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)

Binary Search Tree

Lecture 16

Hyung-Sin Kim

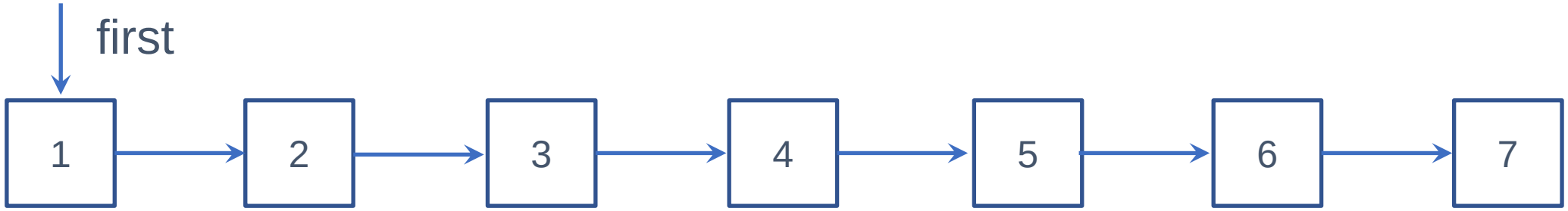
SNU Graduate School of Data Science

Contents

- Tree
- Binary Search Tree
  - Search
  - Insert
  - Delete

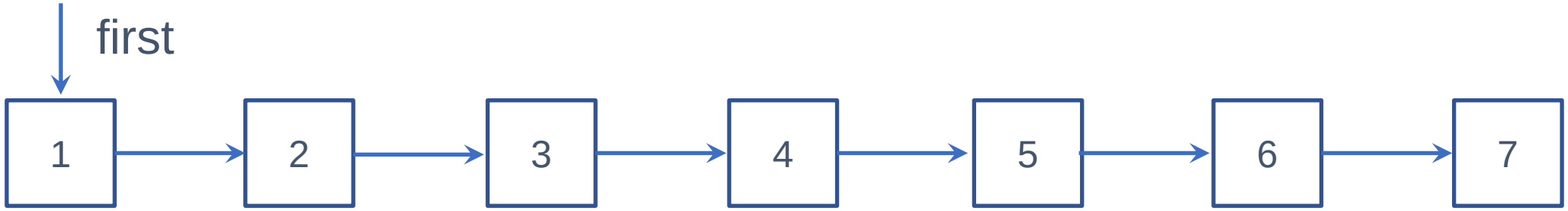
Downside of Linked Lists

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



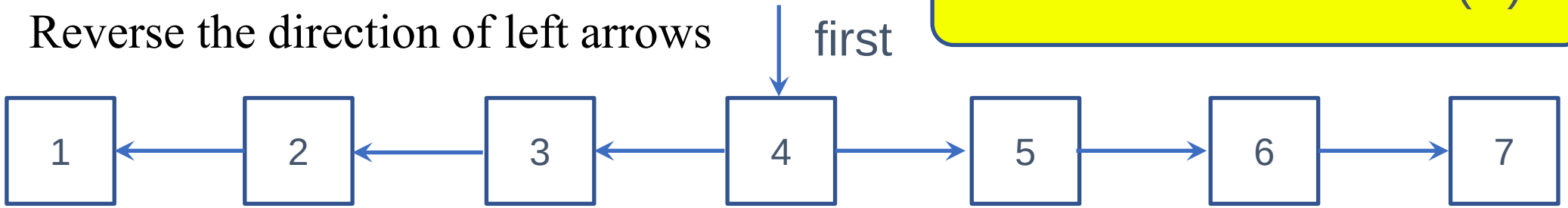
Downside of Linked Lists

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



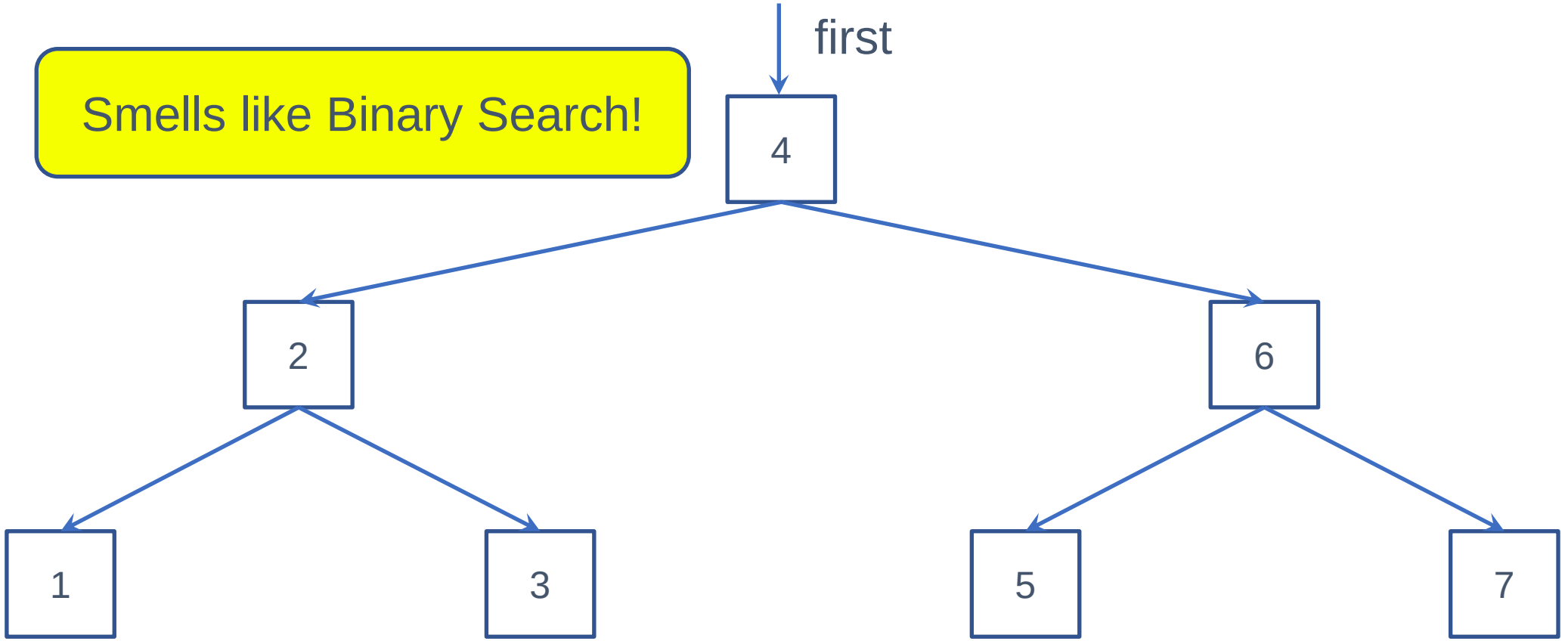
- An improvement for search
  - Change the first node to middle
  - Reverse the direction of left arrows

Twice faster but still O(N)!



Improving Linked Lists

- How about this?



Tree

Computing Foundations for Data Science

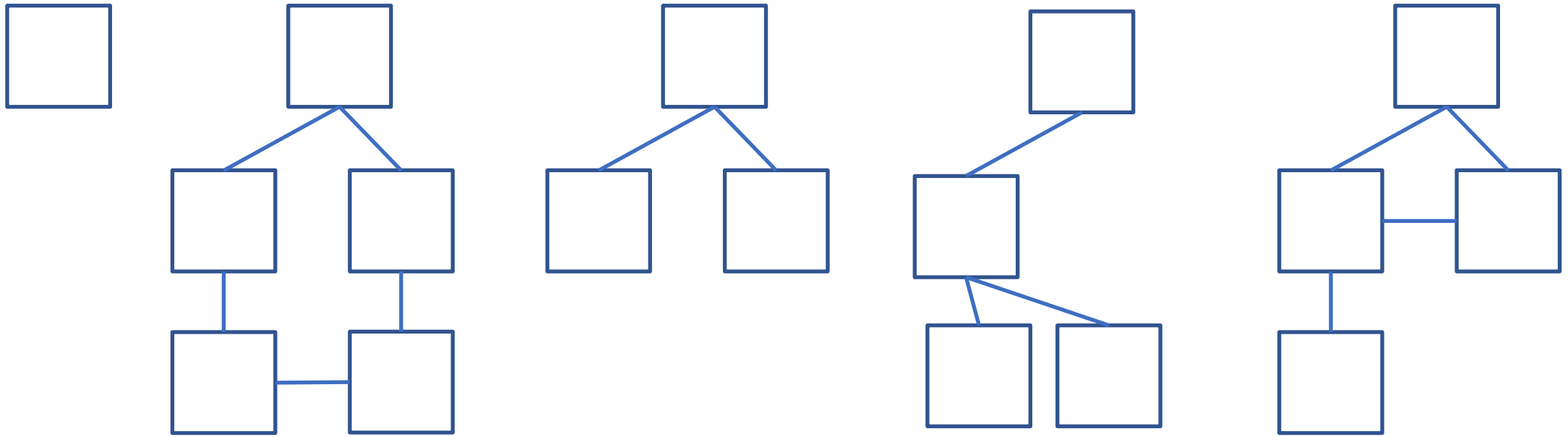
Trees

- 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!

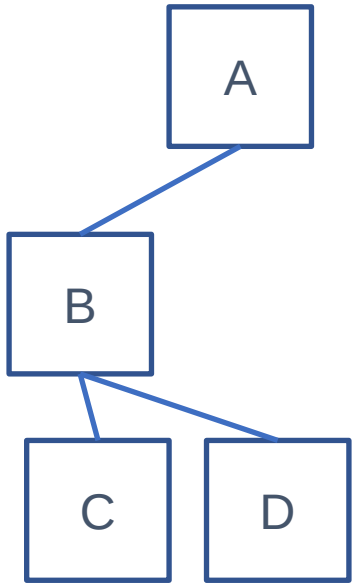


Rooted Binary Trees

- **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**

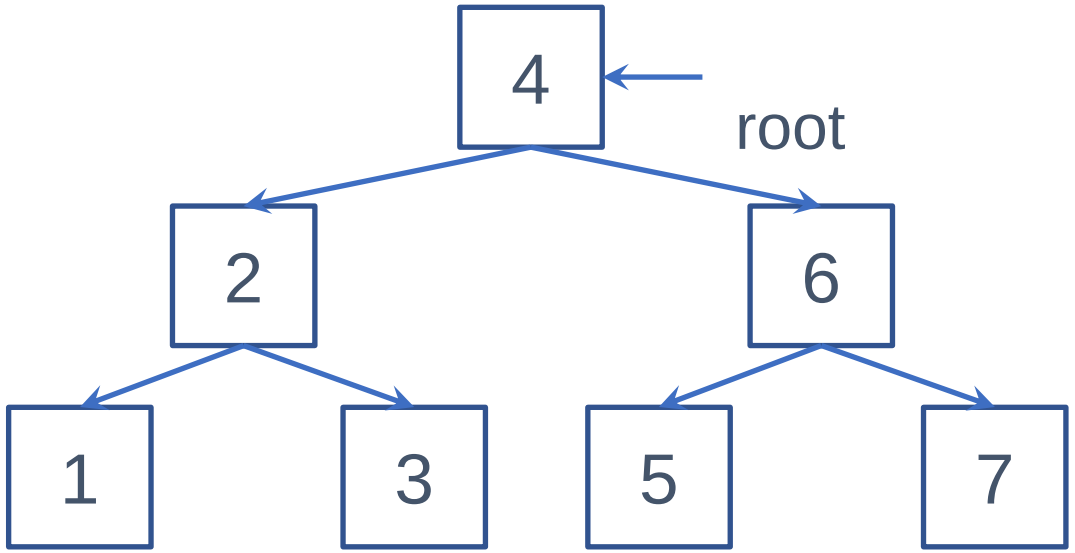
- Relationship
  - A is the root and a parent of B
  - B is a child of A and a parent of C and D
  - C and D are leaves and children of B

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



Binary Search Trees

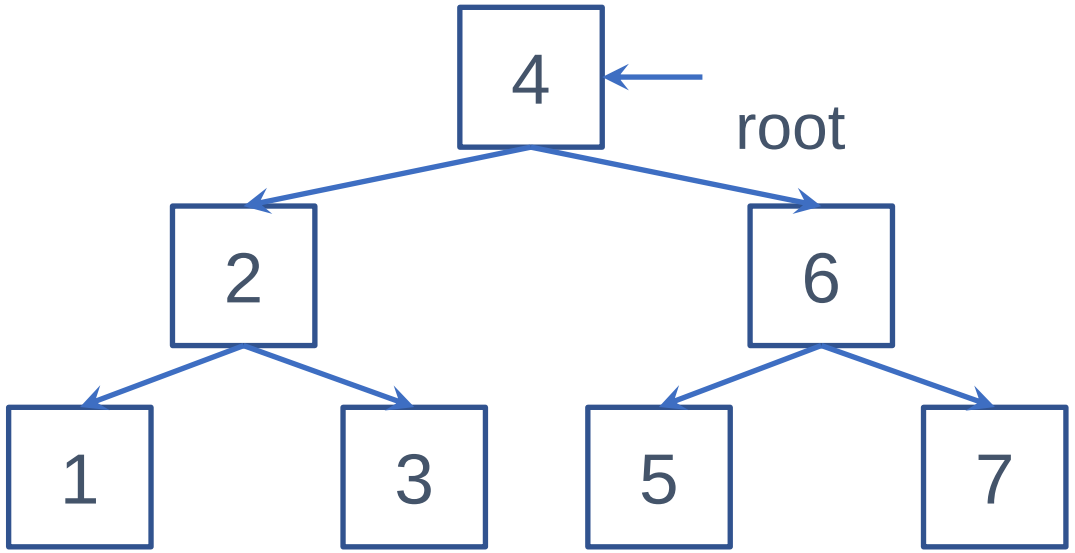
- A binary search tree is a rooted binary tree that has the following two properties
- For every node **x**,
  - **x**'s value is unique in the whole tree
  - Every node **y** in the left subtree of node **x** has value less than **x**'s value
  - Every node **z** in the right subtree of node **x** has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



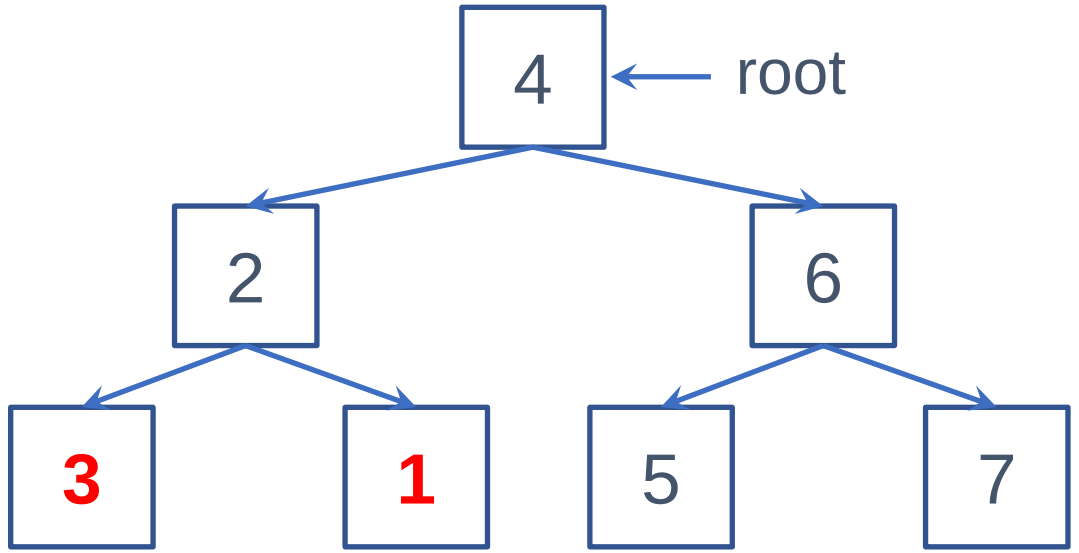
a Binary Search Tree

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 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

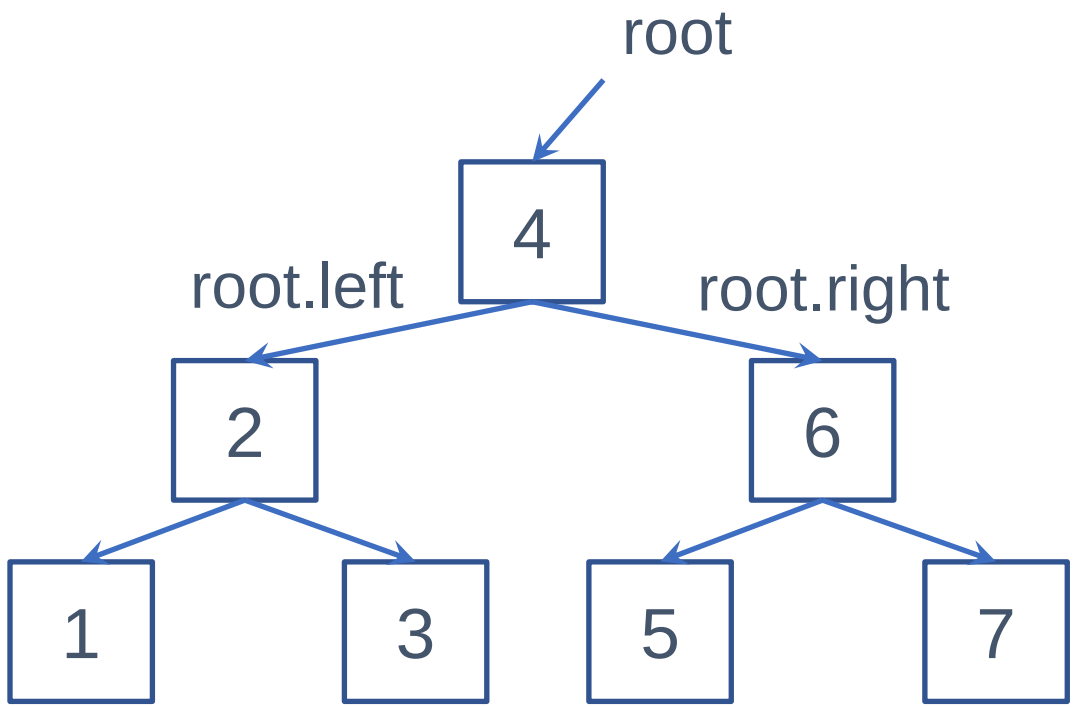


a Binary (not Search) Tree

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):

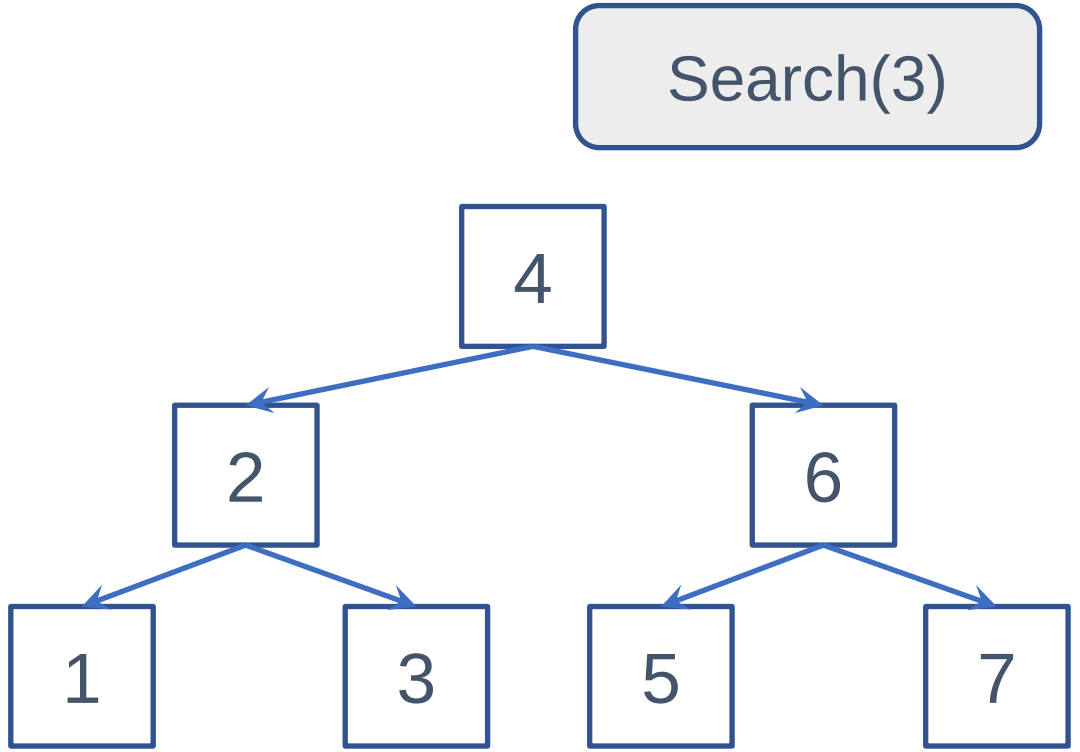


Binary Search Tree

- **Search**
- **Insert**
- **Delete**

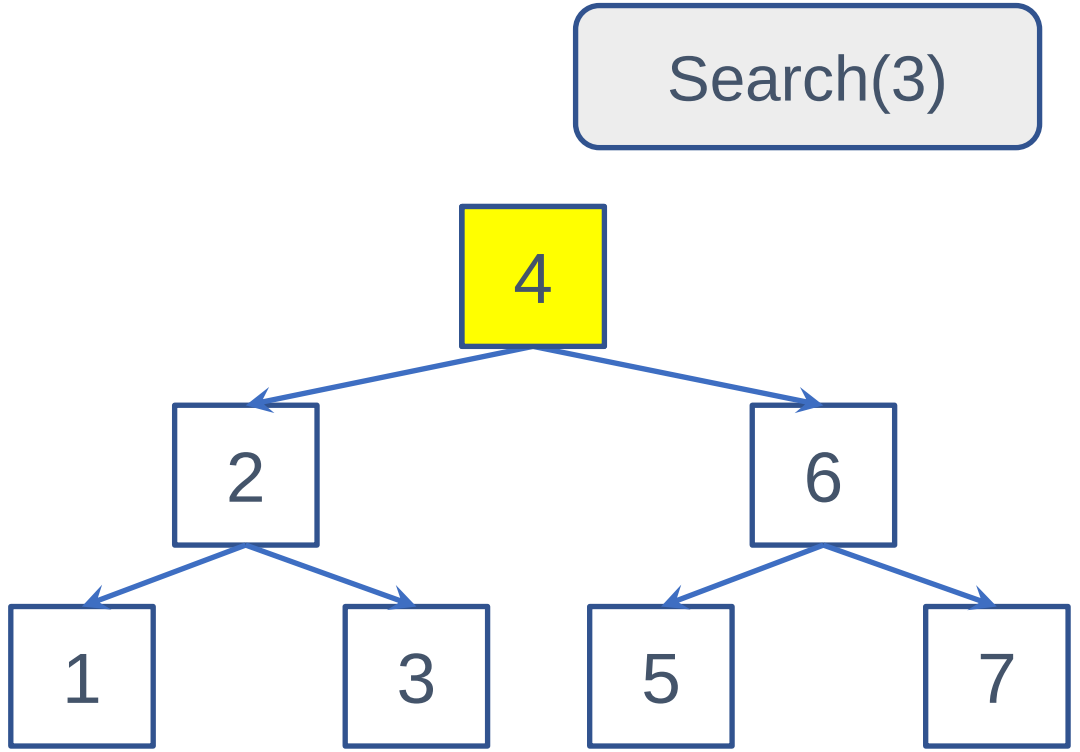
Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```



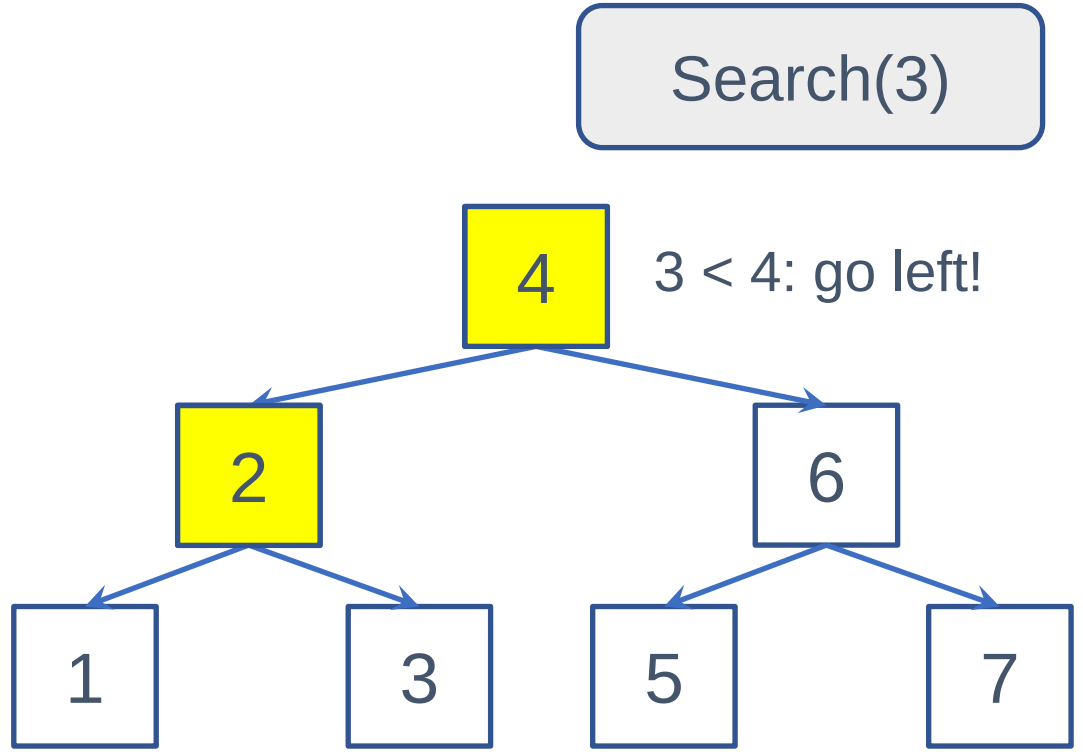
Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```



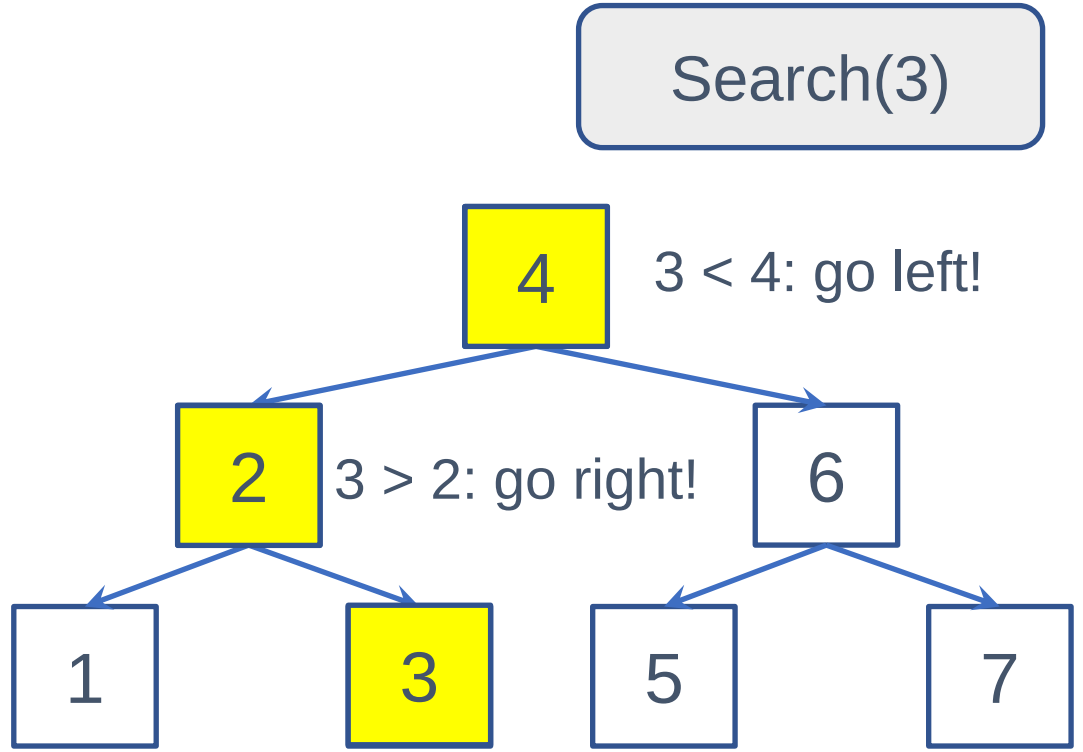
Binary Search Trees – Search

- class BST():
- def **__searchHelp**(self, curNode: TreeNode, x: int) -> 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)
- def **search**(self, x: int) -> TreeNode:
- return self.__searchHelp(self.root, x)



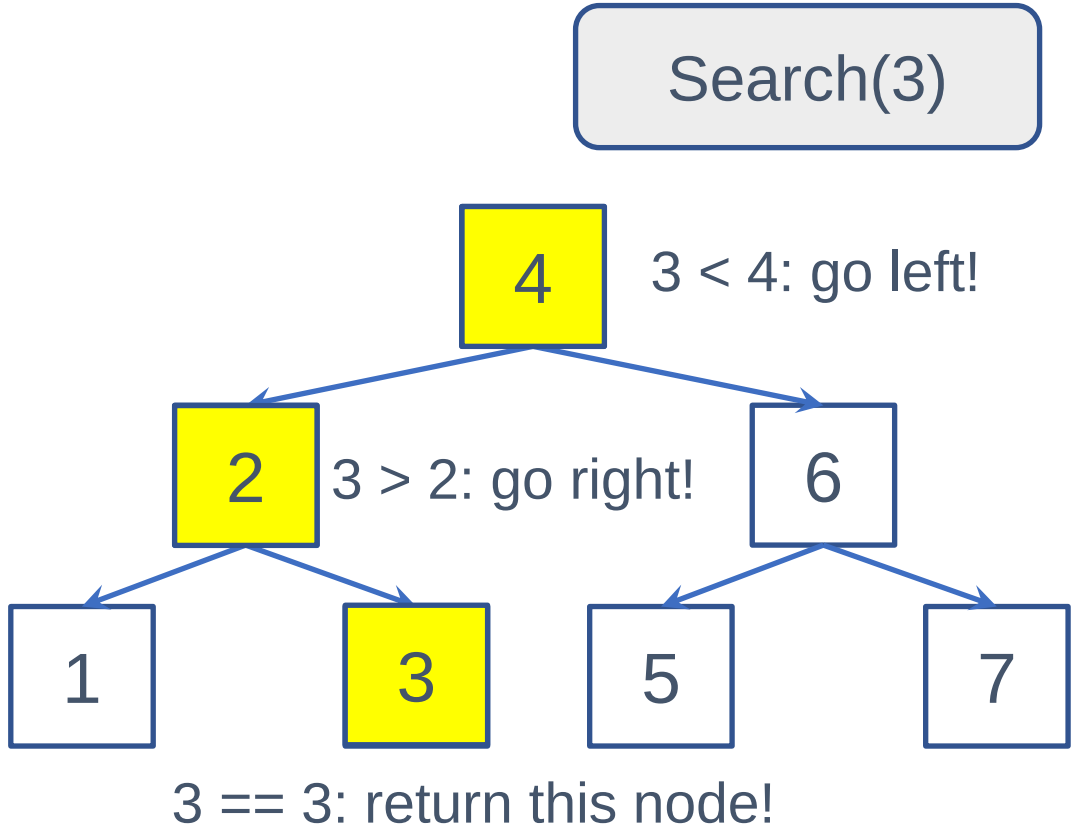
Binary Search Trees – Search

- class BST():
- def **__searchHelp**(self, curNode: TreeNode, x: int) -> 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)**
- def **search**(self, x: int) -> TreeNode:
- return self.__searchHelp(self.root, x)



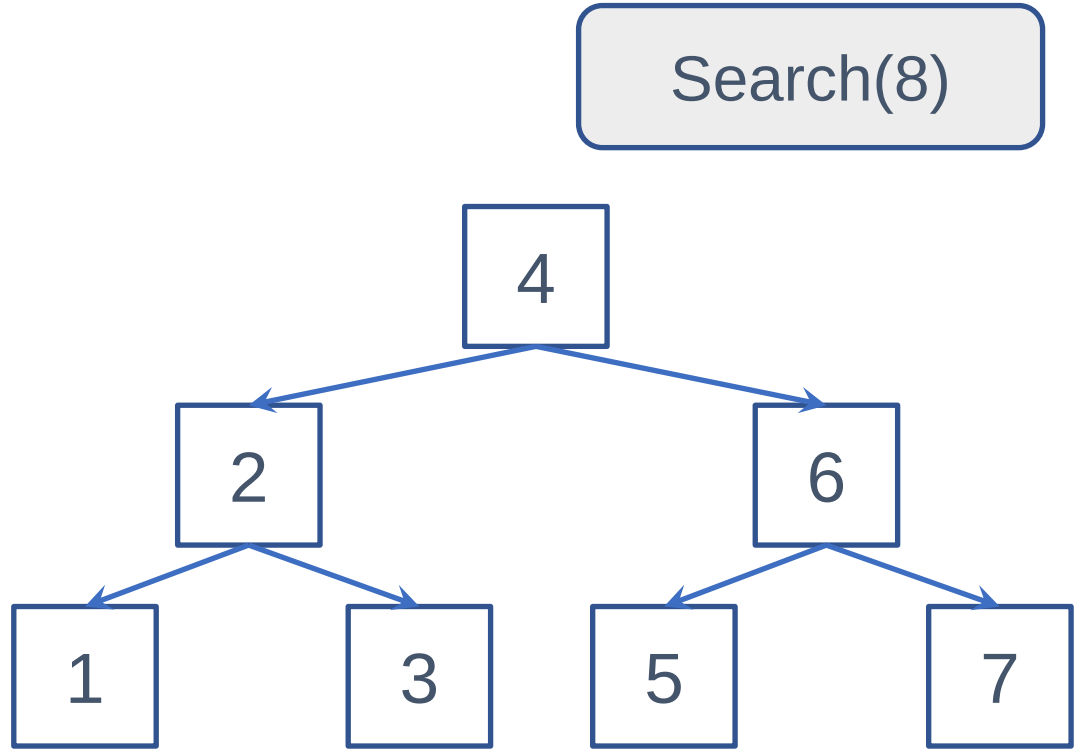
Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```



Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```



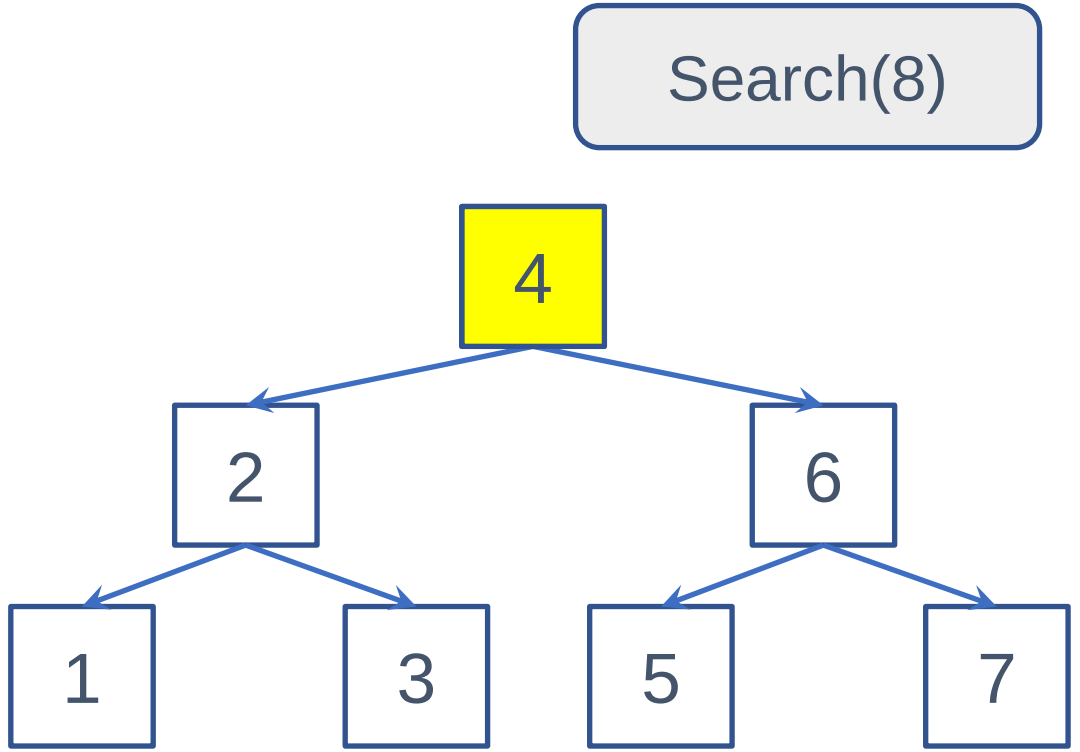
Binary Search Trees – Search

- class BST():
- def **__searchHelp**(self, curNode: TreeNode, x: int) -> 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)
```

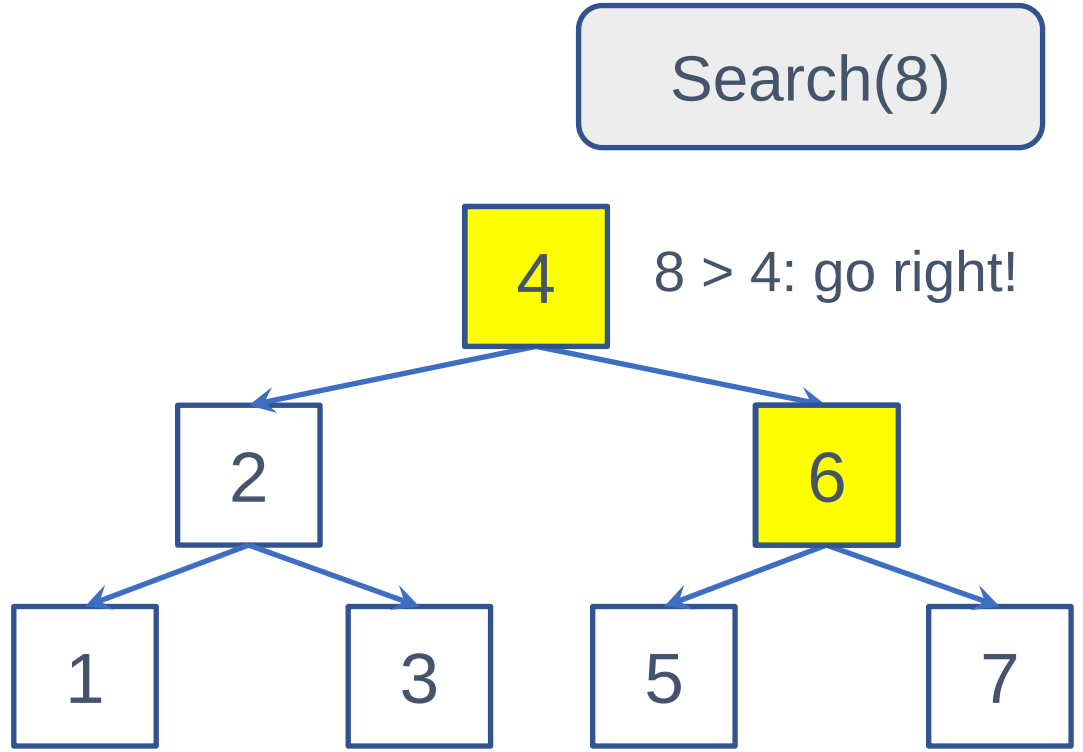
- def **search**(self, x: int) -> TreeNode:

```
return self.__searchHelp(self.root, x)
```



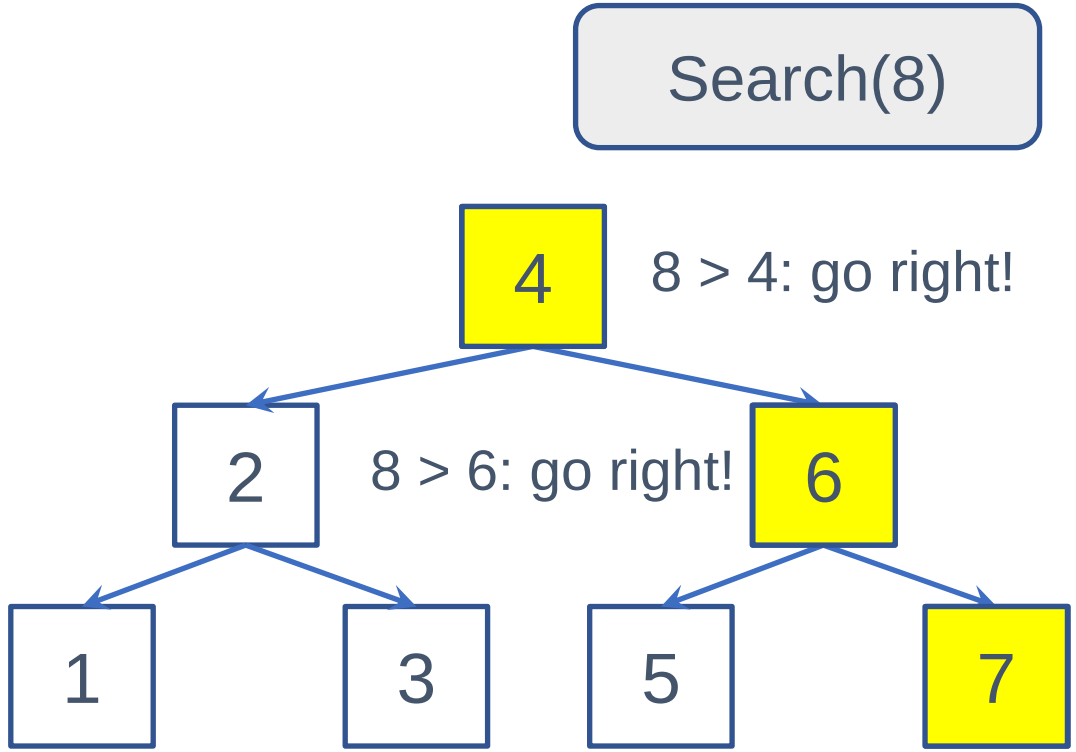
Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```



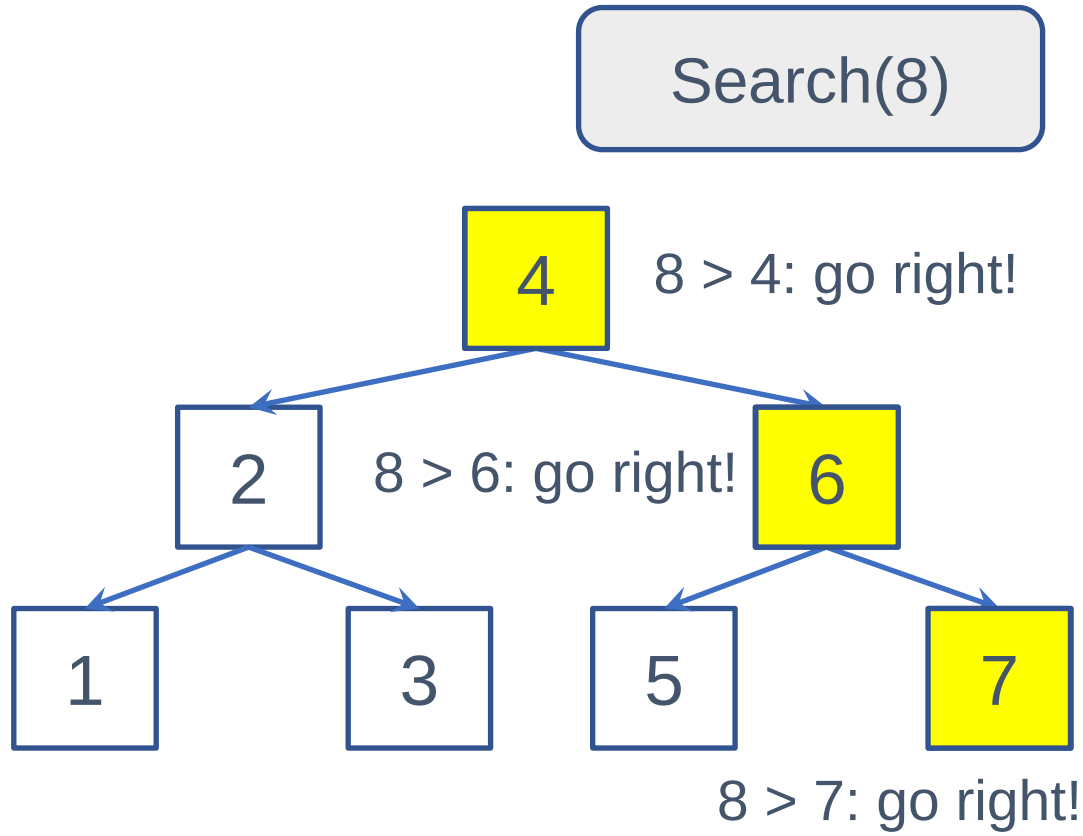
Binary Search Trees – Search

- class BST():
- def **__searchHelp**(self, curNode: TreeNode, x: int) -> 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)**
- def **search**(self, x: int) -> TreeNode:
- return self.__searchHelp(self.root, x)



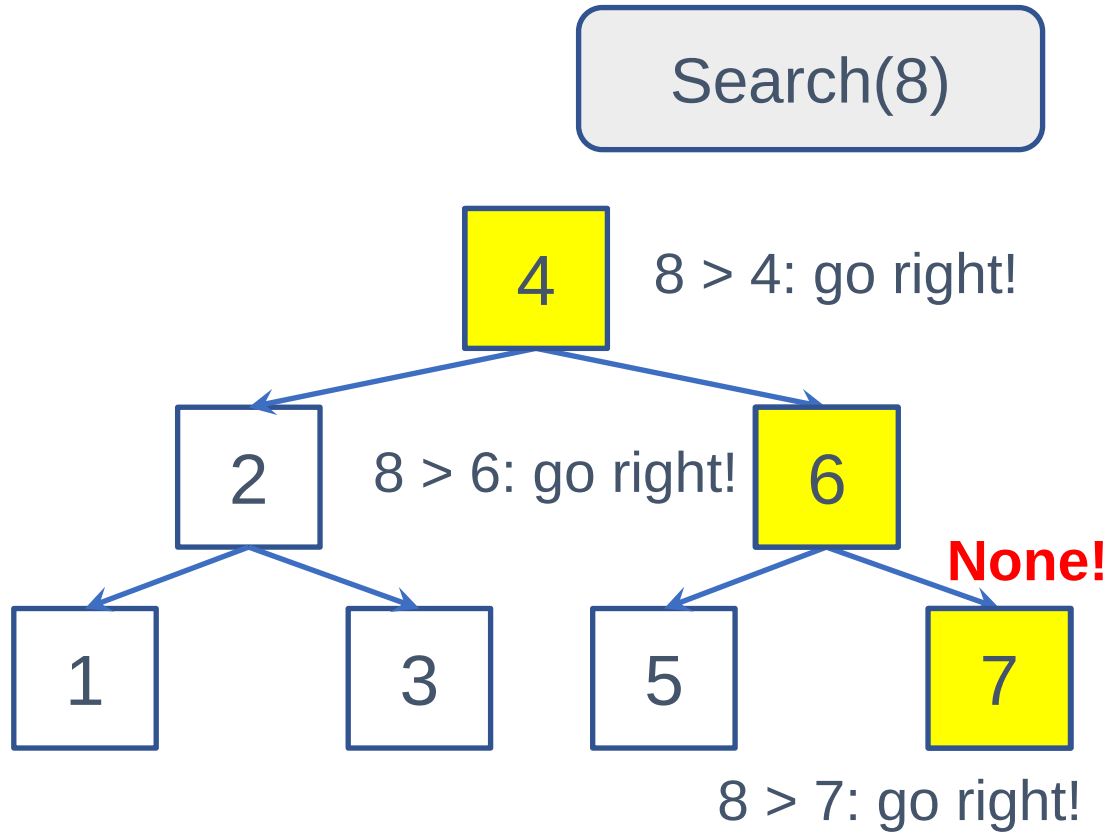
Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```



Binary Search Trees – Search

```
class BST():
    def __searchHelp(self, curNode: TreeNode, x: int) -> 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)

    def search(self, x: int) -> TreeNode:
        return self.__searchHelp(self.root, x)
```

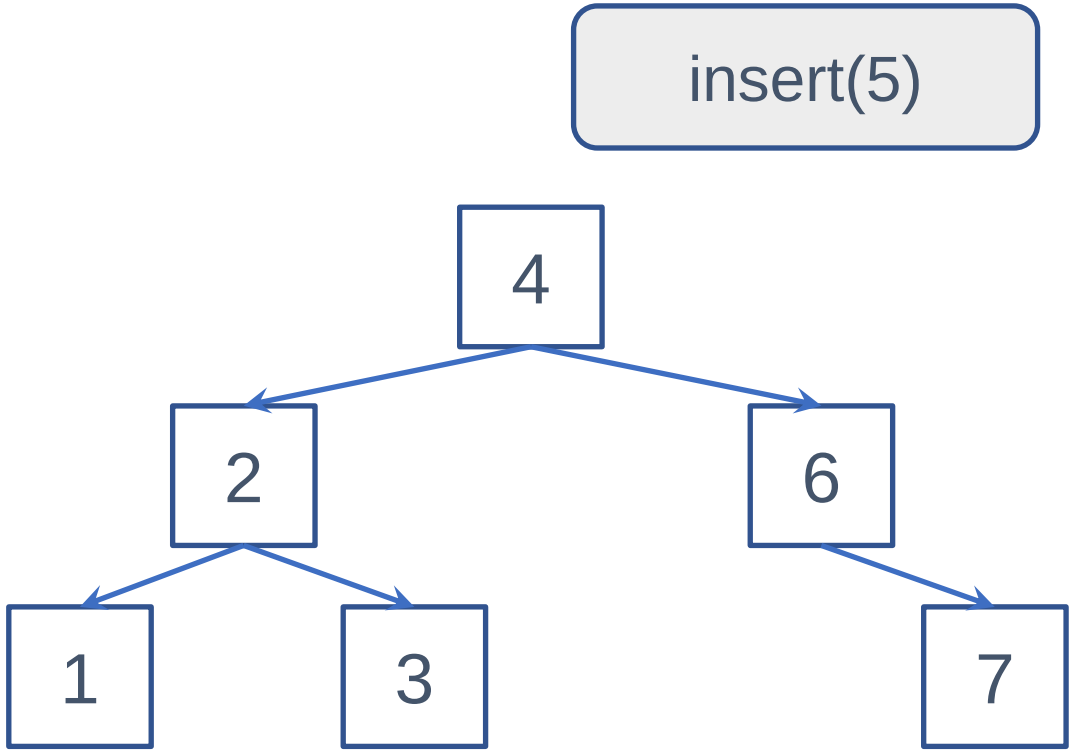


Binary Search Tree

- Search
- **Insert**
- Delete

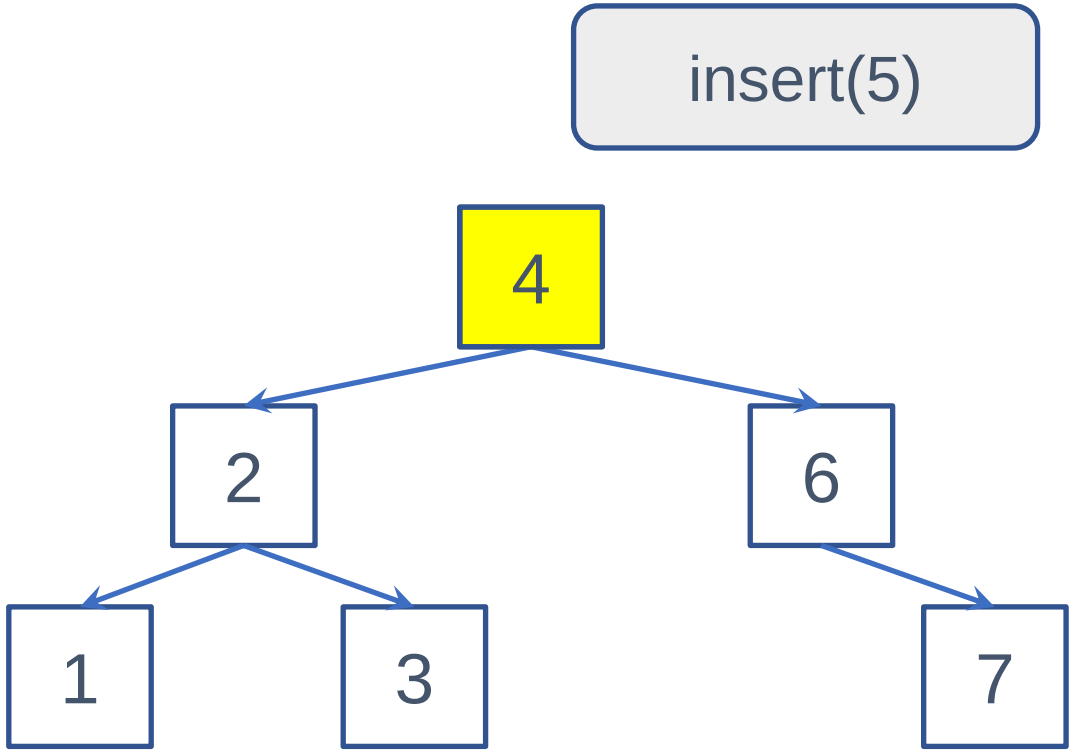
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(self, 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:
        self.root = self.__insertHelp(self.root, x)
```



Binary Search Trees – Insert

- class BST():
- def **__insertHelp**(self, 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:
- self.root = **self.__insertHelp(self.root, x)**



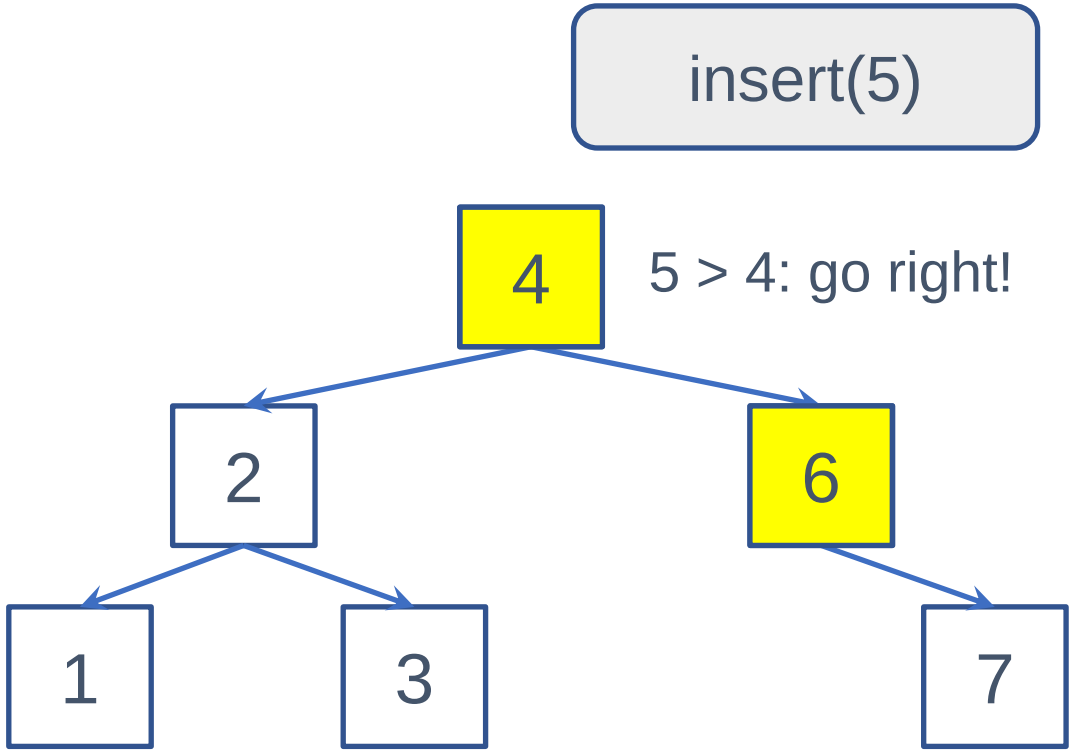
Binary Search Trees – Insert

- class BST():
- def **__insertHelp**(self, 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:

```
self.root = self.__insertHelp(self.root, x)
```



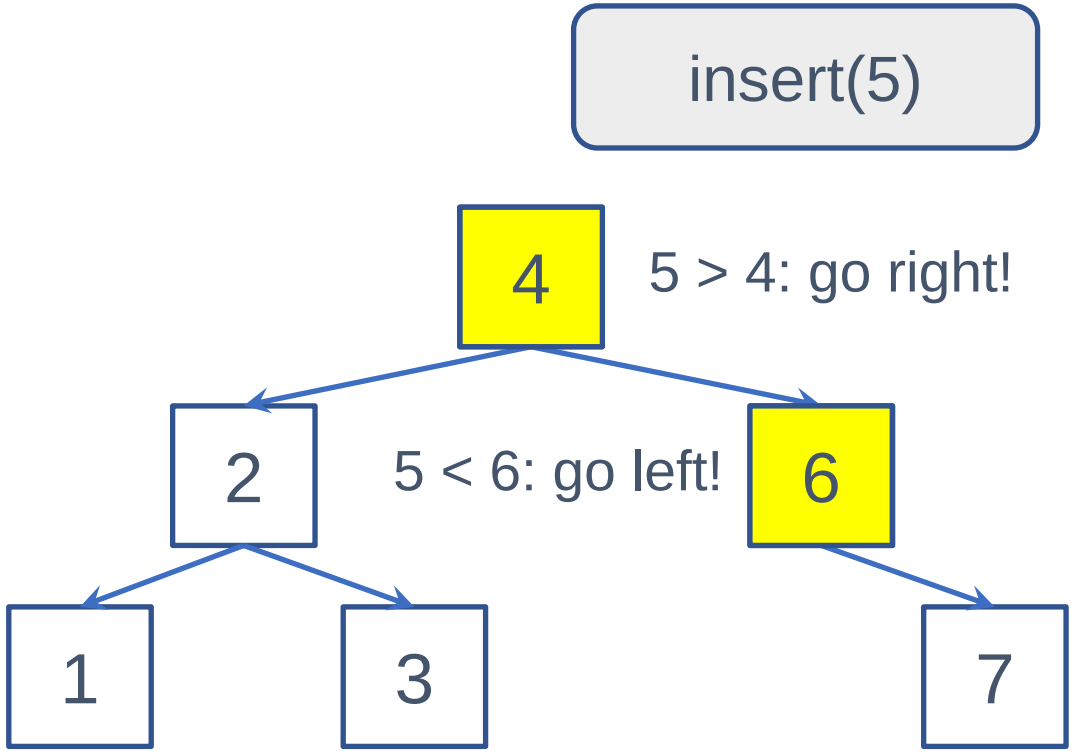
Binary Search Trees – Insert

- class BST():
  - def **__insertHelp**(self, 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:

```
self.root = self.__insertHelp(self.root, x)
```



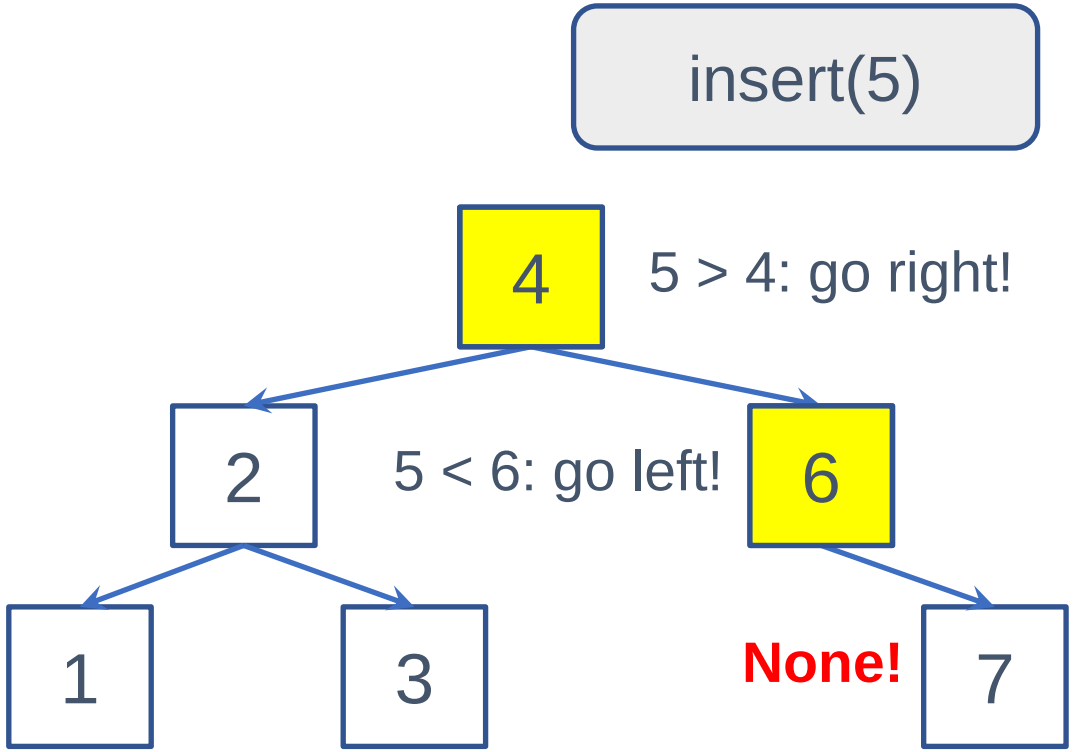
Binary Search Trees – Insert

- class BST():
- def **__insertHelp**(self, 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
```

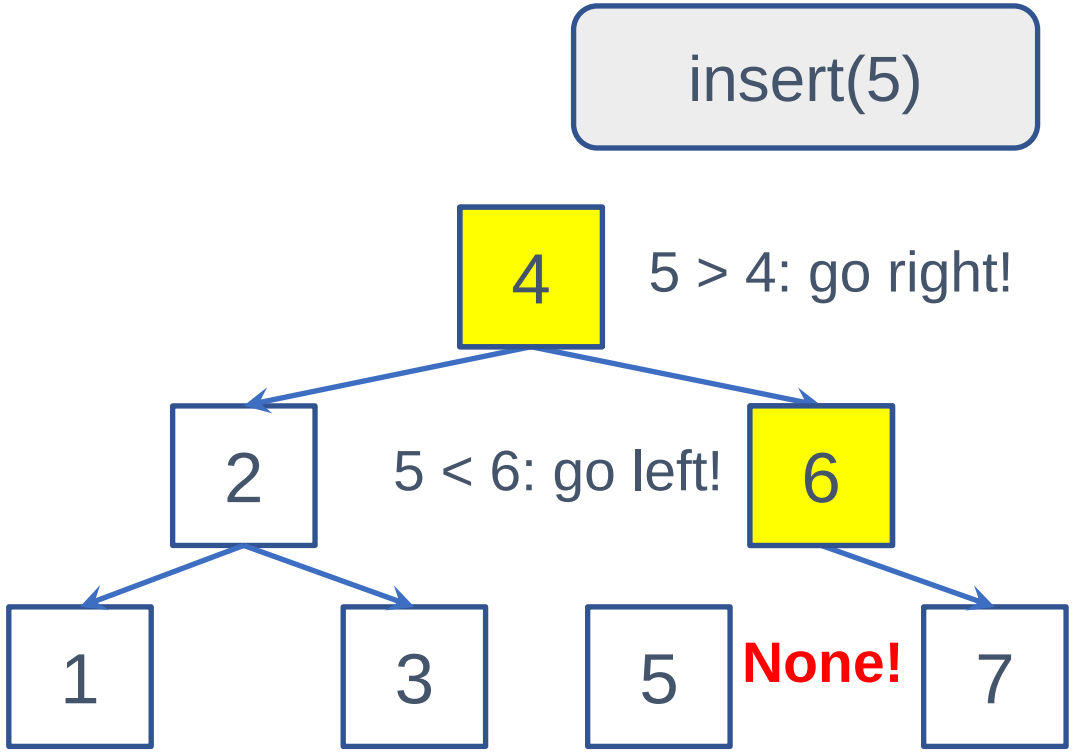
- def **insert**(self, x: int) -> None:

```
        self.root = self.__insertHelp(self.root, x)
```



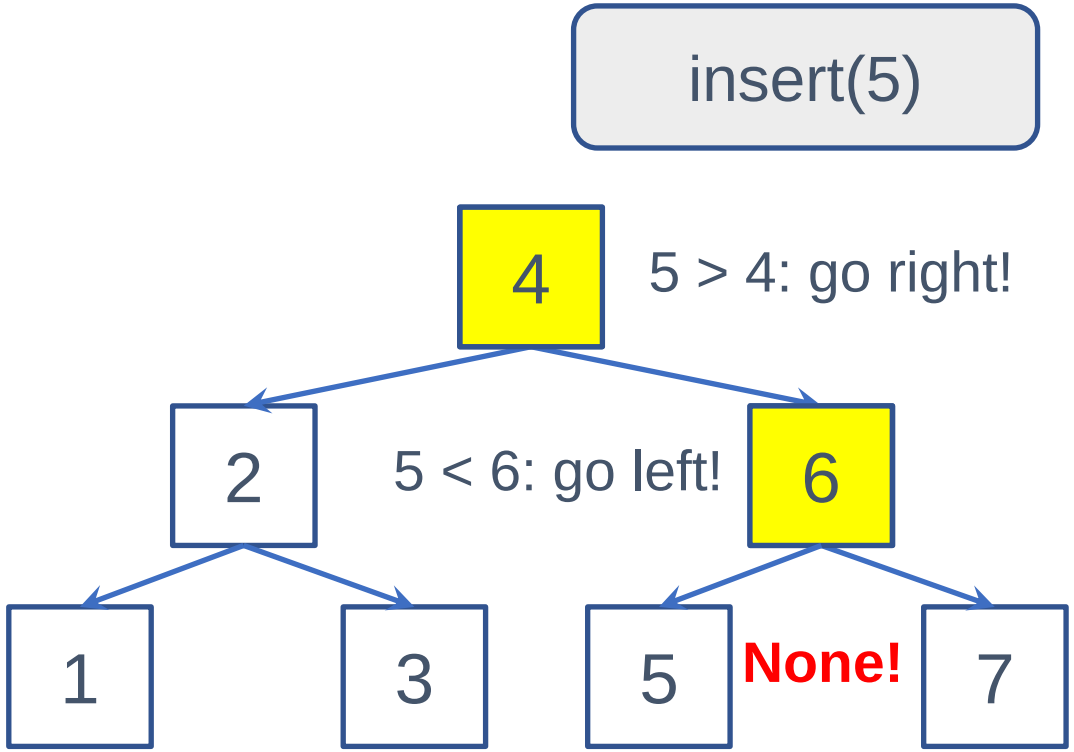
Binary Search Trees – Insert

```
class BST():
    def __insertHelp(self, 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:
        self.root = self.__insertHelp(self.root, x)
```



Binary Search Trees – Insert

- class BST():
- def **__insertHelp**(self, 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:
- self.root = self.__insertHelp(self.root, x)

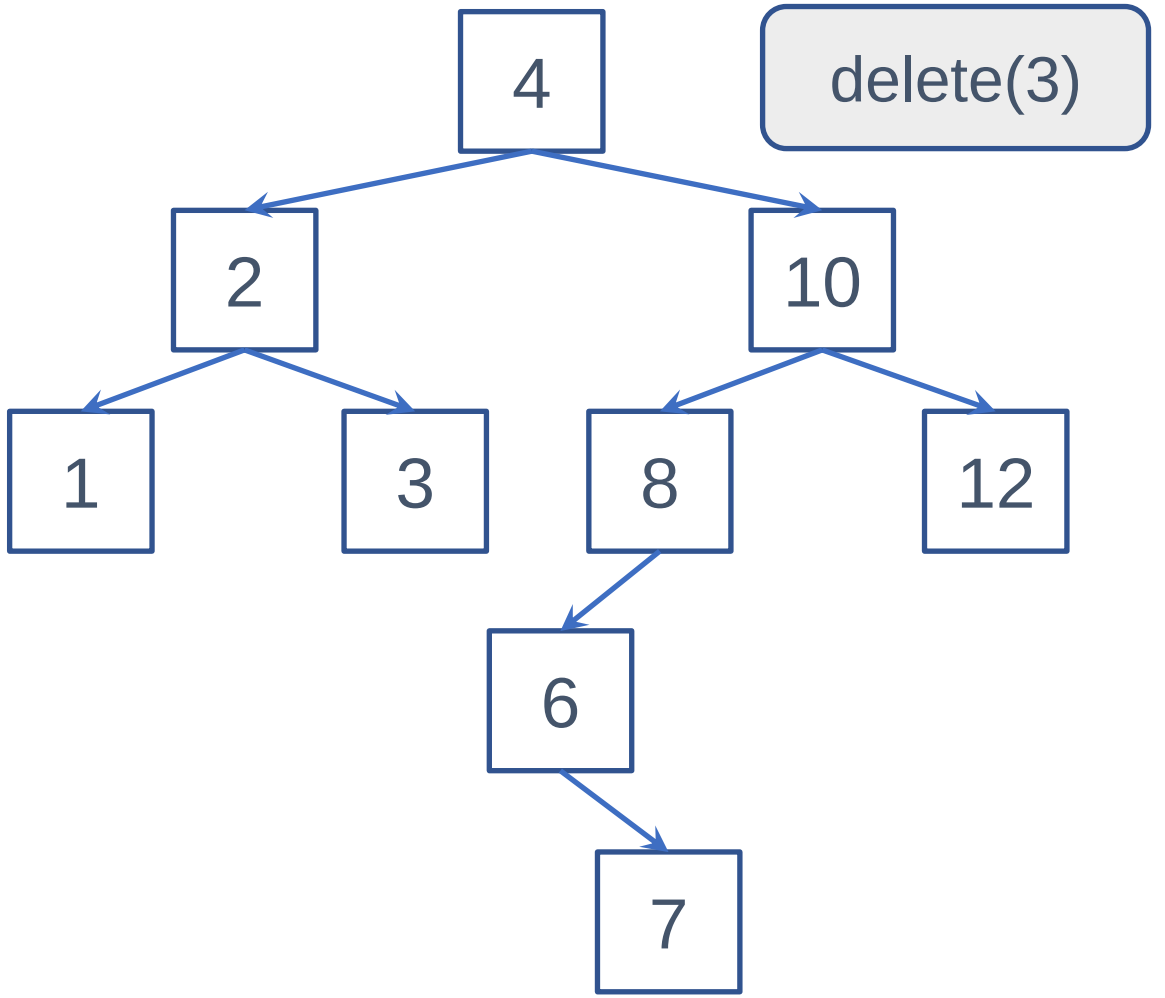


Binary Search Tree

- Search
- Insert
- **Delete**

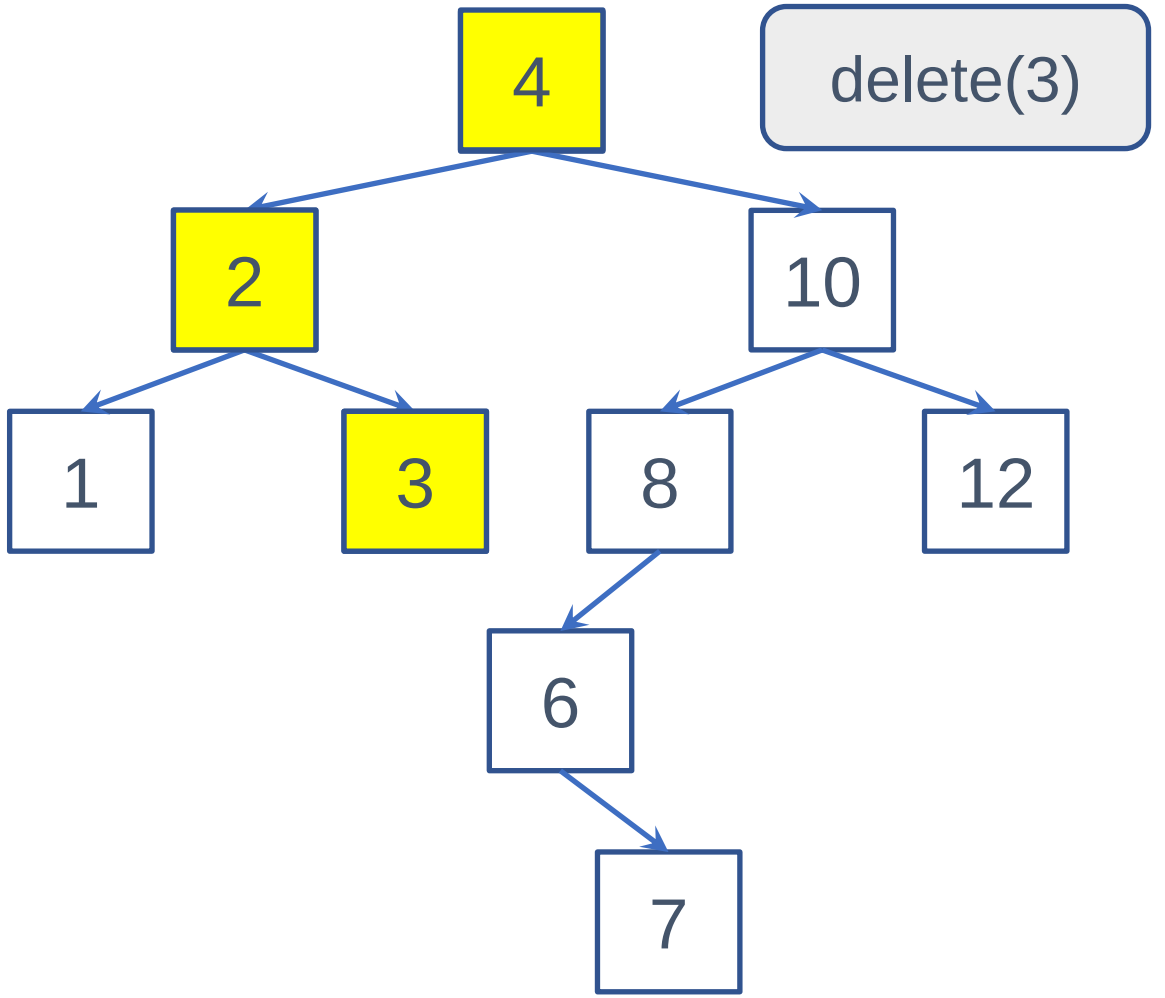
Binary Search Trees – Delete

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



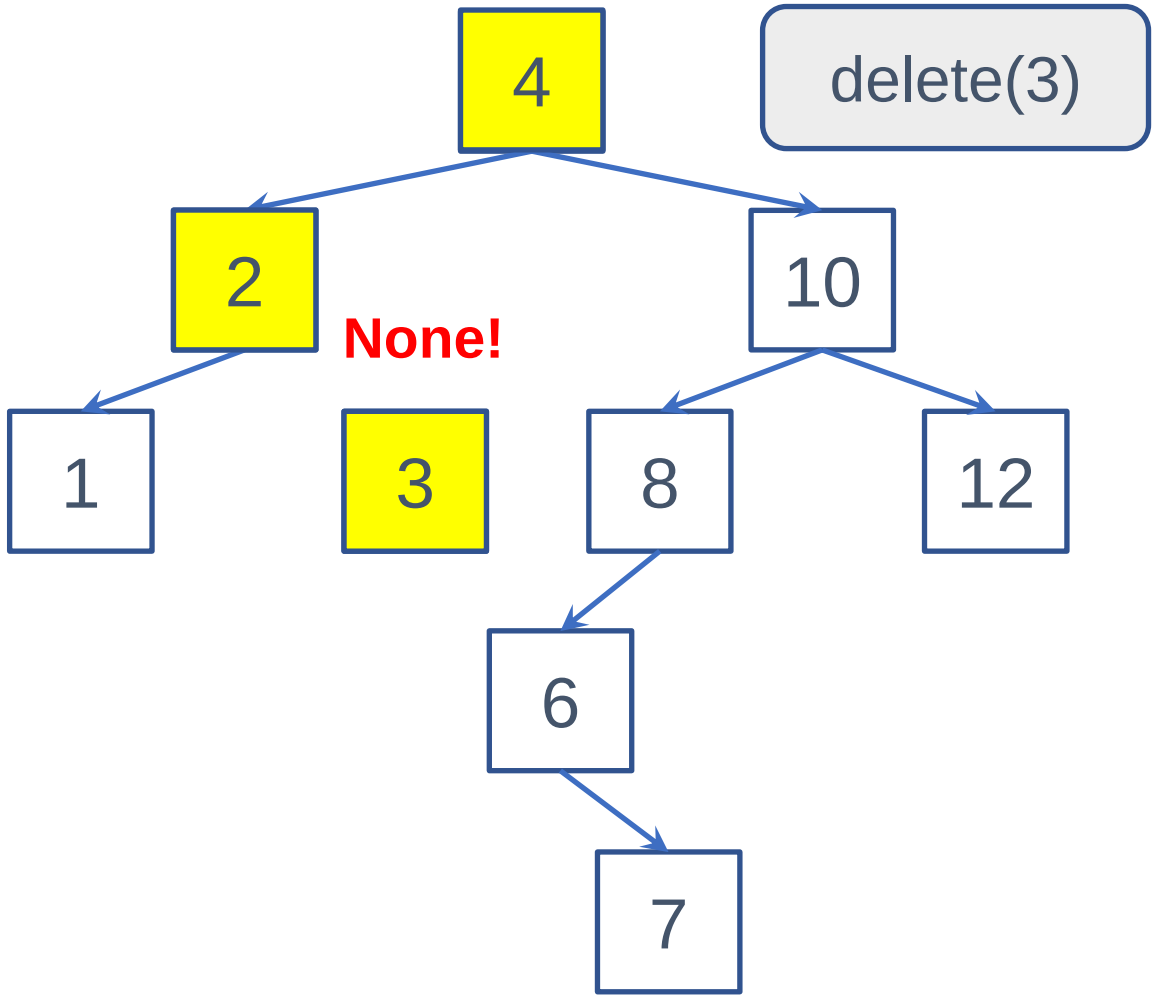
Binary Search Trees – Delete

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



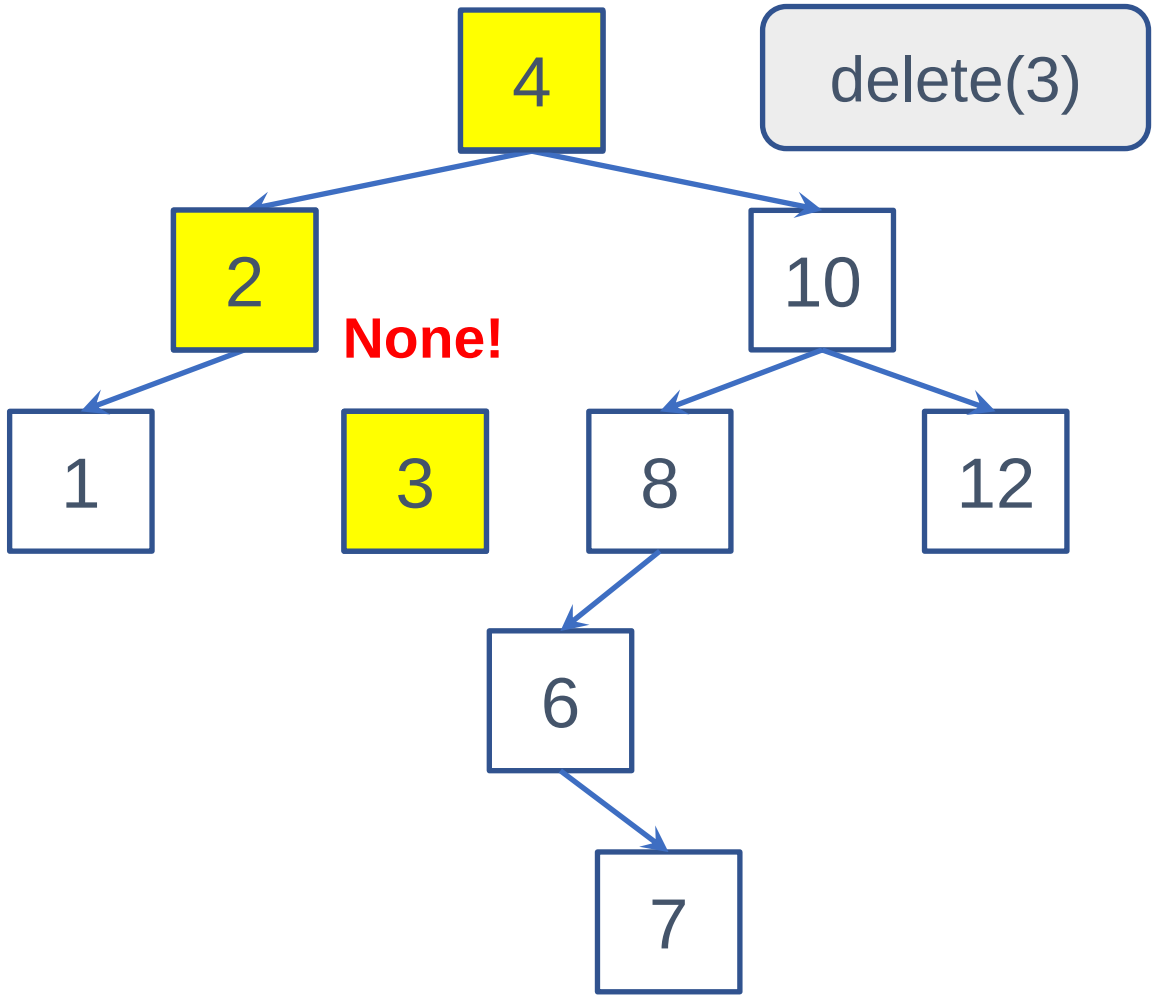
Binary Search Trees – Delete

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



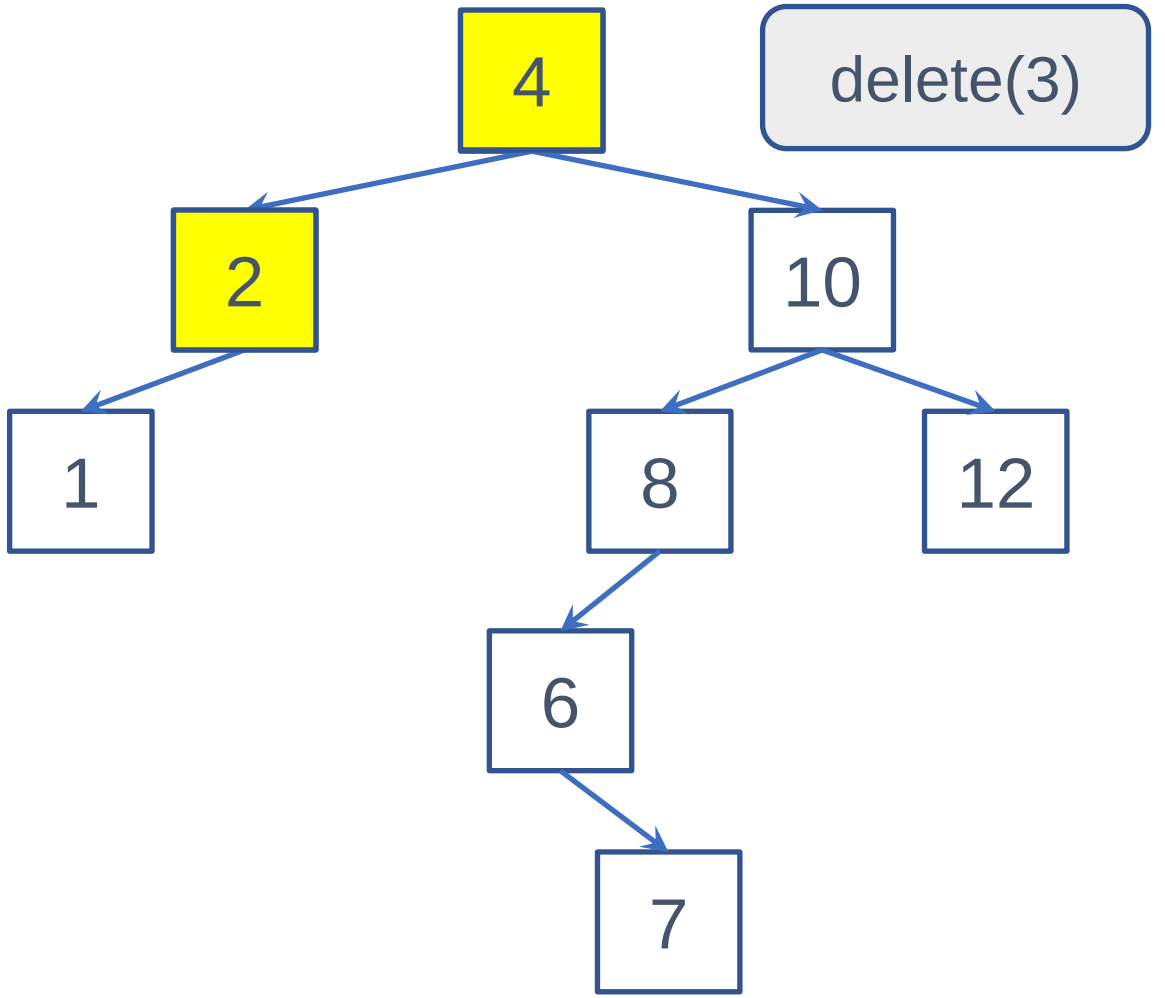
Binary Search Trees – Delete

- **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



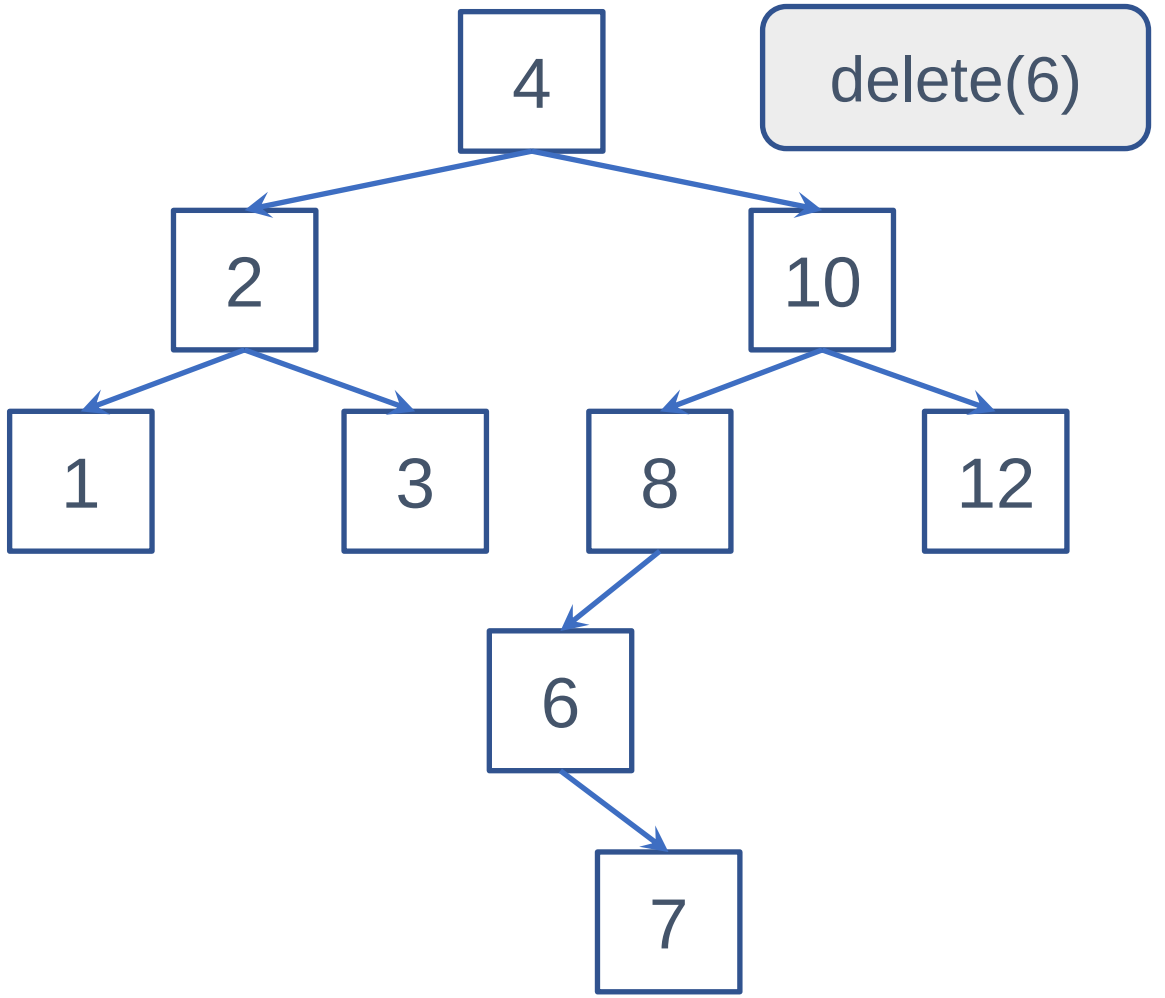
Binary Search Trees – Delete

- **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



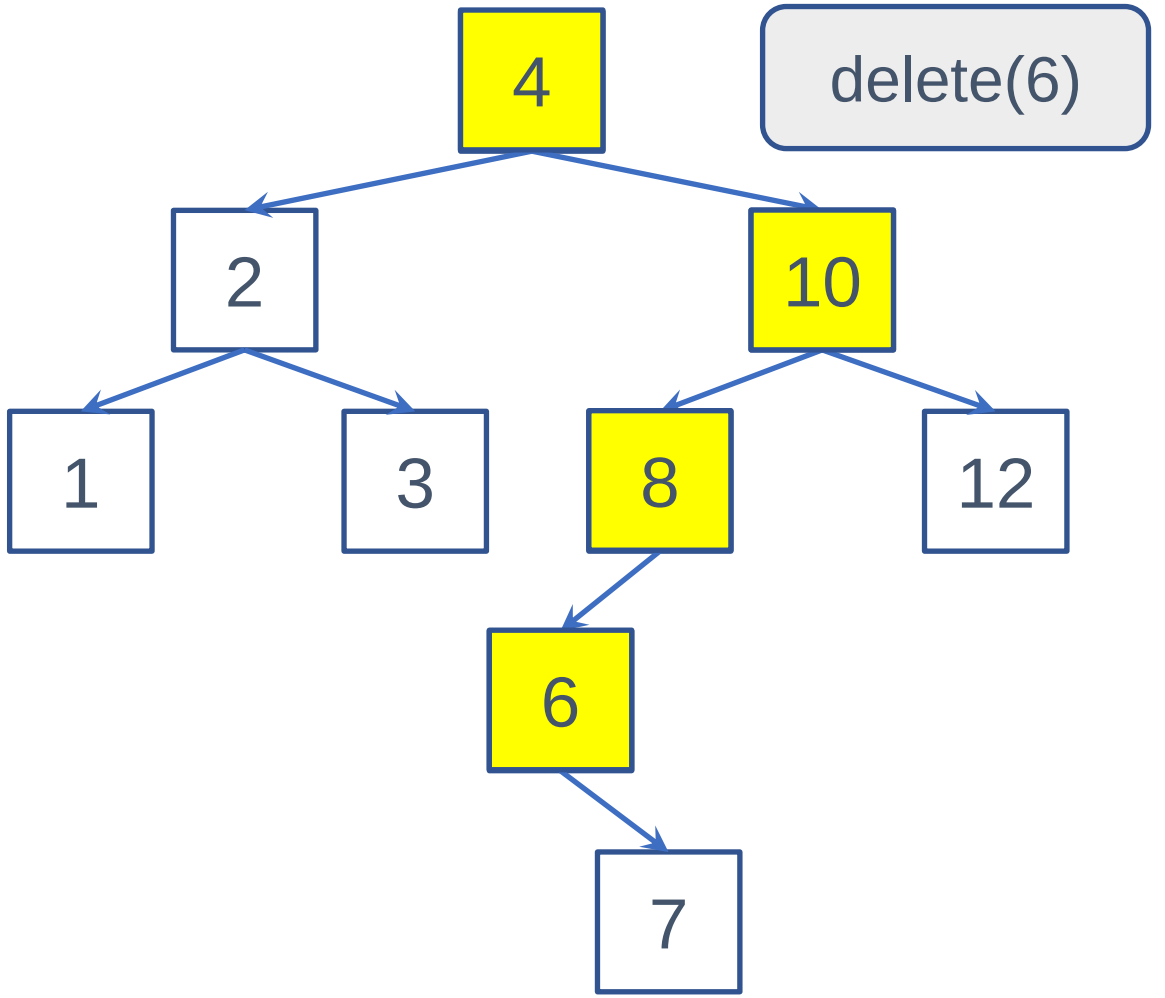
Binary Search Trees – Delete

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



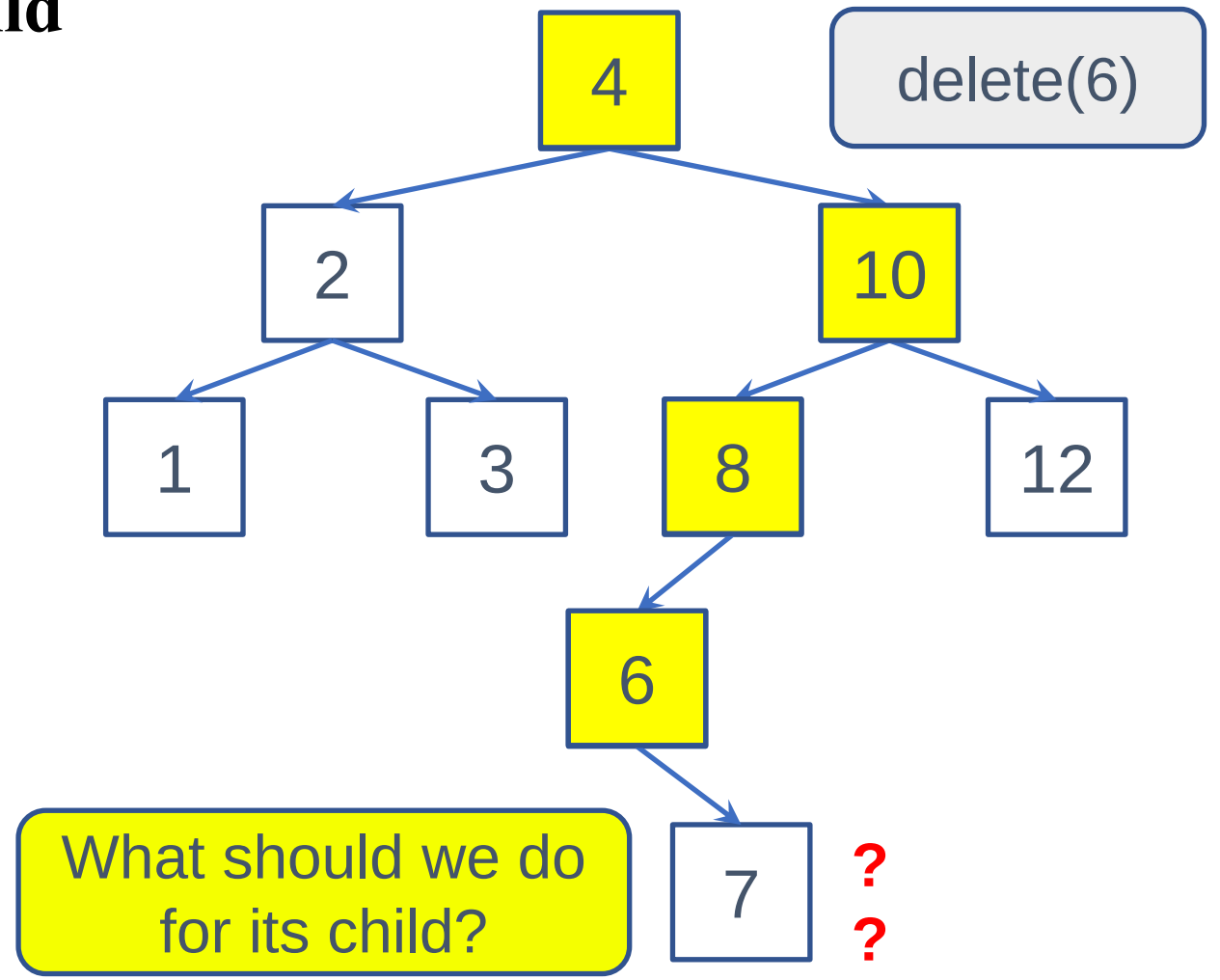
Binary Search Trees – Delete

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



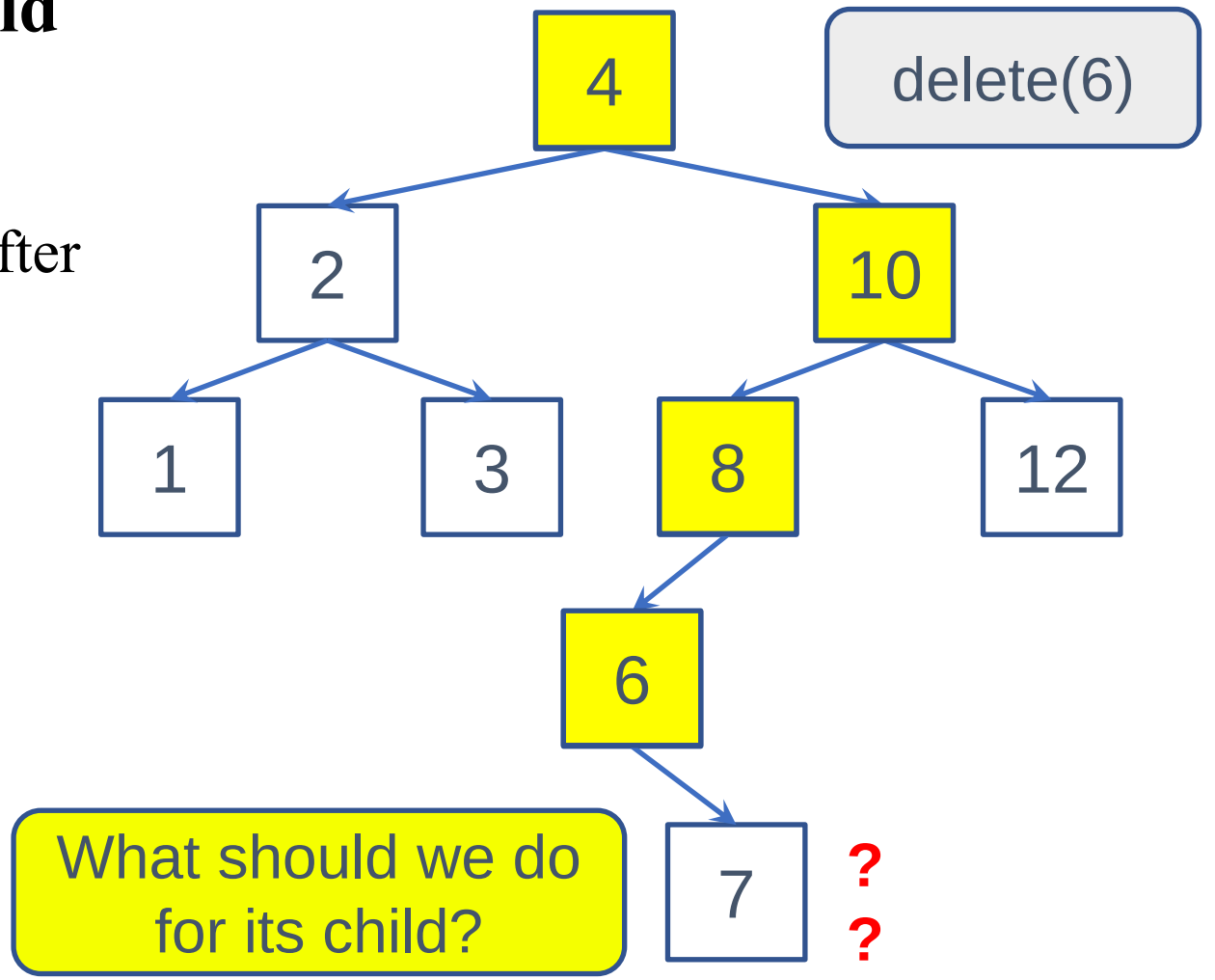
Binary Search Trees – Delete

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



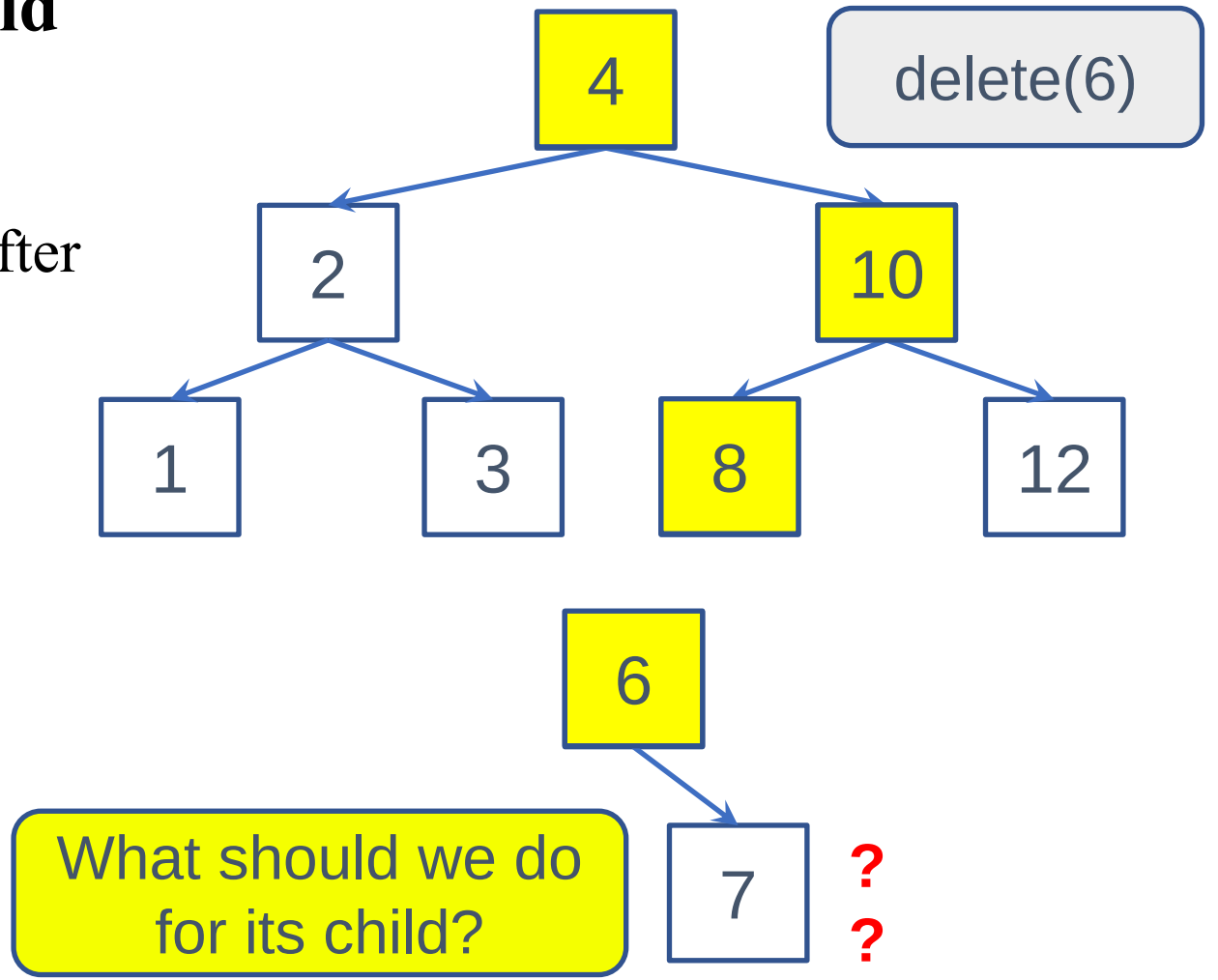
Binary Search Trees – Delete

- **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



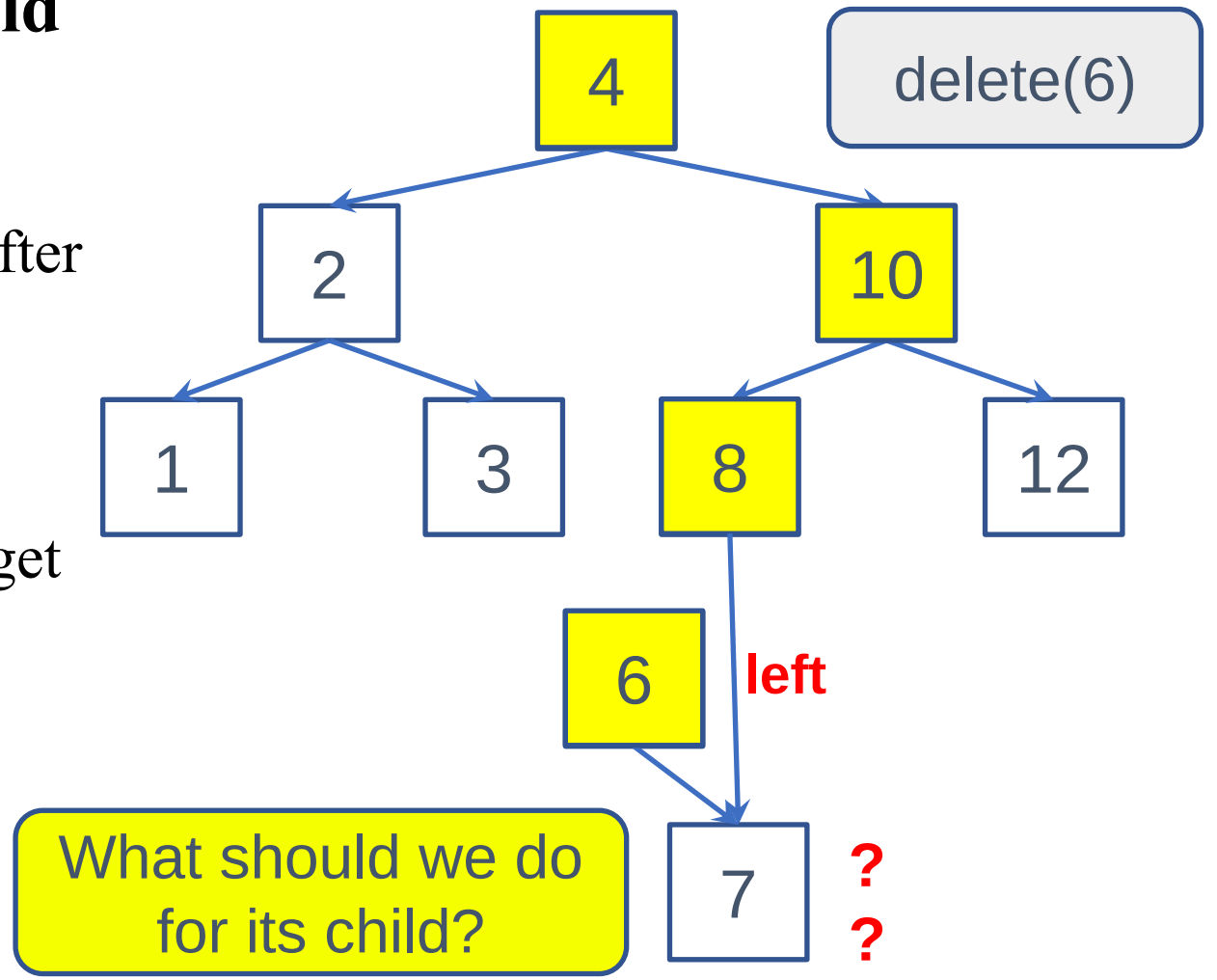
Binary Search Trees – Delete

- **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



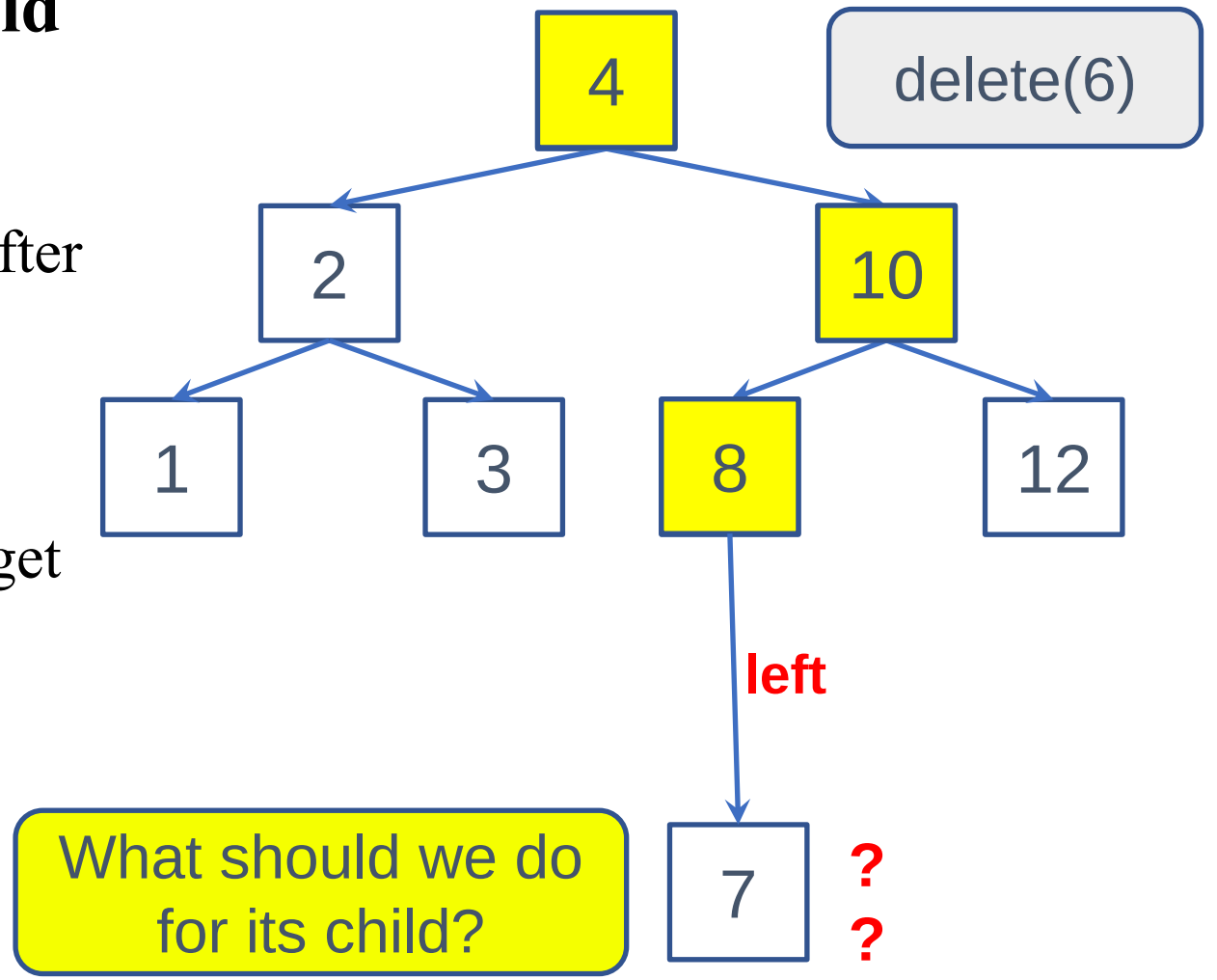
Binary Search Trees – Delete

- **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



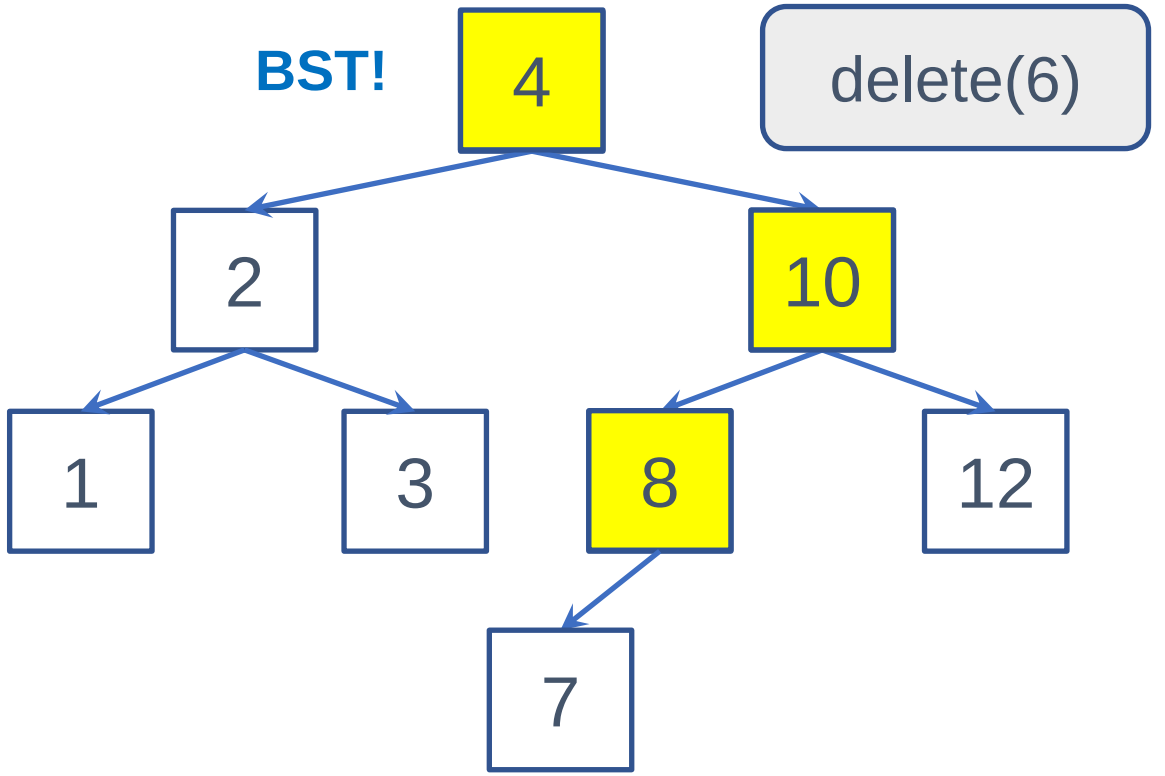
Binary Search Trees – Delete

- **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

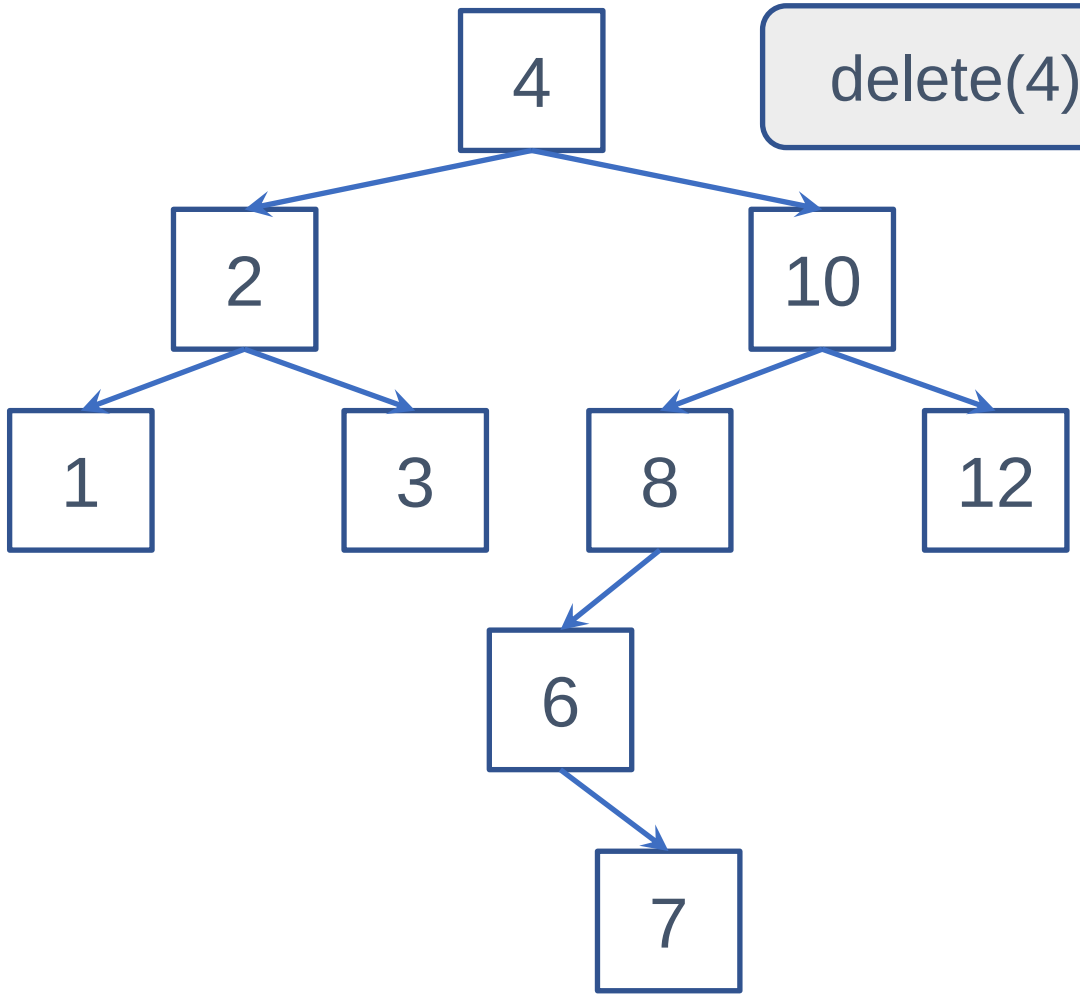
- **Case 2**: Delete a node with **one child**
  - **Search** the node using its key value
  - We should maintain **BST property** after removing the target node
  - Cut the parent's link to the target
  - Move the child node to where the target node was
  - Then the target node is gone



Binary Search Trees – Delete

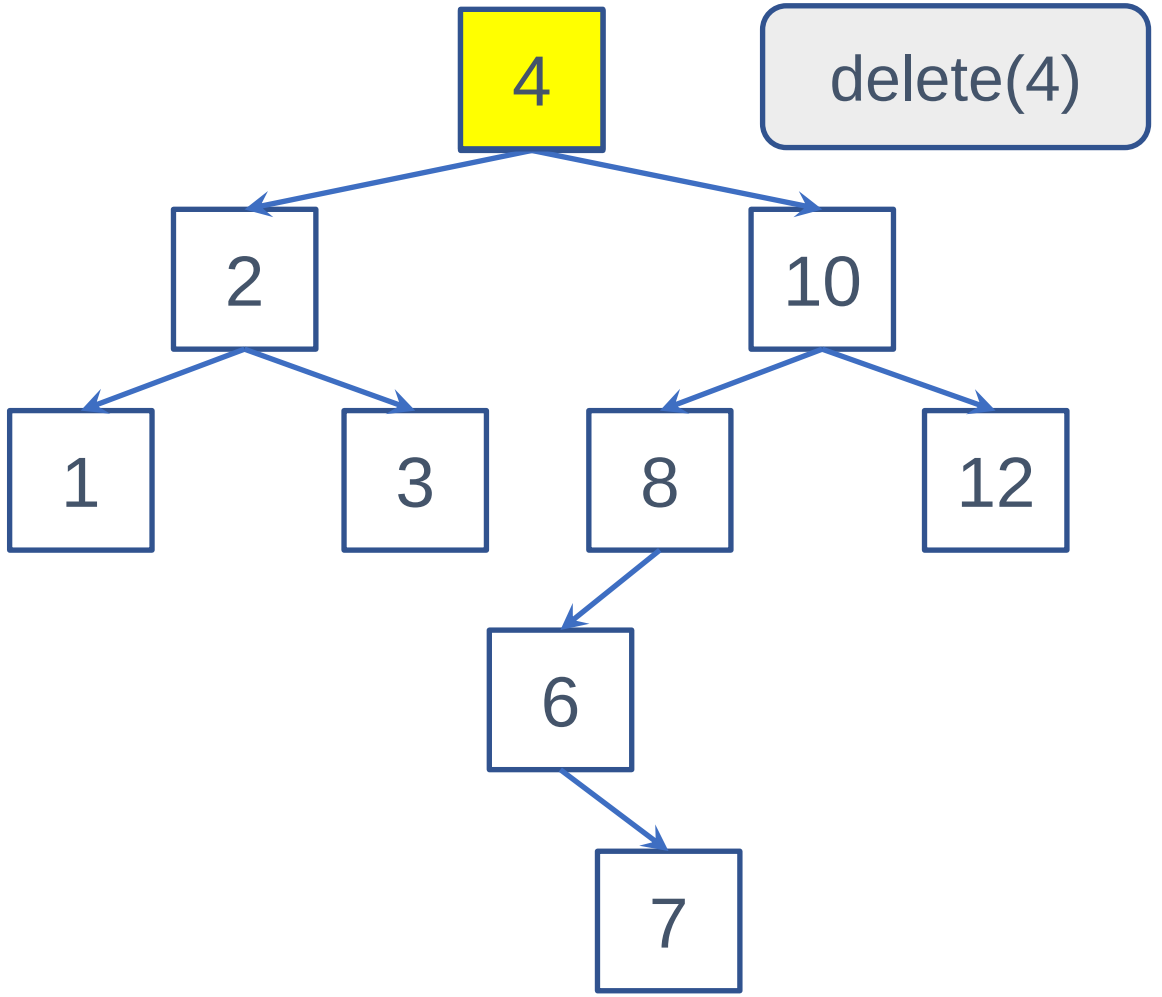
delete(4)

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



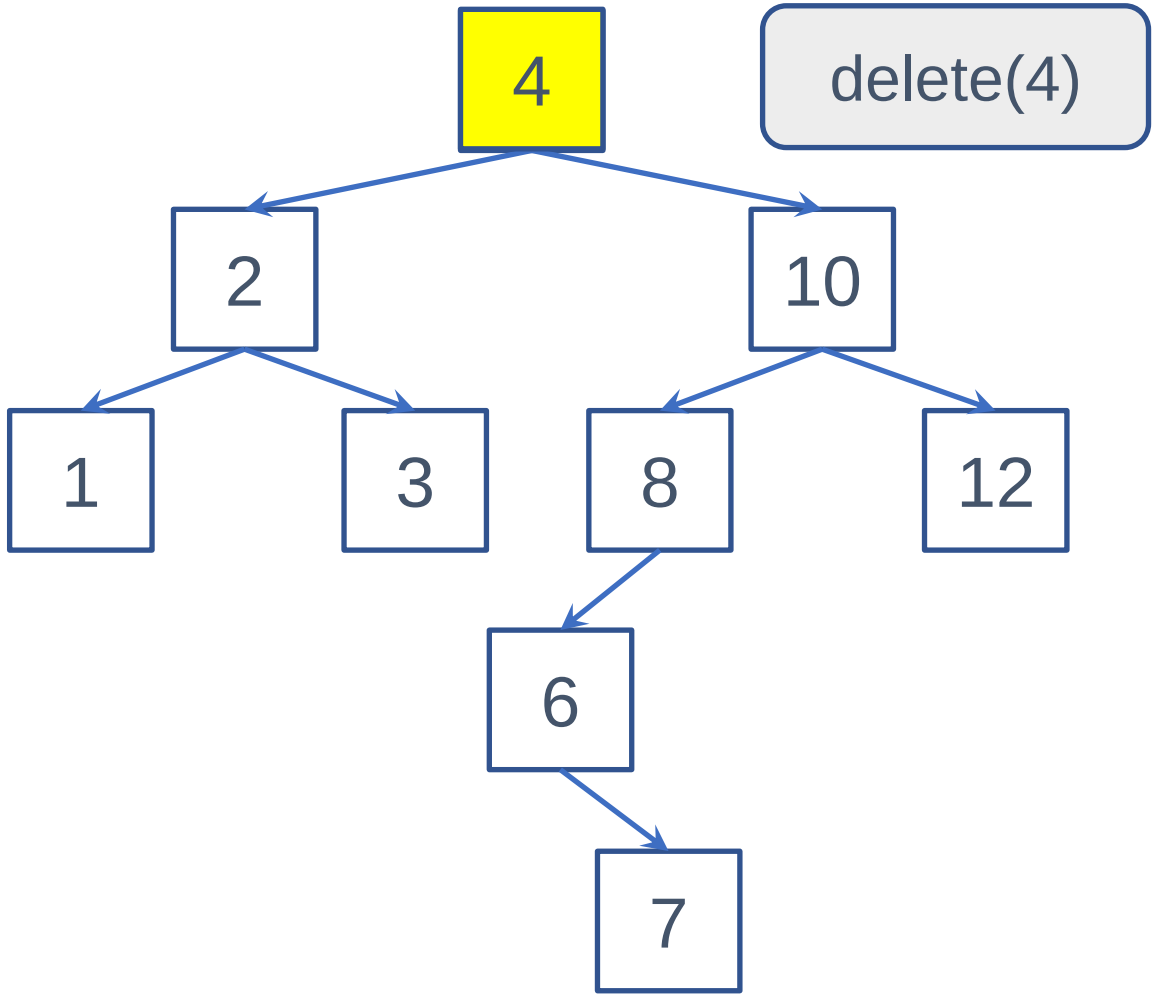
Binary Search Trees – Delete

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



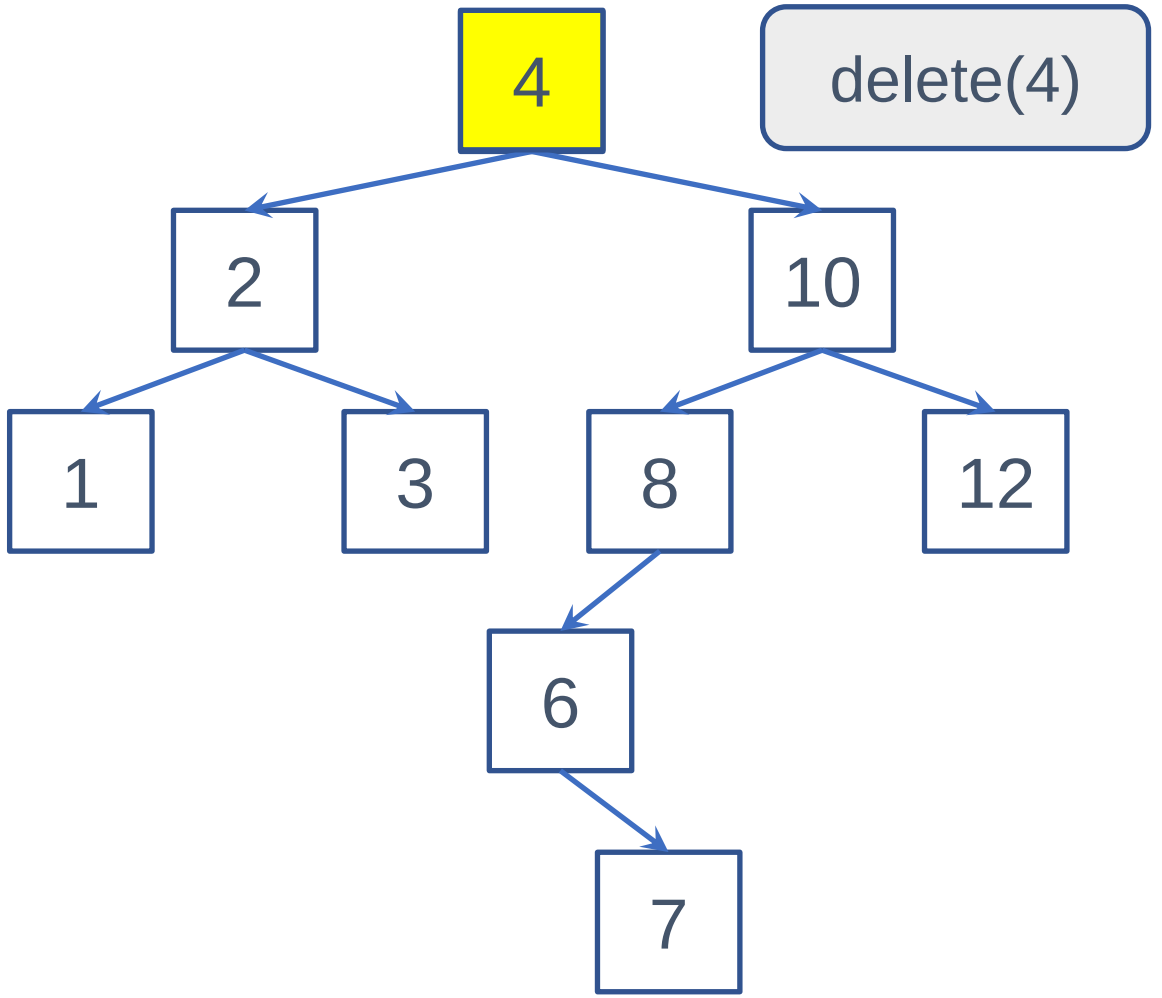
Binary Search Trees – Delete

- **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



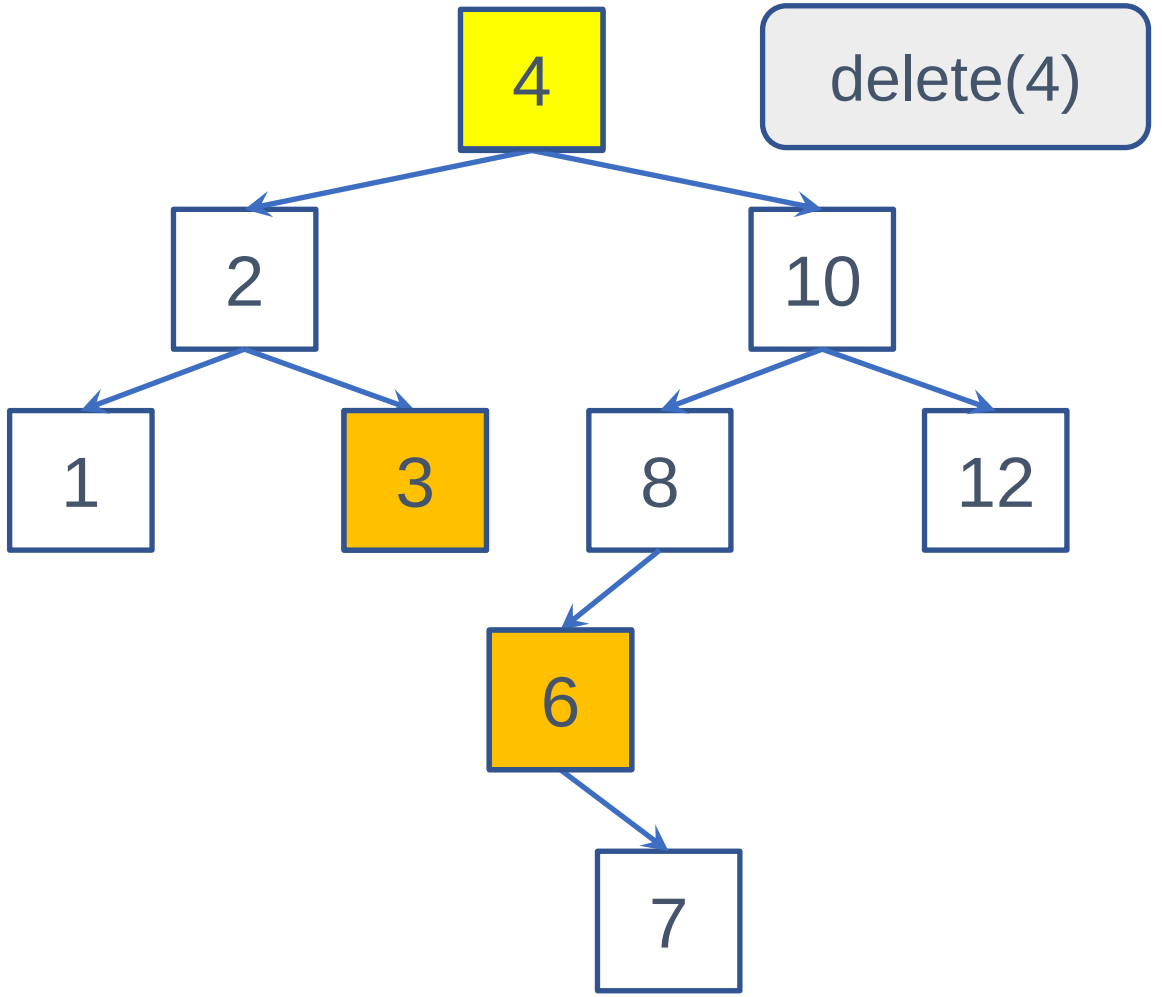
Binary Search Trees – Delete

- **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



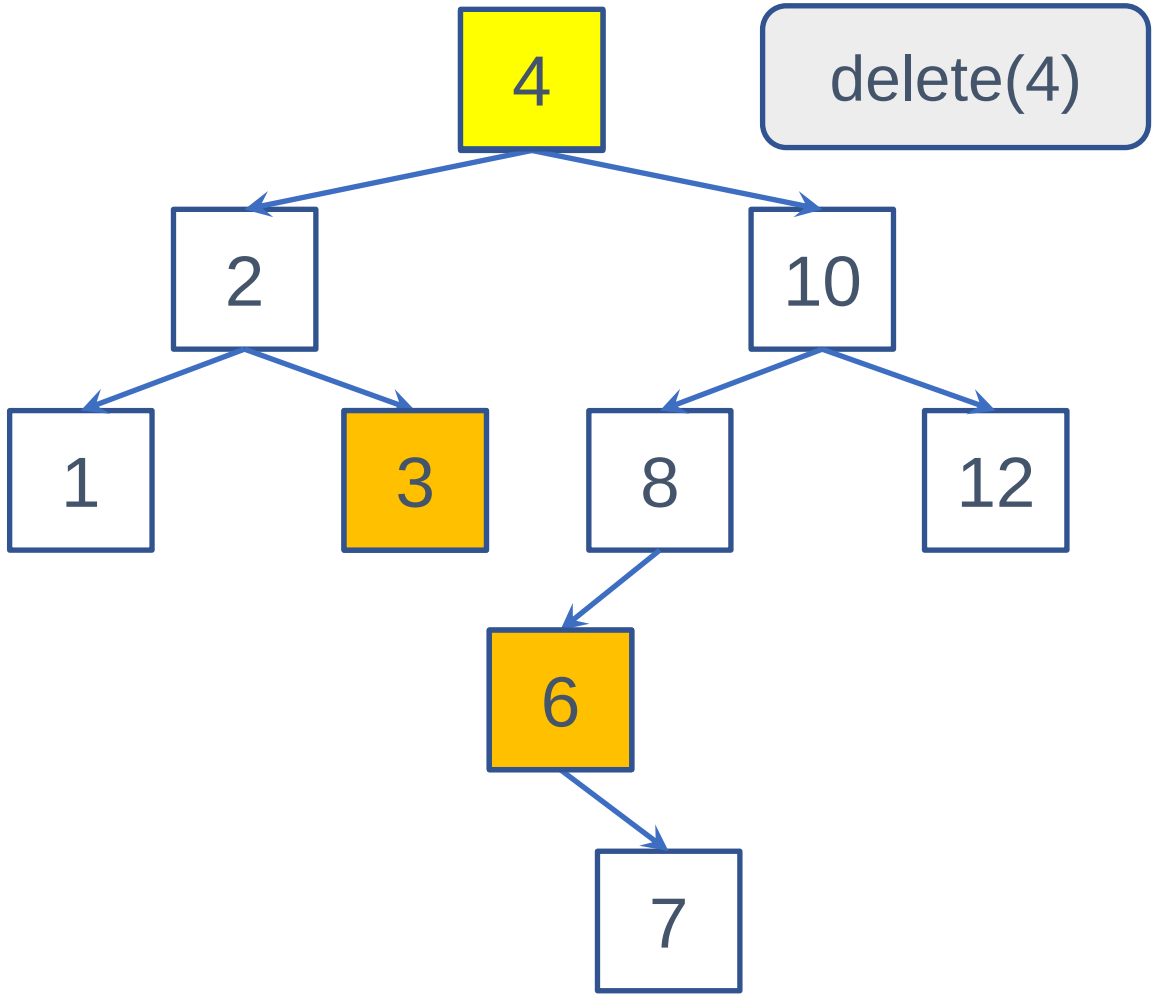
Binary Search Trees – Delete

- **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 the rightmost node in the left subtree or the leftmost node in the right subtree works



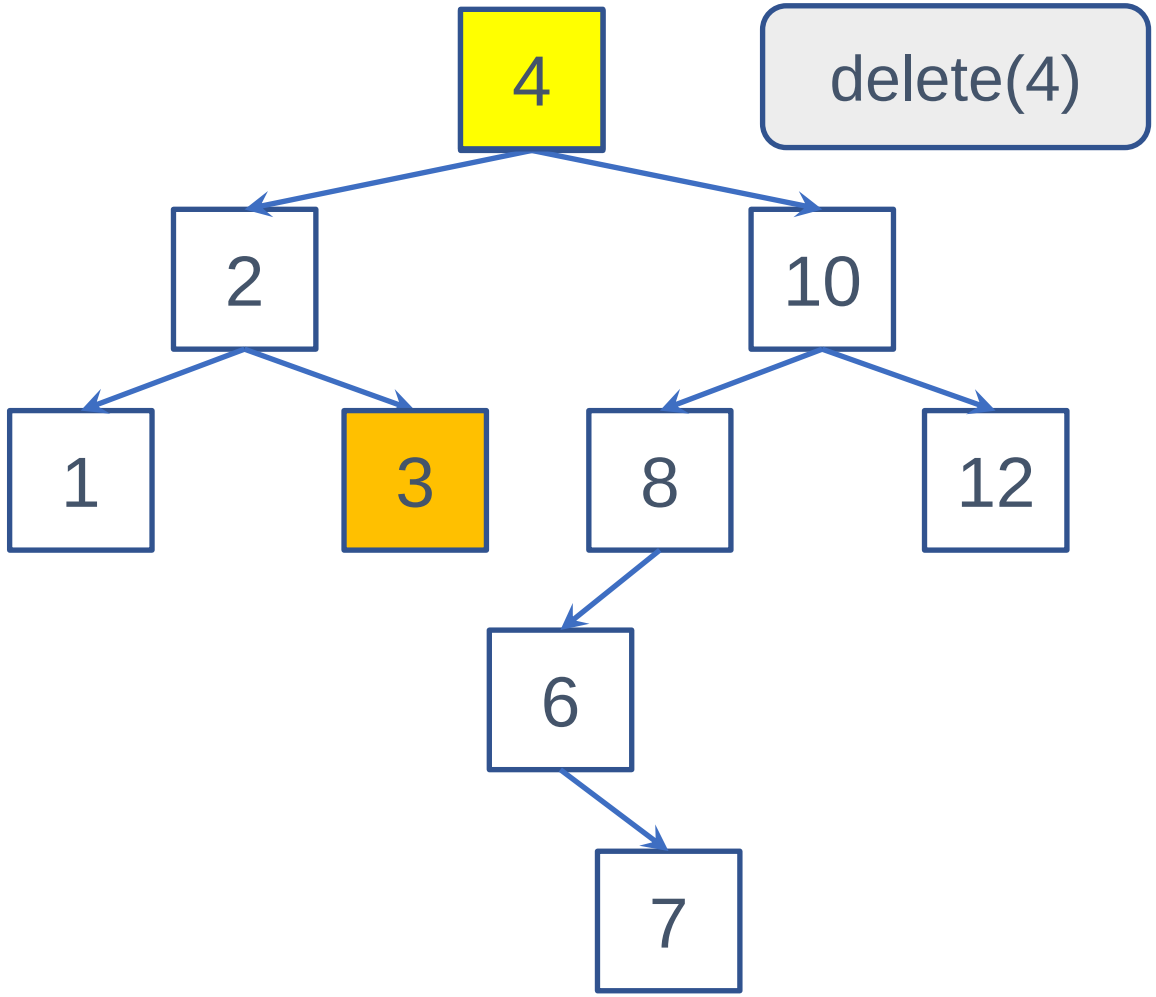
Binary Search Trees – Delete

- **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



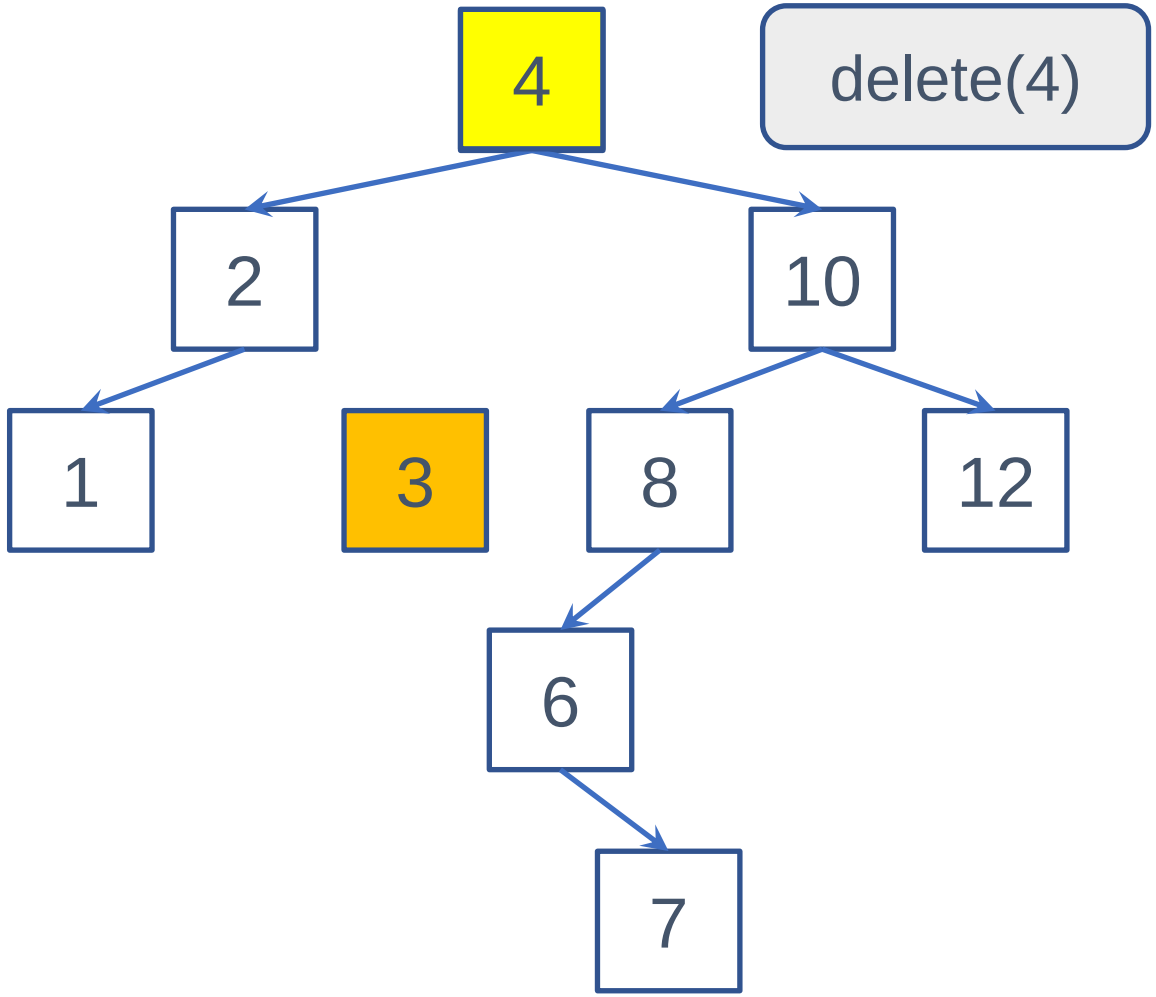
Binary Search Trees – Delete

- **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



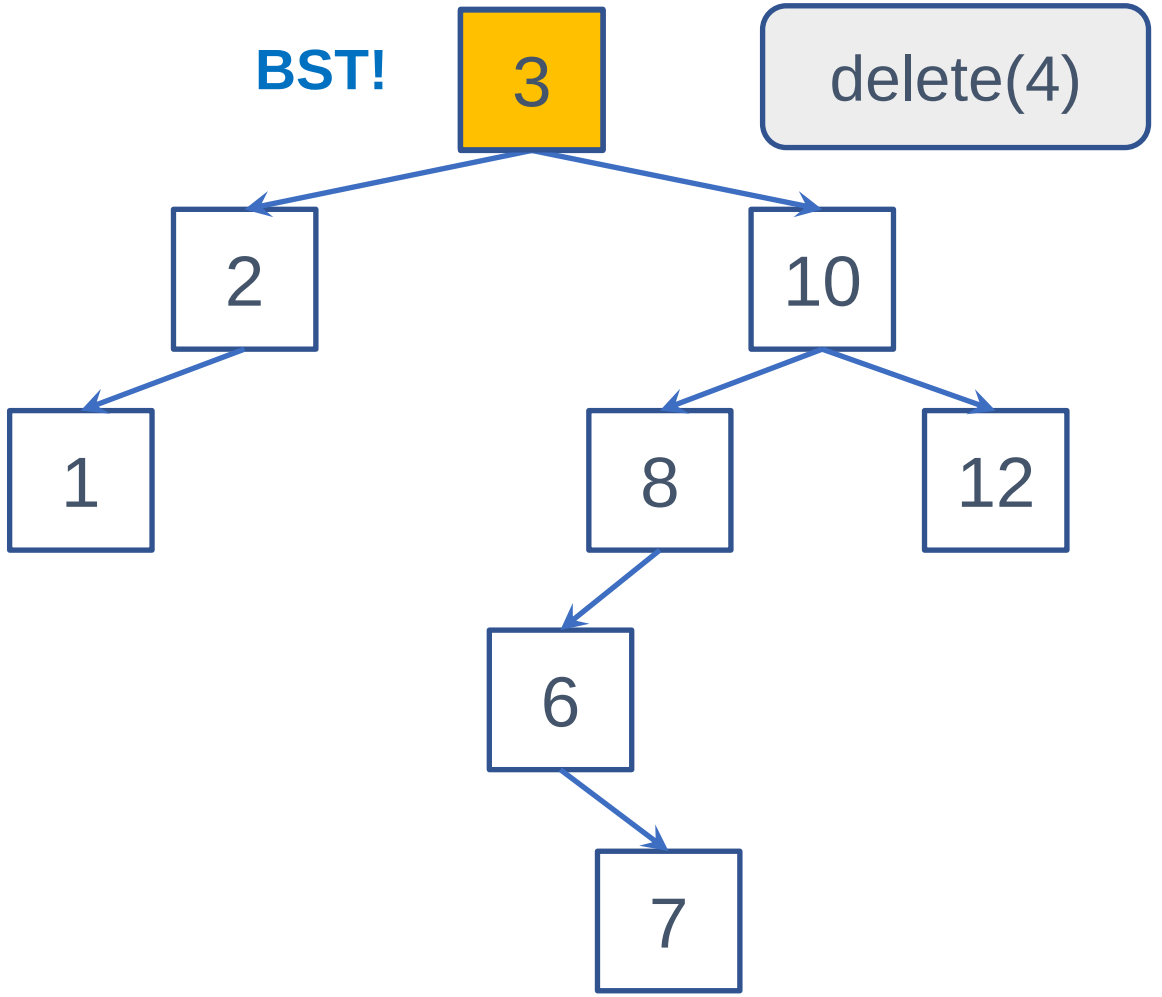
Binary Search Trees – Delete

- **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



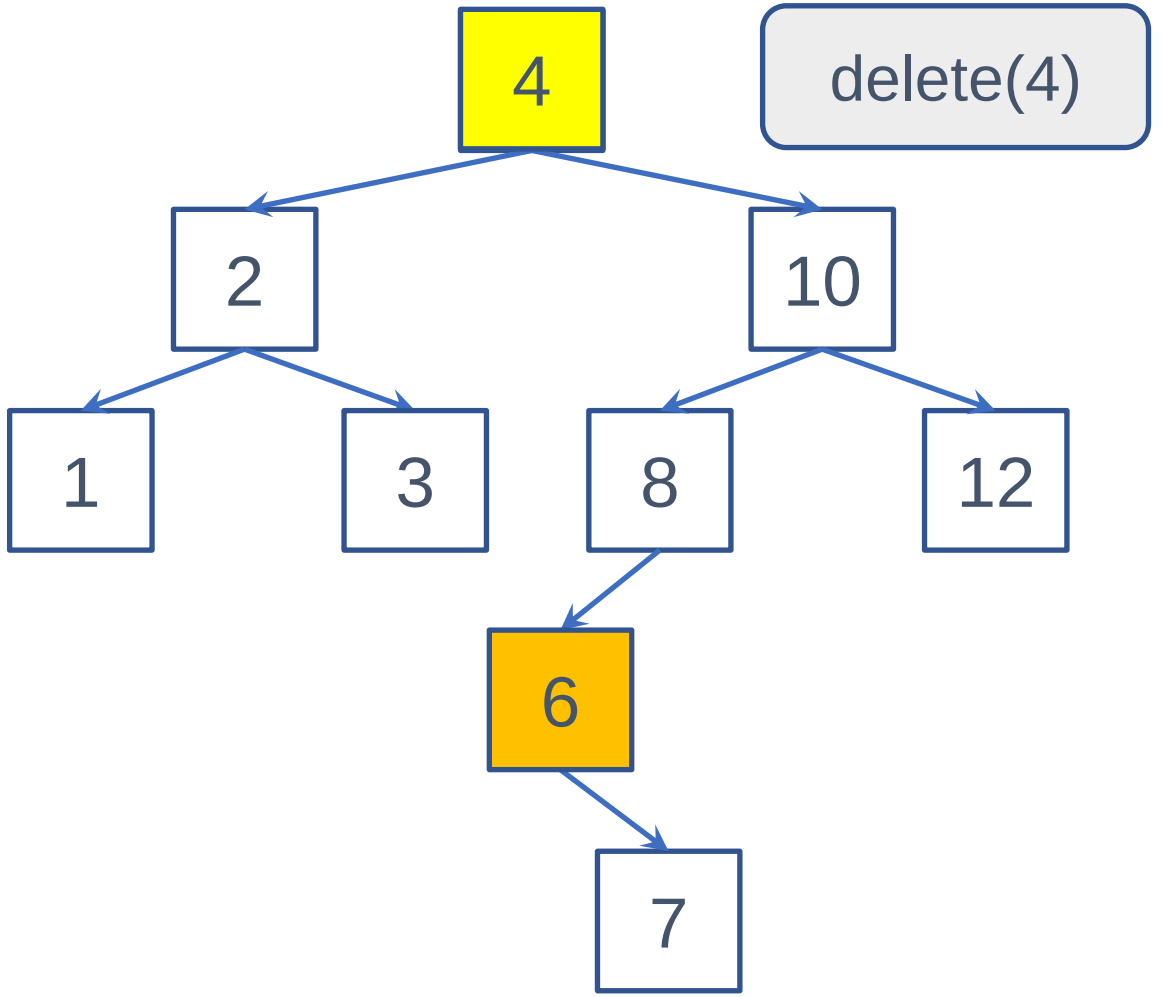
Binary Search Trees – Delete

- **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



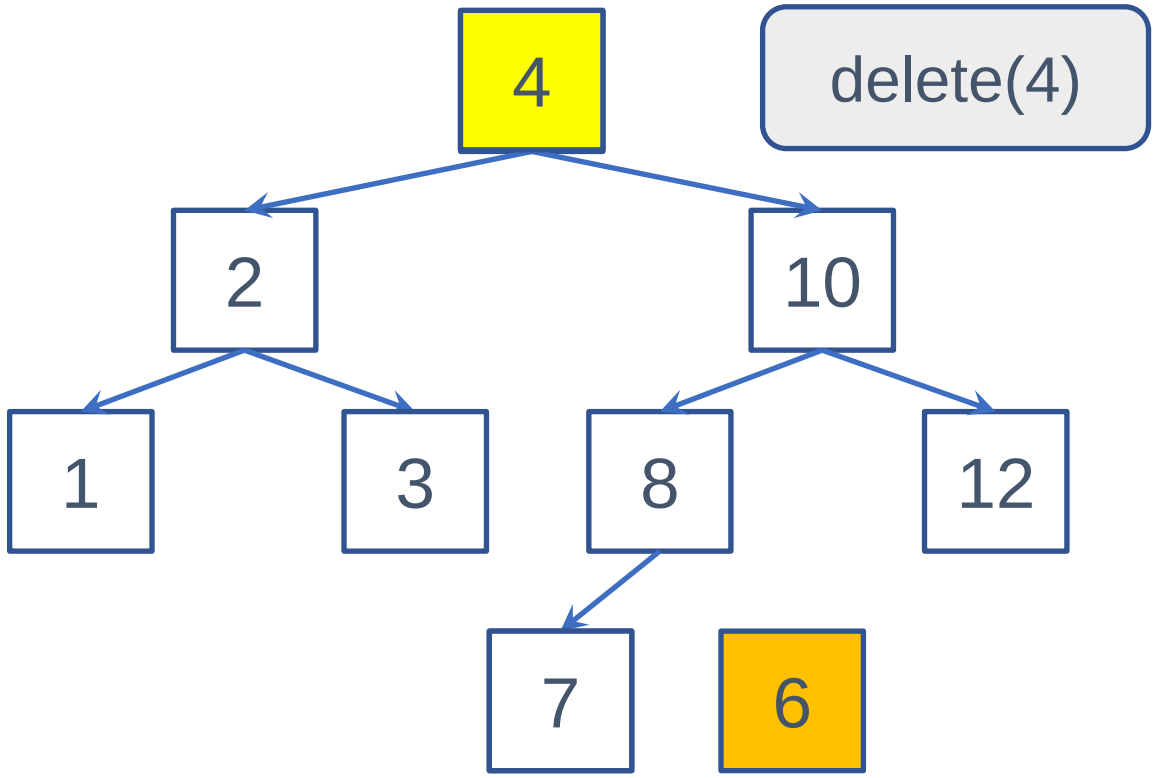
Binary Search Trees – Delete

- **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



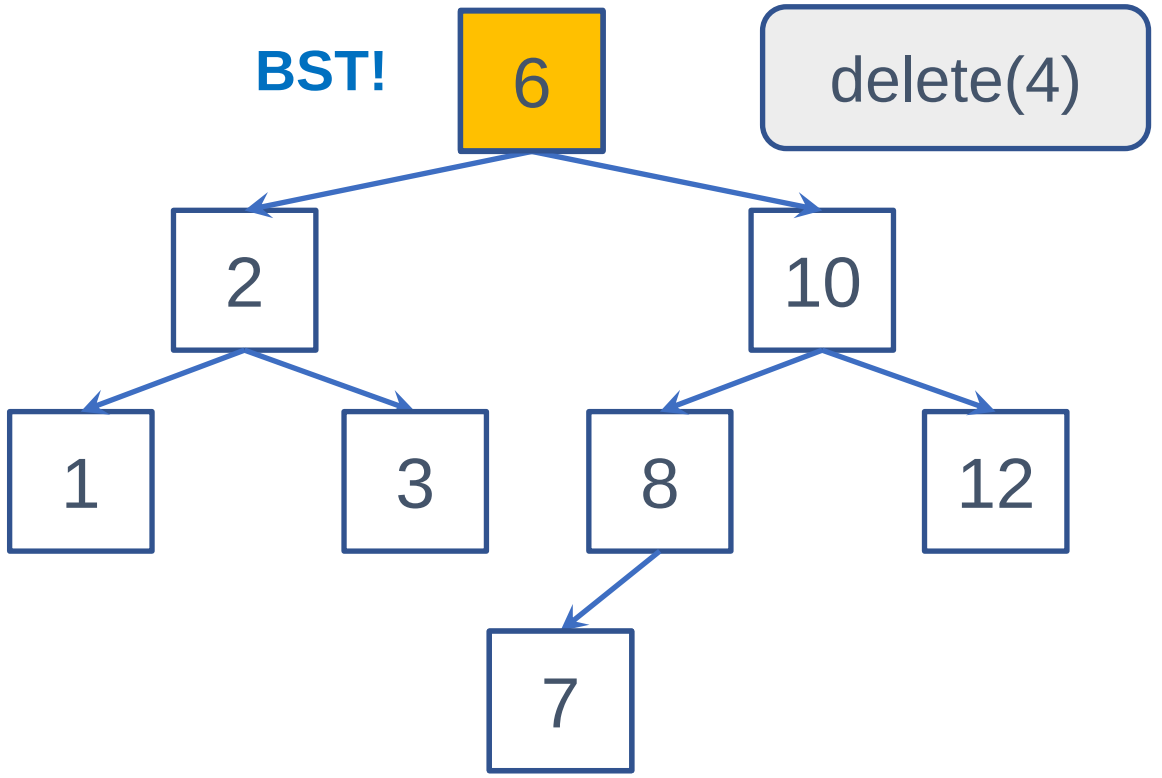
Binary Search Trees – Delete

- **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

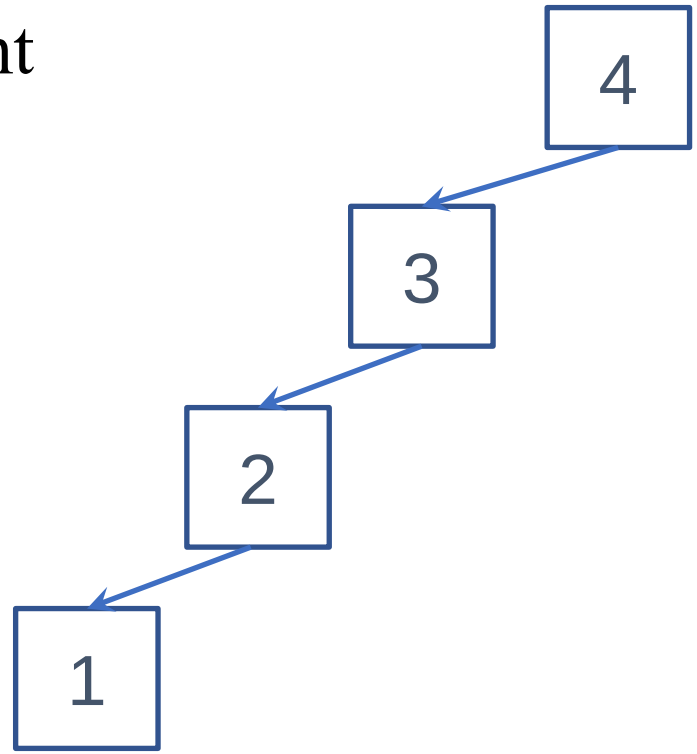
- **Case 3**: Delete a node with **two children**
  - **Search** the node using its key value
  - Delete either of the two
    - The rightmost node in the left subtree
    - The leftmost node in the right subtree
  - And place its copy at the target node's location
  - Ex.1) Delete 3
  - Ex.2) Delete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



Summary

Summary

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

Thanks!