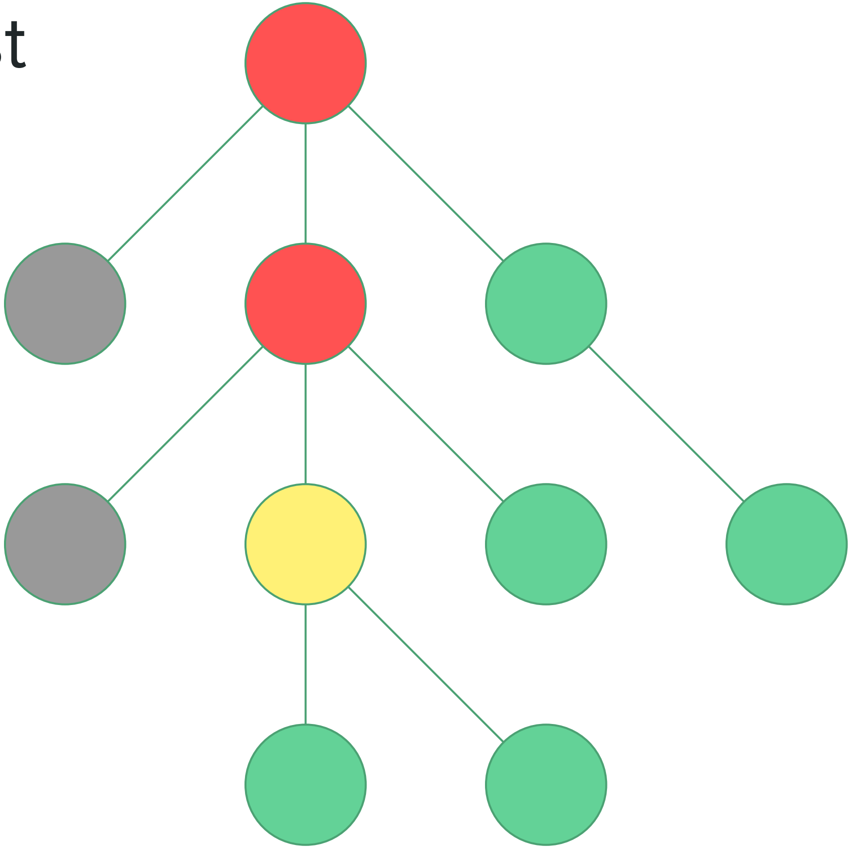
Depth-first walk:

- Start at the root
- Move to its first child
- Which was a leaf, so we move on to its sibling
- Then move to its first child...
    - Hey! This sounds like a recursion...



