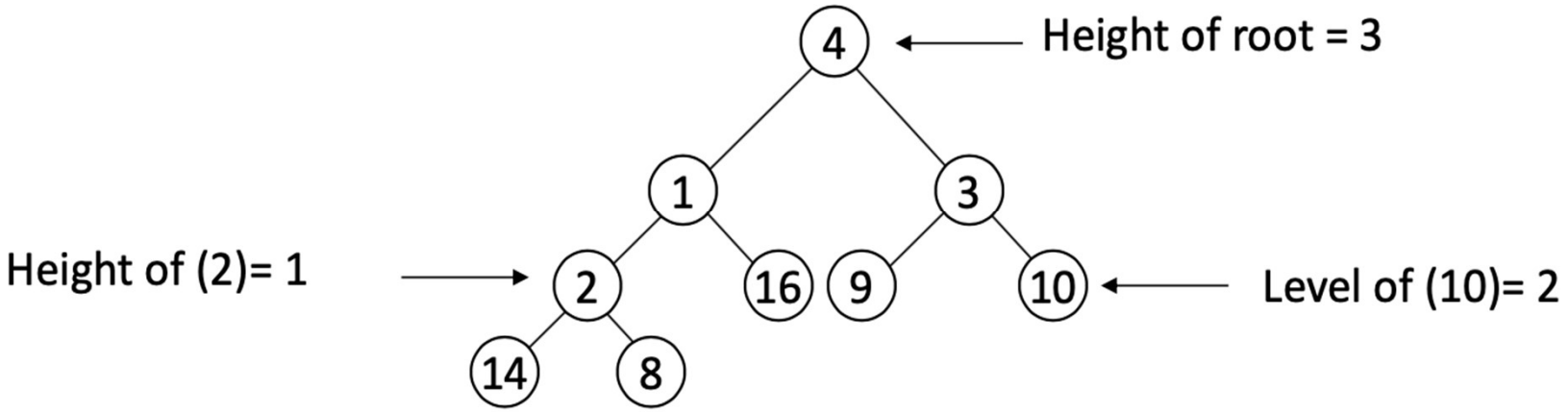
# Heaps

Dr. Sreeja S R

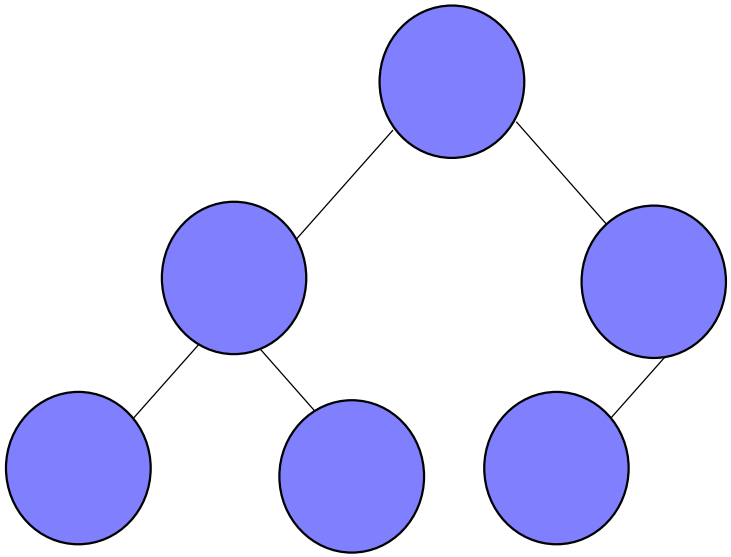
# Useful properties of binary tree

- There are **at most** $2^l$ nodes at level (or depth) $l$ of a binary tree
- A binary tree with height $d$ has **at most** $2^{d+1} - 1$ nodes
- A binary tree with $n$ nodes has height **at least** $\lfloor logn \rfloor$



# Heaps

A **heap** is a certain kind of complete binary tree.



When a complete binary tree is built, its first node must be the root.

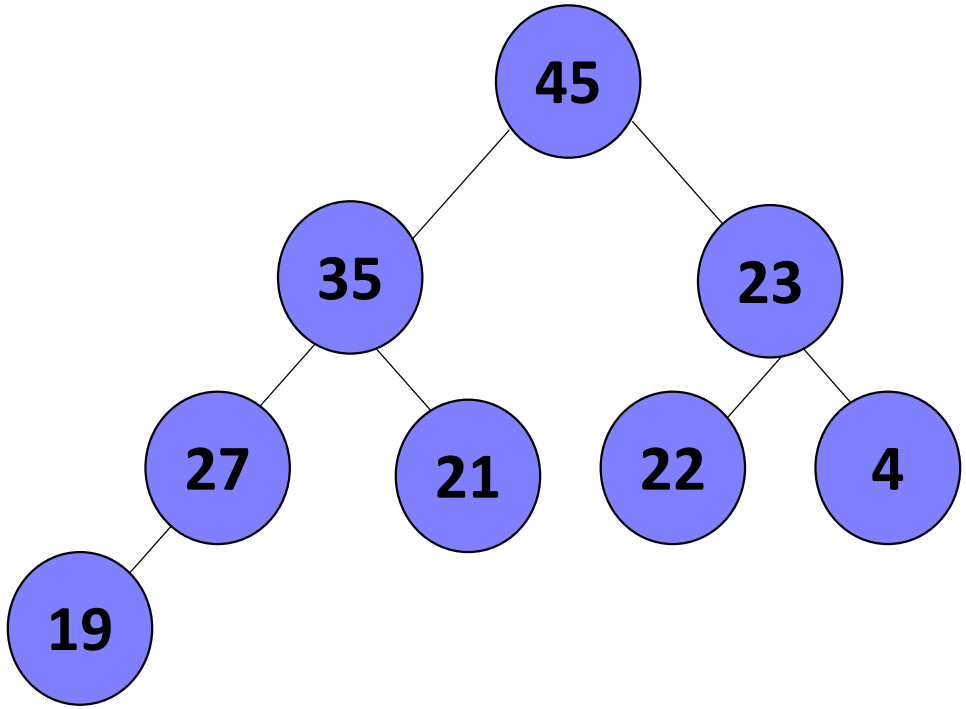
The second node is always the left child of the root.

