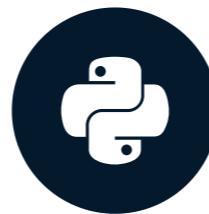


Linear Search and Binary Search

DATA STRUCTURES AND ALGORITHMS IN PYTHON



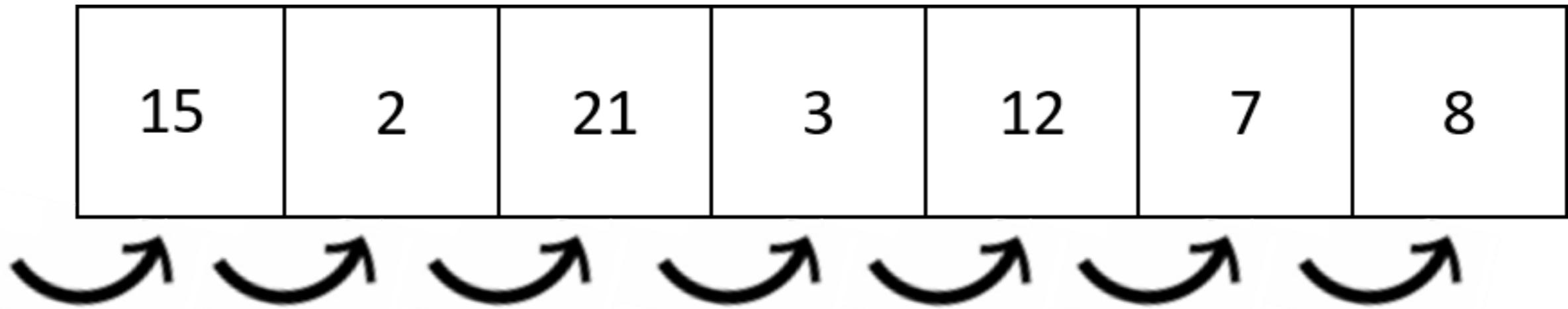
Miriam Antona
Software engineer

Searching algorithms

- **Searching** is an essential operation
 - Several ways
- Algorithms that search for an element within a collection:
 - **Linear search**
 - **Binary search**

Linear search

- Looping through each element



- Element found
 - algorithm stops
 - returns the result
- Element not found
 - algorithm continues

Linear search

```
def linear_search(unordered_list, search_value):  
    for index in range(len(unordered_list)):  
        if unordered_list[index] == search_value:  
            return True  
    return False
```

```
print(linear_search([15,2,21,3,12,7,8], 8))
```

True

```
print(linear_search([15,2,21,3,12,7,8], 800))
```

False

Linear search - complexity

- Complexity: $O(n)$

Binary search

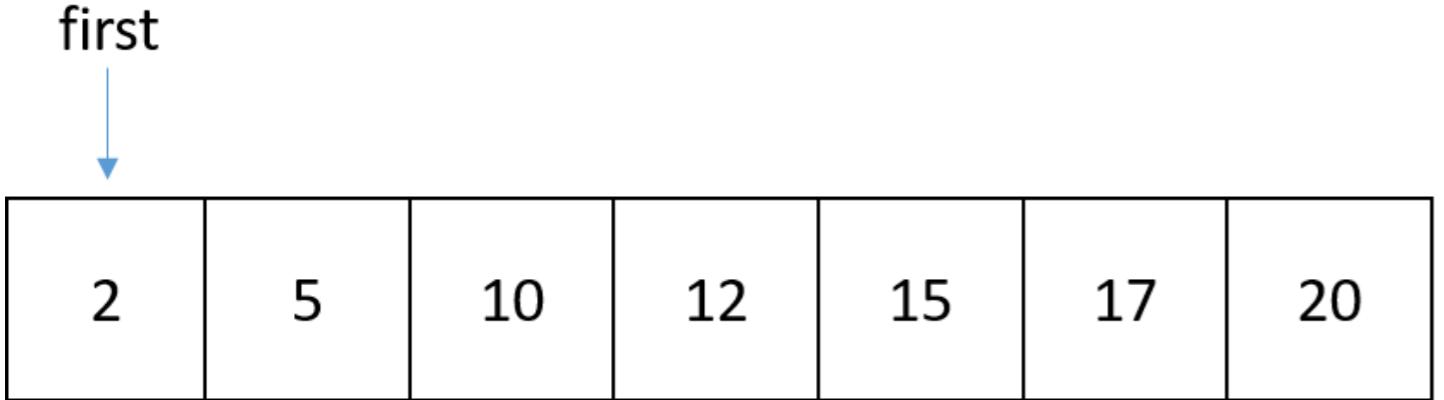
- Only applies to ordered lists

2	5	10	12	15	17	20
---	---	----	----	----	----	----

- 15 ???

Binary search

- Only applies to ordered lists

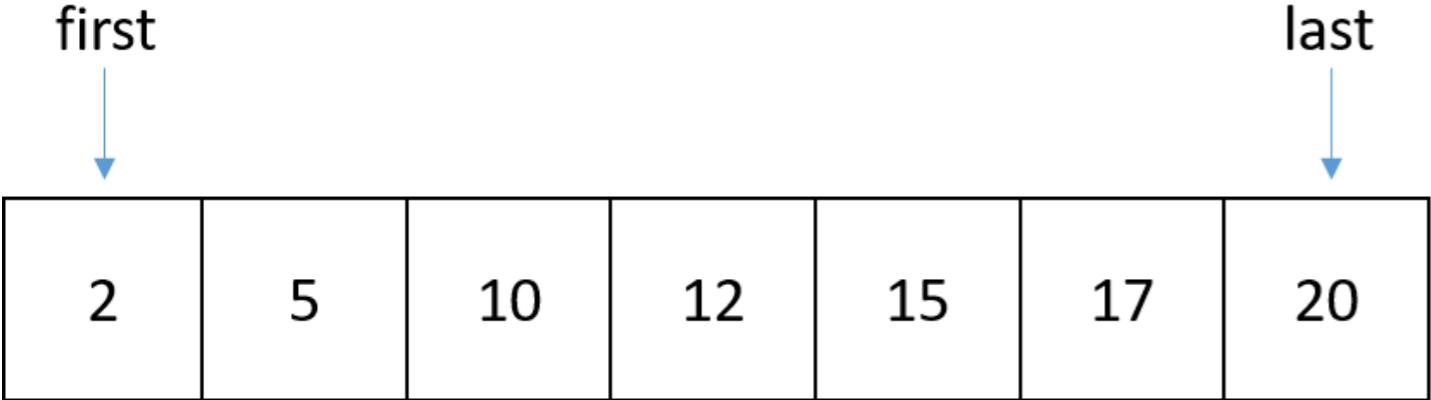


- 15 ???

```
def binary_search(ordered_list, search_value):  
    first = 0
```

Binary search

- Only applies to ordered lists

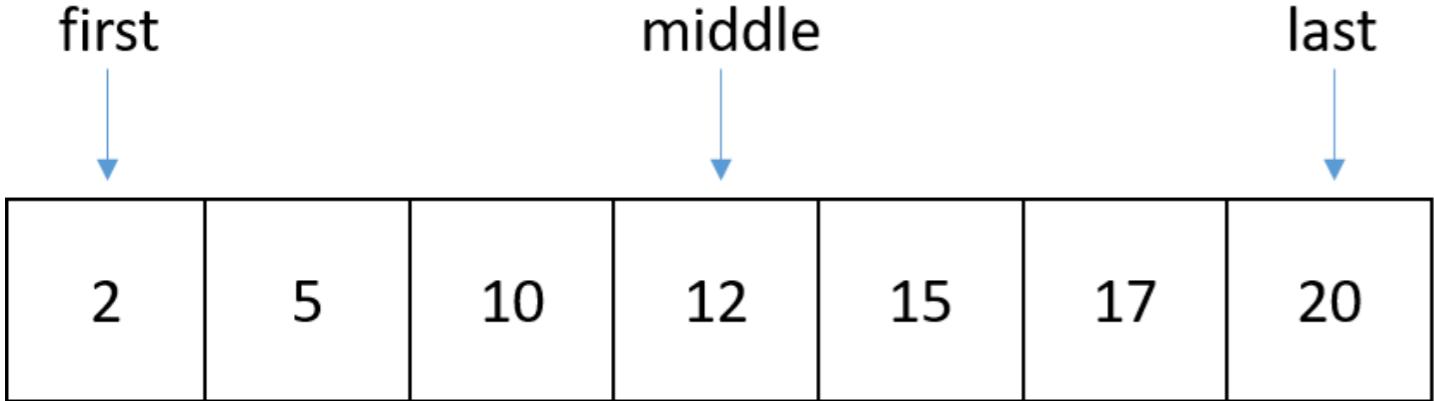


- 15 ???

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:
```

Binary search

- Only applies to ordered lists

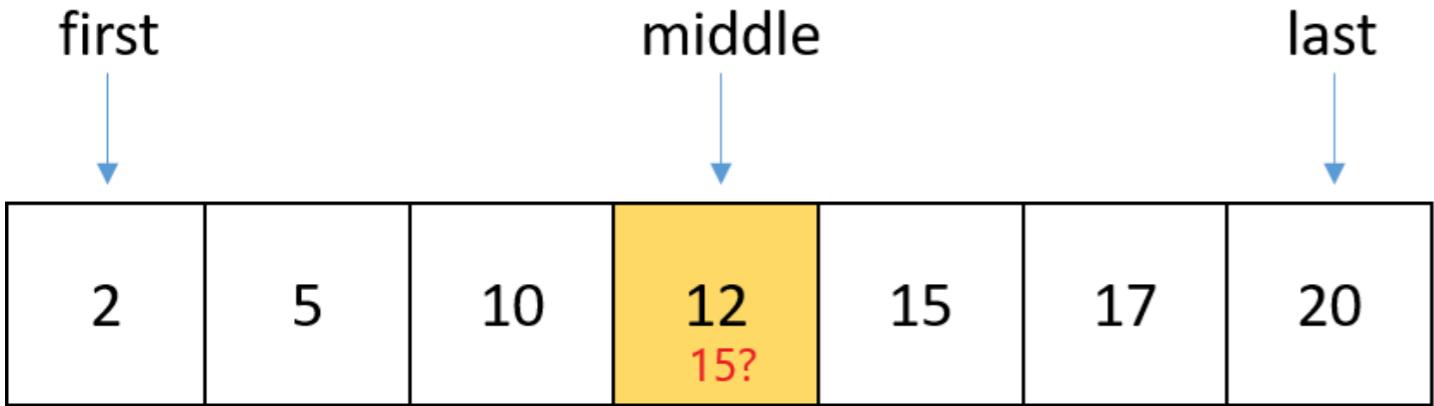


- 15 ???

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2
```

Binary search

- Only applies to ordered lists



- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:
```

Binary search

- Only applies to ordered lists

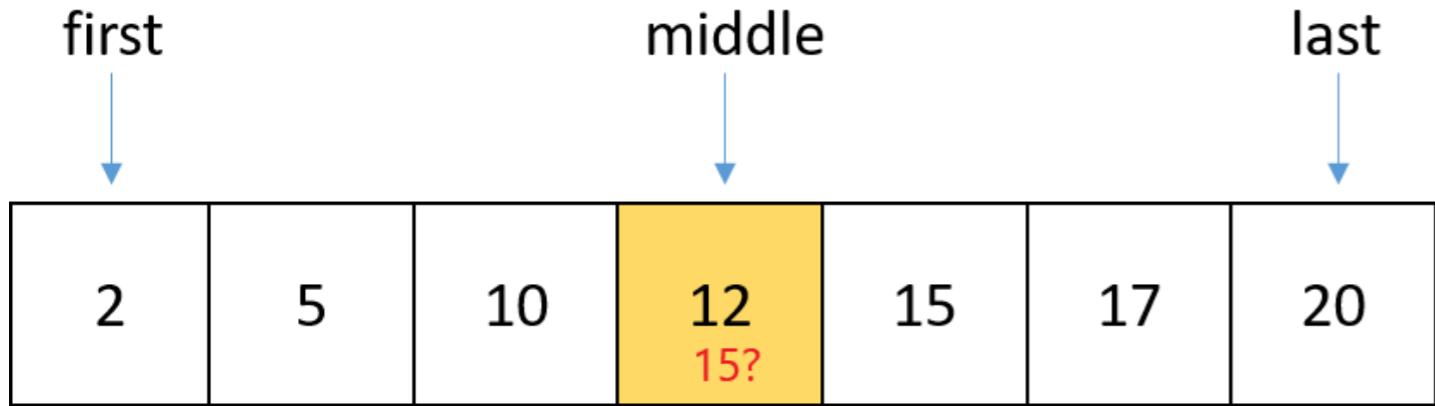


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1
```

Binary search

- Only applies to ordered lists

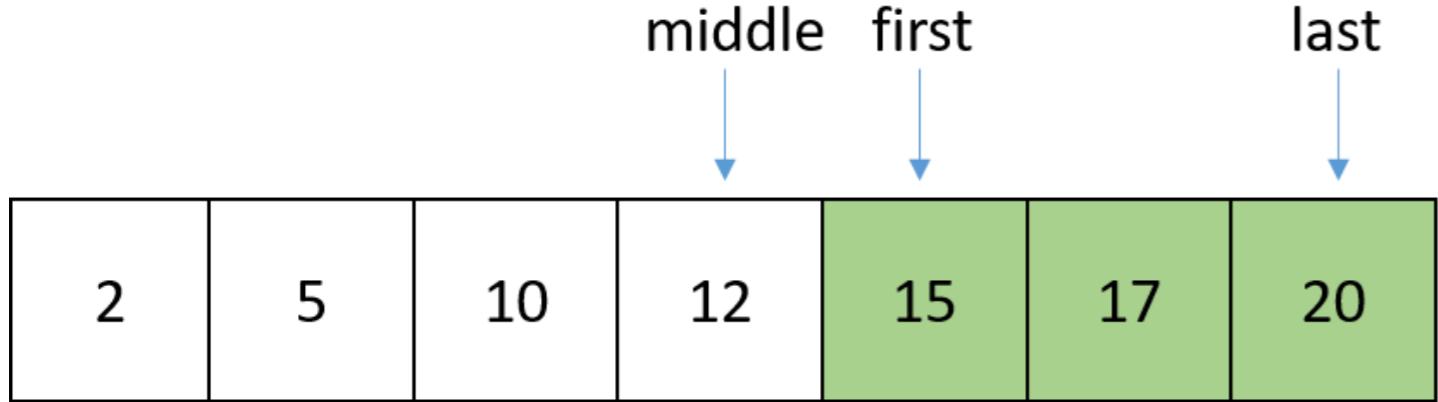


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:
```

Binary search

- Only applies to ordered lists

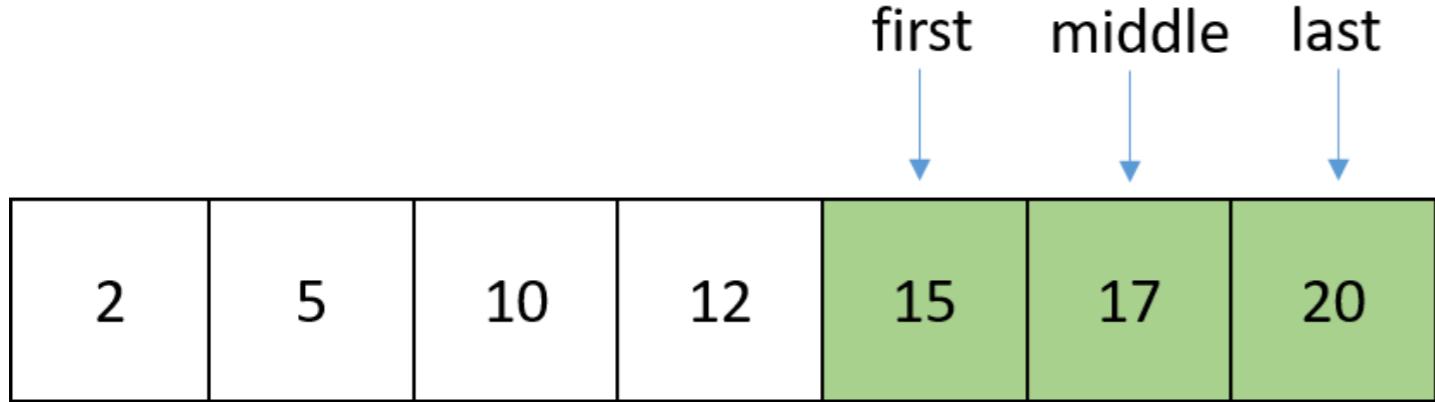


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1
```

Binary search

- Only applies to ordered lists

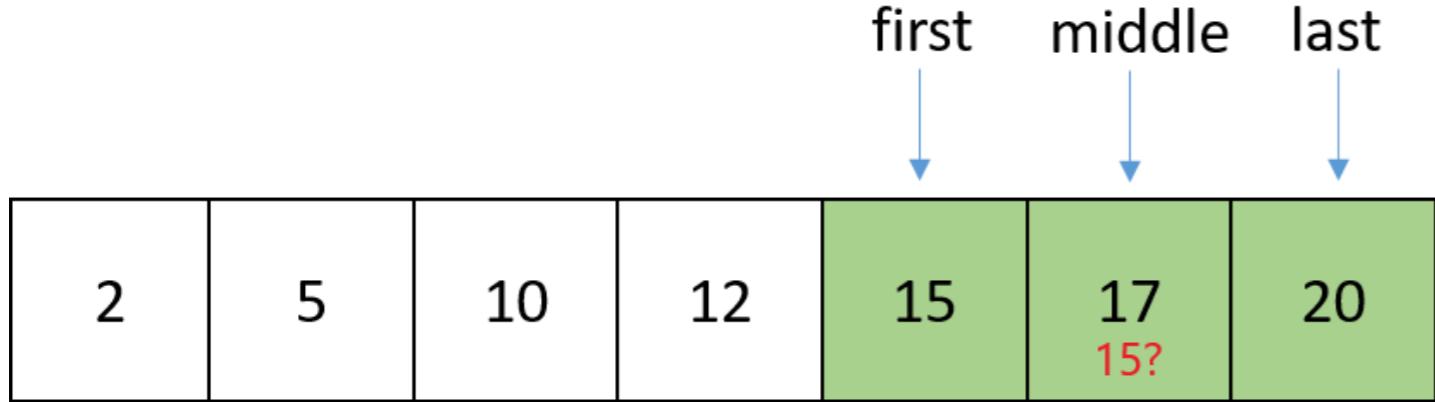


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1
```

Binary search

- Only applies to ordered lists

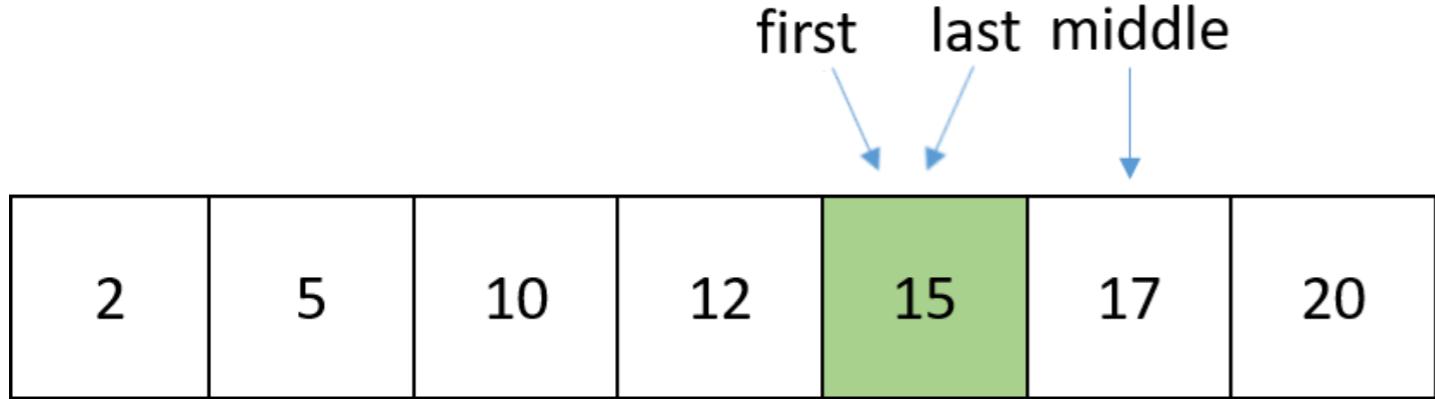


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1
```

Binary search

- Only applies to ordered lists

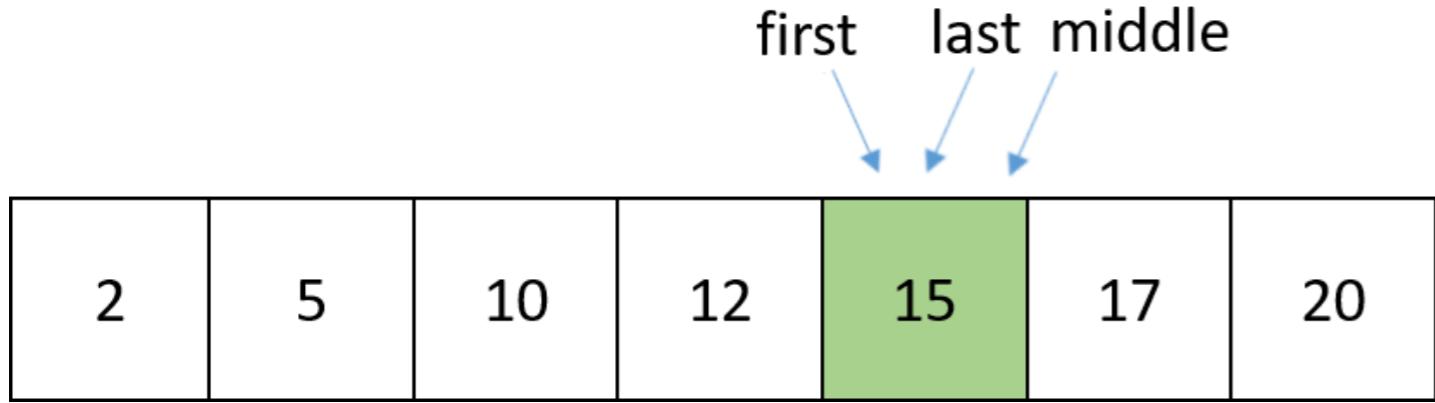


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1
```

Binary search

- Only applies to ordered lists

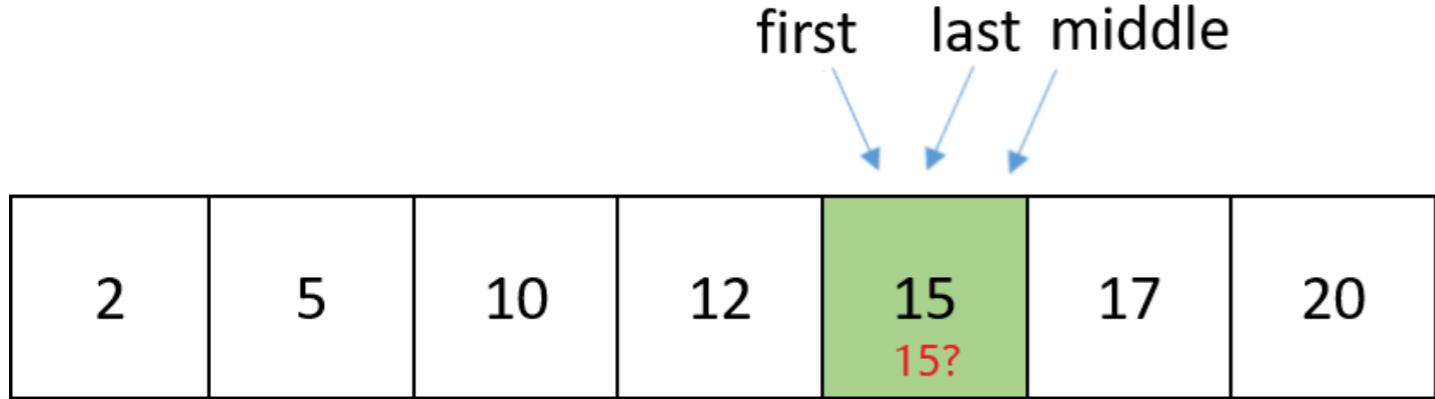


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1
```

Binary search

- Only applies to ordered lists

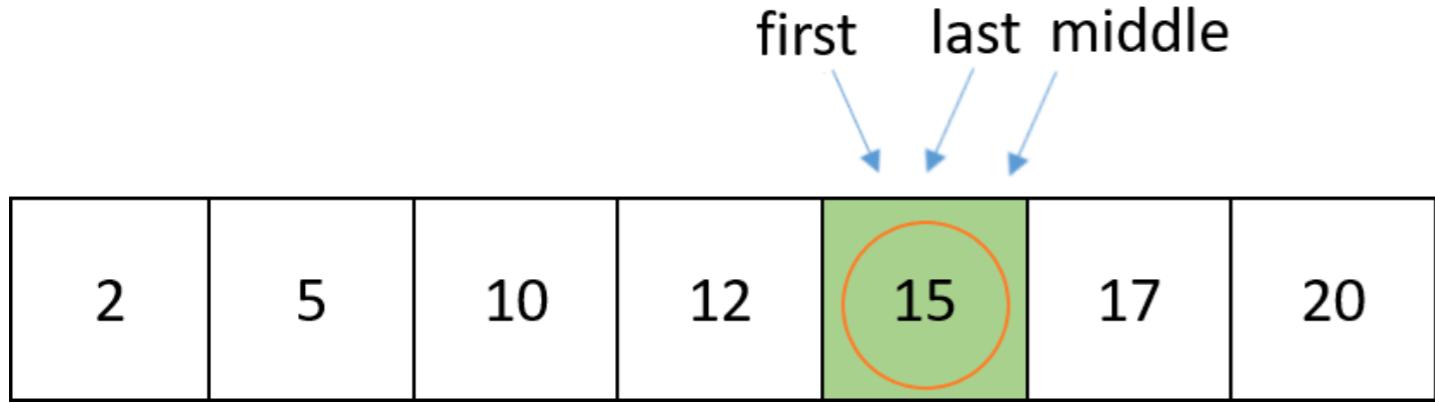


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1
```

Binary search

- Only applies to ordered lists

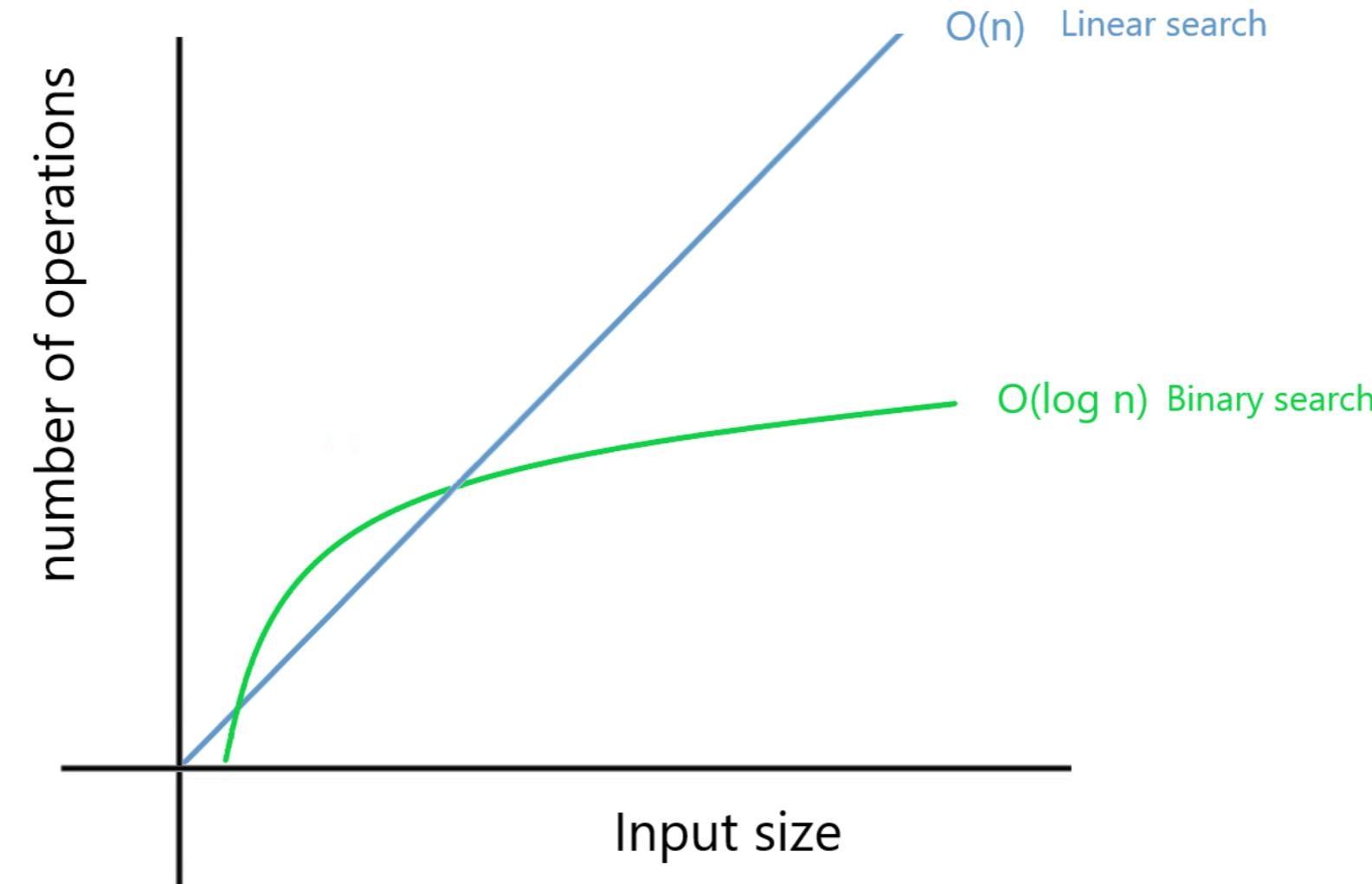


- 15 ???
- Compare `search_value` with the item in the **middle** of the list

```
def binary_search(ordered_list, search_value):  
    first = 0  
    last = len(ordered_list) - 1  
  
    while first <= last:  
        middle = (first + last)//2  
        if search_value == ordered_list[middle]:  
            return True  
        elif search_value < ordered_list[middle]:  
            last = middle - 1  
        else:  
            first = middle + 1  
    return False
```

Binary search - complexity

- Complexity: $O(\log n)$

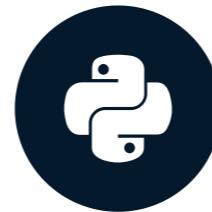


Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Binary Search Tree (BST)

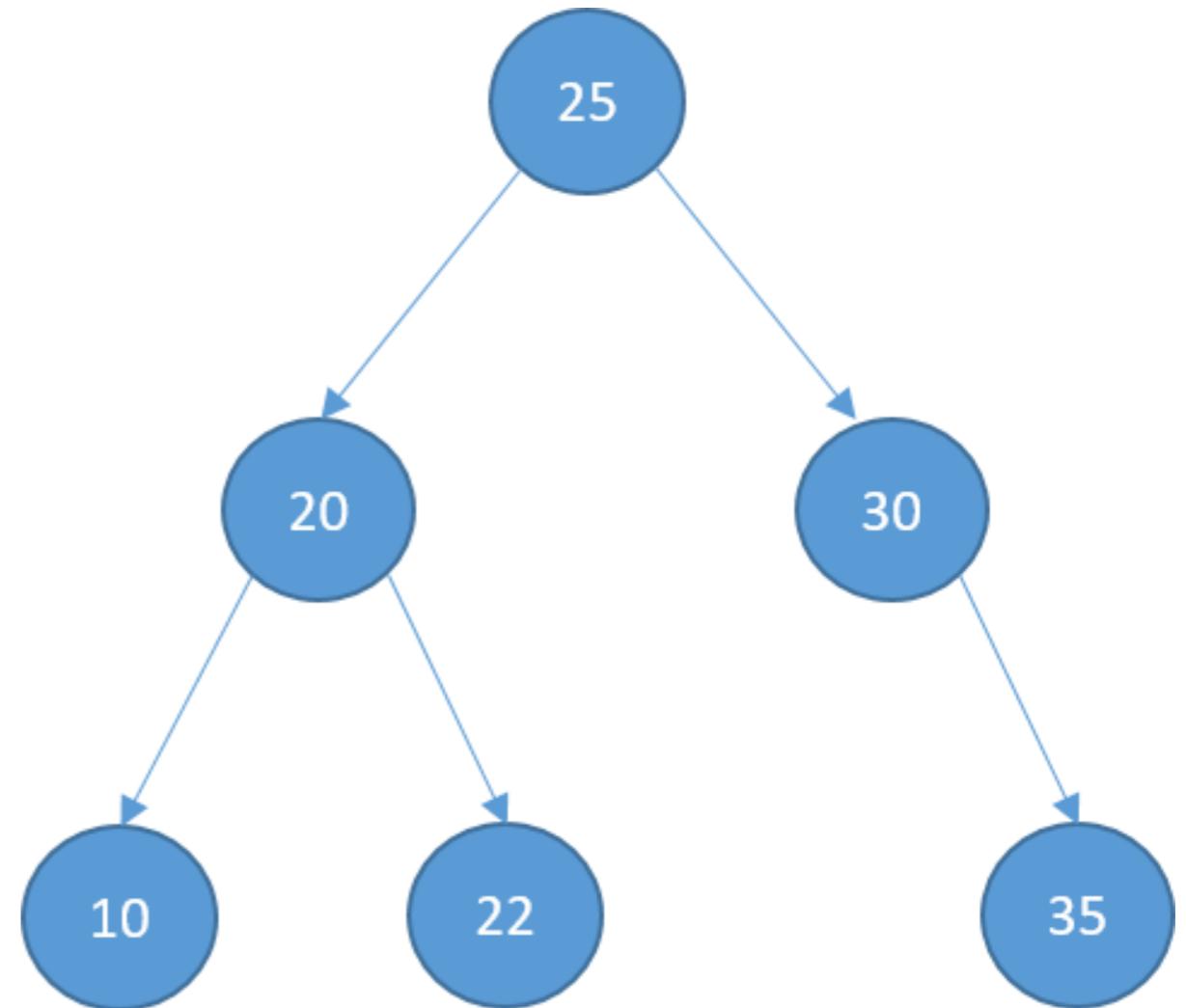
DATA STRUCTURES AND ALGORITHMS IN PYTHON



Miriam Antona
Software engineer

Definition

- **Left subtree** of a node:
 - values **less** than the node itself
- **Right subtree** of a node:
 - values **greater** than the node itself
- Left and right subtrees must be binary search trees



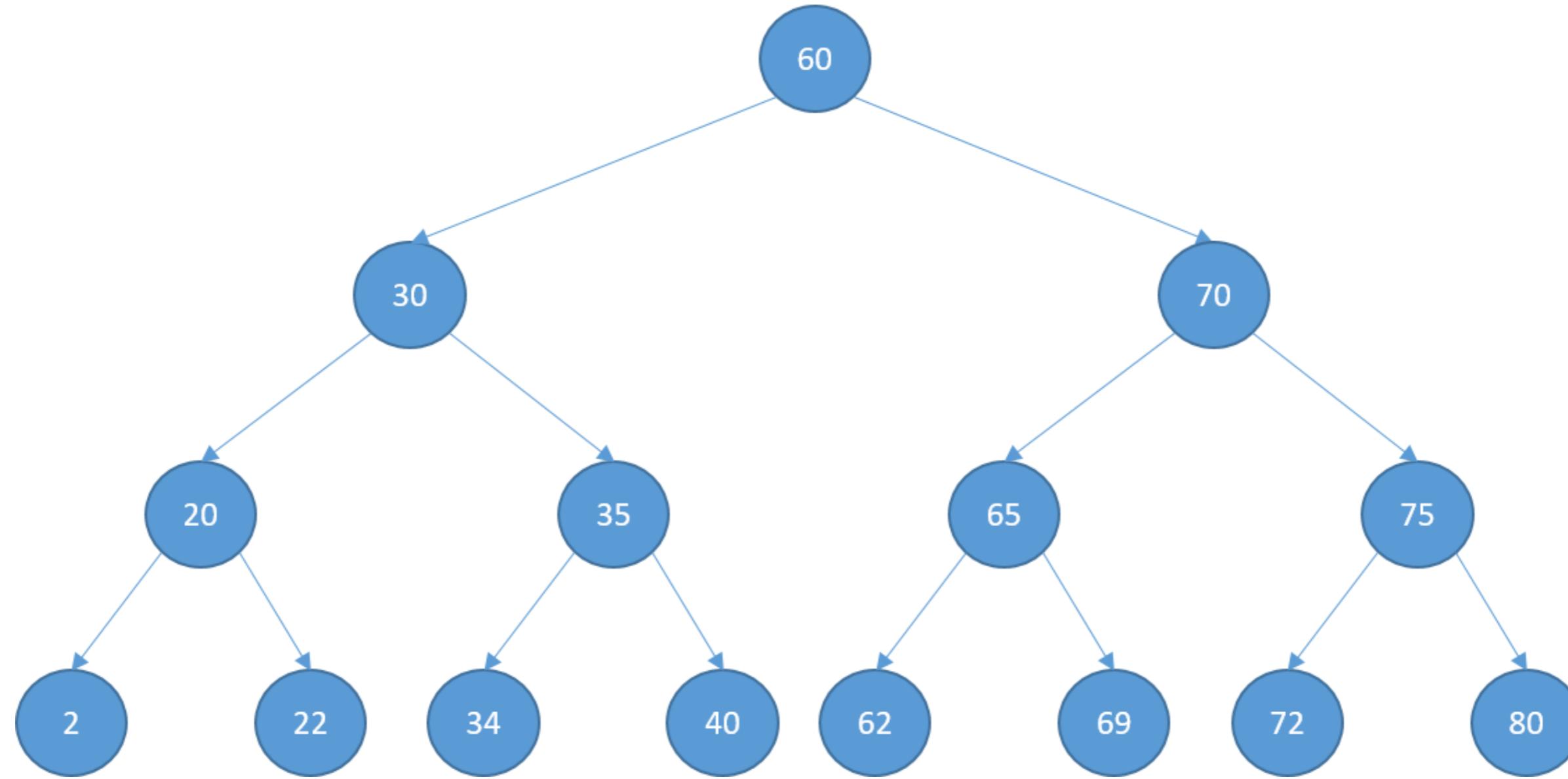
Implementation

```
class TreeNode:  
    def __init__(self, data, left=None, right=None):  
        self.data = data  
        self.left_child = left  
        self.right_child = right
```

```
class BinarySearchTree:  
    def __init__(self):  
        self.root = None
```

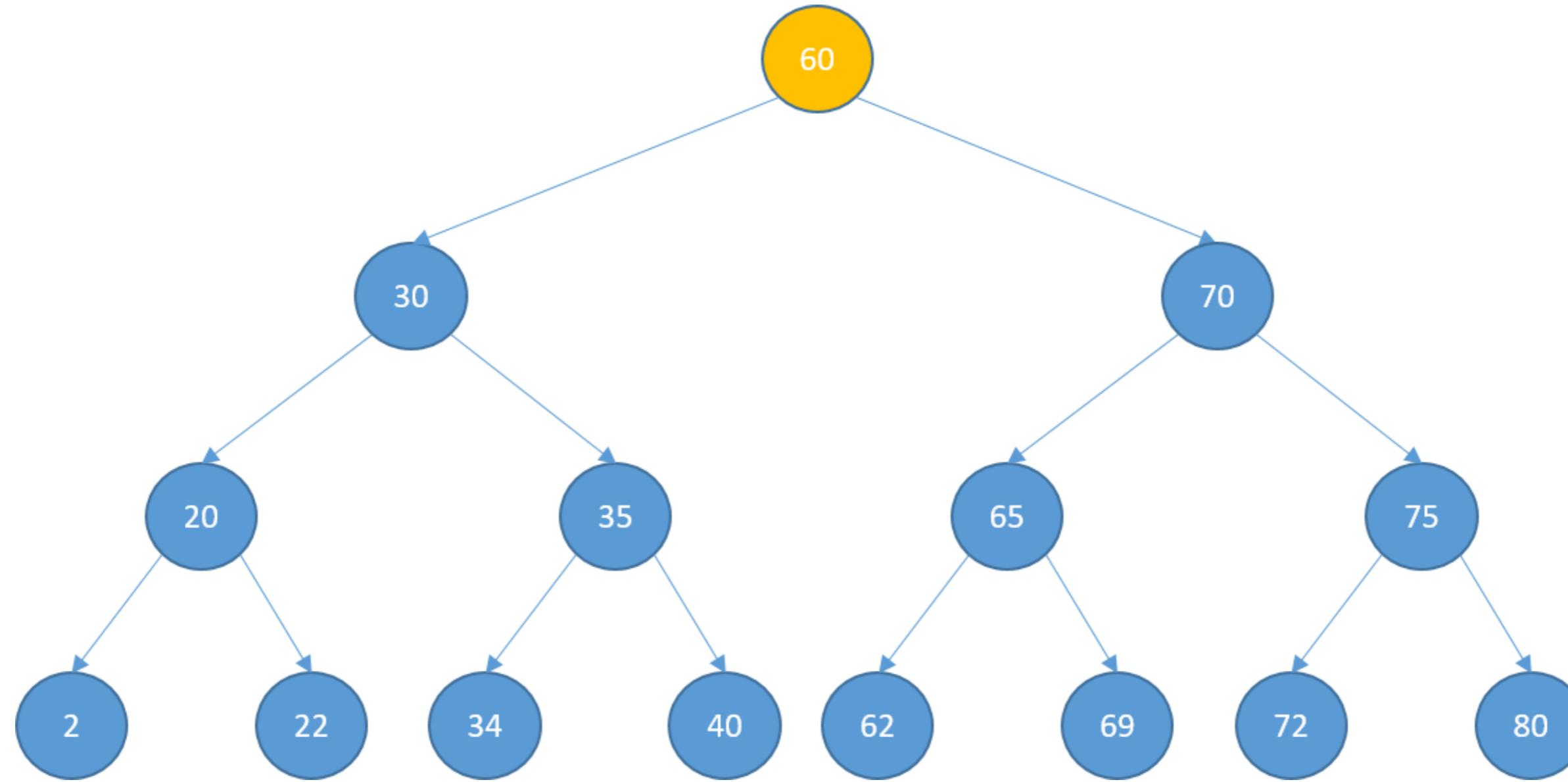
Searching

- Search for 72



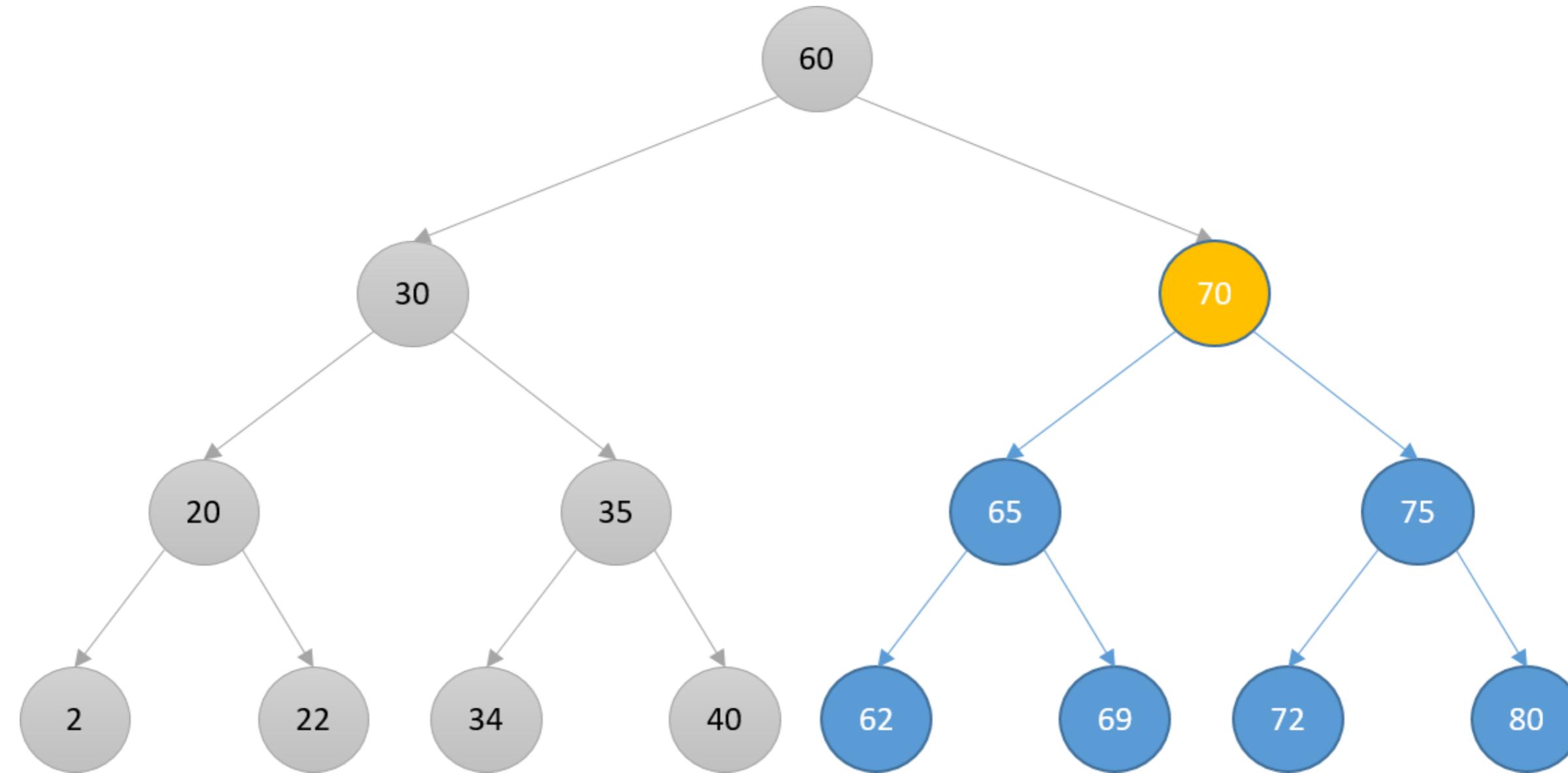
Searching

- Search for 72



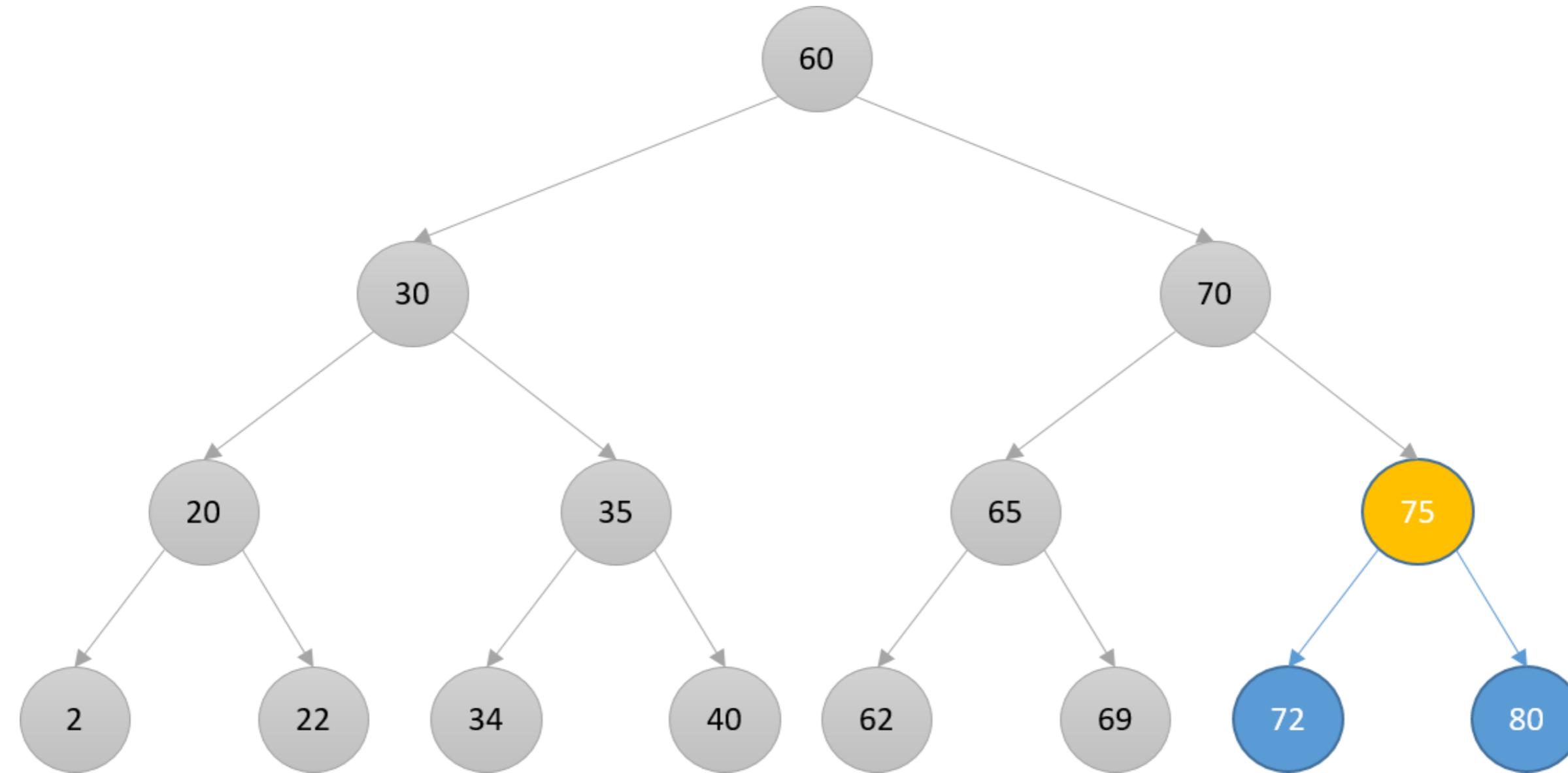
Searching

- Search for 72



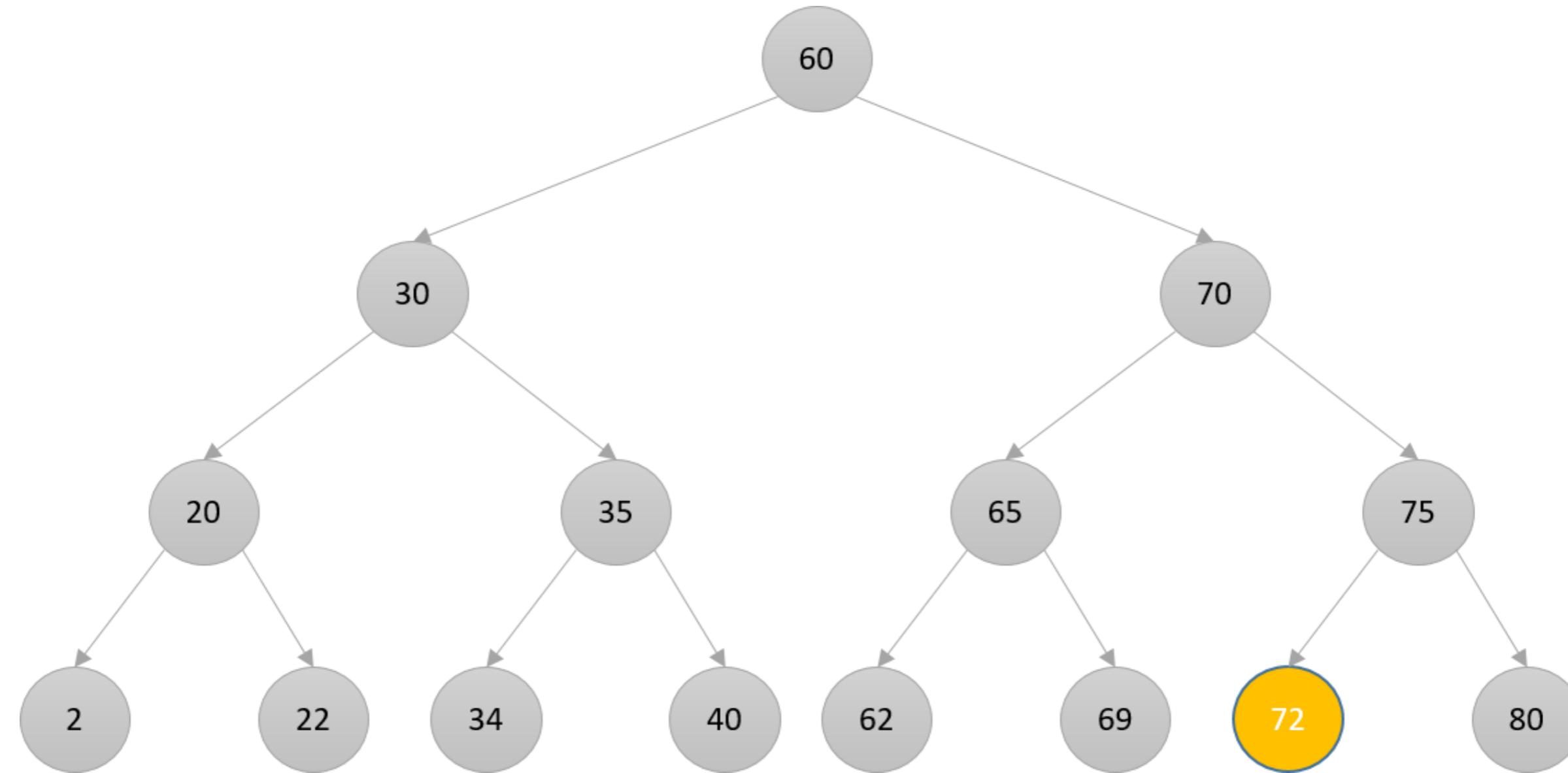
Searching

- Search for 72



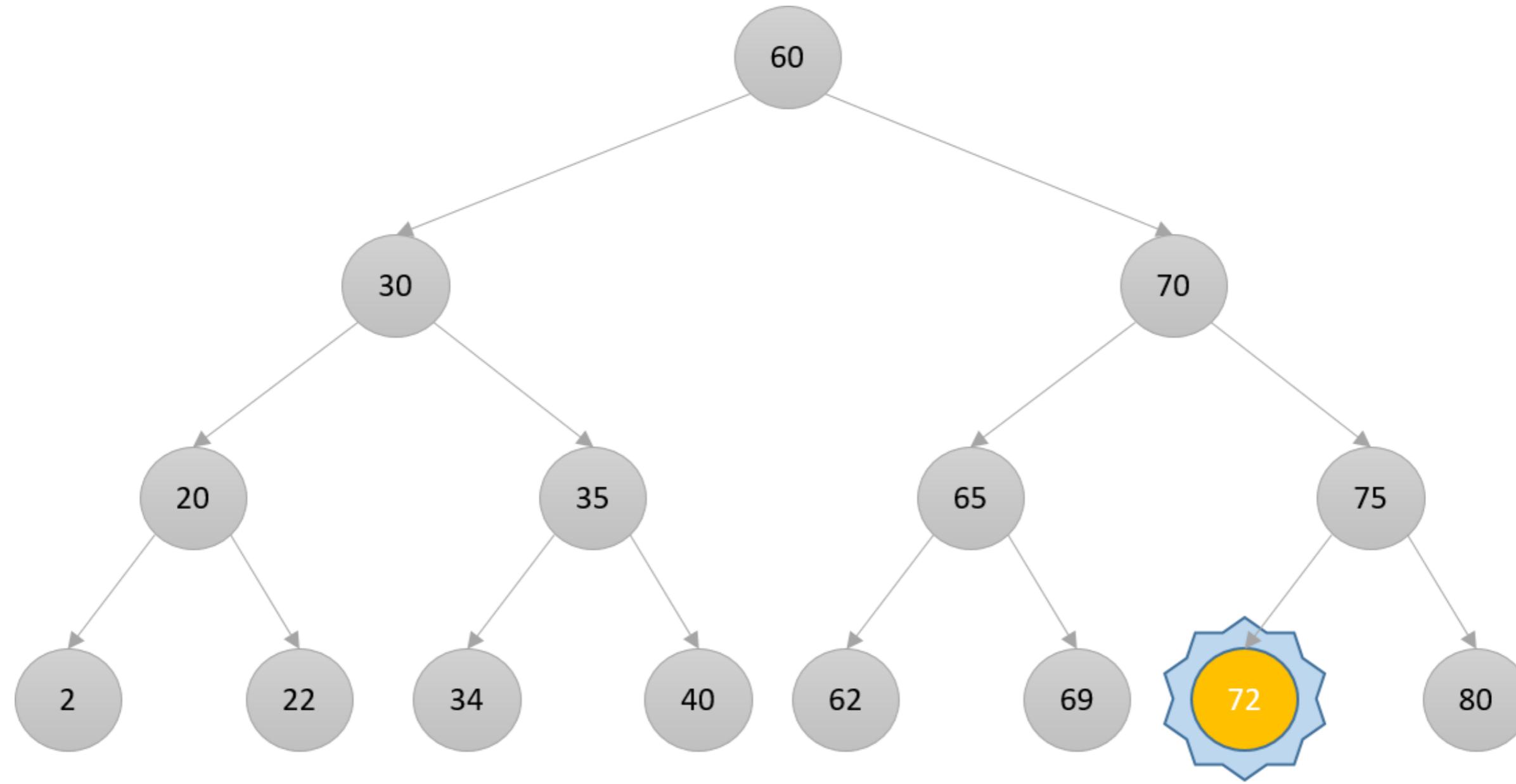
Searching

- Search for 72



Searching

- Search for 72



Searching

