Trees + Binary Search Tree + Priority Queues

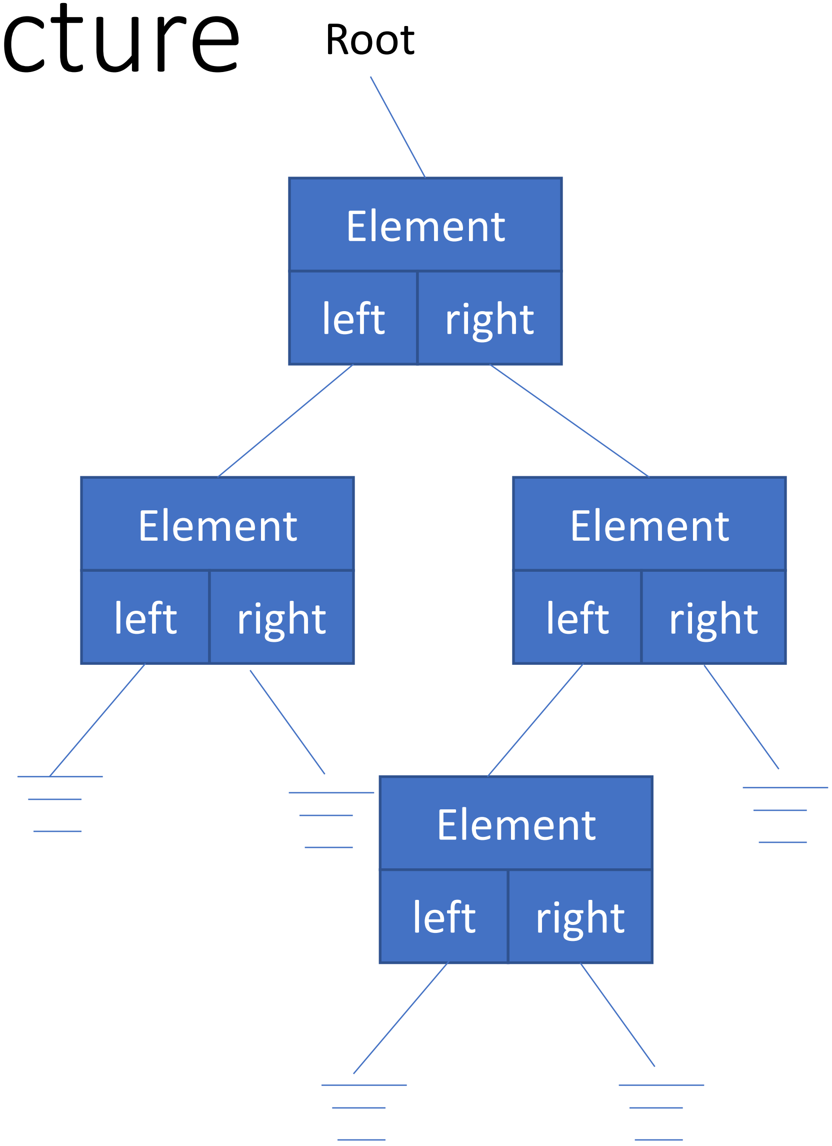
Instructor: Krishna Venkatasubramanian

CSC 212

Based on slides from Prof. Carolina Ruiz at WPI

The Tree Data Structure

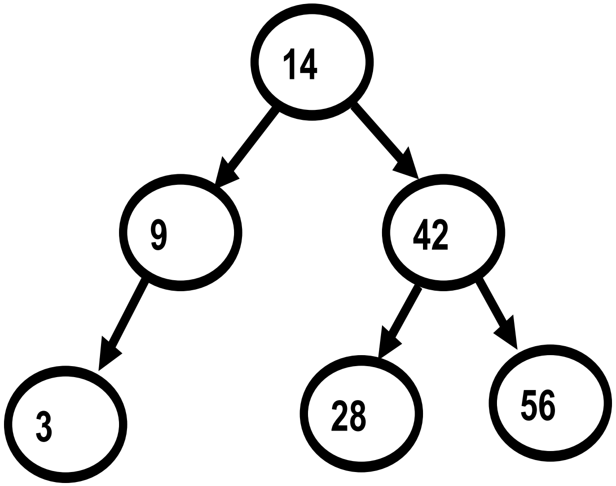
- A tree data structure is like a doubly linked list except now
  - we have a **left-child and right-child**
  - A **root instead of head**
- The leaf nodes have Null (empty object like None in Python) for it’s children
- In the HeapSort case, we imagined an array as a binary tree
- **Here, we actually store information in memory in a tree format.**



Binary Search Tree Definitions

- **Binary tree**
  - **Each node has at most two children**

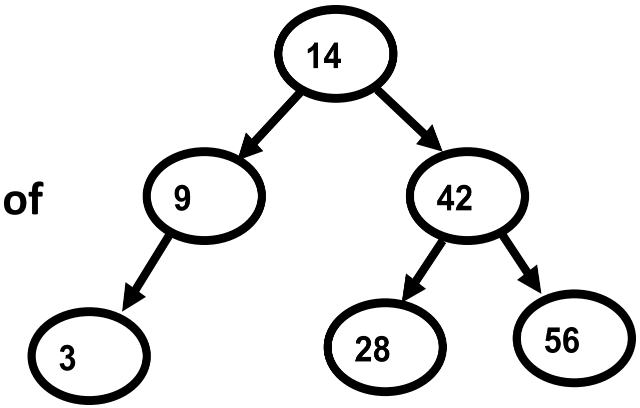
- **Binary Search Tree (BST): Is a binary tree where:**
  - **Left subtree is always less than the node**
  - **Right subtree is always greater than or equal the node**



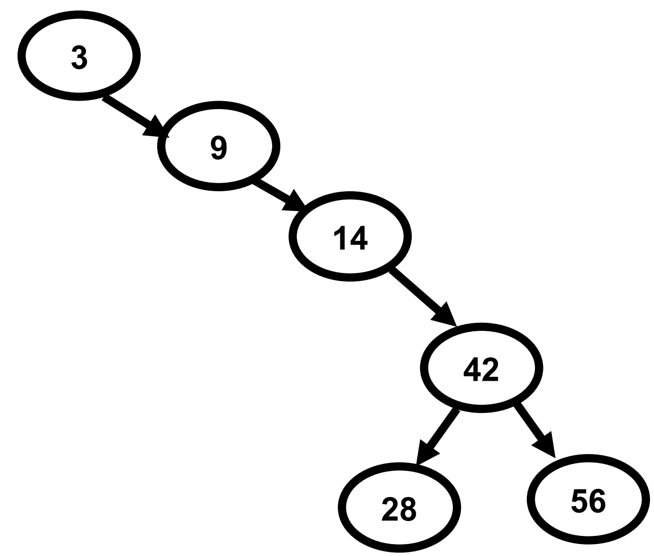
As is usual in Tree diagrams. The Null (None) Links for leaf nodes are not shown, unless needed

Height of BST

- Can be balanced
  - Height of left subtree of the root ≈ Height of right subtree of the root
  - In this case the height O(lg n)

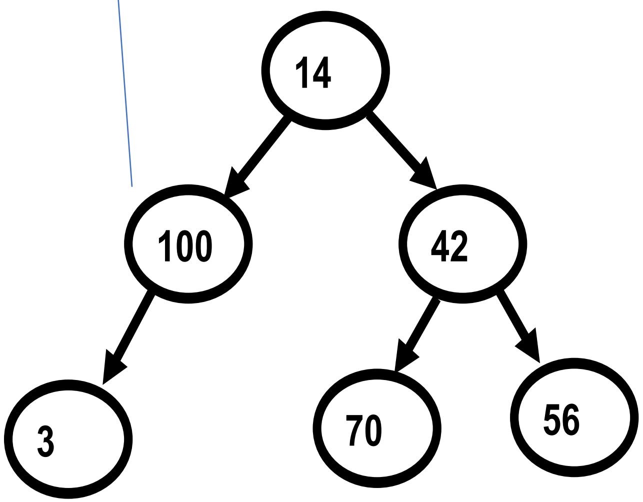


- Can be un-balanced
  - In this case the height is worst case O(n)

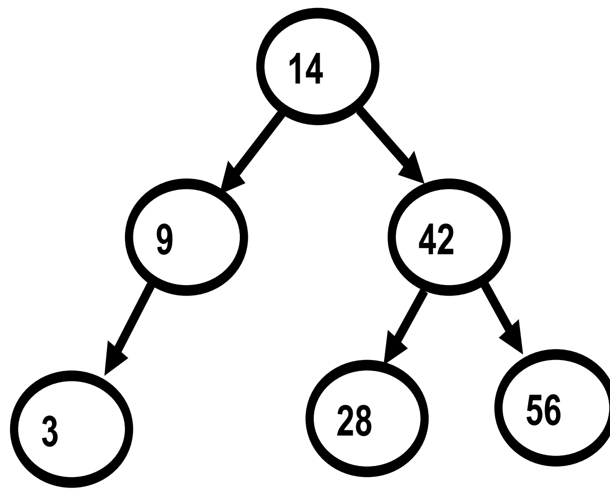


Binary Search Trees (BST)

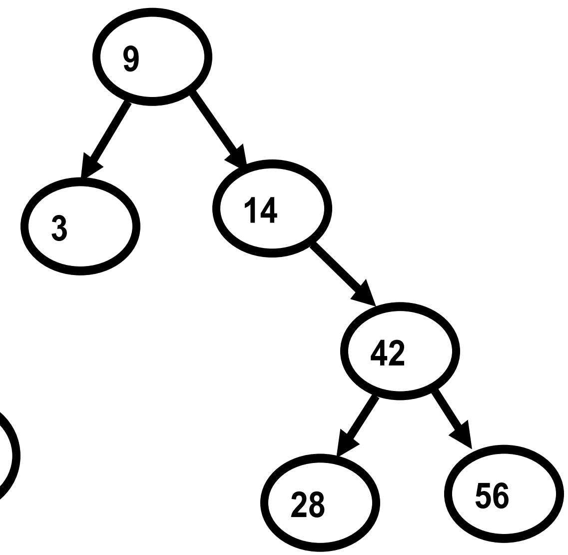
100 as the left child of 14



Binary tree
(but not BST)



Binary Search Tree
(Balanced)



Binary Search Tree
(Un-balanced)

BST Traversal

- With Linked List we talked about traversing through it.

- We can similarly traverse through BST (or any Tree) as well
  - Not as simple as going through a chain that is the Linked List
  - Also there are more than one way to traverse a tree top-bottom

- **Three main types**
  - **Pre-Order Traversal**
  - **Post-Order Traversal**
  - **In-Order Traversal**

Traversing a Binary Search Tree (BST)

Pre-Order

Outline of Pre-Order Traversal

- **For each node**
  - **Do the work first (Current)**
  - **Traverse Left**
  - **Traverse Right**

- **Work can be anything (E.g., print)**

Pre-Order Traversal Procedure (Pseudocode)

Start at the "Root" for a full tree traversal

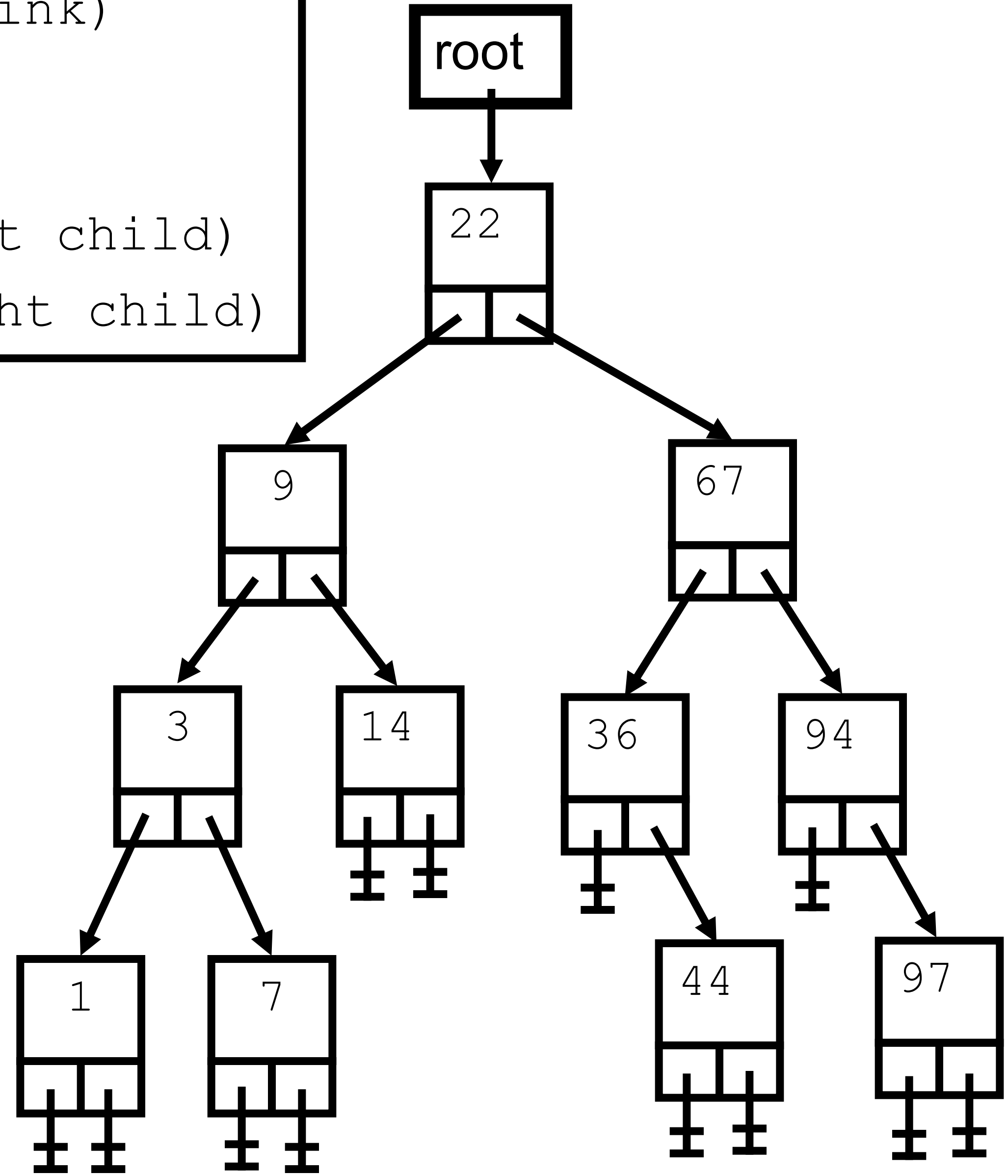
```
def Pre_Order(Link):
    if Link != None:
        Do_Whatever( Link.data )
        Pre_Order( Link.left_child )
        Pre_Order( Link.right_child )
```

