SLIIT ACADEMY

BSc (IT)

Year 2, Semester 1

Design and Analysis of Algorithms

Heap Sort Algorithm

Anuruddha Abeysinghe

anuruddha.a@sliit.lk

Contents for Today

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

  - Maintaining Heap Property

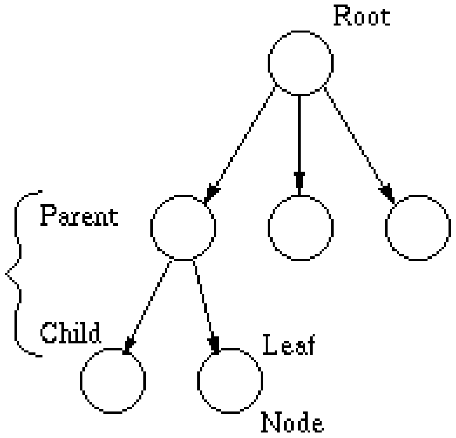
  - Building Heaps

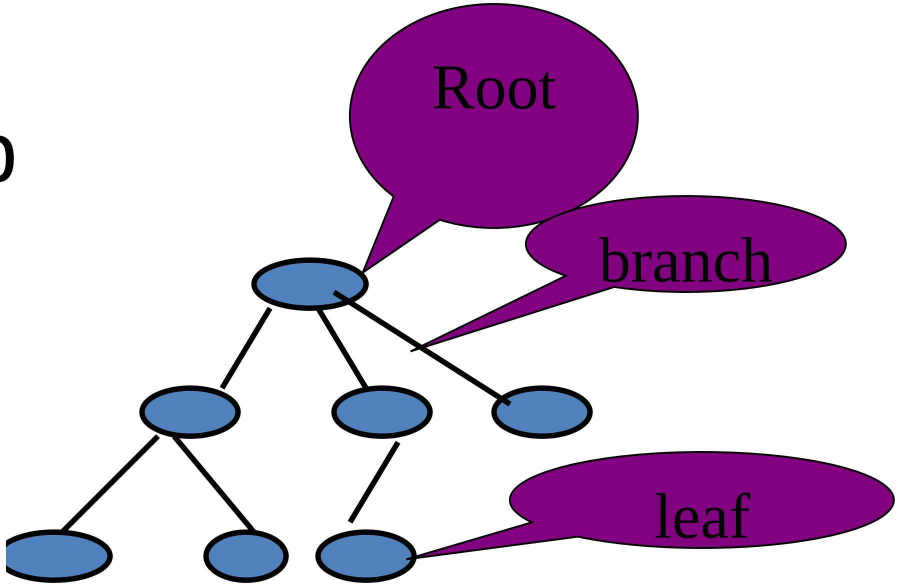
  - HeapSort Algorithms

Tree

- A tree is a <u>connected, acyclic, undirected graph</u>.
- Has components named,
    - root
    - branches
    - leaves
- Drawn with root at the top

Examples





Trees (Contd.)