```
def search(self, search_value):  
    current_node = self.root  
    while current_node:  
        if search_value == current_node.data:  
            return True  
        elif search_value < current_node.data:  
            current_node = current_node.left_child  
        else:  
            current_node = current_node.right_child  
    return False
```

Inserting

```
def insert(self, data):  
    new_node = TreeNode(data)  
    if self.root == None:
```

new_node



Inserting

```
def insert(self, data):  
    new_node = TreeNode(data)  
    if self.root == None:  
        self.root = new_node  
    return
```

root



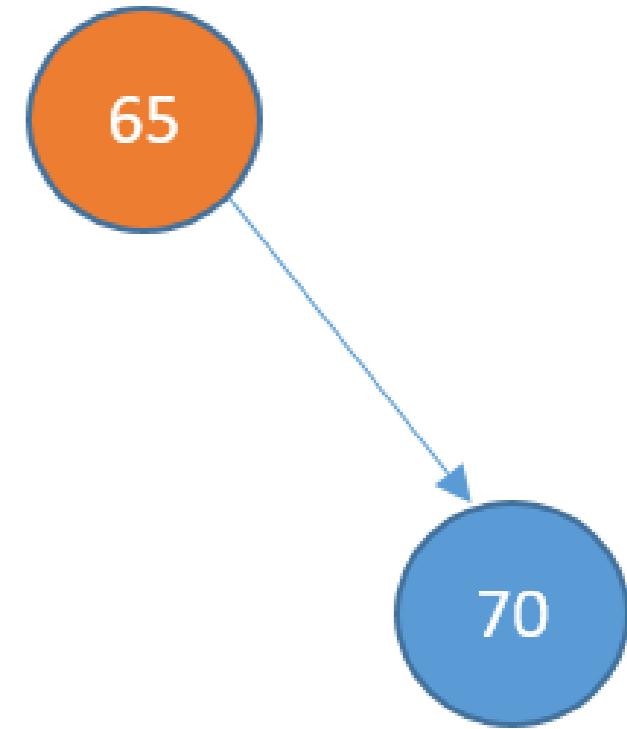
Inserting

```
def insert(self, data):  
    new_node = TreeNode(data)  
    if self.root == None:  
        self.root = new_node  
    return  
else:
```

new_node



root



Inserting

```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
    return
else:
    current_node = self.root
    while True:
        if data < current_node.data:
            if current_node.left_child == None:
```

new_node



root



current_node

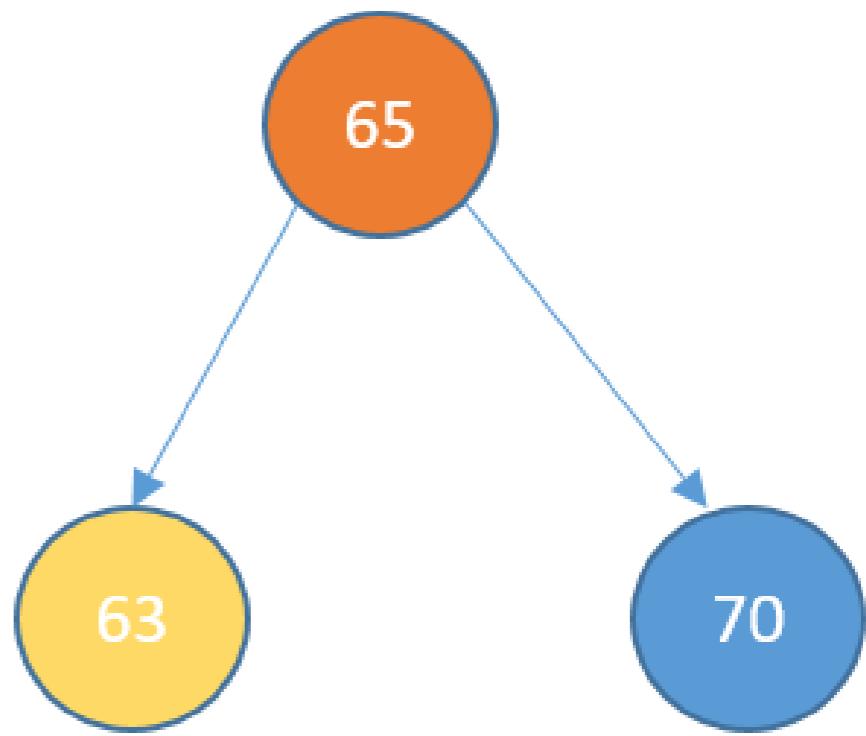


70

Inserting

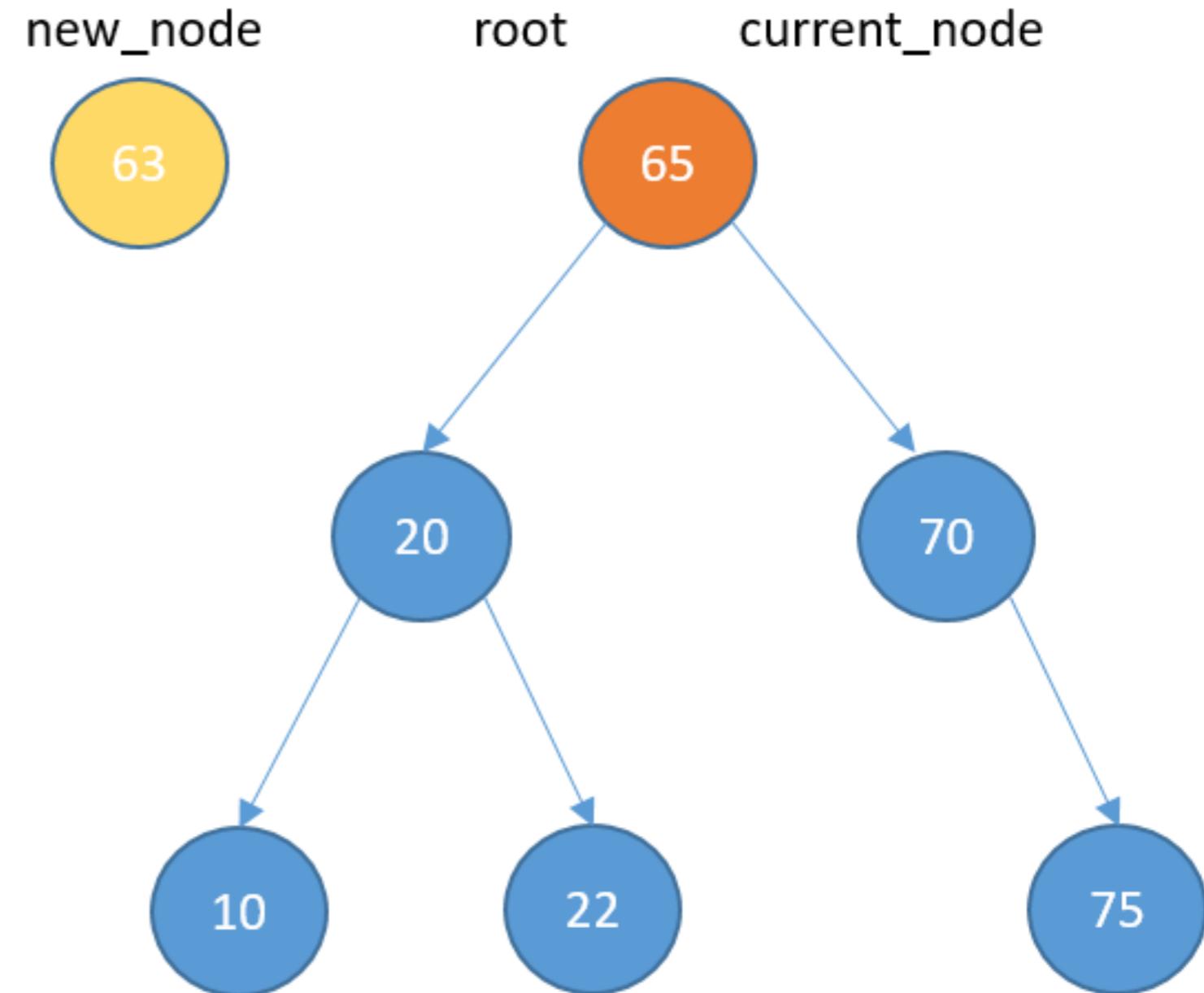
```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
```

root current_node



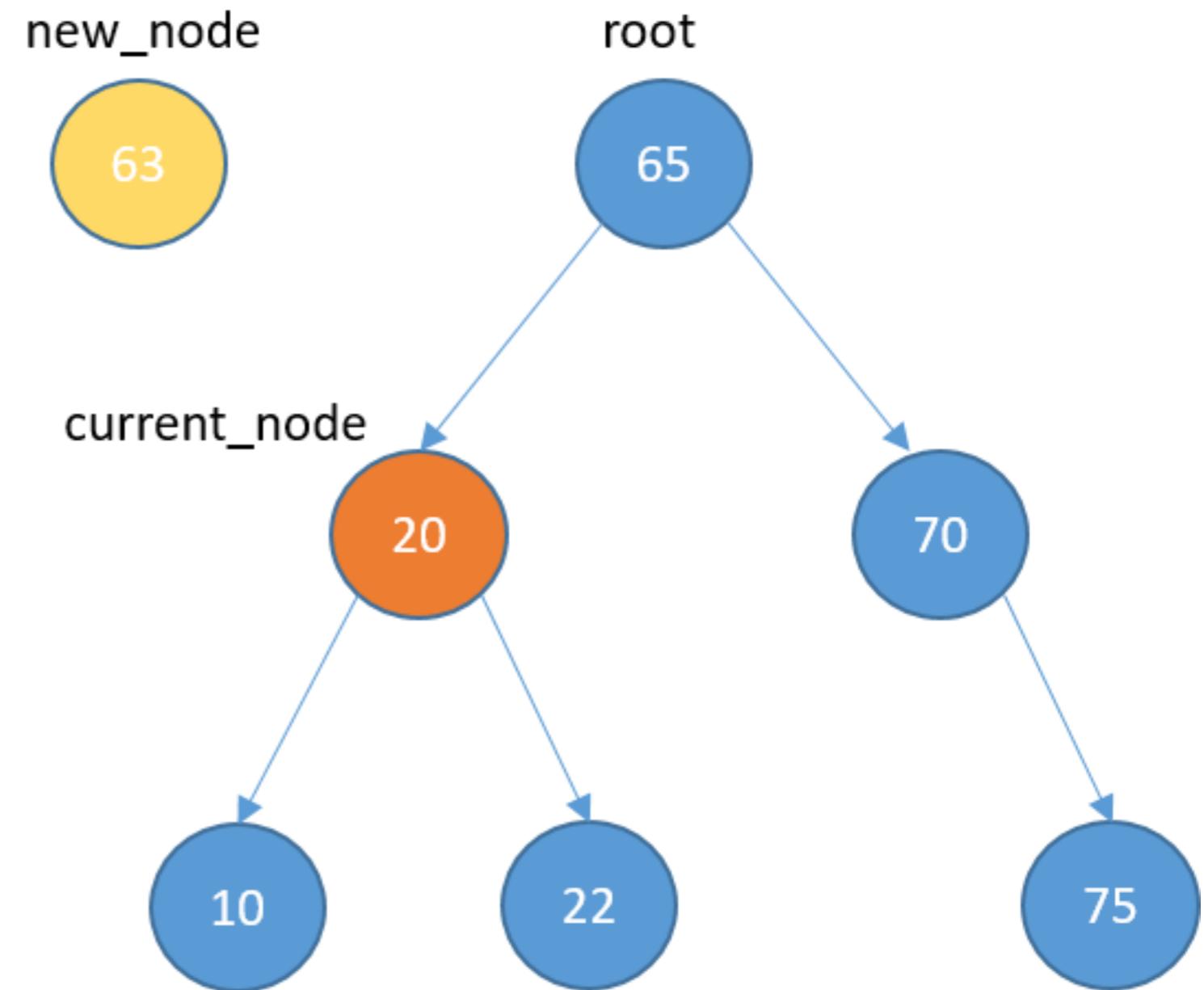
Inserting

```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
            else:
```



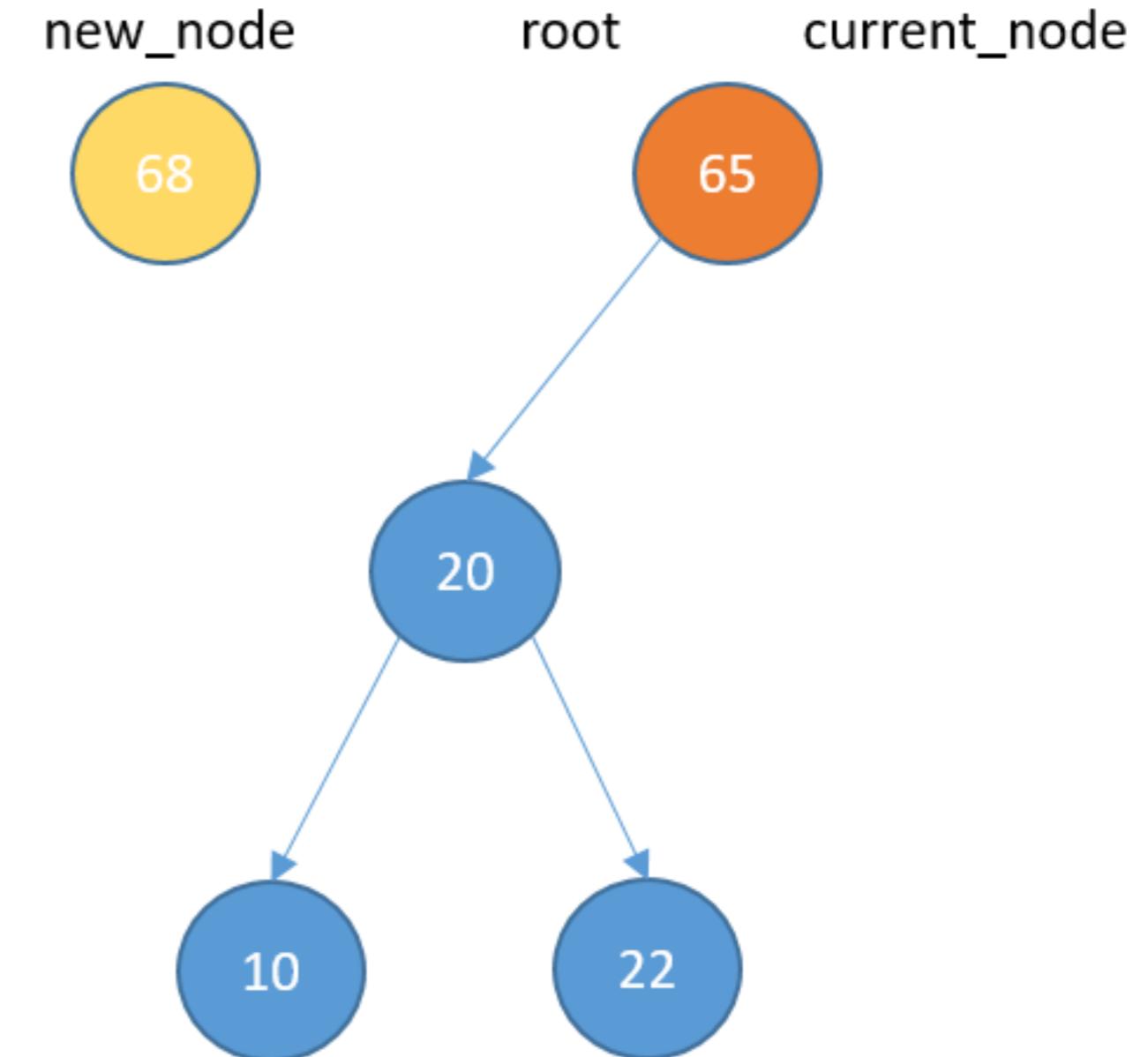
Inserting

```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
                else:
                    current_node = current_node.left_child
```



Inserting

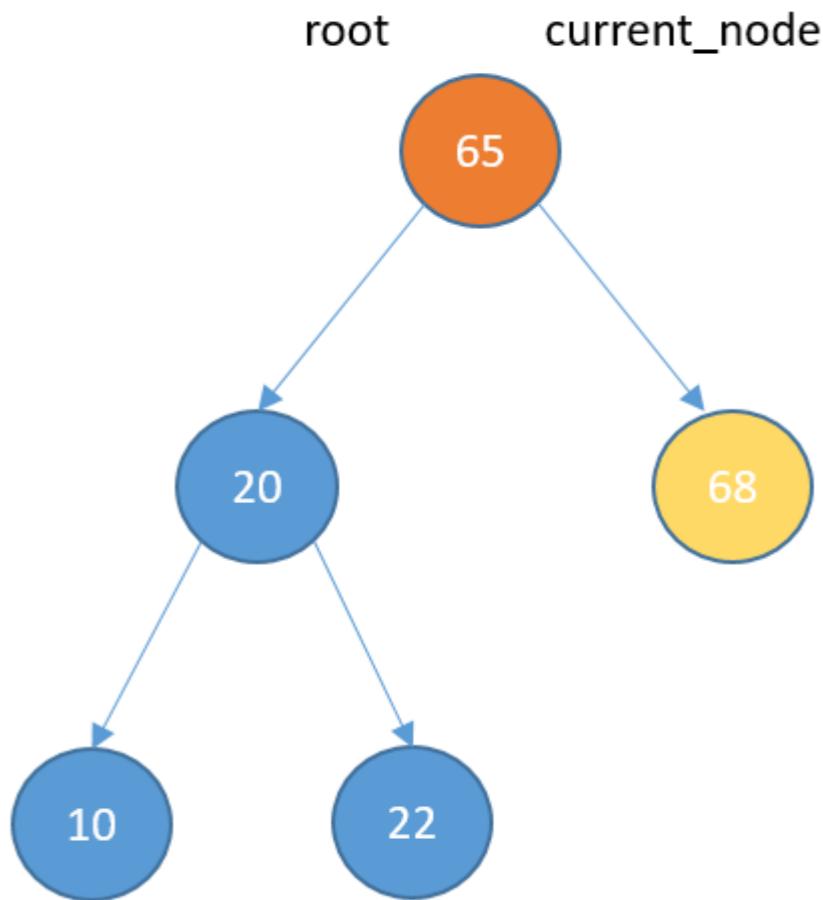
```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
                else:
                    current_node = current_node.left_child
            elif data > current_node.data:
                if current_node.right_child == None:
```



Inserting

```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
                else:
                    current_node = current_node.left_child
            elif data > current_node.data:
                if current_node.right_child == None:
```

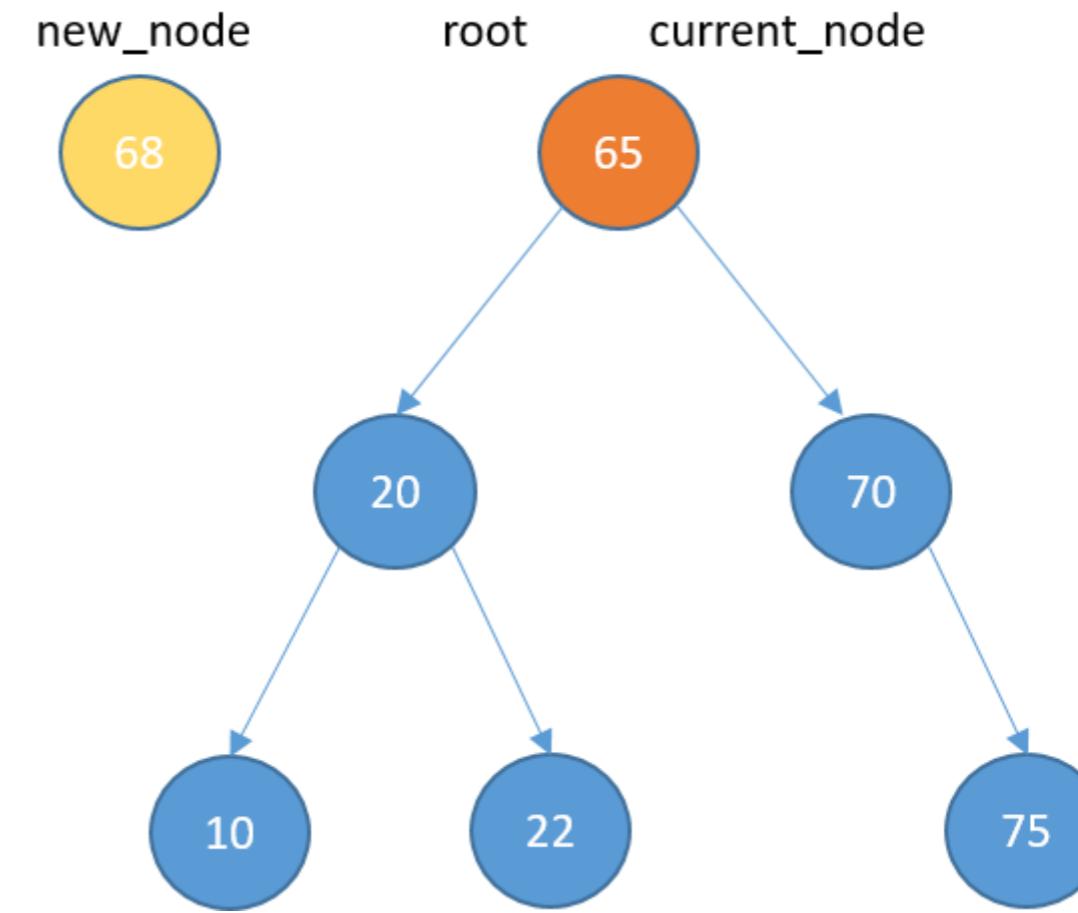
```
        current_node.right_child = new_node
        return
```



Inserting

```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
                else:
                    current_node = current_node.left_child
            elif data > current_node.data:
                if current_node.right_child == None:
```

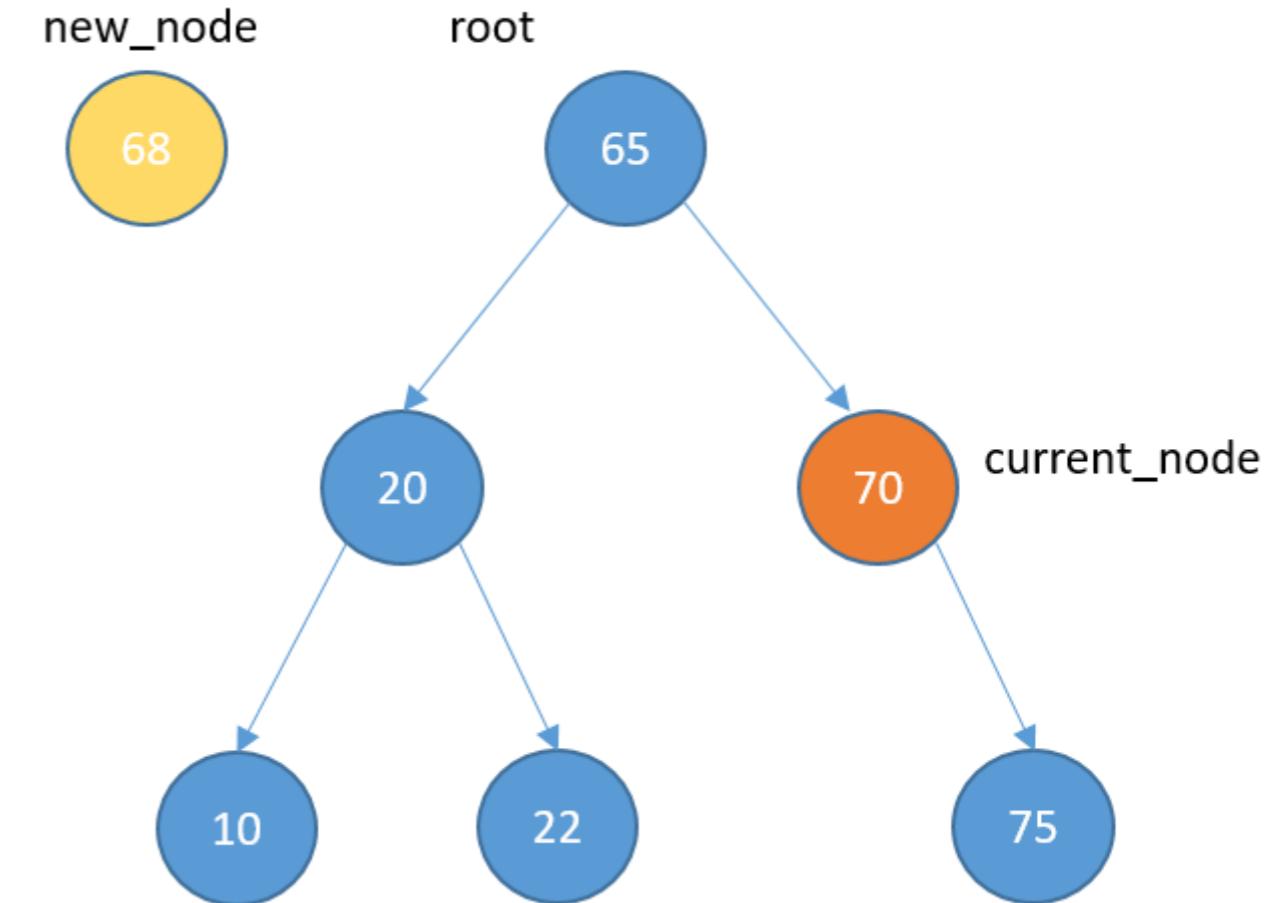
```
        current_node.right_child = new_node
        return
    else:
```



Inserting

```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
                else:
                    current_node = current_node.left_child
            elif data > current_node.data:
                if current_node.right_child == None:
```

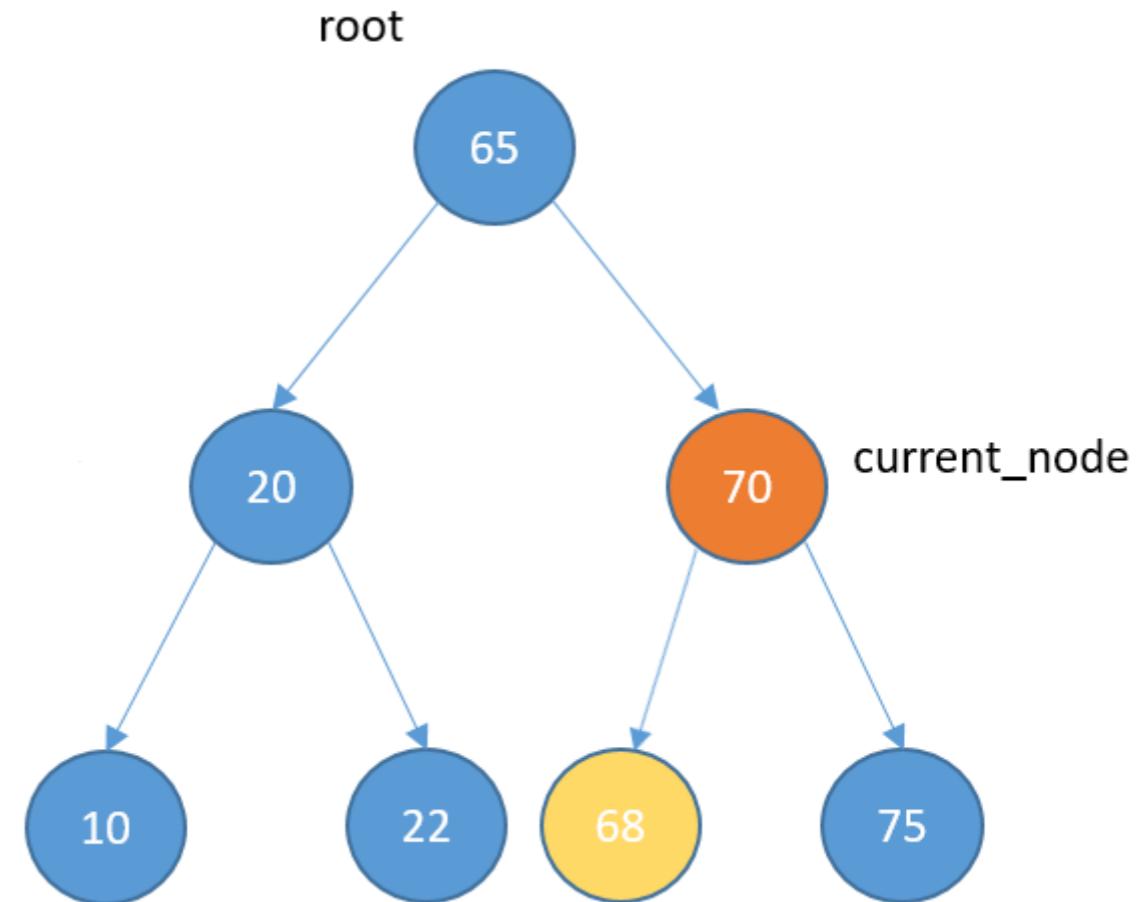
```
        current_node.right_child = new_node
        return
    else:
        current_node = current_node.right_child
```



Inserting

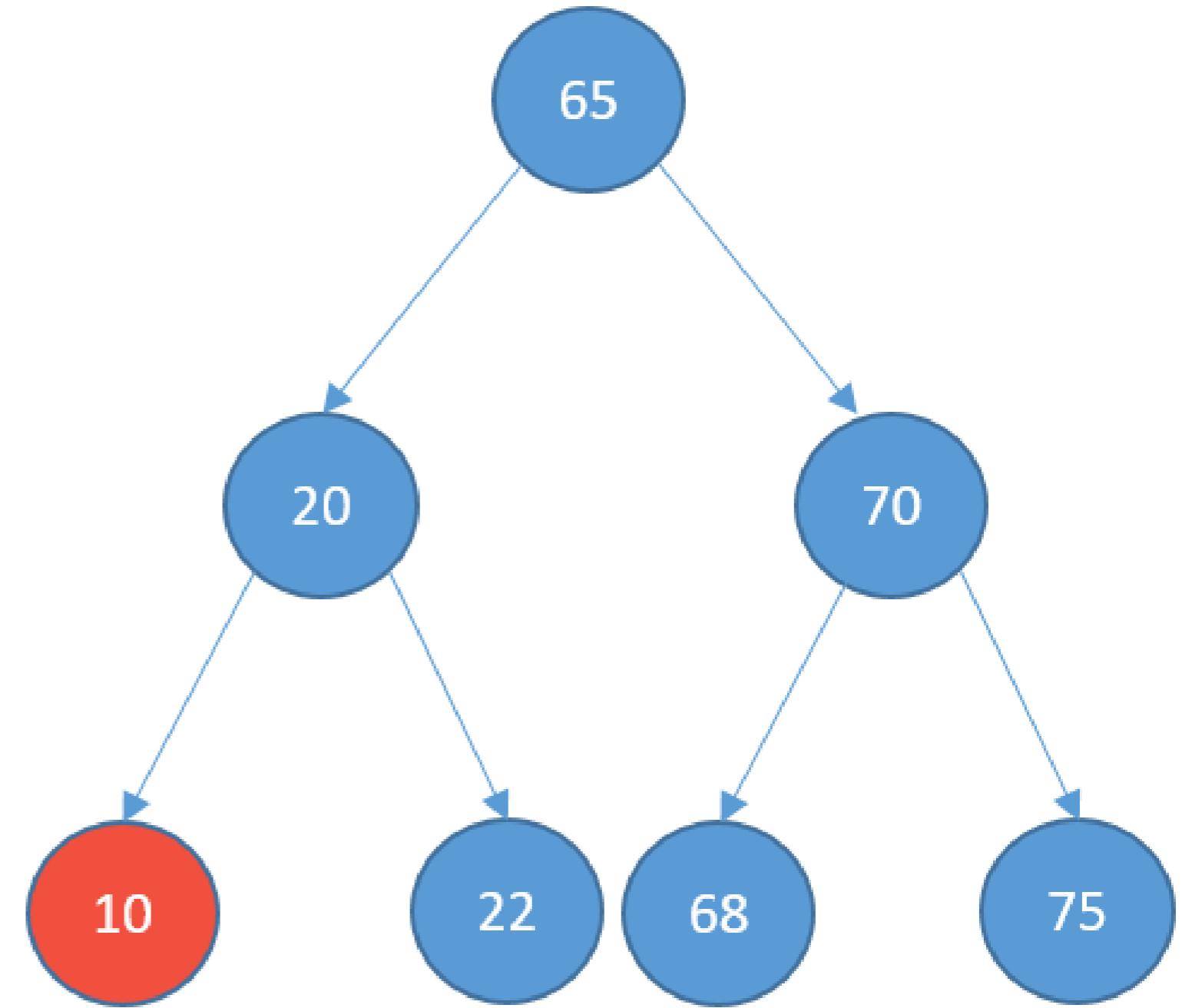
```
def insert(self, data):
    new_node = TreeNode(data)
    if self.root == None:
        self.root = new_node
        return
    else:
        current_node = self.root
        while True:
            if data < current_node.data:
                if current_node.left_child == None:
                    current_node.left_child = new_node
                    return
                else:
                    current_node = current_node.left_child
            elif data > current_node.data:
                if current_node.right_child == None:
```

