Lecture 7

Heap Sort

Md. Asif Bin Khaled

Heap Sort

Like merge sort, but unlike insertion sort, heap sort’s running time is O(nlgn). Moreover, like insertion sort, but unlike merge sort, heapsort sorts in place. Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a “heap,” to manage information. Not only is the heap data structure useful for heapsort, but it also makes an efficient priority queue.

Binary Heap Data Structure

The (binary) heap data structure is an array object that we can view as a
nearly complete binary tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A.

Root, Left & Right Pseudocode

```
Parent(i):
    return ⌊i/2⌋

Left(i):
    //2*i+1 for 0 based array
    return 2*i

Right(i):
    //2*i+2 for 0 based array
    return 2*i+1
```

Types of Binary Heaps

There are two kinds of binary heaps,

1. **Max-Heap:** A[Parent(i)] $\geq$ A[i]

2. **Min-Heap:** A[Parent(i)] $\leq$ A[i]

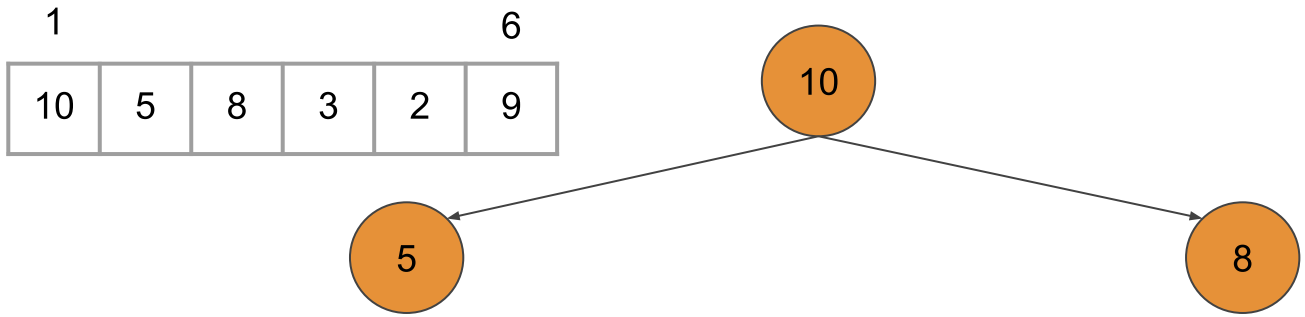
In a max-heap, the max-heap property is that for every node i other than the root, that is, the value of a node is at most the value of its parent. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.

Types of Binary Heaps Continued.

A min-heap is organized in the opposite way, the min-heap property is that for every node i other than the root, The smallest element in a min-heap is at the root. For the heapsort algorithm, we use max-heaps. Min-heaps commonly implement priority queues.

Viewing a heap as a tree, we define the height of a node in a heap to be the
number of edges on the longest simple downward path from the node to a leaf, and we define the height of the heap to be the height of its root. Since a heap of n elements is based on a complete binary tree, its height is, O(lgn)

Root, Left & Right Pseudocode



Array Size: 6

Heap Size: 3

Parent(2) is 1 hence 10

Left(1) is 2 hence 5

Right(1) is 3 hence 8

Maintaining the Heap Property

In order to maintain the max-heap property, we call the procedure Max_Heapify.

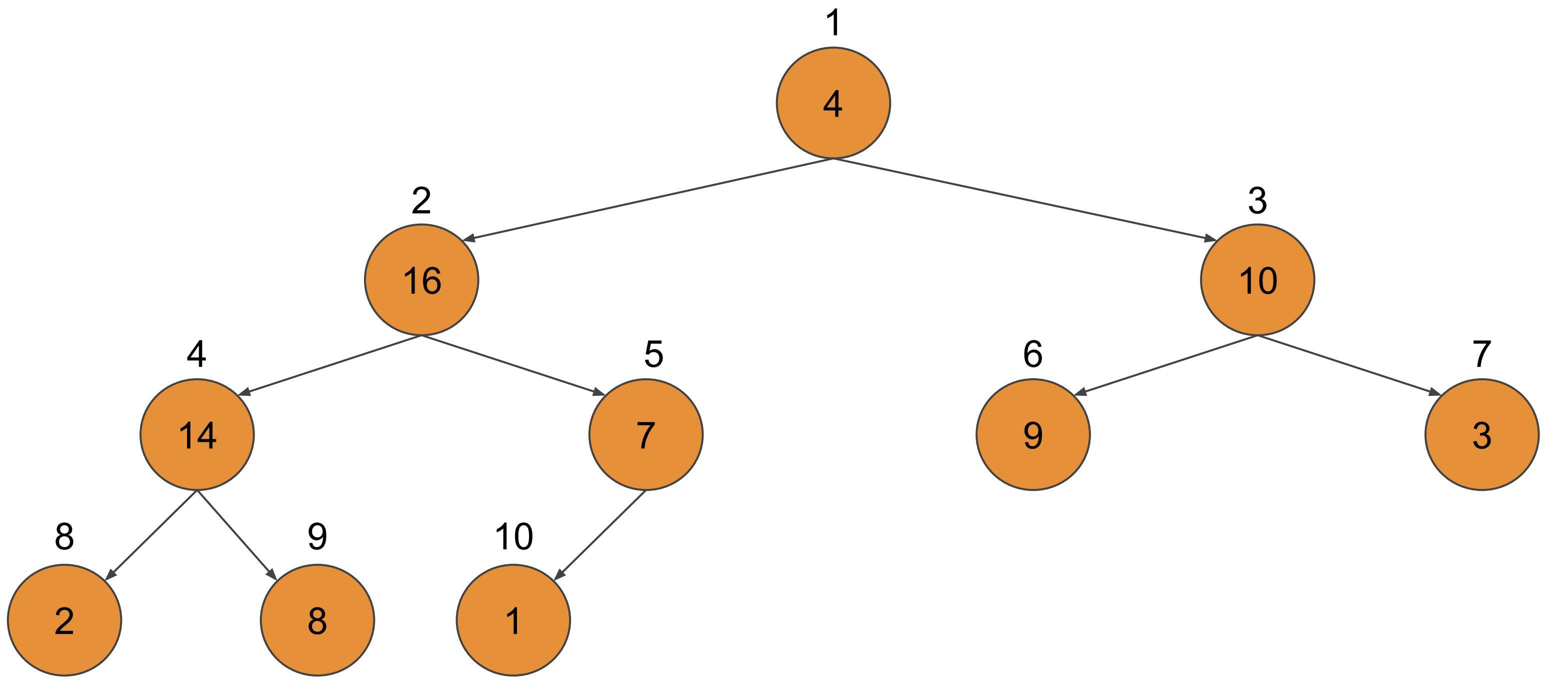
Its inputs are an array A and an index i into the array. When it is called, Max_Heapify assumes that the binary trees rooted at Left(i) and Right(i) are max heaps, but that A[i] might be smaller than its children, thus violating the max-heap property. Max_Heapify lets the value at A[i] “float down” in the max-heap so that the subtree rooted at index i obeys the max-heap property.

Opposite rules will be applied in the case of Min_Heapify.

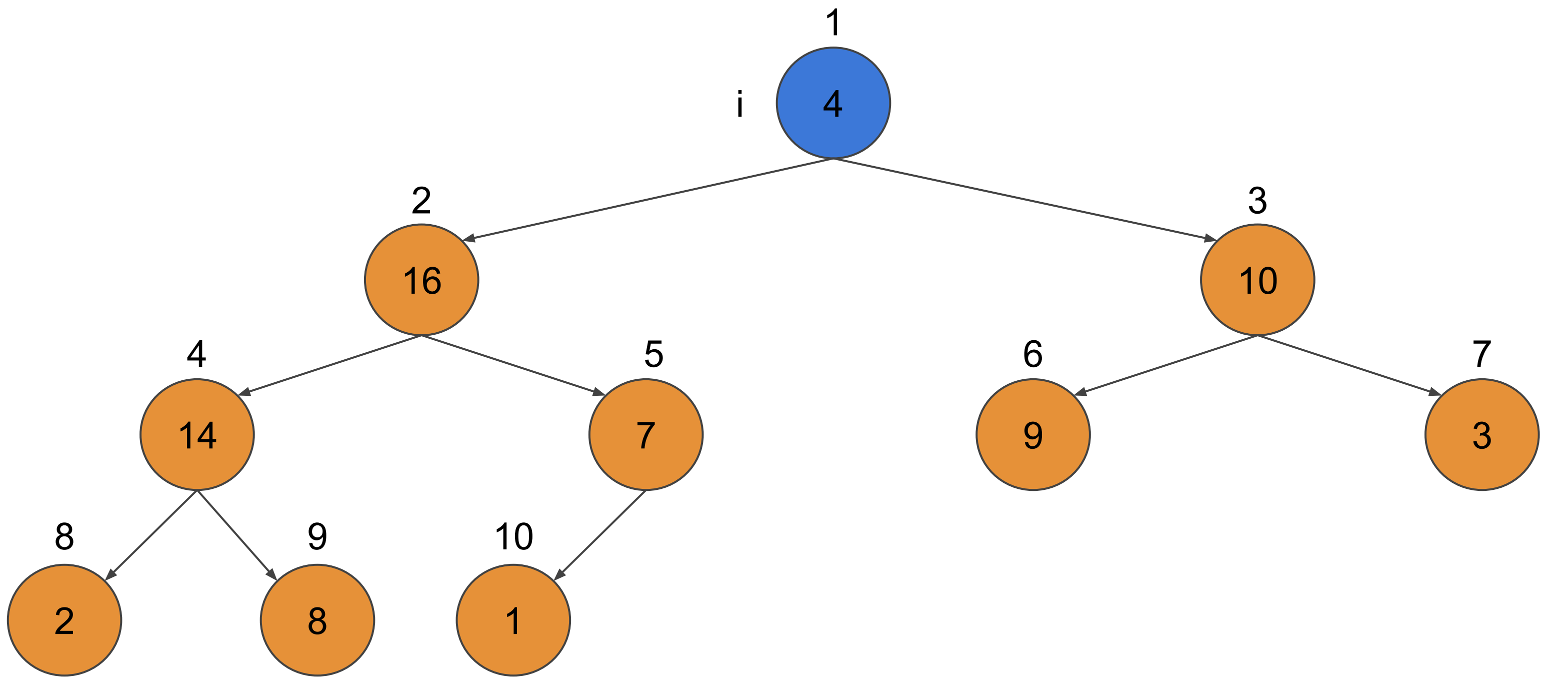
Max Heapify Pseudocode

```
Max_Heapify(A,i):
    l = Left(i)
    r = Right(i)
    if l ≤ A.heap_size and A[l] > A[i]:
        largest = l
    else
        largest = i
    if r ≤ A.heap_size and A[r] > A[largest]:
        largest = r
    if largest != i:
        exchange A[i] with A[largest]
        Max_Heapify(A, largest)
```

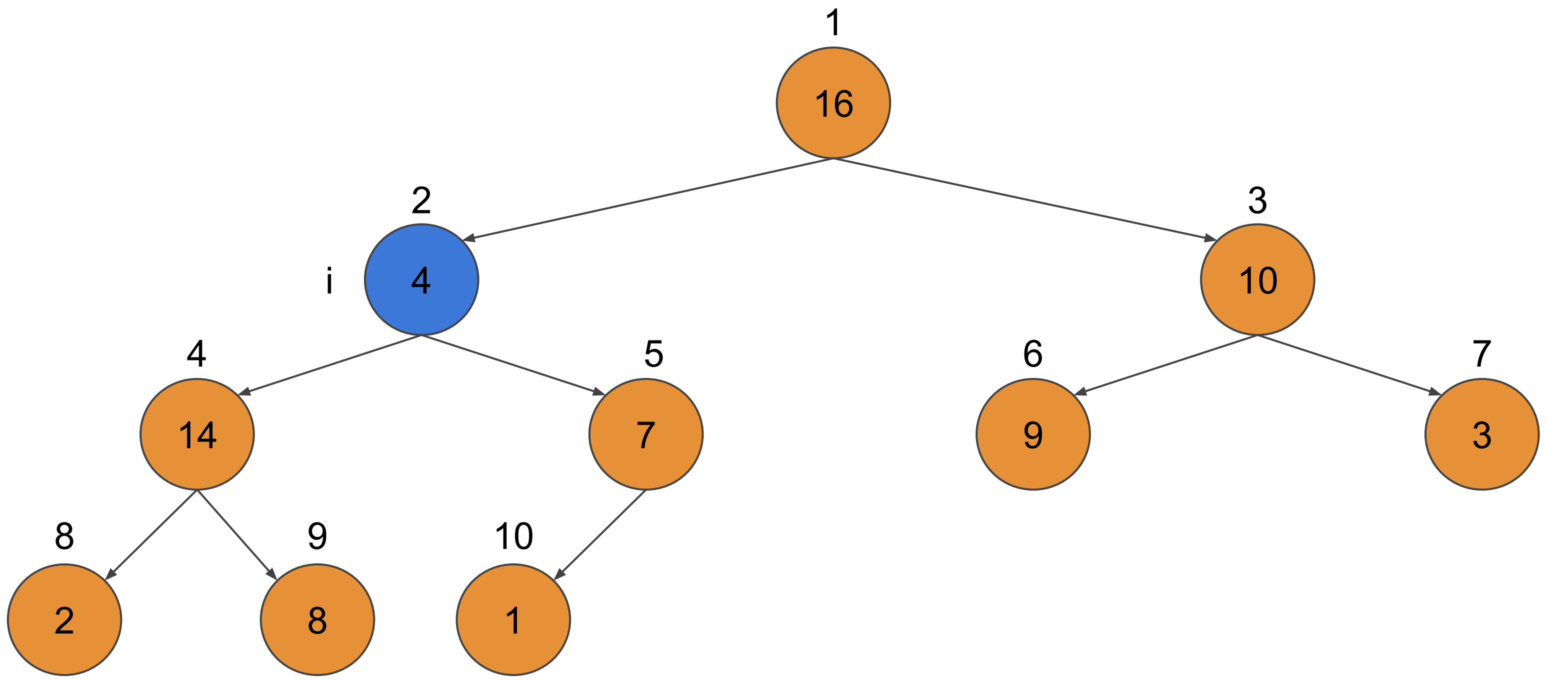
Simulation of Max Heapify



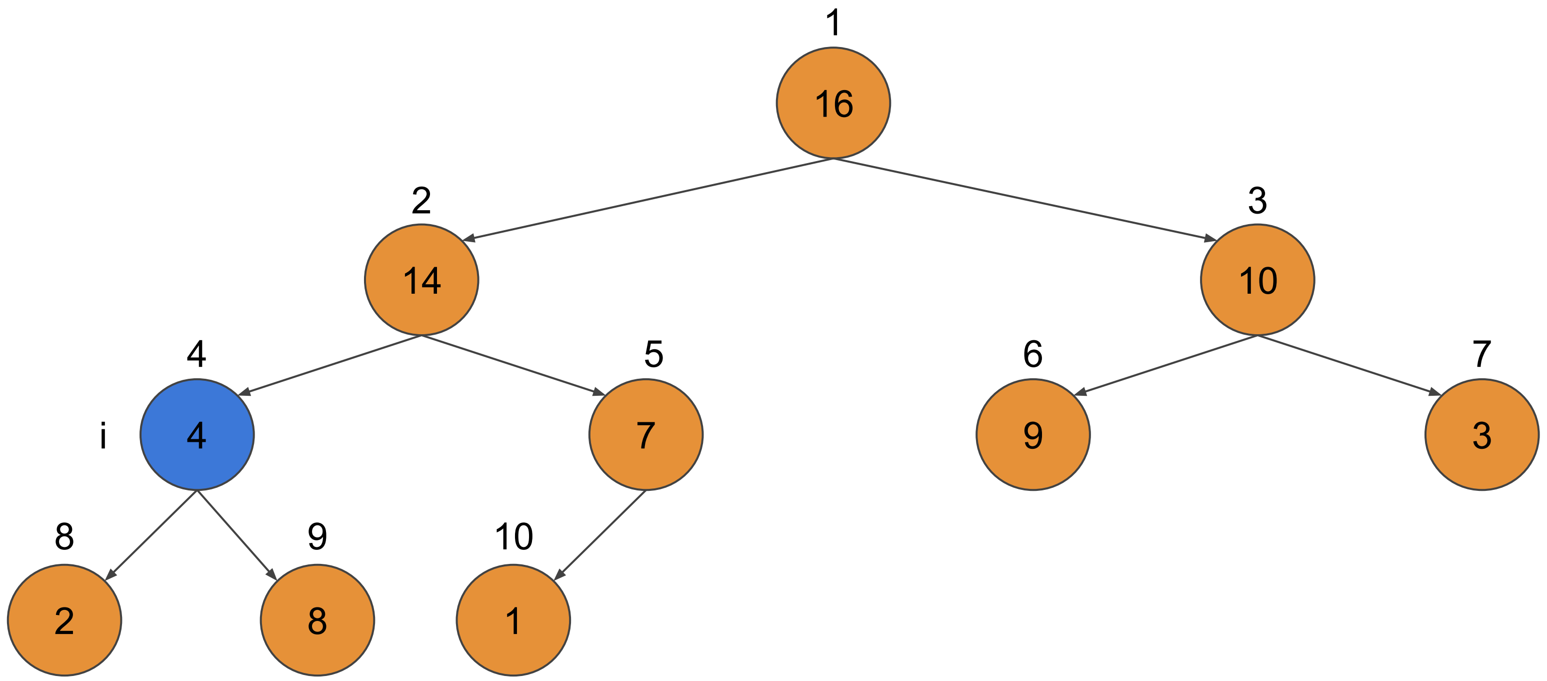
Simulation of Max Heapify Continued



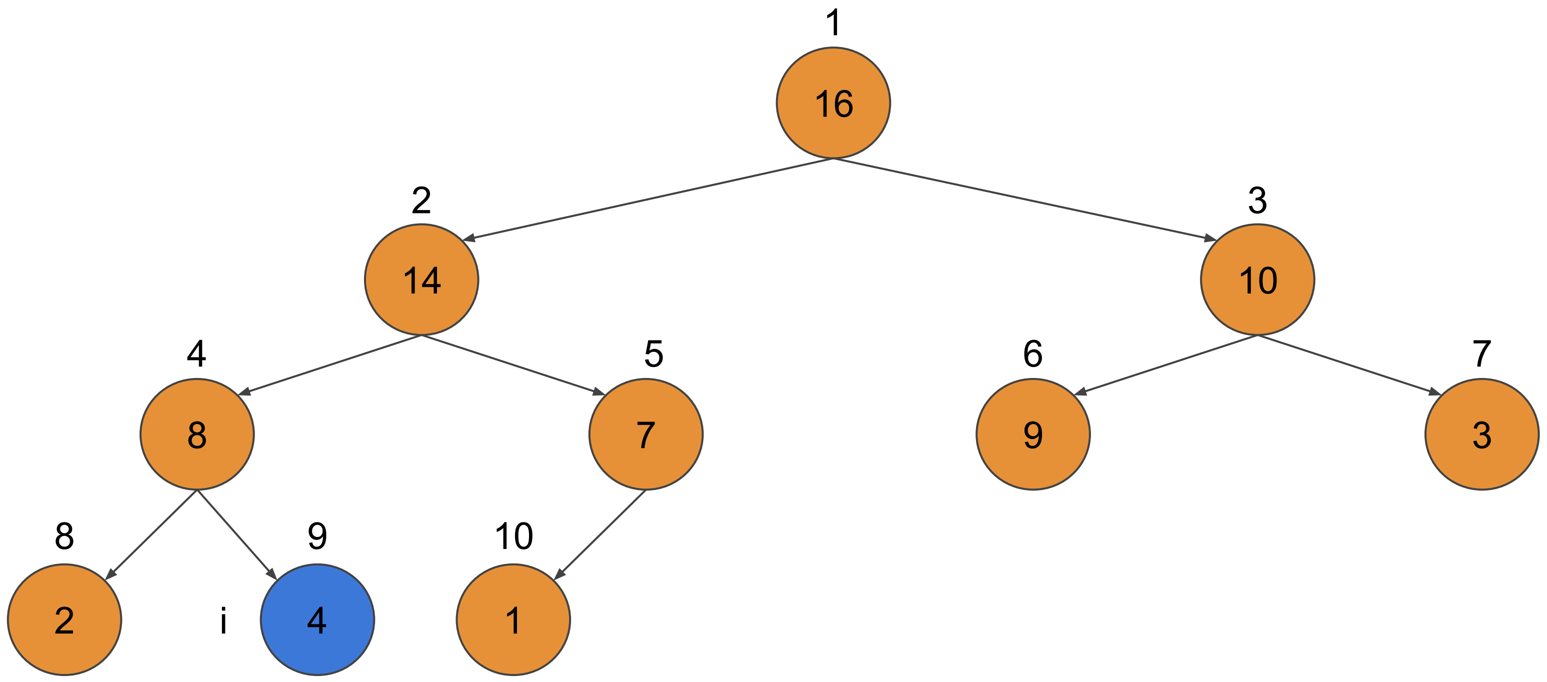
Simulation of Max Heapify Continued



Simulation of Max Heapify Continued



Simulation of Max Heapify Continued



Running Time of Max Heapify

The running time of Max_Heapify on a subtree of size n rooted at a given node i is the Θ(1) time to fix up the relationships among the elements A[i] A[Left(i)] and A[Right(i)] plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children’s subtrees each have size at most 2n/3 and the worst case occurs when the bottom level of the tree is exactly half full and therefore we can describe the running time of Max_Heapify by the recurrence,

$$T(n) \leq T(2n/3) + \Theta(1)$$

Complexity Analysis of Max Heapify Continued

$T(n) \leq T(2n/3) + \Theta(1)$

$\leq T(n/(3/2)) + c$

$\leq T(n/(3/2)^2) + c + c$ [Substitution]

$\leq T(n/(3/2)^3) + c + c + c$ [Substitution]

$\leq T(n/(3/2)^4) + c + c + c + c$ [Substitution]

$\vdots$

$\leq T(n/(3/2)^k) + kc$

Complexity Analysis of Max Heapify Continued

Let,

$n/(3/2)^k = 1$

$n = (3/2)^k$

$lg_{3/2}n = lg_{3/2}(3/2)^k$

$k = lg_{3/2}n$

So,

$T(n) \le T(n/(3/2)^{lg(3/2)\_n}) + lg_{3/2}n*c$

$\le T(1) + lg_{3/2}n*c$

$= \mathbf{O(lg_{3/2}n)}$

$= \mathbf{O(h)}$

Building A Heap

We can use the procedure Max_Heapify in a bottom-up manner to convert an

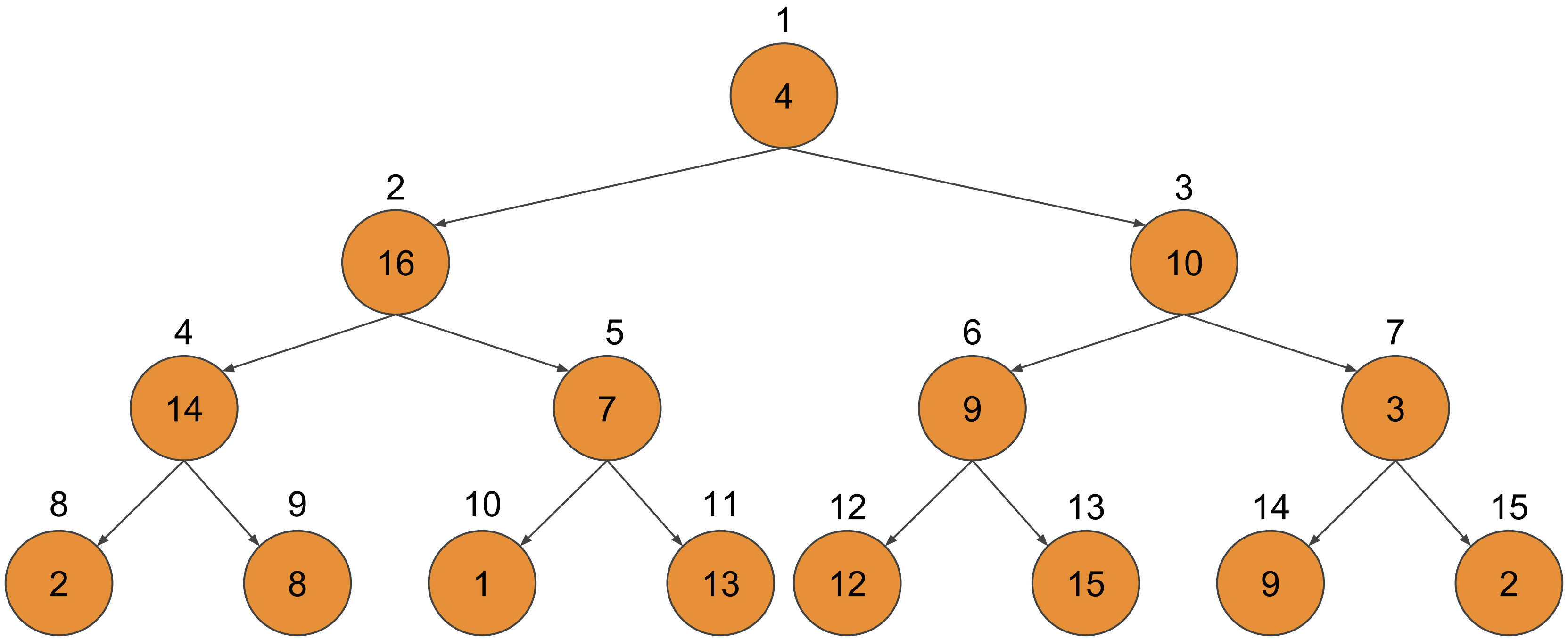
array A[1...n] where n = A.length, into a max-heap. The elements in the subarray A[(⌊n/2⌋ + 1...n] are all leaves of the tree, and so each is a 1-element heap to begin with. The procedure Build_Max_Heap goes through

the remaining nodes of the tree and runs Max_Heapify on each one.

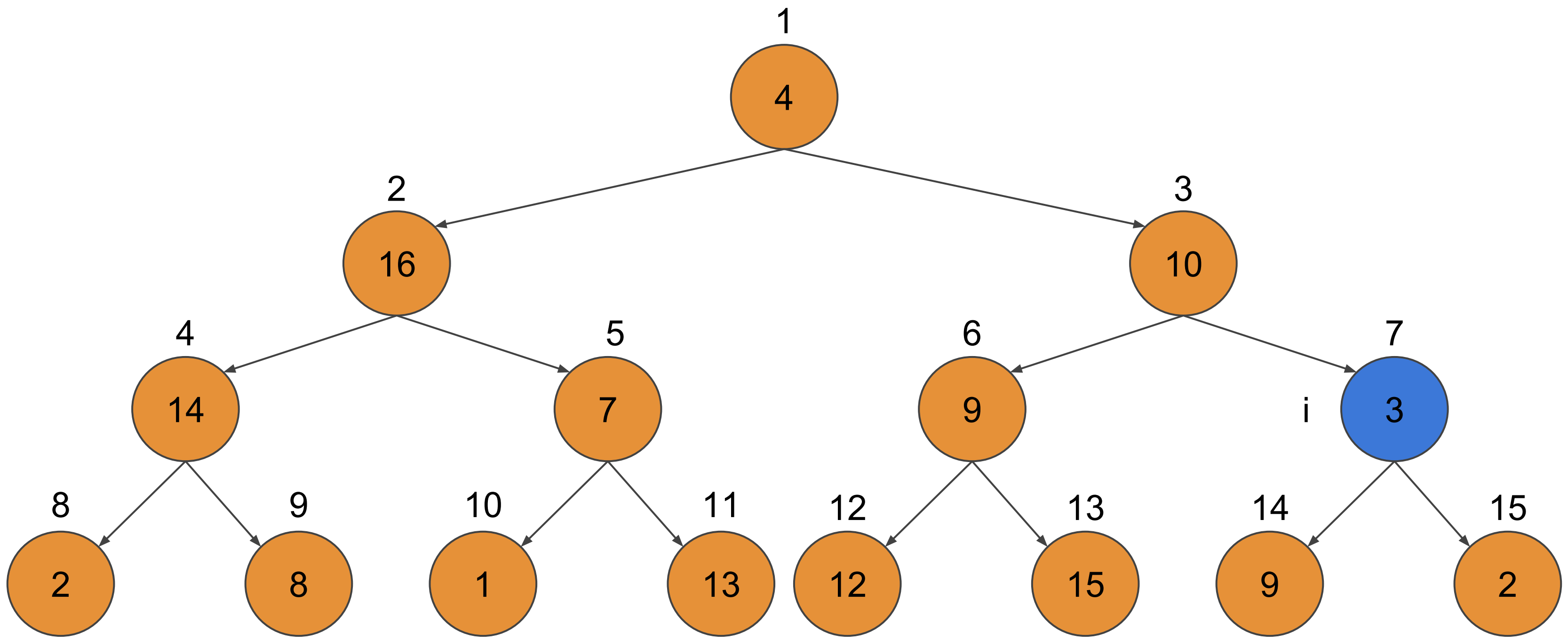
Building Max Heap Pseudocode

```
Build_Max_Heap(A):
    A.heap_size = A.length
    for i = ⌊A.length/2⌋ downto 1:
        Max_Heapify(A,i)
```