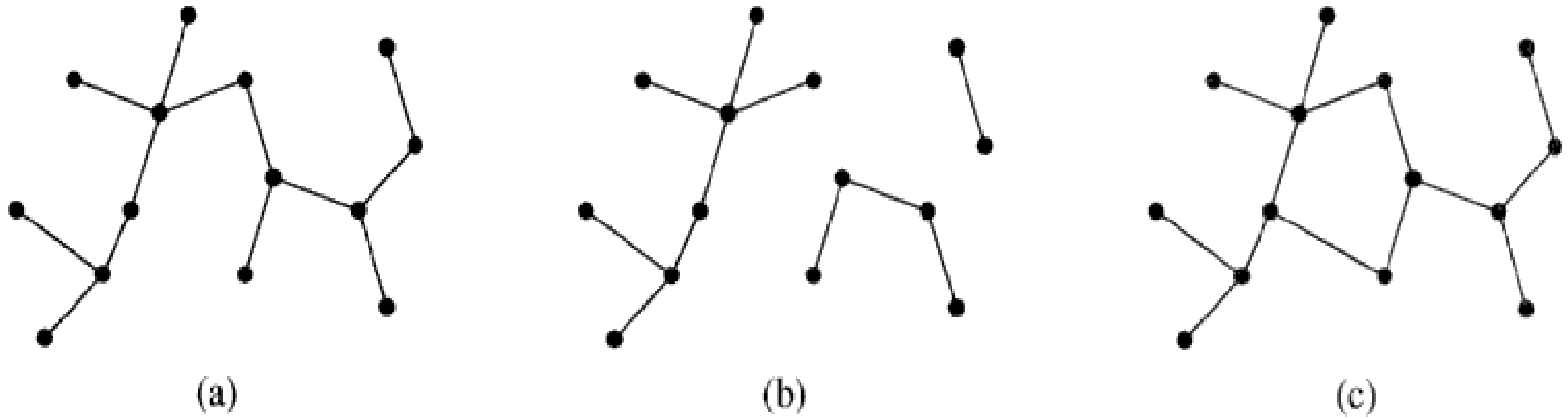
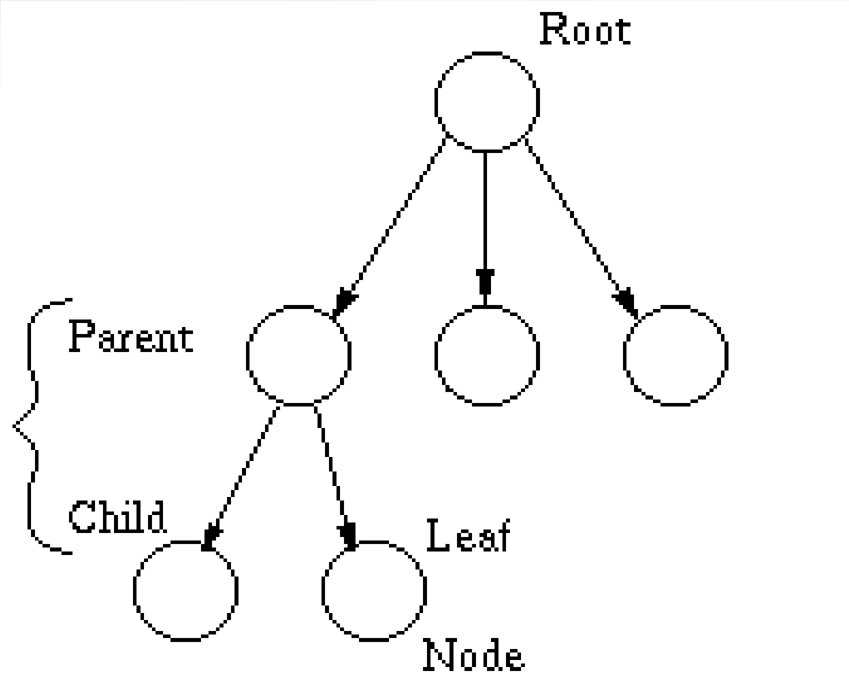


Figure B.4 **(a)** A free tree. **(b)** A forest. **(c)** A graph that contains a cycle and is therefore neither a tree nor a forest.