```
        current_node.right_child = new_node
        return
    else:
        current_node = current_node.right_child
```



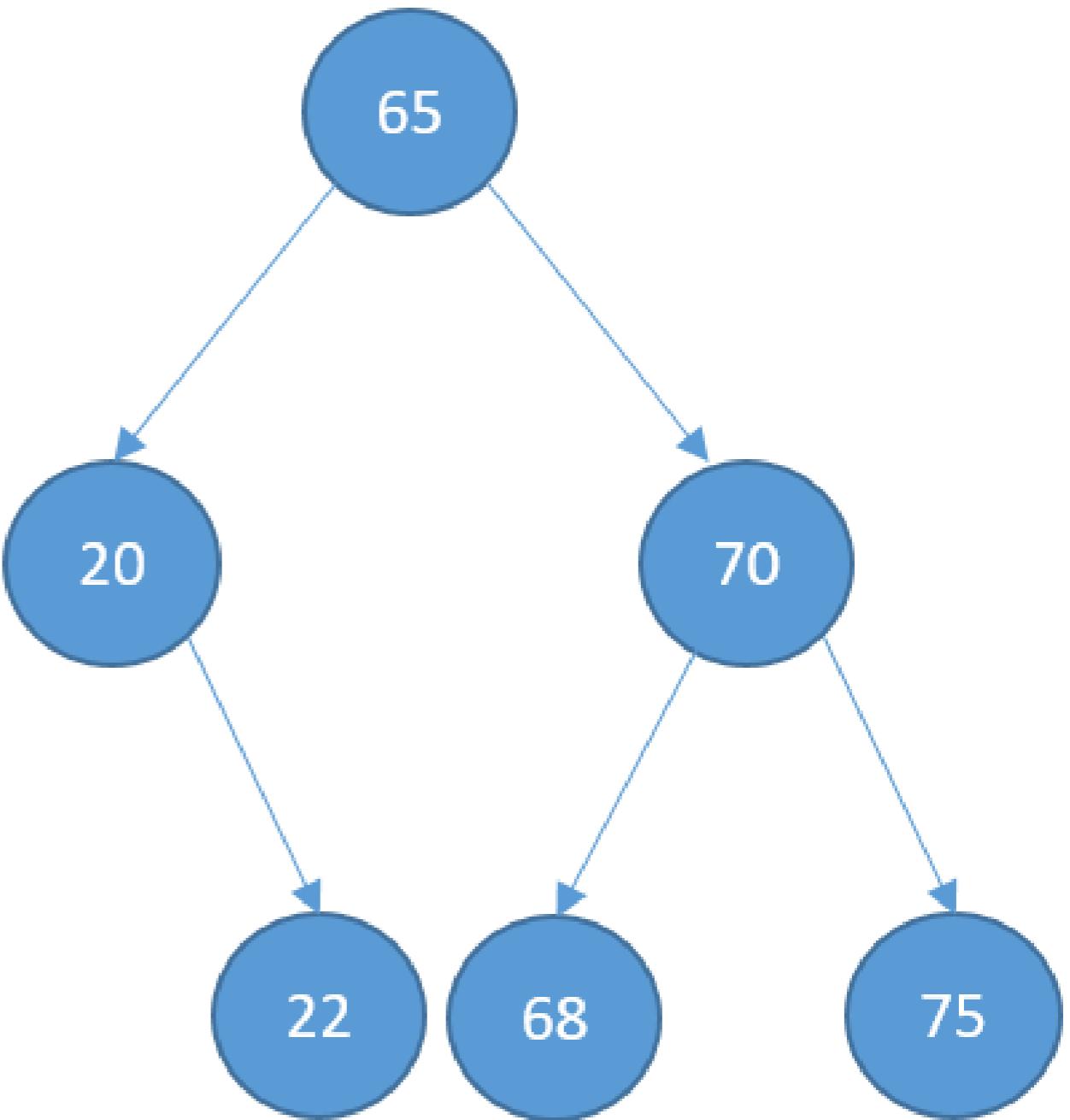
Deleting

- No children



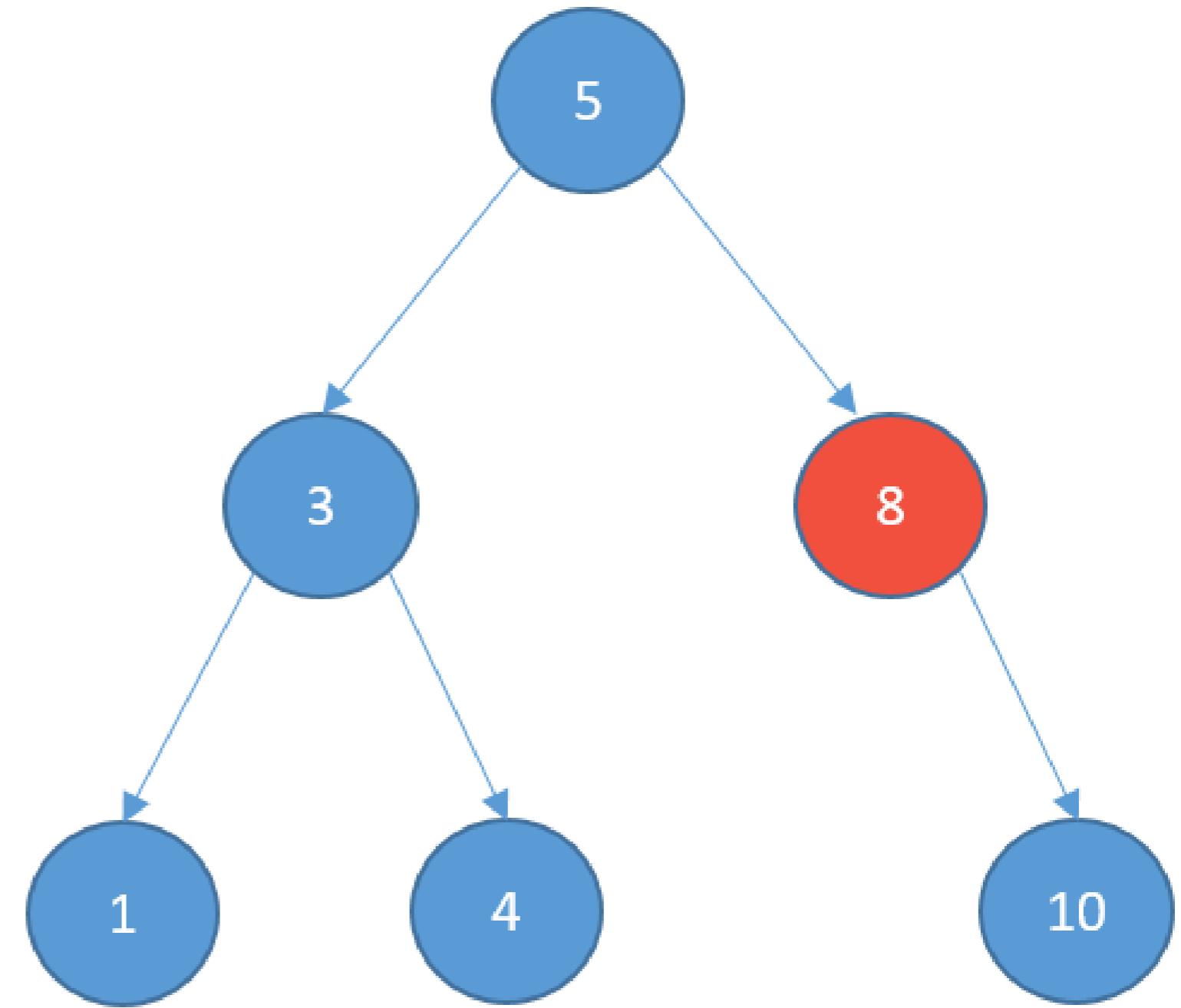
Deleting

- **No children**
 - delete it



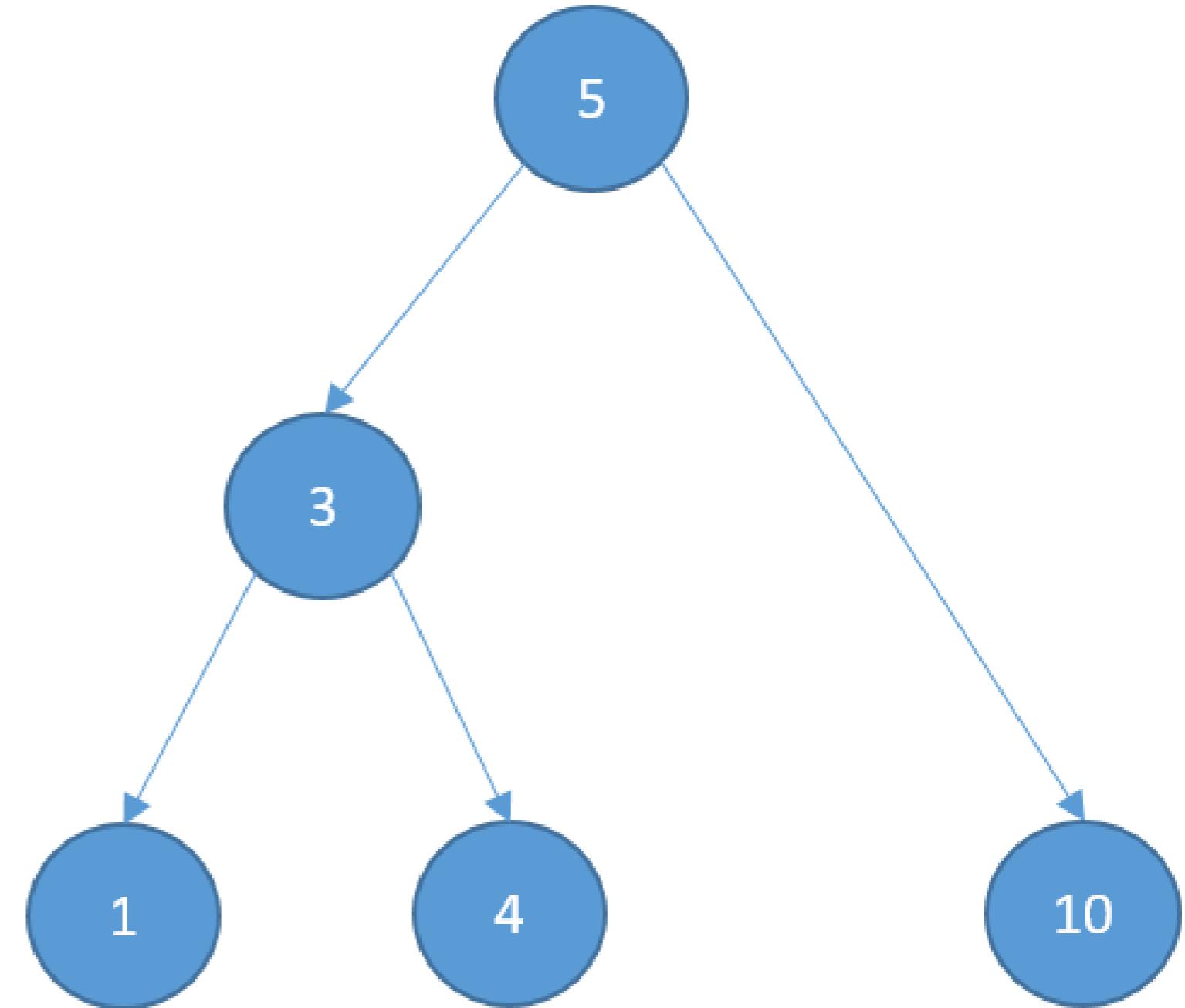
Deleting

- One child



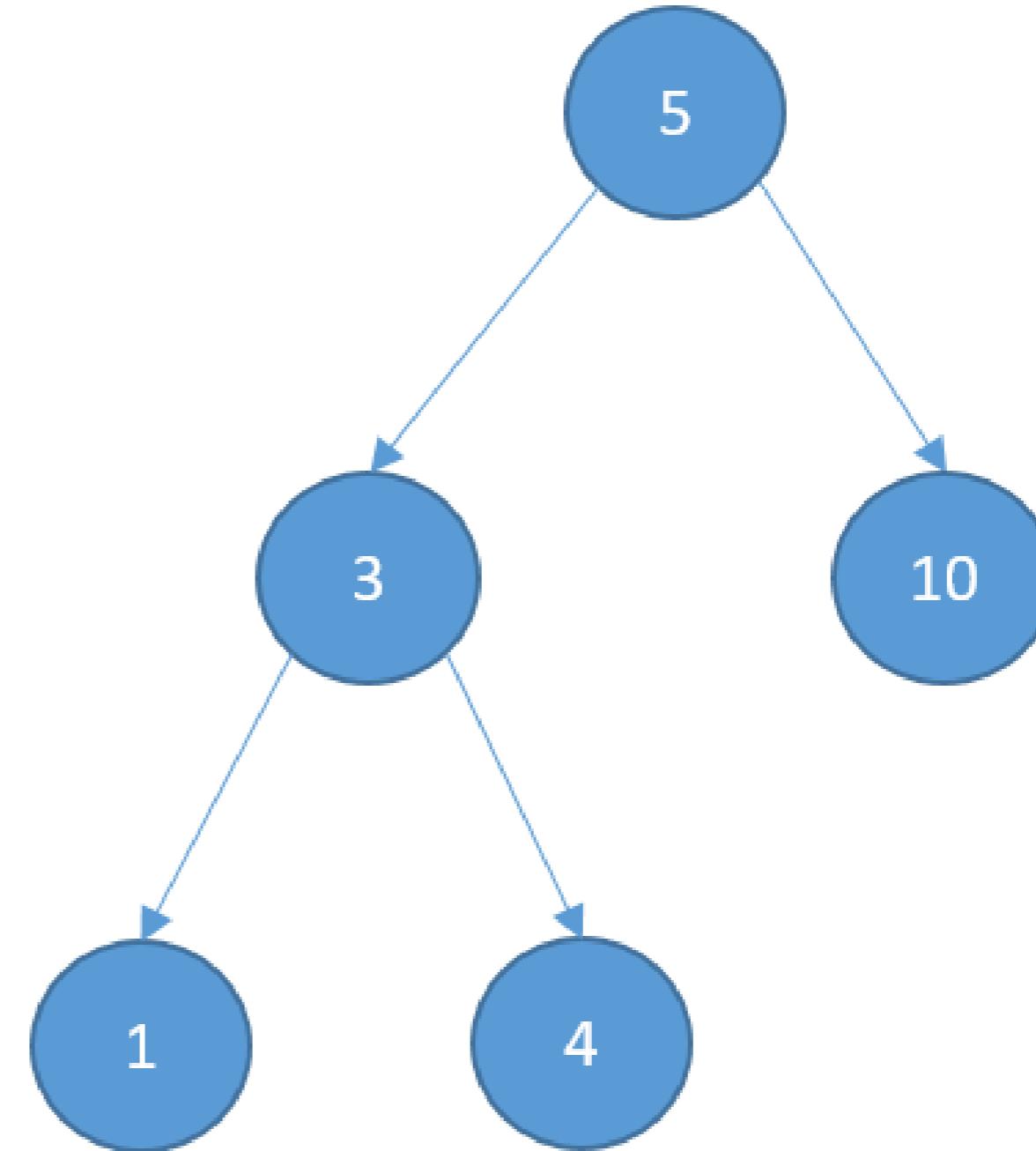
Deleting

- One child
 - delete it
 - connect the child with node's parent



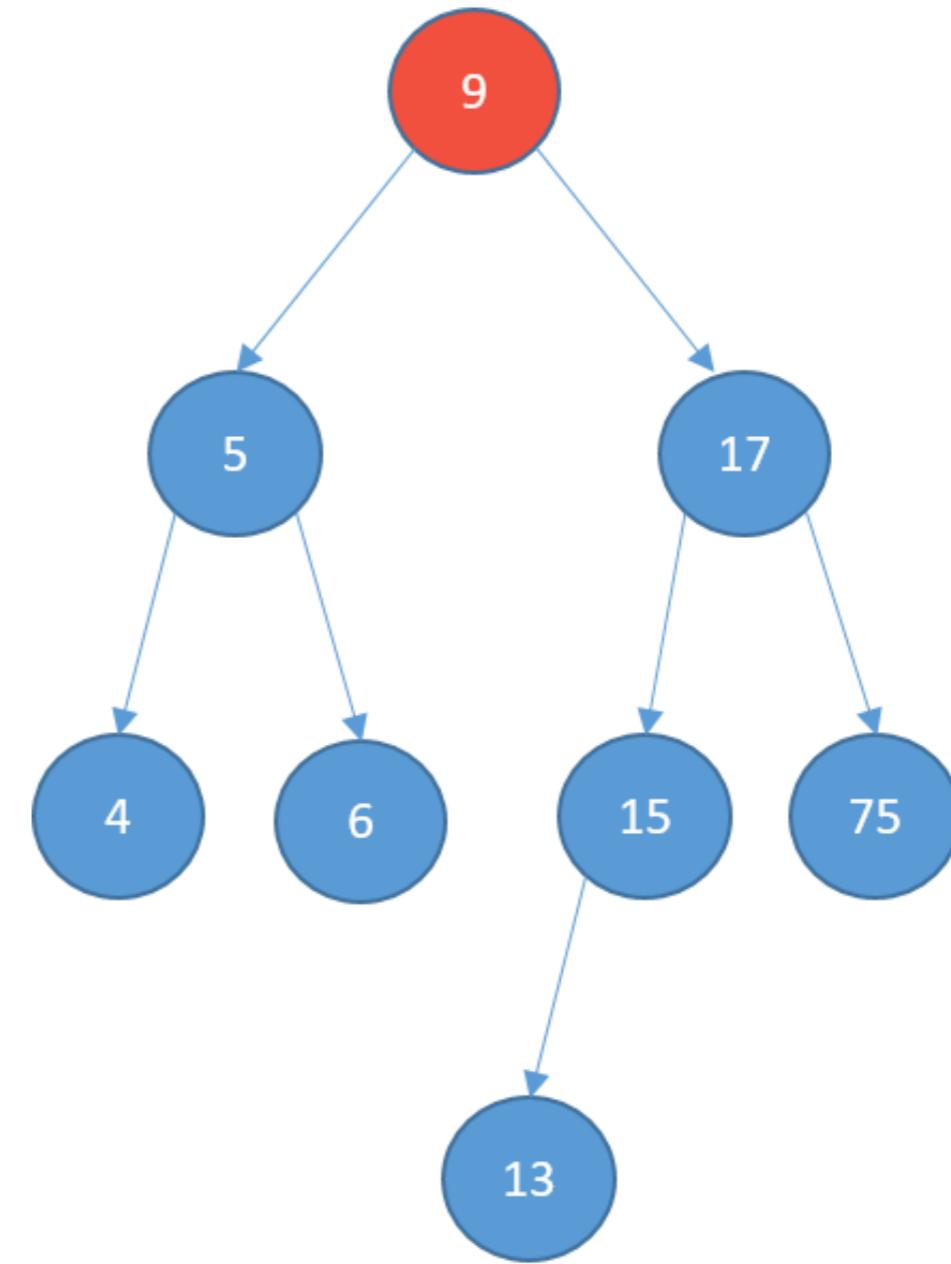
Deleting

- One child
 - delete it
 - connect the child with node's parent



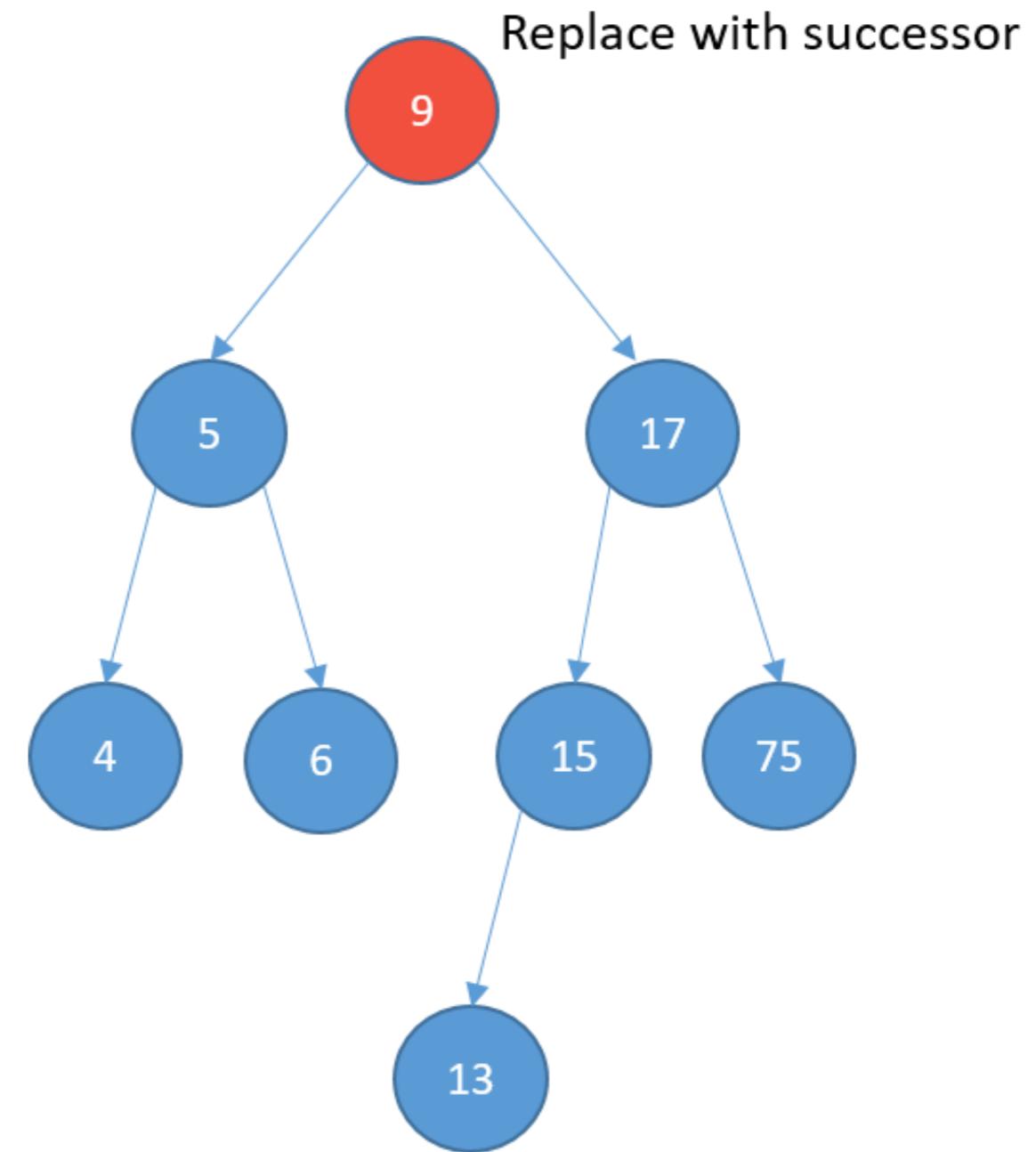
Deleting

- Two children



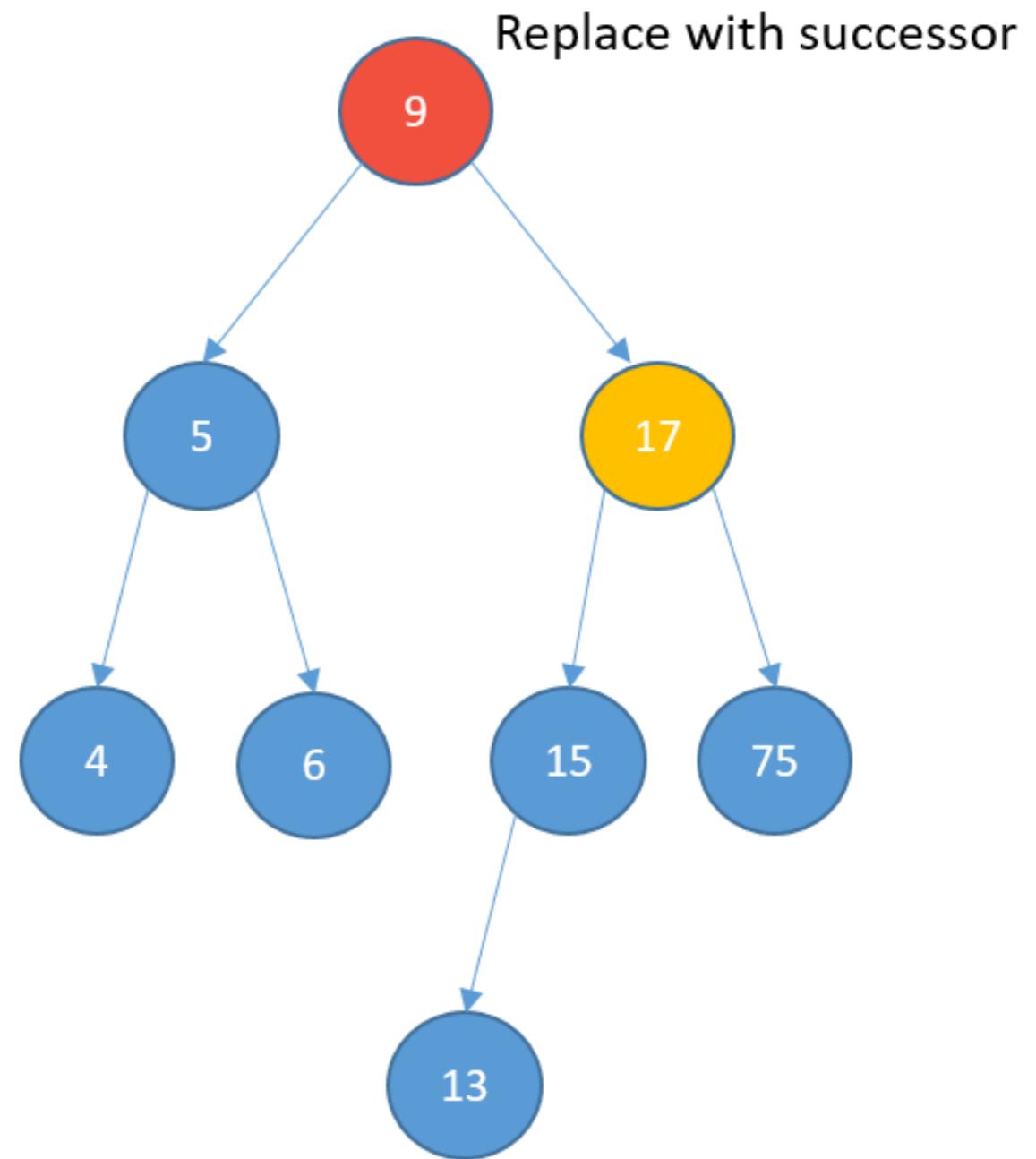
Deleting

- **Two children**
 - replace it with its successor
 - the node with the smallest value greater than the value of the node
 - find the successor:



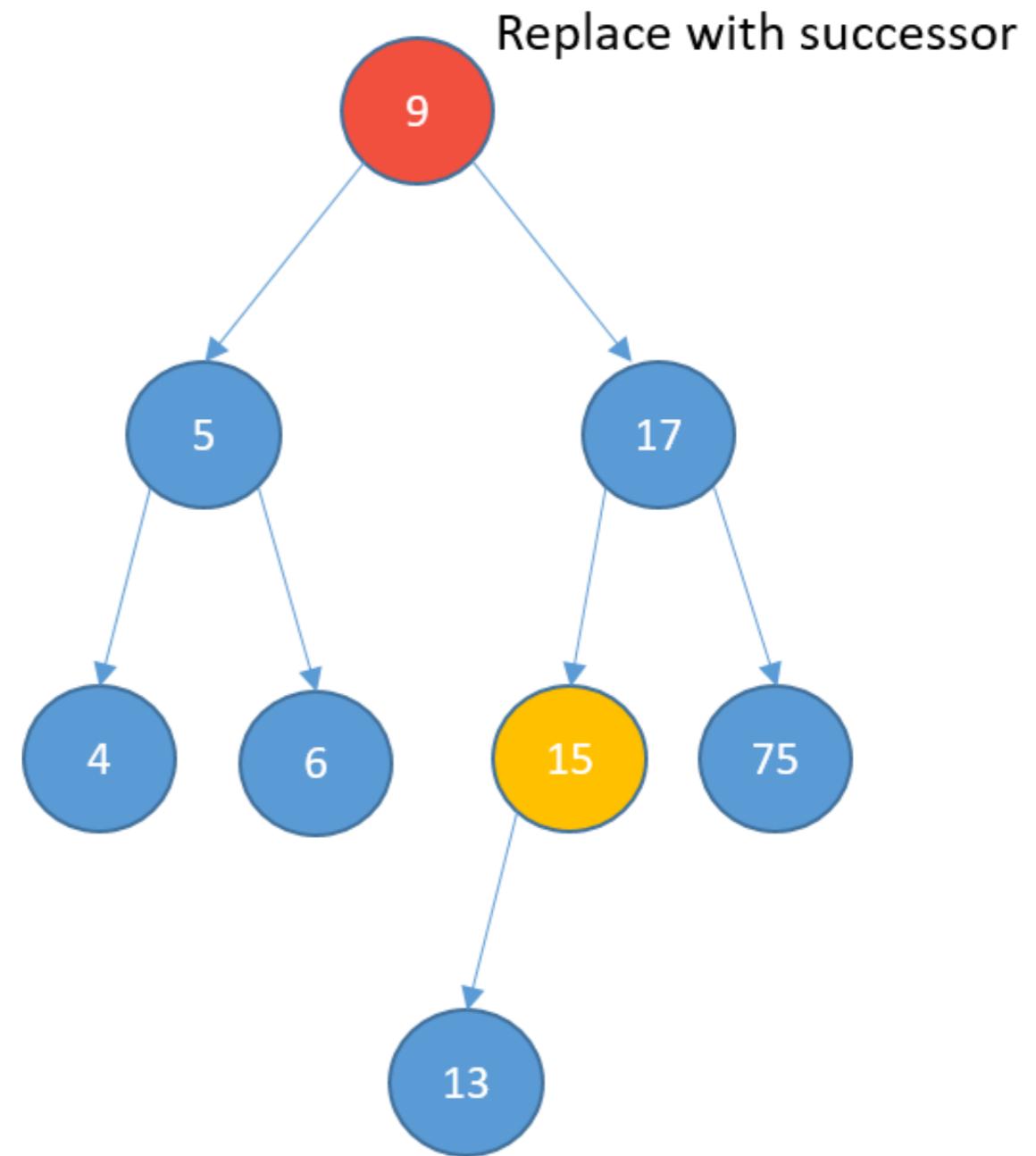
Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child



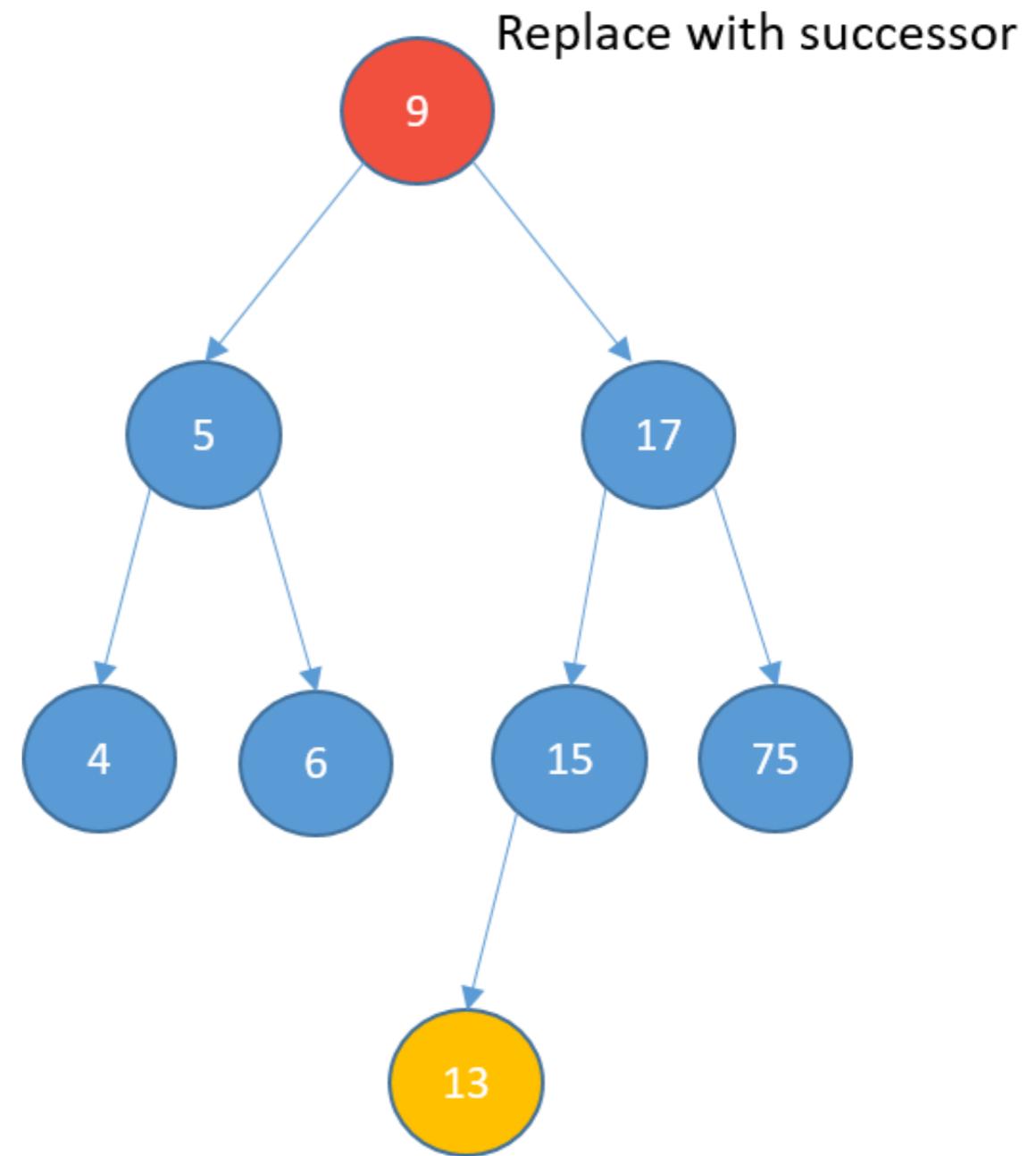
Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child
 - keep visiting the left nodes until the end



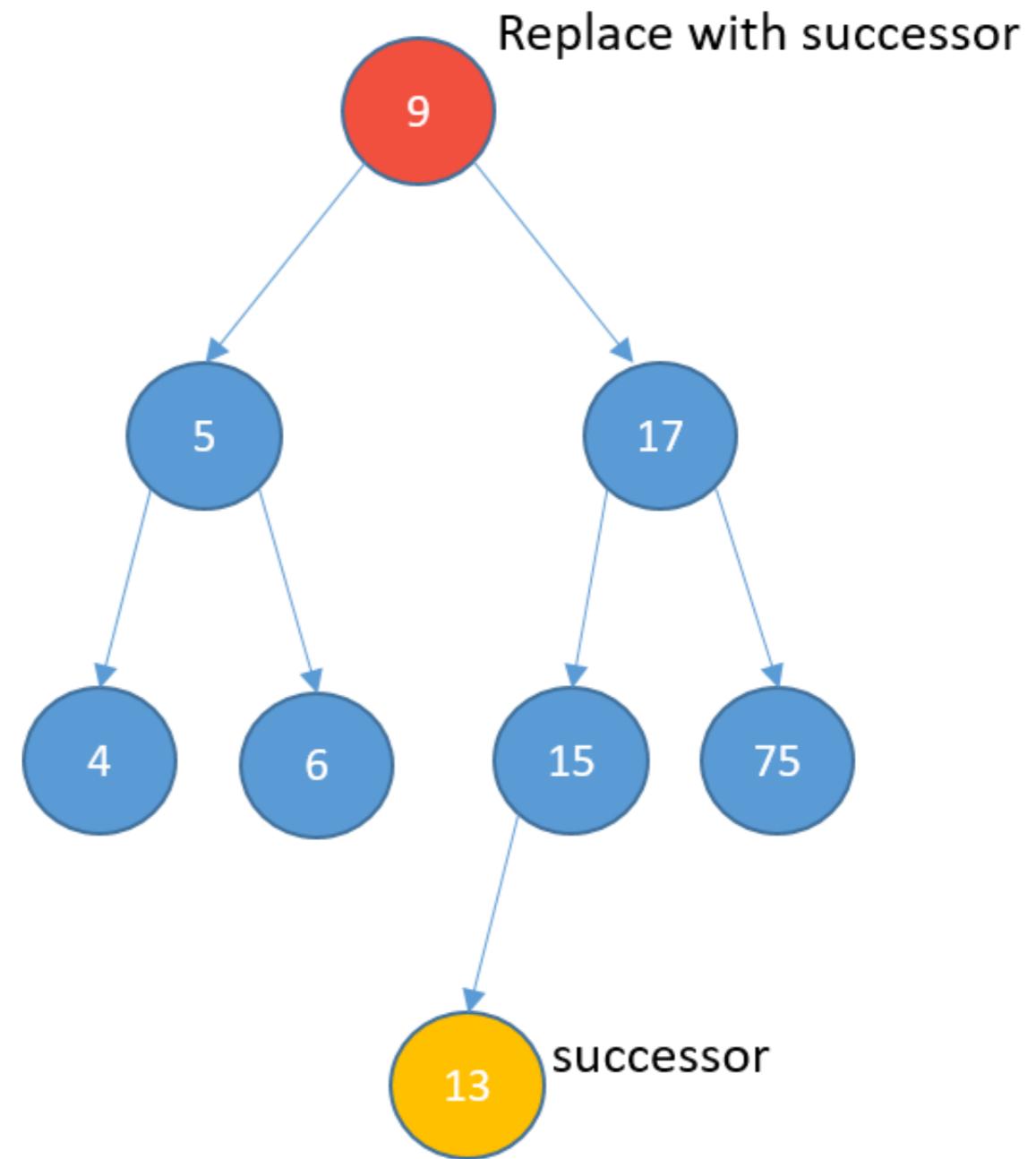
Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child
 - keep visiting the left nodes until the end



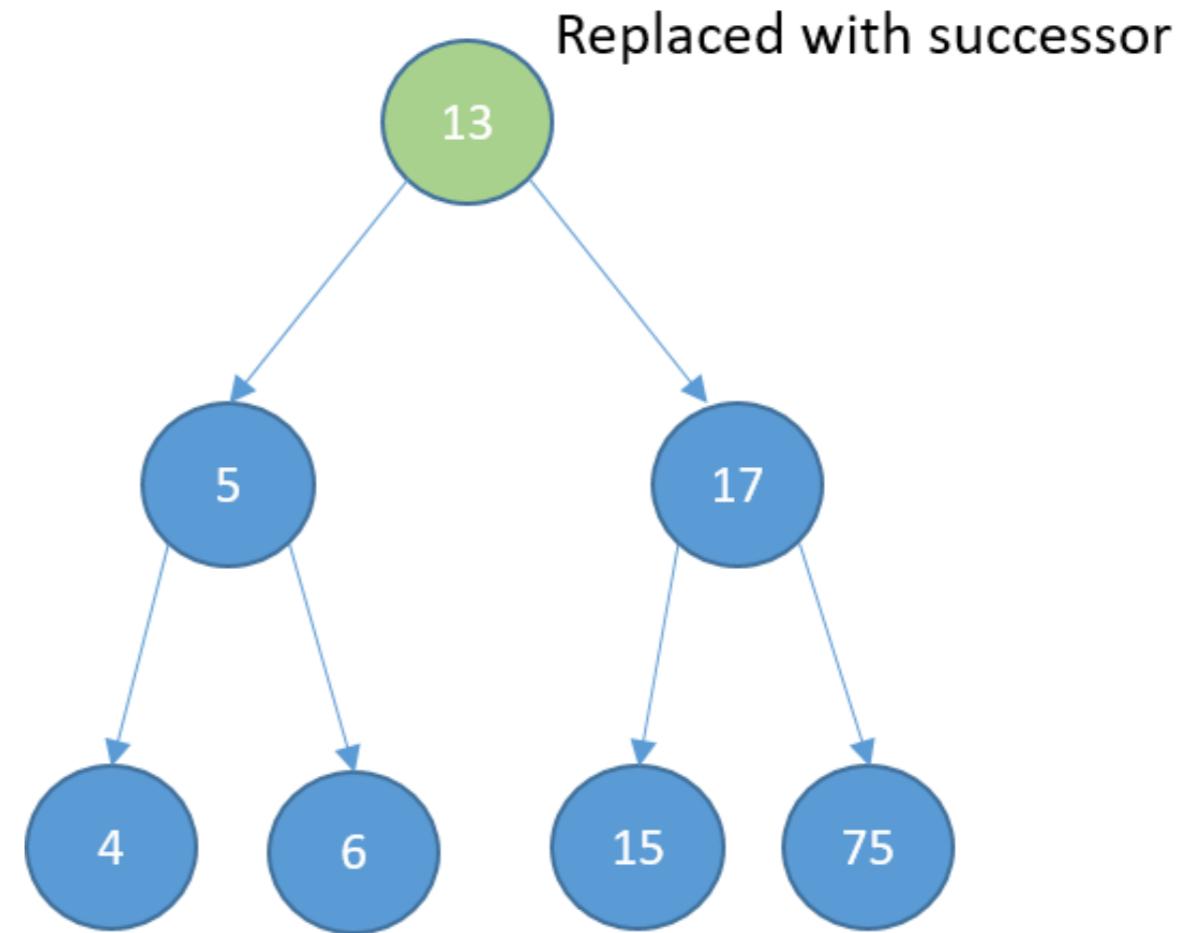
Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child
 - keep visiting the left nodes until the end



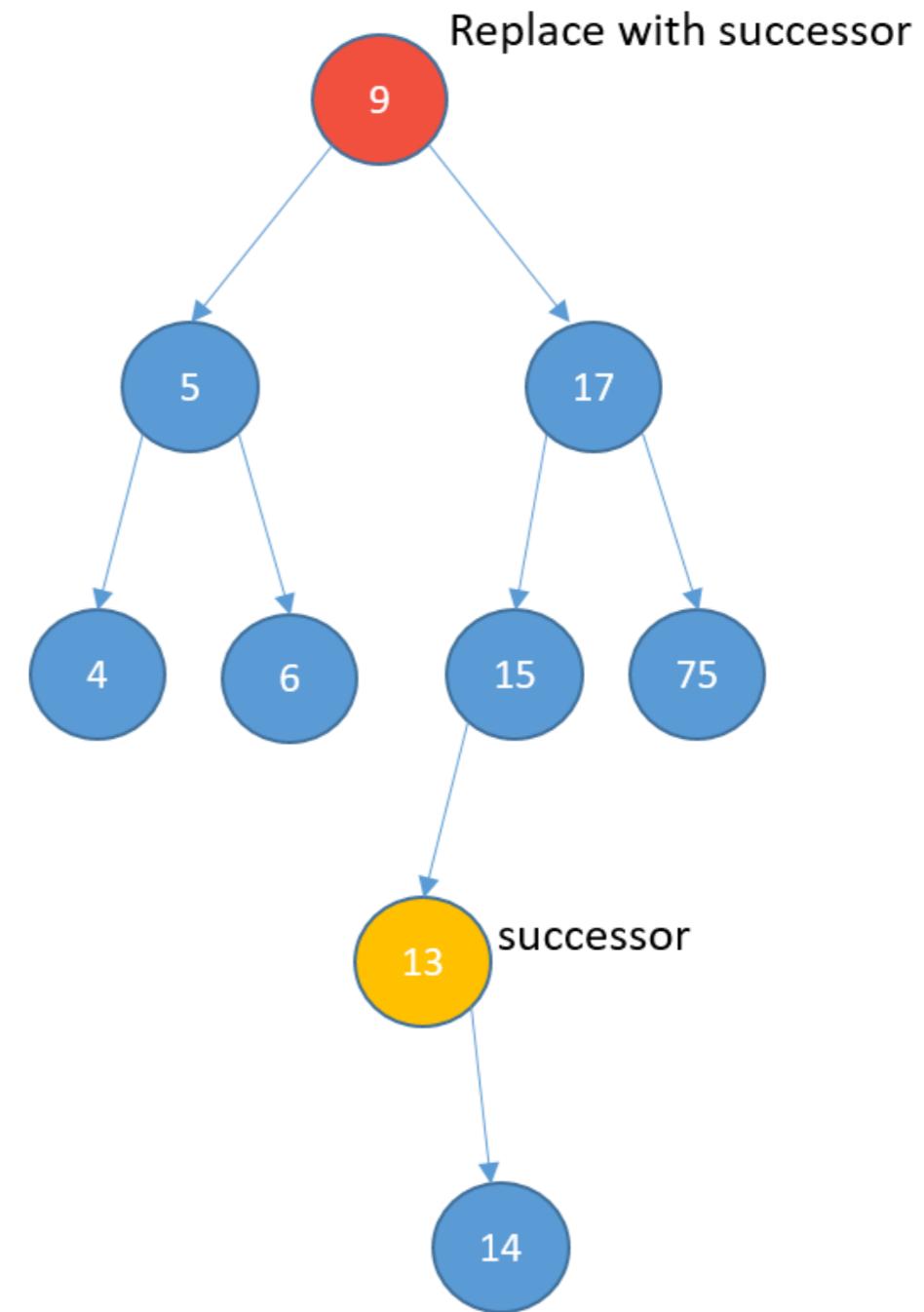
Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child
 - keep visiting the left nodes until the end



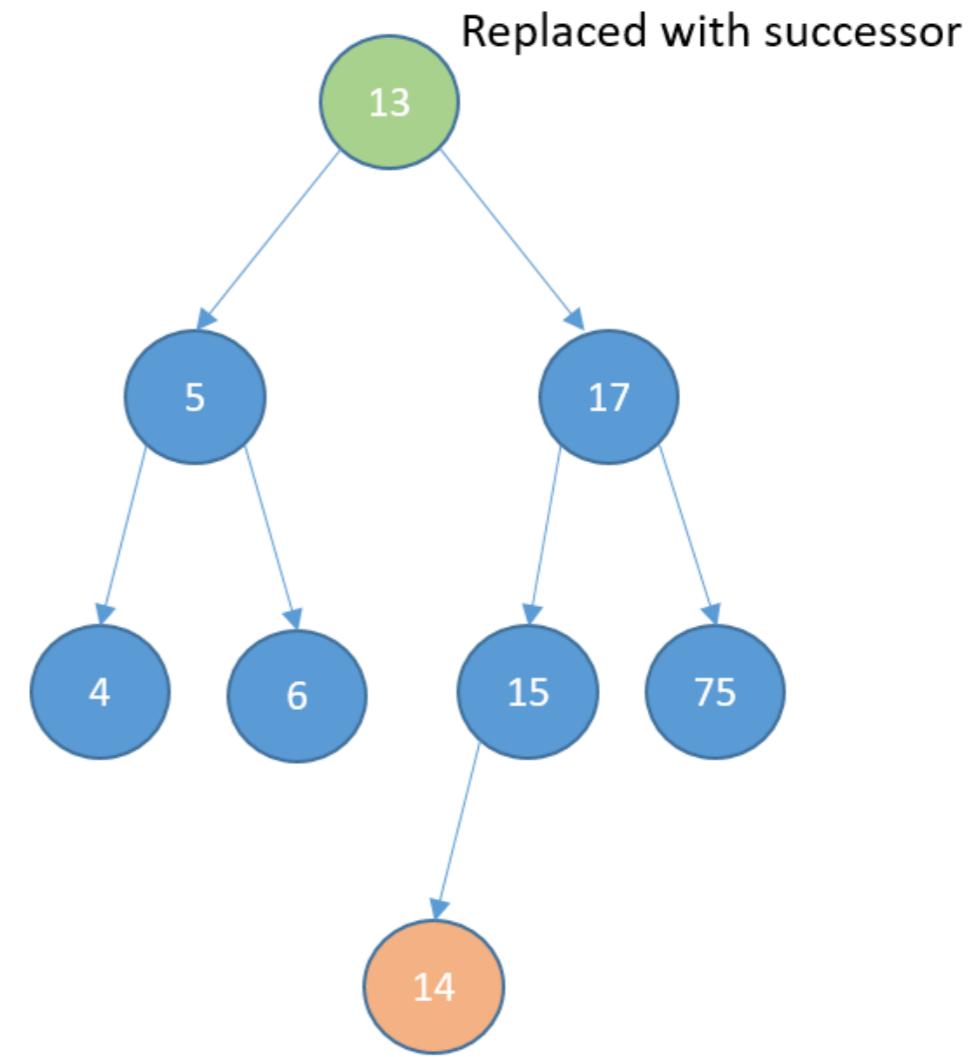
Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child
 - keep visiting the left nodes until the end
 - if the successor has a right child:



Deleting

- **Two children**
 - replace it with its successor
 - the node with the least value greater than the value of the node
 - find the successor:
 - visit the right child
 - keep visiting the left nodes until the end
 - if the successor has a right child:
 - child becomes the left child of successor's parent.



Uses

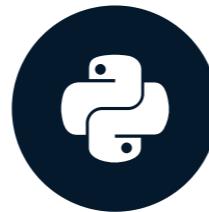
- Order lists efficiently
- Much faster at searching than arrays and linked lists
- Much faster at inserting and deleting than arrays
- Used to implement more advanced data structures:
 - dynamic sets
 - lookup tables
 - priority queues

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Depth First Search (DFS)

DATA STRUCTURES AND ALGORITHMS IN PYTHON



Miriam Antona
Software engineer

Tree/graph traversal