The third node is always the right child of the root.

The next nodes always fill the next level from left-to-right.

# Heaps

- Each node in a heap contains a key that can be compared to other nodes' keys.
- Notice that this is not a binary search tree, but the keys do follow some resemblance of order.




- Can you see what rule is being enforced here?

- The *heap property* requires that each node's key is >= the keys of its children.

- This is a handy property because the biggest node is always at the top. Because of this, a heap can easily implement a priority queue (where we need quick access to the highest priority item).

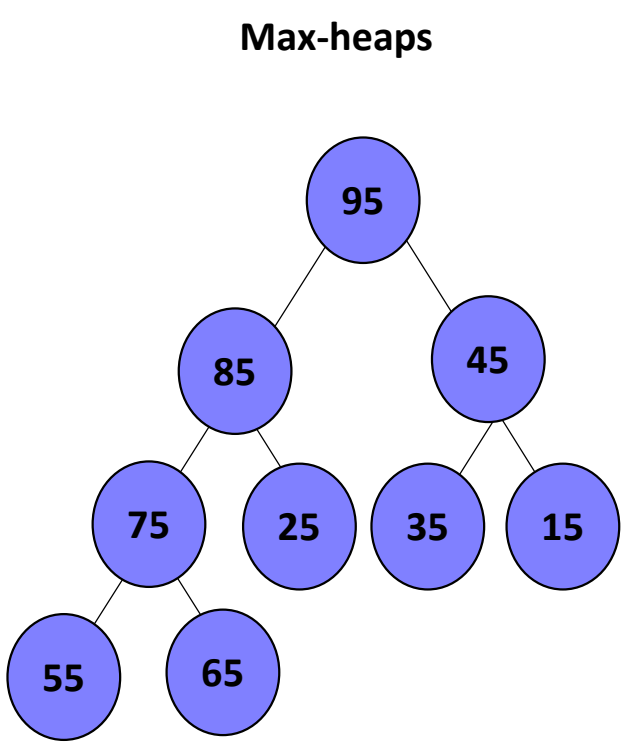
# Heap types

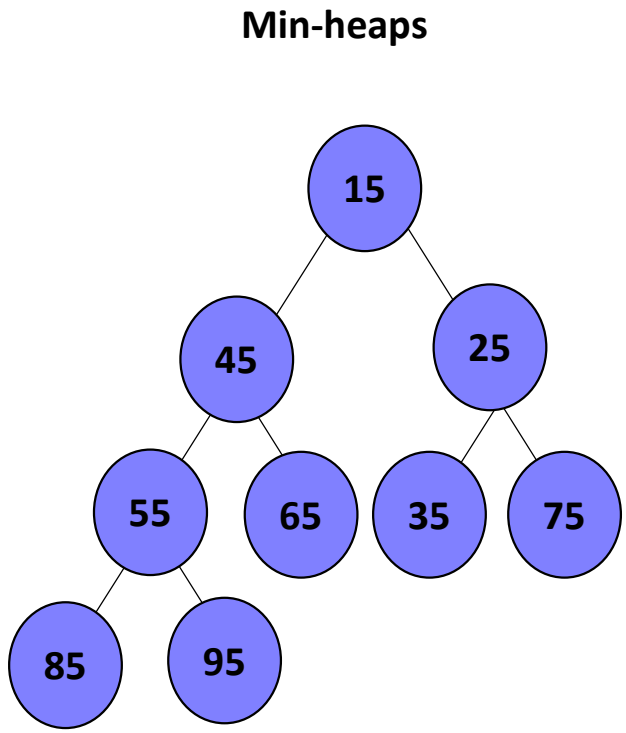
**Max-heaps** (largest element at root), have the *max-heap property:*