Tree Terminology

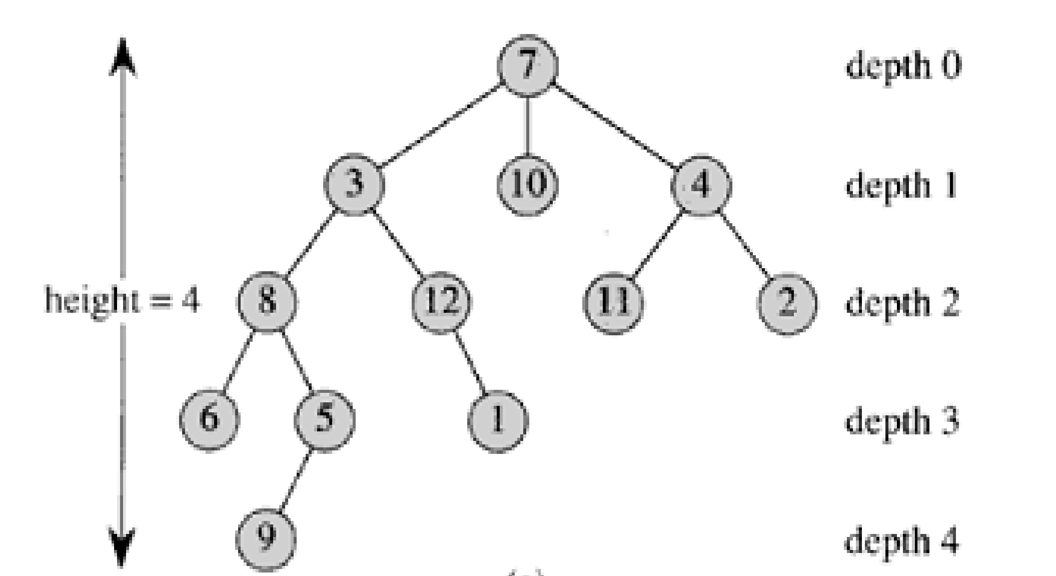


1. Root
2. Node
3. Parent
4. Child
5. Leaf
6. Siblings
7. Degree of a node
8. Depth of a node
9. Height of a tree

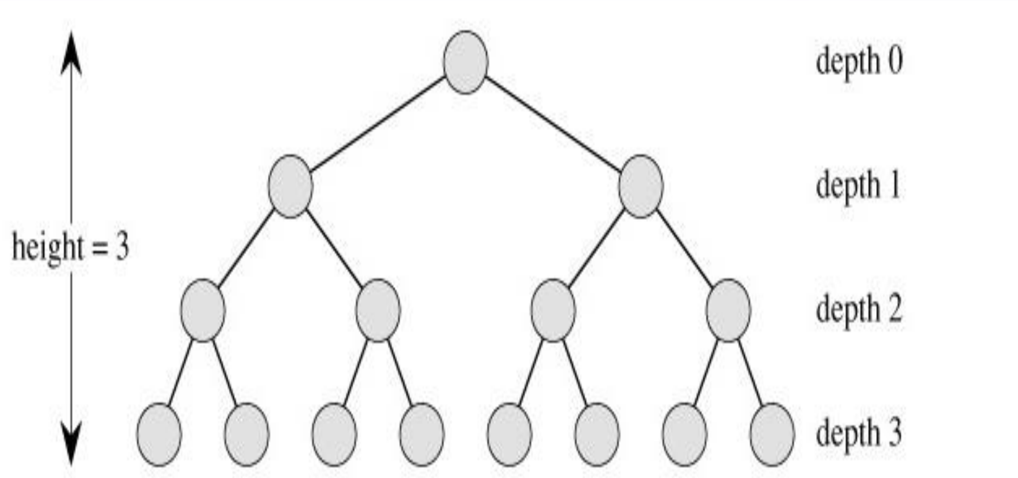
Tree Terminology (Contd.)

- **Root**:It is node with no parent.
- **Node**: It is a vertex of the tree where a data element is stored.
- **Parent** : It is a single node that directly precedes a node.
- **Child** : It is a node that directly follows a node.
- **Leaf** : It is a node with no children, Items at the very bottom of a hierarchical tree structure.
- **Siblings**: The nodes which share same parent
- **Degree of an node:** The number of children it has
- **Depth:** The depth of x in T is the length of the path from the root r to a node x.
- **Height:** The largest depth of any node in a tree is the height.

Tree Terminology (Contd.)

