IT2070 – Data Structures and Algorithms

Lecture 08

Heap Sort Algorithms

Contents for Today

- **Tree**
- **Binary Tree**
- **Complete Binary Tree**
- **Heaps**
- **Heap Algorithms**

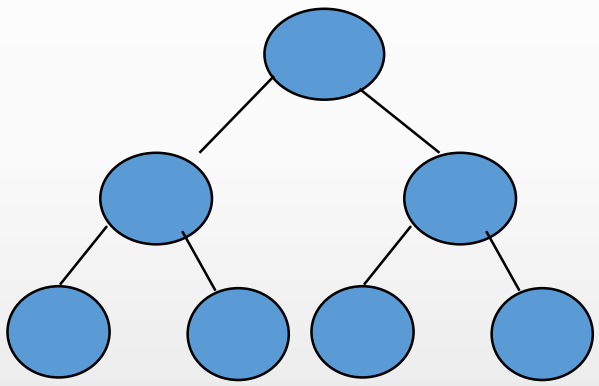
  - Maintaining Heap Property

  - Building Heaps

  - HeapSort Algorithms

Height of a Full Binary Tree

- A Full binary tree of height ***h*** that contains exactly $\mathbf{2^{h+1}-1}$ nodes

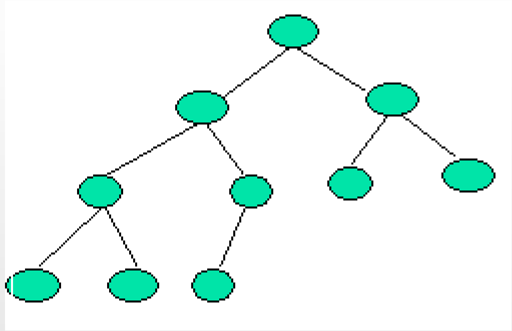


Height, ***h*** = 2,   nodes = $2^{2+1}-1= 7$

Height of a complete binary tree

Height of a complete binary tree that contains ***n*** elements is $\lfloor \mathbf{log_2(\textit{n})} \rfloor$

Example



- Above is a Complete Binary Tree with height = 3
- No of nodes: n = 10
- Height = $\lfloor \mathbf{log_2(\textit{n})} \rfloor = \lfloor \mathbf{log_2(10)} \rfloor$ **=** 3

Heaps

Heap is an array object that can be viewed as a complete binary tree. There are two kinds of heaps

max heaps and **min heaps**

In both case, values in the nodes satisfy **Heap Property** which depend on the kind of heap

max-heap → max-heap property:

The value of each node is greater than or equal to those of its children.

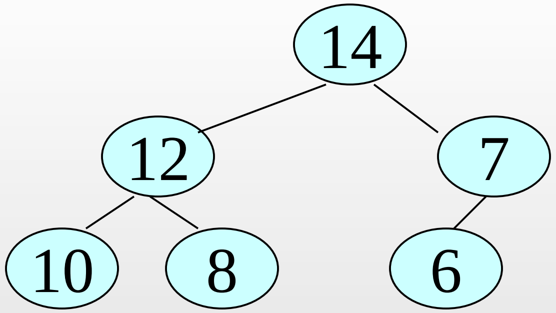
min-heap → min-heap property:

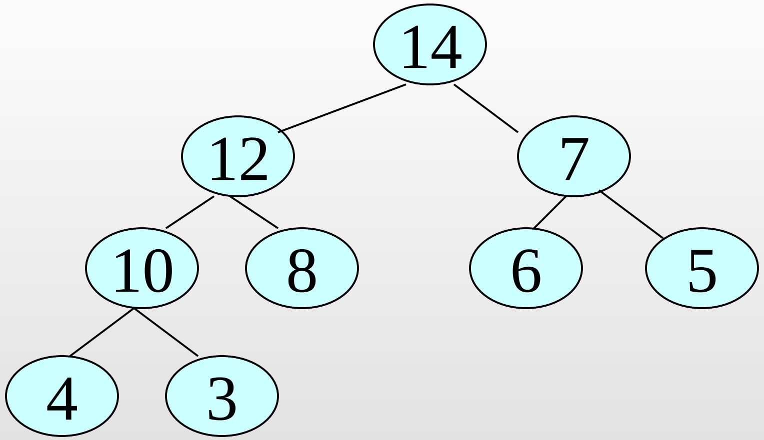
The value of each node is less than or equal to those of its children.

