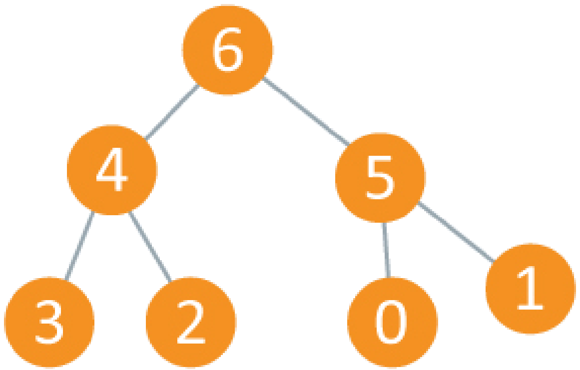
Priority Queues

Chapter 12

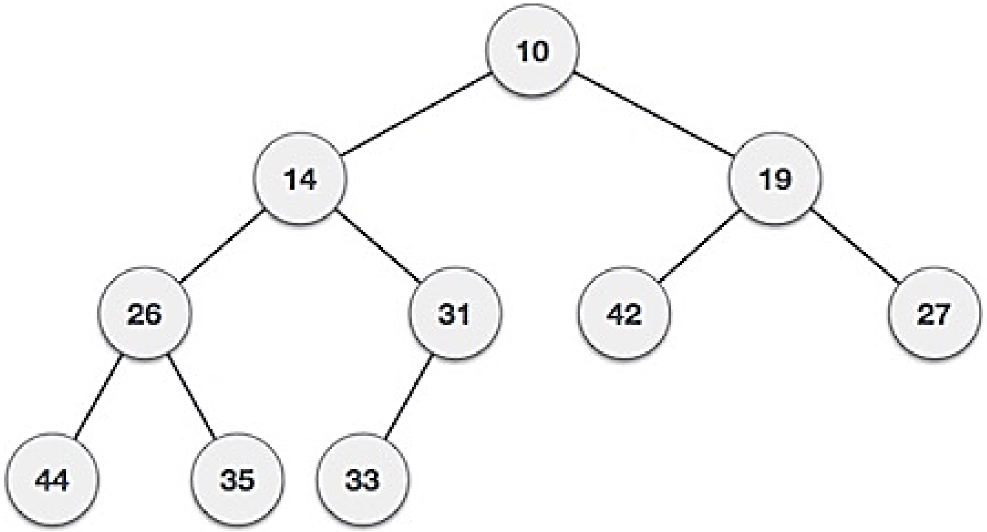
Heap

- A heap is a tree with some special properties
- The basic requirement of a heap is that the value of a node must be $\geq$ (or $\leq$) than the values of its children. This is called *heap property*.
- A heap also has the additional property that all leaves should be at h or $h - 1$ levels (where h is the height of the tree) for some $h > 0$ (*complete binary trees*).

Example: Heap



Max Heap :
Parent >= Children

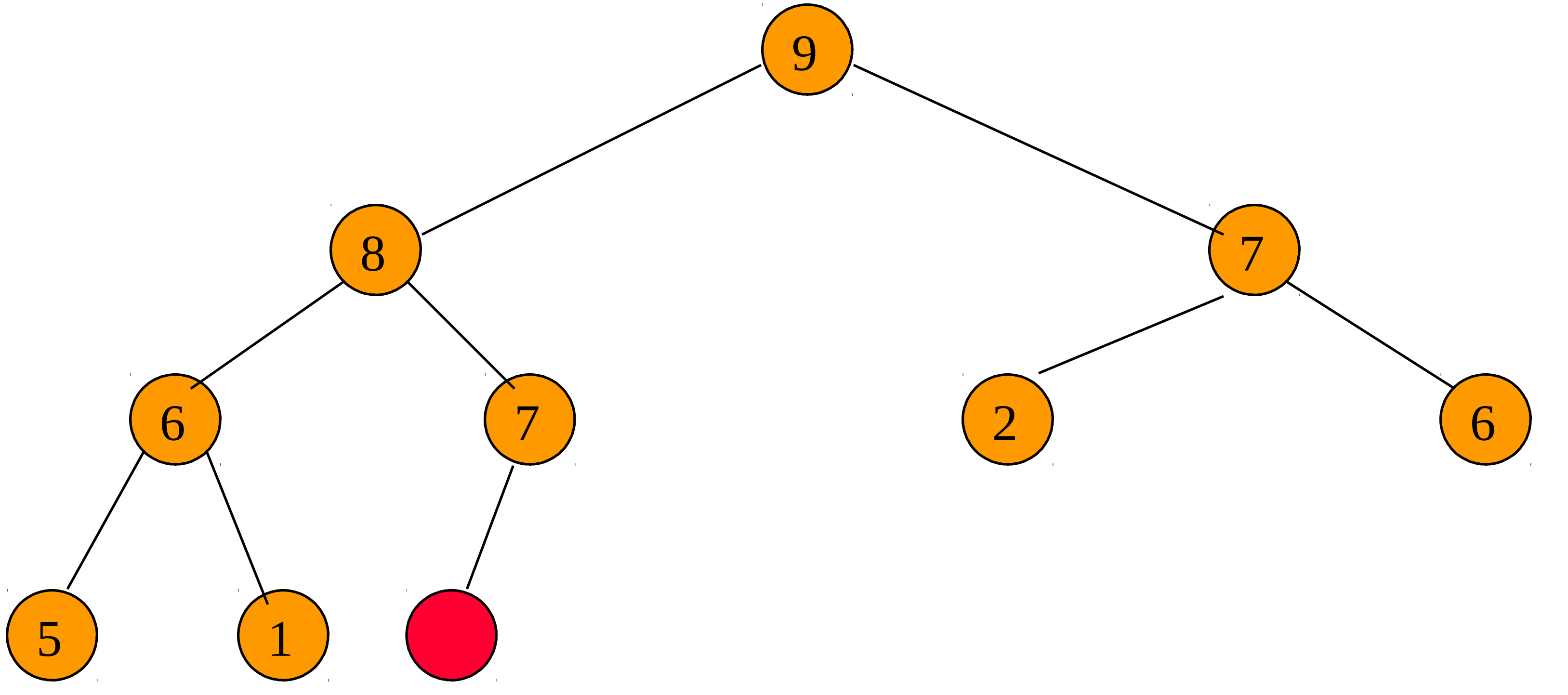


Min Heap :
Parent <= Children

MaxHeap Construction Animation

Input 35 33 42 10 14 19 27 44 26 31

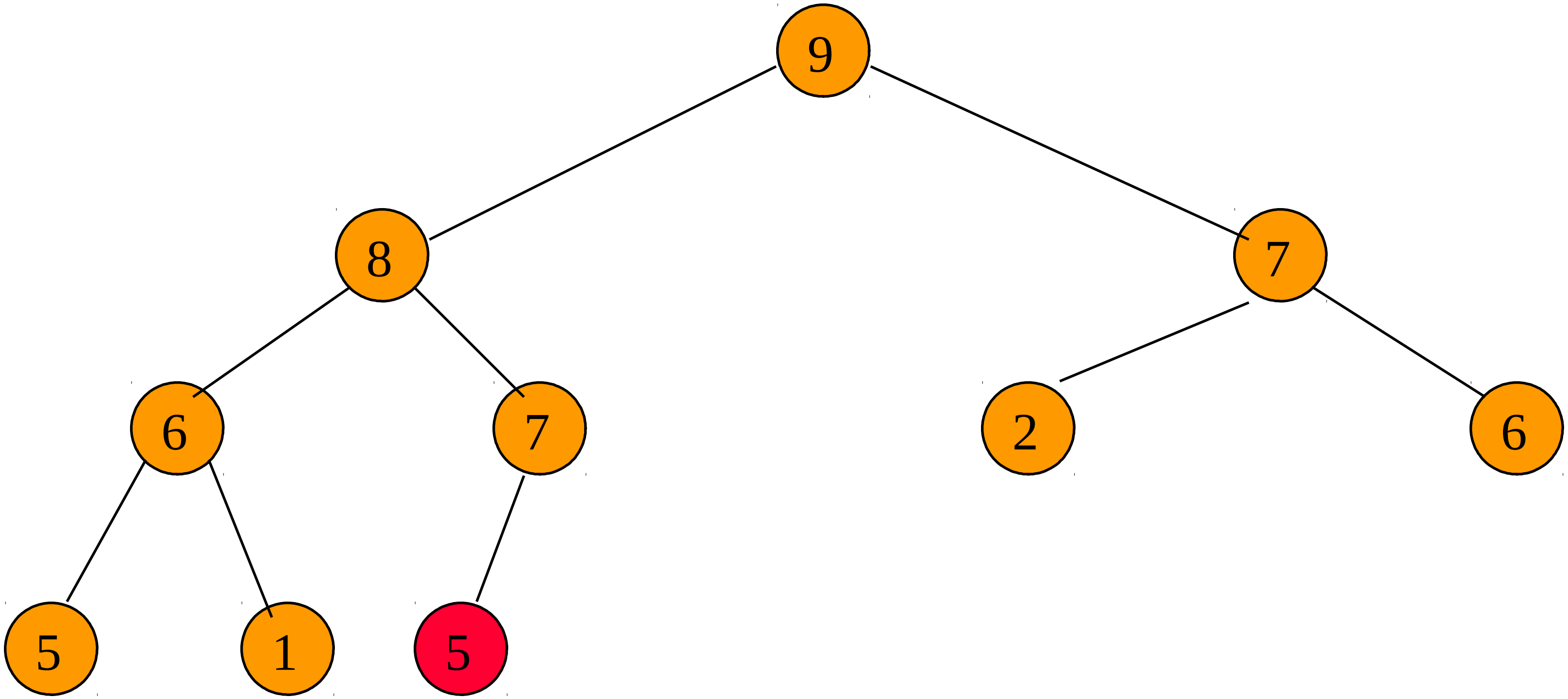
Inserting An Element Into A Max Heap



Complete binary tree with 10 nodes.

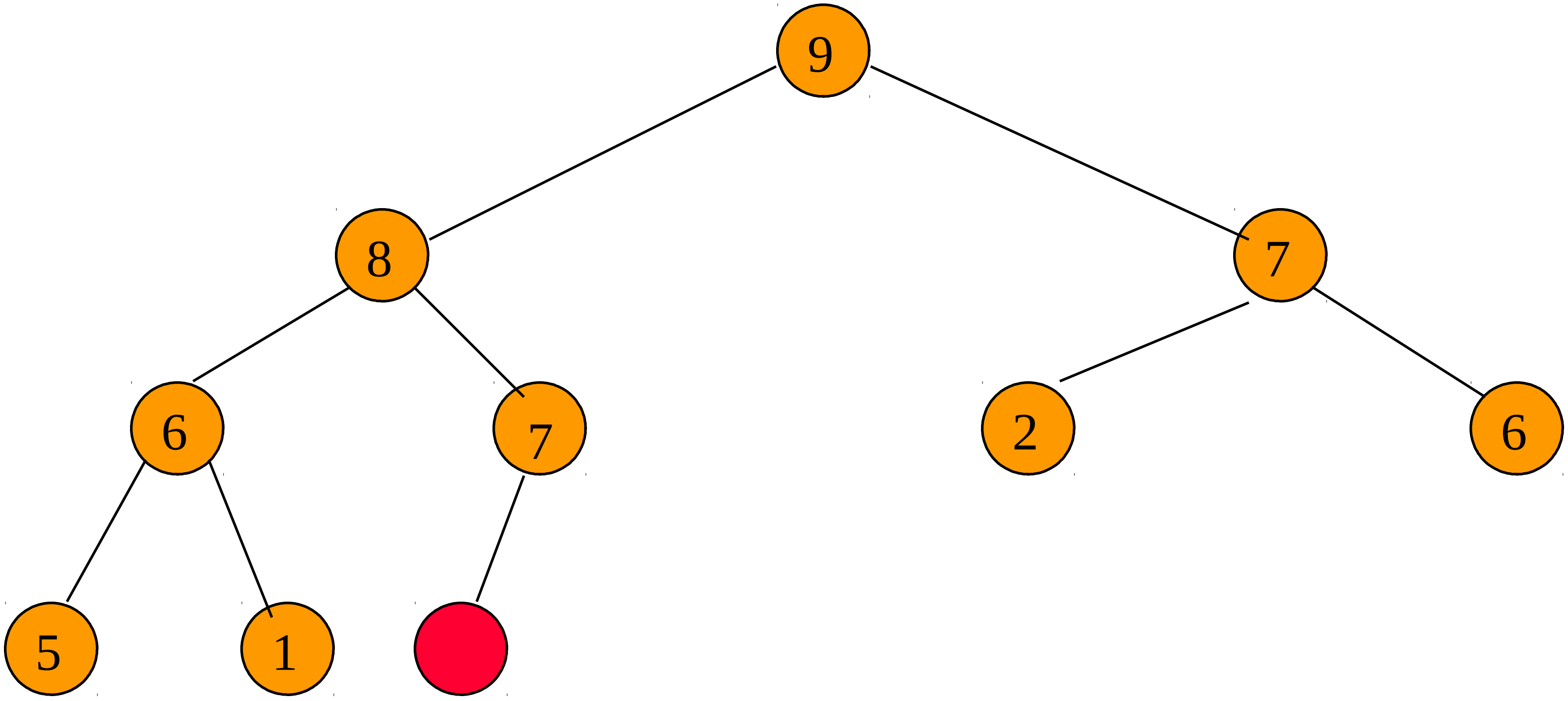
Insert 5 in 10^{th} node.

Inserting An Element Into A Max Heap



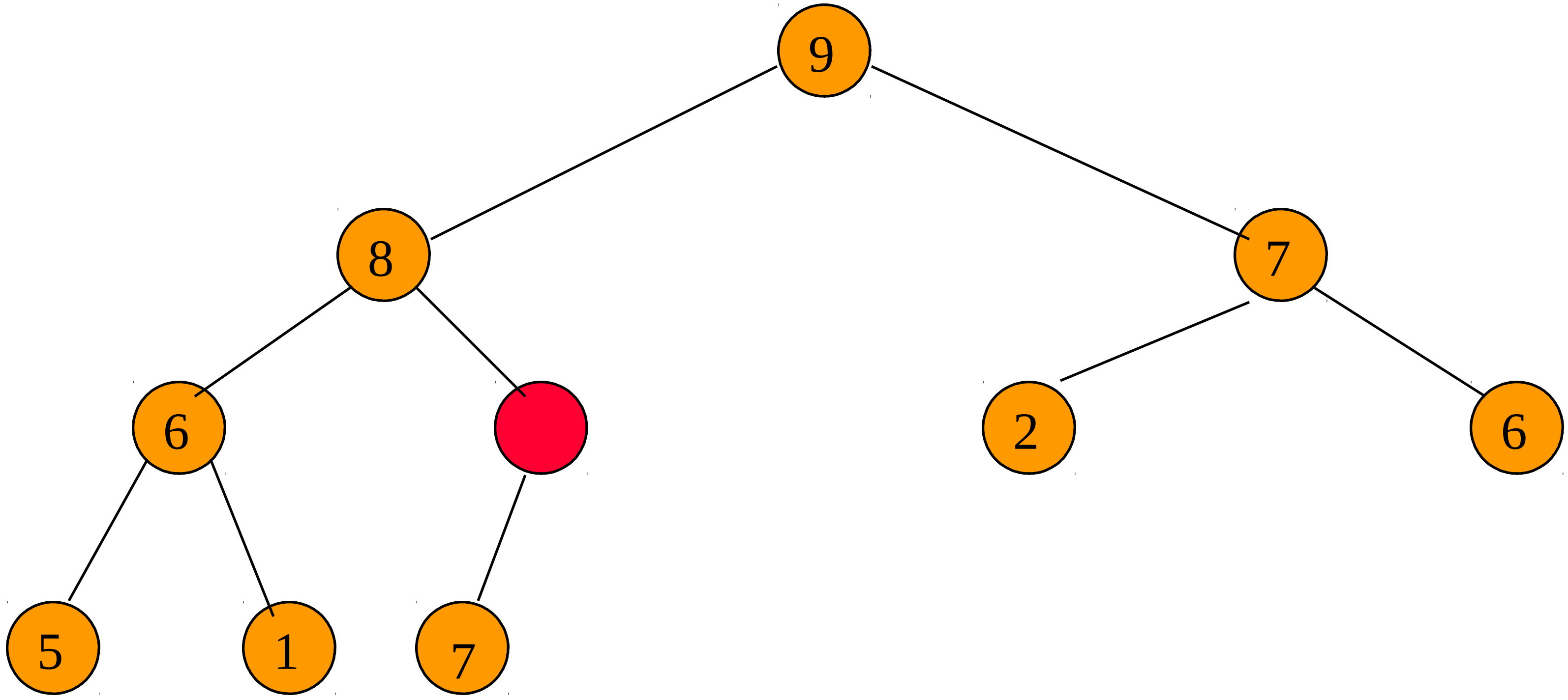
New element is 5. Need not heapify since the inserted element is smaller than parent node.

Inserting An Element Into A Max Heap



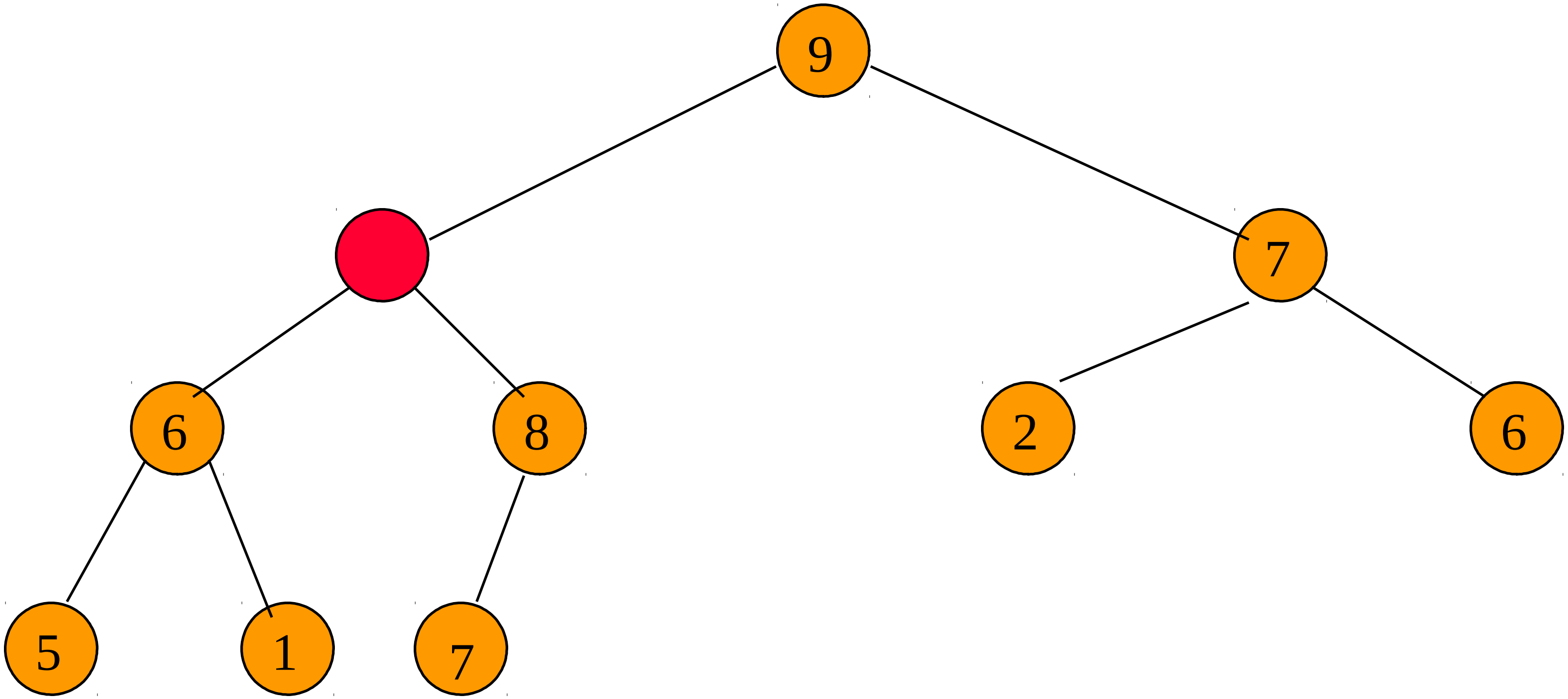
Suppose New element to be inserted in original tree is 20. This needs Heapifying

Inserting An Element Into A Max Heap



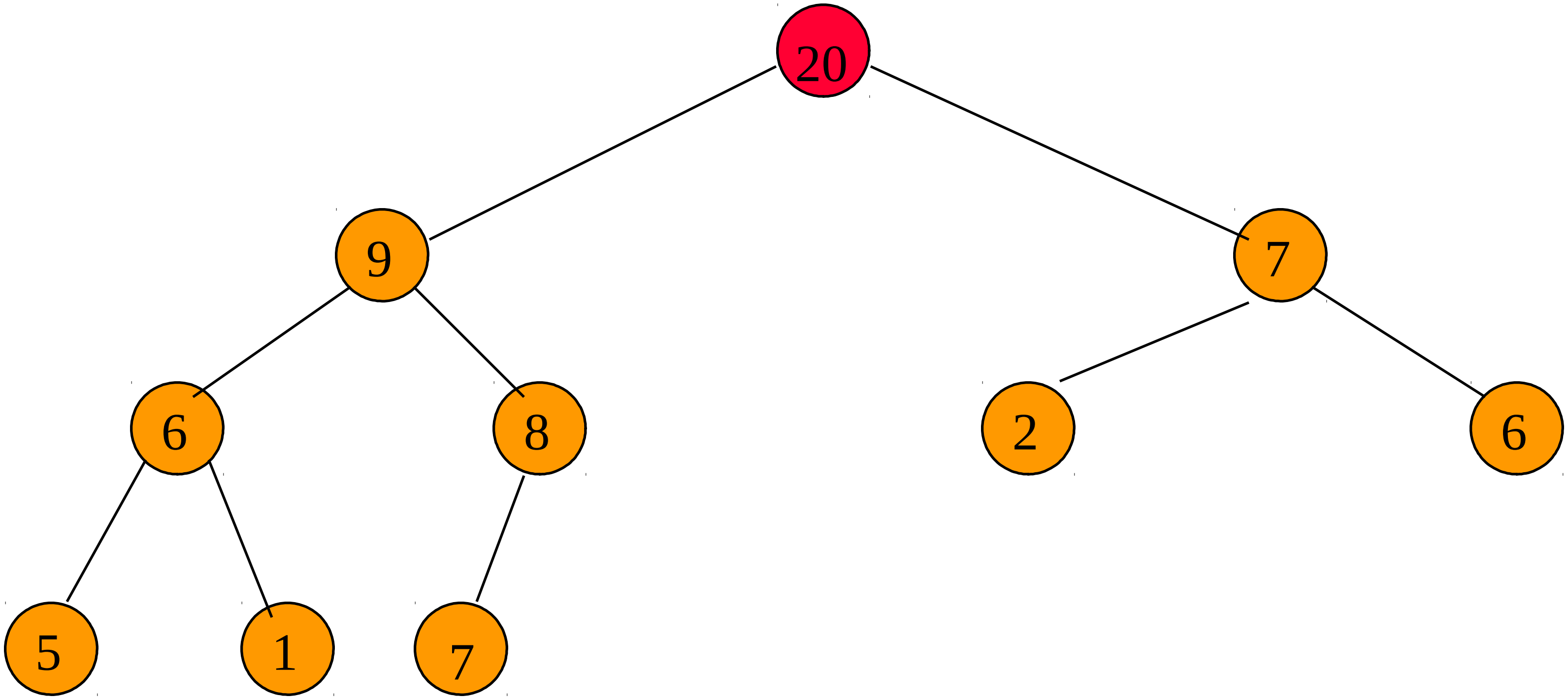
New element is 20.

Inserting An Element Into A Max Heap



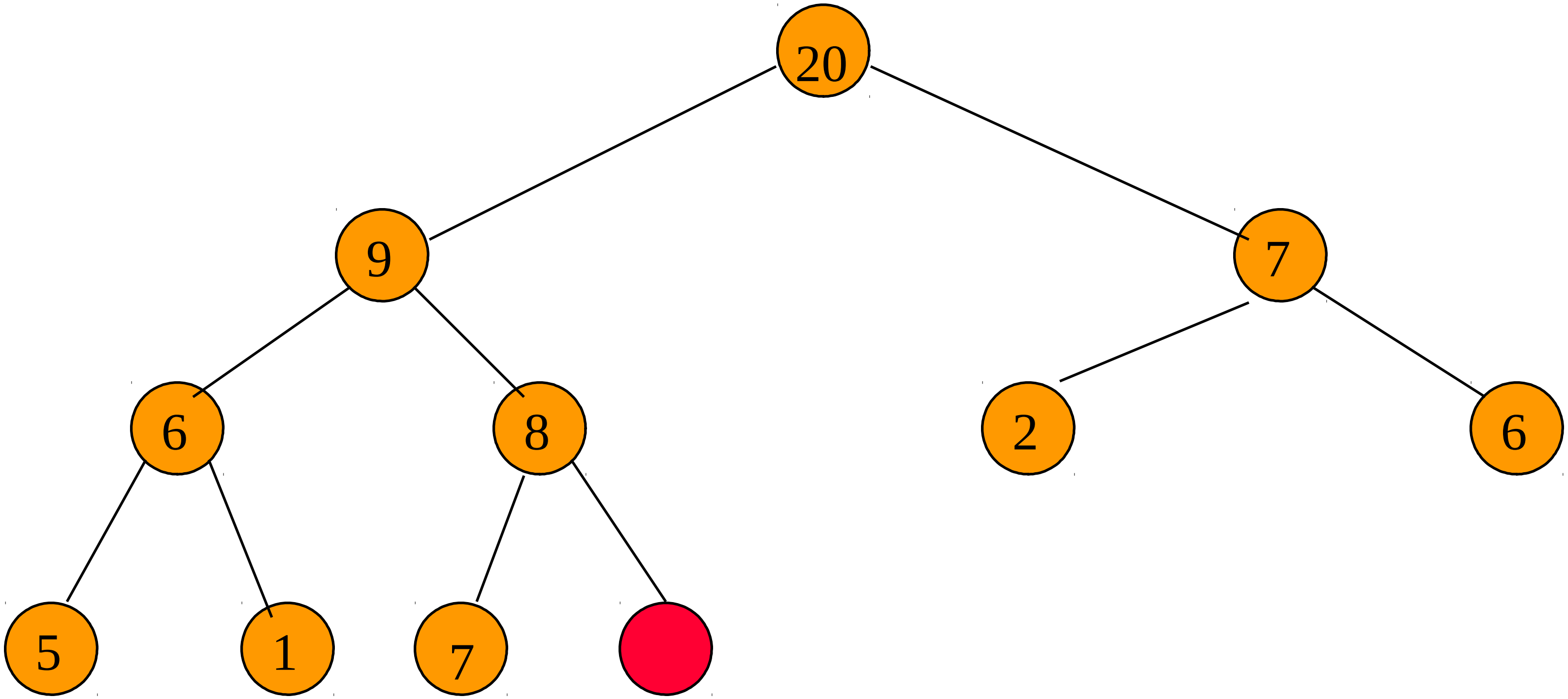
New element is 20.

Inserting An Element Into A Max Heap



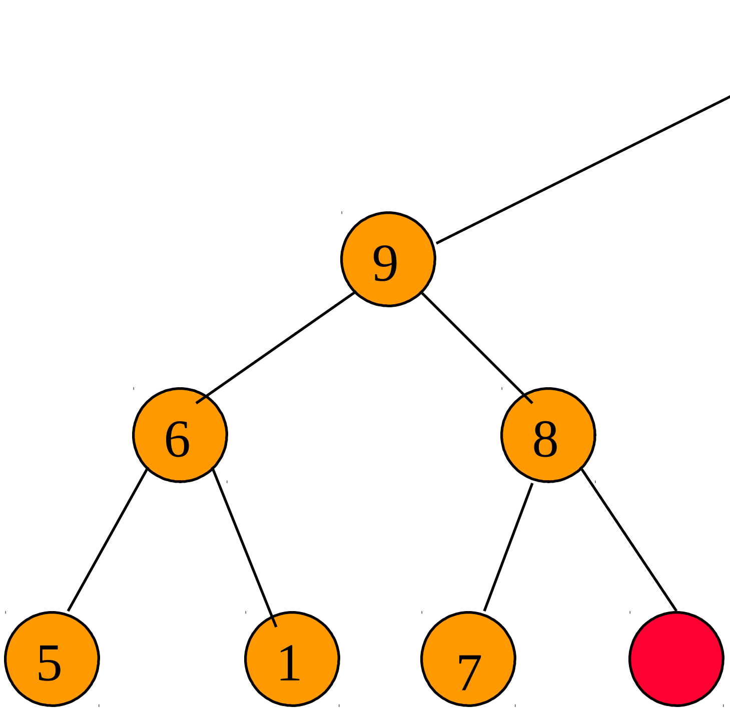
New element is 20.

Inserting An Element Into A Max Heap



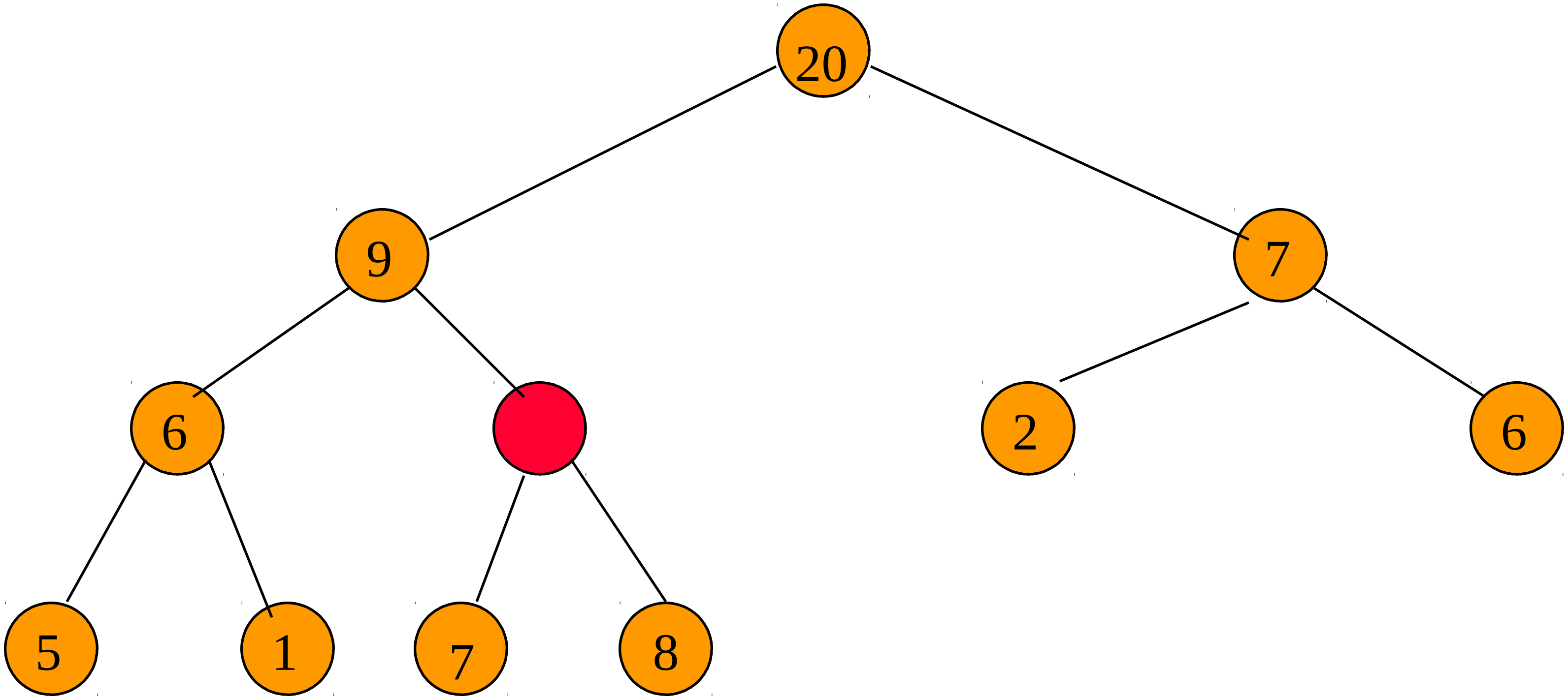
Complete binary tree with 11 nodes.