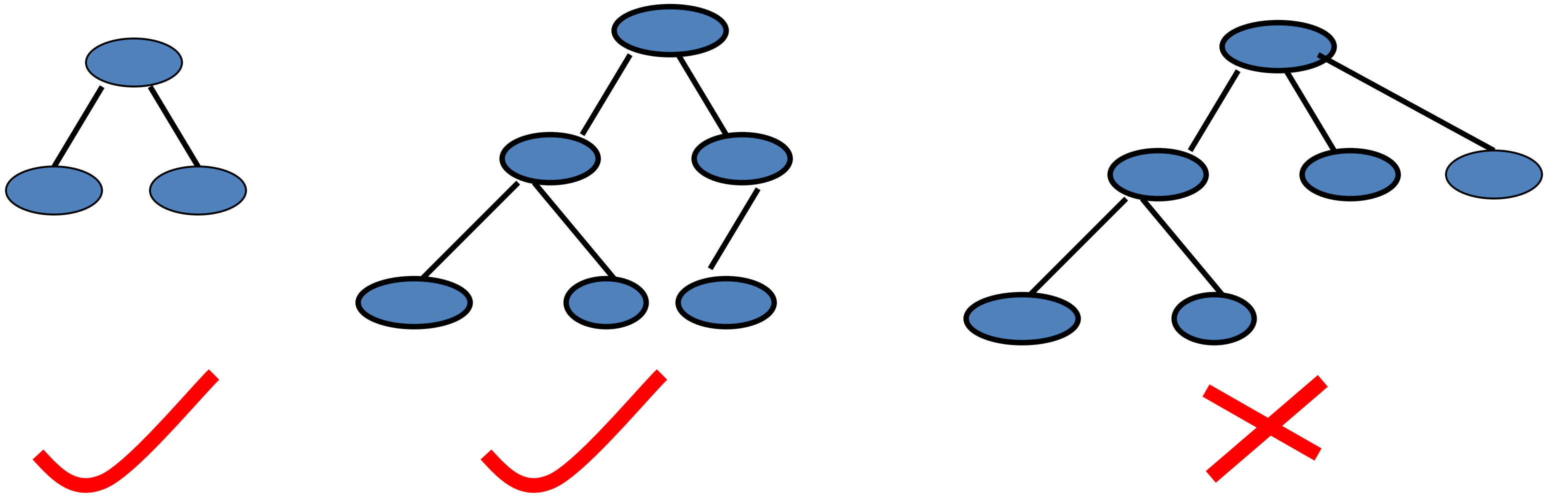


(a)



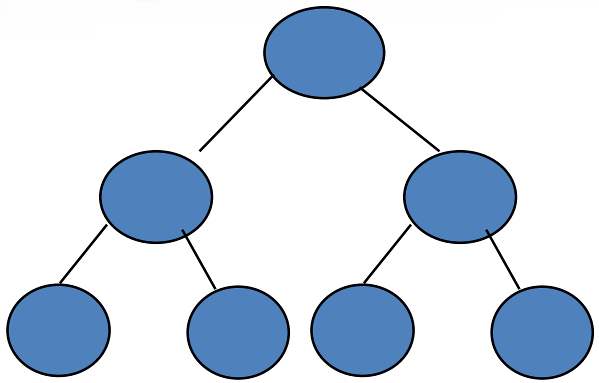
Binary tree

- A **binary tree** is a tree structure in which each node has at most two children.
- Each element in a **Binary Tree** has degree ≤ 2.
- **Examples**