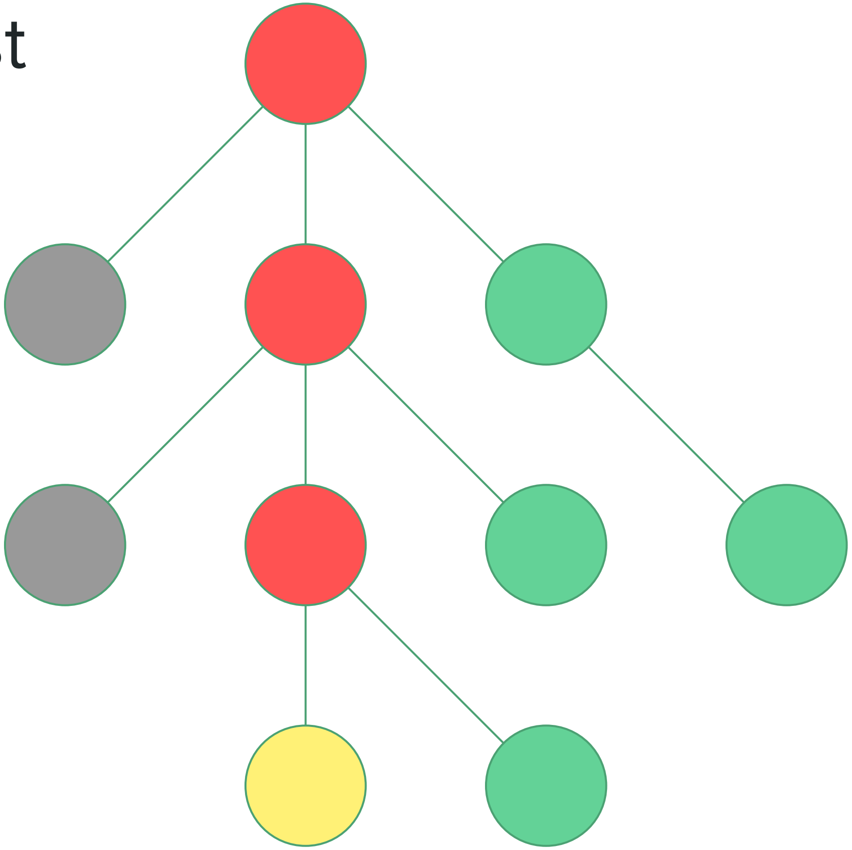
Tree Traversals - Depth-first

Fast-forward



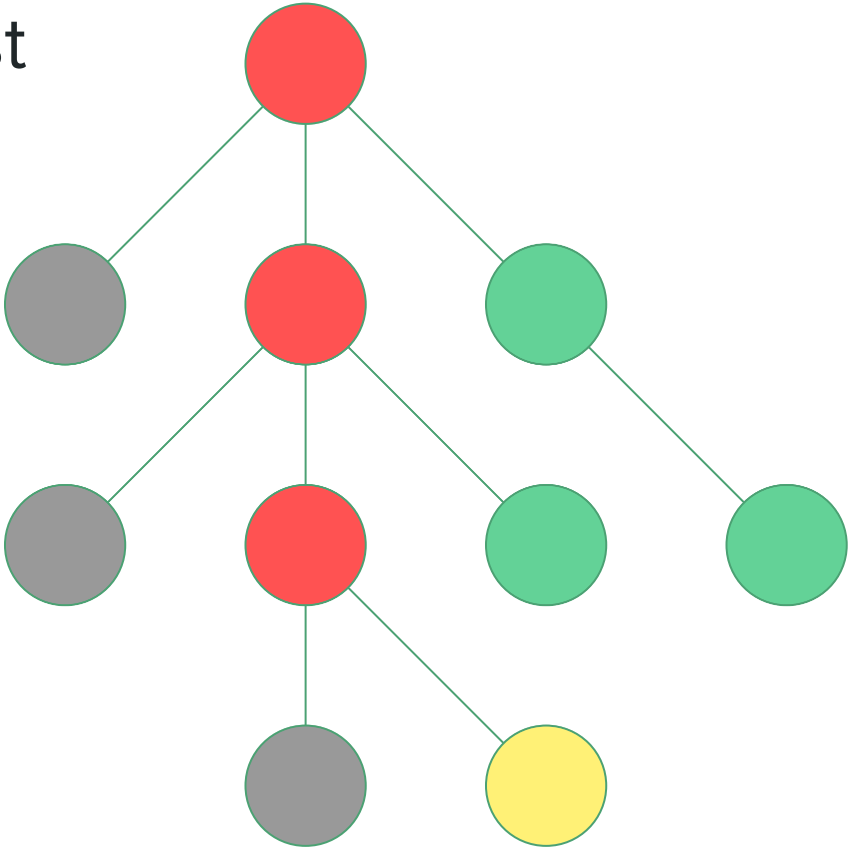
Tree Traversals - Depth-first

Fast-forward



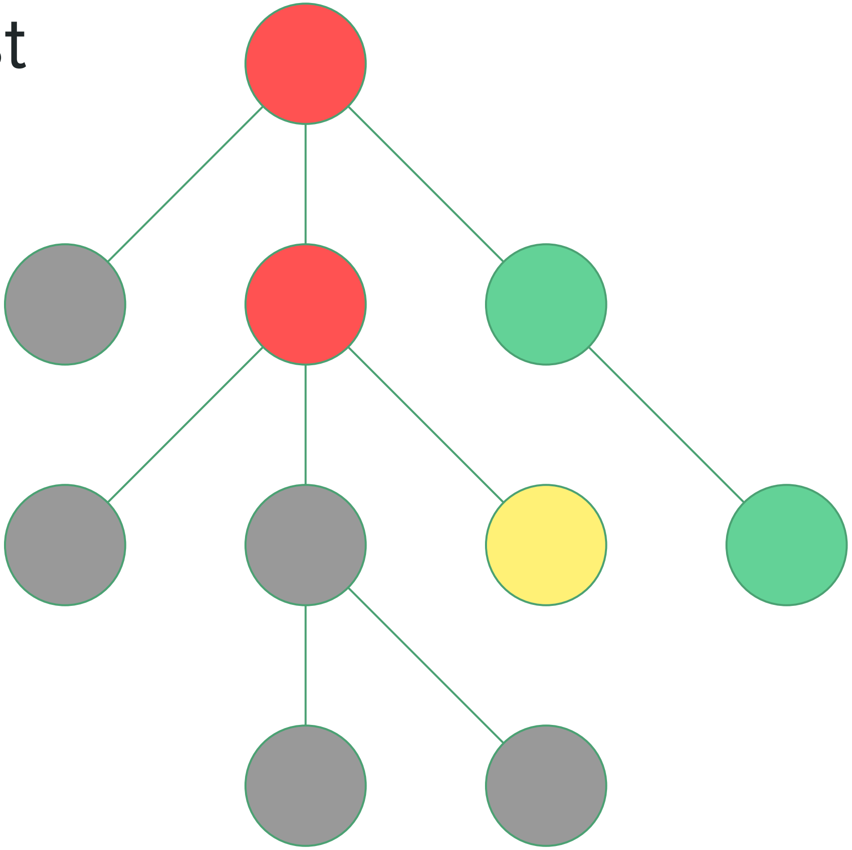
Tree Traversals - Depth-first

Fast-forward



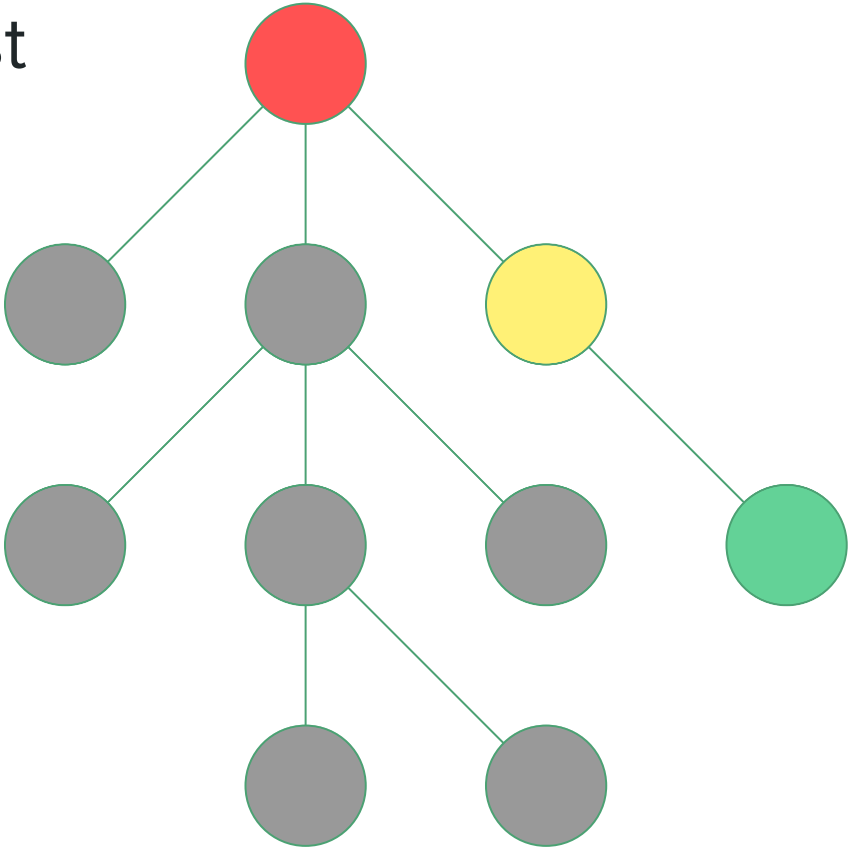
Tree Traversals - Depth-first

Fast-forward



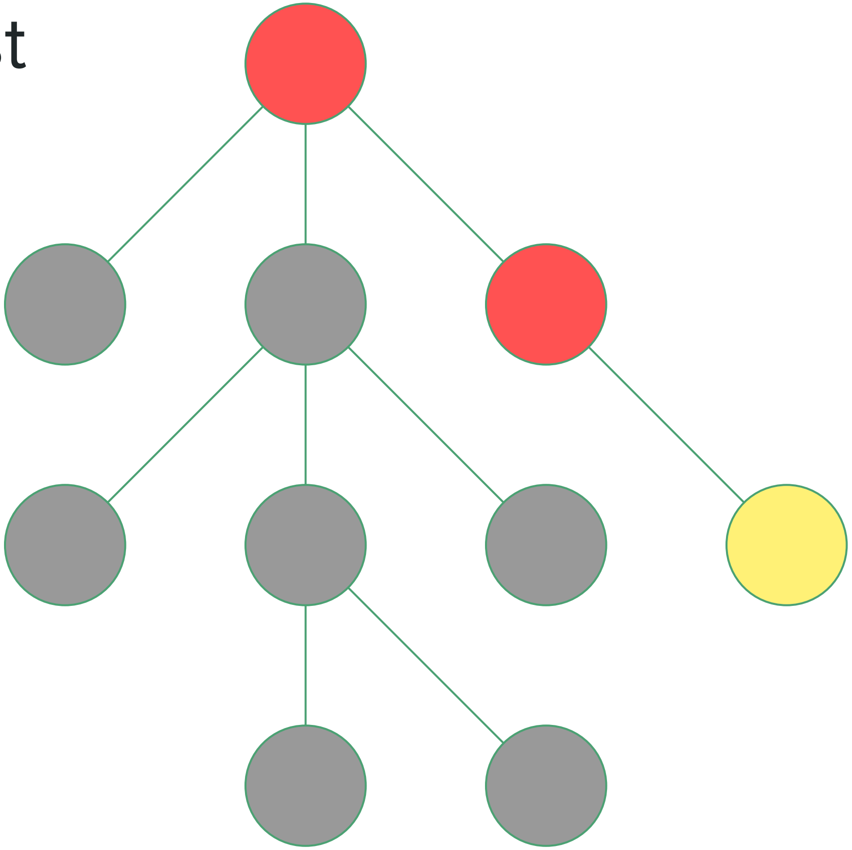
Tree Traversals - Depth-first

Fast-forward



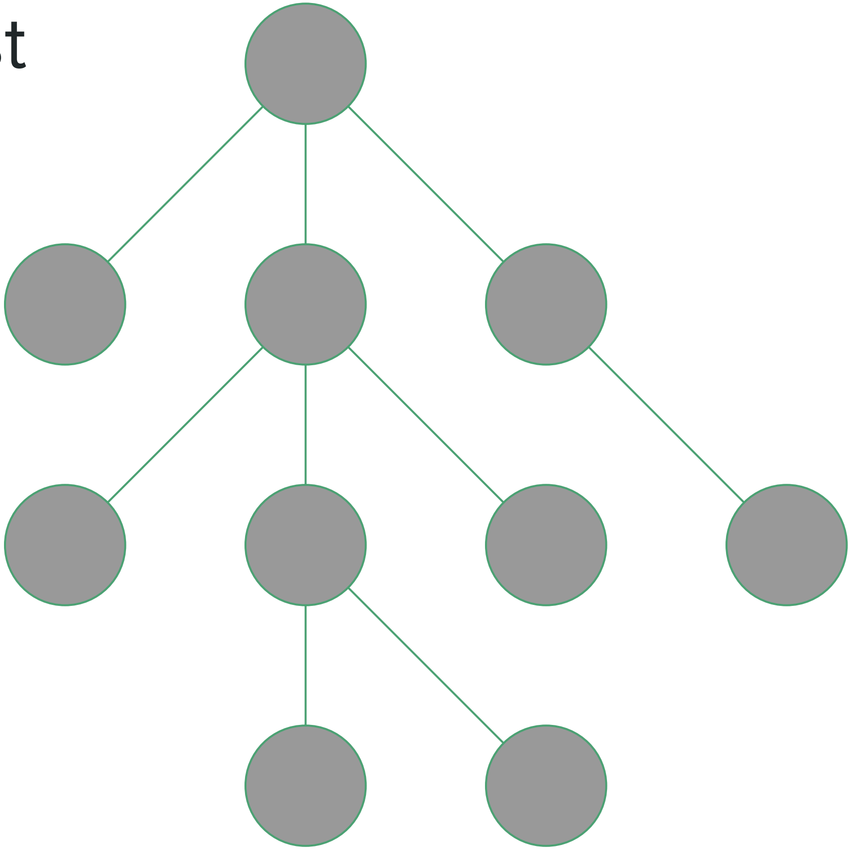
Tree Traversals - Depth-first

Fast-forward



Tree Traversals - Depth-first

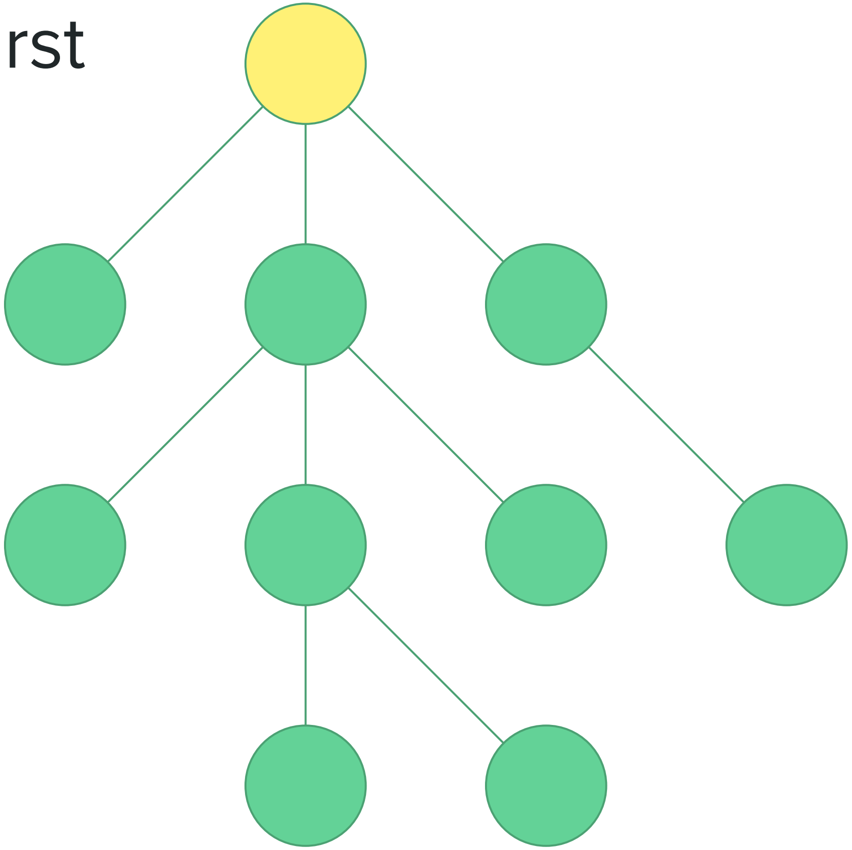
Done



Tree Traversals - Breadth-first

Breadth-first walk:

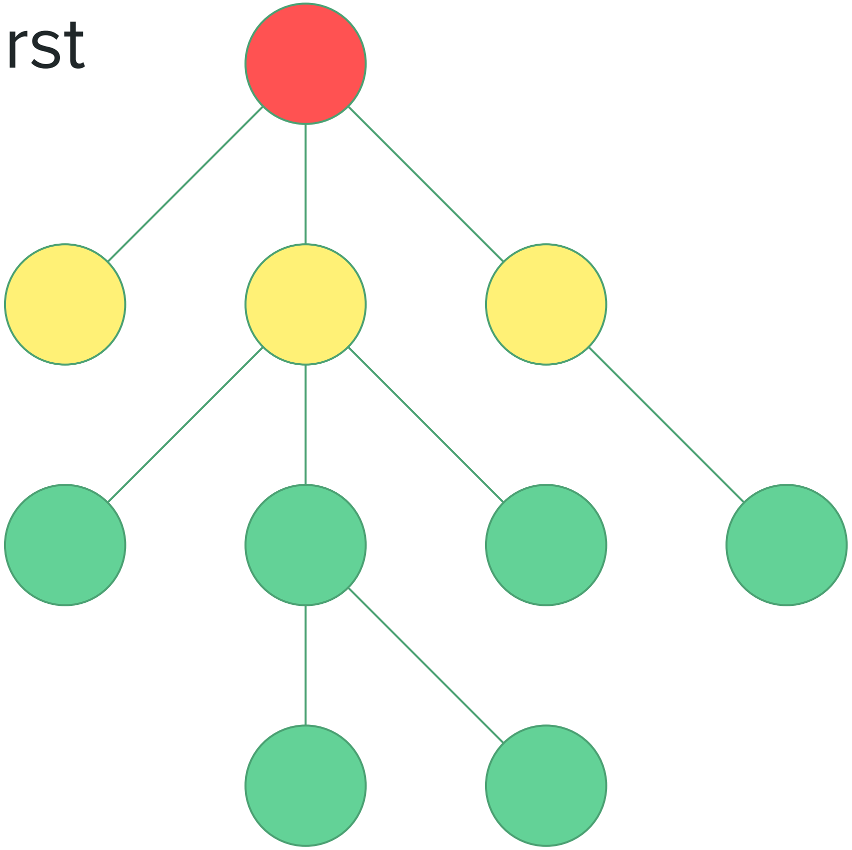
- Start with the root



Tree Traversals - Breadth-first

Breadth-first walk:

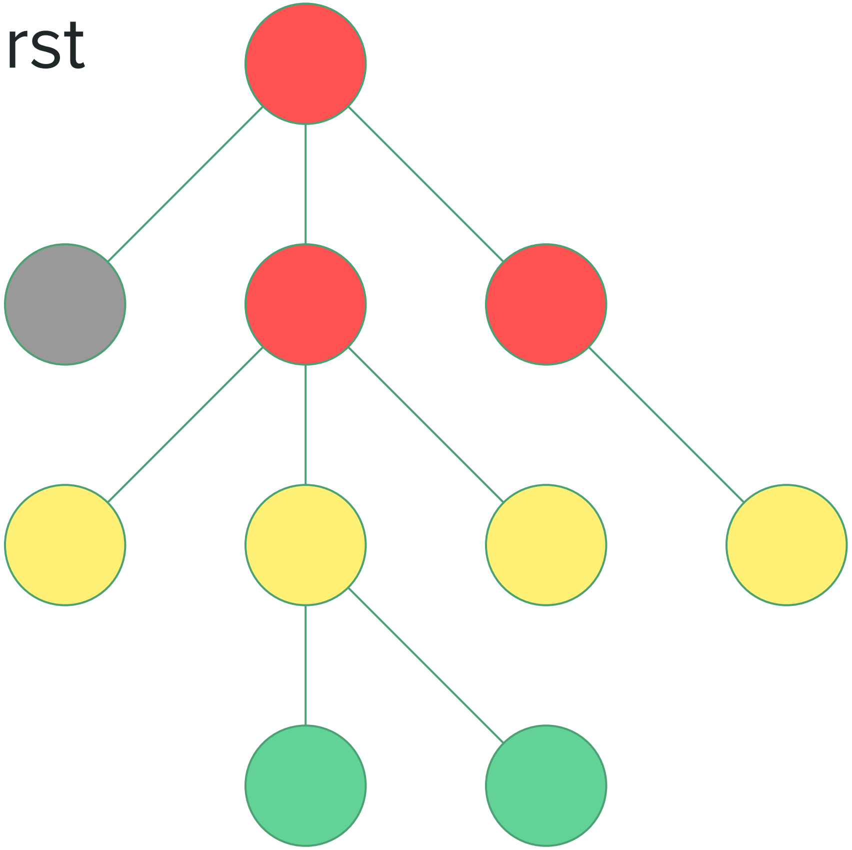
- Start with the root
- Then do the nodes at depth 1



Tree Traversals - Breadth-first

Breadth-first walk:

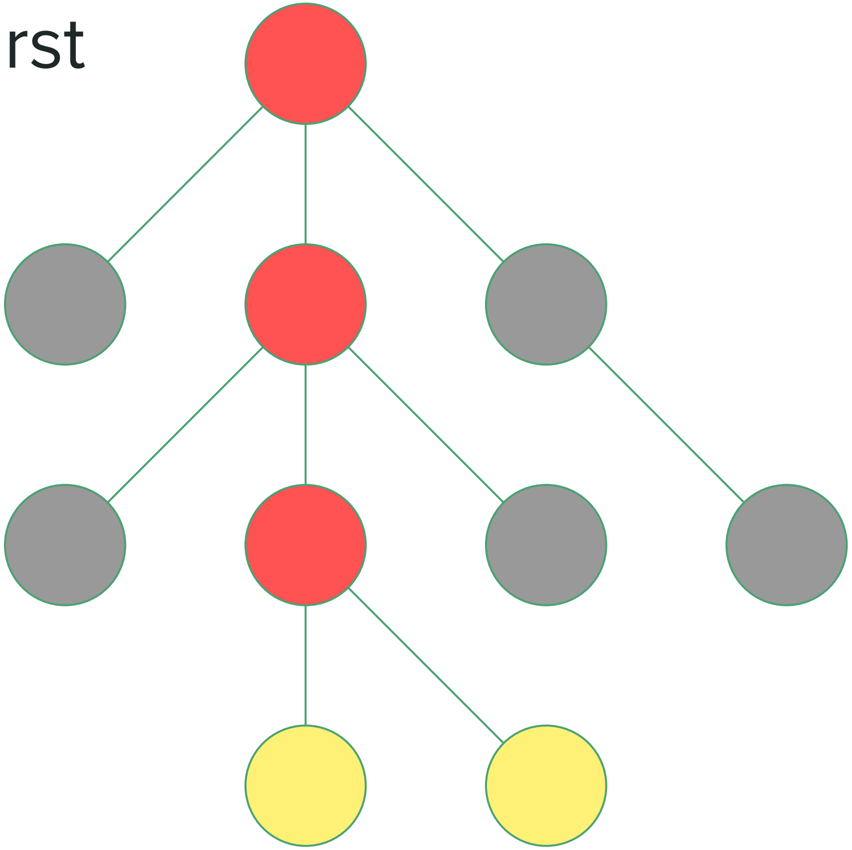
- Start with the root
- Then do the nodes at depth 1
- Then do the nodes at depth 2



Tree Traversals - Breadth-first

Breadth-first walk:

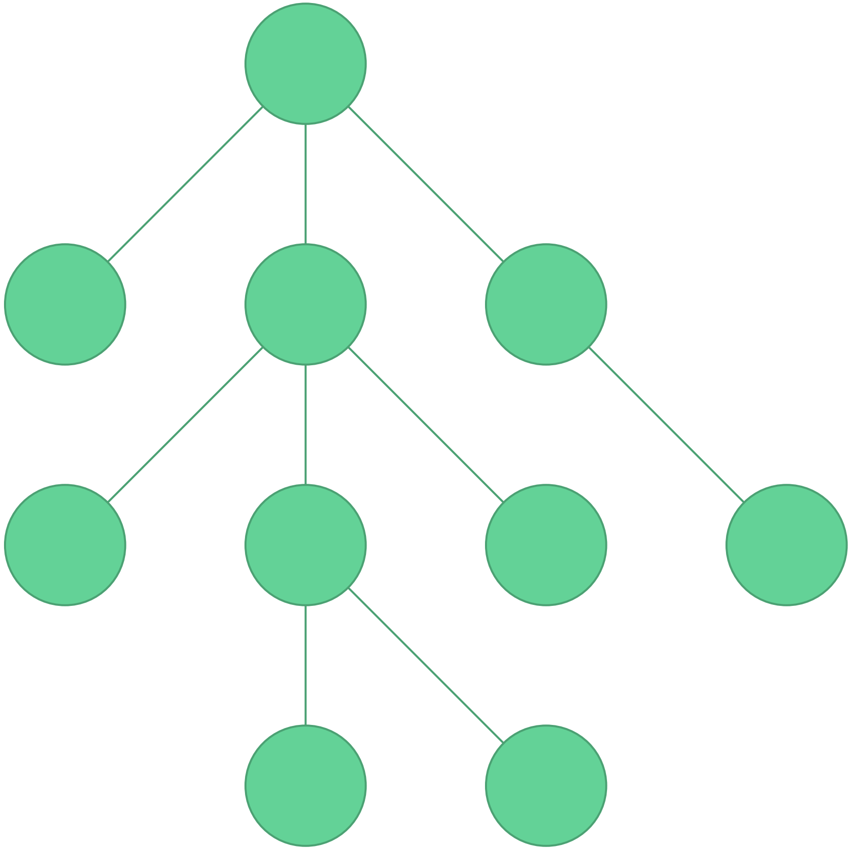
- Start with the root (depth 0)
- Then do the nodes at depth 1
- Then do the nodes at depth 2
- Then do the nodes at depth 3



Tree Traversals

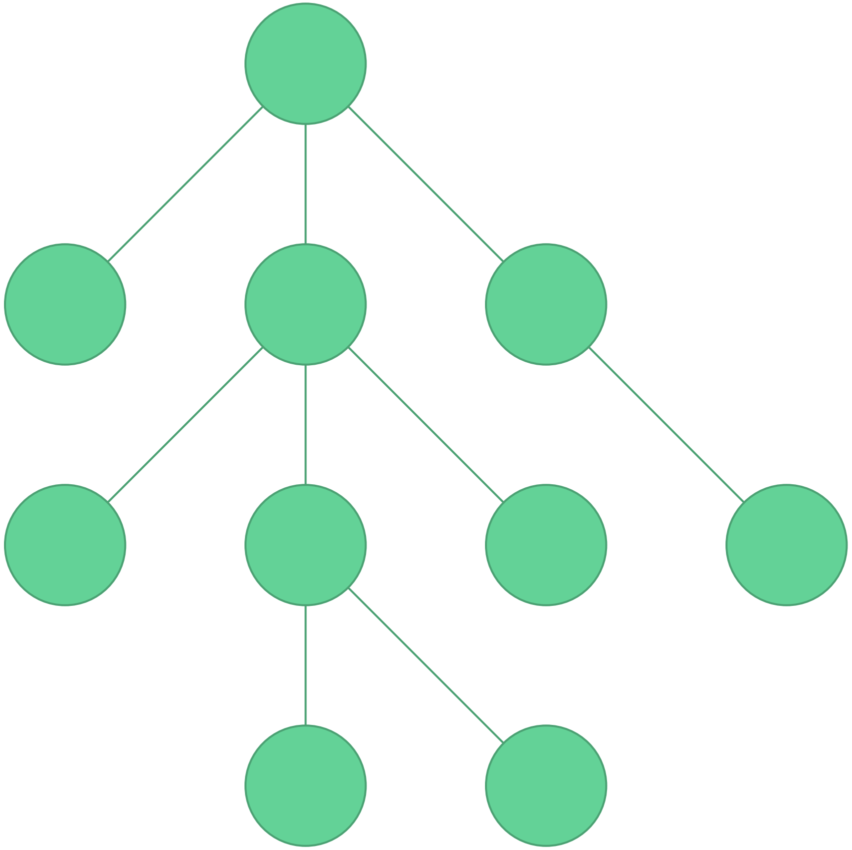
One can walk a tree in two broad categories of traversal:

- ***Depth-first*** walk
    - Goes child to child (subtree to subtree)
- ***Breadth-first*** walk
    - Goes level to level



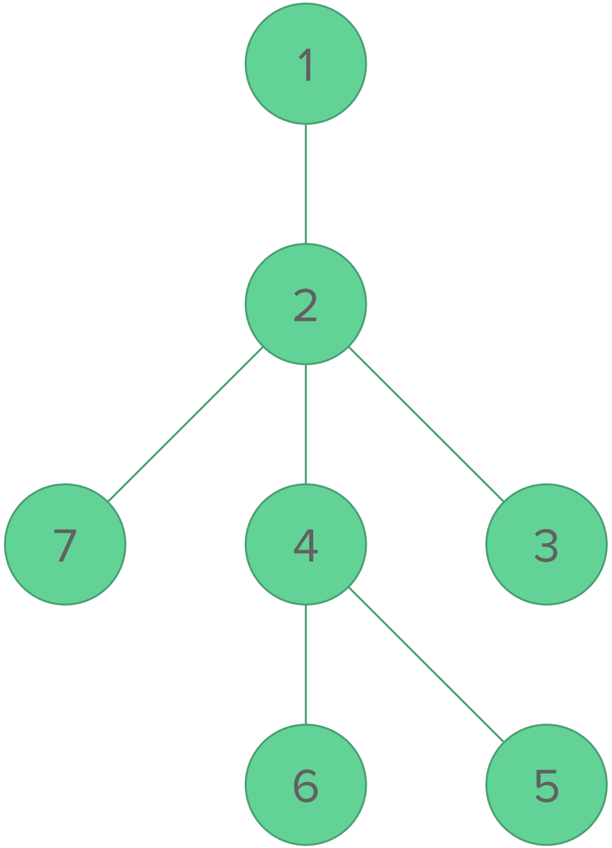
Tree Traversals

- When **depth-first** comes to a node, it walks its children *immediately* and *remembers* what node to come *back to*
- When **breadth-first** comes to a node it *remembers* its children to be walked in the future



Depth-first Walk

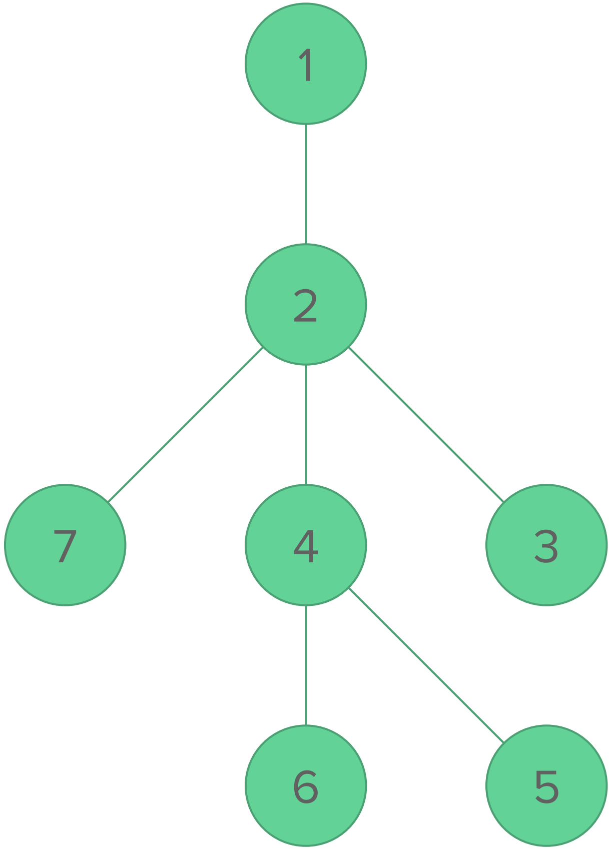
- When we come to a **new** thing, we work on finishing that before our **current** thing is done
- So **new** things take priority over **old** things, but we eventually get back to our **old** things…
- This sounds like a ***stack!***



Depth-first Walk

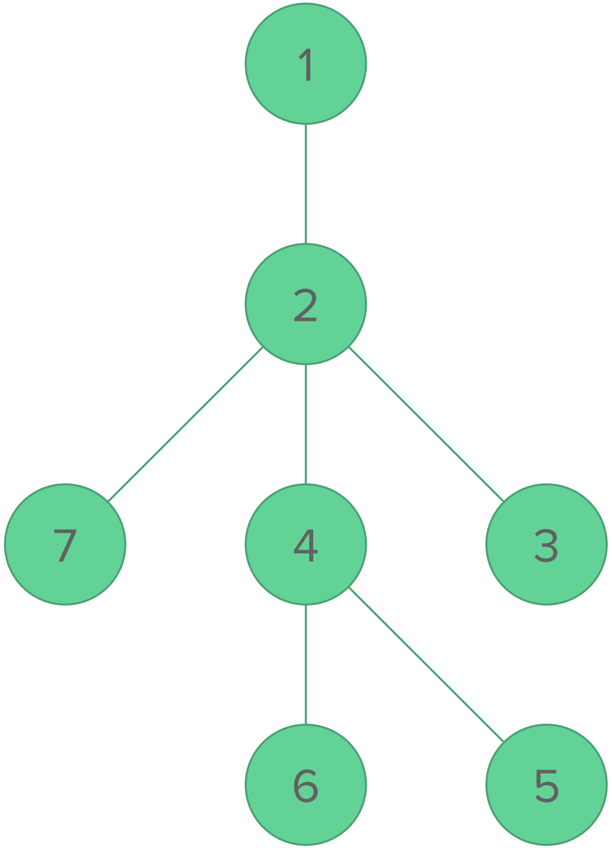
Pseudocode:

```
DepthFirst(root) {
    s = new Stack()
    s.Push(root)
    while(!s.Empty()) {
        n = s.Pop()
        print(n)
        foreach child c of n
            s.Push(n)
}
```



