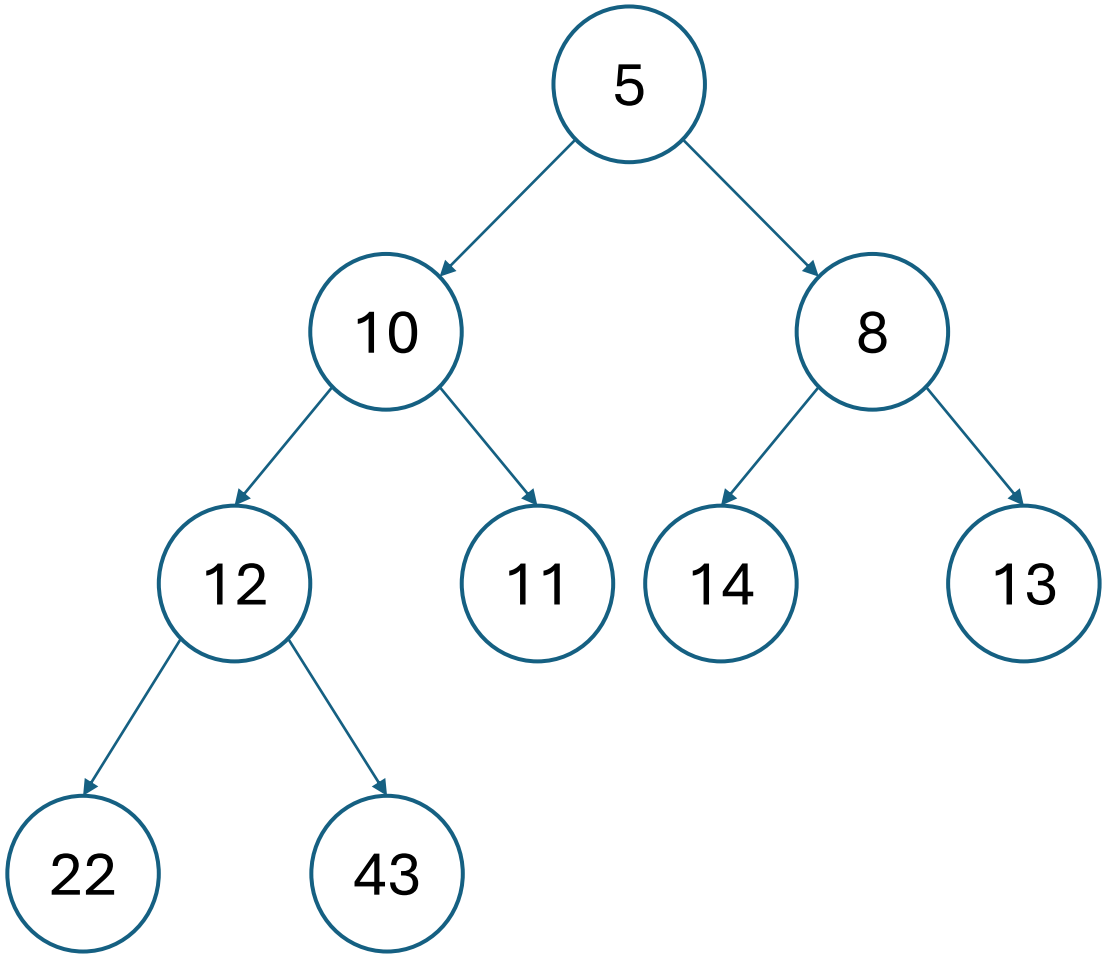
Binary Heaps

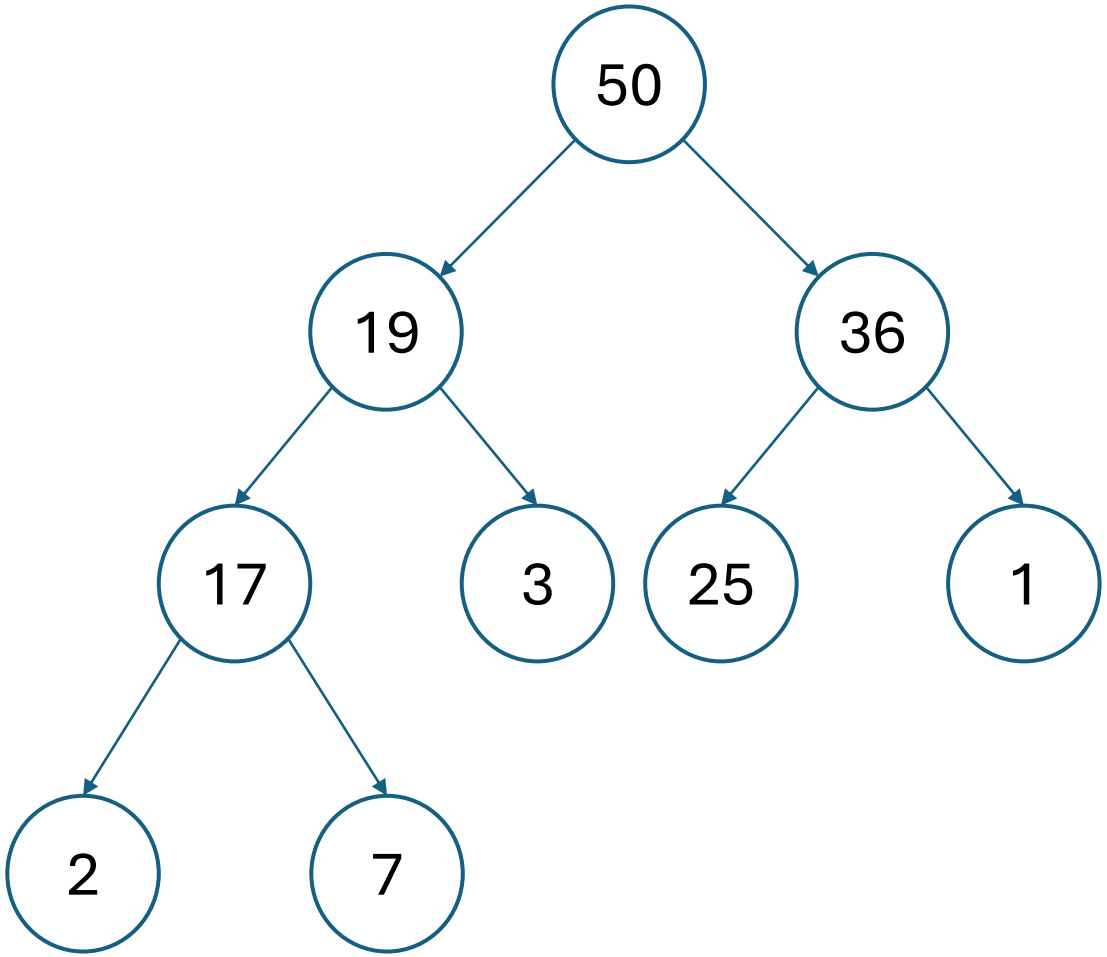
- A **binary heap is a complete binary tree** with one (or both) of the following heap order properties:
  - **MinHeap property**: Each node must have a key that is less than or equal to the key of each of its children.
  - **MaxHeap property**: Each node must have a key that is greater than or equal to the key of each of its children.

Heap property: **Parents have a higher *priority* than any of their children**

- A binary heap satisfying the **MinHeap property** is called a **MinHeap**
- A binary heap satisfying the **MaxHeap property** is called a **MaxHeap**
- A binary heap with **all keys equal is both a MinHeap and a MaxHeap**
- **Stable heap** is a heap data structure where elements with the same value or priority maintain their original relative order when extracted.
- **Multi-key heap** is a heap data structure that allows elements to be associated with multiple keys.

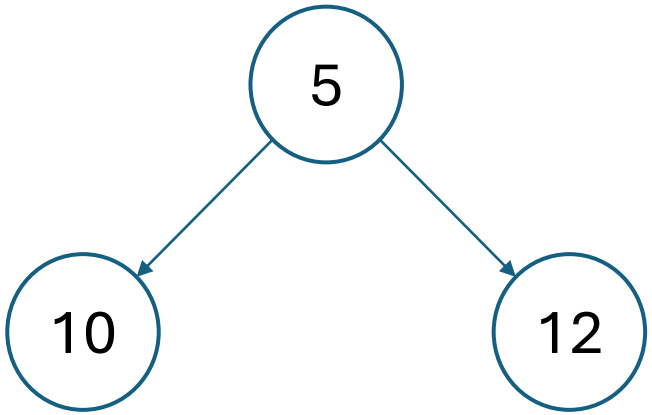


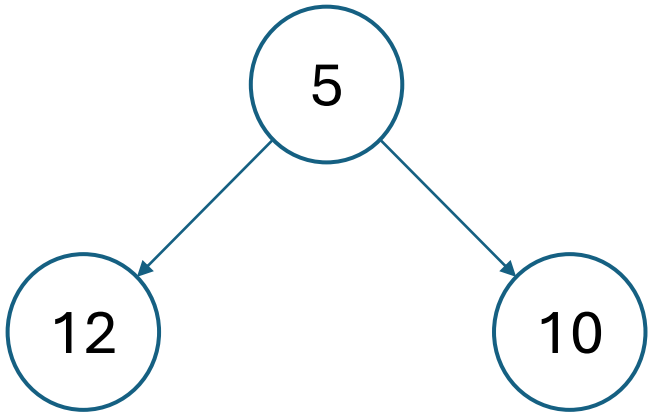
- A min-heap has the smallest element at the root
- A "higher priority" is a smaller number. Thus, 5 is a higher priority than 10.



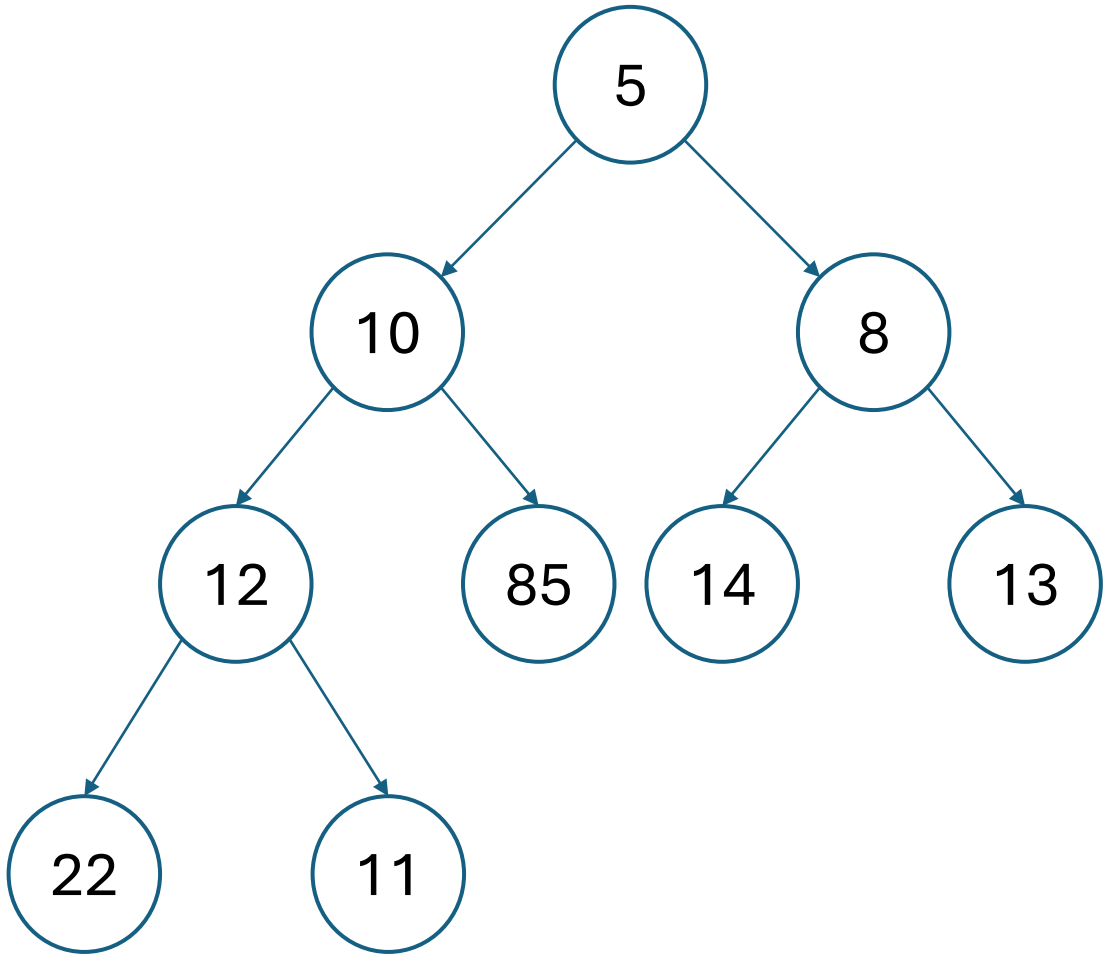
- A max-heap has the largest element at the root
- A "higher priority" is a larger number. Thus, 50 is a higher priority than 19.

