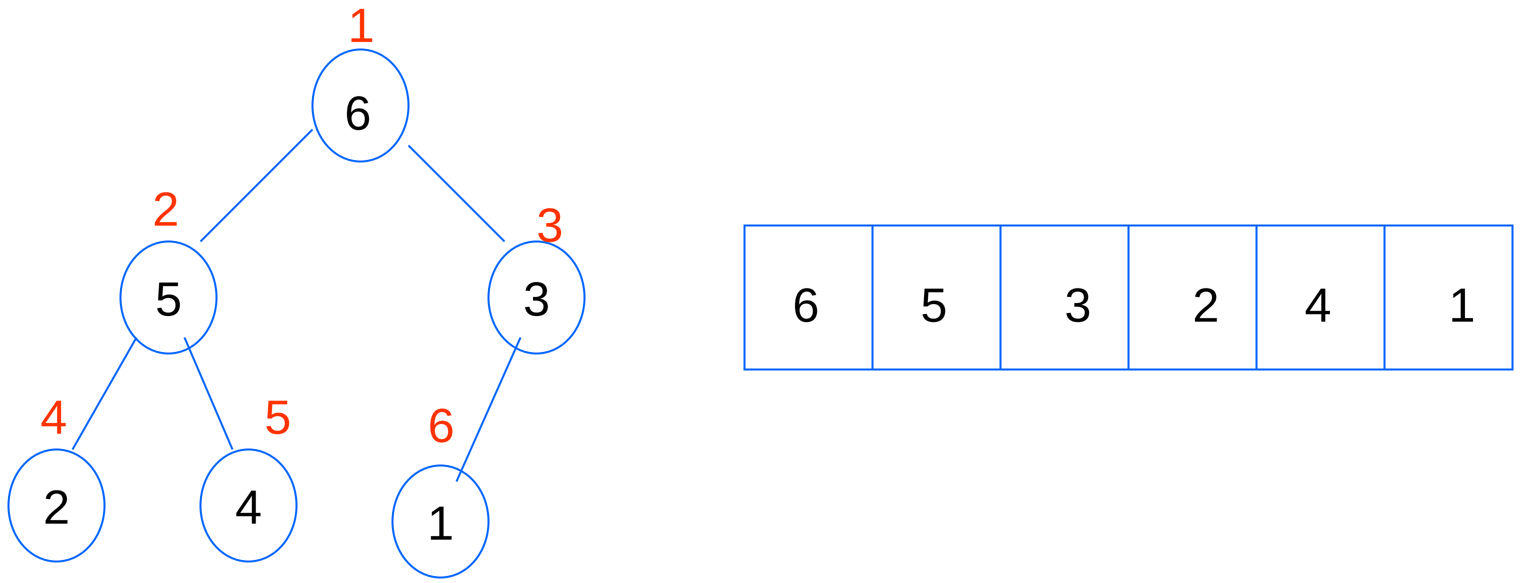
Trees and hierarchical orders, ordered trees, Search trees

Heap Tree

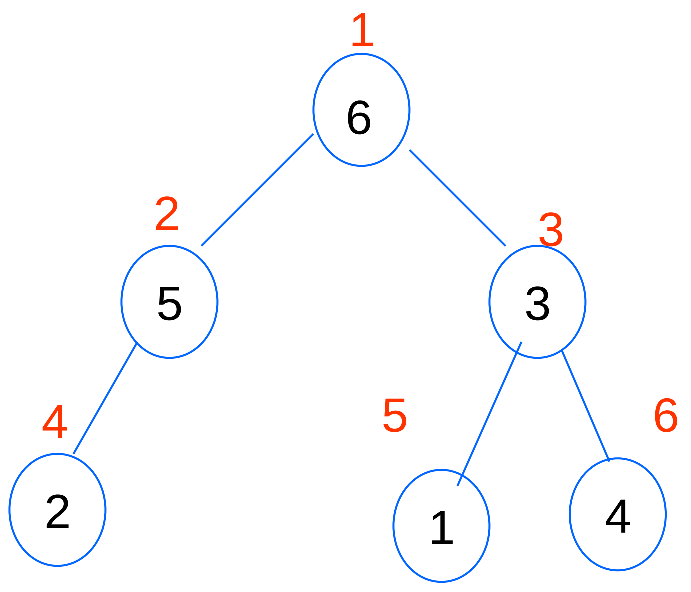
A Data Structure Heap

- A heap is a nearly complete binary tree which can be easily implemented on an array.

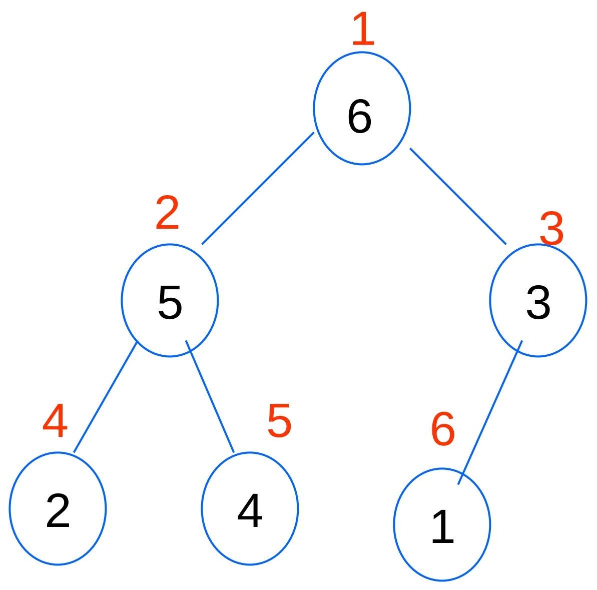


Nearly complete binary tree

- Every level except bottom is complete.
- On the bottom, nodes are placed as left as possible.



No!



Yes!

Concept of Heaps

- A heap (or heap-ordered tree) is a tree-based data structure that satisfies the heap property:
  - each node has a key value
  - if A has child B, then key(A) ≥ key(B).
- This implies that the root node has the highest key value in the heap. Such a heap is called a max-heap.
- min-heap: if A has a child B, then key(A) ≤ key(B)

Applications of Heaps

Heap-sort: insert elements into heap, find-min/find-min delete get one after another, return a sorted list of elements.

Used to implement priority queue. **Heap** Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm for shortest path

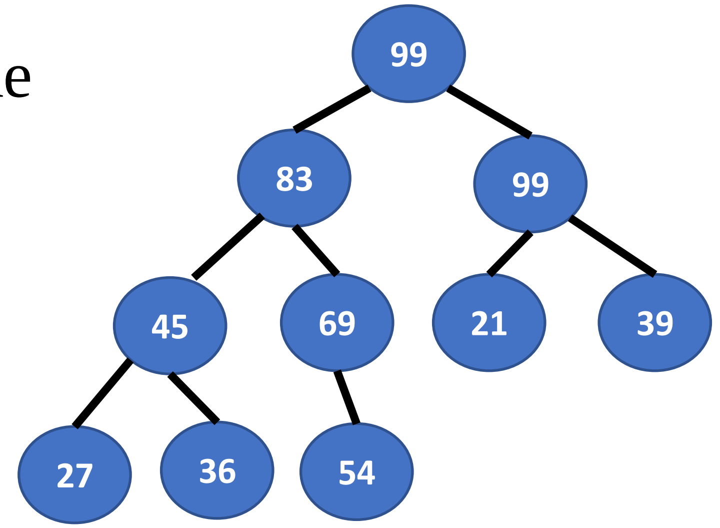
Two Special Heaps

Max-Heap

In a max-heap, every node i other than the root satisfies the following property:

$$A[\text{Parent}(i)] \geq A[i].$$

Binary-max-heap

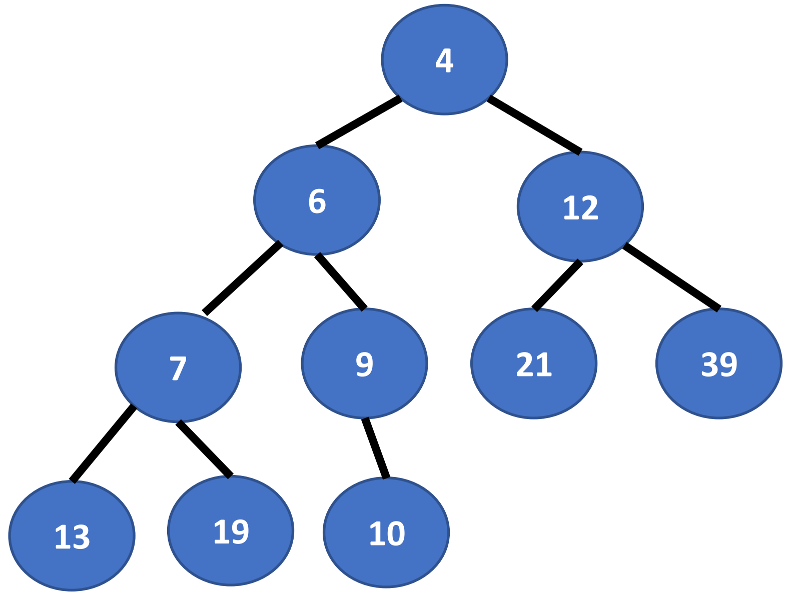


Min-Heap

In a min-heap, every node i other than the root satisfies the following property:

$$A[\text{Parent}(i)] \leq A[i].$$

Binary min-heap

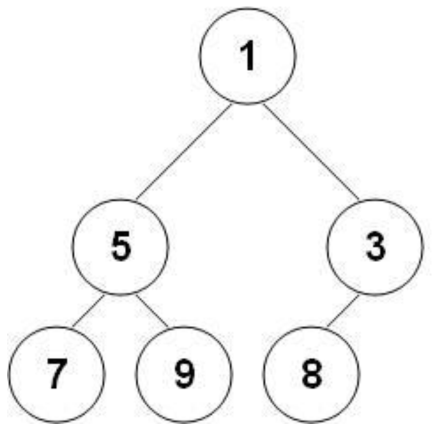


Algorithms associated with Heap

```
Parent($i$)
    return $\lfloor i/2 \rfloor$;

Left($i$)
    return $2i$;

Right ($i$)
    return $2i+1$;
```

Array Representation of Binary Heaps

- A complete binary tree of n element can be represented by an array of n elements
  - Order the complete binary tree node data elements in breadth-first and left-first order and put the elements into array in the same order. The root node has index **0**.
  - The node of index i has children indices at **$2i+1$** and **$2i+2$**, and parent at index **$(i-1)/2$**
- The height of the complete binary tree is $log_2 n$
- In case binary heap, the element at index 0 is the max/min element



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Insertion in a Binary Heap

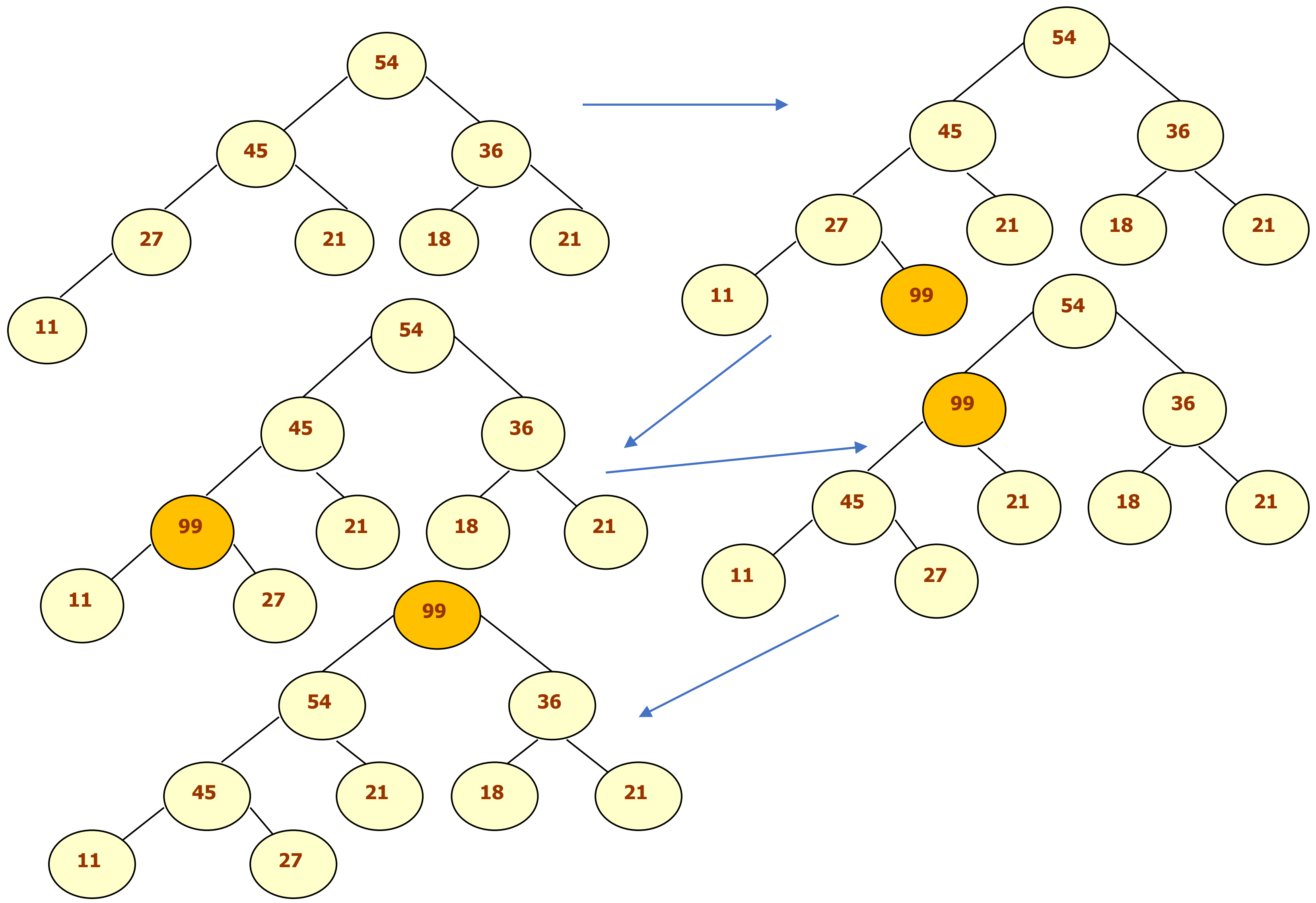
Algorithm: Insert element to binary max-heap H

```
Step 1.  add the new element node to the bottom left H

Step 2.  If the new node value is bigger than its parent, swap their
         values. Set the parent node as new node, go to step 2.

Step 3.  else exit.
```

