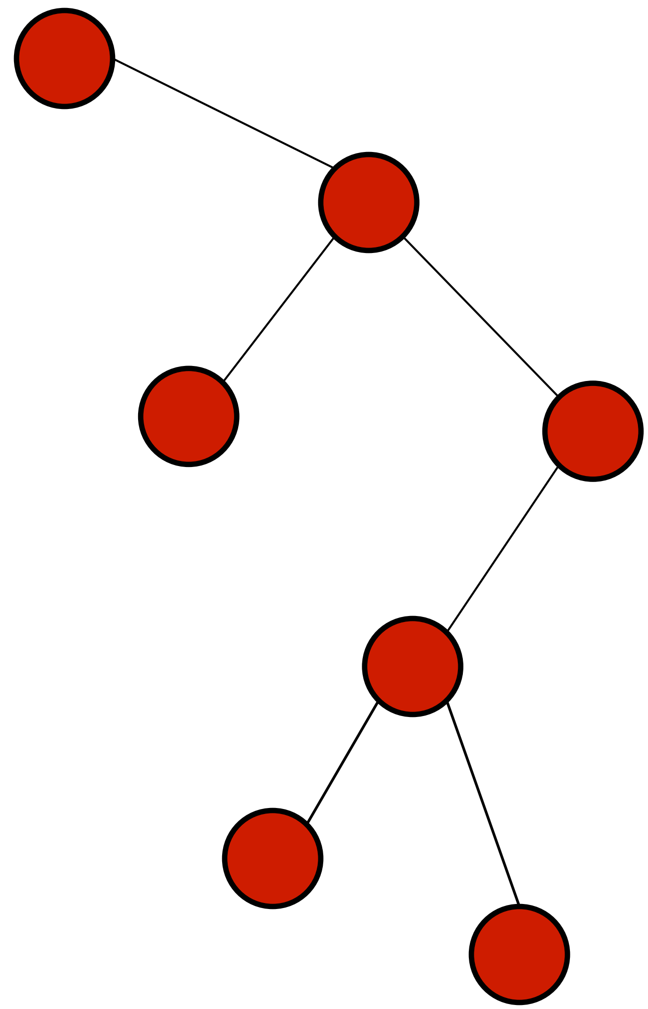
# Lecture 5
# Tree walks
# and tree representations

EECS-214

# Remember trees?

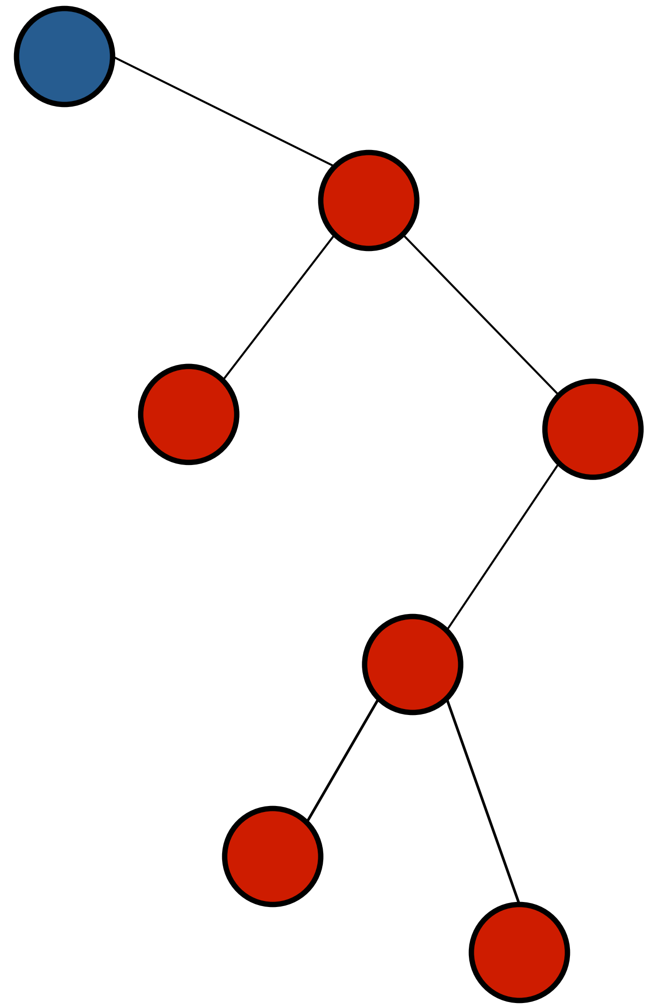
# Trees

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



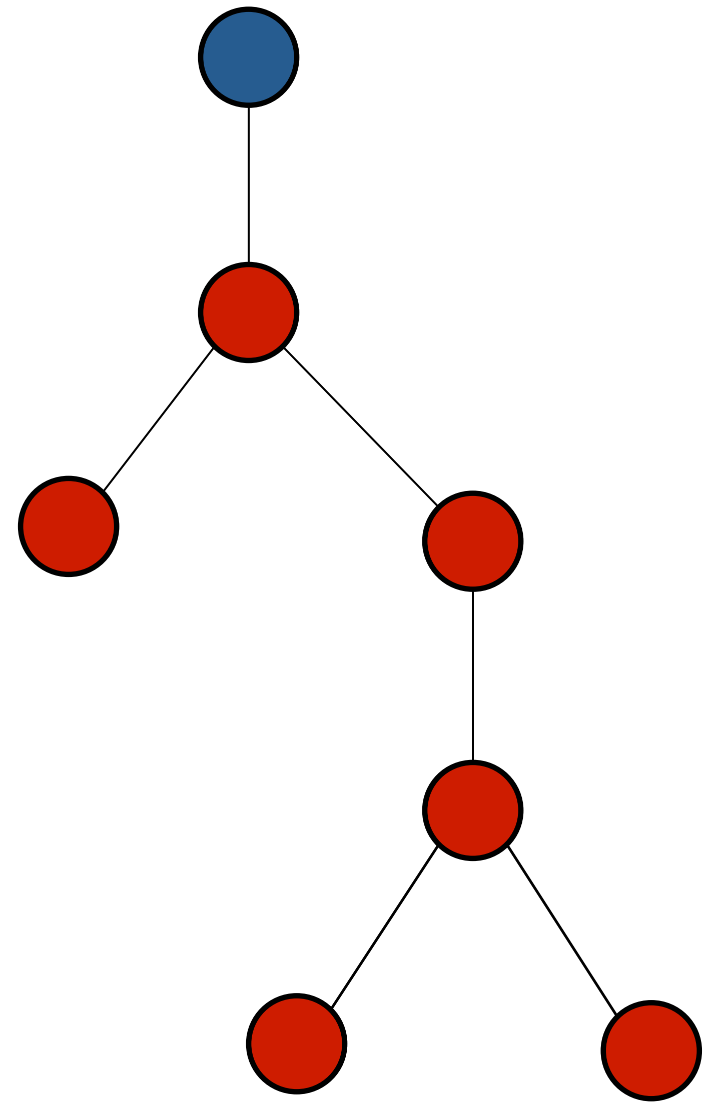
# Trees

- In computer science, we usually think of trees as having a distinguished node call the **root**



# Rooted trees

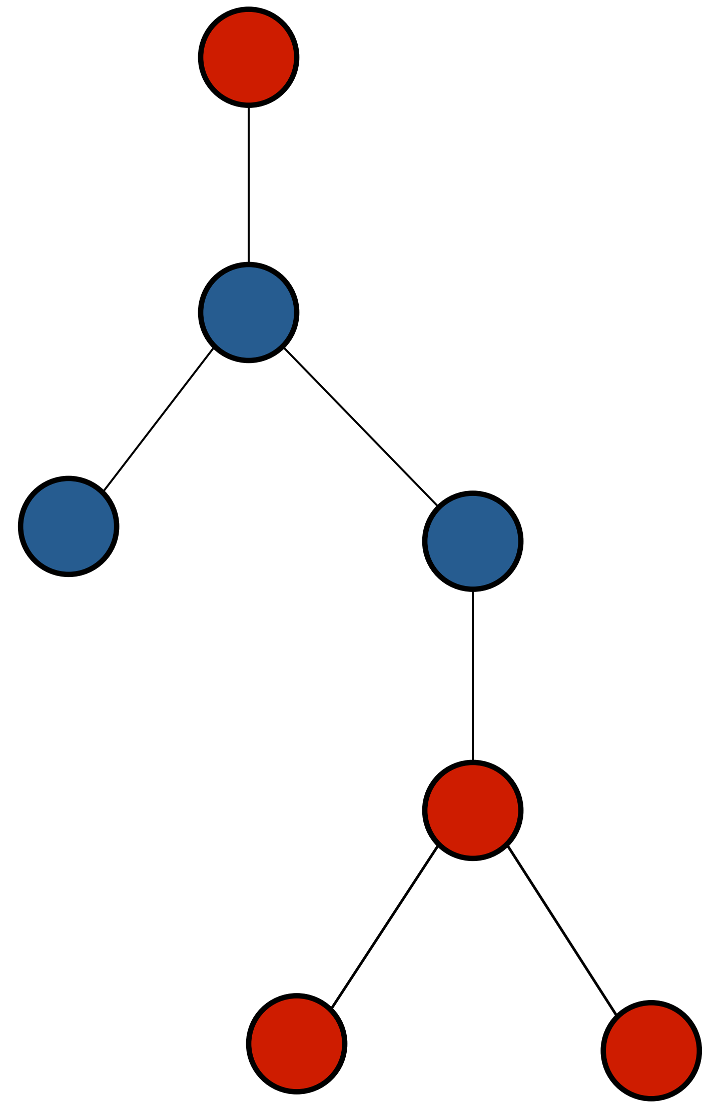
- In computer science, we usually think of trees as having a distinguished vertex called the **root**
- And we draw it
  - With the root at the top
  - And other vertices arranged by their distance from the root



# Children

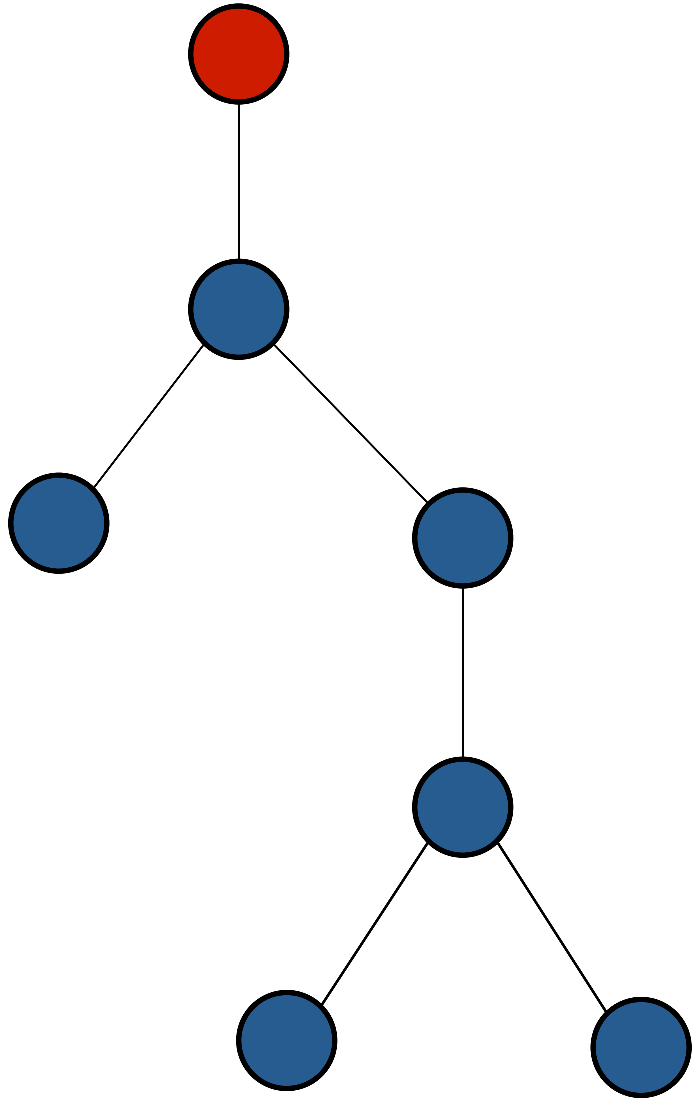
- The nodes adjacent to a node, but at the next lower depth are called its **children**

**Note:** node and vertex are synonyms



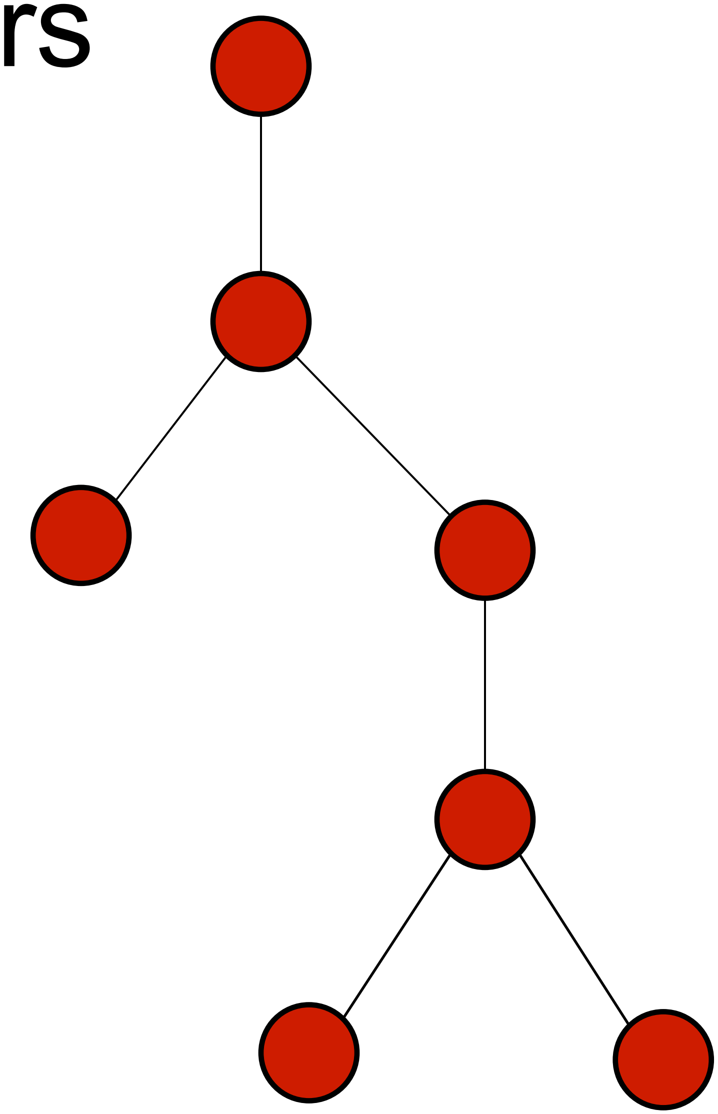
# Children

- The nodes adjacent to a node, but at the next lower depth are called its **children**
- And the nodes that are its children, children's children, etc., are called its **descendants**



# Parents and ancestors

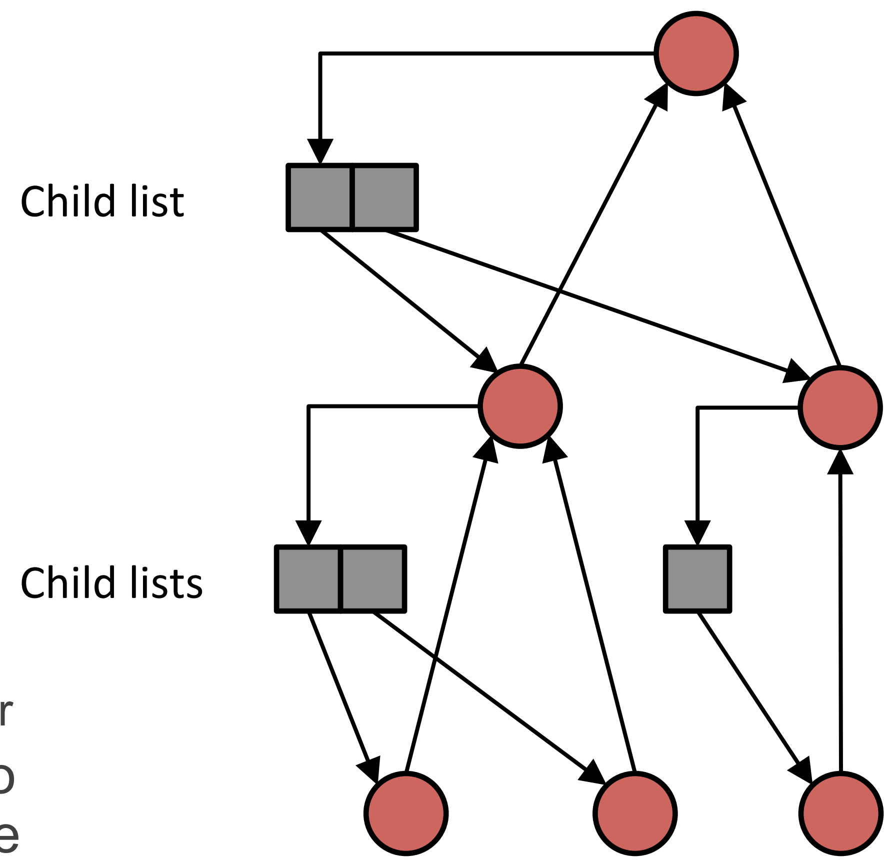
- The nodes above a node in the tree are its **ancestors**
- The node immediately above a node in the tree is its **parent**
  - All nodes have parents except the root
- Nodes with the same parent are called **siblings**



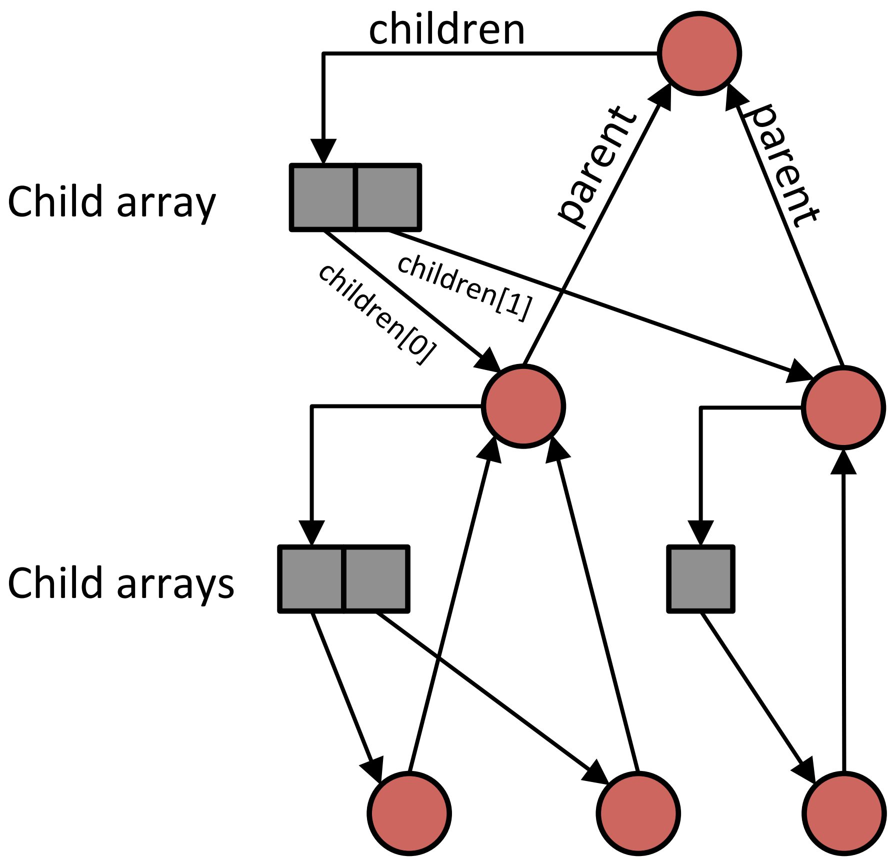
# Data structures for trees

# General representation of trees

- Each tree node is an object
  - (Red circles)
- Each node object contains
  - Parent
    - (Upward arrows)
  - List of children
    - (Grey boxes)
    - linked list, array, whatever
  - Anything else you want to remember about the node

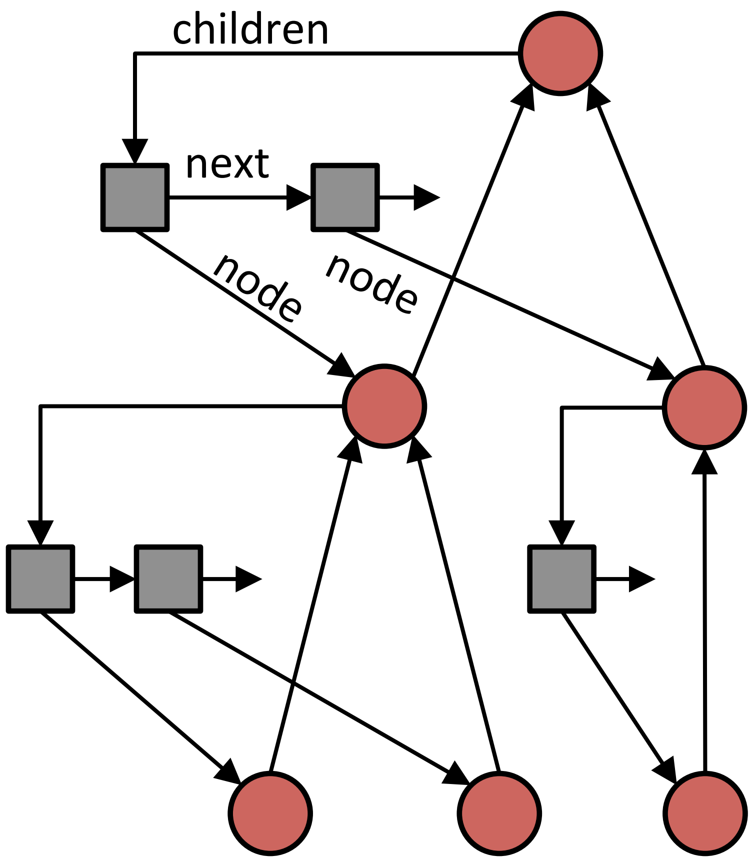


# Using an array



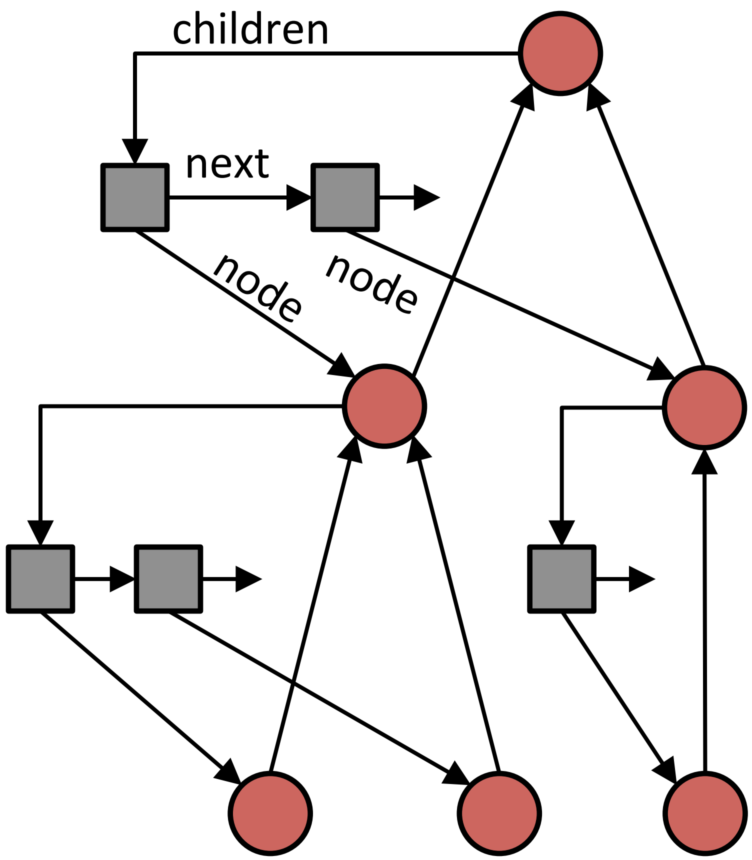

```
class TreeNode {
    TreeNode parent;
    TreeNode[] children;
    … other data …
}
```