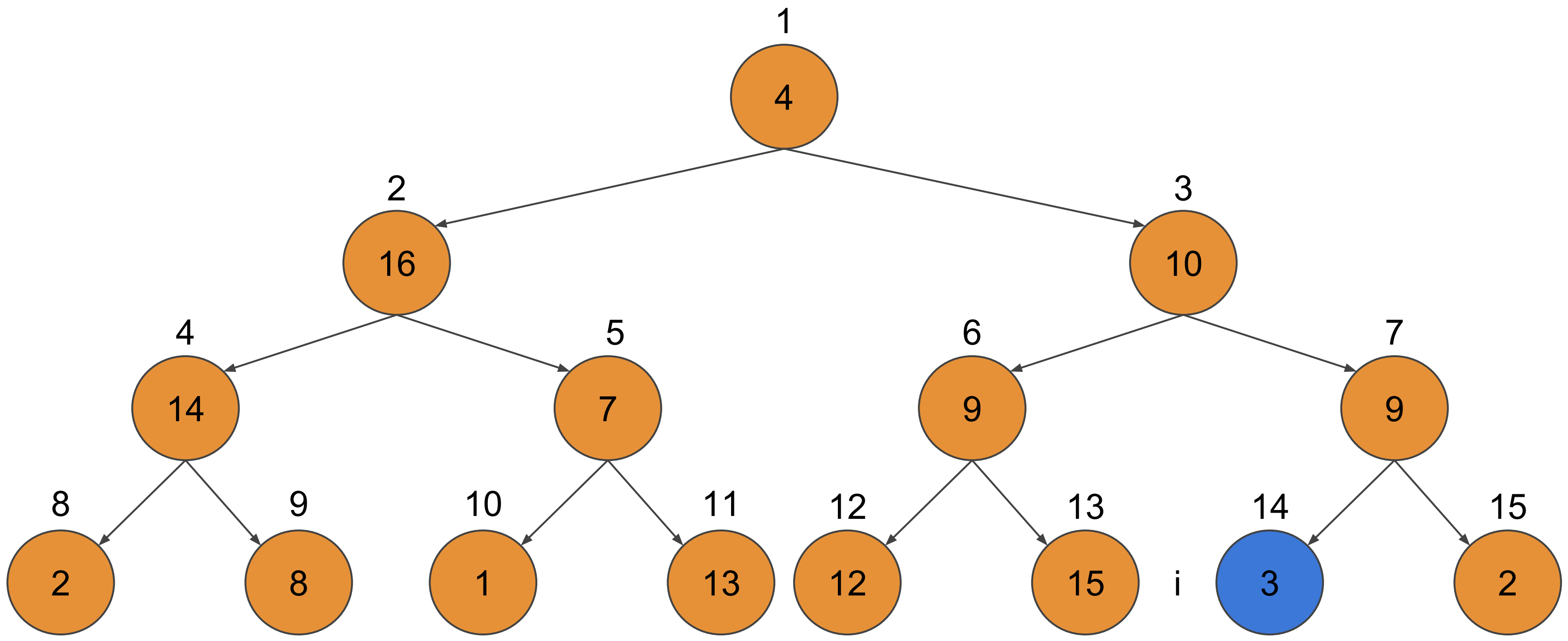
Simulation of Build Max Heap



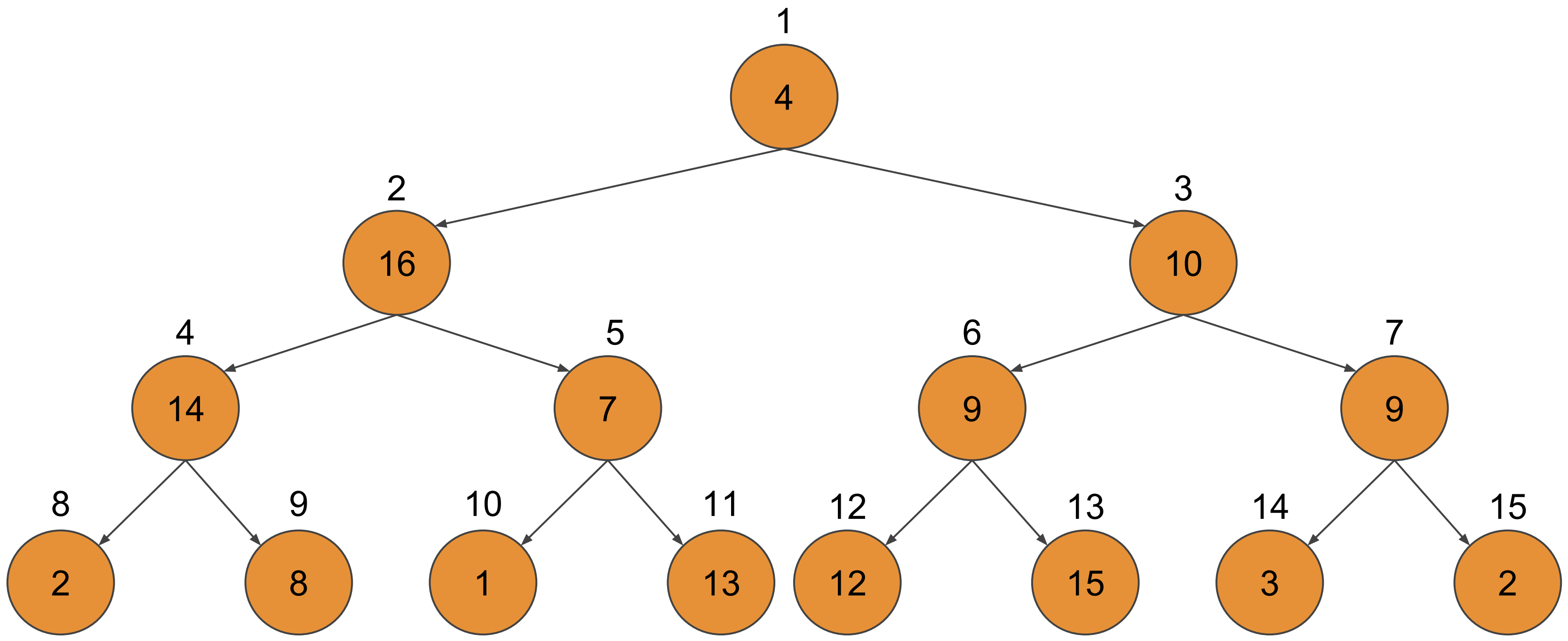
Simulation of Build Max Heap Continued



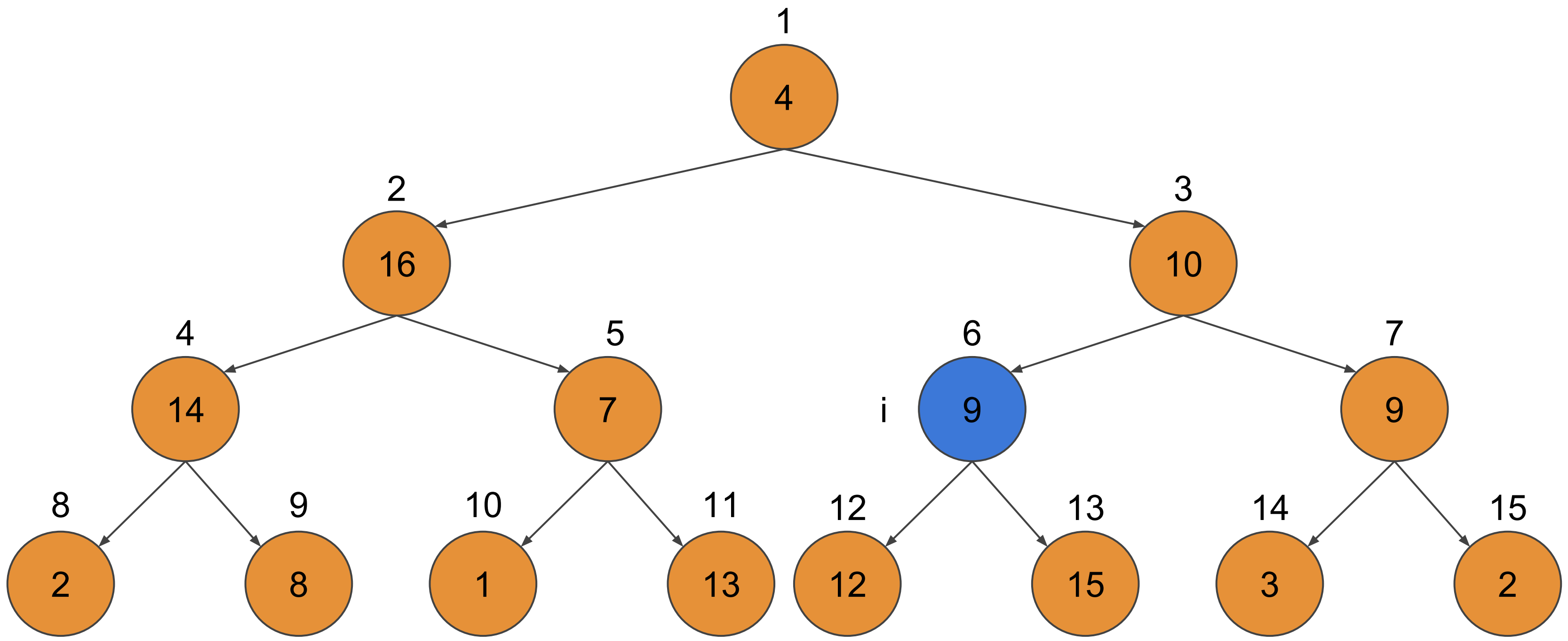
Simulation of Build Max Heap Continued



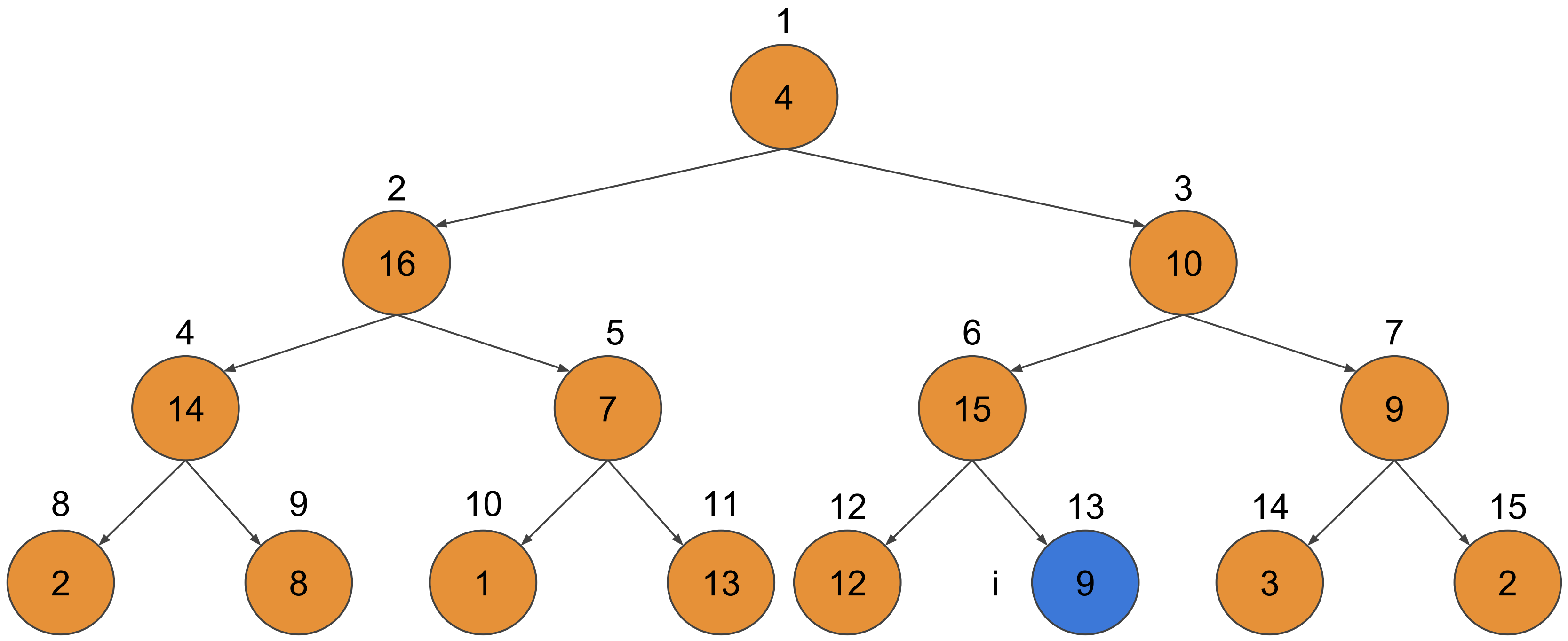
Simulation of Build Max Heap Continued



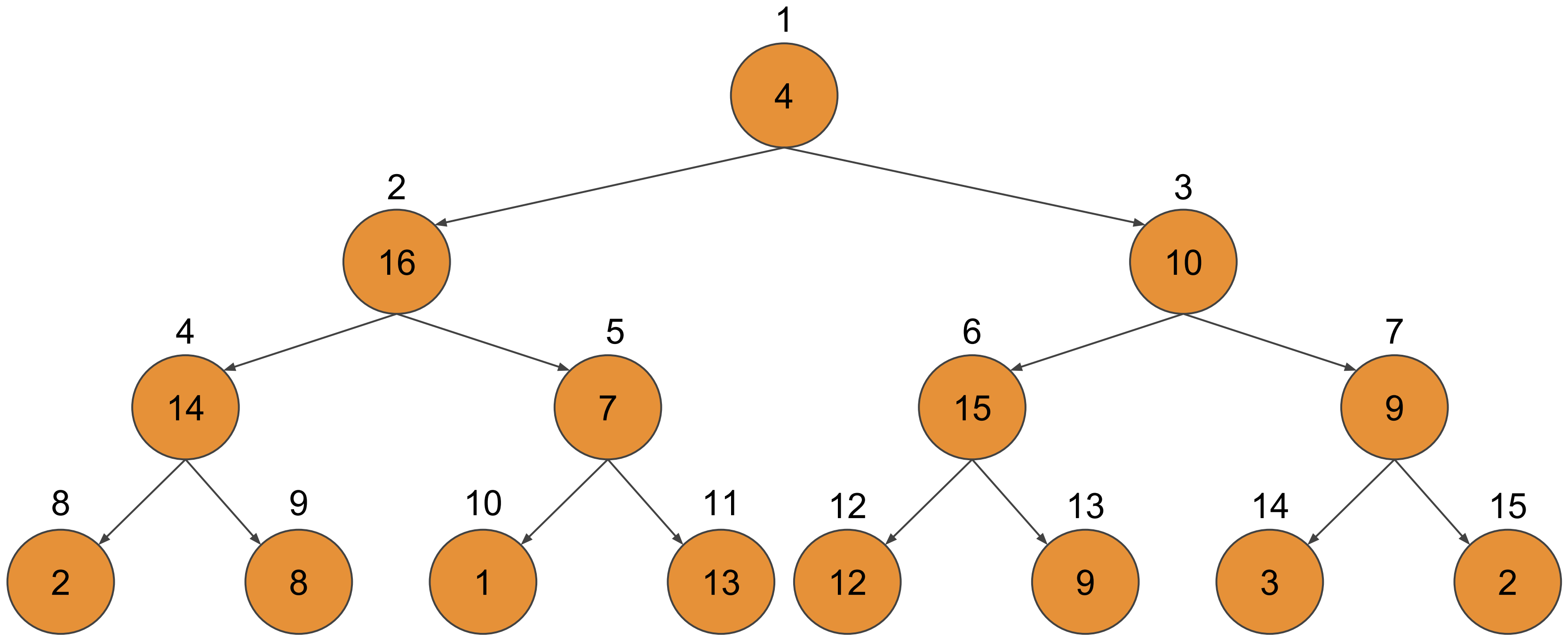
Simulation of Build Max Heap Continued



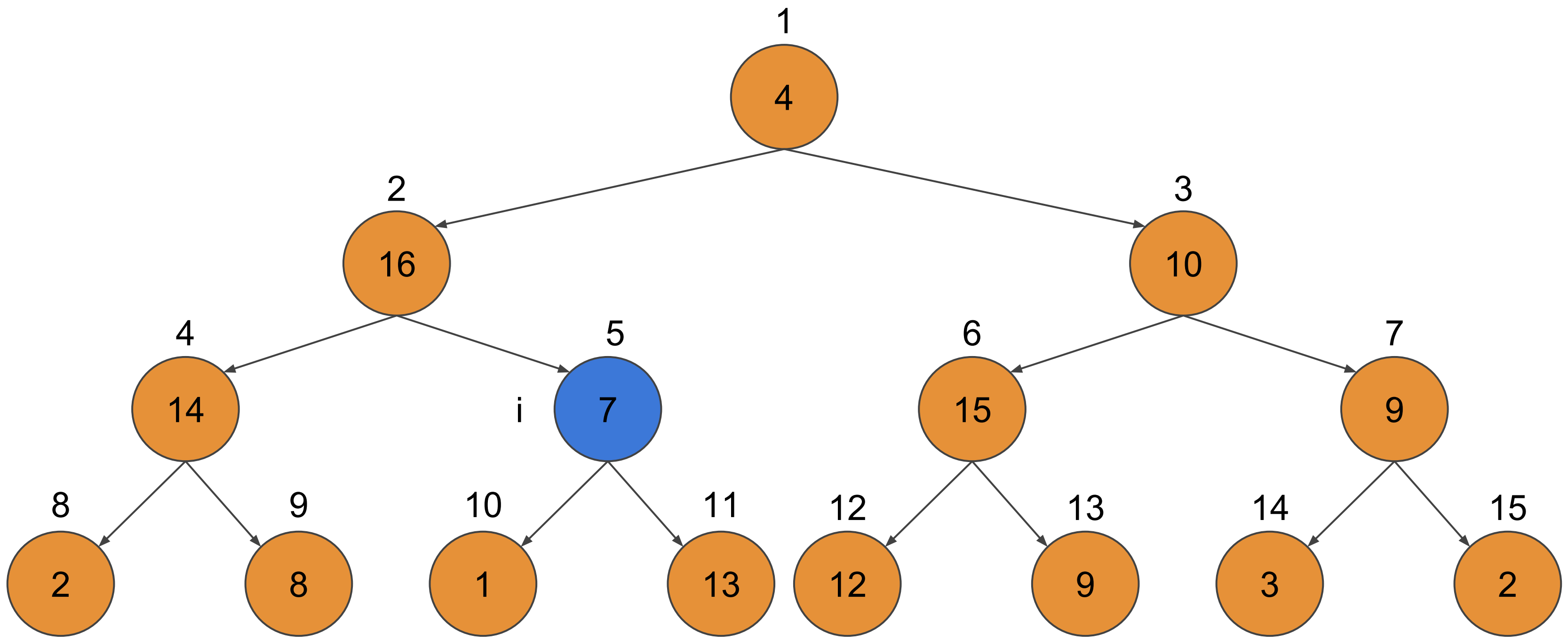
Simulation of Build Max Heap Continued



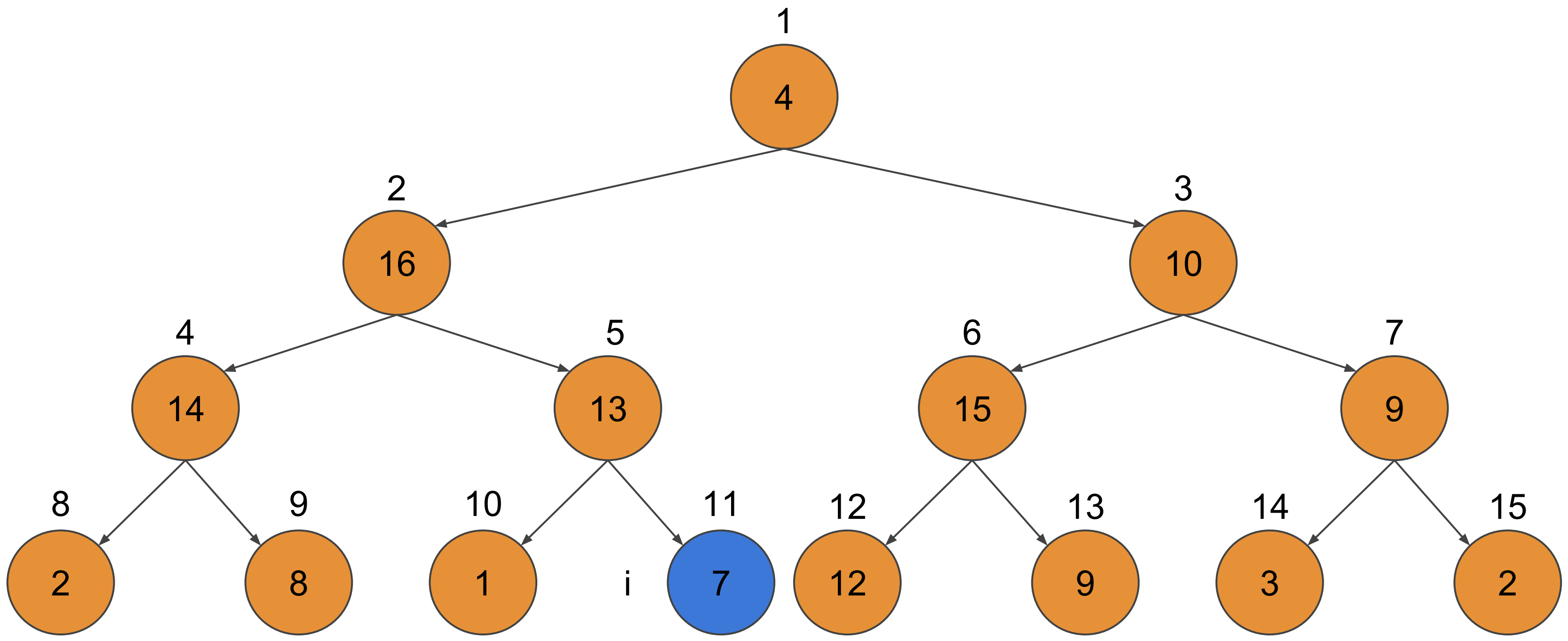
Simulation of Build Max Heap Continued



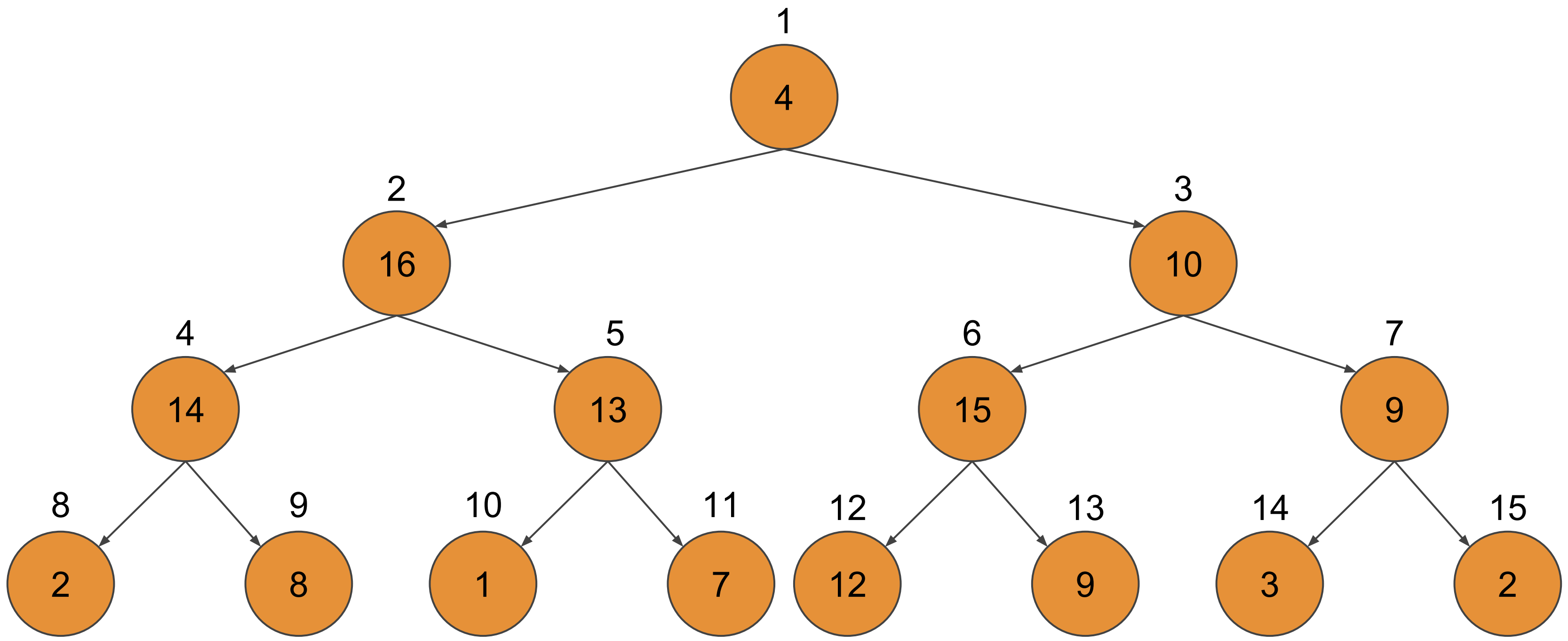
Simulation of Build Max Heap Continued



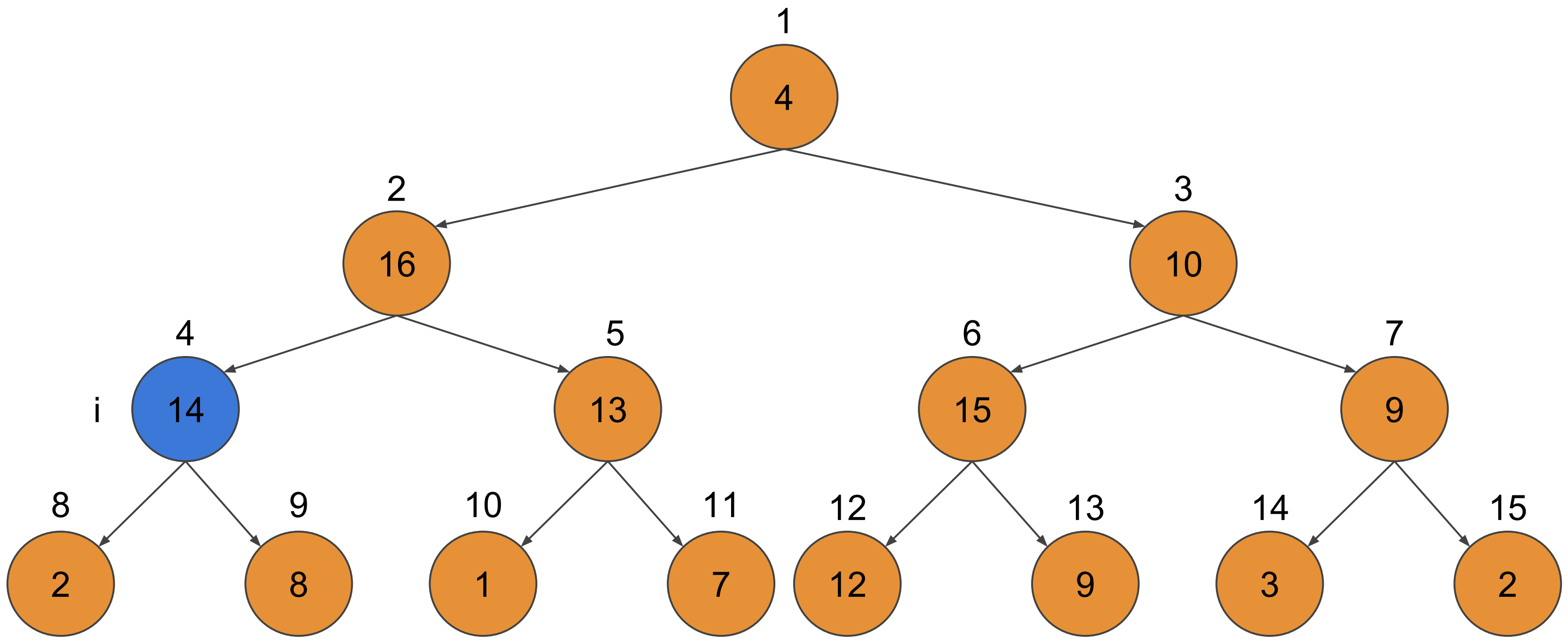
Simulation of Build Max Heap Continued



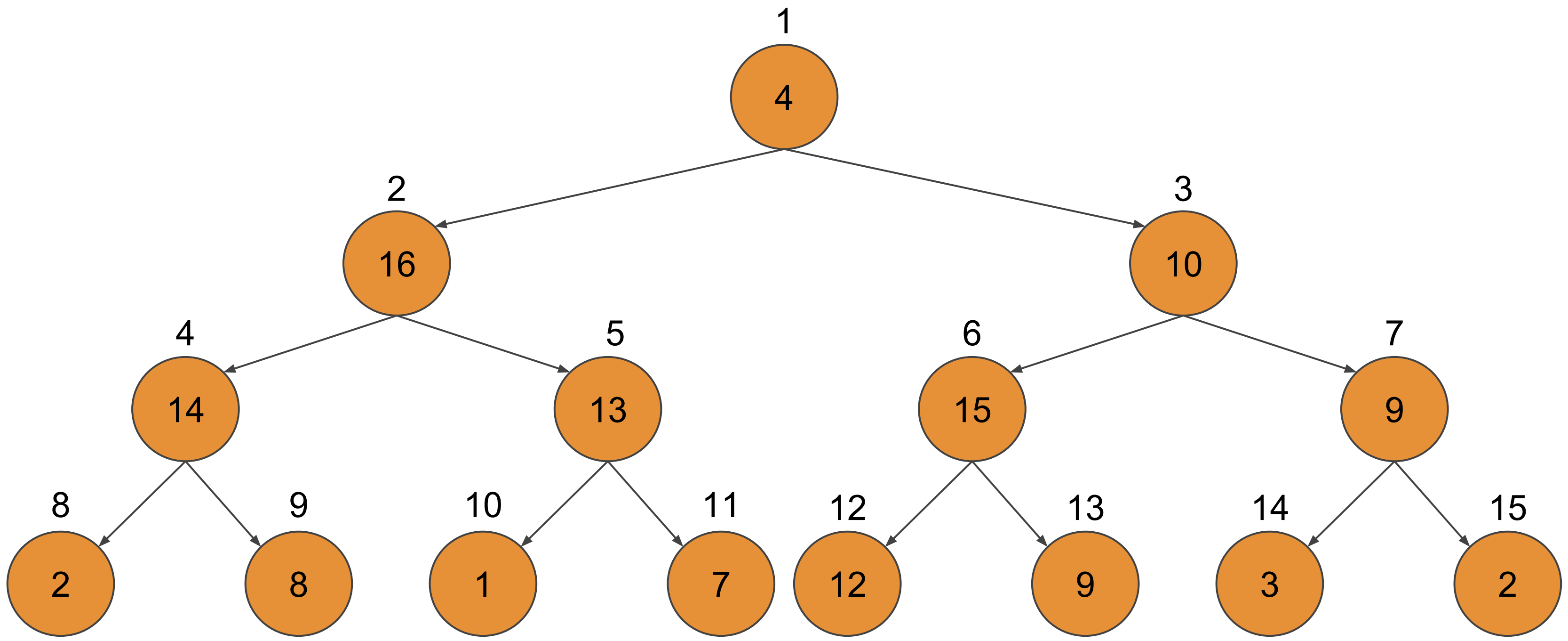
Simulation of Build Max Heap Continued



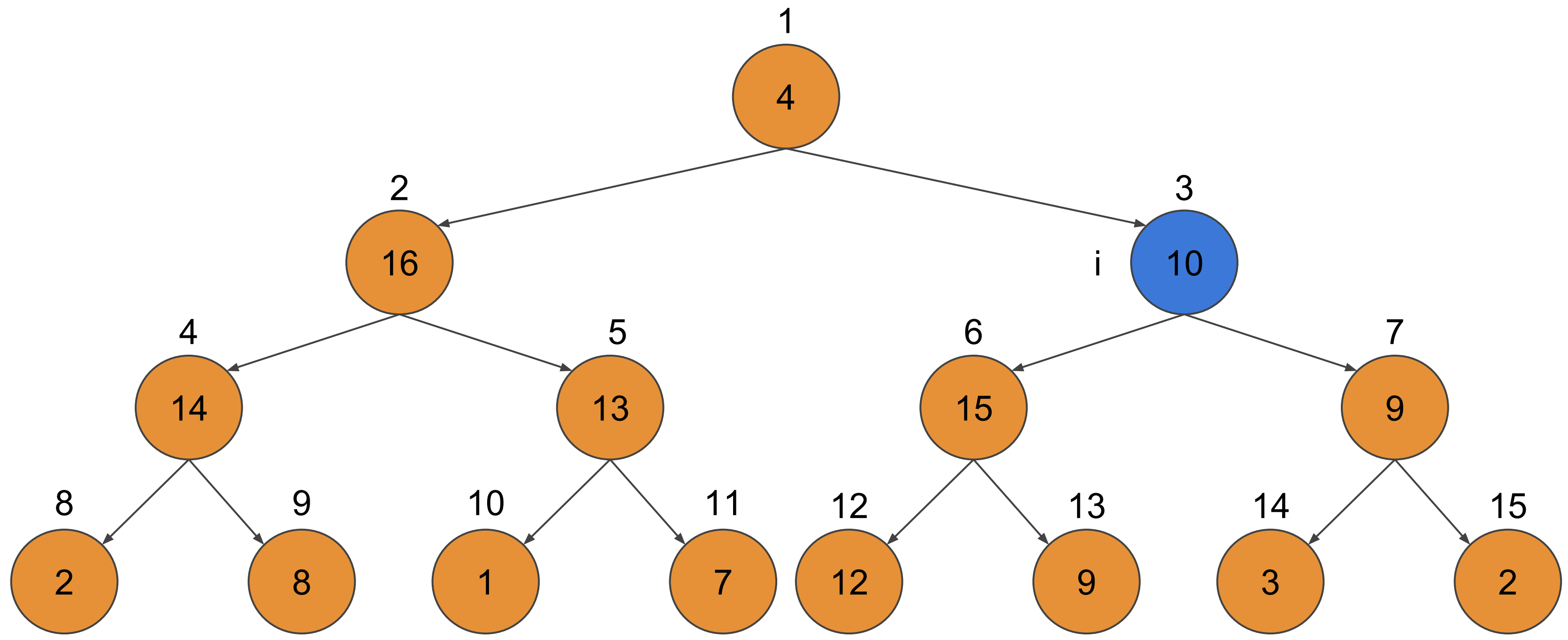
Simulation of Build Max Heap Continued



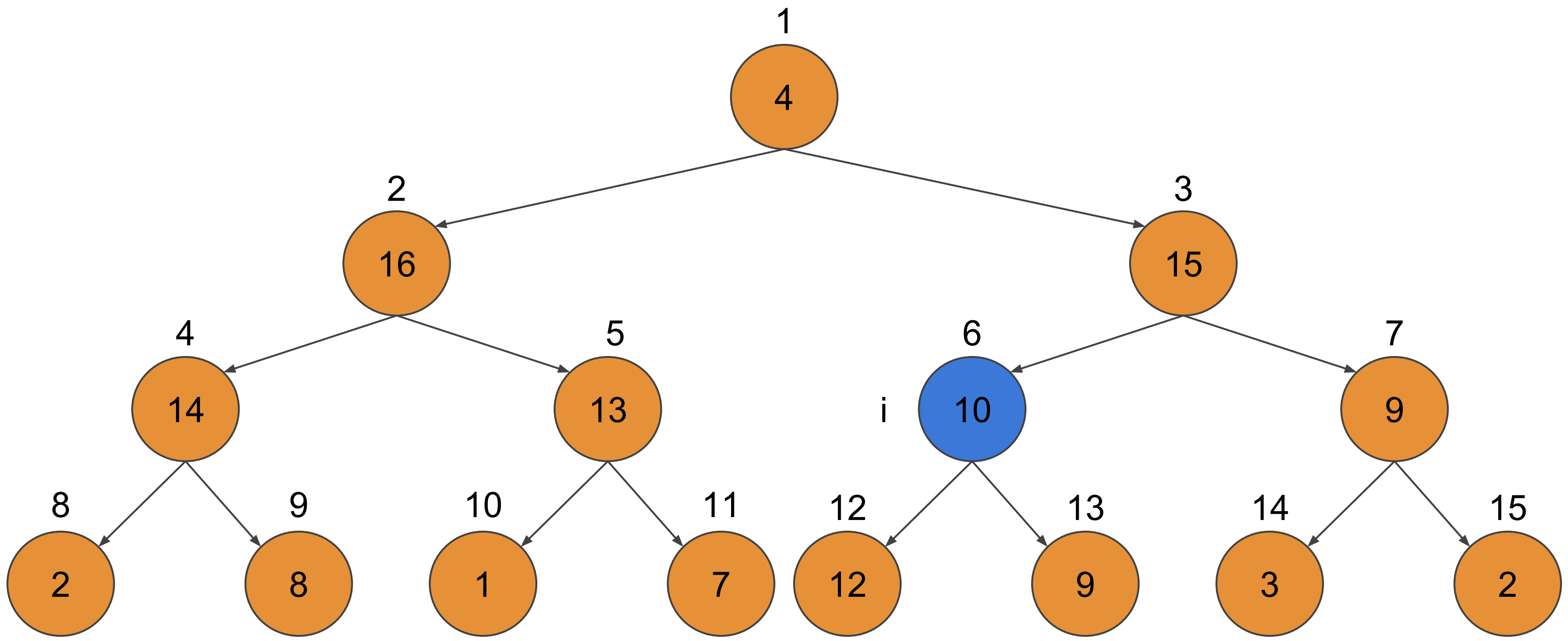
Simulation of Build Max Heap Continued



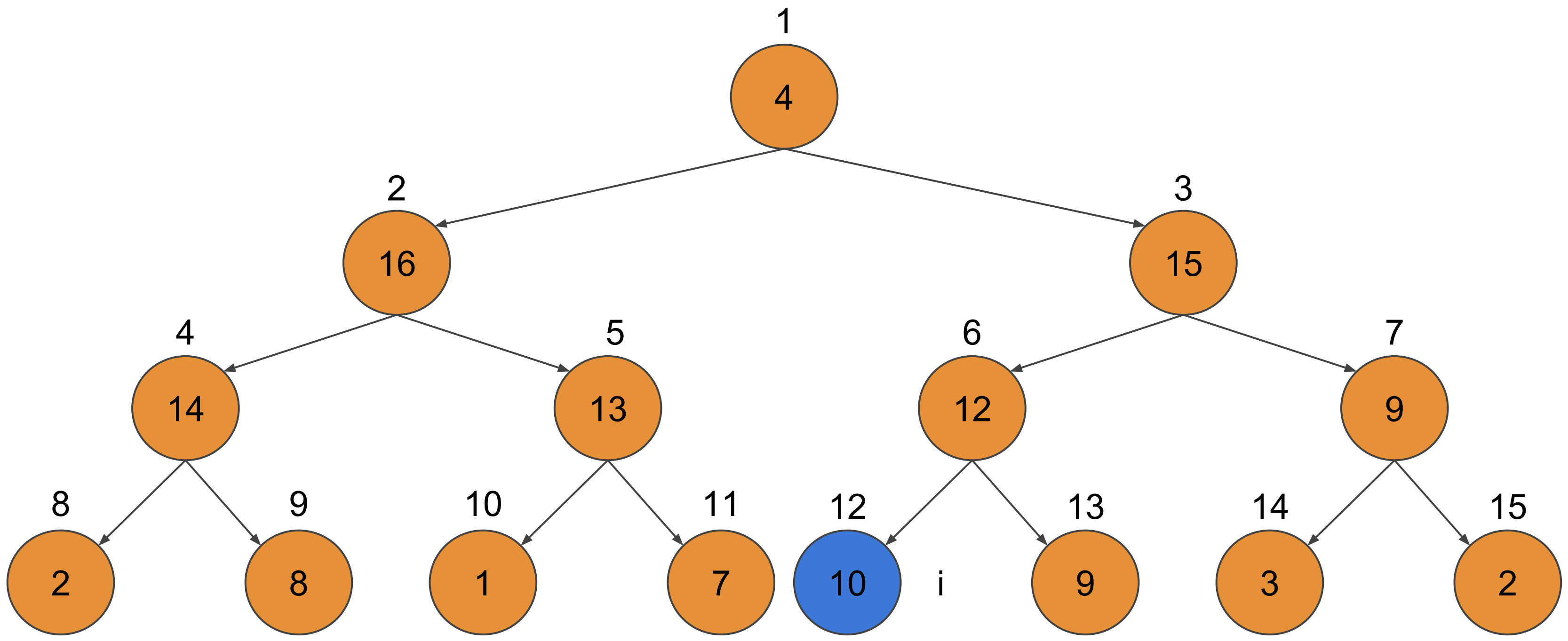
Simulation of Build Max Heap Continued



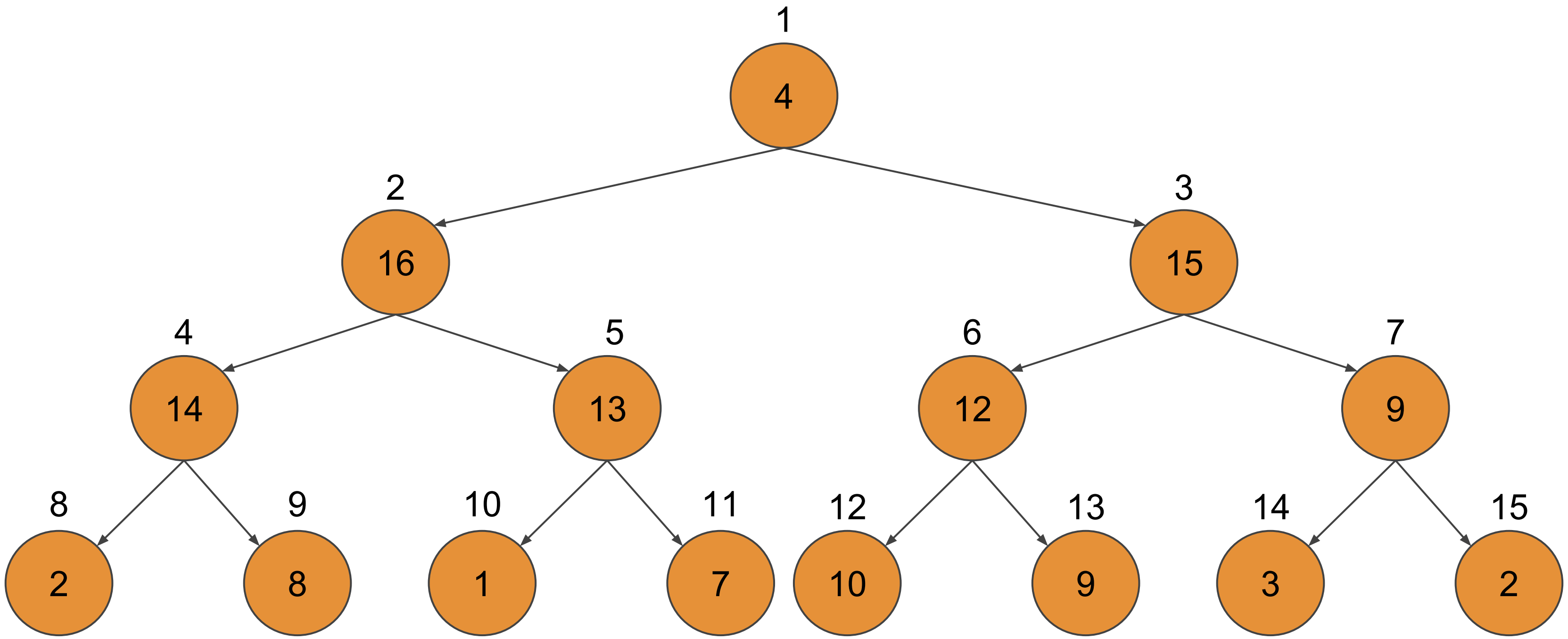
Simulation of Build Max Heap Continued



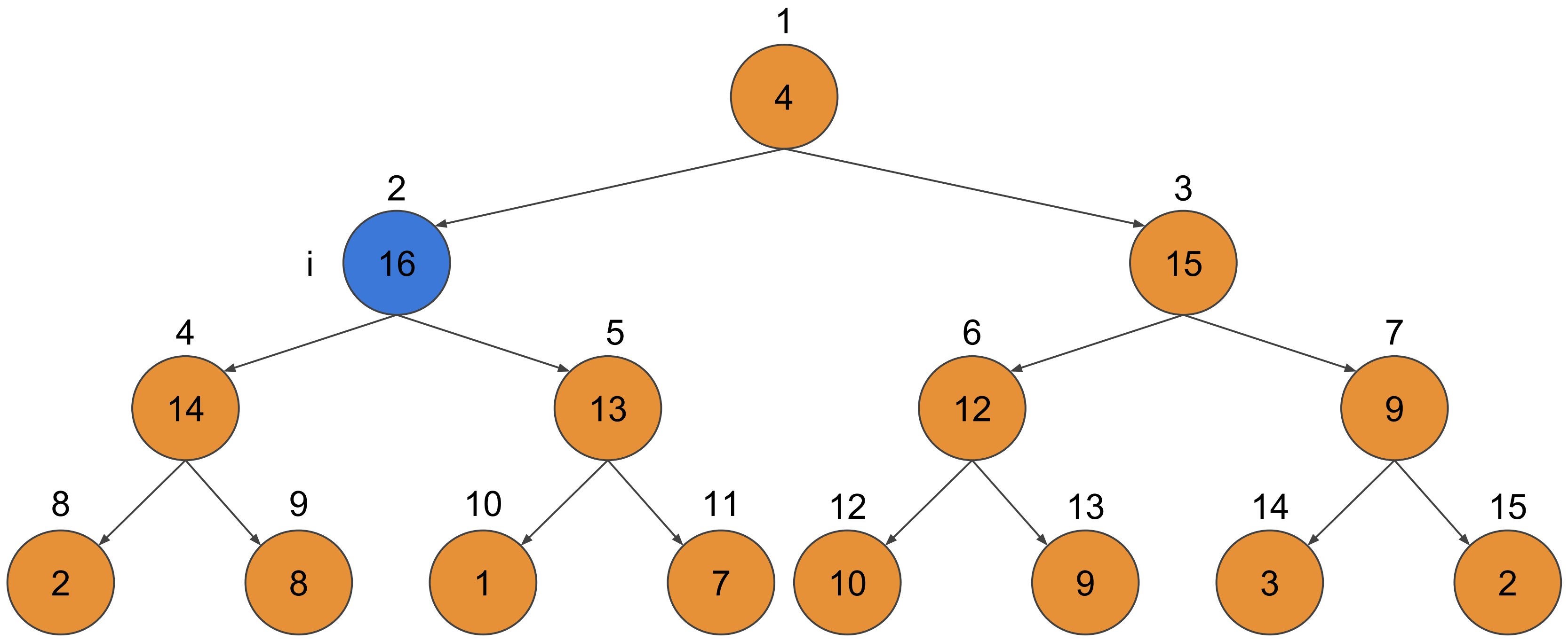
Simulation of Build Max Heap Continued



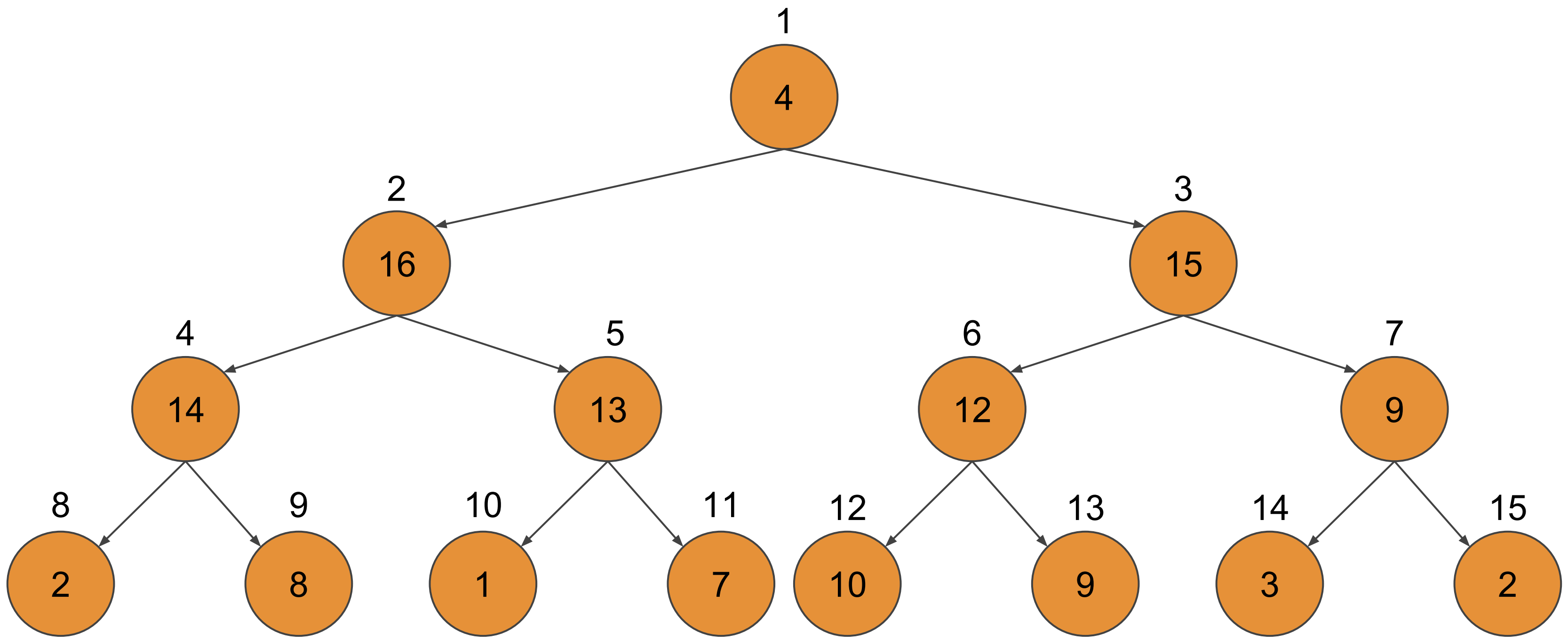
Simulation of Build Max Heap Continued



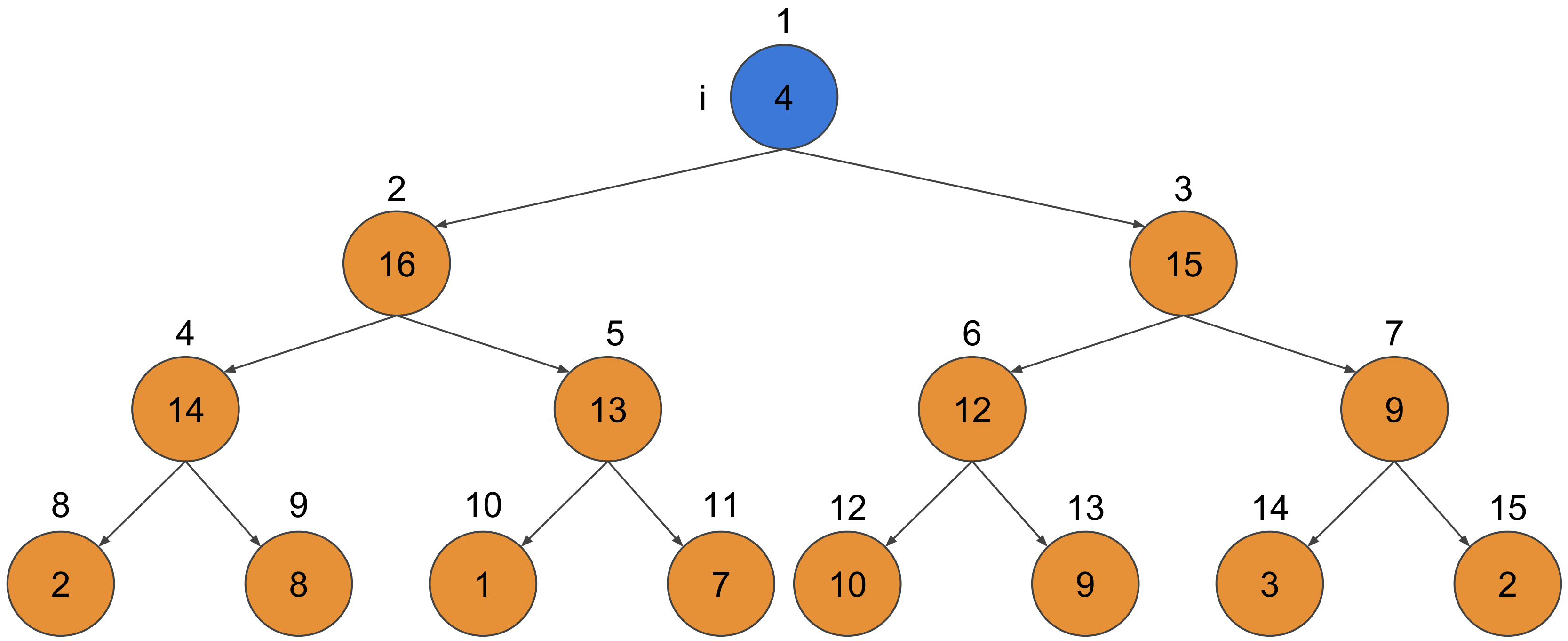
Simulation of Build Max Heap Continued



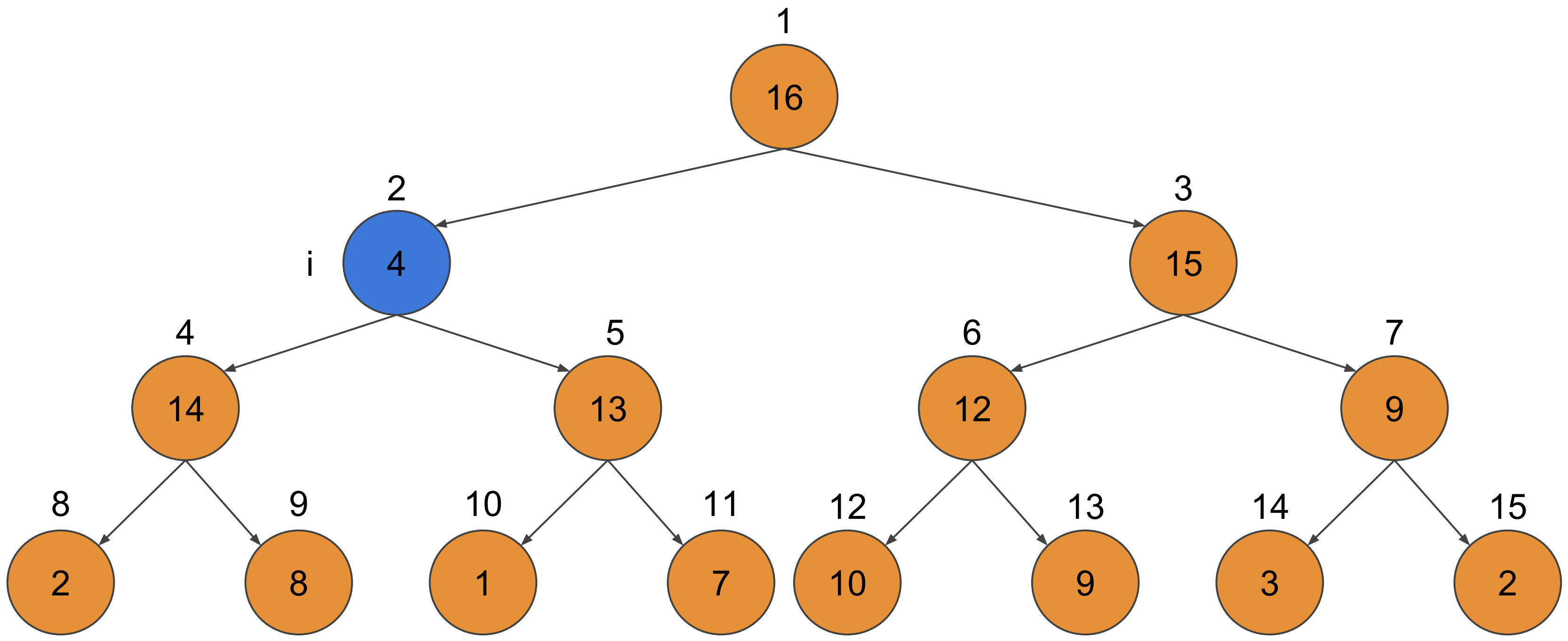
Simulation of Build Max Heap Continued



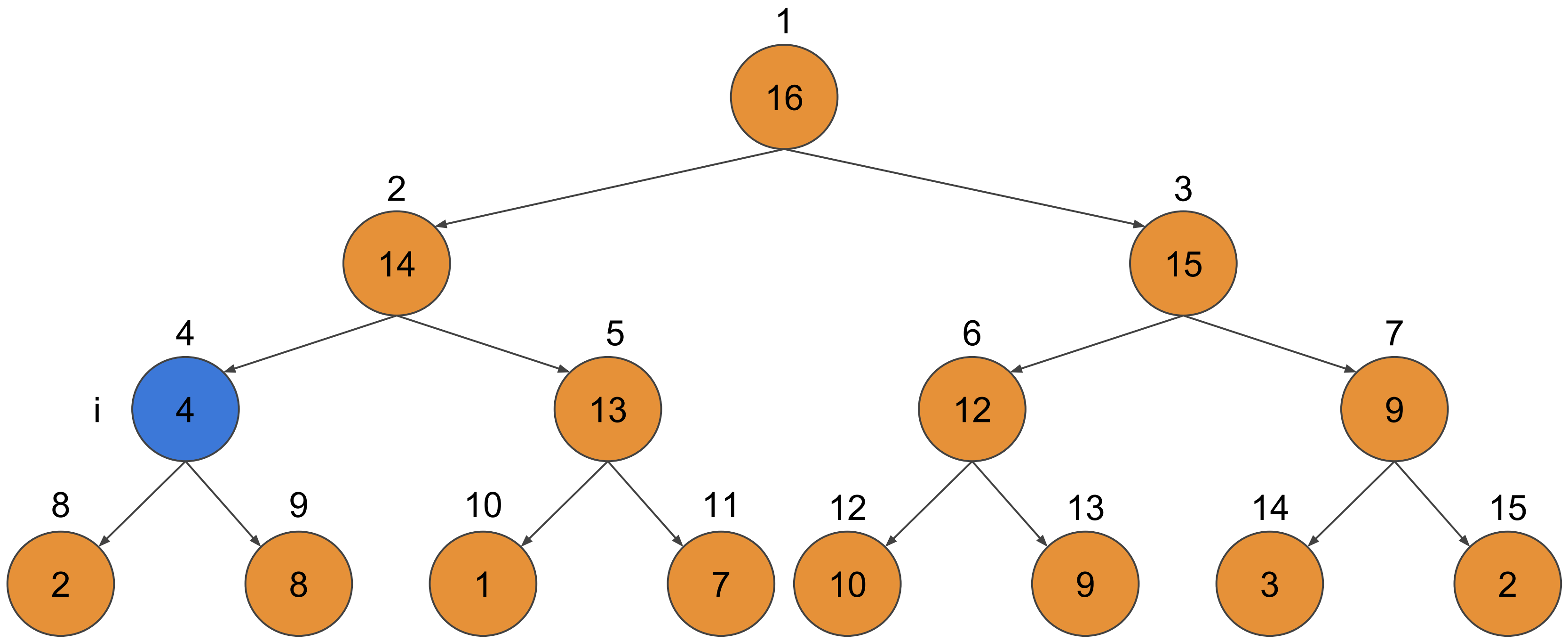
Simulation of Build Max Heap Continued



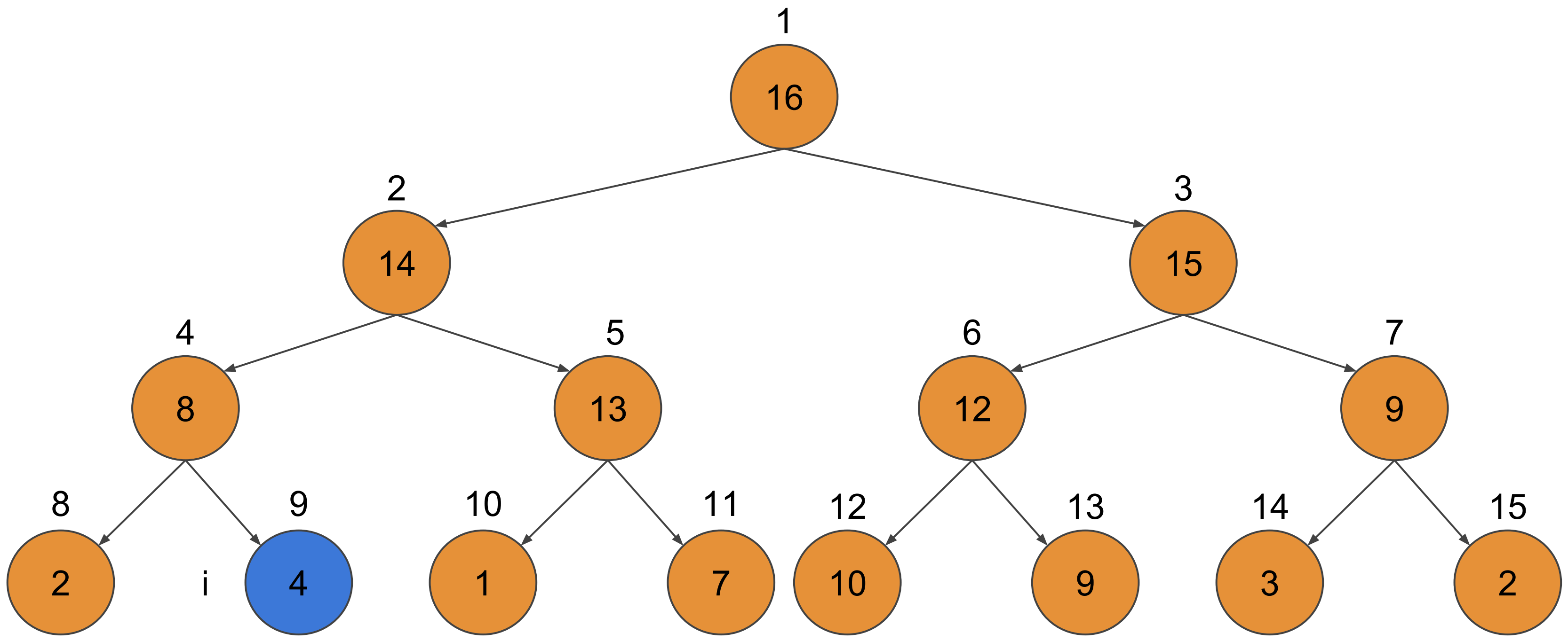
Simulation of Build Max Heap Continued



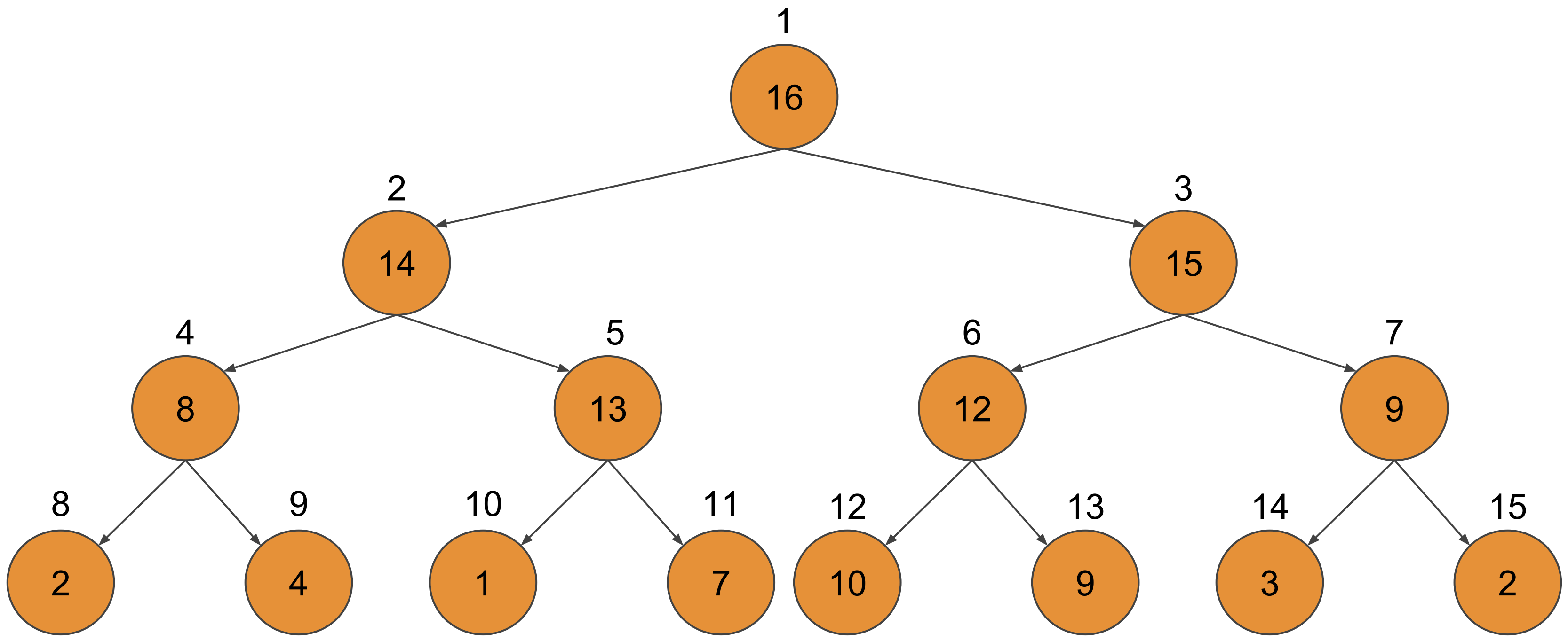
Simulation of Build Max Heap Continued



Simulation of Build Max Heap Continued



Simulation of Build Max Heap Continued



Complexity Analysis of Build Max Heap

We can compute a simple upper bound on the running time of Build_Max_Heap. As each call to Max_Heapify costs O(lgn) time, and Build_Max_Heap makes O(n) such calls. Thus, the running time is O(nlgn). This upper bound, though correct, is not asymptotically tight. We can derive a tighter bound by observing that the time for Max_Heapify to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

Complexity Analysis of Build Max Heap Continued