Step 2 is called heapify

Example of Insertion in a Binary Heap

Consider the heap given below and insert 99 in it



Insertion in a Binary Heap

```
Algorithm to insert a value in the heap
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2:  SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS < 0
Step 4:   SET PAR = POS/2
Step 5:      IF HEAP[POS] <= HEAP[PAR], then
                Goto Step 6.
            ELSE
                SWAP HEAP[POS], HEAP[PAR]
                POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: Return
```

Time complexity: ? Space complexity: ?

Insertion in a Binary Heap (Pseudo Code)

```
void insert(int array[], int newNum)
{
 if (size == 0)
 {
  array[0] = newNum;
  size += 1;
 }
 else
 {
  array[size] = newNum;
  size += 1;
  for (int i = size / 2 - 1; i >= 0; i--)
  {
   heapify(array, size, i);
  }
 }
}
```

```
void heapify(int array[], int size, int i)
{
 if (size == 1)
 {
        printf("Single element in the heap");
 }
 else
 {
        int largest = i;
         int l = 2 * i + 1;
         int r = 2 * i + 2;
         if (l < size && array[l] > array[largest])
                largest = l;
        if (r < size && array[r] > array[largest])
                largest = r;
        if (largest != i)
        {
                swap(&array[i], &array[largest]);
                heapify(array, size, largest);
        }
 } }
```

Deletion in a Binary Heap

- An element is always deleted from the root of the heap. So, deleting an element from the heap is done in two major steps.

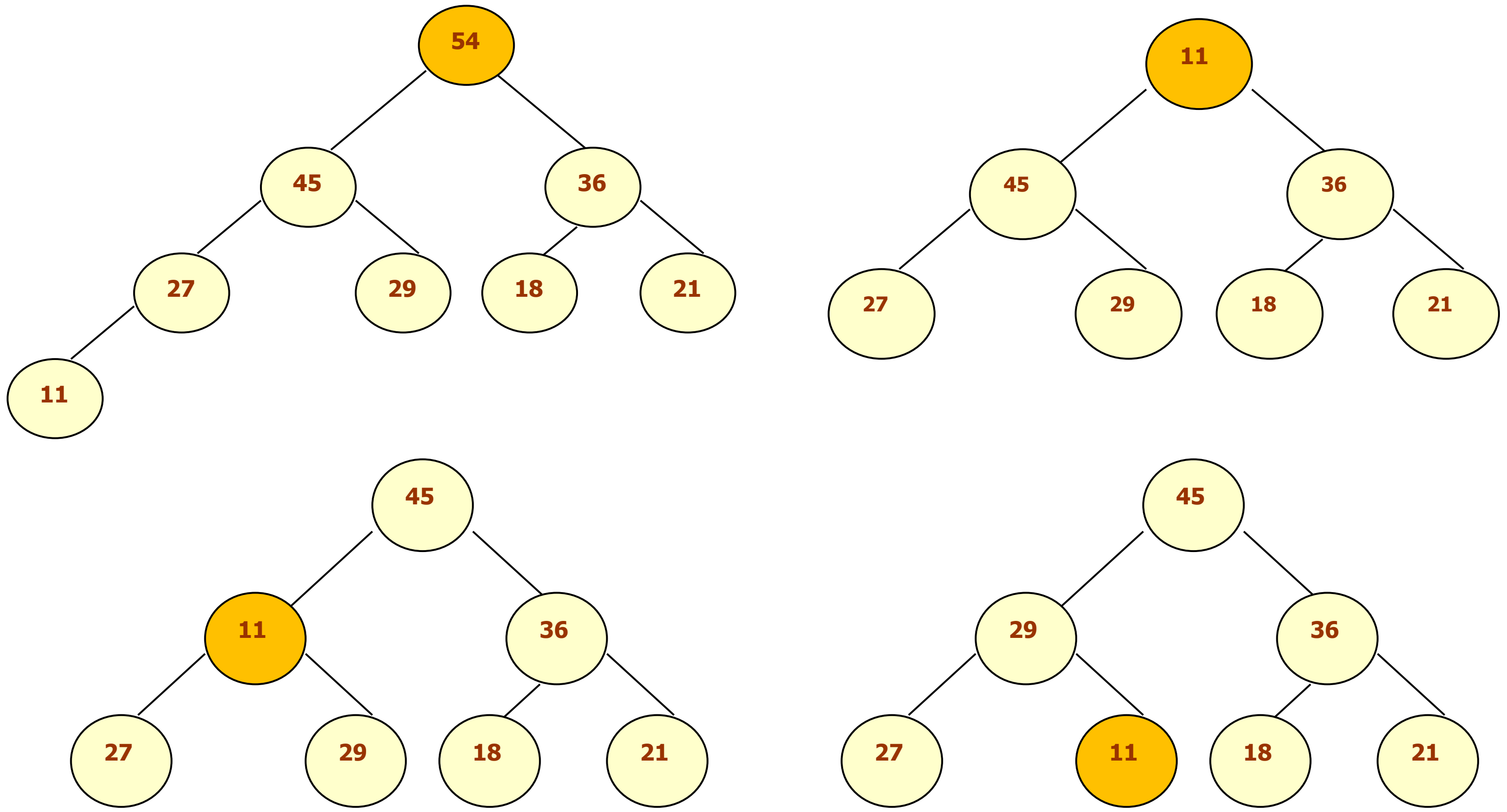
Step 1. **Replace the root node’s value with the last node’s value so that H is still a complete binary tree but not necessarily a heap.**

Step 2. **Delete the last node**.

Step 3. Sink down the new root node’s value so that H satisfies the heap property. In this step, interchange the root node’s value with its child node’s value (whichever is largest among its children).

Deletion in a Binary Heap

- Consider the heap H given below and delete the root node's value.



Deletion in a Binary Heap

Algorithm to delete the root element from the heap

```
DELETE_HEAP(HEAP, N, VAL)
Step 1: [ Remove the last node from the heap]
         SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
         SET PTR = 0, LEFT = 1, RIGHT = 2
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP{PTR] >= HEAP[LEFT] AND HEAP[PTR] >= HEAP[RIGHT],
             then Go to Step 8
        [END OF IF]
Step 6:  IF HEAP[RIGHT] <= HEAP[LEFT], then
            SWAP HEAP[PTR], HEAP[LEFT]
            SET PTR = LEFT
         ELSE
            SWAP HEAP[PTR], HEAP[RIGHT]
             SET PTR = RIGHT
         [END OF IF]
Step 7:  SET LEFT = 2 * PTR and RIGHT = LEFT + 1
         [END OF LOOP]
Step 8: Return
```

Time complexity: ? **Space complexity: ?**

Deletion in a Binary Heap (Pseudo Code)

```
void deleteRoot(int array[], int num)
{
  int i;
  for (i = 0; i < size; i++)
  {
    if (num == array[i])
      break;
  }

  swap(&array[i], &array[size - 1]);
  size -= 1;
  for (int i = size / 2 - 1; i >= 0; i--)
  {
    heapify(array, size, i);
  }
}
```

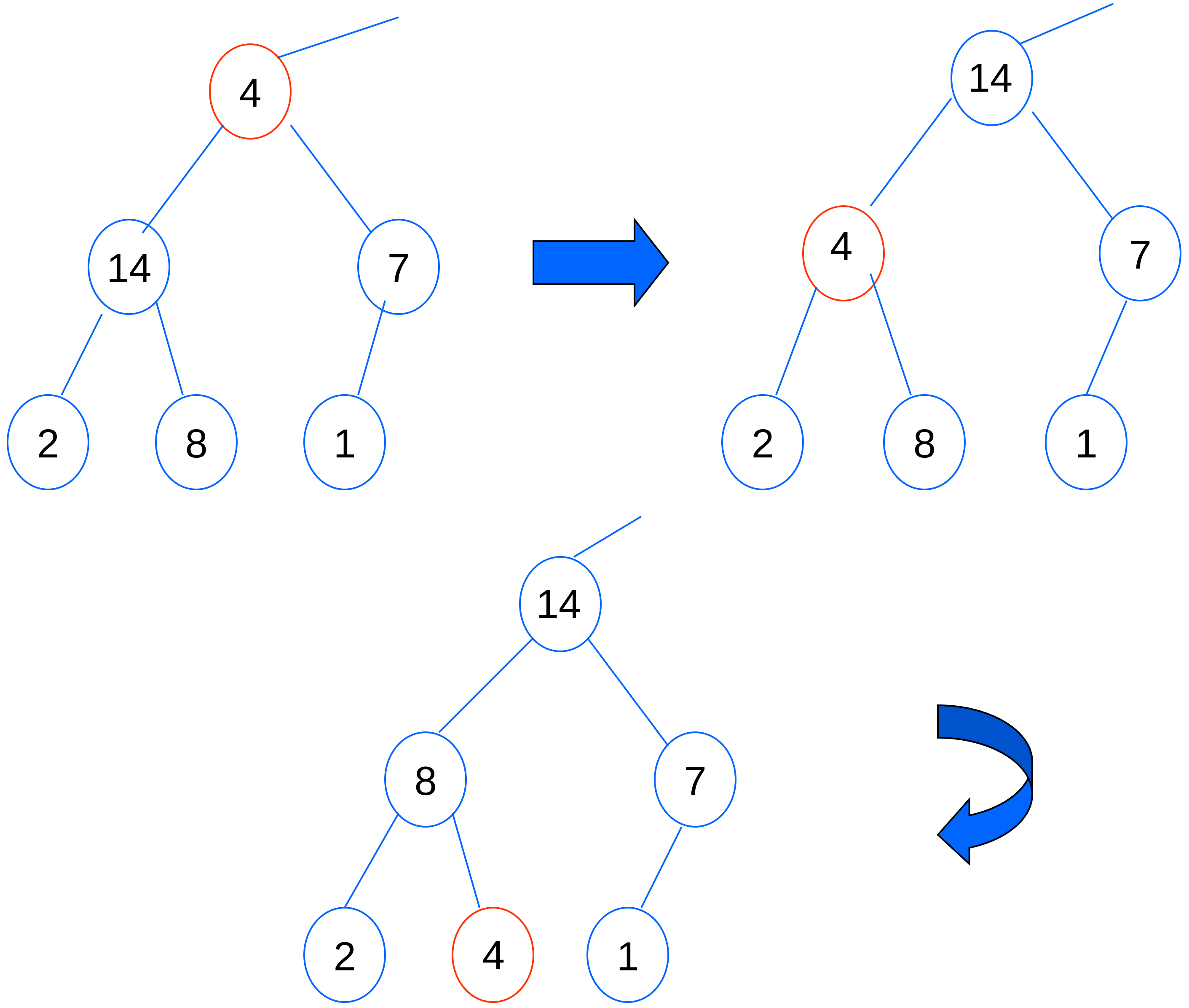
Commonly used term

Steps associated with Max-Heap

- Three algorithms associated with heap.
- In addition, it is associated with two more algorithms: Max-Heapify and Build-Max-Heap.

Max-Heapify

- Max-Heapify(A,i) is a subroutine.
- When it is called, two subtrees rooted at Left(i) and Right(i) are max-heaps, but A[i] may not satisfy the max-heap property.
- Max-Heapify(A,i) makes the subtree rooted at A[i] become a max-heap by letting A[i] “float down”.