Max-heaps are used in heapsort algorithm

Heaps

Complete Binary Tree with the **max-heap property - examples**

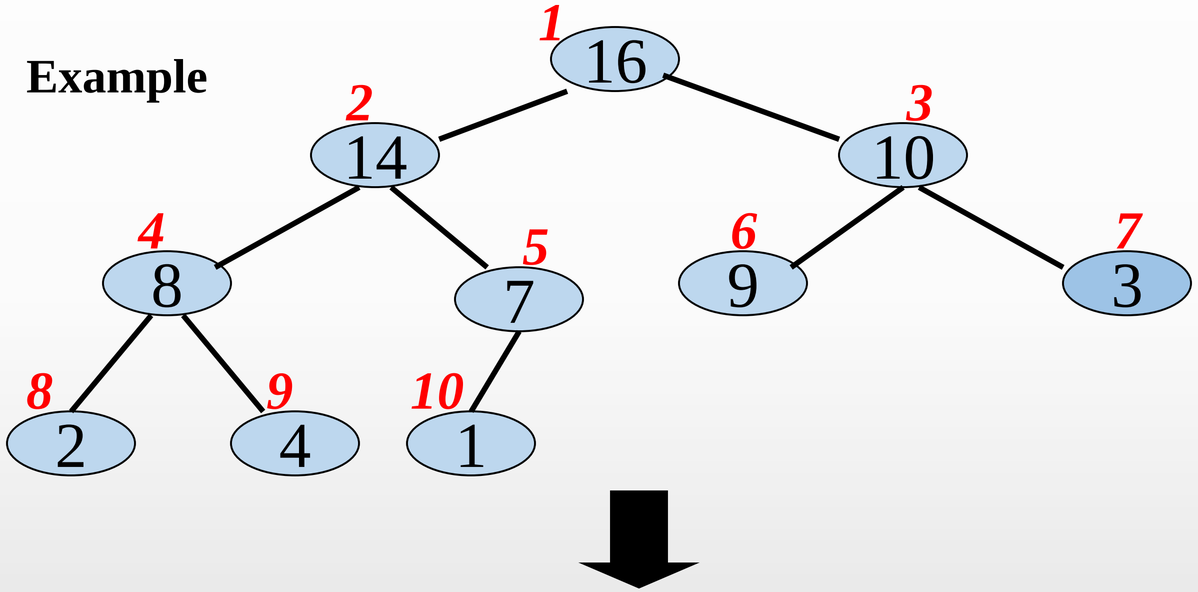




Heaps (contd.)

- A heap can be represented in a one-dimensional array

Example



Max Heap

A

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

Array

Heaps (contd.)

After representing a heap using an array **A**

- Root of the tree is **A[1]**

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

Given node with index *i*,

- ***PARENT(i)*** is the index of parent of ***i***;

  $$PARENT(i) = \lfloor i/2 \rfloor$$

- ***LEFT_CHILD(i)*** is the index of left child of ***i*** ;

  $$LEFT\_CHILD(i) = 2 \times i$$

- ***RIGHT_CHILD(i)*** is the index of right child of ***i***;

  $$RIGHT\_CHILD(i) = 2 \times i + 1$$

Heap Algorithms

- **MAX_HEAPIFY**:

  To maintain max-heap property

  $$\boxed{A[PARENT(i)] \geq A[i]}$$

- **BUILD_MAX_HEAP**

  To build max-heap from an unsorted input array

- **HEAPSORT**

  Sorts an array in place.

MAX_HEAPIFY

- The MAX_HEAPIFY algorithm checks the heap elements for violation of the heap property and restores max-heap property;

$$\boxed{A[\textit{PARENT}(i)] \geq A[i]}$$

- **Input**: An array A and index *i* to the array. *i* =1 if we want to heapify the whole tree. Subtrees rooted at *LEFT_CHILD(i)* and *RIGHT_CHILD(i)* are heaps

- **Output**: The elements of array A forming subtree rooted at *i* satisfy the heap property.

Maintaining the Heap Property

```
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)
```

