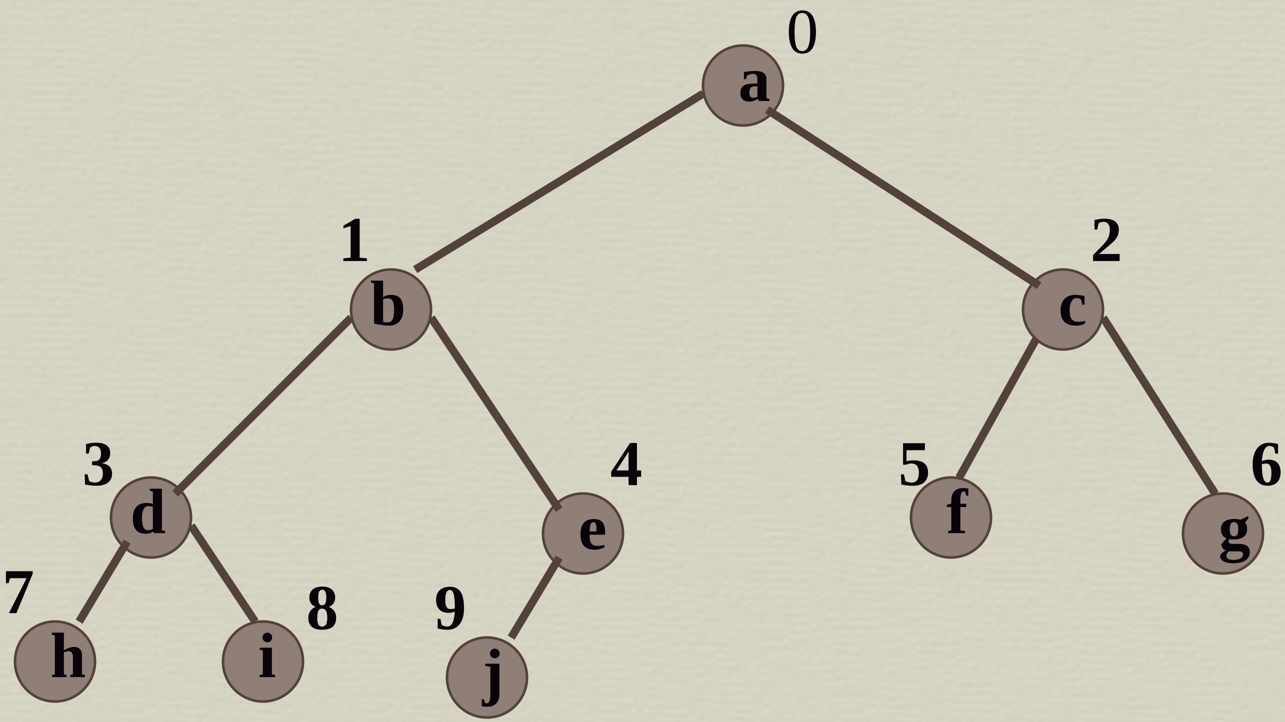
Array Representation

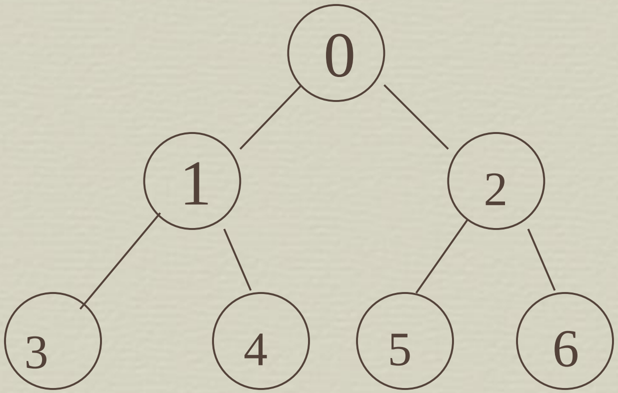
- Number the nodes using the numbering scheme for a complete binary tree. The node that is numbered **i** is stored in the array at **tree[i]**.



tree[10]=

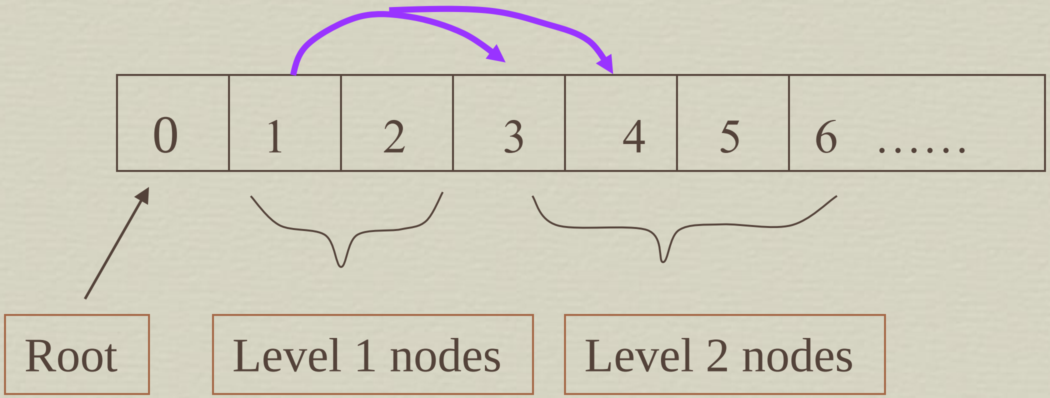
a	b	c	d	e	f	g	h	i	j
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

An array implementation of a Complete Binary Tree:



Number the nodes (starting at 0) by levels, from top to bottom and left to right within level

Parent to children (index i to $2i+1$, $2i+2$)



Complete Binary Tree Representations

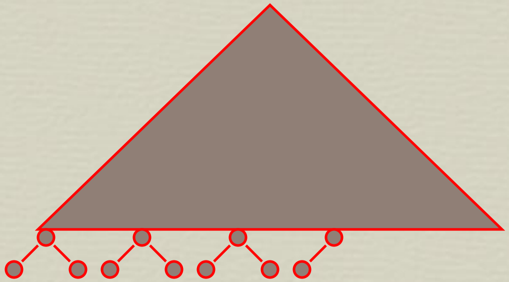
- If a complete binary tree with n nodes is represented sequentially, then for any node with index i, $0<=i<=n-1$, we have:
  - *parent*(i) is at $\lfloor (i-1)/2 \rfloor$ if $i!=0$.
    If $i=0$, i is at the root and has no parent.
  - *leftChild*(i) is at $2i+1$ if $2i+1<n$.
    If $2i+1>=n$, then i has no left child.
  - *rightChild*(i) is at $2i+2$ if $2i+2<n$.
    If $2i+2>n$, then i has no right child.

Heap

Definition:

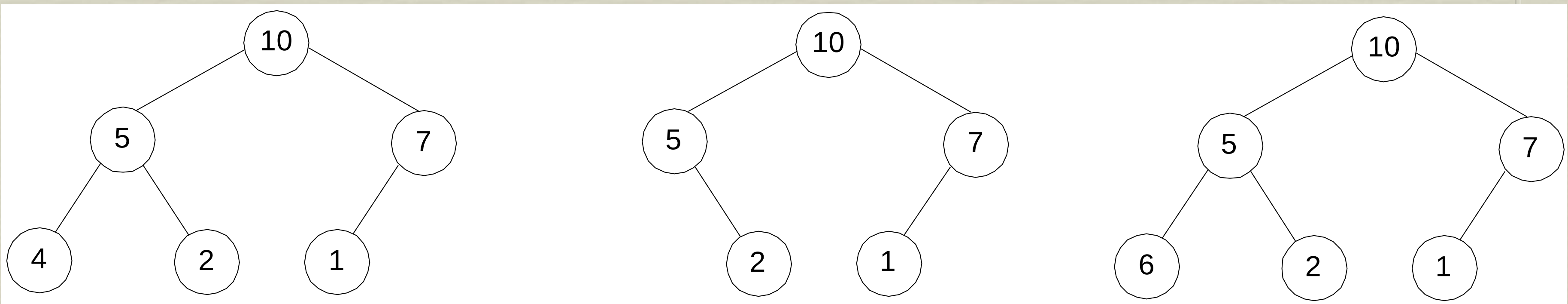
A *heap* is a binary tree with the following conditions:

- it is essentially complete: all its levels are full, except last level where only some rightmost leaves may be missing



- The key at each node is ≥ keys at its children

Example



a heap **not a heap** **not a heap**

Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered from left to right

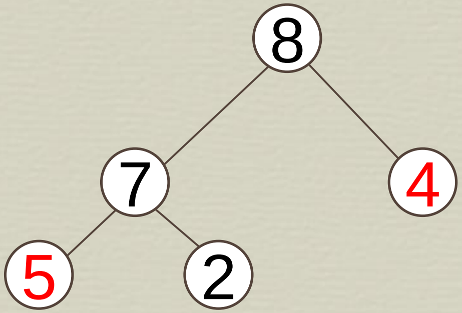
The Heap Data Structure

Def: A heap is a *complete binary tree*

with the following two properties:

- Structural property: all levels are full, except possibly the last one, which is filled from left to right
- Order (heap) property: for any node x

Parent(x) ≥ x



It doesn't matter that 4 in level 1 is smaller than 5 in level 2

Heap (top to bottom and left to right)

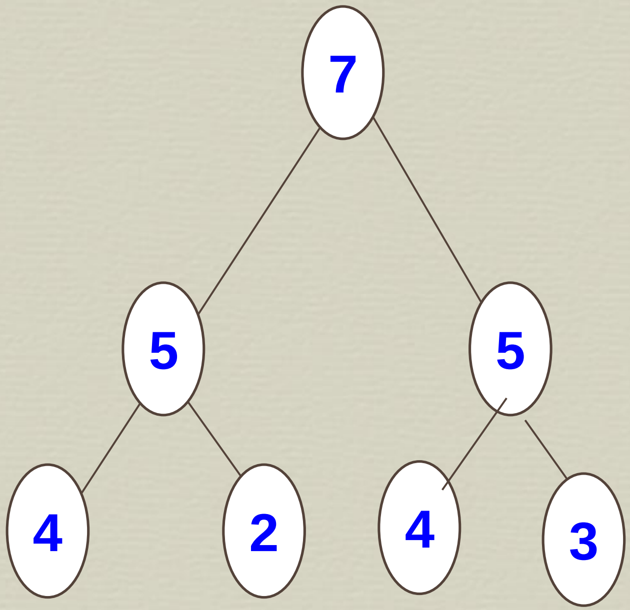
Heap Types

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

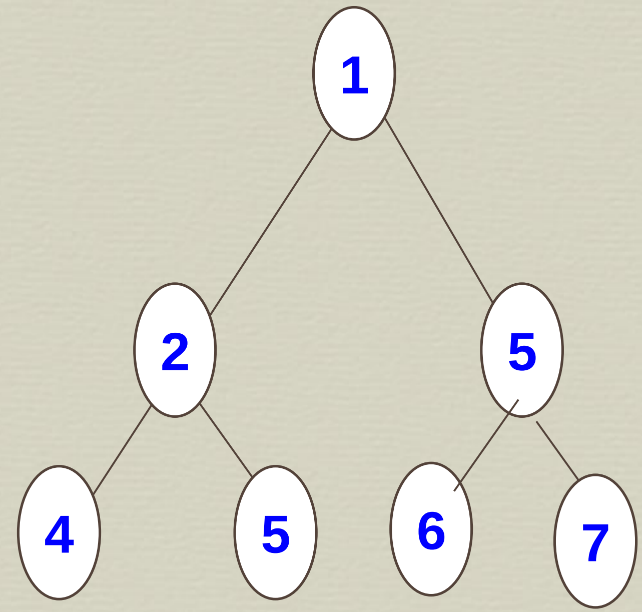
    PARENT $\geq$ child

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

    PARENT $\leq$ child



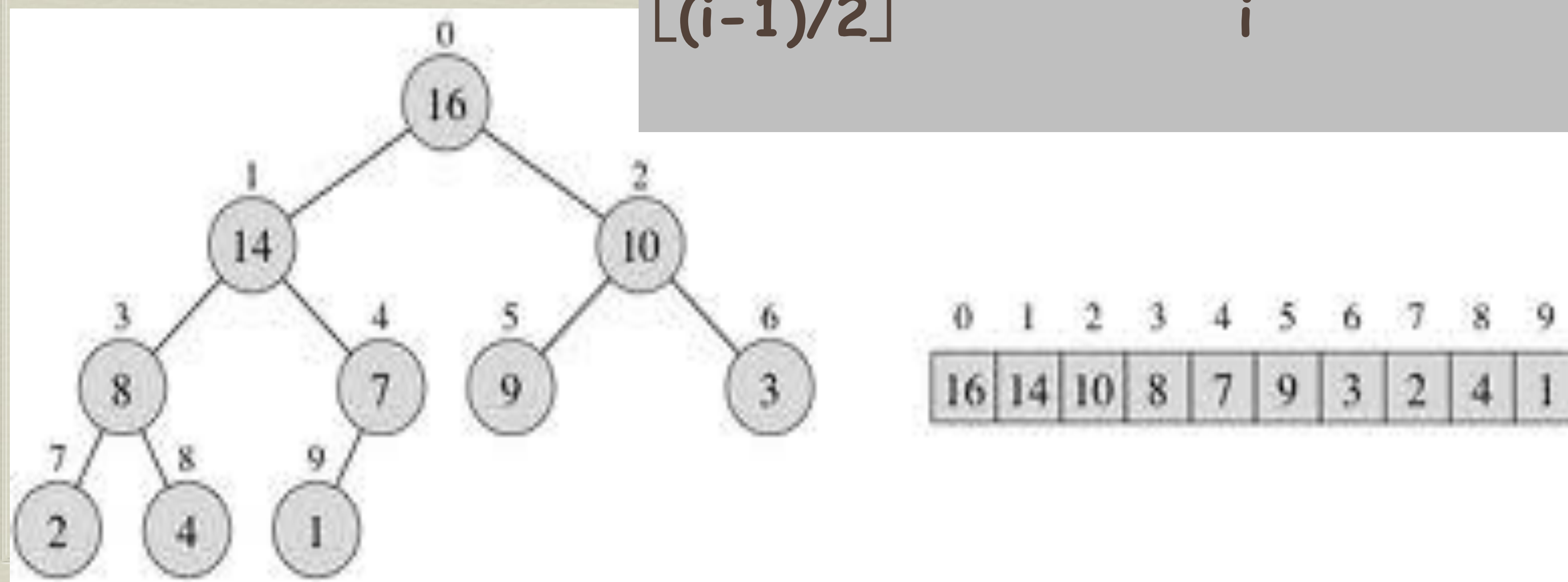
Max-heaps



Min-heaps

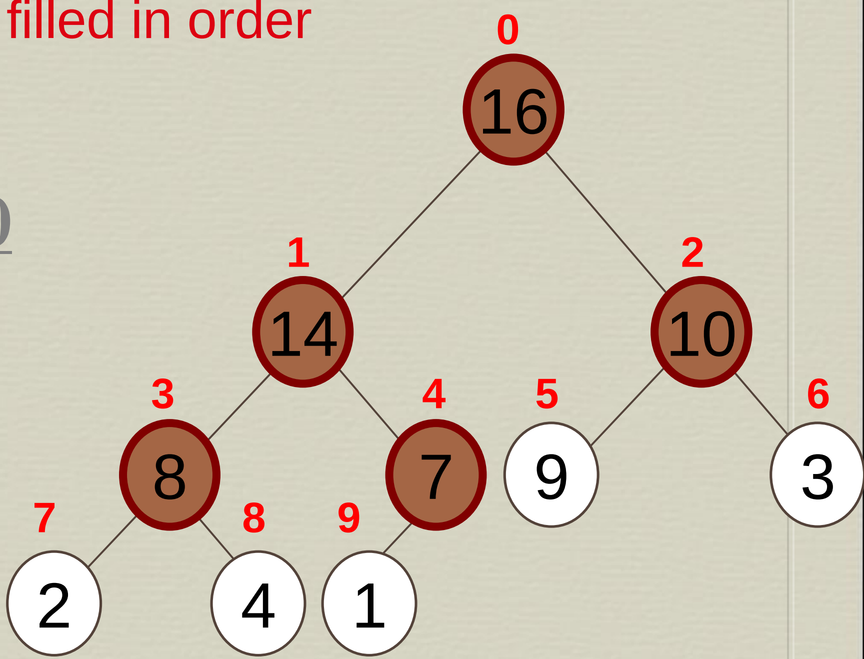
Heap as an array

Parent index	left Child index	right Child index
i	$2i+1$	$2i+2$
$\lfloor (i-1)/2 \rfloor$	i	



Array Representation of Max Heaps

- A heap is a complete binary tree that is filled in order



When index i is from 0 to $n-1$ and

number of element = length of A[]= $n = 10$

- Root of tree is $A[0] = 16$
- Left child of $A[i] = A[2i+1]$
- Right child of $A[i] = A[2i + 2]$
- Parent of $A[i] = A[\lfloor (i-1)/2 \rfloor]$
- Heapsize[A] ≤ length[A]

- The elements in $A[\lfloor n/2 \rfloor] \ldots A[n-1]$ are leaves i.e. A[5] to A[9]
- Parents are $A[0] \ldots\ldots\ldots A[\lfloor n/2 \rfloor - 1]$ i.e. A[0] to A[4]

- The root have the maximum element of the heap

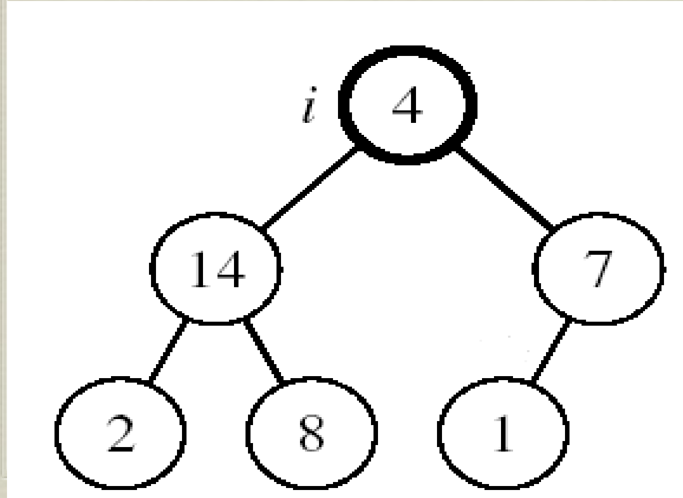
Operations on Heaps

- Maintain the max-heap property
  - MAX-HEAPIFY
- Create a max-heap from an unordered array
  - BUILD-MAX-HEAP
- Sort an heap array
  - HEAPSORT

Maintaining the Heap Property

When:

- Left and Right subtrees of i are max-heaps and
- A[i] breaks the heap property.
- A[i] may be smaller than its children



How to return on heap?

1. Compare i^{th} value with it's left and right child key value to find the largest one.
2. If Largest value is in largest_index
3. And If the largest_index is not equal to i, i.e. largest value is not in i^{th} position then exchange the i^{th} value with largest value.
4. And repeat the recursive procedure for largest_index instead of i.