Our tighter analysis relies on the properties that an n-element heap has height ⌊lgn⌋ and at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

The time required by Max_Heapify when called on a node of height h is O(h) and so we can express the total cost of Build_Max_Heap as being bounded from above by

$$\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h})$$

Complexity Analysis of Build Max Heap Continued

According to the geometric series for real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + x^3 + ... + x^n$$

is a geometric or exponential series and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

When the summation is infinite and $|x| < 1$, we have the infinite decreasing geometric series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x}$$

By differentiating both sides of the infinite geometric series and multiplying by x, we get

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1 - x)^2} \text{.......(A.8)}$$

Complexity Analysis of Build Max Heap Continued

$$\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h})$$

We evalaute the last summation by substituting $x = 1/2$ in the formula (A.8), yielding

$$\sum_{k=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

Thus, we can bound the running time of Build_Max_Heap as

$$O(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n * 2) = O(n)$$

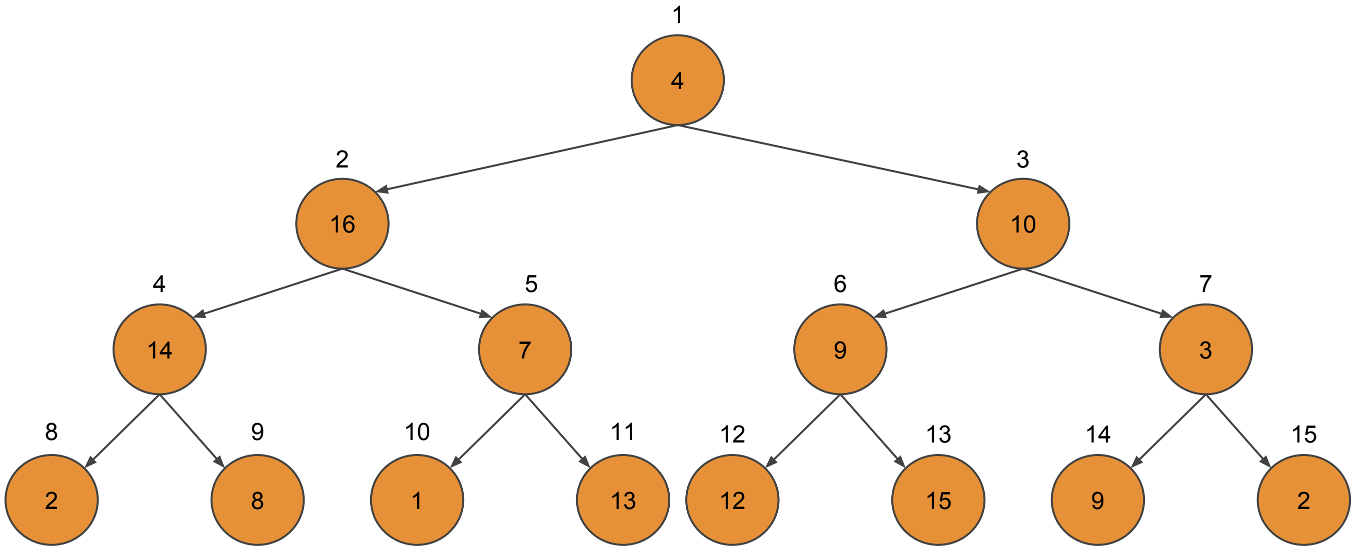
Heap Sort

The heapsort algorithm starts by using Build_Max_Heap to build a max-heap on the input array A[1...n], where n = A.length. Since the maximum element of the array is stored at the root A[1], we can put it into its correct final position by exchanging it with A[n]. If we now discard node n from the heap—and we can do so by simply decrementing A.heap_size we observe that the children of the root remain max-heaps, but the new root element might violate the max-heap property. All we need to do to restore the max-heap property, however, is call Max_Heapify(A, 1), which leaves a max-heap in A[1...n-1]. The heapsort algorithm then repeats this process for the max-heap of size n-1 down to a heap of size 2.

Heap Sort Pseudocode

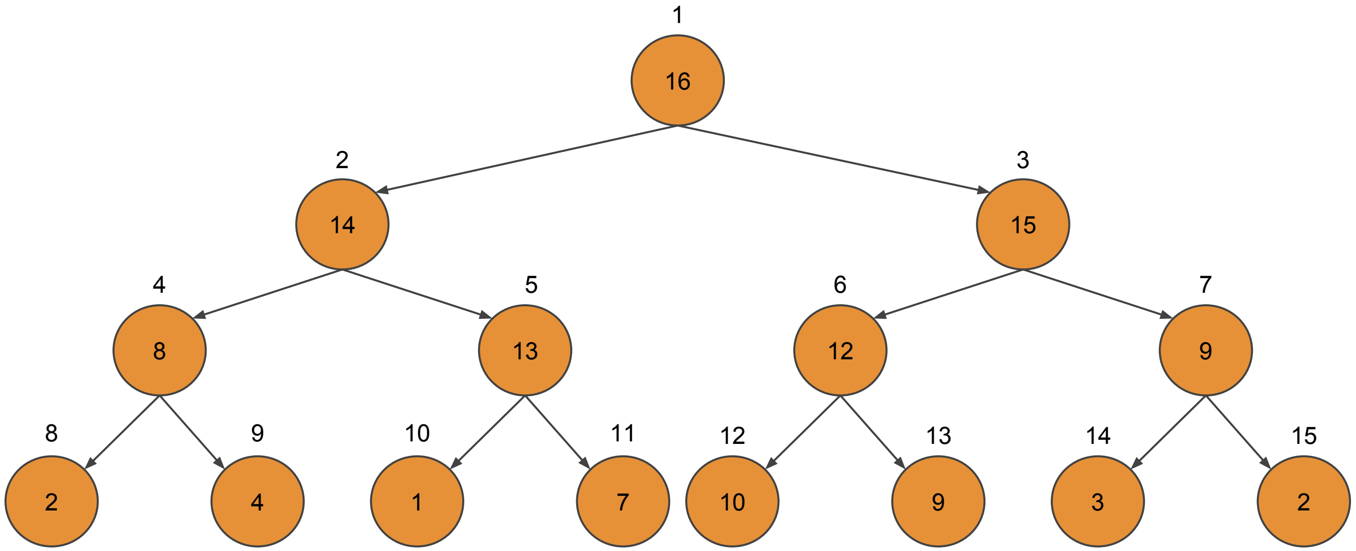
```
Heapsort(A):
    Build_Max_Heap(A)
    for i = A.length downto 2:
        exchange A[1] with A[i]
        A.heap_size = A.heap_size - 1
        Max_heapify(A, 1)
```