Example

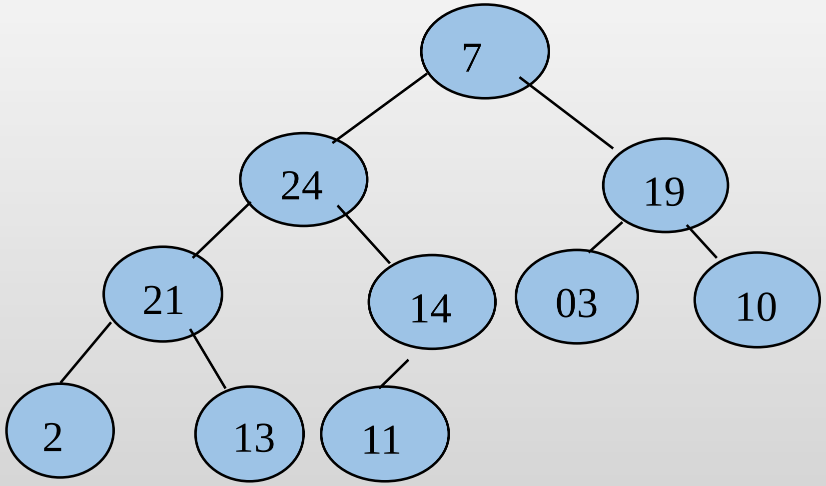
You are given the following array

A

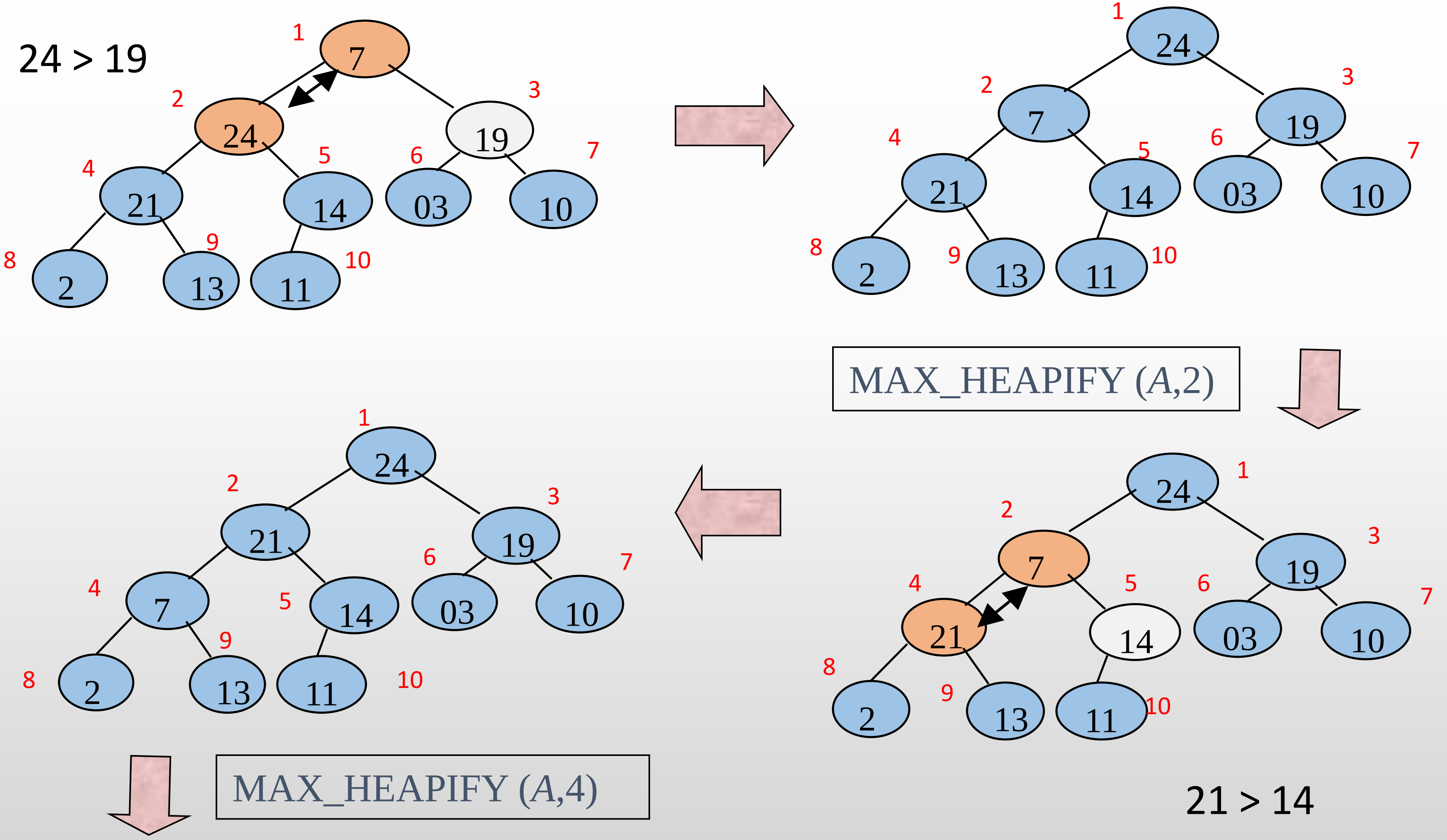
1	2	3	4	5	6	7	8	9	10
7	24	19	21	14	03	10	2	13	11

Now we are going to maintain the max-heap property

Drawing a heap would make our work easy



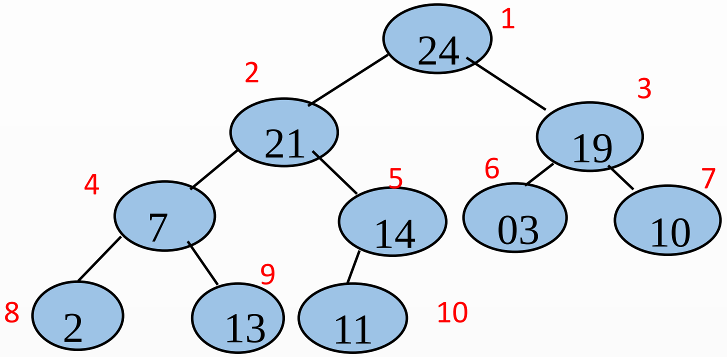
MAX_HEAPIFY (A,1)

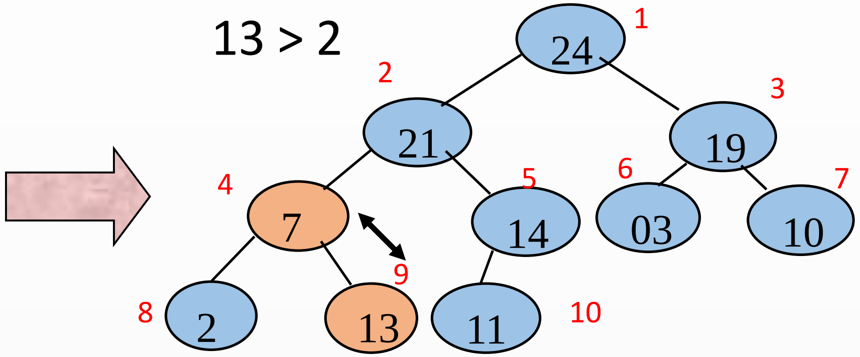


To be contd.

