Computing Foundations of Data Science

Binary Search Tree

Lecture 10

Hyung-Sin Kim

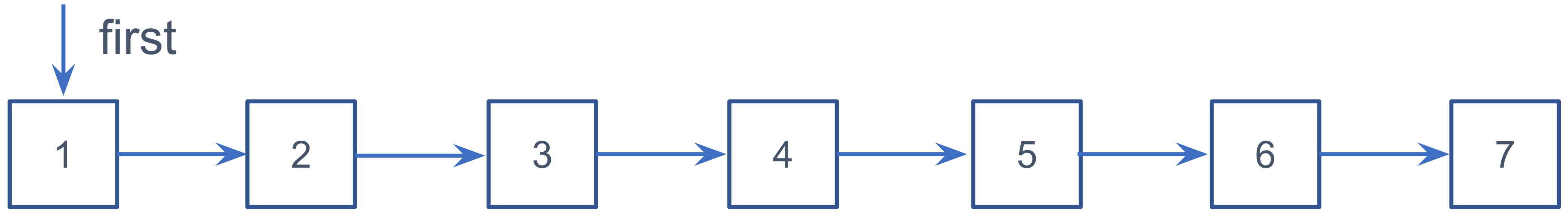
SNU Graduate School of Data Science

Review

- Arrays
  - A sequence of neighboring memory boxes
    - Know where an **arbitrary** (**i-th**) element is located, by using the **neighboring rule**
  - **Limitation**: Fixed length and Expensive resizing
    - Make a brand-new array + copy all the existing elements
  - **Improvement**: Resizing step adjustment
- Linked lists
  - A list of nodes each of which has a link to another node
    - Know where the **next** element is located, by using the **next pointer**
  - **Limitation:** Don't know what is where - Frequent navigation through the list
  - **Improvement:** Caching and sentinel
- Queues (FIFO) and Stacks (LIFO)

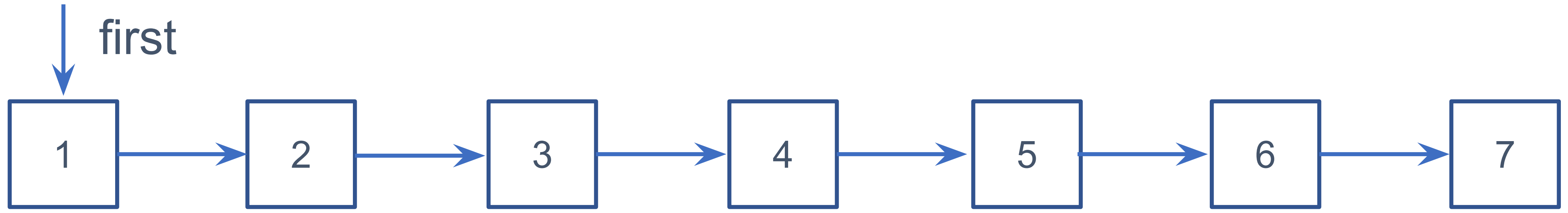
Downside of Linked Lists

- Slow search (O(N)) even when items are sorted



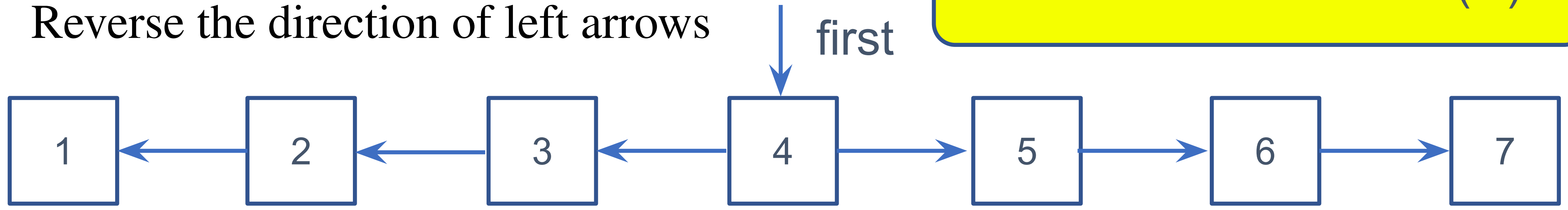
Downside of Linked Lists

- Slow search (O(N)) even when items are sorted



- An improvement for search
  - Change the first node to middle
  - Reverse the direction of left arrows

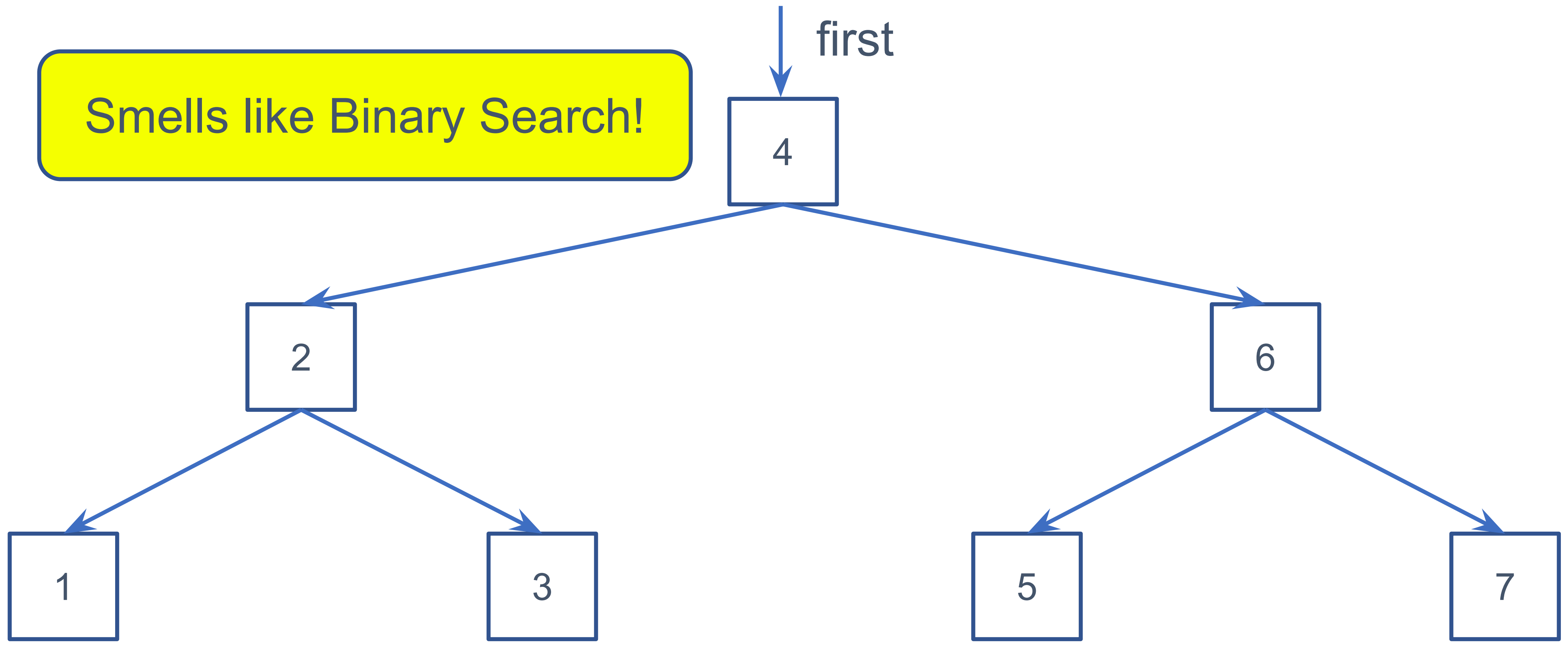
Twice faster but still O(N)!



Improving Linked Lists

- How about this?

Smells like Binary Search!



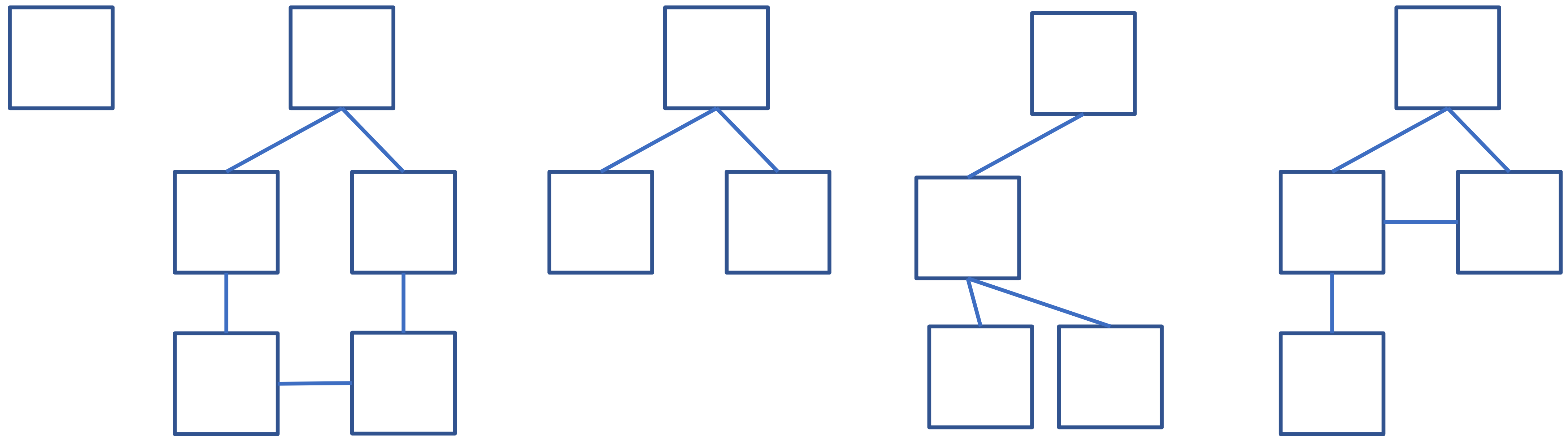
Trees

- A tree comprises a set of **nodes** that are **connected (linked)** to each other
- There is **only one path** between two nodes in a tree

Trees

- A tree comprises a set of **nodes** that are **connected (linked)** to each other
- There is **only one path** between two nodes in a tree

- Choose trees!

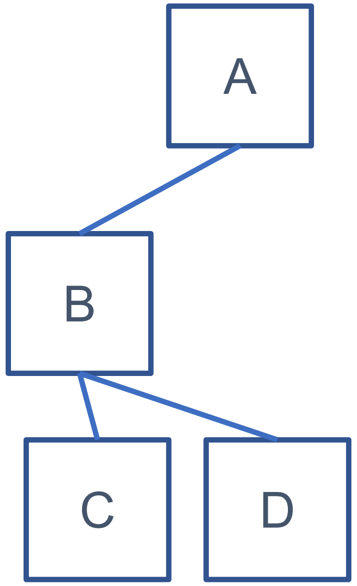


Rooted Binary Trees

- **Rooted** tree
  - There is one **root** node (at the top of the tree)
  - Every node (except the root) has one **parent** – the first node on its path toward the root
  - A node without a child is a **leaf**

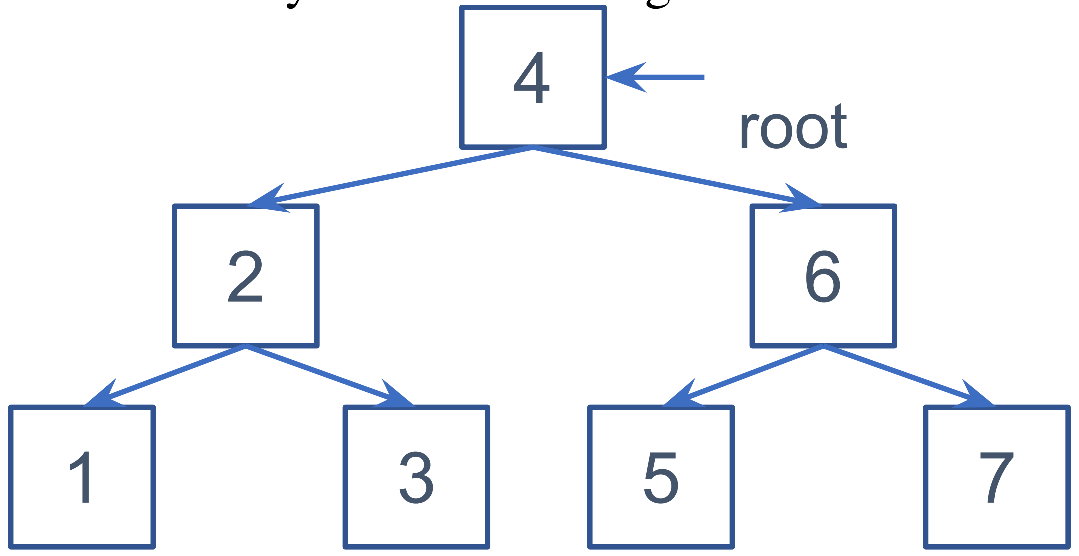
- Relationship
  - A is the root and a parent of B
  - B is a child of A and a parent of C and D
  - C and D are leaves and children of B

- Rooted **binary** tree
  - Each node has at most **two** children nodes



Binary Search Trees

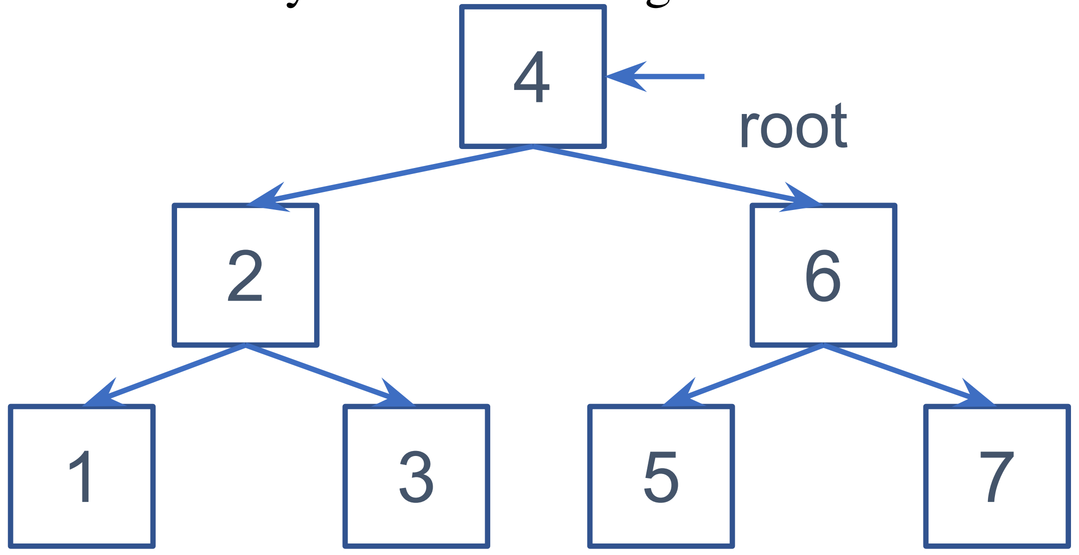
- A binary search tree is a rooted binary tree that has the following two properties
- For every node **x**,
  - **x**'s value is unique in the whole tree
  - Every node **y** in the left subtree of node **x** has value less than **x**'s value
  - Every node **z** in the right subtree of node **x** has value greater than **x**'s value



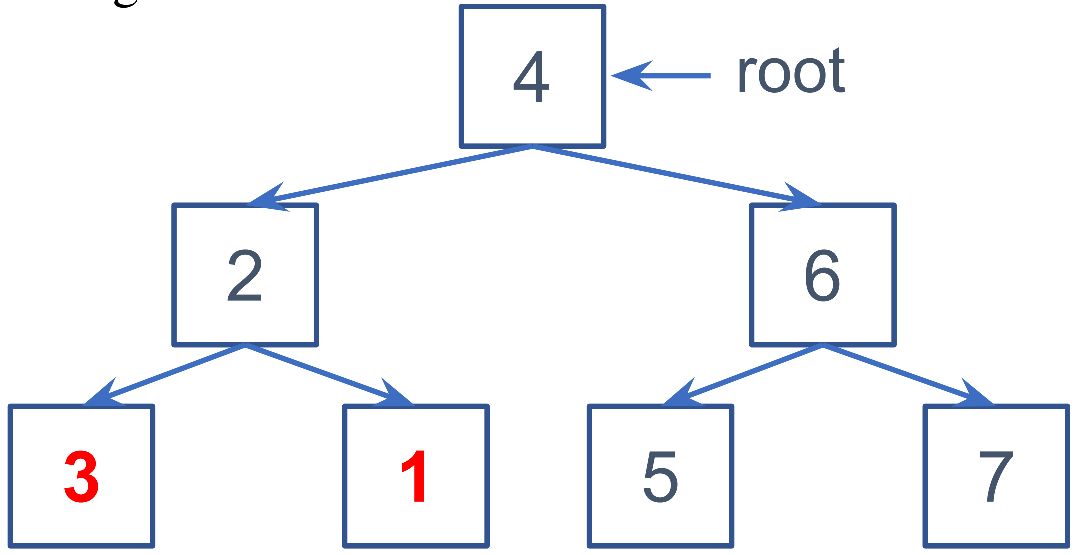
a Binary Search Tree

Binary Search Trees

- A binary search tree is a rooted binary tree that has the following two properties
- For every node **x**,
  - **x**'s value is <u>unique</u> in the whole tree
  - Every node **y** in the left subtree of node **x** has value less than **x**'s value
  - Every node **z** in the right subtree of node **x** has value greater than **x**'s value



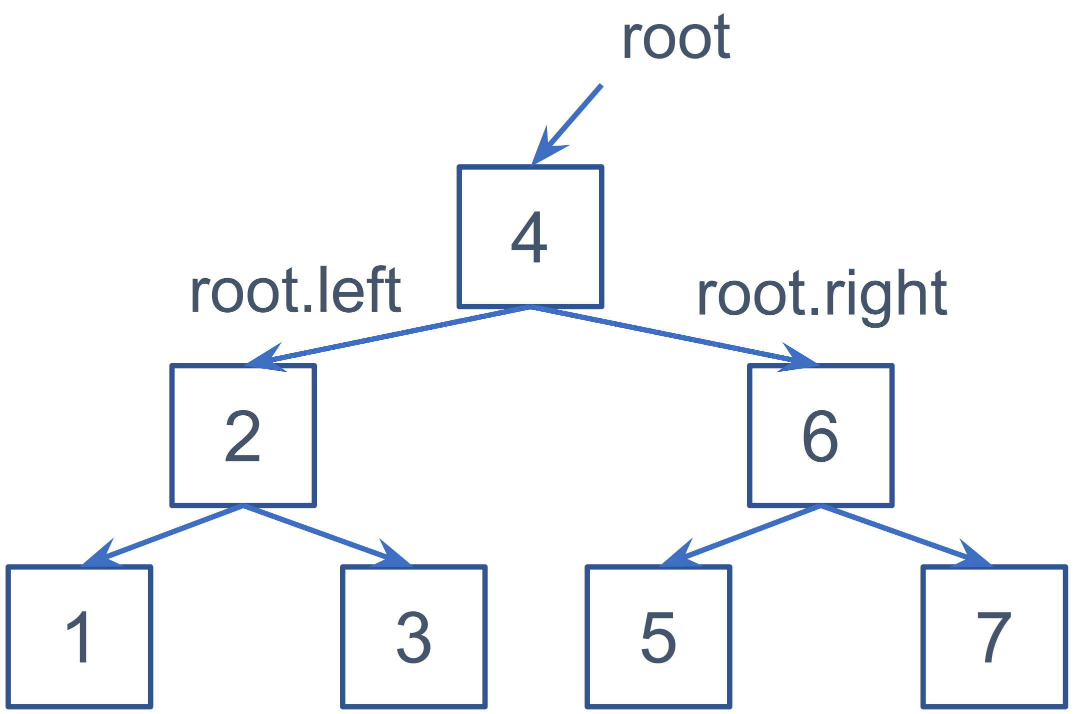
a Binary Search Tree



a Binary (not Search) Tree

Binary Search Trees

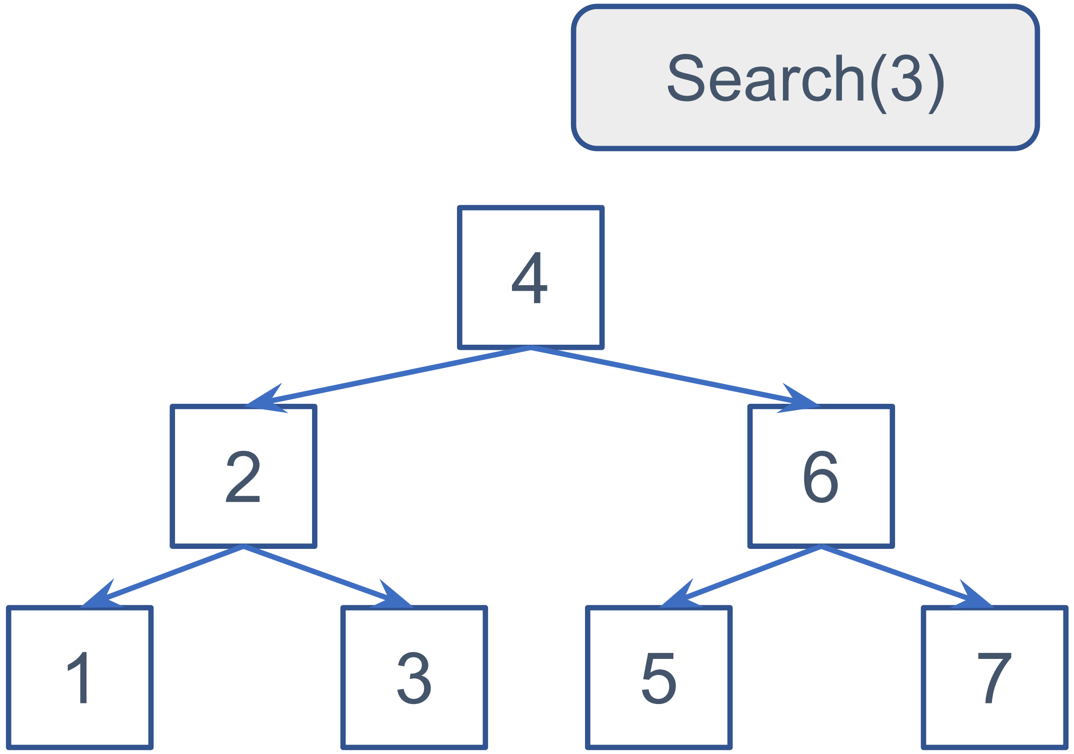
```
class TreeNode():
    def __init__(self, x: int):
        self.val = x
        self.left = None
        self.right = None

class BST():
    def __init__(self):
        self.root = None
    def search(self, x: int):
    def insert(self, x: int):
    def delete(self, x: int):
```



Search

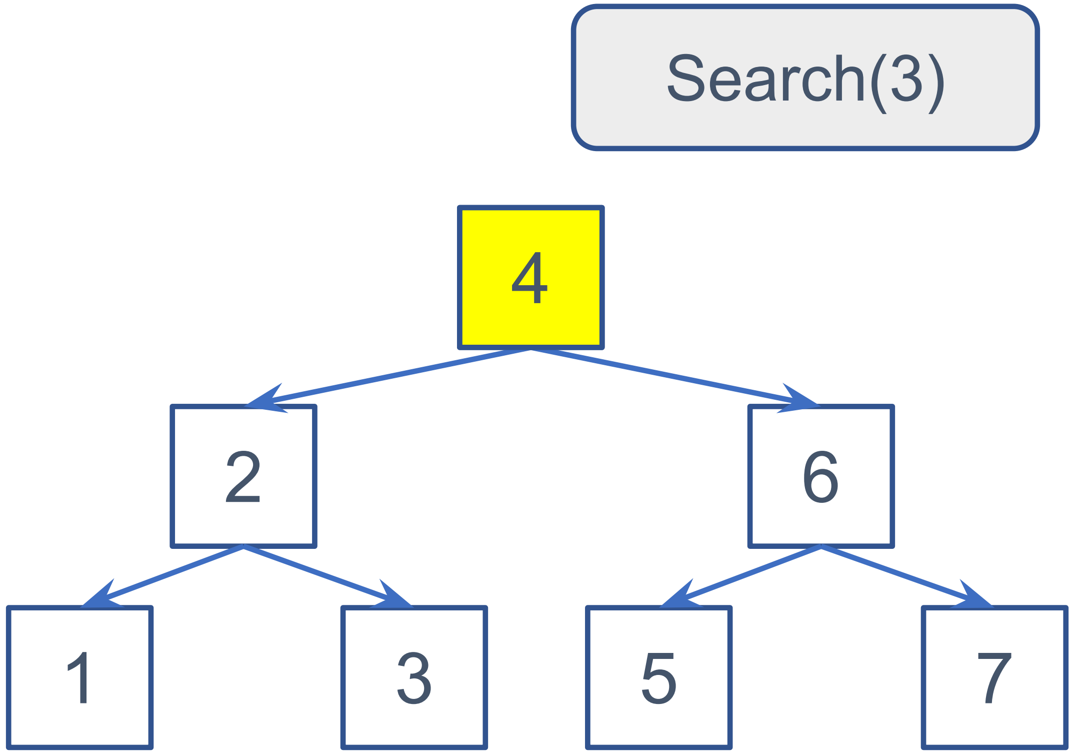
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



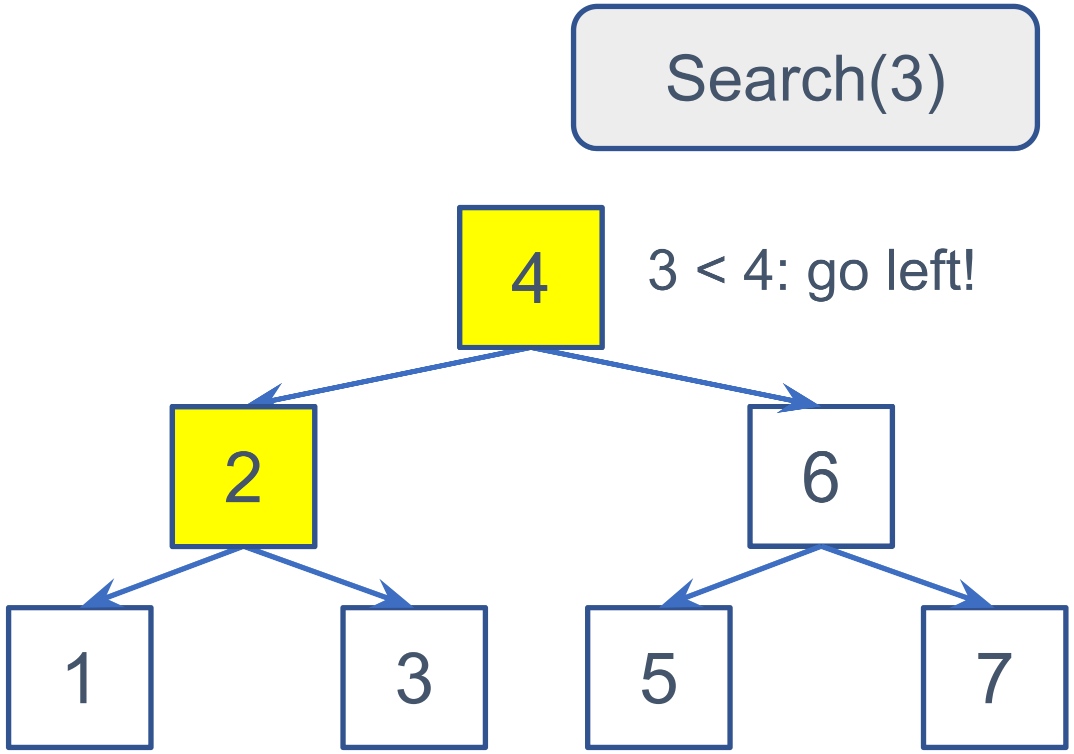
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



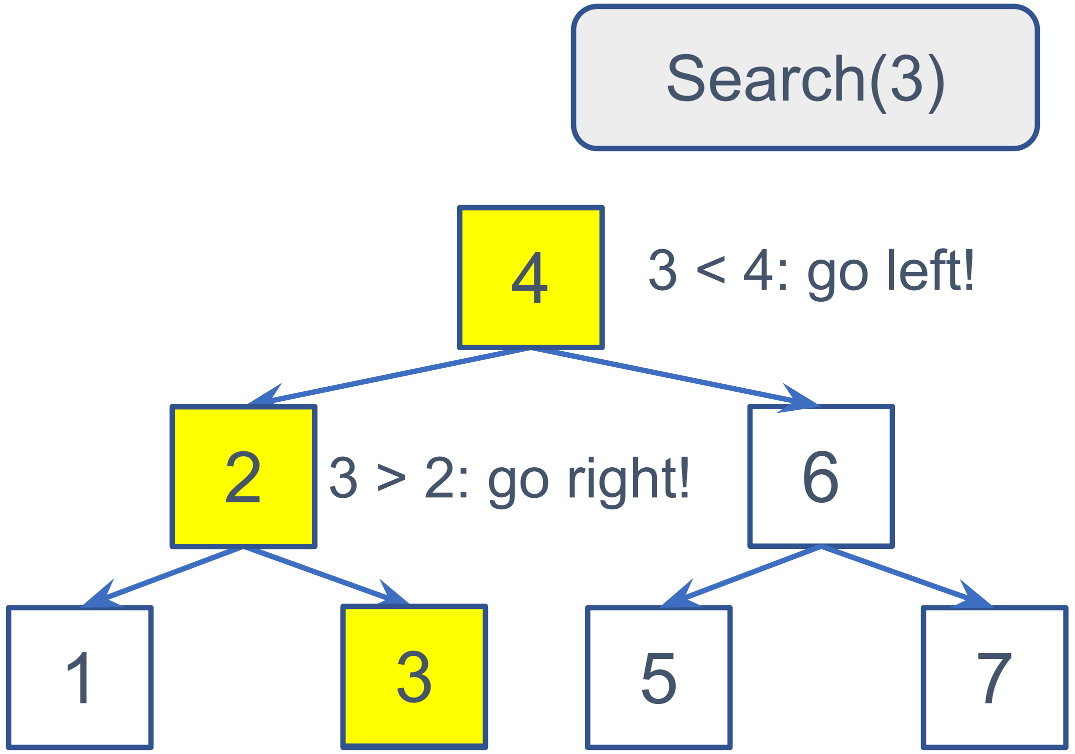
Binary Search Trees – Search

- class BST():
-     def __**searchHelp**(curNode: TreeNode, x) -> TreeNode:
-         if not curNode:
-             return None
-         if x == curNode.val:
-             return curNode
-         **elif x < curNode.val:**
-             **return self.__searchHelp(curNode.left, x)**
-         else:
-             return self.__searchHelp(curNode.right, x)



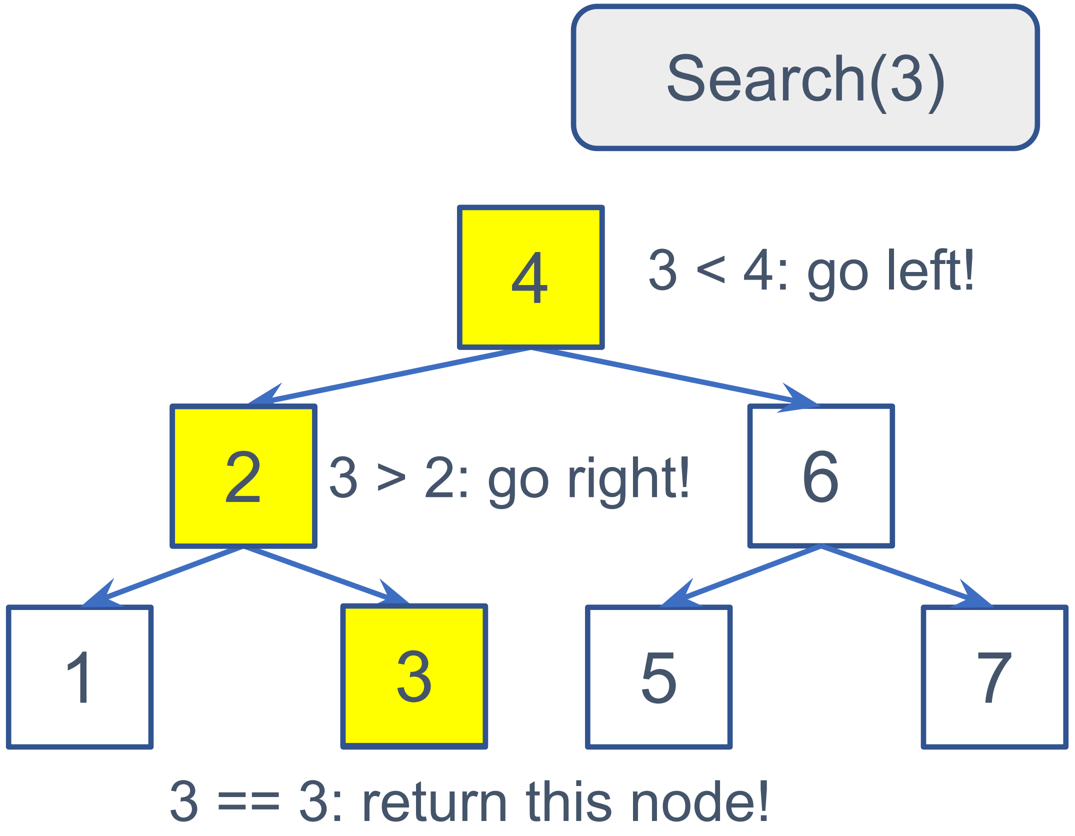
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



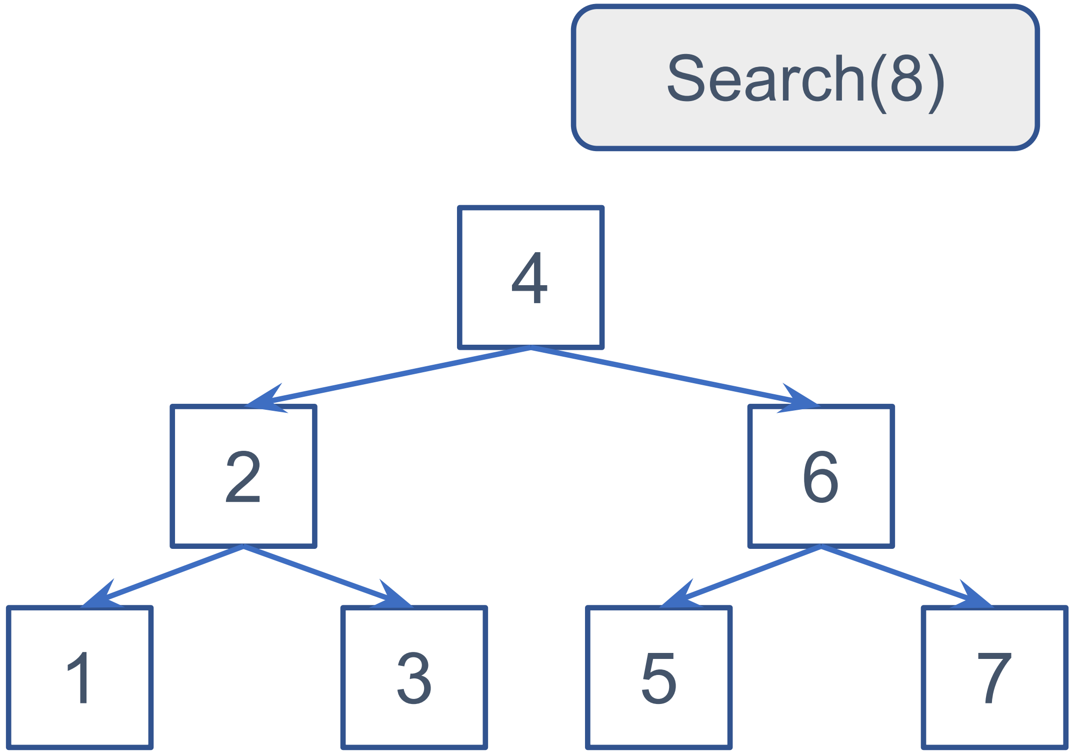
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



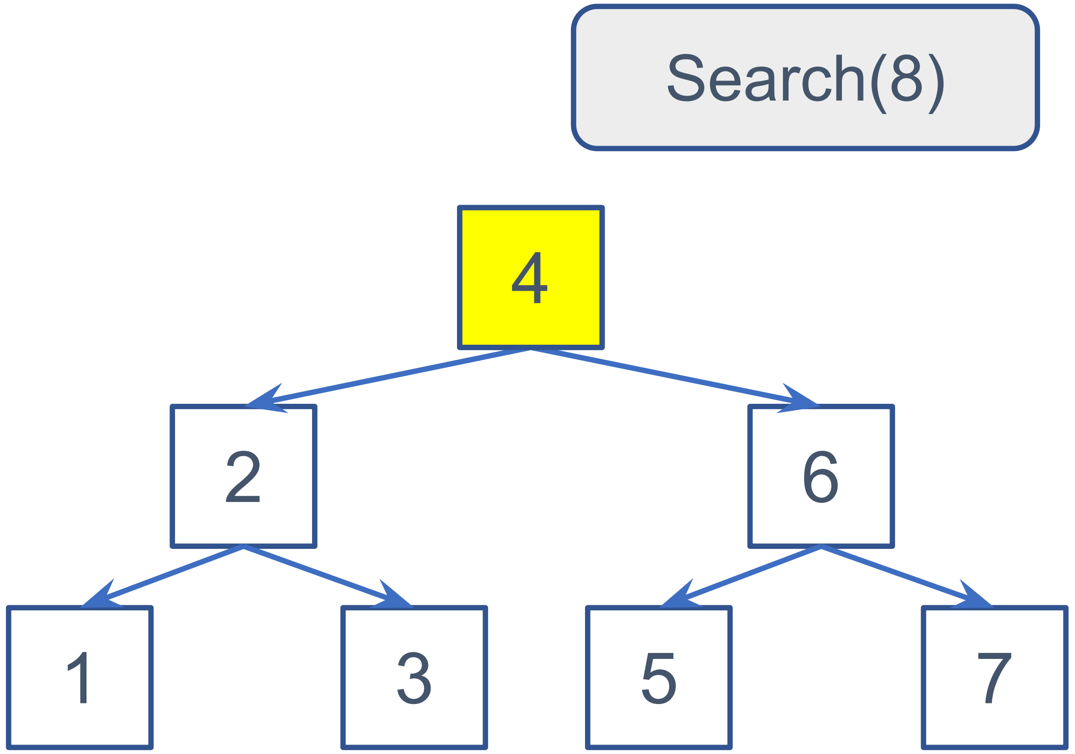
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



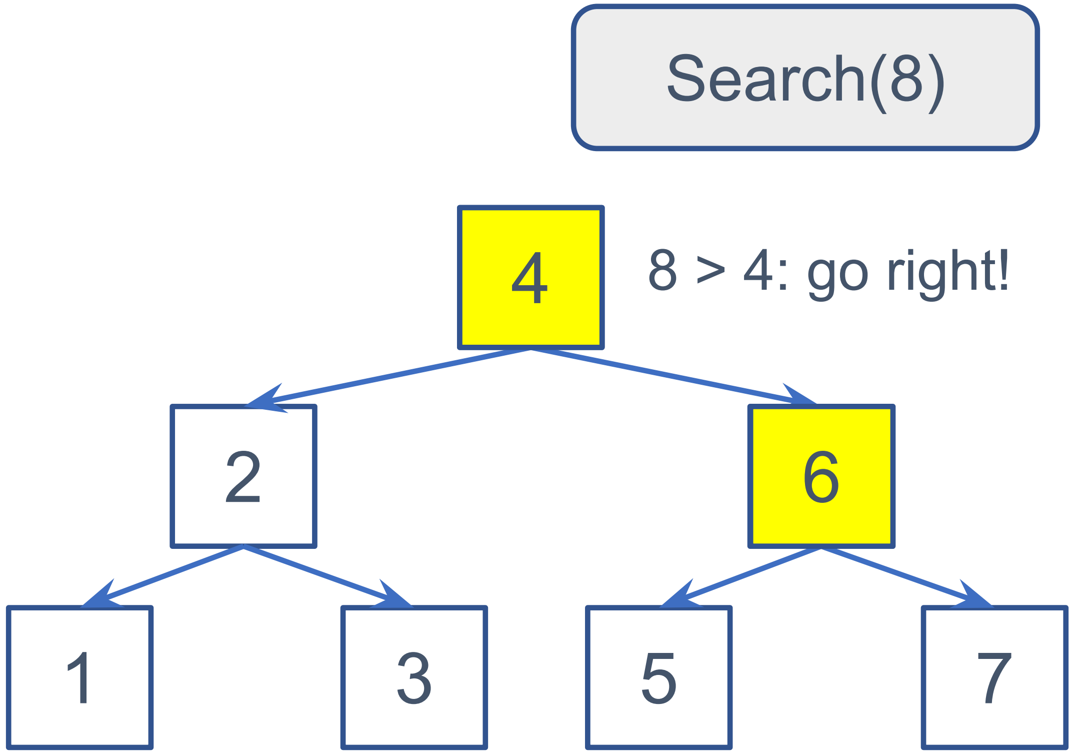
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



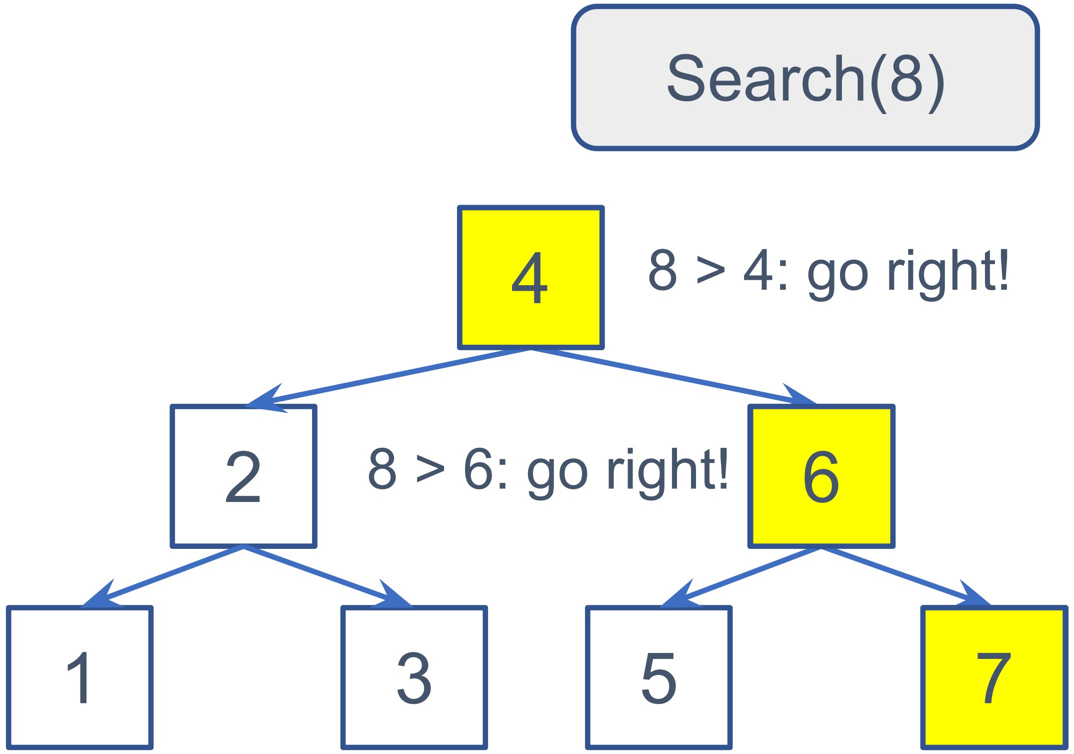
Binary Search Trees – Search

- class BST():
- def **__searchHelp**(curNode: TreeNode, x) -> TreeNode:
- if not curNode:
- return None
- if x == curNode.val:
- return curNode
- elif x < curNode.val:
- return self.__searchHelp(curNode.left, x)
- **else:**
- **return self.__searchHelp(curNode.right, x)**



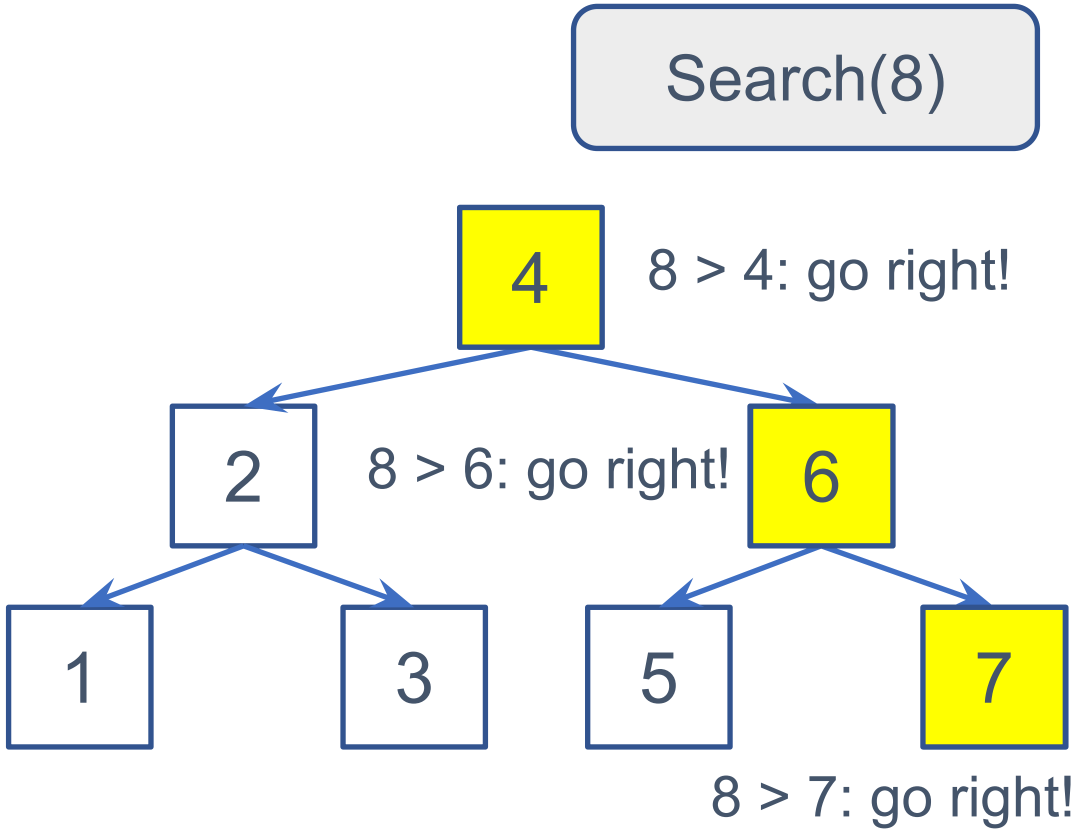
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