Depth-first Walk

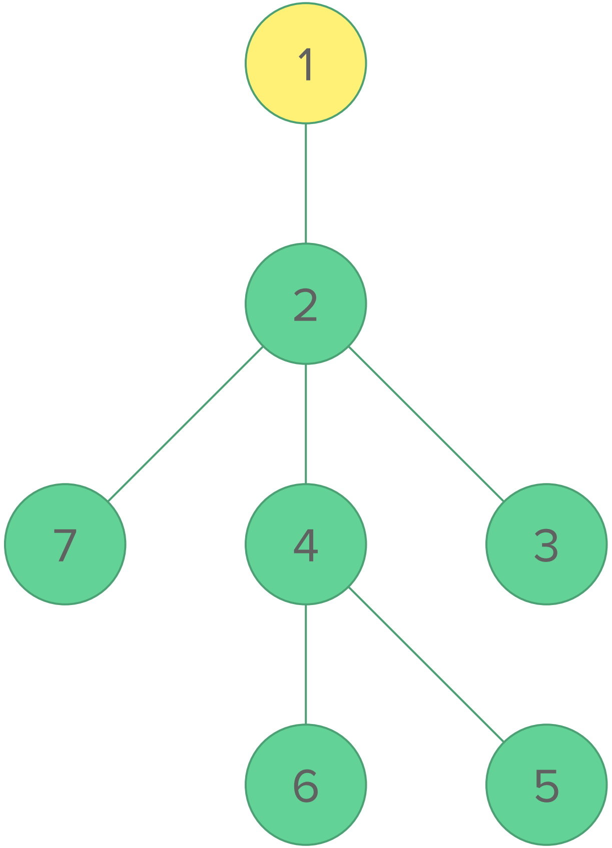
Stack	Output
empty	-



Depth-first Walk

Stack	Output
1	-

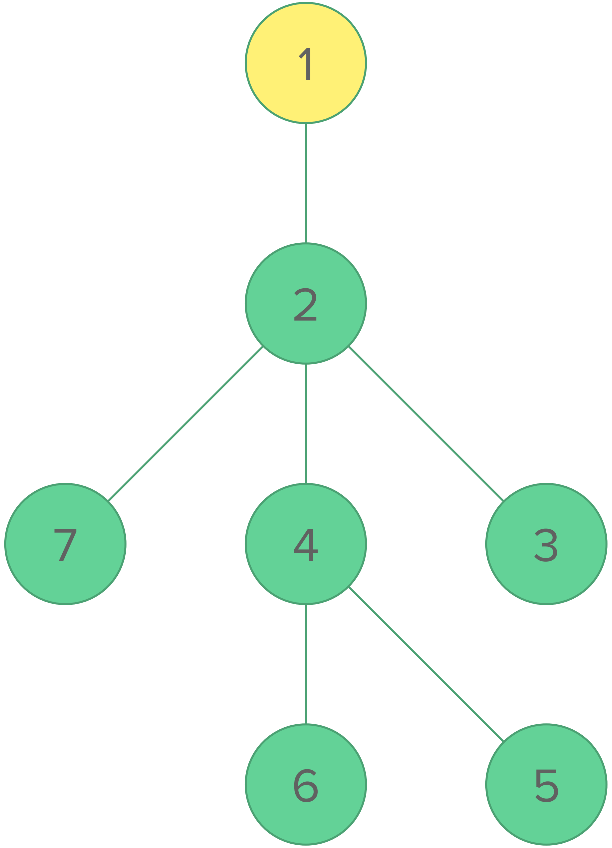
push the root



Depth-first Walk

Stack	Output
empty	1

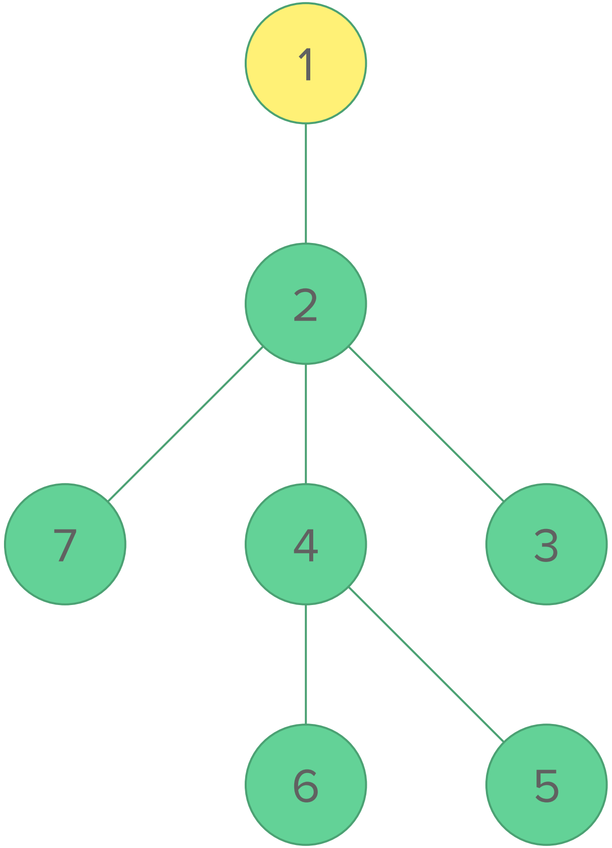
pop and print



Depth-first Walk

Stack	Output
2	1

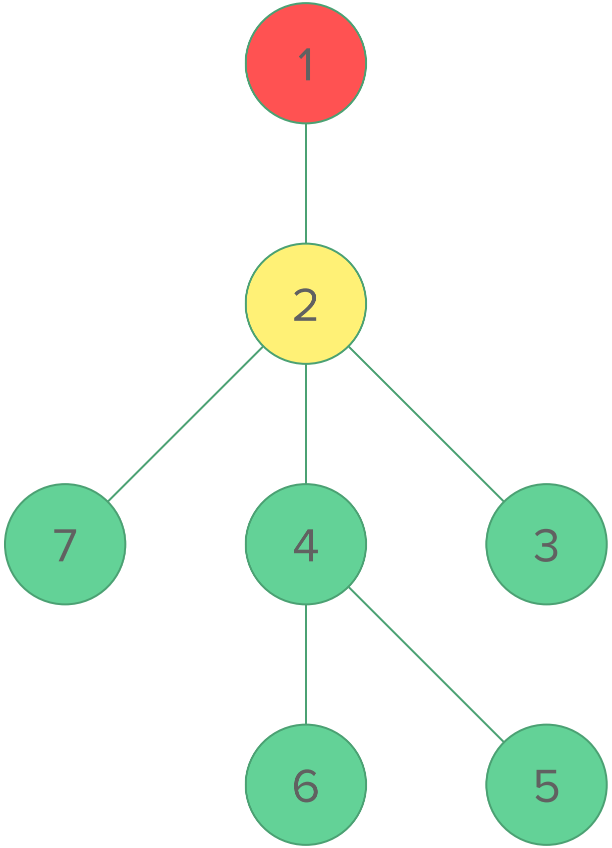
push children



Depth-first Walk

Stack	Output
empty	1 2

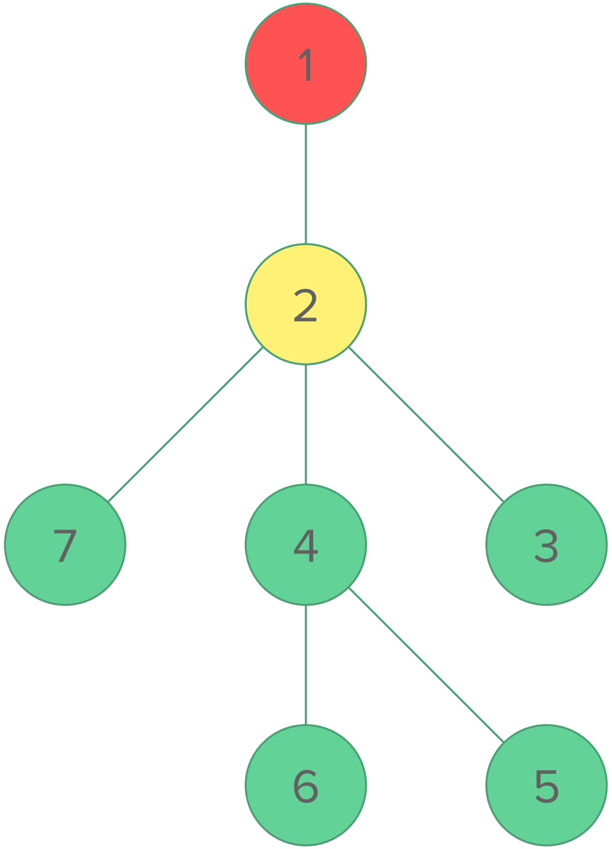
pop and print



Depth-first Walk

Stack	Output
	1
	2
3	
4	
7	

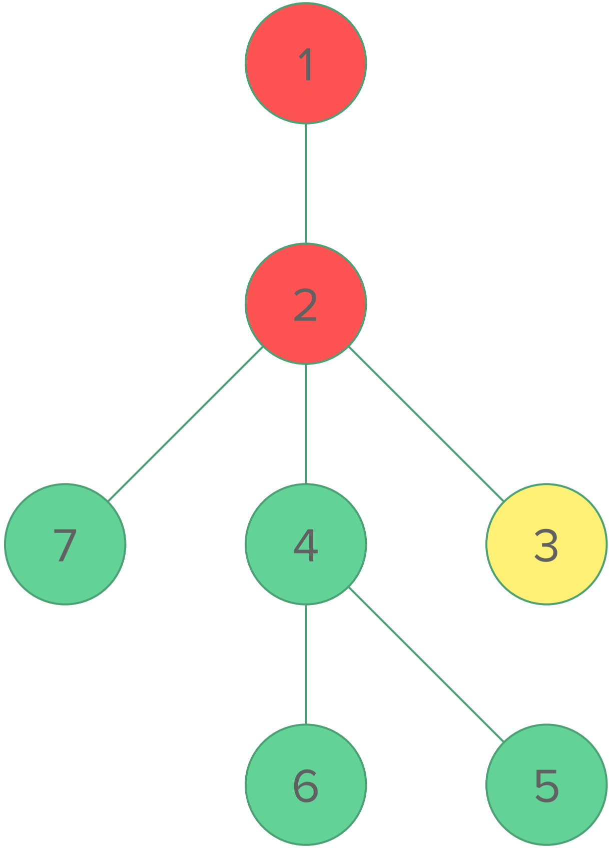
push children



Depth-first Walk

Stack	Output
	1
	2
	3
4	
7	

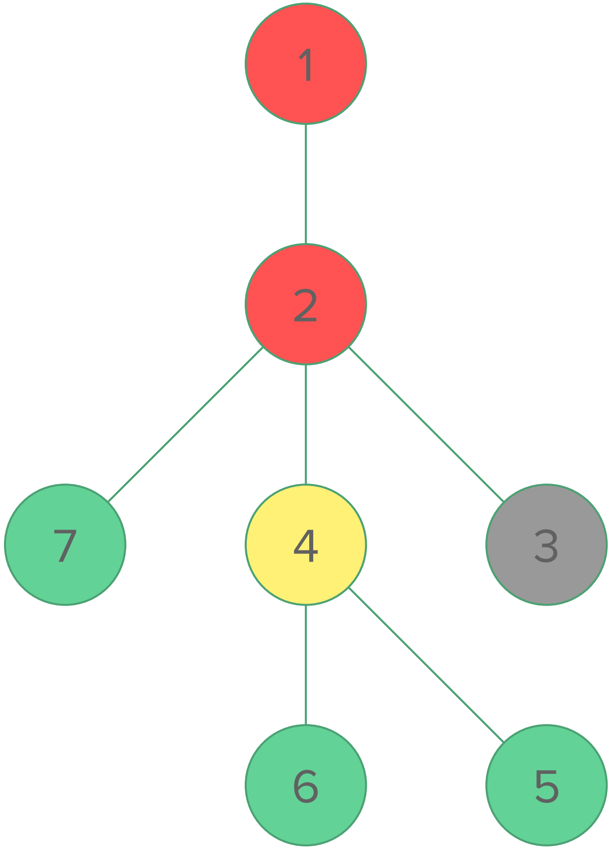
pop and print



Depth-first Walk

Stack	Output
	1
	2
	3
	4
7	

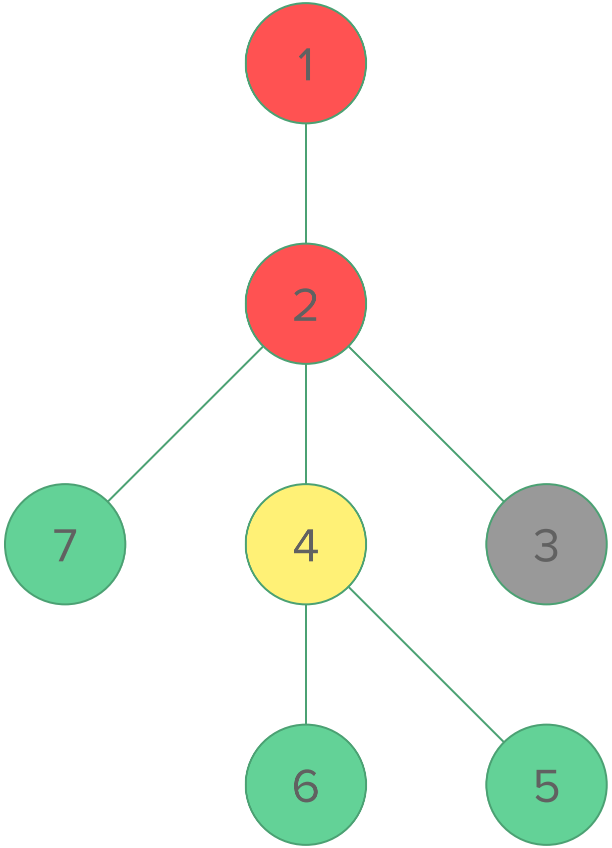
pop and print



Depth-first Walk