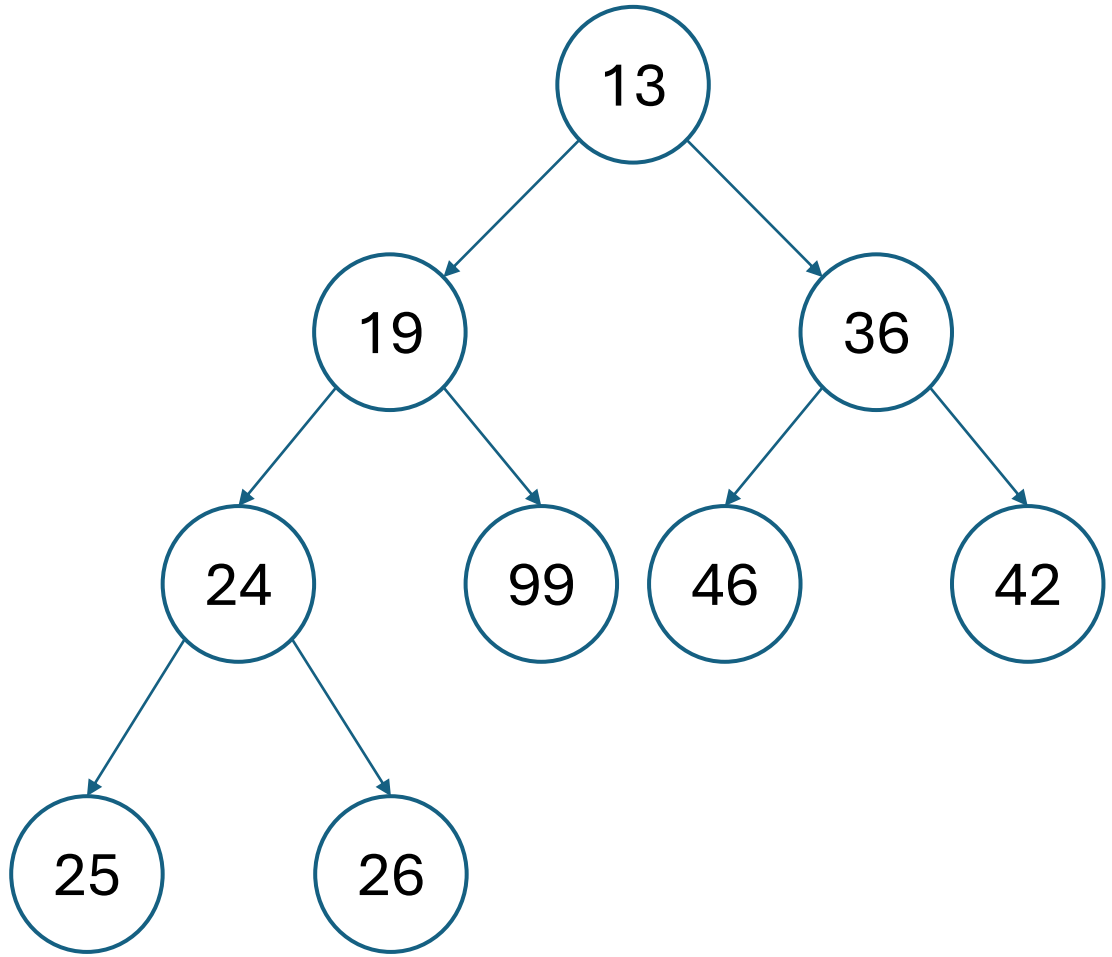
There are no implied orderings between siblings, so both of the trees below are min-heaps



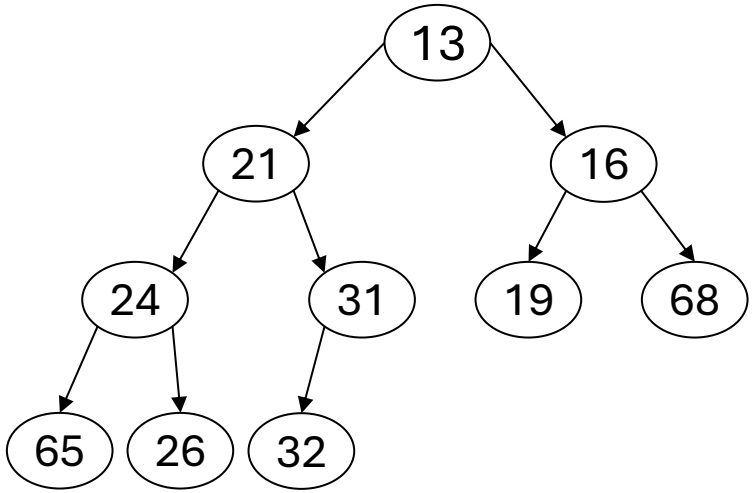


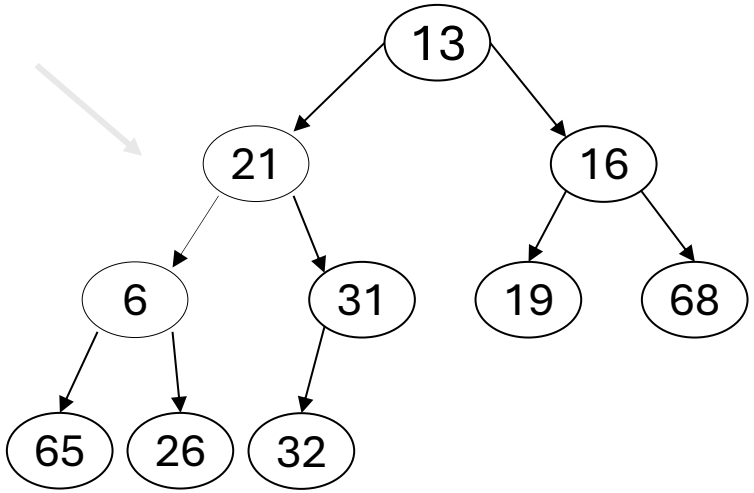
Which of the following are min-heaps?

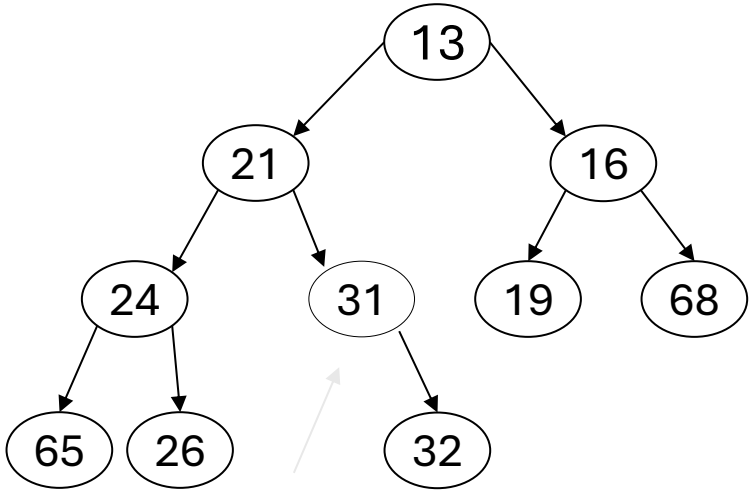


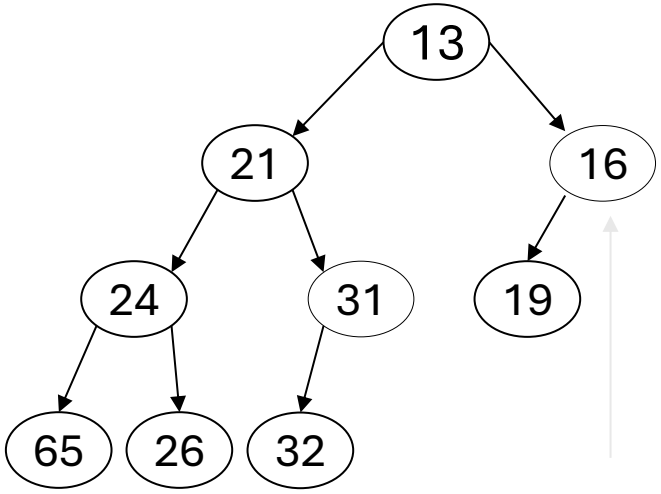


Answer: Only the second one is a heap, because in the first one, the 11 is a child of the 12, and therefore it breaks the heap property that a parent has to be a higher priority than its children (remember: min-heaps designate higher priority as lower numbers).


13
21
16
24
31
19
68
65
26
32

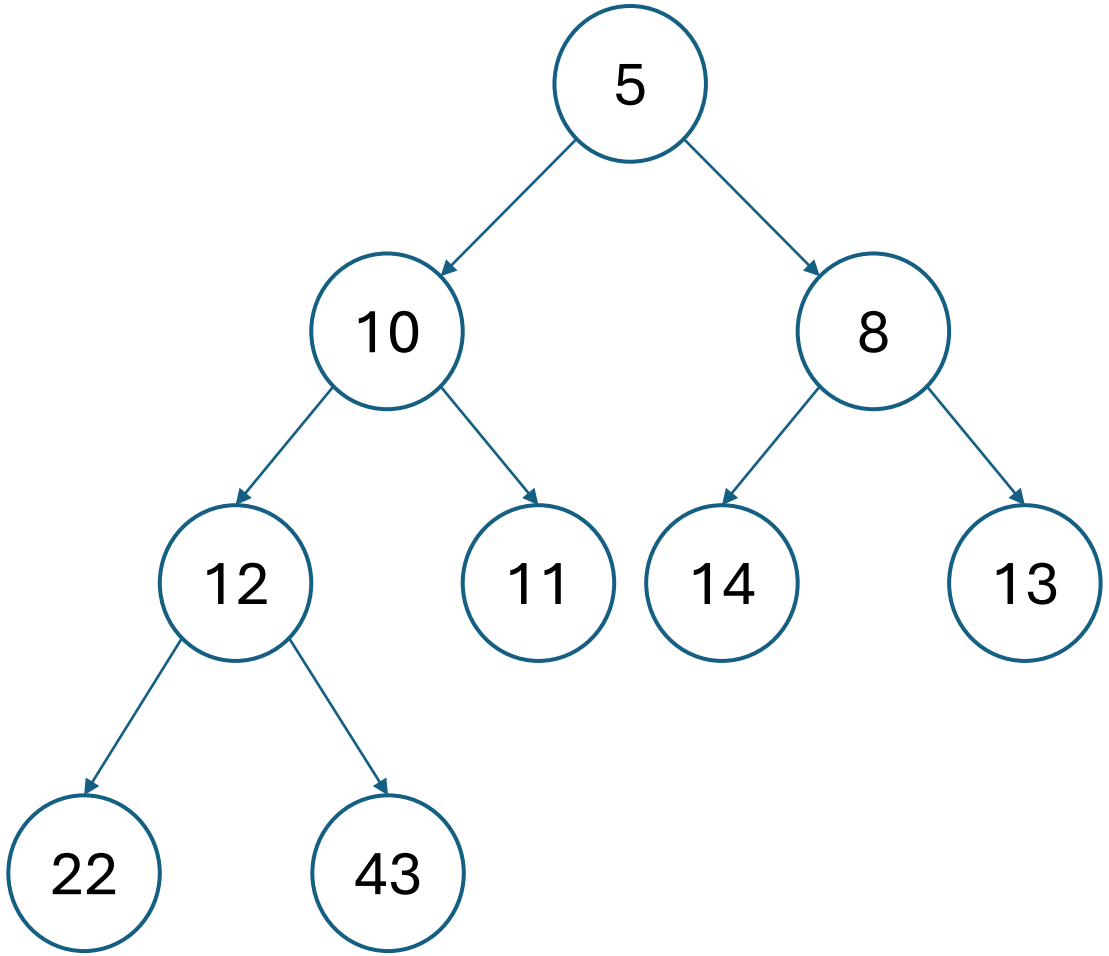

13
21
16
6
31
19
68
65
26
32


13
21
16
24
31
19
68
65
26
32


13
21
16
24
31
19
65
26
32

How do you store a heap?

- Array works great for storing a binary heap
- Reason: Complete nature of the structure, with all levels filled from left to right.



Values:	**5**	**10**	**8**	**12**	**11**	**14**	**13**	**22**	**43**	**?**	**?**	**?**
Index:	0	1	2	3	4	5	6	7	8	9	10	11

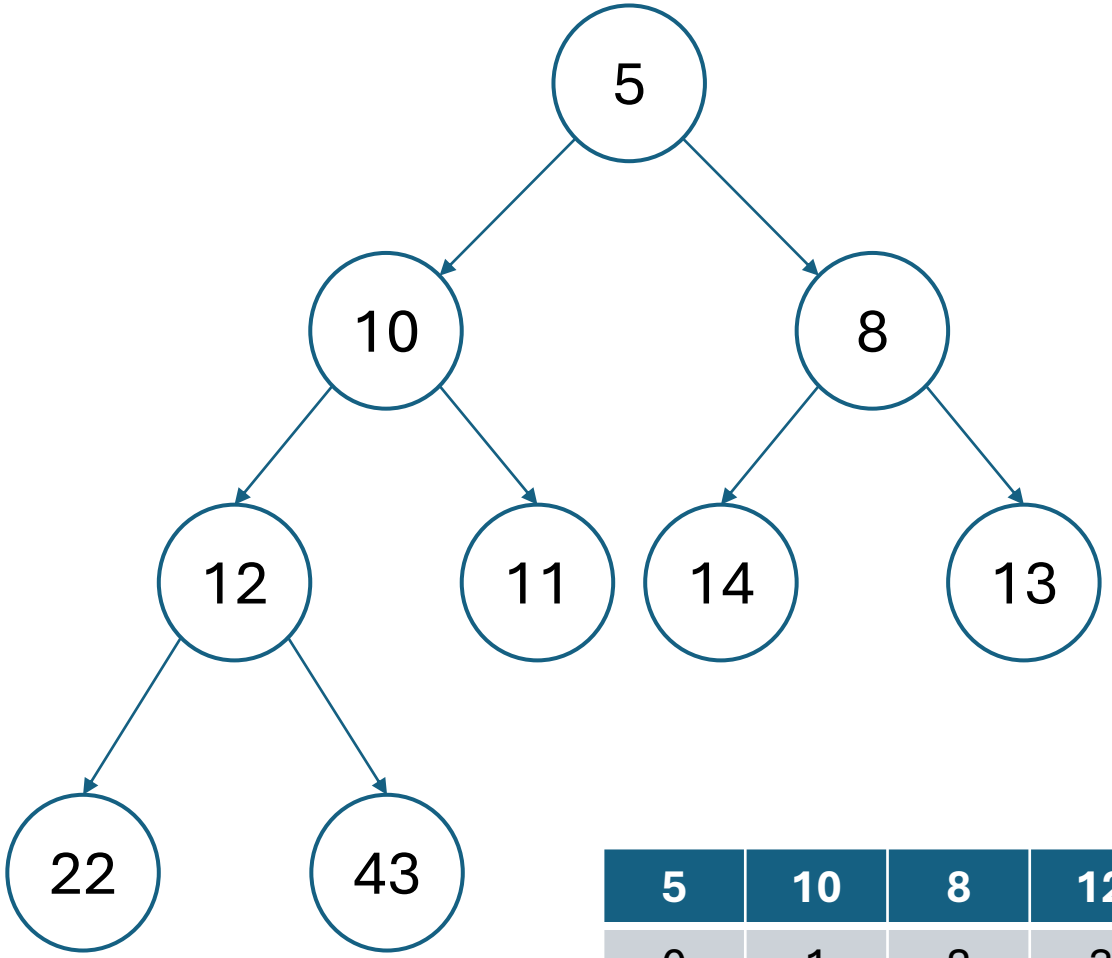
- Array follows the levels, 5, then 10, 8, then 12, 11, 14, 13, then 22, 43 – this is the way we fill a heap.
- The array representation makes determining parents and children a matter of simple arithmetic:
  - For an element at position i:
    - The left child is at 2 * i + 1
    - The right child is at 2 * i + 2
    - The parent is at (i - 1) / 2 (integer math)
- heapSize is the number of elements in the heap (for this heap, heapSize is 9).

Heap Operations

There are three important operations in a heap:

- **peek()**: return the element with the highest priority (lowest number for a min-heap). Don't change the heap at all.
- **enqueue(e)**: insert an element ‘e’ into the heap but retain the heap property!
- **dequeue()**: remove the highest priority (smallest element for a min-heap) from the heap. This changes the heap, and we must return it to a proper heap.

