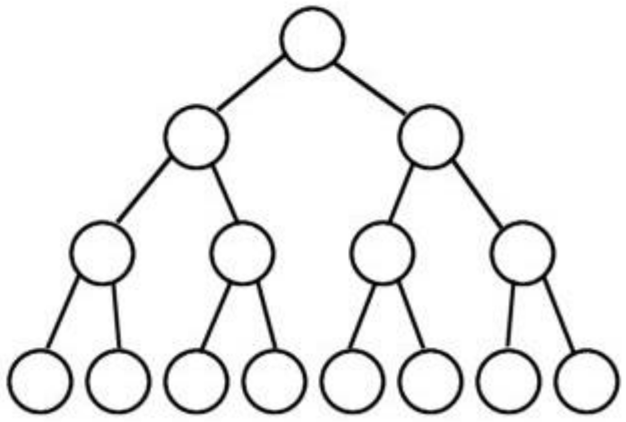
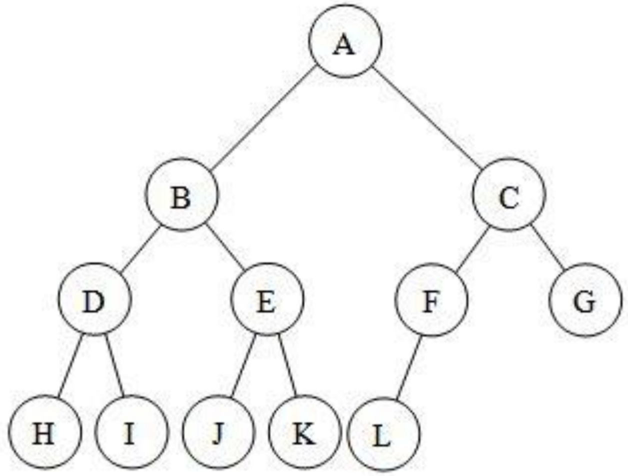
Heaps

Full v.s. Complete Binary Trees





A **full binary tree** is a binary tree in which each node has exactly 0 or 2 children.

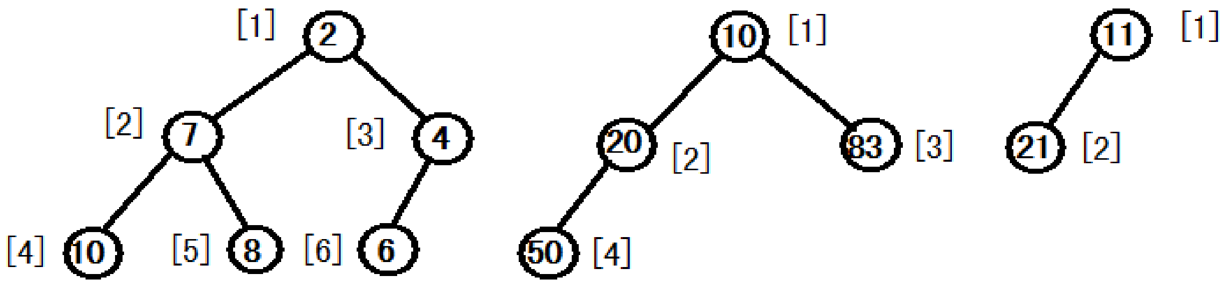
A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

Binary Heap: Definition

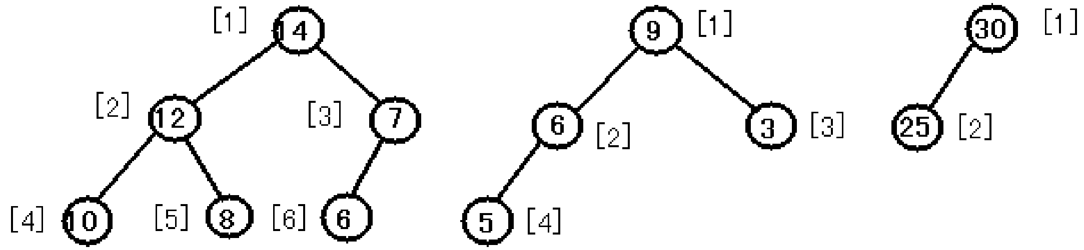
(1) A **complete binary tree** filled on all levels, except last, which is filled from left to right. Not a binary search tree, but the keys do follow some order.

(2) **Heap property**

(a) Min-heap: every child greater than (or equal to) parent (A[Parent(i)] <=A[i])

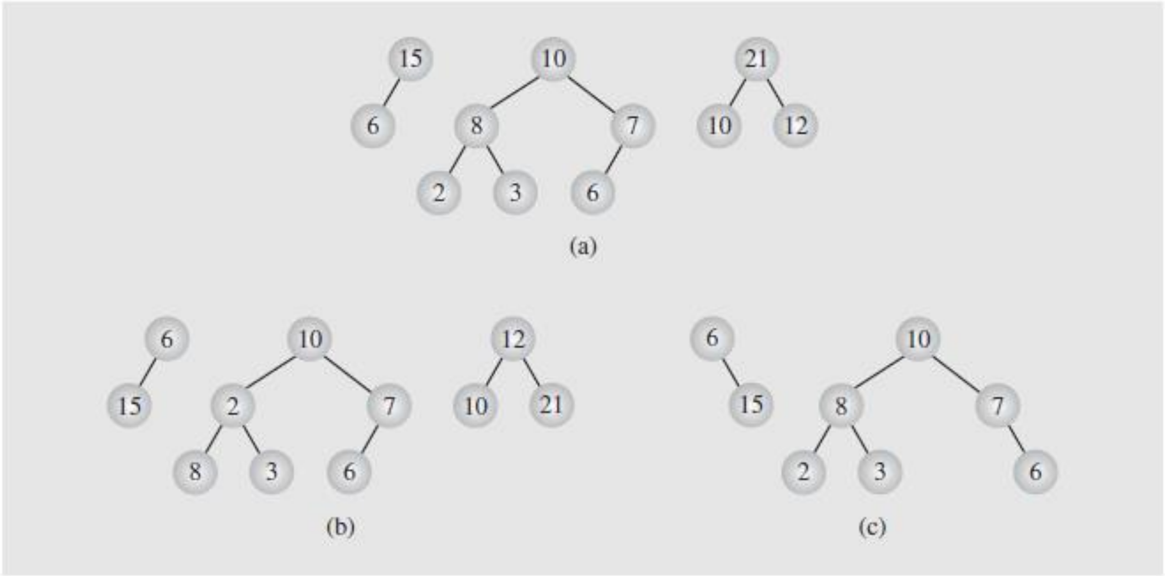


(b) Max-heap: every child smaller than (or equal to) parent (A[Parent(i)] >=A[i])

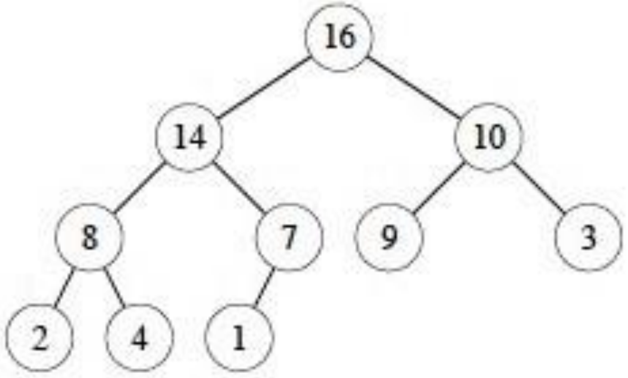


Binary Heap

Examples of (a) heaps and (b–c) nonheaps.



- Max or min element is in root (e.g., b1 is not max-heap)
- Heap with N elements has height = $\lfloor \log_2 N \rfloor$.



N = 10
Height = 3

Examples of priority queue

- **Printer**
  - **Important document → max-heap**
  - **Short documents → min-heap (e.g., 1 page vs 100 page)**

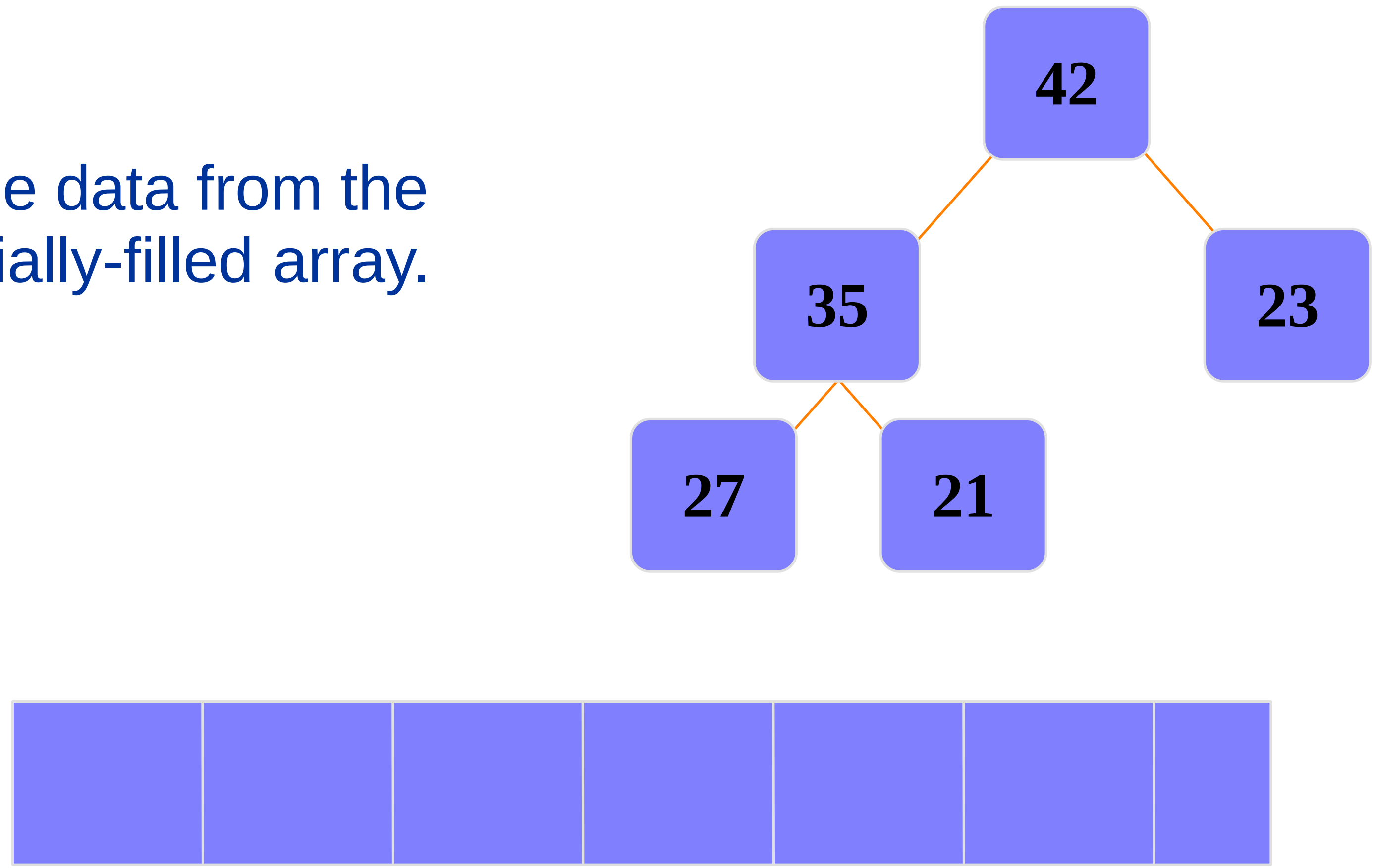
Heap Implementation

Several ways to implement heaps/priority queues

- **Linked list:** fast insertion at the front but delete could be slow.
- **Sorted list:** fast deletion but insertion is expensive.
- **Binary search tree:** O(log N) for insert and delete operations
- **Array or vector:** No nodes, no pointers and no links.
  - The problem with the array implementation is that an estimate of the maximum heap size is required in advance, but typically this is not a problem (and we can **resize** if needed).
  - We shall draw the heaps as **trees**, with the implication that an actual implementation will use simple arrays.

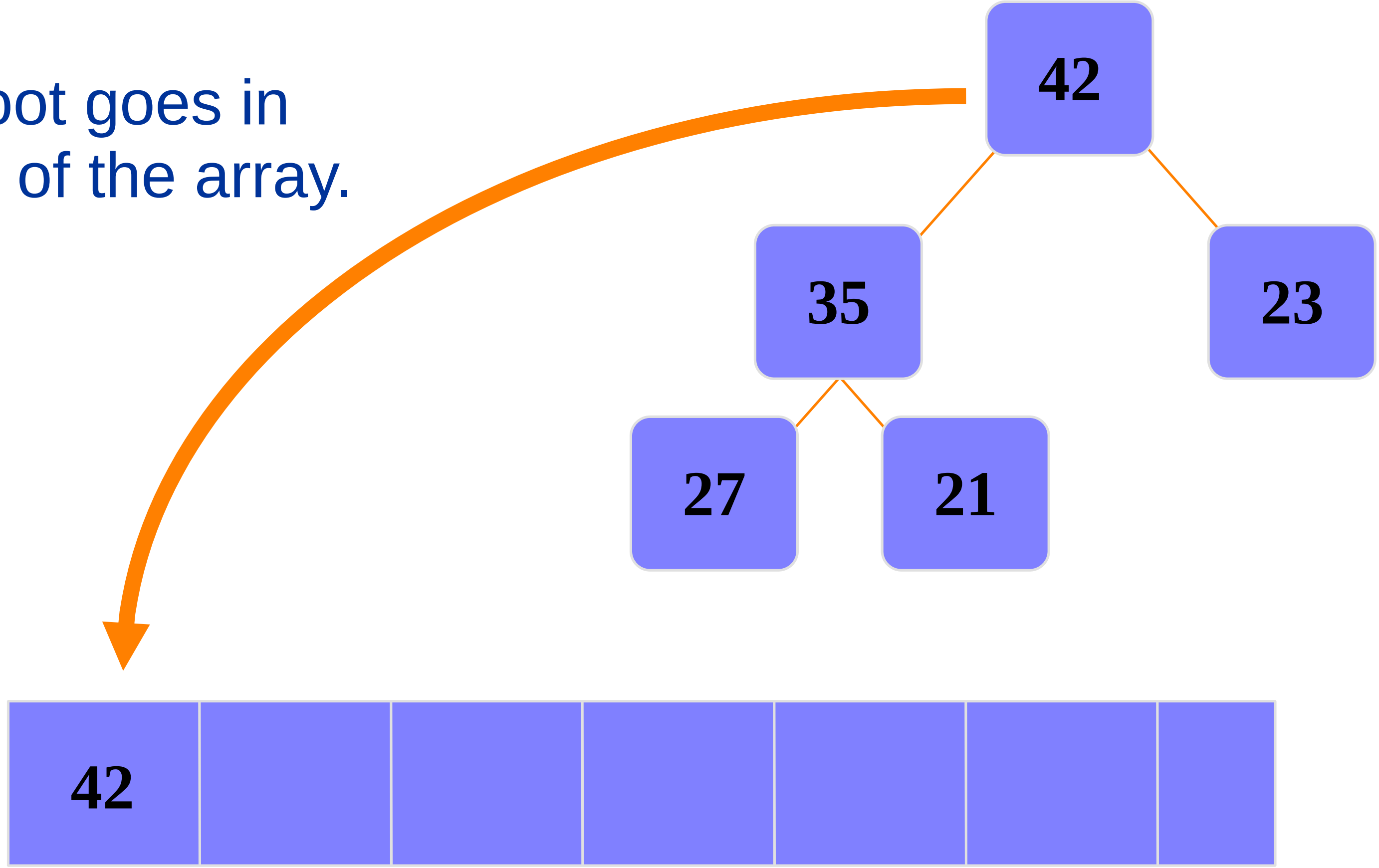
Implementing a Heap

We will store the data from the nodes in a partially-filled array.



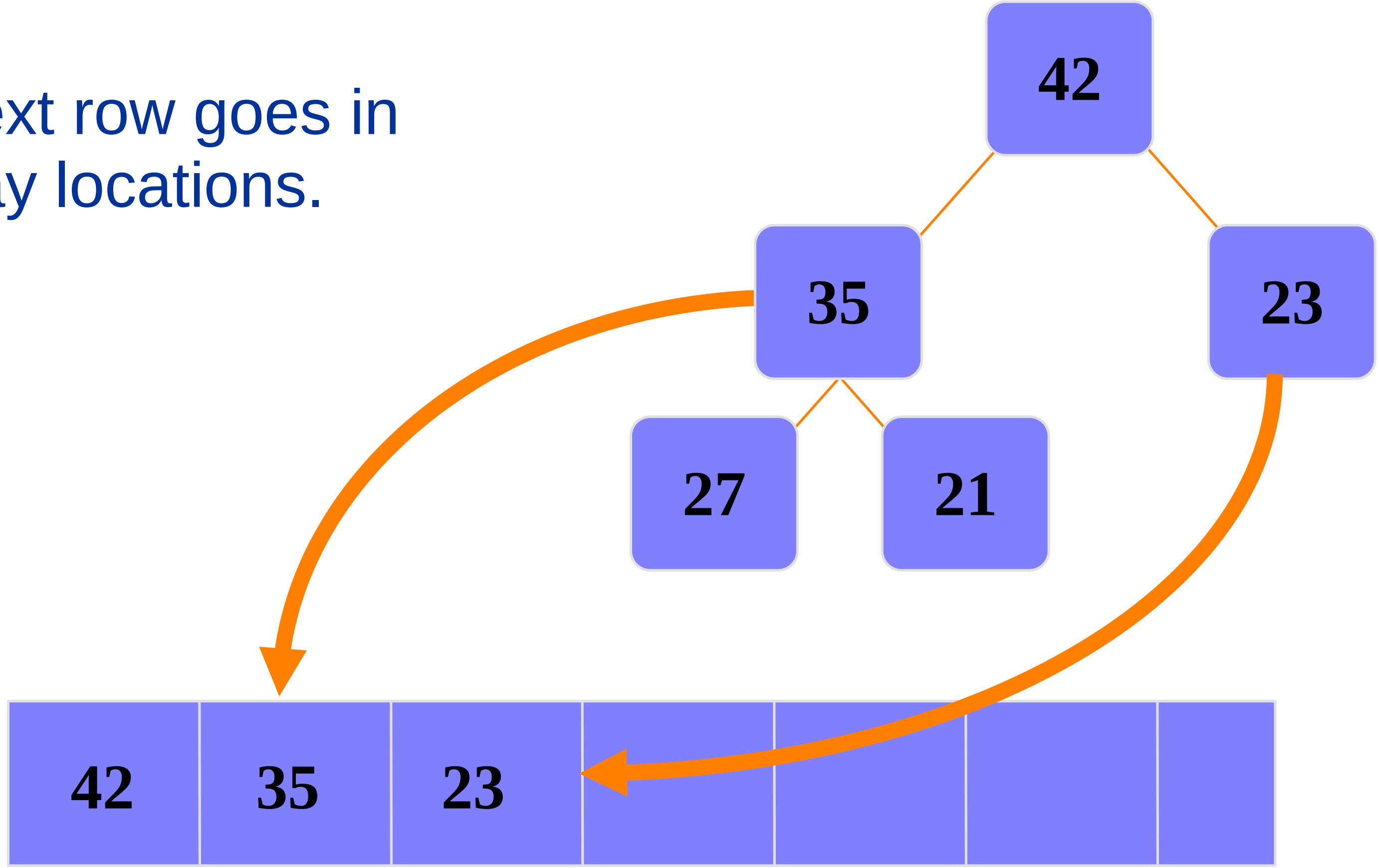
Implementing a Heap

Data from the root goes in the first location of the array.



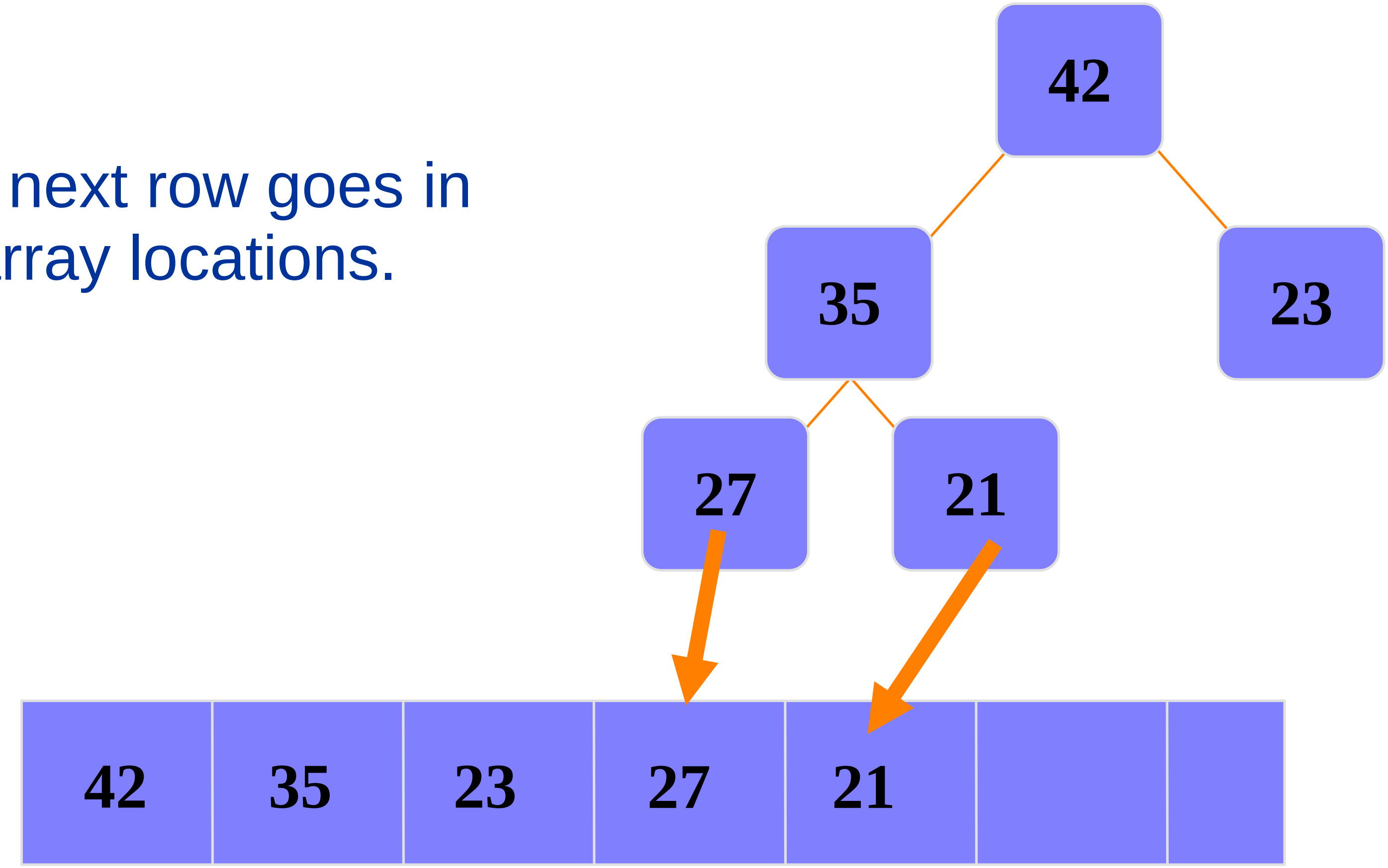
Implementing a Heap

Data from the next row goes in the next two array locations.



