# Lecture 7: Priority Queue & Heap

**김수경**

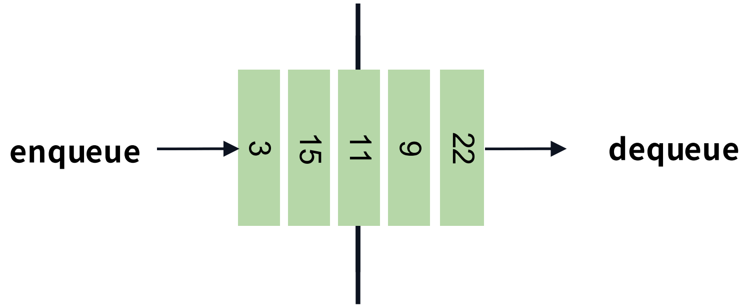
이화여자대학교 인공지능융합전공 소속

# 저작권 안내

upstage Education

## Queue: Recap

- Queue:
  First In First Out (FIFO)
  - Adding a new element (**enqueue**)
  - Retrieving the oldest item (**peek**)
  - Deleting an item (**dequeue**)



| Task | Array-based | Reference-based |
|---|---|---|
| Insertion | O(1) | O(1) |
| Retrieval | O(1) | O(1) |
| Deletion | O(1) | O(1) |

## Priority Queue

- Each element consists of {key (priority), value (element)}
  - Enqueue: Any order
  - Dequeue: Delete by priority
- Example: Airplane boarding group

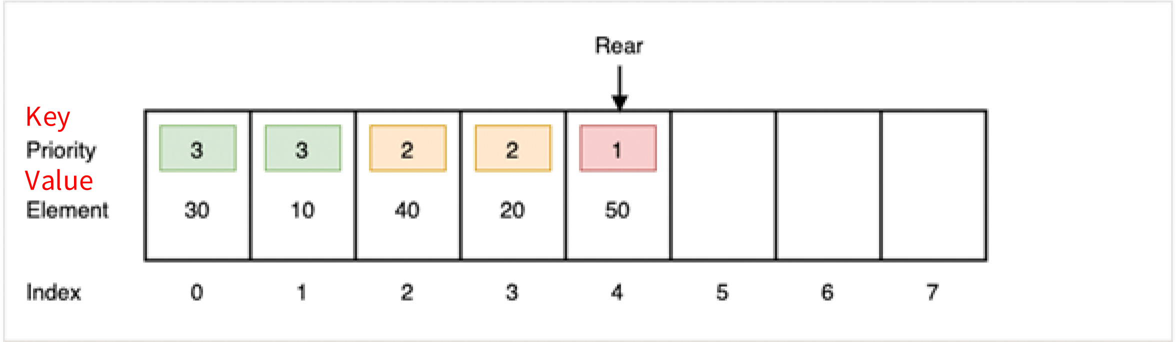
[1]

[1] https://www.usatoday.com/story/todayinthesky/2013/09/30/airlines-promise-a-return-to-civility-for-a-fee/2894645/

## Priority Queue

- Ordered Queue (for faster deletion)
  - Enqueue: O(n)
  - Dequeue: O(1)

[1]

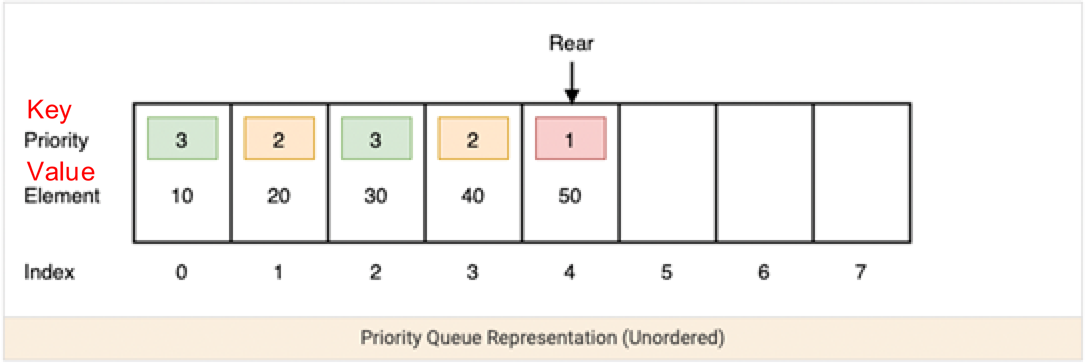


Priority Queue Representation (Ordered)

[1] https://simplerize.com/data-structures/priority-queue

## Priority Queue

- Unordered Queue (for faster insertion)
  - Enqueue: O(1)
  - Dequeue: O(n)

[1] 

- Can we implement enqueue and dequeue in O(log(n))?

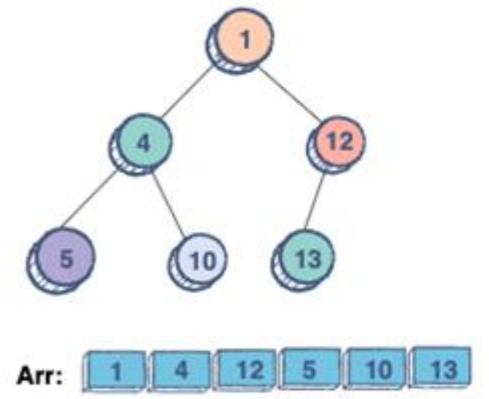
[1] https://simplerize.com/data-structures/priority-queue

01

## Heap

## Binary Heap

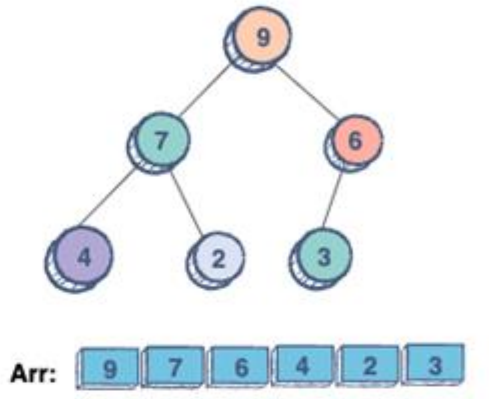
**Min Heap**



[1]

- Heap-Order: $key(v) \geq key(parent(v))$
- Complete Binary Tree

**Max Heap**



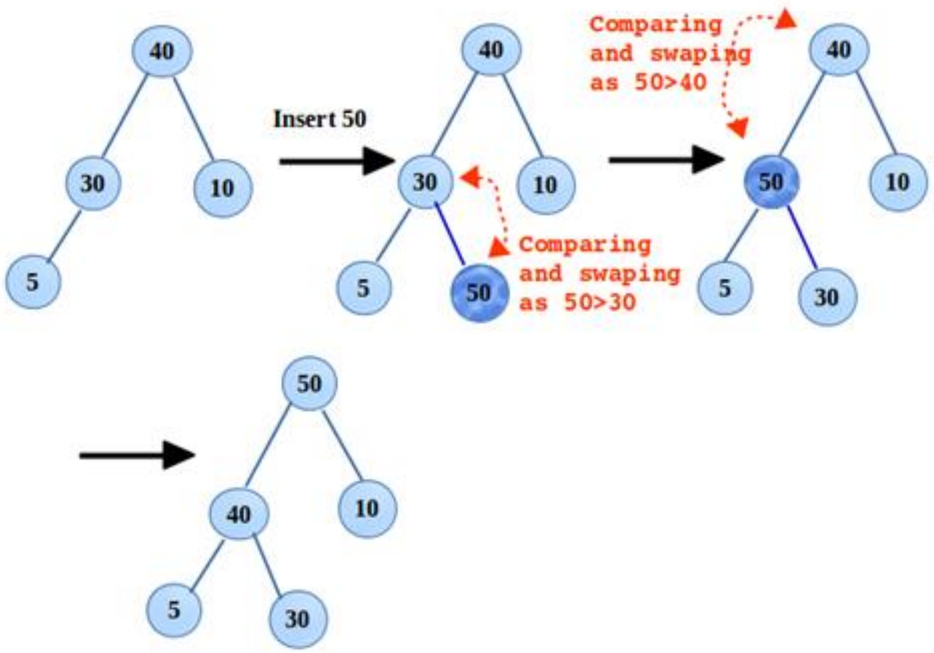
- Heap-Order: $key(v) \leq key(parent(v))$
- Complete Binary Tree

[1] https://junior-datalist.tistory.com/347

## Max-heap: Insertion

- Insertion of a key (priority) k to the heap
- The insertion algorithm consists of 2 steps
  - Store k at the new last node
  - Restore the heap-order property (heapify_up)

## Max-heap: Insertion


Insert 50
Comparing and swaping as 50>30
Comparing and swaping as 50>40
40
30
10
5
50