E.g., print

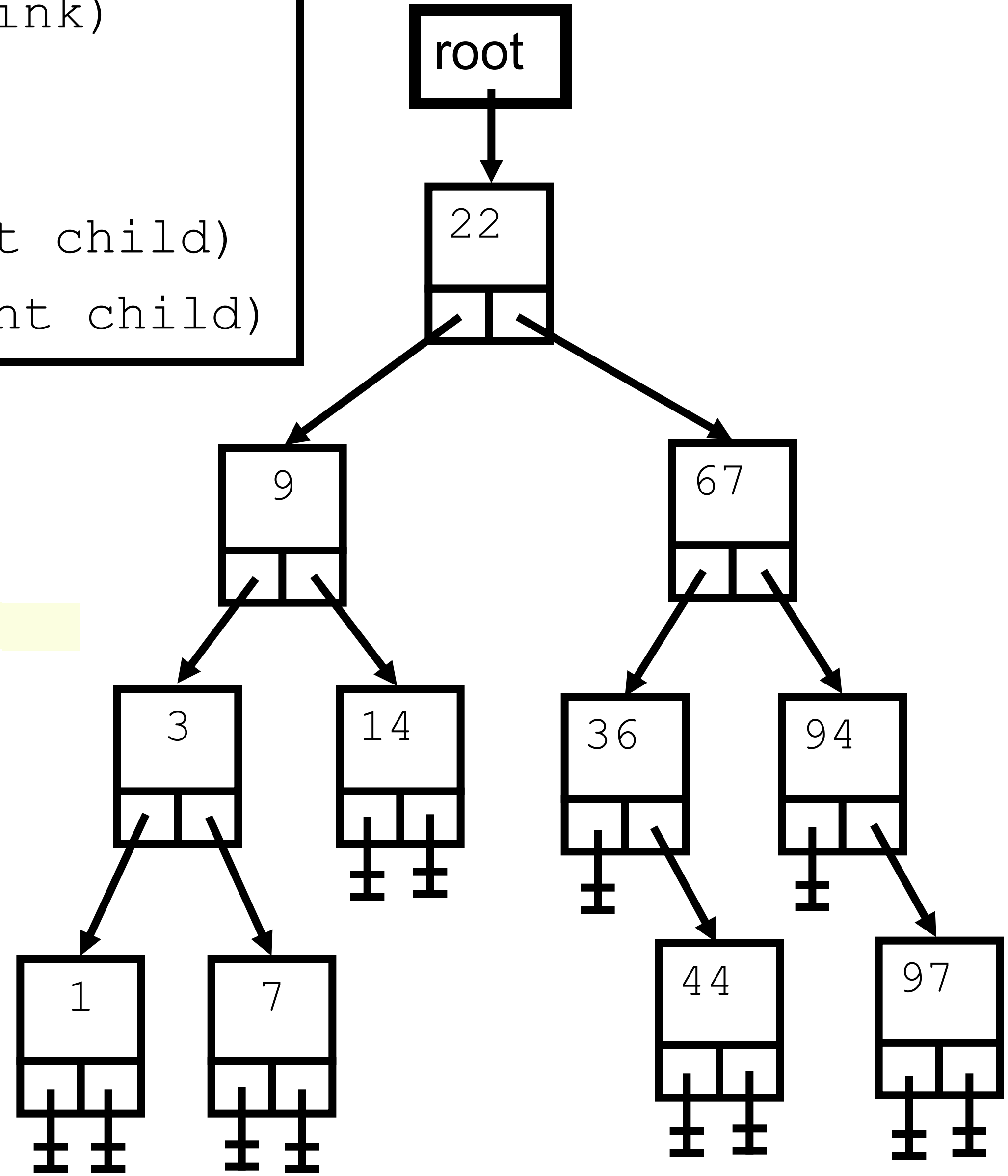
Two recursive calls

```
def PreOrderPrint(Link)
 Link NOT None?
P print(data)
L PreOrderPrint(left child)
P PreOrderPrint(right child)
```

Output



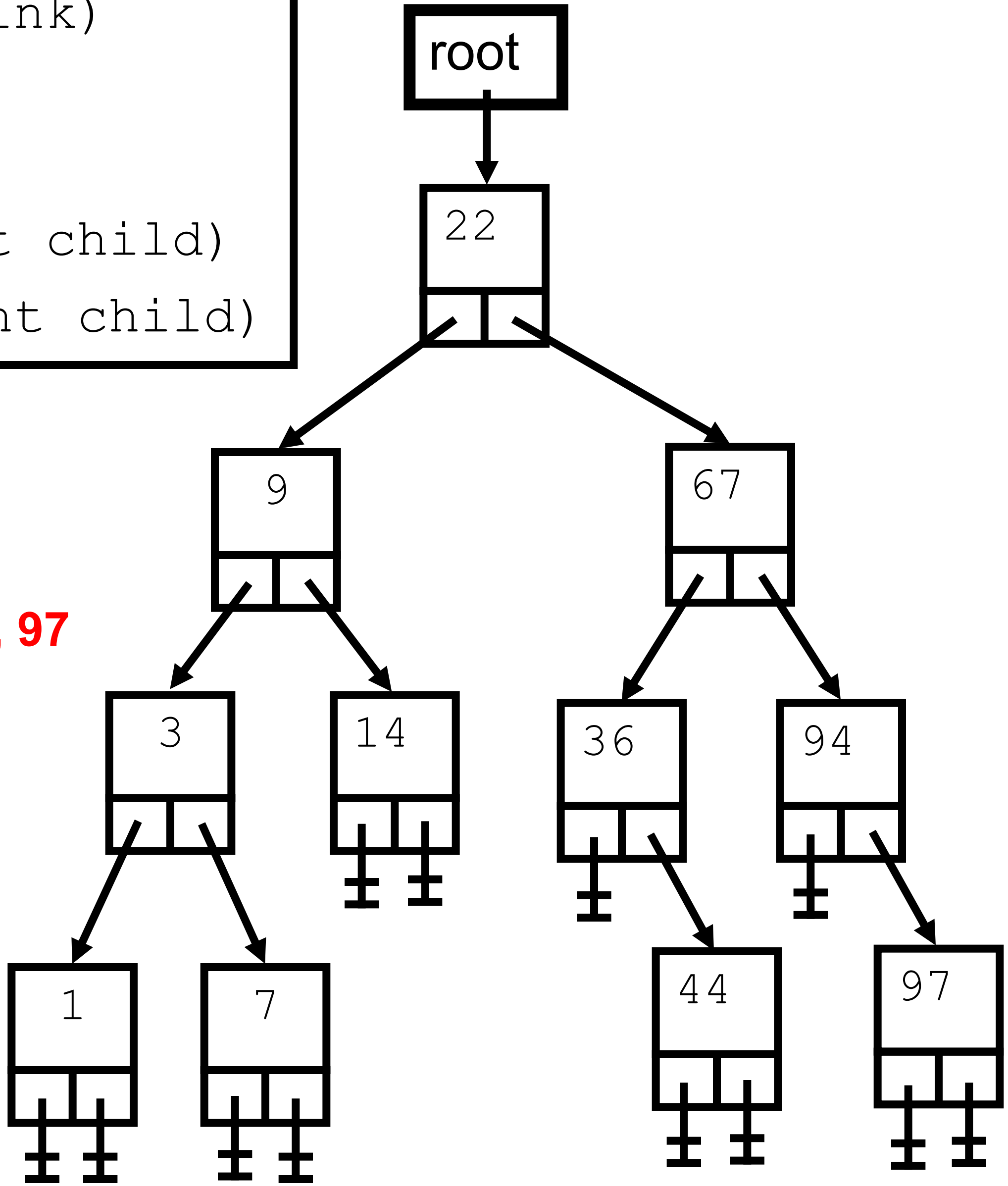
```
def PreOrderPrint(Link)
 Link NOT None?
P print(data)
L PreOrderPrint(left child)
R PreOrderPrint(right child)
```



Output

22

```
def PreOrderPrint(Link)
 Link NOT None?
P print(data)
L PreOrderPrint(left child)
R PreOrderPrint(right child)
```



Output

22, 9, 3, 1, 7, 14, 67, 36, 44, 94, 97

Traversing a Binary Search Tree (BST)

Post-Order

Outline of Post-Order Traversal

- **For each node**
  - **Traverse Left**
  - **Traverse Right**
  - **Do the work (Current)**

Post-Order Traversal Procedure (Pseudocode)

```
def Post_Order(Link):
    if Ptr != NIL:
        Post_Order( Link.left_child )
        Post_Order( Link.right_child )
        Do_Whatever( Link.data )
```

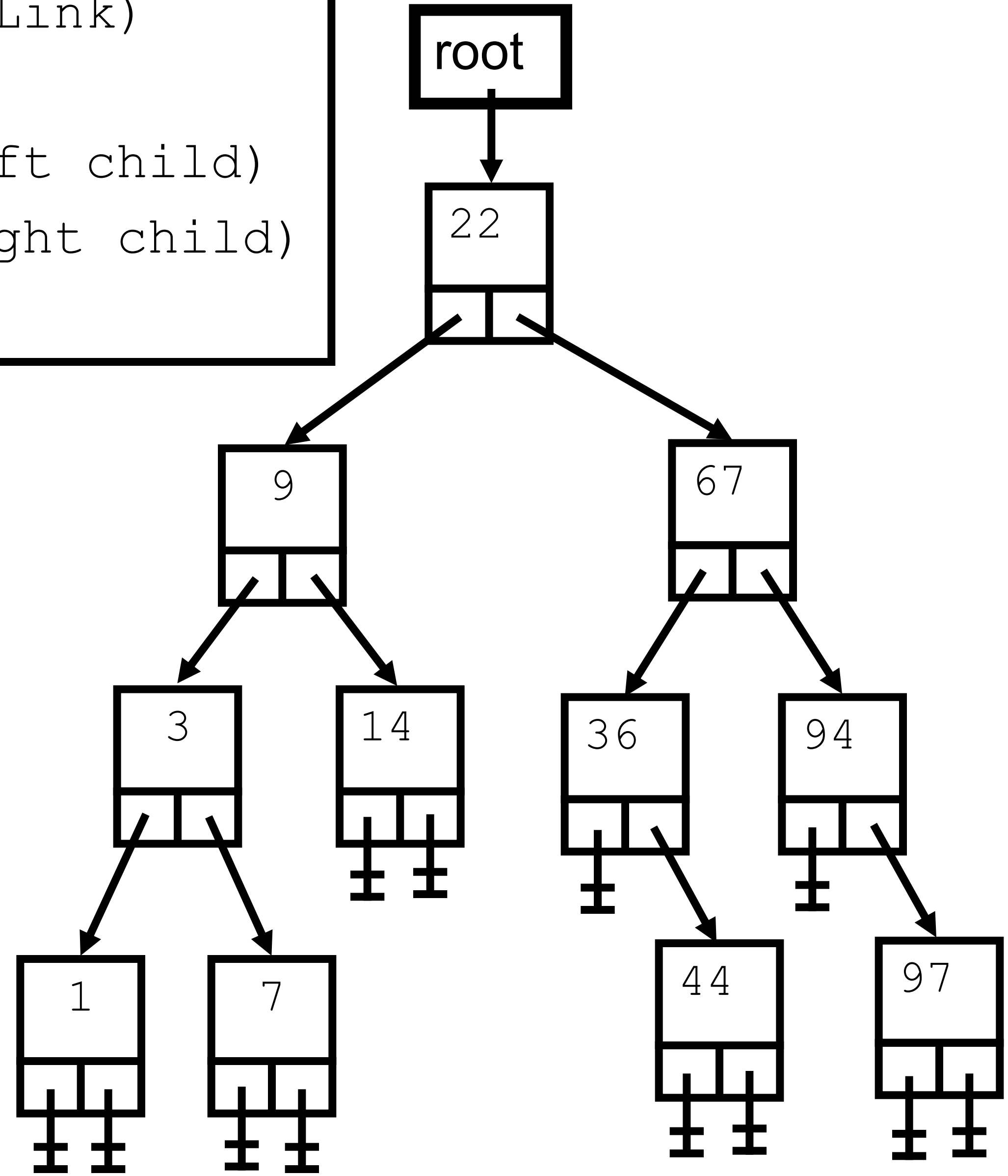
Start at the "Root" for a full tree traversal

Two recursive calls

E.g., print

```
def PostOrderPrint(Link)
 Link NOT None?
Ⓛ PostOrderPrint(left child)
Ⓡ PostOrderPrint(right child)
Ⓟ print(data)
```

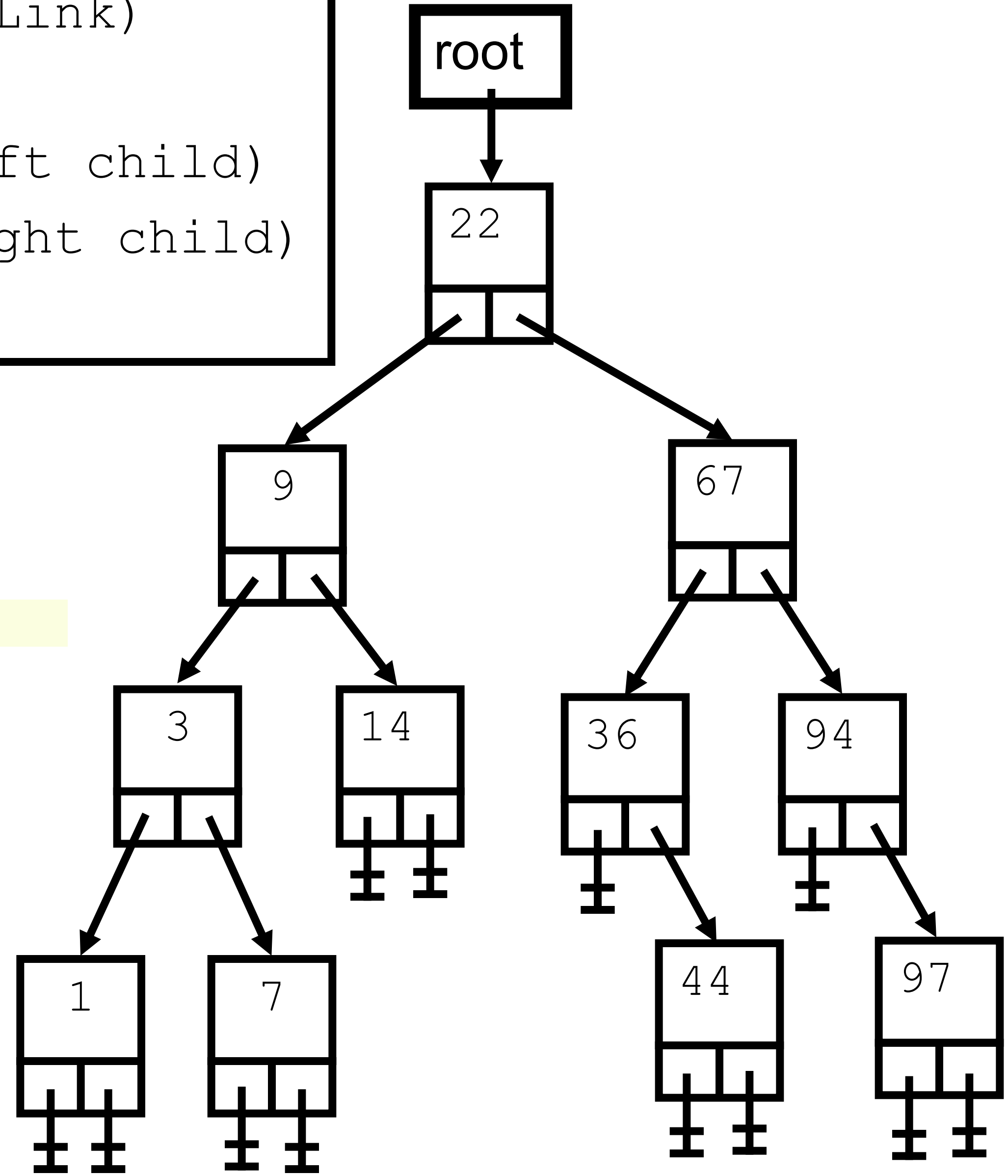
Output



```
def PostOrderPrint(Link)
 Link NOT None?
L PostOrderPrint(left child)
R PostOrderPrint(right child)
P print(data)
```

Output

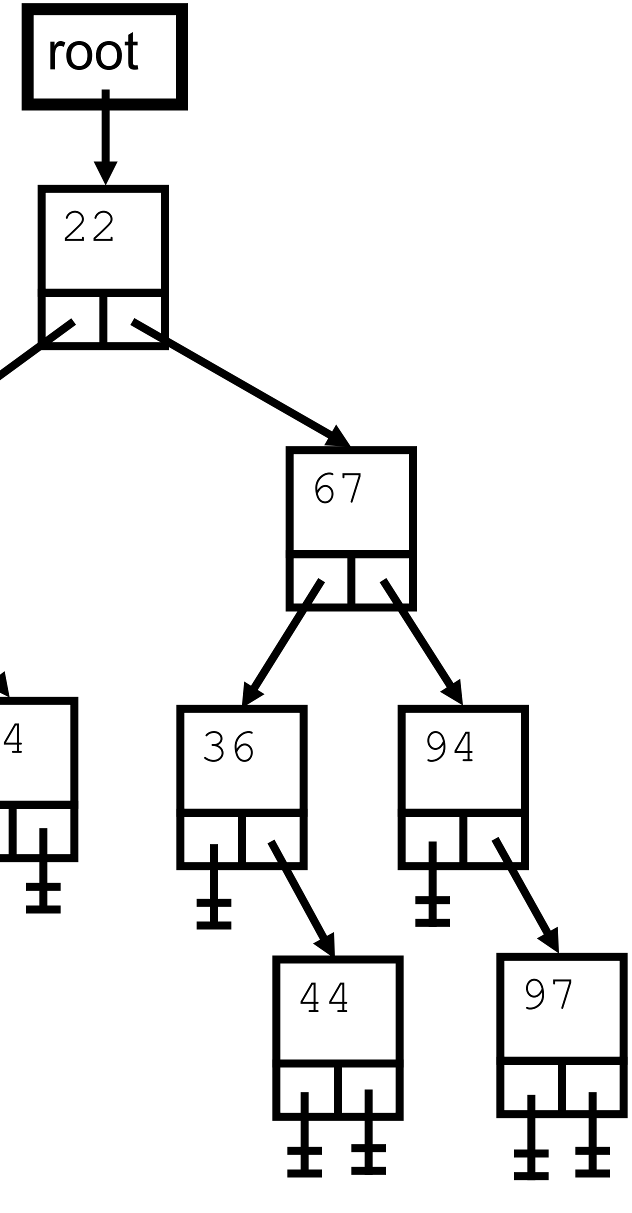
1



```
def PostOrderPrint(Link)
  Link NOT None?
L PostOrderPrint(left child)
R PostOrderPrint(right child)
P print(data)
```

Output

1, 7, 3, 14, 9, 44, 36, 97, 94, 67, 22



Traversing a Binary Search Tree (BST)

In-Order

Outline of In-Order Traversal

- **For each node**
  - **Traverse Left**
  - **Do the work (Current)**
  - **Traverse Right**

In-Order Traversal Procedure (Pseudocode)

```
def In_Order(Link):
    if Link != None:
        In_Order( Link.left_child )
        Do_Whatever( Link.data )
        In_Order( Link.right_child )
```

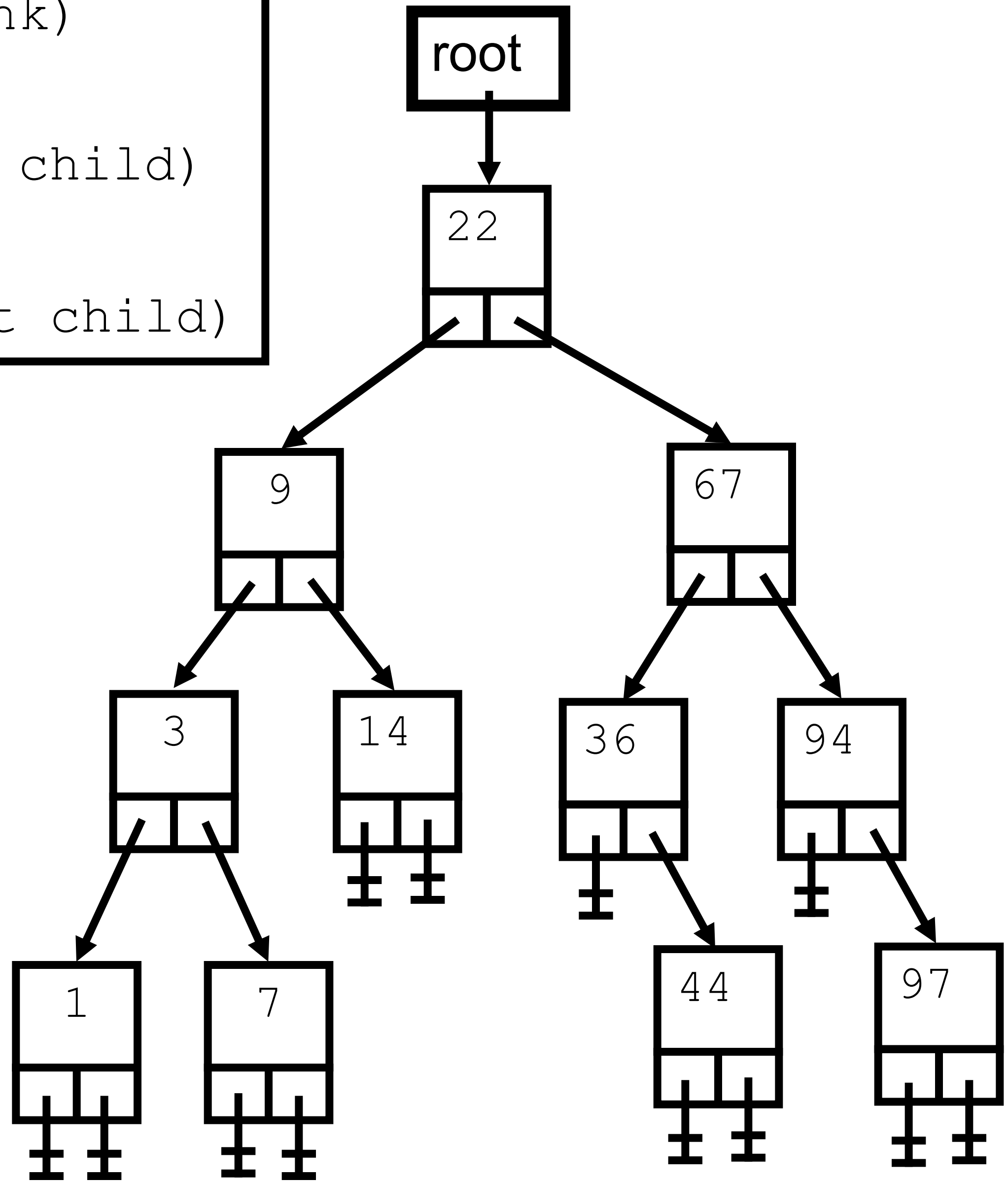
Start at the “Root” for a full tree traversal

Two recursive calls

E.g., print

```
def InOrderPrint(Link)
 Link NOT None?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

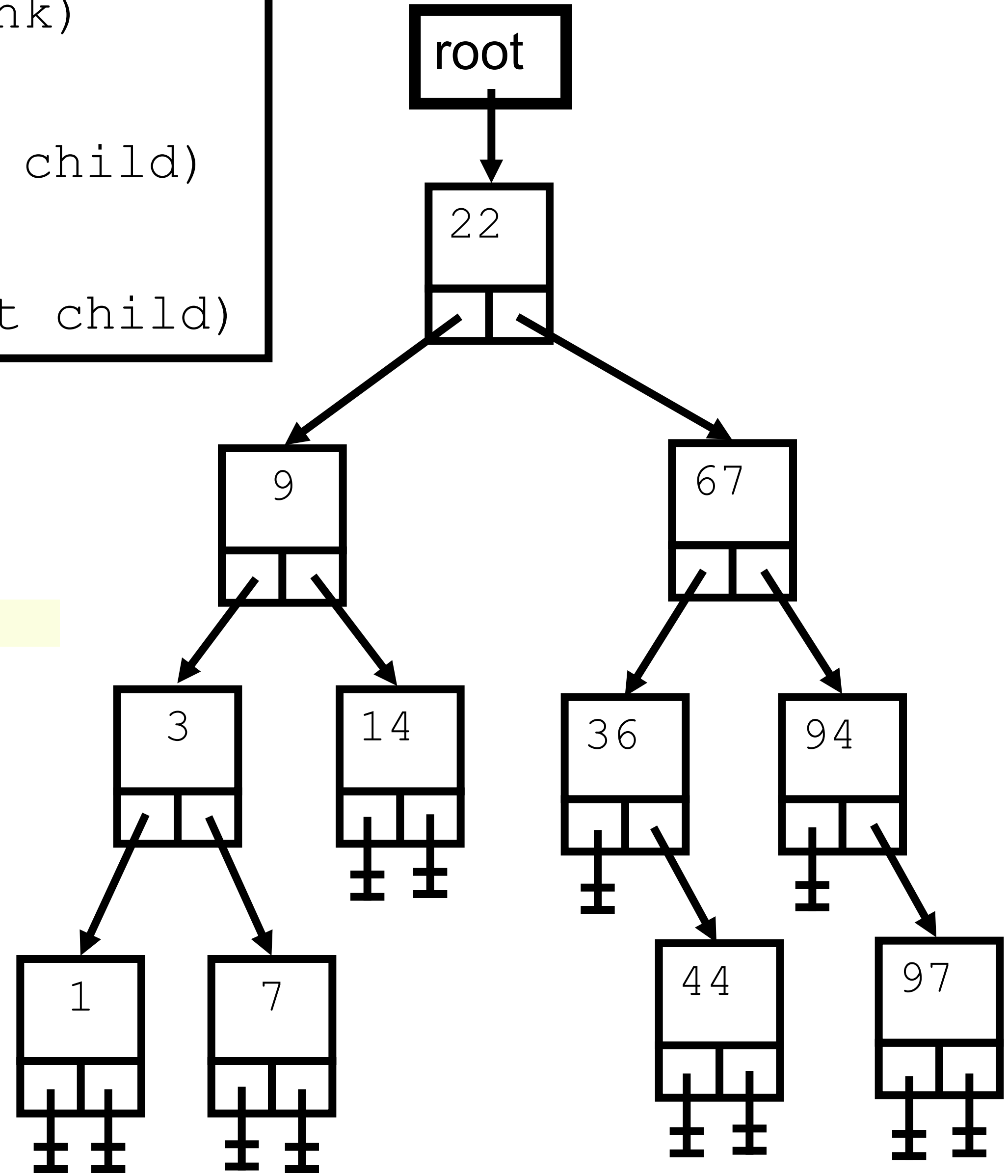
Output



```
def InOrderPrint(Link)
 Link NOT None?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

Output

1

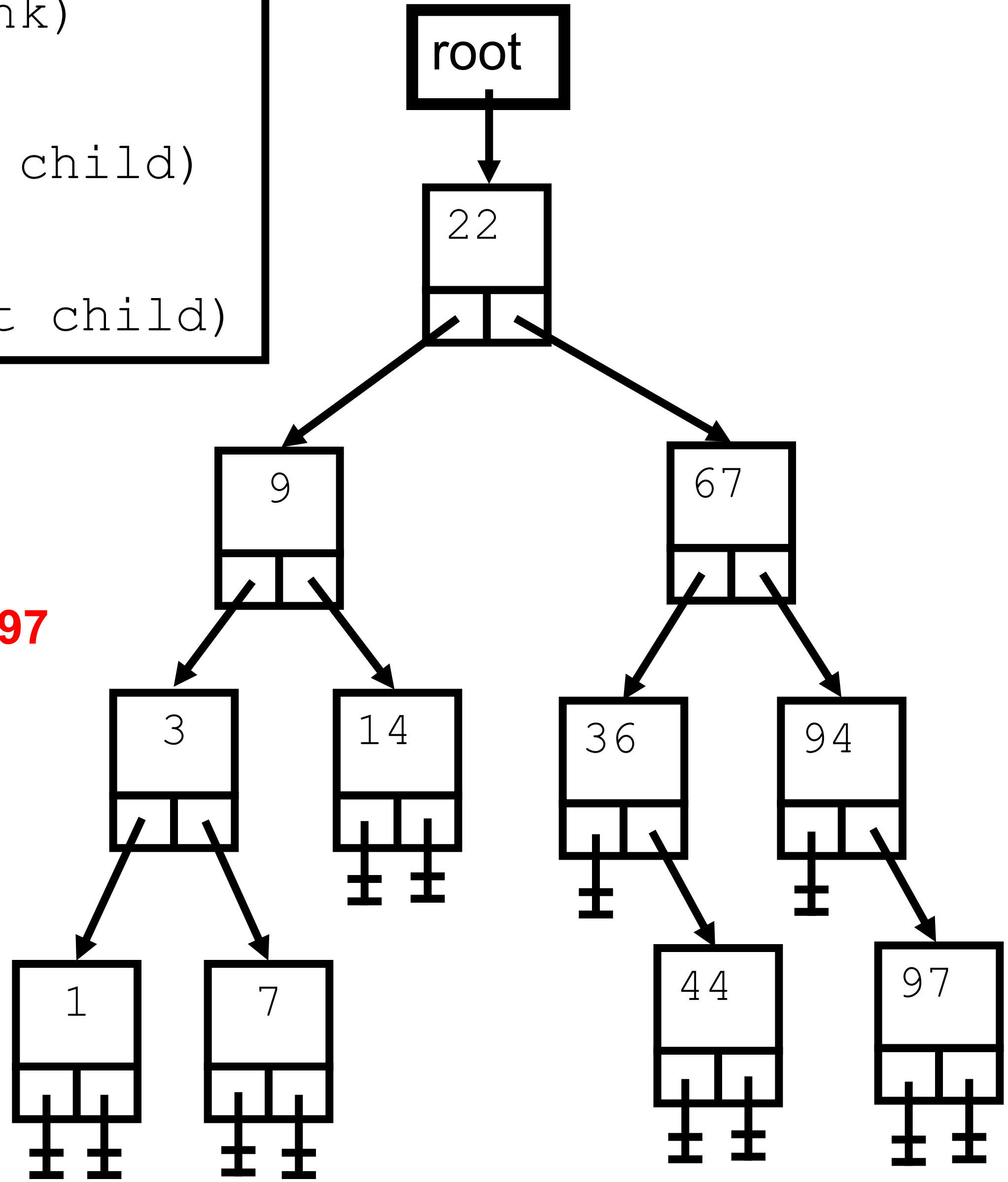


```
def InOrderPrint(Link)
 Link NOT None?
L InOrderPrint(left child)
P print(data)
R InOrderPrint(right child)
```

Output

1, 3, 7, 9, 14, 22, 36, 44, 67, 94, 97

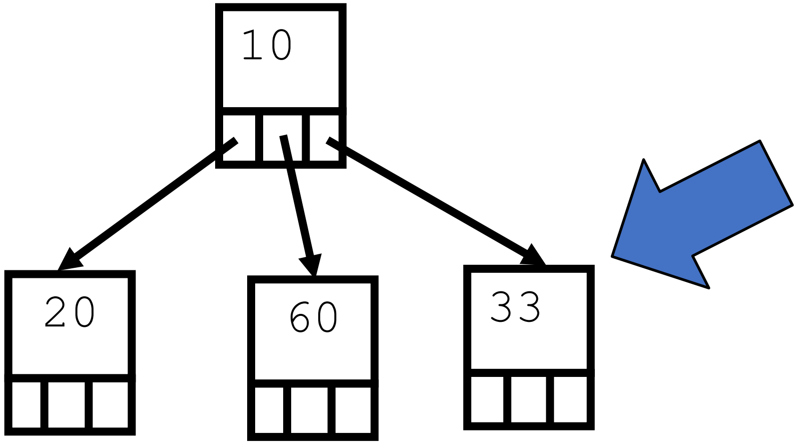
In-order traversal produces a sorted list



Notes

- **The three traversal types (Pre-, In-, Post-)Order apply to all types of binary trees**
  - **Whether binary search tree (BST) or not**

- **In-Order traversal is applicable only for binary trees**
  - **Not for N-way trees, where N > 2**



- **3-way tree**
- **Pre-Order and Post-Order traversal are applicable**
- **In-Order traversal is not applicable**

Analysis of Tree Traversal

- **Assume the tree has n nodes**

- **Any the three types of traversal is linear O(n)**

- **Each node is visited only once (E.g., printing is done once)**

Searching in a Tree: A few Observations

- Accessing a item from a linked list takes O(N) time for an arbitrary element

- Binary Search Trees can improve upon this:
  - reduce access to O(lg N) time for the average case
  - expands on the binary search technique (from Oct 24) and allows insertions and deletions

Searching in a Tree: Problem Definition

- **Input**
  - Binary tree
  - Value k to search for

Algorithm depends on whether it is binary search tree (BST) or not

Searching Binary Trees (Not BST)

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else:
         Tree-Search(Link.left, k)
         Tree-Search(Link.right, k)
```

Pseudocode

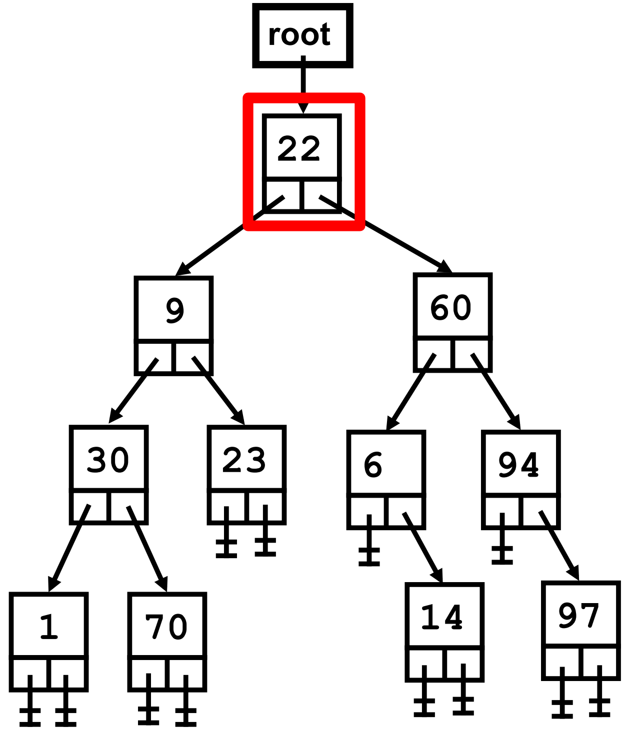
If the current node contains k, return true

Call the left and right subtrees recursively (Will check the LEFT subtree first)

Example: Search For 6

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else:
            Tree-Search(Link.left, k)
            Tree-Search(Link.right, k)
```

Pseudocode



Example: Search For 6

Tree-Search(Link to root, value k)

if Link == NULL:

return 0

else if Link.Data == k:

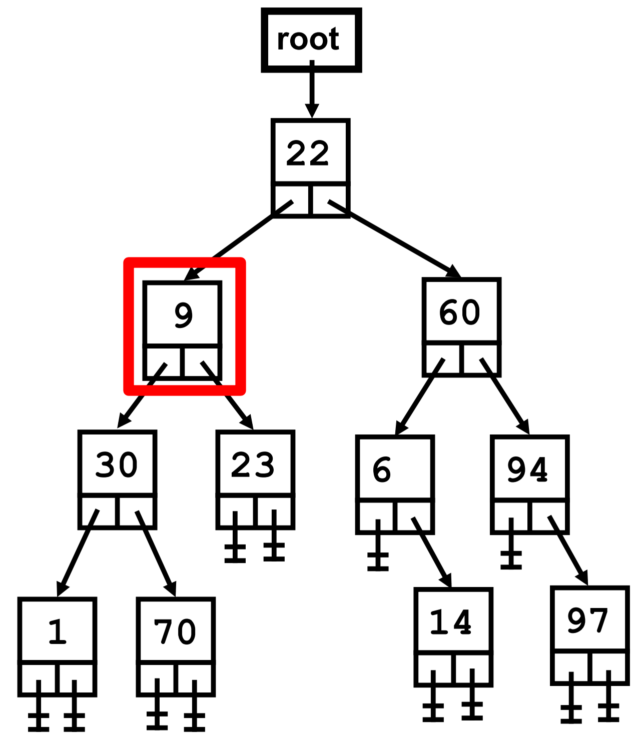
return 1;

else:

Tree-Search(Link.left, k)

Tree-Search(Link.right, k)

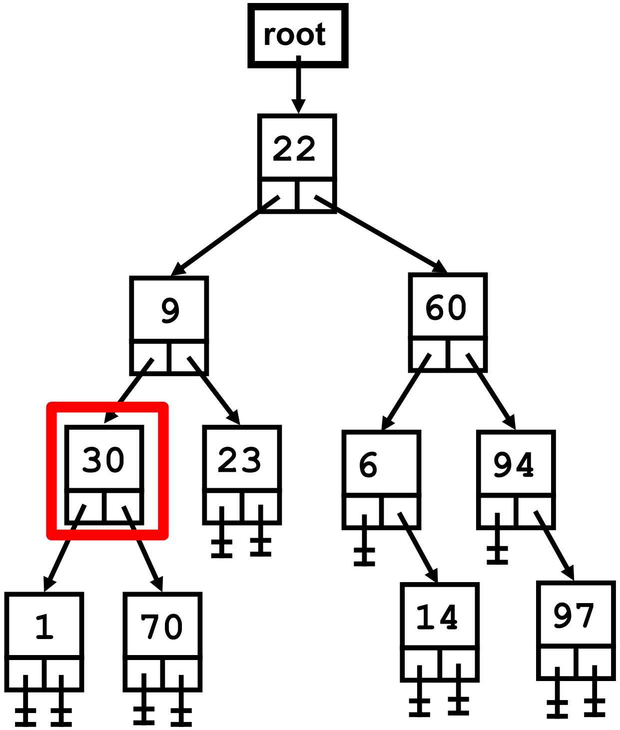
Pseudocode



Example: Search For 6

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else:
            Tree-Search(Link.left, k)
            Tree-Search(Link.right, k)
```

Pseudocode



Example: Search For 6

Tree-Search(Link to root, value k)

if Link == NULL:

return 0

else if Link.Data == k:

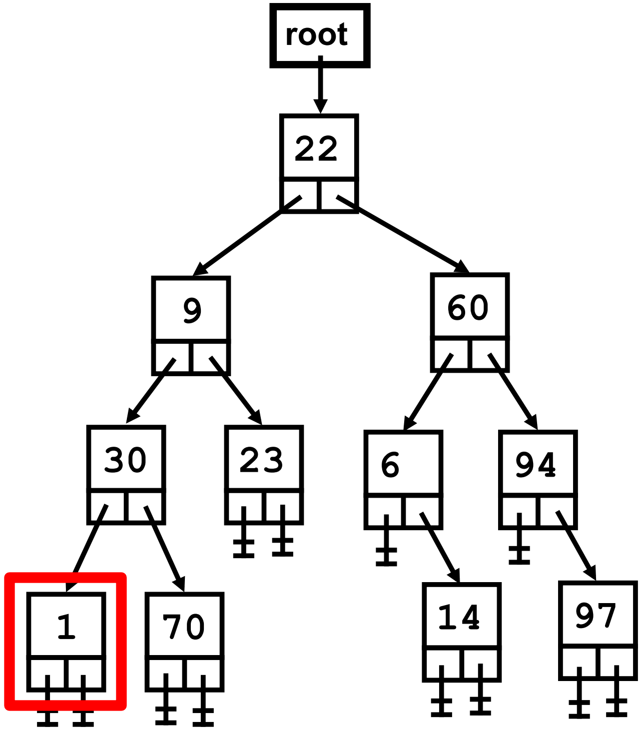
return 1;

else:

Tree-Search(Link.left, k)

Tree-Search(Link.right, k)

Pseudocode



Example: Search For 6

Tree-Search(Link to root, value k)

if Link == NULL:

return 0

else if Link.Data == k:

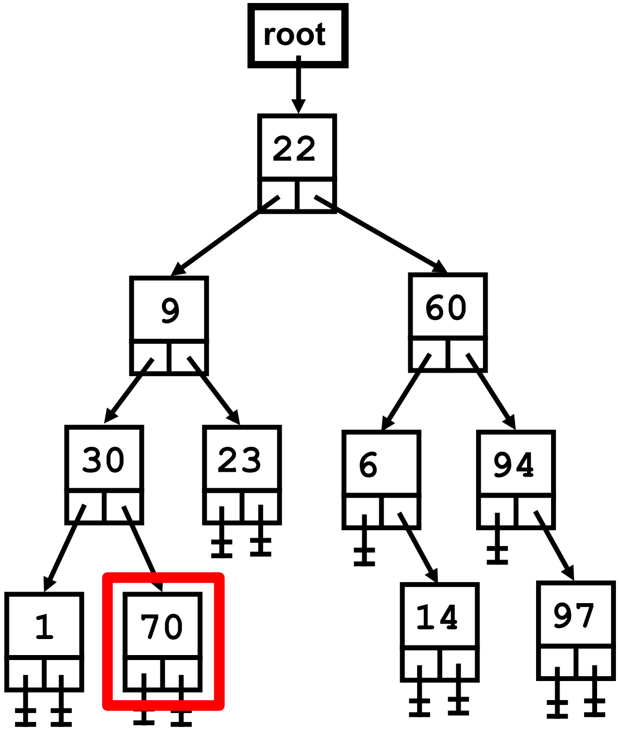
return 1;

else:

Tree-Search(Link.left, k)

Tree-Search(Link.right, k)

Pseudocode



Example: Search For 6

Tree-Search(Link to root, value k)

if Link == NULL:

return 0

else if Link.Data == k:

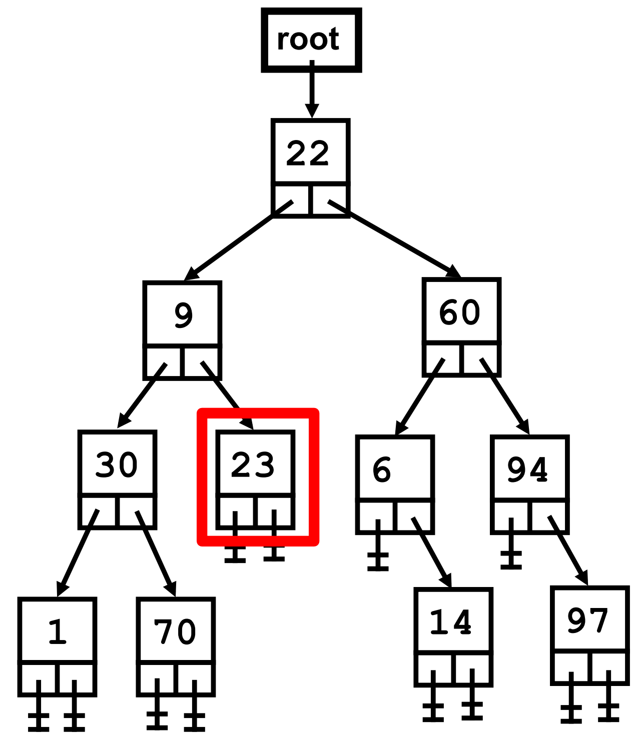
return 1;

else:

Tree-Search(Link.left, k)

Tree-Search(Link.right, k)

Pseudocode



Example: Search For 6

Tree-Search(Link to root, value k)

if Link == NULL:

return 0

else if Link.Data == k:

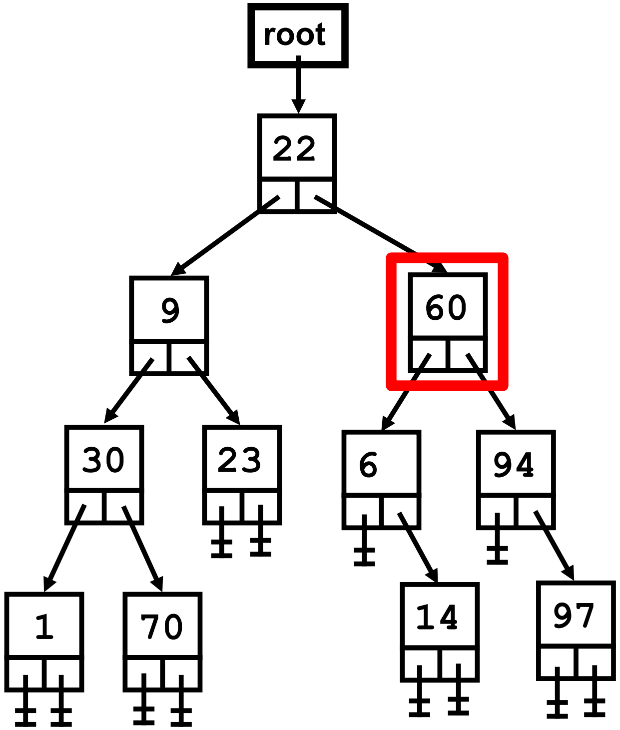
return 1;

else:

Tree-Search(Link.left, k)

Tree-Search(Link.right, k)

Pseudocode



Example: Search For 6

Tree-Search(Link to root, value k)

if Link == NULL:

return 0

else if Link.Data == k:

return 1;

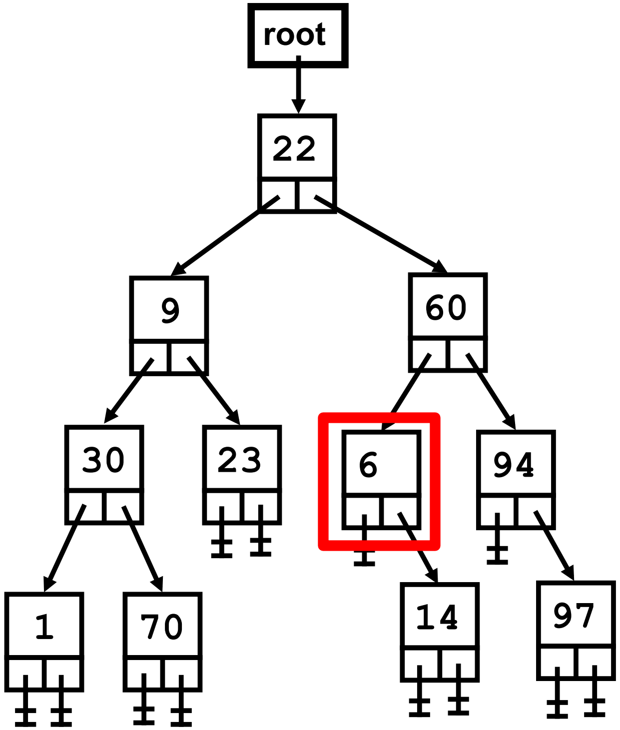
else:

Tree-Search(Link.left, k)

Tree-Search(Link.right, k)

Pseudocode

Similar to Pre-Order traversal
- Check the node first
- Check the Left subtree
- Check the Right subtree



Time Complexity For Binary Tree (Not BST) Searching

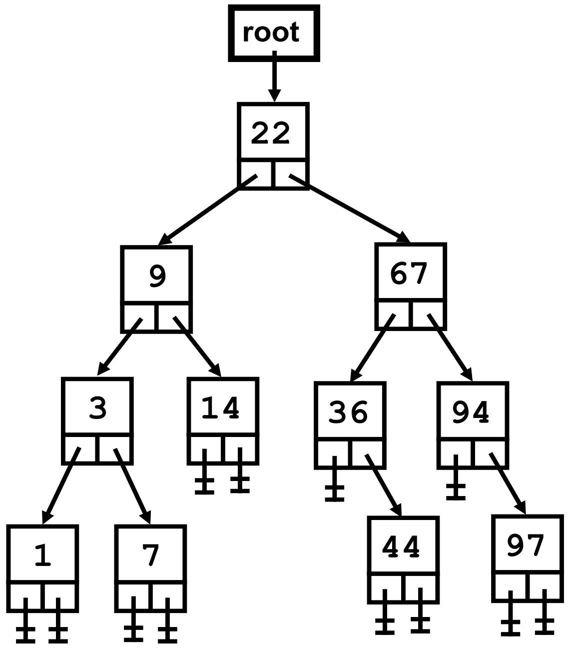
```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else:
            Tree-Search(Link.left, k)
            Tree-Search(Link.right, k)
```

- **How many times each node is checked?**
  - **At most once**
- **Searching a binary tree is linear → O(n)**

Find k in Binary Search Trees (BST)

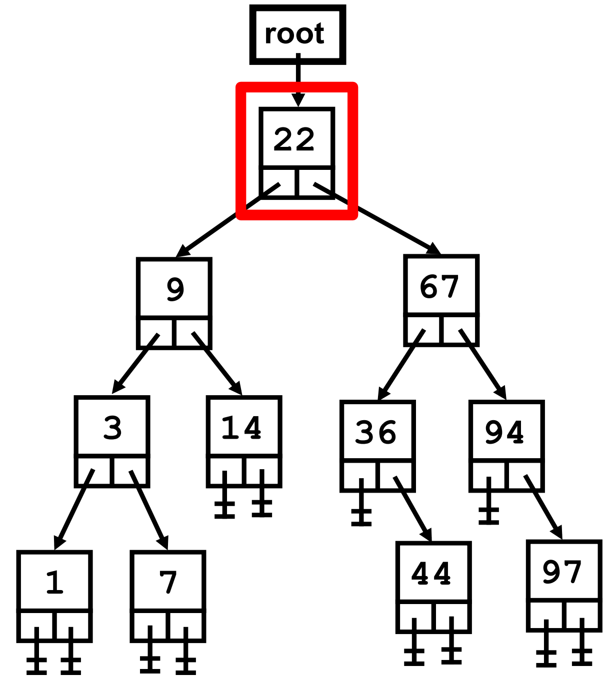
- **Should make use of the BST property to search in an efficient way**

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else if k < Link.Data
            return Tree-Search(Link.left, k)
        else:
            return Tree-Search(Link.right, k)
```



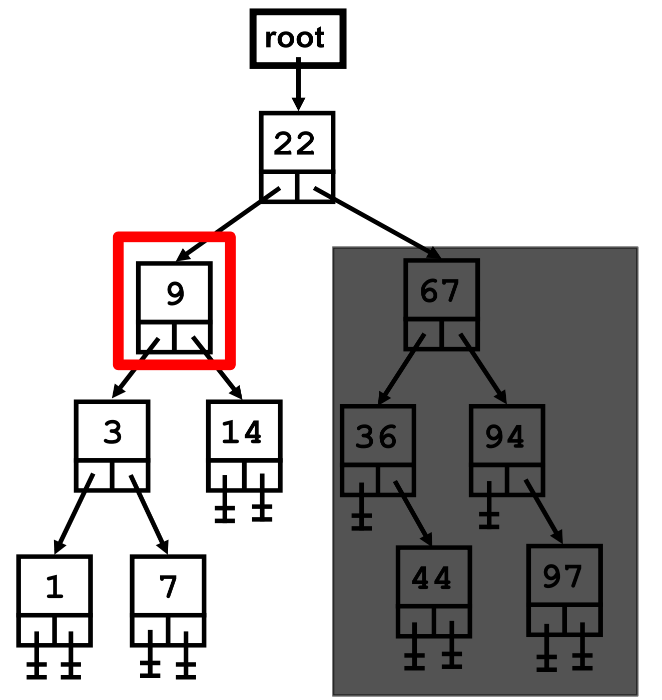
Example: Search For 6

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else if k < Link.Data
            return Tree-Search(Link.left, k)
        else:
            return Tree-Search(Link.right, k)
```



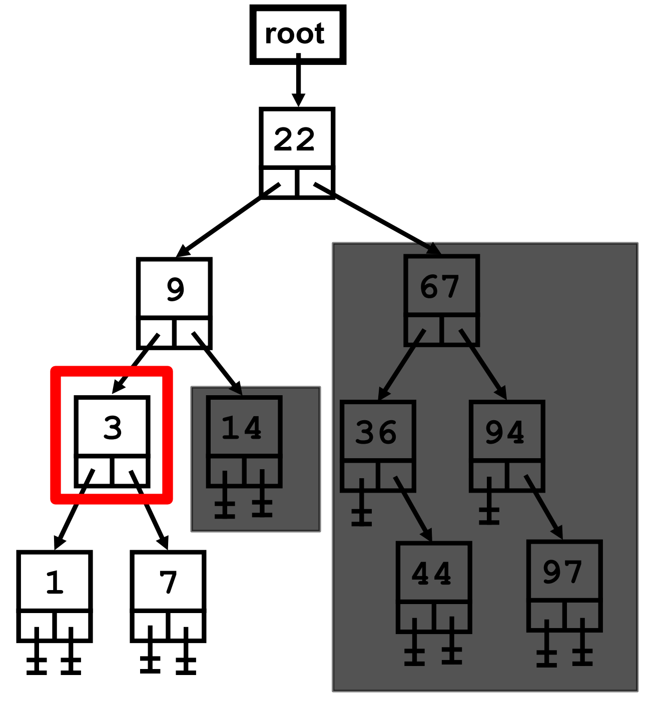
Example: Search For 6

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else if k < Link.Data
                return Tree-Search(Link.left, k)
        else:
                return Tree-Search(Link.right, k)
```



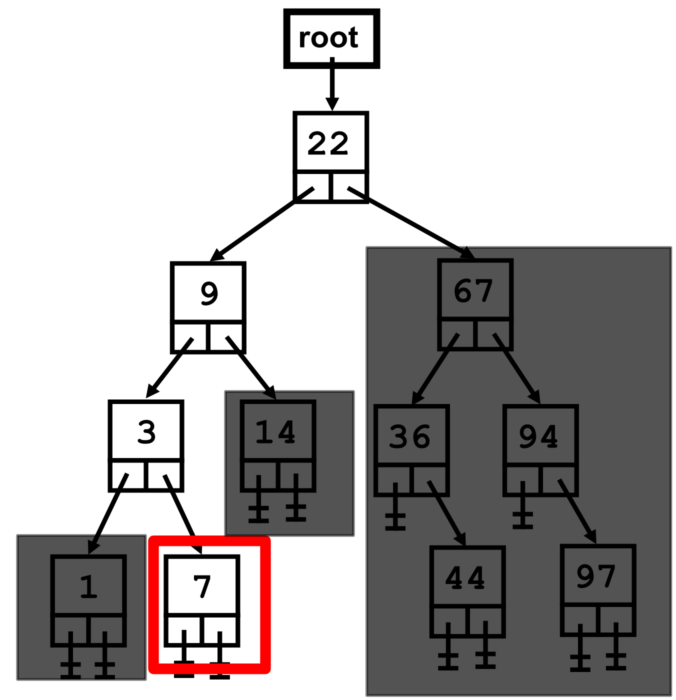
Example: Search For 6

```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else if k < Link.Data
            return Tree-Search(Link.left, k)
        else:
            return Tree-Search(Link.right, k)
```



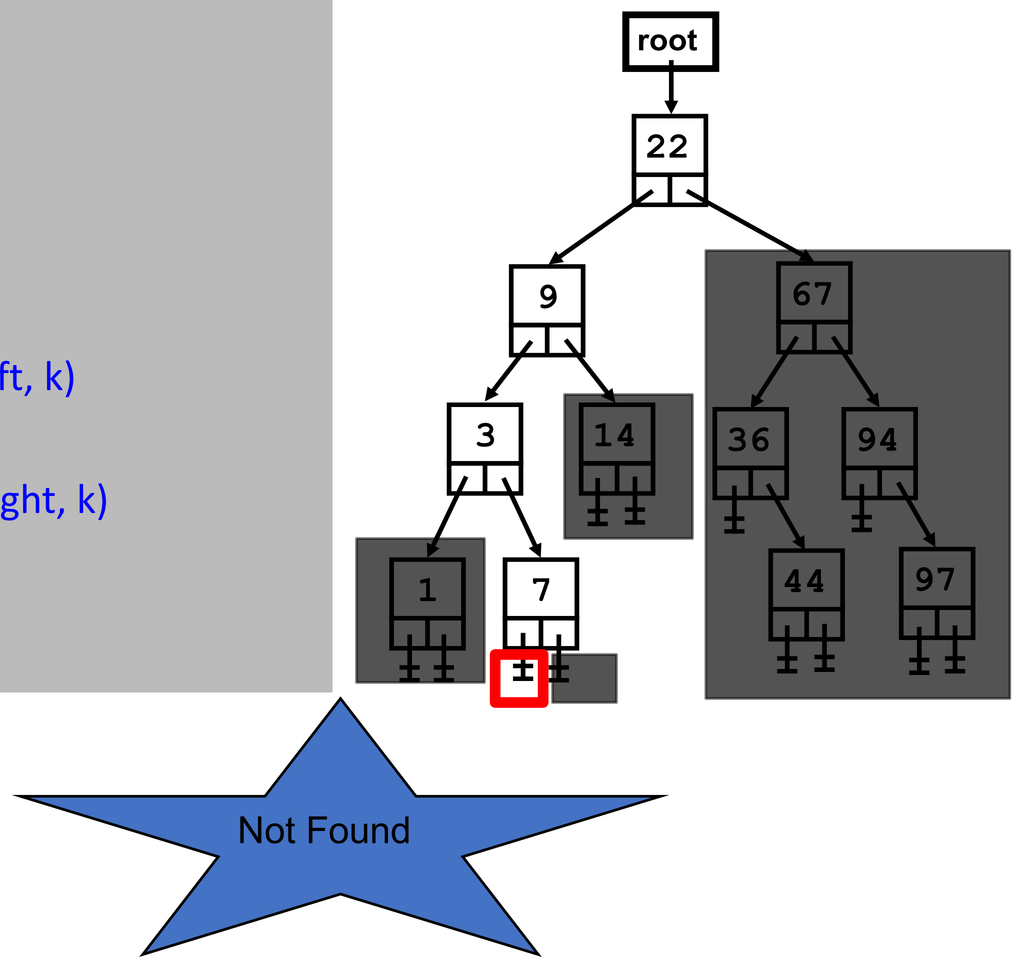
Example: Search For 6

```
Tree-Search(Link to root, value k)
        if Link == NULL:
          return 0
        else if Link.Data == k:
          return 1;
        else if k < Link.Data
            return Tree-Search(Link.left, k)
        else:
            return Tree-Search(Link.right, k)
```



Example: Search For 6

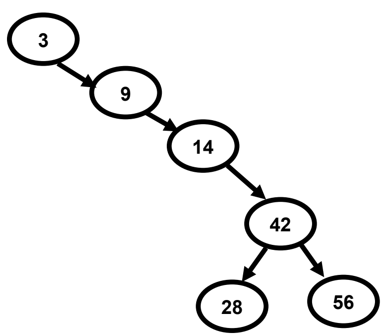
```
Tree-Search(Link to root, value k)
        if Link == NULL:
            return 0
        else if Link.Data == k:
            return 1;
        else if k < Link.Data
            return Tree-Search(Link.left, k)
        else:
            return Tree-Search(Link.right, k)
```

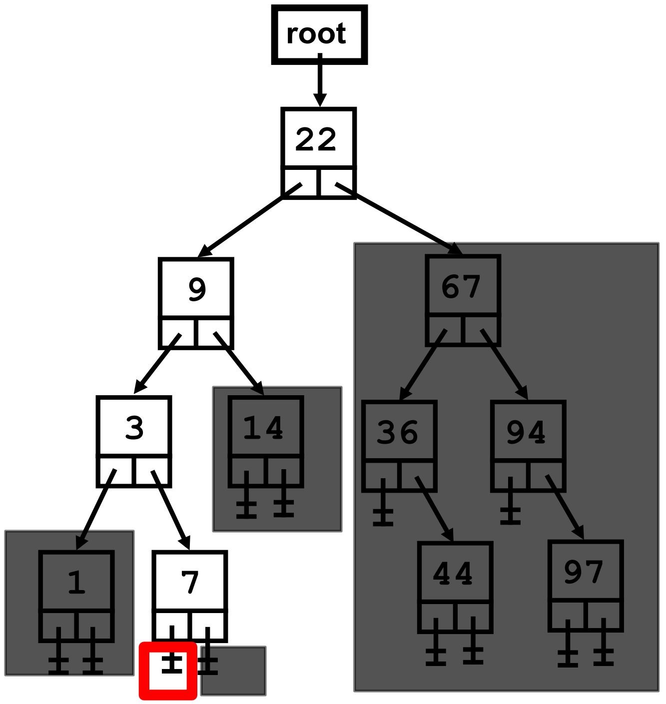


Time Complexity For BST Searching

- **In BST search we follow ONLY on path from root to a leaf**

- **What is the *Worst Case Complexity?***
  - **O(h), where *h* is the tree height = lg n**
  - **If BST is balanced ➔ O (lg n)**
  - **If BST is not balanced ➔ O (n)**





Insertion and Deletion in BST

- In Binary Search (from Oct 24), we needed to sort the an array of elements first before we could search in O(lg n) time.
  - Otherwise, one has to search every element in the array and that has O(n) time-complexity

- With BST we can store elements in such a way that we can always get O(lgn) time-complexity for search.

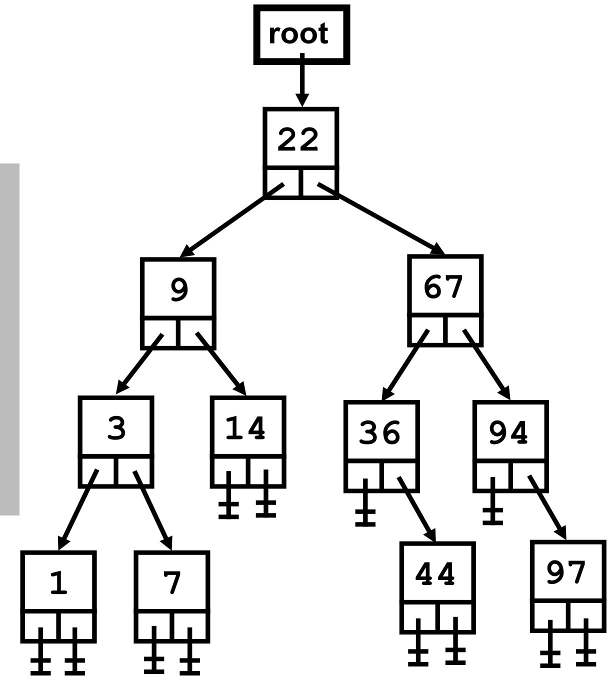
- It depends on how we insert and delete elements in the BST such that **the BST property is always maintained**

Insertion of *V* in BST

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**

    if V < current node → move left
  - **Empty: Find link with Null value**

Pseudocode

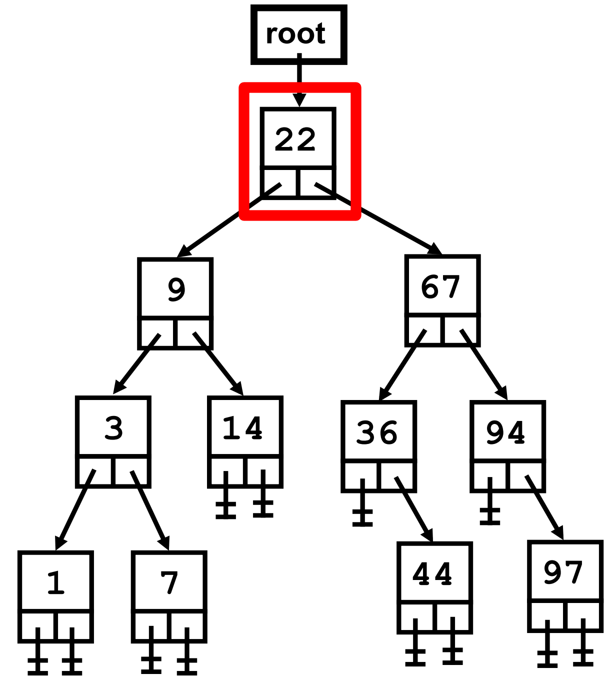


Example: Insert value 6

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**

    if V < current node → move left
  - **Empty: Find link with Null value**

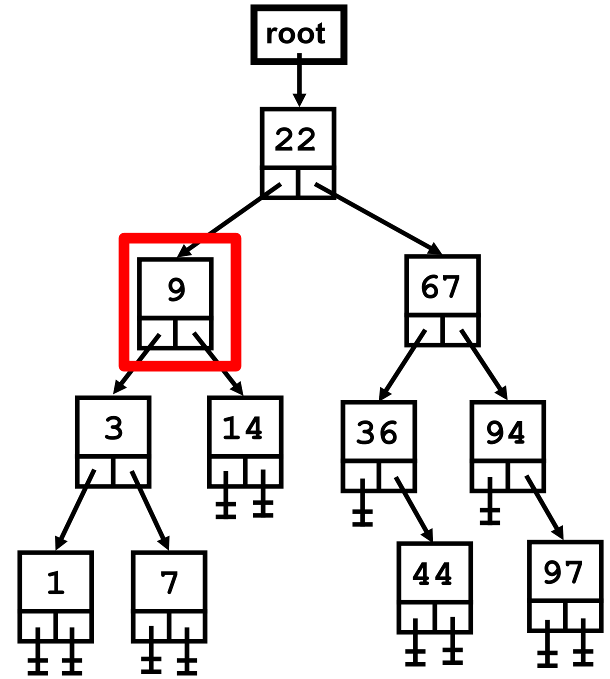
Pseudocode



Example: Insert value 6

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**
    if V < current node → move left
  - **Empty: Find link with Null value**

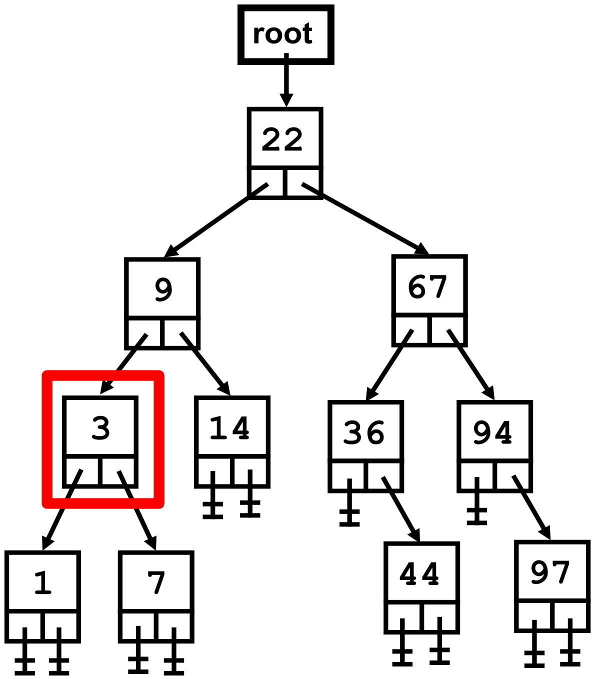
Pseudocode



Example: Insert value 6

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**
    if V < current node → move left
  - **Empty: Find link with Null value**

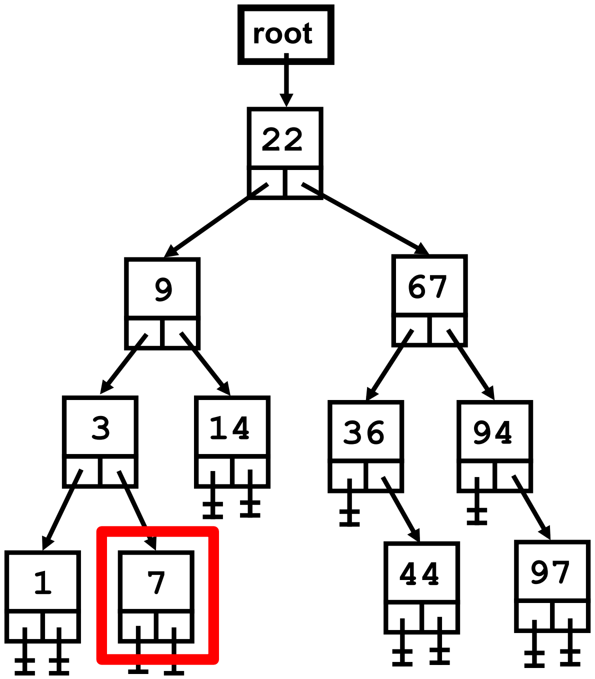
Pseudocode



Example: Insert value 6

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**
    if V < current node → move left
  - **Empty: Find link with Null value**

Pseudocode

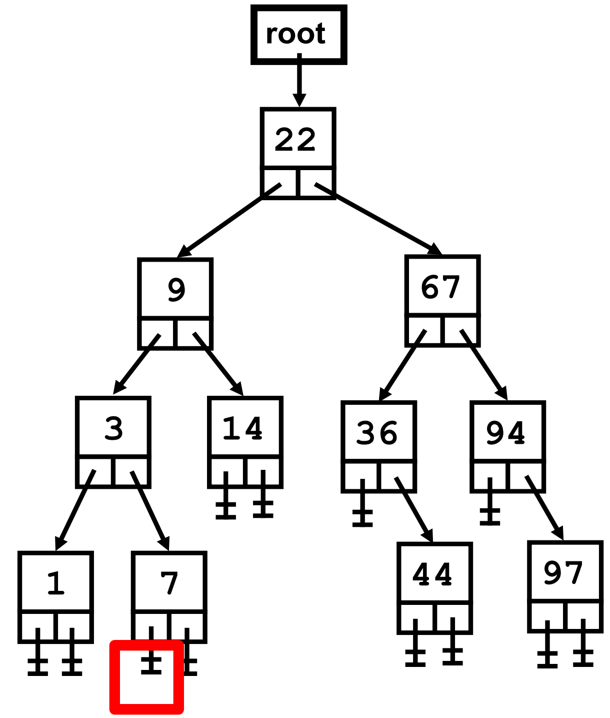


Example: Insert value 6

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**

    if V < current node → move left
  - **Empty: Find link with Null value**

Pseudocode

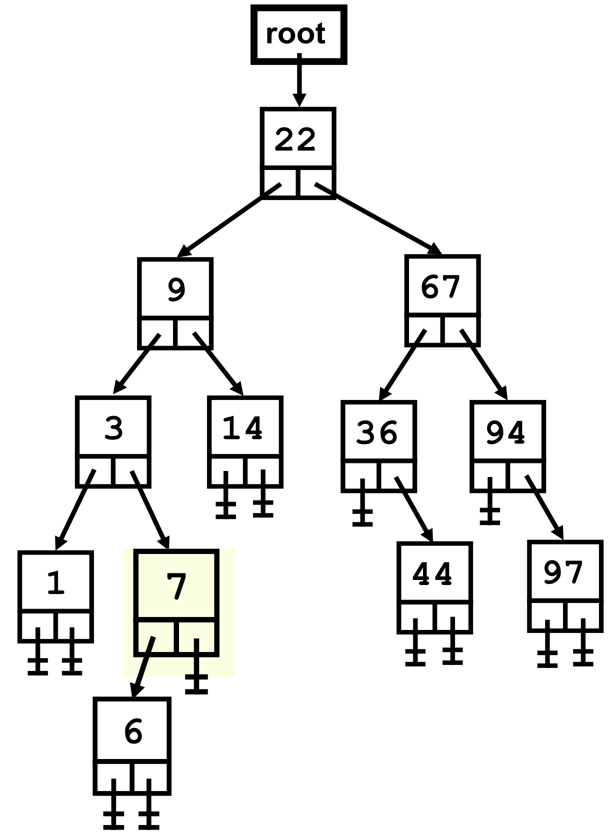


Example: Insert value 6

- **Search for the correct empty place to put the new node**
  - **Correct: if V >= current node → move right**
    if V < current node → move left
  - **Empty: Find link with Null value**

Pseudocode

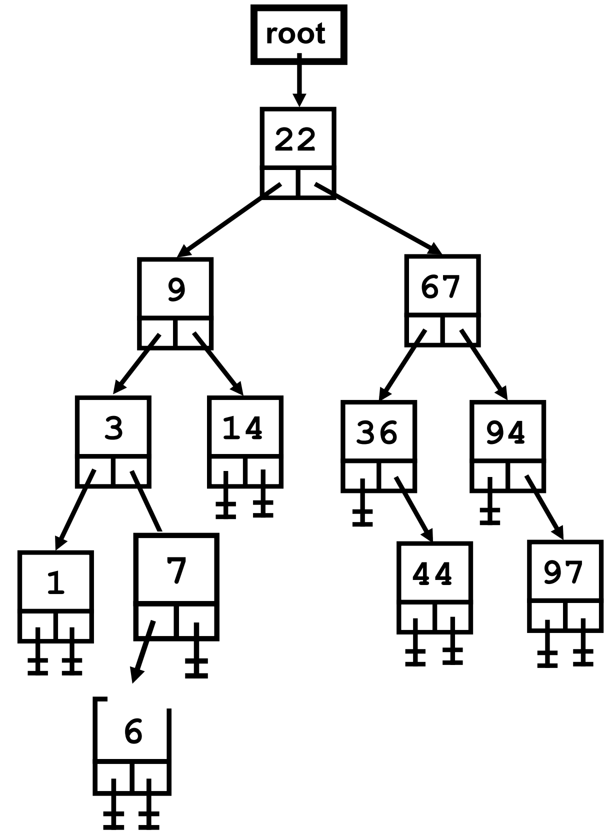
Time complexity for insertion is O(h),
h: is the tree height



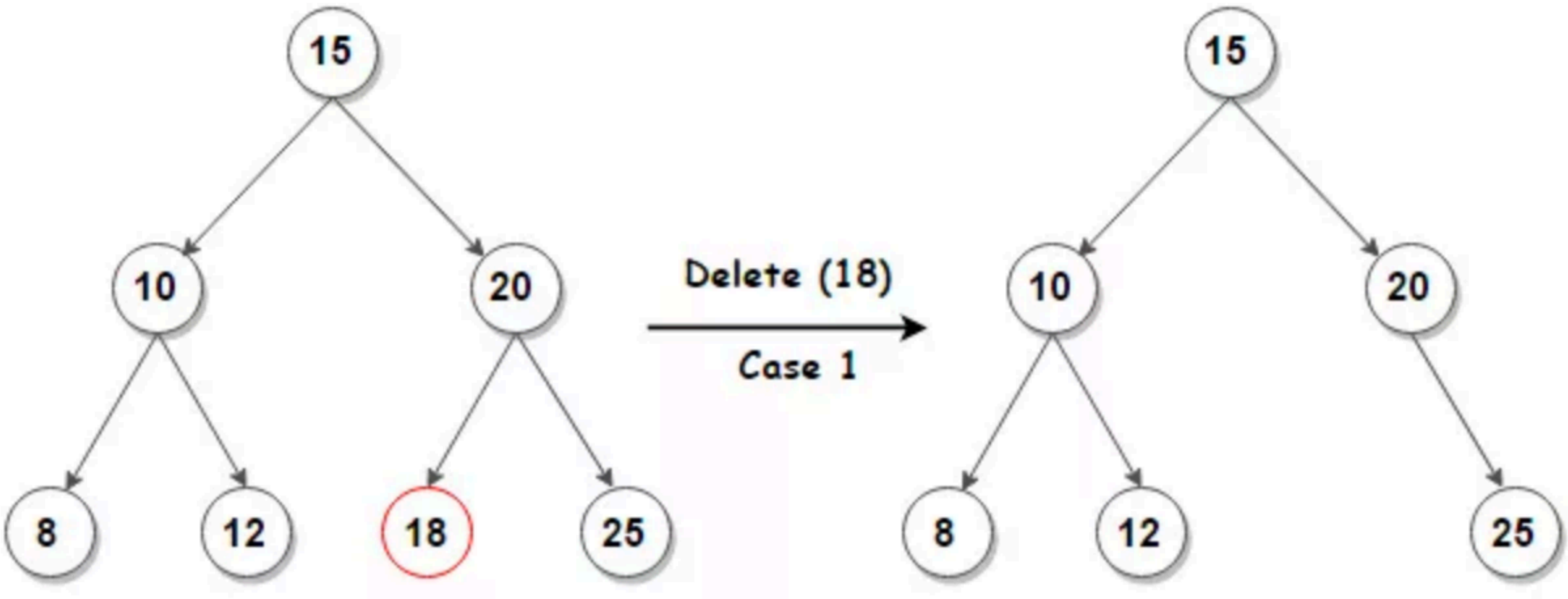
Deletion of *V* from BST

- **Search for V in the tree**
  - **Not Found** ➔ **We are done**
  - **Found**
    - **Case 1: Node V has no children**
      - **Delete V**
    - **Case 2: Node V has one child**
      - **The child takes the place of V**
    - **Case 3: Node V has two children**
      - **Find the <u>predecessor</u> node of V → Say X**
      - **Put X in the place of V**

How to find this value (Predecessor of V)??

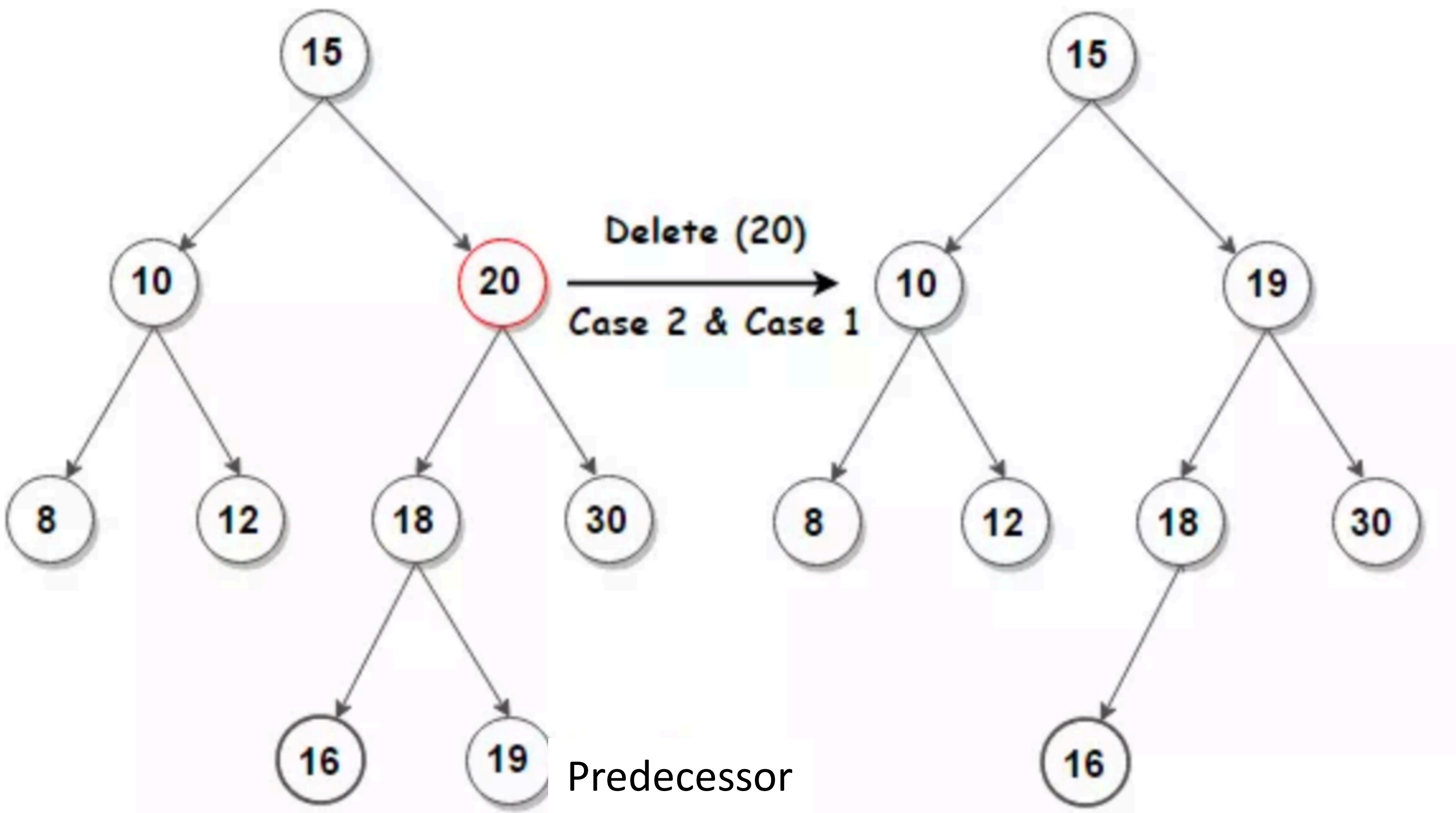


Example of BST Deletion:



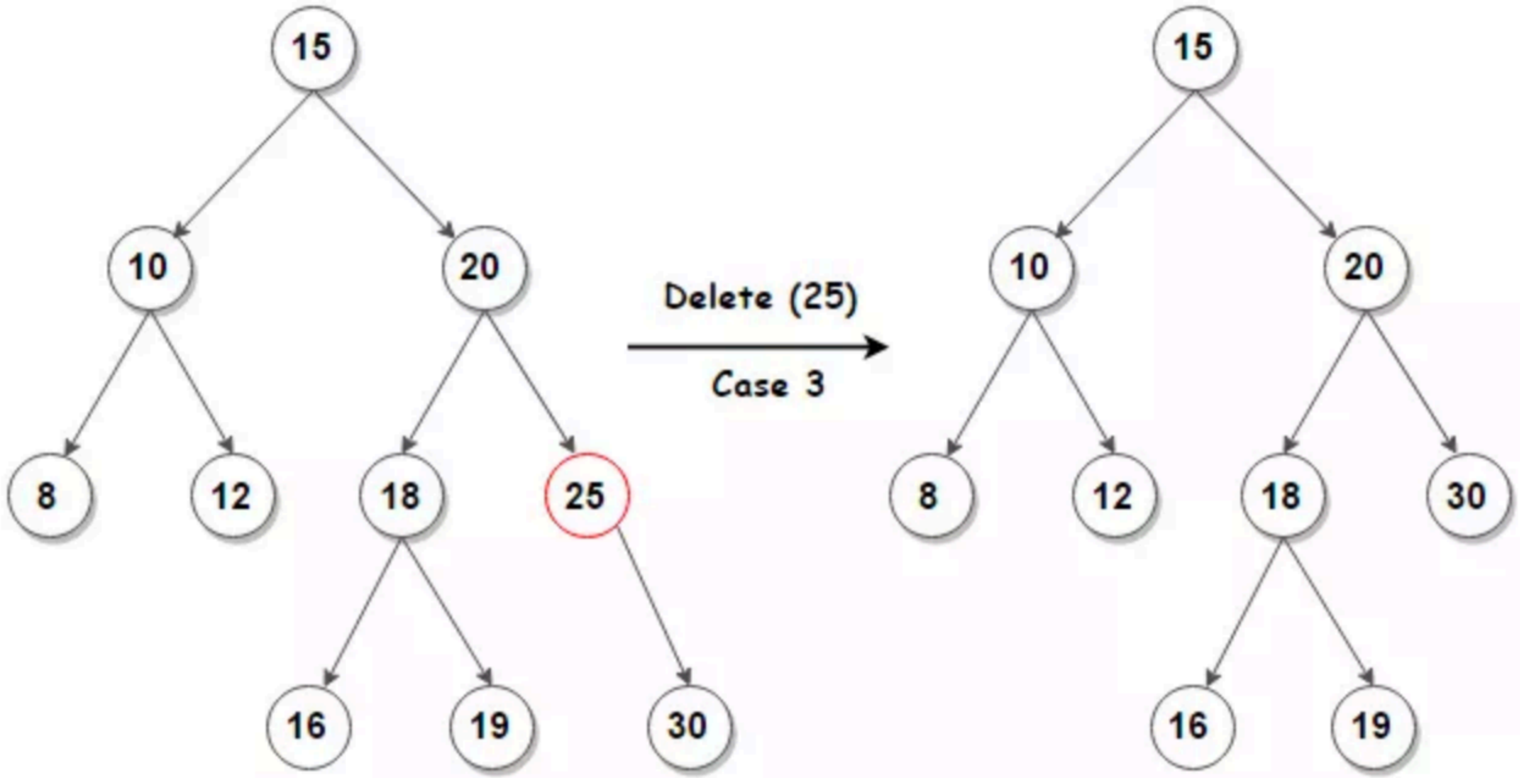
Courtesy: https://www.techiedelight.com/deletion-from-bst/

Example of BST Deletion:



Courtesy: https://www.techiedelight.com/deletion-from-bst/

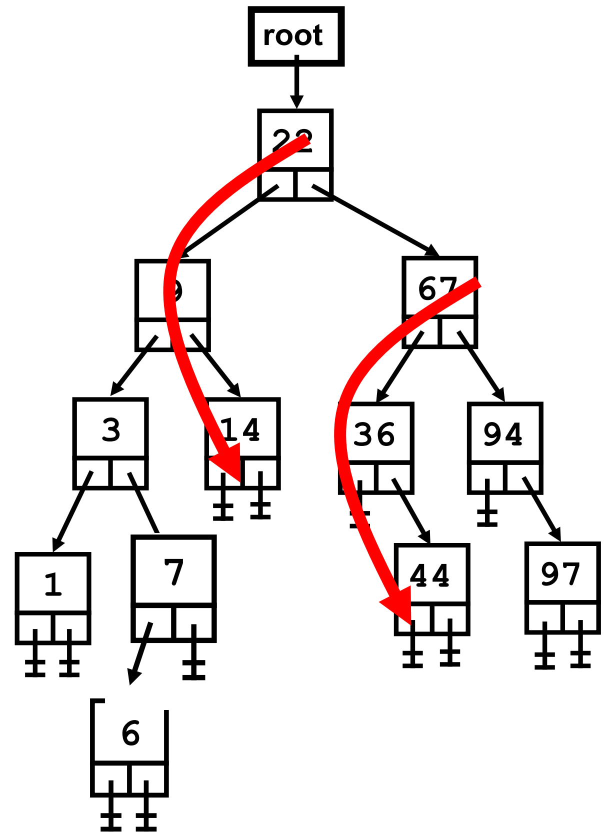
Example of BST Deletion:



Courtesy: https://www.techiedelight.com/deletion-from-bst/

Find the Predecessor Of Given Node

- **Predecessor of V: is the smallest value larger than V**
- **How to find it**
  - **Go to the LEFT child node**
  - **Follow the RIGHT link all the way (until None)**
- **Predecessor of 22**
  - **14**
- **Predecessor of 67**
  - **44**



Time complexity for finding the predecessor is O(h), h is the tree height

Priority Queues

Heaps (RECAP)

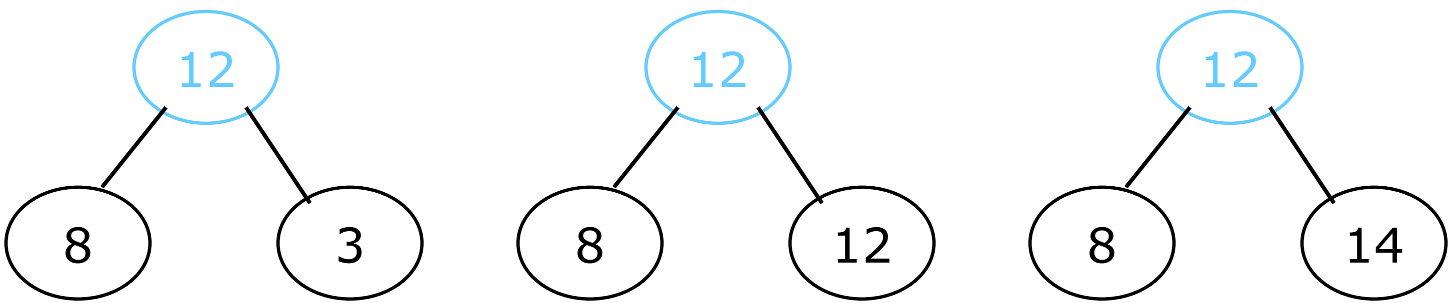
- A Binary Tree with a specific property like BST – but different, of course.

- Heaps (MAX-HEAPS or MIN-HEAPS) were used as mental model for sorting in HeapSort

- The MAX-HEAPS/MIN-HEAPS can also be created in memory!

The MAX-Heap property (RECAP)

- A node has the **MAX-Heap property** if the **value in the node is >= the values in its children**



Blue node <u>has</u> MAX-heap property

Blue node <u>has</u> MAX-heap property

Blue node <u>does not have</u> MAX-heap property

- All **leaf nodes automatically have the MAX-Heap property**
- A **binary tree is a MAX-heap if *all* nodes in it have the MAX-heap property**
- <u>You can similarly have MIN-heap(with the smallest element as parent)</u>

MAX-Heap and Min-Heaps are generally called HEAPS

Priority Queues

- A queue where we add objects, each with a value ("priority").

- Priority queues are very common for *job scheduling*

- Two Types:
  - **Max-Priority Queue** ← we use MAX-HEAPS
  - **Min-Priority Queue** ← we use MIN-HEAPS

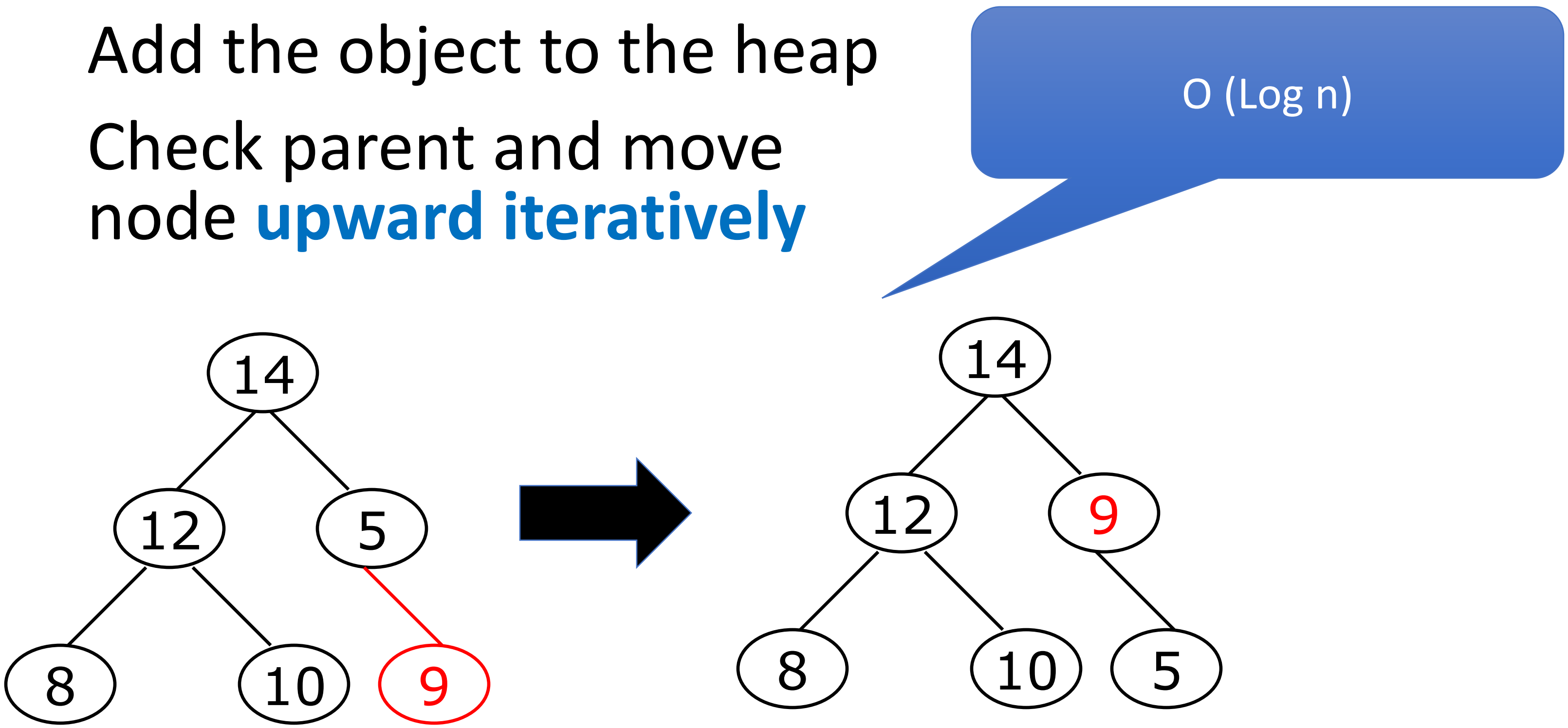
Operations on Priority Queues (Assume Max Queue)

- 1 → **Add** a **new** object with priority K
- 2→ **Return** the object with the **highest** priority
- 3 → **Remove** the object with the **highest** priority
- 4 → **Increase** the **priority** of object O

Heap data structure can implement all these operations efficiently

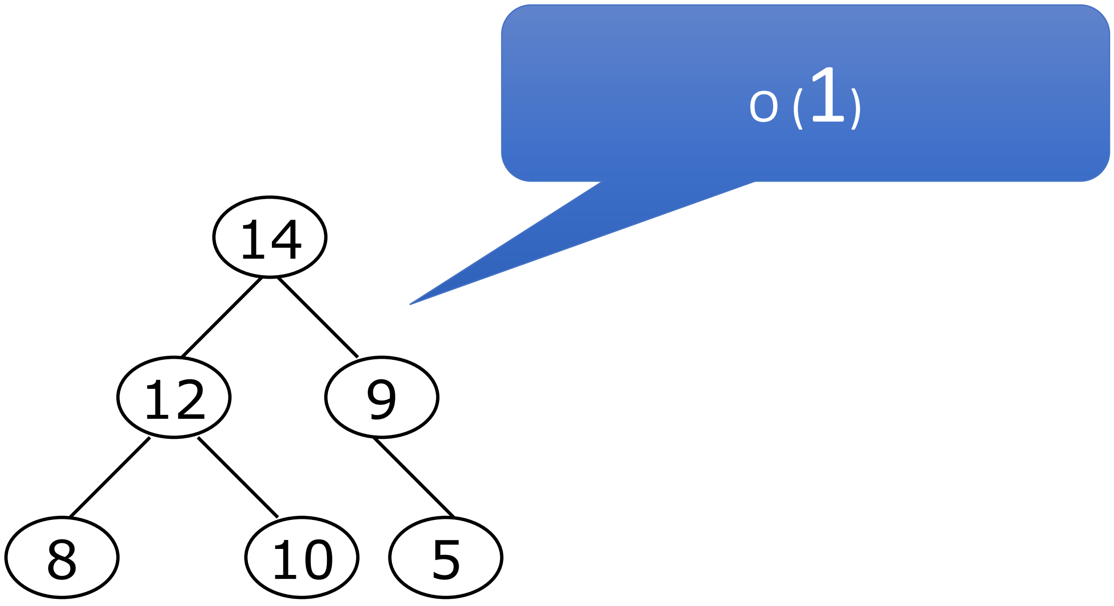
1- Add New Object With Priority 9

- Add the object to the heap
- Check parent and move node **upward iteratively**



2- Return the Highest-Priority Object

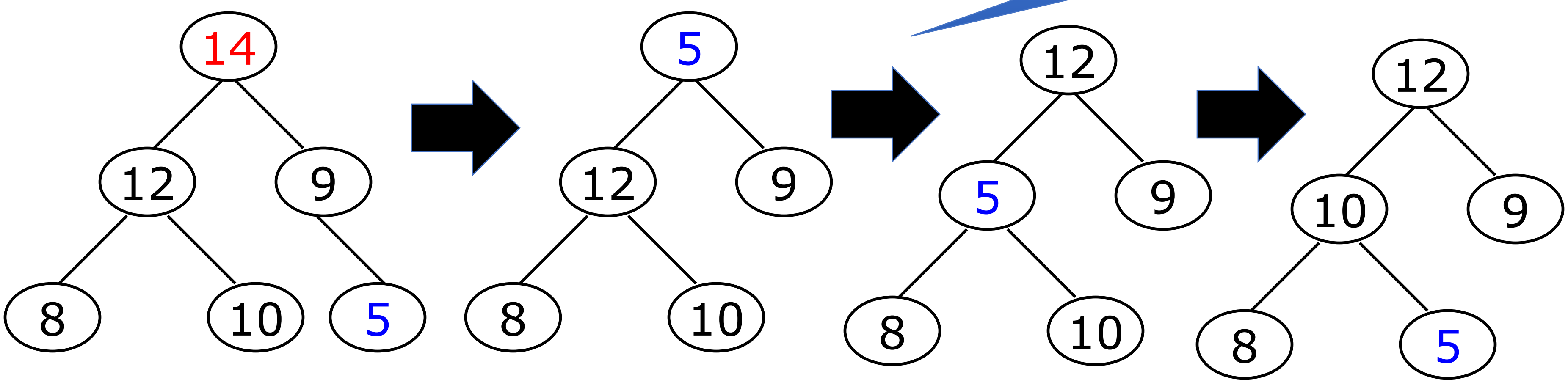
- Return the root of the tree
- Same as: Return the first element in the Heap array
- In our example, return 14



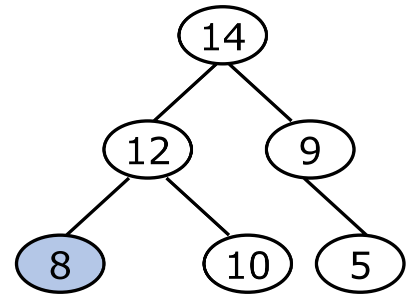
3- Remove the Highest-Priority Object

- Remove the root of the tree
- Replace it with the lowest right-most object
- MAX-Heapify

O (Log n)

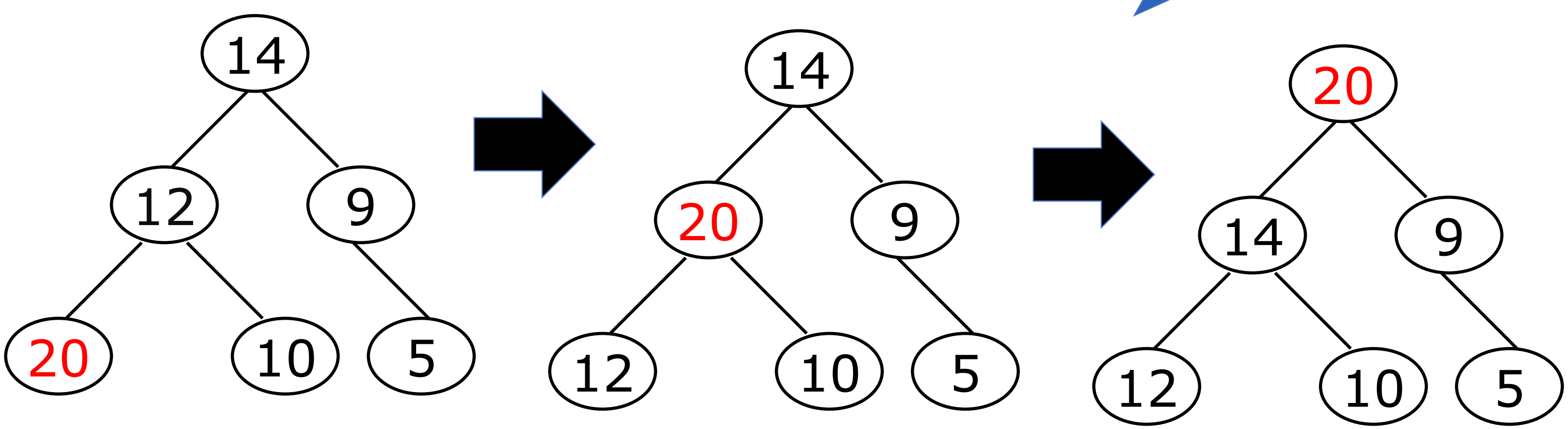


4- Increase the Priority of Object O



- Change 8 to 20
- Check parent and move node **upward iteratively**

O (Log n)



Next Class…

- Quiz 4 will be on Tuesday (Nov 12)

- Quiz 4 will cover all the materials from Nov 5 and Nov 7 (inclusive)


That's all Folks!
Any Question ?