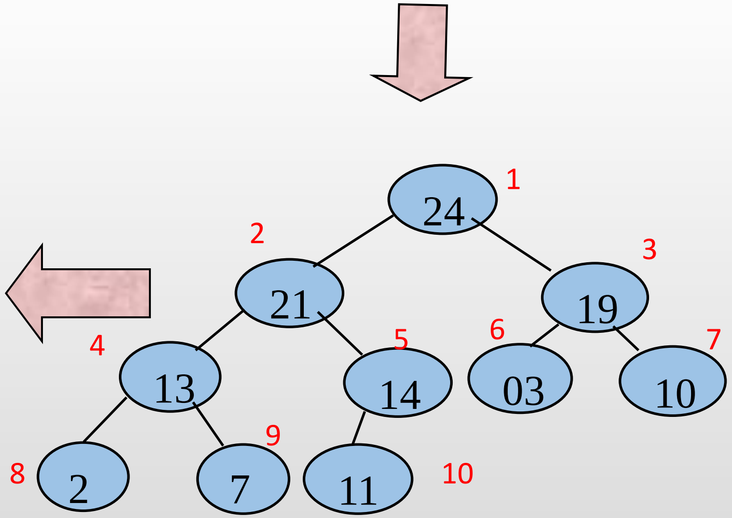
SLIIT
Discover Your Future

MAX_HEAPIFY (*A*,1) (contd.)





MAX_HEAPIFY (*A*,4)

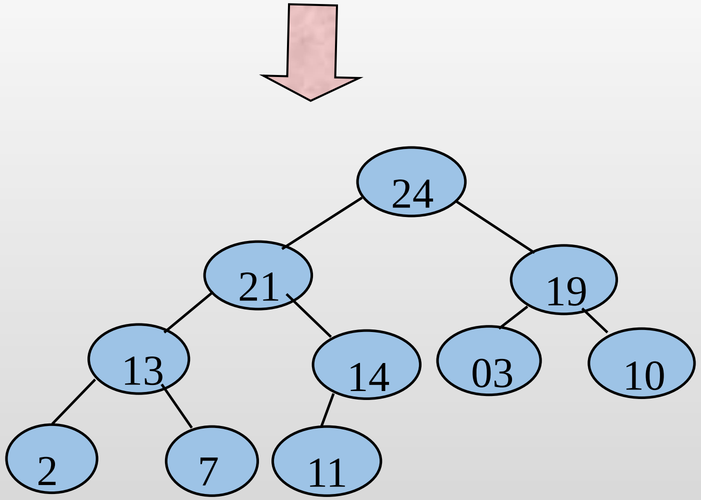


<u>Important point</u> Although we represent this process using a heap actually all the task in done on the input array

Resulting Heap

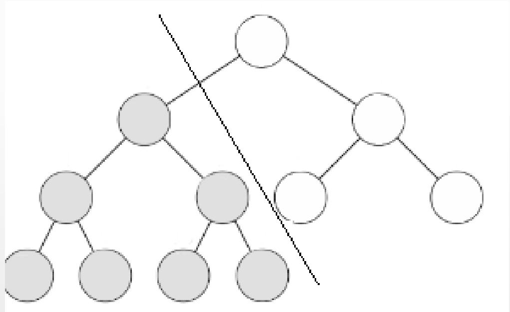
Array view of MAX_HEAPIFY Algorithm

7	24	19	21	14	03	10	02	13	11
24	7	19	21	14	03	10	02	13	11
24	21	19	07	14	03	10	02	13	11
24	21	19	13	14	03	10	02	07	11



Analysis of Heapify Algorithm.

- The running time of MAX-HEAPIFY on a subtree of size *n* rooted at given node *i* is the Θ(1) time plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node *i*.
- The children's subtrees -the worst case occurs when the last row of the tree is exactly half full



- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height *h* as $O(h)$.

The solution to this recurrence is

$$T(n) = O(\lg n)$$

SLIIT
Discover Your Future

BUILD_HEAP

Input **:** An array A of size *n* = A.length , A.heap_size
Output **:** A heap of size *n*

BUILD_MAX_HEAP (A)

1. *A.heap_size* =A.*length*
2. **for** *i* = $\lfloor$A.*length*/2$\rfloor$ **downto** 1
3. MAX_HEAPIFY(*A*,*i*)

Exercise: We are given the following unordered array to build the heap.