– for all nodes i, excluding the root: $A[PARENT(i)] \geq A[i]$

**Min-heaps** (smallest element at root), have the *min-heap property:*

– for all nodes i, excluding the root: $A[PARENT(i)] \leq A[i]$

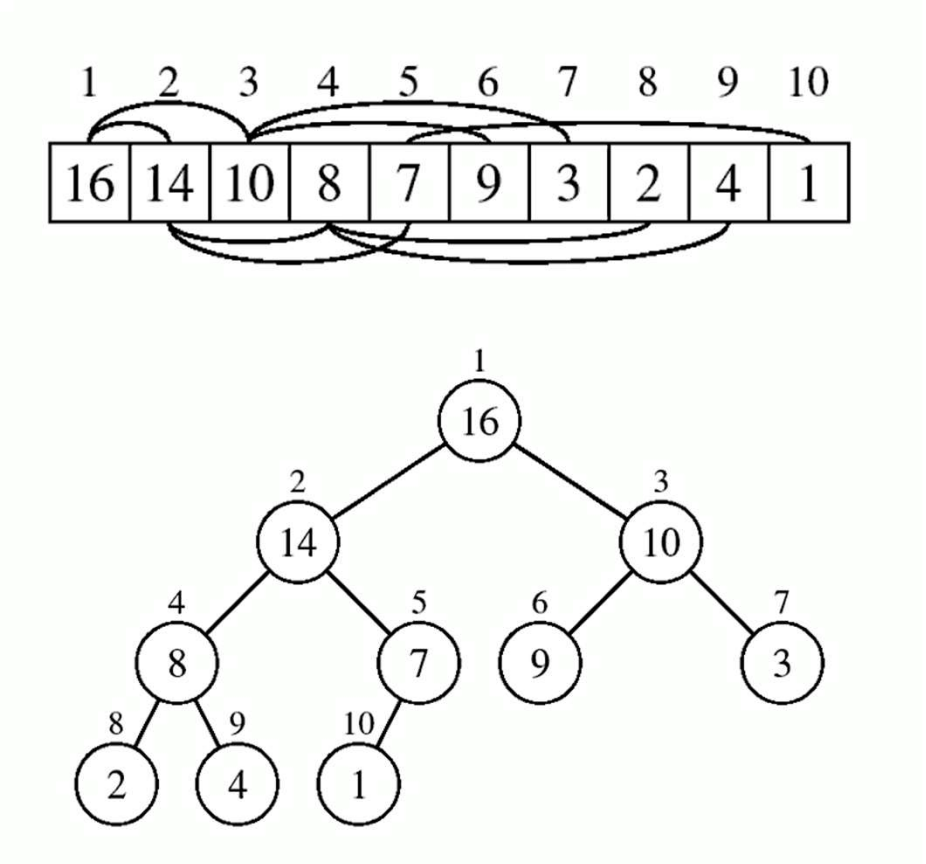


# Array representation of Heaps

- A heap can be stored as an array *A*.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$

- The elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \,..\, n]$ are leaves




No pointers required! Height of a binary heap is O(log n)

# Heap Operations

*MAX_HEAPIFY* : Maintain/Restore the max-heap property

*INSERT* : insert a new element, and reorder the heap to maintain the heap property.

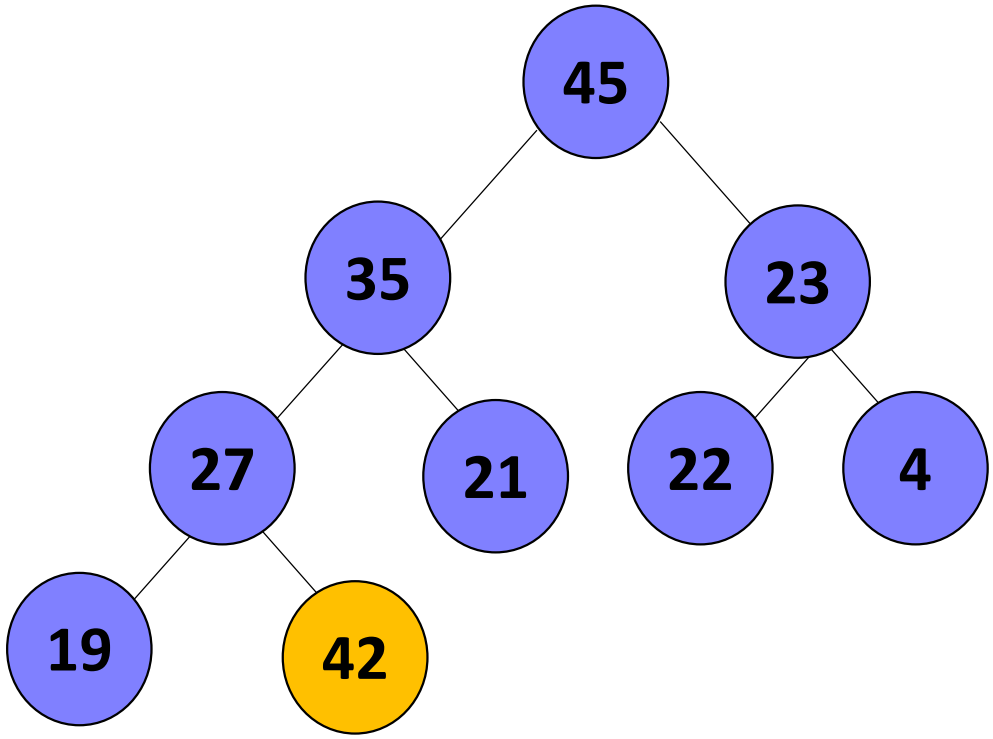
*BUILD_HEAP* : produce a max-heap/min-heap from an unordered array

EXTRACT_MAX/EXTRACT_MIN : remove the largest (smallest) element from the heap and reorder the heap to maintain the heap property.