MAX-HEAPIFY(*A*, i, n)

1. LEFT_child = 2i+1
2. RIGHT_child =2i+2
3. If LEFT_child < *n* and *A*[LEFT_child] > *A*[i]

   then *largest_index* = LEFT_child

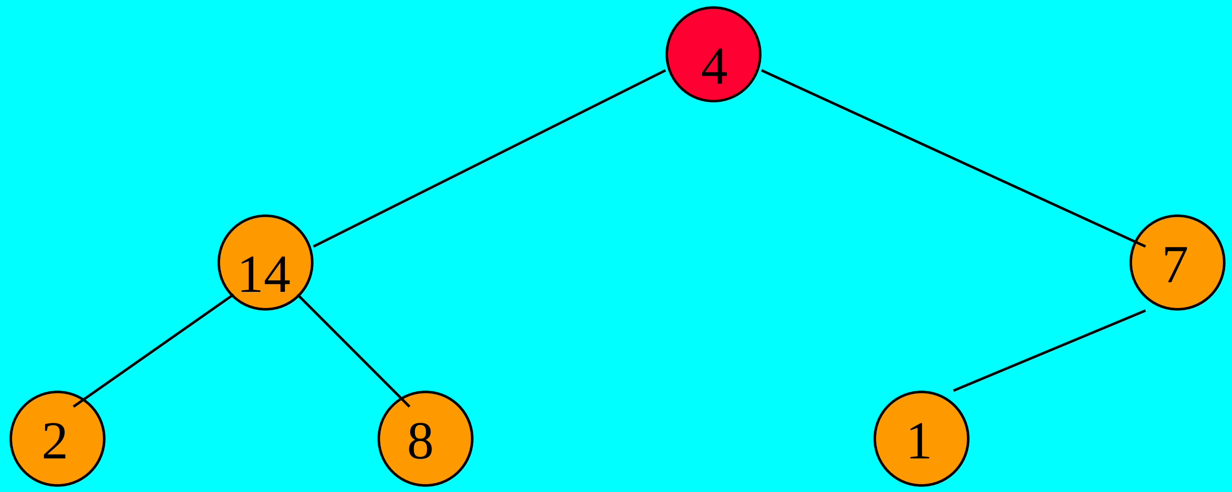
   else *largest_index* = i
4. If RIGHT_child < *n* and *A*[RIGHT_child] > *A*[*largest_index*]

   then *largest_index* = RIGHT_child
5. if *largest_index* ≠ i

   then exchange *A*[i] ↔ *A*[*largest_index*]

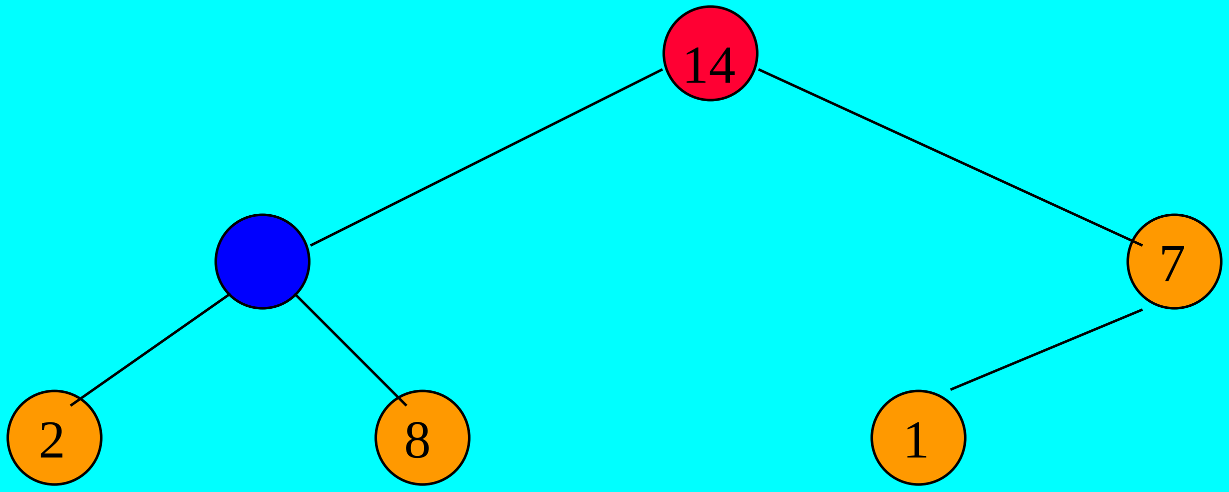
   MAX-HEAPIFY(*A*, *largest_index*, n)

Single node adjustment to maintain the Heap Property in a Max Heap

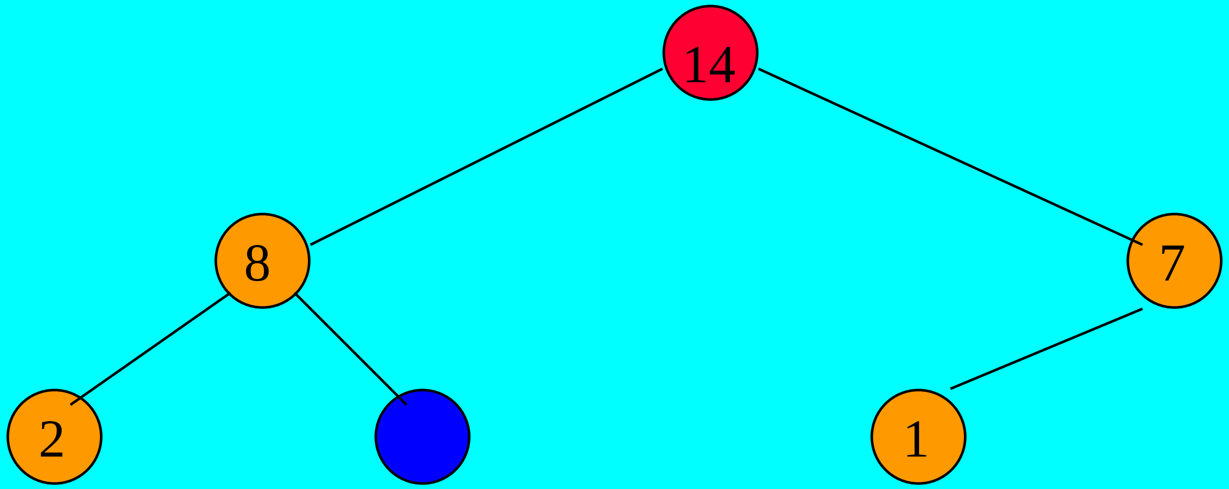


Find the position (home) for 4.

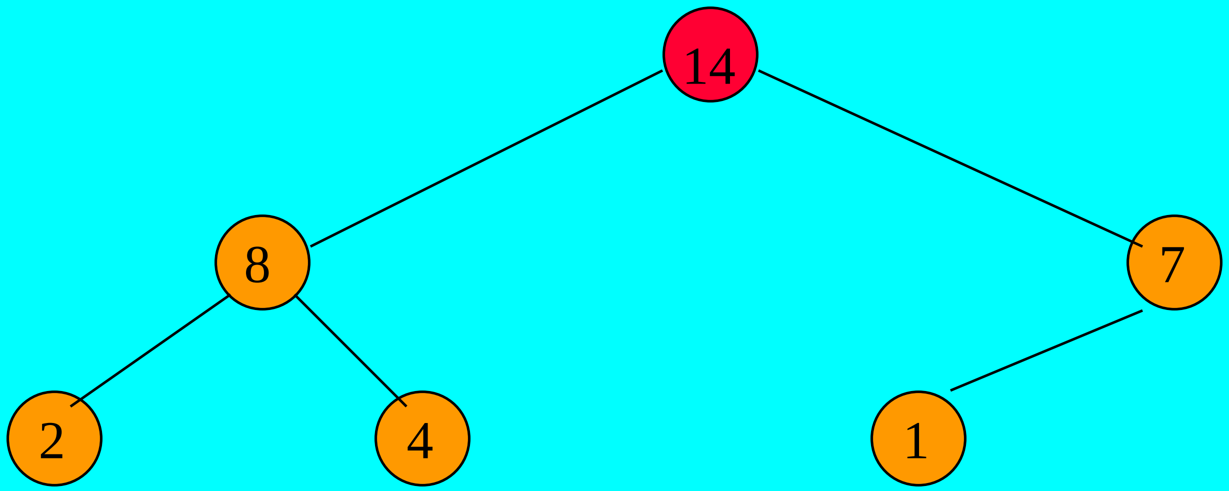
Adjusting a Max Heap with heapify() function



continuation…..



continuation…..



Done.

Building a Heap

- **Convert an array $A[0 \ldots n-1]$ into a max-heap**

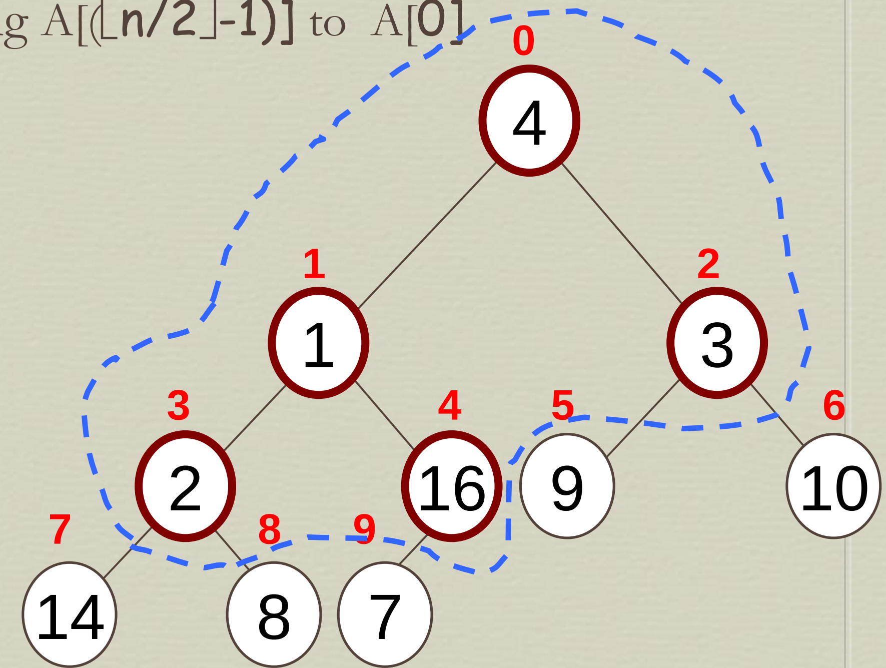
  when, n = length of A[] = number of element

- The elements in the sub-array $A[\lfloor n/2 \rfloor] \ldots\ldots A[n-1]$ are leaves

- Apply MAX-HEAPIFY() on elements among $A[(\lfloor n/2 \rfloor-1)]$ to $A[0]$

Alg: BUILD-MAX-HEAP(A)

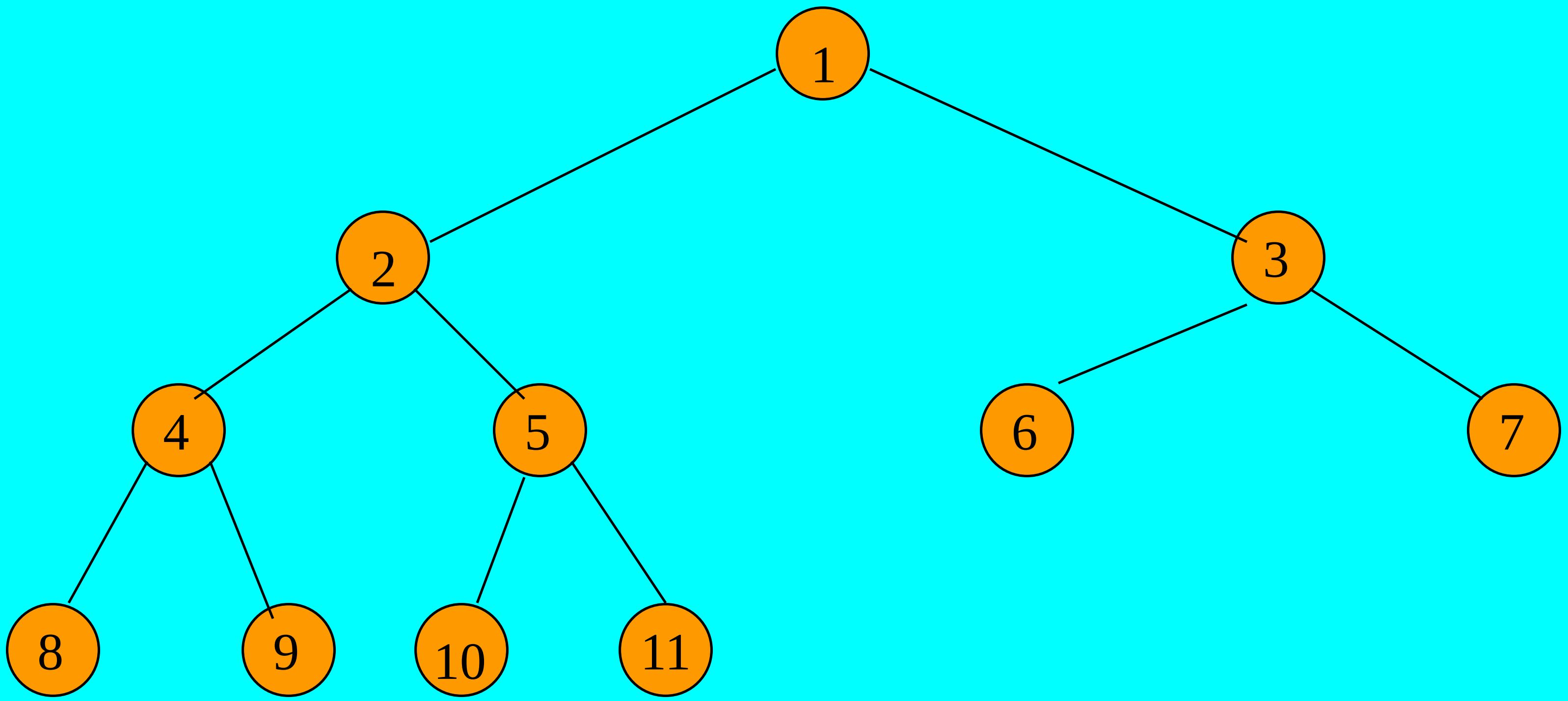
1. n = length of A[] i.e. no of element
2. for i = from $\lfloor n/2 \rfloor$-1 down to 0
3. do MAX-HEAPIFY(A, i, n)



A:

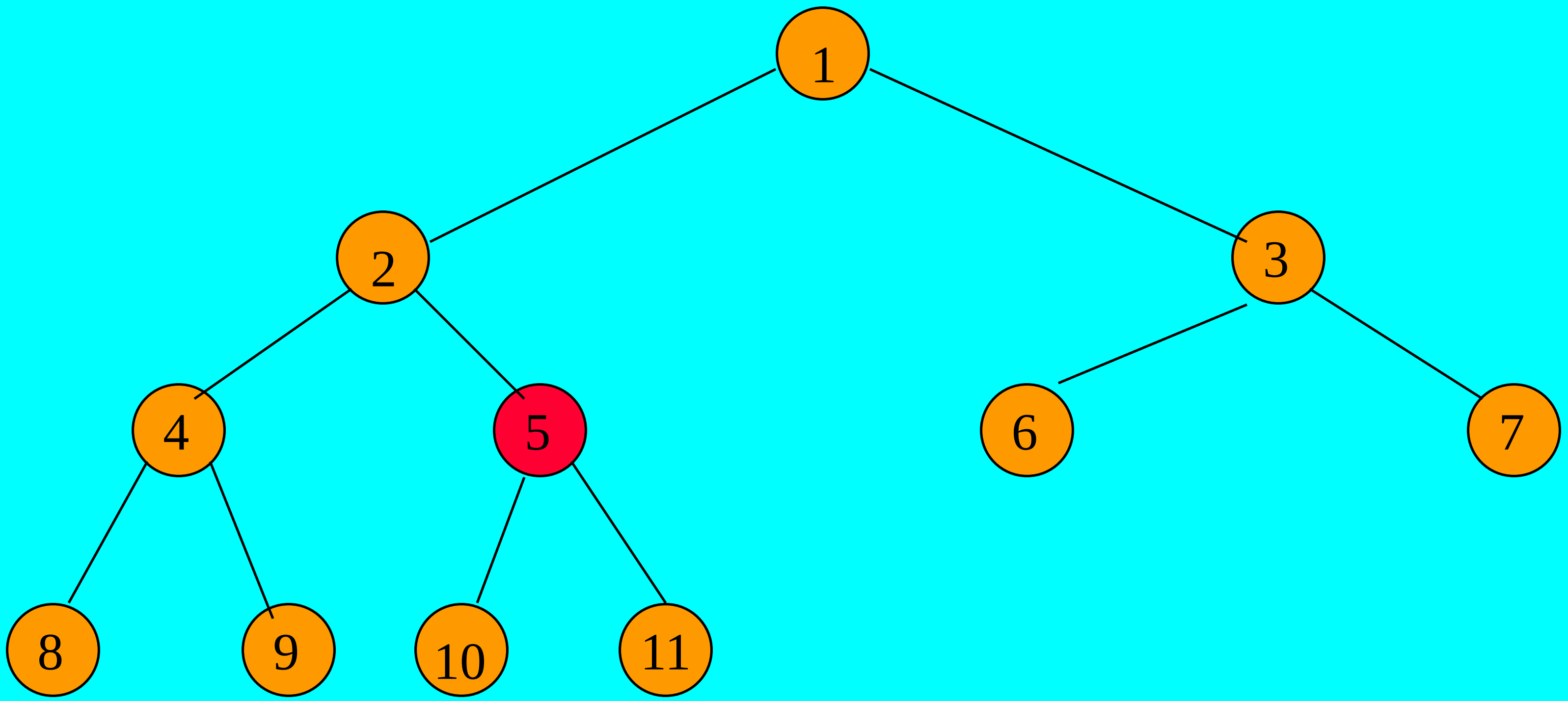
4	1	3	2	16	9	10	14	8	7

Building A Max Heap



input array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

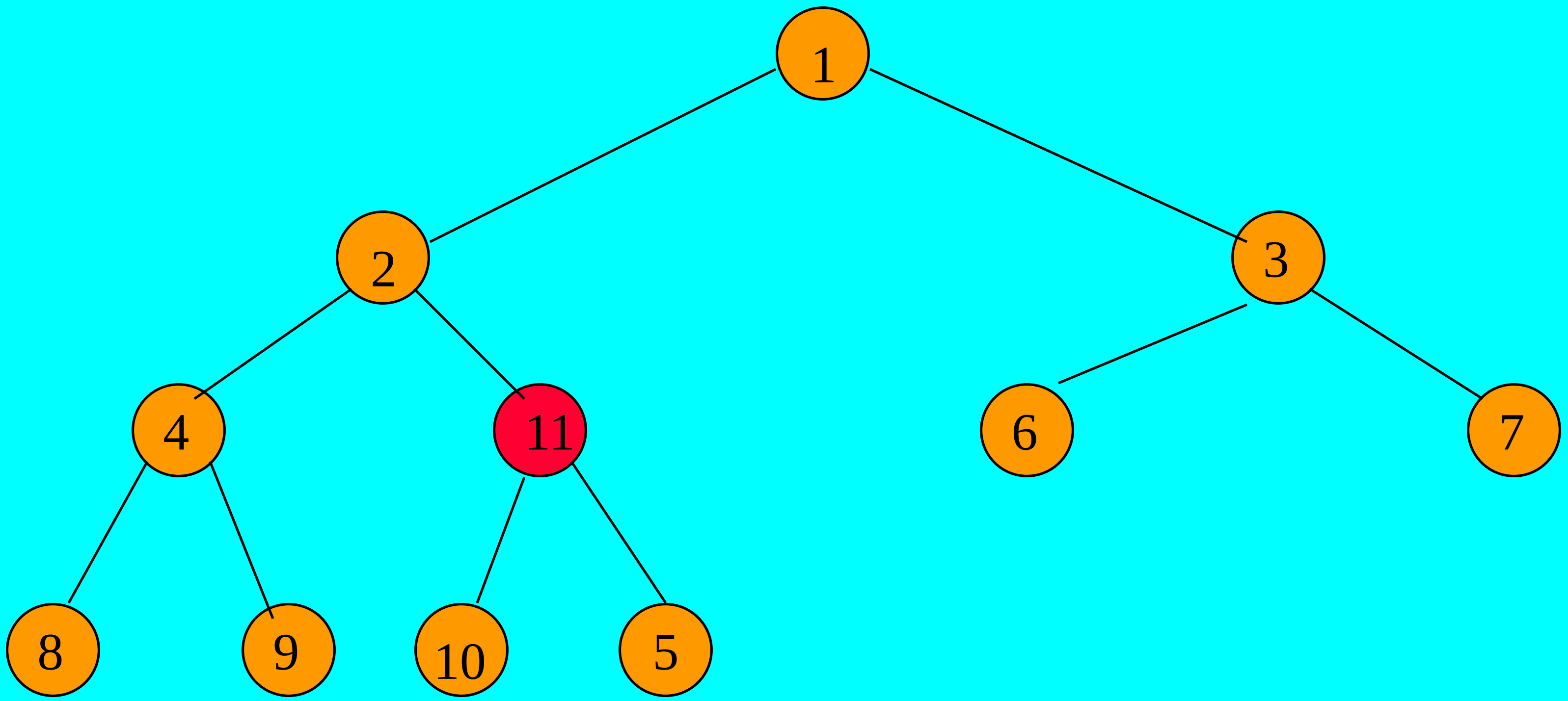
Initializing A Max Heap



Start at rightmost node that has a child i.e. last parent.

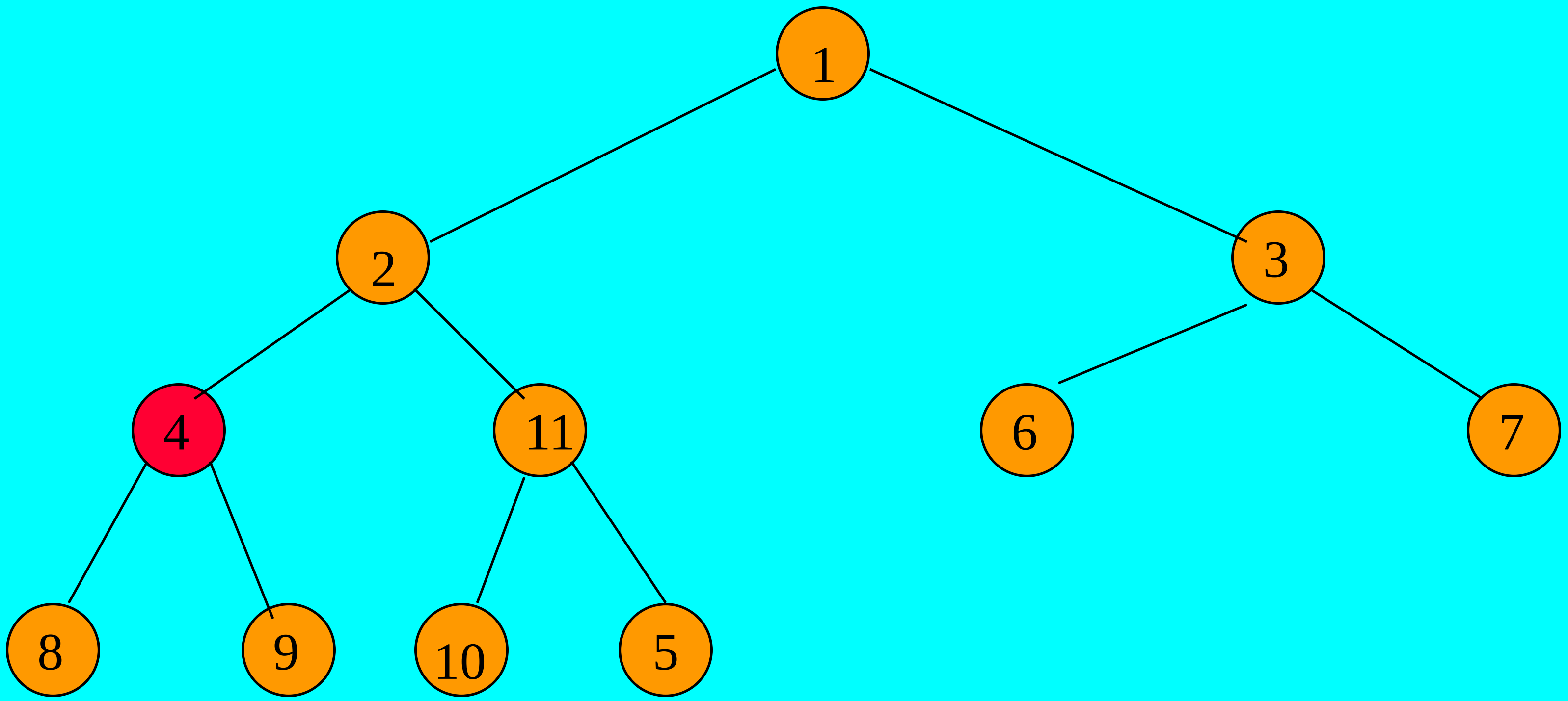
Index is $(\lfloor n/2 \rfloor - 1)$.