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

Solution

Step1

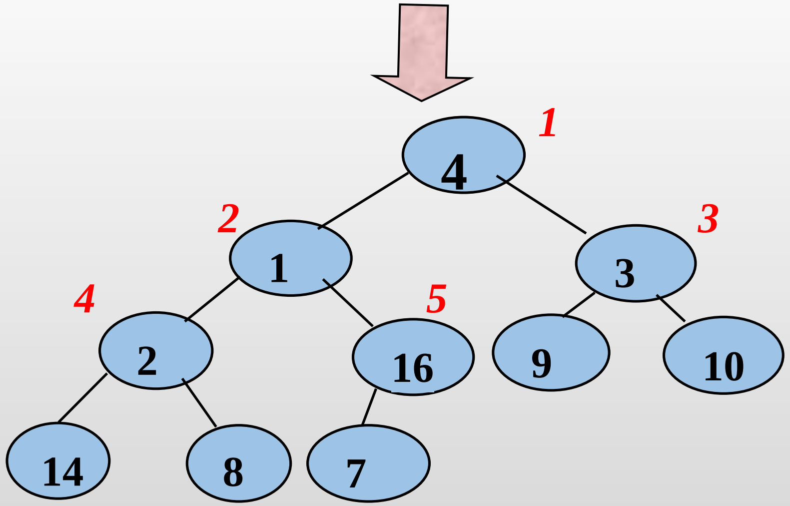
$i = \lfloor length[A]/2 \rfloor = \lfloor 10/2 \rfloor = 5$

MAX_HEAPIFY(A,5)

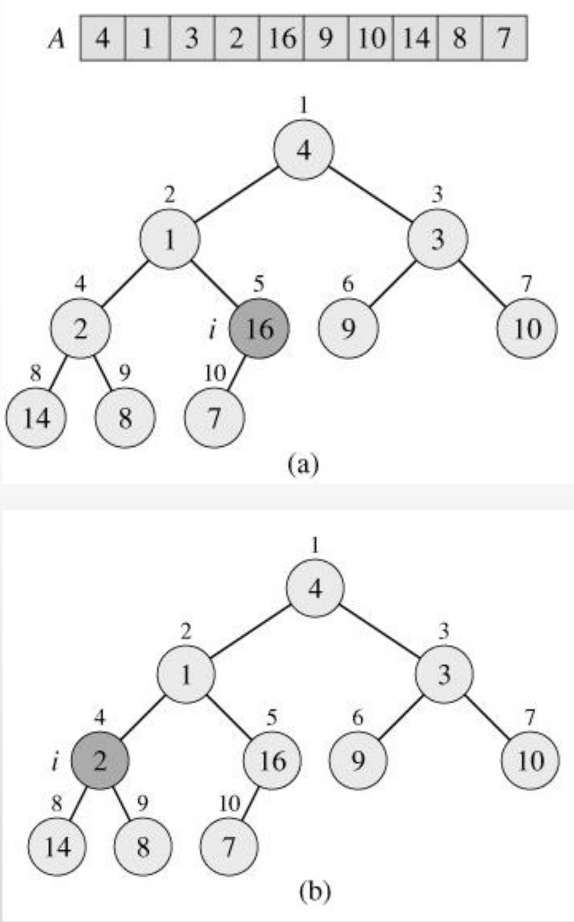
***i* =5**

A

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

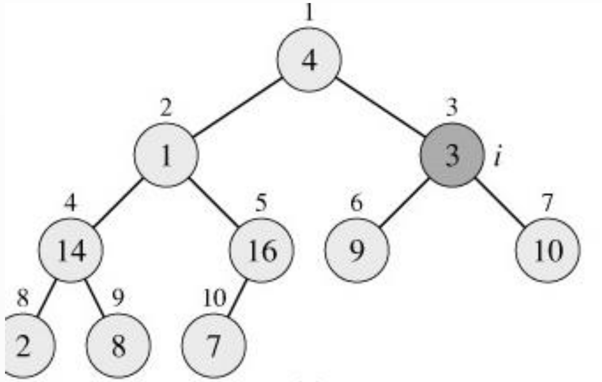


Solution (Contd.)



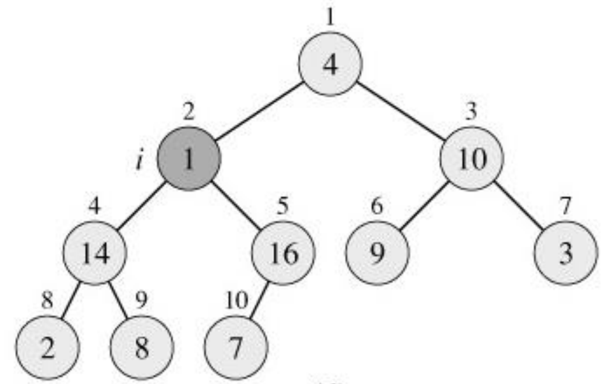
loop index *i* refers to node 5

loop index *i* for the next iteration refers to node 4



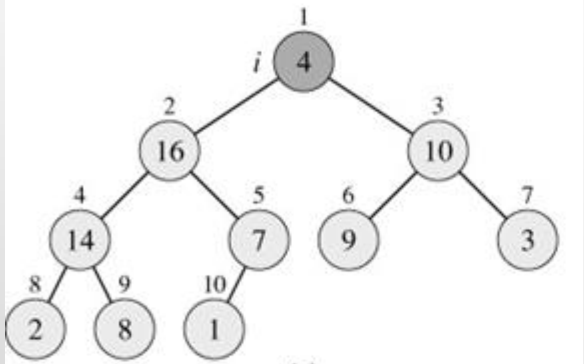
(c)