- Process of visiting **all nodes**
- Depth first search (DFS)
- Breadth first search (BFS)

Depth first search - binary trees

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

In-order traversal

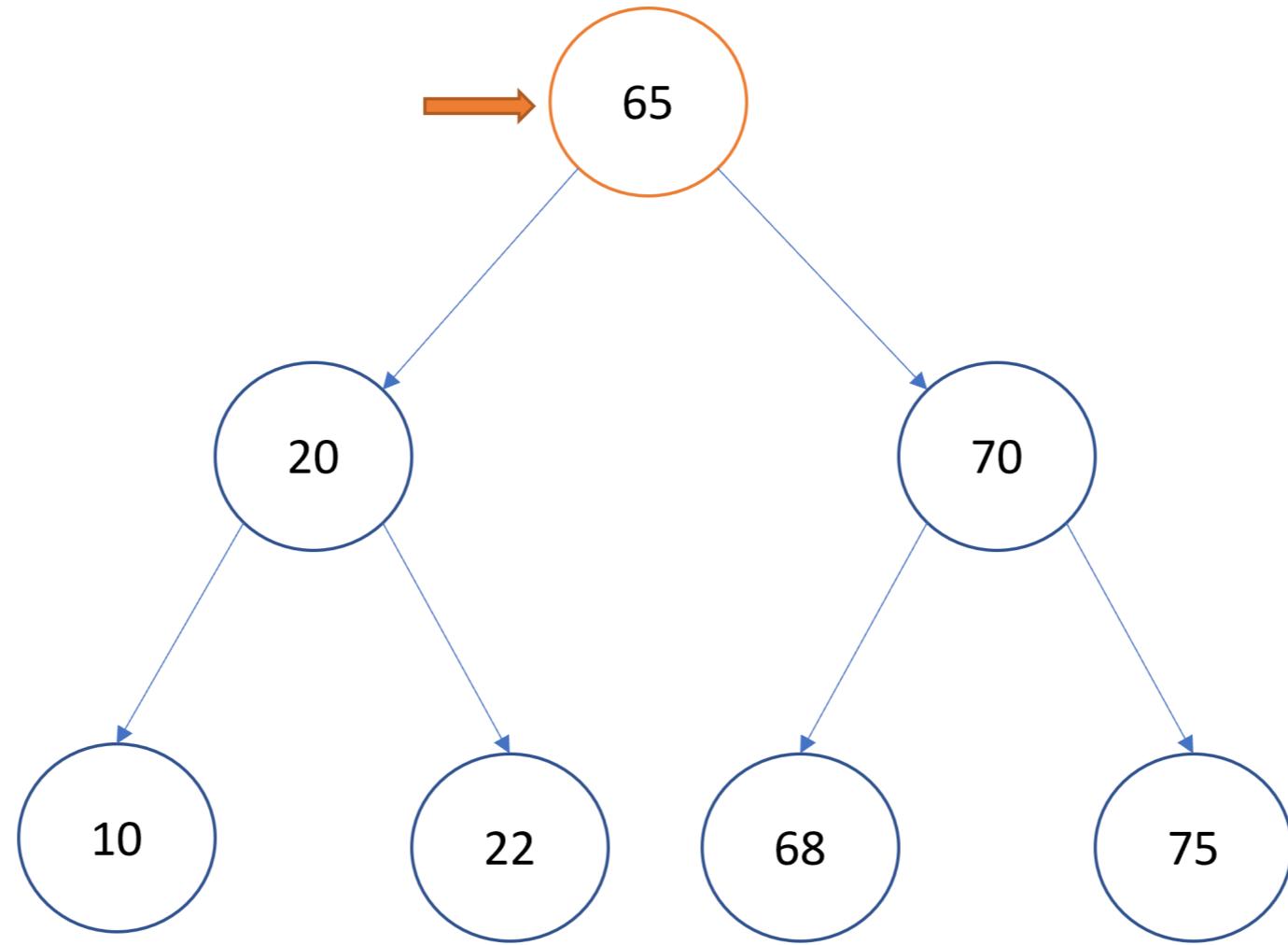
- Order: Left

In-order traversal

- Order: Left -> Current

In-order traversal

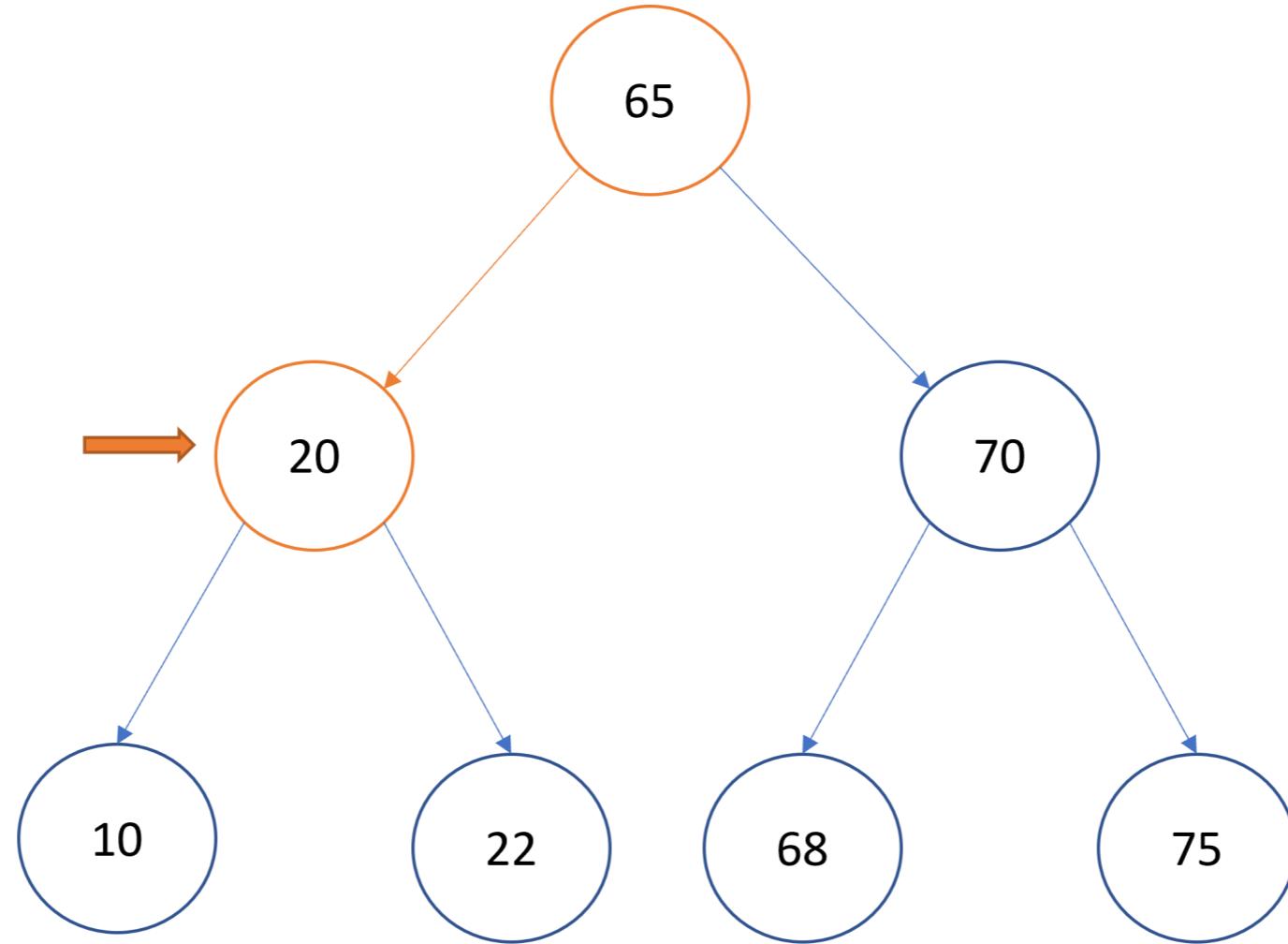
- Order: Left -> Current -> Right



Visited nodes:

In-order traversal

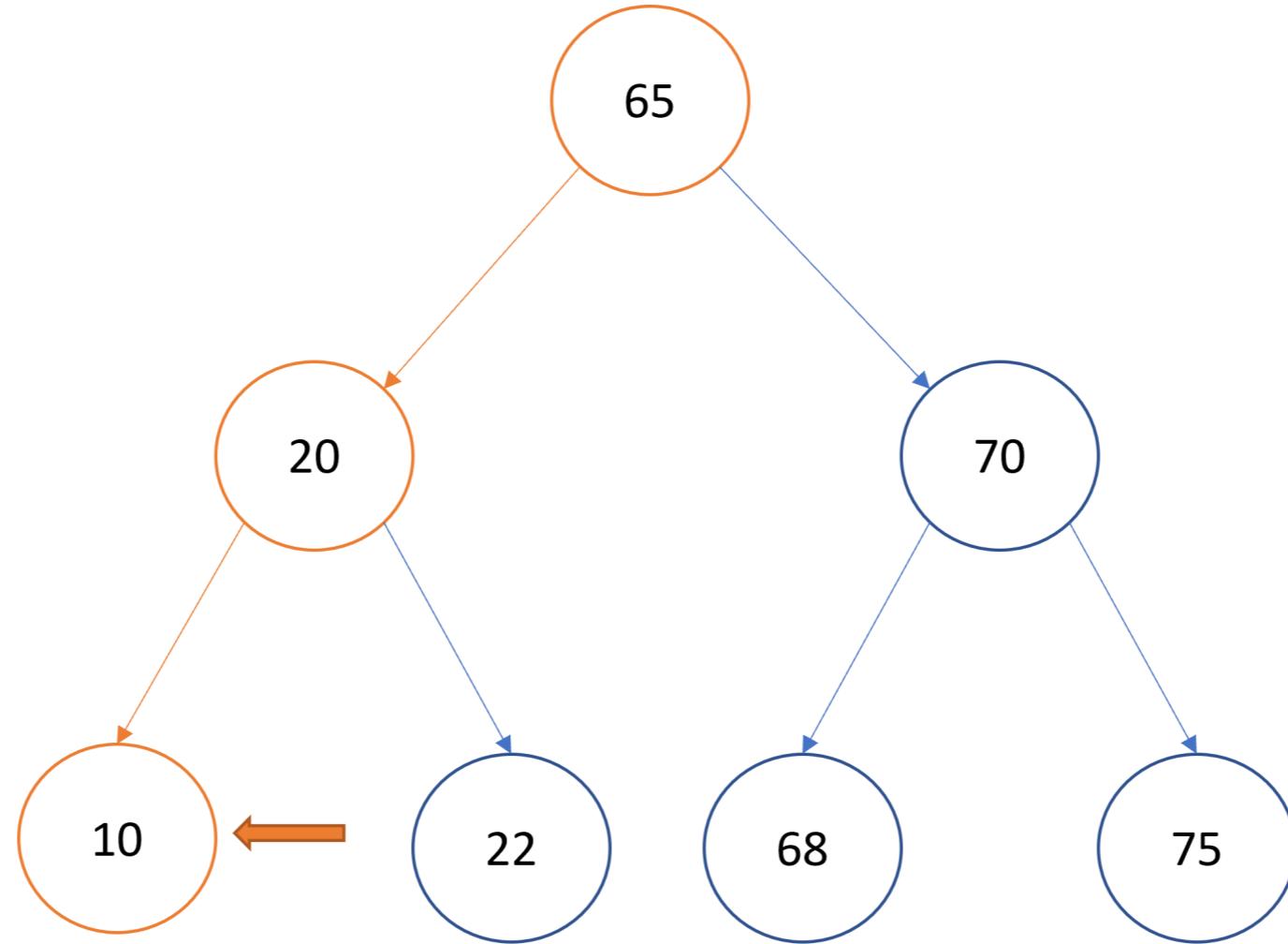
- Order: Left -> Current -> Right



Visited nodes:

In-order traversal

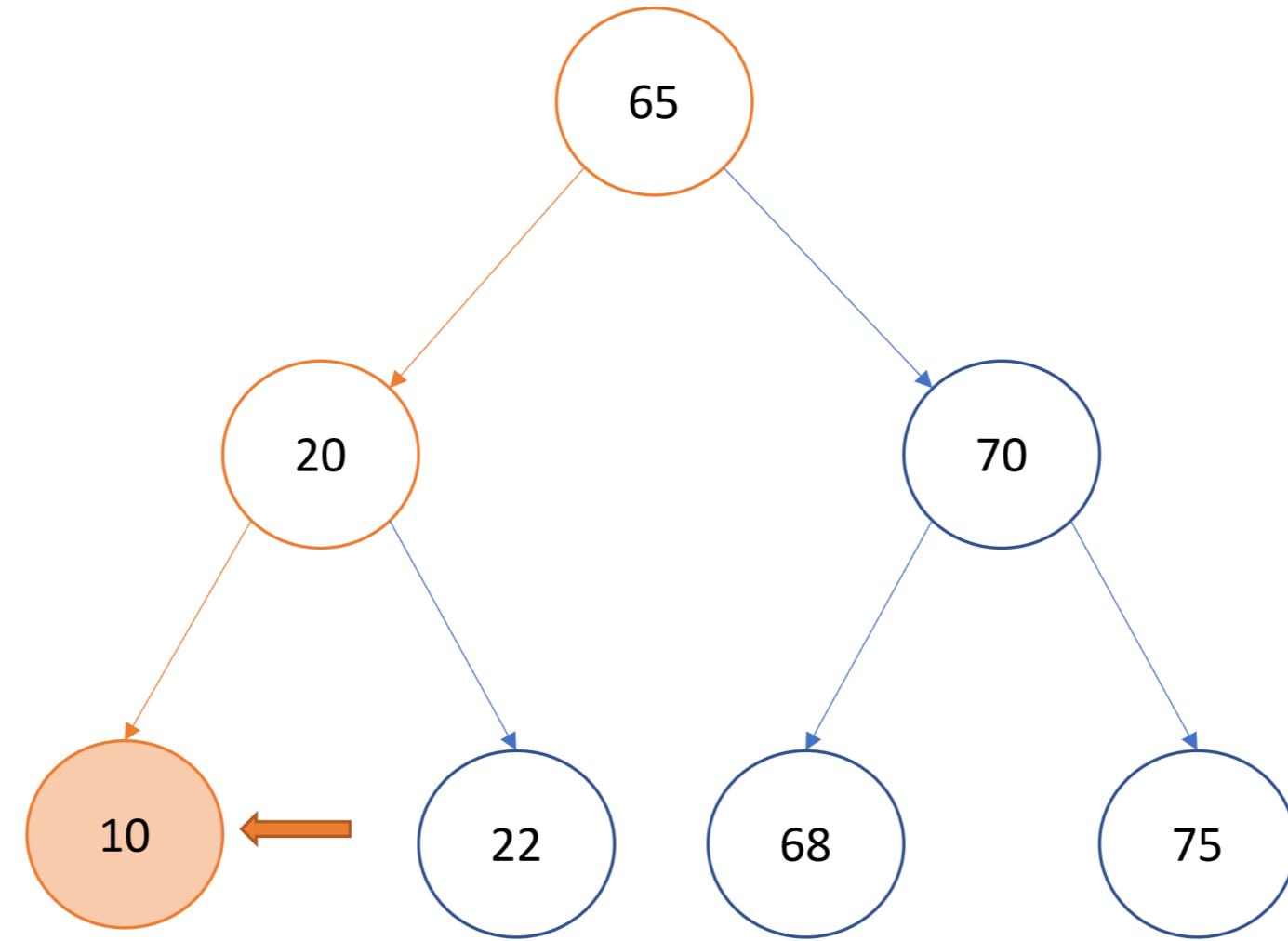
- Order: Left -> Current -> Right



Visited nodes:

In-order traversal

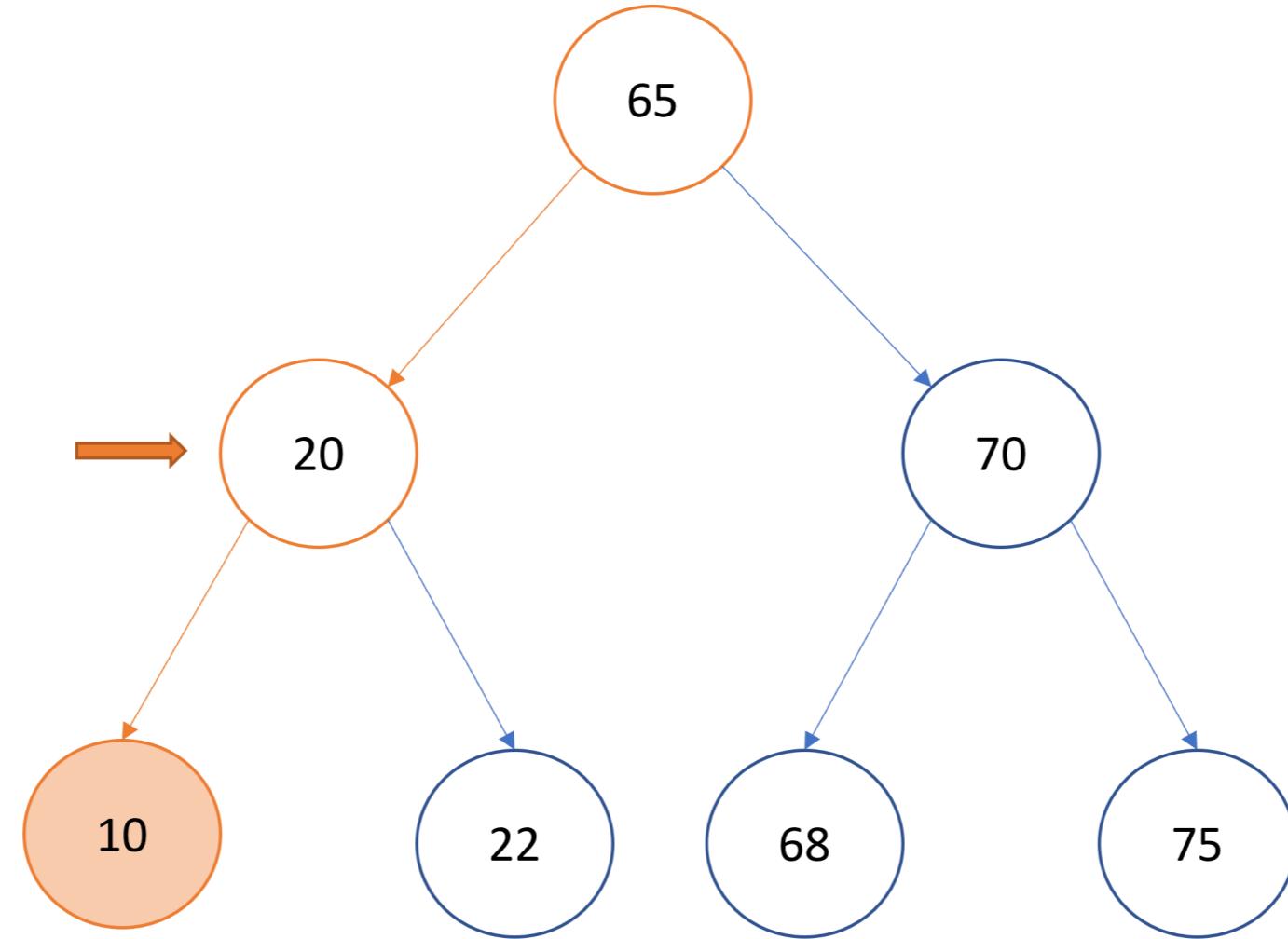
- Order: Left -> Current -> Right



Visited nodes: 10

In-order traversal

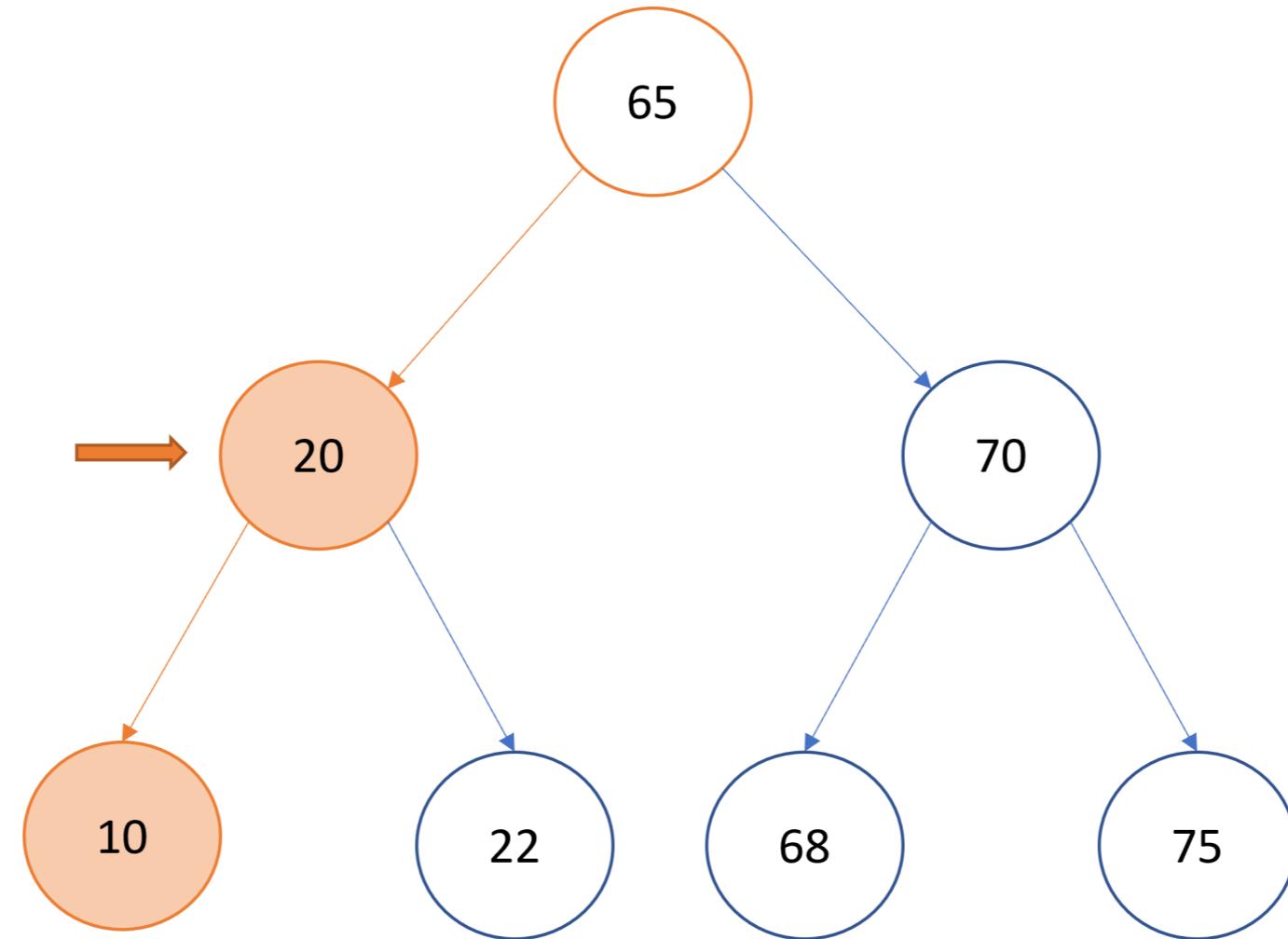
- Order: Left -> Current -> Right



Visited nodes: 10

In-order traversal

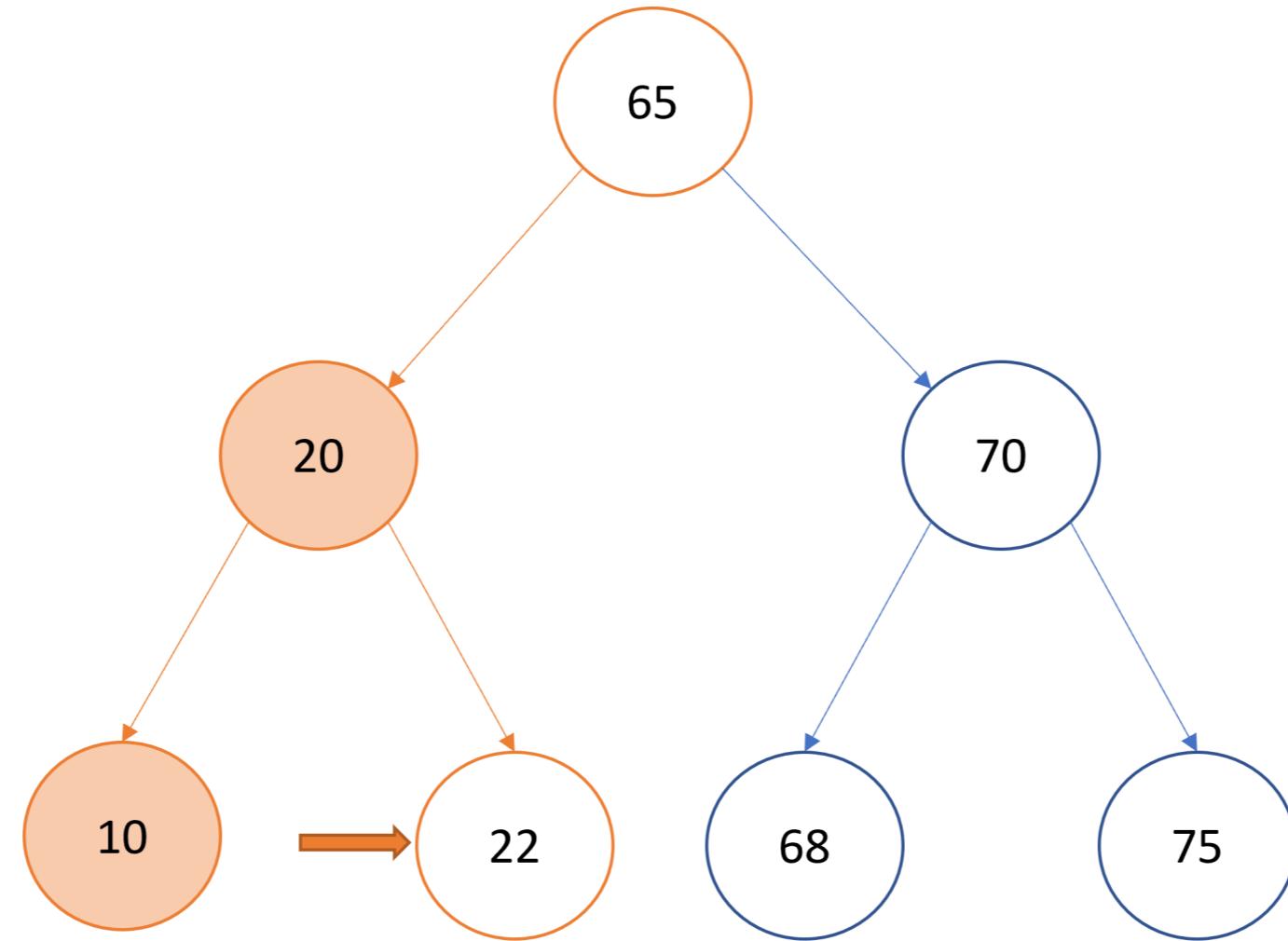
- Order: Left -> Current -> Right



Visited nodes: 10, 20

In-order traversal

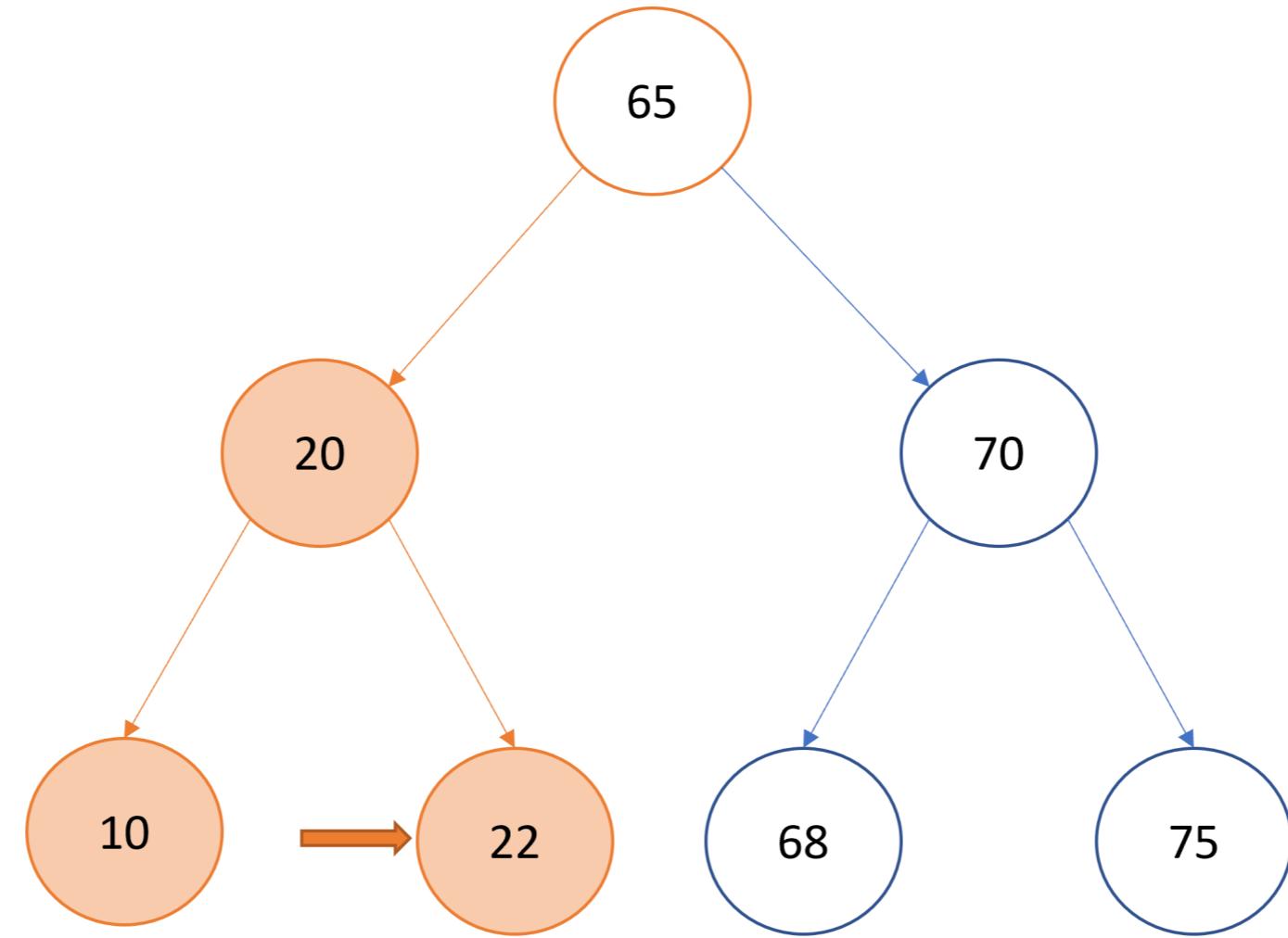
- Order: Left -> Current -> Right



Visited nodes: 10, 20

In-order traversal

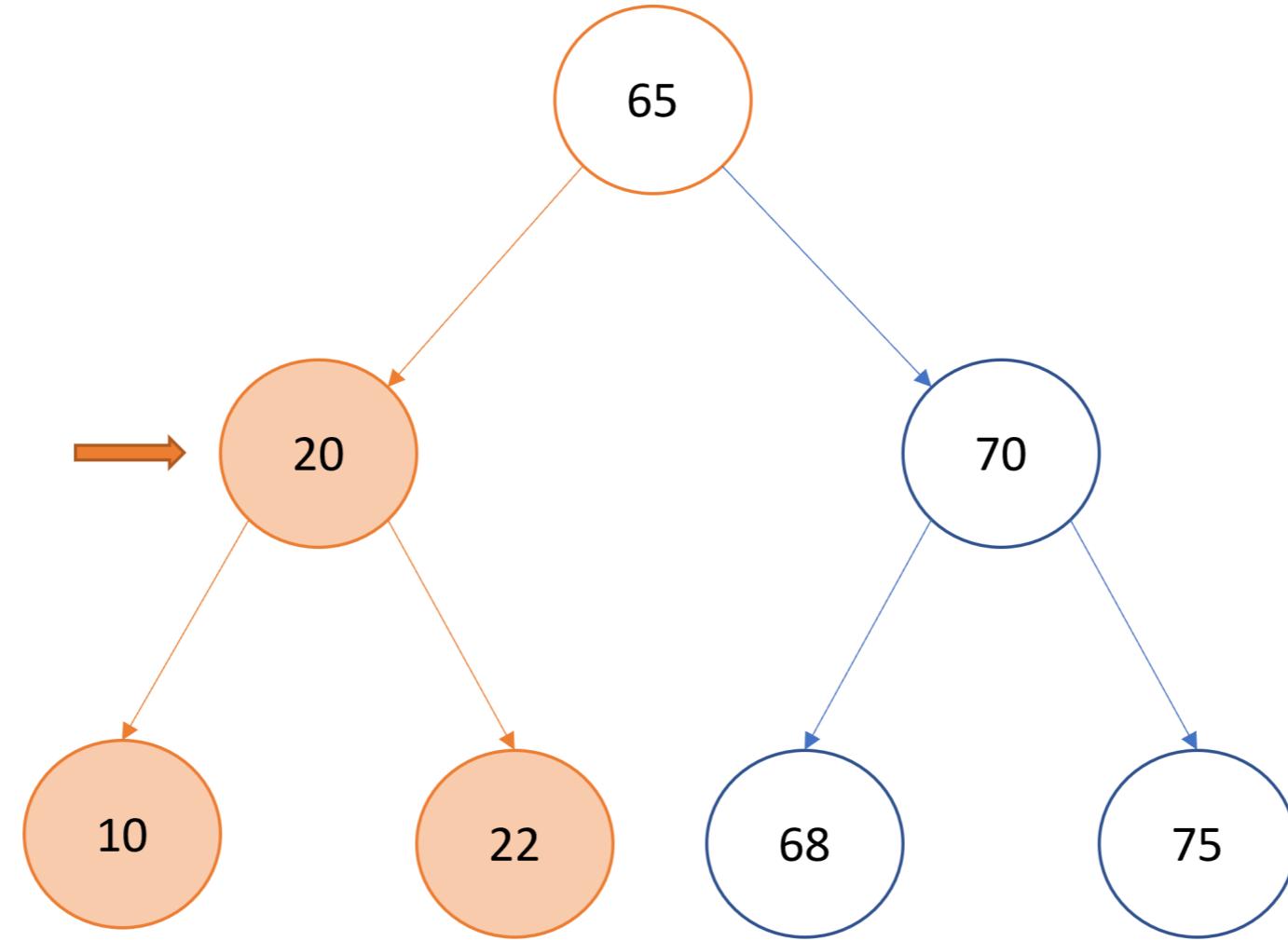
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22

In-order traversal

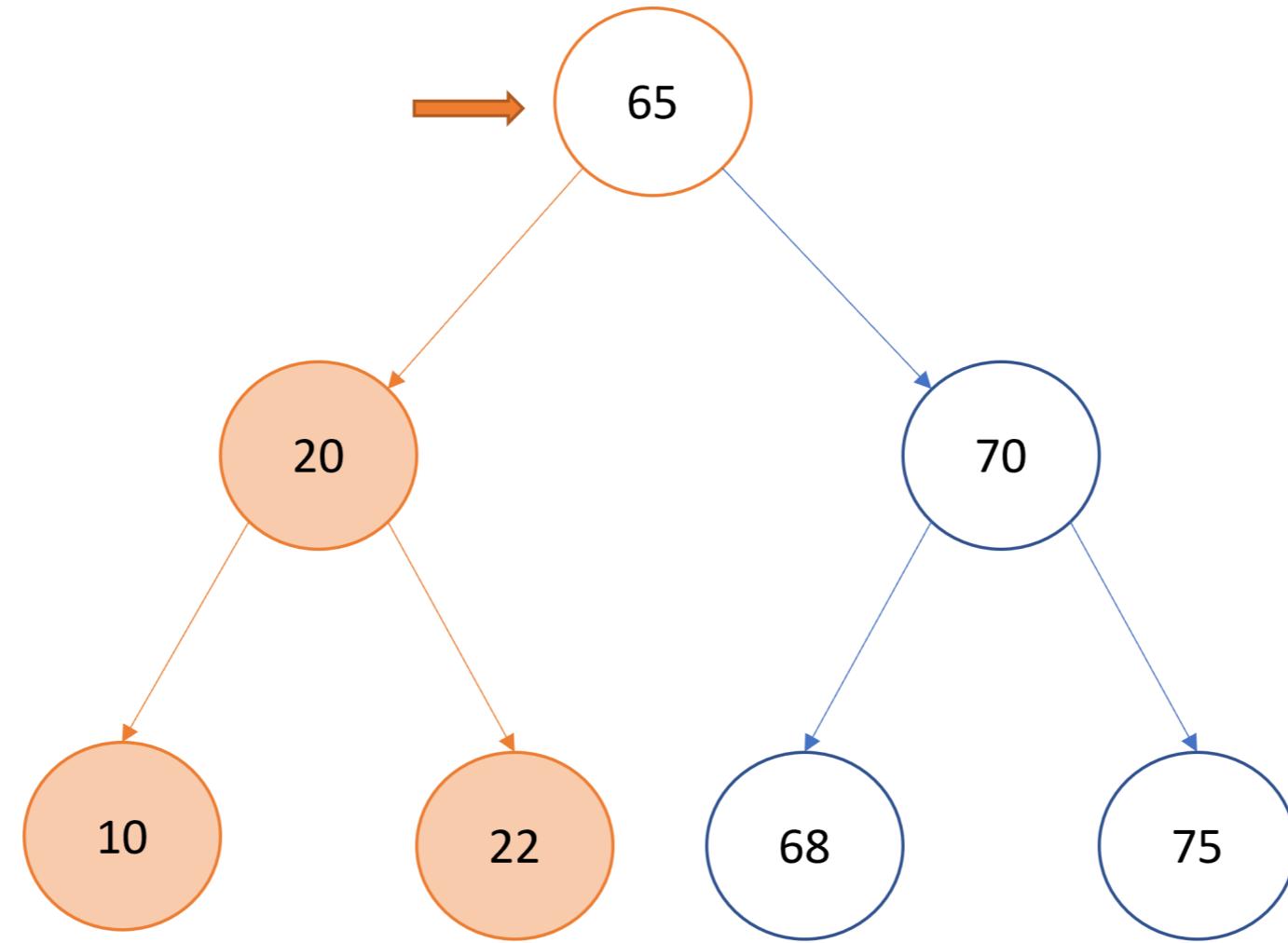
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22

In-order traversal

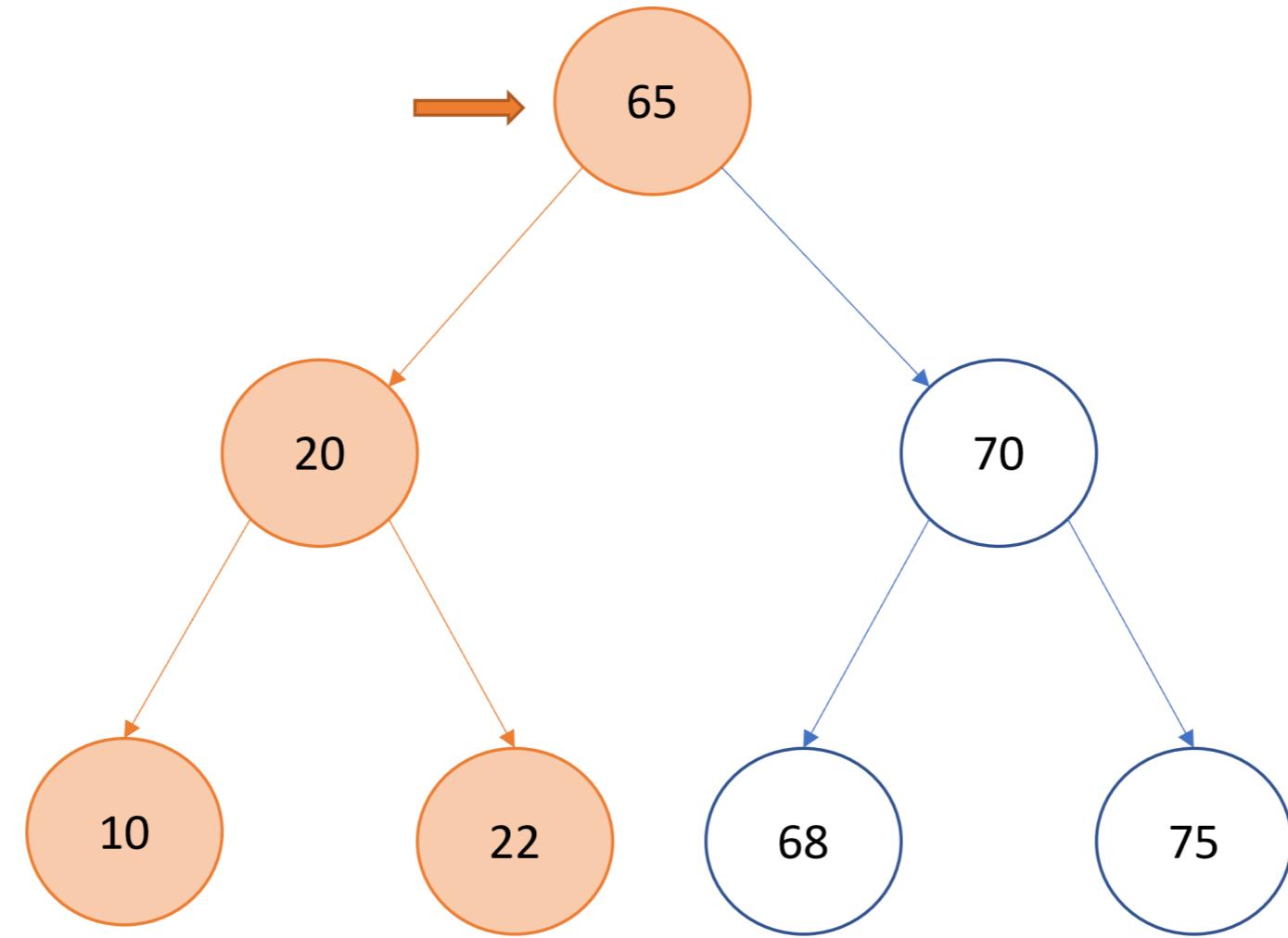
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22

In-order traversal

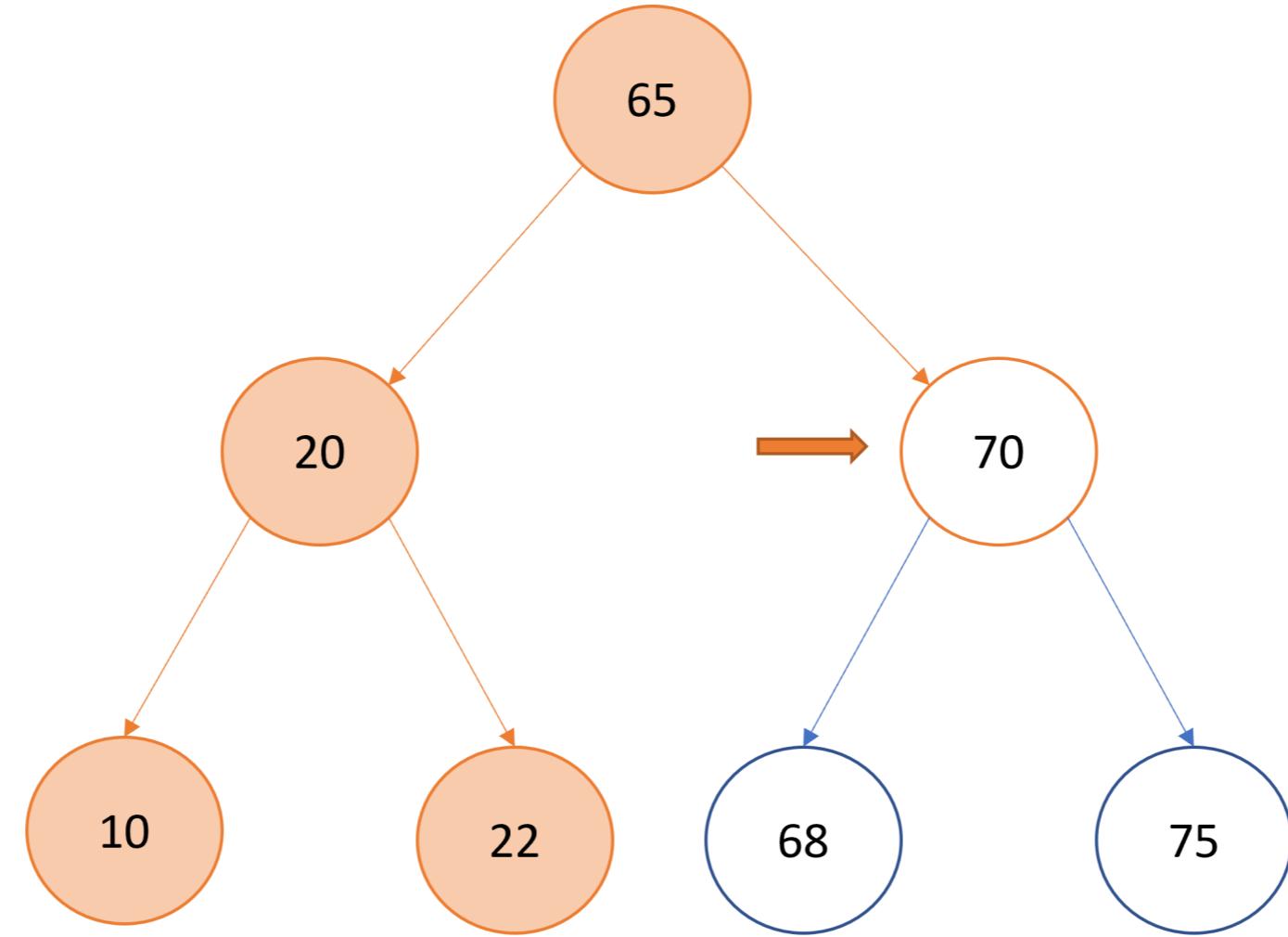
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65

In-order traversal

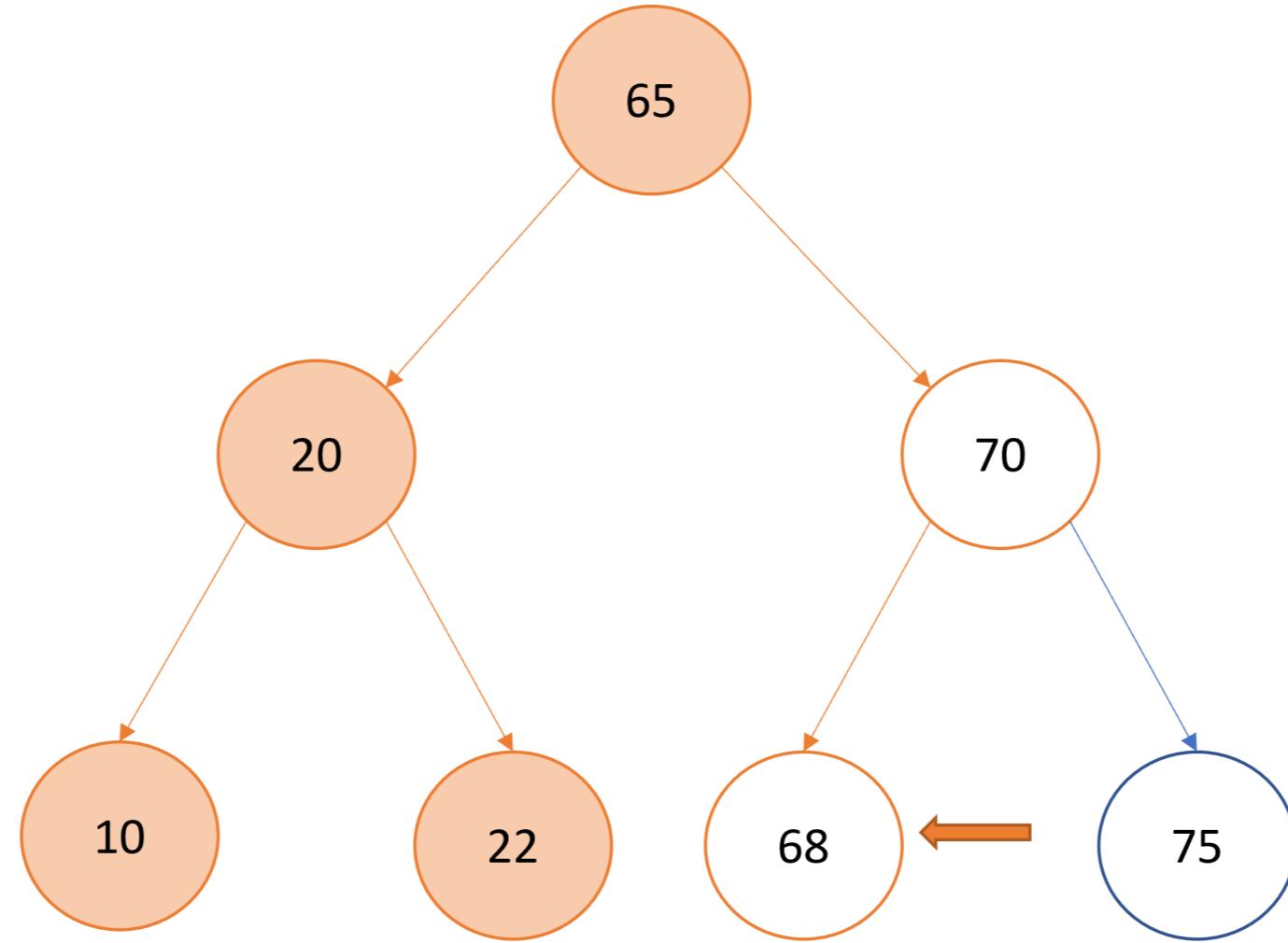
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65

In-order traversal

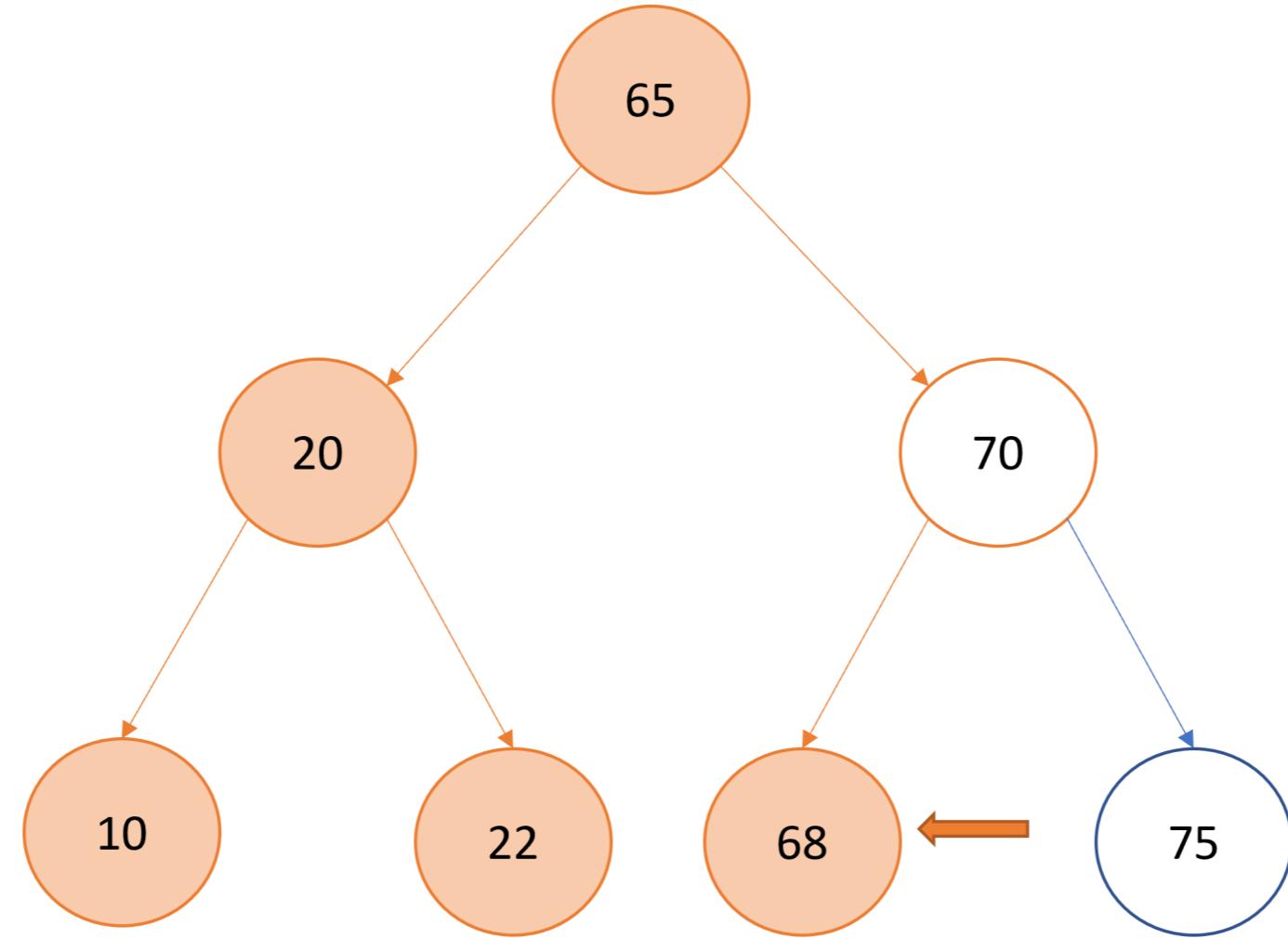
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65

In-order traversal

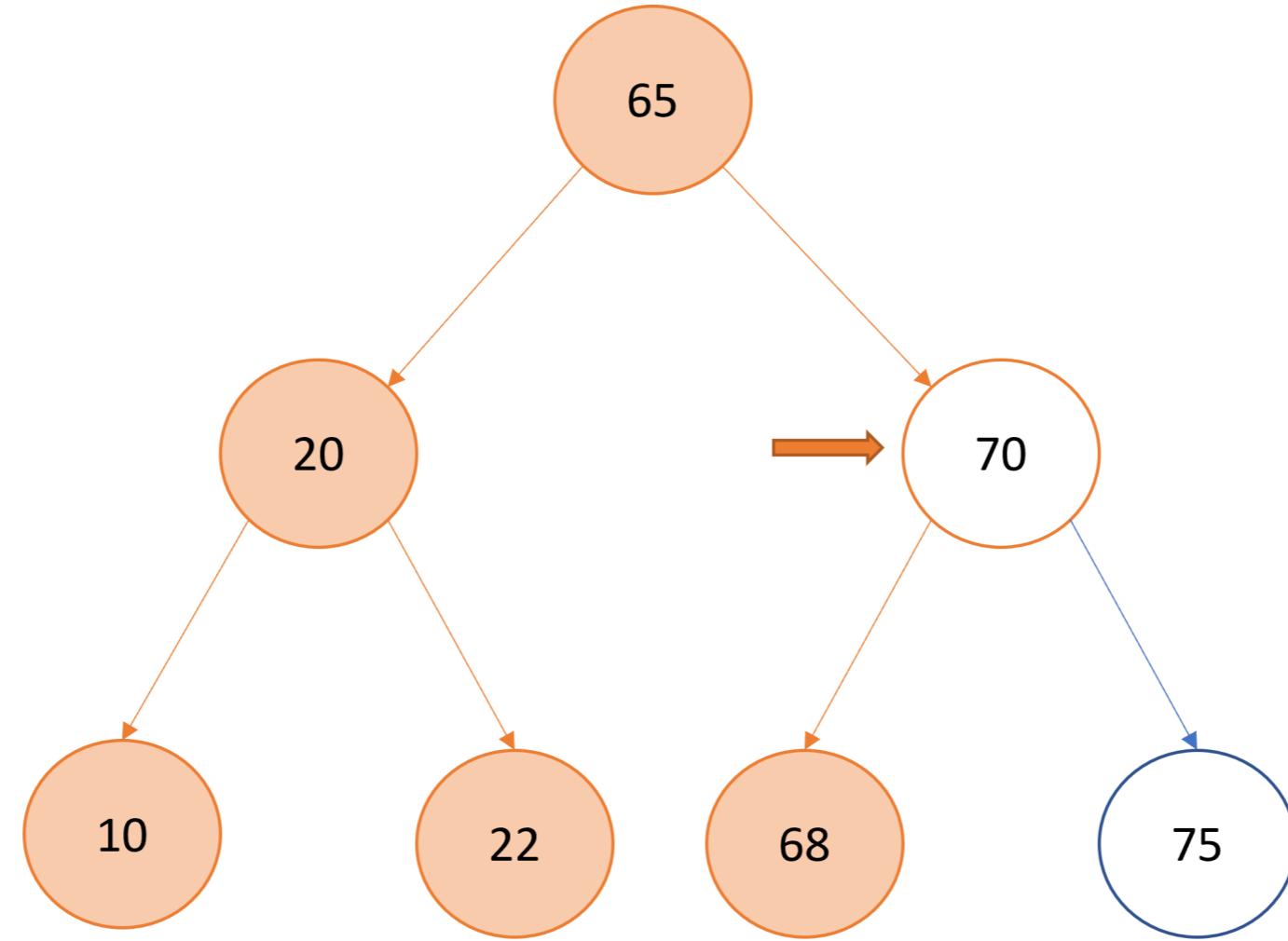
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65, 68

In-order traversal

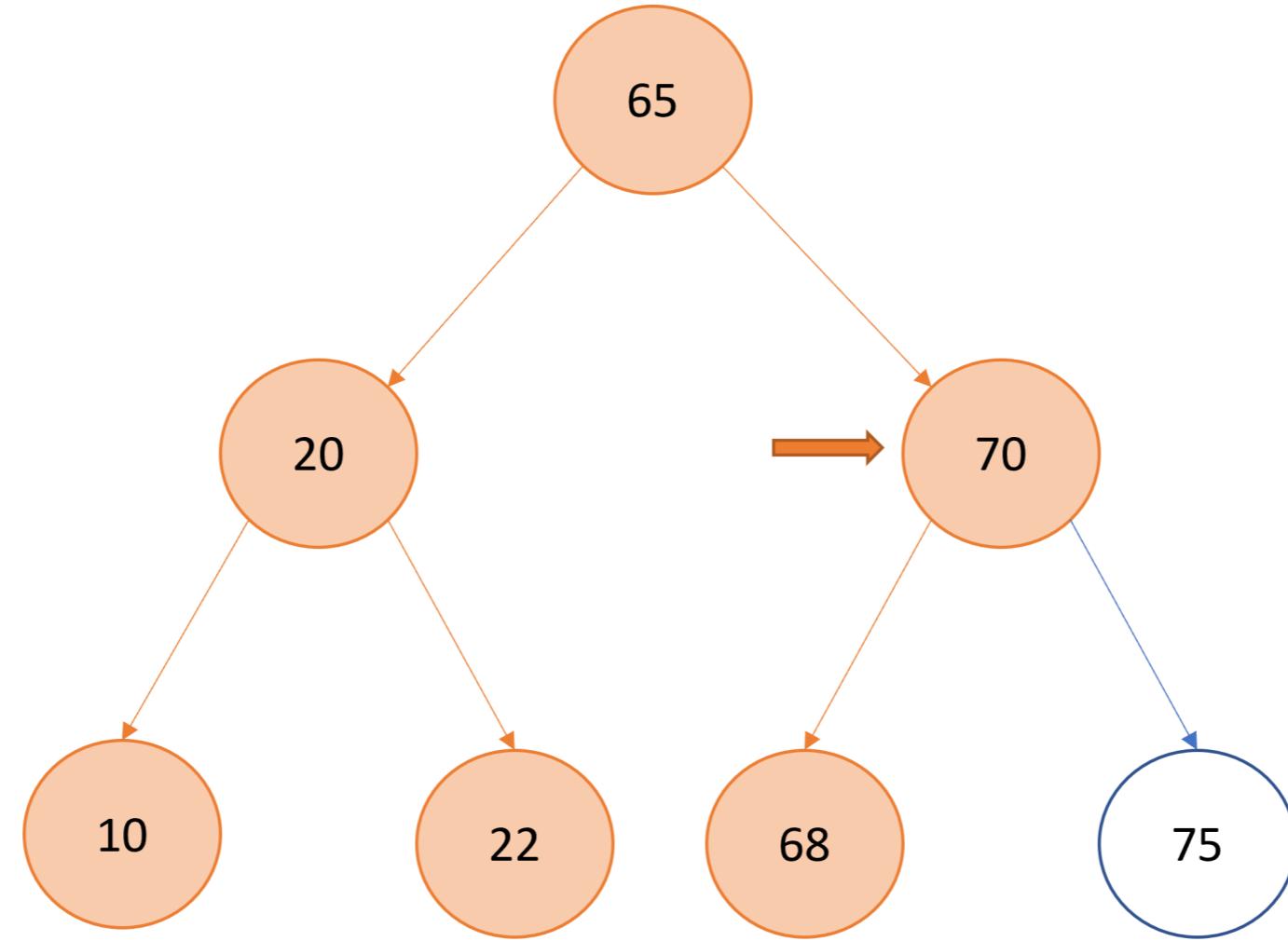
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65, 68

In-order traversal

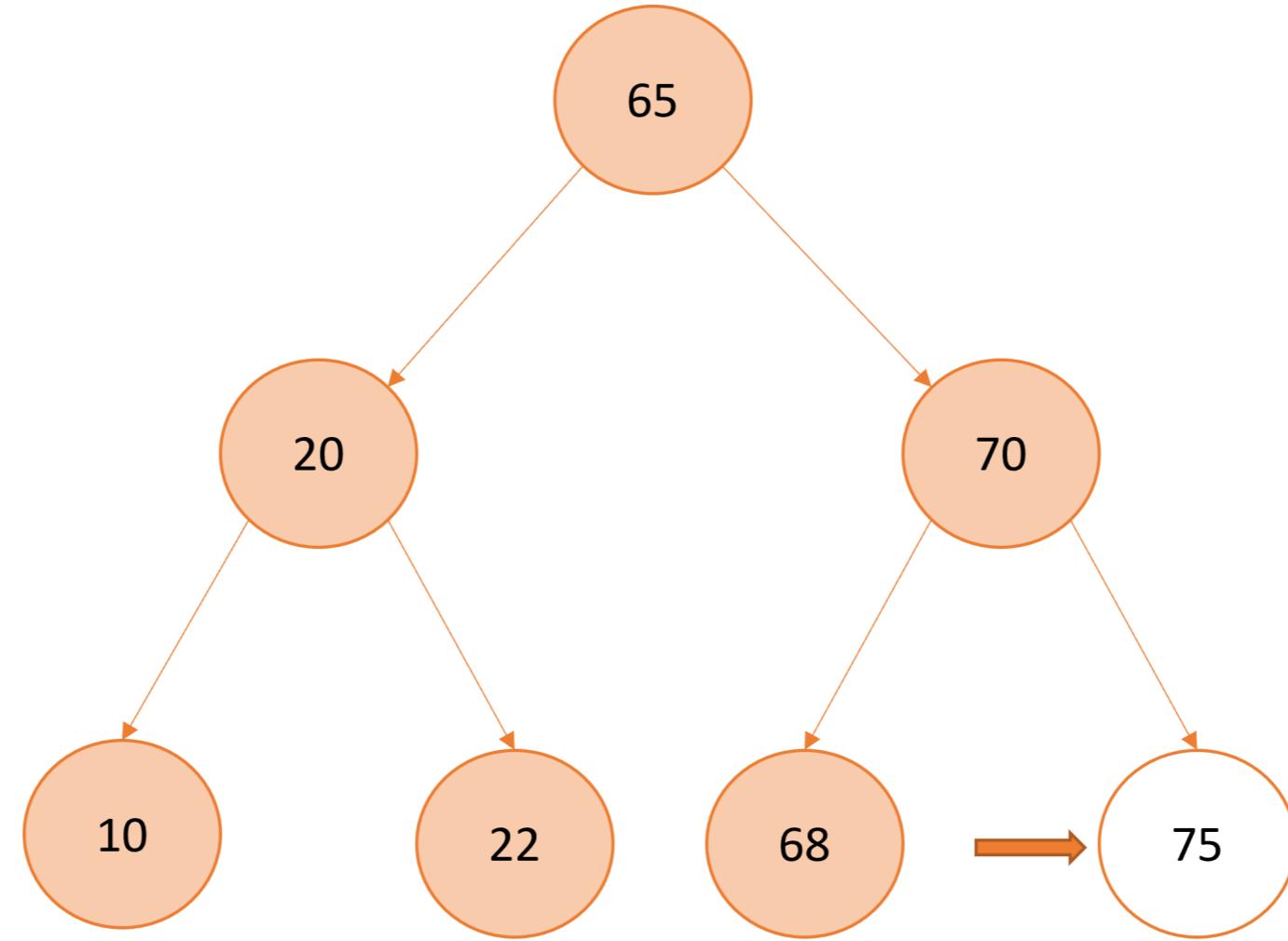
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65, 68, 70

In-order traversal

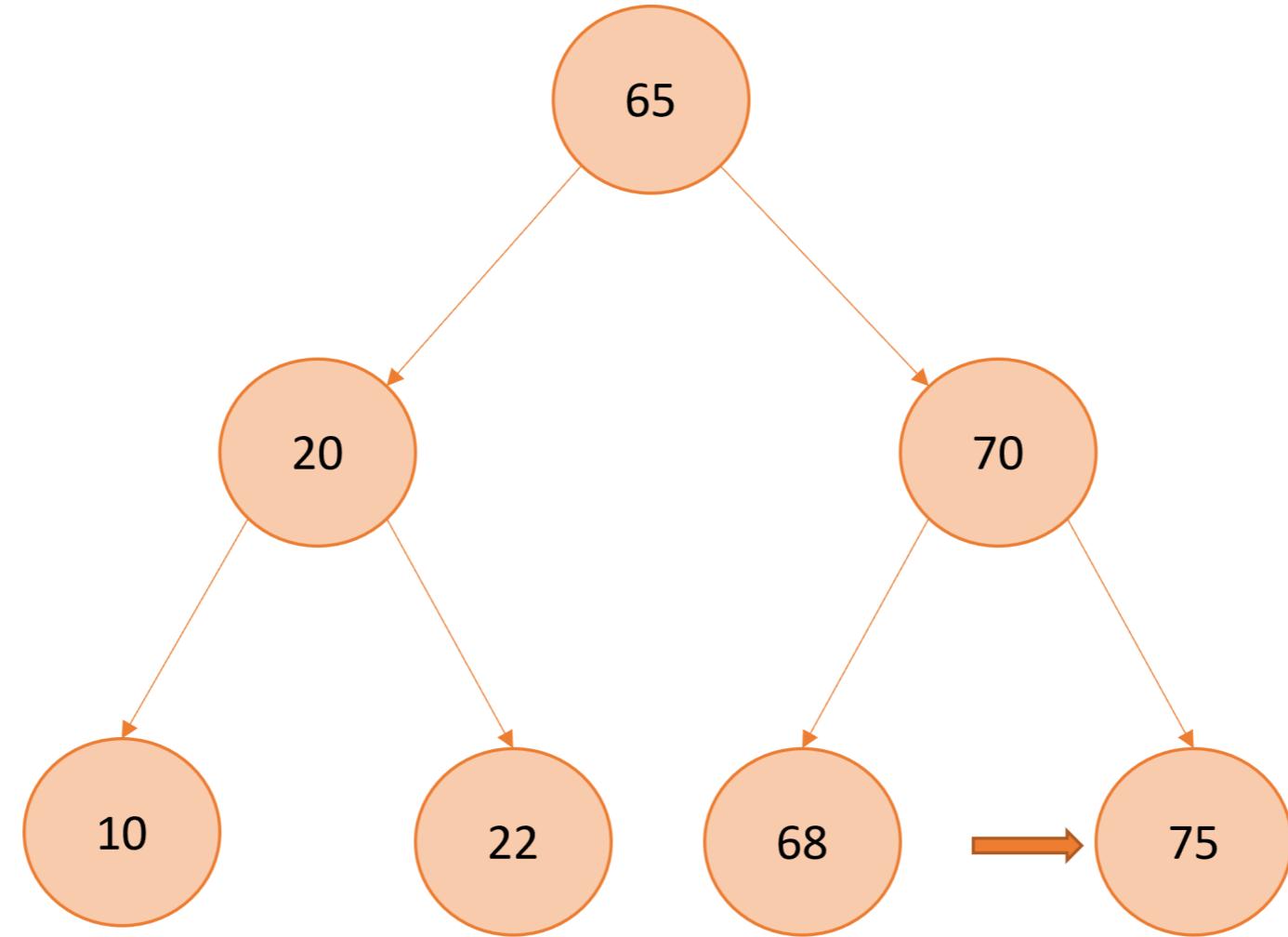
- Order: Left -> Current -> Right



Visited nodes: 10, 20, 22, 65, 68, 70

In-order traversal

- Order: Left -> Current -> Right



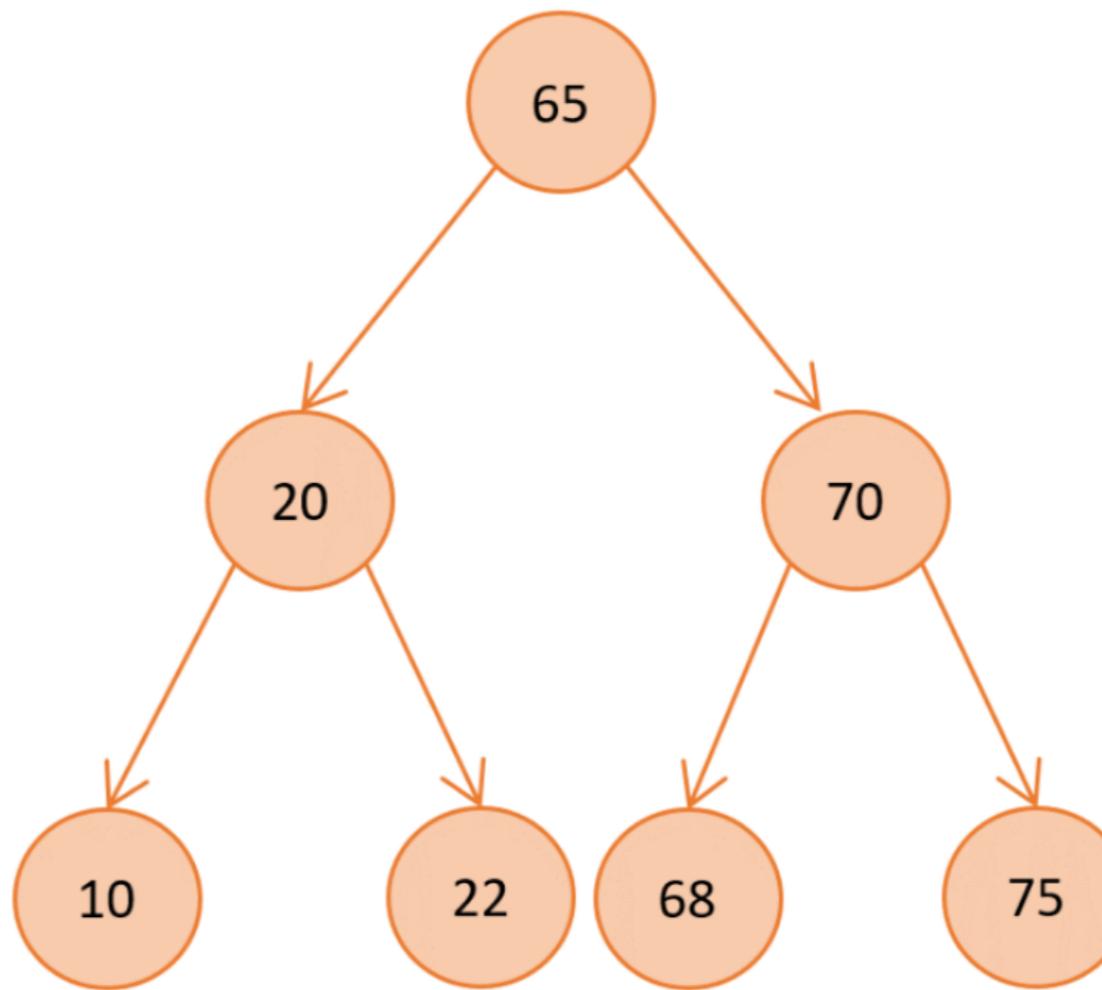
Visited nodes: 10, 20, 22, 65, 68, 70, 75

In-order traversal - implementation

- Order: Left -> Current -> Right