Find max, delete, update are other auxiliary operations in heap.

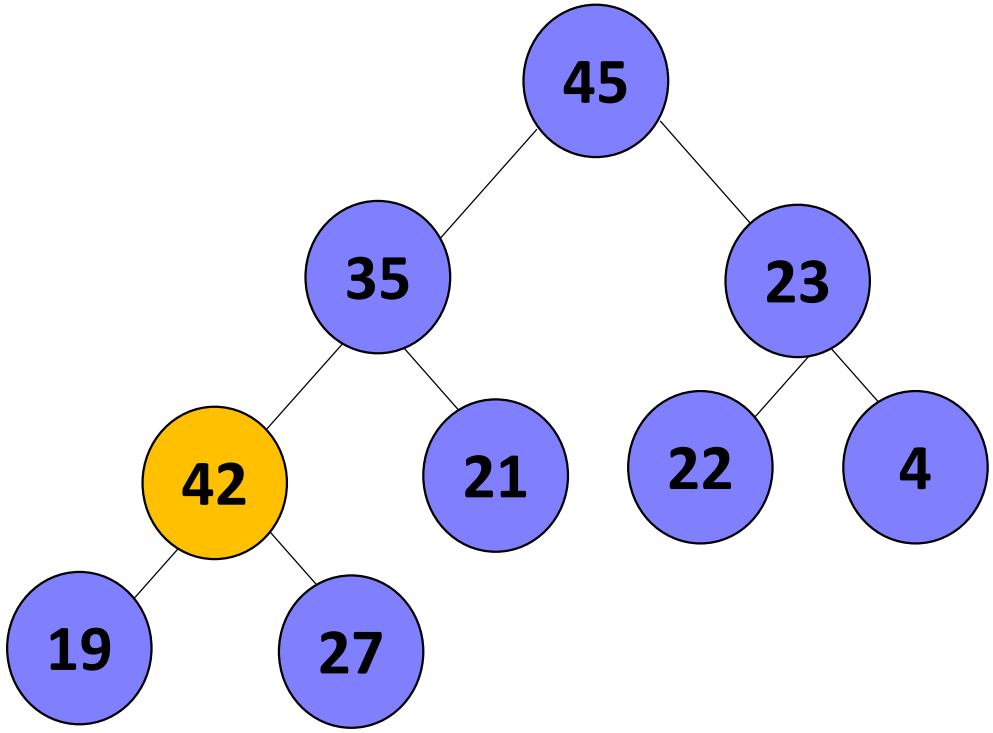
# Inserting an element to a heap

- Put the new node in the next available spot in the heap.
- The heap property is no longer valid. The new node 42 is bigger than its parent 27.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



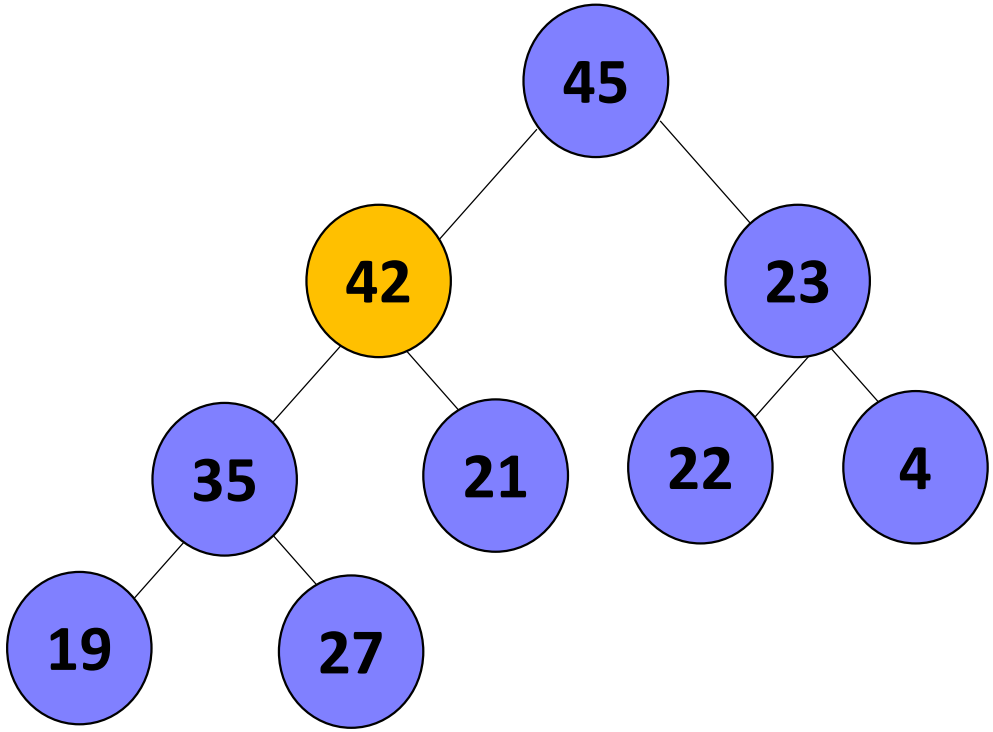
# Inserting an element to a heap

- The heap property is no longer valid. The new node 42 is bigger than its parent 35.
- Push the new node upward, swapping with its parent until the new node reaches an acceptable location.