Simulation of Heap Sort



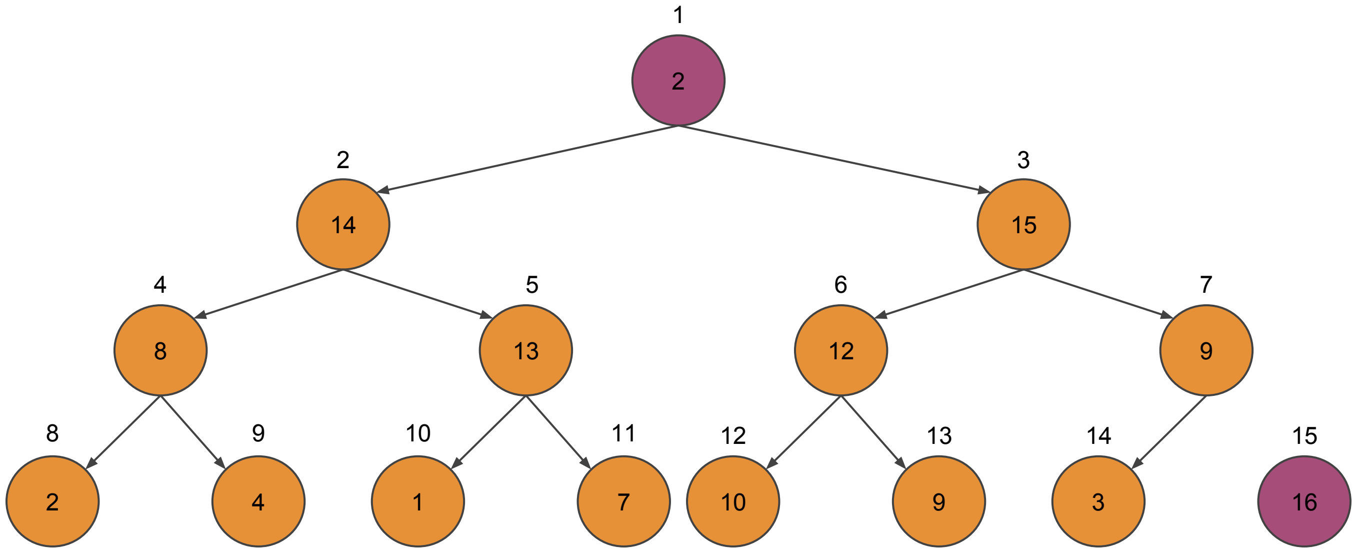
4	16	10	14	7	9	3	2	8	1	13	12	15	9	2

Simulation of Heap Sort



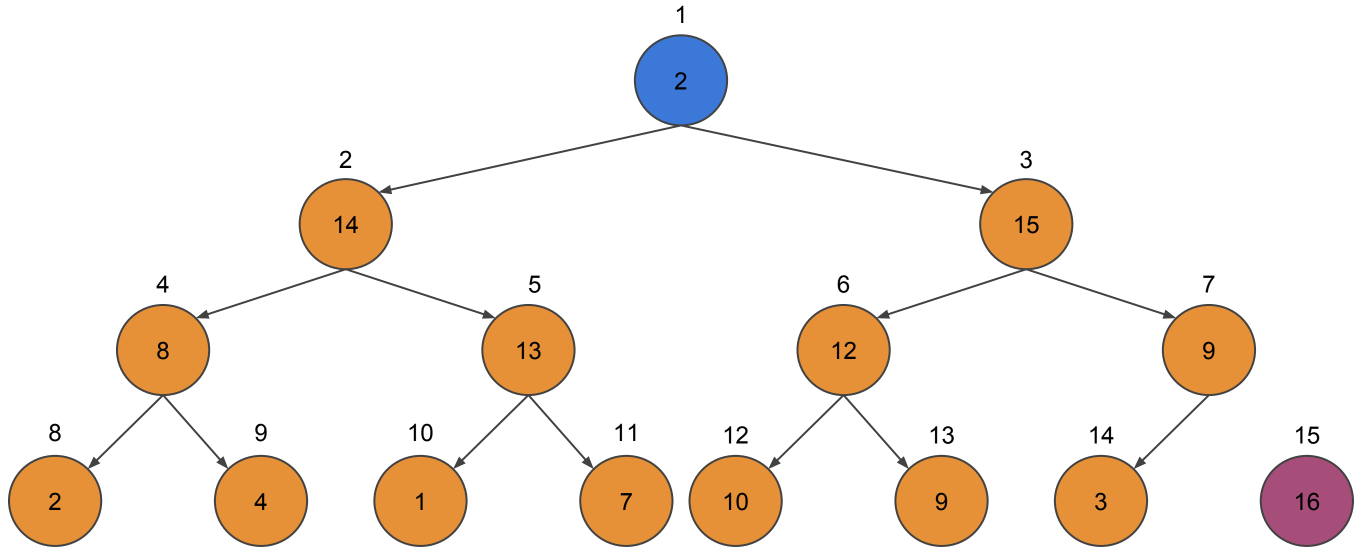
16	14	15	8	13	12	9	2	4	1	7	10	9	3	2

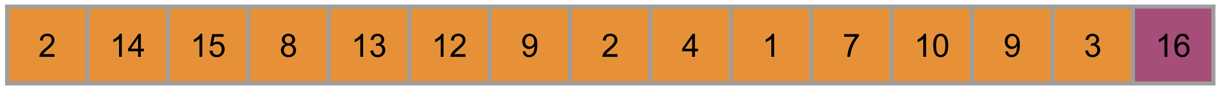
Simulation of Heap Sort



2	14	15	8	13	12	9	2	4	1	7	10	9	3	16

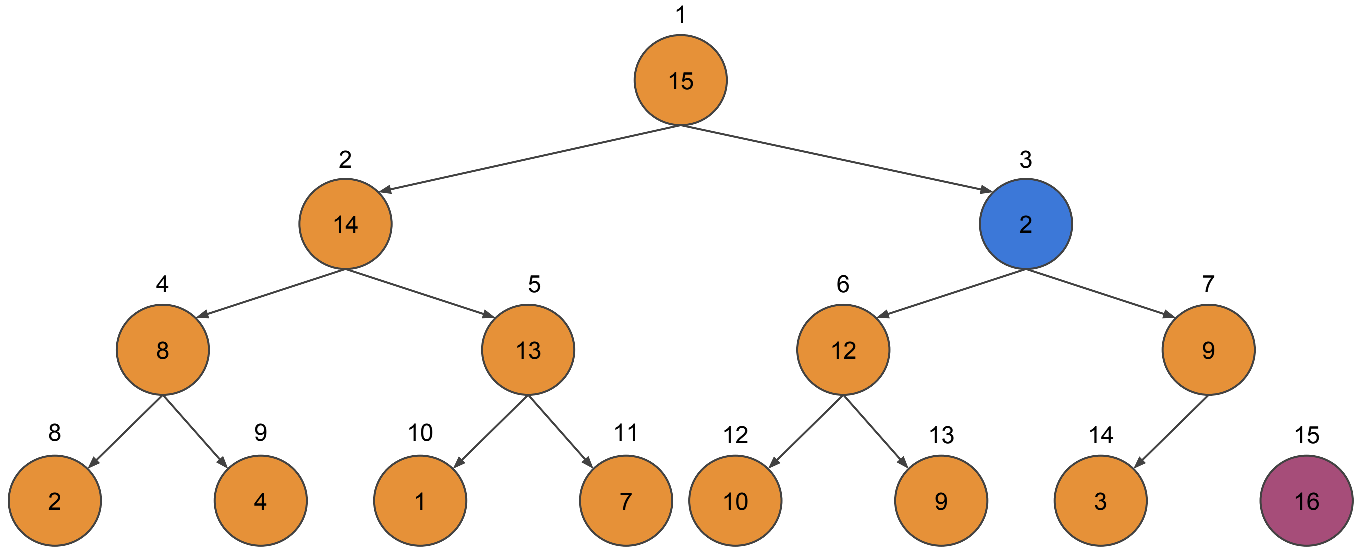
Simulation of Heap Sort





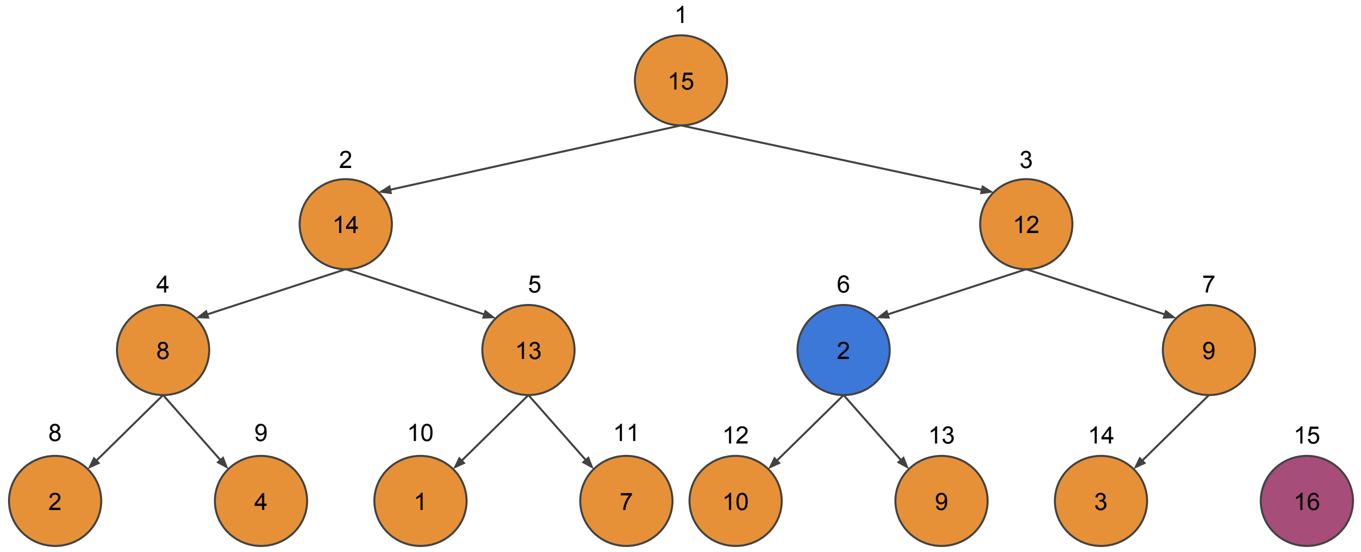
2	14	15	8	13	12	9	2	4	1	7	10	9	3	16

Simulation of Heap Sort



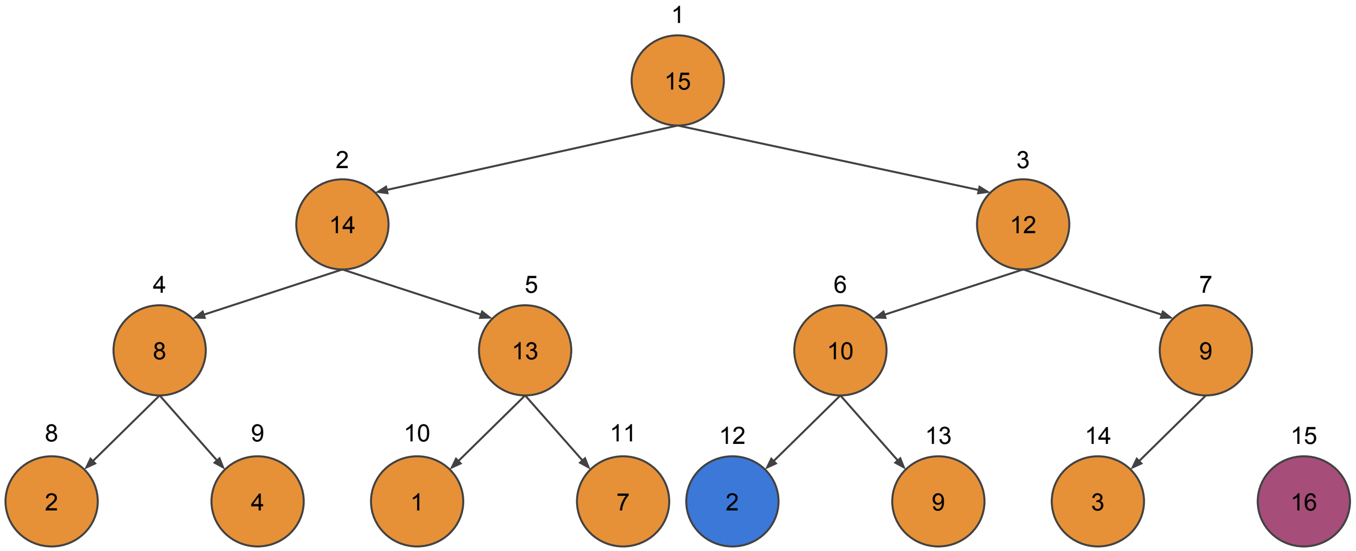
15	14	2	8	13	12	9	2	4	1	7	10	9	3	16

Simulation of Heap Sort



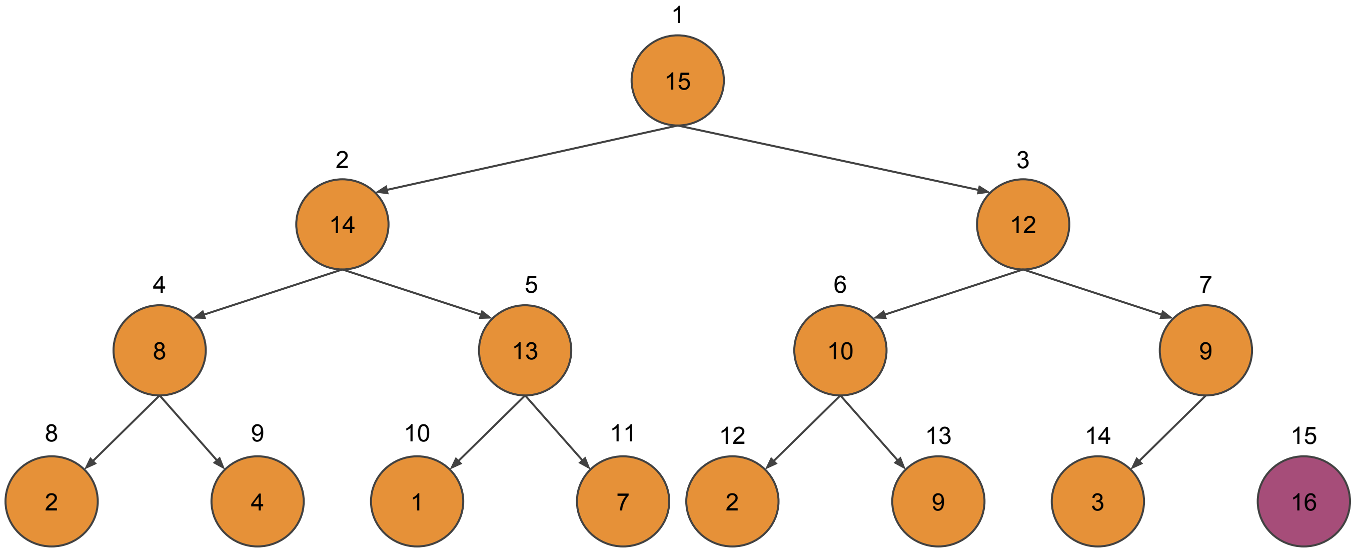
15	14	12	8	13	2	9	2	4	1	7	10	9	3	16

Simulation of Heap Sort



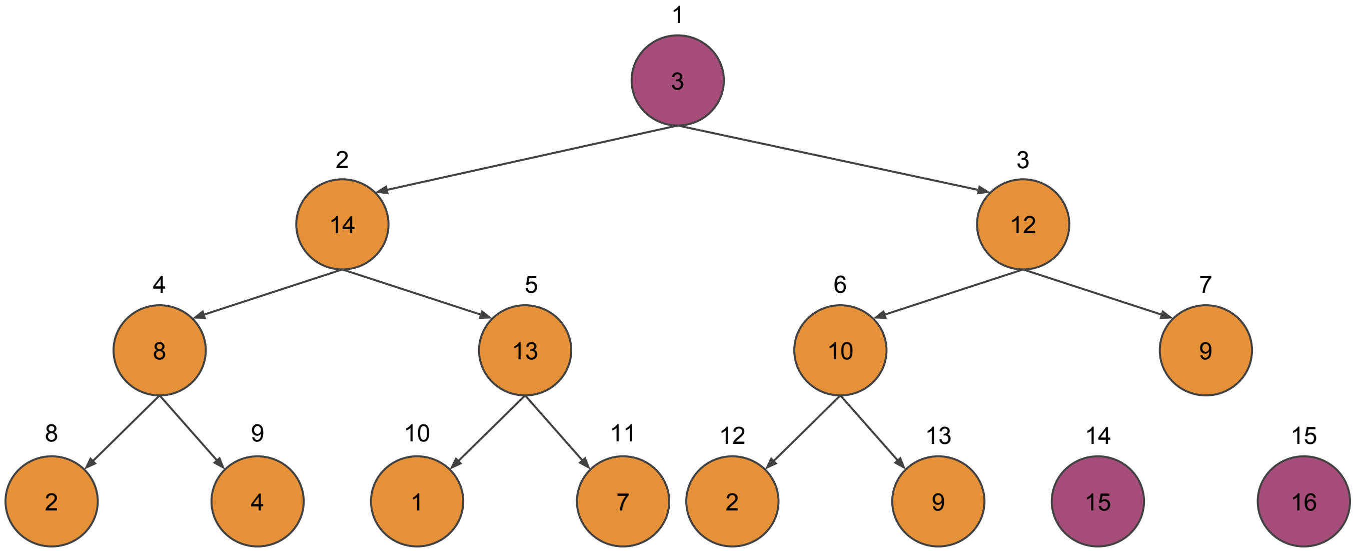
15	14	12	8	13	10	9	2	4	1	7	2	9	3	16

Simulation of Heap Sort



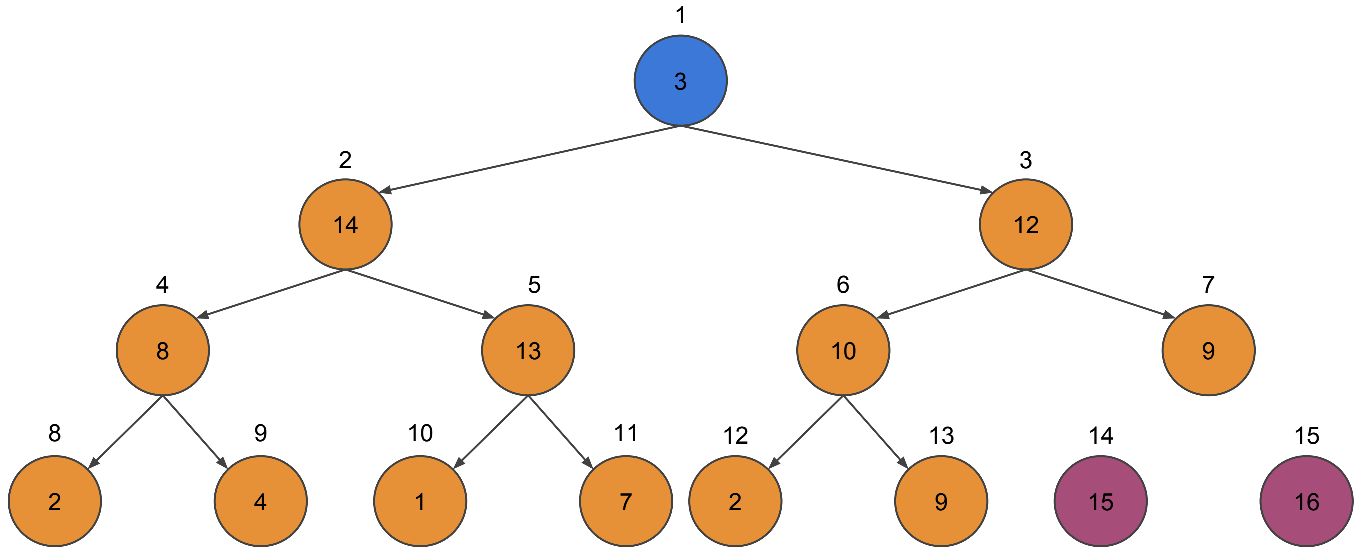
15	14	12	8	13	10	9	2	4	1	7	2	9	3	16

Simulation of Heap Sort



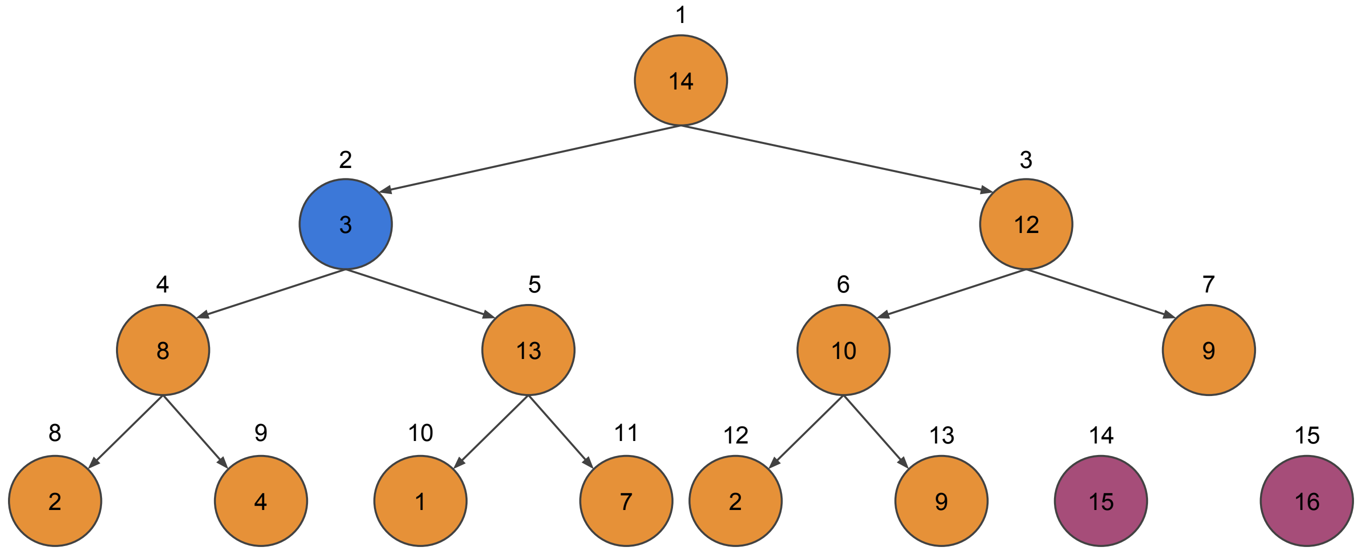
3	14	12	8	13	10	9	2	4	1	7	2	9	15	16

Simulation of Heap Sort



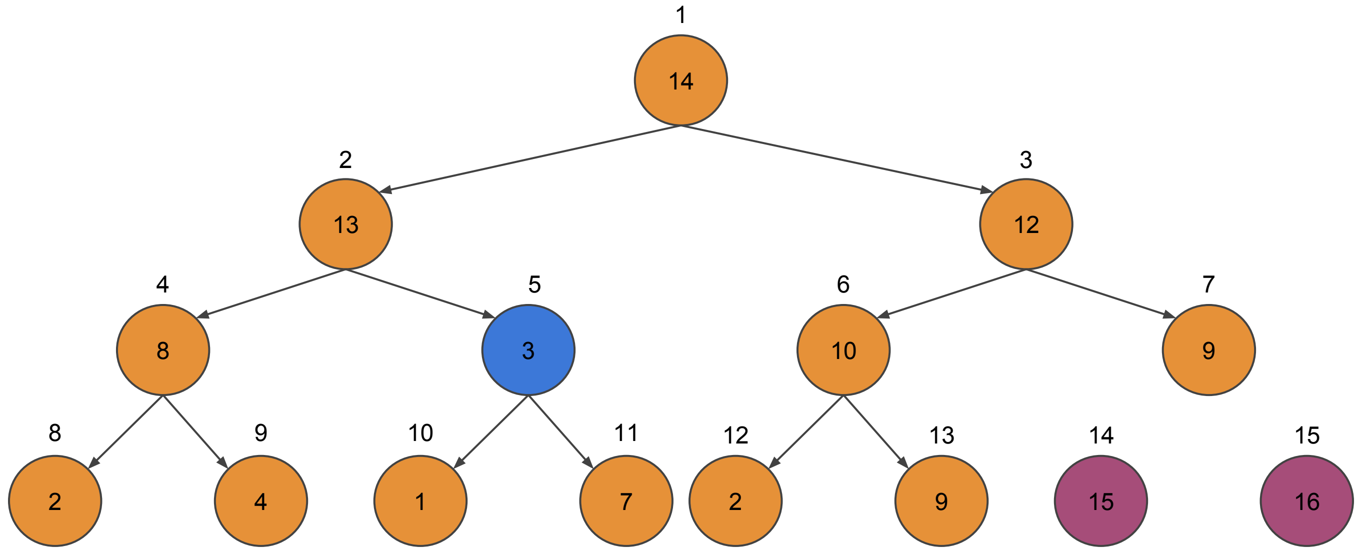
3	14	12	8	13	10	9	2	4	1	7	2	9	15	16

Simulation of Heap Sort



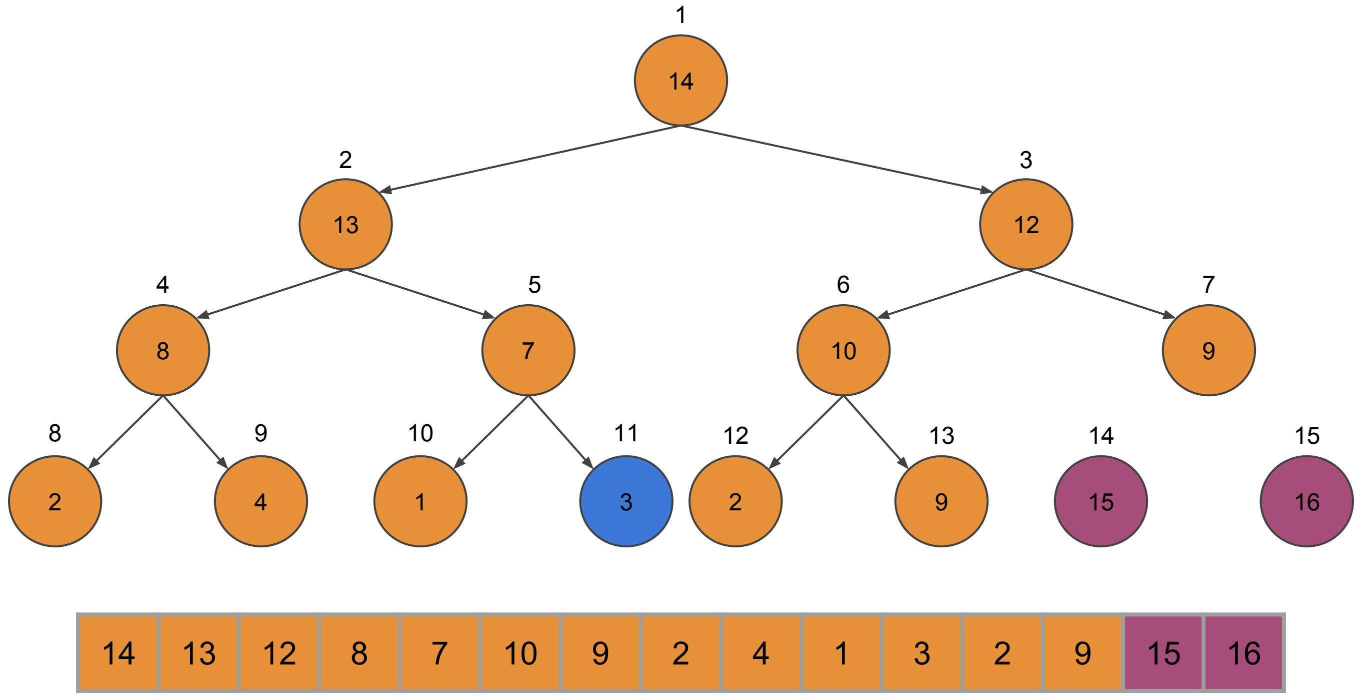
14	3	12	8	13	10	9	2	4	1	7	2	9	15	16

Simulation of Heap Sort

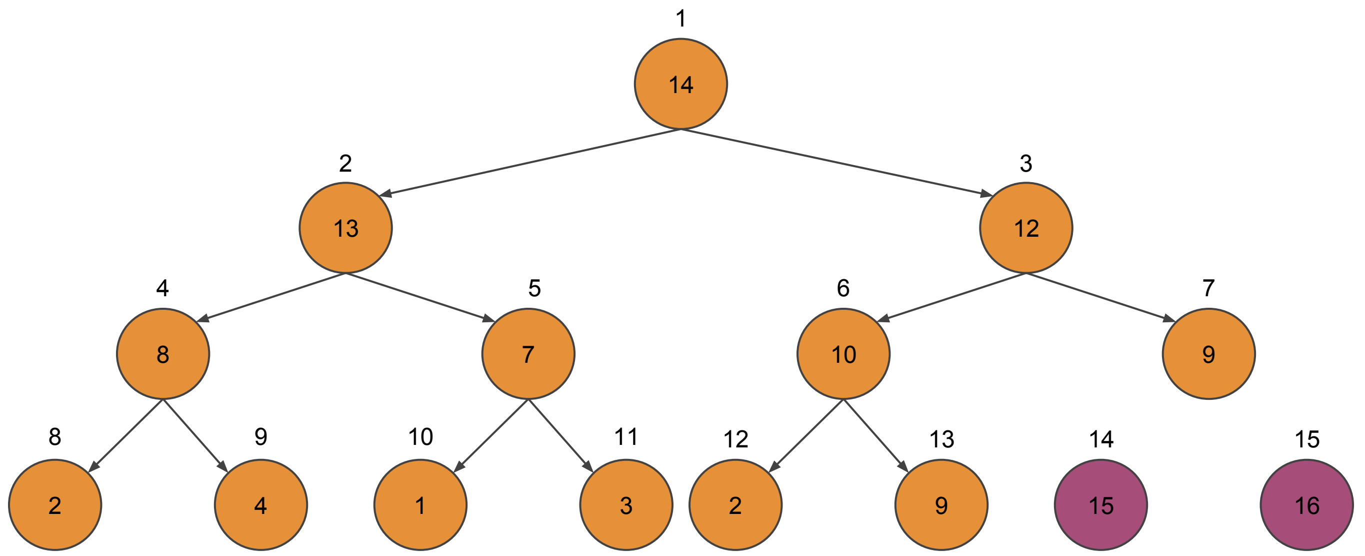


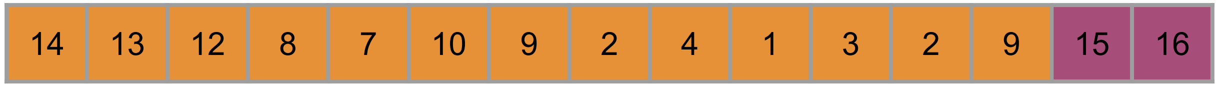
14	13	12	8	3	10	9	2	4	1	7	2	9	15	16

Simulation of Heap Sort



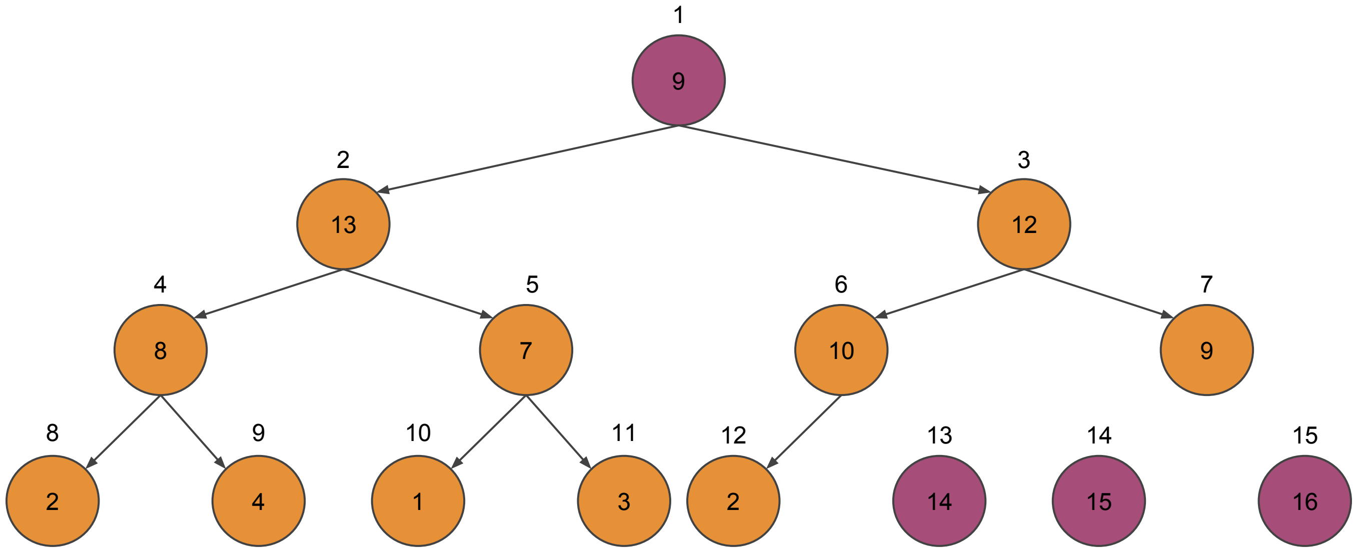
Simulation of Heap Sort





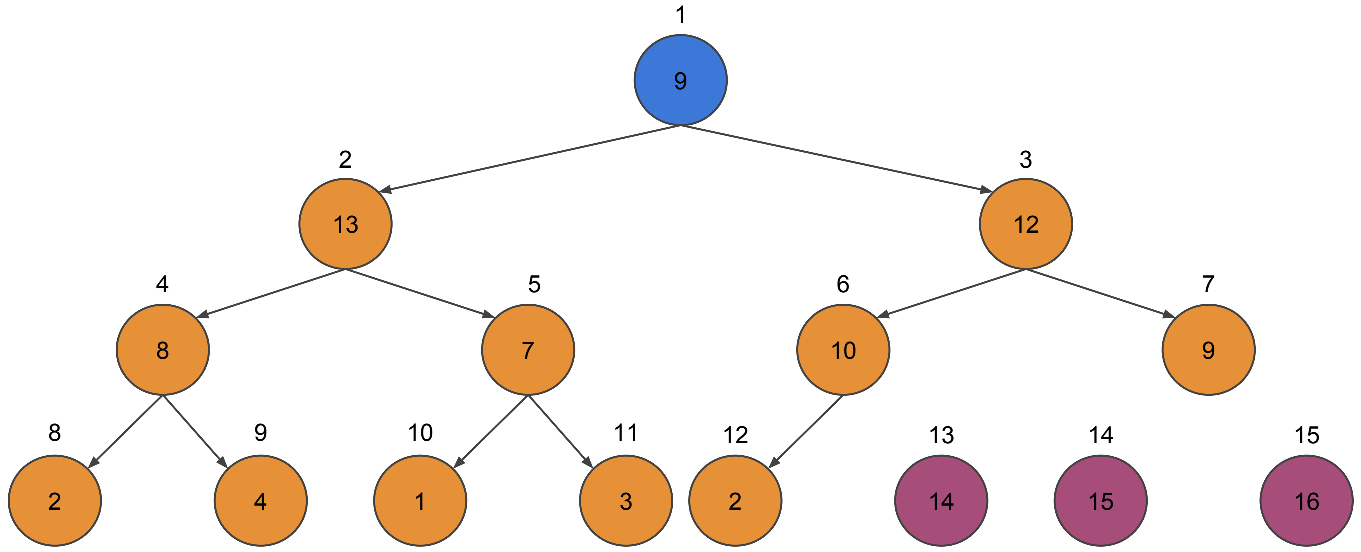
14	13	12	8	7	10	9	2	4	1	3	2	9	15	16

Simulation of Heap Sort



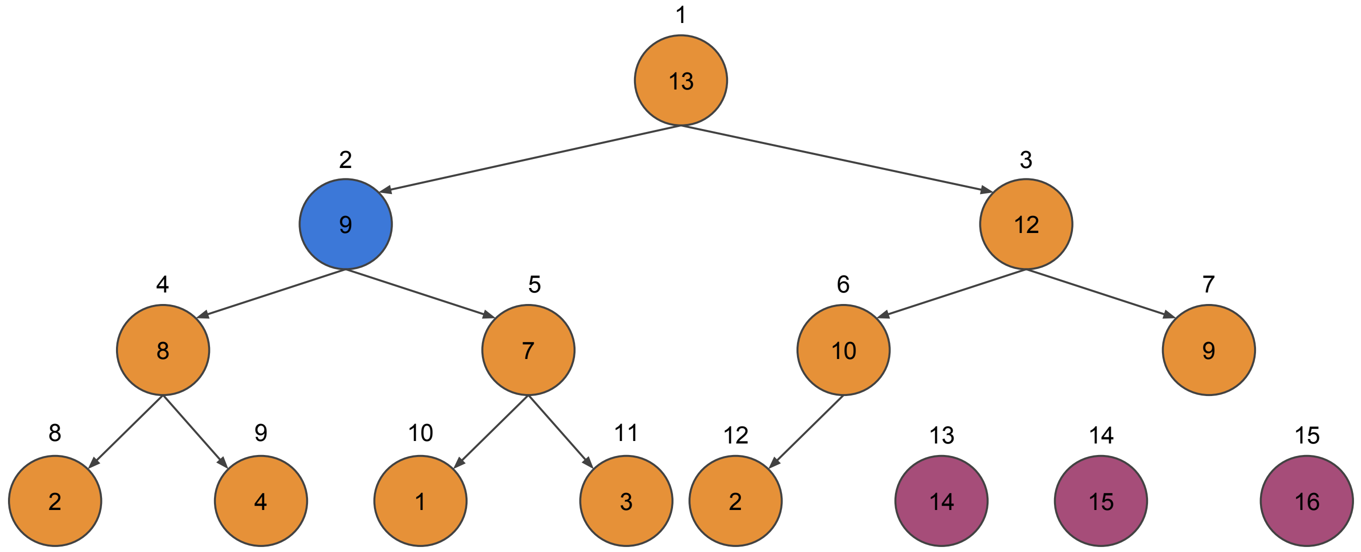
9	13	12	8	7	10	9	2	4	1	3	2	14	15	16

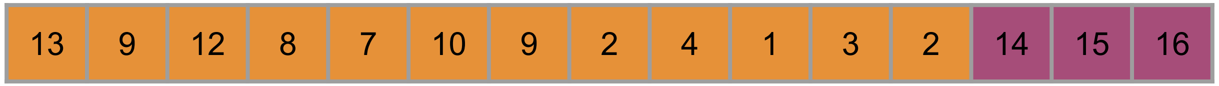
Simulation of Heap Sort



9	13	12	8	7	10	9	2	4	1	3	2	14	15	16

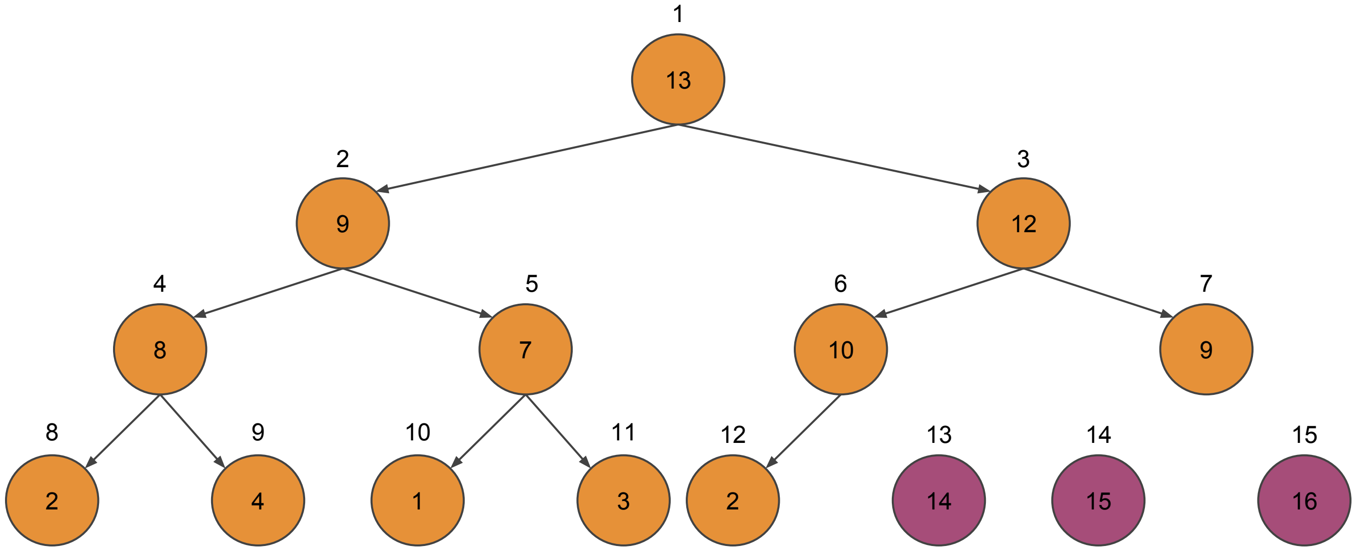
Simulation of Heap Sort

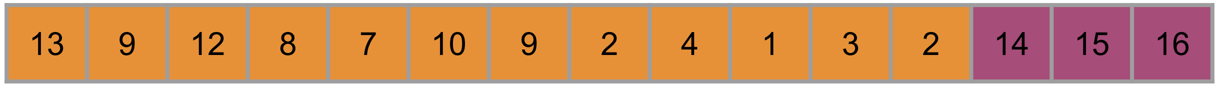




13	9	12	8	7	10	9	2	4	1	3	2	14	15	16

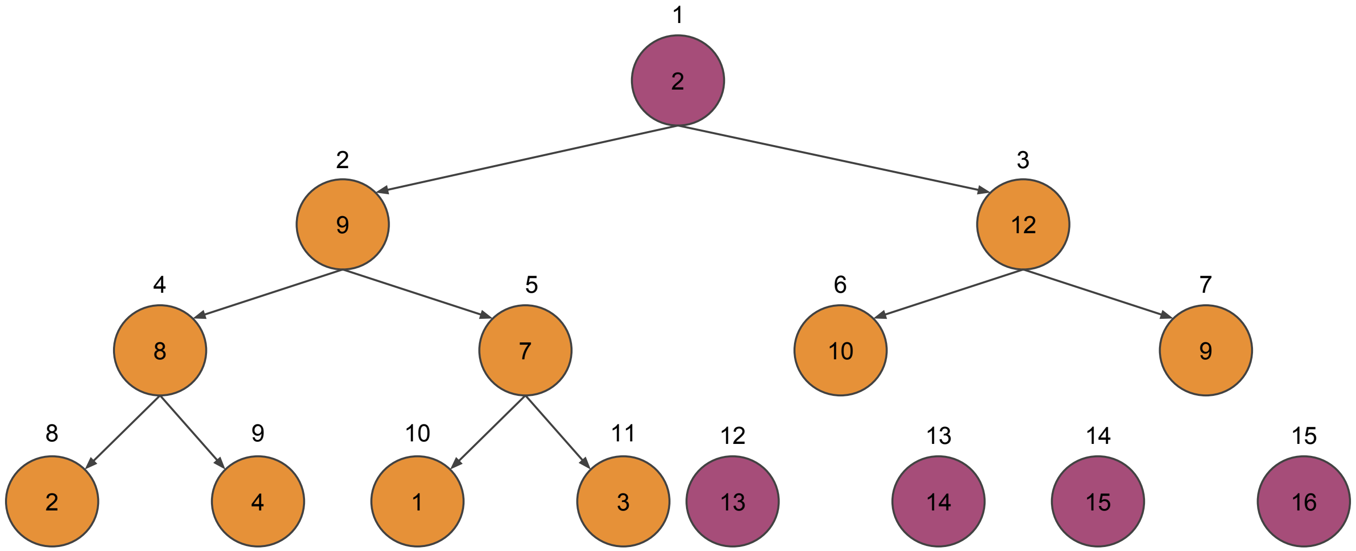
Simulation of Heap Sort





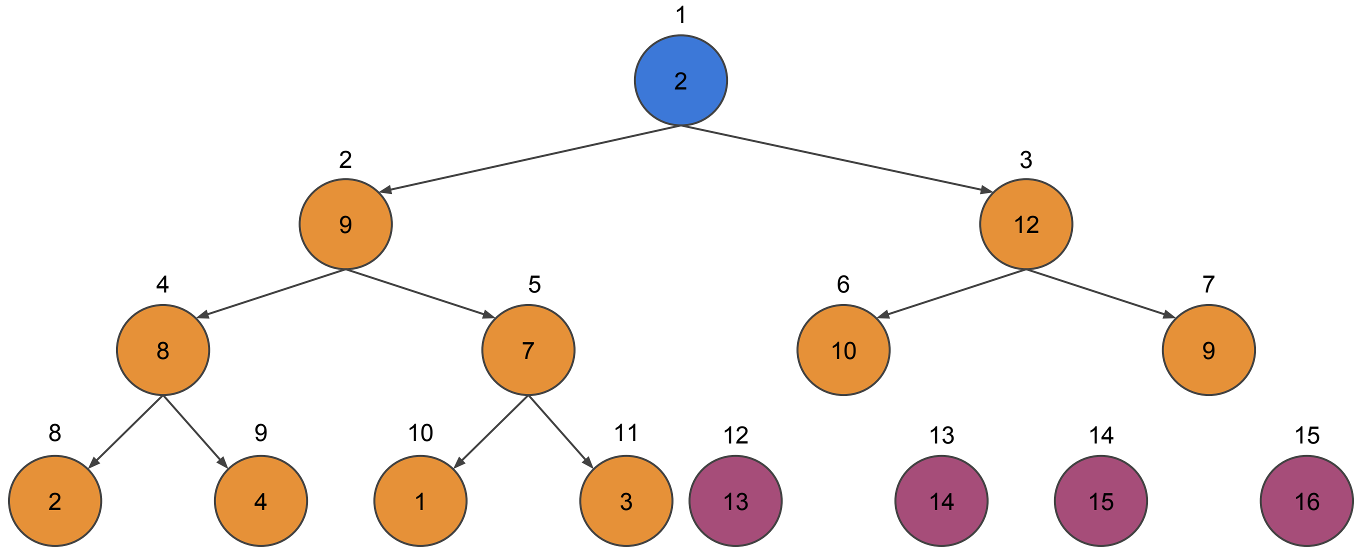
