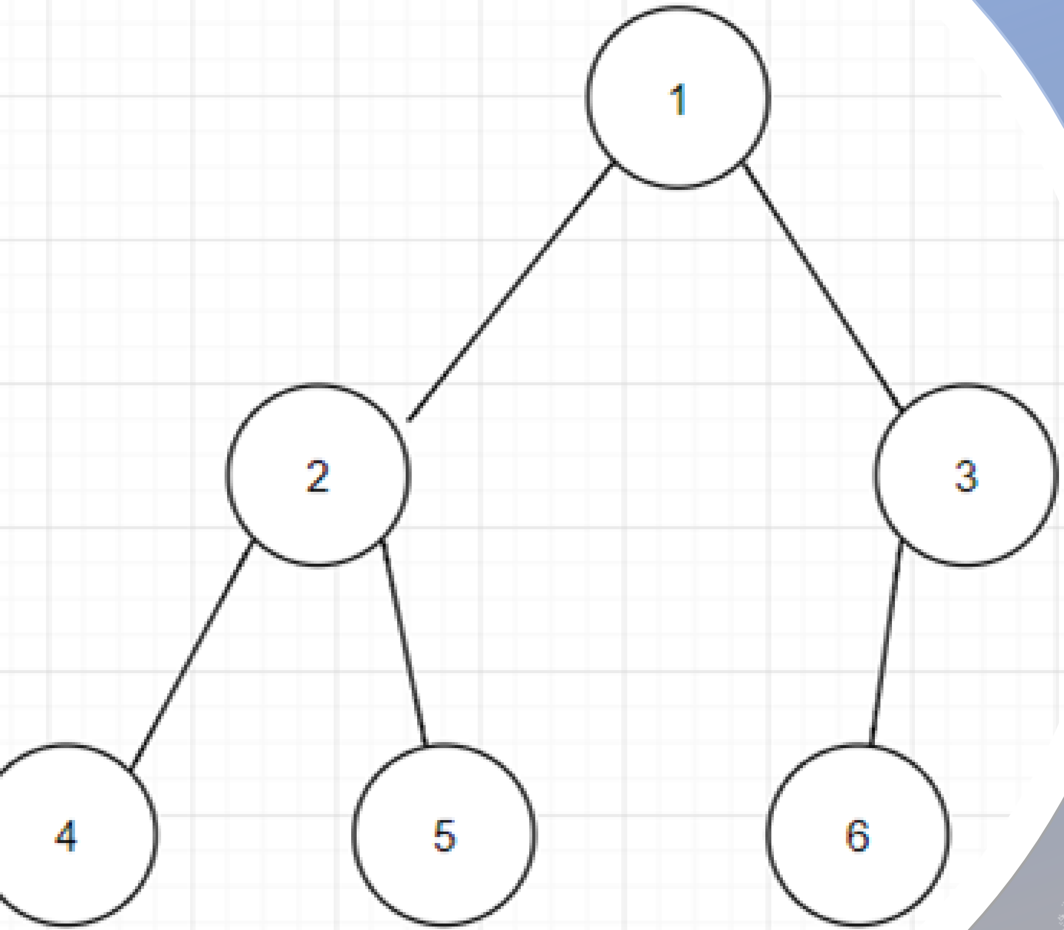


- Complete Binary Tree
- Max-Heap

$n = 6$

$h = 2$

$2^h = 4$

$2^{h+1} - 1 = 7$

Heap

- What is a complete binary tree?
- What is a heap?
  - What is a max heap?
  - What is a min heap?
- Two ways to build a heap
  - Top down (Time complexity O(nlog n)) is iterative and in place.
  - Bottom up (Time complexity O(n)) is iterative and in place.
- Heap Sort
  - Phase I : Build the heap bottom up (Reason: it is O(n) and we have all the data)
  - Phase II : Sort (Time complexity O(nlog n)) is iterative and in place.
- Priority Queue
  - Insert (Time complexity : O(log n))
  - Delete ((Time complexity : O(log n). You must restore the heap)
  - Build the heap top down (Reason : we do not have all the data)

Note : An **in-place algorithm** is an algorithm which transforms input using no auxiliary **data structure**. (Can use O(1) temporary variables).

Build Max-Heap in-place Iteratively

- There two ways you can build the heap
- Both are iterative and in-place

An **in-place algorithm** is an algorithm which transforms input using no auxiliary **data structure**. (Can use O(1) temporary variables).

Top-down: O(nlog n).

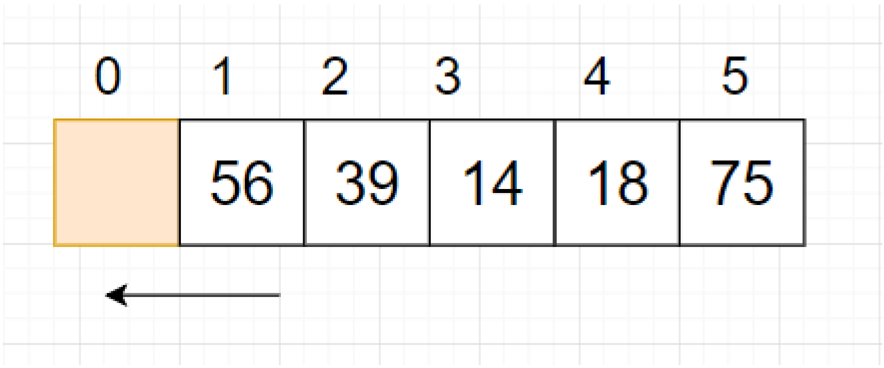
Bottom-up: O(n)