Inserting An Element Into A Max Heap



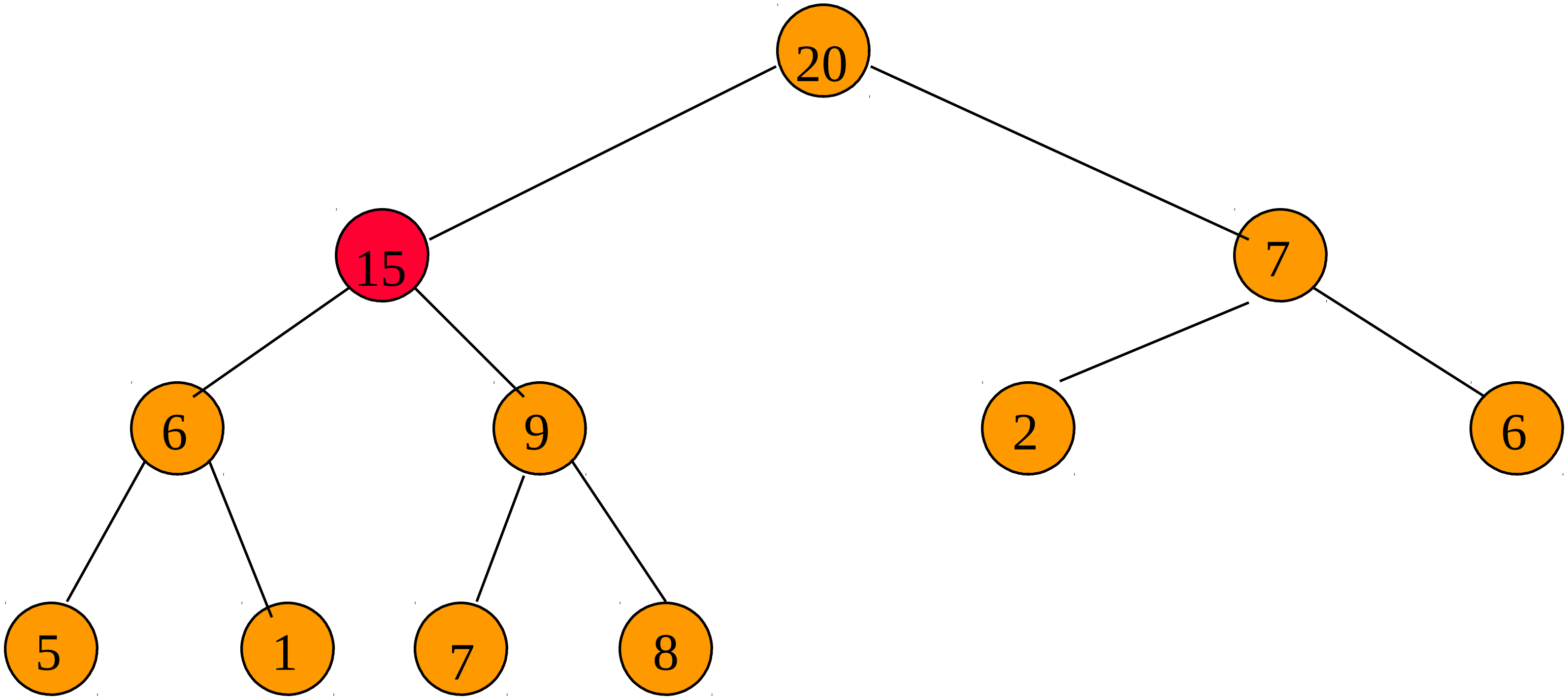
Suppose New element to be inserted is 15 in 11th position. Again heapify

Inserting An Element Into A Max Heap



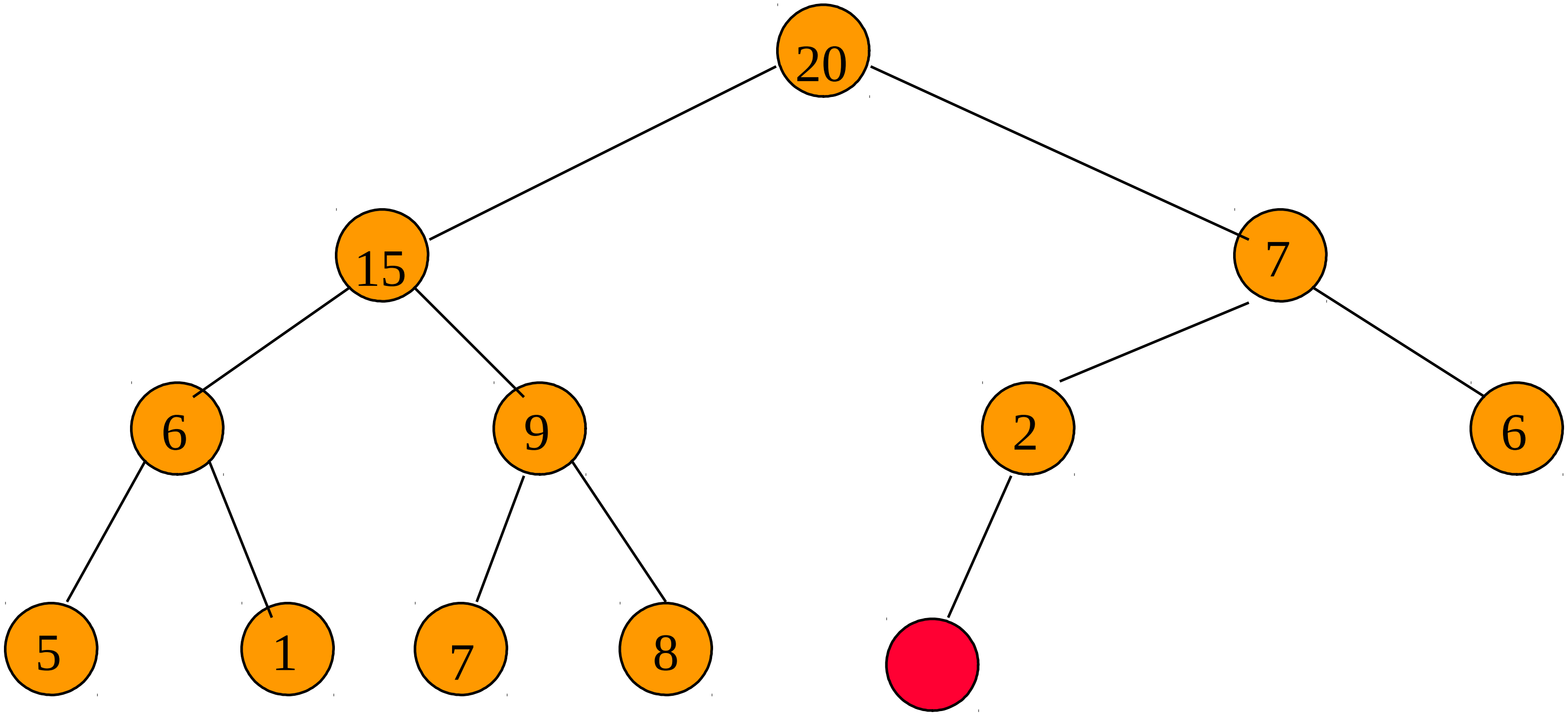
New element is 15.

Inserting An Element Into A Max Heap



New element is 15.

Complexity Of Insert

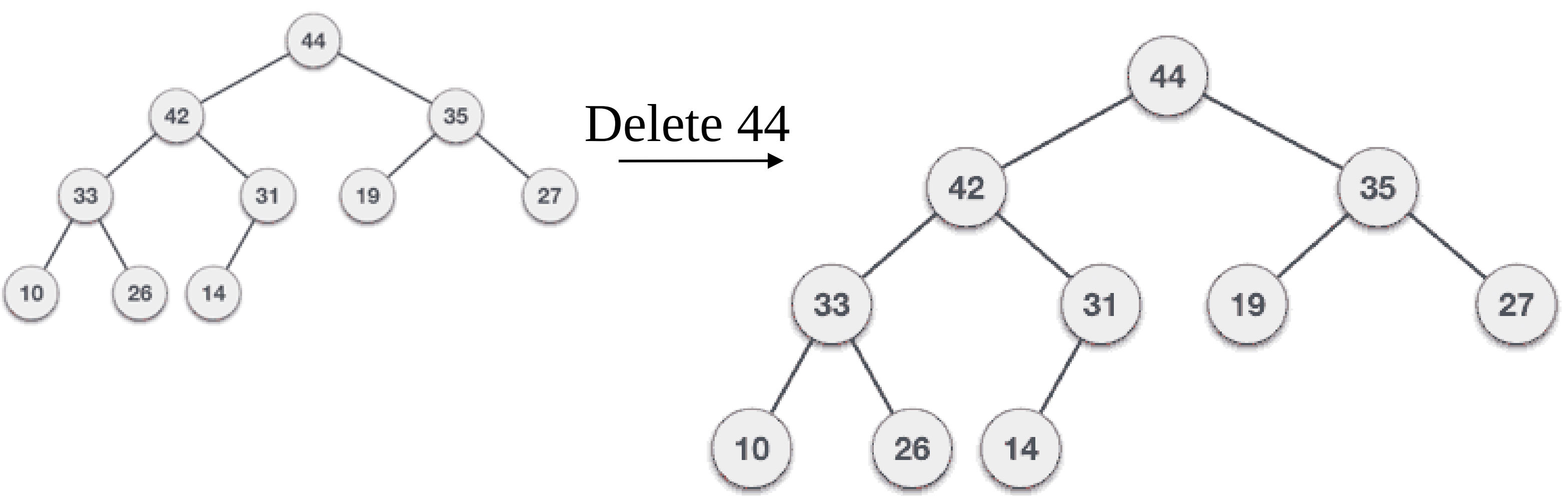


Complexity is O(log n), where n is heap size.

Max Heap Deletion Animation

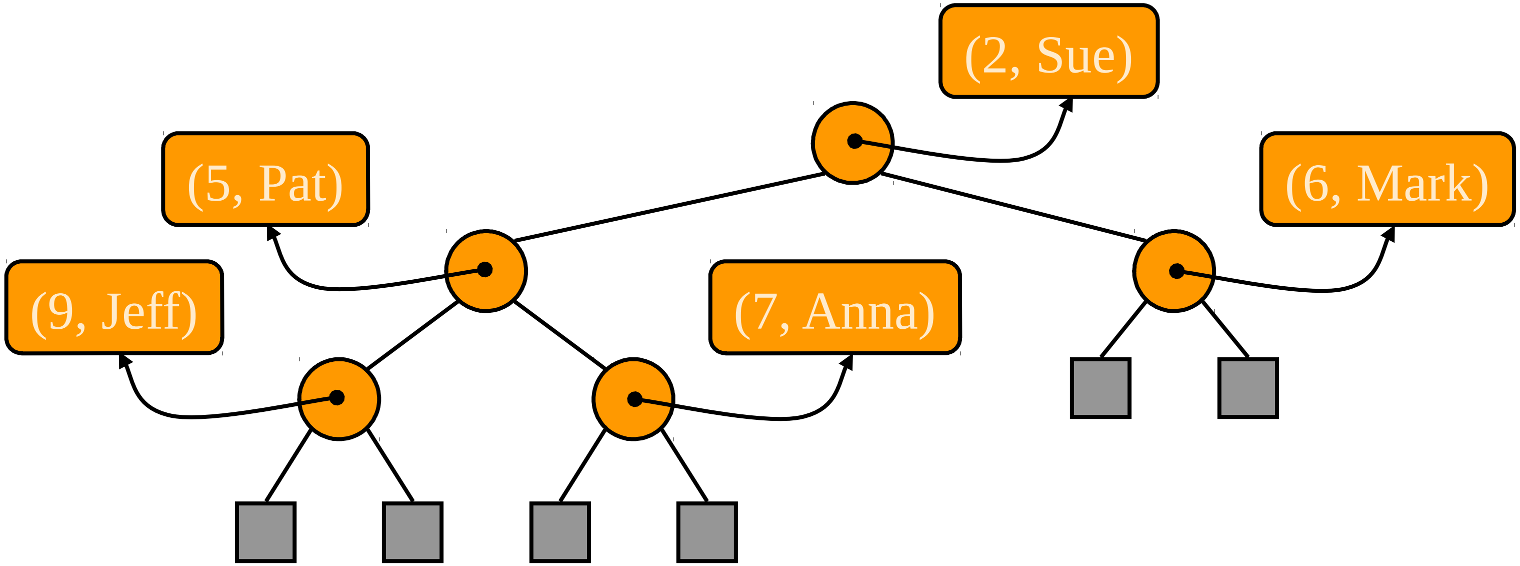
- Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

```
Step 1 - Remove root node.
Step 2 - Move the last element of last level to root.
Step 3 - Compare the value of this child node with its parent.
Step 4 - If value of parent is less than child, then swap them.
Step 5 - Repeat step 3 & 4 until Heap property holds.
```



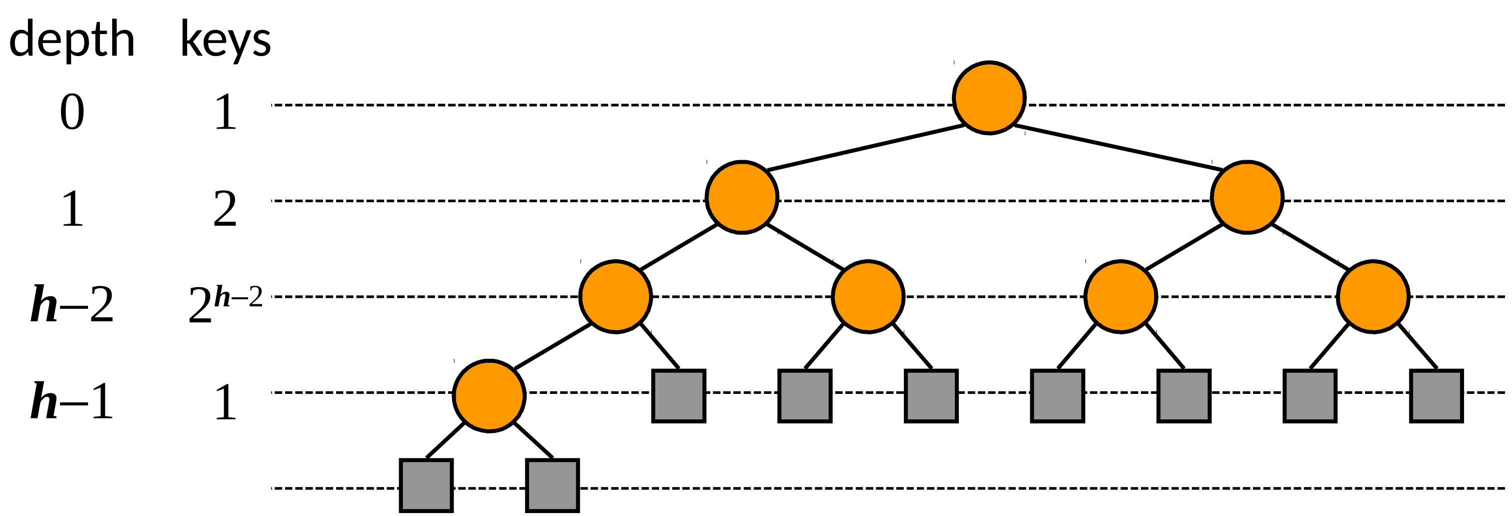
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



Height of a Heap

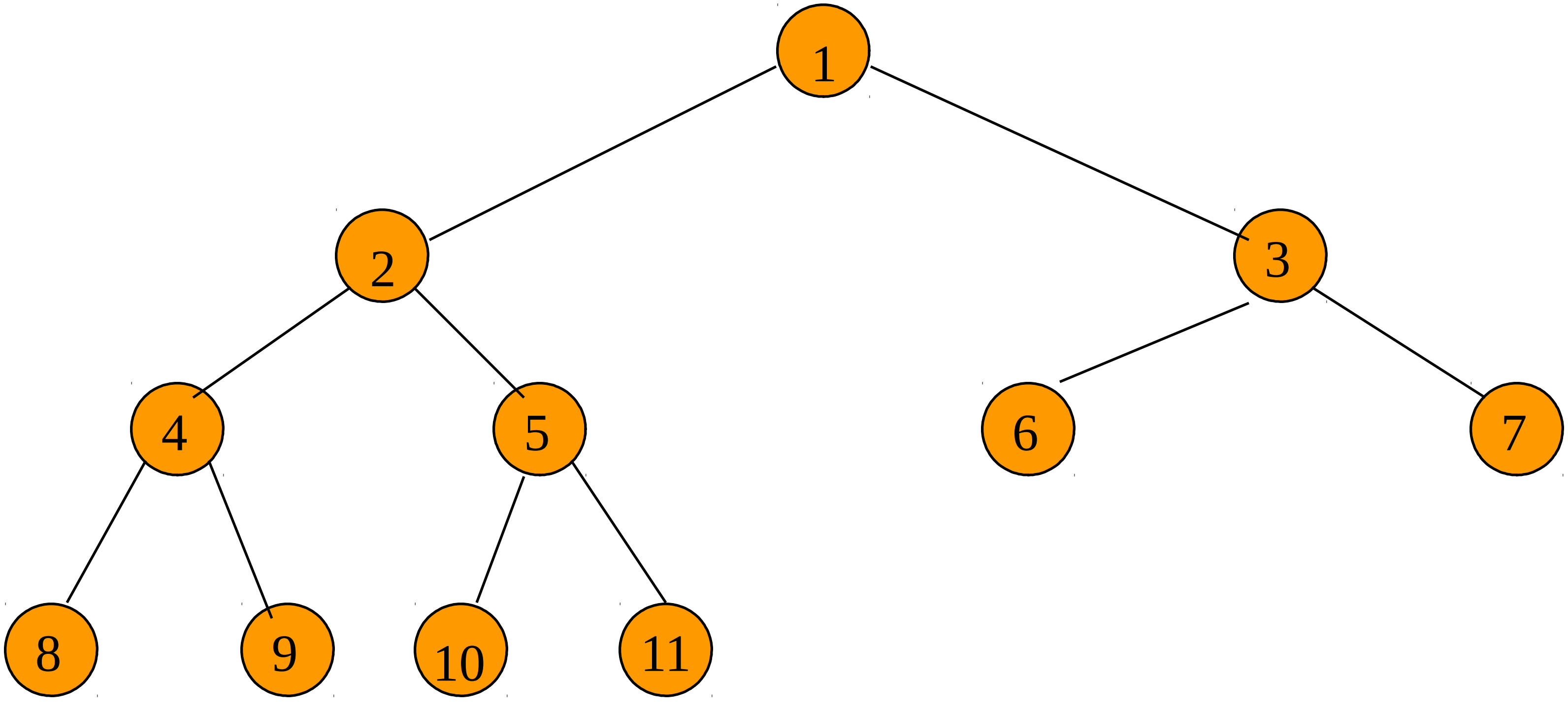
- Theorem: A heap storing n keys has height $O(\log n)$
- Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



Heap Height

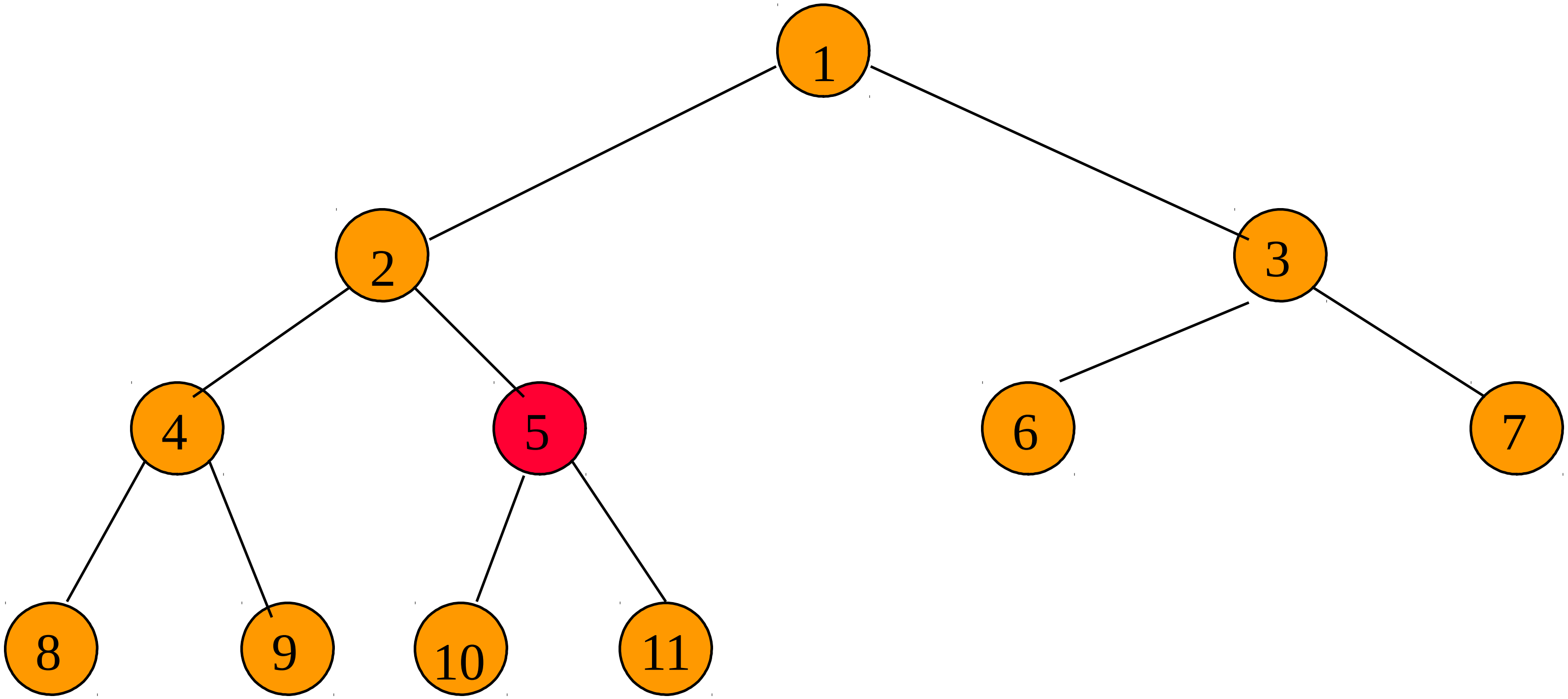
Since a heap is a complete binary tree, the height of an n node heap is $\log_2 (n+1)$.

Initializing A Max Heap



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

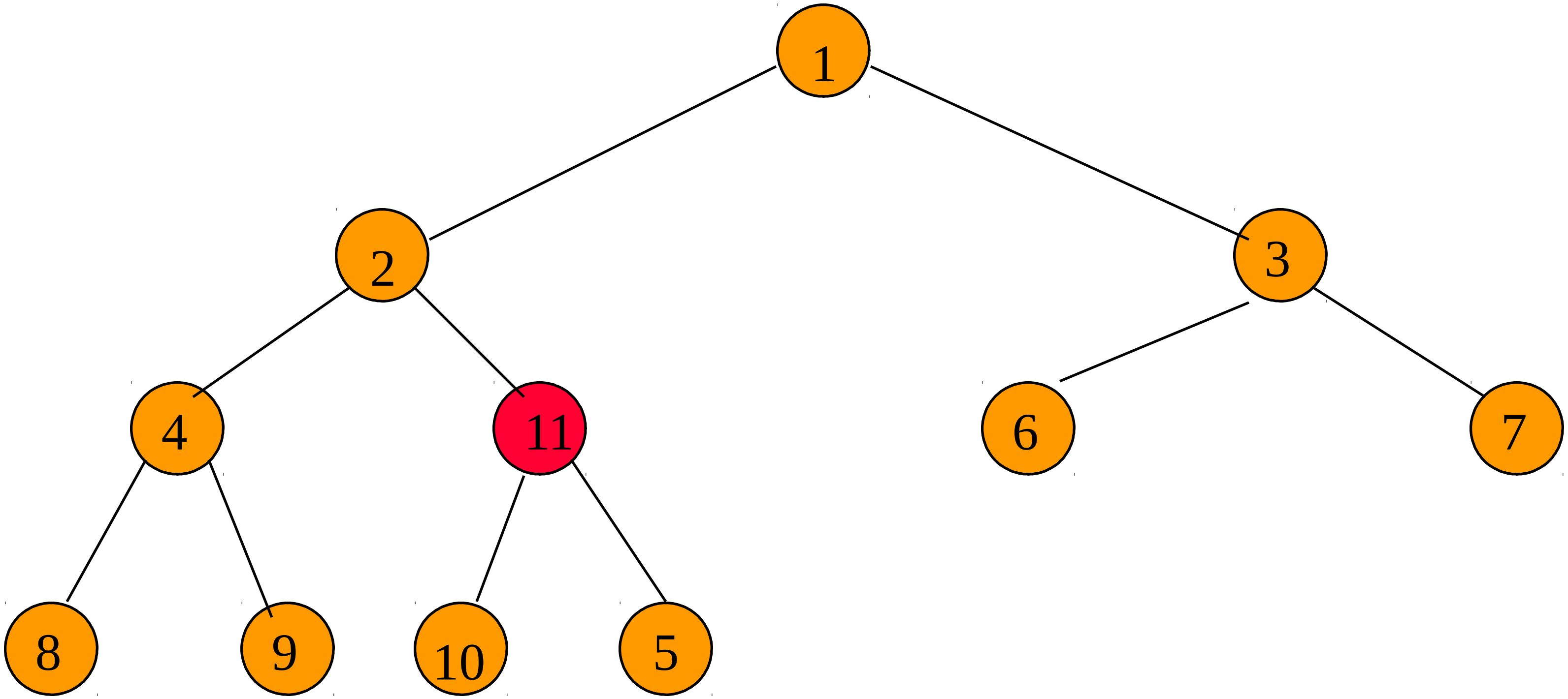
Initializing A Max Heap



Start at rightmost array position that has a child.

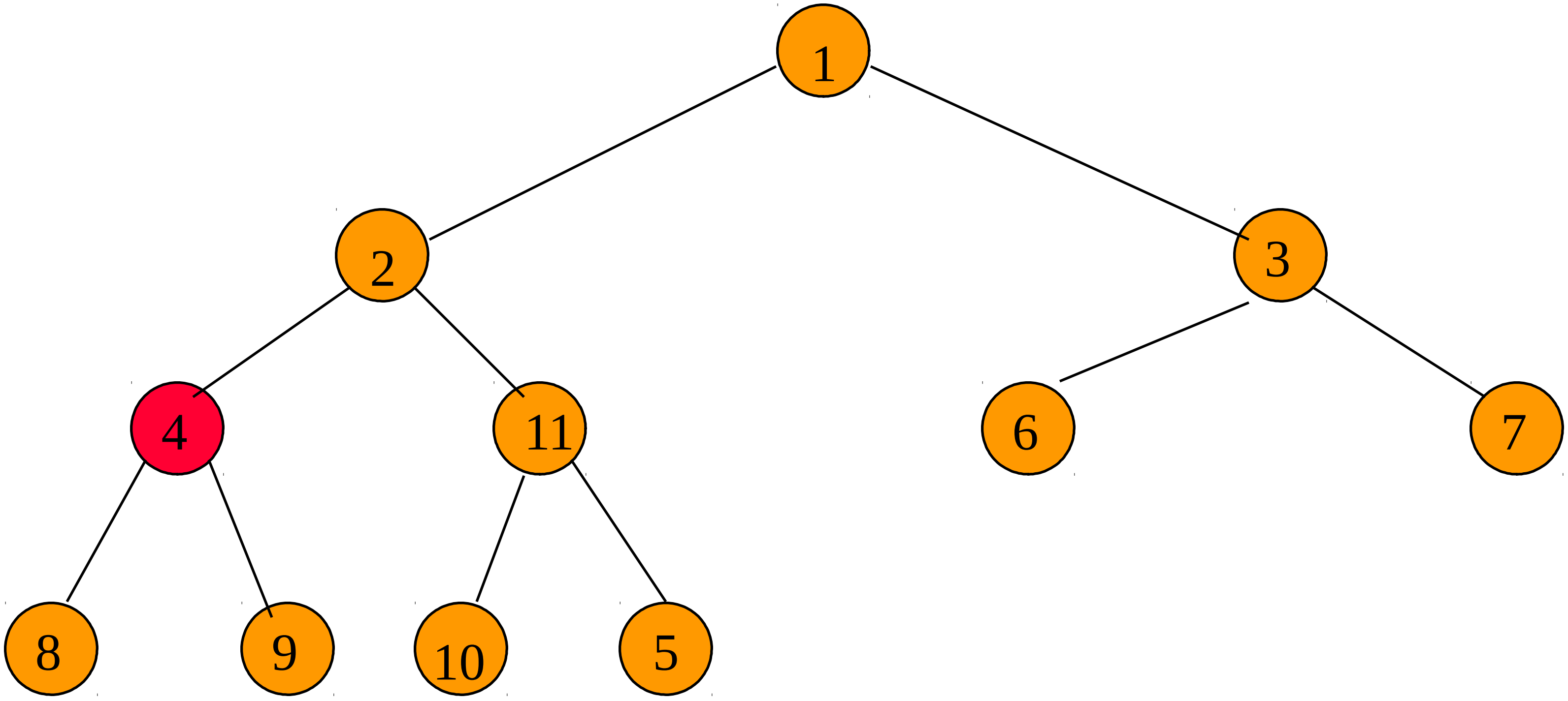
Index is n/2.

Initializing A Max Heap

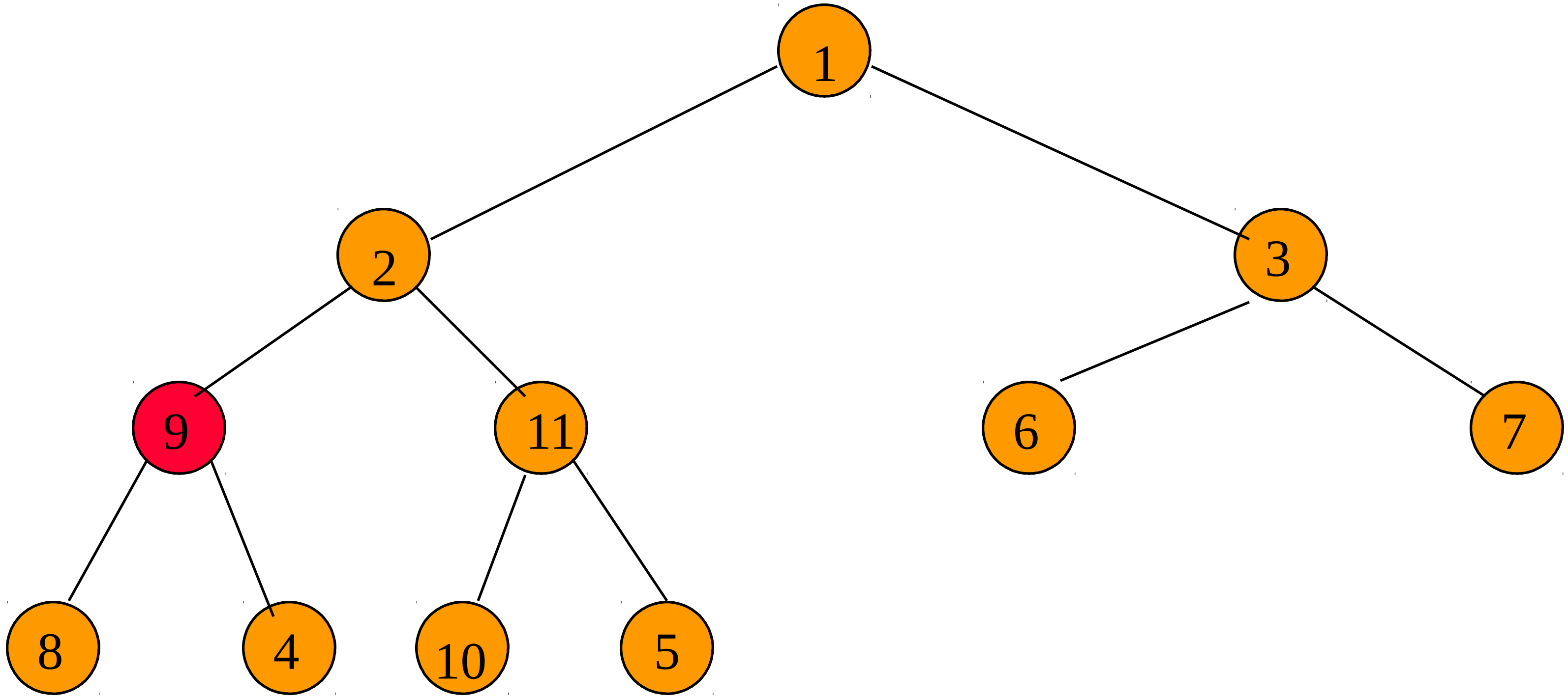


Move to next lower array position.