Initializing A Max Heap

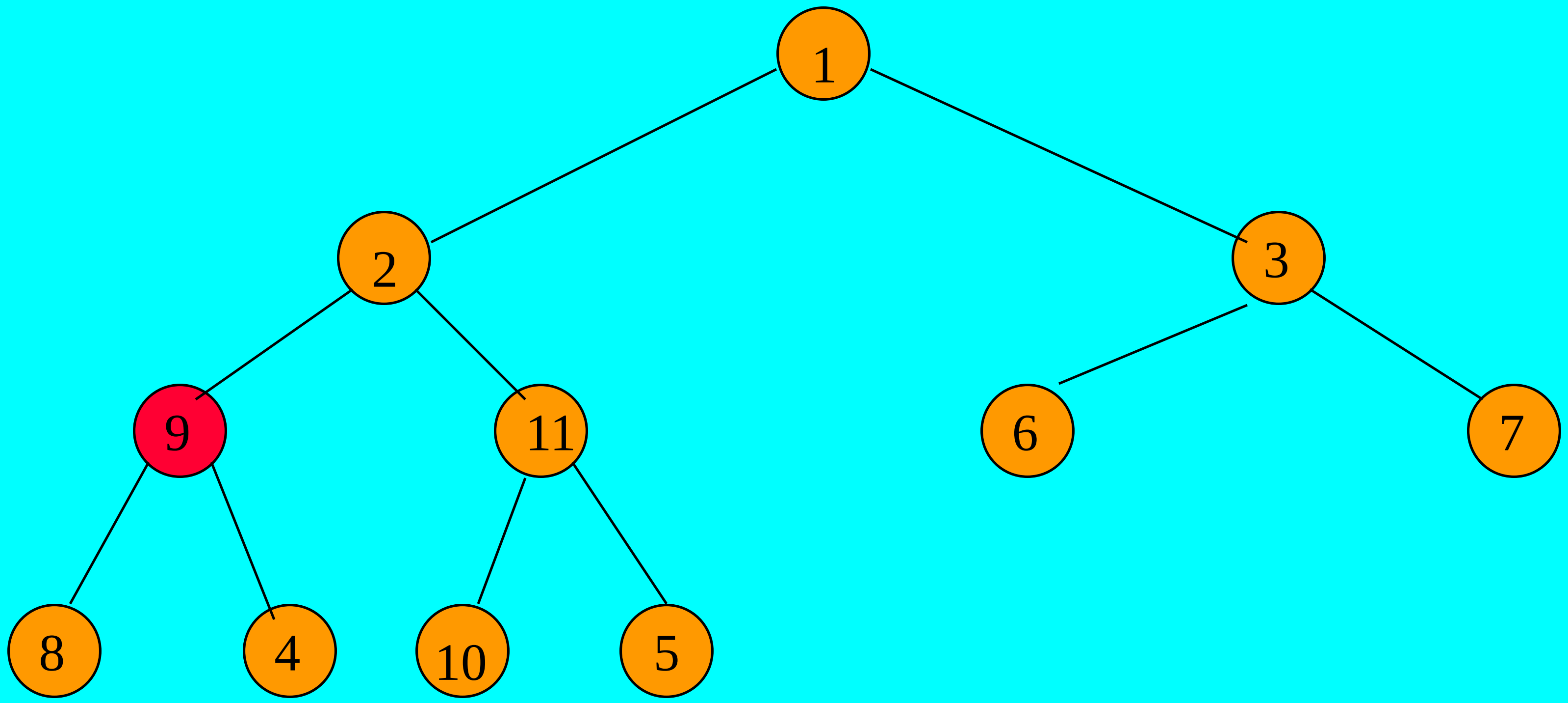


Move to next lower array position. Repeat it up to root.

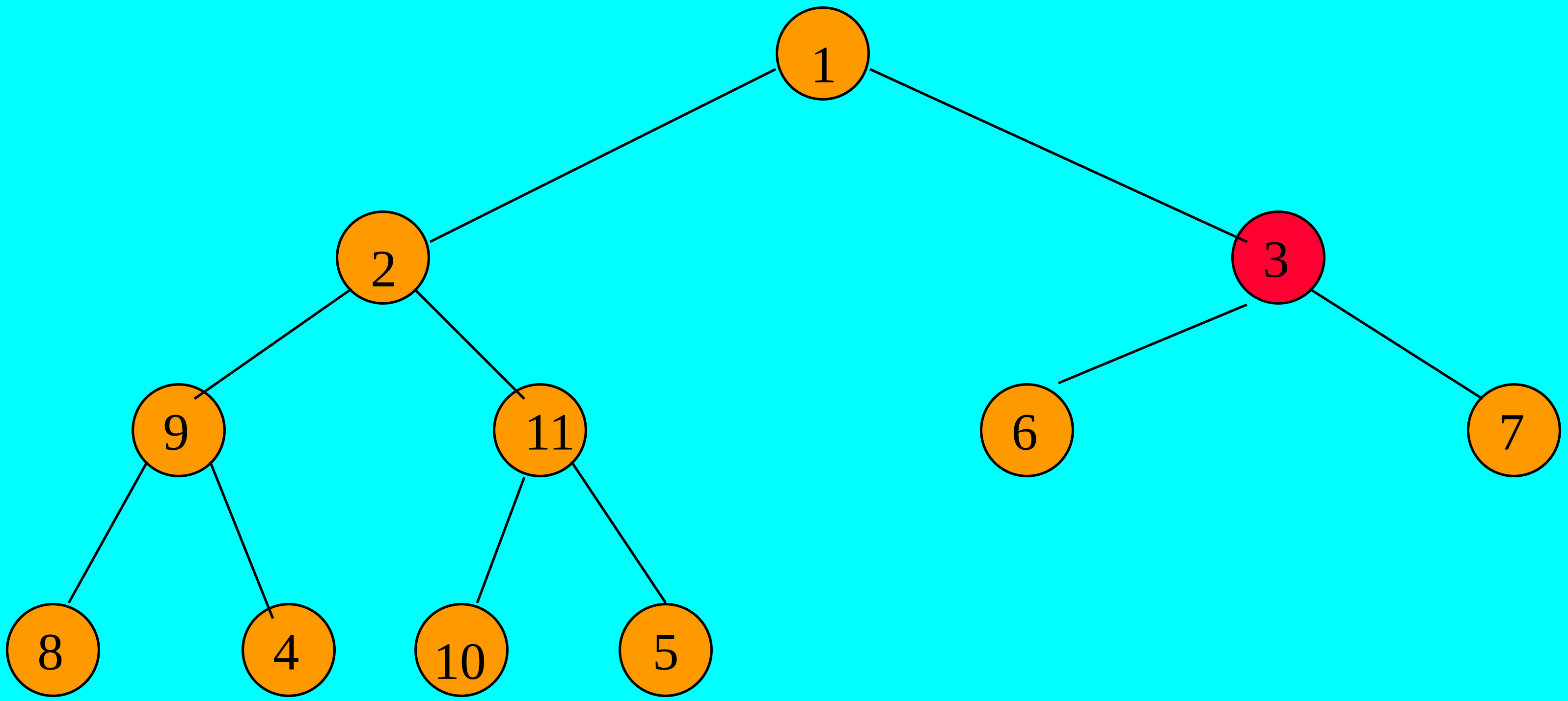
Initializing A Max Heap



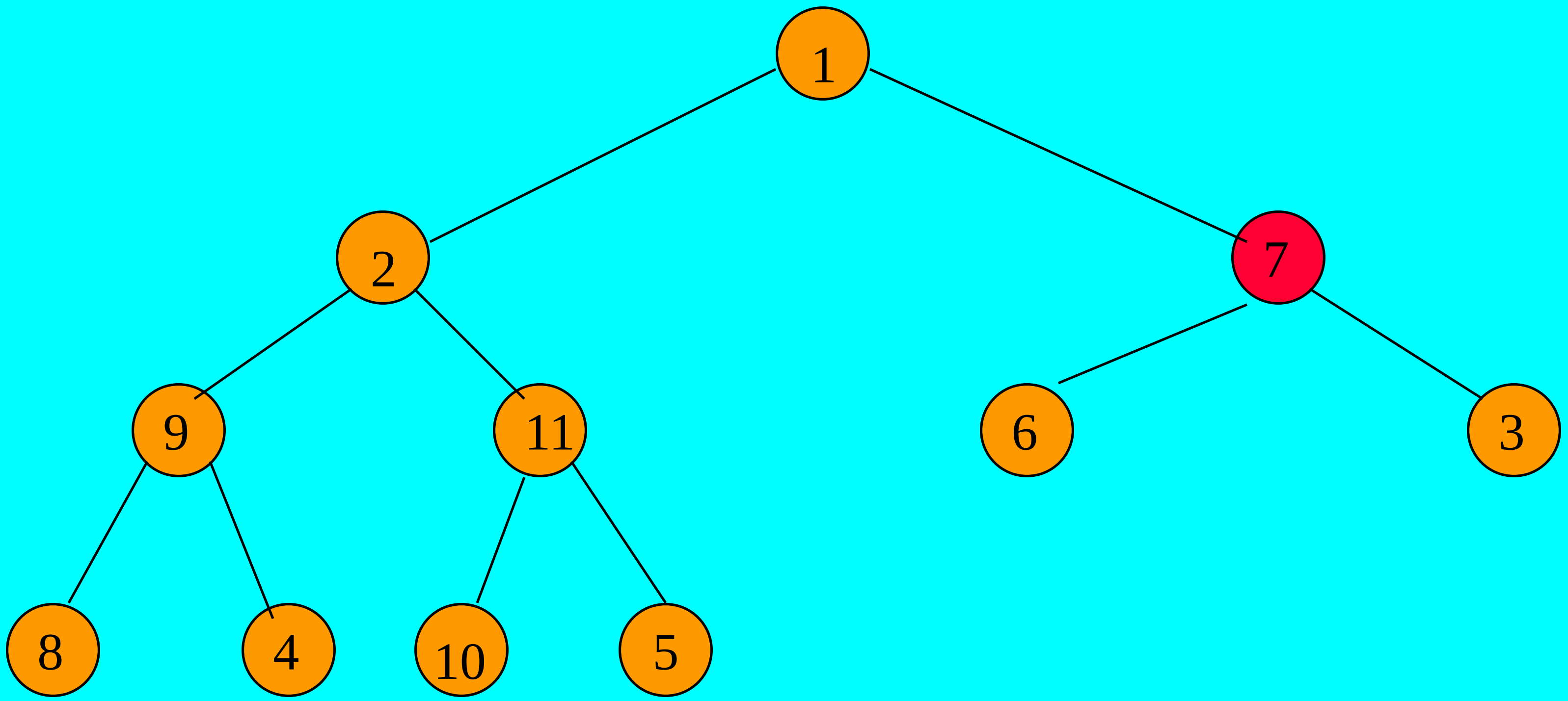
Initializing A Max Heap



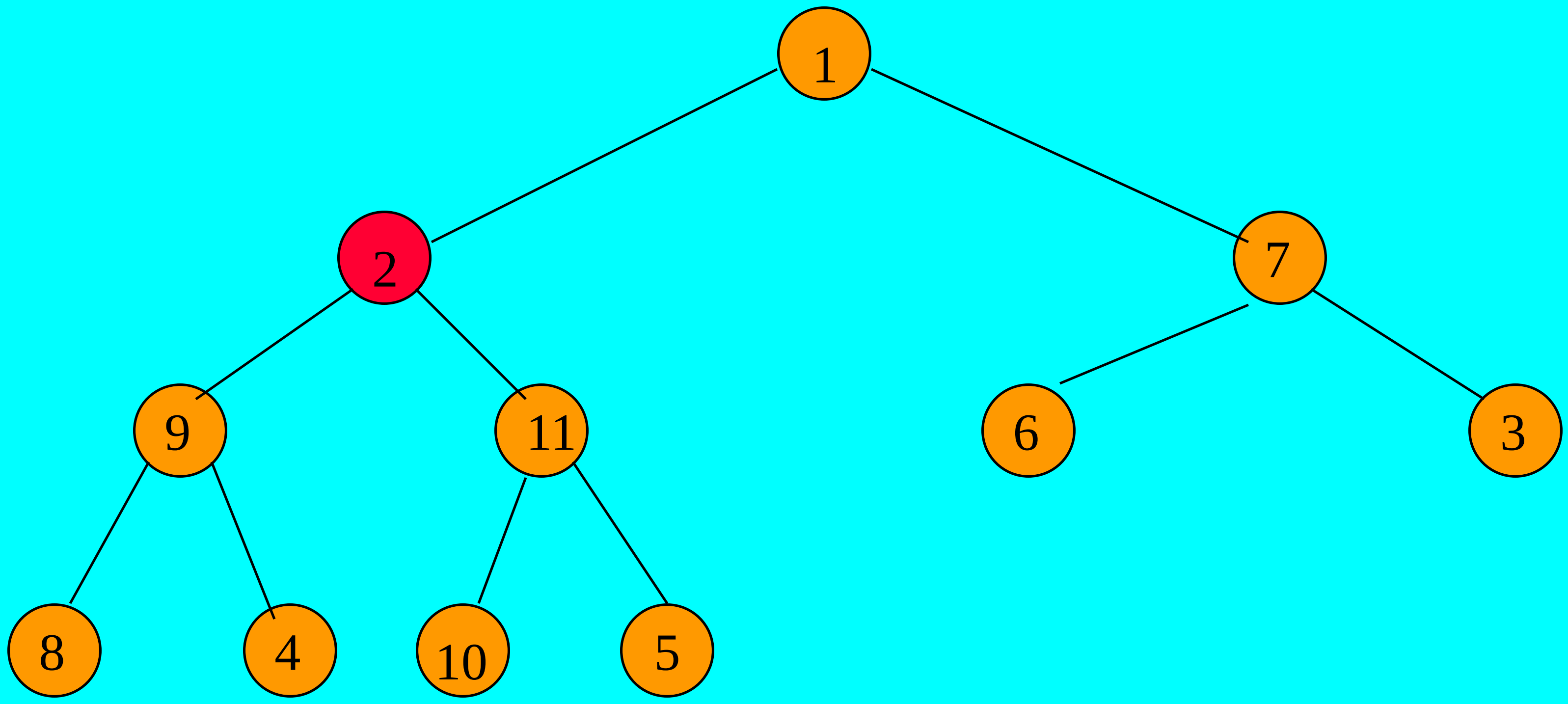
Initializing A Max Heap



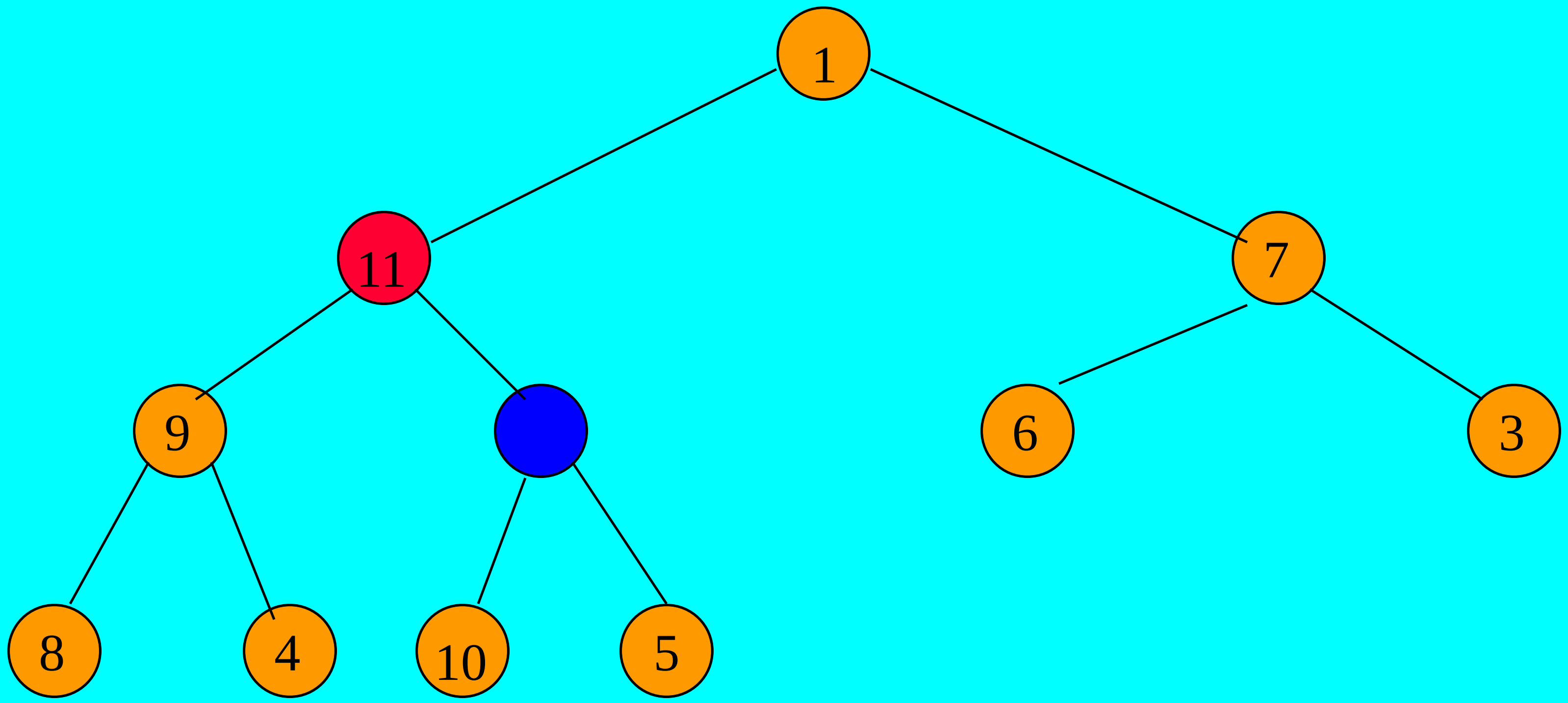
Initializing A Max Heap



Initializing A Max Heap

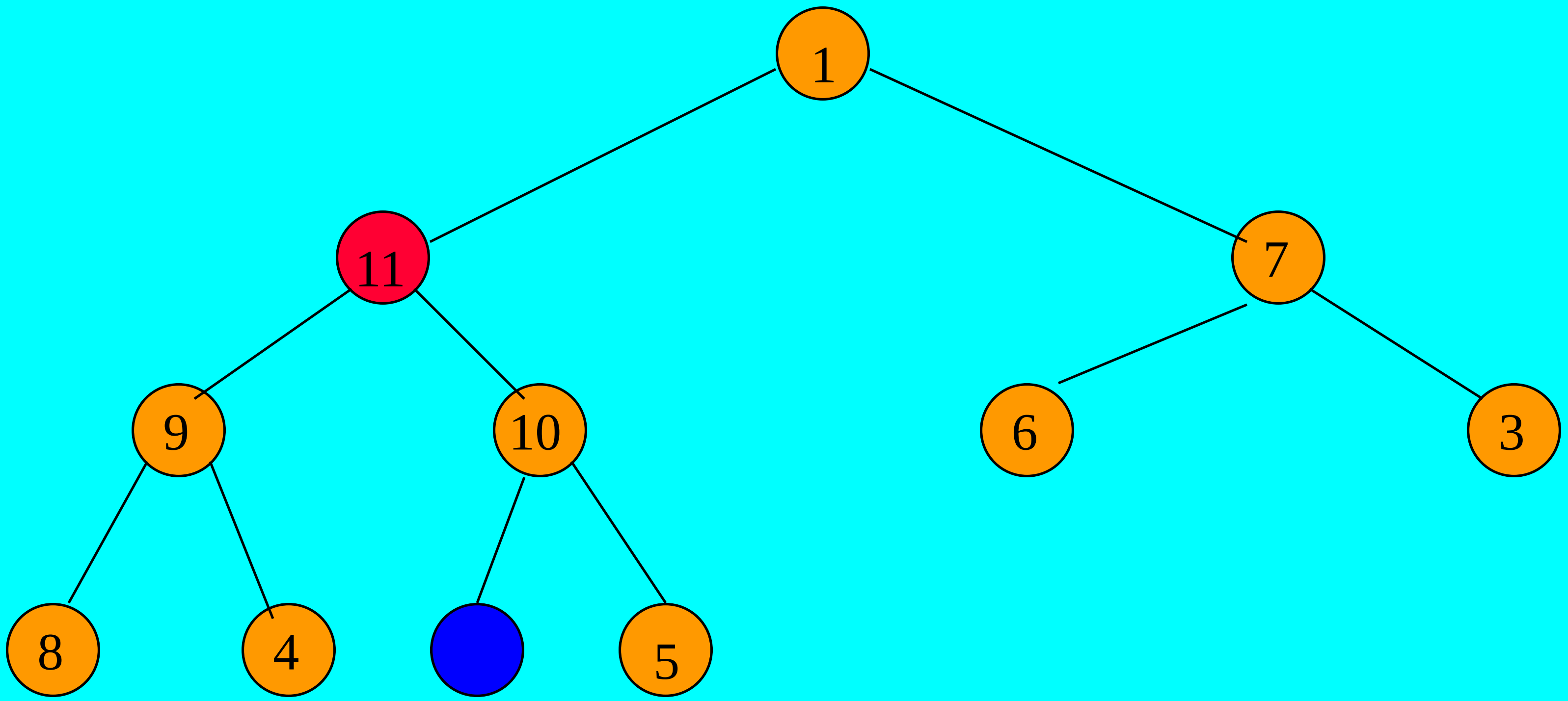


Initializing A Max Heap



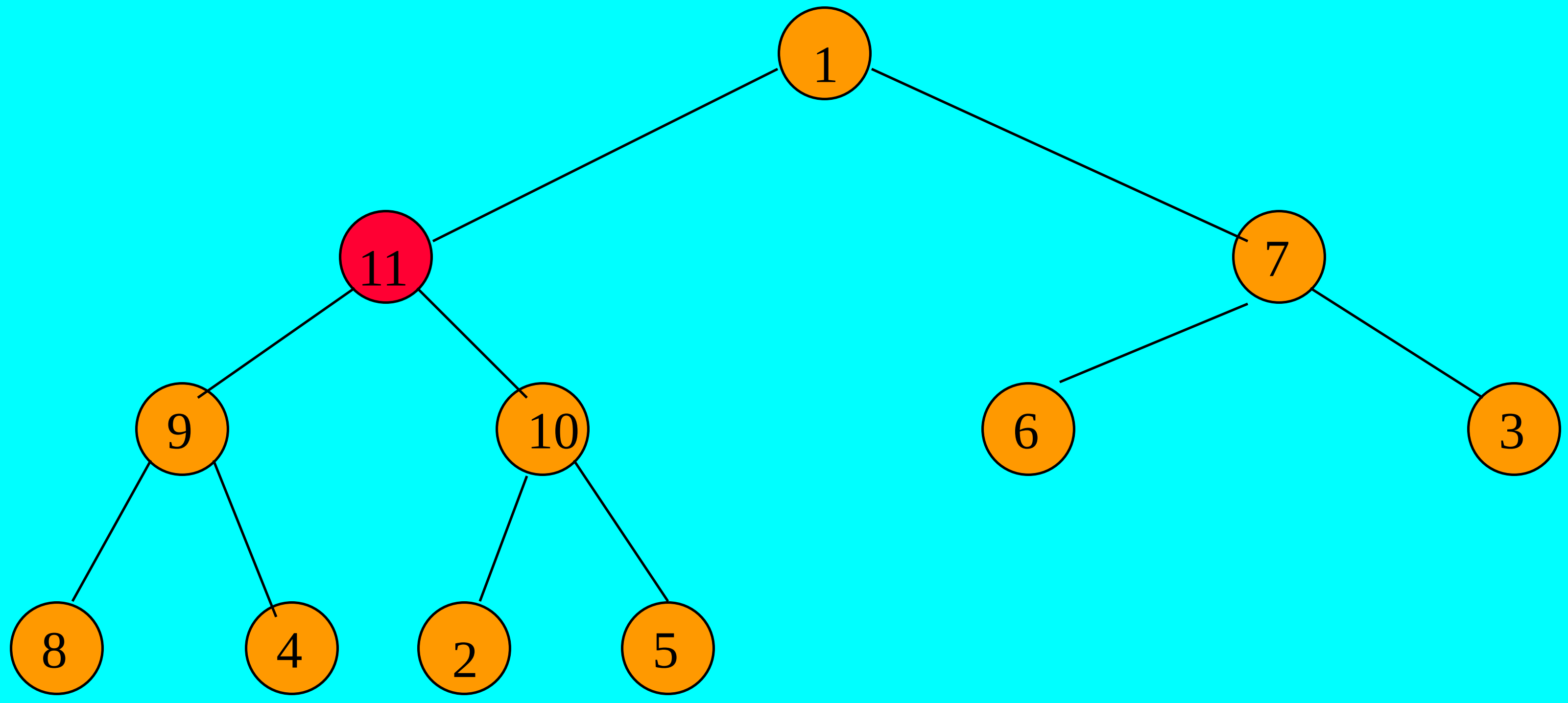
Find a home for 2.

