Data Structures and Algorithms

LECTURE 08: TREES

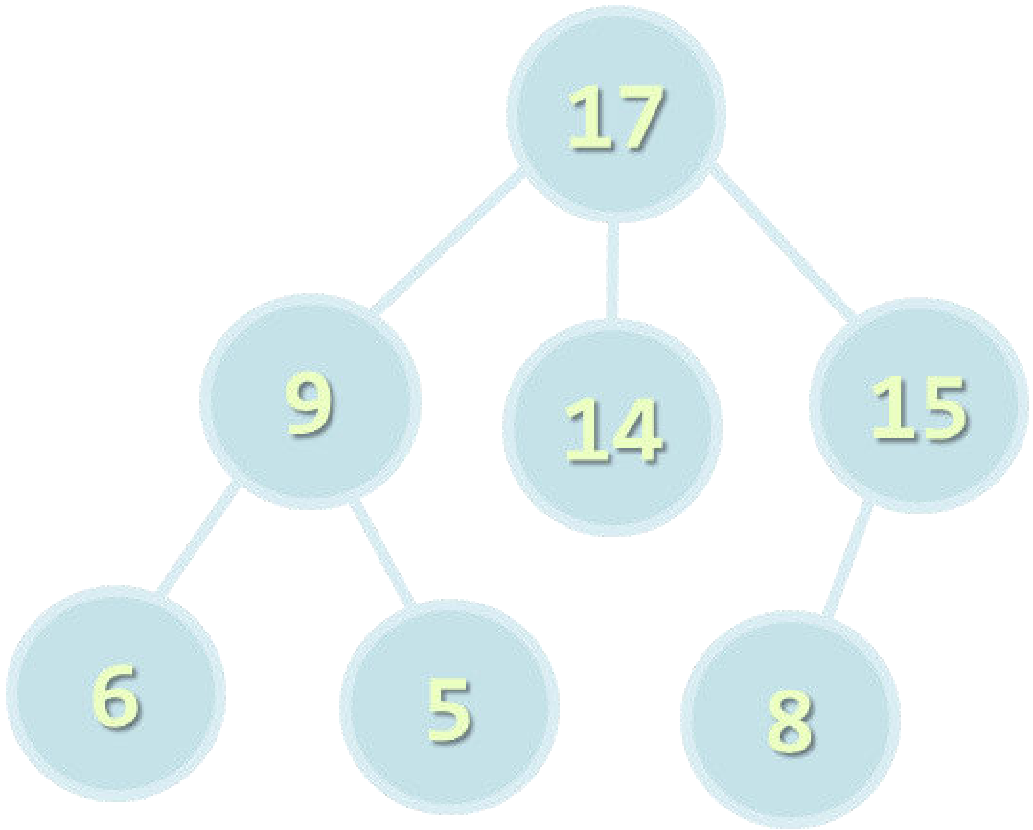
UNIVERSITY of GREENWICH

Alliance with FPT Education

Pearson BTEC

Contents

Why Trees?



Why Trees?

- So far we have learned how to implement linear data structures like: List, Queue, Stack, LinkedList etc…
- We did great job and learned how to take the best complexity we can, **was that enough**?
- Actually more of the operations we want to do like **search**, **insert** or **remove** are **linear** for **unordered** structures (sometimes we can do O(1)) but **not for search**

Why Trees?

- We used two types of implementation approaches:
  - Atop an **array** – this gave us the ability to **add elements with O(1)**, removing and searching were with **O(n)**. For sorted array we can search with **O(log(n))** but we need to **sort each time we add**.
  - By using **Node** implementation – we could **add and remove** elements **we have pointer** to with **O(1)**, however every other **operation is O(n)**. This time even if we keep the elements **sorted** we **can't get** search in **O(log(n)) but why**?

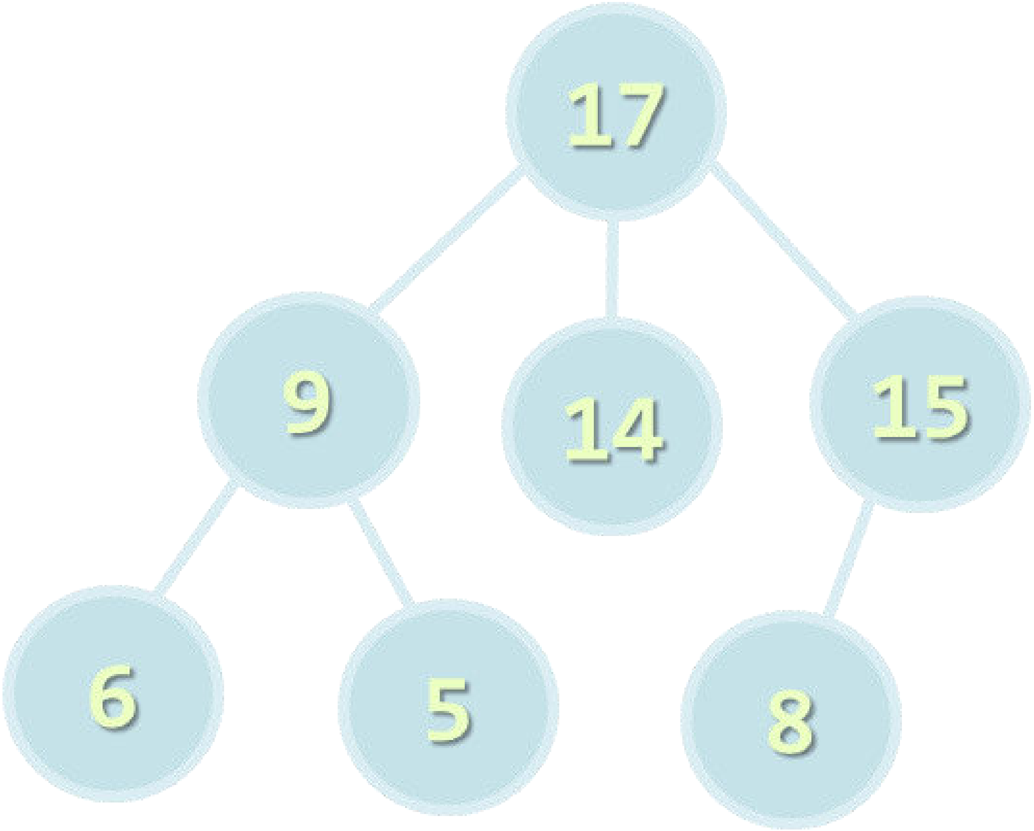
Why Trees?

- We want not only to store data **add** or **remove** elements in efficient manner but also to **search** for elements but **can** we do better than **O(n)**?
- Lets try to get **down** to **O(log(n))** by using **trees** and see if we can

Other Tree Benefits

- By learning how to work with trees you **actually** learn how to **work with**:
  - **Hierarchical** structures like: file system, project structures and code branching, NoSQL data storage etc…
  - **Markup** languages:
    - HTML
    - XML
- **DFS** and **BFS** algorithms

Trees and Related Terminology



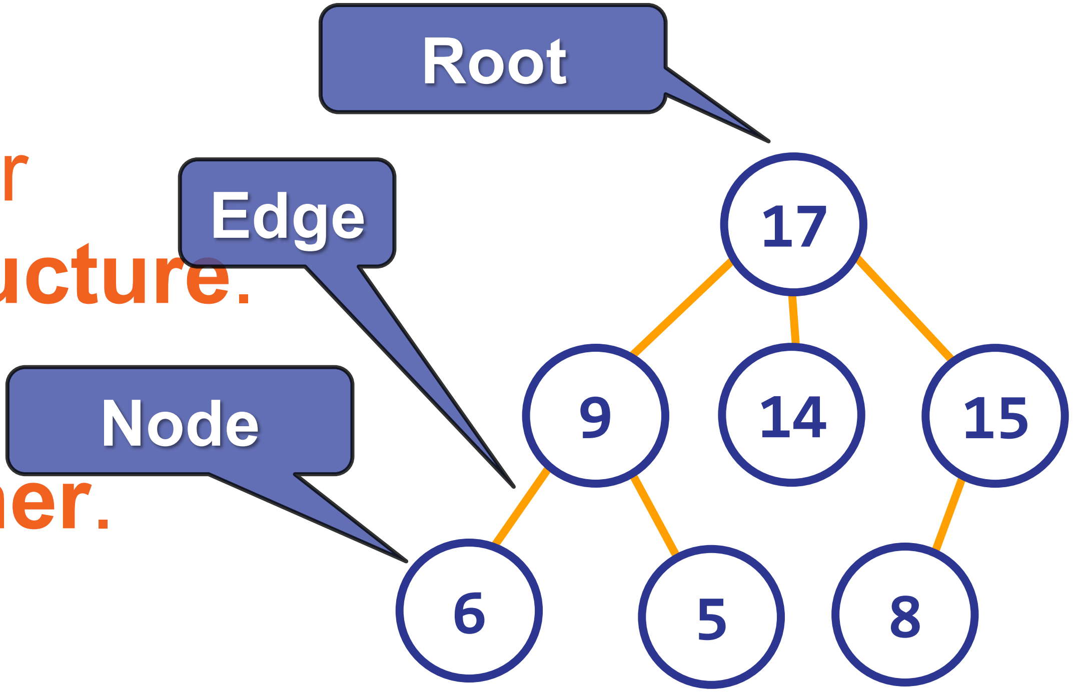
Node, Edge, Root, etc.

Tree Definition

- Tree is a widely used **abstract data type** (ADT) that simulates a hierarchical **tree structure**, with a root value and subtrees of children with a **parent node**, represented as a set of linked **nodes**.
- **Recursive definition** – a tree consists of a value and a forest (the subtrees of its children)
- One **reference** can point to **any given node** (a node has at **most** a **single** parent), and **no node** in the **tree point to the root**. Every node (other than the root) **must** have exactly **one parent**, and the **root must** have **no parents**.

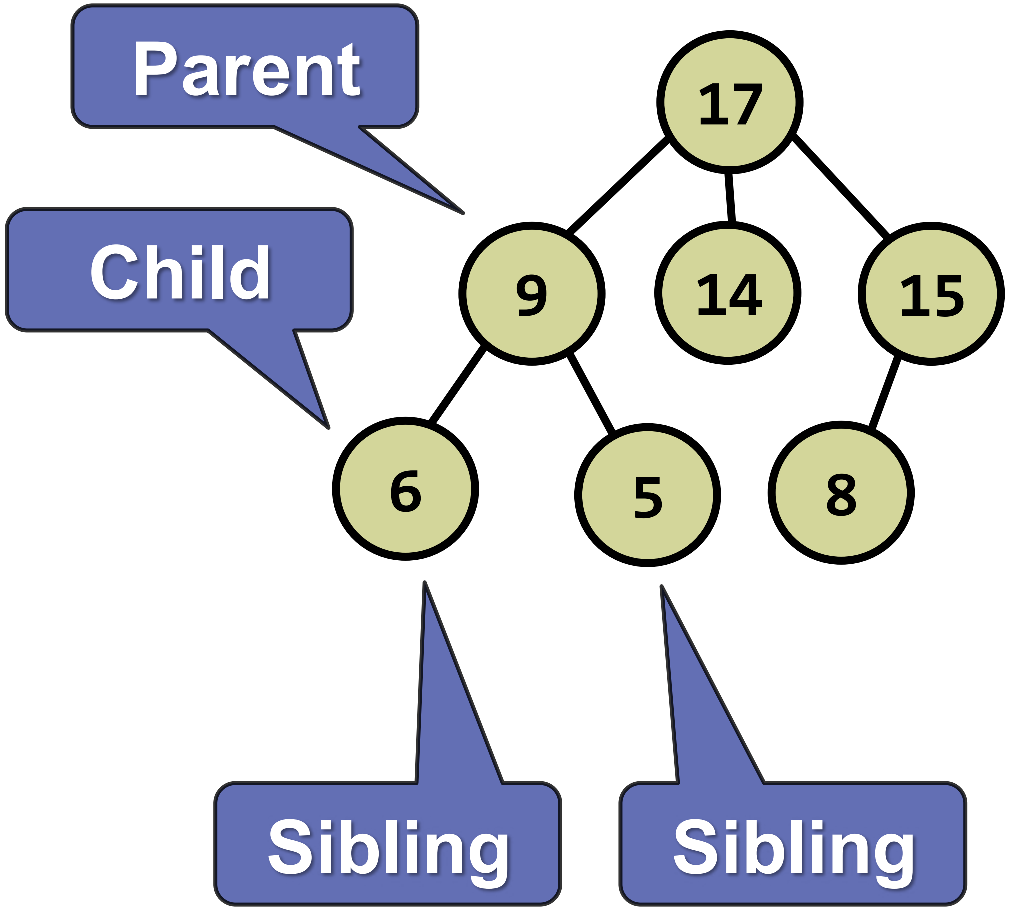
Tree Data Structure – Terminology

- **Node** – a structure which may contain a **value** or condition, or represent a separate **data structure**.
- **Edge** – the **connection between** one **node** and **another**.
- **Root** – the **top** node in a **tree**, the **prime ancestor**.



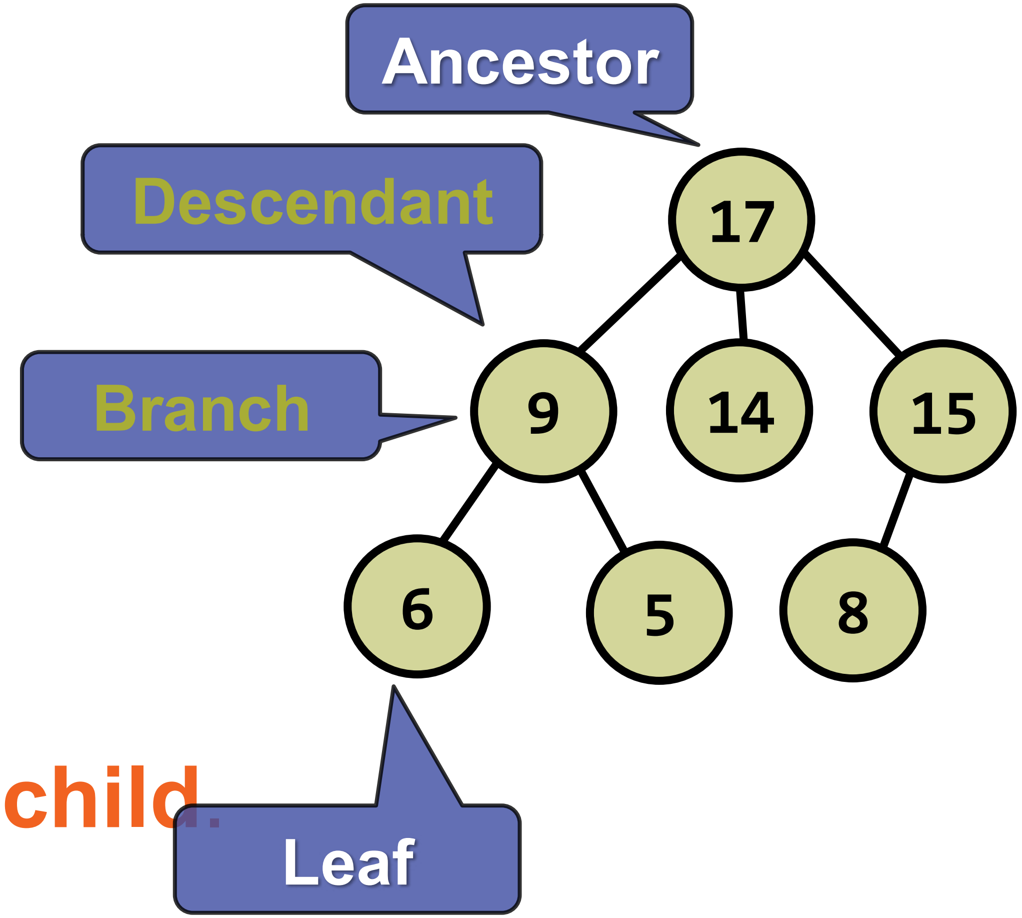
Tree Data Structure – Terminology

- **Parent** – the **converse** notion of a **child**, an **immediate ancestor**.
- **Child** – node **directly** connected to **another** node when moving **away** from the **root**, an immediate descendant.
- **Siblings** – a **group** of **nodes** with the **same parent**.



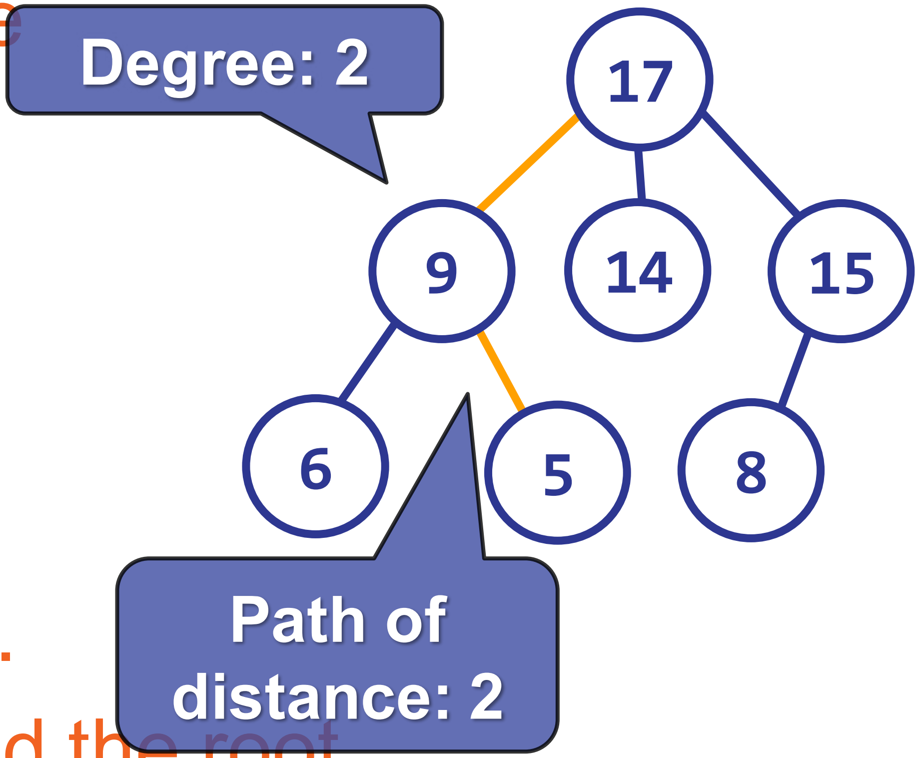
Tree Data Structure – Terminology

- **Ancestor** – node reachable by repeated proceeding **from child to parent**.
- **Descendant** – node reachable by repeated proceeding **from parent to child**.
- **Leaf** – node with **no children**.
- **Branch** – node with **at least one child**.



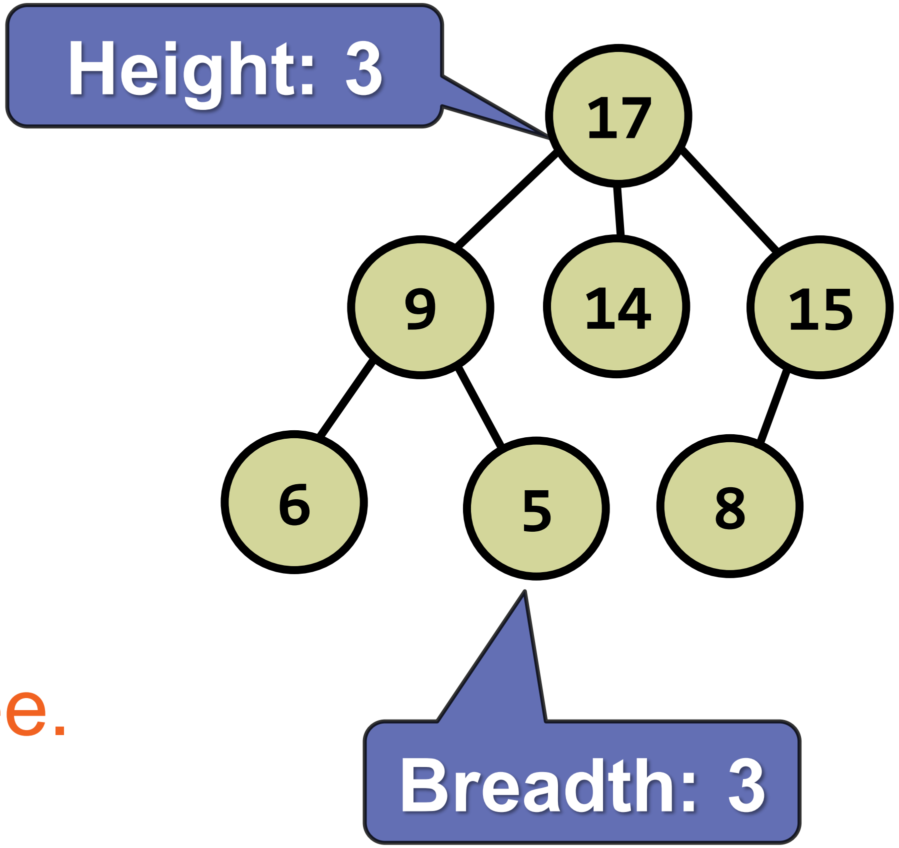
Tree Data Structure – Terminology

- **Degree** – number of children for node zero for a leaf.
- **Path** – sequence of nodes and edges connecting a node with a descendant.
- **Distance** – number of edges along the shortest path between two nodes.
- **Depth** – distance between a node and the root.