Implementing a Heap

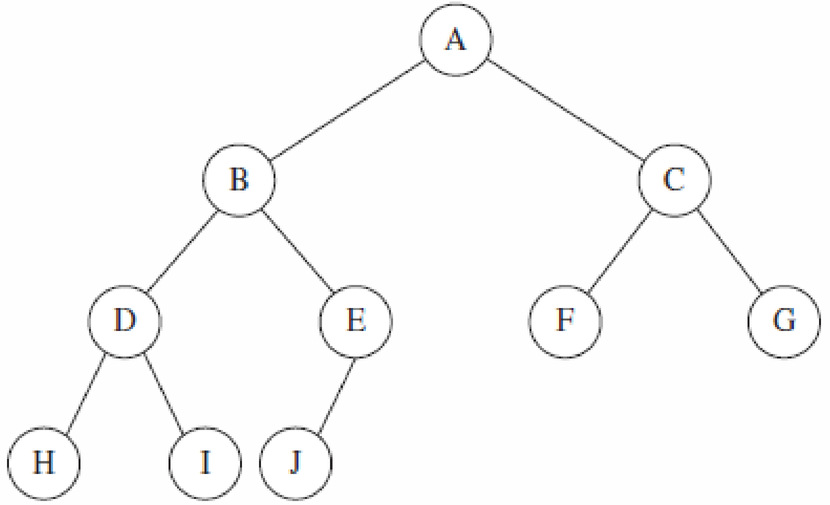
Data from the next row goes in the next two array locations.



Array Implementation

For any element in array position i

- **If index starts from 1:**
  - **Left (i)** = the left child is in position = 2i
  - **Right (i)=** the right child is in position = 2i + 1
  - **Parent(i)** = the parent is in position = $\lfloor i/2 \rfloor$
- **If index starts from 0:**
  - Left (i) = 2i +1
  - Right (i)= 2i + 2
  - Parent(i) = $\lfloor (i-1)/2 \rfloor$



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

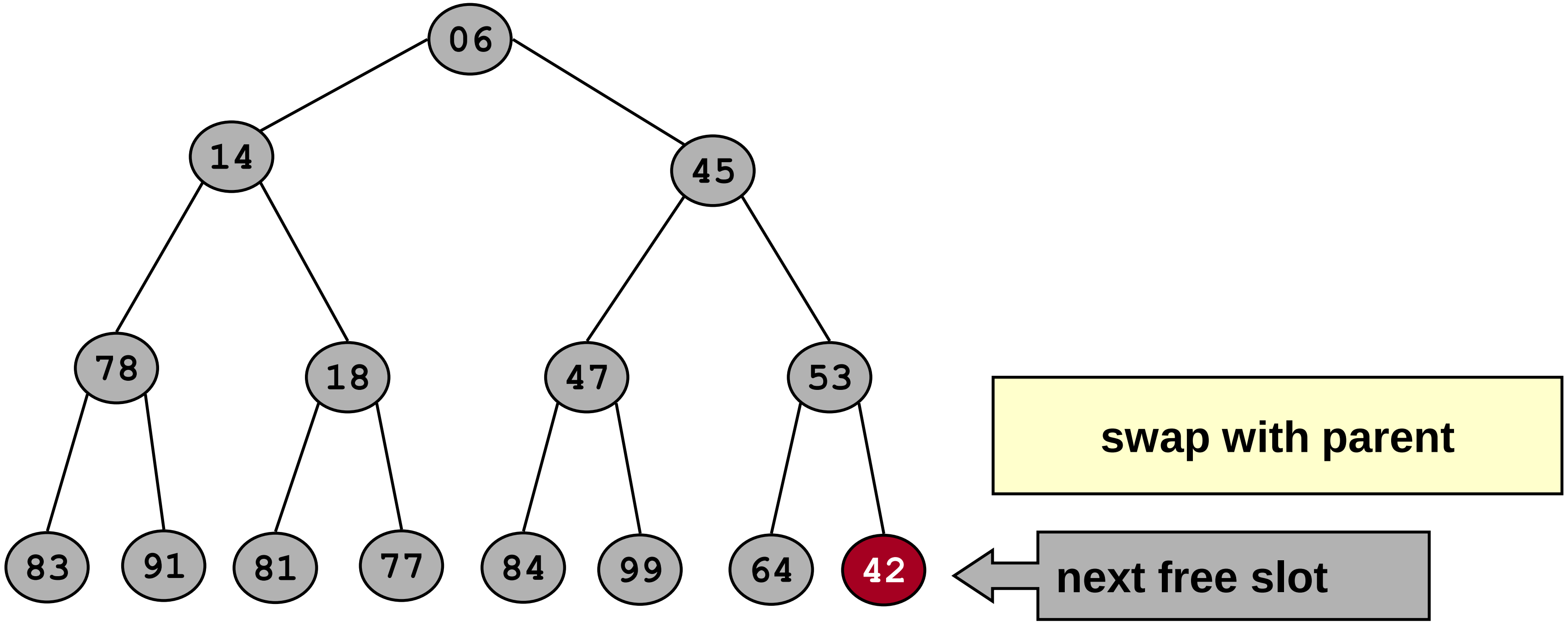
Array implementation of complete binary tree

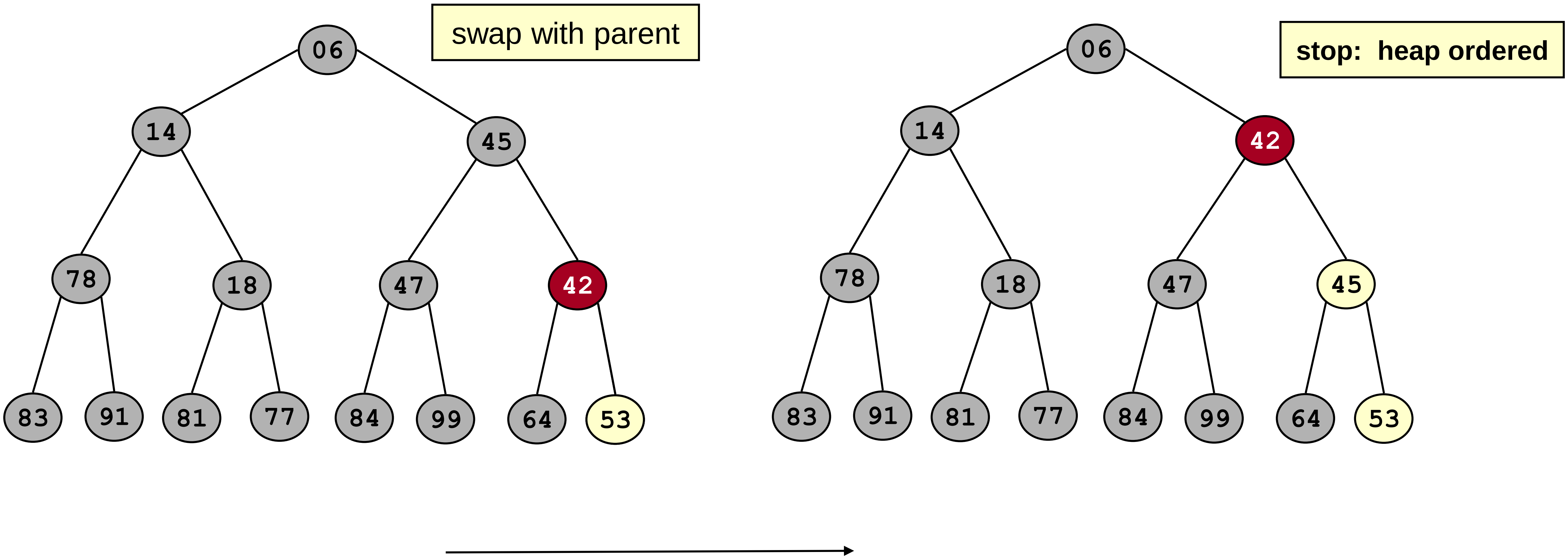
Binary Heap: Insertion

(1) Insert into next available slot.

(2) Bubble up until it's heap ordered.

- The process of pushing the new node upward is called **reheapification upward.**

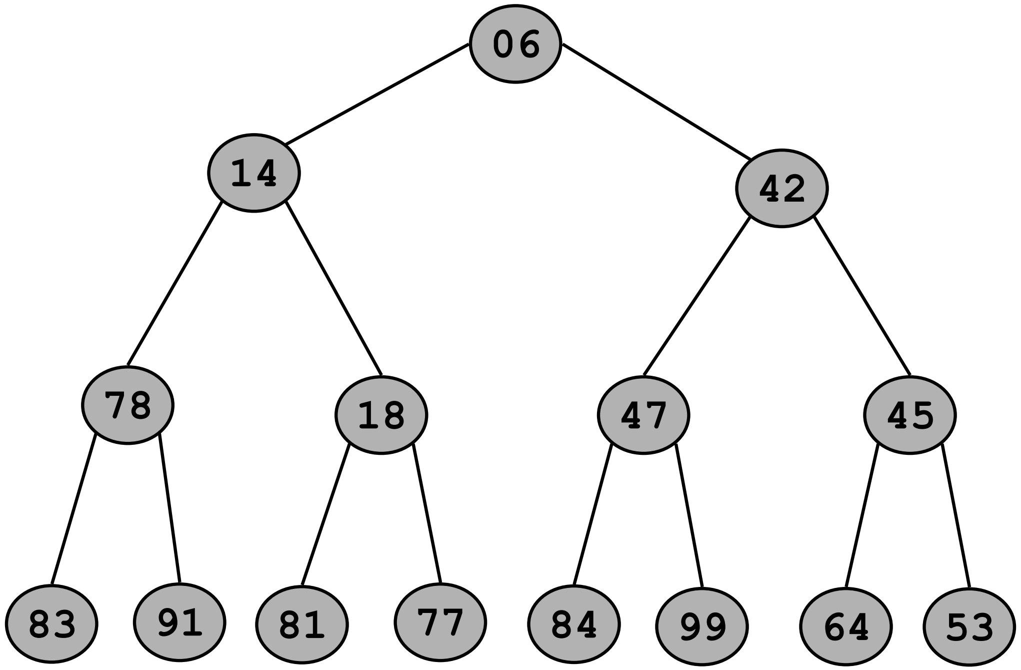



swap with parent
06
14
45
78
18
47
42
83
91
81
77
84
99
64
53
stop: heap ordered
06
14
42
78
18
47
45
83
91
81
77
84
99
64
53

Binary Heap: Decrease Key

Decrease key of element x to k.

- **Bubble up until it's heap ordered.**

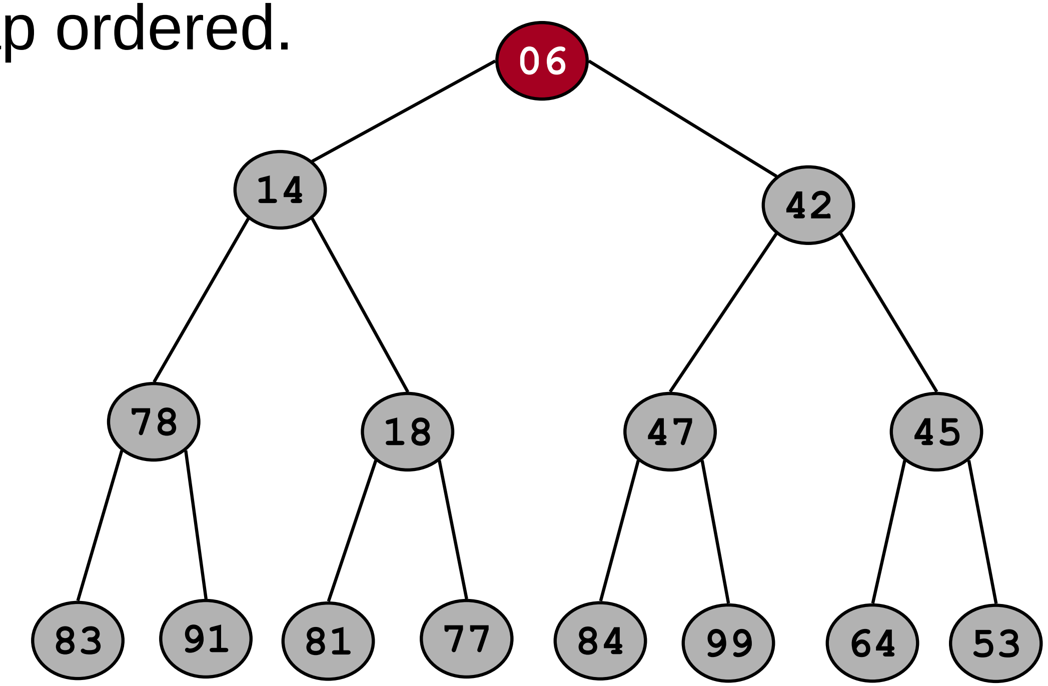


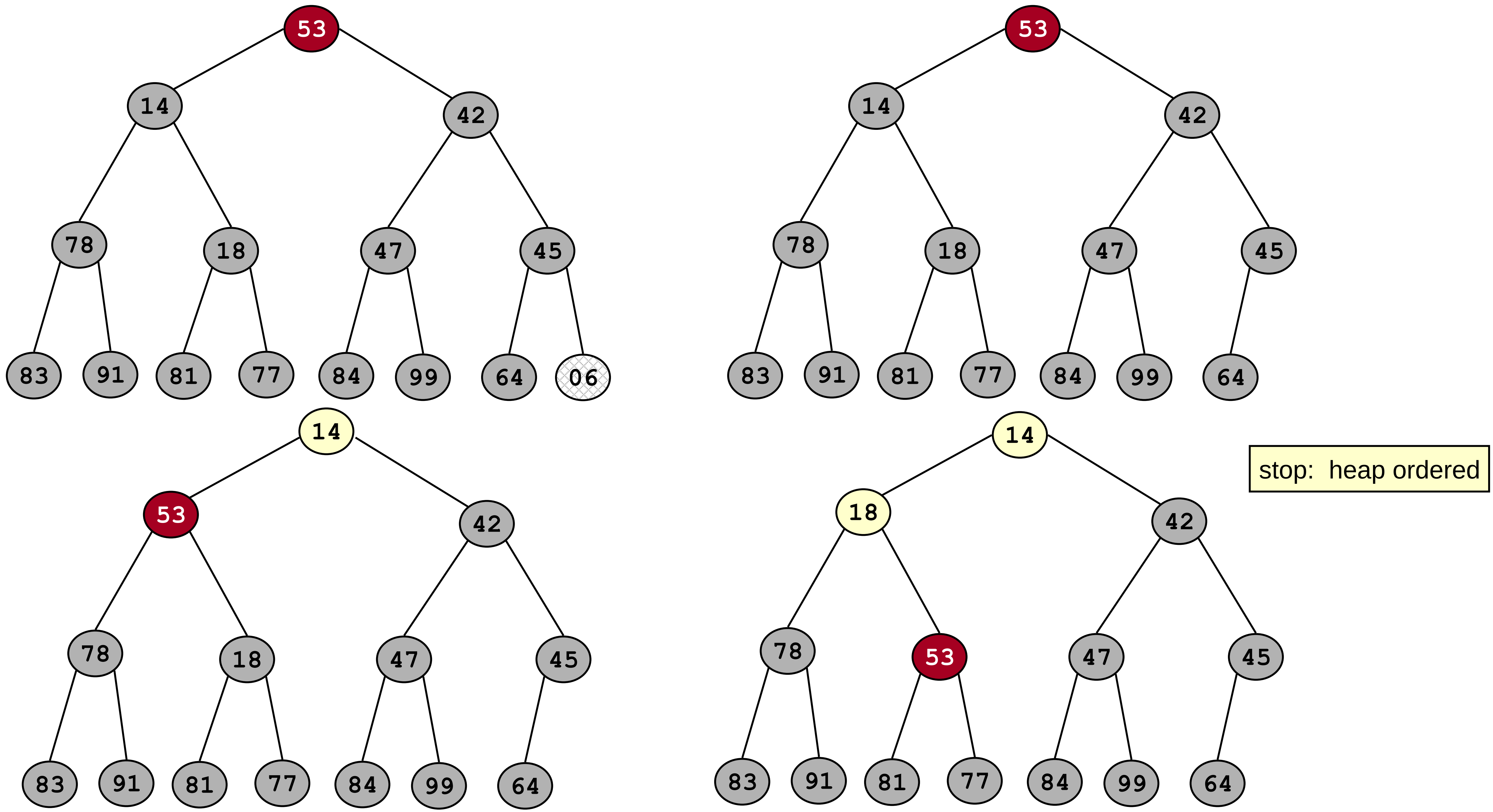
Binary Heap: Delete root (min/max)

(1) **Exchange root** with **rightmost leaf.**

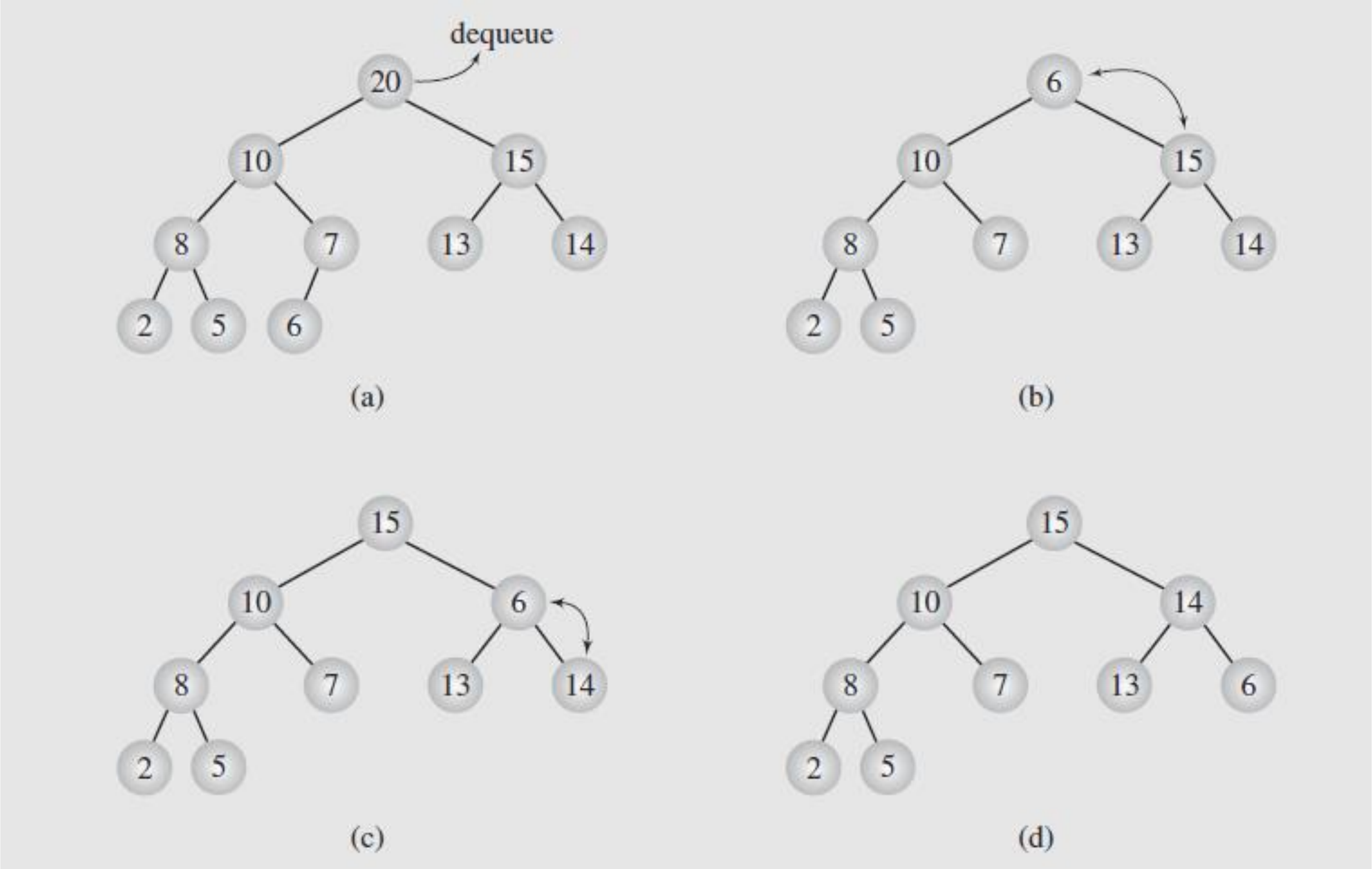
(2) **Delete** the rightmost leaf.

(3) **Bubble** root **down** until it's heap ordered.

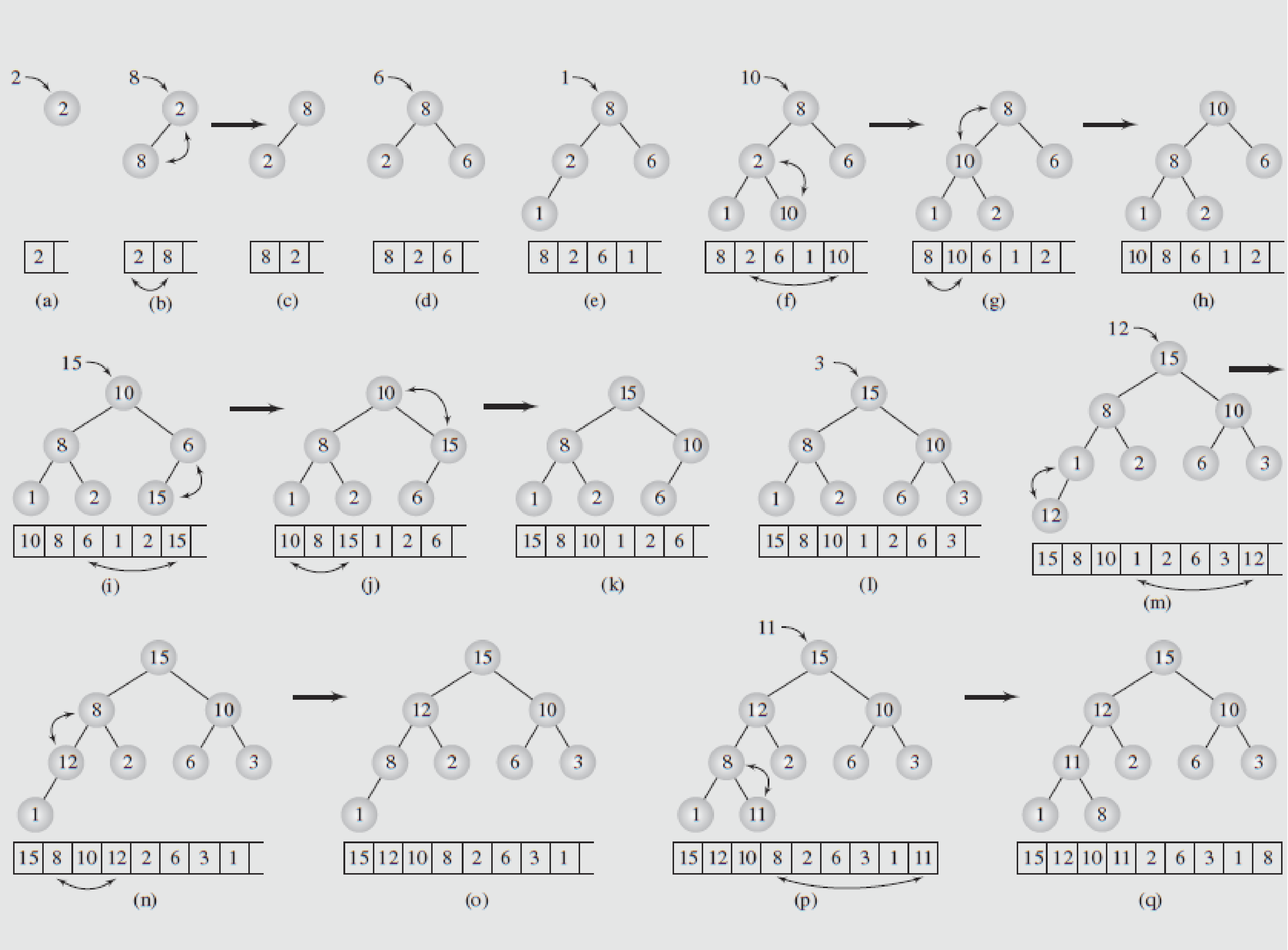



53
14
42
78
18
47
45
83
91
81
77
84
99
64
06
stop: heap ordered

Example: delete in a max-heap



Example: Insertion to max-heap 2,8,6,1,10,15,3,12,11



Heapify

Max-Heapify or Heapify Down maintains heap property by “floating” a value down the heap that starts at i until it is in the correct position. If the value bubble up then it is called Min-Heapify or Heapify Up. A = Array, n = heapsize.