Initializing A Max Heap



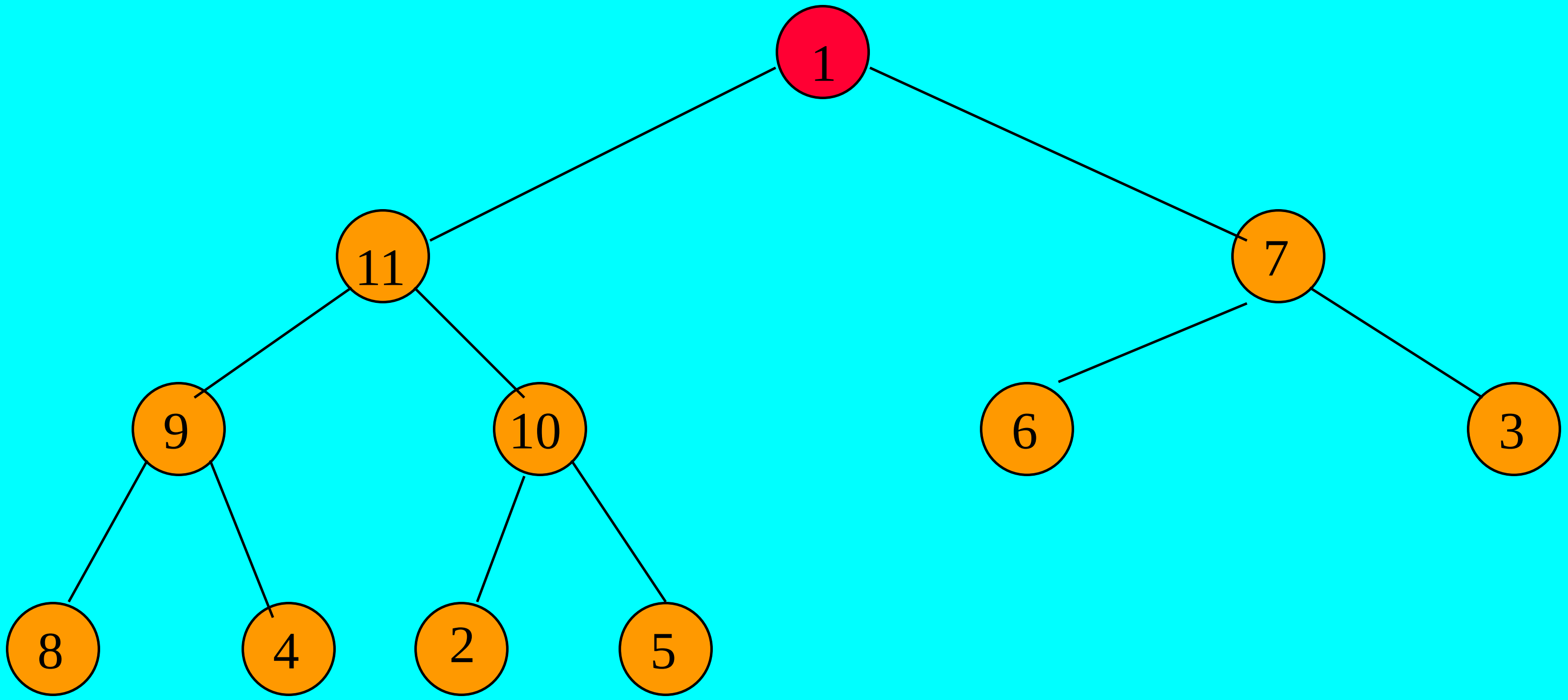
Find a home for 2.

Initializing A Max Heap



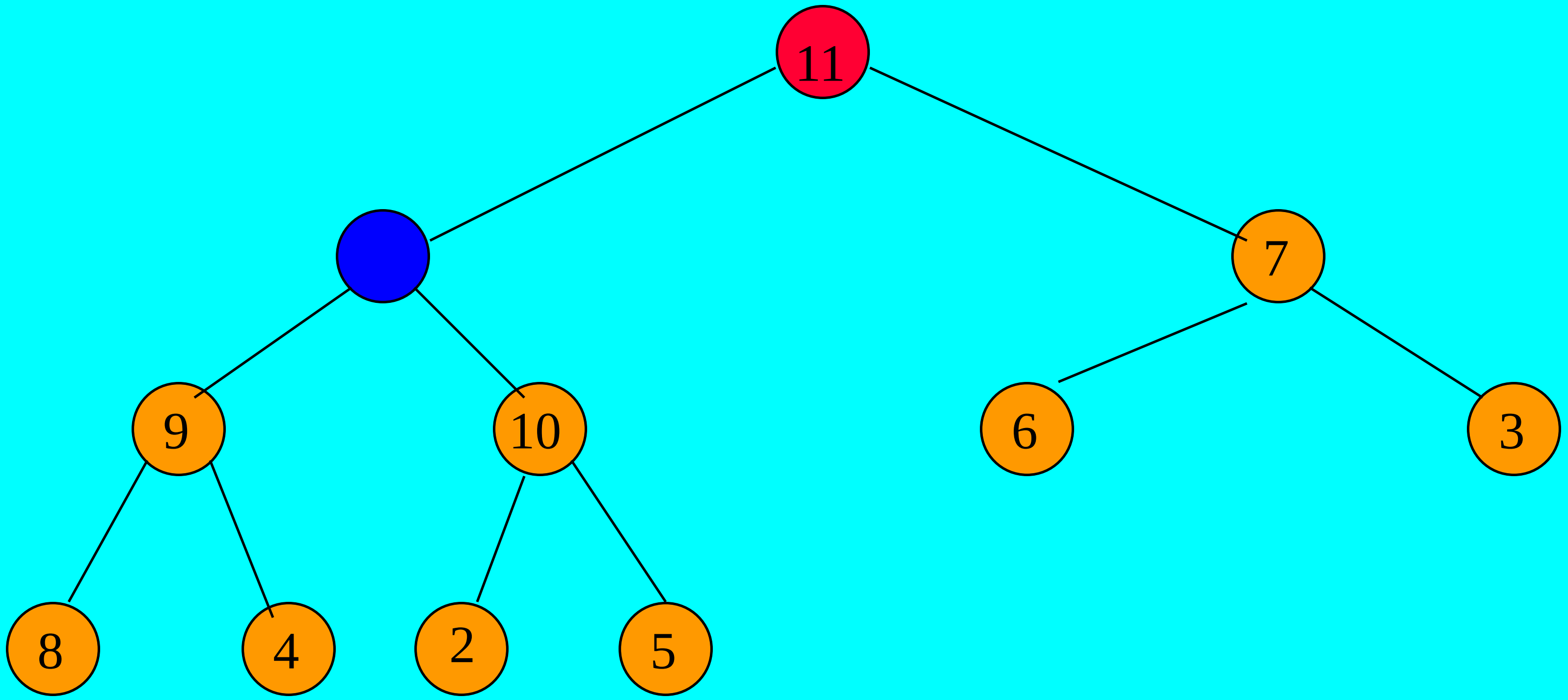
Done, move to next lower array position.

Initializing A Max Heap



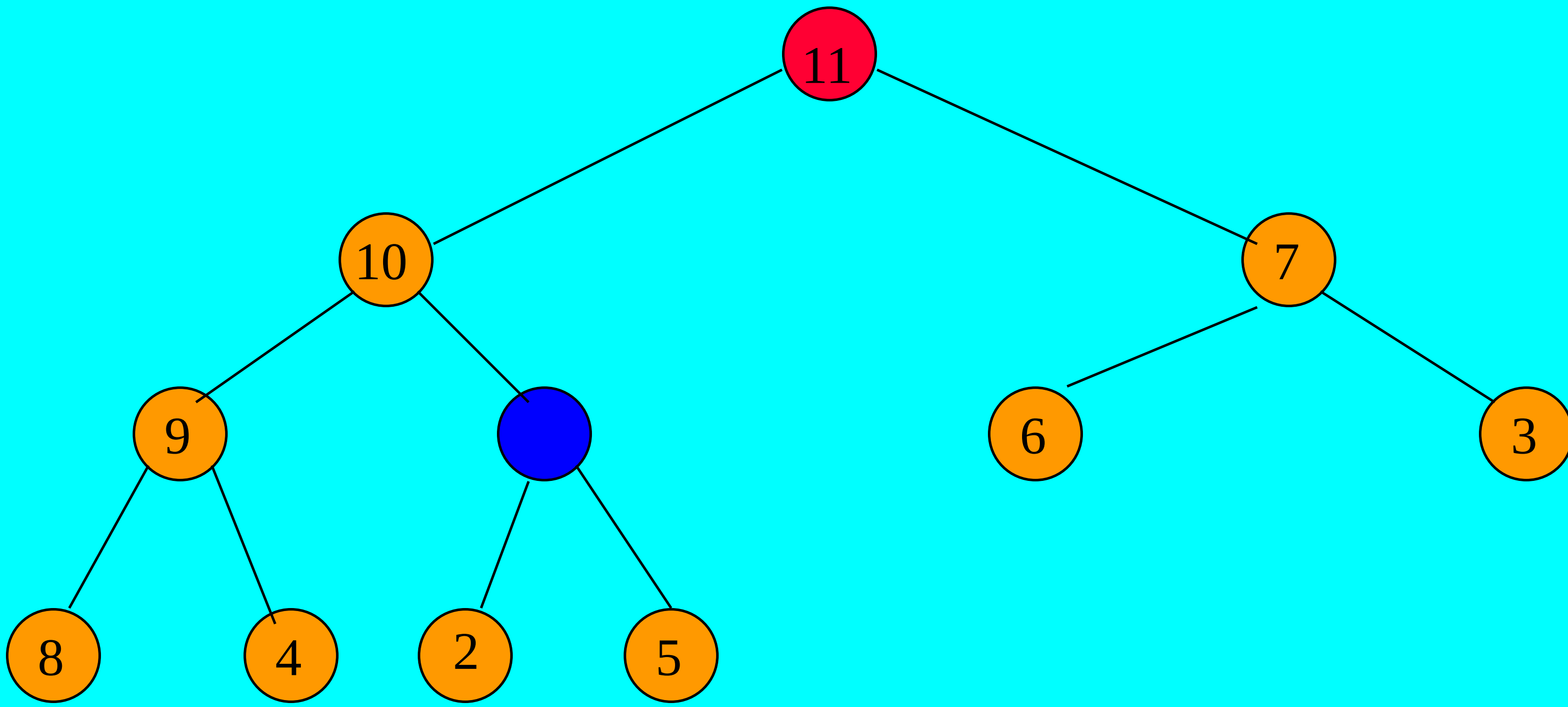
Find home for 1.

Initializing A Max Heap



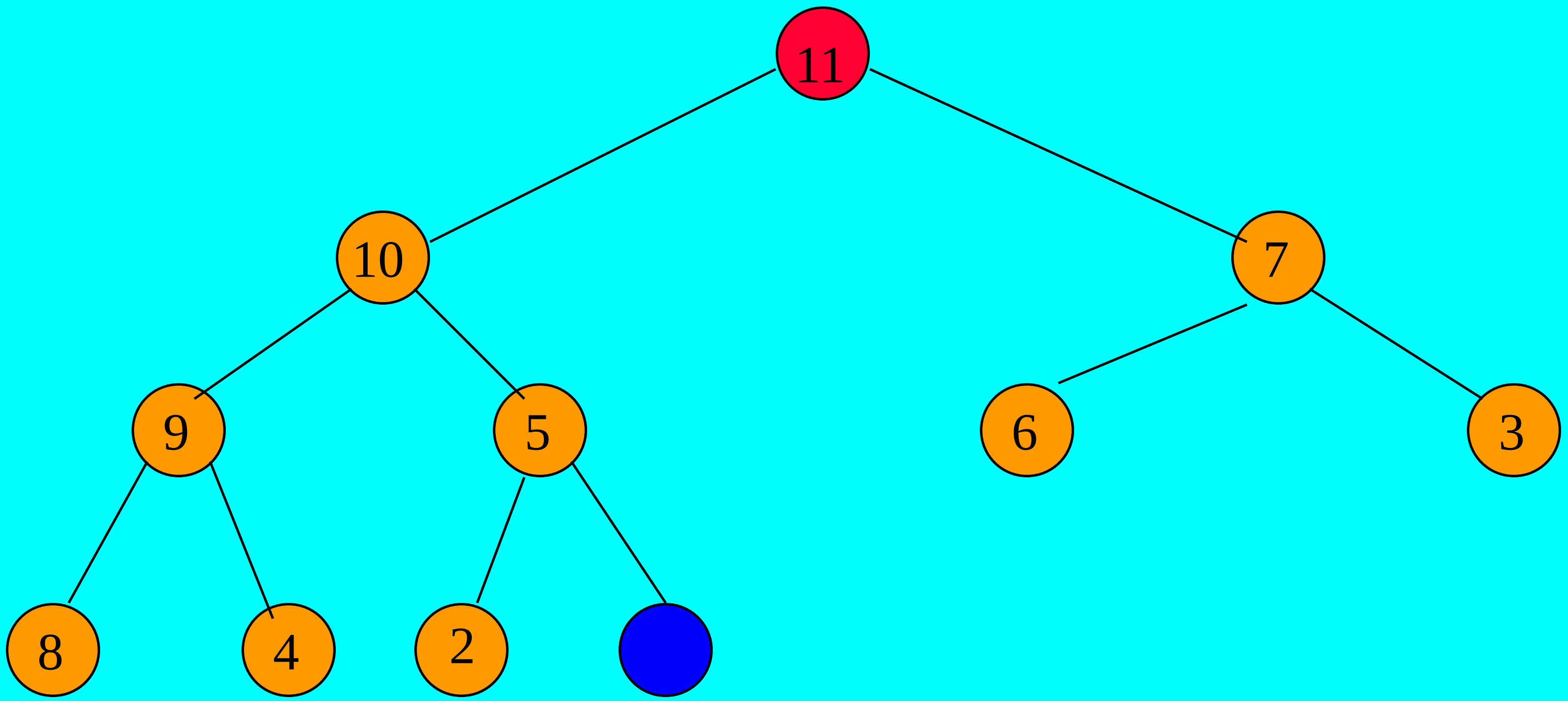
Find home for 1.

Initializing A Max Heap



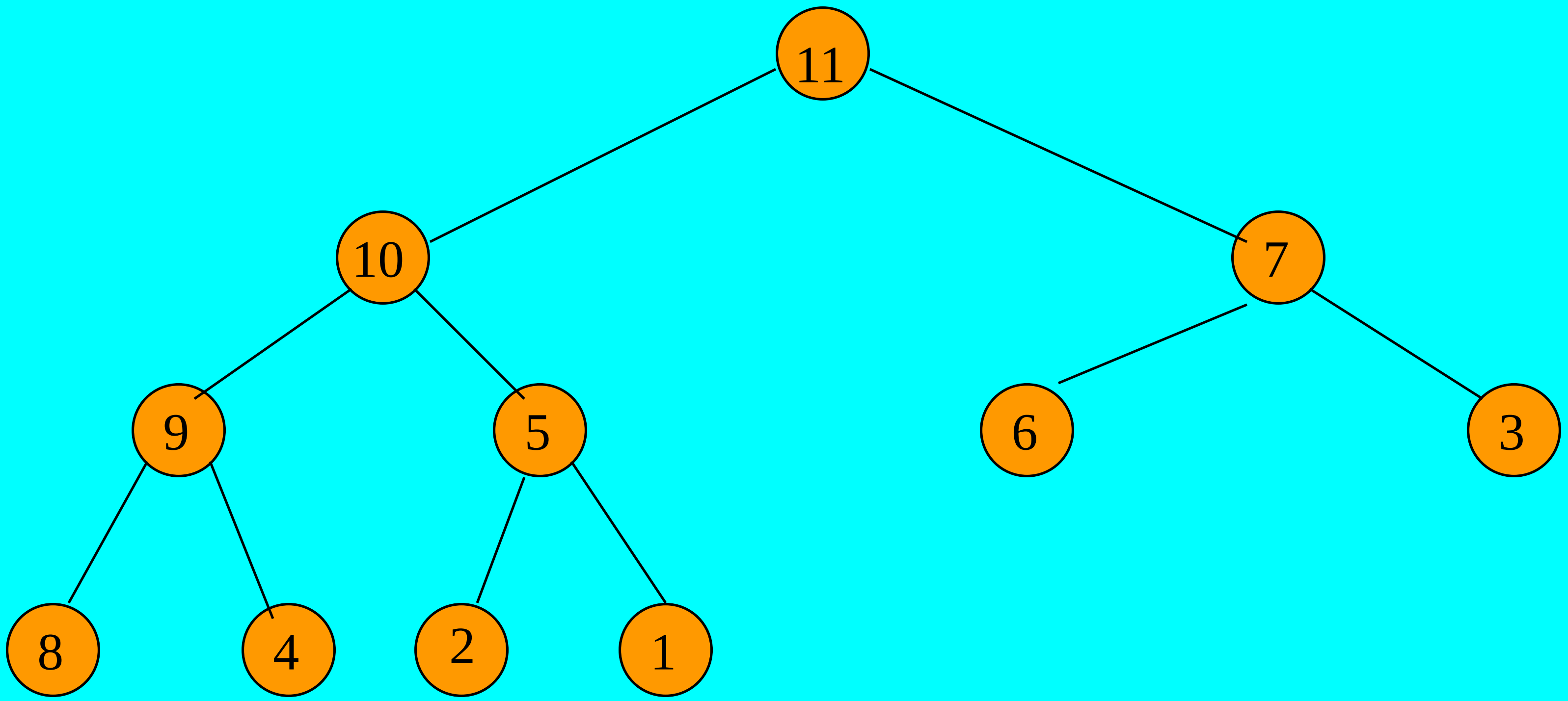
Find home for 1.

Initializing A Max Heap



Find home for 1.