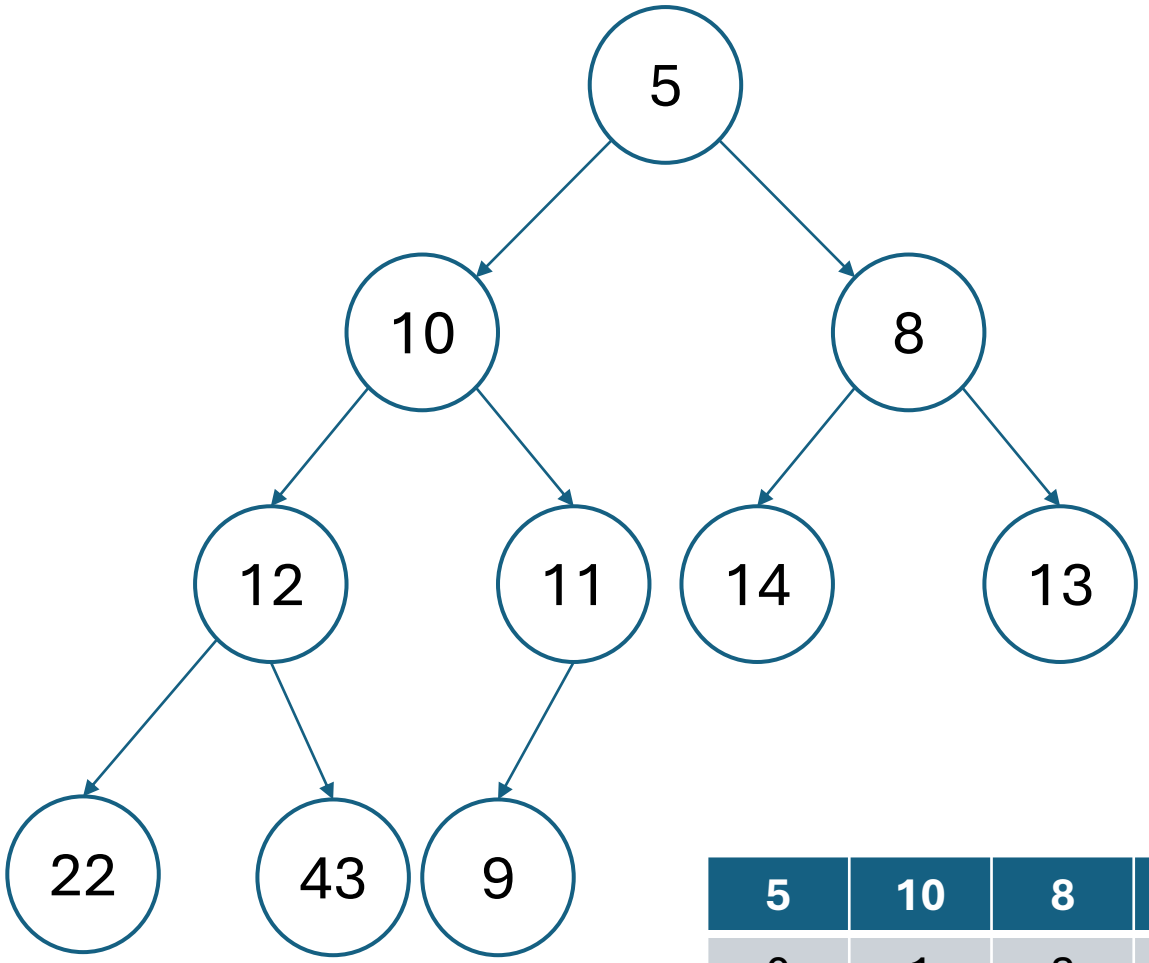
Heap operations: peek()



- peek():
  - Just return the root! return heap[0]
  - This is O(1)

5	10	8	12	11	14	13	22	43	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

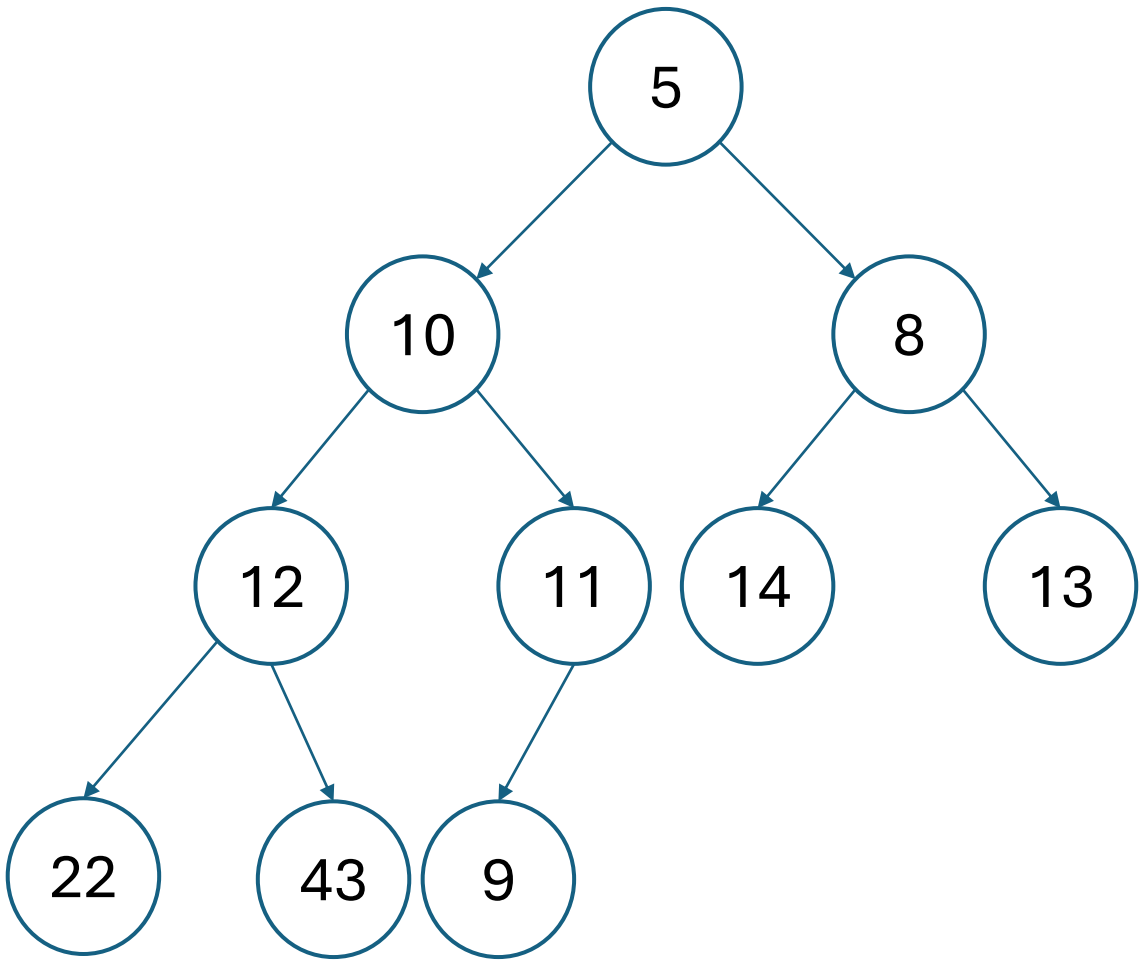
Heap operations: enqueue(e)

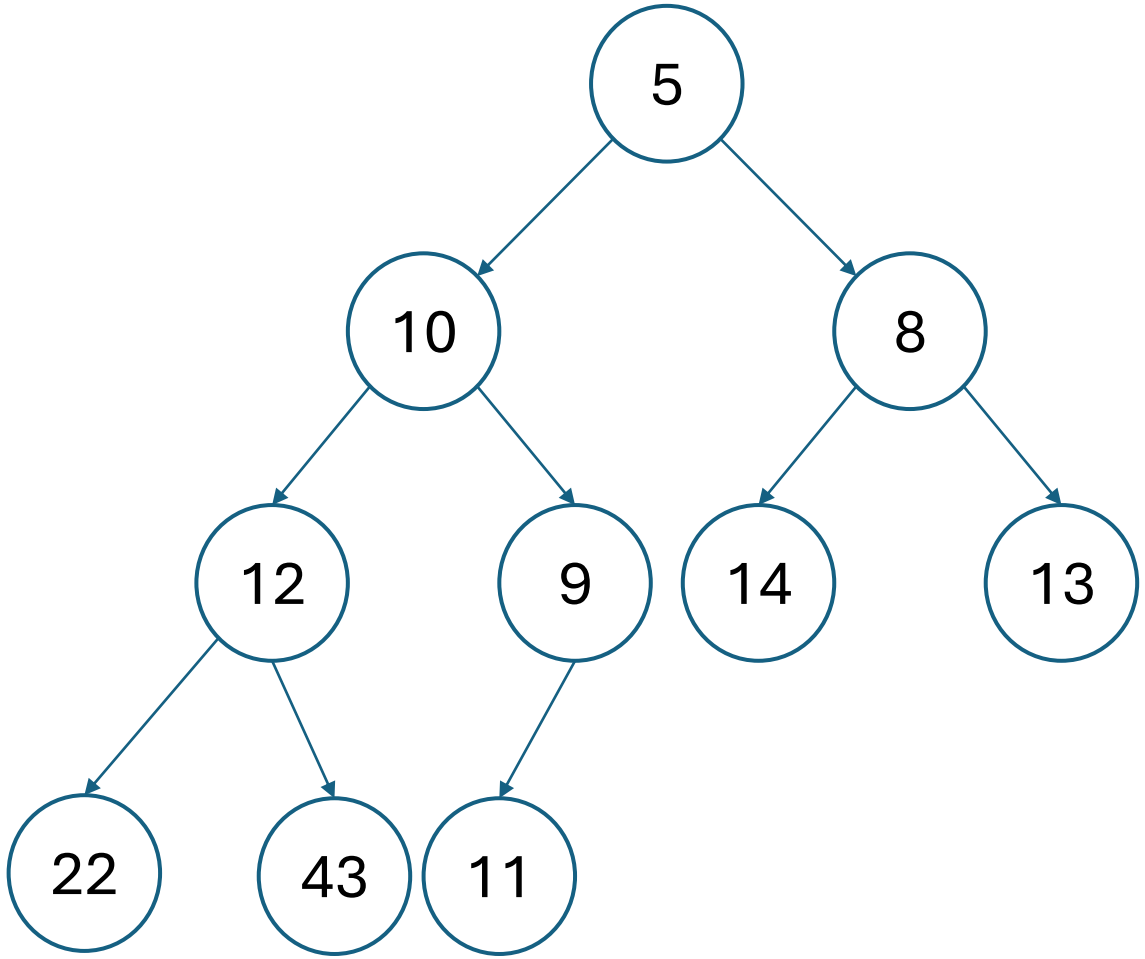


- enqueue(9) into the heap above?
- Fill each level from left to right.
- Start by putting the element as the left child of the 11.
- This destroys the heap property.
- Simply insert at heap[heapSize]:

5	10	8	12	11	14	13	22	43	9	?	?
0	1	2	3	4	5	6	7	8	9	10	11

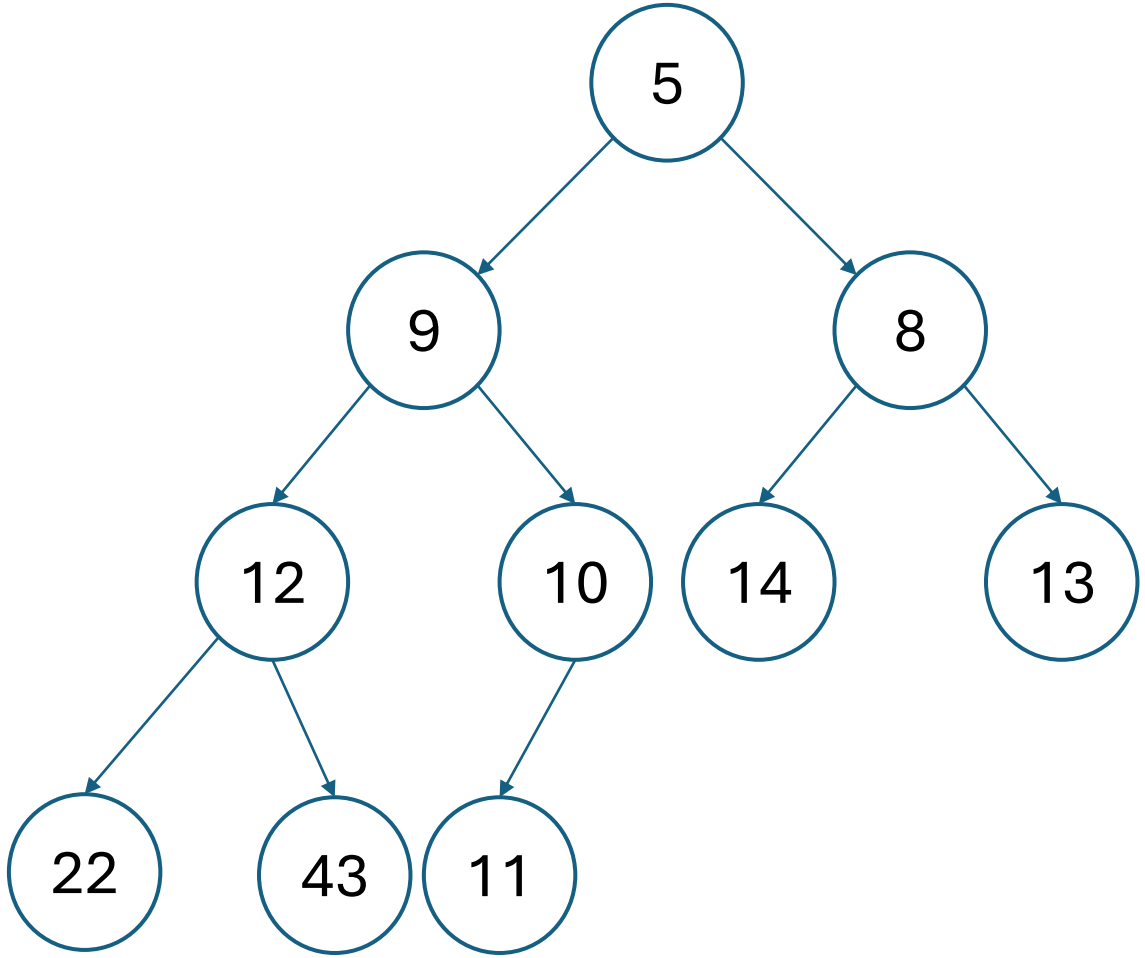
A swapping (heapifying) is required to retain the heap property





5	10	8	12	9	14	13	22	43	11	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Repeat the process until the element we added is either larger than its parent, or at the root.
- Compare 9 to its new parent, 10
  - (how do we know that from the array? 9 is at index 4, and (4 - 1) / 2 == 1, so we look at index 1 and find 10).
- Since 9 is smaller than the 10, swap them.