# Using a linked list



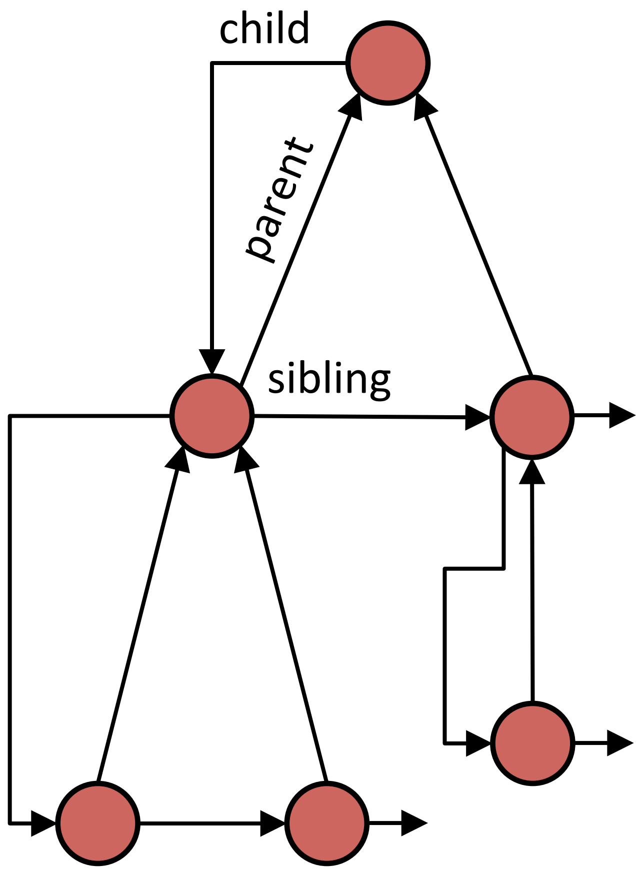

```
class TreeNode {
    TreeNode parent;
    TreeNodeList children;
    … other data …
}

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

# Using a linked list



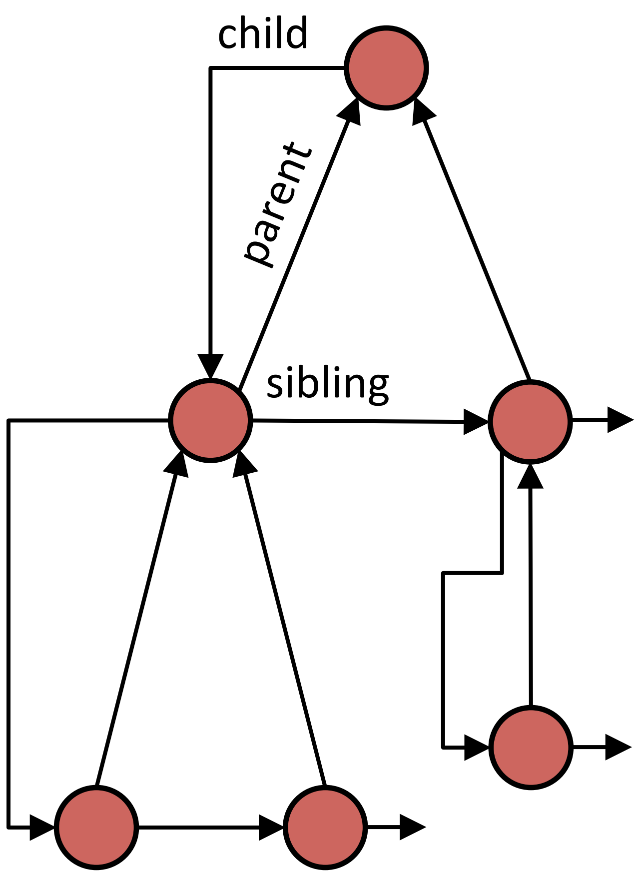

- Notice that we have exactly one linked list cell per node
  - (except the root)
- So we can just move the next pointer into the node itself

# Moving the link into the node




- Notice that we have exactly one linked list cell per node
  - (except the root)
- So we can just move the next pointer into the node itself
- And remove the linked list cells

# Moving the link into the node

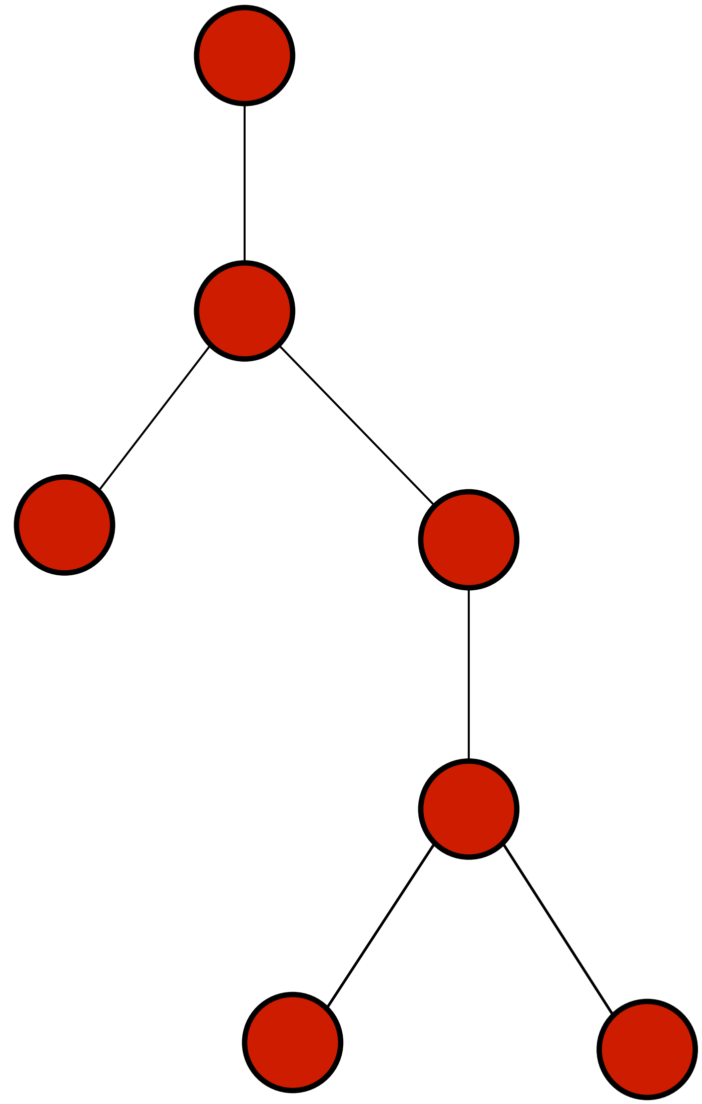



class TreeNode {
TreeNode parent;
TreeNode firstChild;
TreeNode nextSibling;
… other data …
}

The CLR book calls this the left child/right sibling representation

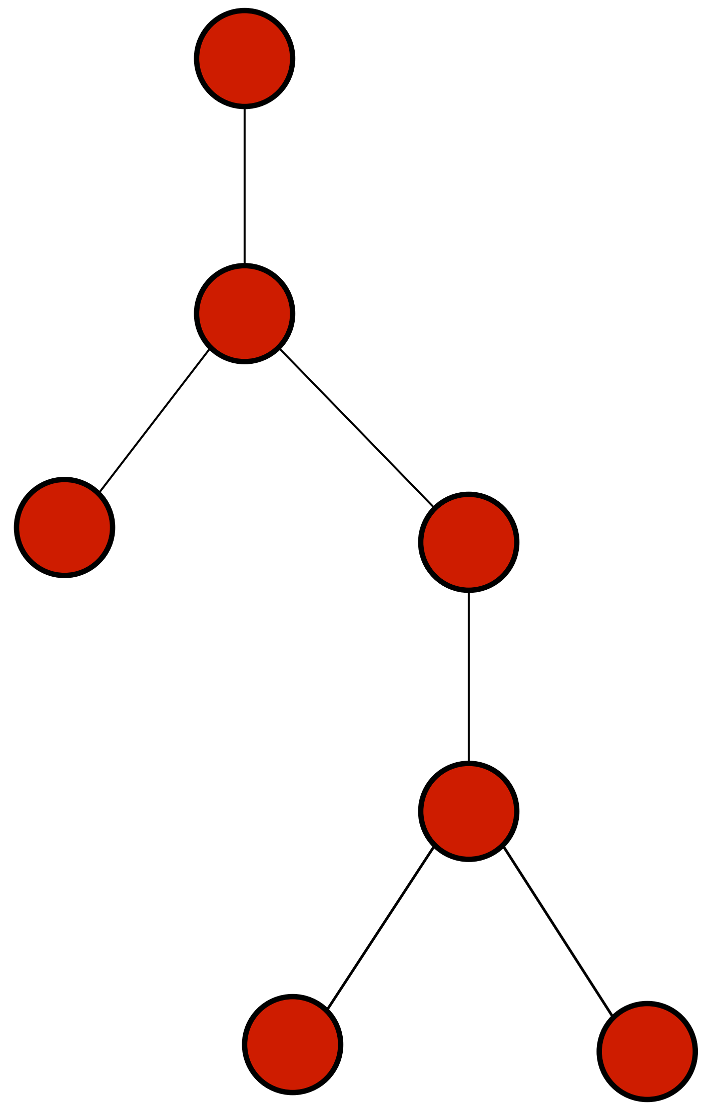
# Tree walks

- We generally don't talk about **iterating** through the nodes in a tree
- We talk about **walking** through them or **traversing** them
  - That's partly because tree walks are often written as recursions
  - And we think of iterations as being while and for loop, and recursions being something different



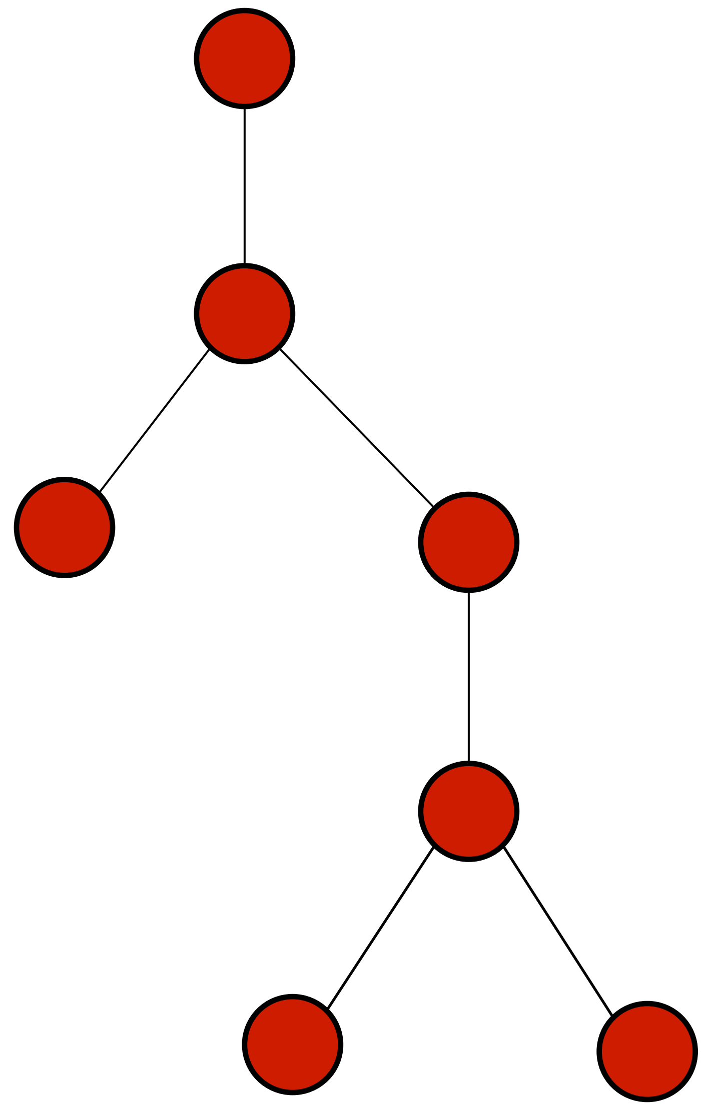
# Tree walks

- You can walk trees in different orders



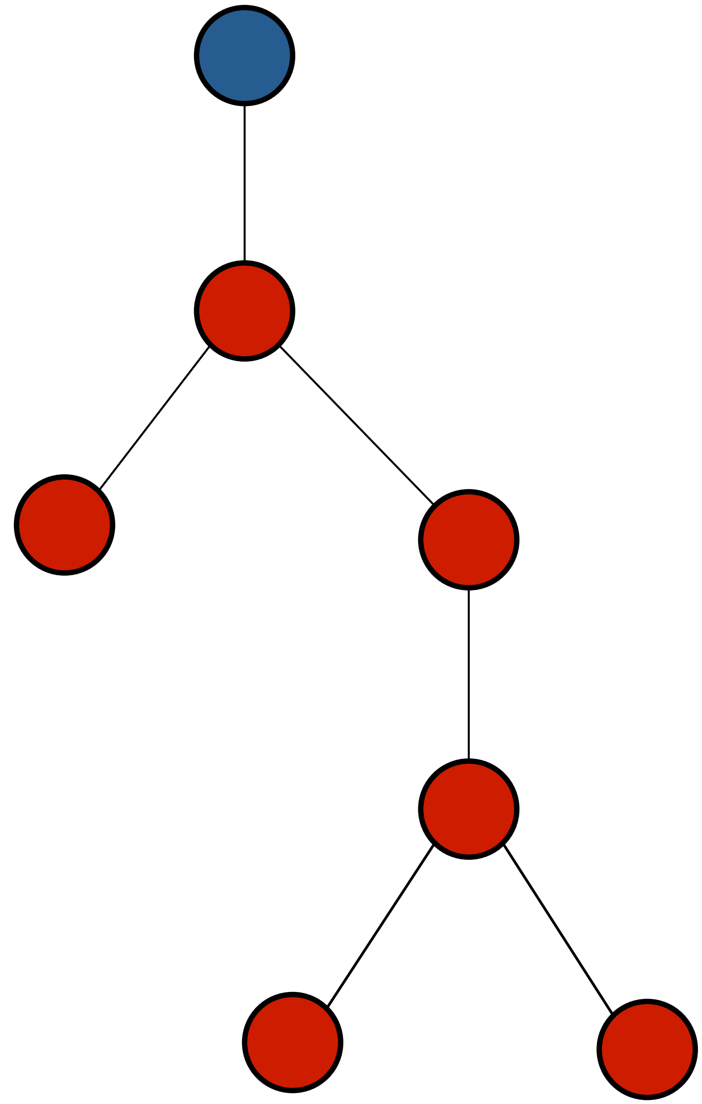
# Tree walks

- You can walk trees in different orders
- **Breadth-first** walks



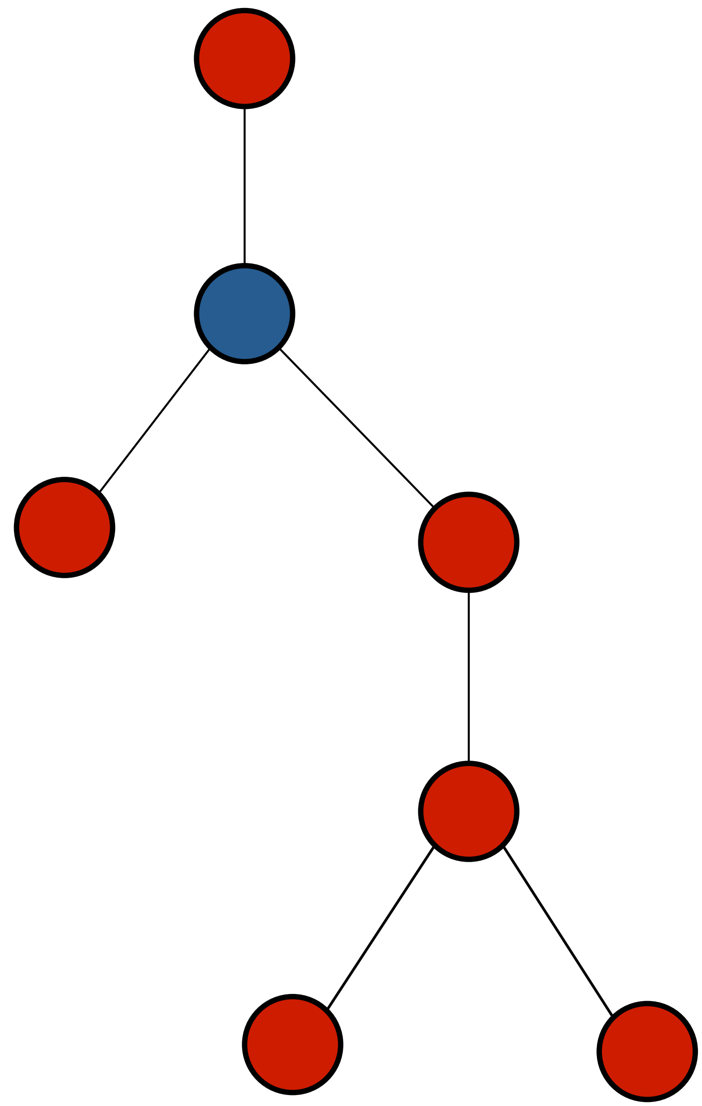
# Tree walks

- You can walk trees in different orders
- **Breadth-first** walks
  - Start with the root



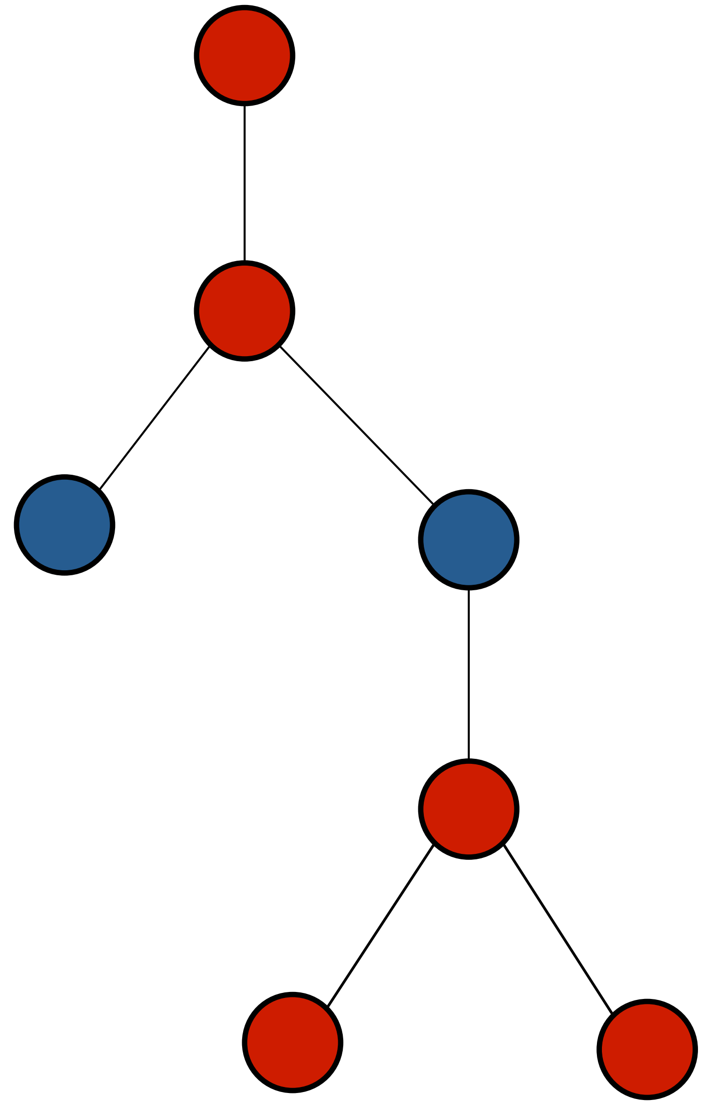
# Tree walks

- You can walk trees in different orders
- **Breadth-first** walks
  - Start with the root
  - Then do all the nodes at depth 1 (just one in this case)



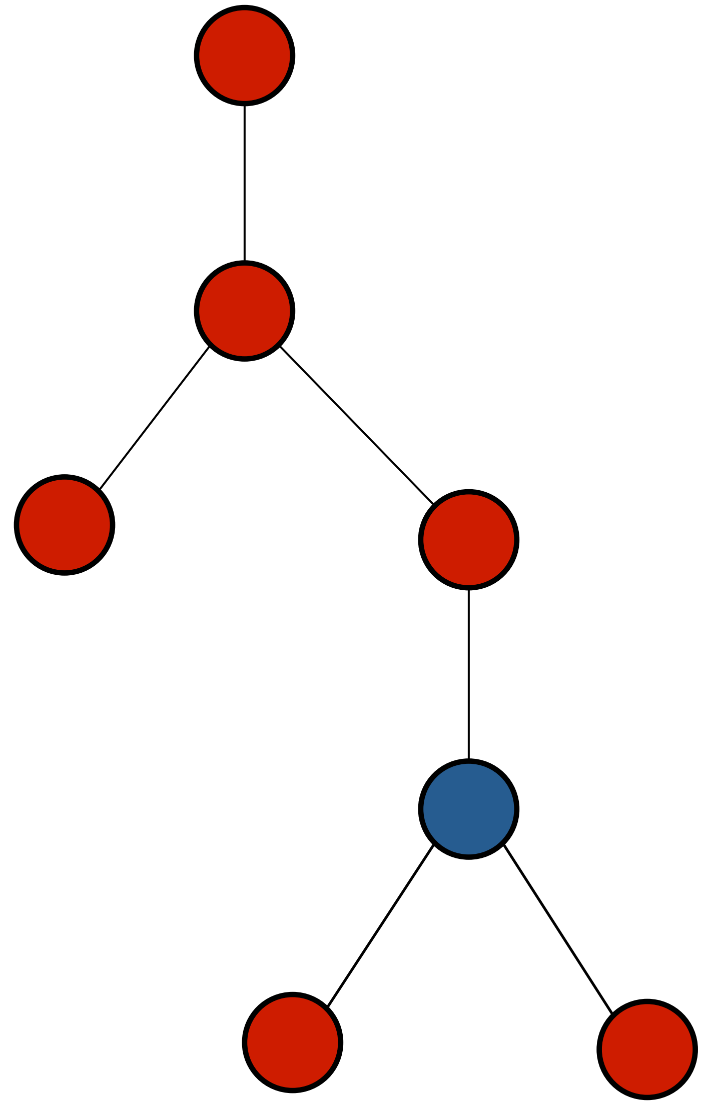
# Tree walks

- You can walk trees in different orders
- **Breadth-first** walks
  - Start with the root
  - Then do all the nodes at depth 1
  - Then depth 2



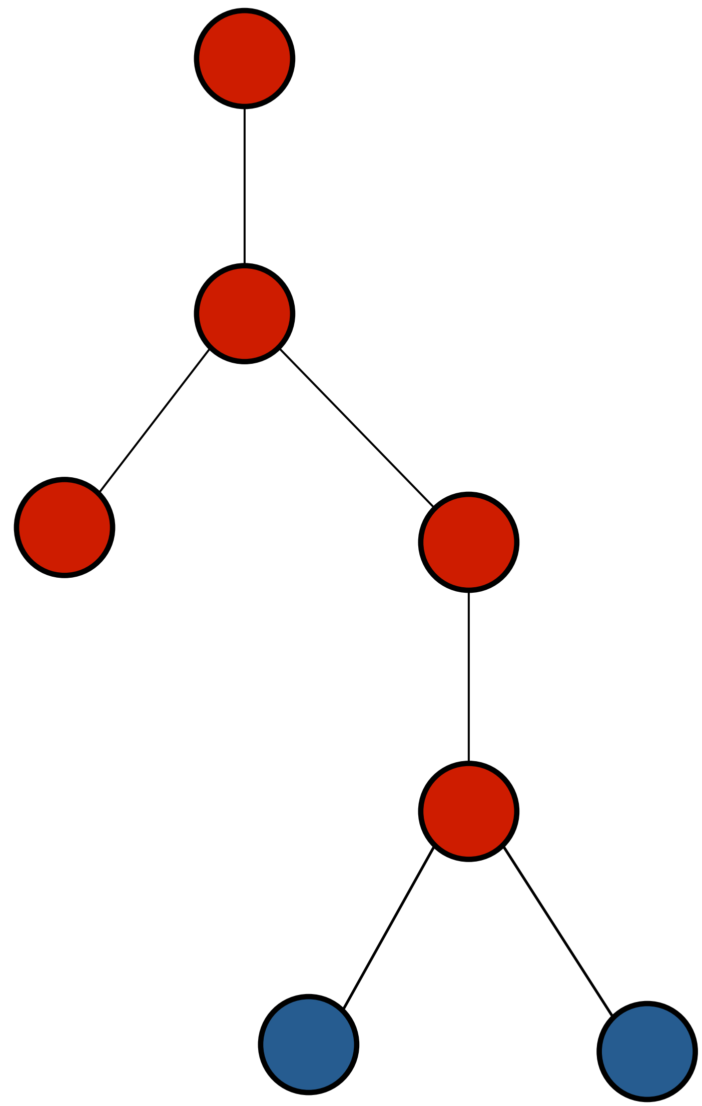
# Tree walks

- You can walk trees in different orders
- **Breadth-first** walks
  - Start with the root
  - Then do all the nodes at depth 1
  - Then depth 2
  - Then depth 3



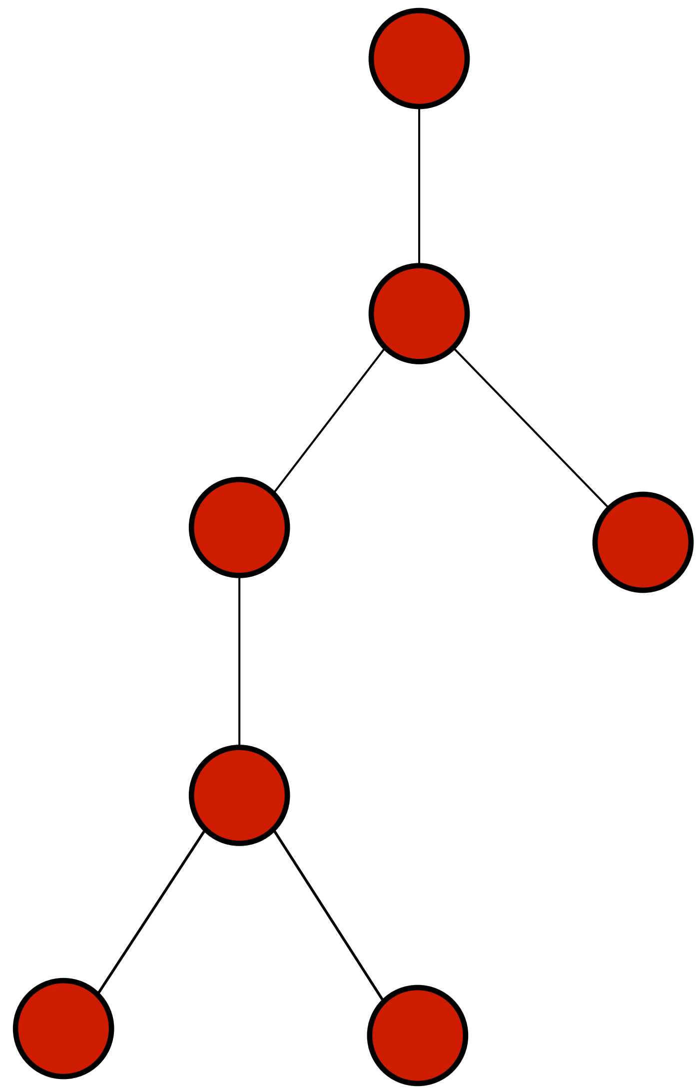
# Tree walks

- You can walk trees in different orders
- **Breadth-first** walks
  - Start with the root
  - Then do all the nodes at depth 1
  - Then depth 2
  - Then depth 3
  - And so on …



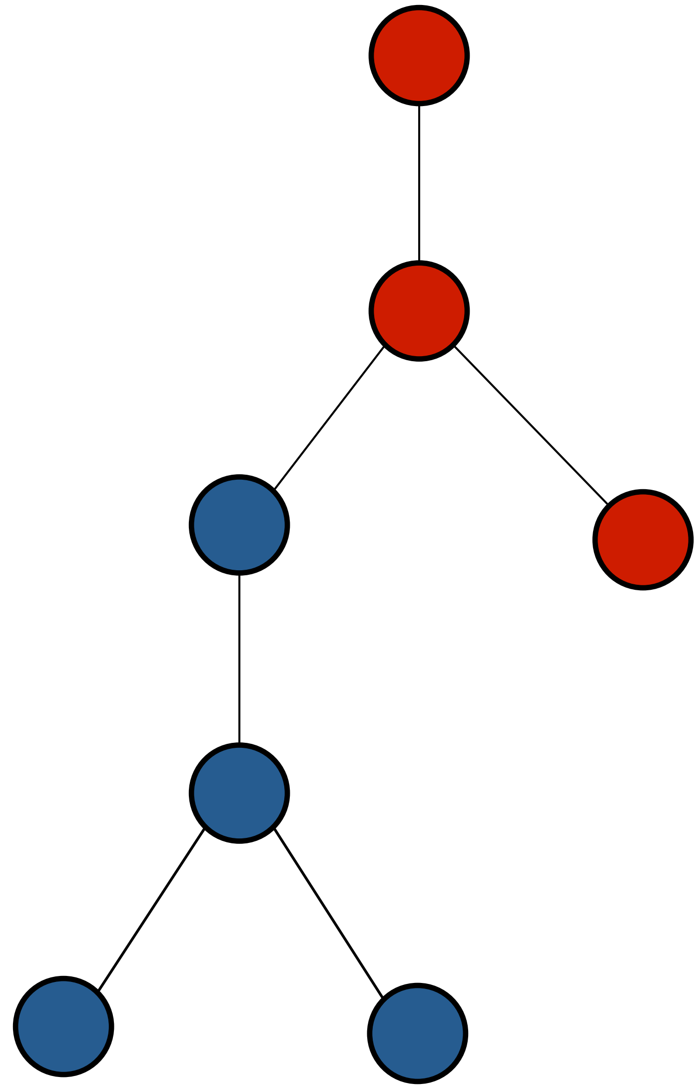
# Tree walks

- A **depth-first** walk



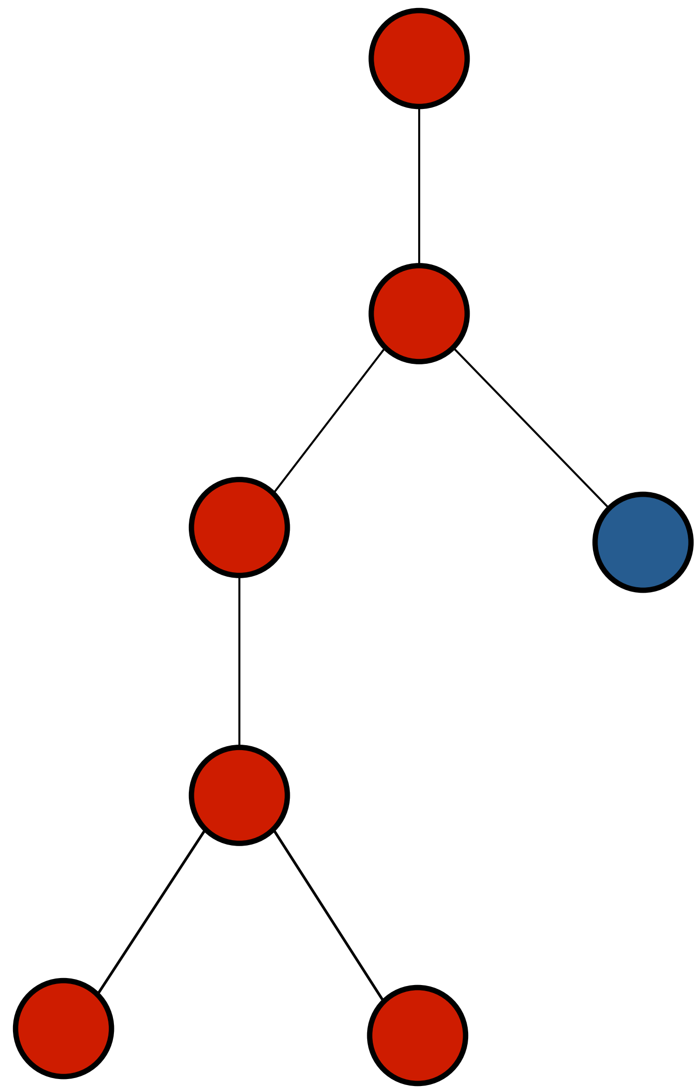
# Tree walks

- A **depth-first** walk
  - Goes through all the decedents of a node



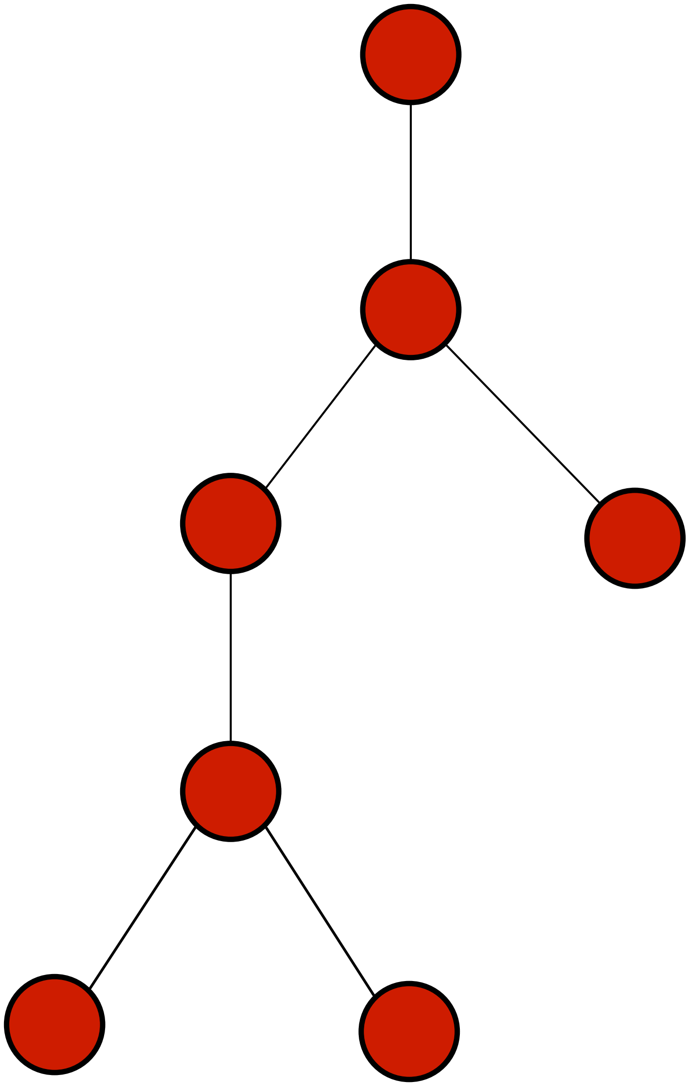
# Tree walks

- A **depth-first** walk
  - Goes through all the decedents of a node
  - Before moving to the node's sibling



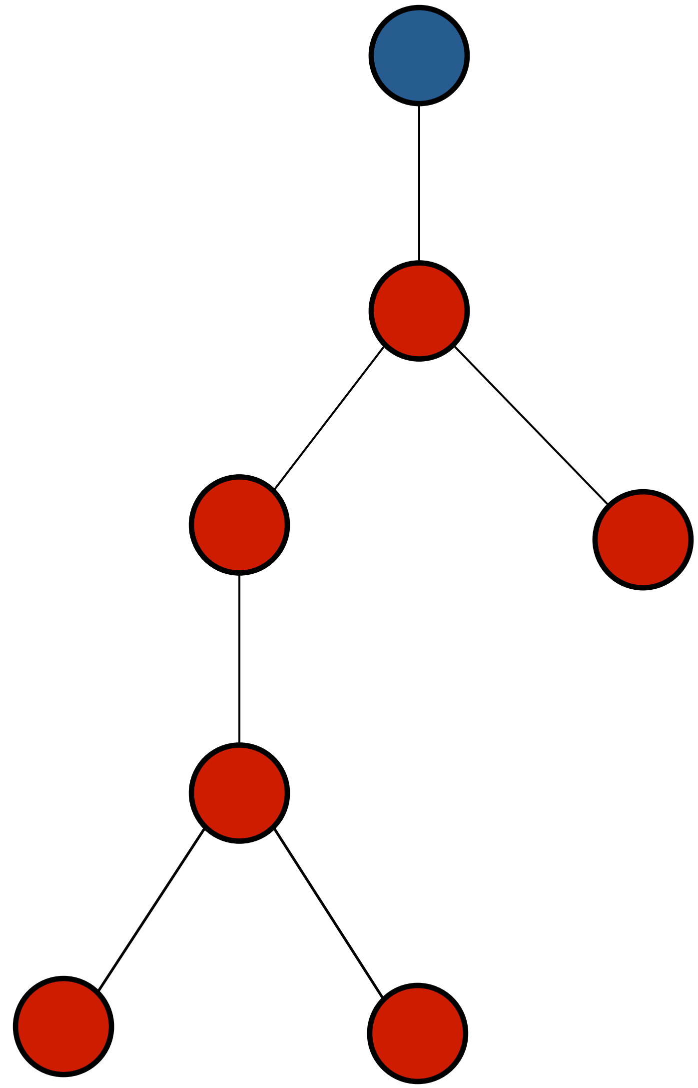
# Tree walks

- A **depth-first** walk



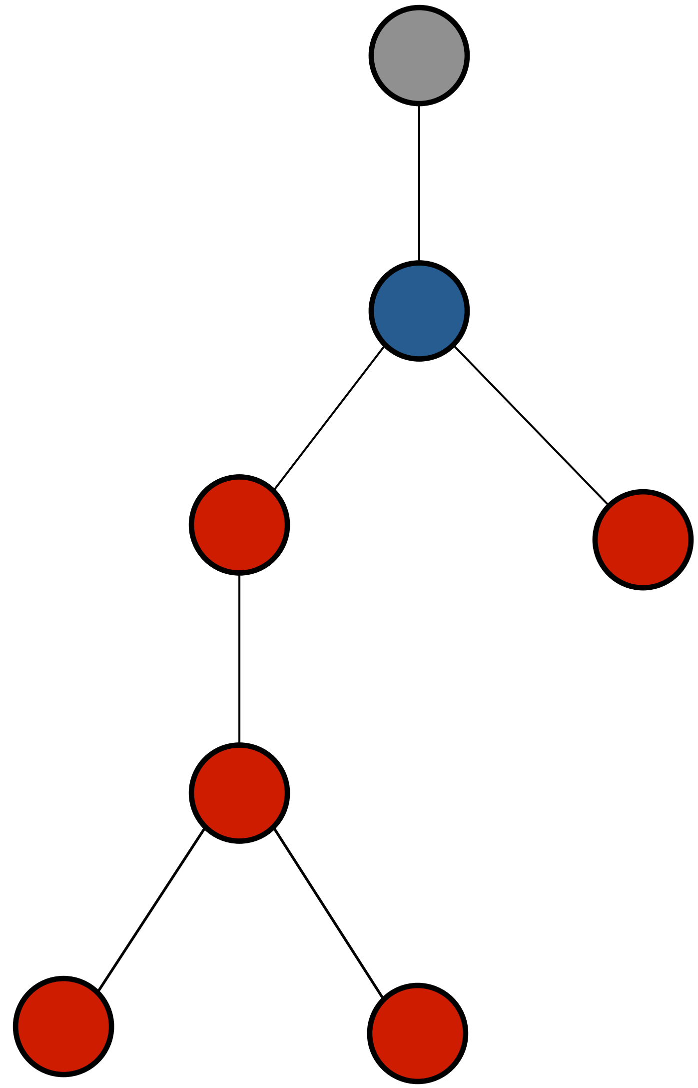
# Tree walks

- A **depth-first** walk
  - Starts with the root



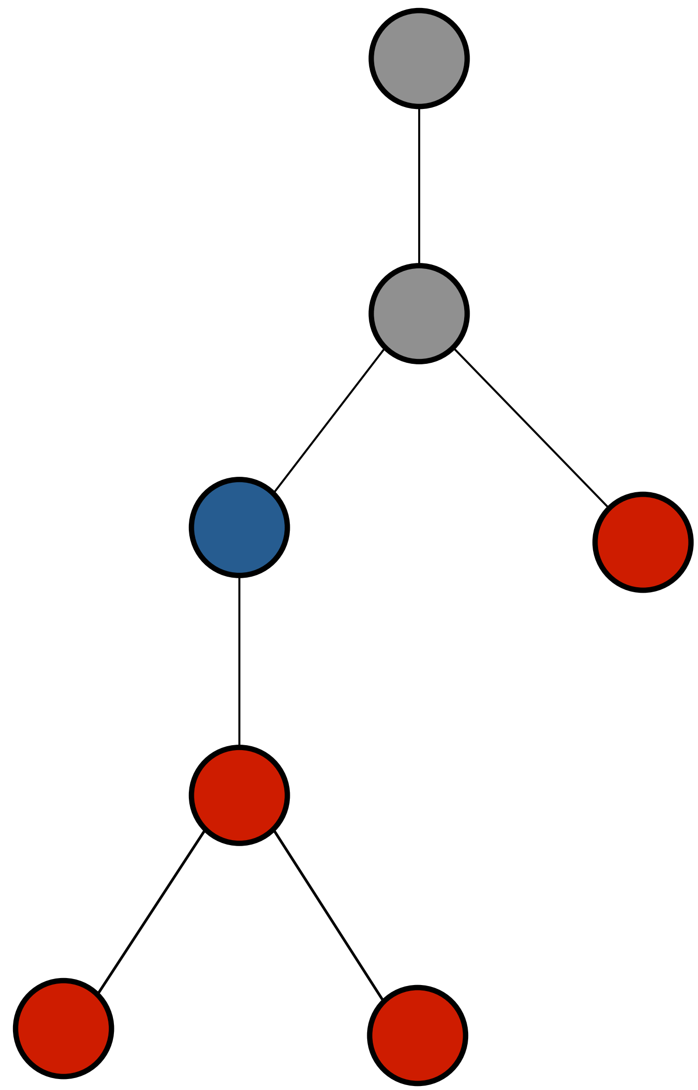
# Tree walks

- A **depth-first** walk
  - Starts with the root
  - Moves to its first child (in this case, the only child)



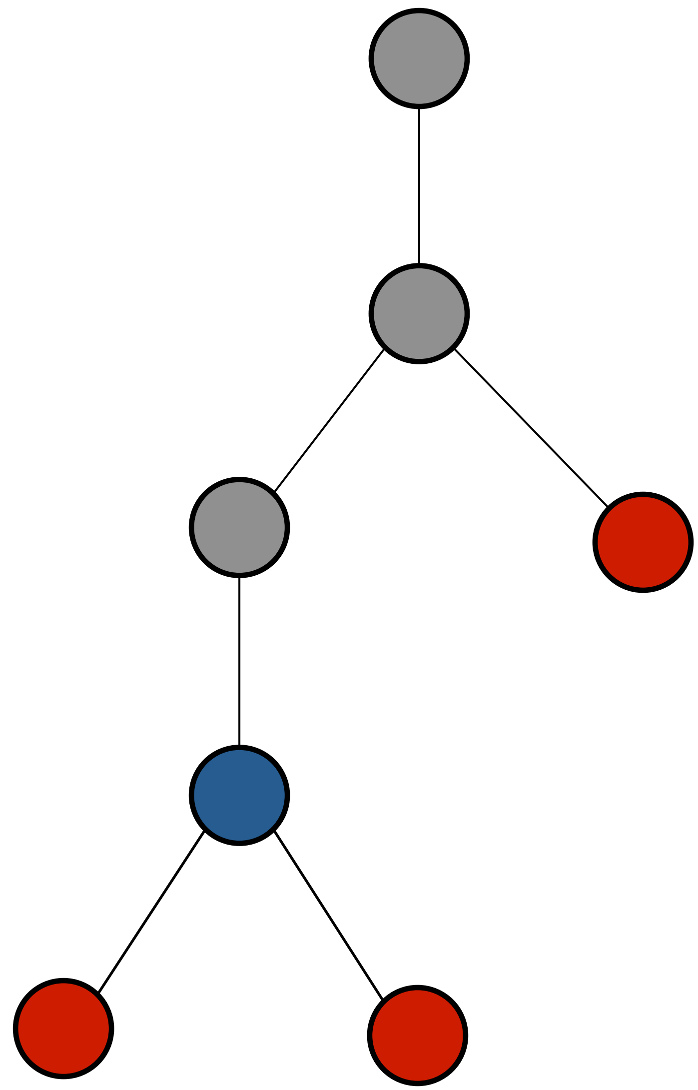
# Tree walks

- A **depth-first** walk
  - Starts with the root
  - Moves to its first child
  - Then its first child



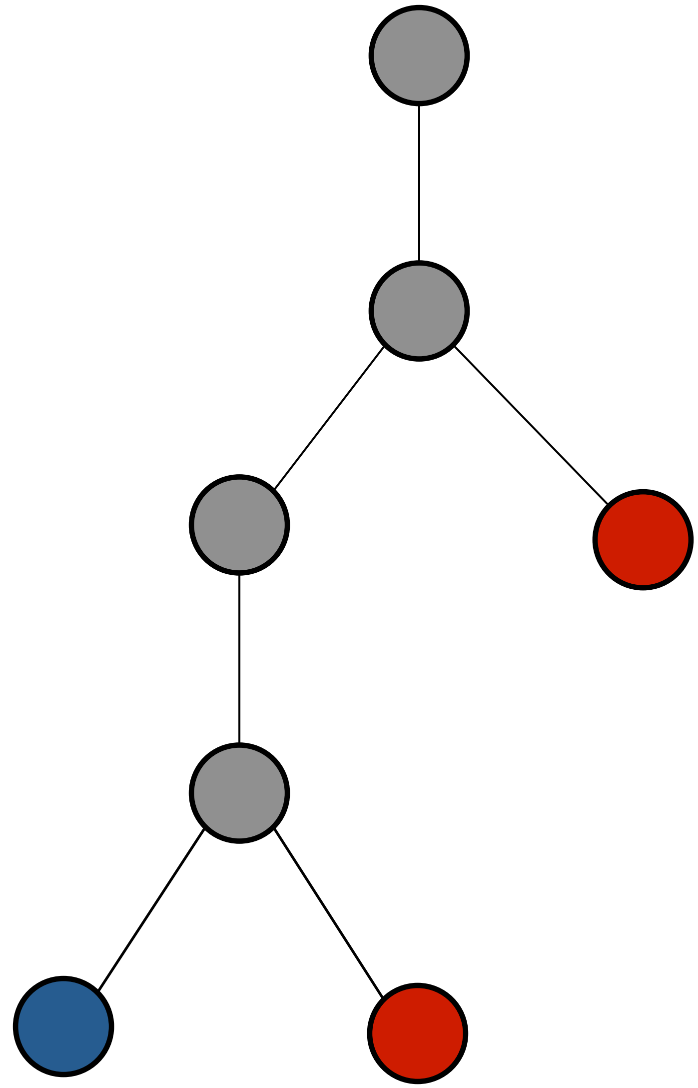
# Tree walks

- A **depth-first** walk
  - Starts with the root
  - Moves to its first child
  - Then its first child
  - And so on



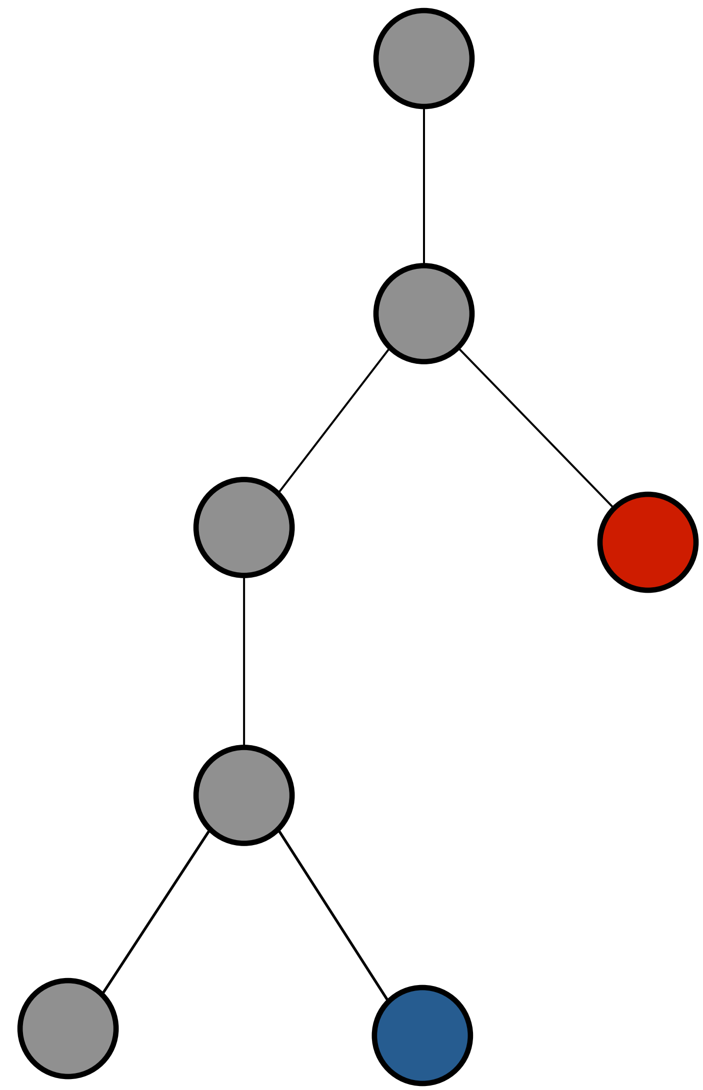
# Tree walks

- A **depth-first** walk
  - Starts with the root
  - Moves to its first child
  - Then its first child
  - And so on
  - Until we hit a leaf (a node with no children)



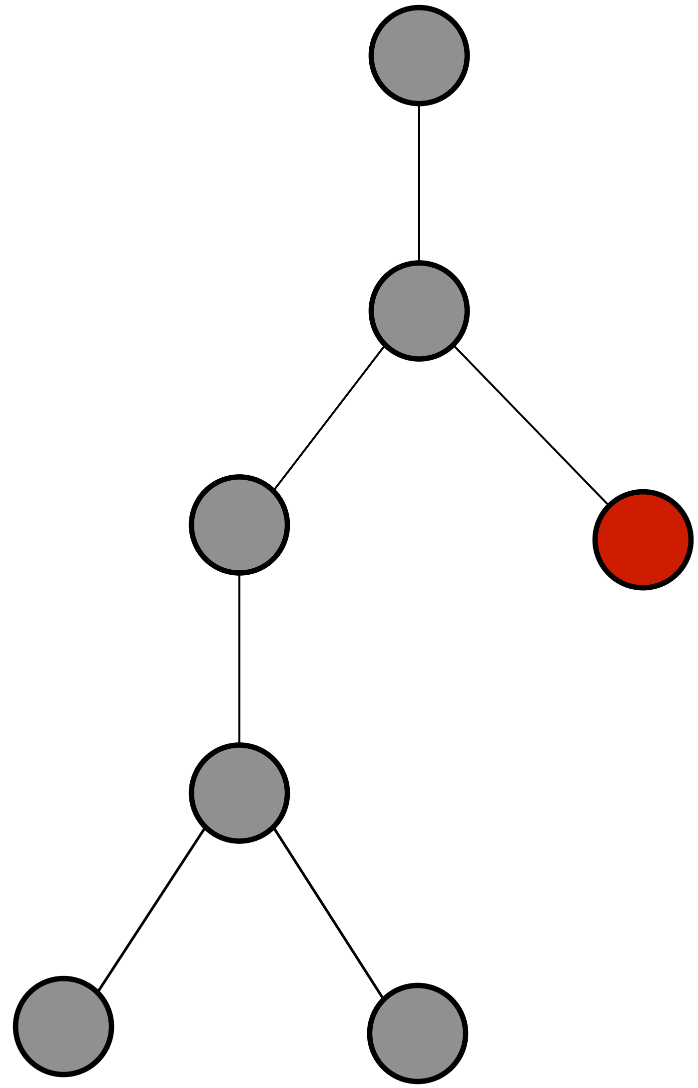
# Tree walks

- A **depth-first** walk
  - Starts with the root
  - Moves to its first child
  - Then its first child
  - And so on
  - Until we hit a leaf
  - Then moves to the leaf's sibling



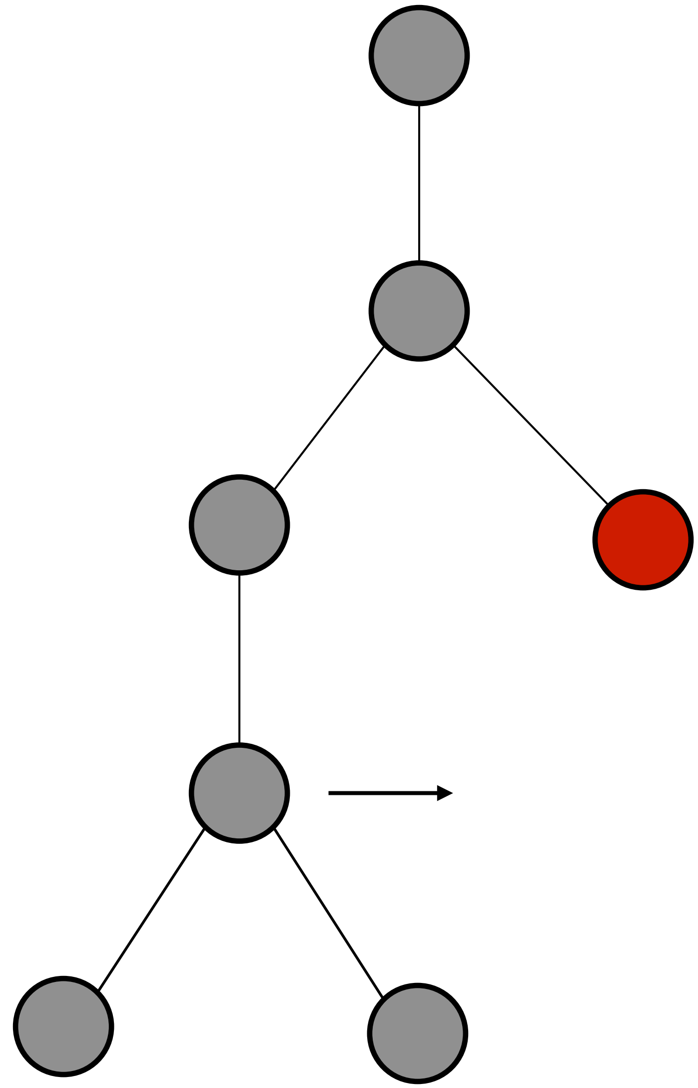
# Tree walks

- A **depth-first** walk
  - Starts with the root
  - Moves to its first child
  - Then its first child
  - And so on
  - Until we hit a leaf
  - Then moves to the leaf's sibling
  - Until we run out of siblings



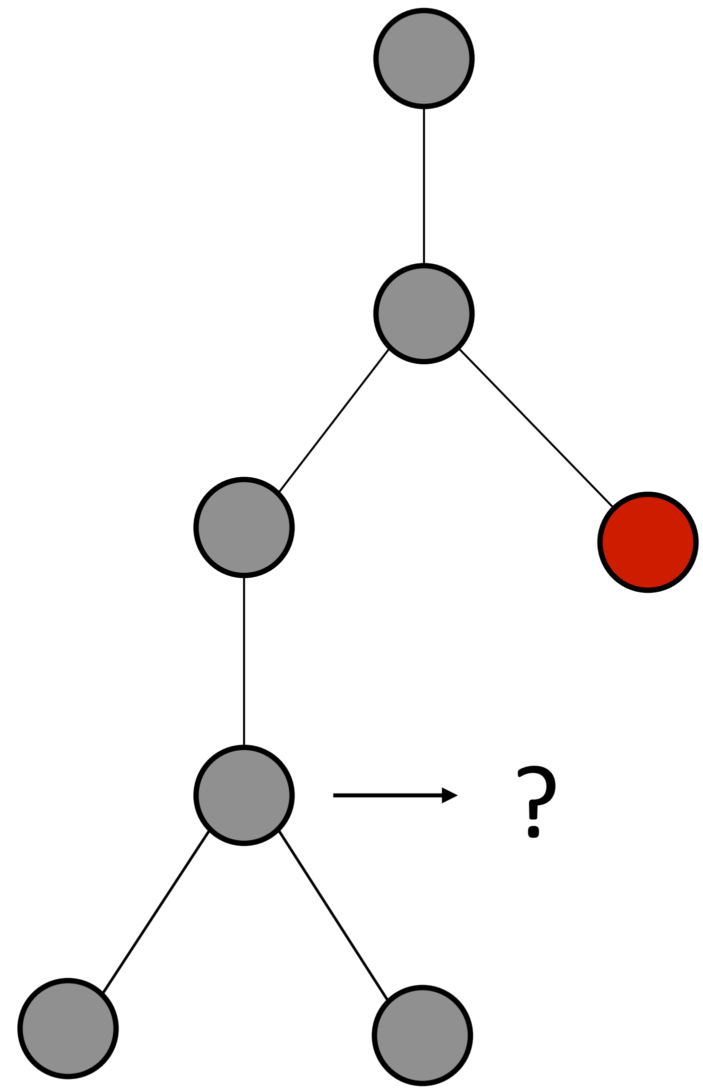
# Tree walks

- A **depth-first** walk
  - Then we try the parent's next sibling



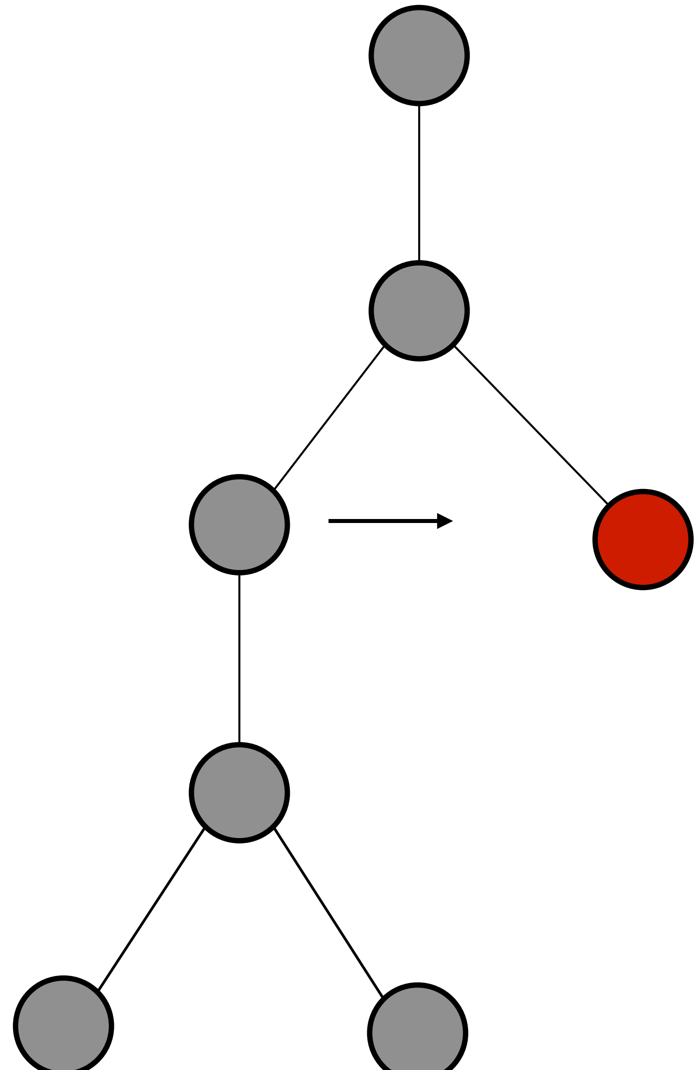
# Tree walks

- A **depth-first** walk
  - Then we try the parent's next sibling
  - But, tragically, it's an only child



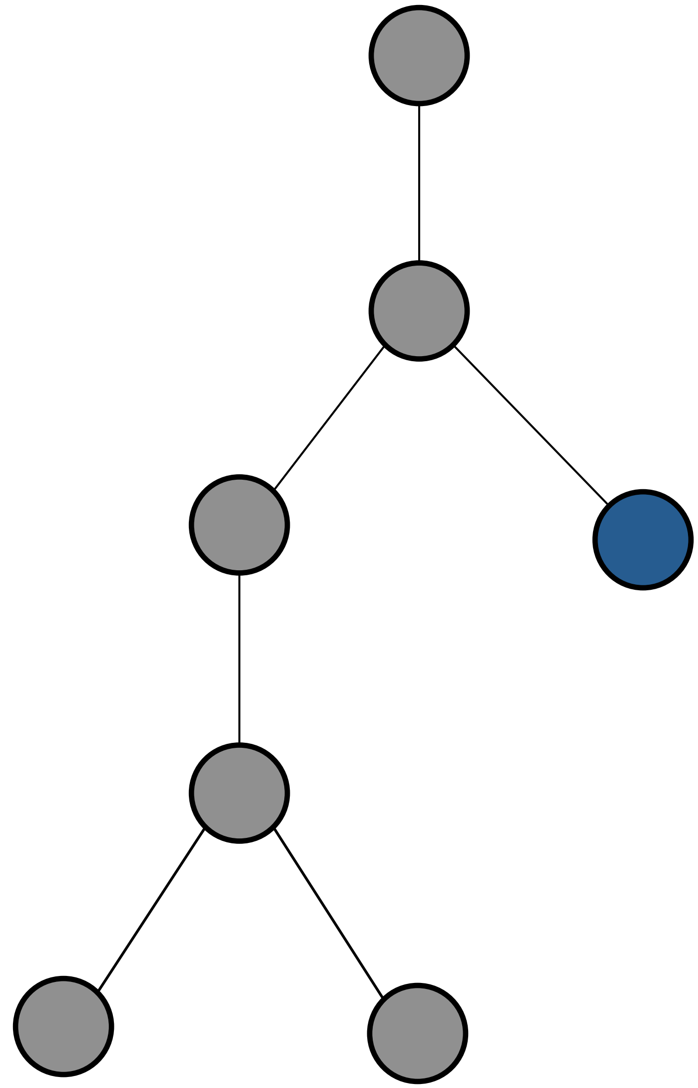
# Tree walks

- A **depth-first** walk
  - Then we try the parent's next sibling
  - But, tragically, it's an only child
  - So we move to its parent



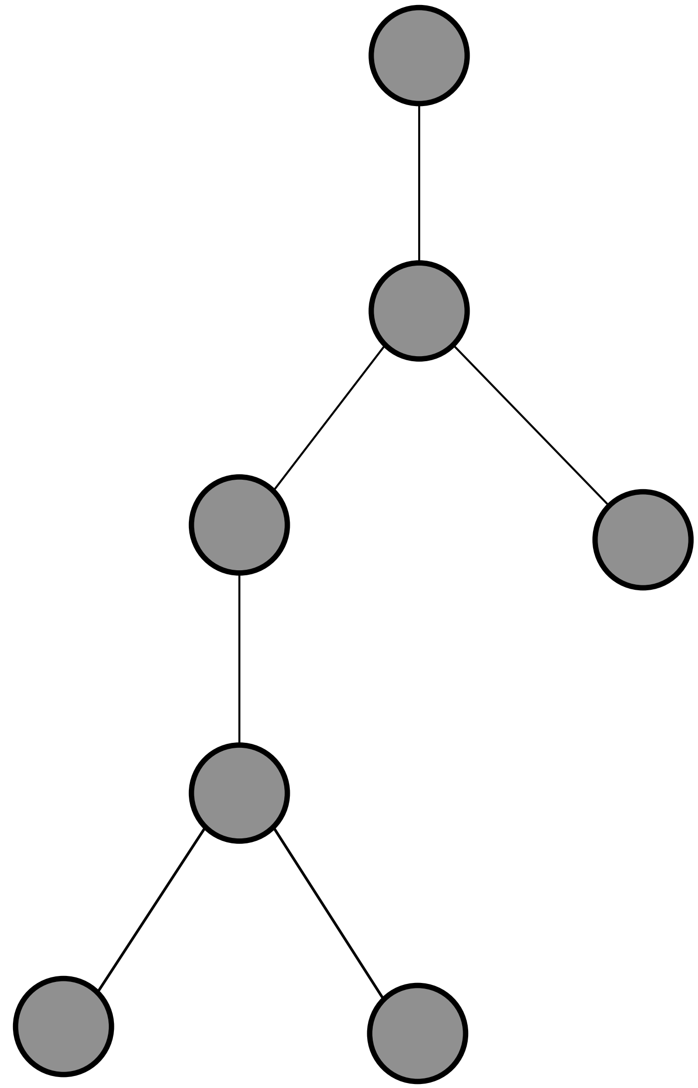
# Tree walks

- A **depth-first** walk
  - Then we try the parent's next sibling
  - But, tragically, it's an only child
  - So we move to its parent
  - And select its next sibling



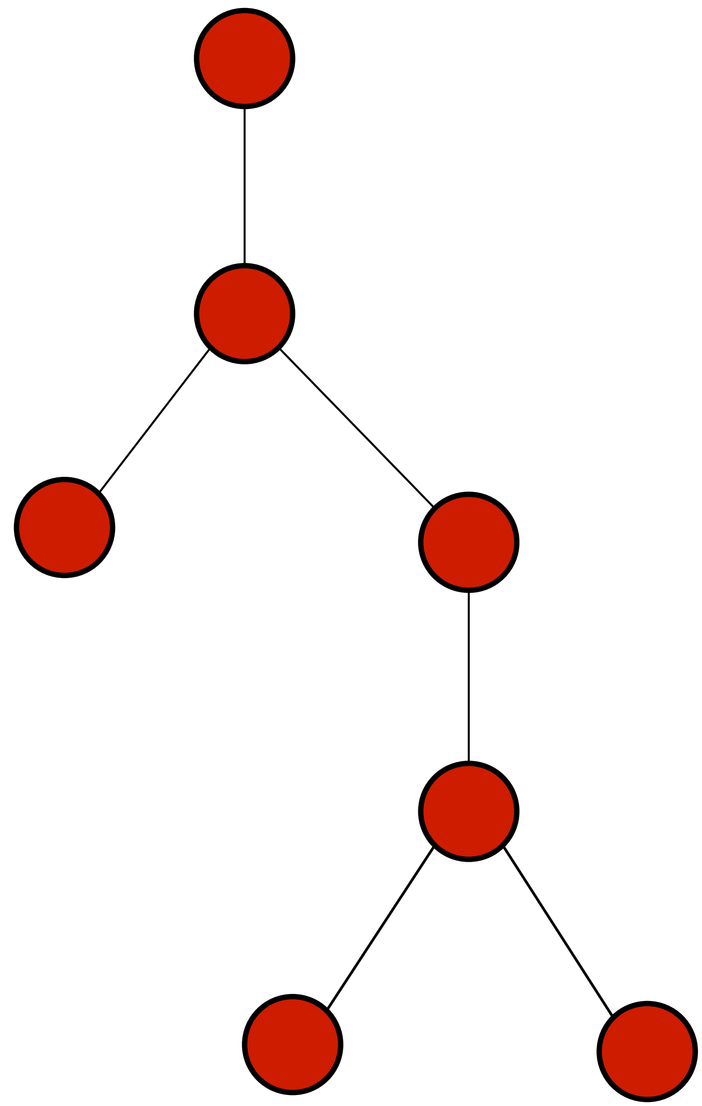
# Tree walks

- A **depth-first** walk
  - Then we try the parent's next sibling
  - But, tragically, it's an only child
  - So we move to its parent
  - And select its next sibling
  - After which, we've walked the whole tree



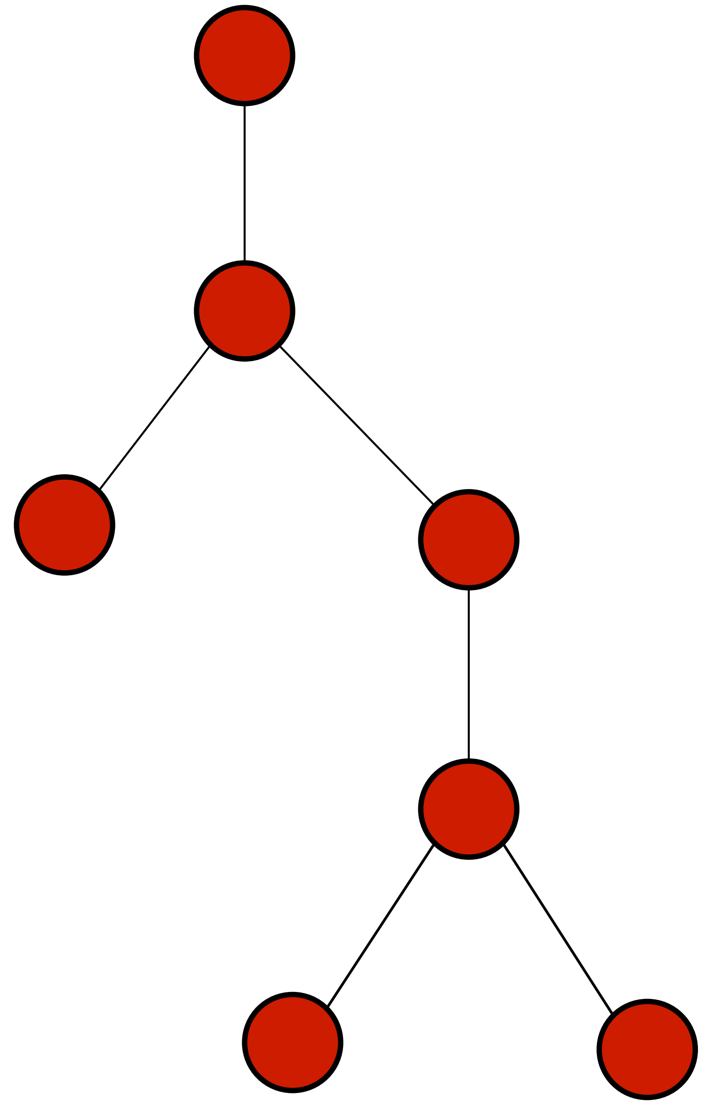
# Tree walks

- Depth-first walks go **subtree to subtree**
- Breadth-first walks go **level to level**

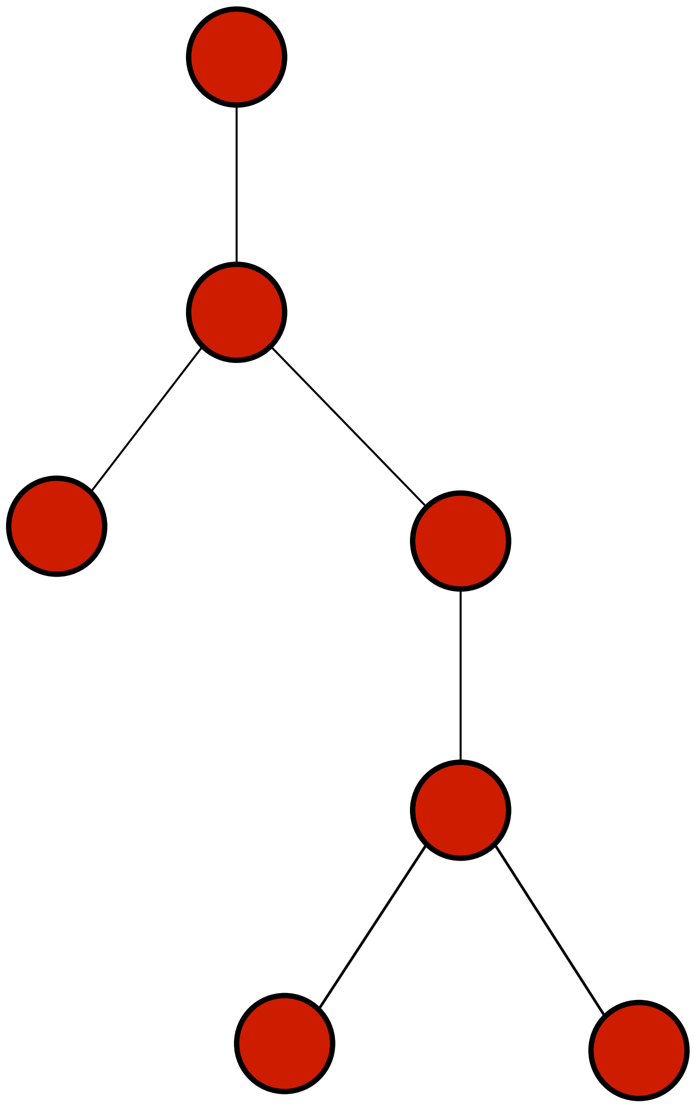


# Tree walks

- When **depth-first** comes to a node
  - It walks it’s children **immediately**
- When **breadth-first** comes to a node
  - It **remembers** its children to be walked in the future

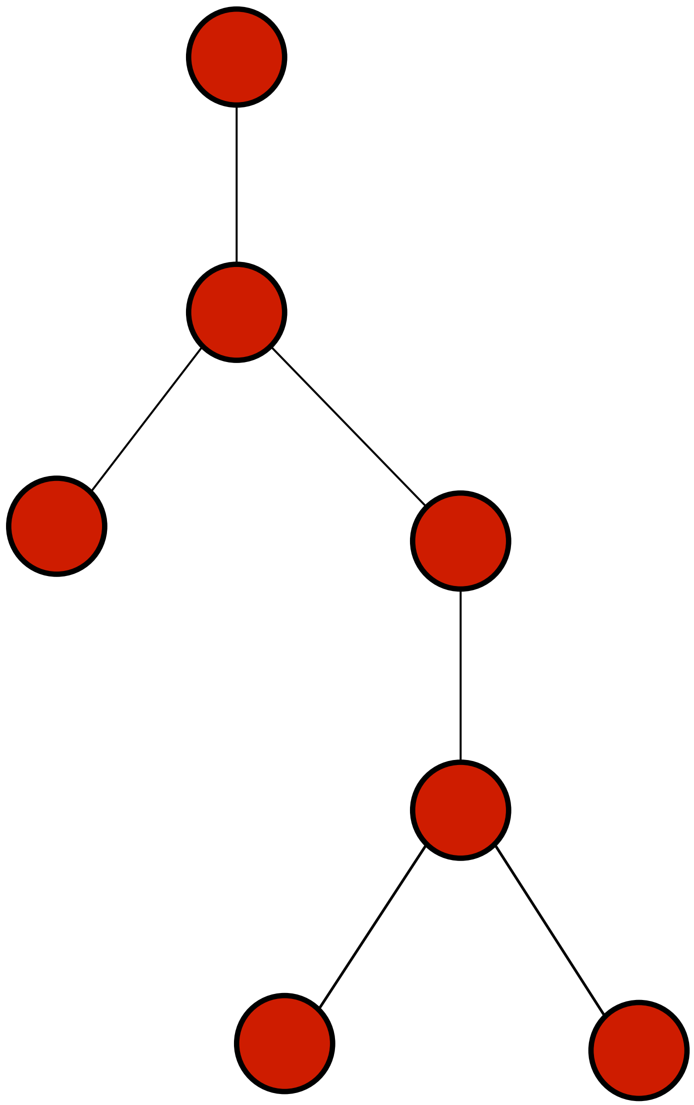


# Breadth-first walk



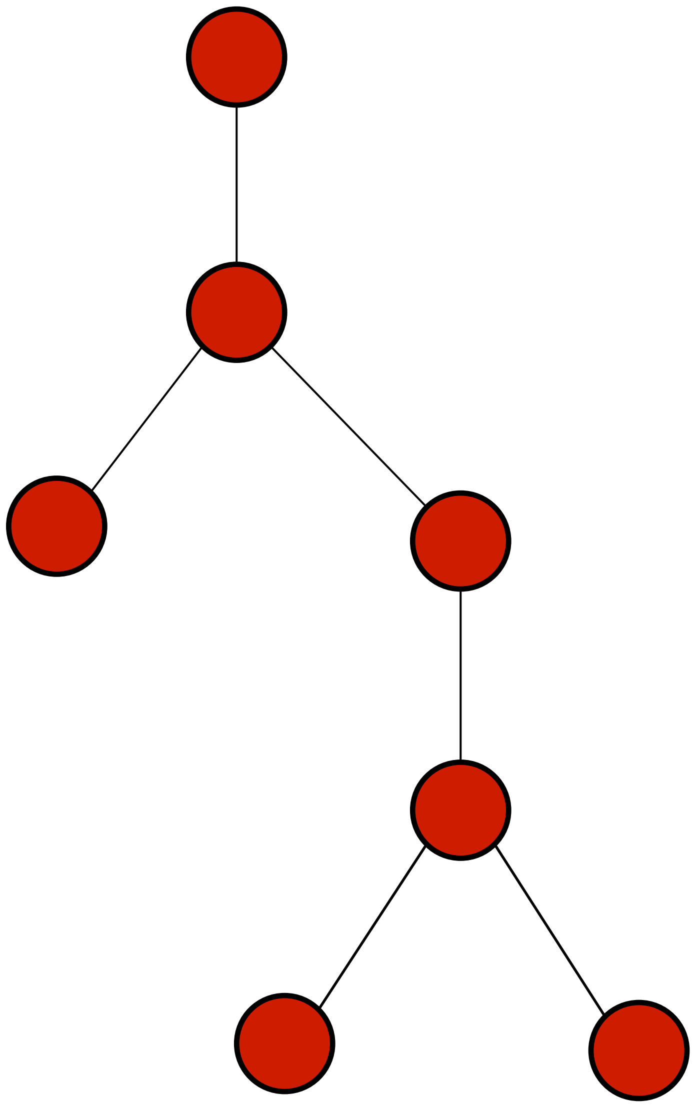
- Needs to remember
  - The children of nodes it's walked
  - But that haven't yet been walked themselves
- So it needs to store a collection of nodes
- What kind of collection?

# Breadth-first walk



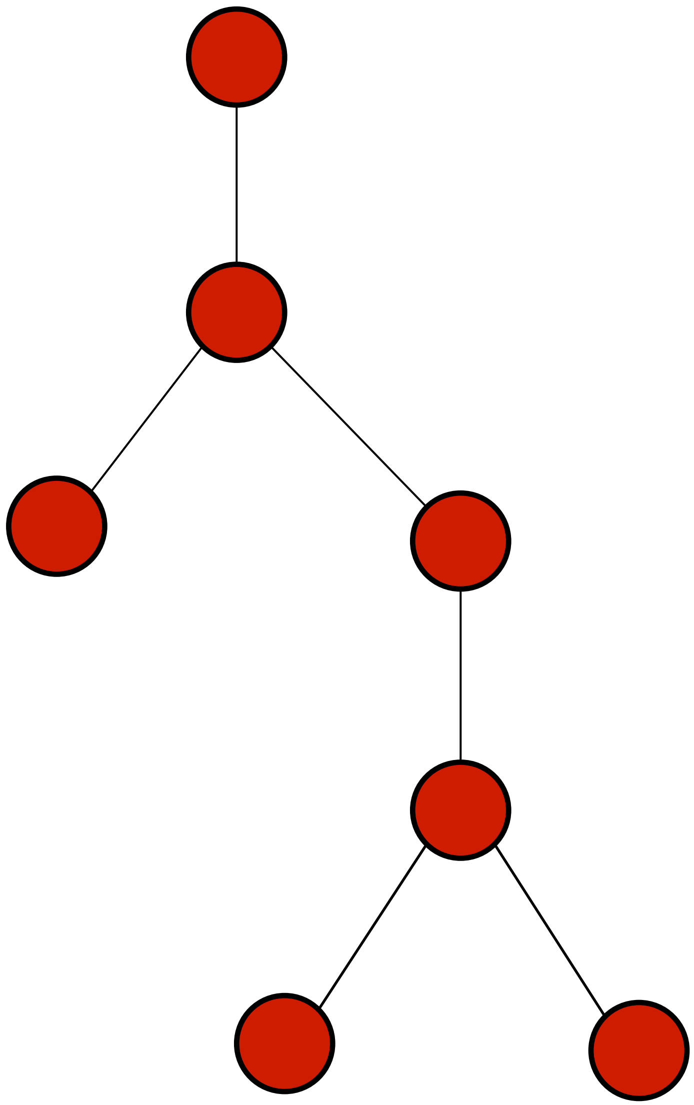
- Needs to finish current level of tree
  - Before moving to nodes in the next level
- So the nodes that get walked first
  - Are the nodes that are added first to the collection

# Breadth-first walk



- So we walk the nodes in the order that they're added to the collection
- That sounds like a queue!

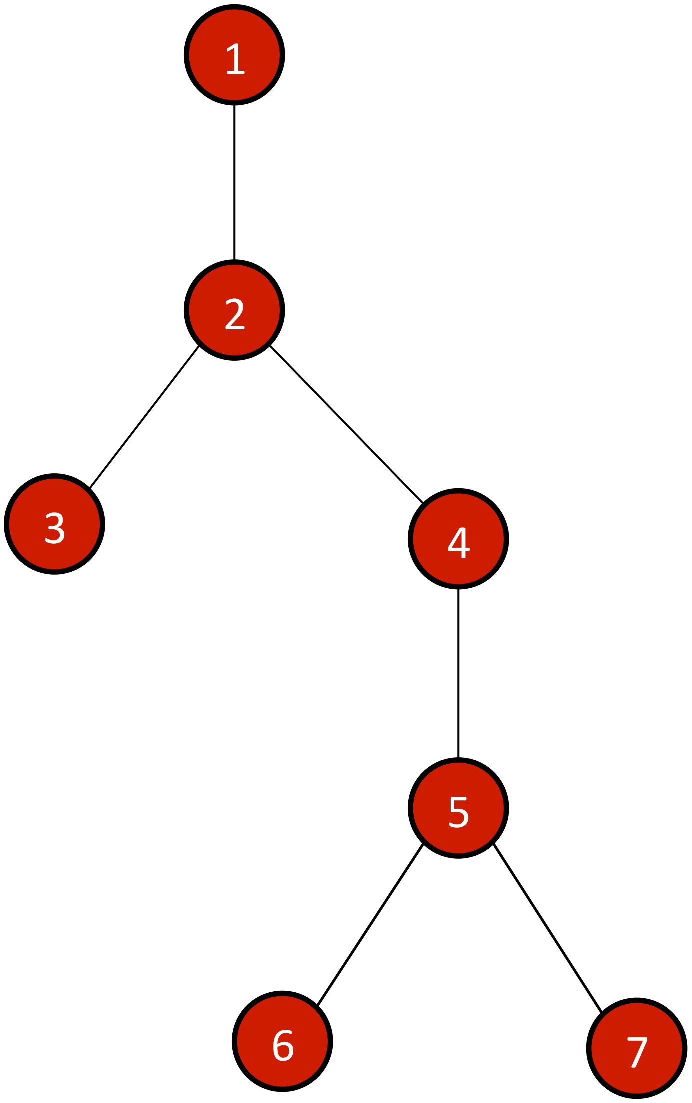
# Breadth-first walk



*Pseudocode:*