Build Max-Heap in-place Iteratively Top-down

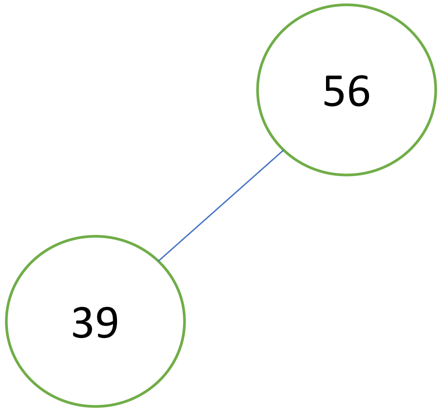
```
build_MaxHeap_TopDown(A, n)
    for (i <- 1 to n)
        upHeap(A, i)
```

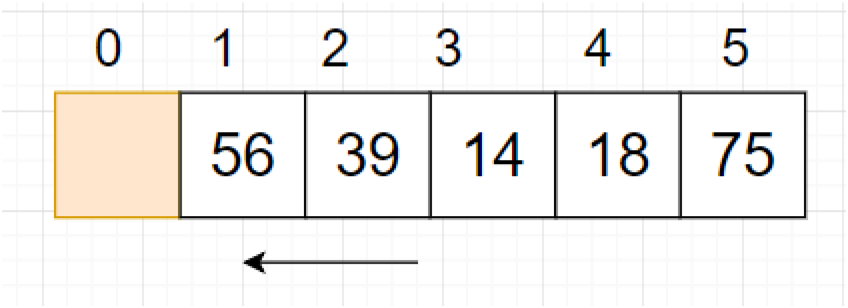
```
upHeap(A, i)
j <- i
while (j > 1 & A[j/2] < A[j])
    swap(A[j], A[j/2])
    j <- j/2
```

Build Max-Heap in-place Iteratively Top-down

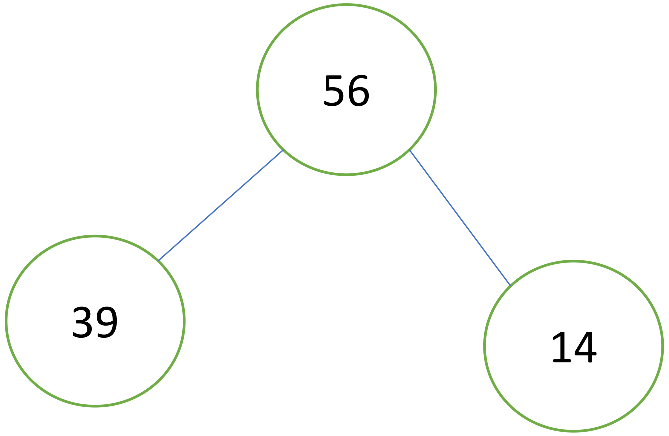


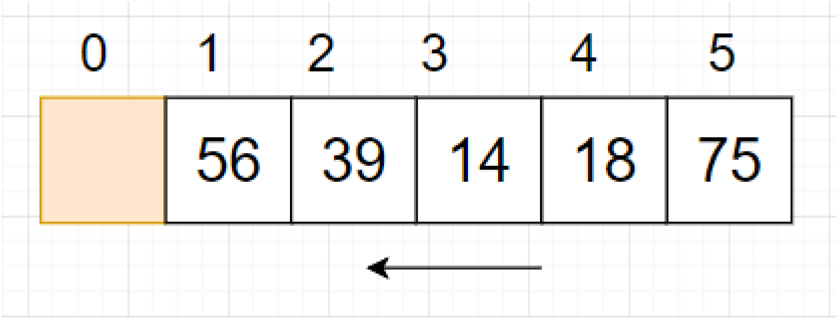
Build Max-Heap in-place Iteratively Top-down