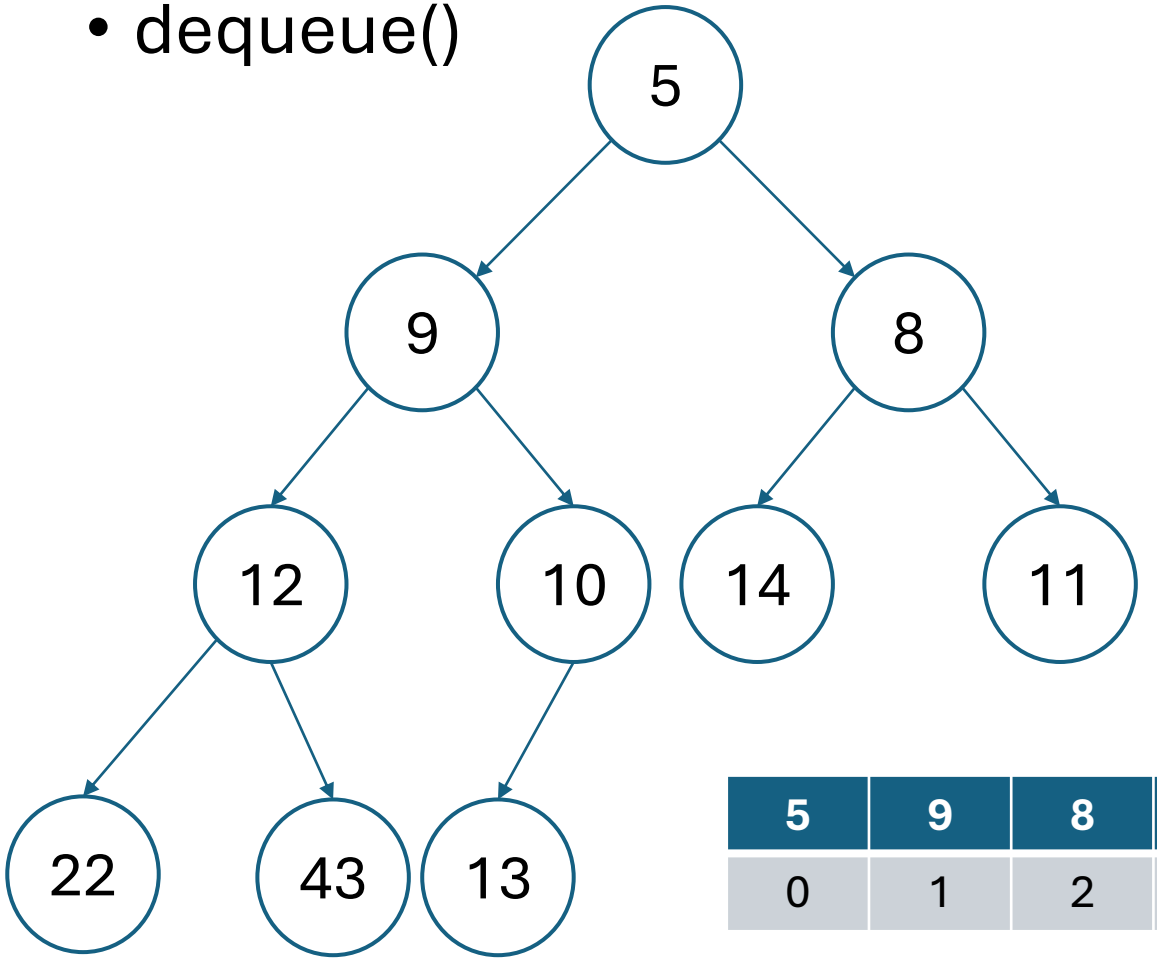
5	9	8	12	10	14	13	22	43	11	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Do one more comparison, between the 9 and its new parent, the 5.
- Since 5 is less than 9, we're done.
- What is the complexity? O(log n)
  - Insertion at the End: O(1)
  - Swapping (heapifying):
    - Swapping process continues up the tree until the heap property is restored or the element reaches the root.
    - Since a binary heap is a complete binary tree, the height of the tree is log n, where n is the number of elements in the heap.

Heap operations: dequeue()

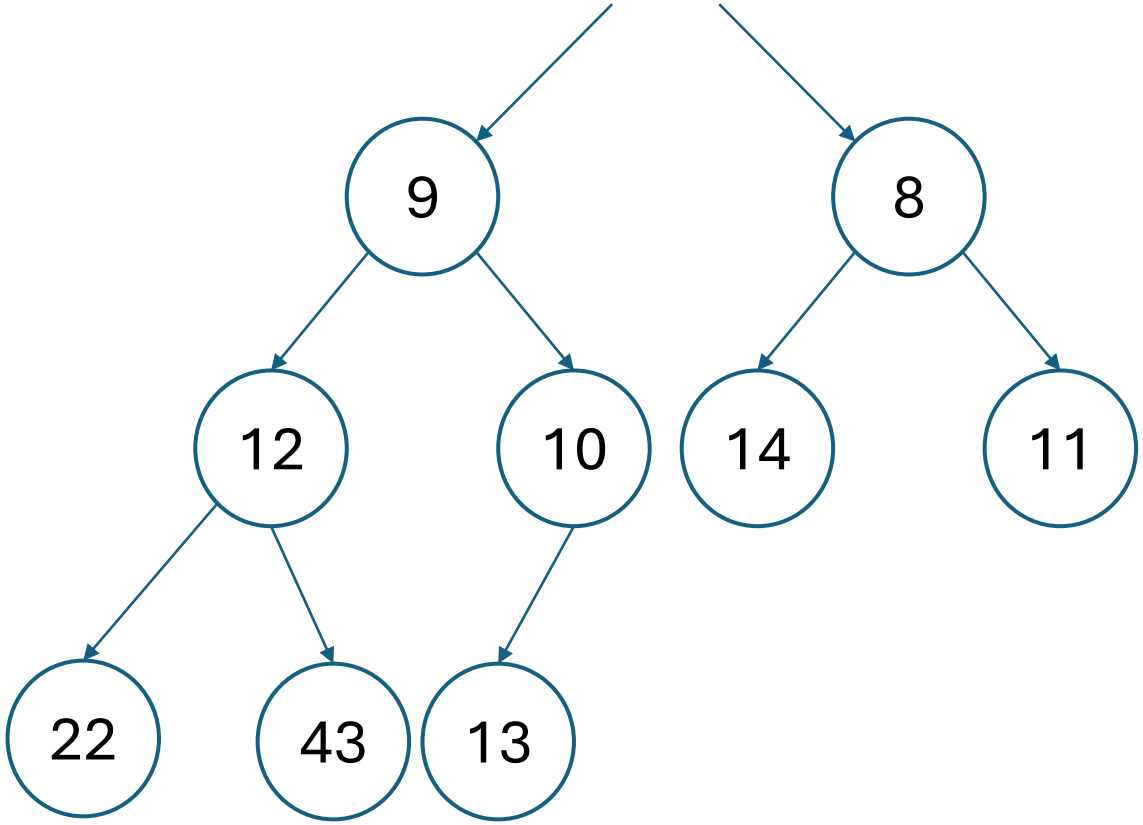
How to remove the highest priority (minimum)?

- dequeue()



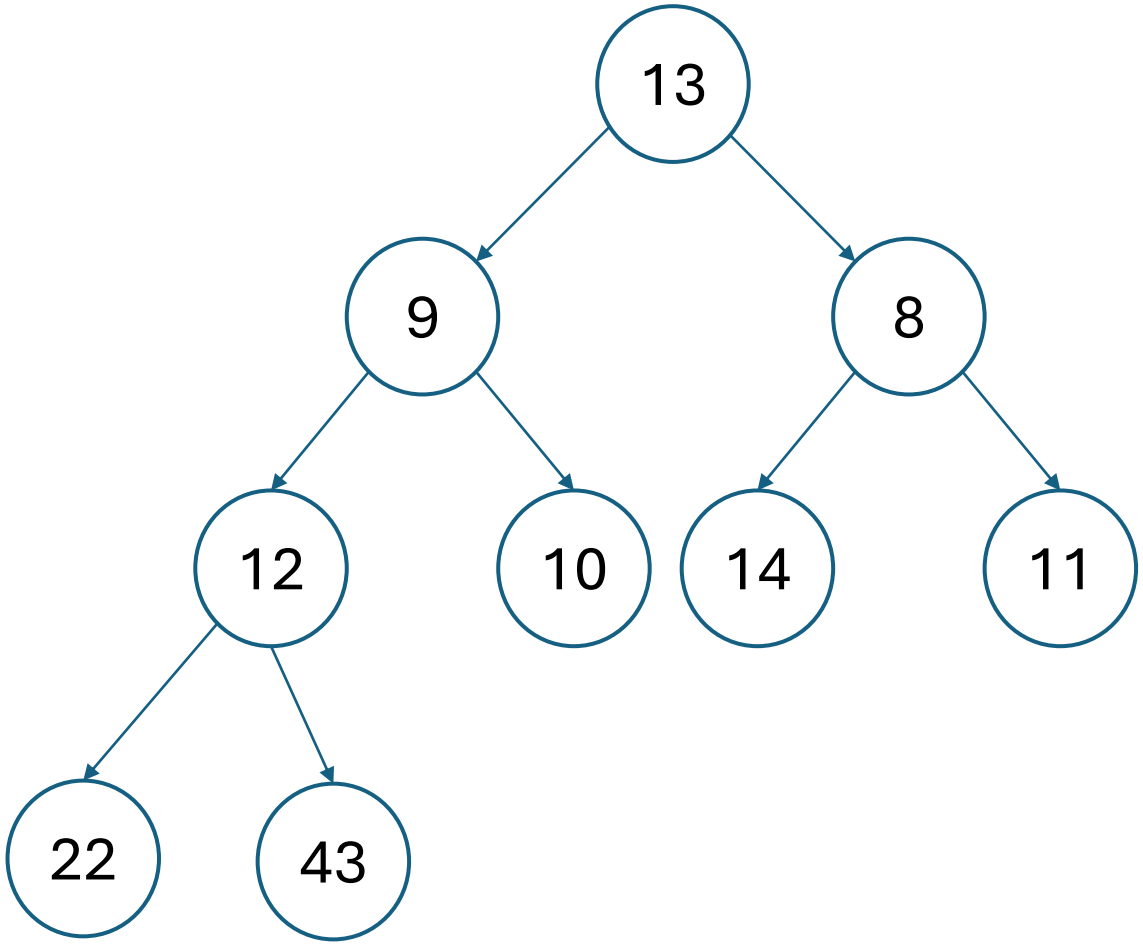
5	9	8	12	10	14	11	22	43	13	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Minimum is always at the root.
- To delete the minimum number, remove root
- This operation destroys the heap because we are removing the root



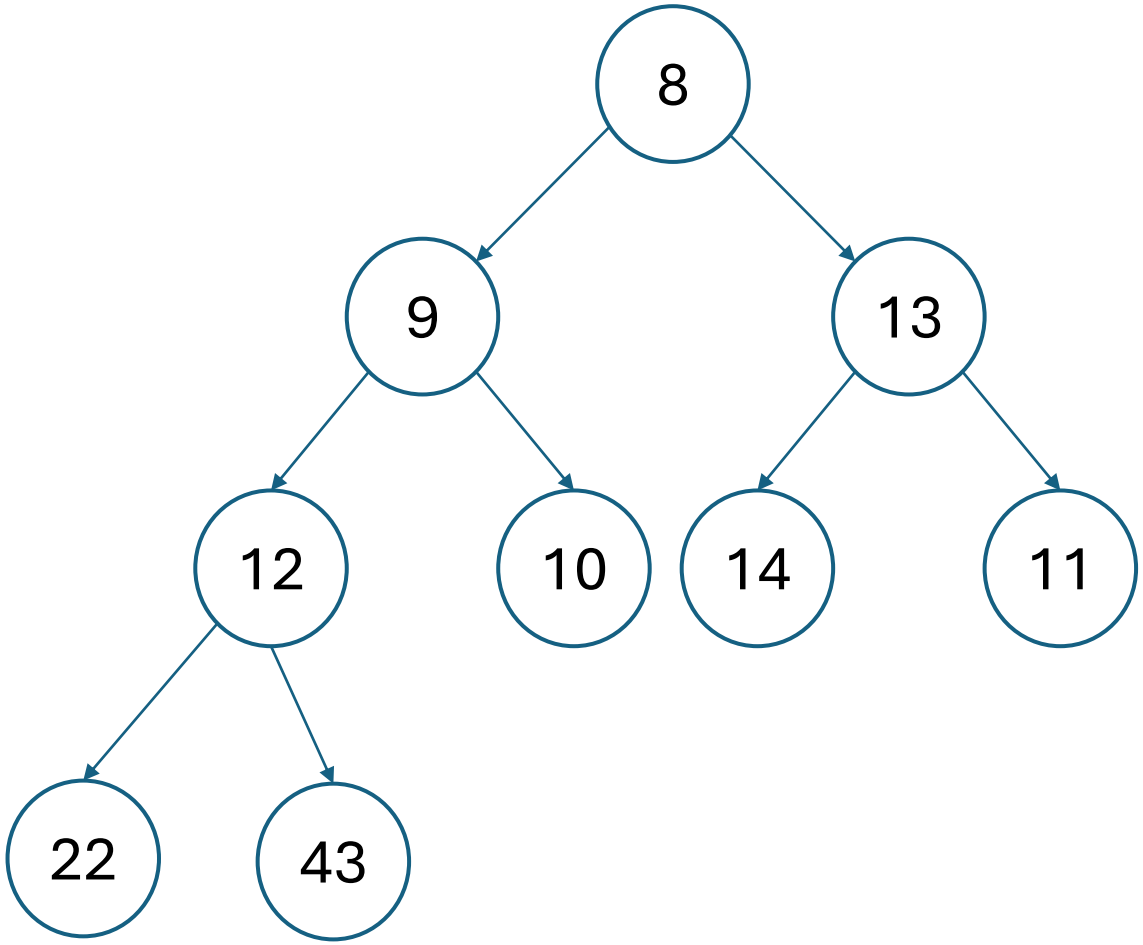
-	9	8	12	10	14	11	22	43	13	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- After we remove an element, the heap will be one element smaller.
- Where should that element come from?
  - The last level, on the right.
  - We know this will be an empty location at the end.
  - So, what we do is we take that element, and we put it at the root (the heap property will almost certainly be still destroyed):