```
BreadthFirst(root) {
   q = empty queue
   q.Enqueue(root)
   while q not empty {
      node = q.Dequeue()
      for each child c of node
         q.Enqueue(c)
   }
}
```

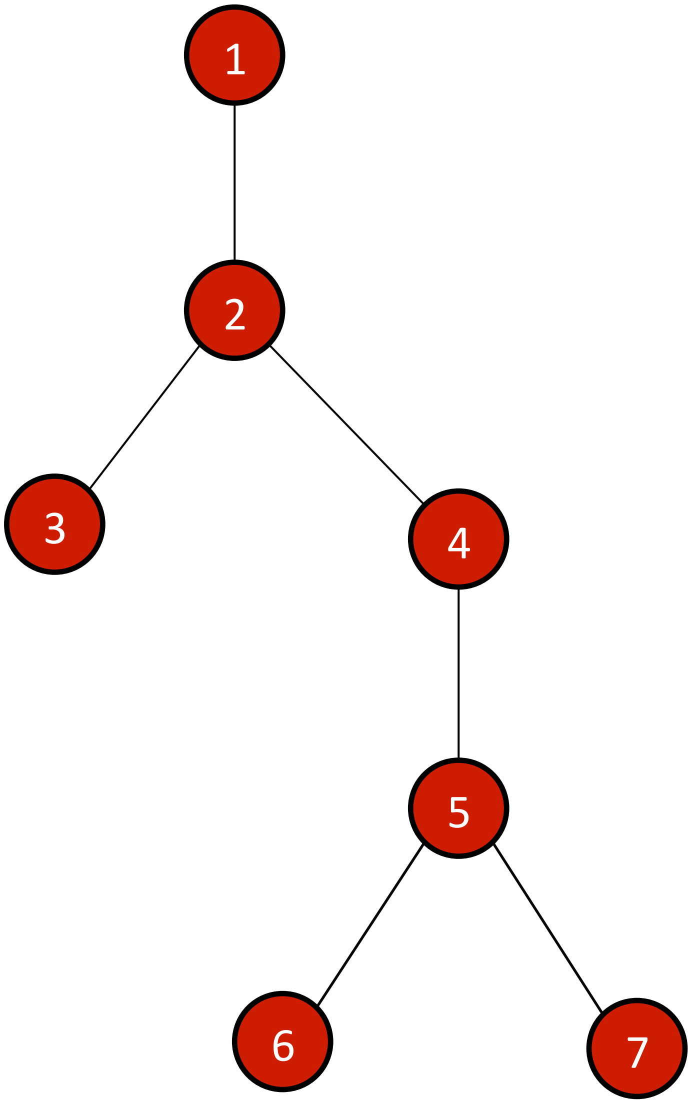
# Breadth-first walk





Start with empty queue

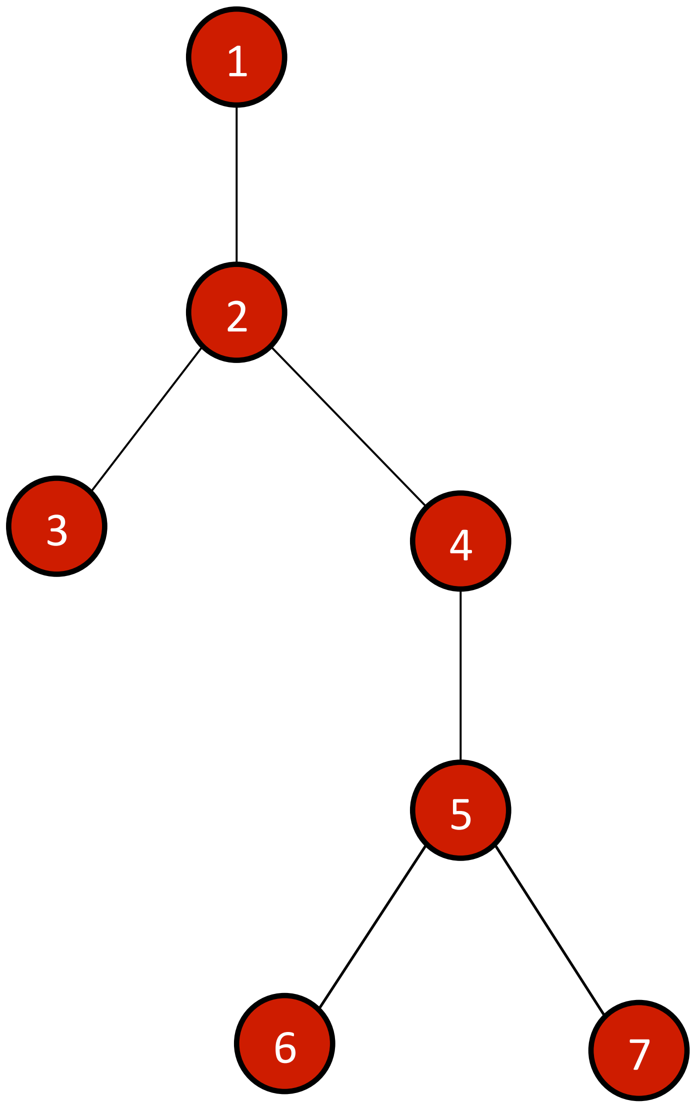
# Breadth-first walk




Queue


Add the root

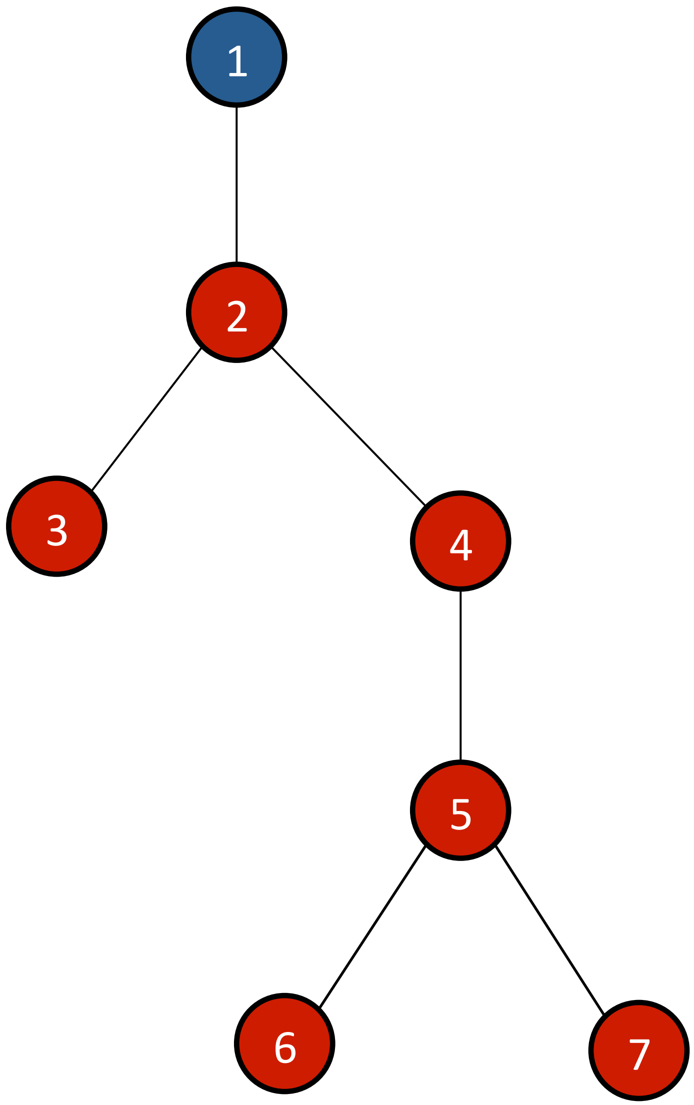
# Breadth-first walk




Queue


Now repeat until the queue is empty:

# Breadth-first walk

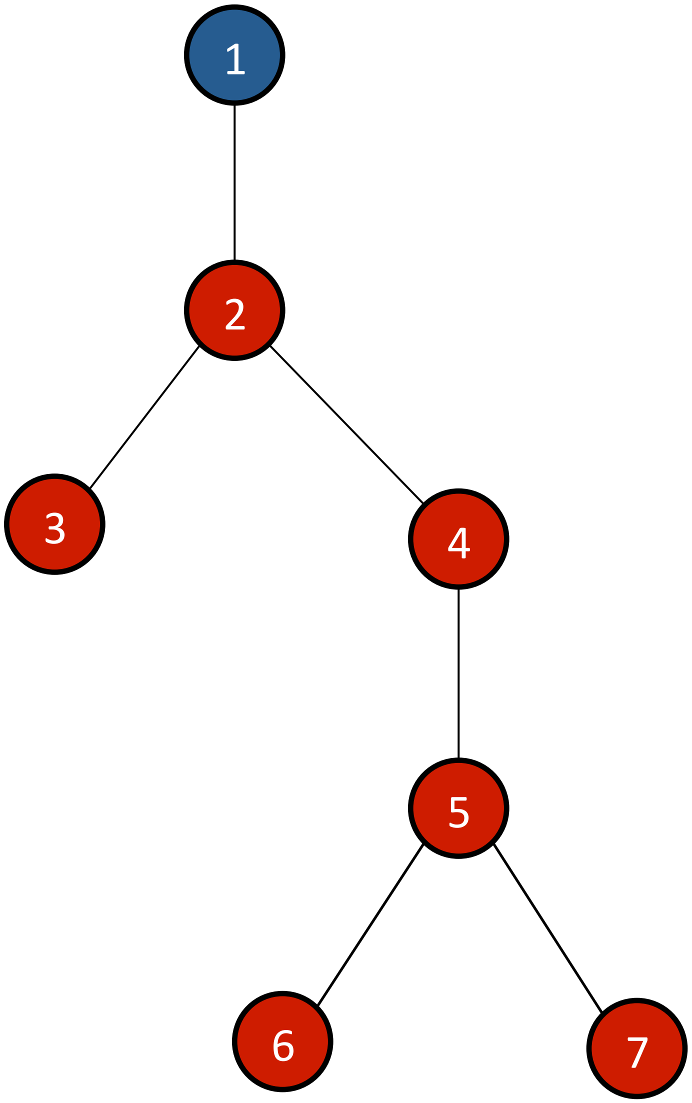



Queue

empty

Dequeue the next node

# Breadth-first walk

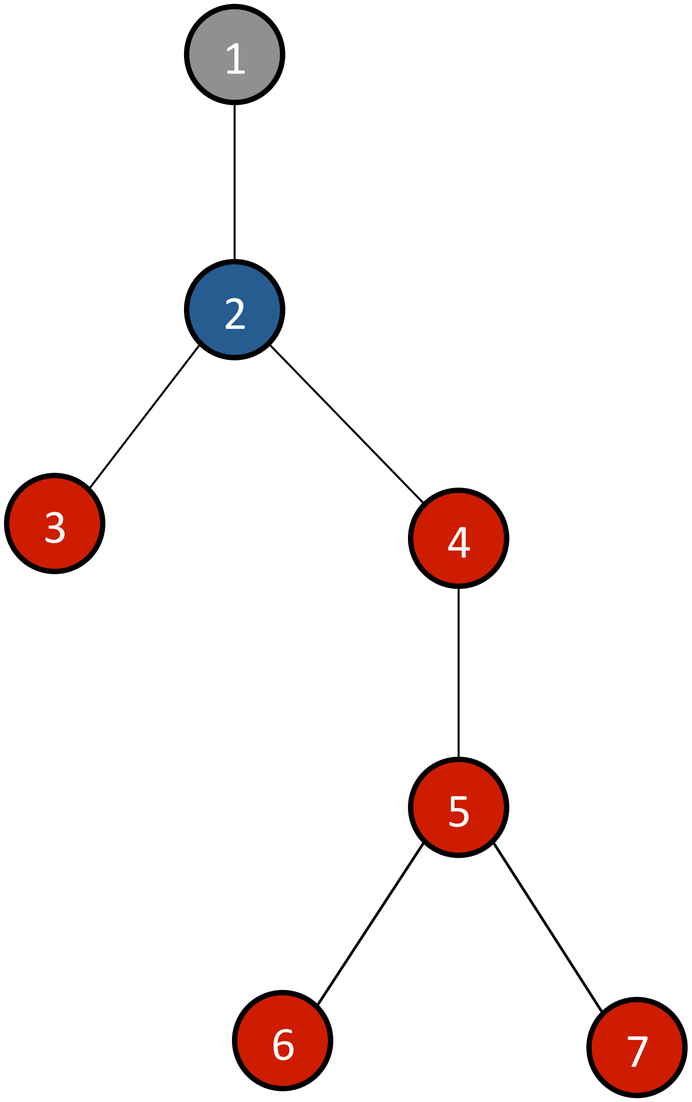



Queue

| |
|---|
| 2 |

Add its children to queue

# Breadth-first walk

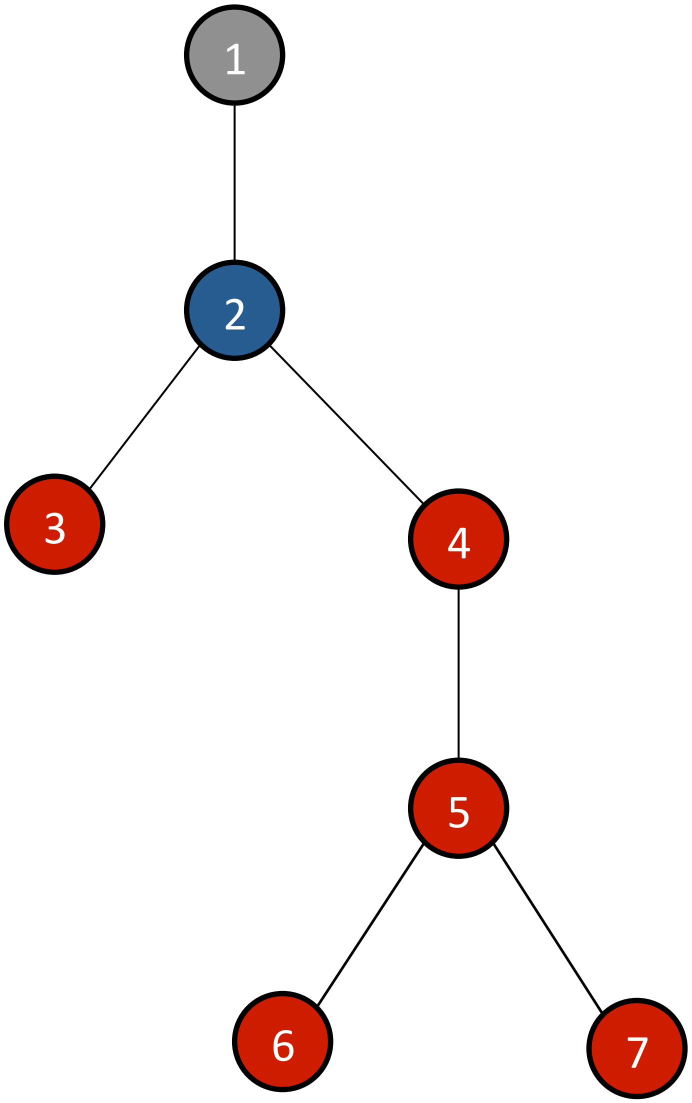



Queue

empty

Dequeue

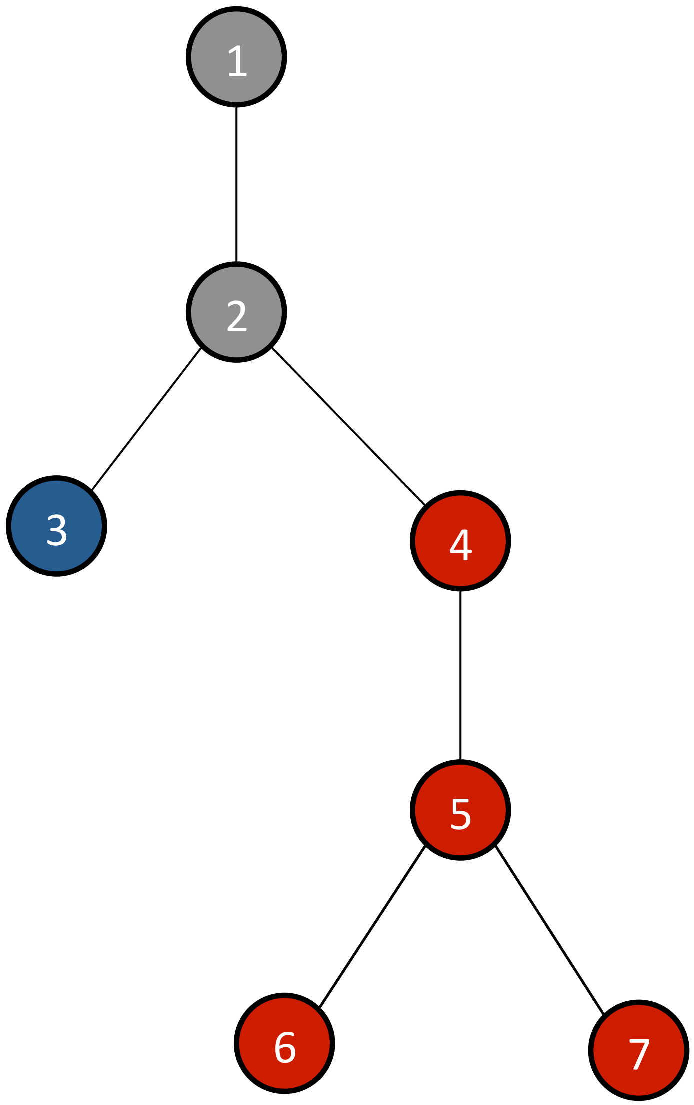
# Breadth-first walk




Queue


Enqueue children

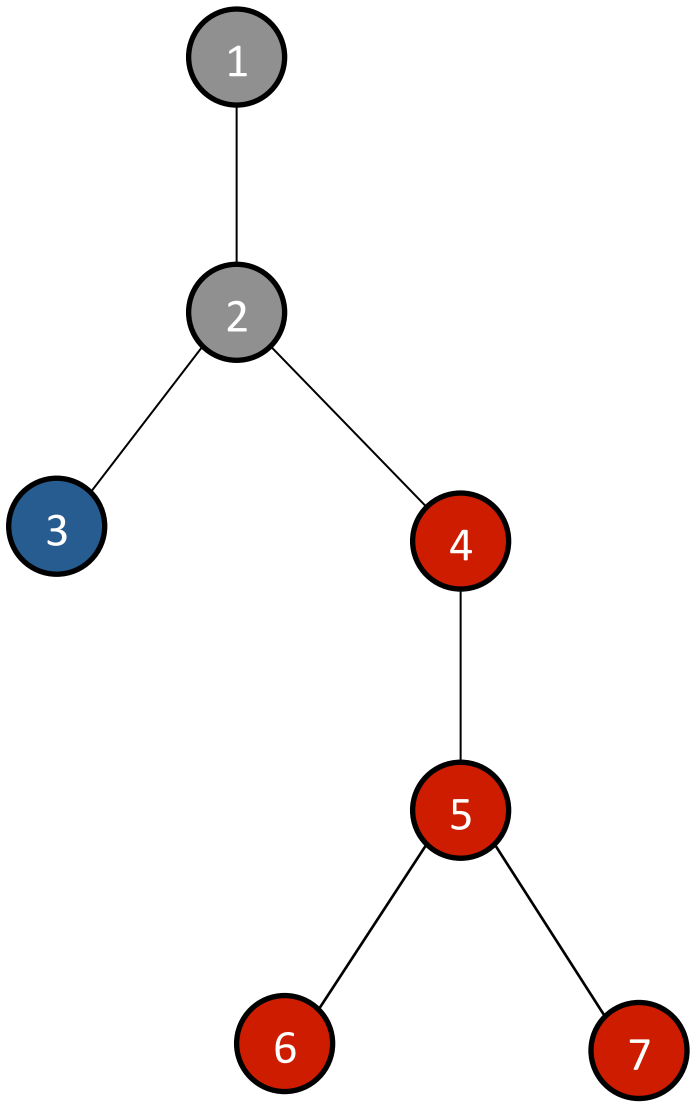
# Breadth-first walk




Queue


Dequeue

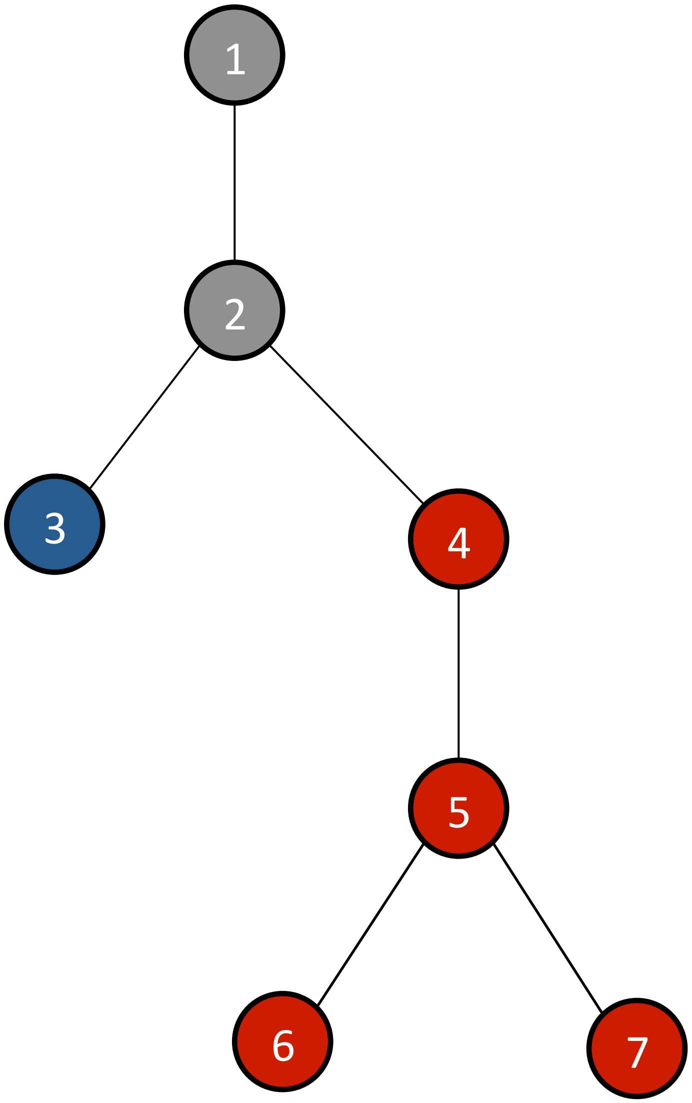
# Breadth-first walk




Queue


Enqueue children

# Breadth-first walk

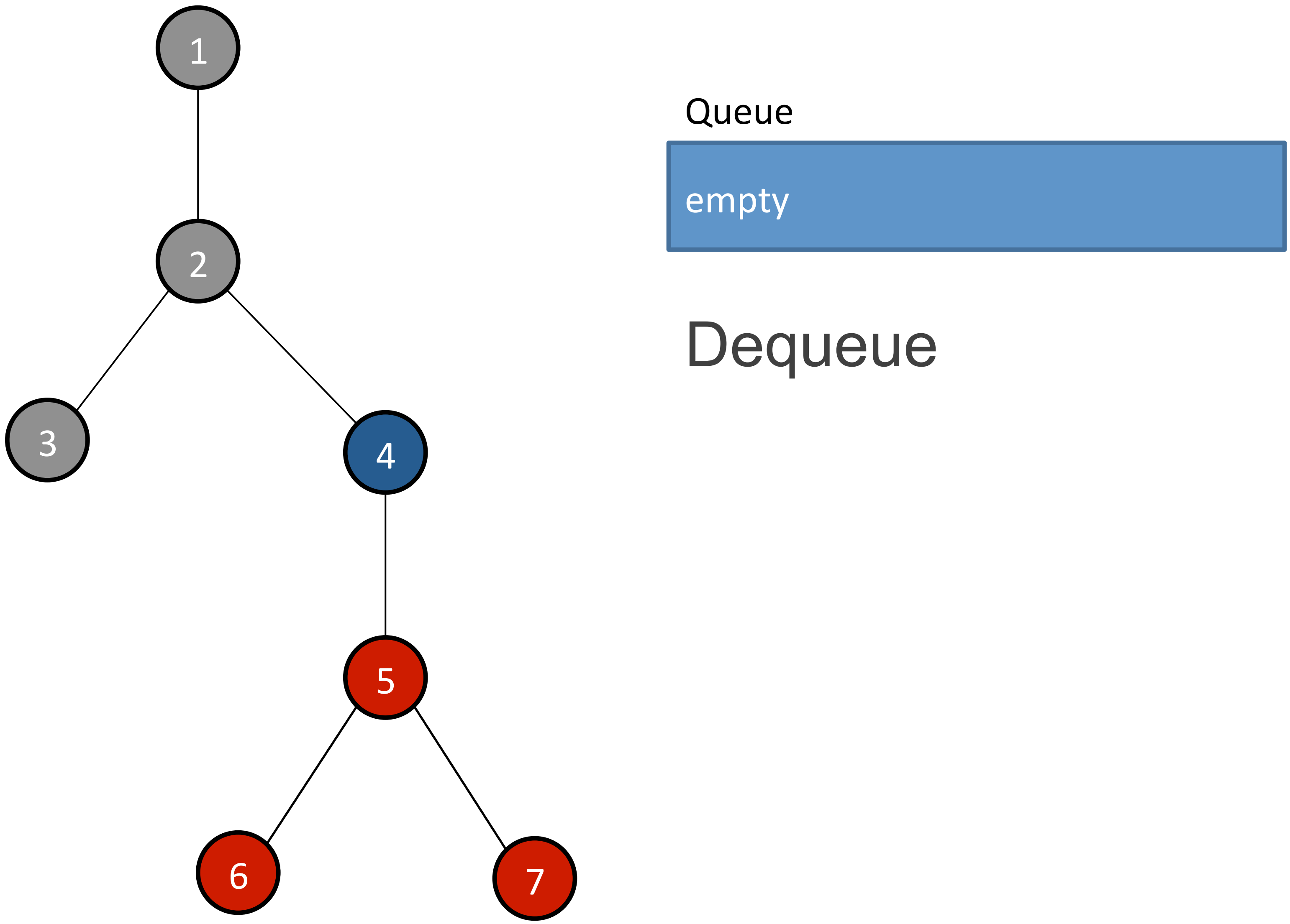



Queue


Enqueue children

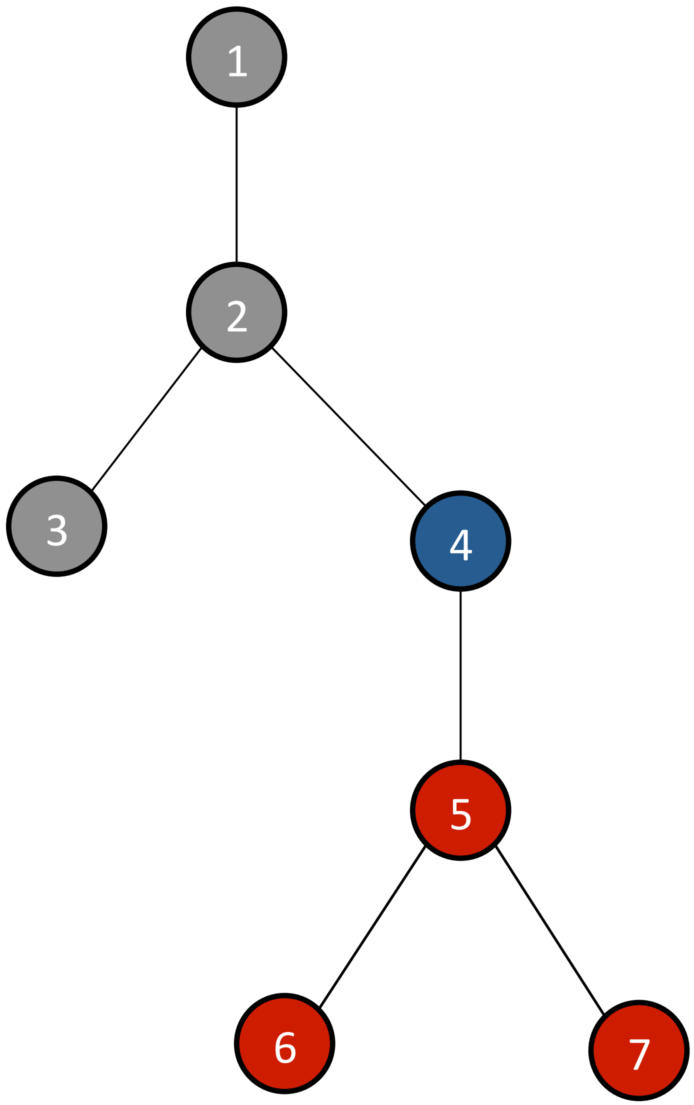
(of course, there aren't any)

# Breadth-first walk




Dequeue

# Breadth-first walk

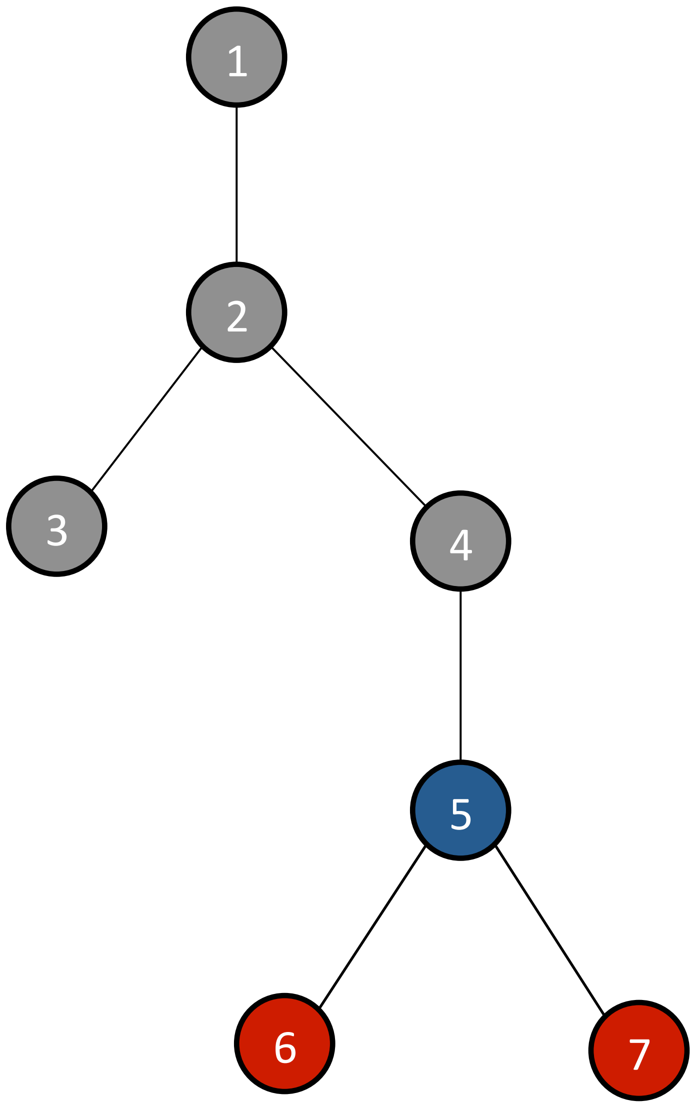



Queue

| |
|---|
| 5 |

Add children

# Breadth-first walk

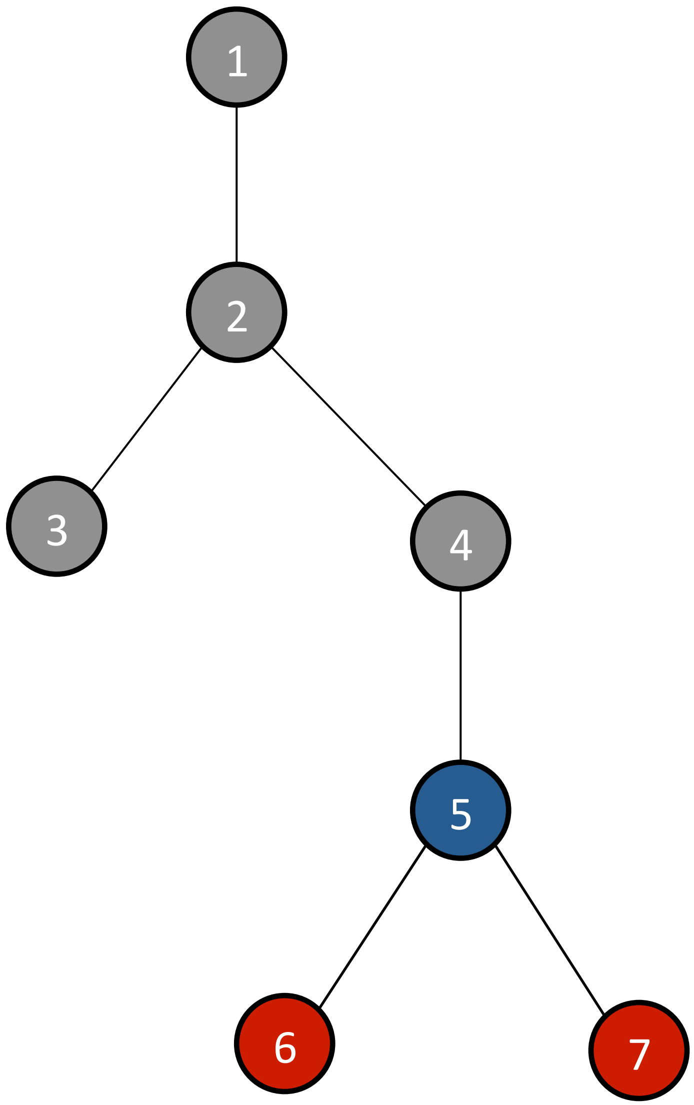



Queue

empty

Dequeue

# Breadth-first walk

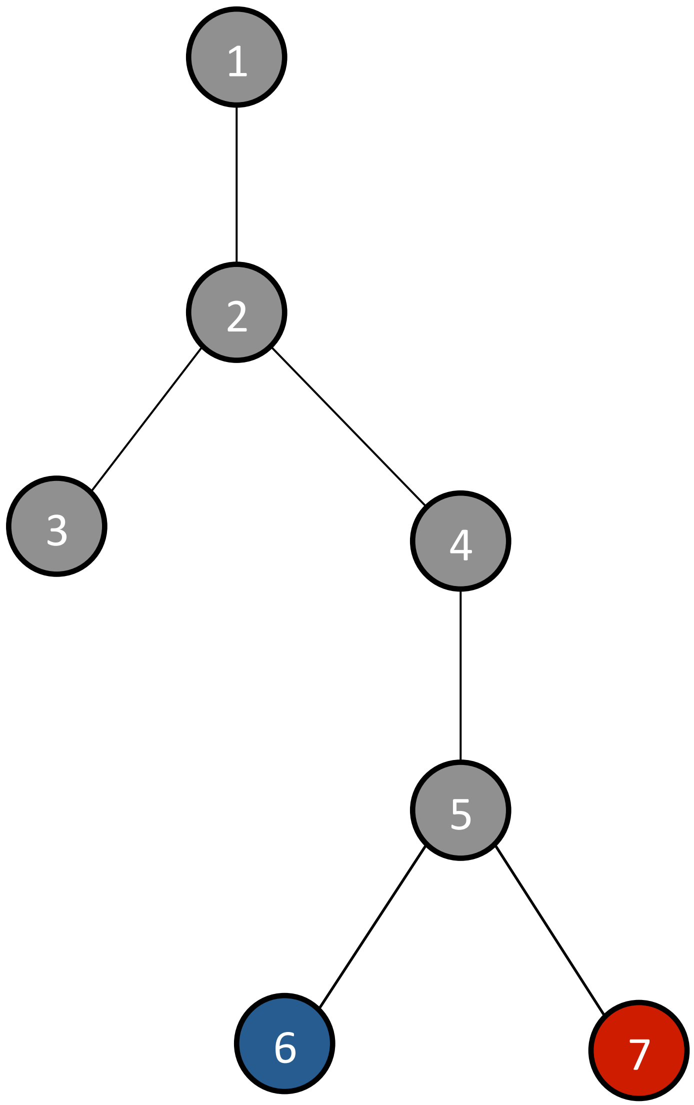



Queue

6 7

Add children

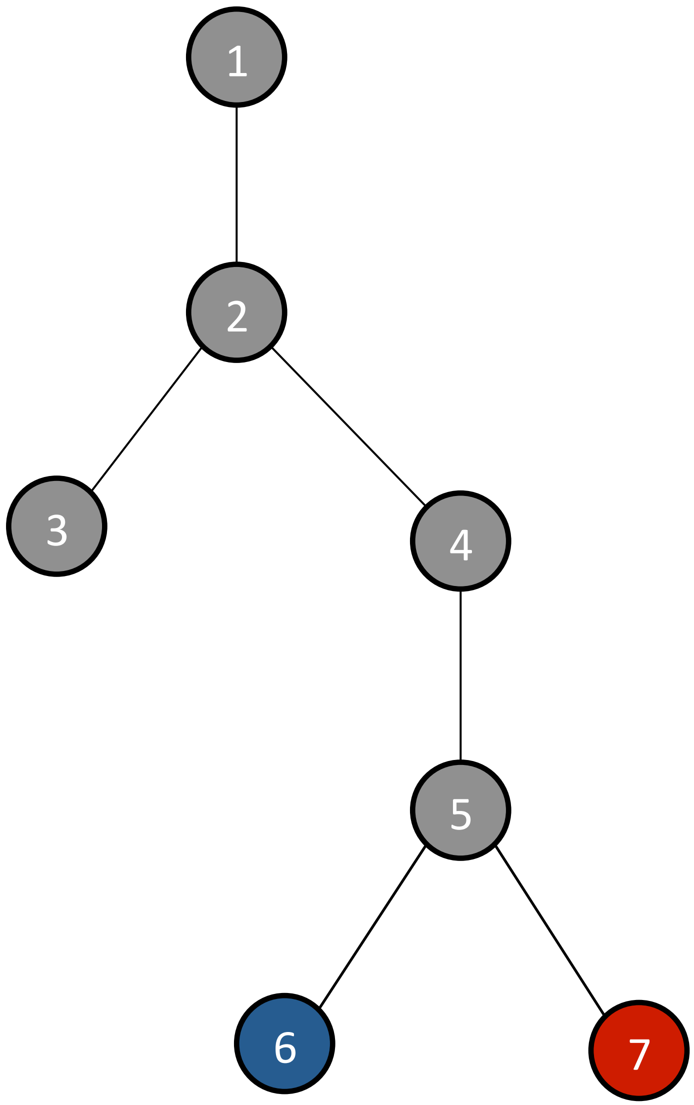
# Breadth-first walk




Queue


Dequeue

# Breadth-first walk

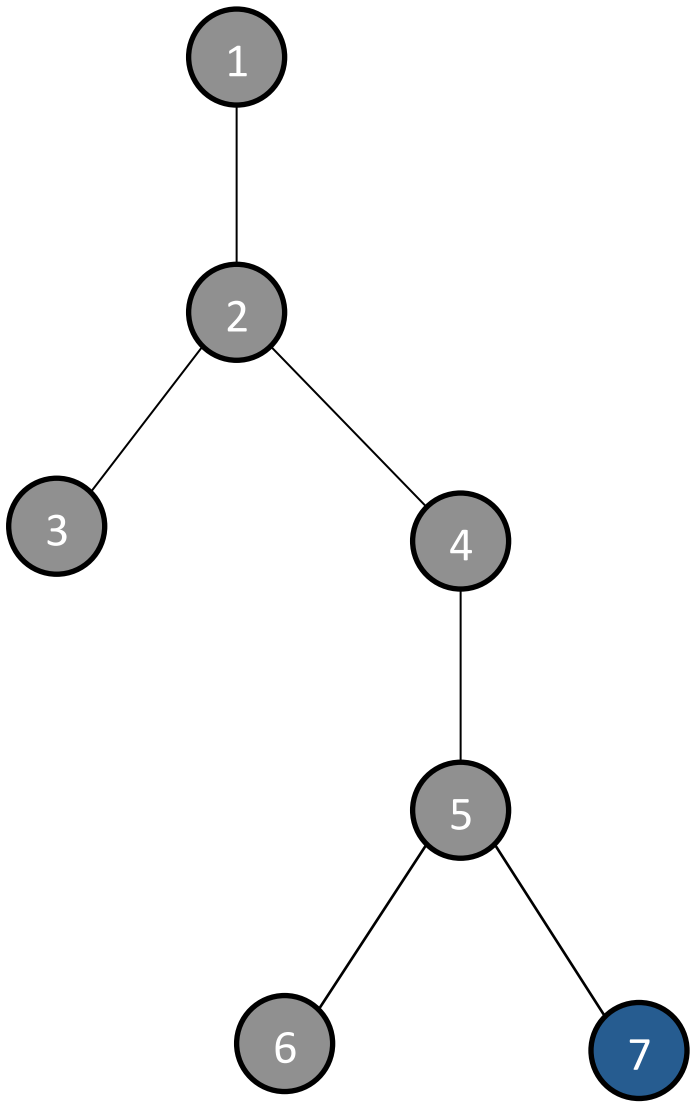



Queue

7

Add children (none)

# Breadth-first walk

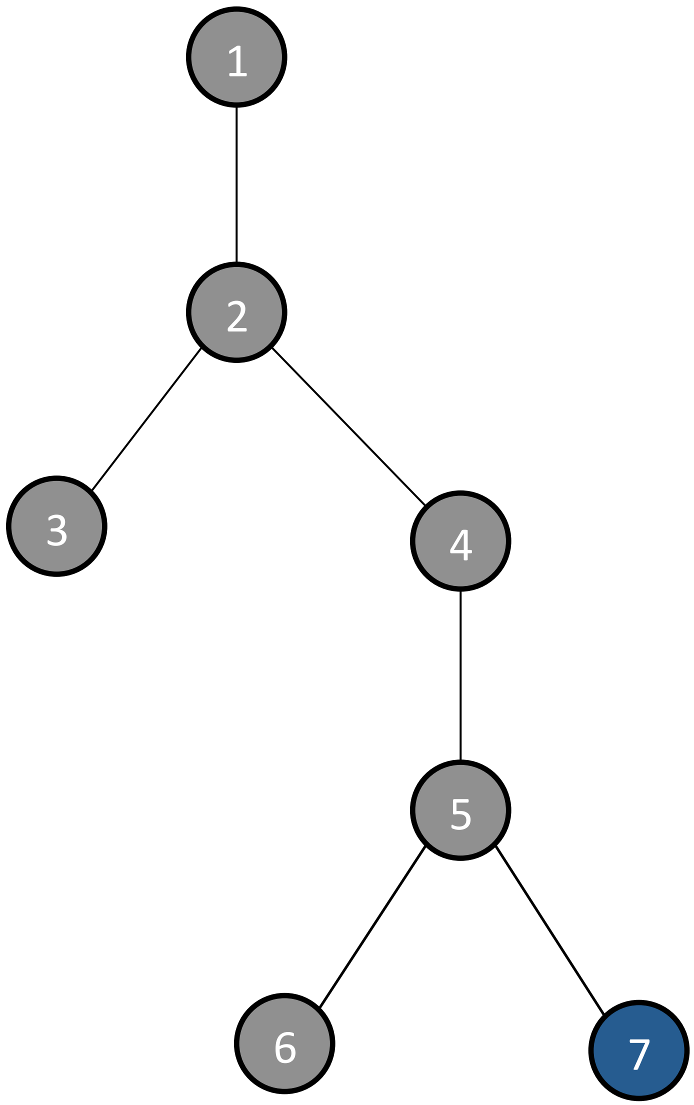



Queue

empty

Dequeue

# Breadth-first walk

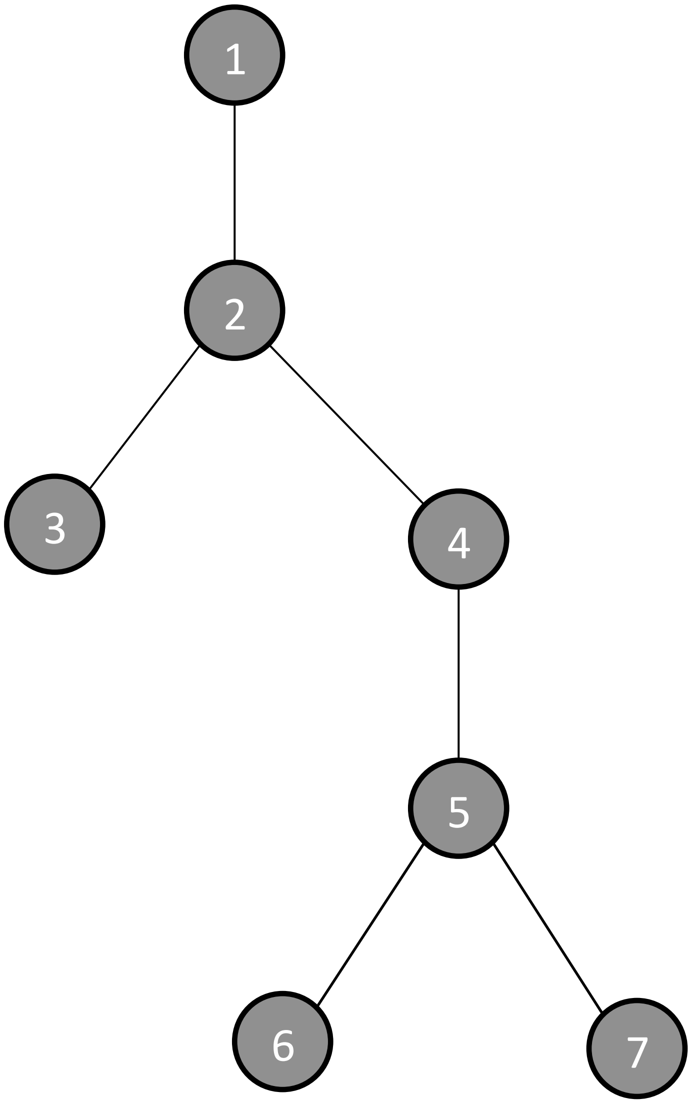



Queue

empty

Add children (none)

# Breadth-first walk

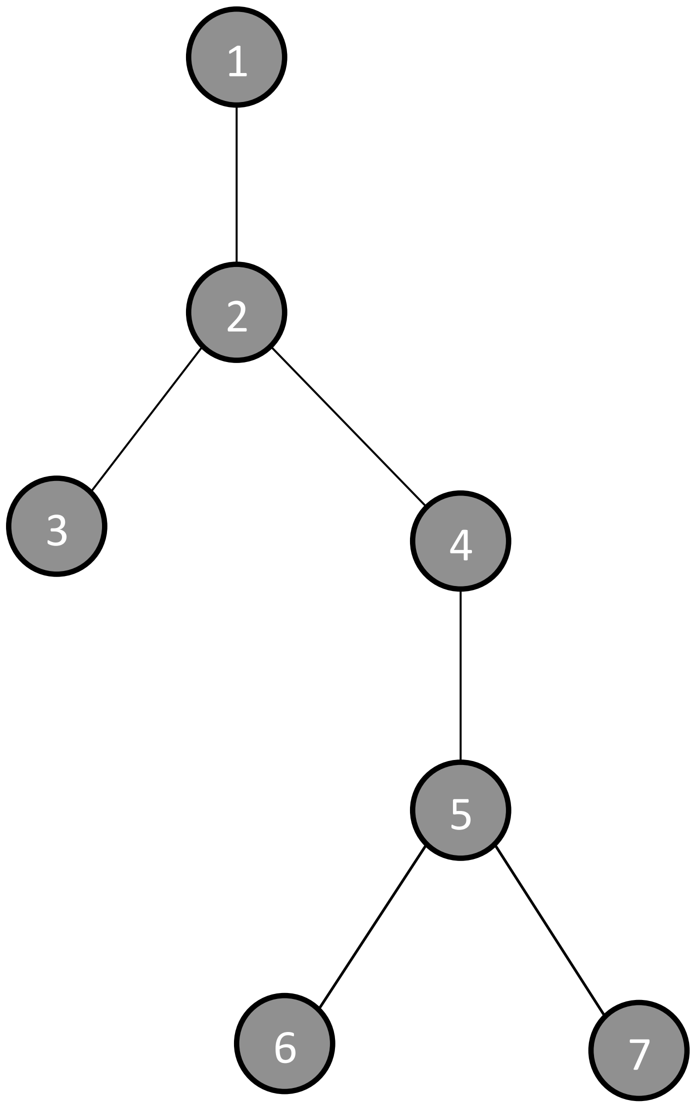



Queue

empty

Dequeue

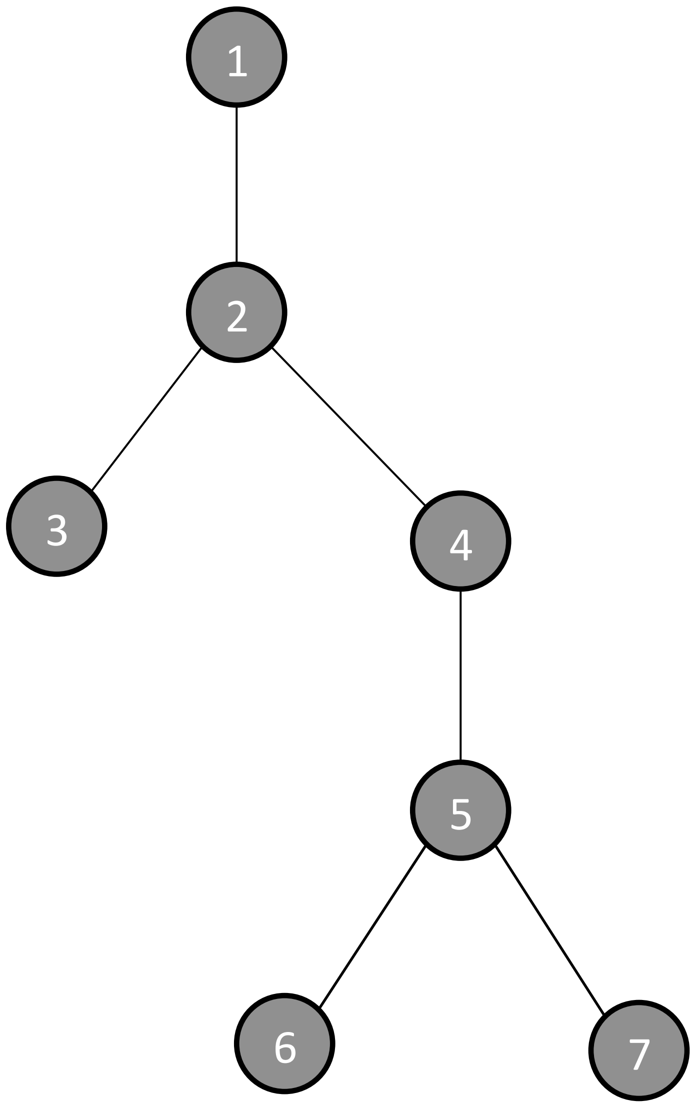
# Breadth-first walk





Dequeue – queue empty

# Breadth-first walk




Queue

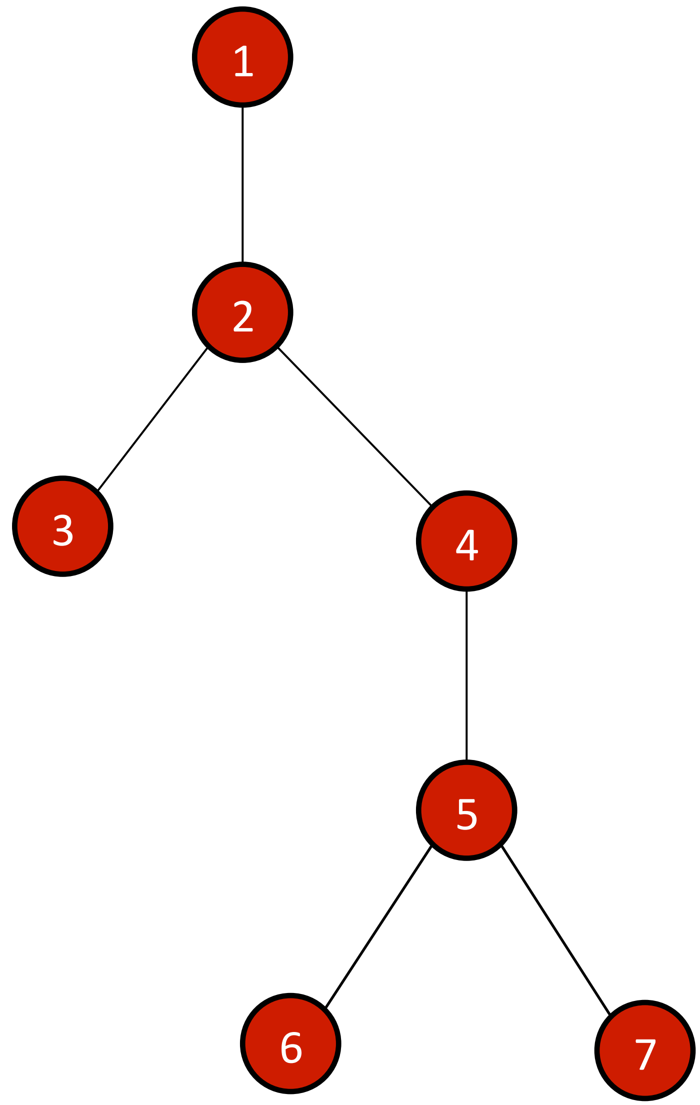
empty

Finished

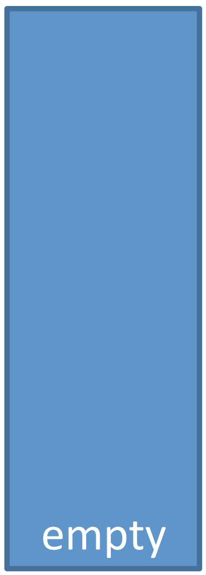
# Using a stack

What happens if we change the queue to a stack?

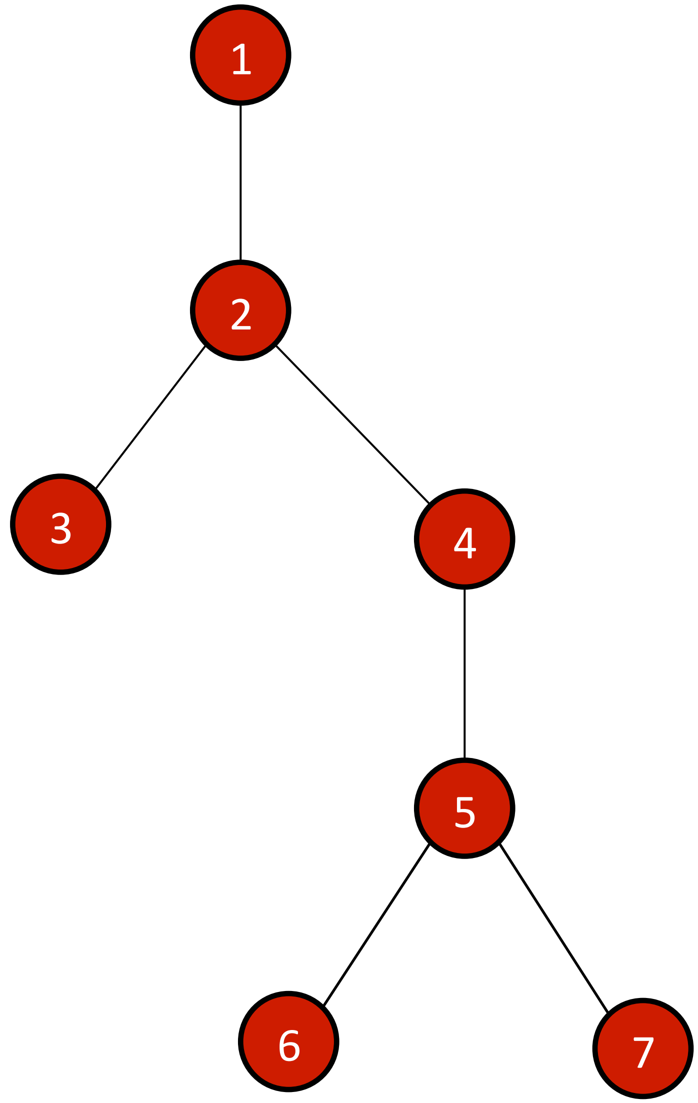
```
StackWalk(root) {
   s = empty stack
   s.Push(root)
   while s not empty {
      node = s.Pop()
      for each child c of node
         s.Push(c)
   }
}
```



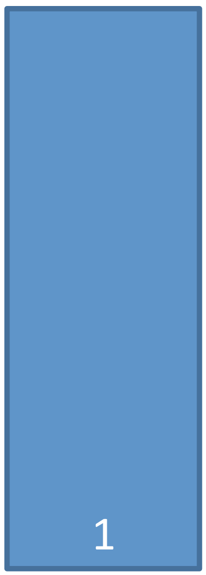
# Using a stack



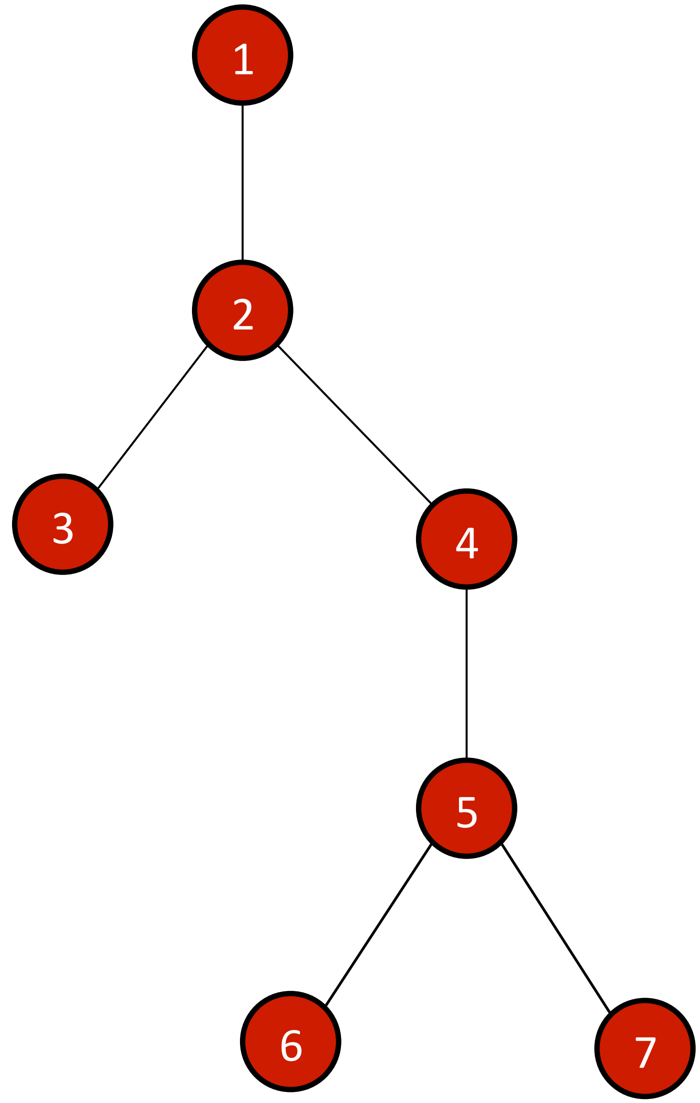

Start with empty stack



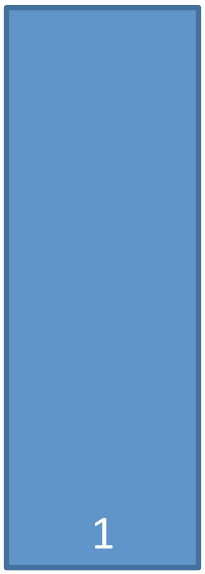
# Using a stack



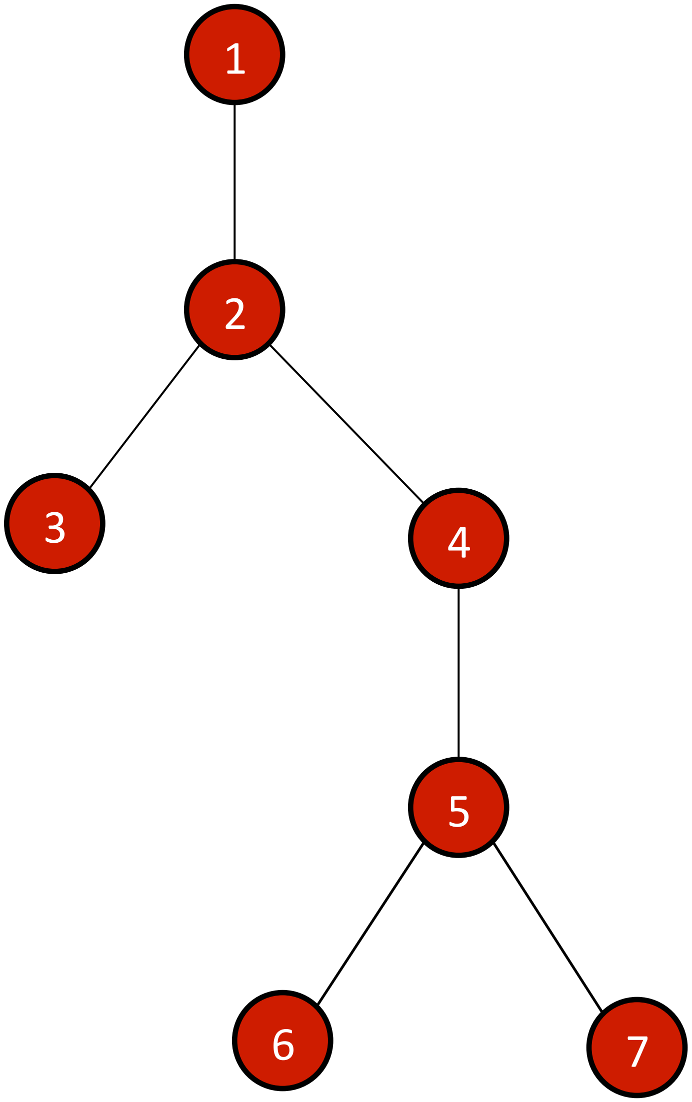

Push the root