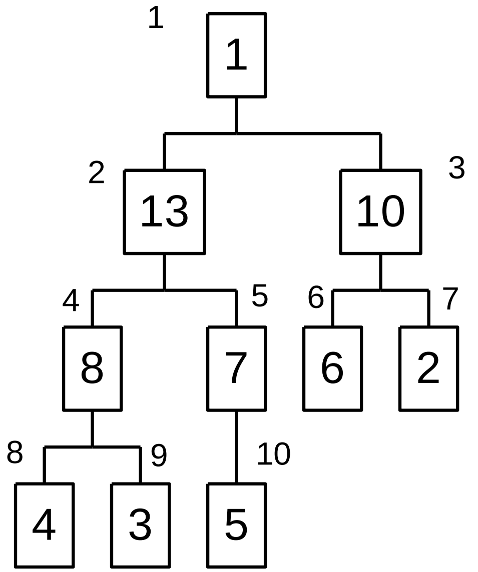
```
Max-Heapify(A,i,n)
L = Left(i)
R = Right(i)
if L <=n  and A[L] > A[i]
     largest = L
else largest = i
if R <= n and A[R] > A[largest]
     largest = R
if largest ≠ i
   exchange A[i] ↔ A[largest]
   Max-Heapify(A,largest,n)
```

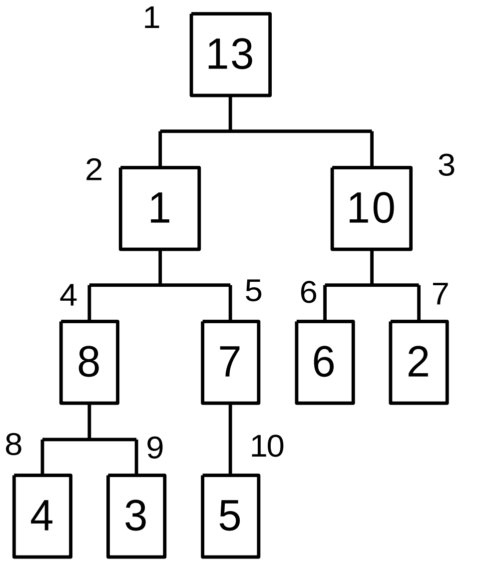
Find the largest node between the current node and its children

Max-Heapify Example

Heapify(A,1,10)

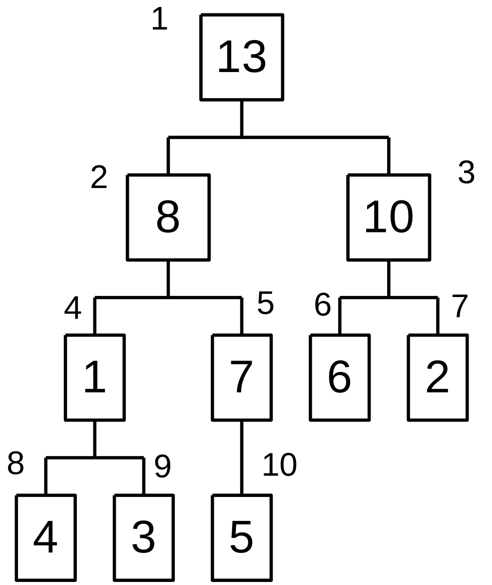
A=[1 13 10 8 7 6 2 4 3 5]





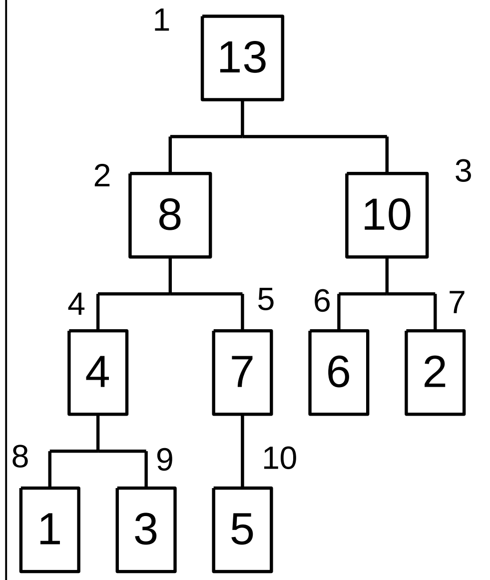
Next is Heapify(A,2,10)

A=[13 1 10 8 7 6 2 4 3 5]



Next is Heapify(A,4,10)

A=[13 8 10 1 7 6 2 4 3 5]



Next is Heapify(A,8,10)

A=[13 8 10 4 7 6 2 1 3 5]

On this iteration we have reached a leaf and are finished.

Building the Heap

- Given an array A, we want to build this array into a heap.
- **Note**: Leaves are already a heap!

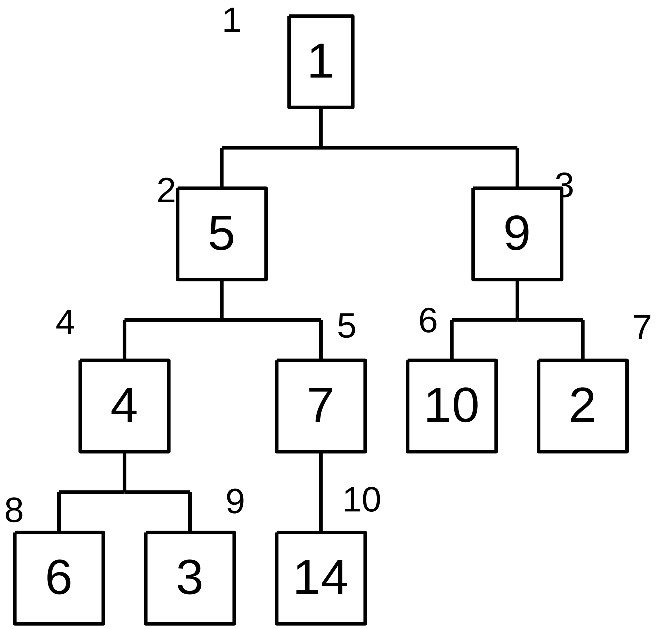
```
Build-Max-Heap(A,n)
   for i = n downto 1        // could we start at n/2?
        Max-Heapify(A,i,n)
```

- Start with the leaves (last ½ of A) and consider each leaf as a 1 element heap. Call heapify on the parents of the leaves, and continue recursively to call heapify, moving up the tree to the root.

Build-Heap Example

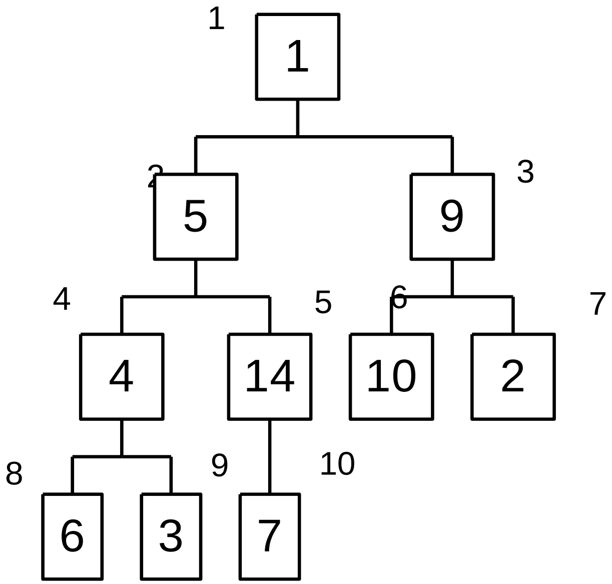
Build-Max-Heap(A,10)

A=[1 5 9 4 7 10 2 6 3 14]



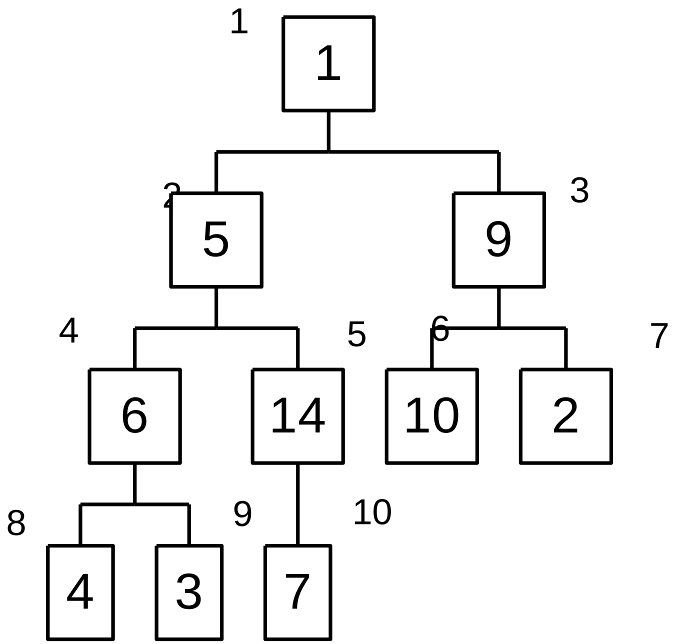
Max-Heapify(A,10,10) exits since this is a leaf.
Max-Heapify(A,9,10) exits since this is a leaf.
Max-Heapify(A,8,10) exits since this is a leaf.
Max-Heapify(A,7,10) exits since this is a leaf.
Max-Heapify(A,6,10) exits since this is a leaf.
Max-Heapify(A,5,10) puts the largest of A[5] and its children, A[10] into A[5]:

Build-Heap Example



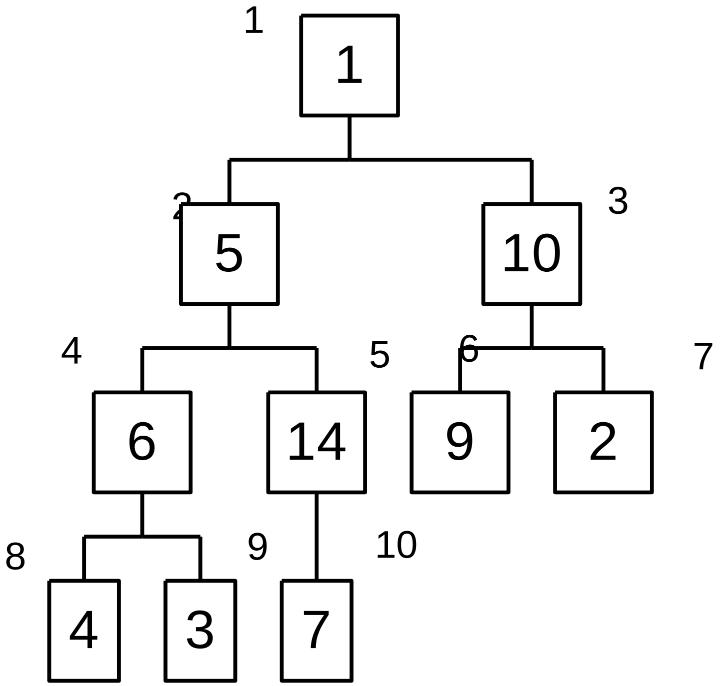
A=[1 5 9 4 14 10 2 6 3 7]
Max-Heapify(A,4,10)

Build-Heap Example



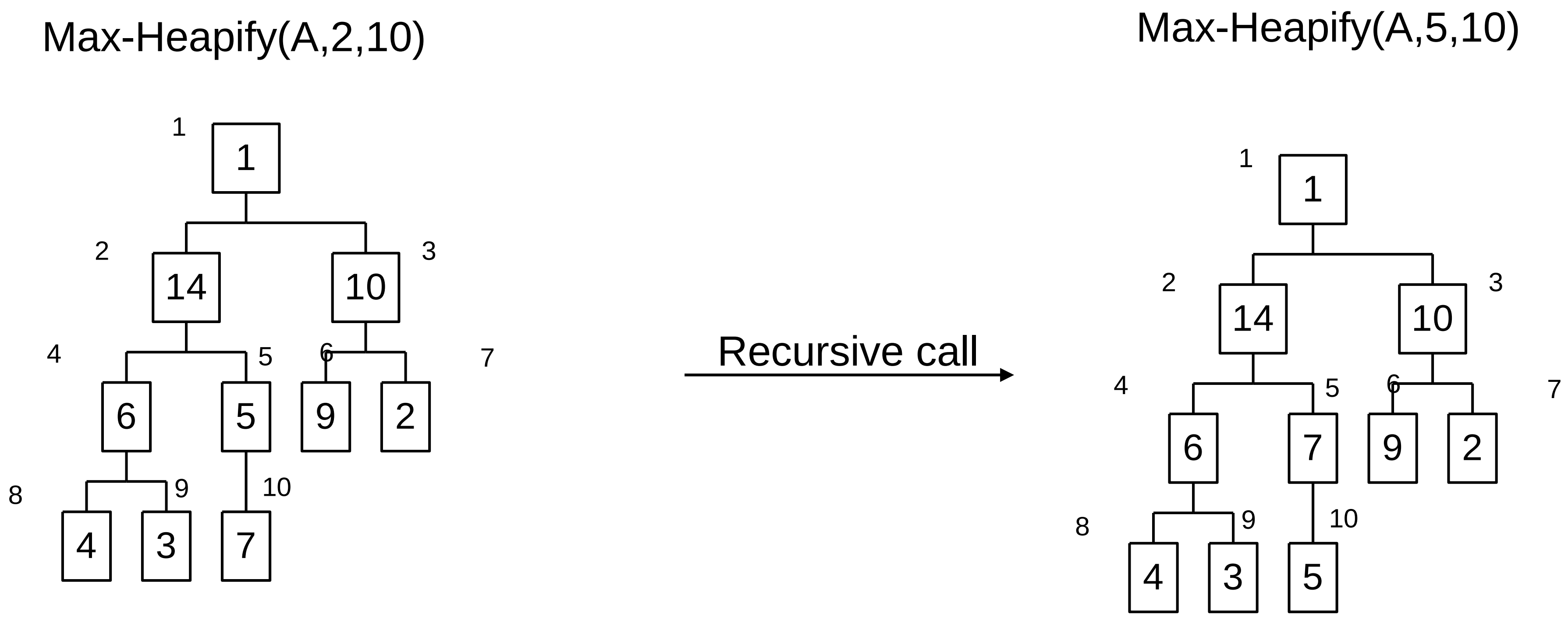
A=[1 5 9 6 14 10 2 4 3 7]
Max-Heapify(A,3,10):

Build-Heap Example



A=[1 5 10 6 14 9 2 4 3 7]
Max-Heapify(A,2,10):

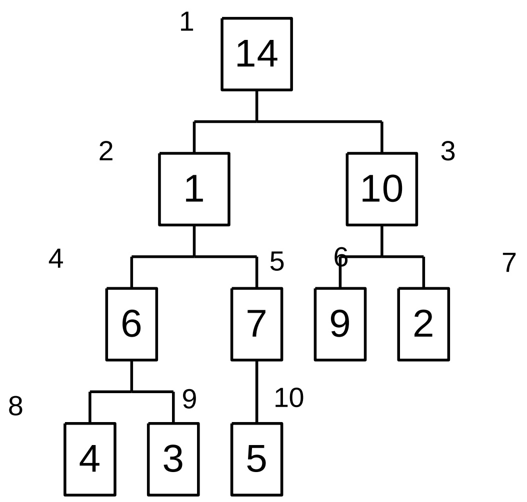
Build-Heap Example



A=[1 14 10 6 7 9 2 4 3 5]
Max-Heapify(A,1,10):

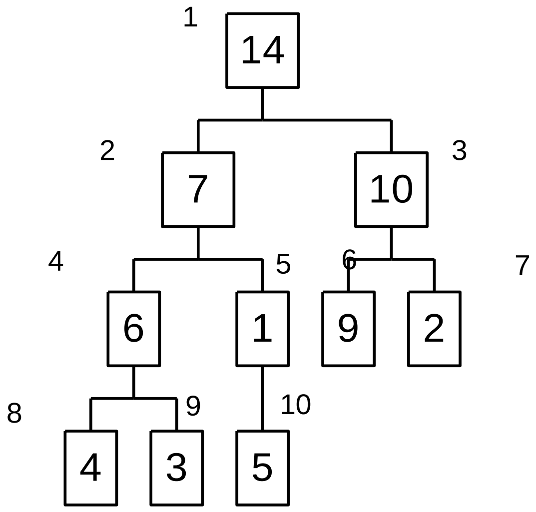
Build-Heap Example

Max-Heapify(A,1,10)



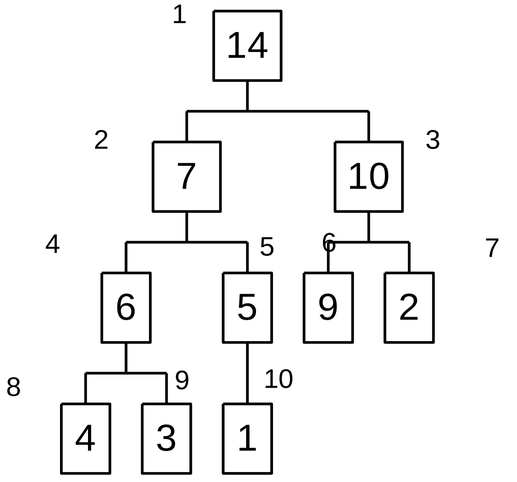
Recursive call

Max-Heapify(A,2,10)



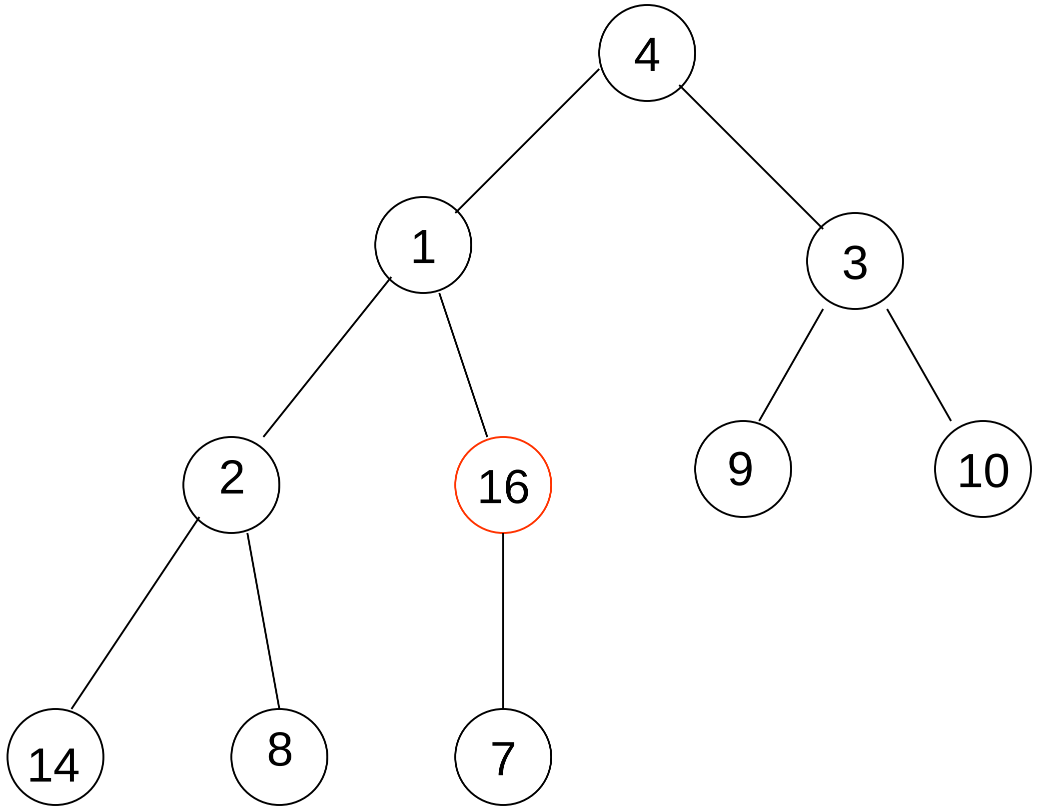
Recursive call

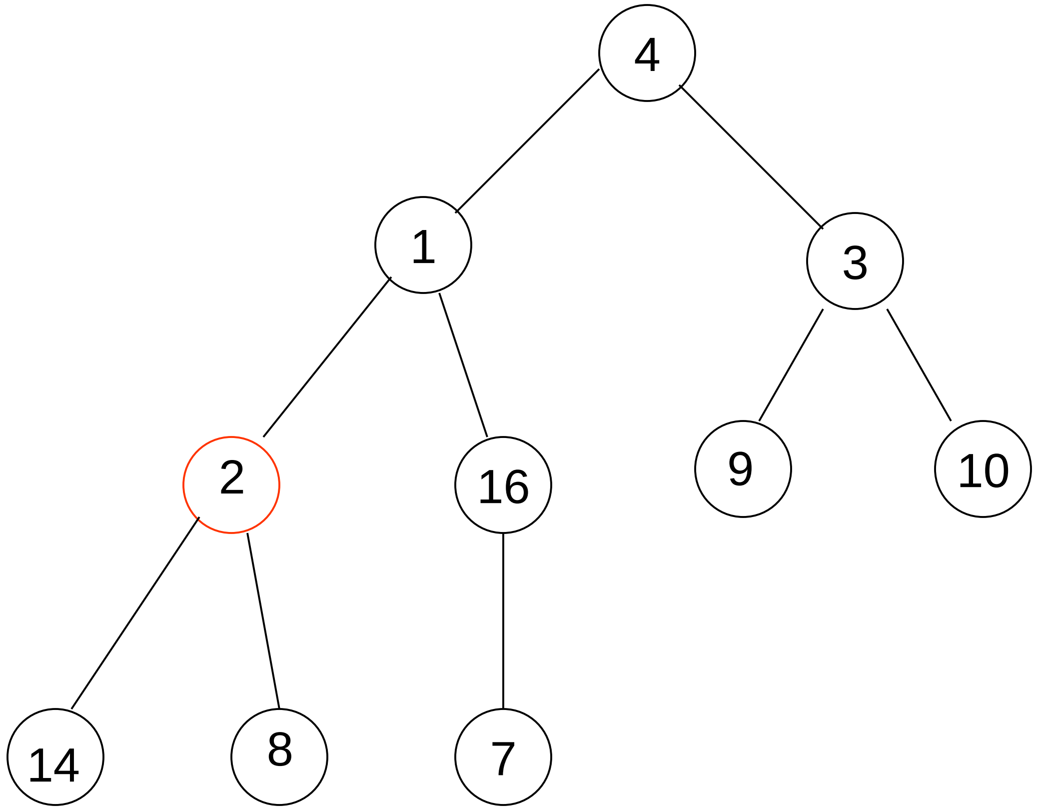
Max-Heapify(A,5,10)