4
14
7
2
8
1
14
4
7
2
8
1
14
8
7
2
4
1

Algorithm / Steps

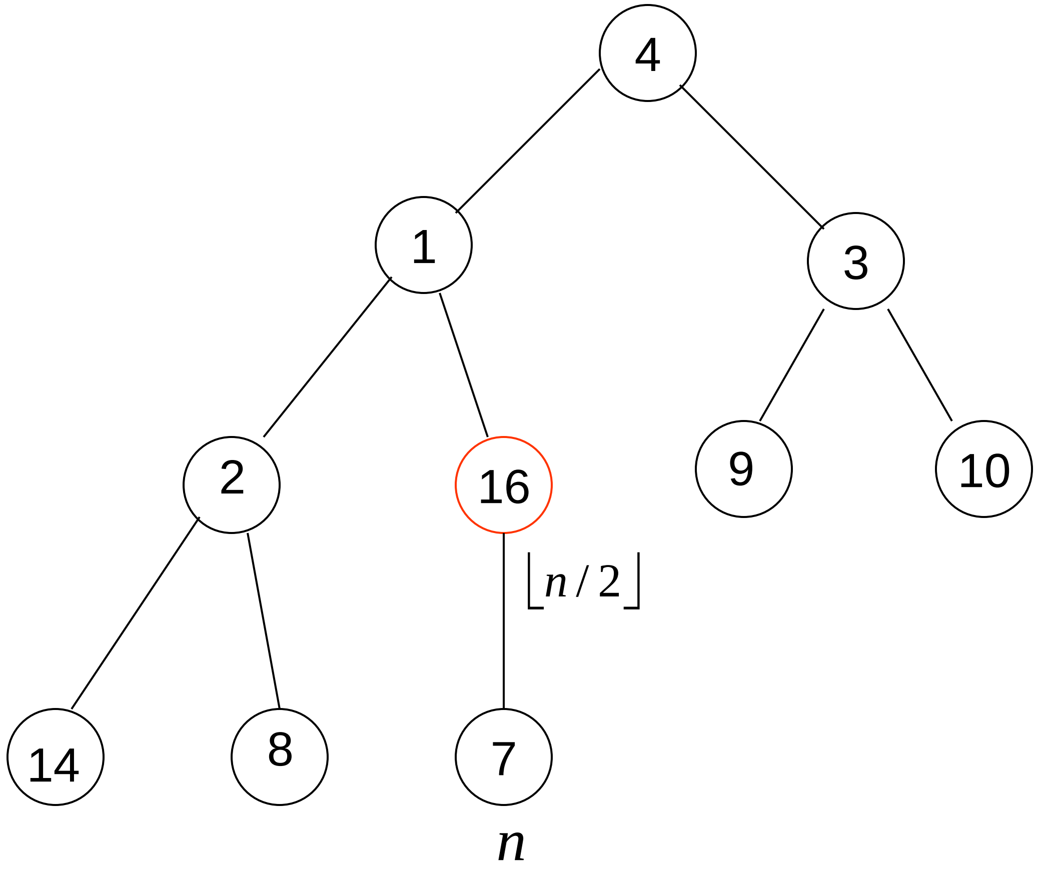
```
$\text{Max-Heapify}(A, i)$
$l \leftarrow \text{Left}(i);$
$r \leftarrow \text{Right}(i);$
if $l \leq heap-size[A]$ and $A[l] > A[i]$
    then $largest \leftarrow l$
    else $largest \leftarrow i;$
if $r \leq heap-size[A]$ and $A[r] > A[largest]$
    then $largest \leftarrow r;$
if $largest \neq i$
    then begin exchange $A[i] \leftrightarrow A[largest];$
        $\text{Max-Heapify}(A, largest);$
    end-if
```

Building a Max-Heap

```
$\text{Build-Max-Heap}(A)$
$heap\text{-}size[A] \leftarrow length[A];$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    do $\text{Max-Heapify}(A, i);$
```

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

The last location who has a child is $\lfloor n/2 \rfloor$.

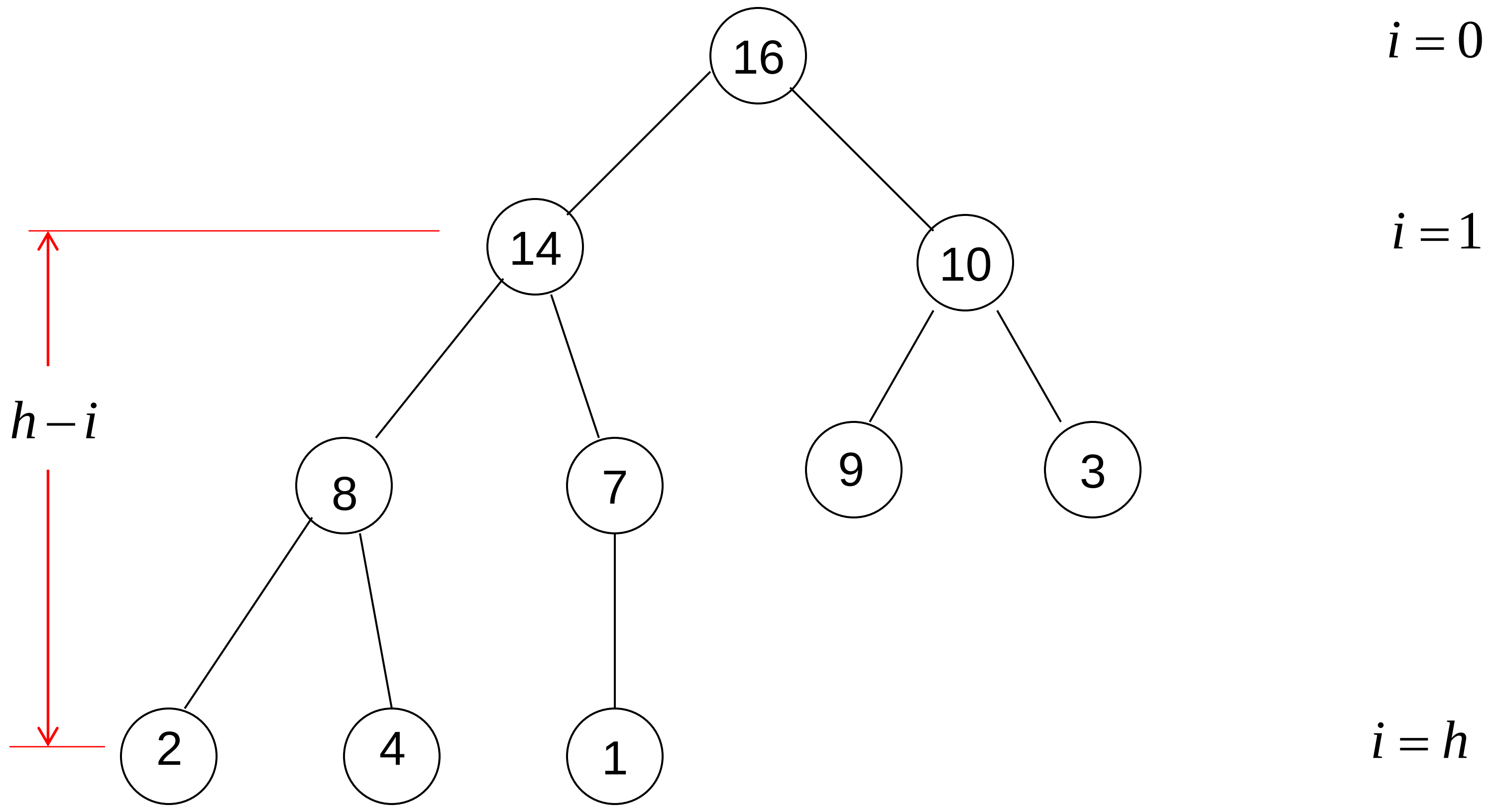


Proof. It is parent of location n

Building a Max-Heap

```
$\text{Build-Max-Heap}(A)$
$heap\text{-}size[A] \leftarrow length[A];$
for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
    do $\text{Max-Heapify}(A, i);$
```

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

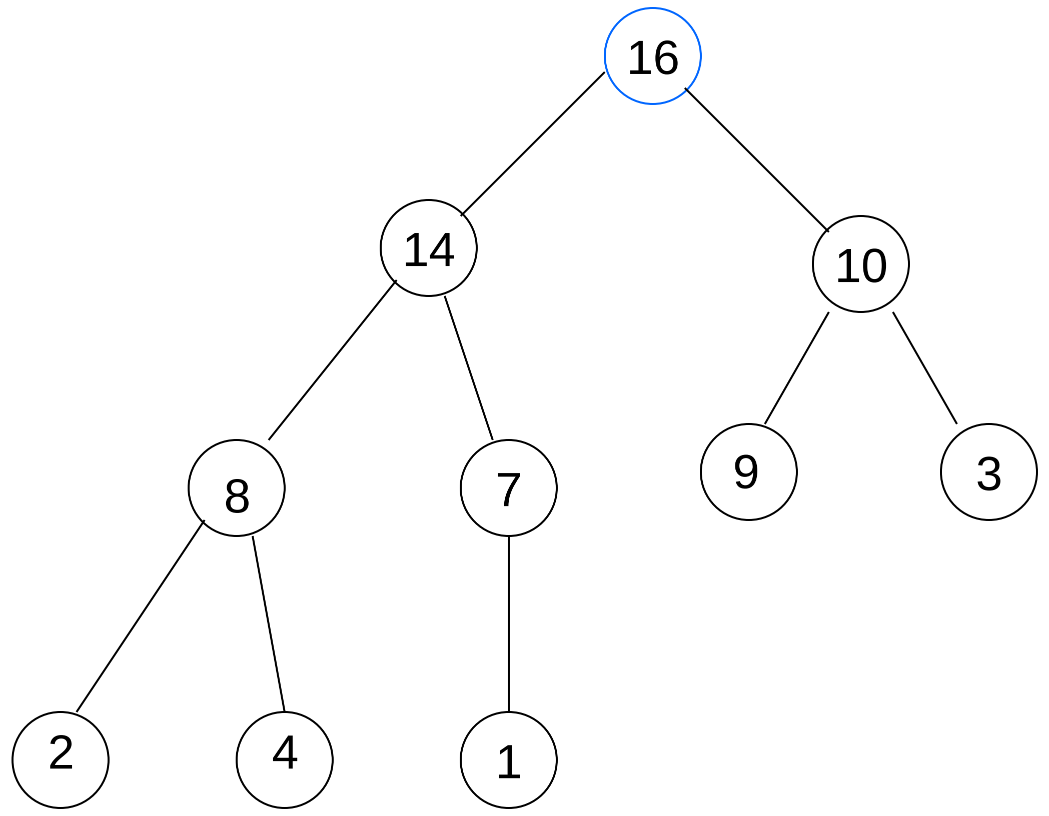

16
14
10
8
7
9
3
2
4
1
$h-i$
$i=0$
$i=1$
$i=h$

Heapsort

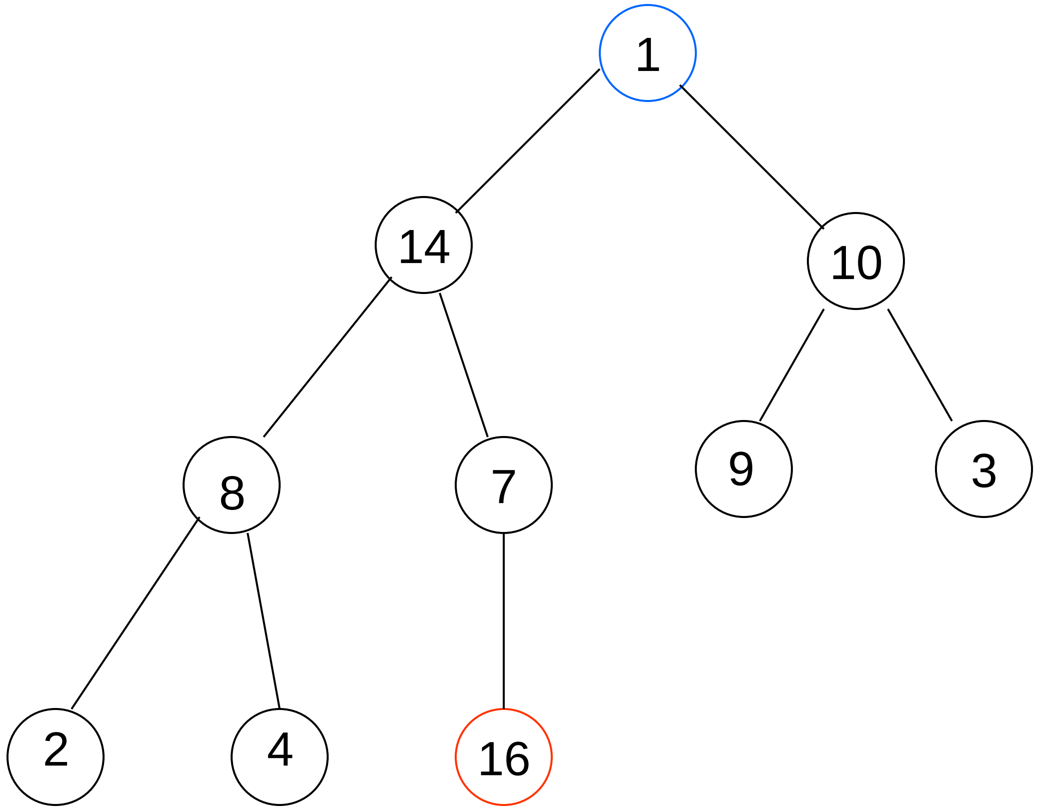
```
$\text{Heapsort}(A)$
  $\text{Buid-Max-Heap}(A);$
  $\text{for } i \leftarrow length[A] \text{ downto } 2$
    $\text{do begin}$
        $\text{exchange } A[1] \leftrightarrow A[i];$
        $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1;$
        $\text{Max-Heapify}(A, 1);$
    $\text{end-for}$
```