```
def in_order(self, current_node):  
    if current_node:  
        self.in_order(current_node.left_child)  
        print(current_node.data)  
        self.in_order(current_node.right_child)  
  
my_tree.in_order(my_tree.root)
```

10
20
22
65
68
70
75



- Complexity: $O(n)$
 - $n \rightarrow$ number of nodes

Pre-order traversal

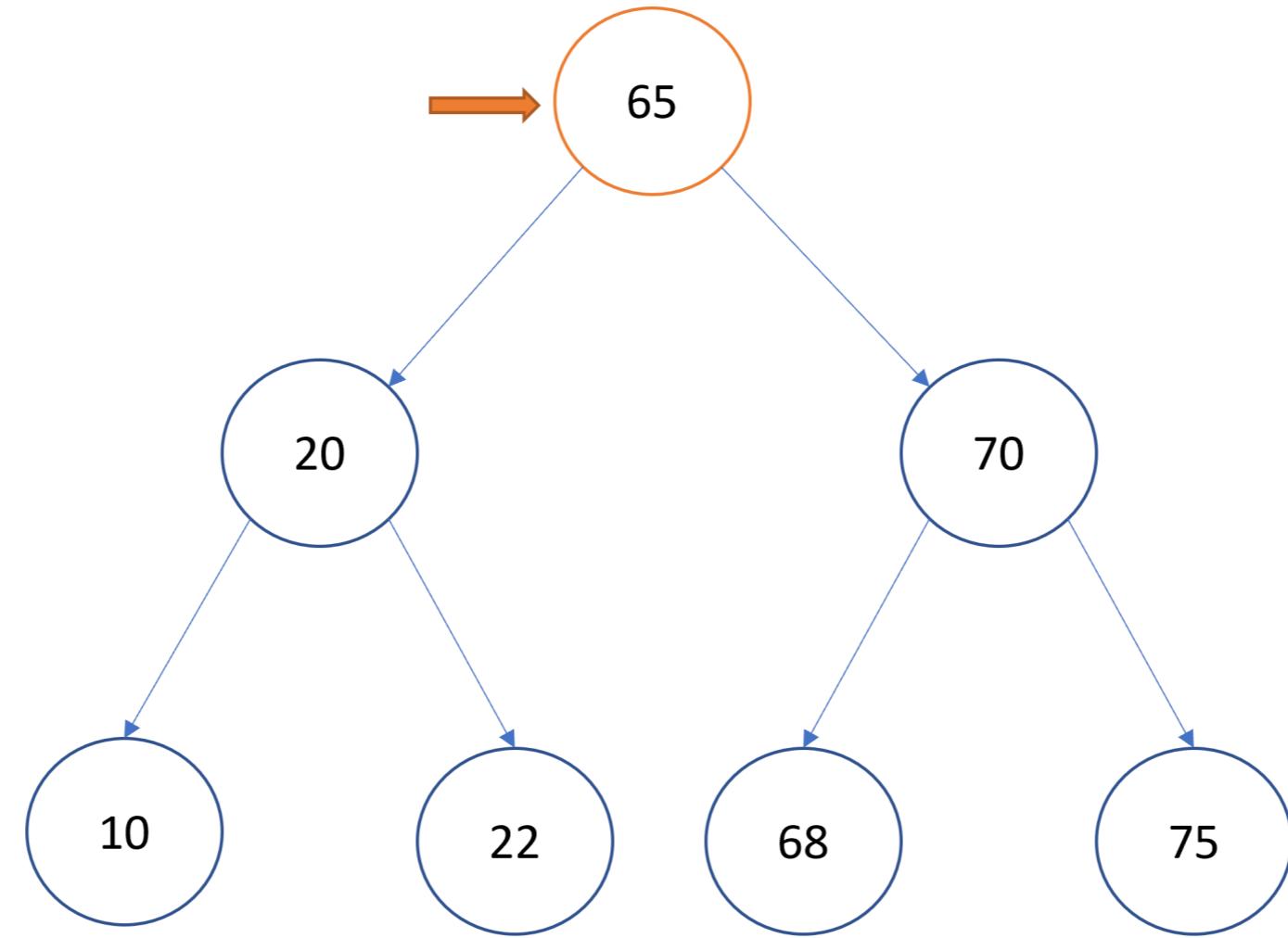
- Order: Current

Pre-order traversal

- Order: Current -> Left

Pre-order traversal

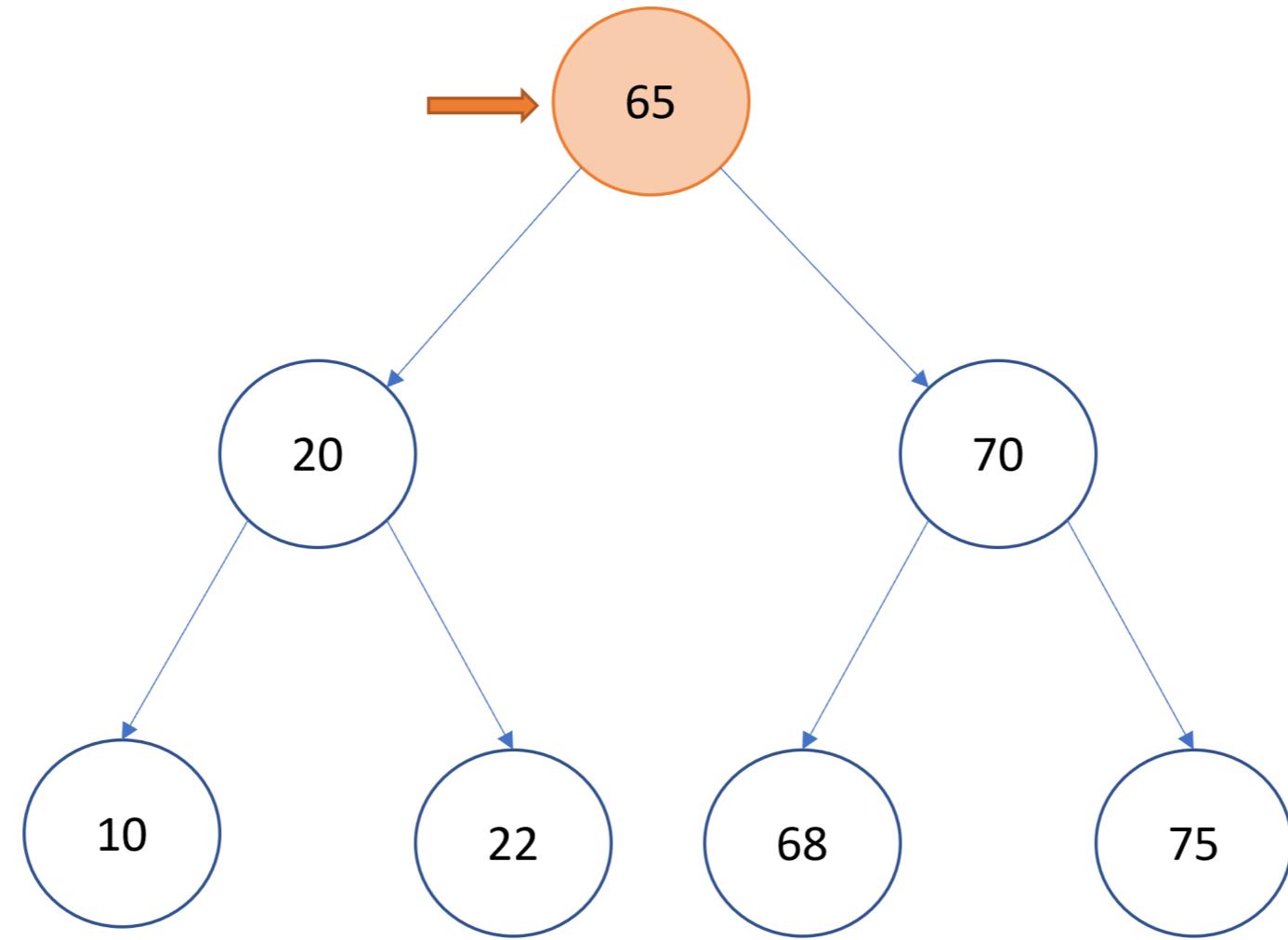
- Order: Current -> Left -> Right



Visited nodes:

Pre-order traversal

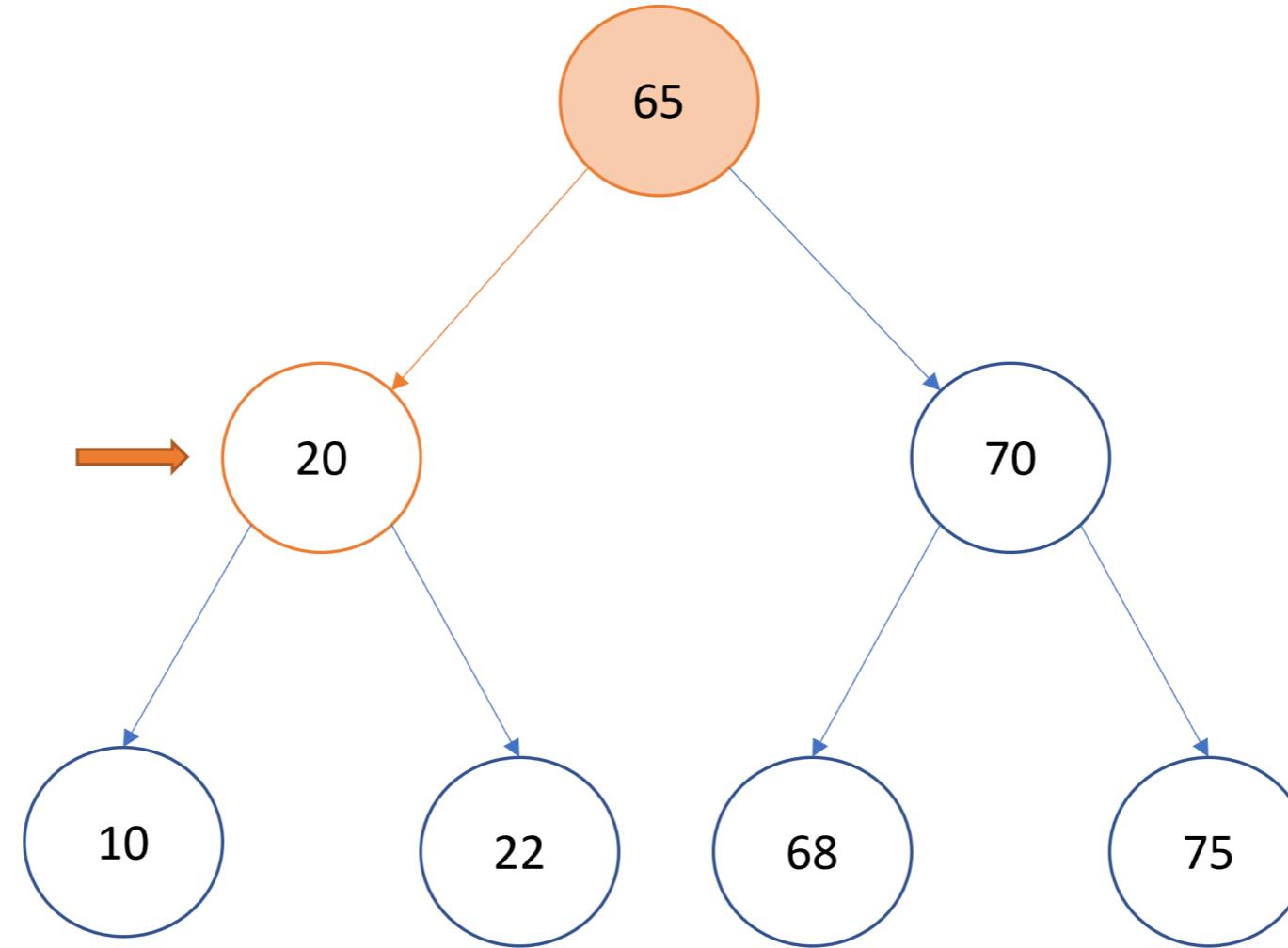
- Order: Current -> Left -> Right



Visited nodes: 65

Pre-order traversal

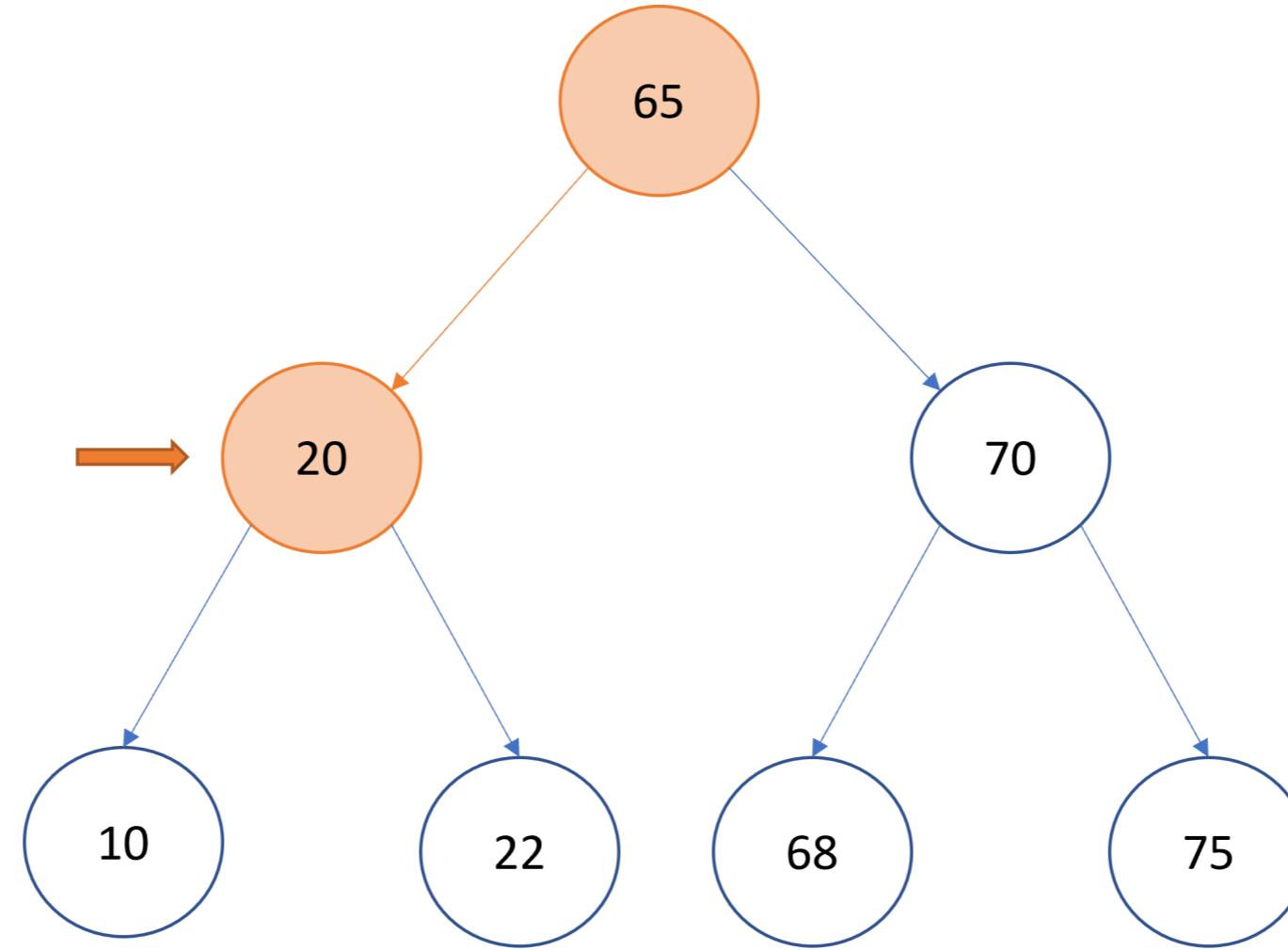
- Order: Current -> Left -> Right



Visited nodes: 65

Pre-order traversal

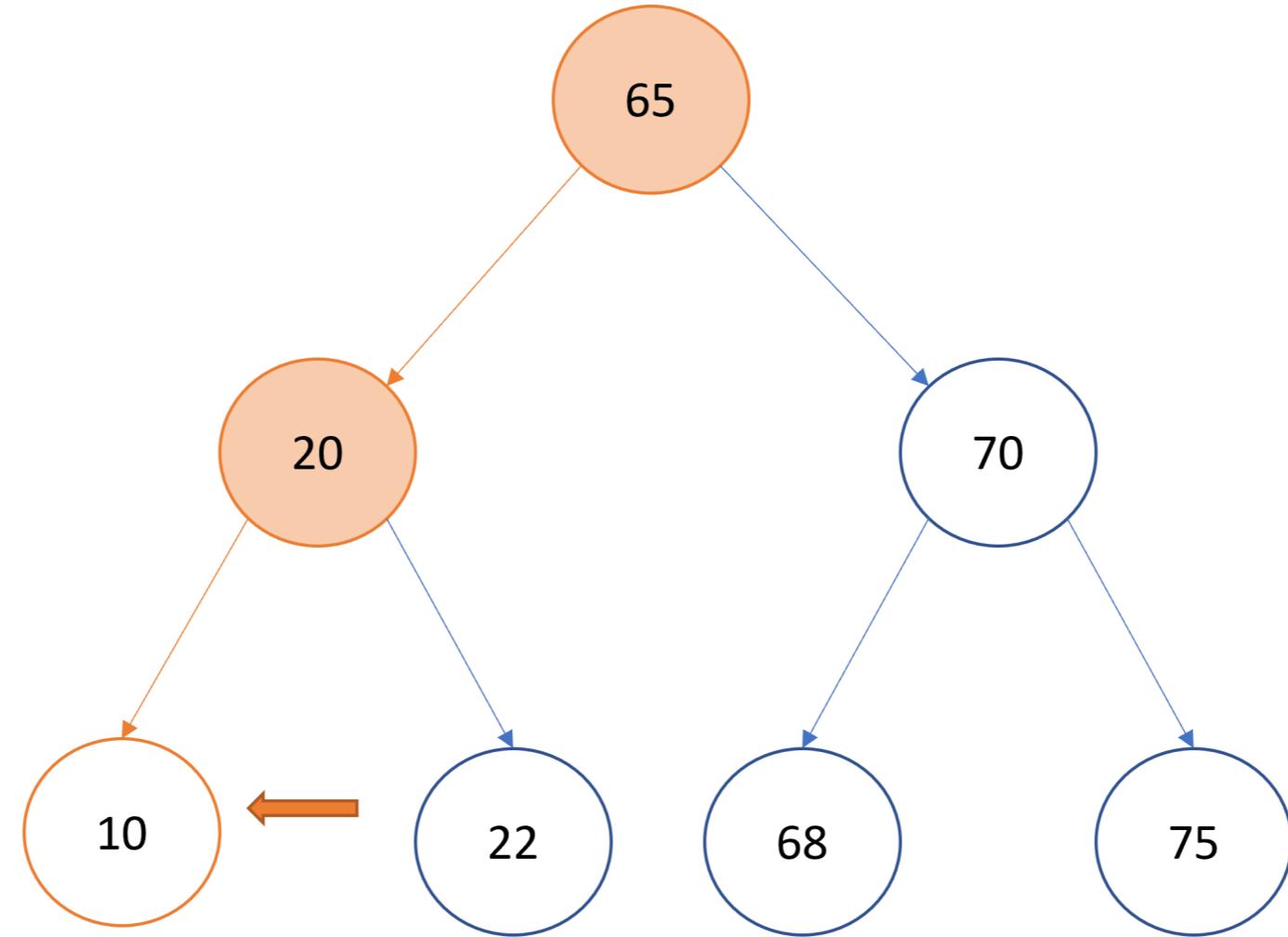
- Order: Current -> Left -> Right



Visited nodes: 65, 20

Pre-order traversal

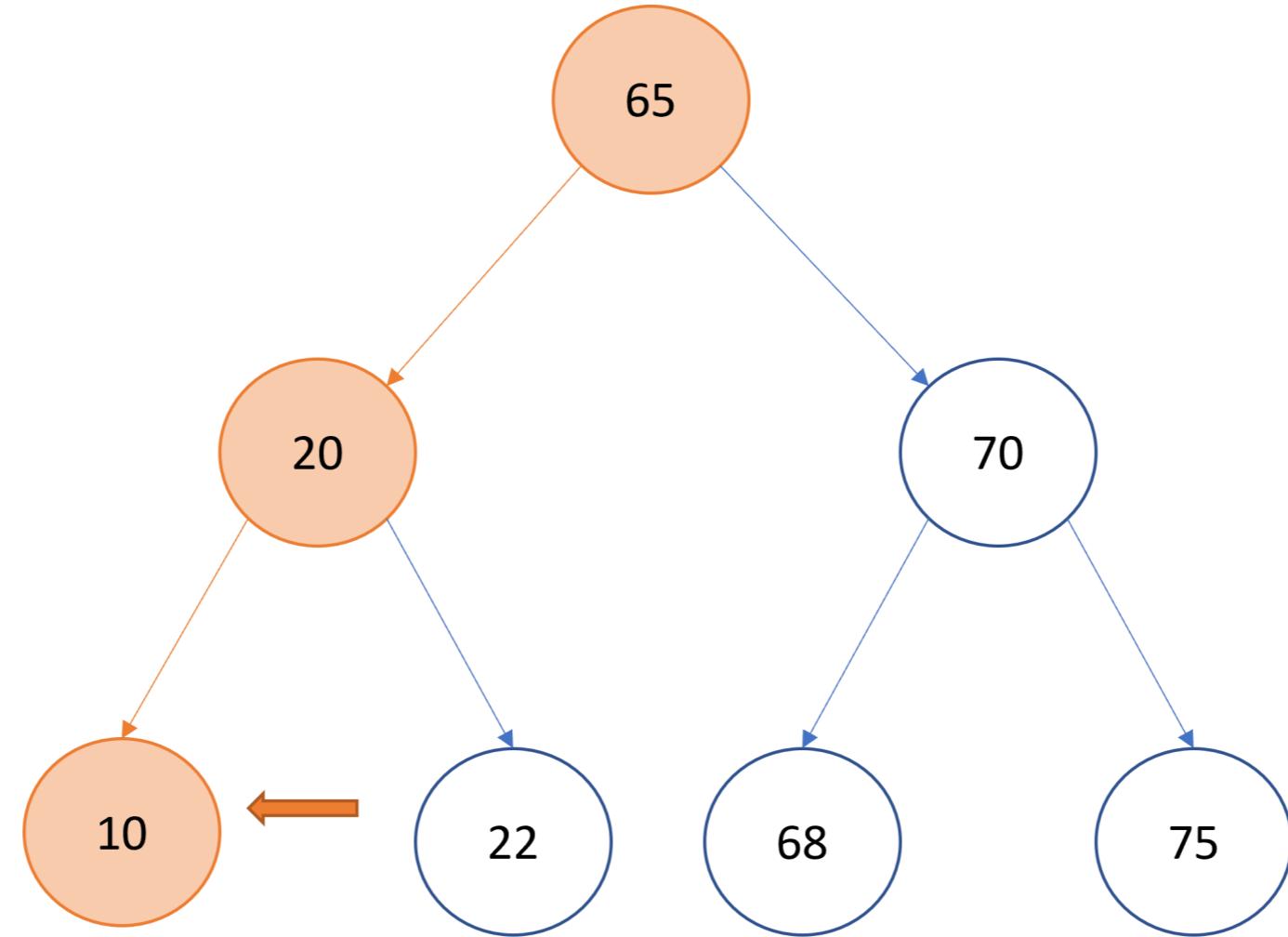
- Order: Current -> Left -> Right



Visited nodes: 65, 20

Pre-order traversal

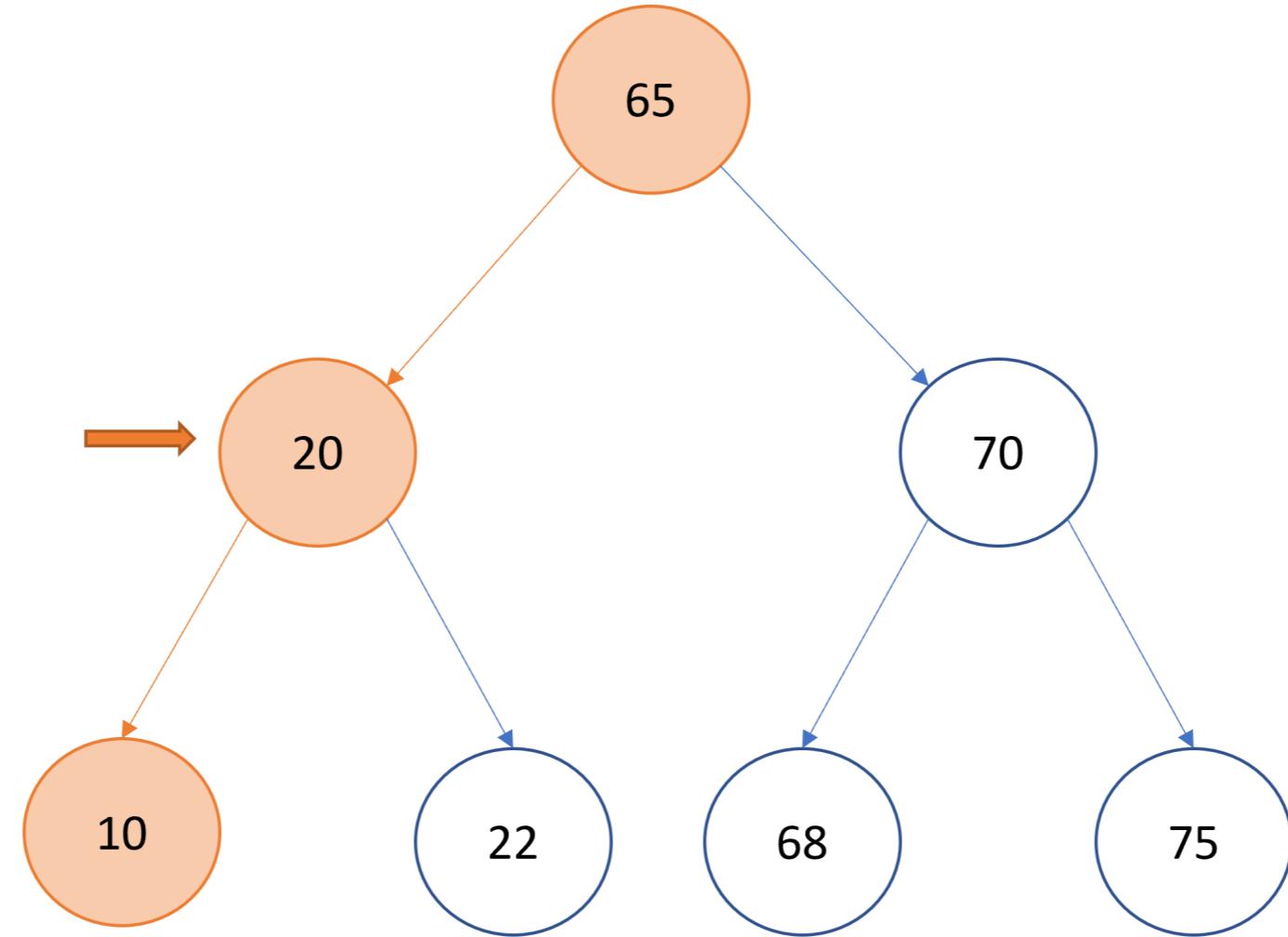
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10

Pre-order traversal

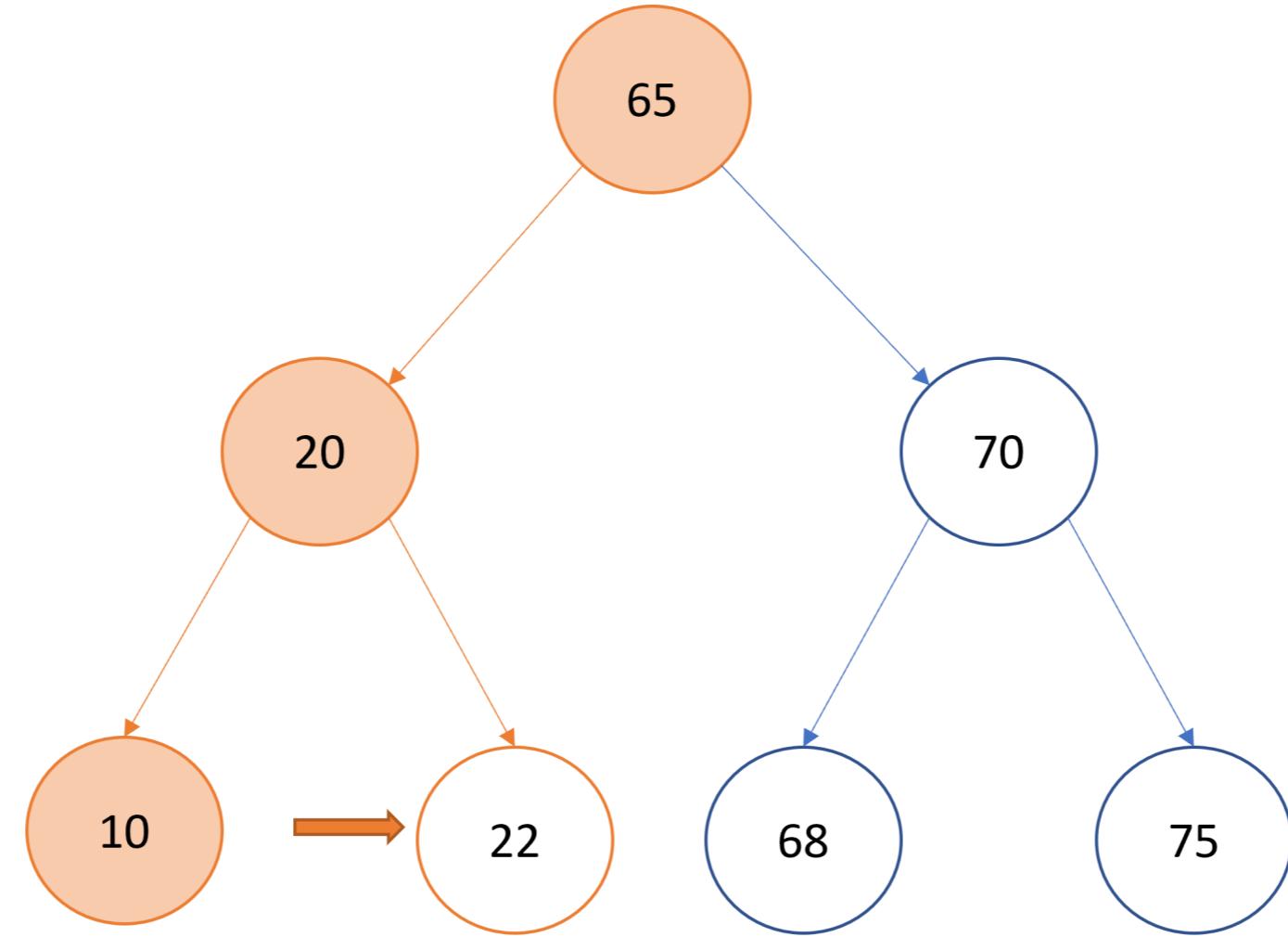
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10

Pre-order traversal

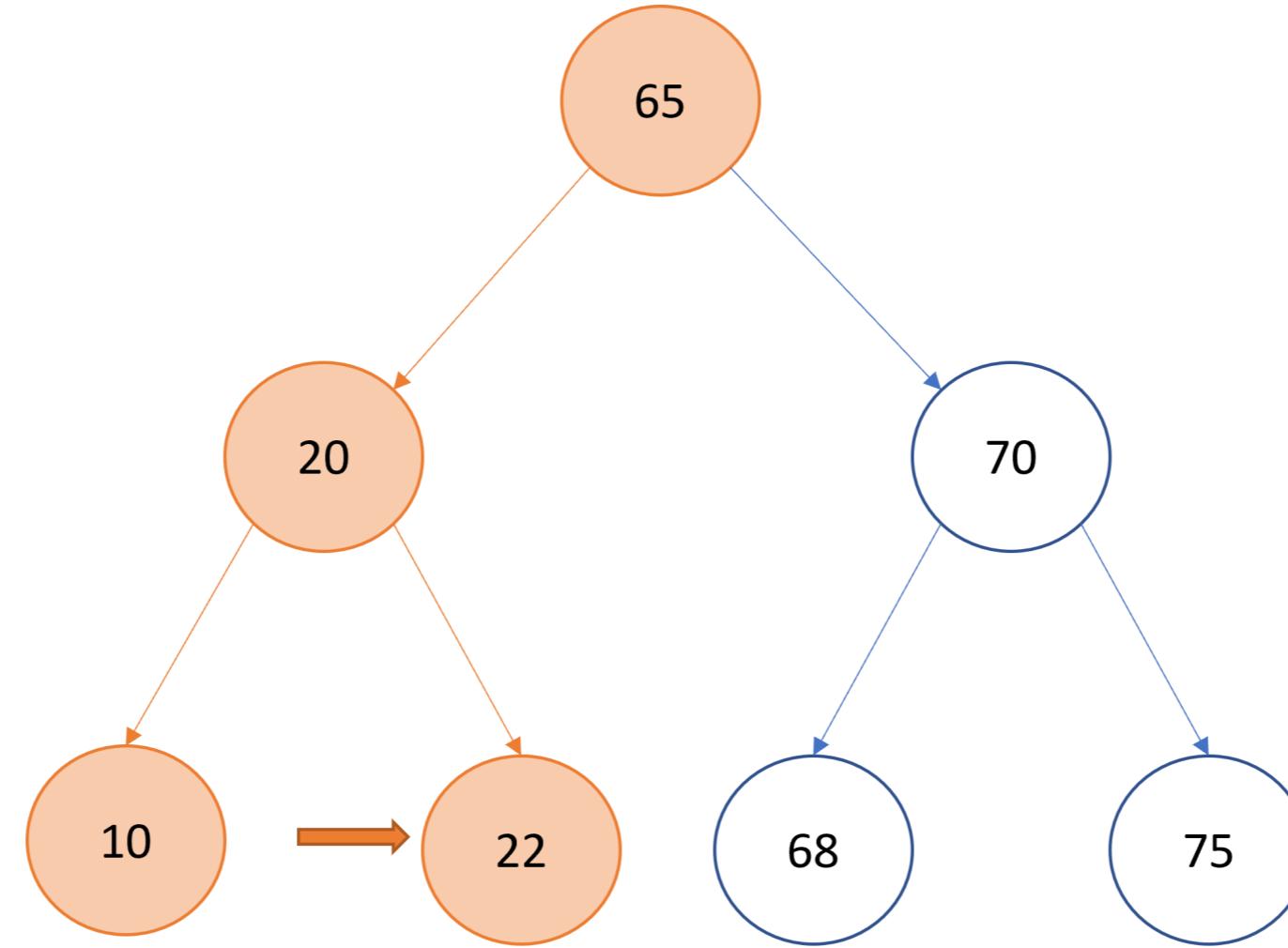
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10

Pre-order traversal

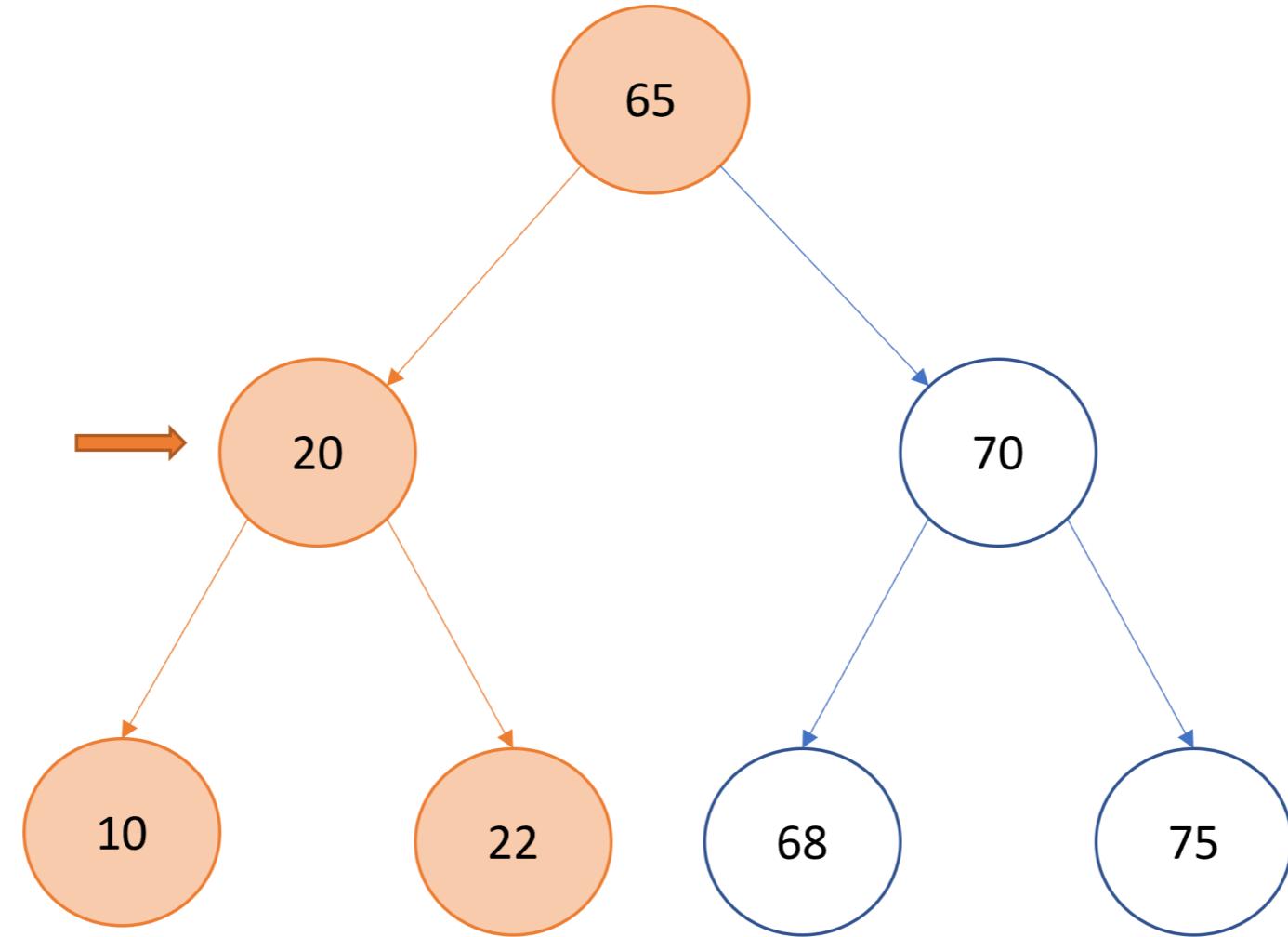
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22

Pre-order traversal

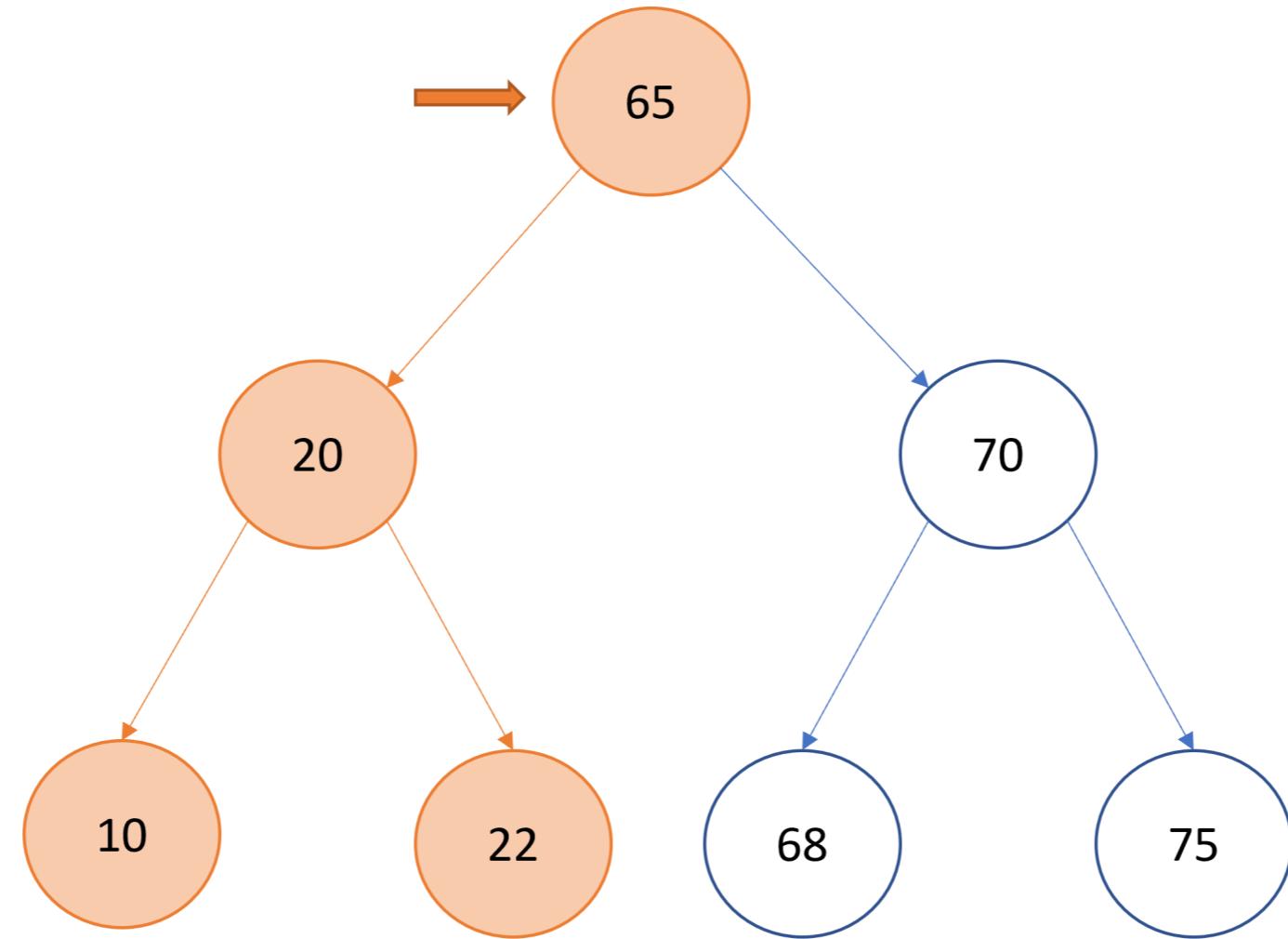
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22

Pre-order traversal

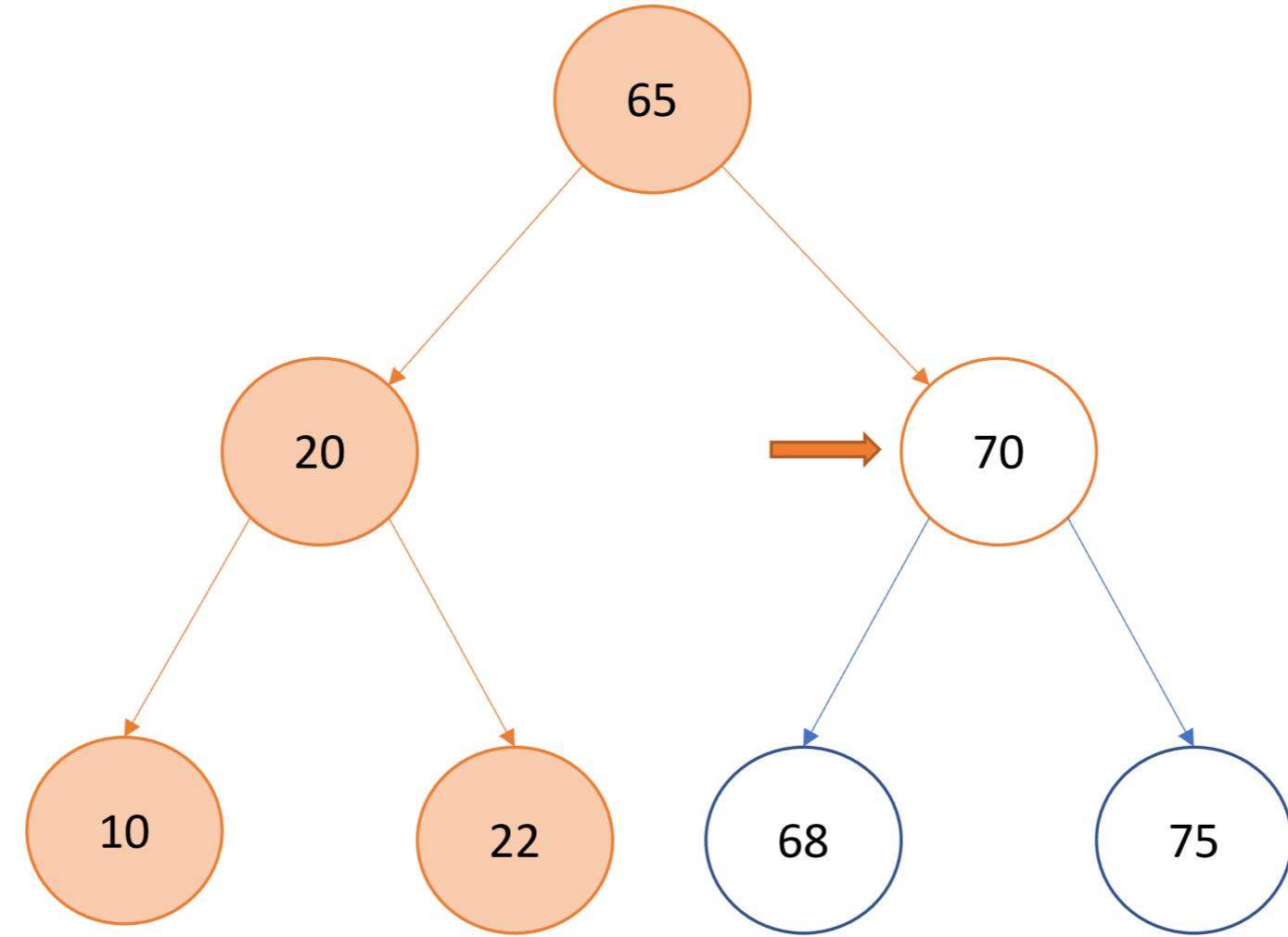
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22

Pre-order traversal

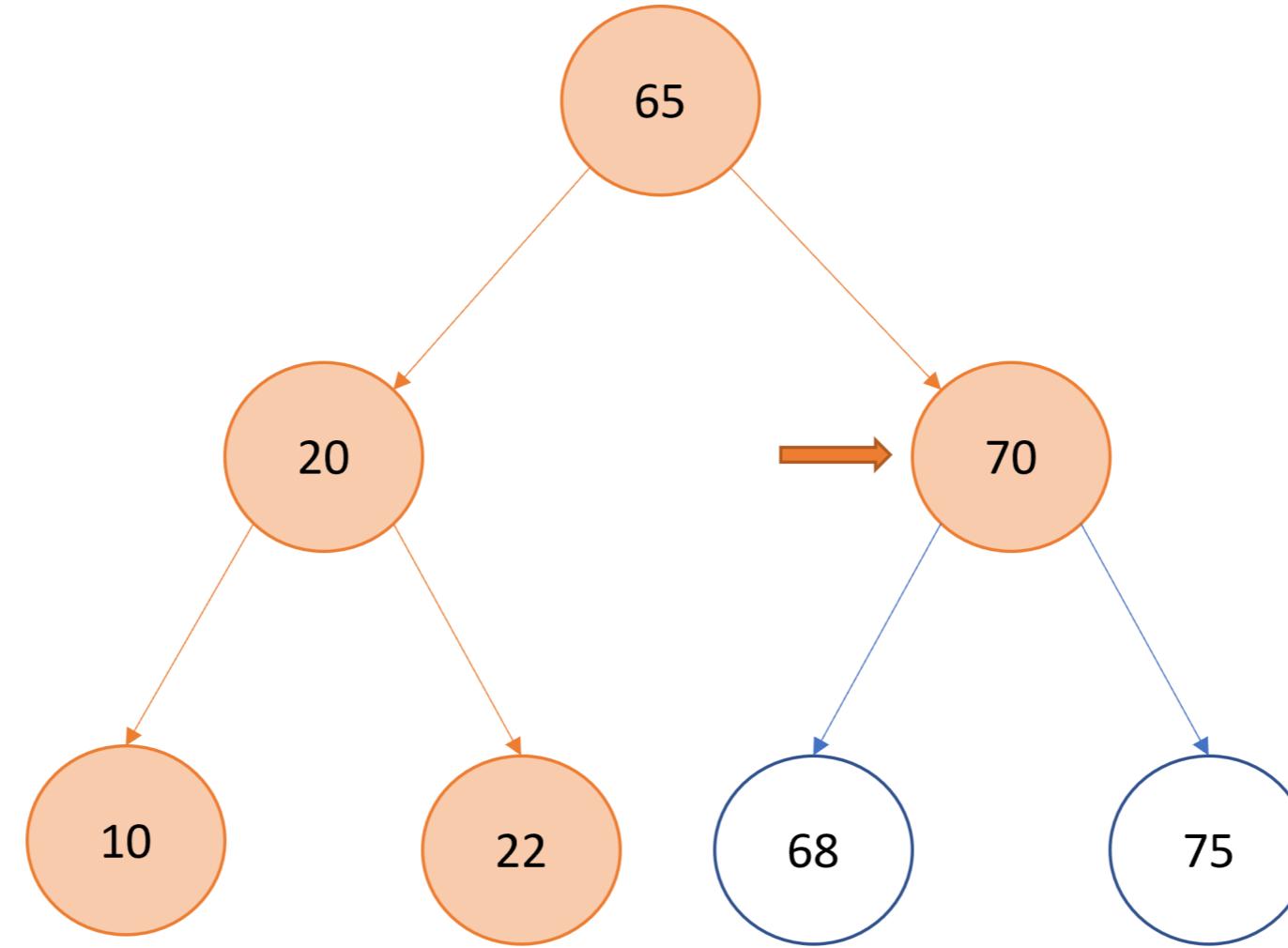
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22

Pre-order traversal

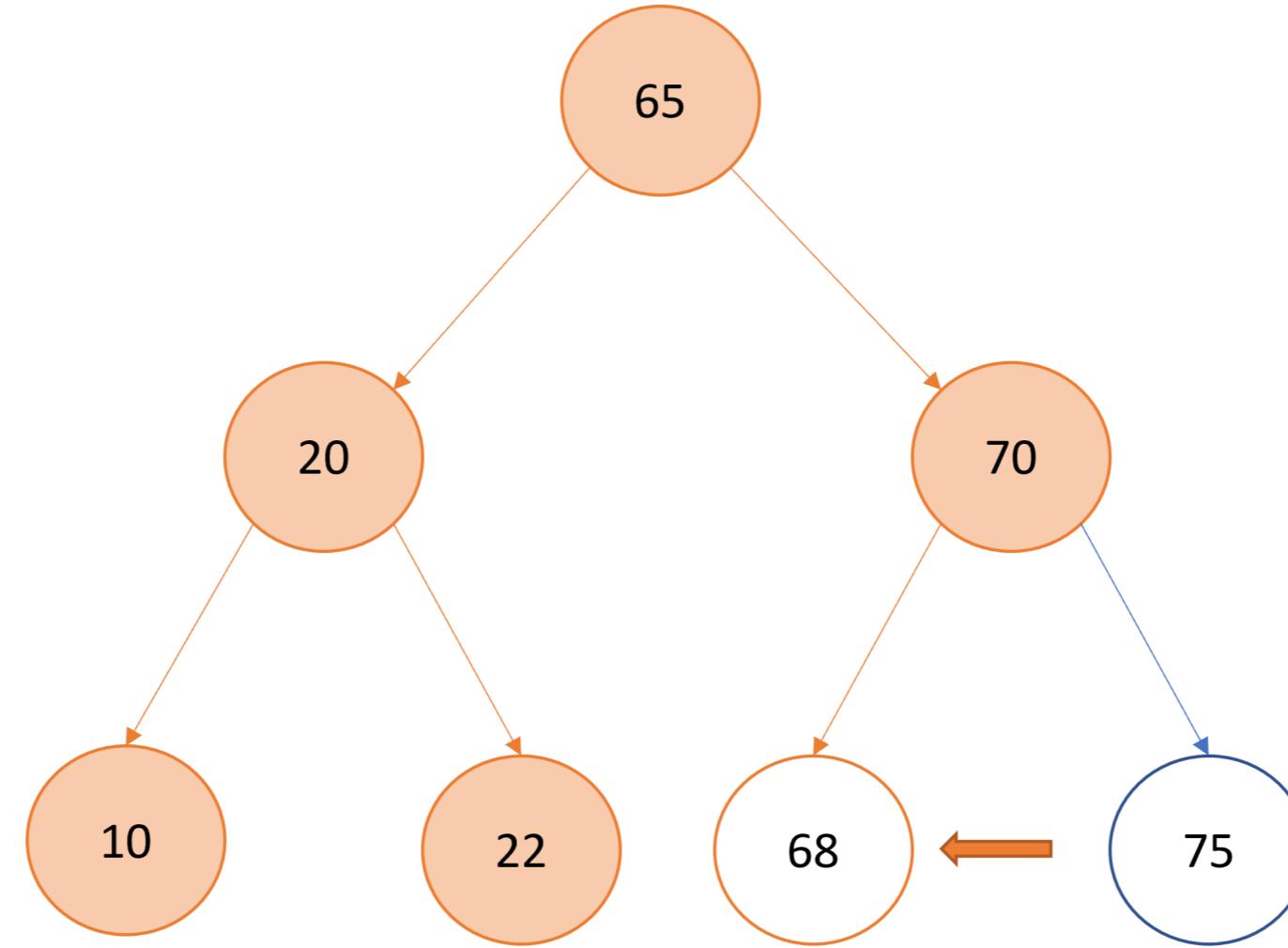
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22, 70

Pre-order traversal

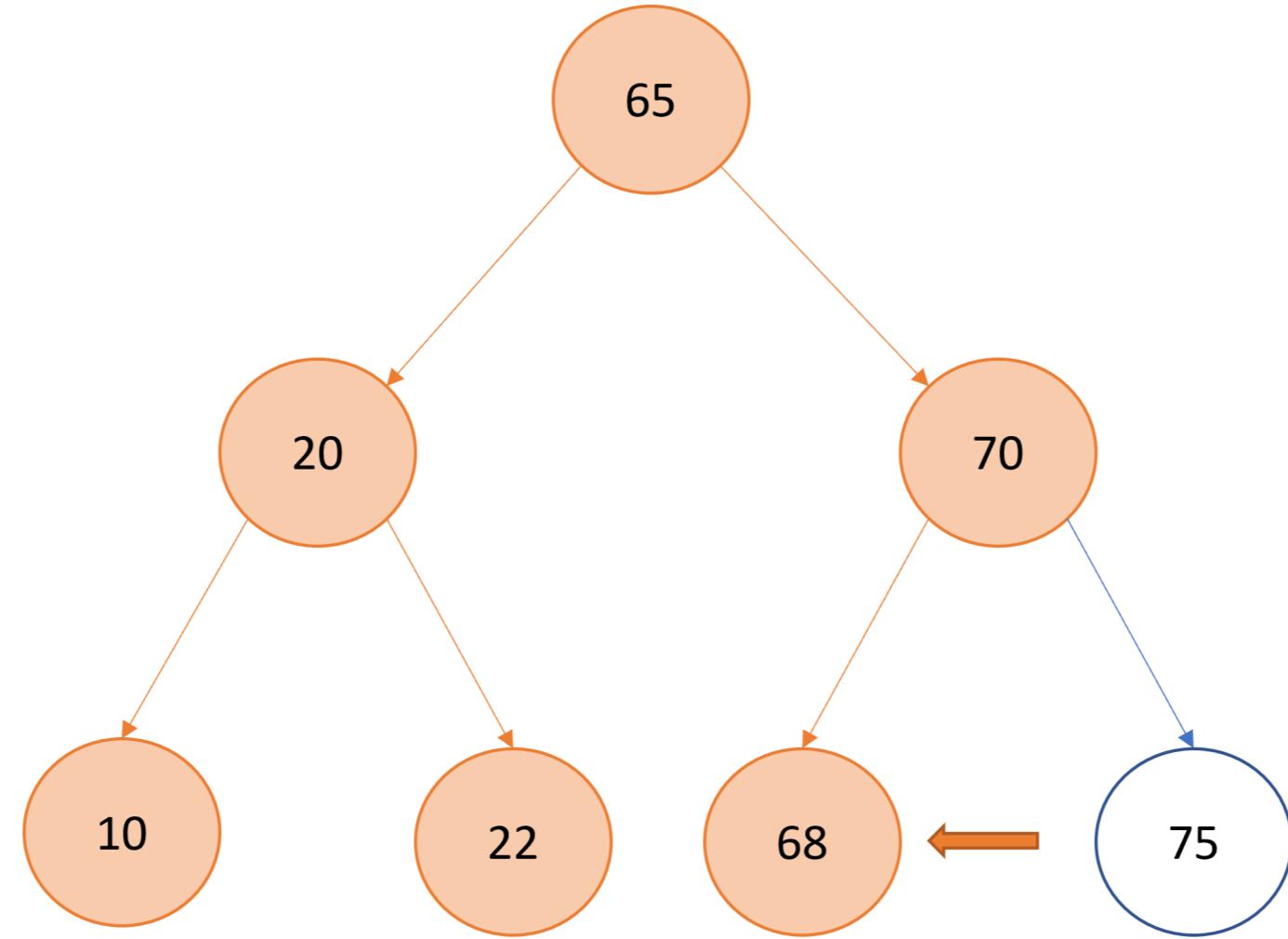
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22, 70

Pre-order traversal

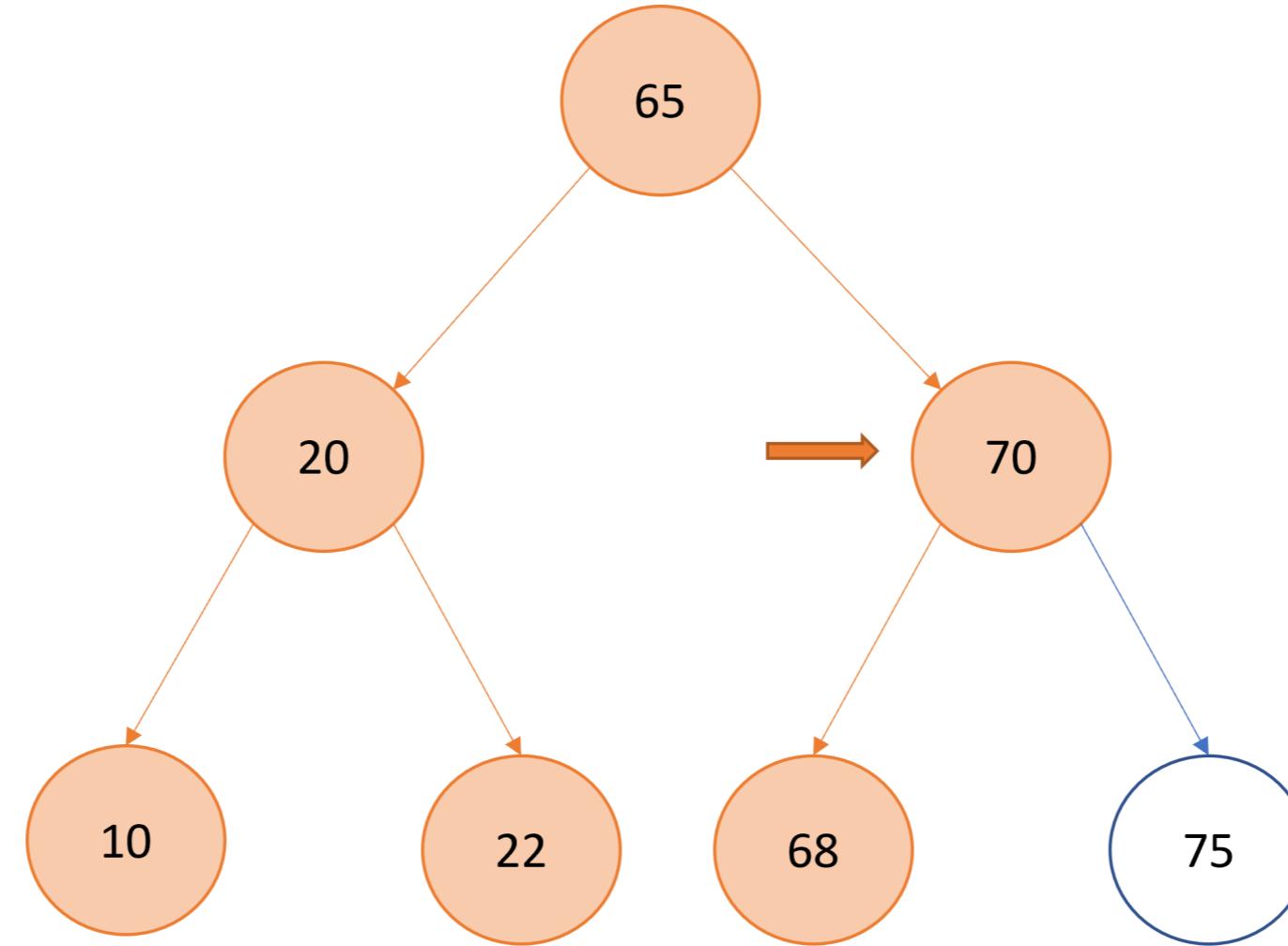
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22, 70, 68

Pre-order traversal

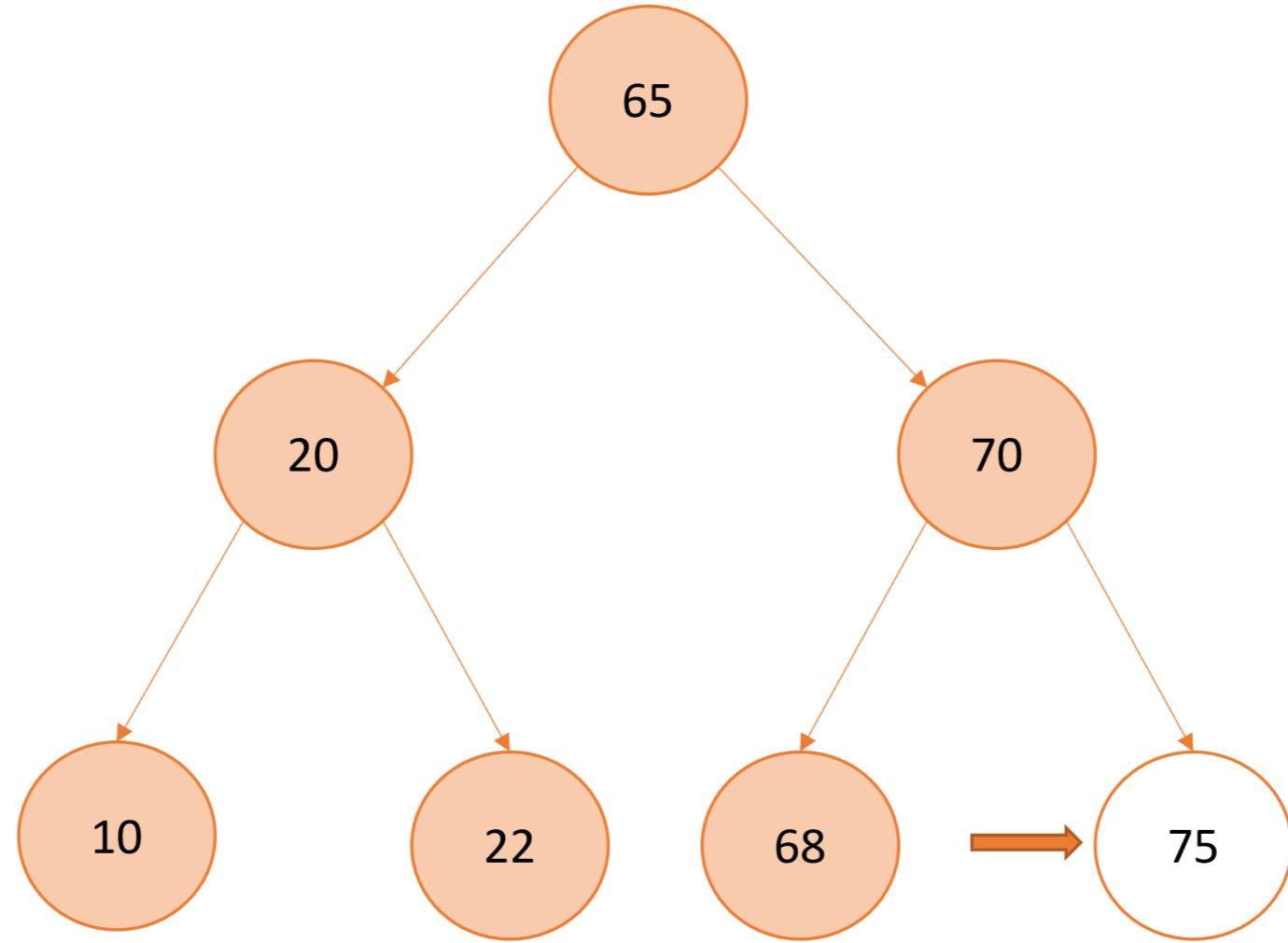
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22, 70, 68

Pre-order traversal

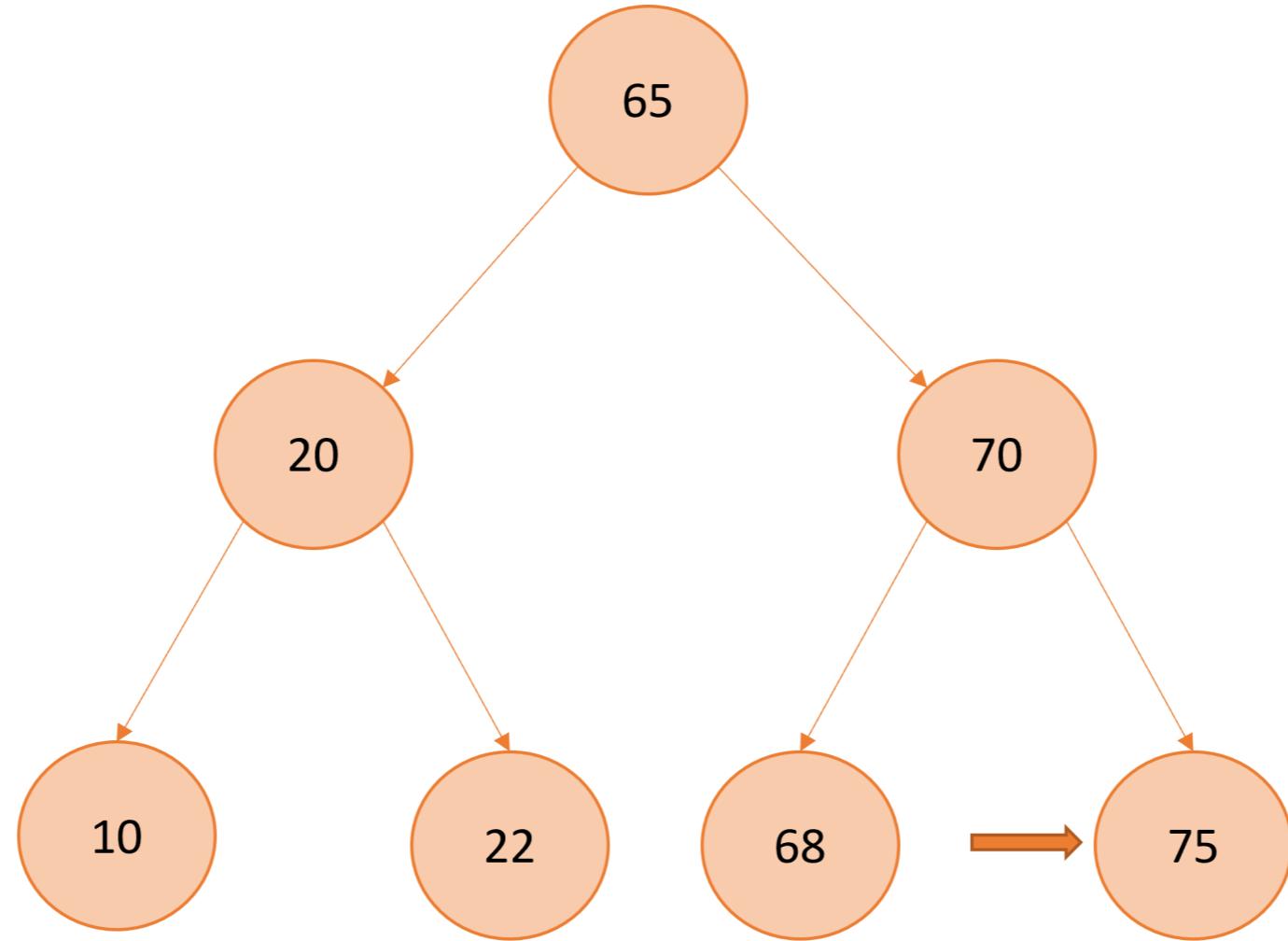
- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22, 70, 68

Pre-order traversal

- Order: Current -> Left -> Right



Visited nodes: 65, 20, 10, 22, 70, 68, 75

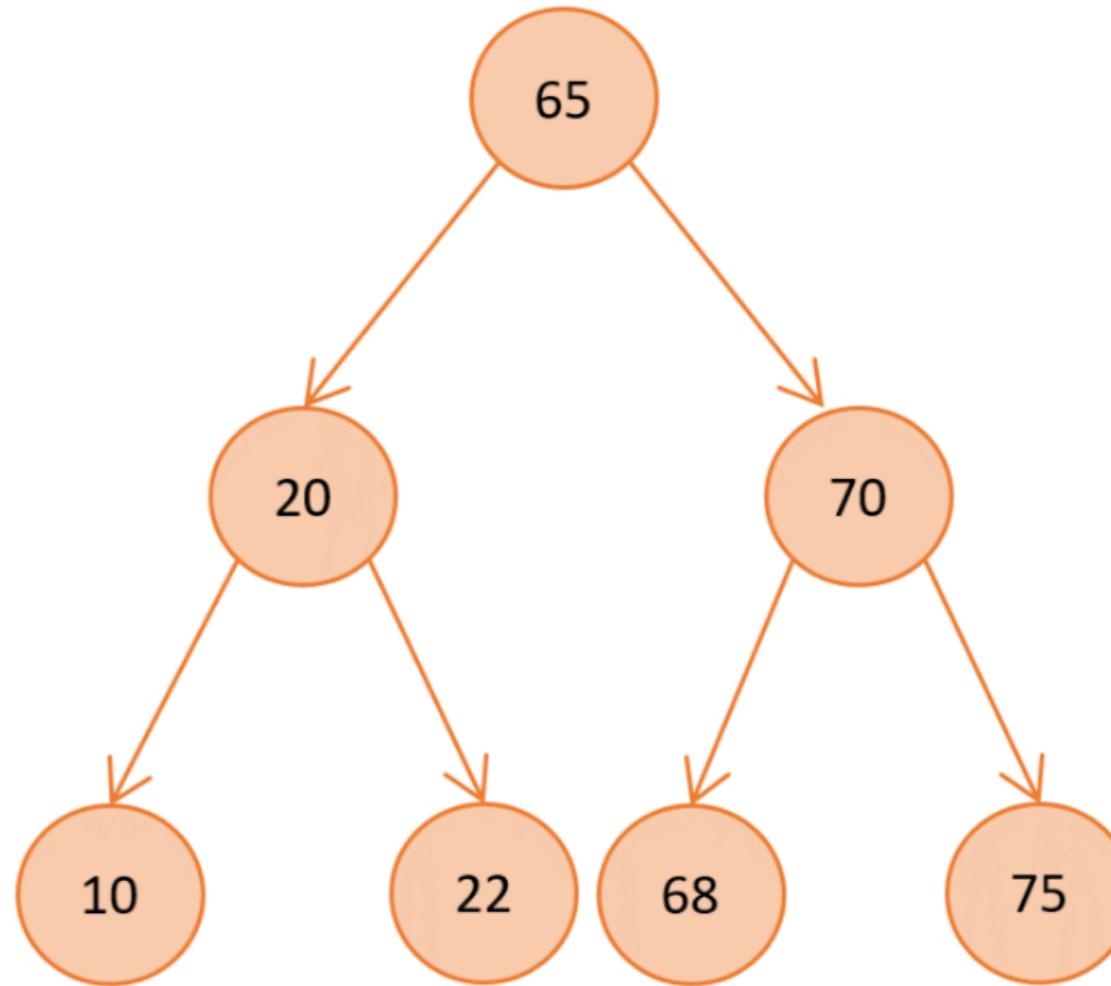
Pre-order traversal - implementation

- Order: Root -> Left -> Right