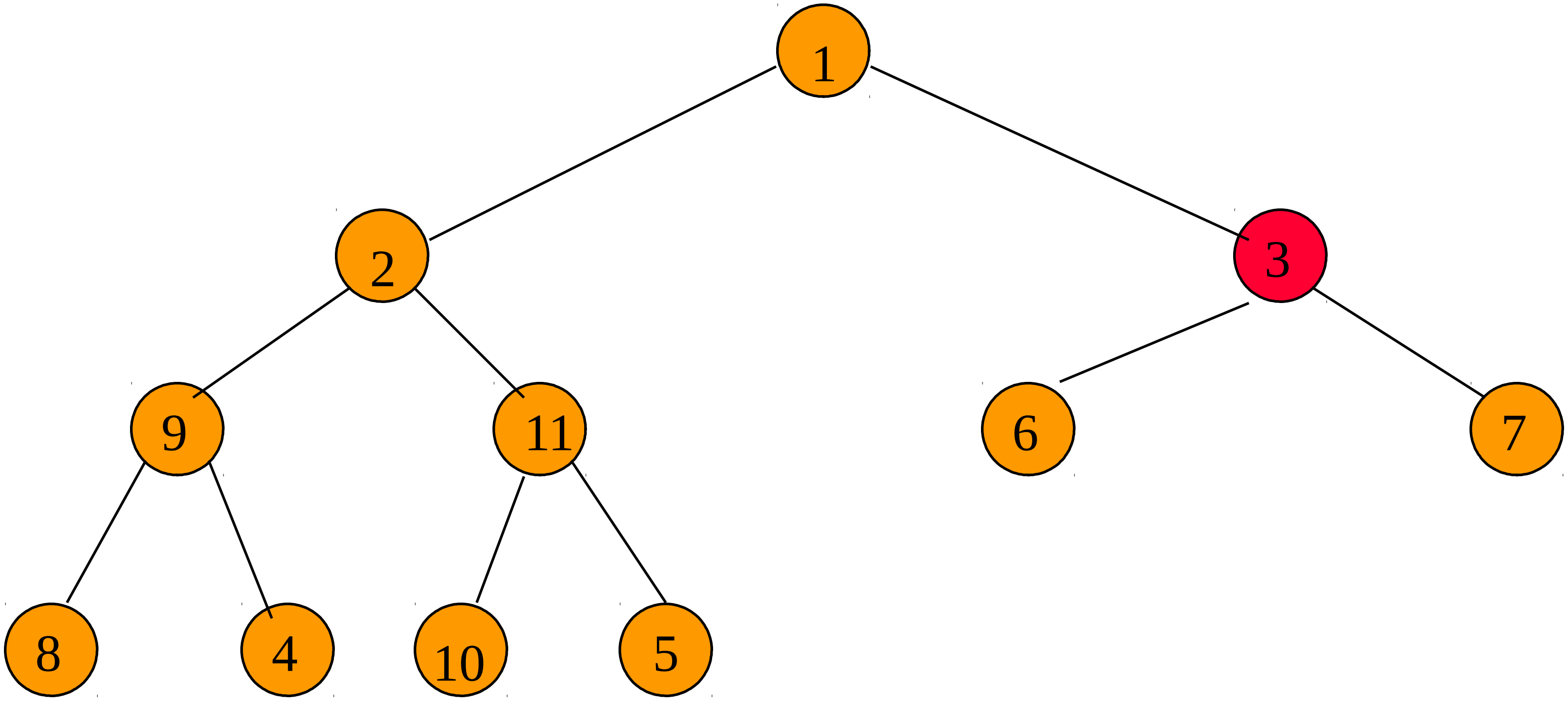
Initializing A Max Heap



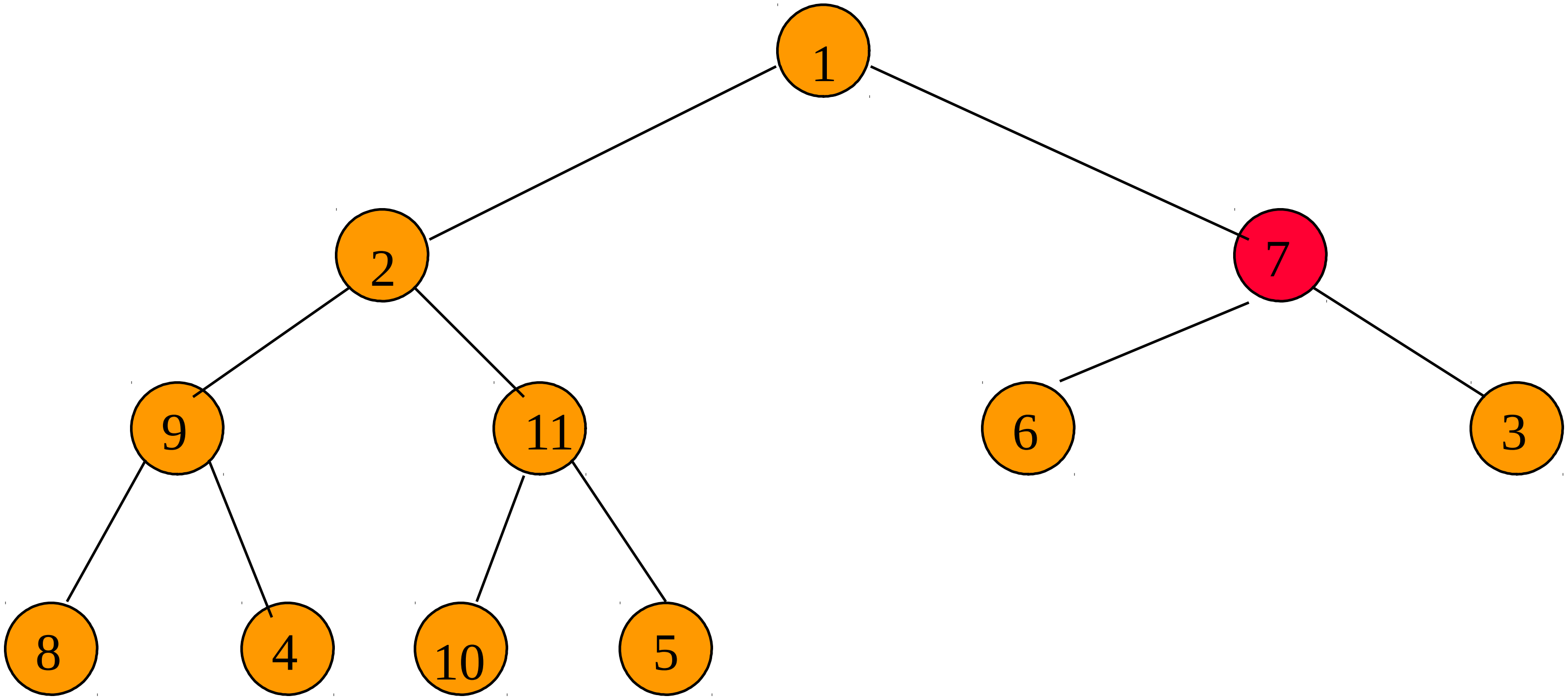
Initializing A Max Heap



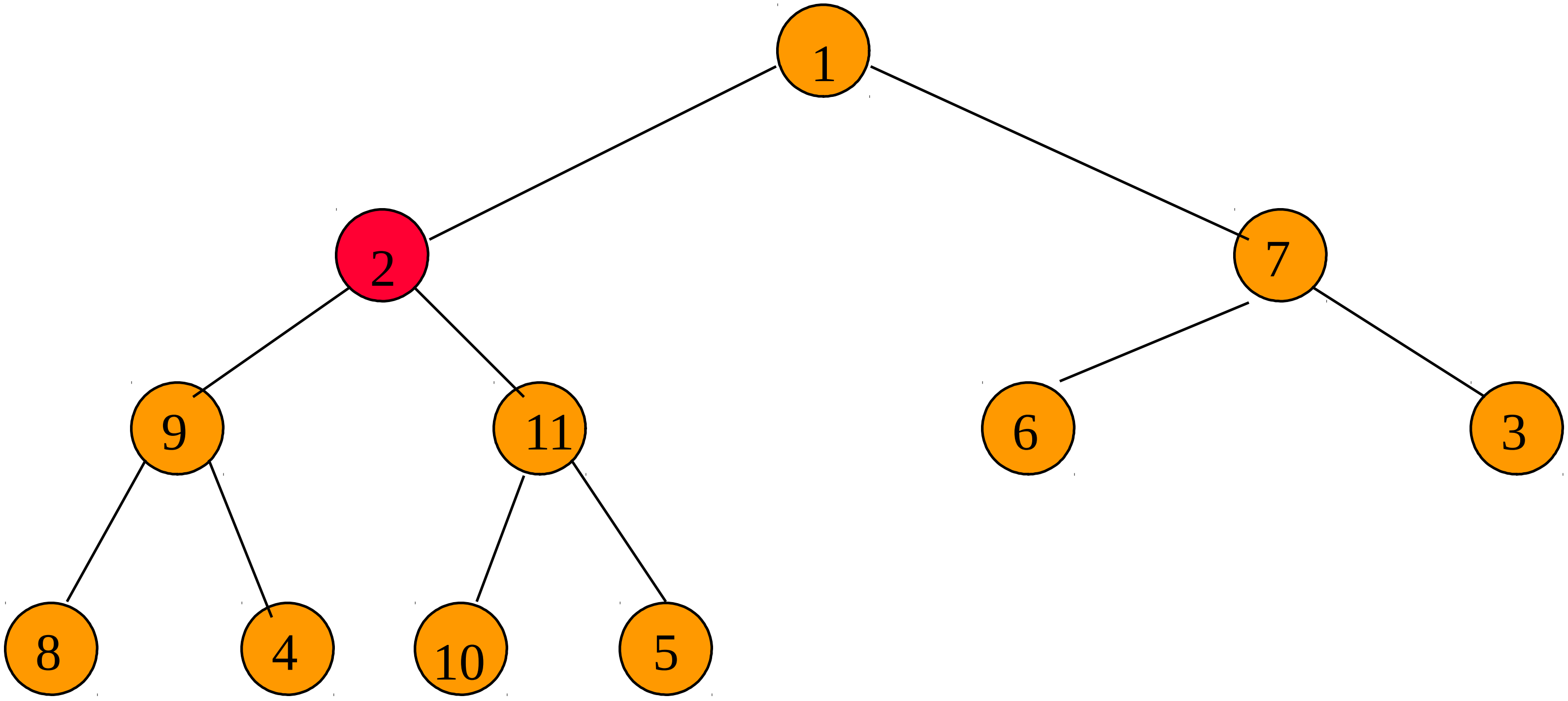
Initializing A Max Heap



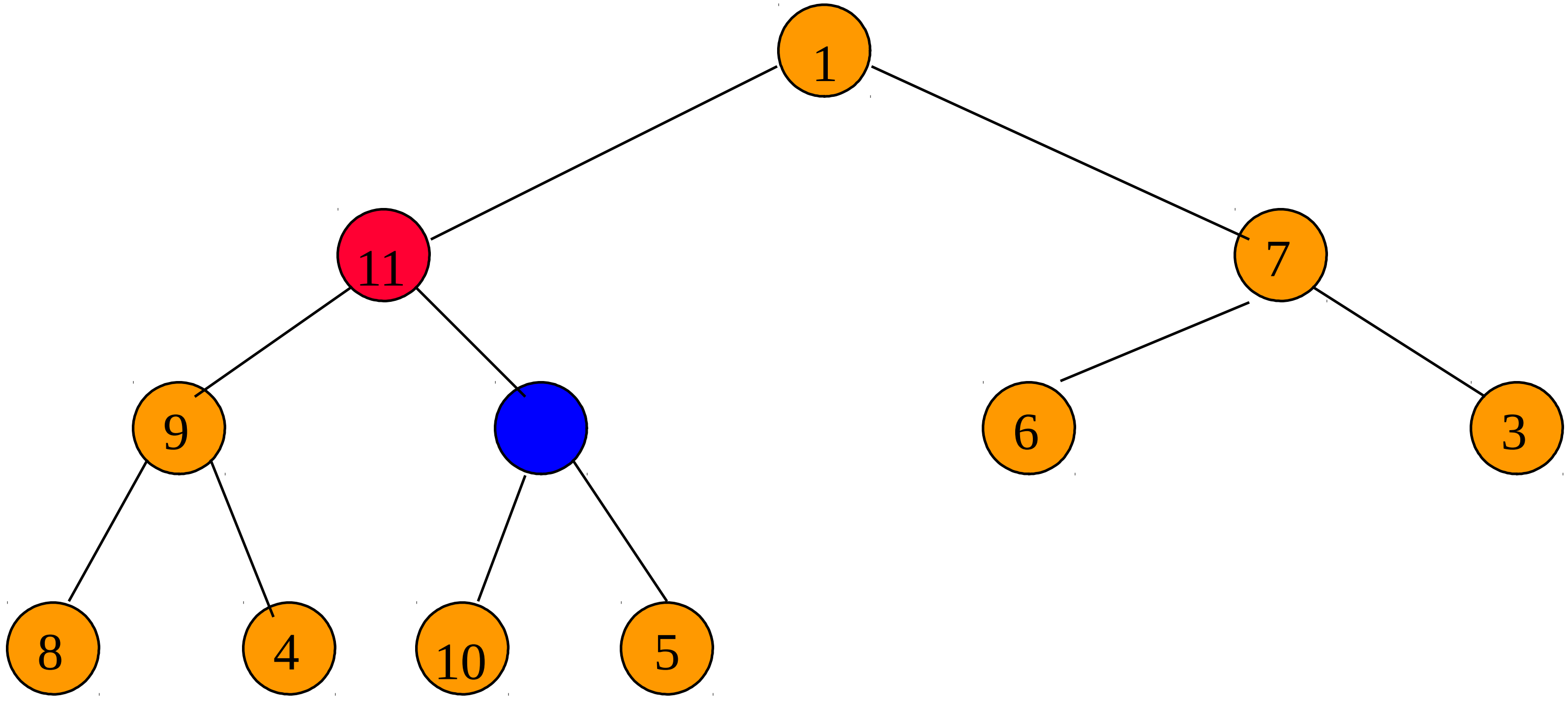
Initializing A Max Heap



Initializing A Max Heap

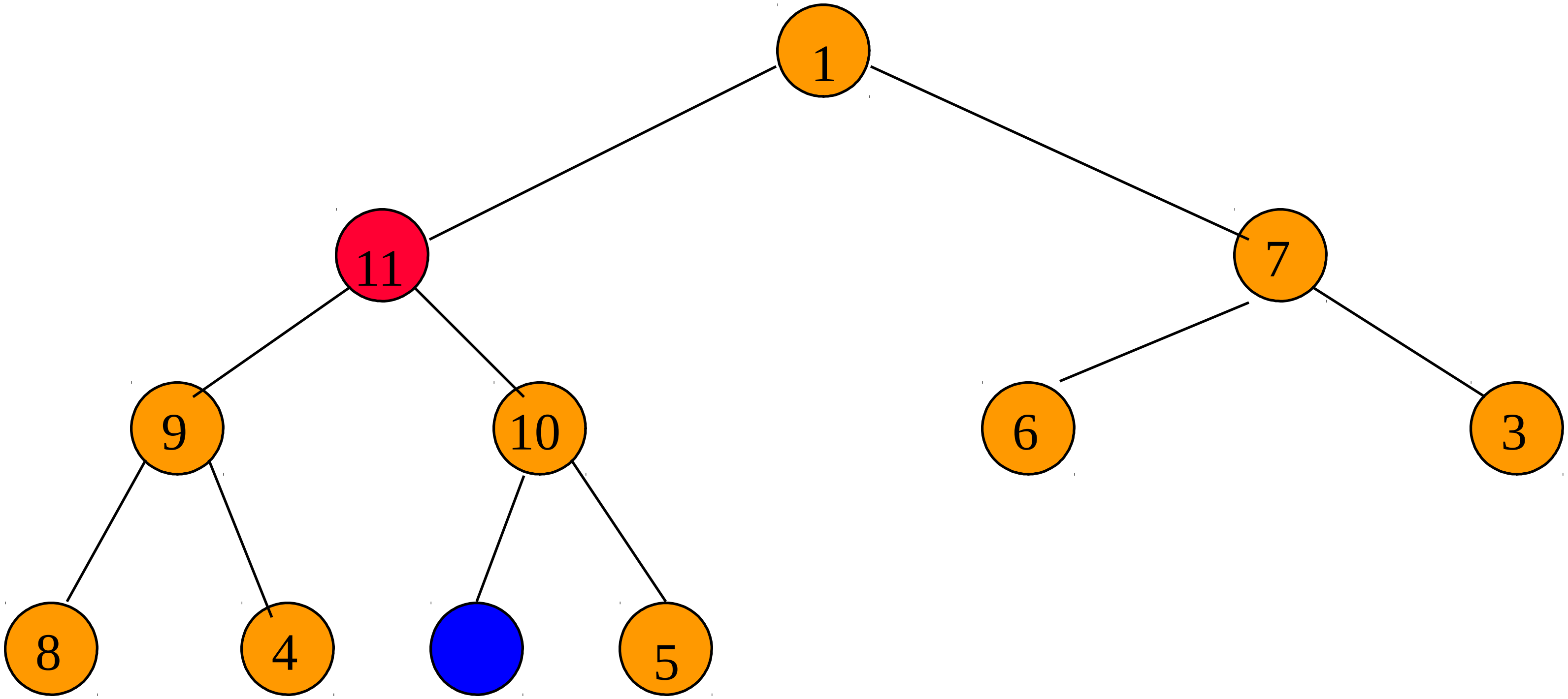


Initializing A Max Heap



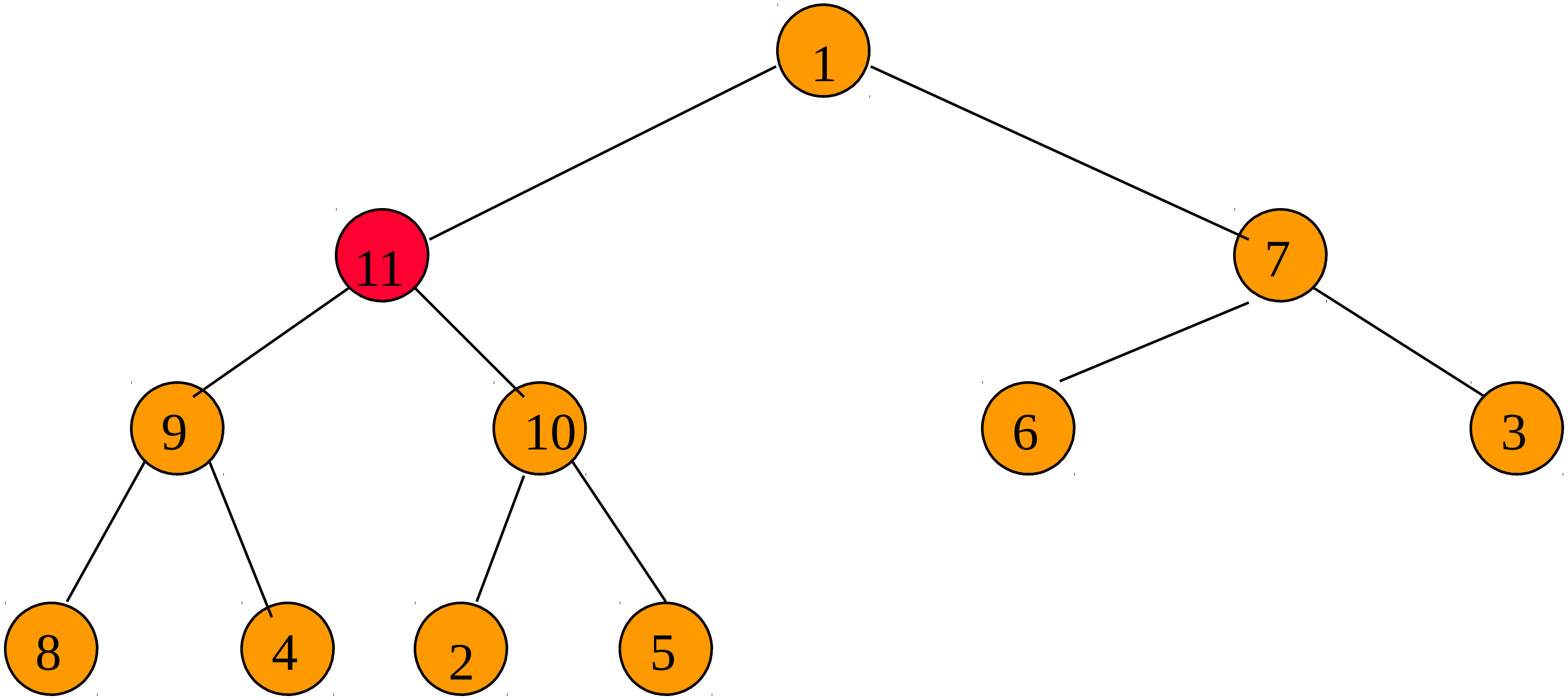
Find a home for 2.

Initializing A Max Heap



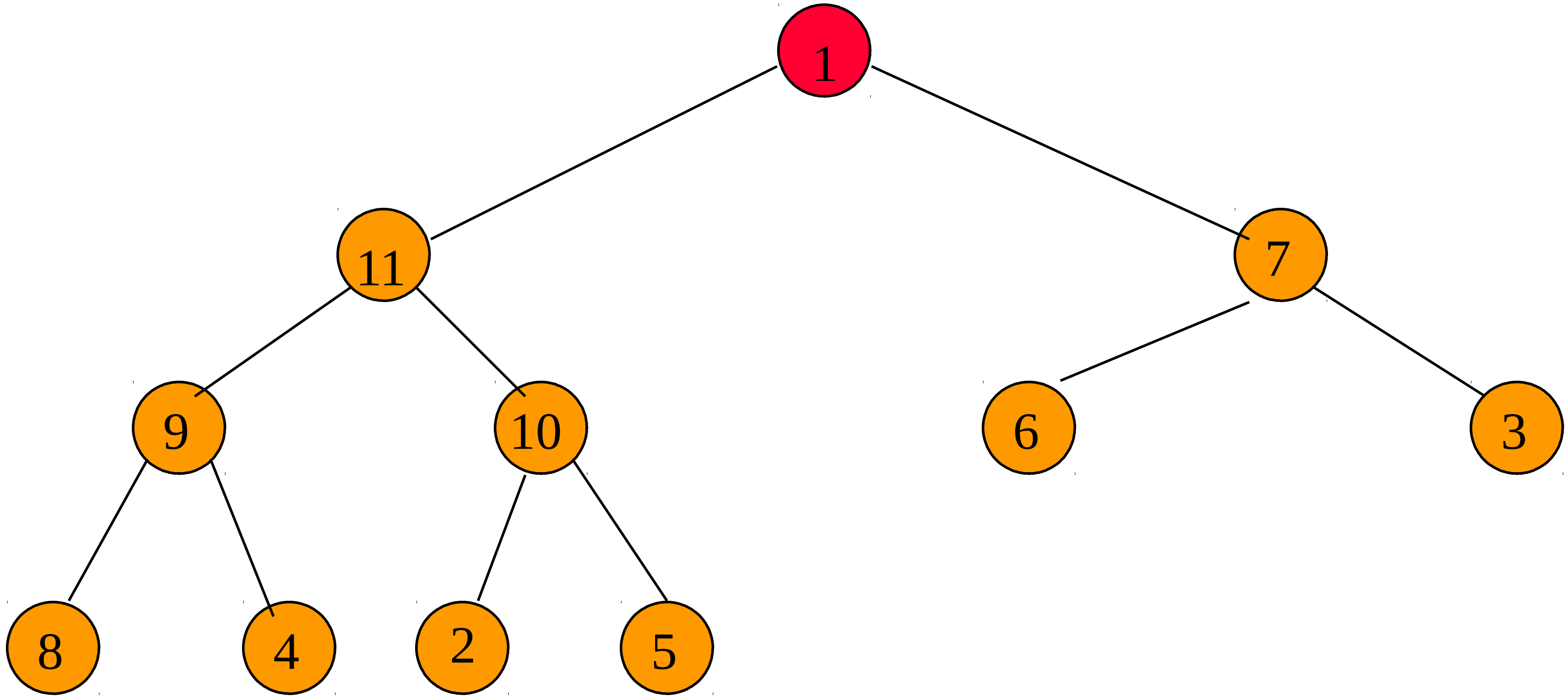
Find a home for 2.

Initializing A Max Heap



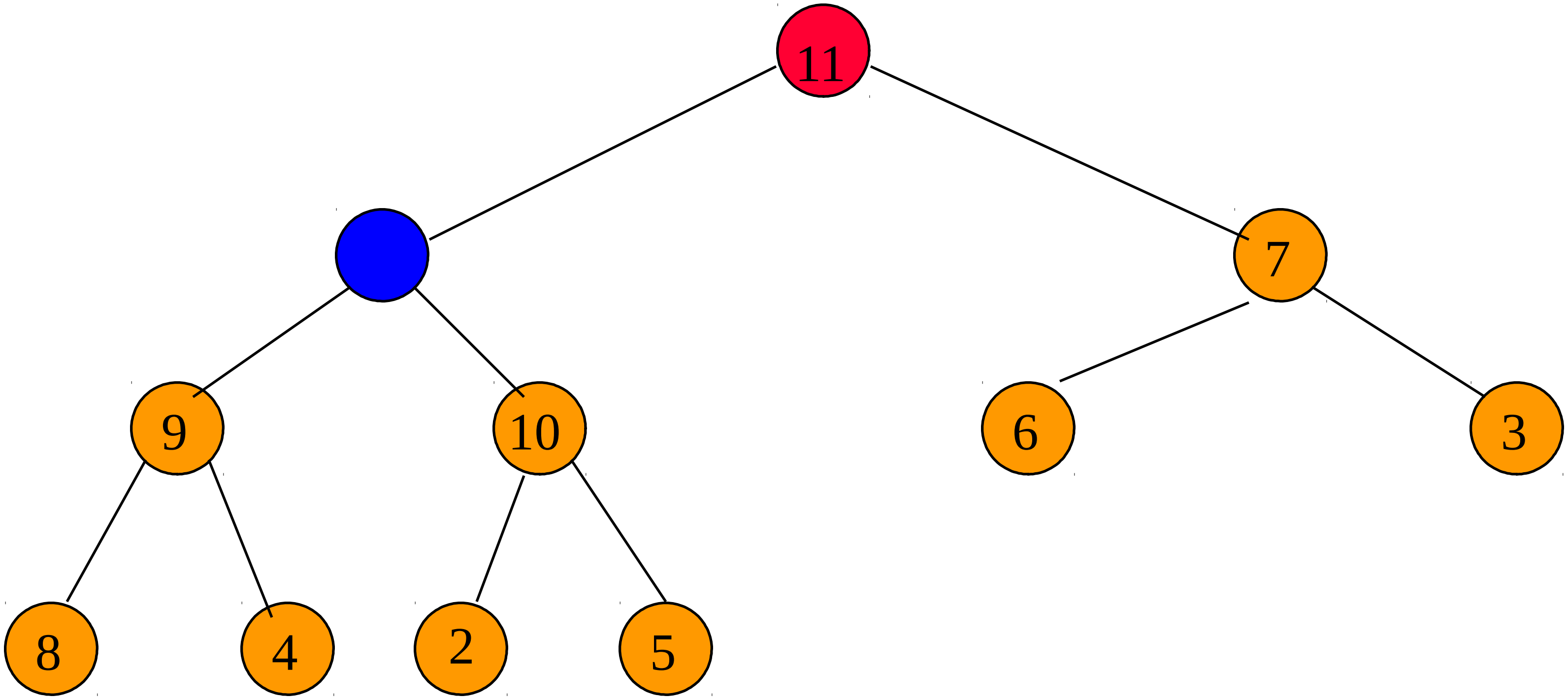
Done, move to next lower array position.

Initializing A Max Heap



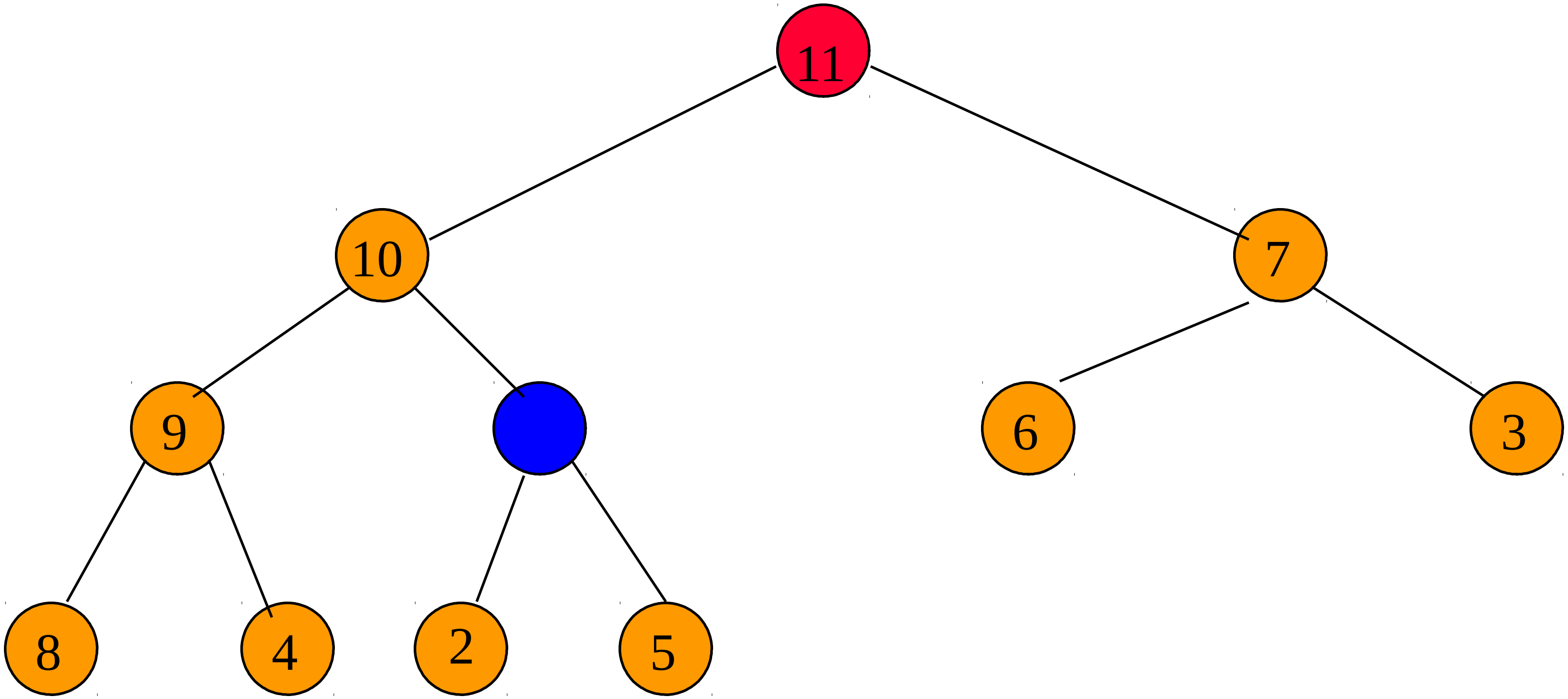
Find home for 1.

Initializing A Max Heap



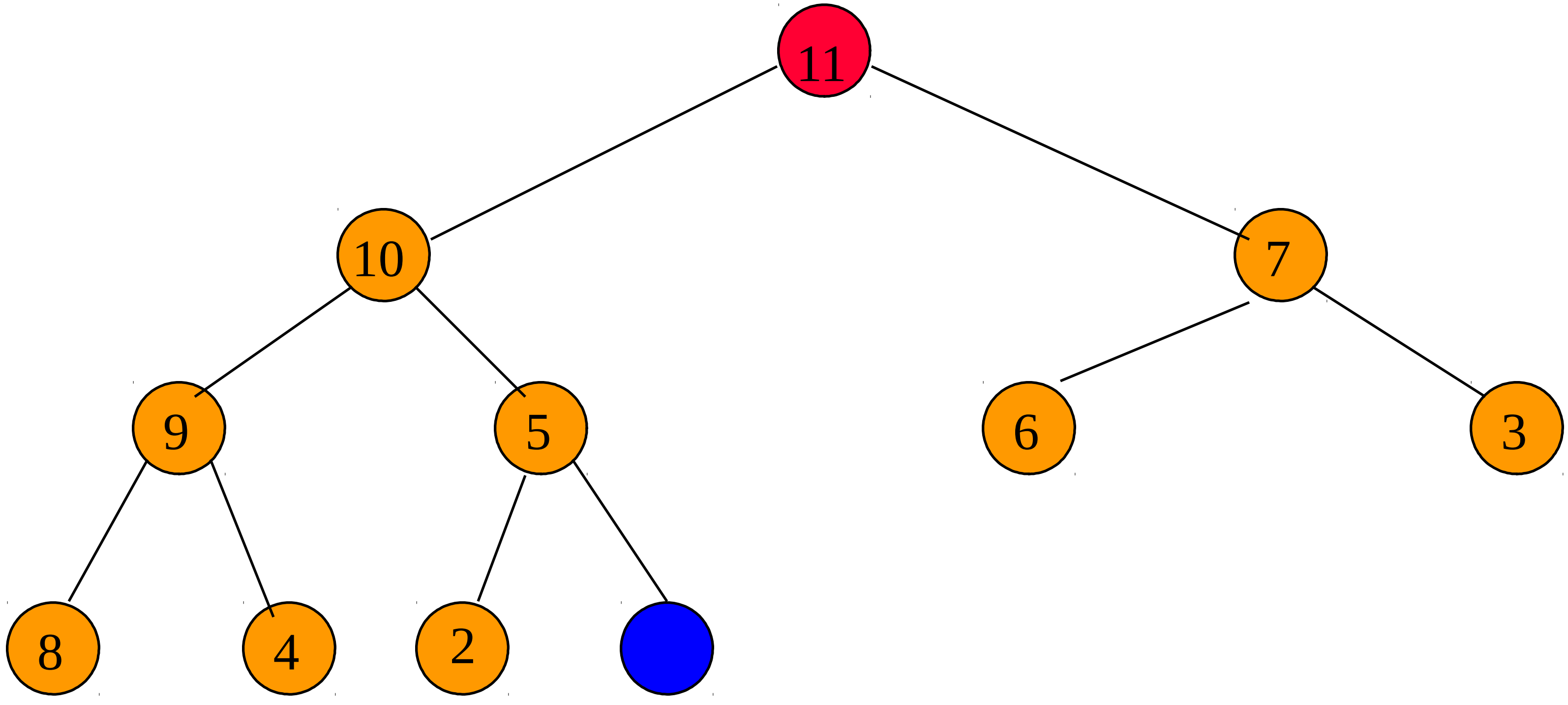
Find home for 1.

Initializing A Max Heap



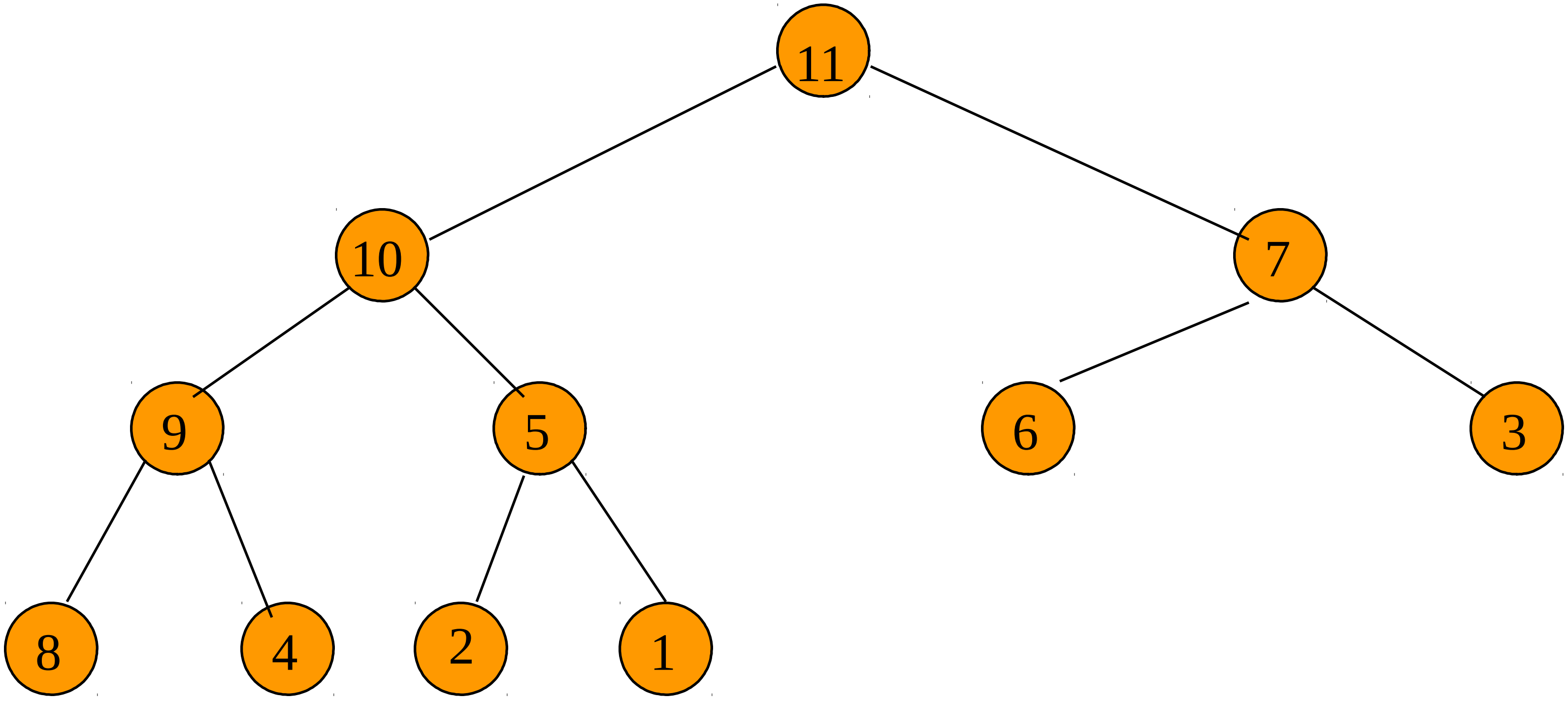
Find home for 1.

Initializing A Max Heap



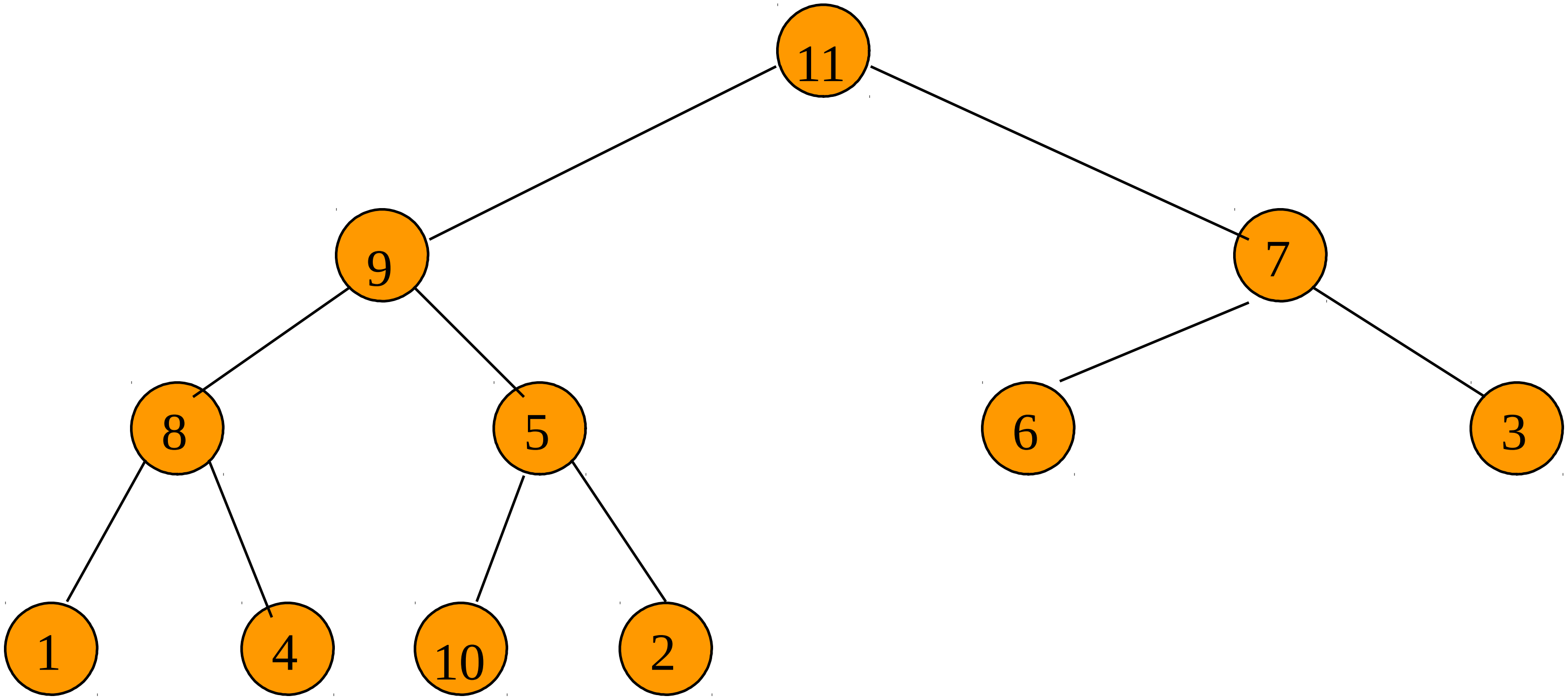
Find home for 1.

Initializing A Max Heap



Done.

Time Complexity



Height of heap = h.

Number of subtrees with root at level j is <= 2^{j-1}.

Time for each subtree is O(h-j+1).