Stack	Output
	1
	2
	3
	4
5	
6	
7	

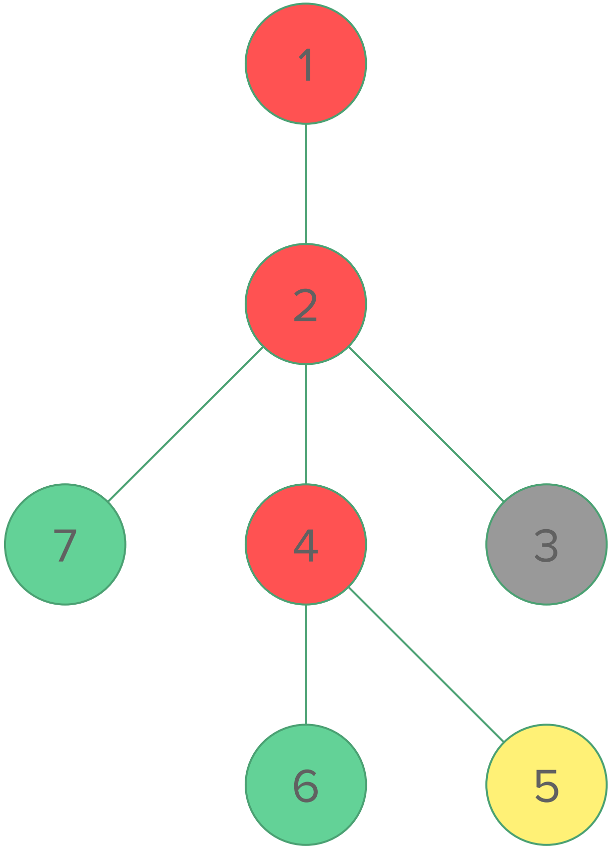
push children



Depth-first Walk

Stack	Output
	1
	2
	3
	4
	5
6	
7	

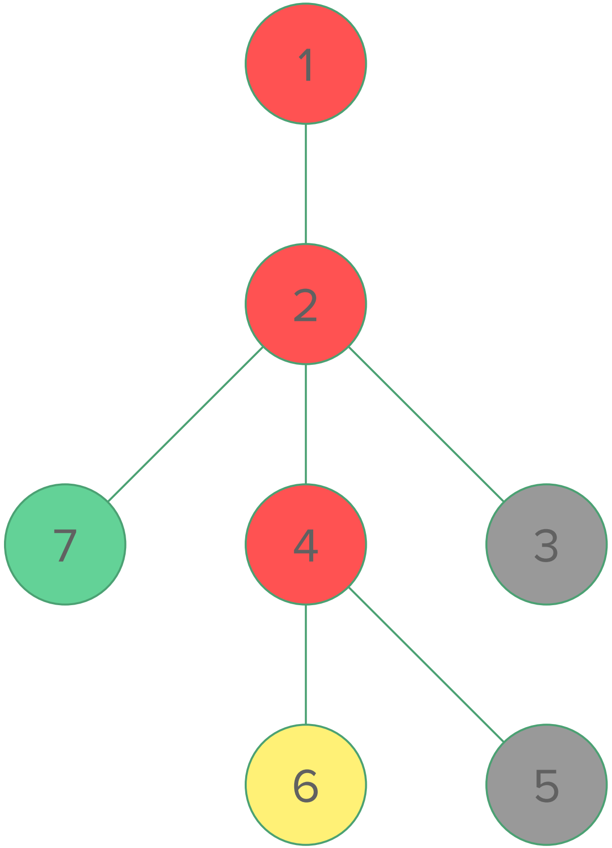
pop and print



Depth-first Walk

Stack	Output
7	1 2 3 4 5 6

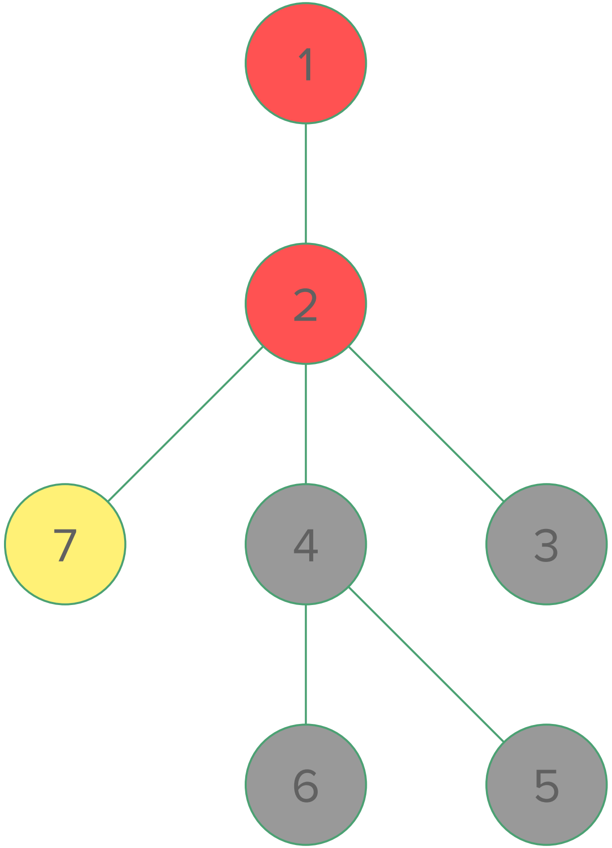
pop and print



Depth-first Walk

Stack	Output
	1
	2
	3
	4
	5
	6
	7
empty	

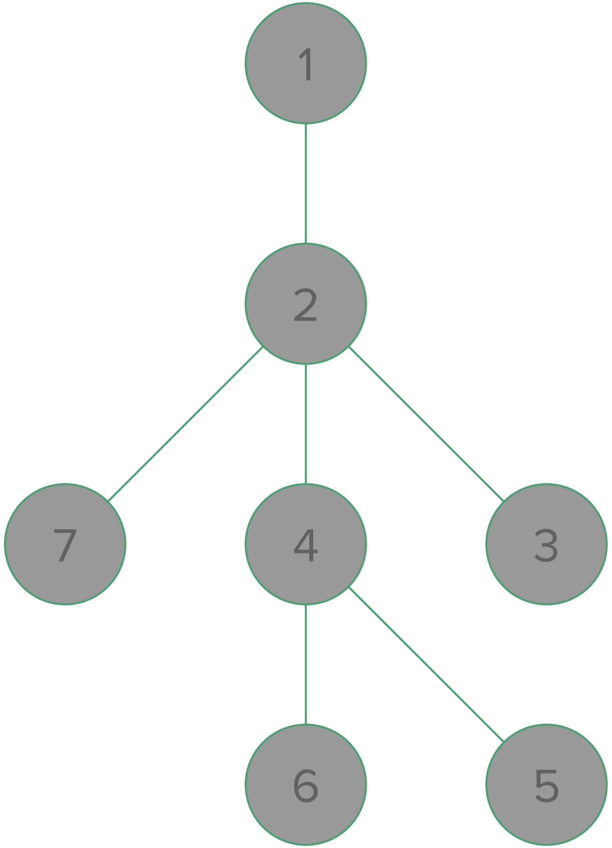
pop and print



Depth-first Walk

Stack	Output
empty	1 2 3 4 5 6 7

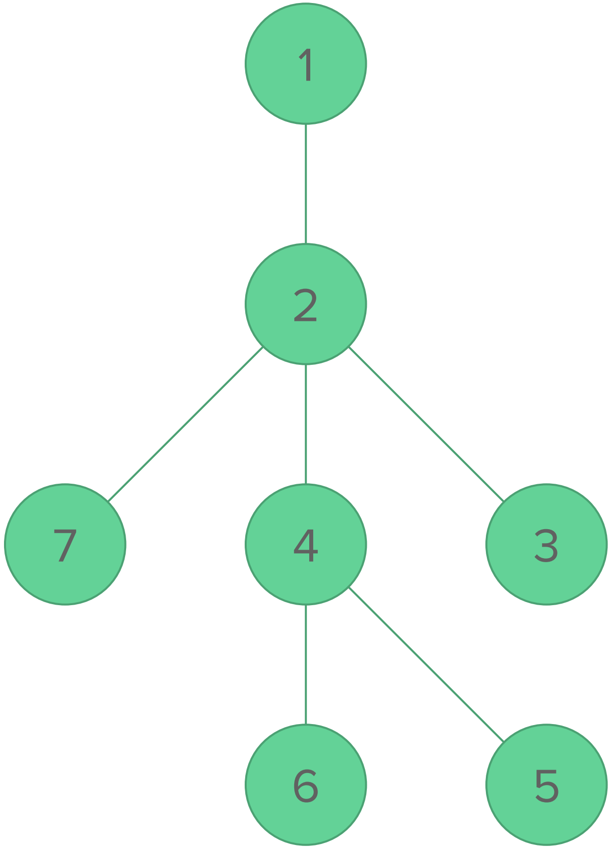
-



Breadth-first Walk

If we replace the stack with a queue:

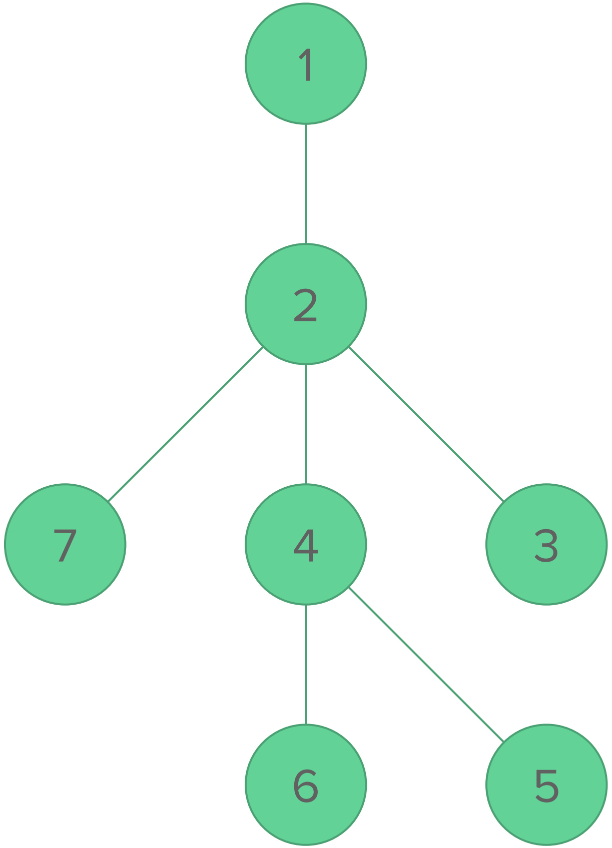
```
DepthFirst(root) {
    s = new Queue()
    s.Enqueue(root)
    while(!s.Empty()) {
        n = s.Dequeue()
        print(n)
        foreach child c of n
            s.Enqueue(n)
}
```



