# CS 225

## Data Structures

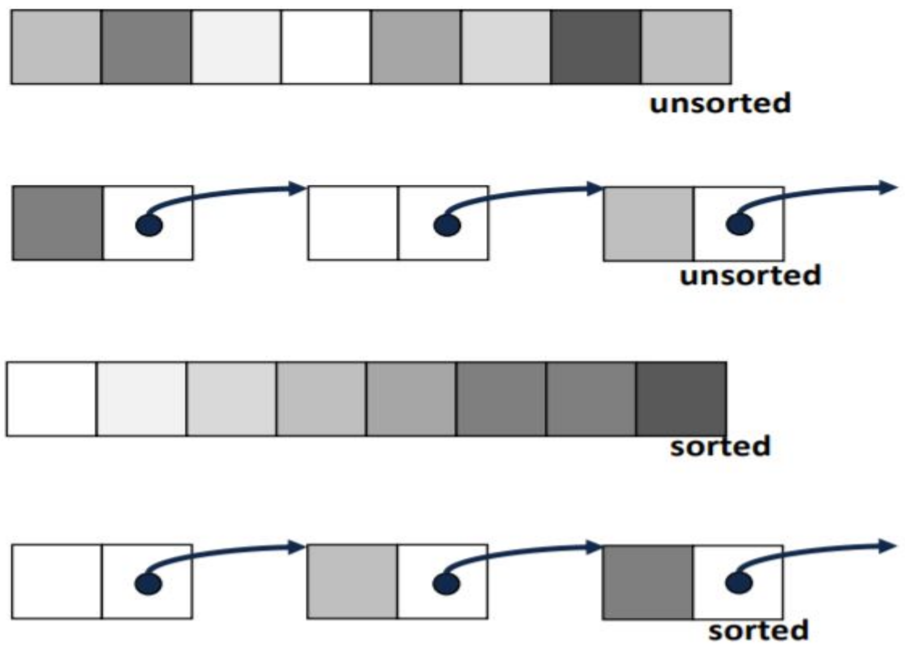
# Previously in lectures:

- Priority Queue ADT
- (min)Heap
- insert
- heapifyUp
- removeMin
- heapifyDown
- BuildHeap

# Priority Queue Implementation

- **Scenario**: *n* elements, each with an associated priority $p_i$, is given to you one element at a time.
- **Task:** How would you remove the highest priority element?

| Insert | removeMin |
|---|---|
| O(1)* | O(n) |
| O(1) | O(n) |
| O(n) | O(1) |
| O(n) | O(1) |

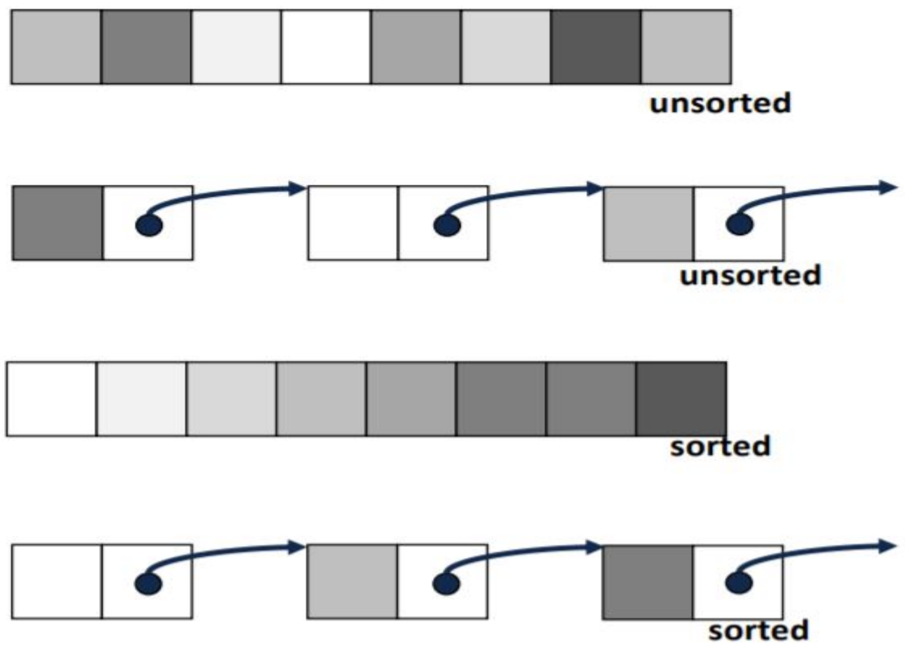


# Priority Queue Implementation

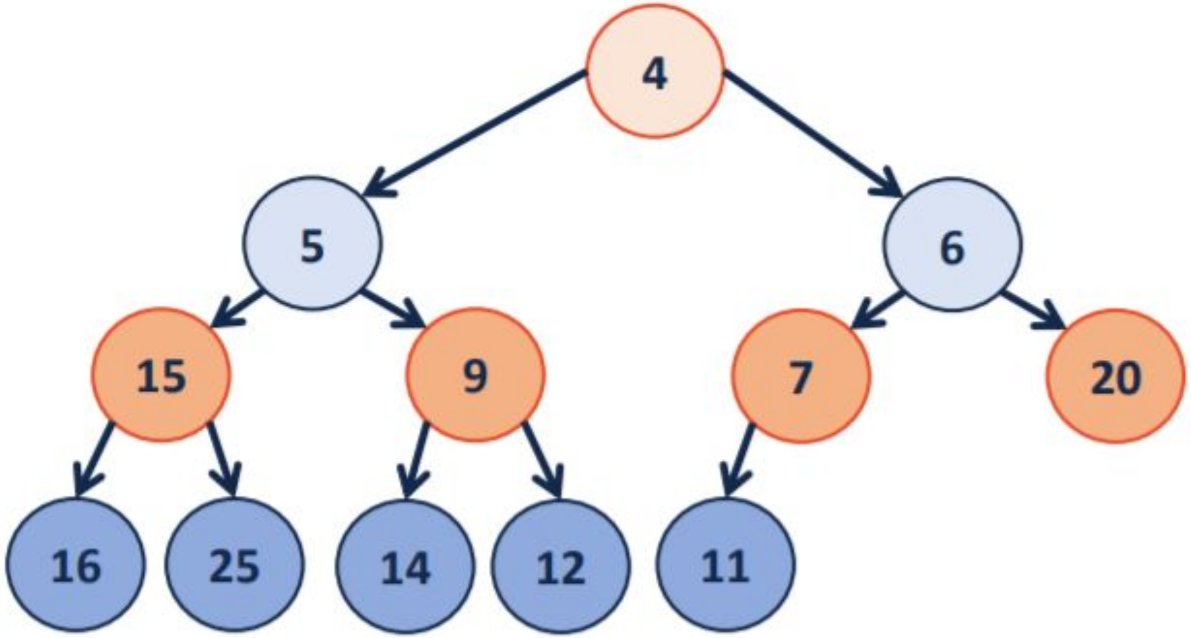
**We Can Do Better! Enter Heaps!**

- **Scenario**: *n* elements, each with an associated priority $p_i$, is given to you one element at a time.
- **Task:** How would you remove the highest priority element?

| Insert | removeMin |
|---|---|
| O(1)* | O(n) |
| O(1) | O(n) |
| O(n) | O(1) |
| O(n) | O(1) |



# Heaps

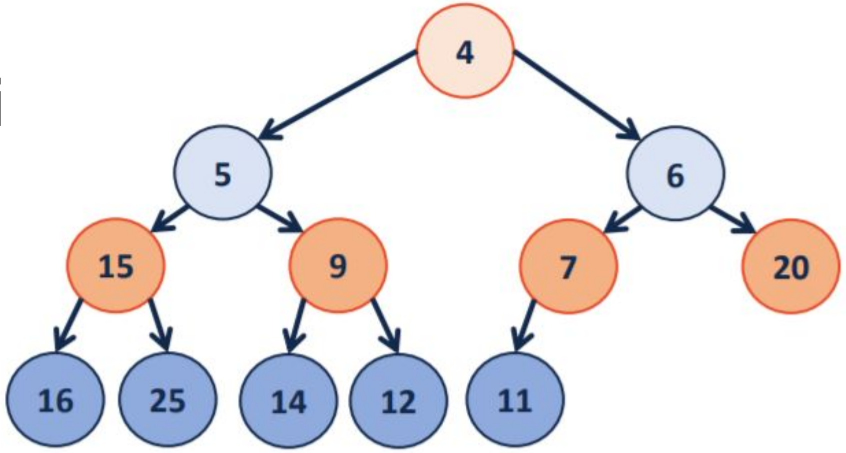



- Complexity
  - *Insert*: O(log n)
  - *removeMin*: O(log n)
  - *building heap*: O(n) [One-time]
- each node is an element with an associated priority
- root *always* has the highest priority (expressed as number)
- Binary Heaps: Complete binary tree
  - *min-heap* : minimum between two numbers; root is smallest element.
  - *max-heap*: maximum between two numbers; root is largest element.
- Not BST: in-order traversal may not yield an ordered list.