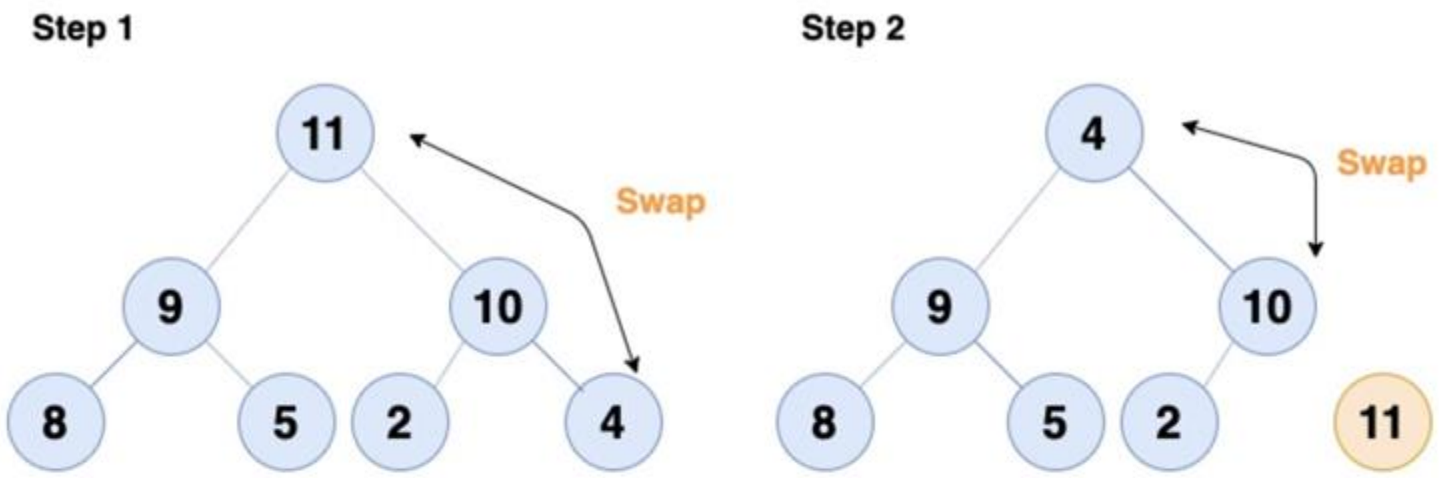
[1]

[1] \https://www.codingeek.com/data-structure/binary-heap-introduction-explanation-and-implementation/

## Max-Heap: Deletion

- Removal of the root key (highest priority = maximum number) from the heap
- The removal algorithm consists of 2 steps
  - Replace the root key with the key of the last node.
  - Restore the heap-order property (Heapify-down)

## Max-Heap: Deletion



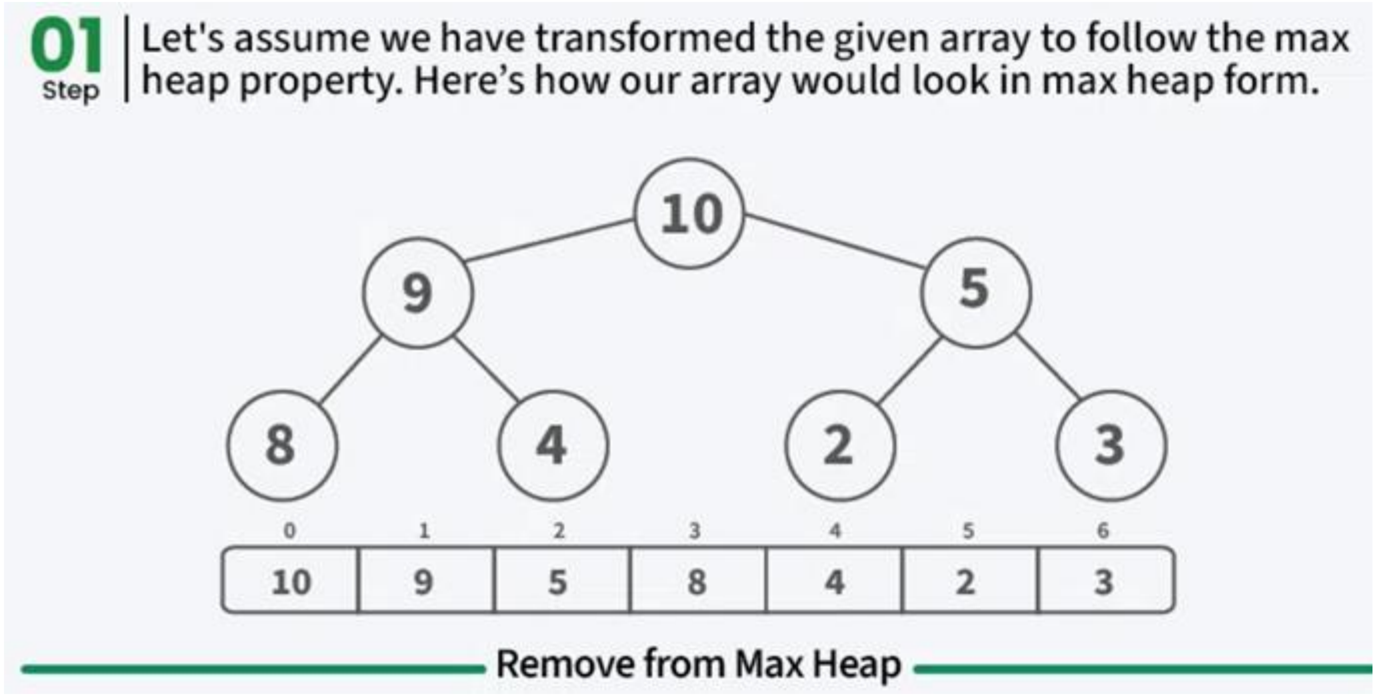
[1]

[1] https://afteracademy.com/blog/heap-building-and-heap-sort/

## Max Heap: Heap sort

- **Build a Max-Heap**:
  - Convert the array into a max-heap, ensuring the largest element is at the root.
  - Complexity: O(n).
- **Extract Maximum**:
  - Swap the root (largest element) with the last element.
  - Reduce heap size by 1 and restore the max-heap property (heapify-down).
  - Repeat until all elements are sorted.
  - Complexity: O(nlogn).

## Max Heap: Heap sort



**01** Step | Let's assume we have transformed the given array to follow the max heap property. Here's how our array would look in max heap form.

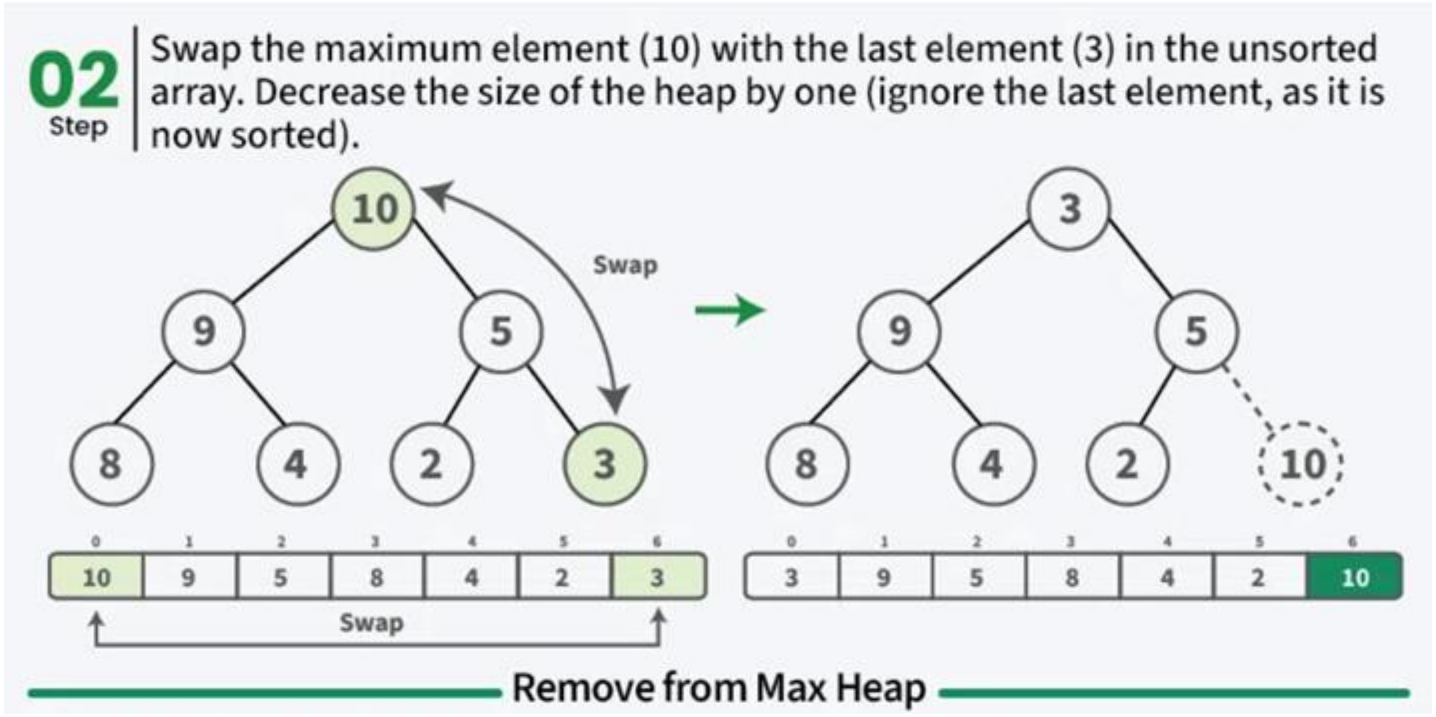
|  0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 10 | 9 | 5 | 8 | 4 | 2 | 3 |

Remove from Max Heap

[1]