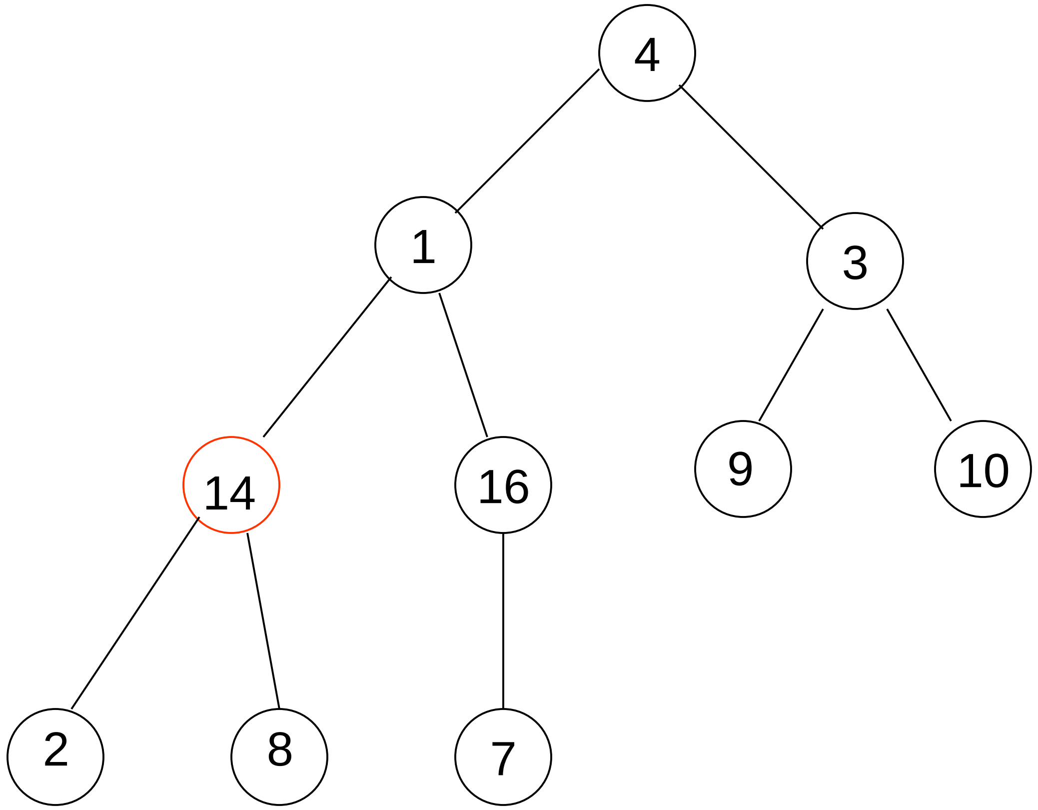


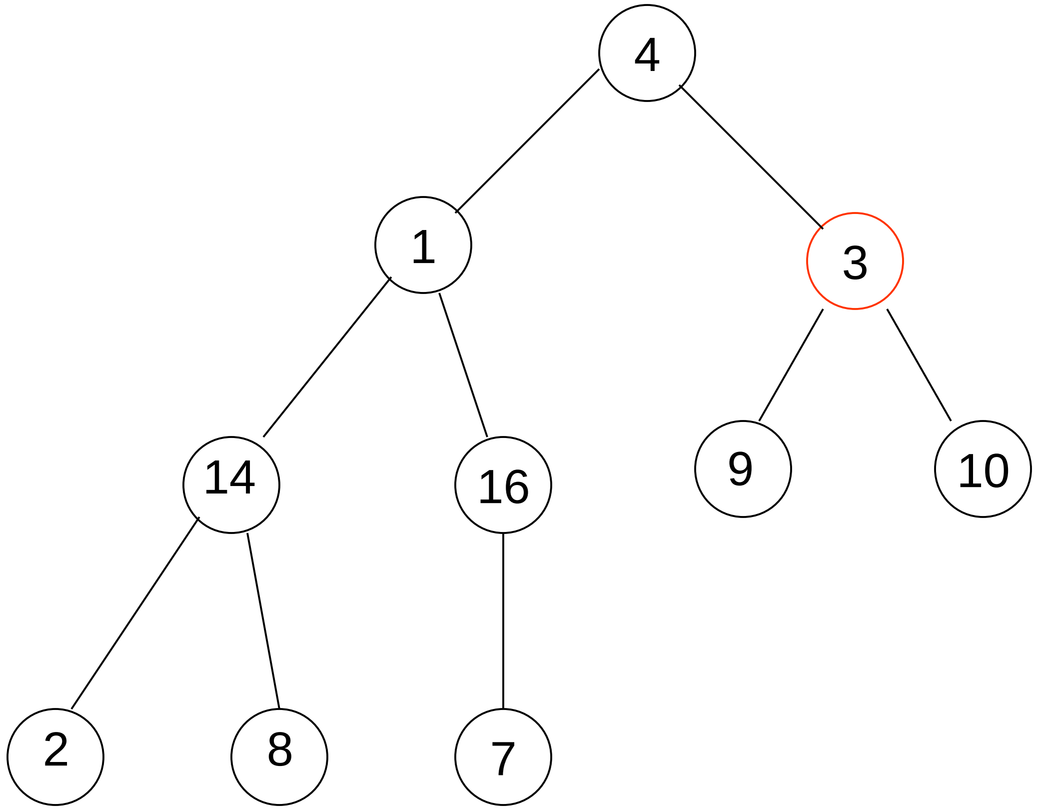
Finished heap: A=[14 7 10 6 5 9 2 4 3 1]

