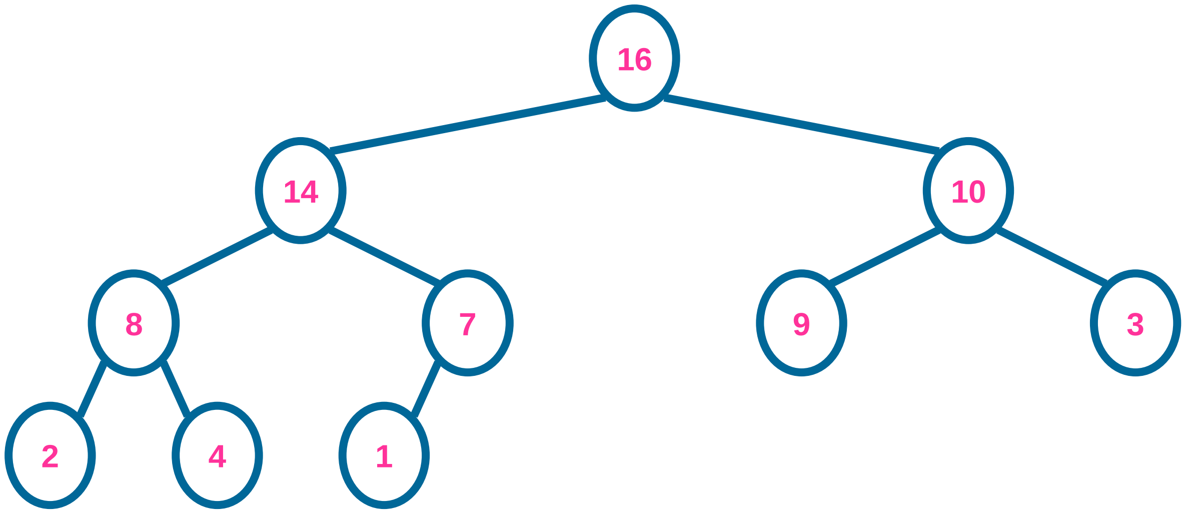
Binary Heap & Heap Sort

Heap

- A Heap is a data structure

  An array that can be seen as a nearly complete binary tree.

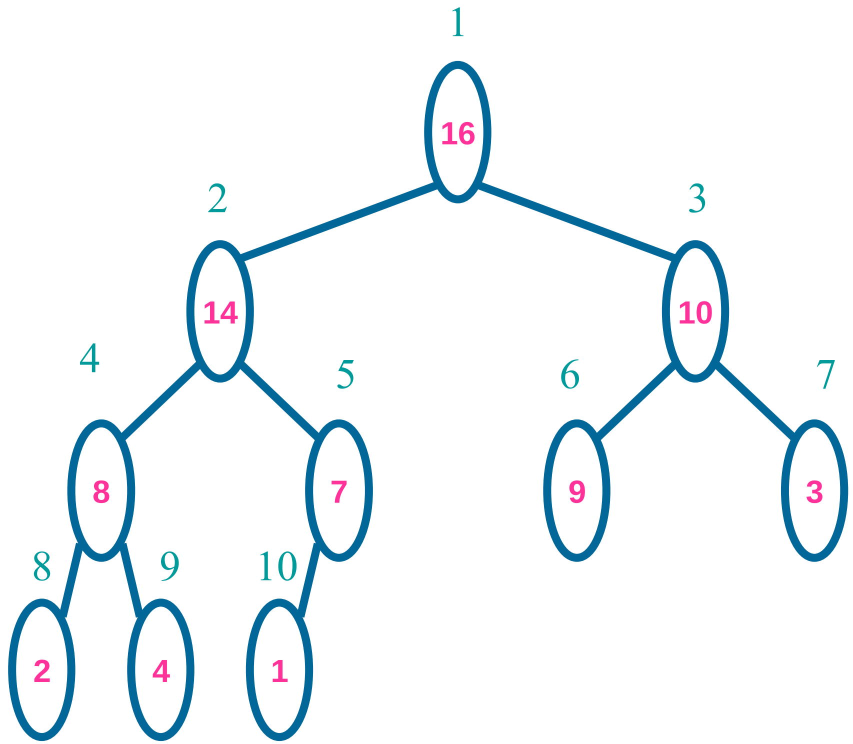
16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*