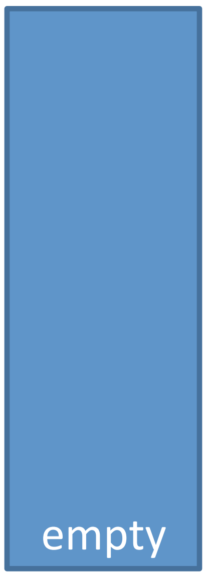
# Using a stack



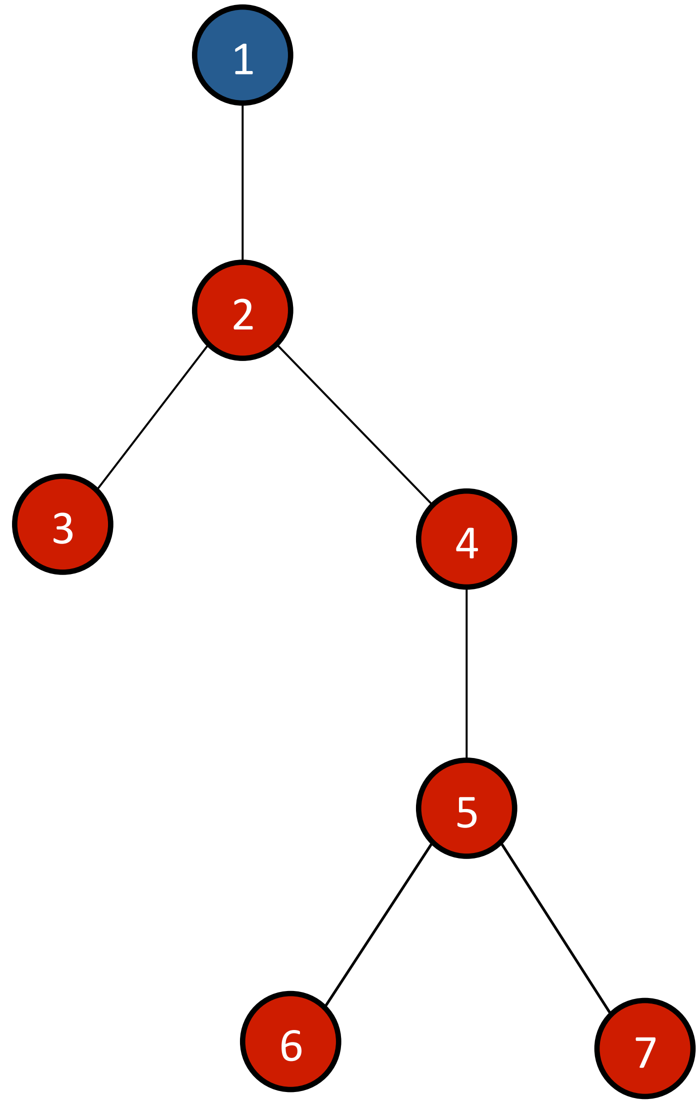

Repeat until stack empty:



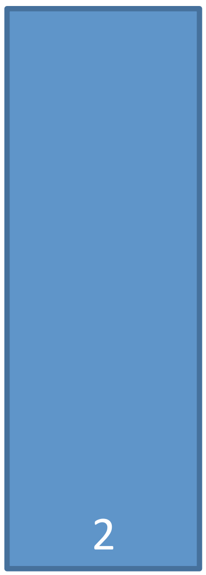
# Using a stack



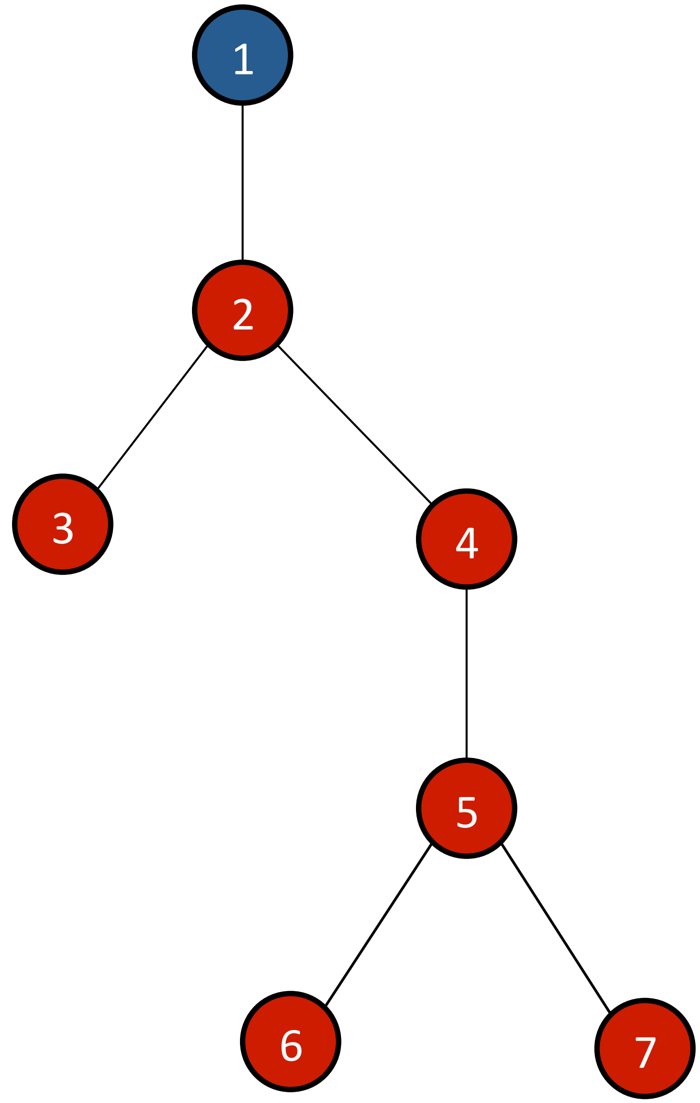

Pop node



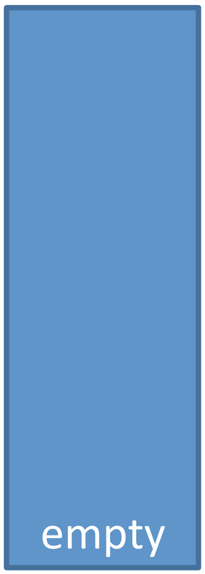
# Using a stack



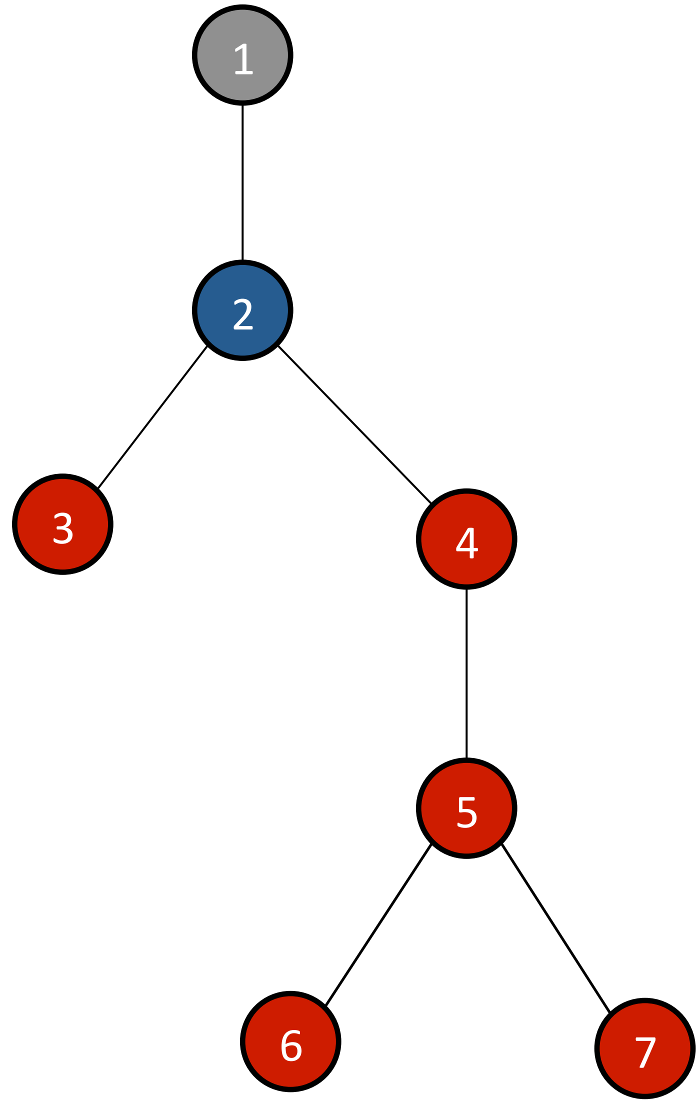

Push children



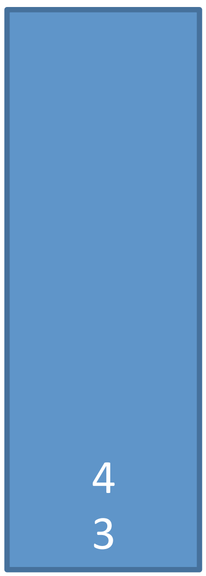
# Using a stack



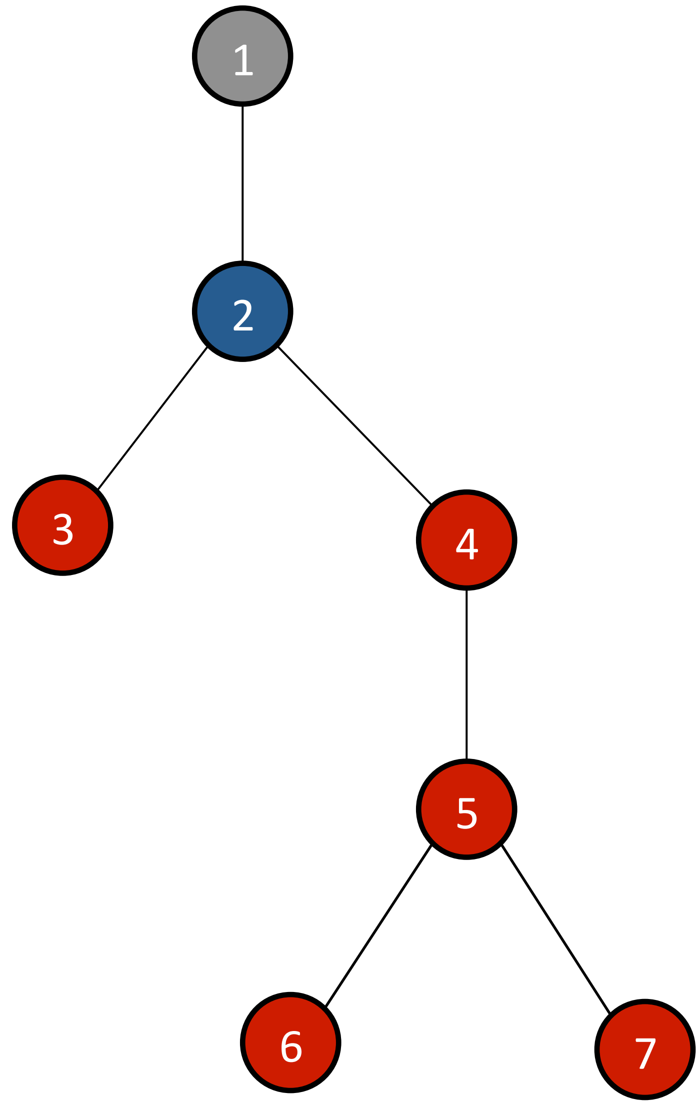

Pop node



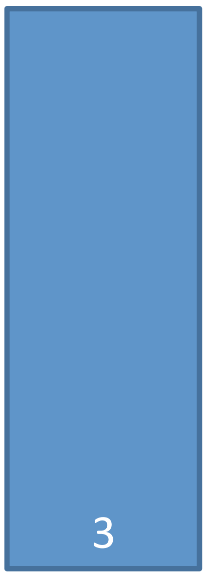
# Using a stack



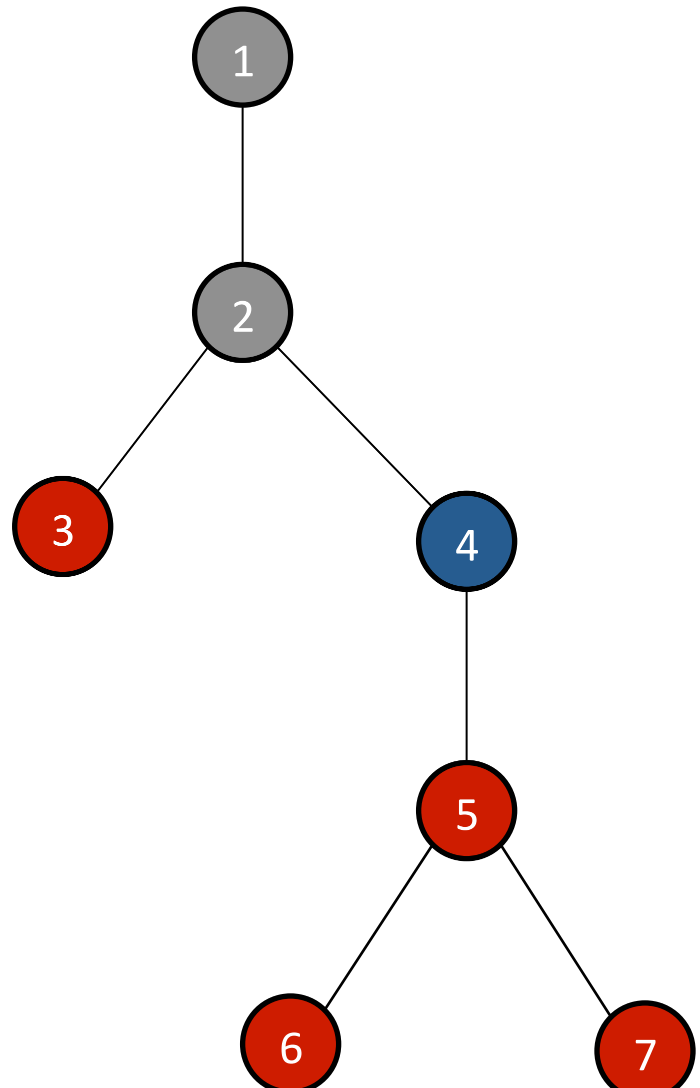

Push children



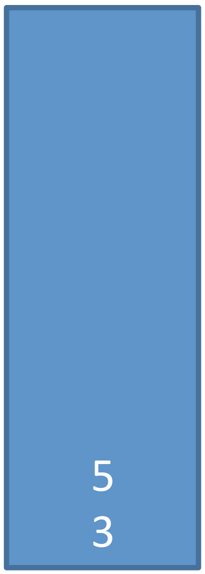
# Using a stack



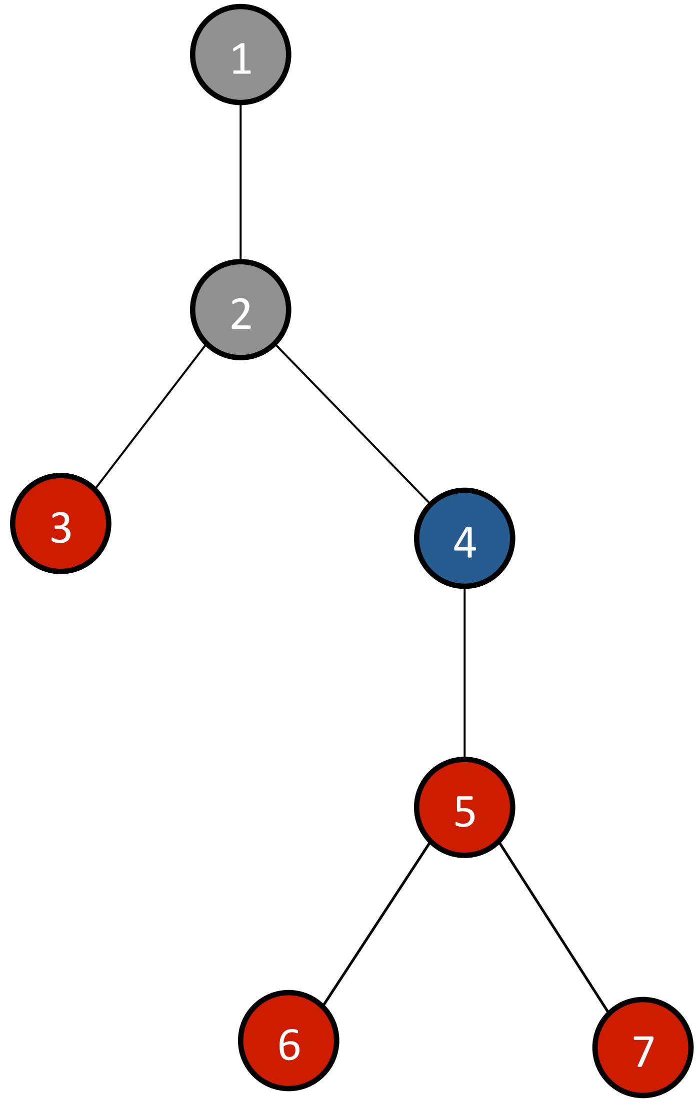

Pop node



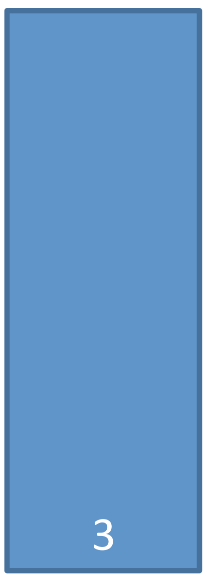
# Using a stack



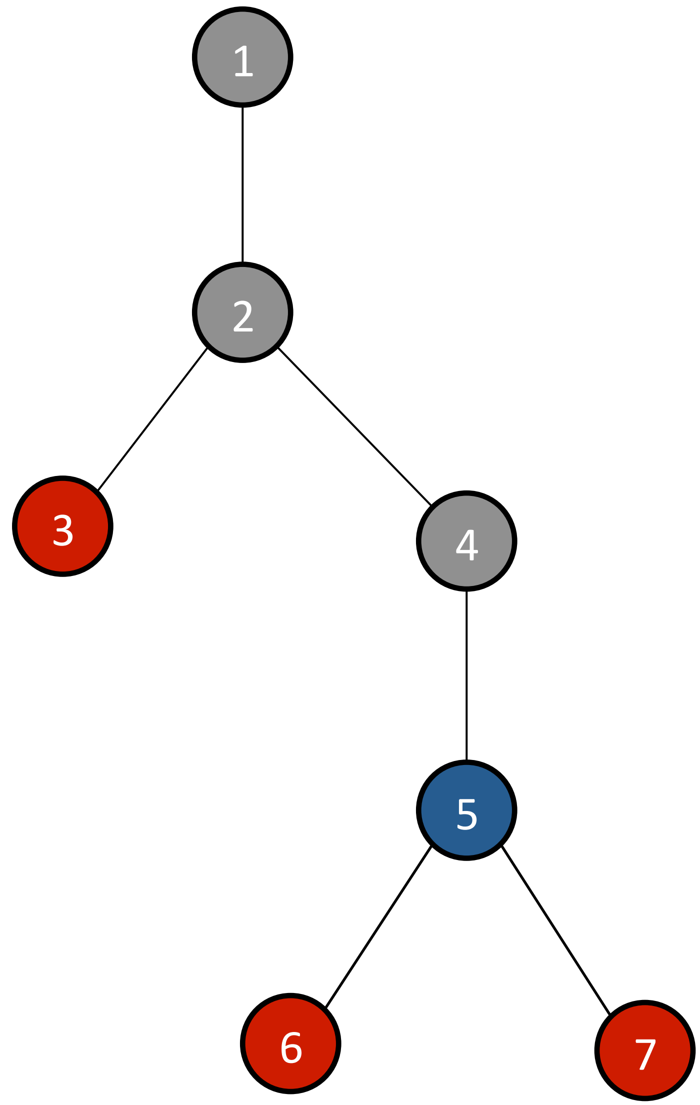

Push children



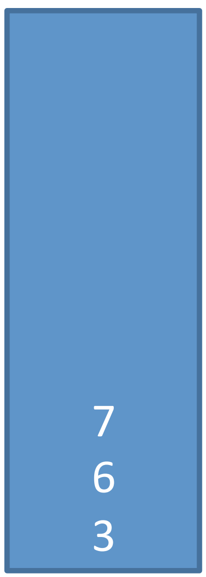
# Using a stack



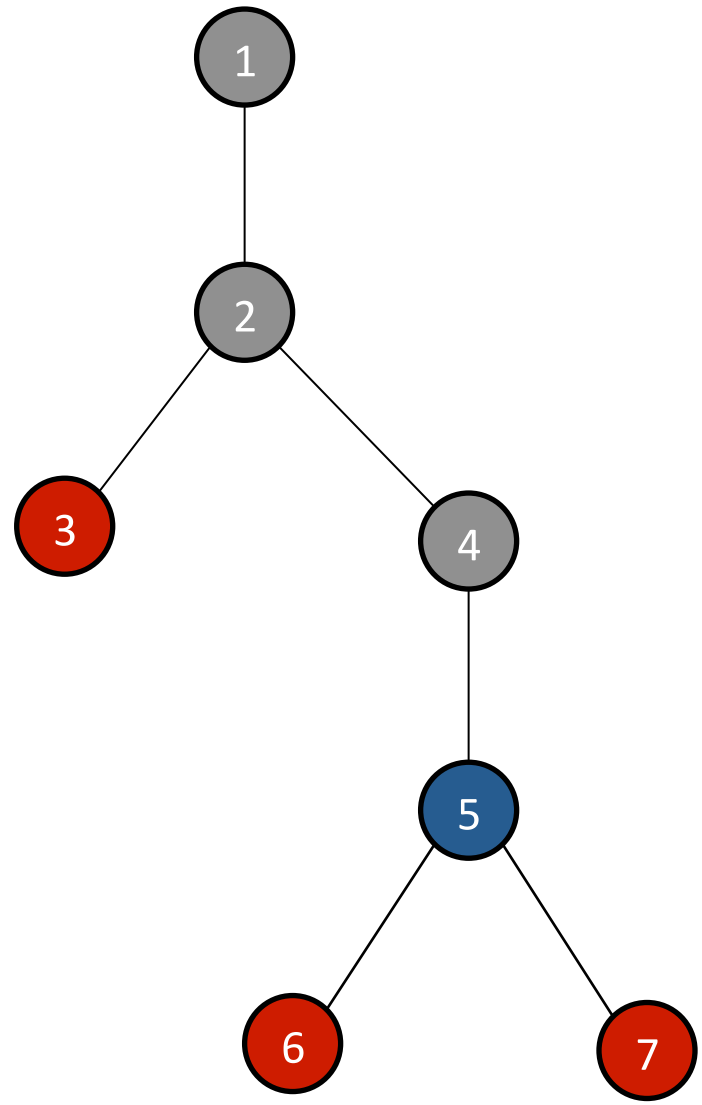

Pop node



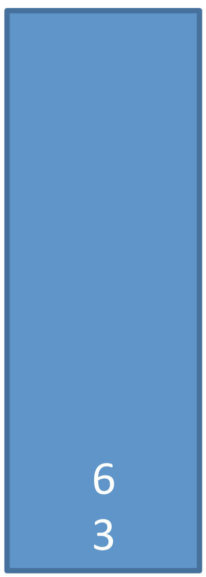
# Using a stack



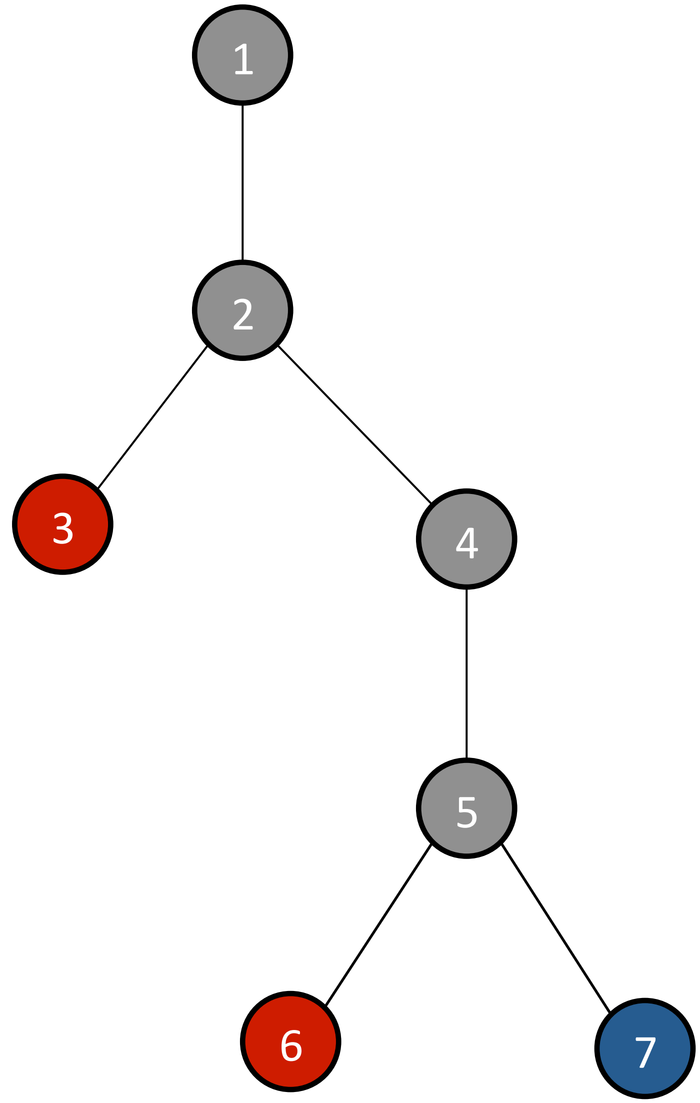

Push children



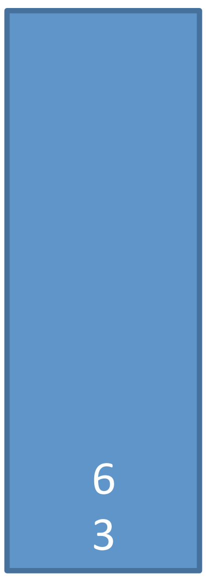
# Using a stack



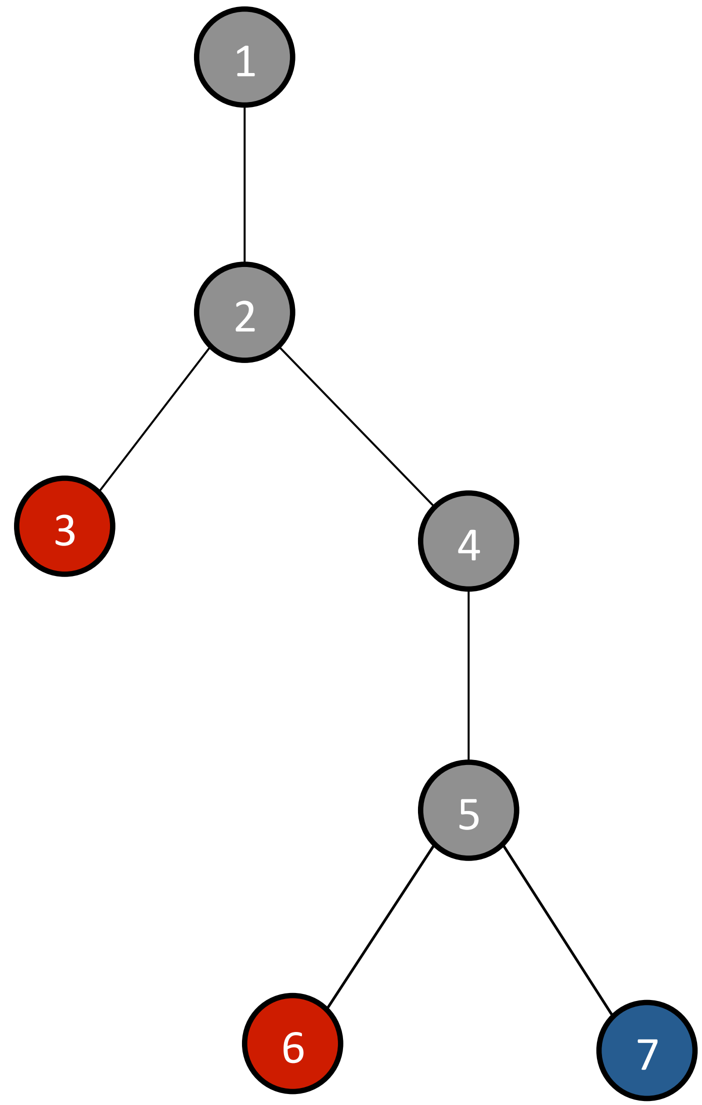

Pop node



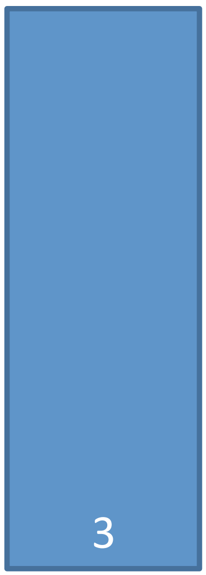
# Using a stack



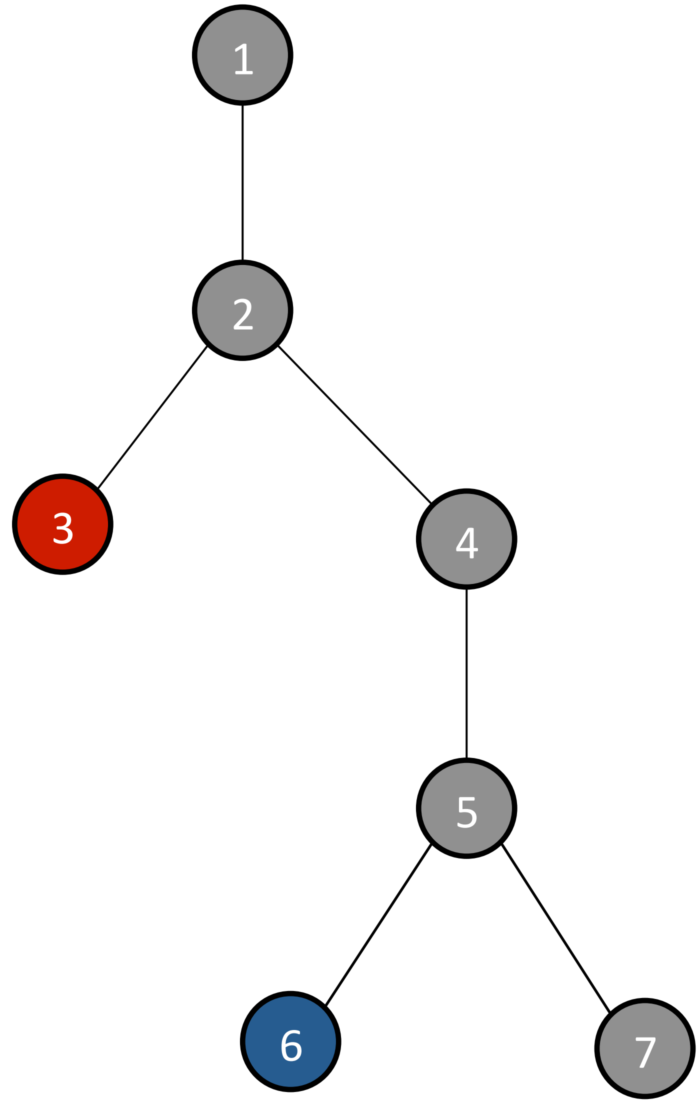

Push children (none)



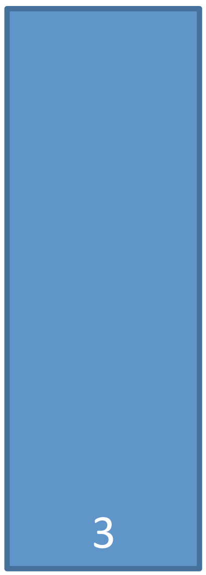
# Using a stack



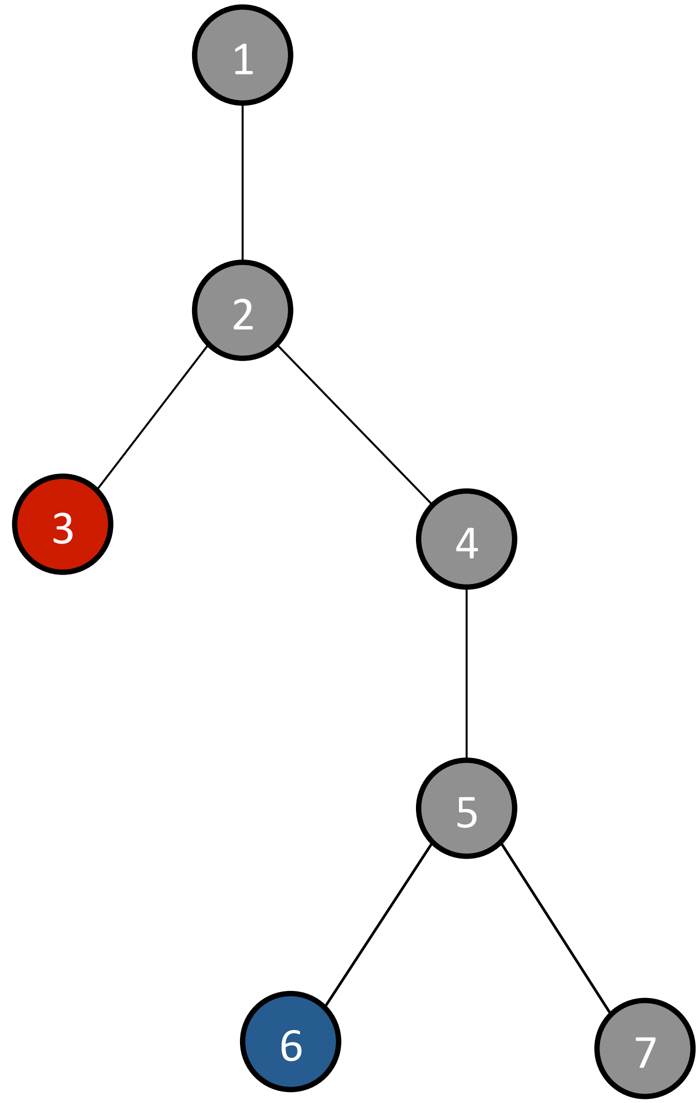

Pop node



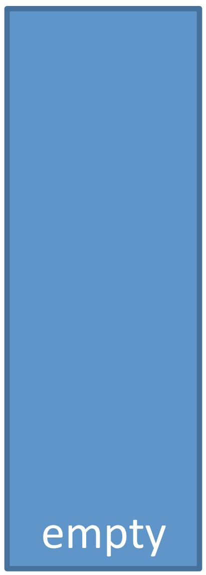
# Using a stack



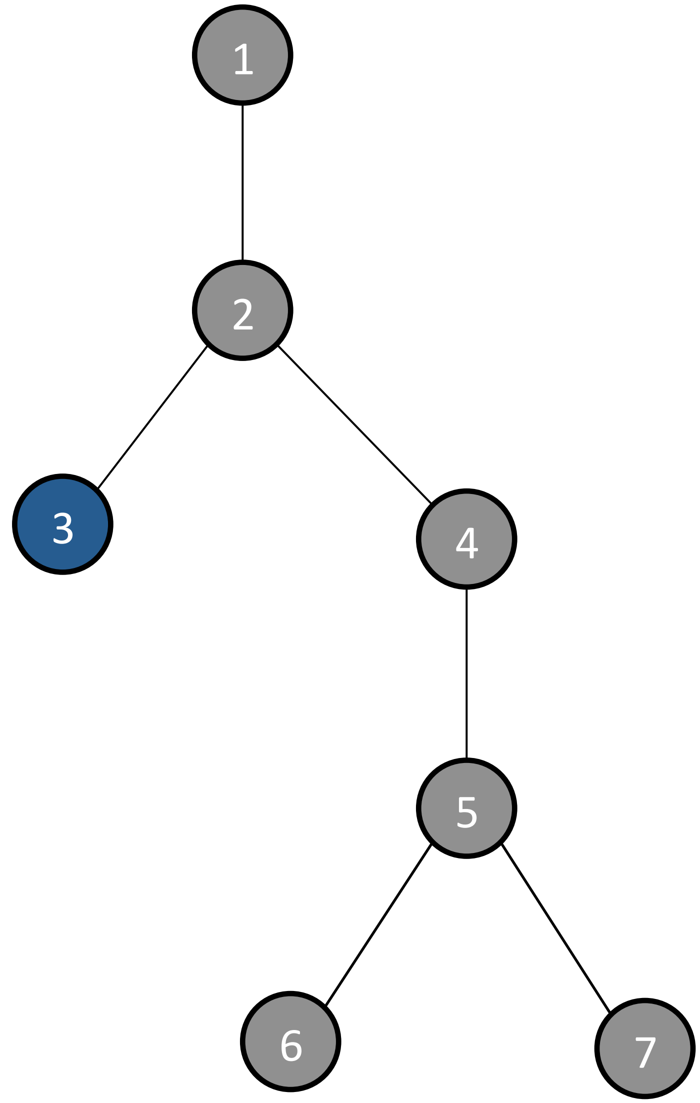

Push children (none)



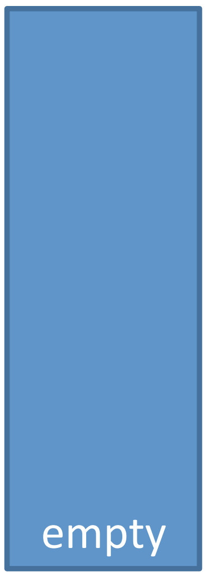
# Using a stack



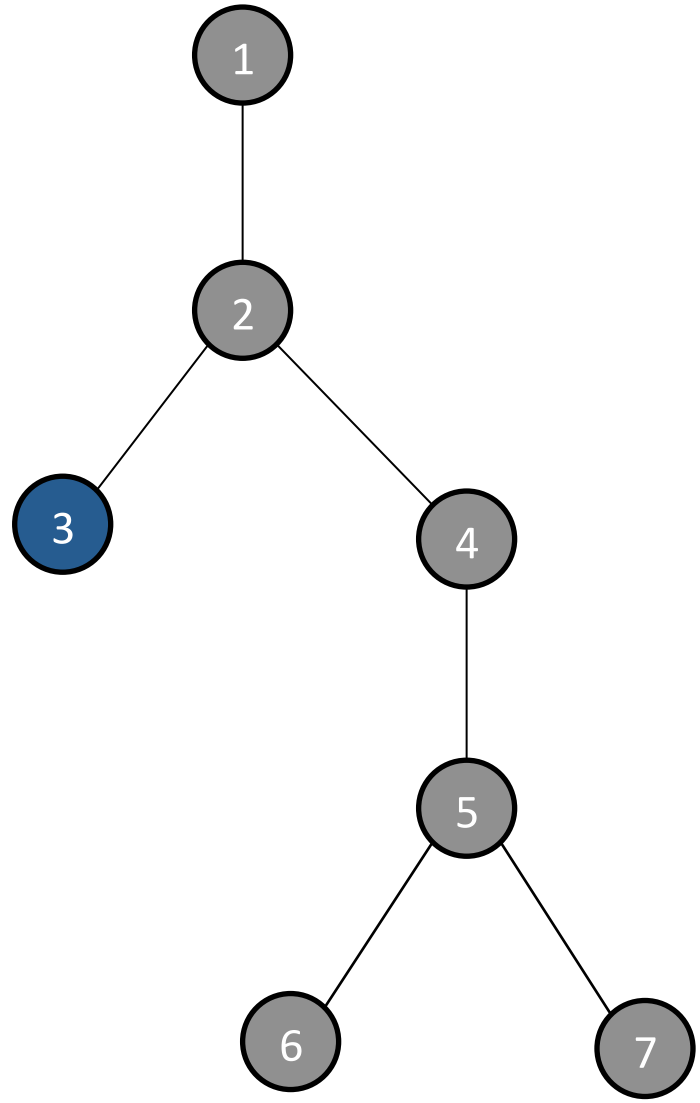

Pop node



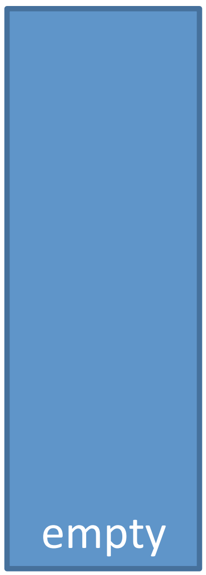
# Using a stack



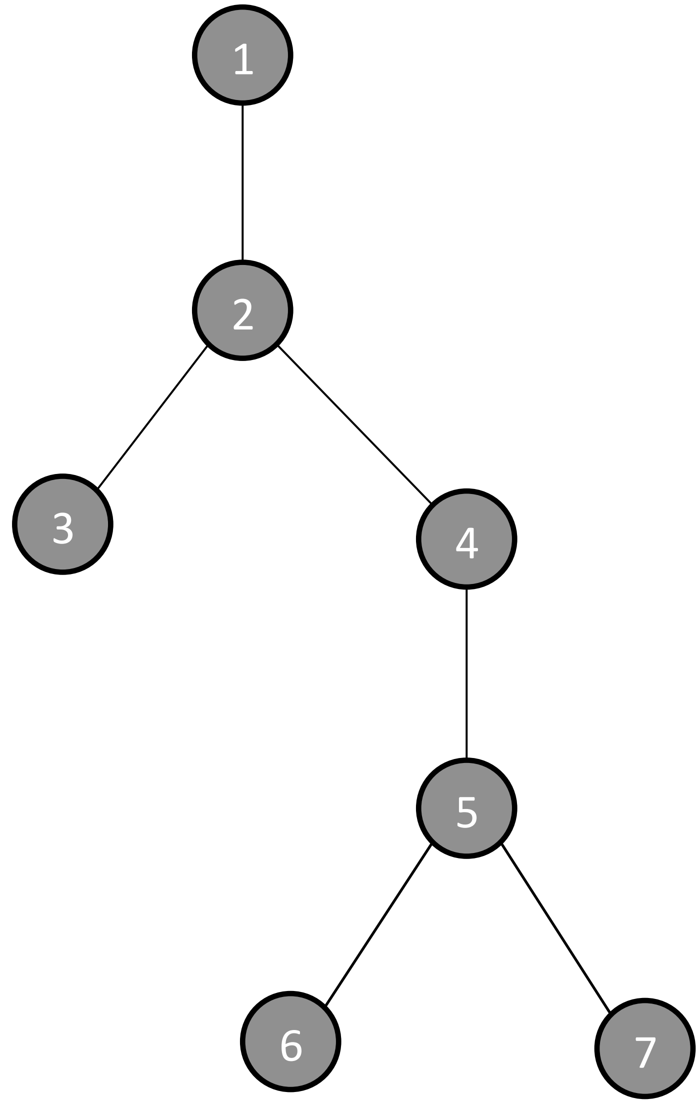

Push children (none)



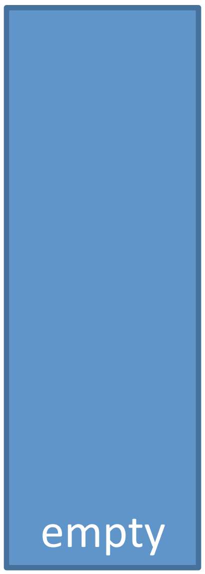
# Using a stack



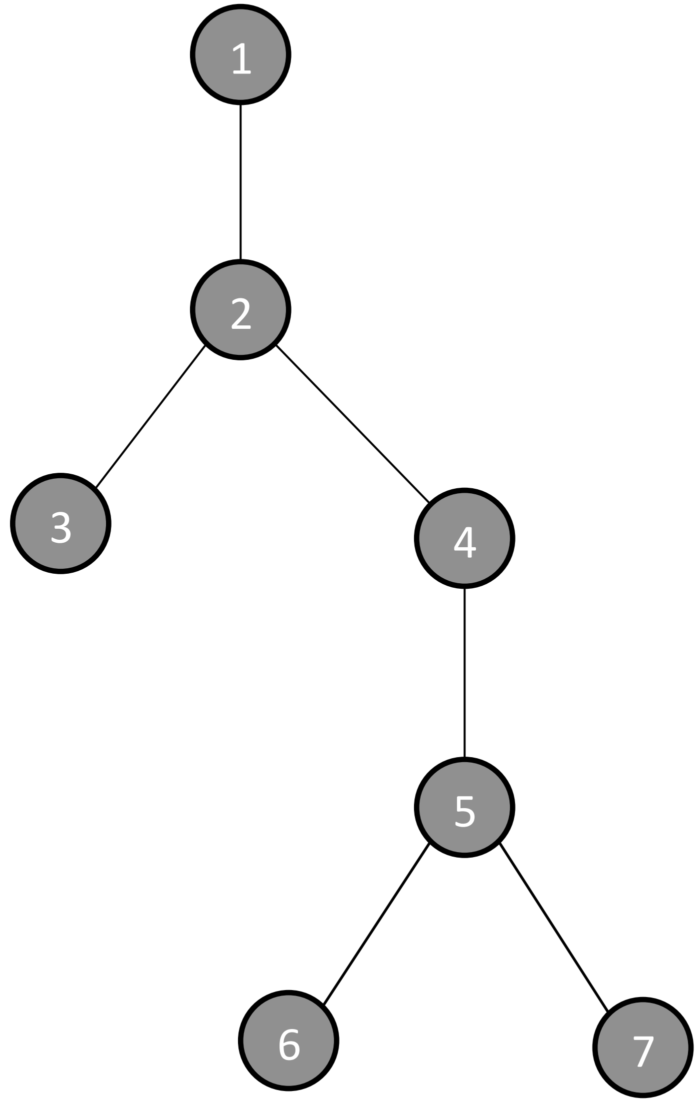

Pop node



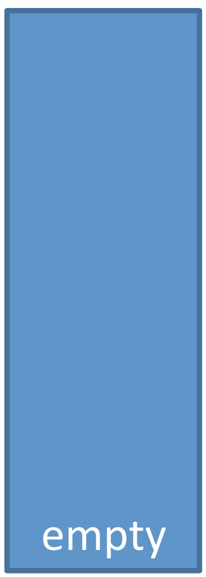
# Using a stack



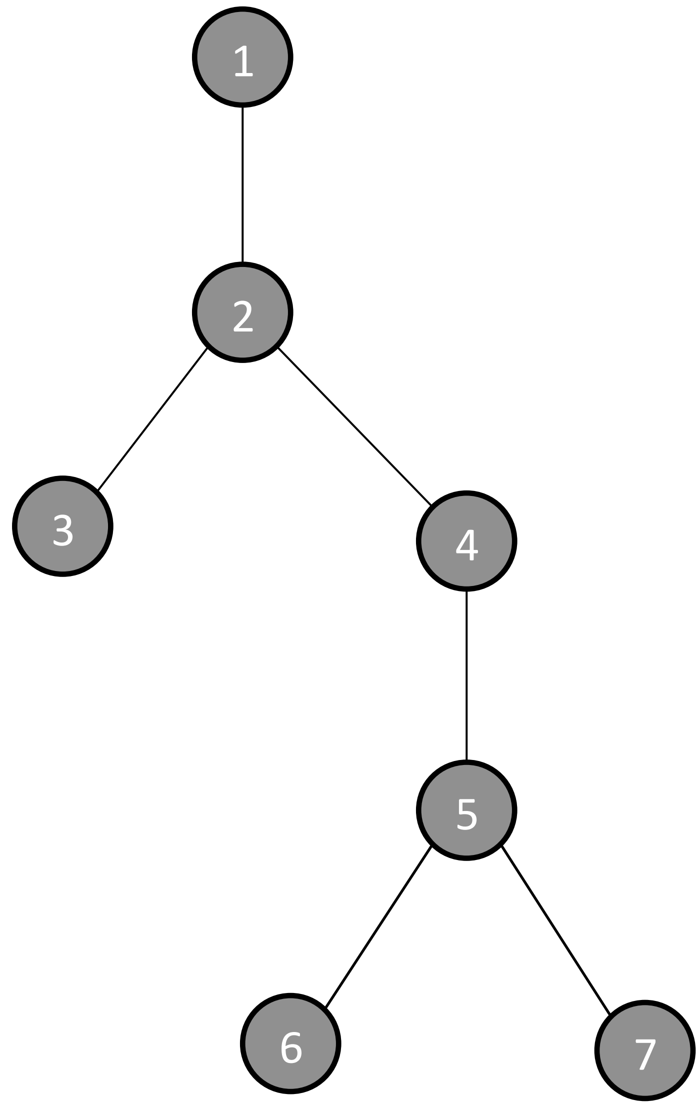

Pop node – stack empty



# Using a stack



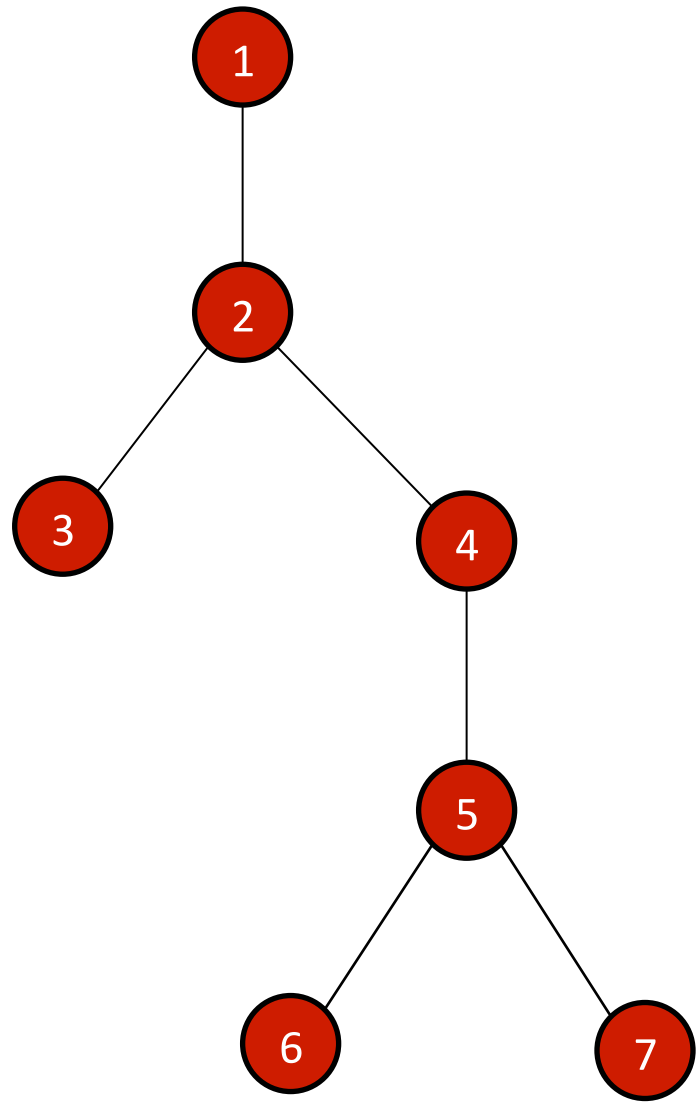

Finished



# Depth-first walk

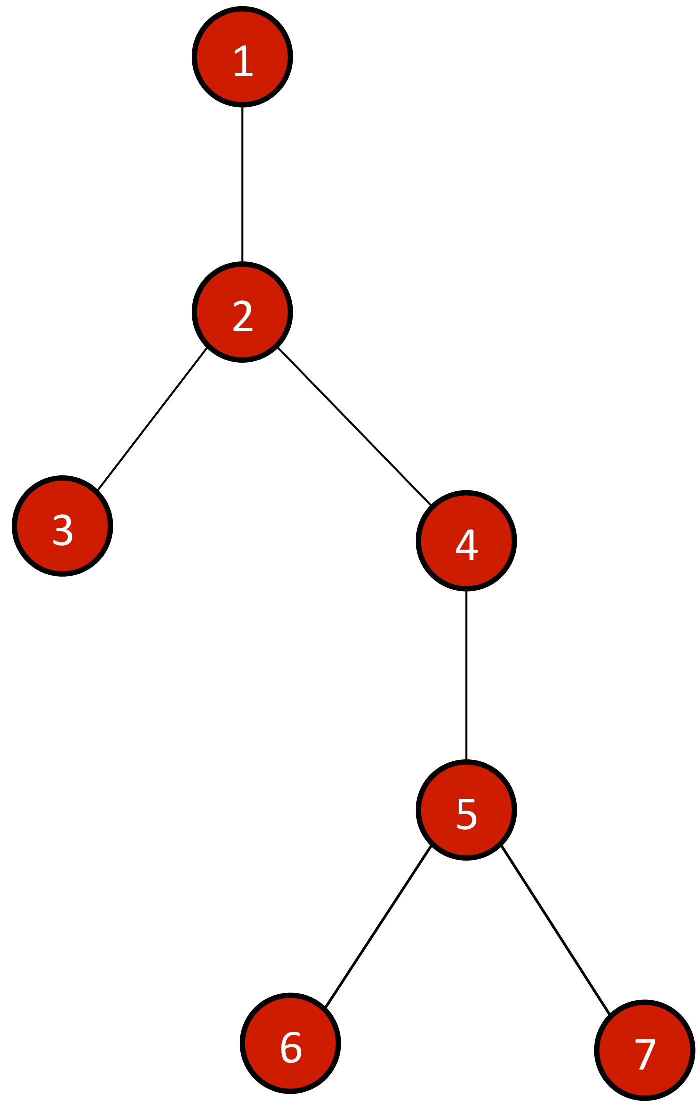
- **Substituting a stack** for a queue gives us depth-first traversal instead of breadth-first

- Although in this case, it visited the children right to left instead of left to right
  - If we cared, we could just push the children on the stack in reverse order



# Depth-first walk

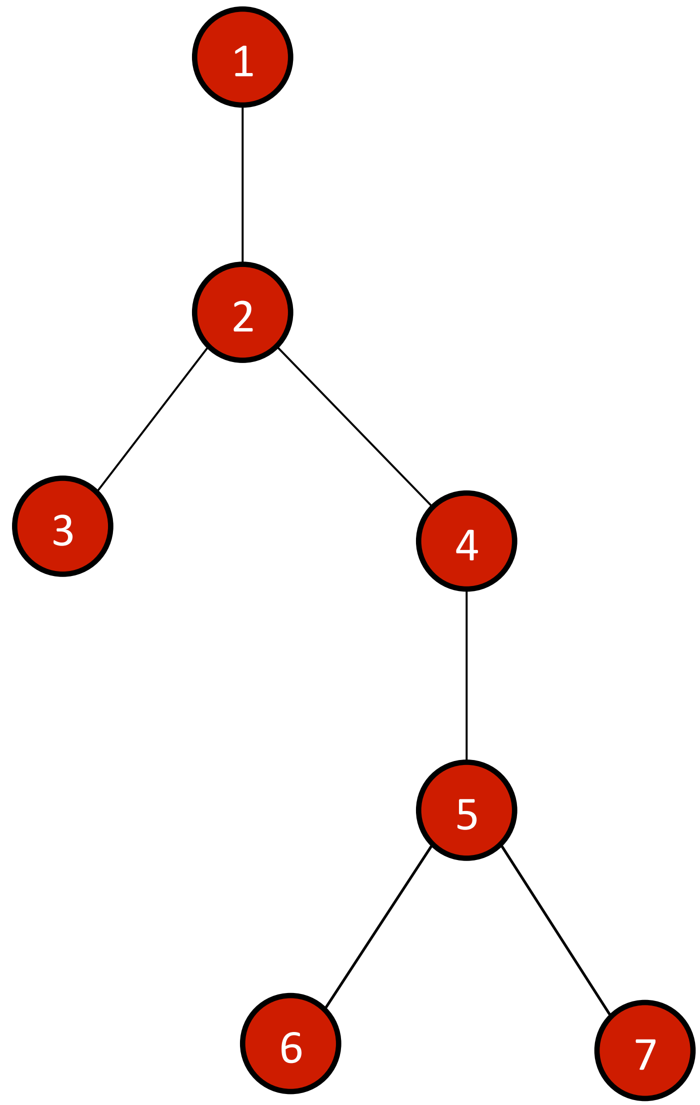
- Of course, we already have a perfectly good stack
- The **execution stack**
  - Every procedure call pushes the execution stack
  - Every return pops it
- Why not just use that stack?



# Recursive depth-first walk

Pseudocode:

```
DepthFirst(node)
  for each child c of node
    DepthFirst(c)
```

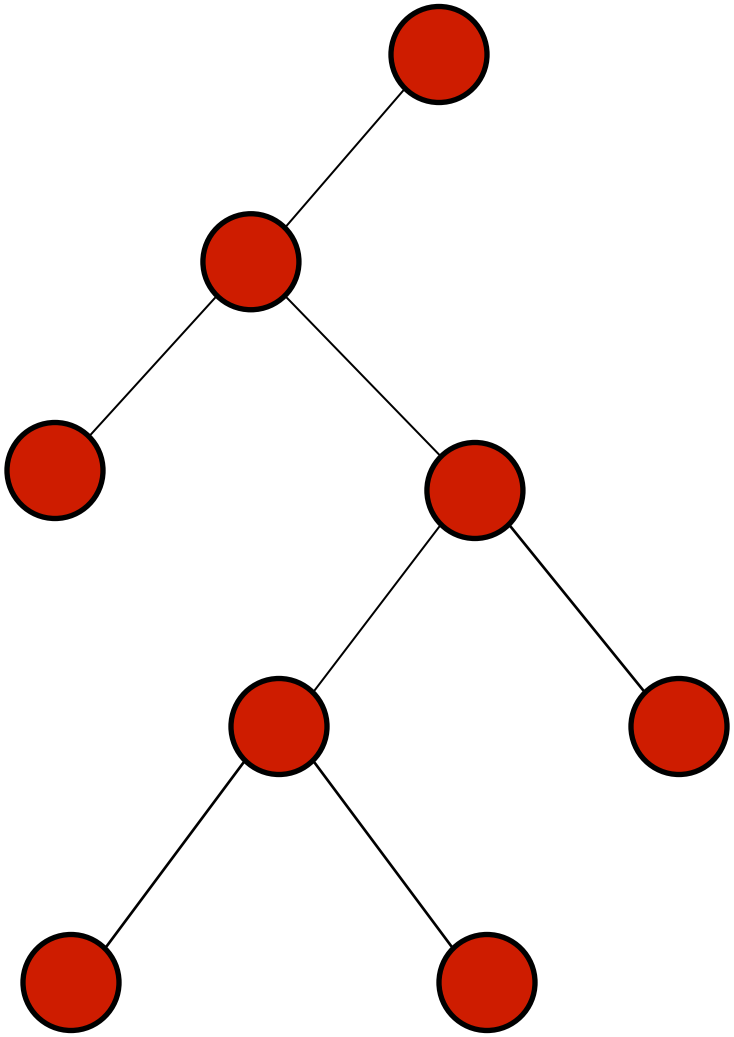



- Most of the time it's easier and clearer to write depth-first traversals as recursions
- And you thought recursion always made things more complicated!

# Specialized tree representations

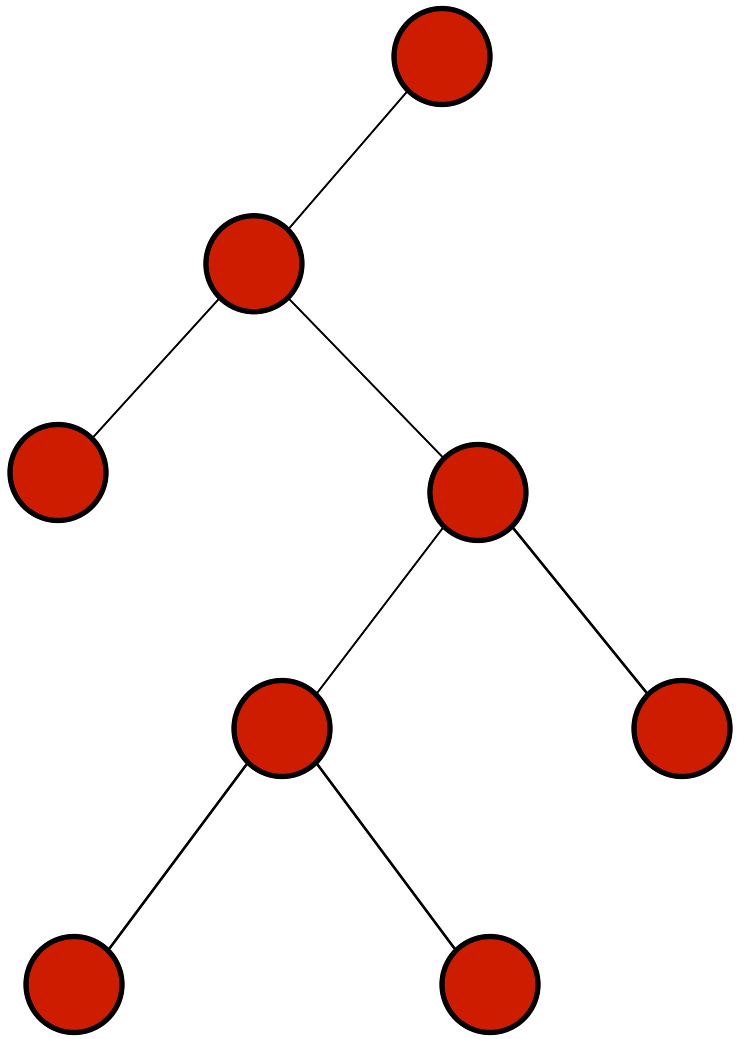
- Small, fixed branching factor
  - Put child pointers directly in nodes
  - Usually uses only for trees with a branching factor of 2 or 3
- Restricted information in nodes
  - No parent pointer
    - Used if you never need to find the parent of a node
    - Example: lists in lisp and scheme
  - No child pointers
    - Used in disjoint set representation
- Heaps
  - We'll talk about these later

# Binary trees



- Common case
- Fixed branching factor of 2
  - Every node has at most 2 children
  - Referred to as the left child and right child

# Optimized binary tree representation

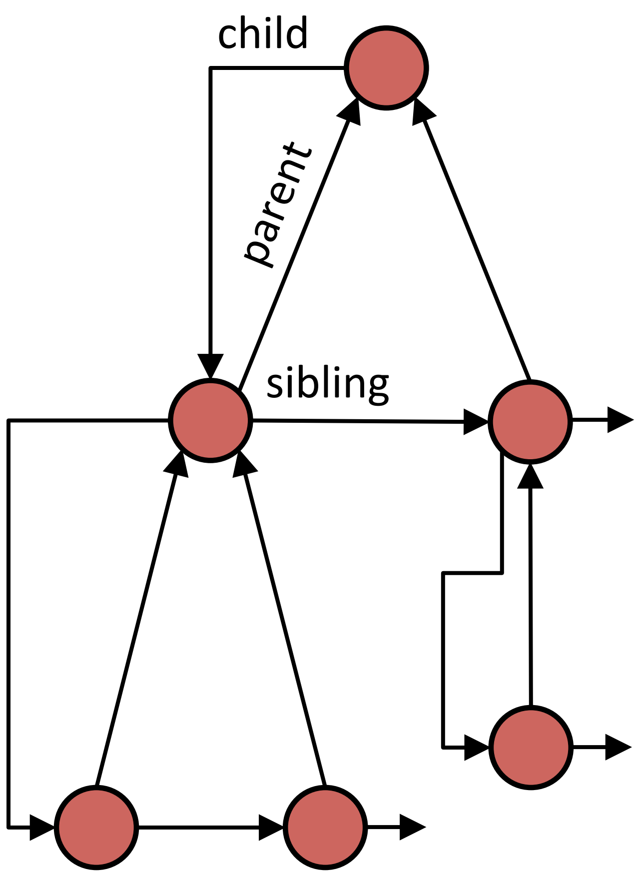