loop index *i* refers to node 3



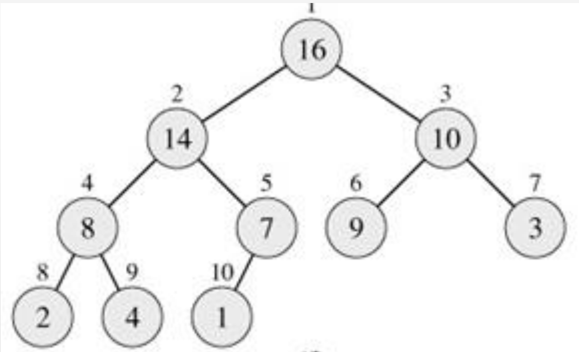
(d)

loop index *i* refers to node 2



(e)

loop index *i* refers to node 1



(f)

max-heap after BUILD-MAX-HEAP finishes.

Analysis of Build Max Heap Algorithm.

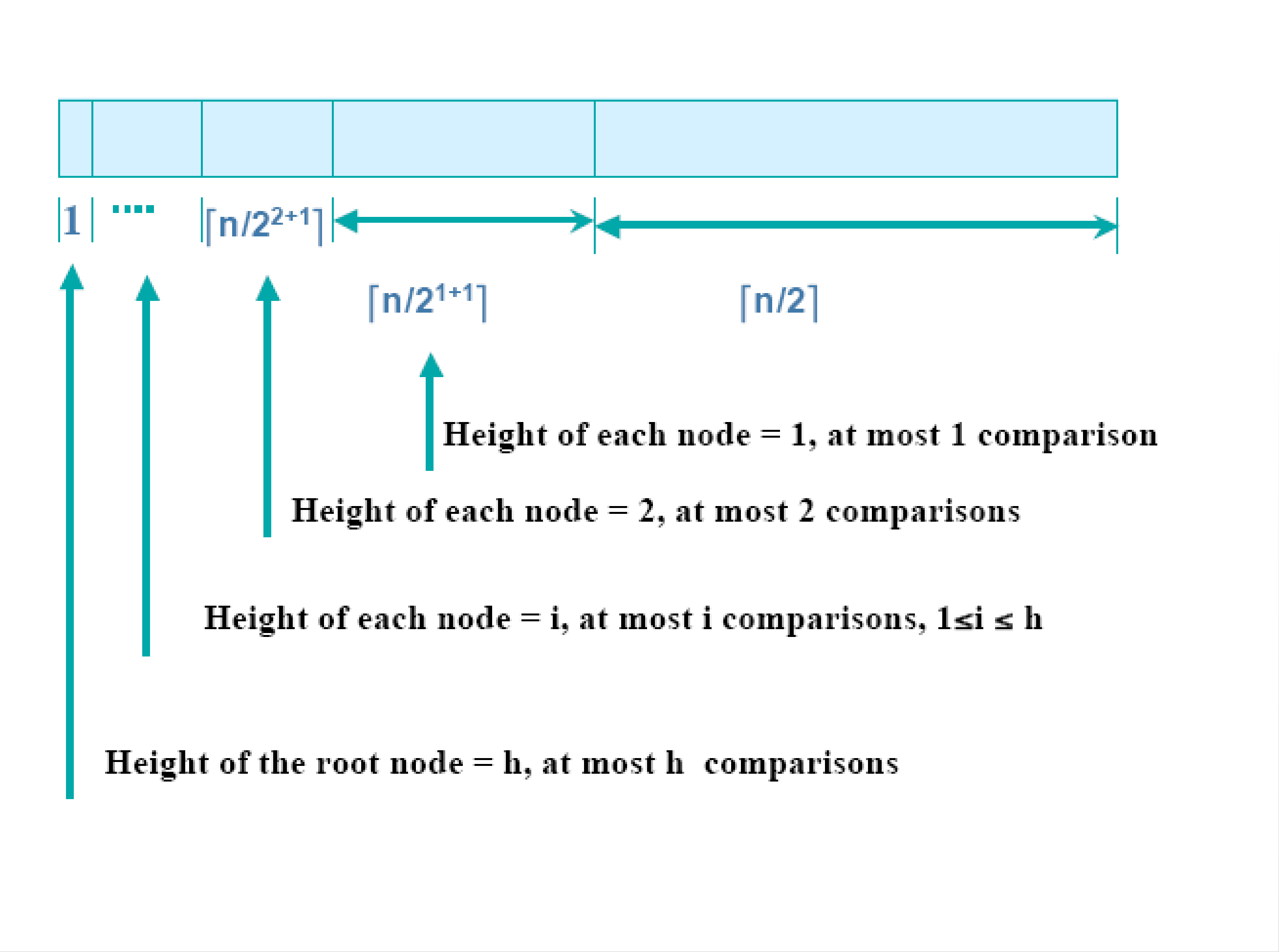
Time for MAX-HEAPIFY to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

- n-element heap has height $\lfloor \lg n \rfloor$

  and

- at most $\lceil n/2^{h+1} \rceil$ nodes of any height h.