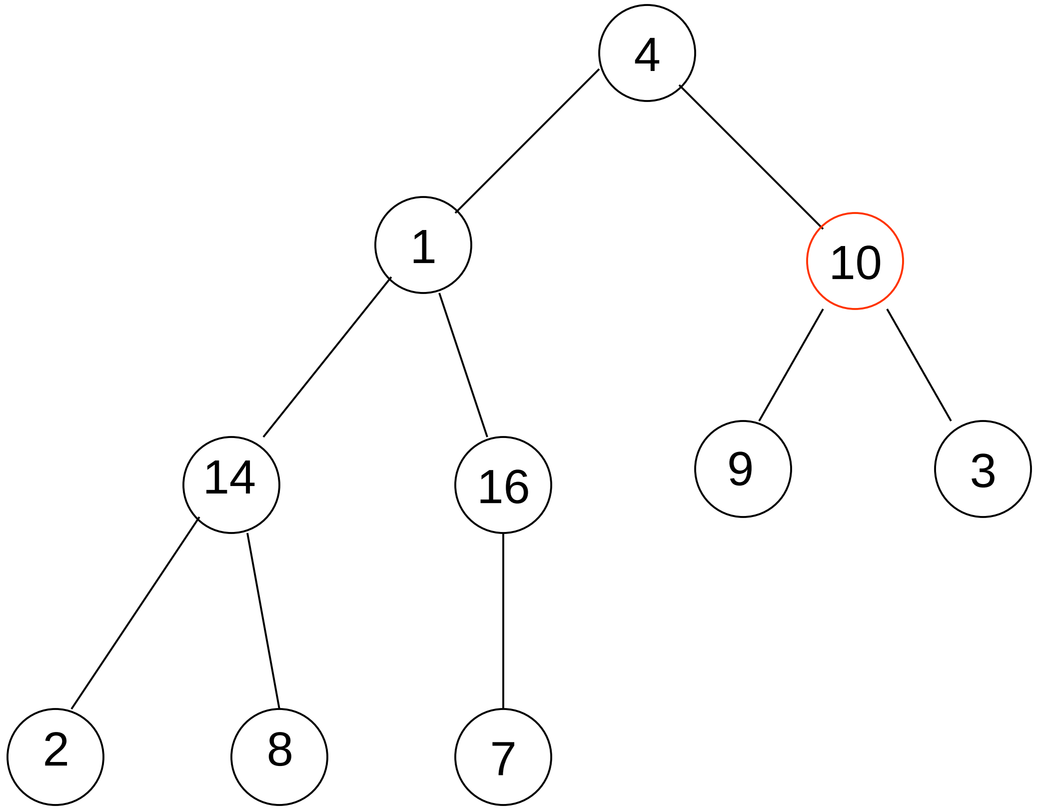
Build-Max-Heap Example 2



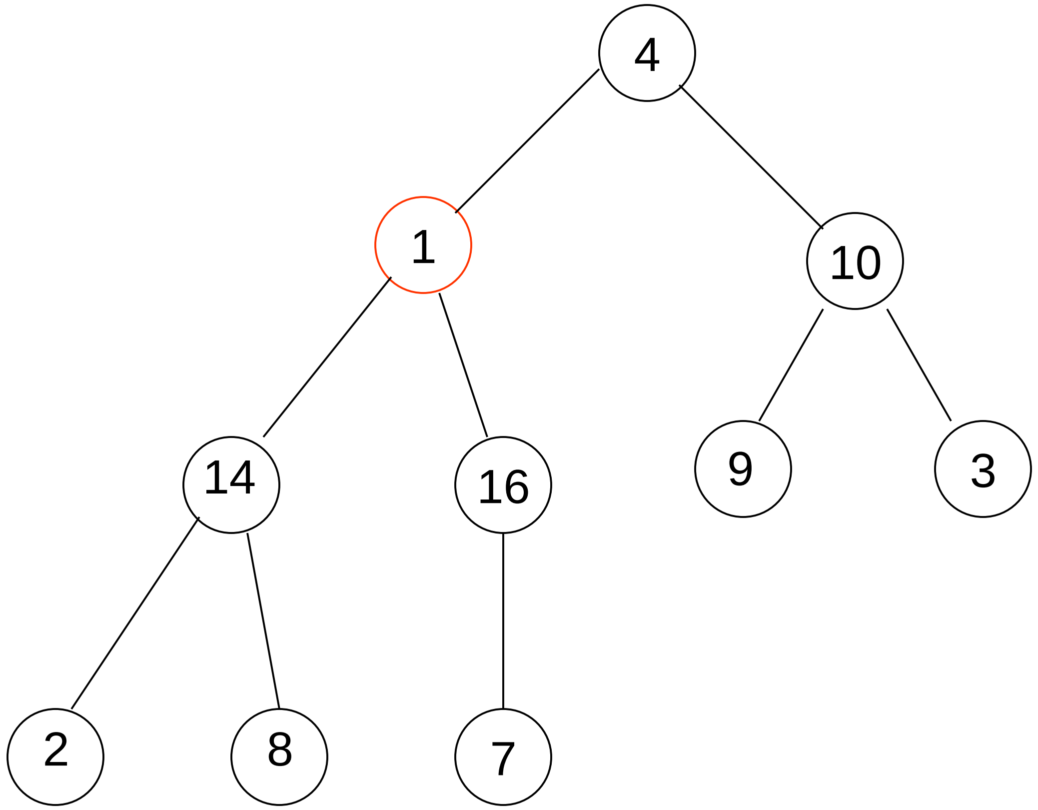

4
1
3
2
16
9
10
14
8
7

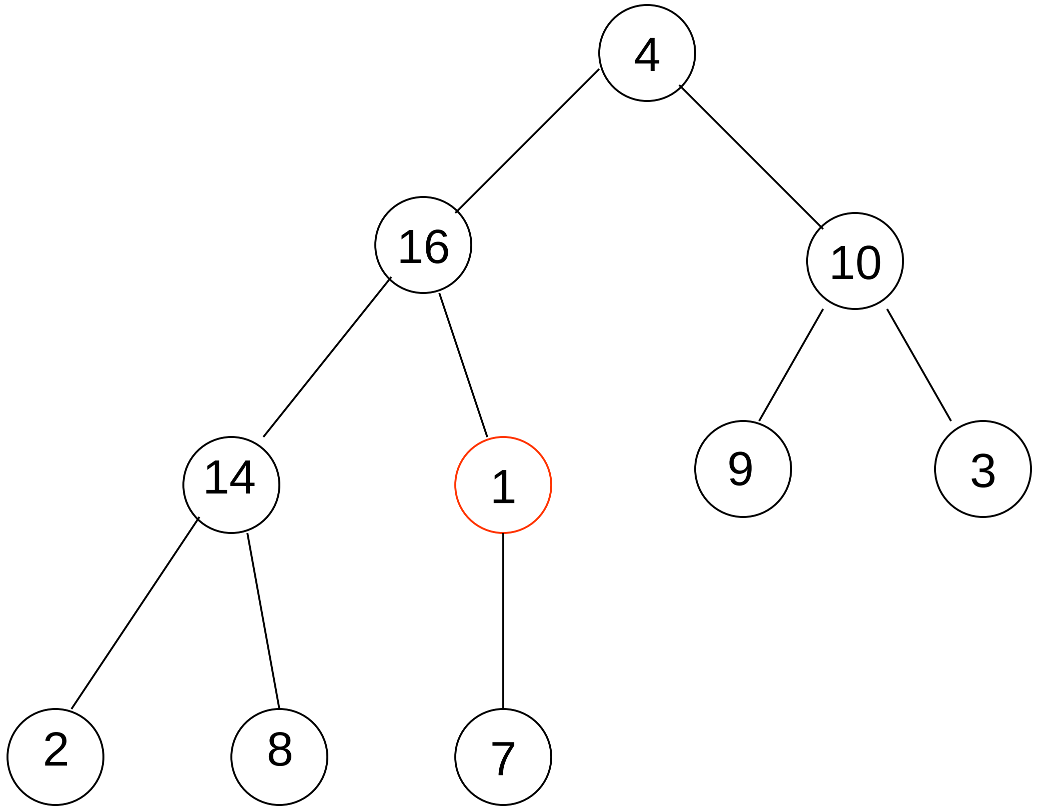

4
1
3
14
16
9
10
2
8
7

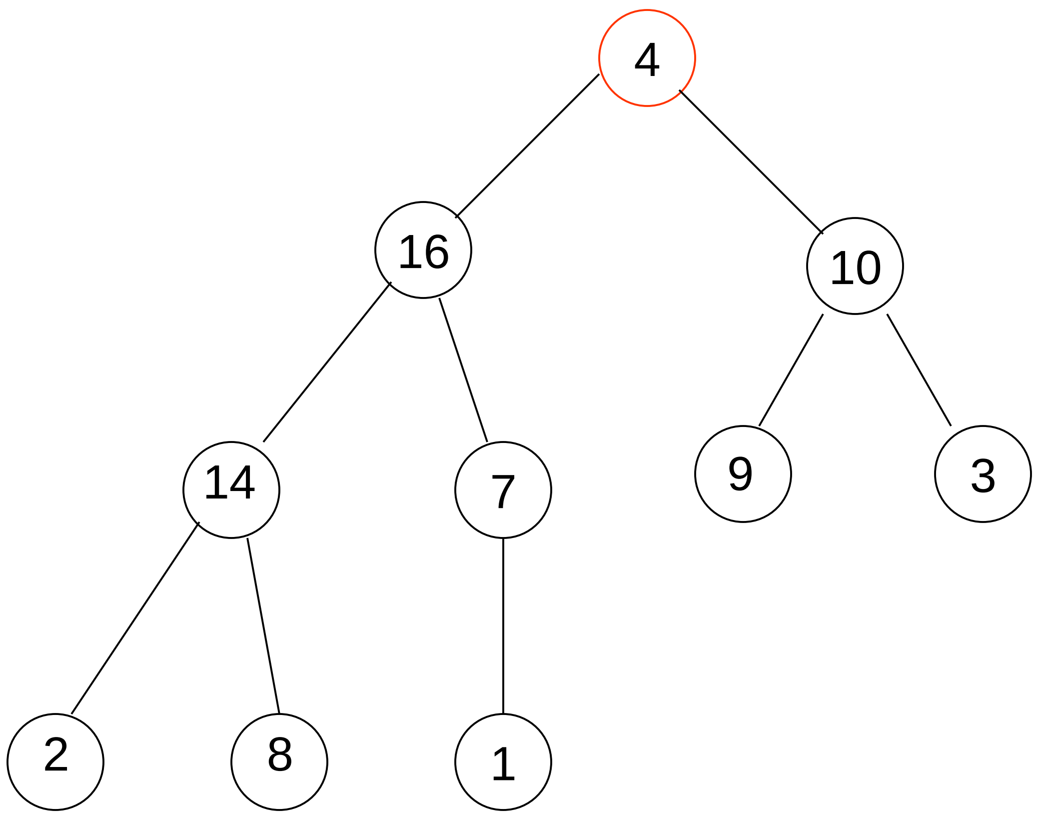

4
1
3
14
16
9
10
2
8
7

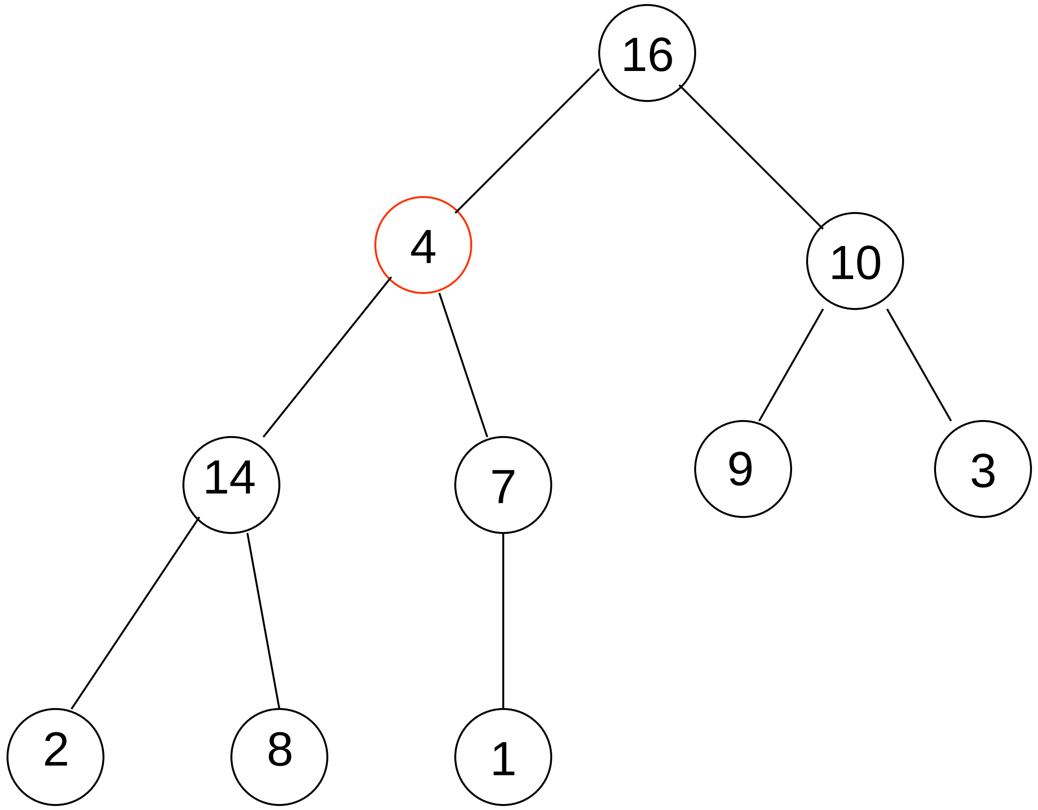


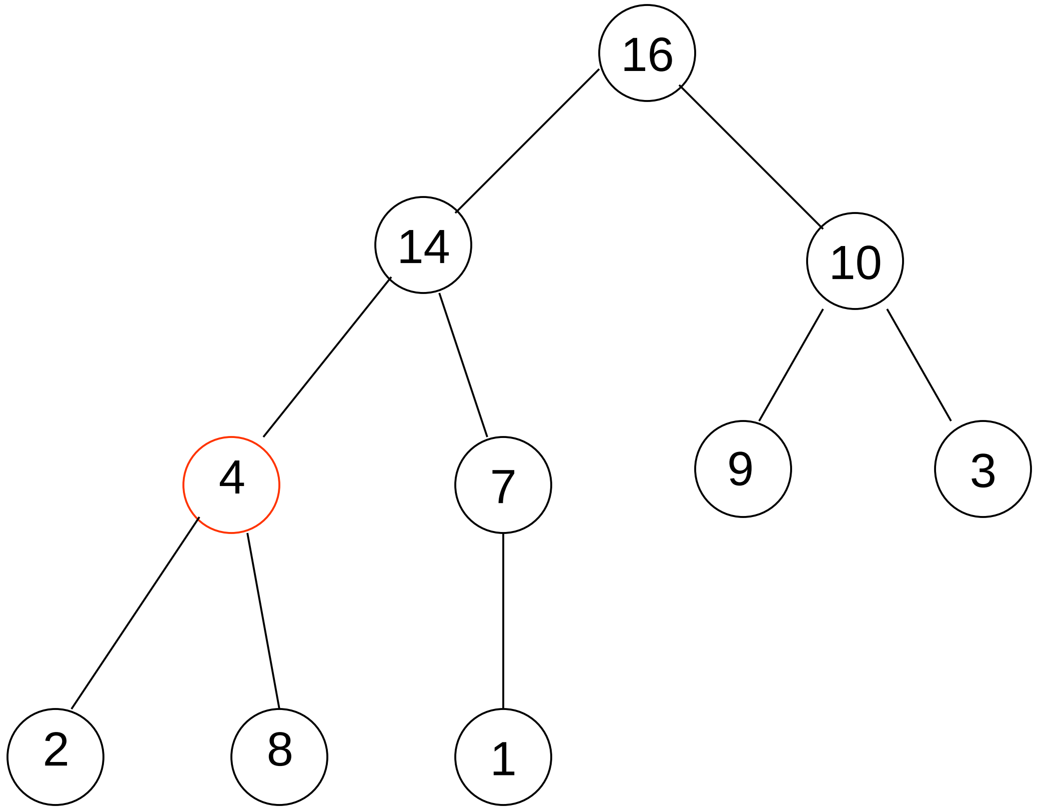
4
1
10
14
16
9
3
2
8
7

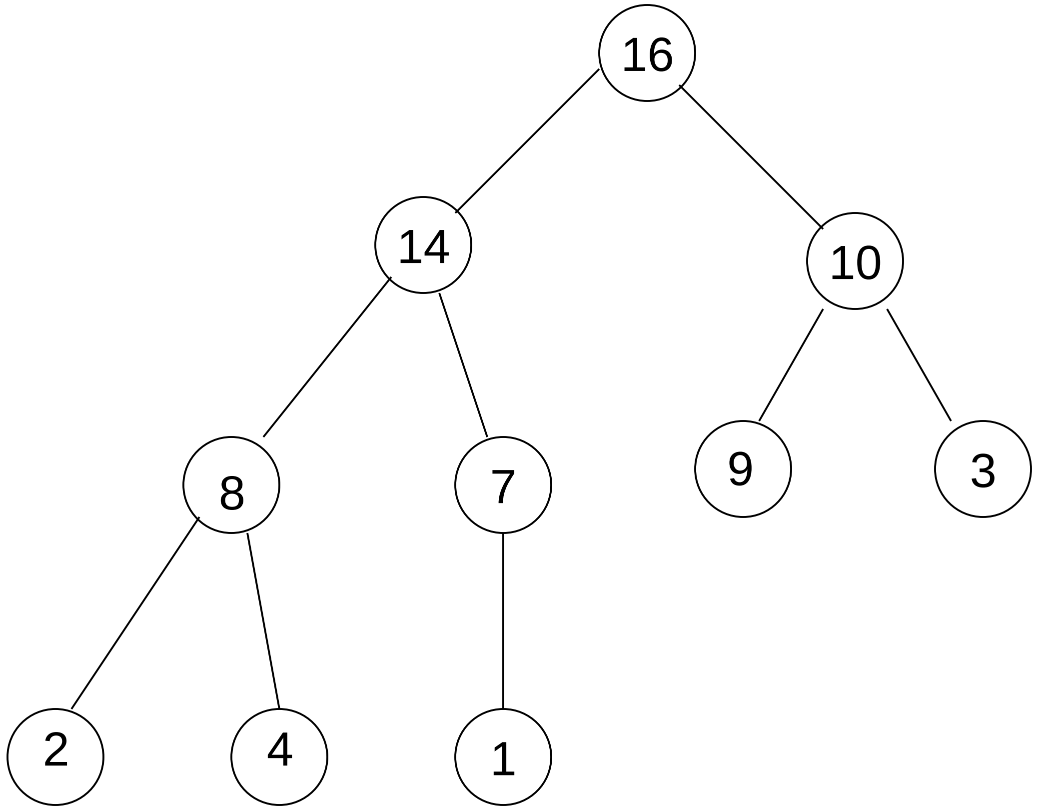

4
1
10
14
16
9
3
2
8
7


4
16
10
14
1
9
3
2
8
7


4
16
10
14
7
9
3
2
8
1


16
4
10
14
7
9
3
2
8
1


16
14
10
4
7
9
3
2
8
1


16
14
10
8
7
9
3
2
4
1

Heapsort

Once we can build a heap and heapify, sorting is easy… just remove max N times

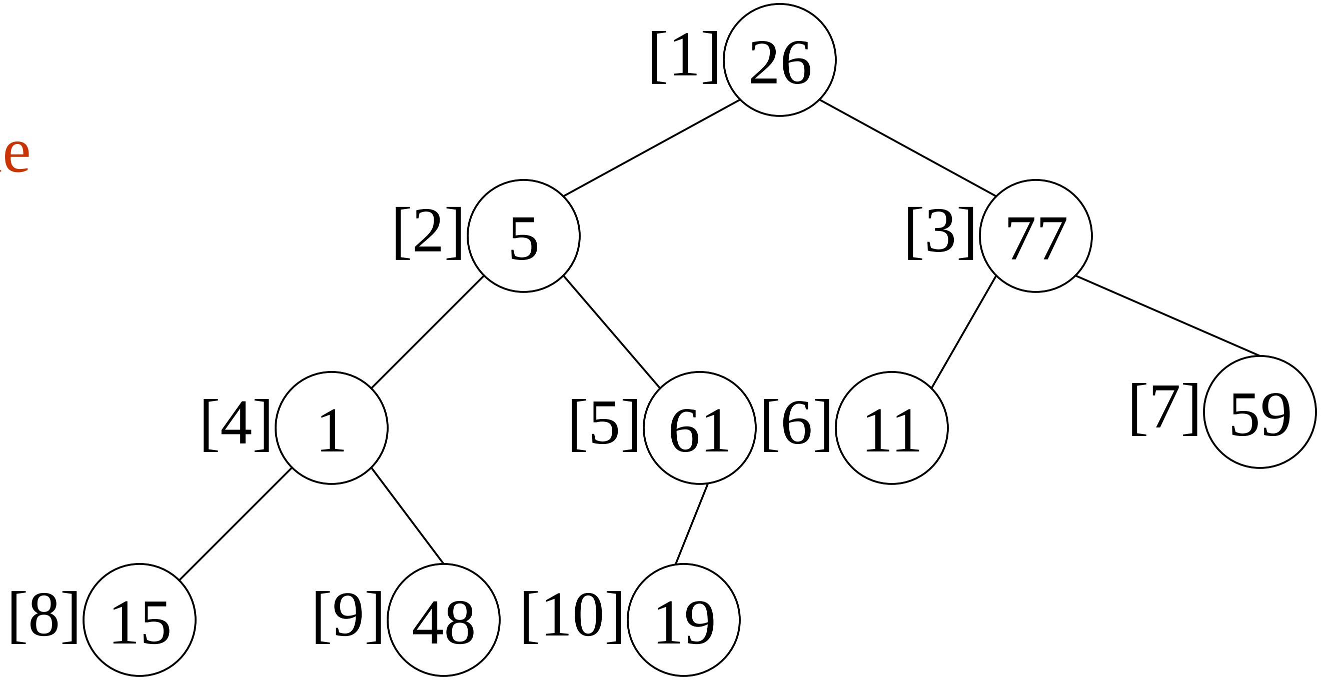
```
HeapSort(A,n)
    Build-MAX-Heap(A,n)
    for i ←n downto 2
        exchange A[1]↔A[i]
        n = n - 1
        Max-Heapify(A,1,n)
```

Example: Heap Sort

- **Array interpreted as a binary tree**

1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

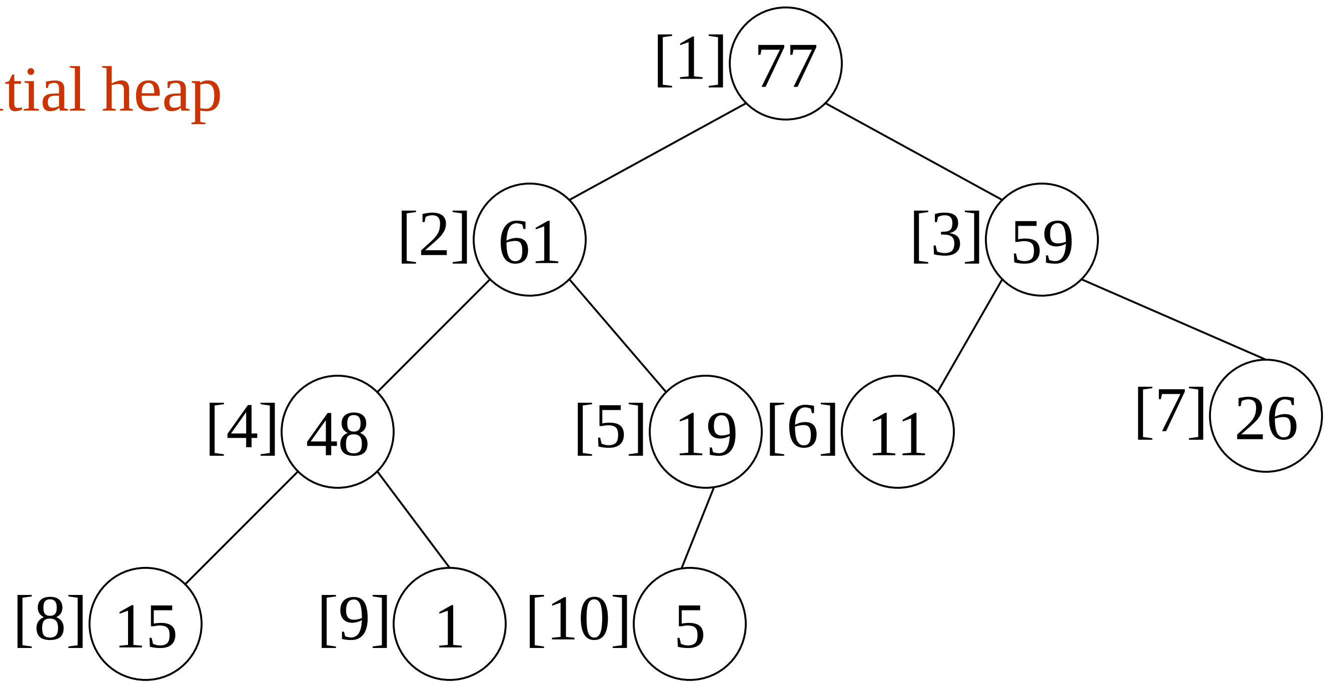
input file



Heap Sort

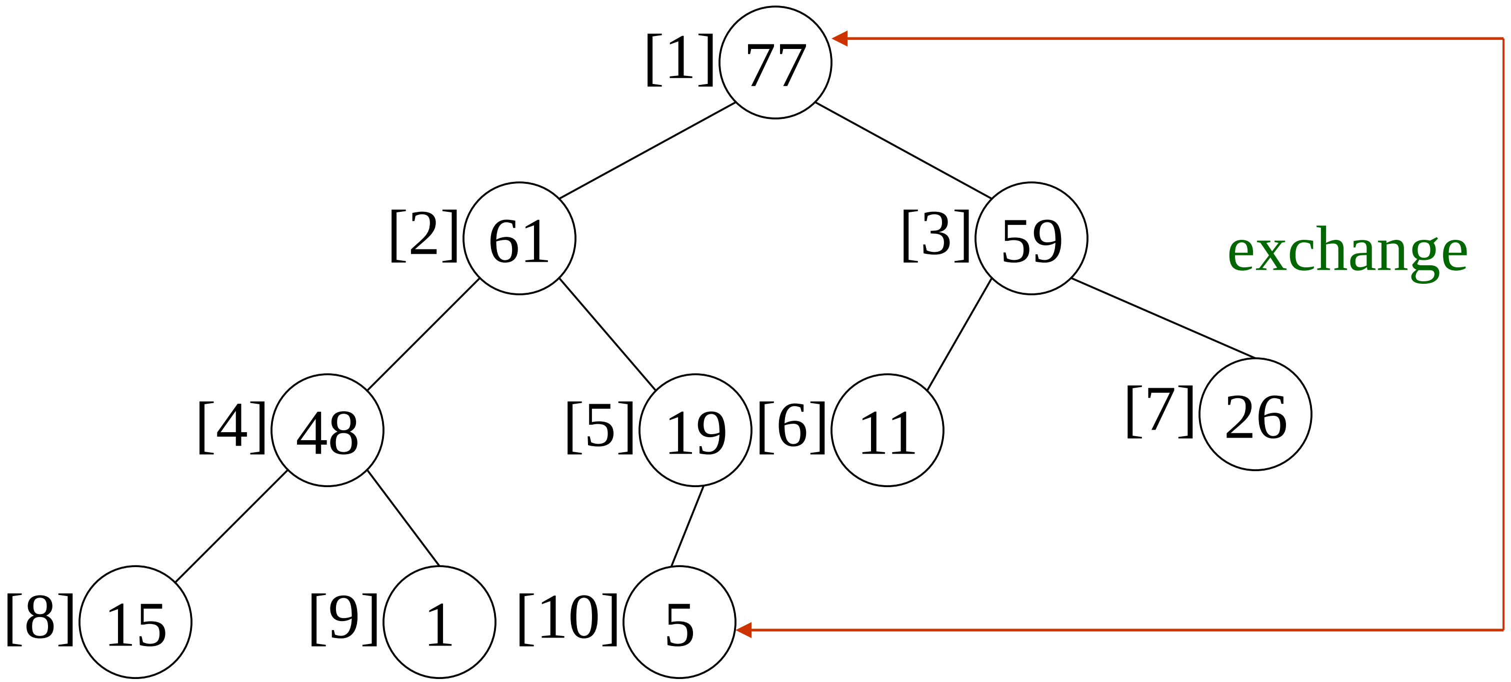
Adjust it to a MaxHeap

initial heap

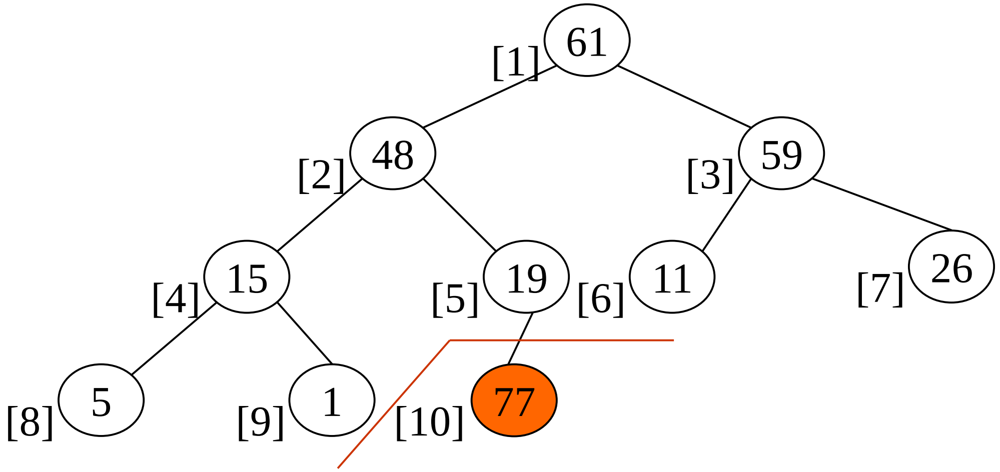


Heap Sort

Exchange and adjust

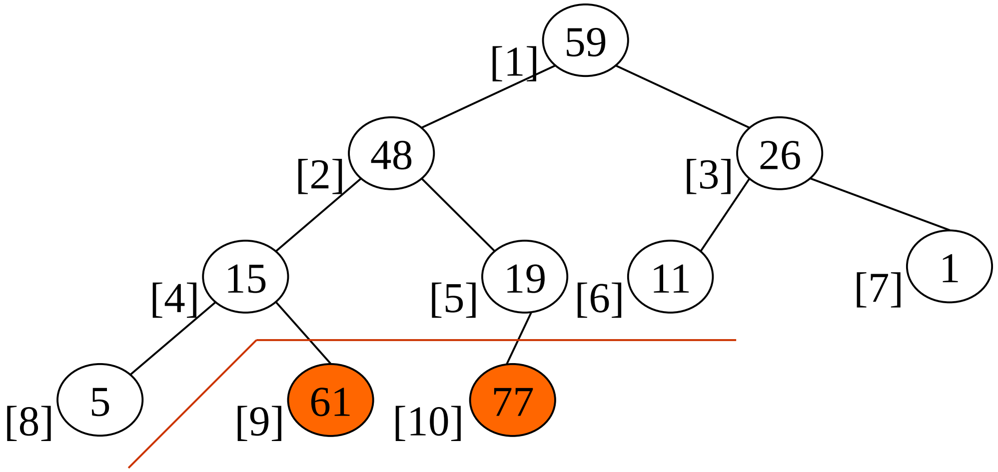


Heap Sort



[1] 61
[2] 48
[3] 59
[4] 15
[5] 19
[6] 11
[7] 26
[8] 5
[9] 1
[10] 77

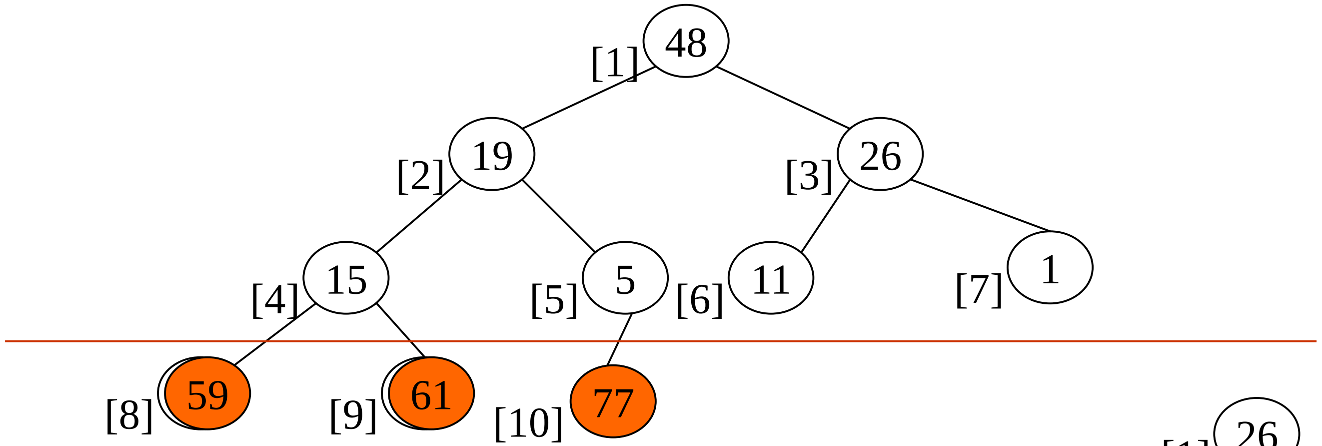
(a)



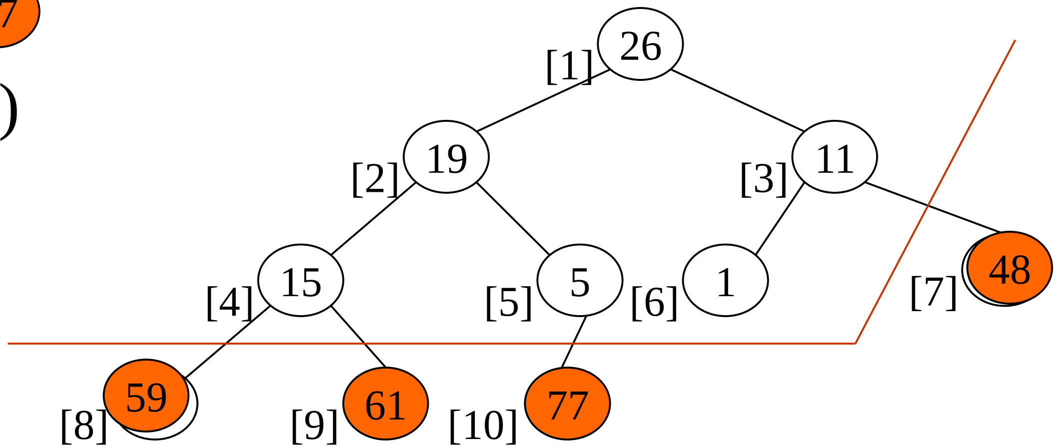
[1] 59
[2] 48
[3] 26
[4] 15
[5] 19
[6] 11
[7] 1
[8] 5
[9] 61
[10] 77

(b)

Heap Sort

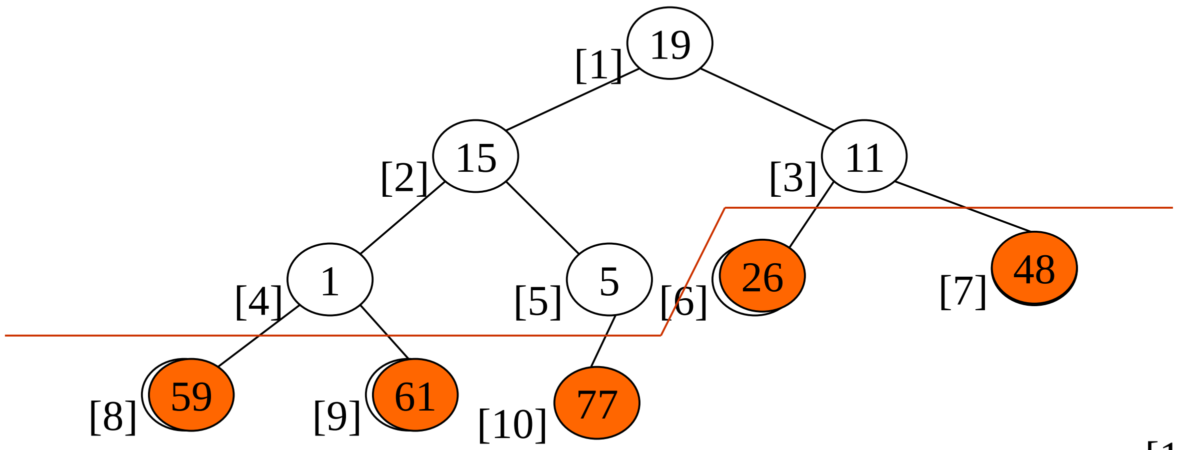

48
[1]
19
[2]
26
[3]
15
[4]
5
[5]
11
[6]
1
[7]
59
[8]
61
[9]
77
[10]

(c)


26
[1]
19
[2]
11
[3]
15
[4]
5
[5]
1
[6]
48
[7]
59
[8]
61
[9]
77
[10]

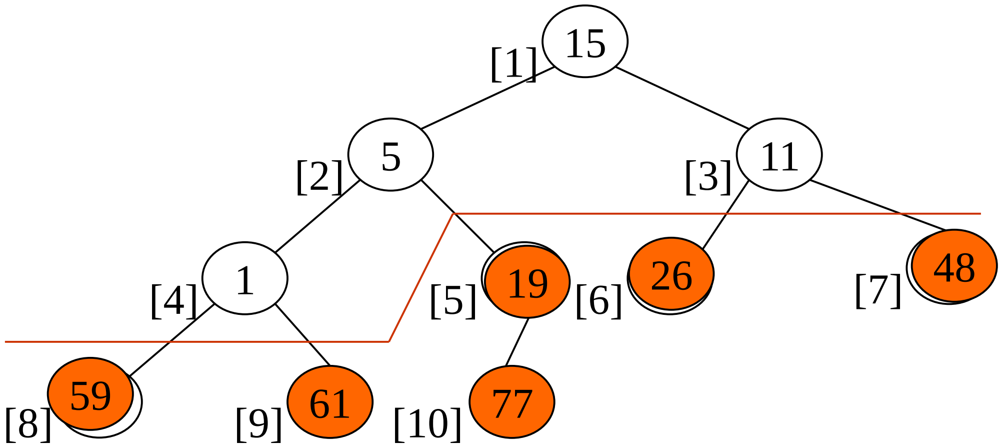
(d)

Heap Sort



19
15
11
1
5
26
48
59
61
77
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]

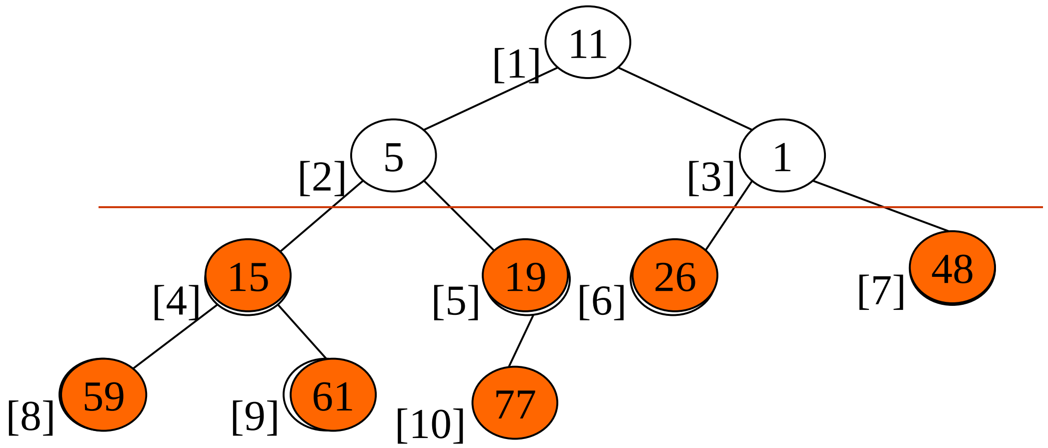
(e)



15
5
11
1
19
26
48
59
61
77
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]

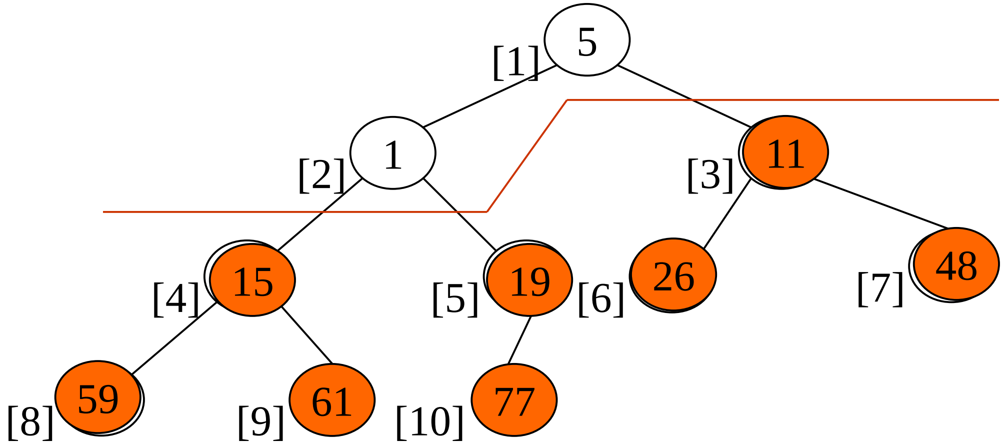
(f)