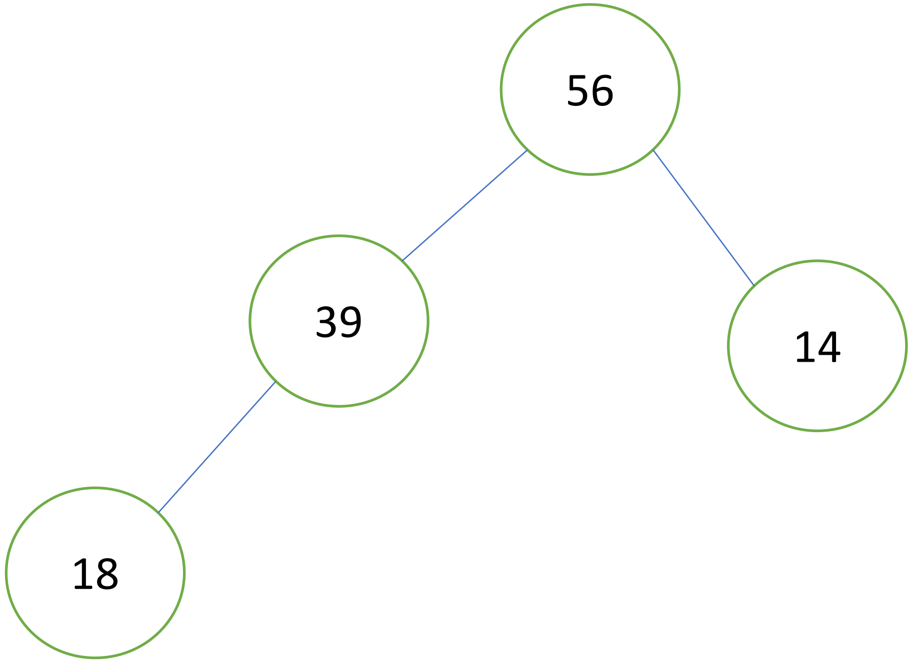


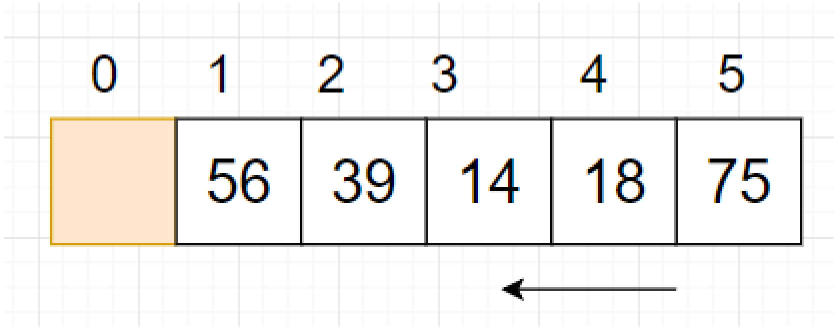
Build Max-Heap in-place Iteratively Top-down



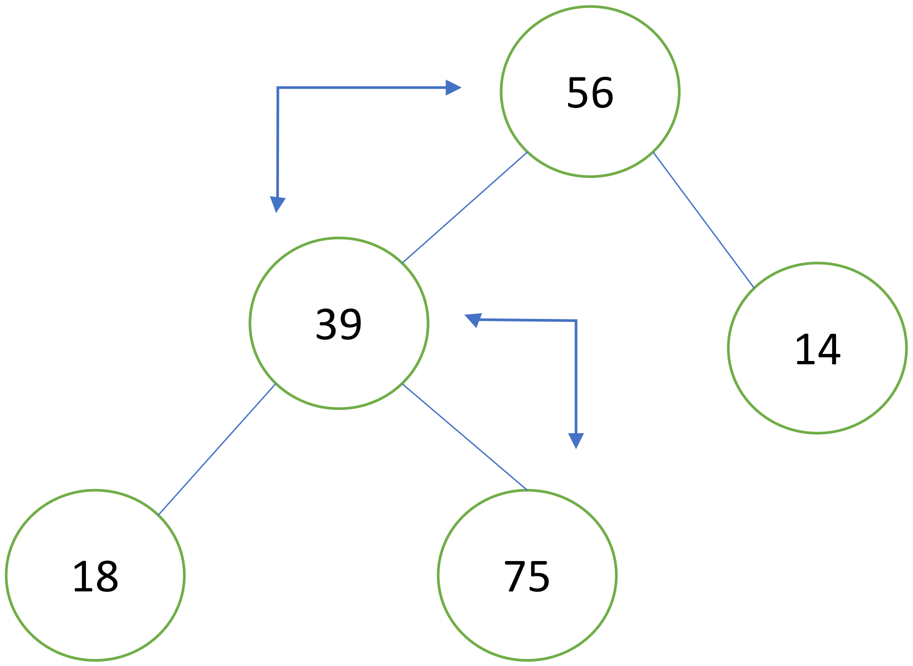


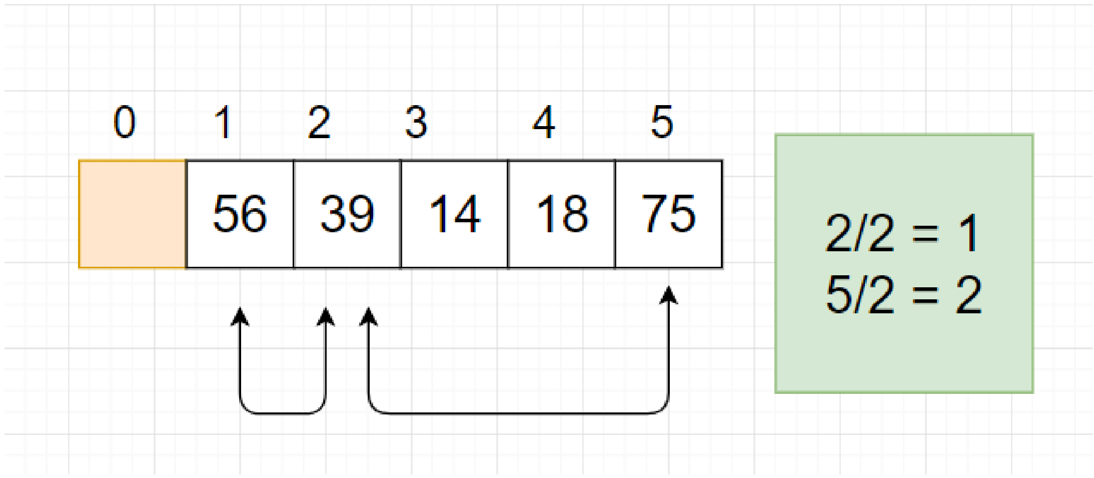
Build Max-Heap in-place Iteratively Top-down