Breadth-first Walk

Queue	1
Output	-

enqueue root



Breadth-first Walk

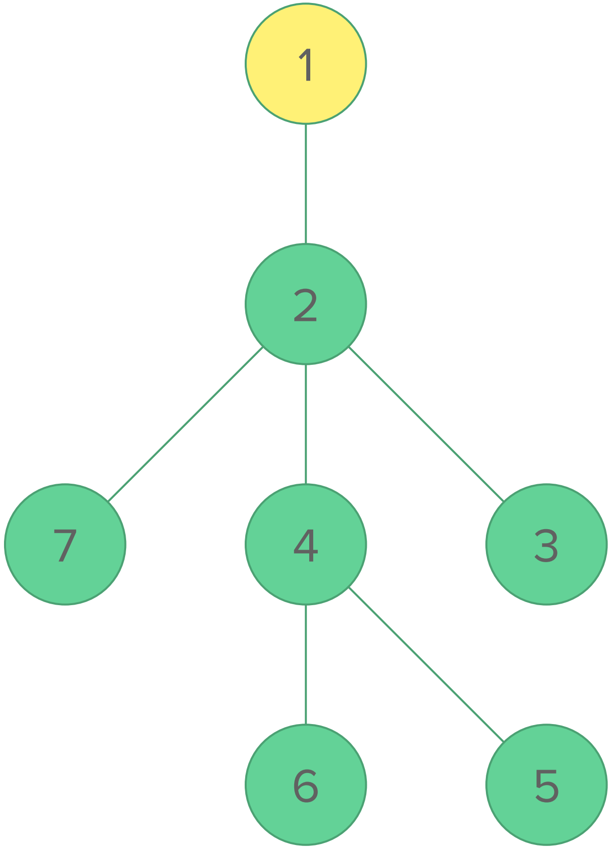
Queue

empty

Output

1

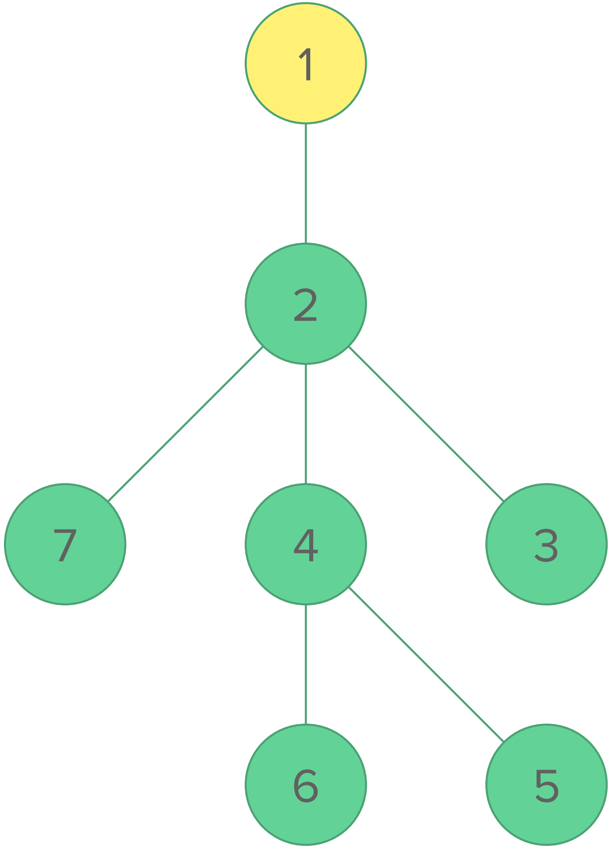
dequeue and print



Breadth-first Walk

Queue	2
Output	1

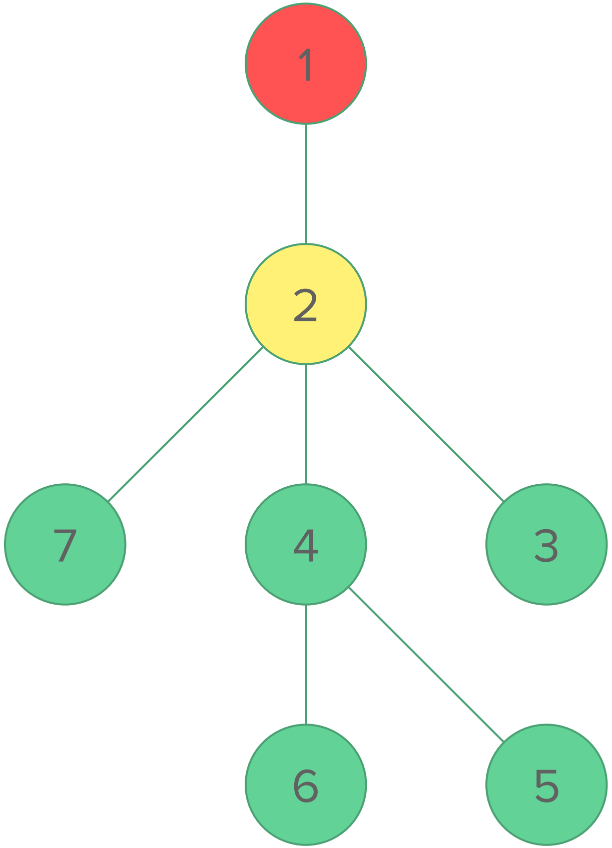
enqueue children



Breadth-first Walk

Queue	empty
Output	1 2

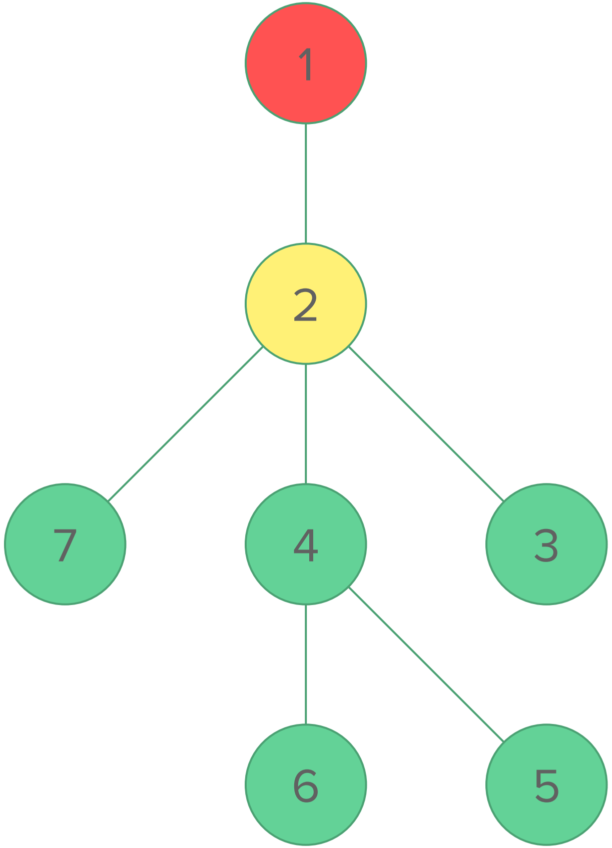
dequeue and print



Breadth-first Walk

Queue	7 4 3
Output	1 2

enqueue children



Breadth-first Walk

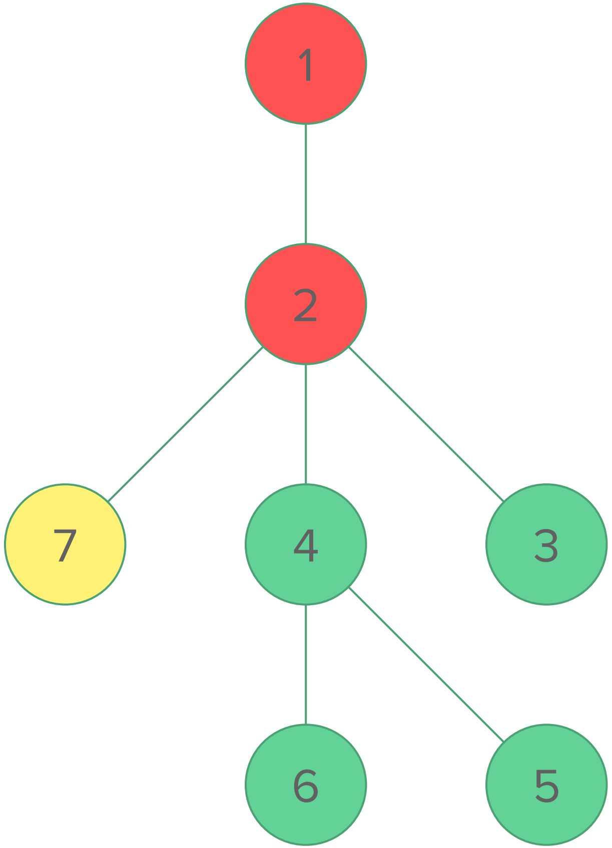
Queue

4 3

Output

1 2 7

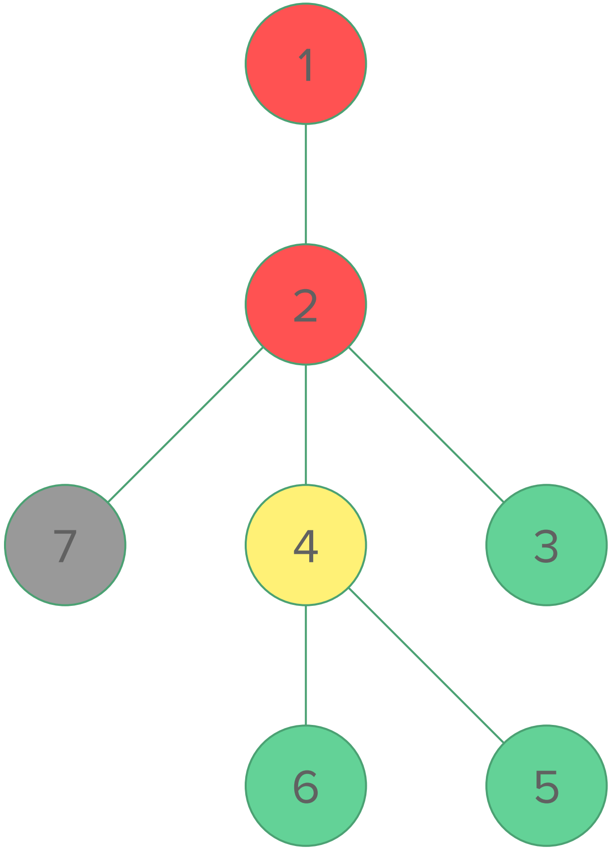
dequeue and print



Breadth-first Walk

Queue	3
Output	1 2 7 4

dequeue and print



Breadth-first Walk

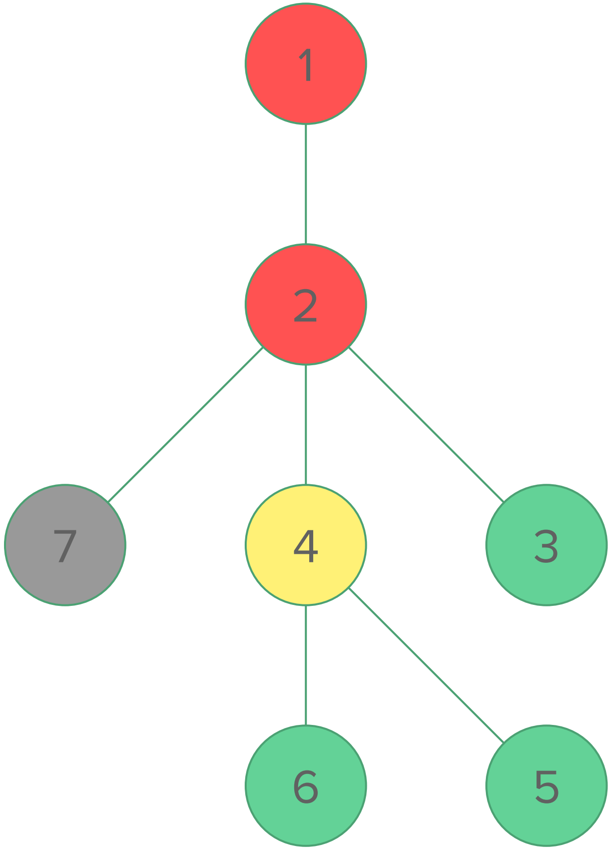
Queue

3 6 5

Output

1 2 7 4

enqueue children

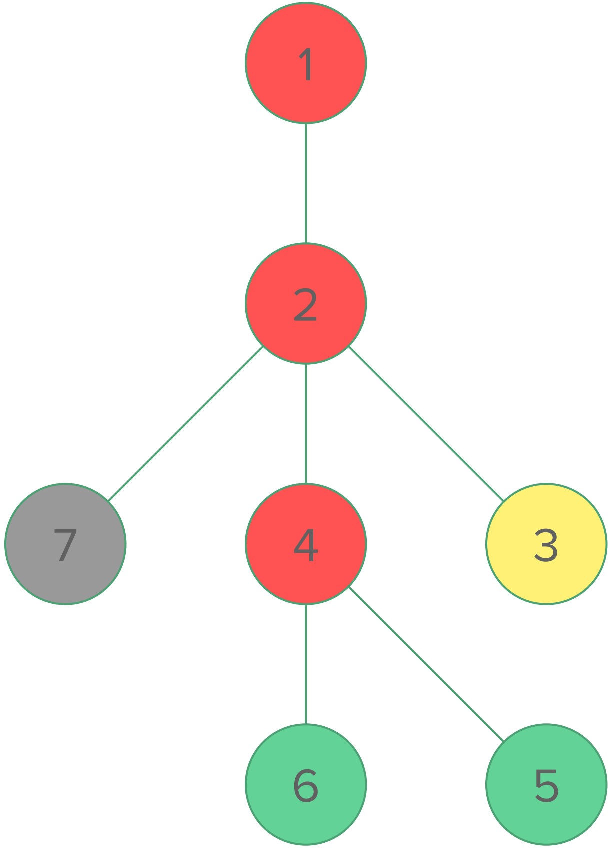


Breadth-first Walk

Queue: 6 5

Output: 1 2 7 4 3

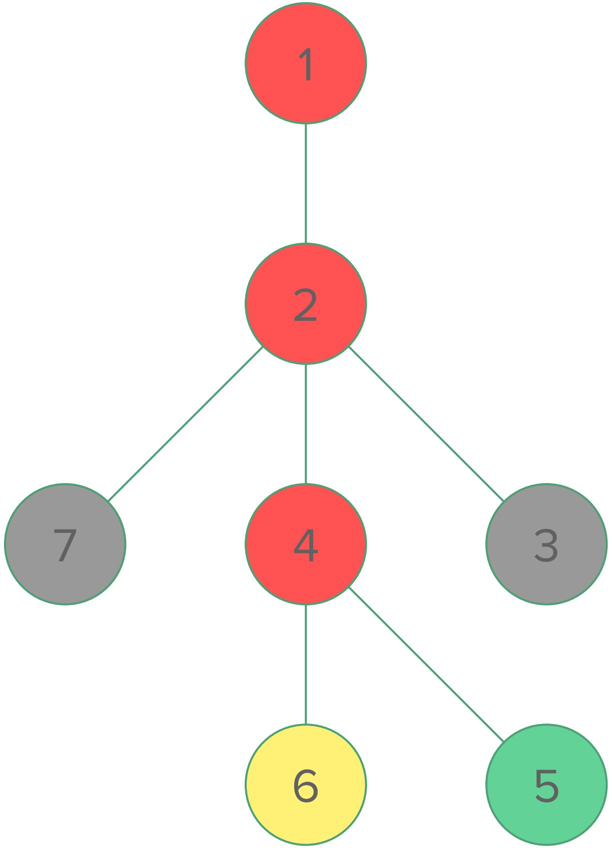
dequeue and print



Breadth-first Walk

Queue	5
Output	1 2 7 4 3 6

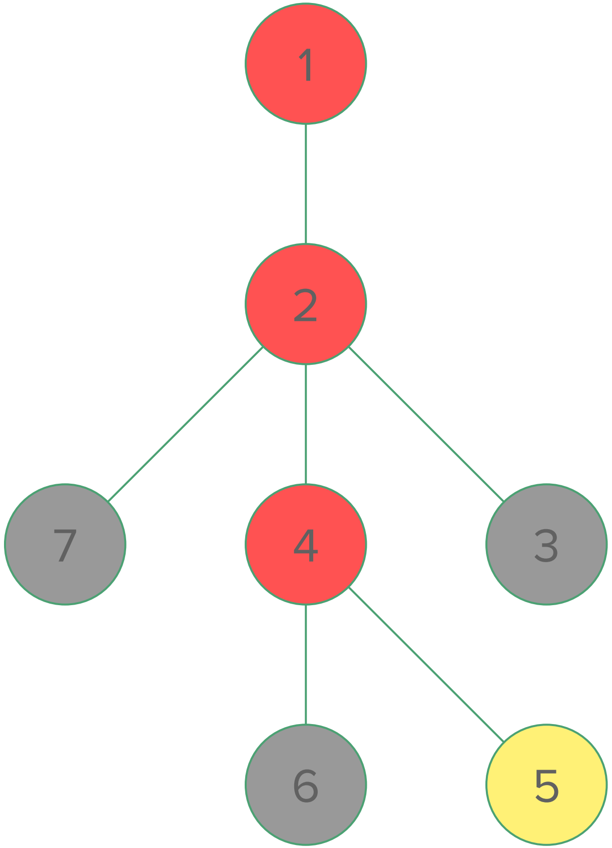
dequeue and print



Breadth-first Walk

Queue	empty
Output	1 2 7 4 3 6 5