```
class BTNode {
   BTNode parent;
   BTNode leftChild;
   BTNode rightChild;
}
```

- Don’t even bother with an array or linked list
  - Just put pointers in the node for both children
  - Leave them null if they aren’t used

Why are binary trees popular?

# Left child/right sibling is technically a binary tree



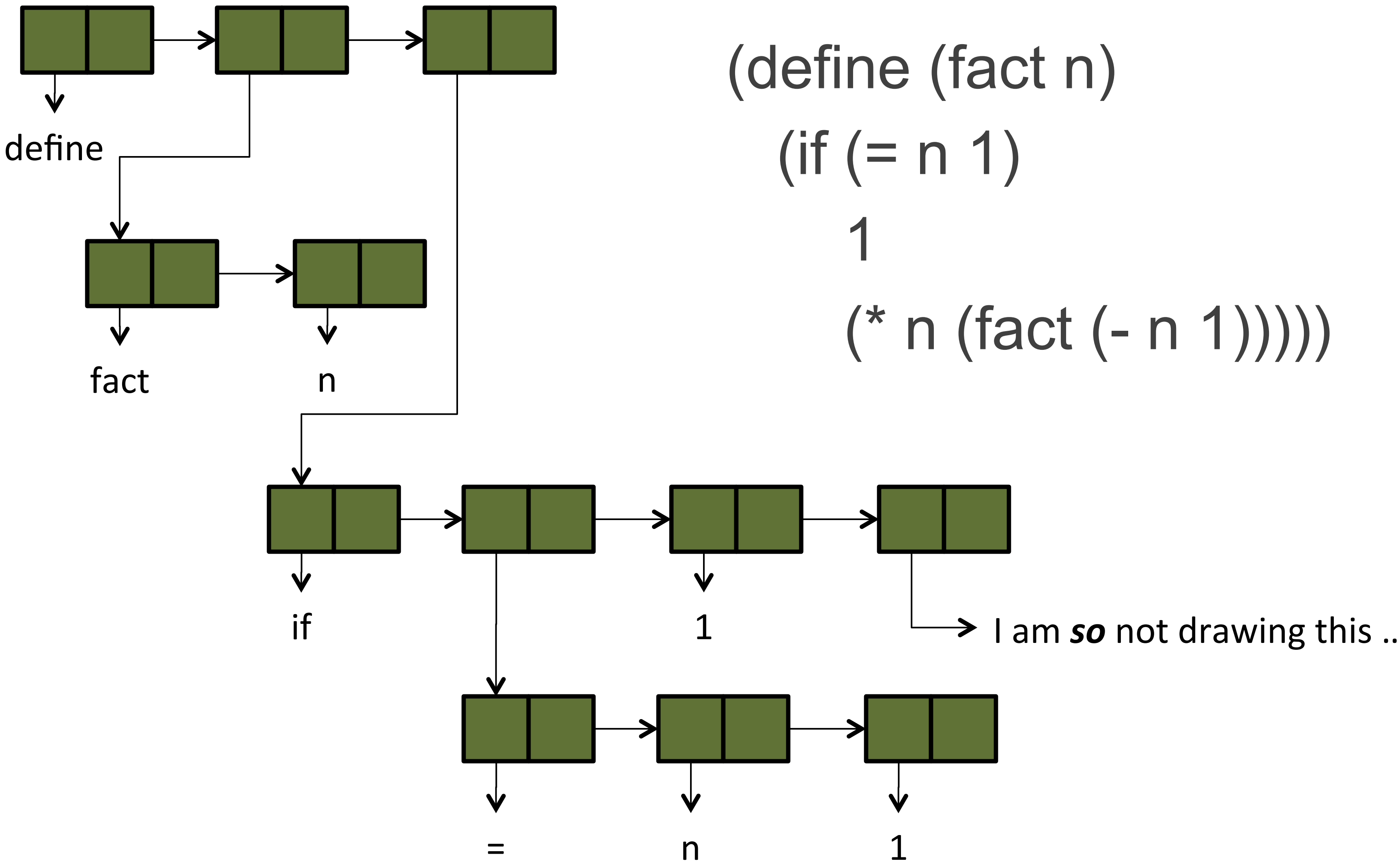