Heap Sort



11
5
1
15
19
26
48
59
61
77
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]

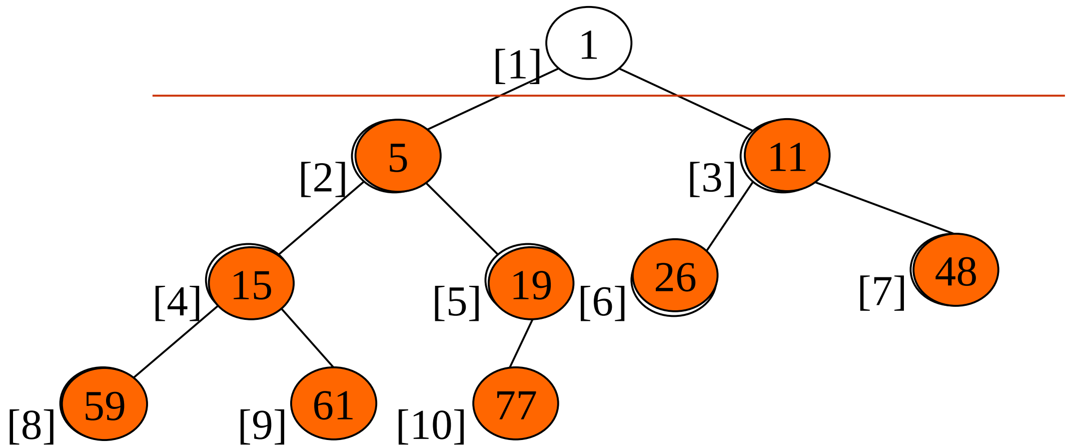
(g)



5
1
11
15
19
26
48
59
61
77
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]

(h)

Heap Sort

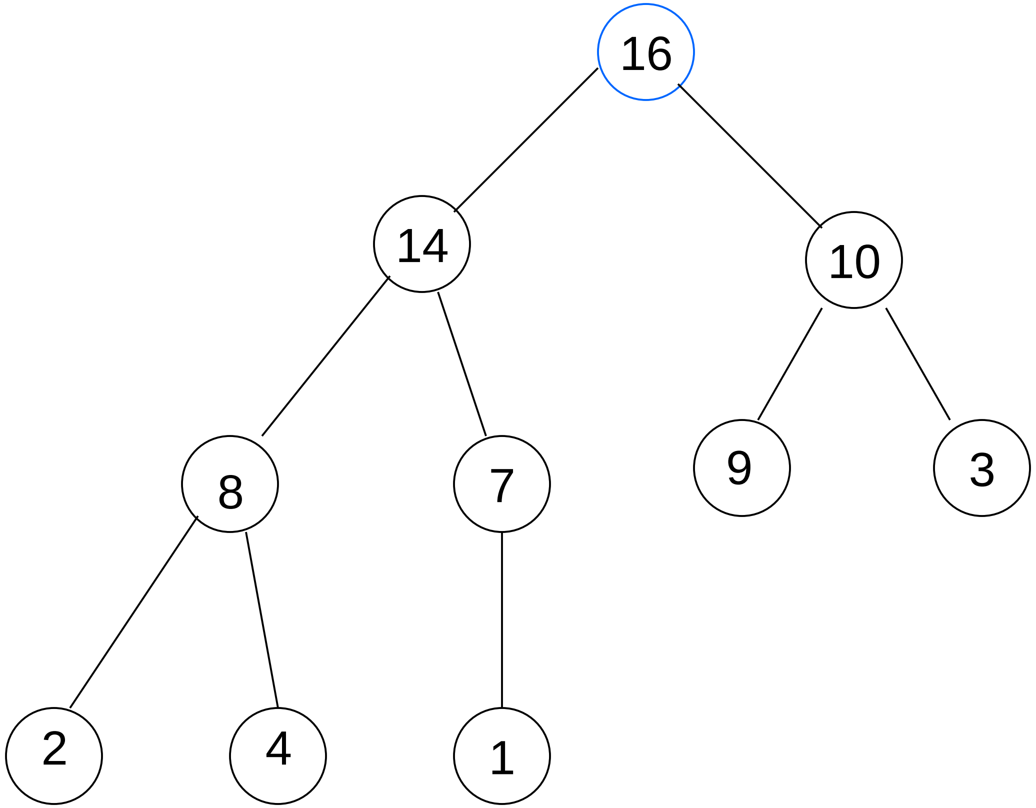


(i)

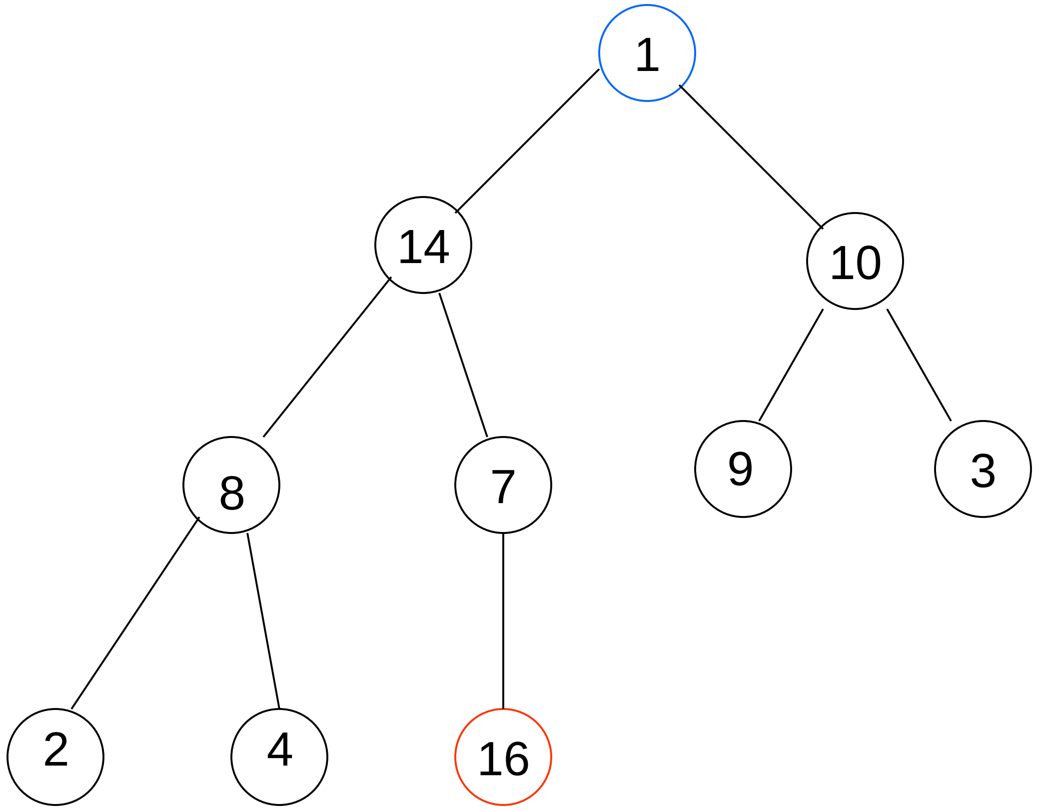
Heap Sort Example 2

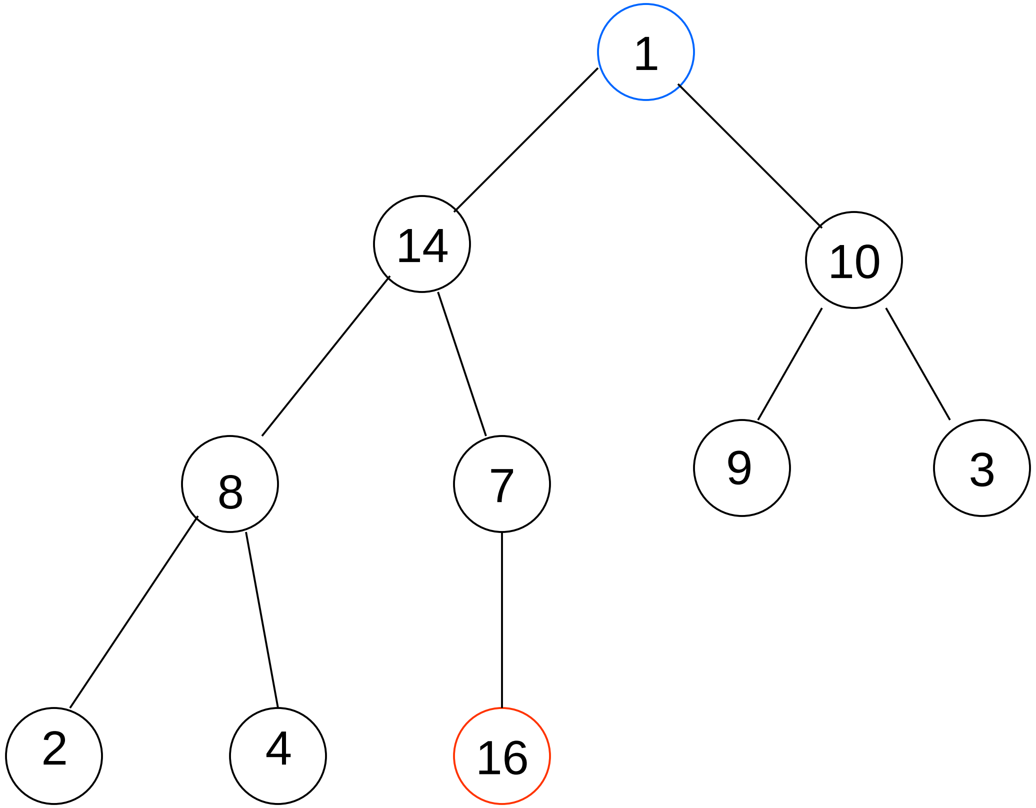
Input: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

Build a max-heap

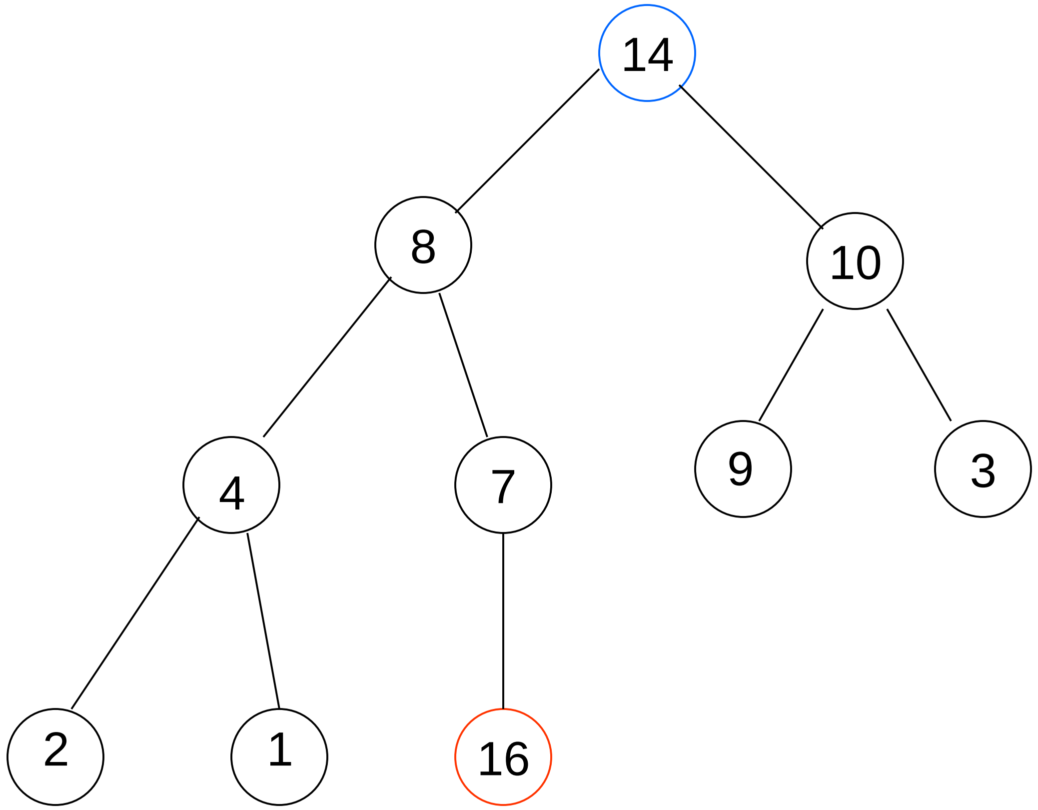


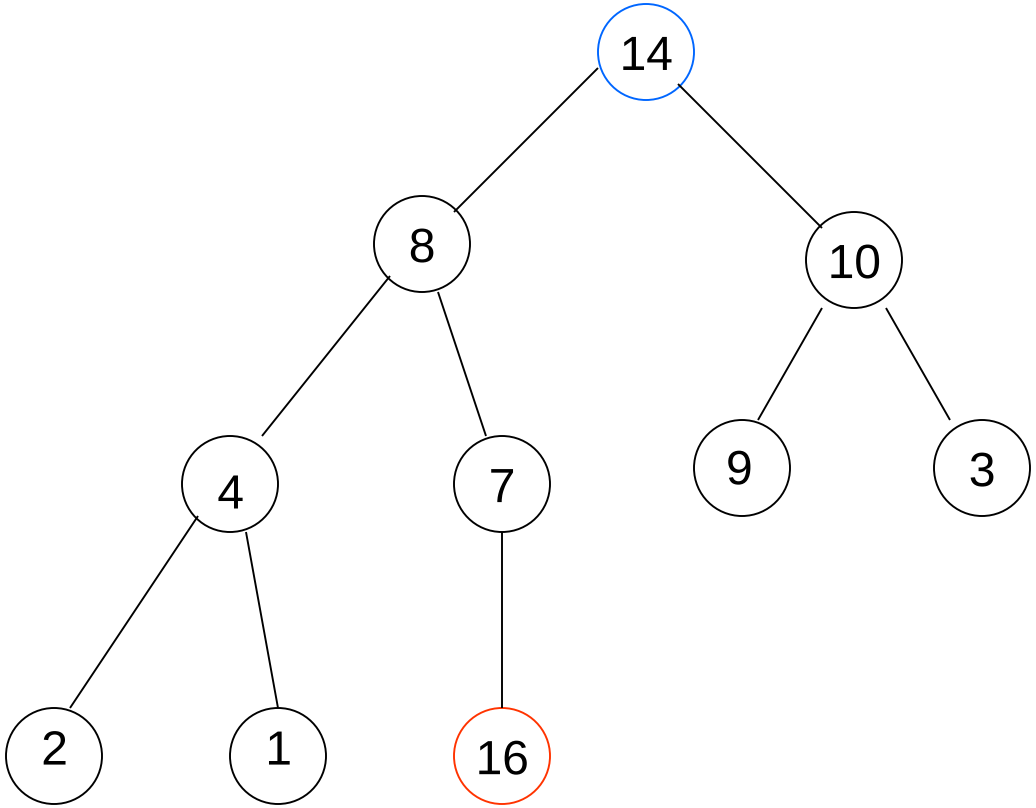
16, 14, 10, 8, 7, 9, 3, 2, 4, 1.


1
14
10
8
7
9
3
2
4
16

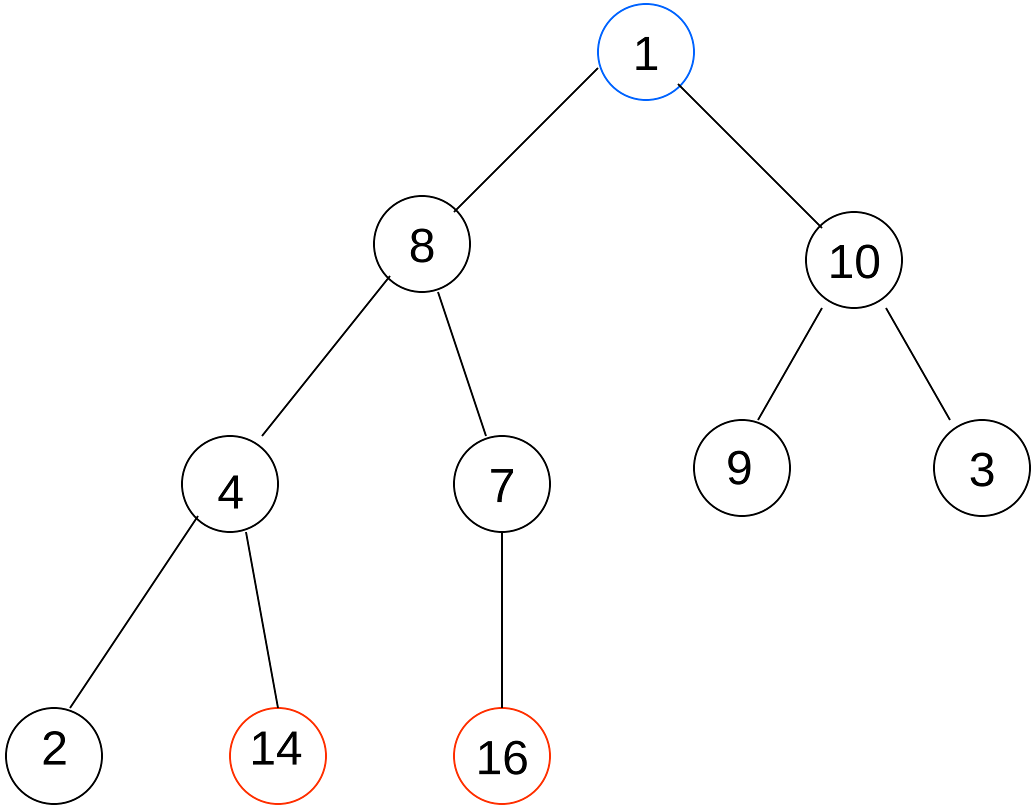


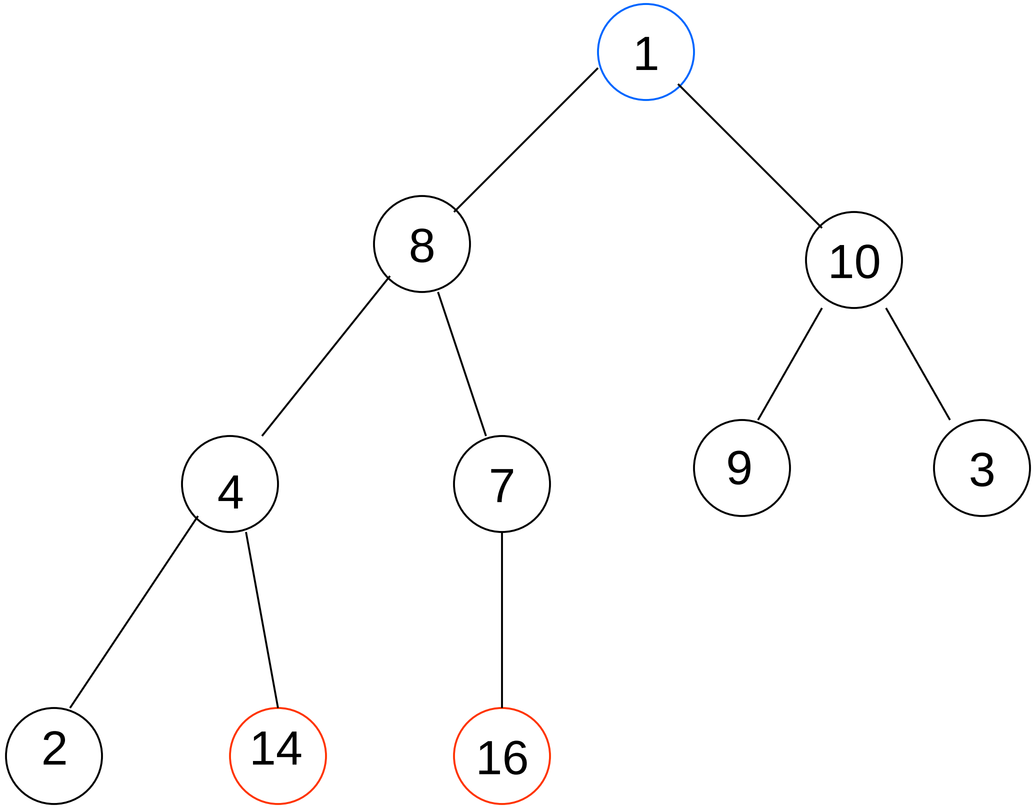
1, 14, 10, 8, 7, 9, 3, 2, 4, 16.


