Lecture 7-8.
Heapsort

Introduction to Algorithms

Da Nang University of Science and Technology

Dang Thien Binh

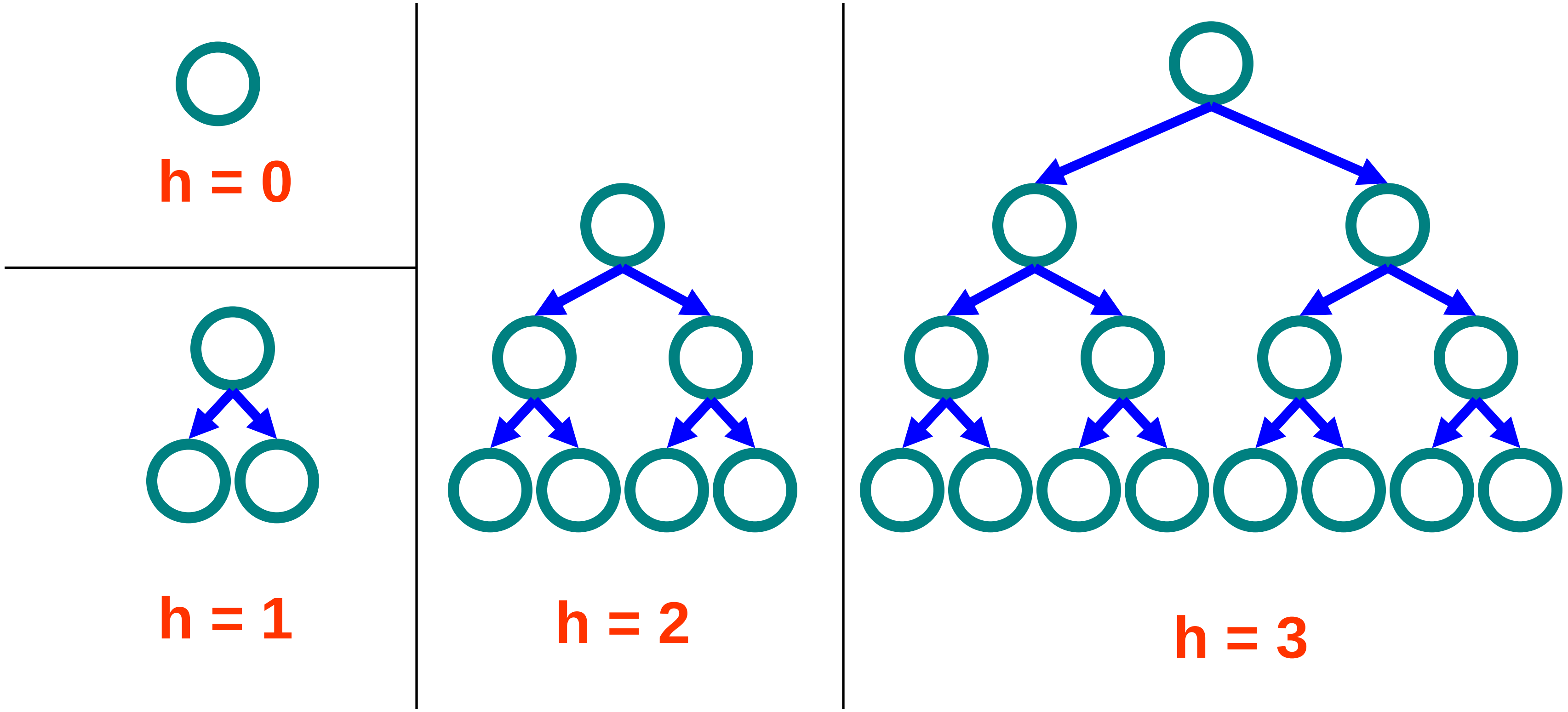
dtbinh@dut.udn.vn

Introduction for Heapsort

- Heapsort
  - Running time: $O(n \lg n)$
    - Like merge sort
  - Sorts in place: only a constant number of array elements are stored outside the input array at any time
    - Like insertion sort
- Heap
  - A data structure used by Heapsort to manage information during the execution of the algorithm
  - Can be used as an efficient priority queue

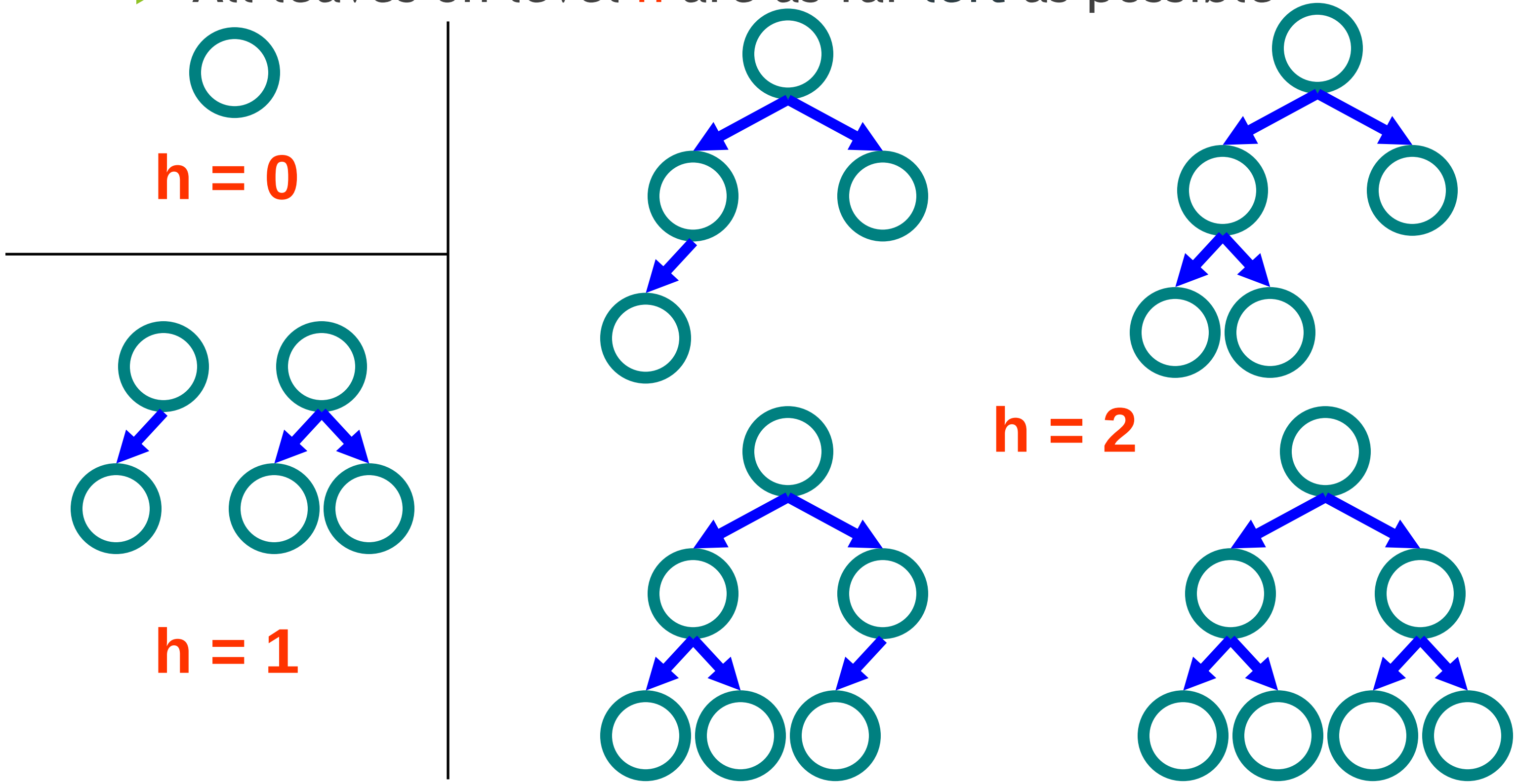
Perfect Binary Tree

- For binary tree with height h
  - All nodes at levels h–1 or less have 2 children (full)

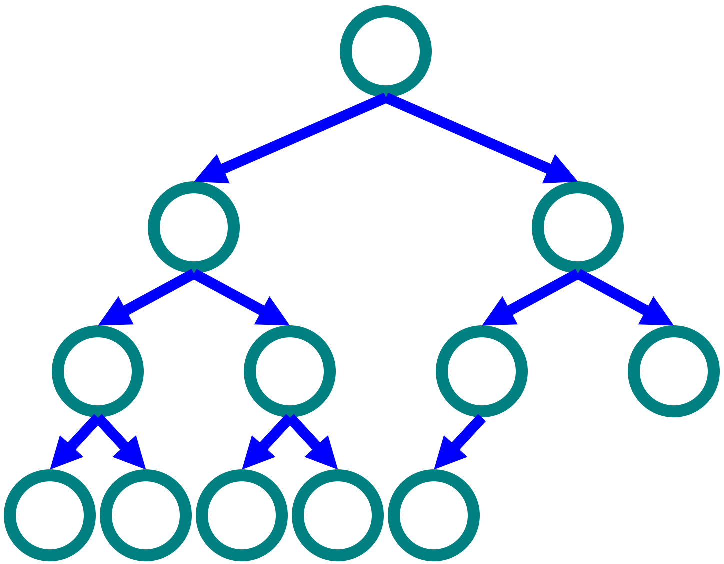


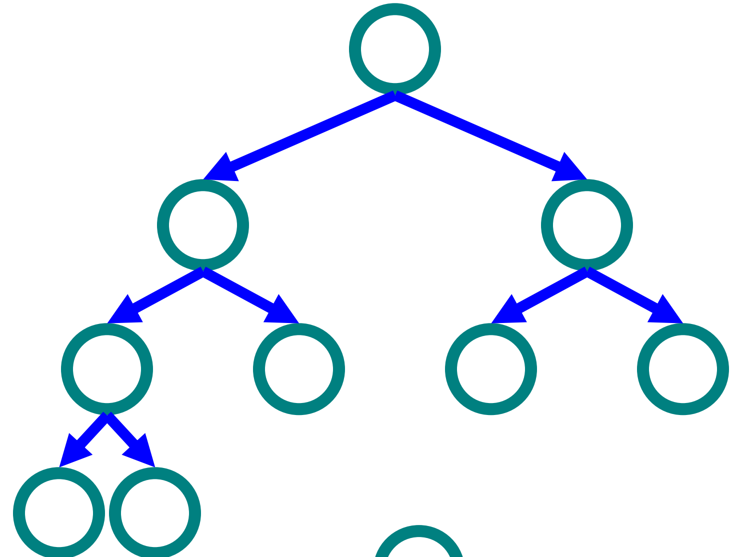
Complete Binary Trees

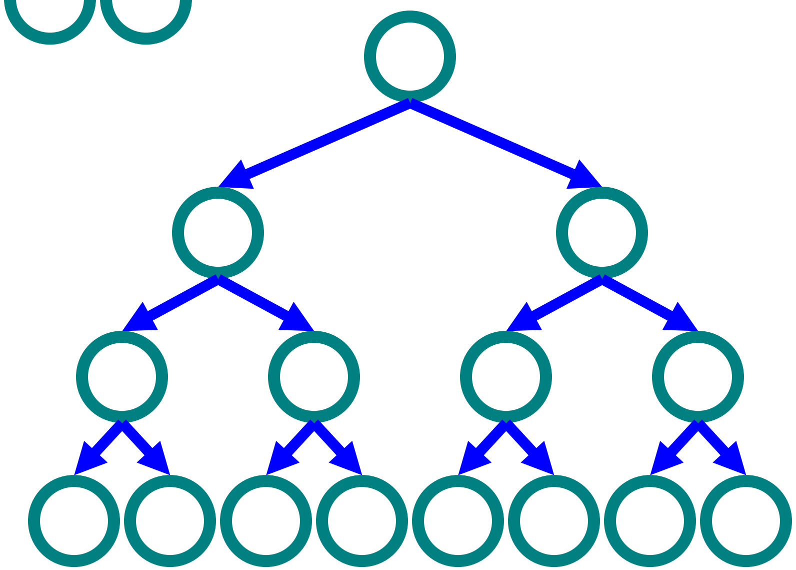
- For binary tree with height h
  - All nodes at levels h-2 or less have 2 children (full)
  - All leaves on level h are as far left as possible