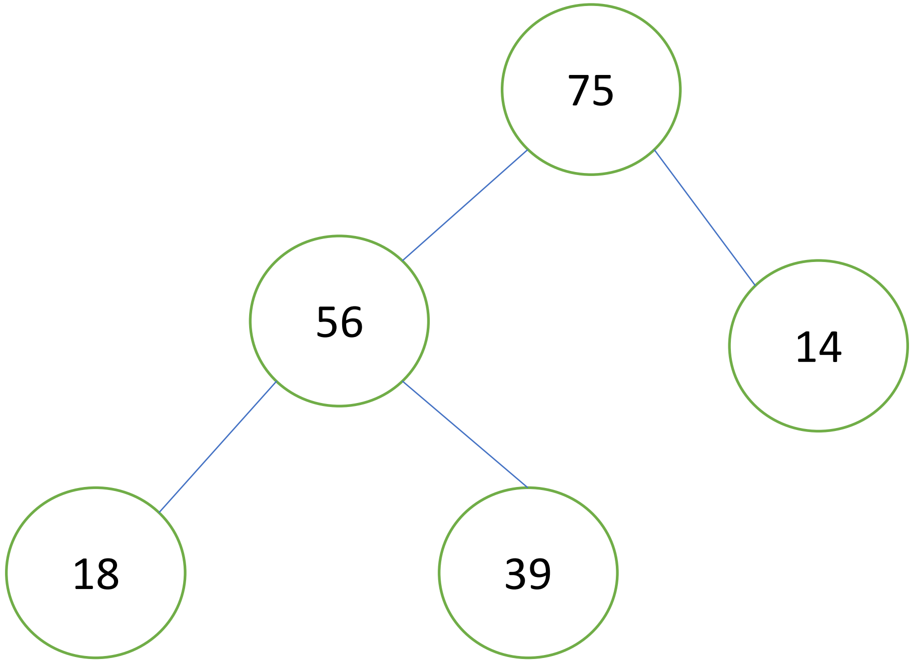


Build Max-Heap in-place Iteratively Top-down





Build Max-Heap in-place Iteratively Top-down



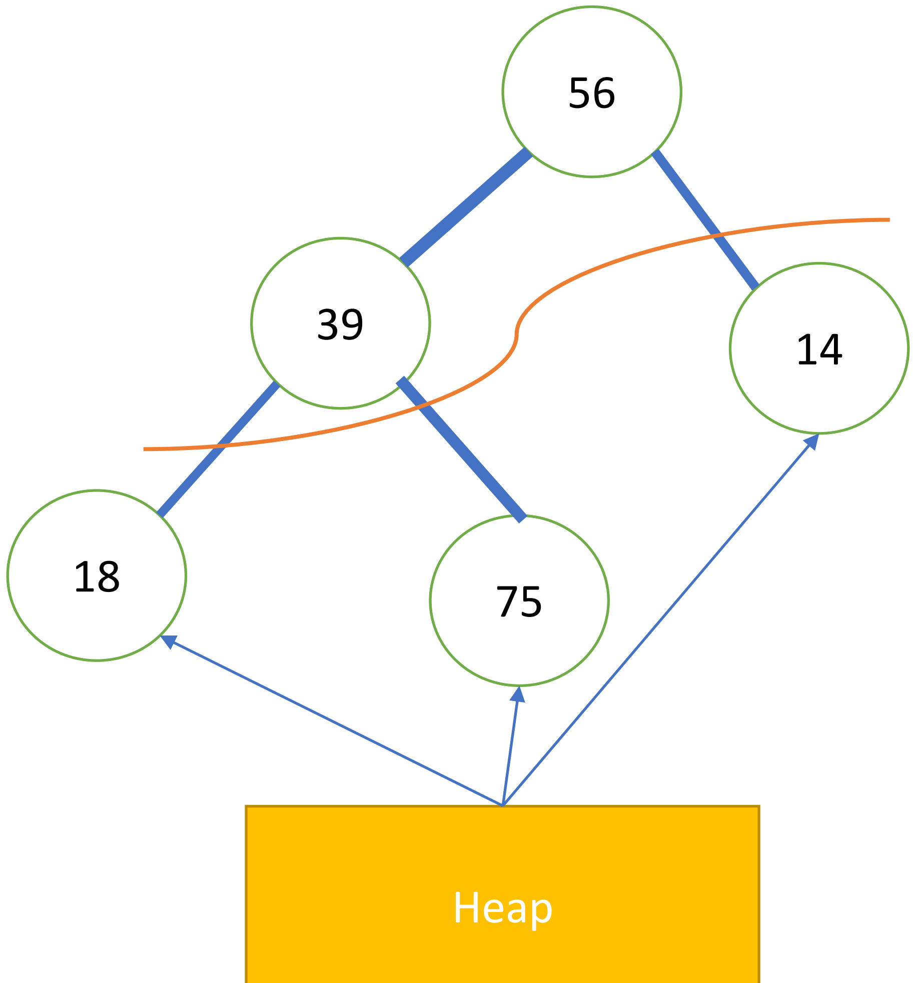
Heap Sort

There are two phases.