[1] https://www.geeksforgeeks.org/c/c-program-for-heap-sort/

## Max Heap: Heap sort



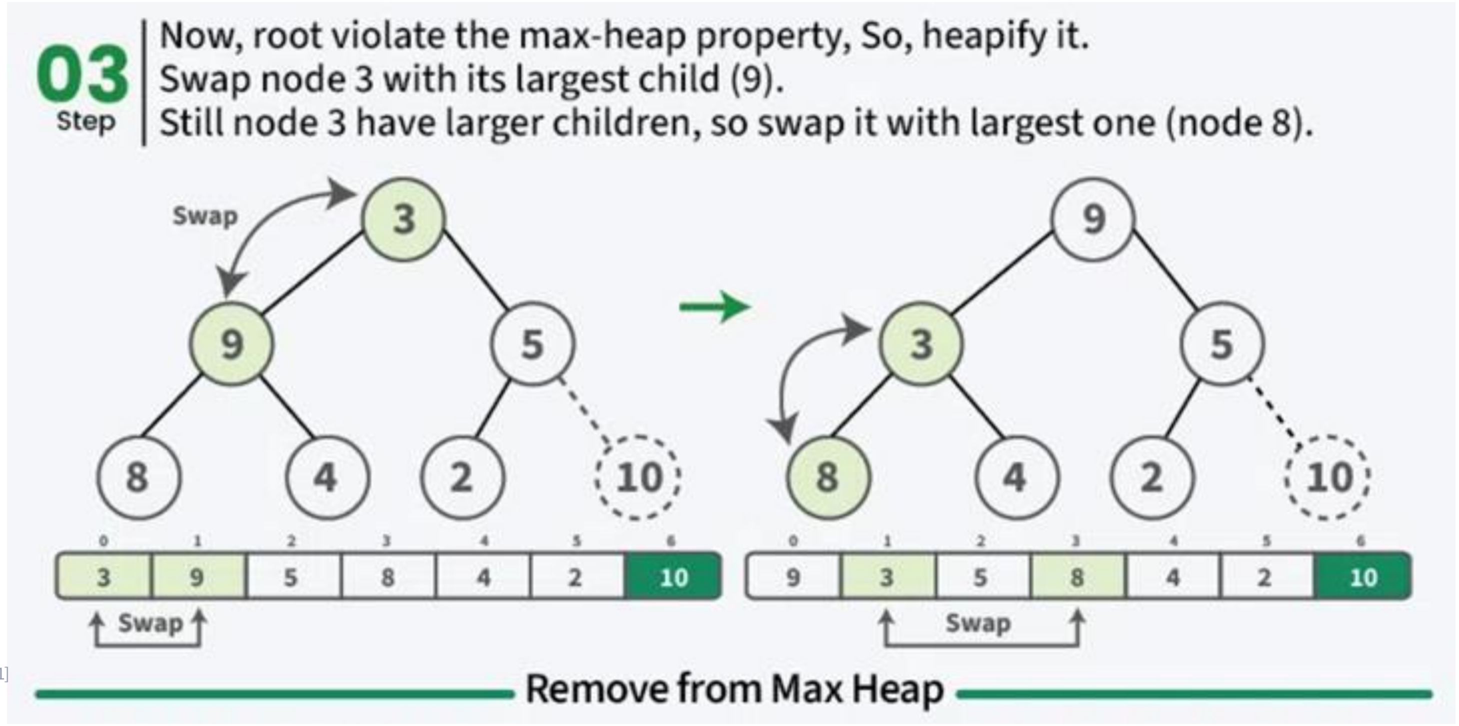
Swap the maximum element (10) with the last element (3) in the unsorted array. Decrease the size of the heap by one (ignore the last element, as it is now sorted).

[1]

[1] https://www.geeksforgeeks.org/c/c-program-for-heap-sort/

upstage Education

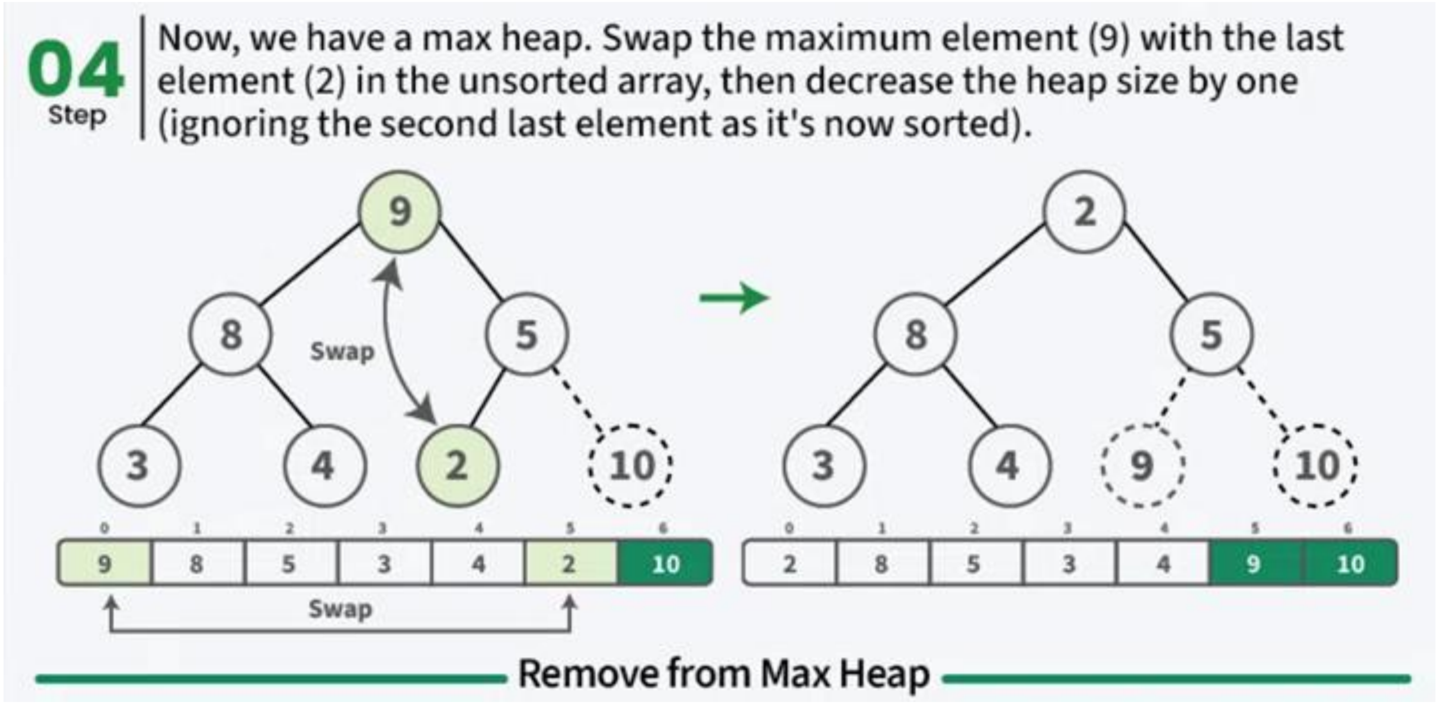
## Max Heap: Heap sort


03
Step
Now, root violate the max-heap property, So, heapify it.
Swap node 3 with its largest child (9).
Still node 3 have larger children, so swap it with largest one (node 8).
Swap
3
9
5
8
4
2
10
0 1 2 3 4 5 6
3 9 5 8 4 2 10
Swap
9
3
5
8
4
2
10
0 1 2 3 4 5 6
9 3 5 8 4 2 10
Swap
Remove from Max Heap

[1]

[1] https://www.geeksforgeeks.org/c/c-program-for-heap-sort/

# Max Heap: Heap sort


**04** Step

Now, we have a max heap. Swap the maximum element (9) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the second last element as it's now sorted).