Complete Binary Trees



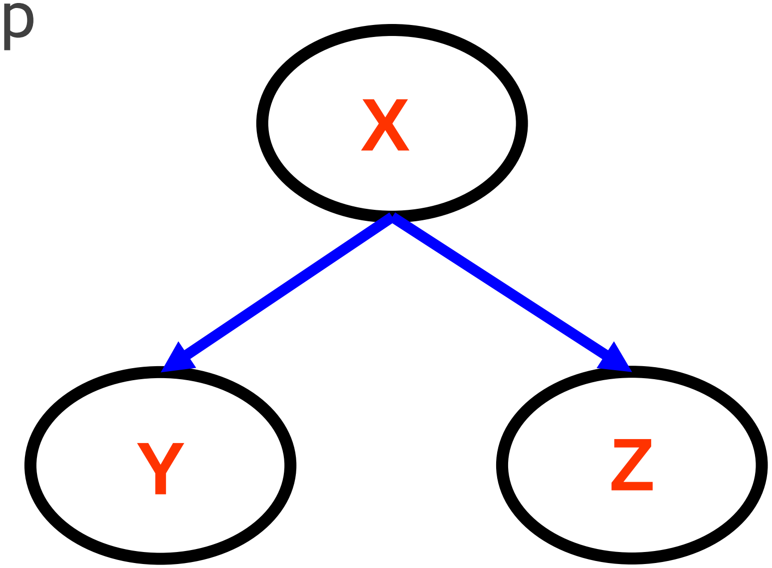




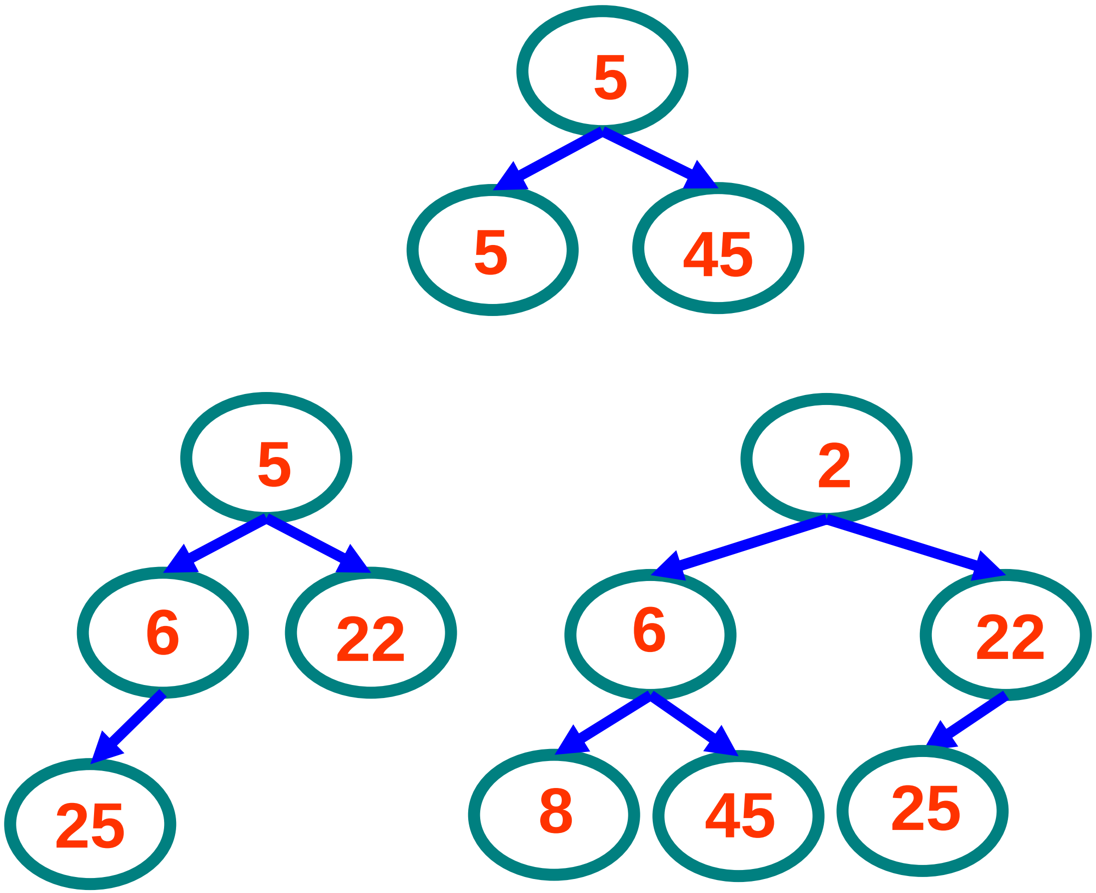
h = 3

Heaps

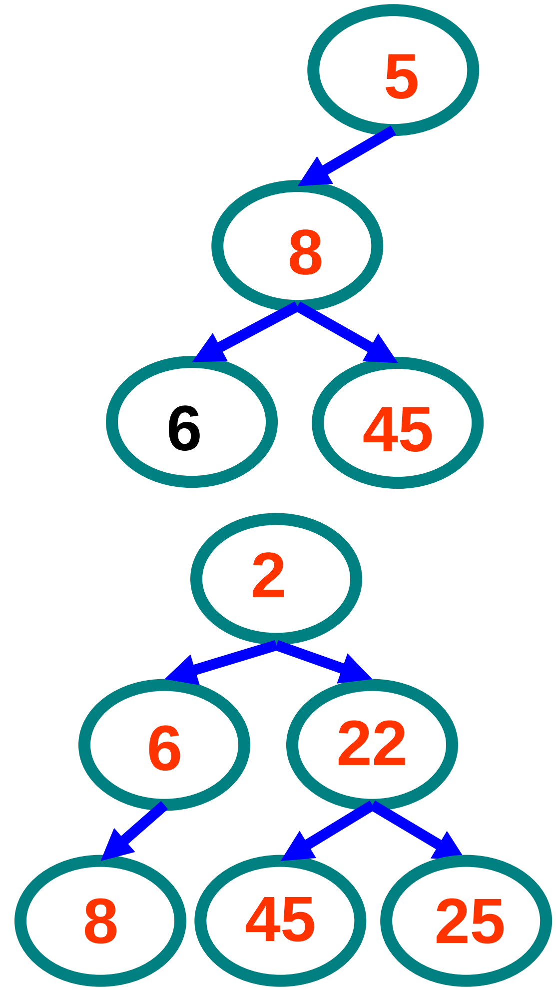
- Two key properties
  - Complete binary tree
  - Value at node
    - Smaller than or equal to values in subtrees
    - Greater than or equal to values in subtrees
- Example max-heap
  - $Y \leq X$
  - $Z \leq X$



Heap and Non-heap Examples



Min-heaps

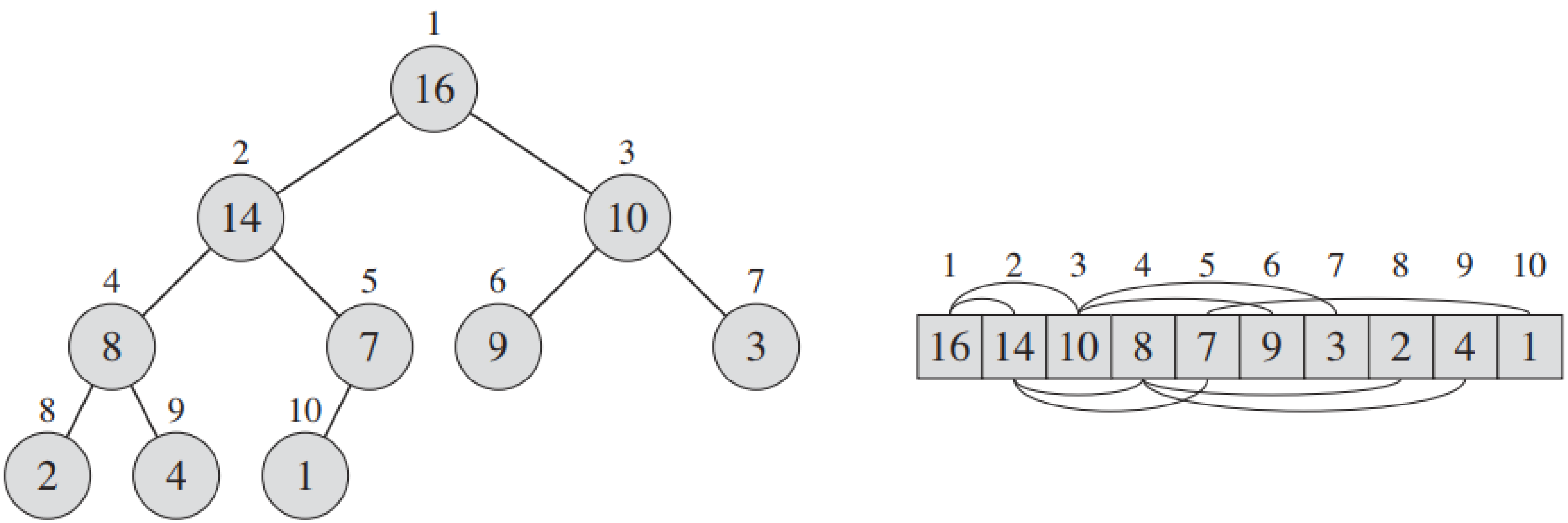


Non-heaps

Binary Heap

- An array object that can be viewed as a nearly complete binary tree
  - Each tree node corresponds to an array element that stores the value in the tree node
  - The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point
  - A has two attributes
    - length[A]: number of elements in the array
    - heap-size[A]: number of elements in the heap stored within A
    - heap-size[A] $\leq$ length[A]
  - max-heap and min-heap

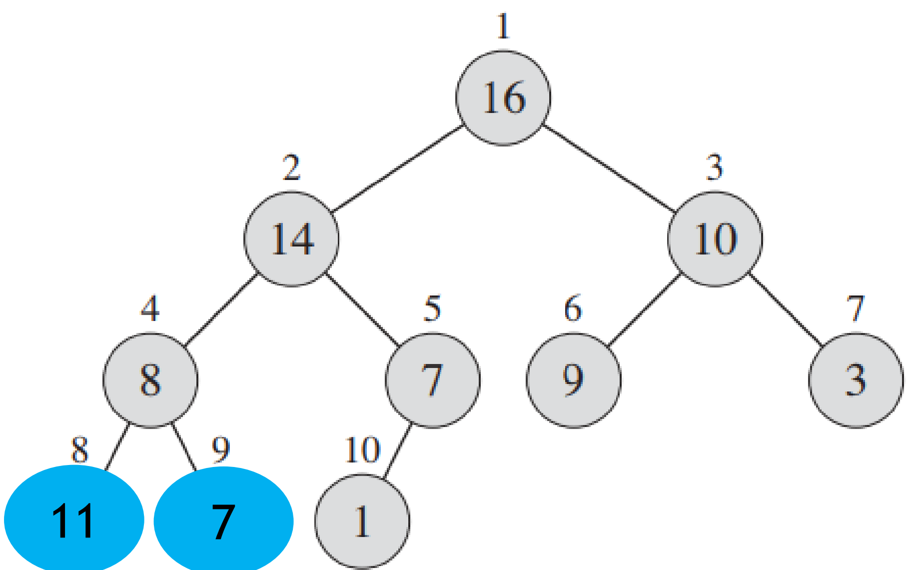
Max-heap



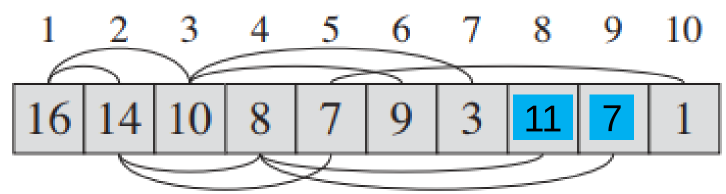
(a) (b)

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Length and Heap-Size



(a)



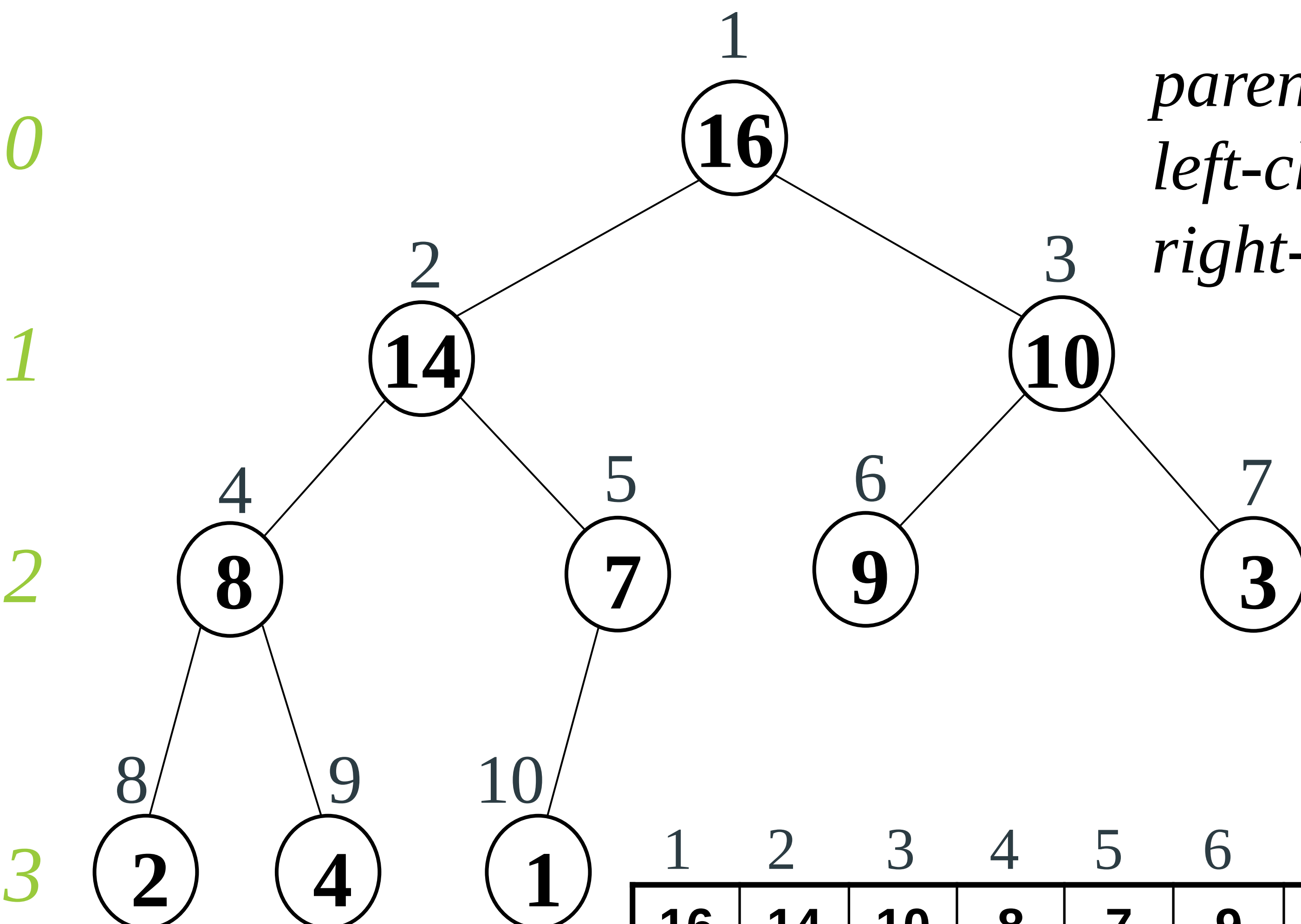
(b)