Phase I : Build the heap. In-place, bottom-up, iteratively

Phase II : Sort. In-place, iteratively

Build Max-Heap in-place Iteratively Bottom-up



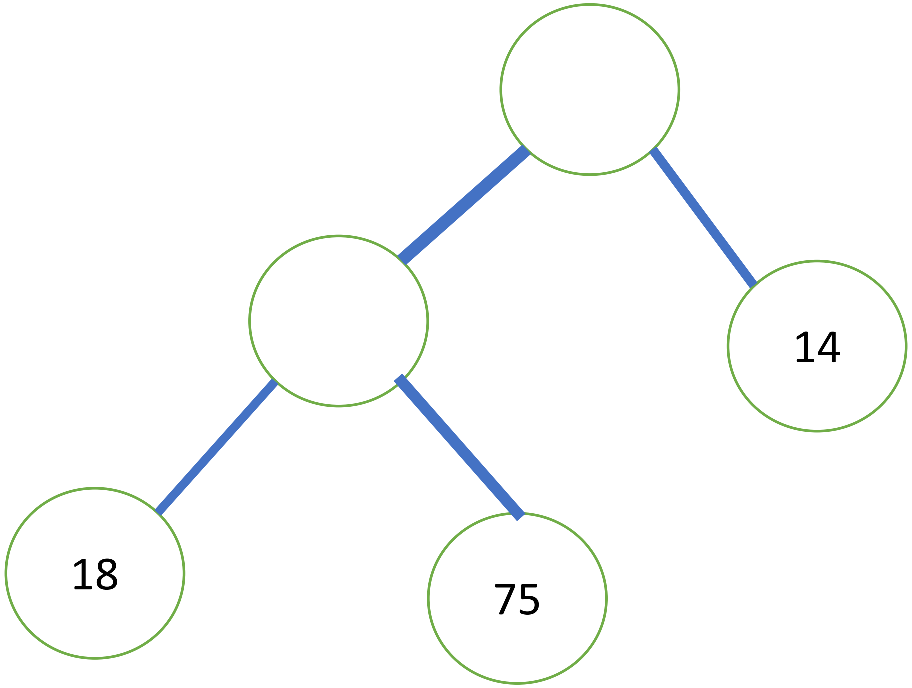
- $\lceil n/2 \rceil$ = 3 leaves are already heaps.
- There are $\lfloor n/2 \rfloor$ = 2 internal nodes.

```
build_MaxHeap_BottomUp(A, n)
        for (i <- ⌊n/2⌋ to 1) downHeap(A, i)

downHeap(A, i)
        j <- i
        k <- maxChildIndex(A, j)
        while (k !=0)
              swap(A[j], A[k])
              j <- k
              k <- maxChildIndex(A, j)
```

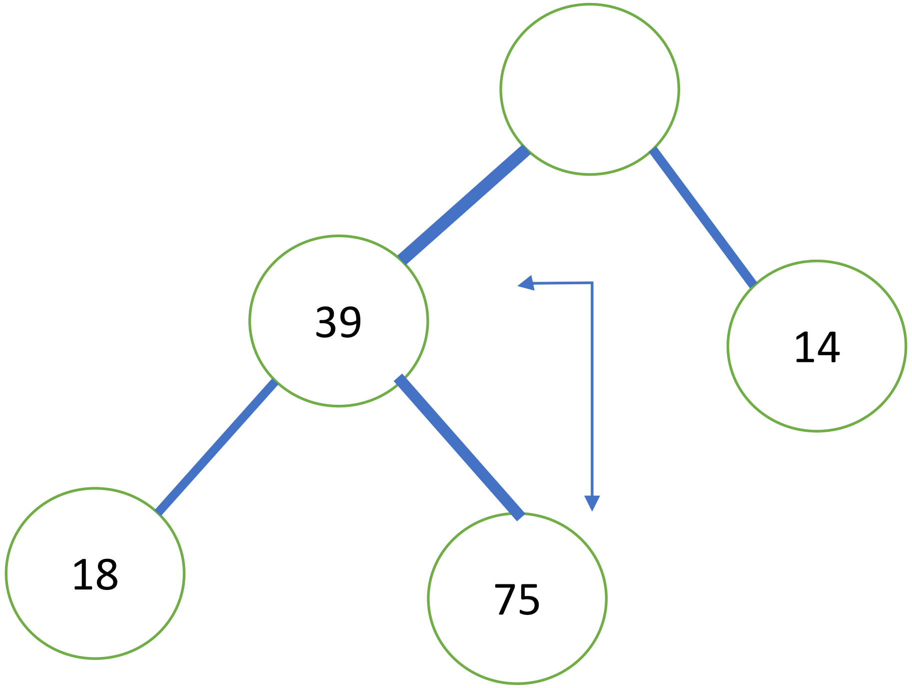
Note: maxChildIndex returns the index of a child of A[j] with maximum value among A[j], A[2j], A[2j+1]. If there is no such child it returns 0.

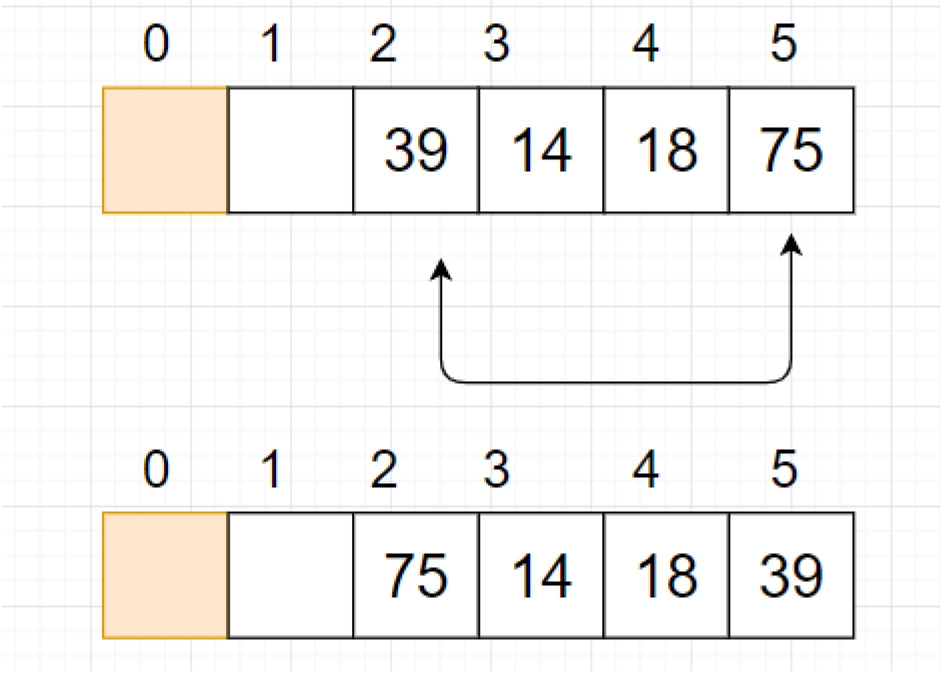
Build Max-Heap in-place Iteratively Bottom-up