13	9	12	8	7	10	9	2	4	1	3	2	14	15	16

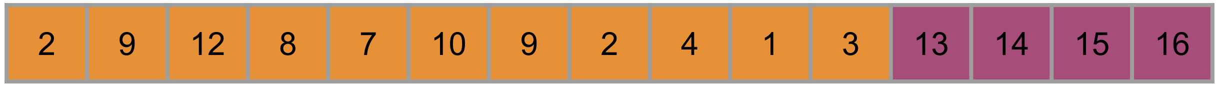
Simulation of Heap Sort



2	9	12	8	7	10	9	2	4	1	3	13	14	15	16

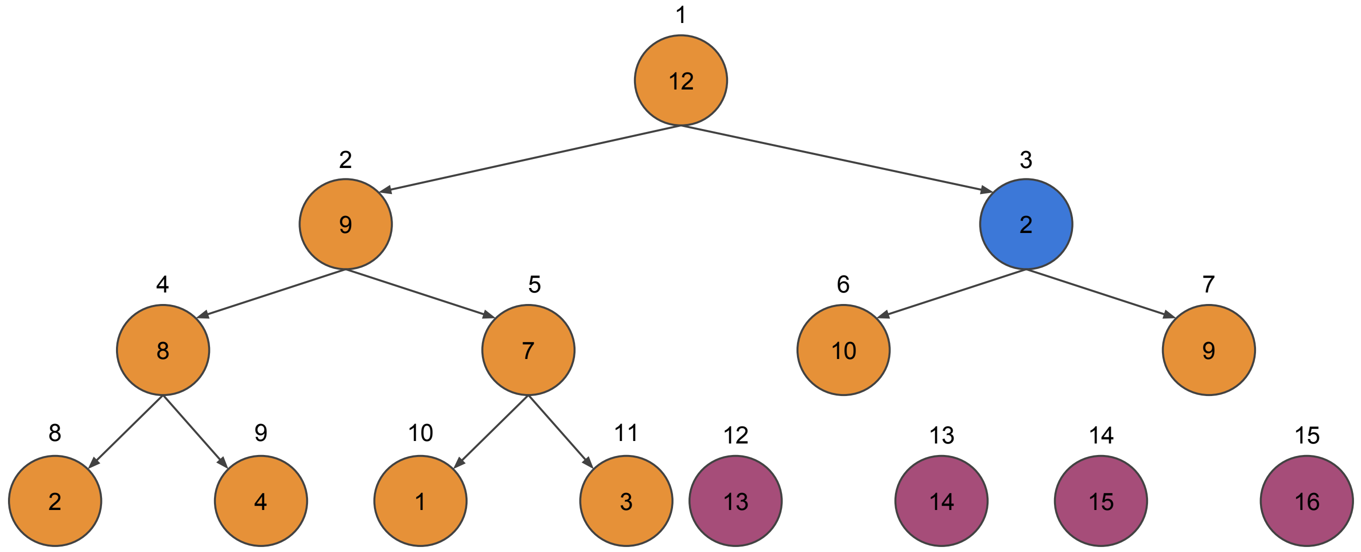
Simulation of Heap Sort





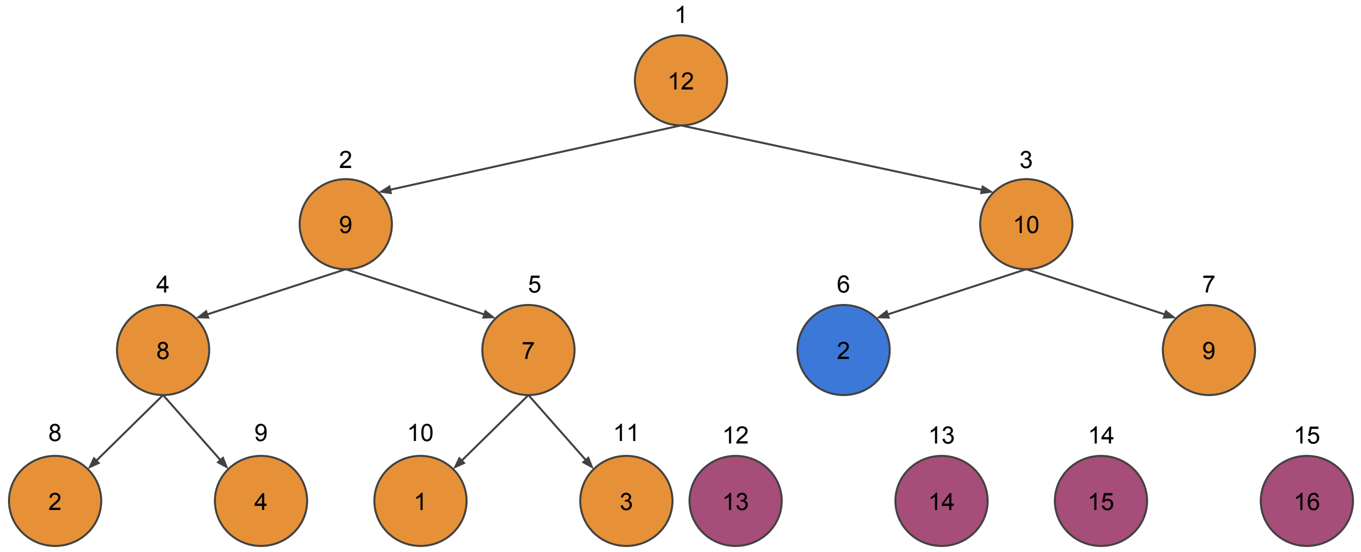
2	9	12	8	7	10	9	2	4	1	3	13	14	15	16

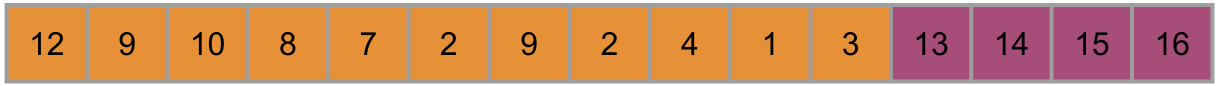
Simulation of Heap Sort



12	9	2	8	7	10	9	2	4	1	3	13	14	15	16

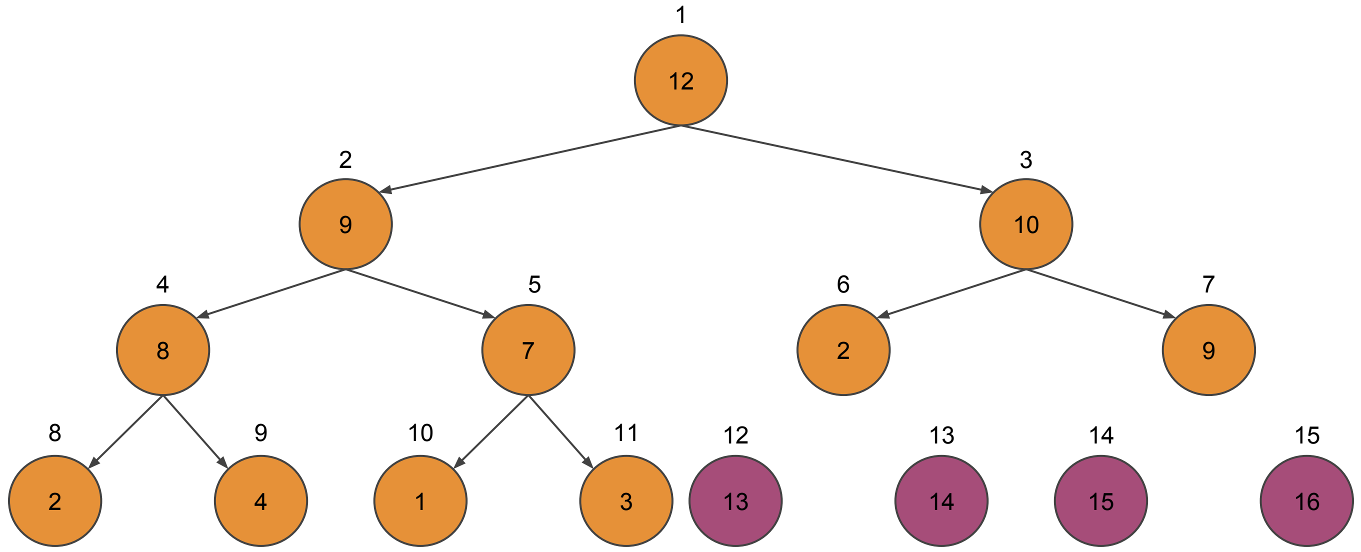
Simulation of Heap Sort





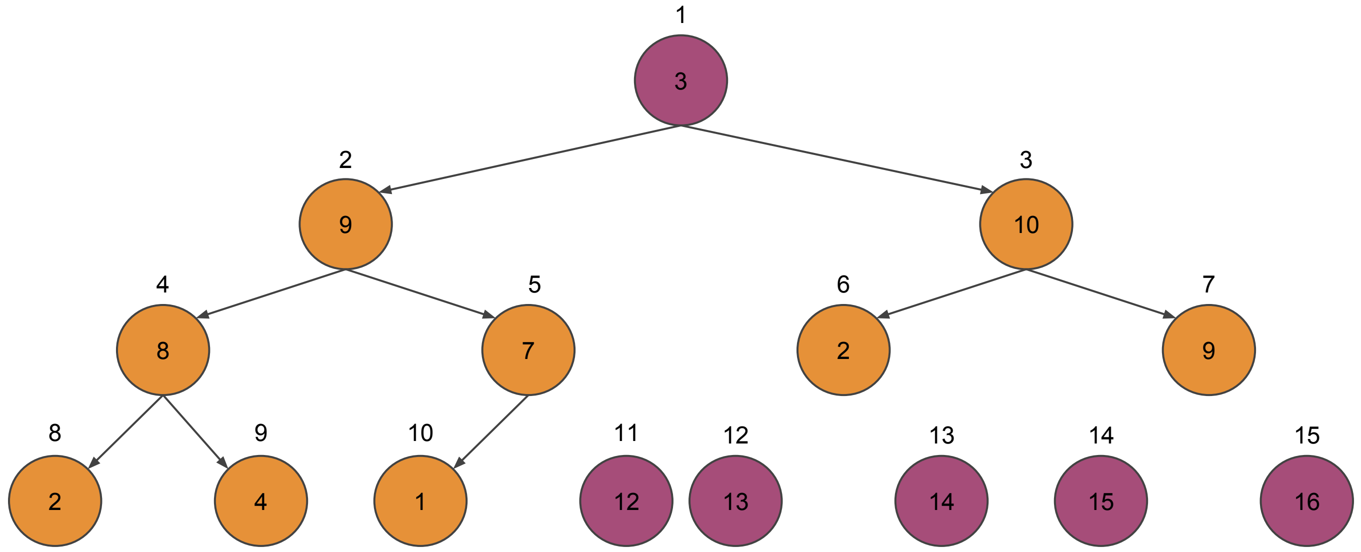
12	9	10	8	7	2	9	2	4	1	3	13	14	15	16

Simulation of Heap Sort



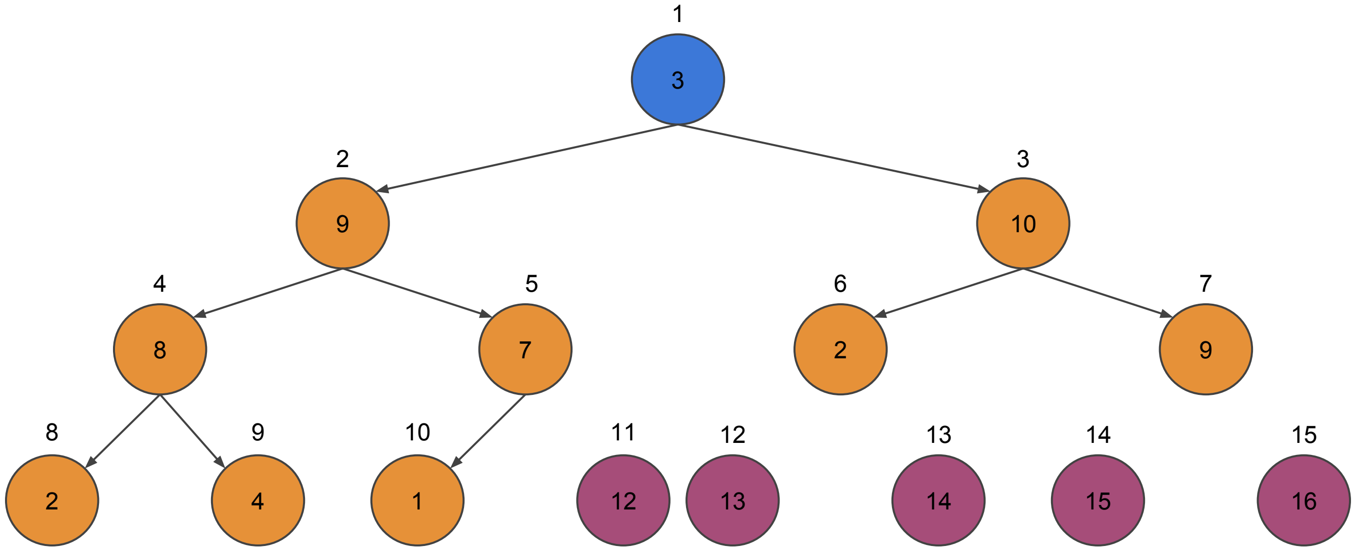
12	9	10	8	7	2	9	2	4	1	3	13	14	15	16

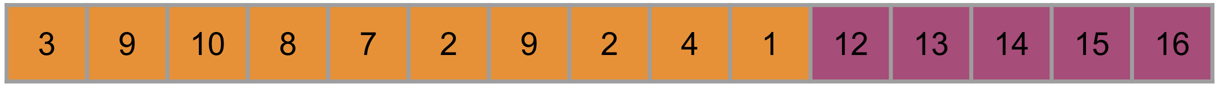
Simulation of Heap Sort



3	9	10	8	7	2	9	2	4	1	12	13	14	15	16

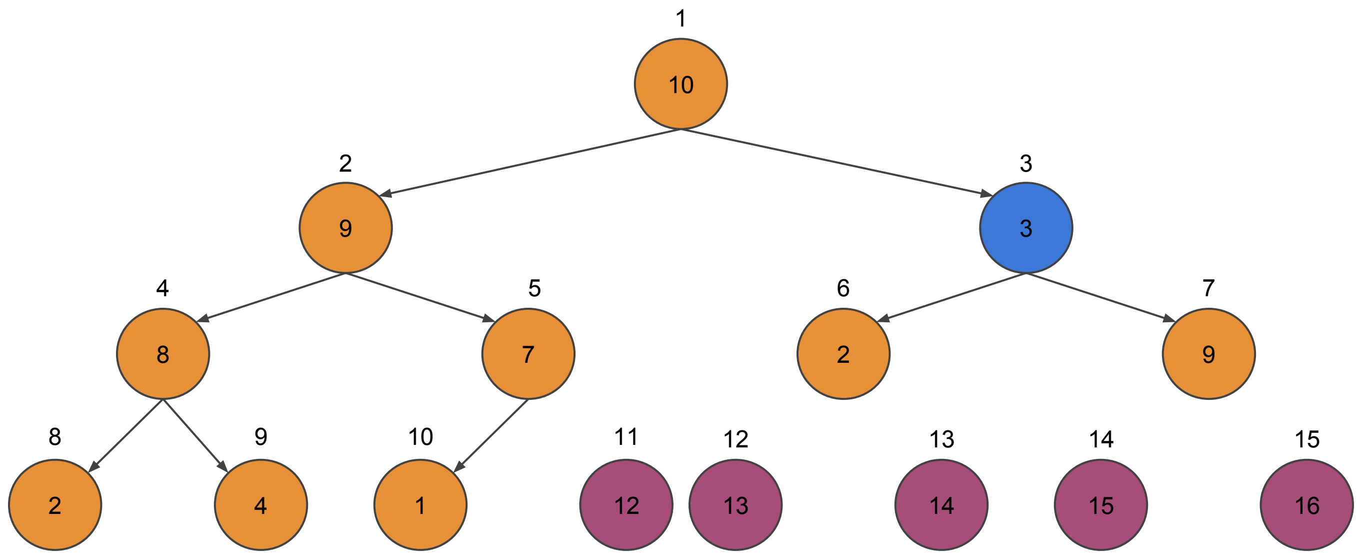
Simulation of Heap Sort





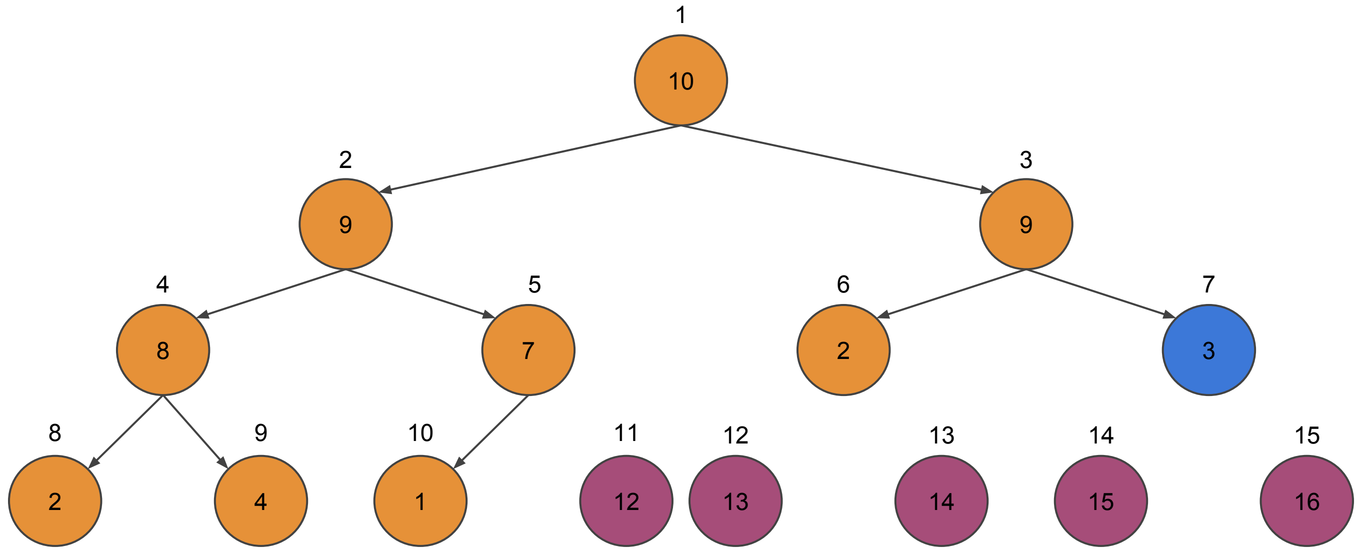
3	9	10	8	7	2	9	2	4	1	12	13	14	15	16

Simulation of Heap Sort



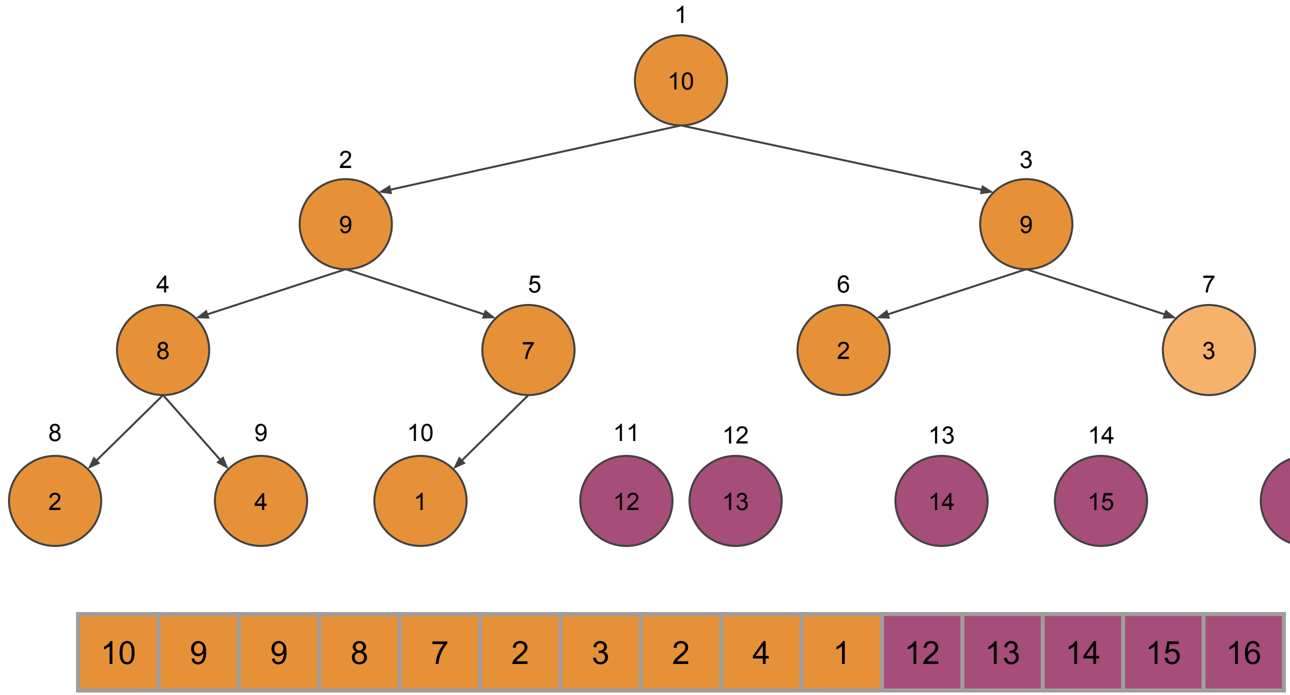
10	9	3	8	7	2	9	2	4	1	12	13	14	15	16

Simulation of Heap Sort

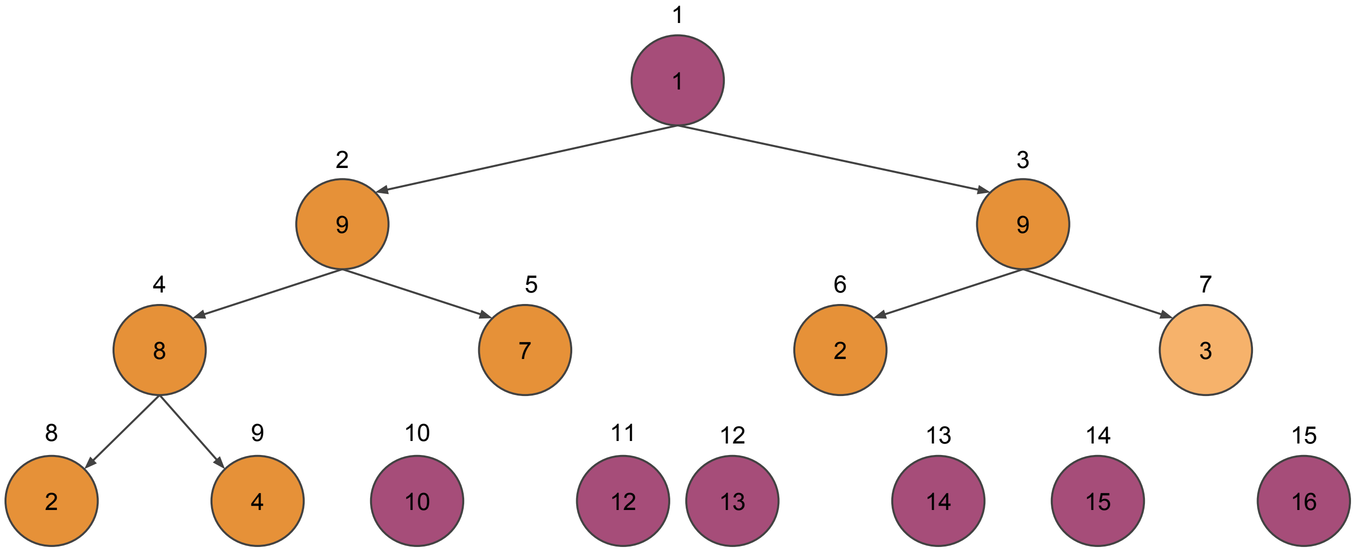


10	9	9	8	7	2	3	2	4	1	12	13	14	15	16

Simulation of Heap Sort

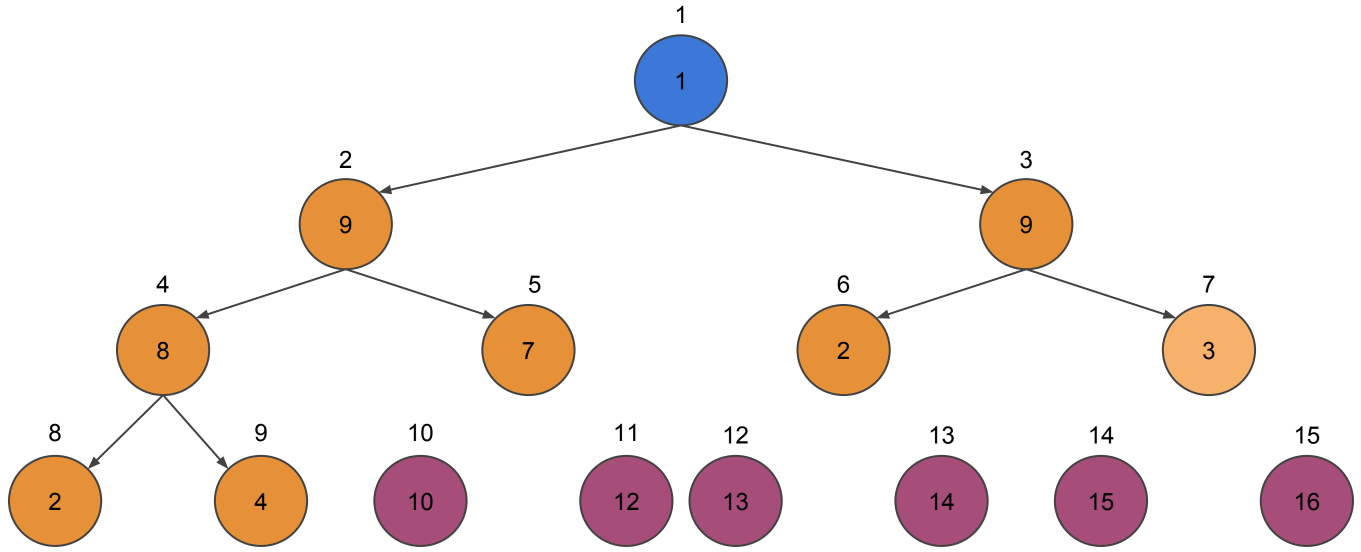


Simulation of Heap Sort



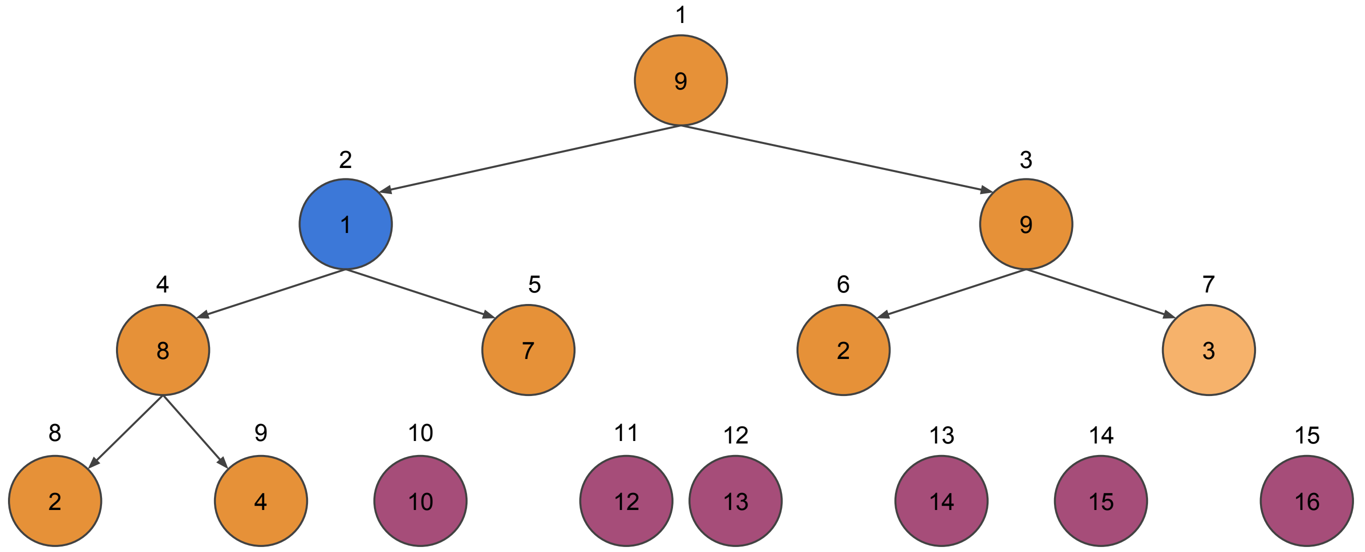
1	9	9	8	7	2	3	2	4	10	12	13	14	15	16

Simulation of Heap Sort



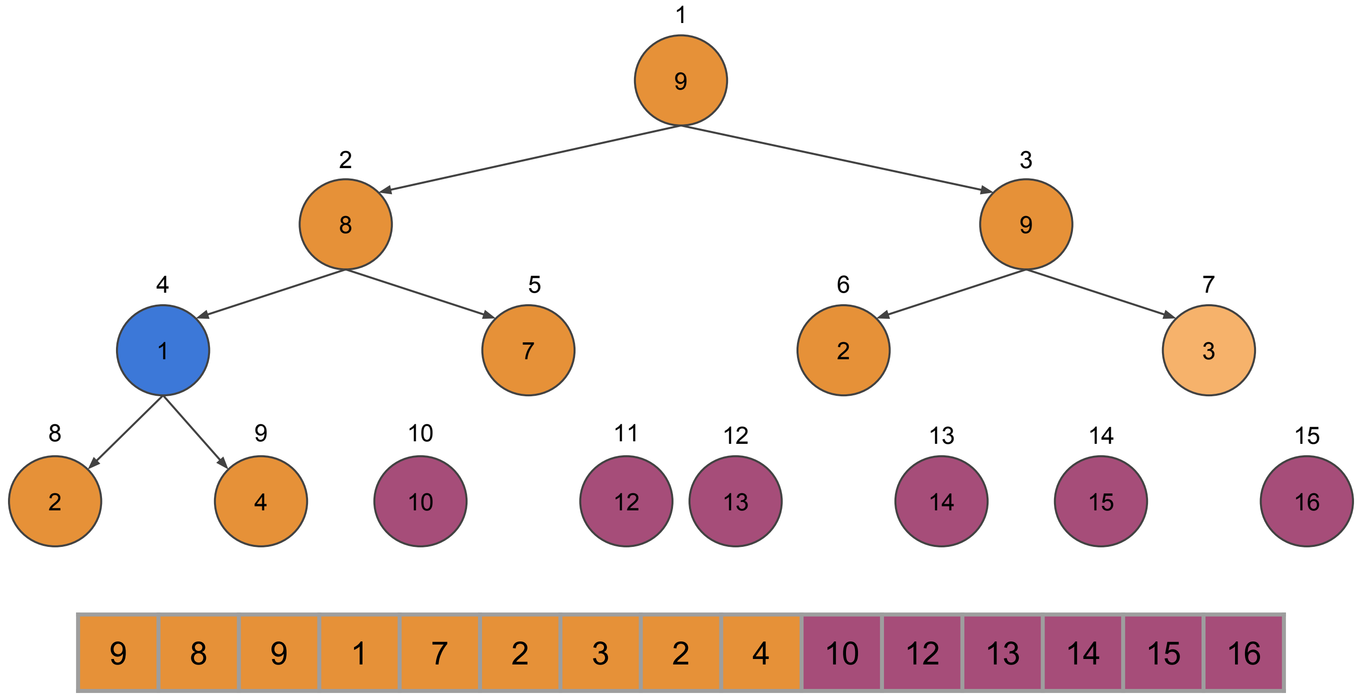
1	9	9	8	7	2	3	2	4	10	12	13	14	15	16

Simulation of Heap Sort



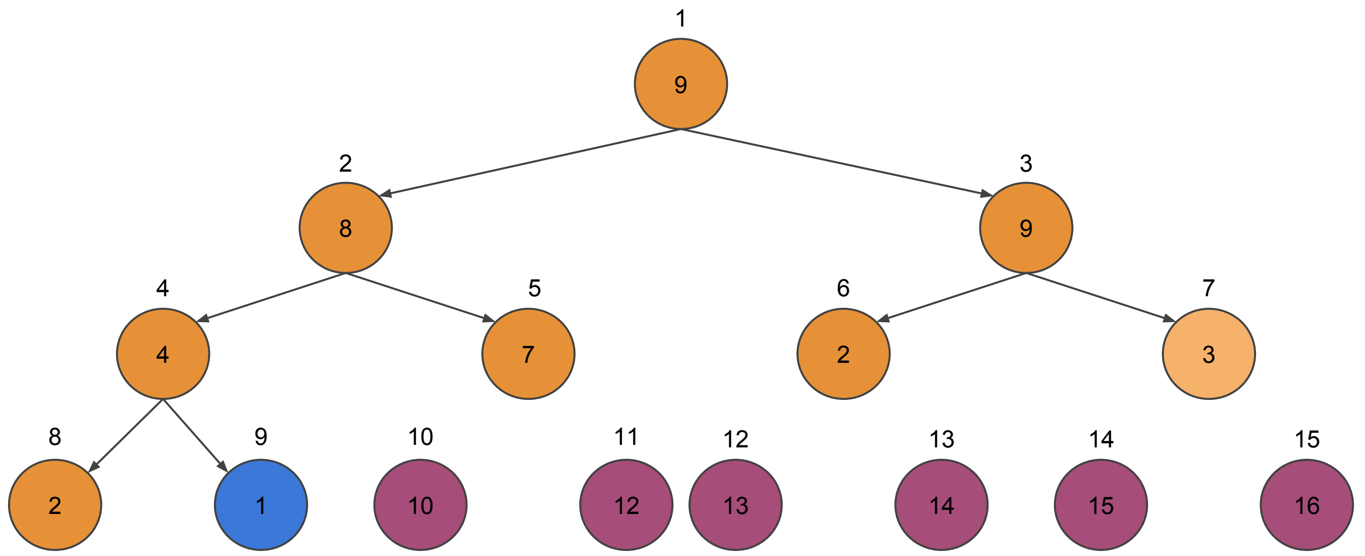
9	1	9	8	7	2	3	2	4	10	12	13	14	15	16

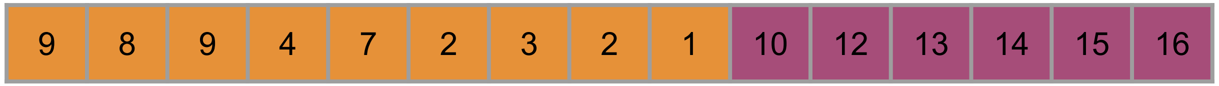
Simulation of Heap Sort



9	8	9	1	7	2	3	2	4	10	12	13	14	15	16

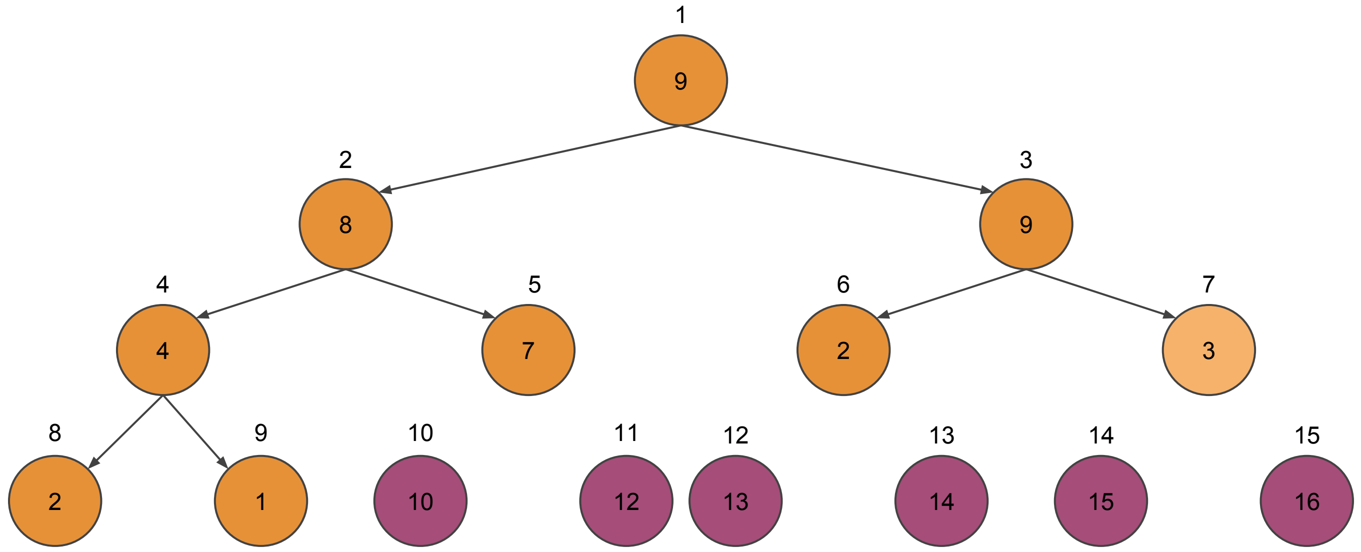
Simulation of Heap Sort

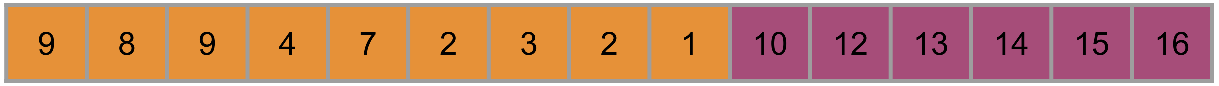




9	8	9	4	7	2	3	2	1	10	12	13	14	15	16

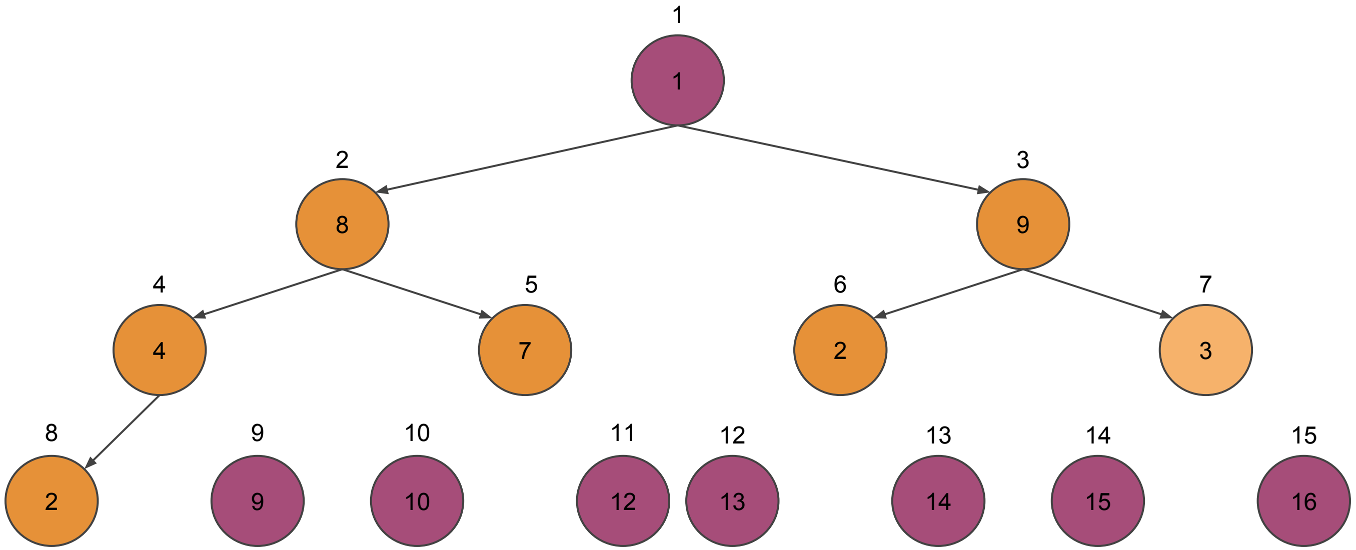
Simulation of Heap Sort





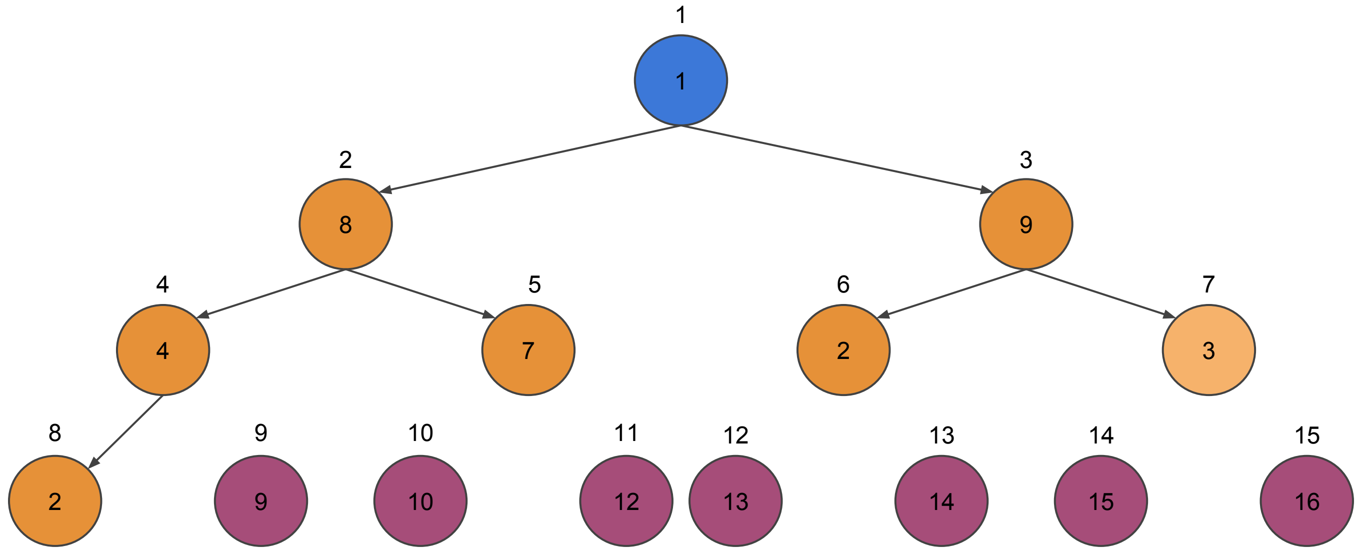
9	8	9	4	7	2	3	2	1	10	12	13	14	15	16

Simulation of Heap Sort



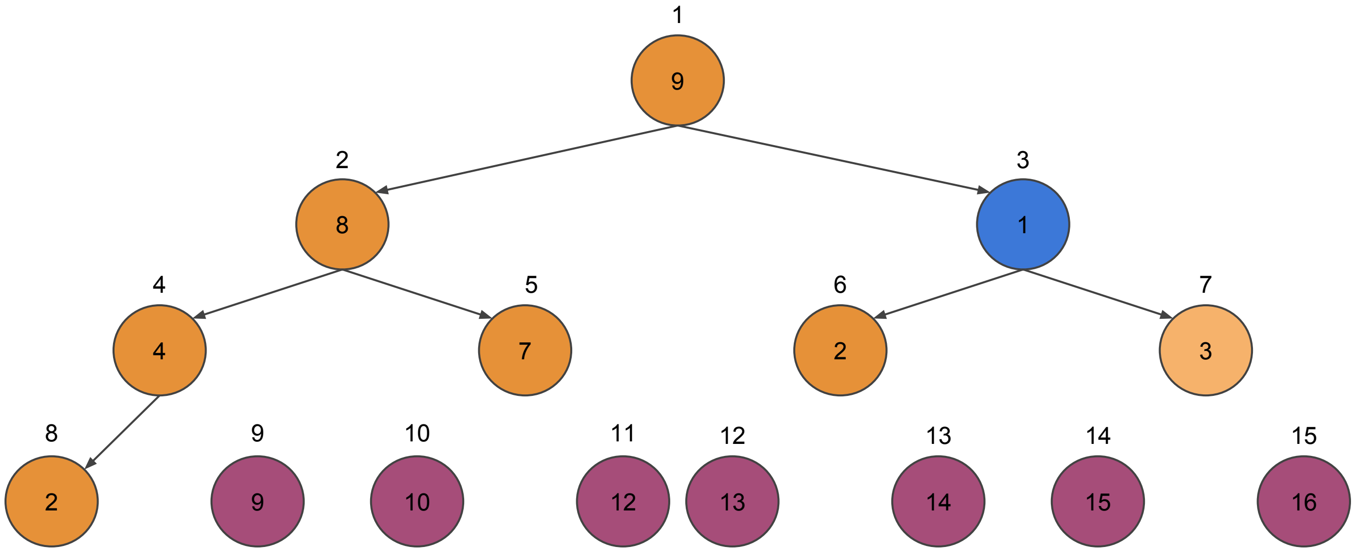