```
class TreeNode {
    TreeNode parent;
    TreeNode firstChild;   leftChild
    TreeNode nextSibling;  rightChild
    … other data …
}
```

The CLR book calls this the left child/right sibling representation

# Scheme lists are binary trees

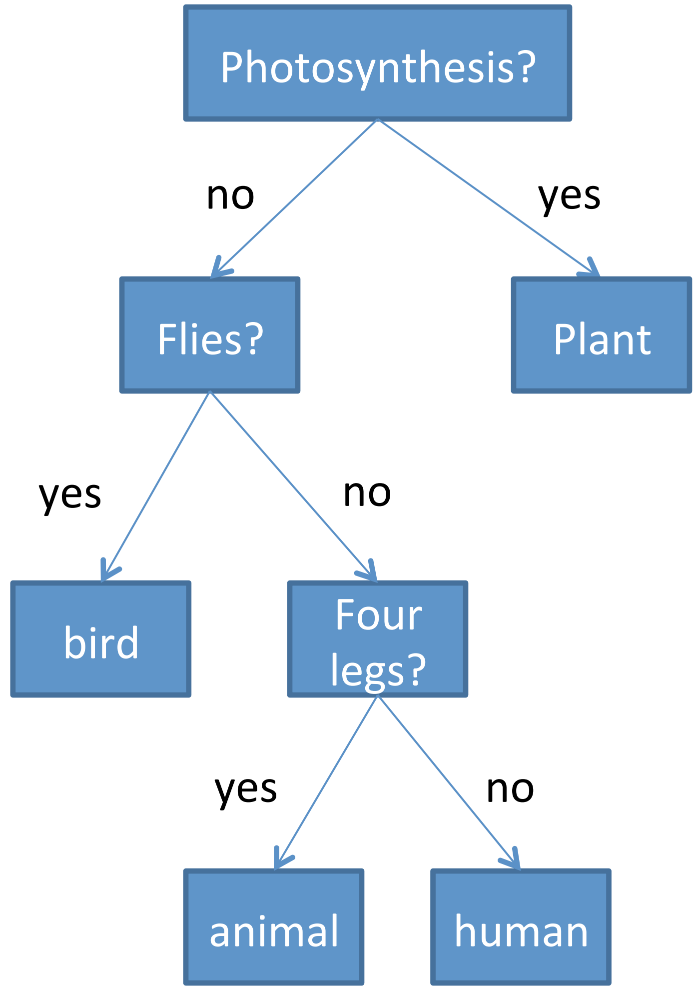
(without parent pointers)

