CS 61BL

Lab 09

Announcements

- Project 2 released! You must lock in your partners today; make sure to add your partner on Beacon.

- Design Documents for Project 2 due Saturday, July 13 at 11:59pm.

- On Monday during lab, we will have Design Review Meetings, where your group will have a one-on-one conversation with us about your design.

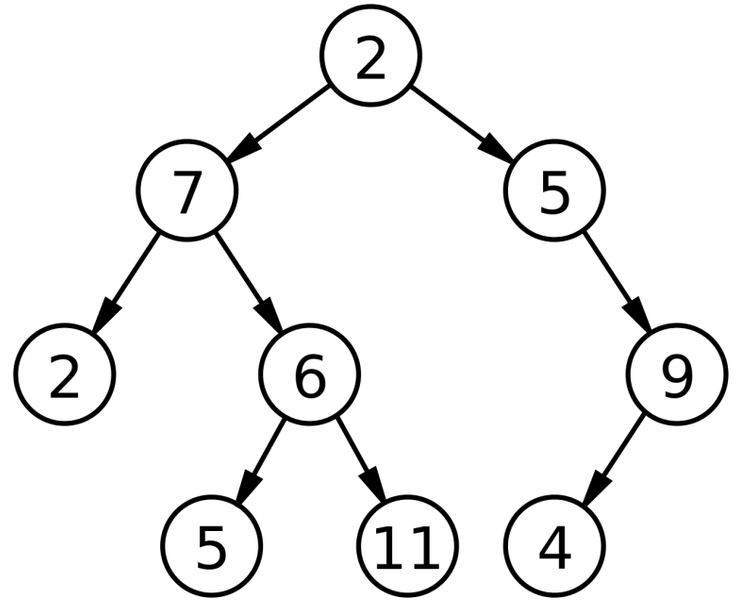
    - Sign up for a timeslot at rpurp.com/signup/ (only one partner should sign up)

Visiting Nodes

- How do we visit all of the nodes in a tree?

- ("Visit" = process the node in some way, for example, printing its value or doubling it)

- Recursion seems like a reasonable choice: after we call our visit method on the whole tree, we would want to call it on the left-and-right subtree.

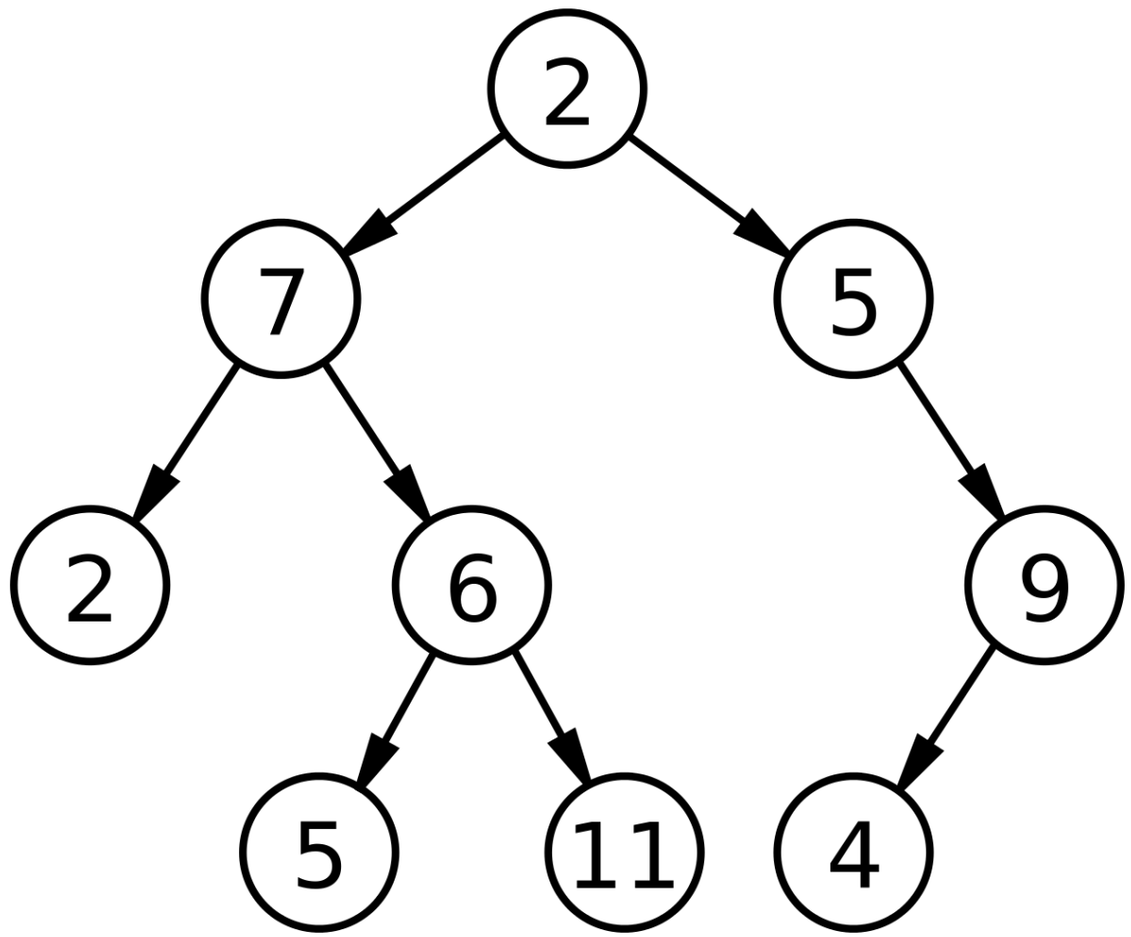


Pre-order Traversal

- Idea: process the current node, and then traverse the left subtree, and then traverse the right subtree.

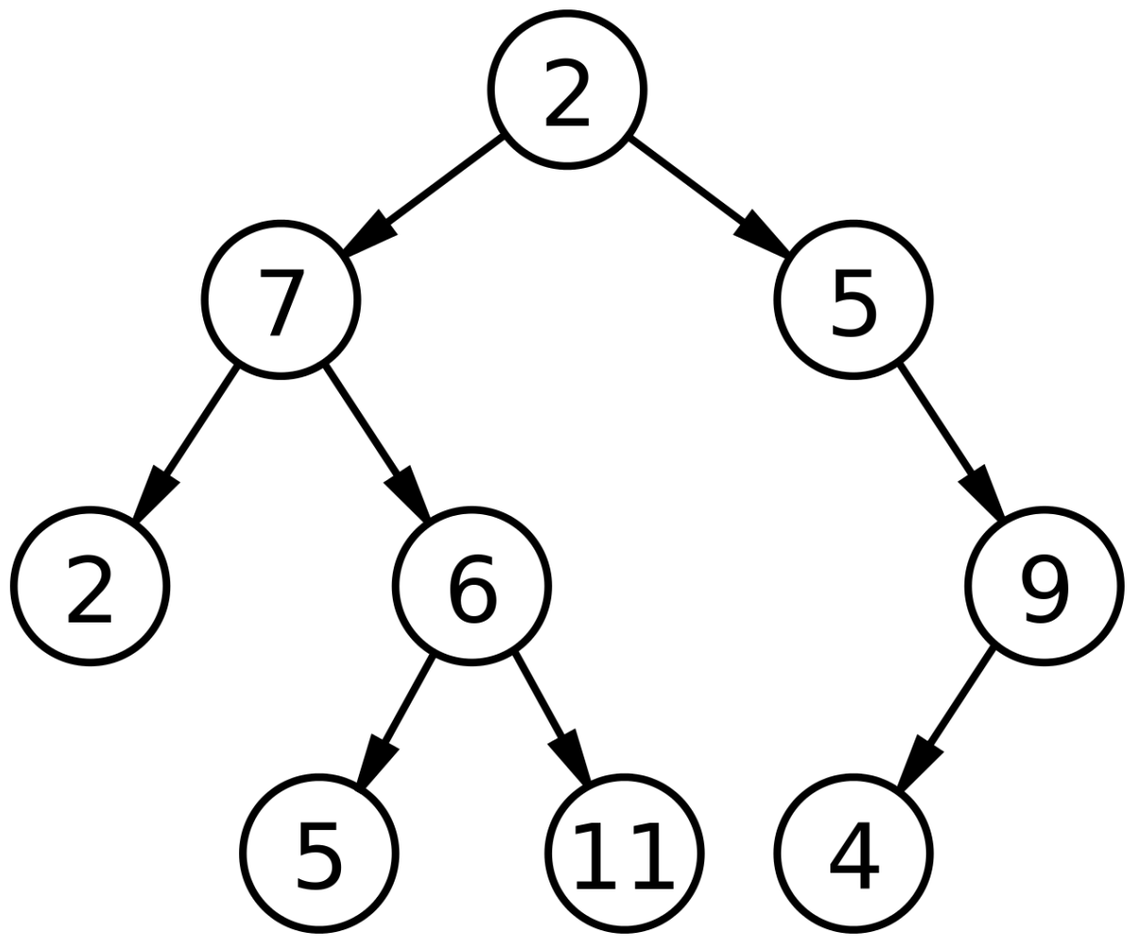
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```



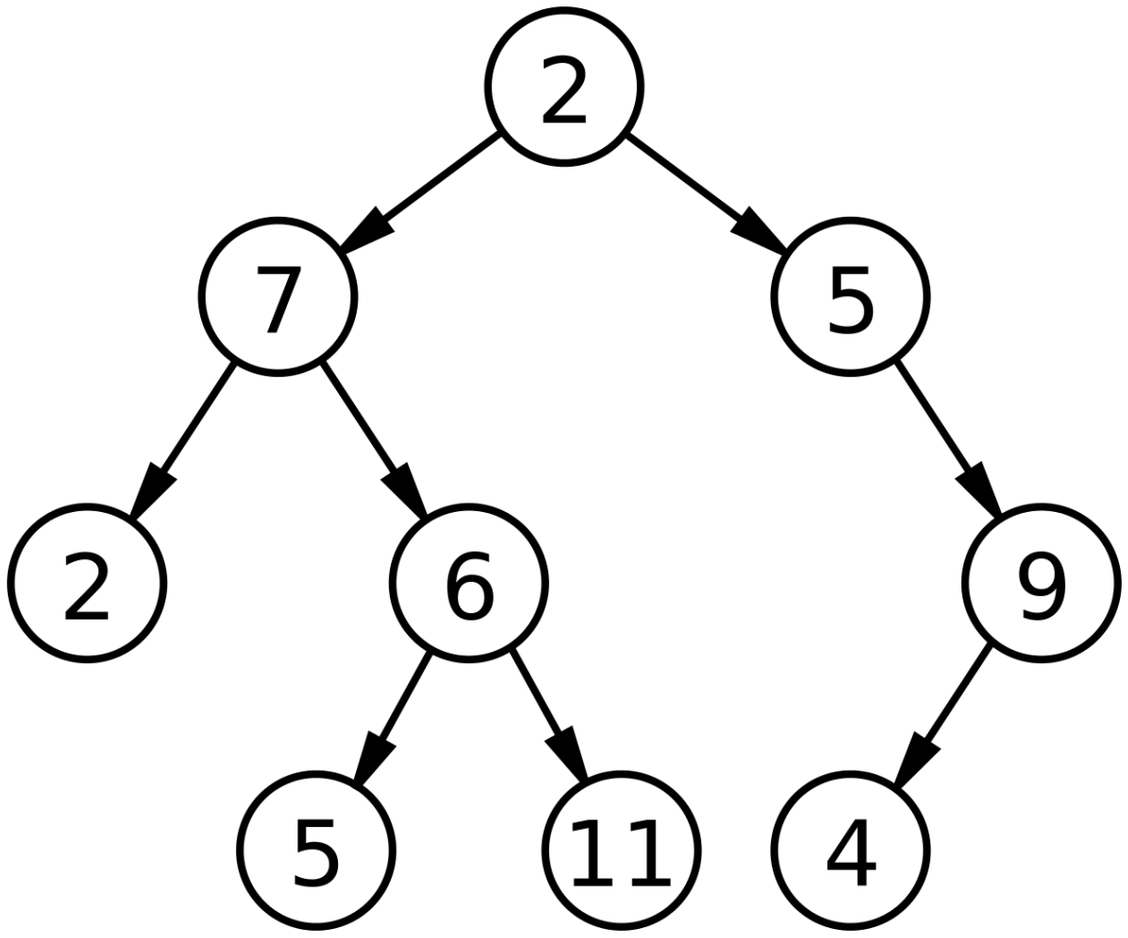
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

2



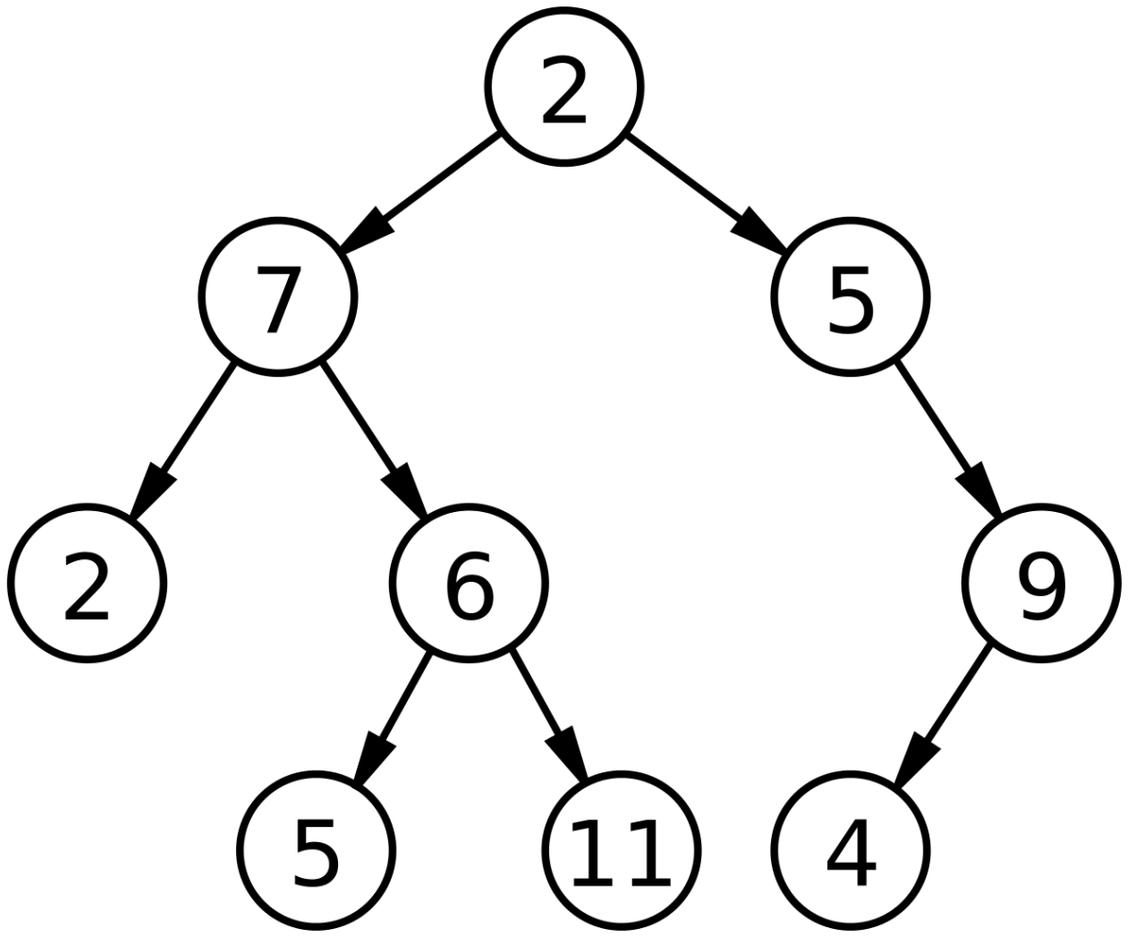
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

2
7



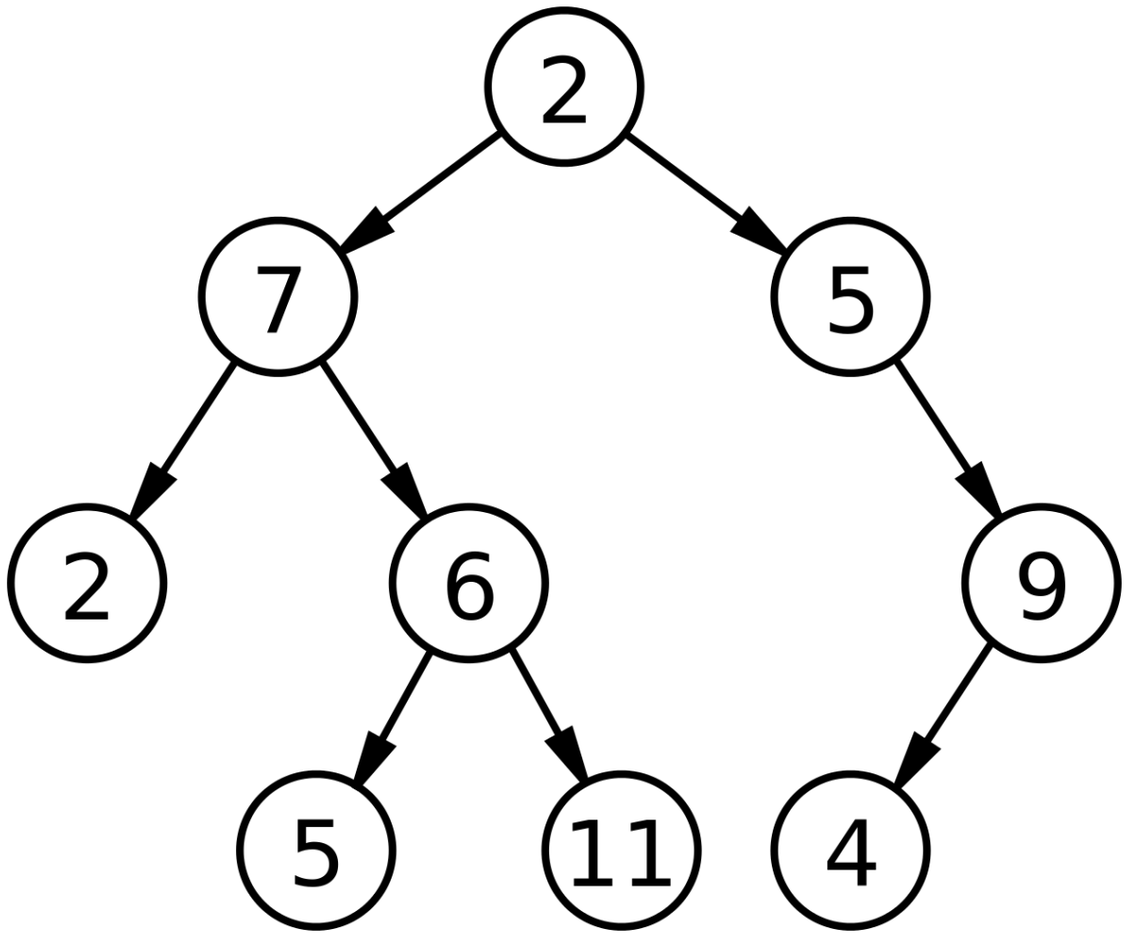
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

2
7
2



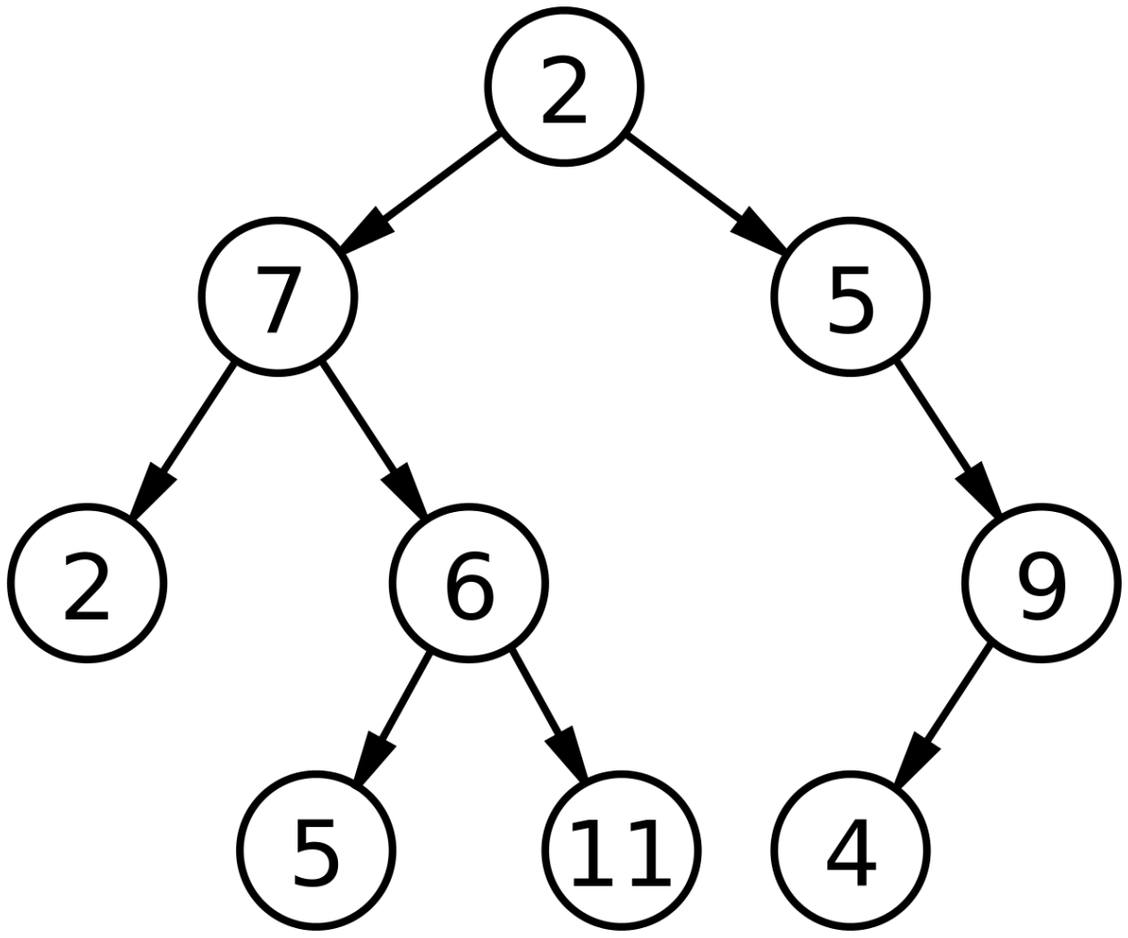
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

2
7
2
6



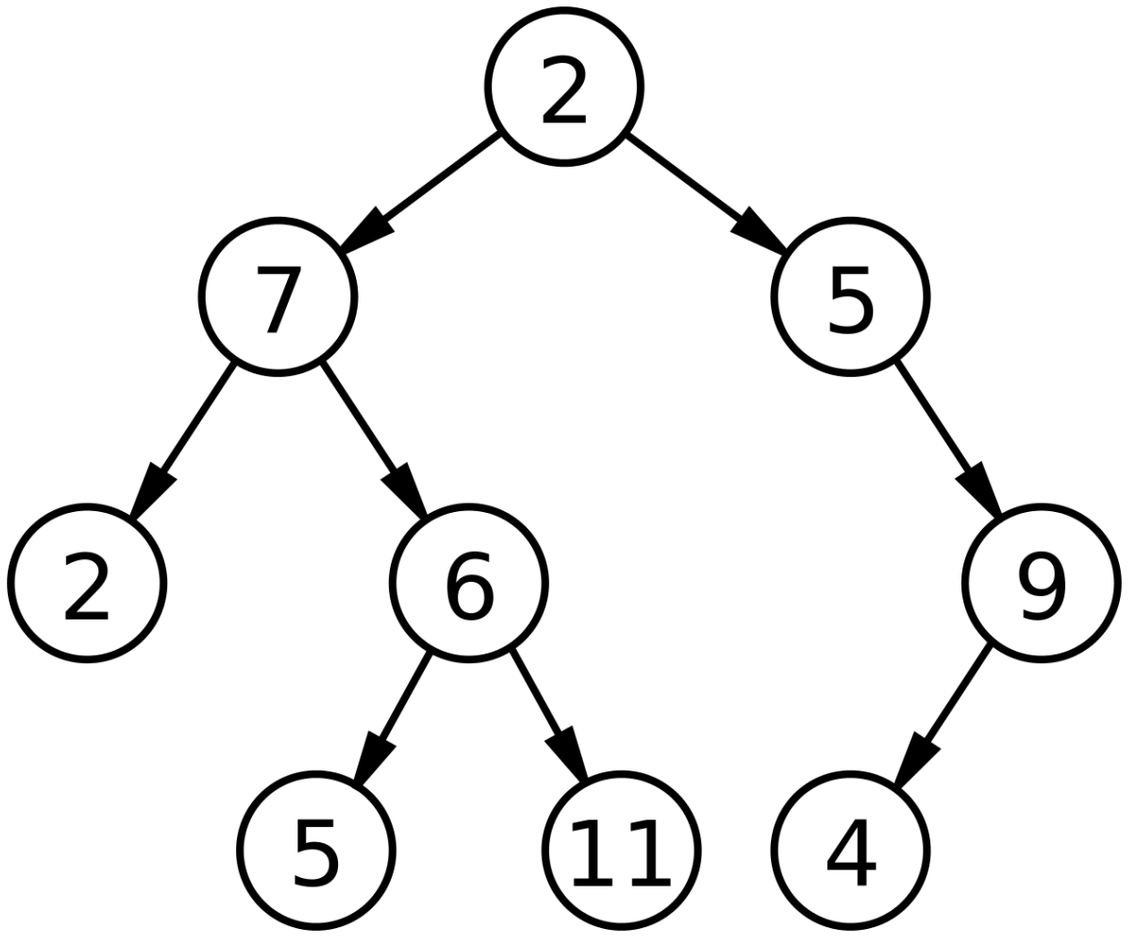
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

2
7
2
6
5



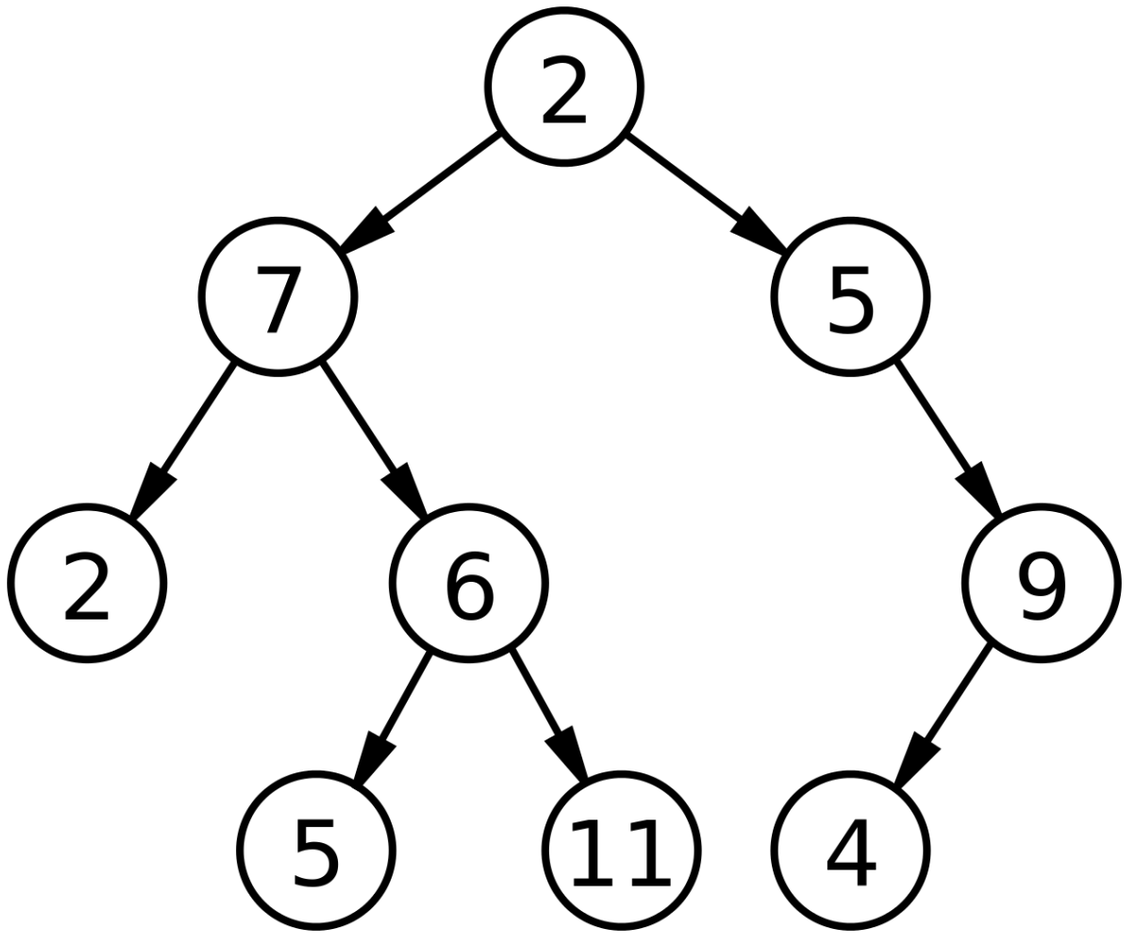
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

```
2
7
2
6
5
11
```