Length = 10
Heap-Size = 7

Heap Computation

- Given the index *i* of a node, the indices of its parent, left child, and right child can be computed simply:
  - $PARENT(i): return \quad \lfloor i/2 \rfloor$
  - $LEFT(i): return \quad 2i$
  - $RIGHT(i): return \quad 2i+1$

Heap Computation



$parent(i) = \mathrm{floor}(i/2)$

$left\text{-}child(i) = 2i$

$right\text{-}child(i) = 2i + 1$

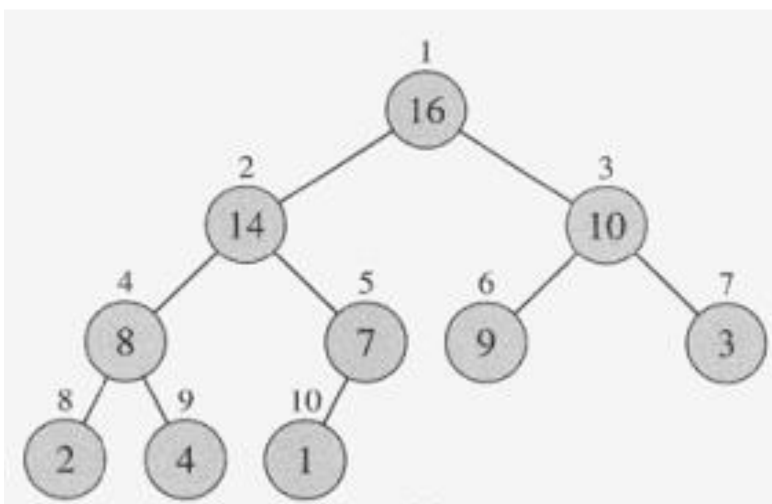
1	2	3	4	5	6	7	8	9	10
16	**14**	**10**	**8**	**7**	**9**	**3**	**2**	**4**	**1**

Heap Property

- Heap property
  - The property that the values in the node must satisfy
- Max-heap property, for every node i other than the root
  - $A[PARENT(i)] \geq A[i]$
  - The value of a node is at most the value of its parent
  - The largest element in a max-heap is stored at the root
  - The subtree rooted at a node contains values no larger than value of the node itself

Heap Height

- The height of a node in a heap
  - The number of edges on the longest simple downward path from the node to a leaf
- The height of a heap is the height of its root
  - The height of a heap of n elements is $\Theta(\lg n)$
    - Exercise 6.1-2 on page 153



Heap Procedures

- MAX-HEAPIFY
  - Maintains the max-heap property
  - $O(\lg n)$
- BUILD-MAX-HEAP
  - Produces a max-heap from an unordered input array
  - $O(n)$
- HEAPSORT
  - Sorts an array in place
  - $O(n \lg n)$

Maintaining the Heap Property

- MAX-HEAPIFY
  - Inputs: an array A and an index i into the array
  - Assume the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but A[i] may be smaller than its children
    - violate the max-heap property
  - MAX-HEAPIFY lets the value at A[i] “float down” in the max-heap

MAX-HEAPIFY

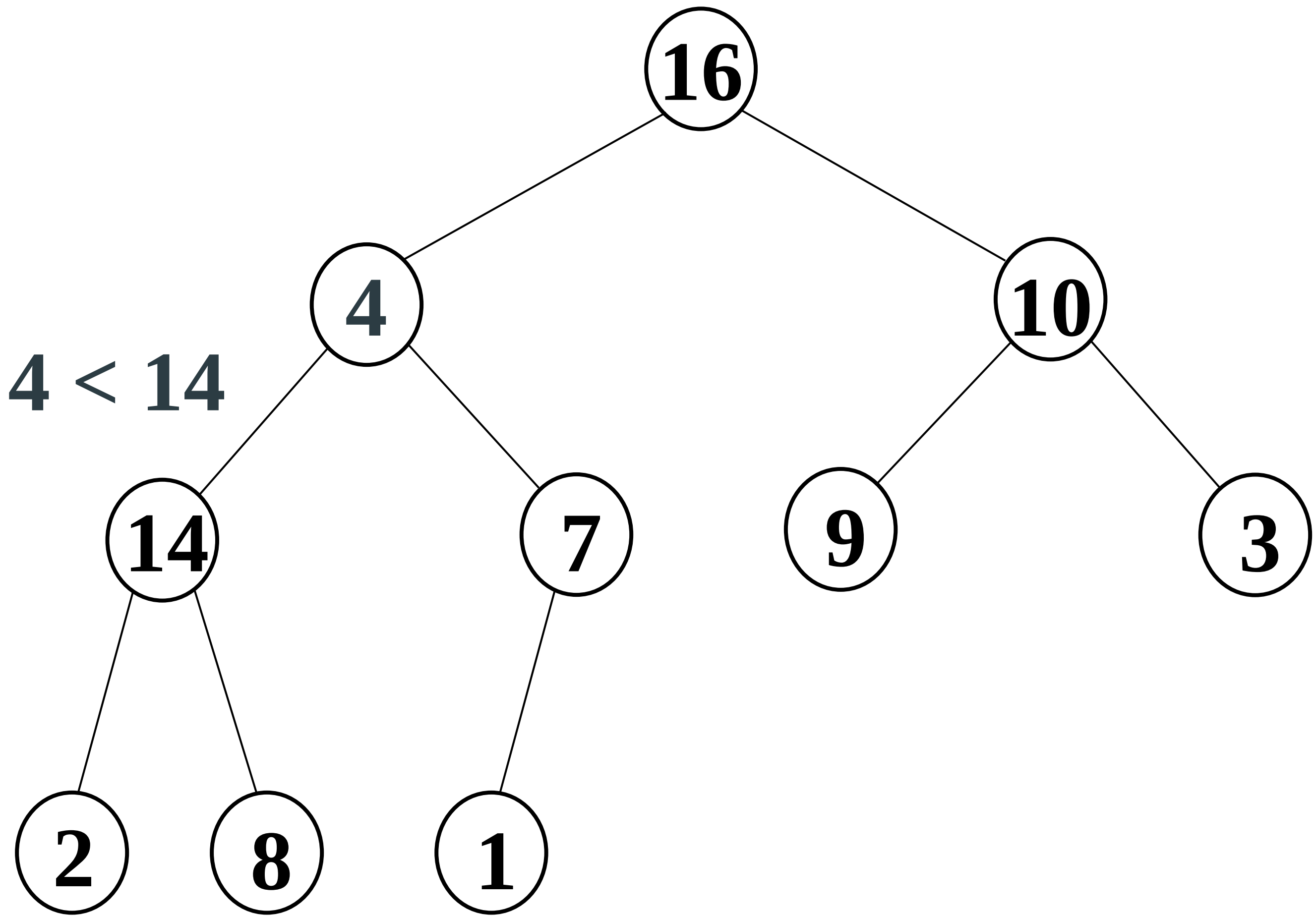
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```

Extract the indices of LEFT and RIGHT children of i

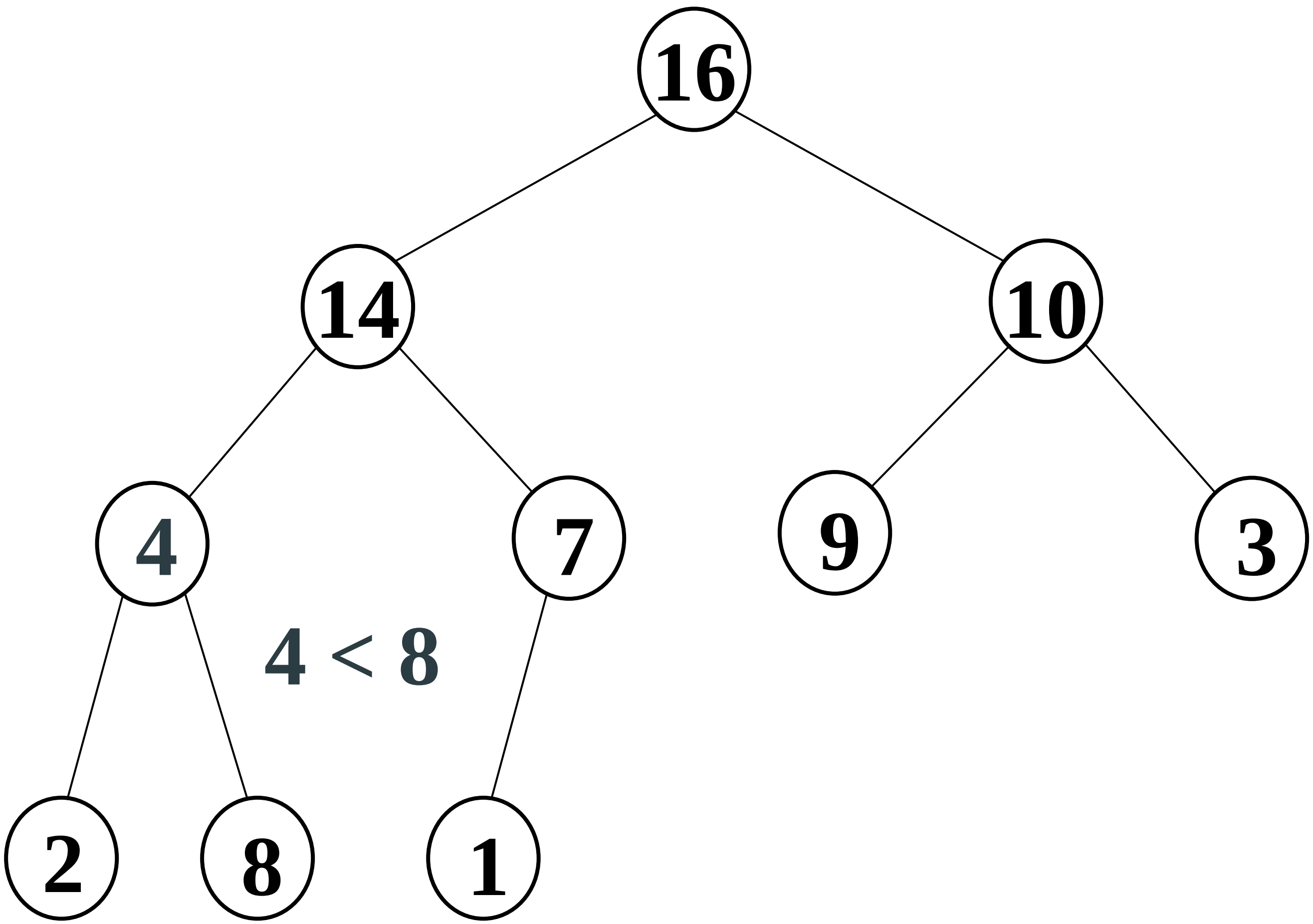
Choose the largest of A[i], A[l], A[r]