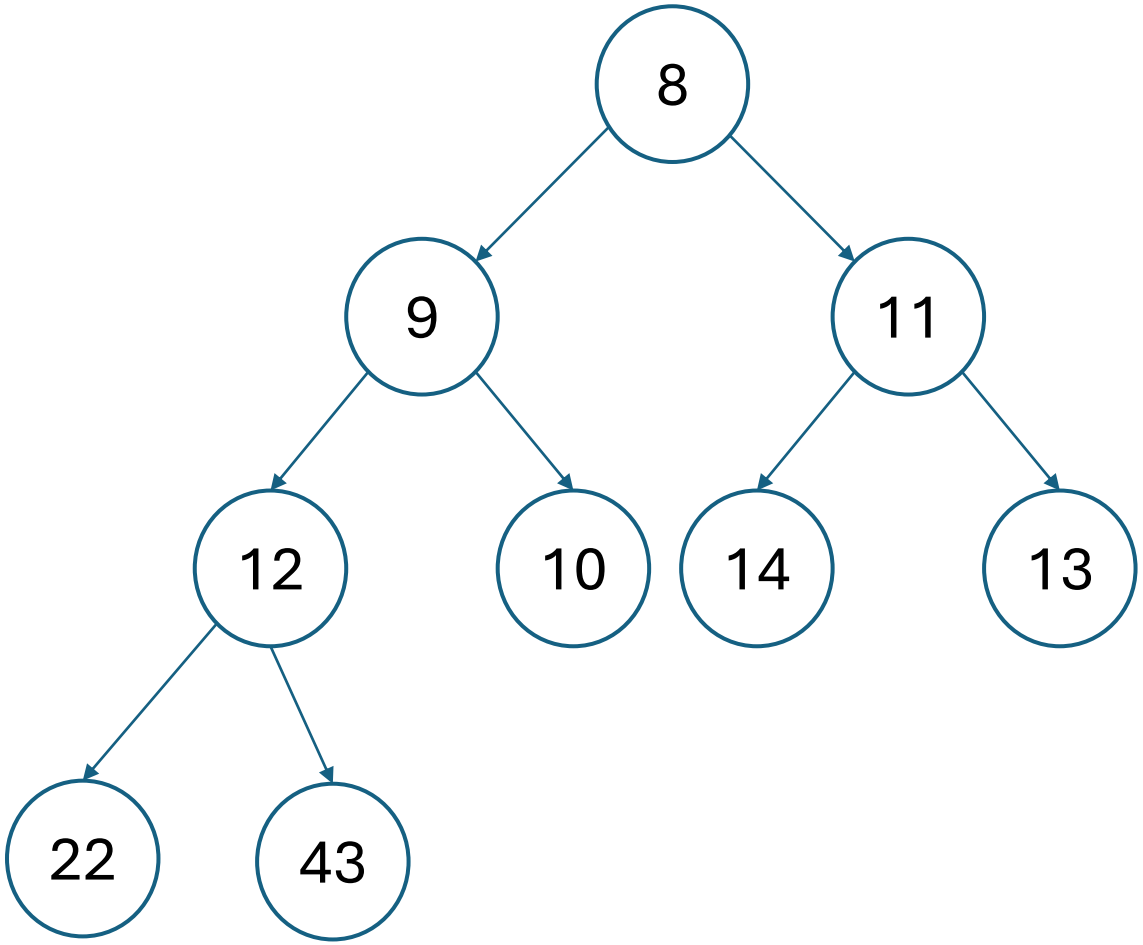
13	9	8	12	10	14	11	22	43	-	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Then do swapping. We need to take the 13, and move it down.
- Swap the element in question with its smaller child
- Therefore, swap 13 with 8, which is its smallest child



8	9	13	12	10	14	11	22	43	-	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Look at 13 and its children. Need to swap with the smallest child, 11



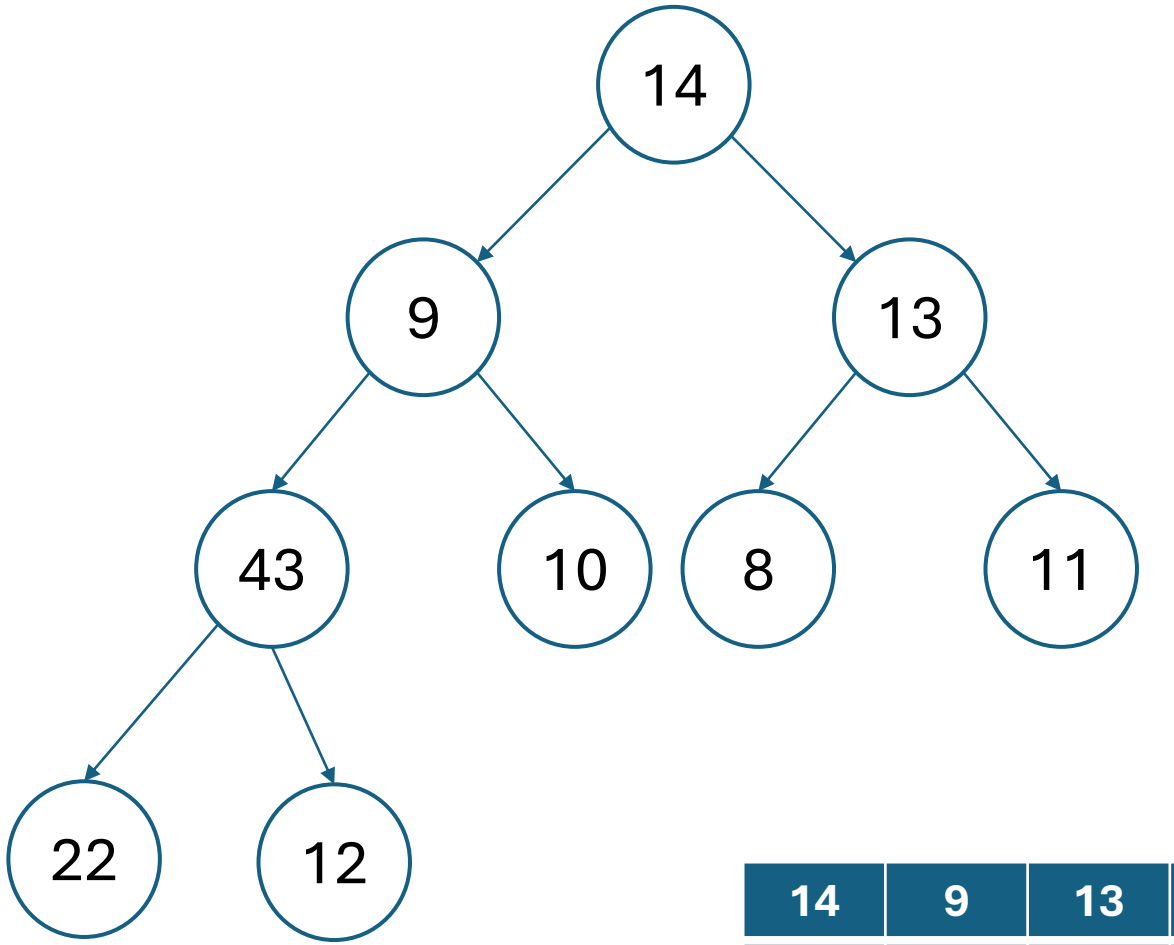
8	9	11	12	10	14	13	22	43	-	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Now, 13 has no more children, so we have completed the swapping
- What is the complexity? - O(log n)
  - Remove the Root: O(1)
  - Move the Last Element to the Root: O(1)
  - Retaining the heap property:
    - The new root might violate the heap property
    - To restore the heap property, the element is swapped with its larger child (in a max-heap) or smaller child (in a min-heap)
    - This swapping continues until the element reaches a position where it satisfies the heap property relative to its children or the element reaches a leaf node
    - A binary heap is a complete binary tree, which means it has a balanced structure with a height of O(log n)
    - In the worst case, the element moved to the root has to move down to the leaf level, involving at most the number of swaps equal to the height of the tree.

Building a heap from scratch

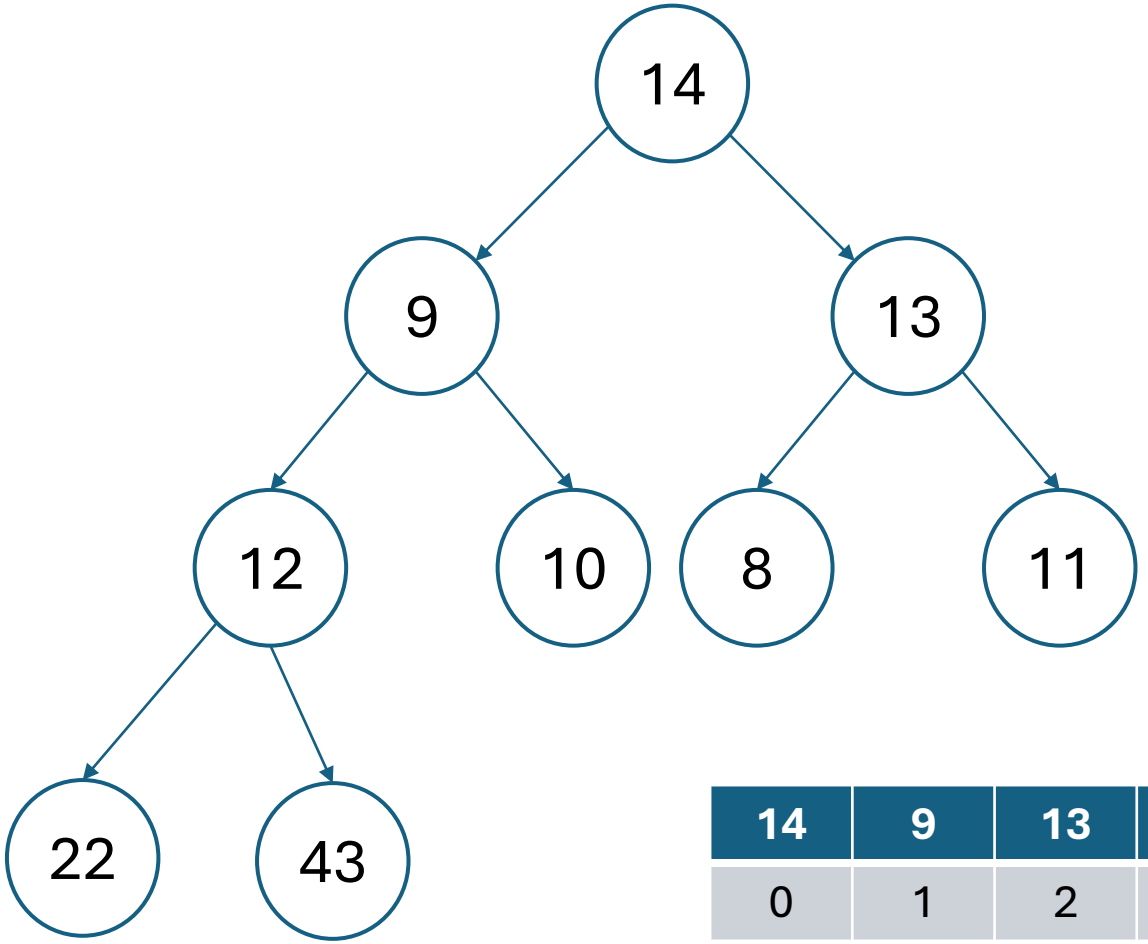
- Create a heap from: 14, 9, 13, 43, 10, 8, 11, 22, 12
- An insertion takes O(log n), and we have to insert n elements, so we have a O(n log n) (top down approach)
- There is a better way!! (bottom-up approach)
  - Insert all elements in the order they appear, without doing any other arrangement
  - Starting from the lowest completely filled level at the right-most node with children (e.g., at position n/2 - 1), rearrange each element, going backwards in the array (also O(n) to heapify the whole tree)

14, 9, 13, 43, 10, 8, 11, 22, 12



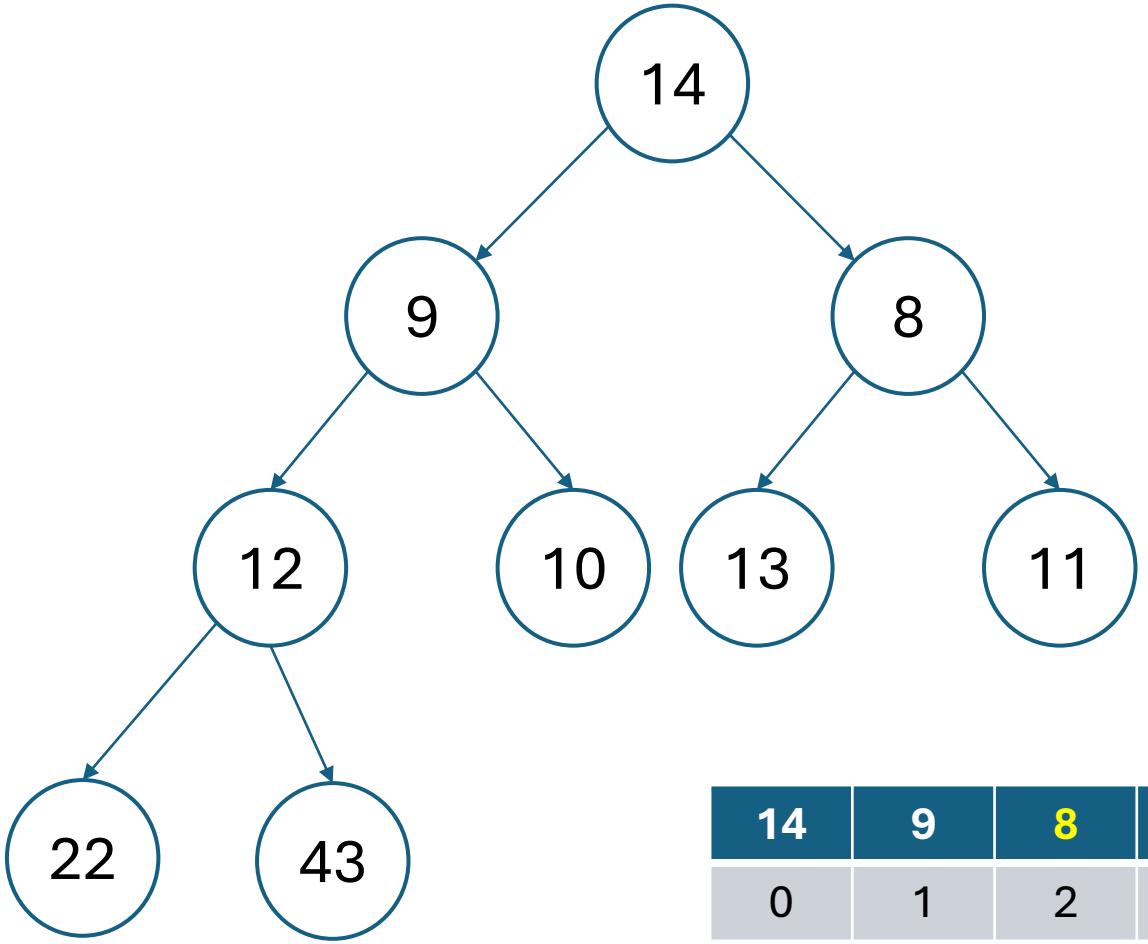
14	9	13	43	10	8	11	22	12	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Start with the 43, which is at heapSize() / 2 - 1 (index 3)
- 43 swaps with the 12