# Inserting an element to a heap

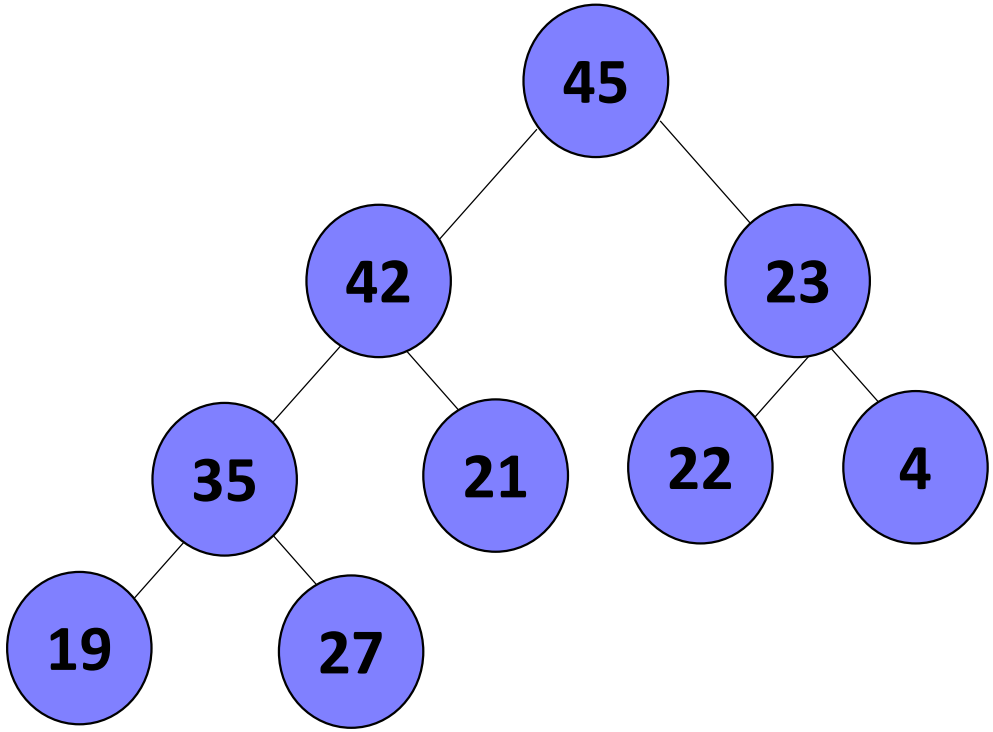
- Can we stop now? So, what are the conditions that can stop the pushing upward.



# Inserting an element to a heap

In general, there are two conditions that can stop the pushing upward:

- The parent has a key that is >= new node, or
- The node reaches the root.
- The process of pushing the new node upward is called **reheapification upward**.



# Maintaining the Heap Property

```
MAX-HEAPIFY(A, i, n)
1. l ← LEFT(i)
2. r ← RIGHT(i)
3. if l ≤ n and A[l] > A[i]
4.       then largest ← l
5.       else largest ← i
6. if r ≤ n and A[r]>A[largest]
7.       then largest ← r
8. if largest ≠ i
9.       then exchange A[i] and A[largest]
10.          MAX-HEAPIFY(A, largest, n)
```

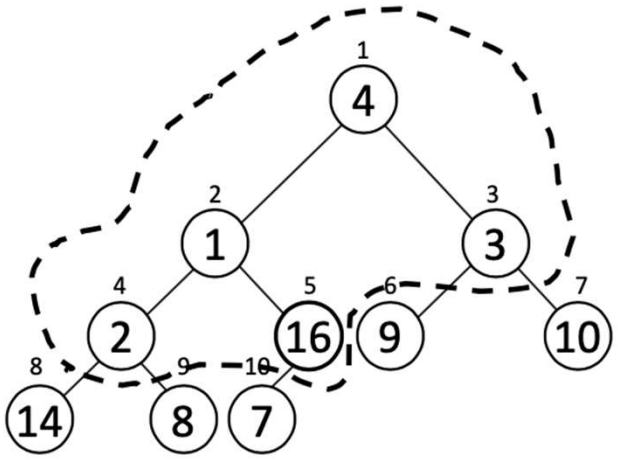
Running time of MAX-HEAPIFY is O(log n)

# Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap ($n = length[A]$)
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

*Alg:* BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n)