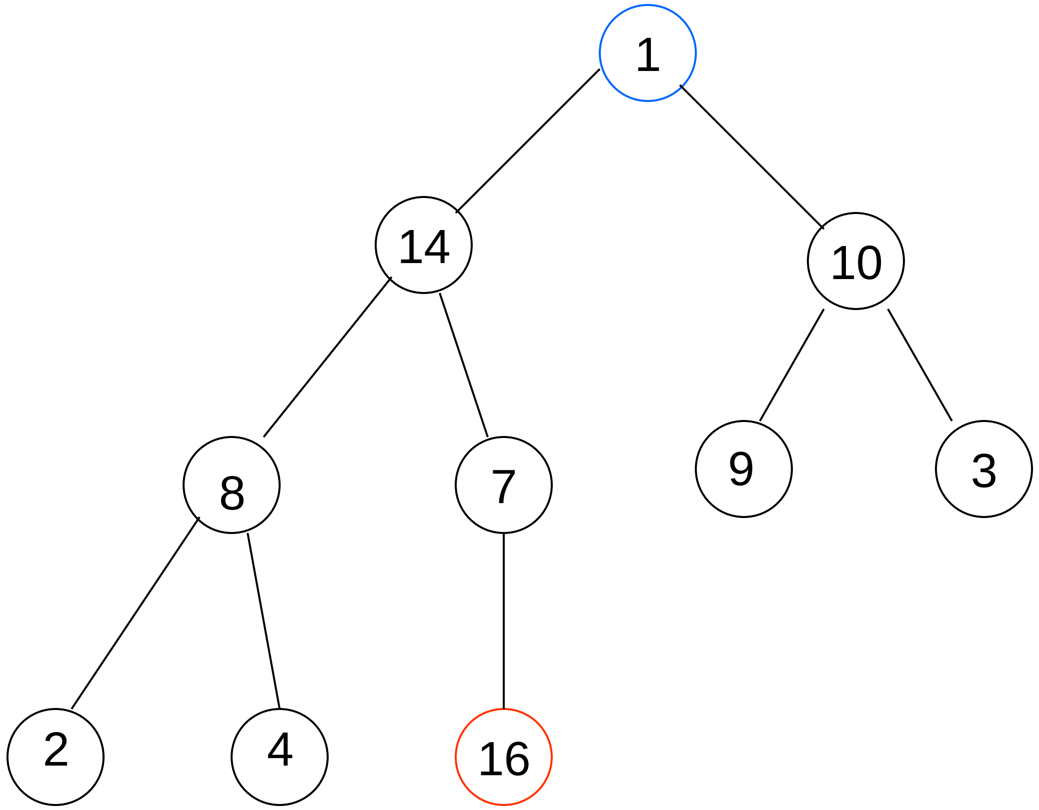
Input: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

Build a max-heap

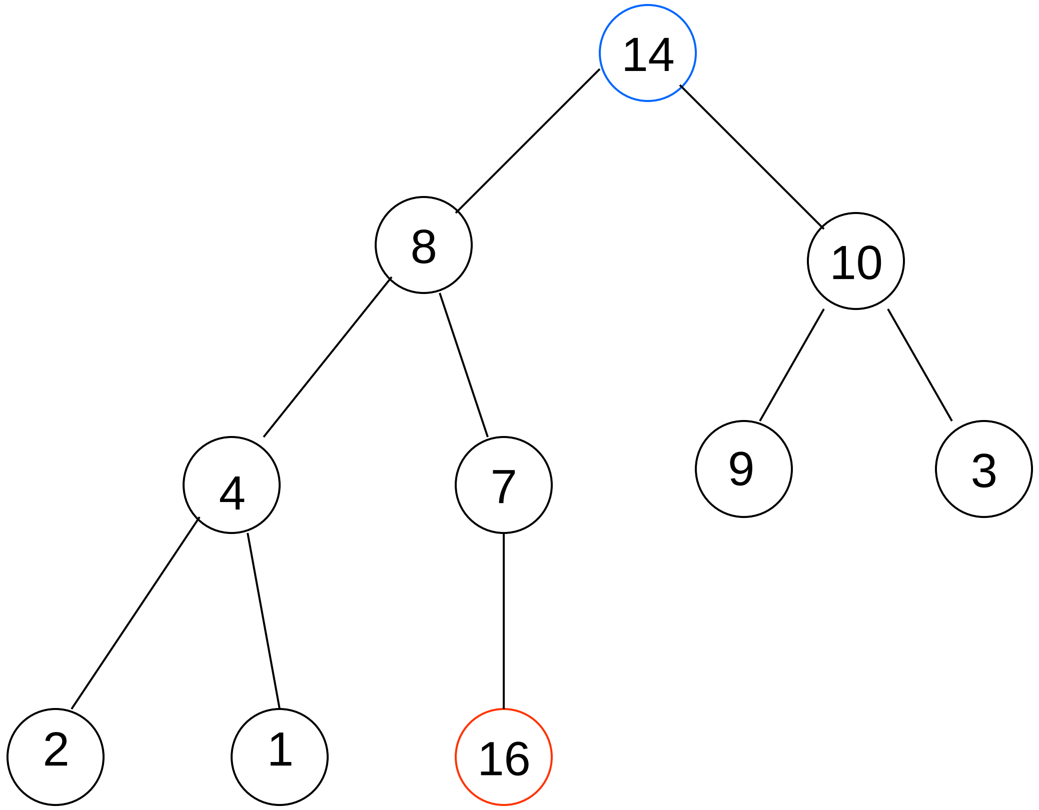


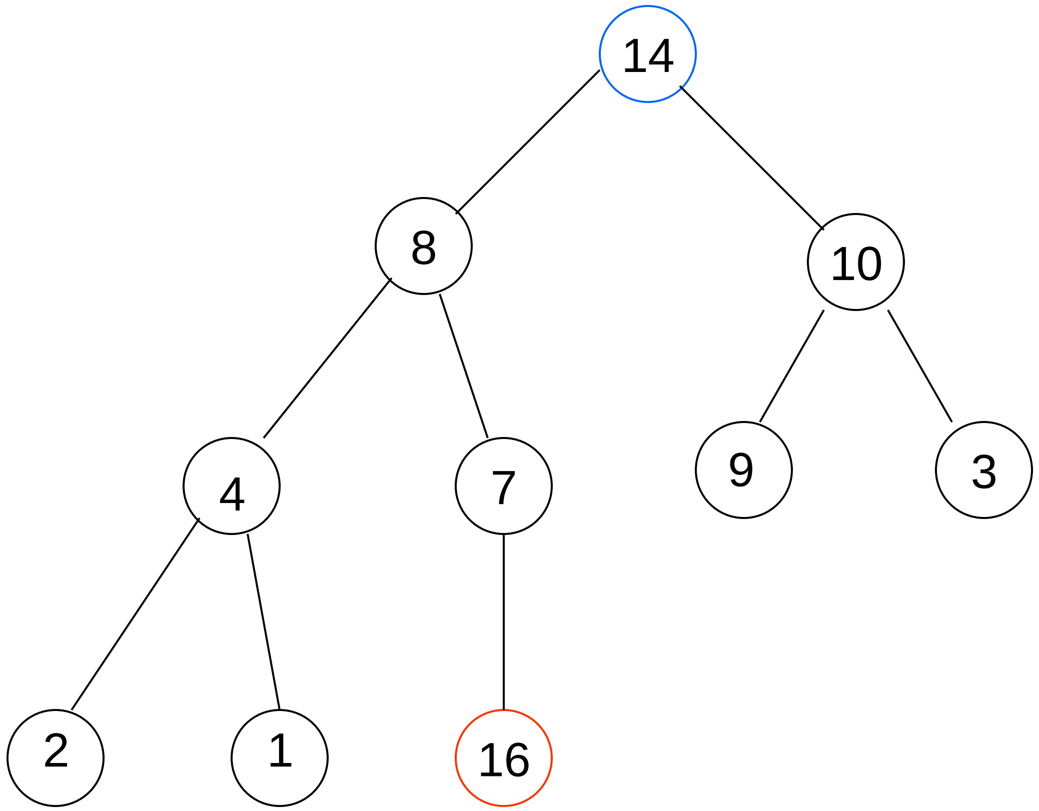
16, 14, 10, 8, 7, 9, 3, 2, 4, 1.


1
14
10
8
7
9
3
2
4
16

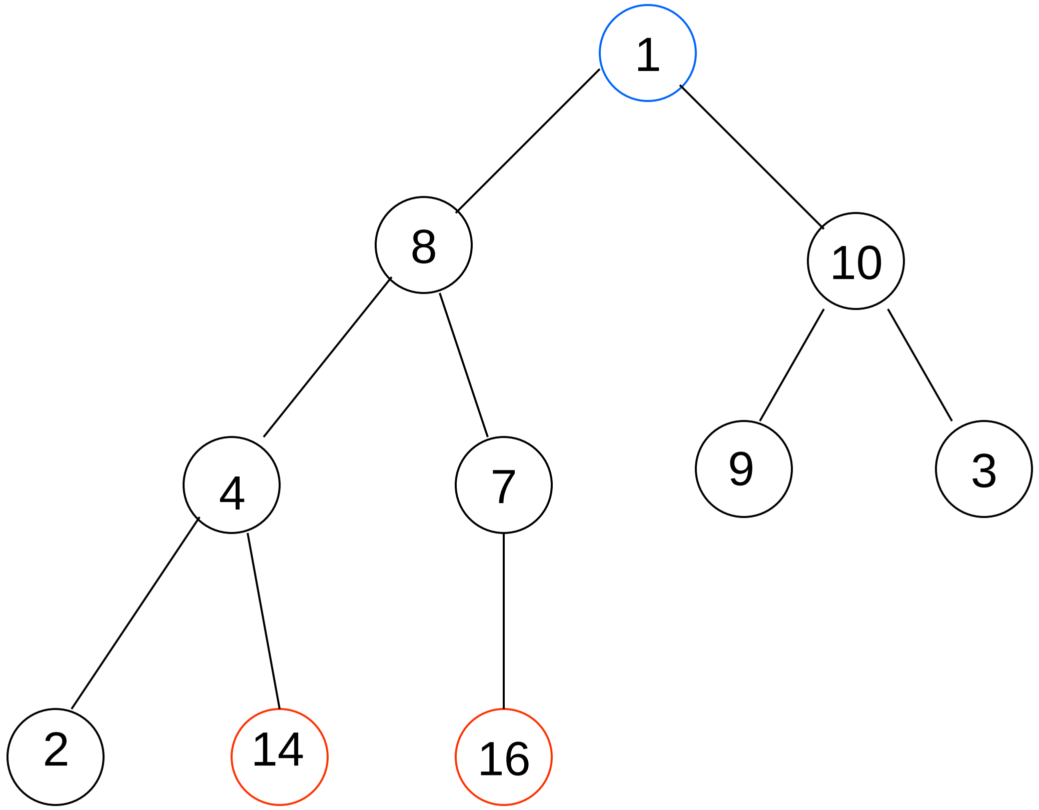


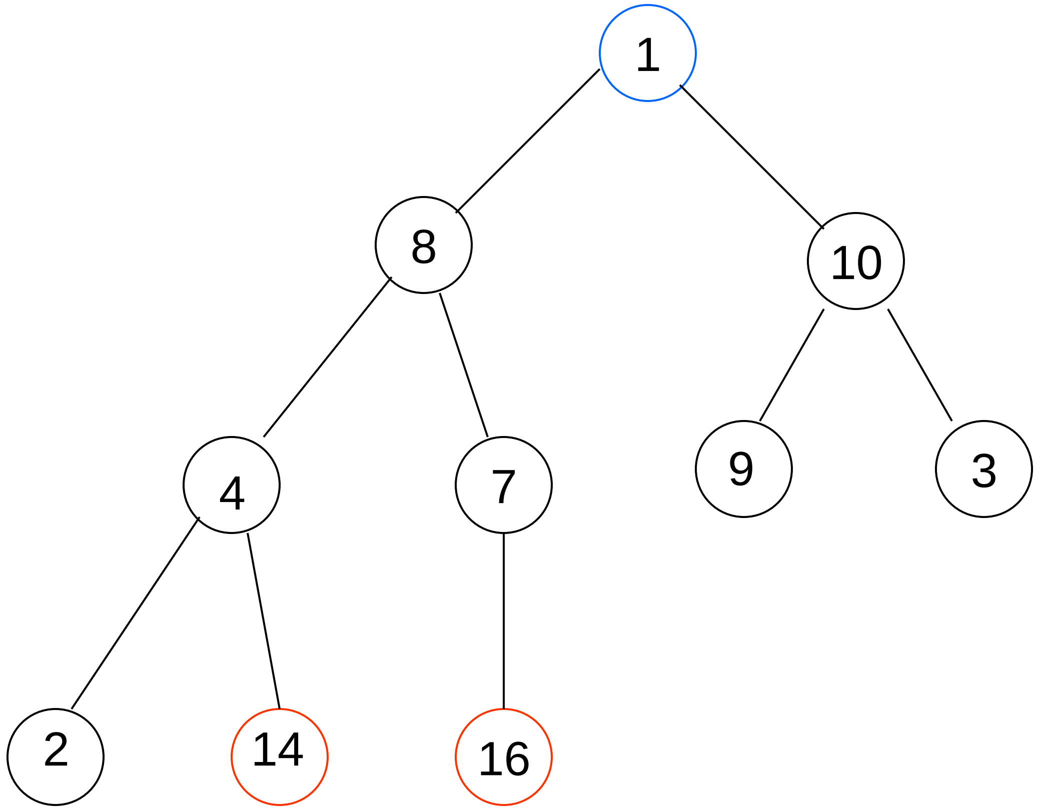
1, 14, 10, 8, 7, 9, 3, 2, 4, 16.