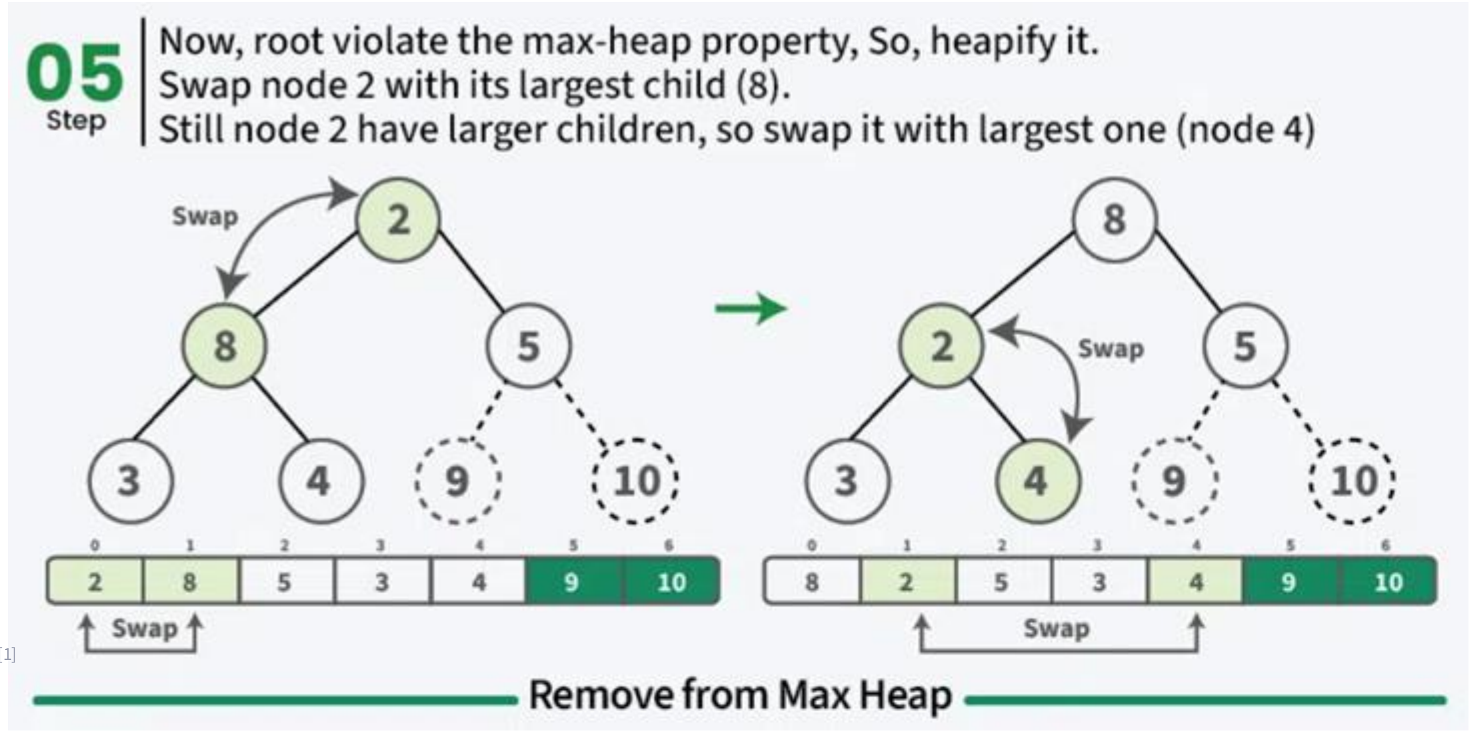
Remove from Max Heap

[1]

[1] https://www.geeksforgeeks.org/c/c-program-for-heap-sort/

upstage Education

## Max Heap: Heap sort

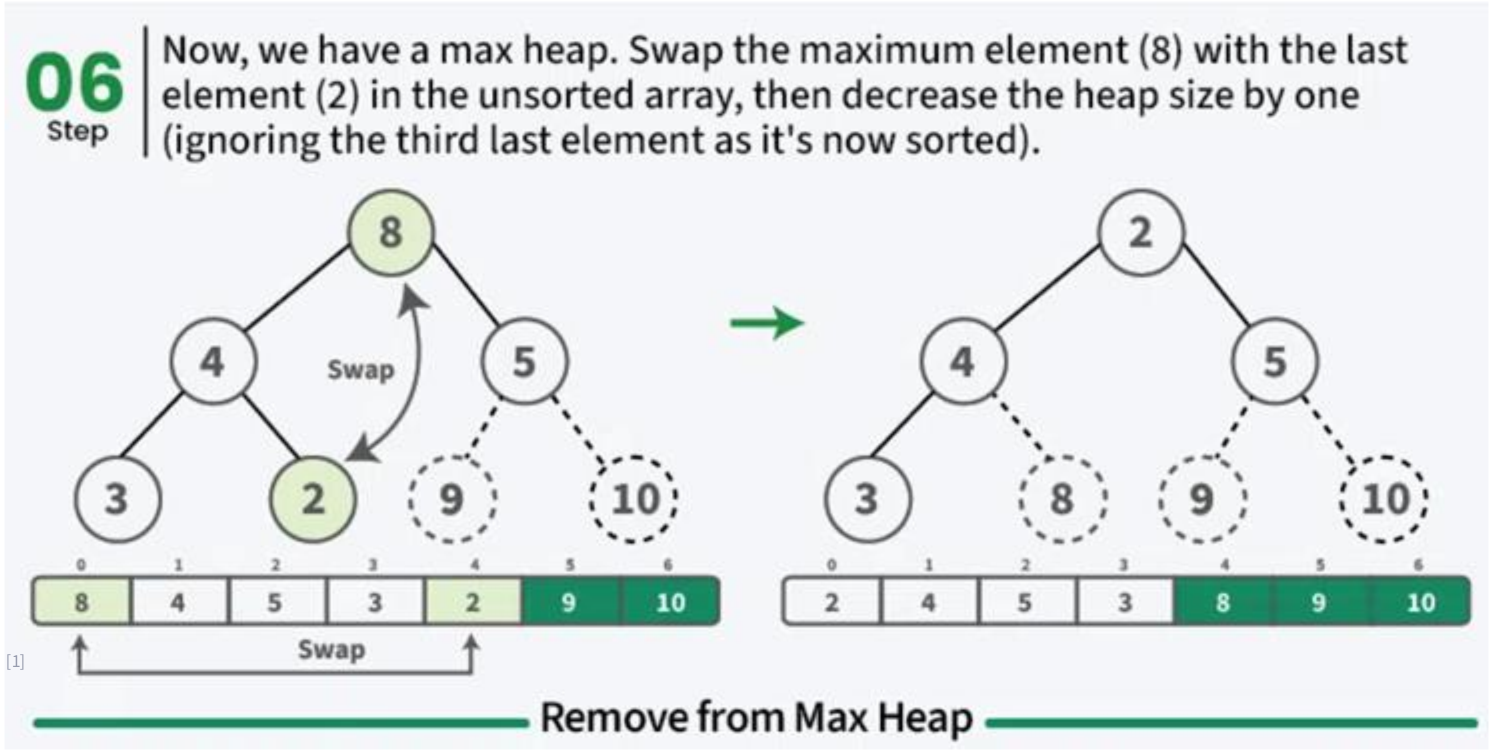

05
Step
Now, root violate the max-heap property, So, heapify it.
Swap node 2 with its largest child (8).
Still node 2 have larger children, so swap it with largest one (node 4)
Swap
2
8
5
3
4
9
10
0 1 2 3 4 5 6
2 8 5 3 4 9 10
Swap
8
2
Swap
5
3
4
9
10
0 1 2 3 4 5 6
8 2 5 3 4 9 10
Swap
Remove from Max Heap

[1]

[1] https://www.geeksforgeeks.org/c/c-program-for-heap-sort/

upstage Education

## Max Heap: Heap sort


06
Step
Now, we have a max heap. Swap the maximum element (8) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the third last element as it's now sorted).
8
4
5
3
2
9
10
Swap
2
4
5
3
8
9
10
0 1 2 3 4 5 6
8 4 5 3 2 9 10
Swap
2 4 5 3 8 9 10
Remove from Max Heap

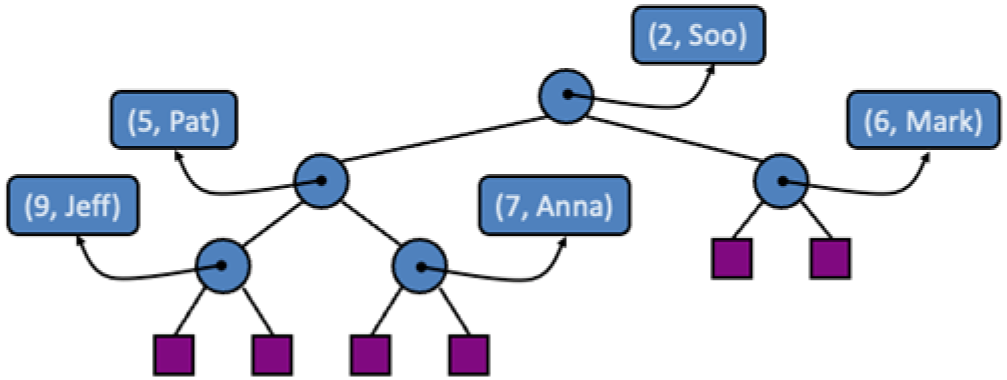
[1]

[1] https://www.geeksforgeeks.org/c/c-program-for-heap-sort/

upstage Education

## Heap and Priority Queue

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the picture

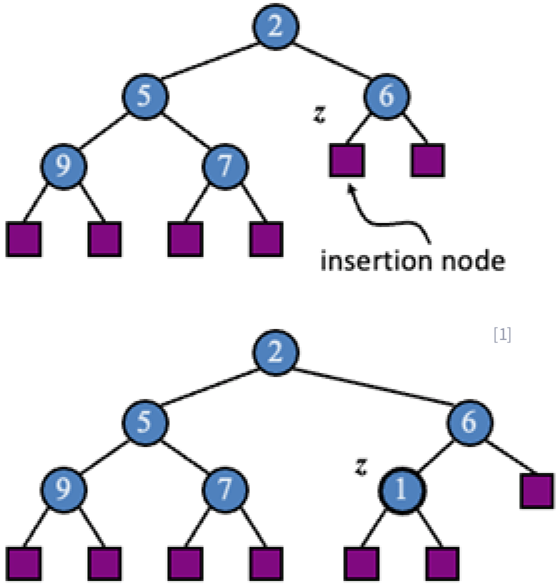


[1]