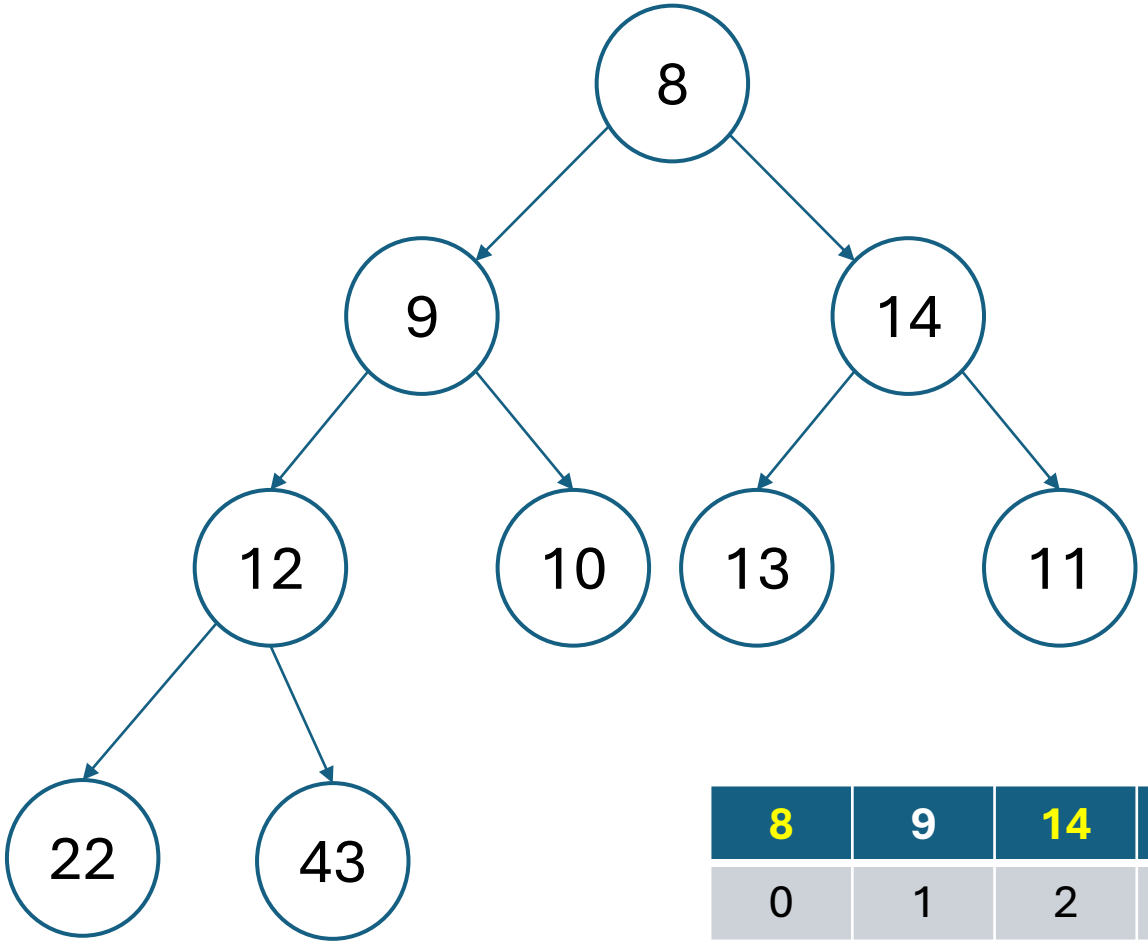
14	9	13	12	10	8	11	22	43	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Because we are looping backwards, the next index we look at is index 2, which holds the 13
- 13 swaps with the 8



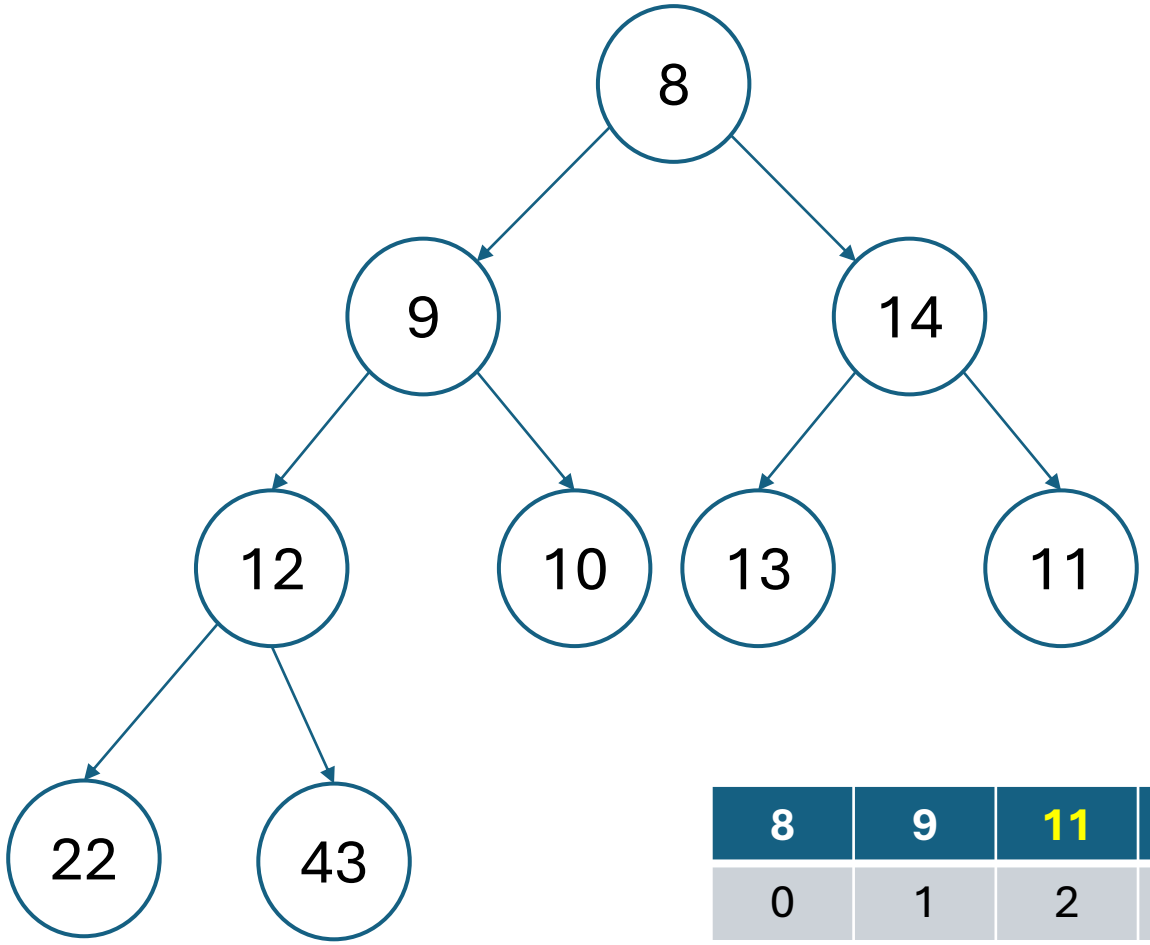
14	9	8	12	10	13	11	22	43	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- The next index we look at is 1, and the 9 does not need to be swapped, as it is already higher priority than its children.
- Finally, change the position of 14, which needs to first swap with the 8



8	9	14	12	10	13	11	22	43	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

- Rearrange the position of 14, swapping it with the 11



8	9	11	12	10	13	14	22	43	?	?	?
0	1	2	3	4	5	6	7	8	9	10	11

Heap Sort

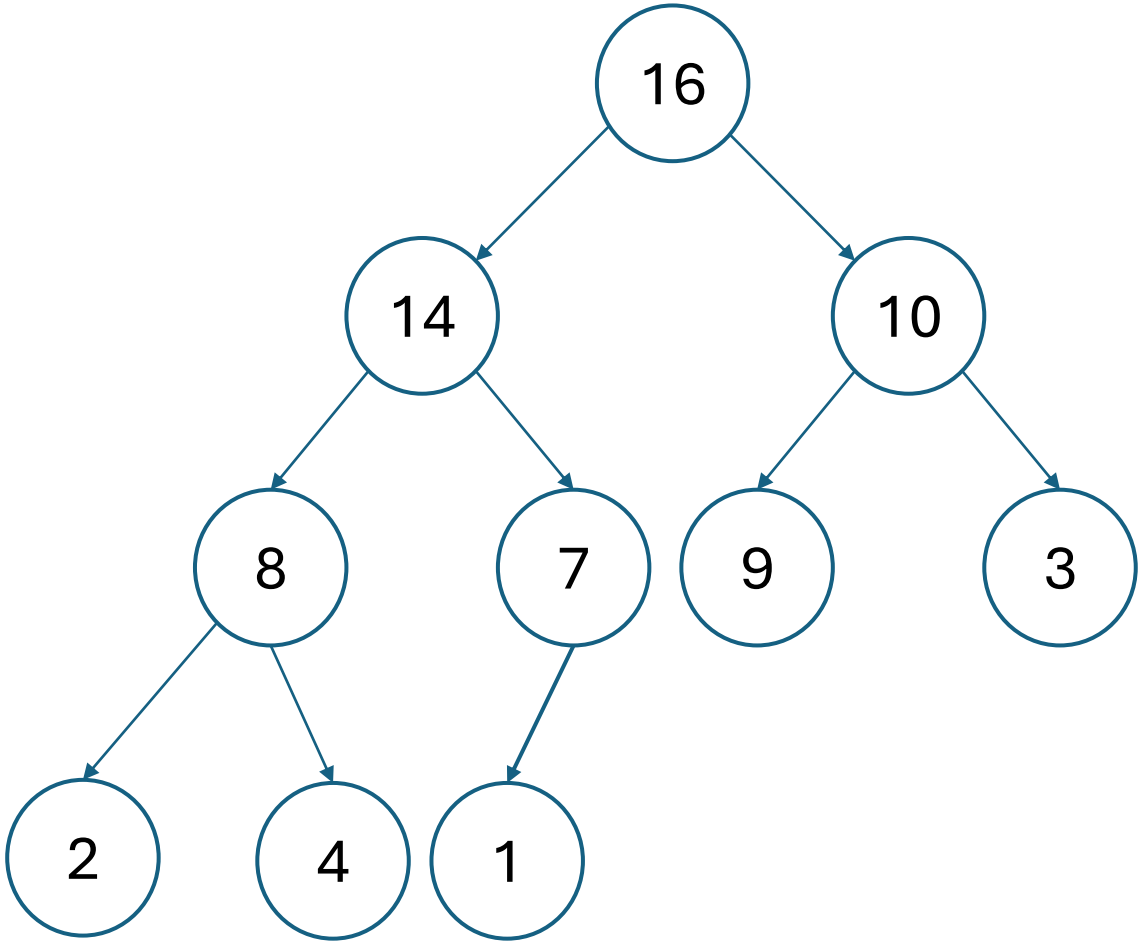
Sorting strategy

1. Build Max Heap from unordered array
2. Find maximum element heap[0]
3. Swap elements heap[n] and heap[0]
   a) now max element is at the end of the array
4. Discard node n from heap
   a) by decrementing heap-size variable
5. New root may violate max heap property, but its children are max heaps. Run MaxHeap to fix this
6. Go to Step 2 unless heap is empty

heap

0	1	2	3	4	5	6	7	8	9
16	14	10	8	7	9	3	2	4	1

heap size = 10



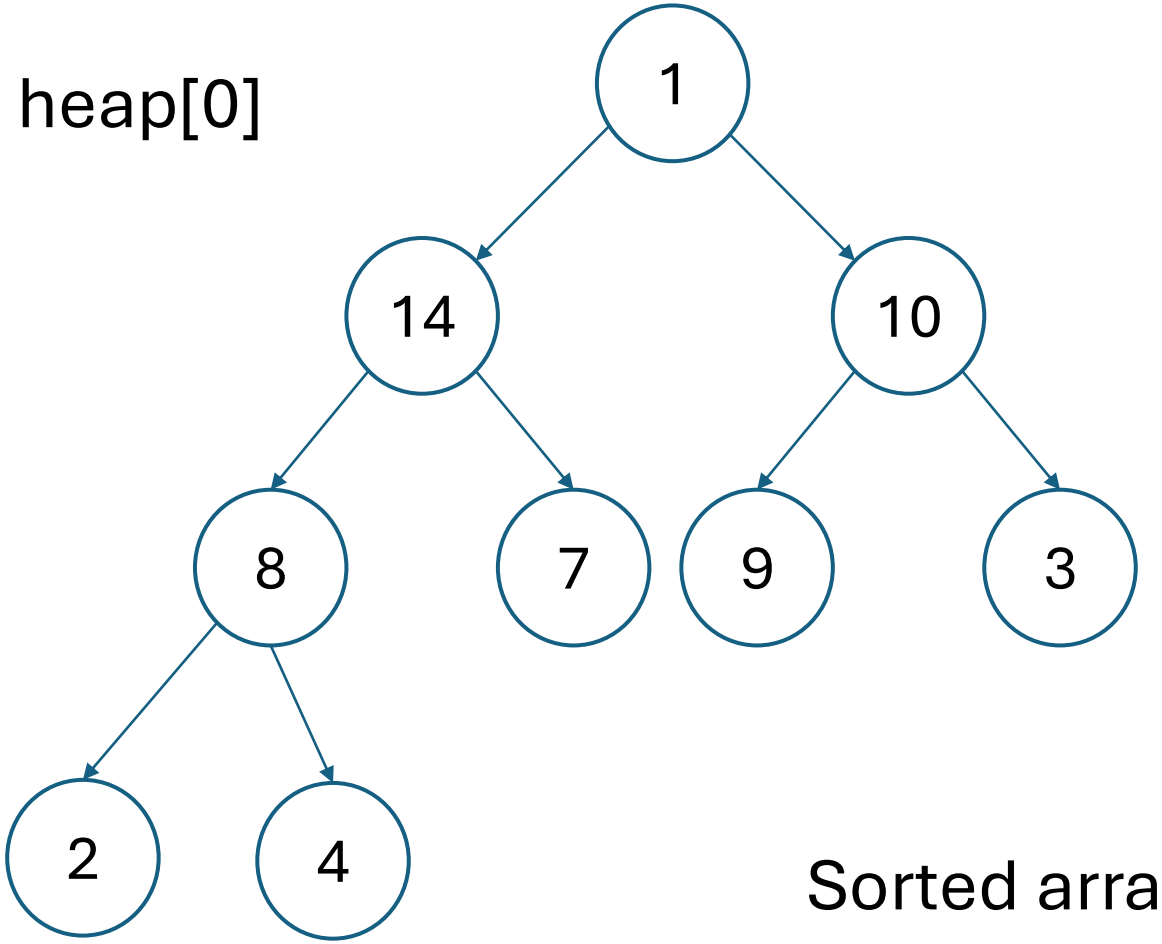
heap

0	1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4	

Swap heap[9] and heap[0]