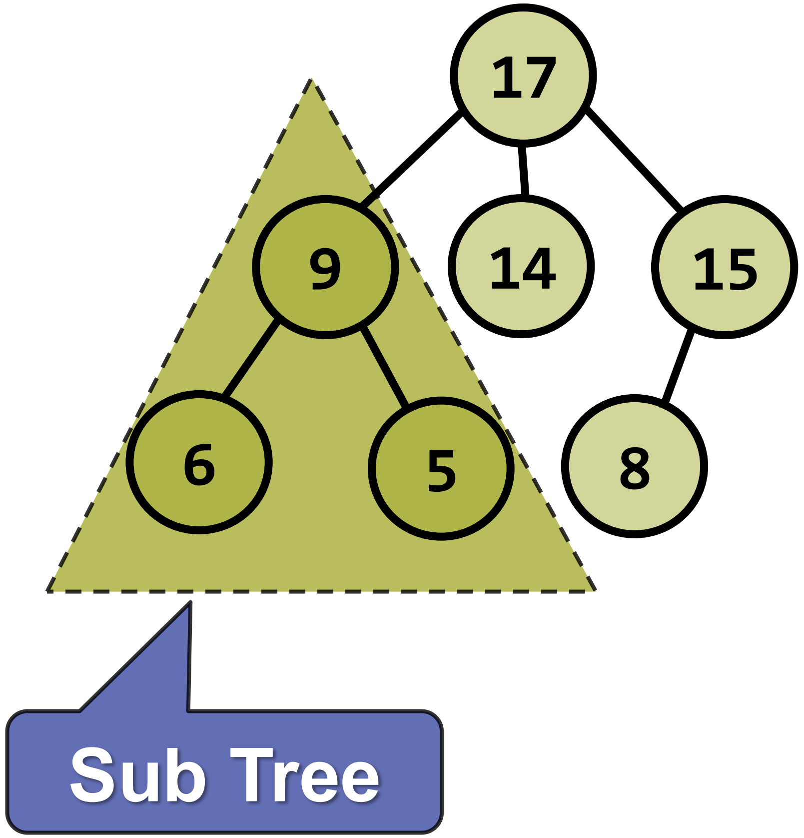
Tree Data Structure – Terminology

- **Level** – depth + 1.
- **Height** – The number of edges on the longest path between a node and a descendant leaf.
- **Width** – number of nodes in a level.
- **Breadth** – number of leaves.
- **Height** – the maximum level in the tree.

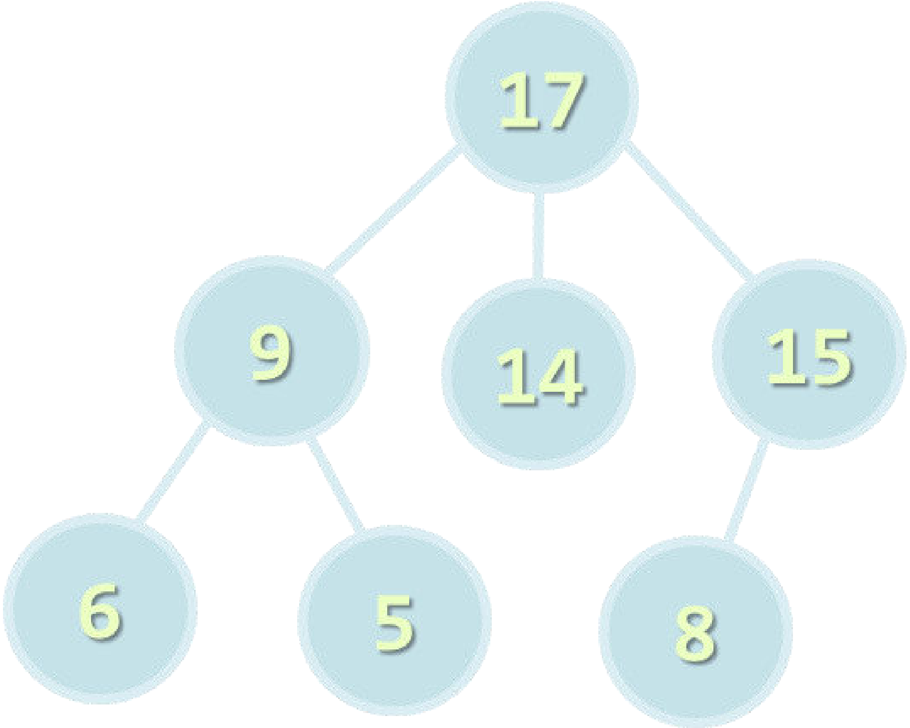


Tree Data Structure – Terminology

- **Forest** – set of disjoint trees.
  - {17}, {9, 6, 5}, {14}, {15, 8}
- **Sub Tree** – tree T is a tree consisting of a node in T and all of its descendants in T.



Implementing Trees



Recursive Tree Data Structure

Recursive Tree Definition

- The recursive definition for **tree** data structure:
  - A single node **is a tree**
  - Nodes have **zero or multiple children** that are **also trees**

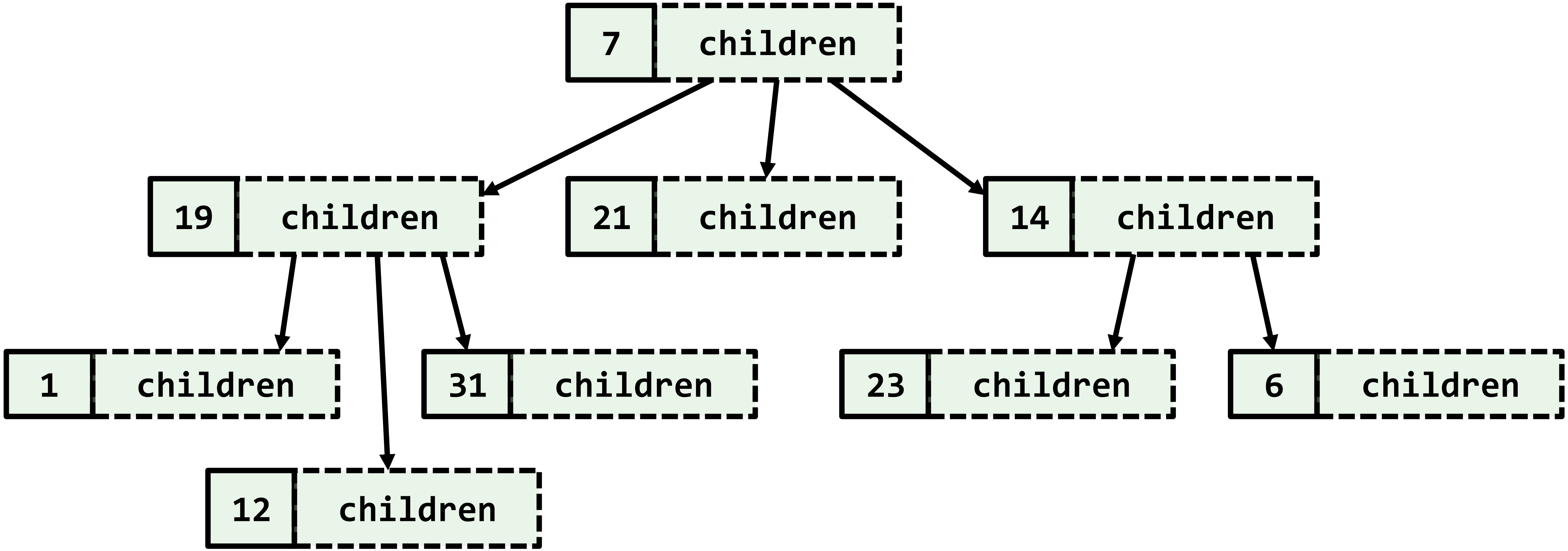
```
public class Tree<E> {
    private E key;
    private Tree<E> parent;
    private List<Tree<E>> children;
}
```

The stored key

The parent

List of child nodes

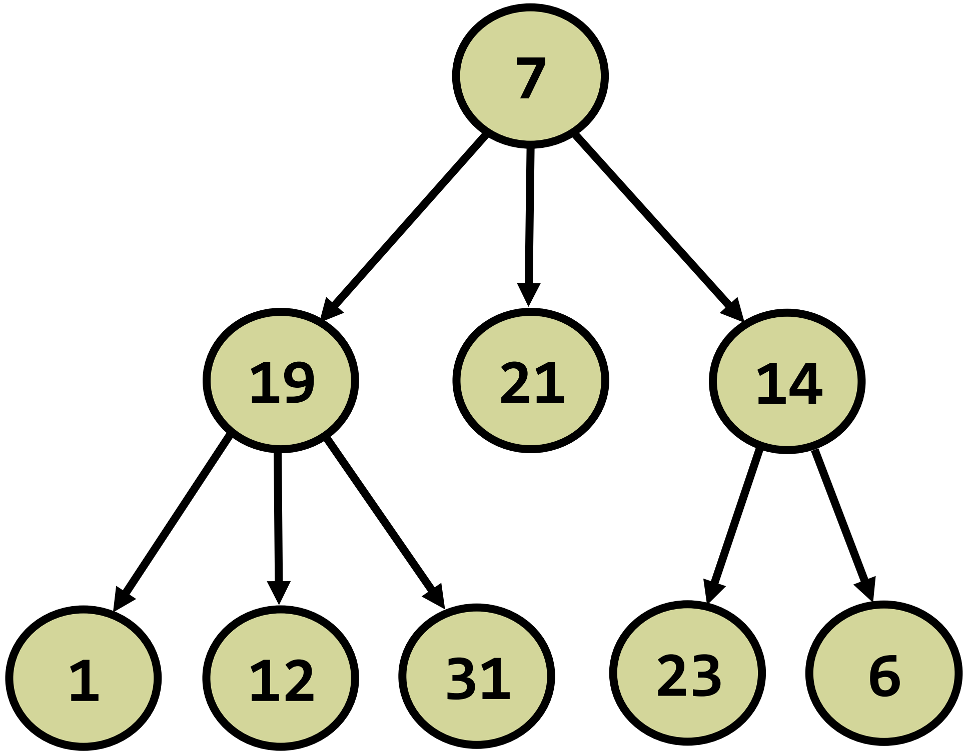
Tree<Integer> Structure – Example



Problem: Implement Tree Node

- Create a **recursive tree definition** in order to create trees

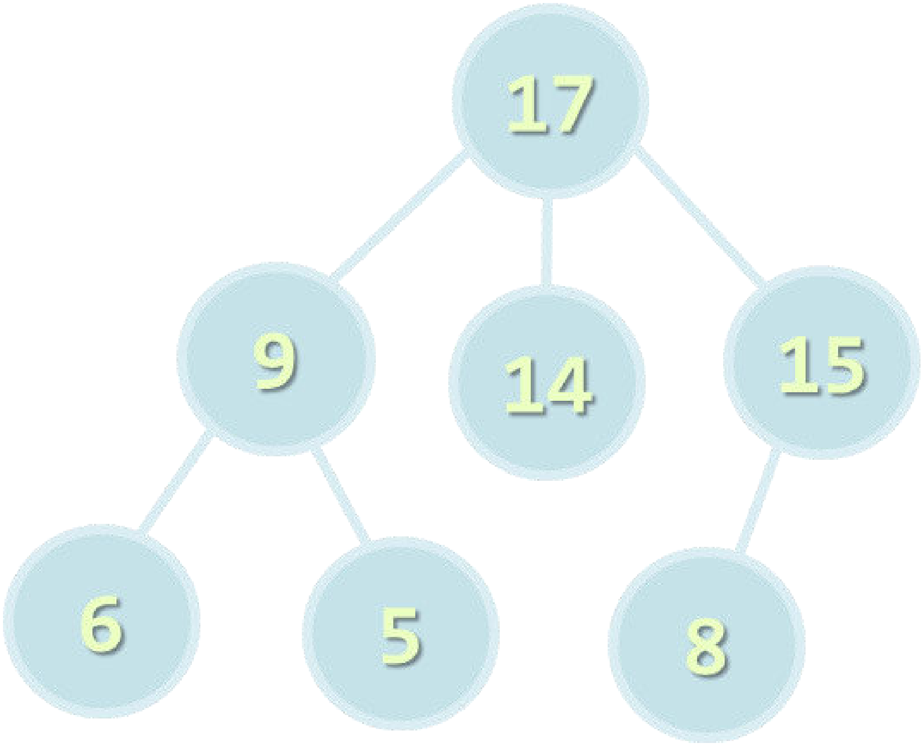
```
Tree<Integer> tree =
   new Tree<>(7,
      new Tree<>(19,
         new Tree<>(1),
         new Tree<>(12),
         new Tree<>(31)),
      new Tree<>(21),
      new Tree<>(14,
         new Tree<>(23),
         new Tree<Integer>(6))
);
```



Solution: Implement Tree

```
public class Tree<E> implements AbstractTree<E> {
    private E key;
    private Tree<E> parent;
    private List<Tree<E>> children;
    public Tree(E key, Tree<E>... children) {
        this.key = key;
        this.children = new ArrayList<>();
        for (Tree<E> child : children) {
            this.children.add(child);
            child.parent = this;
        }
    }
}
```

Traversing Tree-Like Structures



DFS and BFS Traversals

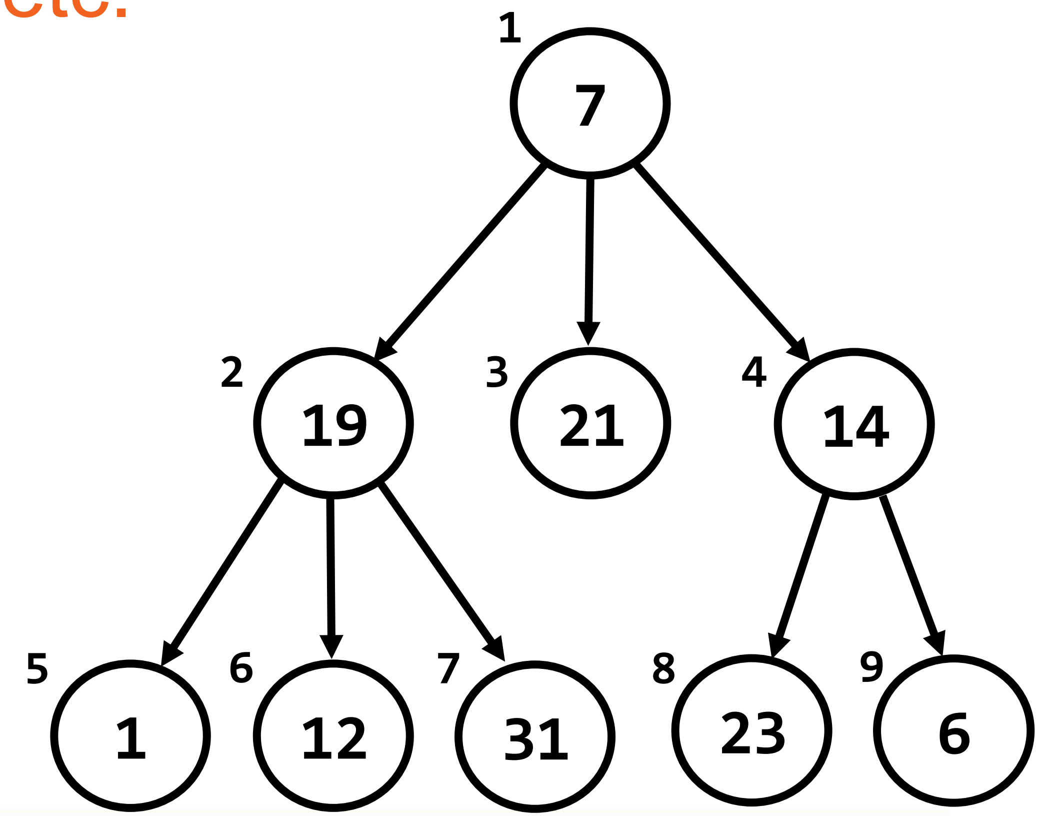
Tree Traversal Algorithms

- **Traversing a tree** means to visit each of its nodes exactly once
  - The **order of visiting nodes** may vary on the traversal algorithm
  - **Depth-First Search** (DFS)
    - Visit node's successors first
    - Usually implemented by recursion
  - **Breadth-First Search** (BFS)
    - Nearest nodes visited first
    - Implemented by a queue

Breadth-First Search (BFS)

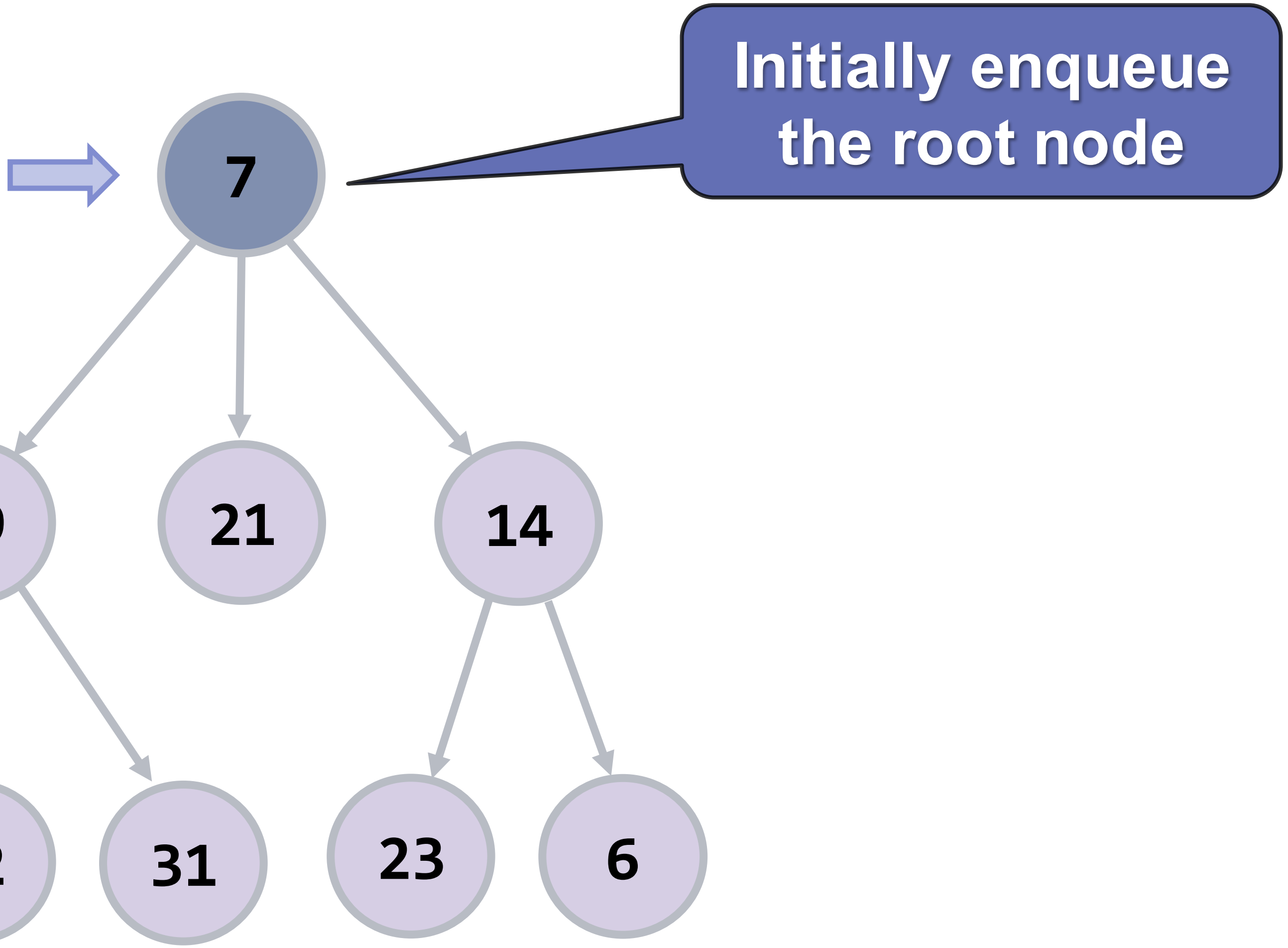
- **Breadth-First Search** (BFS) first visits the neighbor nodes, then the neighbors of neighbors, etc.
- BFS algorithm pseudo code:

```
BFS (node) {
  queue ← node
  while queue not empty
    v ← queue
    print v
    for each child c of v
      queue ← c
}
```



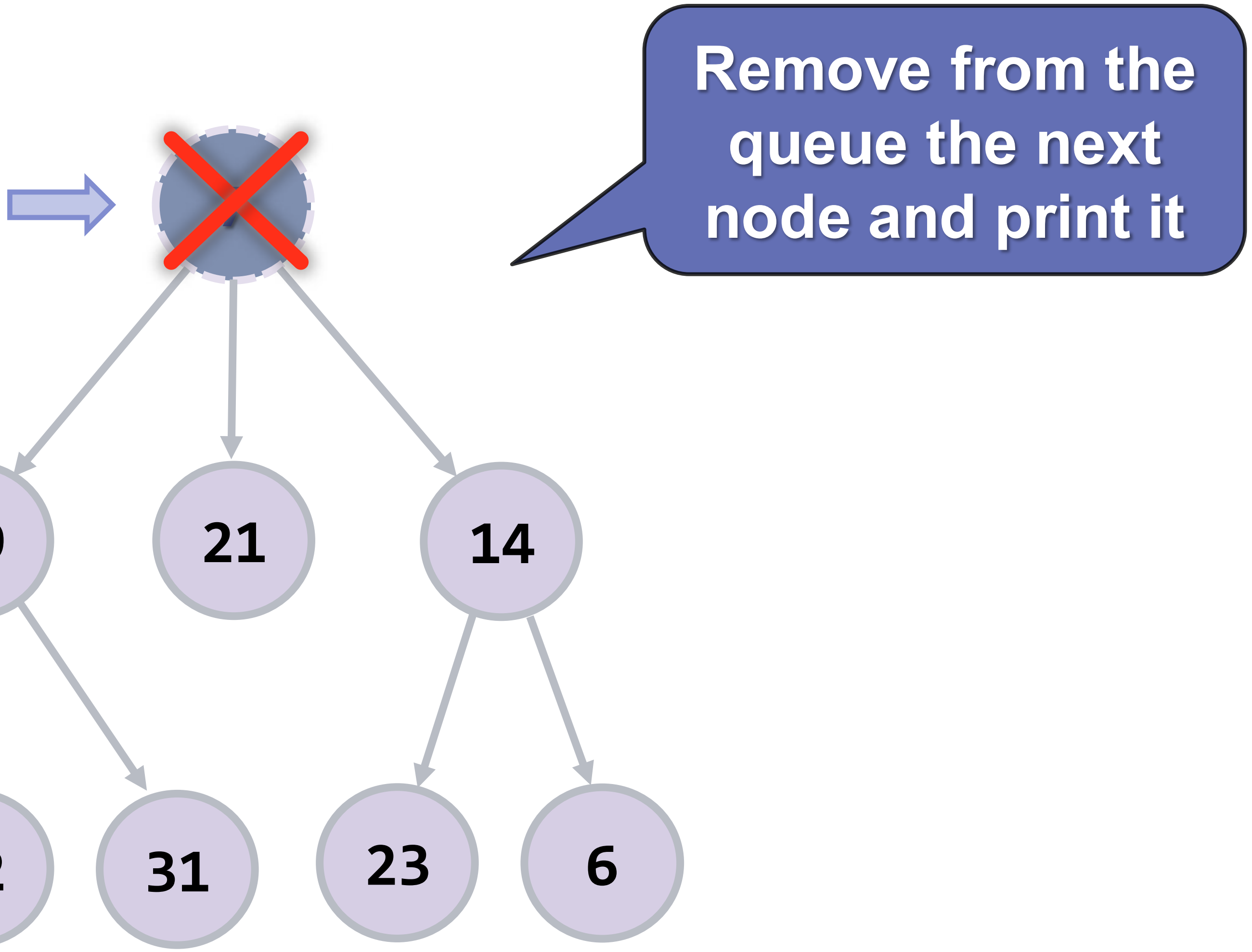
BFS in Action (Step 1)