[1] https://nikolay.kirov.be/2025/CITB308/slides/ch07/ch07_2.html

## Insertion

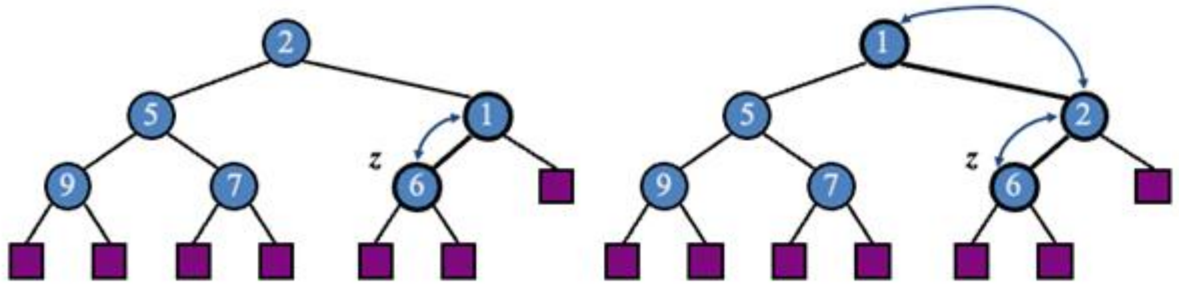
- Insertion of a key (priority) k to the heap
- The insertion algorithm consists of 2 steps
  - Store k at the new last node
  - Restore the heap-order property (heapify_up: next slide)



## Heapify-up

- After the insertion of a new key $k$, the heap-order property may be violated
- Algorithm heapify-up restores the heap-order property by swapping $k$ along an upward path from the insertion node
- Heapify-up terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$
- Since a heap has height $O(\log n)$, heapify-up runs in $O(\log n)$ time

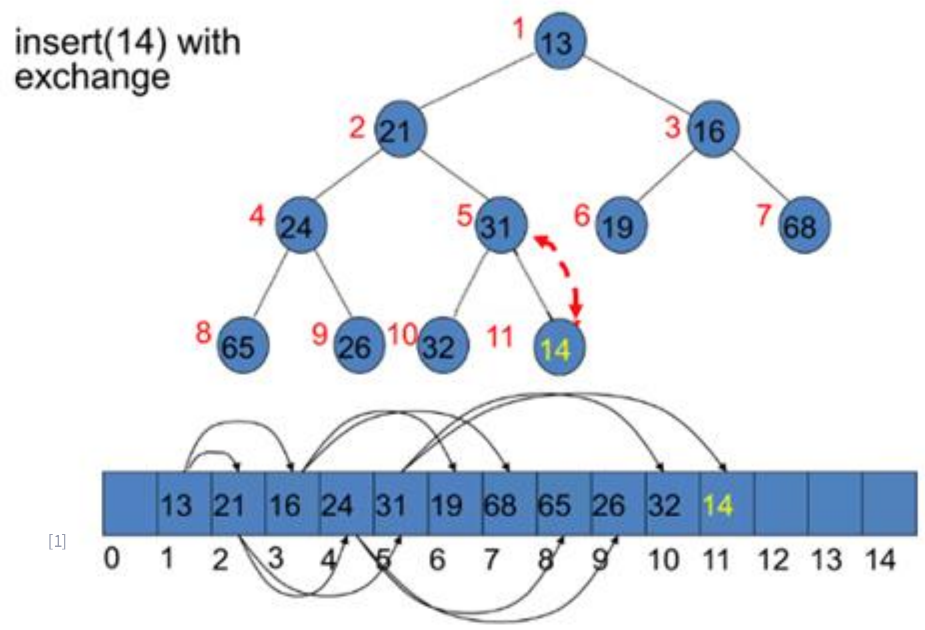
[1]