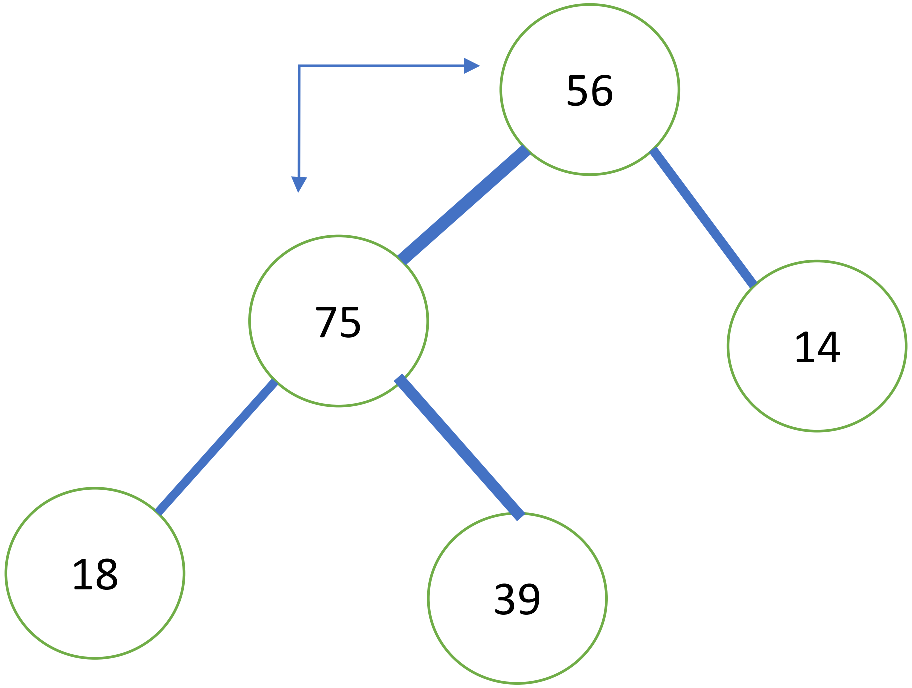
0	1	2	3	4	5
			14	18	75

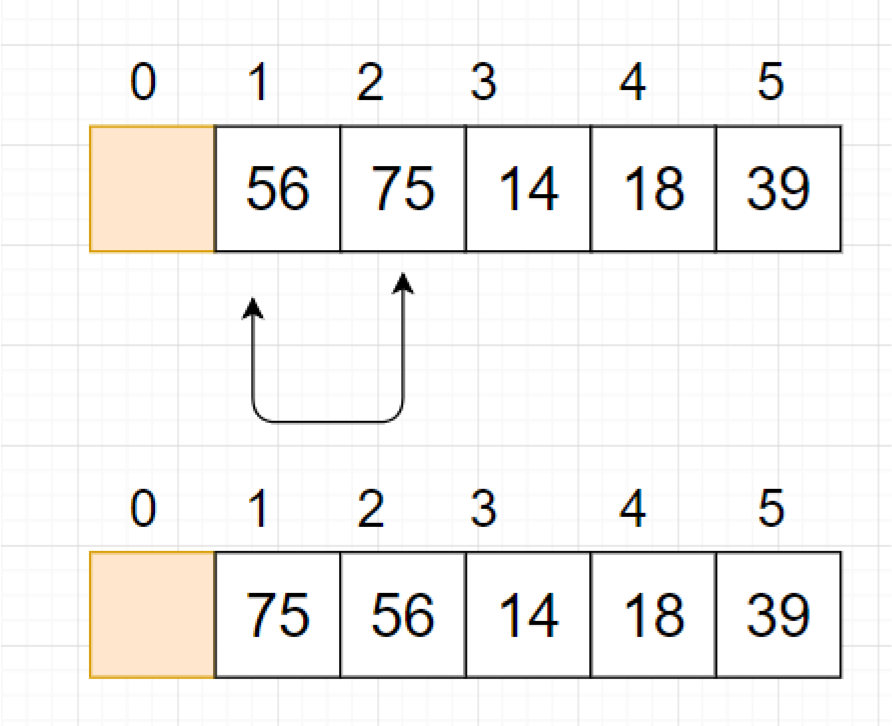
Build Max-Heap in-place Iteratively Bottom-up