- Queue: 7
- Output:



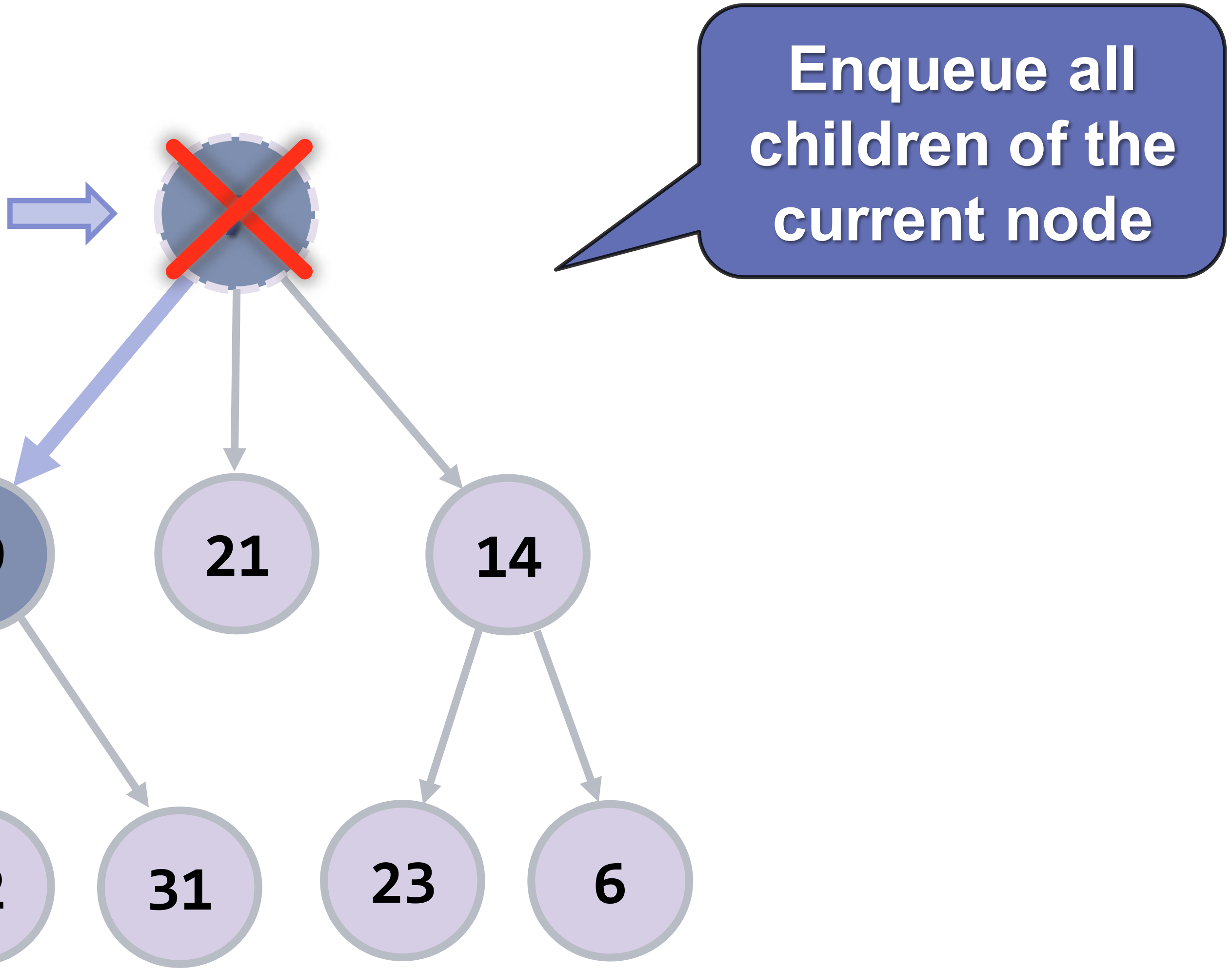
BFS in Action (Step 2)

- Queue: ~~7~~
- Output: 7



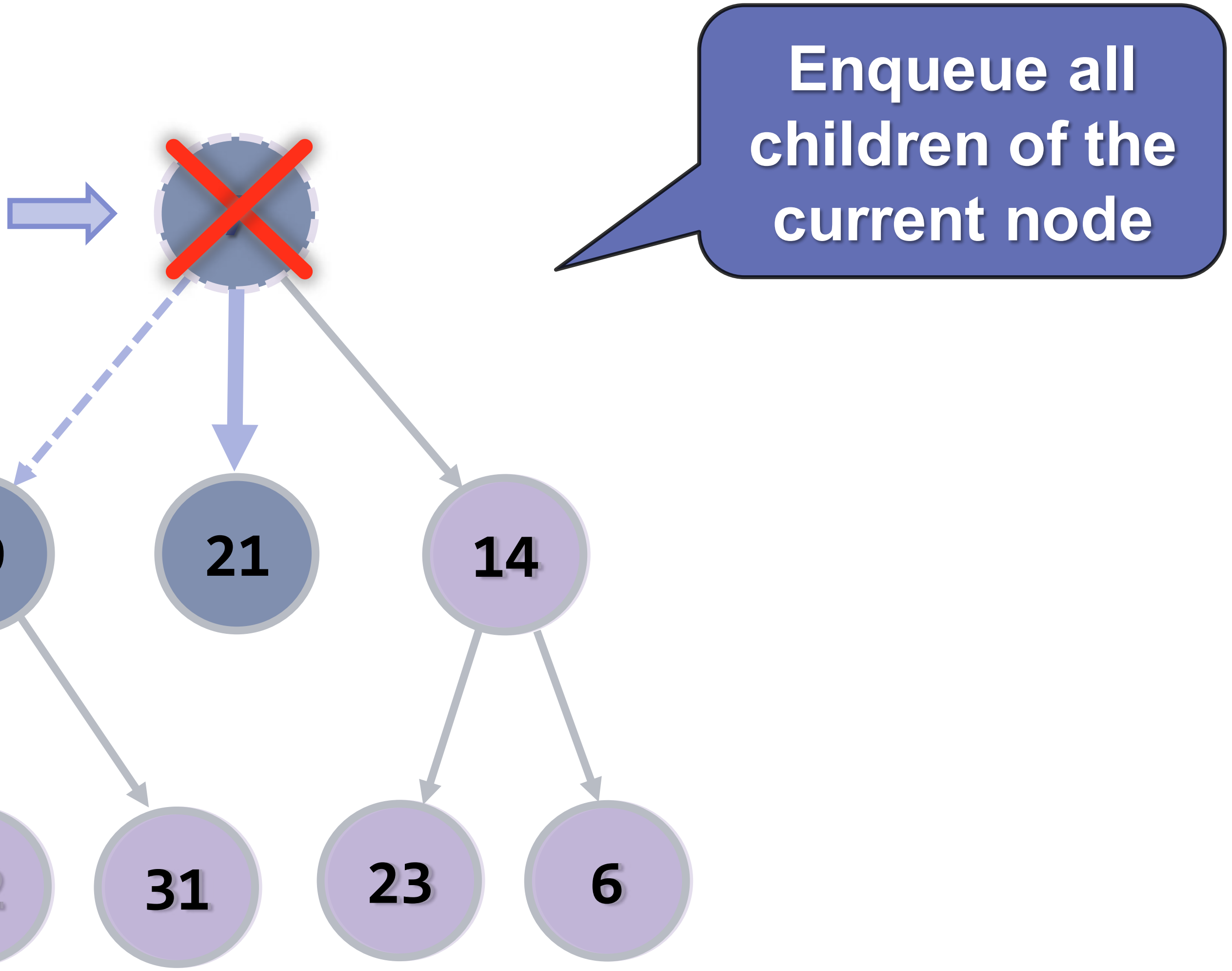
BFS in Action (Step 3)

- Queue: ~~7~~, 19
- Output: 7



BFS in Action (Step 4)

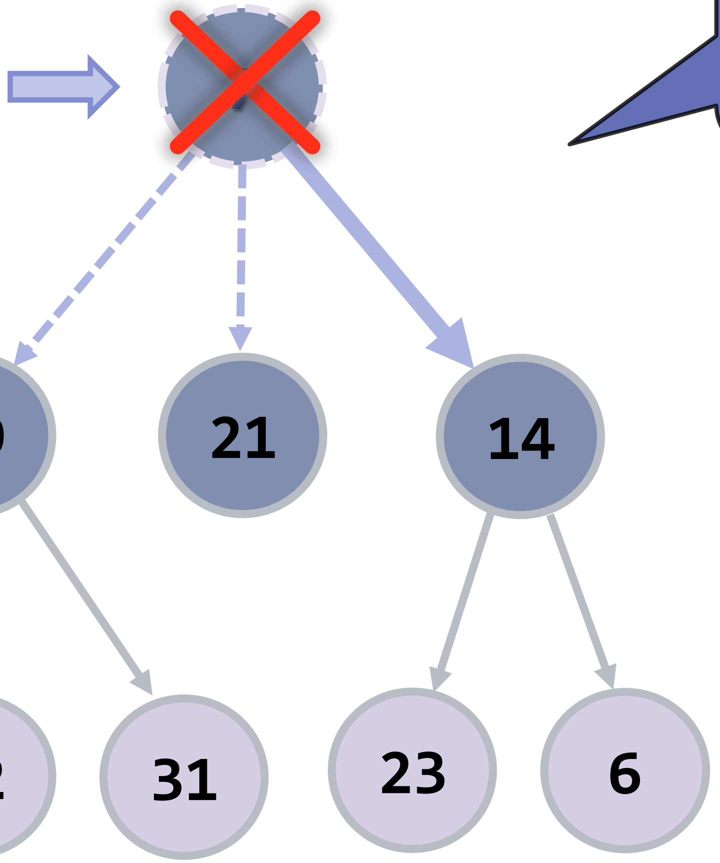
- Queue: ~~7~~, 19, 21
- Output: 7



BFS in Action (Step 5)

- Queue: ~~7~~, 19, 21, 14
- Output: 7

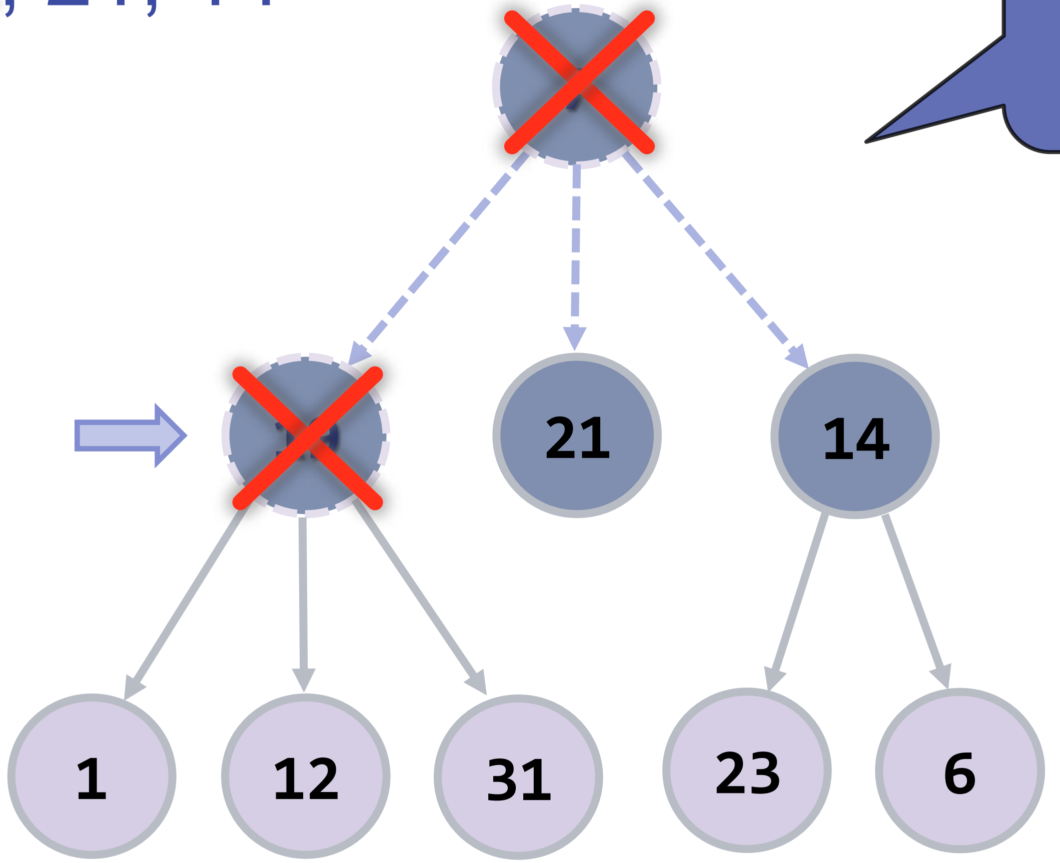
Enqueue all children of the current node



BFS in Action (Step 6)

- Queue: ~~7~~, ~~19~~, 21, 14
- Output: 7, 19

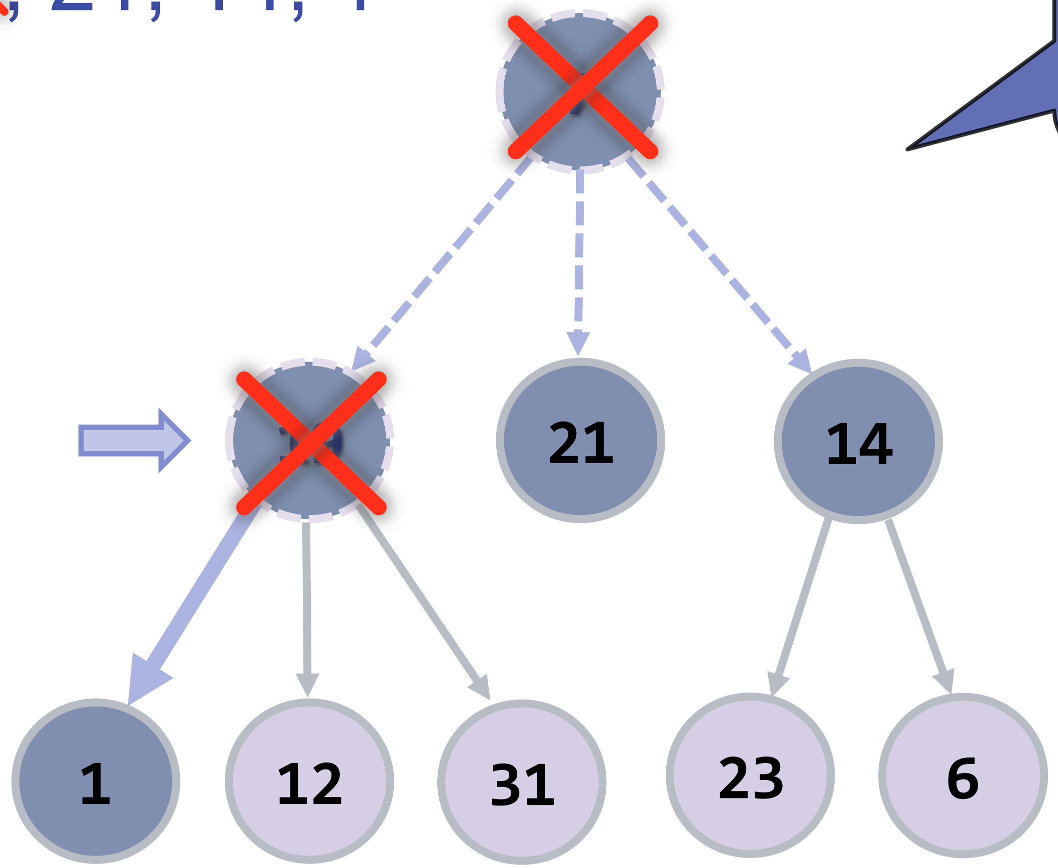
Remove from the queue the next node and print it



BFS in Action (Step 7)

- Queue: ~~7~~, ~~19~~, 21, 14, 1
- Output: 7, 19

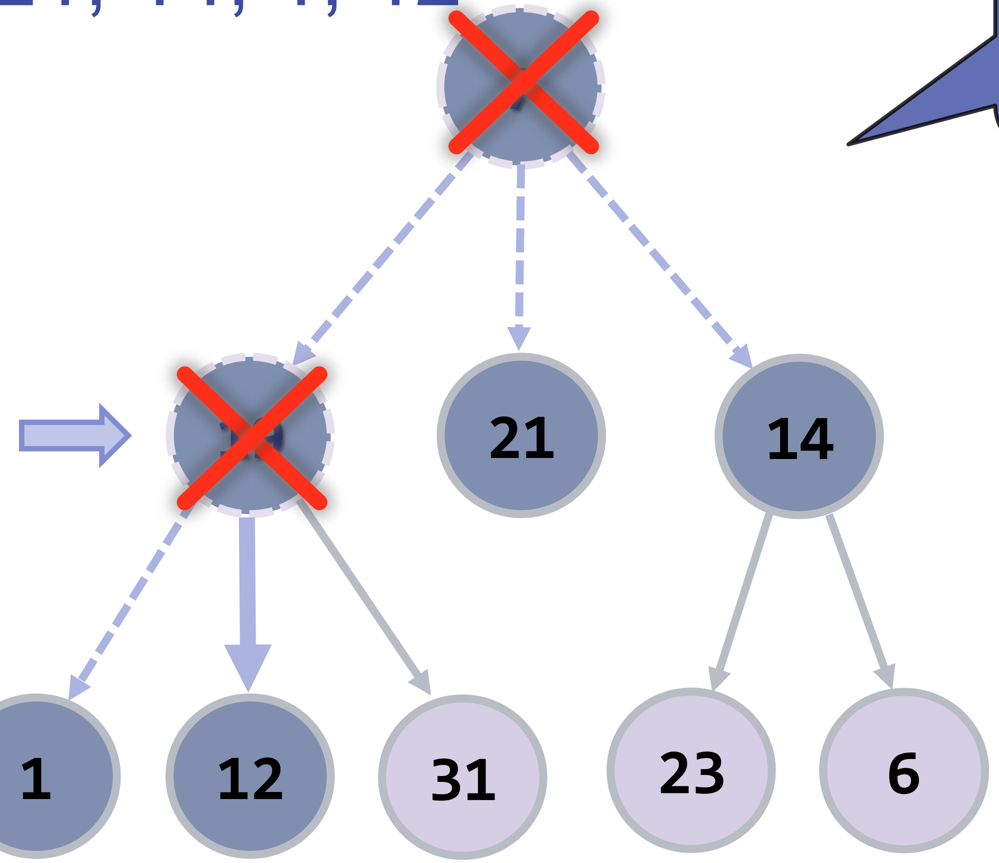
Enqueue all children of the current node



BFS in Action (Step 8)

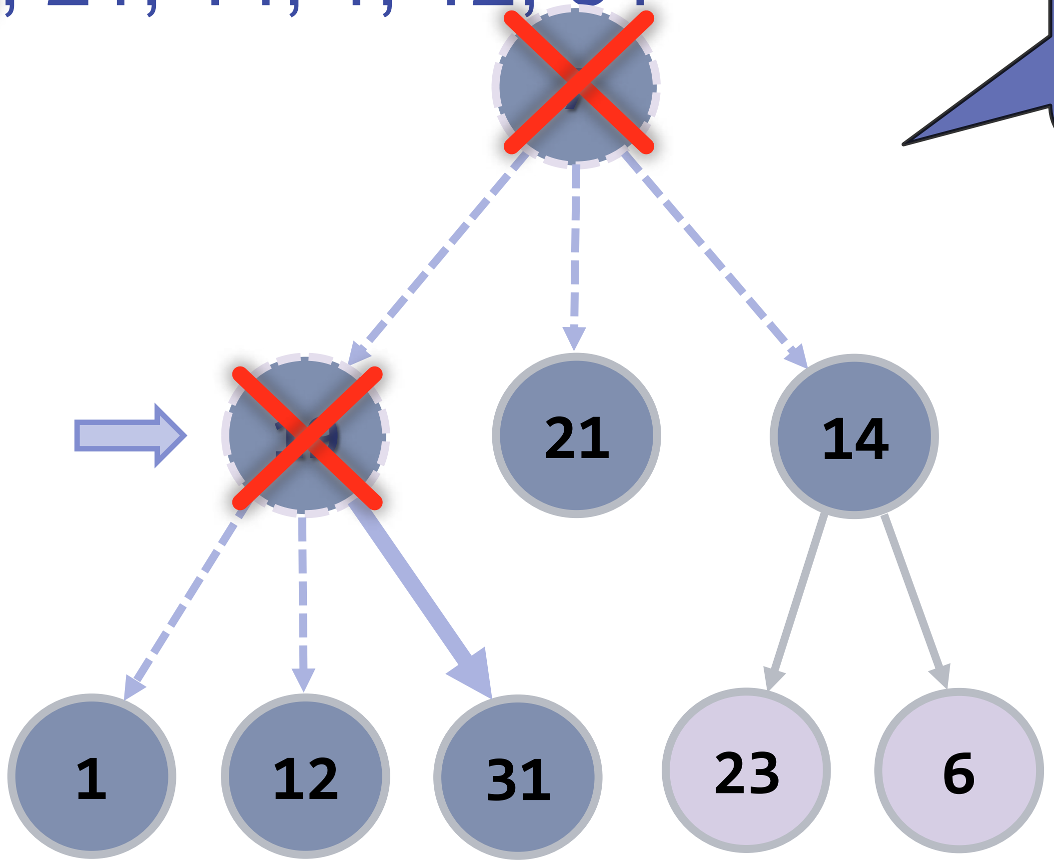
- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12
- Output: 7, 19