```
(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```



# Decision trees

- A decision tree is a binary tree in which
  - The leaf nodes are decisions
  - The interior nodes are yes/no questions for deciding between decisions




A very bad decision tree for classifying lifeforms

# Depth-first traversals of binary trees

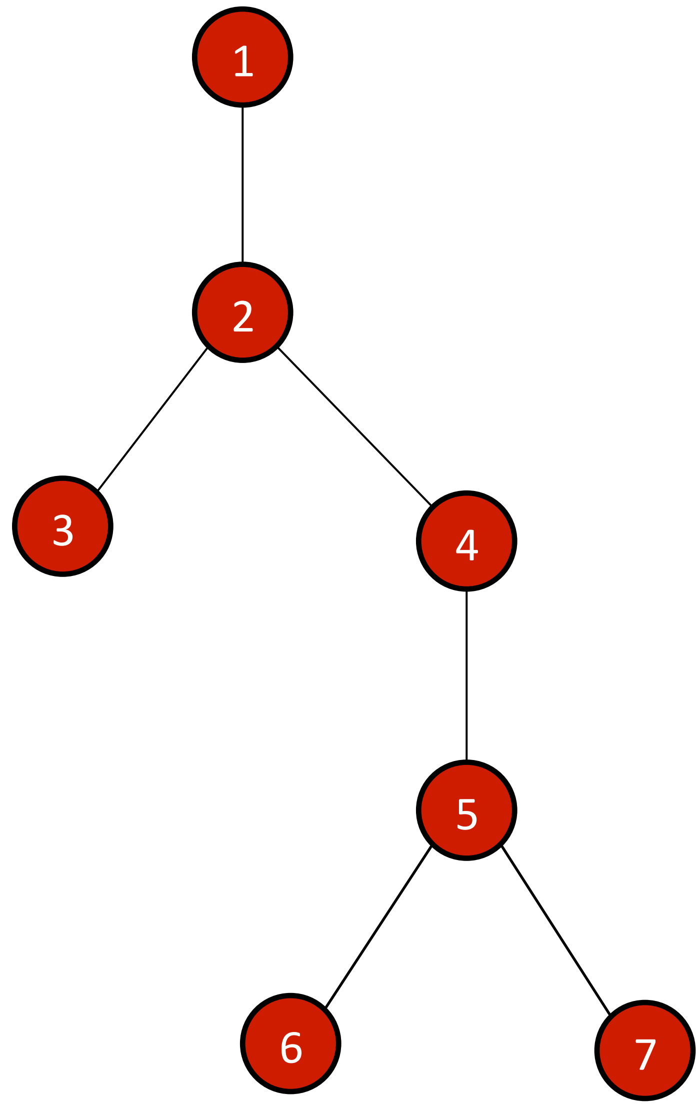
- Suppose you want to print all the nodes in a binary tree
  - You could use a depth-first traversal
  - But there are actually three different DFTs you could use
- We call these three versions the **preorder**, **postorder**, and **inorder** traversals of the tree

```
Preorder(node) {
  print node
  Preorder(node.leftChild)
  Preorder(node.rightChild)
}

Postorder(node) {
  Postorder(node.leftChild)
  Postorder(node.rightChild)
  print node
}

Inorder(node) {
  Inorder(node.leftChild)
  print node
  Inorder(node.rightChild)
}
```

# Preorder traversal

```
Preorder(node) {
   print node
   Preorder(node.leftChild)
   Preorder(node.rightChild)
}
```

Output:

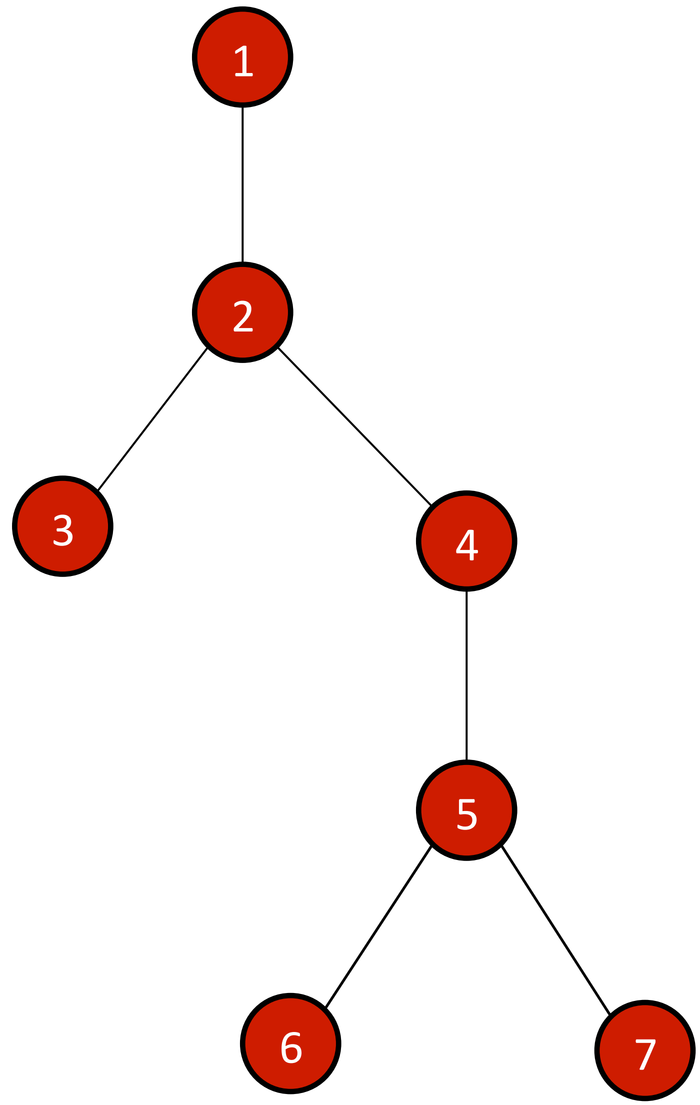


# Preorder traversal

```
Preorder(node) {
   print node
   Preorder(node.leftChild)
   Preorder(node.rightChild)
}
```

Output:

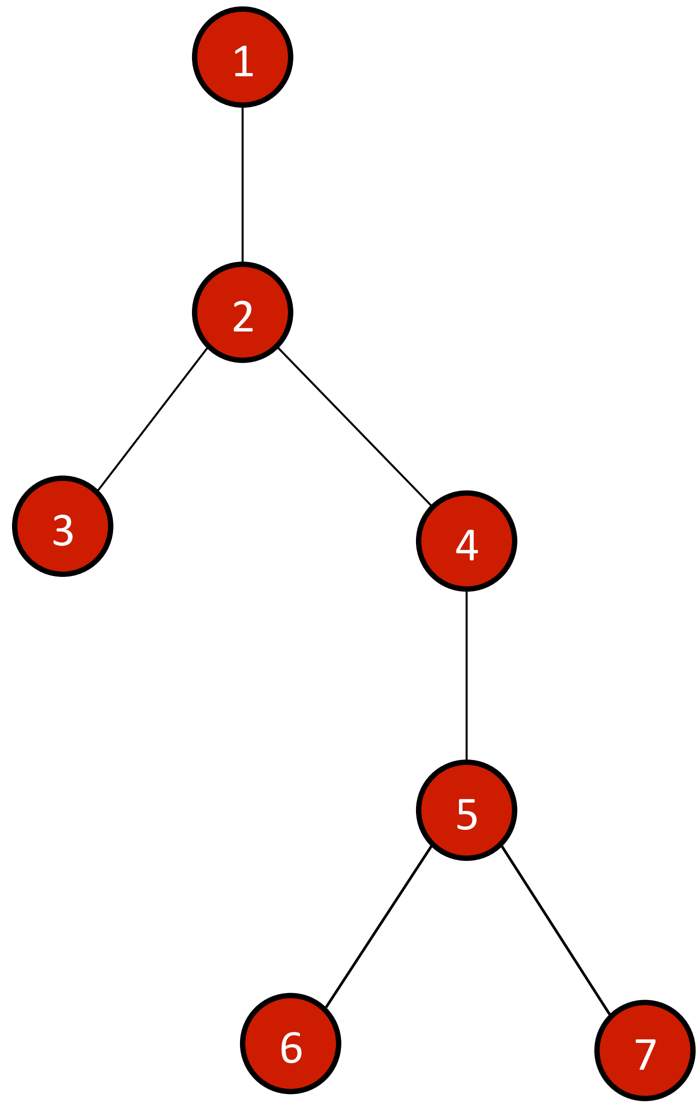
1 2 3 4 5 6 7



# Postorder traversal

```
Postorder(node) {
   Postorder(node.leftChild)
   Postorder(node.rightChild)
   print node
}
```

Output:

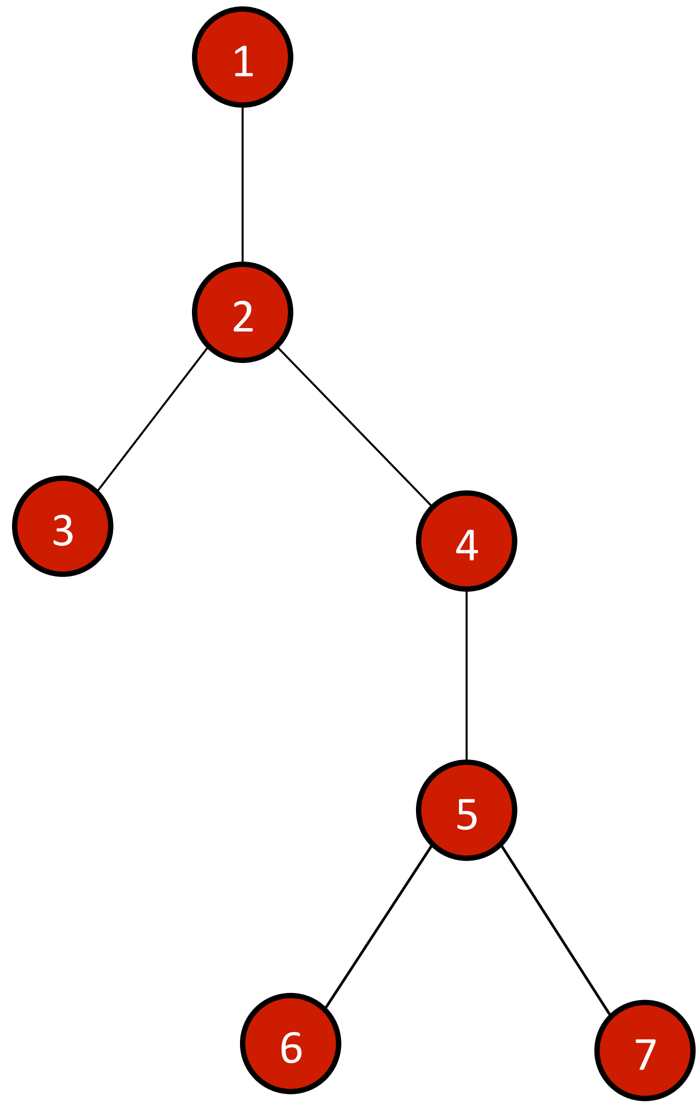


# Postorder traversal

```
Postorder(node) {
   Postorder(node.leftChild)
   Postorder(node.rightChild)
   print node
}
```

Output:

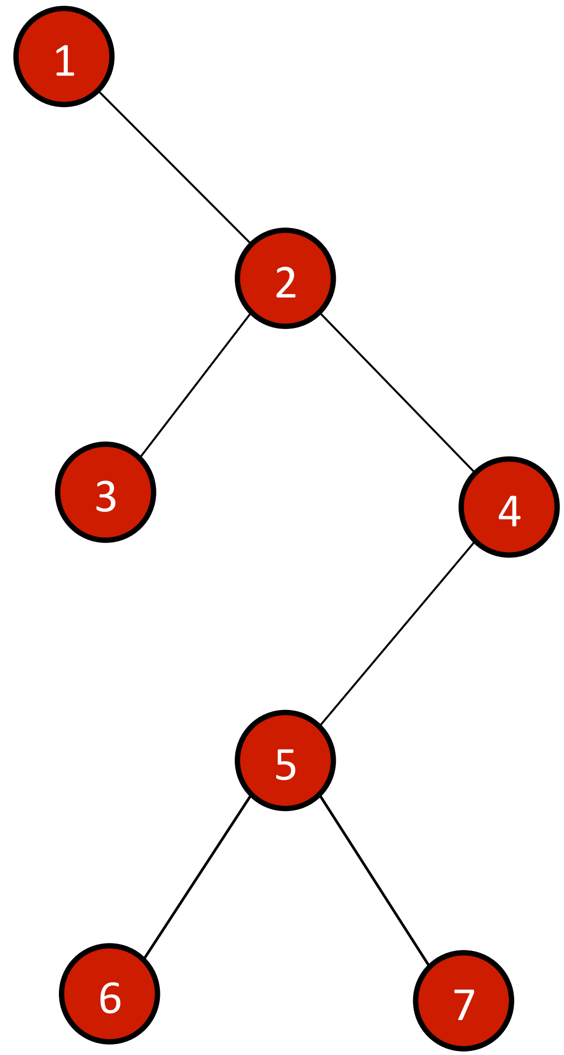
3 6 7 5 4 2 1



# Inorder traversal

```
Inorder(node) {
   Inorder(node.leftChild)
   print node
   Inorder(node.rightChild)
}
```

Output:

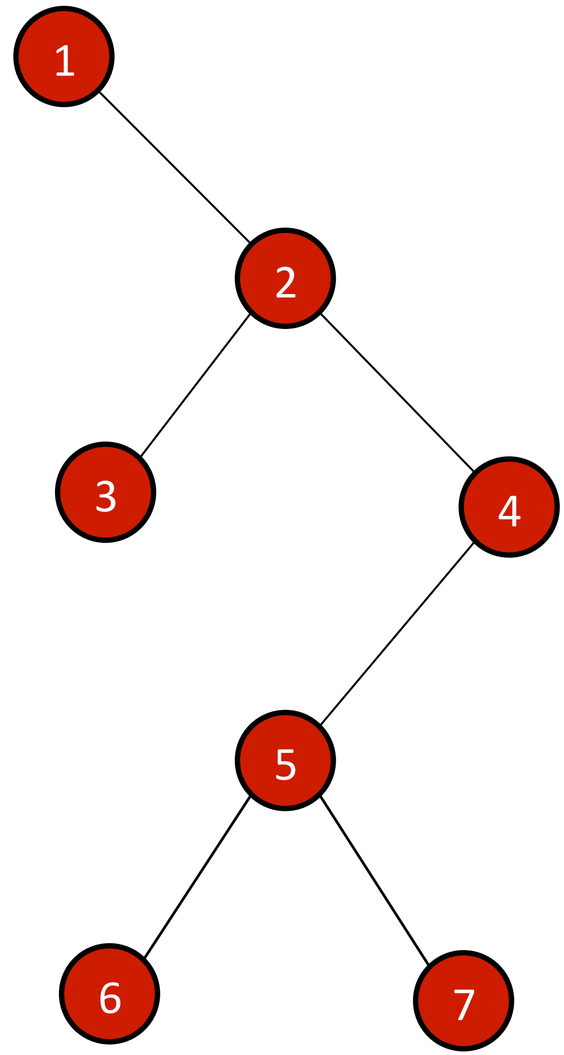


# Inorder traversal

```
Inorder(node) {
   Inorder(node.leftChild)
   print node
   Inorder(node.rightChild)
}
```

Output:

1 3 2 6 5 7 4



# Reading

- CLRS chapter 10, section 4 “Representing rooted trees”