Analysis of Build Max Heap Algorithm.



Complexity analysis of Build-Heap (1)

- For each height $0 < h \leq \lg n$, the number of nodes in the tree is at most $n/2^{h+1}$
- For each node, the amount of work is proportional to its height h, $O(h) \rightarrow n/2^{h+1} . O(h)$
- Summing over all heights, we obtain:

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

SLIIT
Discover Your Future

Complexity analysis of Build-Heap (2)

- We use the fact that $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad for\ |x|<1$

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

- Therefore:

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

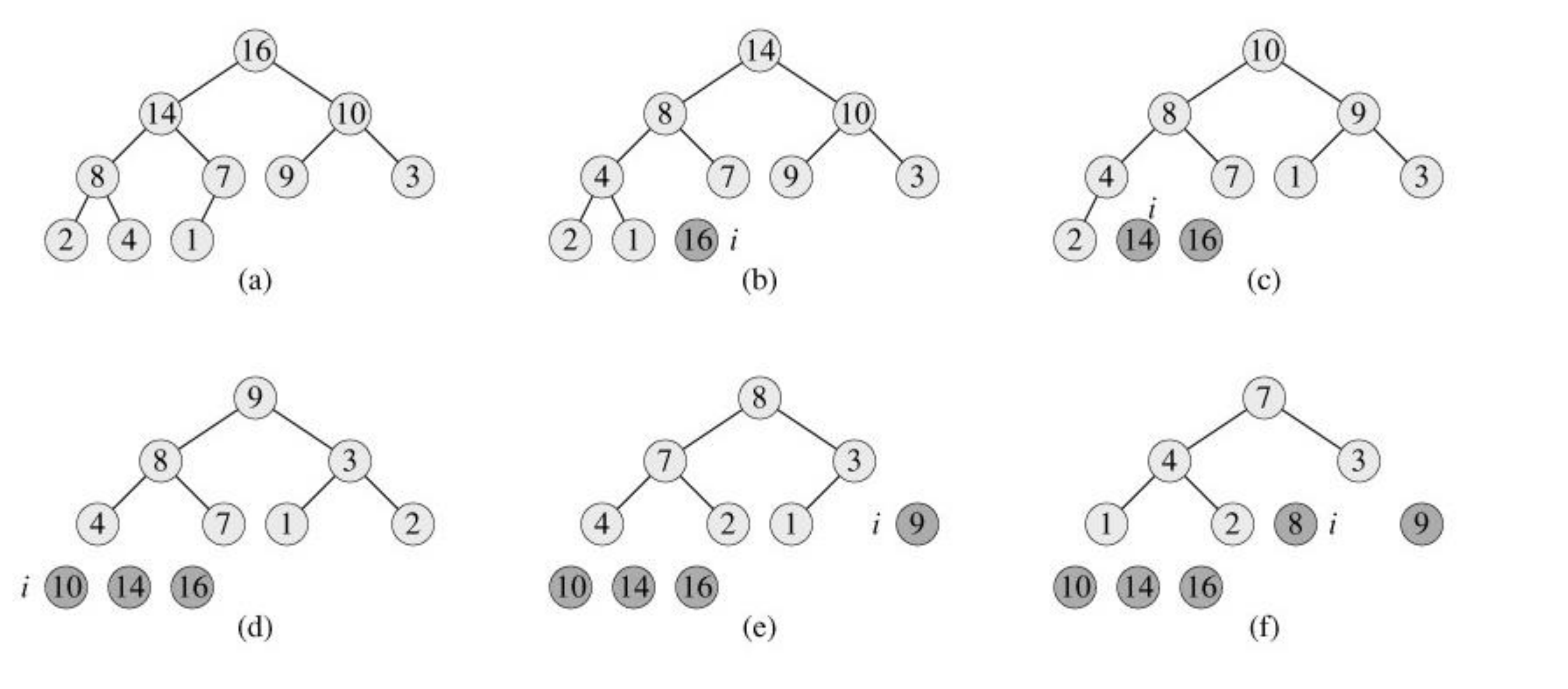
- Building a heap takes only linear time and space!