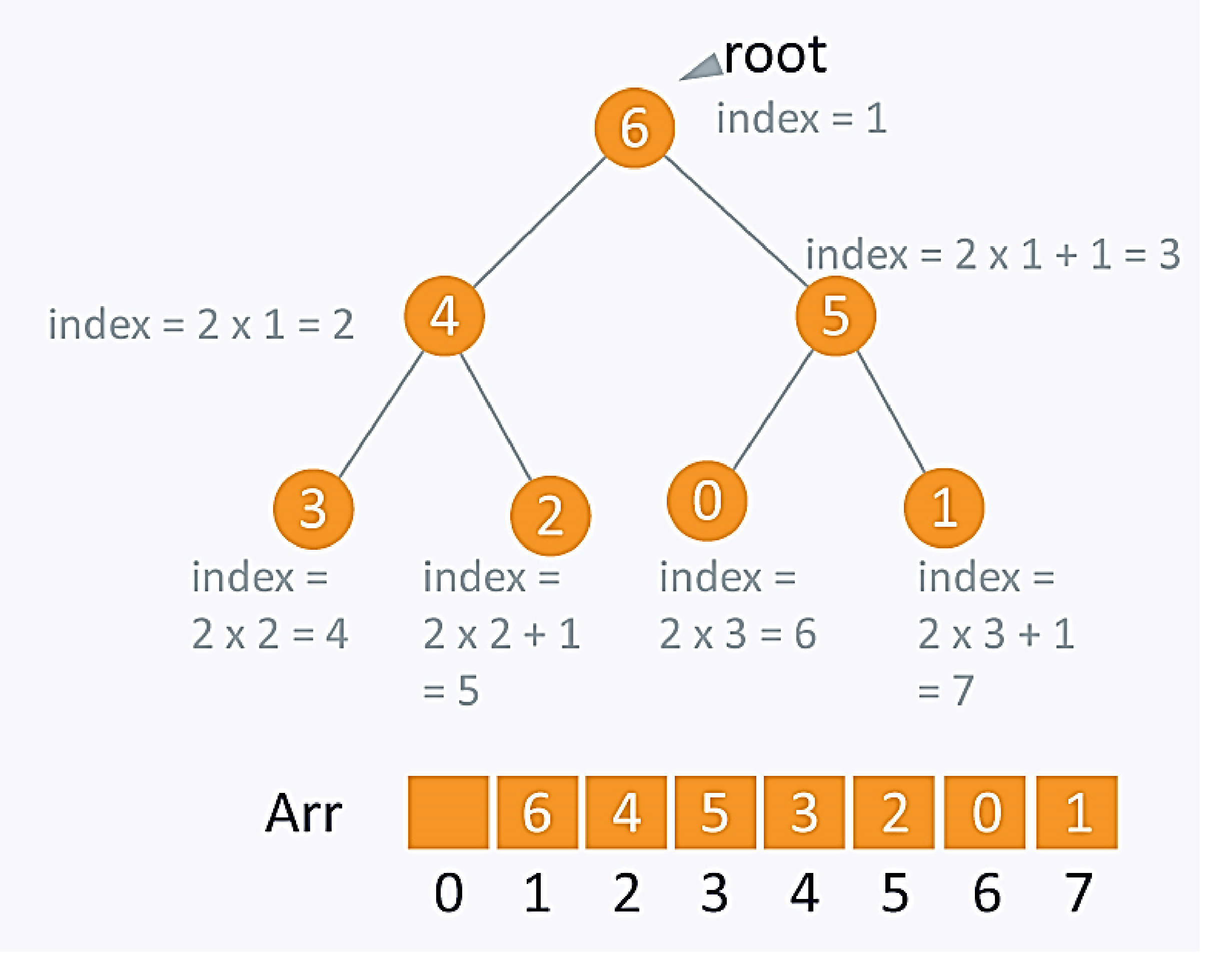
Complexity

Time for level j subtrees is $\le 2^{j-1}(h-j+1) = t(j)$.

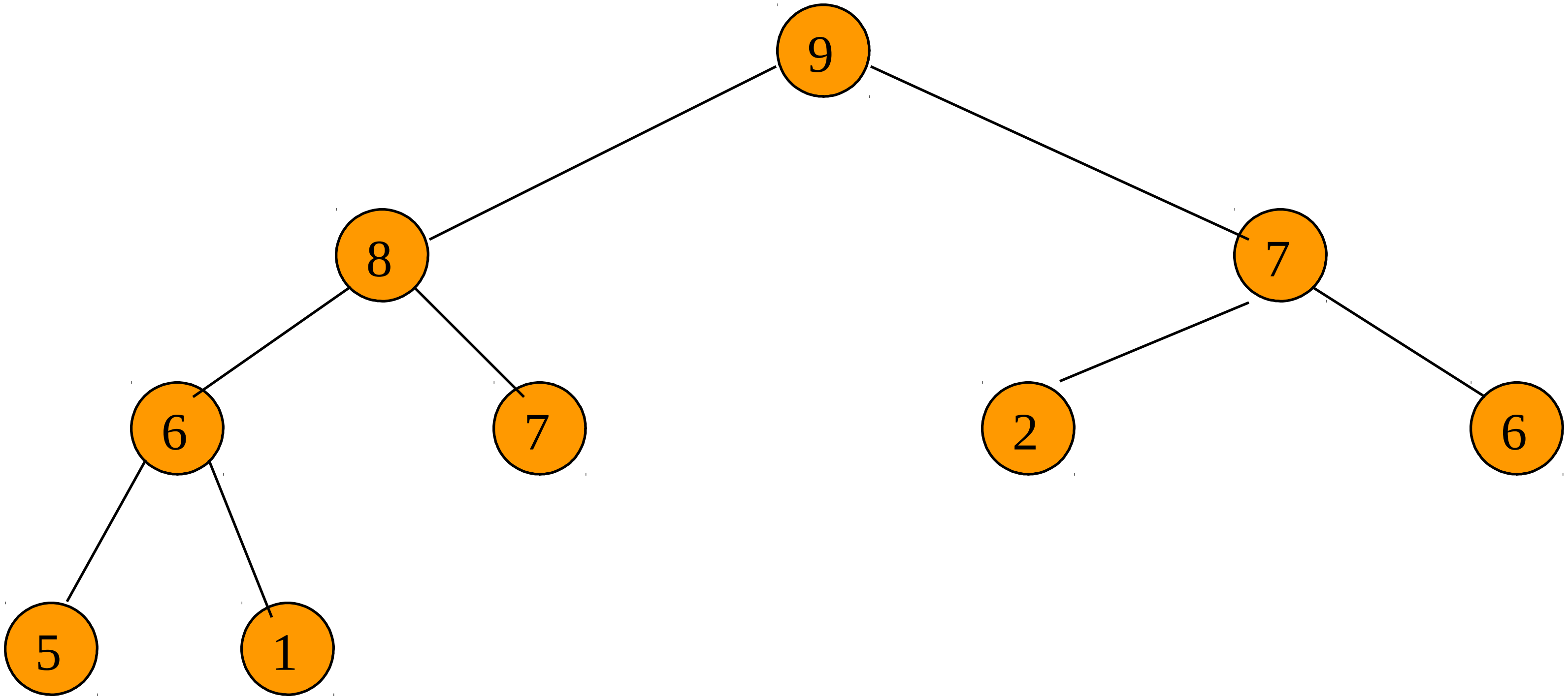
Total time is $t(1) + t(2) + \ldots + t(h-1) = O(n)$.

Heap using Array

If we are storing one element at index ‘i’ in array Arr , then its parent will be stored at index ‘i/2’ (unless its a root, as root has no parent) and can be accessed by Arr[i/2], and its left child can be accessed by Arr[2*i] and its right child can be accessed by Arr[2*i+1]. Index of root will be 1 in an array.

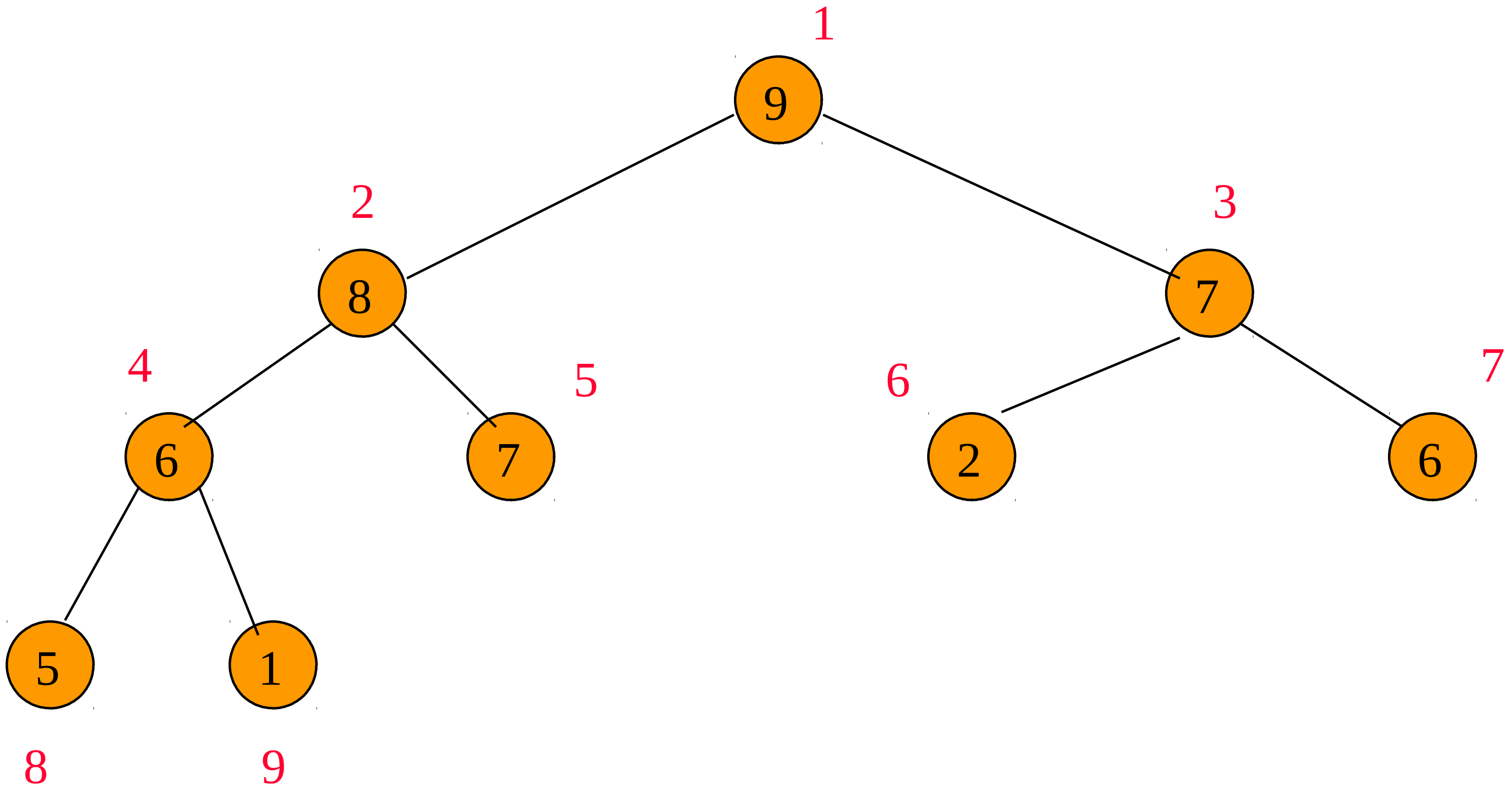


A Heap Is Efficiently Represented As An Array



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

Moving Up And Down A Heap

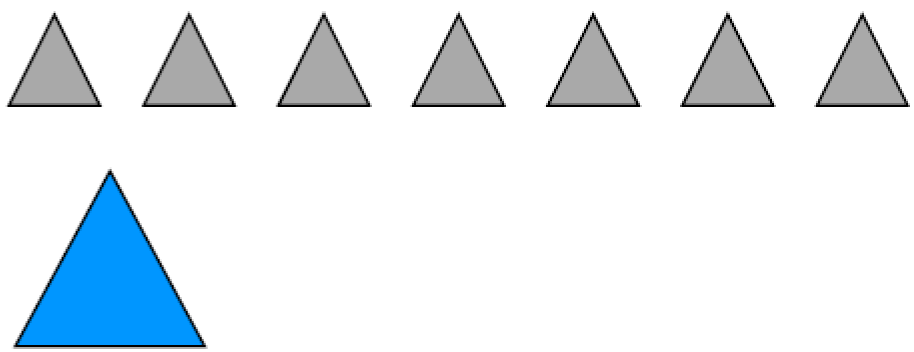


Heapify

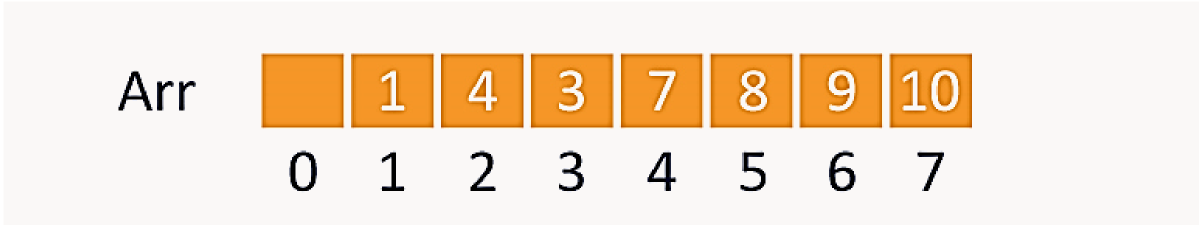
heapify(L): given a list of heaps H_1, H_2, ..., H_k, return a new heap that contains the union of keys in all of them.

(As usual, we're allowed to destroy each H_i and the list.)

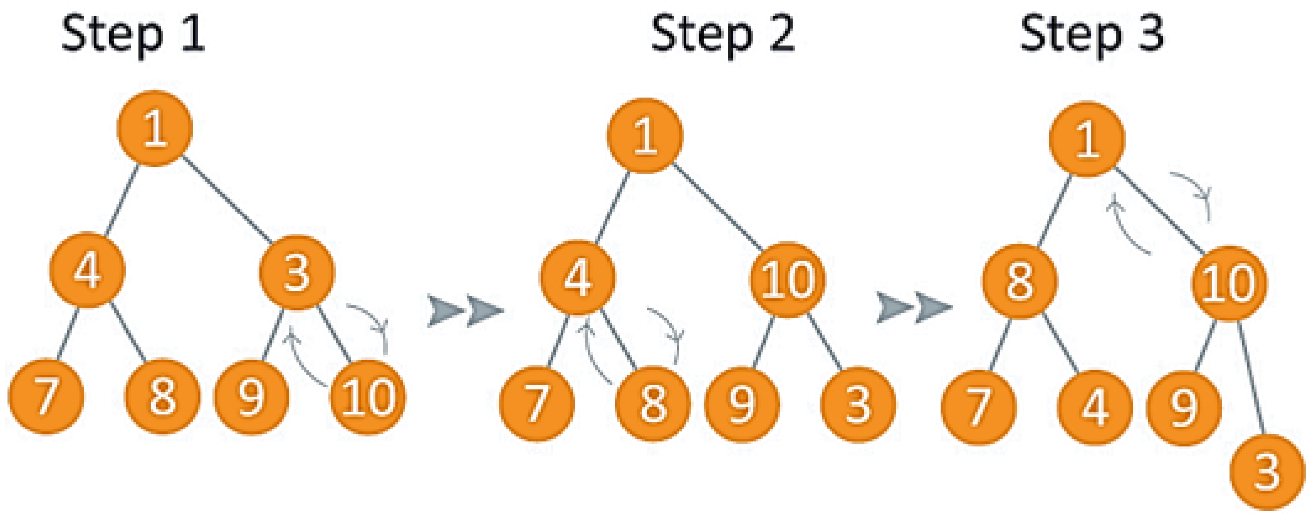
Treat L as a queue
Repeat until only 1 heap left:
1. *meld* the front two items
2. *enqueue* the resulting heap:

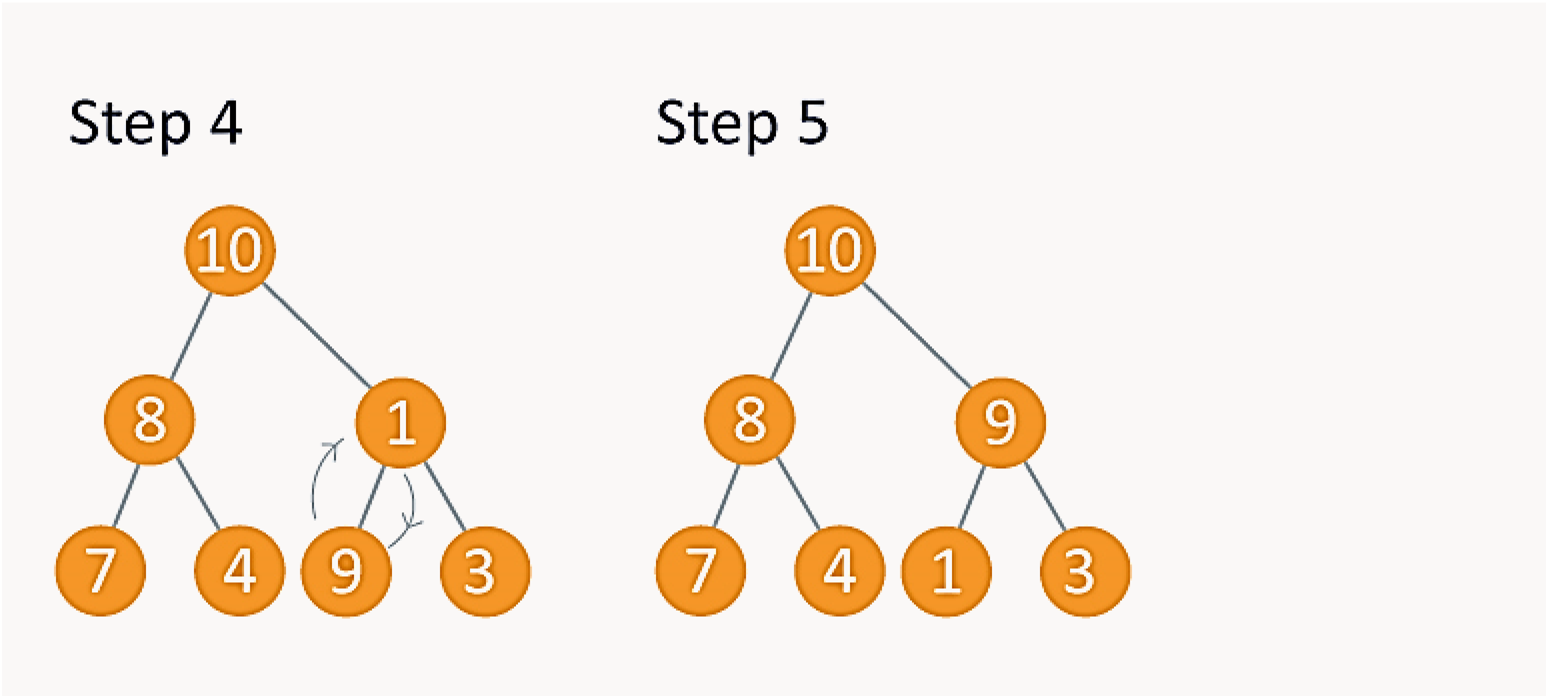
L = []

Heapify an Array - Example



Suppose the Array Arr is given. We construct the Max Heap as follows.





Final Array

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

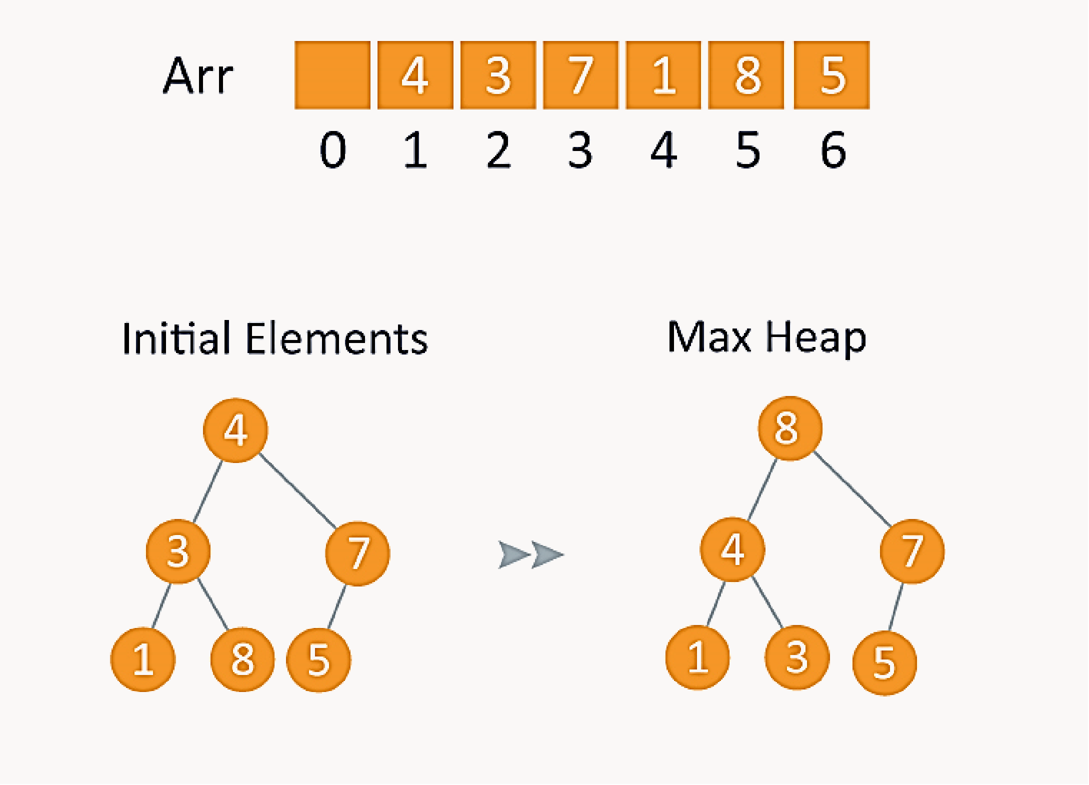
Heap Sort

Uses a max priority queue that is implemented as a heap.

Initial insert operations are replaced by a heap initialization step that takes O(n) time.

Using Heap to sort the array - Example

Initially there is an unsorted array Arr having 6 elements. We begin by building max-heap.

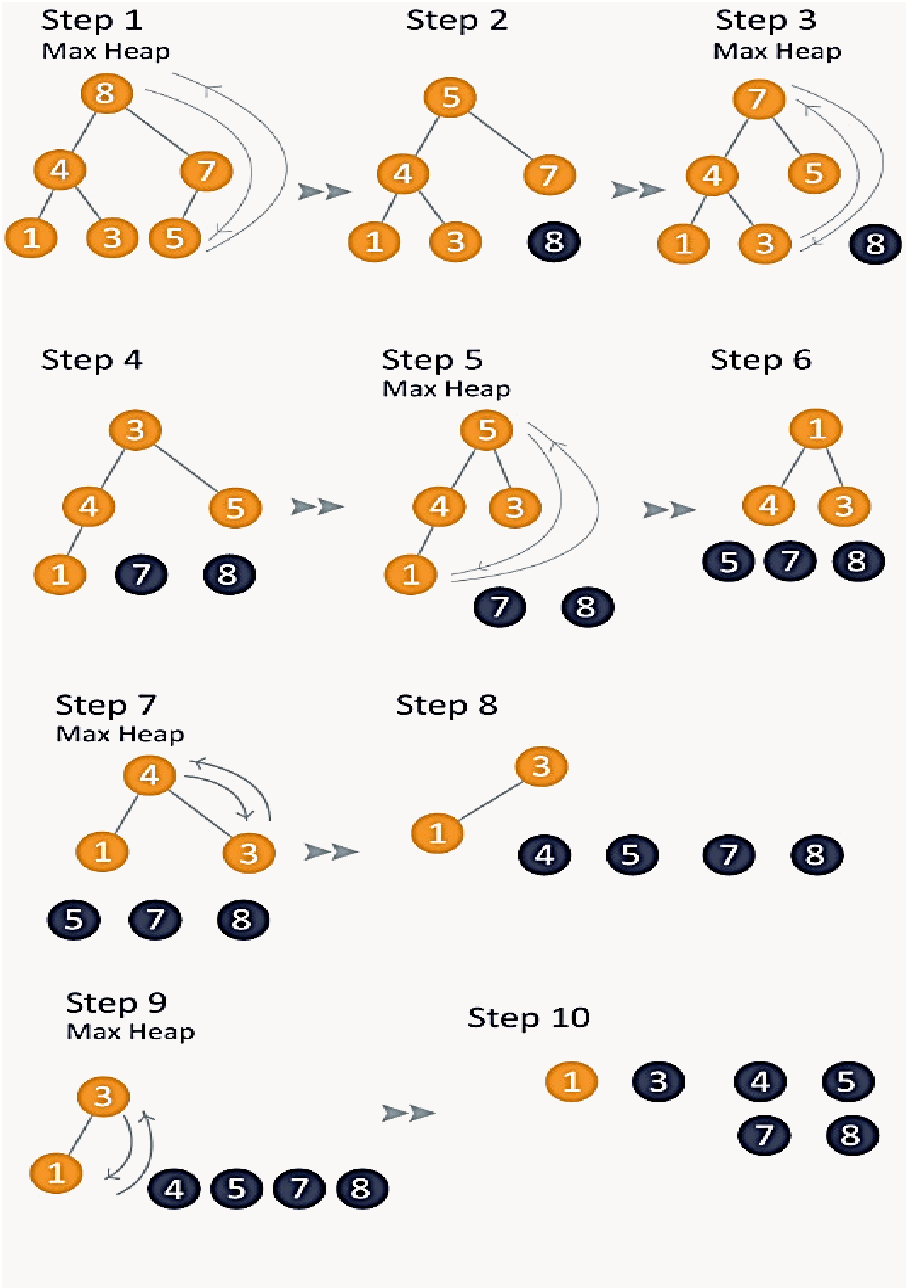


After building max-heap, the elements in the array Arr will be:

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

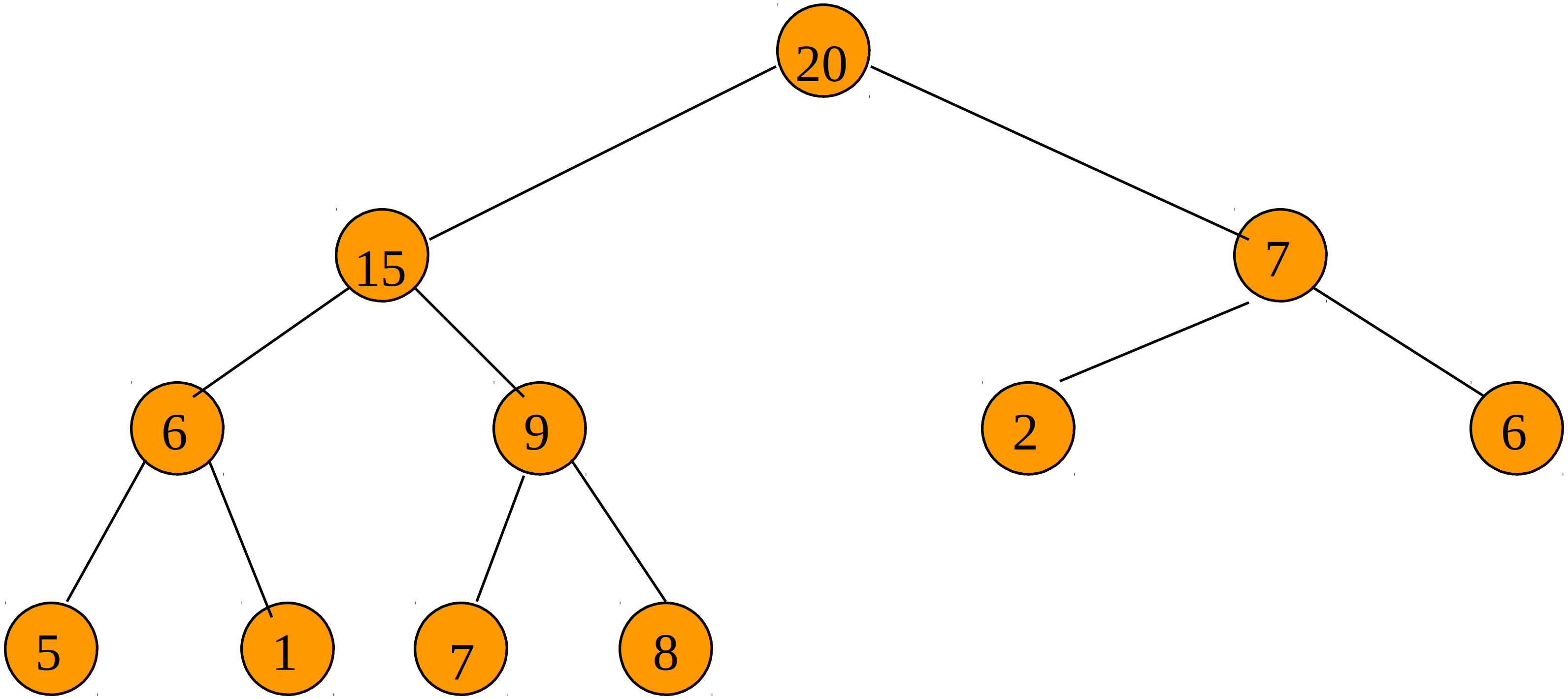
Processing:

Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.



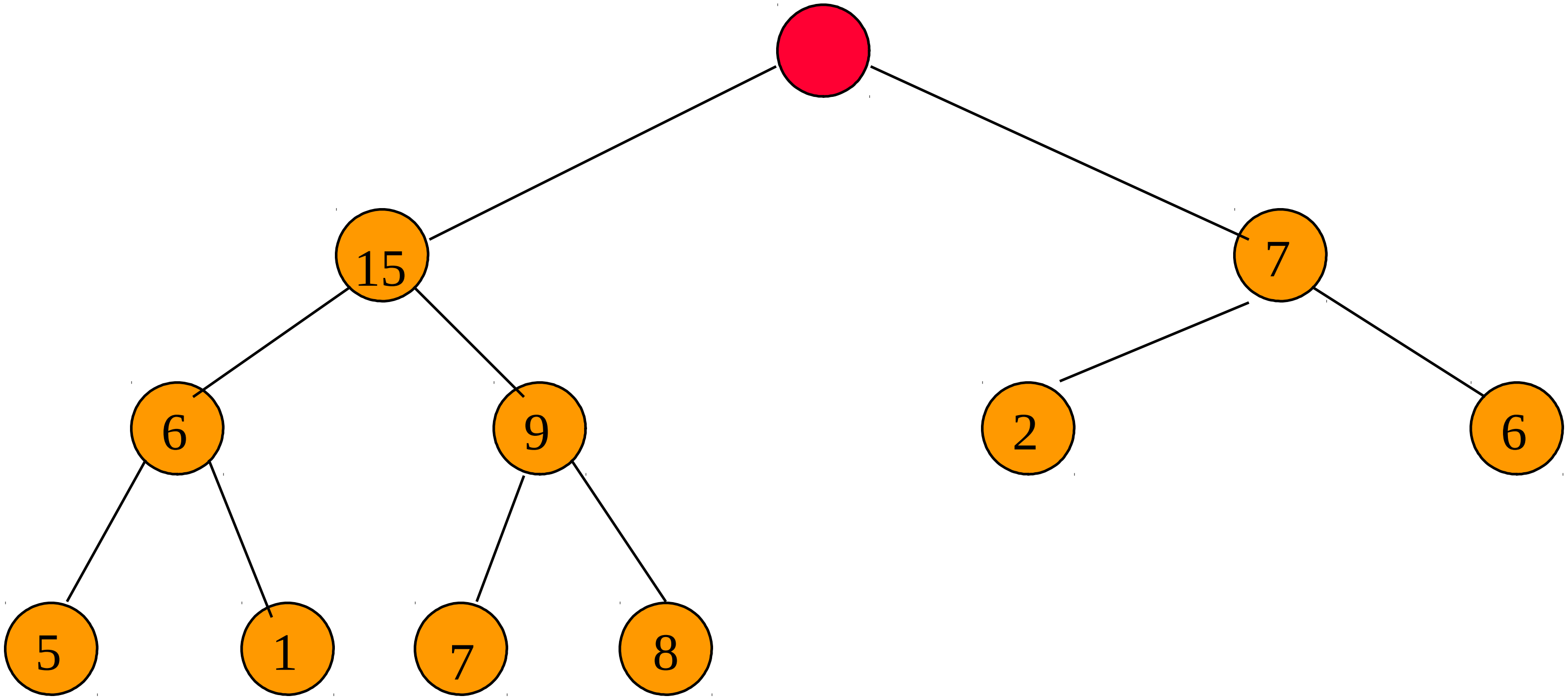
After all the steps, we will get a sorted array.

Removing The Max Element



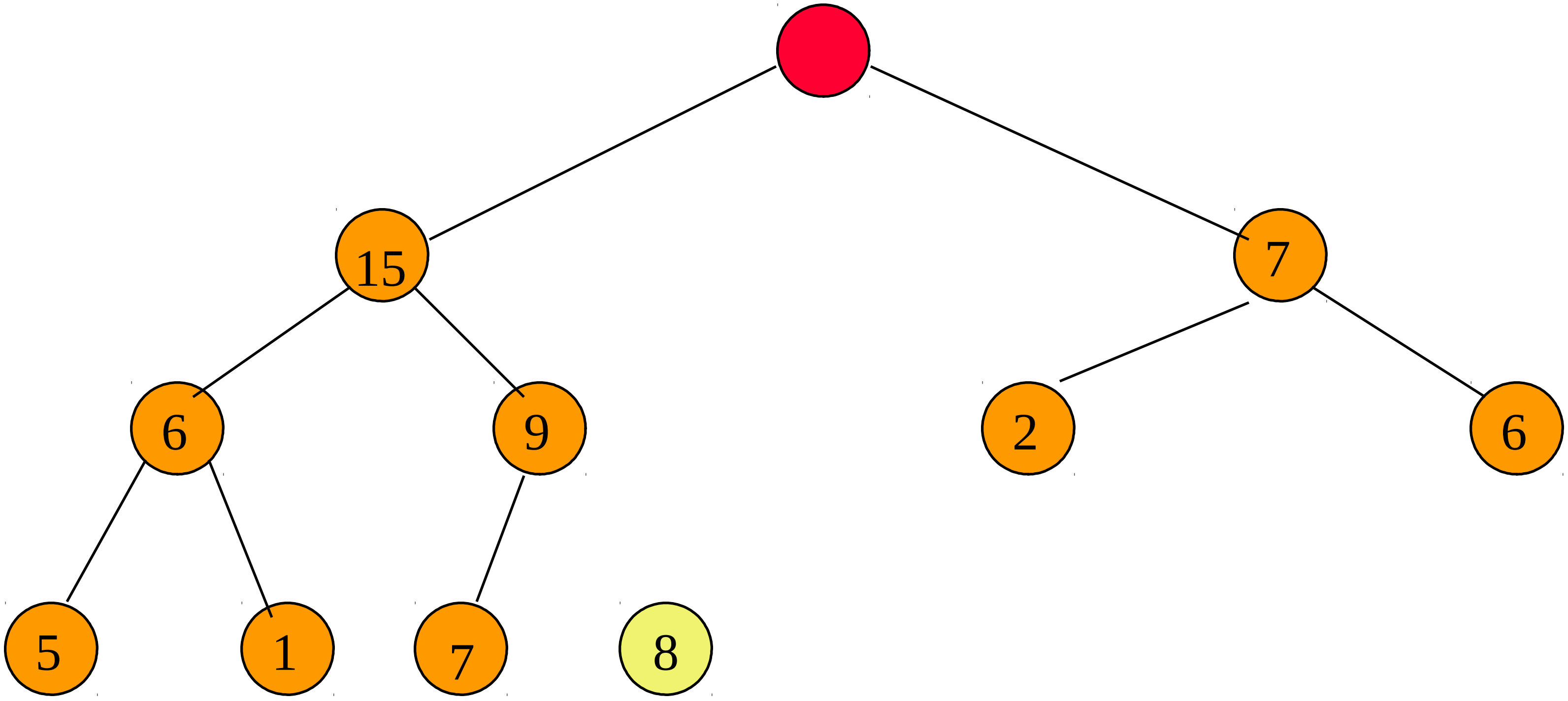
Max element is in the root.