Delete heap[9]

heap size = 9



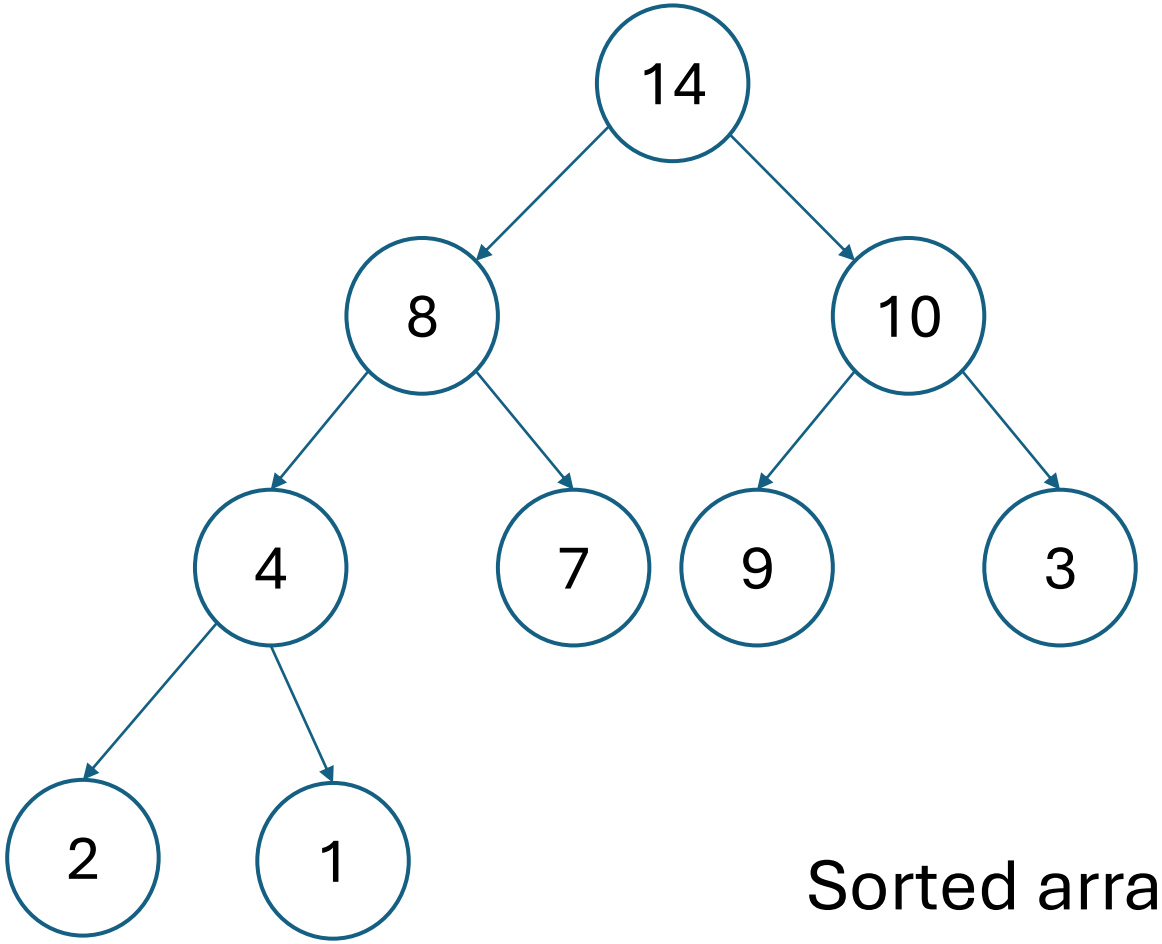
Sorted array = [16]

heap

0	1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1	

heap size = 9

heapify



Sorted array = [16]

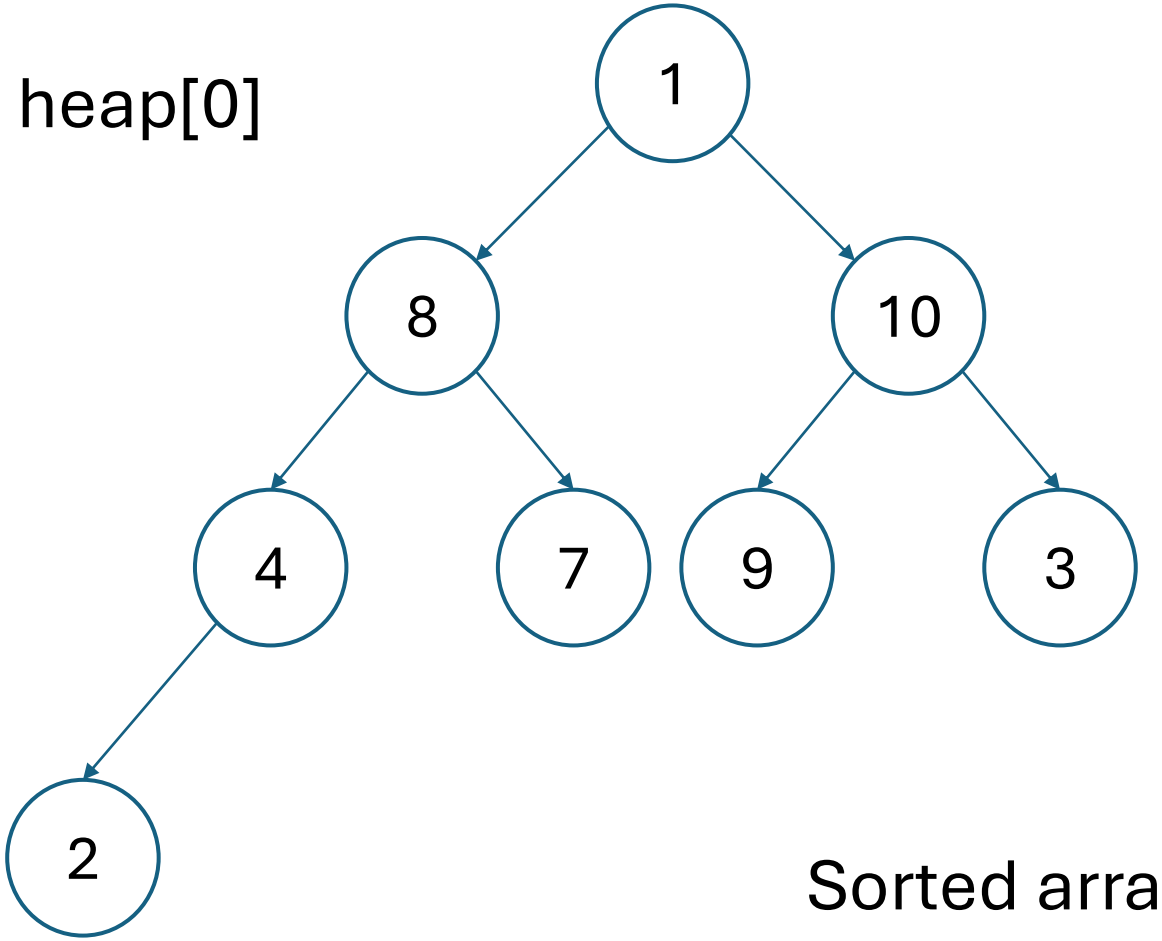
heap

0	1	2	3	4	5	6	7	8	9
1	8	10	4	7	9	3	2		

Swap heap[8] and heap[0]

Delete heap[8]

heap size = 8



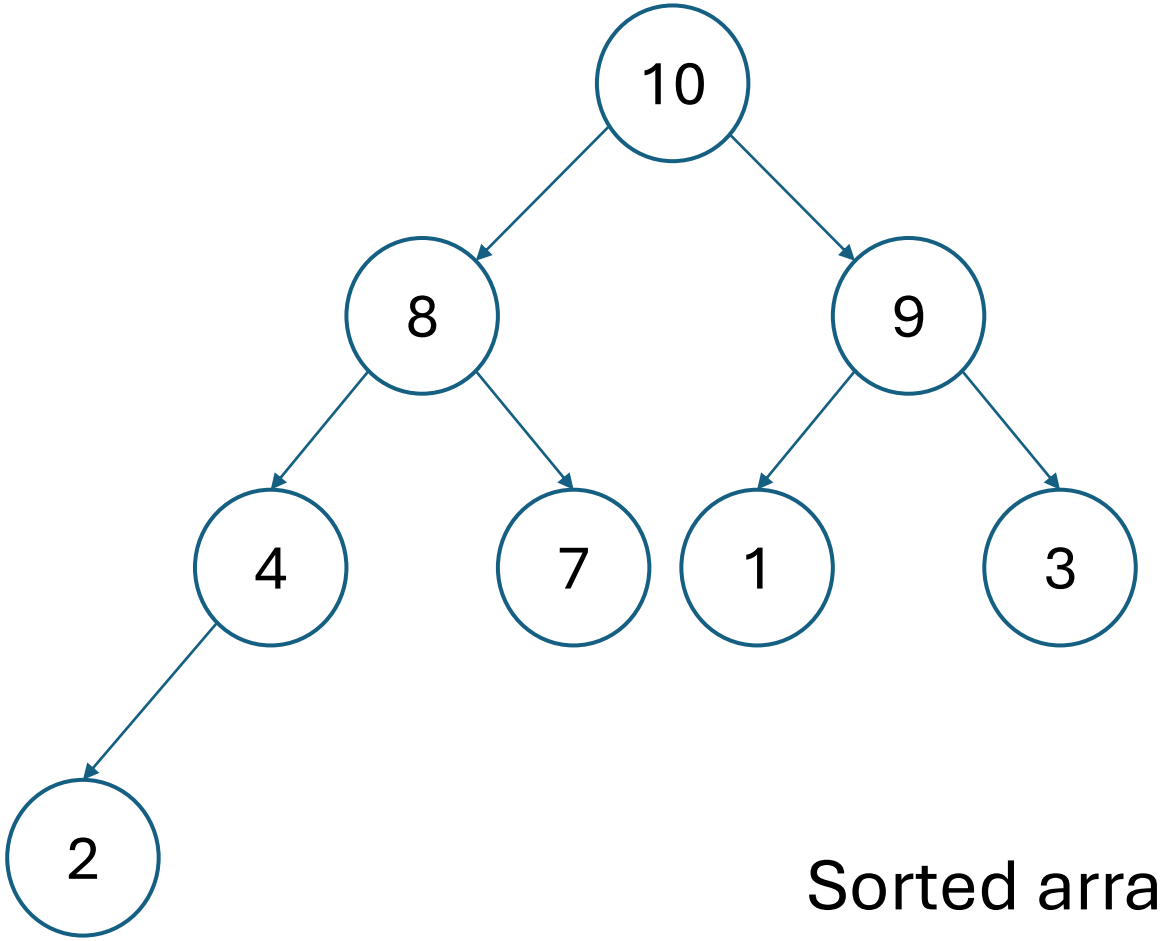
Sorted array = [14, 16]

heap

0	1	2	3	4	5	6	7	8	9
10	8	9	4	7	1	3	2		

heap size = 8

heapify



Sorted array = [14, 16]

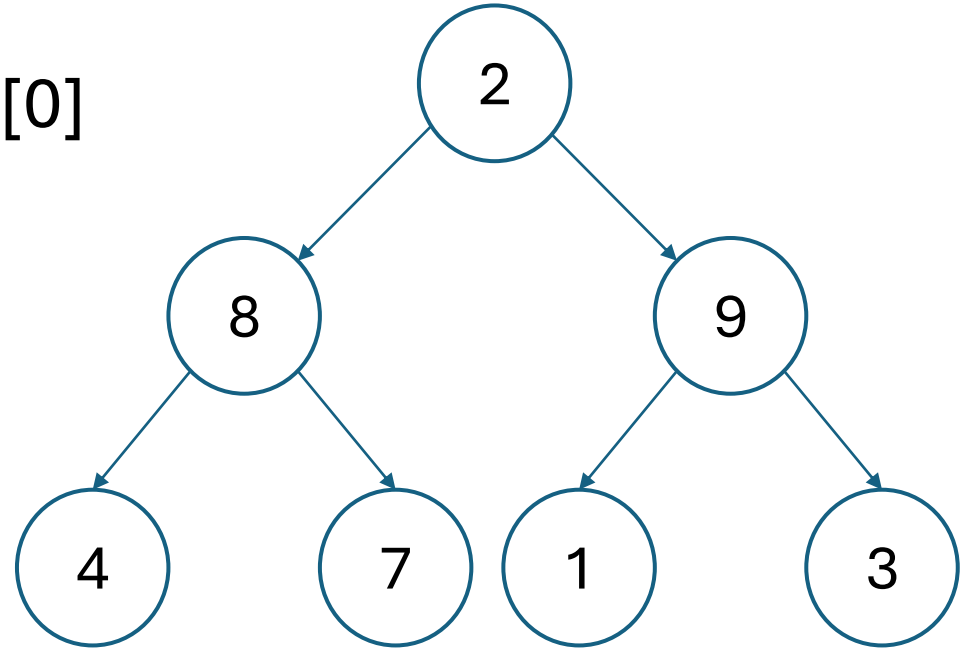
heap

0	1	2	3	4	5	6	7	8	9
2	8	9	4	7	1	3			

Swap heap[7] and heap[0]

Delete heap[7]

heap size = 7



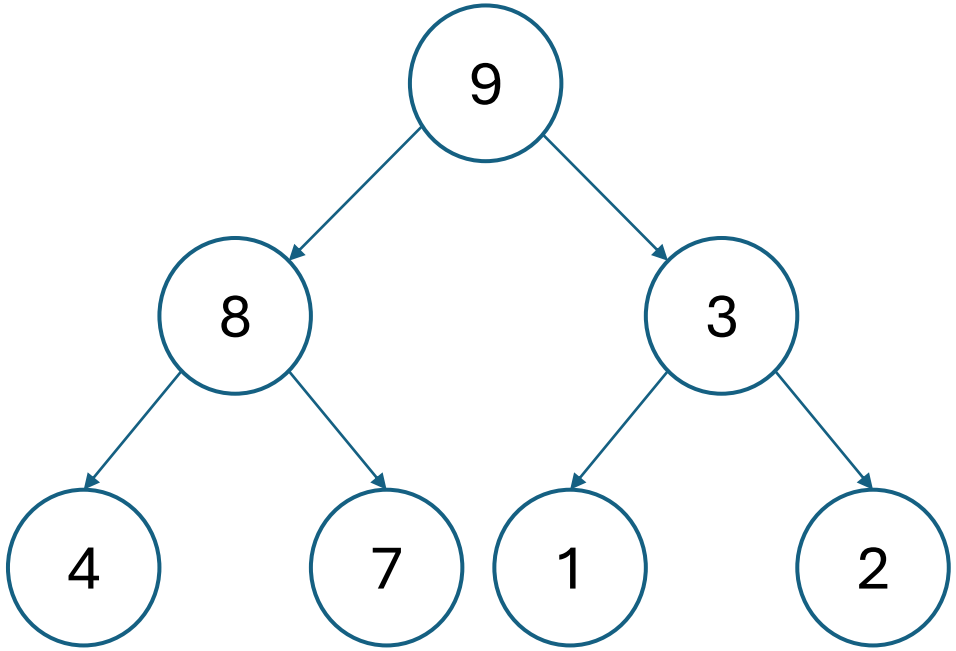
Sorted array = [10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
9	8	3	4	7	1	2			

heap size = 7

heapify



Sorted array = [10, 14, 16]