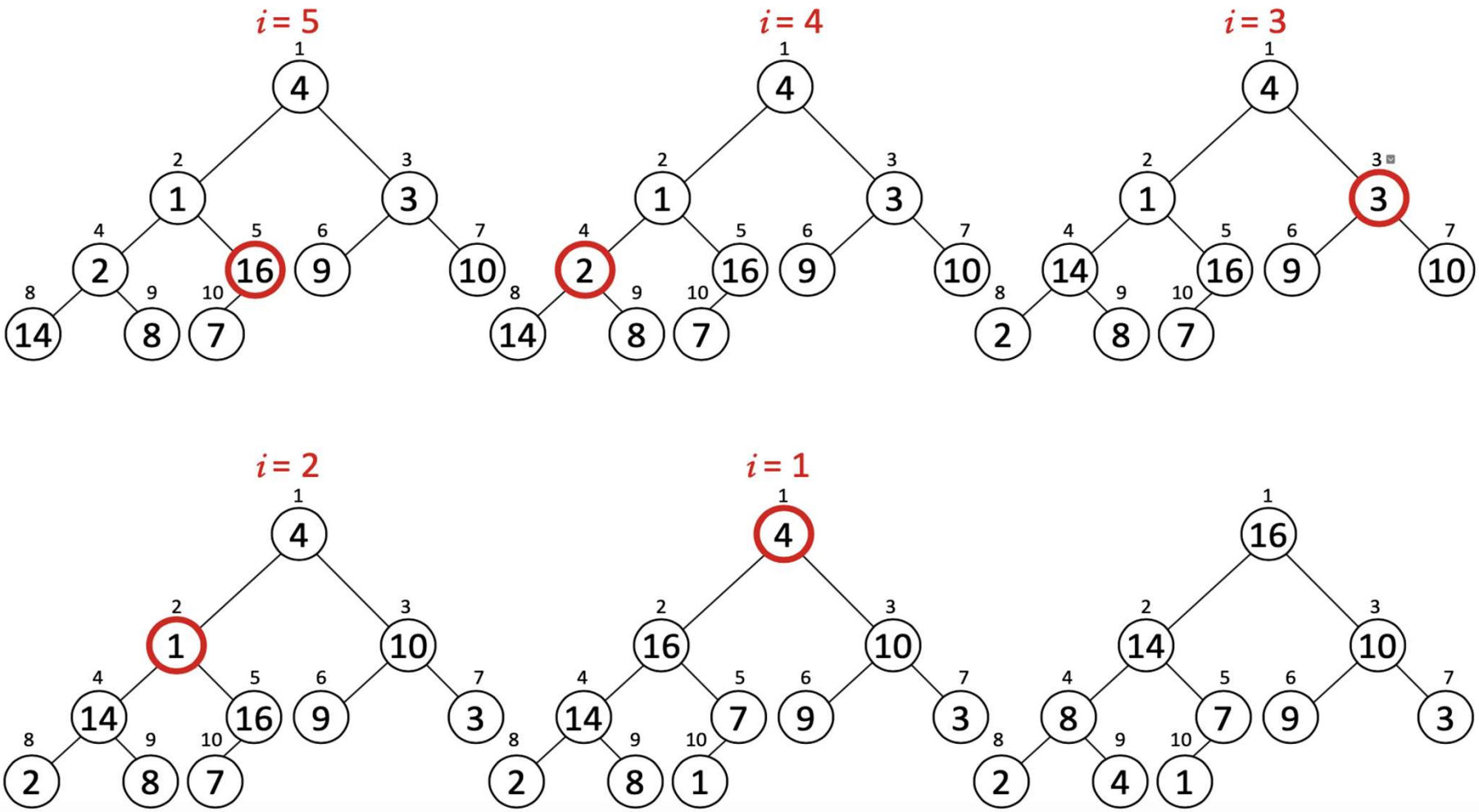



A:

| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|

# Example: A

| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|



# Running Time of BUILD-MAX-HEAP

*Alg:* BUILD-MAX-HEAP(*A*)

1. n = length[A]
2. **for** i ← ⌊n/2⌋ **downto** 1
3. **do** MAX-HEAPIFY(*A*, i, n) → O(log n)

O(n)

Running time of BUILD-MAX-HEAP: O(n logn)

# Running Time of BUILD-MAX-HEAP

- We have a loop of n/2 times, and each time we call MAX-HEAPIFY which runs in (logn). This implies a bound of O(nlogn). This is correct, but is a loose bound! We can do better.

- **Key observation:** Each time MAX-HEAPIFY is run within the loop, it is not run on the entire tree. We run it on subtrees, which have a lower height, so these subtrees do not take logn time to run. Since the tree has more nodes on the leaf, most of the time the heaps are small compared to the size of n.

So, Running time of BUILD-MAX-HEAP: O(n)