**Remember**: *The parent has higher priority than any of its children*

# Worksheet Exercise #1

**Exercise 1:** Suppose the current node is at index **i**, fill in the blank for the indices of certain locations i the tree. You will be implementing these as functions in your lab. Remember, these formulas depend on whether you are populating the 0th index of the array or not.




Current node = **i**

Root index = 0

Left child =

Right child =

Parent =

Root index = 1

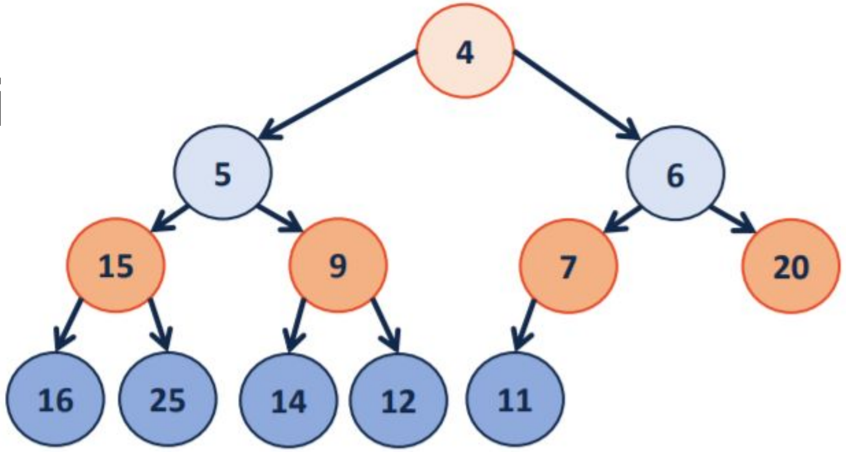
Left child =

Right child =

Parent =

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 | | | |

**Exercise 1:** Suppose the current node is at index **i**, fill in the blank for the indices of certain locations i the tree. You will be implementing these as functions in your lab. Remember, these formulas depend on whether you are populating the 0th index of the array or not.




Current node = **i**

Root index = 0

Left child = 2*i + 1

Right child = 2*i + 2

Parent = floor[ (i-1)/2 ]

Root index = 1

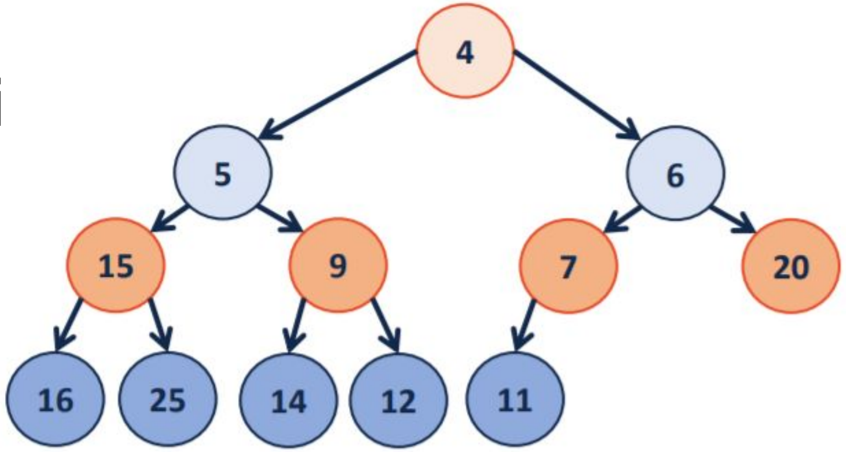
Left child =

Right child =

Parent =

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 | | | |

**Exercise 1:** Suppose the current node is at index **i**, fill in the blank for the indices of certain locations i the tree. You will be implementing these as functions in your lab. Remember, these formulas depend on whether you are populating the 0th index of the array or not.




Current node = **i**

Root index = 0

Left child = 2*i + 1

Right child = 2*i + 2

Parent = floor[ (i-1)/2 ]

Root index = 1

Left child = 2*i

Right child = 2*i + 1

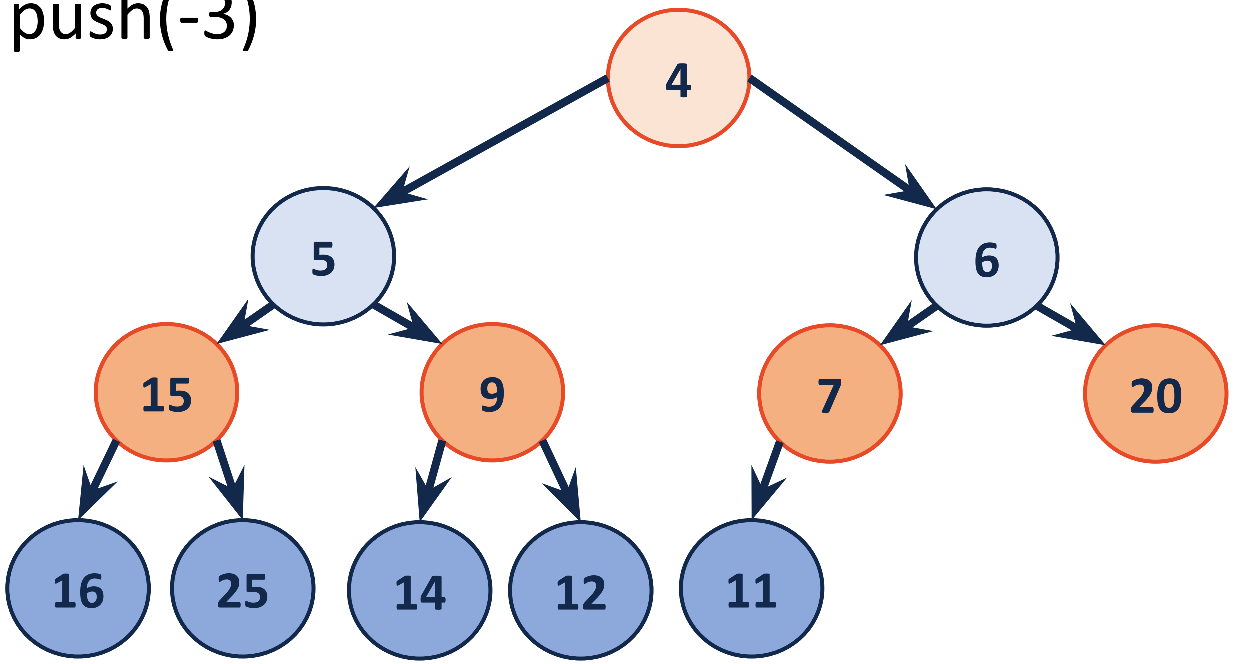
Parent = floor[ i/2 ]

# Insert/Push

# push(key)

1. Add new element in the end of the vector (index `i`)
2. Reset heap property from index `i` – `heapifyUp(i)`

push(-3)