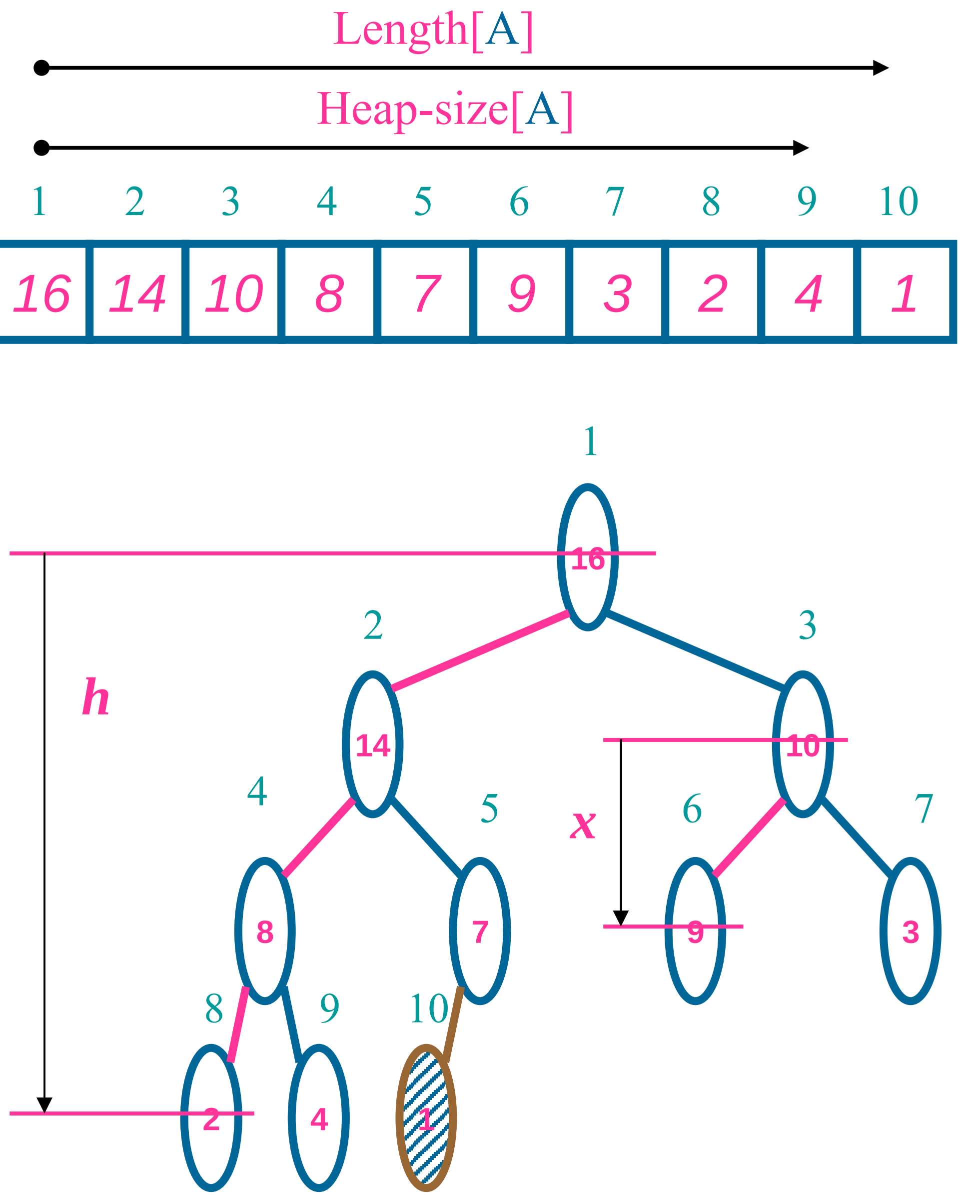
Building a Heap

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

- The root A[1]
- For any node indexed i
  - Left(i) $\Rightarrow$ $2i$
  - Right(i) $\Rightarrow 2i+1$
  - Perent(i) $\Rightarrow \lfloor i/2 \rfloor$



Definitions



- Array length ⇒ length[A]
  - # of elements in A
- Heap-size ⇒ heap-size[A]
  - # of elements in heap
- Node height (x)
  - # of edges on the longest downward path to a leaf
- Heap height (h)
  - Is the height of its root

- What is the height h of n elements heap ?
- What is the max. and min. numbers of elements in a heap with height h ?