1	8	9	4	7	2	3	2	9	10	12	13	14	15	16

Simulation of Heap Sort



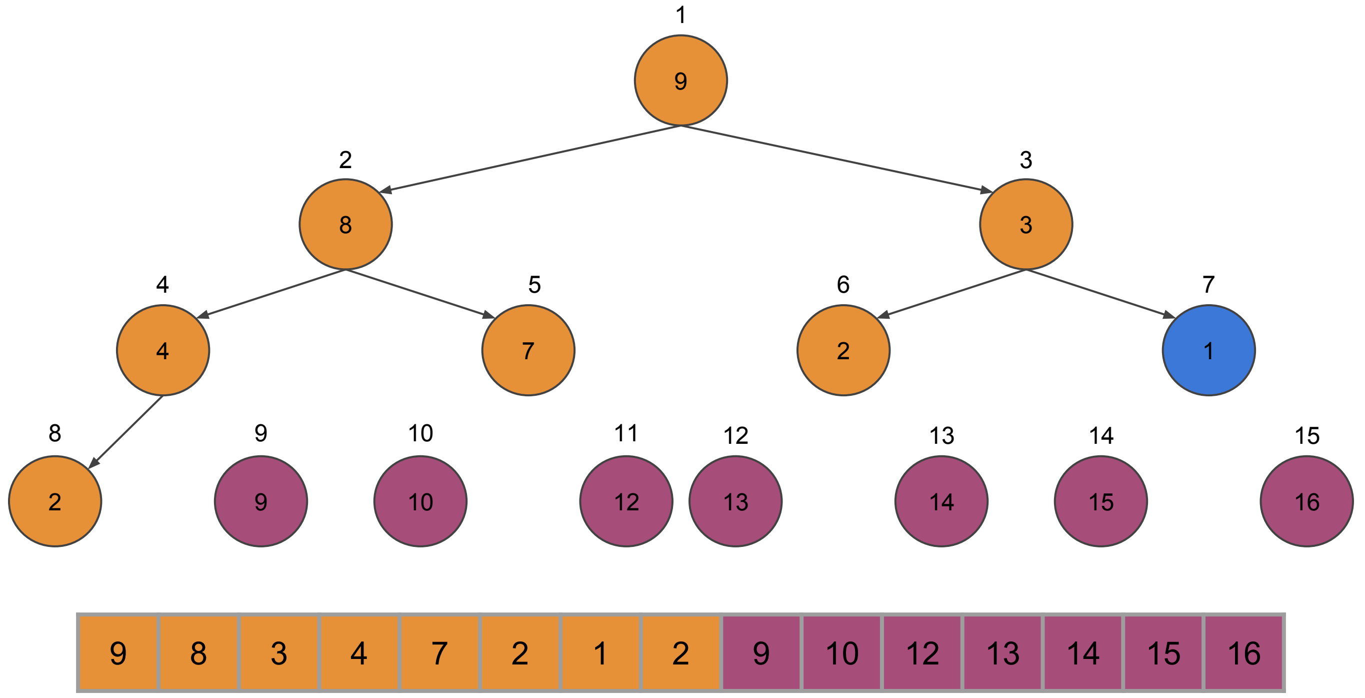
1	8	9	4	7	2	3	2	9	10	12	13	14	15	16

Simulation of Heap Sort

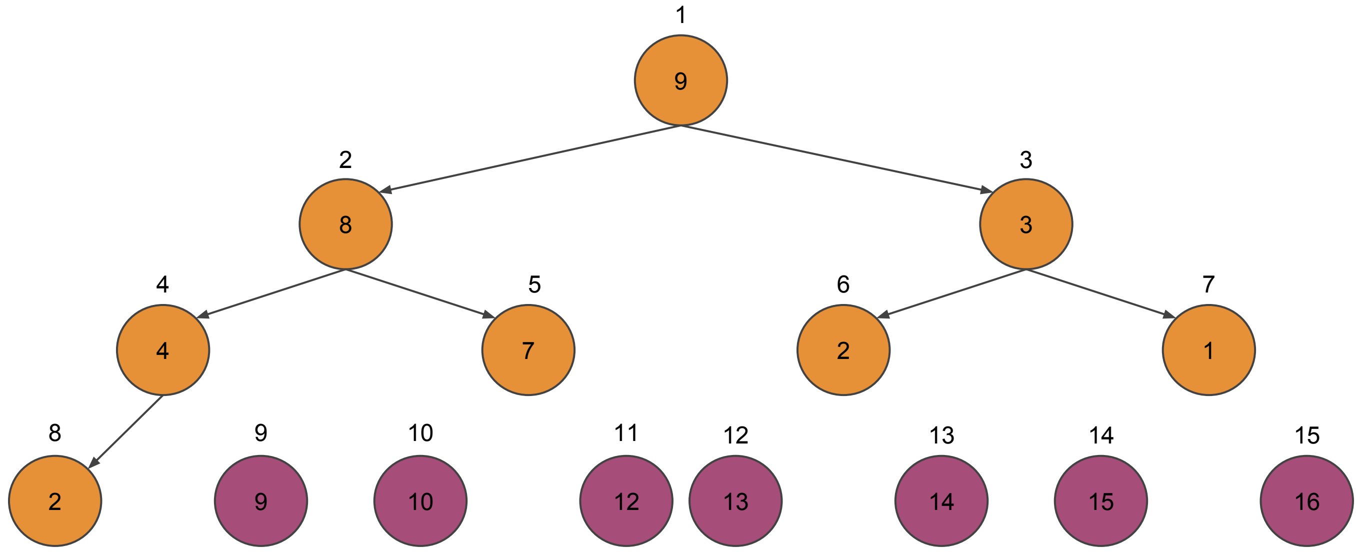


9	8	1	4	7	2	3	2	9	10	12	13	14	15	16

Simulation of Heap Sort

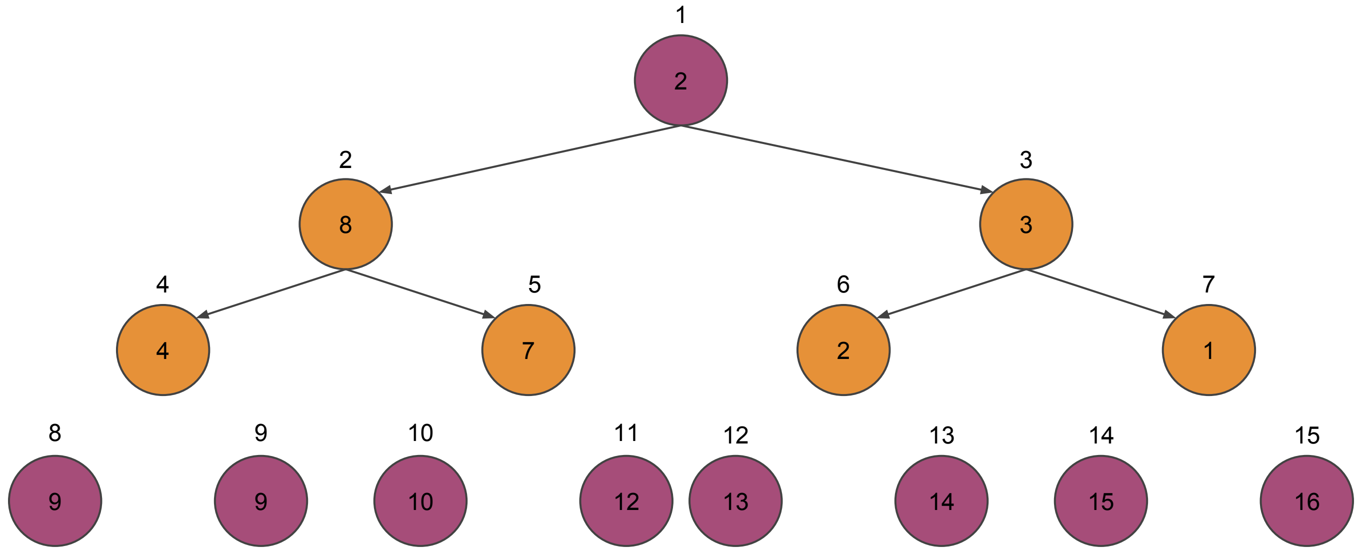


Simulation of Heap Sort



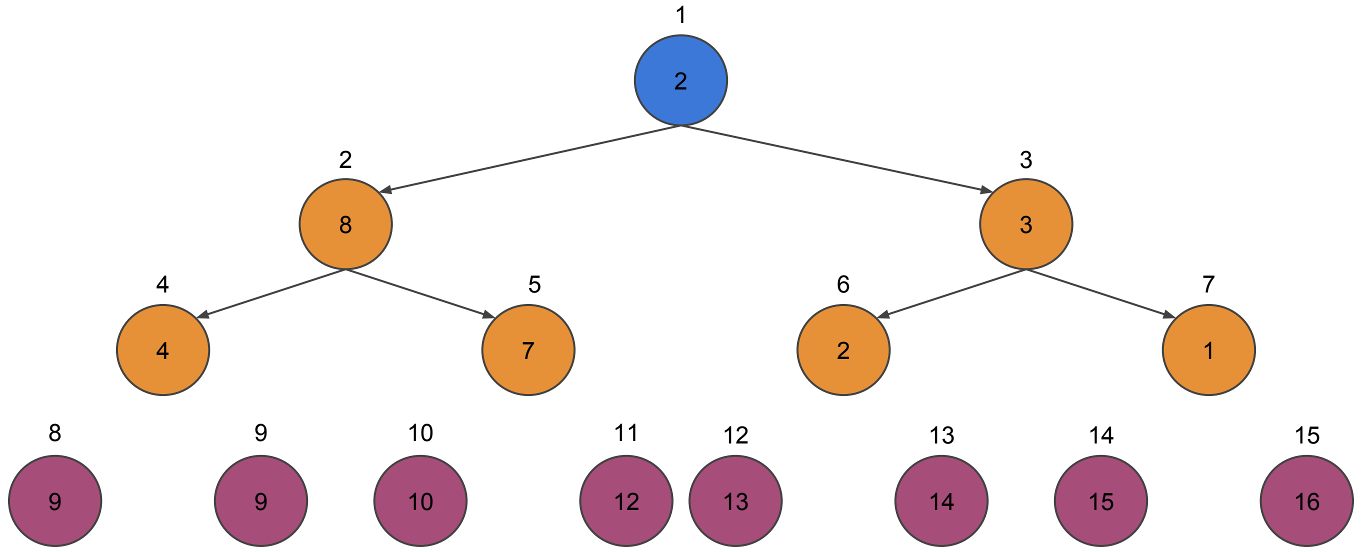
9	8	3	4	7	2	1	2	9	10	12	13	14	15	16

Simulation of Heap Sort



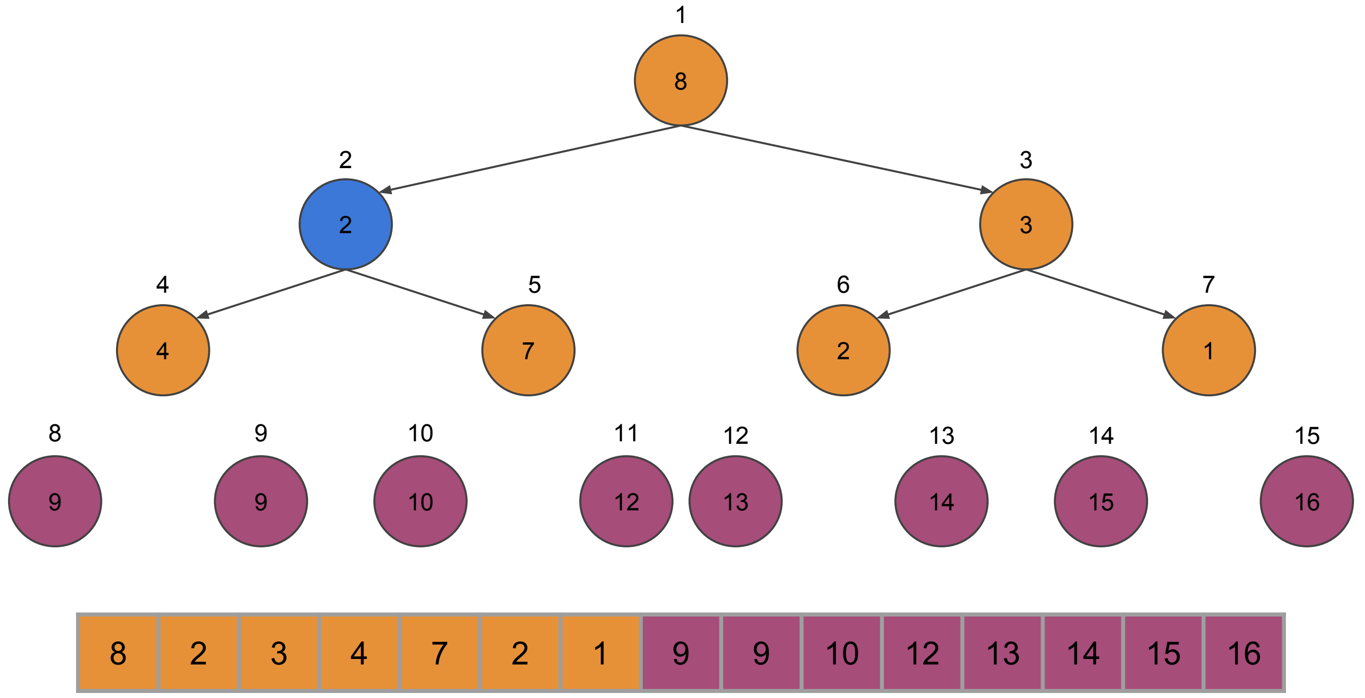
2	8	3	4	7	2	1	9	9	10	12	13	14	15	16

Simulation of Heap Sort

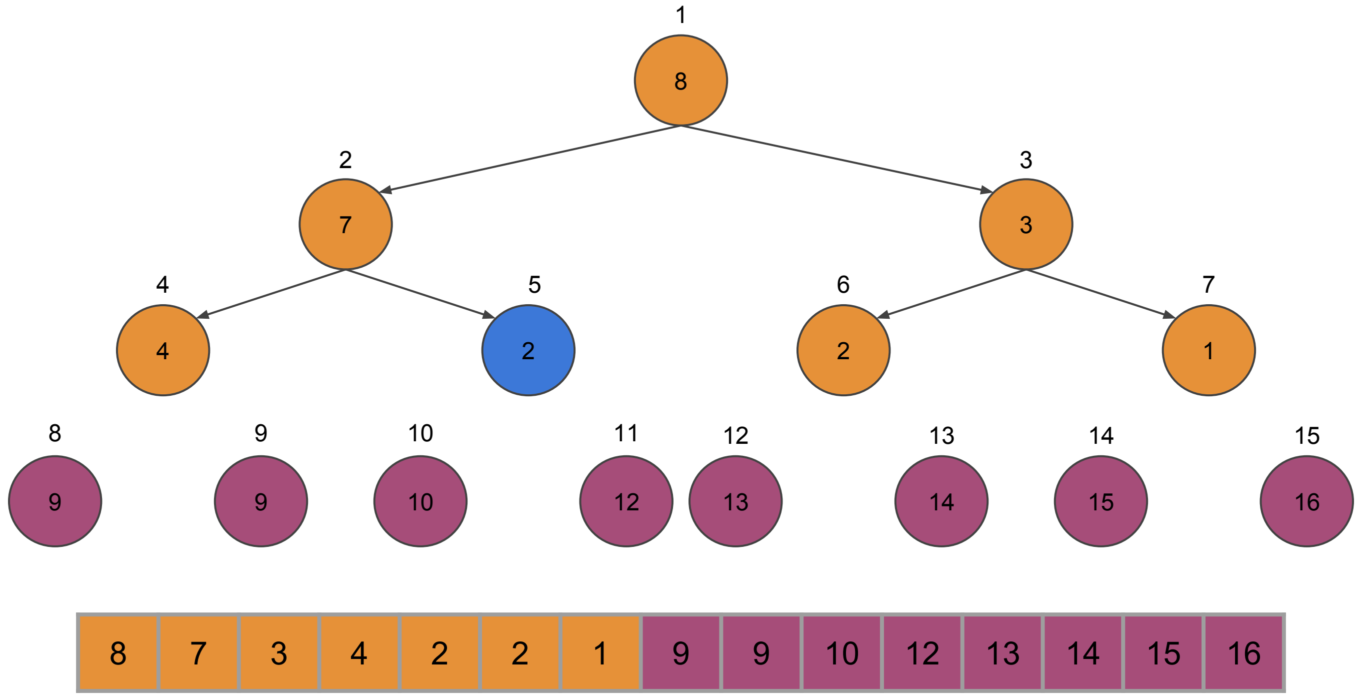


2	8	3	4	7	2	1	9	9	10	12	13	14	15	16

Simulation of Heap Sort

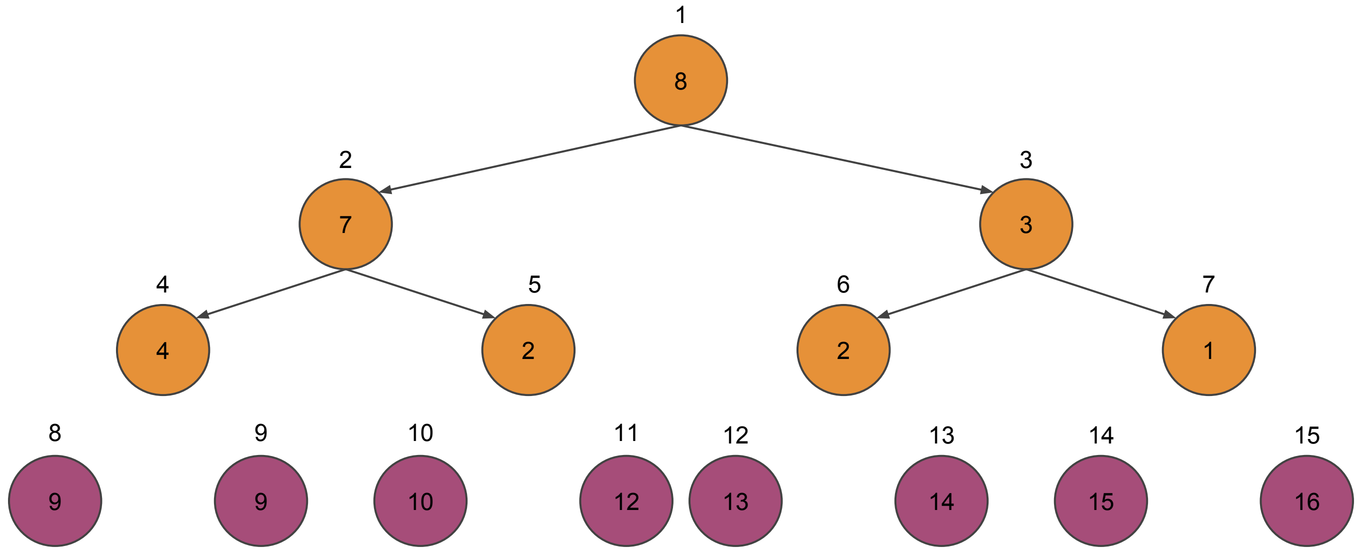


Simulation of Heap Sort



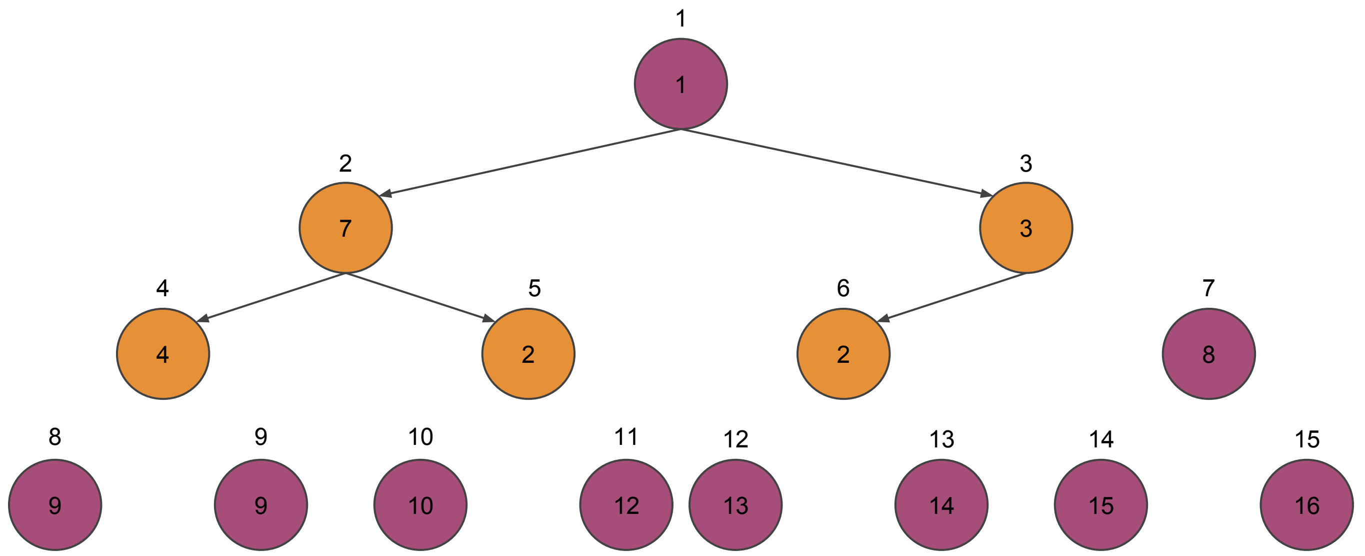
8	7	3	4	2	2	1	9	9	10	12	13	14	15	16

Simulation of Heap Sort



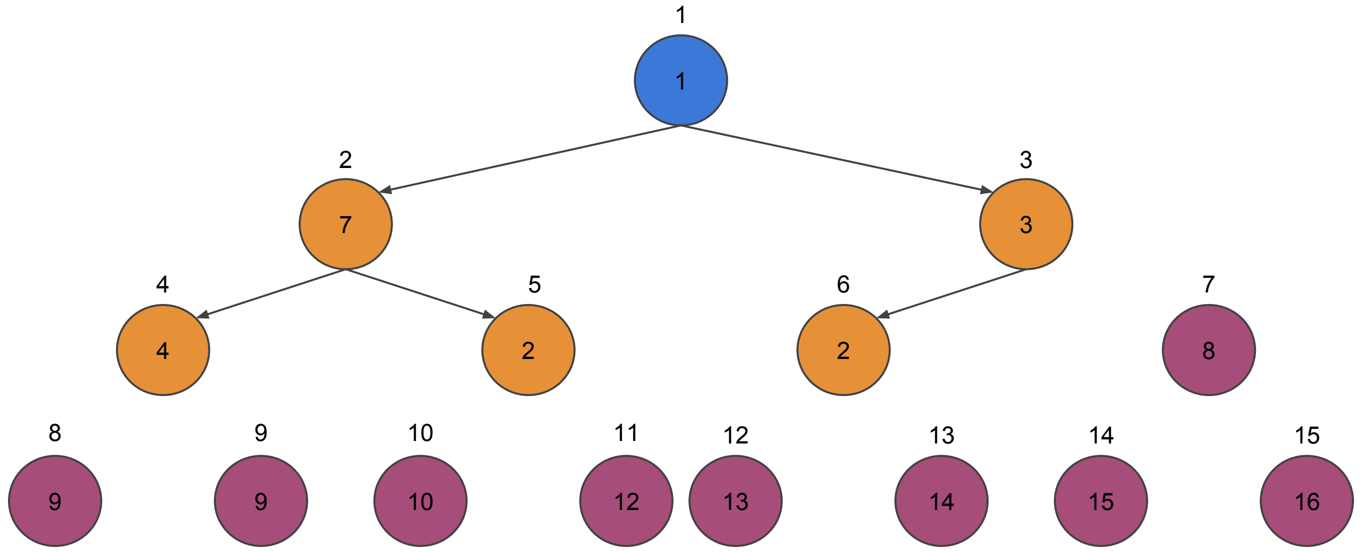
8	7	3	4	2	2	1	9	9	10	12	13	14	15	16

Simulation of Heap Sort



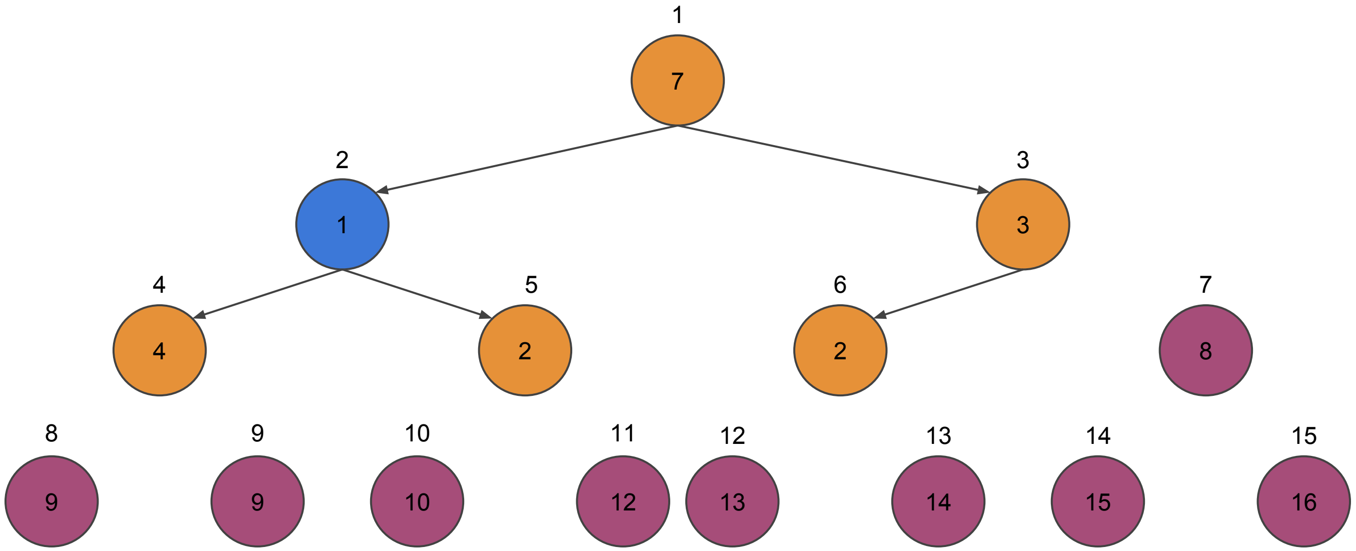
1	7	3	4	2	2	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



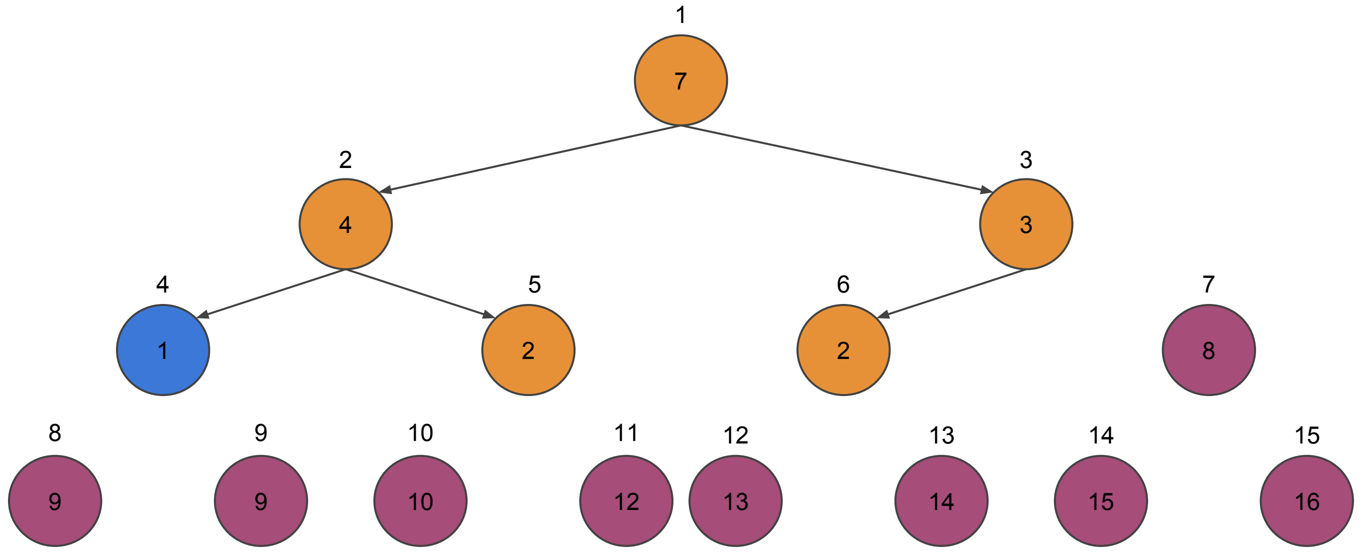
1	7	3	4	2	2	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



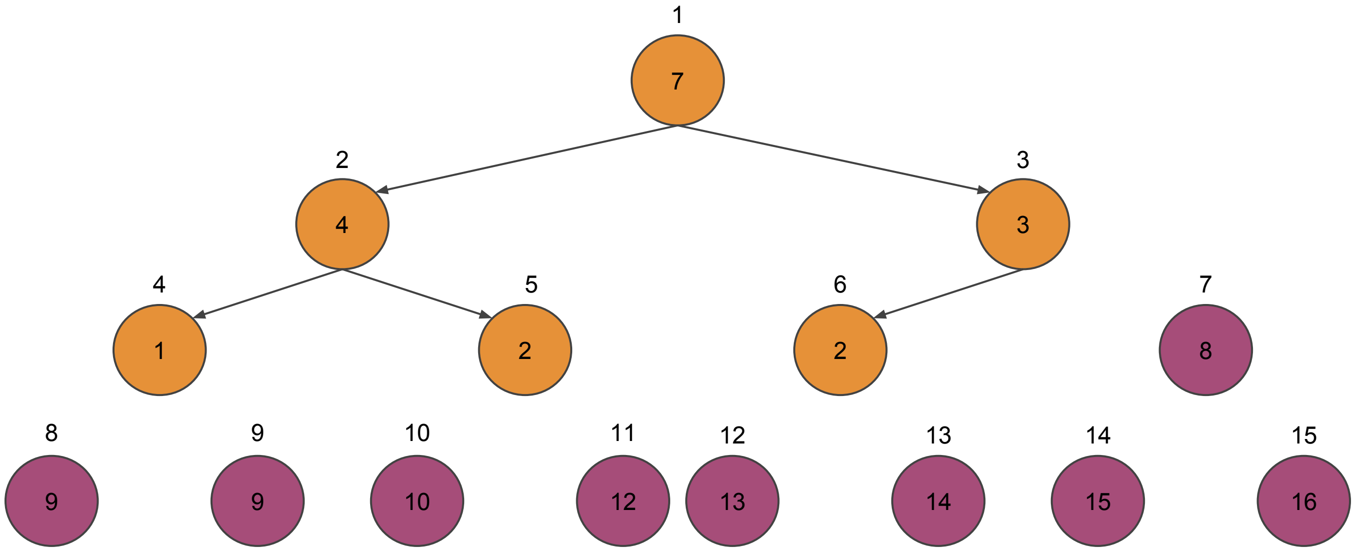
7	1	3	4	2	2	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



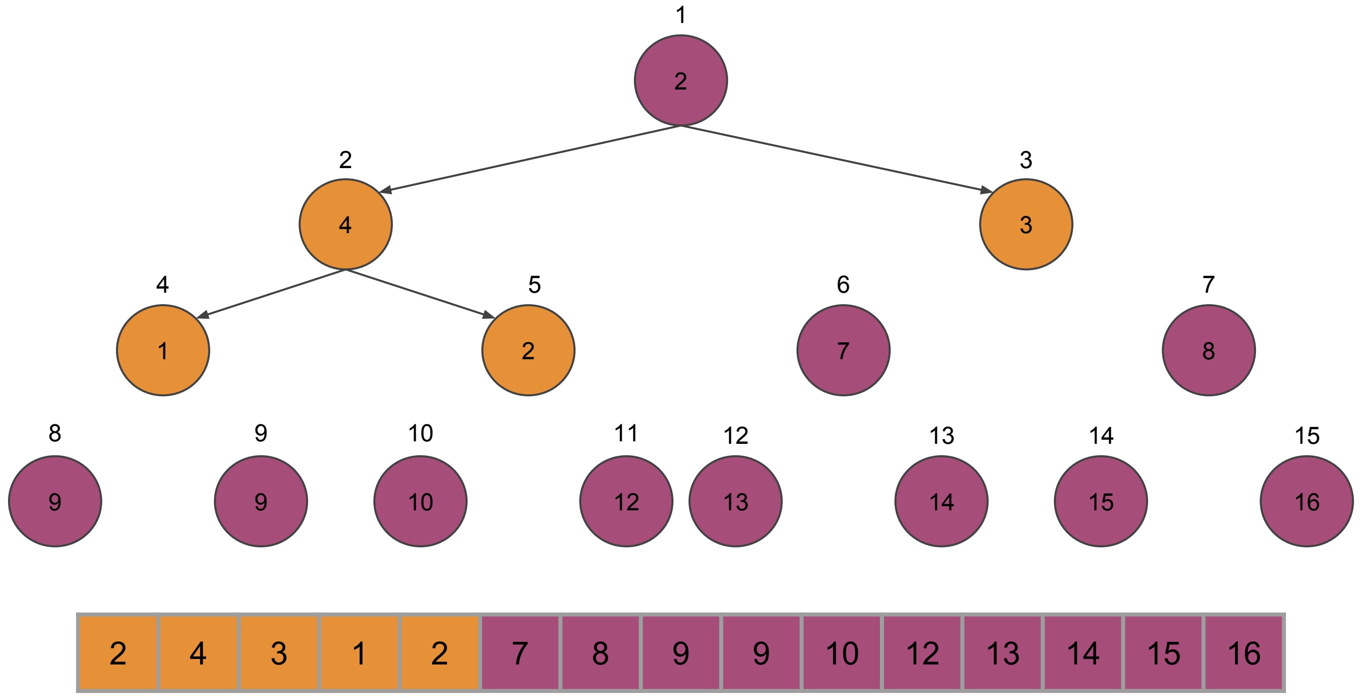
7	4	3	1	2	2	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort

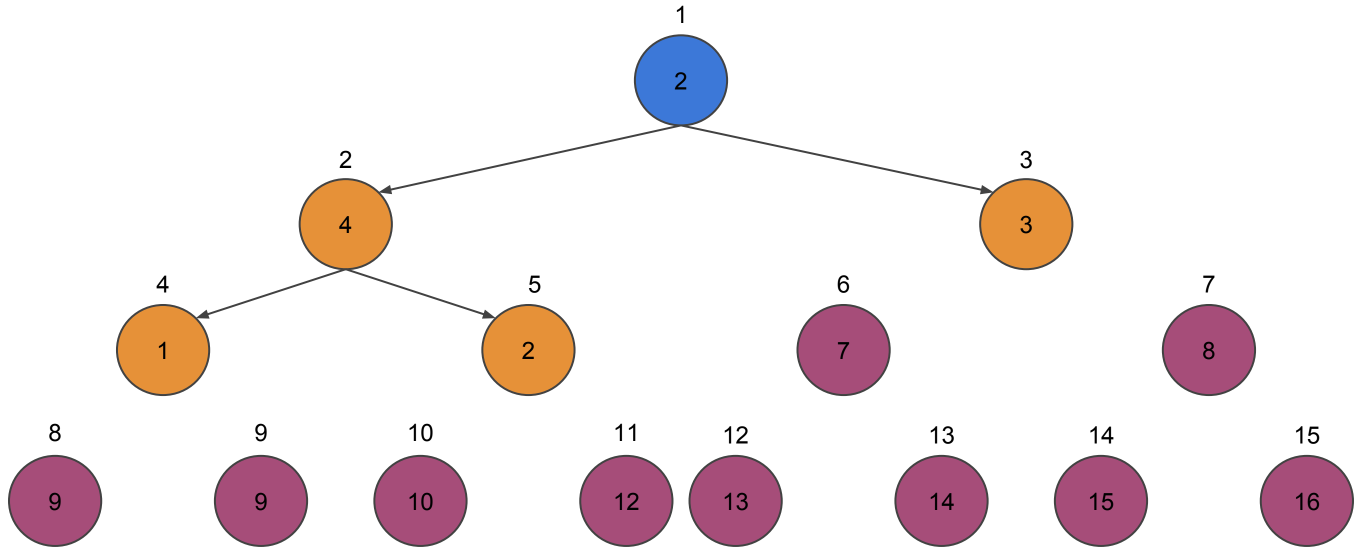


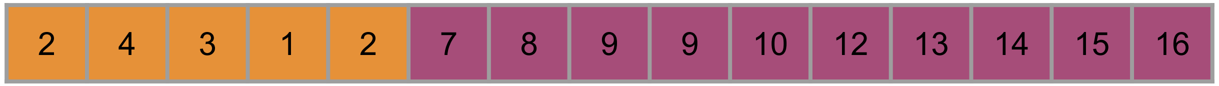
7	4	3	1	2	2	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



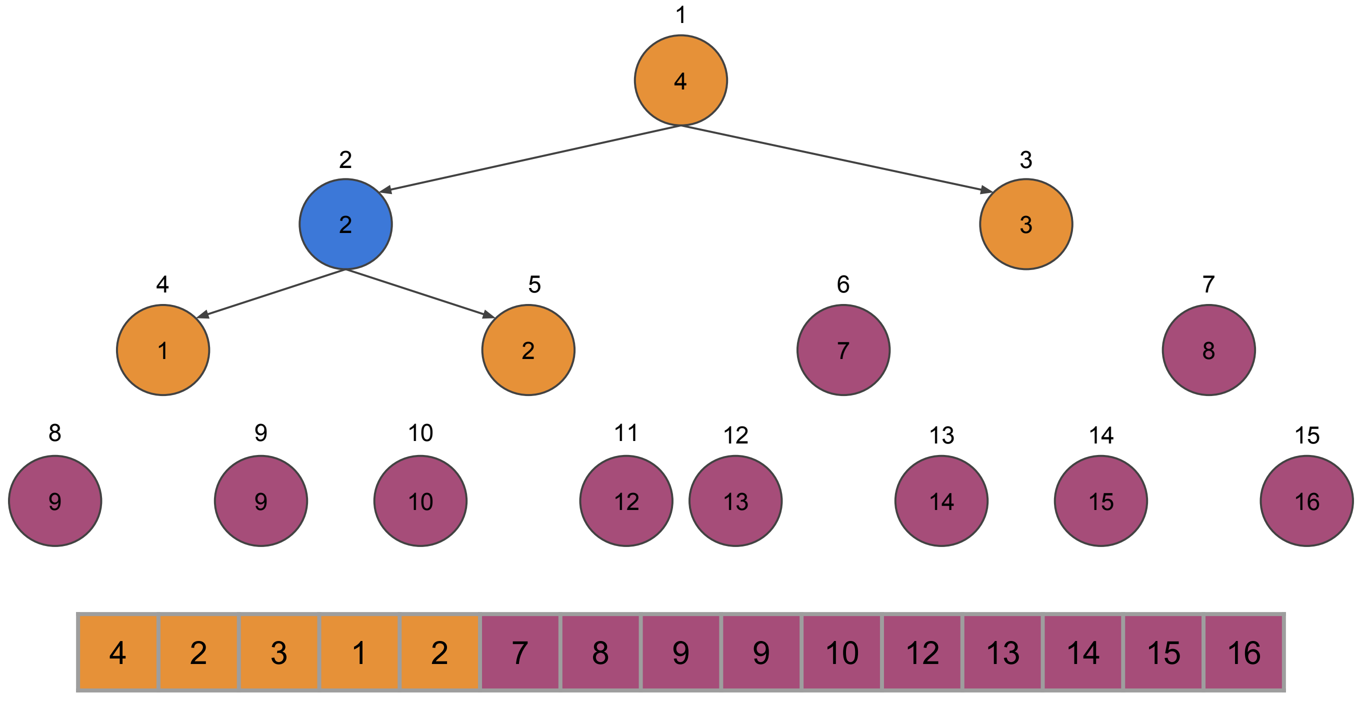
Simulation of Heap Sort





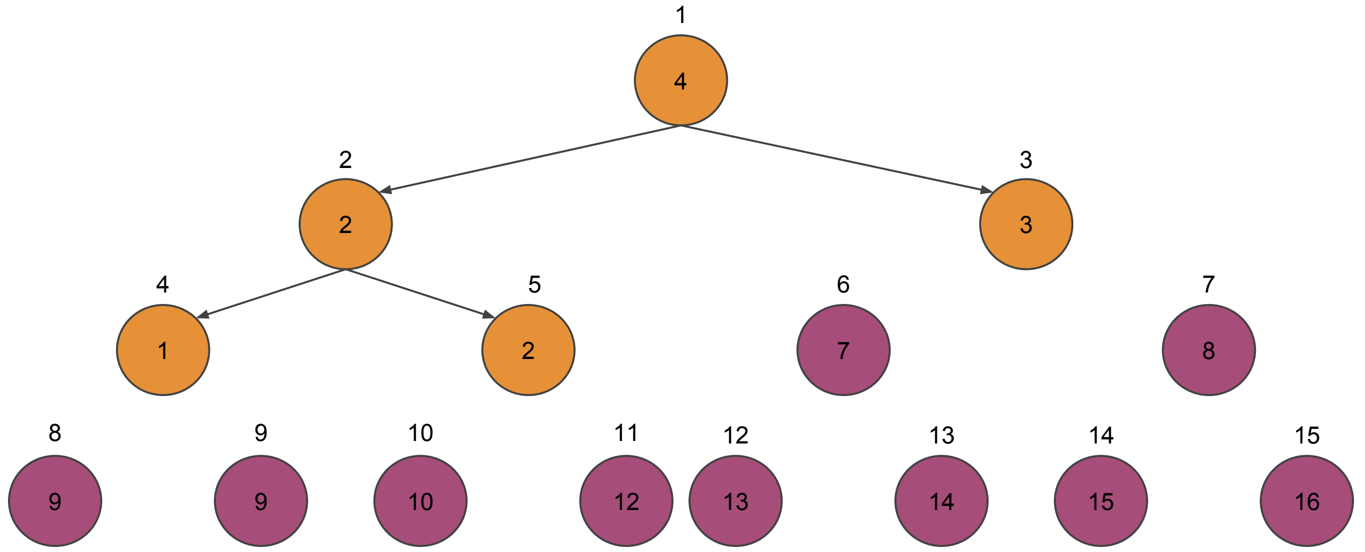
2	4	3	1	2	7	8	9	9	10	12	13	14	15	16

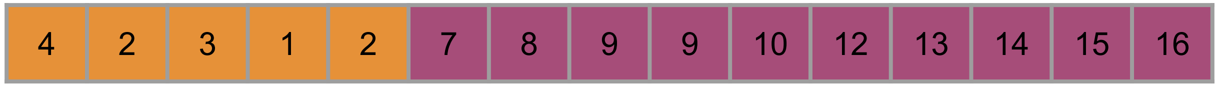
Simulation of Heap Sort



4	2	3	1	2	7	8	9	9	10	12	13	14	15	16

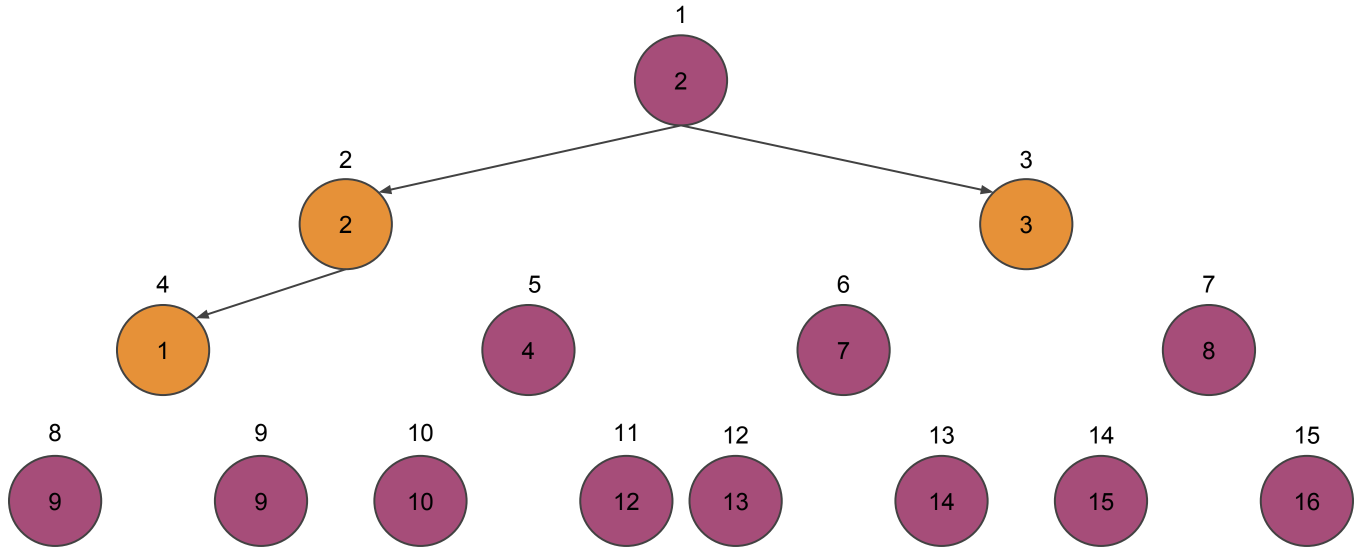
Simulation of Heap Sort





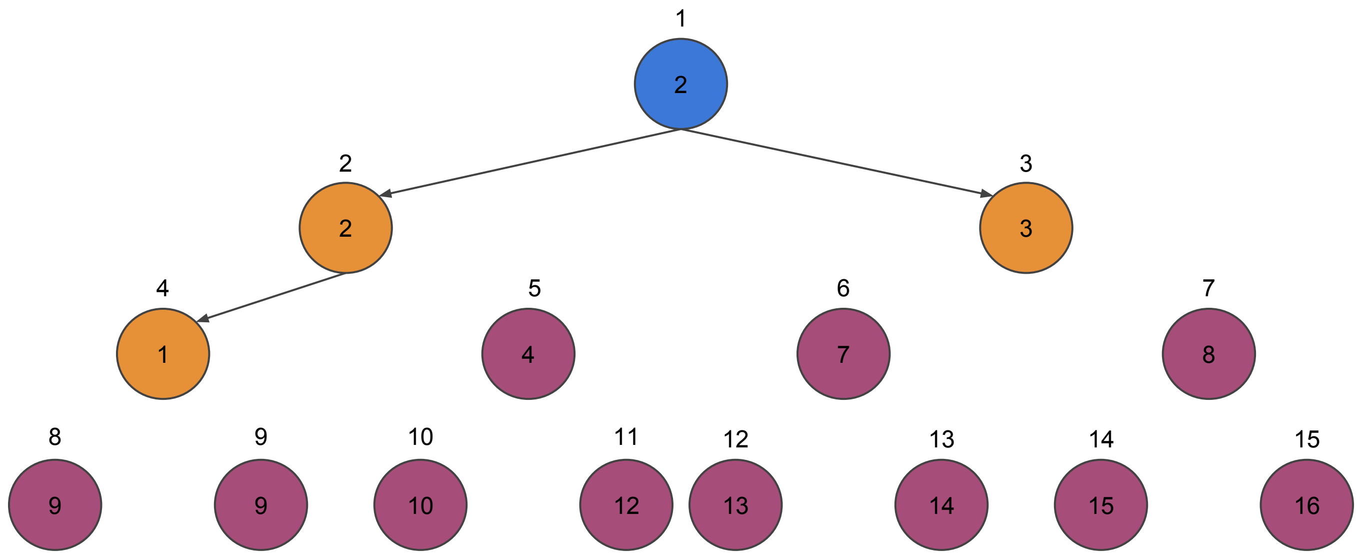
4	2	3	1	2	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