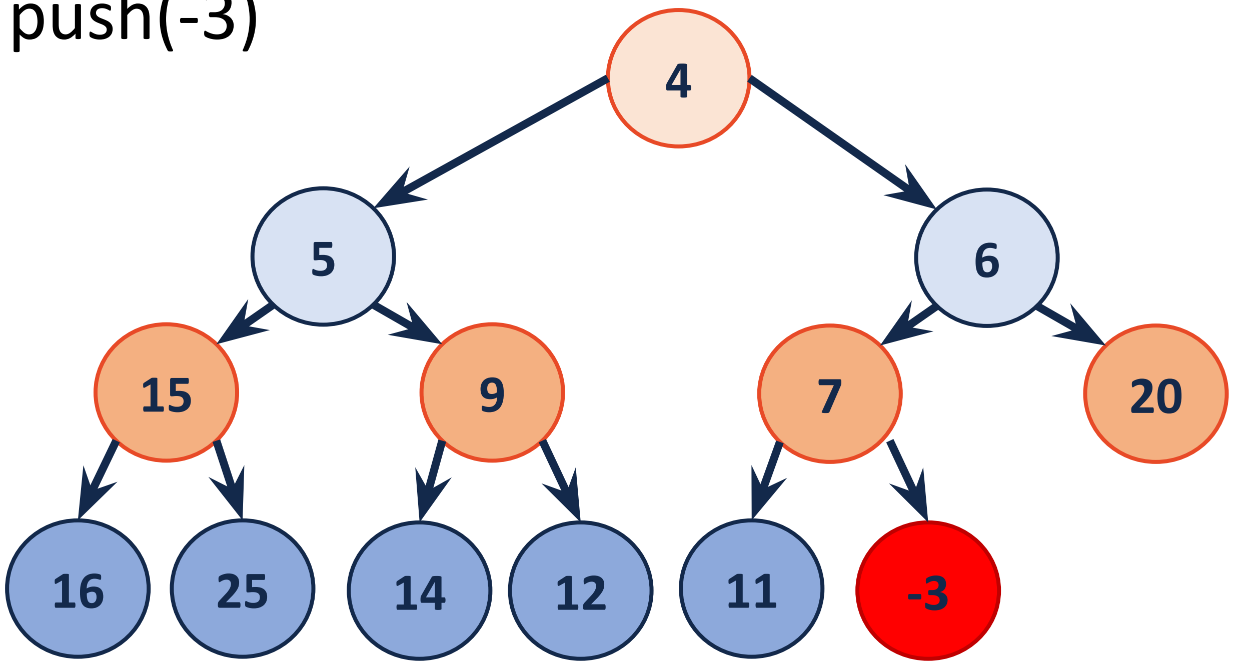



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 |

# push(key)

1. Add new element in the end of the vector (index `i`)
2. Reset heap property from index `i` using `heapifyUp(i)`

push(-3)




| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 | -3 |

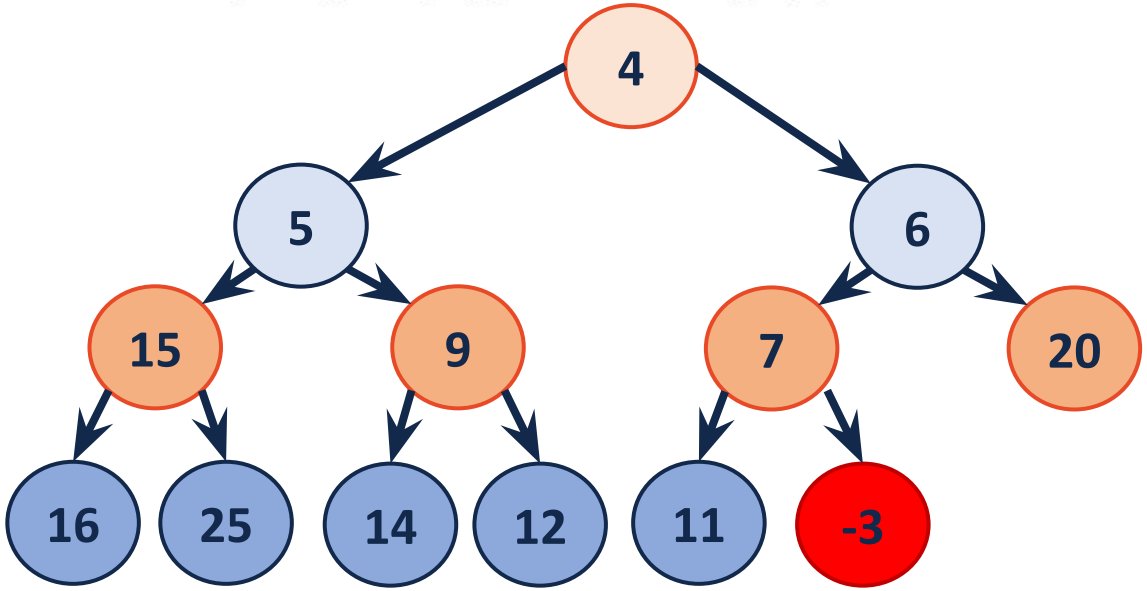
While the new element on given index has a smaller value than its parent, swap the element and its parent.

**heapifyUp($i$)**

if $i$ != rootIndex && A[$i$] < A[parent($i$)]

  swap($i$, parent($i$))

  heapifyUp(parent($i$))

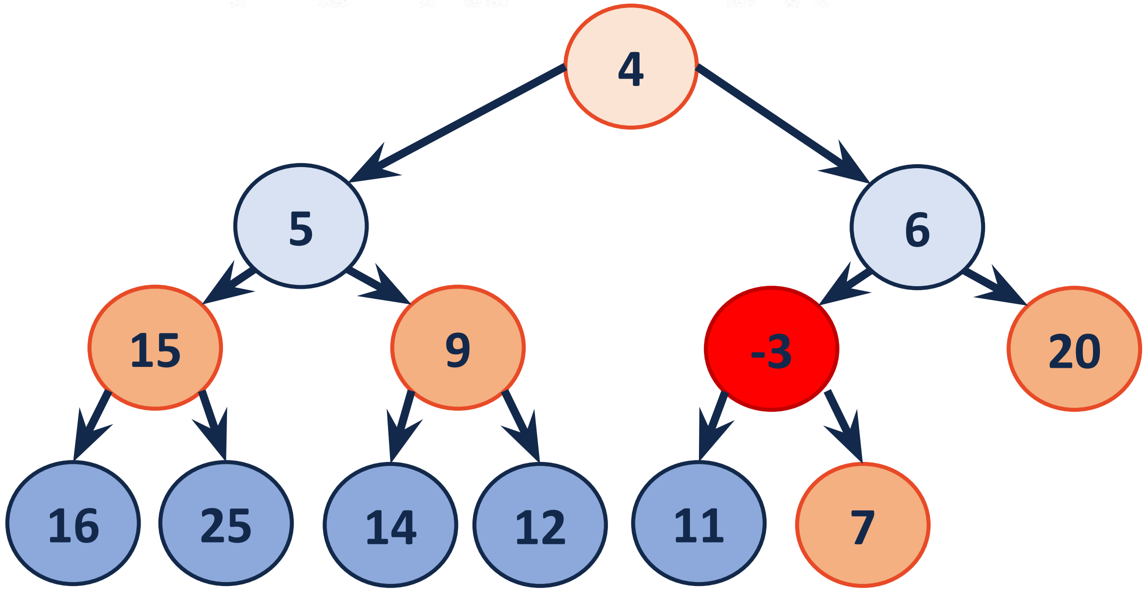



| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 | -3 |

While the new element on given index has a smaller value than its parent, swap the element and its parent.