Removing The Max Element



After max element is removed.

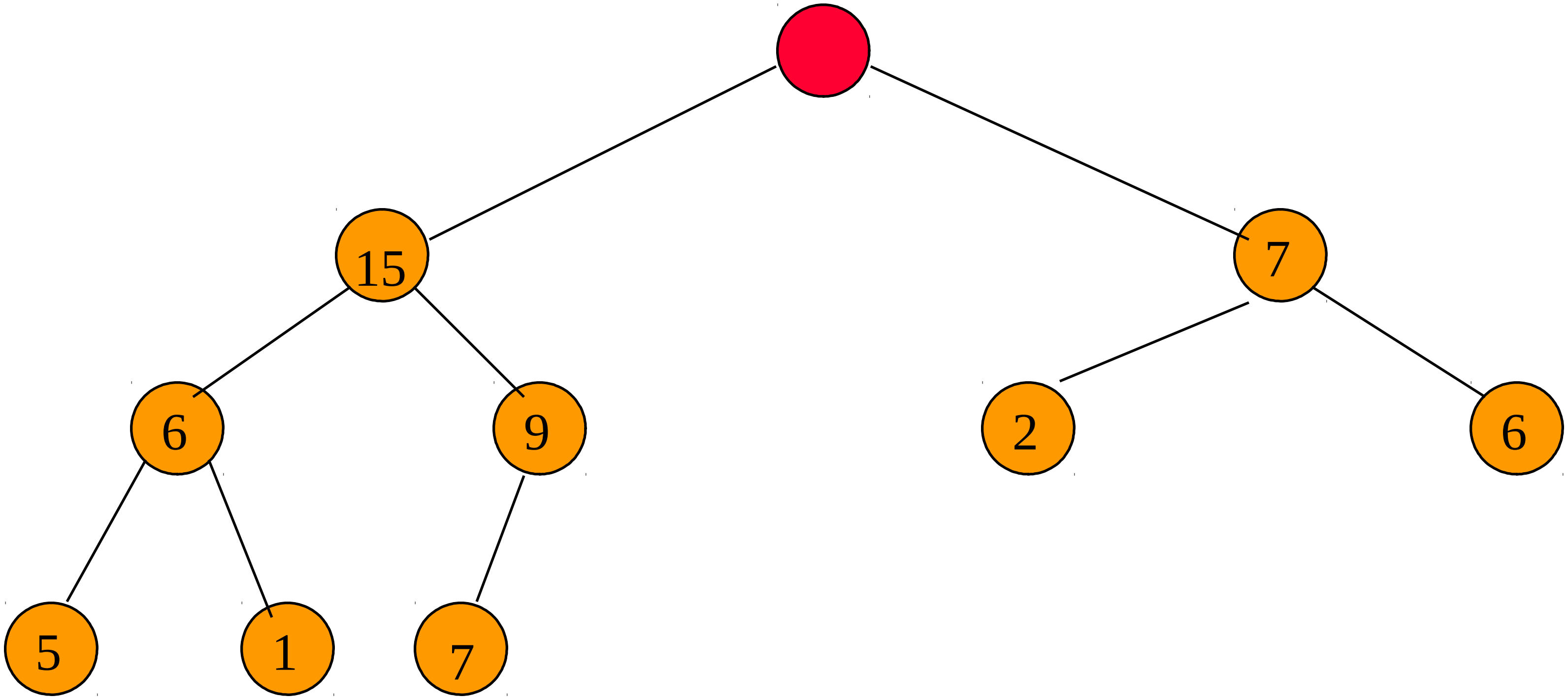
Removing The Max Element



Heap with 10 nodes.

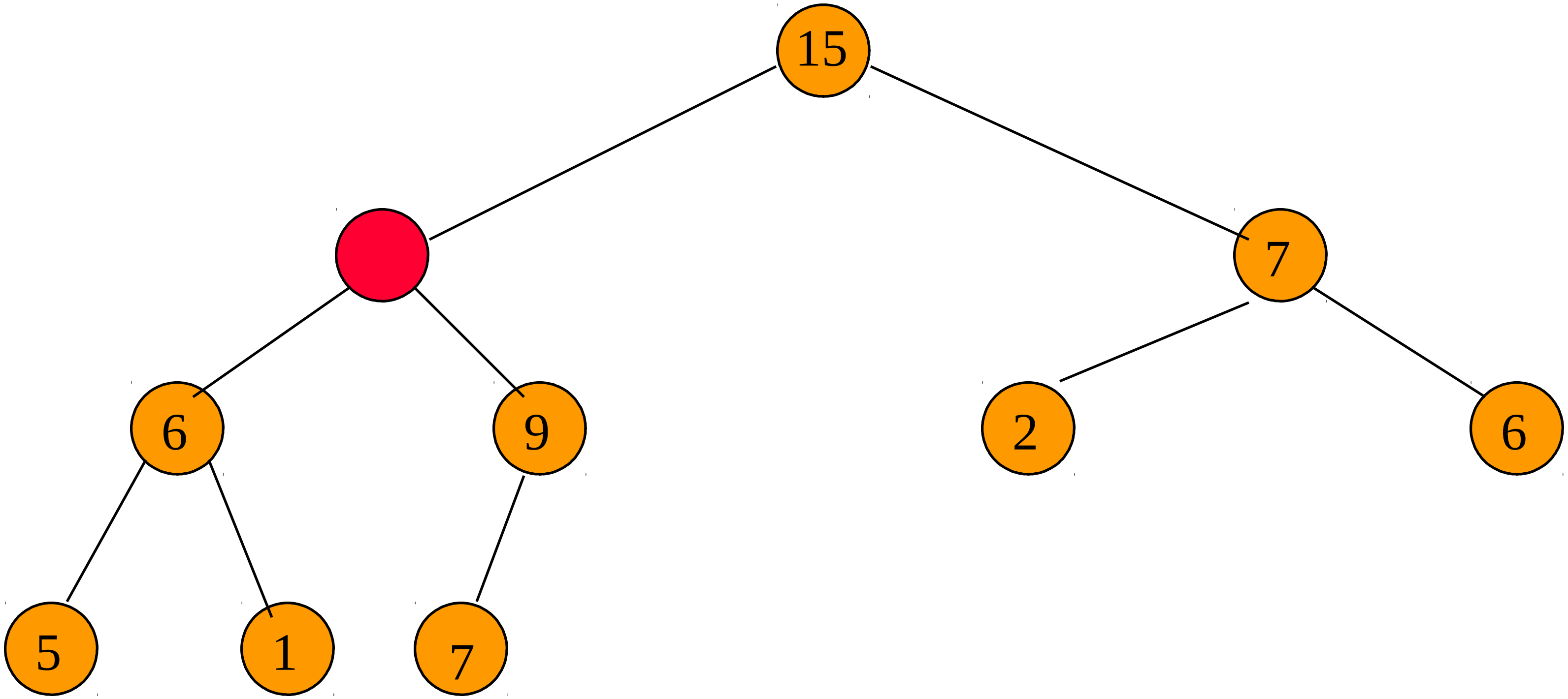
Reinsert 8 into the heap.

Removing The Max Element



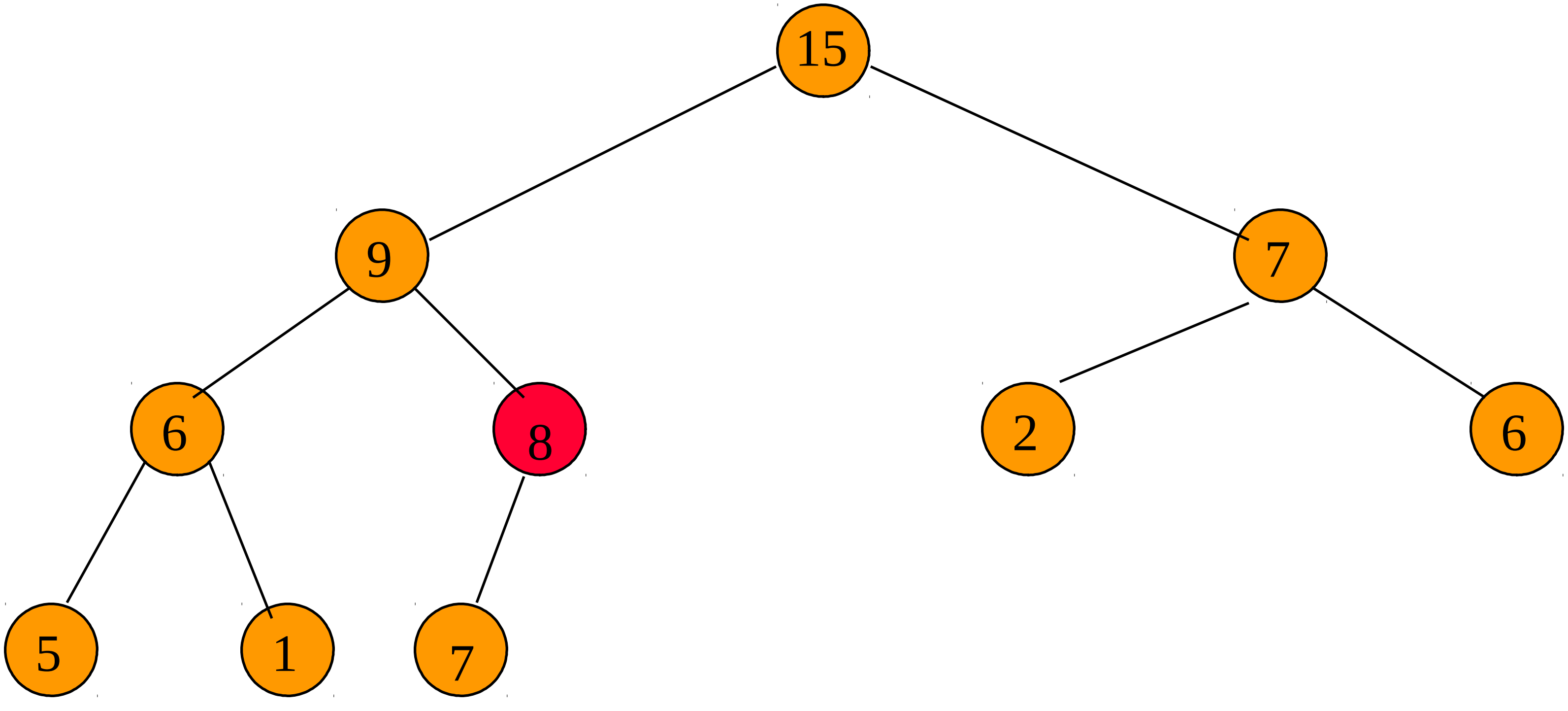
Reinsert 8 into the heap.

Removing The Max Element



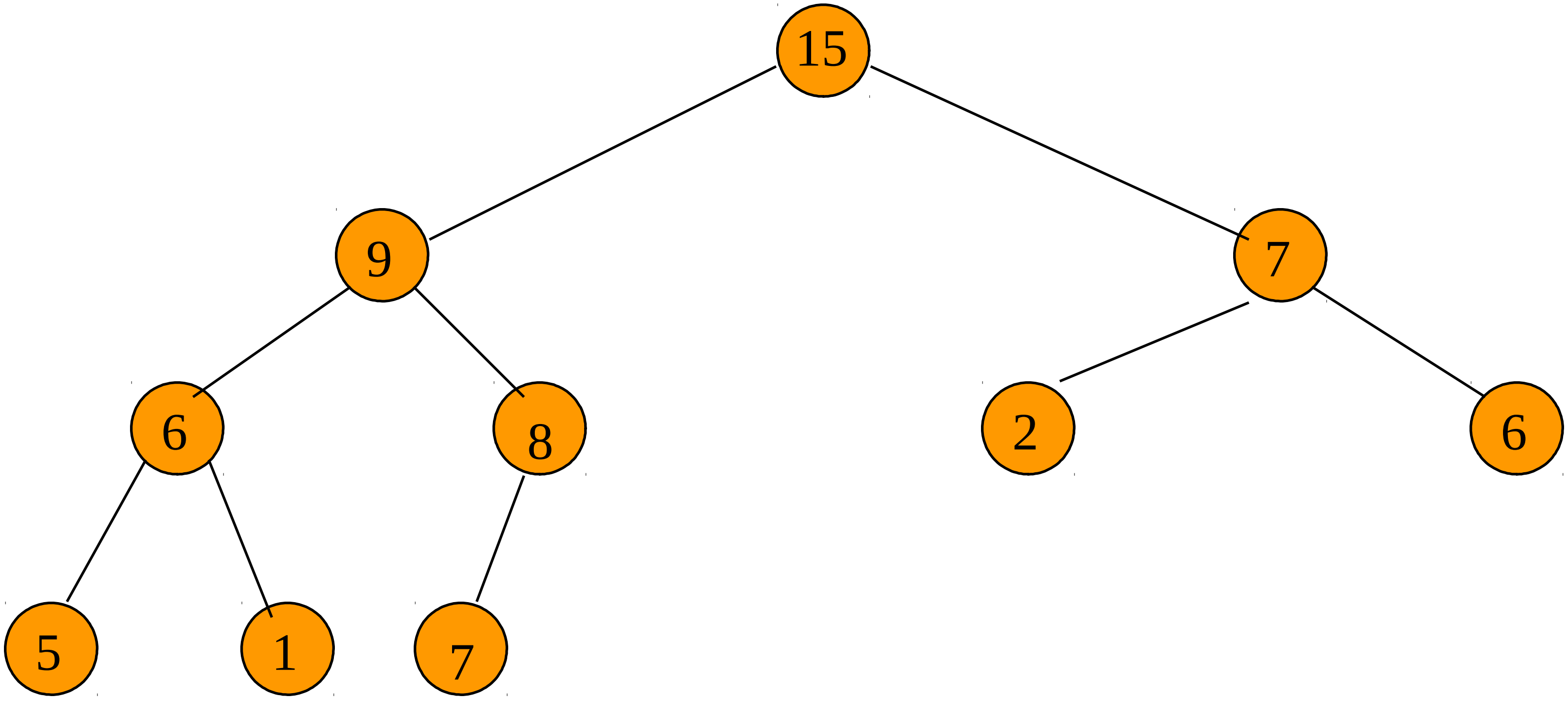
Reinsert 8 into the heap.

Removing The Max Element



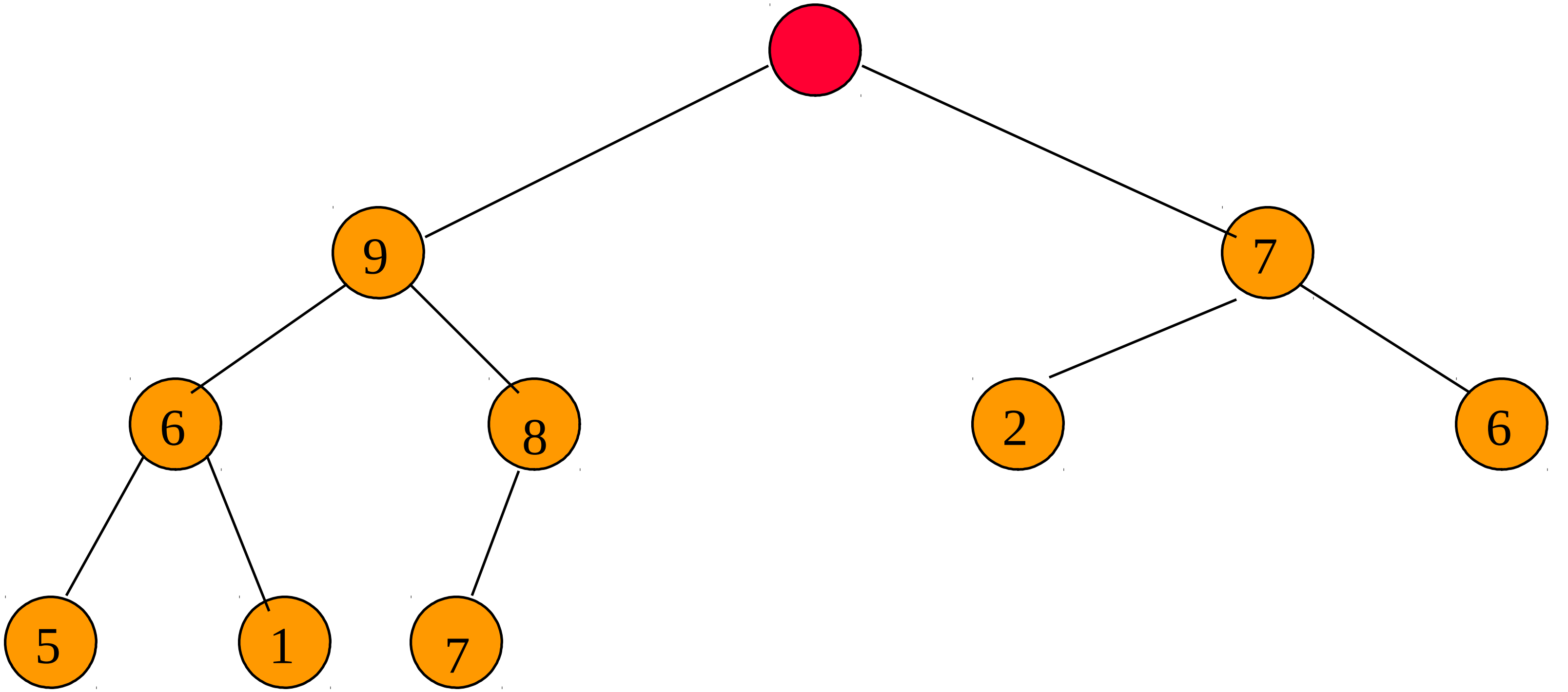
Reinsert 8 into the heap.

Removing The Max Element



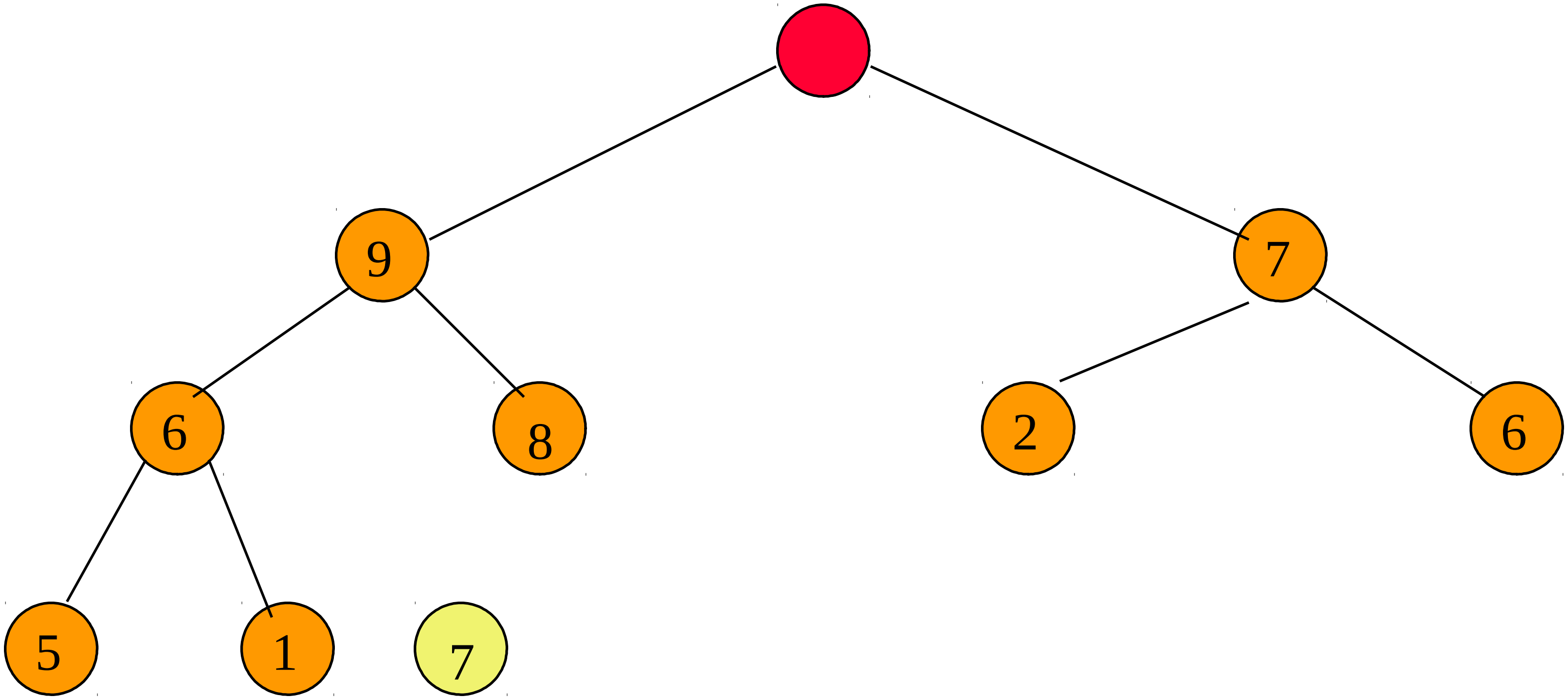
Max element is 15.

Removing The Max Element



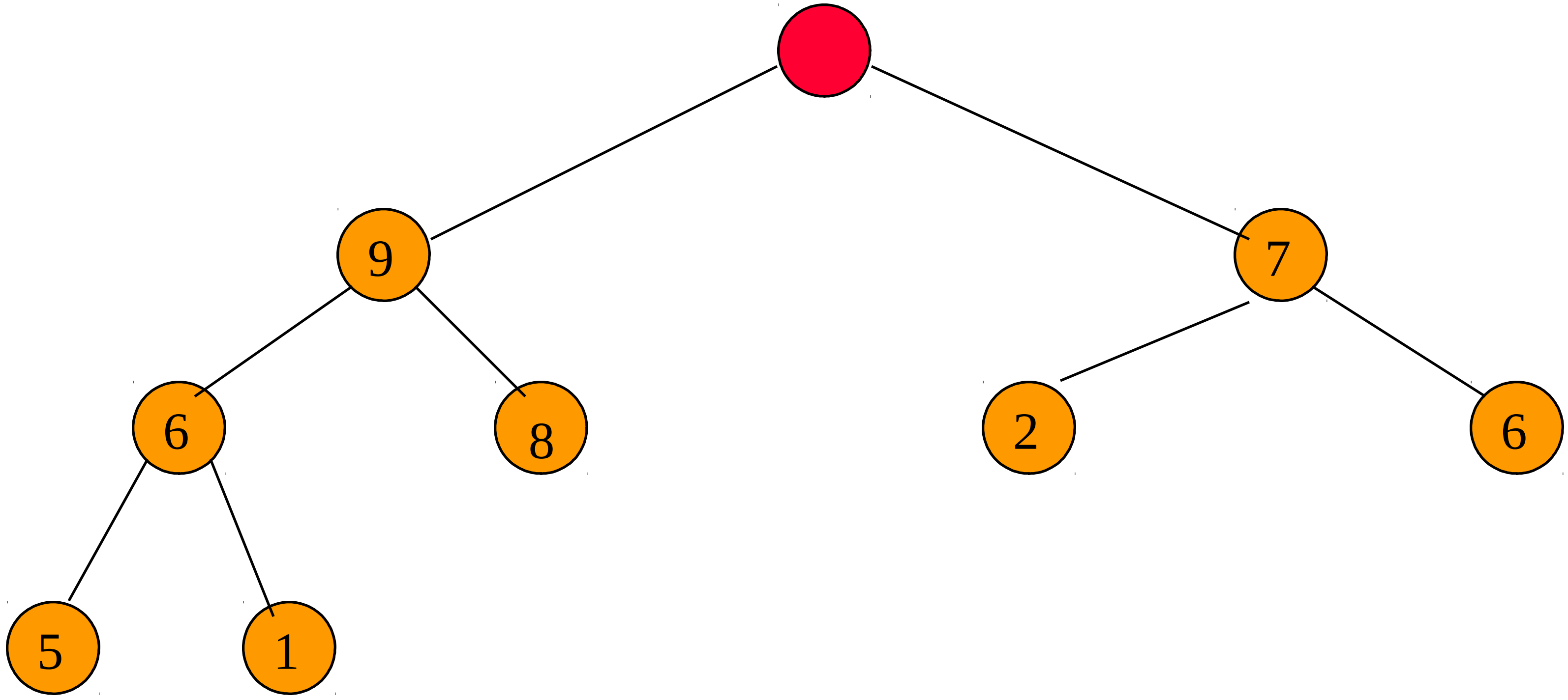
After max element is removed.

Removing The Max Element



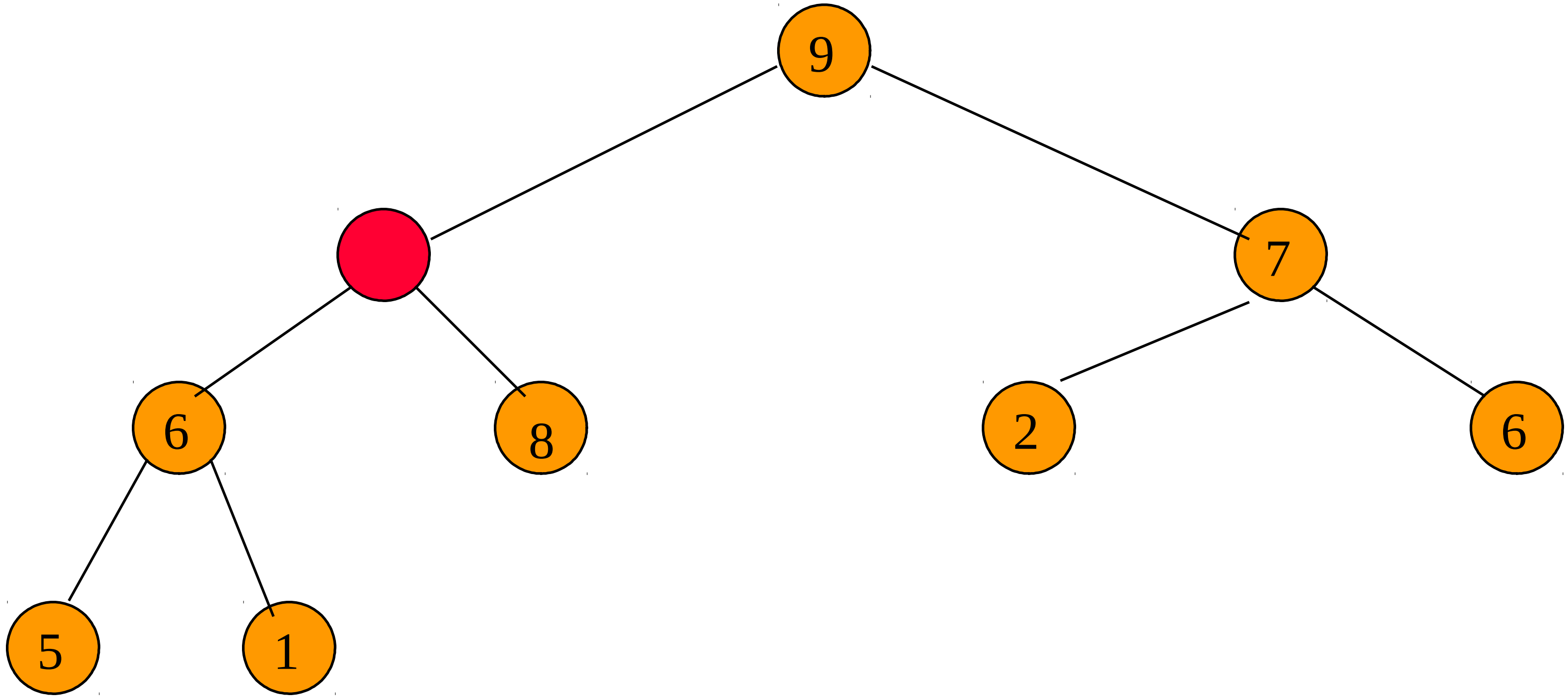
Heap with 9 nodes.

Removing The Max Element



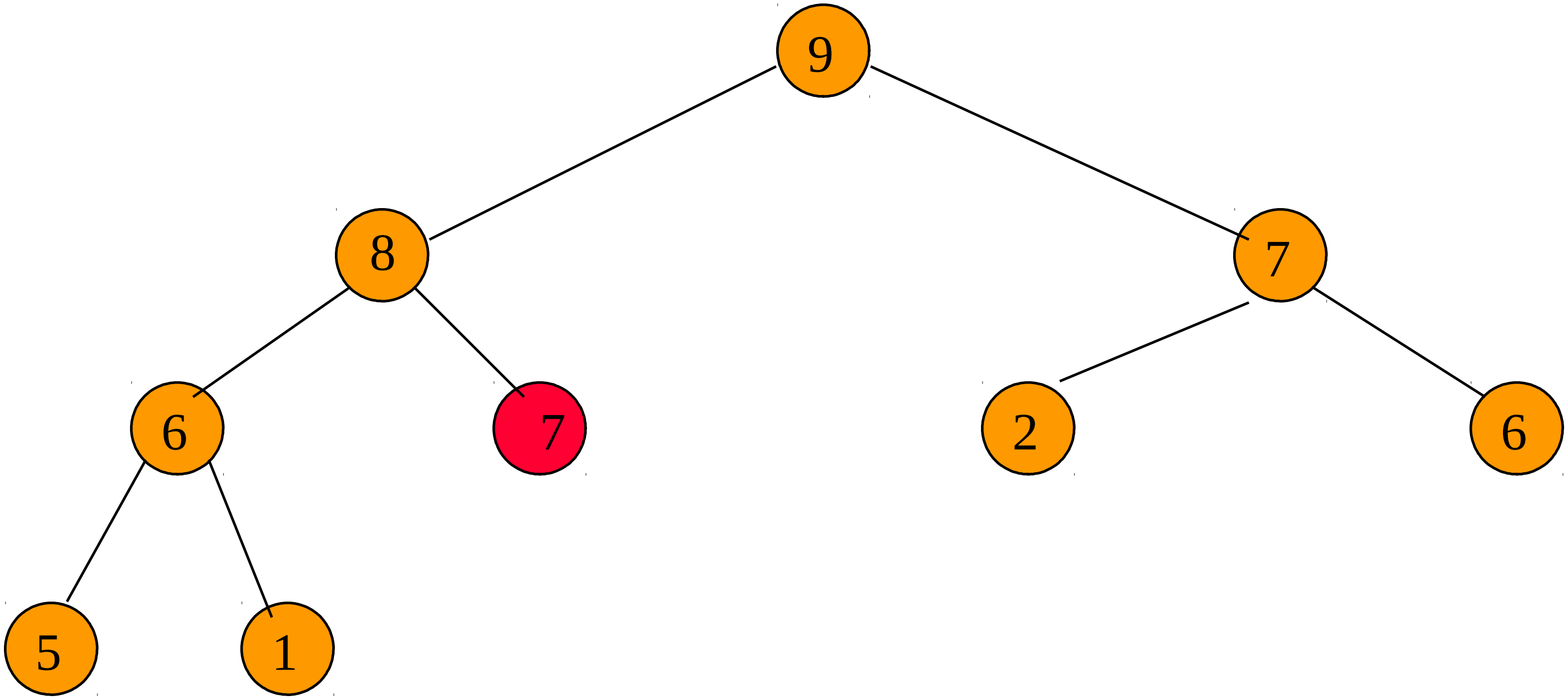
Reinsert 7.

Removing The Max Element



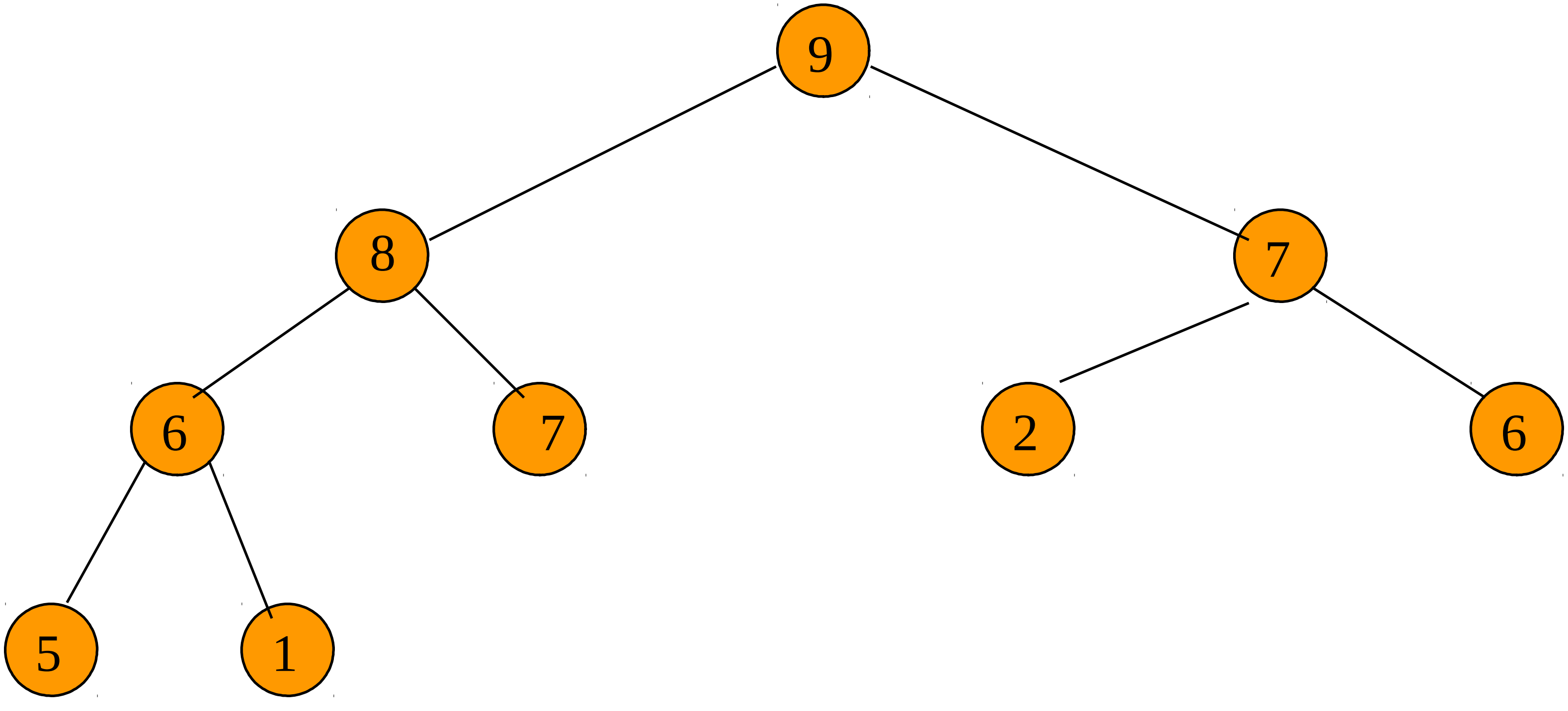
Reinsert 7.

Removing The Max Element



Reinsert 7.

Complexity Of Remove Max Element



Complexity is O(log n).