14
8
10
4
7
9
3
2
1
16

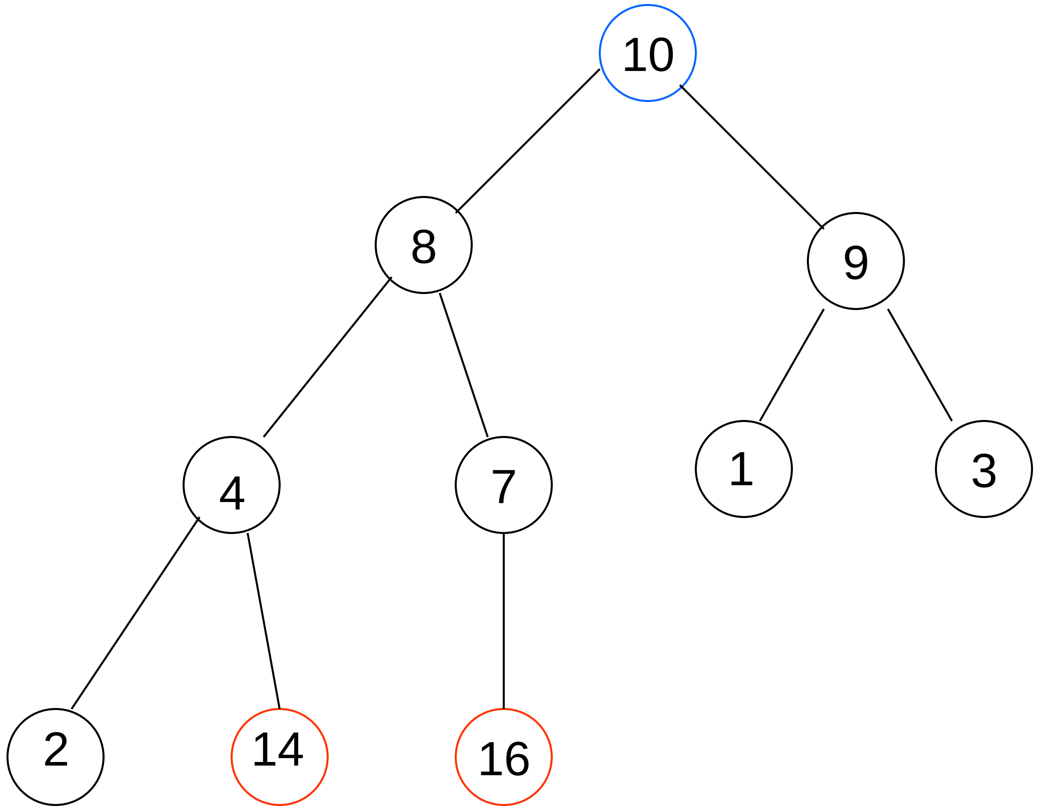


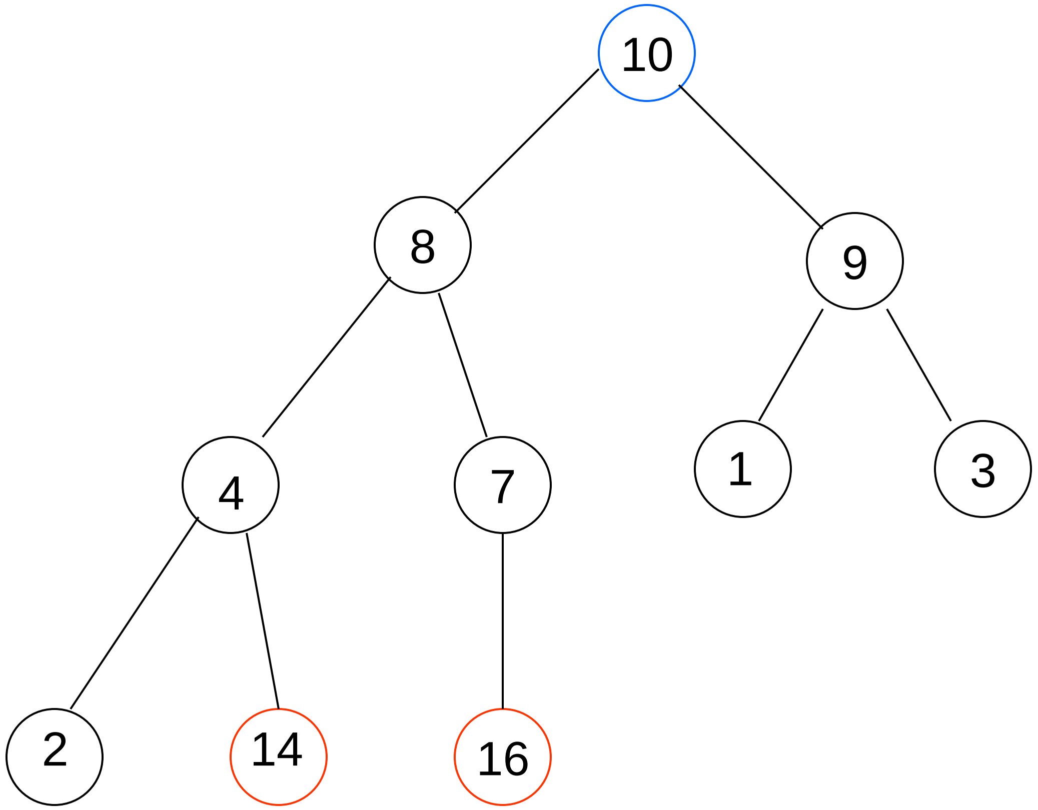
14, 8, 10, 4, 7, 9, 3, 2, 1, 16.


1
8
10
4
7
9
3
2
14
16

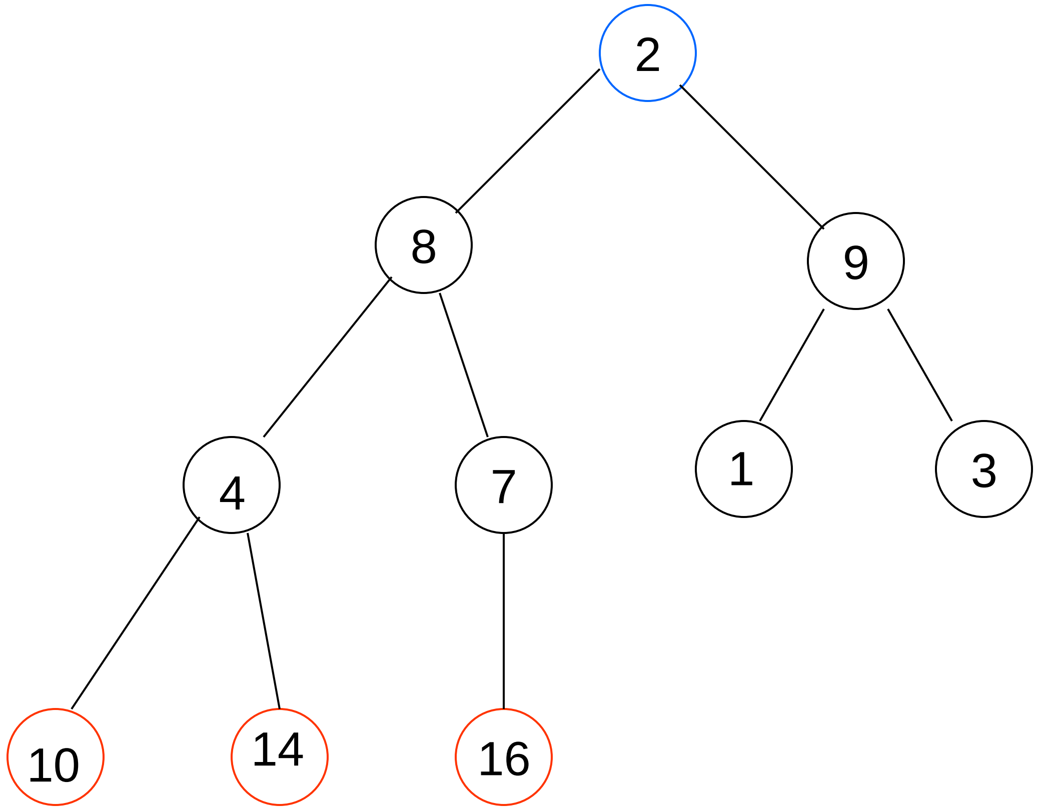


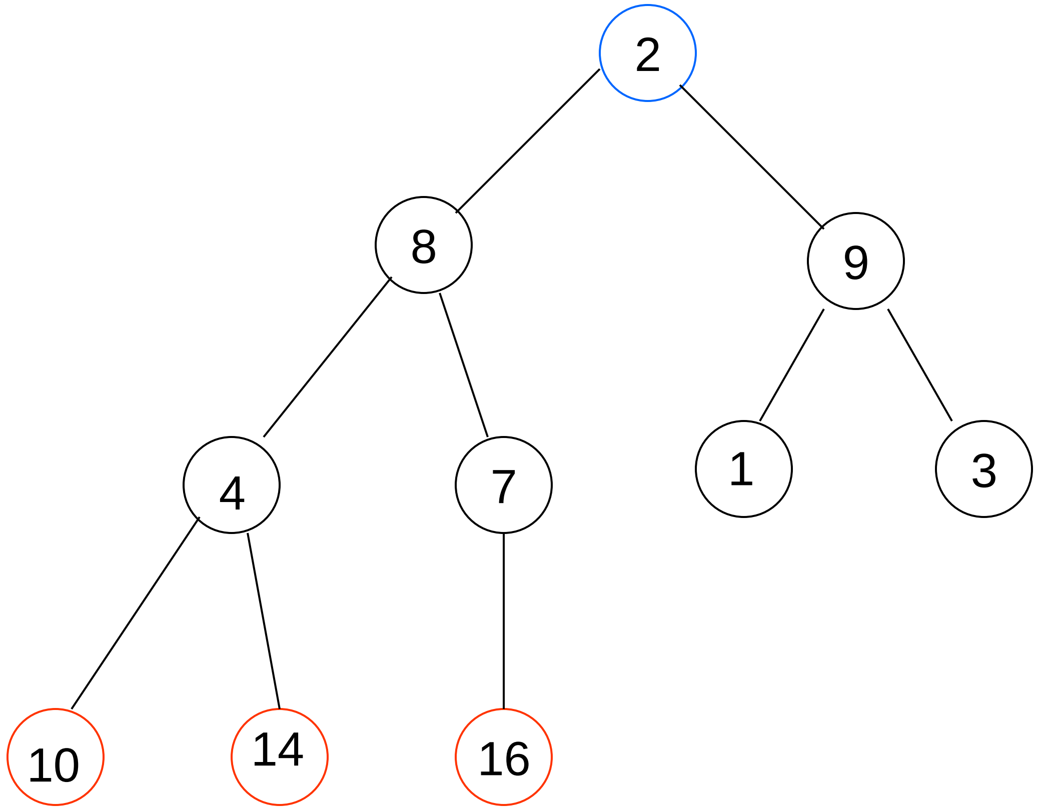
1, 8, 10, 4, 7, 9, 3, 2, 14, 16.