Enqueue all children of the current node



BFS in Action (Step 9)

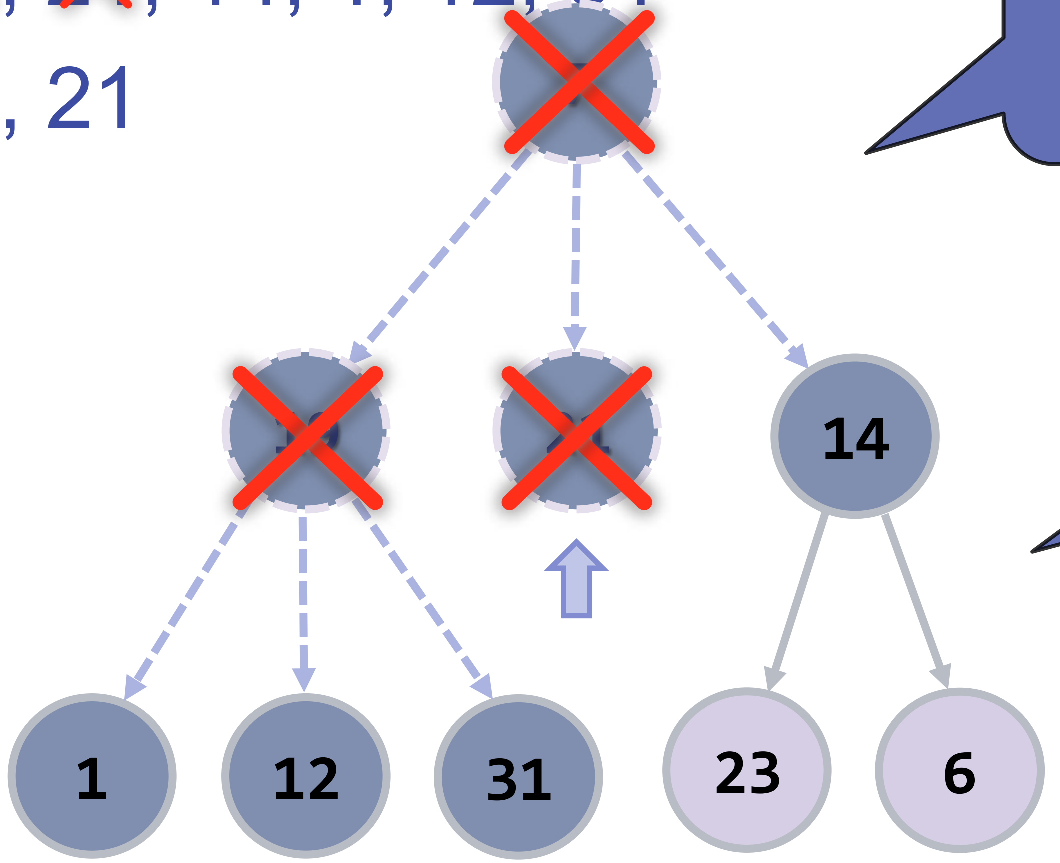
- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12, 31
- Output: 7, 19



Enqueue all children of the current node

BFS in Action (Step 10)

- Queue: ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31
- Output: 7, 19, 21



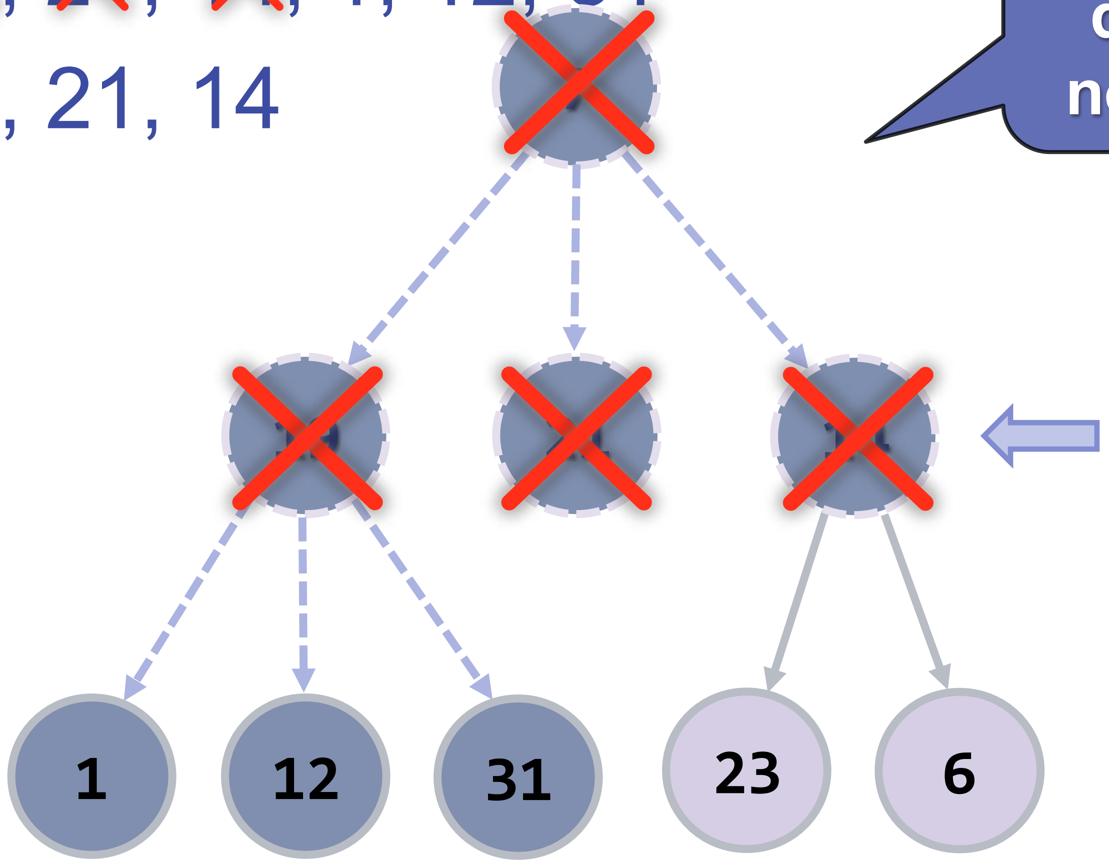
Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 11)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31
- Output: 7, 19, 21, 14

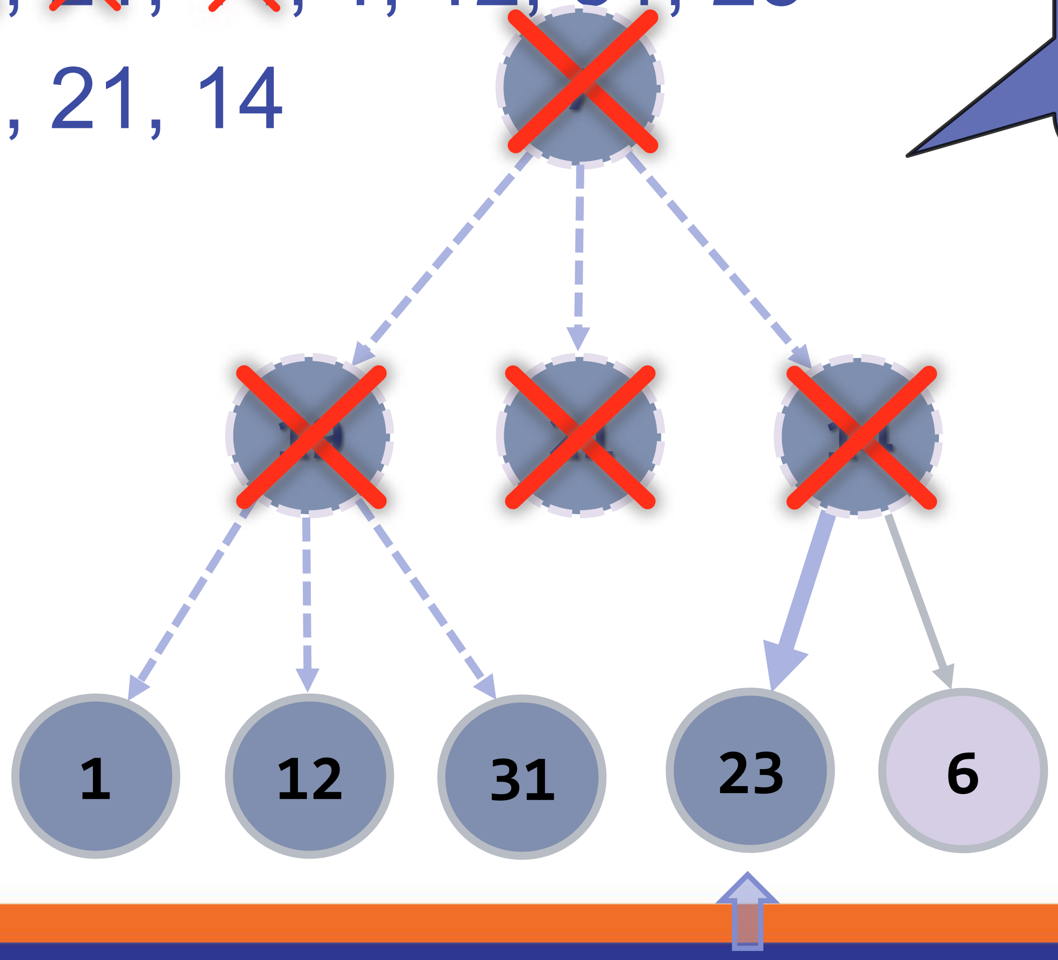
Remove from the queue the next node and print it



BFS in Action (Step 12)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31, 23
- Output: 7, 19, 21, 14

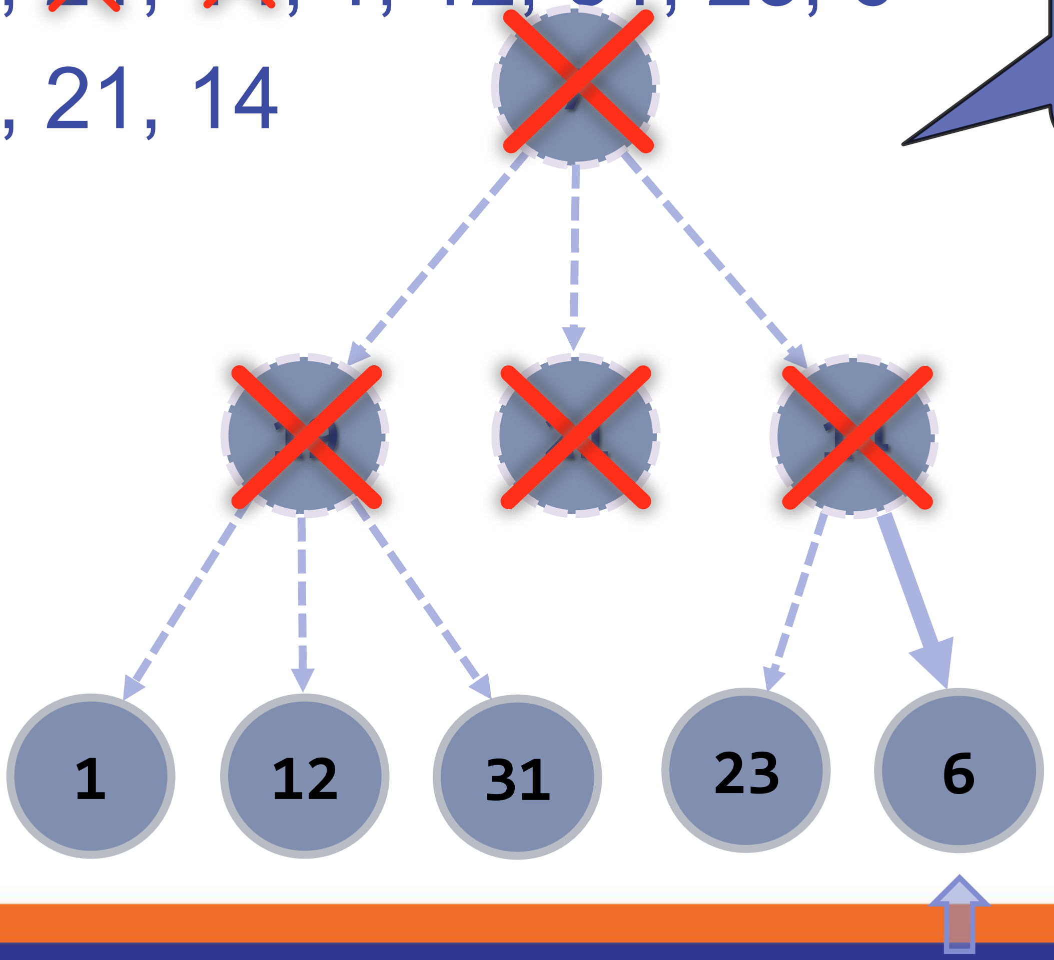
Enqueue all children of the current node



BFS in Action (Step 13)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31, 23, 6
- Output: 7, 19, 21, 14

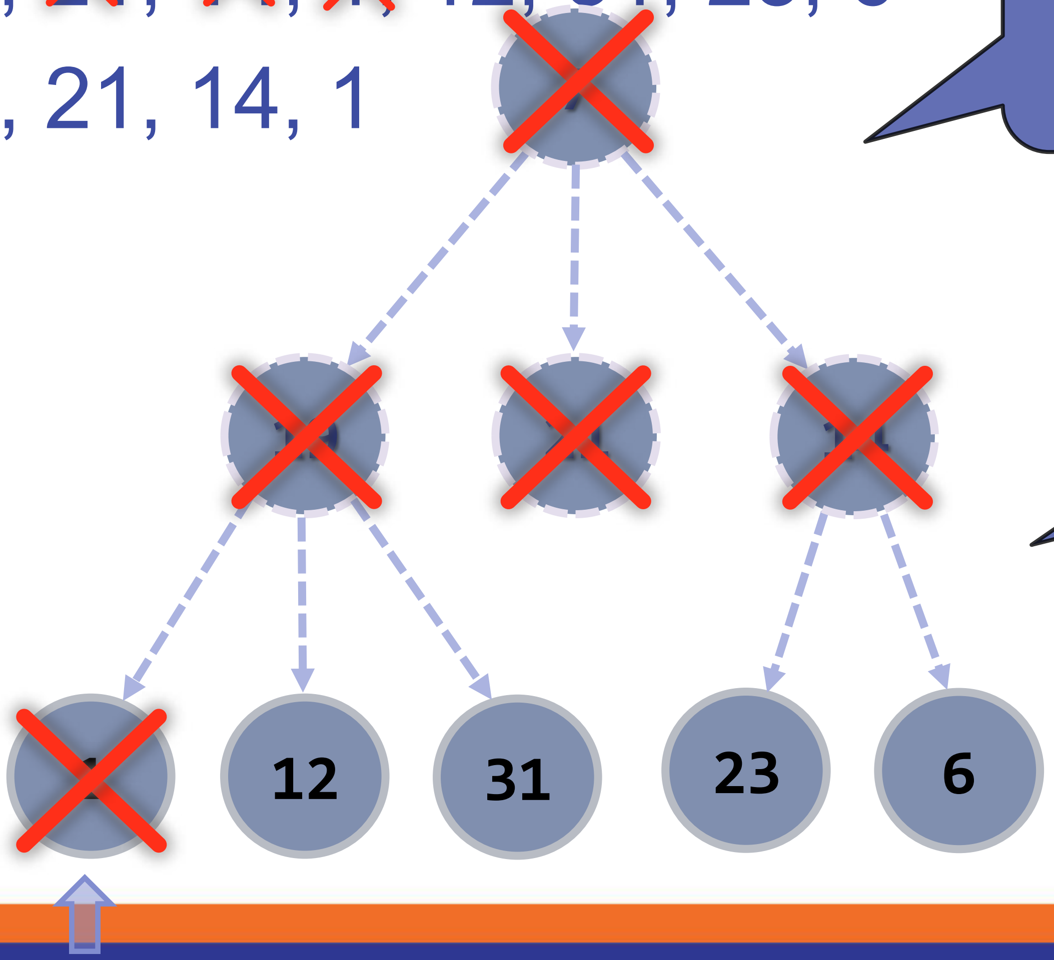
Enqueue all children of the current node



BFS in Action (Step 14)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, 12, 31, 23, 6
- Output: 7, 19, 21, 14, 1

Remove from the queue the next node and print it

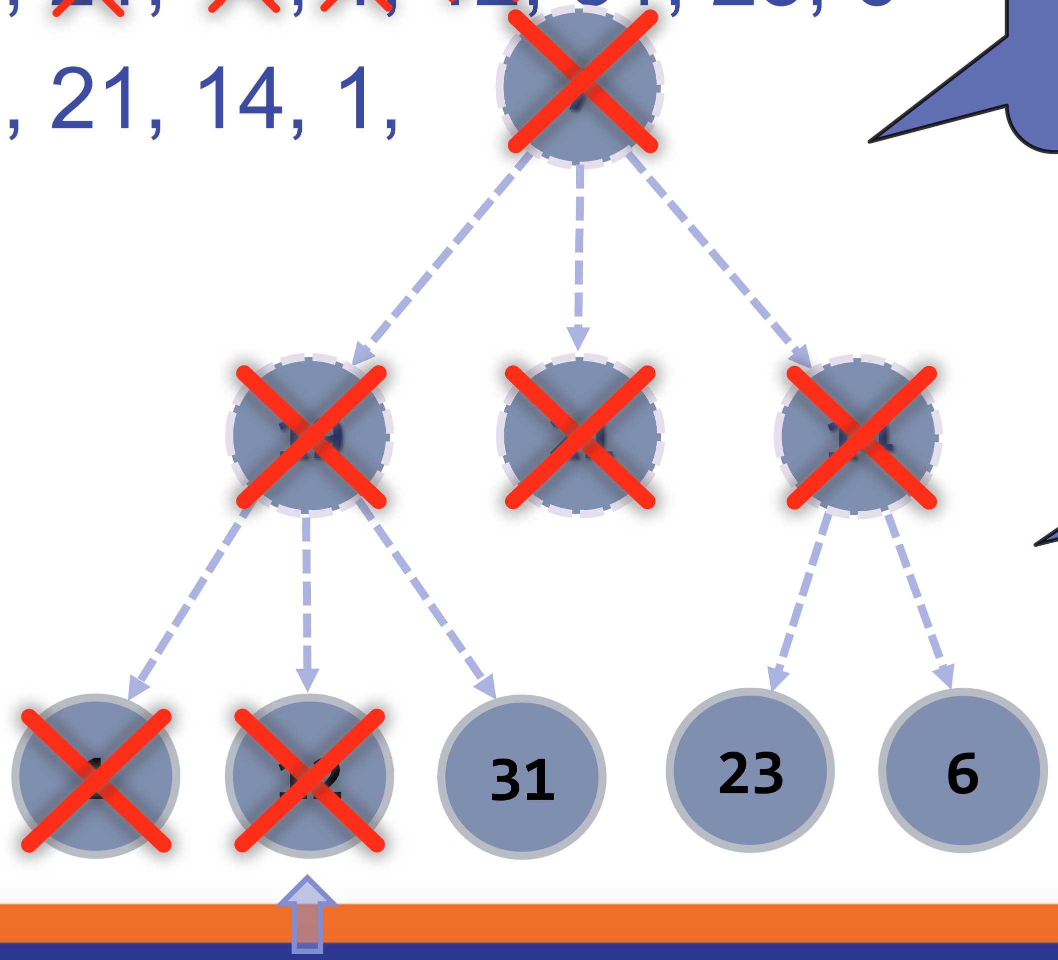


No child nodes to enqueue

BFS in Action (Step 15)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, 31, 23, 6
- Output: 7, 19, 21, 14, 1, 12