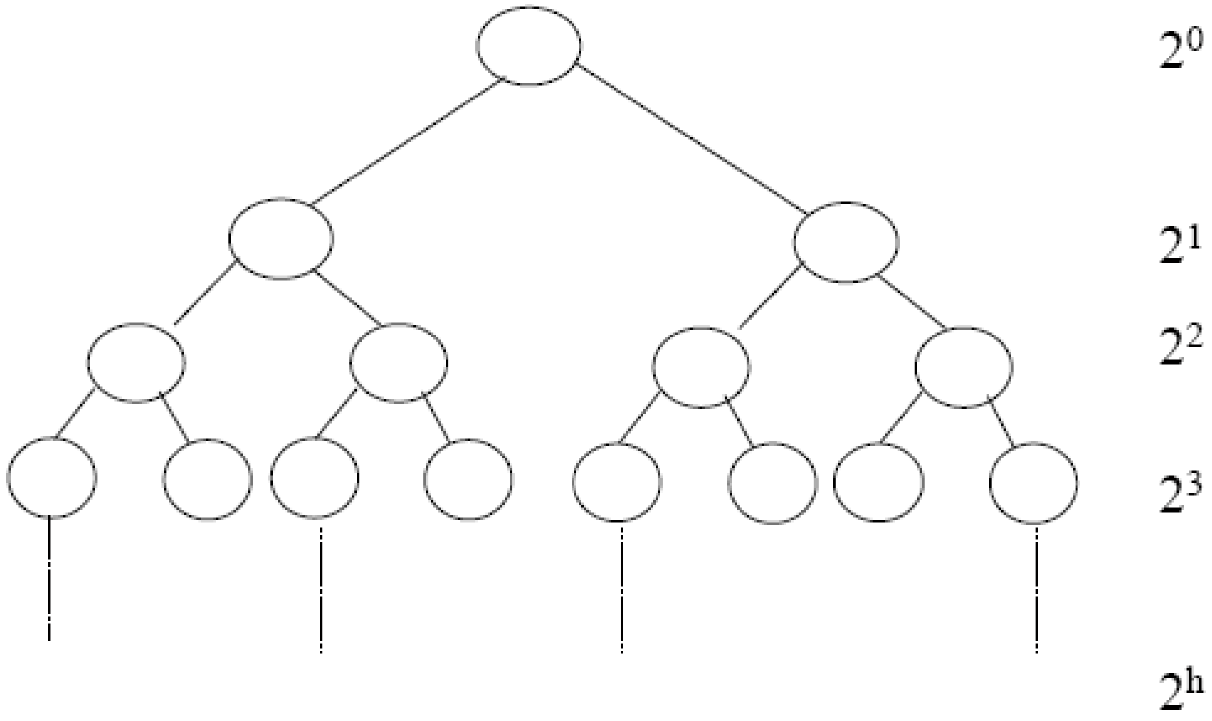
Full Binary Tree

- A Binary tree of height ***h*** that contains exactly **2^{h+1}-1** nodes



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

Binary Heap Tree

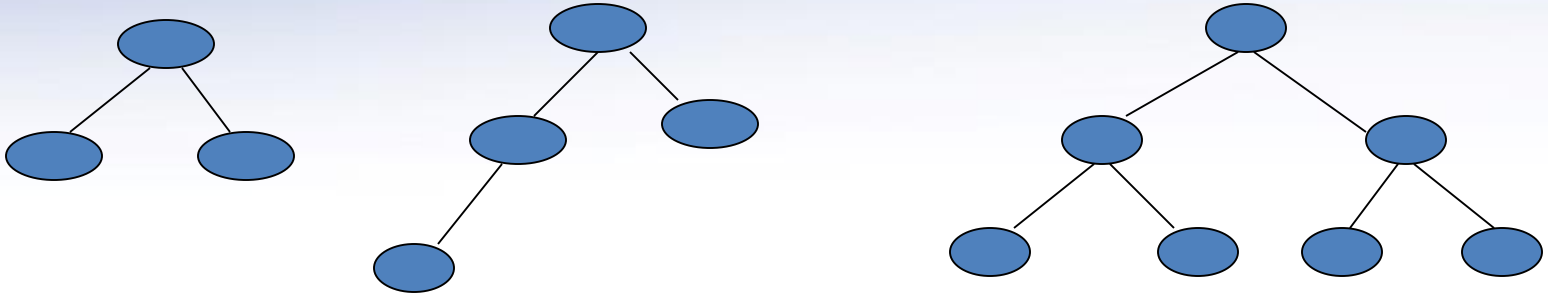


$$n=2^0+2^1+2^2+2^3+\ldots+2^h=2^{h+1}-1$$

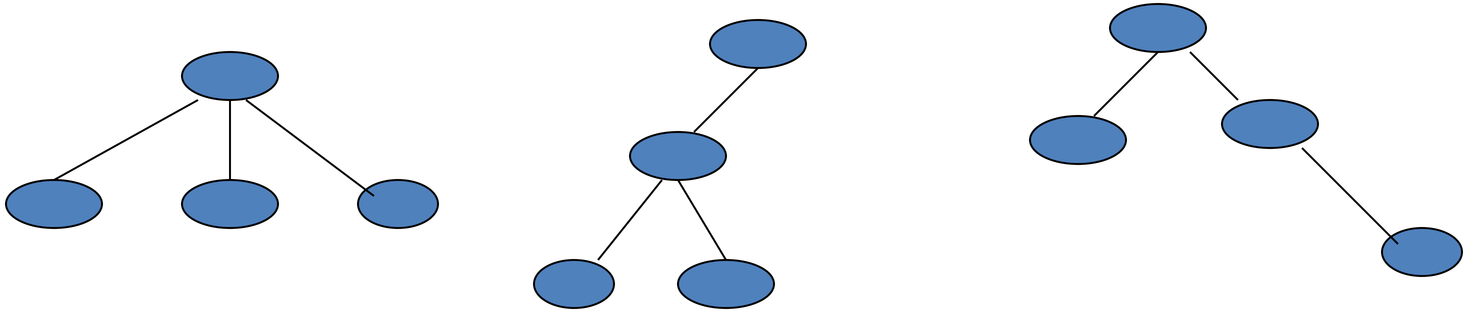
Complete Binary Tree

- It is a Binary tree where each node is either a leaf or has degree $\leq$ **2**.
- Completely filled, except possibly for the bottom level and level above
- Each level is filled from **left to right**.
- All nodes at the lowest level are as far to the left as possible