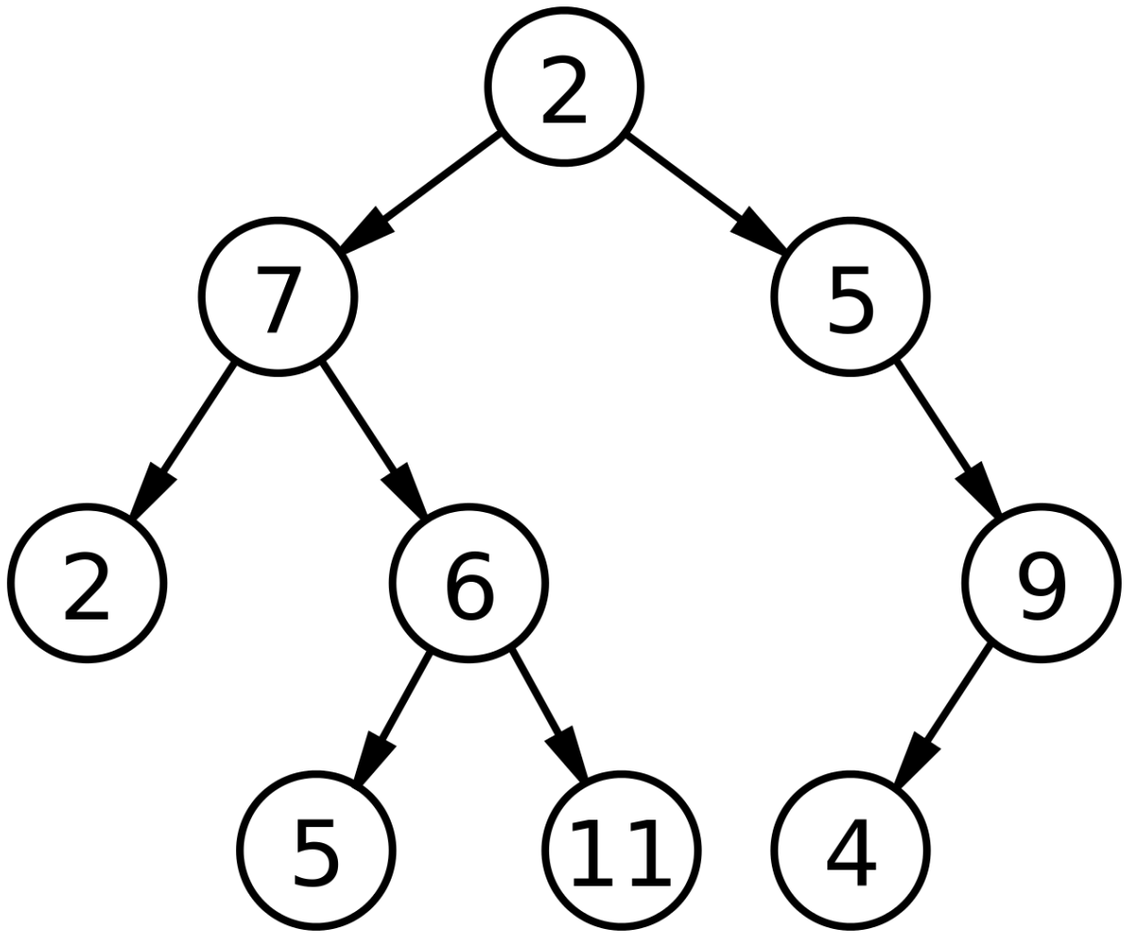
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

```
2
7
2
6
5
11
5
```



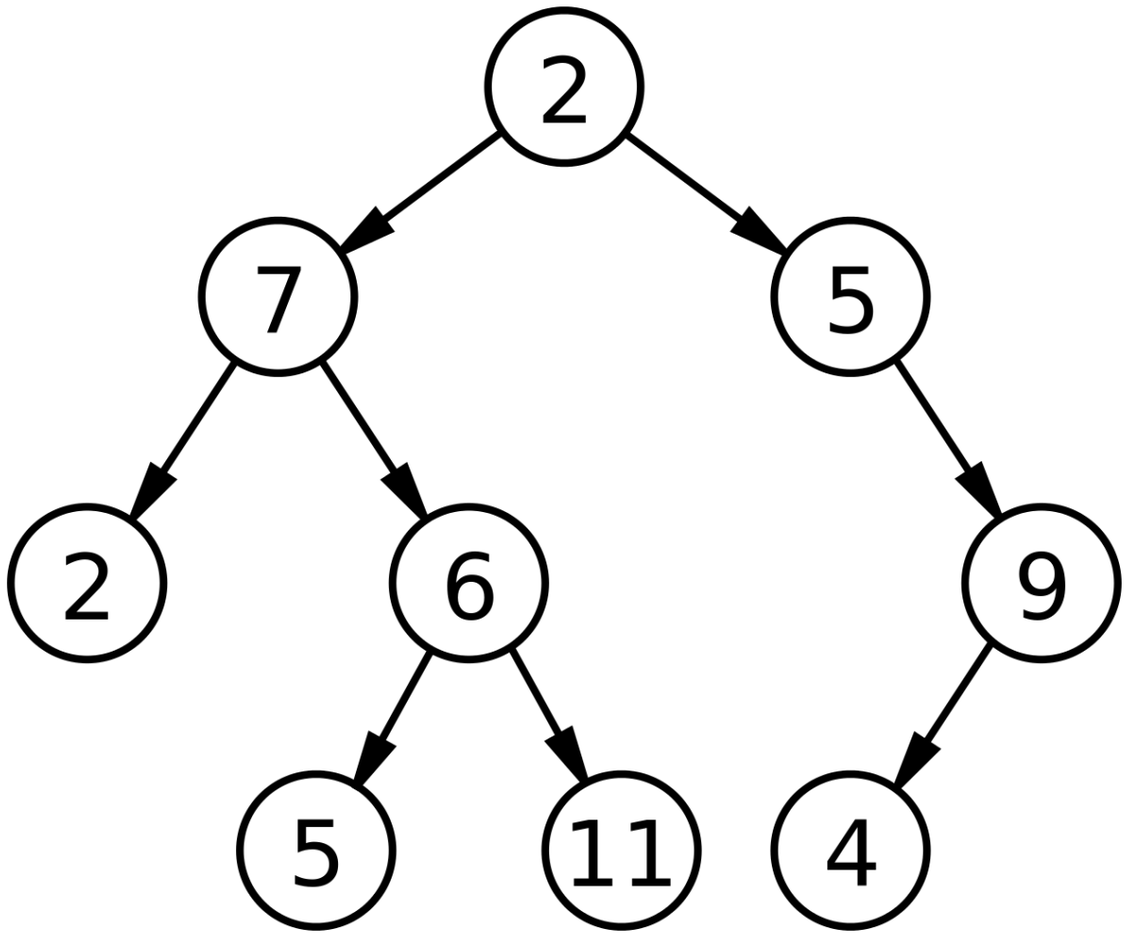
```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

```
2
7
2
6
5
11
5
9
```



```
function traverse_preorder(node):
        visit the current node
        if node.left != null:
            traverse_preorder(node.left)
        if node.right != null:
            traverse_preorder(node.right)
```

```
2
7
2
6
5
11
5
9
4
```

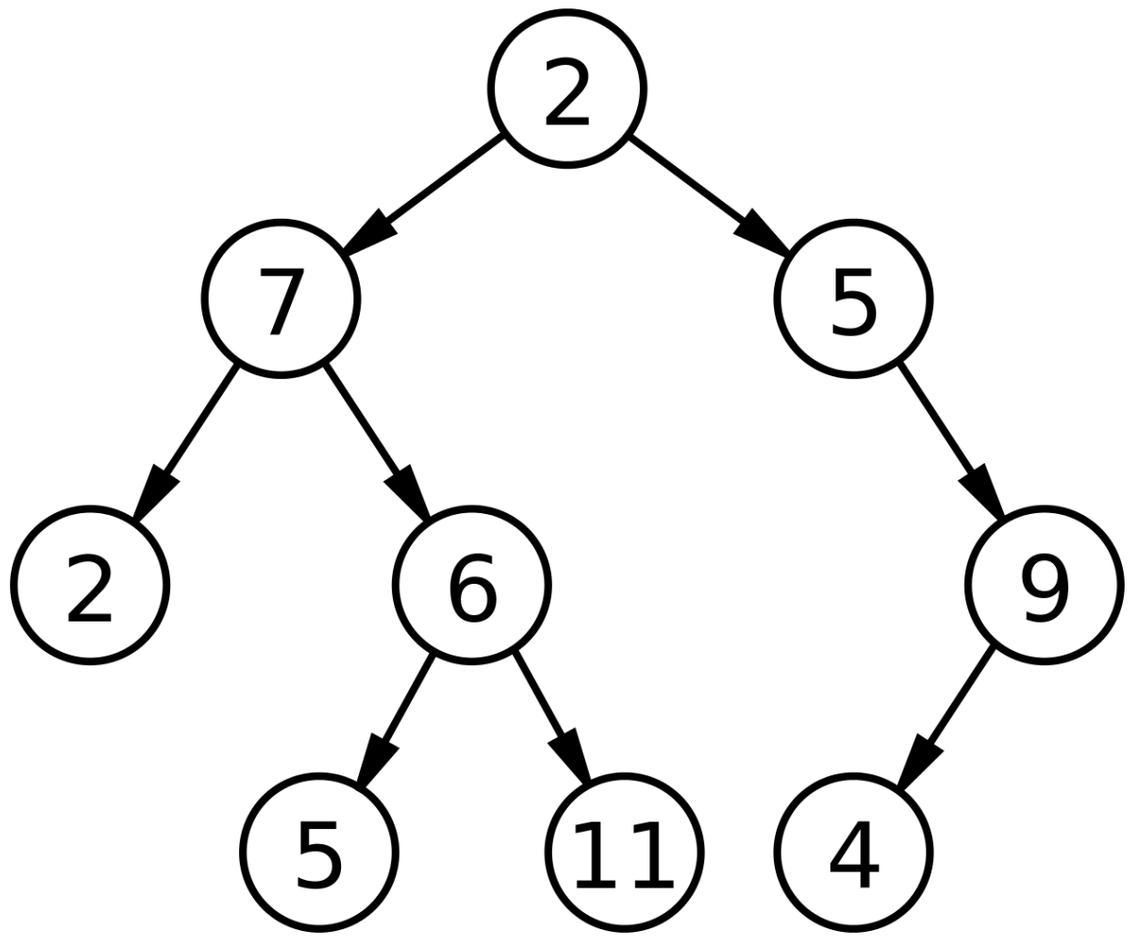


In-order Traversal

- What if we process our node after we process our left subtree but before we process our right subtree?

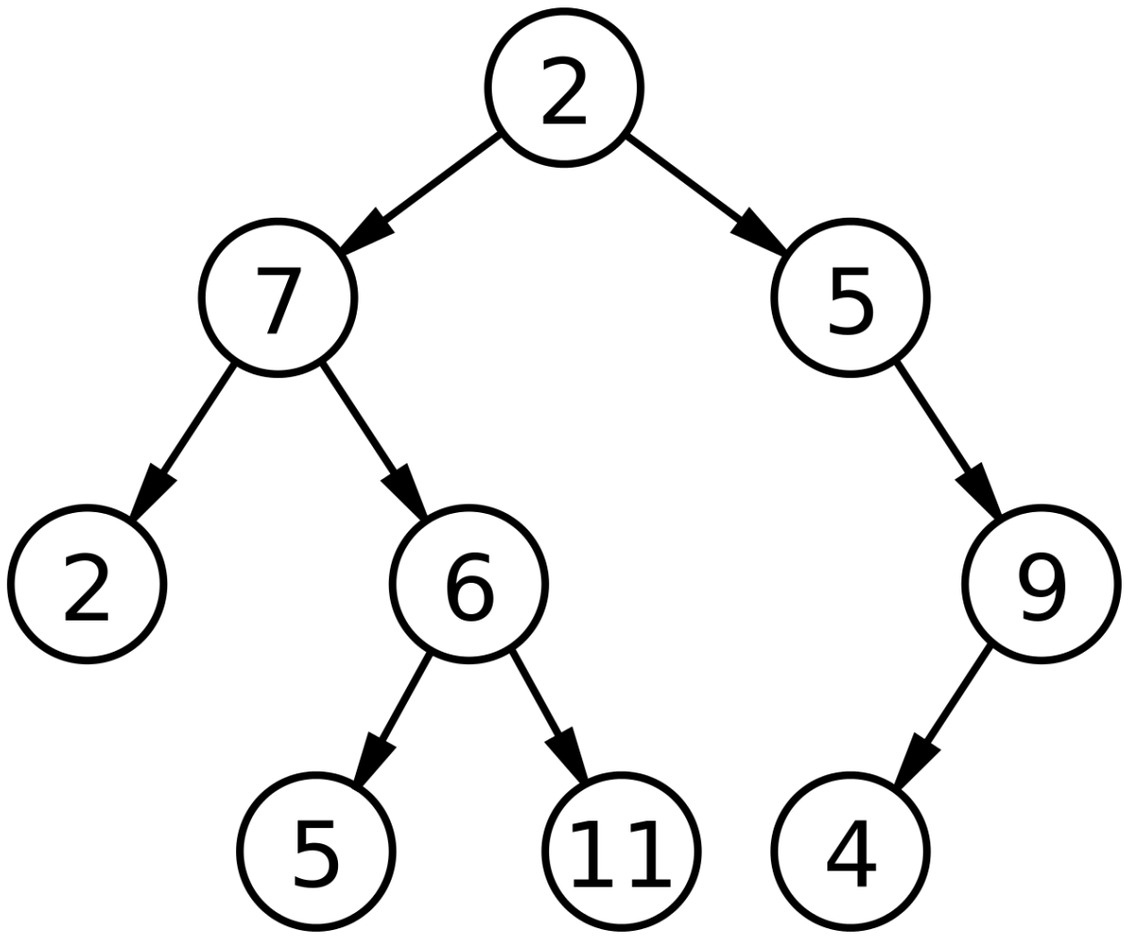
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```



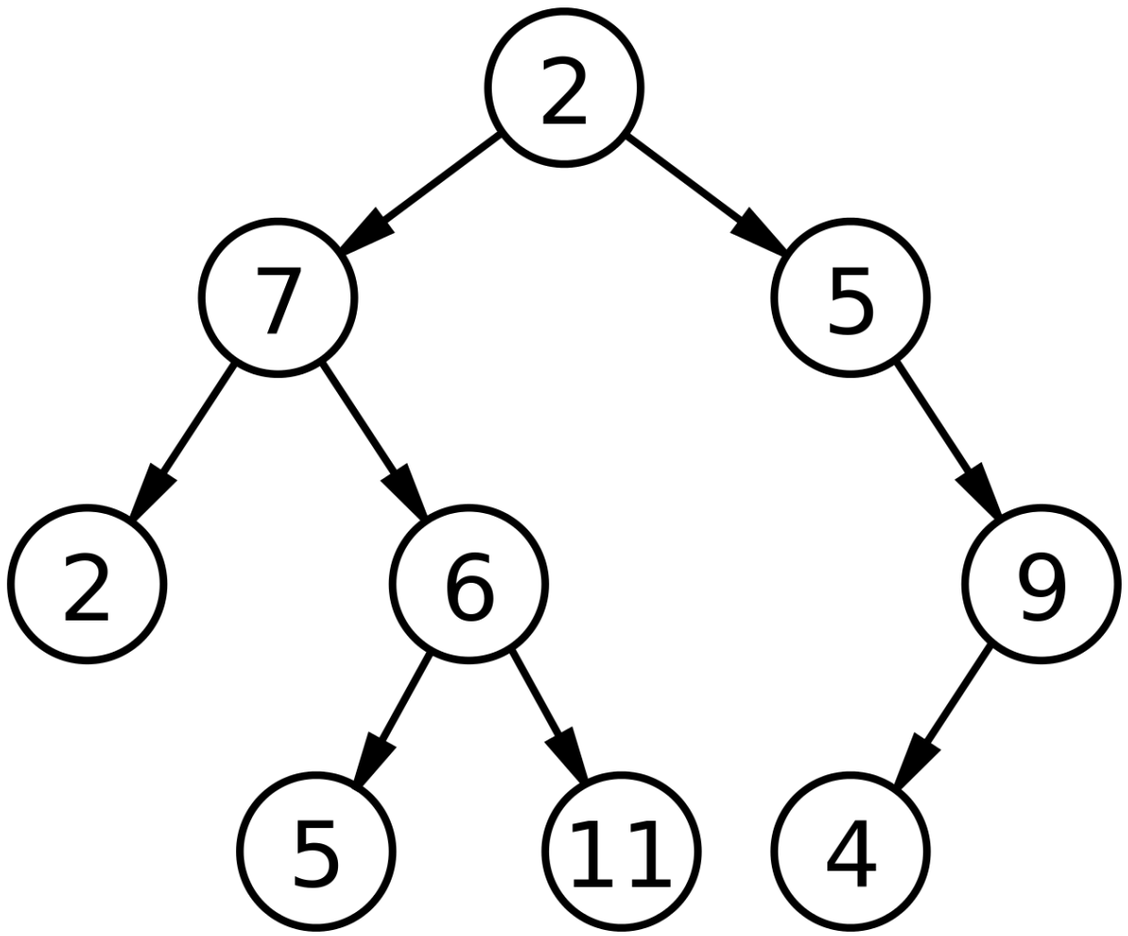
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2



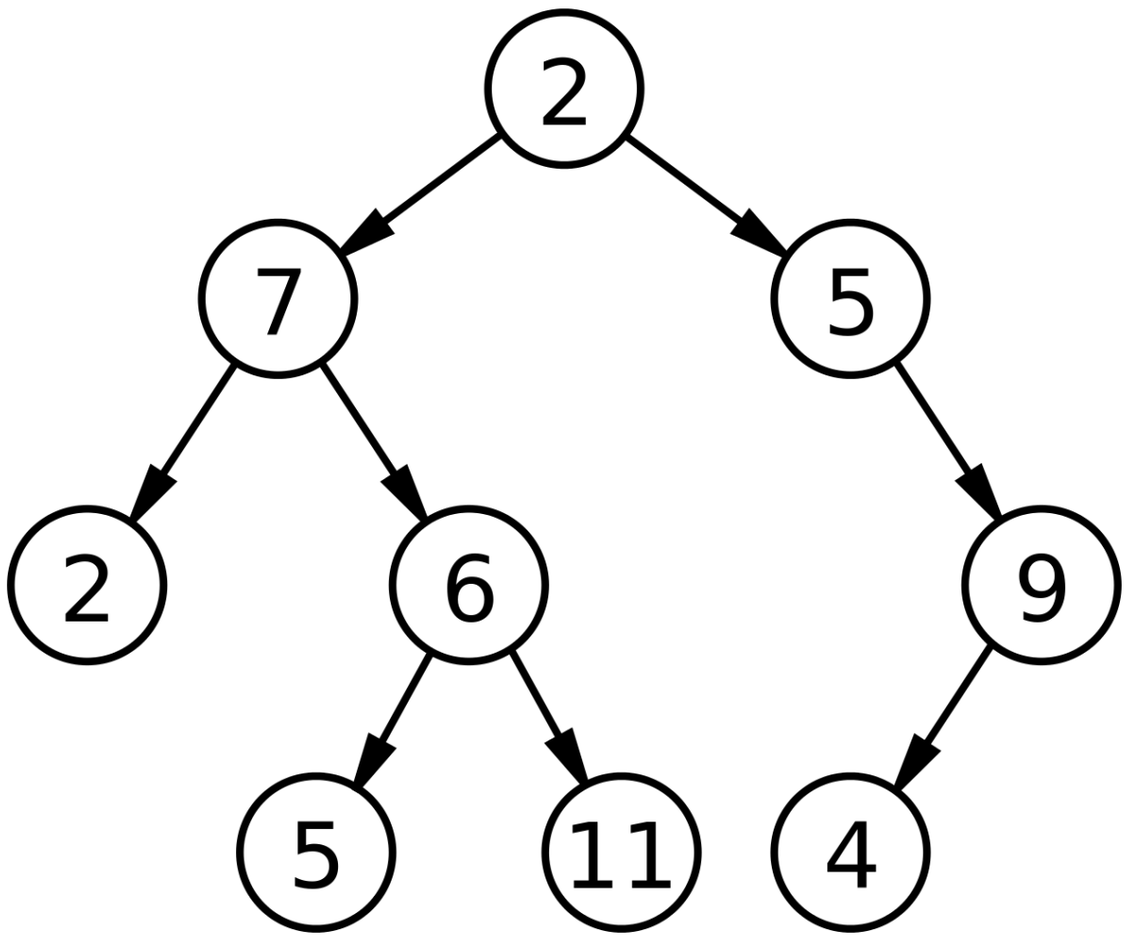
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7



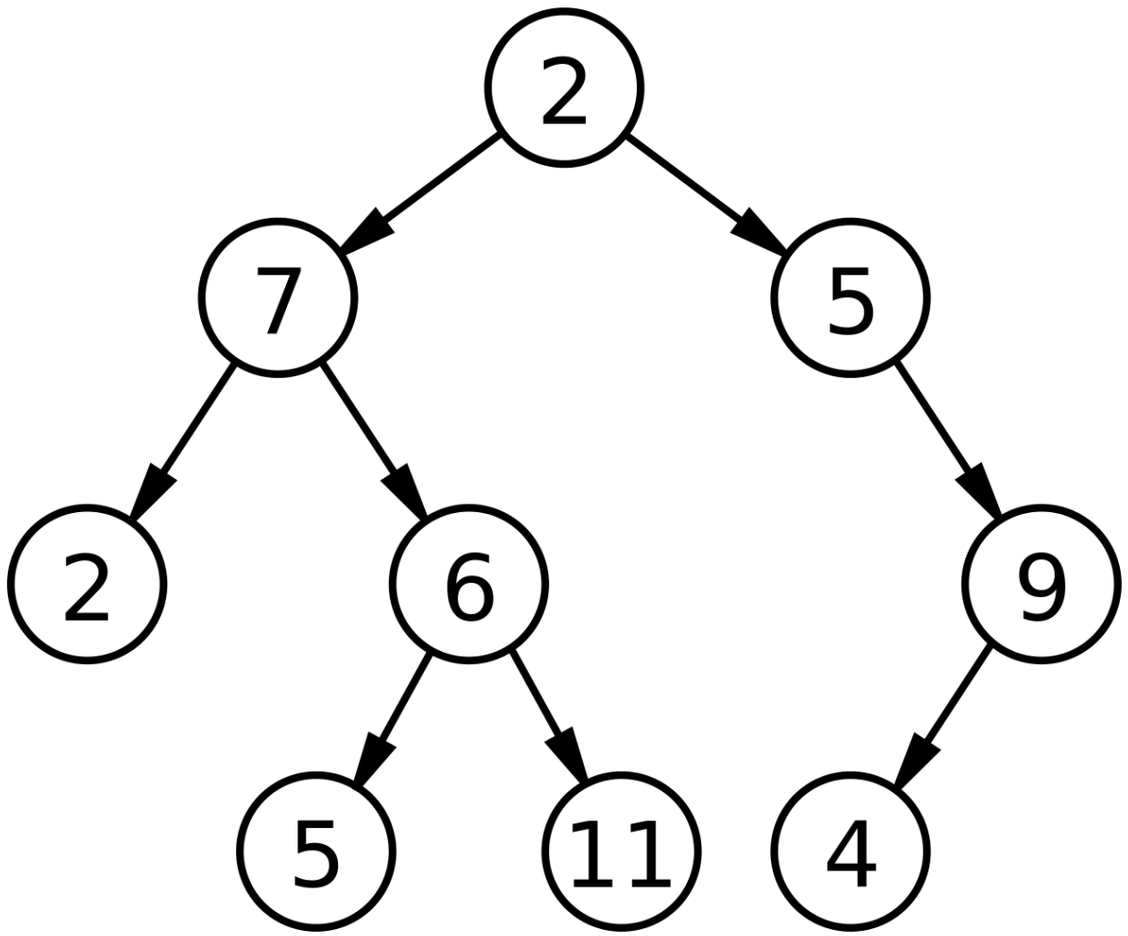
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5



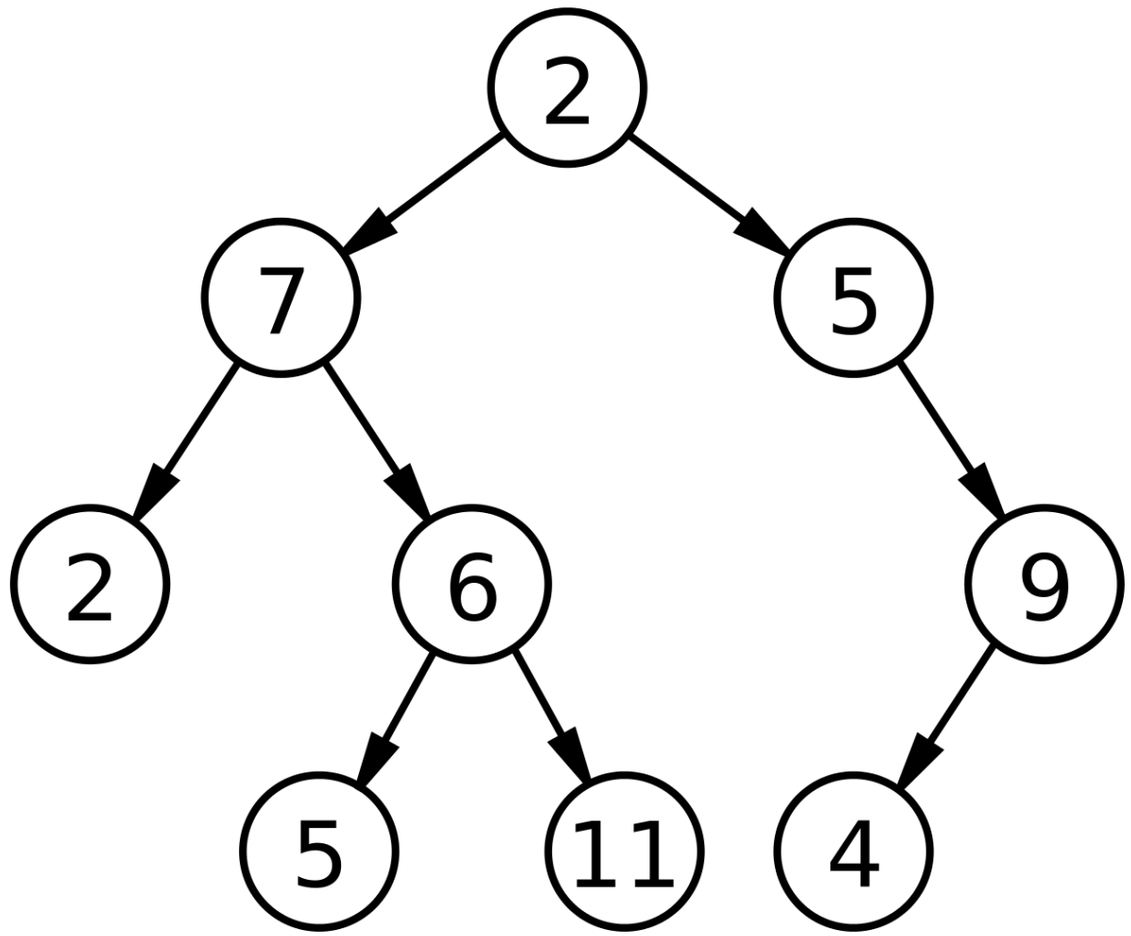
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5
6



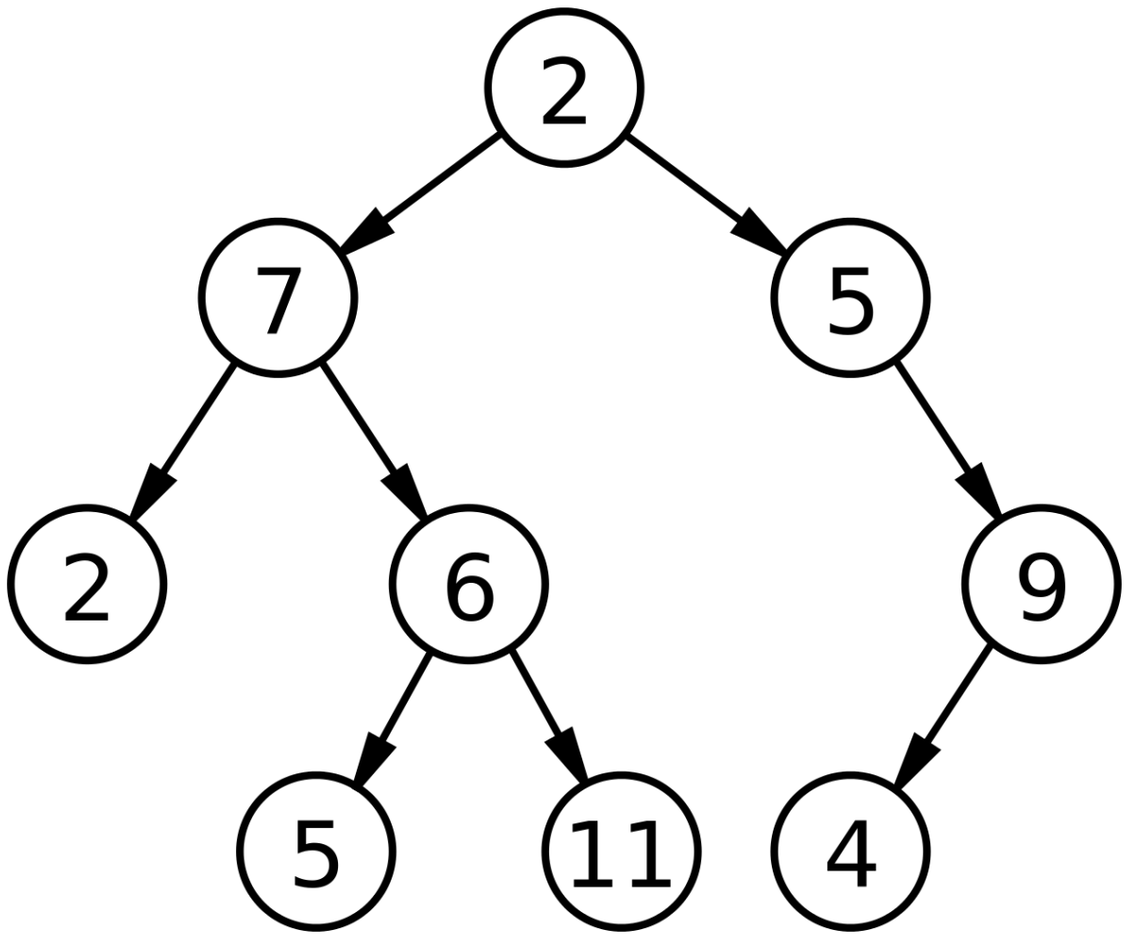
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5
6
11



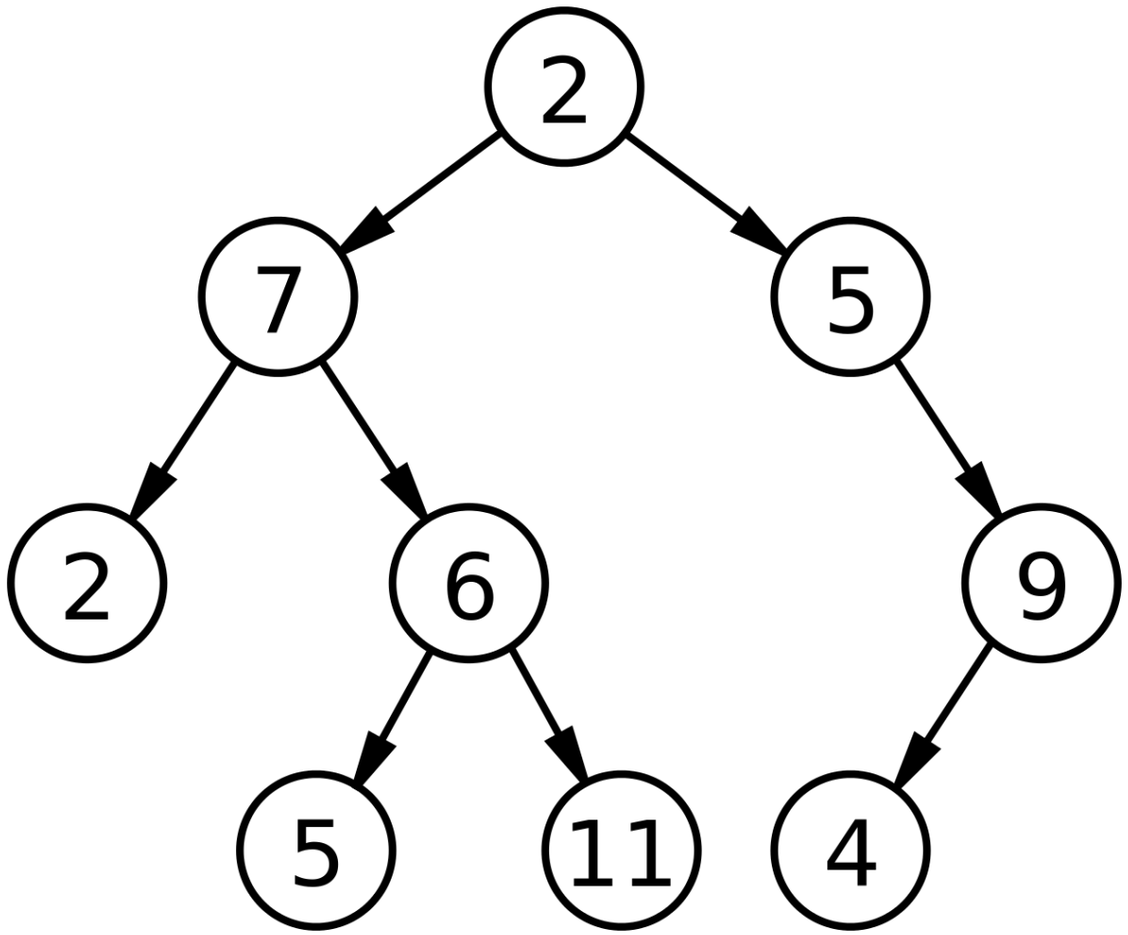
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5
6
11
2



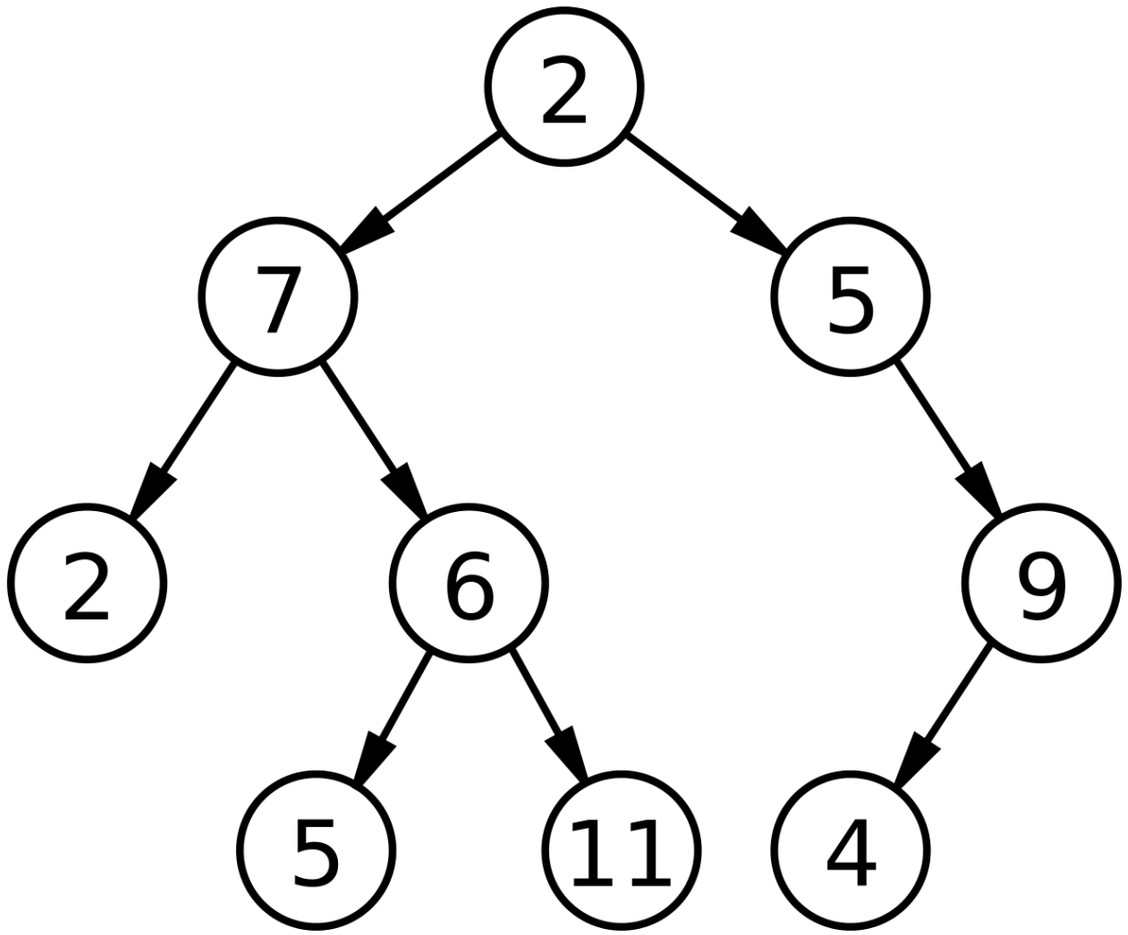
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5
6
11
2
5



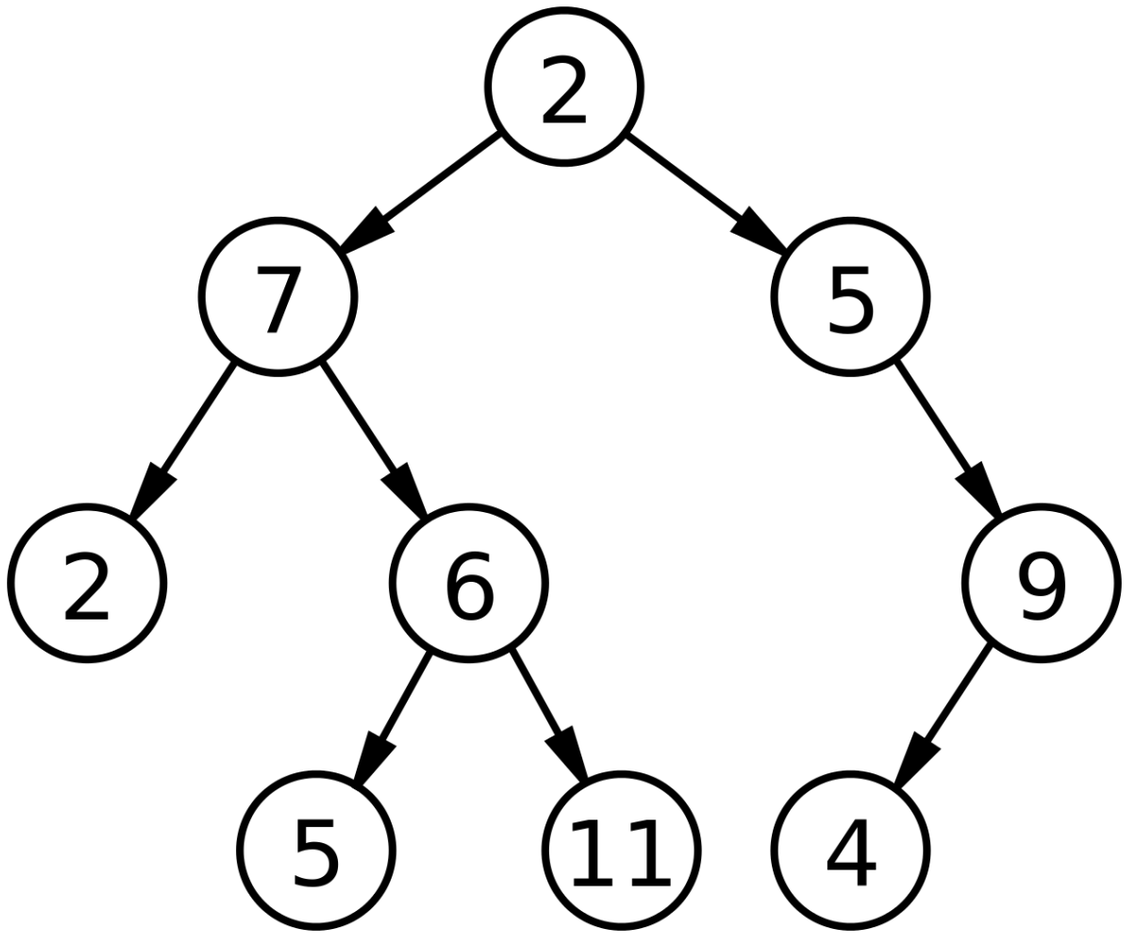
```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5
6
11
2
5
4



```
function traverse_inorder(node):
        if node.left != null:
            traverse_inorder(node.left)
        visit the current node
        if node.right != null:
            traverse_inorder(node.right)
```