Float down A[i] recursively

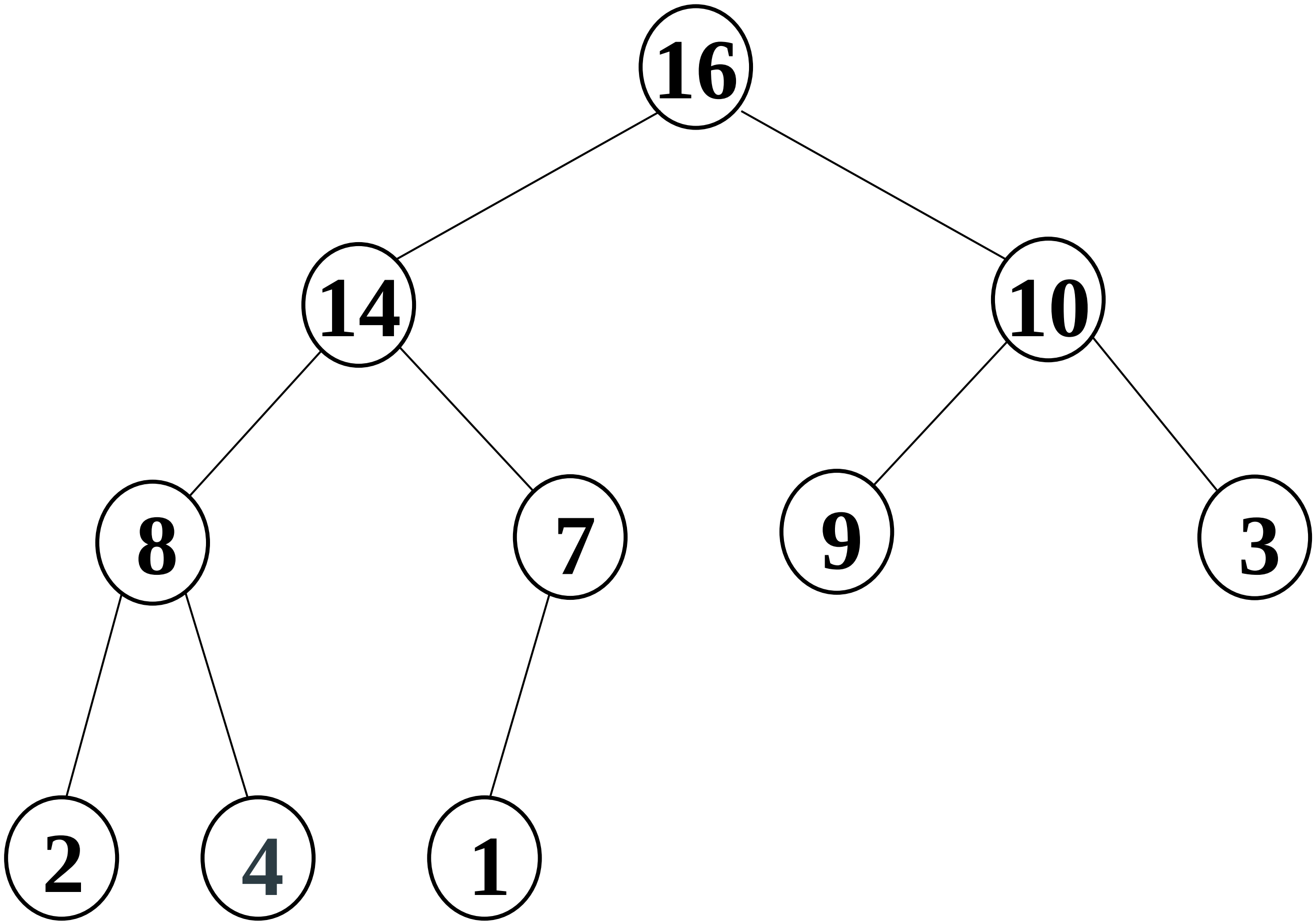
Example of MAX-HEAPIFY (1/3)



Example of MAX-HEAPIFY (2/3)



Example of MAX-HEAPIFY (3/3)



MAX-HEAPIFY

```
void max_heapify(int *a, int i, int n)
{
    int j, temp;
    temp = a[i];
    j = 2*i;
    while (j <= n)
    {
        if (j < n && a[j+1] > a[j])
            j = j+1;
        if (temp > a[j])
            break;
        else if (temp <= a[j])
        {
            a[j/2] = a[j];
            j = 2*j;
        }
    }
    a[j/2] = temp;
    return;
}
```

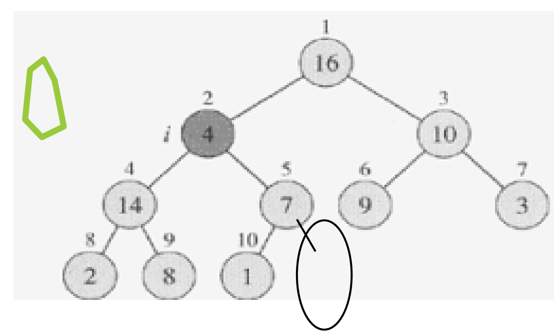
Iterative version

```
void max_heapify(int *a, int i, int n){
    int largest = 0;
    int l = 2*i;
    int r = 2*i+1;
    if (l <= n && a[l] > a[i]){
        largest = l;
    }
    else{
        largest = i;
    }
    if(r <= n && a[r] > a[largest]){
        largest = r;
    }
    if (largest != i){
        int t = a[i];
        a[i] = a[largest];
        a[largest] = t;
        max_heapify(a, largest, n);
    }
}
```

Recursive version

Running Time of MAX-HEAPIFY

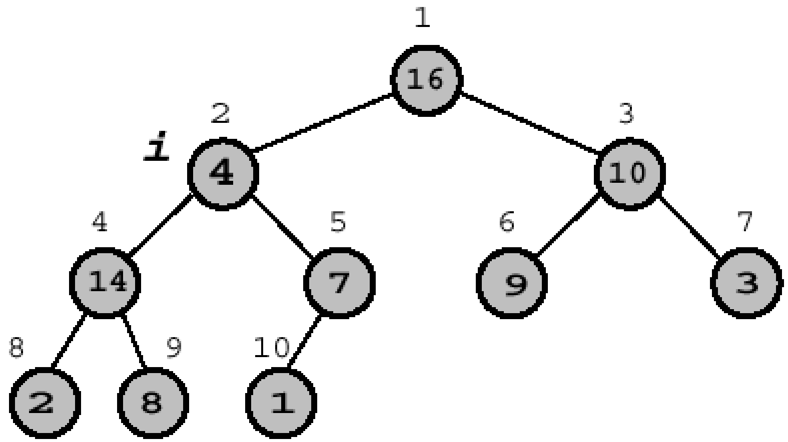
- $\Theta(1)$ to find out the largest among A[i], A[LEFT(i)], and A[RIGHT(i)]
- Plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i
  - The children's subtrees each have size at most 2n/3 (why?)
    - the worst case occurs when the last row of the tree is exactly half full
- $T(n) \le T(2n/3) + \Theta(1)$
  - By case 2 of the master theorem
  - $T(n) = O(\lg n)$



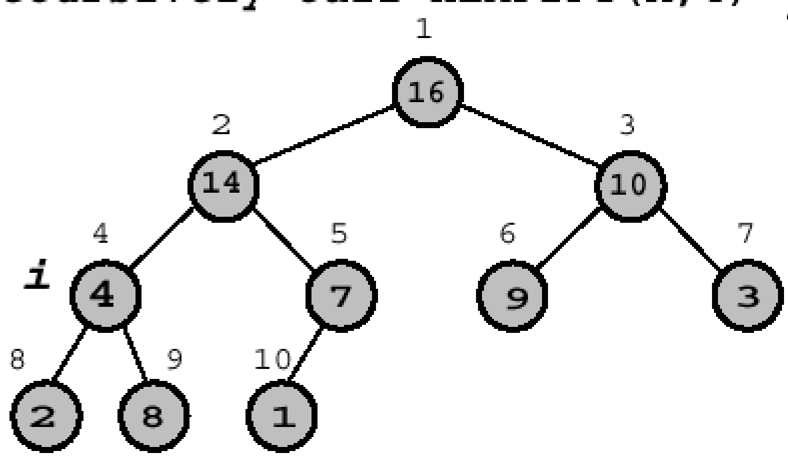
7/11 = 0.63

Heapify Example

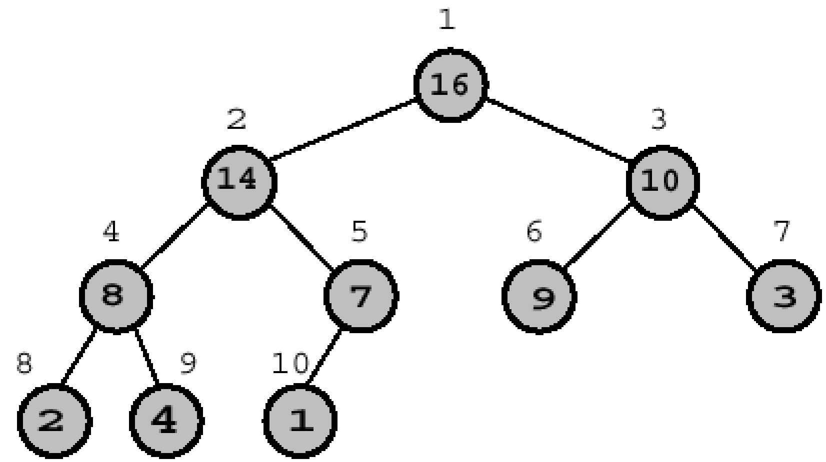
1. Call HEAPIFY(A,2)



2. Exchange A[2] with A[4] and recursively call HEAPIFY(A,4)

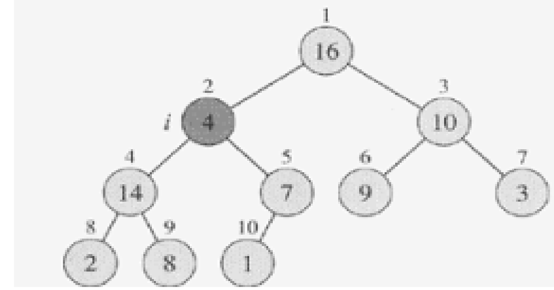


3. Exchange A[4] with A[9] and recursively call HEAPIFY(A,9)



4. Node 9 has no children, so we are done.

Building a Max-Heap



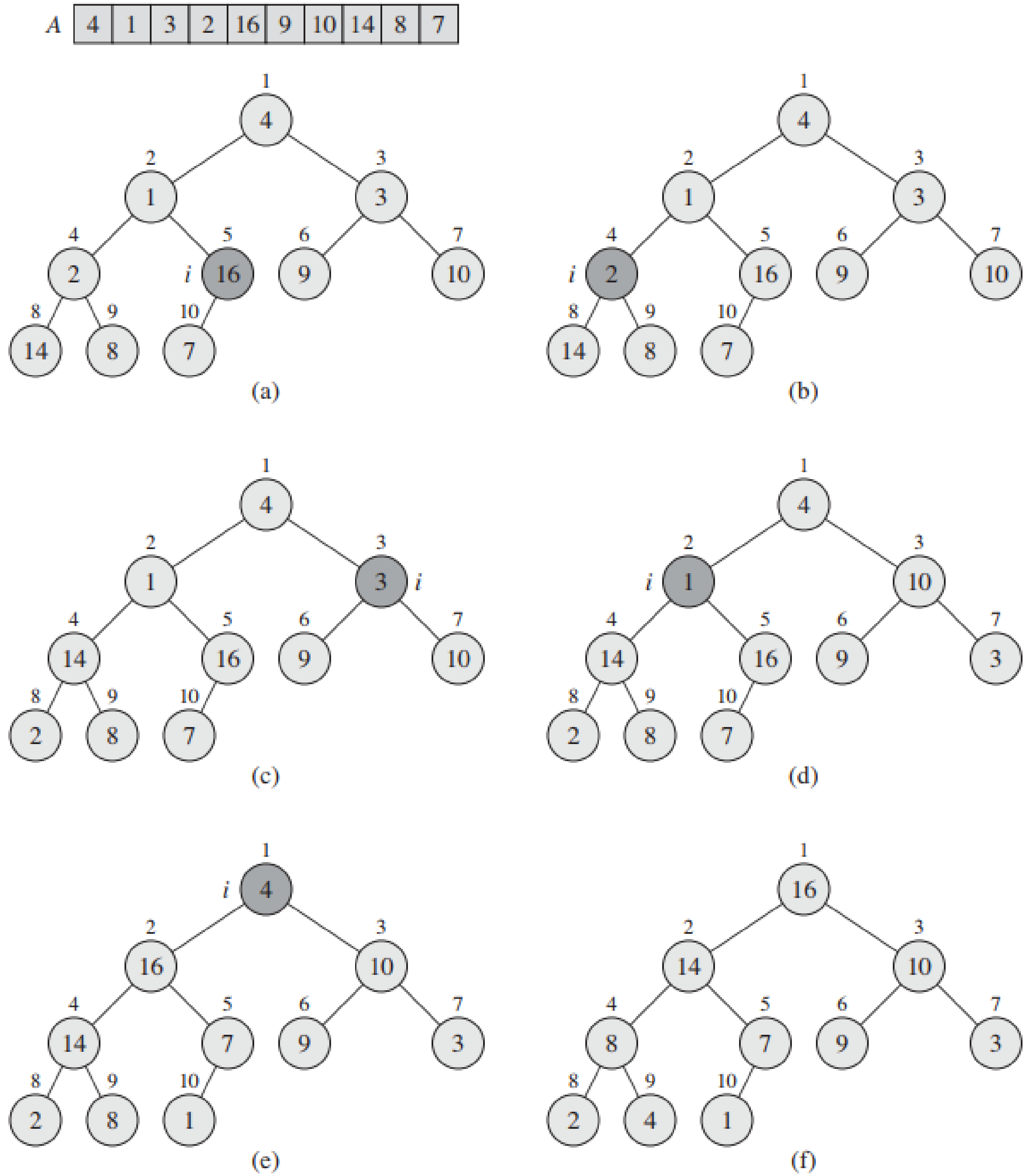
- Observation: $A[(\lfloor n/2 \rfloor+1)..n]$ are all leaves of the tree
  - Exercise 6.1-7 on page 154
  - Each is a 1-element heap to begin with
- Upper bound on the running time
  - O(lg *n*) for each call to MAX-HEAPIFY, and call *n* times → O(*n* lg *n*)
    - Not tight

BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

```
void build_maxheap(int *a, int n)
{
    int i;
    for(i = n/2; i >= 1; i--)
    {
        max_heapify(a, i, n);
    }
}
```

Building a Max-Heap



Loop Invariant

- At the start of each iteration of the ***for*** loop of lines 2-3, each node i+1, i+2, .., n is the root of a max-heap
  - **Initialization**: Prior to the first iteration of the loop, i = $\lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor$+1, $\lfloor n/2 \rfloor$+2,.., n is a leaf and the root of a trivial max-heap.
  - **Maintenance**: Observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes i+1, i+2, …, n are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.
  - **Termination**: At termination, i=0. By the loop invariant, each node 1, 2, …, n is the root of a max-heap. In particular, node 1 is.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Cost for Build-MAX-HEAP

- Heap-properties of an n-element heap
  - Height = $\lfloor \lg n \rfloor$
  - At most $\lceil n/2^{h+1} \rceil$ nodes of any height h
    - Exercise 6.3-3 on page 159

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

Ignore the constant ½

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

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$ (for |x| < 1)

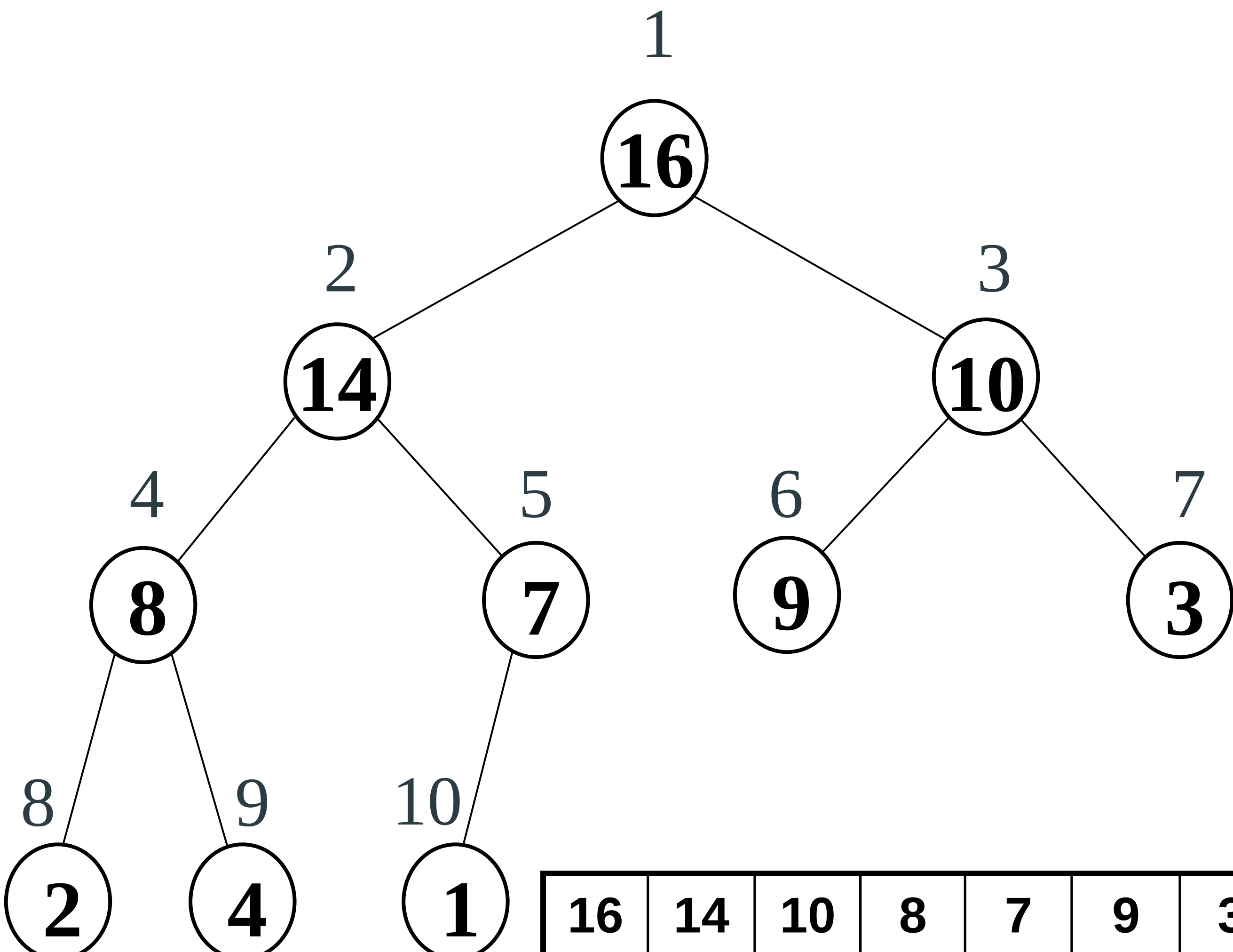
Heapsort

- Using BUILD-MAX-HEAP to build a max-heap on the input array A[1..n], where n=length[A]
- Put the maximum element, A[1], to A[n]
  - Then discard node n from the heap by decrementing heap-size(A)
- A[2..n-1] remain max-heaps, but A[1] may violate
  - call MAX-HEAPIFY(A, 1) to restore the max-heap property for A[1..n-1]
- Repeat the above process from n down to 2
- Cost: O(n lg n)
  - BUILD-MAX-HEAP: O(n)
  - Each of the n-1 calls to MAX-HEAPIFY takes time O(lg n)

Heapsort Algorithm

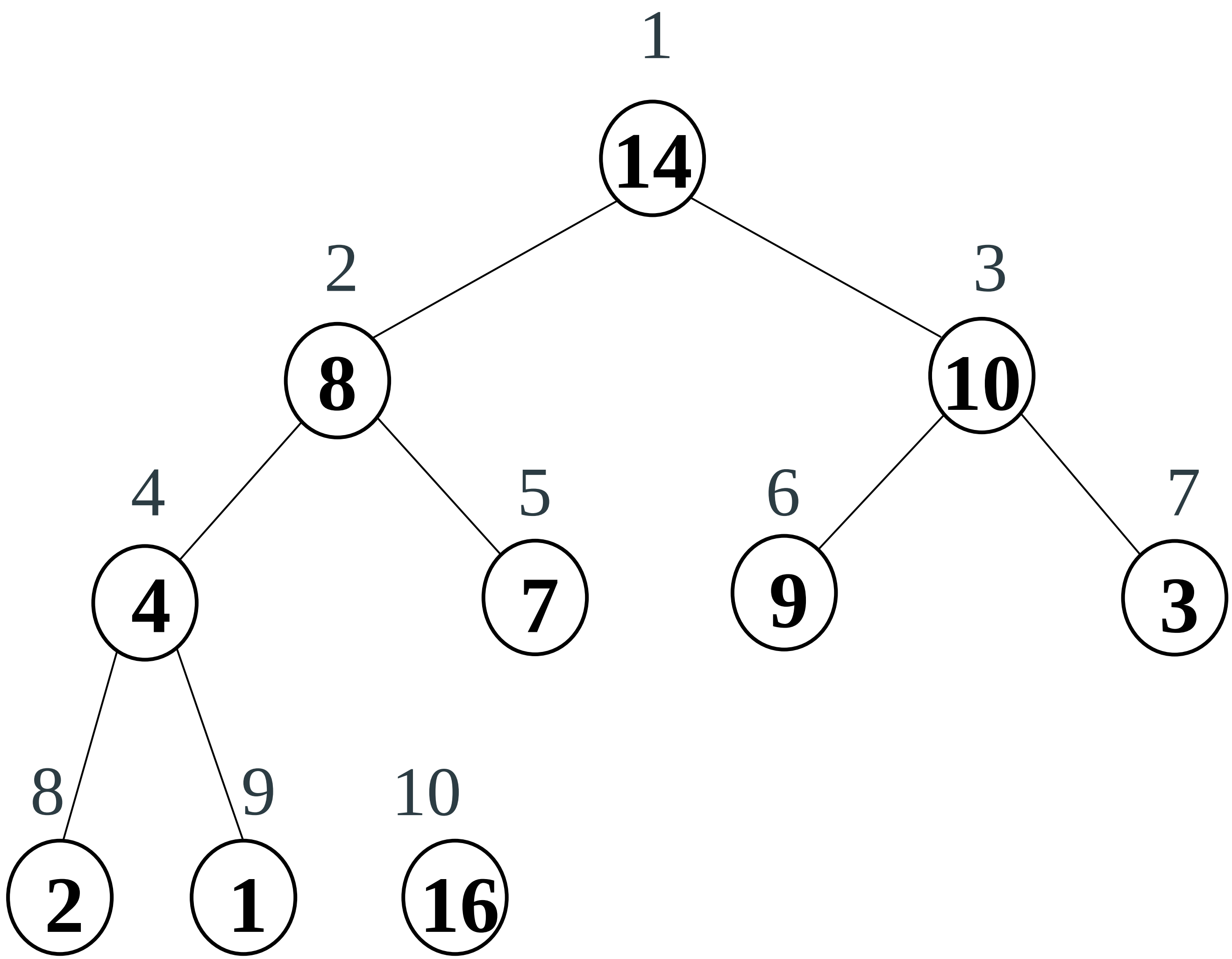
```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size - 1
        MAX-HEAPIFY(A, 1)
```

Example: Heapsort (1/8)