Initializing A Max Heap

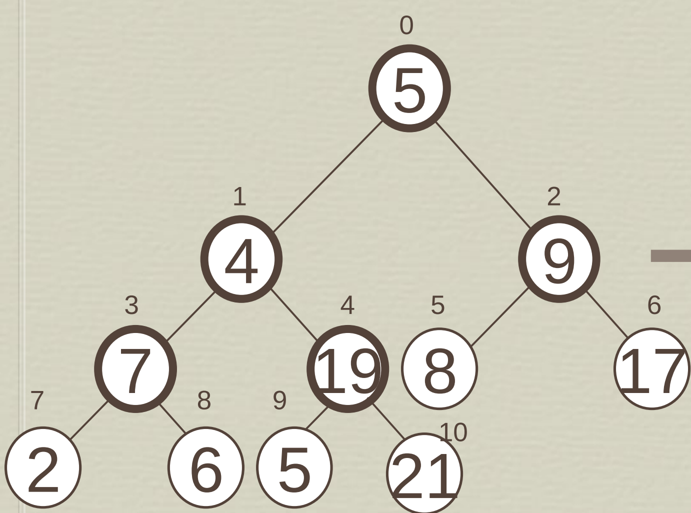


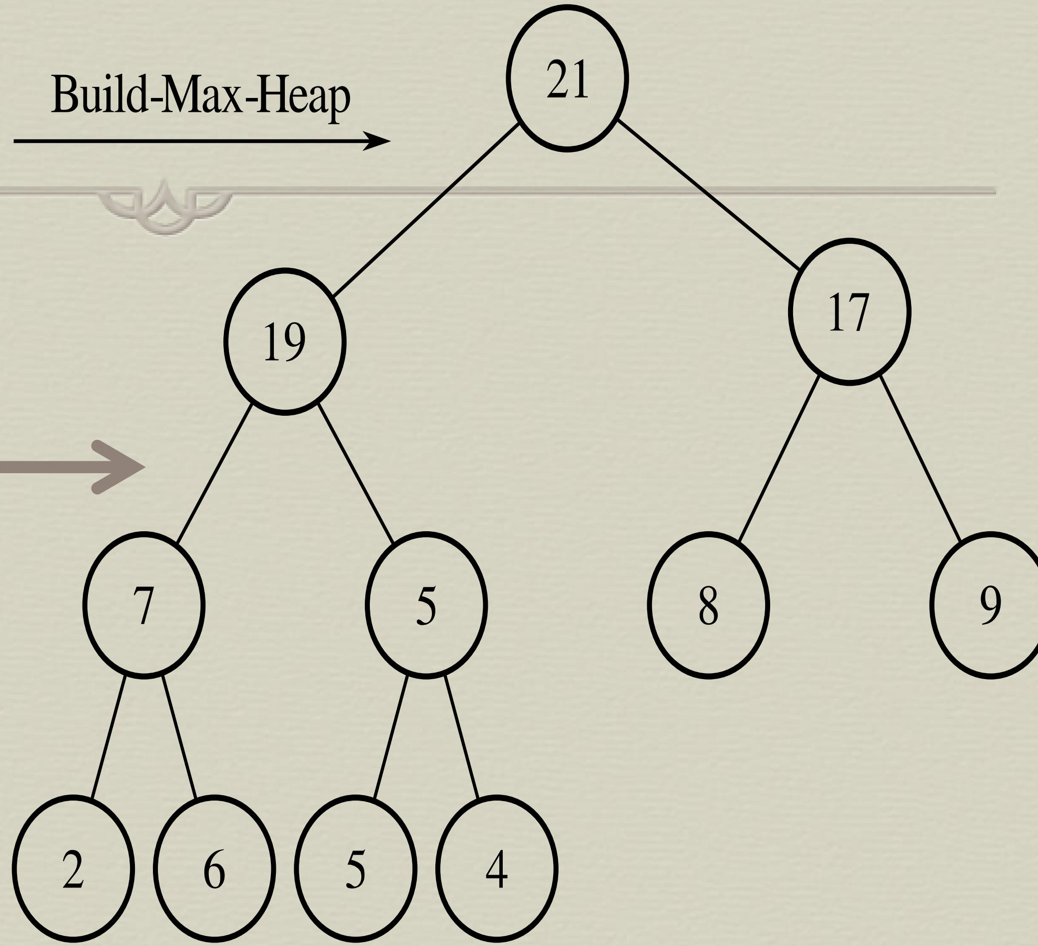
Done.

Exercise: Arrange this array as a heap tree/Array?

Array

5	4	9	7	19	8	17	2	6	5	21

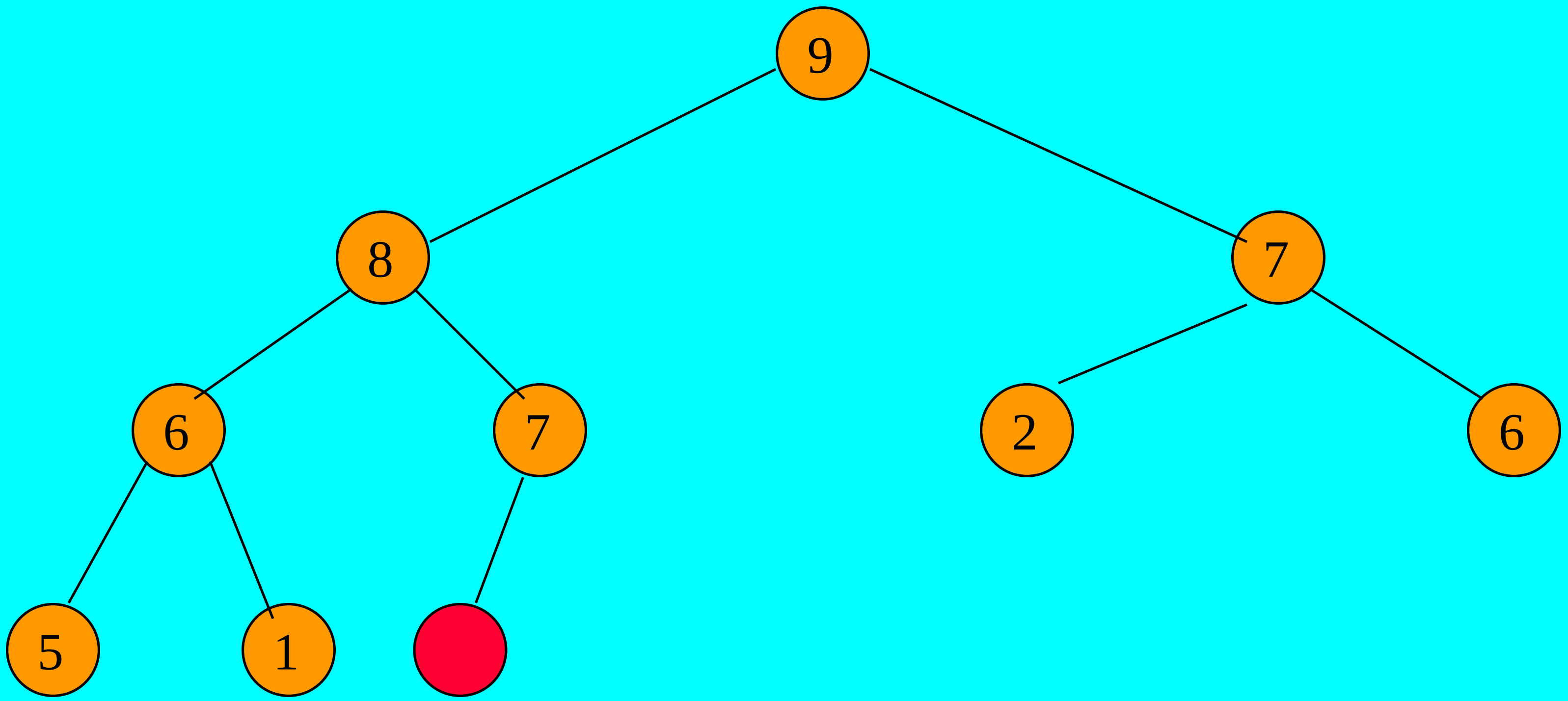




Heap array

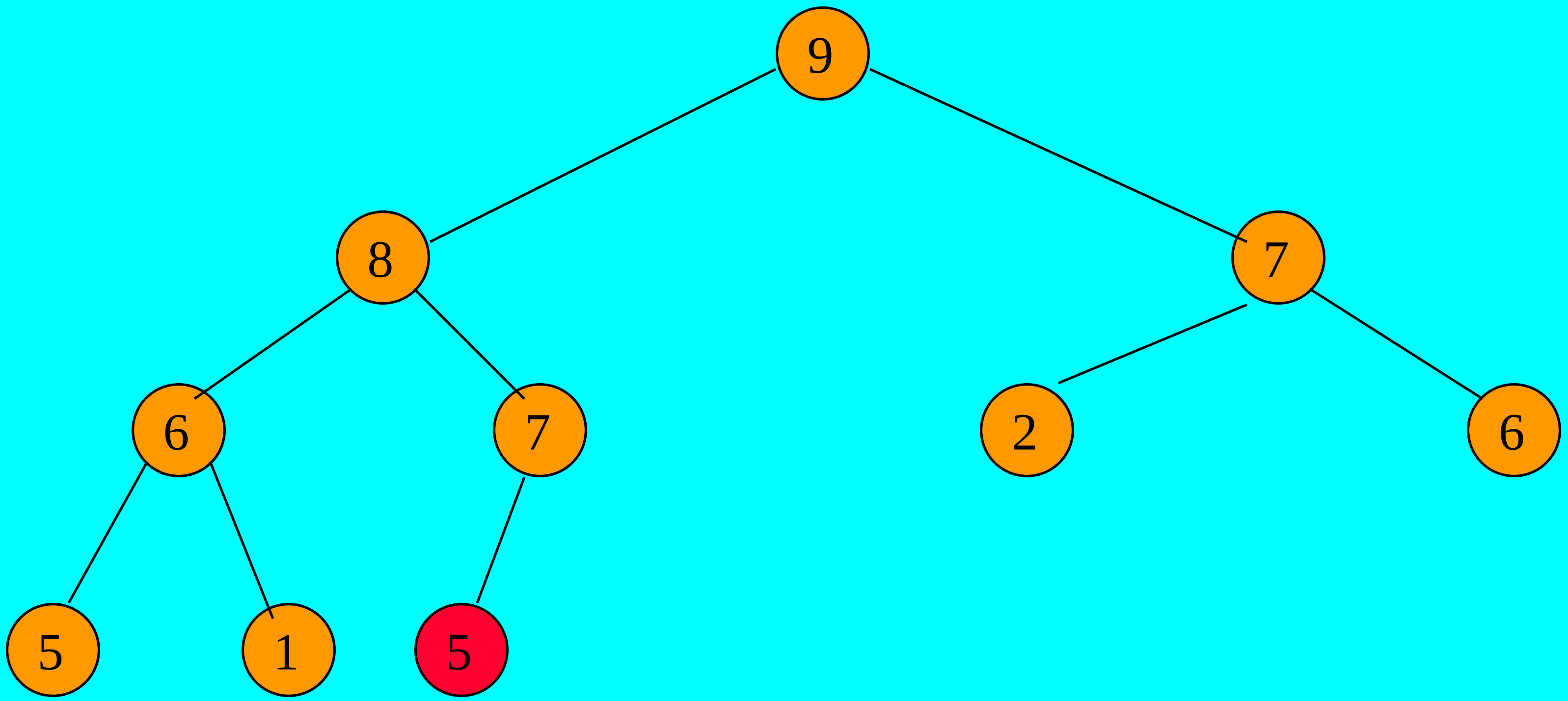
21	19	17	7	5	8	9	2	6	5	4
0	1	2	3	4	5	6	7	8	9	10

Putting An Element Into A Max Heap



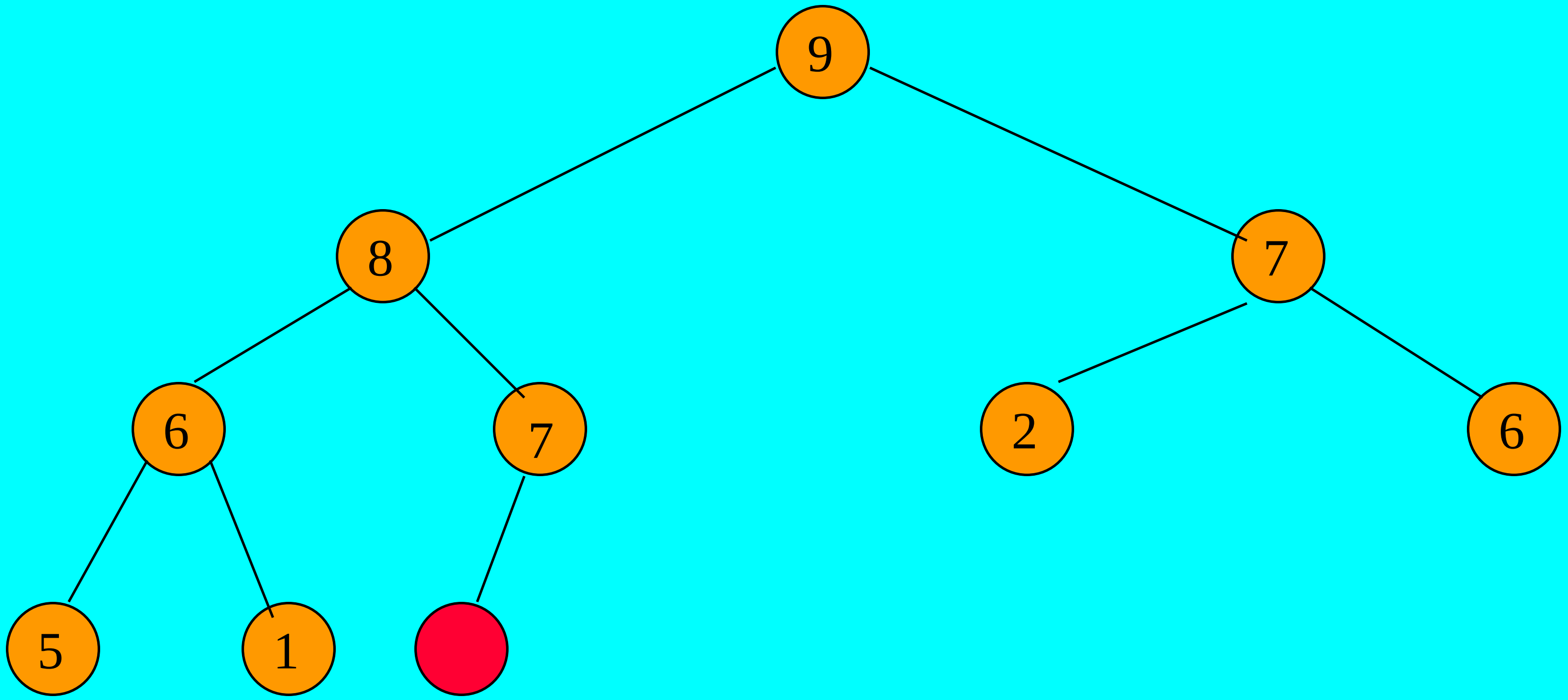
Complete binary tree with 10 nodes.

Putting An Element Into A Max Heap



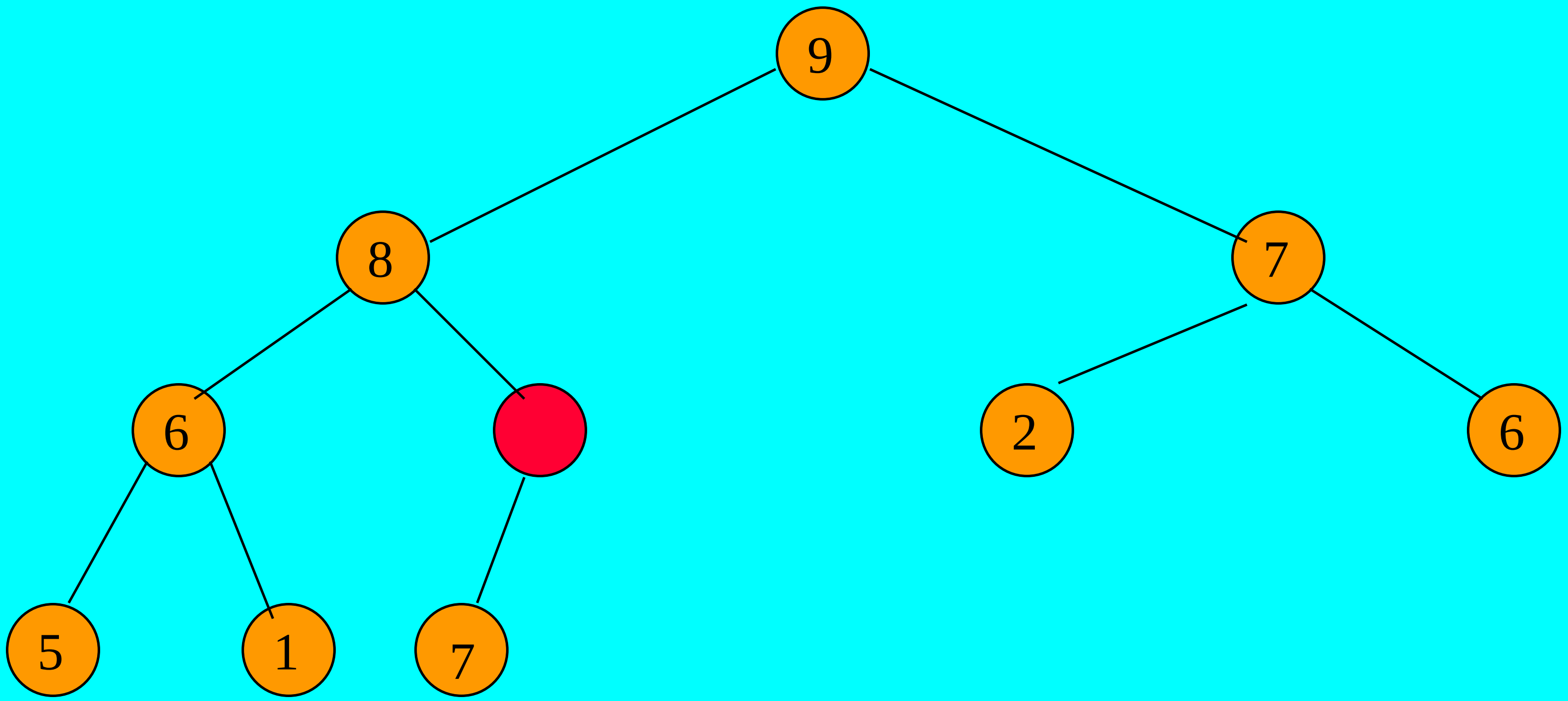
New element is 5.

Putting An Element Into A Max Heap



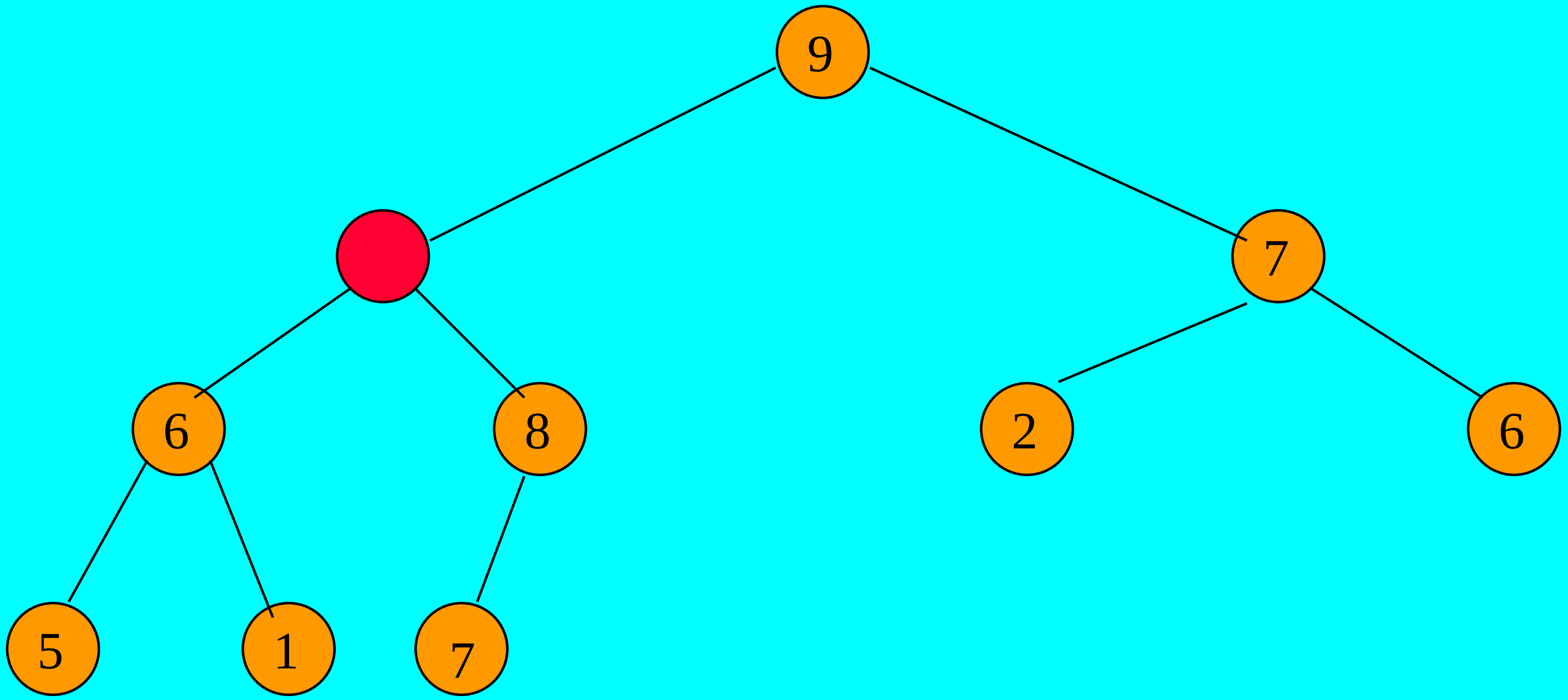
If the new element is 20 rather than 5.