2
7
5
6
11
2
5
4
9

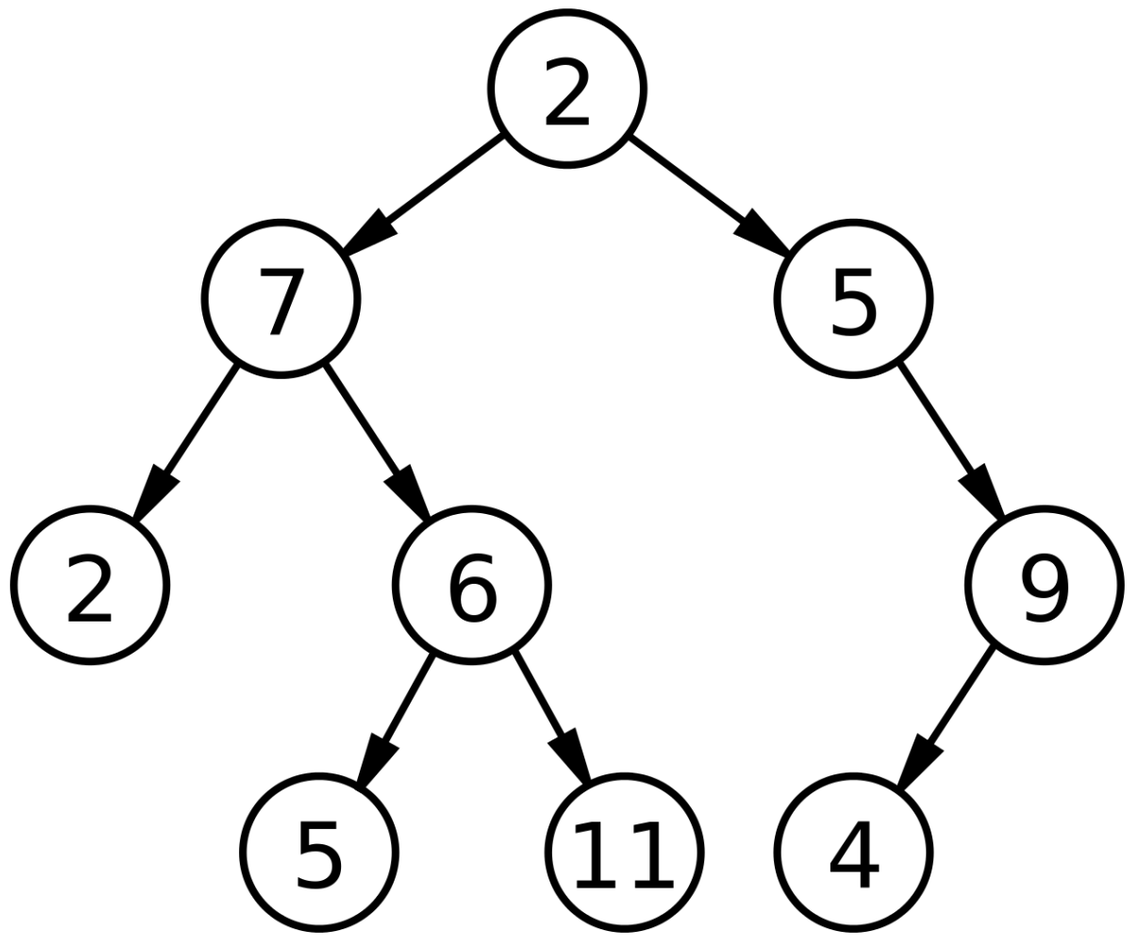


Post-order Traversal

- What if we don't process our node until after we traverse both the left *and* right subtrees?

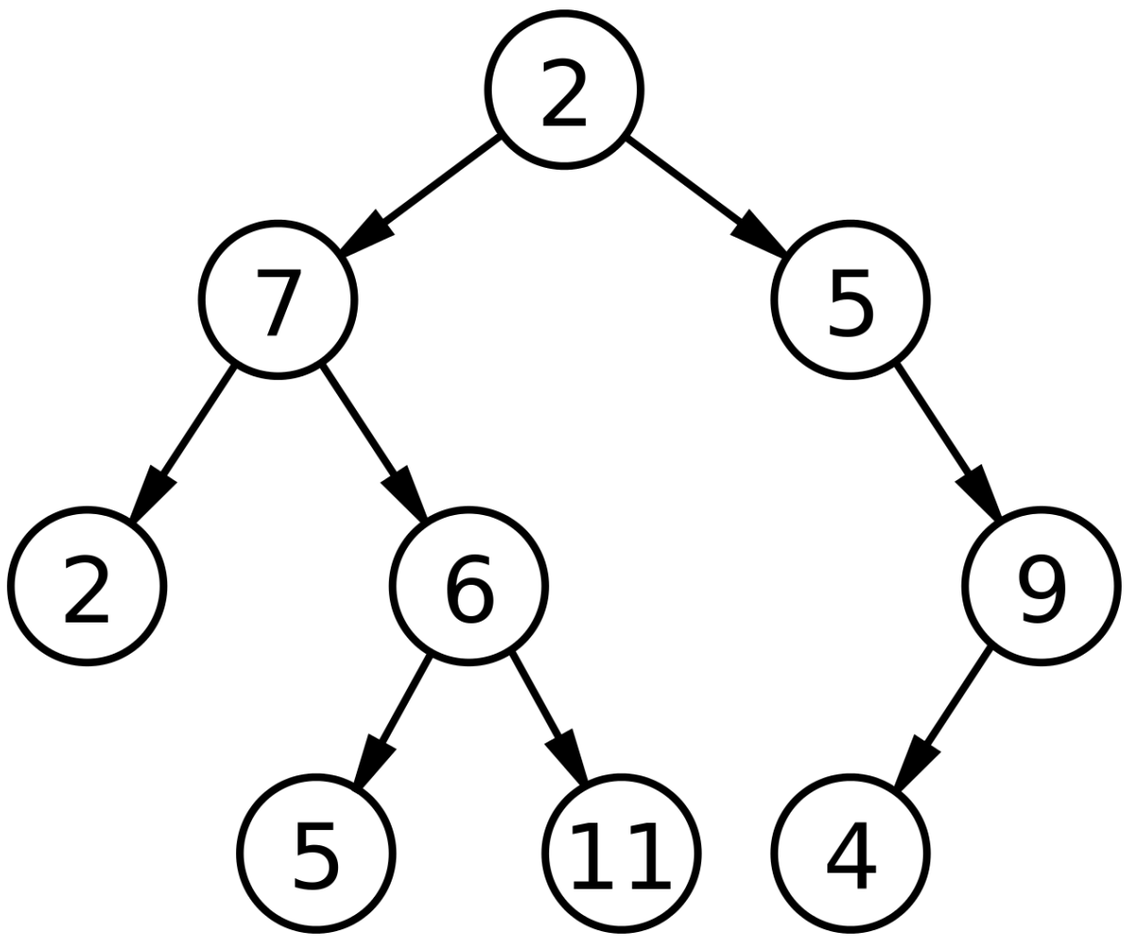
```
function traverse_postorder(node):
        if node.left != null:
                traverse_postorder(node.left)
        if node.right != null:
                traverse_postorder(node.right)
        visit the current node
```

```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
            traverse_postorder(node.right)
      visit the current node
```



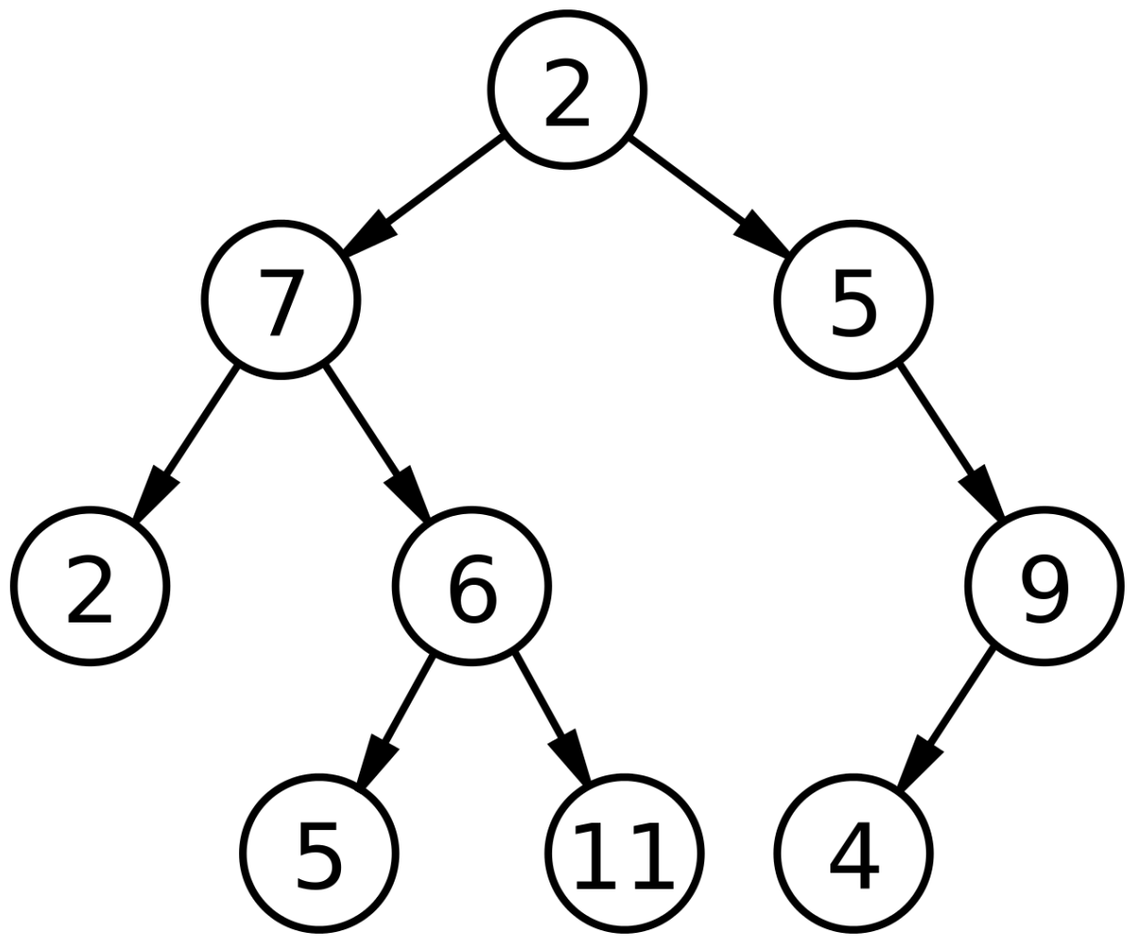
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
           traverse_postorder(node.right)
      visit the current node
```

2



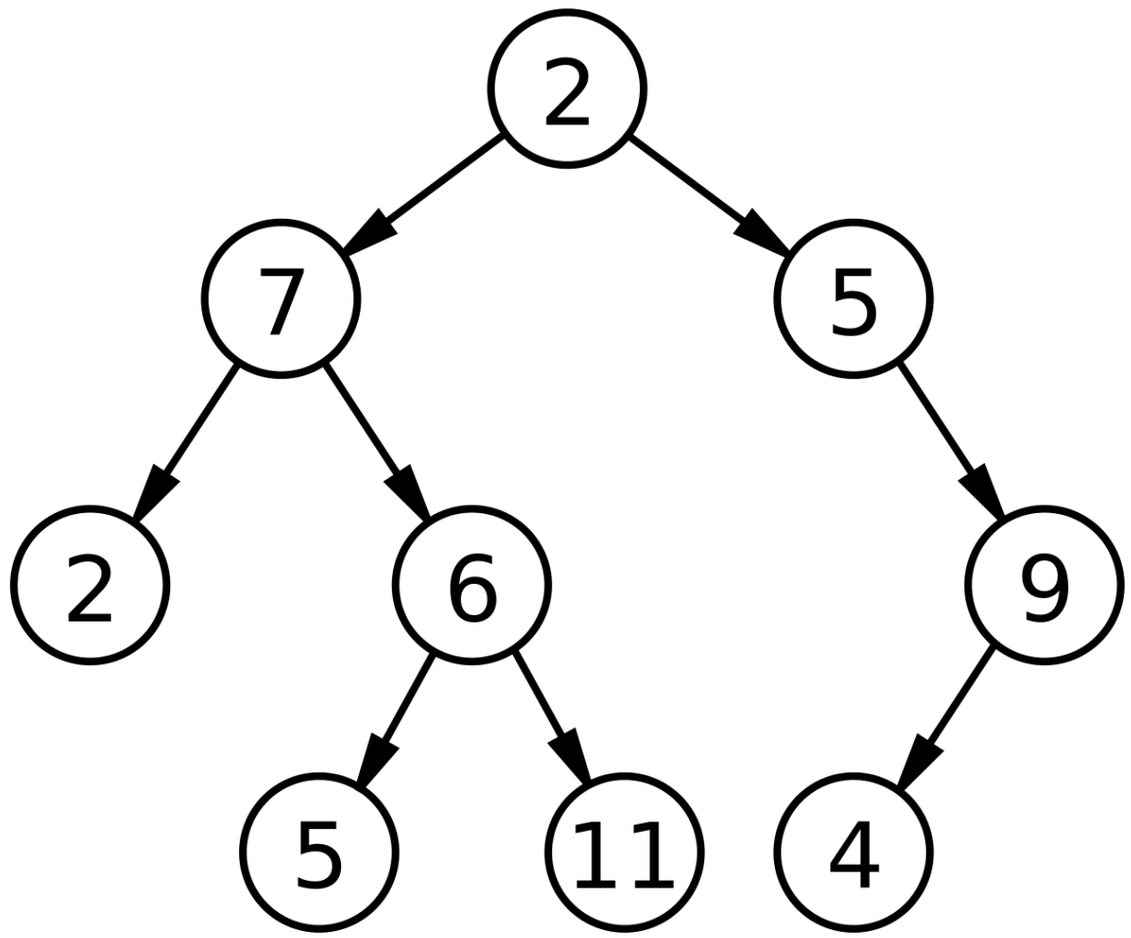
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
           traverse_postorder(node.right)
      visit the current node
```