[1] https://nikolay.kirov.be/2025/CITB308/slides/ch07/ch07_2.html

upstage Education

## Insertion example

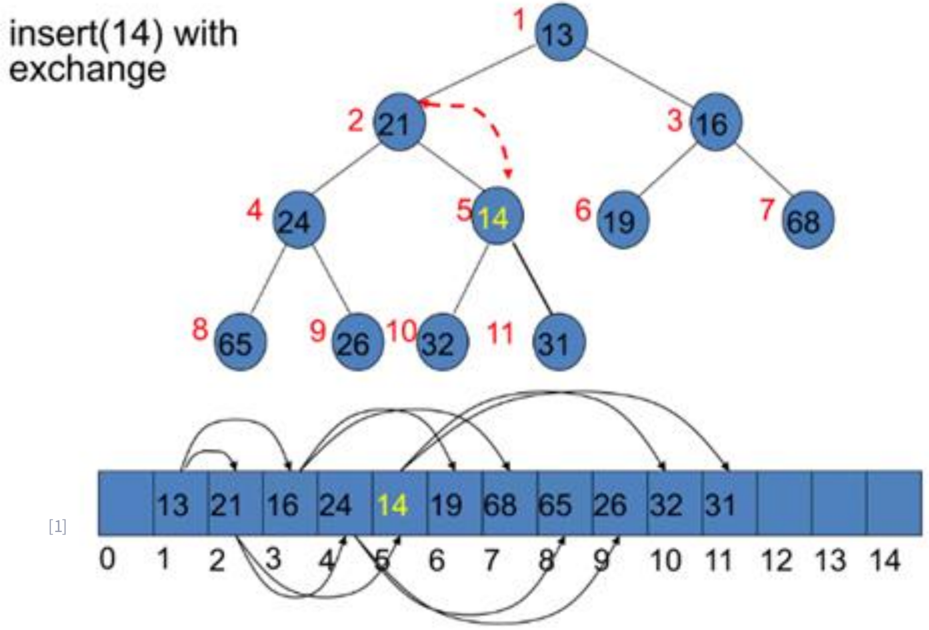


[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Insertion example

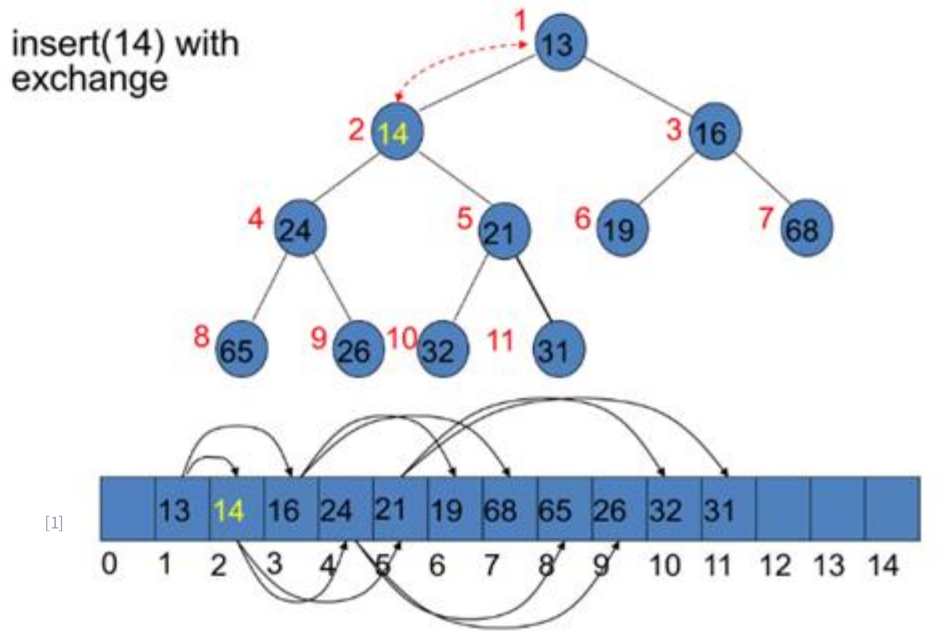


[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Insertion example

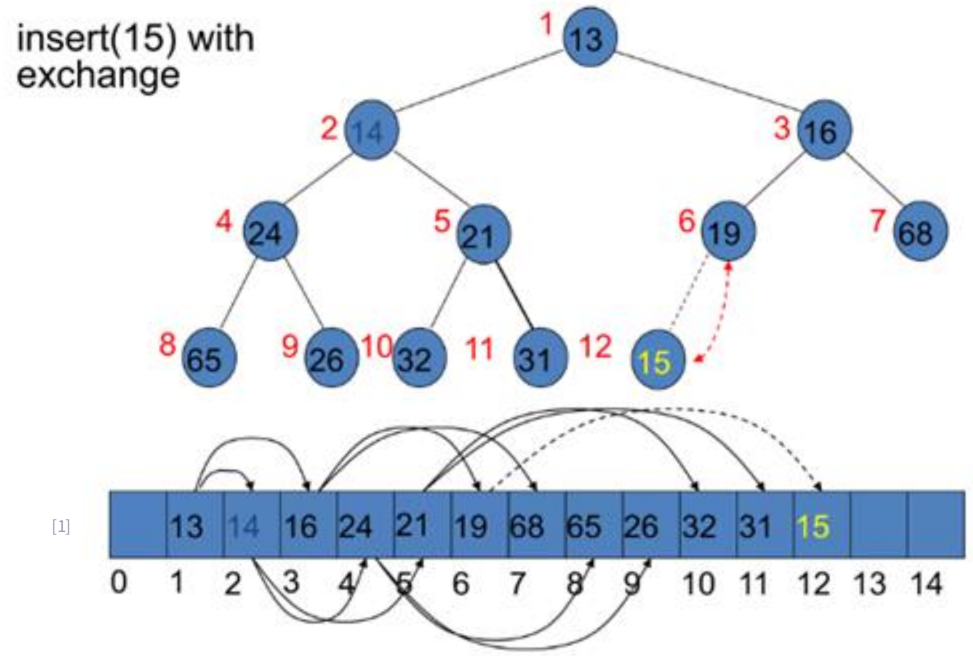


[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Insertion example

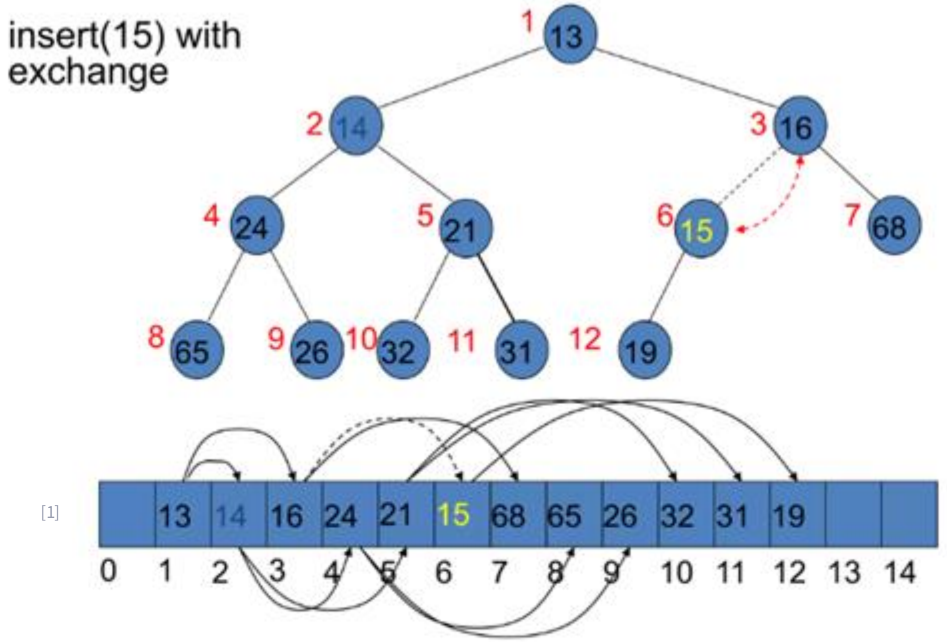


[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Insertion example

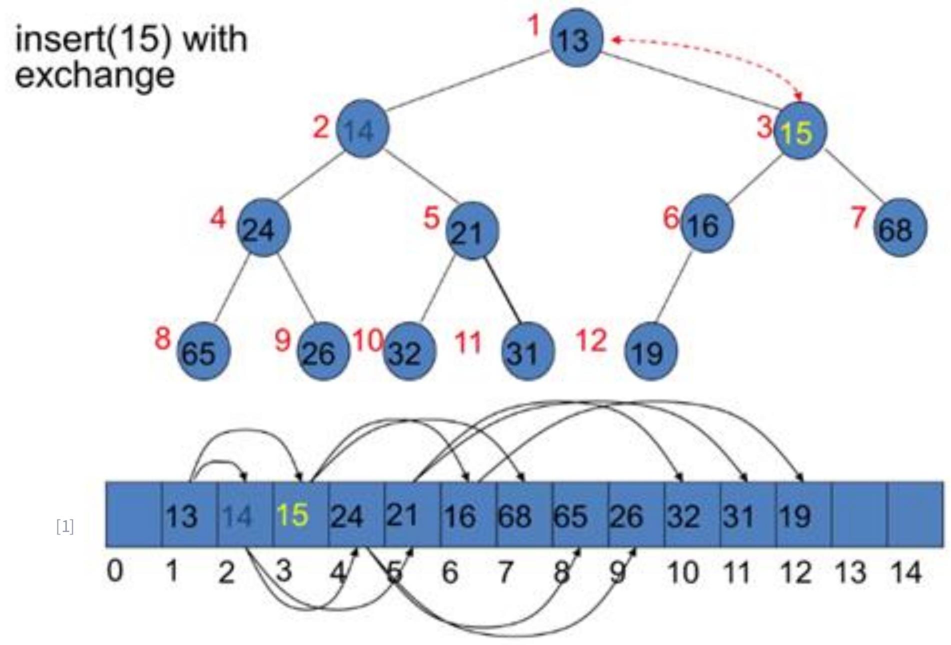


[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Insertion example

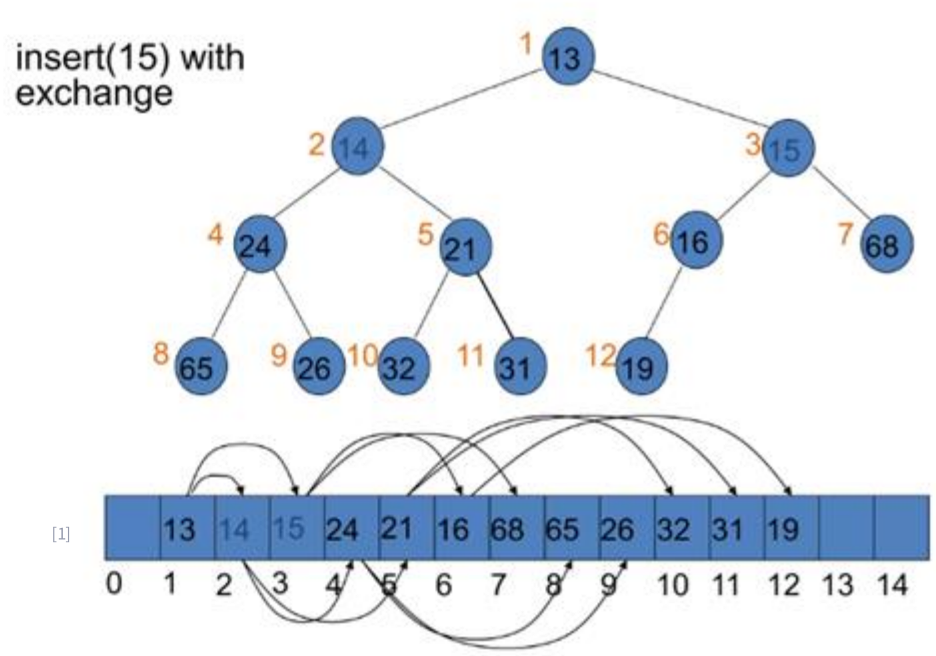


[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Insertion example



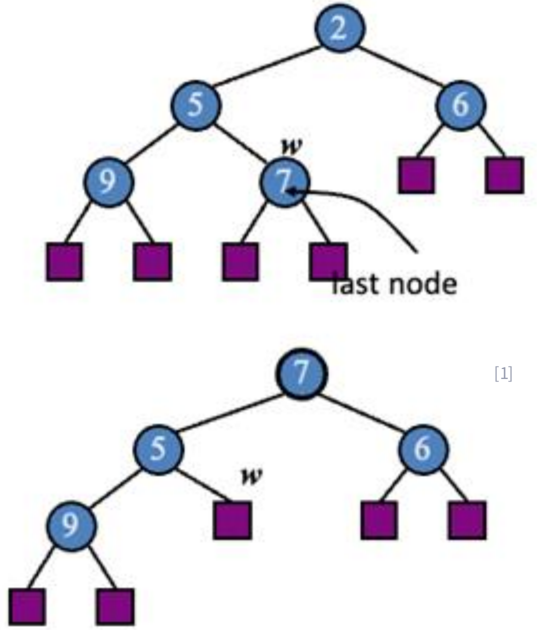
[1]

- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

[1] https://www.slideshare.net/slideshow/lec-8ds-algoheapspdf/251538564

## Deletion

- Removal of the root key (highest priority) from the heap
- The removal algorithm consists of 2 steps
  - Replace the root key with the key of the last node.
  - Restore the heap-order property (Heapify-down: next)

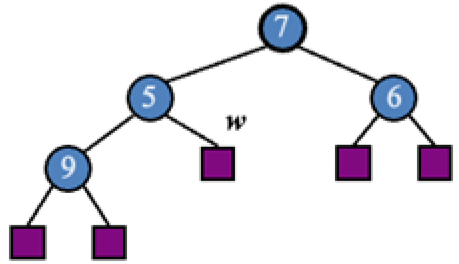


[1]

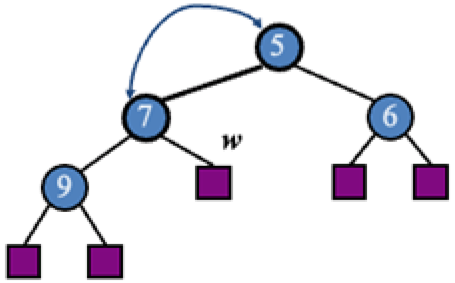
[1] https://nikolay.kirov.be/2025/CITB308/slides/ch07/ch07_2.html

## Heapify-down

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm heapify-down restores the heap property by swapping key k with the child with the *smallest key* along a downward path from the root
- Heapify-down terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height O(log n), heapify-down runs in O(log n) time