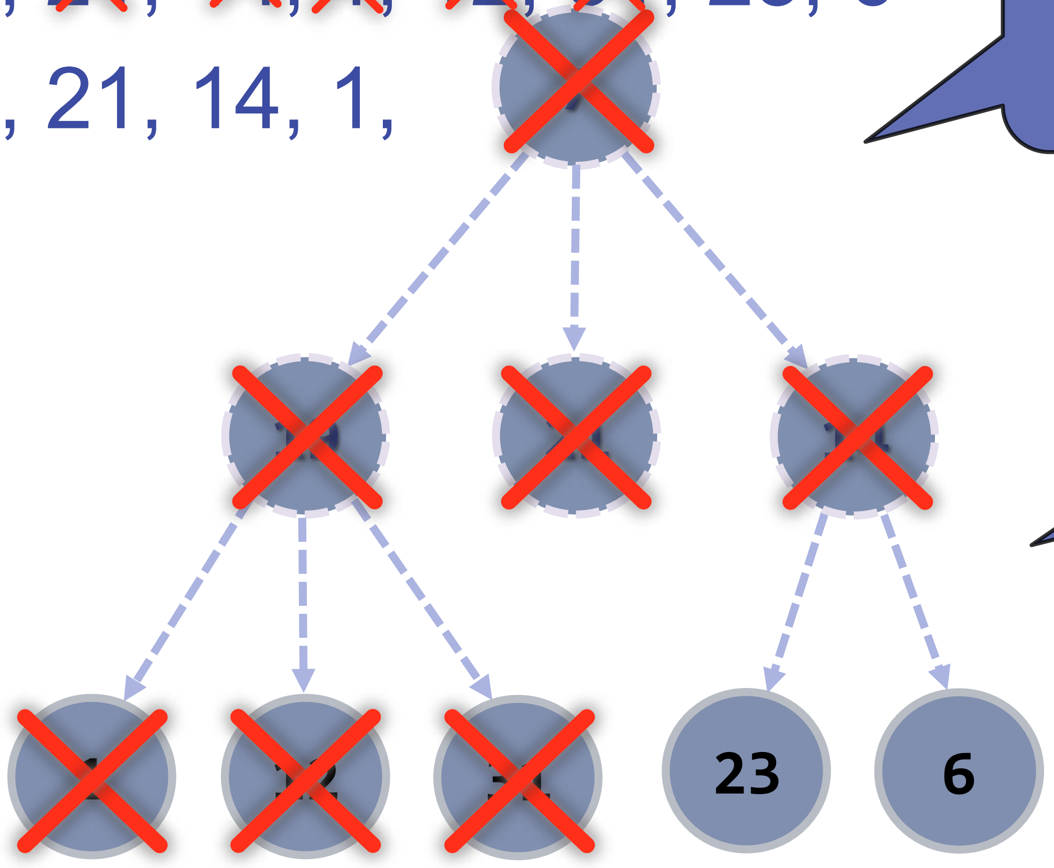
Remove from the queue the next node and print it



No child nodes to enqueue

BFS in Action (Step 16)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, 23, 6
- Output: 7, 19, 21, 14, 1, 12, 31



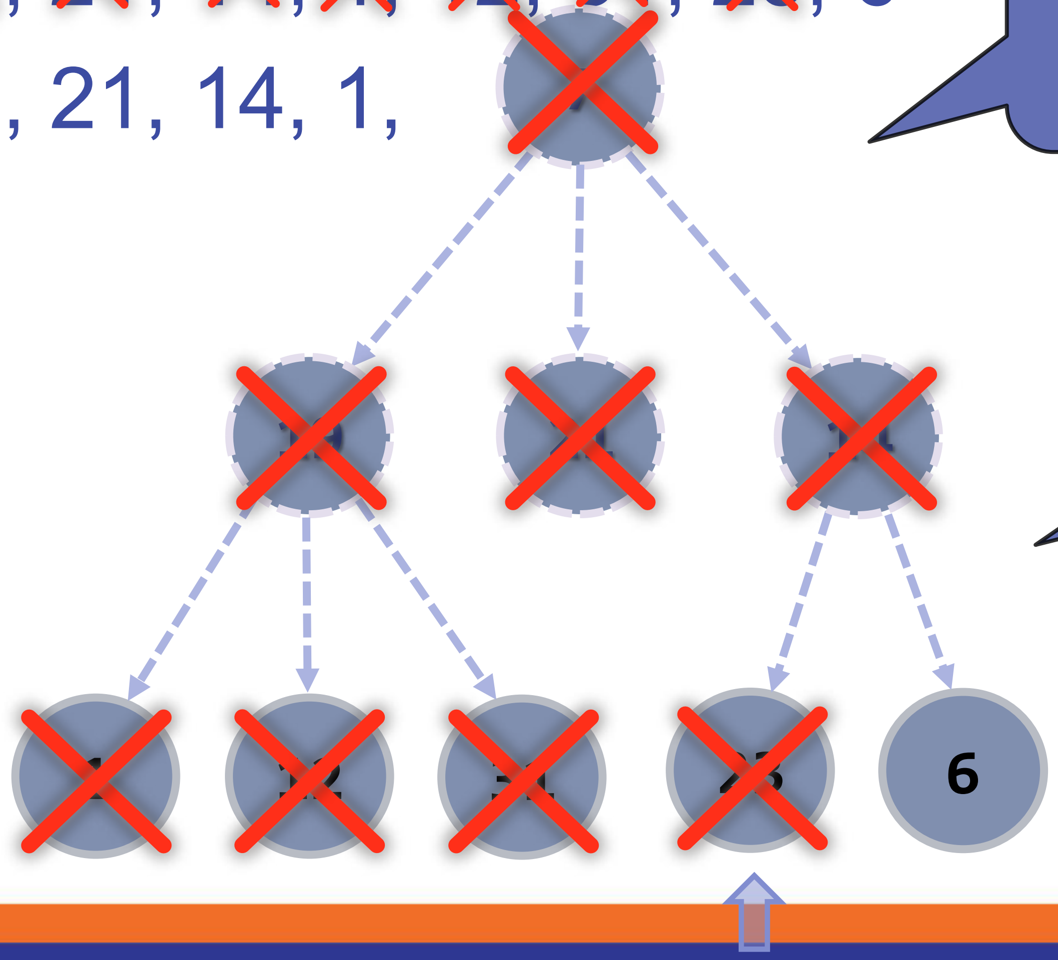
Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 17)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, 6
- Output: 7, 19, 21, 14, 1, 12, 31, 23

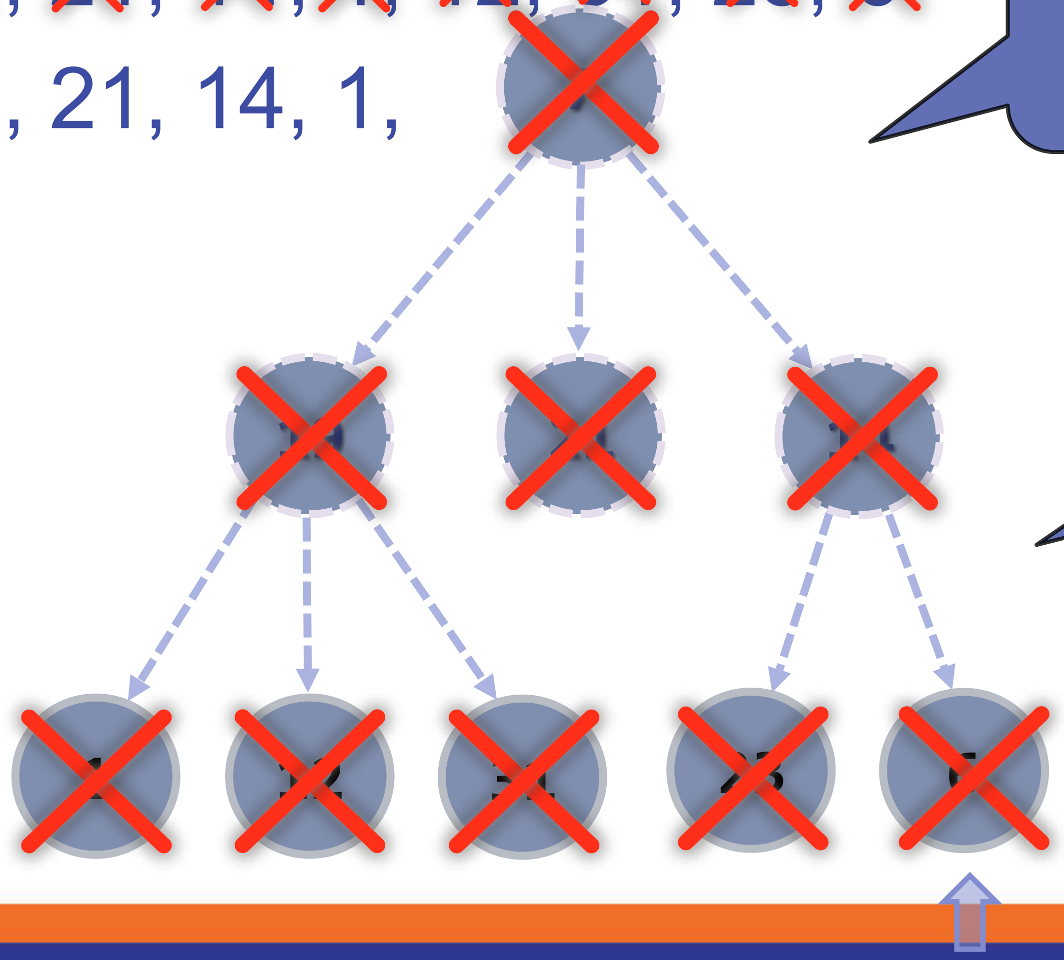
Remove from the queue the next node and print it



No child nodes to enqueue

BFS in Action (Step 18)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~
- Output: 7, 19, 21, 14, 1, 12, 31, 23, 6

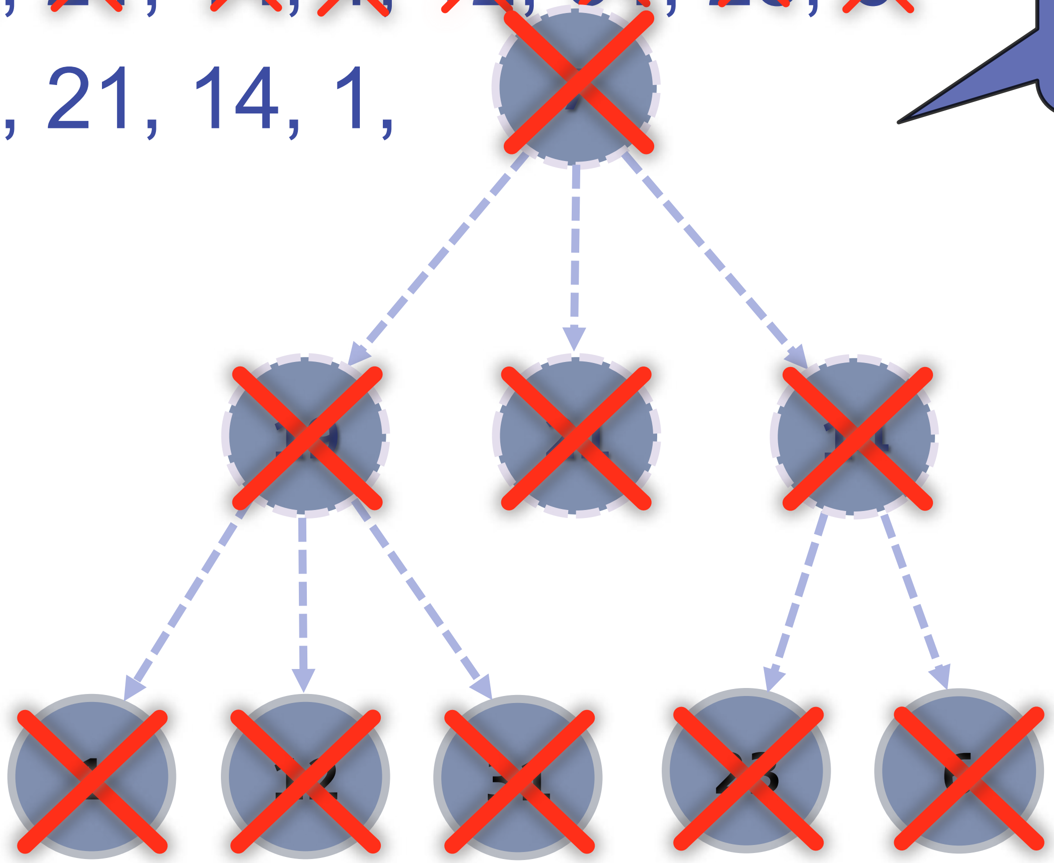


Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 19)

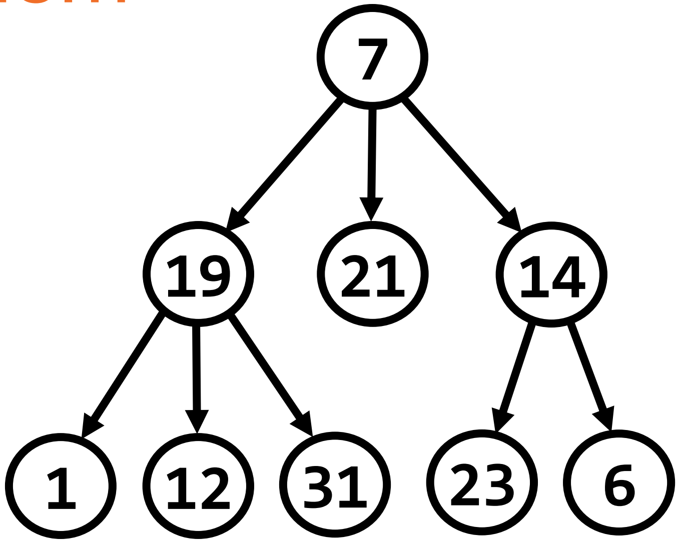
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~
- Output: 7, 19, 21, 14, 1, 12, 31, 23, 6



The queue is empty → stop

Problem: Order BFS

- Given the **Tree<E>** structure, define a method
  - **List<E> orderBfs()**
- That returns elements in order of BFS algorithm visiting them



7 19 21 14 1 12 31 23 6

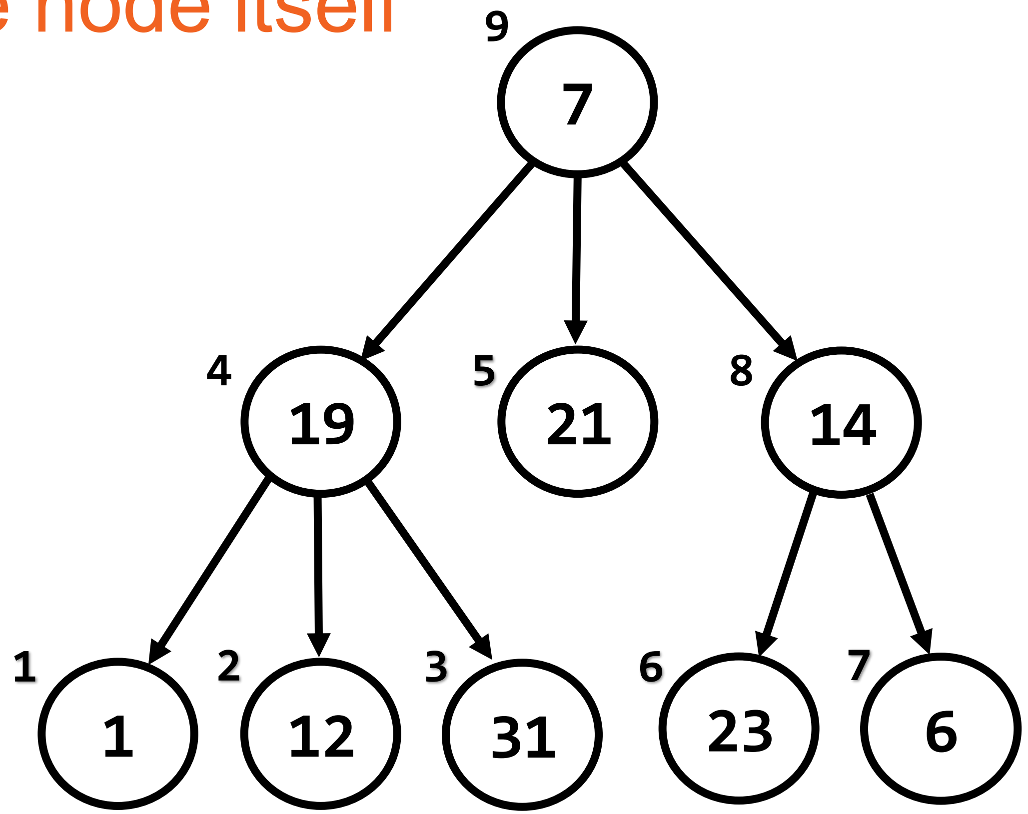
Solution: Order DFS

```
public List<E> orderBfs() {
  List<E> result = new ArrayList<>();
  Deque<Tree<E>> queue = new ArrayDeque<>();
  queue.offer(this);
  while (queue.size() > 0) {
    Tree<E> current = queue.poll();
    result.add(current.key);
    for (Tree<E> child : current.children)
      queue.offer(child);
  }
  return result;
}
```

Depth-First Search (DFS)

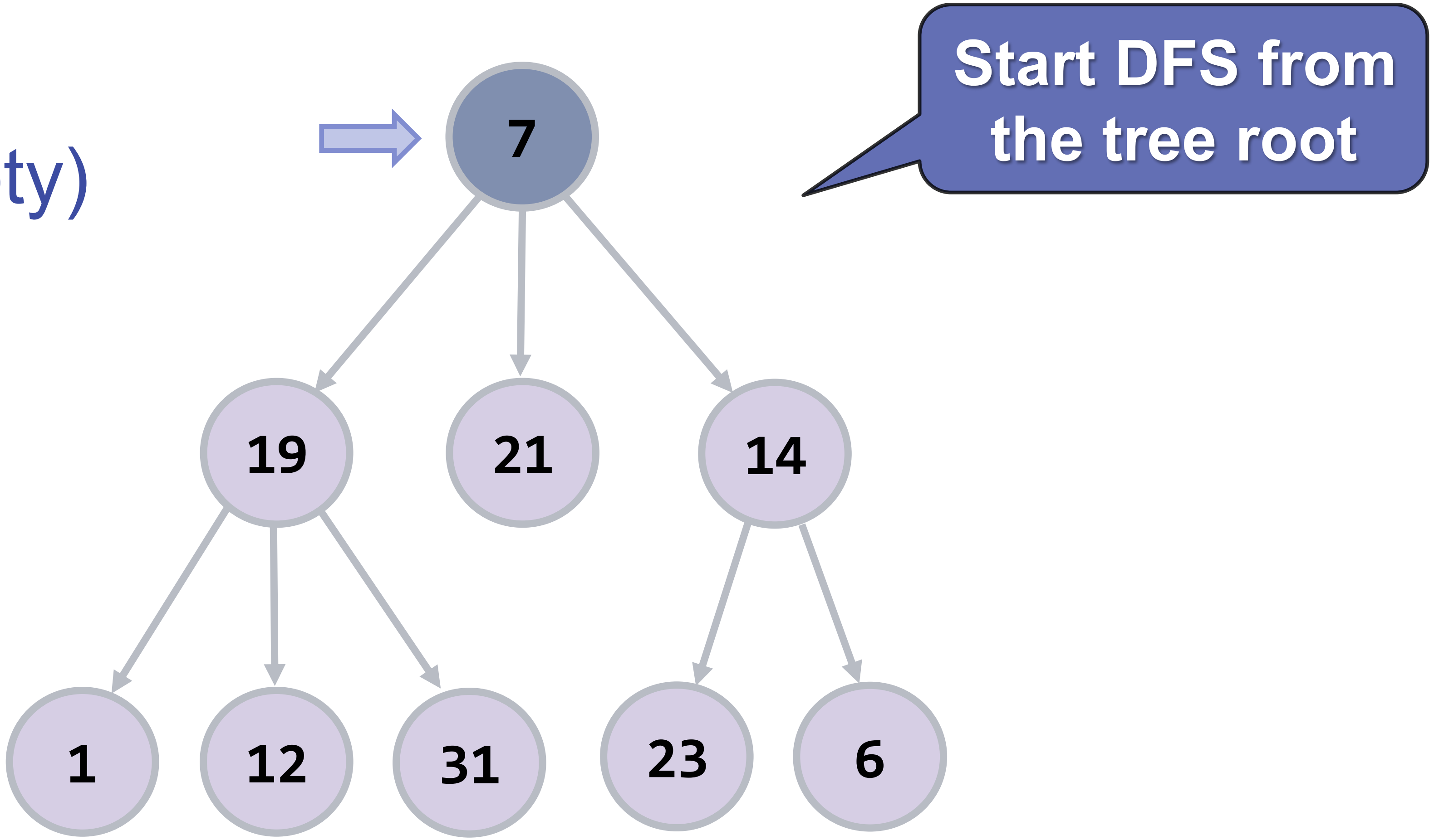
- **Depth-First Search** (**DFS**) first visits all descendants of given node recursively, finally visits the node itself
- DFS algorithm pseudo code:

```
DFS (node) {
  for each child c of node
    DFS(c);
  print node;
}
```