# HEAP_EXTRACT_MAX

**Goal:**

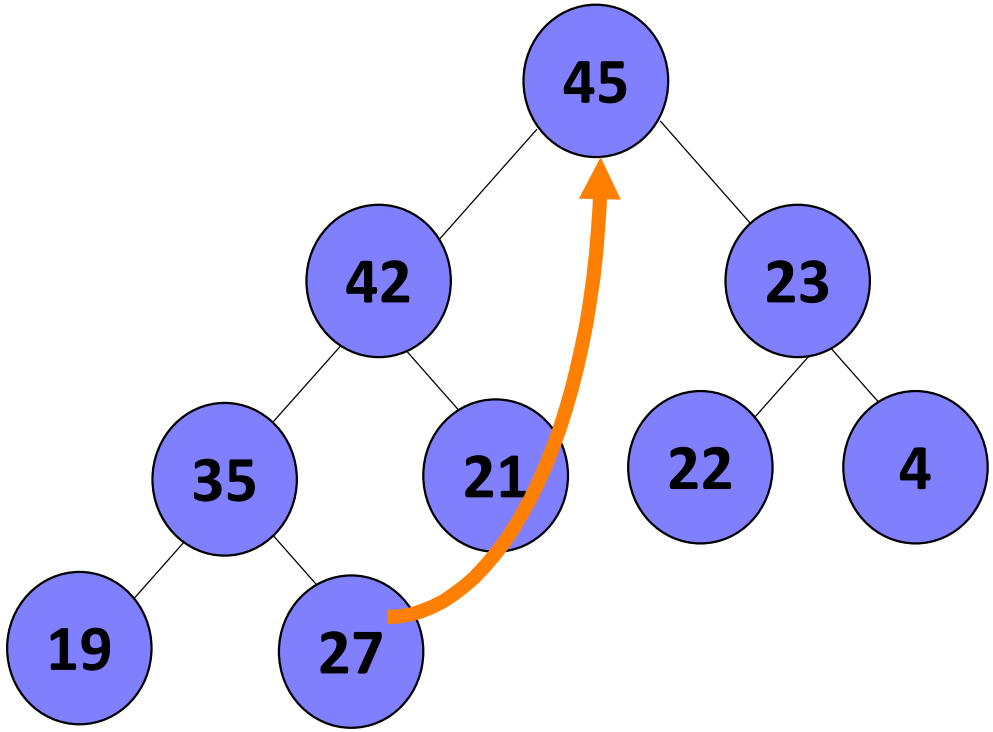
- Extract the largest element of the heap (i.e., return the max value) and also remove that element from the heap.

**Idea:**

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

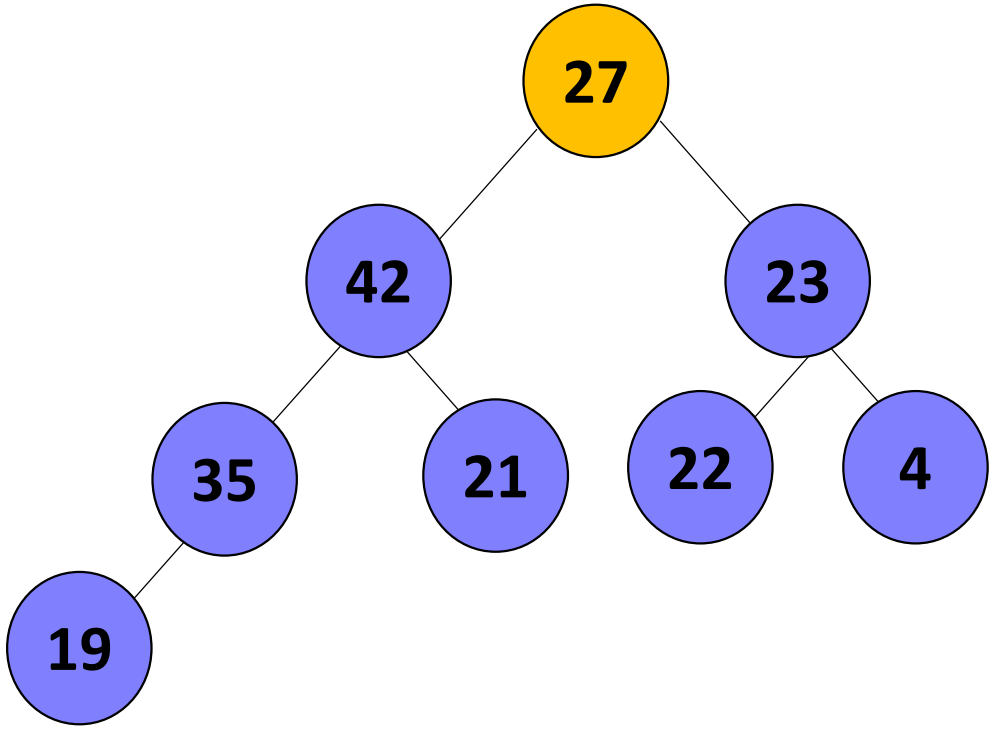
# HEAP_EXTRACT_MAX

- The first step of the removal is to move the last node of the tree onto the root.
- Decrease the size of the heap by 1 element
- In this example, 27 should be moved to root.