```
def pre_order(self, current_node):  
    if current_node:  
        print(current_node.data)  
        self.pre_order(current_node.left_child)  
        self.pre_order(current_node.right_child)
```

```
my_tree.pre_order(my_tree.root)
```

```
65  
20  
10  
22  
70  
68  
75
```



- Complexity: $O(n)$
 - $n \rightarrow$ number of nodes

Post-order

- Order: Left

Post-order

- Order: Left -> Right

Post-order

- **Order:** Left -> Right -> Current
- **Complexity:** $O(n)$
 - $n \rightarrow$ number of nodes

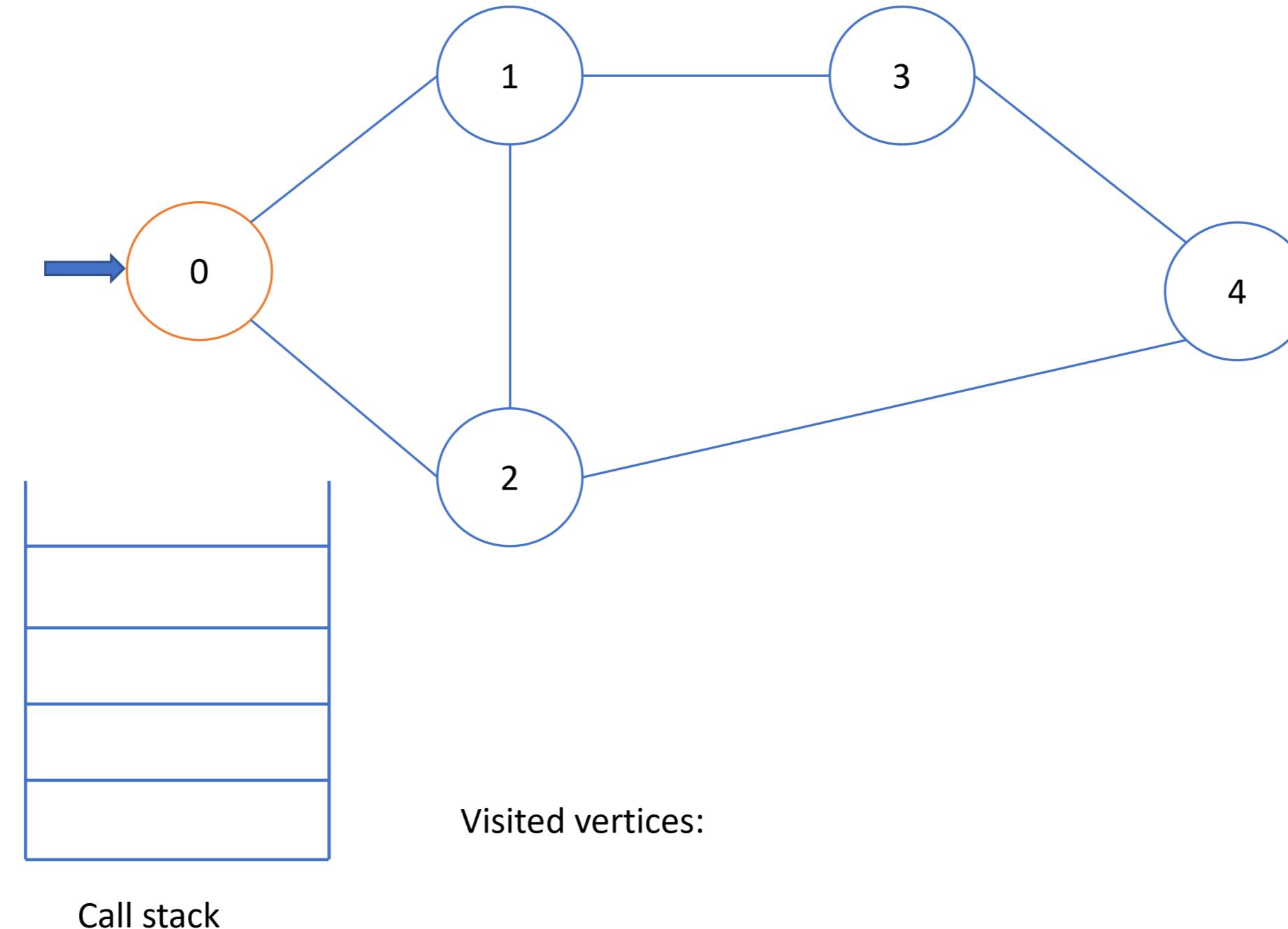
When to use in-order, pre-order, and post-order

- **in-order**
 - used BST to obtain the node's values in ascending order
- **pre-order**
 - create copies of a tree
 - get prefix expressions
- **post-order**
 - delete binary trees
 - get postfix expressions

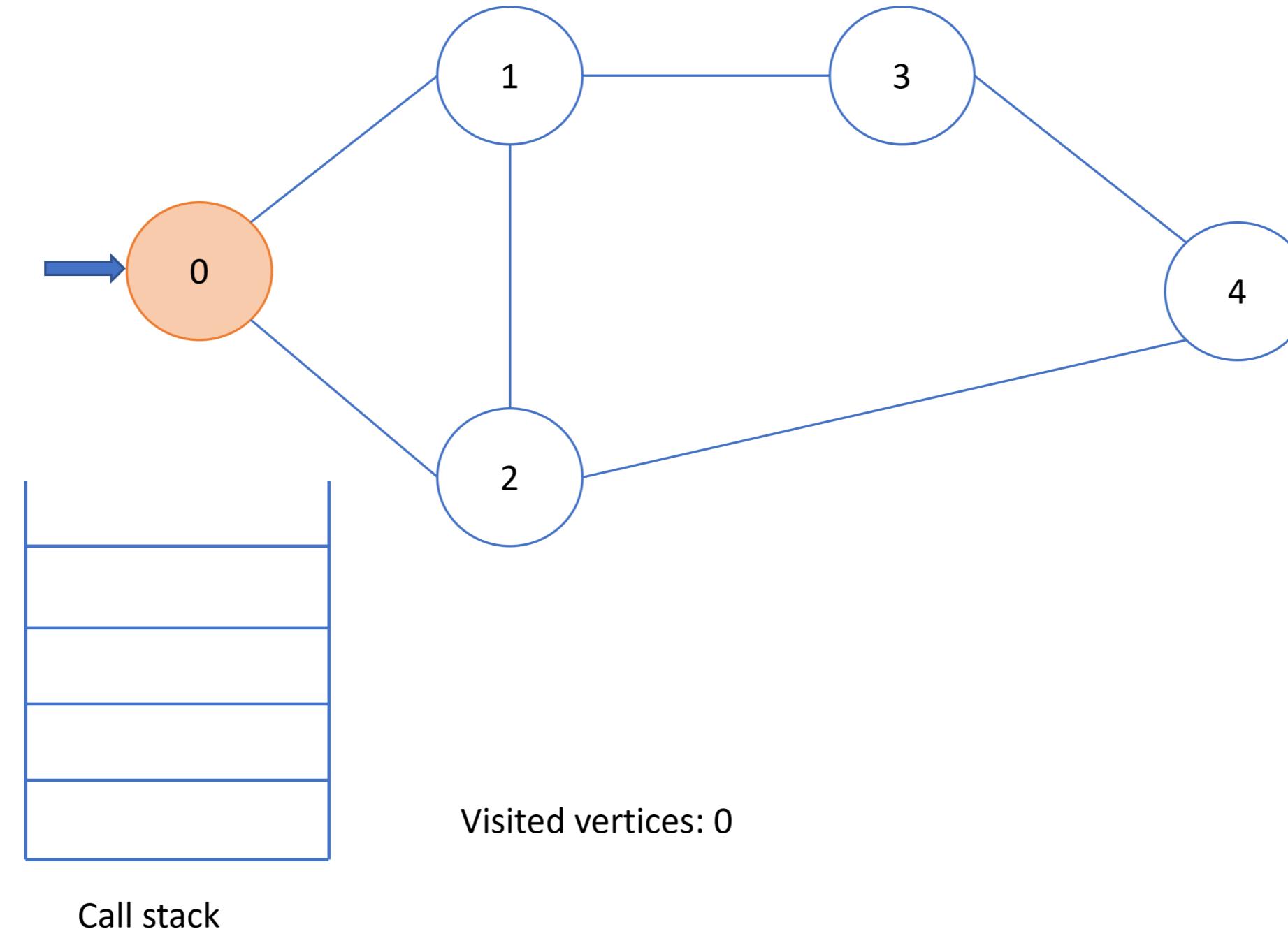
Depth first search - graphs

- Graphs can have cycles
 - need to keep track of visited vertices
- Steps:
 1. Start at any vertex
 2. Tracks current vertex to visited vertices list
 3. For each current node's adjacent vertex
 - If it has been visited -> ignore it
 - If it hasn't been visited -> recursively perform DFS

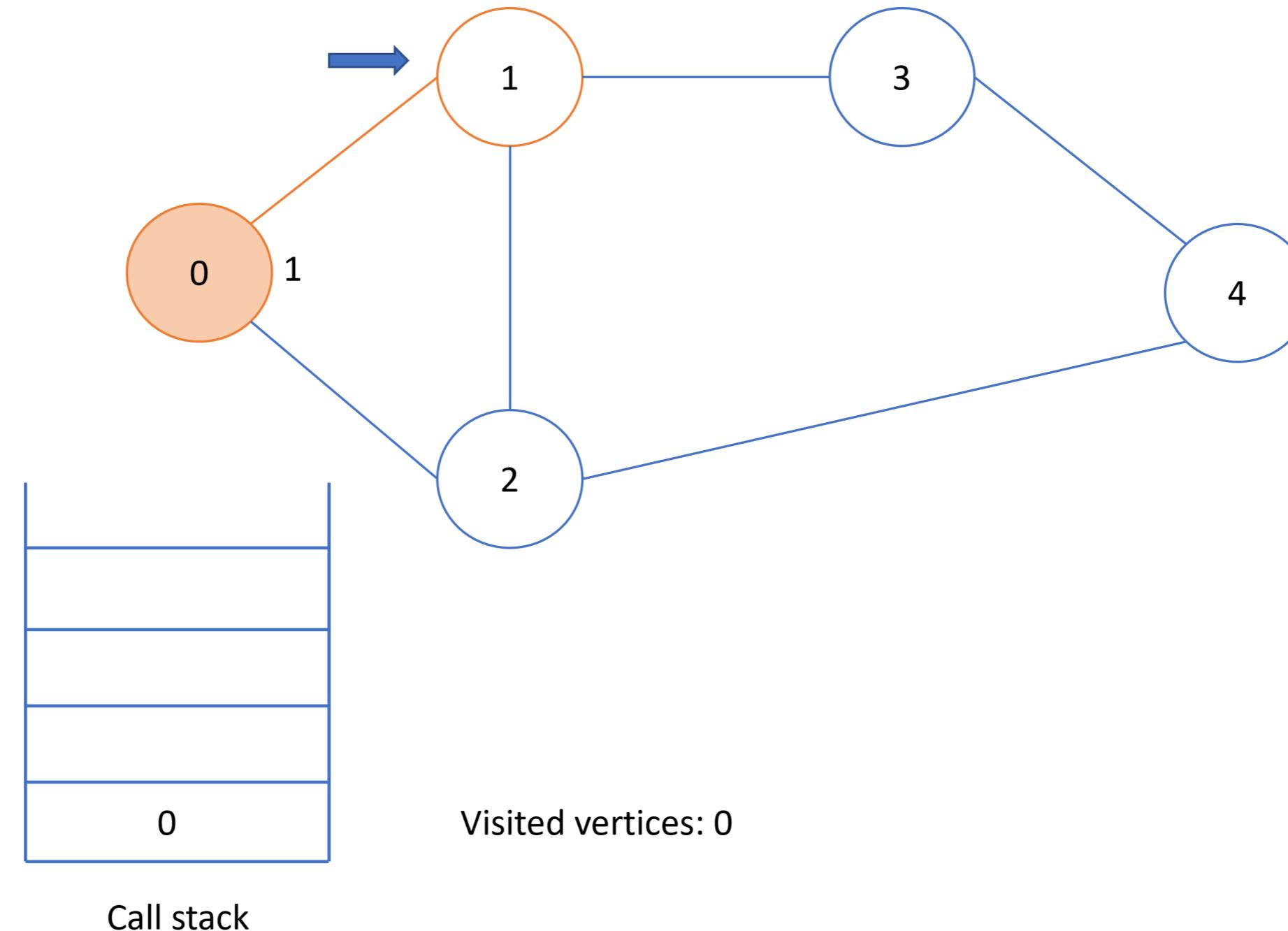
Depth first search - graphs



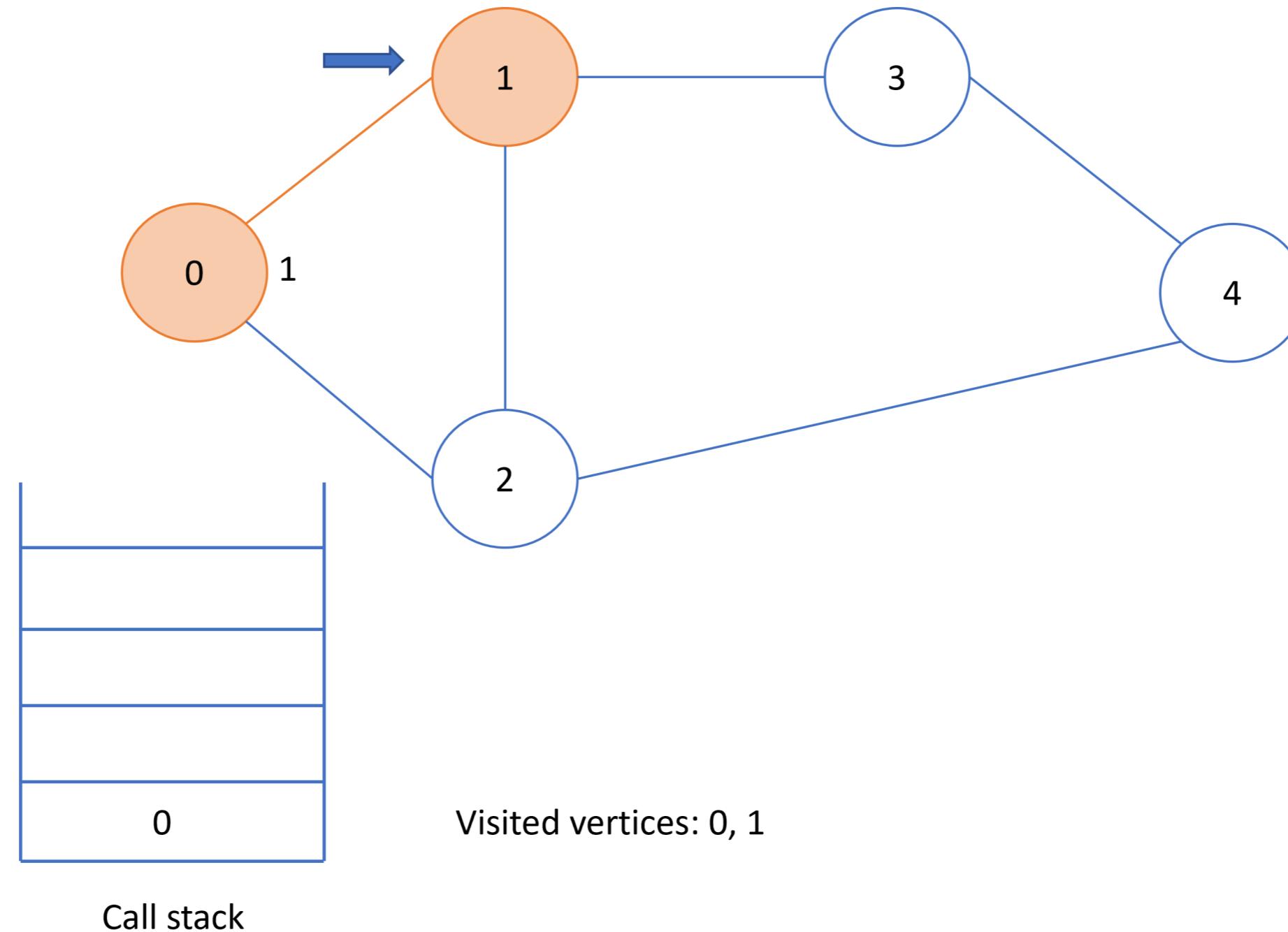
Depth first search - graphs



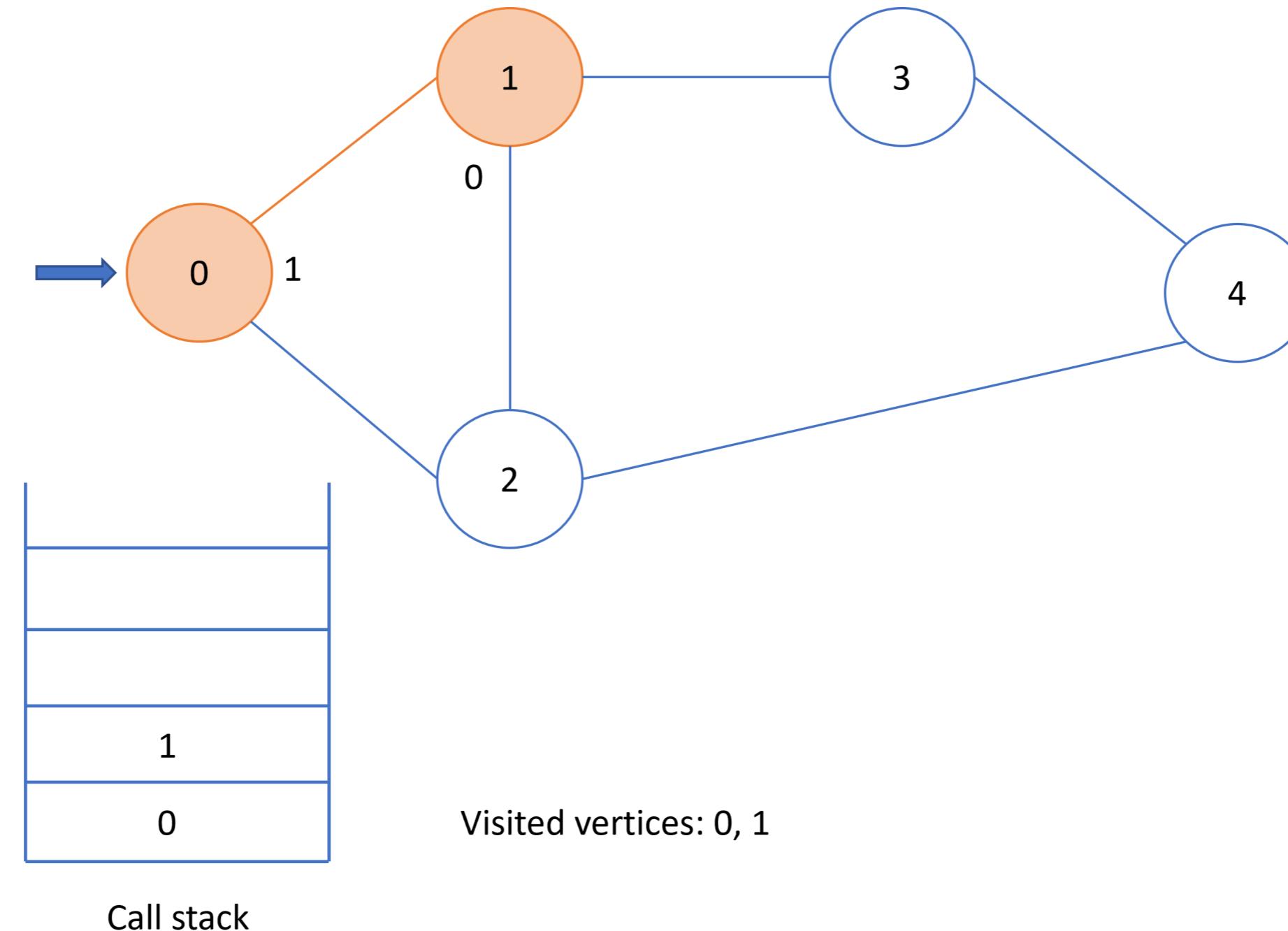
Depth first search - graphs



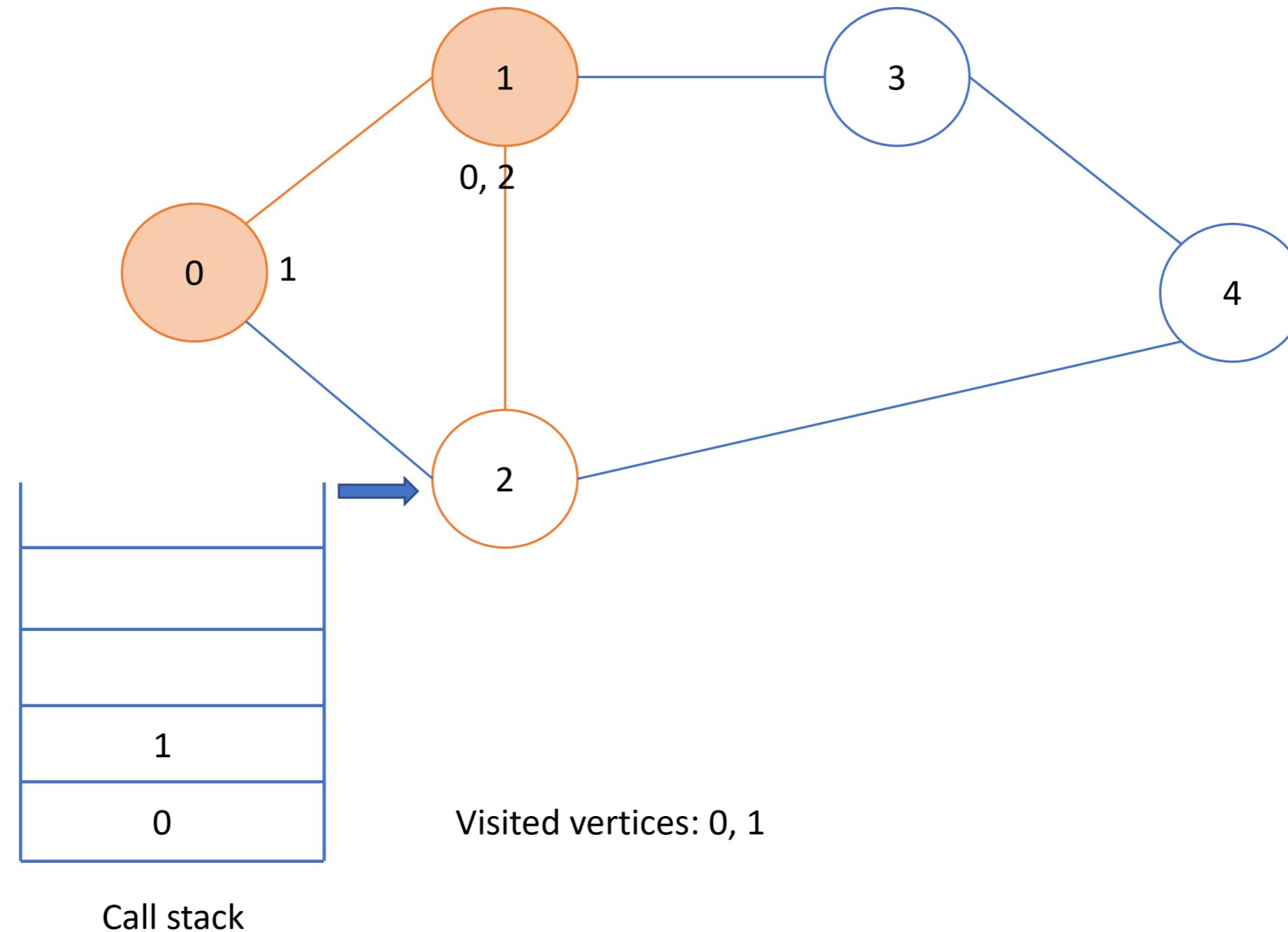
Depth first search - graphs



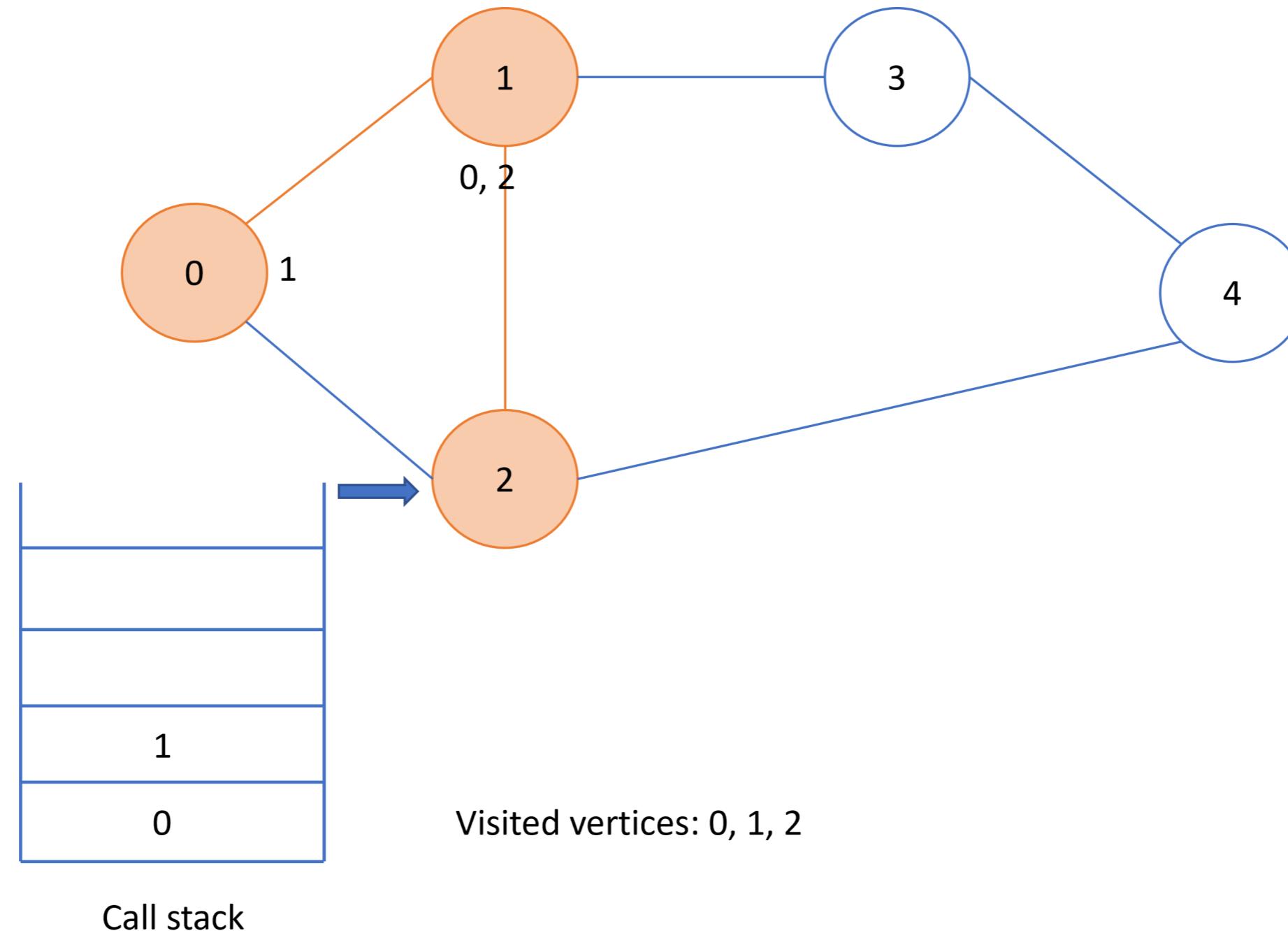
Depth first search - graphs



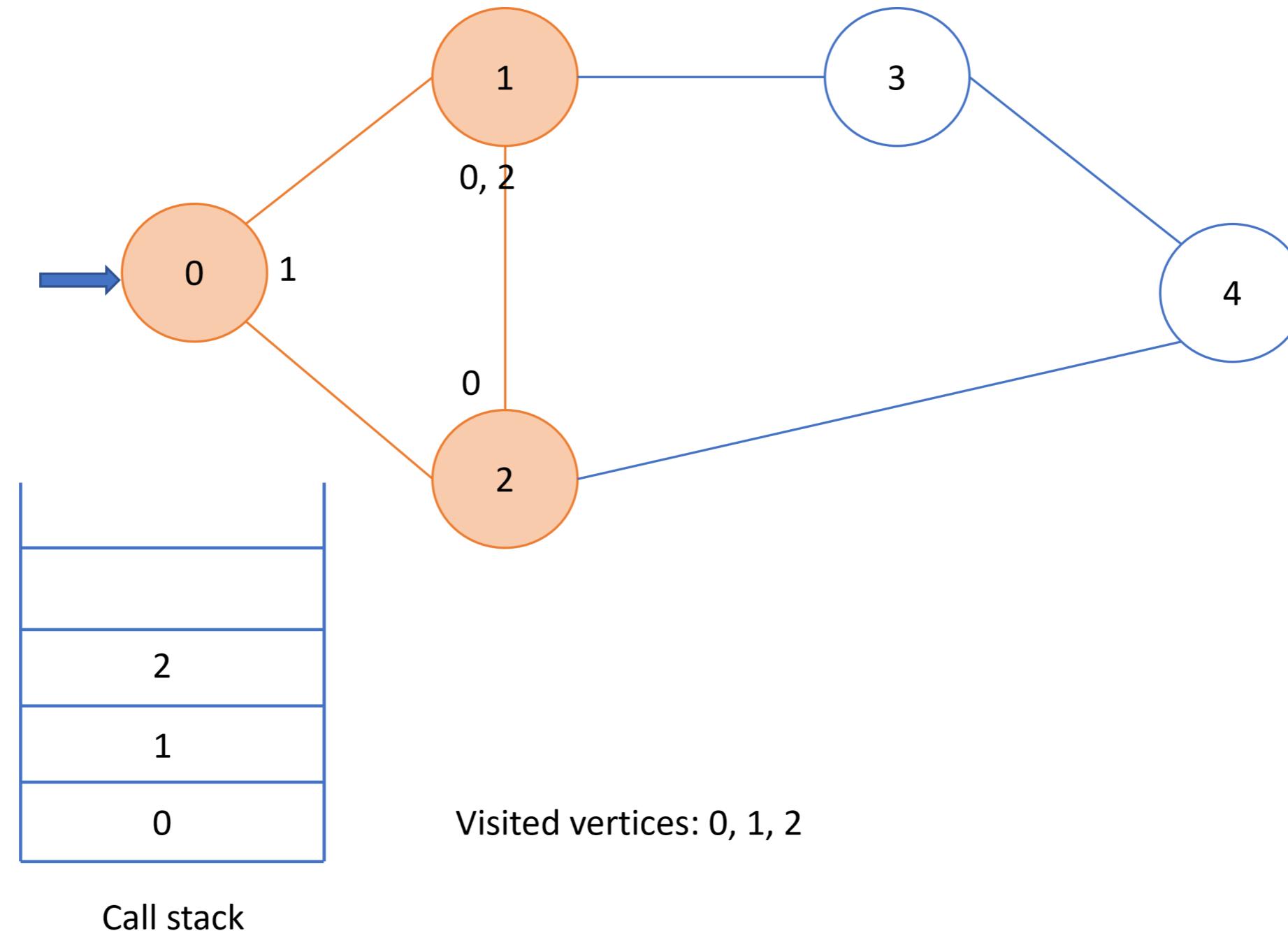
Depth first search - graphs



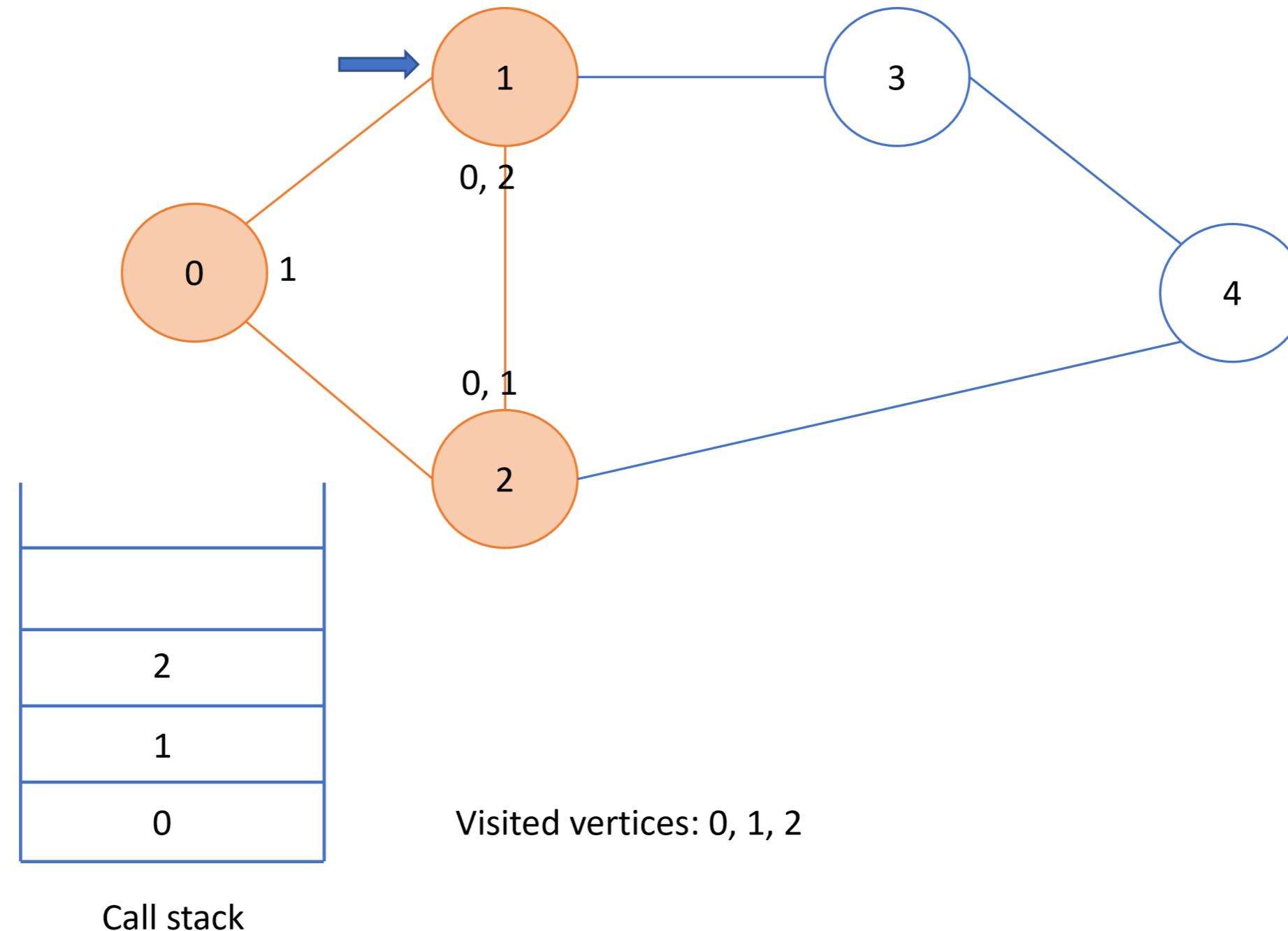
Depth first search - graphs



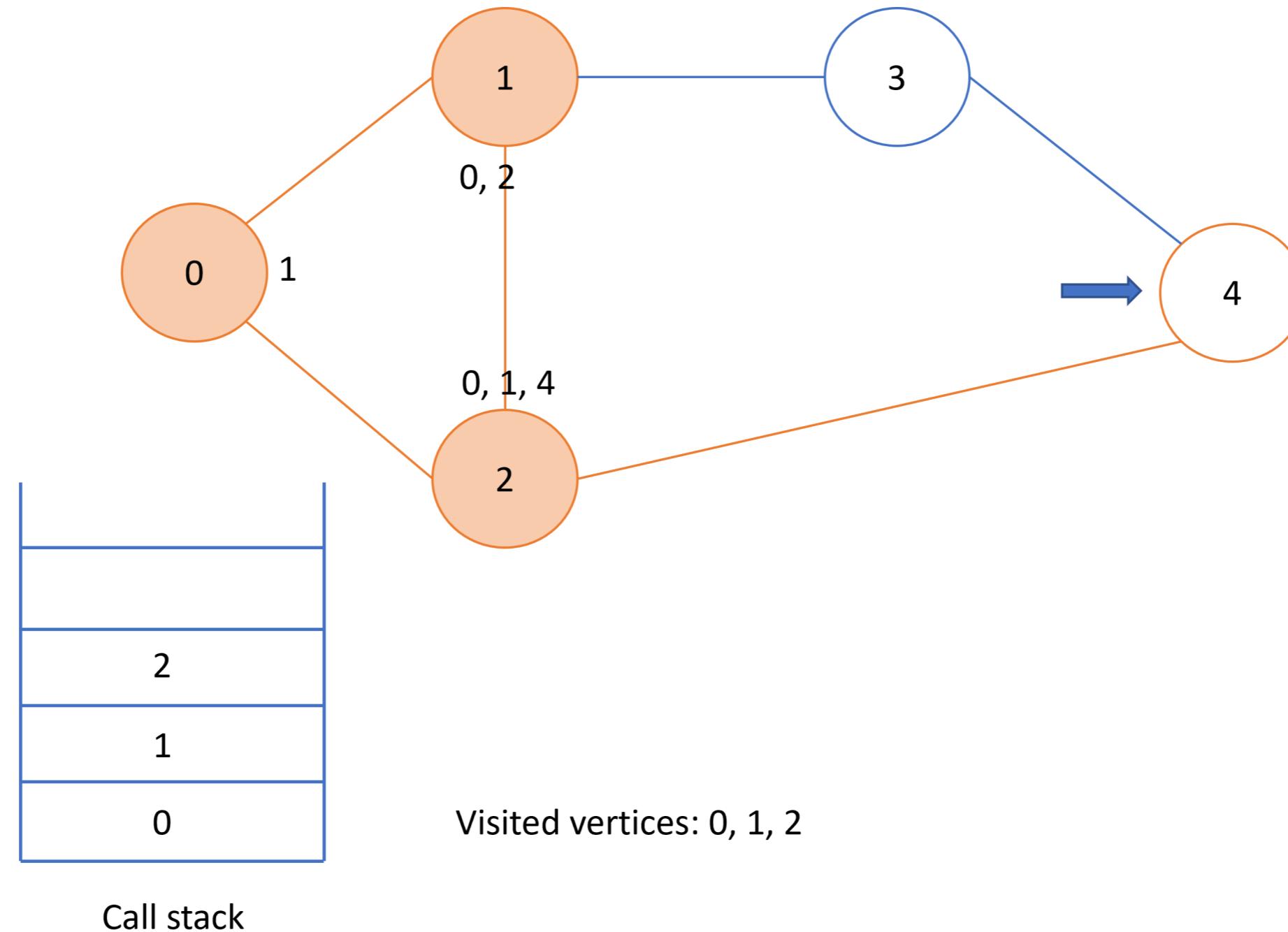
Depth first search - graphs



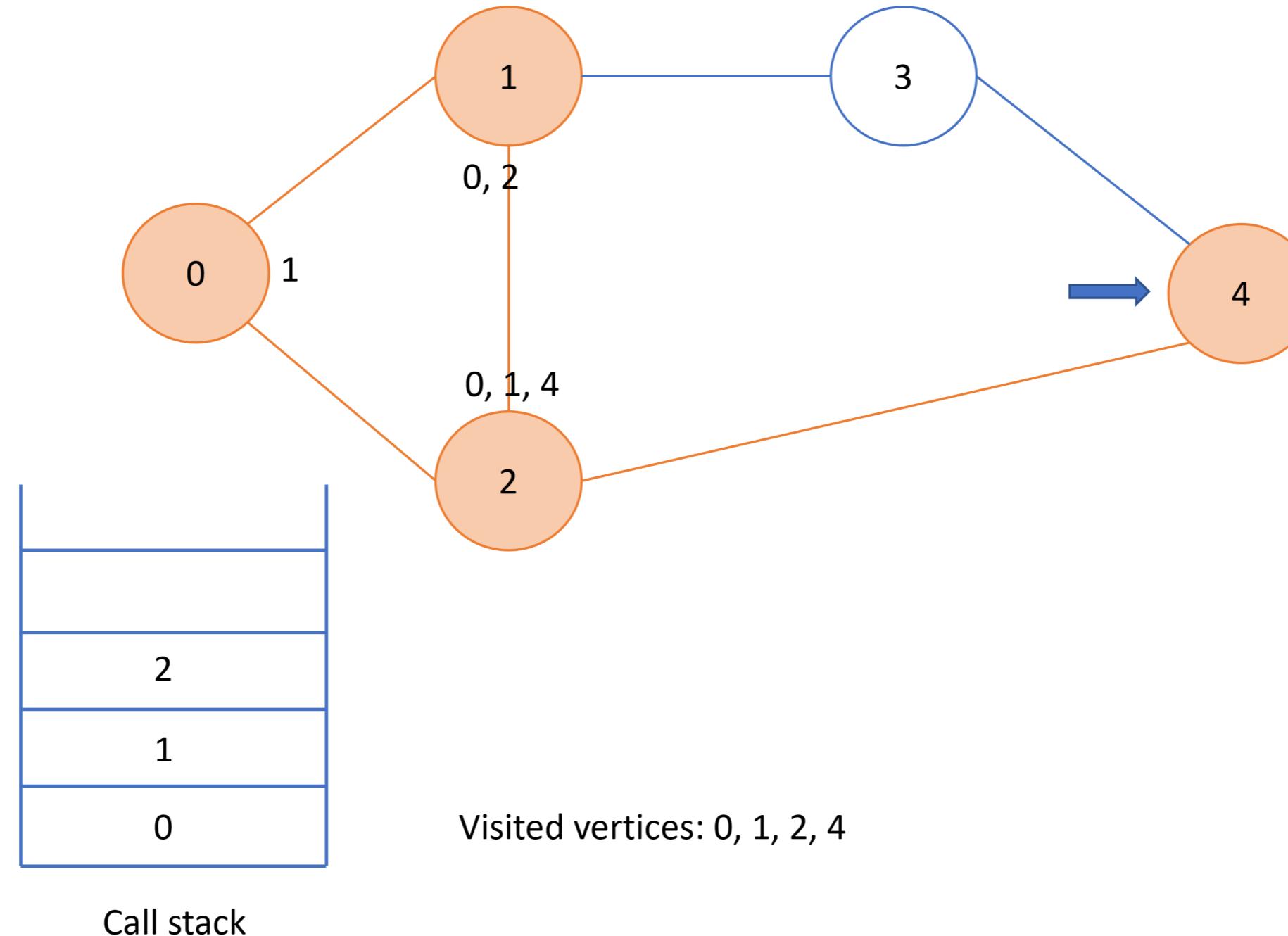
Depth first search - graphs



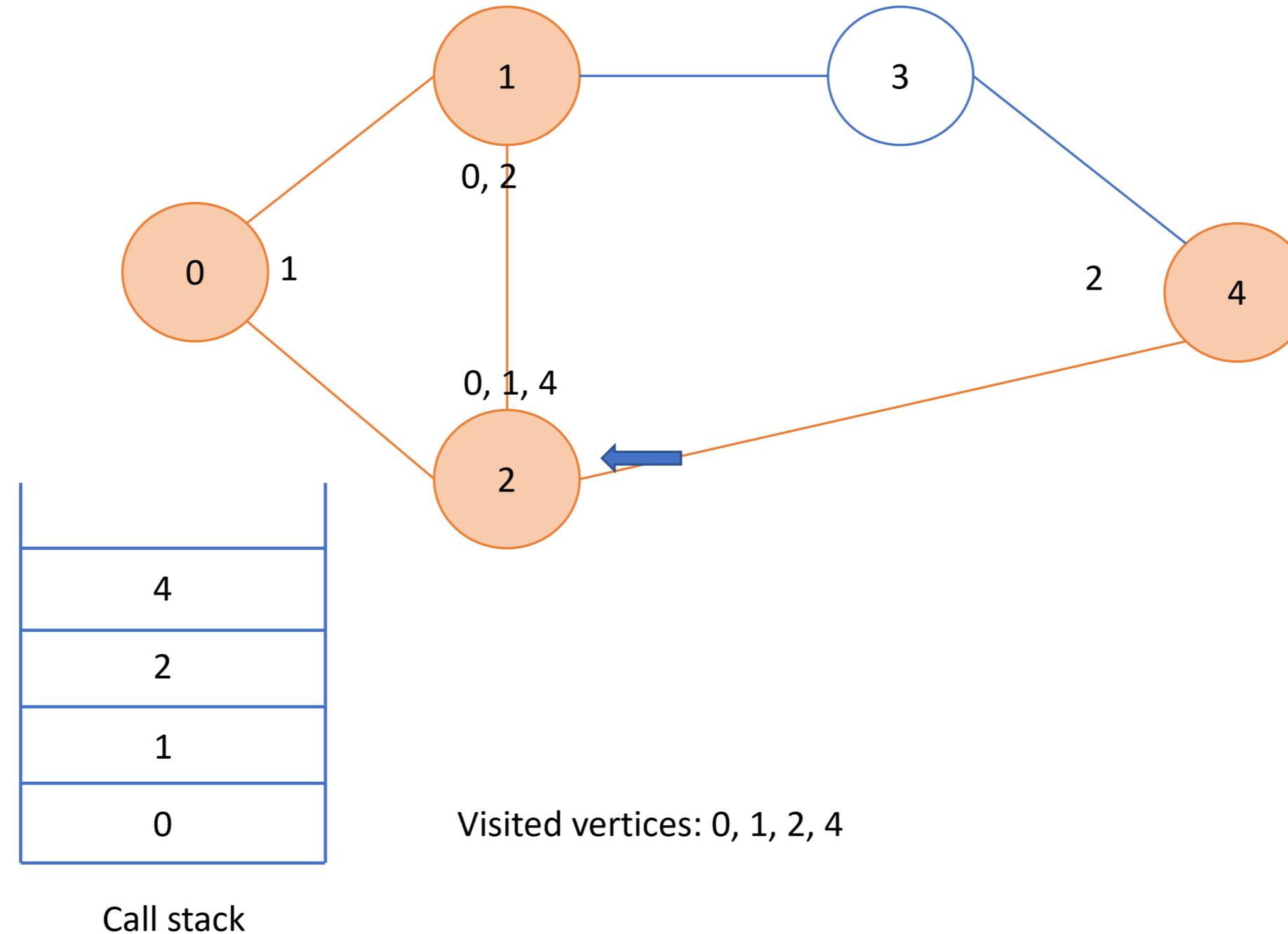
Depth first search - graphs



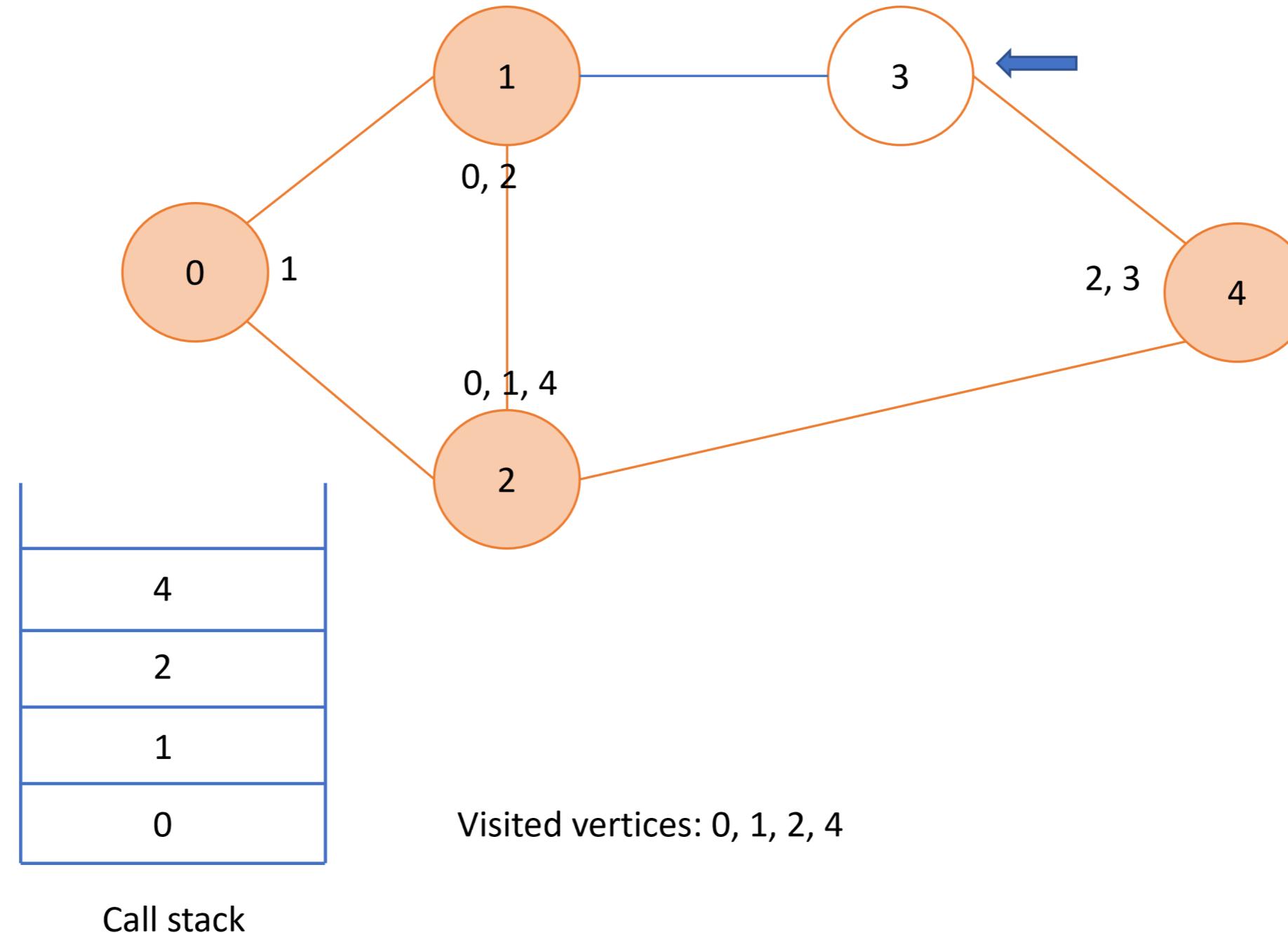
Depth first search - graphs



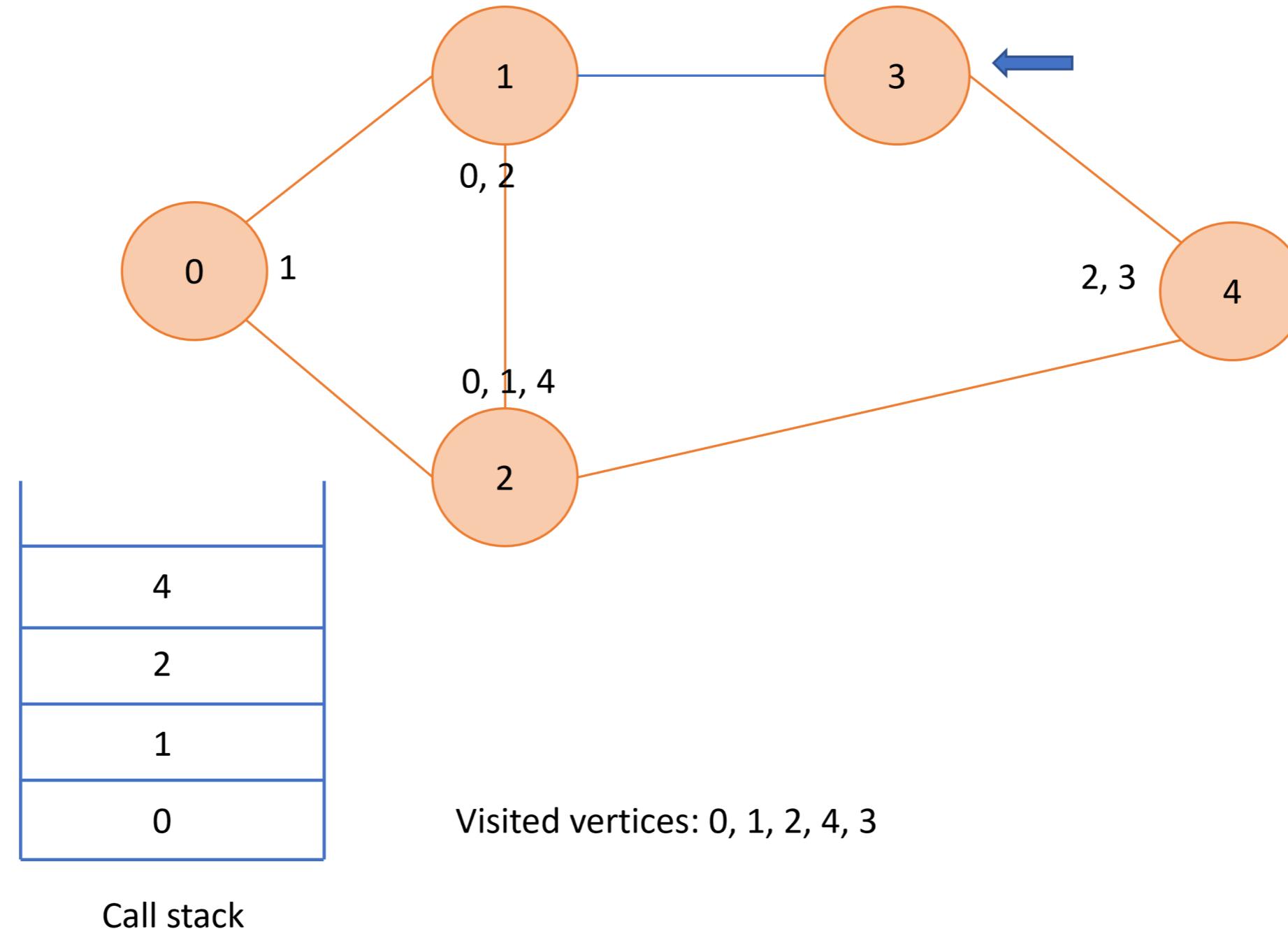
Depth first search - graphs



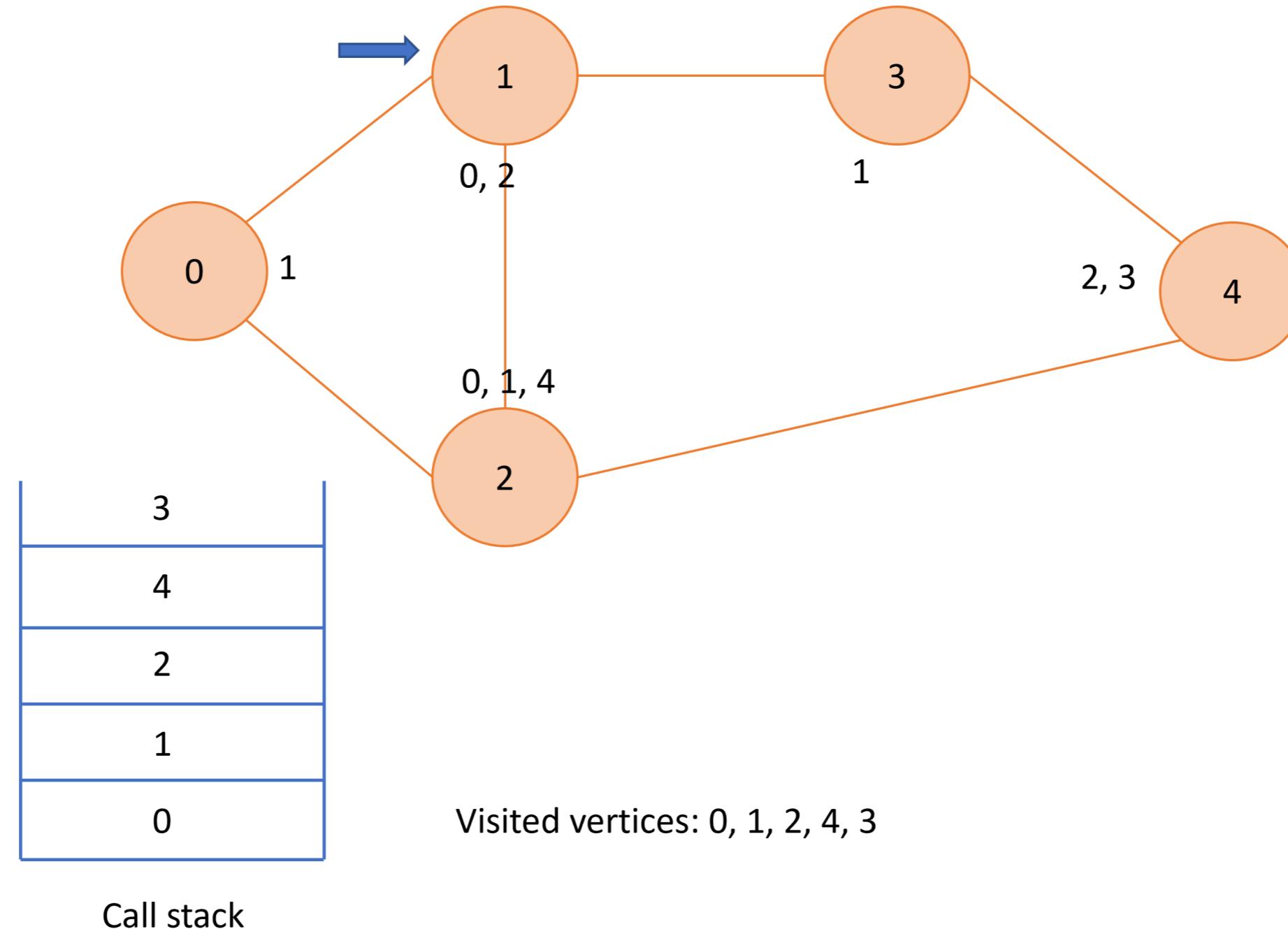
Depth first search - graphs



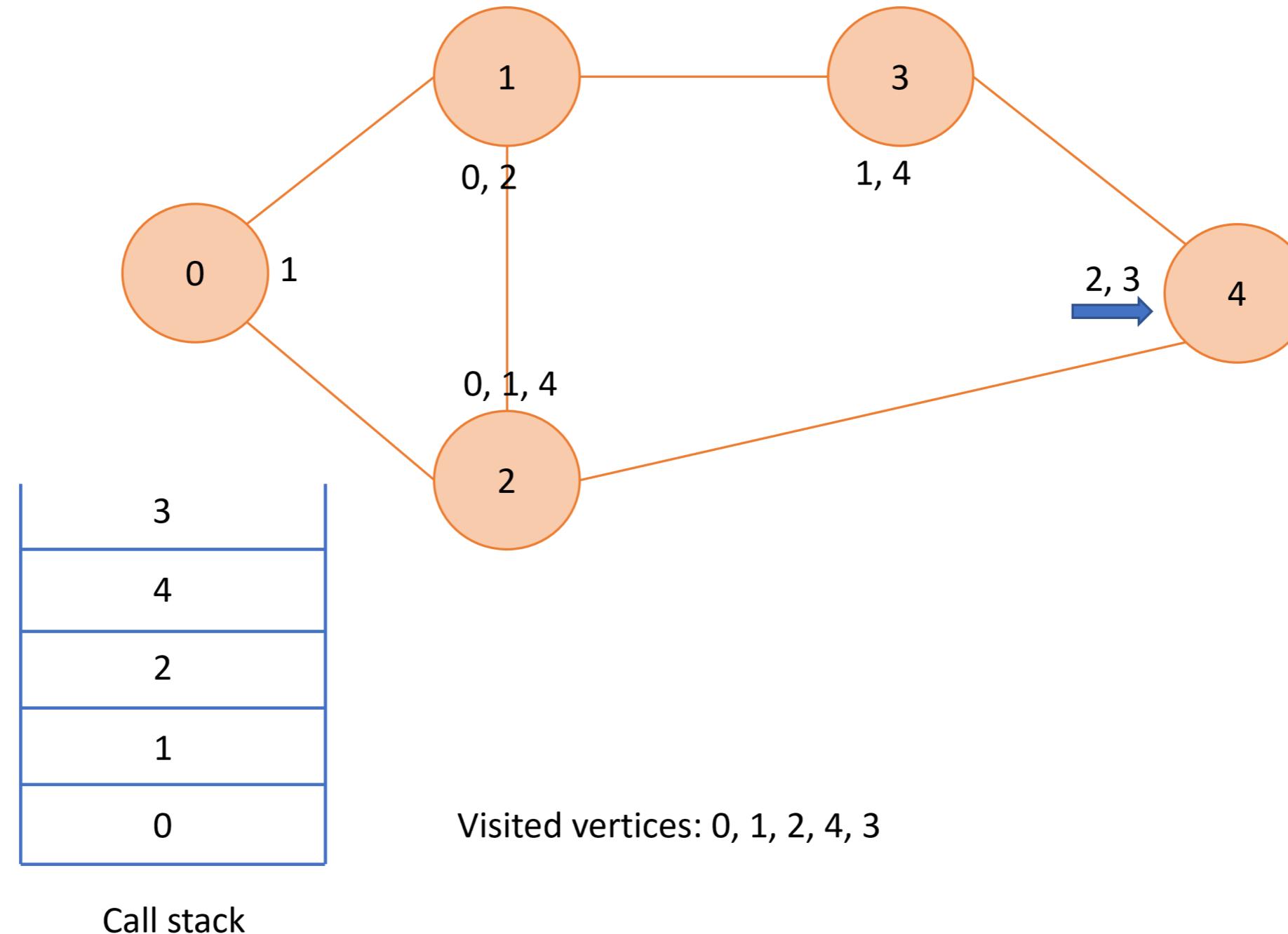
Depth first search - graphs



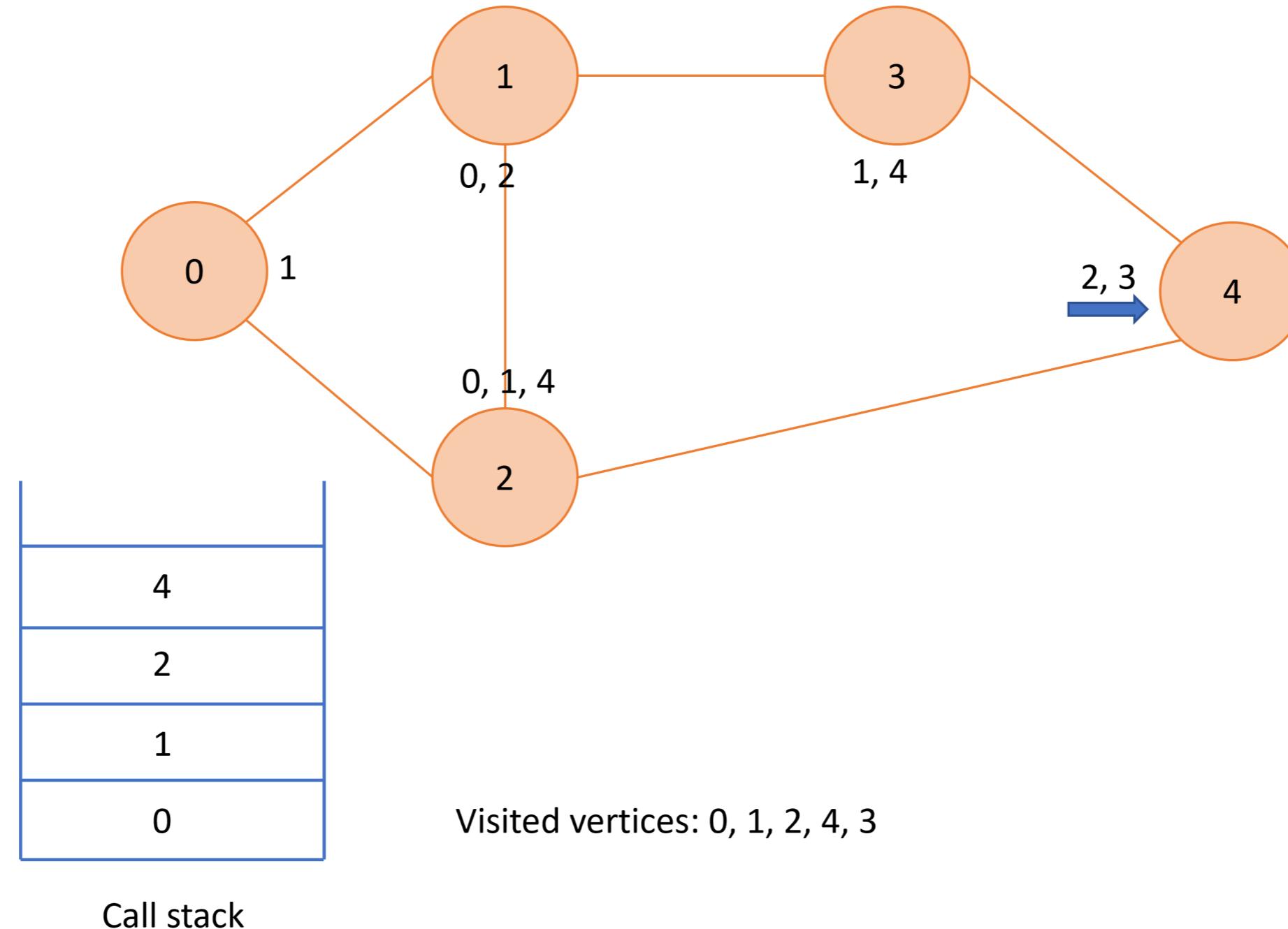
Depth first search - graphs



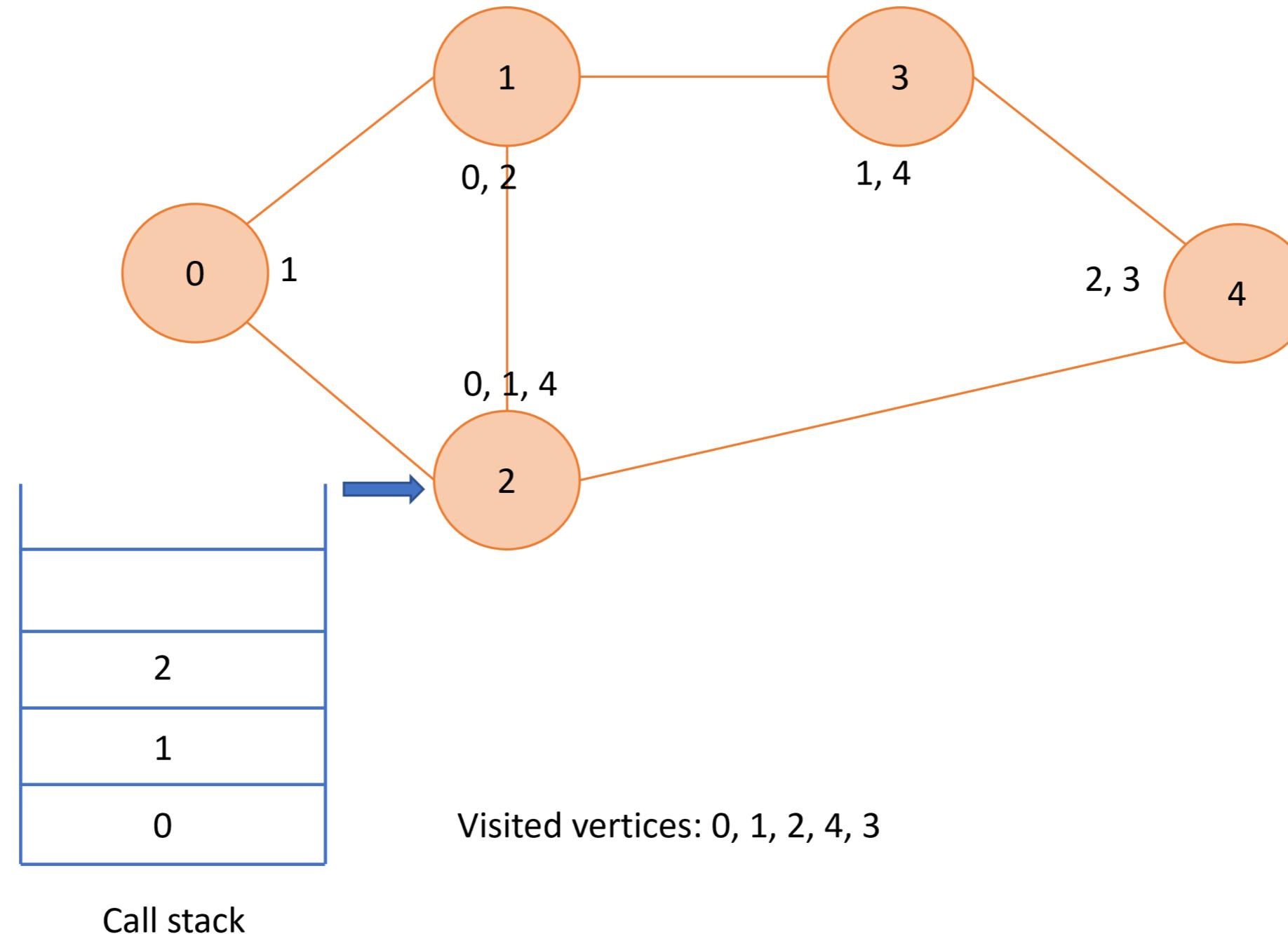
Depth first search - graphs



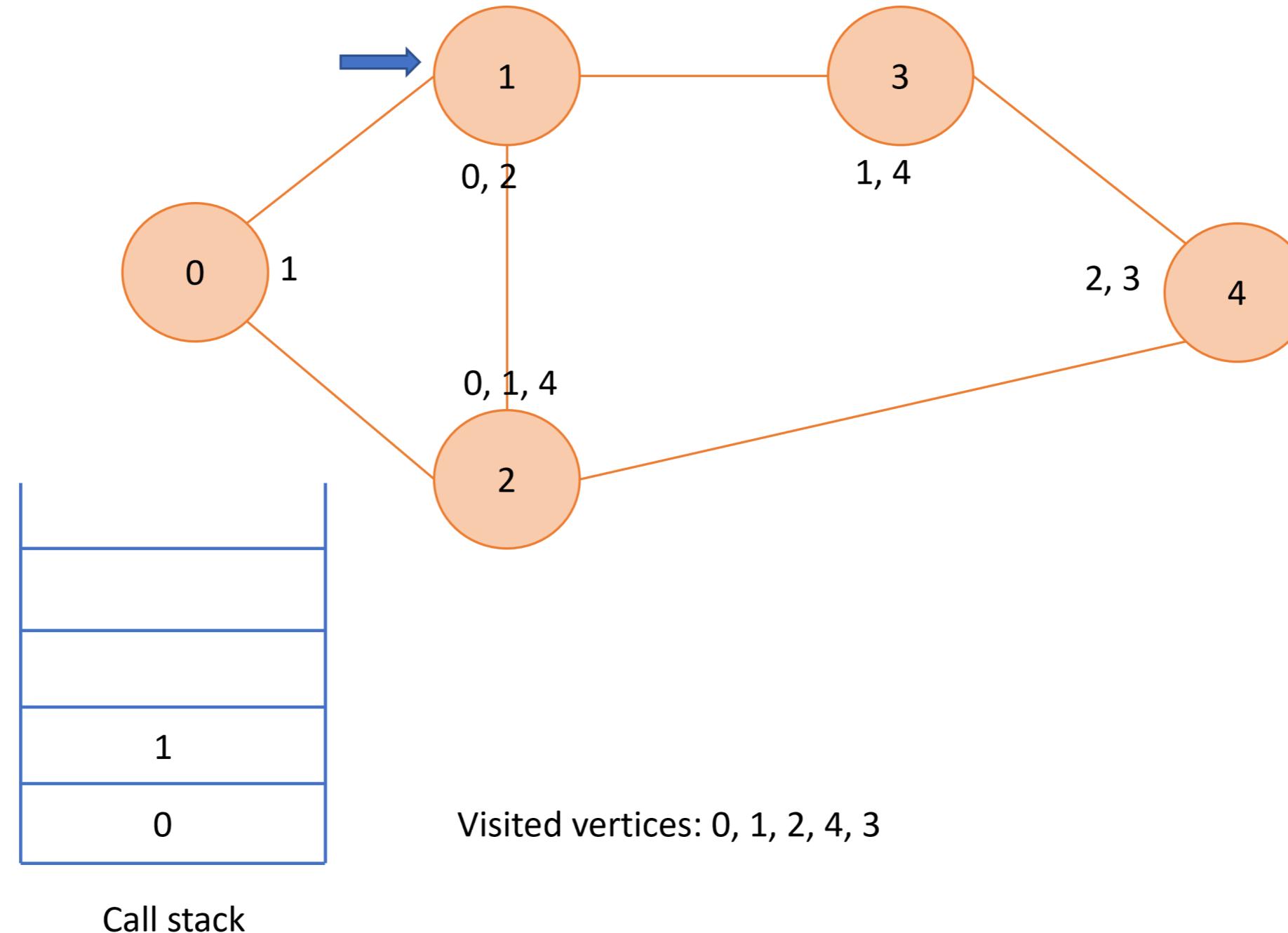
Depth first search - graphs



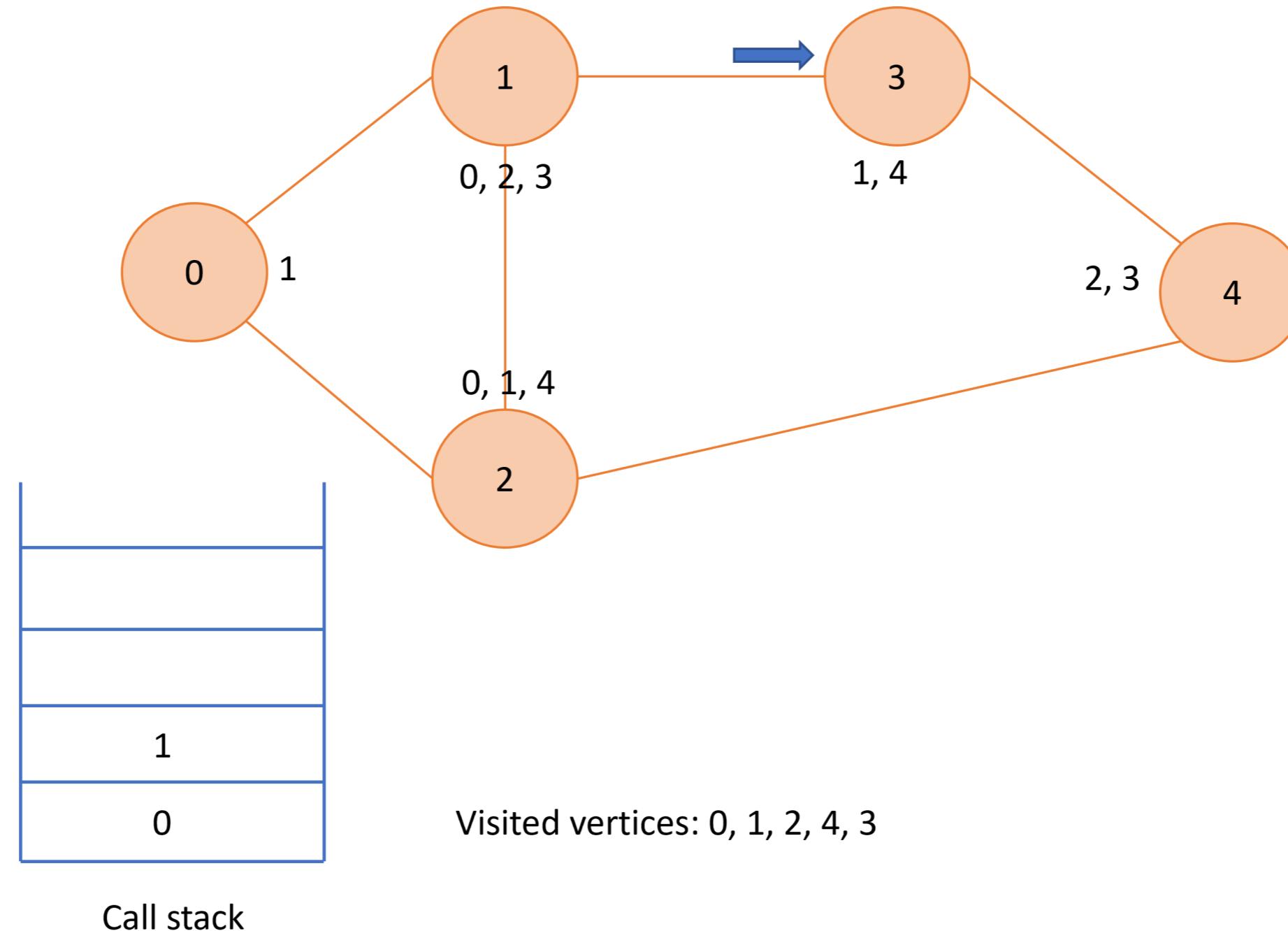
Depth first search - graphs



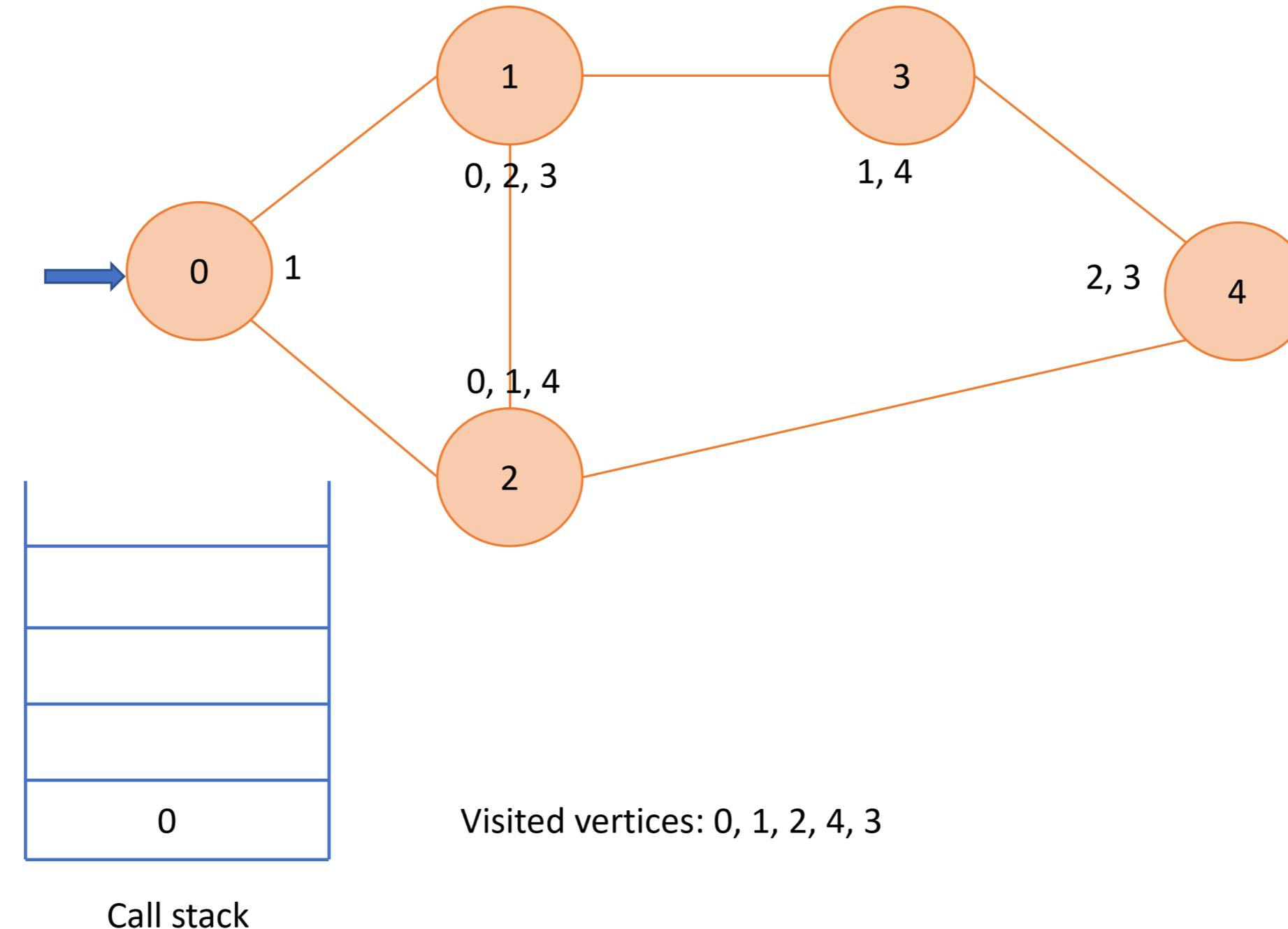
Depth first search - graphs



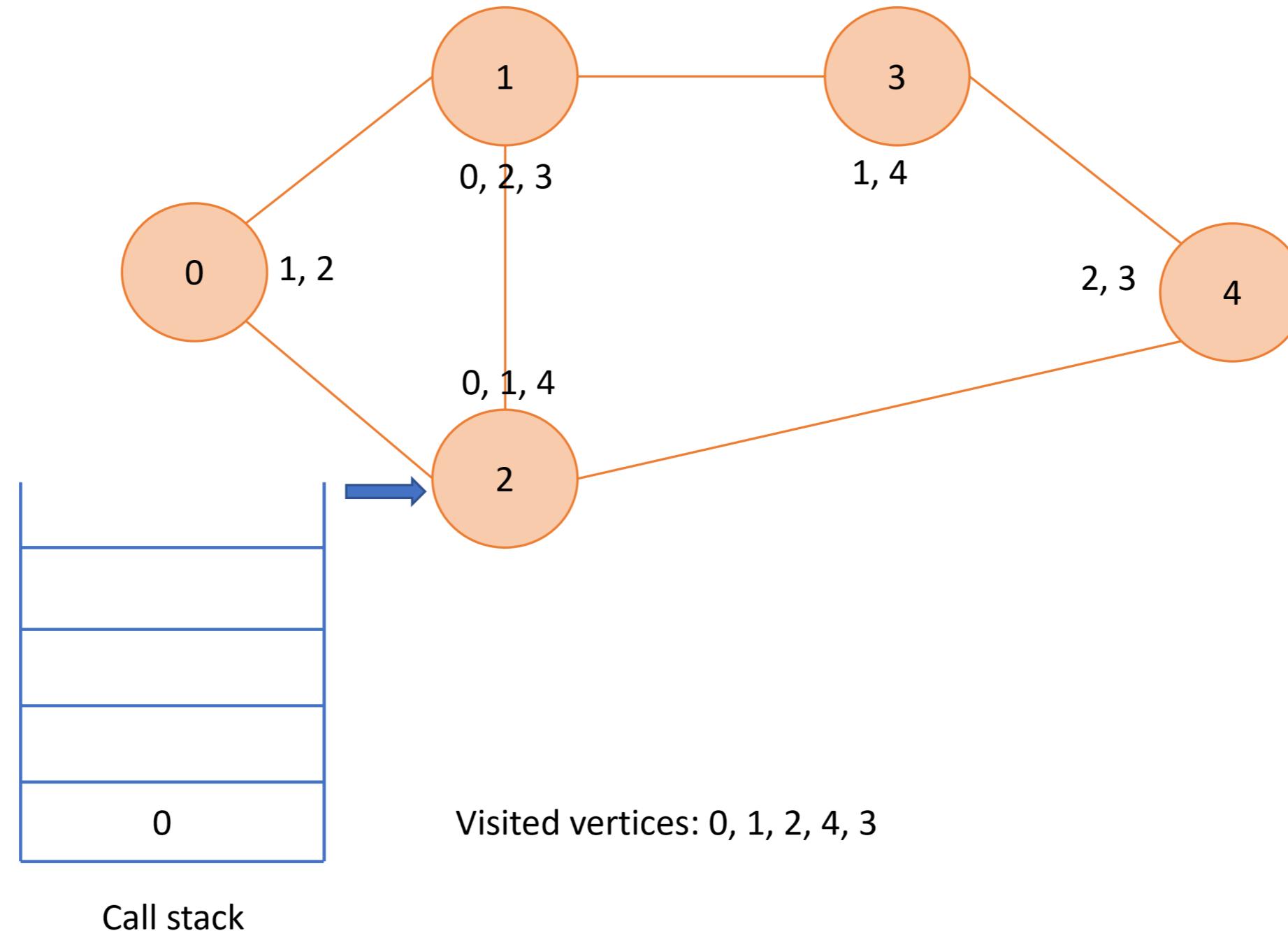
Depth first search - graphs



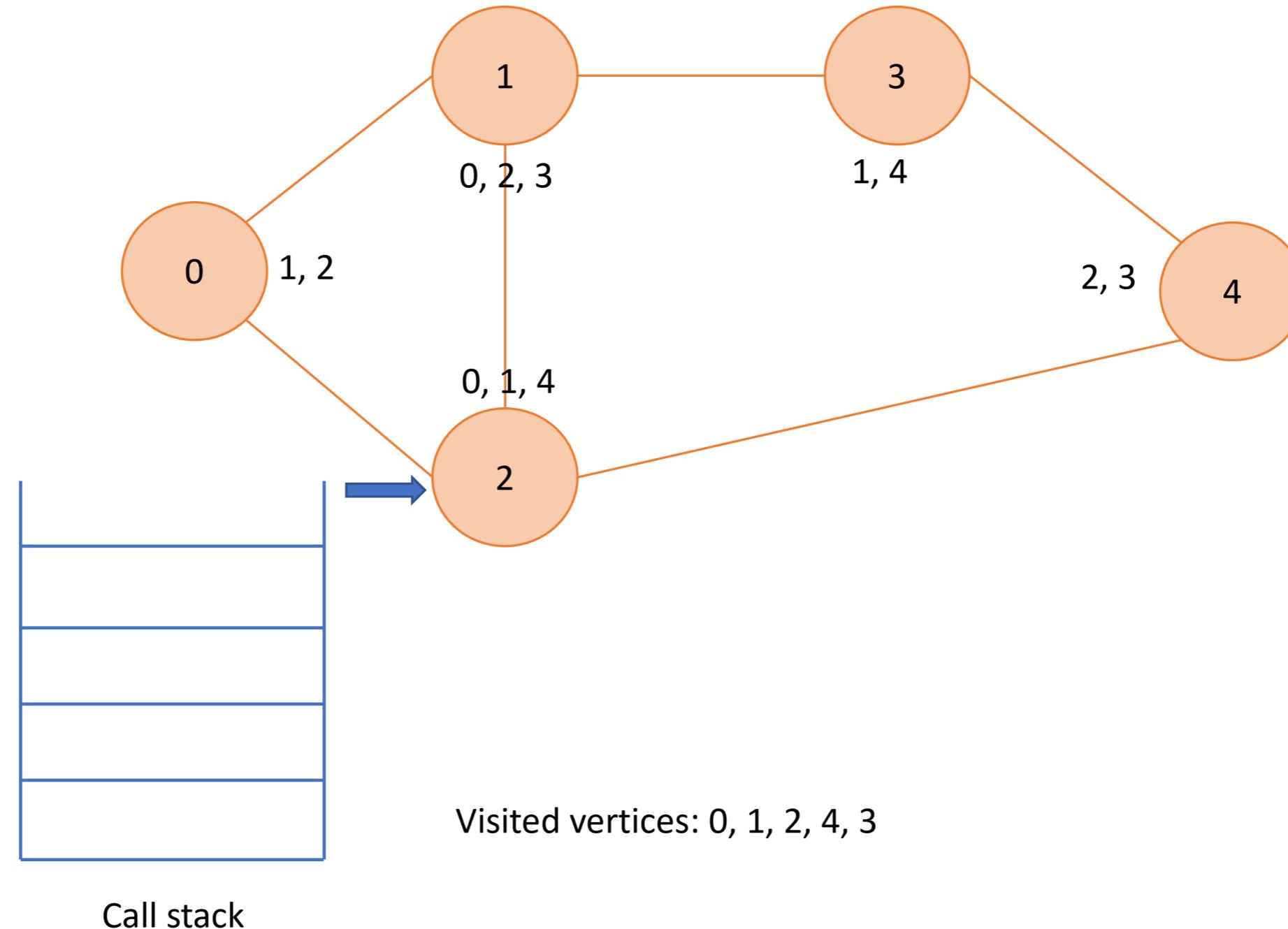
Depth first search - graphs



Depth first search - graphs



Depth first search - graphs



Depth first search - implementation