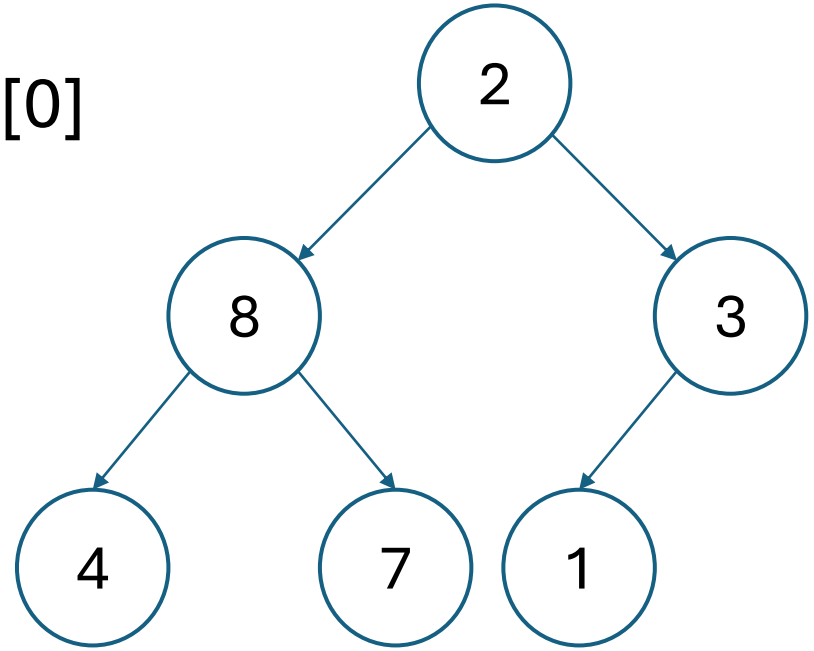
heap

0	1	2	3	4	5	6	7	8	9
2	8	3	4	7	1				

Swap heap[6] and heap[0]

Delete heap[6]

heap size = 6



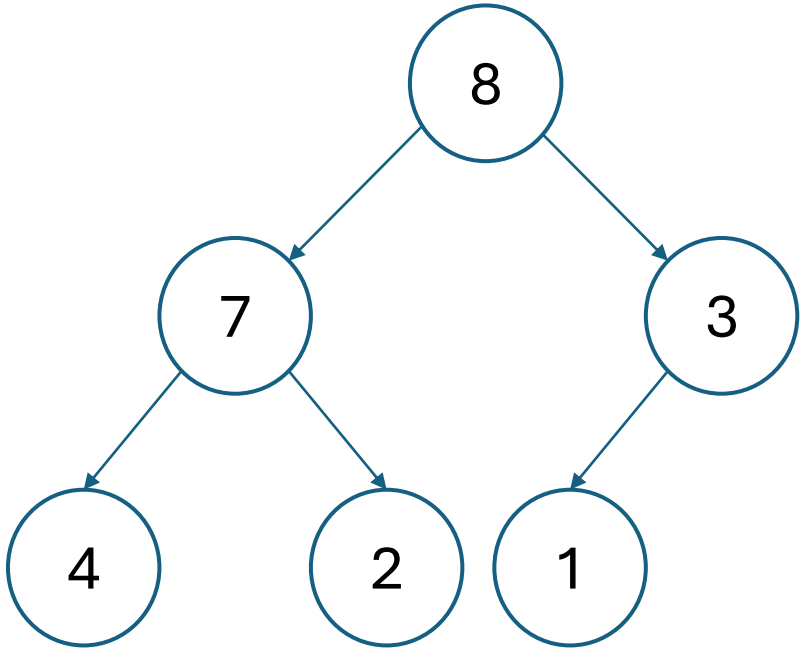
Sorted array = [9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
8	7	3	4	2	1				

heap size = 6

heapify



Sorted array = [9, 10, 14, 16]

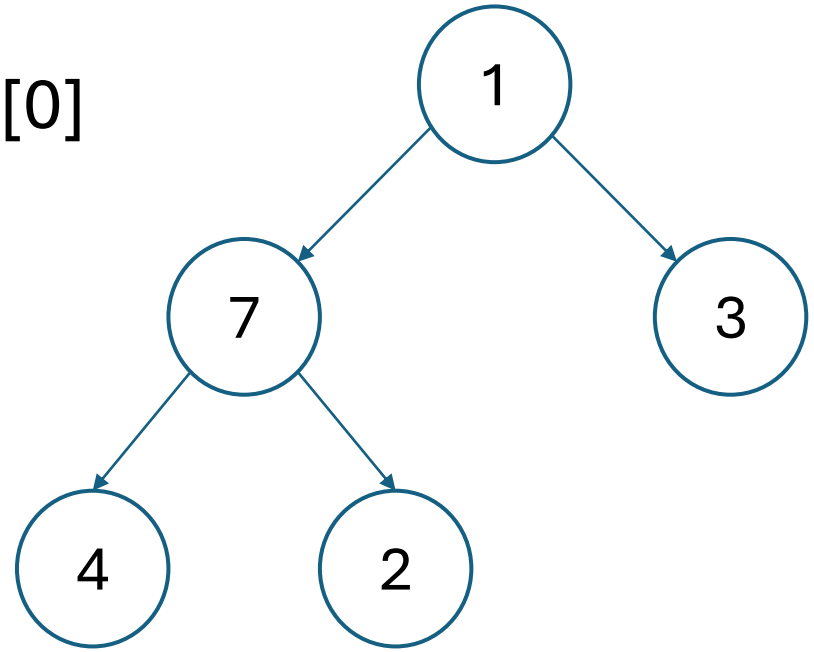
heap

0	1	2	3	4	5	6	7	8	9
1	7	3	4	2					

Swap heap[5] and heap[0]

Delete heap[5]

heap size = 5



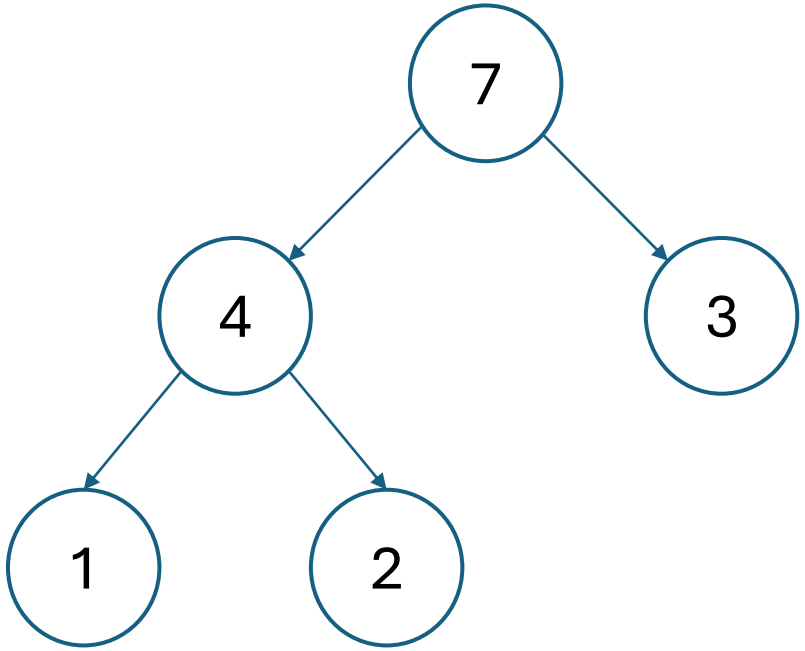
Sorted array = [8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
7	4	3	1	2					

heap size = 5

heapify



Sorted array = [8, 9, 10, 14, 16]

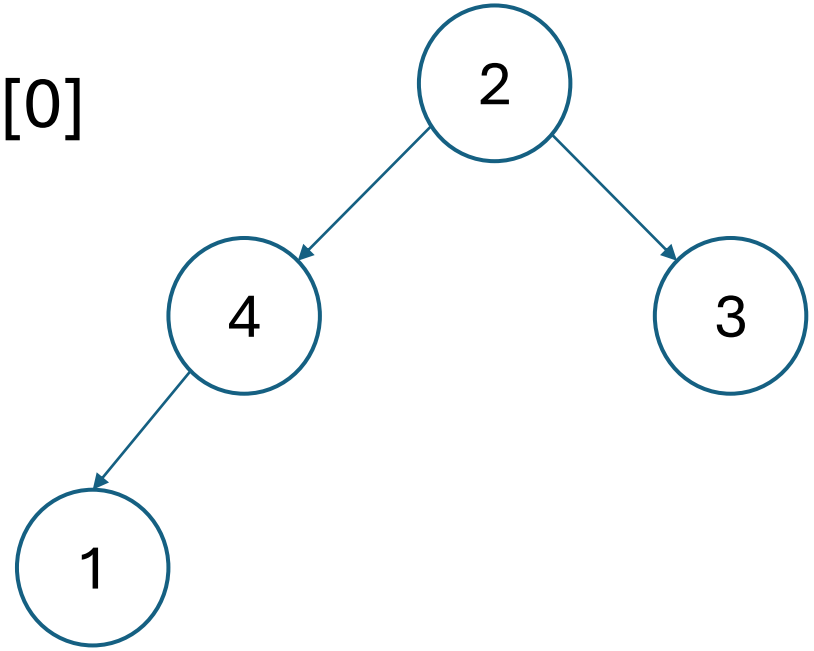
heap

0	1	2	3	4	5	6	7	8	9
2	4	3	1						

Swap heap[4] and heap[0]

Delete heap[4]

heap size = 4



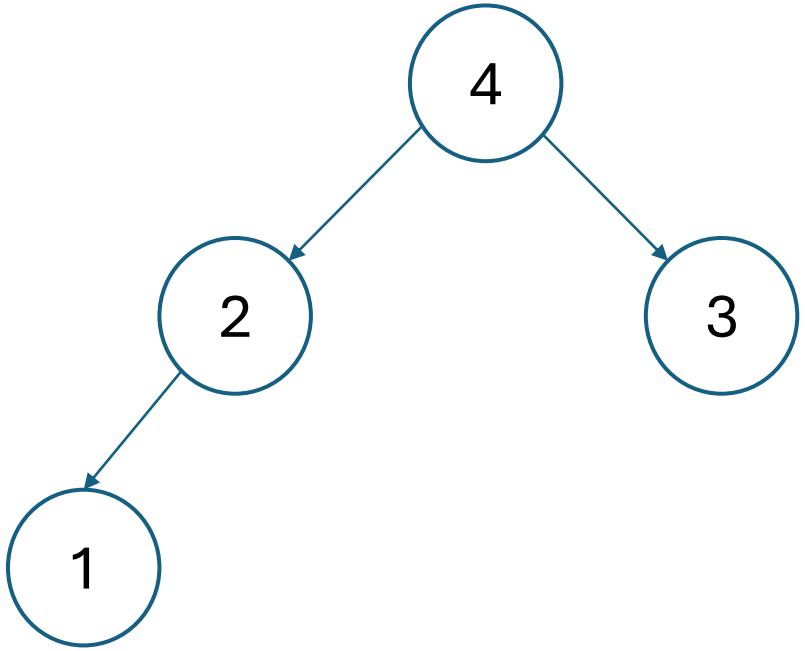
Sorted array = [7, 8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
4	2	3	1						

heap size = 4

heapify



Sorted array = [7, 8, 9, 10, 14, 16]

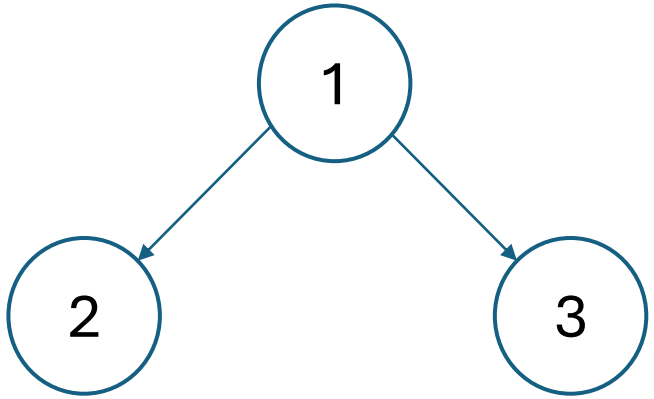
heap

0	1	2	3	4	5	6	7	8	9
1	2	3							

Swap heap[3] and heap[0]

Delete heap[3]

heap size = 3



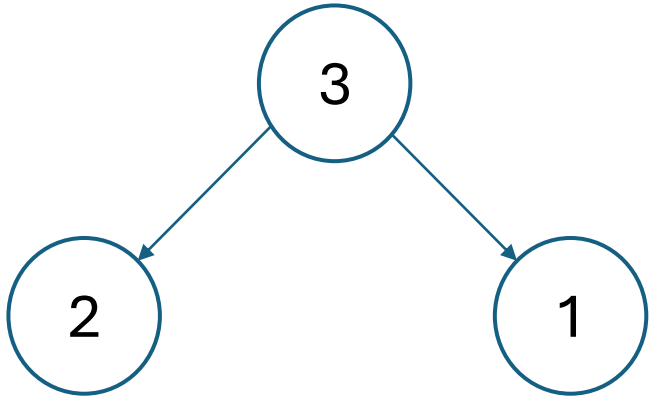
Sorted array = [4, 7, 8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
3	2	1							

heap size = 3

heapify



Sorted array = [4, 7, 8, 9, 10, 14, 16]

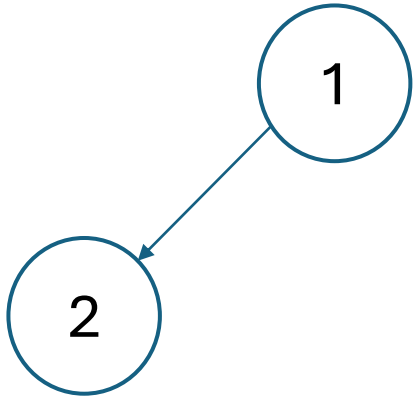
heap

0	1	2	3	4	5	6	7	8	9
1	2								

Swap heap[2] and heap[0]

Delete heap[2]

heap size = 2



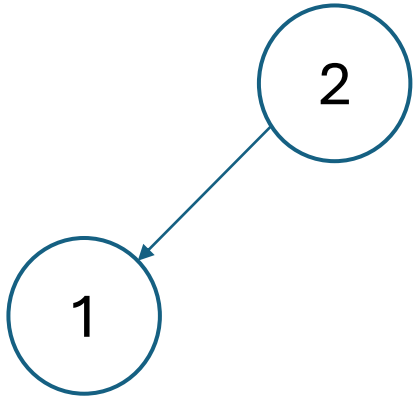
Sorted array = [3, 4, 7, 8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
2	1								

heap size = 2

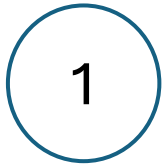
heapify



Sorted array = [3, 4, 7, 8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
1									



Swap heap[1] and heap[0]

Delete heap[1]

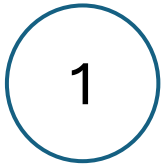
heap size = 1

Sorted array = [2, 3, 4, 7, 8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9
1									

heap size = 1



heapify

Sorted array = [2, 3, 4, 7, 8, 9, 10, 14, 16]

heap

0	1	2	3	4	5	6	7	8	9

Swap heap[0] and heap[0]

Delete heap[0]

heap size = 0

Sorted array = [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]