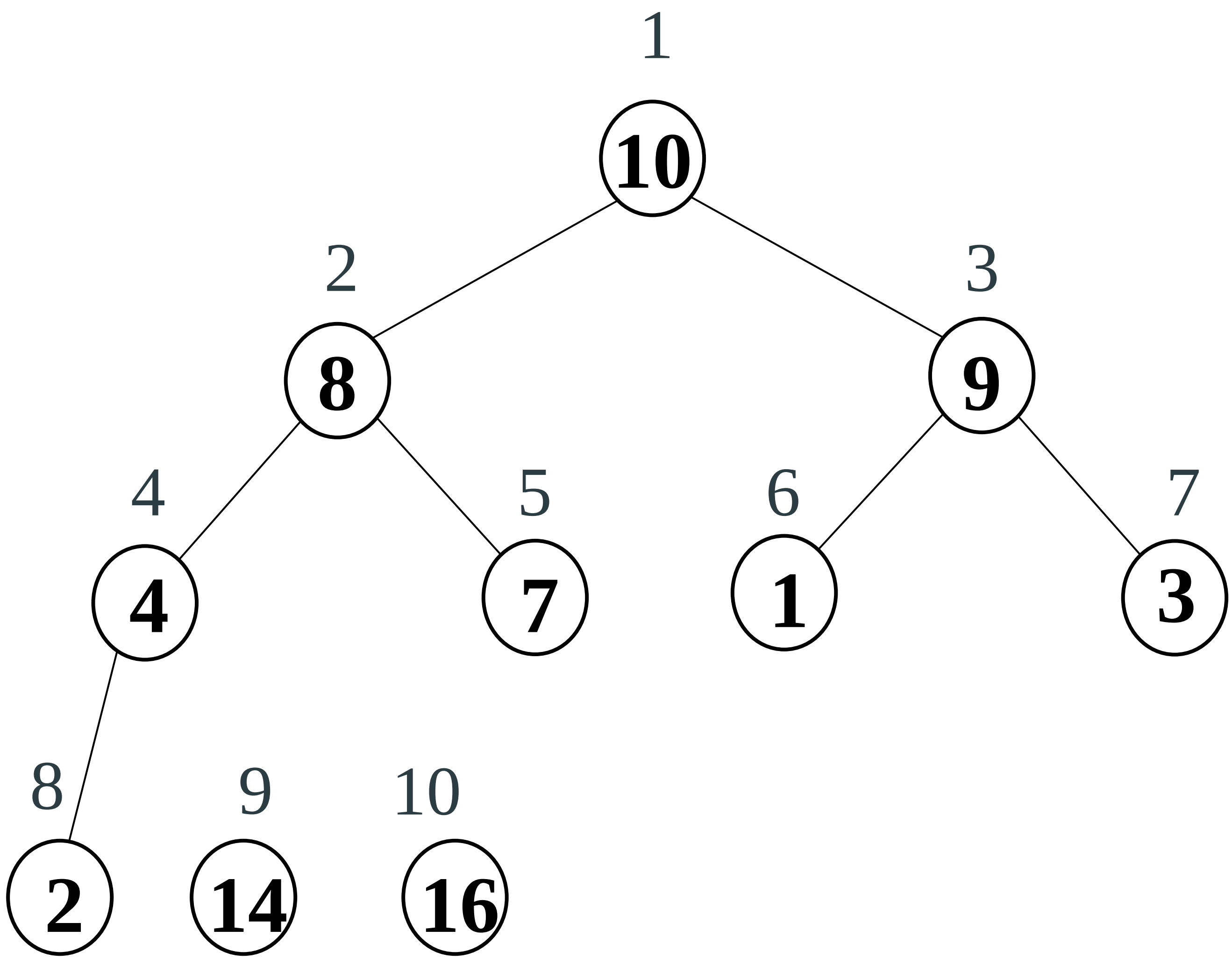


16	14	10	8	7	9	3	2	4	1

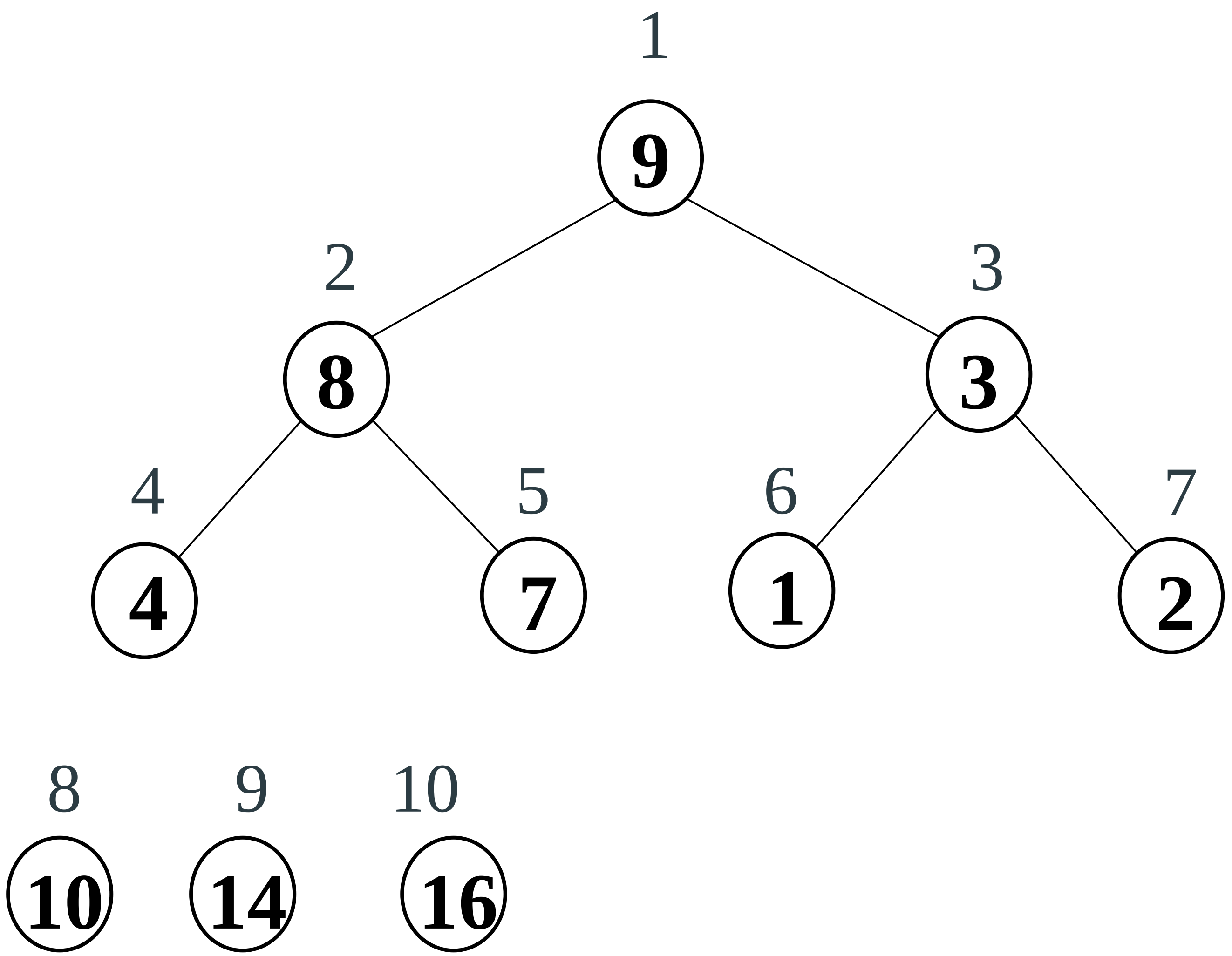
Example: Heapsort (2/8)



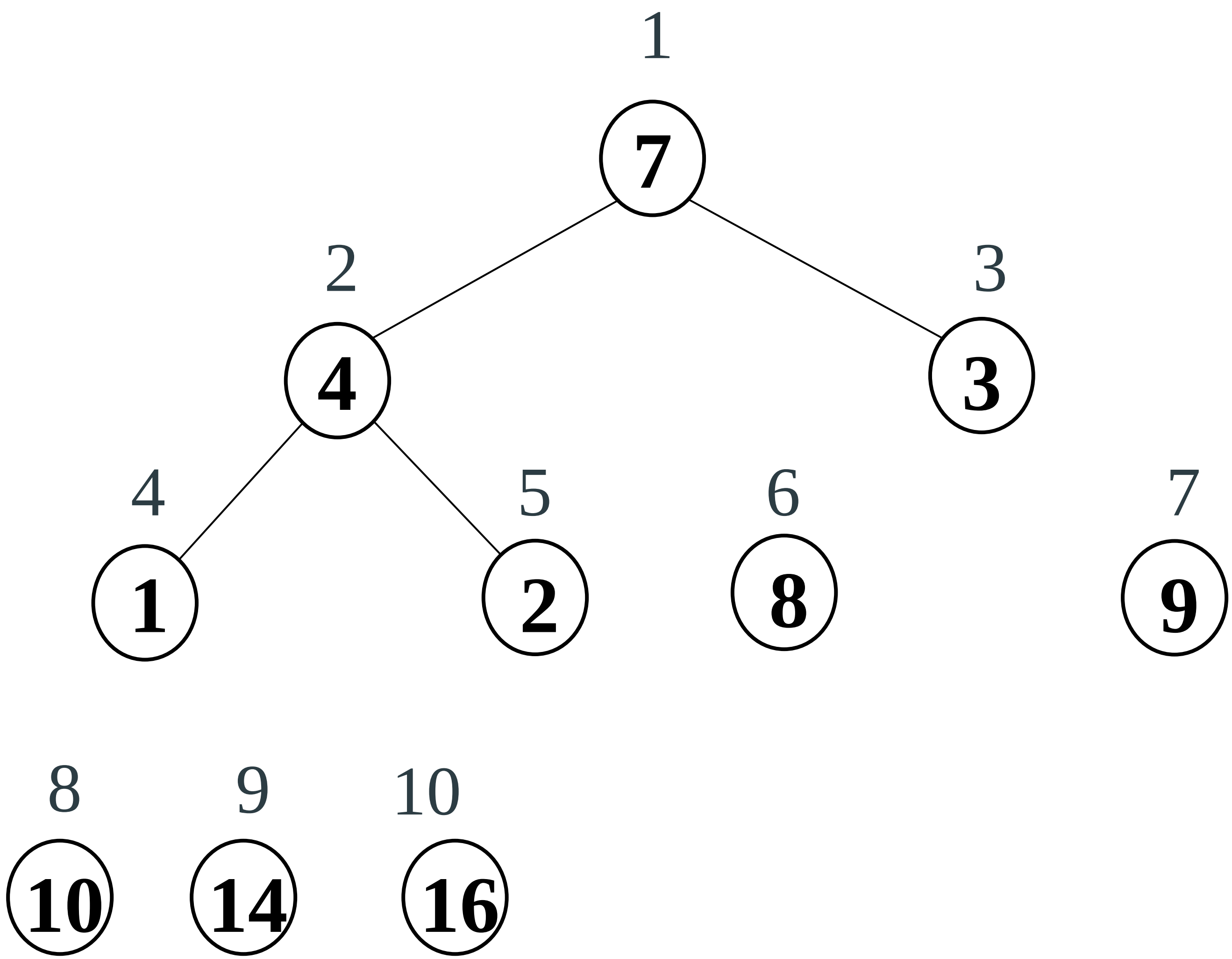
Example: Heapsort (3/8)



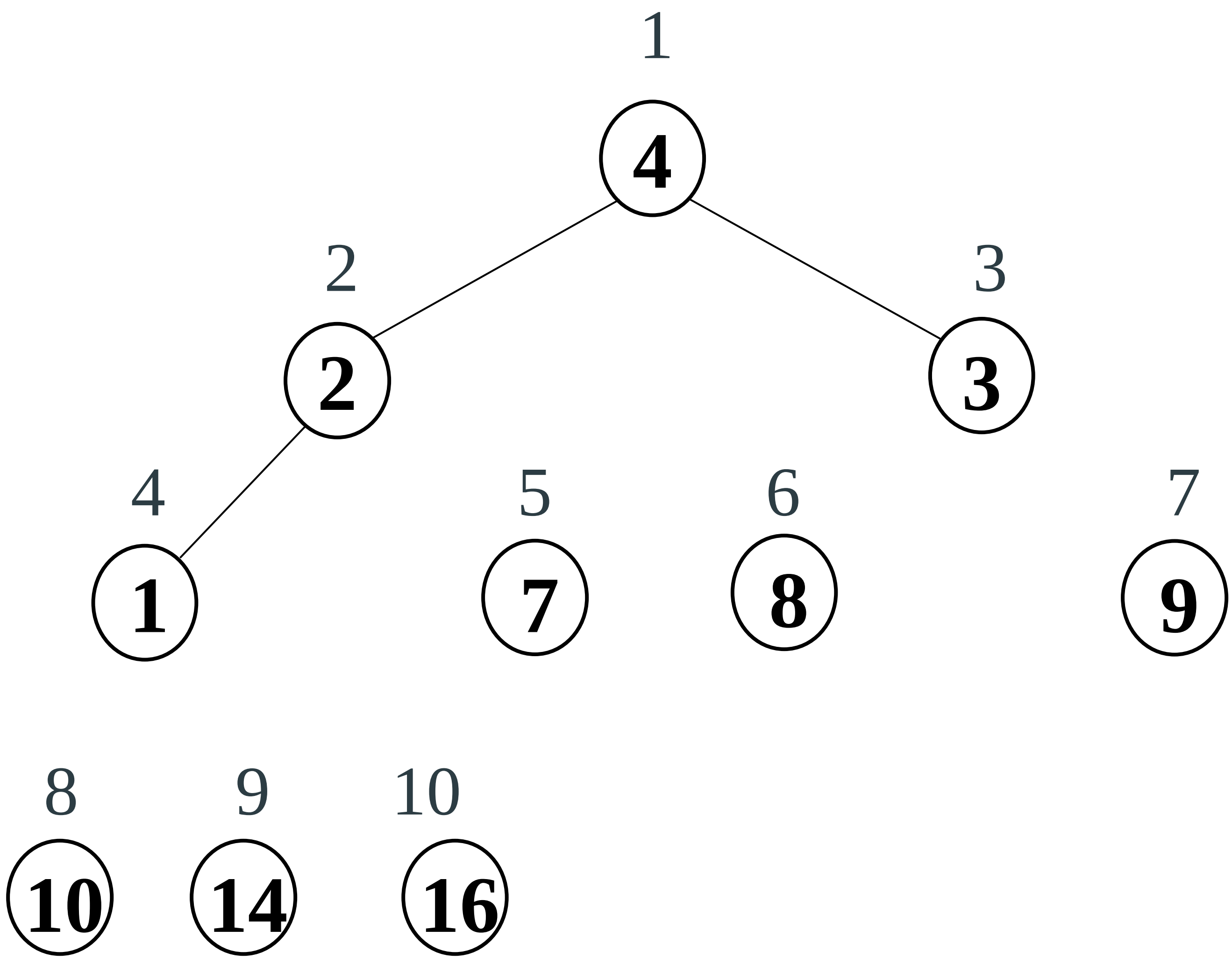
Example: Heapsort (4/8)