# HEAP_EXTRACT_MAX

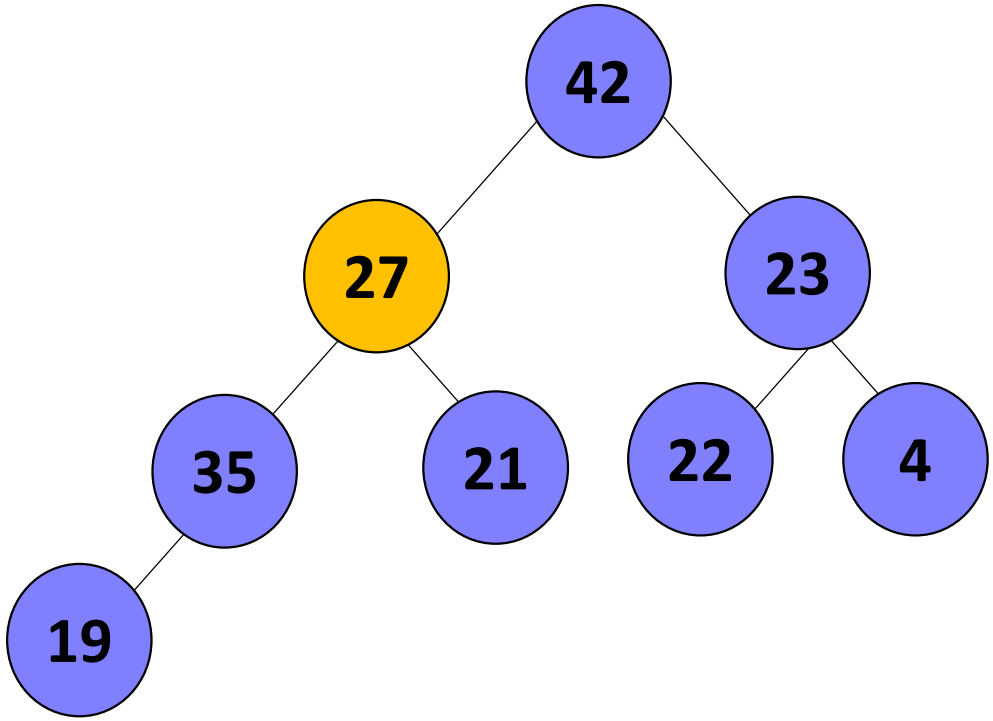
- Now the 27 is on top of the heap, and the original root (45) is no longer around. But the heap property is once again violated.
- Push the out-of-place node downward, swapping with its larger child until the new node reaches an acceptable location.




Can you guess what the downward pushing is called? reheapification downward.

# HEAP_EXTRACT_MAX

- When a node is pushed downward it is important to swap it with its largest child.
- Should we continue with the reheapification downward?

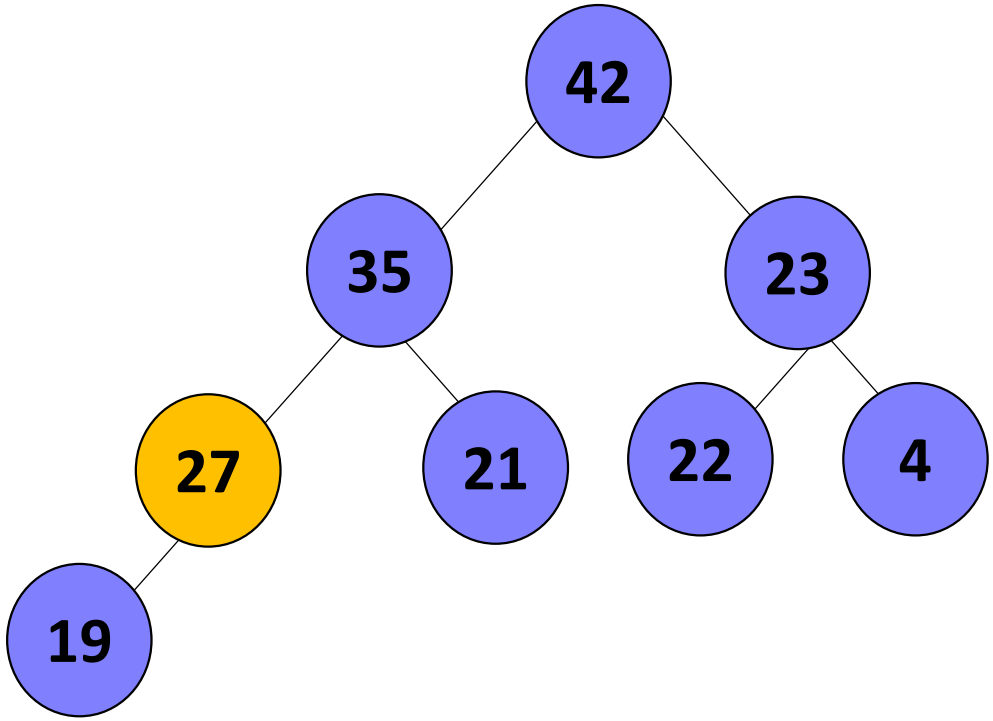


# HEAP_EXTRACT_MAX

Reheapification downward can stop under two circumstances:

1. The children all have keys that are <= the out-of-place node.

2. The out-of-place node reaches a leaf.



# HEAP_EXTRACT_MAX

Alg: HEAP-EXTRACT-MAX(A, n)

```
1. If n<1
       then error "heap underflow"
2. max ← A[1]
3. A[1] ← A[n]
4. MAX-HEAPIFY(A, 1, n-1)
5. return max
```

What is the Running time of HEAP_EXTRACT_MAX?

# Uses of Heaps

- It can be used to improve running times for several network optimization algorithms.
- It can be used to assist in dynamically-allocating memory partitions.
- A heapsort is considered to be one of the best sorting methods being in-place with no quadratic worst-case scenarios. (will see in next lecture)
- Finding the min, max, both the min and max, median, or even the k-th largest element can be done in linear time using heaps.

# Thank you!