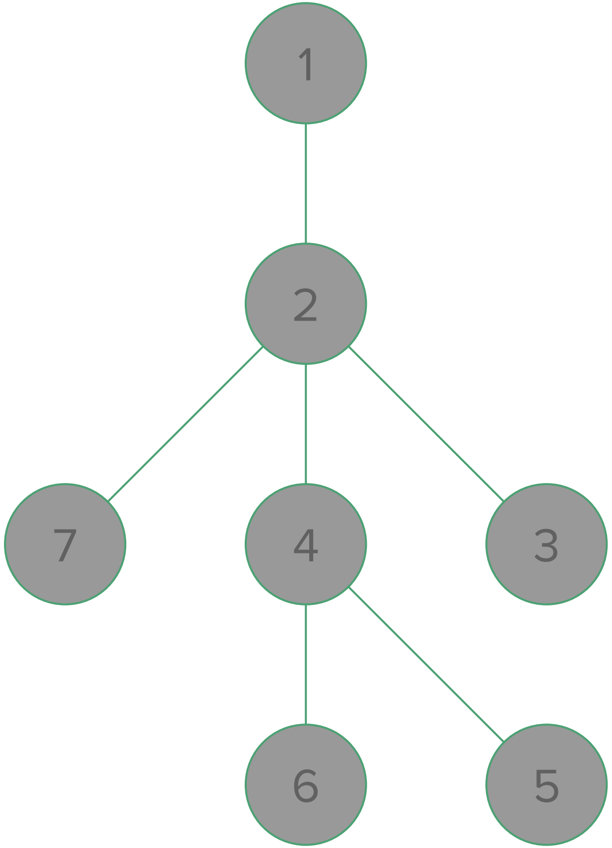
dequeue and print



Breadth-first Walk

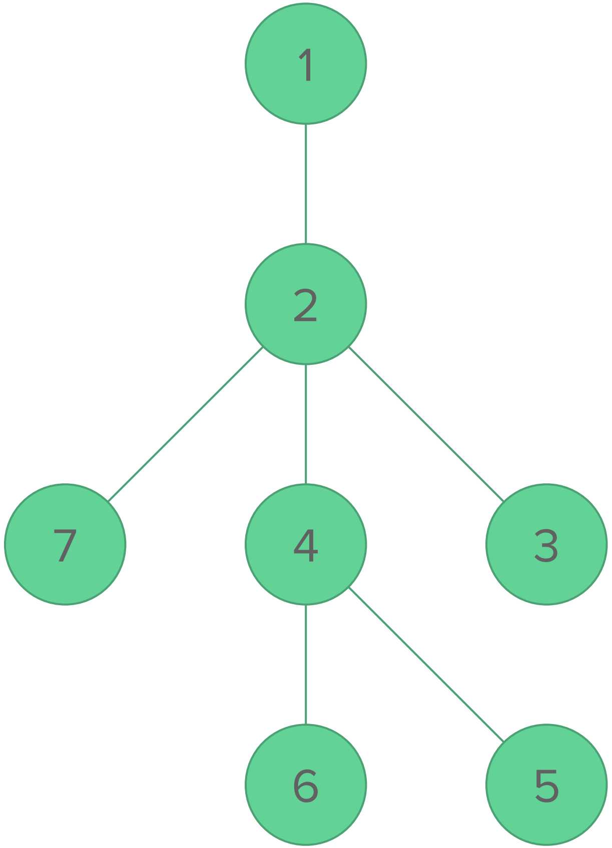
Queue	empty
Output	1 2 7 4 3 6 5

-



Breadth-first Walk

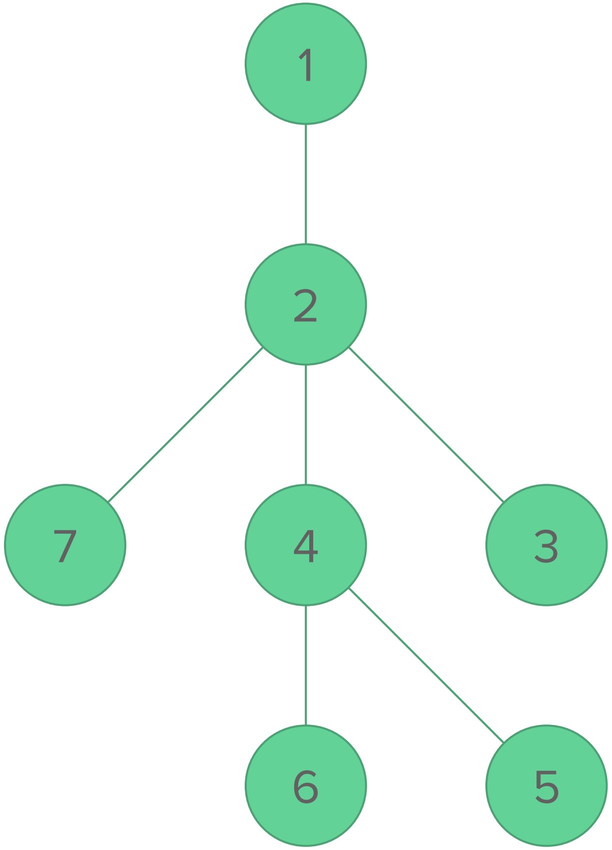
- Substituting a **queue** for a **stack** changes our **depth-first** walk into a **breadth-first** one
- It visited our children in a weird order, but only because we numbered our tree the way we did
- In practice, neither kind of walk is the "right" order by default; you'll need to pick the appropriate algorithm for your task



Depth-first walk Revisited

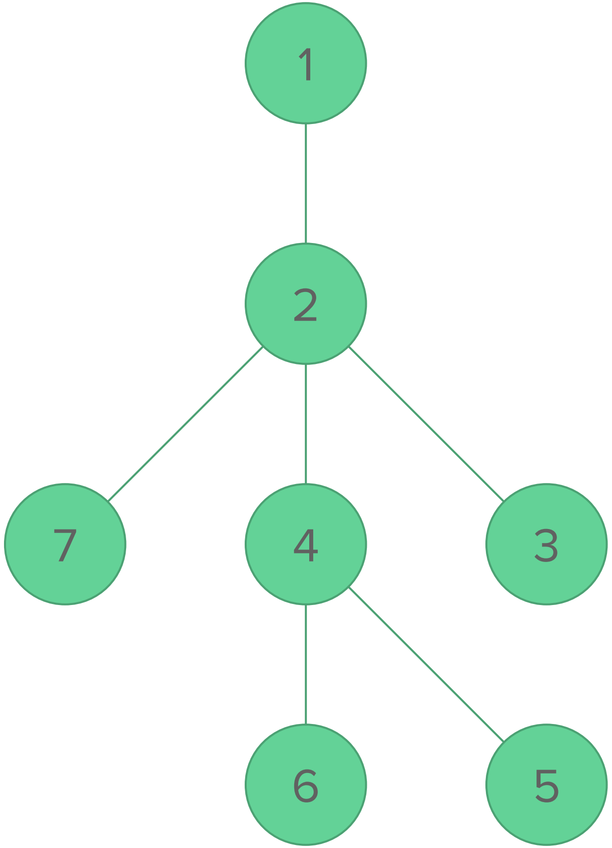
Our original algorithm for depth-first traversal looked like this:

```
DepthFirst(root) {
    s = empty Stack()
    s.Push(root)
    while(!s.Empty()) {
        n = s.Pop()
        print(n)
        foreach child c of n
            s.Push(n)
}
```



Depth-first walk Revisited

- Of course, we don't actually *need* a `Stack` data structure - we have a perfectly good stack already
- The ***execution stack***
  - Every procedure call pushes the **execution stack**
  - Every return pops it
- Why not use that stack instead?