Putting An Element Into A Max Heap



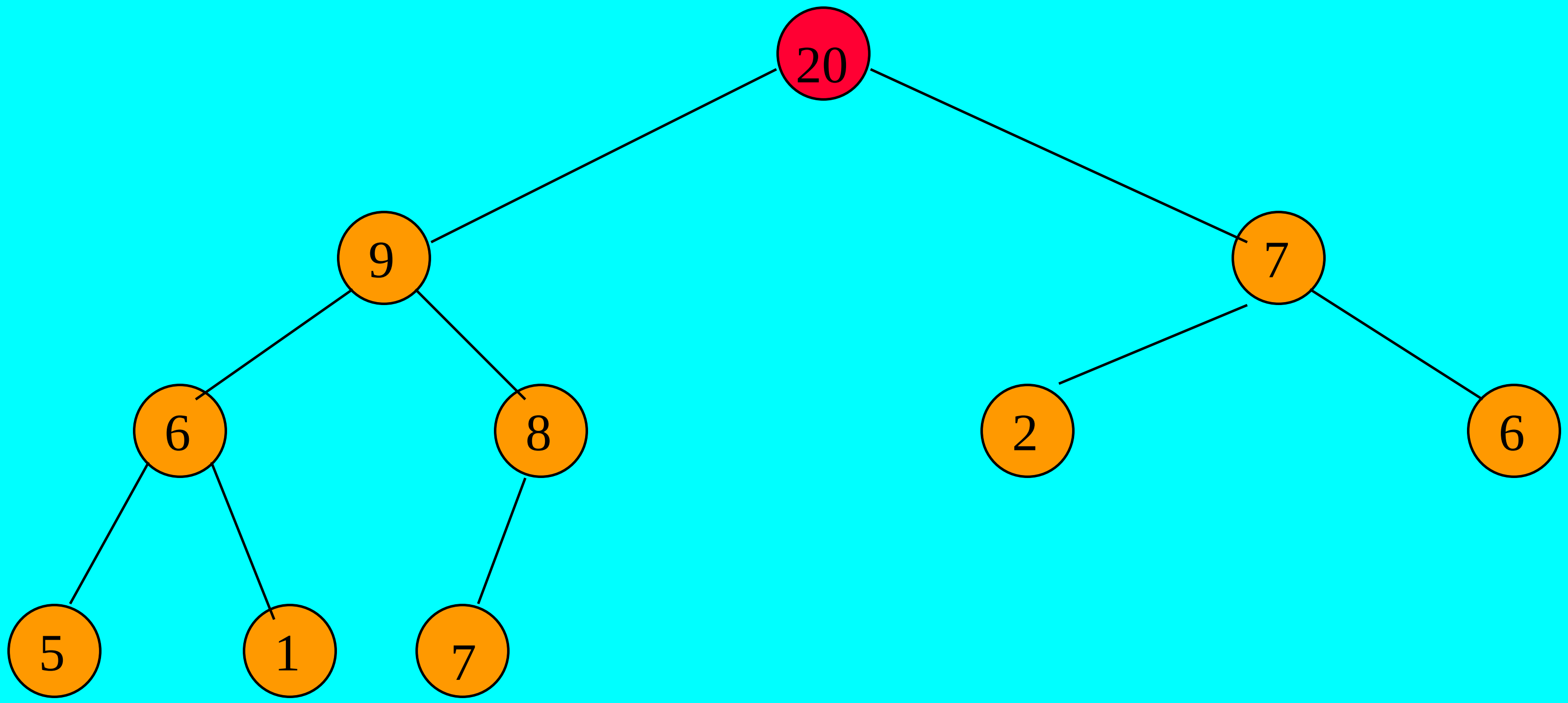
New element is 20.

Putting An Element Into A Max Heap



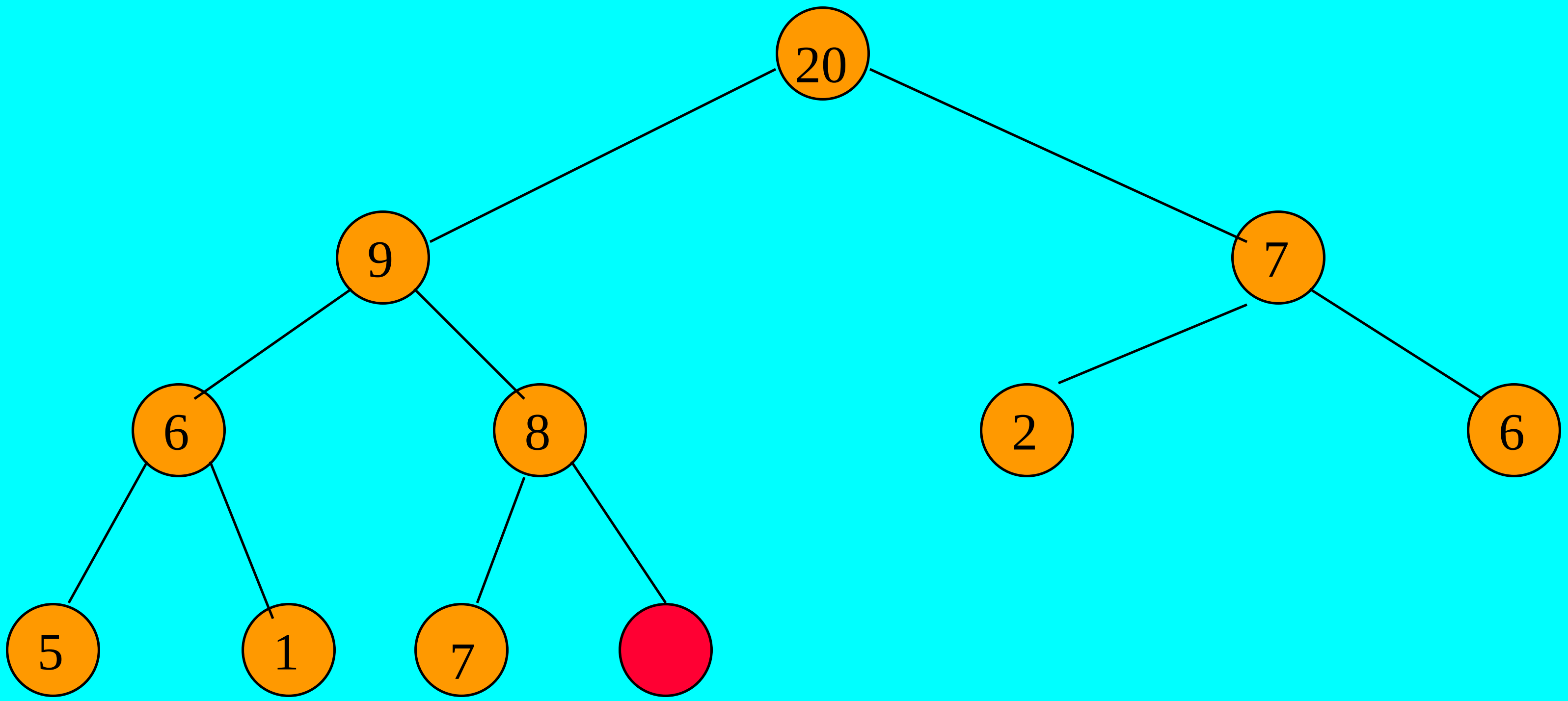
New element is 20.

Putting An Element Into A Max Heap



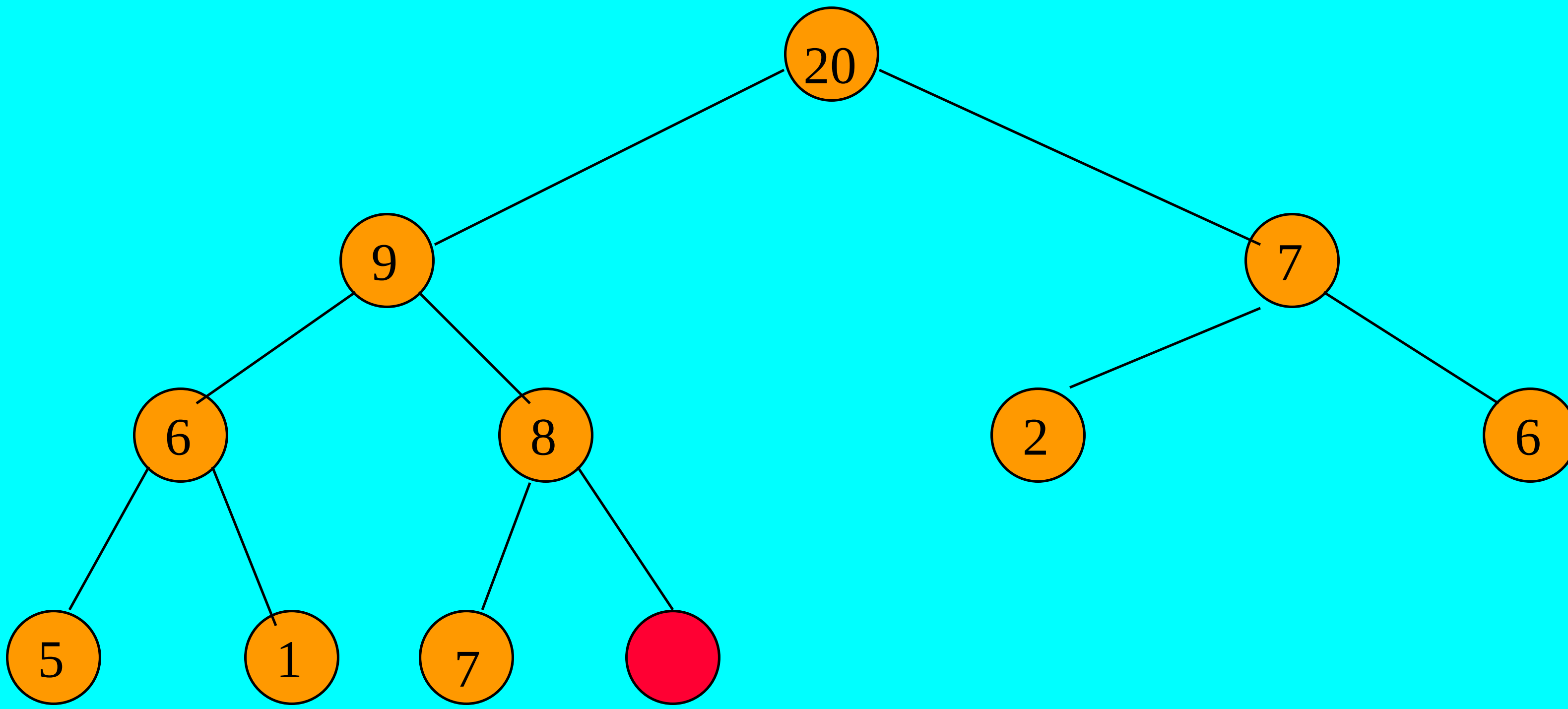
New element is 20.

Putting An Element Into A Max Heap



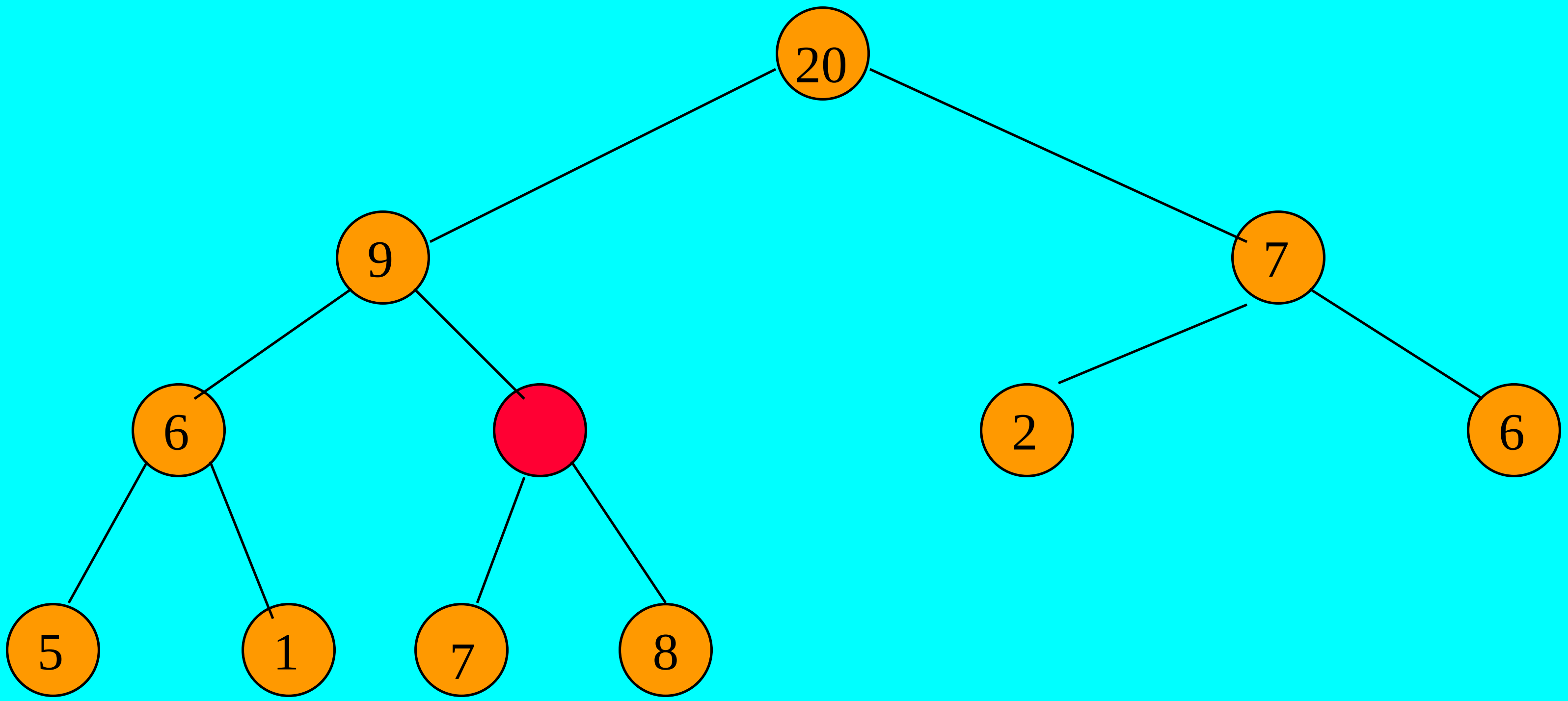
Complete binary tree with 11 nodes.

Putting An Element Into A Max Heap



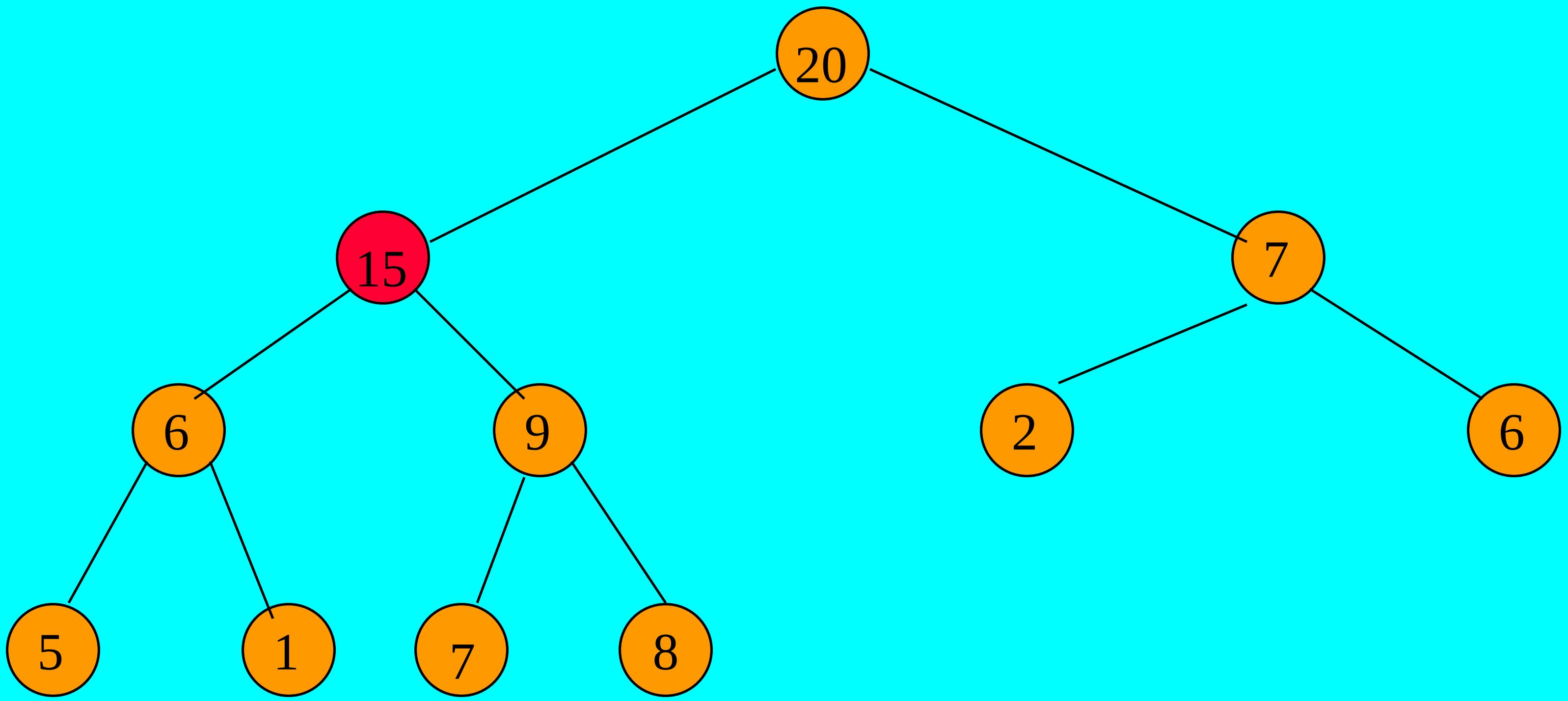
New element is 15.

Putting An Element Into A Max Heap



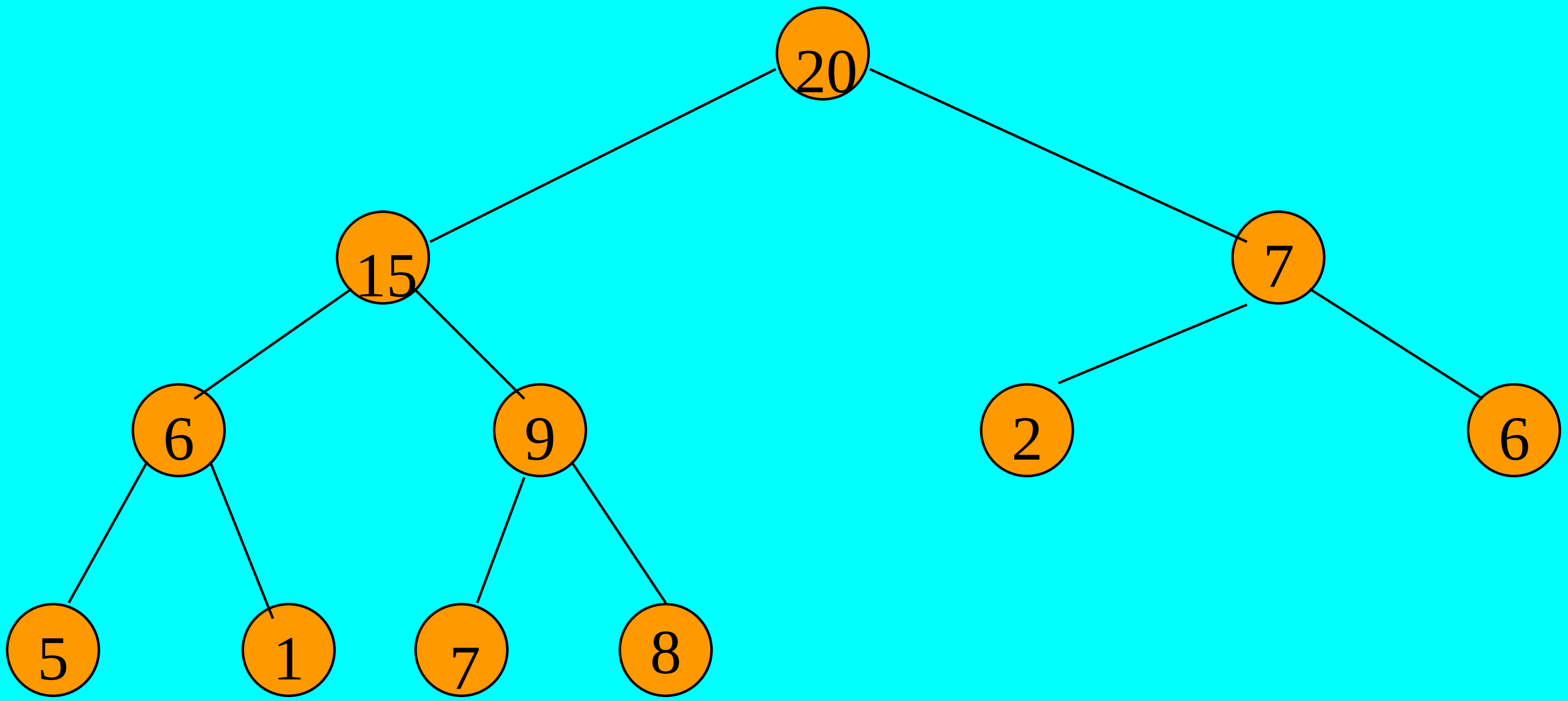
New element is 15.