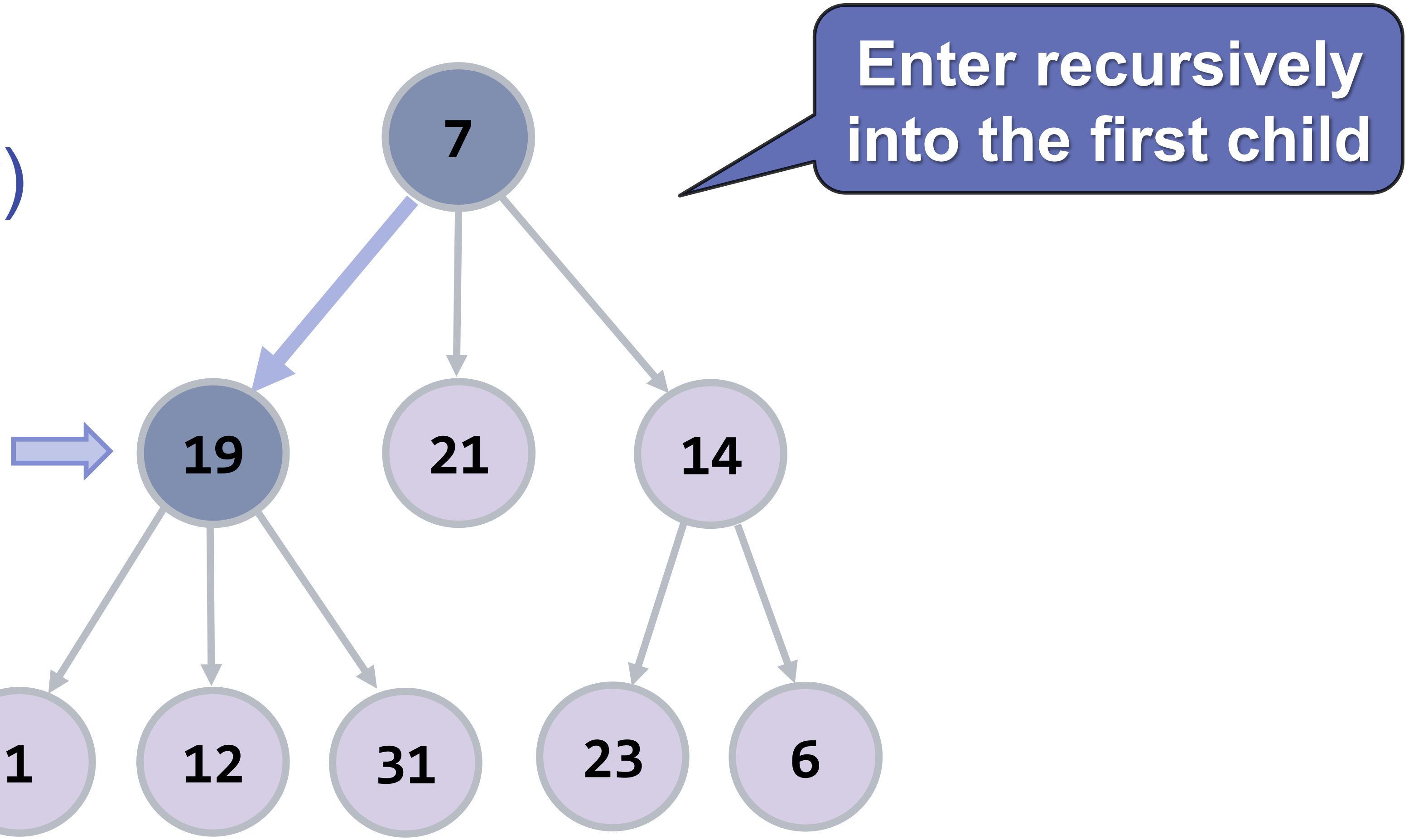
DFS in Action (Step 1)

- Stack: 7
- Output: (empty)



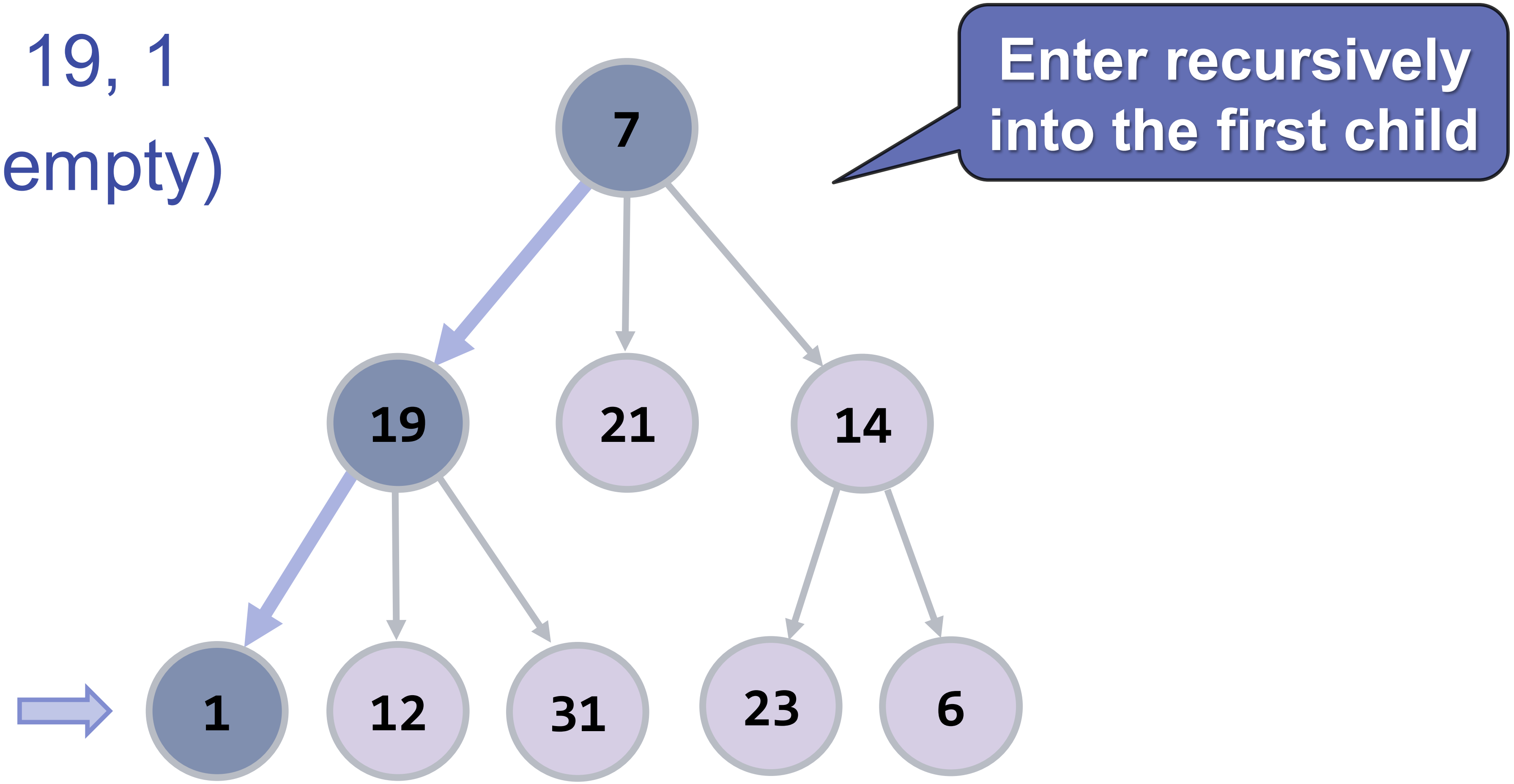
DFS in Action (Step 2)

- Stack: 7, 19
- Output: (empty)



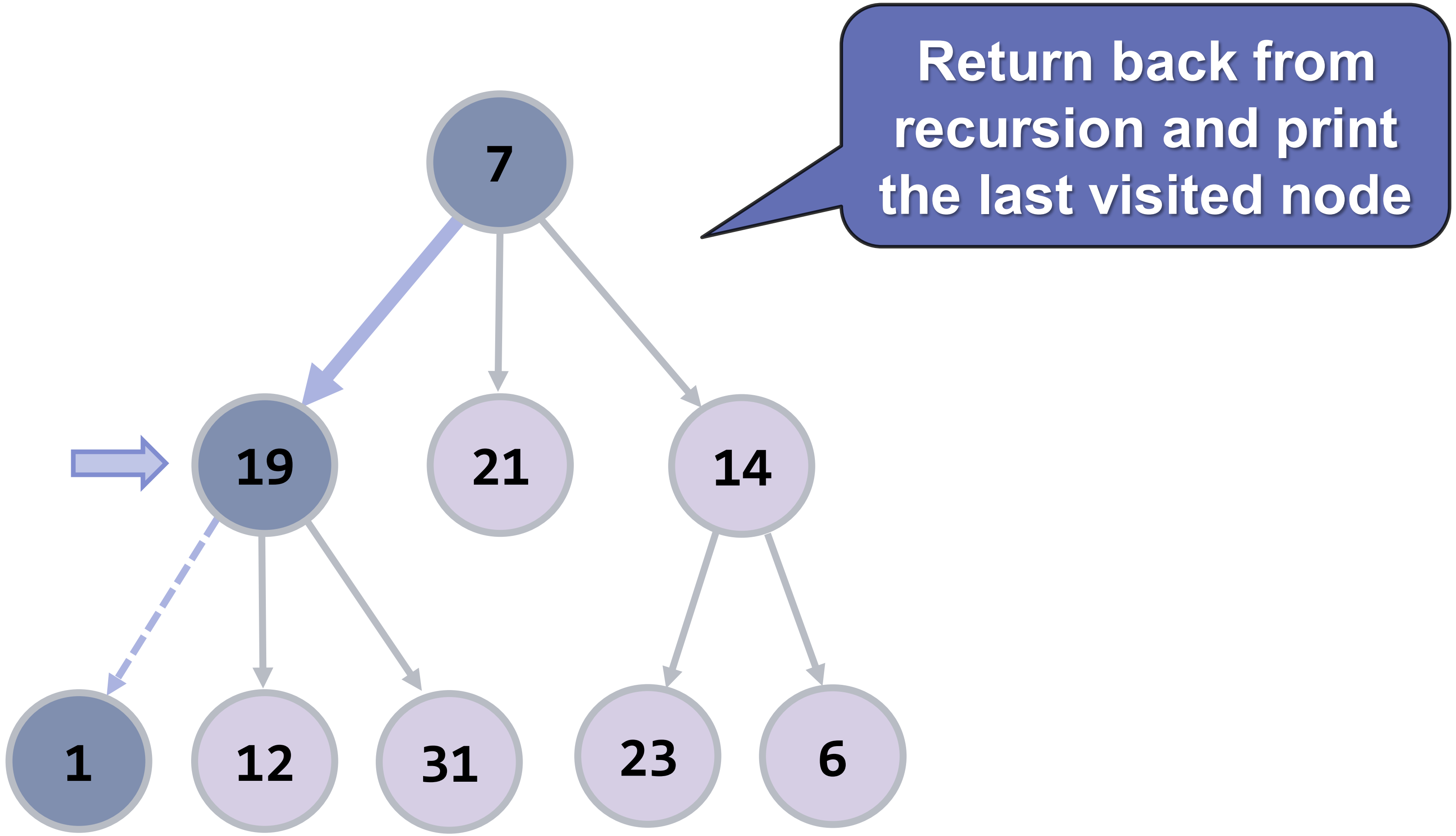
DFS in Action (Step 3)

- Stack: 7, 19, 1
- Output: (empty)



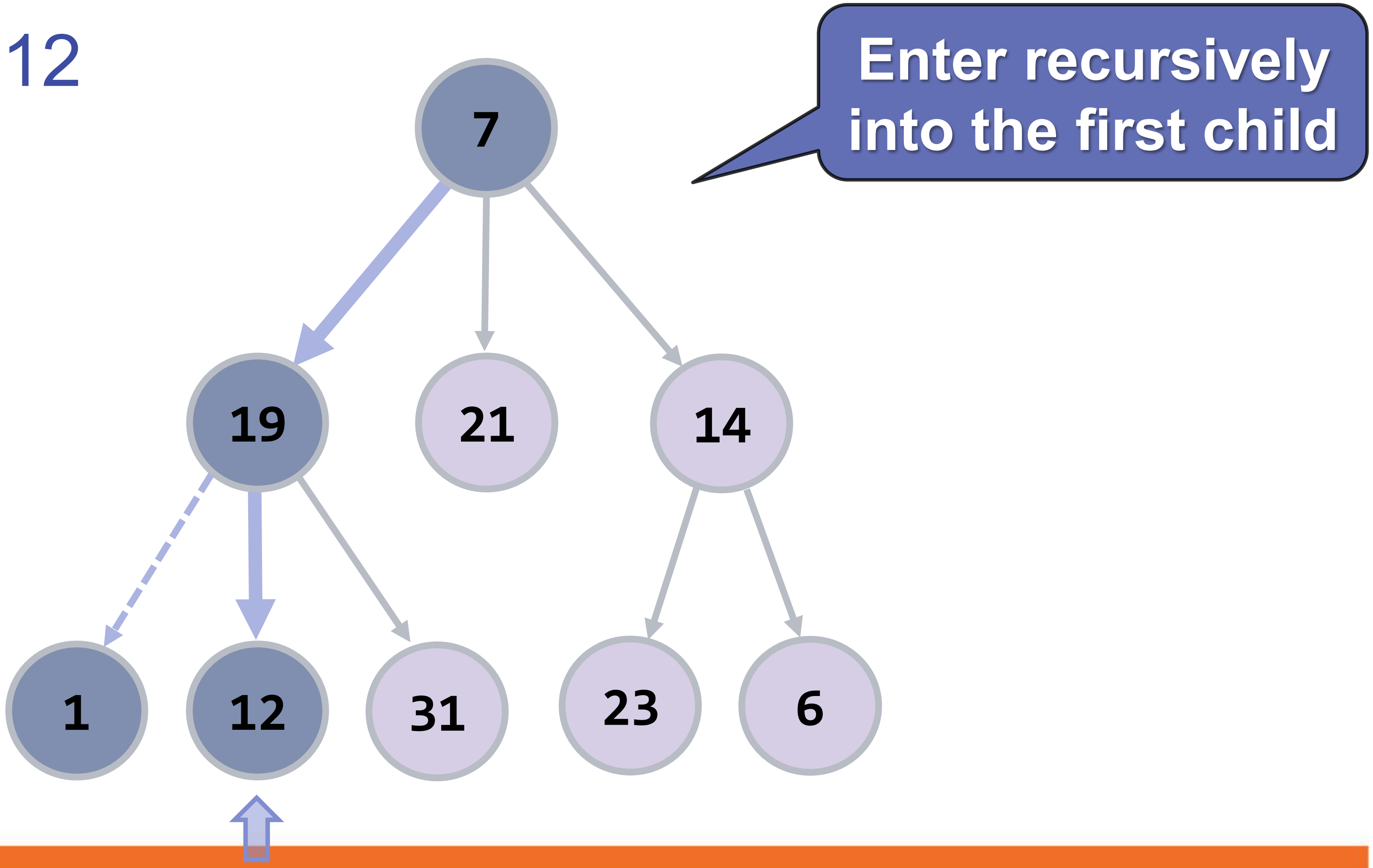
DFS in Action (Step 4)

- Stack: 7, 19
- Output: 1



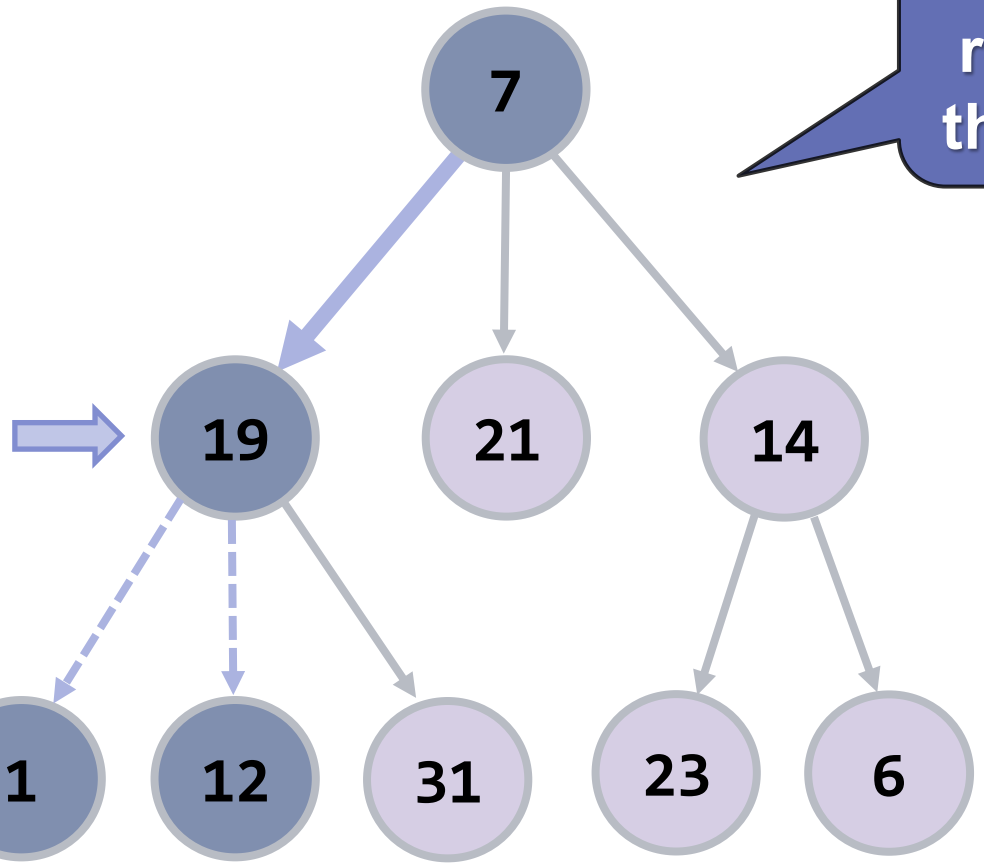
DFS in Action (Step 5)

- Stack: 7, 19, 12
- Output: 1



DFS in Action (Step 6)

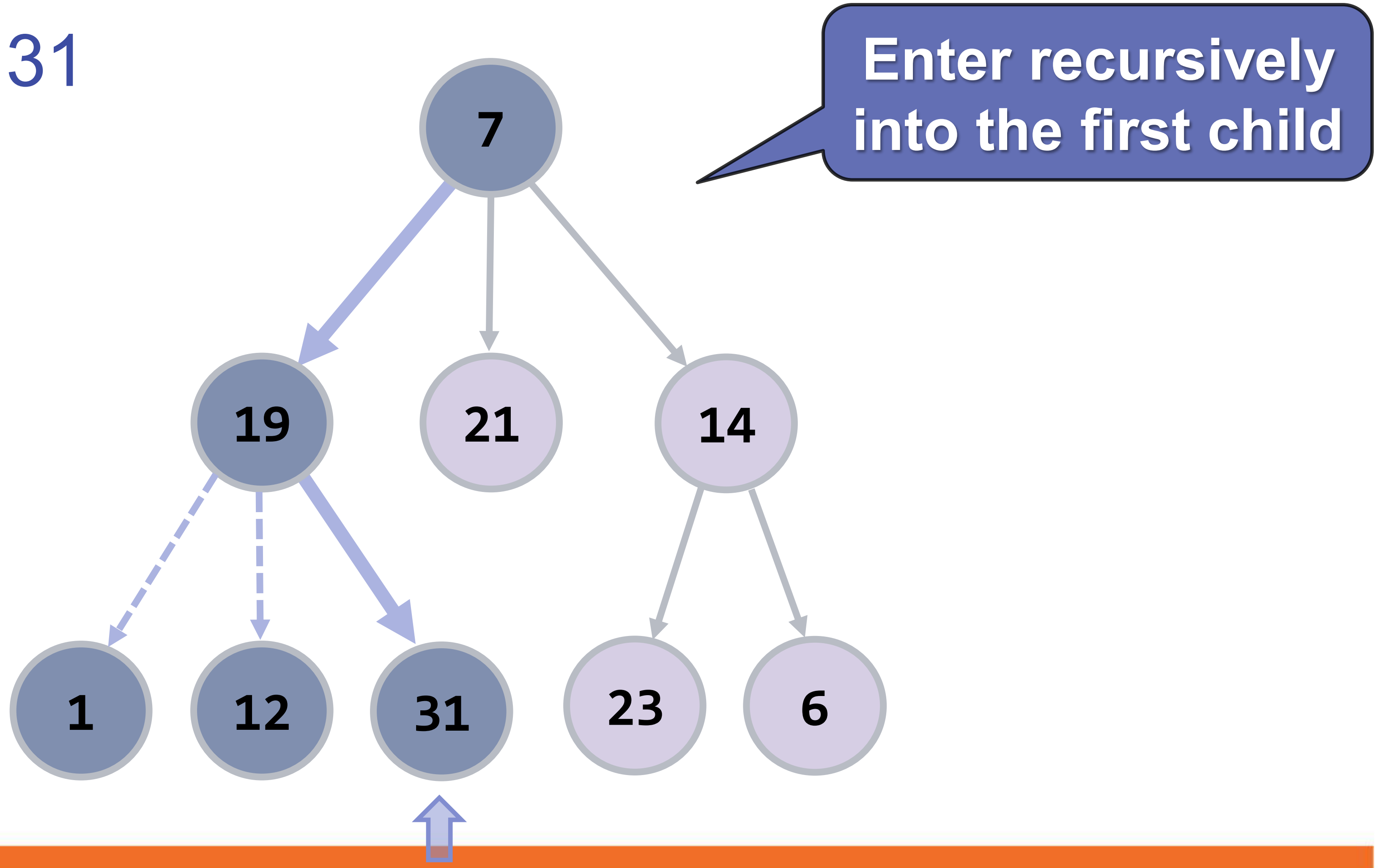
- Stack: 7, 19
- Output: 1, 12



Return back from recursion and print the last visited node

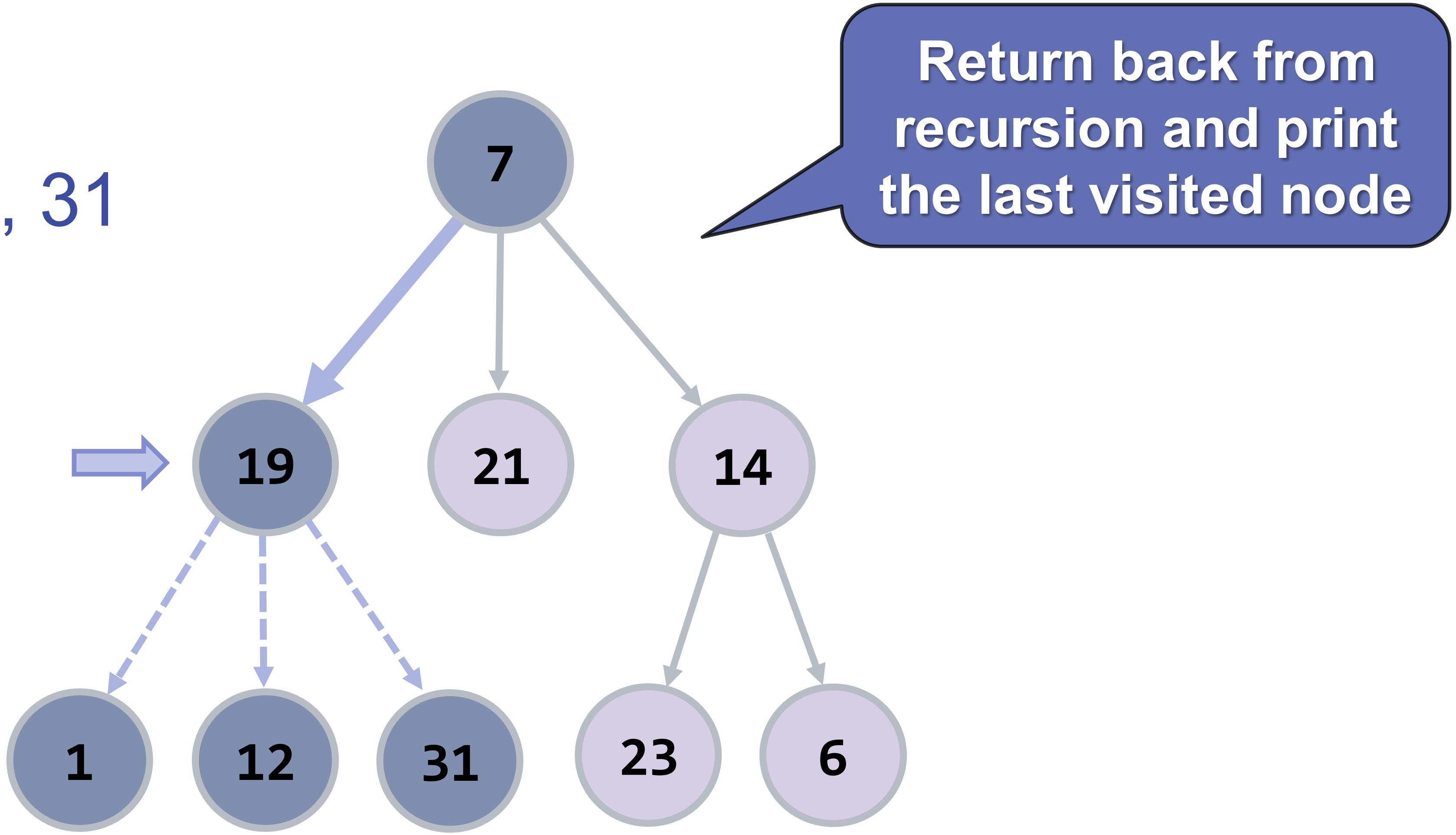
DFS in Action (Step 7)