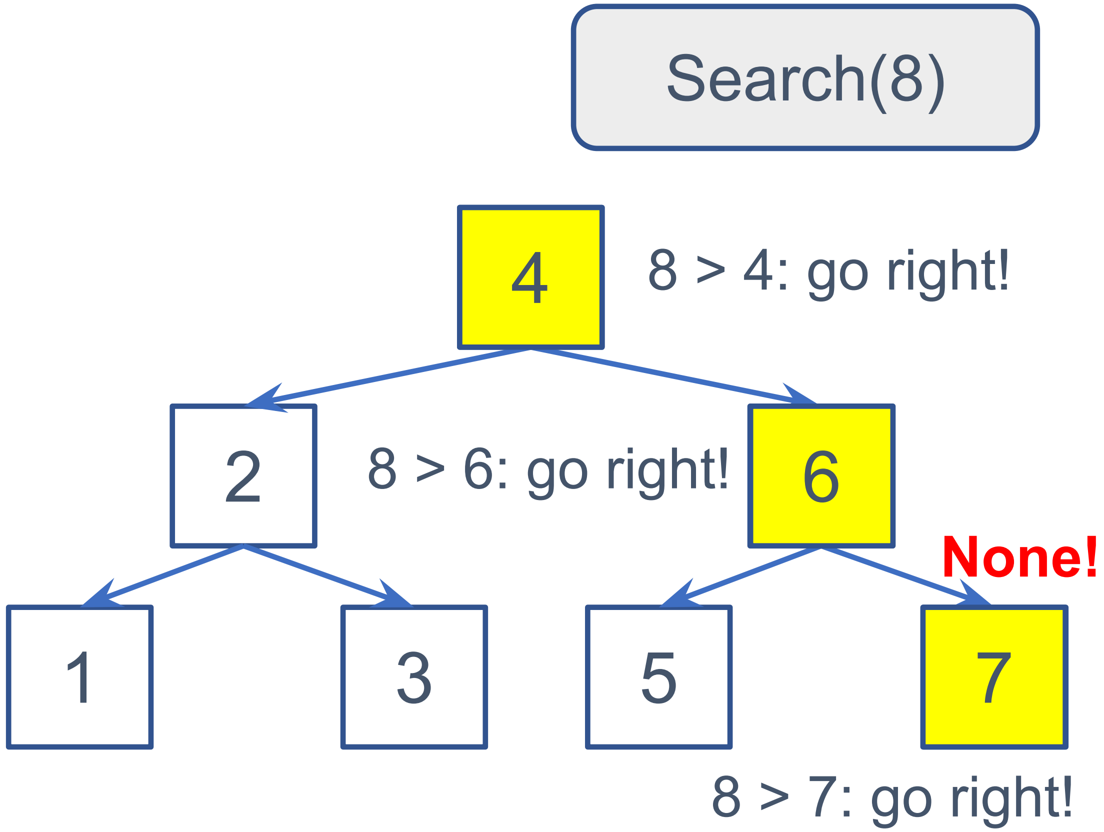
Binary Search Trees – Search

```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



Binary Search Trees – Search

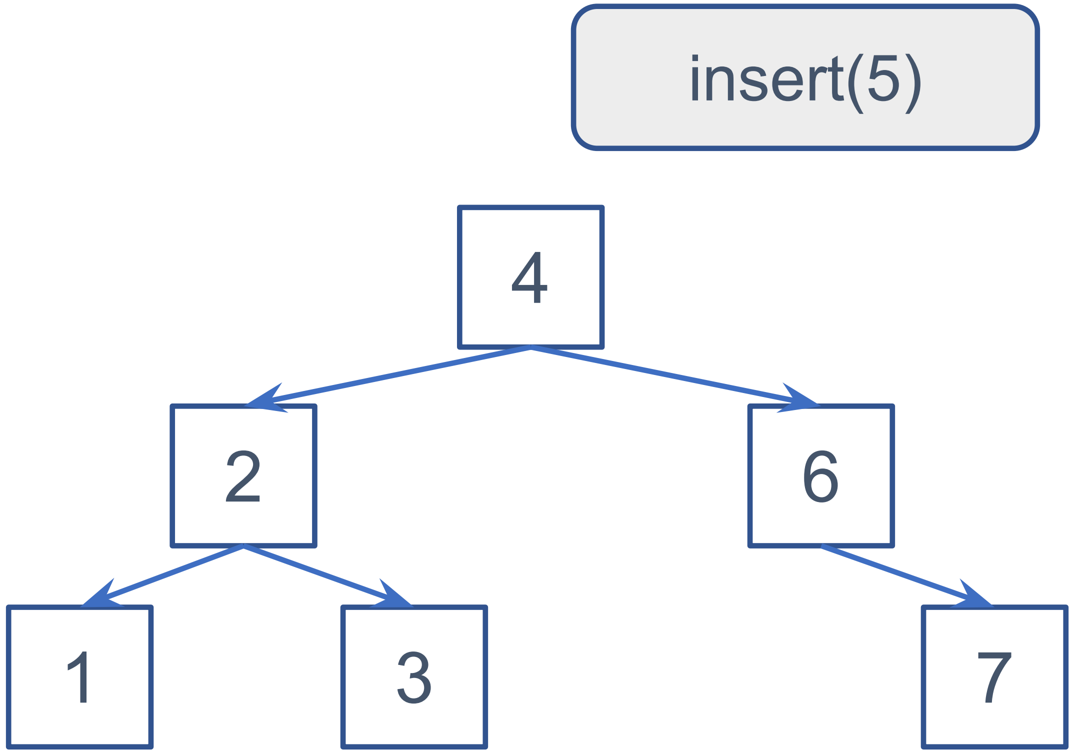
```
class BST():
    def __searchHelp(curNode: TreeNode, x) -> TreeNode:
        if not curNode:
            return None
        if x == curNode.val:
            return curNode
        elif x < curNode.val:
            return self.__searchHelp(curNode.left, x)
        else:
            return self.__searchHelp(curNode.right, x)
```



Insert

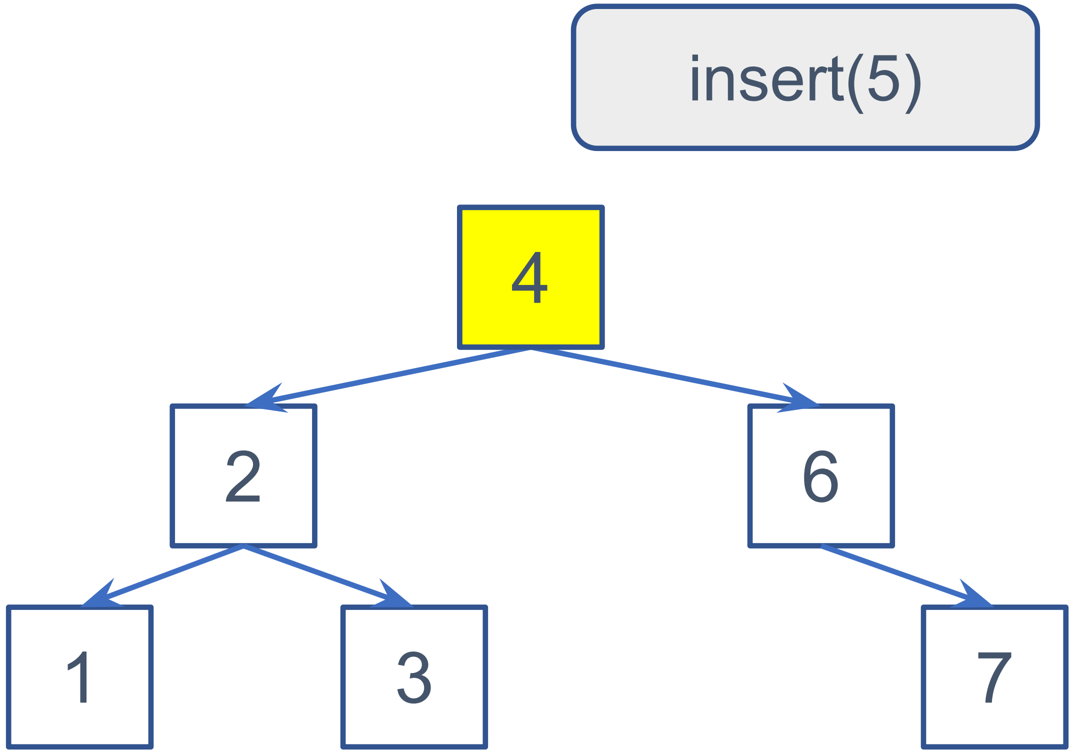
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(curNode: TreeNode, x: int) -> TreeNode:
        if not curNode:
            return TreeNode(x)
        if x < curNode.val:
            curNode.left = self.__insertHelp(curNode.left, x)
        elif x > curNode.val:
            curNode.right = self.__insertHelp(curNode.right, x)
        return curNode

    def insert(self, x: int) -> None:
```



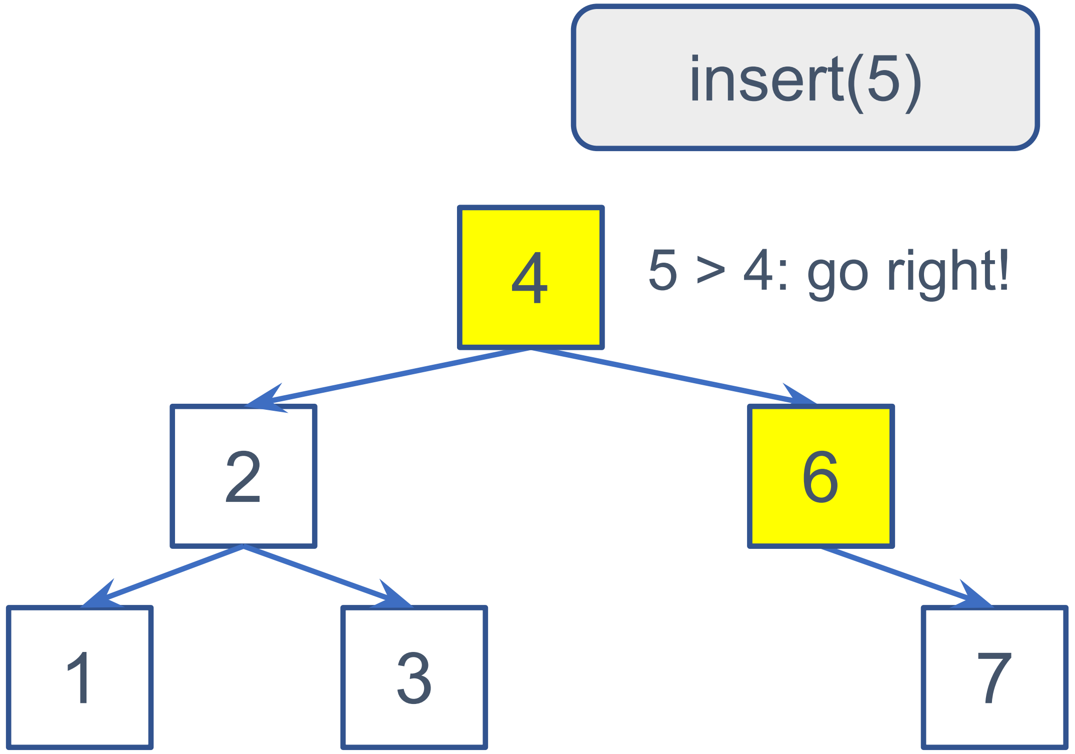
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(curNode: TreeNode, x: int) -> TreeNode:
        if not curNode:
            return TreeNode(x)
        if x < curNode.val:
            curNode.left = self.__insertHelp(curNode.left, x)
        elif x > curNode.val:
            curNode.right = self.__insertHelp(curNode.right, x)
        return curNode

    def insert(self, x: int) -> None:
```



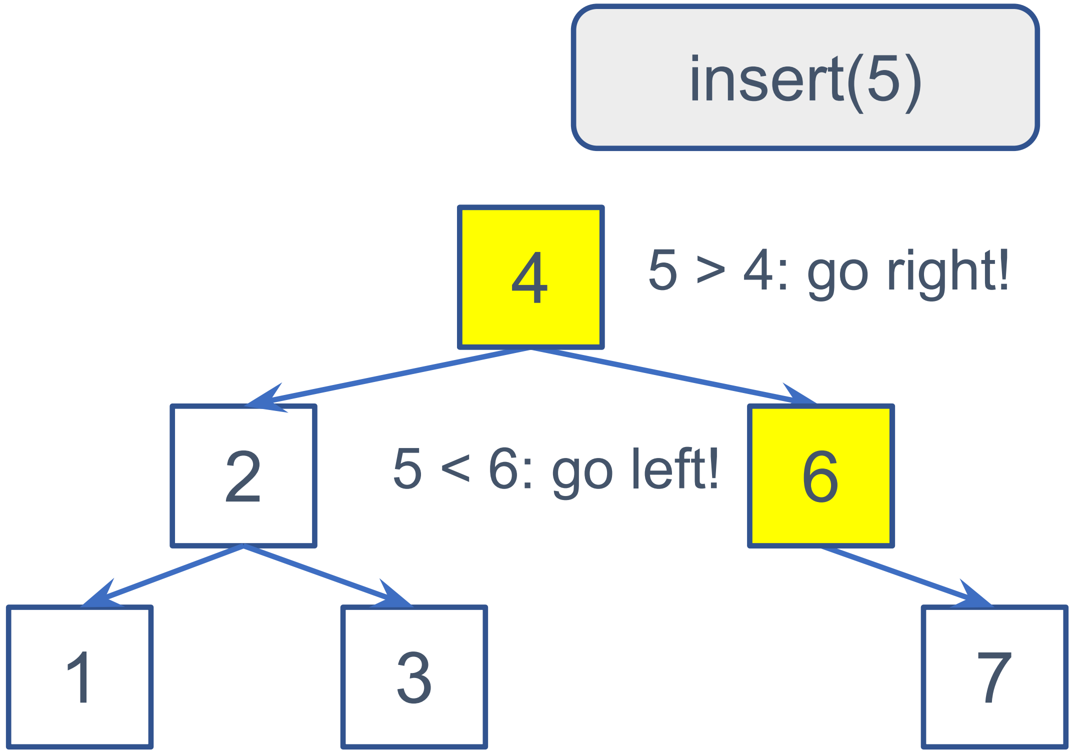
Binary Search Trees – Insert

- class BST():
- def **__insertHelp**(curNode: TreeNode, x: int) -> TreeNode:
- if not curNode:
- return TreeNode(x)
- if x < curNode.val:
- curNode.left = self.__insertHelp(curNode.left, x)
- **elif x > curNode.val:**
- **curNode.right = self.__insertHelp(curNode.right, x)**
- return curNode
- def **insert**(self, x: int) -> None:



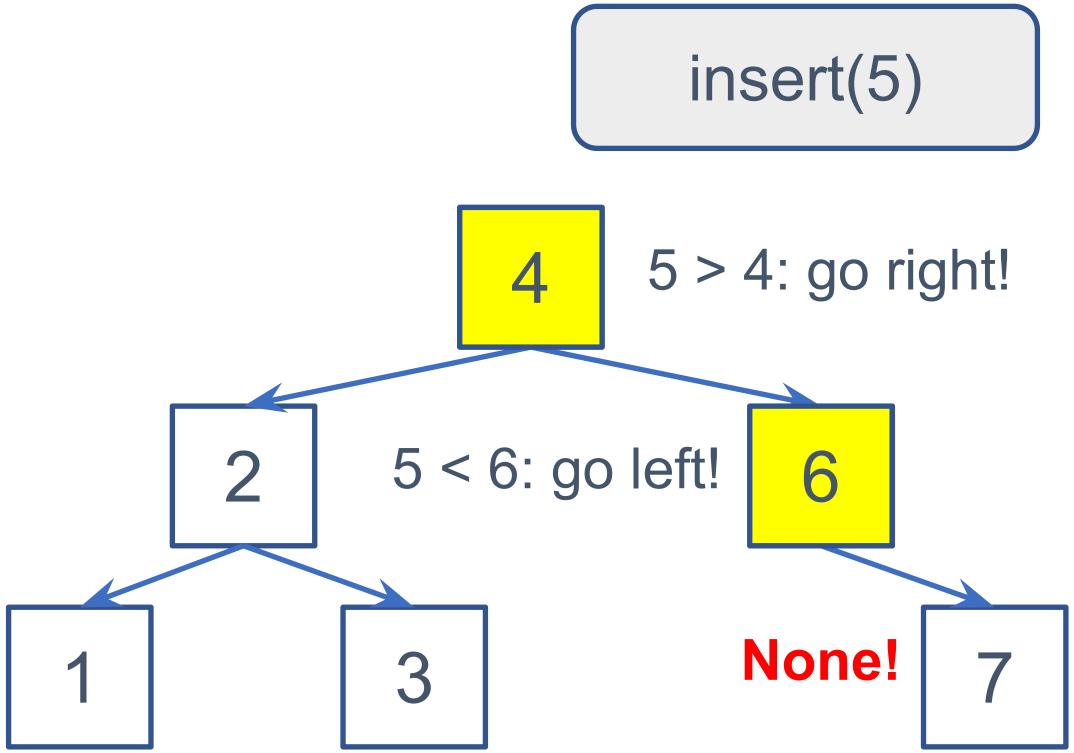
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(curNode: TreeNode, x: int) -> TreeNode:
        if not curNode:
            return TreeNode(x)
        if x < curNode.val:
            curNode.left = self.__insertHelp(curNode.left, x)
        elif x > curNode.val:
            curNode.right = self.__insertHelp(curNode.right, x)
        return curNode

    def insert(self, x: int) -> None:
```



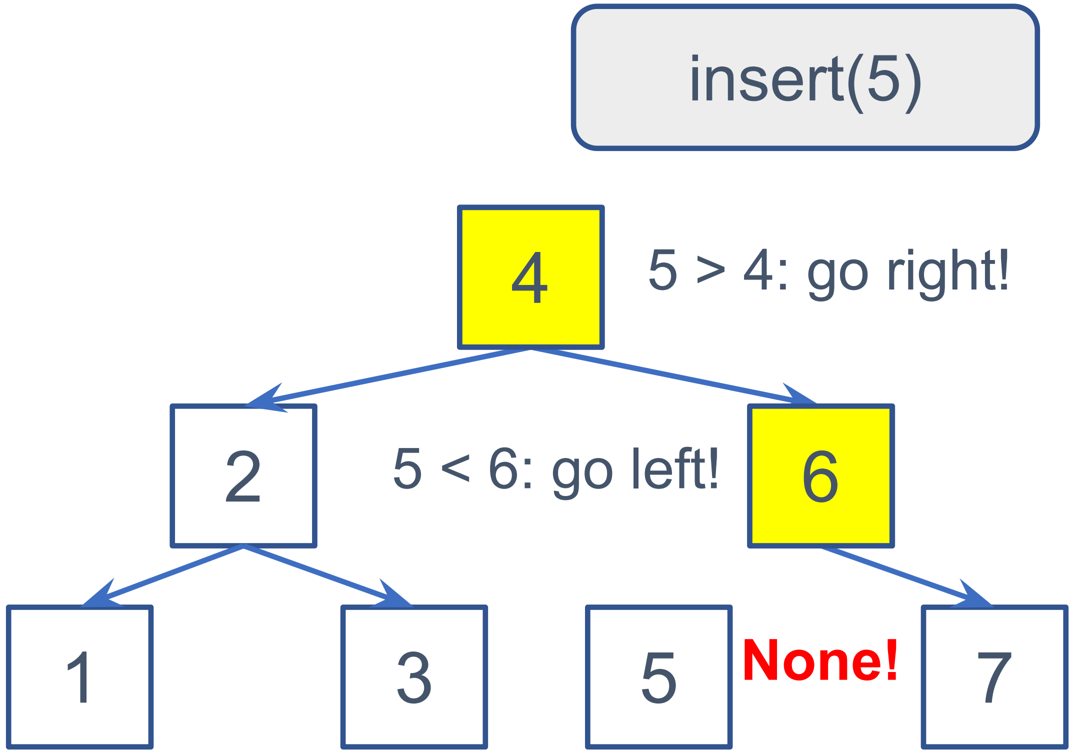
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(curNode: TreeNode, x: int) -> TreeNode:
        if not curNode:
            return TreeNode(x)
        if x < curNode.val:
            curNode.left = self.__insertHelp(curNode.left, x)
        elif x > curNode.val:
            curNode.right = self.__insertHelp(curNode.right, x)
        return curNode

    def insert(self, x: int) -> None:
```



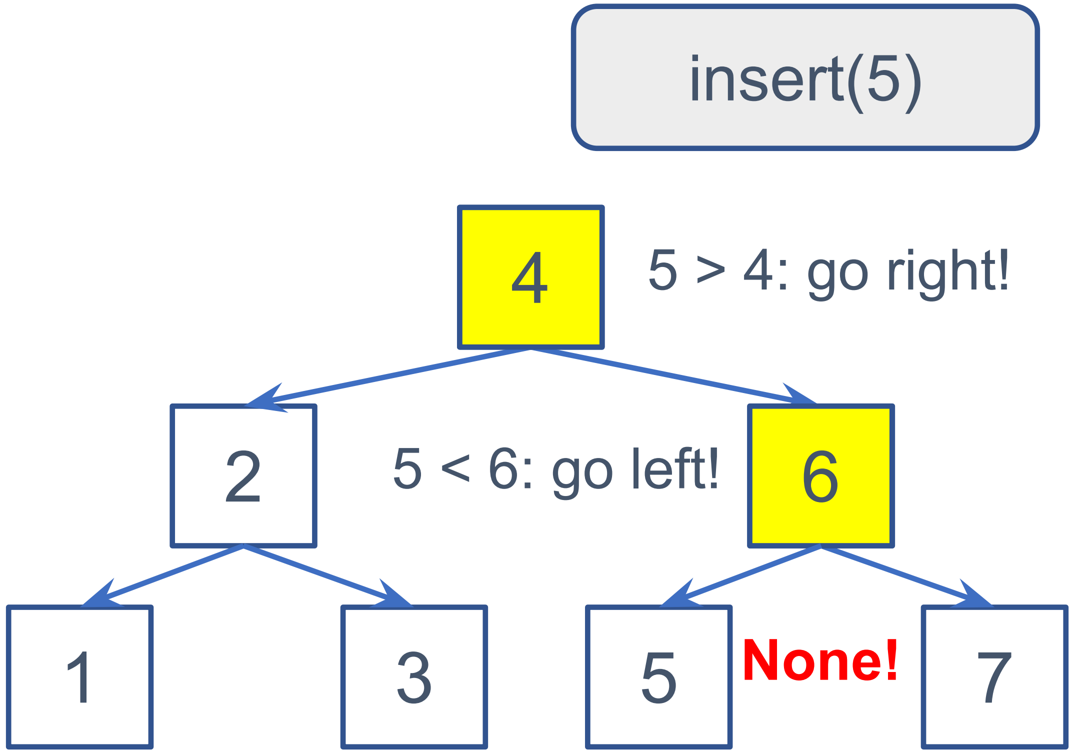
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(curNode: TreeNode, x: int) -> TreeNode:
        if not curNode:
            return TreeNode(x)
        if x < curNode.val:
            curNode.left = self.__insertHelp(curNode.left, x)
        elif x > curNode.val:
            curNode.right = self.__insertHelp(curNode.right, x)
        return curNode

    def insert(self, x: int) -> None:
```



Binary Search Trees – Insert

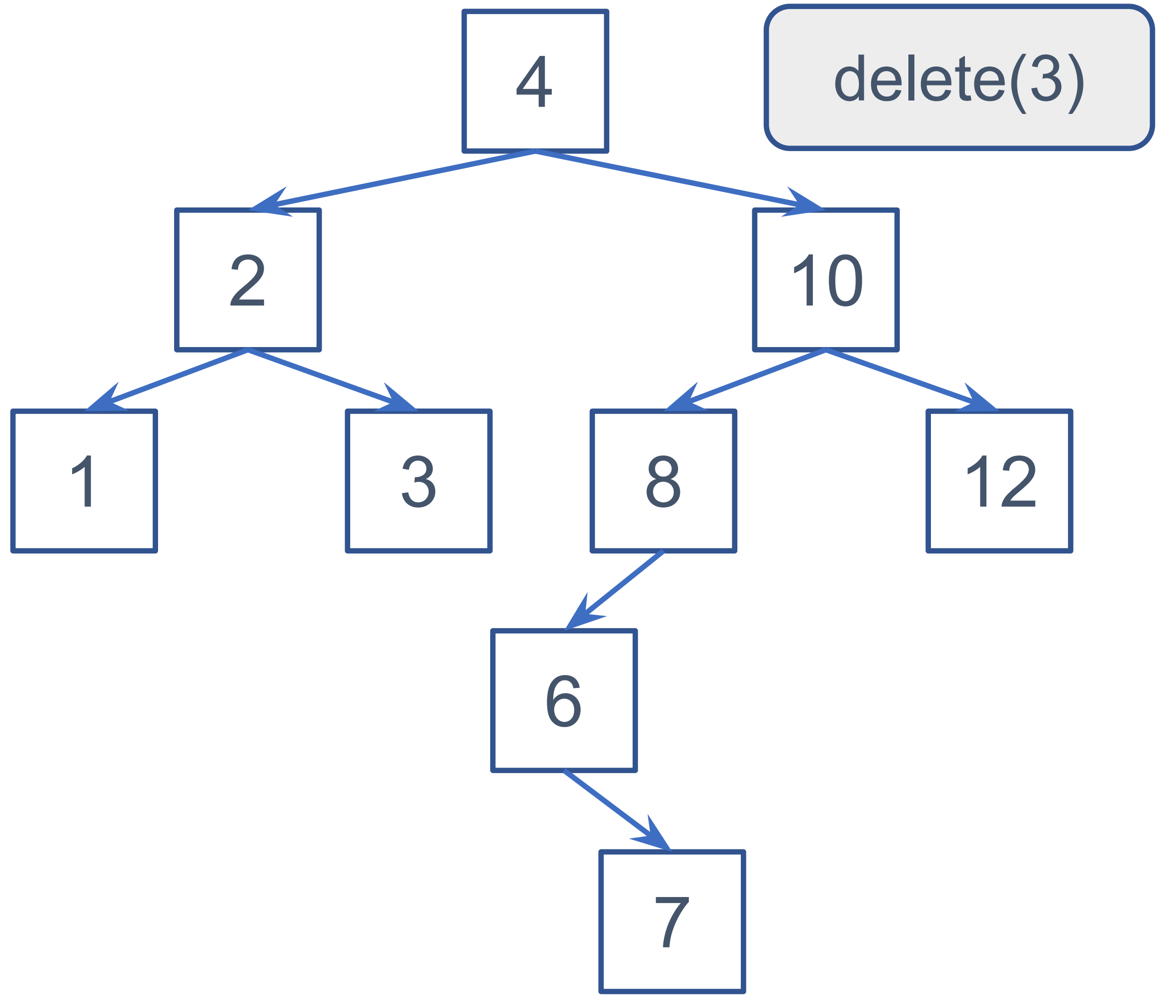
```
class BST():
    def __insertHelp(curNode: TreeNode, x: int) -> TreeNode:
        if not curNode:
            return TreeNode(x)
        if x < curNode.val:
            curNode.left = self.__insertHelp(curNode.left, x)
        elif x > curNode.val:
            curNode.right = self.__insertHelp(curNode.right, x)
        return curNode

    def insert(self, x: int) -> None:
```



Delete

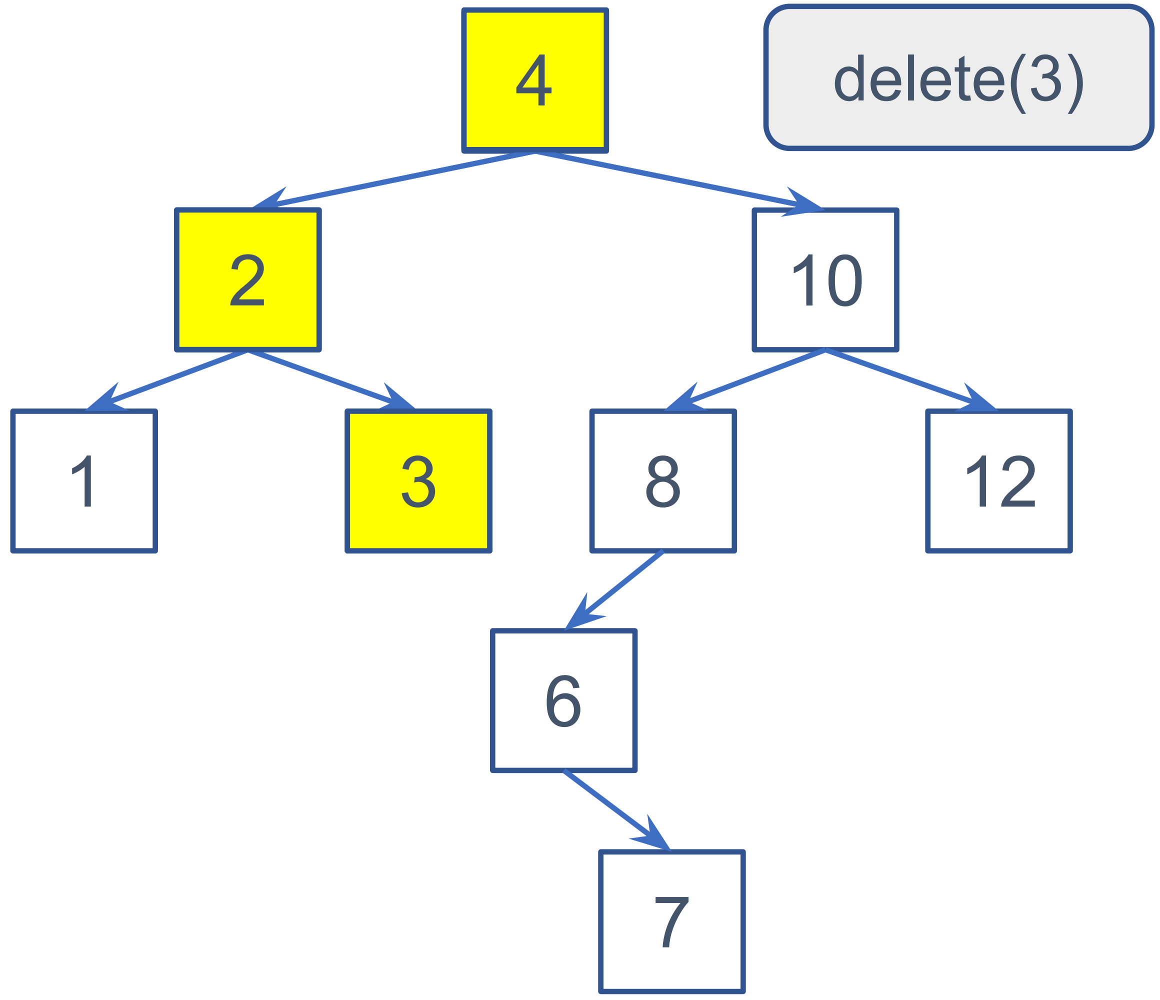
Binary Search Trees – Delete

- **Case 1**: Delete a **leaf** node (no child)



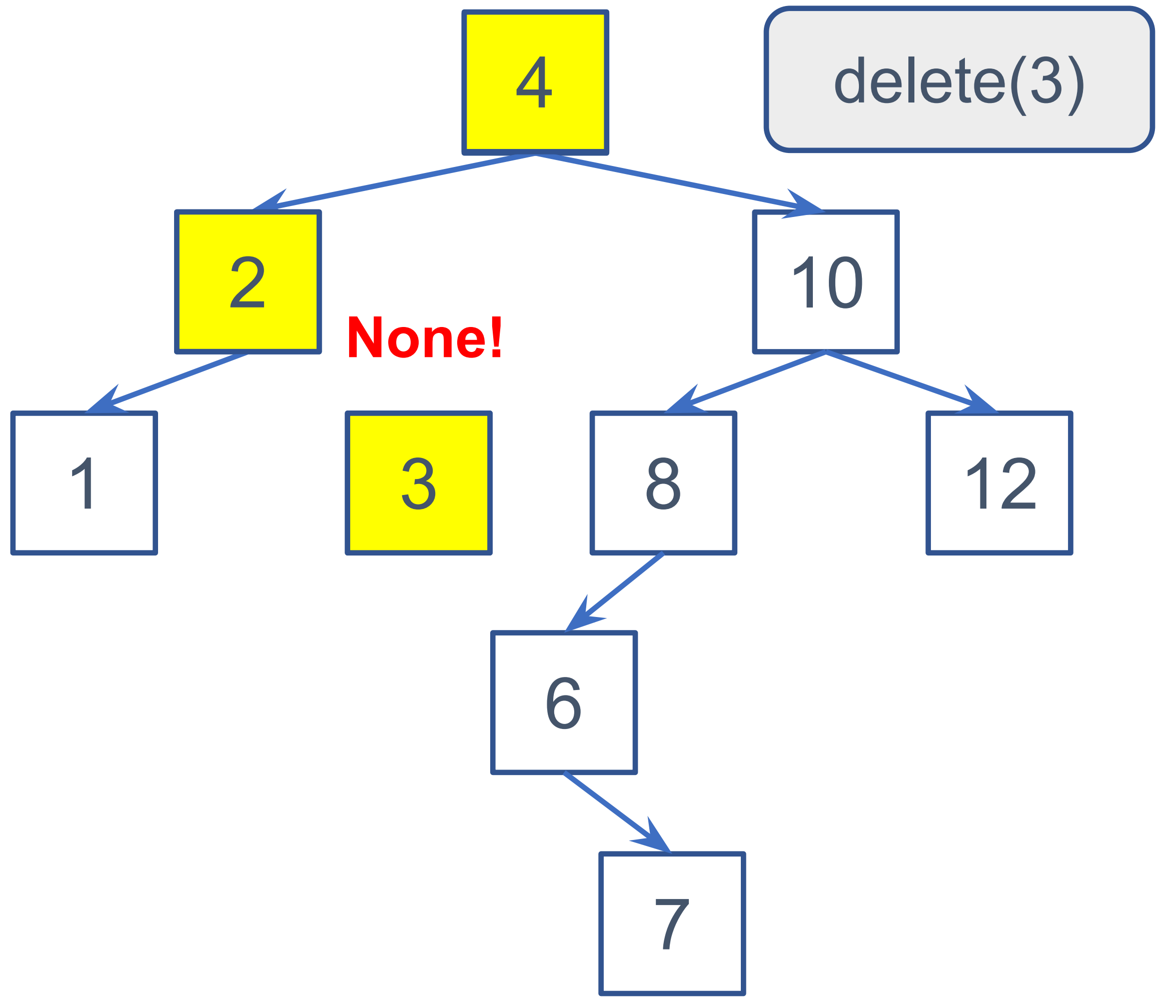
Binary Search Trees – Delete

- **Case 1**: Delete a **leaf** node (no child)
  - **Search** the node using its key value



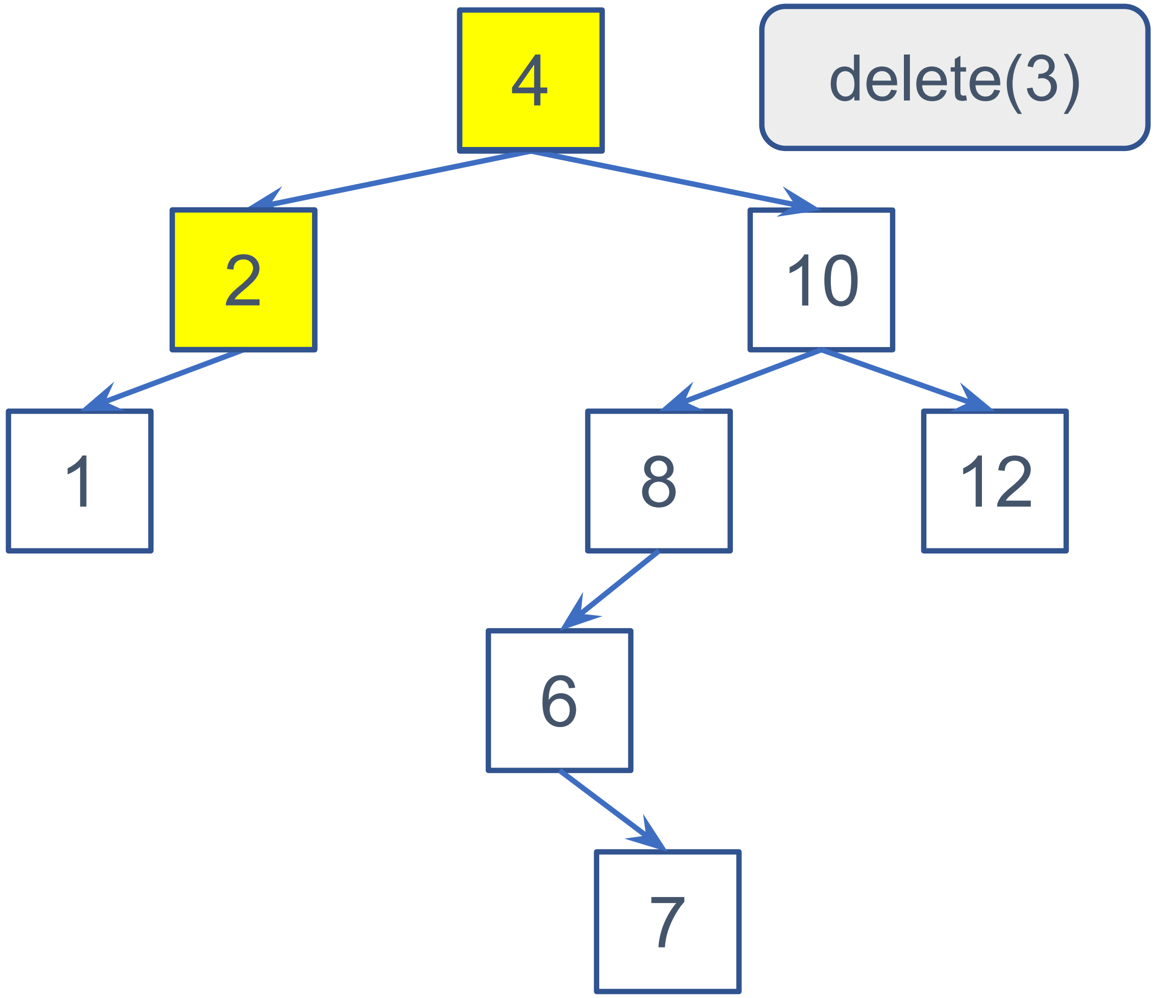
Binary Search Trees – Delete

- **Case 1**: Delete a **leaf** node (no child)
  - **Search** the node using its key value
  - Simply **cut** the parent's link



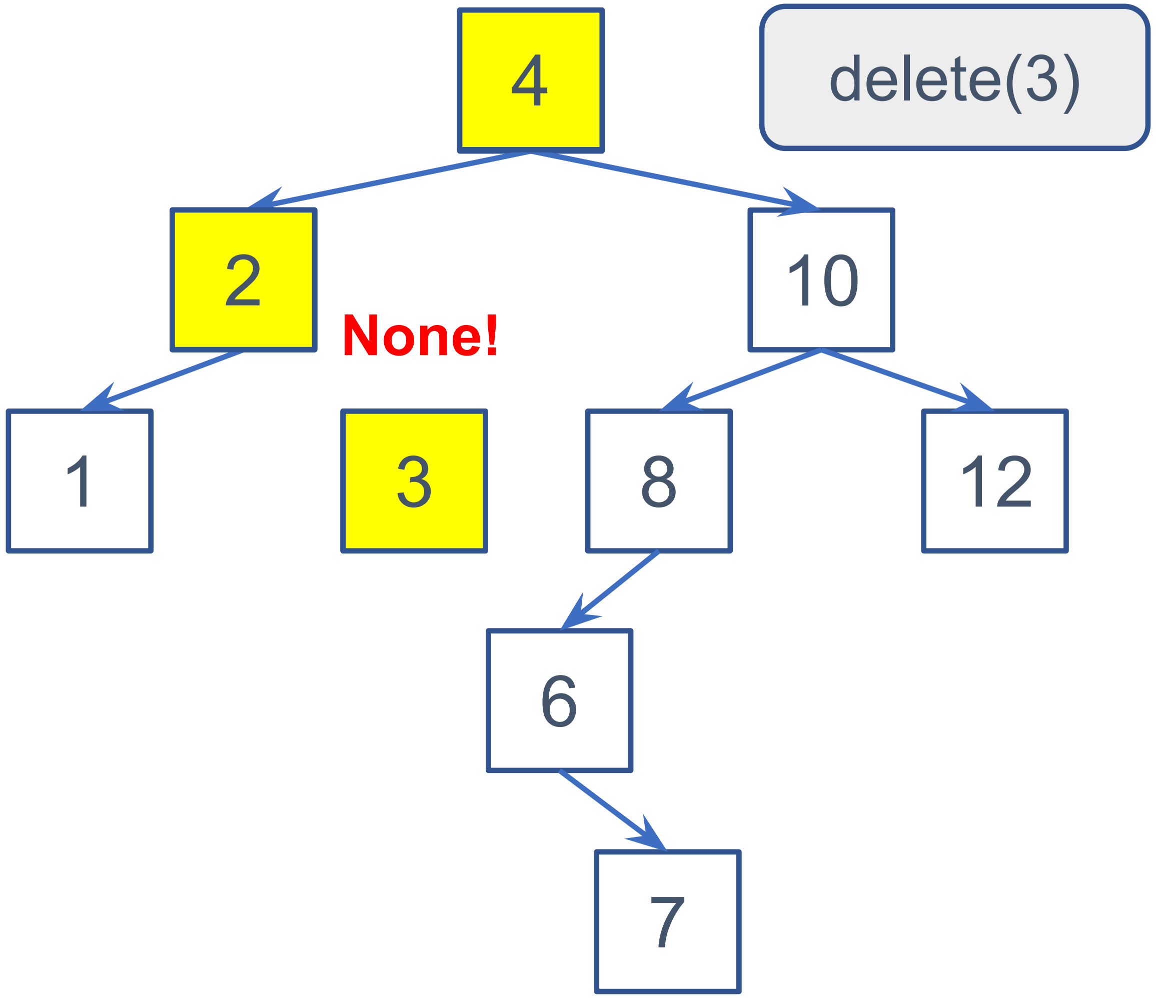
Binary Search Trees – Delete

- **Case 1**: Delete a **leaf** node (no child)
  - **Search** the node using its key value
  - Simply **cut** the parent's link
  - Then the target node is gone



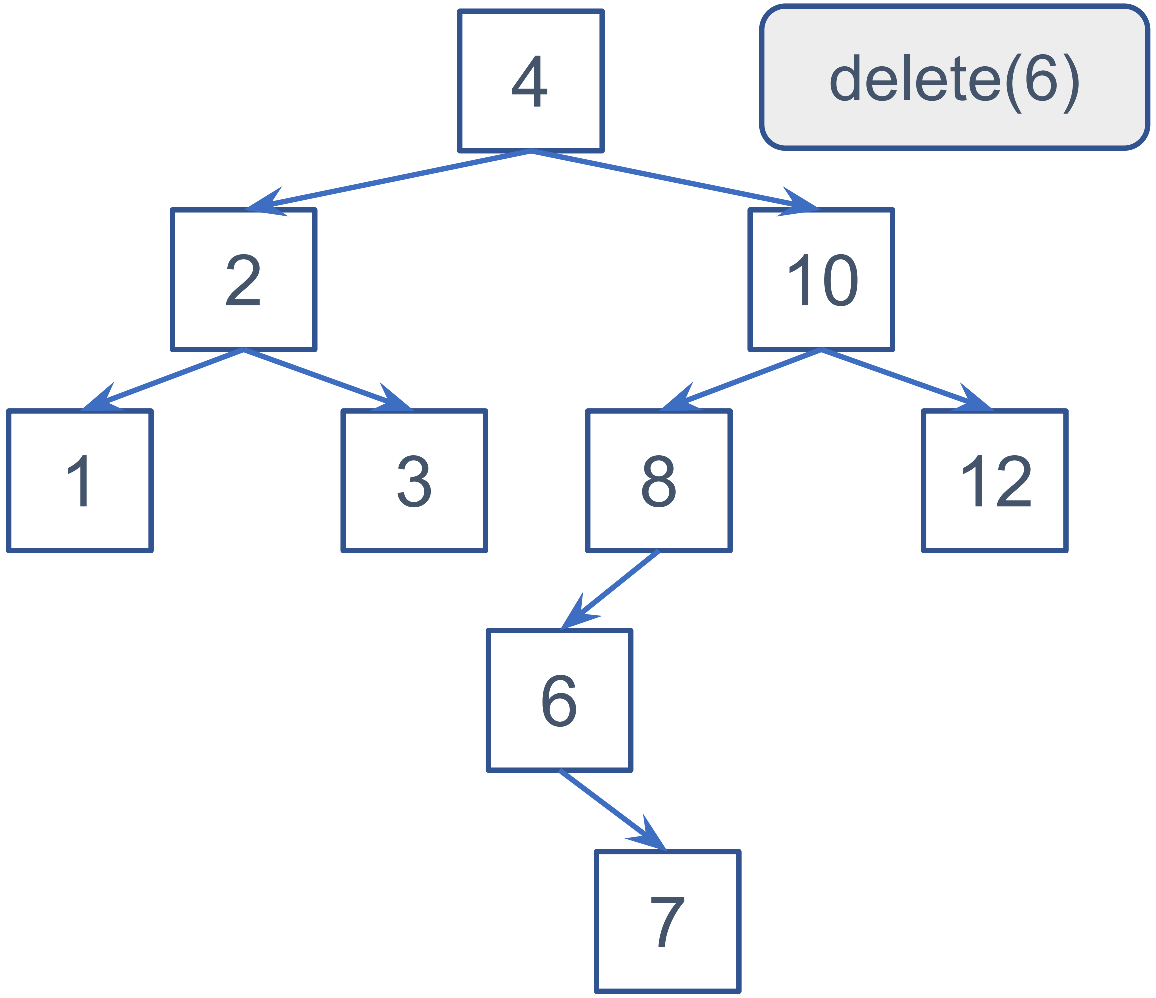
Binary Search Trees – Delete

- **Case 1**: Delete a **leaf** node (no child)
    - **Search** the node using its key value
    - Simply **cut** the parent's link
    - Then the target node is gone



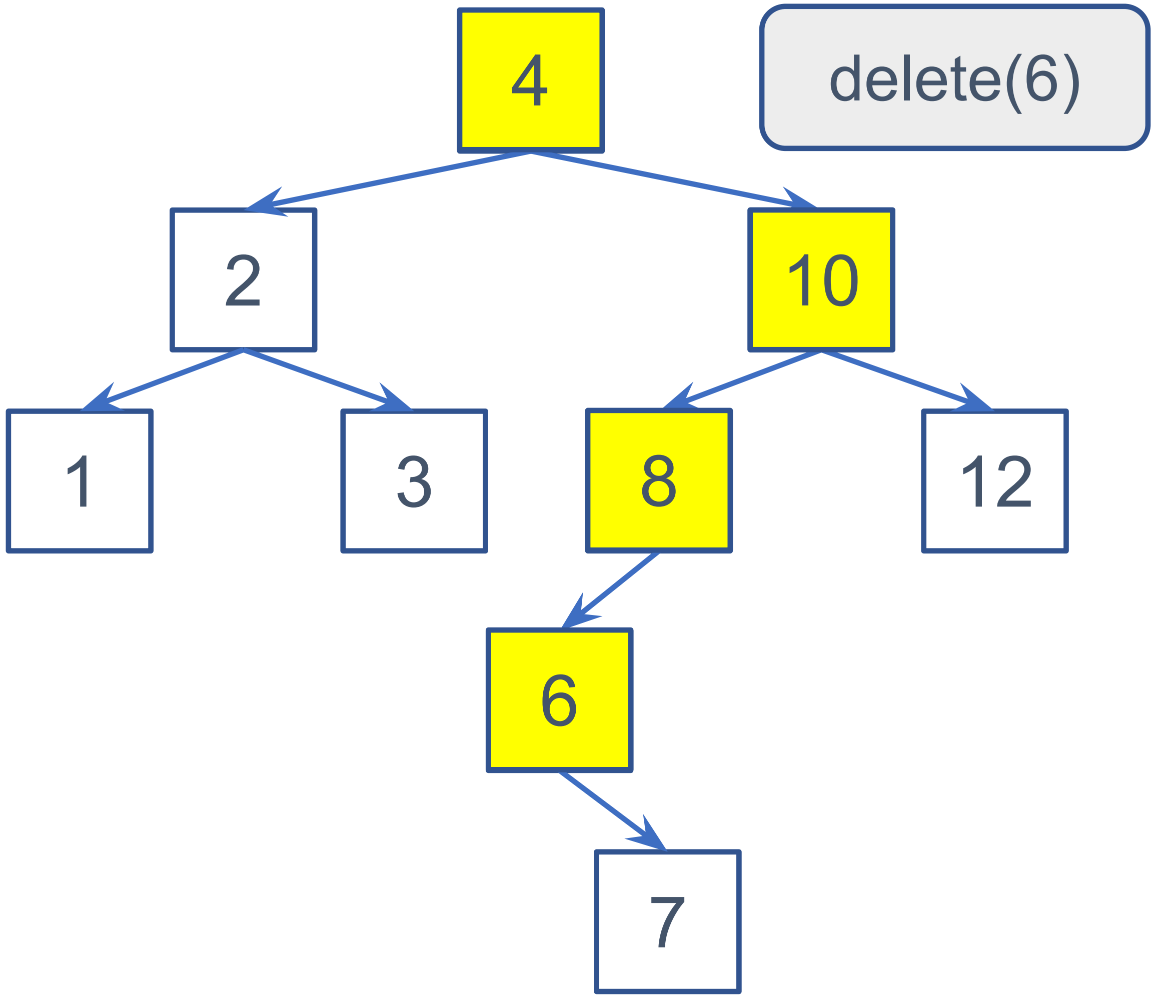
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**