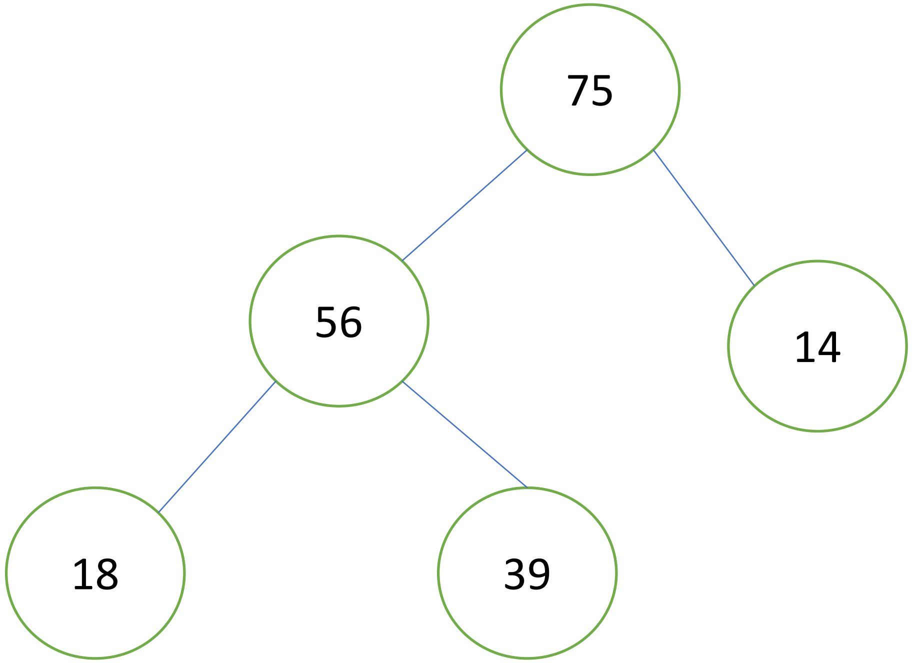


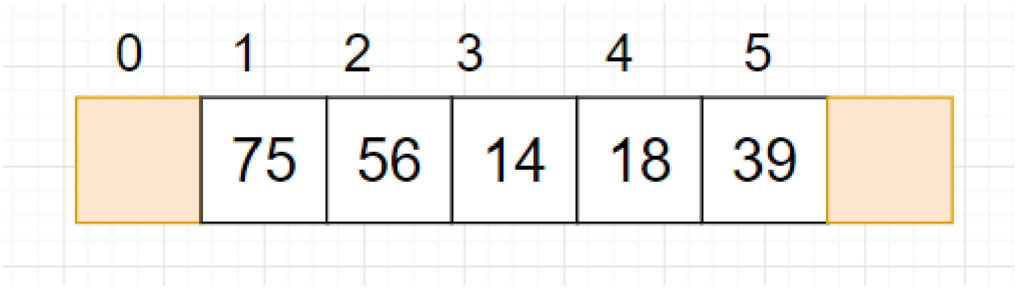
Build Max-Heap in-place Iteratively Bottom-up