10
8
9
4
7
1
3
2
14
16

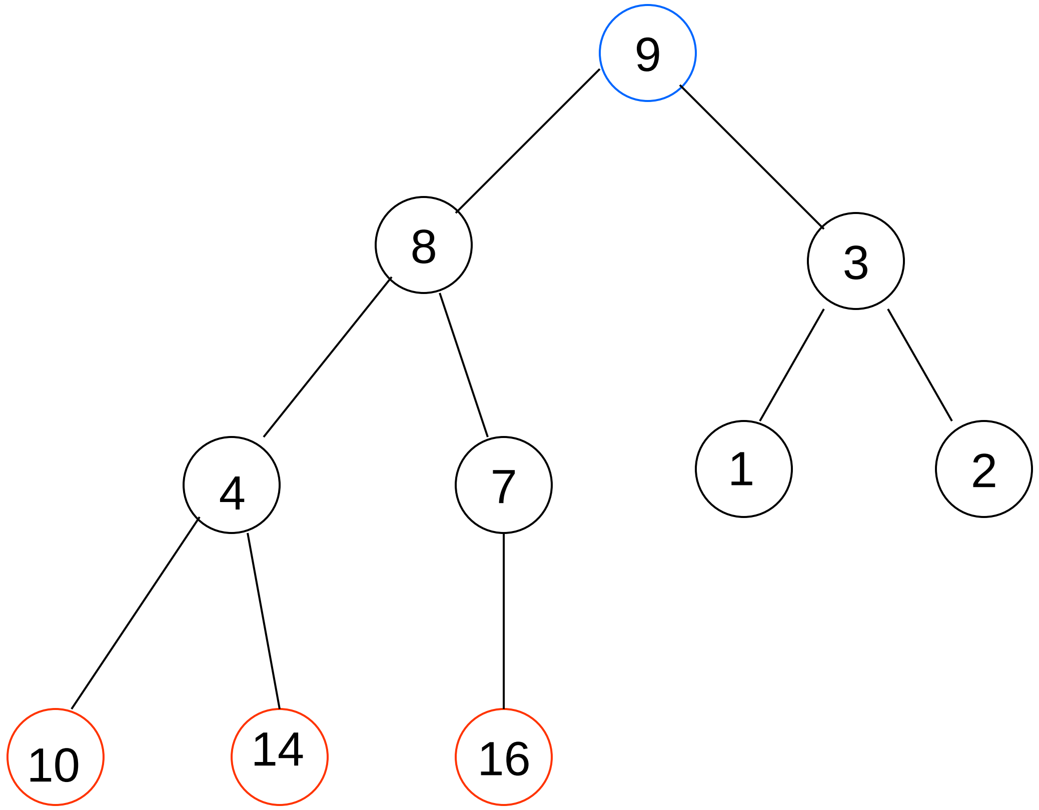


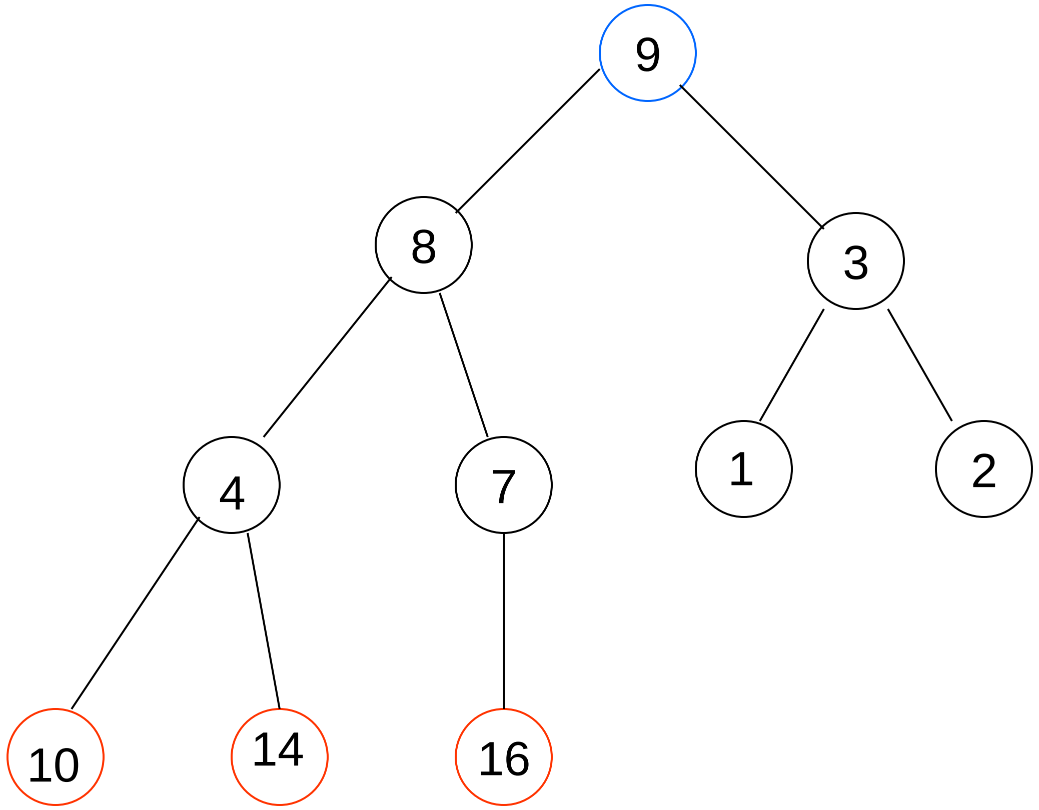
10, 8, 9, 4, 7, 1, 3, 2, 14, 16.


2
8
9
4
7
1
3
10
14
16

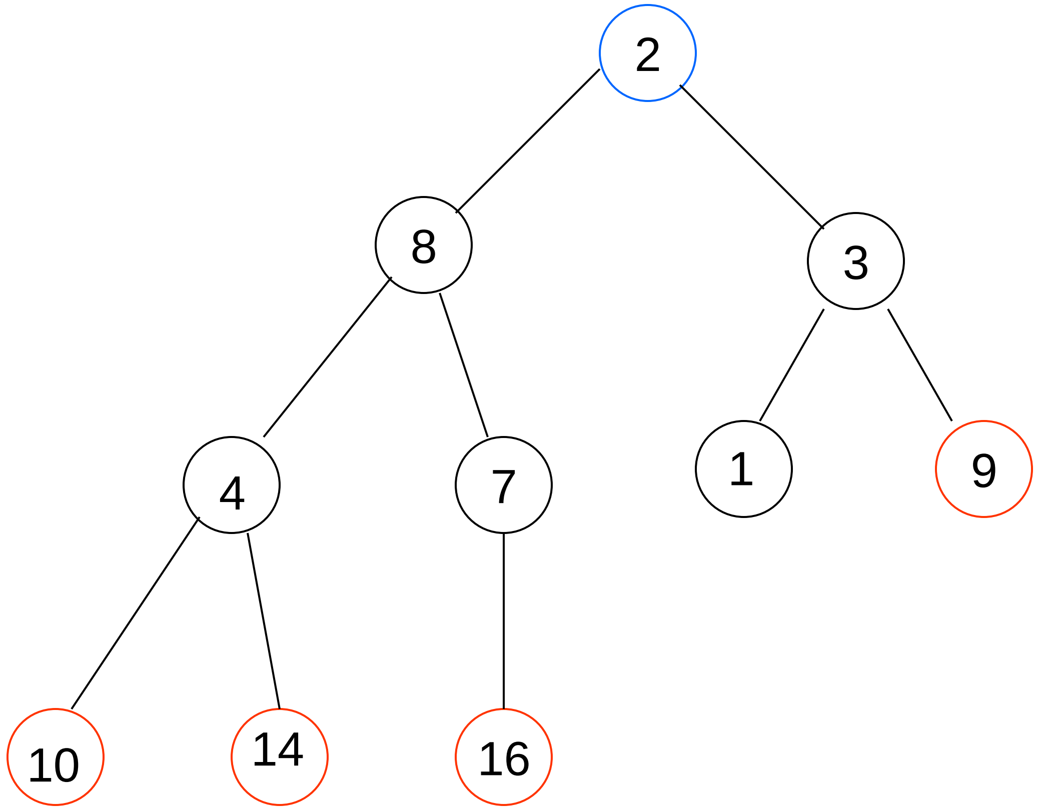


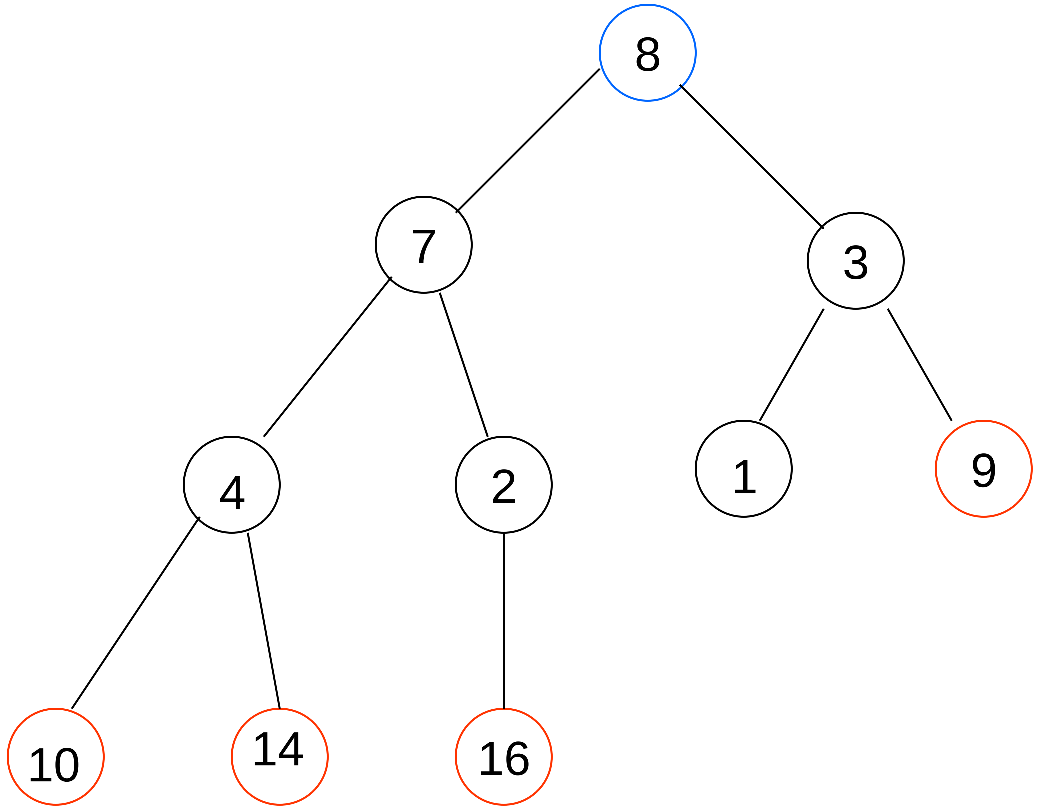
2, 8, 9, 4, 7, 1, 3, 10, 14, 16.