14
8
10
4
7
9
3
2
1
16

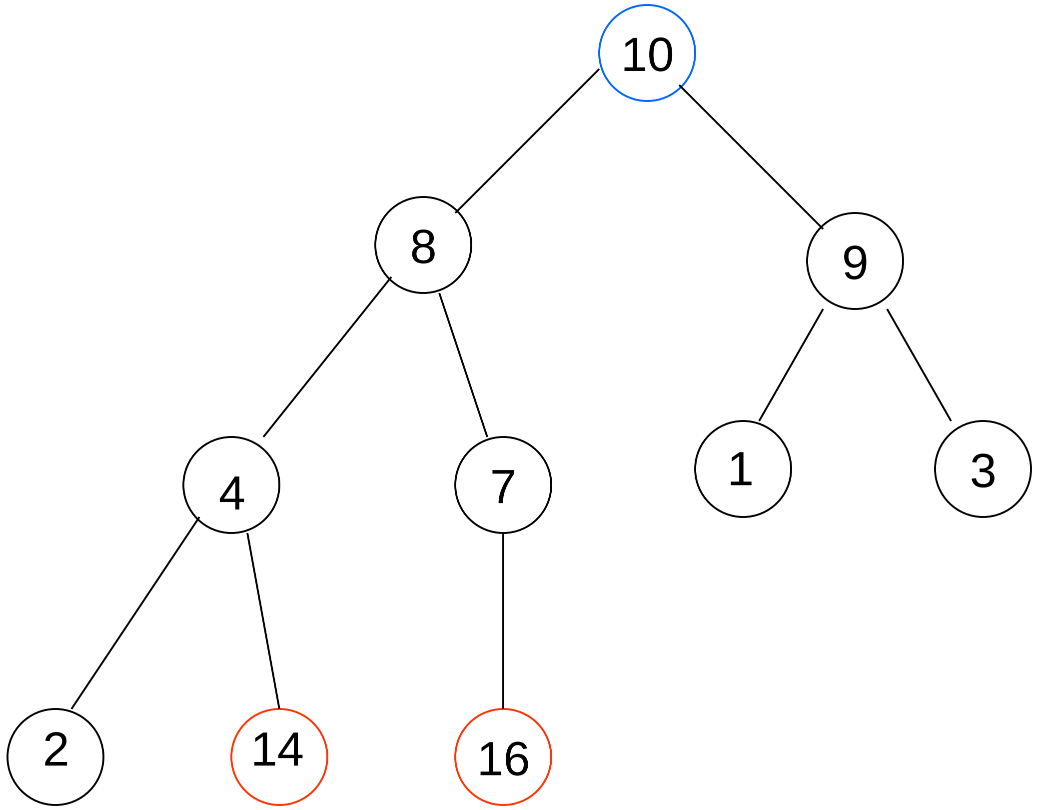


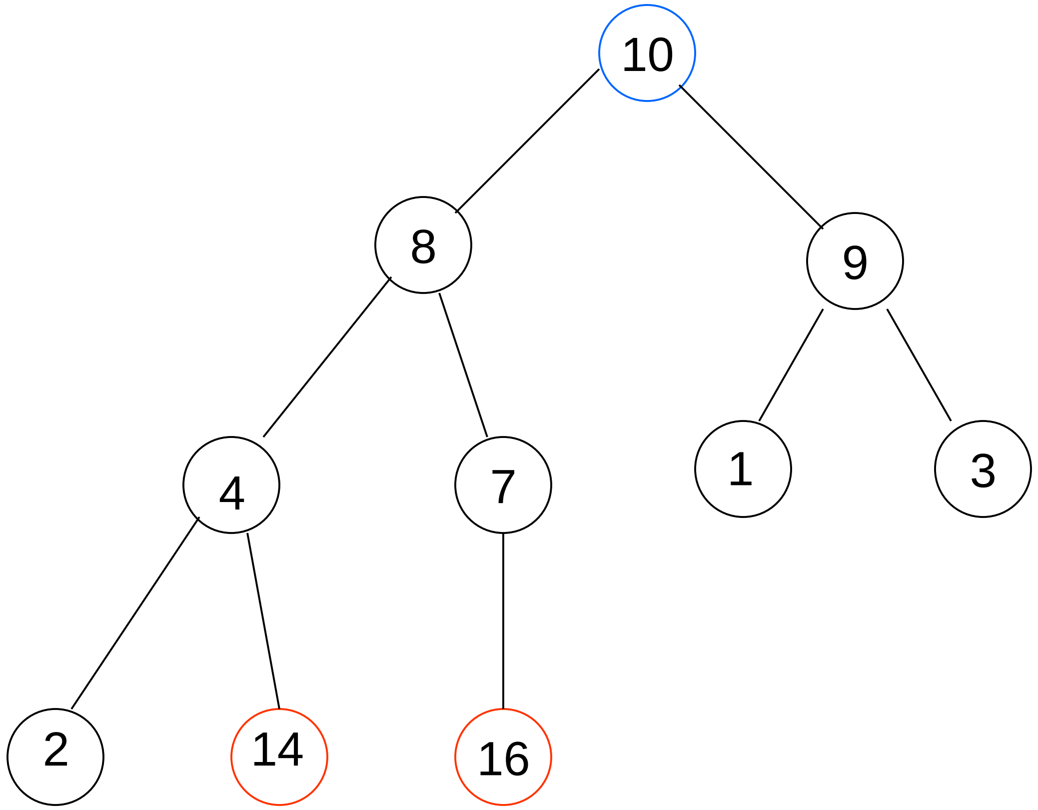
14, 8, 10, 4, 7, 9, 3, 2, 1, 16.


1
8
10
4
7
9
3
2
14
16

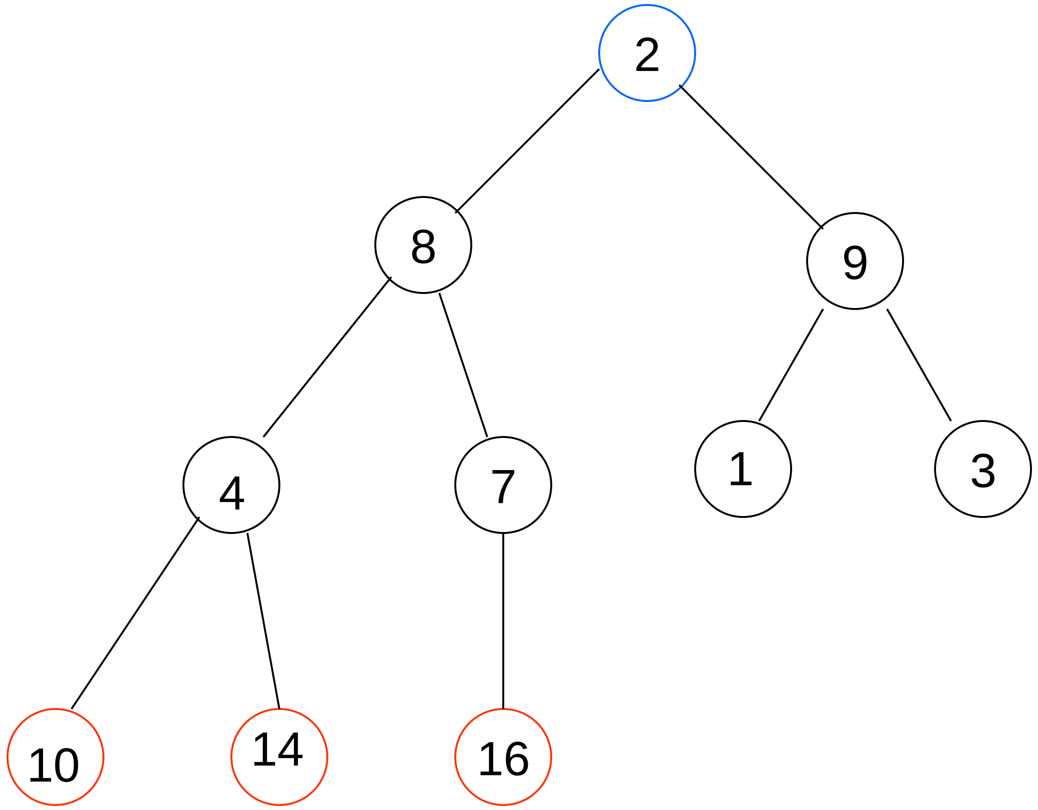


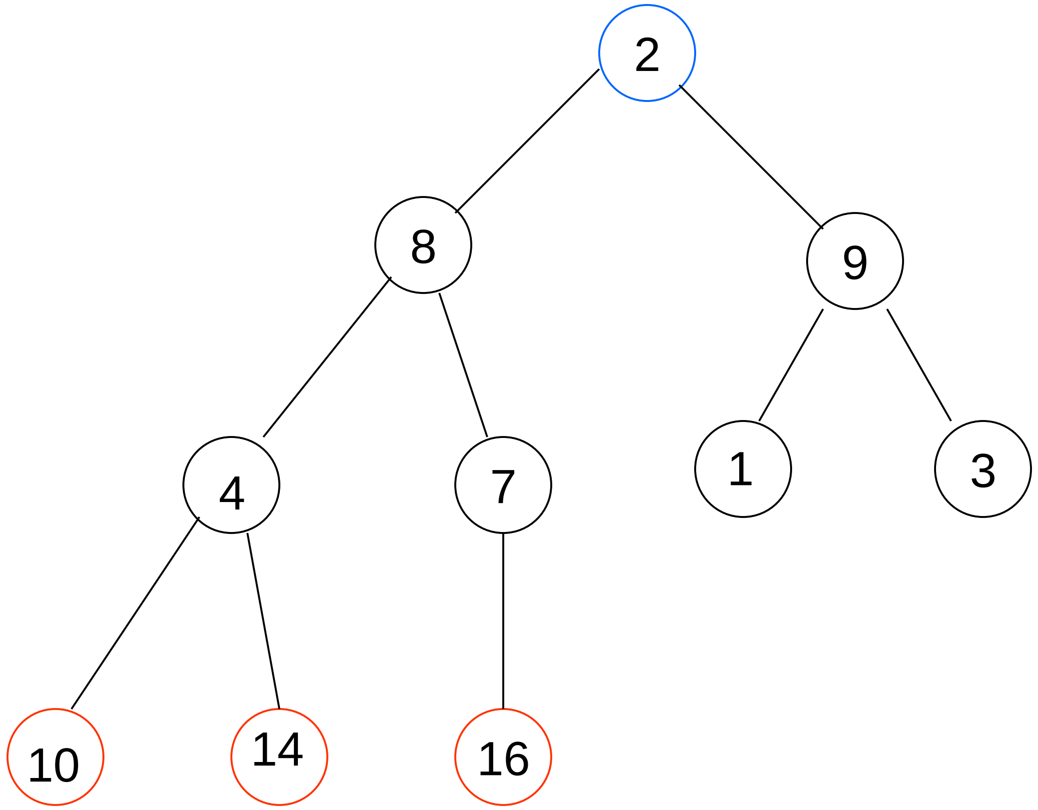
1, 8, 10, 4, 7, 9, 3, 2, 14, 16.


10
8
9
4
7
1
3
2
14
16

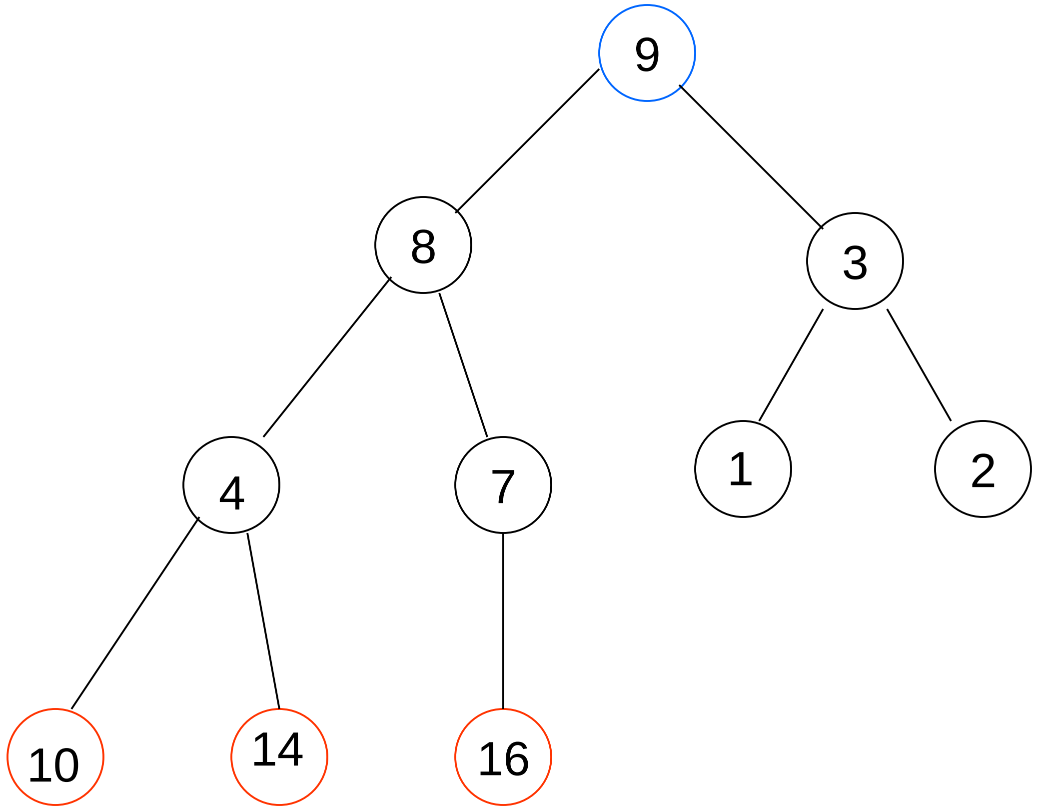


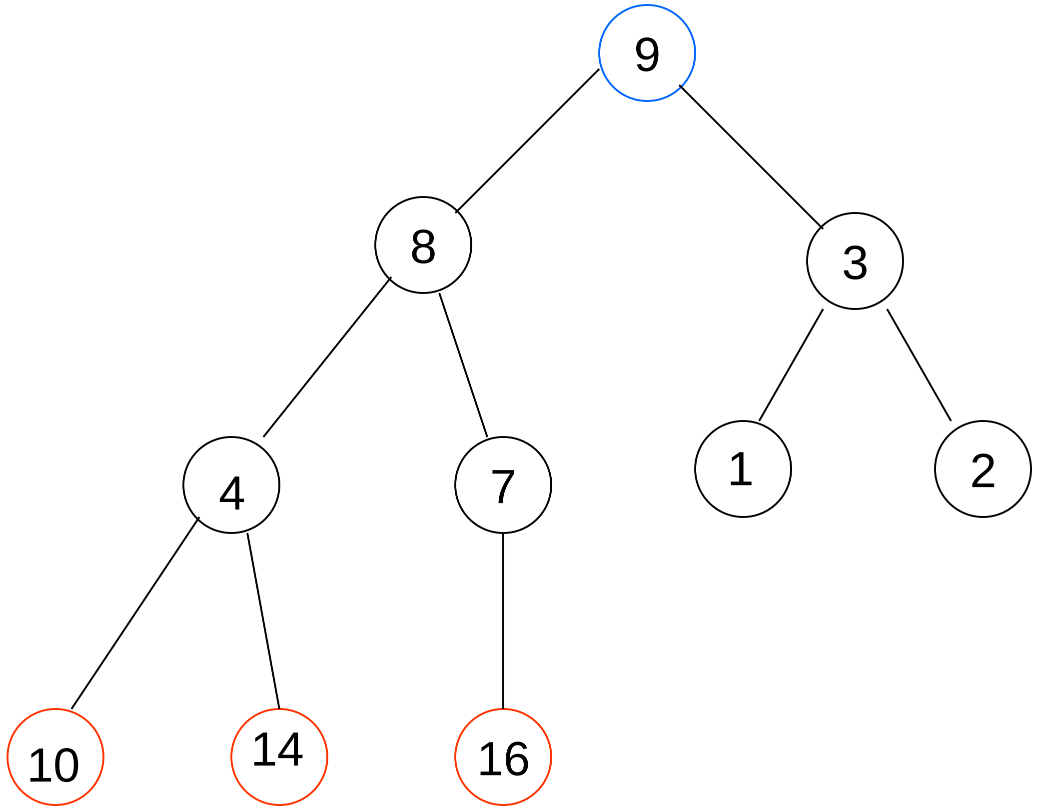
10, 8, 9, 4, 7, 1, 3, 2, 14, 16.


2
8
9
4
7
1
3
10
14
16

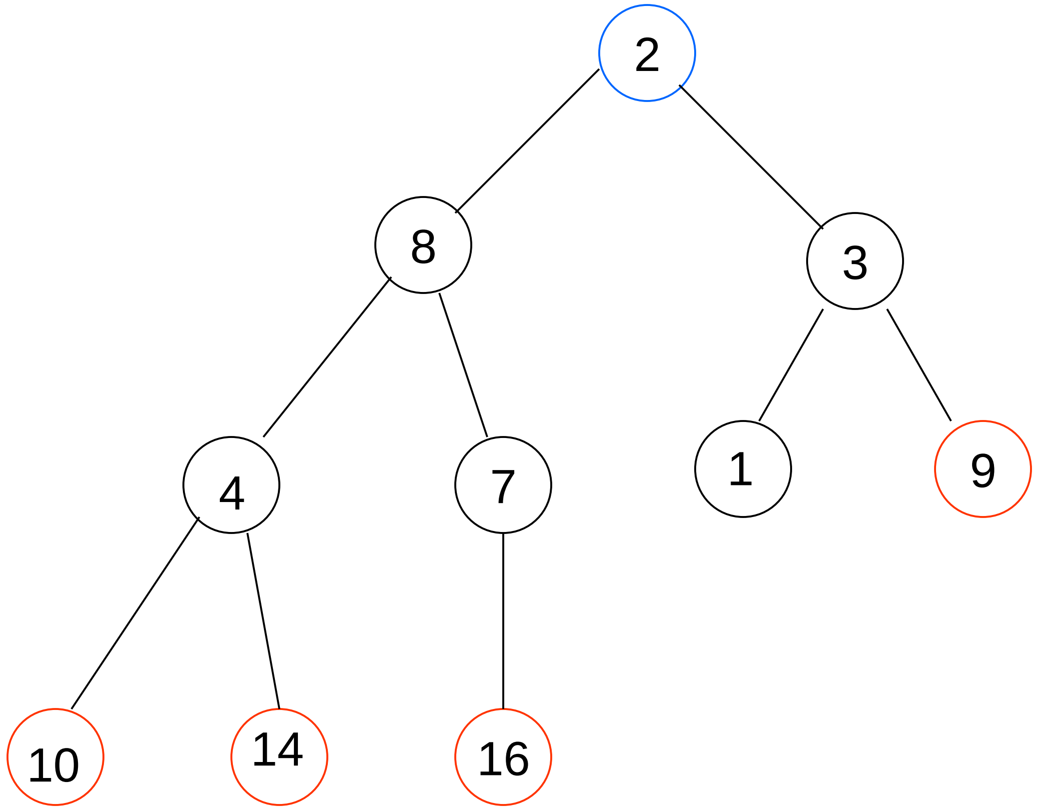


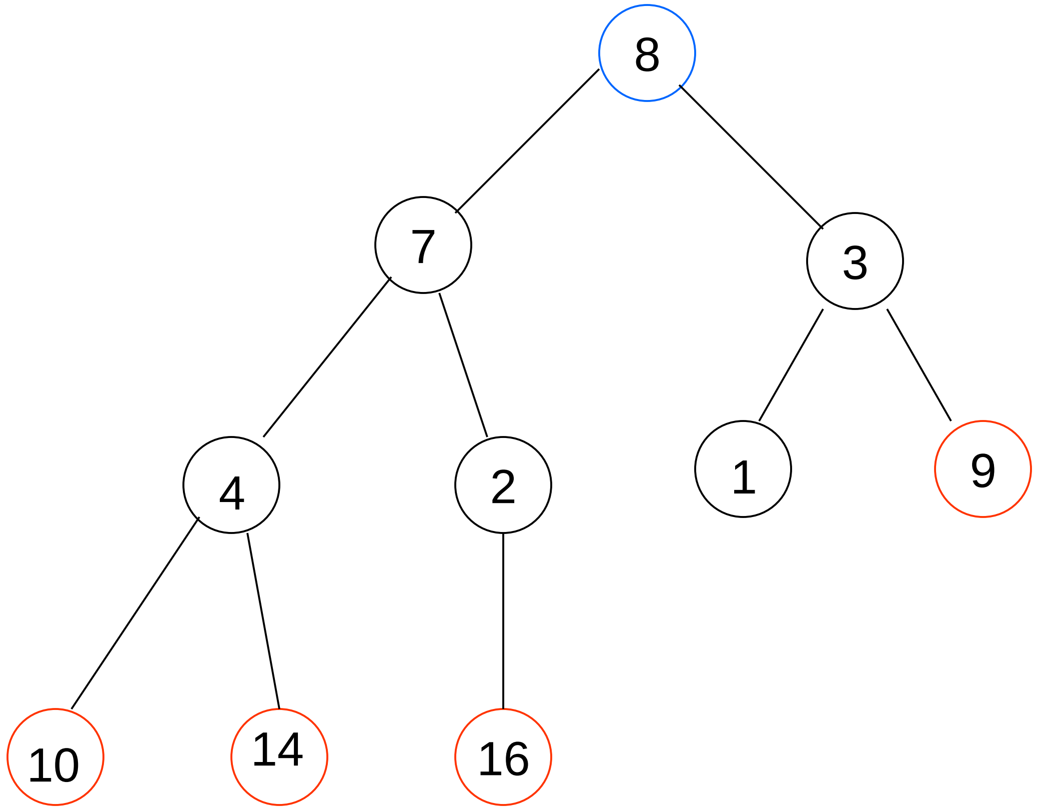
2, 8, 9, 4, 7, 1, 3, 10, 14, 16.