2
5



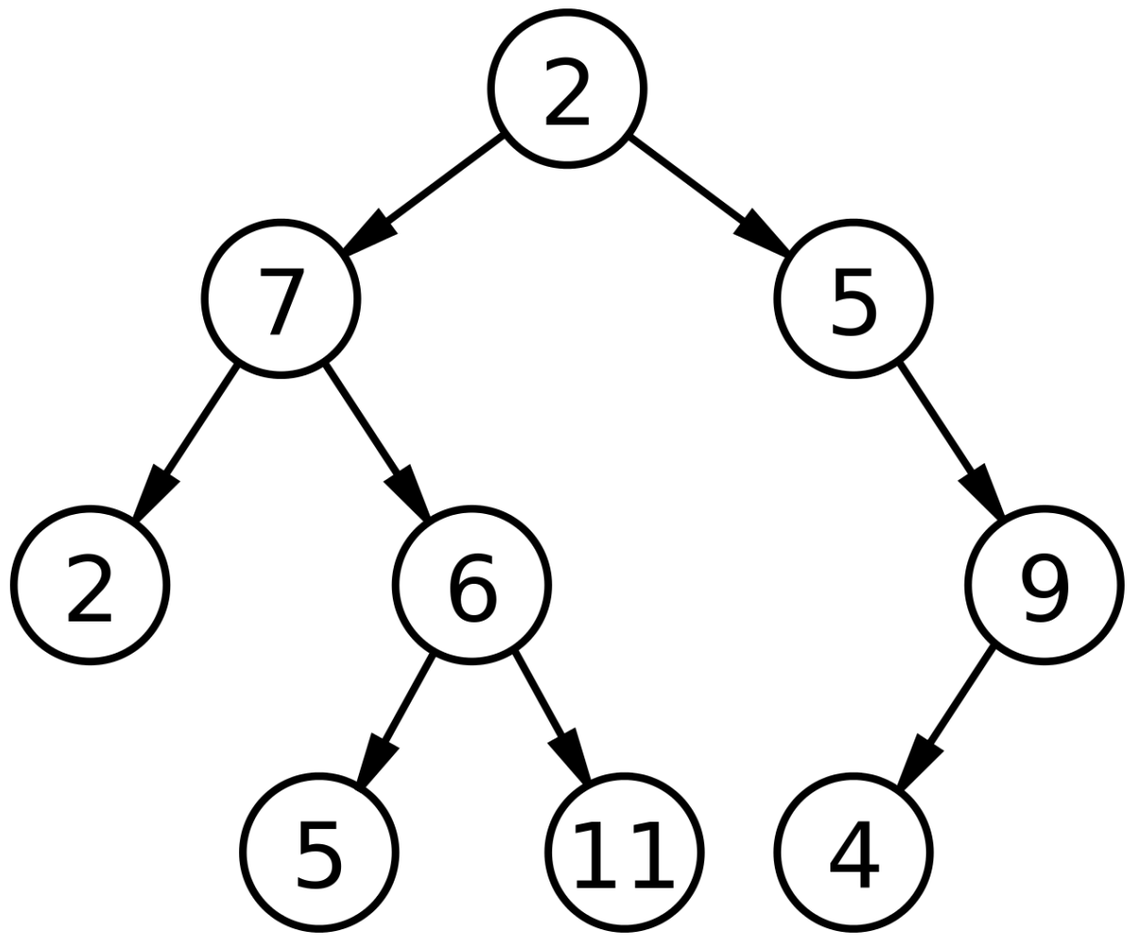
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
           traverse_postorder(node.right)
      visit the current node
```

2
5
11



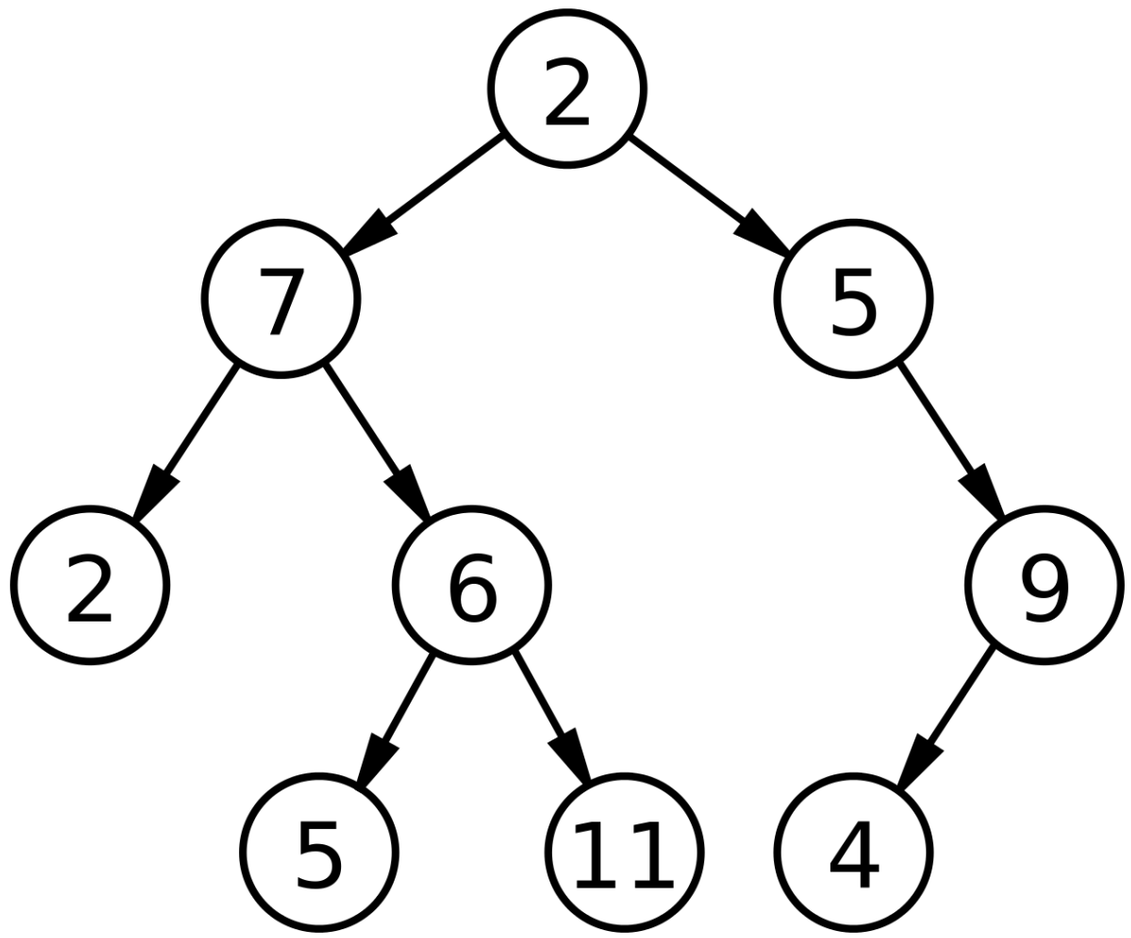
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
           traverse_postorder(node.right)
      visit the current node
```

2
5
11
6



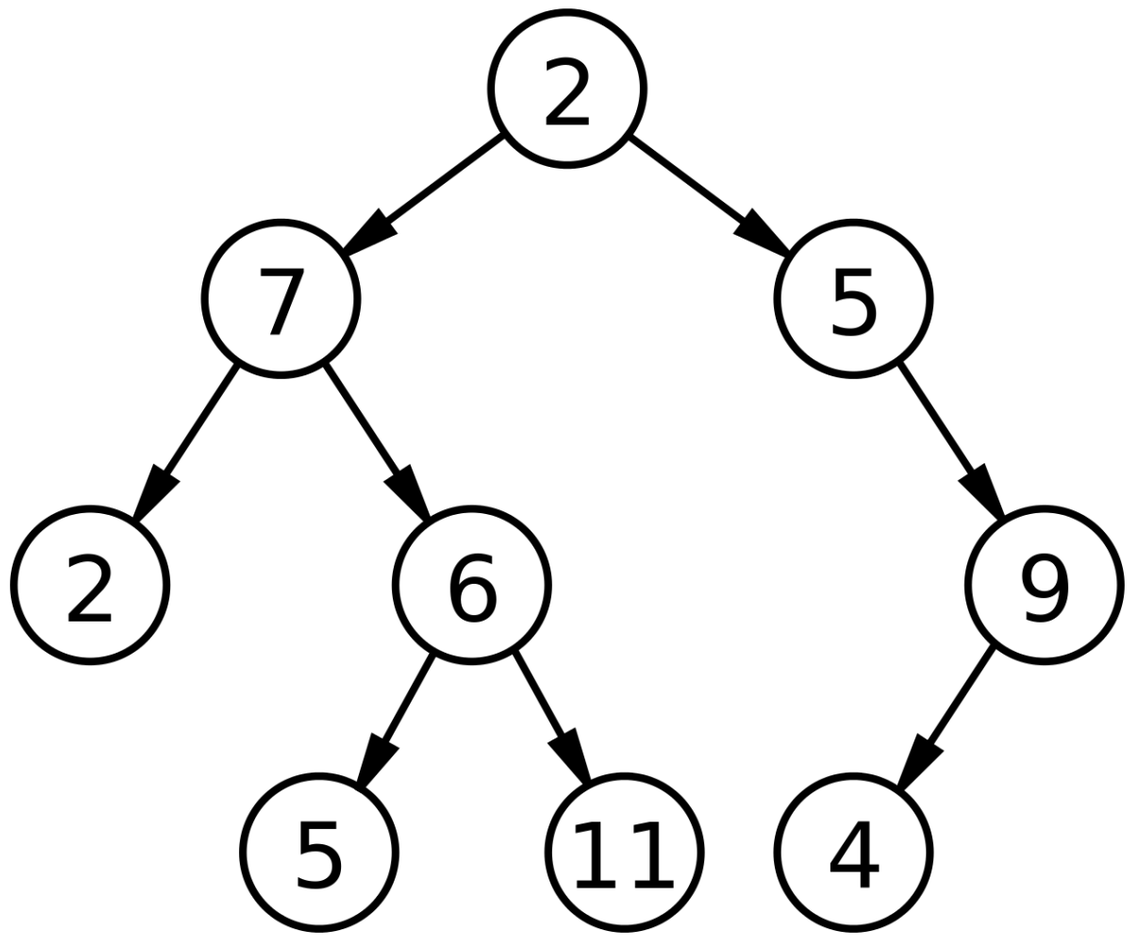
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
           traverse_postorder(node.right)
      visit the current node
```

2
5
11
6
7



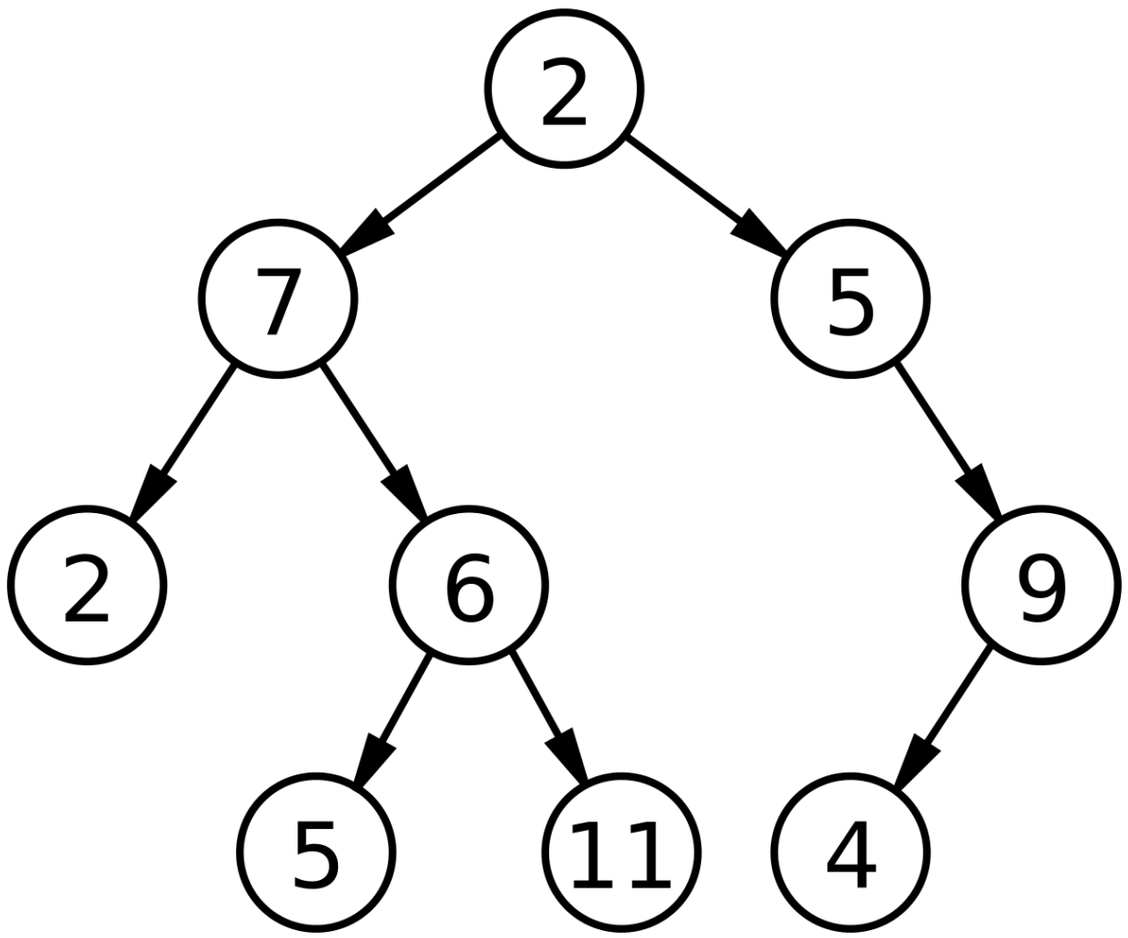
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
            traverse_postorder(node.right)
      visit the current node
```

2
5
11
6
7
4



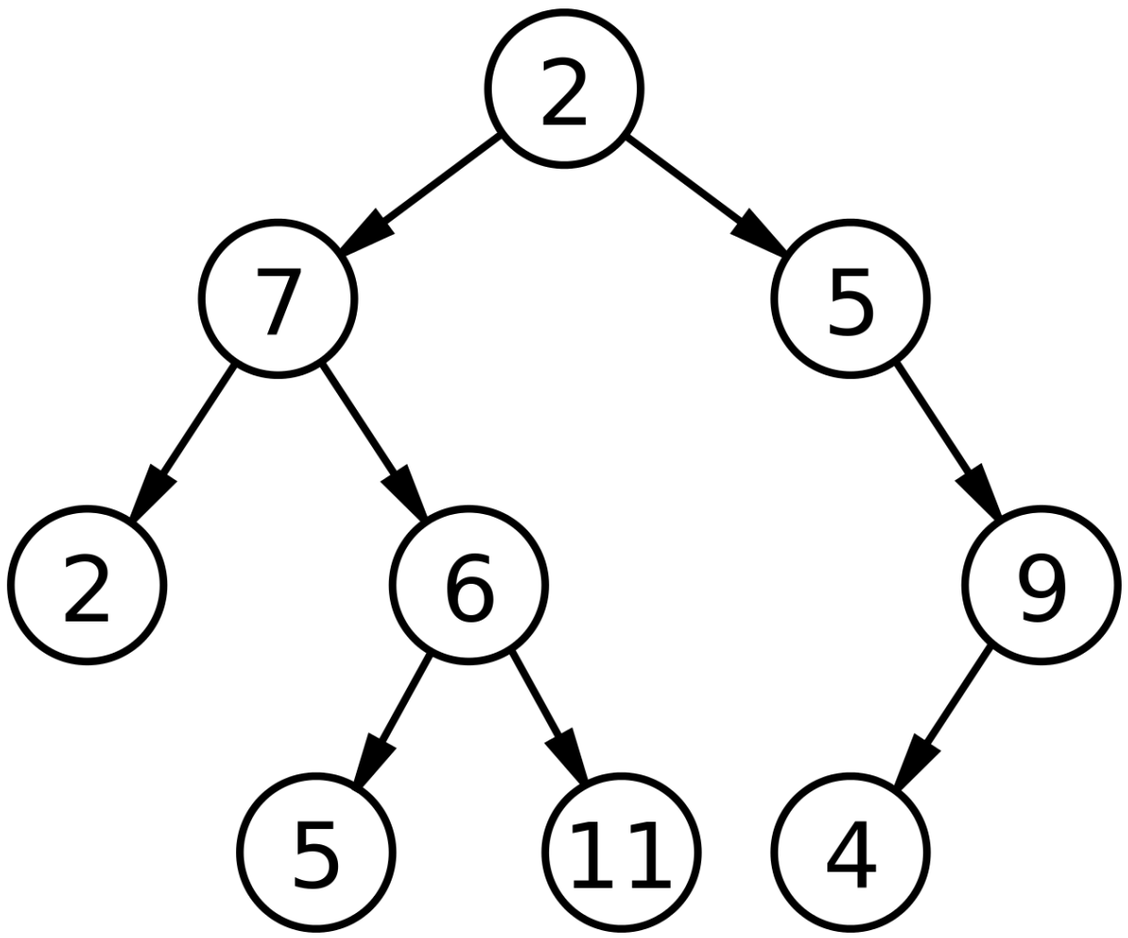
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
            traverse_postorder(node.right)
      visit the current node
```

2
5
11
6
7
4
9



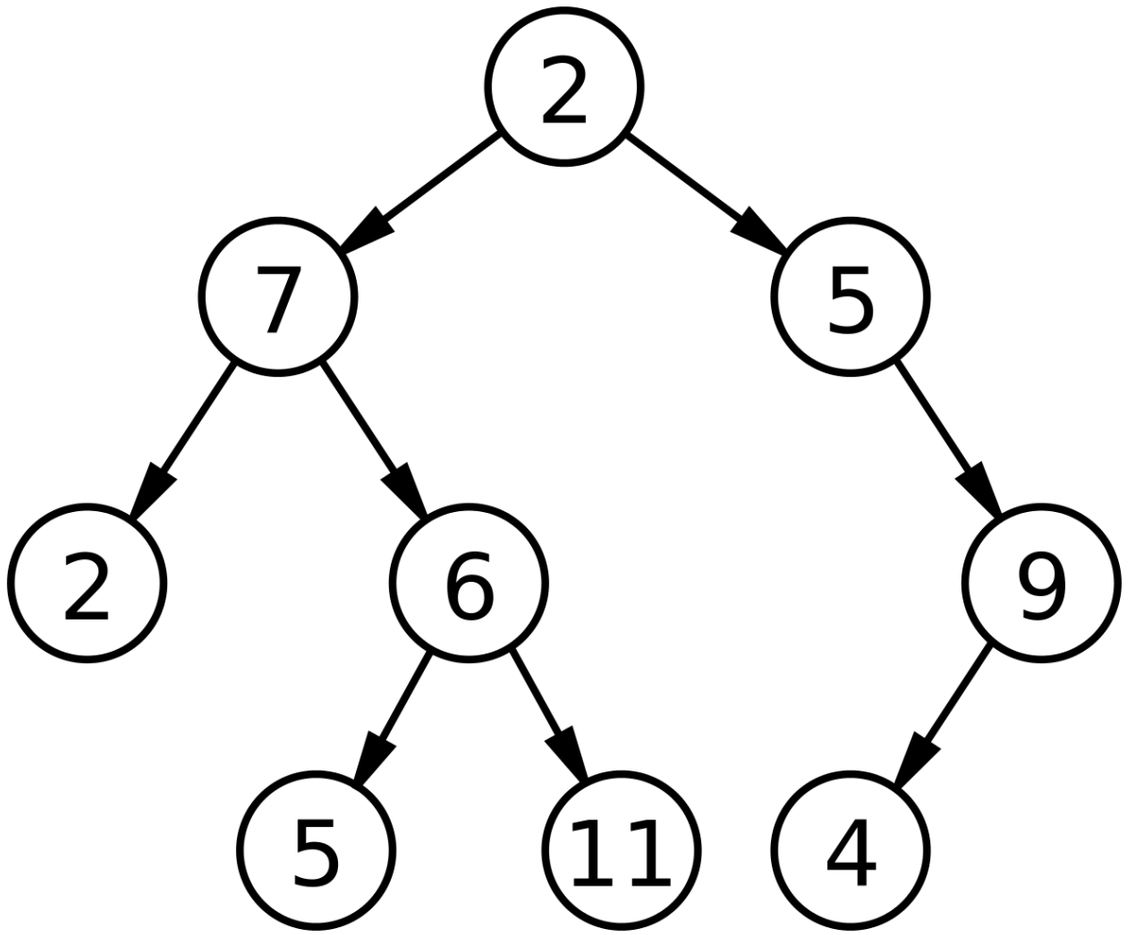
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
           traverse_postorder(node.right)
      visit the current node
```

2
5
11
6
7
4
9
5



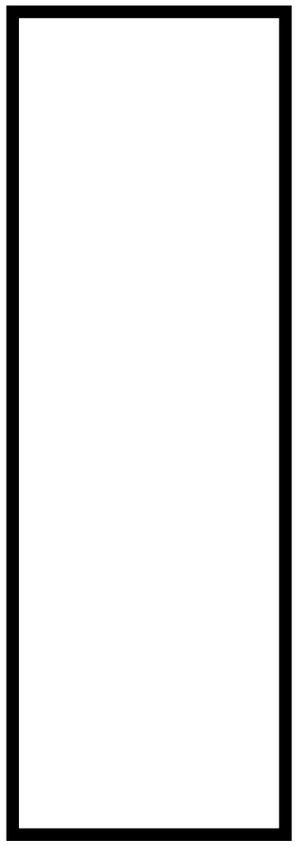
```
function traverse_postorder(node):
      if node.left != null:
            traverse_postorder(node.left)
      if node.right != null:
            traverse_postorder(node.right)
      visit the current node
```

2
5
11
6
7
4
9
5
2



Stacks

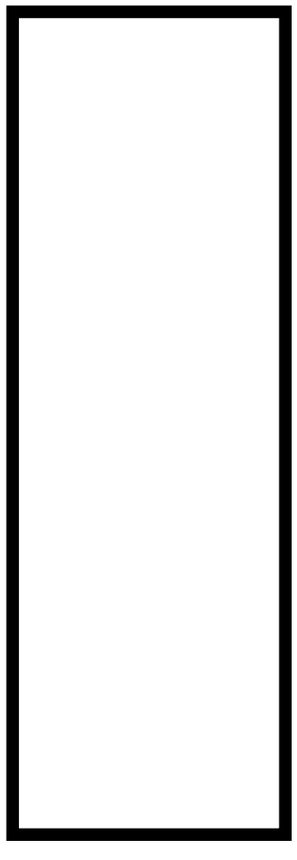
- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

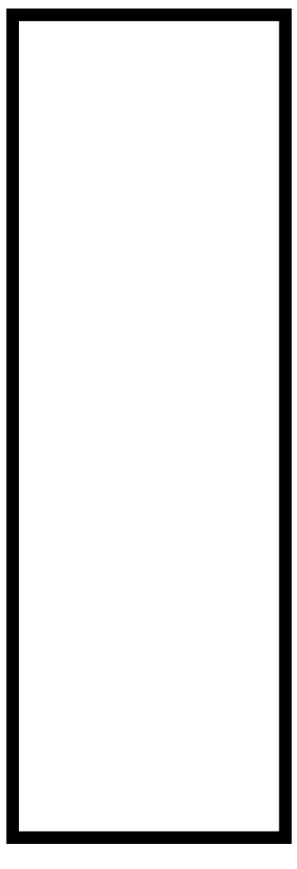
push(1)



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

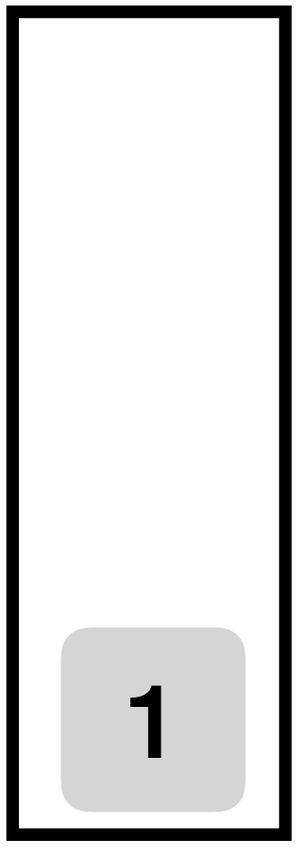
```
push(1)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

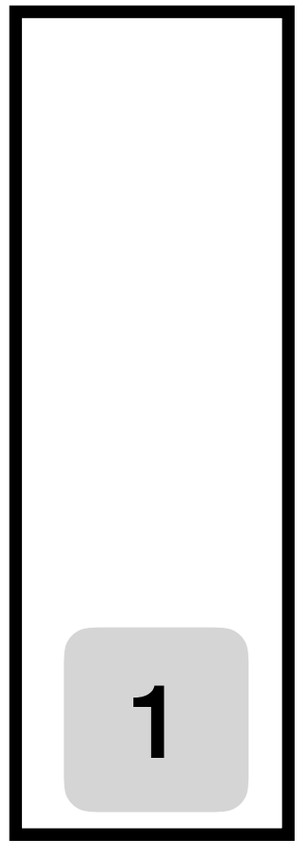
```
push(1)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

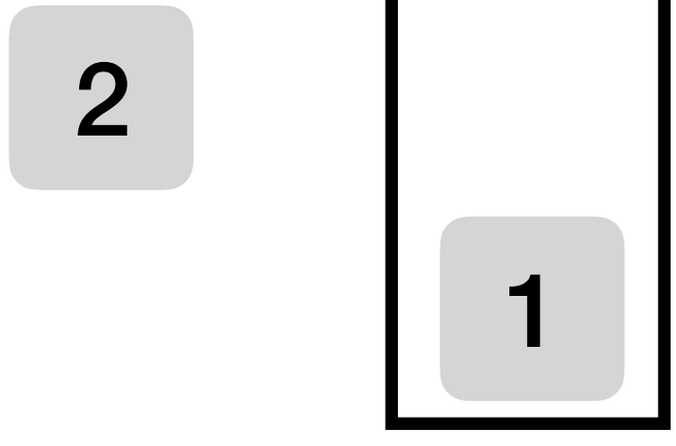
```
push(1)
push(2)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

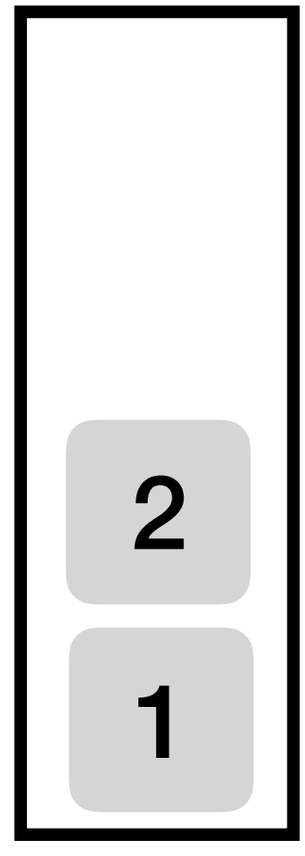
```
push(1)
push(2)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