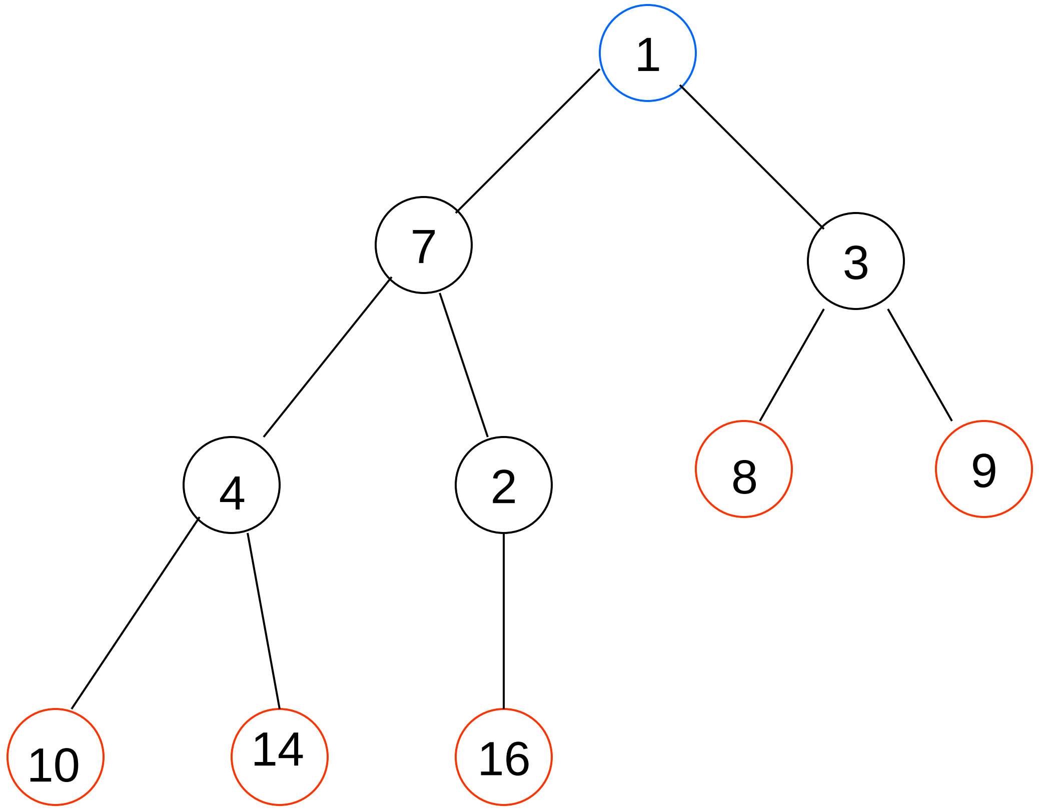

9
8
3
4
7
1
2
10
14
16

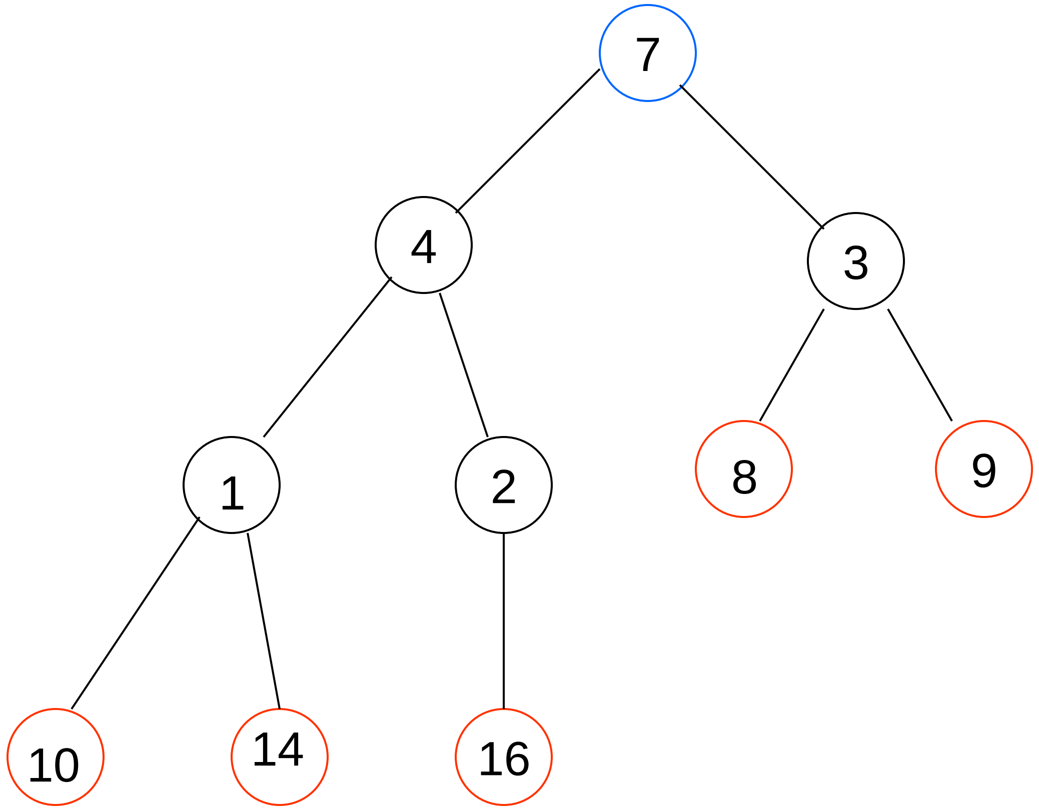


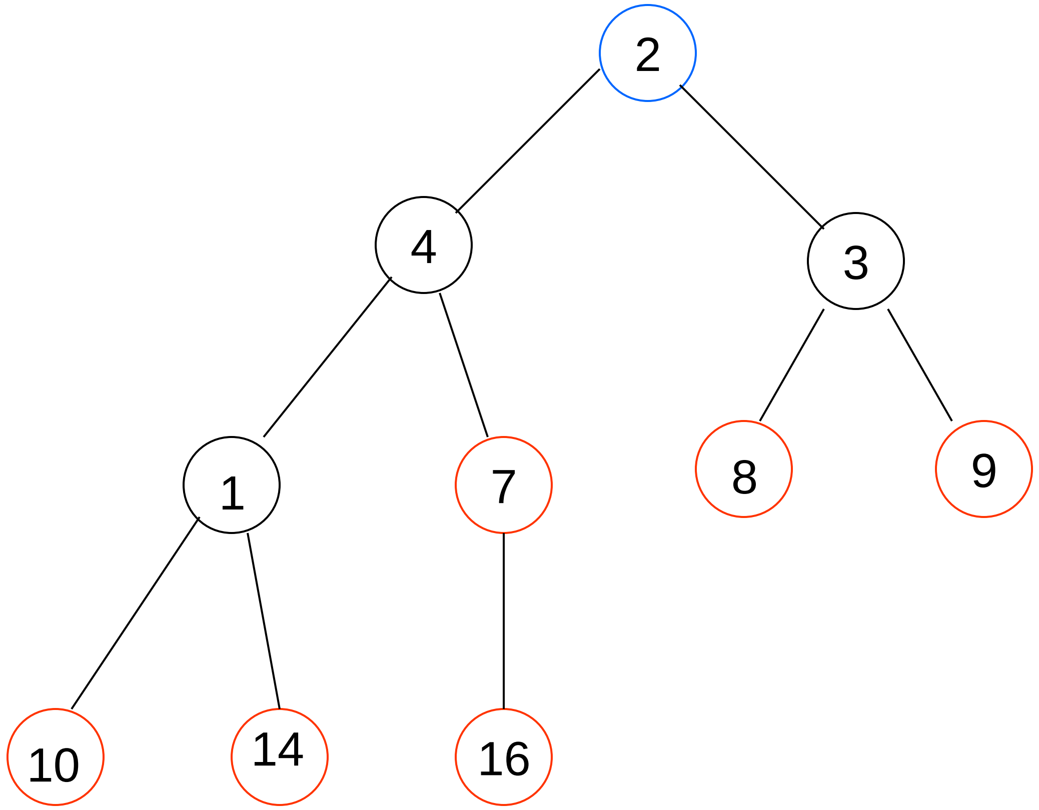
9, 8, 3, 4, 7, 1, 2, 10, 14, 16.