**heapifyUp($i$)**

```
if i != rootIndex && A[i] < A[parent(i)]
    swap(i, parent(i))
    heapifyUp(parent(i))
```




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | -3 | 20 | 16 | 25 | 14 | 12 | 11 | 7 |

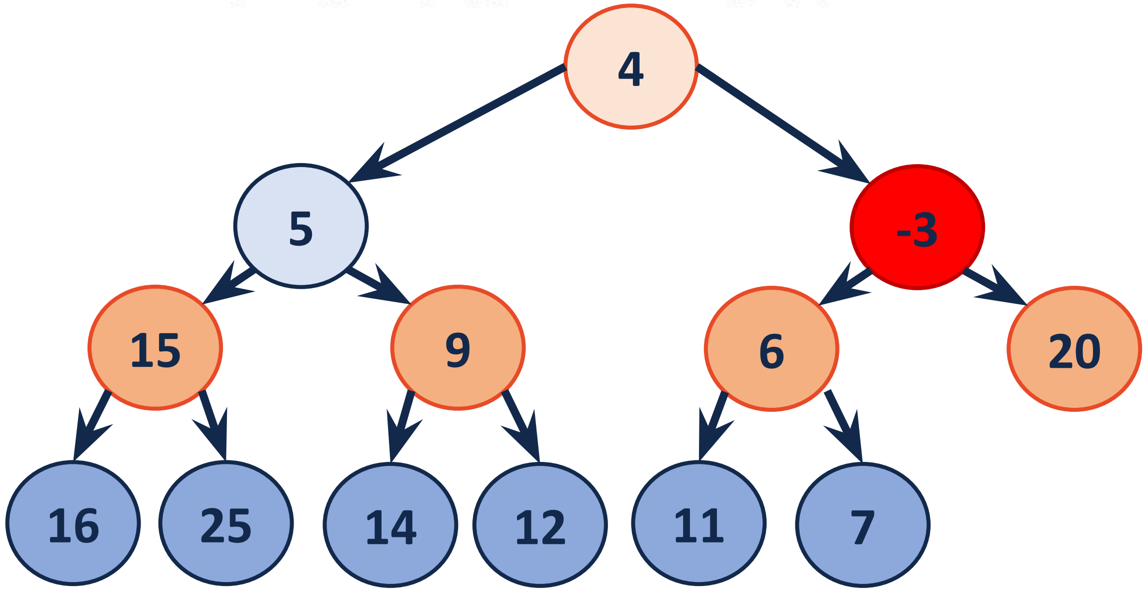
While the new element on given index has a smaller value than its parent, swap the element and its parent.

**heapifyUp($i$)**

if $i$ != rootIndex && A[$i$] < A[parent($i$)]

swap($i$, parent($i$))

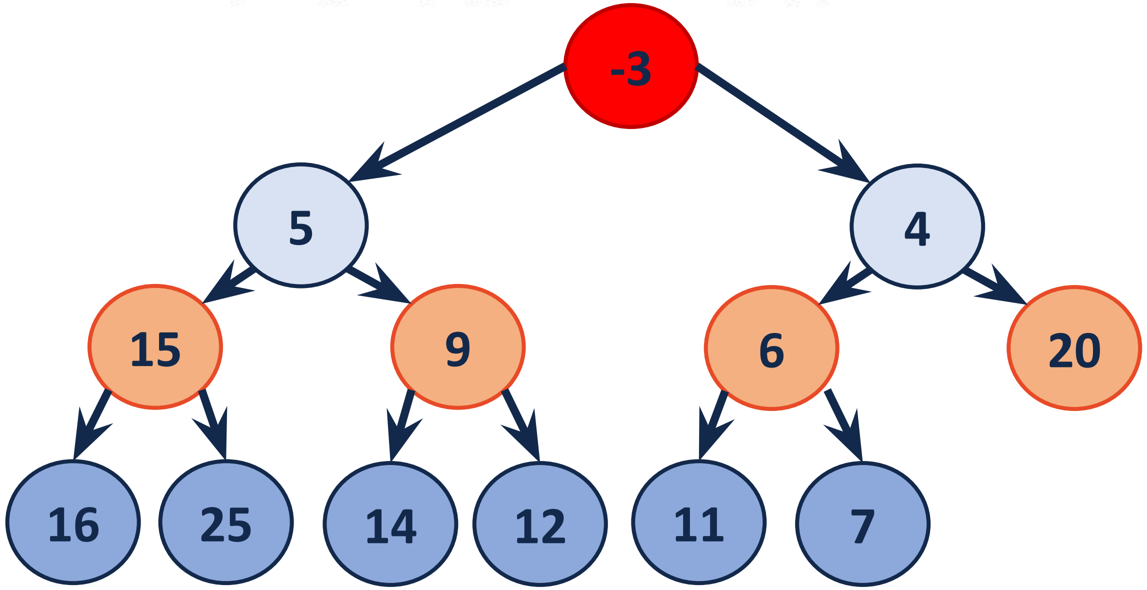
heapifyUp(parent($i$))




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | -3 | 15 | 9 | 6 | 20 | 16 | 25 | 14 | 12 | 11 | 7 |

While the new element on given index has a smaller value than its parent, swap the element and its parent.

```
heapifyUp(i)
    if i != rootIndex && A[i] < A[parent(i)]
        swap(i, parent(i))
        heapifyUp(parent(i))
```



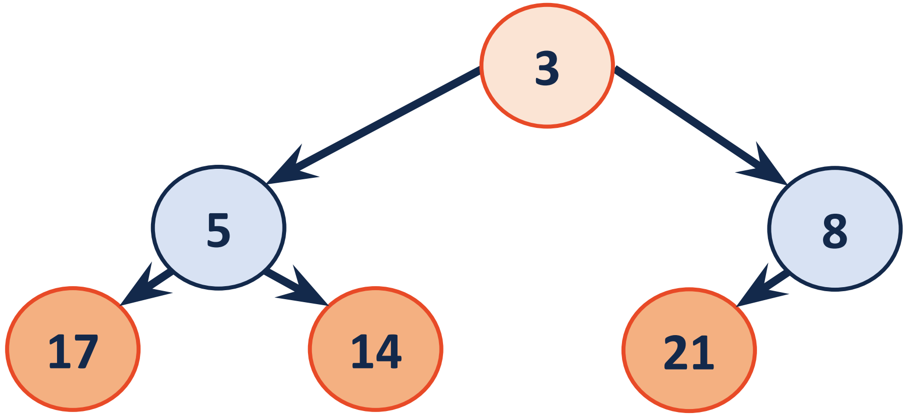

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | -3 | 5 | 4 | 15 | 9 | 6 | 20 | 16 | 25 | 14 | 12 | 11 | 7 |

✓

# Worksheet
# Exercise #2.1 and #2.2

Suppose 7 is inserted into the heap below.

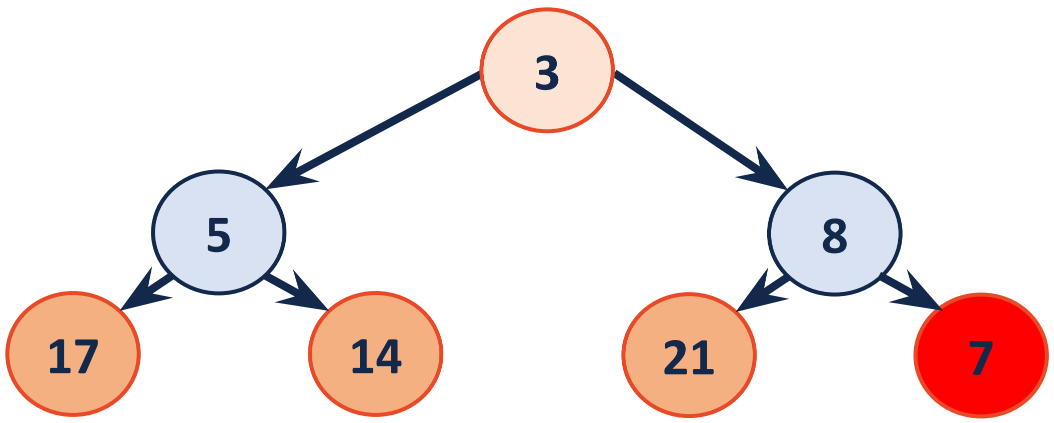
- **Exercise 2.1:** Suppose 7 is inserted into the heap below. What will 7’s left and right children be?
- **Exercise 2.2:** What is the array representation of the tree after **7** is inserted?




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 3 | 5 | 8 | 17 | 14 | 21 | |

Suppose 7 is inserted into the heap below.