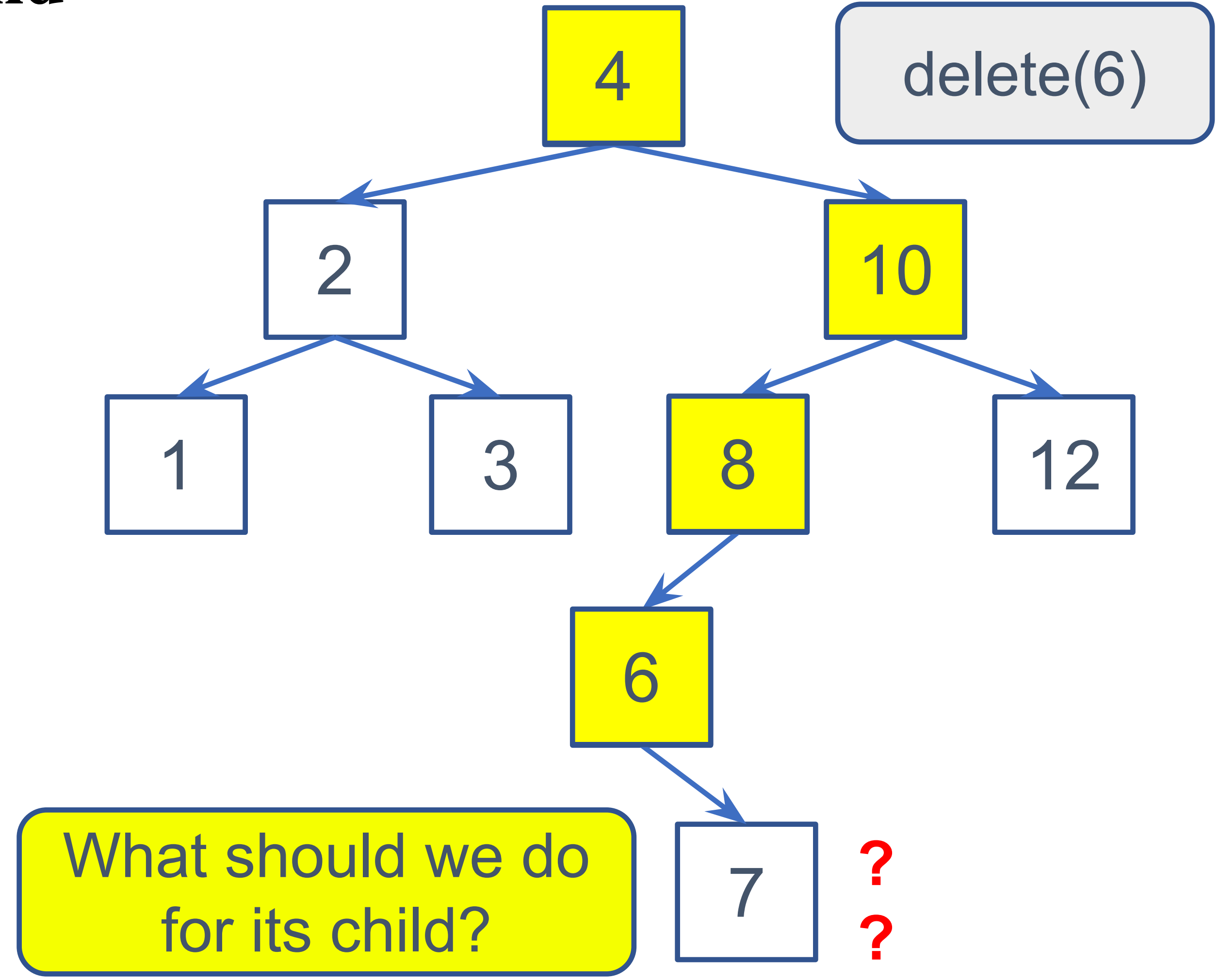
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value



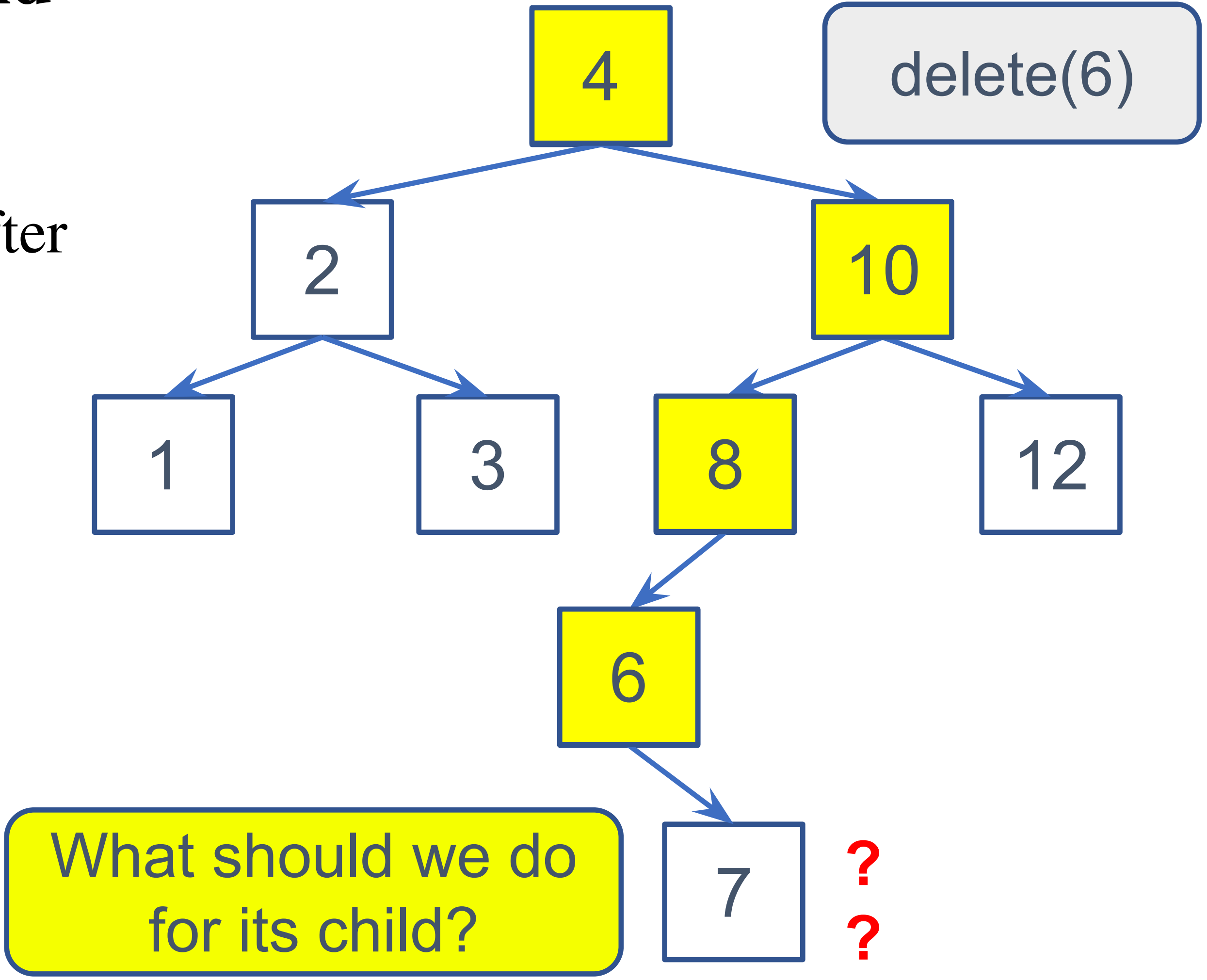
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value



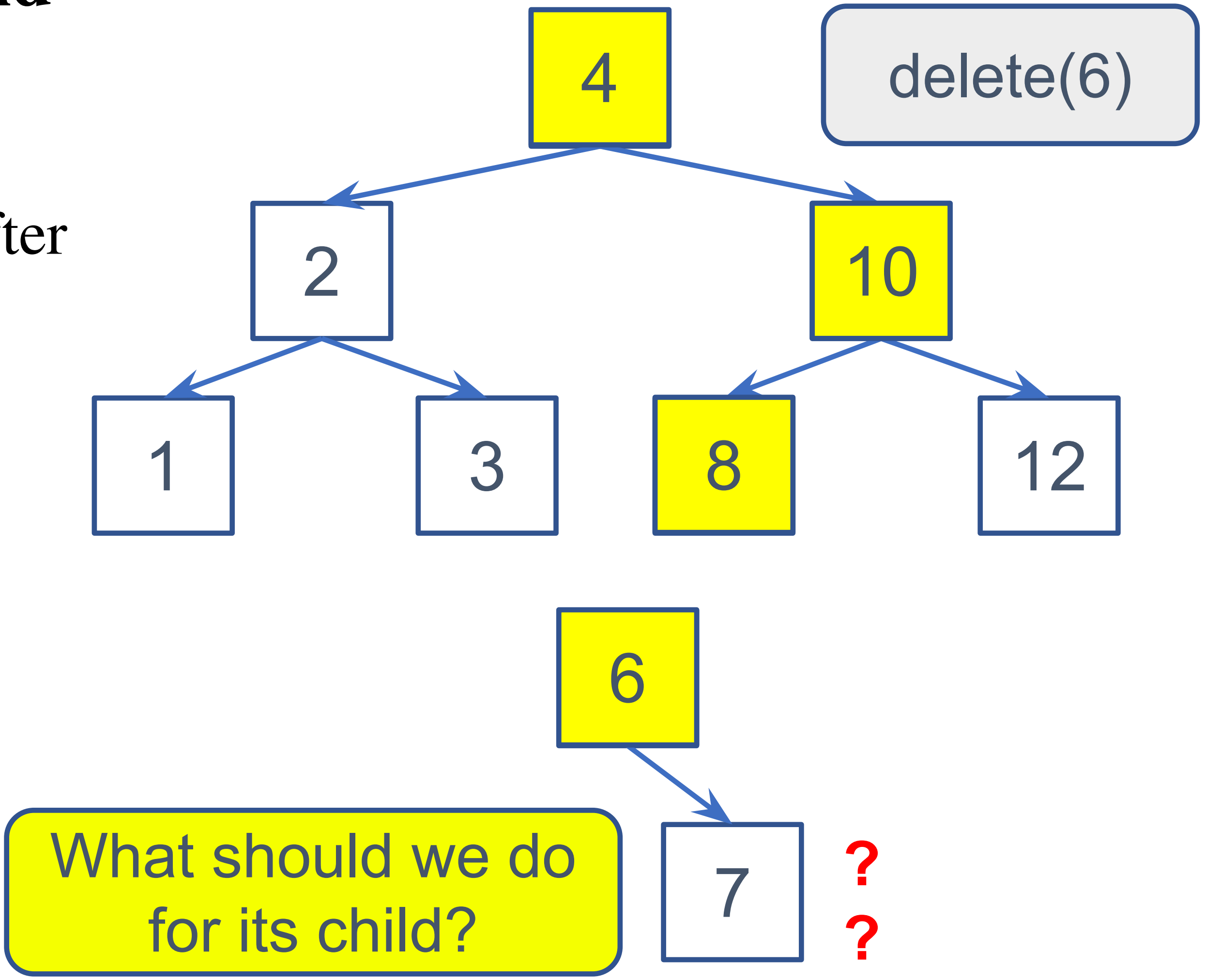
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node



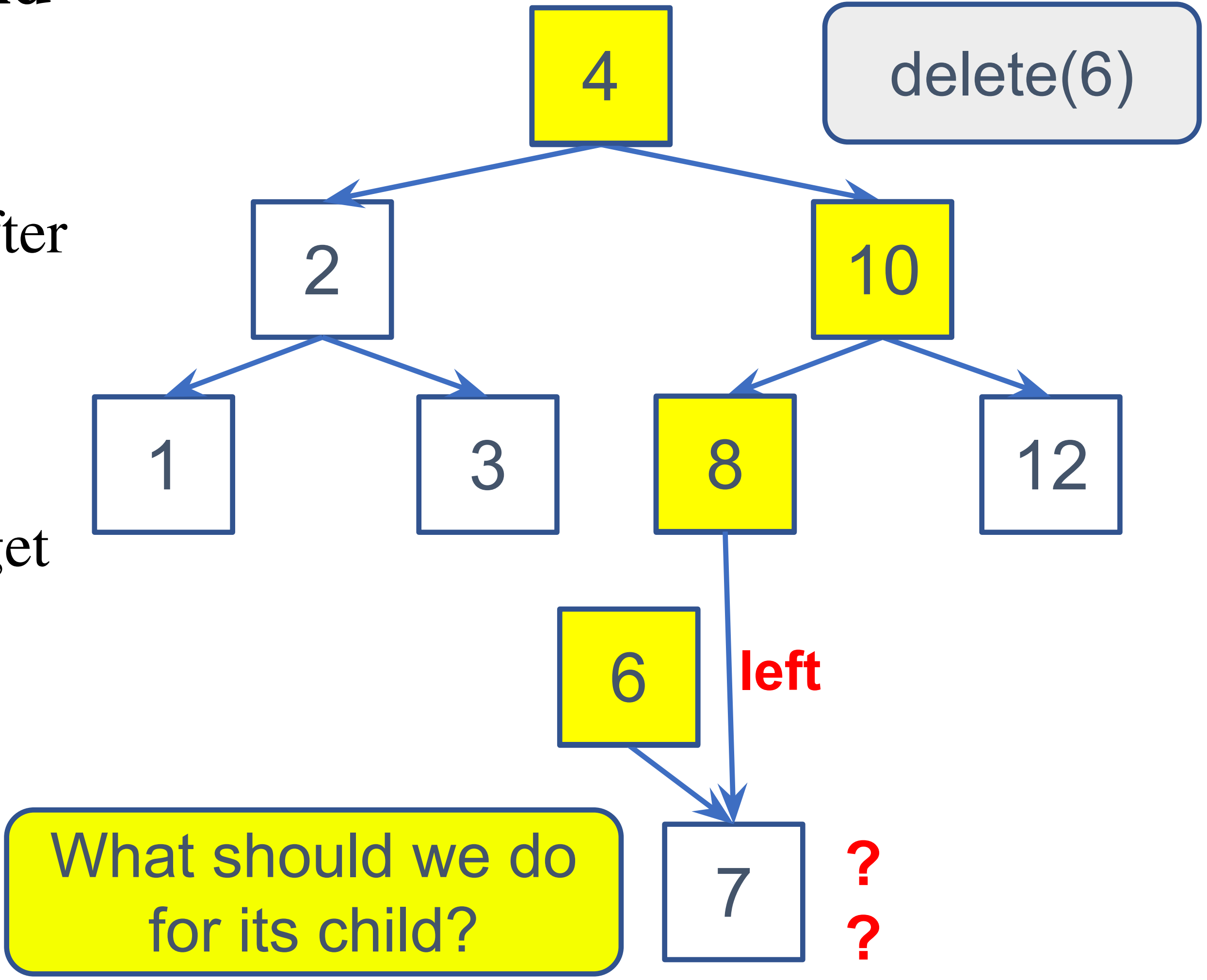
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
  - Cut the parent's link to the target



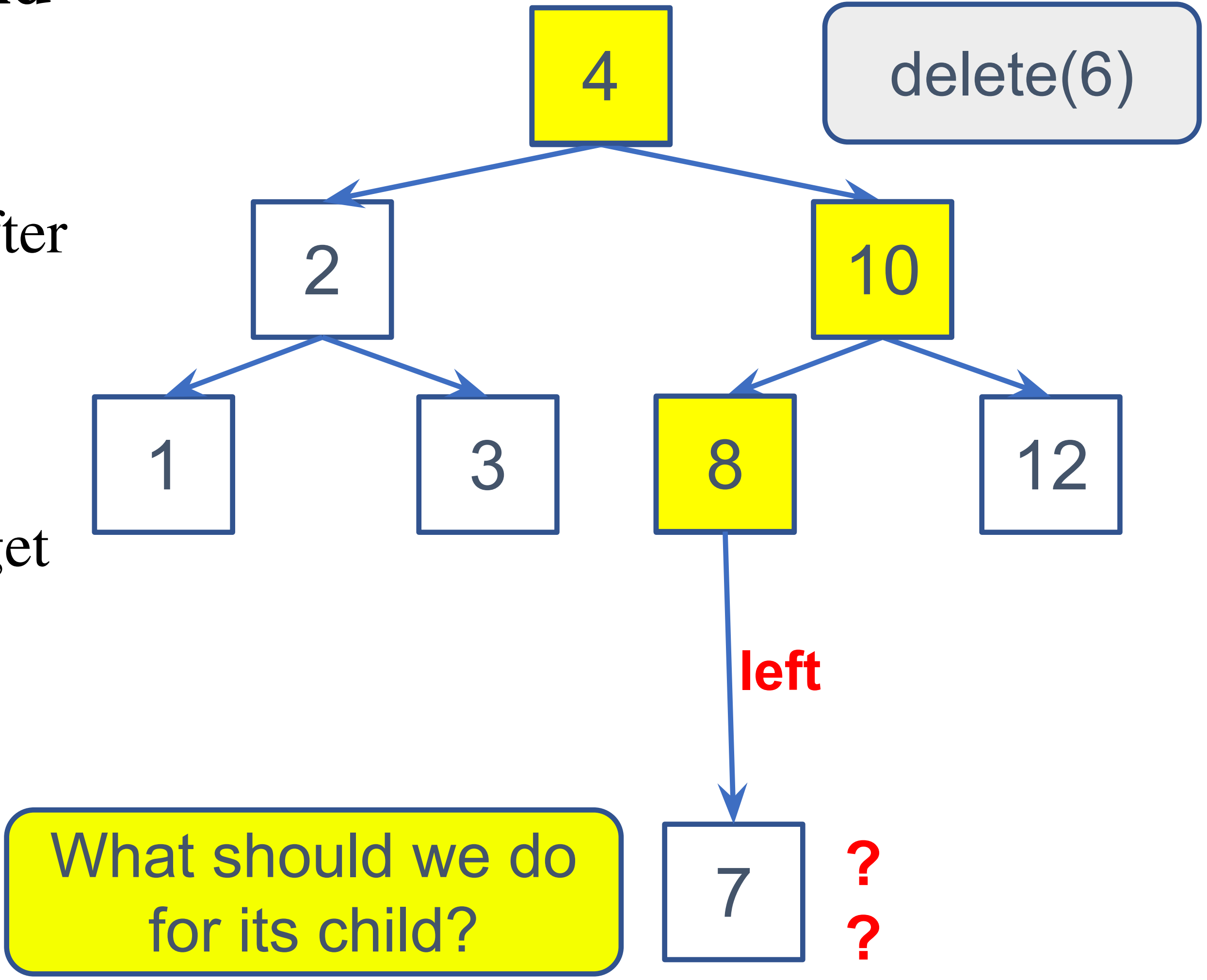
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
  - Cut the parent's link to the target
  - Move the child node to where the target node was



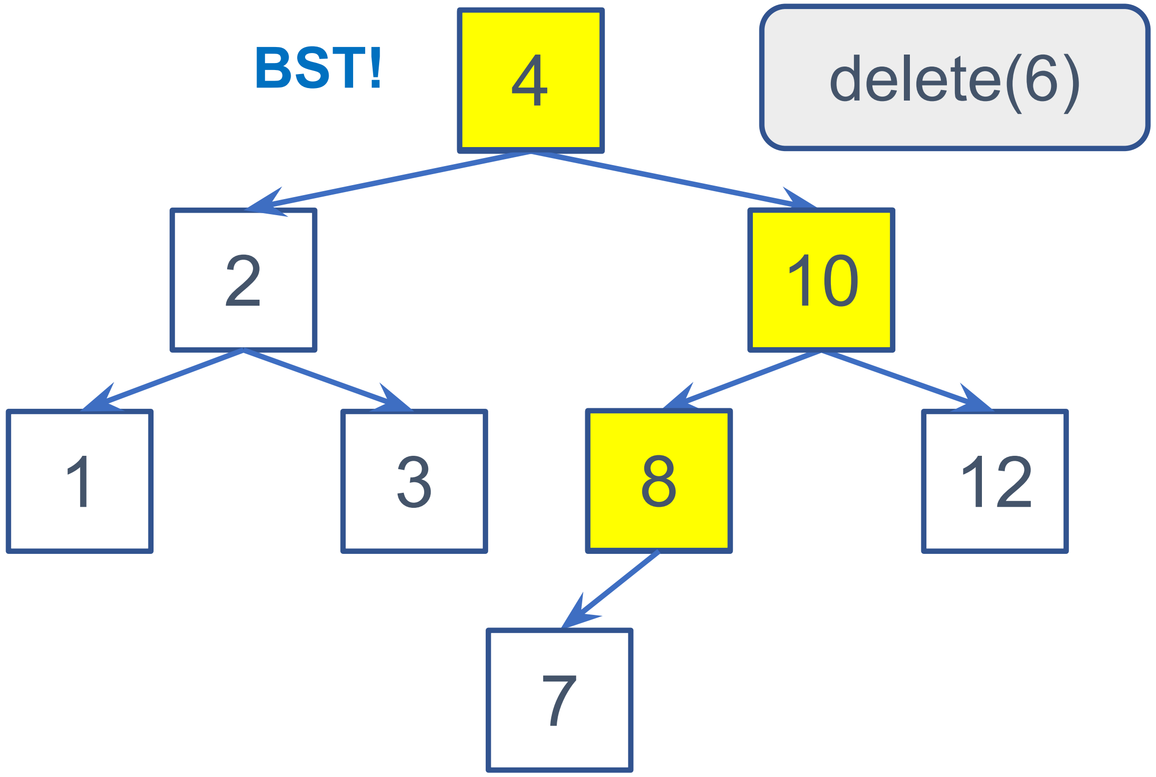
Binary Search Trees – Delete

- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
  - Cut the parent's link to the target
  - Move the child node to where the target node was
  - Then the target node is gone



Binary Search Trees – Delete

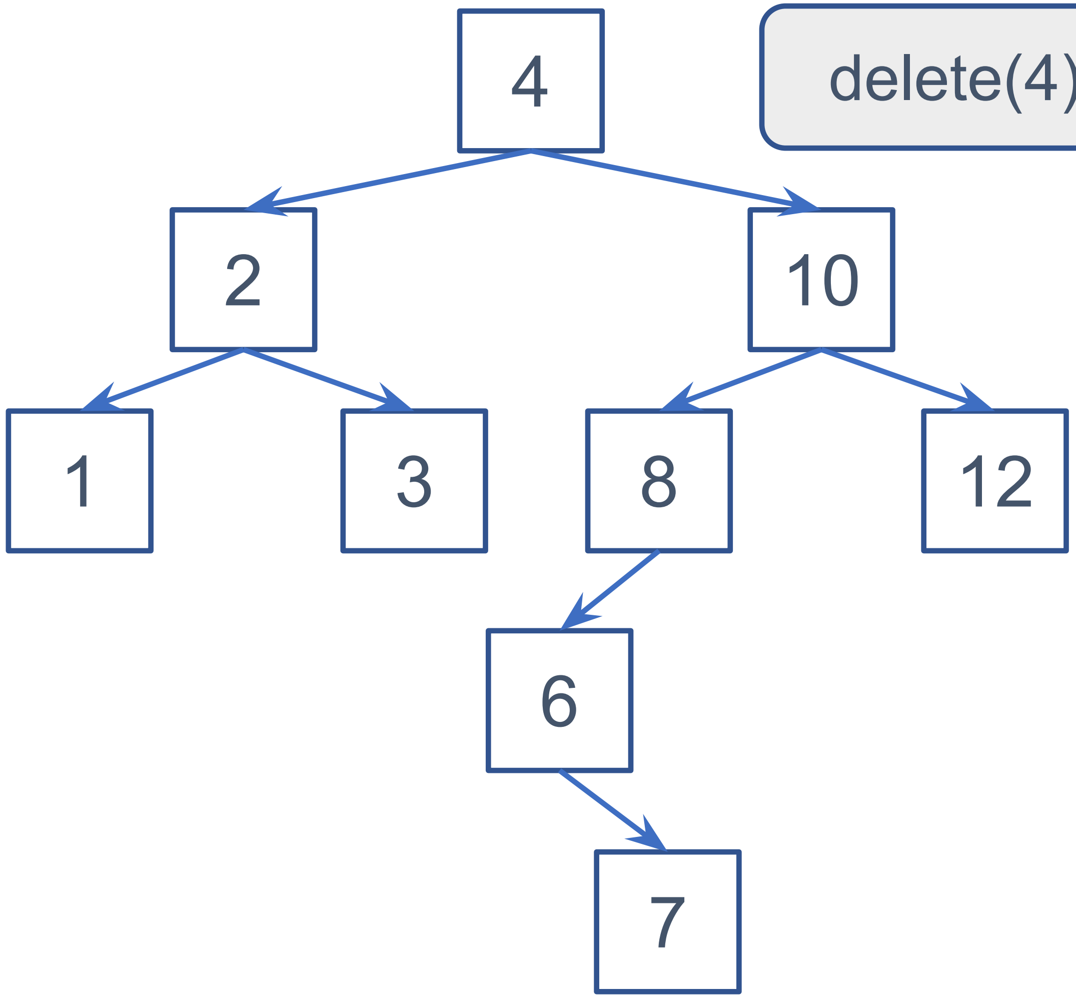
- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
  - Cut the parent's link to the target
  - Move the child node to where the target node was
  - Then the target node is gone



Binary Search Trees – Delete

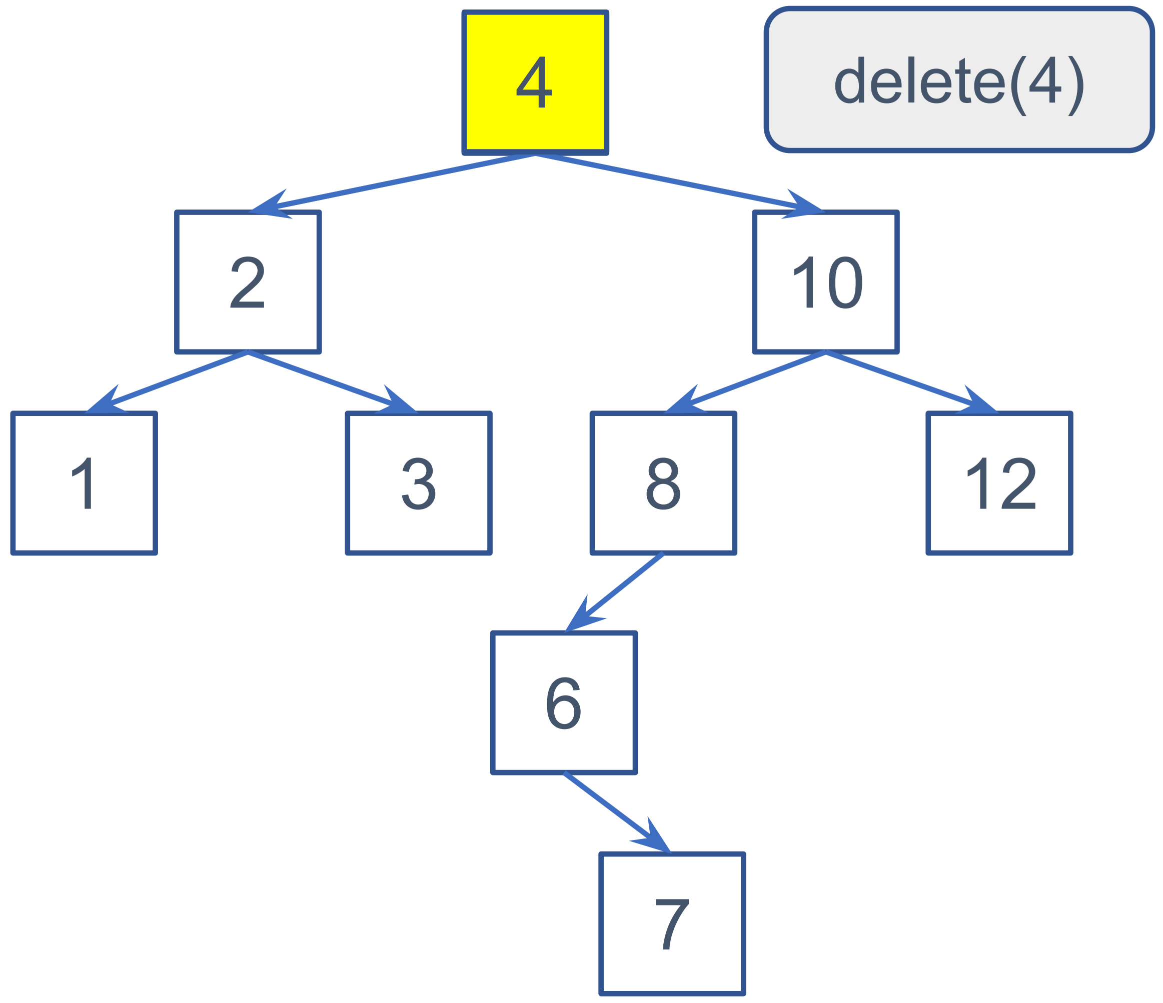
- **Case 3**: Delete a node with **two children**

delete(4)



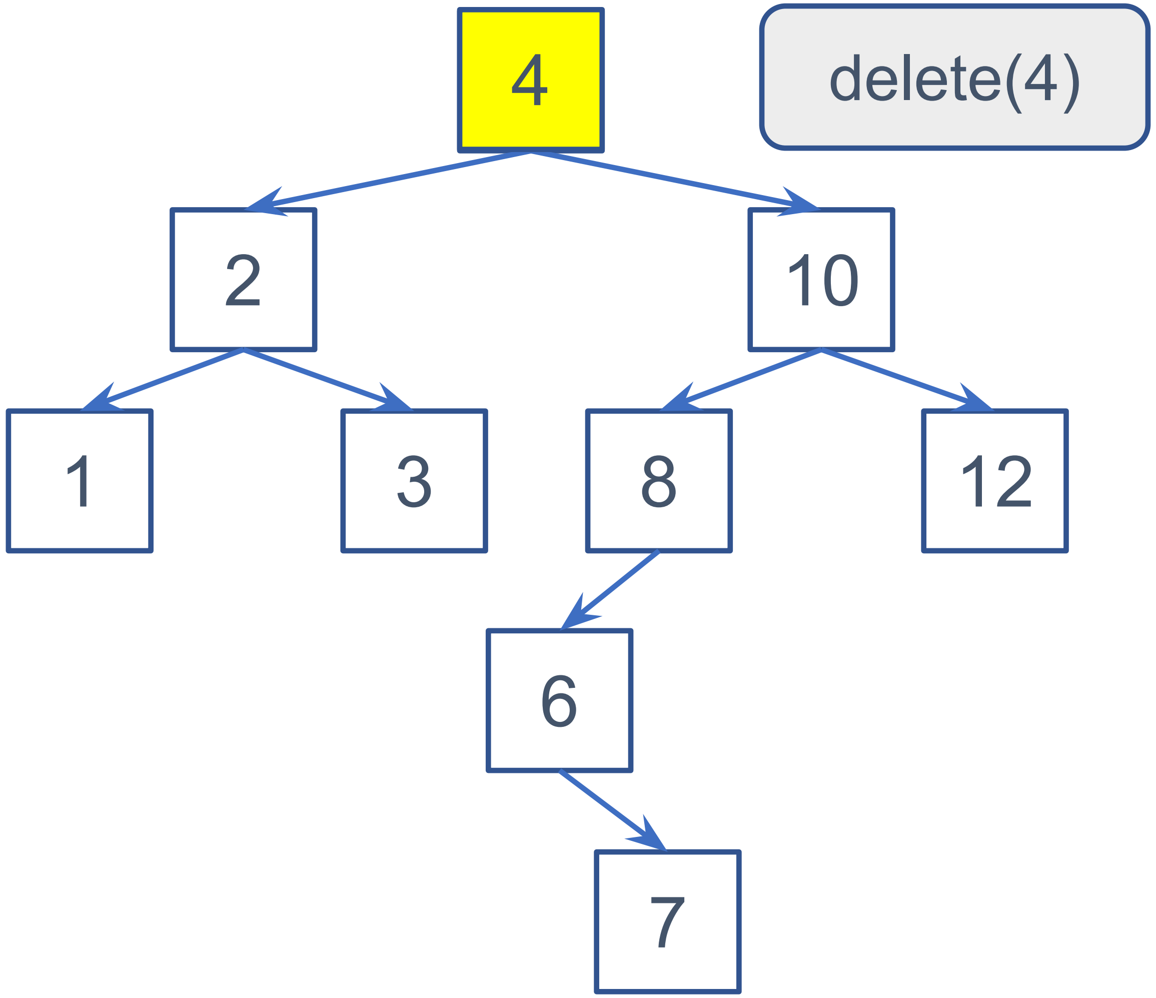
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value



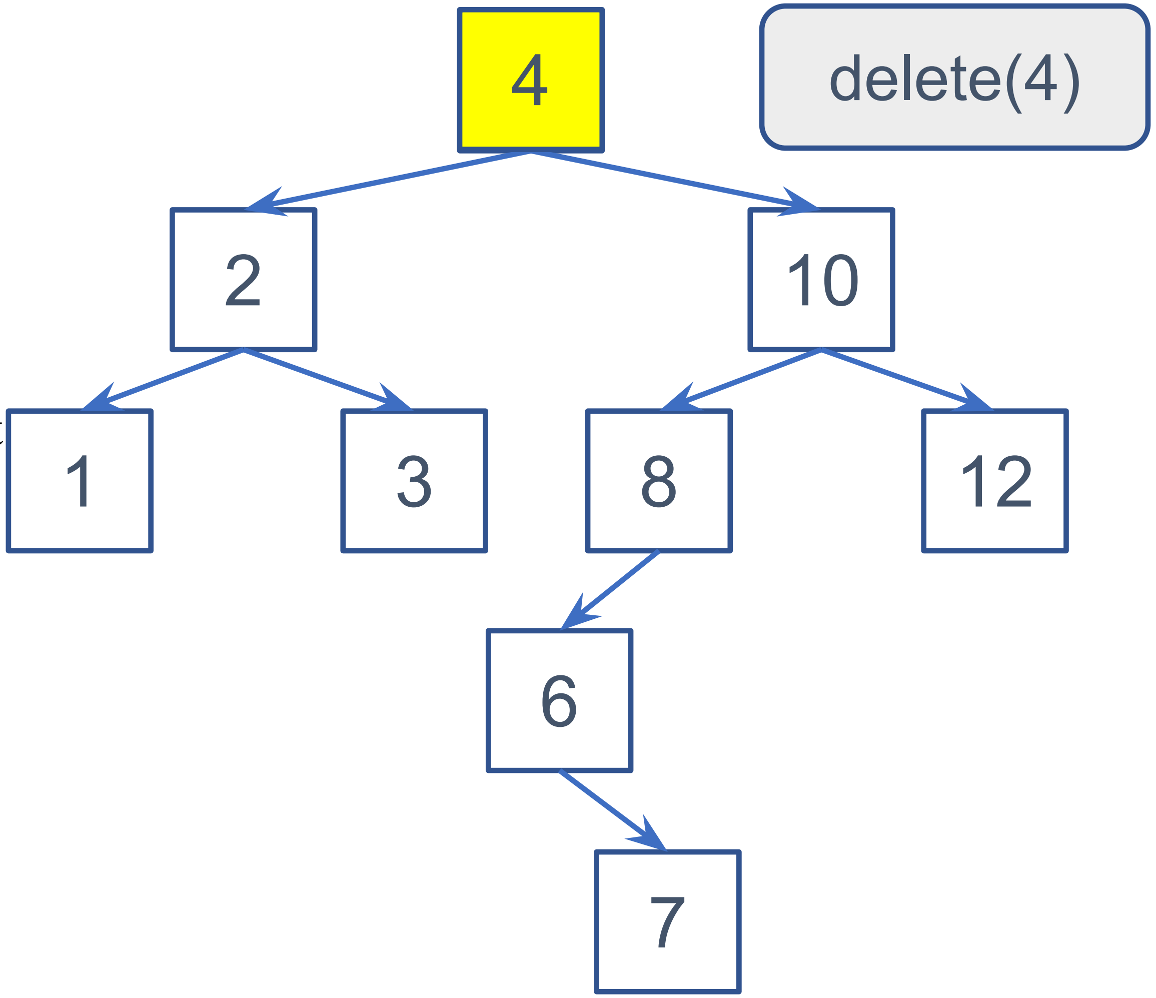
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
    - Find a subtree **node** that can be located at the target node's location



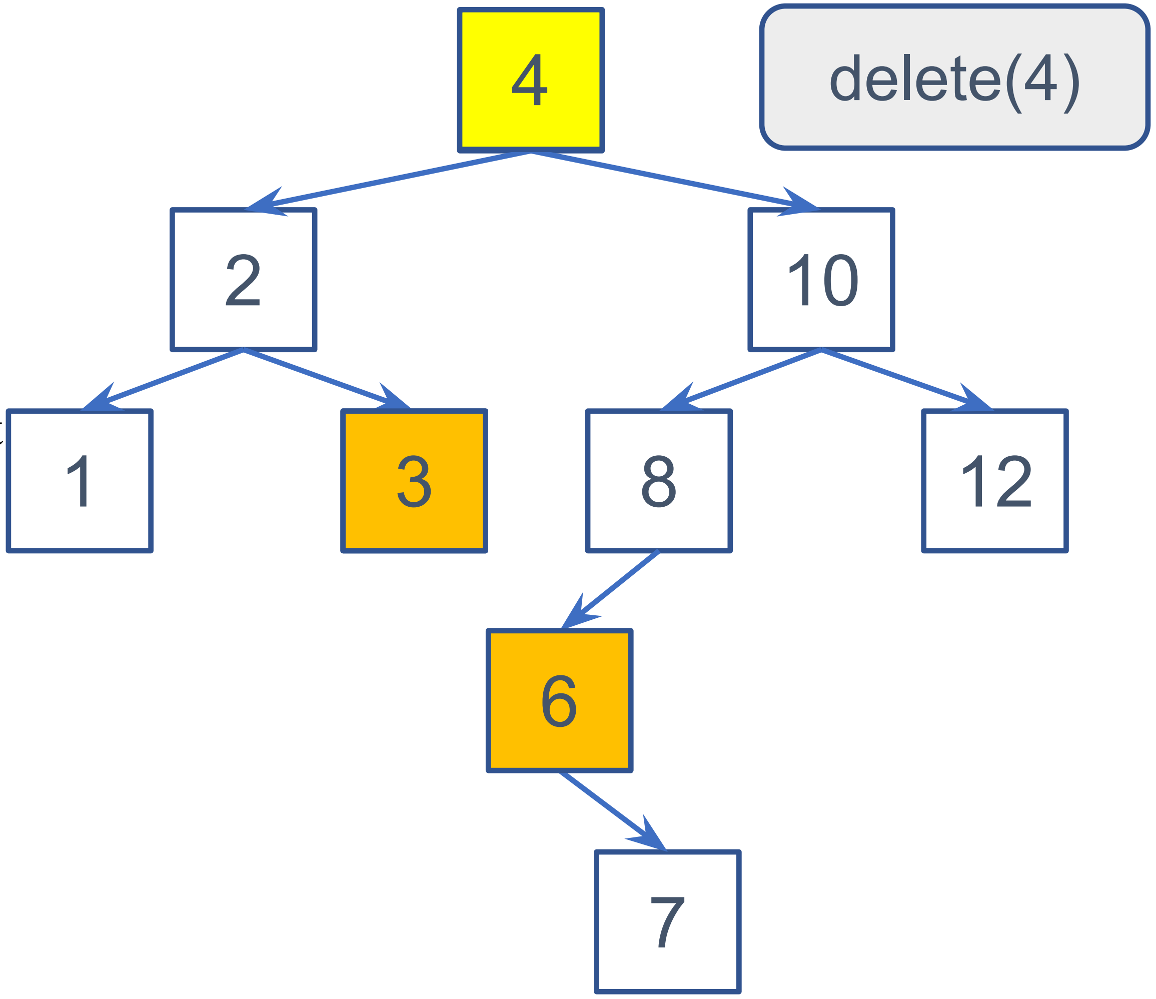
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
    - Find a subtree **node** that can be located at the target node's location
    - The node's value must be **larger than** all the left subtree nodes' values
    - The node's value must be **smaller than** all the right subtree nodes' values



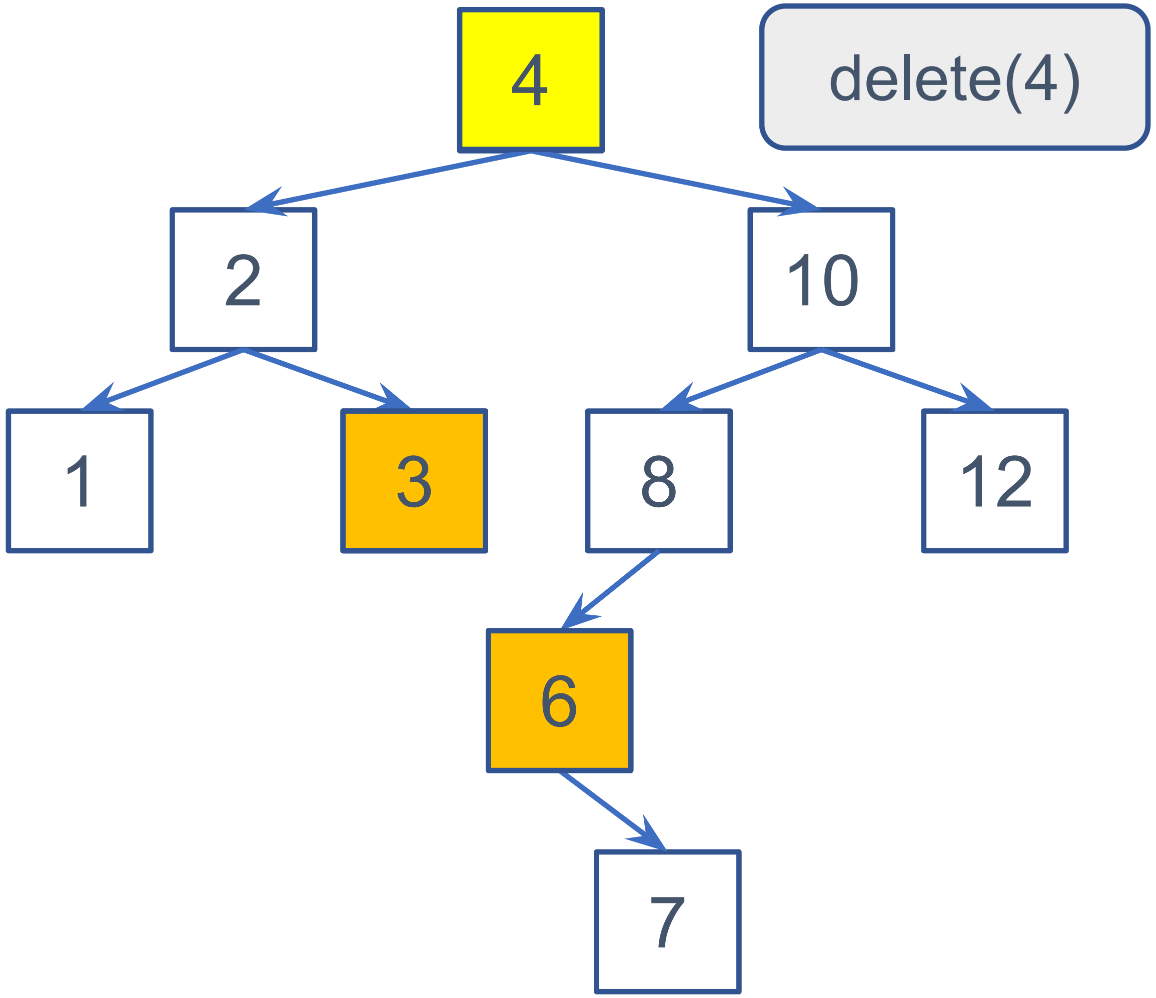
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
    - Find a subtree **node** that can be located at the target node's location
    - The node's value must be **larger than** all the left subtree nodes' values
    - The node's value must be **smaller than** all the right subtree nodes' values
  - Either <u>the rightmost node in the left subtree</u> or <u>the leftmost node in the right subtree</u> works



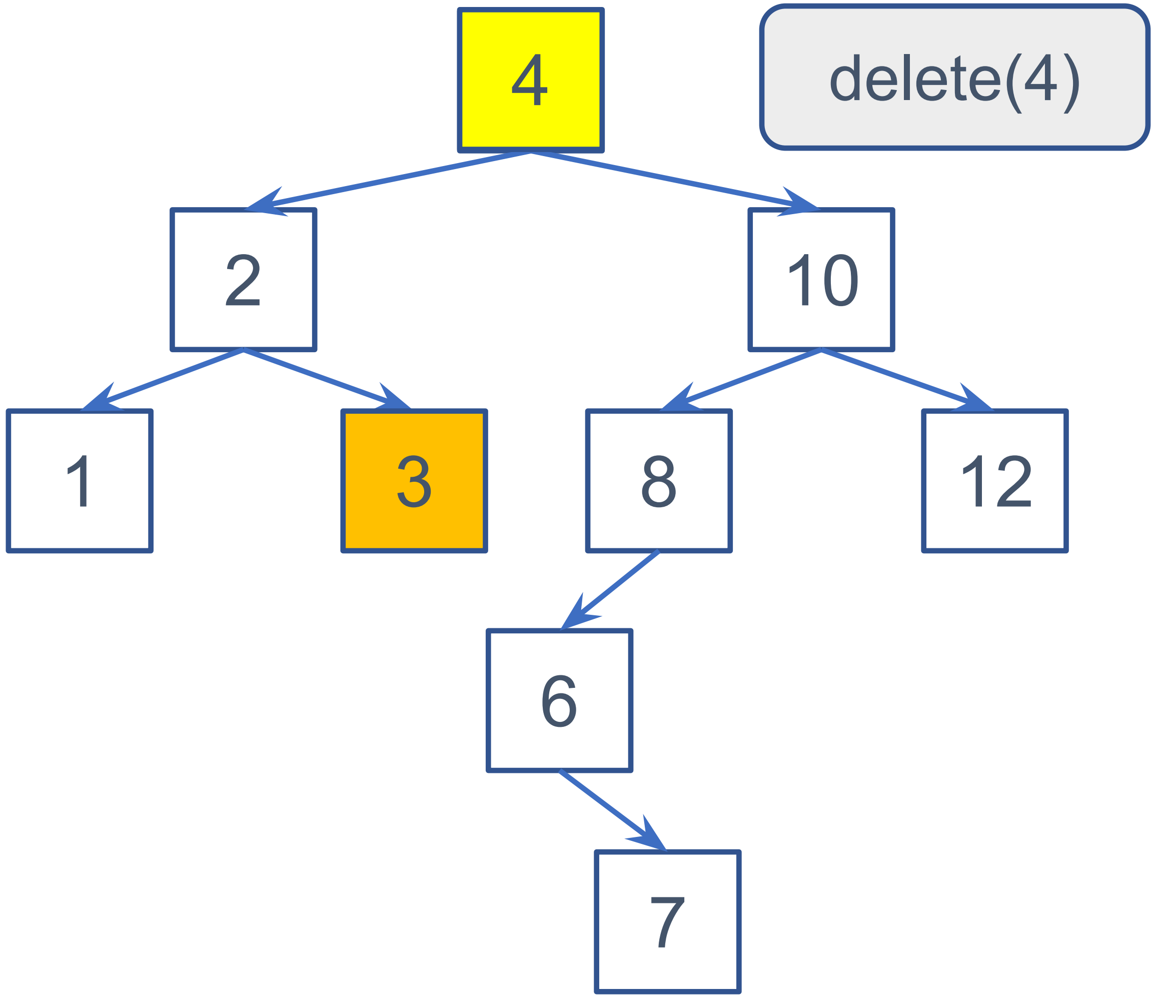
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location