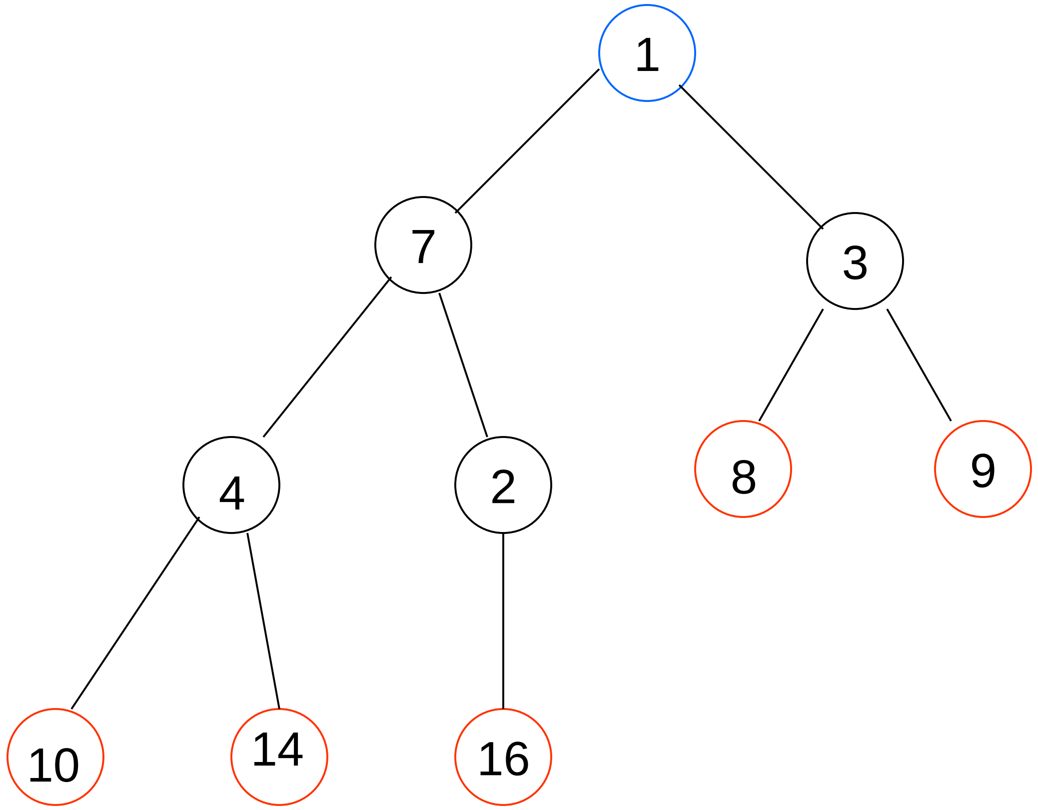

9
8
3
4
7
1
2
10
14
16

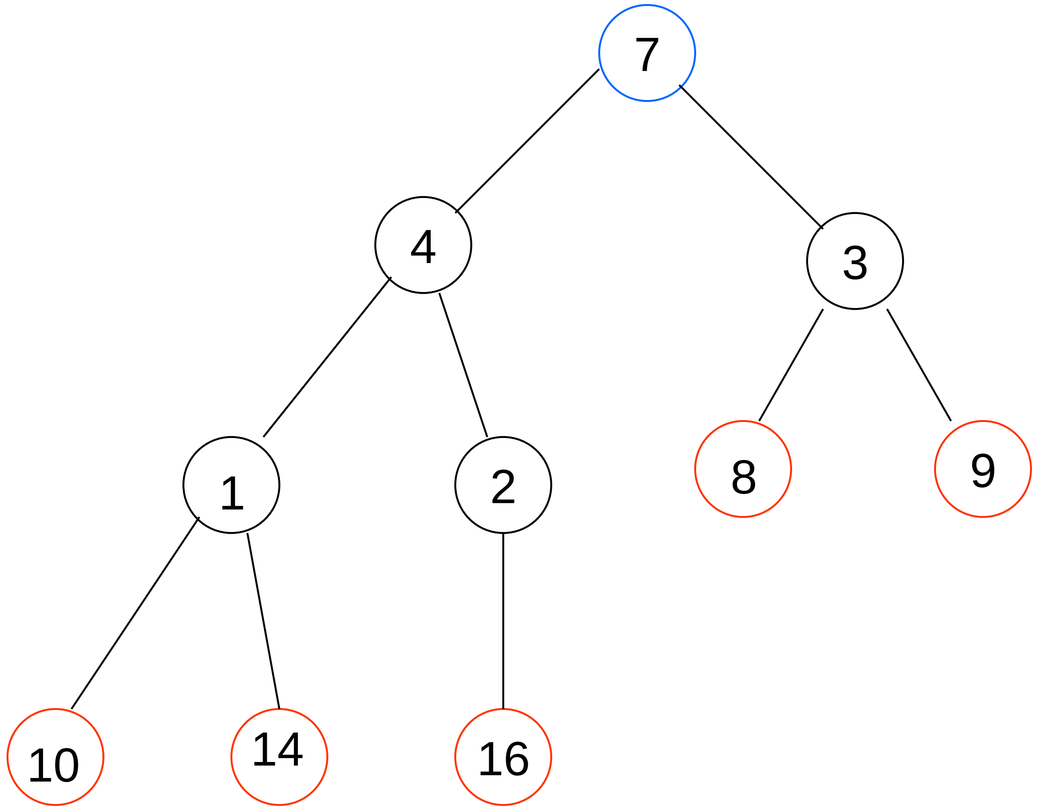


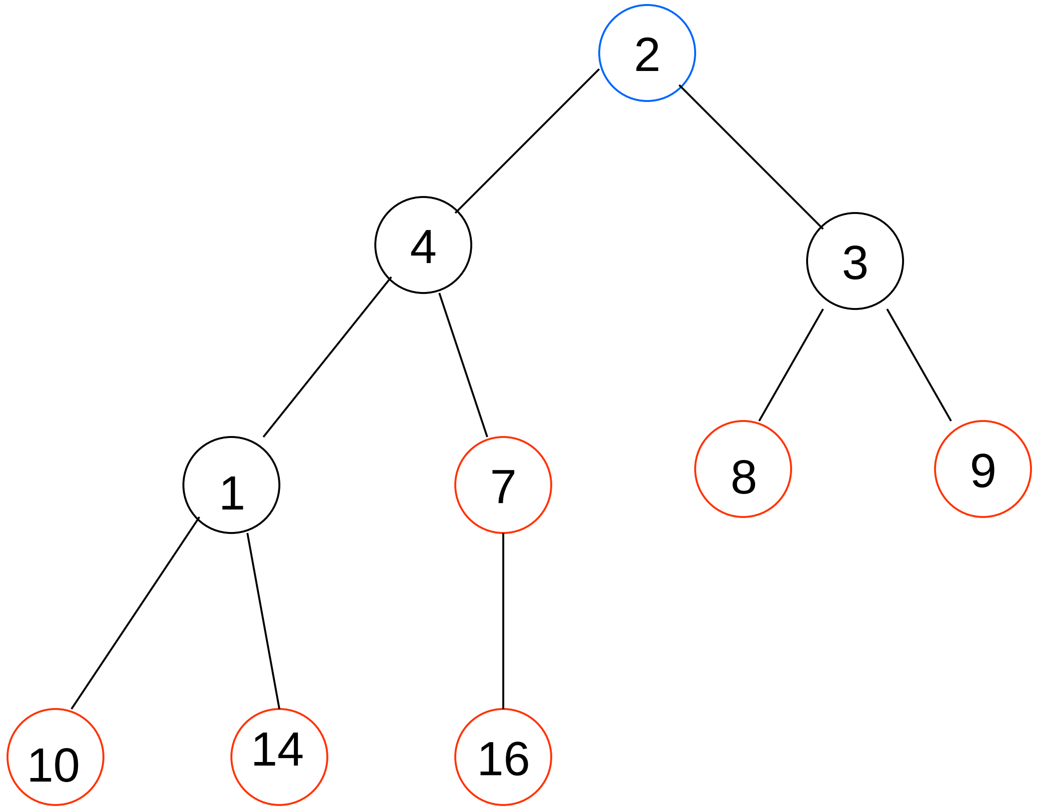
9, 8, 3, 4, 7, 1, 2, 10, 14, 16.

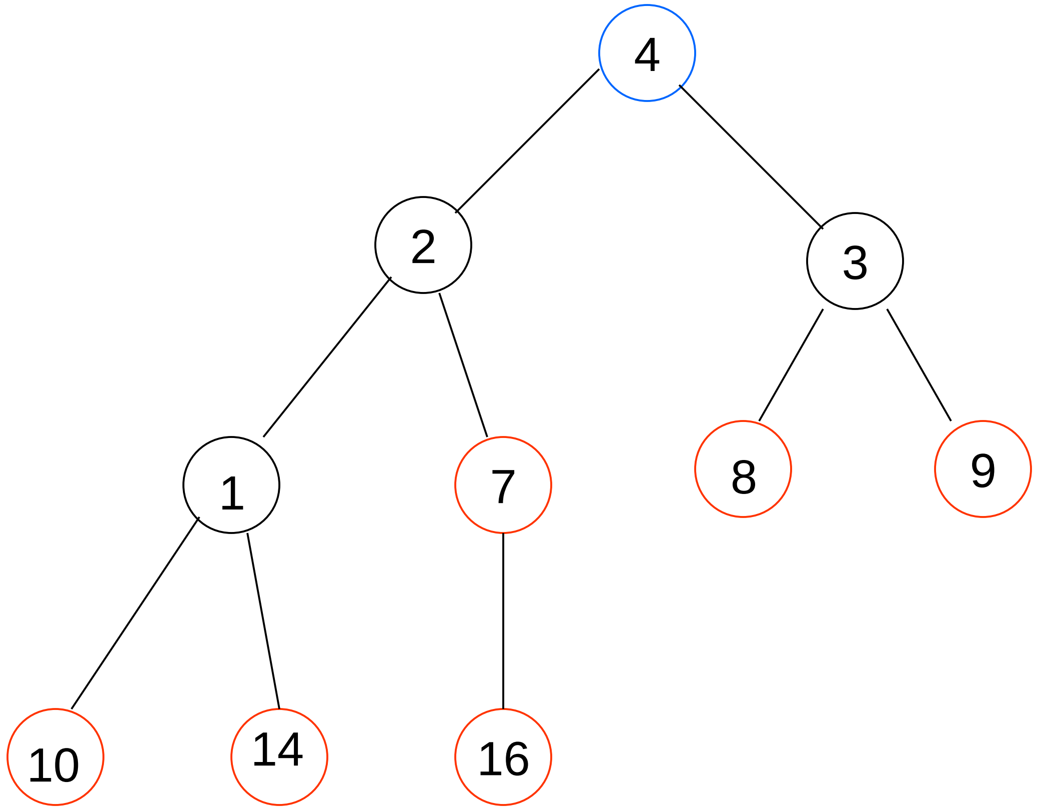

2
8
3
4
7
1
9
10
14
16

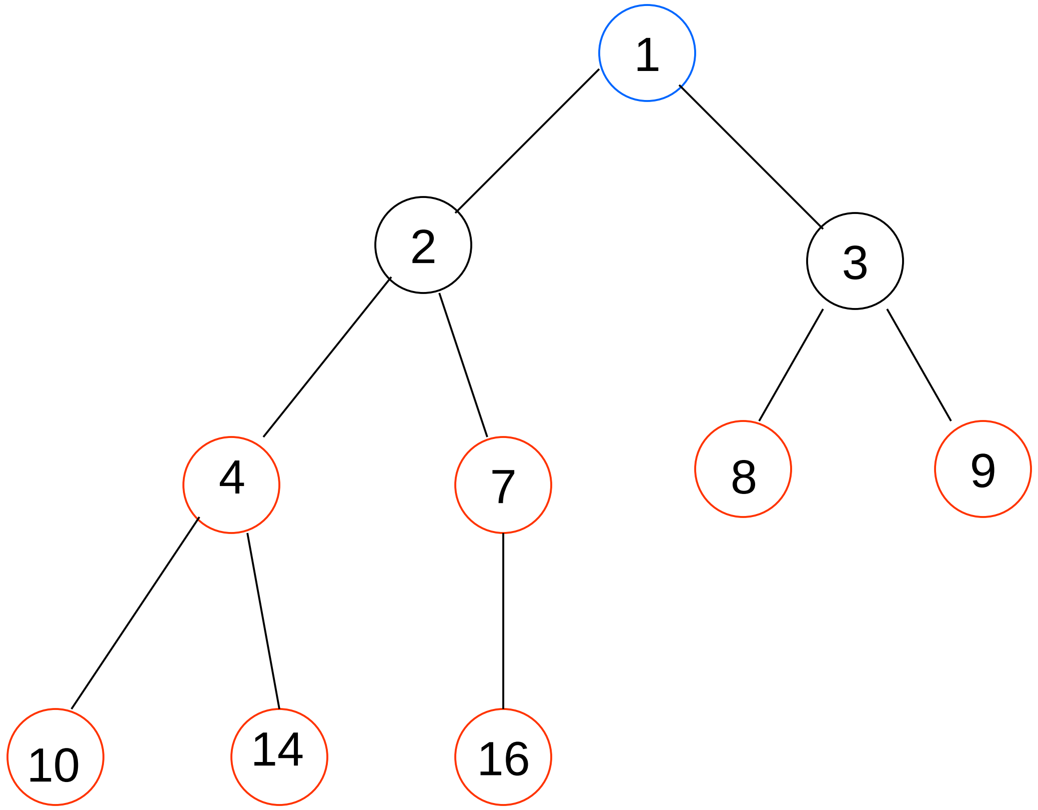

8
7
3
4
2
1
9
10
14
16

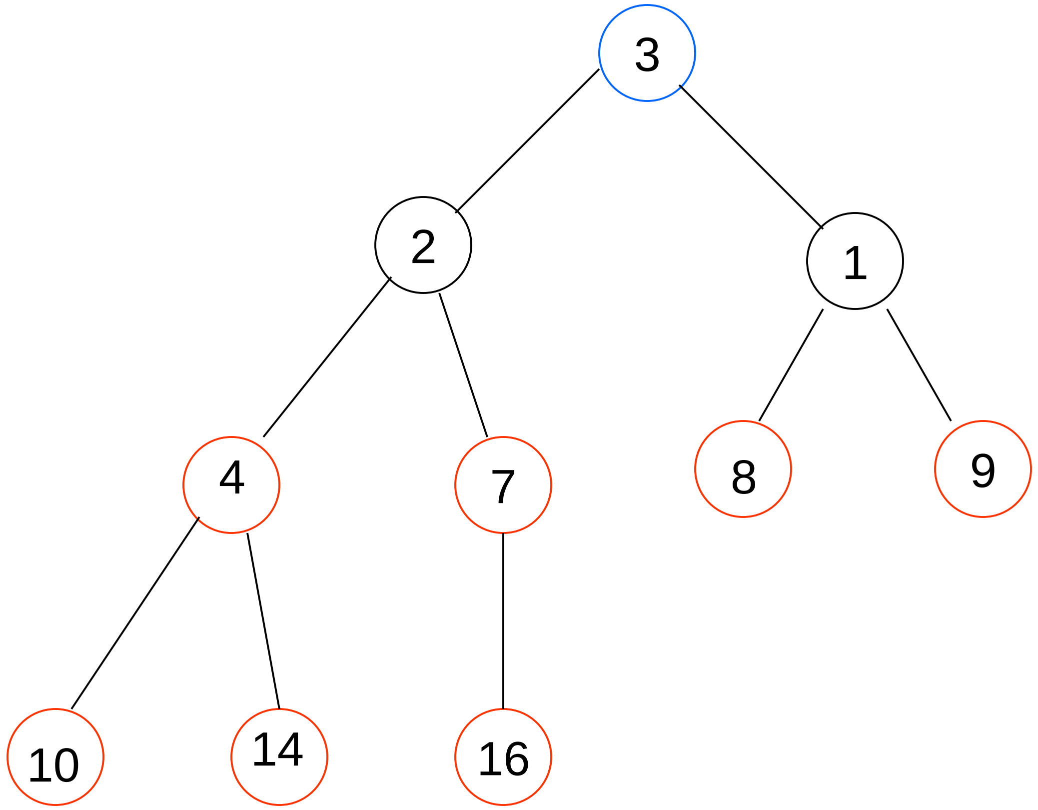

1
7
3
4
2
8
9
10
14
16

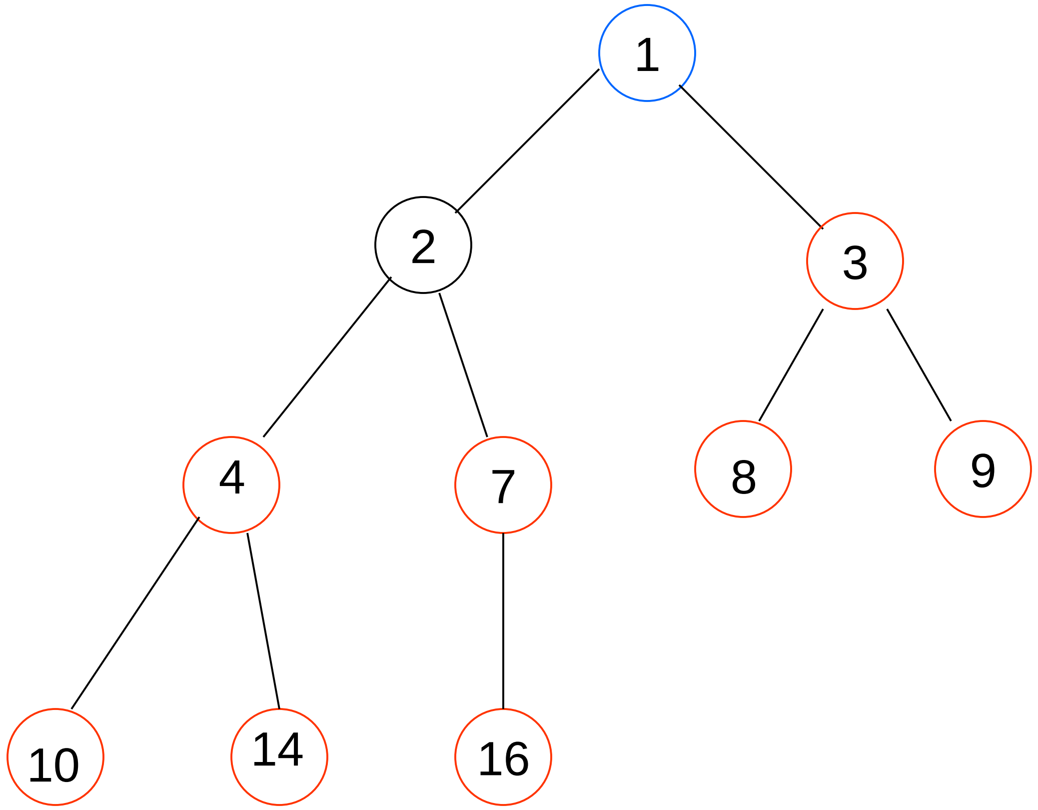

7
4
3
1
2
8
9
10
14
16

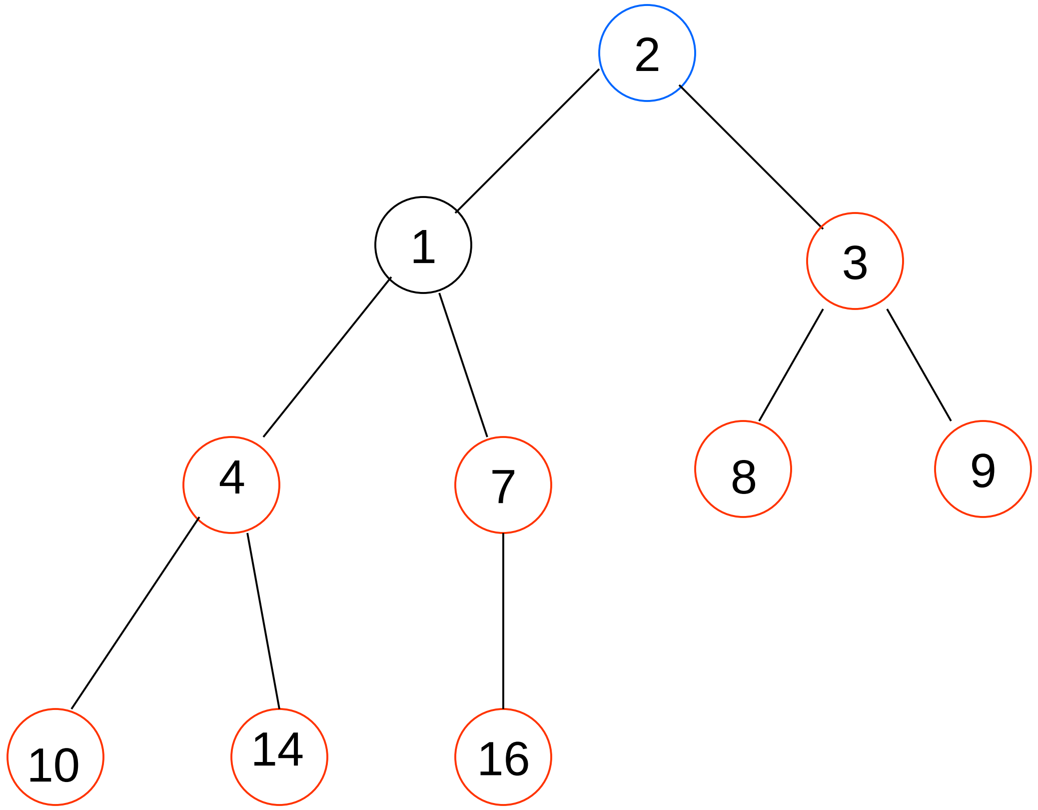

2
4
3
1
7
8
9
10
14
16

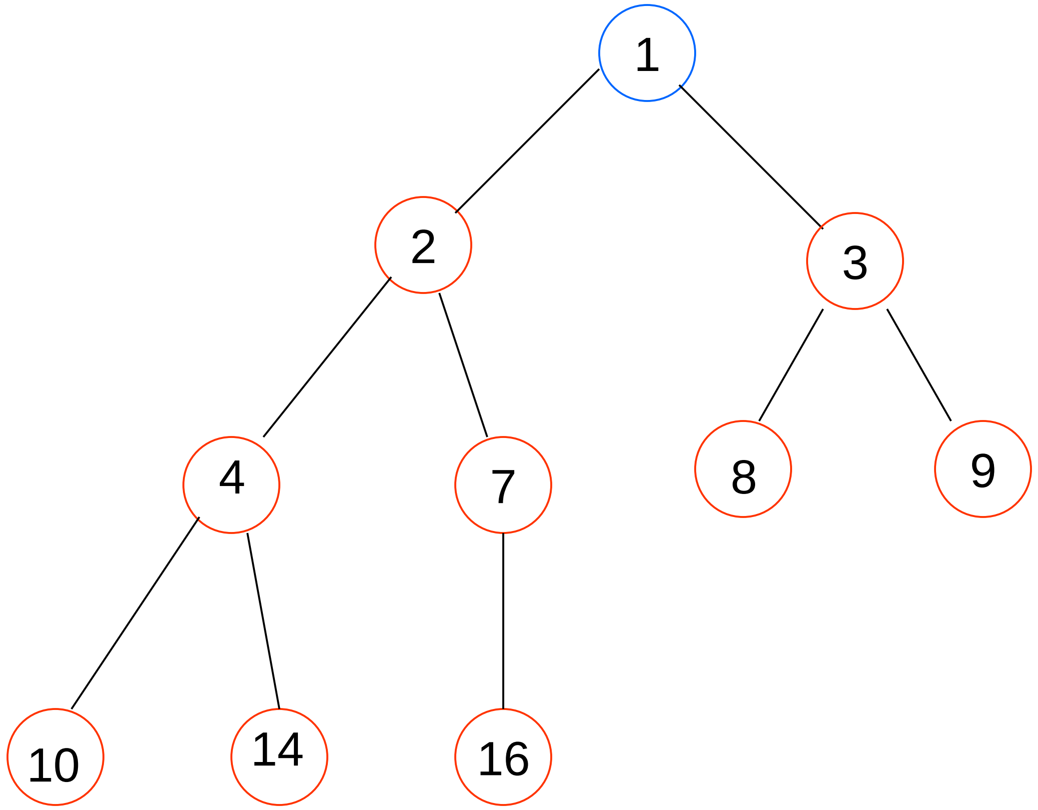

4
2
3
1
7
8
9
10
14
16


1
2
3
4
7
8
9
10
14
16


3
2
1
4
7
8
9
10
14
16


1
2
3
4
7
8
9
10
14
16


2
1
3
4
7
8
9
10
14
16



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

Extracting Max

Extract the maximum element from the heap:

```
Heap-Extract-Max(A,n)       // n size of heap
 if n < 1
    error "heap underflow"
 max = A[1]
 A[1] = A[n]
 n = n - 1                  // decrease size of heap
 Max-Heapify(A,1,n)
 return max
```

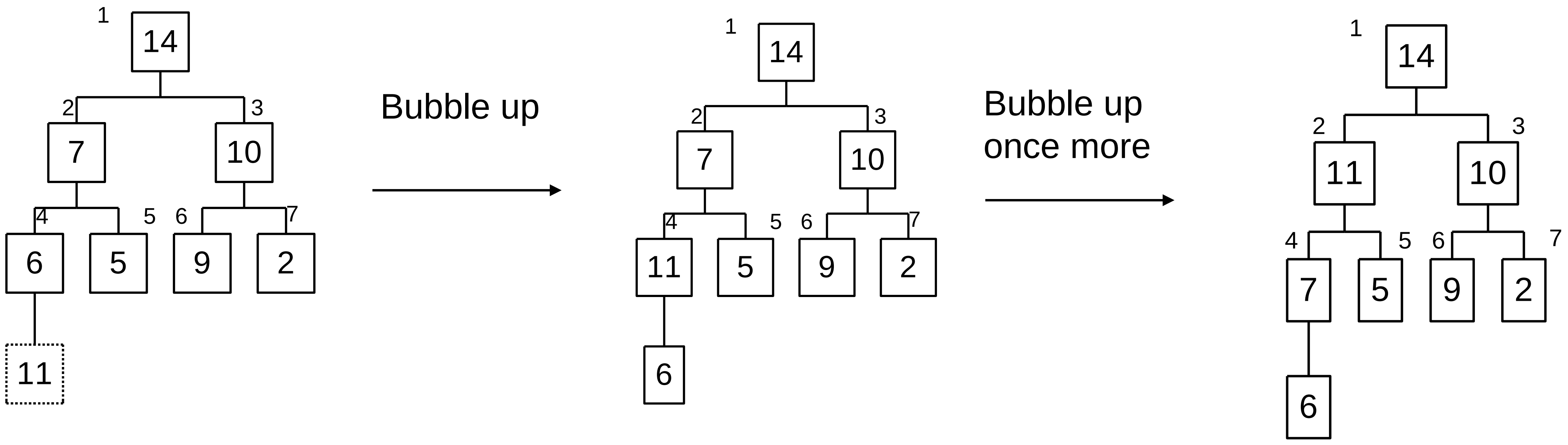
Heap-Insert (A,x)

Similar idea to heapify, put new element at end, bubble up to proper place toward root

```
Max-Heap-Insert(A,key)      // n size of heap
n <- n+1
i <- n
while i>1 and A[i/2] <key
   do A[i] <- A[i/2]
      i <- i/2
 A[i] <- key
```

Insert Example

Insert new element “11” starting at new node on bottom, i=8



Stop at this point, since parent (index 1, value 14) has a larger value

Useful Links

- Heaps, Part 1: Definition, Insertion, and Deletion https://www.youtube.com/watch?v=-6-xKgLOZPM