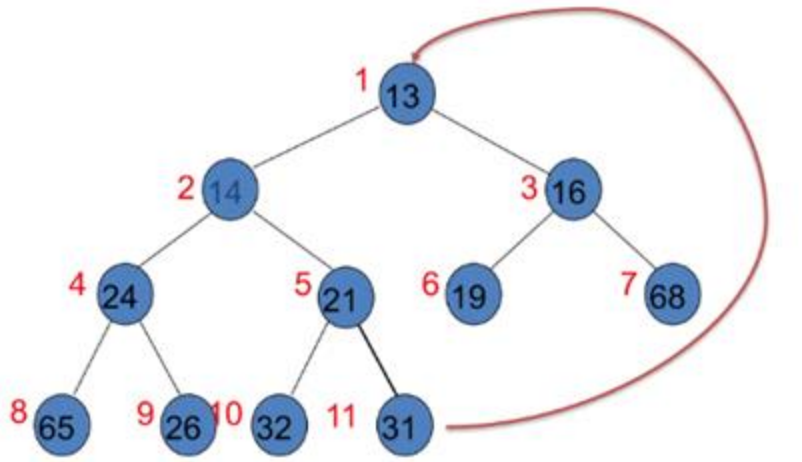
[1]



upstage Education

## Deletion Example

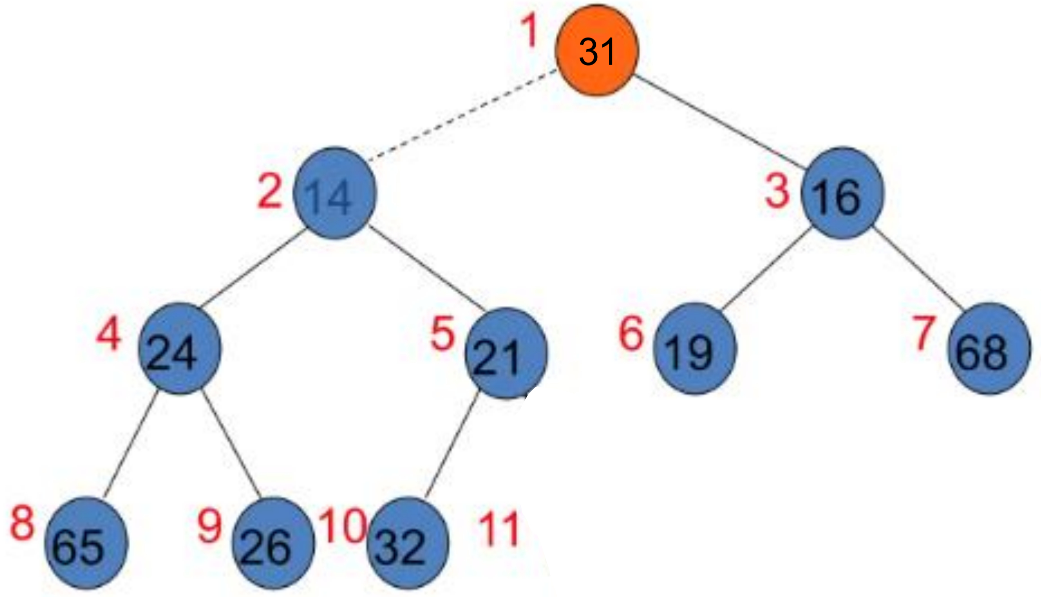
- Deletion in Max(or Min) Heap always happens at the root to remove Maximum (or Minimum) value
- Deleting it (or removing it) from root cause a hole which needs to be filled.



- left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- parent:
  `[child index - 1] // 2`

upstage Education

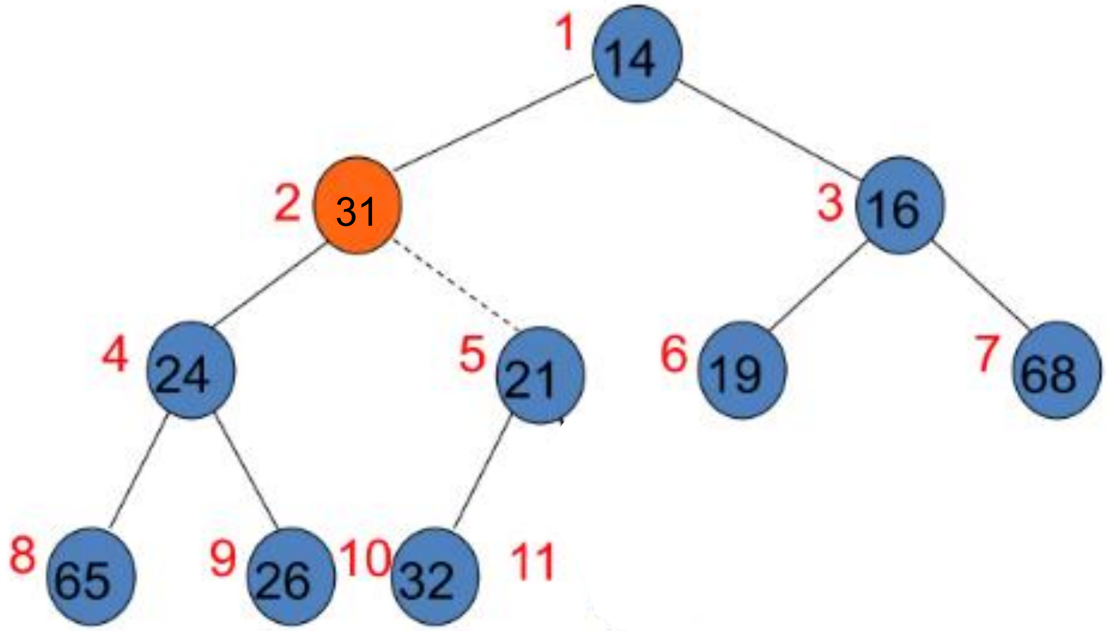
## Deletion Example



- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

upstage Education

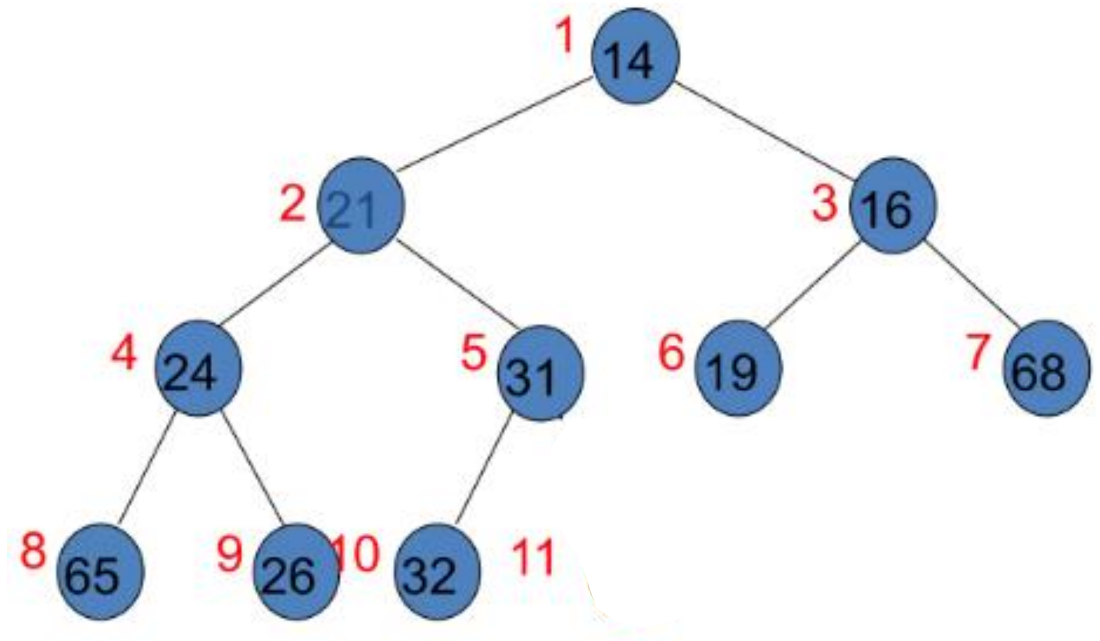
## Deletion Example



- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

upstage Education

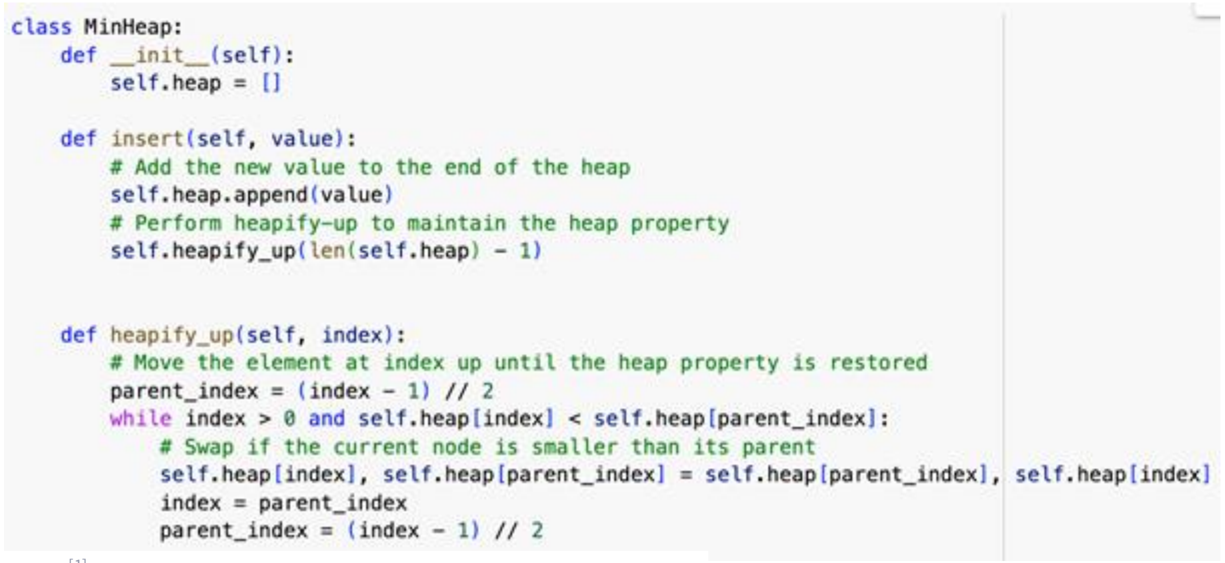
## Deletion Example



- Left child:
  `[parent index] * 2 + 1`
- Right child:
  `[parent index] * 2 + 2`
- Parent:
  `[child index - 1] // 2`