- Stack: 7, 19, 31
- Output: 1, 12



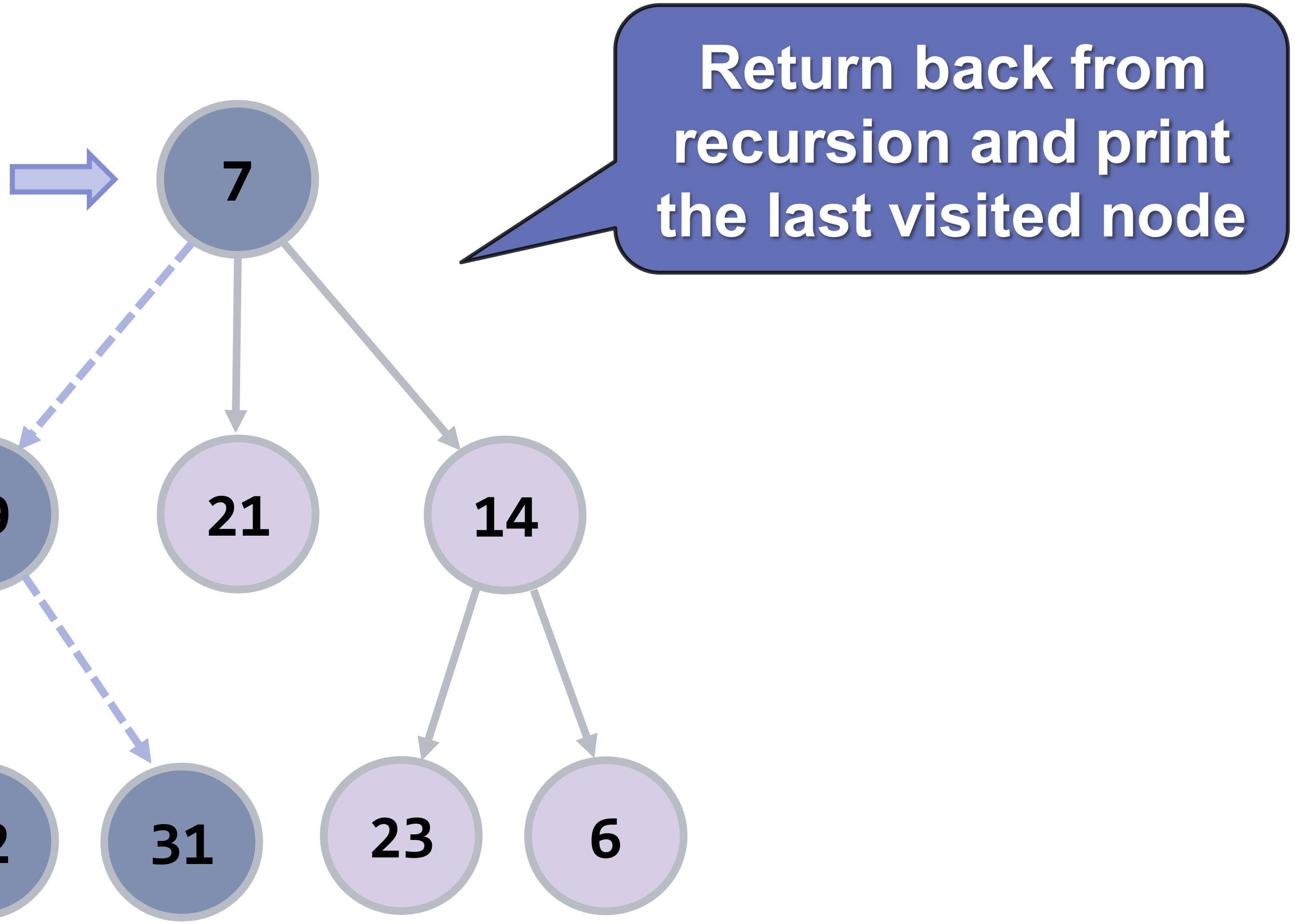
DFS in Action (Step 8)

- Stack: 7, 19
- Output: 1, 12, 31



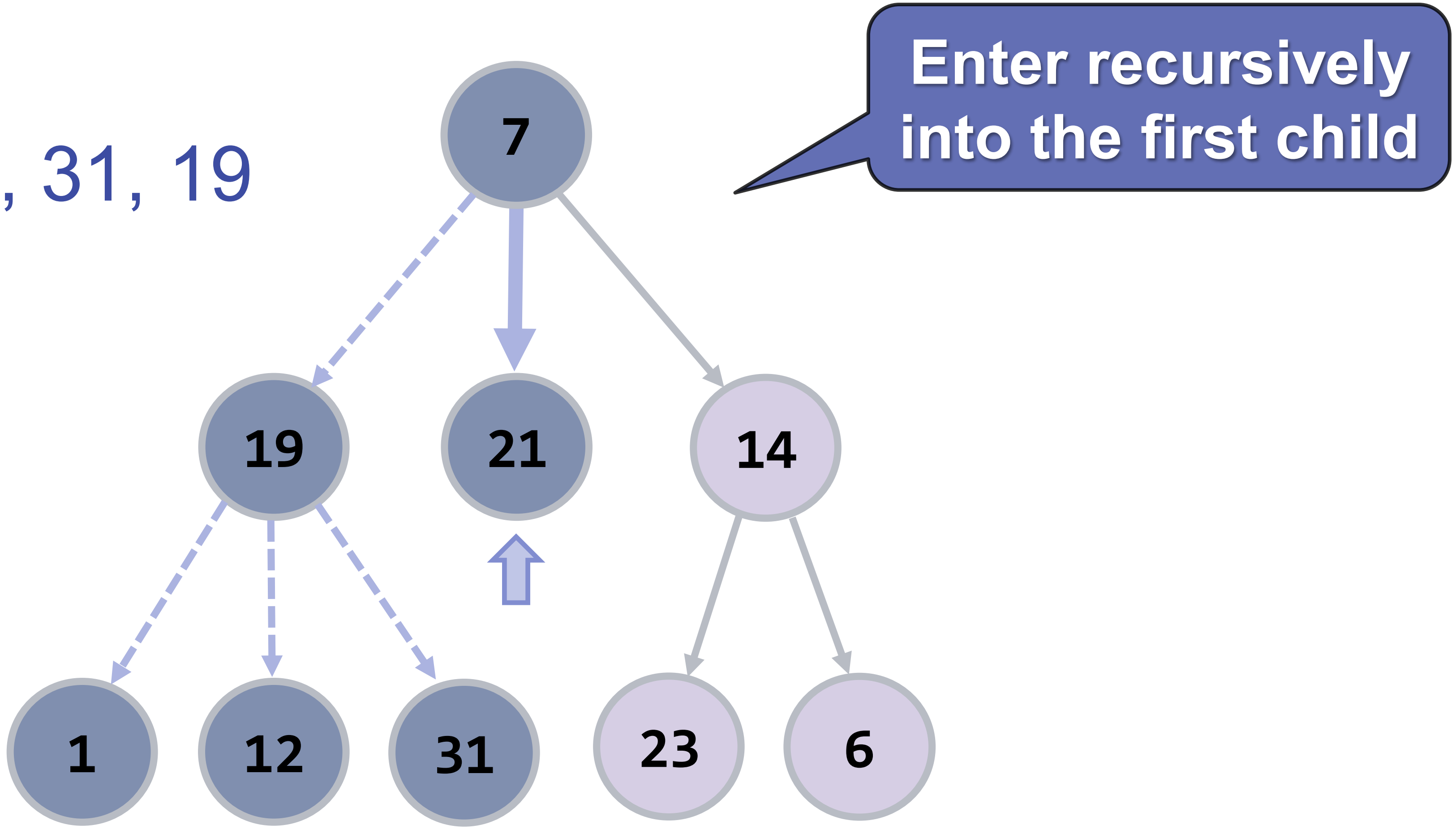
DFS in Action (Step 9)

- Stack: 7
- Output: 1, 12, 31, 19



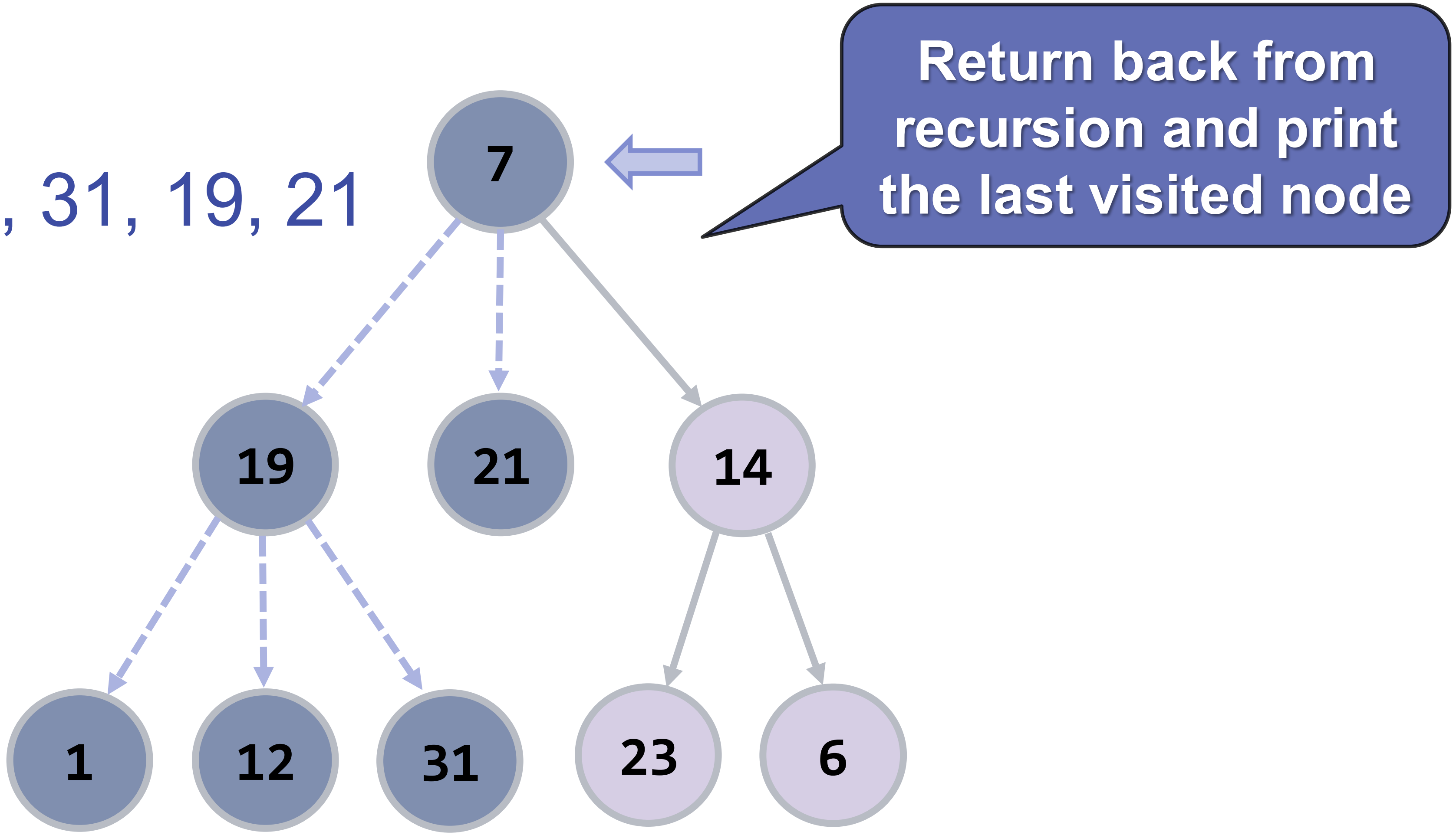
DFS in Action (Step 10)

- Stack: 7, 21
- Output: 1, 12, 31, 19



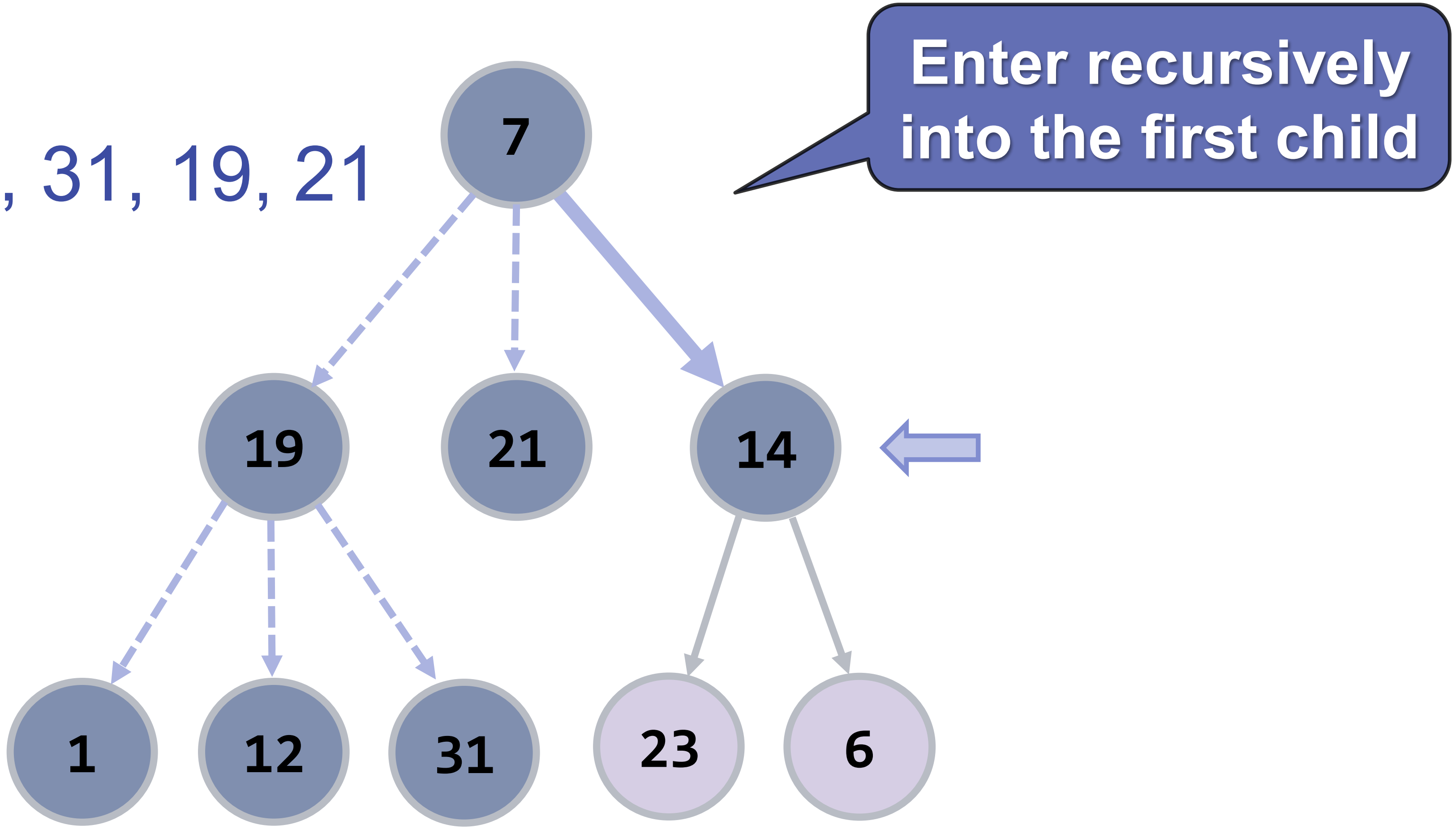
DFS in Action (Step 11)

- Stack: 7
- Output: 1, 12, 31, 19, 21



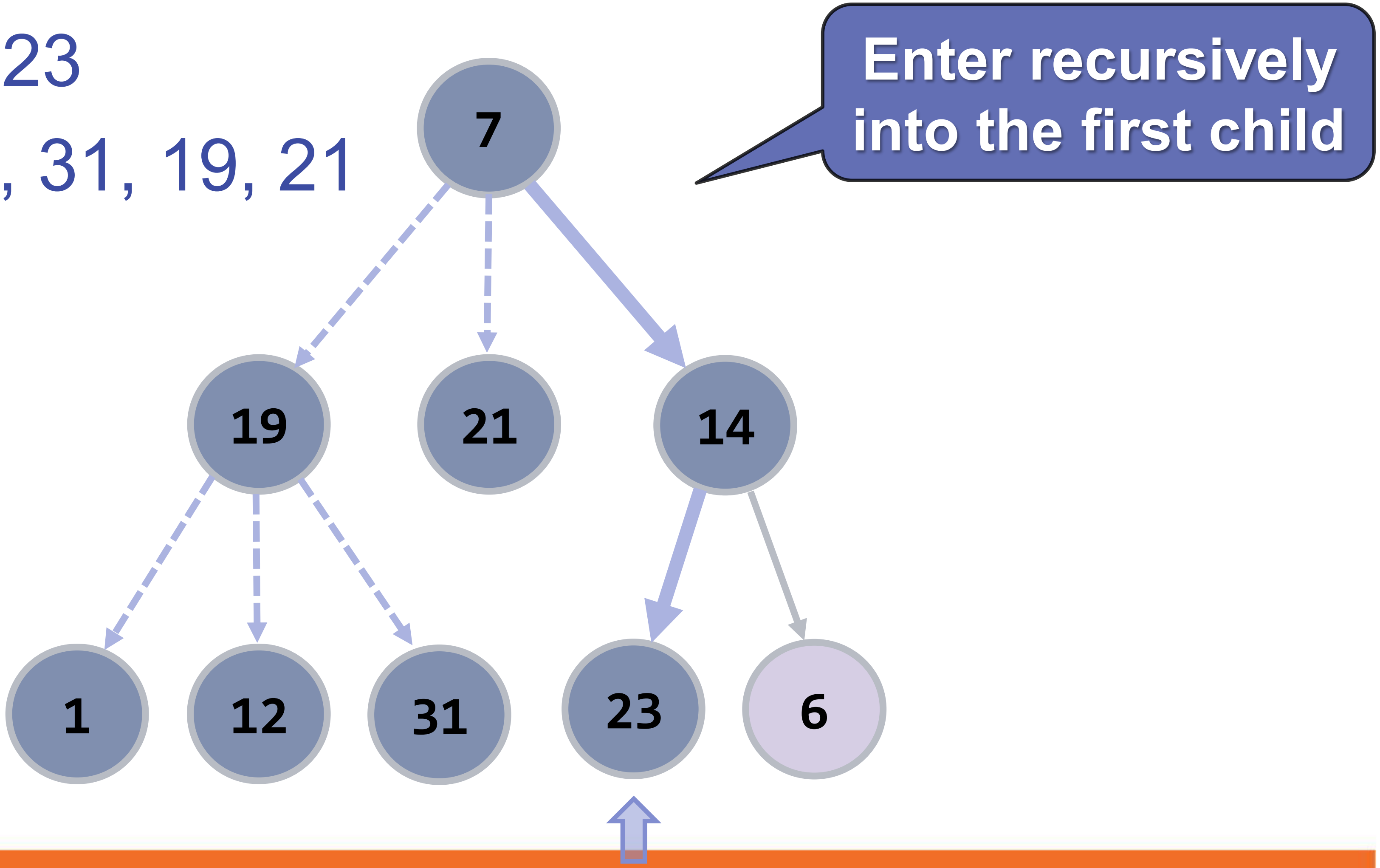
DFS in Action (Step 12)