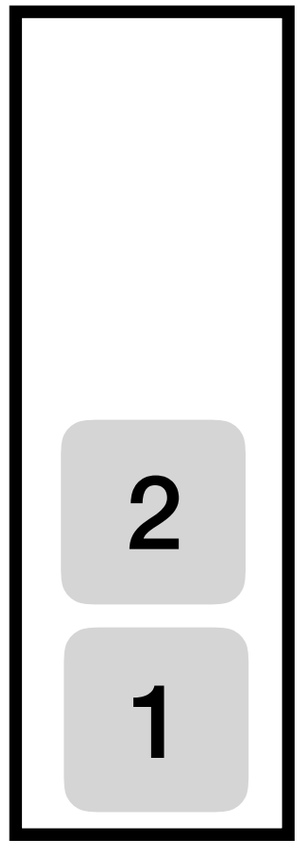
```
push(1)
push(2)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

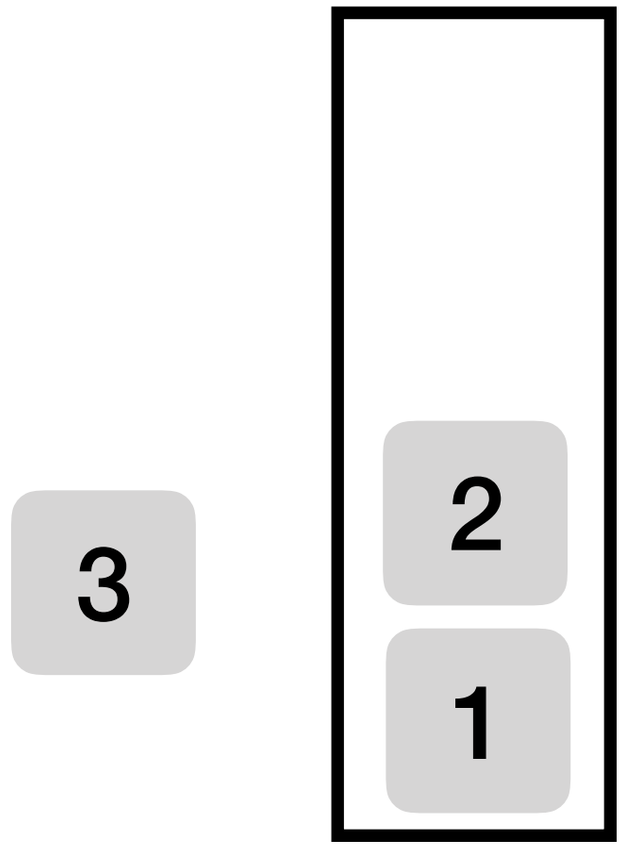
```
push(1)
push(2)
push(3)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

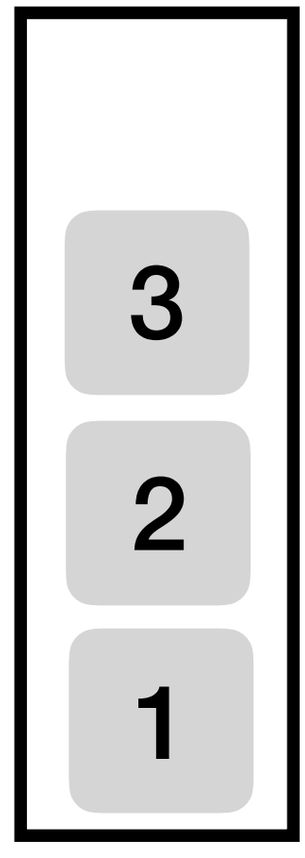
```
push(1)
push(2)
push(3)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

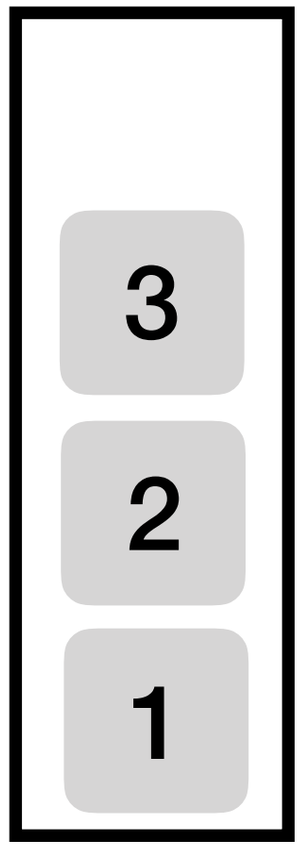
```
push(1)
push(2)
push(3)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

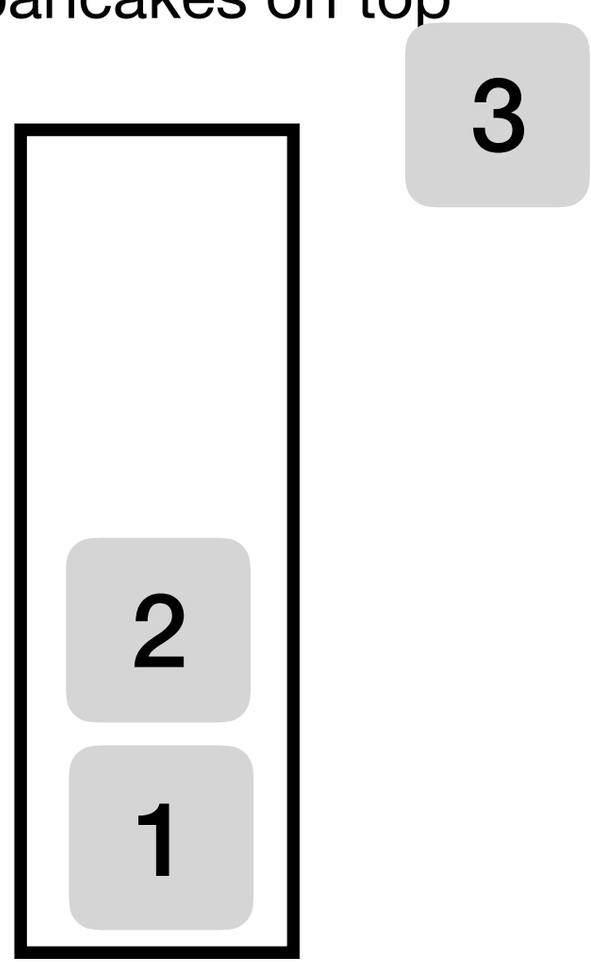
```
push(1)
push(2)
push(3)
 pop()
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

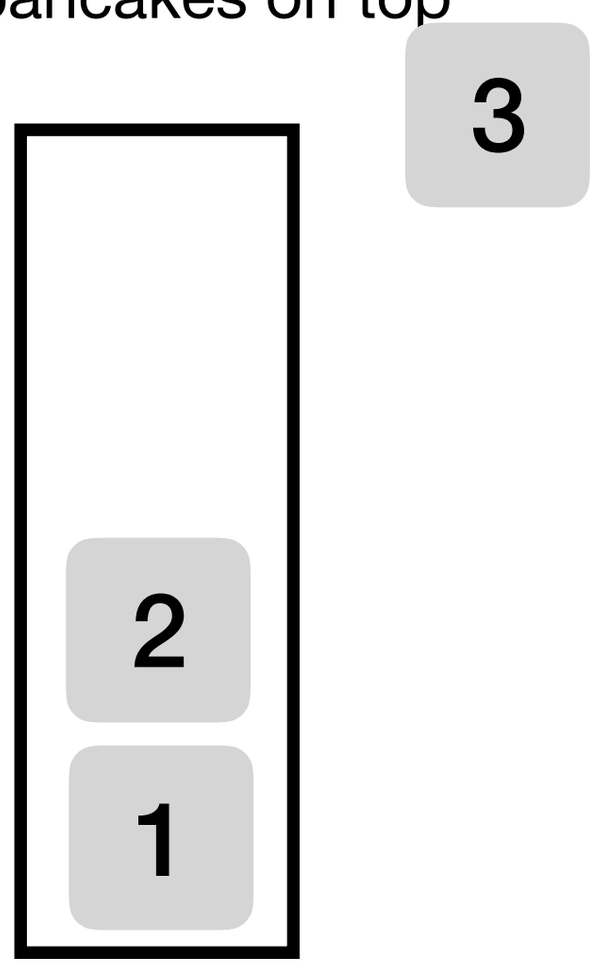
```
push(1)
push(2)
push(3)
 pop()
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

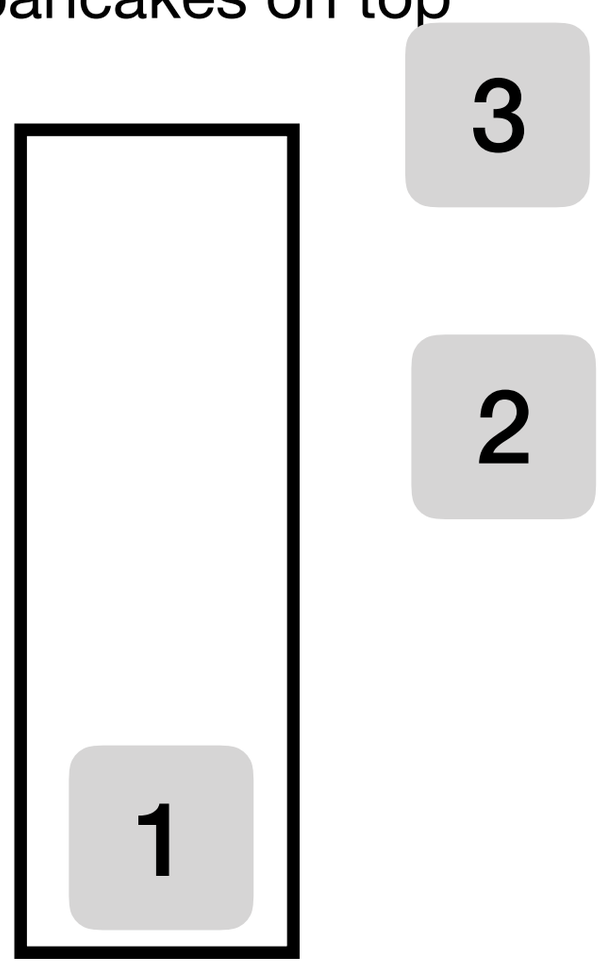
```
push(1)
push(2)
push(3)
 pop()
 pop()
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

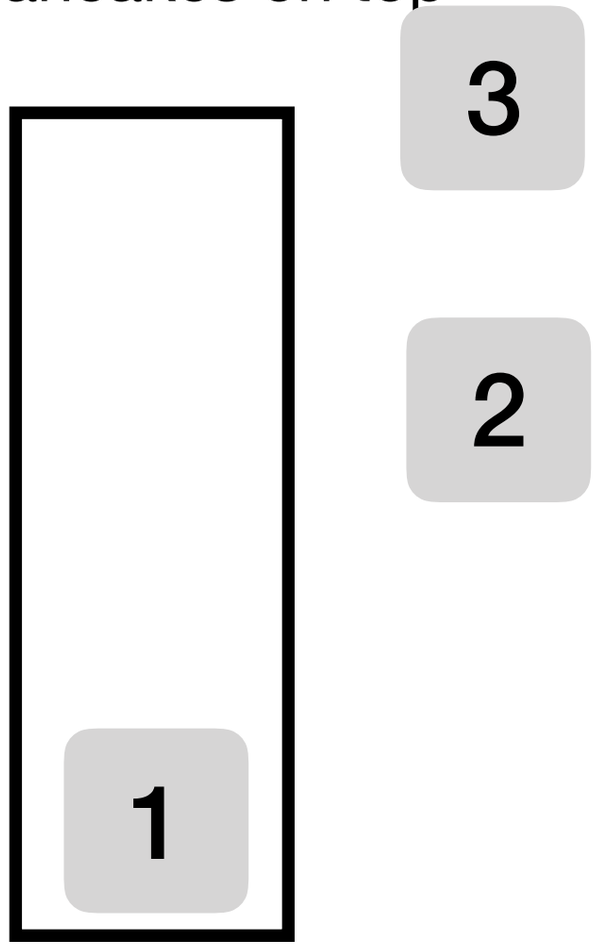
```
push(1)
push(2)
push(3)
 pop()
 pop()
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

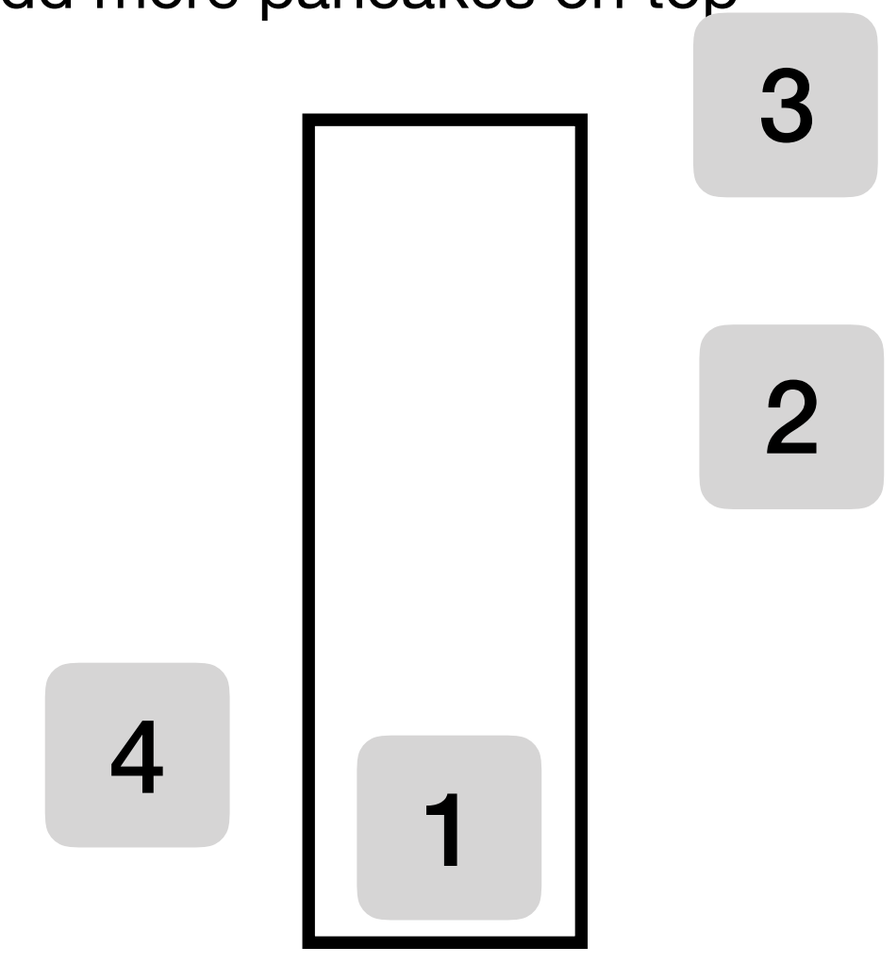
```
push(1)
push(2)
push(3)
 pop()
 pop()
push(4)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

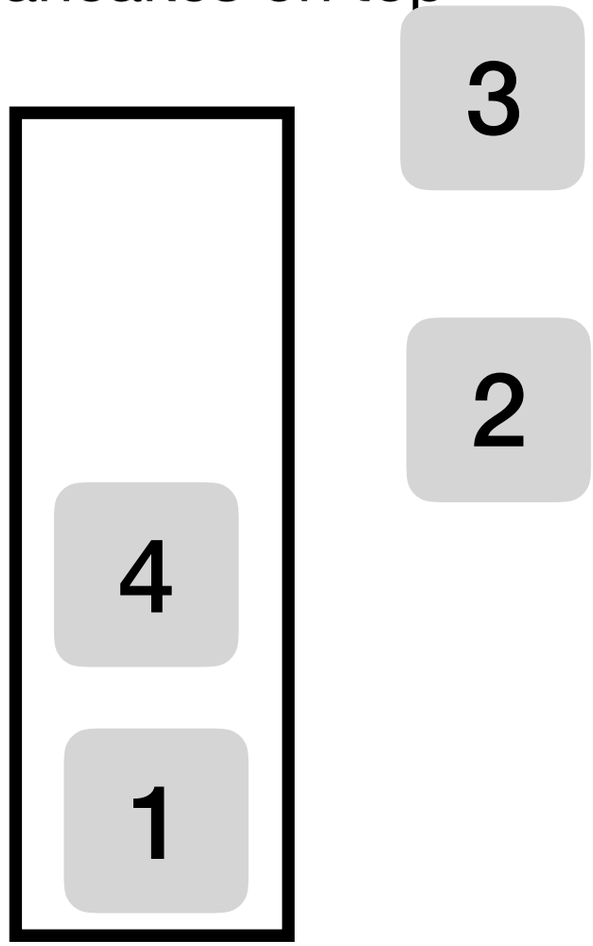
```
push(1)
push(2)
push(3)
 pop()
 pop()
push(4)
```



Stacks

- A stack is a data type that stores a collection of items.
- You can ***push*** (add) items onto the top of the stack and ***pop*** (remove) things off the top of the stack.
- Think of it like a stack of pancakes: you can add more pancakes on top or remove the top-most pancake.

```
push(1)
push(2)
push(3)
 pop()
 pop()
push(4)
```



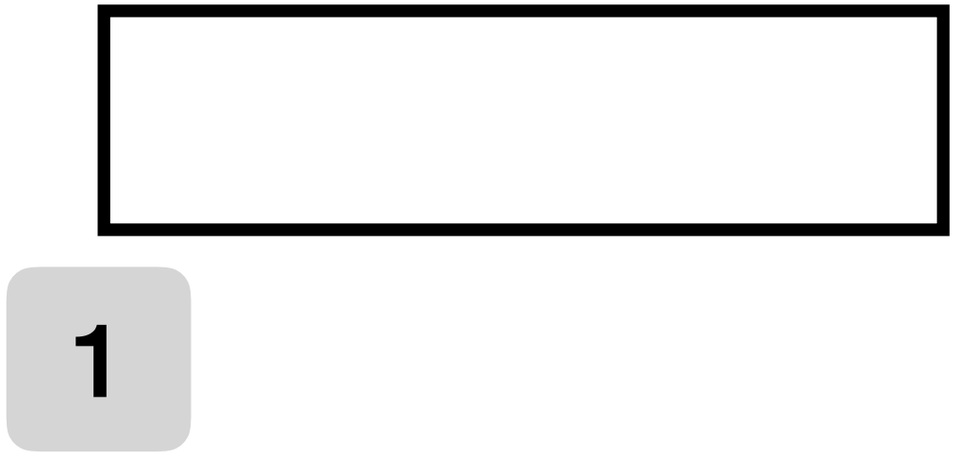
Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

`enqueue(1)`



Queues

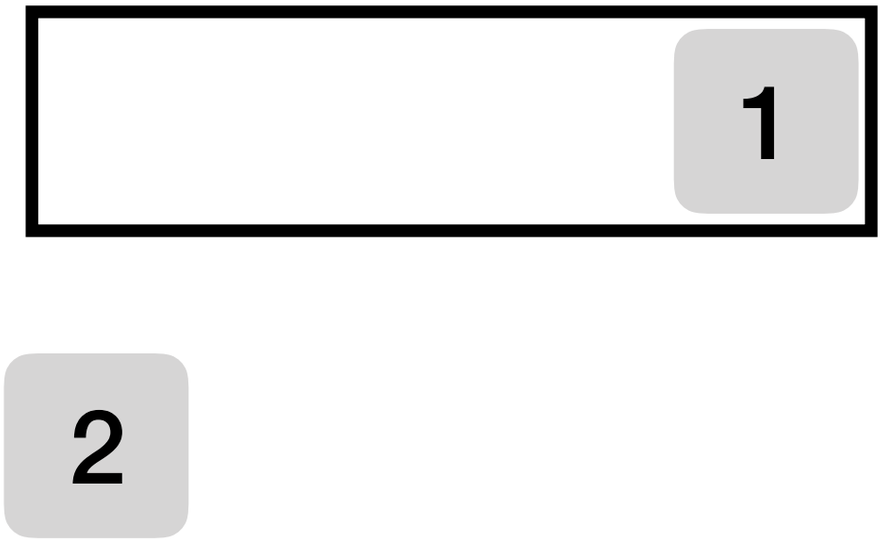
- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

`enqueue(1)`

Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

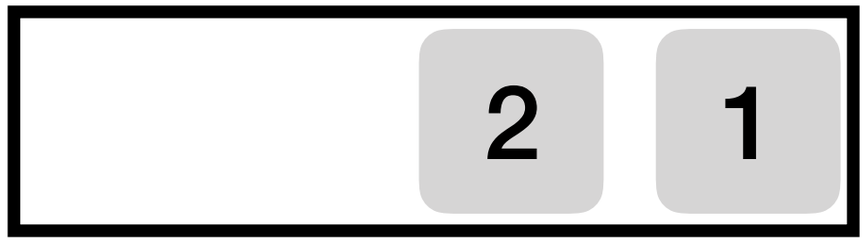
```
enqueue(1)
enqueue(2)
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

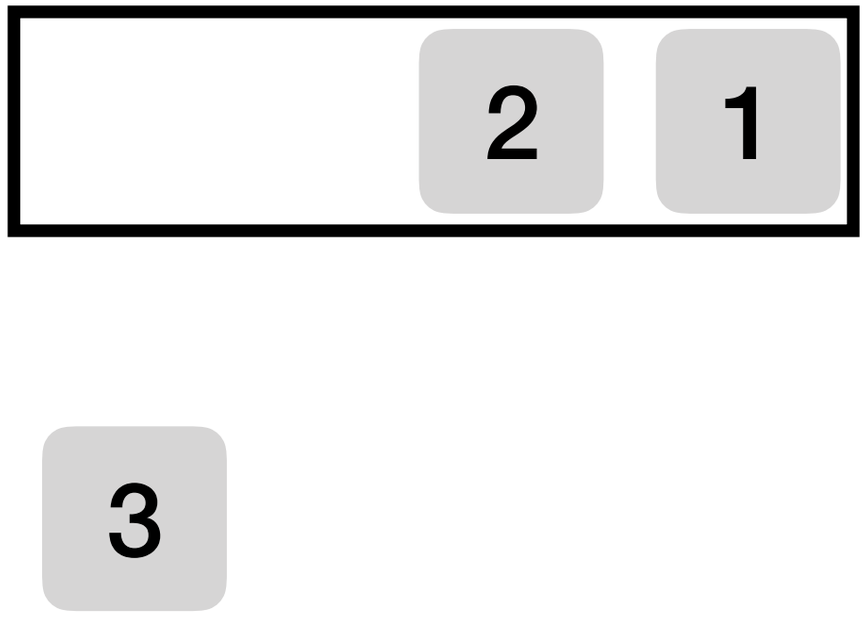
```
enqueue(1)
enqueue(2)
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

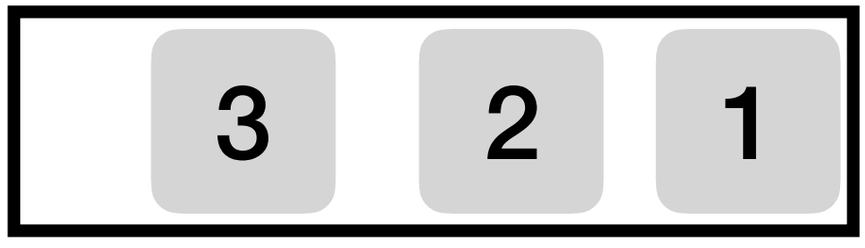
```
enqueue(1)
enqueue(2)
enqueue(3)
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

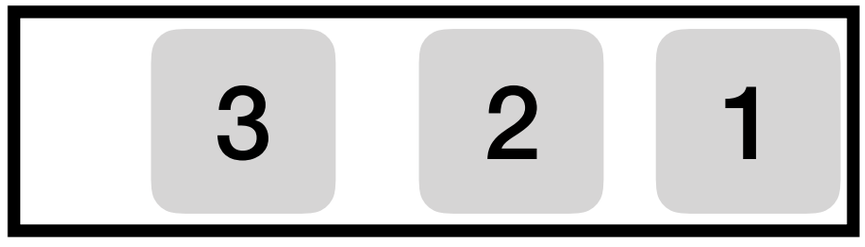
```
enqueue(1)
enqueue(2)
enqueue(3)
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

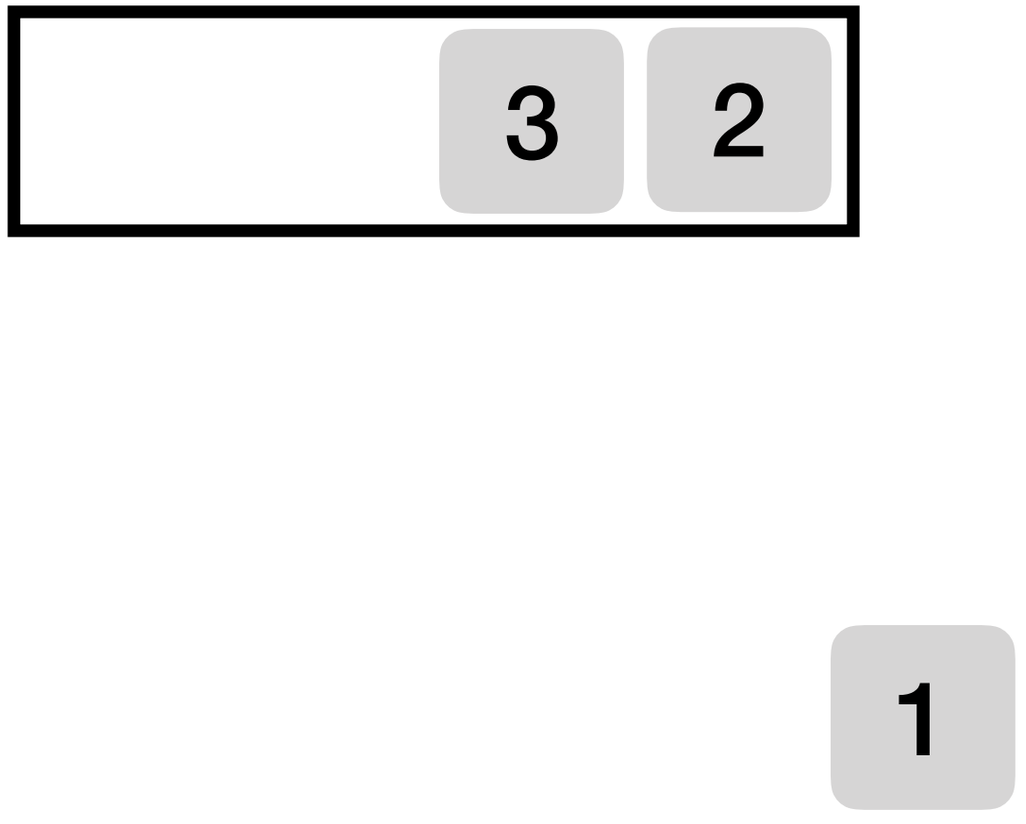
```
enqueue(1)
enqueue(2)
enqueue(3)
dequeue()
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