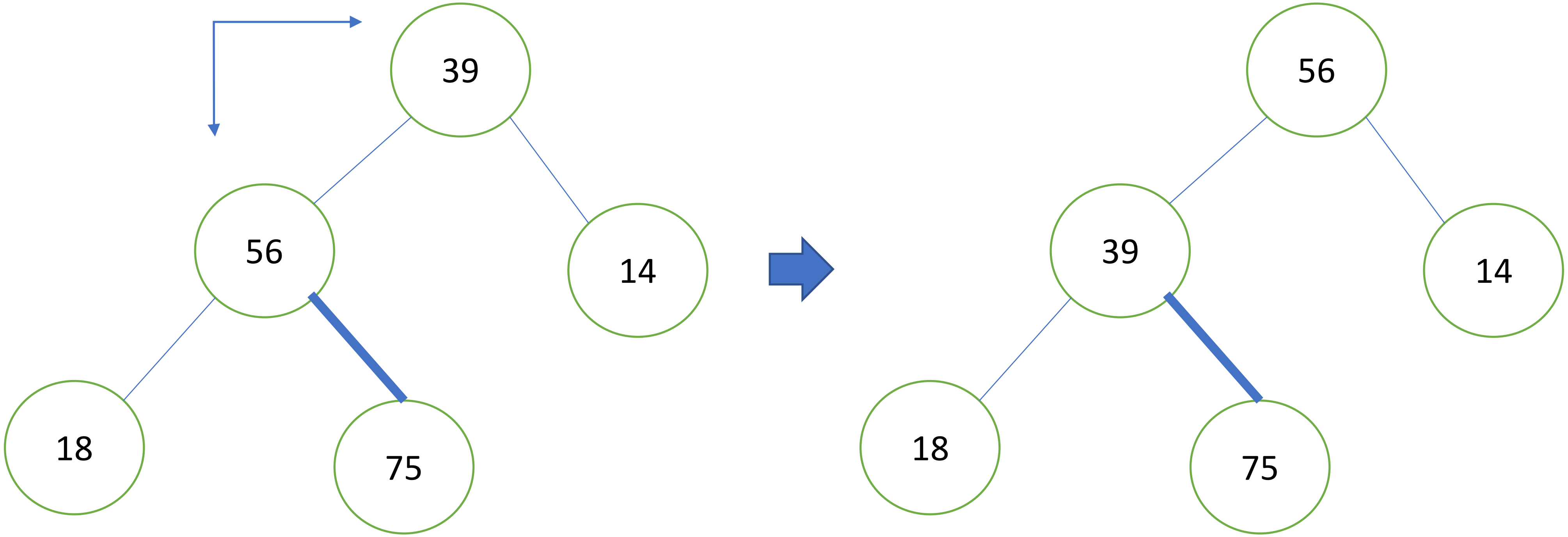
Phase II of Heap Sort



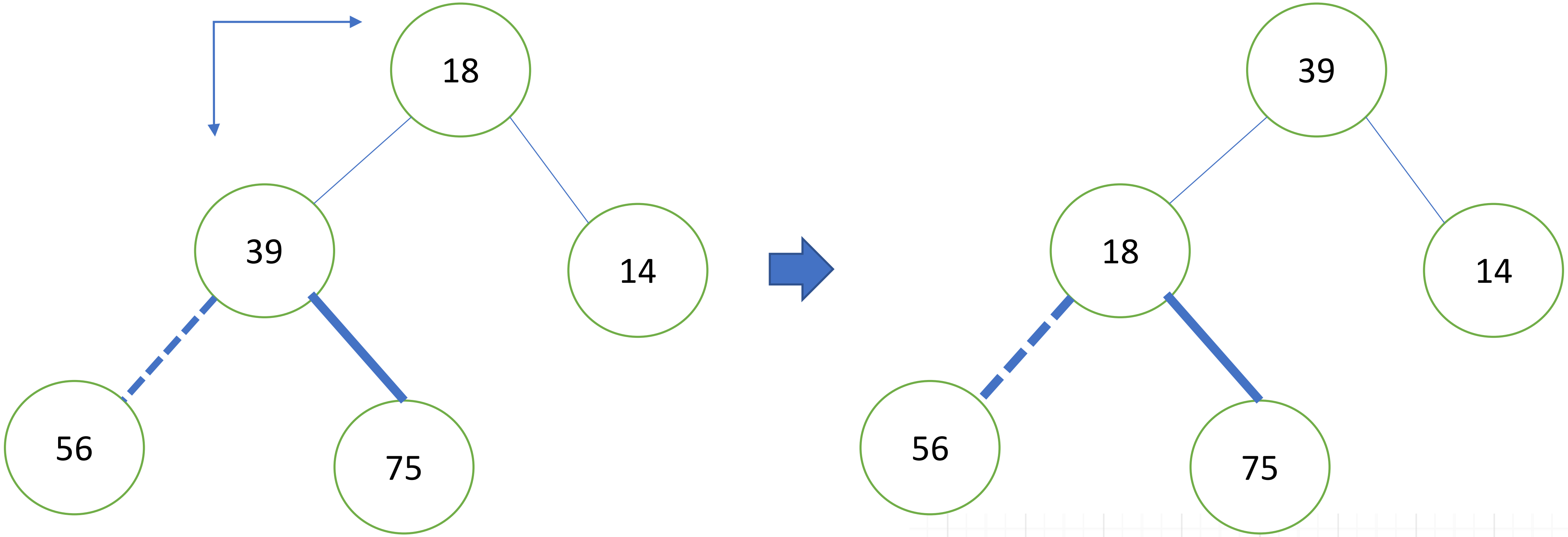


Put the root to sorted list, place the last element in the heap to the root and rebuild the heap

Phase II of Heap Sort

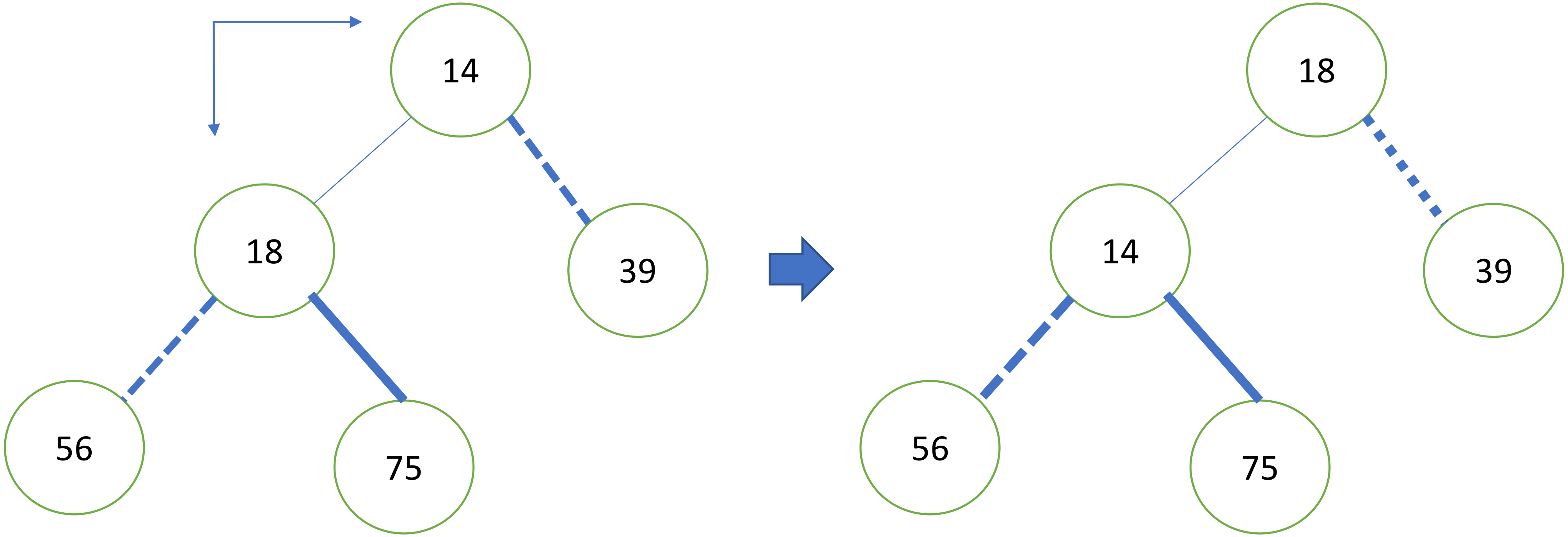


Phase II of Heap Sort



0	1	2	3	4	5
	39	18	14	56	75

Phase II of Heap Sort



0	1	2	3	4	5
	18	14	39	56	75

Phase II of Heap Sort