Putting An Element Into A Max Heap



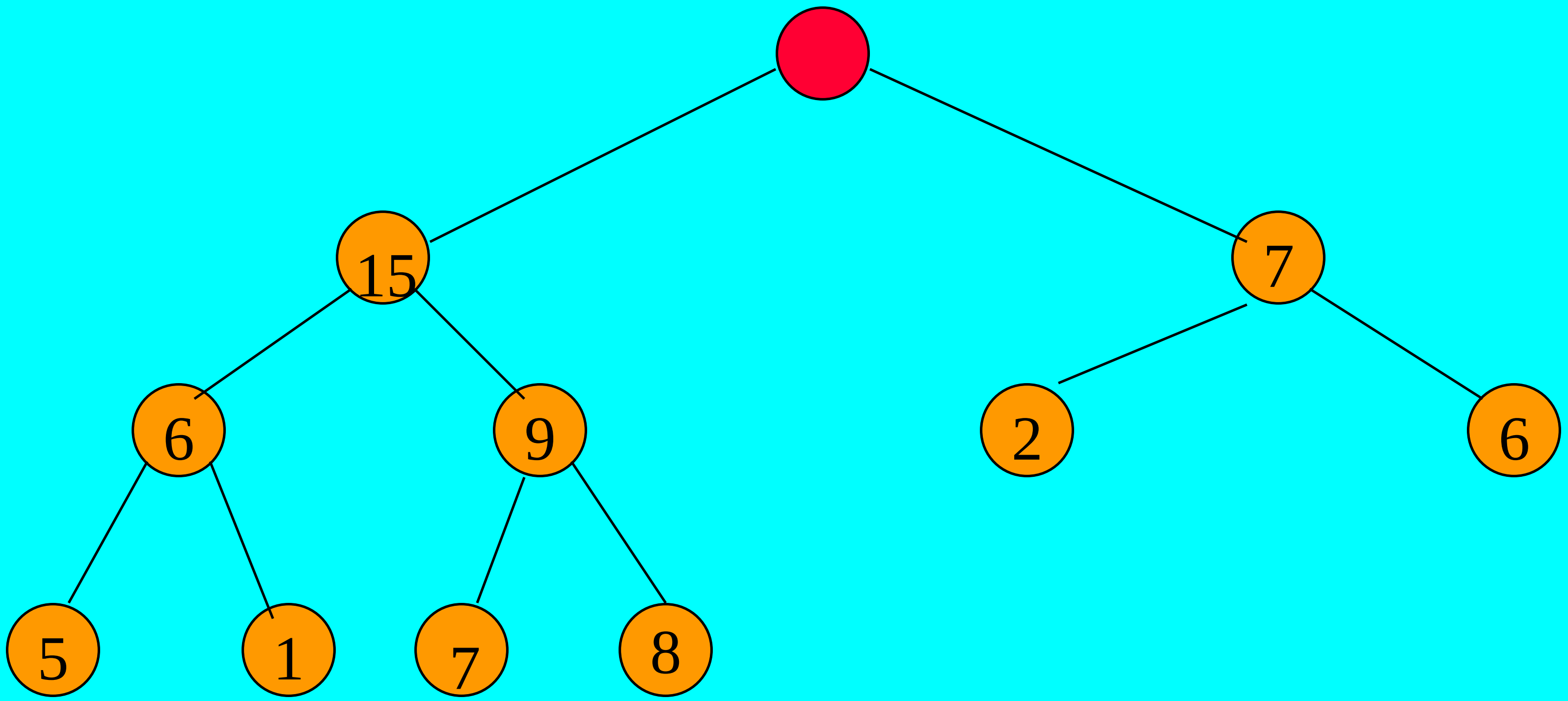
New element is 15.

Removing The Max Element



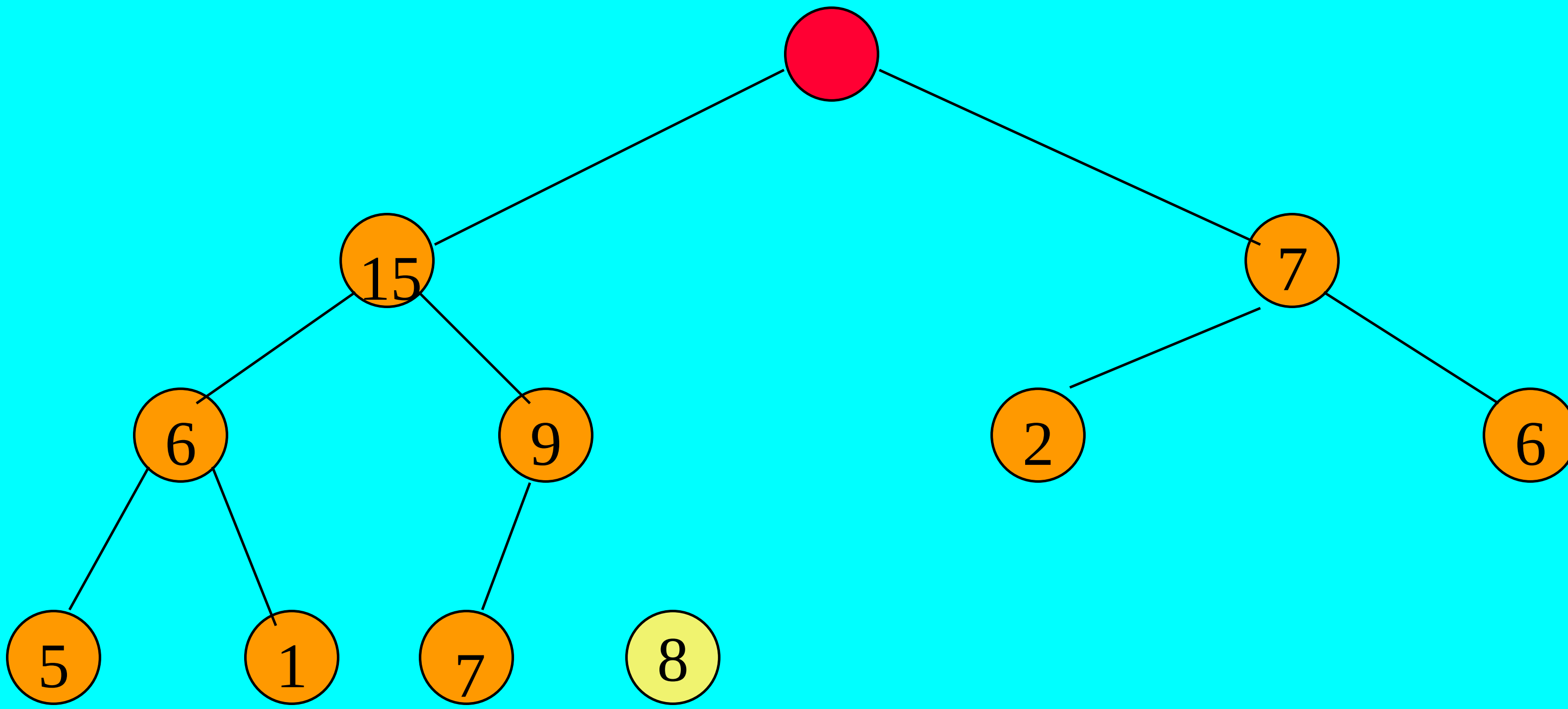
Max element is in the root.

Removing The Max Element



After max element is removed.

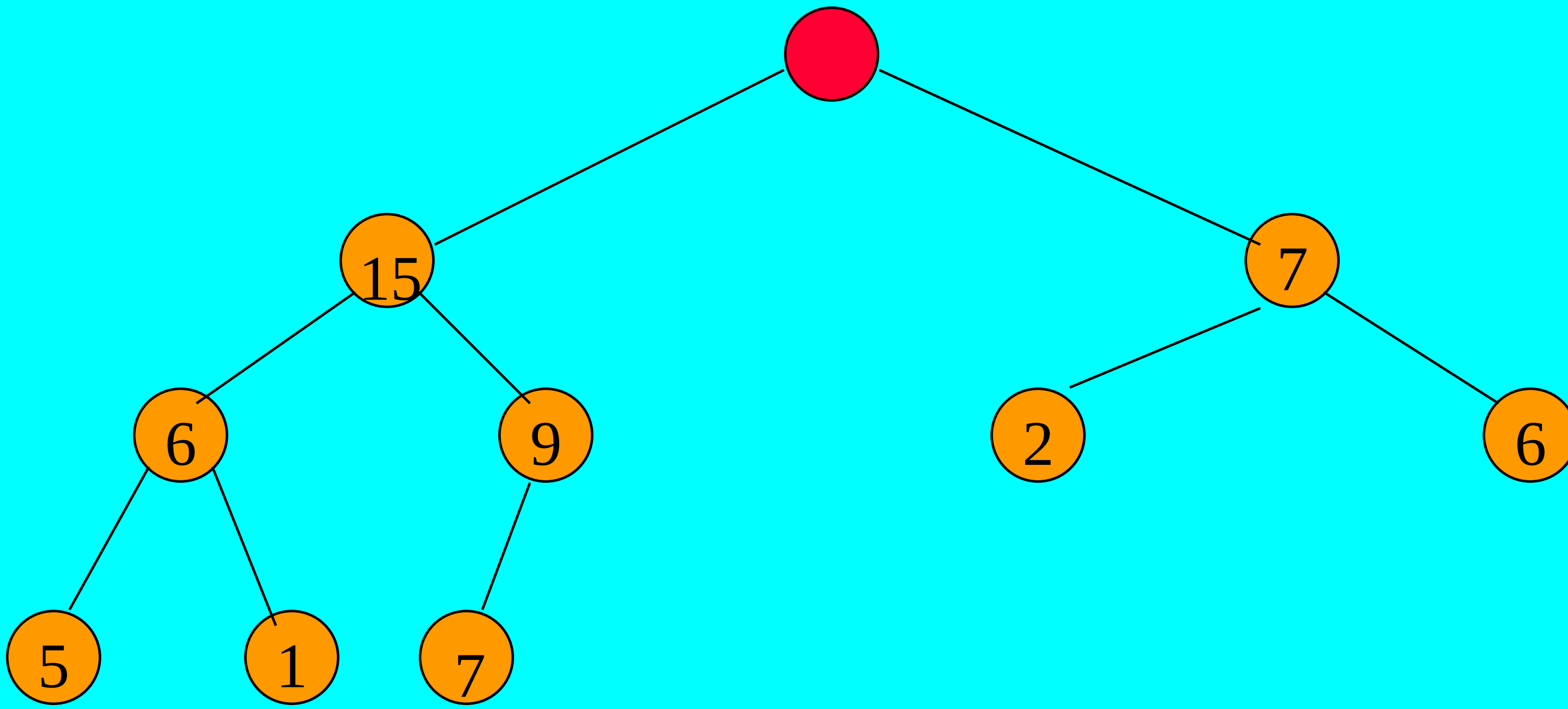
Removing The Max Element



Heap with 10 nodes.

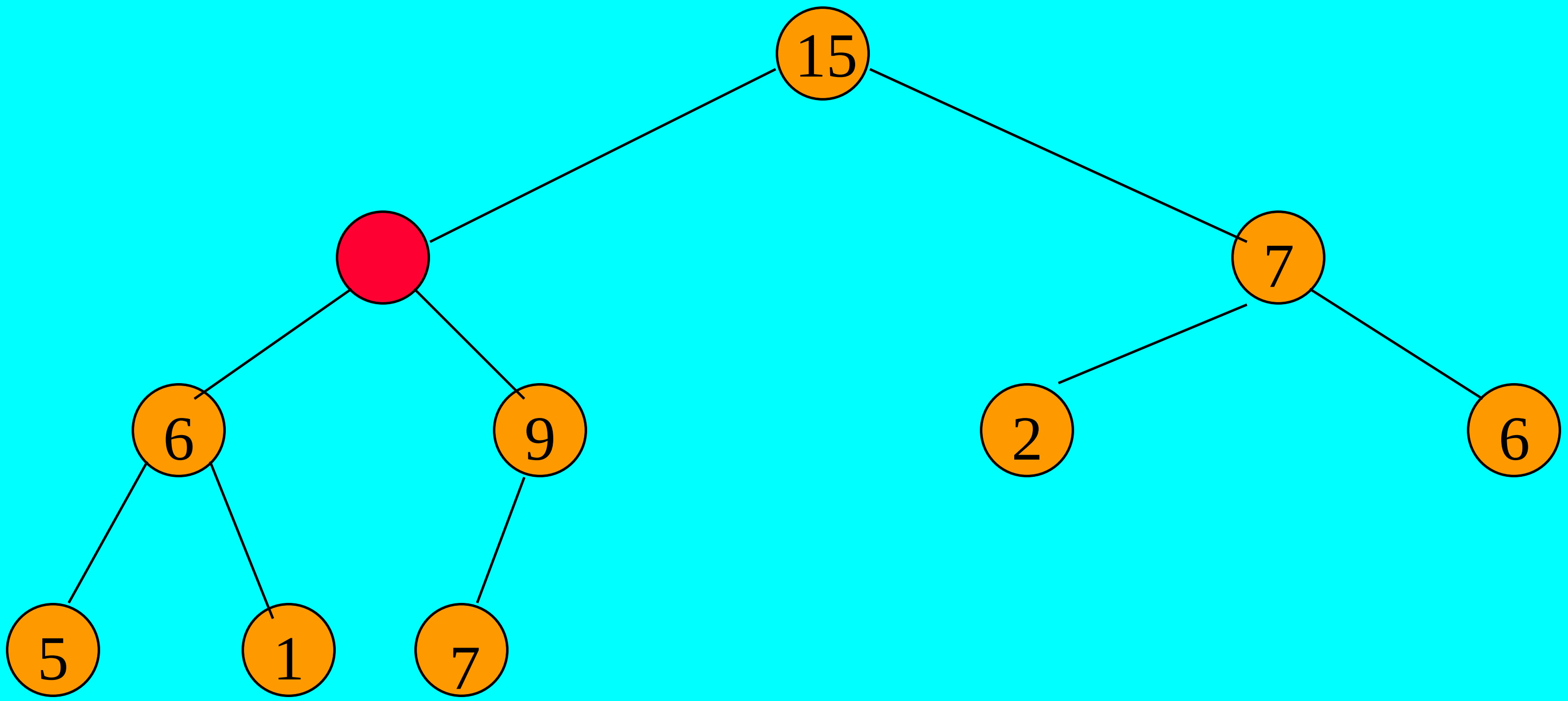
Reinsert 8 into the heap.

Removing The Max Element



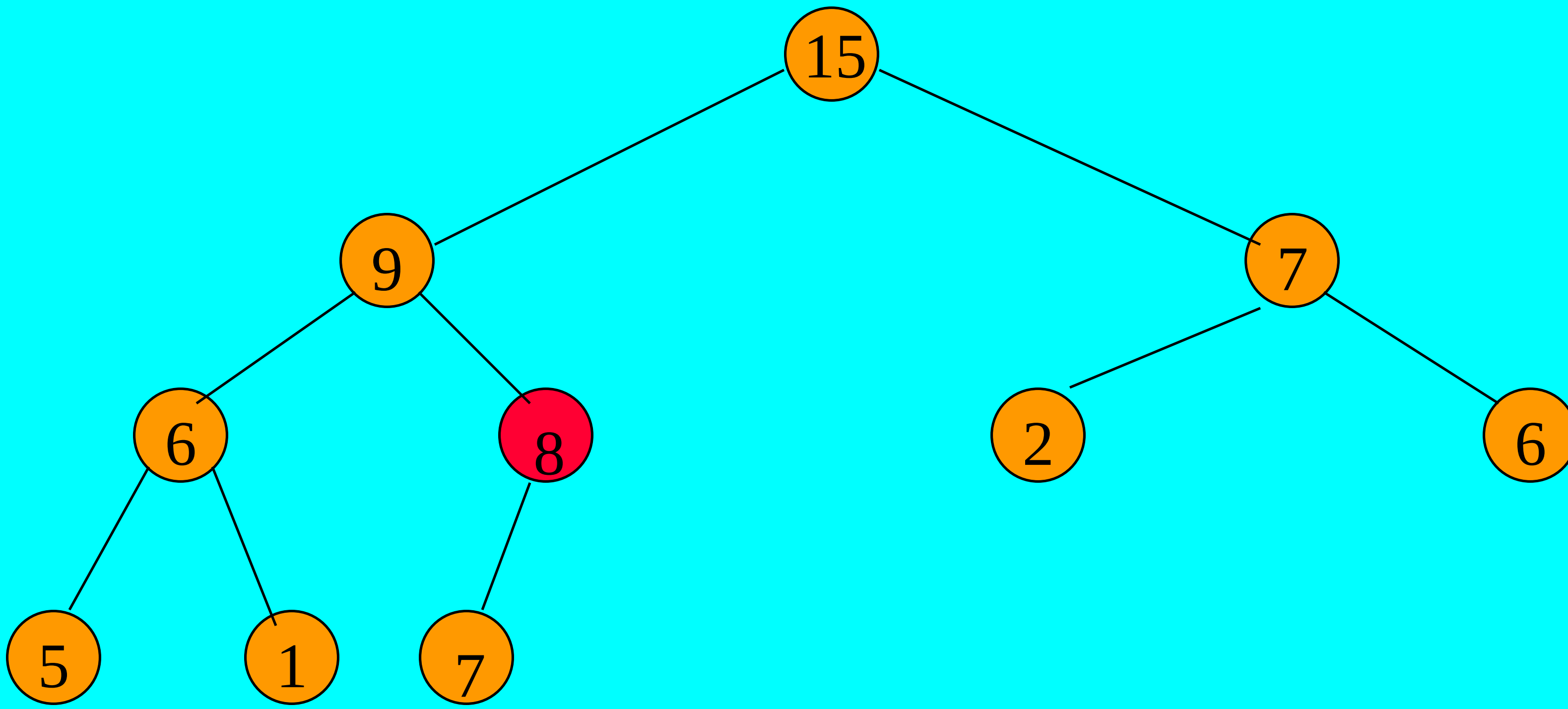
Reinsert 8 into the heap.

Removing The Max Element



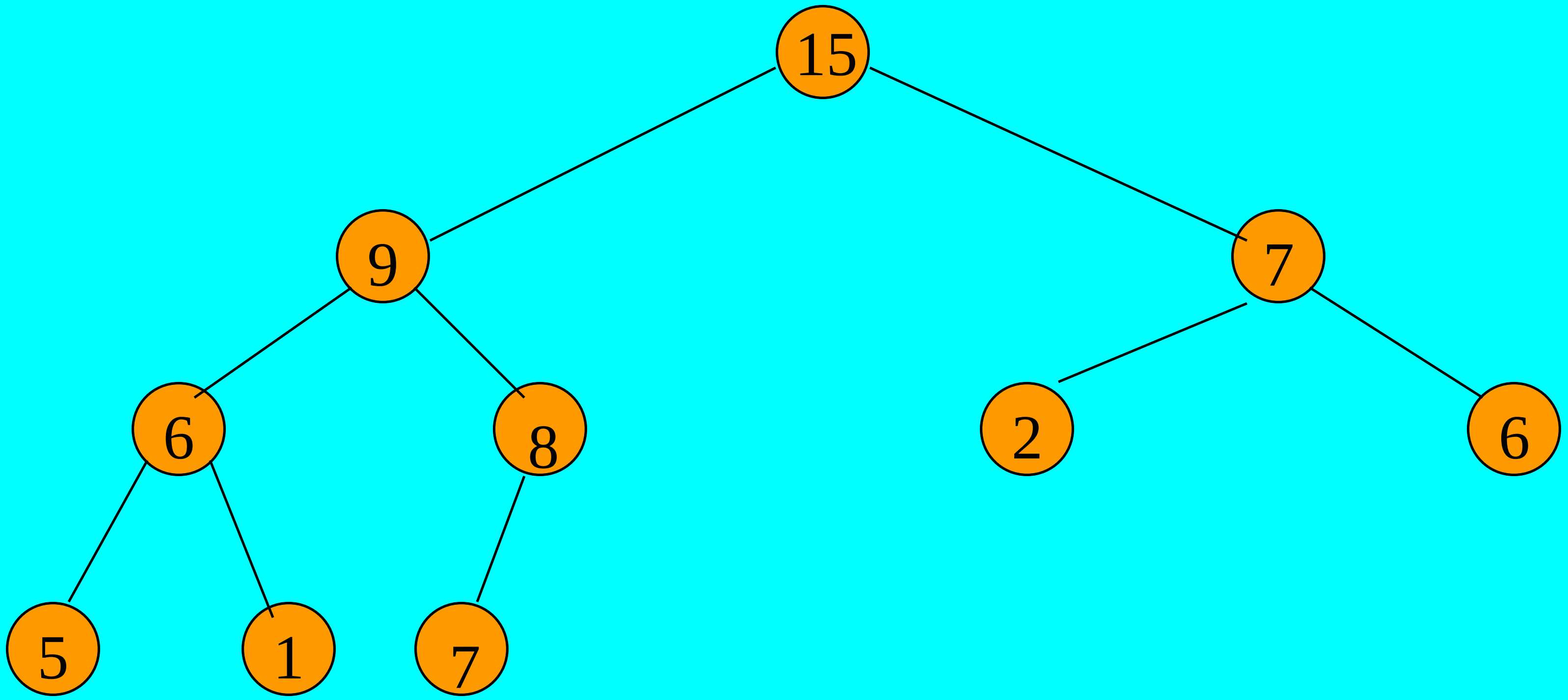
Reinsert 8 into the heap.

Removing The Max Element



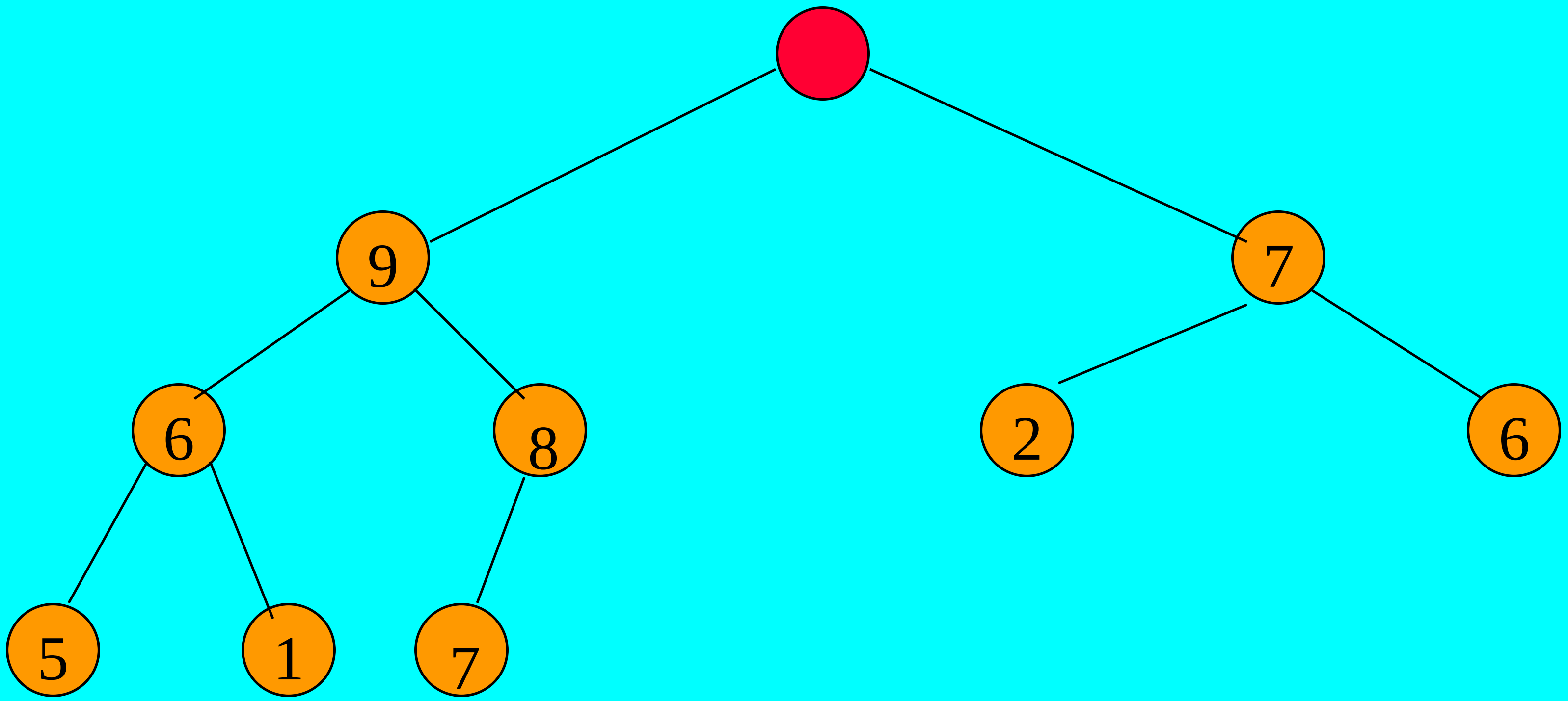
Reinsert 8 into the heap.