```
def dfs(visited_vertices, graph, current_vertex):
    if current_vertex not in visited_vertices:
        print(current_vertex)
        visited_vertices.add(current_vertex)
        for adjacent_vertex in graph[current_vertex]:
            dfs(visited_vertices, graph, adjacent_vertex)
```

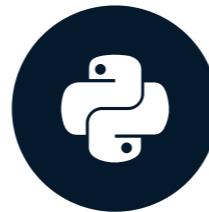
- Complexity: $O(V + E)$
 - V -> number of vertices
 - E -> number of edges

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Breadth First Search (BFS)

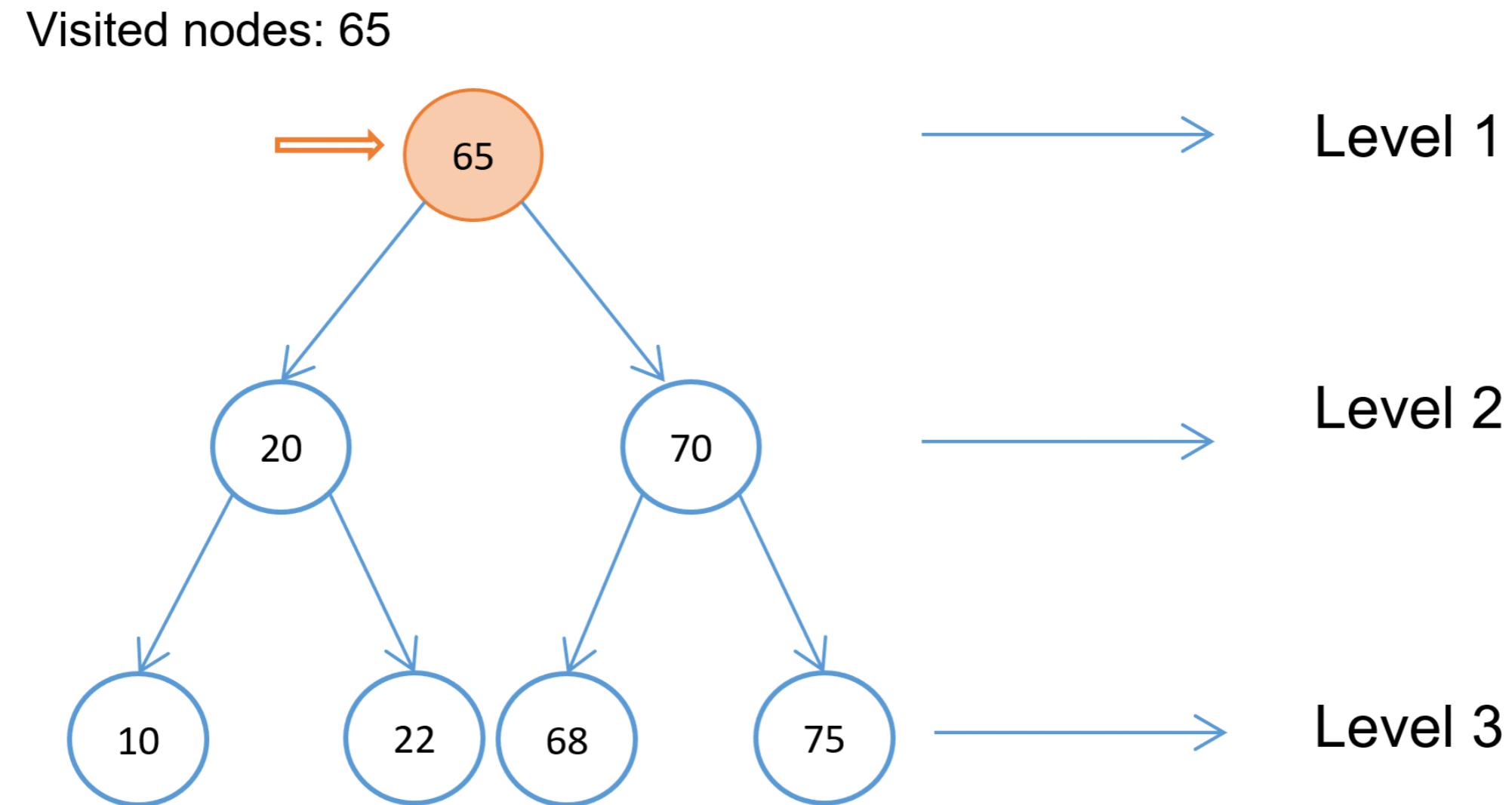
DATA STRUCTURES AND ALGORITHMS IN PYTHON



Miriam Antona
Software engineer

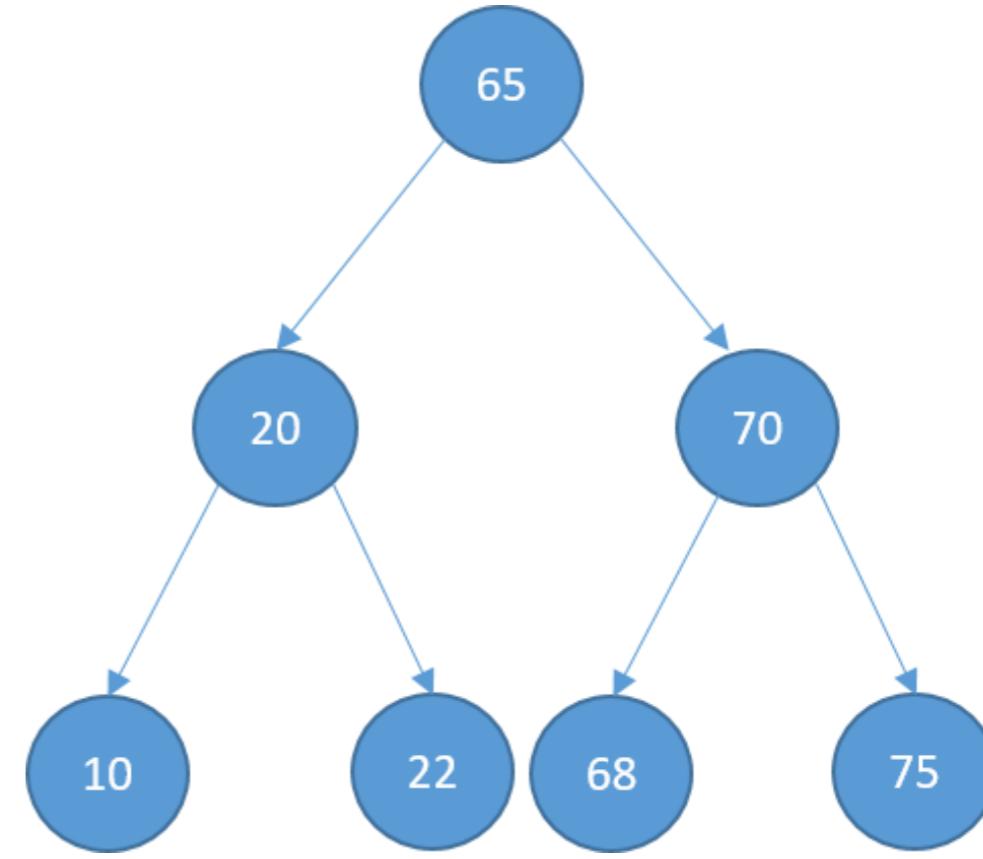
Breadth first search - binary trees

- Starts from the root
- Visits every node of every level



Breadth first search - binary trees

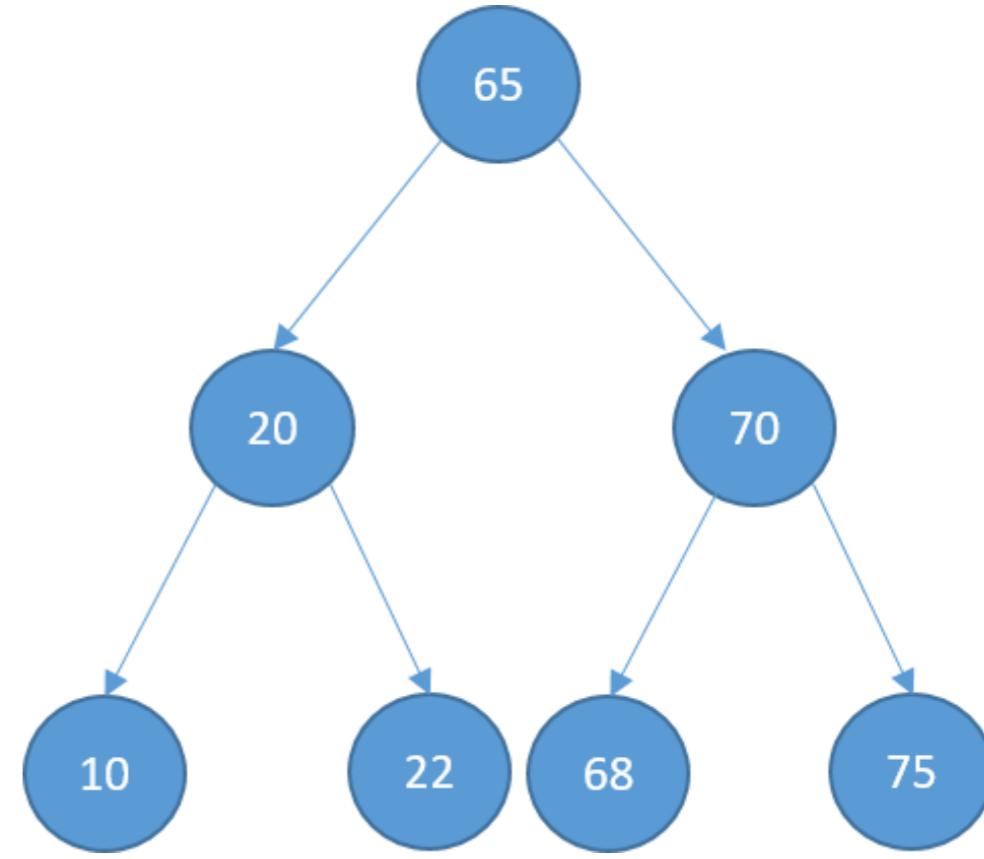
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()
```



- `visited_nodes`:
- `bfs_queue`:

Breadth first search - binary trees

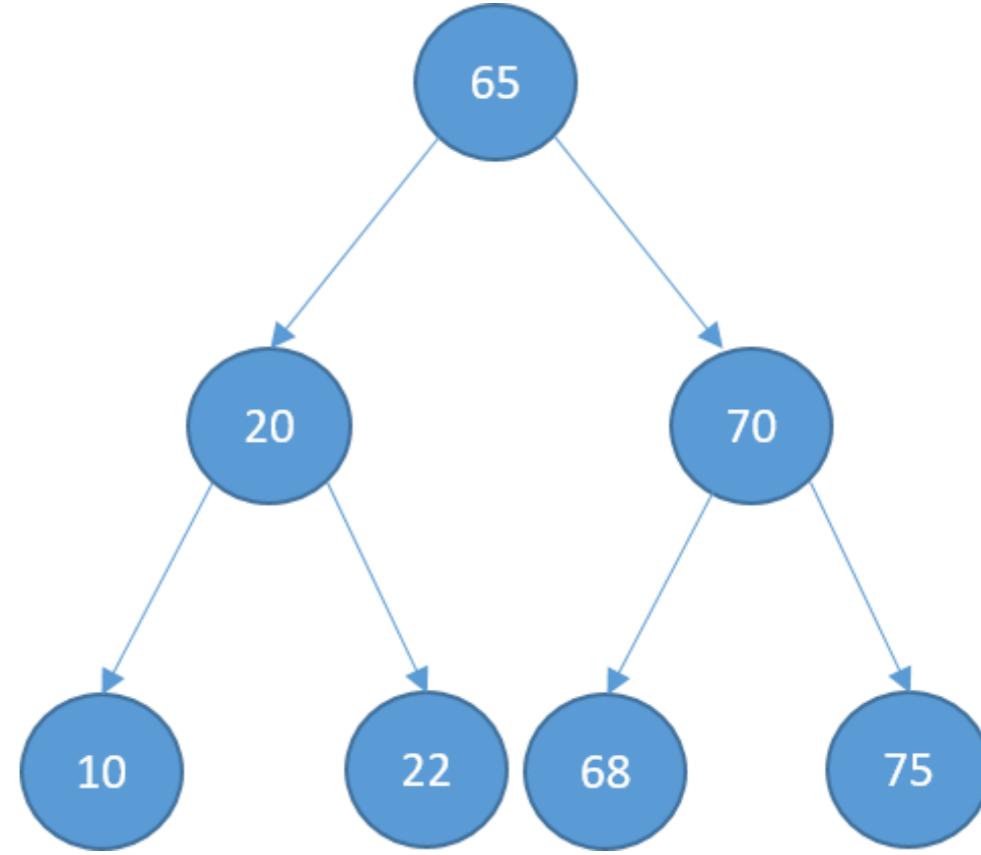
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():
```



- `visited_nodes`:
- `bfs_queue`: 65

Breadth first search - binary trees

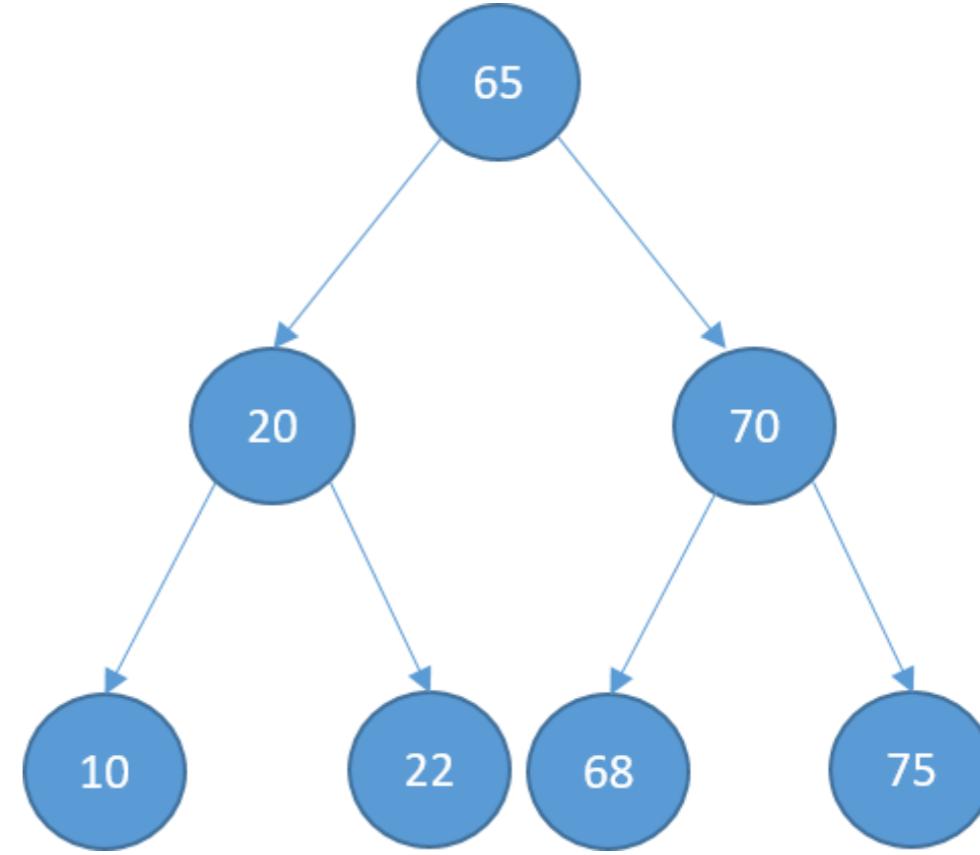
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()
```