Priority Queues

Priority queue is a collection of zero or more elements. Each element has a priority or value.

Operations:

- Find an element (Function – top)
- Insert an element (Function – Push)
- Remove an element (Function – Pop)

Two kinds of priority queues:

- Min priority queue. (Find and remove the element with minimum priority value.)
- Max priority queue. (Find and remove the element with maximum priority value.)

AbstractDataType *maxPriorityQueue*
{
instances
finite collection of elements, each has a priority

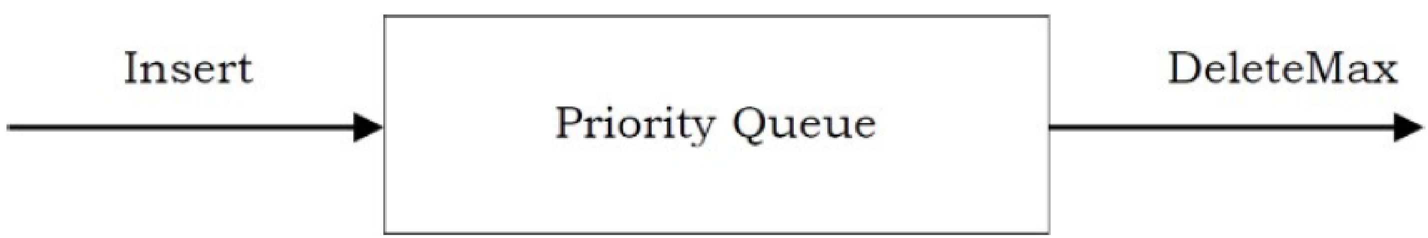
operations
empty() : return **true** iff the queue is empty

size() : return number of elements in the queue

top() : return element with maximum priority

pop() : remove the element with largest priority from the queue;

push(x) : insert the element x into the queue
}



Complexity Of Operations

empty, size, and top => O(1) time

insert (push) and remove (pop) => O(log n) time where n is the size of the priority queue

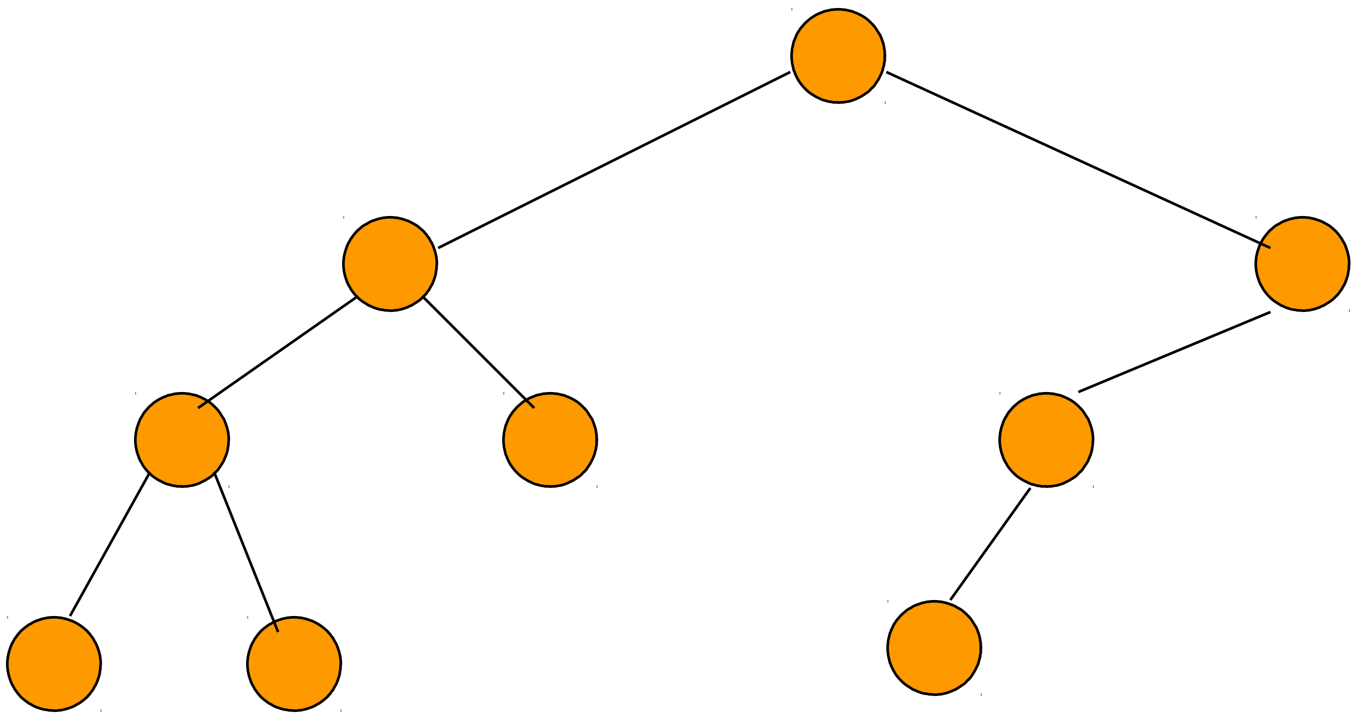
Leftist Trees

- **Leftist tree** is a linked data structure suitable for the implementation of a priority queue.
- A tree which tends to **“lean” to the left**.
- Linked binary tree.
- Can do everything a heap can do and in the same asymptotic complexity.
- Can meld two leftist tree priority queues in O(log n) time.

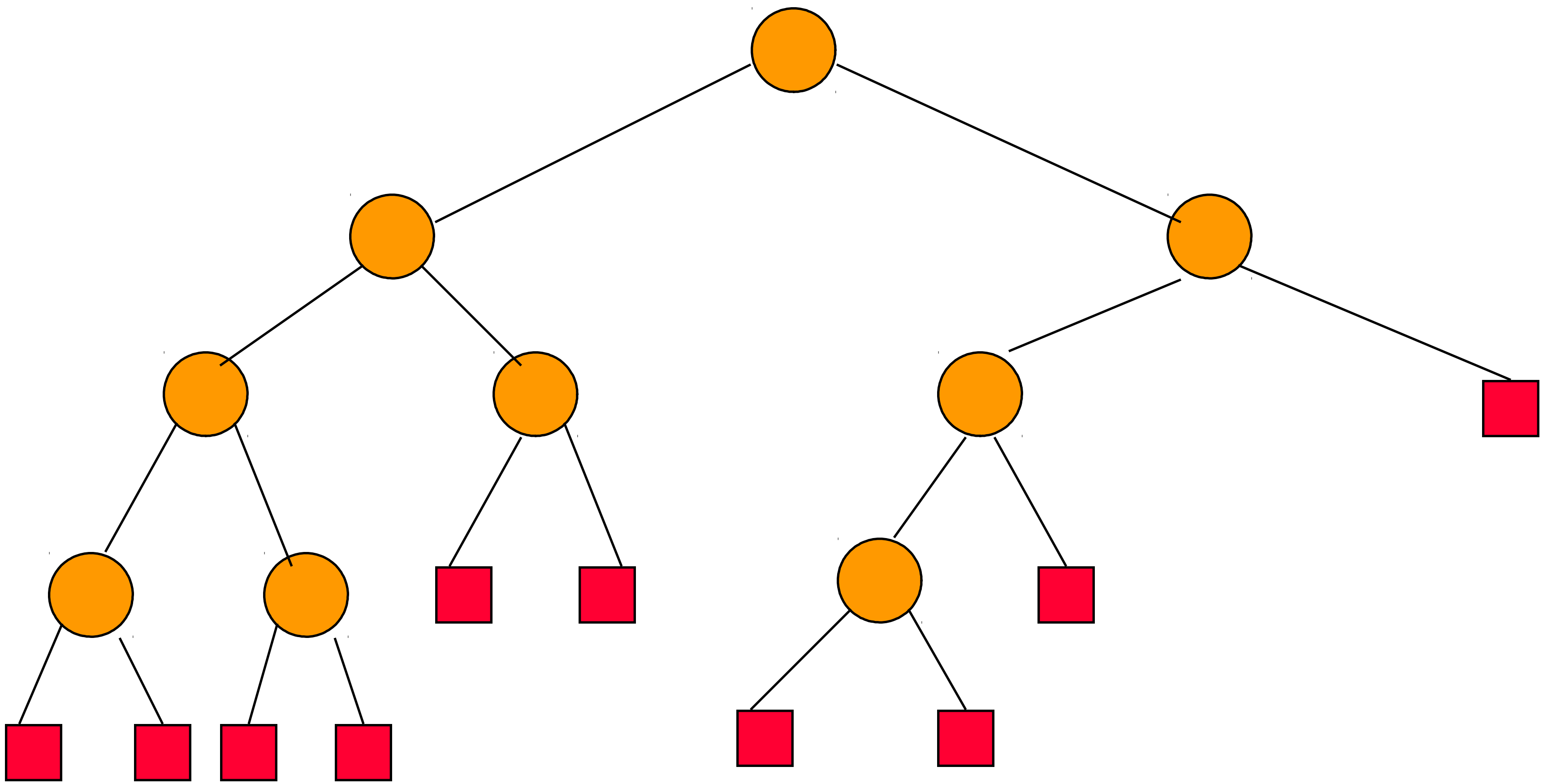
Extended Binary Trees

- External node – a special node that replaces each empty subtree
- Internal node – a node with non-empty subtrees
- Extended binary tree – a binary tree with external nodes added. Start with any binary tree and add an external node wherever there is an empty subtree.
- Result is an extended binary tree.

A Binary Tree

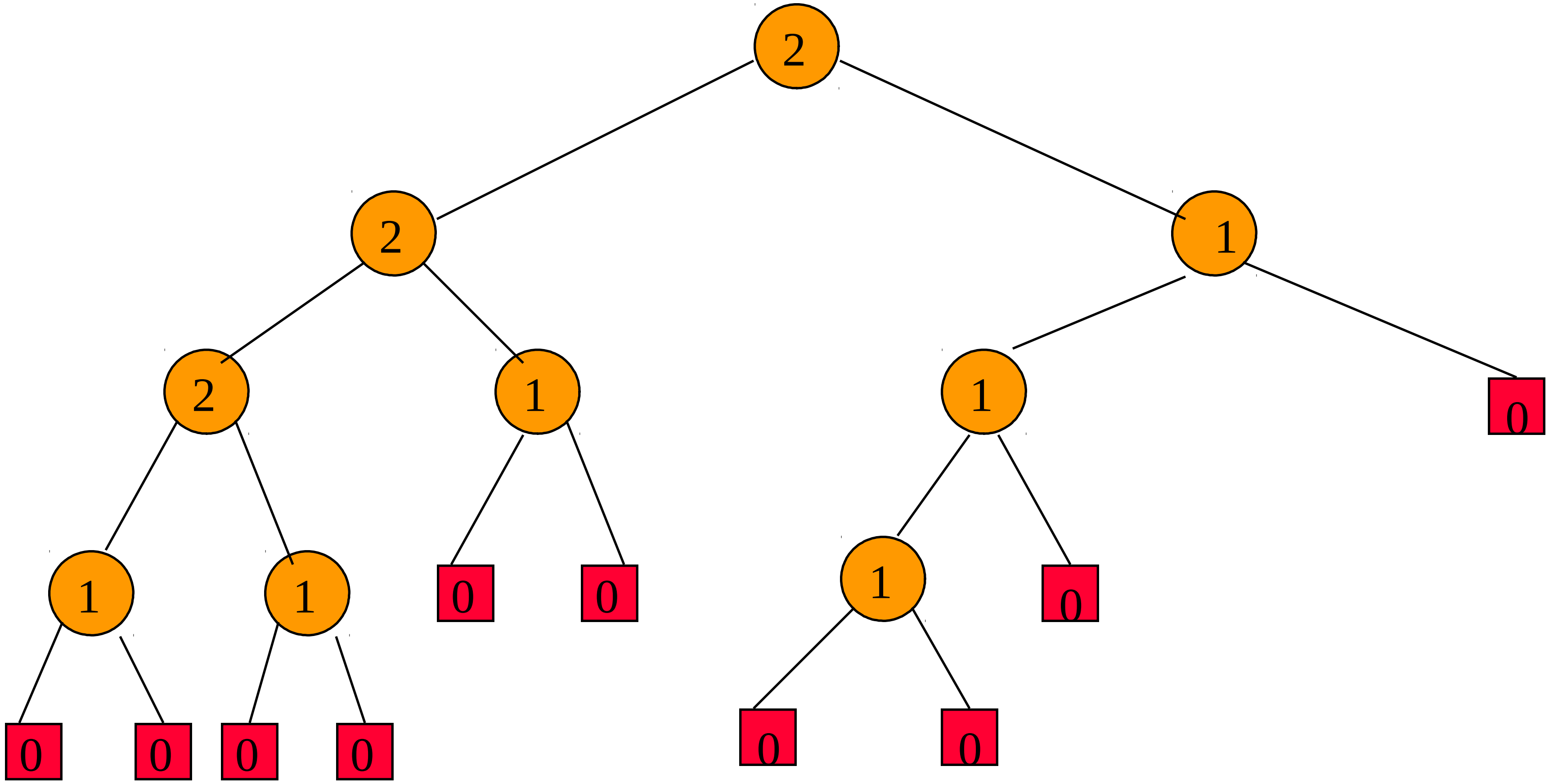


An Extended Binary Tree



number of external nodes is n+1

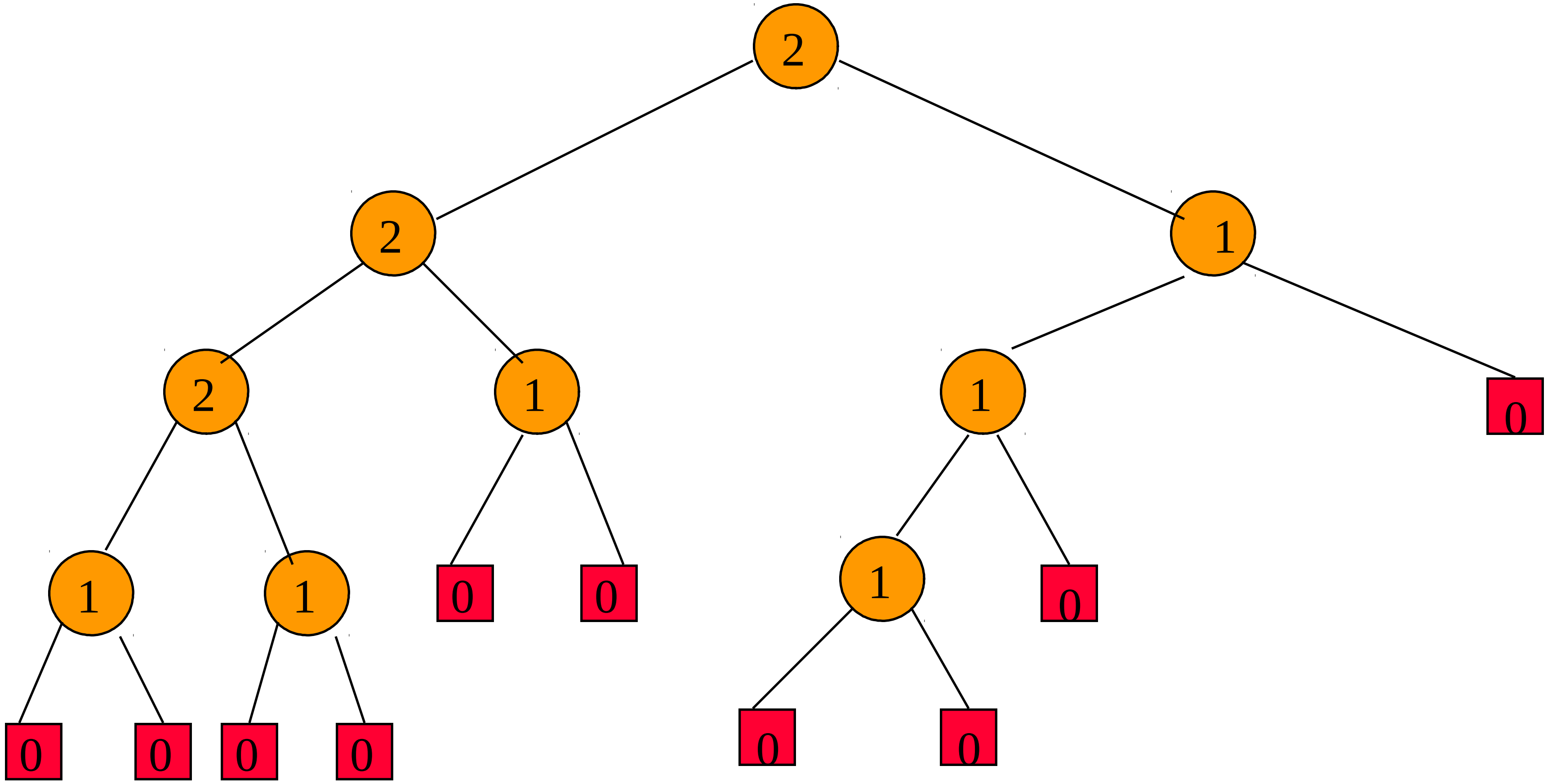
A Leftist Tree



Leftist Trees--Property 1

In a leftist tree, the rightmost path is a shortest root to external node path and the length of this path is s(root).

A Leftist Tree



Length of rightmost path is 2.

Leftist Trees—Property 2

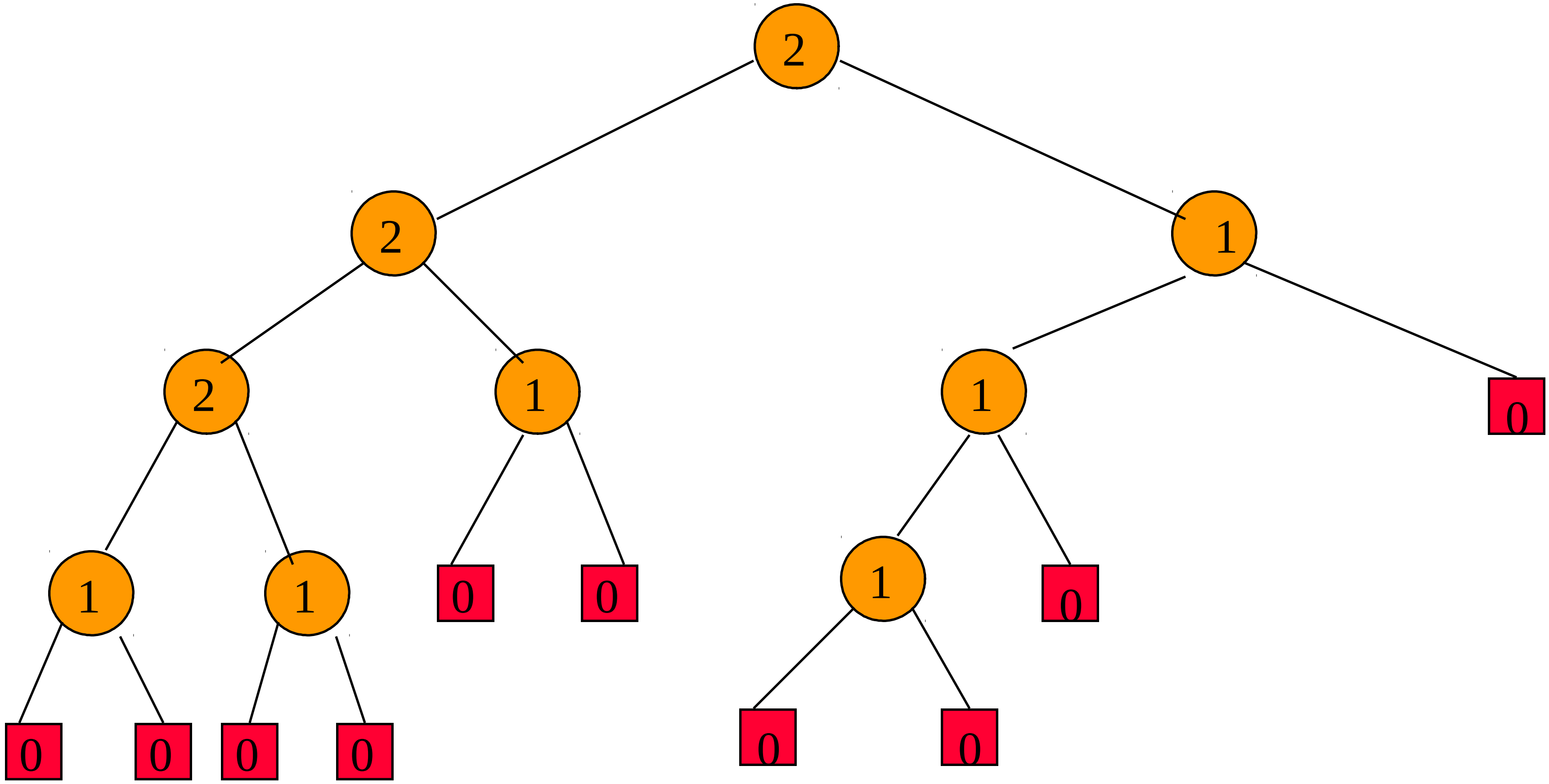
The number of internal nodes is at least

$2^{s(root)} - 1$

Because levels 1 through s(root) have no external nodes.

So, $s(root) \le \log(n+1)$

A Leftist Tree



Levels 1 and 2 have no external nodes.

Leftist Trees—Property 3

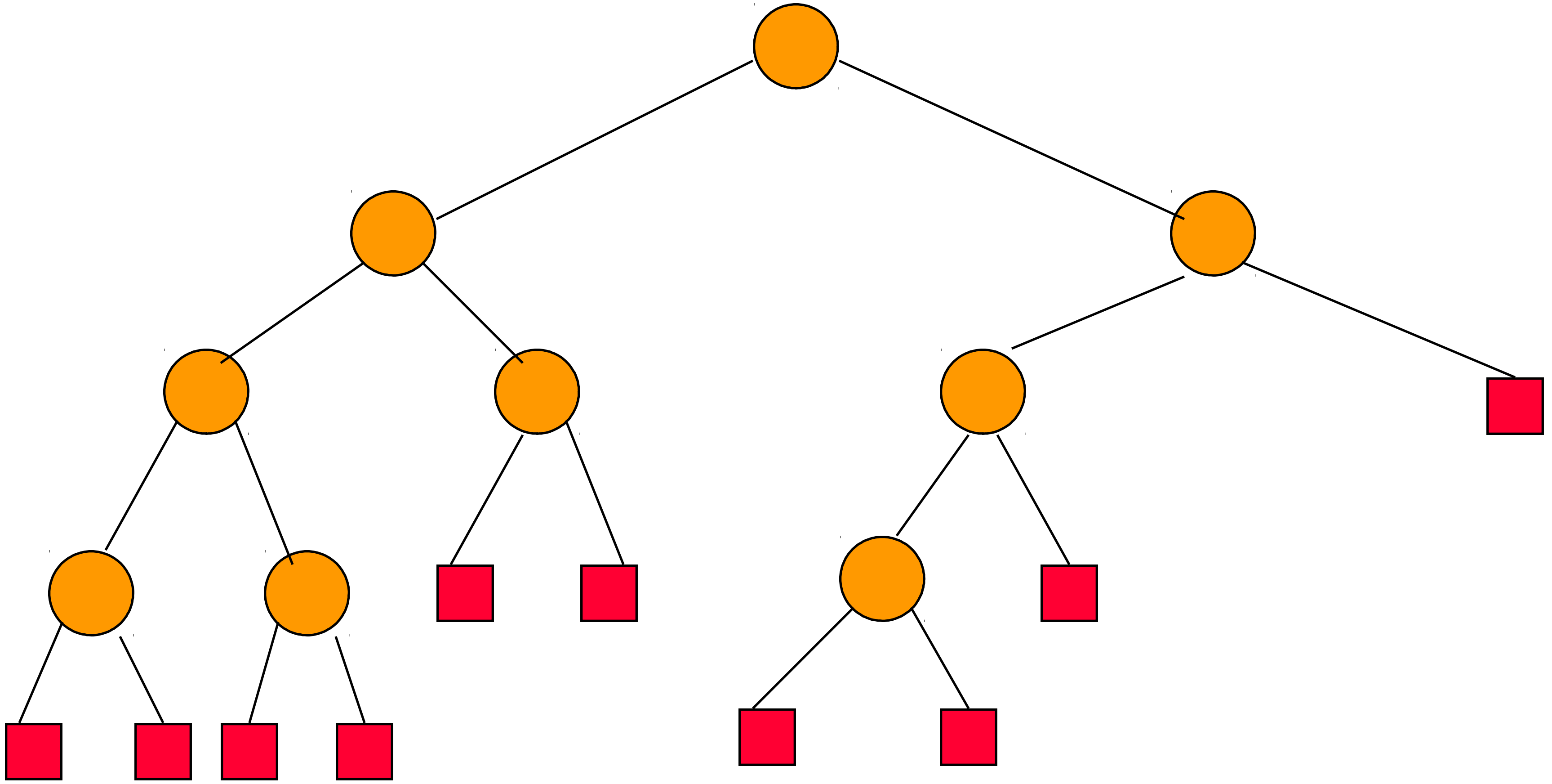
Length of rightmost path is O(log n), where n is the number of nodes in a leftist tree.

Follows from Properties 1 and 2.

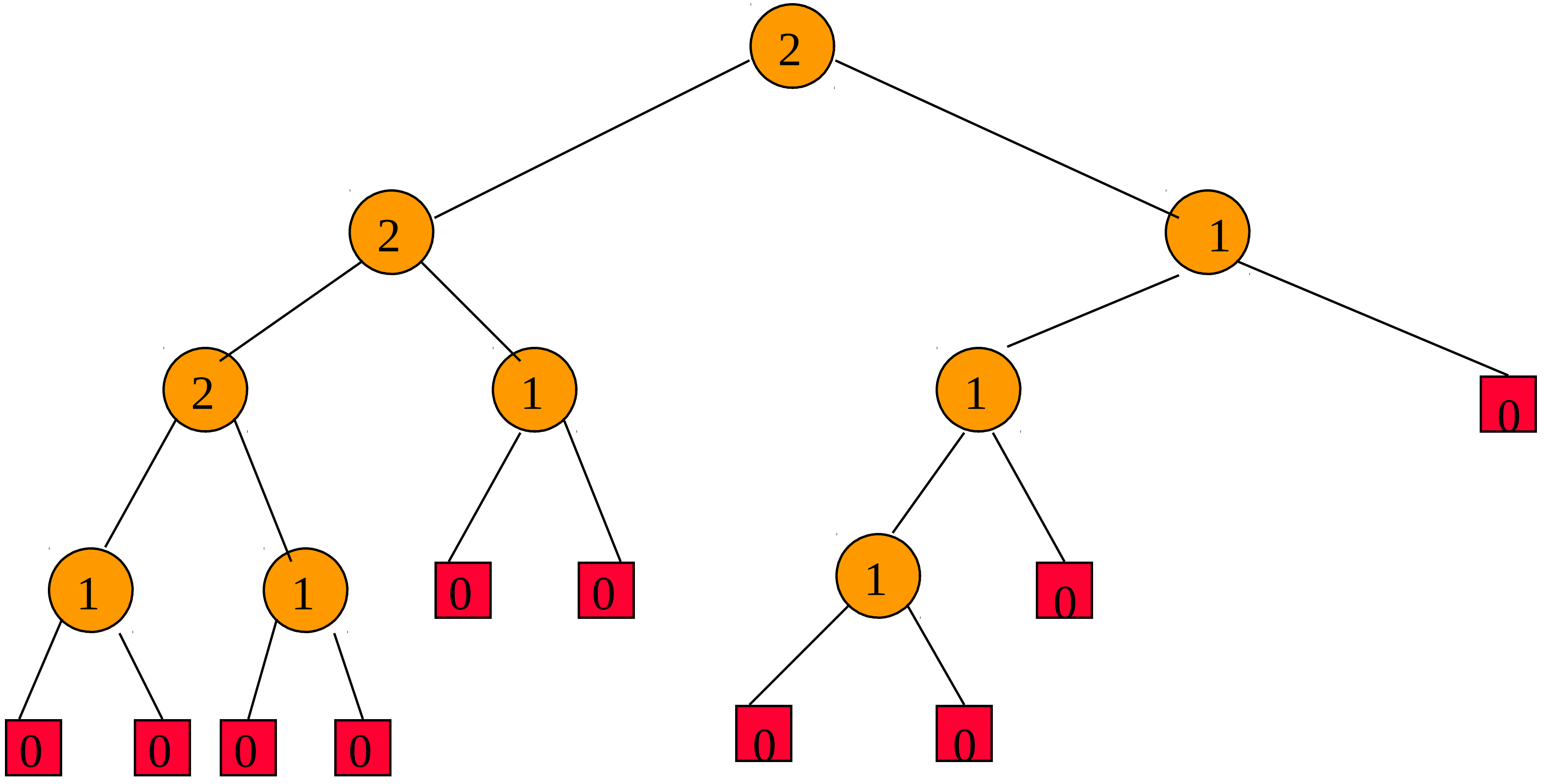
The Function s()

For any node x in an extended binary tree, s(x) is the length of a shortest path from x to an external node in the subtree rooted at x.

s() Values Example



s() Values Example



Properties Of s()

If x is an external node, then $s(x) = 0$.

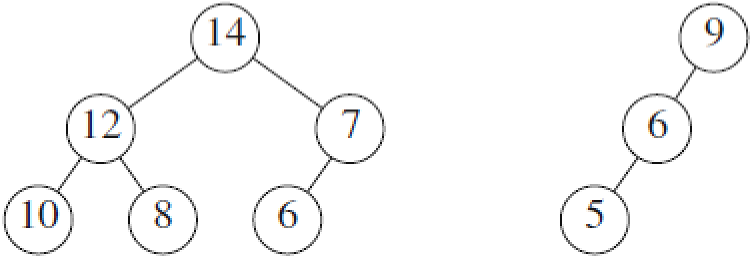
Otherwise,

$$s(x) = \min \{s(\mathrm{leftChild}(x)), \\ s(\mathrm{rightChild}(x))\} + 1$$

Height Biased Leftist Trees

A binary tree is a Height Biased Leftist Tree **(HBLT)** iff for every internal node x,

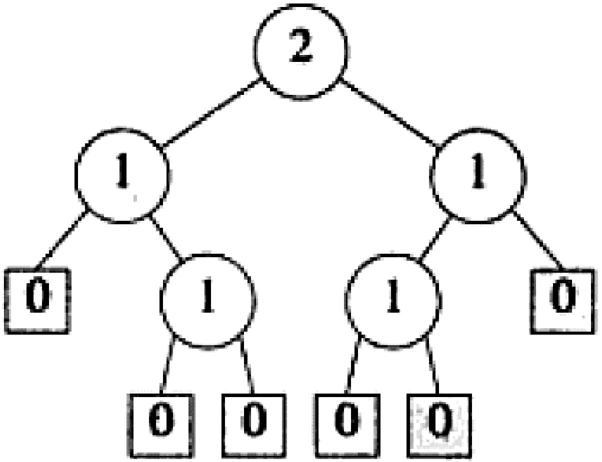
s(leftChild(x)) >= s(rightChild(x))



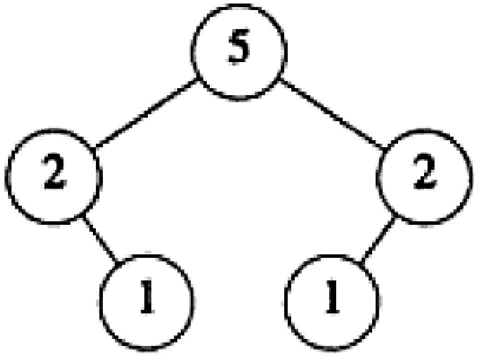
A max HBLT is an HBLT that is also a max tree. A min HBLT is an HBLT that is also a min tree.

W values

- The weight w(x) of a node x is the number of internal nodes in the subtree with the root x.
- If x is external node, its weight is 0.
- If x is internal node, its weight is 1 more than the sum of the weights of its children.



S values



W values

Weight Biased Leftist Tree

- A binary tree is a weight biased leftist tree (WBLT) iff at every internal node the w value of the left child is great than or equal to the w value of the right child.

A max WBLT is a WBLT that is also a max tree. A min WBLT is a WBLT that is also a min tree.

Some Min Leftist Tree Operations

empty()

size()

top()

push()

pop()

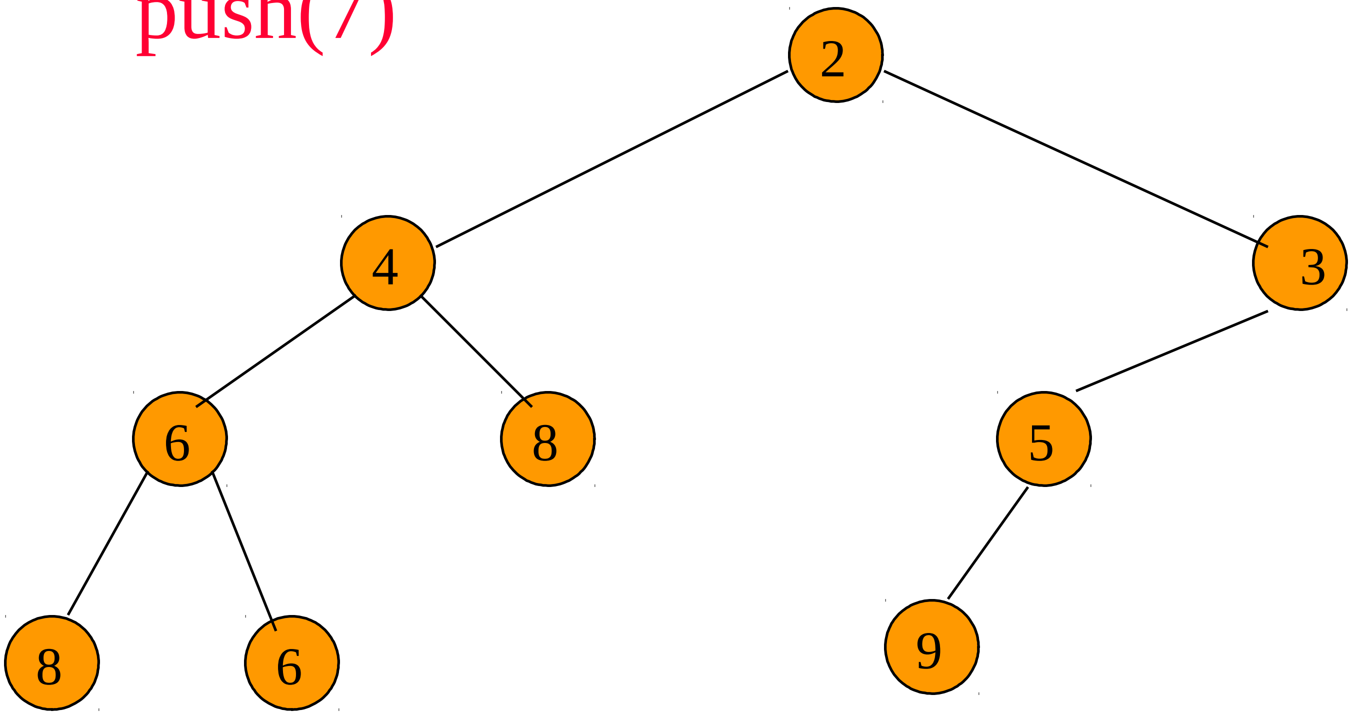
meld()

initialize()

push() and pop() use meld().