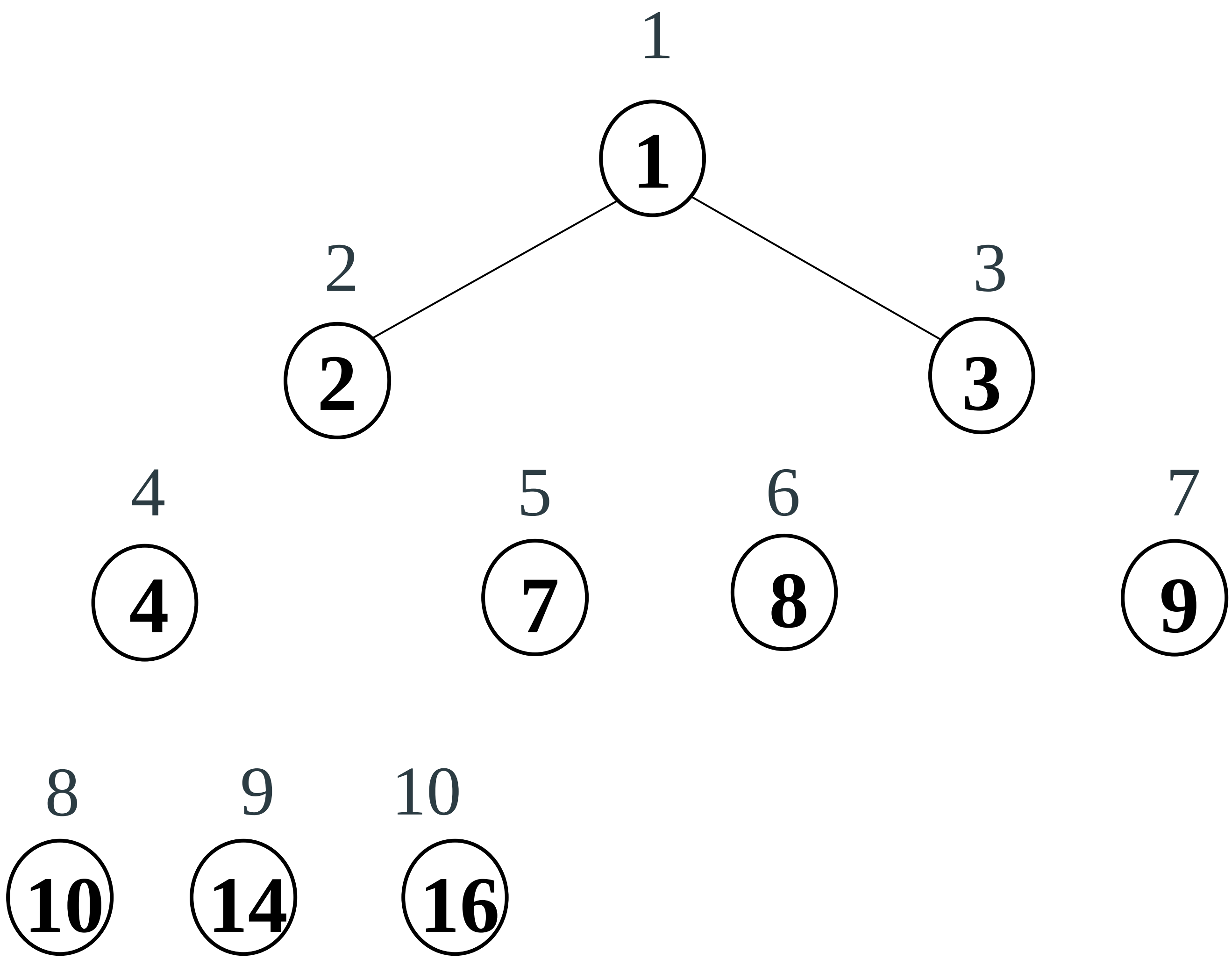
Example: Heapsort (5/8)



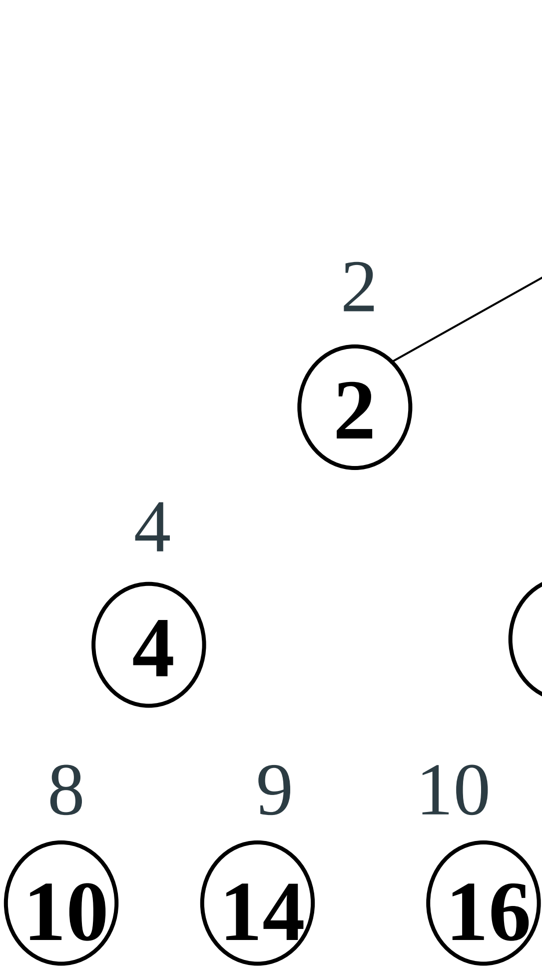
Example: Heapsort (6/8)



Example: Heapsort (7/8)



Example: Heapsort (8/8)



1	2	3	4	7	8	9	10	14	16

Heapsort Algorithm

```
void heapsort(int *a, int n)
{
    int i, temp;
    for (i = n; i >= 2; i--)
    {
        temp = a[i];
        a[i] = a[1];
        a[1] = temp;
        max_heapify(a, 1, i - 1);
    }
}
```

```
int main()
{
    int n, i, x;
    cout<<"enter no of elements of array\n";
    cin>>n;
    int a[20];
    for (i = 1; i <= n; i++)
    {
        cout<<"enter element"<<(i)<<endl;
        cin>>a[i];
    }
    build_maxheap(a,n);
    heapsort(a, n);
    cout<<"sorted output\n";
    for (i = 1; i <= n; i++)
    {
        cout<<a[i]<<endl;
    }
    getch();
}
```

Heap & Heap Sort Algorithm

- Video Content
  - An illustration of Heap and Heap Sort.

Heap & Heap Sort Algorithm

Priority Queues

- We can implement the priority queue ADT with a heap. The operations are:
  - Maximum(A) returns the maximum element
  - Extract-Max(A) removes and returns the maximum element
  - Increase-Key(A,i,key) increases value of i^{th} node to key
  - Insert(A,key) inserts element key into A

Extract-Max

$\Theta(1)$

```
HEAP-MAXIMUM(A)
    return A[1]
```

$O(\lg n)$

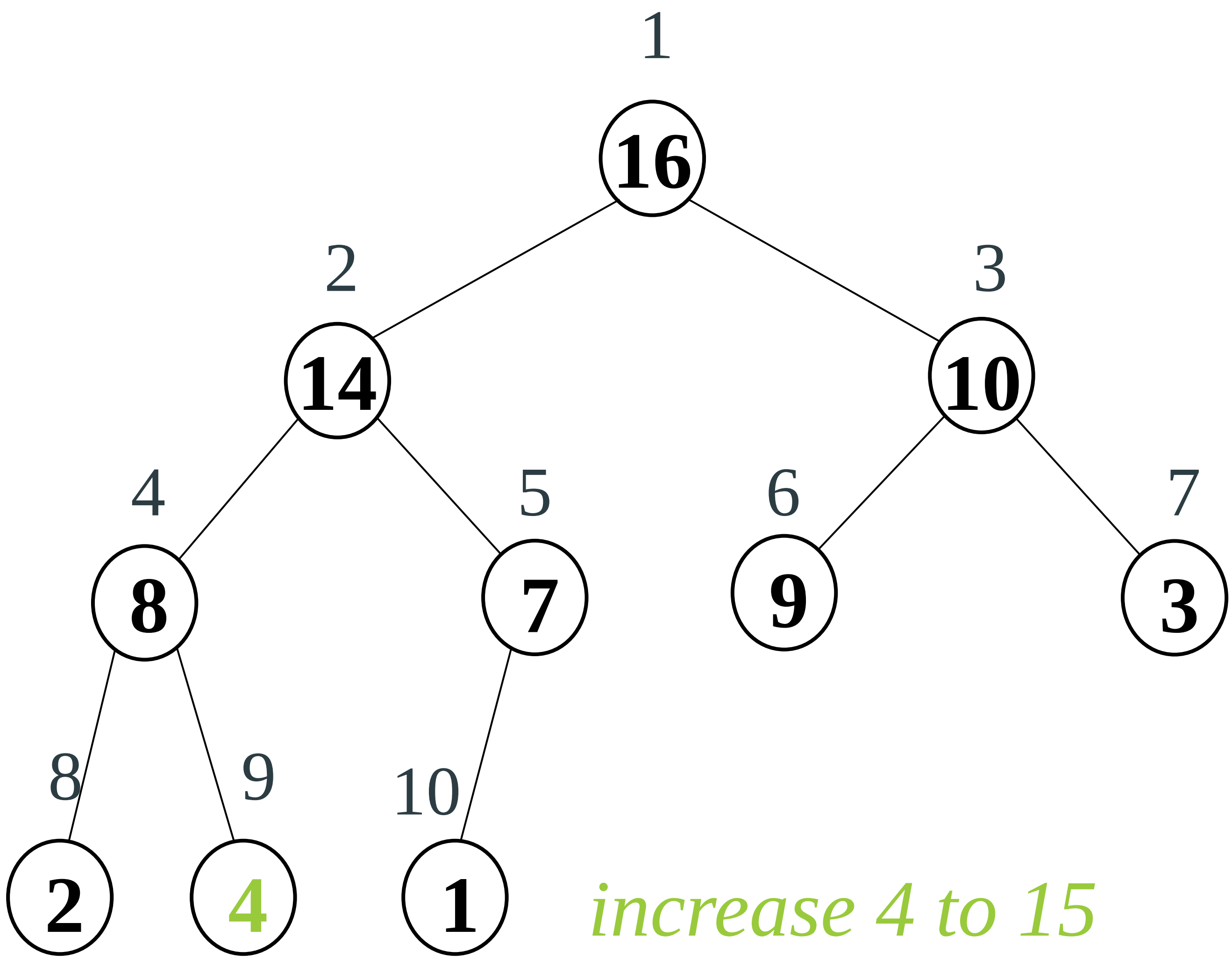
```
HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heap-size]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
    return max
```

Increase-Key

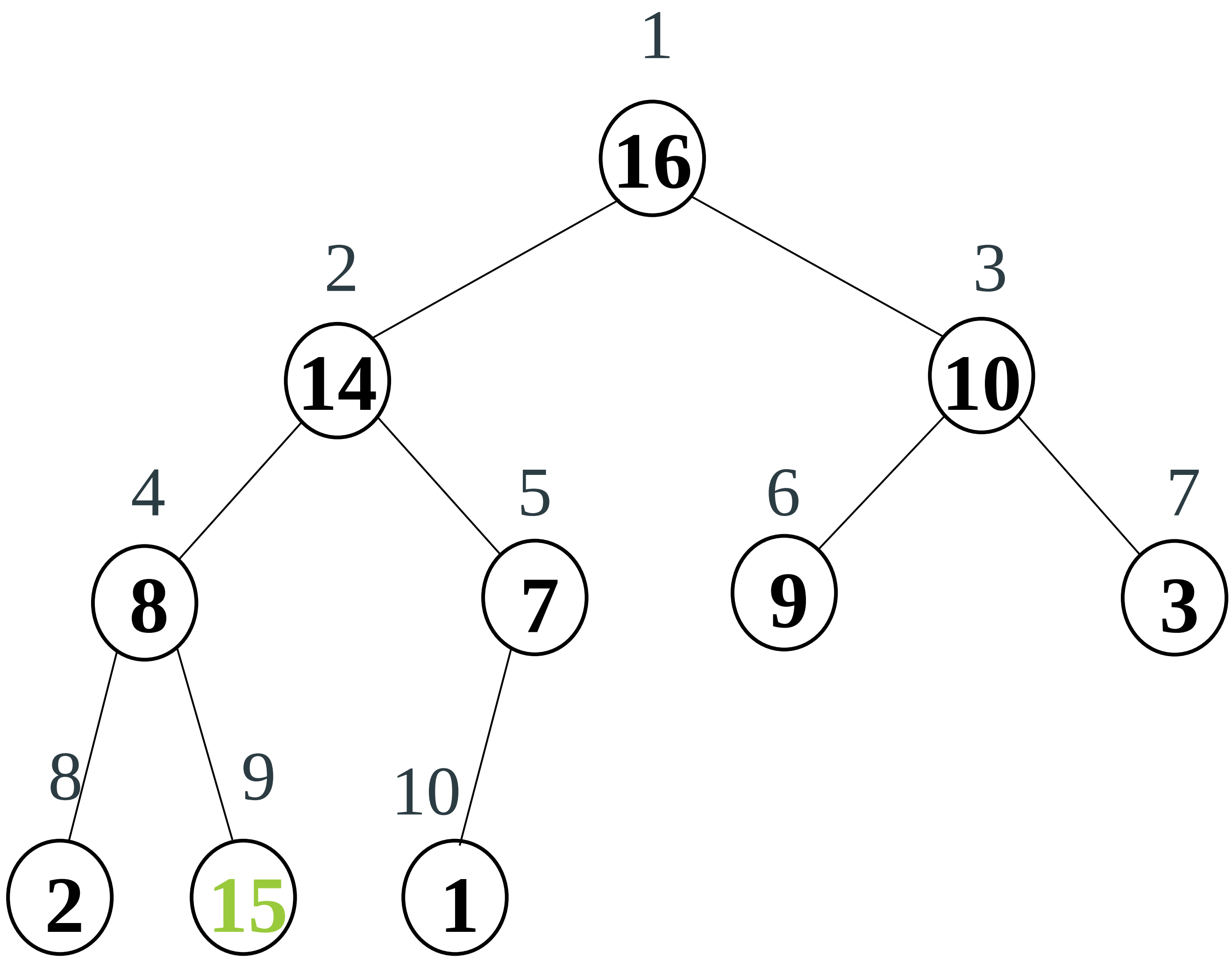
O(lg *n*)

```
HEAP-INCREASE-KEY(A, i, key)
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)
```