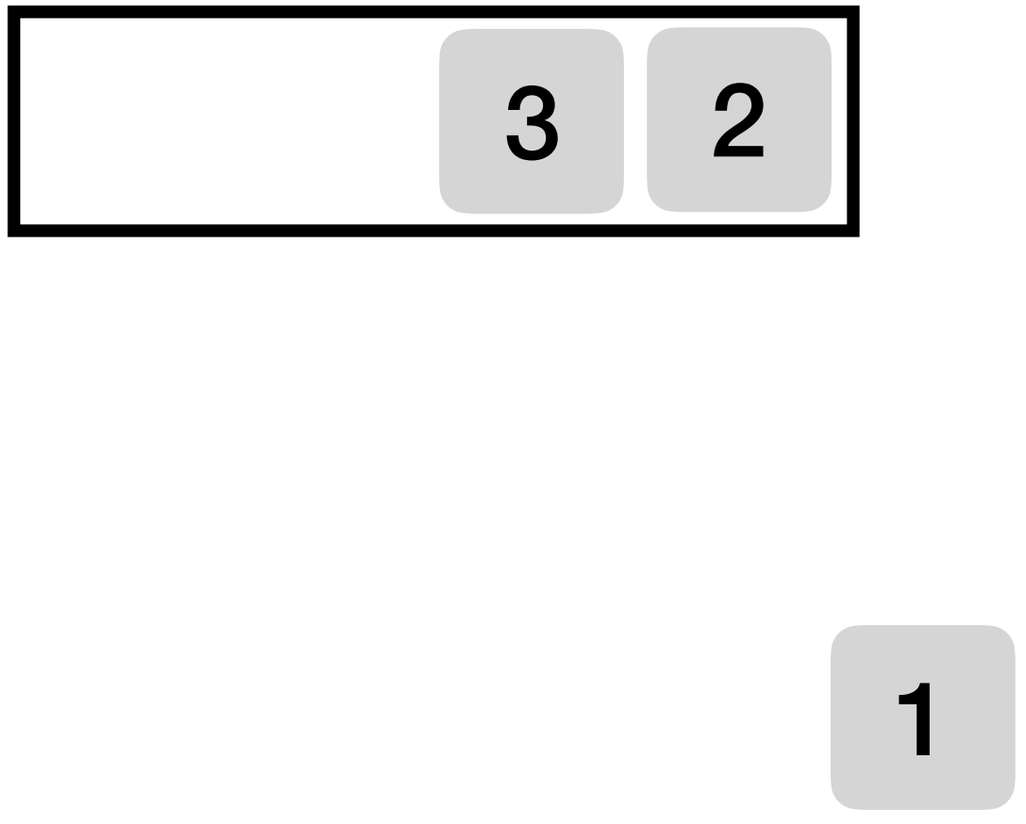
```
enqueue(1)
enqueue(2)
enqueue(3)
dequeue()
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

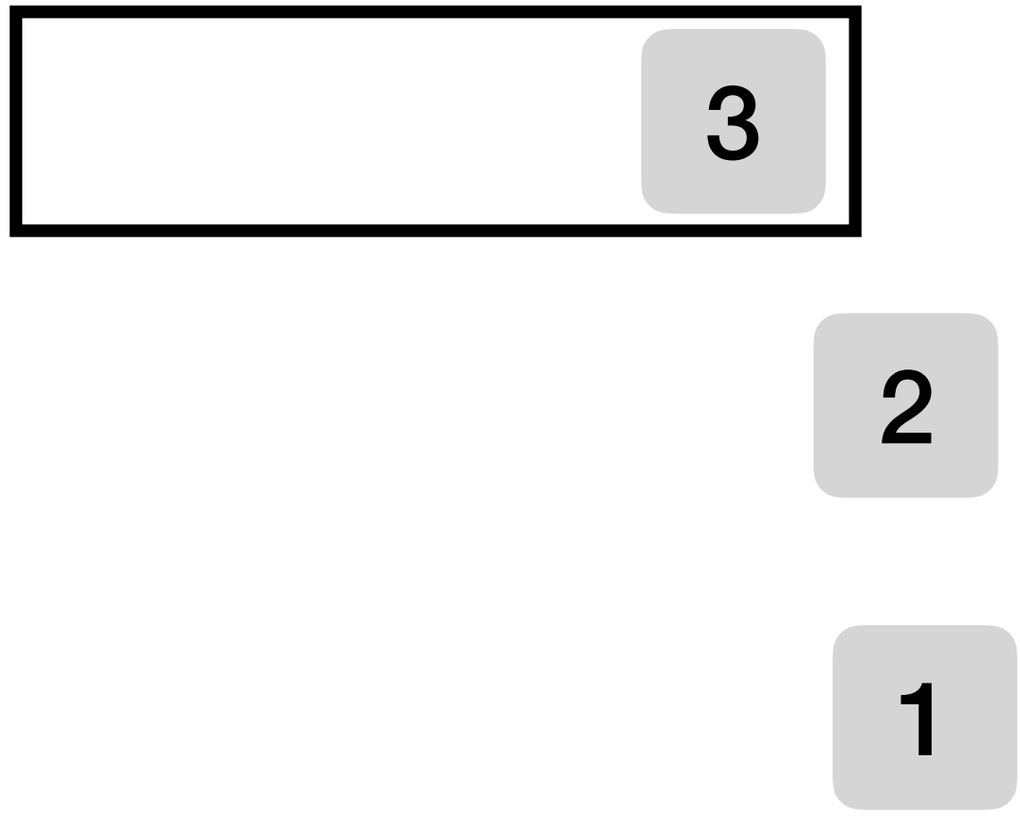
```
enqueue(1)
enqueue(2)
enqueue(3)
 dequeue()
 dequeue()
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

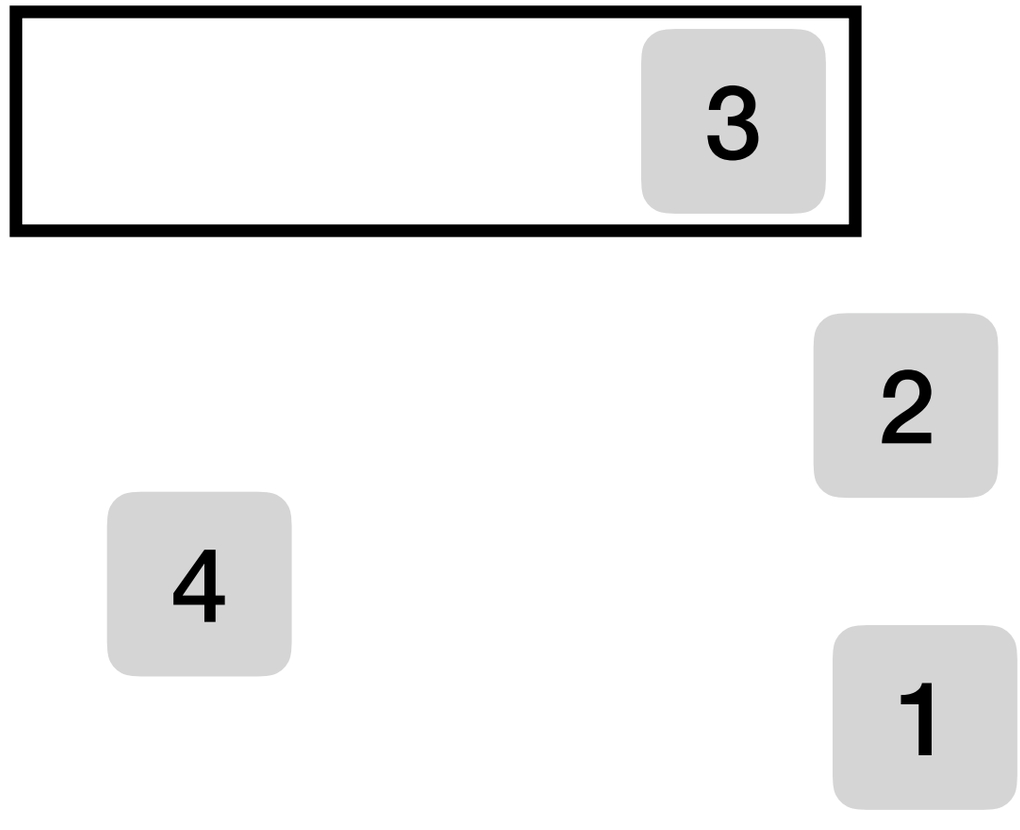
```
enqueue(1)
enqueue(2)
enqueue(3)
 dequeue()
 dequeue()
```



Queues

- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

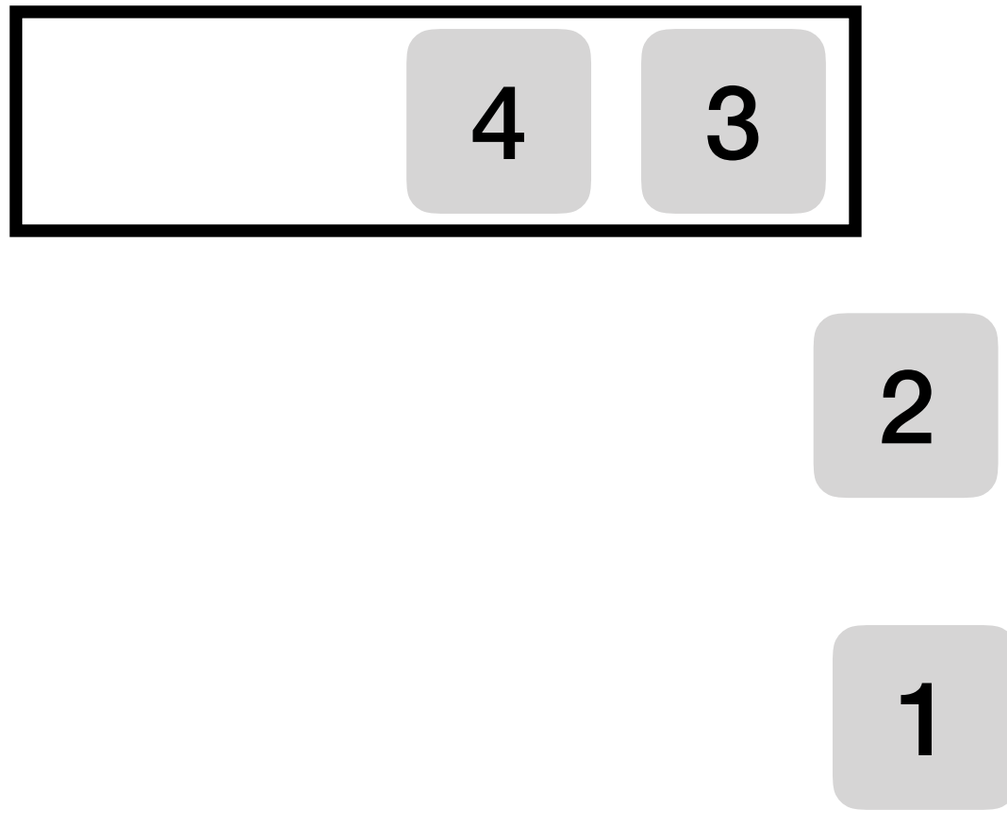
```
enqueue(1)
enqueue(2)
enqueue(3)
 dequeue()
 dequeue()
enqueue(4)
```



Queues

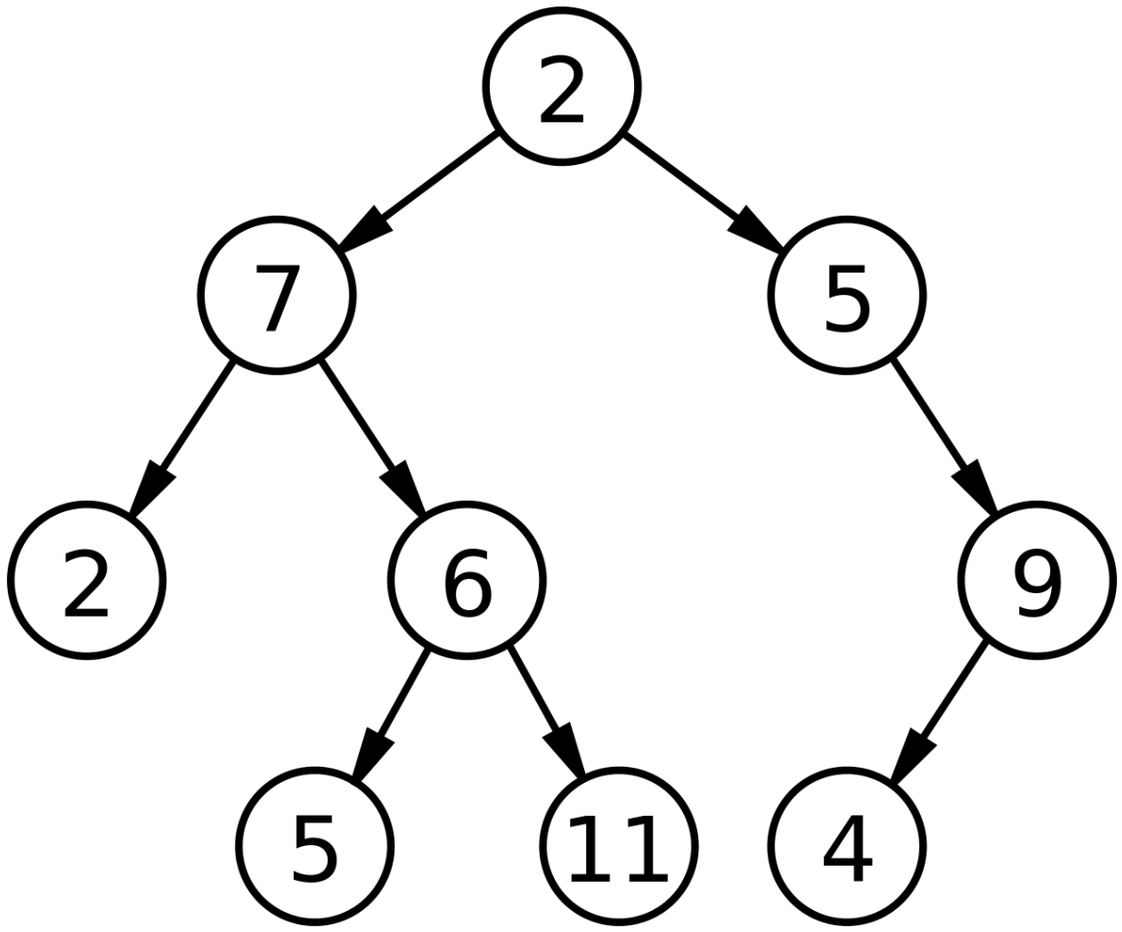
- A queue is also a data type that stores a collection of items.
- You can ***enqueue*** (add) items onto the back of the queue and ***dequeue*** (remove) things off the front of the queue.
- Think of it like a line of people: you get in line at the back and then have to wait until it is your turn.

```
enqueue(1)
enqueue(2)
enqueue(3)
 dequeue()
 dequeue()
enqueue(4)
```



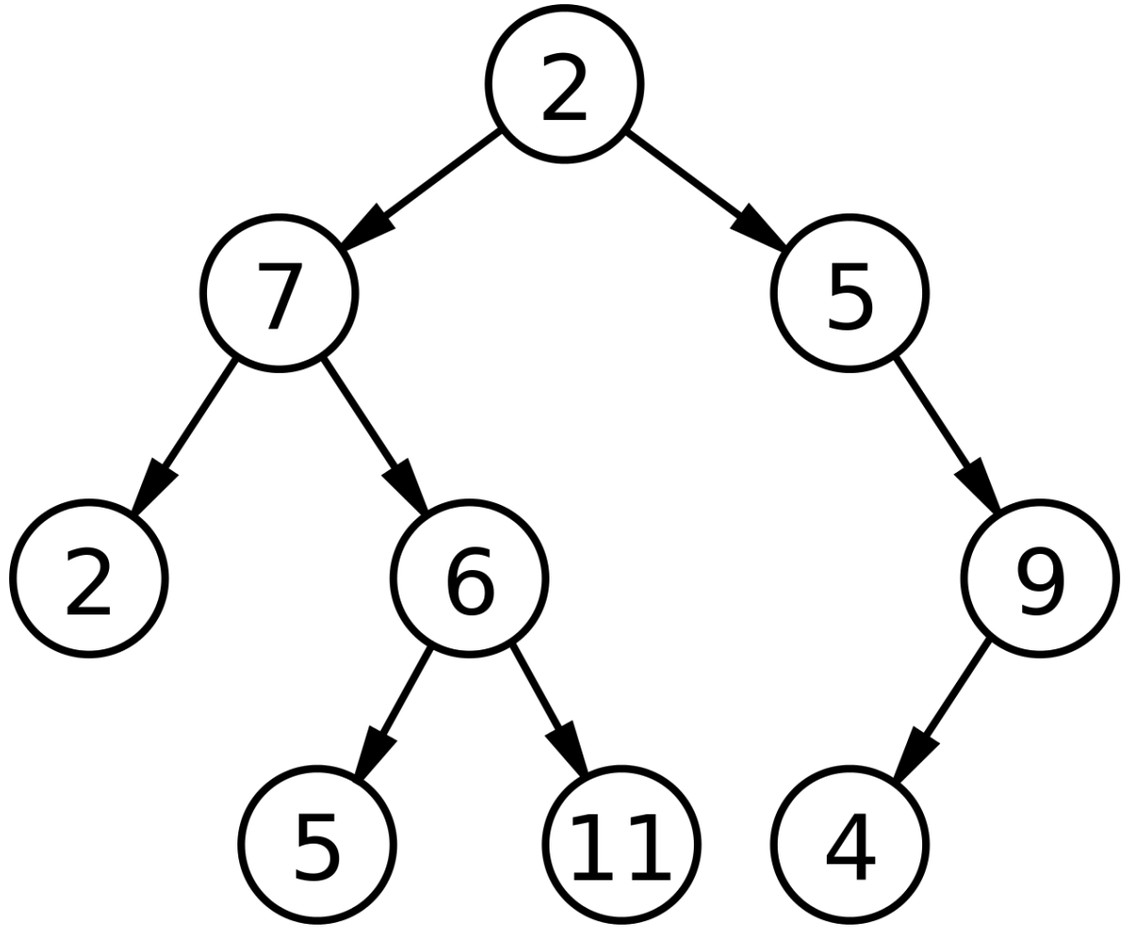
Iterative Traversals

- These data structures are great for traversing the tree iteratively!



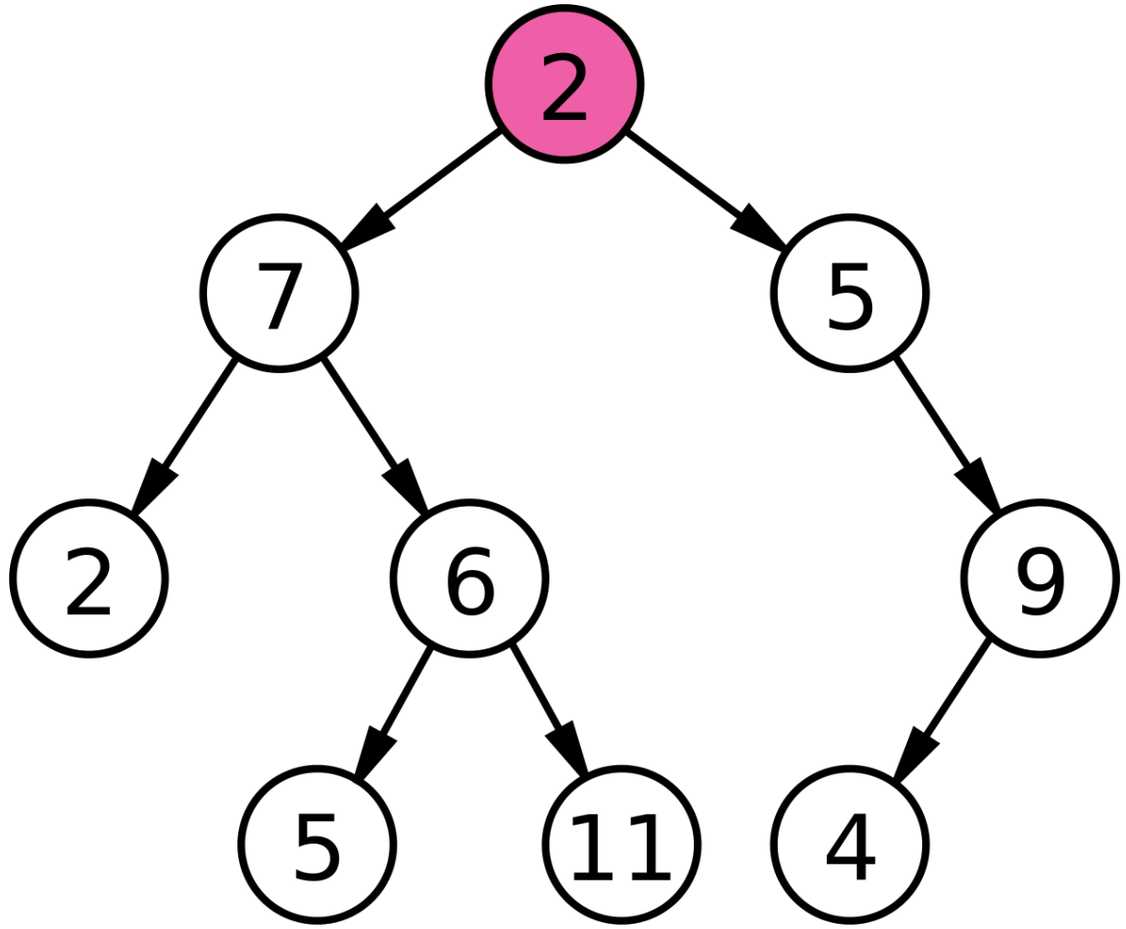
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



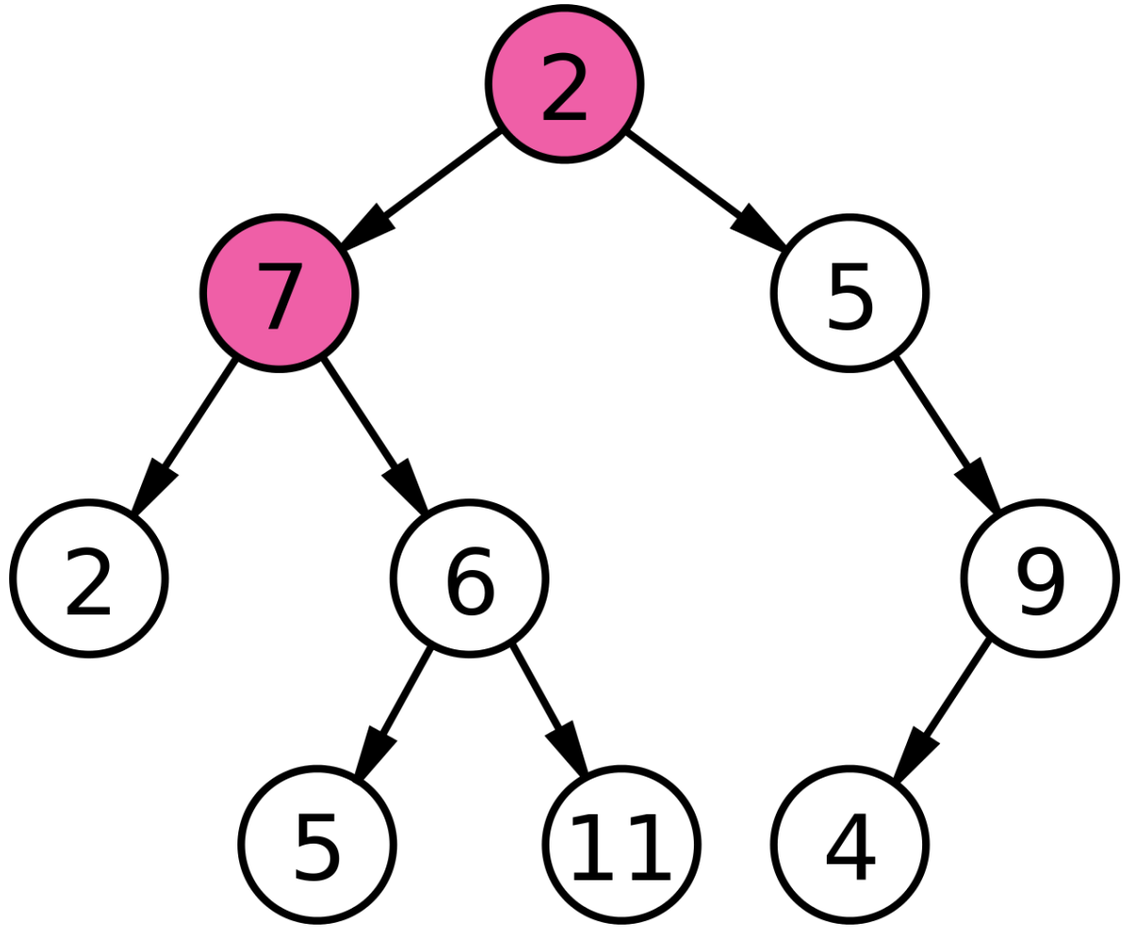
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



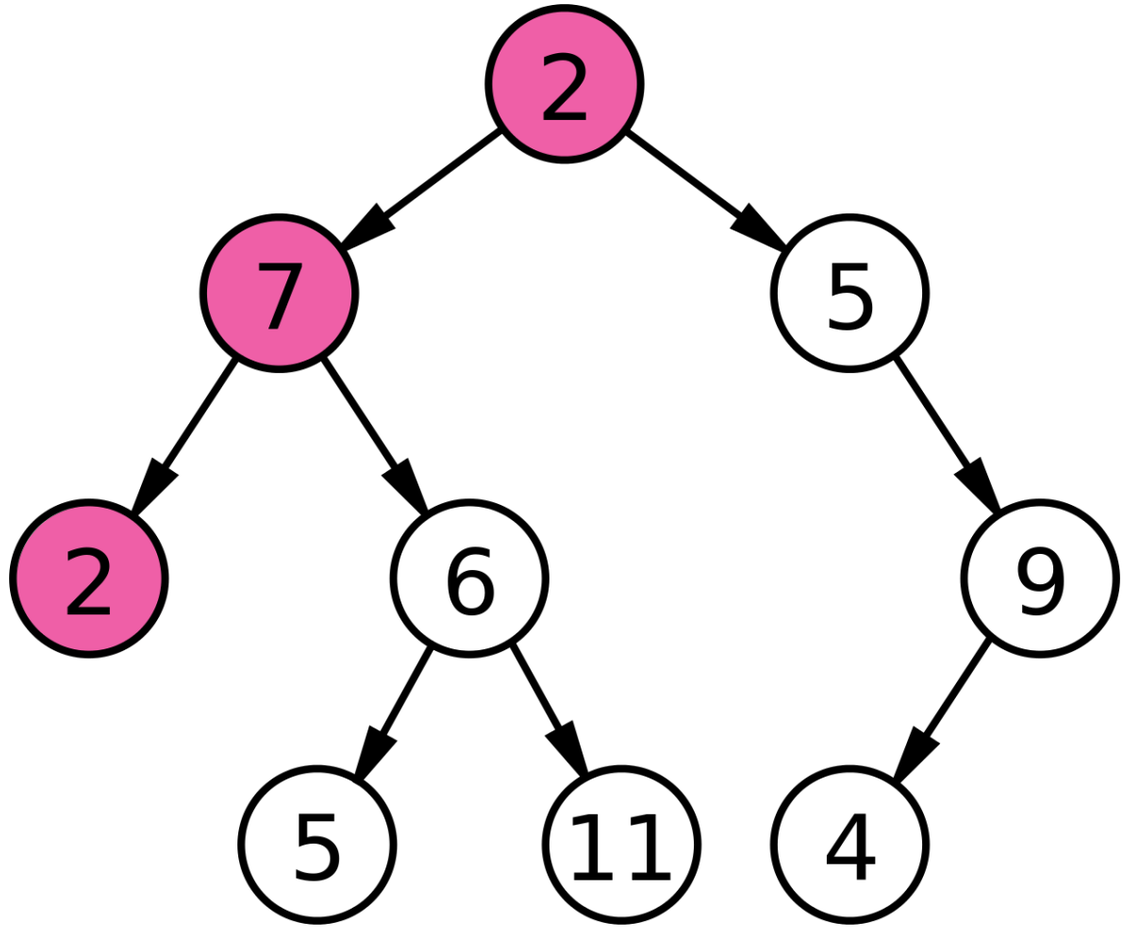
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