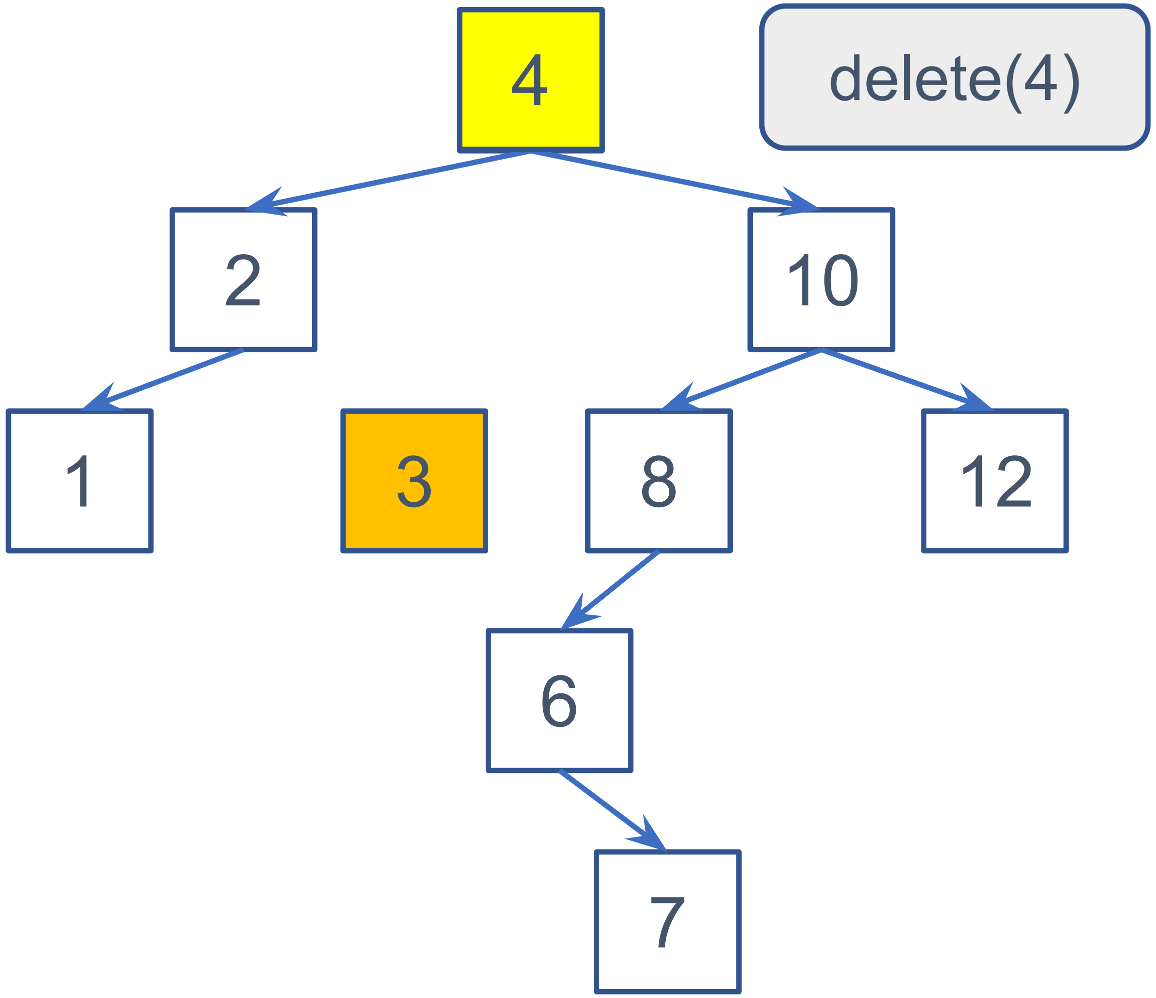
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location
  - Ex.1) Delete 3



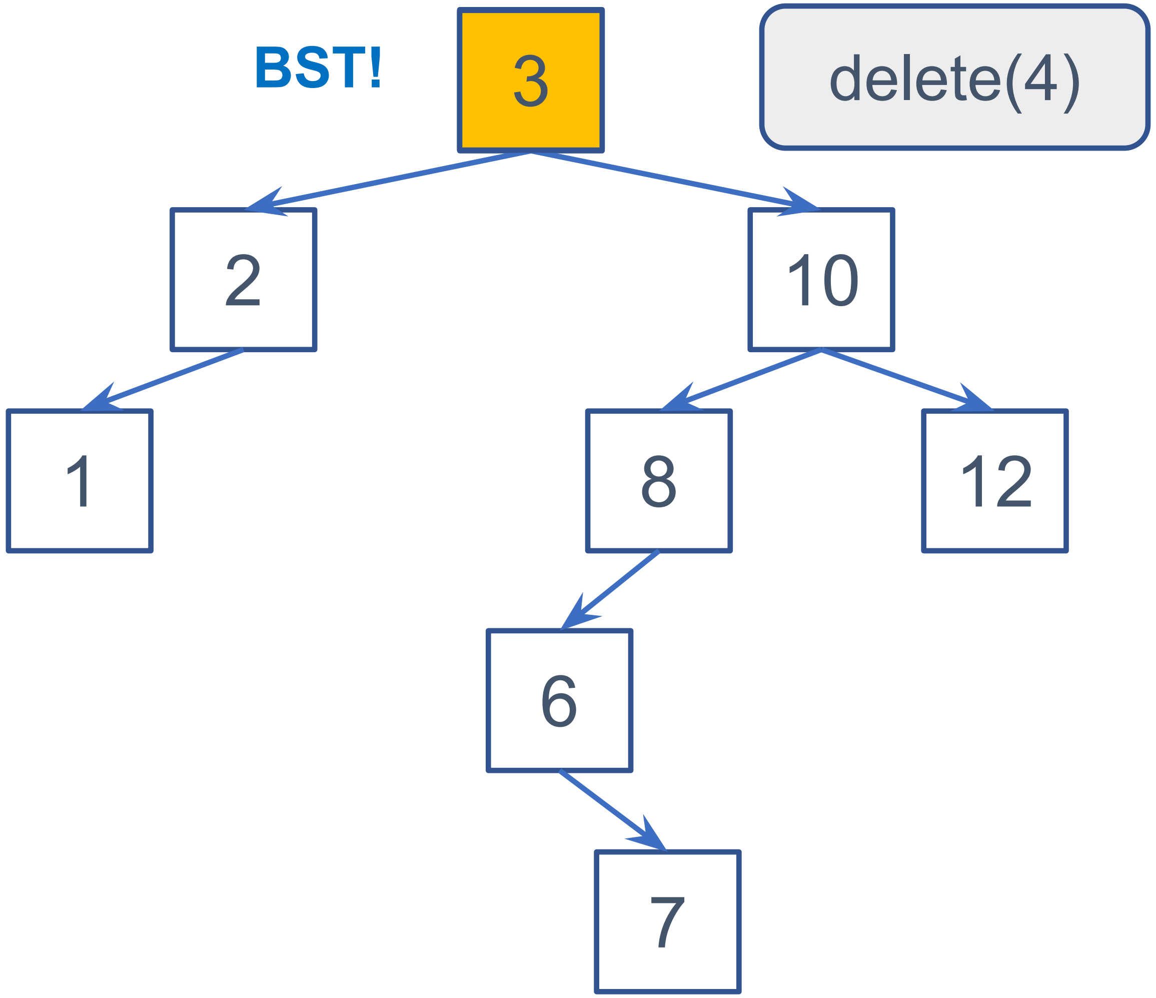
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location
  - Ex.1) Delete 3



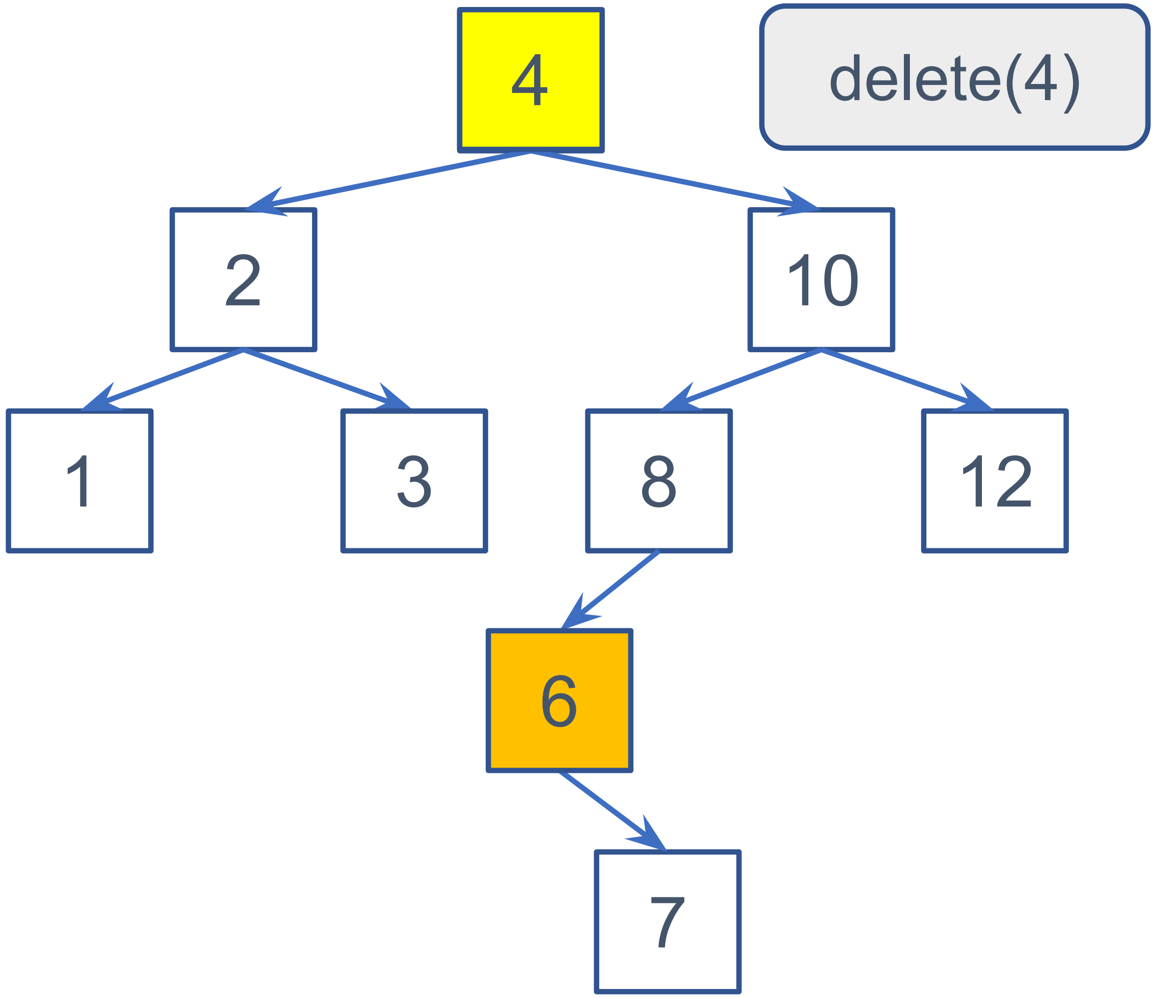
Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location
  - Ex.1) Delete 3



Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location
  - Ex.1) Delete 3
  - Ex.2) Delete 6

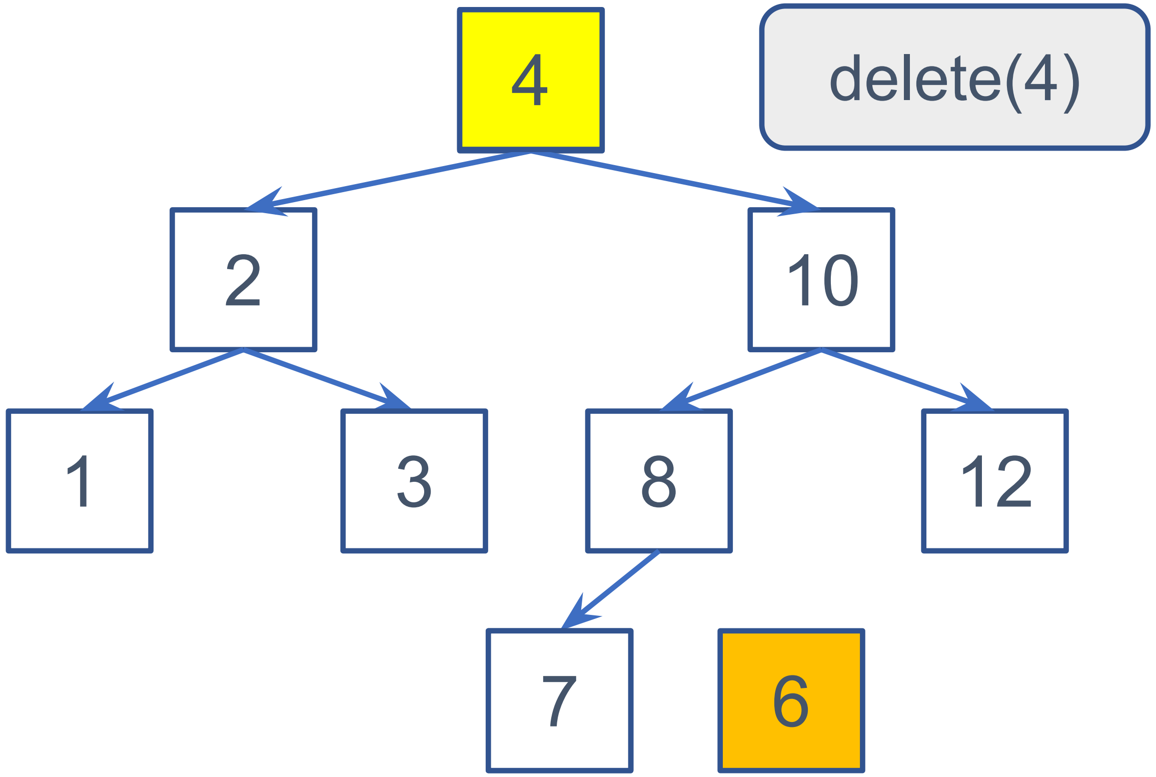


Binary Search Trees – Delete

- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value

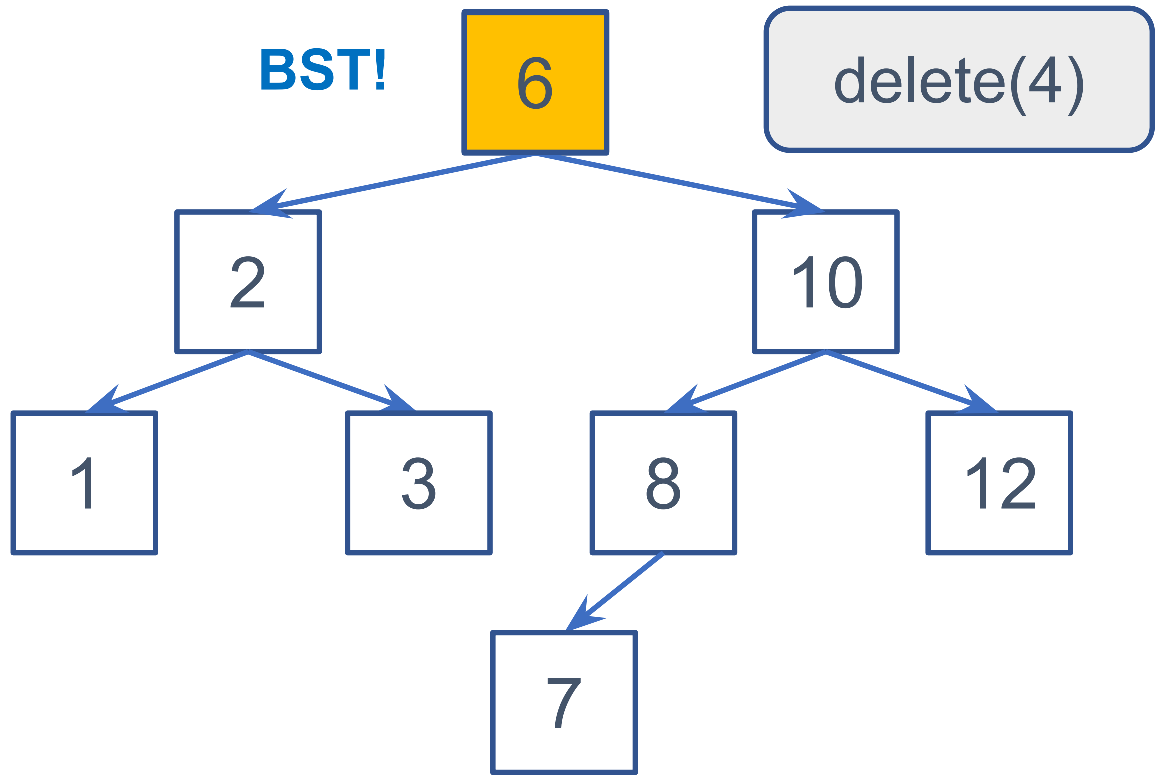
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location

  - Ex.1) Delete 3
  - Ex.2) Delete 6



Binary Search Trees – Delete

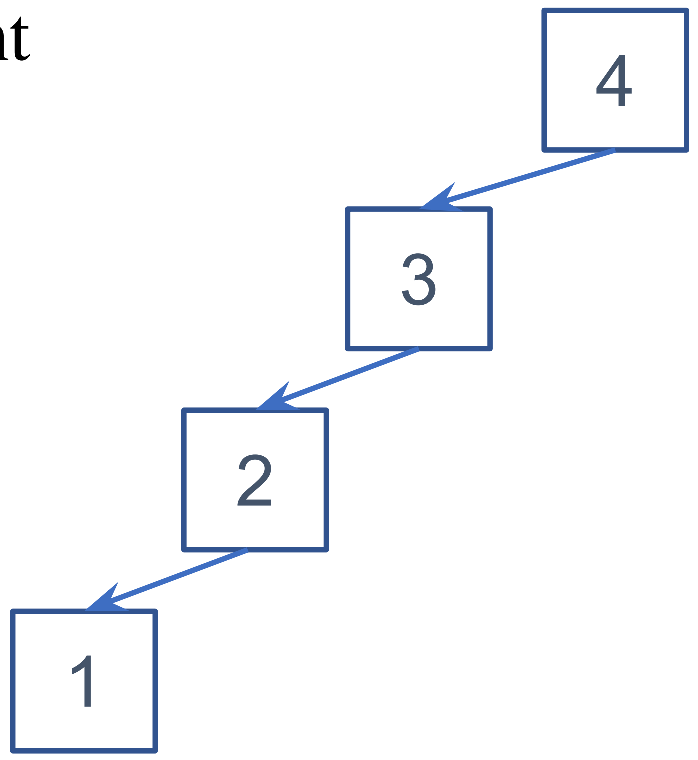
- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location
  - Ex.1) Delete 3
  - Ex.2) Delete 6



Binary Search Trees – Performance

- BST operations require **O(log N)**, which is its **depth**
  - Only if the BST is balanced

- Maintaining a BST to be **balanced** is very important to maximize its performance!
  - Which is out of scope of this course ☺



Computing Foundations of Data Science

Trees

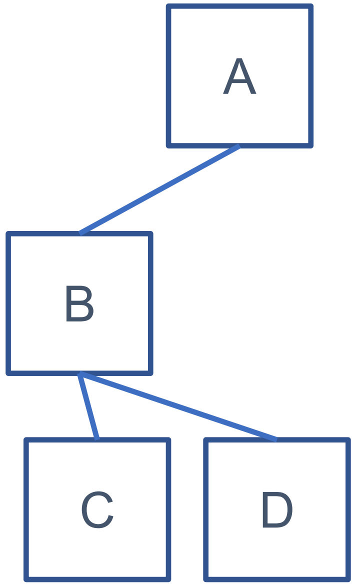
Lecture 11

Hyung-Sin Kim

SNU Graduate School of Data Science

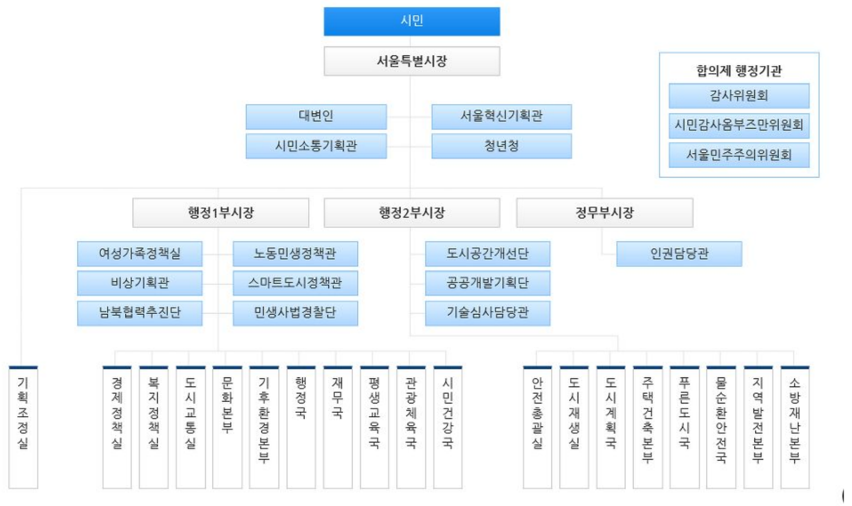
Review

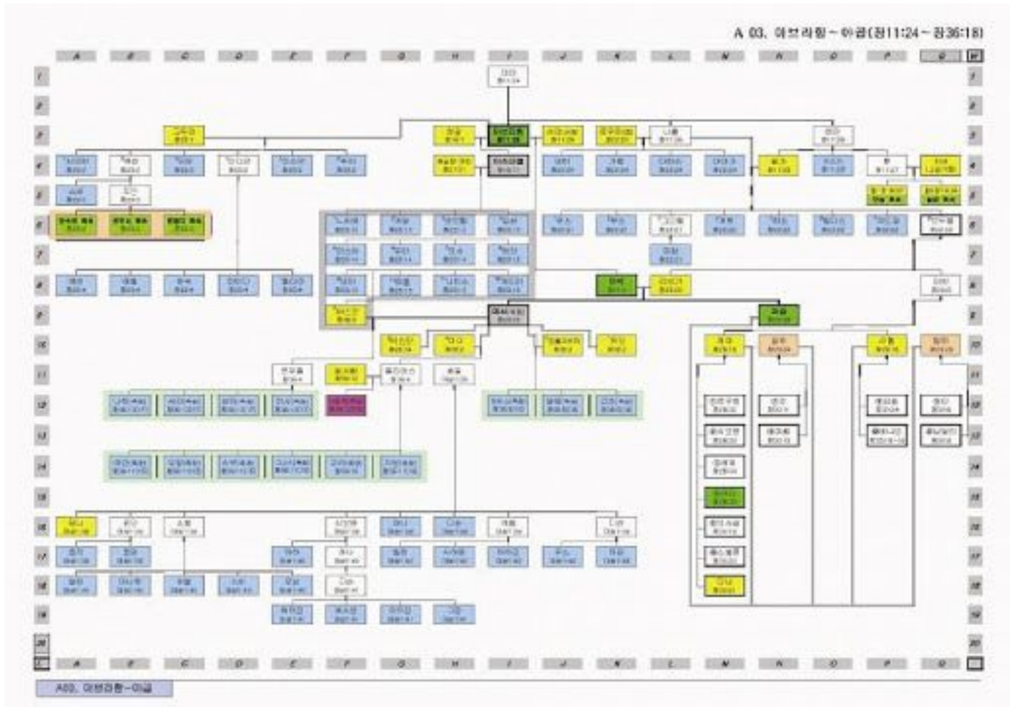
- Tree
- Rooted tree
- Rooted binary tree
- Binary search tree

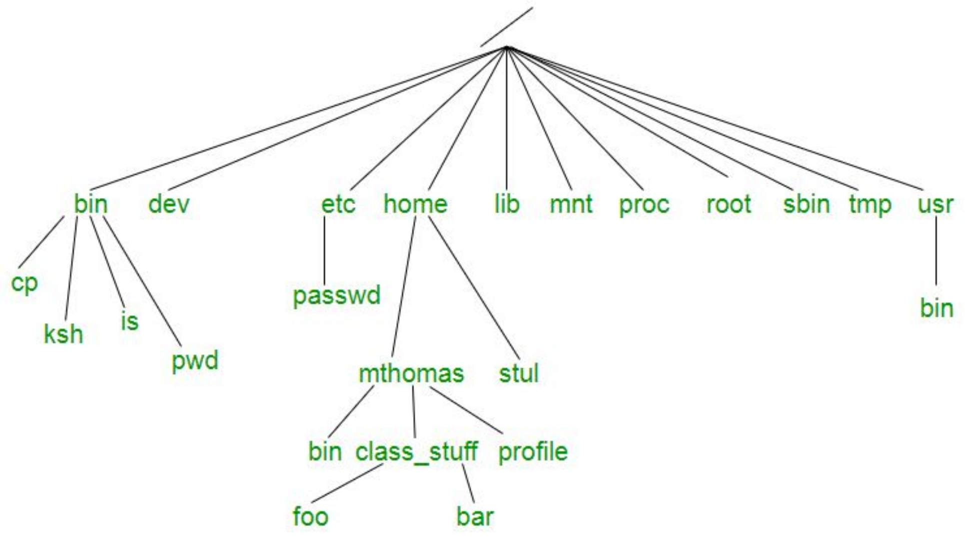


Trees are Everywhere

- Organization chart
- Genealogy (family tree)
- File system







K-ary Trees

- A general tree node does not have to have only two children nodes

- A tree that allows each node to have up to k children nodes is called **k-ary tree**

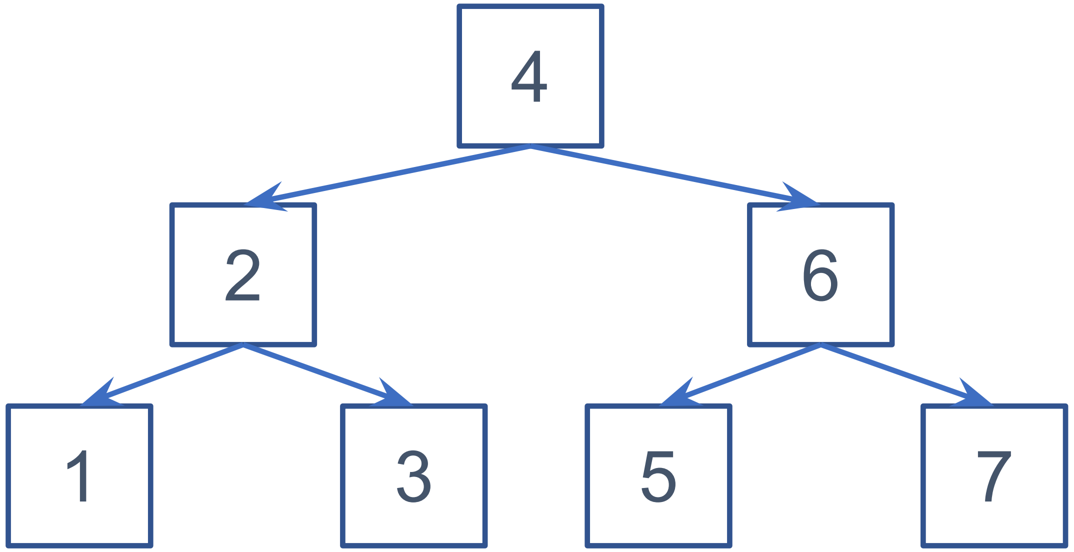
```
class TreeNode():
    def __init__(self, x: int, k: int) -> None:
        self.val = x
        self.arity = k
        self.child = [None]*k
```

How to navigate the whole tree conveniently?

Breadth-First Traversal

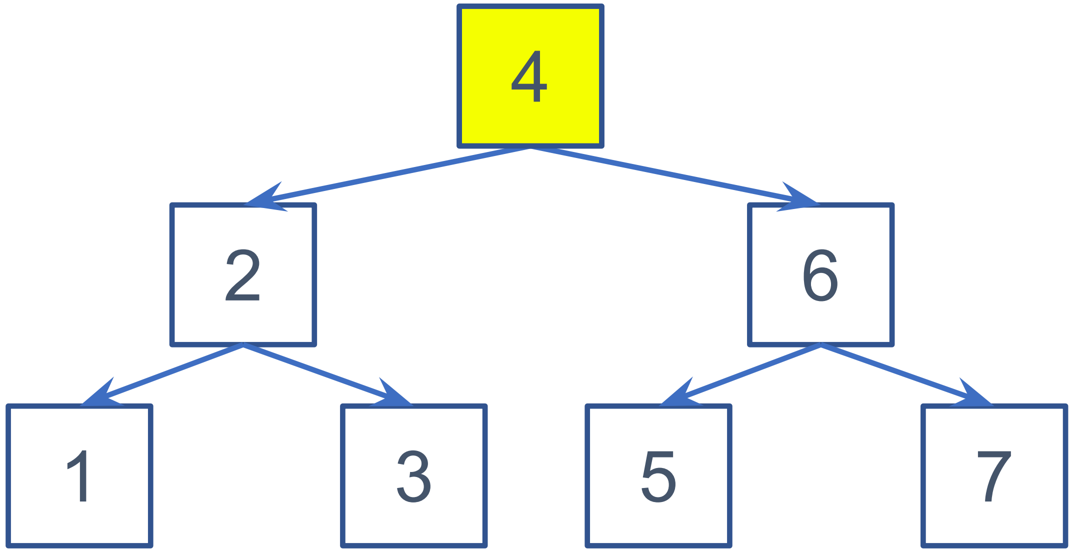
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



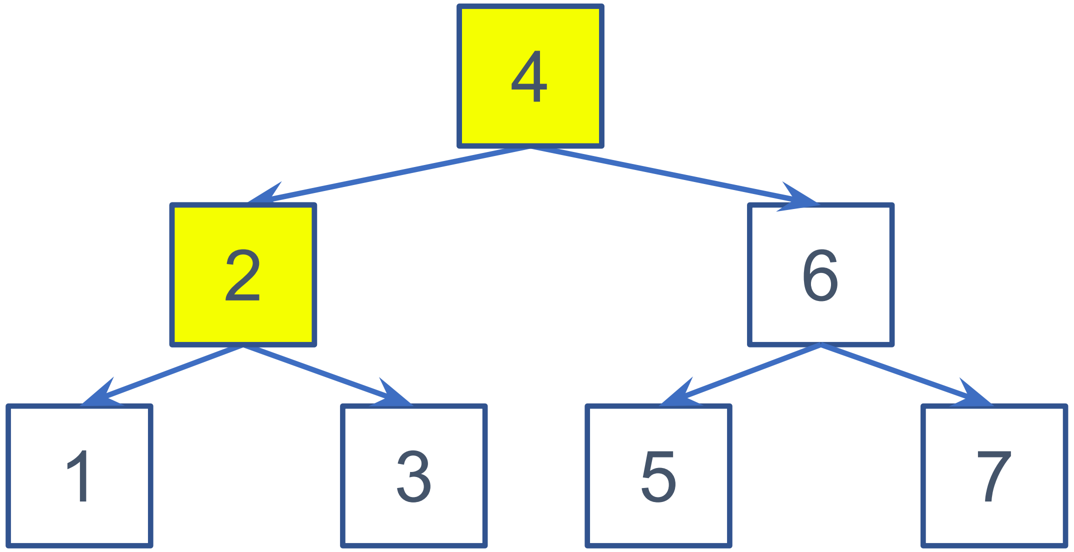
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



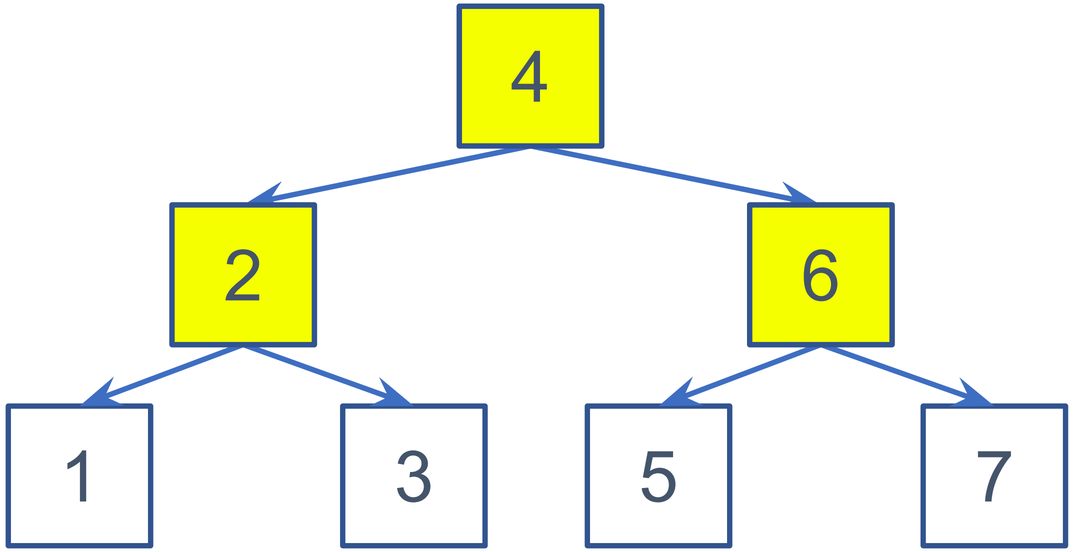
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



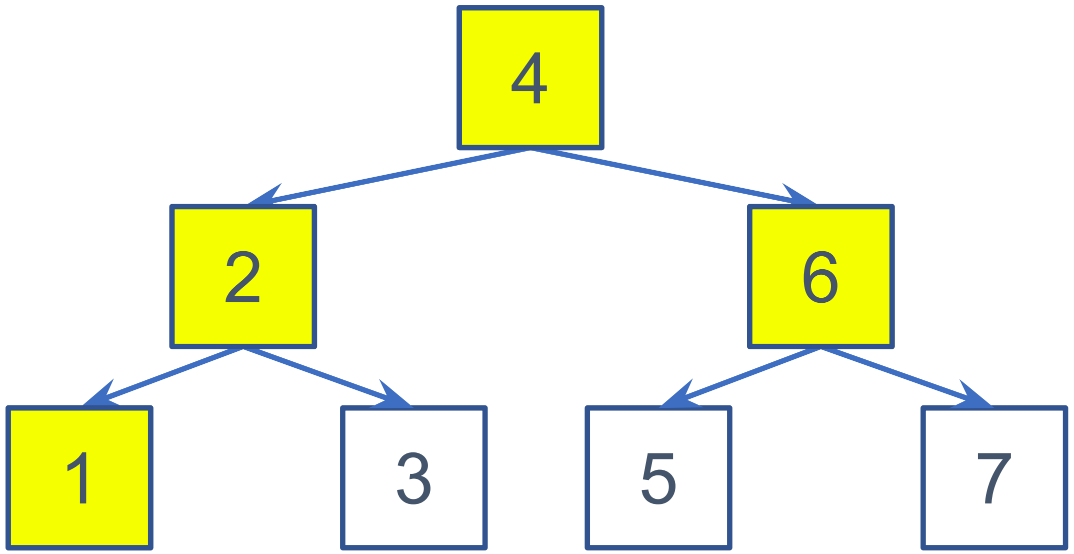
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



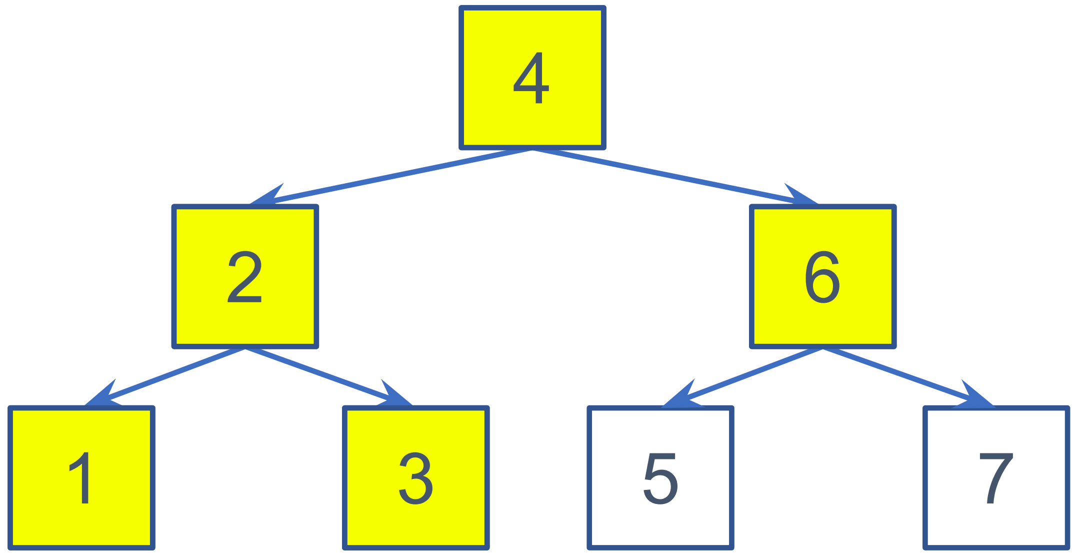
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



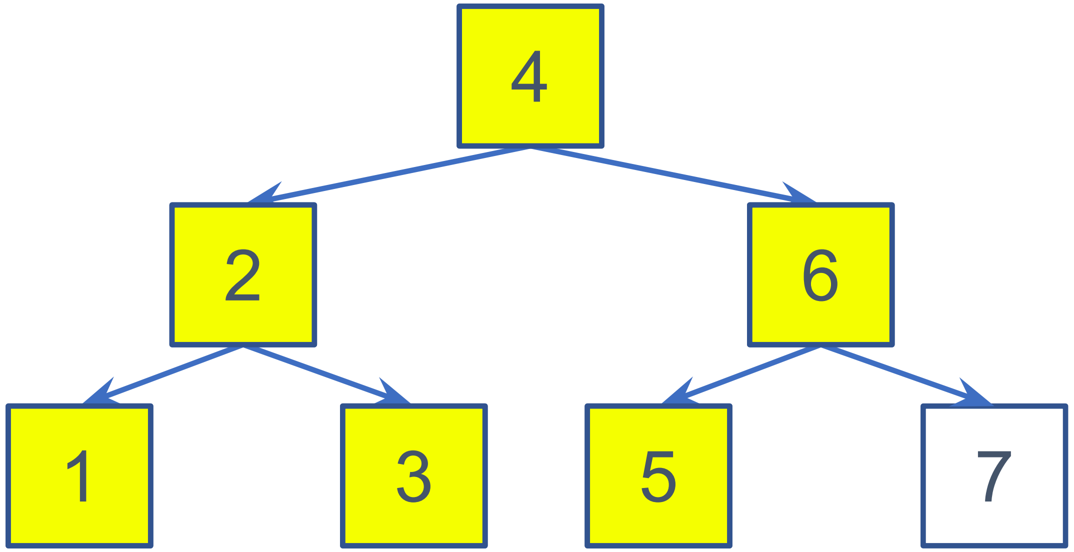
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



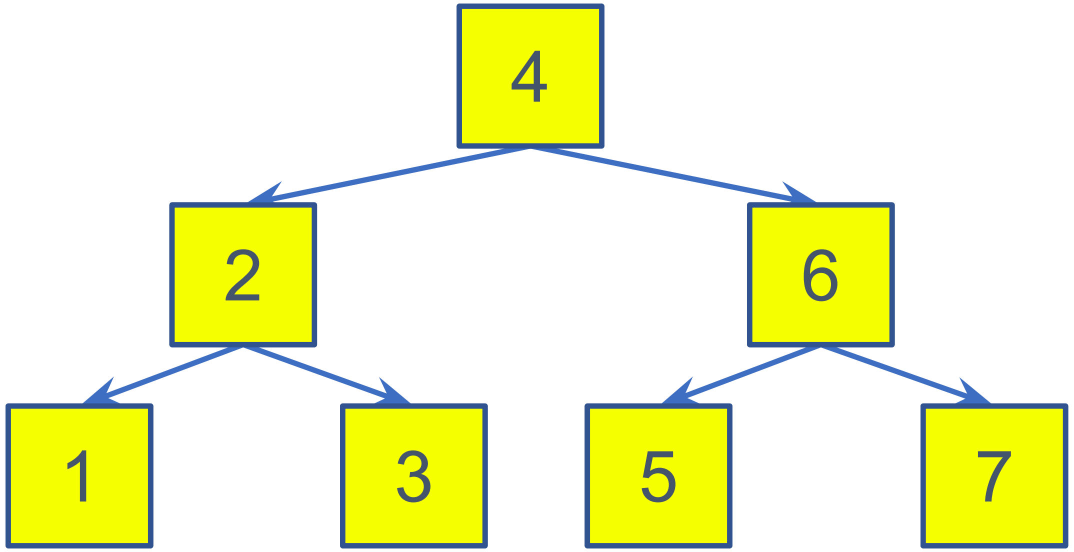
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom



Level-order (Breadth-First) Traversal

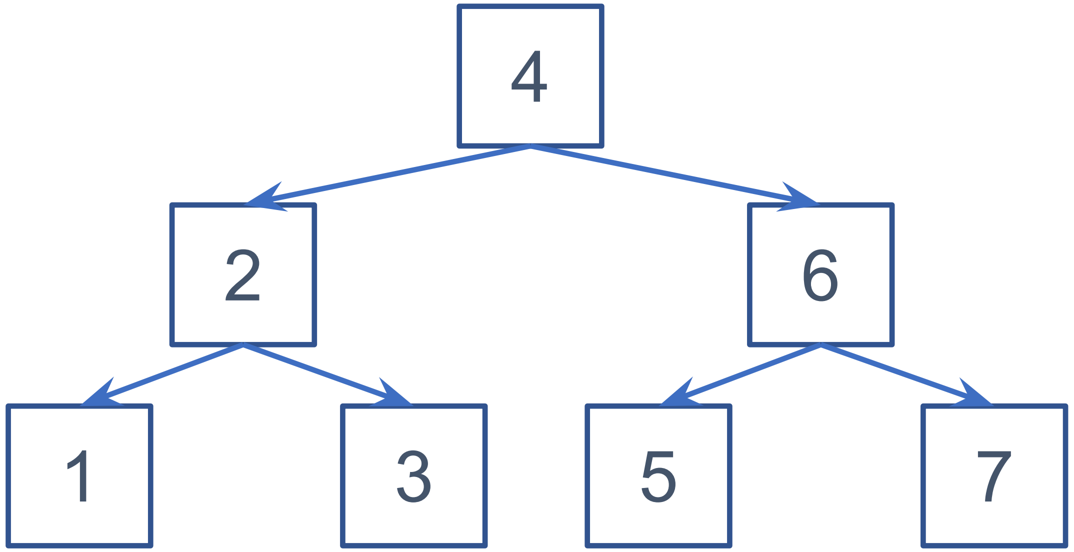
- Visit nodes from left to right, and from top to bottom



Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```



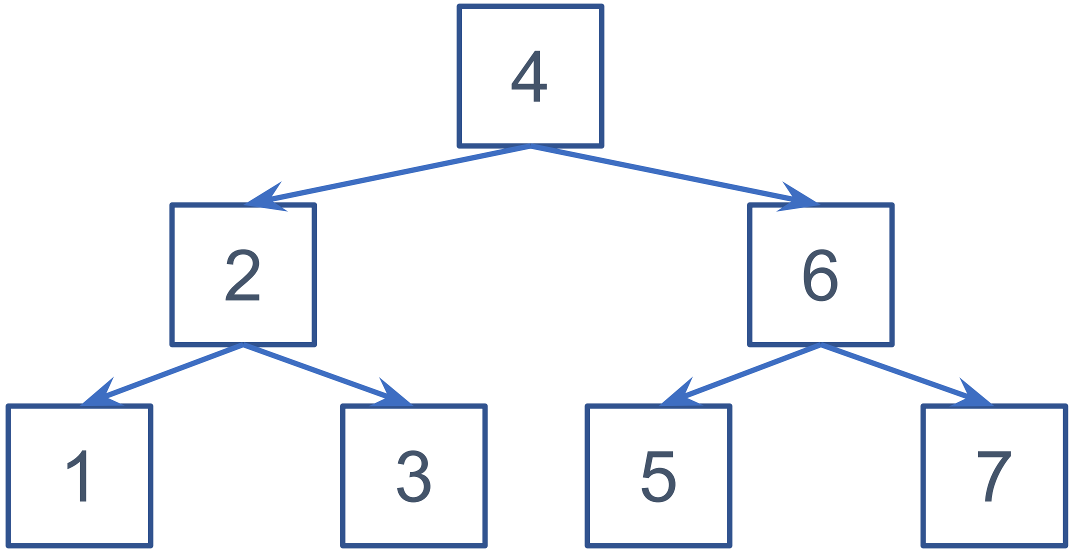
Out:

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(4)]

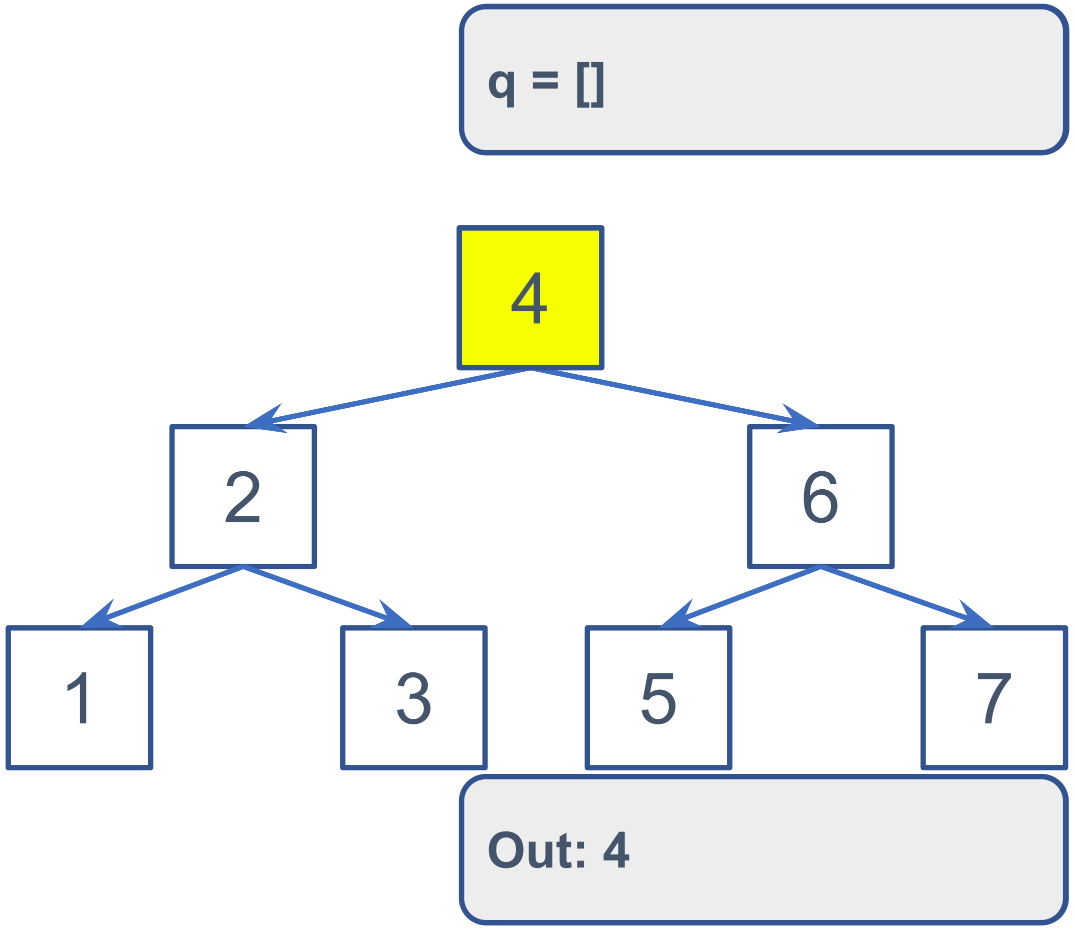


Out:

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

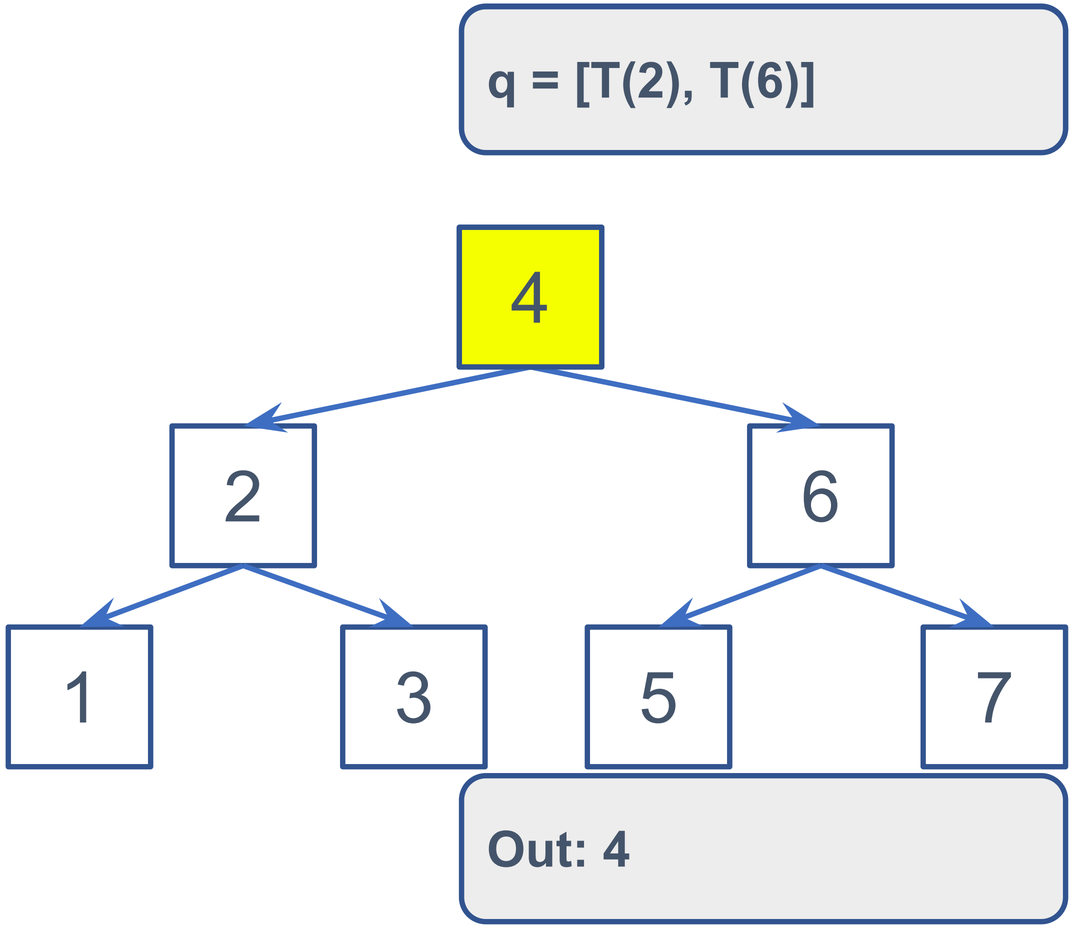
```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```



Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

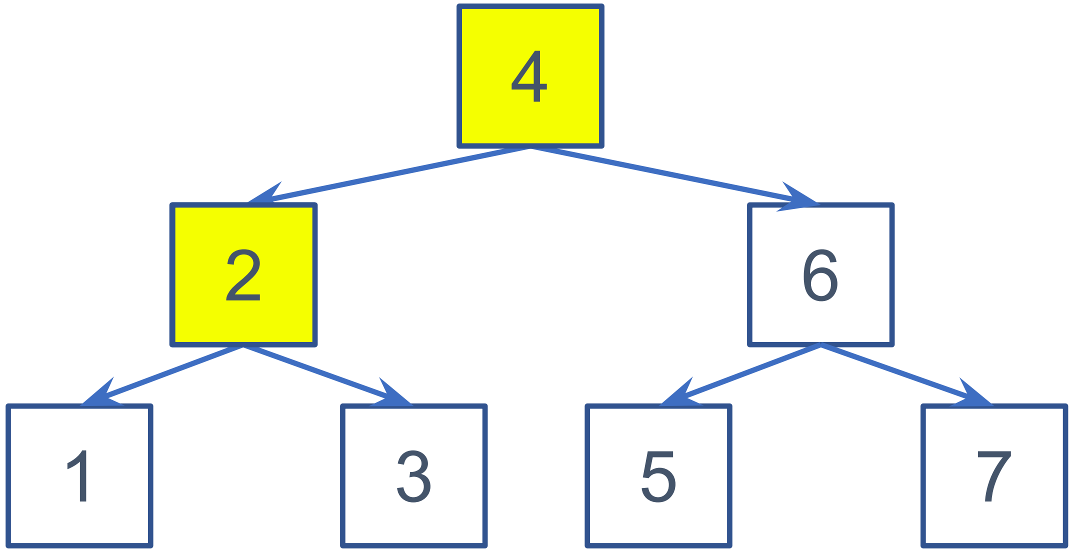


Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(6)]



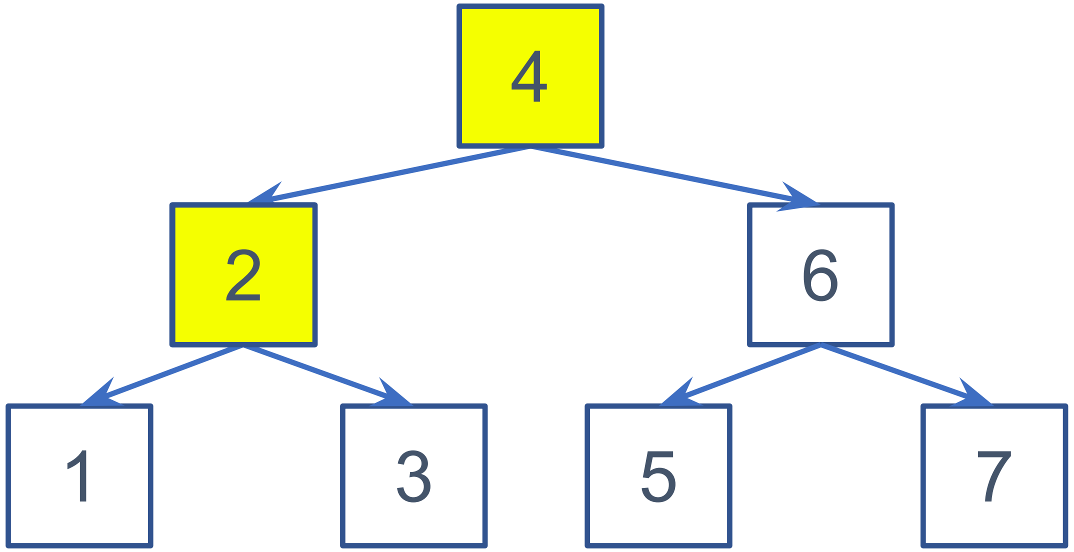
Out: 4 2

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(6), T(1), T(3)]



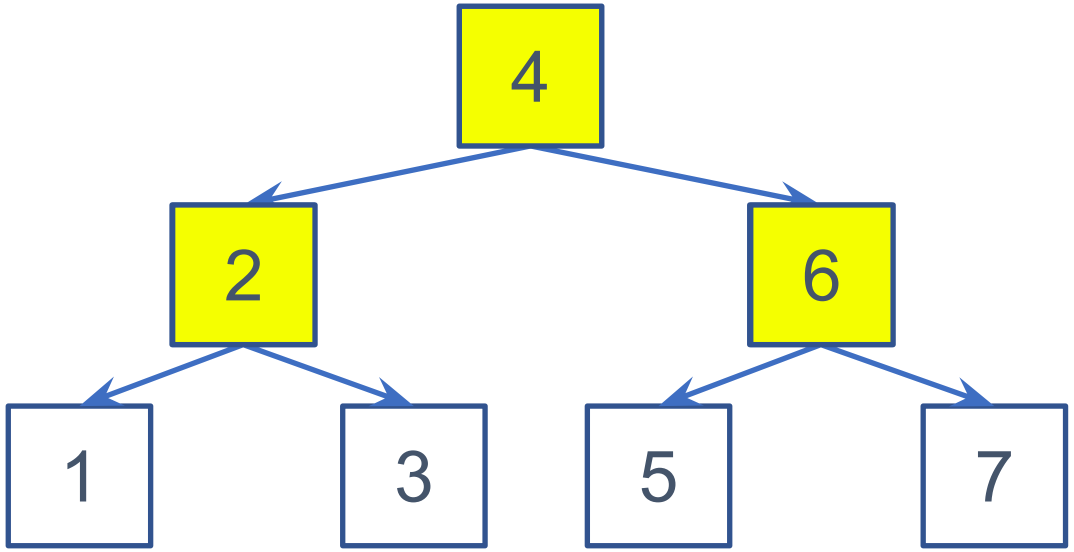
Out: 4 2

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(1), T(3)]



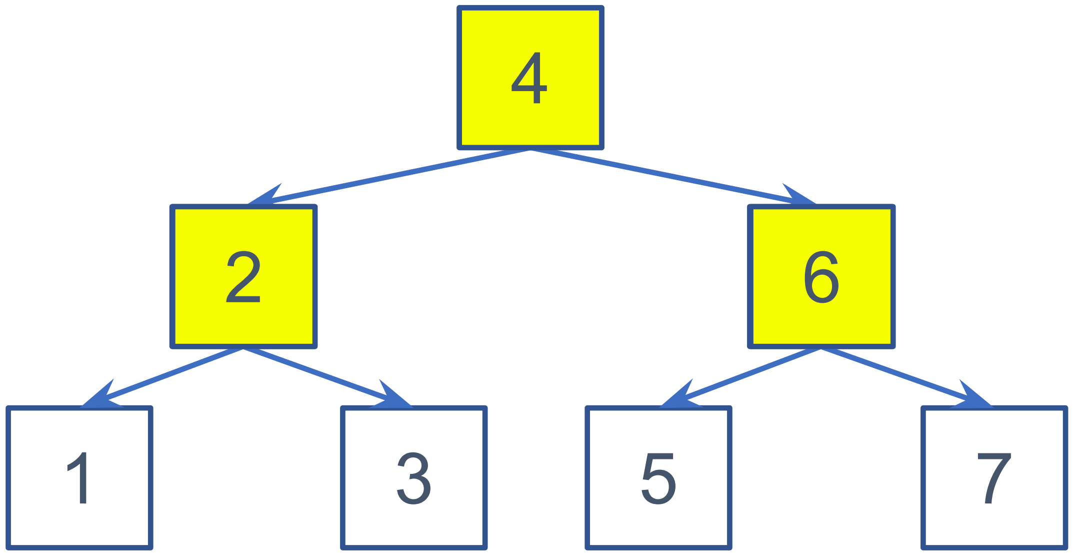
Out: 4 2 6

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(1), T(3), T(5), T(7)]



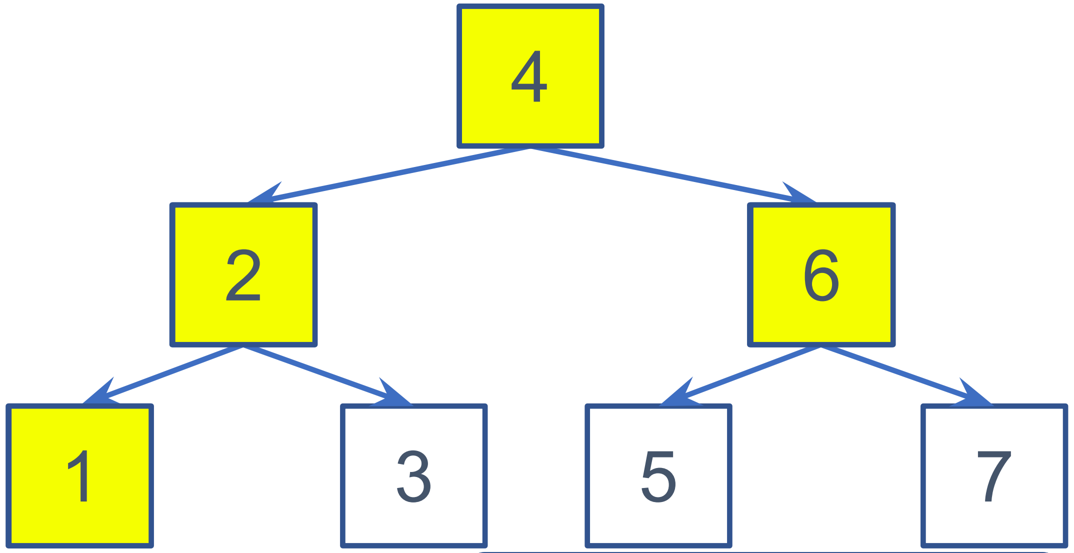
Out: 4 2 6

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(3), T(5), T(7)]



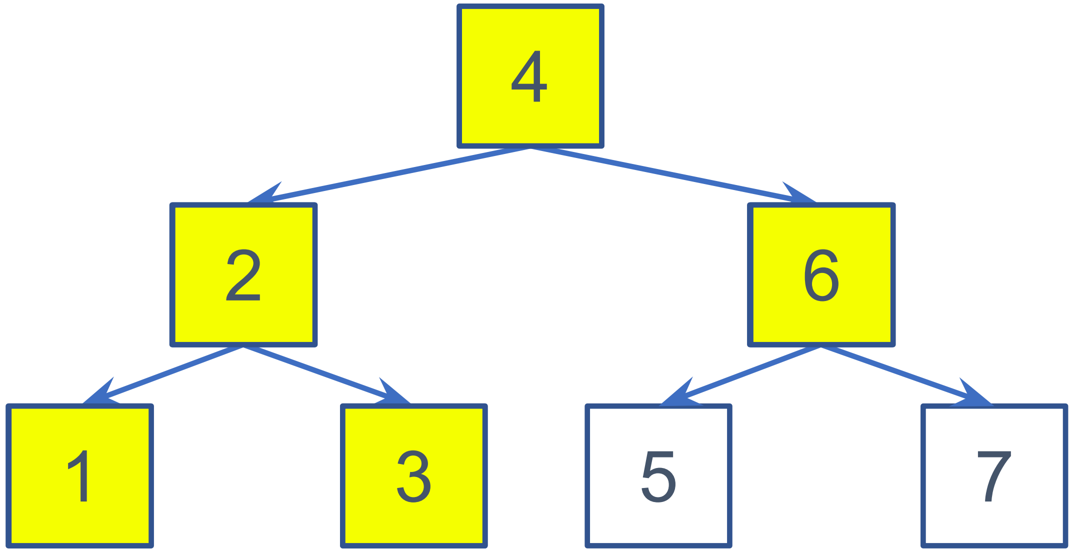
Out: 4 2 6 1

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(5), T(7)]



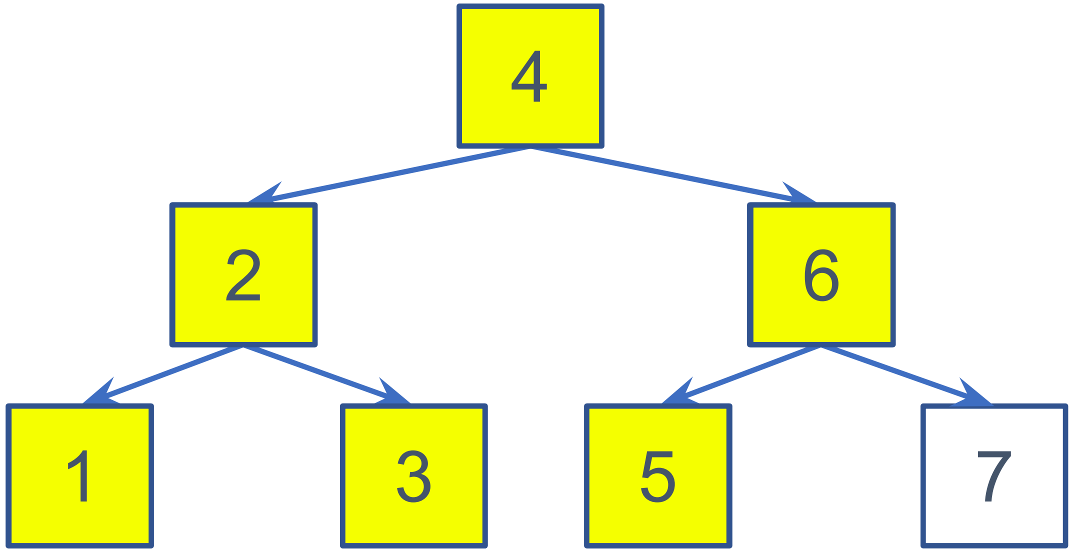
Out: 4 2 6 1 3

Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = [T(7)]



Out: 4 2 6 1 3 5

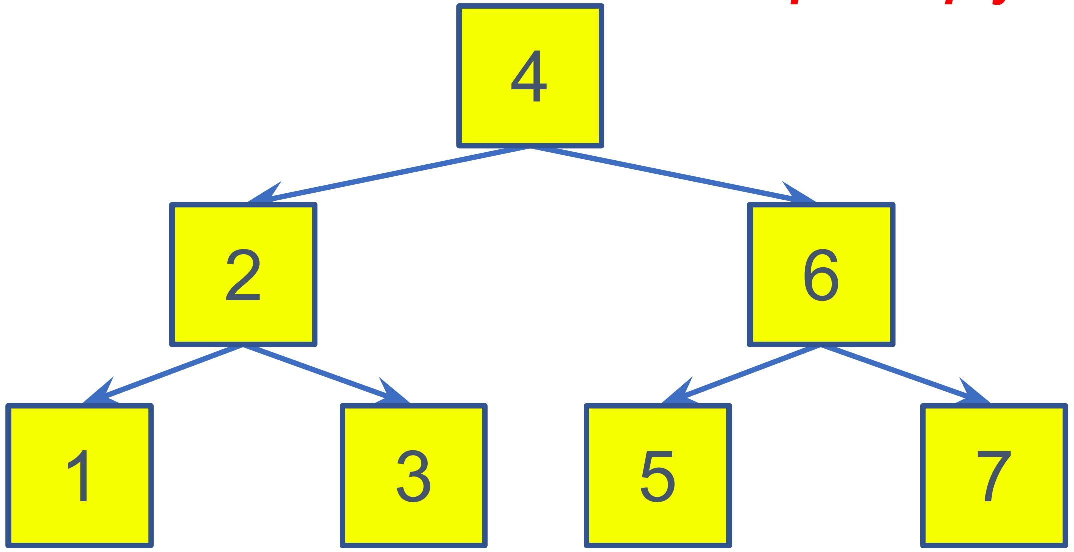
Level-order (Breadth-First) Traversal

- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = [self.root]
        while q:
            curNode = q.pop(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

q = []

Now q is empty!



Out: 4 2 6 1 3 5 7

Level-order (Breadth-First) Traversal – Deque

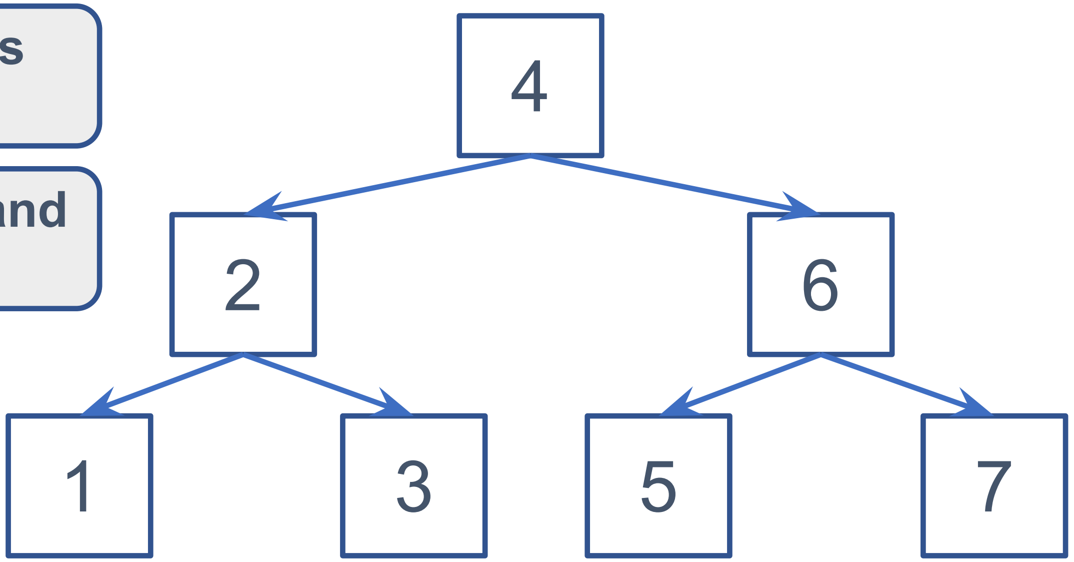
- Visit nodes from left to right, and from top to bottom

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def BFT(self):
        if self.root == None:
            return
        q = deque([self.root])
        while q:
            curNode = q.popleft(0)
            self.visit(curNode)
            for childNode in curNode.child:
                if childNode:
                    q.append(childNode)
```

from **collections** import **deque**

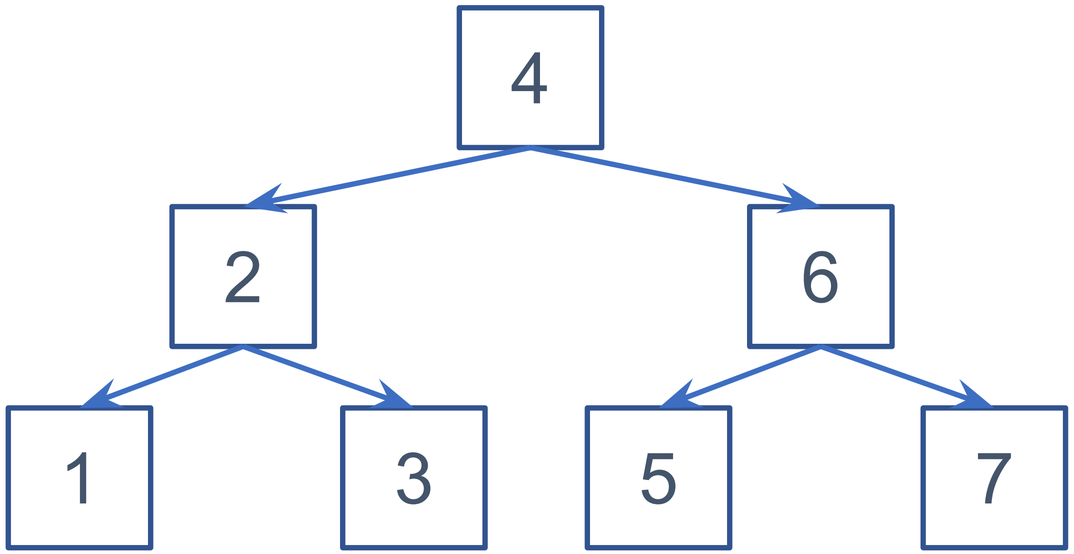
Faster pushing and popping!

Doubly-linked list that provides
- append(x), appendleft(x),
- pop(), popleft()