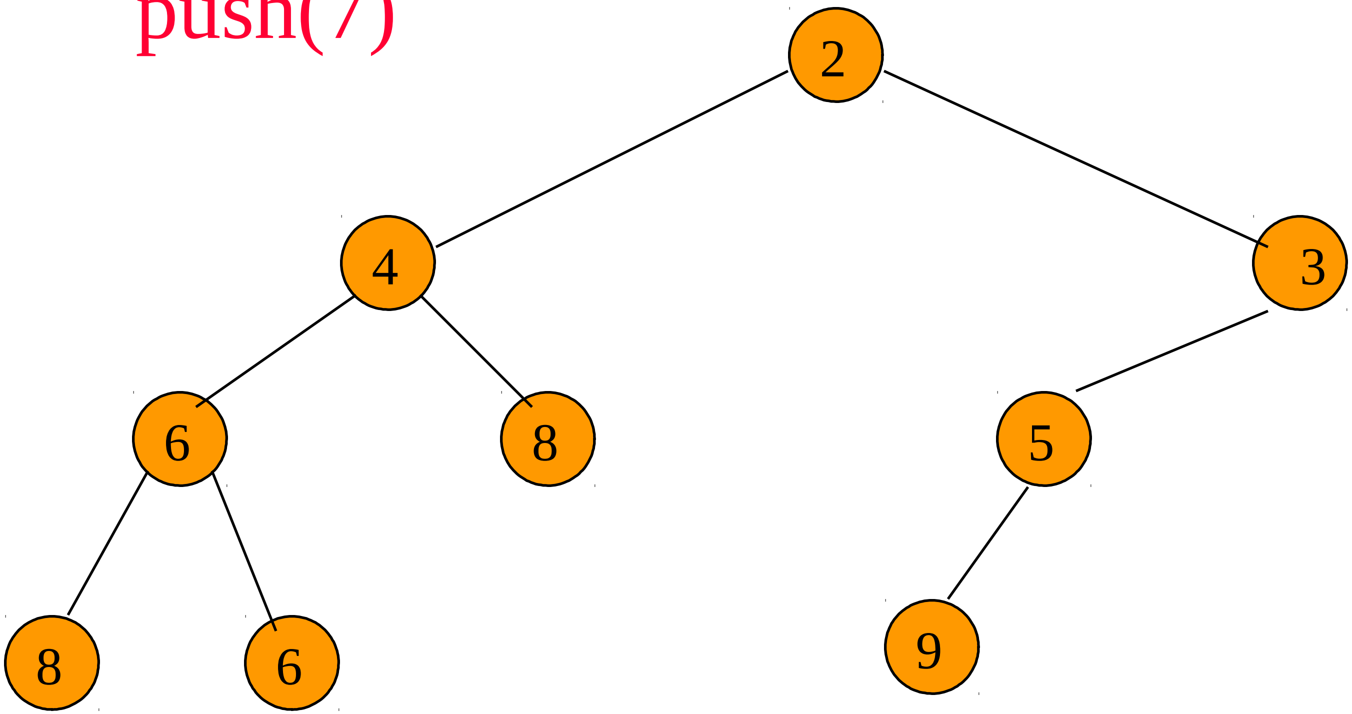
Push Operation

push(7)



Push Operation

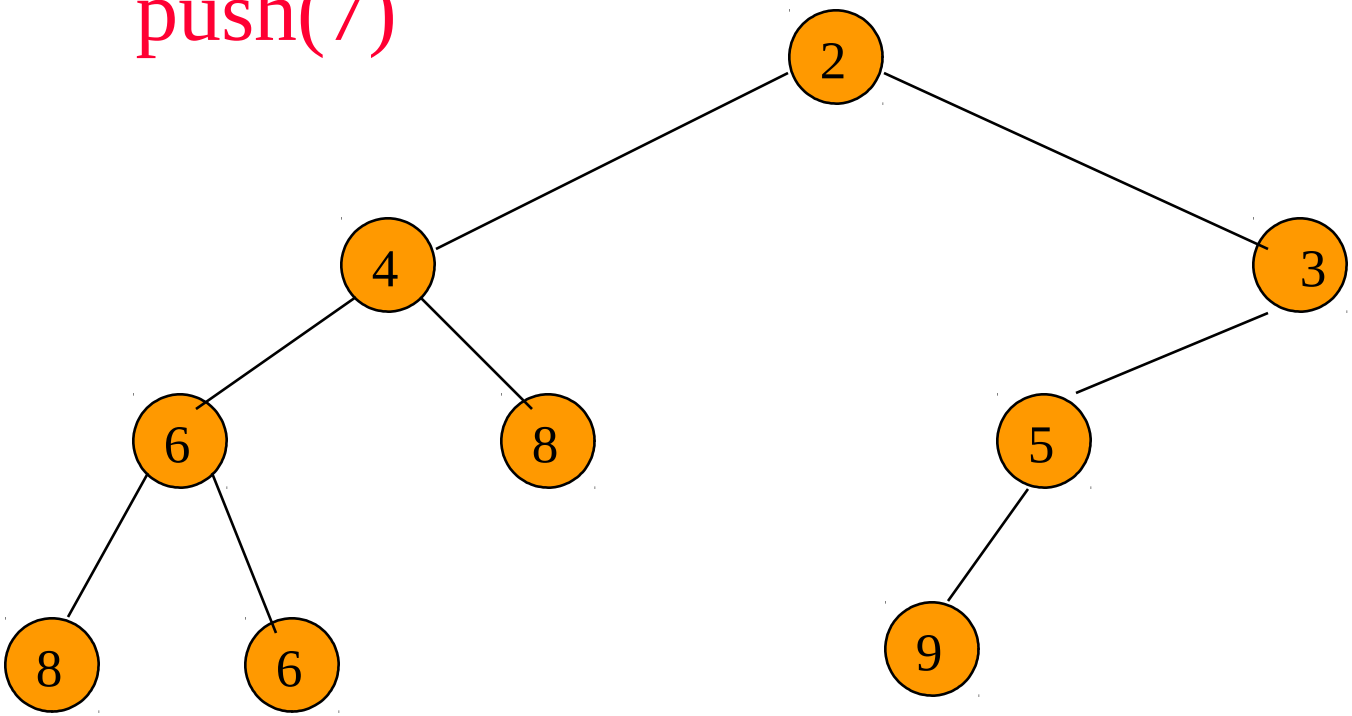
push(7)



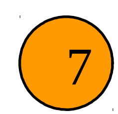
Create a single node min leftist tree.

Push Operation

push(7)

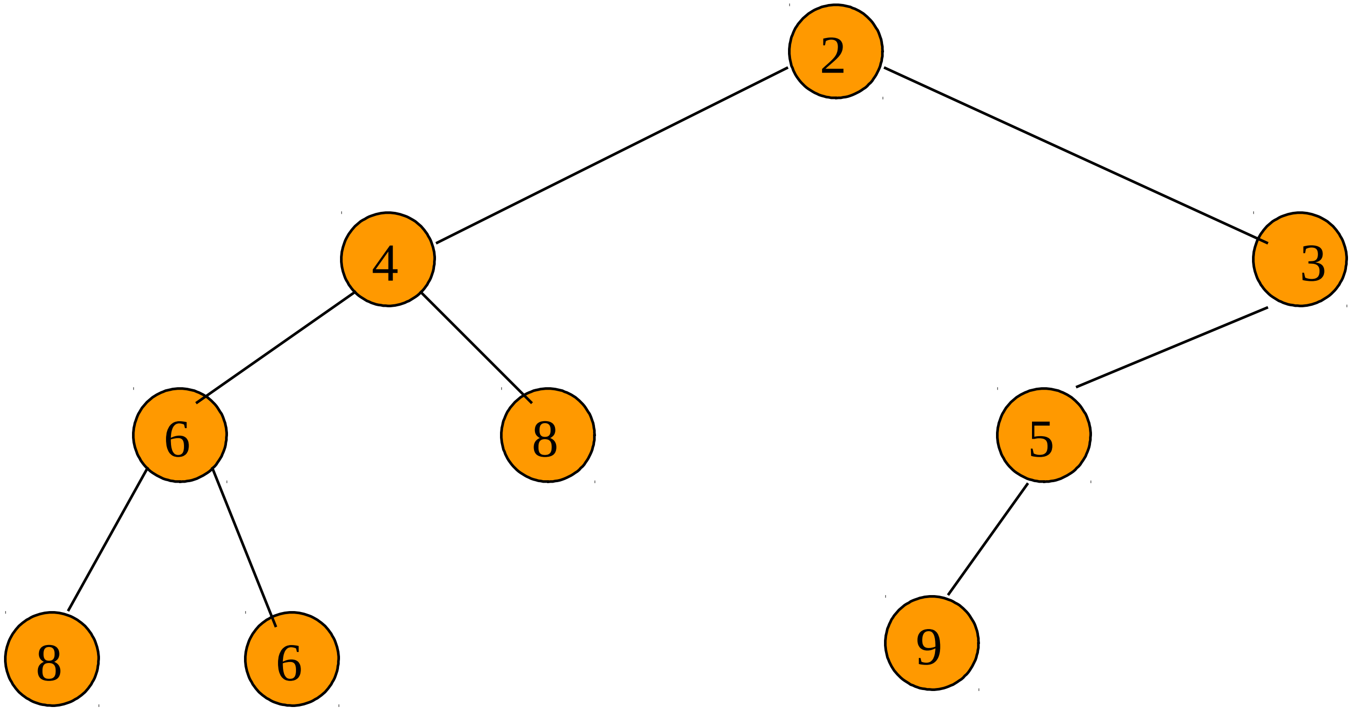


Create a single node min leftist tree.

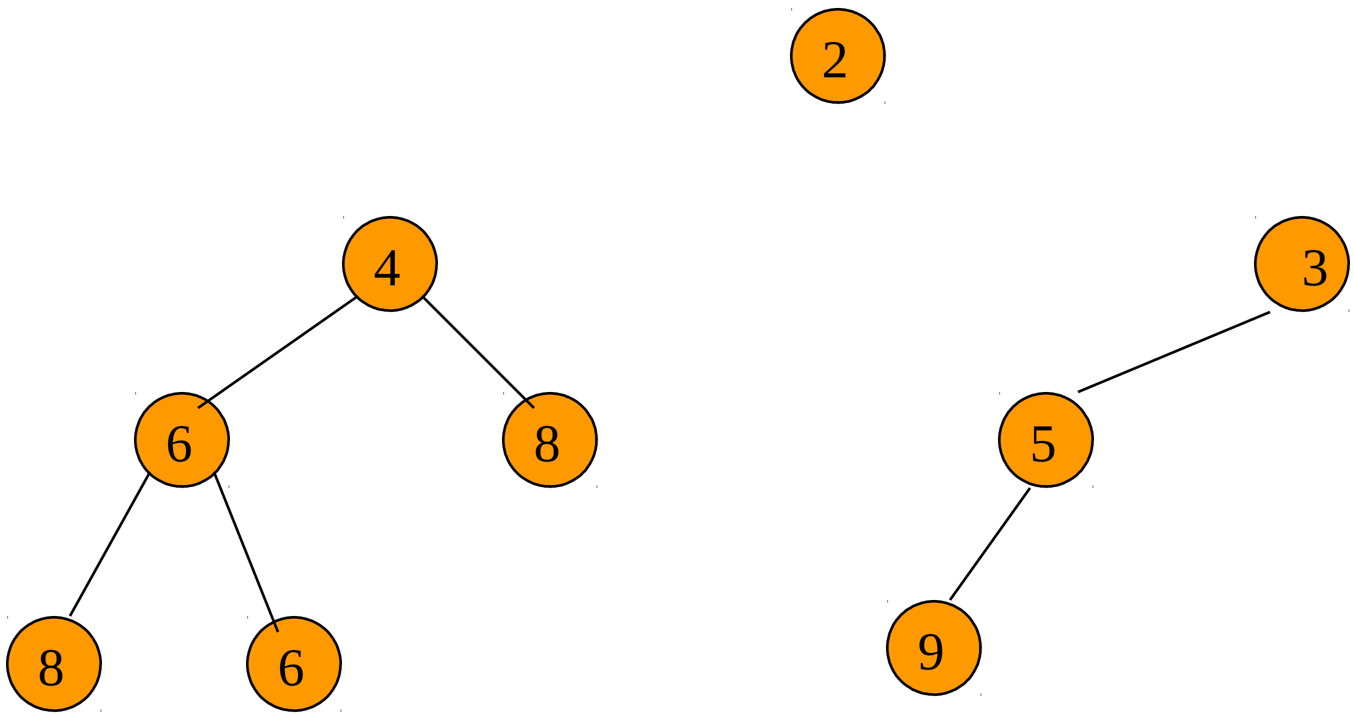


Meld the two min leftist trees.

Remove Min (pop)

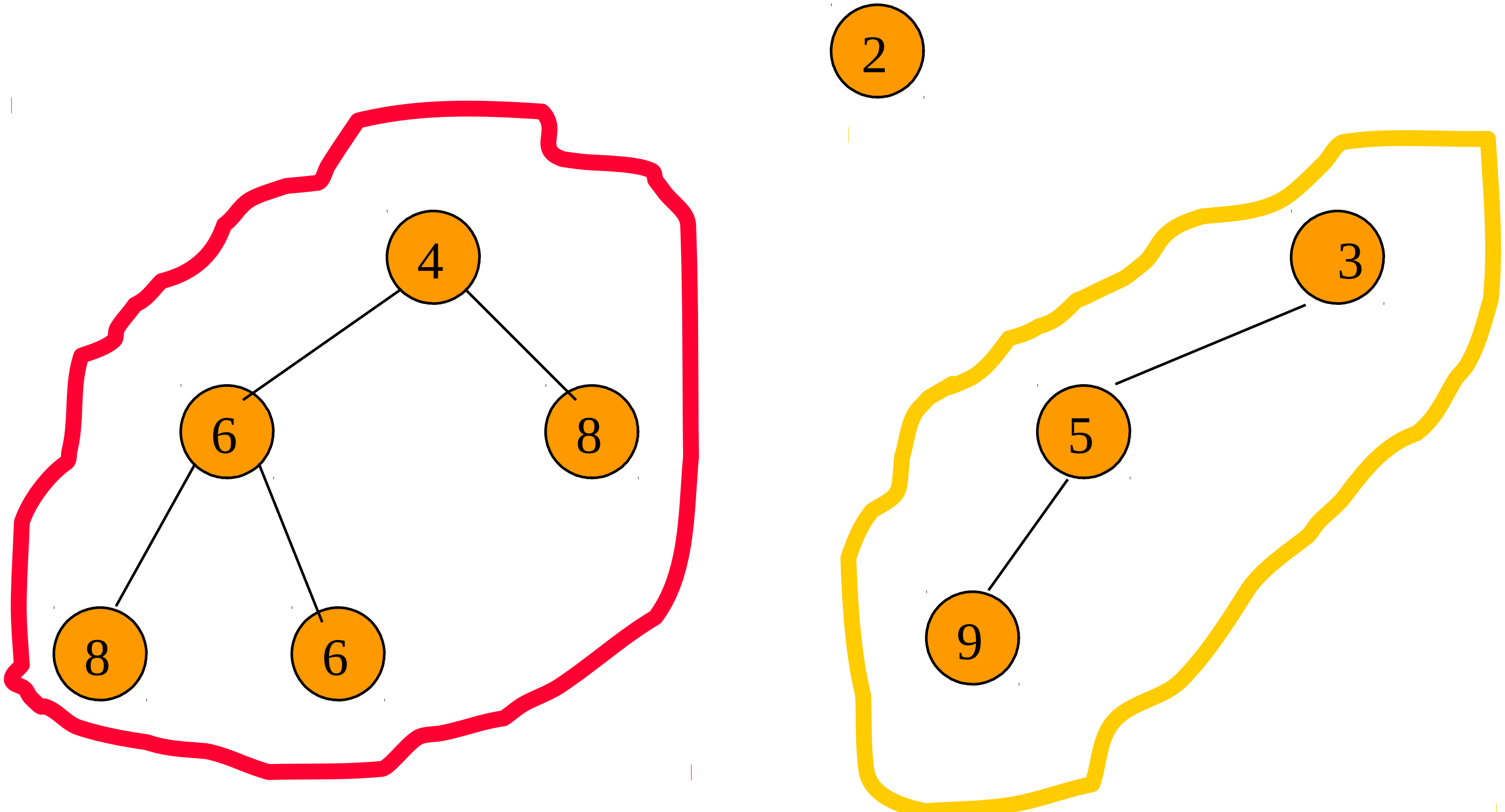


Remove Min (pop)



Remove the root.

Remove Min (pop)



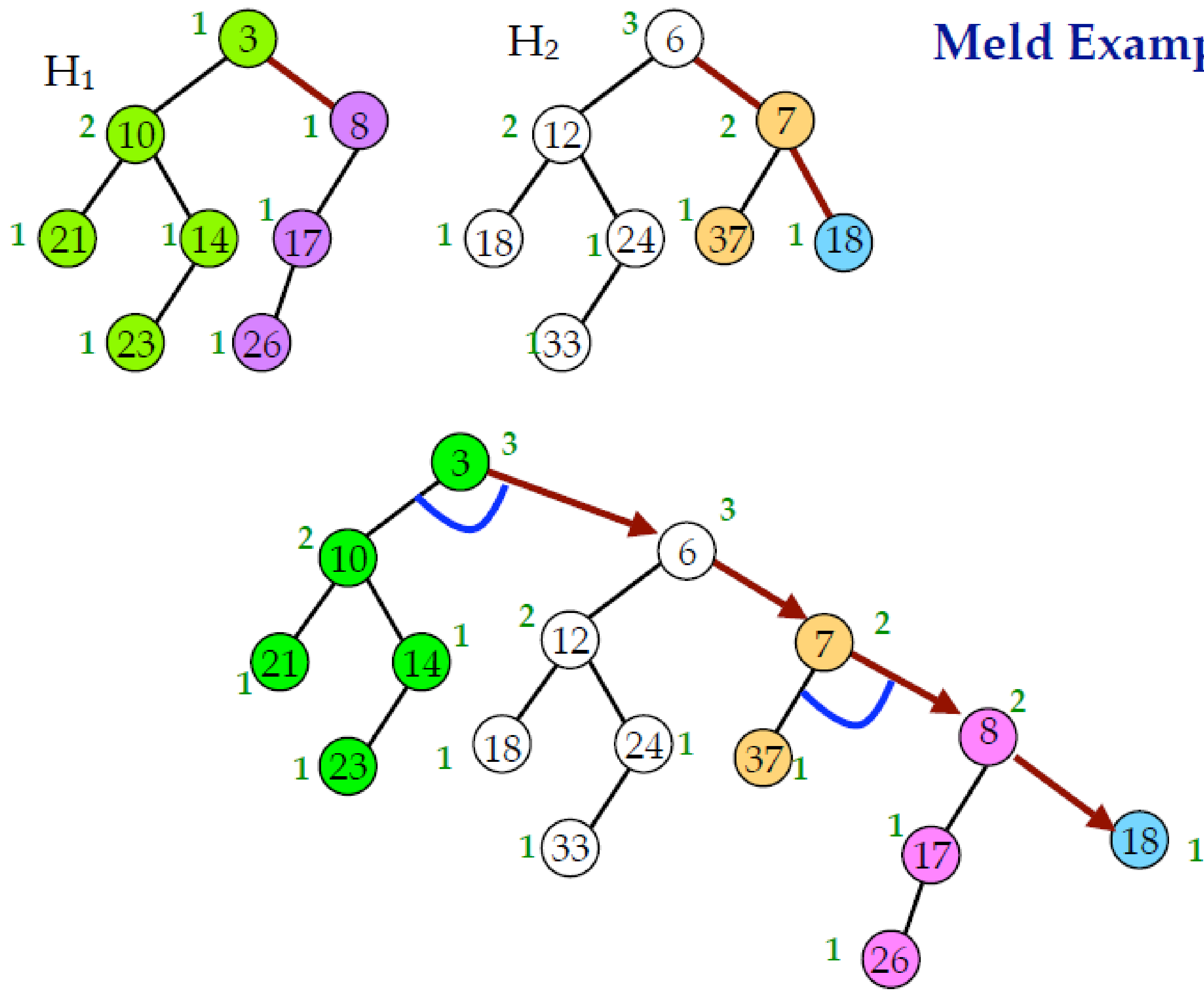
Remove the root.

Meld the two subtrees.

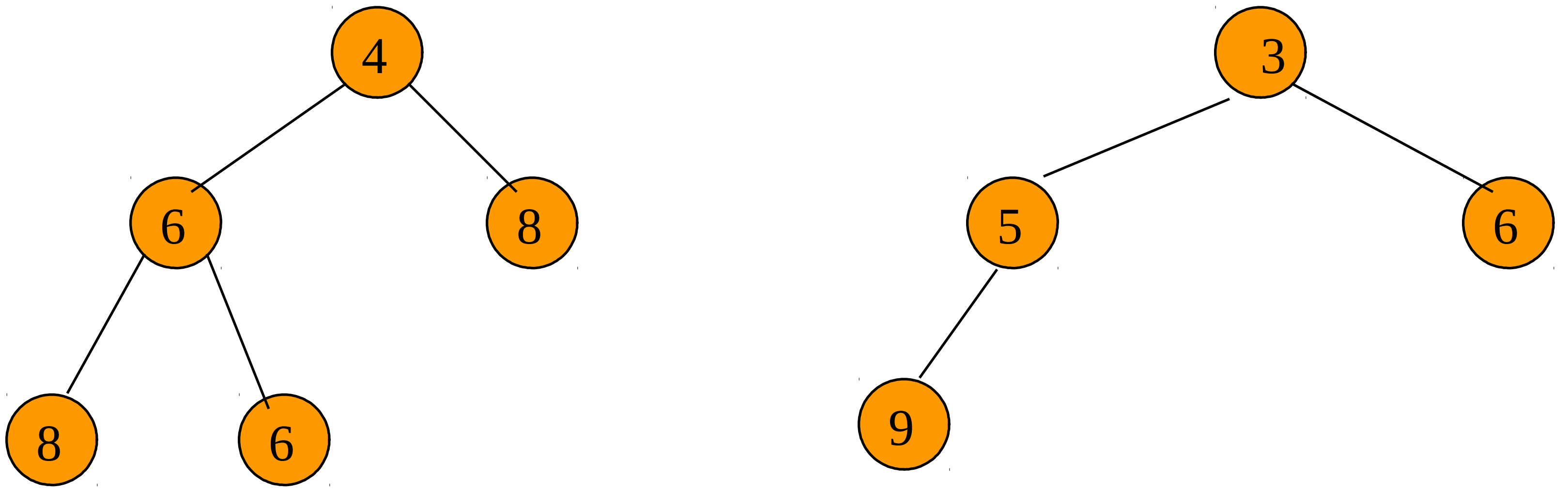
Meld Two Leftist Trees

meld(H_1, H_2): return new heap with the keys from H_1 and H_2, destroying heaps H_1 and H_2.

Meld Example

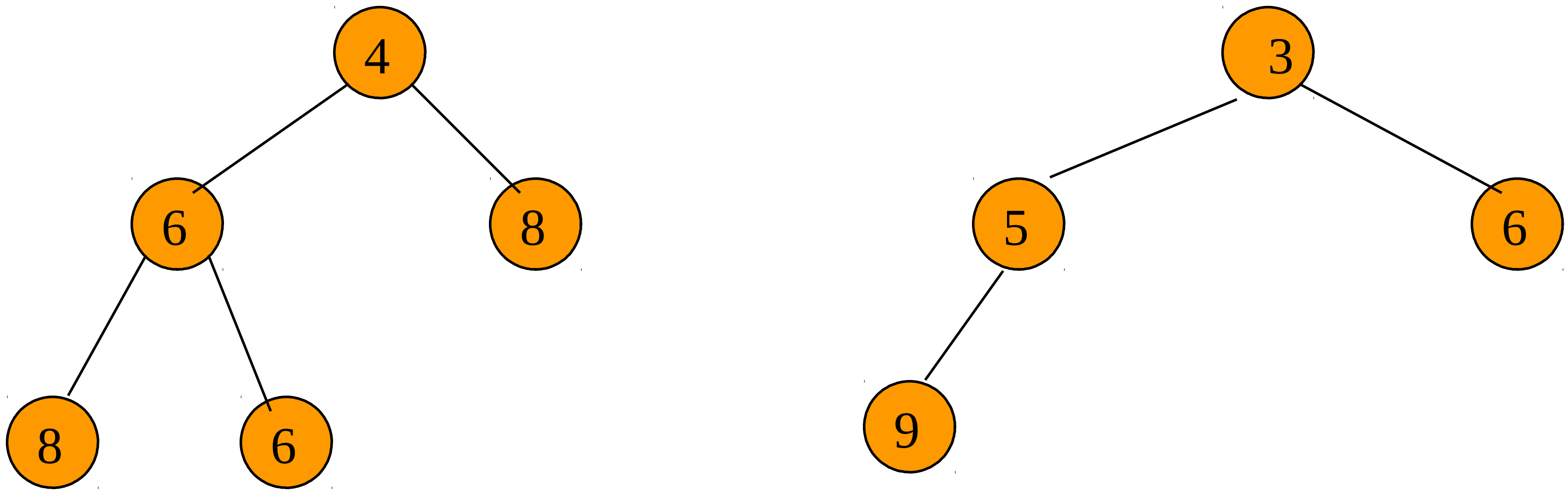


Another example – Meld operation



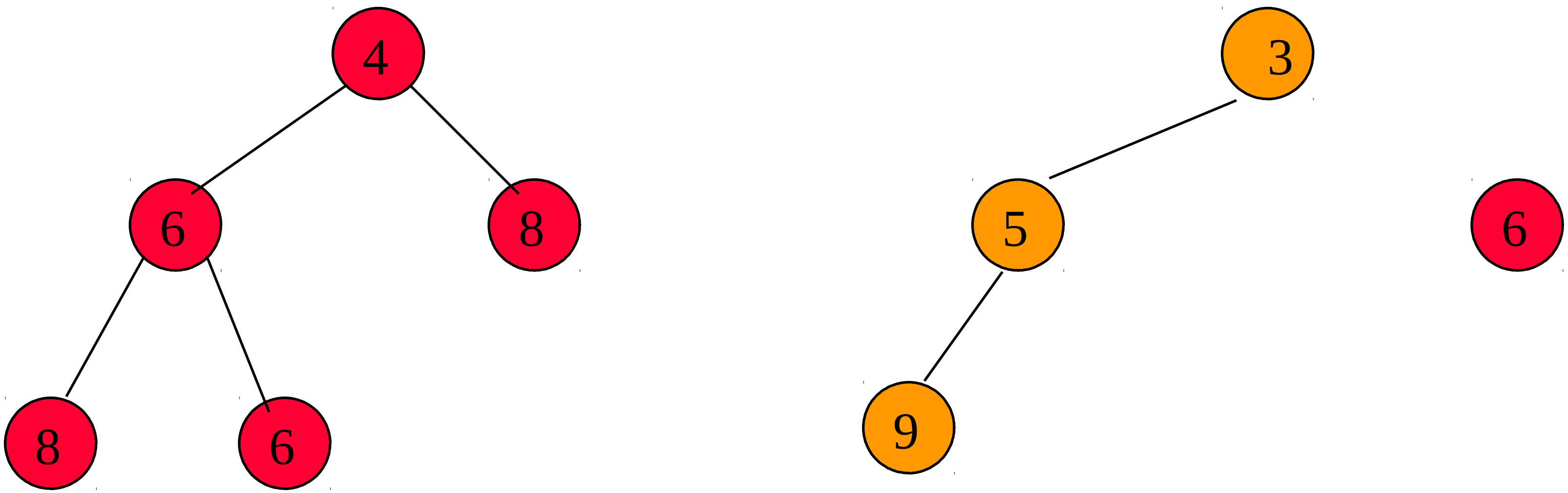
Consider two trees H1 and H2 to be melded.
Traverse only the rightmost paths so as to get logarithmic performance.

Meld Two Min Leftist Trees



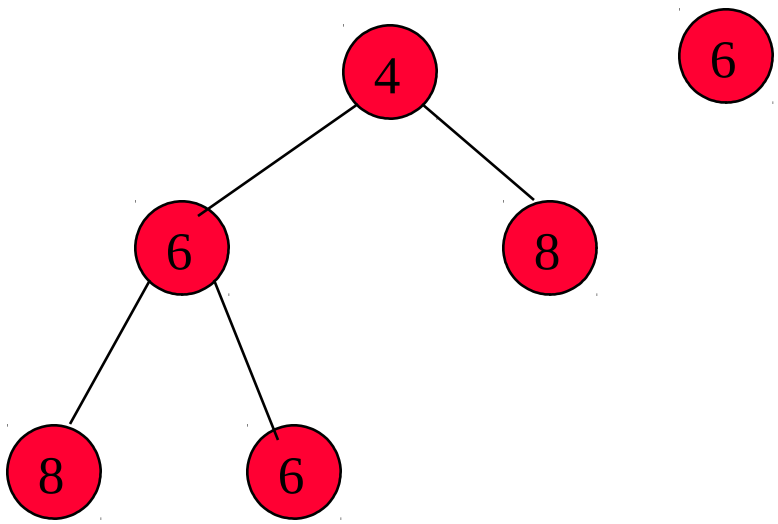
Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

Meld Two Min Leftist Trees



Meld right subtree of tree with smaller root and all of other tree.

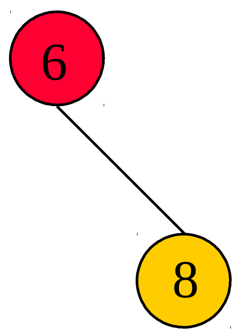
Meld Two Min Leftist Trees

Meld right subtree of tree with smaller root and all of other tree.

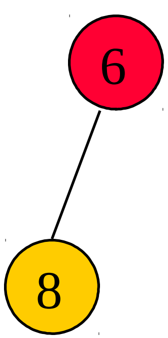
Right subtree of 6 is empty. So, result of melding right subtree of tree with smaller root and other tree is the other tree.

Meld Two Min Leftist Trees

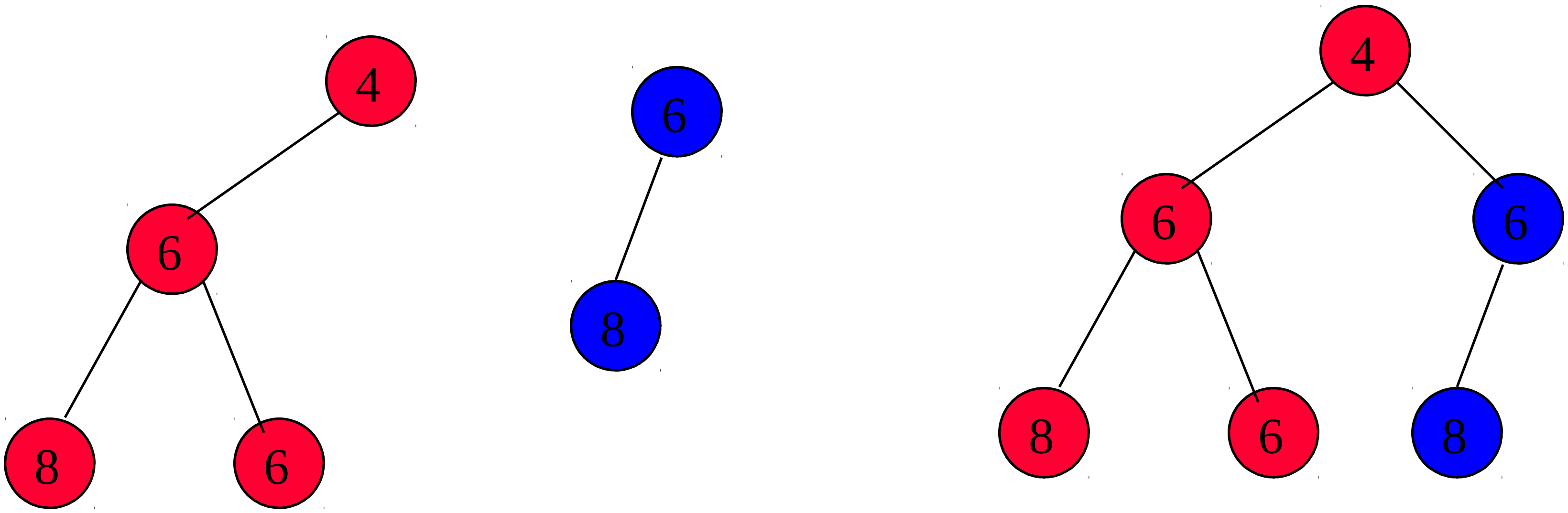
Make melded subtree right subtree of smaller root.



Swap left and right subtree if s(left) < s(right).



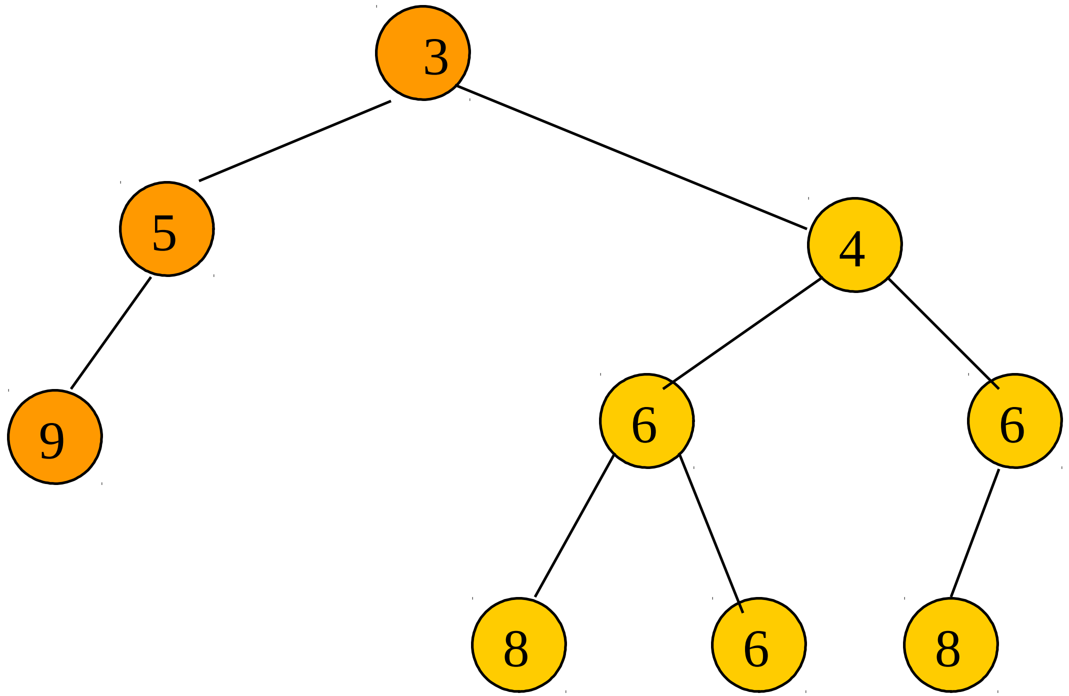
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

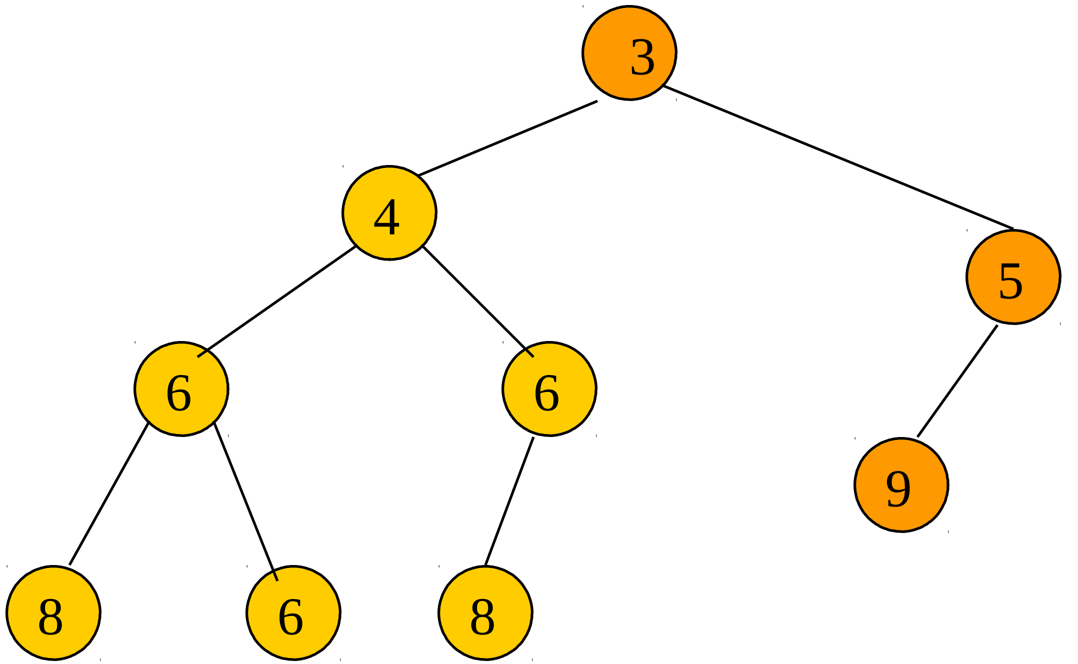
Meld Two Min Leftist Trees



Make melded subtree right subtree of smaller root.