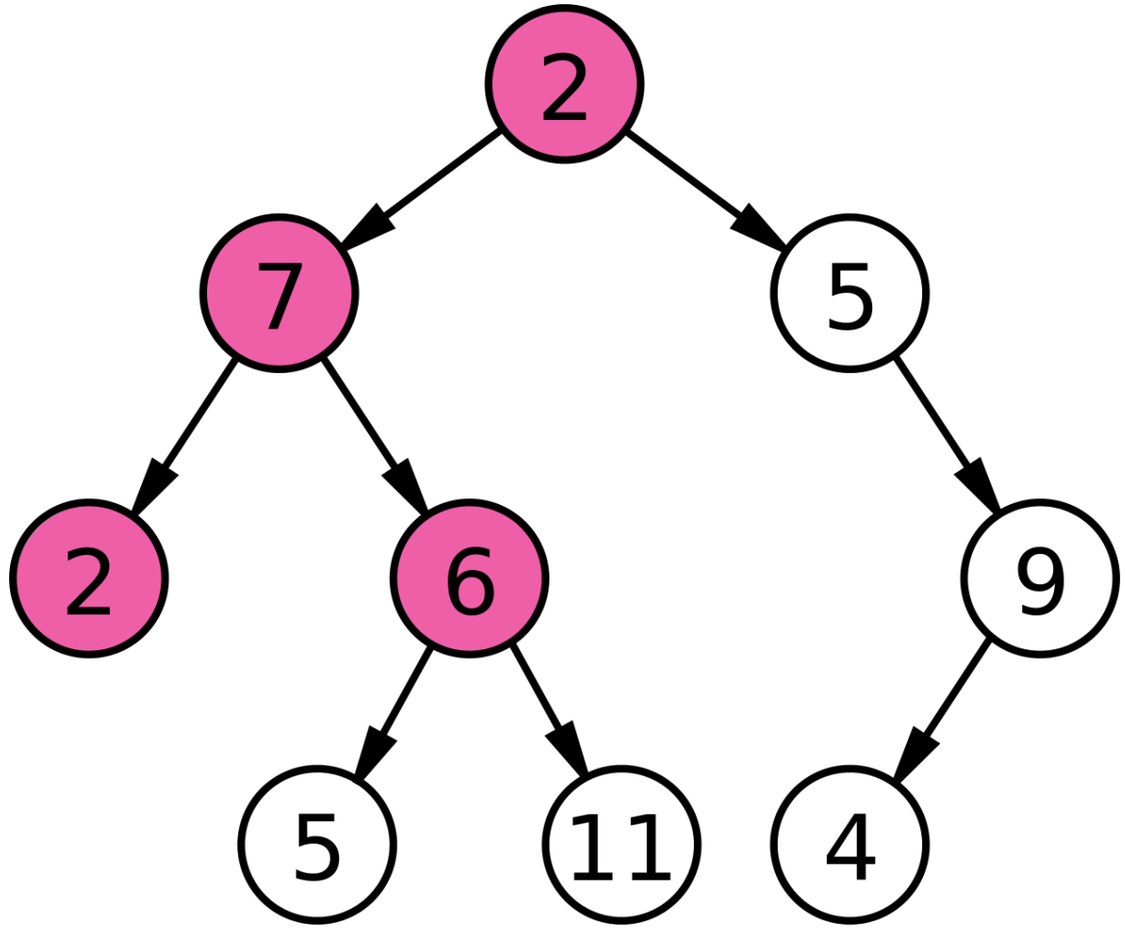
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



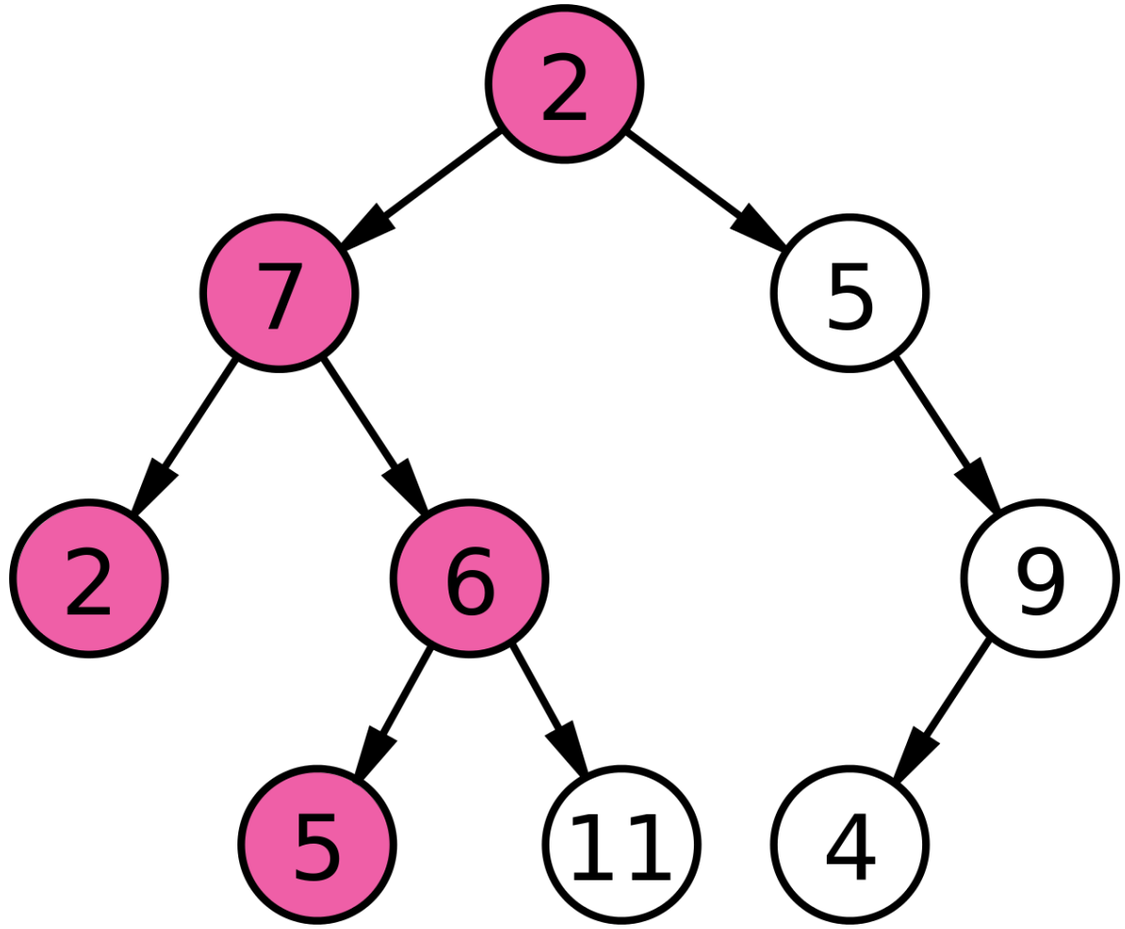
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



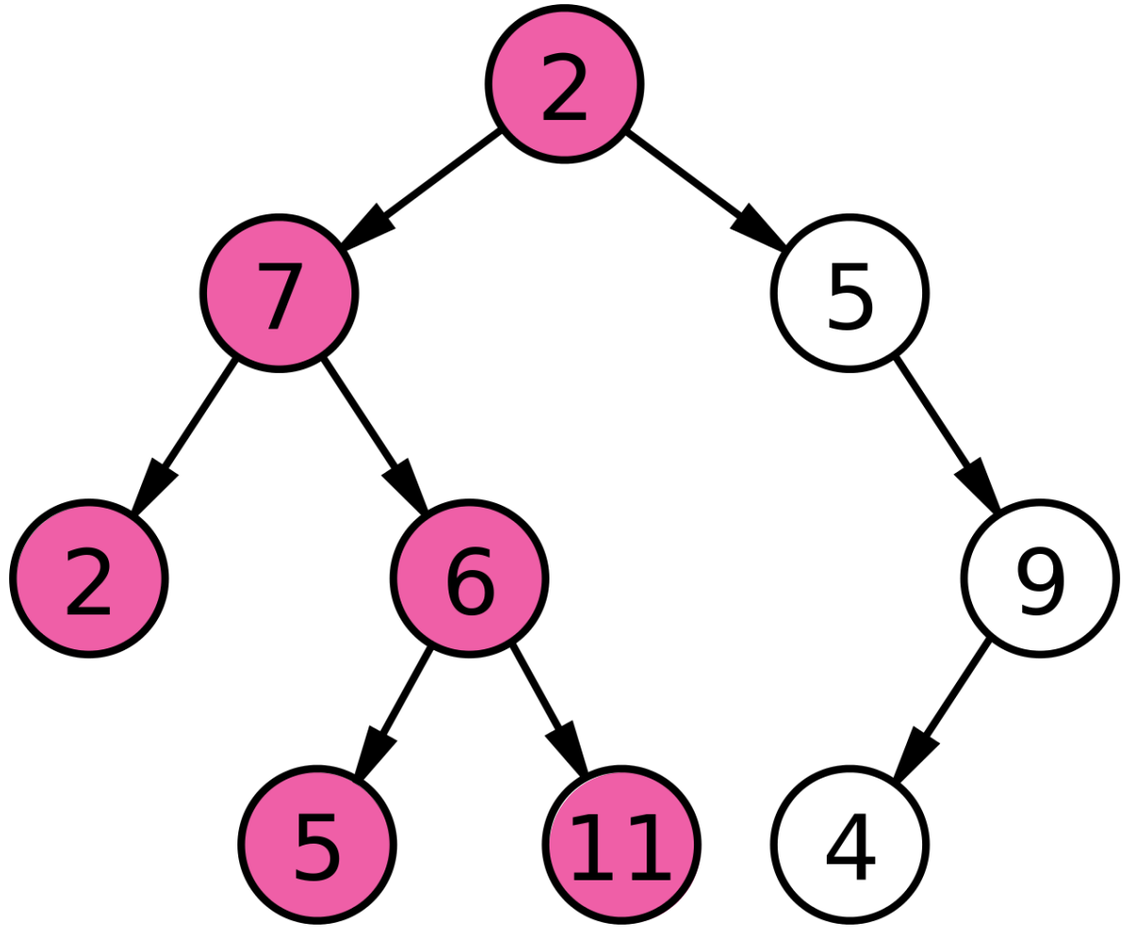
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



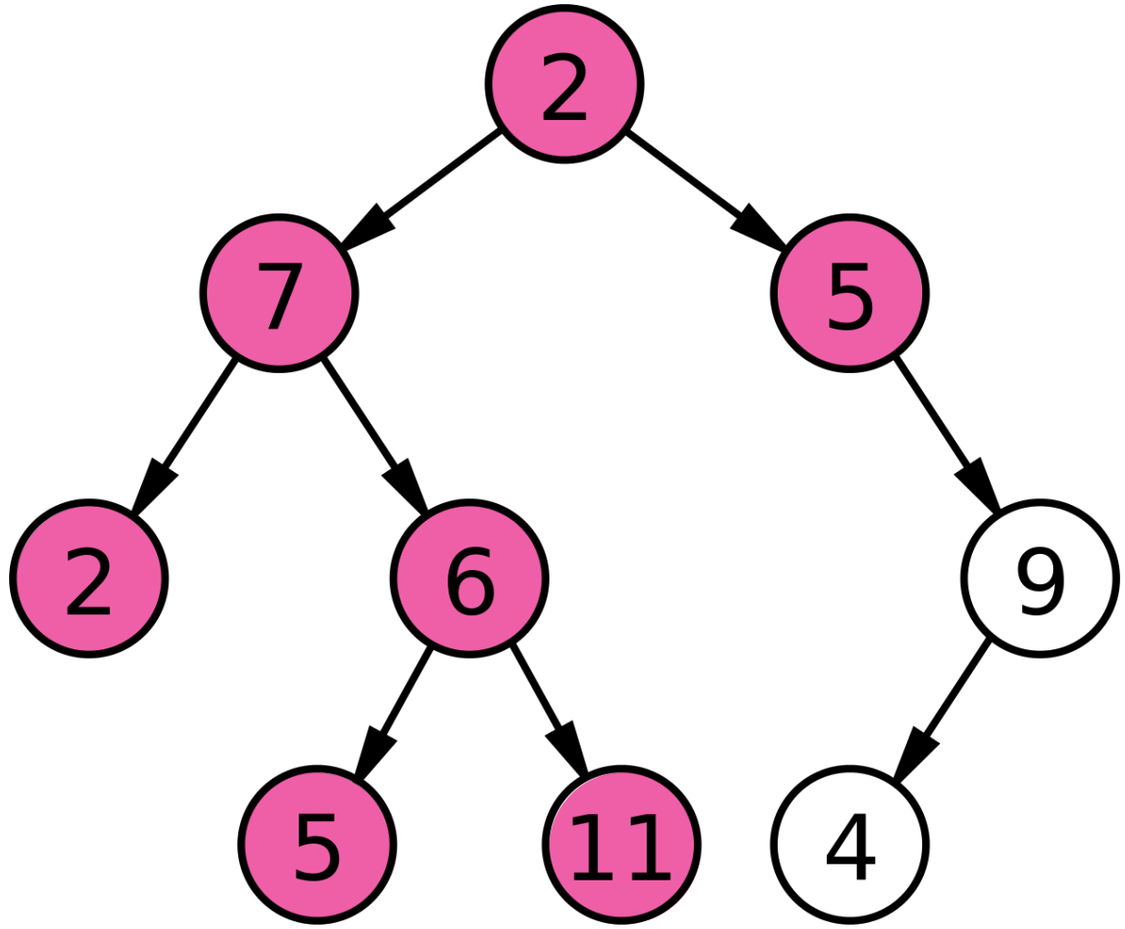
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



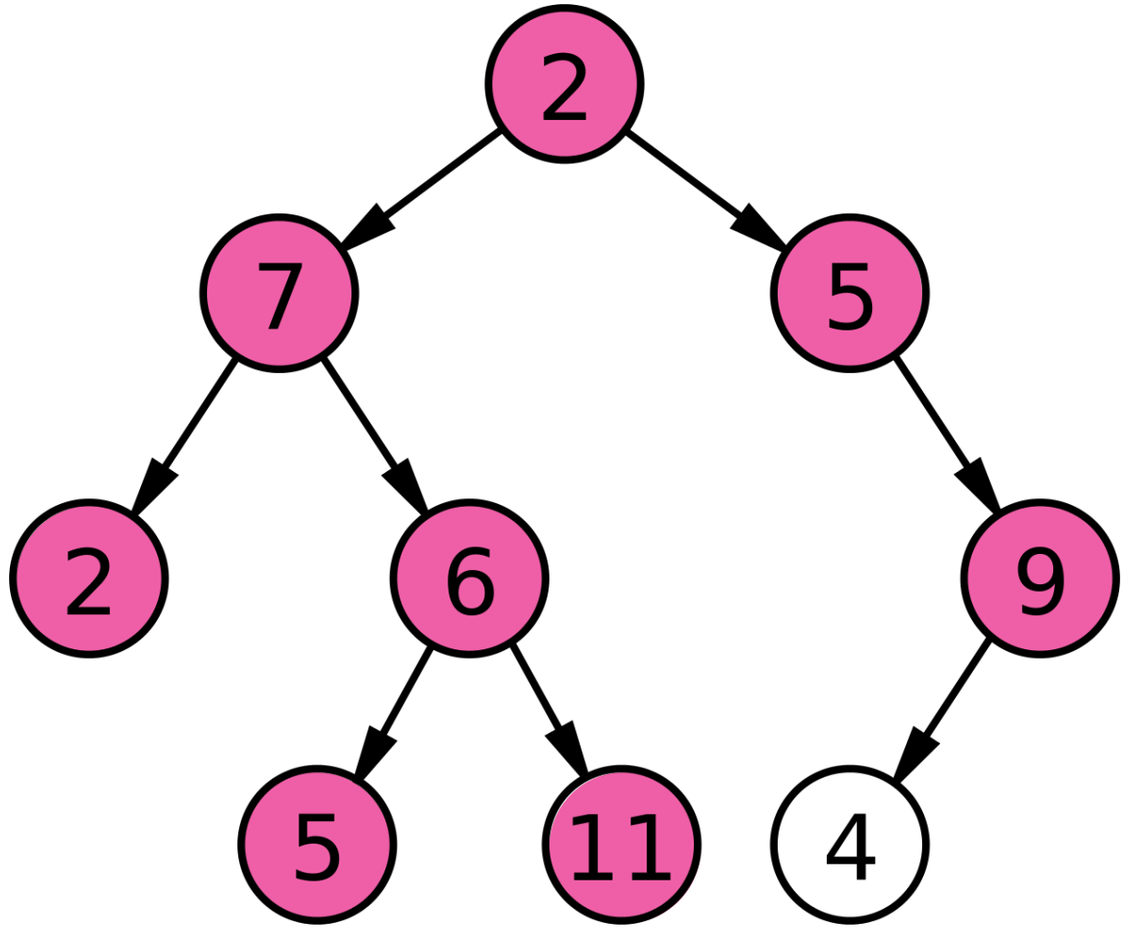
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



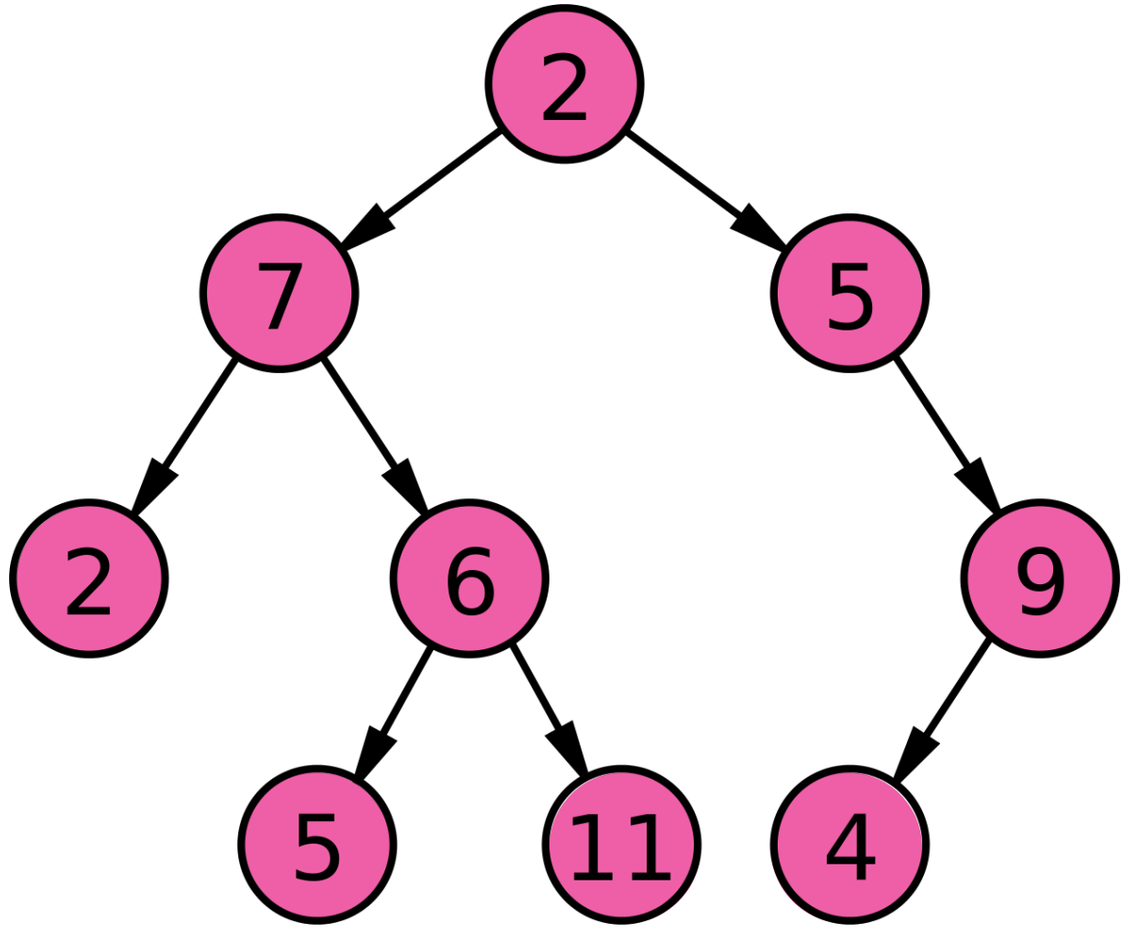
Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



Depth-First Search (DFS)

- Visit a node, and then completely visit each child subtree before moving on to the next child.



Pseudocode

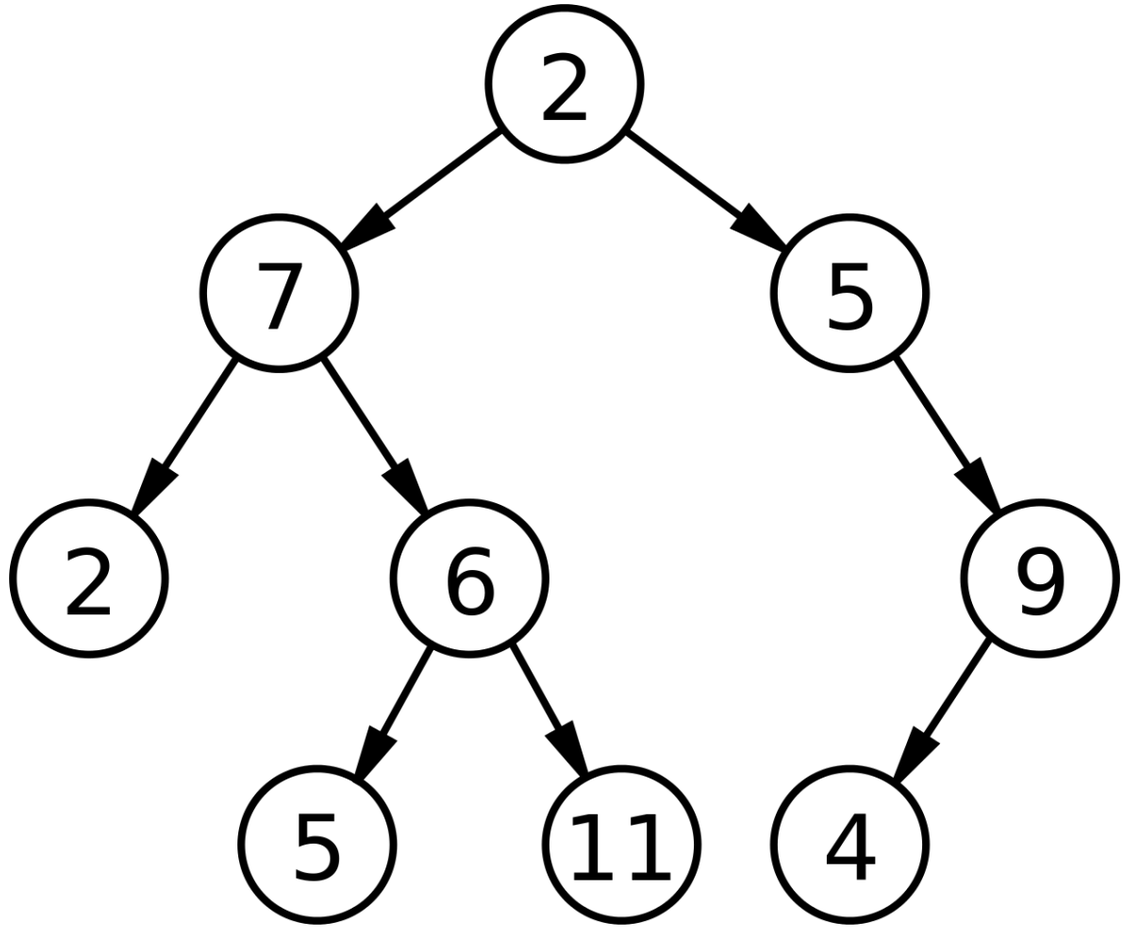
```
initialize a stack called the "fringe"

add root to the fringe

while the fringe is not empty:
    pop the topmost node from the fringe
    push all of its children onto the fringe
    Process removed node
```

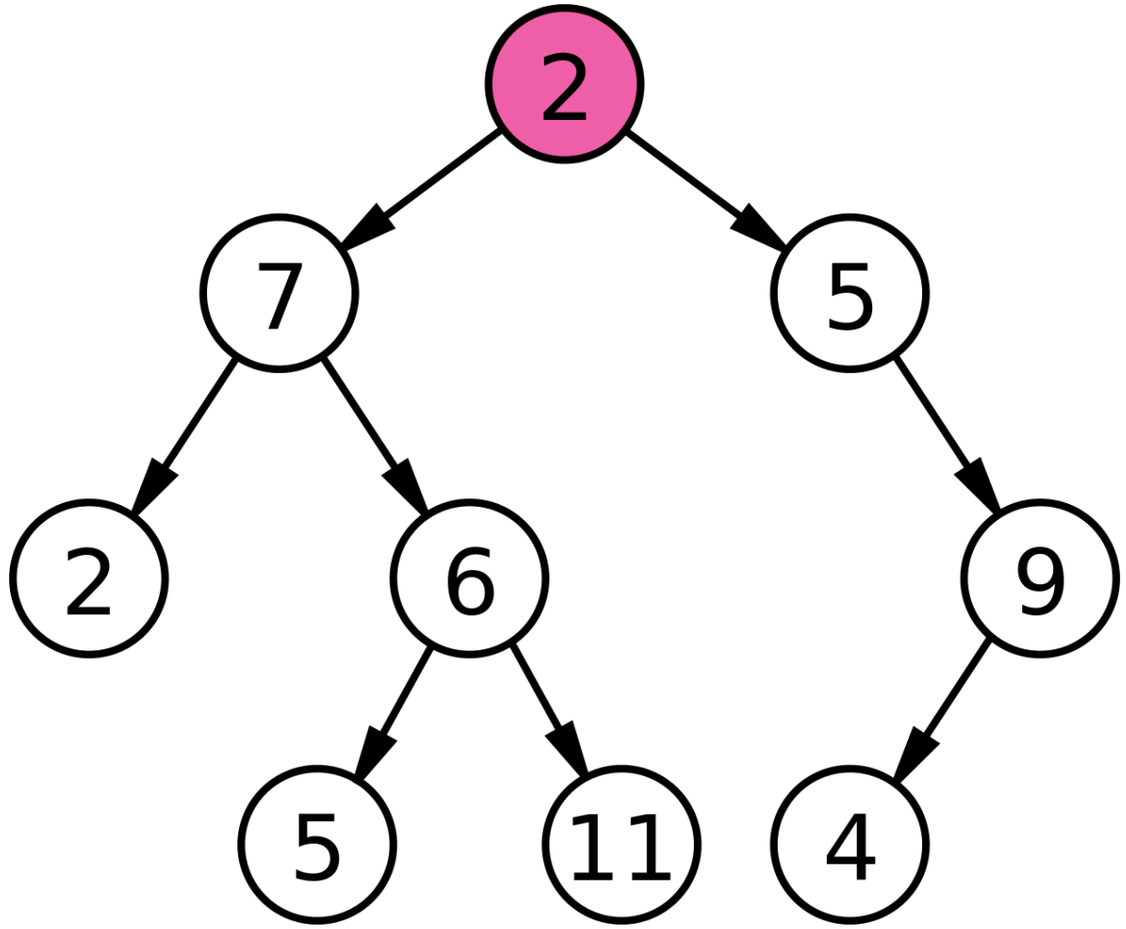
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