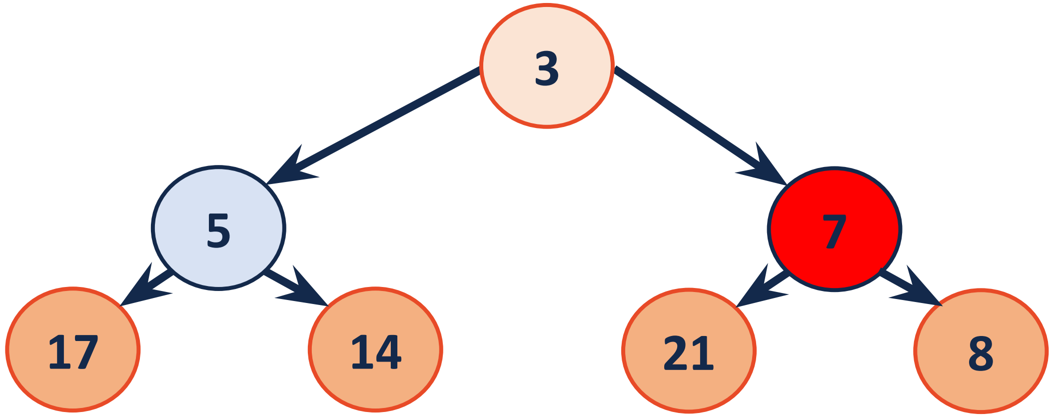
- **Exercise 2.1:** Suppose 7 is inserted into the heap below. What will 7’s left and right children be?
- **Exercise 2.2:** What is the array representation of the tree after **7** is inserted?




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 3 | 5 | 8 | 17 | 14 | 21 | 7 |

Suppose 7 is inserted into the heap below.

- **Exercise 2.1:** Suppose 7 is inserted into the heap below. What will 7’s left and right children be? Left child = 21; Right Child = 8;
- **Exercise 2.2:** What is the array representation of the tree after **7** is inserted?



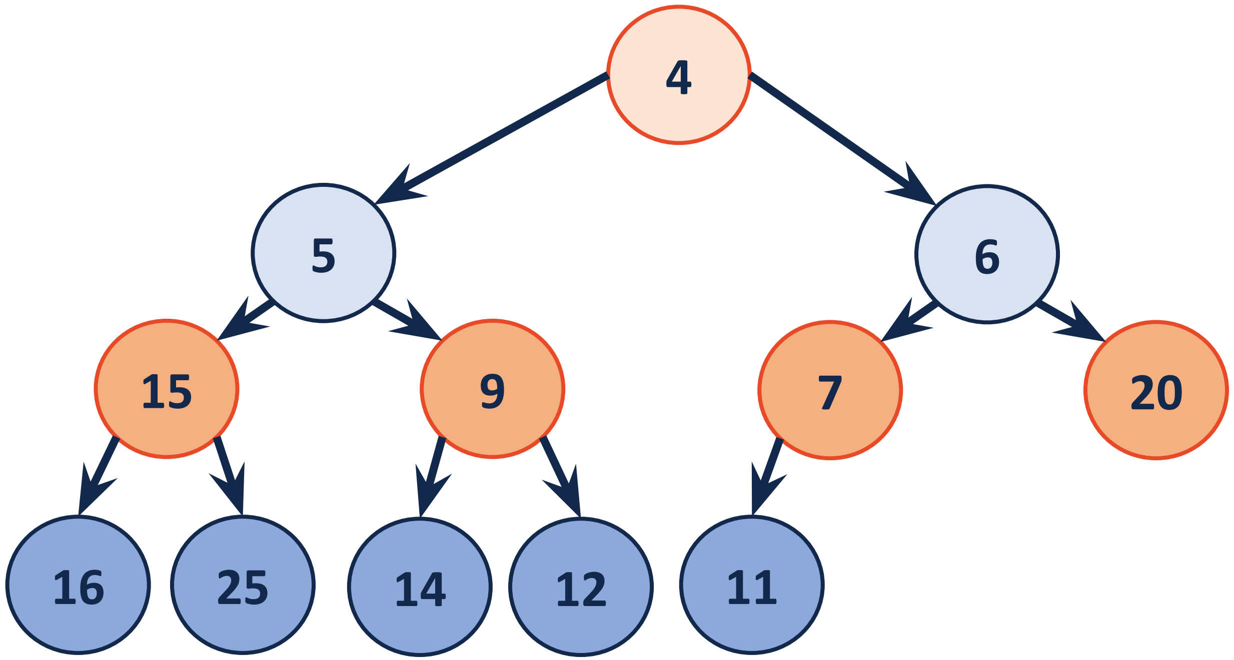

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 3 | 5 | 7 | 17 | 14 | 21 | 8 |

✓

# Delete/Pop

# pop()

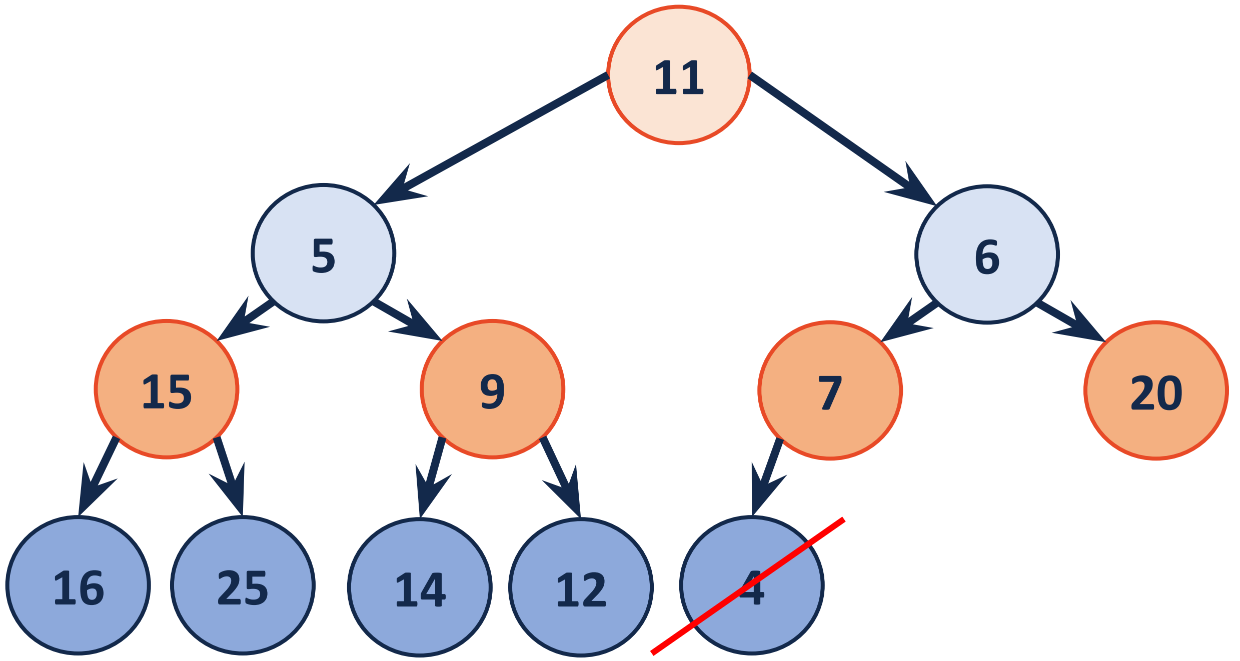
1. Swap the last element(index `i`) for the root
2. `result=_elems[i]`
3. Remove the last element: (`_elems.pop_back()`)
4. `HeapifyDown(root);`
5. Return `result;`




| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 4 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | 11 |

# pop()

1. Swap the last element(index `i`) for the root
2. `result=_elems[i]`
3. Remove the last element: (`_elems.pop_back()`)
4. `HeapifyDown(root);`
5. Return `result;`




| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 11 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 | ~~4~~ |

**HeapifyDown**(current)

If ! isLeaf(current)