The HEAPSORT Algorithm

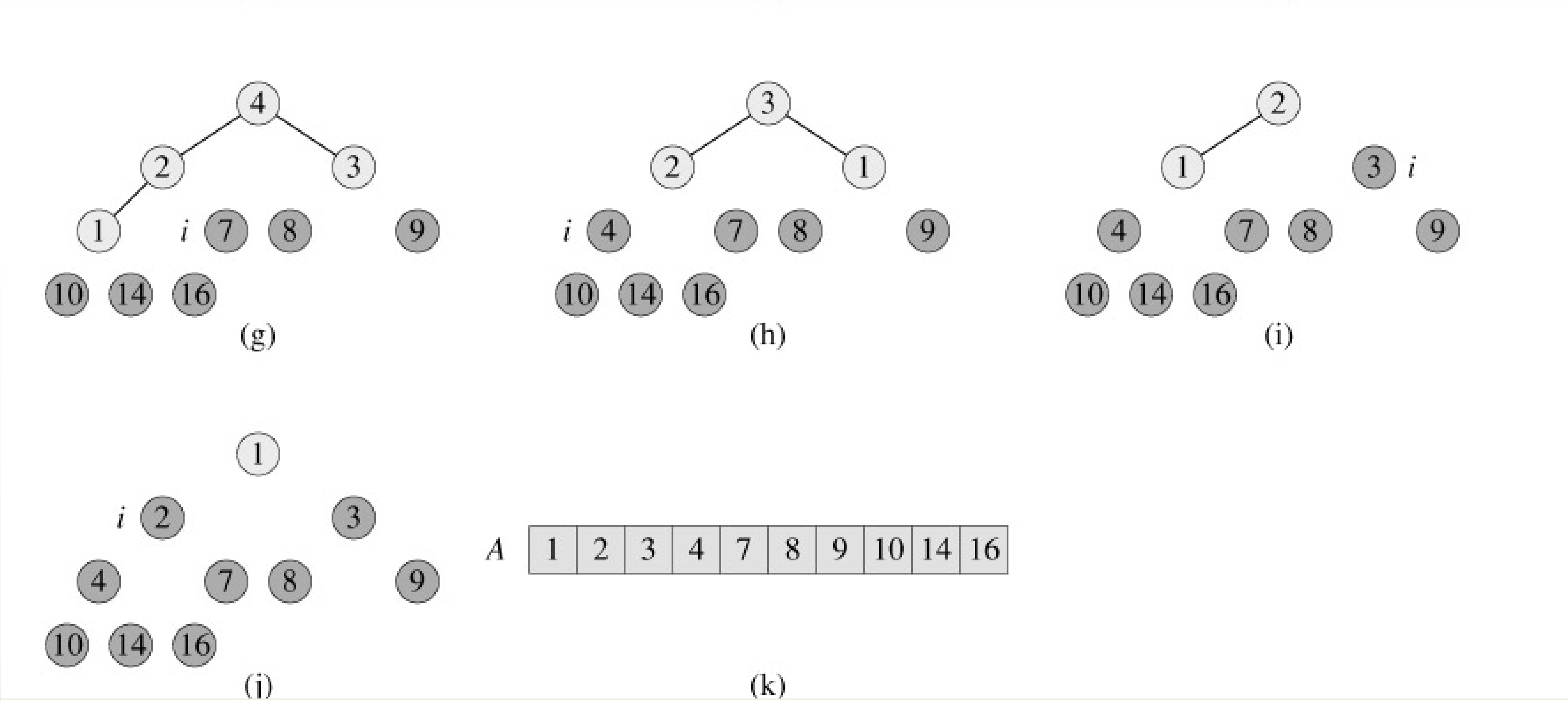
Input : Array A[1…n], n = A.length
Output : Sorted array A[1…n]

```
HEAPSORT(A)
1.      BUILD_MAX_HEAP[A]
2.      for  i = A.length down to 2
3.              exchange A[1] with A[i]
4.              A.heap_size = A.heap_size - 1;
5.              MAX_HEAPIFY(A, 1)
```

The operation of HEAPSORT.



The operation of HEAPSORT.



Heapsort Complexity

Running Time:

Step1 : BUILD_MAX_HEAP takes O(n)

Step 2 to 5 : MAX_HEAPIFY takes O(log n) and there are (n -1) calls

Running Time is O(n log n)

Priority Queues

- Heap data structure itself has many uses.
- One of the most popular applications of a heap: its use as an efficient **priority queue**.
- As with heaps, there are two kinds of priority queues:
  - max-priority queues
  - min-priority queues
- We will focus here on how to implement **max-priority queues**, which are in turn based on max-heaps

Priority queues.

- ***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.

- INSERT(S, x) inserts the element x into the set S. This operation could be written as $S = S \cup \{x\}$.

- EXTRACT-MAX(S) removes and returns the element of S with the largest key.

Priority queues.

- One application of max-priority queues is to schedule jobs on a shared computer.

The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

HEAP_EXTRACT_MAX

HEAP_EXTRACT_MAX(A[1 .. n])

This will remove the maximum element from heap and return it

Input : heap(A)

Output : Maximum element or root, heap(A[1..n-1])

```
1. if A.heap_size >= 1
2.      max = A[1]
3.      A[1] = A[A.heap_size]
4.      A.heap_size = A.heap_size -1
5.      MAX_HEAPIFY(A,1)
6.      return max
```

Running time : O(log n)

HEAP_INSERT

HEAP_INSERT(A, key)

This will add a new element to the heap

Input : heap(A[1..n]), key - the new element

Output : heap(A[1..n+1]), with k in the heap

1. A.heap_size = A.heap_size + 1
2. i = A.heap_size // assume A[i] = - ∞
3. while i > 1 and A[PARENT(i)] < key
4.   A[i] = A[PARENT(i)]
5.   i = PARENT(i)
6. A[i] = key

Running time : O(lg n)

Summary

- Complete binary Tree
- Heap property
- Heap
- Maintaining heap Property(HEAPIFY)
- Building Heaps
- HeapSort Algorithm
- Priority queues.
- Heap Extract Max.
- Heap Insert.