Heap Properties

- Max heap
  - $A[\text{parent}(i)] \geq A[i]$
- Min heap
  - $A[\text{parent}(i)] \leq A[i]$

Maintaining the Max-Heap Property

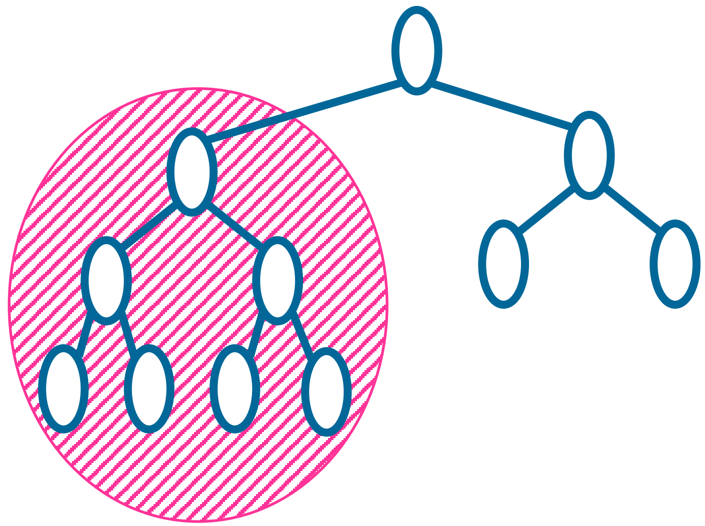
```
MAX-HEAPIFY( A, i )
l ← LEFT(i)
r ← RIGHT(i)
if  l ≤ heap-size[A] and A[l] > A[i]
        then largest ← l
        else largest ← i
if  r ≤ heap-size[A] and A[r] > A[largest]
        then largest ← r
if largest ≠ i
        then exchange A[i] ↔ A[largest]
MAX-HEAPIFY(A, largest)
```

*

Max-heapify running time

- Max-heapify (A , *i*) Module
  - running time of Max-heapify
    = Θ(1)+the time to run Max-heapify on a subtree rooted at one children of node *i*
  - A children subtree has size at most 2n/3
  - The running time of max-heapify $T(n) \le T(2n/3)+\Theta(1)$

    $T(n)=O(\lg n)=O(h)$



Worst Case children subtree

Building a heap

```
BULID-MAX-HEAP(A)
1- heap-size[A] ← length[A]
2- for i ←⌊ length[A]/2 ⌋ downto 1
3-         do MAX-HEAPIFY(A,i)
```

- Build-Max-Heap(A) Module
  - Elements in the subarray $A[(\lfloor n/2 \rfloor+1)..n]$ are all leaves
  - Apply max-heapify to all elements starting from $\lfloor n/2 \rfloor$ downto 1 to build a max-heap
  - The running time = $o(n \lg n)$
    - *(Not asymptotically tight)*

Asymptotically tighter running time of Build-max-heap

- n element heap has
  - height $h = \lfloor \lg n \rfloor$
  - At most $\lceil n/2^{h+1} \rceil$ nodes of any height h
- $$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$
- $T(n) = O(n)$

Heap Sort

```
HEAPSORT(A)
1- BUILD-MAX-HEAP(A)
2- for i ← length[A] downto 2
3-    do exchange A[1] ↔ A[i]
4-          heap-size[A] ← heap-size[A] - 1
            MAX-HEAPFIY(A,1)
```

- Calling Build-Max-Heap $\Rightarrow$ $T(n)=O(n)$
- Calling Max-heapify *n* times $\Rightarrow$ $T(n)=O(n\lg n)$
- Total running time of $= O(n \lg n)$

Priority Queues

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called *key*
- Types of priority queues
  - Max-priority queue
  - Min-priority queue

Priority Queues Operations

- Max-priority queue
  - Maximum(S)
  - Extract-Max(S)
  - Increase-Key(S , x , k)
  - Insert(S , x)
- Min-priority queue
  - Minimum(S)
  - Extract-Min(S)
  - Decrease-Key(S , x , k)
  - Insert(S , x)

Max-priority Queue Operations

- Mximmum(*S*)

```
HEAP-MAXIMUM(A)
1-  retrun A[1]
```

- Heap-Extract-Max(*A*)

HEAP-EXTRACT-MAX(A)

1- **if** *heap-size*[A] < 1

2- **then error** "underflow"

3- *max* ← A[1]

4- A[1] ← A[*heap-size*[A]]

5- *heap-size*[A] ← *heap=size*[A]-1

6- MAX-HEAPIFY (A , 1)

7- return *max*

- Heap-Increase-Key(*S* , *i*, *key*)

```
HEAP-INCREASE-KEY( A,i,key)
1-  if  key <  A[i]
2-          then error "new key is
             smaller than current key"
3- A[i] ← key
4- while i  > 1 and A[Parent(i)] < A[i]
5-     do exchange A[i] ↔ A[Parent(i)]
7-          i = Parent(i)
```

- Max-Heap-Insert(*A*, *key*)

```
MAX-HEAP-INSERT( A,key)
1-  heap-size[A] ← heap-size[A] +1
2- A[heap-size[A]] ← -∞
3- HEAP-INCREASE-KEY(A,heap-size[A],key)
```