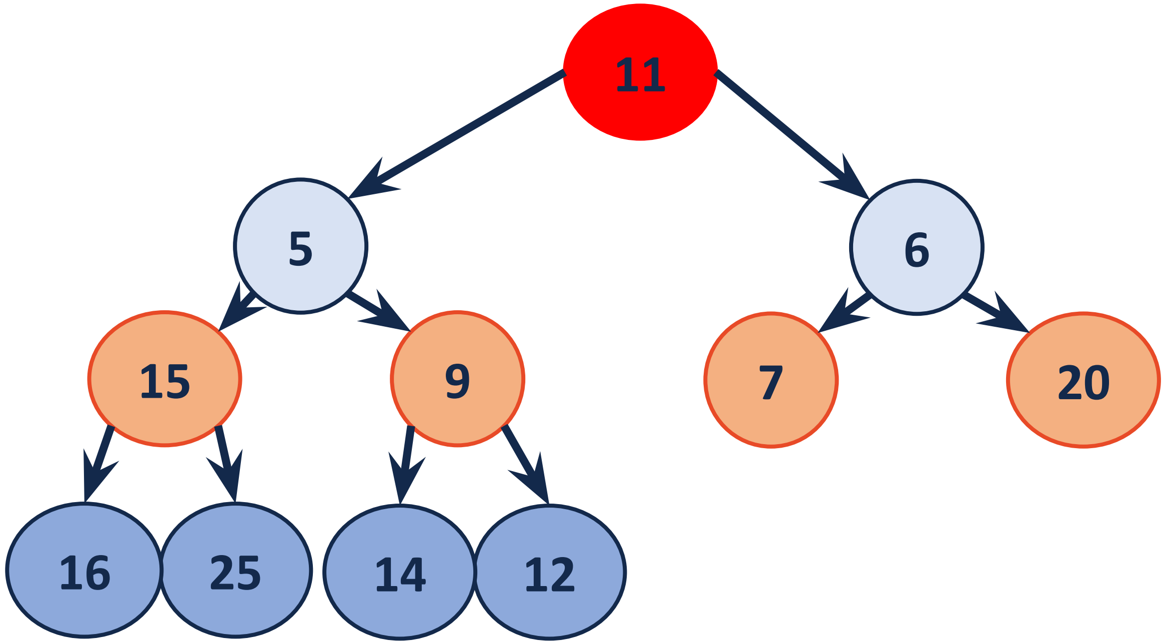
Find the min = index of min child of current

If A[current] > A[min] child

swap A[current] and A[min]

HeapifyDown(min)

//heapifyDown restores heap property only when left subtree and right subtree are already heaps!




| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 11 | 5 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 |

**HeapifyDown**(`current`)

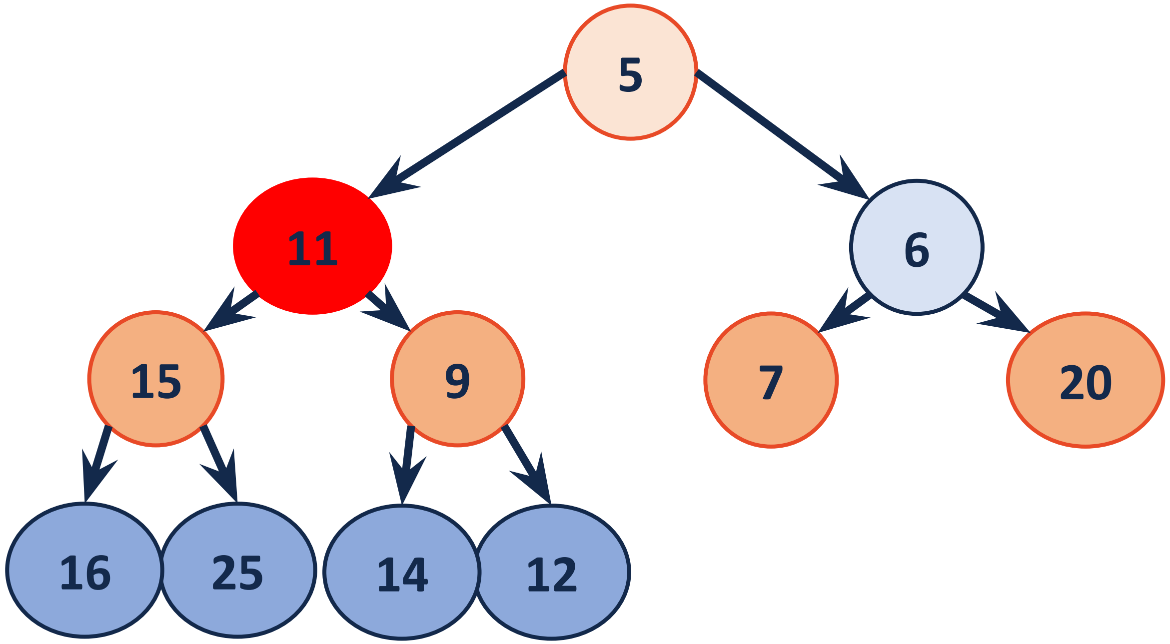
`If ! isLeaf(current)`

Find the `min` = index of min child of `current`

If A[`current`] > A[`min`] child

  swap A[`current`] and A[`min`]

  HeapifyDown(`min`)




| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | 11 | 6 | 15 | 9 | 7 | 20 | 16 | 25 | 14 | 12 |