Removing The Max Element



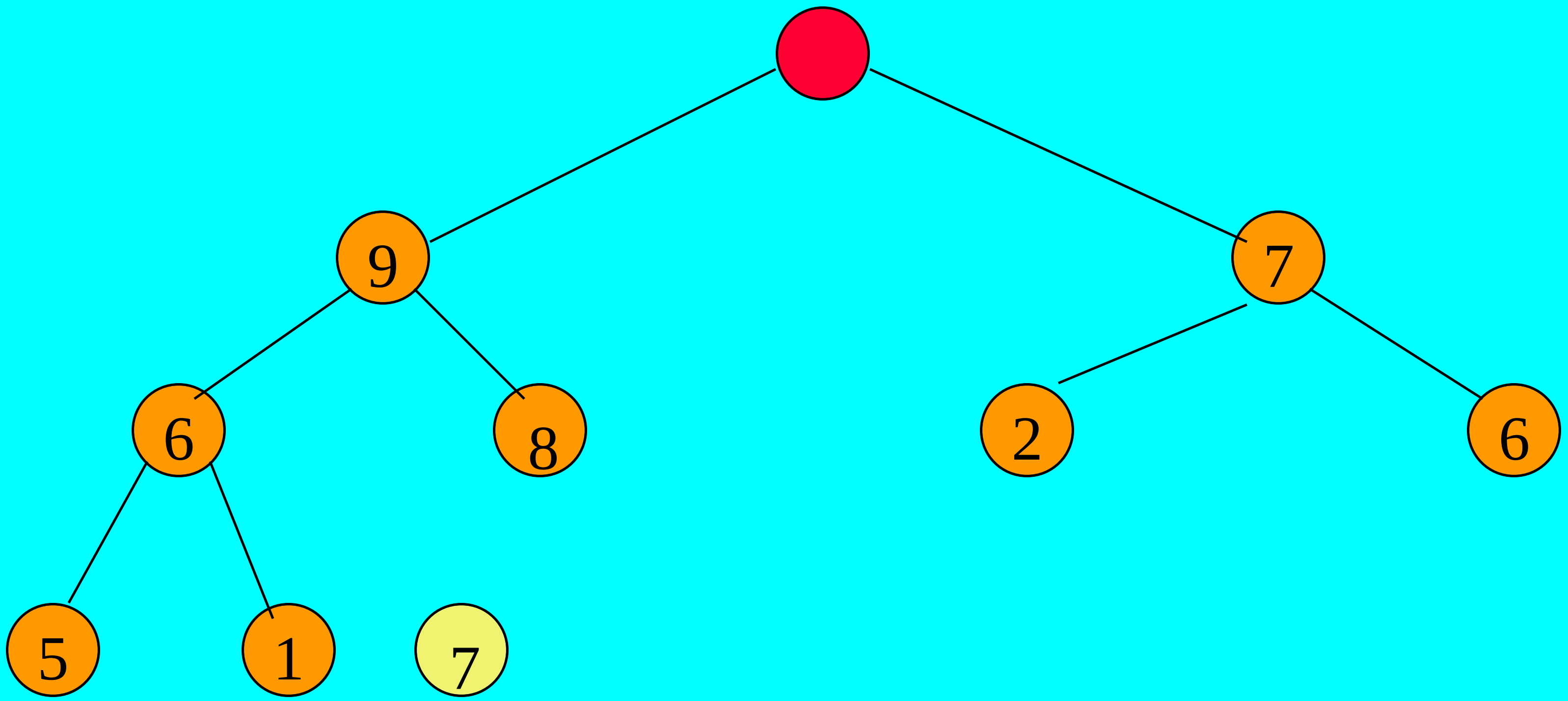
Max element is 15.

Removing The Max Element



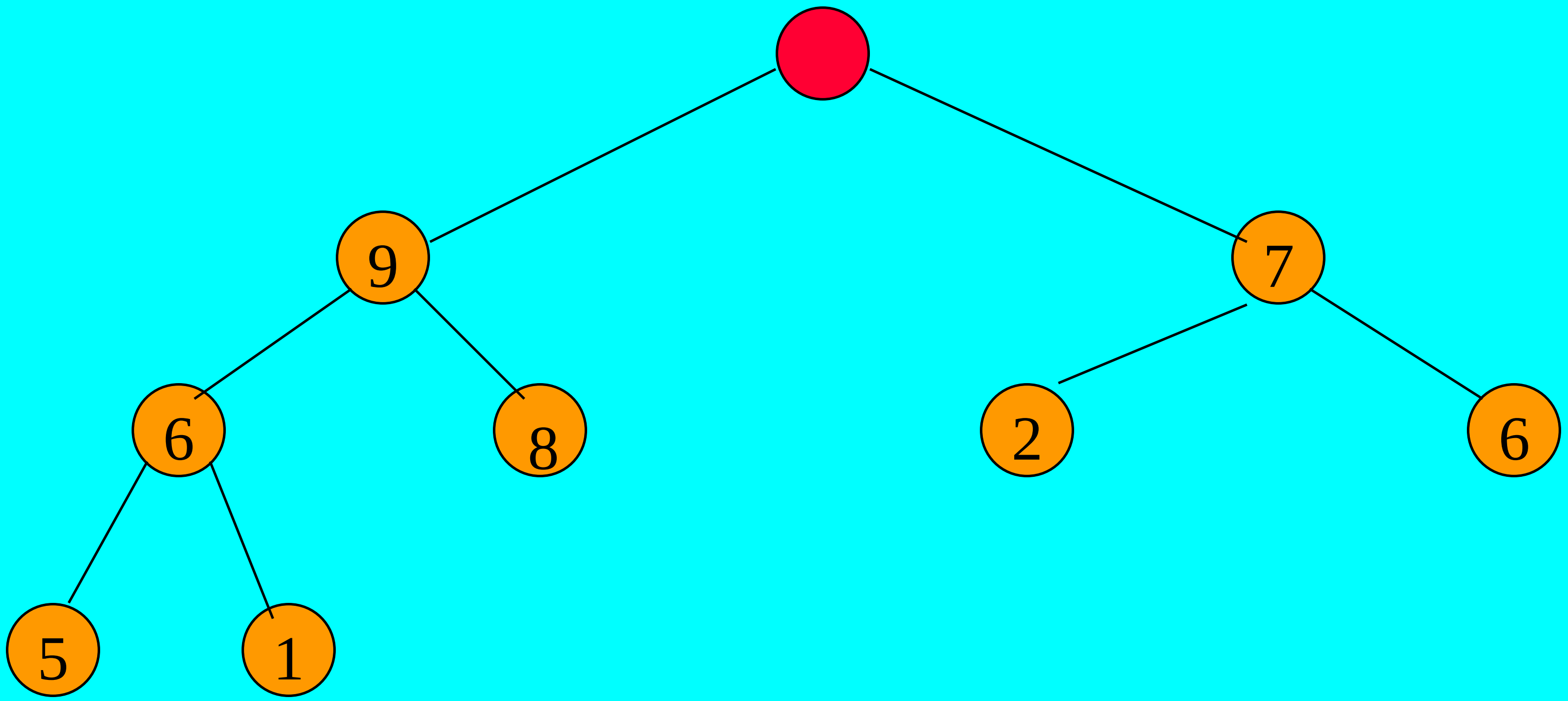
After max element is removed.

Removing The Max Element



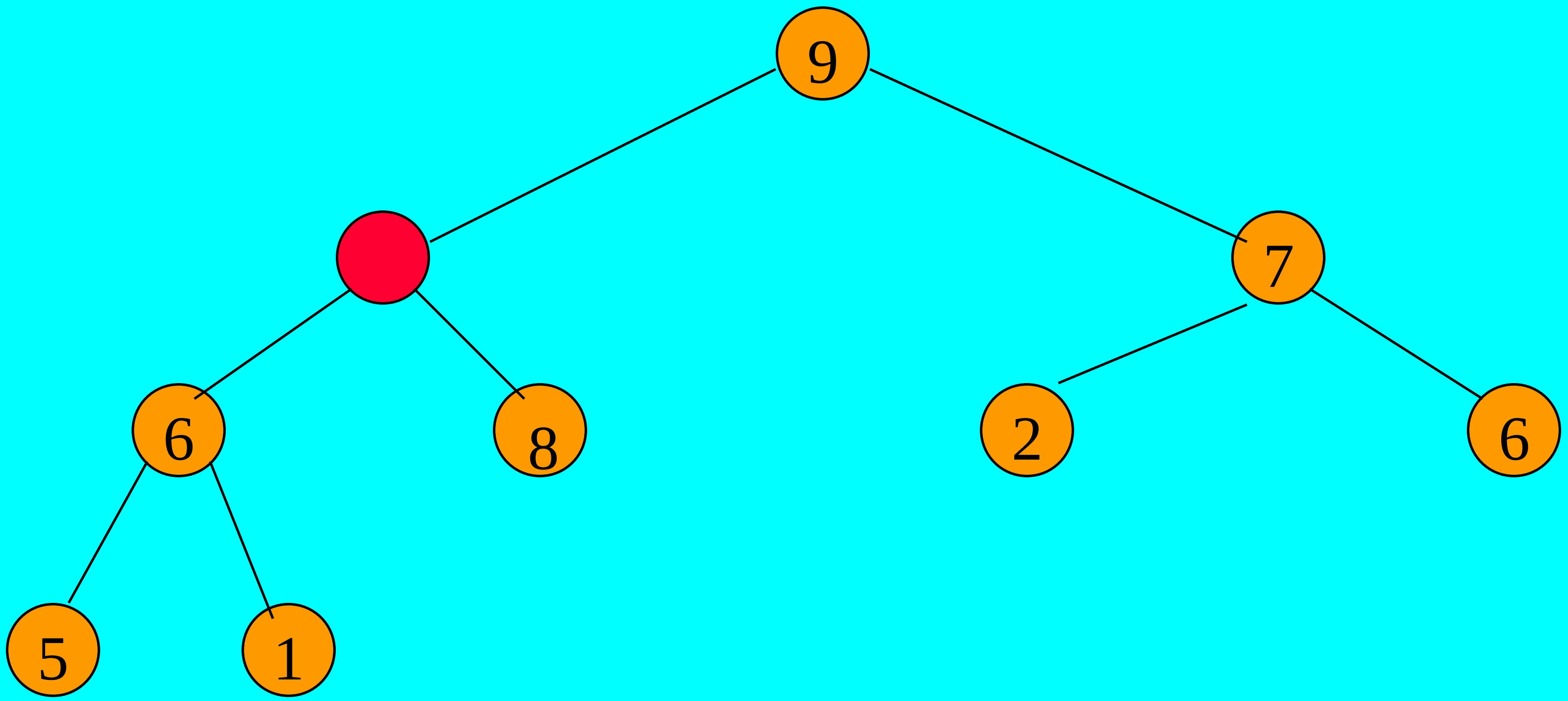
Heap with 9 nodes.

Removing The Max Element



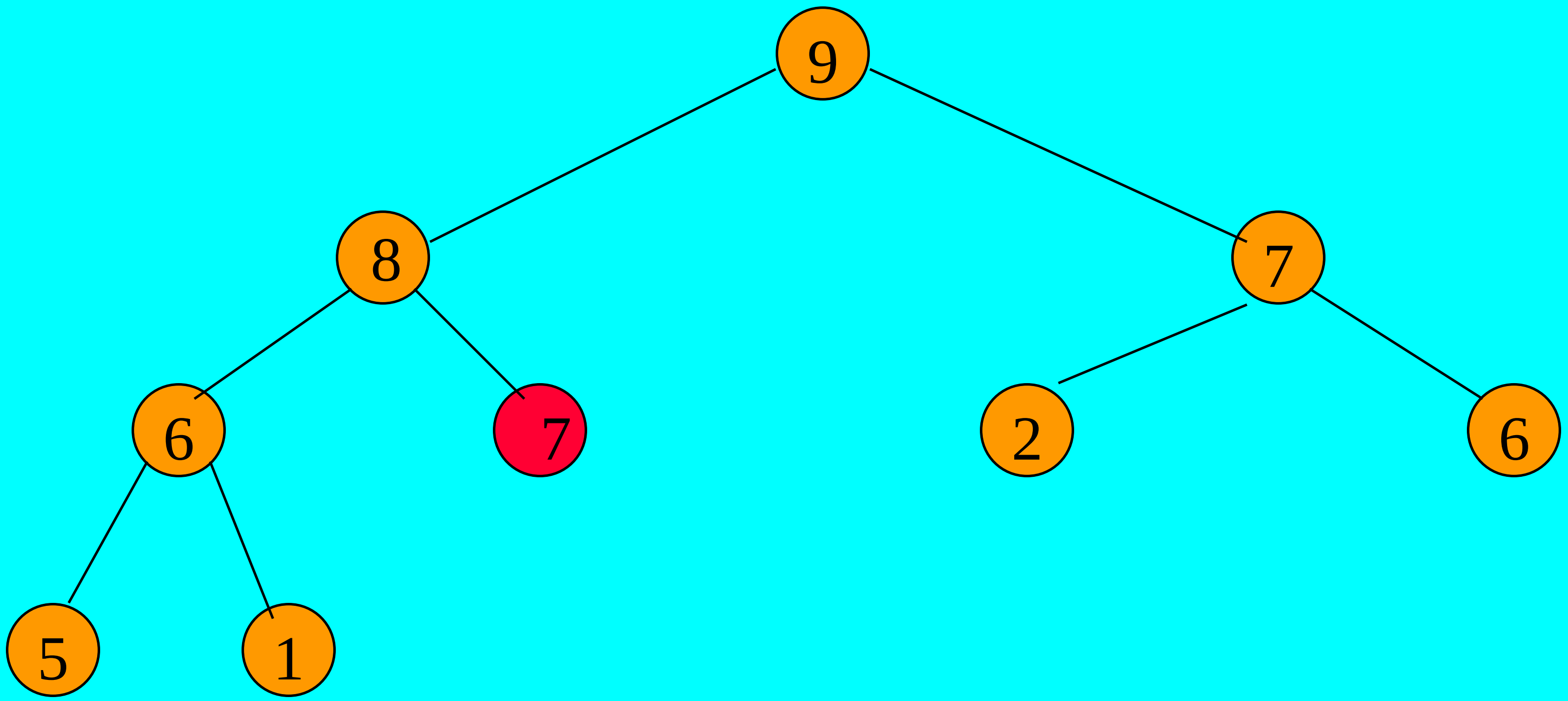
Reinsert 7.

Removing The Max Element



Reinsert 7.

Removing The Max Element



Reinsert 7.

Removal of root: The procedure for deleting the root from the heap (effectively extracting the max/min element in a max-heap or in a min-heap) and restoring the properties is called down-heap (also known as heapify-down, cascade-down and extract-min/max).

Delete

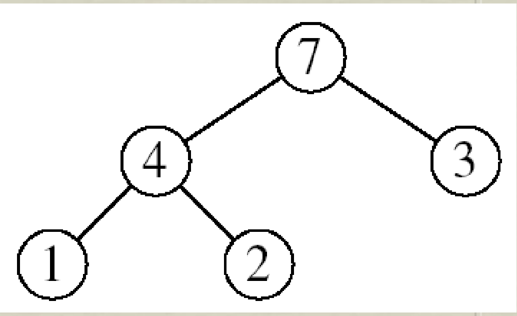
1. **Replace/exchange** *the root* of the heap with *the last* element on the last level.

2. **Reduce** the heap size by one and

3. **Perform the** MAX-HEAPIFY**(A, 0, n-1)** **function for the root i=0.**

Heapsort

- Goal:
  - Sort an array using heap representations



- Idea:
  1. Build a max-heap from the array
  2. Swap the root (the maximum element) with the last element in the array
  3. "Discard" this last node by decreasing the heap size
  4. Call MAX-HEAPIFY on the new root
  5. Repeat this process until only one node remains

Alg: HEAPSORT(*A*)

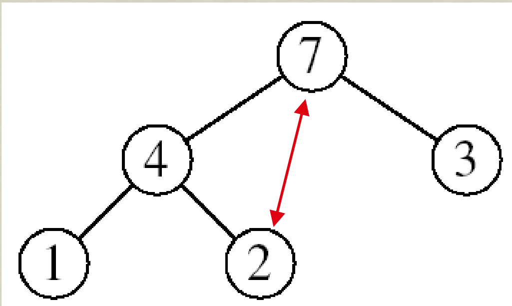
1. BUILD-MAX-HEAP(A) ………………. O(n)
2. for i ← from n-1 down to 1
3. do exchange A[0] ↔ A[i]

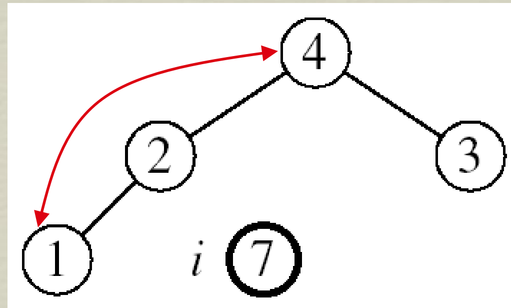
   MAX-HEAPIFY(A, 0, i) …. O(logn)

n-1 times

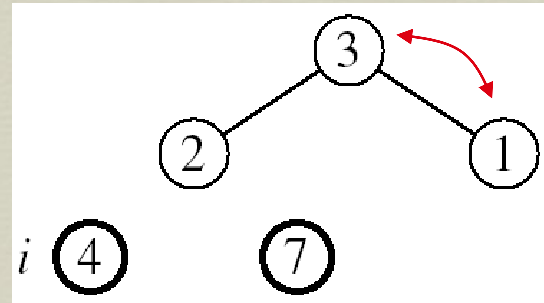
- Running time: O(nlogn)

Example: A=[7, 4, 3, 1, 2]

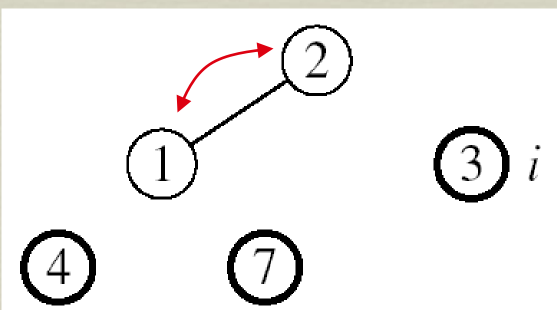




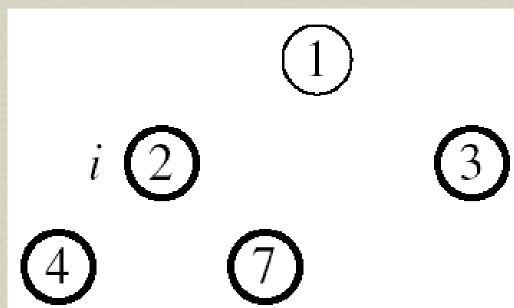
MAX-HEAPIFY(A, 0, 3)



MAX-HEAPIFY(A, 0, 2)



MAX-HEAPIFY(A, 0, 1)



A	1	2	3	4	7

Uses of Heaps

There are two main uses of heaps.

- The first is as a way of implementing a special kind of queue, called a priority queue.
- The second application is sorting.

- To sort an array, or list, containing N values there are two steps:
  - insert each value into a heap (initially empty)
  - remove each value form the heap in ascending order (this is done by N successive calls to get_smallest).

What is the complexity of the HeapSort algorithm?

(N insert operations) + (N delete operations)

- Each insert and delete operation is O(logN) at the very worst - the heap does not always have all N values in it. So, the complexity is certainly no greater than O(NlogN).

Some Important Properties of a Heap

- Given n, there exists a unique binary tree with n nodes that

  is essentially complete, with $h = \lfloor \log_2 n \rfloor$

- The root contains the largest key

- The subtree rooted at any node of a heap is also a heap

- A heap can be represented as an array

- A Heap is a data structure used to efficiently find the smallest (or largest) element in a set.
- Min-heaps make it easy to find the smallest element. Max-heaps make it easy to find the largest element.
- Heaps are based upon trees. These trees maintain the heap property.
  - The Heap invariant. The value of Every Child is greater than the value of the parent. We are describing Min-heaps here (Use less than for Max-heaps).
- The trees must be mostly balanced for the costs listed below to hold.
- Access to elements of a heap usually have the following costs.
  - The cost to find the smallest (largest) element takes constant time.
  - The cost to delete the smallest (largest) element takes time proportional to the log of the number of elements in the set.
  - The cost to add a new element takes time proportional to the log of the number of elements in the set.

- Heaps can be implemented using arrays (using the tree embedding described above) or by using balanced binary trees

- Trees with the leftist property have the following invariant.
  - The leftist invariant. The rank of every left-child is equal to or greater than the rank of the cooresponding right-child. The rank of a tree is the length of the right-most path.

- Heaps form the basis for an efficient sort called <u>heap sort that has cost proportional to n*log(n) where n is the number of elements to be sorted.</u>

- Heaps are the data structure most often used to implement <u>priority queues.</u>