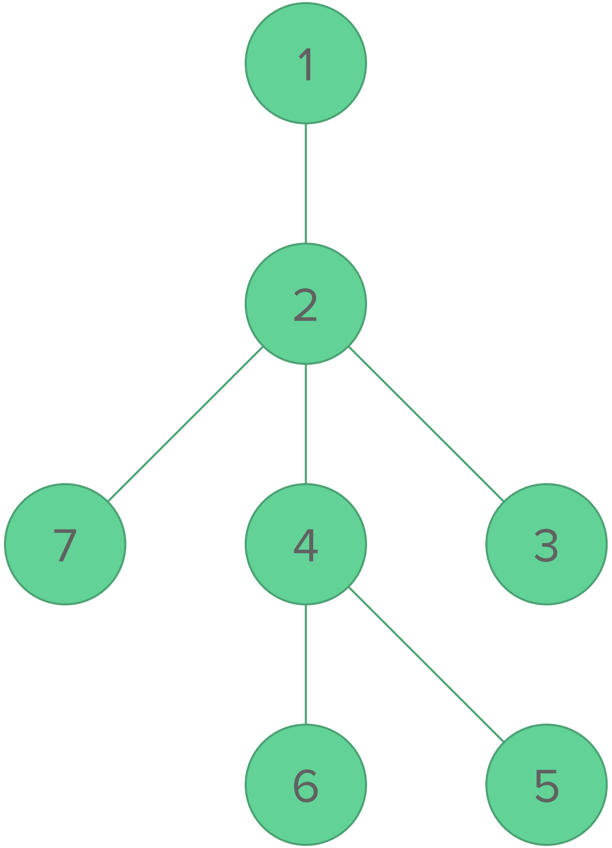


Depth-first walk Revisited

```
DepthFirst(node) {
    print(node)
    foreach child c of node
        DepthFirst(c)
}
```

Most of the time, it's easier (and clearer!) to write depth-first traversals as recursions.

(And you though to recur was to make things more complicated!)



(Slightly) Specialized Representations

Specialized Representations

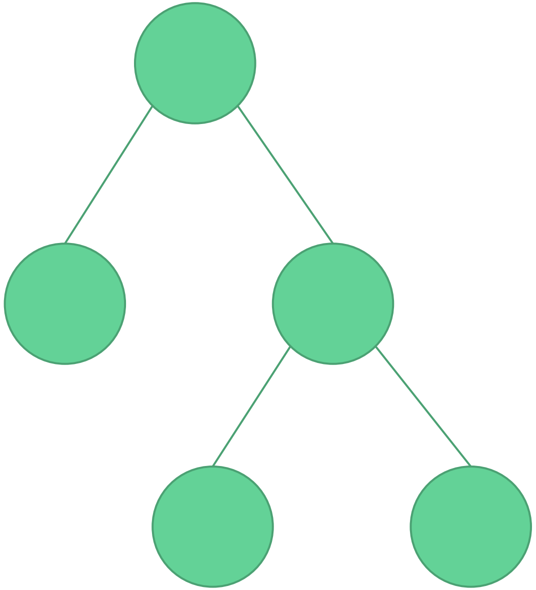
- Small trees with a ***fixed branching factor***
  - Put child pointers directly into the nodes
  - Usually used in trees with a branching factor of 2 or 3
- **Restricted information** in nodes
  - No parent pointer (e.g. cons pairs in Lisp and Scheme)
  - No child pointers (e.g. disjoint sets)
- **Heaps**
  - V important
  - We’ll talk about these later

- A tree with a fixed branching factor of ***n*** is referred to as an ***n-ary tree***
  - binary tree, ternary tree, 4-ary tree, etc.

- Food for thought: What is the name of a 1-ary (unary) tree where nodes have no access to their parents?

Binary Trees

- A **fixed branching factor tree** with a branching factor of **2**
- Every node has **at most 2** children
  - Referred to as ***left child*** and ***right child***
- Honestly, probably the most common case of trees
  - You will see this everywhere

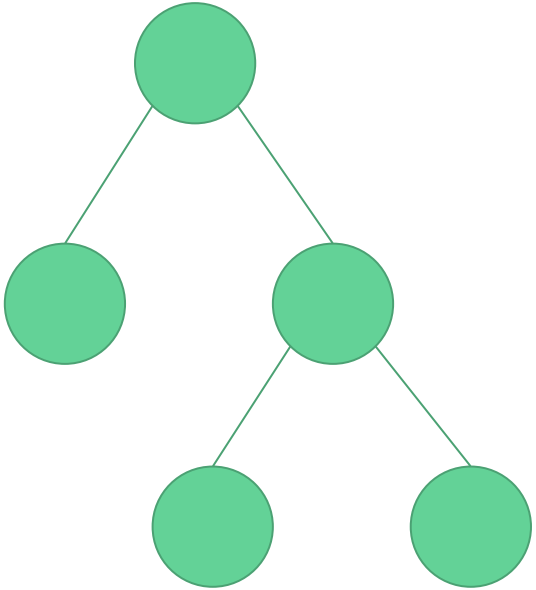


Binary Trees - Continued

```
class BinaryTree {
    BinaryTree _parent;
    BinaryTree _leftChild;
    BinaryTree _rightChild;
}
```

Don’t even bother with an array or linked list

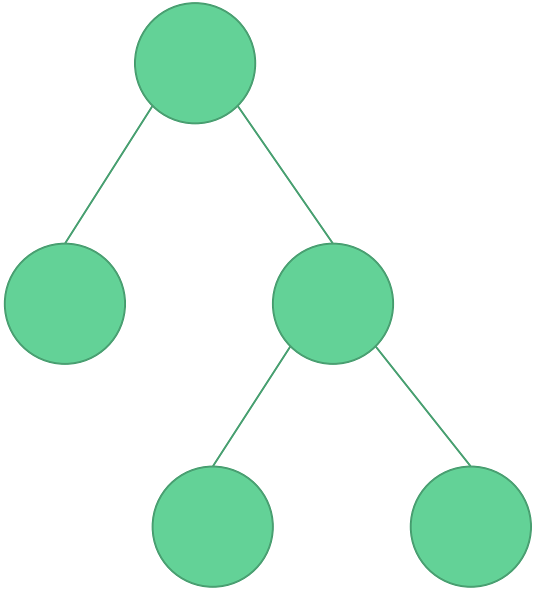
- Just put the pointers in the node for both children
- A null pointer indicates no child (or no parent, in the case of the root)



Why Are Binary Trees So Popular?

Why are binary trees *so popular*?

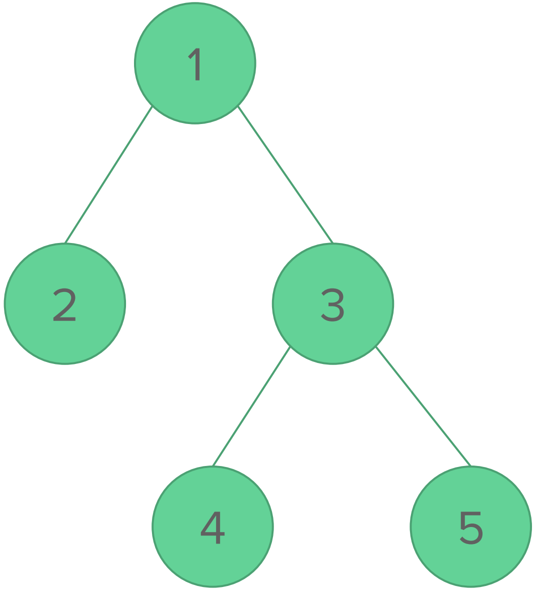
- **Left child, right sibling** representation is a binary tree
  - Saw these back in Act I
- Lisp and Scheme cons pairs (lists) are binary trees
- Decision trees (seen in classes on machine learning) are binary trees
- Binary search trees are a whole lecture in this class
  - They're also popular for interviews



Depth-first Traversal of Binary Trees

Suppose you cared to write (recursive, of course) **depth-first** traversal of a binary tree, how would you do it?

- There are actually three (equally valid) answers to this question
- We call them ***preorder***, ***inorder***, and ***postorder*** traversals of the tree



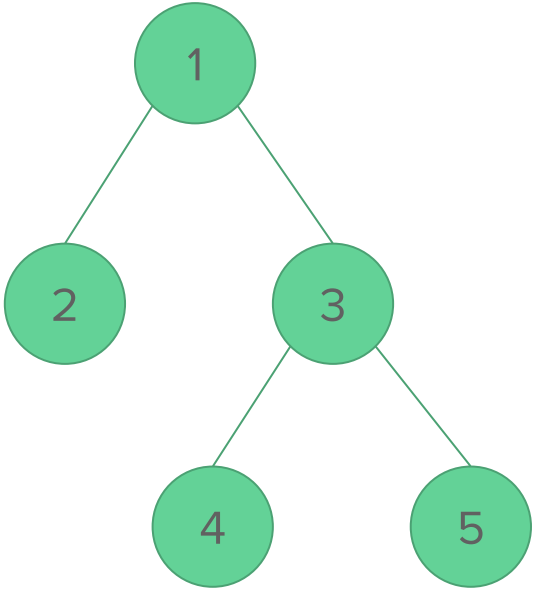
Depth-first Traversal of Binary Trees

Preorder traversal of the tree:

```
Preorder(node) {
    print(node)
    Preordor(node.Left)
    Preorder(node.Right)
}
```

Output:

1, 2, 3, 4, 5



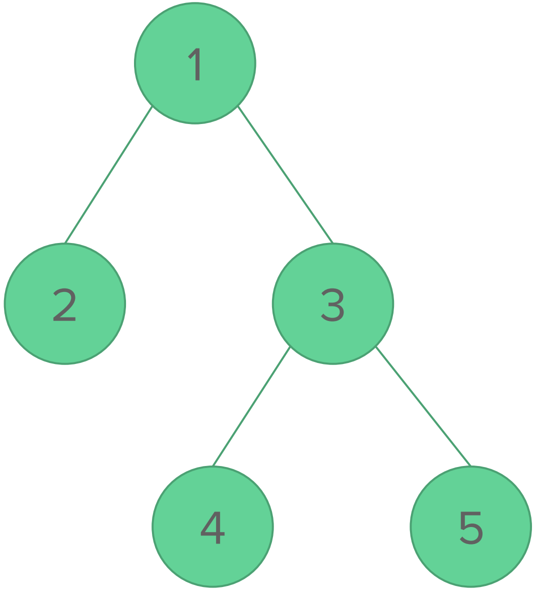
Depth-first Traversal of Binary Trees

Inorder traversal of the tree:

```
Inorder (node) {
    Inorder(node.Left)
    print(node)
    Inorder(node.Right)
}
```

Output:

2, 1, 4, 3, 5



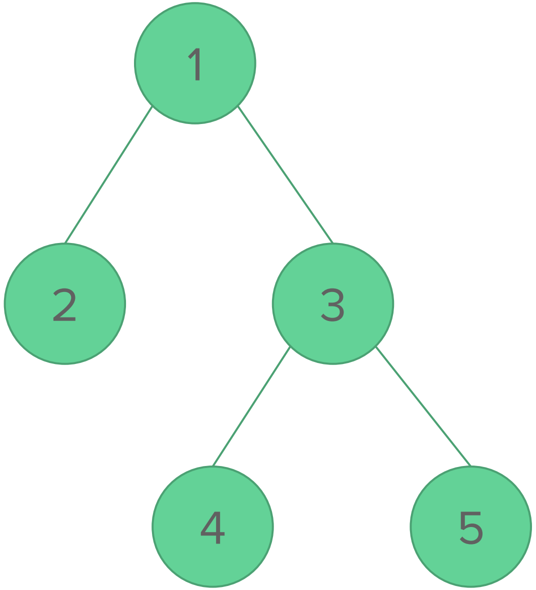
Depth-first Traversal of Binary Trees

Postorder traversal of the tree:

```
Postorder(node) {
    Postordor(node.Left)
    Postorder(node.Right)
    print(node)
}
```

Output:

2, 4, 5, 3, 1



Reading

- CLRS Chapter 10, Section 4 “Representing Rooted Trees”