- `visited_nodes`:
- `bfs_queue`:
- `current_node`: 65

Breadth first search - binary trees

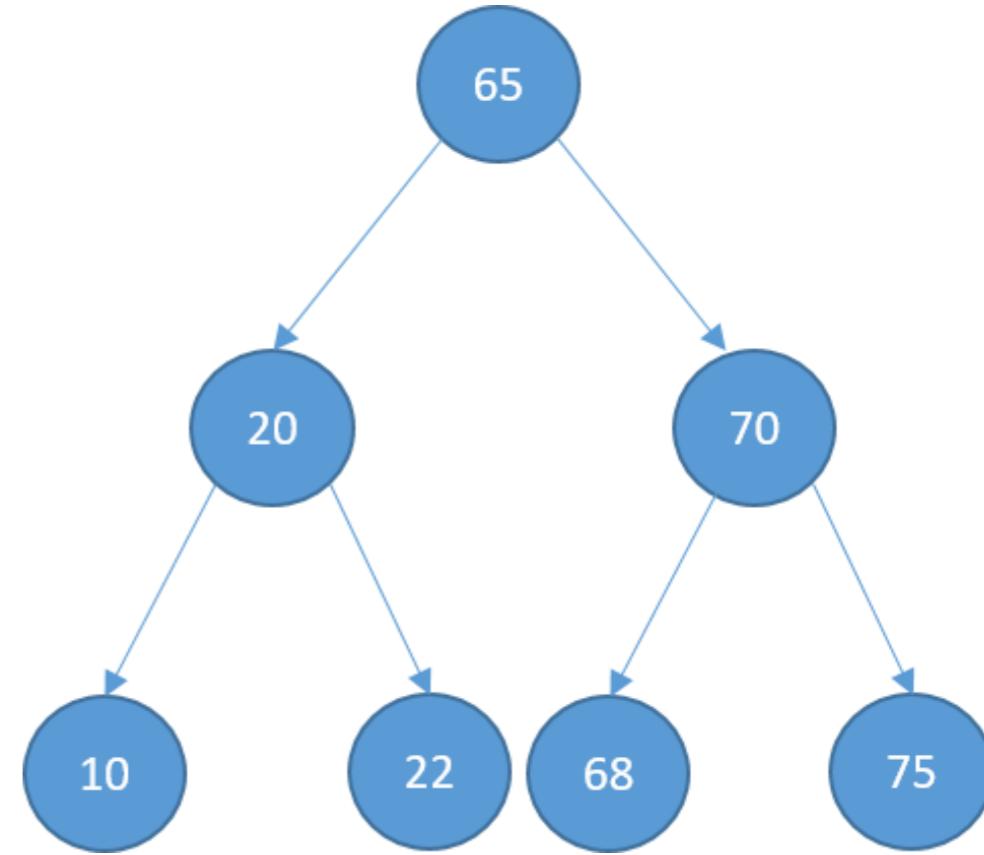
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:
```



- visited_nodes: 65
- bfs_queue:
- current_node: 65

Breadth first search - binary trees

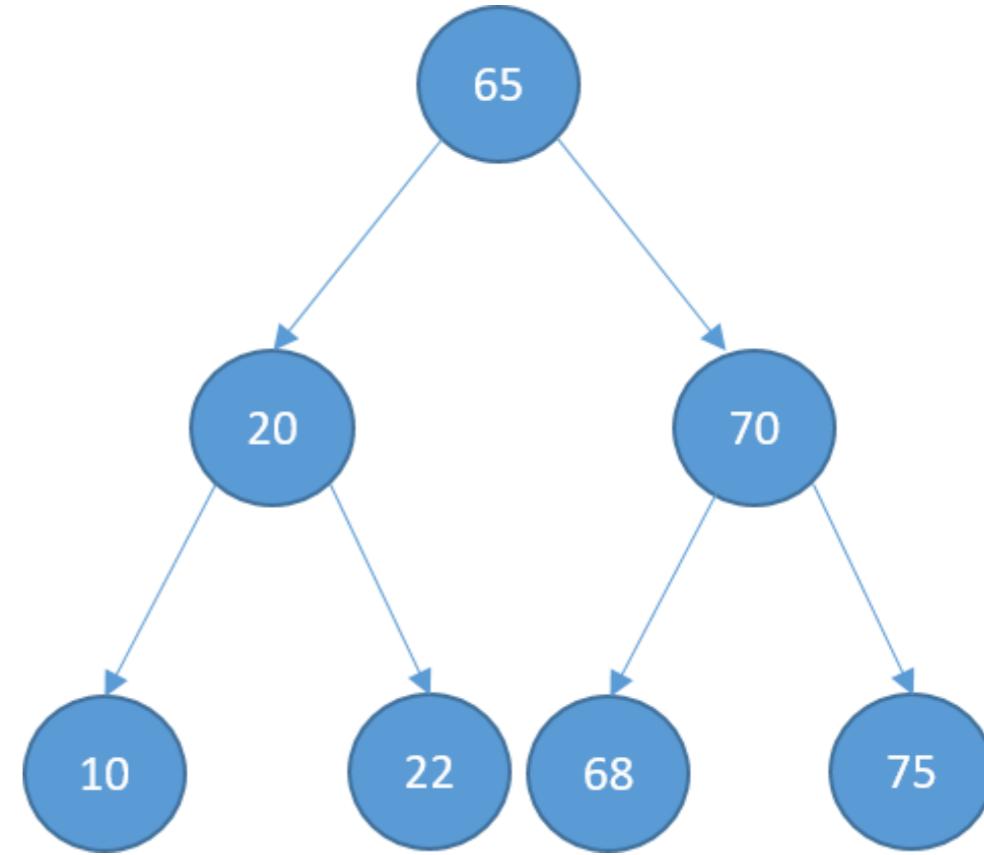
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)
```



- visited_nodes: 65
- bfs_queue: 20
- current_node: 65

Breadth first search - binary trees

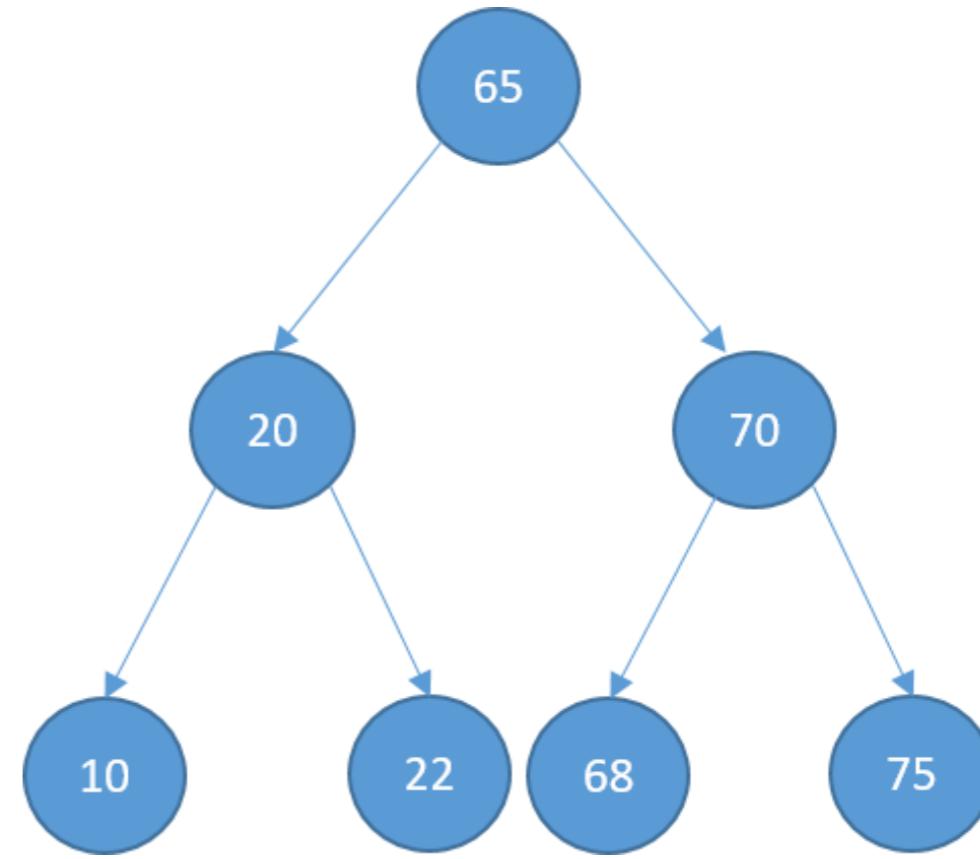
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65
- bfs_queue: 20, 70
- current_node: 65

Breadth first search - binary trees

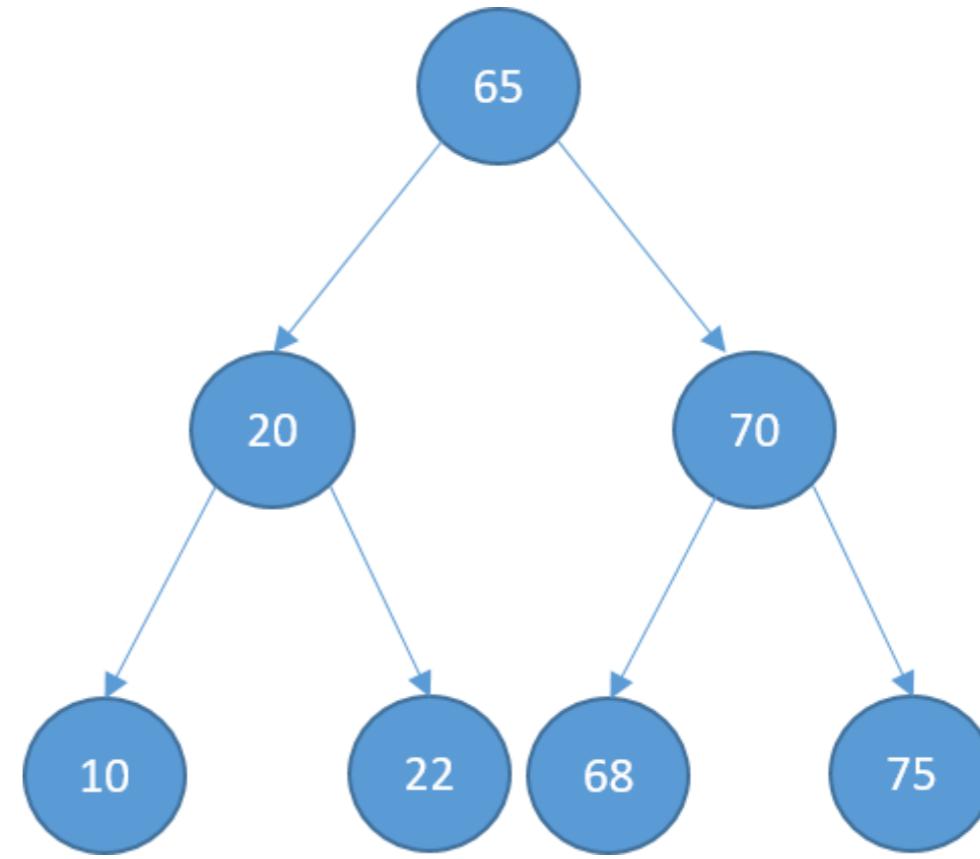
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65
- bfs_queue: 70
- current_node: 20

Breadth first search - binary trees

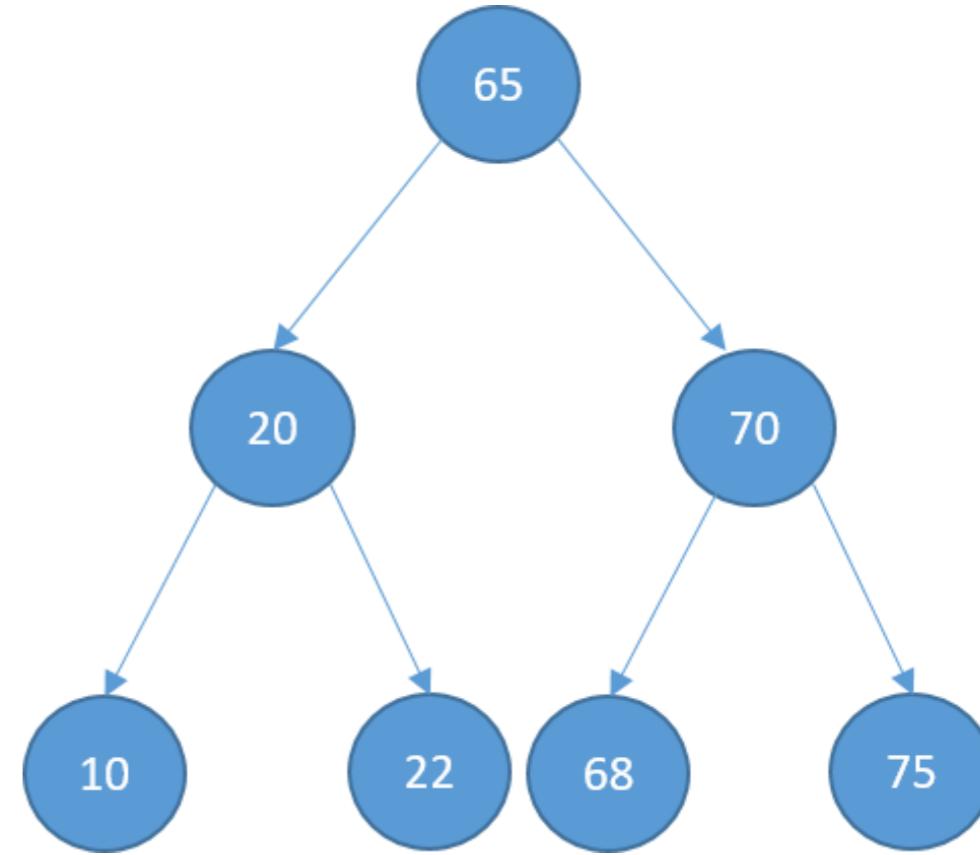
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20
- bfs_queue: 70
- current_node: 20

Breadth first search - binary trees

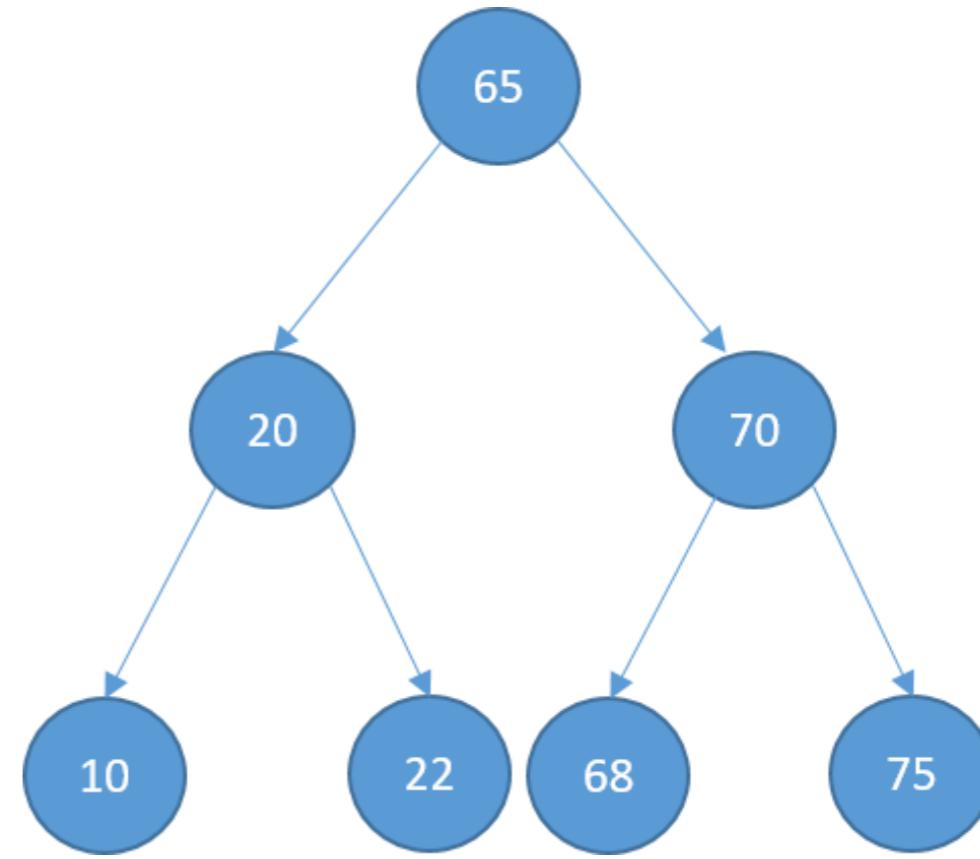
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20
- bfs_queue: 70, 10
- current_node: 20

Breadth first search - binary trees

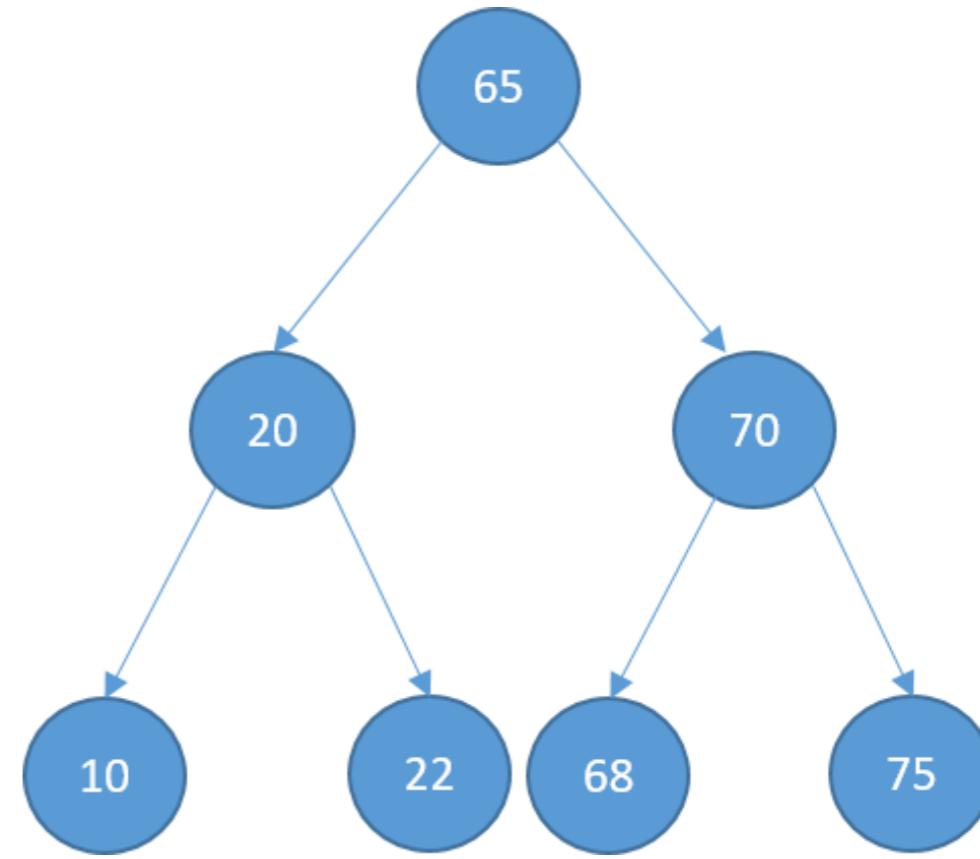
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20
- bfs_queue: 70, 10, 22
- current_node: 20

Breadth first search - binary trees

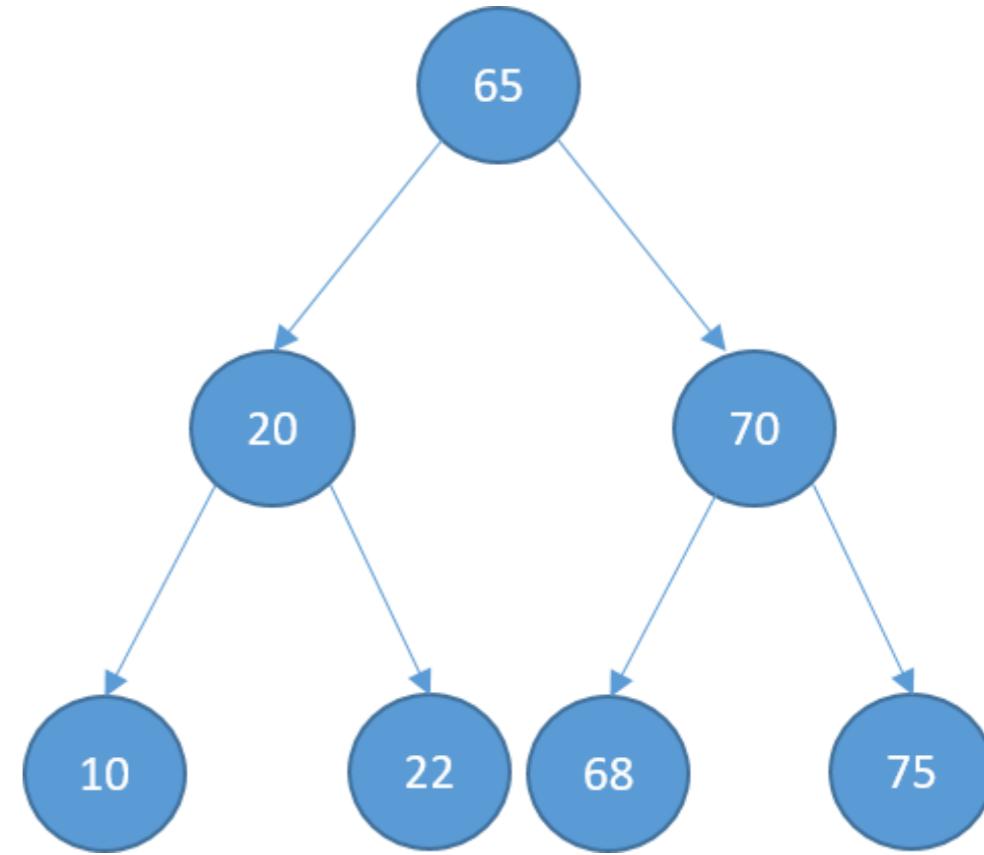
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20
- bfs_queue: 10, 22
- current_node: 70

Breadth first search - binary trees

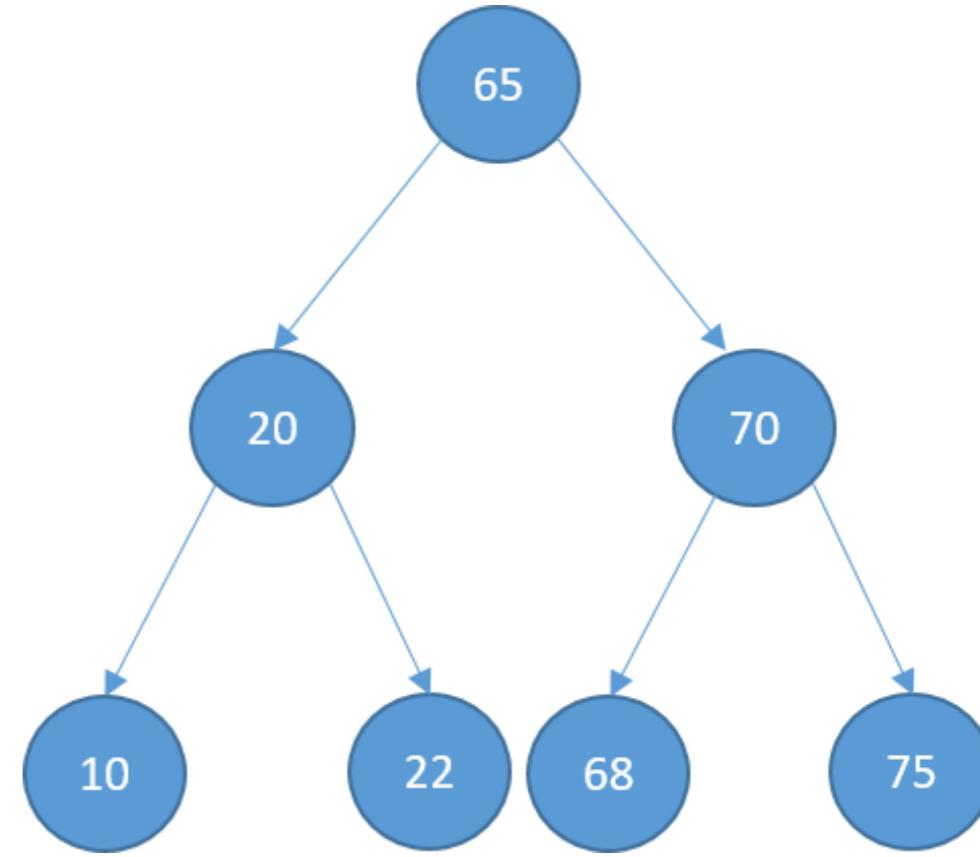
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70
- bfs_queue: 10, 22
- current_node: 70

Breadth first search - binary trees

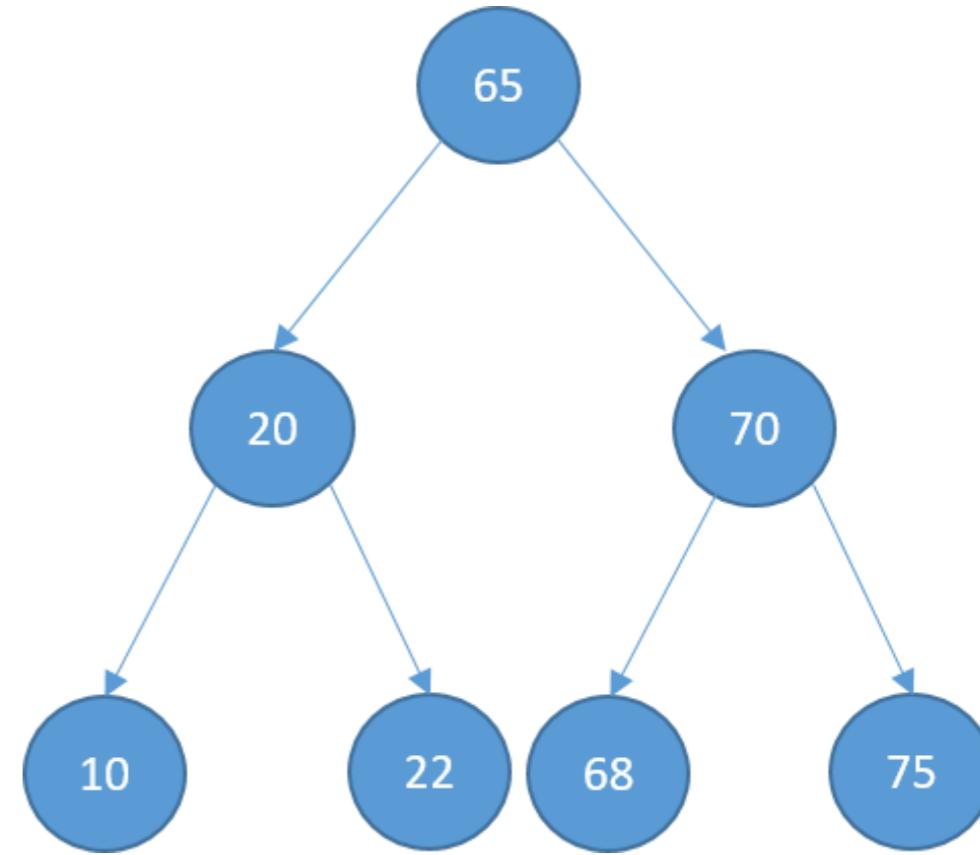
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70
- bfs_queue: 10, 22, 68
- current_node: 70

Breadth first search - binary trees

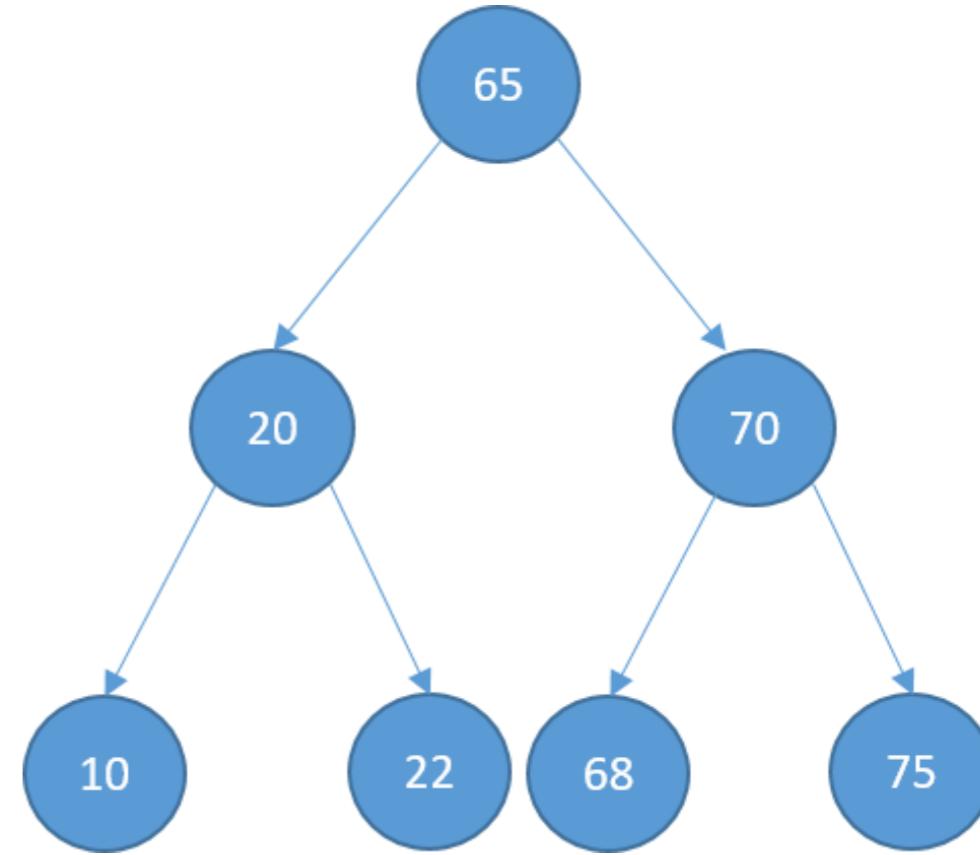
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70
- bfs_queue: 10, 22, 68, 75
- current_node: 70

Breadth first search - binary trees

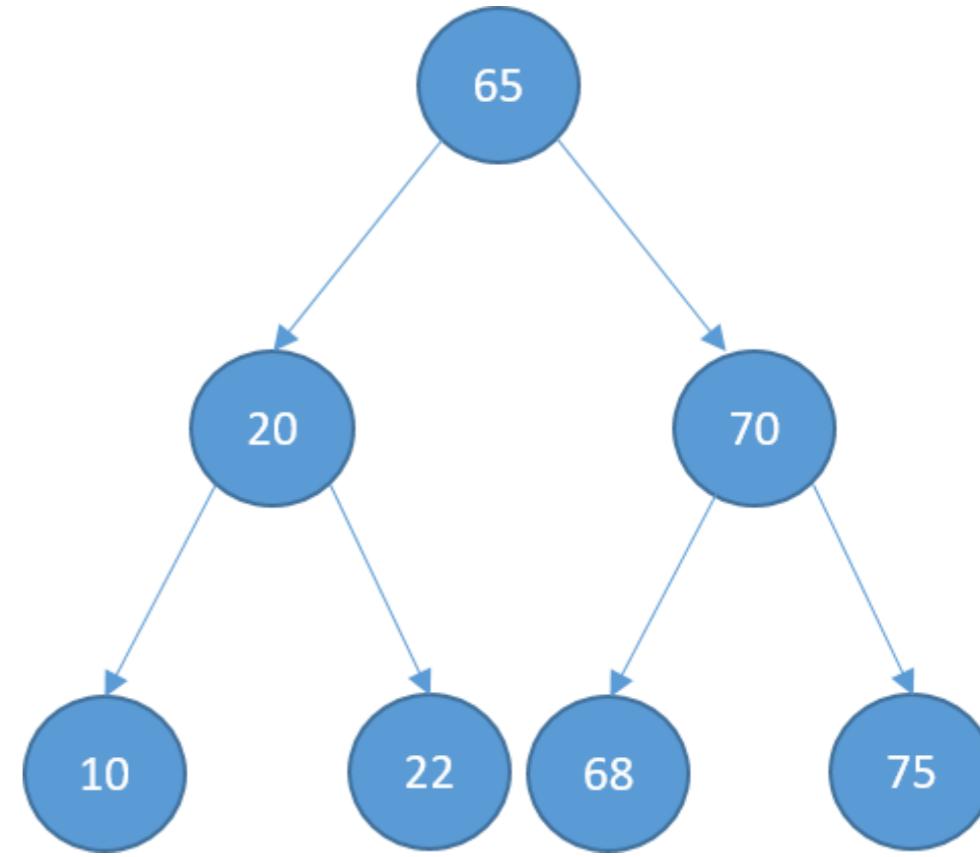
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70, 10
- bfs_queue: 22, 68, 75
- current_node: 10

Breadth first search - binary trees

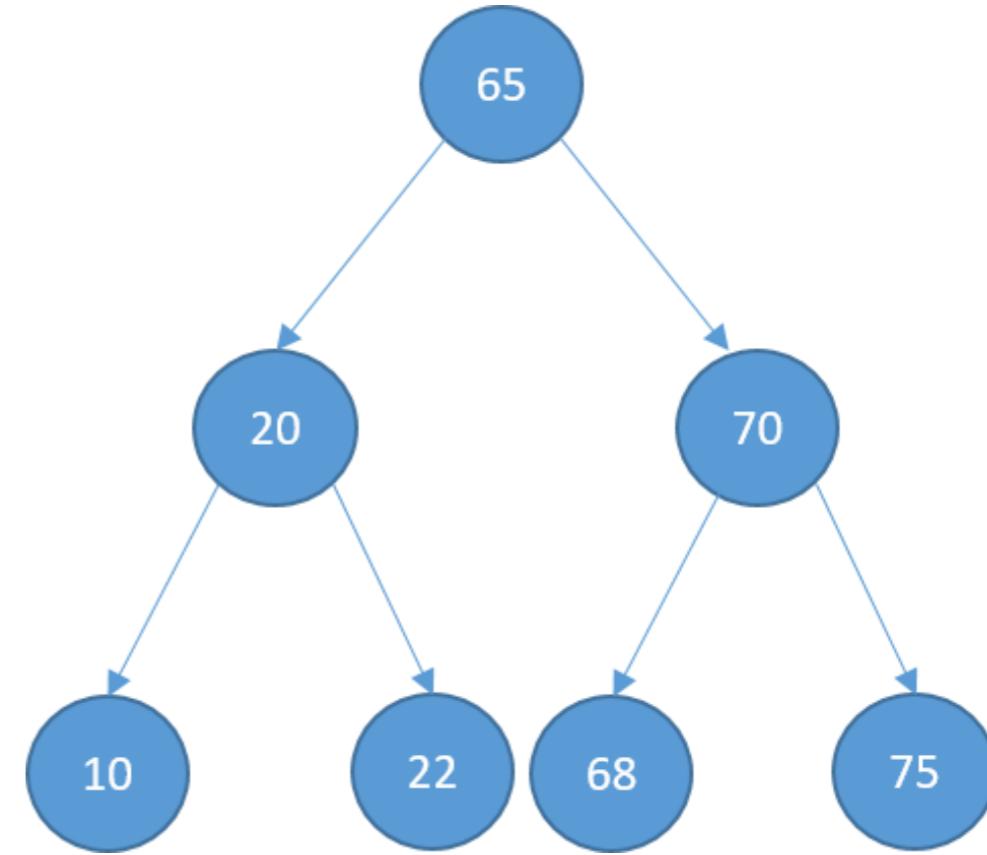
```
def bfs(self):
    if self.root:
        visited_nodes = []
        bfs_queue = queue.SimpleQueue()
        bfs_queue.put(self.root)
        while not bfs_queue.empty():
            current_node = bfs_queue.get()
            visited_nodes.append(current_node.data)
            if current_node.left:
                bfs_queue.put(current_node.left)
            if current_node.right:
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70, 10
- bfs_queue: 68, 75
- current_node: 22

Breadth first search - binary trees

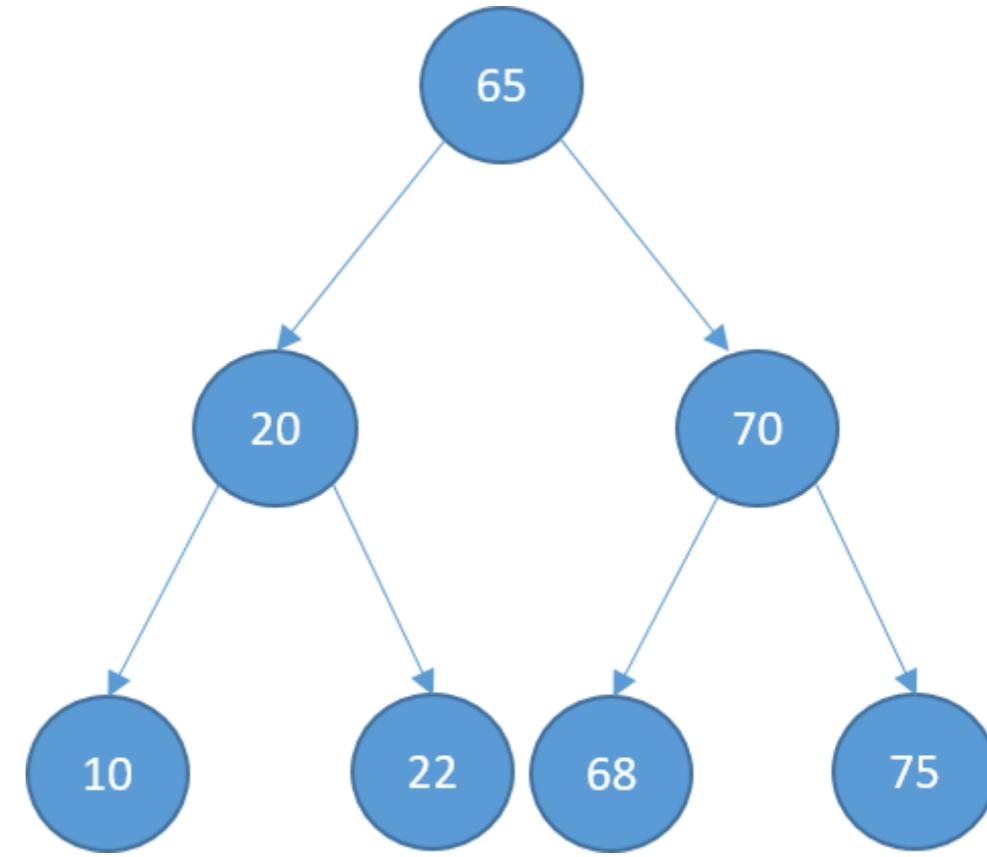
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70, 10, 22
- bfs_queue: 68, 75
- current_node: 22

Breadth first search - binary trees

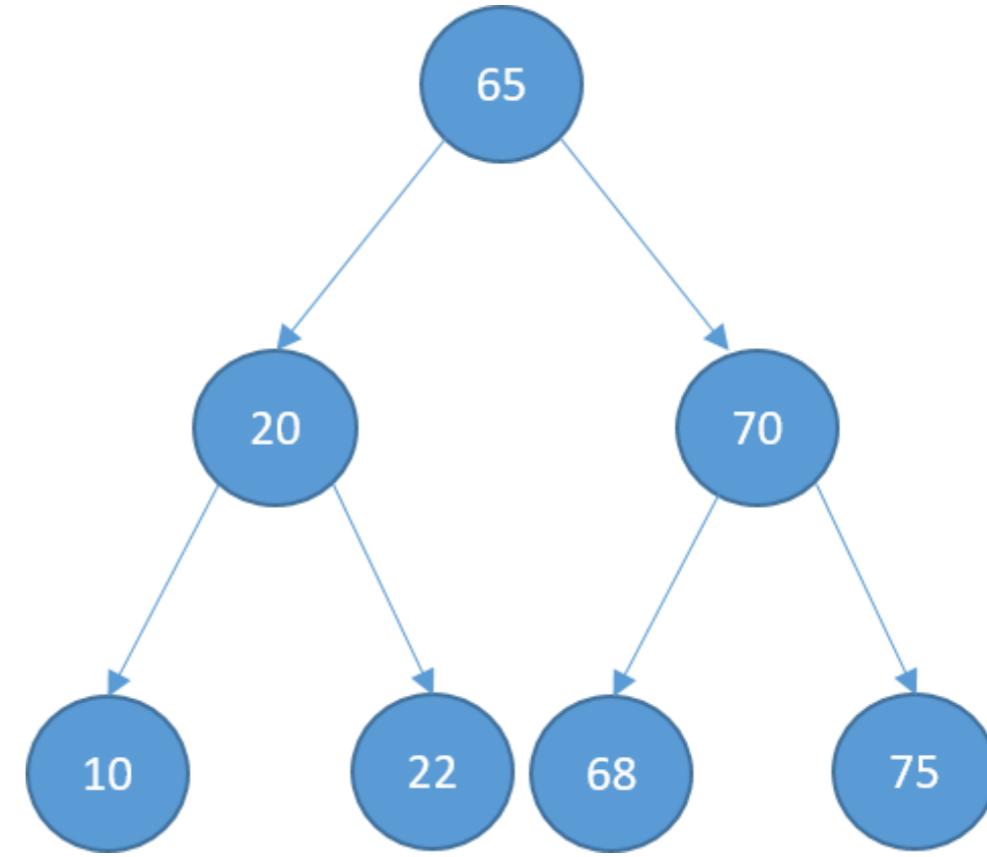
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70, 10, 22, 68
- bfs_queue: 75
- current_node: 68

Breadth first search - binary trees

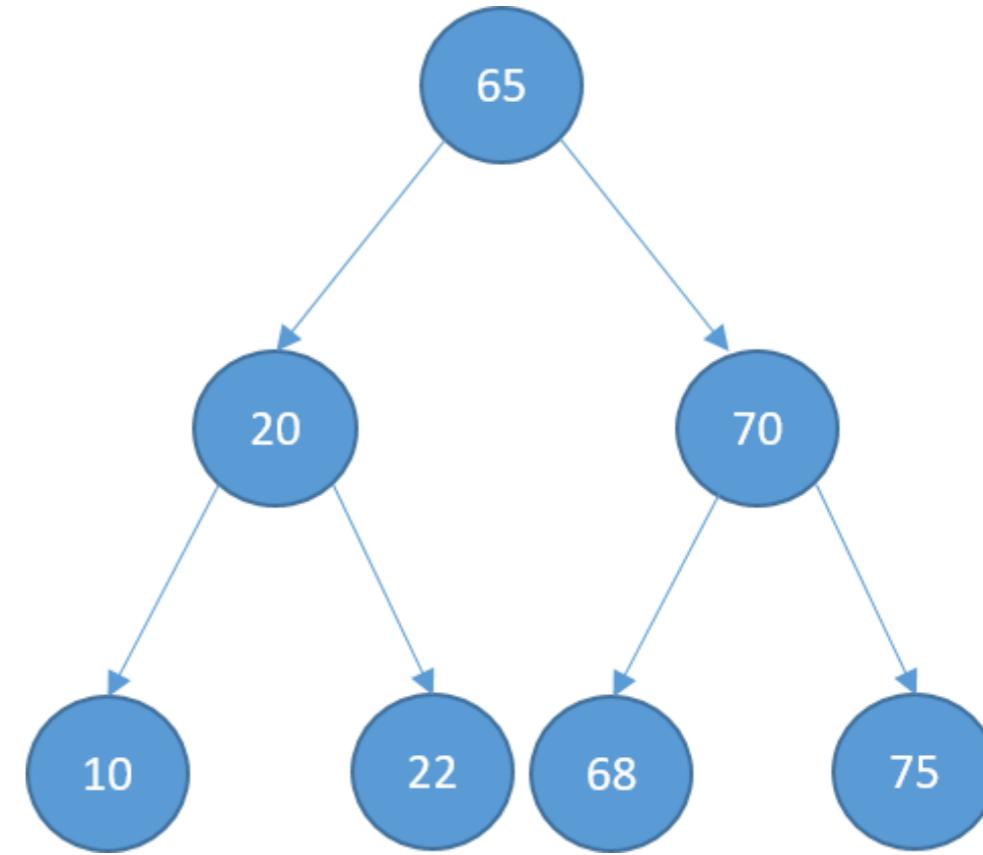
```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```



- visited_nodes: 65, 20, 70, 10, 22, 68
- bfs_queue:
- current_node: 75

Breadth first search - binary trees

```
def bfs(self):  
    if self.root:  
        visited_nodes = []  
        bfs_queue = queue.SimpleQueue()  
        bfs_queue.put(self.root)  
        while not bfs_queue.empty():  
            current_node = bfs_queue.get()  
            visited_nodes.append(current_node.data)  
            if current_node.left:  
                bfs_queue.put(current_node.left)  
            if current_node.right:  
                bfs_queue.put(current_node.right)
```

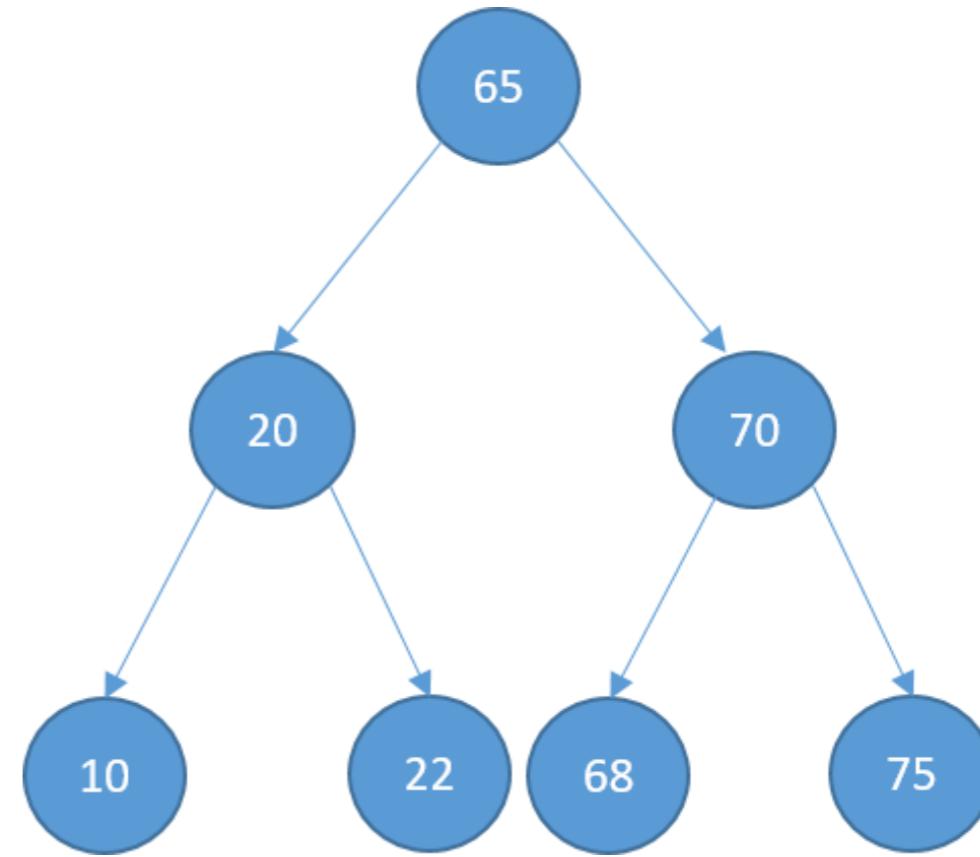


- visited_nodes: 65, 20, 70, 10, 22, 68, 75
- bfs_queue:
- current_node: 75

Breadth first search - binary trees

```
def bfs(self):
    if self.root:
        visited_nodes = []
        bfs_queue = queue.SimpleQueue()
        bfs_queue.put(self.root)
        while not bfs_queue.empty():
            current_node = bfs_queue.get()
            visited_nodes.append(current_node.data)
            if current_node.left:
                bfs_queue.put(current_node.left)
            if current_node.right:
                bfs_queue.put(current_node.right)
    return visited_nodes
```

- Complexity: $O(n)$



- visited_nodes: 65, 20, 70, 10, 22, 68, 75
- bfs_queue:
- current_node: 75

Breadth first search - graphs

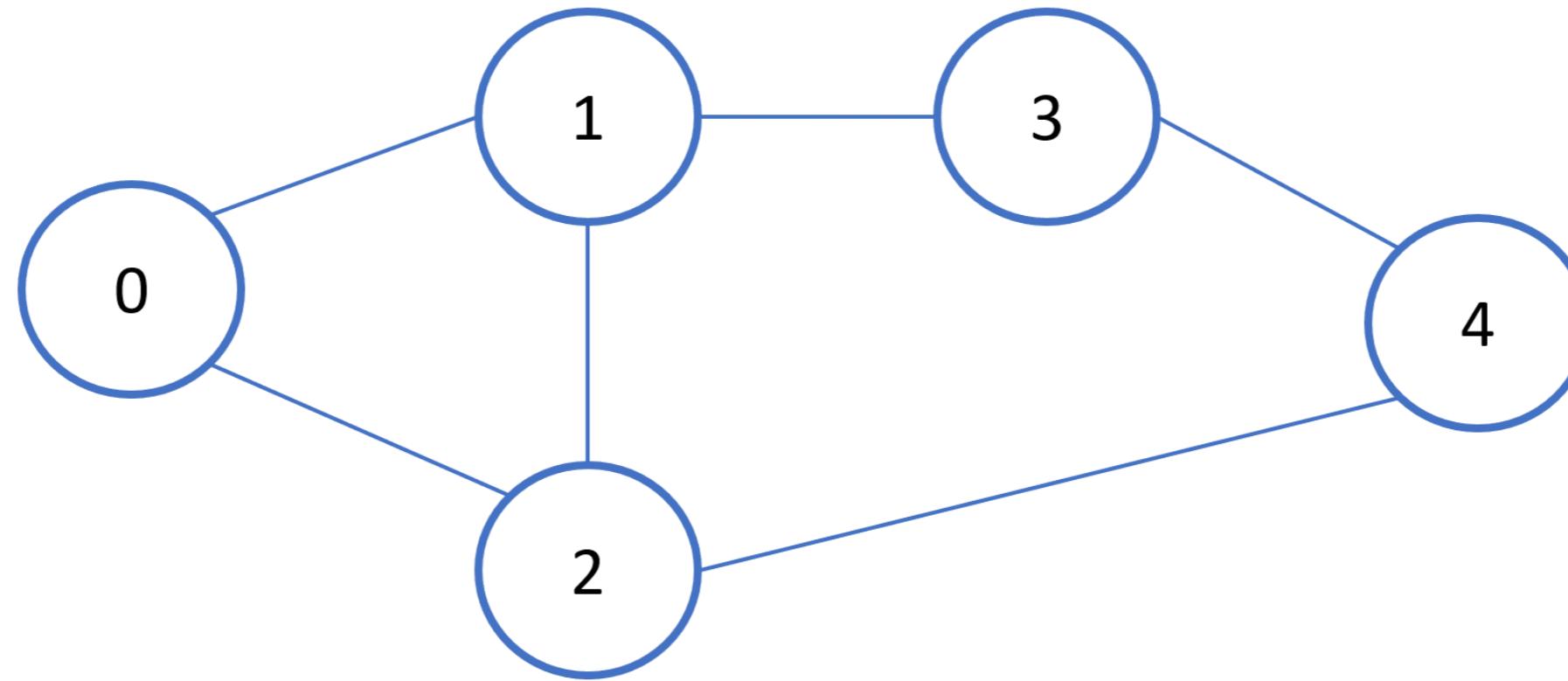
- Graphs can have cycles
 - Need to check if the vertices have already been visited

Breadth first search - graphs

```
def bfs(graph, initial_vertex):
    visited_vertices = []
    bfs_queue = queue.SimpleQueue()
    bfs_queue.put(initial_vertex)
    visited_vertices.append(initial_vertex)
    while not bfs_queue.empty():
        current_vertex = bfs_queue.get()
        for adjacent_vertex in graph[current_vertex]:
            if adjacent_vertex not in visited_vertices:
                visited_vertices.append(adjacent_vertex)
                bfs_queue.put(adjacent_vertex)
    return visited_vertices
```

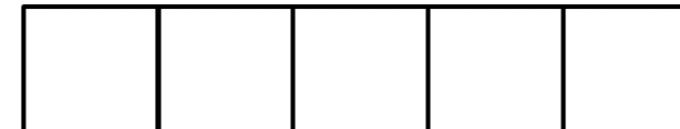
- Complexity: $O(V + E)$
 - $V \rightarrow$ number of vertices
 - $E \rightarrow$ number of edges

Breadth first search - graphs



Visited vertices:

Queue:



Current vertex:

BFS vs DFS

BFS

- Target **close to the starting vertex**
- Applications:
 - Web crawling
 - Finding shortest path in unweighted graphs
 - Finding connected locations using GPS
 - Used as part of other more complex algorithms

DFS

- Target **far away from the starting vertex**
- Applications:
 - Solving puzzles with only one solution (e.g. mazes)
 - Detecting cycles in graphs
 - Finding shortest path in a weighted graph
 - Used as part of other more complex algorithms

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON