Data Structure

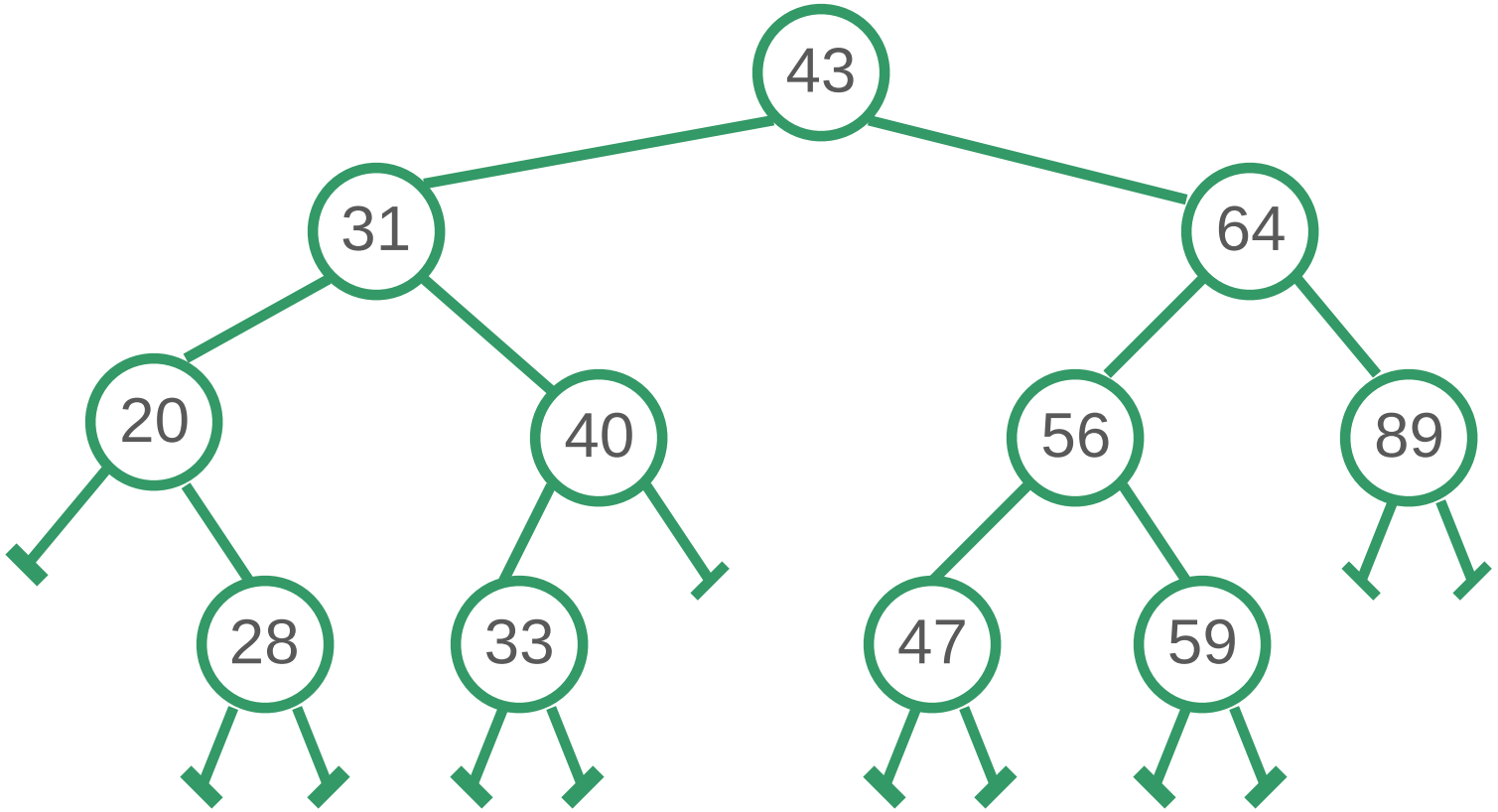
Tree Part II

Traversal

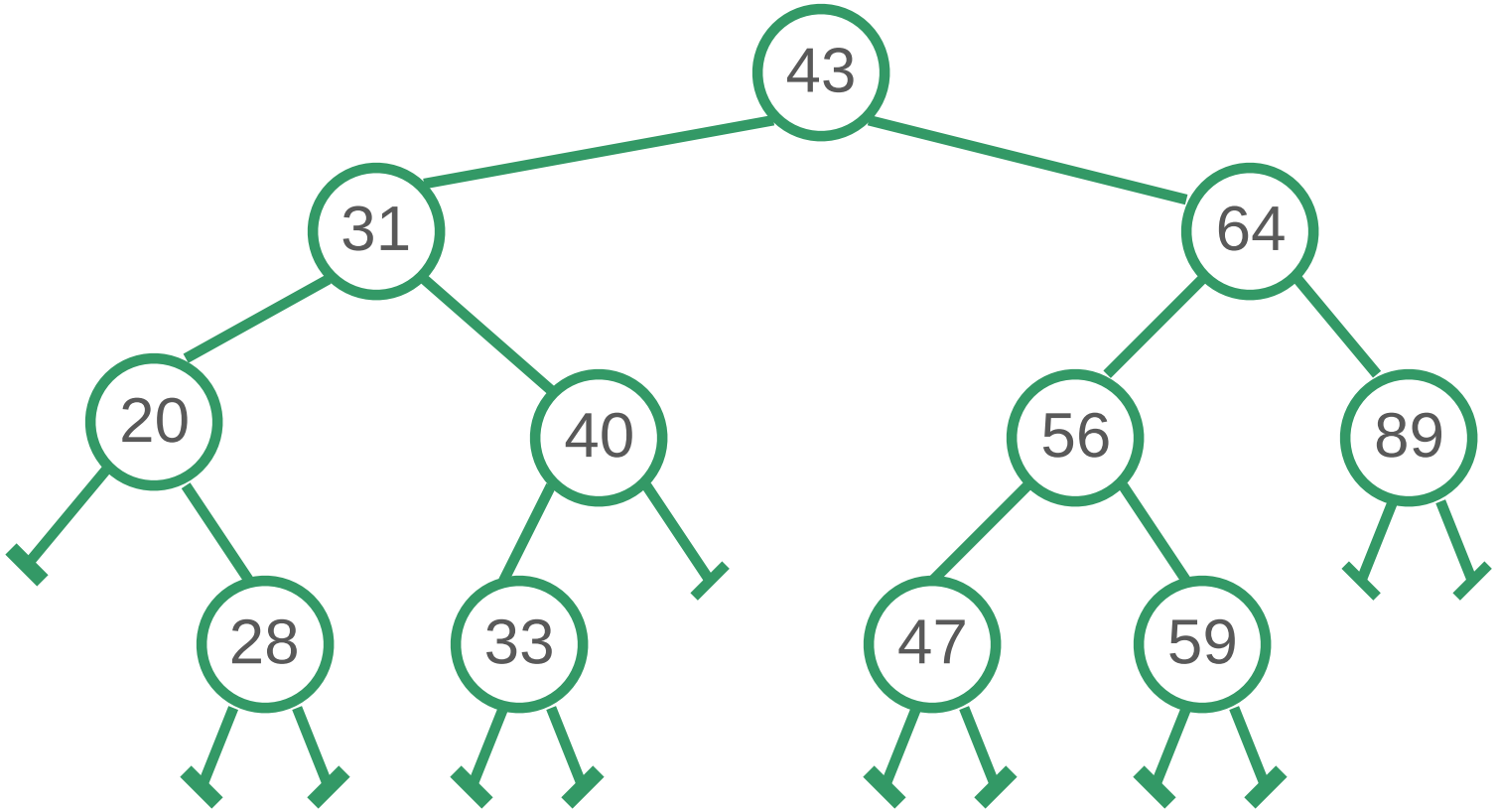
- Systematic way of **visiting**/processing **all the nodes**
- **Methods**: Preorder, Inorder, and Postorder
- They **all** traverse the left subtree before the right subtree. It's all about the **position of the root**.


Preorder
root
Left subtree
Right subtree

Example: Preorder



Example: Preorder



43	31	20	28	40	33	64	56	47	59	89

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
def print_preorder(self):
```

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
def print_preorder(self):
    self._print_preorder_aux(self.root)
```

The helper method receives a reference to the “next root”

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
```

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
    if current is not None:  # if not a base case
```

Work to do…

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
class TreeNode:
    def __init__(self, item=None, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.item)
```

```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
    if current is not None:  # if not a base case
        print(current)
```

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
    if current is not None:  # if not a base case
        print(current)
        self._print_preorder_aux(current.left)
```

Print Preorder Traversal

1) Print the **root** node
2) Traverse the **left** subtree
3) Traverse the **right** subtree

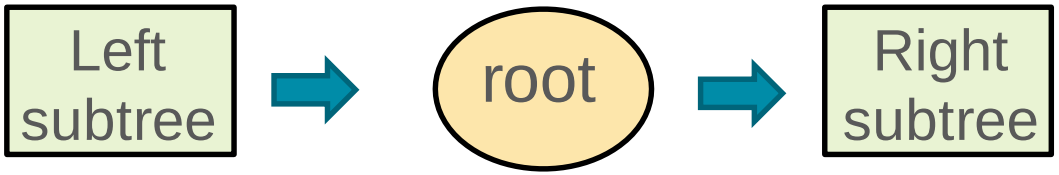
```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
    if current is not None:  # if not a base case
        print(current)
        self._print_preorder_aux(current.left)
        self._print_preorder_aux(current.right)
```

Print Preorder Traversal

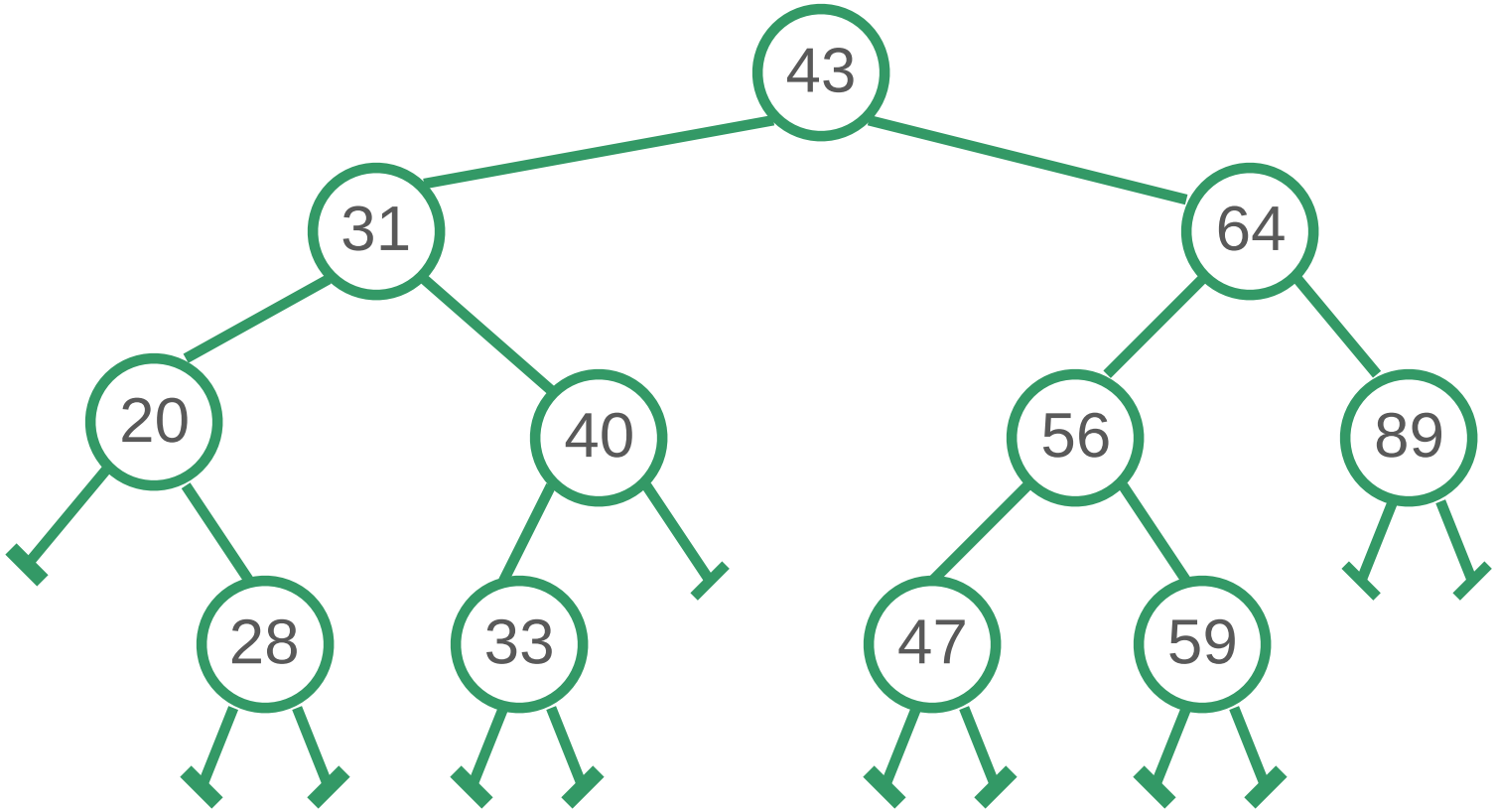
```
def print_preorder(self):
    self._print_preorder_aux(self.root)

def _print_preorder_aux(self, current):
    if current is not None:  # if not a base case
        print(current)
        self._print_preorder_aux(current.left)
        self._print_preorder_aux(current.right)
```

Print Inorder Traversal

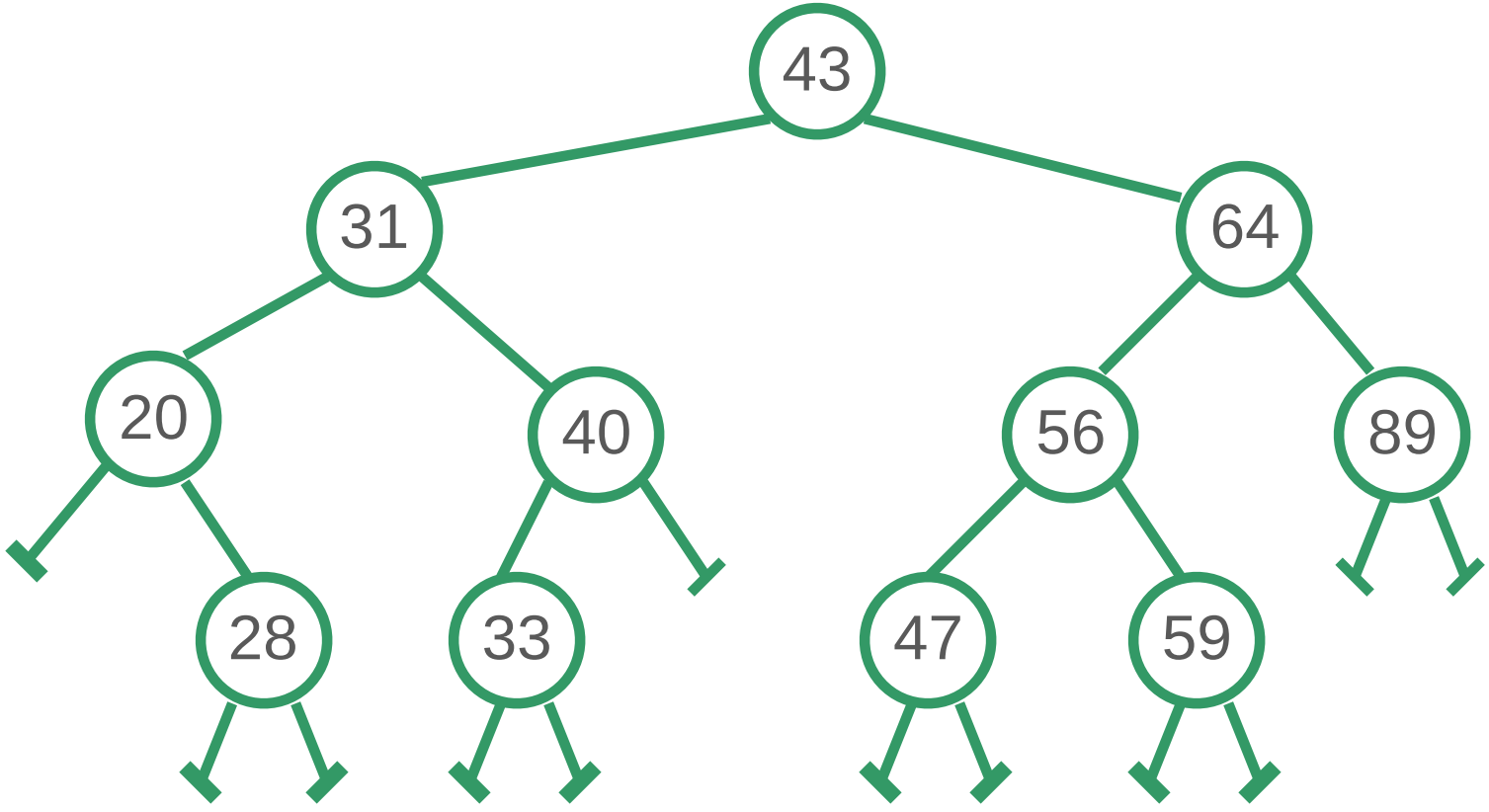
Inorder



Example: Inorder



Example: Inorder



20	28	31	33	40	43	47	56	59	64	89

Print In-order Traversal

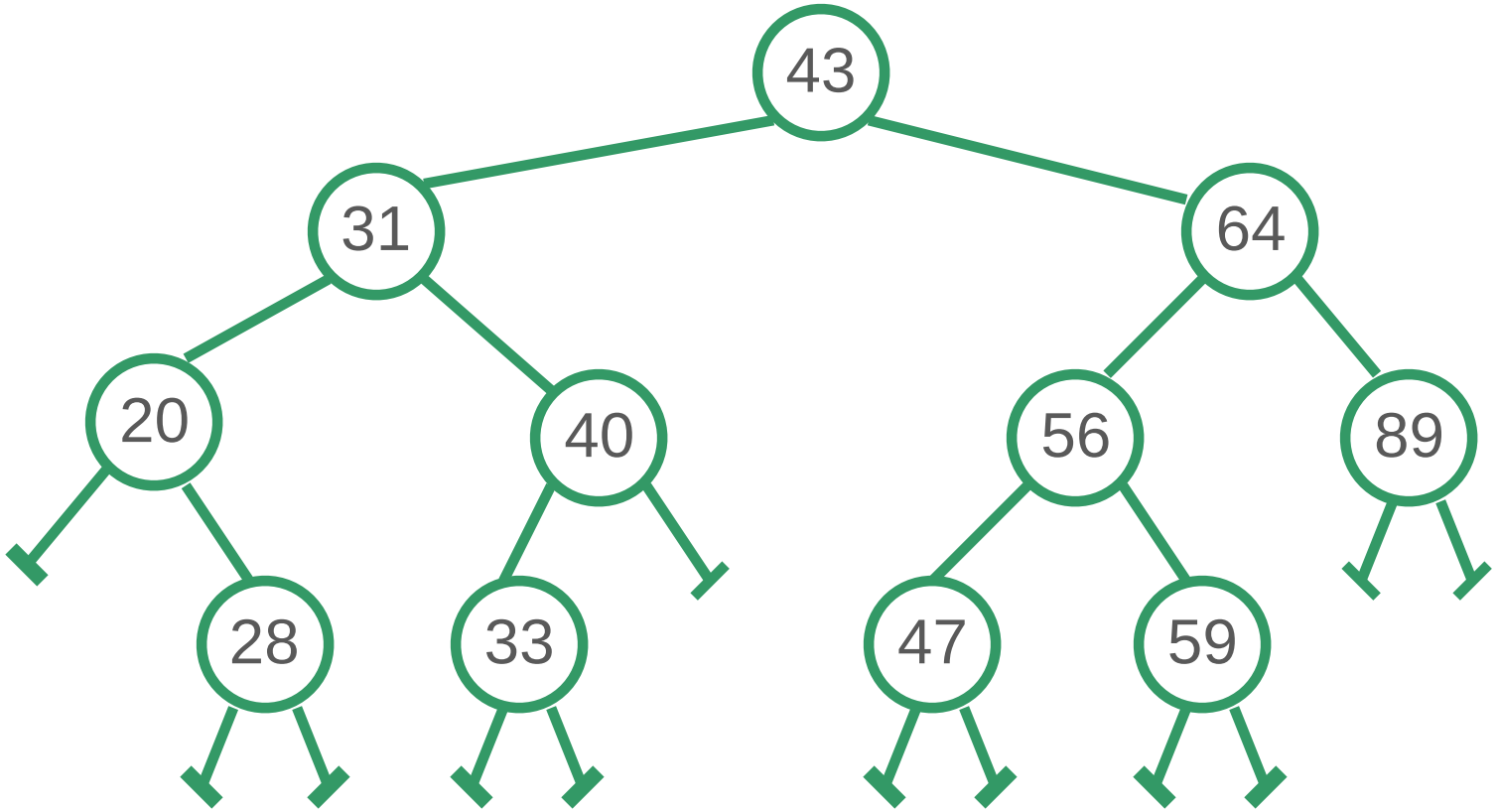
1) Traverse the **left** subtree
2) Print the **root** node
3) Traverse the **right** subtree

```
def print_inorder(self):
    self._print_inorder_aux(self.root)

def _print_inorder_aux(self, current):
    if current is not None:  # if not a base case
        self._print_inorder_aux(current.left)
        print(current)
        self._print_inorder_aux(current.right)
```

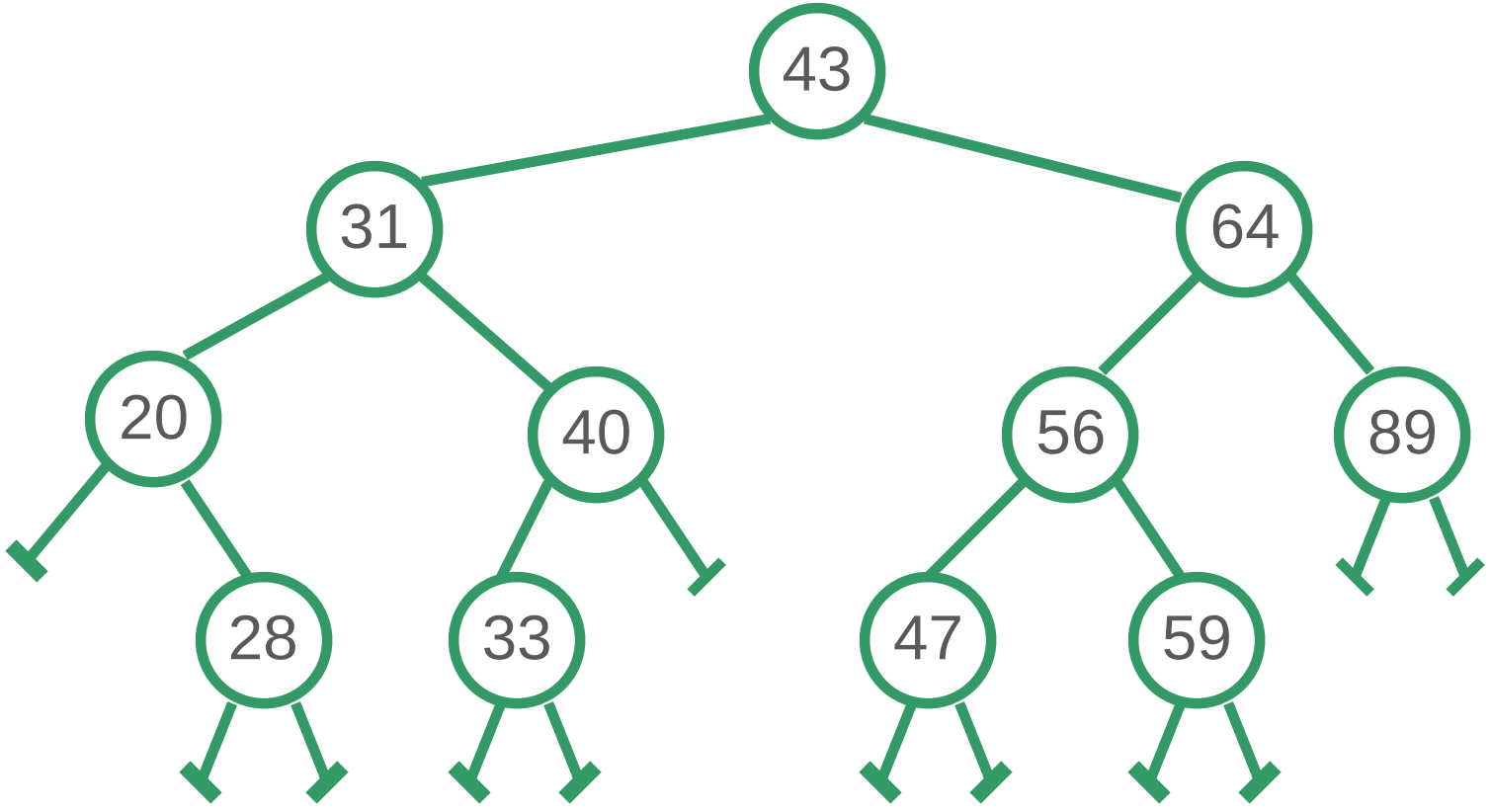
Print Postorder Traversal

Postorder

Example: Postorder



Example: Postorder



28	20	33	40	31	47	59	56	89	64	43

Print Post-order Traversal

1) Traverse the **left** subtree
2) Traverse the **right** subtree
3) Print the **root** node

```
def print_postorder(self):
    self._print_postorder_aux(self.root)

def _print_postorder_aux(self, current):
    if current is not None:  # if not a base case
        self._print_postorder_aux(current.left)
        self._print_postorder_aux(current.right)
        print(current)
```

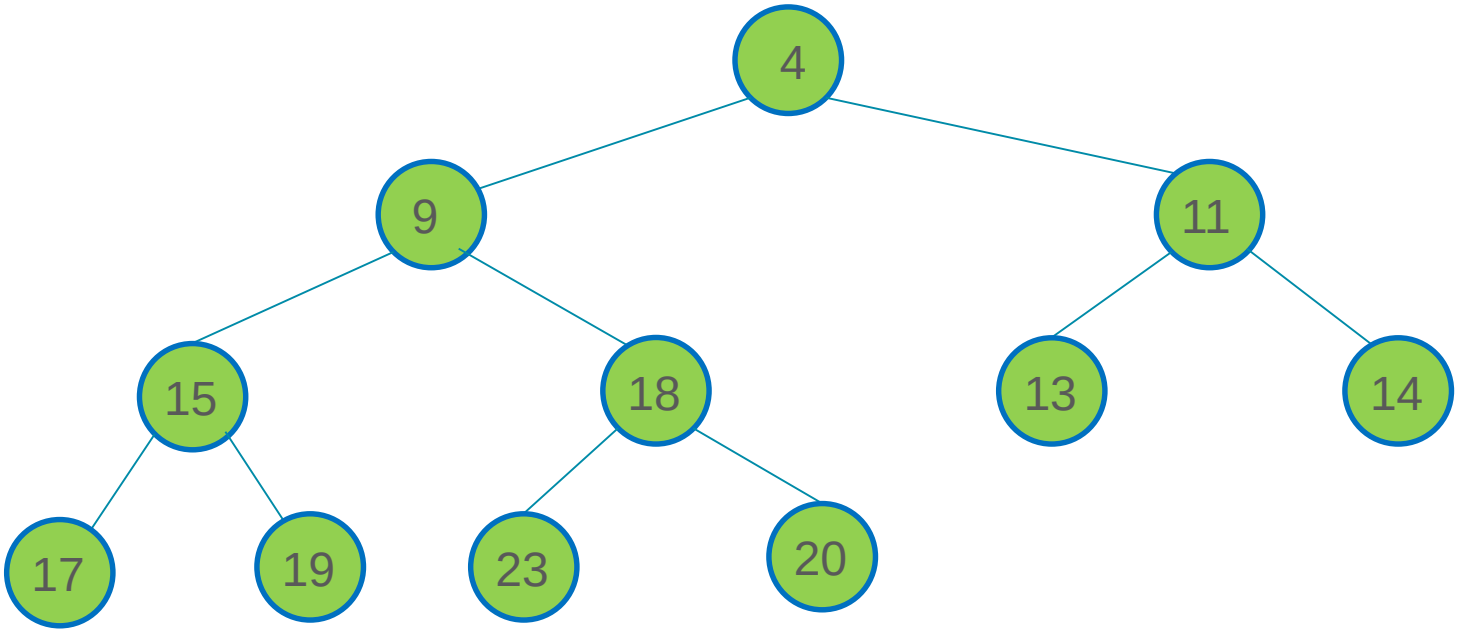
Heap

- A specialized tree-based data structure which is a complete binary tree.
- There are two types:
- **Max-heap**
- **Min-heap**

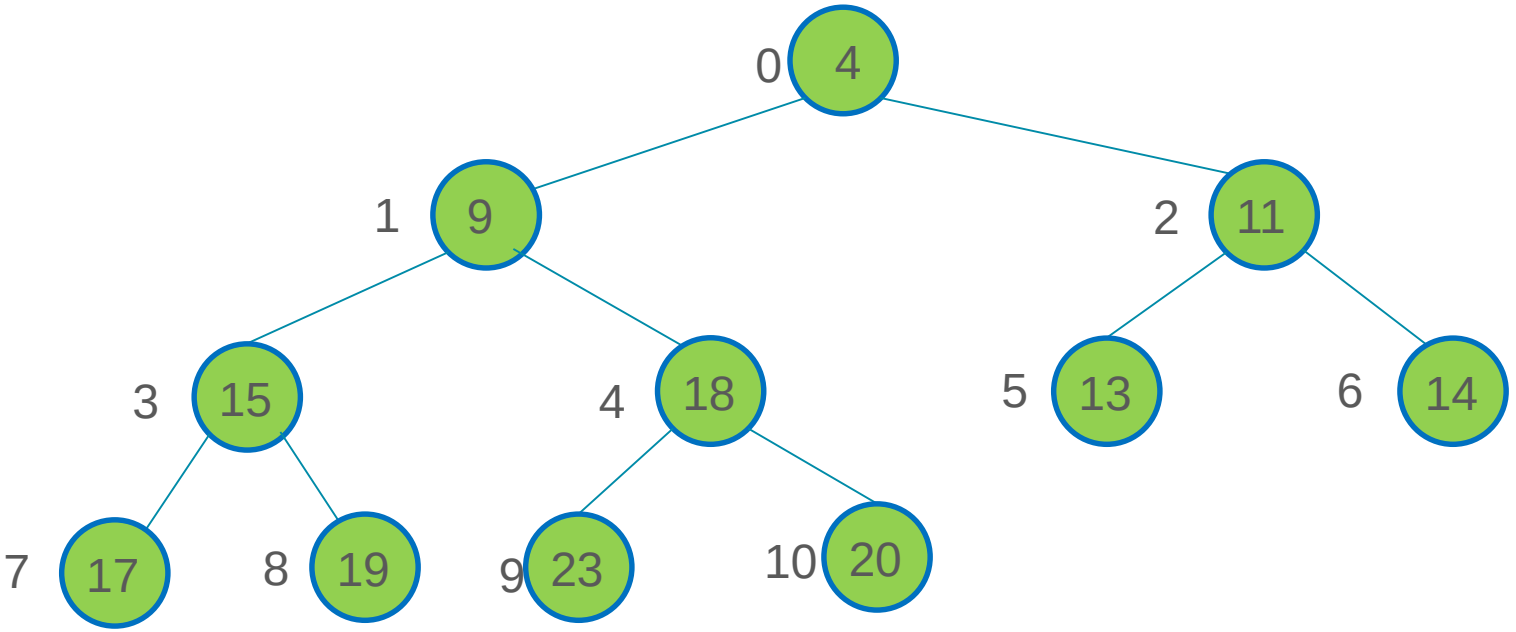
Min-Heap

- A min-heap is a balanced binary tree in which:

✓ each parent is always smaller than or equal to its children (this implies that the root is the smallest element in the heap)

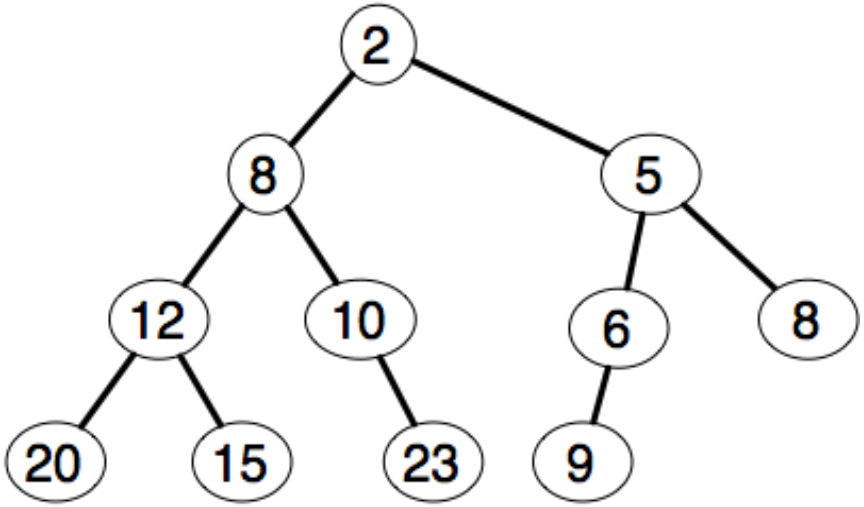


Heap can be represented as List



4	9	11	15	18	13	14	17	19	23	20
0	1	2	3	4	5	6	7	8	9	10

The following tree is a min-Heap.



A. Yes

B. No

Consider a min-heap represented by the list shown below, which of the following is false?

A. E is a right child of C

B. D is a left child of C

C. H is a parent of L

D. D is a parent of K

E. A is a parent of C

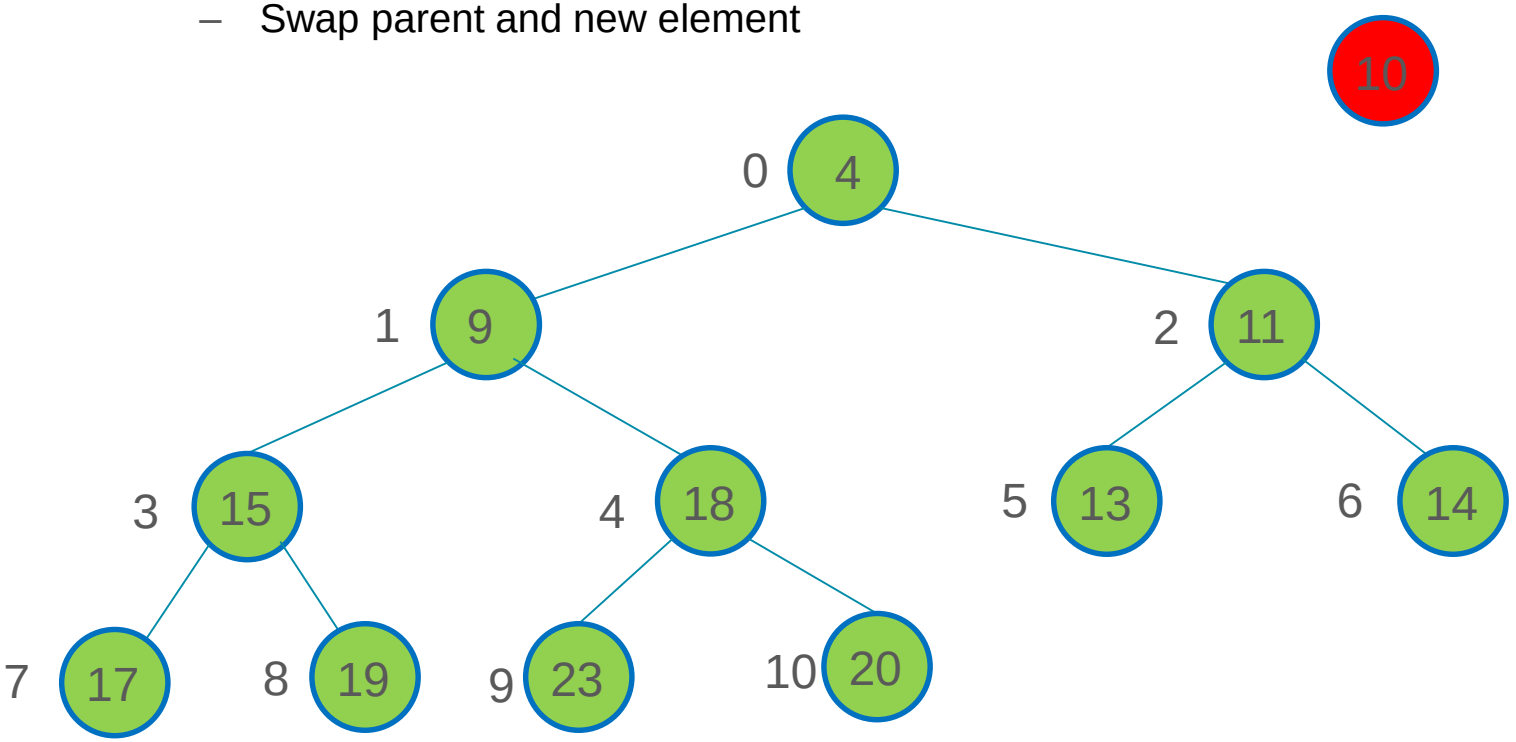
A	B	C	F	H	D	E	G	J	L	K
0	1	2	3	4	5	6	7	8	9	10

Min-Heap Operations

- Insert: insert a value in the min-heap and ensure that heap properties are maintained
- Extraction: Retrieve and delete the minimum value from heap and ensure that heap properties are maintained

Insertion in Heap (up-Heap)

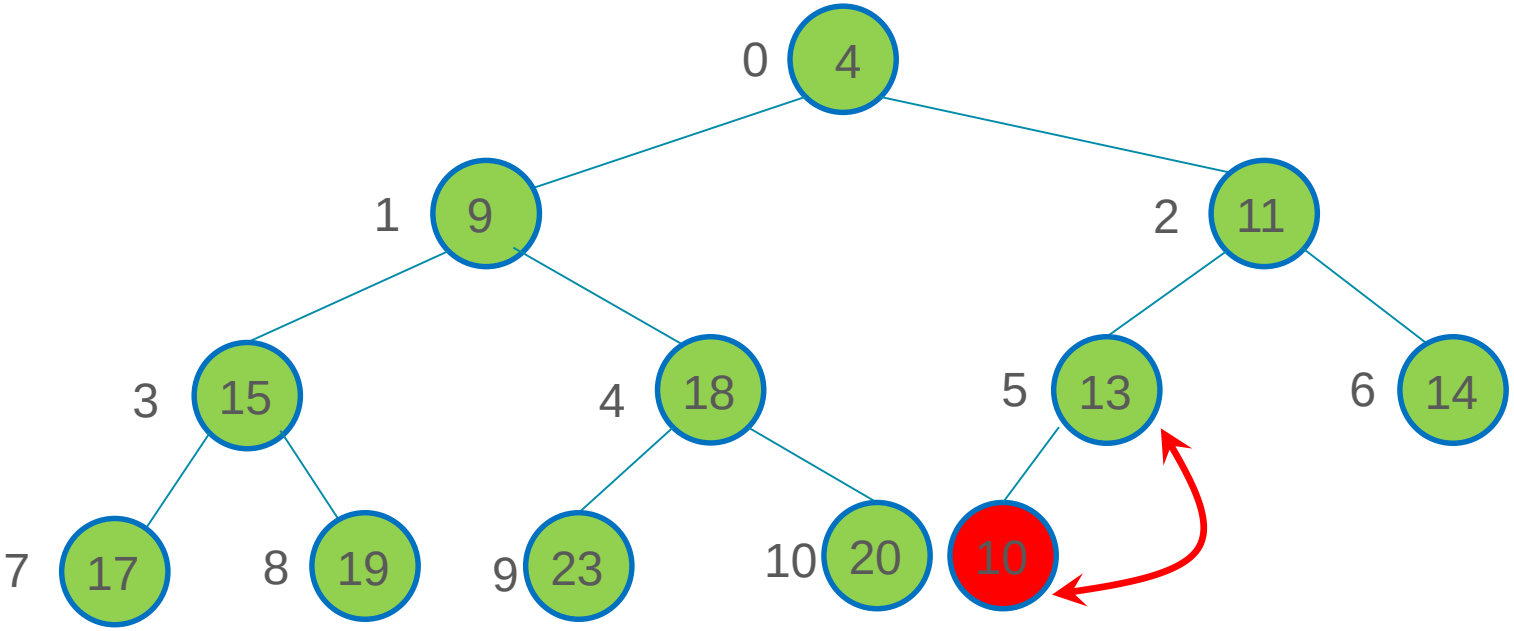
- Append the new element in hList
- While parent(new) > new
  - Swap parent and new element



4	9	11	15	18	13	14	17	19	23	20	
0	1	2	3	4	5	6	7	8	9	10	11

Insertion in Heap (up-Heap)

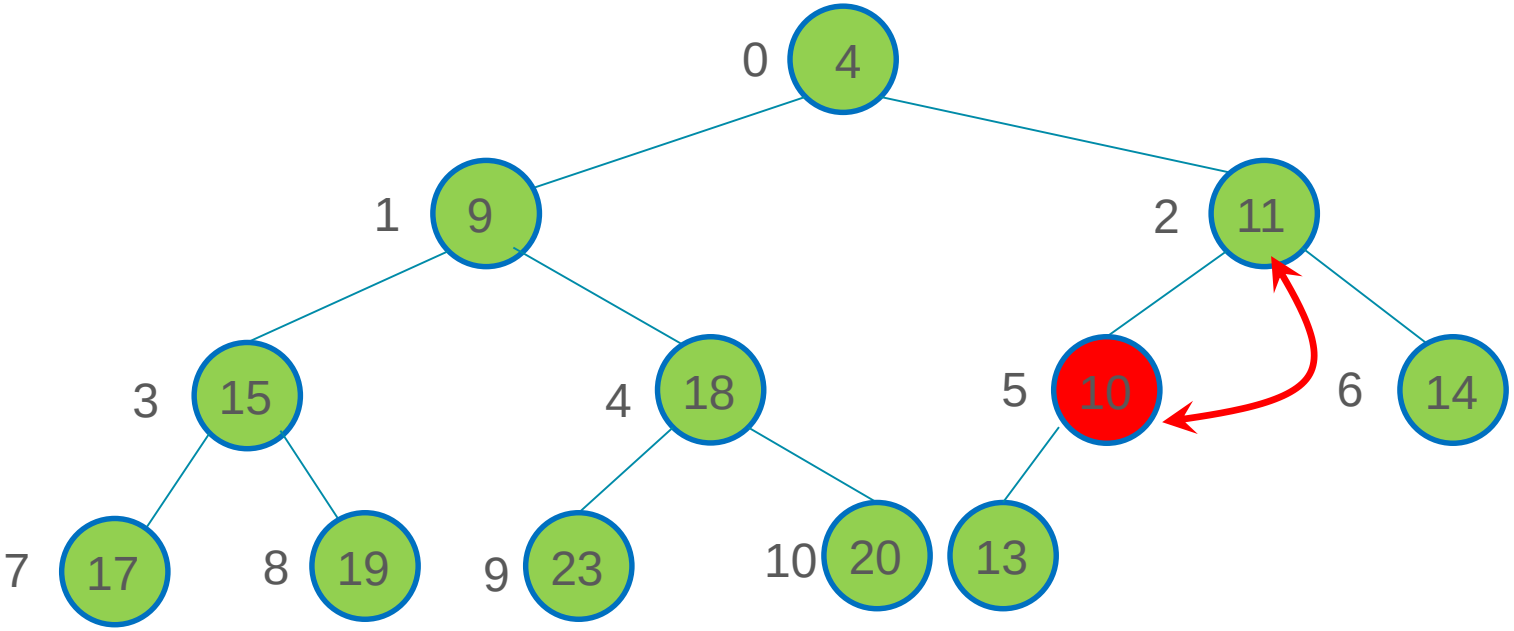
- Append the new element in hList
- While parent(new) > new
  - Swap parent and new element



4	9	11	15	18	13	14	17	19	23	20	10
0	1	2	3	4	5	6	7	8	9	10	11

Insertion in Heap (up-Heap)

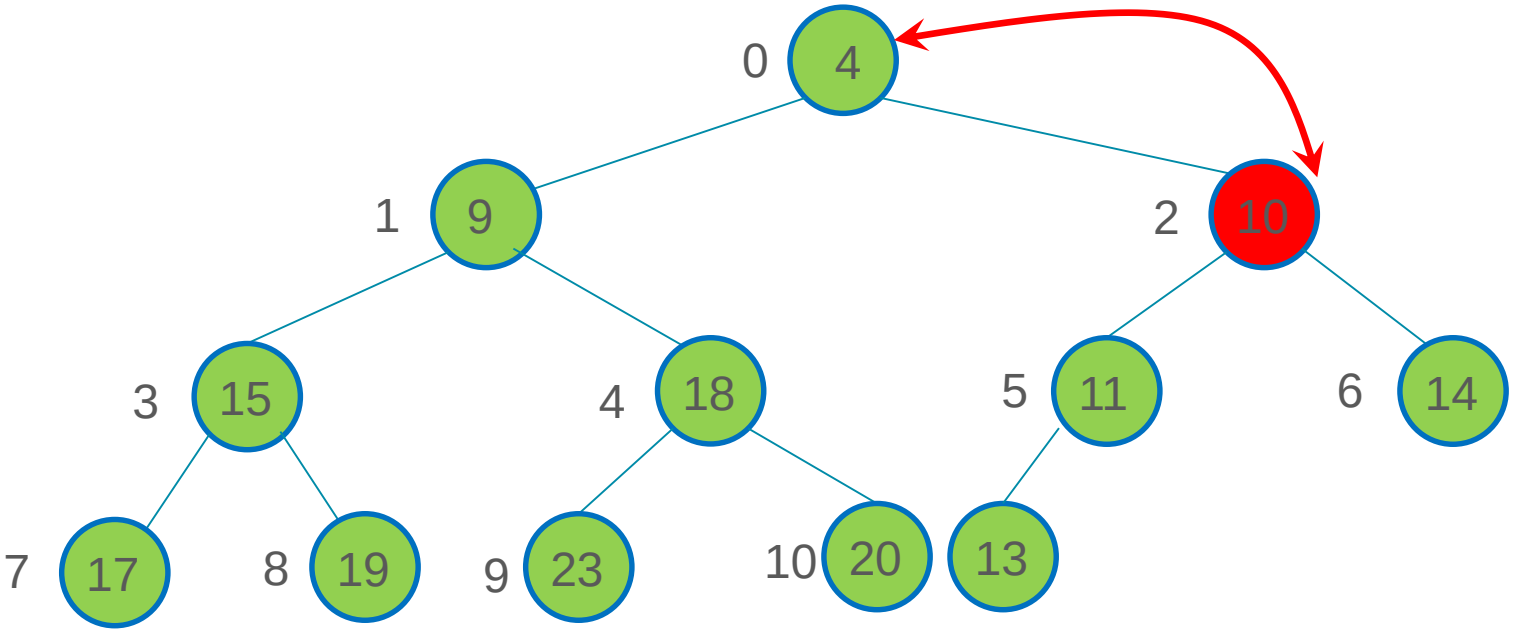
- Append the new element in hList
- While parent(new) > new
  - Swap parent and new element



4	9	11	15	18	10	14	17	19	23	20	13
0	1	2	3	4	5	6	7	8	9	10	11

Insertion in Heap (up-Heap)

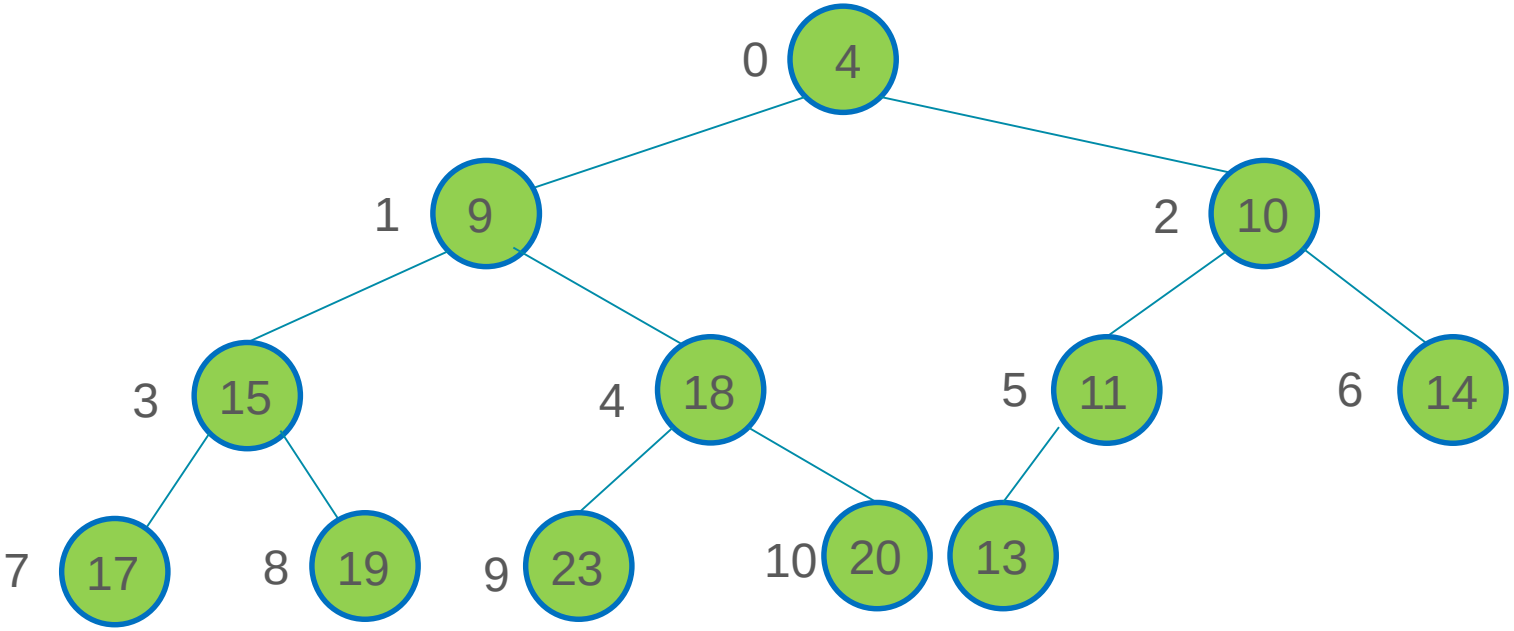
- Append the new element in hList
- While parent(new) > new
  - Swap parent and new element



4	9	10	15	18	11	14	17	19	23	20	13
0	1	2	3	4	5	6	7	8	9	10	11

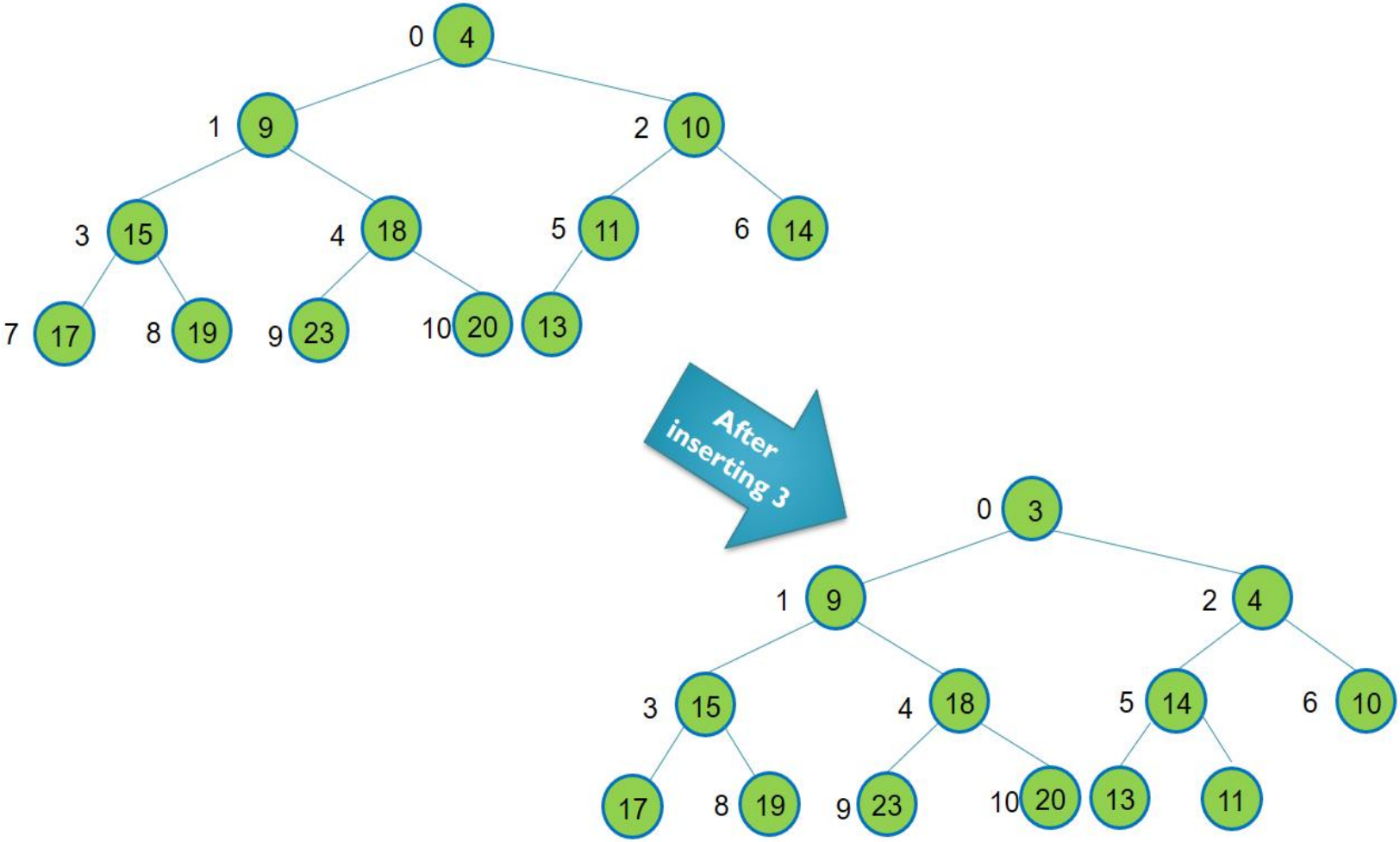
Insertion in Heap (up-Heap)

- Append the new element in hList
- While parent(new) > new
  - Swap parent and new element



4	9	10	15	18	11	14	17	19	23	20	13
0	1	2	3	4	5	6	7	8	9	10	11

After inserting 3, is the heap updated as shown below?

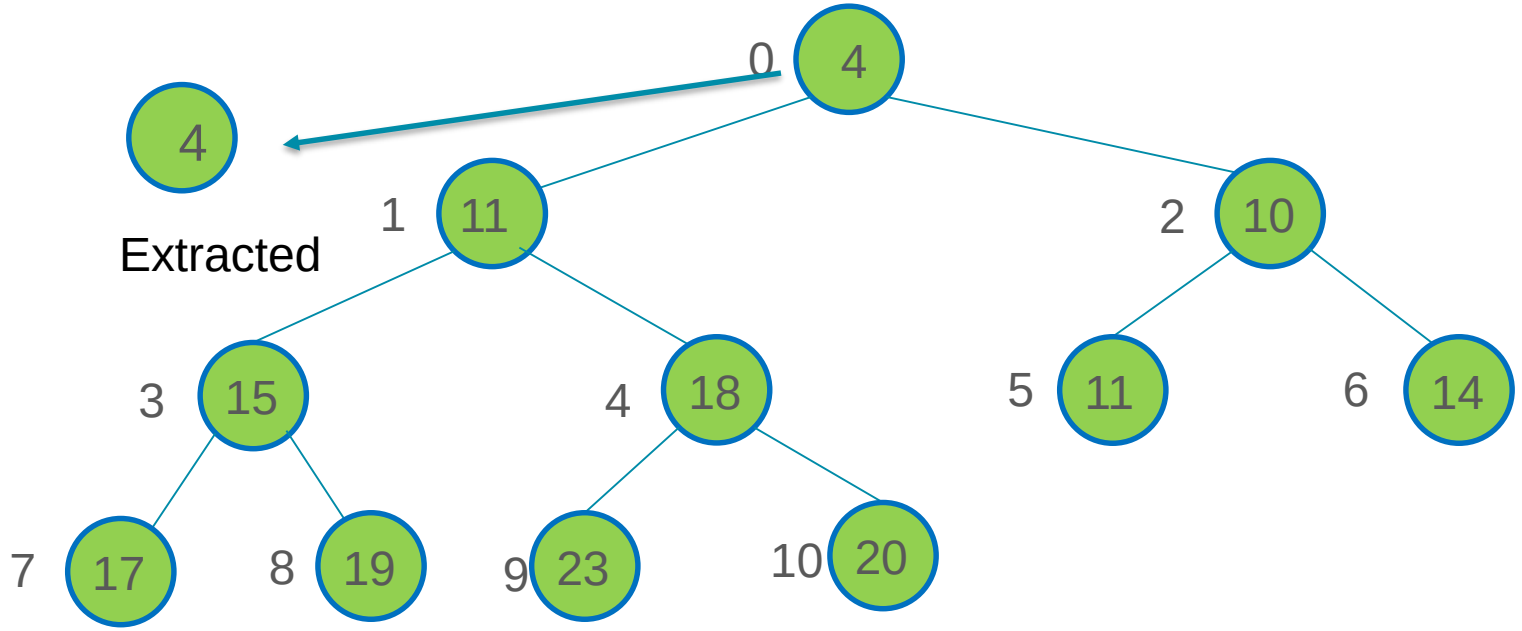


Insertion to Max-heap

- Insert the following items (in the order they are listed) into a max-heap:
- 10, 16, 2, 29, 6, 14, 11, 4.

extractMin from Heap (downHeap)

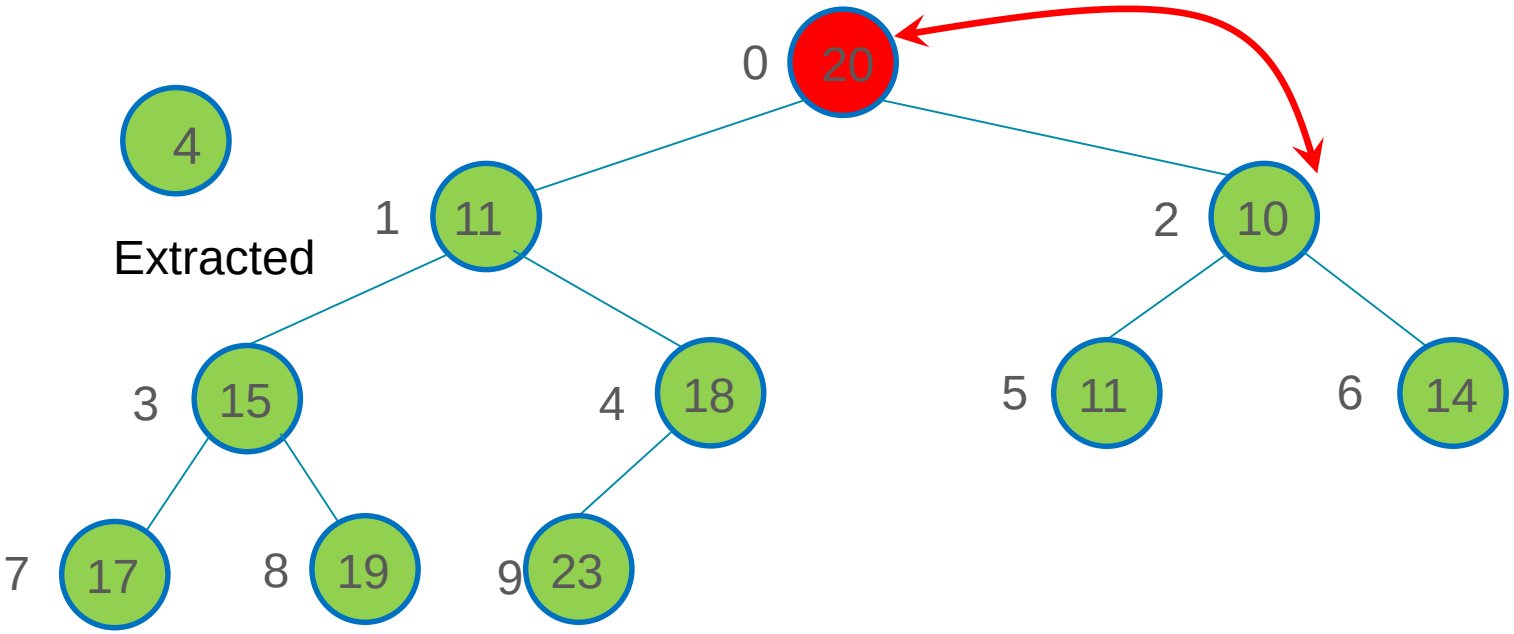
- Store and delete hList[0]
- Move hList[N-1] (called last) to hList[0]



4	11	10	15	18	11	14	17	19	23	20
0	1	2	3	4	5	6	7	8	9	10	11

extractMin from Heap (downHeap)

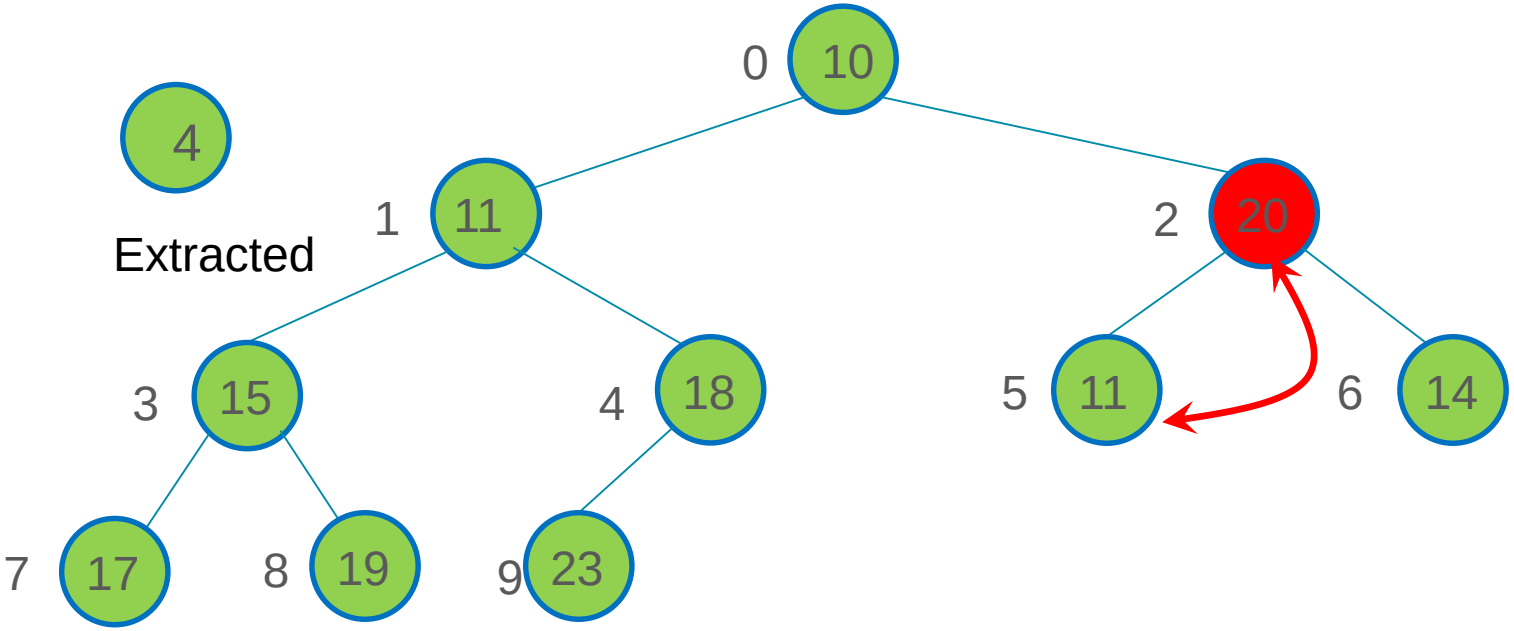
- Store and delete hList[0]
- Move hList[N-1] (called last) to hList[0]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)



20	11	10	15	18	11	14	17	19	23	
0	1	2	3	4	5	6	7	8	9	10	11

extractMin from Heap (downHeap)

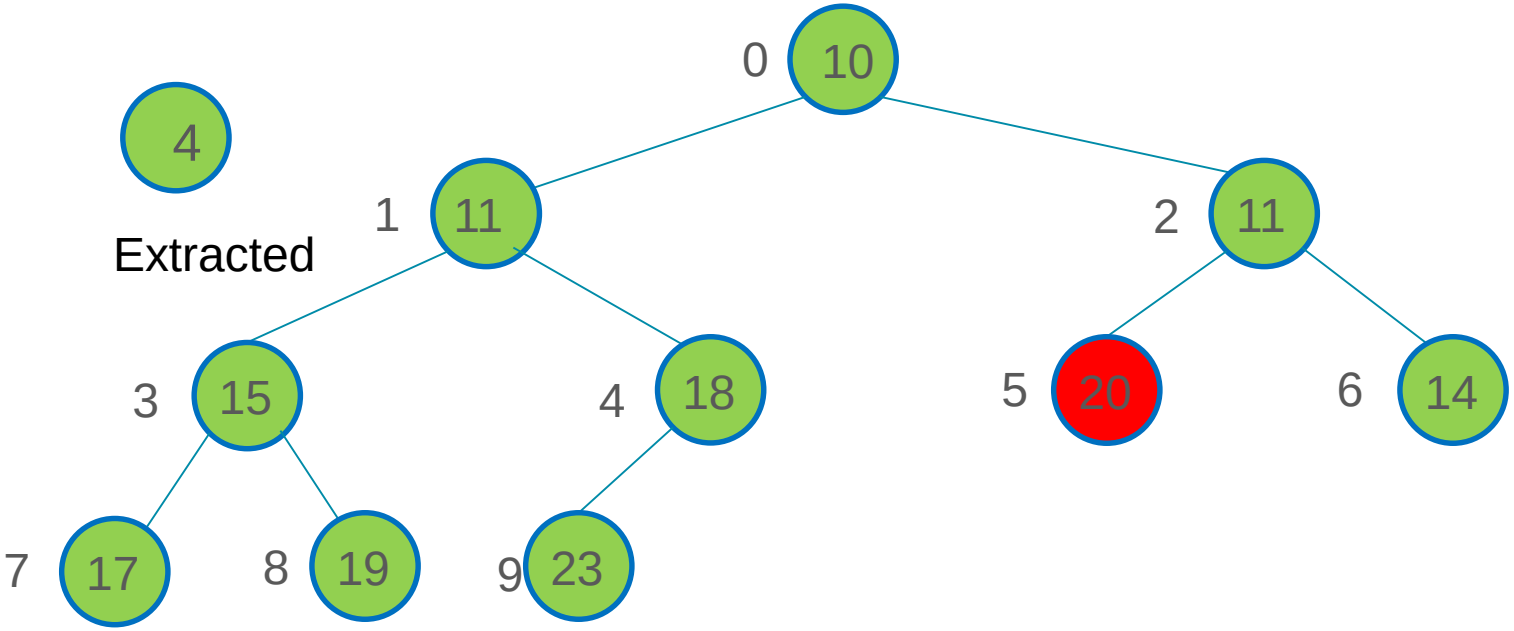
- Store and delete hList[0]
- Move hList[N-1] (called last) to hList[0]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)



10	11	20	15	18	11	14	17	19	23	
0	1	2	3	4	5	6	7	8	9	10	11

extractMin from Heap (downHeap)

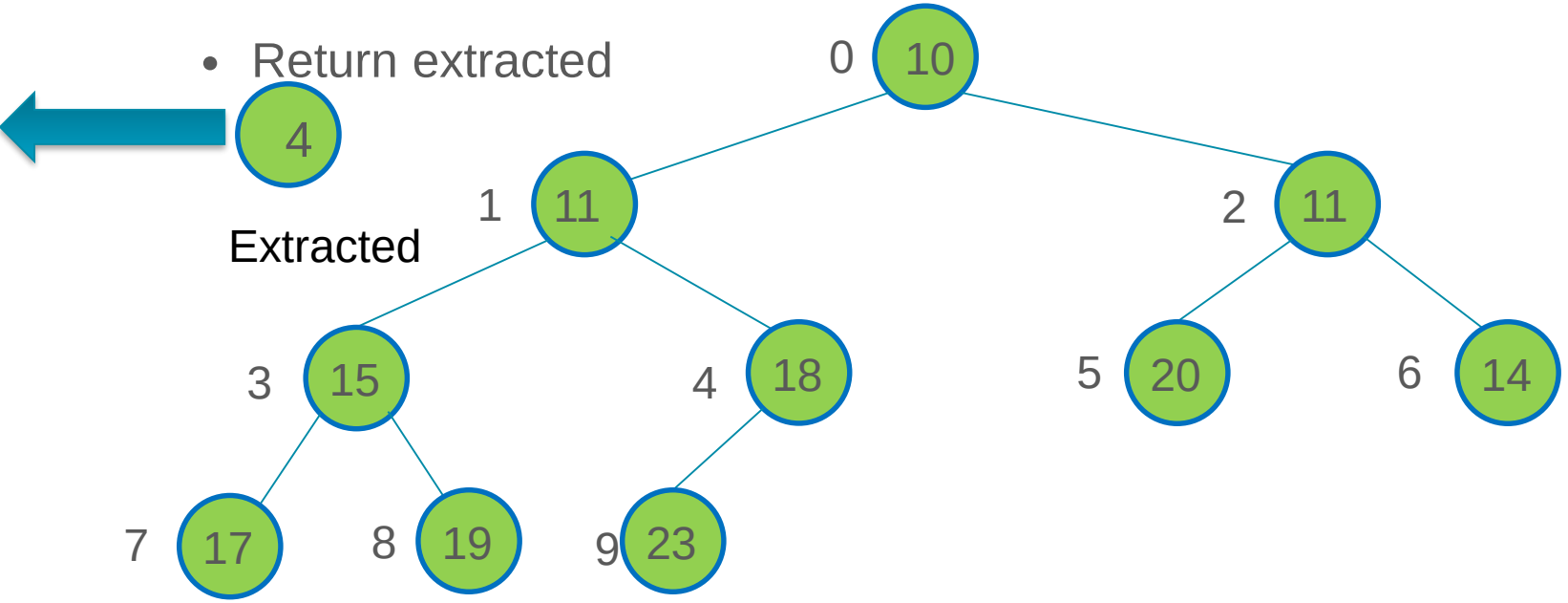
- Store and delete hList[0]
- Move hList[N-1] (called last) to hList[0]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)



10	11	11	15	18	20	14	17	19	23	
0	1	2	3	4	5	6	7	8	9	10	11

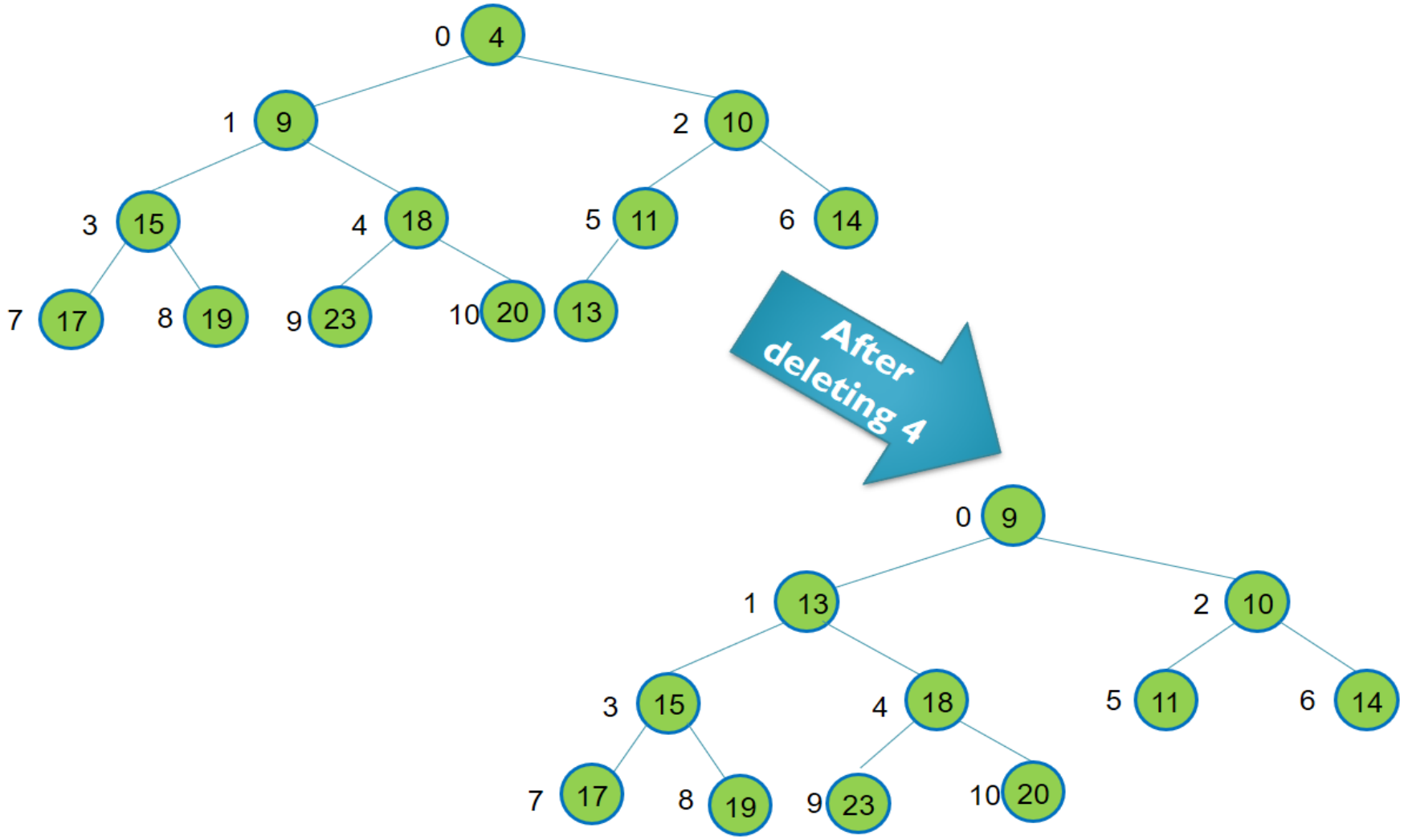
extractMin from Heap (downHeap)

- Store and delete hList[0]
- Move hList[N-1] (called last) to hList[0]
- While leftchild(last) < last or rightchild(last) < last
  - Swap last with minimum(leftchild,rightchild)
- Return extracted



10	11	11	15	18	20	14	17	19	23	
0	1	2	3	4	5	6	7	8	9	10	11

After deleting 4, is the heap updated as shown below?



How the **Heap Sort** works?

- Discuss how a **heap** can be used to **sort a list**. Using a heap, sort the list [10, 16, 2, 29, 6, 14, 11, 4] in **ascending order**. Show the state of the heap at every step.

Final Announcement

- No More Lab Session this week, instead there will be a revision session on Thursday at 1:00 pm.
- Lab Effort Mark
- Thank you for your attendance and cooperation
- Please complete LUMES module evaluation
- Thank you and Stay Safe ☺