- Stack: 7, 14
- Output: 1, 12, 31, 19, 21



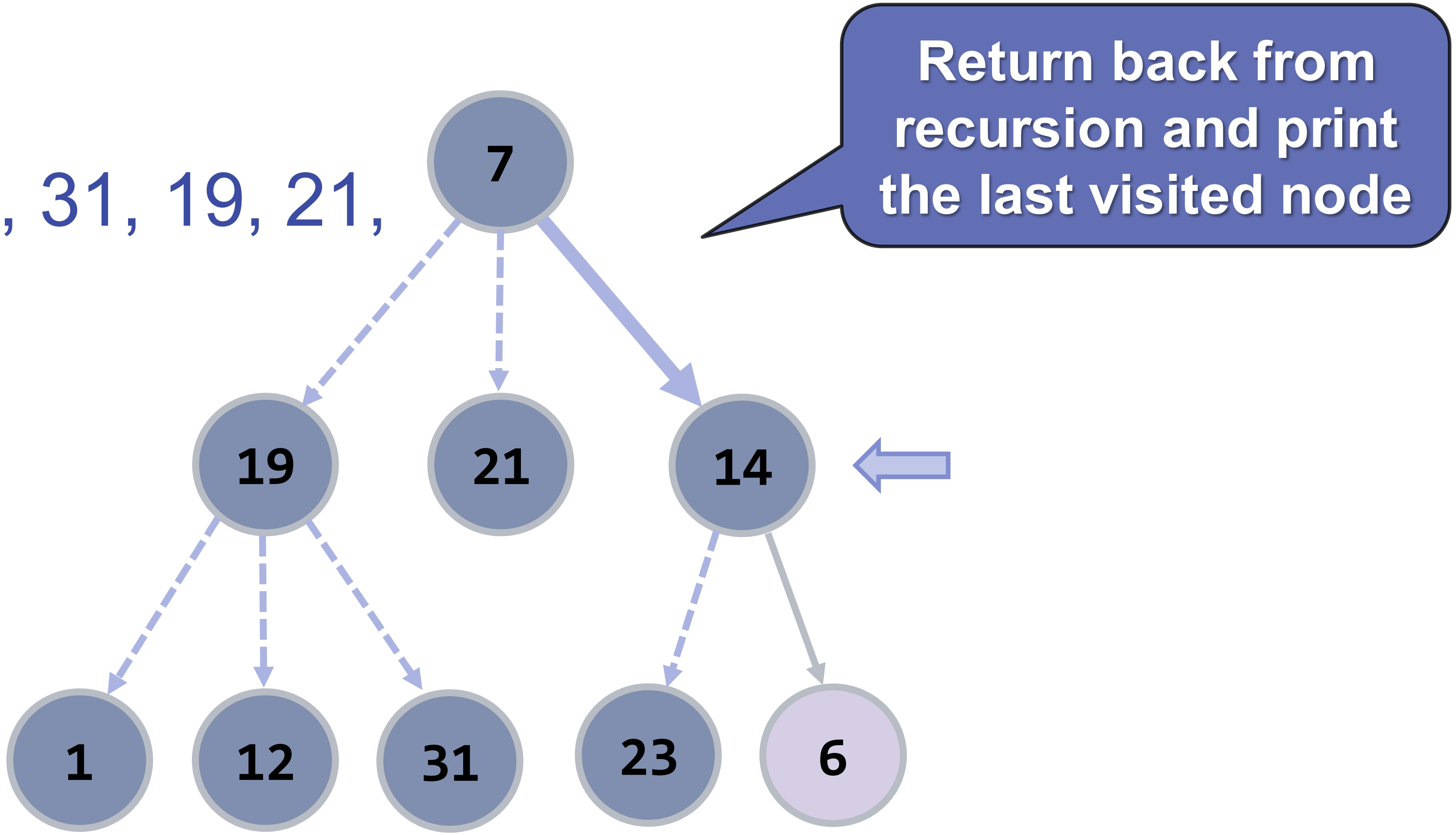
DFS in Action (Step 13)

- Stack: 7, 14, 23
- Output: 1, 12, 31, 19, 21



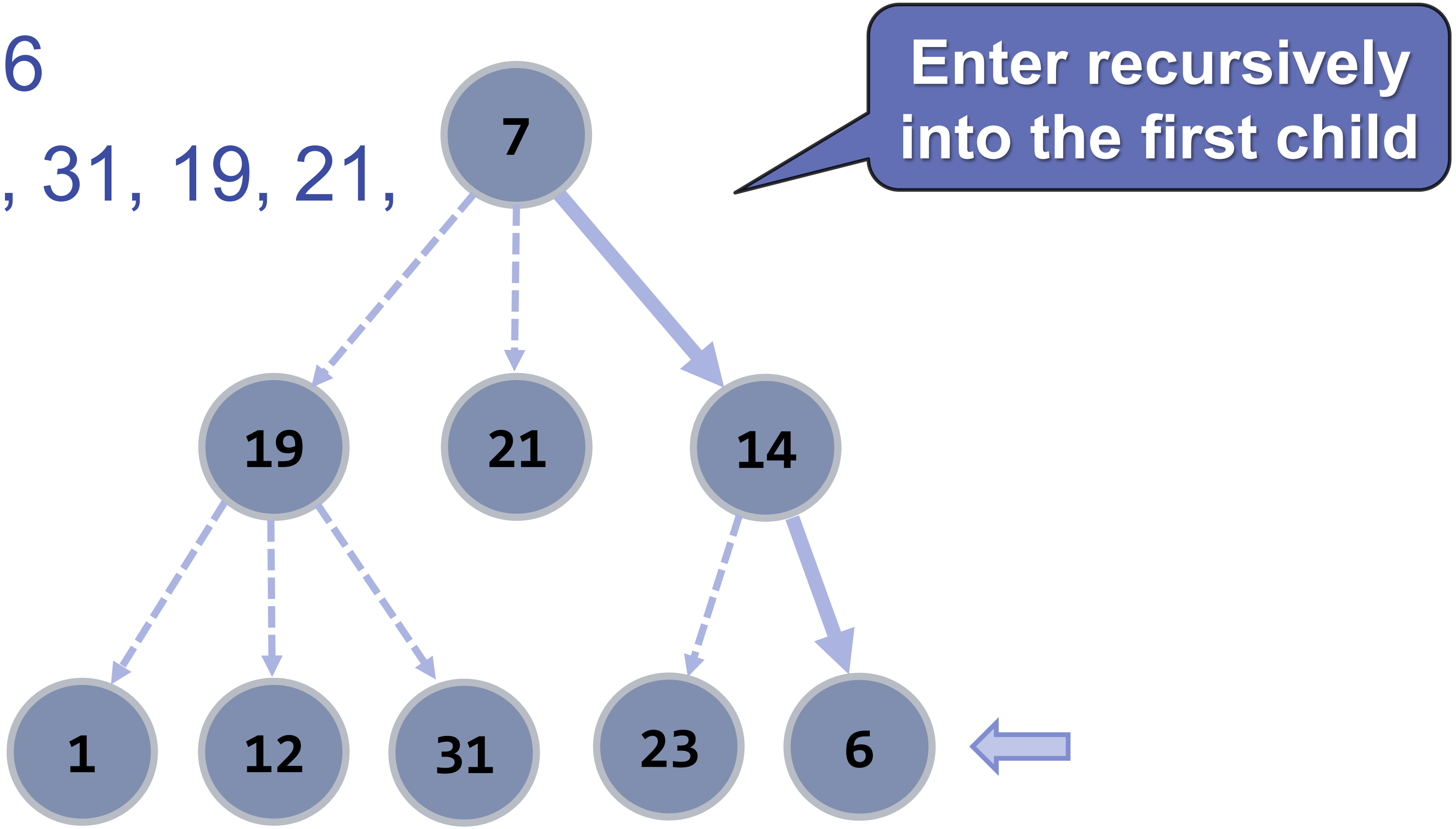
DFS in Action (Step 14)

- Stack: 7, 14
- Output: 1, 12, 31, 19, 21, 23



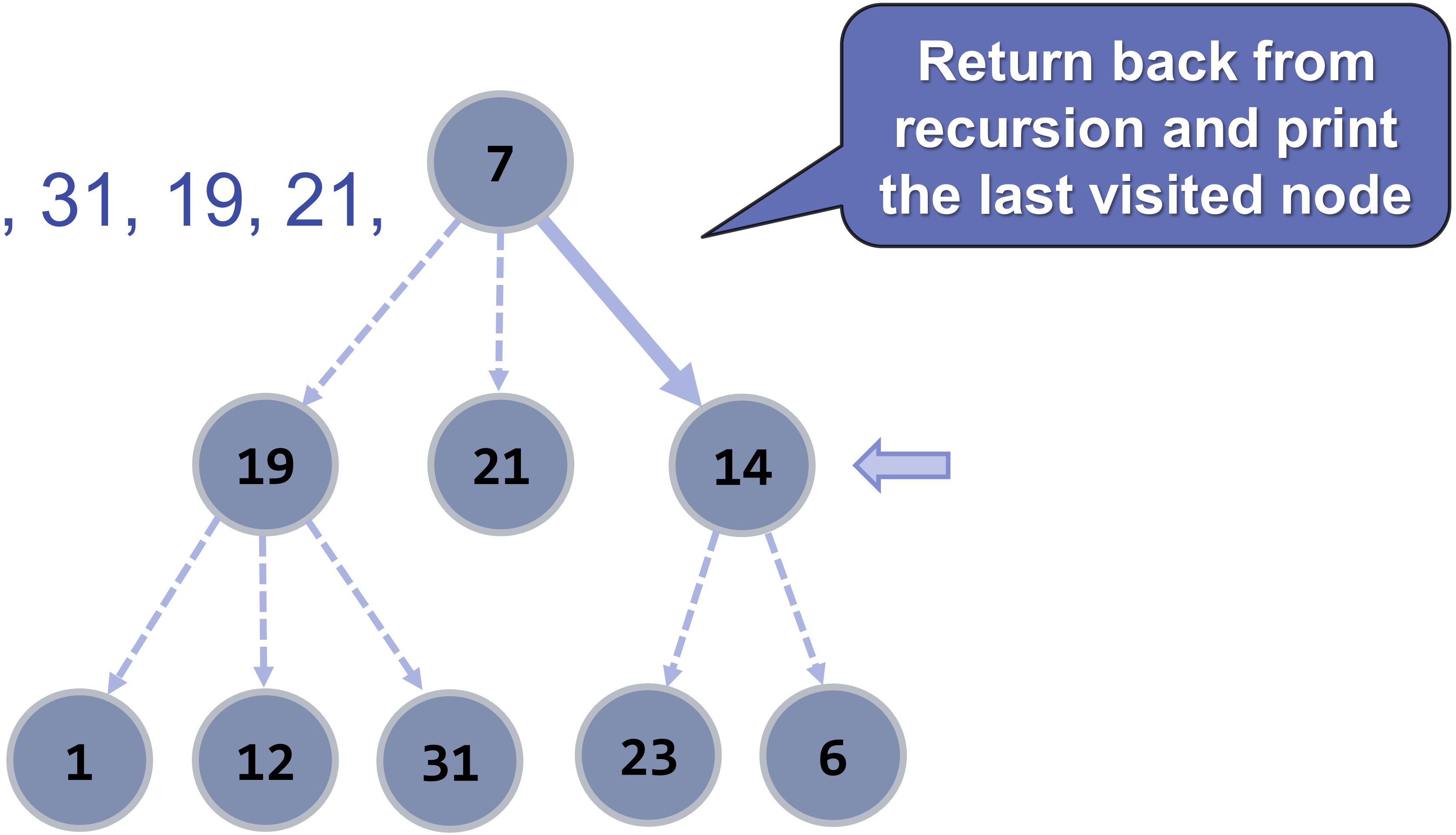
DFS in Action (Step 15)

- Stack: 7, 14, 6
- Output: 1, 12, 31, 19, 21, 23



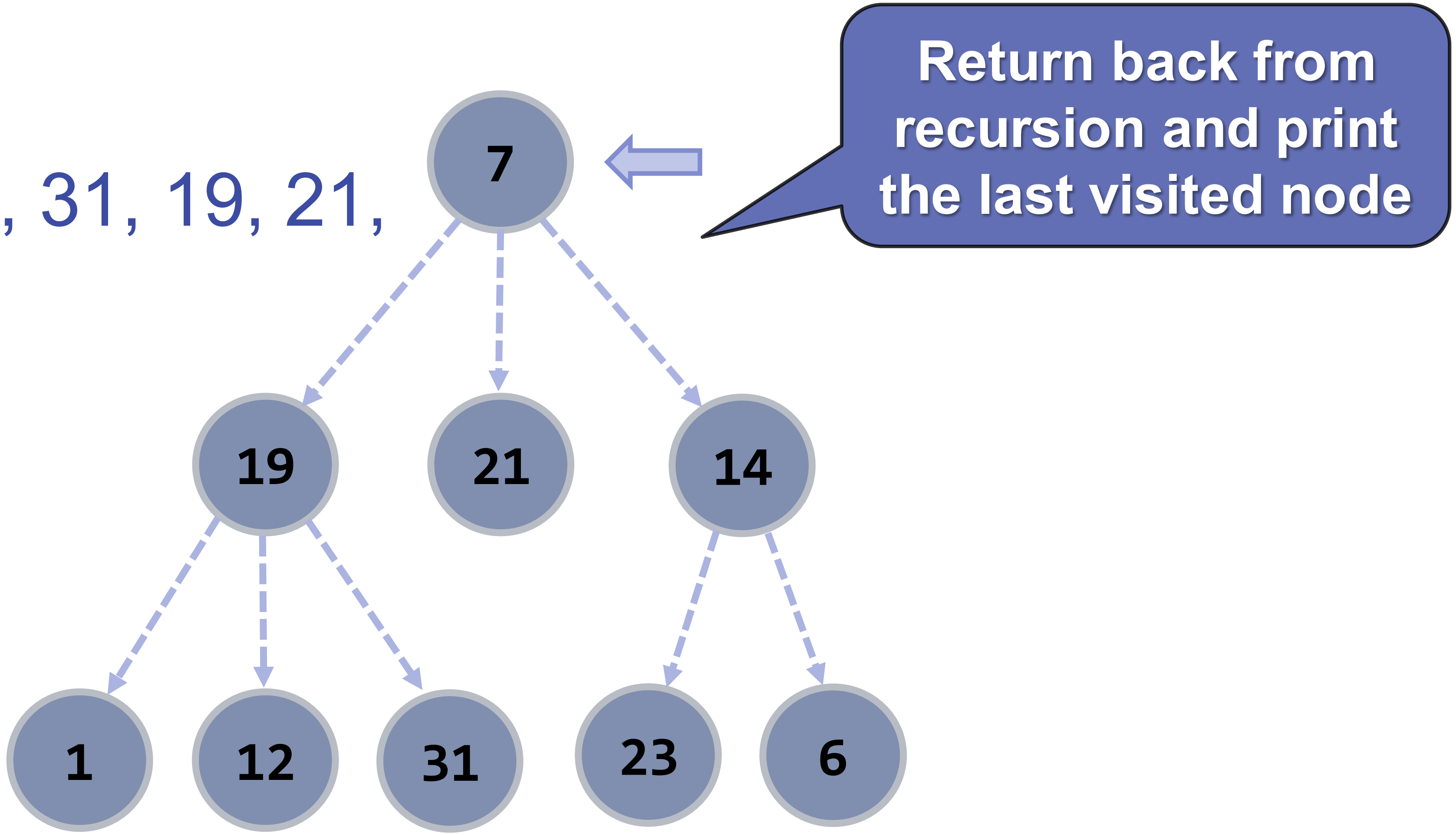
DFS in Action (Step 16)

- Stack: 7, 14
- Output: 1, 12, 31, 19, 21, 23, 6



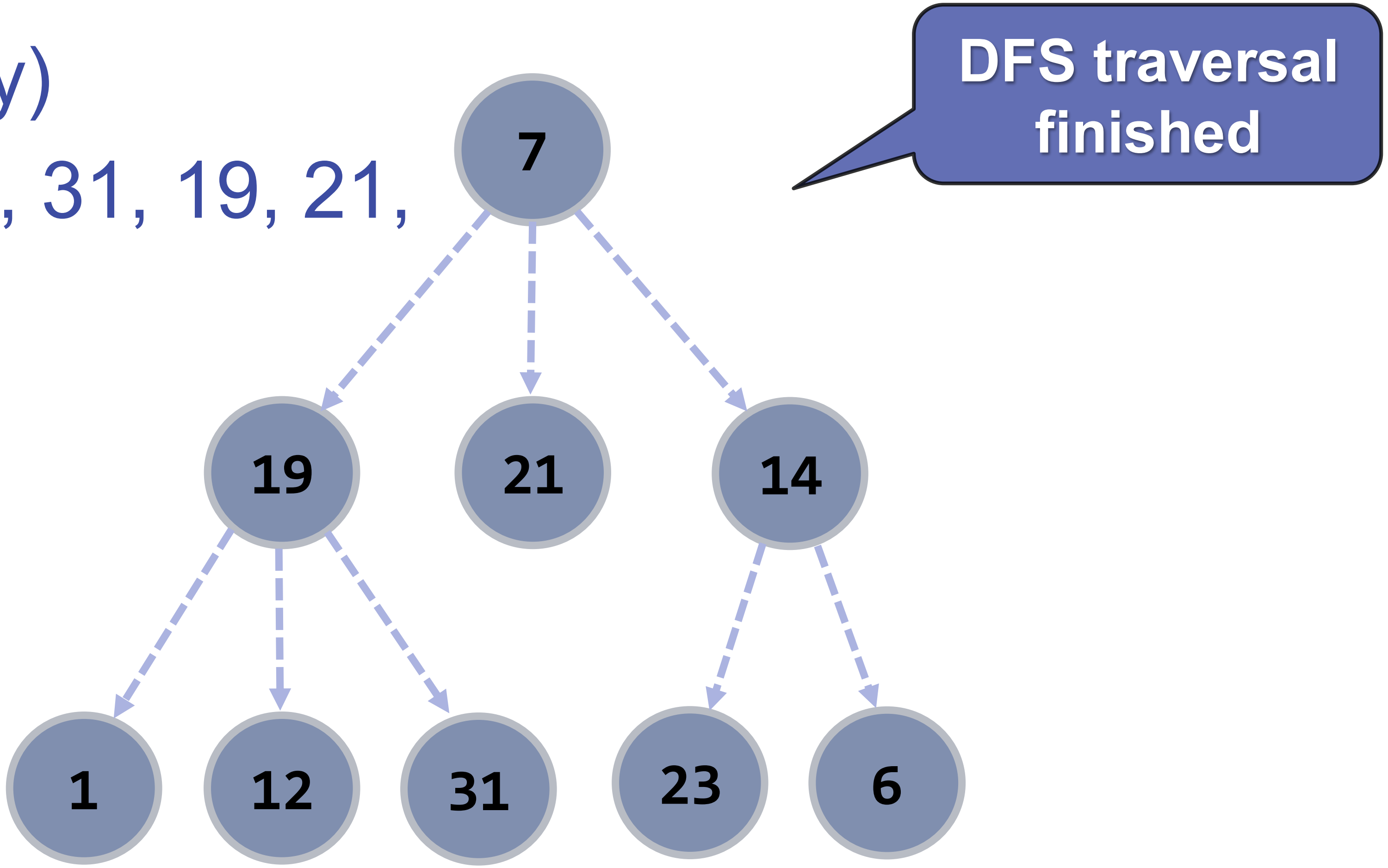
DFS in Action (Step 17)

- Stack: 7
- Output: 1, 12, 31, 19, 21, 23, 6, 14



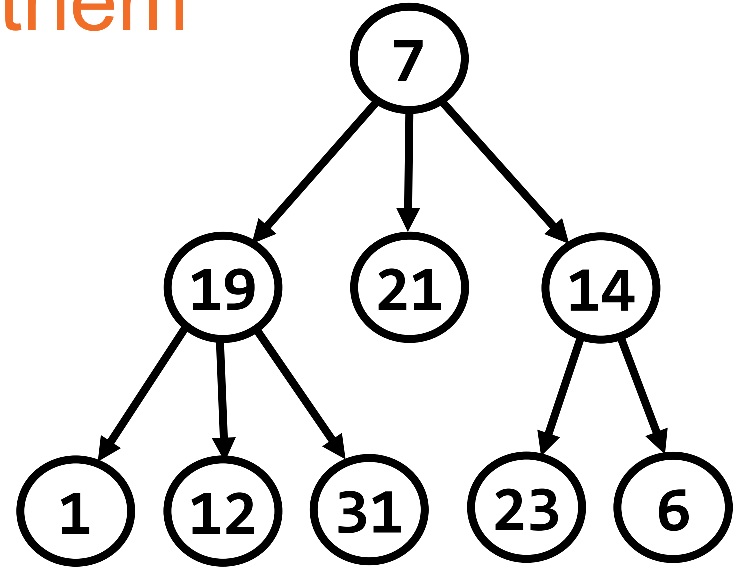
DFS in Action (Step 18)

- Stack: (empty)
- Output: 1, 12, 31, 19, 21, 23, 6, 14, 7



Problem: Order DFS

- Given the **Tree<E>** structure, define a method
  - **List<E> orderDfs()**
- That returns elements in order of DFS algorithm visiting them



1 12 31 19 21 23 6 14 7

Solution: Order DFS

```java
public List<E> orderDfs() {
    List<E> order = new ArrayList<>();
    this.dfs(this, order);
    return order;
}

private void dfs(Tree<E> tree, List<E> order) {
    for (Tree<E> child : tree.children) {
        this.dfs(child, order);
    }
    order.add(tree.key);
}
```

Conclusion

- What did we got so far?
  - Had we achieved any **better complexity**?
  - Are we working **with O(log(n))**?
- Well the answer is…
  - **No!**
  - We **had not**, why? Still we are stuck at **linear complexity** for searching operations
- We will try to solve that with **BST**

Summary

- **Trees** are recursive data structures
  - A tree is a node holding a set of children (which are also nodes)
  - Edges connect Nodes
- **DFS** → children first, **BFS** → root first