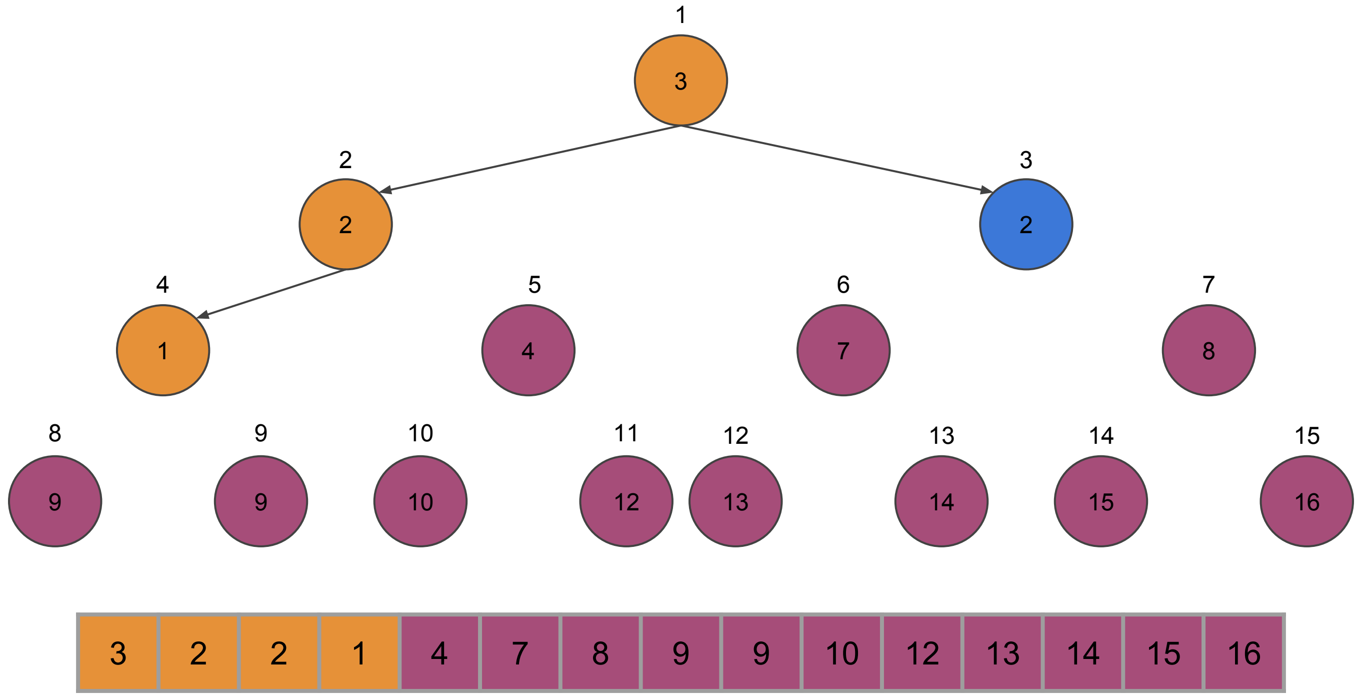
2	2	3	1	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort

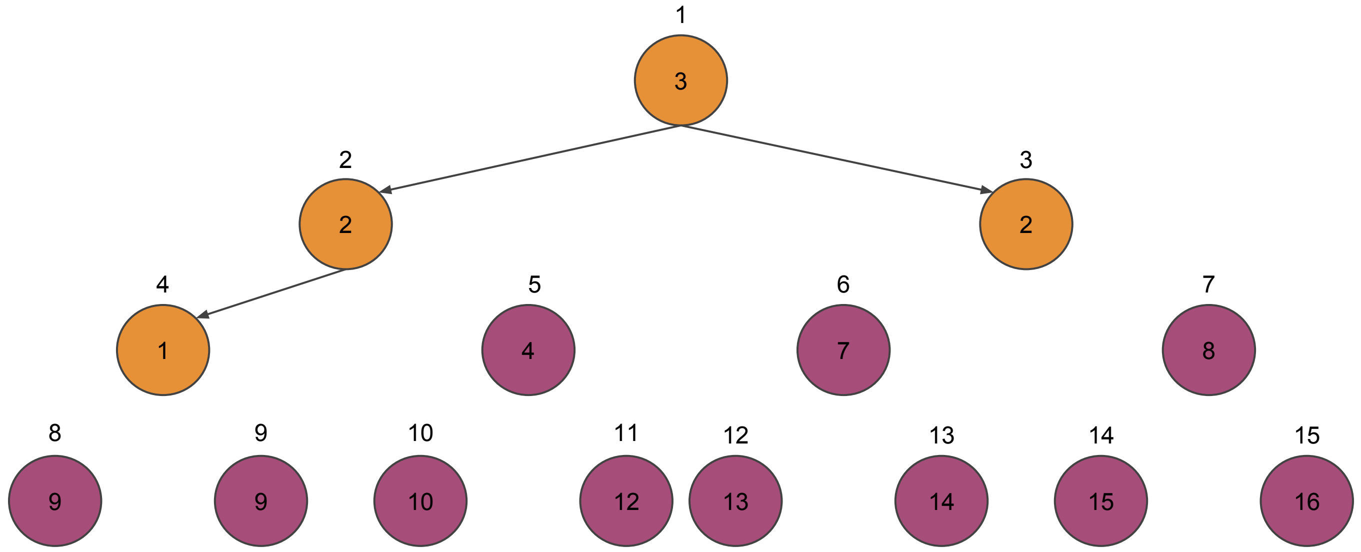


2	2	3	1	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort

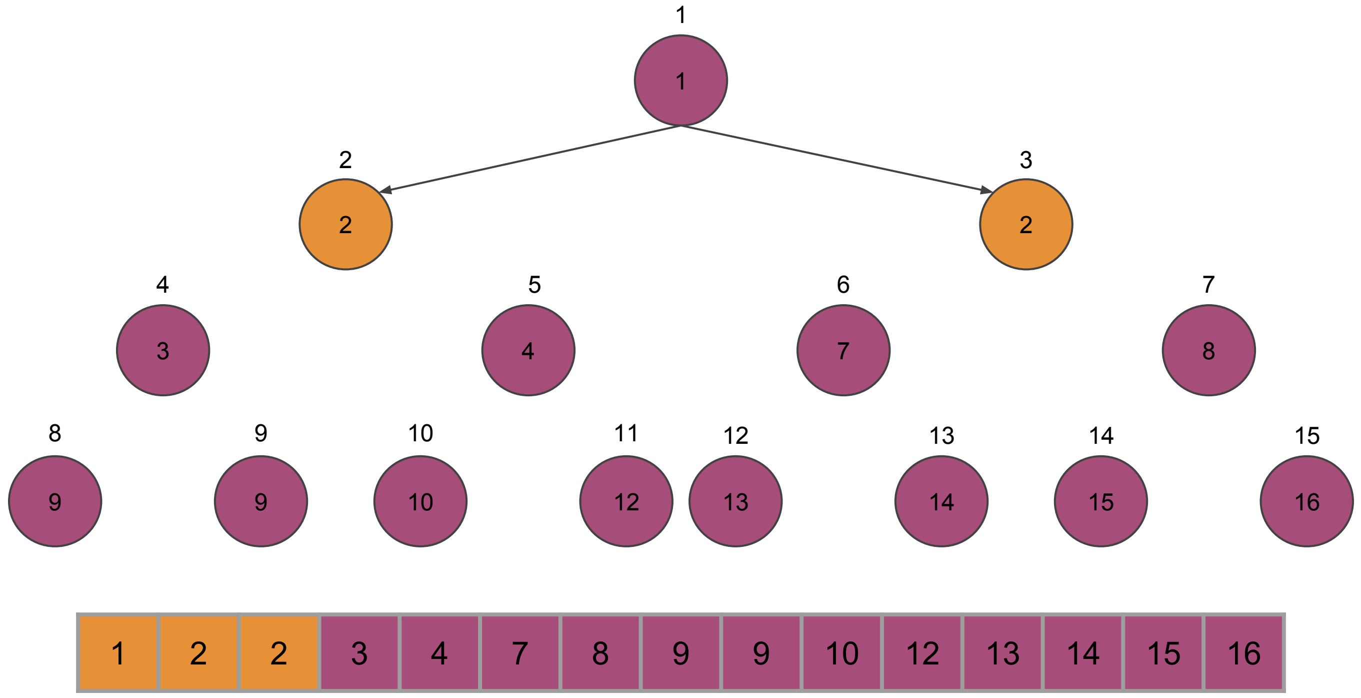


Simulation of Heap Sort

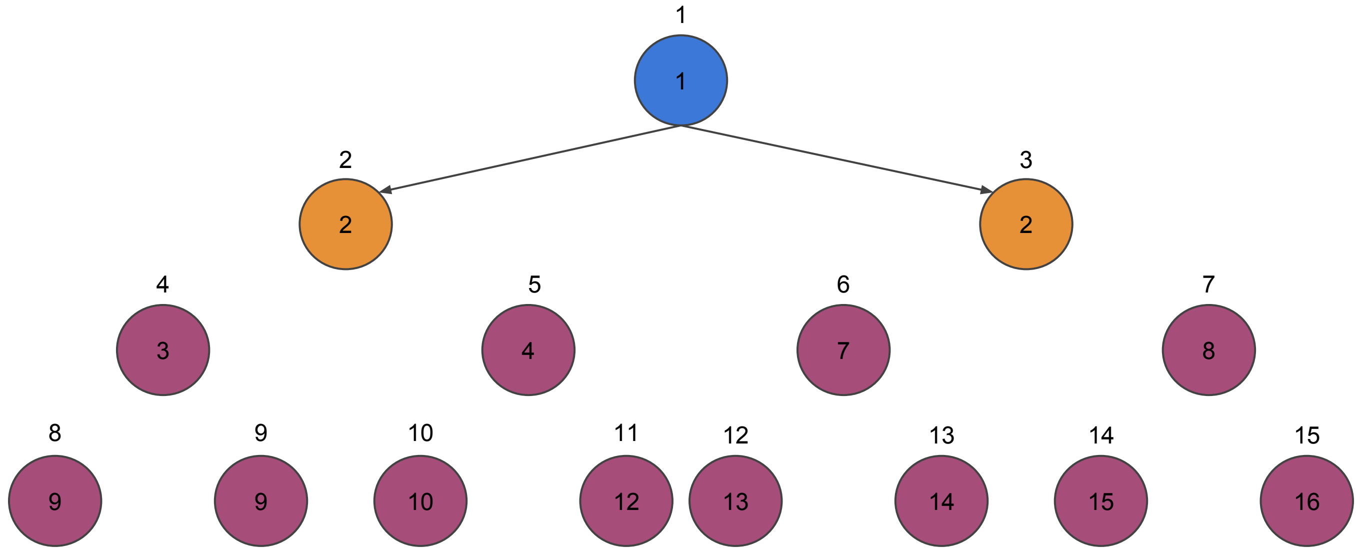


3	2	2	1	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort

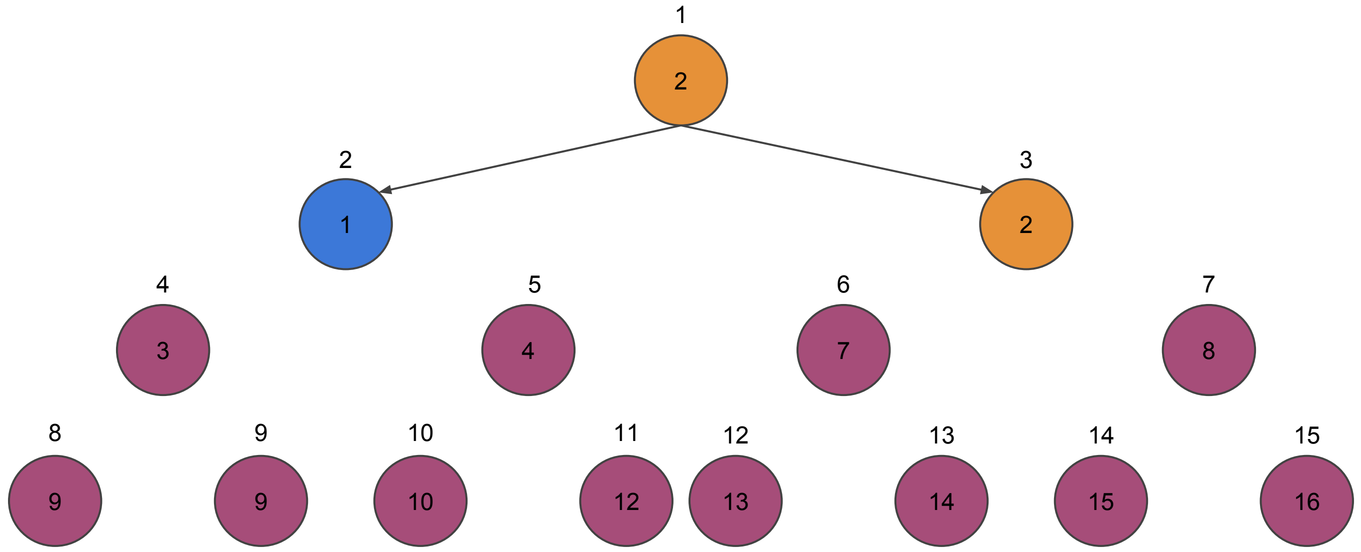


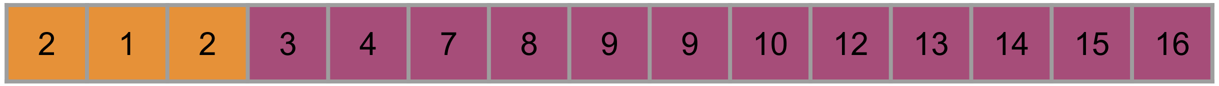
Simulation of Heap Sort



1	2	2	3	4	7	8	9	9	10	12	13	14	15	16

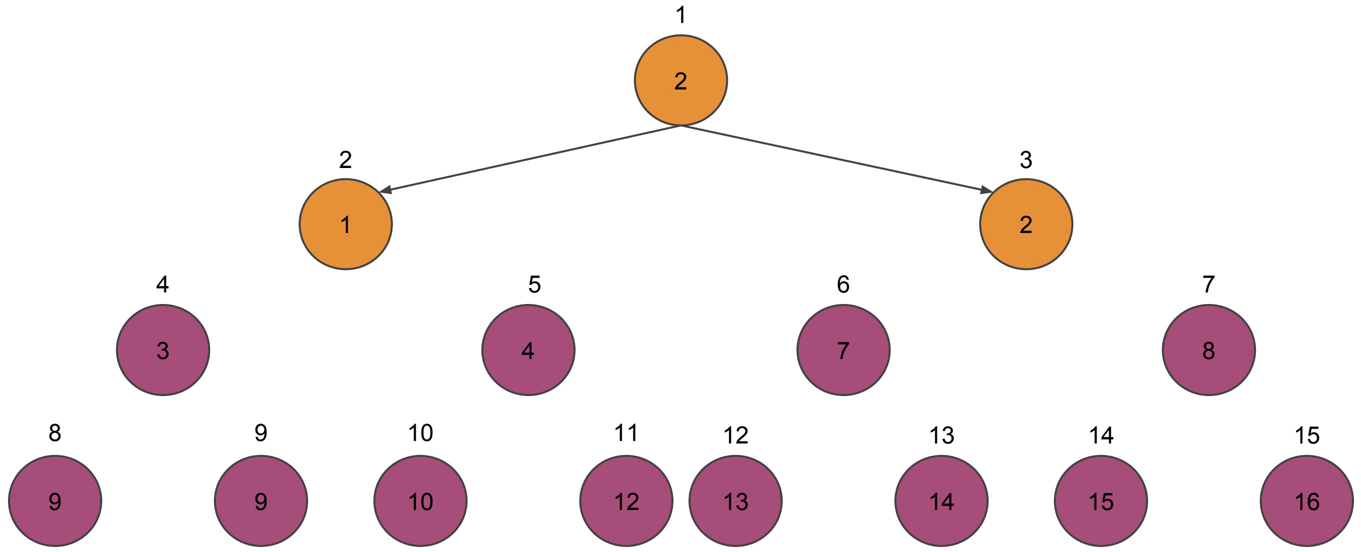
Simulation of Heap Sort





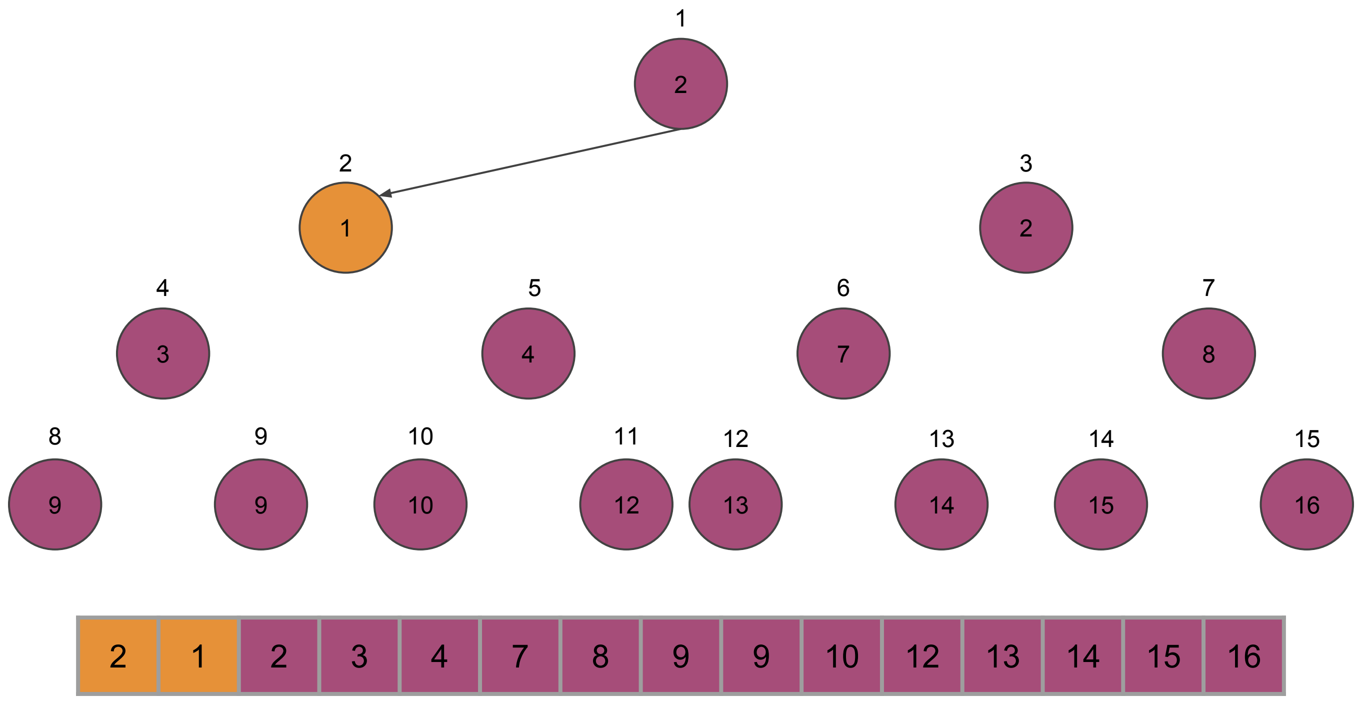
2	1	2	3	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



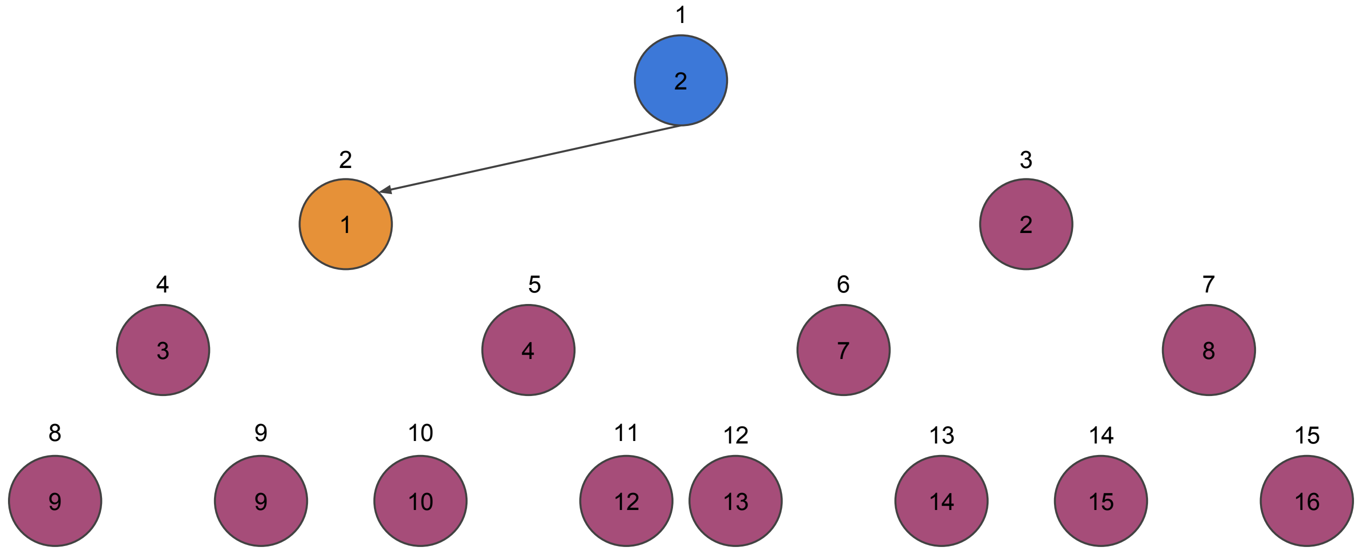
2	1	2	3	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



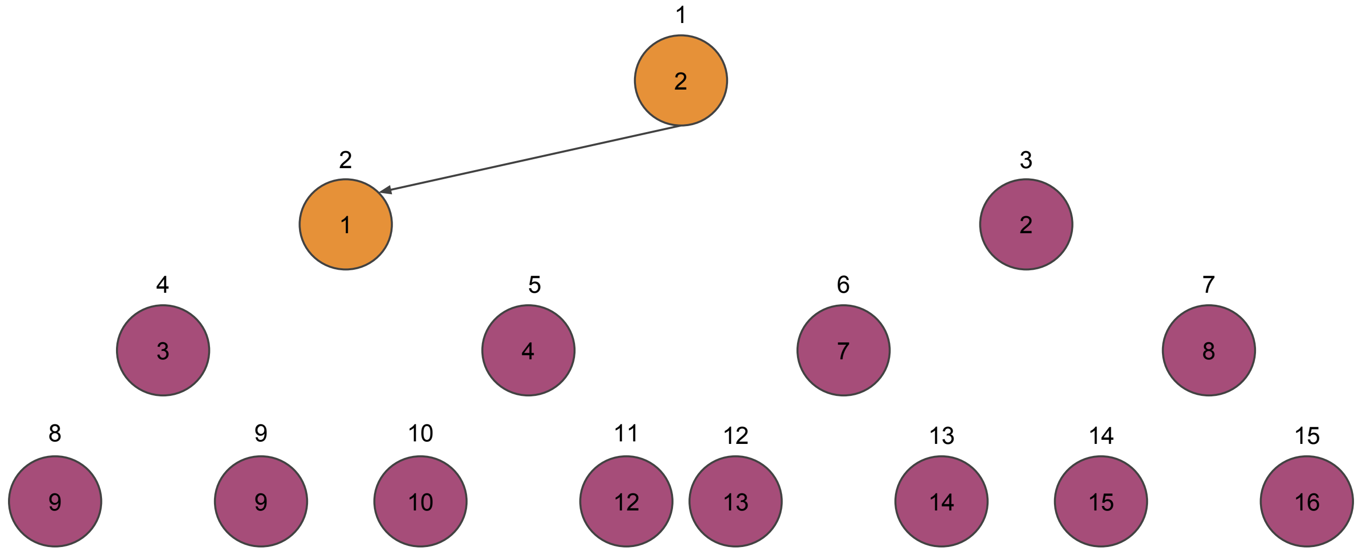
2	1	2	3	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



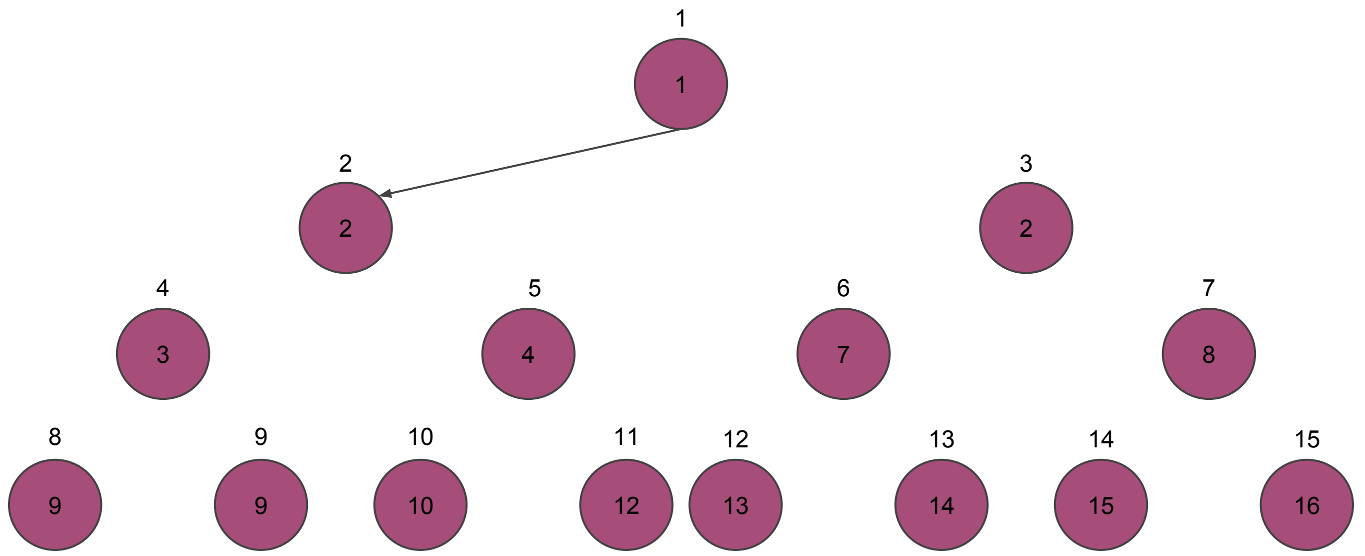
2	1	2	3	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort



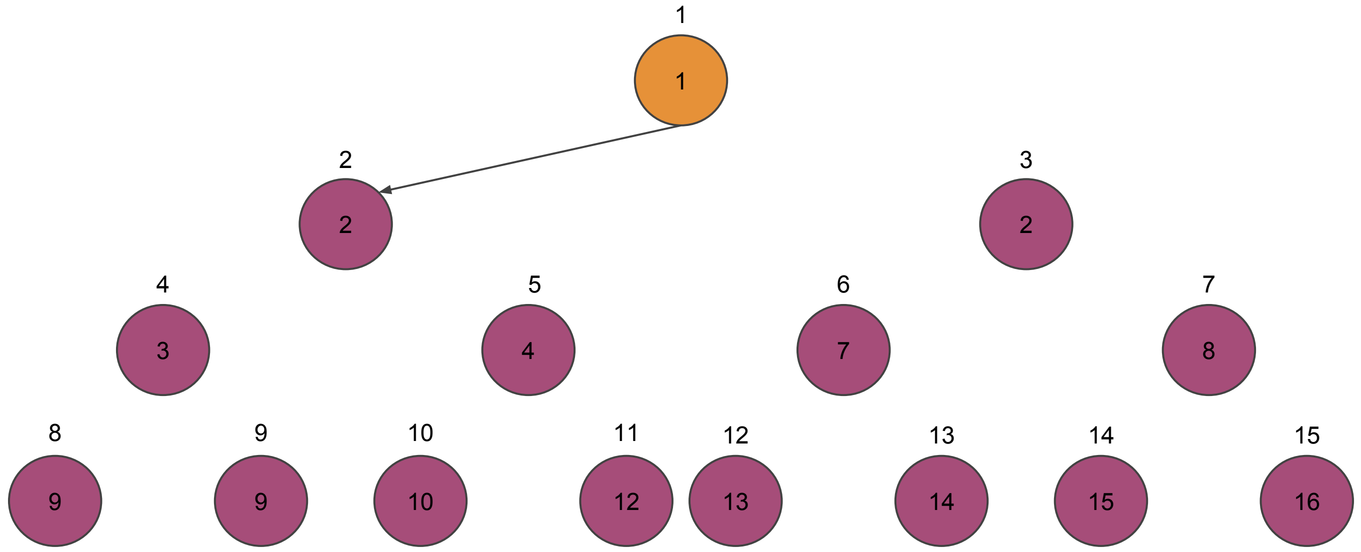
2	1	2	3	4	7	8	9	9	10	12	13	14	15	16

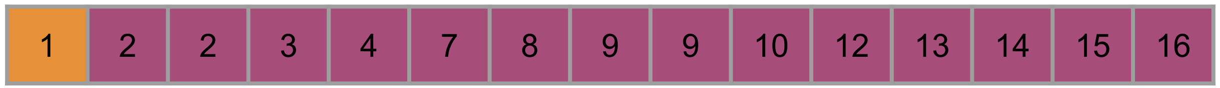
Simulation of Heap Sort



1	2	2	3	4	7	8	9	9	10	12	13	14	15	16

Simulation of Heap Sort





1	2	2	3	4	7	8	9	9	10	12	13	14	15	16

Priority Queue

Heapsort is an excellent algorithm, but a good implementation of quicksort, taught before, usually beats it in practice. Nevertheless, the heap data structure itself has many uses. One of the most popular application of heap lies beyond sorting. It can used as an efficient priority queue. As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We will focus here on how to implement max-priority queues, which are in turn based on max heaps.

Priority Queue Continued

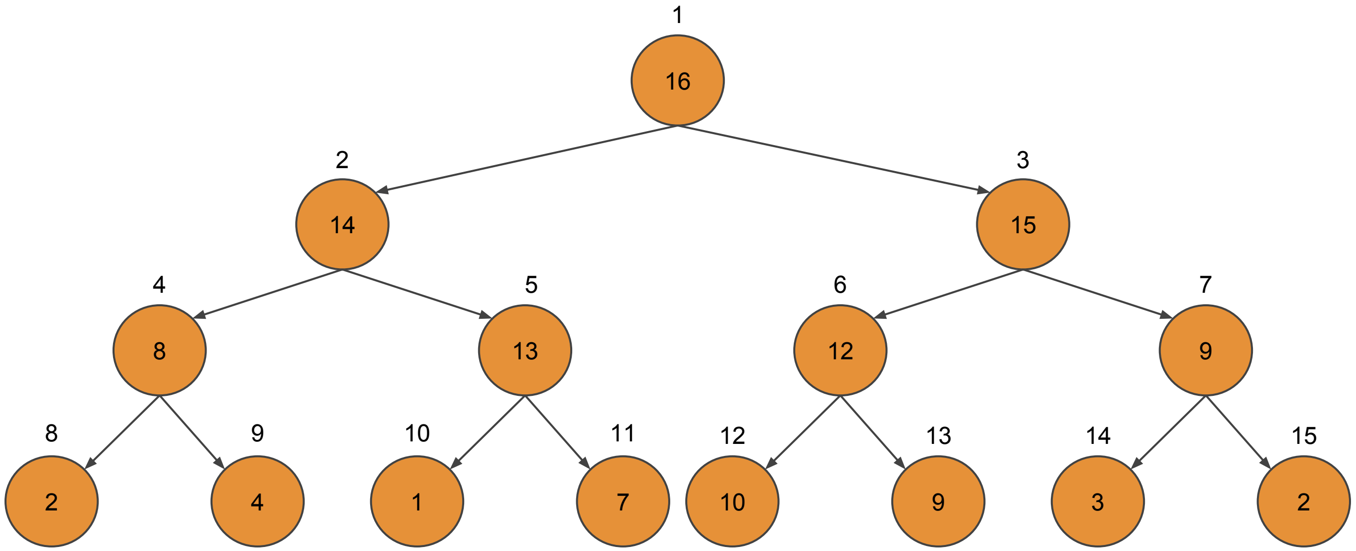
A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max priority queue supports the following operations:

1. **Maximum(S):** returns the element of S with the largest key.
2. **Extract_Max(S):** removes and returns the element of S with the largest key.
3. **Increase_Key(S, x, k):** increases the value of element x’s key to the new value k, which is assumed to be at least as large as x’s current key value.
4. **Insert(S, x):** inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

Maximum

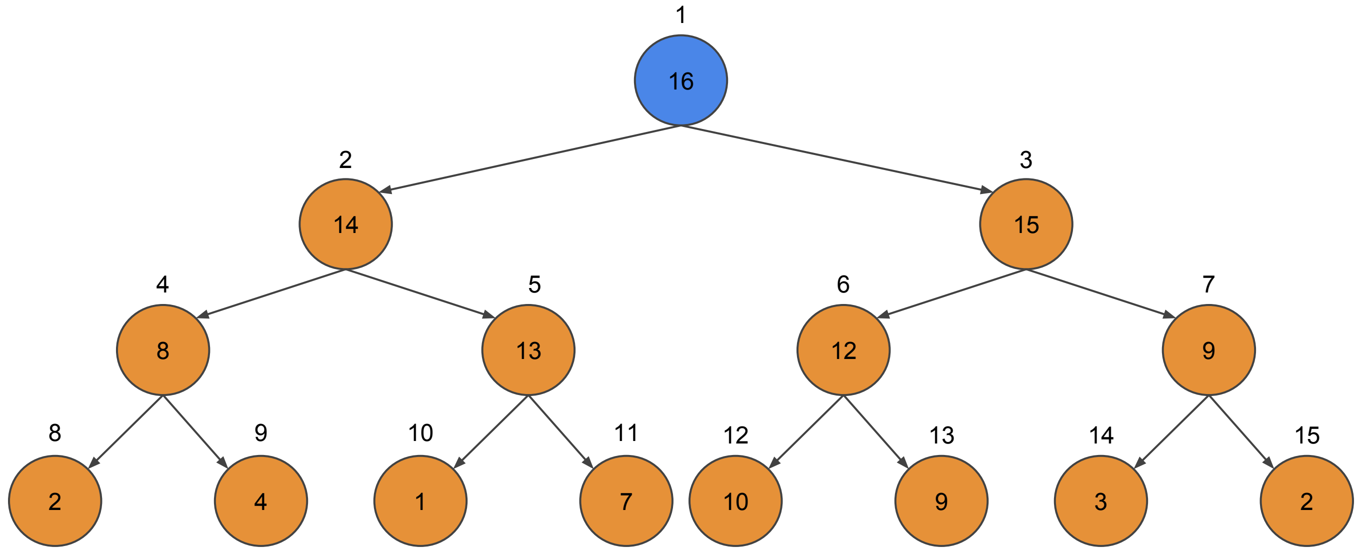
```
**Maximum**(S)
    return A[1]
```

Simulation of Maximum



16	14	15	8	13	12	9	2	4	1	7	10	9	3	2

Simulation of Maximum Continued

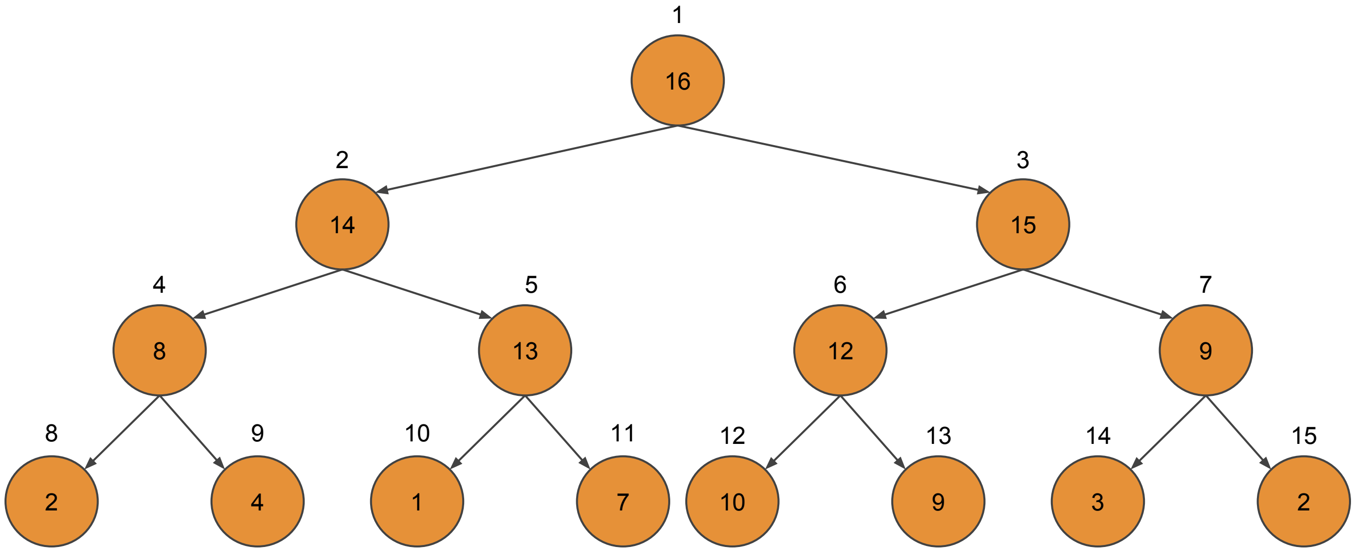


16	14	15	8	13	12	9	2	4	1	7	10	9	3	2

Extract Max

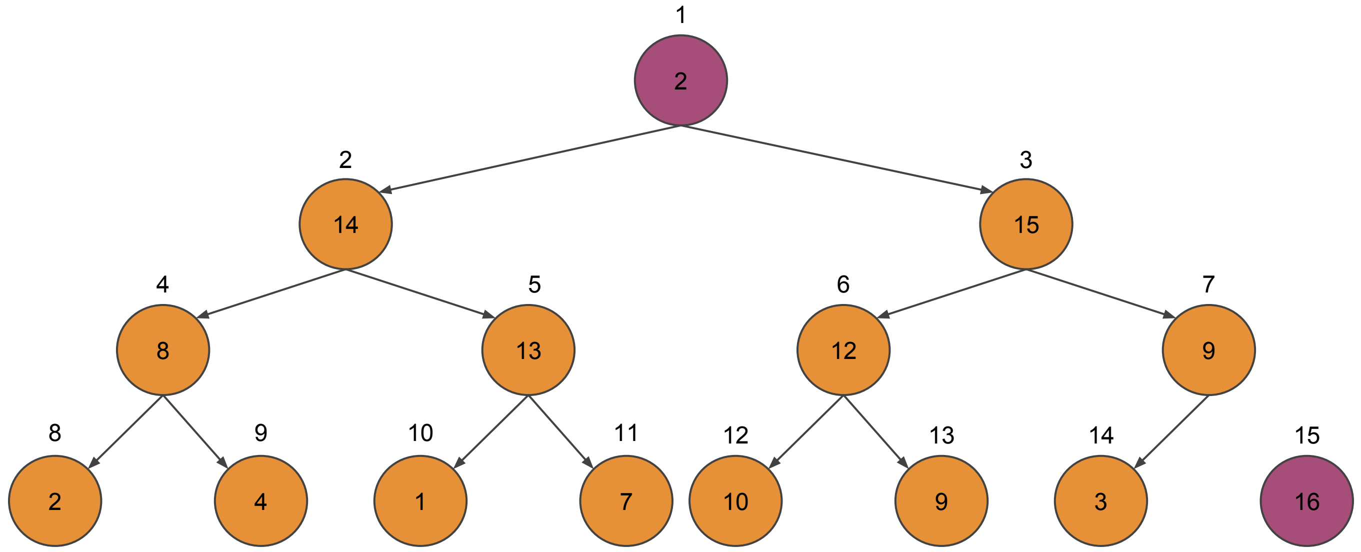
```
**Extract_Max**(S)
    if A.heap-size < 1
        error “heap underflow”
    max = A[1]
    A[1] = A[A:heap-size]
    A.heap-size = A.heap-size  1
    Max_Heapify(A, 1)
    return max
```