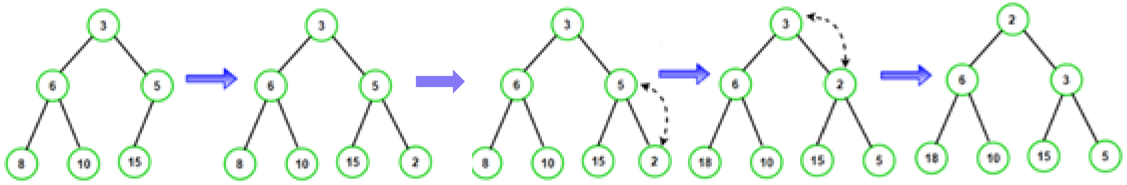
upstage Education

## Implementation (array)



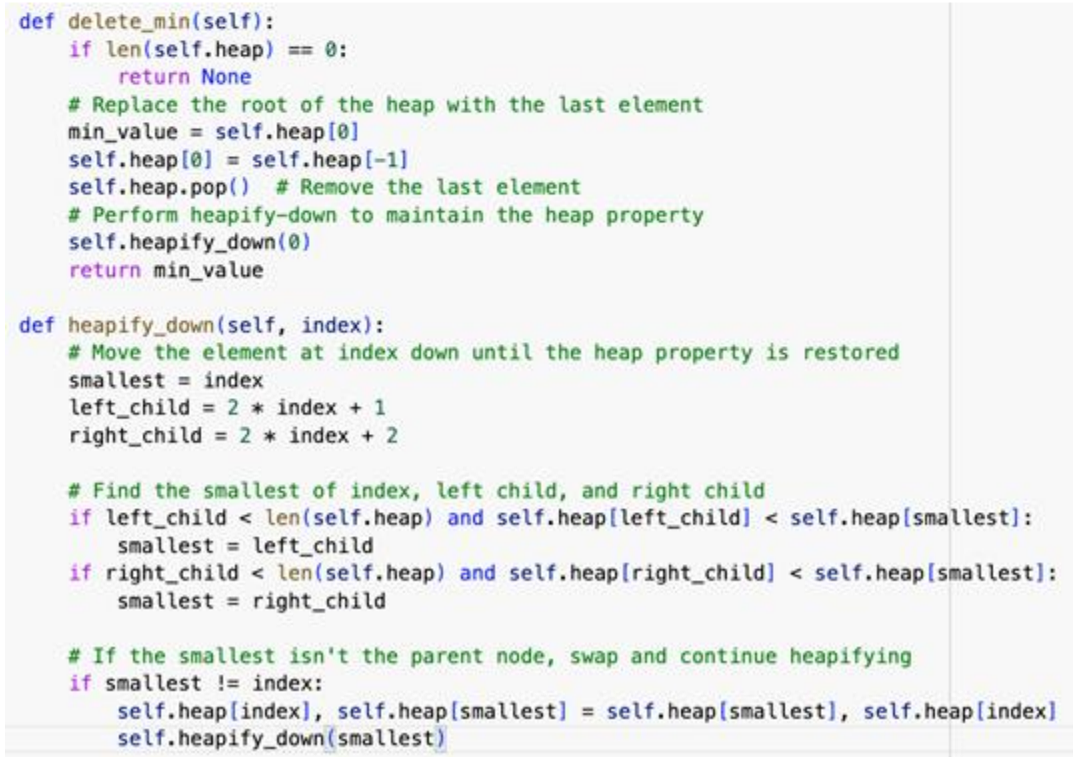
```
class MinHeap:
    def __init__(self):
        self.heap = []

    def insert(self, value):
        # Add the new value to the end of the heap
        self.heap.append(value)
        # Perform heapify-up to maintain the heap property
        self.heapify_up(len(self.heap) - 1)


    def heapify_up(self, index):
        # Move the element at index up until the heap property is restored
        parent_index = (index - 1) // 2
        while index > 0 and self.heap[index] < self.heap[parent_index]:
            # Swap if the current node is smaller than its parent
            self.heap[index], self.heap[parent_index] = self.heap[parent_index], self.heap[index]
            index = parent_index
            parent_index = (index - 1) // 2
```

[1]



[1] https://velog.io/@wlgns2223/leetcode-215-Kth-Largest-Element-in-an-Array

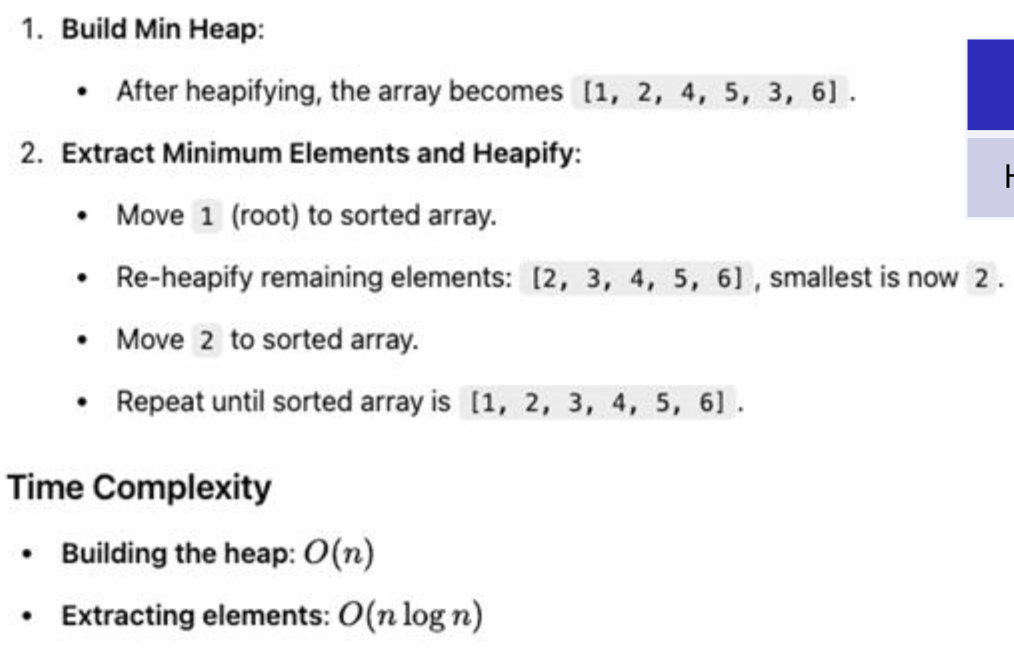
## Implementation (array)



```python
def delete_min(self):
    if len(self.heap) == 0:
        return None
    # Replace the root of the heap with the last element
    min_value = self.heap[0]
    self.heap[0] = self.heap[-1]
    self.heap.pop()  # Remove the last element
    # Perform heapify-down to maintain the heap property
    self.heapify_down(0)
    return min_value

def heapify_down(self, index):
    # Move the element at index down until the heap property is restored
    smallest = index
    left_child = 2 * index + 1
    right_child = 2 * index + 2

    # Find the smallest of index, left child, and right child
    if left_child < len(self.heap) and self.heap[left_child] < self.heap[smallest]:
        smallest = left_child
    if right_child < len(self.heap) and self.heap[right_child] < self.heap[smallest]:
        smallest = right_child

    # If the smallest isn't the parent node, swap and continue heapifying
    if smallest != index:
        self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
        self.heapify_down(smallest)
```

upstage Education

## Heap-sort (Min heap)



1. **Build Min Heap:**
   - After heapifying, the array becomes `[1, 2, 4, 5, 3, 6]`.
2. **Extract Minimum Elements and Heapify:**
   - Move `1` (root) to sorted array.
   - Re-heapify remaining elements: `[2, 3, 4, 5, 6]`, smallest is now `2`.
   - Move `2` to sorted array.
   - Repeat until sorted array is `[1, 2, 3, 4, 5, 6]`.

### Time Complexity

- **Building the heap:** $O(n)$
- **Extracting elements:** $O(n \log n)$

| | Insert | Remove Min | PQ-Sort Total |
|---|---|---|---|
| Heap Sort | O(logn) | O(logn) | O(n logn) |

***O***(***n*** log ***n***) Sorting!

upstage

# Building intelligence for the future of work