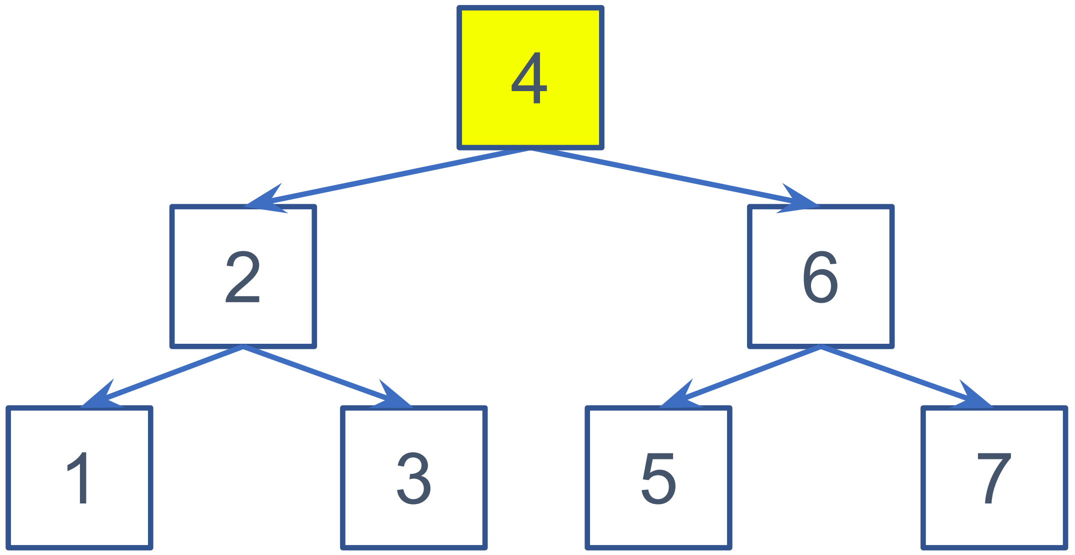


Depth-First Traversal

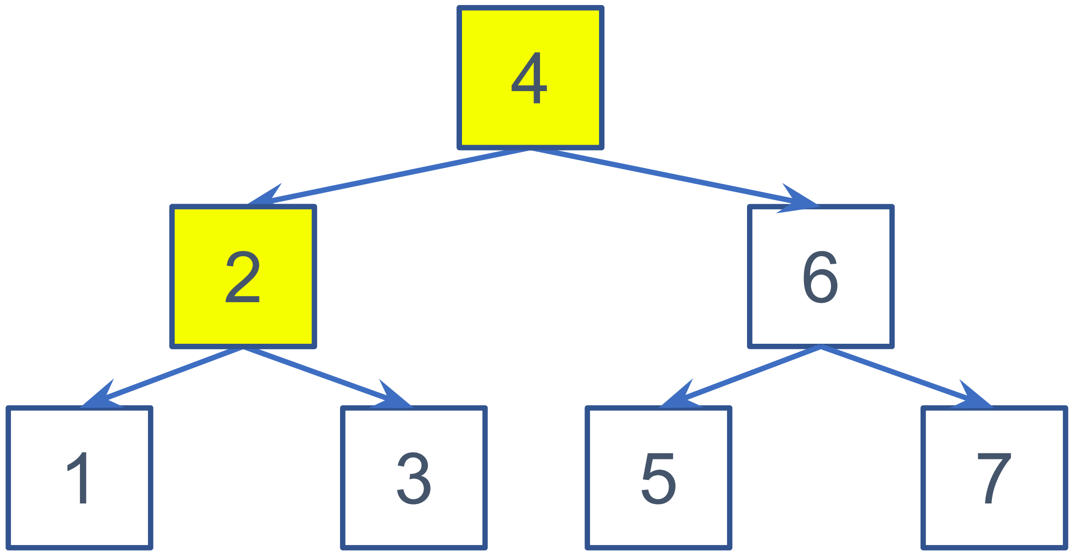
Depth First Traversals



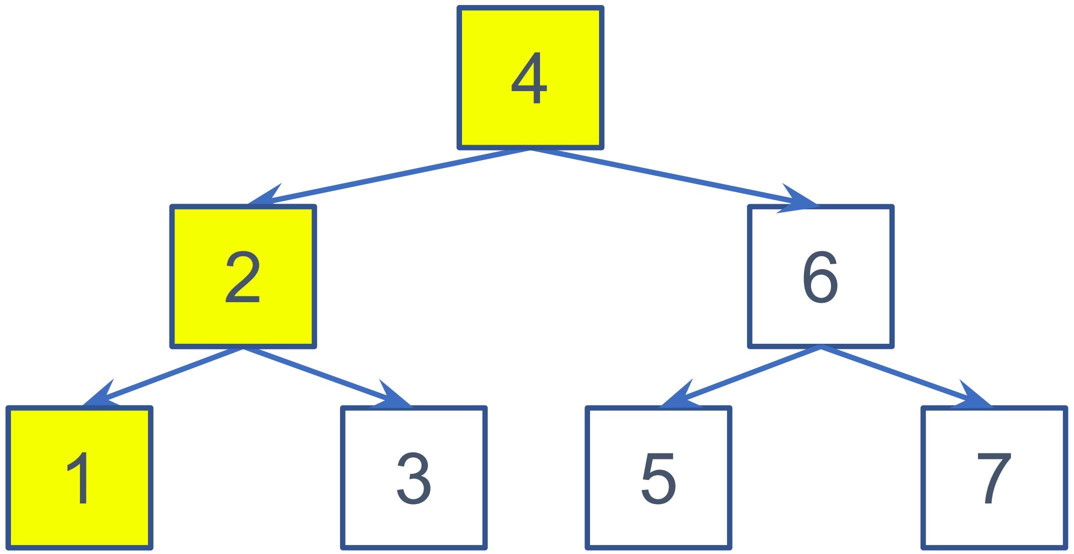
Depth First Traversals



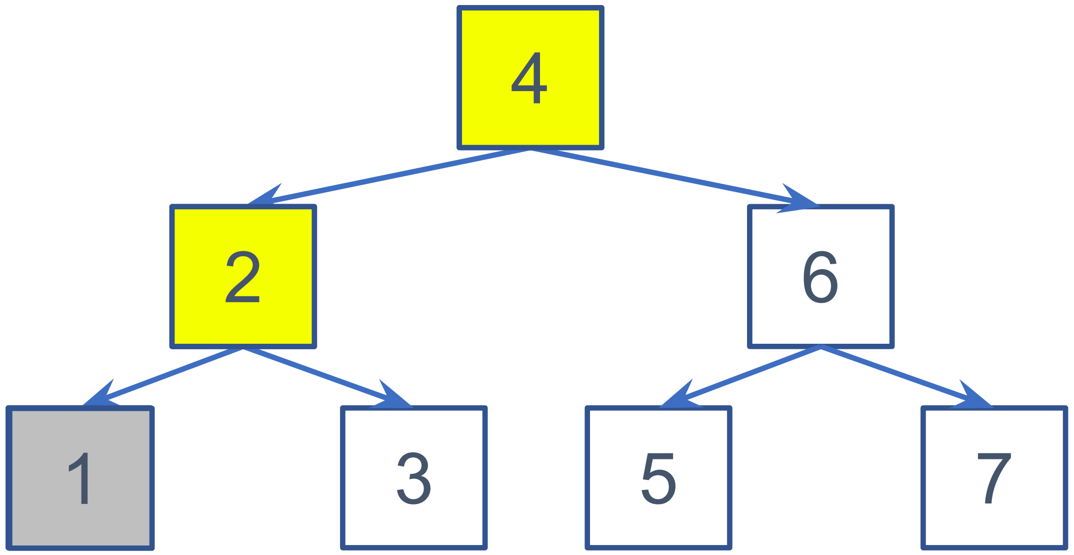
Depth First Traversals



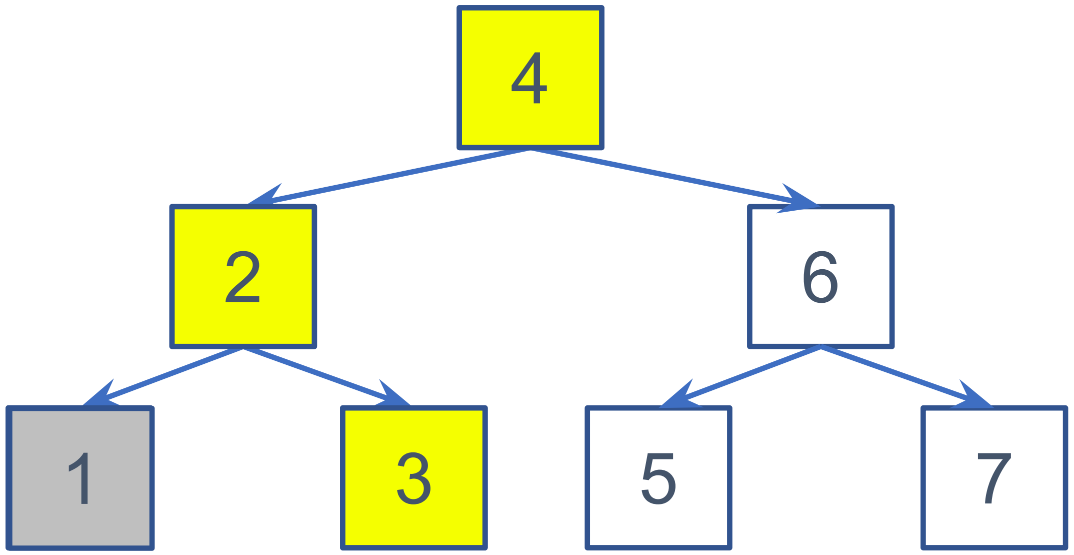
Depth First Traversals



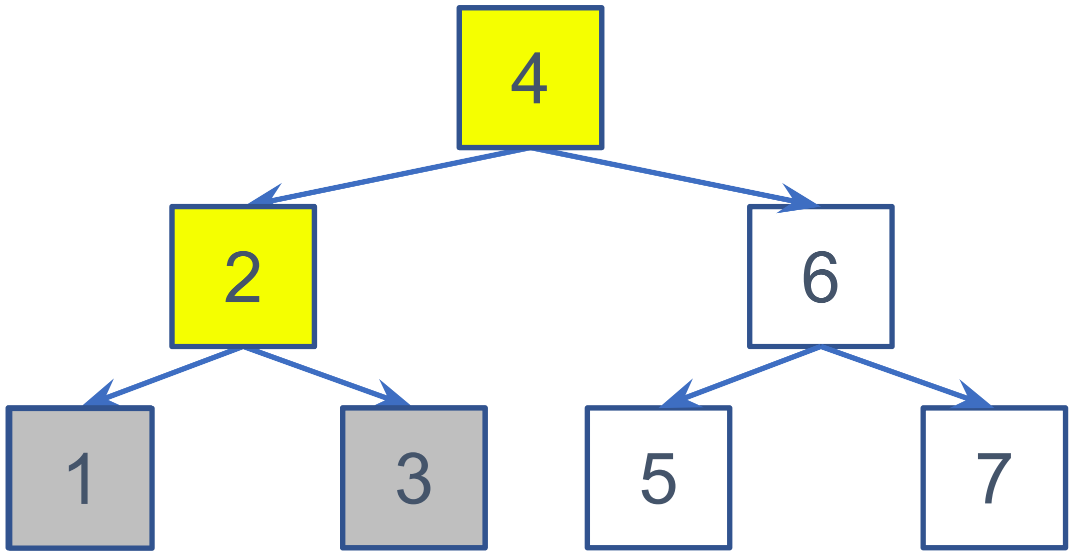
Depth First Traversals



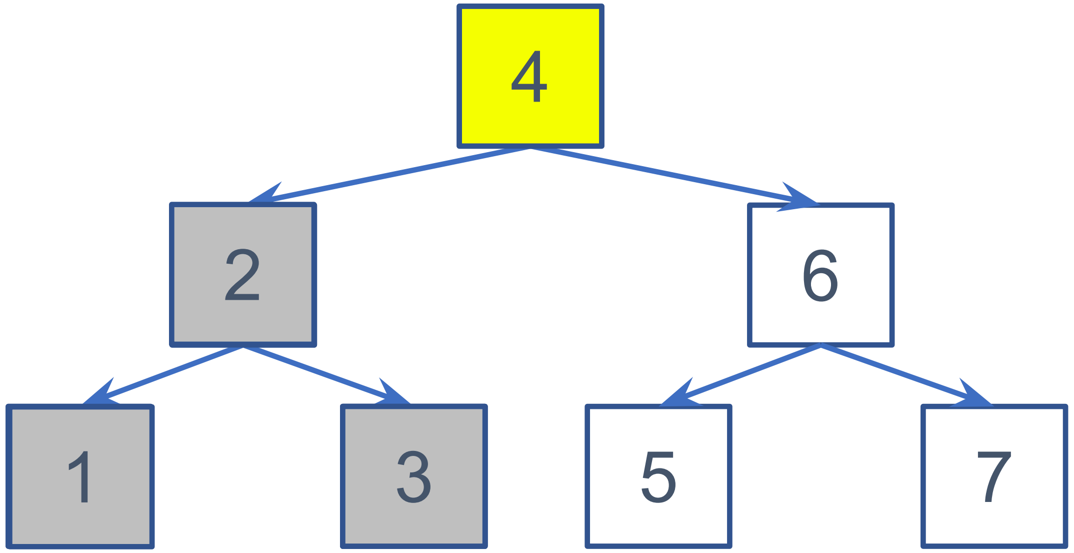
Depth First Traversals



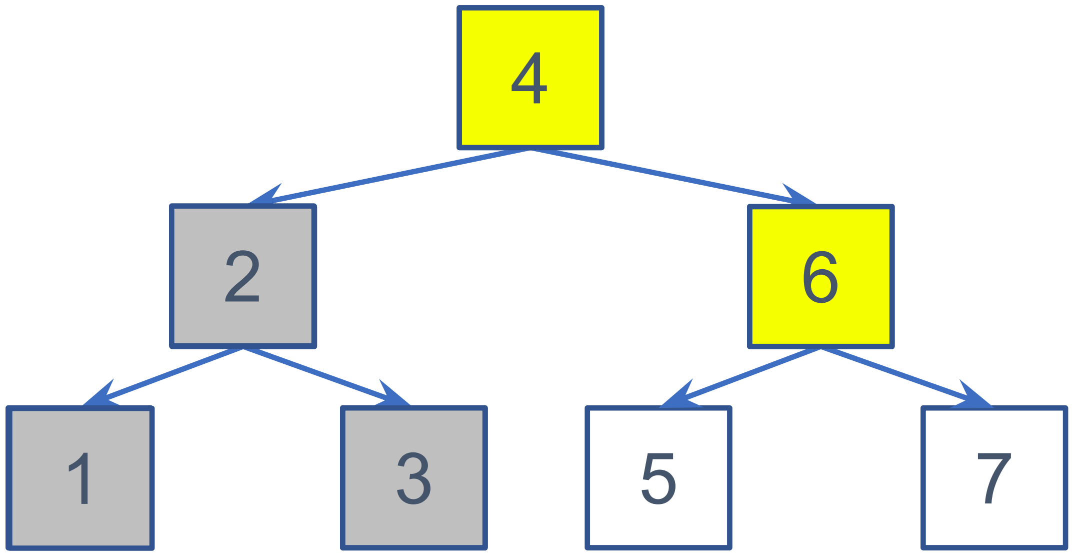
Depth First Traversals



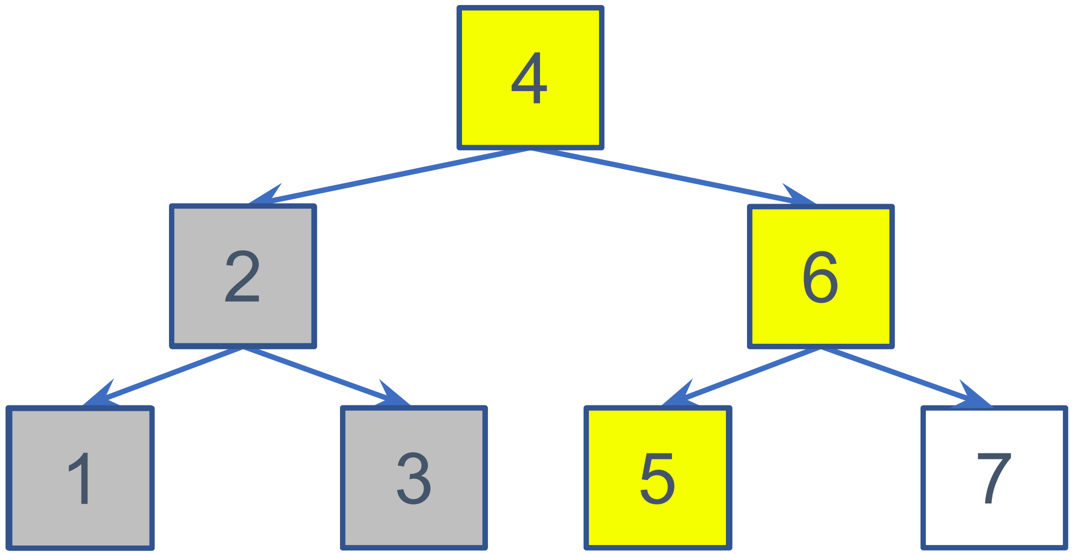
Depth First Traversals



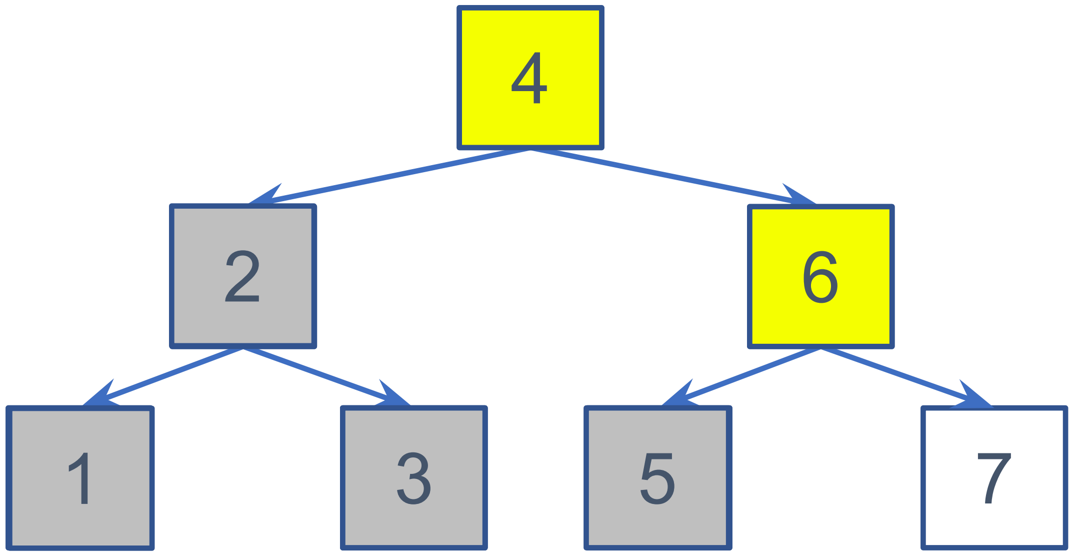
Depth First Traversals



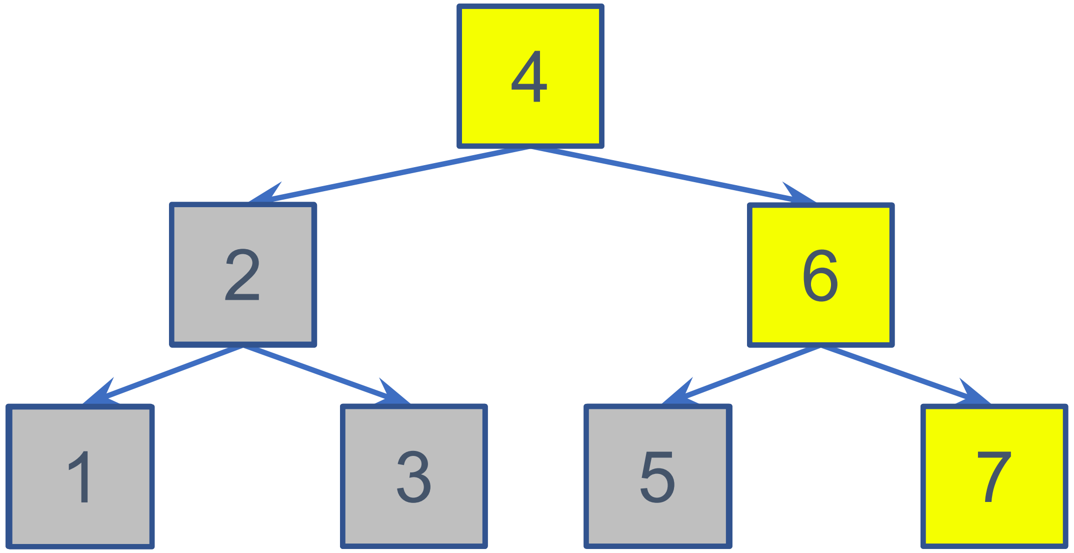
Depth First Traversals



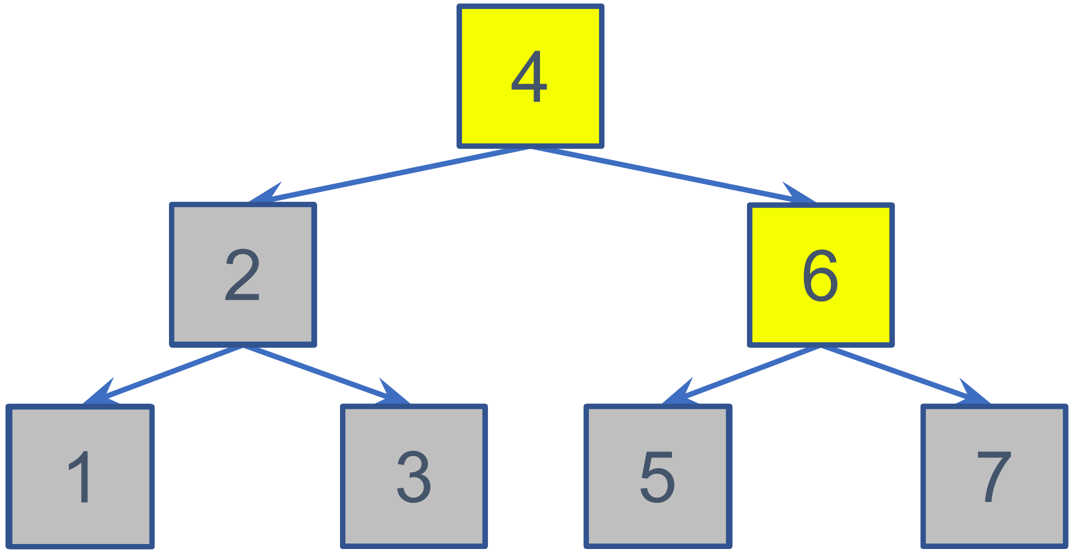
Depth First Traversals



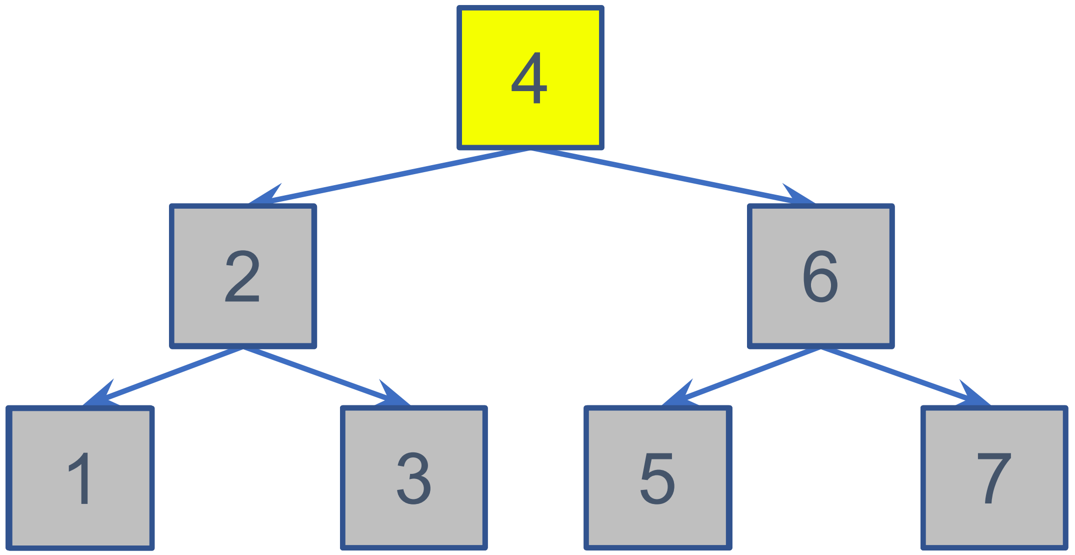
Depth First Traversals



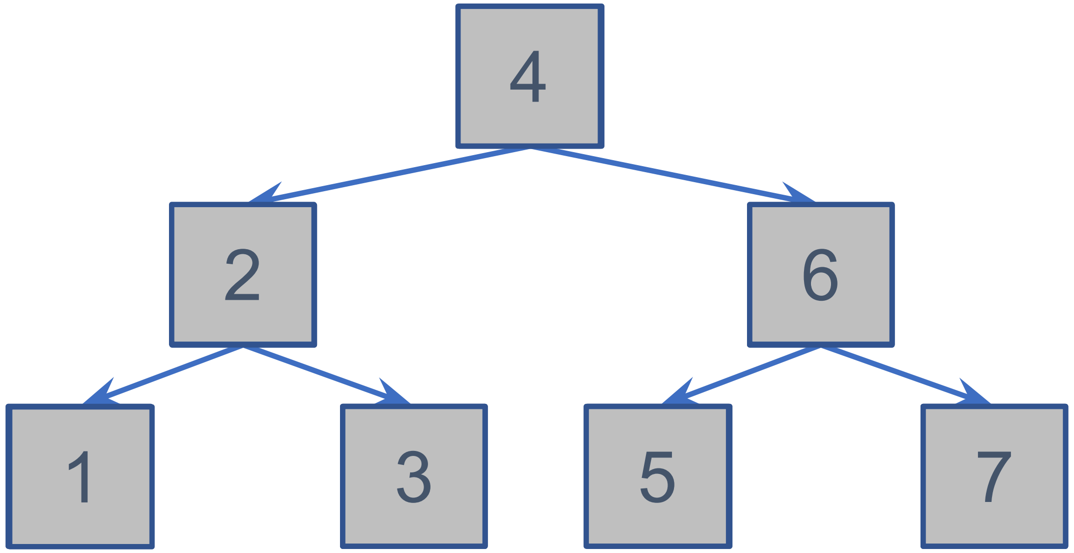
Depth First Traversals



Depth First Traversals

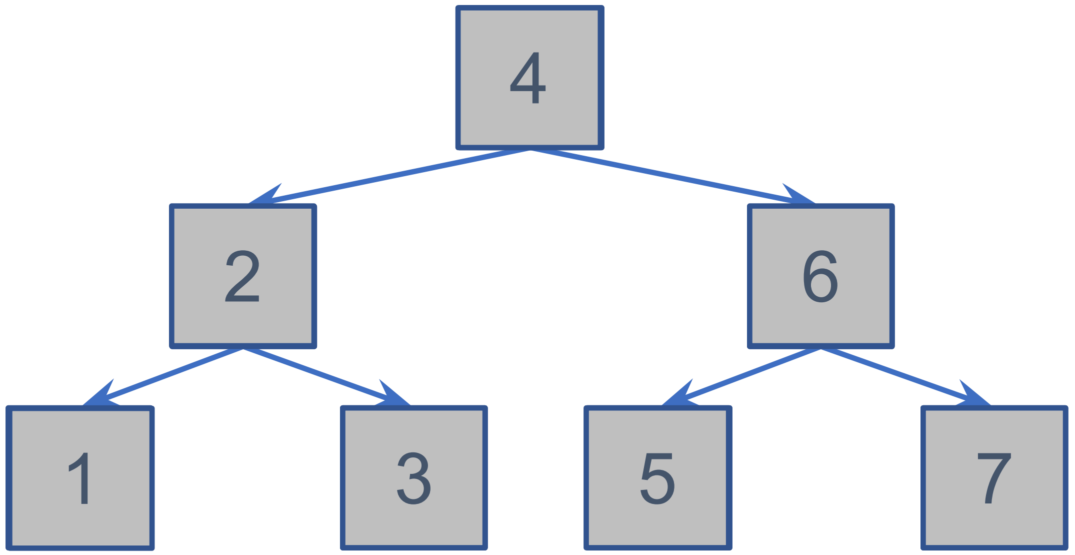


Depth First Traversals



Depth First Traversals

- Three types
  - Preorder, inorder, and postorder



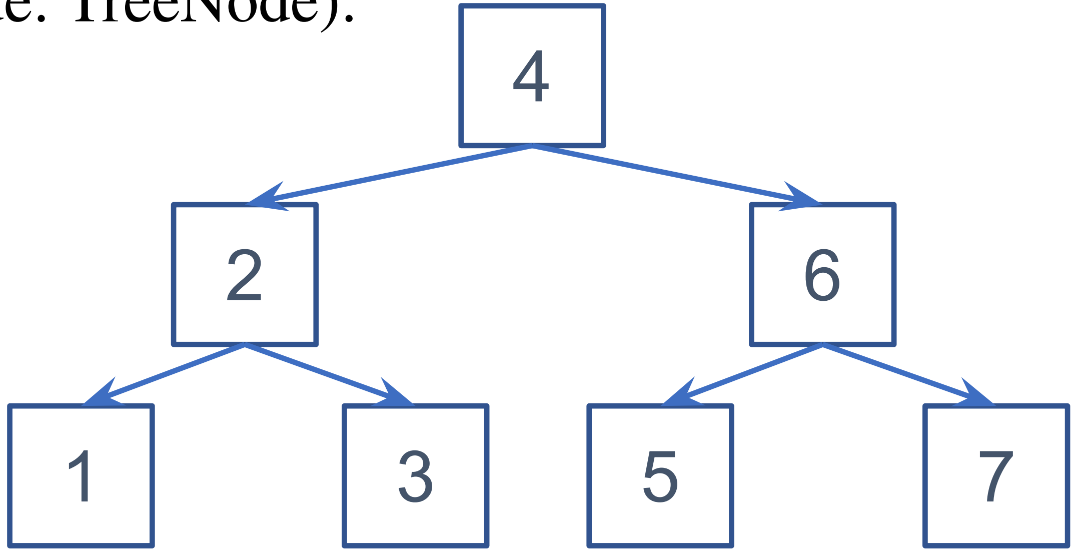
Depth-First Traversal

- Preorder -

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

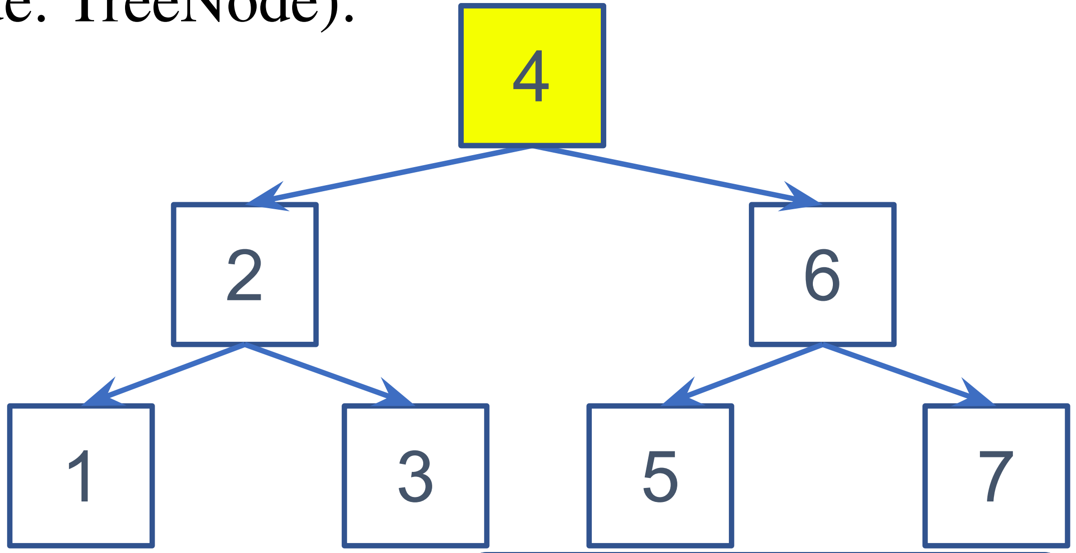
```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```



Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

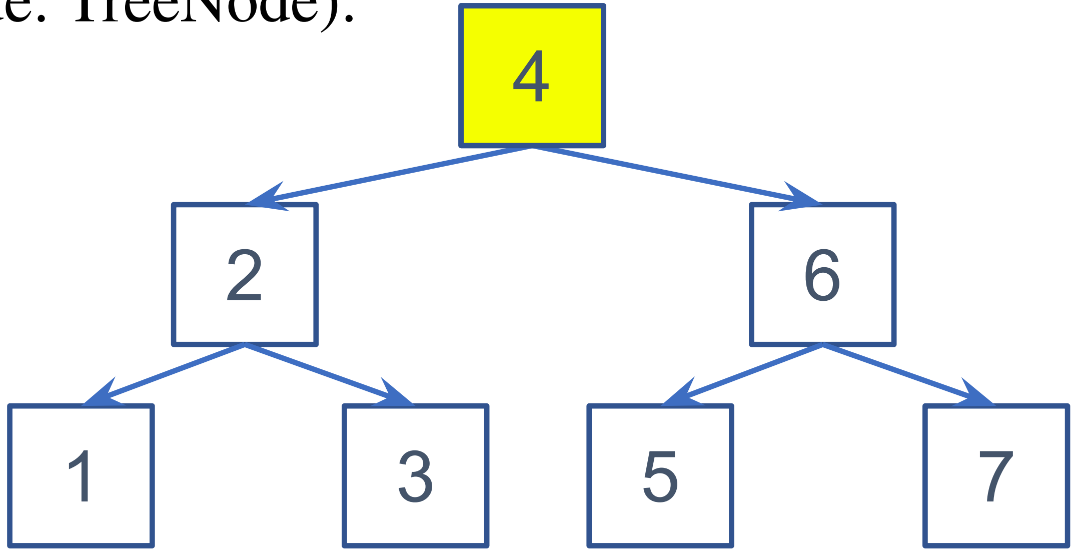


Out:

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

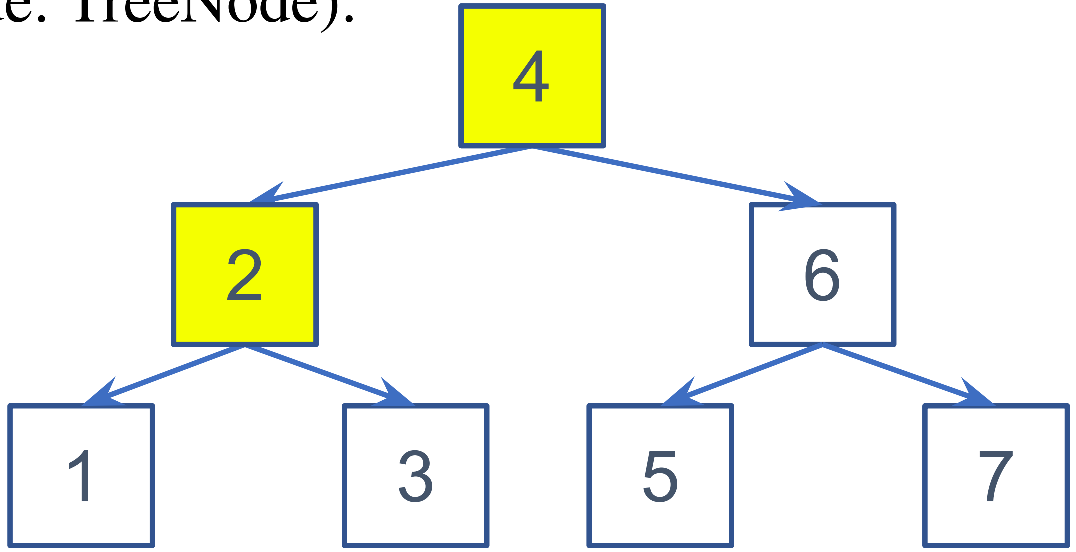


Out: 4

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

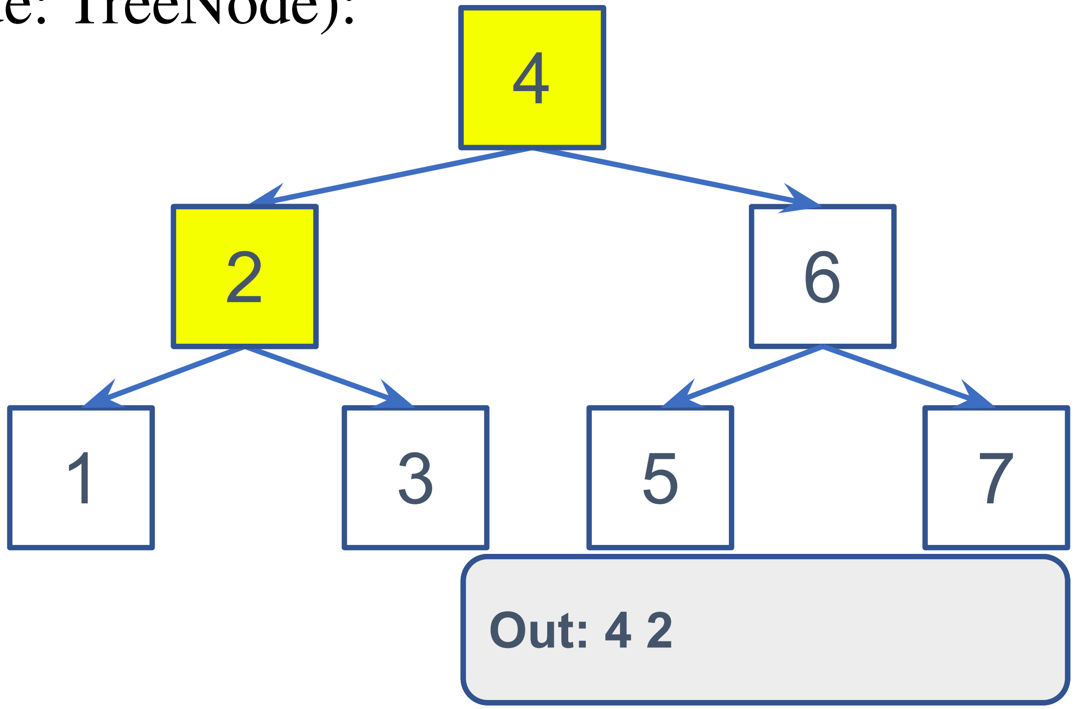


Out: 4

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

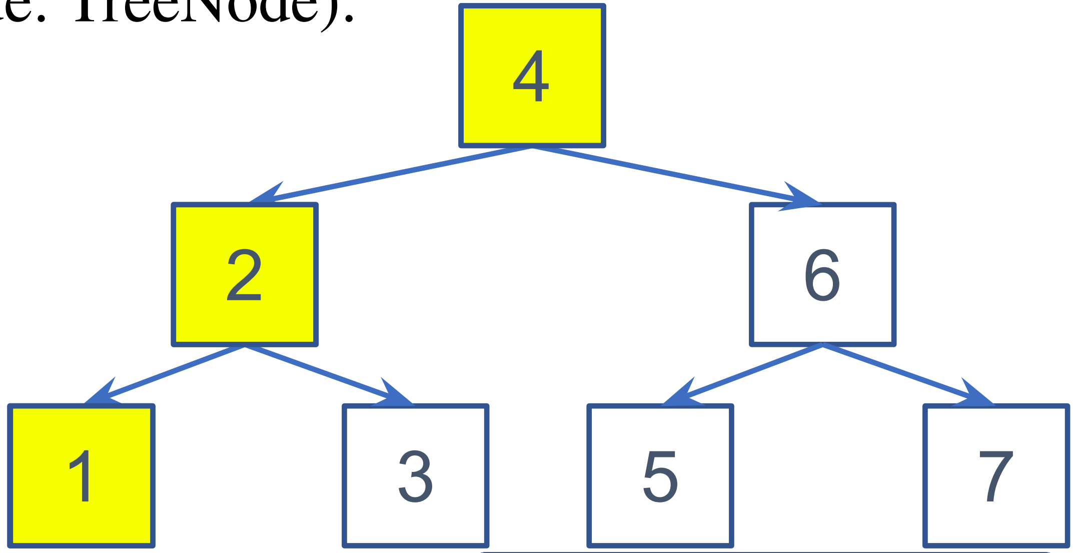
```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```



Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

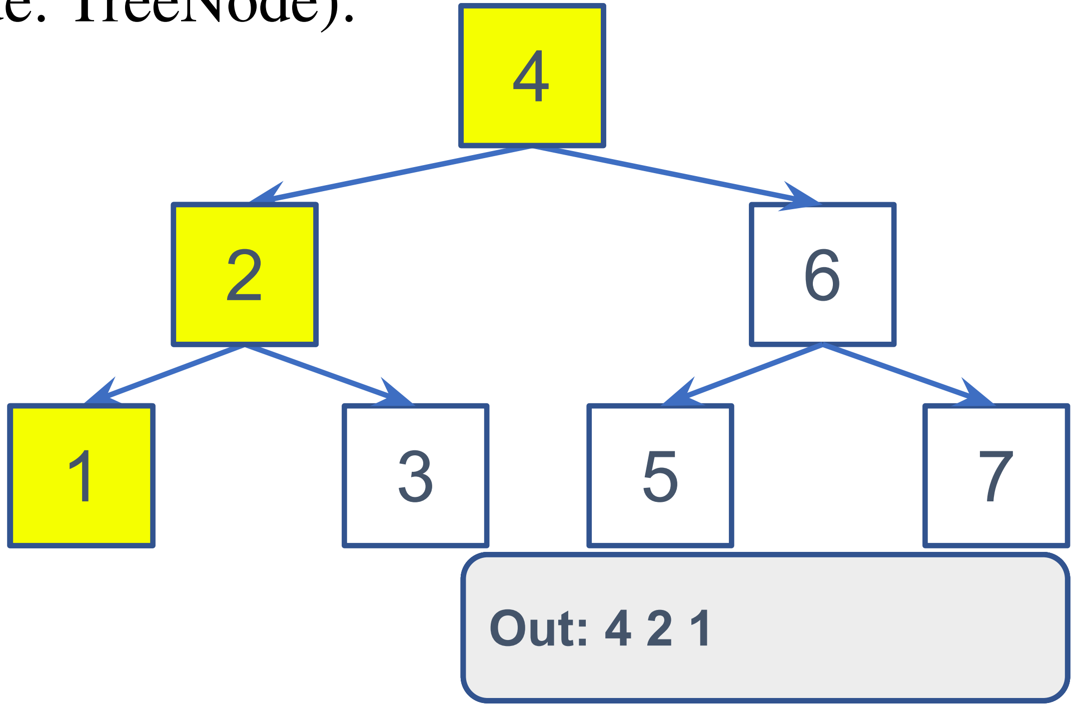


Out: 4 2

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

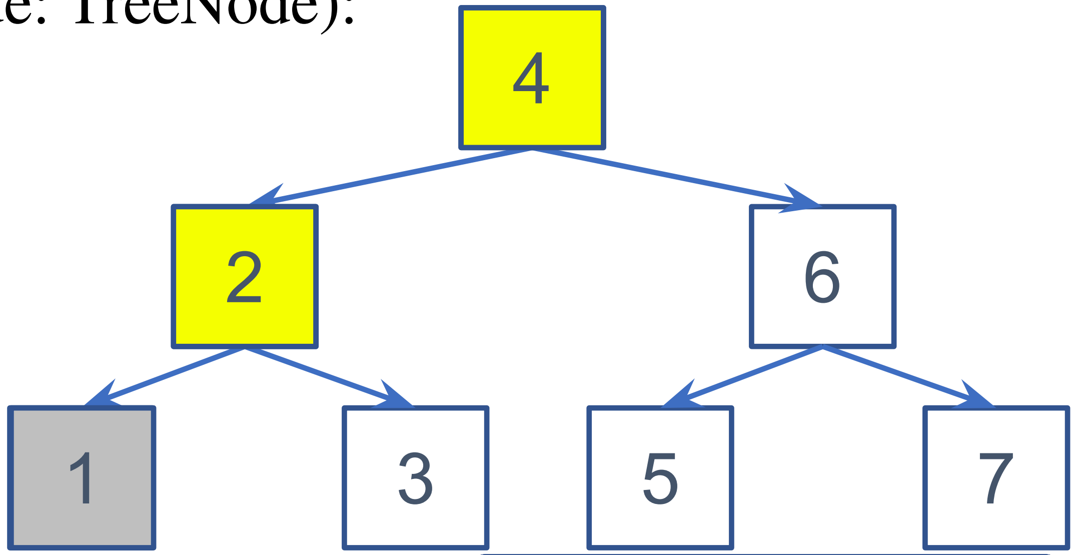
```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```



Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

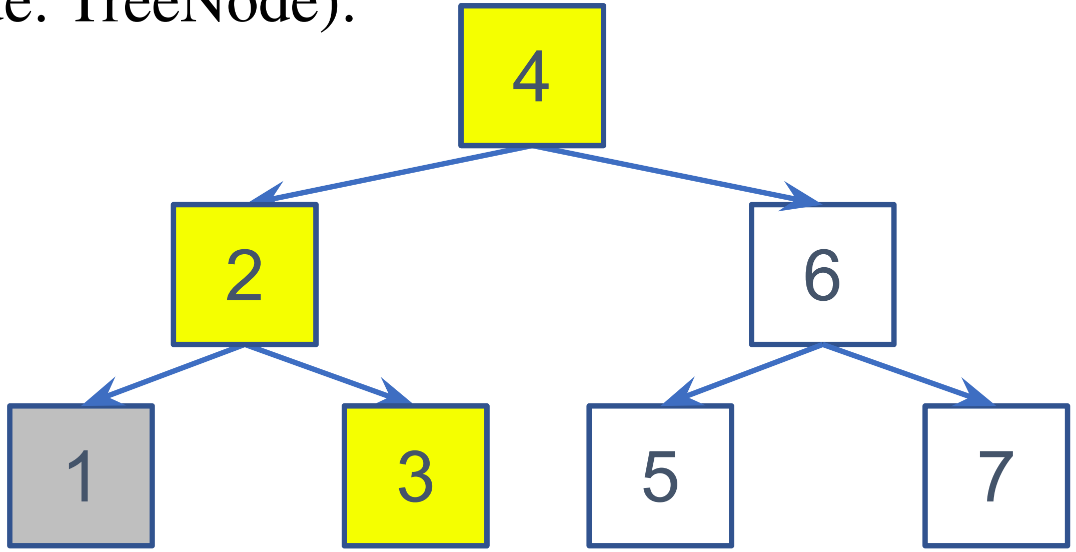


Out: 4 2 1

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

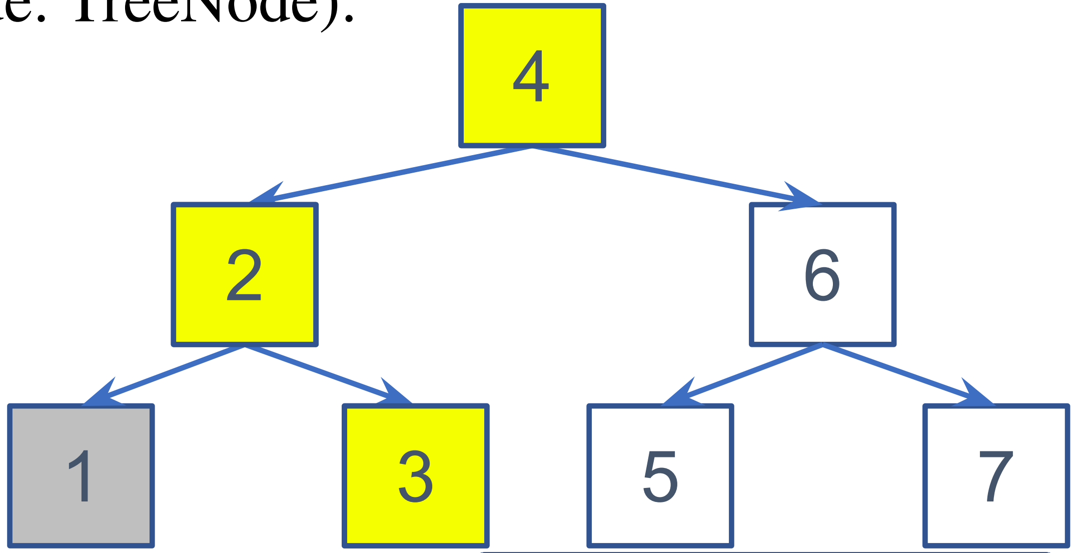


```
Out: 4 2 1
```

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

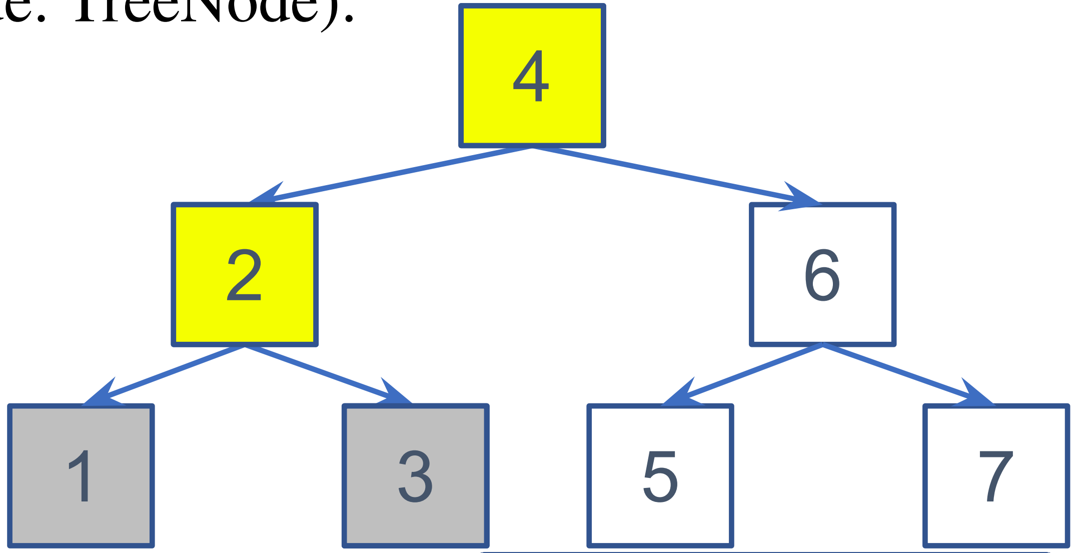


Out: 4 2 1 3

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

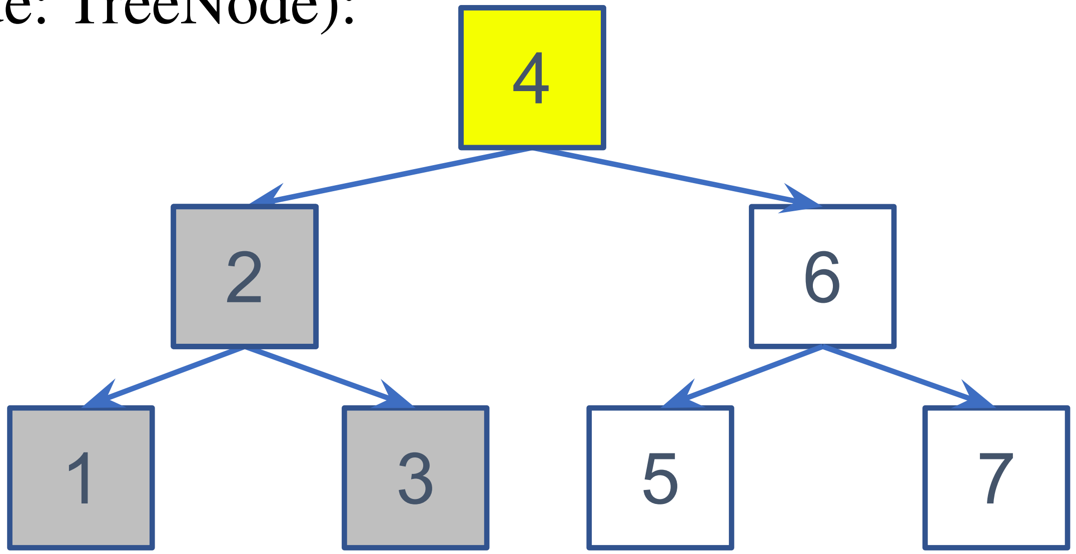


```
Out: 4 2 1 3
```

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

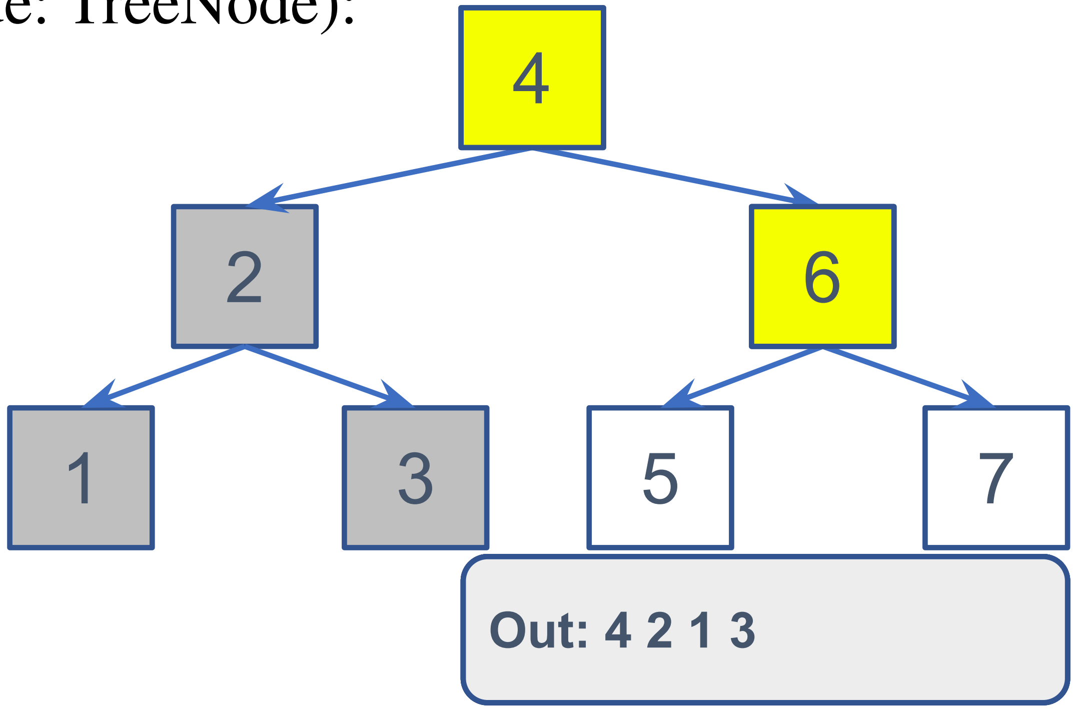


Out: 4 2 1 3

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

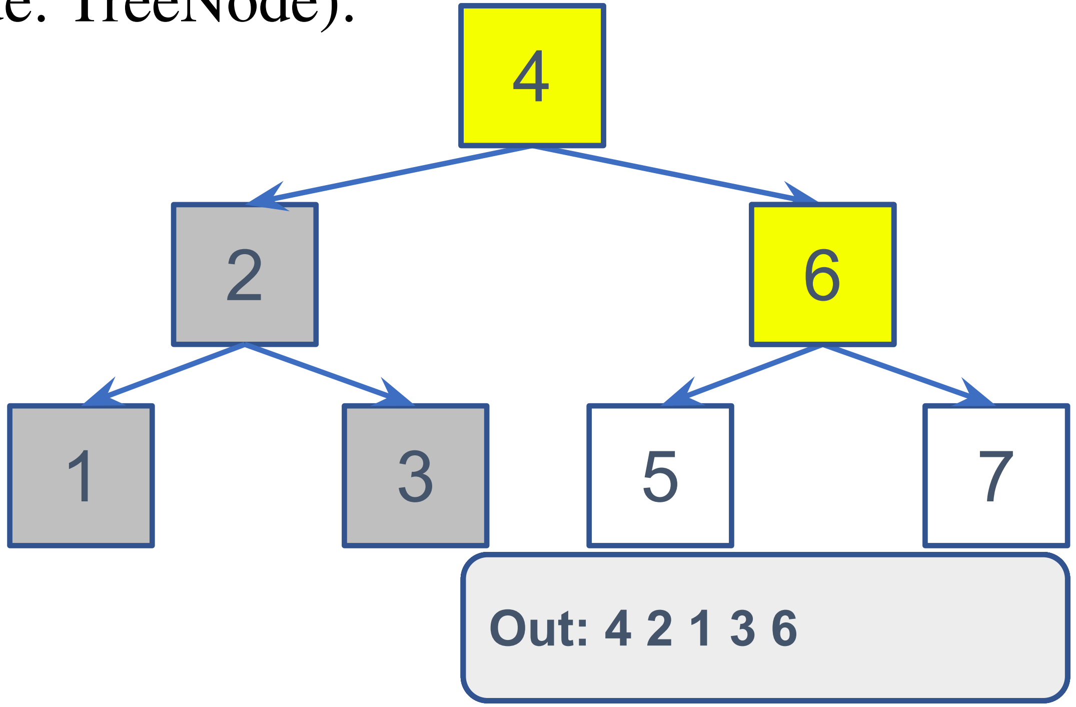
```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```



Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

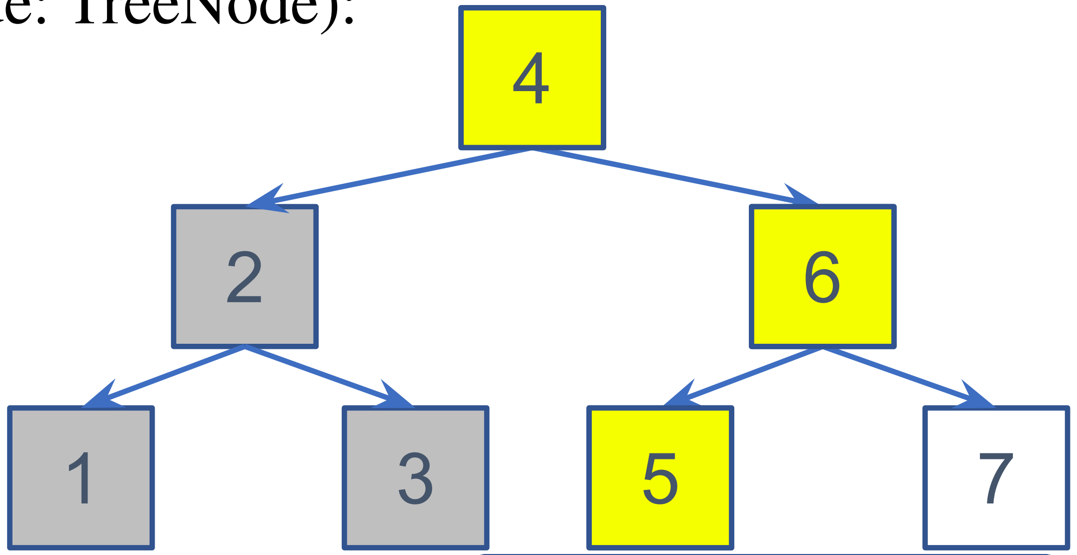
```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```



Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

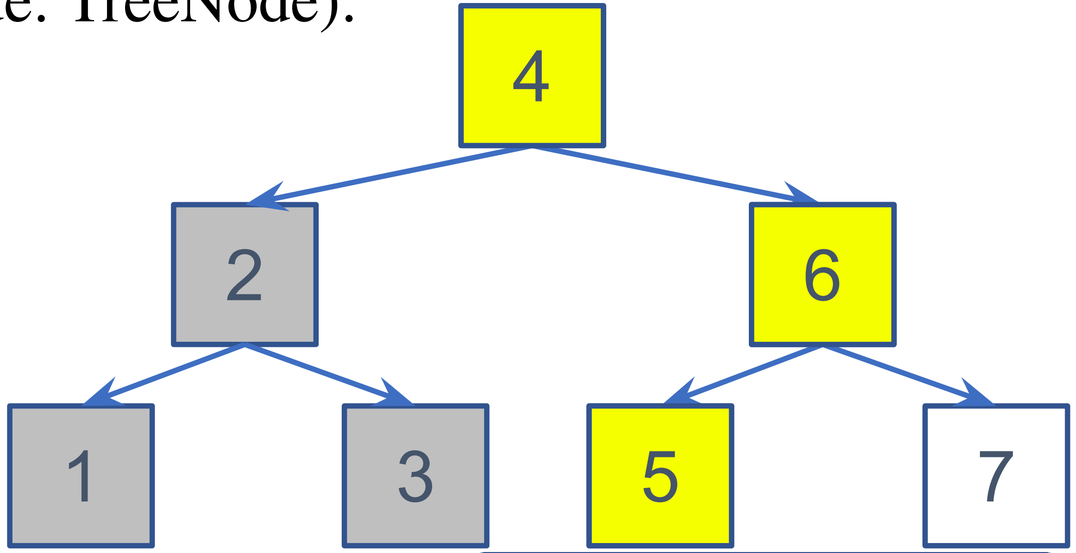


Out: 4 2 1 3 6

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

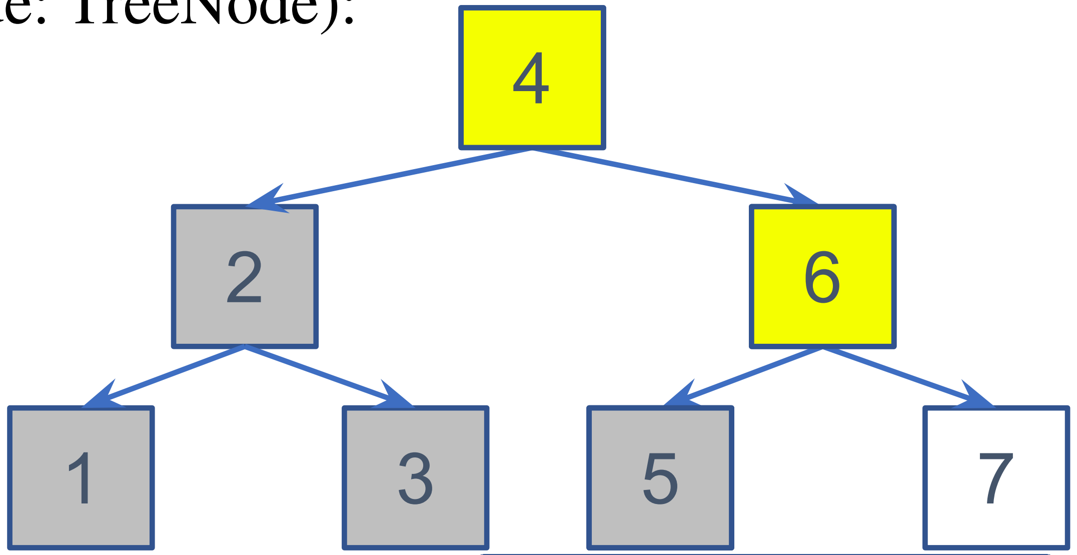


Out: 4 2 1 3 6 5

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

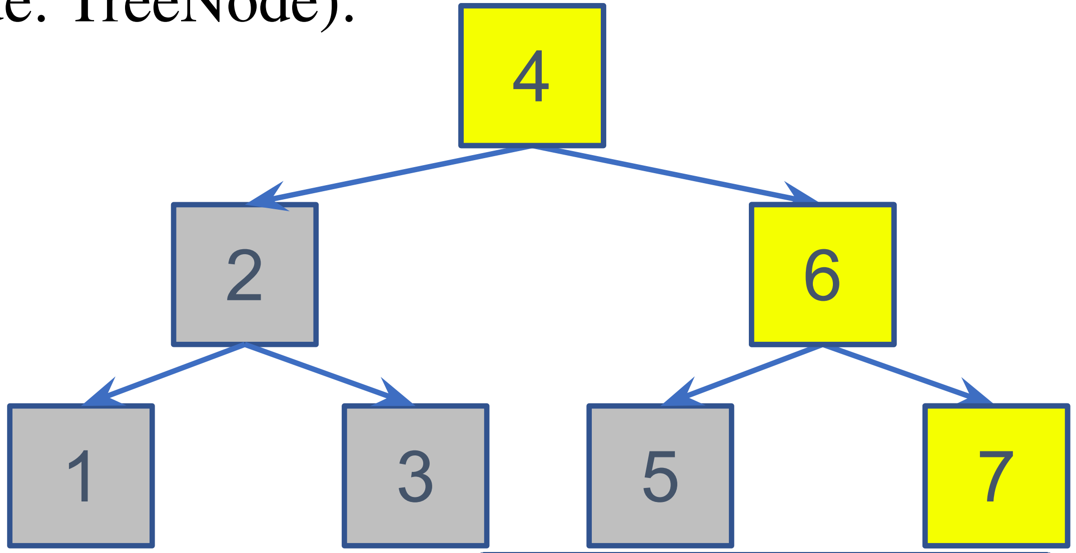


Out: 4 2 1 3 6 5

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

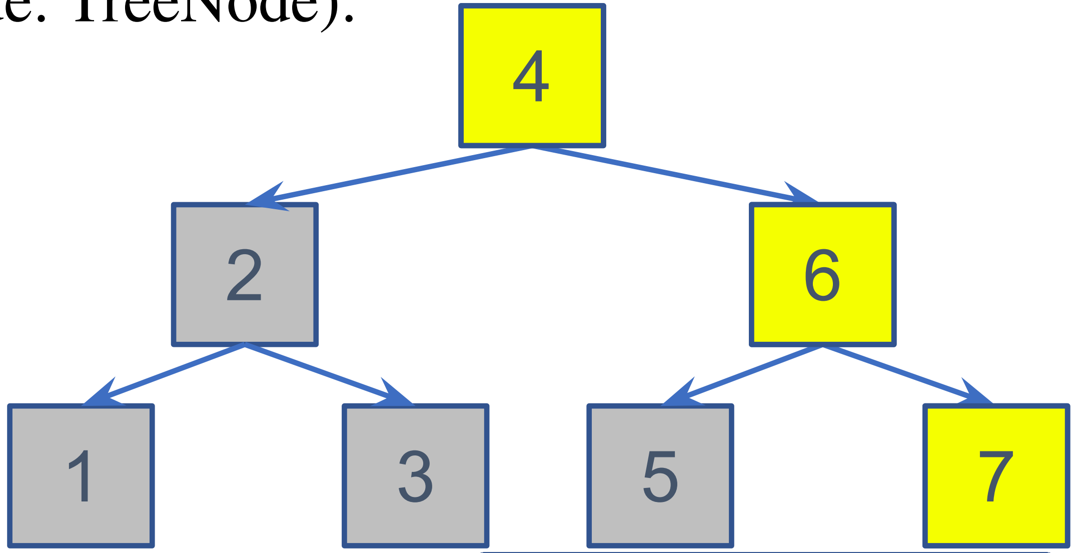


Out: 4 2 1 3 6 5

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

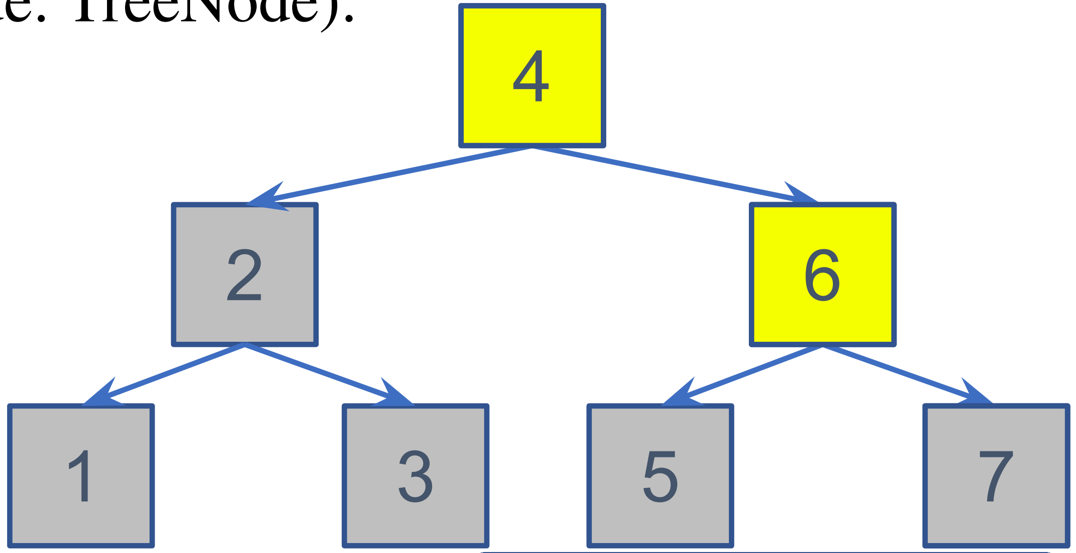


```
Out: 4 2 1 3 6 5 7
```

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right
  - class Tree():
  - def **visit**(self, node: TreeNode):
  - print(node.val)
  -
  - def **__DFT_preorderHelp**(self, curNode: TreeNode):
  - if curNode == None:
  - return
  - self.visit(curNode)
  - **for childNode in curNode.child:**
  - **self.__DFT_preorderHelp(childNode)**
  -
  - def **DFT_preorder**(self):
  - self.__DFT_preorderHelp(self.root)

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

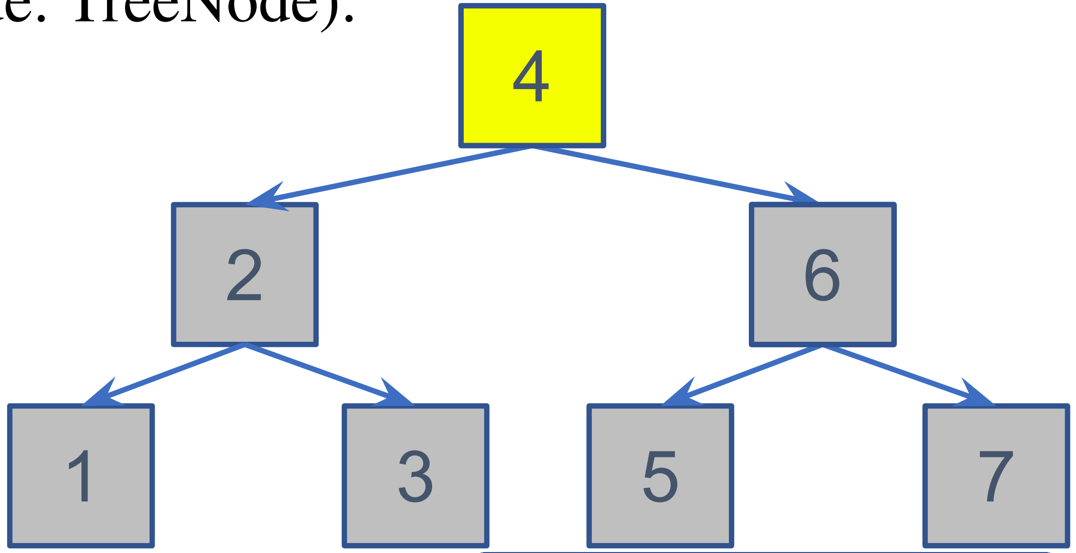


Out: 4 2 1 3 6 5 7

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```



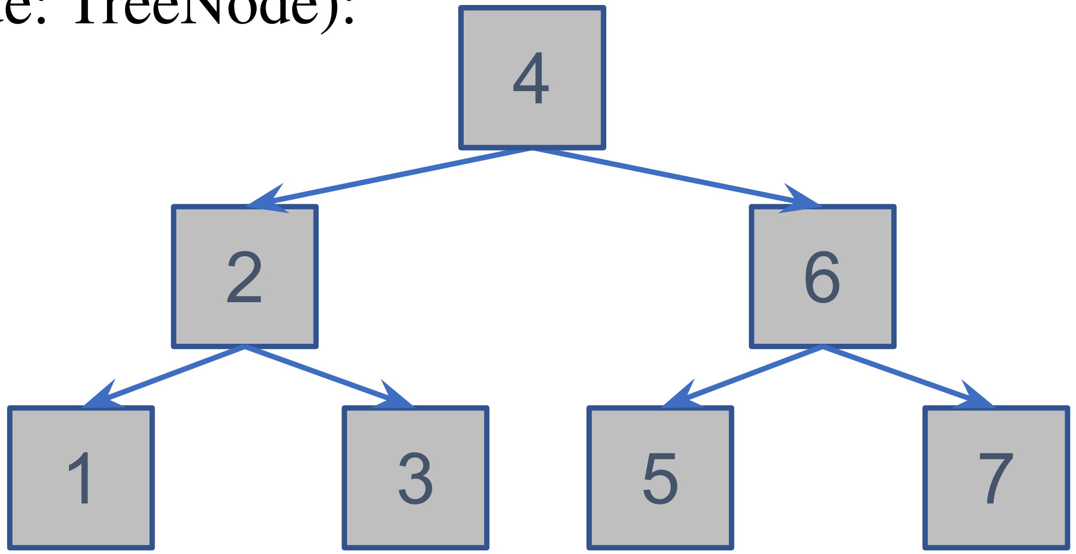
```
Out: 4 2 1 3 6 5 7
```

Depth First Traversals – Preorder

- Visit a node **before** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_preorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        self.visit(curNode)
        for childNode in curNode.child:
            self.__DFT_preorderHelp(childNode)

    def DFT_preorder(self):
        self.__DFT_preorderHelp(self.root)
```

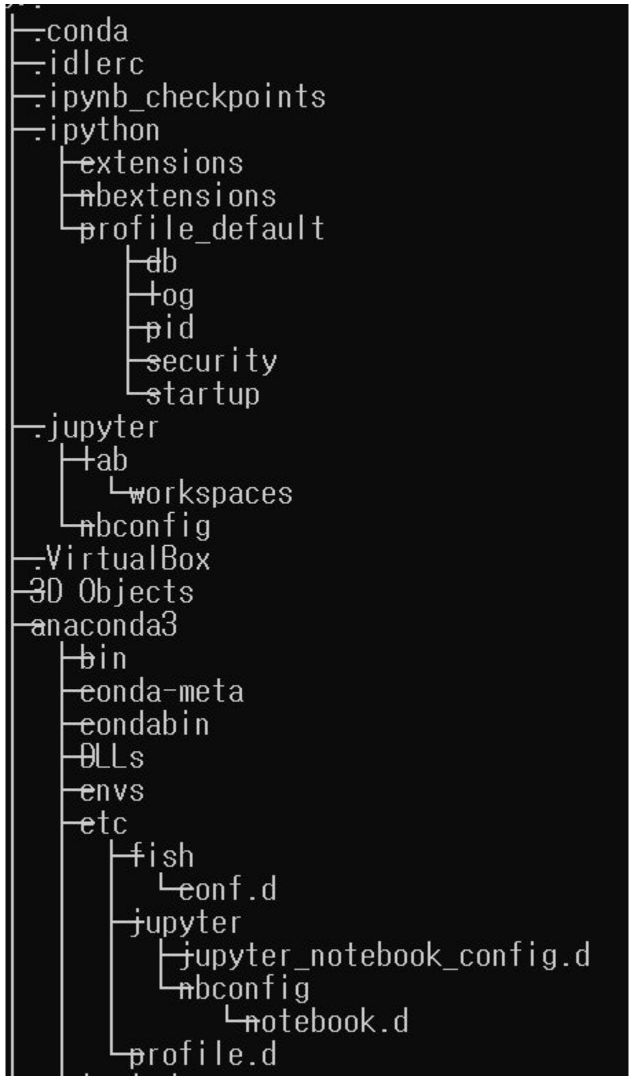
Done!



```
Out: 4 2 1 3 6 5 7
```

Depth First Traversals – Preorder

- **Application**: Directory listing (type "Tree" for fun)



```
├─.conda
├─.idlerc
├─.ipynb_checkpoints
├─.ipython
│  ├─extensions
│  ├─nbextensions
│  └─profile_default
│      ├─db
│      ├─log
│      ├─pid
│      ├─security
│      └─startup
├─.jupyter
│  ├─lab
│  │  └─workspaces
│  └─nbconfig
├─.VirtualBox
├─3D Objects
├─anaconda3
│  ├─bin
│  ├─conda-meta
│  ├─condabin
│  ├─DLLs
│  ├─envs
│  ├─etc
│  │  ├─fish
│  │  │  └─conf.d
│  │  ├─jupyter
│  │  │  ├─jupyter_notebook_config.d
│  │  │  └─nbconfig
│  │  │      └─notebook.d
│  │  └─profile.d
```

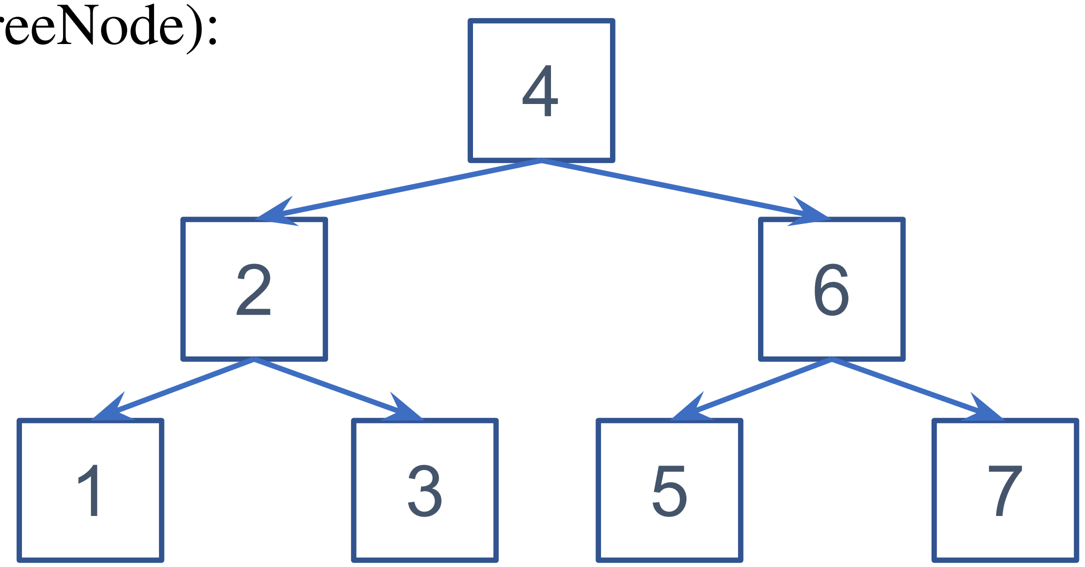
Depth-First Traversal

- Inorder -

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

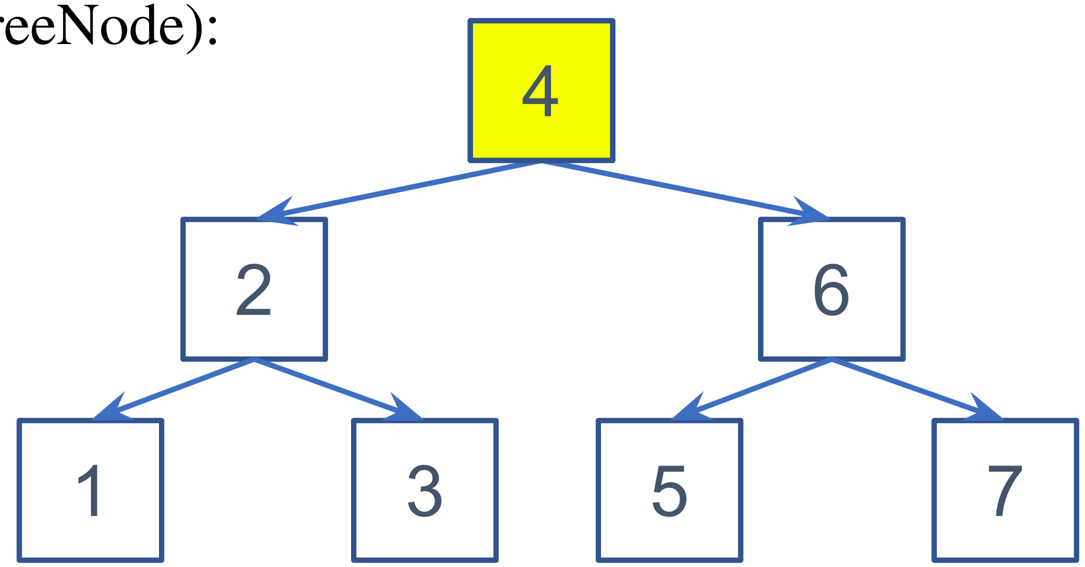


Out:

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

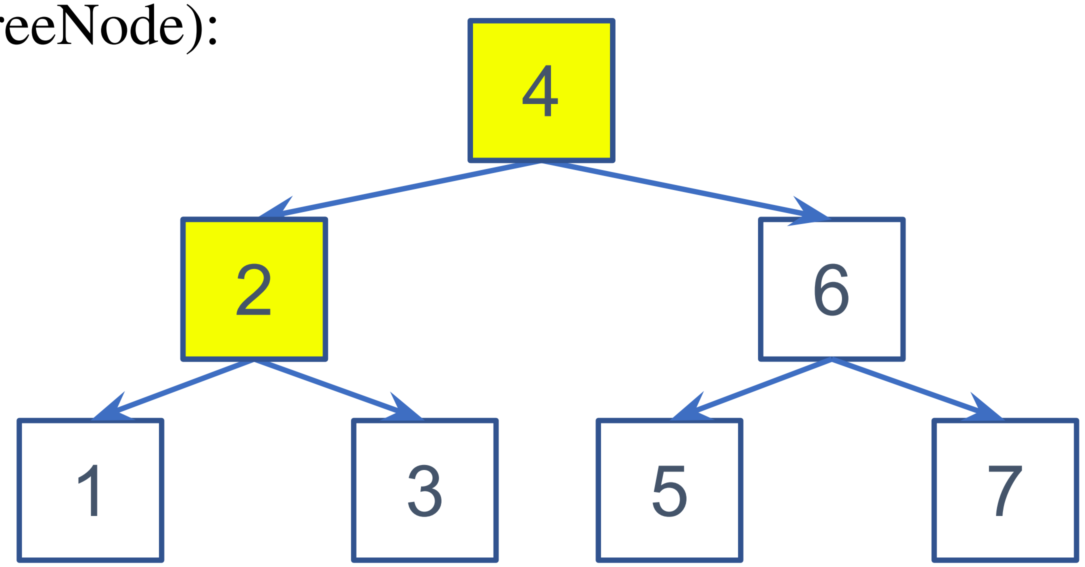


Out:

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

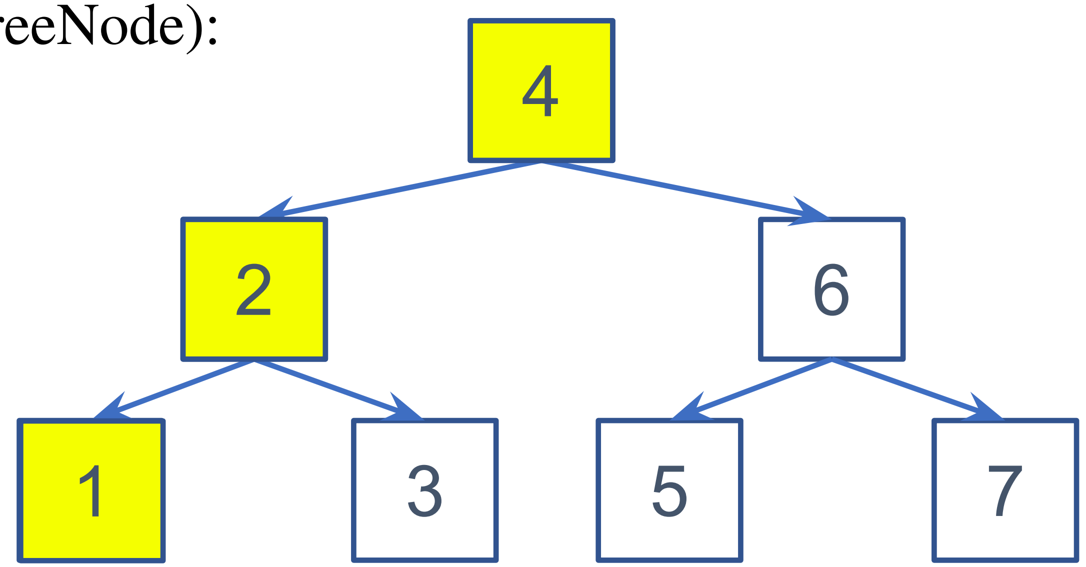


Out:

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

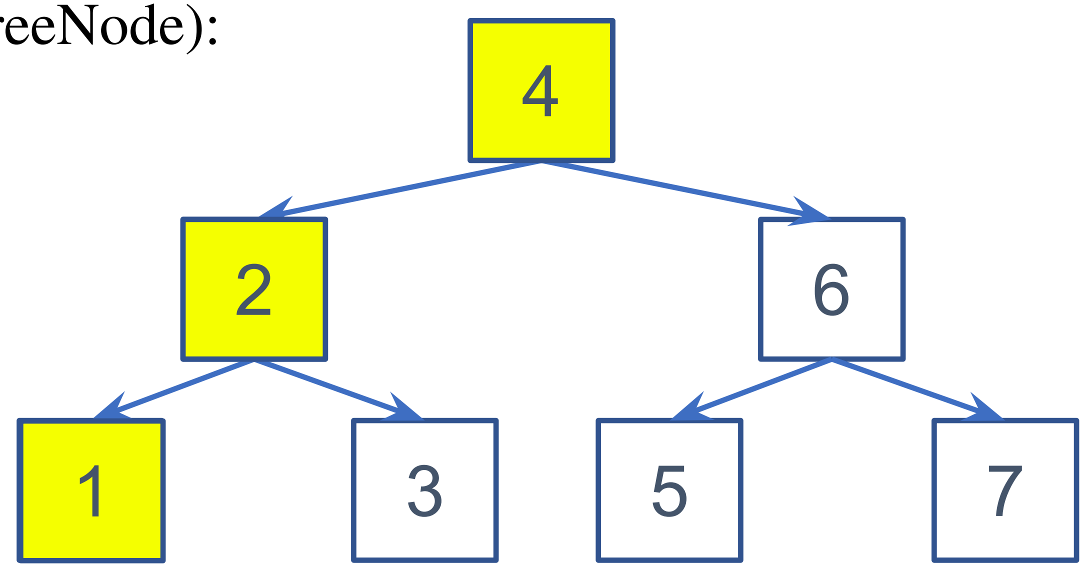


Out:

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

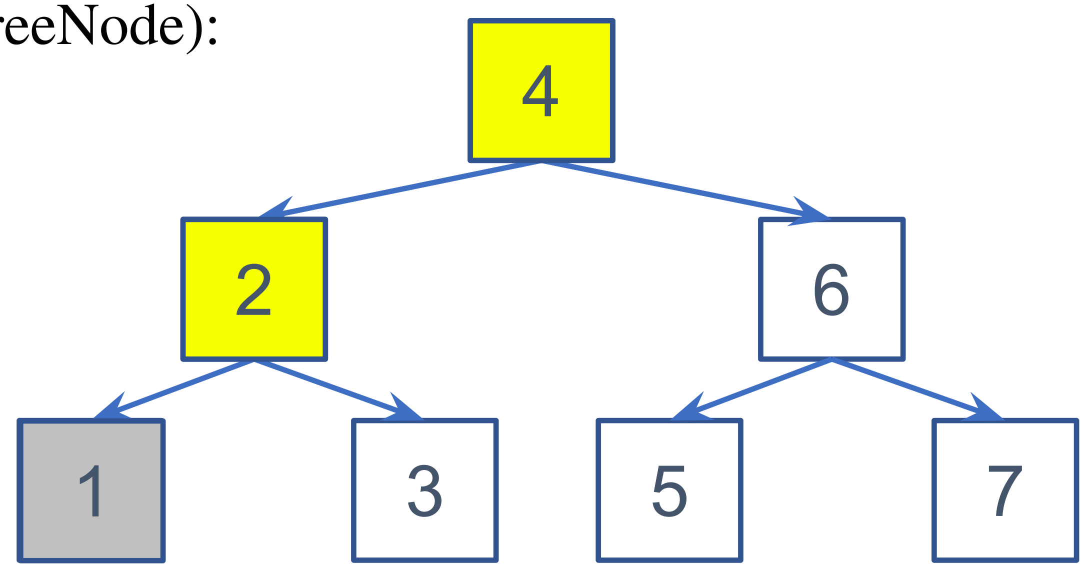


```
Out: 1
```

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

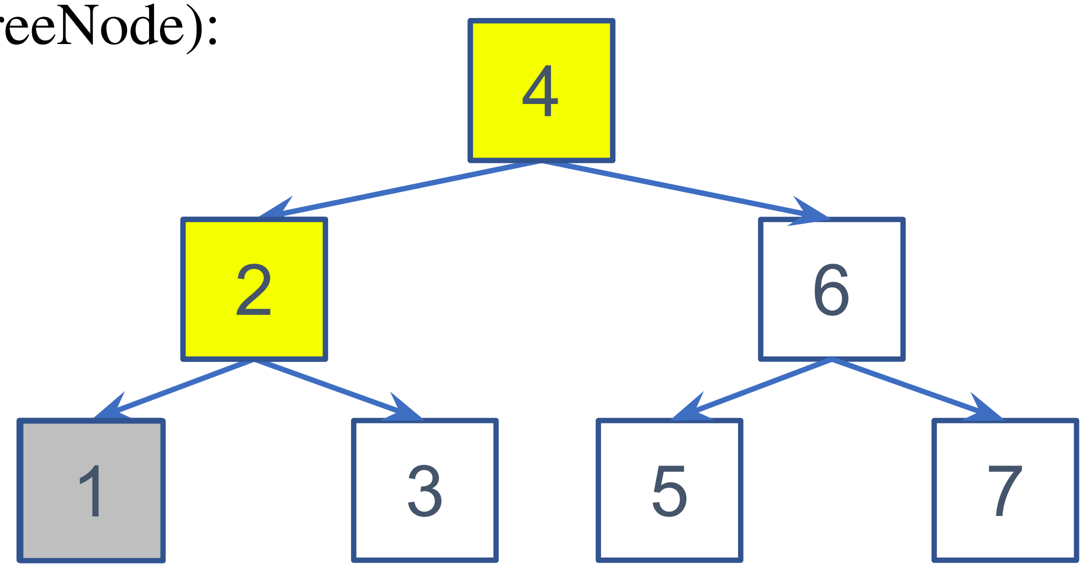


```
Out: 1
```

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

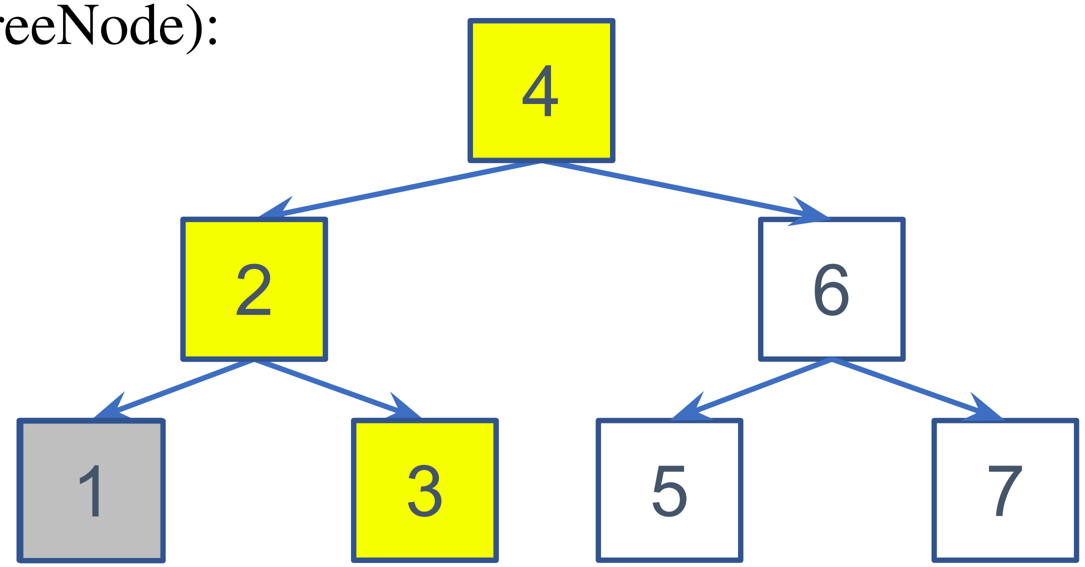


Out: 1 2

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

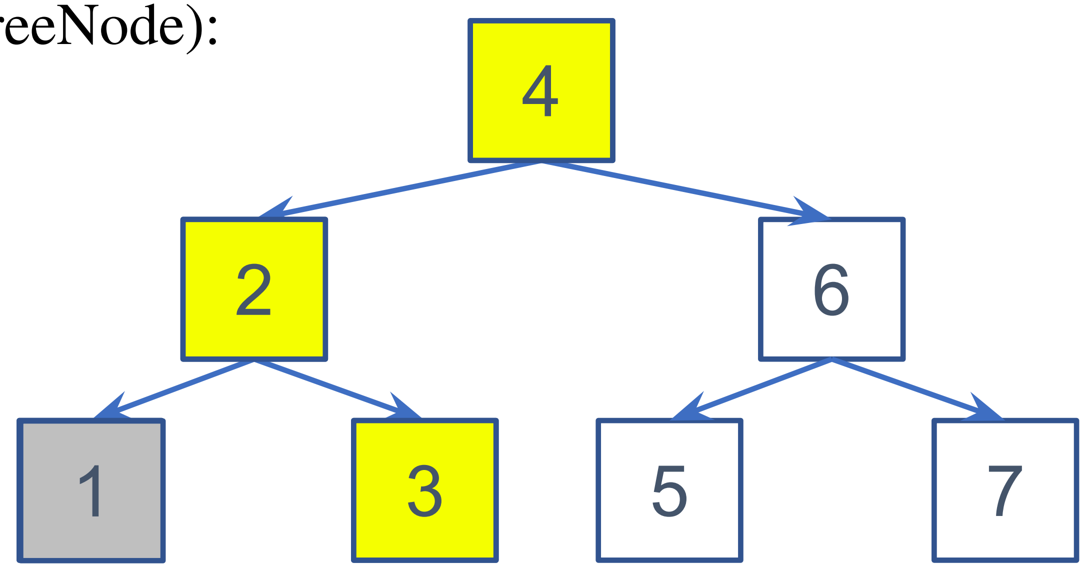


```
Out: 1 2
```

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

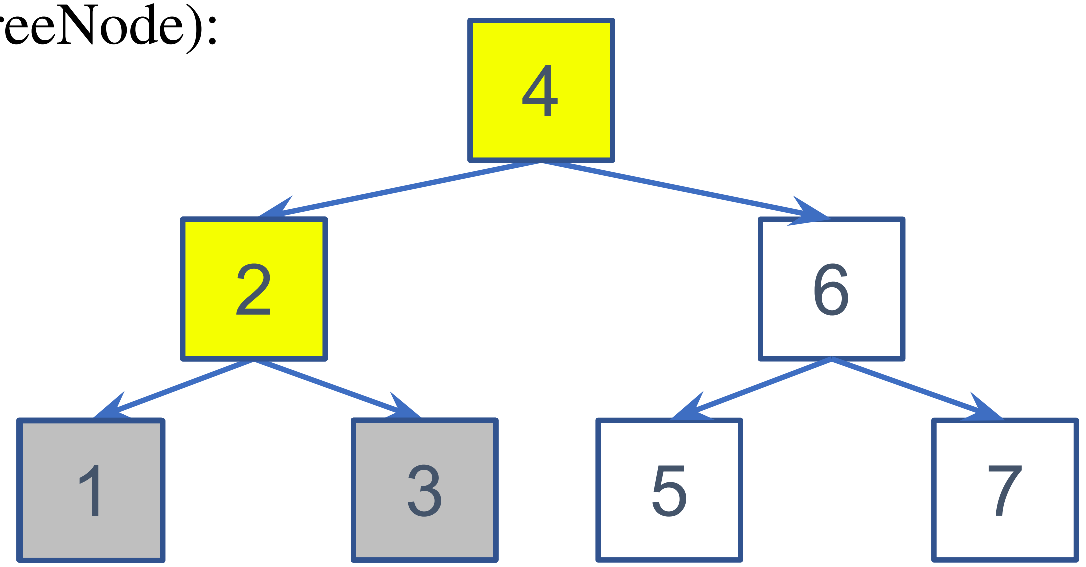


```
Out: 1 2 3
```

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

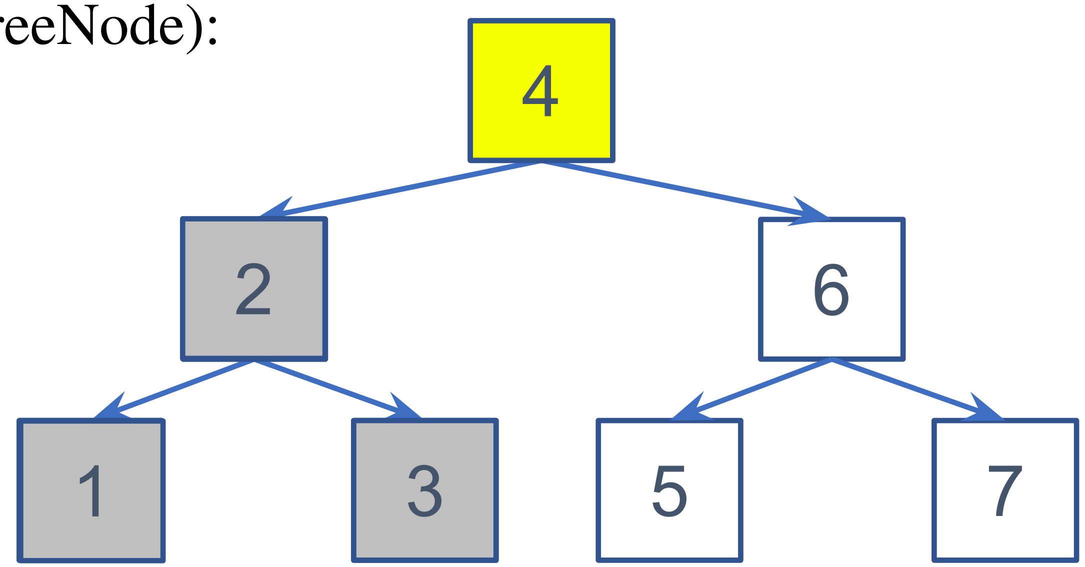


Out: 1 2 3

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

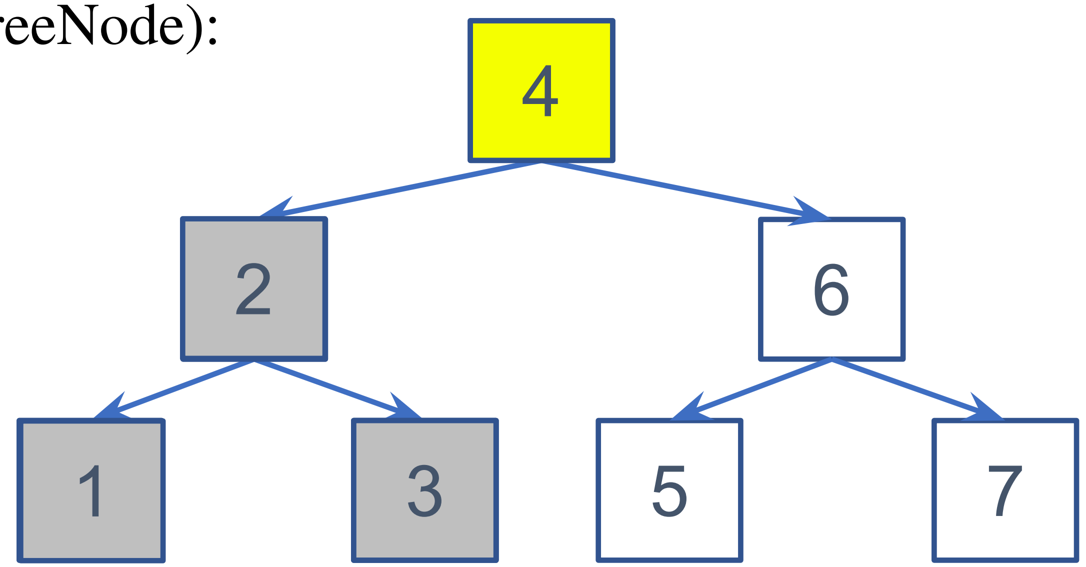


Out: 1 2 3

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

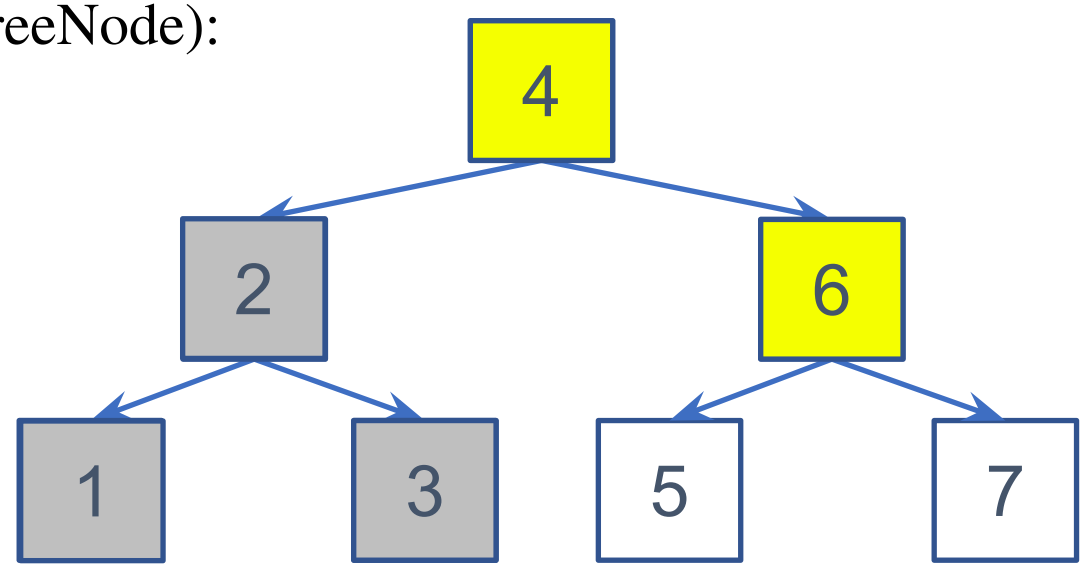


Out: 1 2 3 4

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

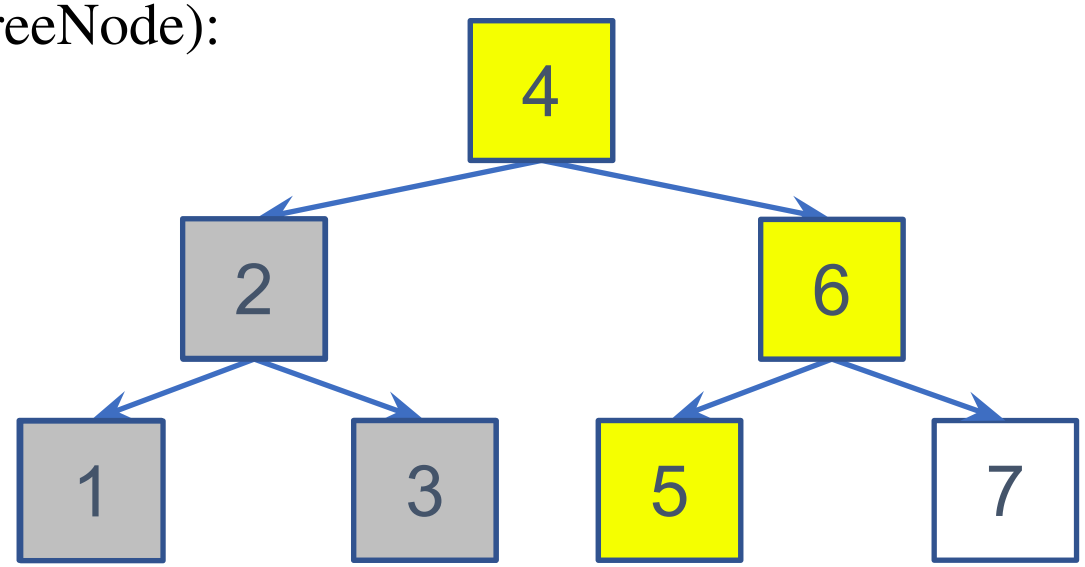


Out: 1 2 3 4

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

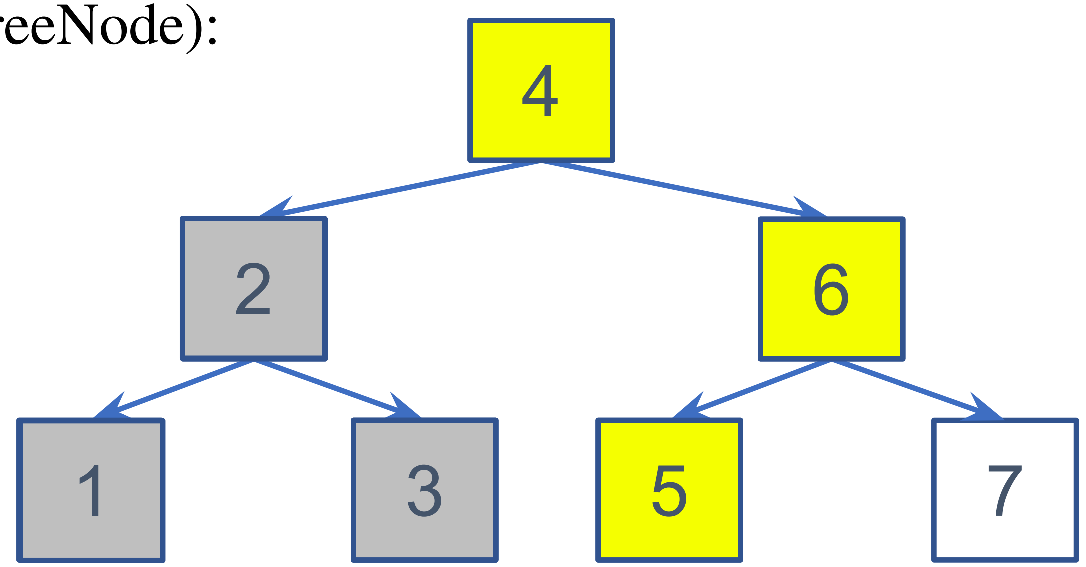


Out: 1 2 3 4

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

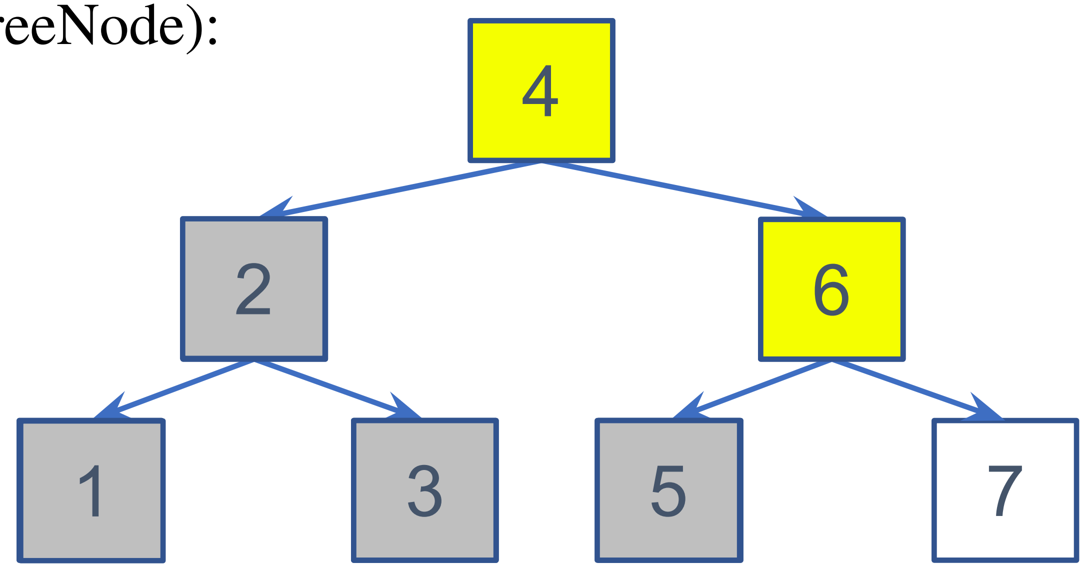


Out: 1 2 3 4 5

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

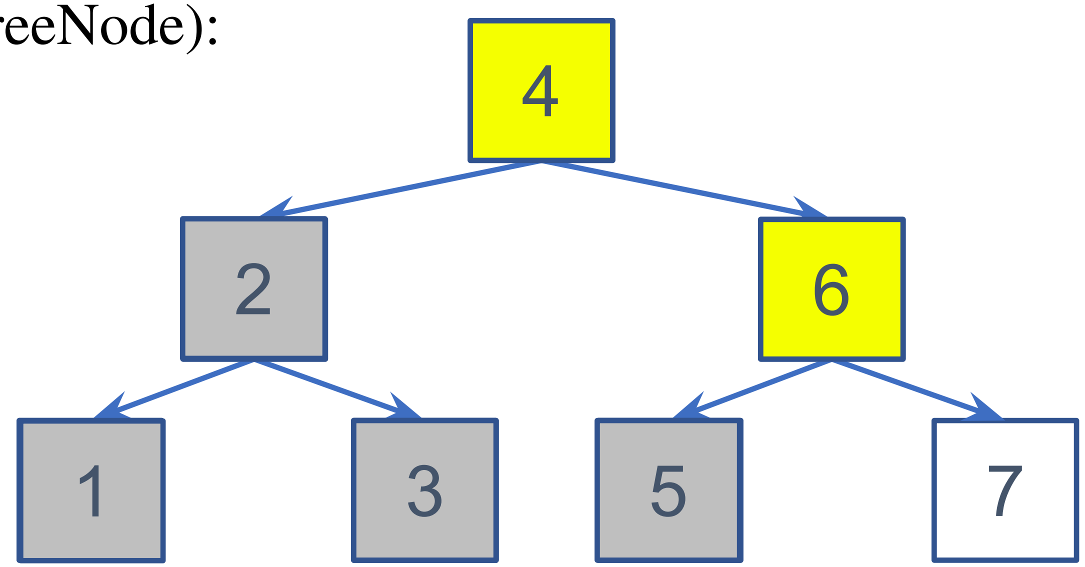


Out: 1 2 3 4 5

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

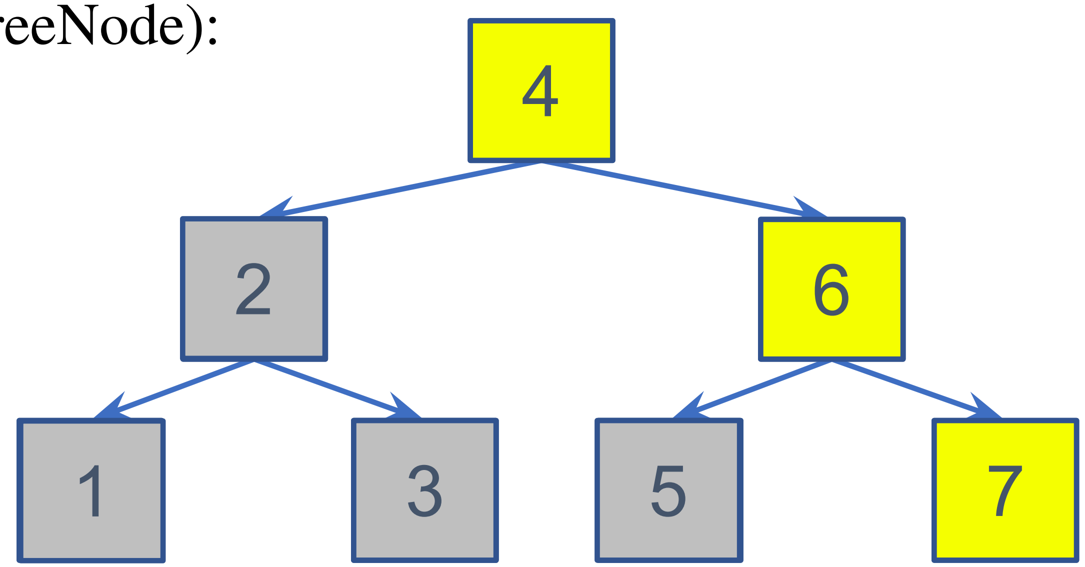


```
Out: 1 2 3 4 5 6
```

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

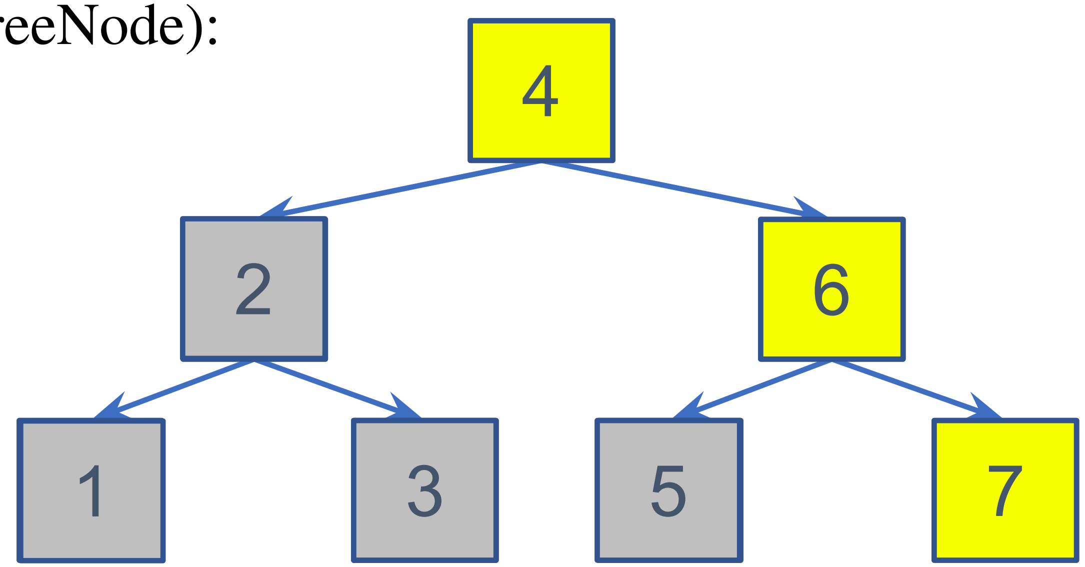


Out: 1 2 3 4 5 6

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

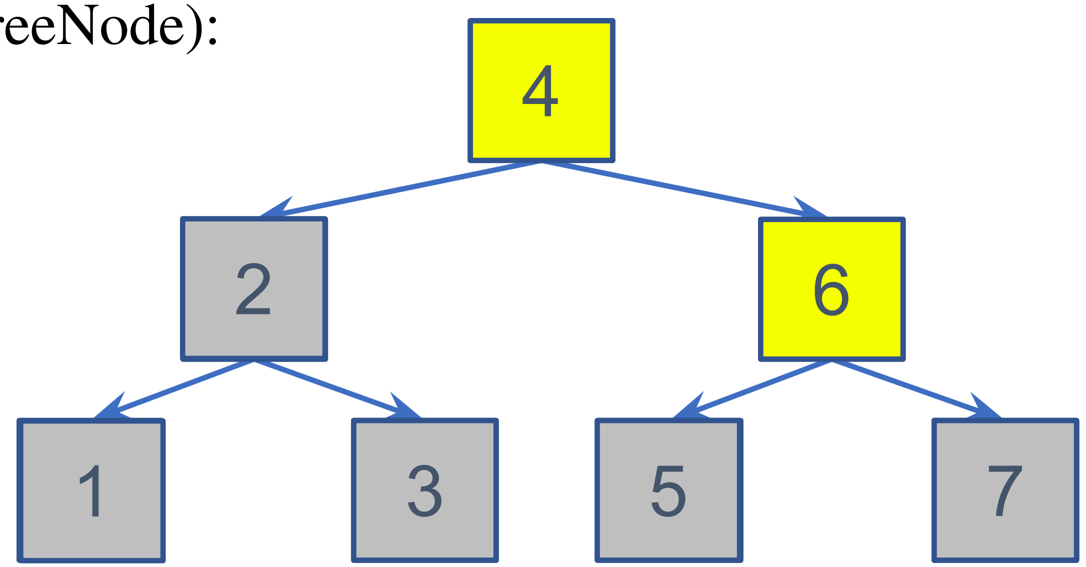


Out: 1 2 3 4 5 6 7

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

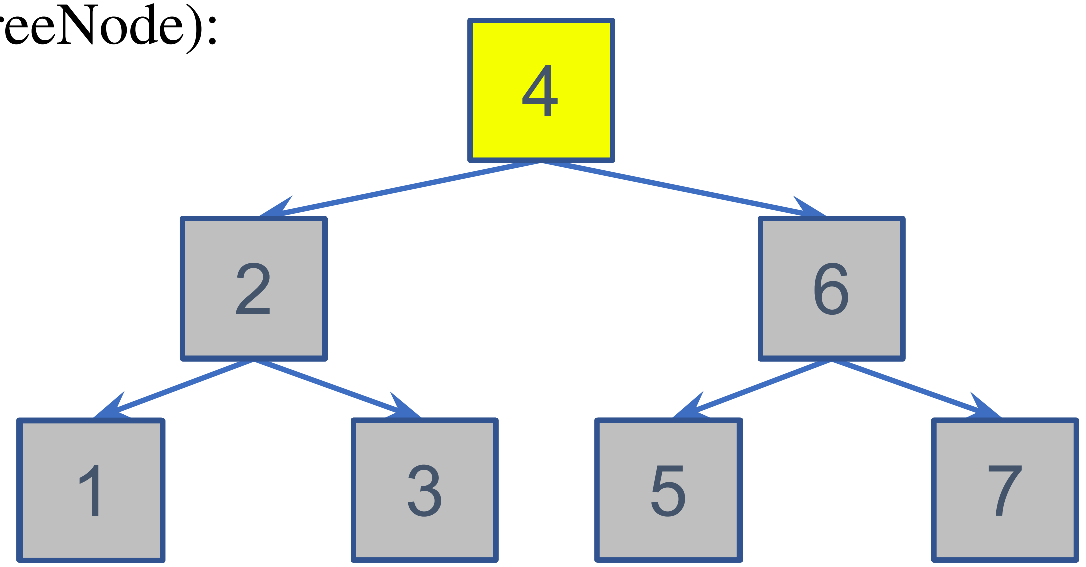


```
Out: 1 2 3 4 5 6 7
```

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```

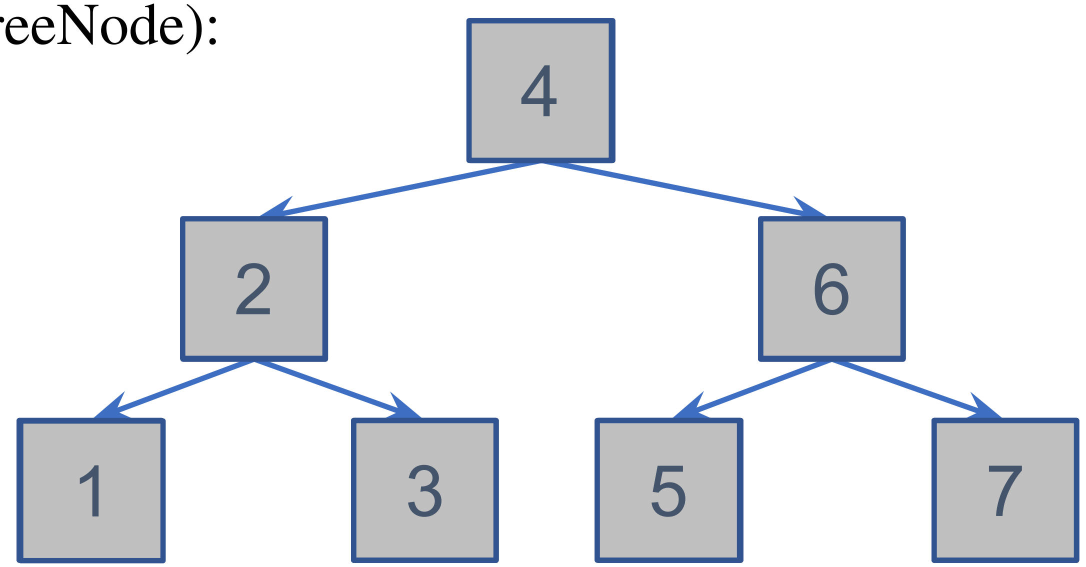


Out: 1 2 3 4 5 6 7

Depth First Traversals – Inorder

- Traverse a node's children from left to right and visit the node **in the middle**

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_inorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            if i == 1:
                self.visit(curNode)
            self.__DFT_inorderHelp(curNode.child[i])

    def DFT_inorder(self):
        self.__DFT_inorderHelp(self.root)
```



```
Out: 1 2 3 4 5 6 7
```

Depth First Traversals – Inorder

- **Application**: Covert a binary search tree to a sorted list (Flattening a BST)

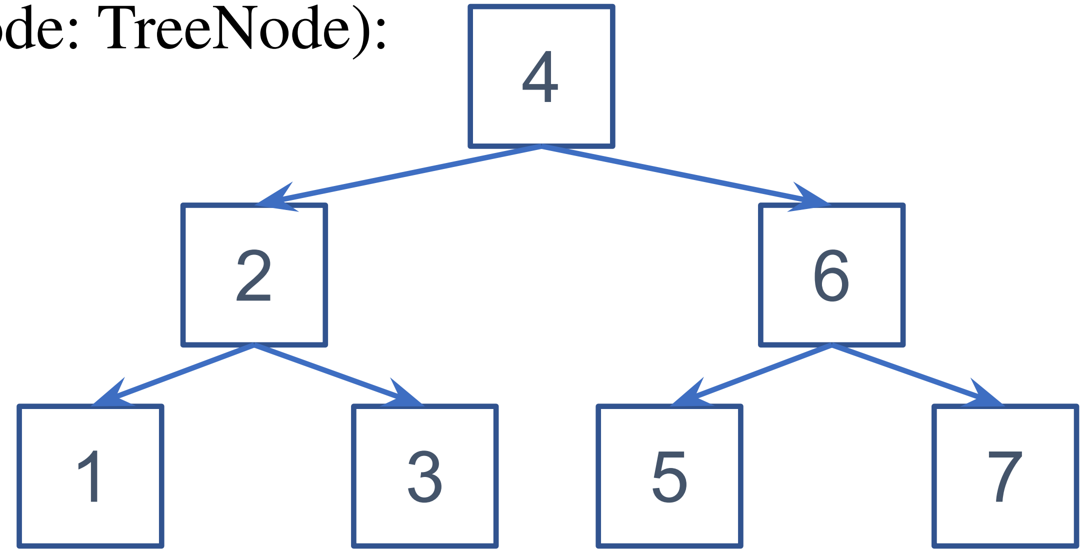
Depth-First Traversal

- Postorder -

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

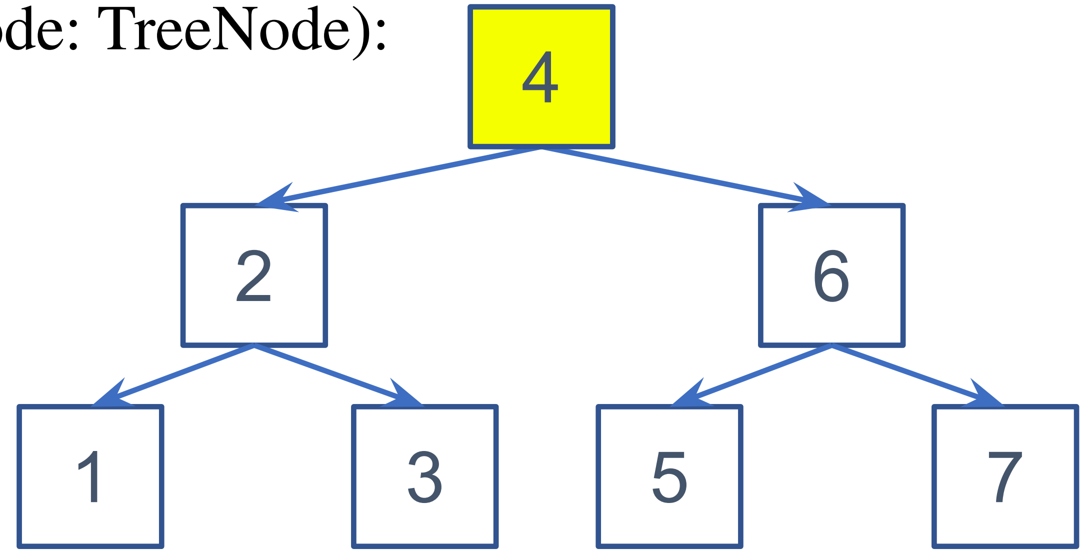


Out:

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
      if curNode == None:
          return
      for i in range(len(curNode.child)):
          self.__DFT_postorderHelp(curNode.child[i])
      self.visit(curNode)

    def DFT_postorder(self):
      self.__DFT_postorderHelp(self.root)
```

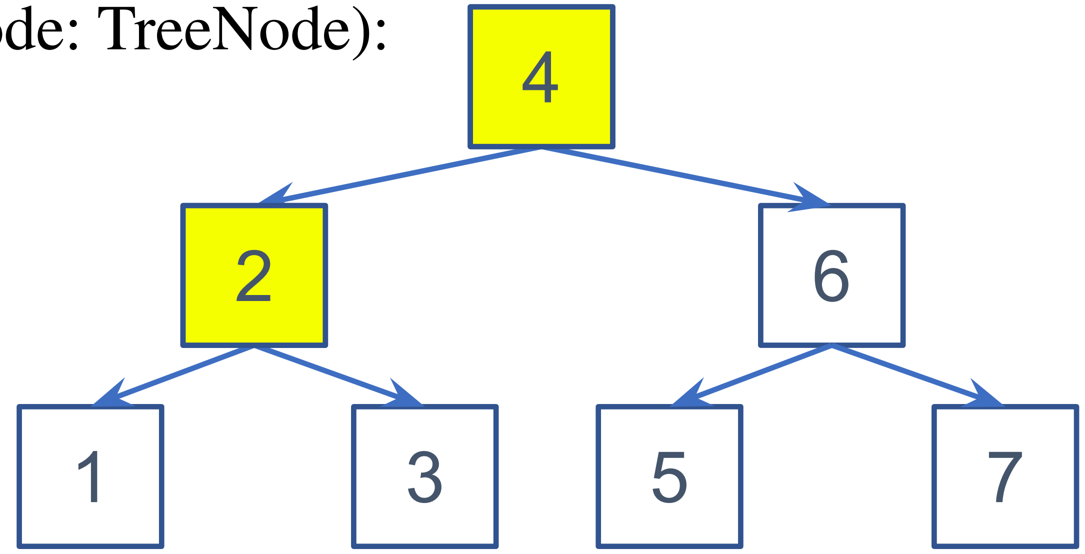


Out:

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

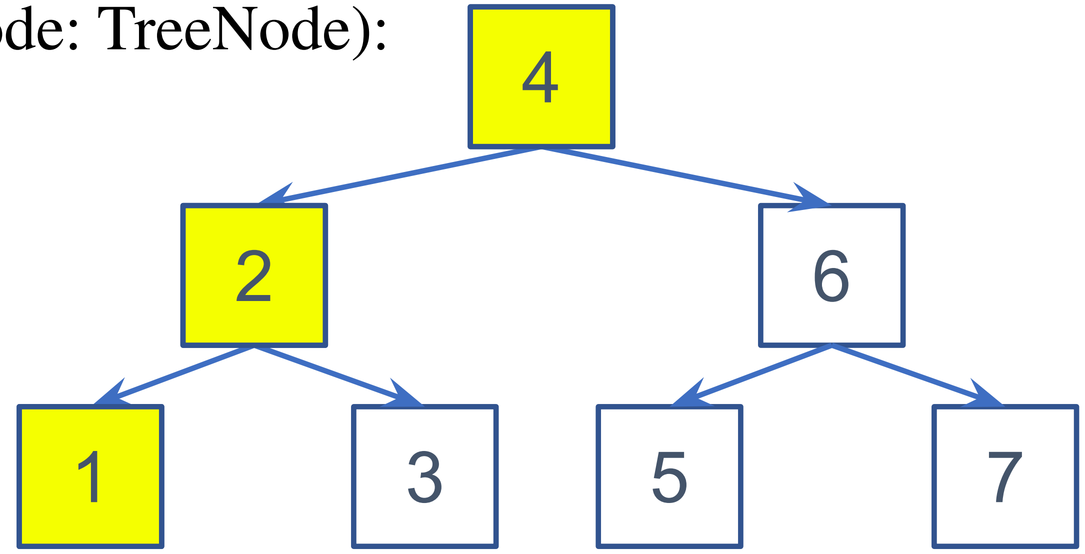


Out:

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

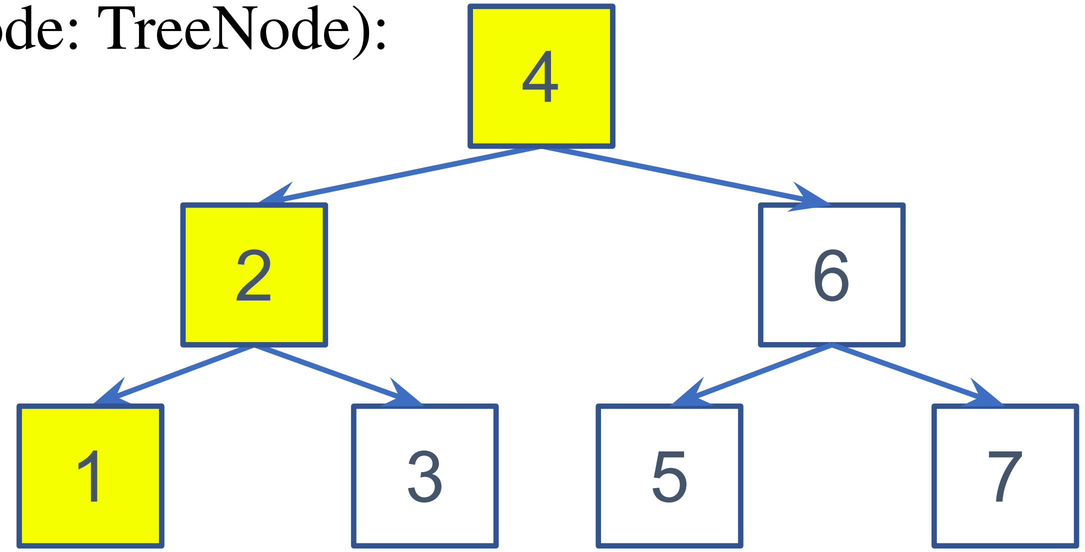


Out:

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

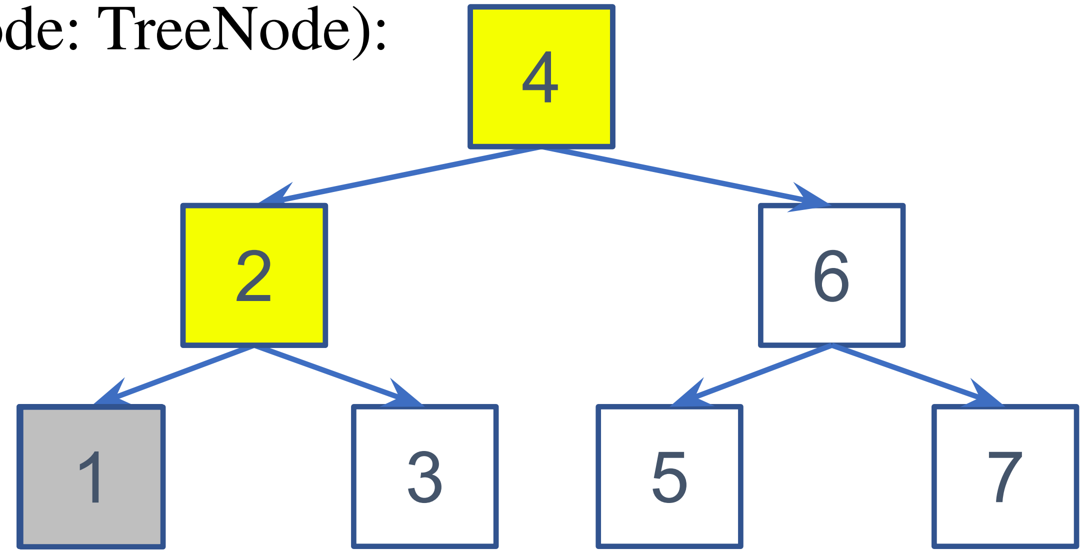


```
Out: 1
```

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

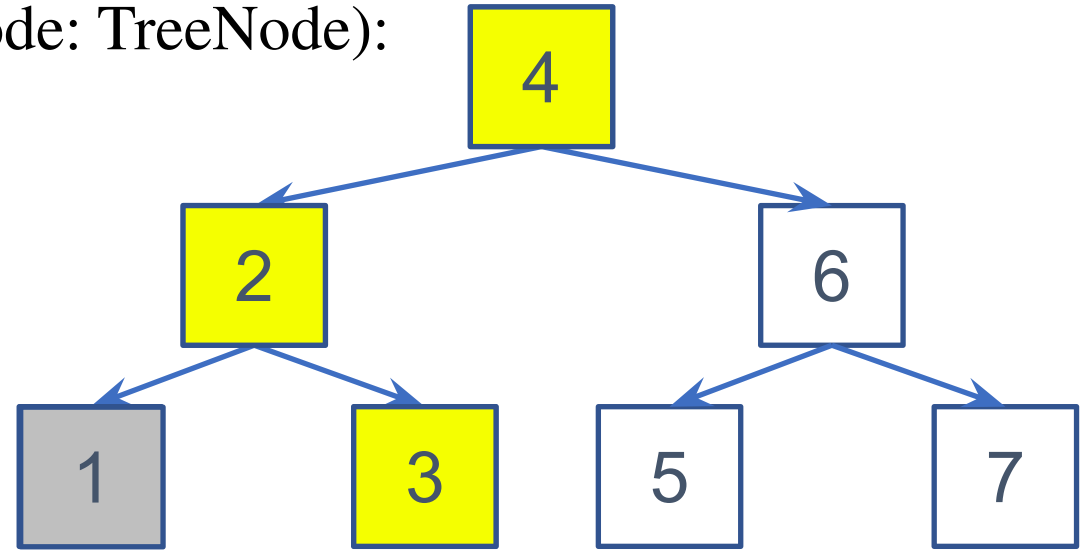


Out: 1

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

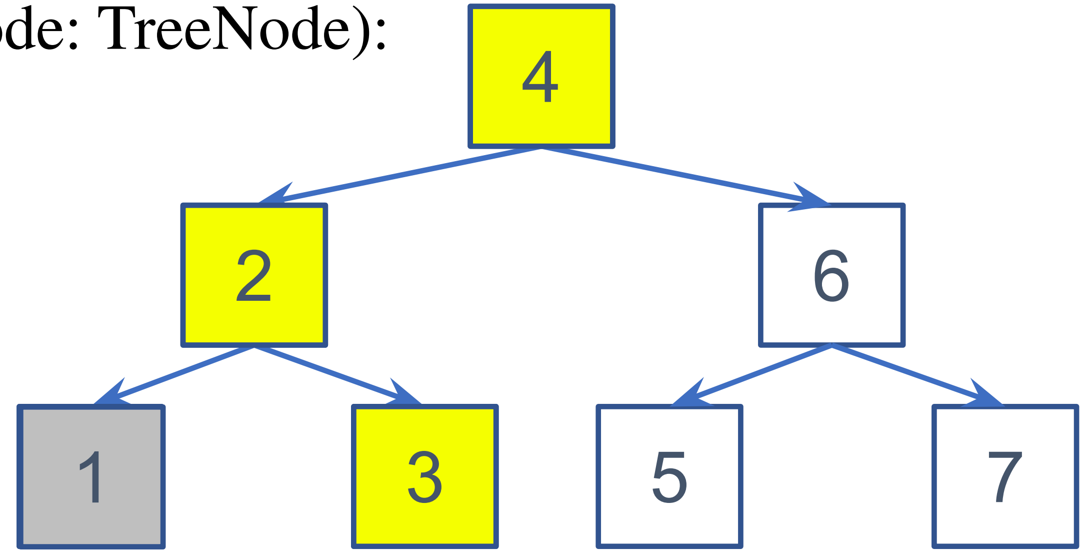


```
Out: 1
```

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
      if curNode == None:
          return
      for i in range(len(curNode.child)):
          self.__DFT_postorderHelp(curNode.child[i])
      self.visit(curNode)

    def DFT_postorder(self):
      self.__DFT_postorderHelp(self.root)
```

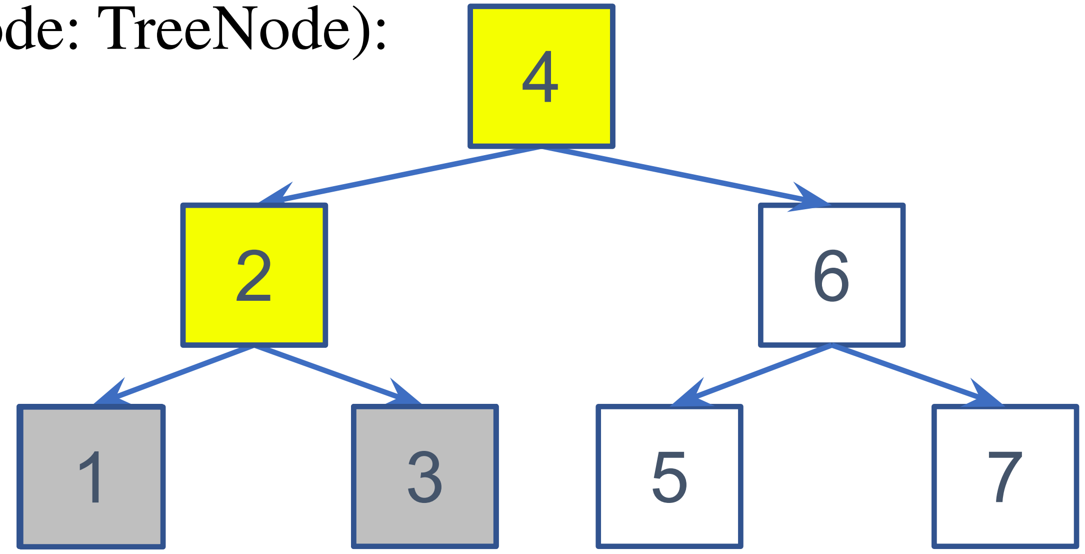


Out: 1 3

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

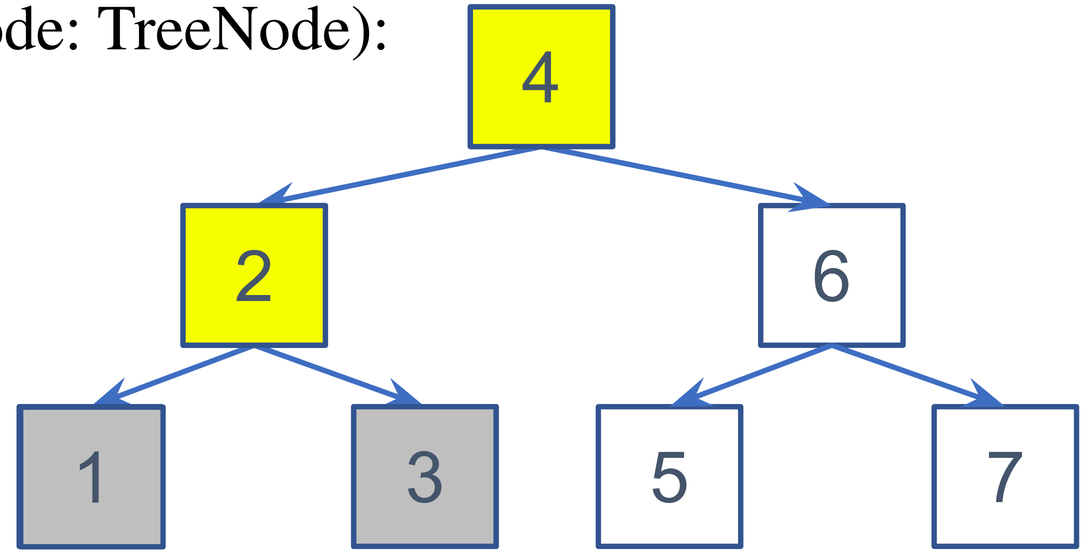


Out: 1 3

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

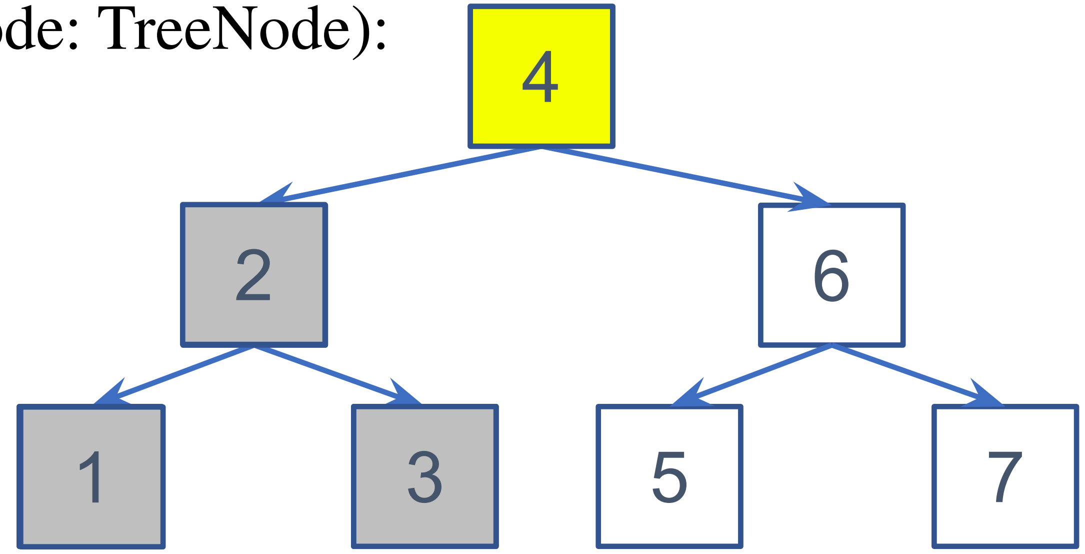


Out: 1 3 2

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

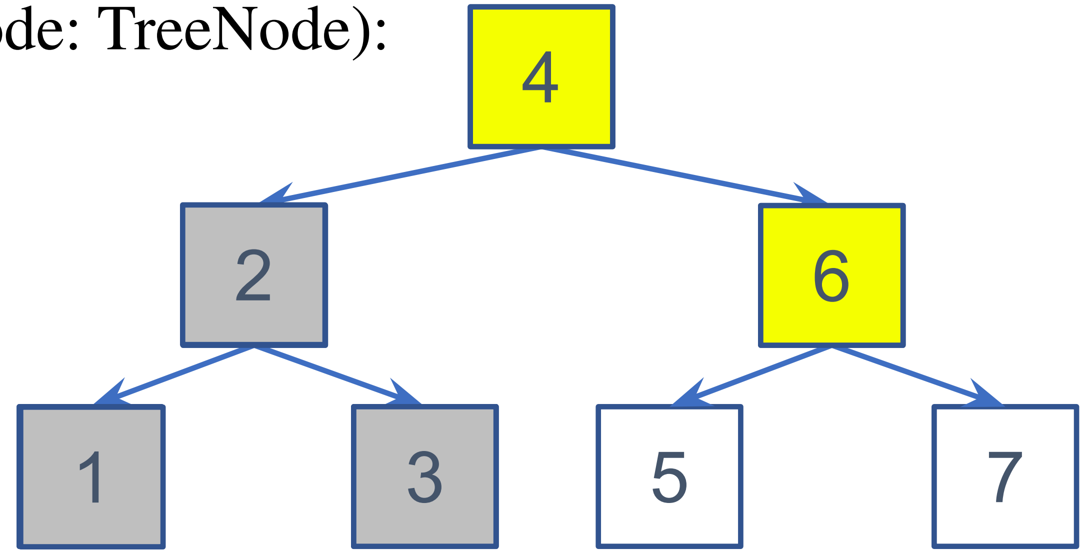


```
Out: 1 3 2
```

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

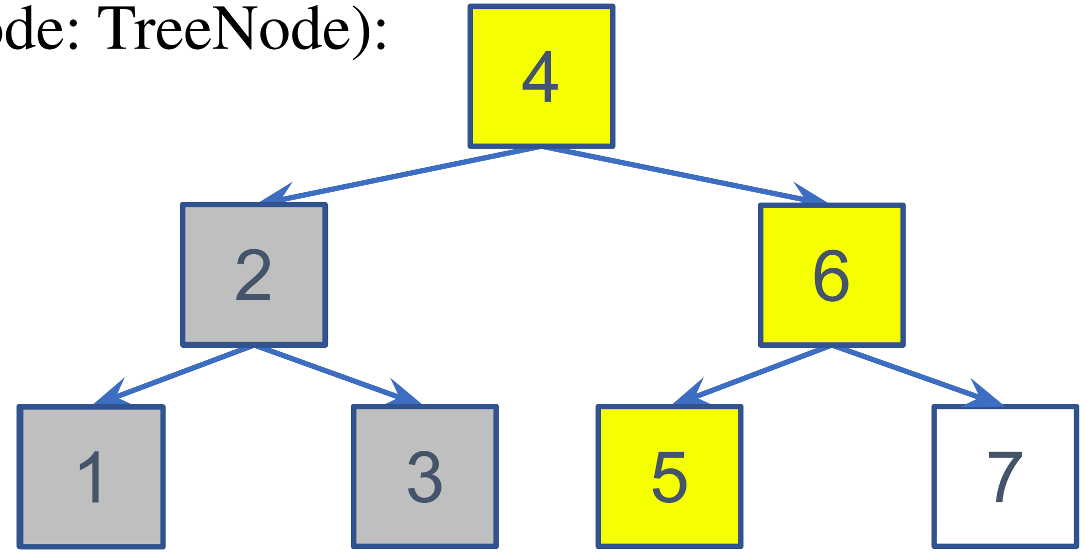


Out: 1 3 2

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

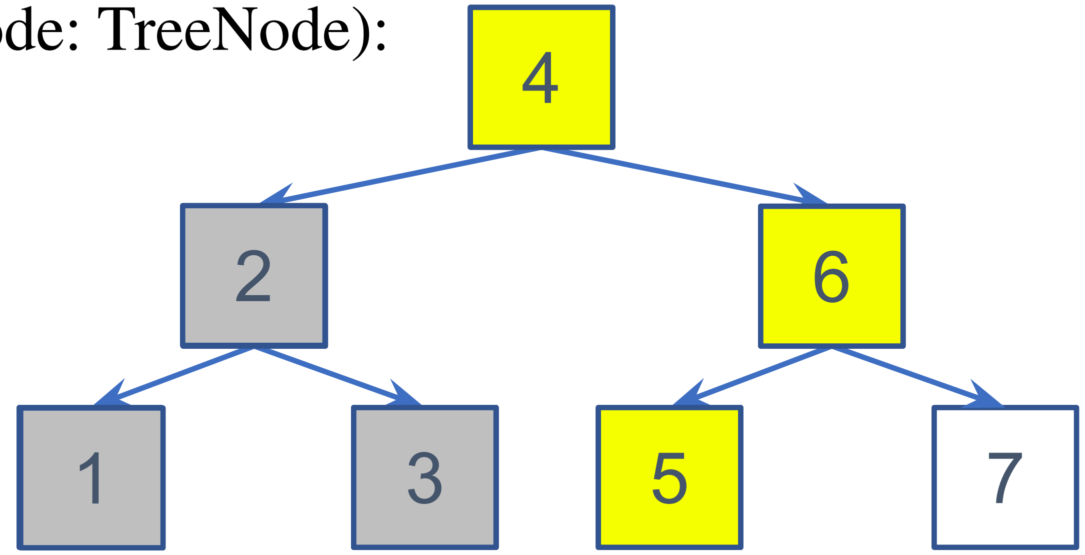


Out: 1 3 2

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

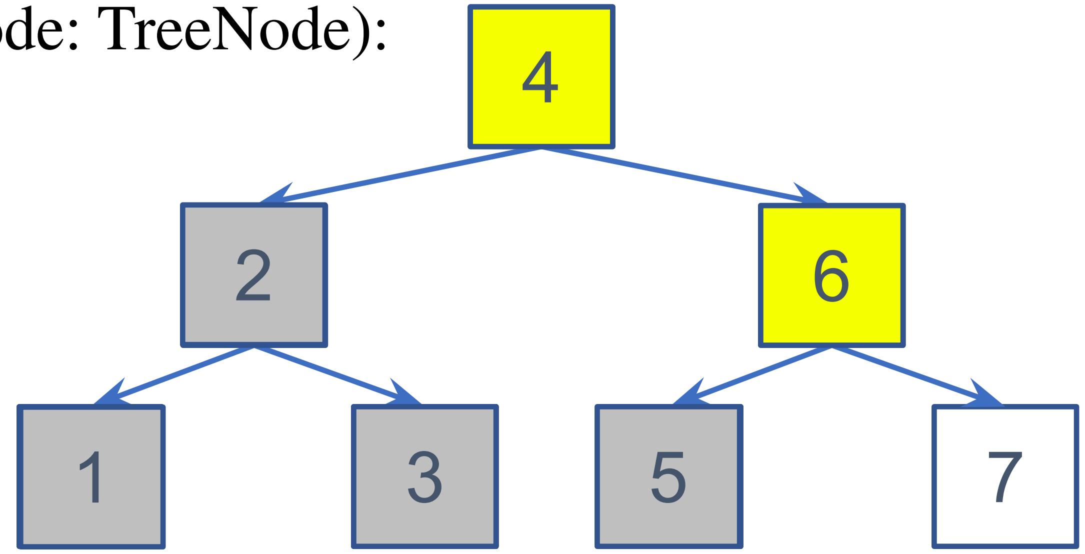


Out: 1 3 2 5

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

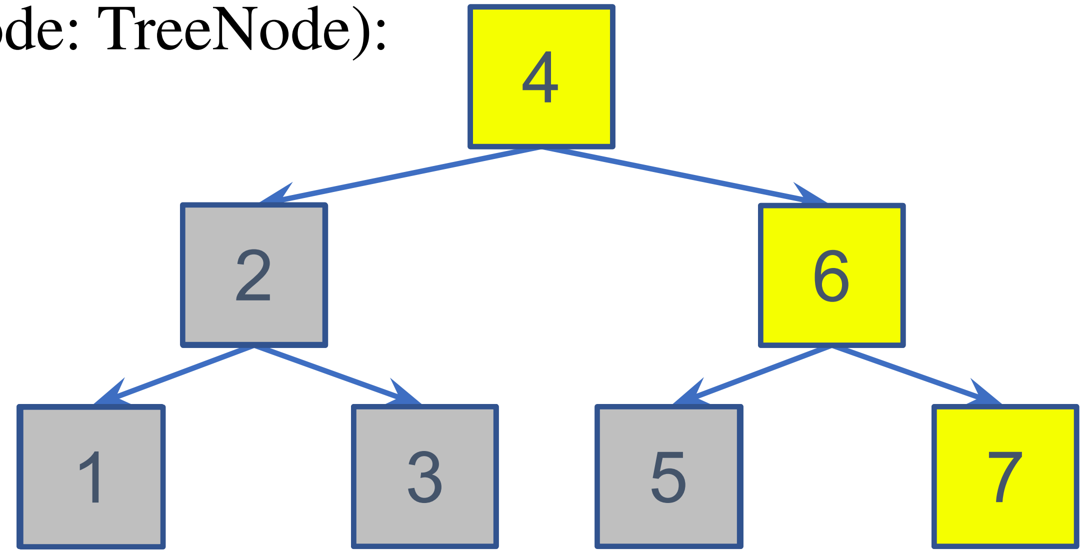


Out: 1 3 2 5

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

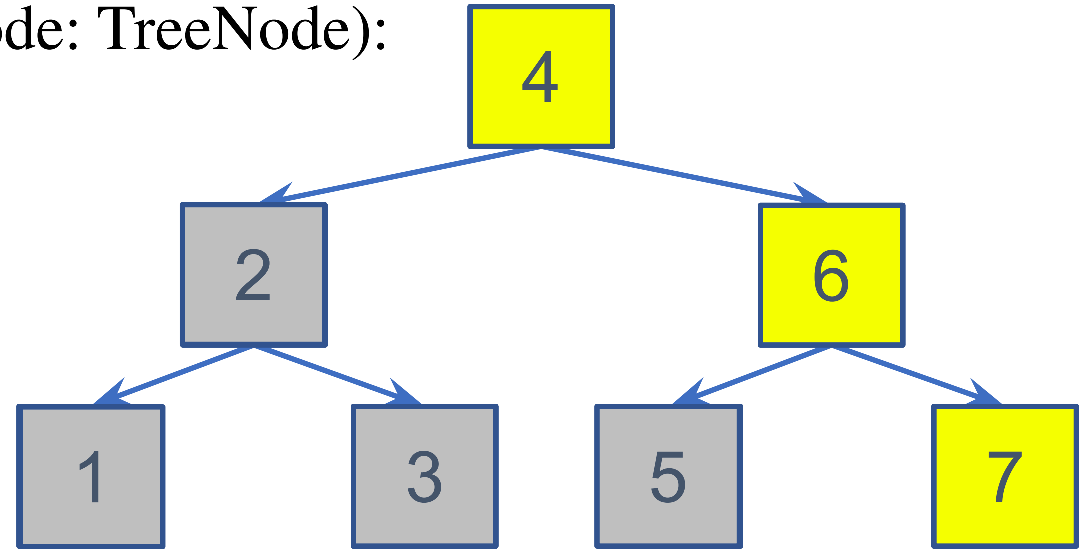


Out: 1 3 2 5

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

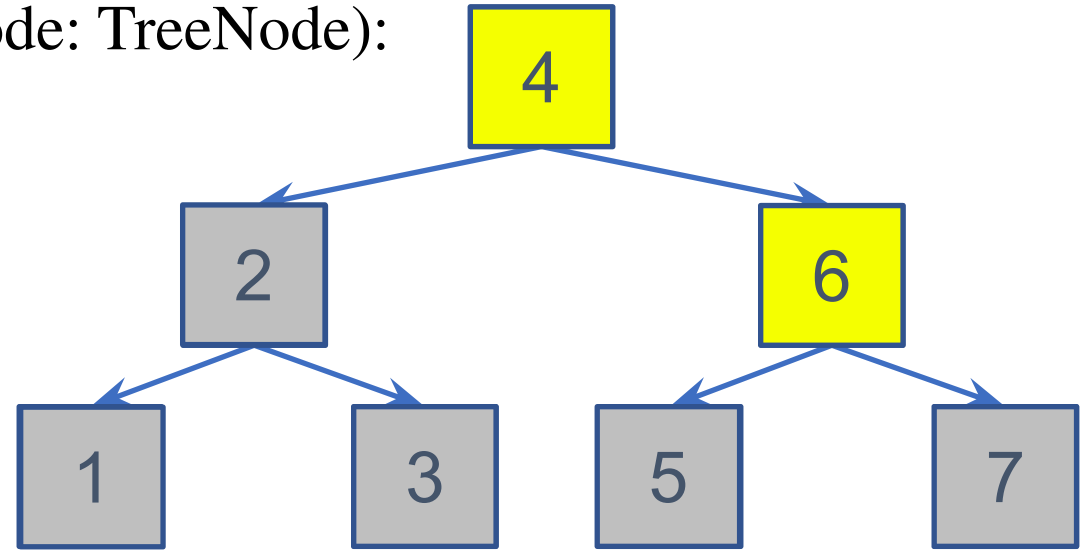


Out: 1 3 2 5 7

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

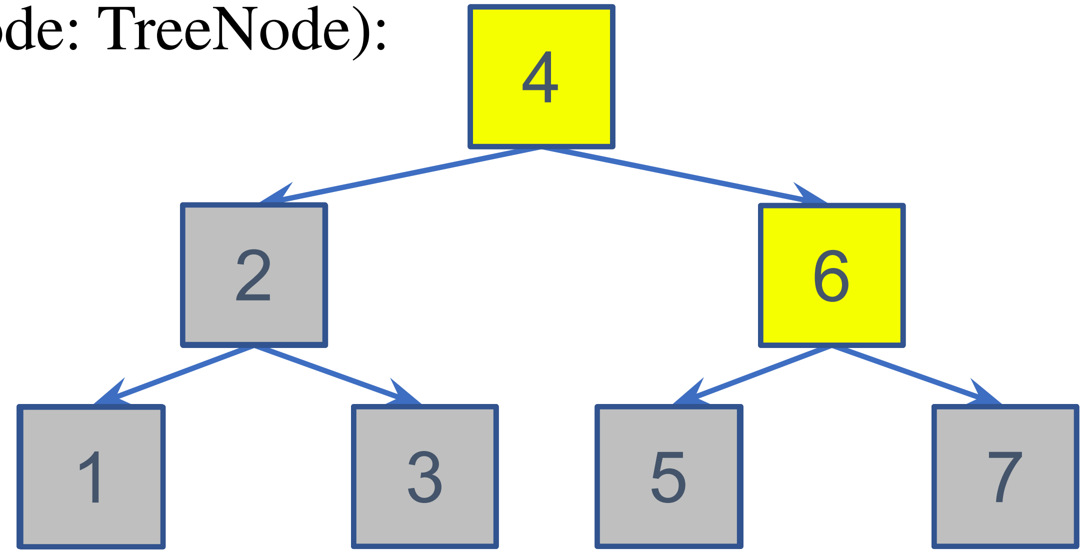


Out: 1 3 2 5 7

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

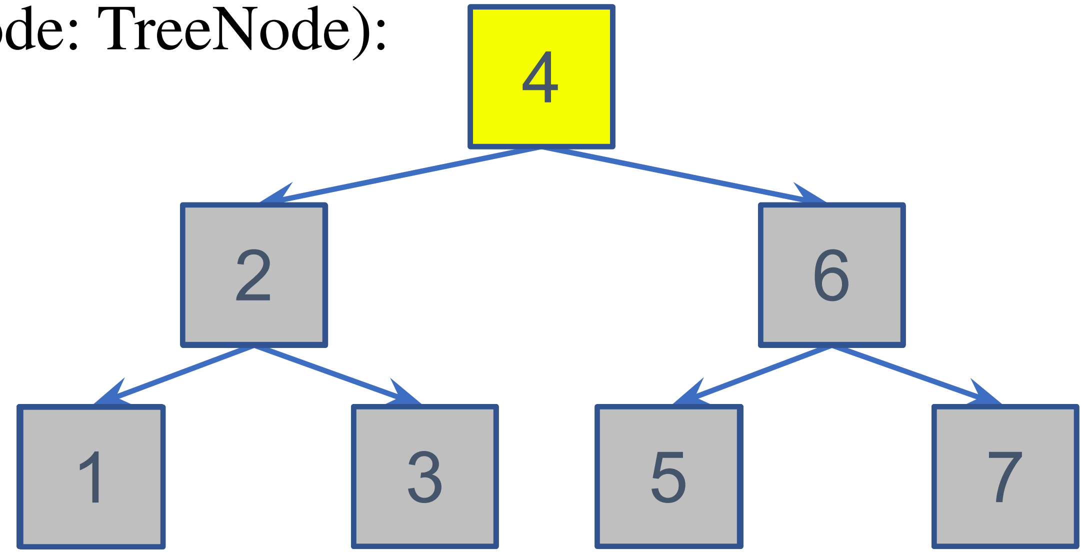


```
Out: 1 3 2 5 7 6
```

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

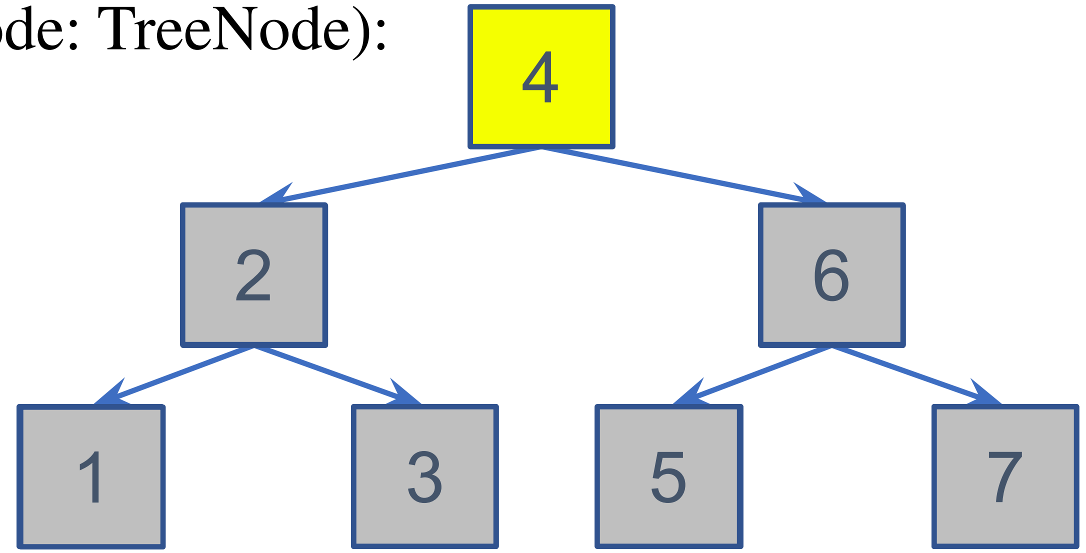


Out: 1 3 2 5 7 6

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

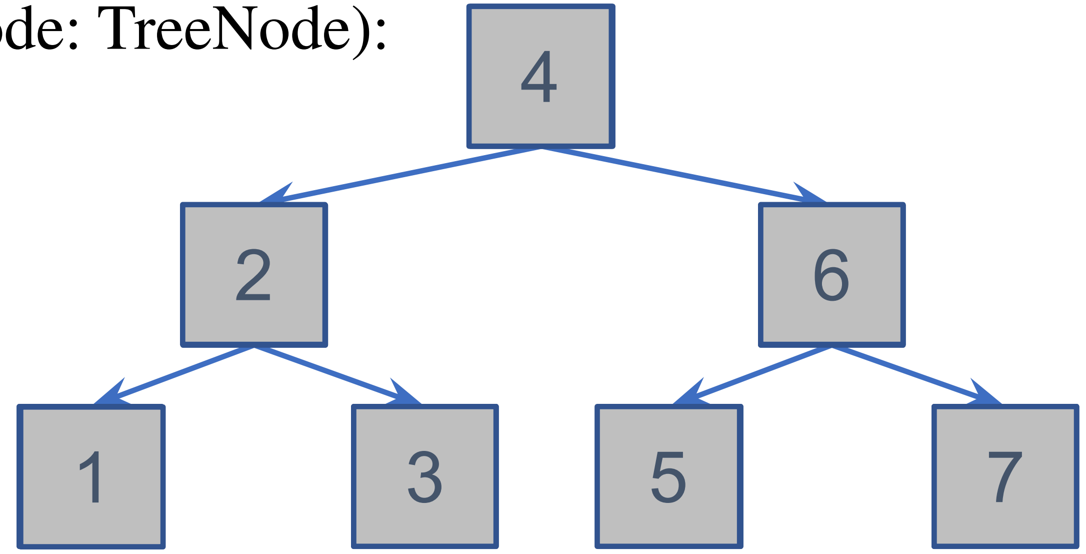


Out: 1 3 2 5 7 6 4

Depth First Traversals – Postorder

- Visit a node **after** traversing its children from left to right

```
class Tree():
    def visit(self, node: TreeNode):
        print(node.val)

    def __DFT_postorderHelp(self, curNode: TreeNode):
        if curNode == None:
            return
        for i in range(len(curNode.child)):
            self.__DFT_postorderHelp(curNode.child[i])
        self.visit(curNode)

    def DFT_postorder(self):
        self.__DFT_postorderHelp(self.root)
```

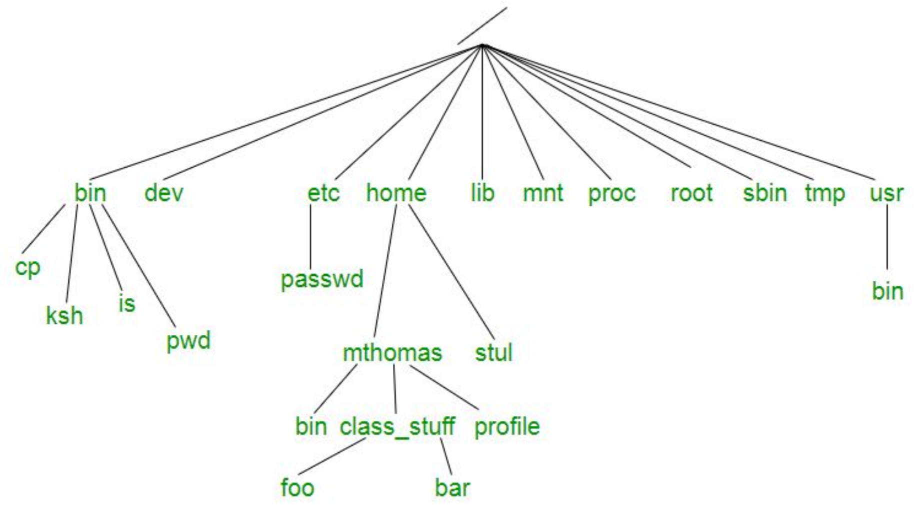


Out: 1 3 2 5 7 6 4

Depth First Traversals – Postorder

- **Application**: File size calculation

```
class Tree():
    def visit(self, node: TreeNode, x: float) -> float:
        return node.val

    def __DFT_postorderHelp(curNode: TreeNode) -> float:
        ans = 0
        if curNode:
            for i in range(len(curNode.child)):
                ans += self.__DFT_postorderHelp(curNode.child[i])
            ans += self.visit(curNode)
        return ans

    def DFT_postorder(self) -> float:
        return self.__DFT_postorderHelp(self.root)
```



Summary

- Breadth-first traversal
  - Implementation using FIFO queue (deque in Python)

- Depth-first traversal
  - Implementation using recursion (or LIFO stack – also using deque in Python)
  - Three types for different purposes
    - Preorder
    - Inorder
    - Postorder