Simulation of Extract Max



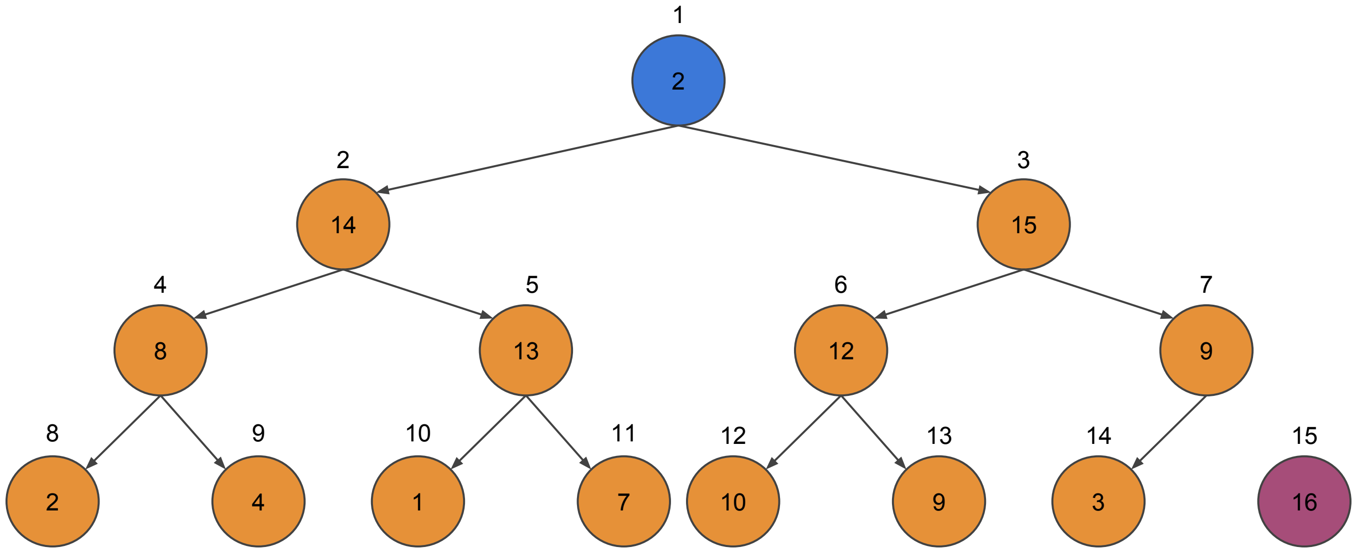
16	14	15	8	13	12	9	2	4	1	7	10	9	3	2

Simulation of Extract Max Continued



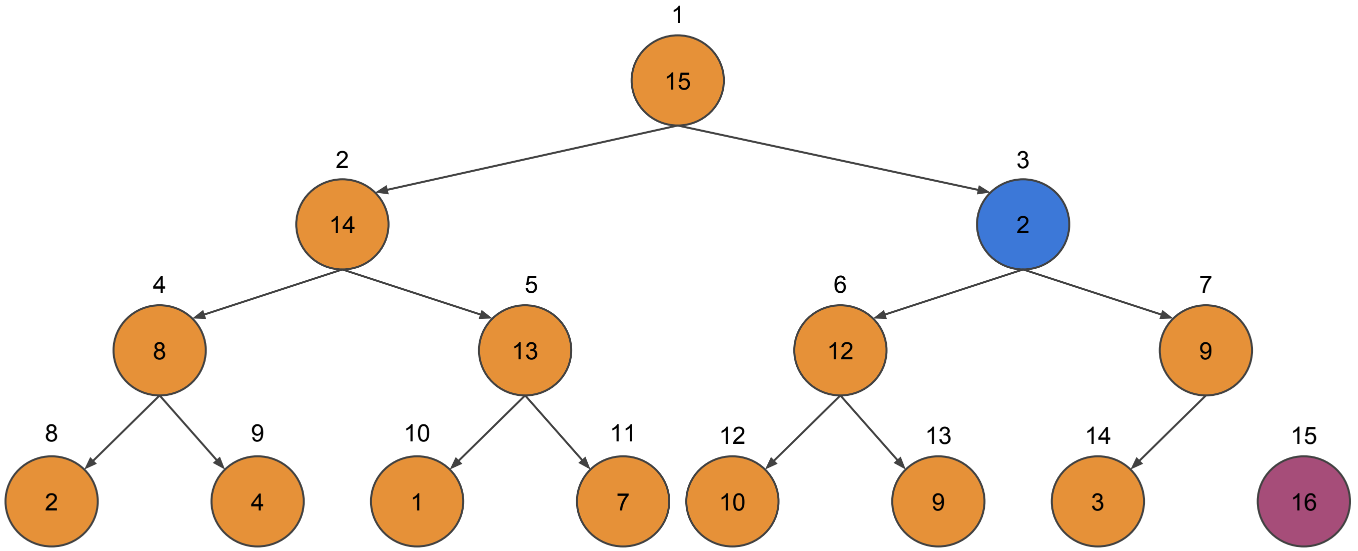
2	14	15	8	13	12	9	2	4	1	7	10	9	3	16

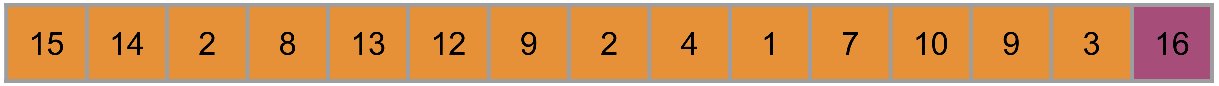
Simulation of Extract Max Continued



2	14	15	8	13	12	9	2	4	1	7	10	9	3	16

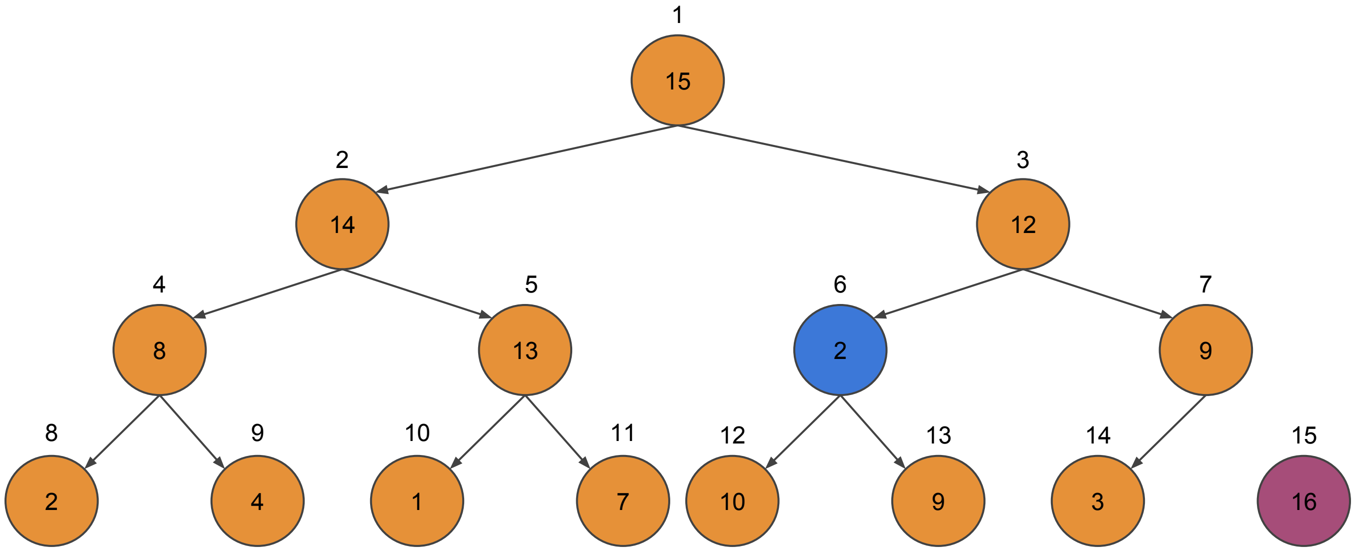
Simulation of Extract Max Continued





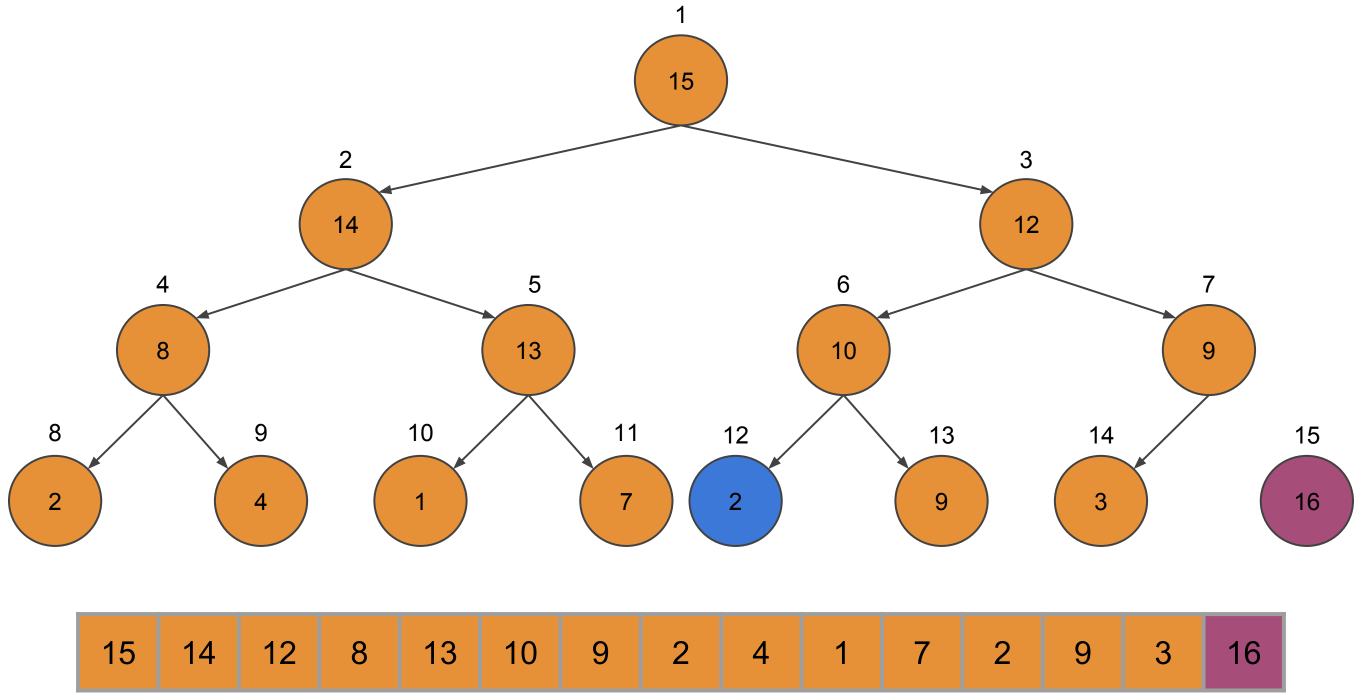
15	14	2	8	13	12	9	2	4	1	7	10	9	3	16

Simulation of Extract Max Continued

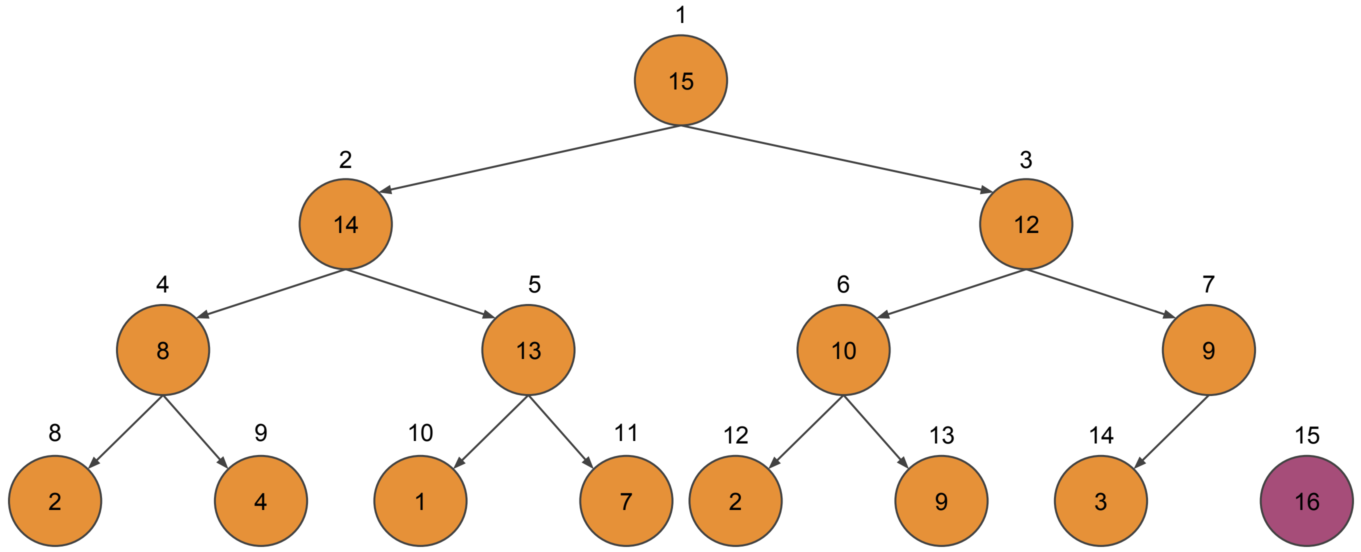


15	14	12	8	13	2	9	2	4	1	7	10	9	3	16

Simulation of Extract Max Continued



Simulation of Extract Max Continued

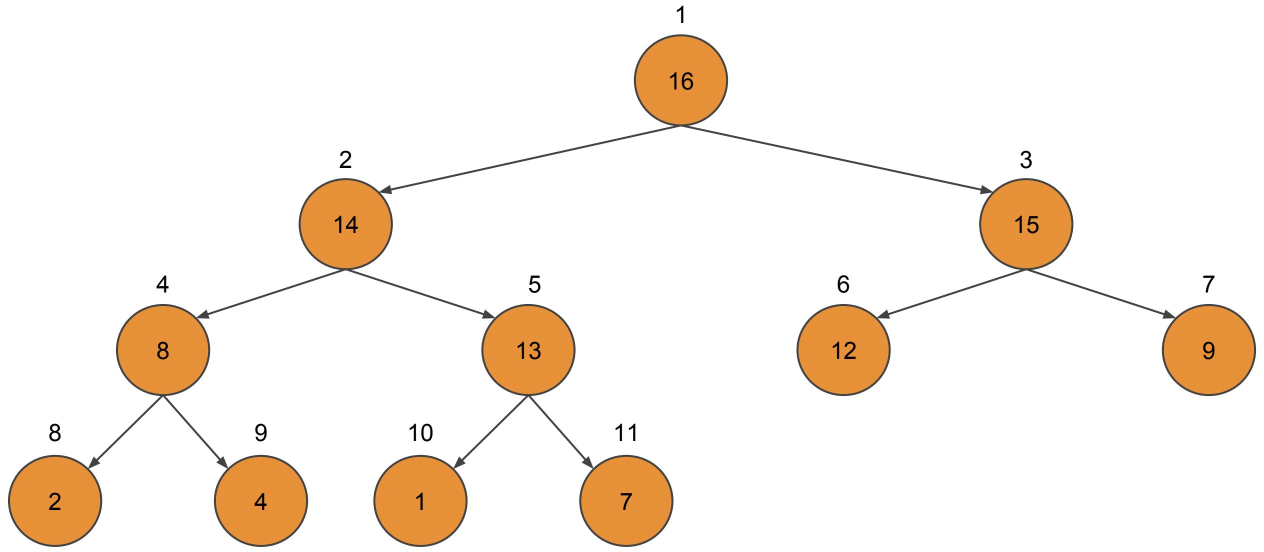


15	14	12	8	13	10	9	2	4	1	7	2	9	3	16

Increase Key

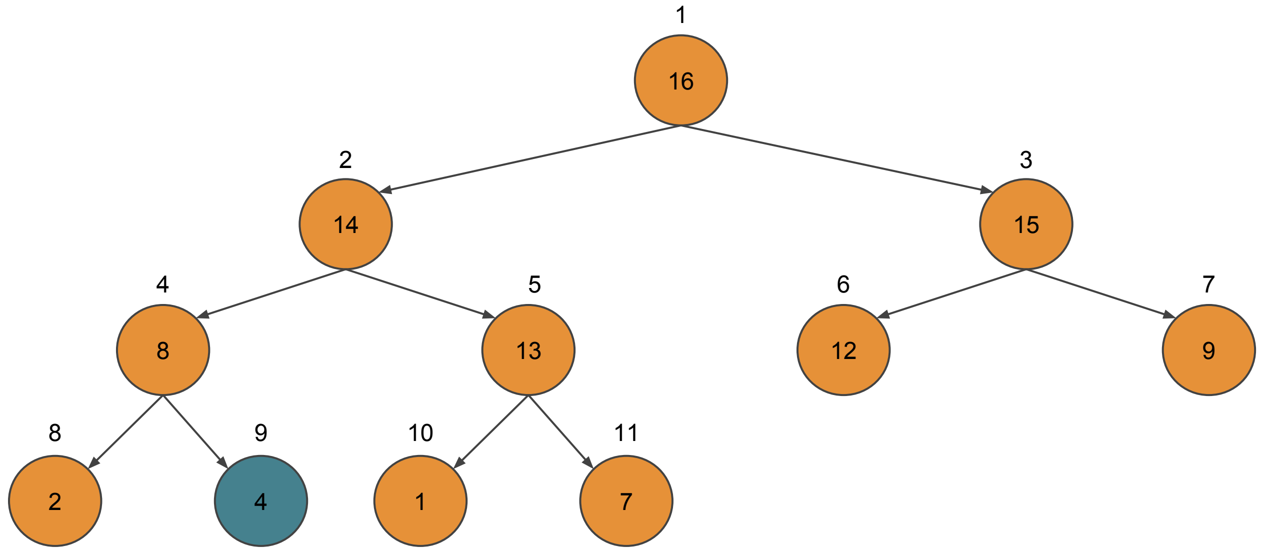
```
Increase_Key(S, n, x)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i>1 and A[Parent(i)] < A[i]
        exchange A[i] with A[Parent(i)]
        i = Parent(i)
```

Simulation of Increase Key



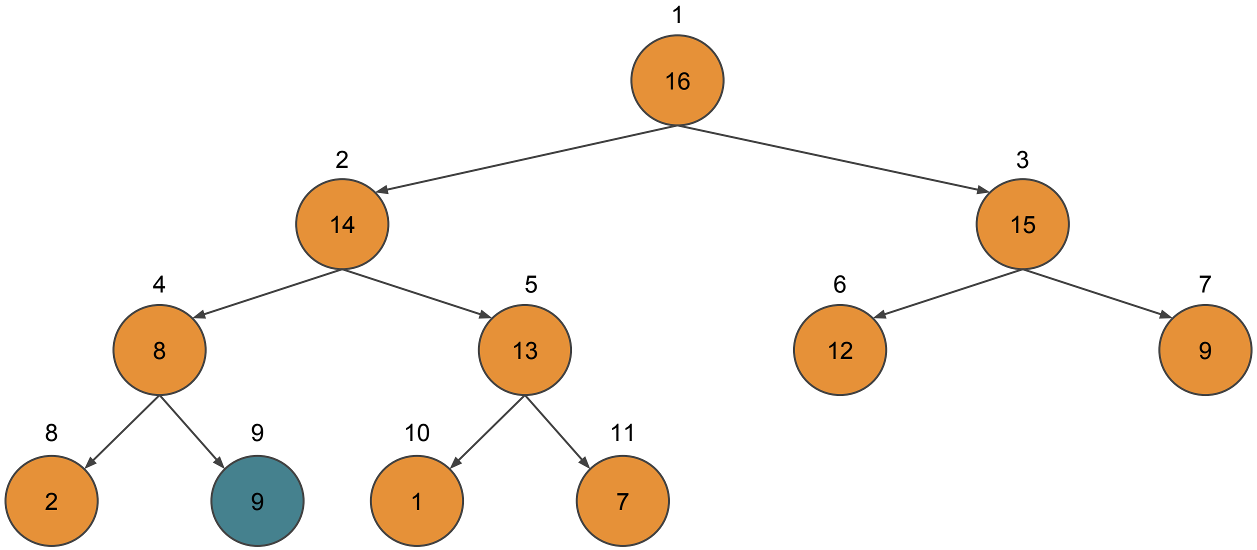
16	14	15	8	13	12	9	2	4	1	7

Simulation of Increase Key Continued



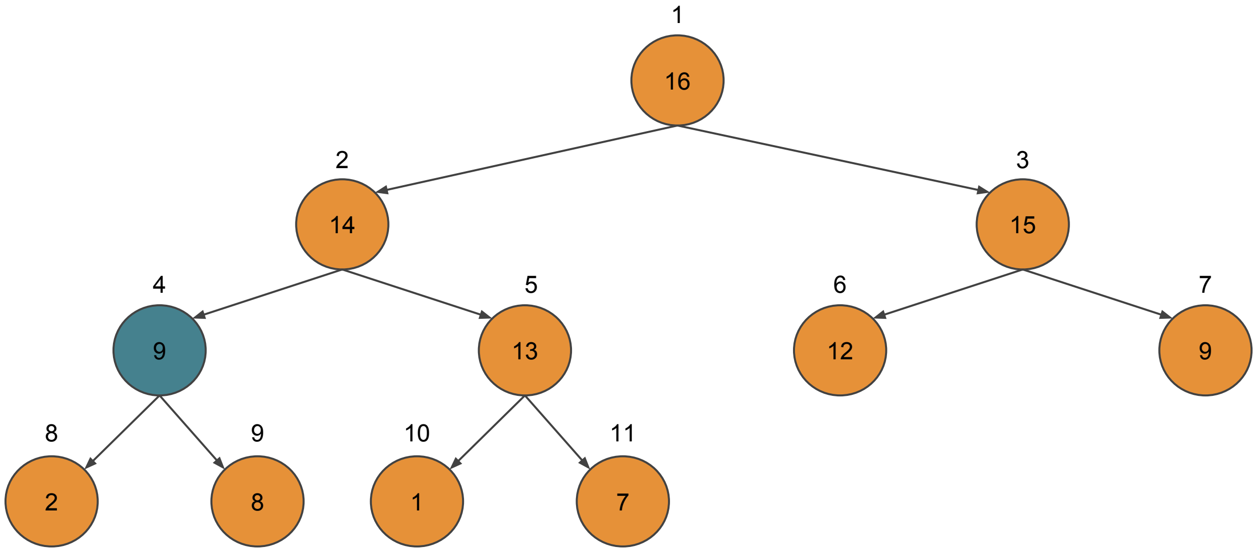
16	14	15	8	13	12	9	2	4	1	7

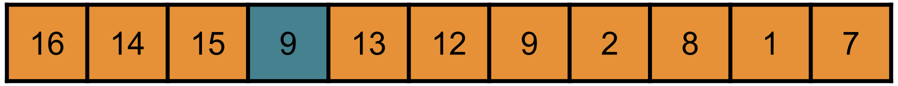
Simulation of Increase Key Continued



16	14	15	8	13	12	9	2	9	1	7

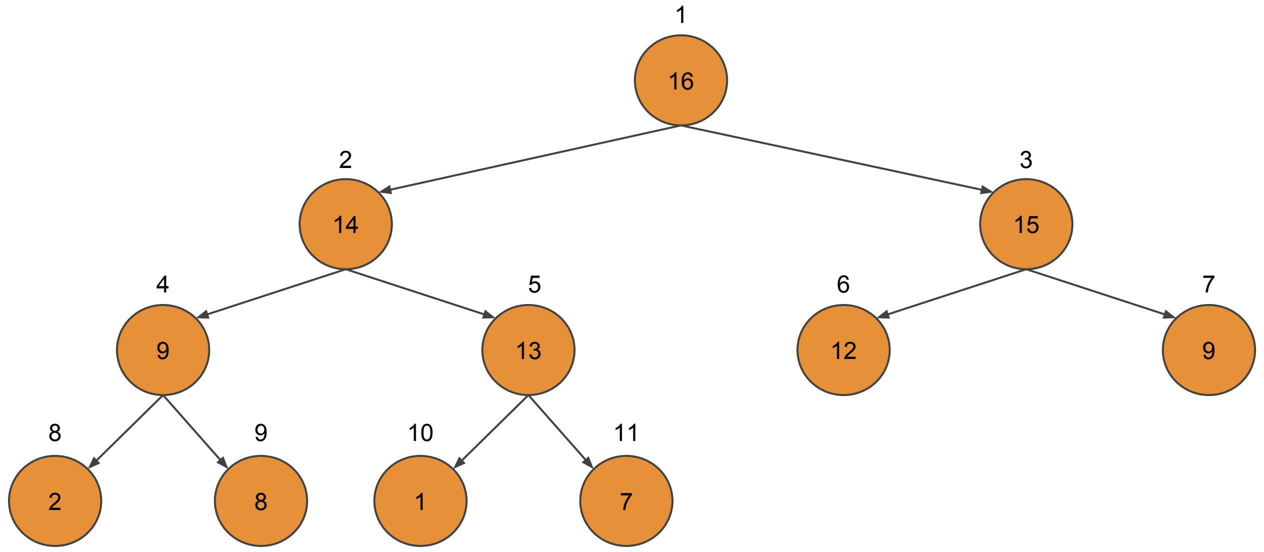
Simulation of Increase Key Continued





16	14	15	9	13	12	9	2	8	1	7

Simulation of Increase Key Continued

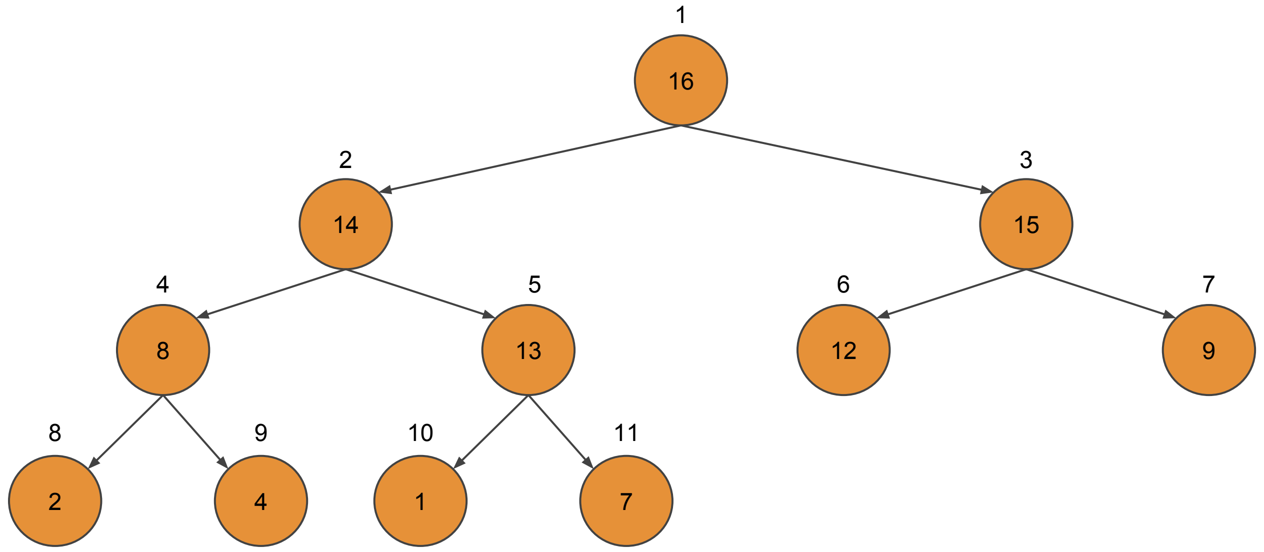


16	14	15	9	13	12	9	2	8	1	7

Insert

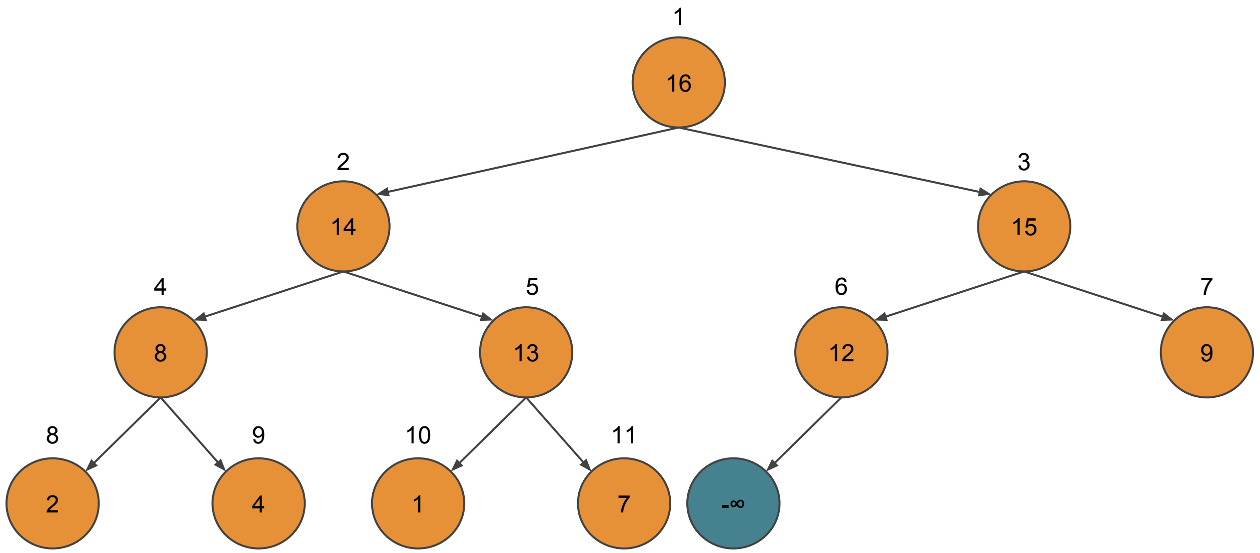
```
Insert(S, key)
    A.heap-size = A.heap-size + 1
    A[A:heap-size] = -∞
    Increase_Key(A, A.heap-size, key)
```

Simulation of Insert



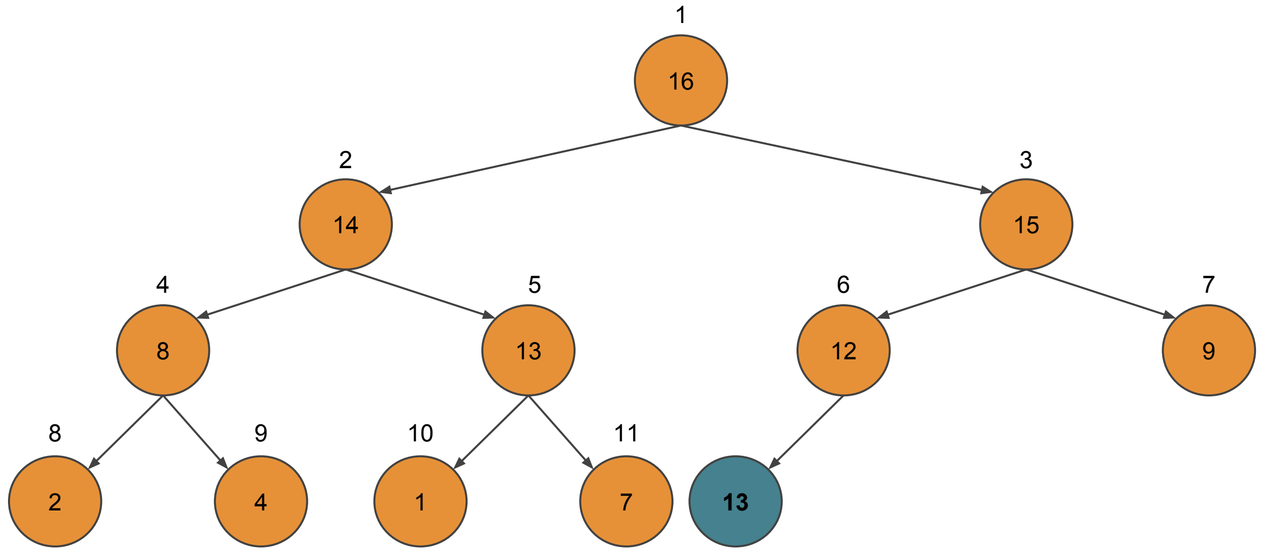
16	14	15	8	13	12	9	2	4	1	7	

Simulation of Insert Continued



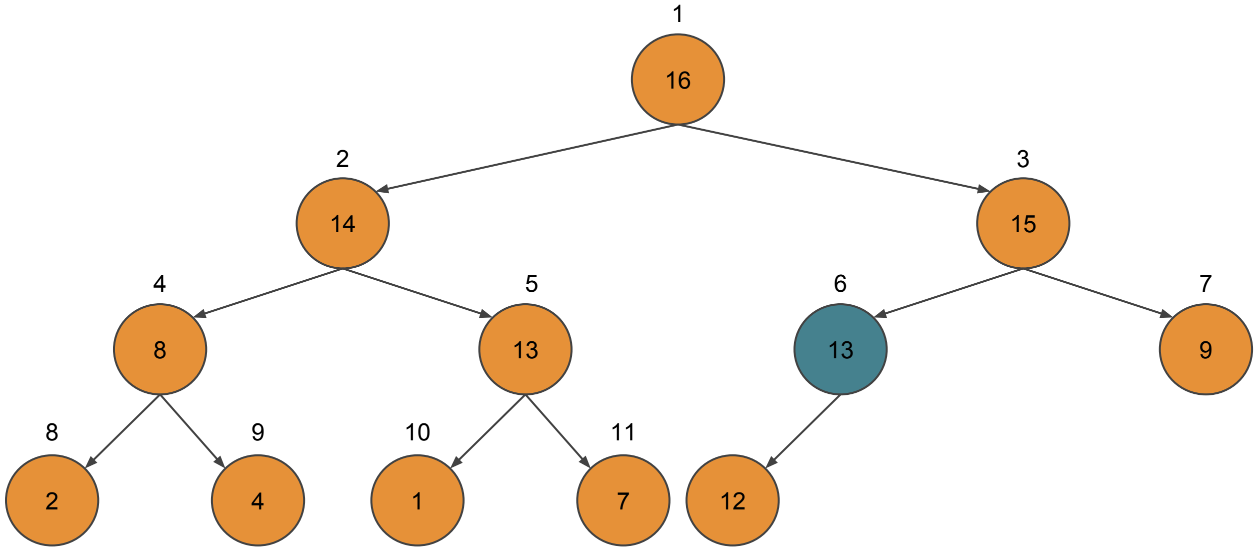
16	14	15	8	13	12	9	2	4	1	7	-∞

Simulation of Insert Continued



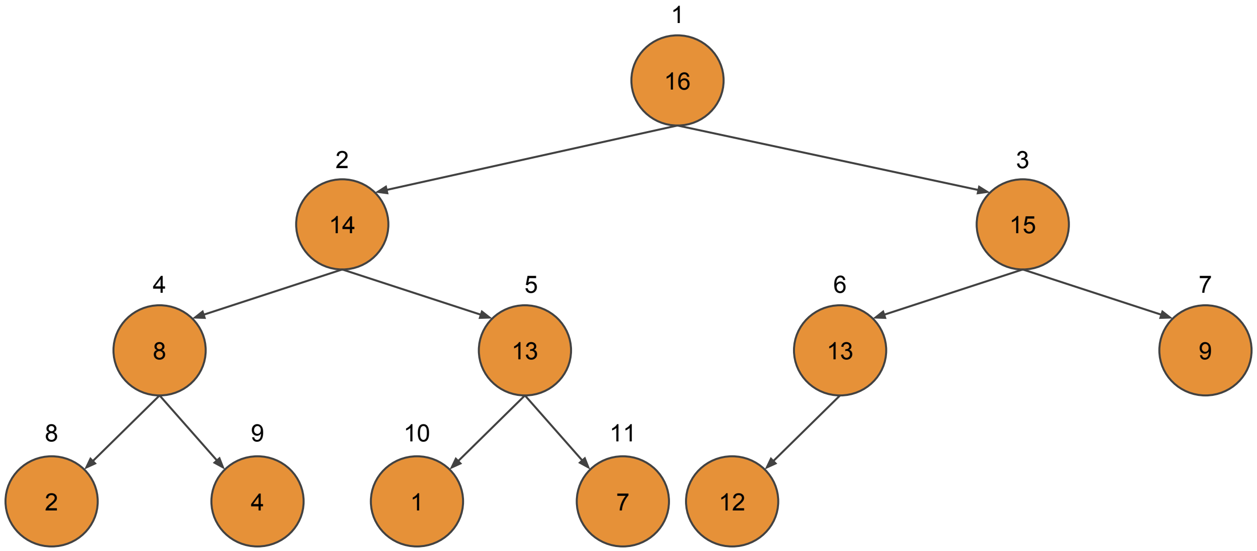
16	14	15	8	13	12	9	2	4	1	7	13

Simulation of Insert Continued



16	14	15	8	13	13	9	2	4	1	7	12

Simulation of Insert Continued



16	14	15	8	13	13	9	2	4	1	7	12