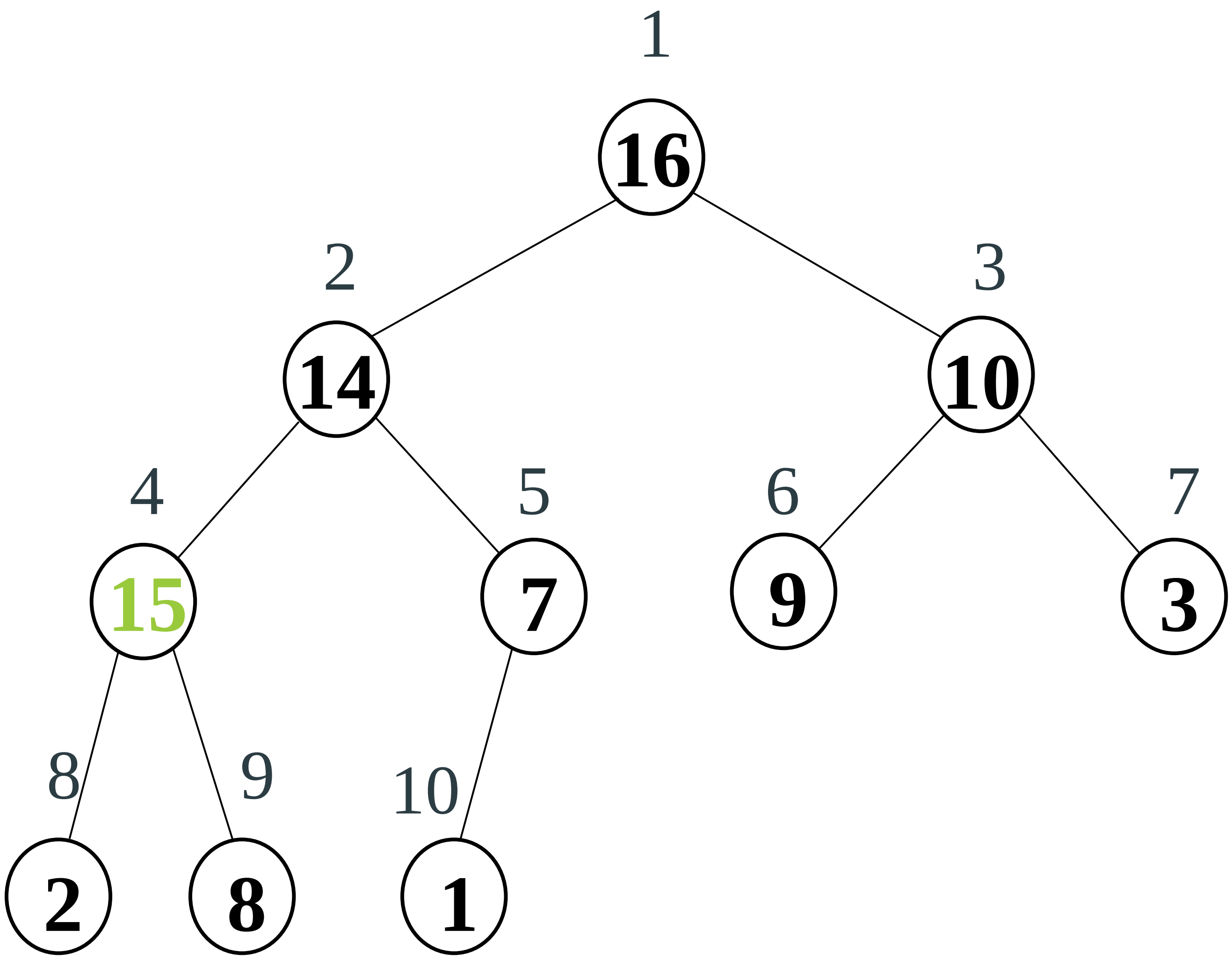
Example: increase key (1/4)



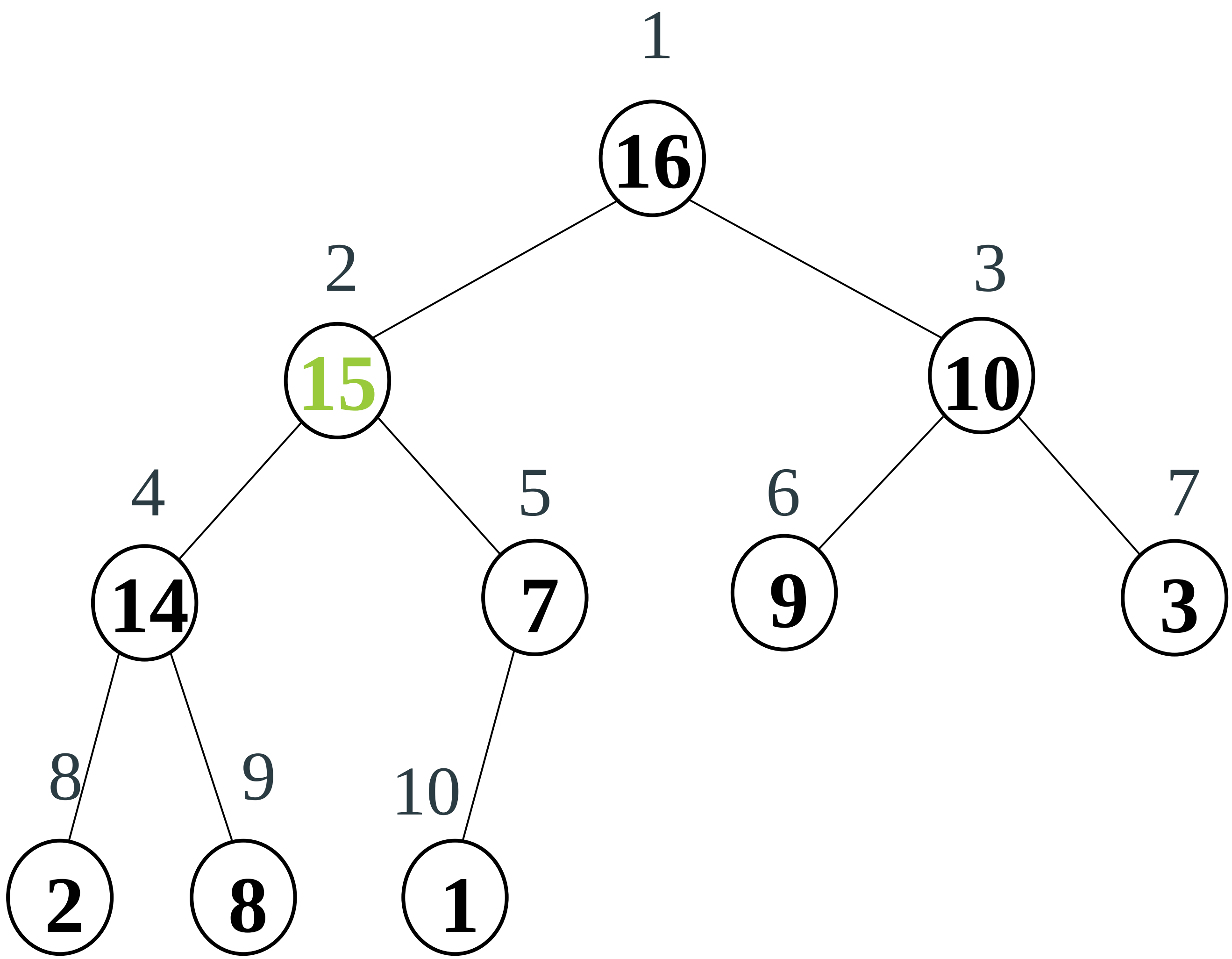
Example: increase key (2/4)



Example: increase key (3/4)



Example: increase key (4/4)



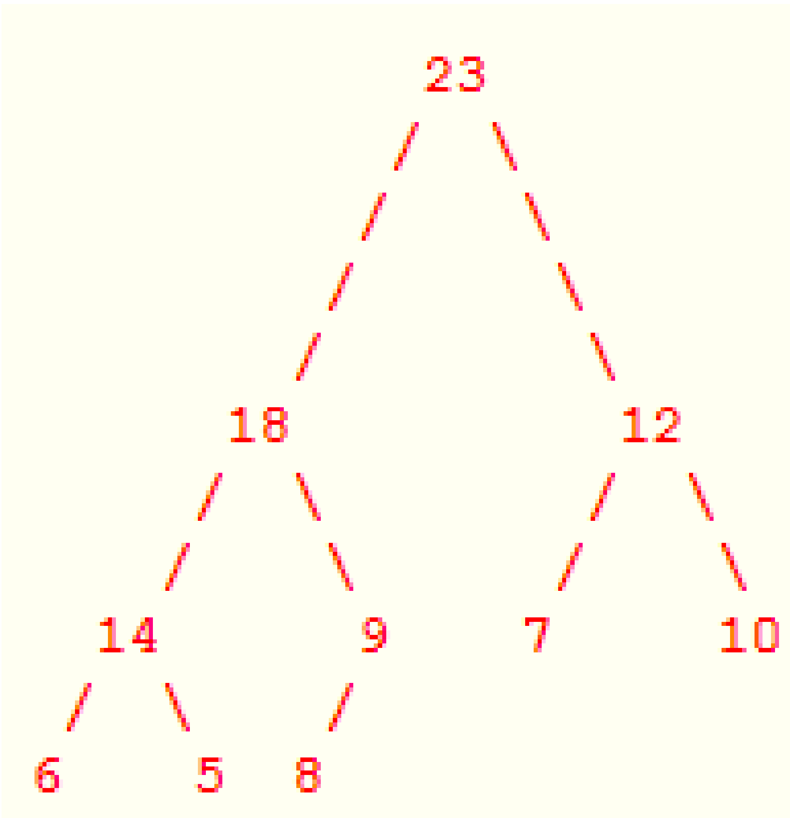
Insert-Max

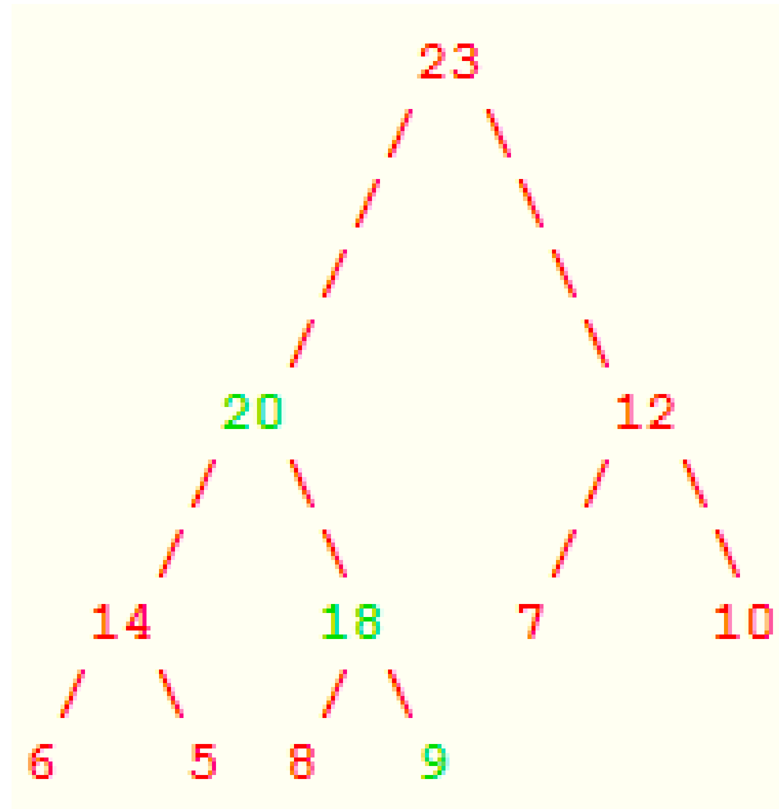
O(lg n) $\text{MAX-HEAP-INSERT}(A, key)$

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 $\text{HEAP-INCREASE-KEY}(A, A.heap\text{-}size, key)$

Practice Problems

- Show the resulting heap after insert 20 into the following heap





Thanks to contributors

Mr. Pham Van Nguyen (2022)

Dr. Thien-Binh Dang (2017 – 2022)

Prof. Hyunseung Choo (2001 – 2022)