**HeapifyDown**(`current`)

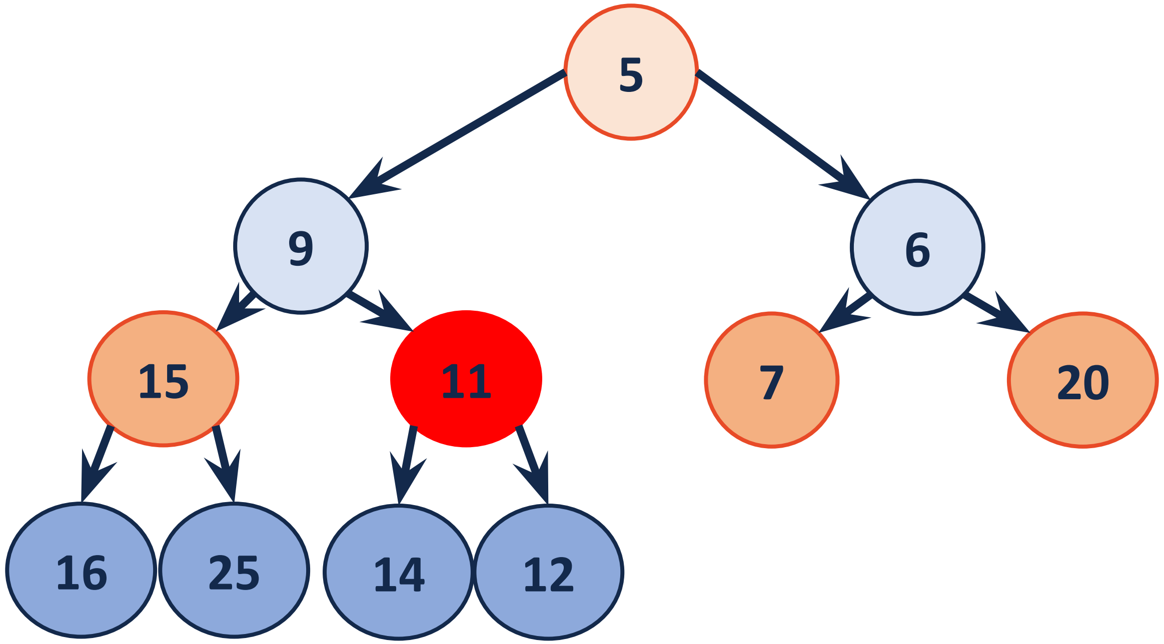
`If ! isLeaf(current)`

Find the `min` = index of min child of `current`

If A[`current`] > A[`min`] child

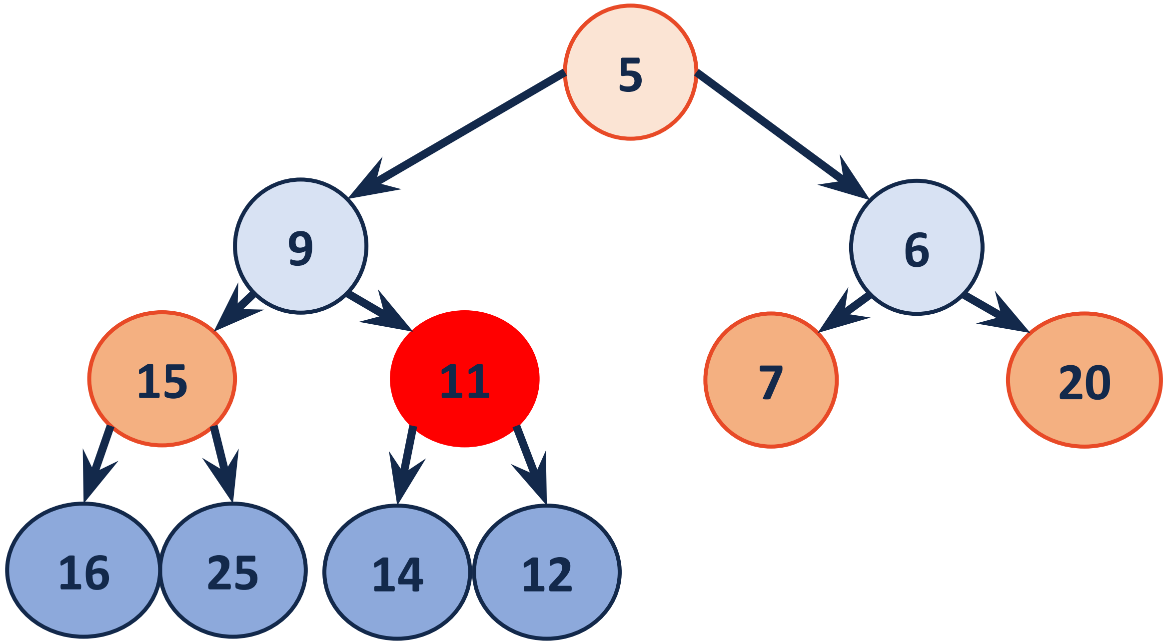
swap A[`current`] and A[`min`]

HeapifyDown(`min`)




| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | 9 | 6 | 15 | 11 | 7 | 20 | 16 | 25 | 14 | 12 |

```
HeapifyDown(current)
If ! isLeaf(current)
Find the min = index of min child of current
If A[current] > A[min] child
    swap A[current] and A[min]
    HeapifyDown(min)
```

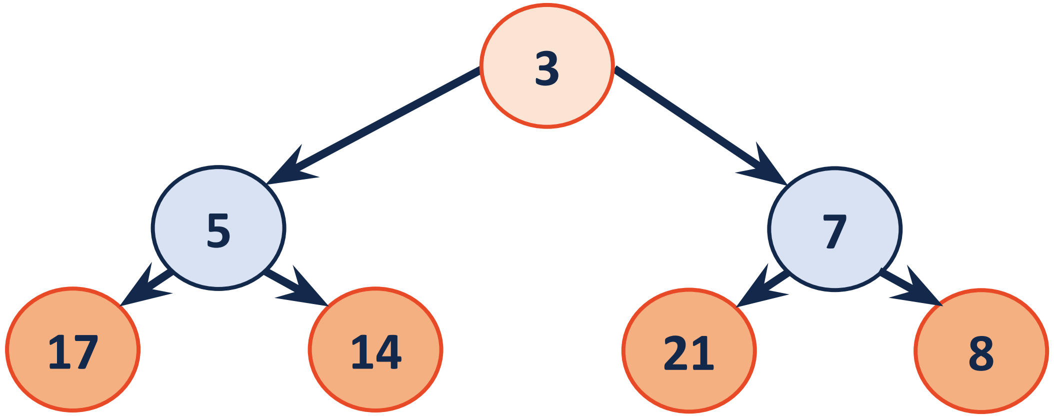



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | 9 | 6 | 15 | 11 | 7 | 20 | 16 | 25 | 14 | 12 |

✓

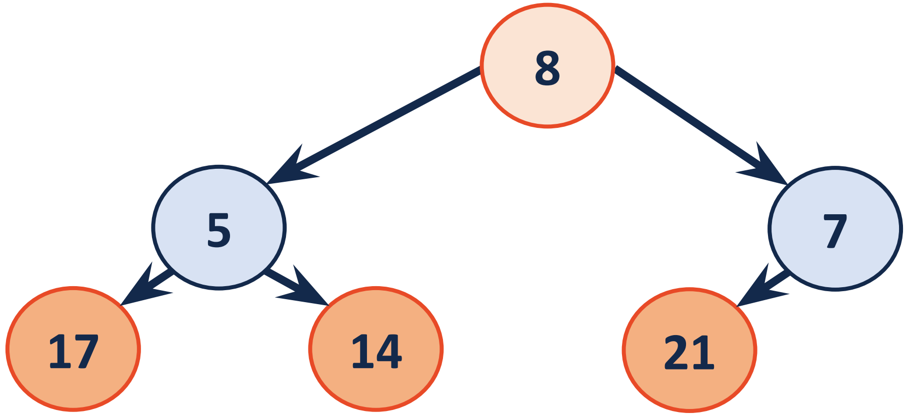
# Worksheet
# Exercise #2.3

**Exercise 2.3:** What is the array representation of the tree if **3** is removed?



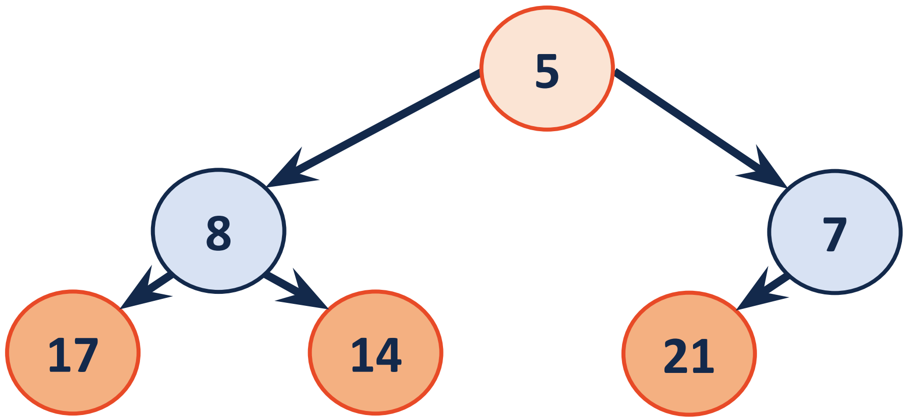

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 3 | 5 | 7 | 17 | 14 | 21 | 8 |

**Exercise 2.3:** What is the array representation of the tree if **3** is removed?




| | | | | | | |
|---|---|---|---|---|---|---|
| | 8 | 5 | 7 | 17 | 14 | 21 |

**Exercise 2.3:** What is the array representation of the tree if **3** is removed?

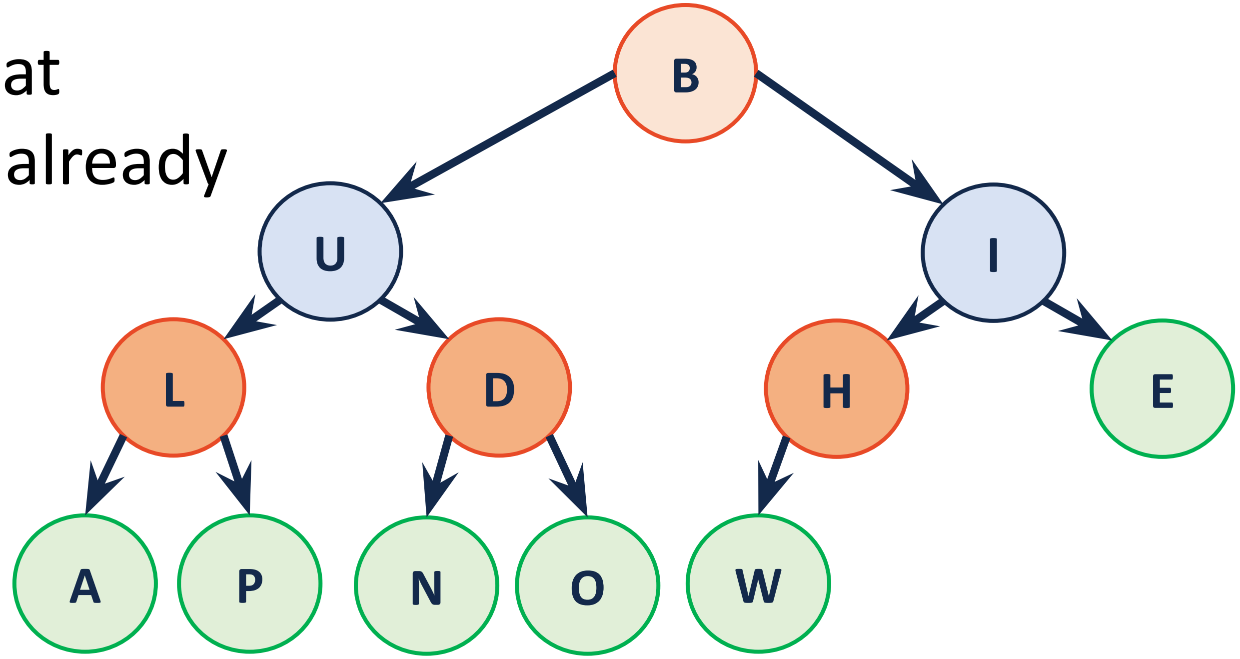



| | | | | | | |
|---|---|---|---|---|---|---|
| | 5 | 8 | 7 | 17 | 14 | 21 |

# Build a Heap

# buildHeap

We can take advantage of the fact that subtrees containing only a leaf node already satisfy heap property! ---- (1)




| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | B | U | I | L | D | H | E | A | P | N | O | W |