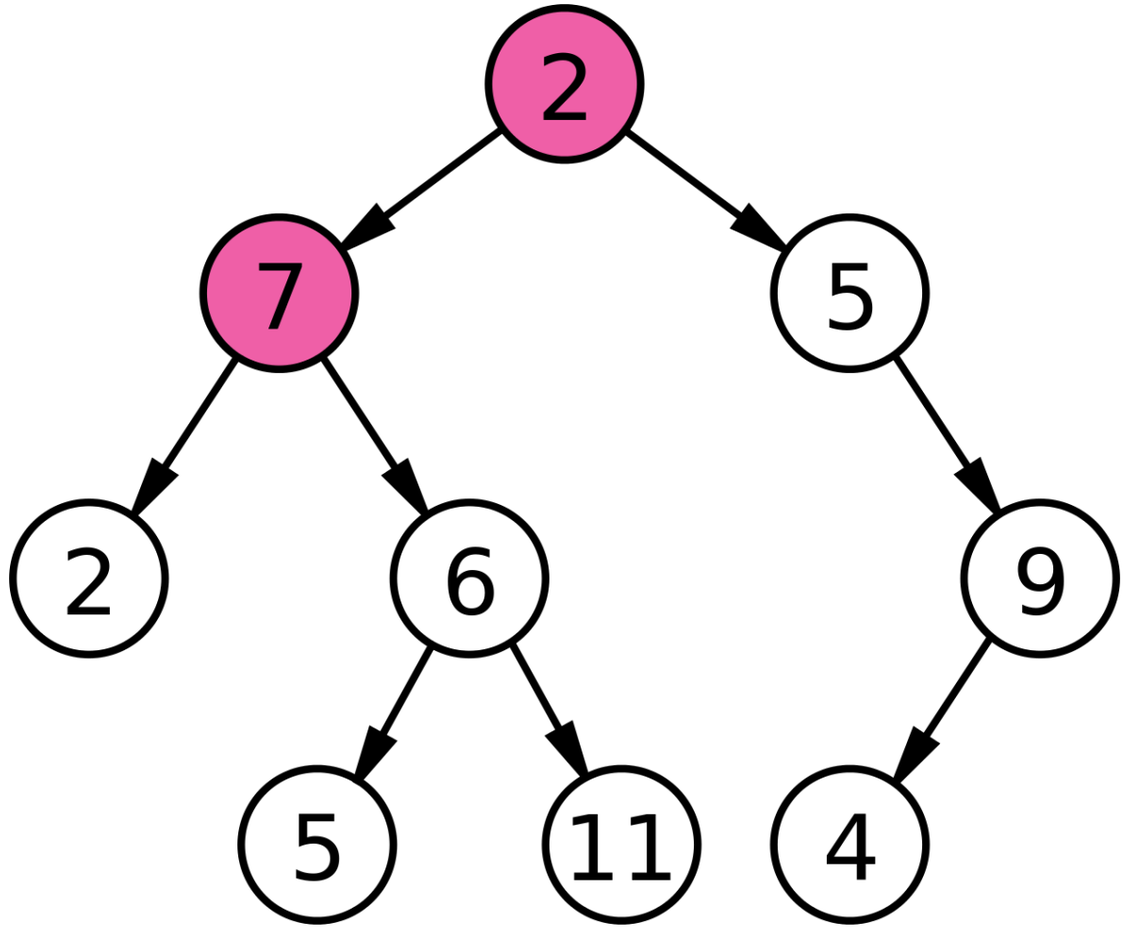
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



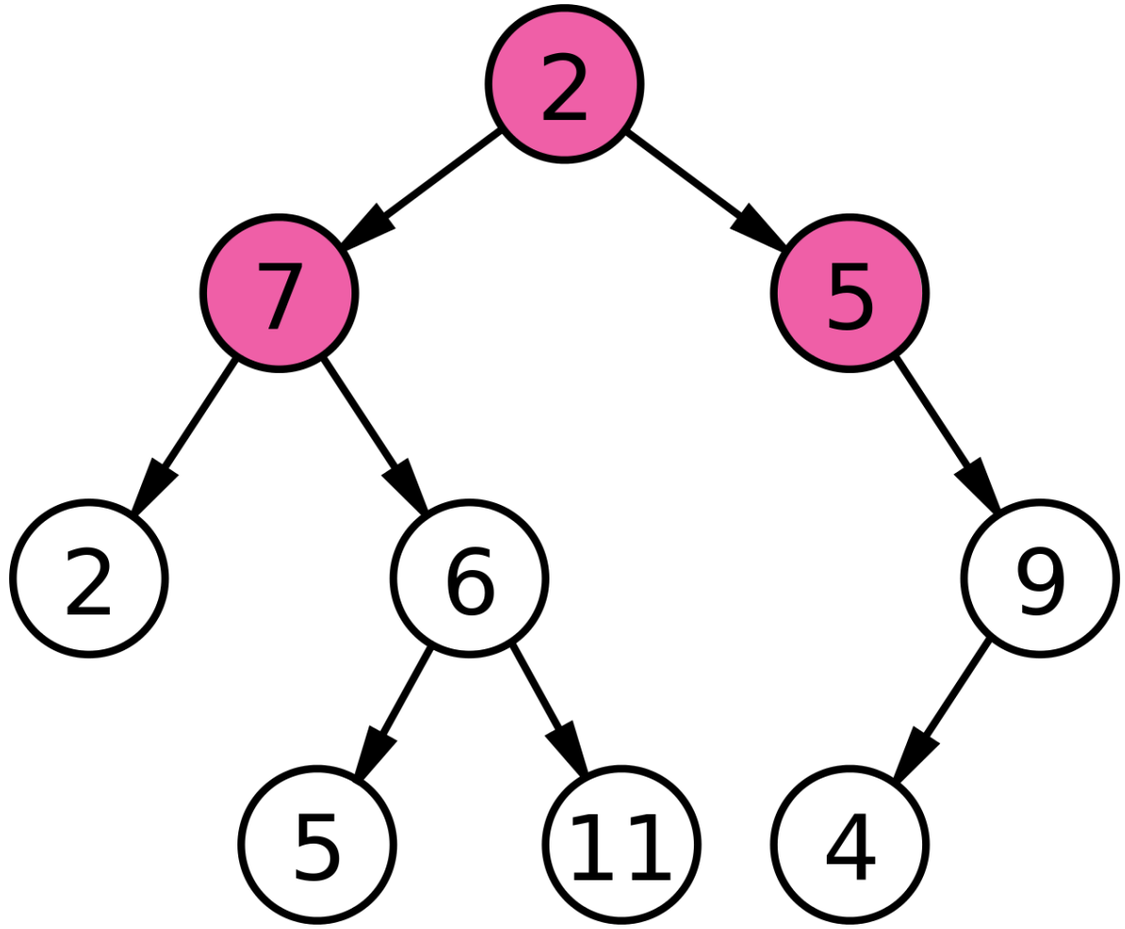
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



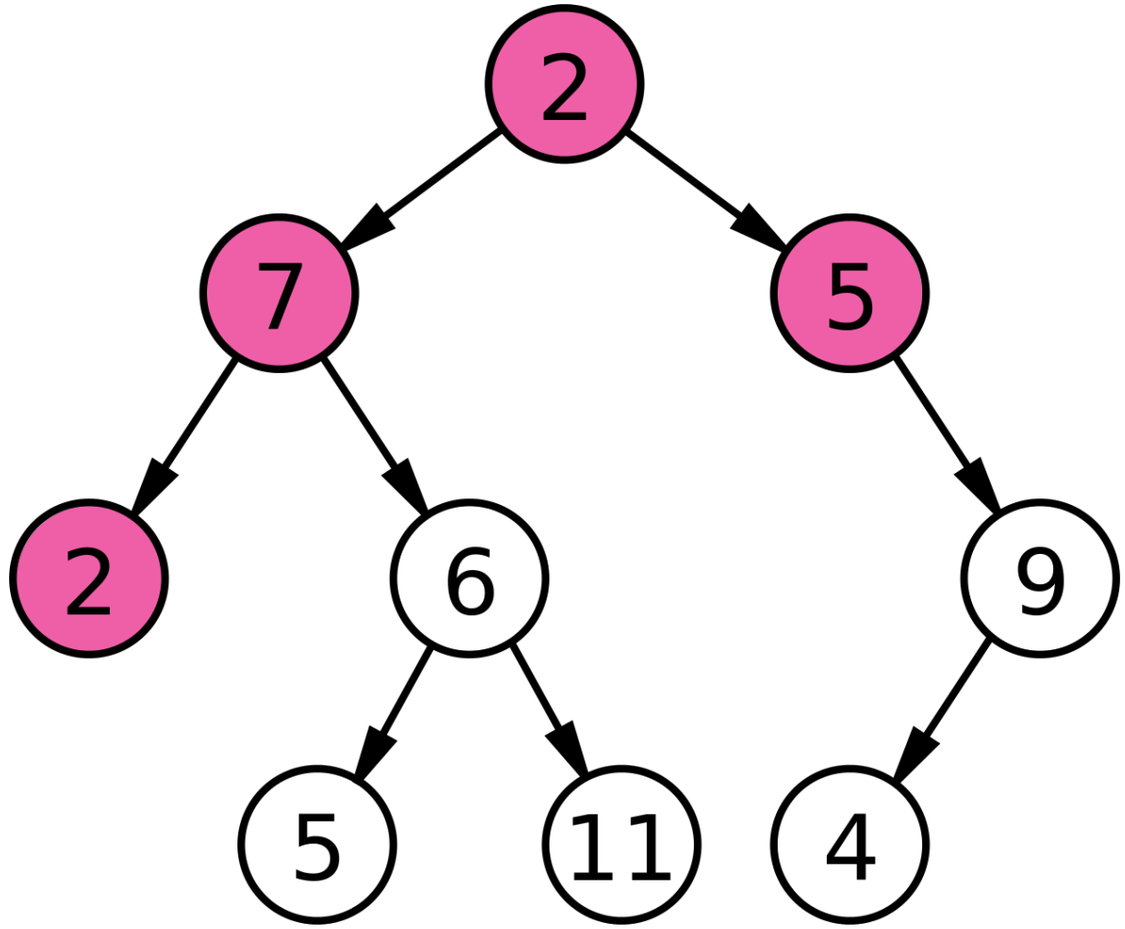
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



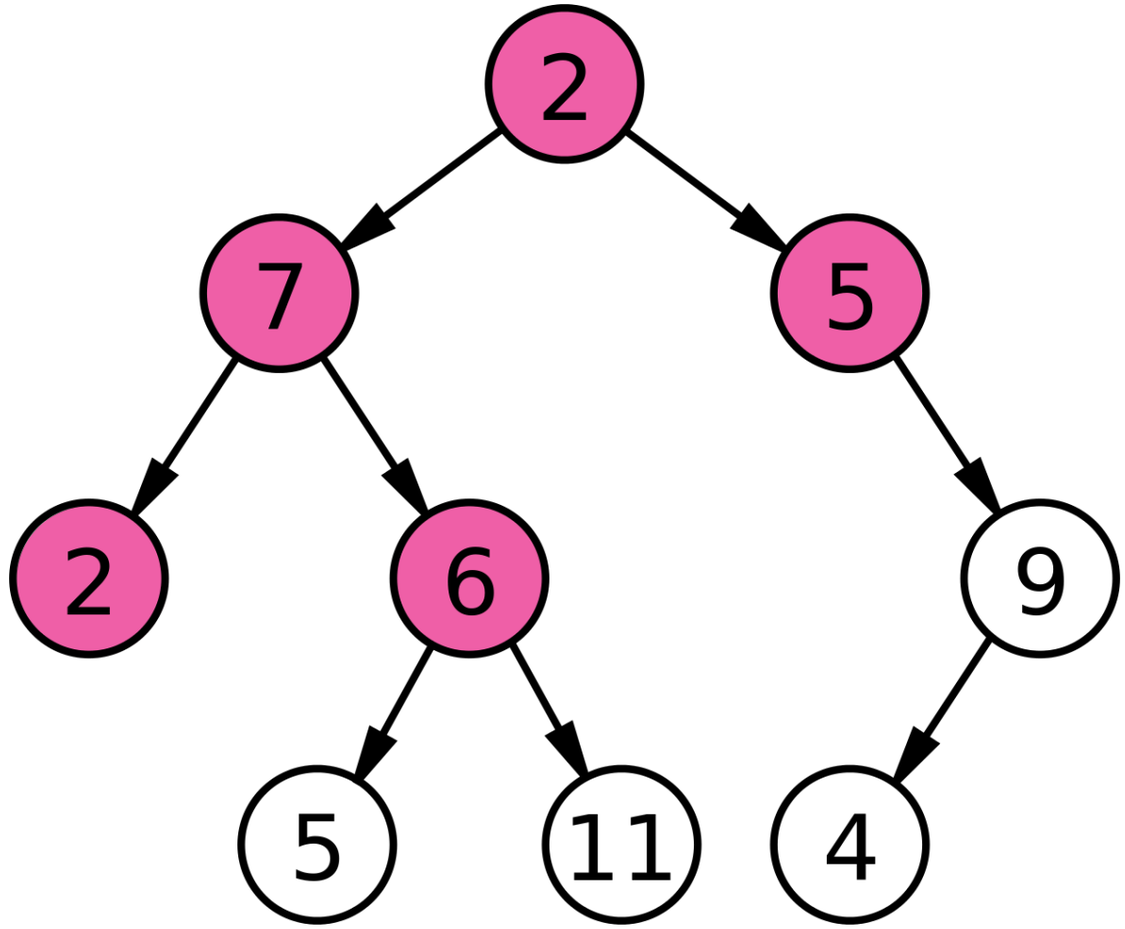
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



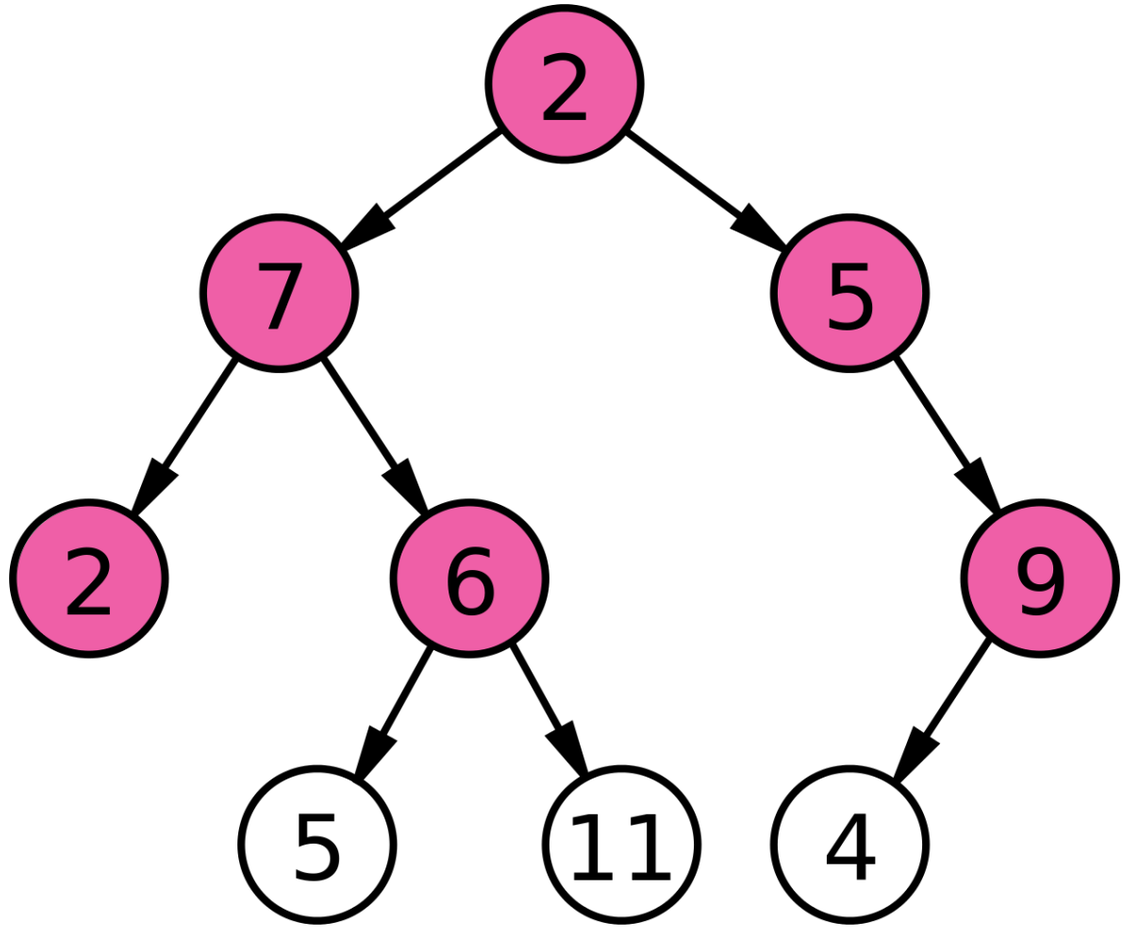
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



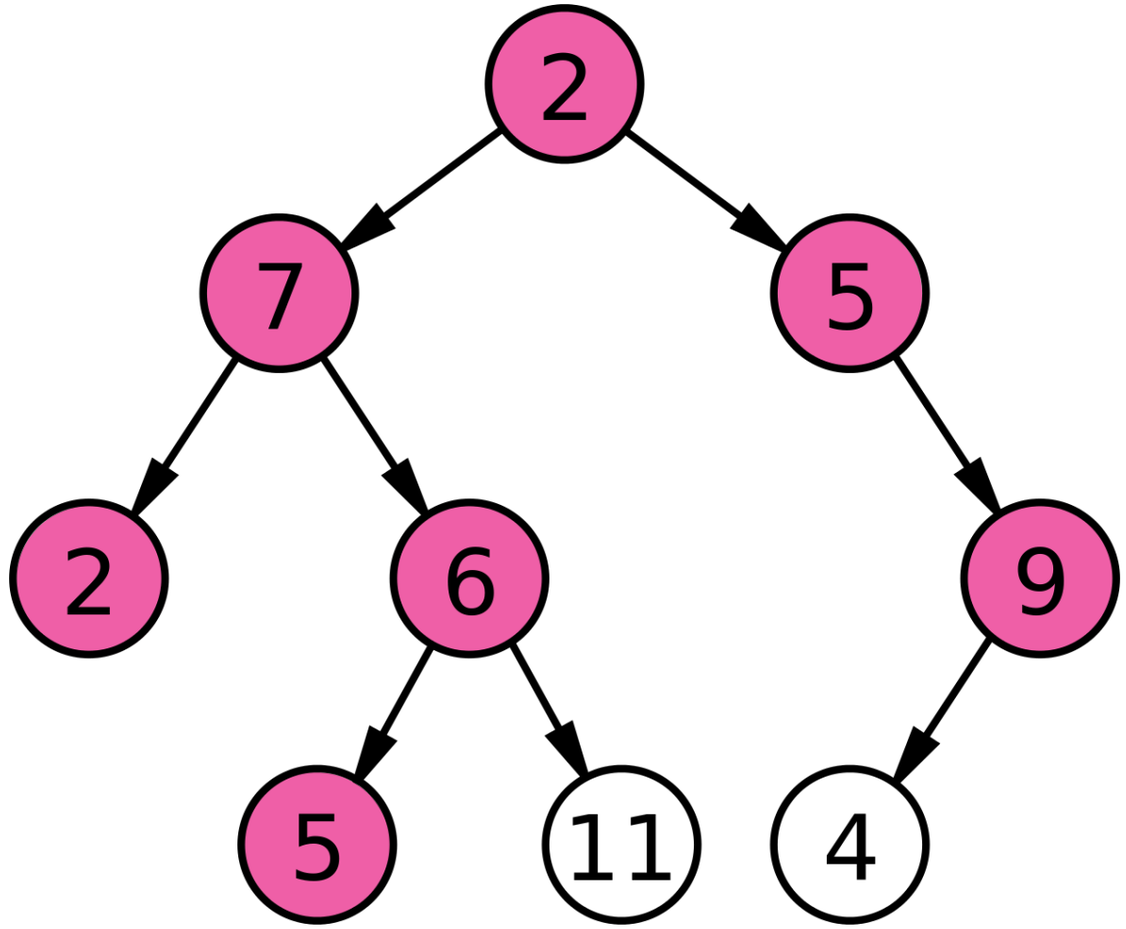
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



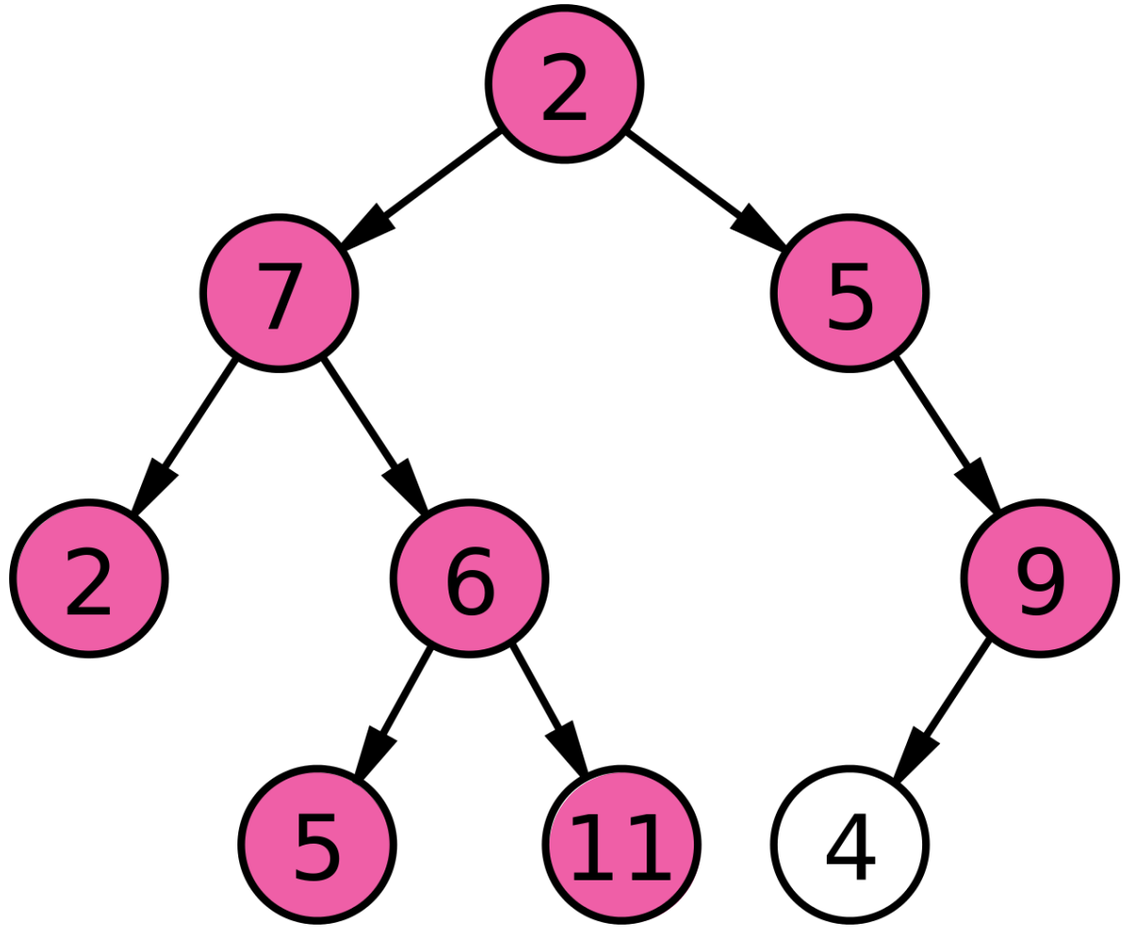
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



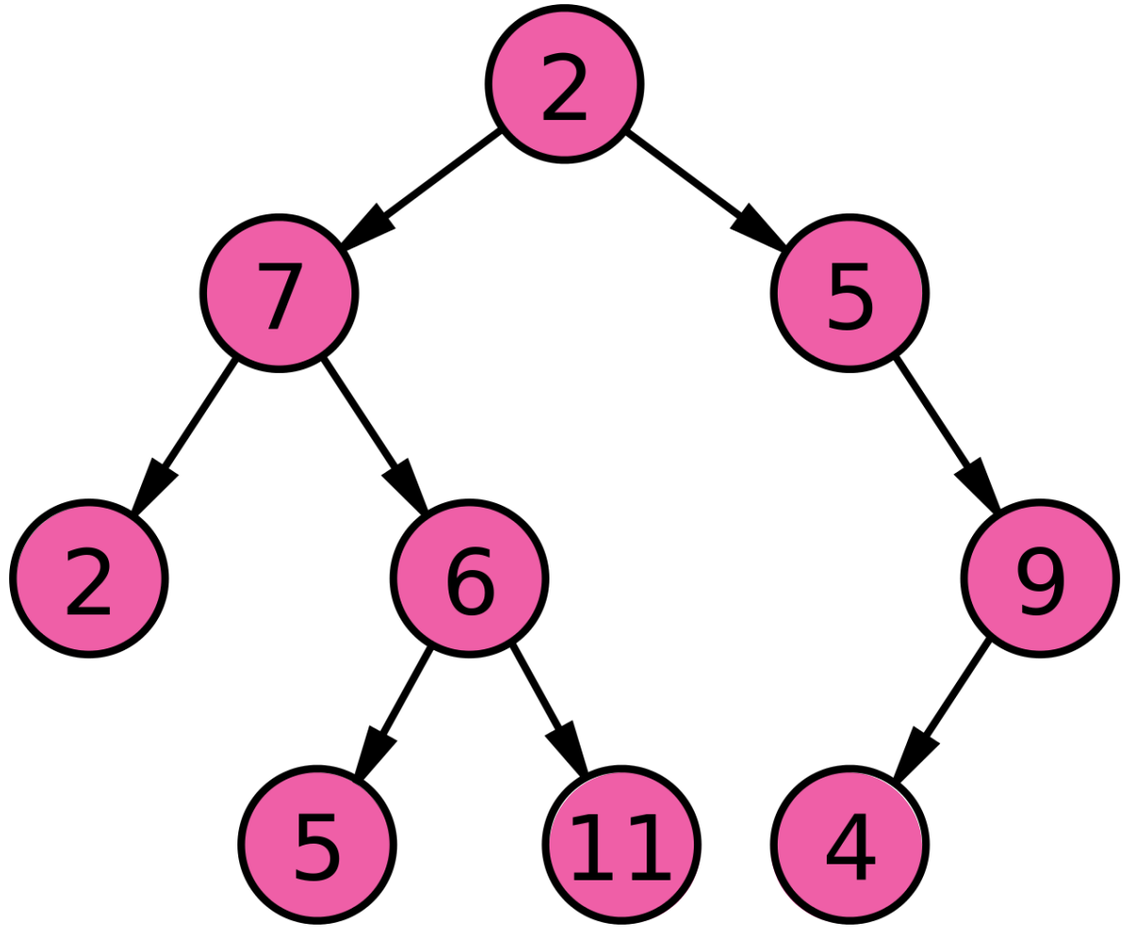
Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



Breadth-First Search (DFS)

- Visit a node, visit each children, and then visit the children's children.



Pseudocode

```
initialize a queue called the "fringe"

add root to the fringe

while the fringe is not empty:
    dequeue the topmost node from the fringe
    enqueue its children onto the fringe
    Process removed node
```