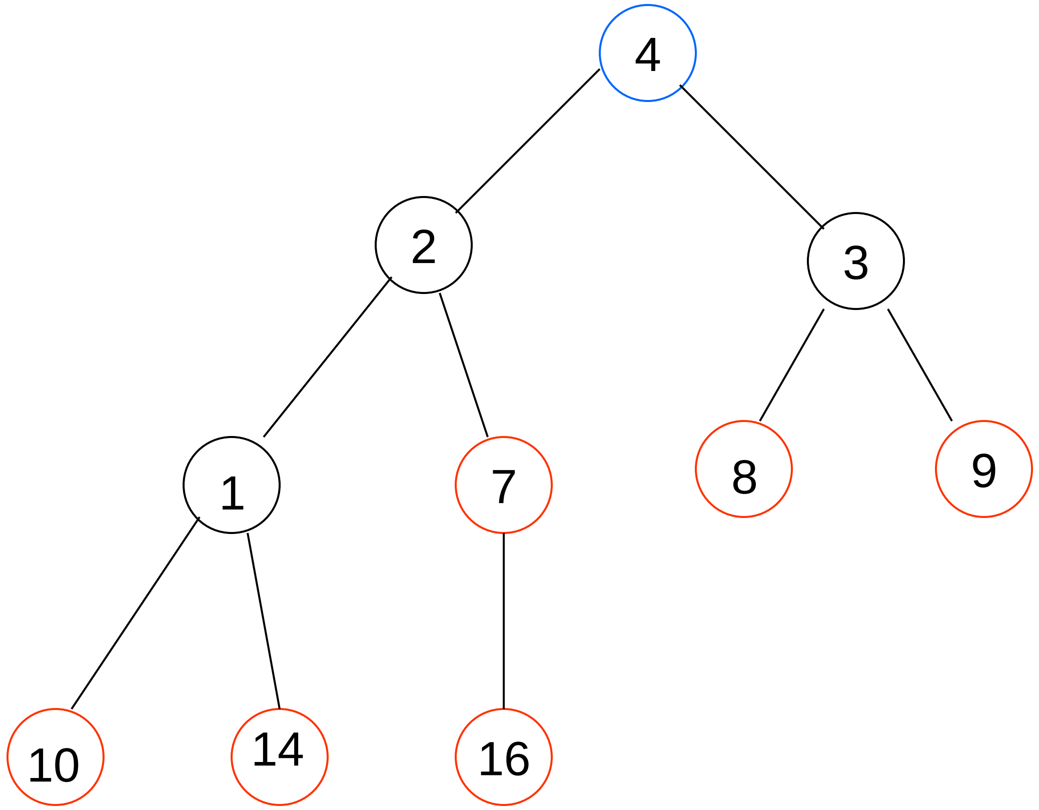

2
8
3
4
7
1
9
10
14
16

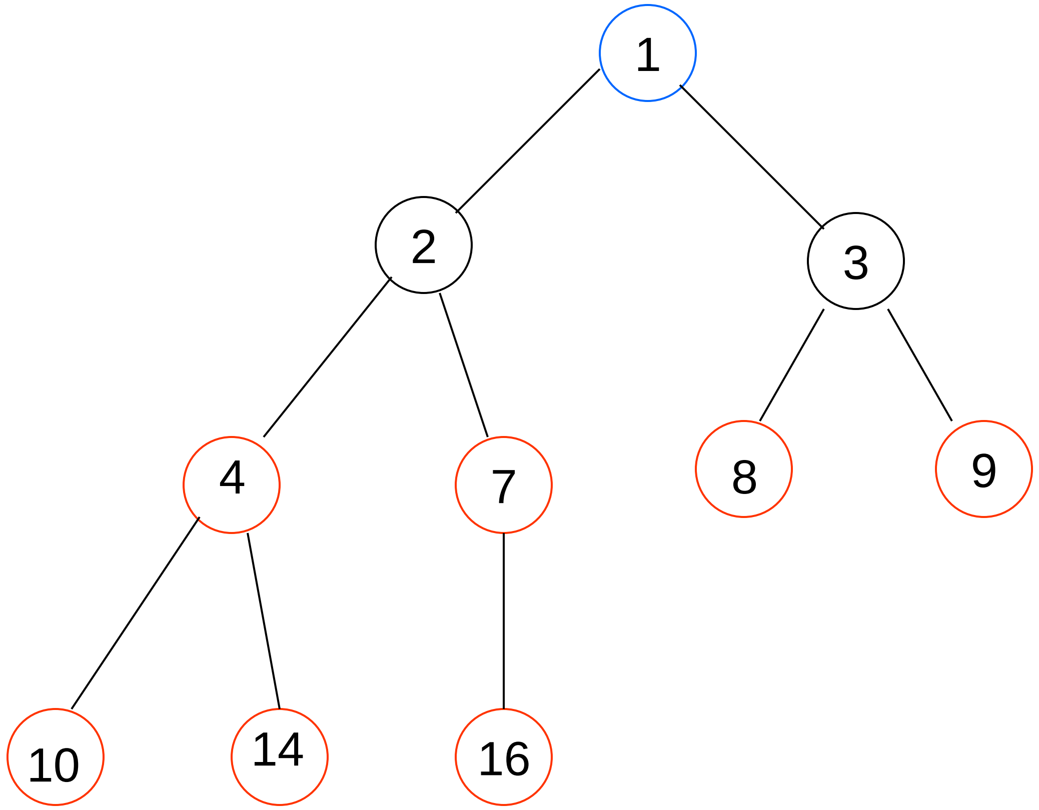

8
7
3
4
2
1
9
10
14
16

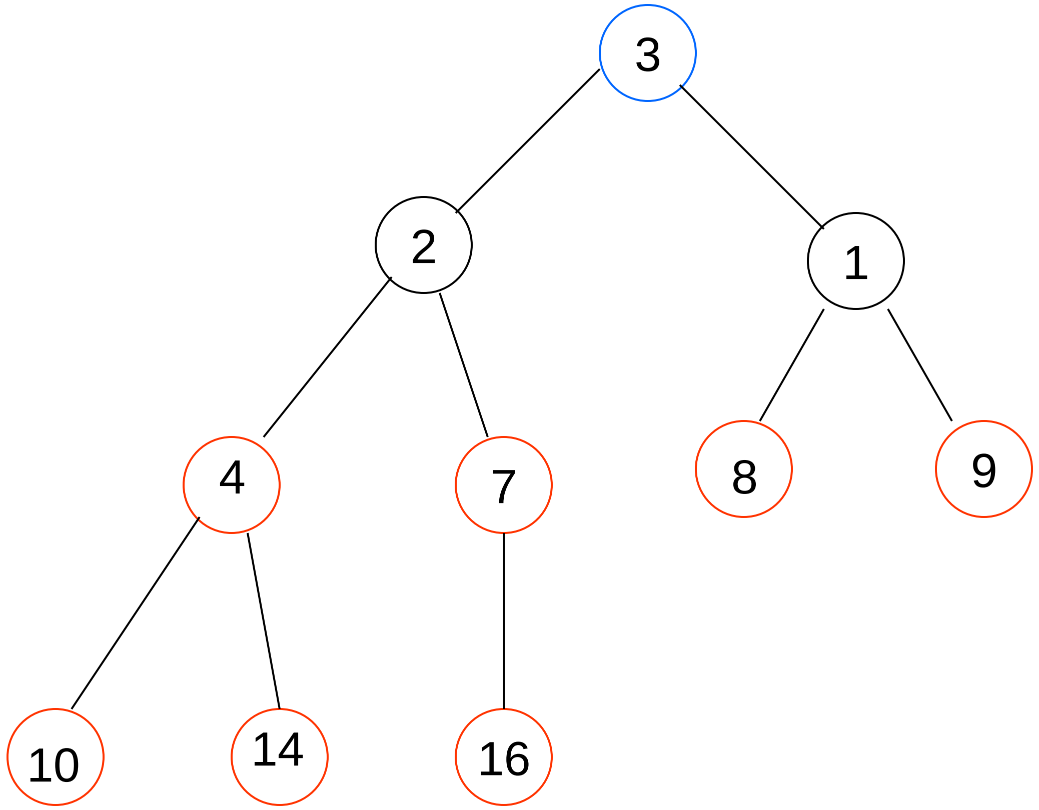

1
7
3
4
2
8
9
10
14
16

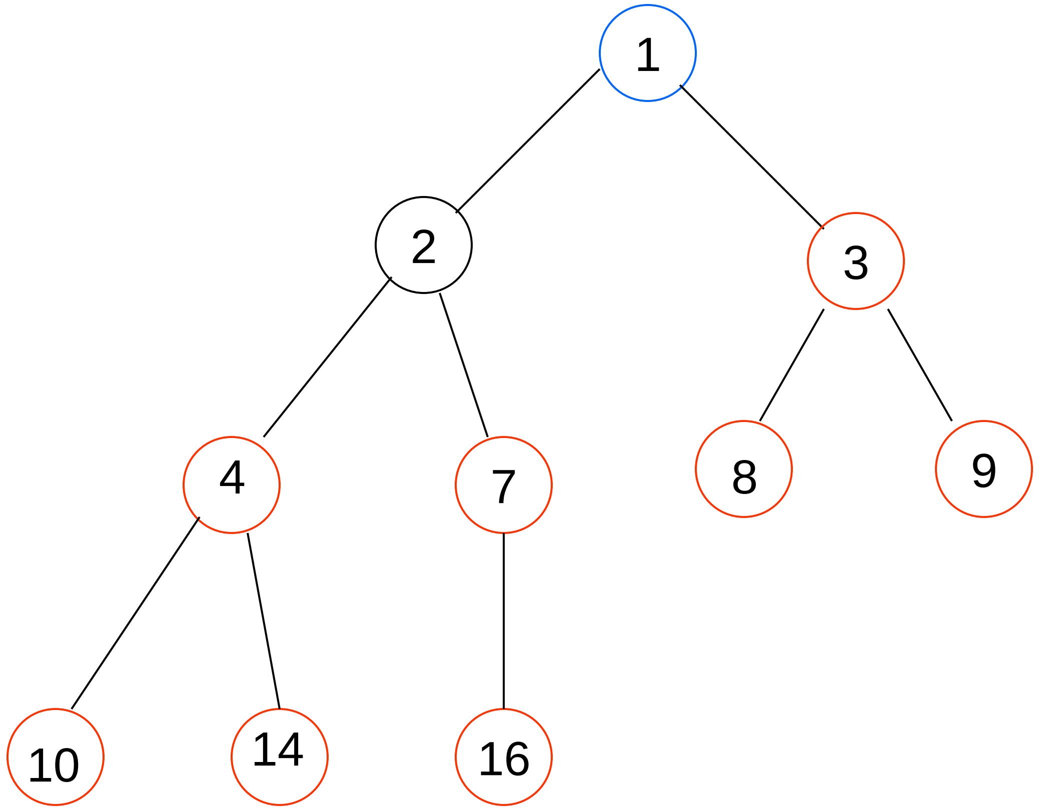

7
4
3
1
2
8
9
10
14
16

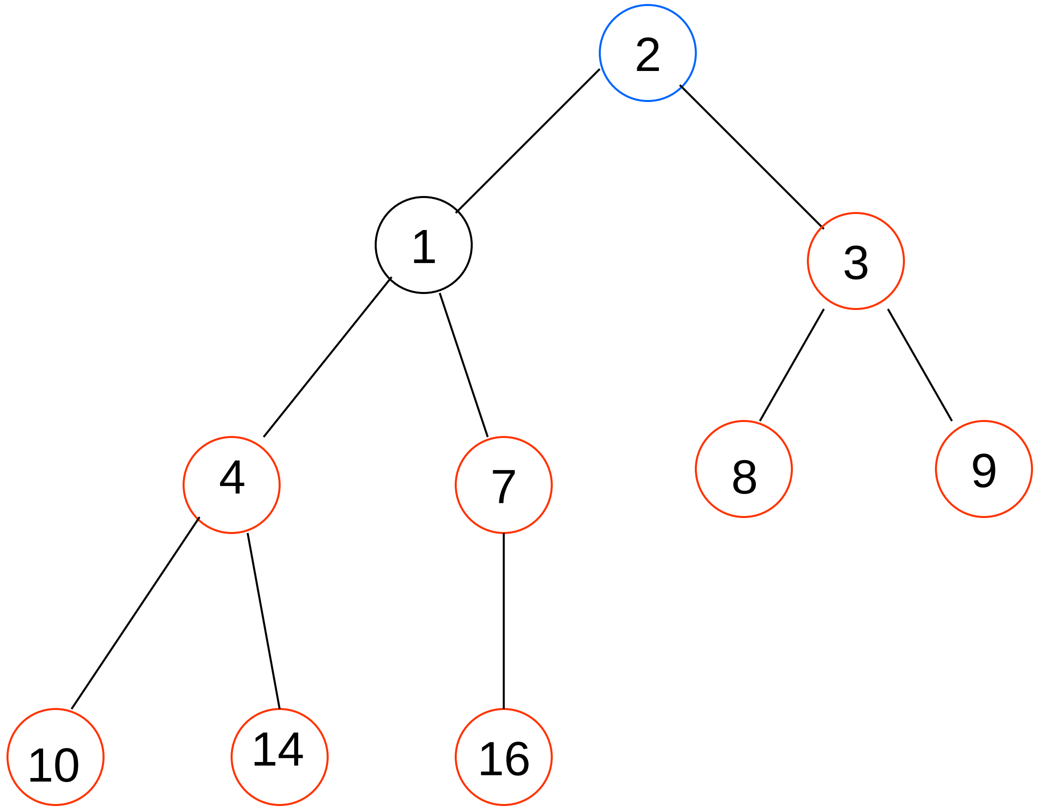

2
4
3
1
7
8
9
10
14
16

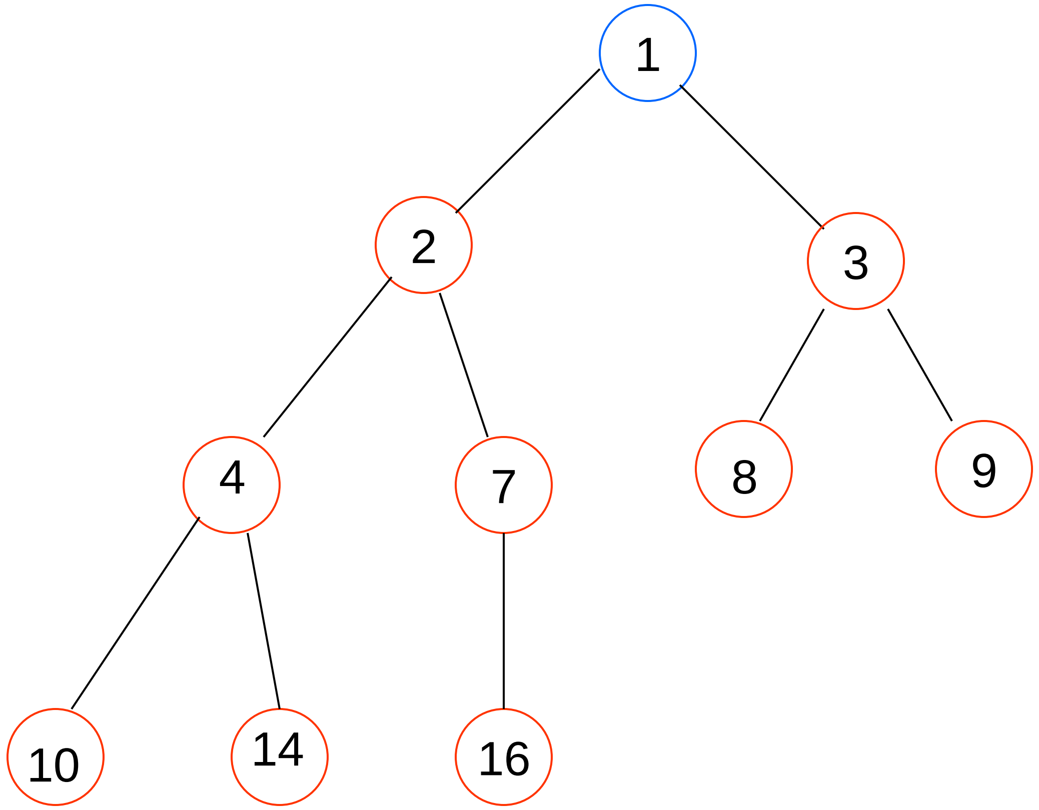

4
2
3
1
7
8
9
10
14
16


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

Running Time

```
$\text{Heapsort}(A)$
    $\text{Buid-Max-Heap}(A);$                                O(n)
    $\text{for } i \leftarrow length[A] \text{ downto } 2$
      $\text{do begin}$
          $\text{exchange } A[1] \leftrightarrow A[i];$
          $heap\text{-}size[A] \leftarrow heap\text{-}size[A] - 1;$
          $\text{Max-Heapify}(A, 1);$                         O(lg n)
      $\text{end-for}$
                                                              $O(n \lg n)$
```

Pseudocode-Heap Sort

```
// Main function to do heap sort
void heapSort(int arr[], int N)
{
      // Build max heap
    for (int i = N / 2 - 1; i >= 0; i--)
          heapify(arr, N, i);
      // Heap sort
    for (int i = N - 1; i >= 0; i--) {
          swap(&arr[0], &arr[i]);
          // Heapify root element to get highest
element at
        // root again
        heapify(arr, i, 0);
    }
}
```

```
void swap(int* a, int* b)
{
      int temp = *a;
      *a = *b;
      *b = temp;
}
```

```
// To heapify a subtree rooted with node i which is an index in
arr[]. n is size of heap
void heapify(int arr[], int N, int i)
{
    // Find largest among root, left child and right child
      // Initialize largest as root
    int largest = i;
      // left = 2*i + 1
    int left = 2 * i + 1;
      // right = 2*i + 2
    int right = 2 * i + 2;
      // If left child is larger than root
    if (left < N && arr[left] > arr[largest])
          largest = left;
      // If right child is larger than largest
    // so far
    if (right < N && arr[right] > arr[largest])
          largest = right;
      // Swap and continue heapifying if root is not largest
    // If largest is not root
    if (largest != i) {
          swap(&arr[i], &arr[largest]);
          // Recursively heapify the affected
        // sub-tree
        heapify(arr, N, largest);
    }
}
```

Pseudocode-Heap Sort

```
# Python program for implementation of heap Sort
# To heapify subtree rooted at index i. n is size of
heap

def heapify(arr, n, i):
    largest = i  # Initialize largest as root
    l = 2 * i + 1 # left = 2*i + 1
    r = 2 * i + 2 # right = 2*i + 2

 # See if left child of root exists and is greater than
root
    if l < n and arr[i] < arr[l]:
        largest = l

 # See if right child of root exists and is greater
than root
    if r < n and arr[largest] < arr[r]:
        largest = r

 # Change root, if needed

    if largest != i:
        (arr[i], arr[largest]) = (arr[largest],
arr[i])  # swap

  # Heapify the root.

        heapify(arr, n, largest)
```

```
# The main function to sort an array of given size

def heapSort(arr):
    n = len(arr)

 # Build a maxheap.
 # Since last parent will be at ((n//2)-1) we can
start at that location.

    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

 # One by one extract elements

    for i in range(n - 1, 0, -1):
        (arr[i], arr[0]) = (arr[0], arr[i])  # swap
        heapify(arr, i, 0)

# Driver code to test above

arr = [12, 11, 13, 5, 6, 7, ]
heapSort(arr)
n = len(arr)
print('Sorted array is')
for i in range(n):
    print(arr[i])
```

Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority value. And elements are served on the basis of their priority. That is, higher priority elements are served first.
- However, if elements with the same priority occur, they are served according to their order in the queue.
- Difference between Priority Queue and Normal Queue
  - In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.
- Implementation of Priority Queue
  - Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Try it in home

Algorithm for insertion of an element into priority queue (max-heap)

```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

Peeking from the Priority Queue (Find max/min)

Algorithm for deletion of an element in the priority queue (max-heap)

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```