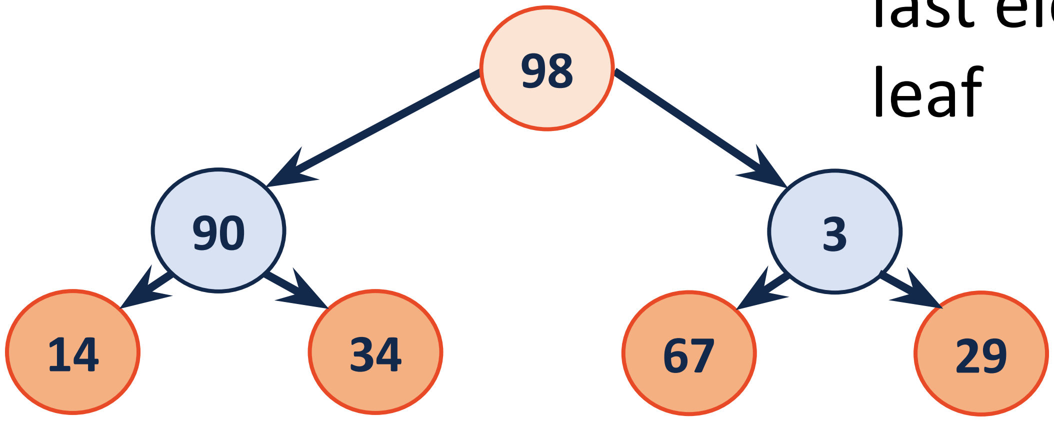
We have to restore heap properties in every subtree started from last element's parent ($i$), because of (1) we can call HeapfyDown on every element started from index $i$ to root!

# Worksheet Exercise #3

**Exercise 3:** After using the buildHeap algorithm on the array below, draw the corresponding tree.

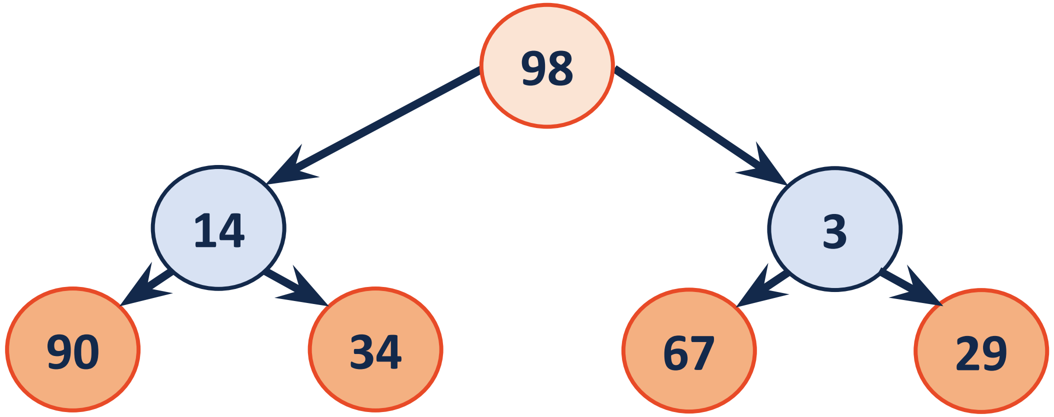
heapifyDown() from the last element that is not a leaf

The right half of the array (All of the nodes that are leaves) is already in a heap structure



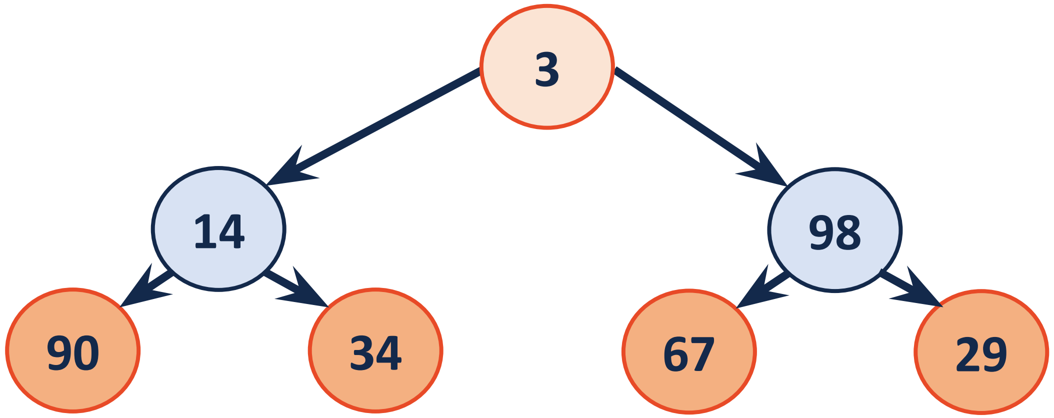

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 98 | 90 | 3 | 14 | 34 | 67 | 29 |

**Exercise 3:** After using the buildHeap algorithm on the array above, draw the corresponding tree.



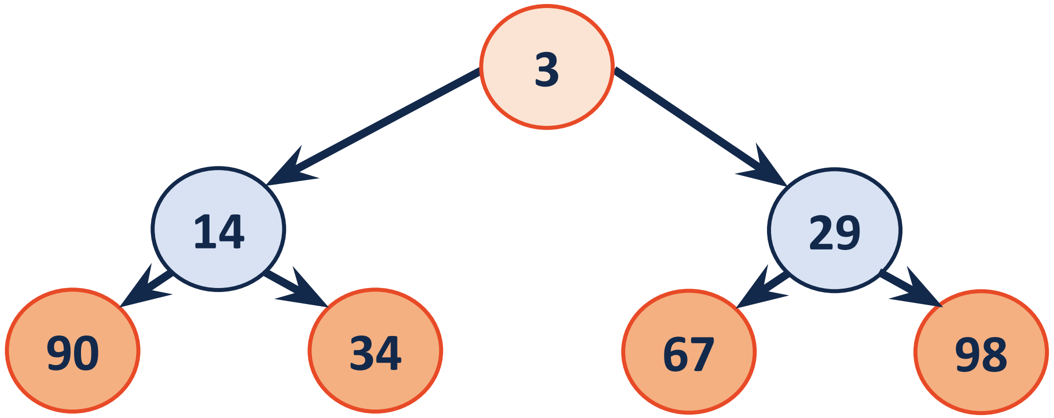

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 98 | 14 | 3 | 90 | 34 | 67 | 29 |

**Exercise 3:** After using the buildHeap algorithm on the array above, draw the corresponding tree.




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 3 | 14 | 98 | 90 | 34 | 67 | 29 |

**Exercise 3:** After using the buildHeap algorithm on the array above, draw the corresponding tree.




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 3 | 14 | 29 | 90 | 34 | 67 | 98 |

✓

Helper function that restores the heap property by bubbling a node up the tree as necessary.

*- already defined for you*

```
heapifyUp(i)
    if i != rootIndex && A[i] < A[parent(i)]
        swap(i, parent(i))
        heapifyUp(parent(i))
```

Helper function that restores the heap property by sinking a node down the tree as necessary.

```
heapifyDown(current)
If ! isLeaf(current)
Find the min = index of min child of current
If A[current] > A[min] child
    swap A[current] and A[min]
    HeapifyDown(min)
```