Swap left and right subtree if s(left) < s(right).

Meld Two Min Leftist Trees



Initializing In O(n) Time

- create n single node min leftist trees and place them in a FIFO queue
- repeatedly remove two min leftist trees from the FIFO queue, meld them, and put the resulting min leftist tree into the FIFO queue
- the process terminates when only 1 min leftist tree remains in the FIFO queue
- analysis is the same as for heap initialization

Applications

Sorting

- use element key as priority
- insert elements to be sorted into a priority queue
- remove/pop elements in priority order
  - if a min priority queue is used, elements are extracted in ascending order of priority (or key)
  - if a max priority queue is used, elements are extracted in descending order of priority (or key)

Sorting

Algorithm PriorityQueueSort(S, P):

    Input: A sequence S storing n elements, on which a total order relation is defined, and a Priority Queue P that compares keys with the same relation

    Output: The Sequence S sorted by the total order relation

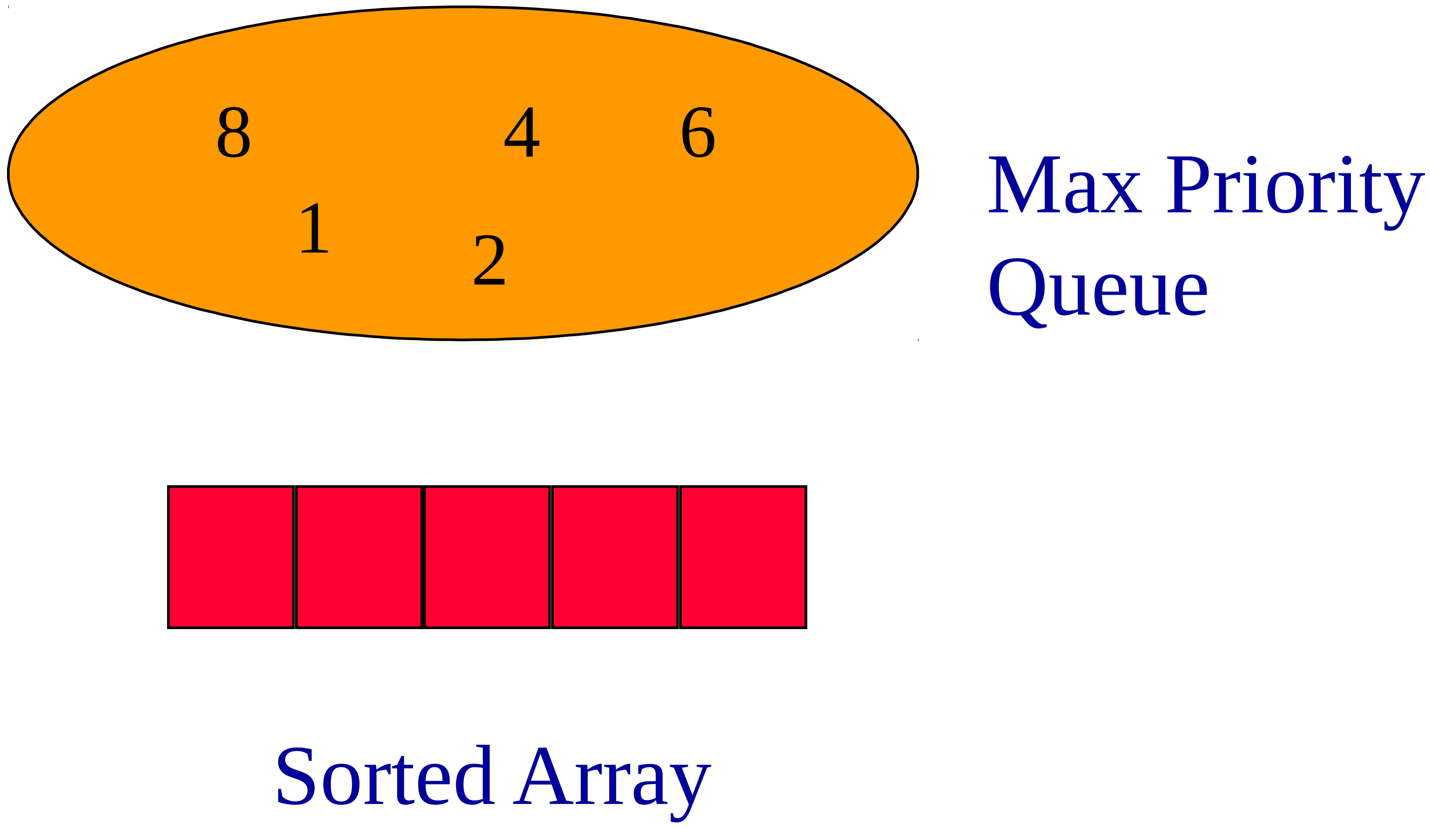
```
while !S.isEmpty() do
    e ← S.removeFirst()
    P.insertItem(e, e)
while P is not empty do
    e ← P.removeMin()
    S.insertLast(e)
```

Sorting Example

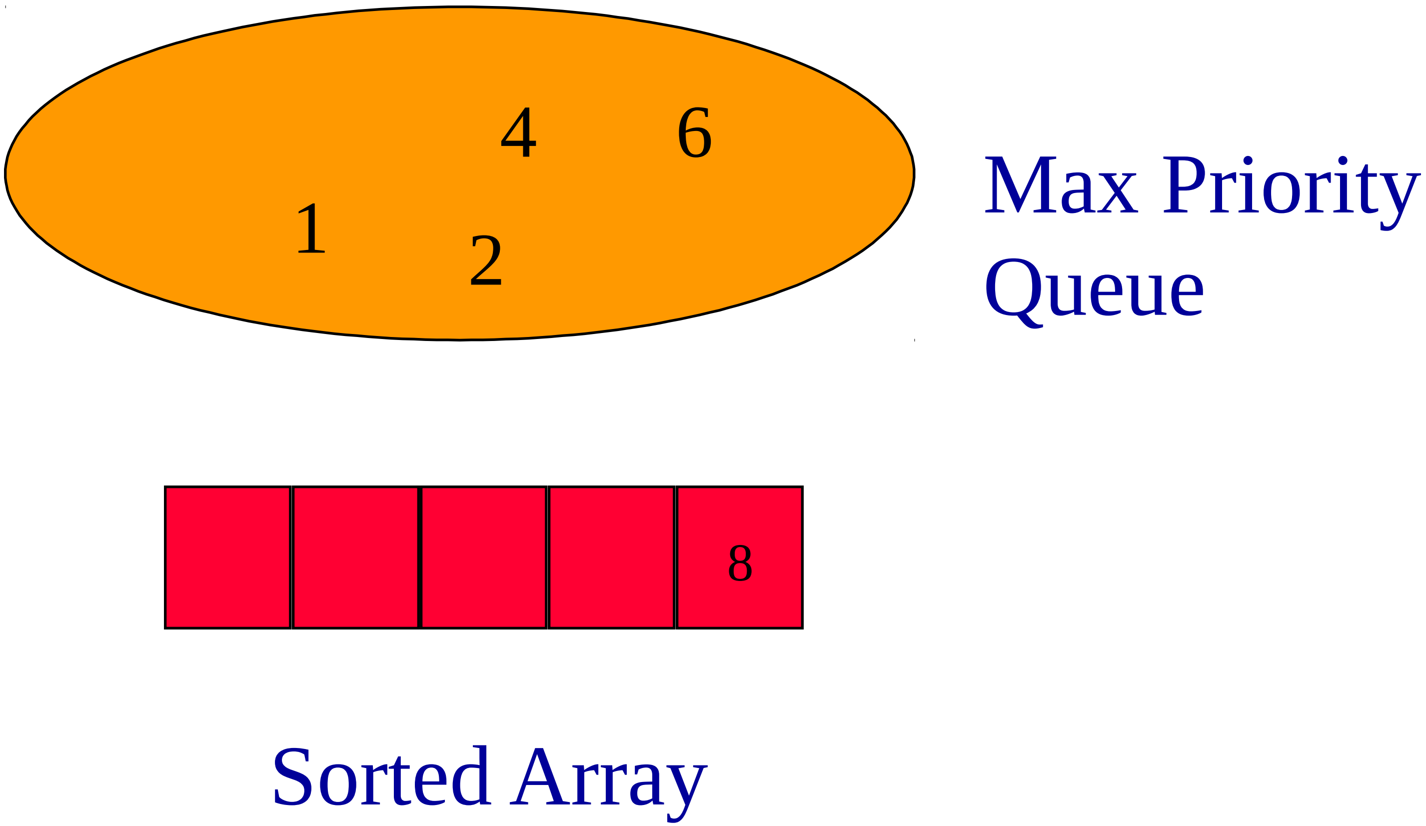
Sort five elements whose keys are 6, 8, 2, 4, 1 using a max priority queue.

- Insert the five elements into a max priority queue.
- Do five remove max operations placing removed elements into the sorted array from right to left.

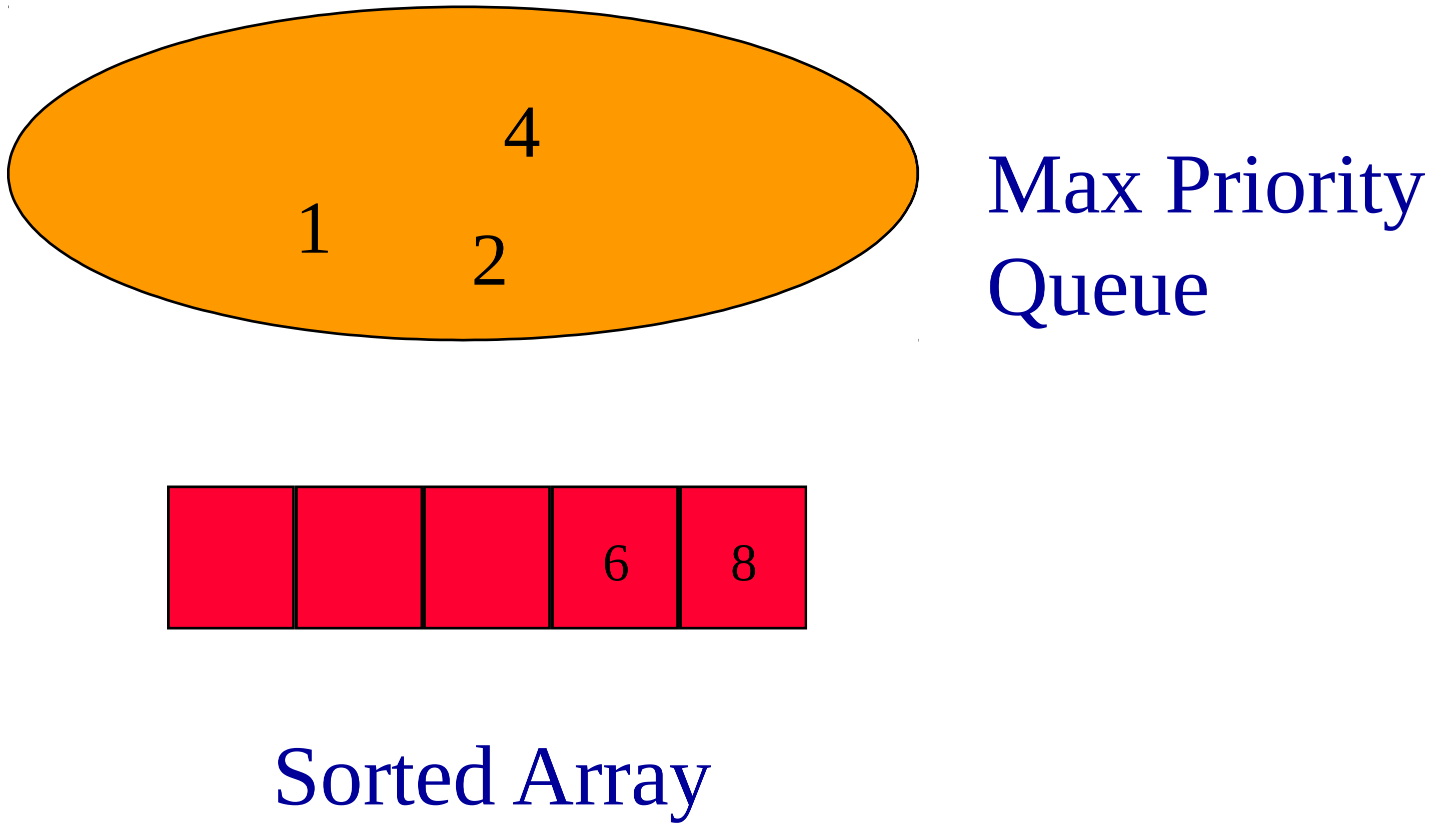
After Inserting Into Max Priority Queue



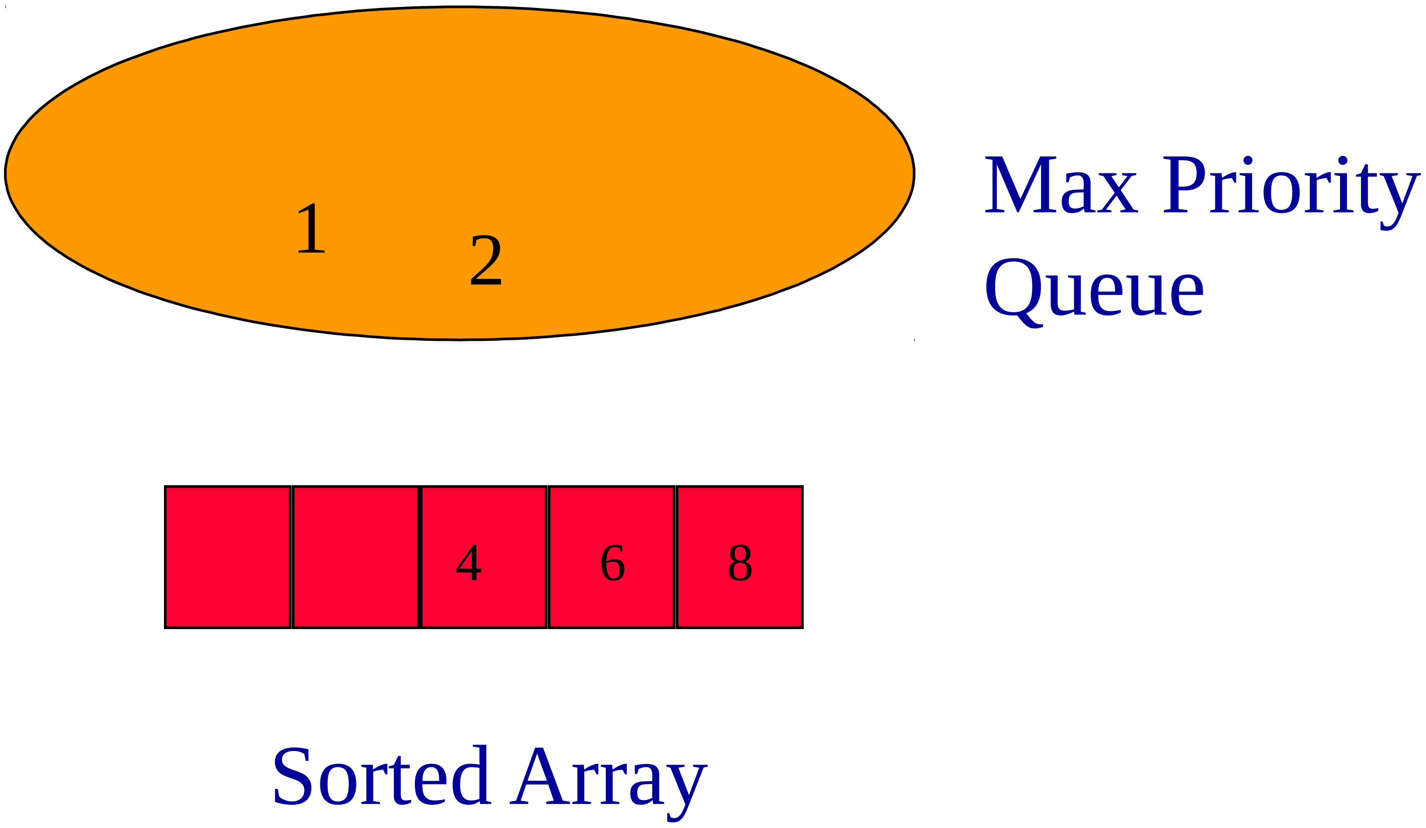
After First Remove Max Operation



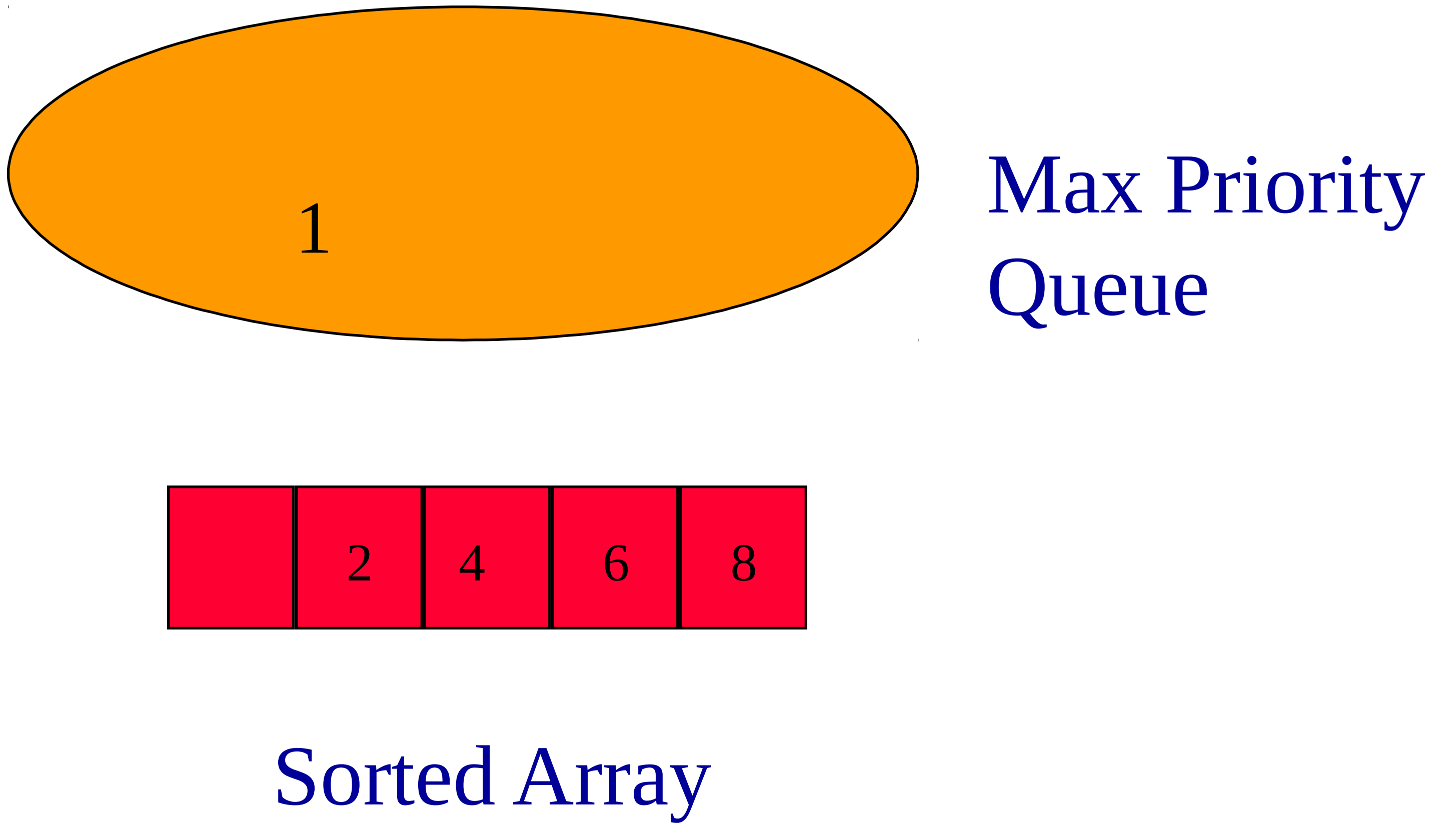
After Second Remove Max Operation



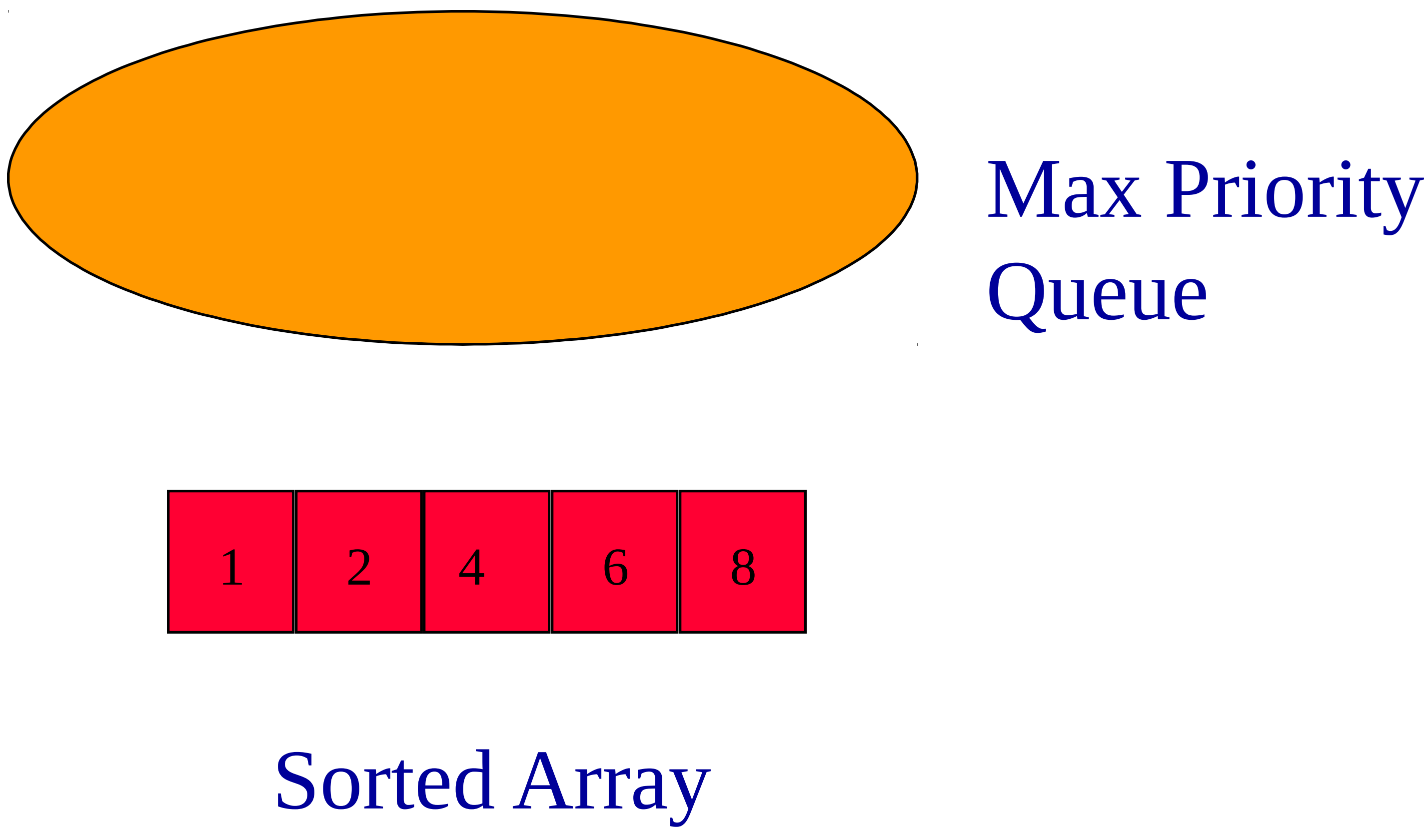
After Third Remove Max Operation



After Fourth Remove Max Operation



After Fifth Remove Max Operation



Complexity Of Sorting

Sort n elements.

- n insert operations => $O(n \log n)$ time.
- n remove max operations => $O(n \log n)$ time.
- total time is $O(n \log n)$.
- compare with $O(n^2)$ for insertion sort.

Machine Scheduling

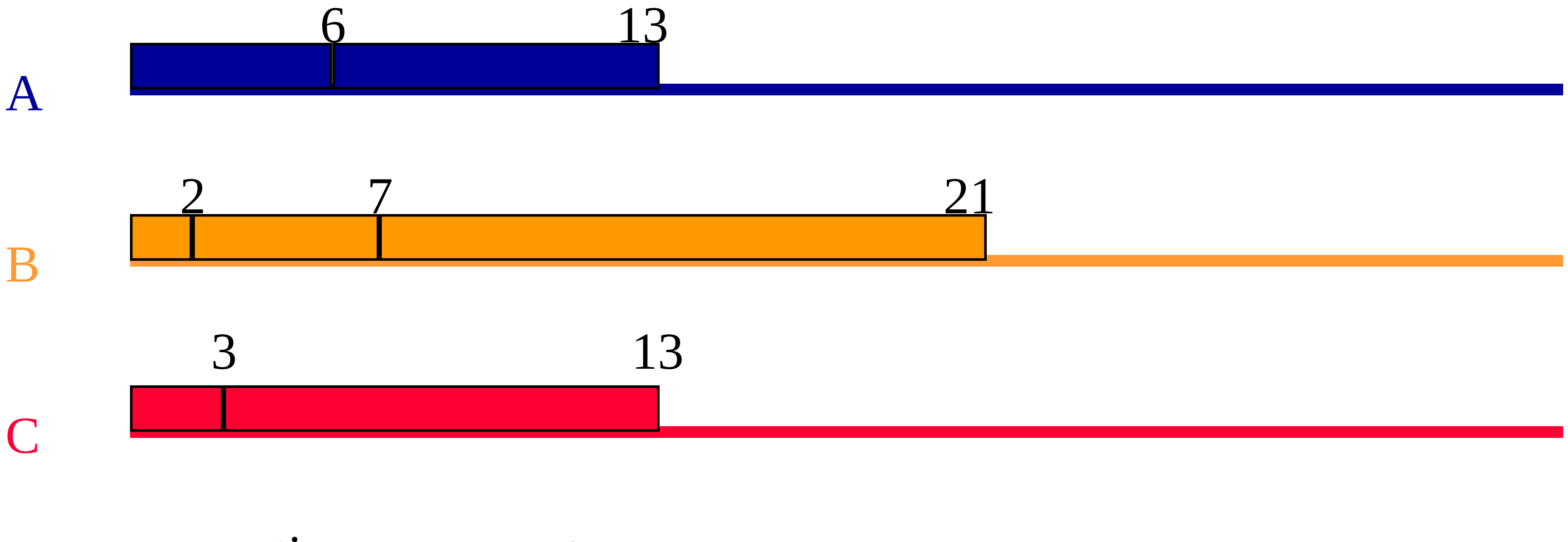
- m identical machines
- n jobs/tasks to be performed
- assign jobs to machines so that the time at which the last job completes is minimum

Machine Scheduling Example

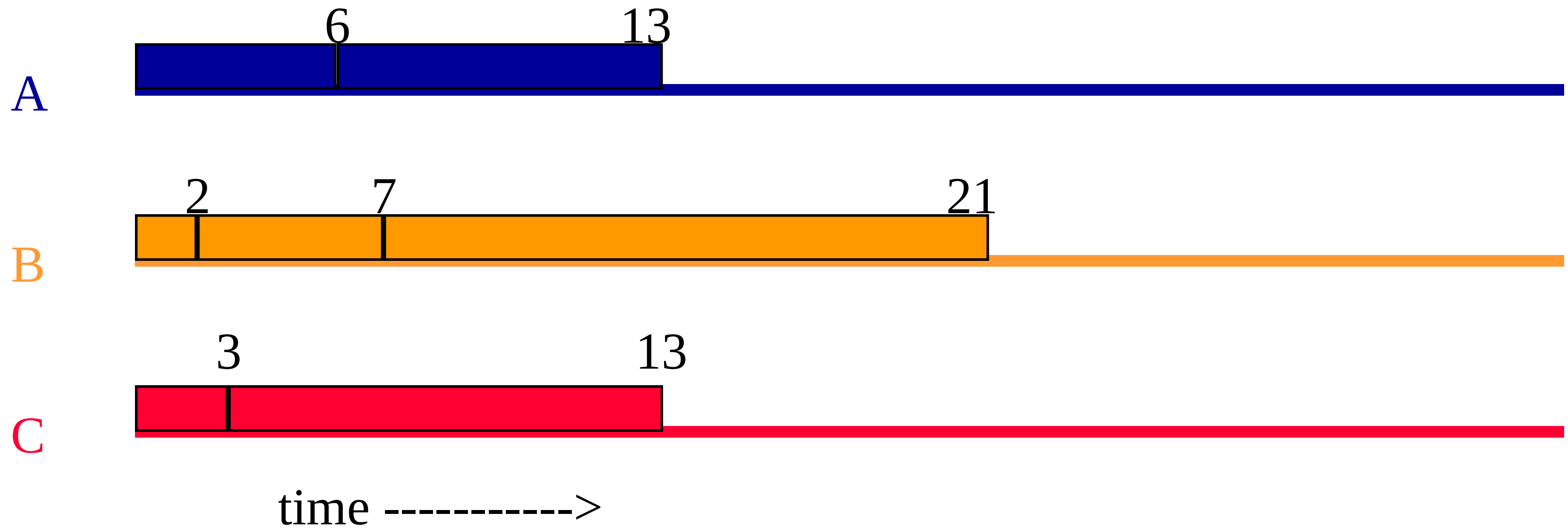
3 machines and 7 jobs

job times are [6, 2, 3, 5, 10, 7, 14]

possible schedule



Machine Scheduling Example



Finish time = 21

Objective: Find schedules with minimum finish time.

LPT Schedules

Longest Processing Time first.

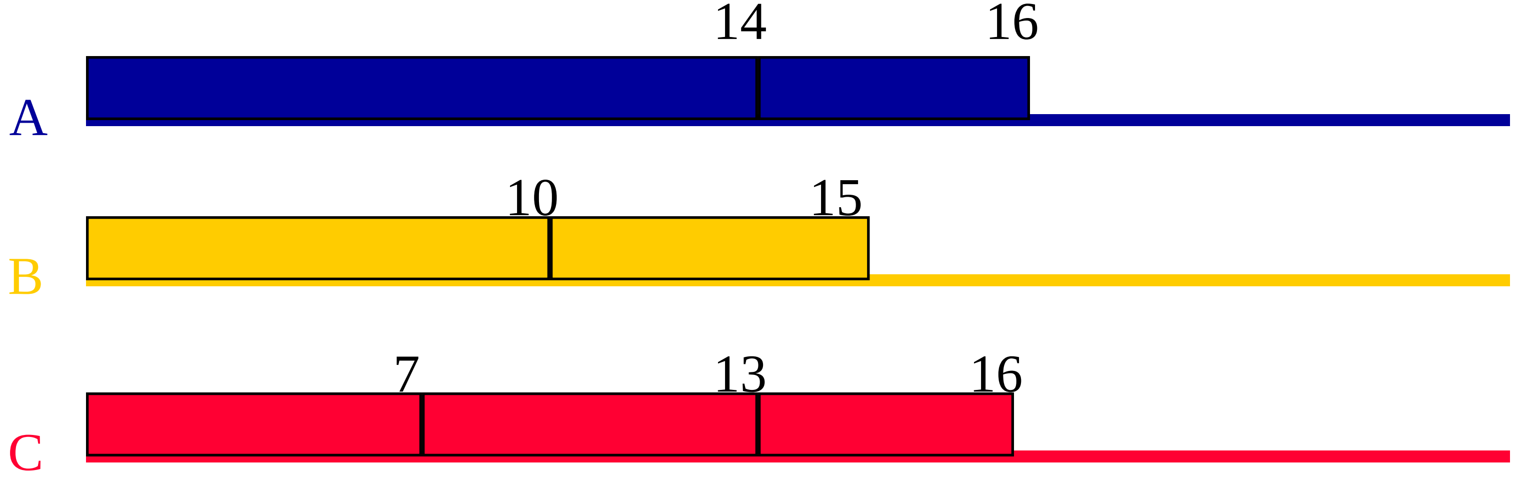
Jobs are scheduled in the order

14, 10, 7, 6, 5, 3, 2

Each job is scheduled on the machine on which it finishes earliest.

LPT Schedule

[14, 10, 7, 6, 5, 3, 2]



Finish time is 16!

LPT Schedule

- LPT rule does not guarantee minimum finish time schedules.
- (LPT Finish Time)/(Minimum Finish Time) <= 4/3 - 1/(3m) where m is number of machines.
- Usually LPT finish time is much closer to minimum finish time.
- Minimum finish time scheduling is NP-hard.

NP-hard Problems

- Infamous class of problems for which no one has developed a polynomial time algorithm.
- That is, no algorithm whose complexity is $O(n^k)$ for any constant k is known for any NP-hard problem.
- The class includes thousands of real-world problems.
- Highly unlikely that any NP-hard problem can be solved by a polynomial time algorithm.

NP-hard Problems

- Since even polynomial time algorithms with degree $k > 3$ (say) are not practical for large n, we must change our expectations of the algorithm that is used.
- Usually develop fast heuristics for NP-hard problems.
  - Algorithm that gives a solution close to best.
  - Runs in acceptable amount of time.
- LPT rule is good heuristic for minimum finish time scheduling.

Complexity Of LPT Scheduling

- Sort jobs into decreasing order of task time.
  - O(n log n) time (n is number of jobs)
- Schedule jobs in this order.
  - assign job to machine that becomes available first
  - must find minimum of m (m is number of machines) finish times
  - takes O(m) time using simple strategy
  - so need O(mn) time to schedule all n jobs.

Using A Min Priority Queue

- Min priority queue has the finish times of the m machines.
- Initial finish times are all 0.
- To schedule a job remove machine with minimum finish time from the priority queue.
- Update the finish time of the selected machine and insert the machine back into the priority queue.

Using A Min Priority Queue

- m put operations to initialize priority queue
- 1 remove min and 1 insert to schedule each job
- each insert and remove min operation takes $O(\log m)$ time
- time to schedule is $O(n \log m)$
- overall time is

$$O(n \log n + n \log m) = O(n \log (mn))$$

Skew Heap

- Similar to leftist tree
- No s() values stored
- Swap left and right subtrees of all nodes on rightmost path rather than just when $s(l(x)) < s(r(x))$
- Amortized complexity of insert, remove min, meld is $O(\log n)$