- Full binary tree is also a complete binary tree.

Examples of Complete Binary Trees



- Followings are not CBTs



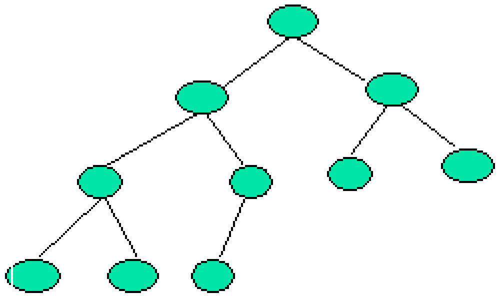
Degree > 2

Not Completely filled

Filled from right to left

Height of a complete binary tree

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

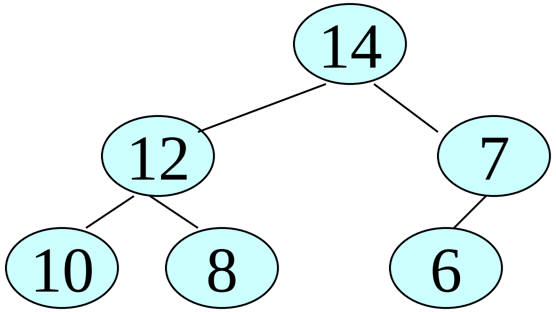


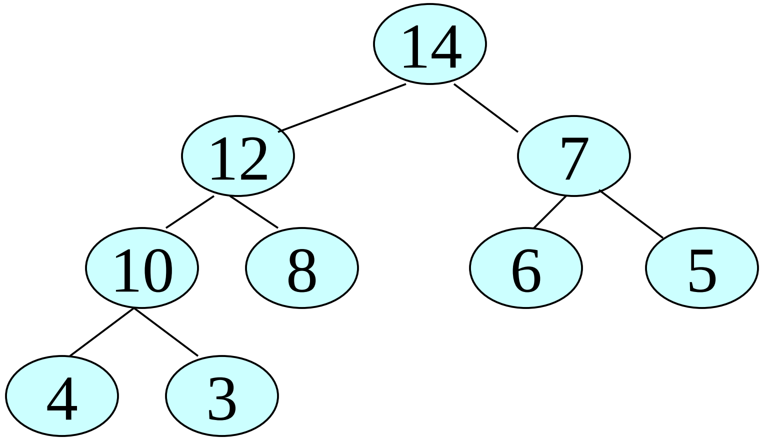
  - Above is a Complete Binary Tree with height = 3
  - No of nodes: n = 10

$\therefore$ Height = $\lfloor \mathbf{log_2(}\boldsymbol{n}\mathbf{)} \rfloor = \lfloor \mathbf{log_2(10)} \rfloor$ **=** 3

Heaps

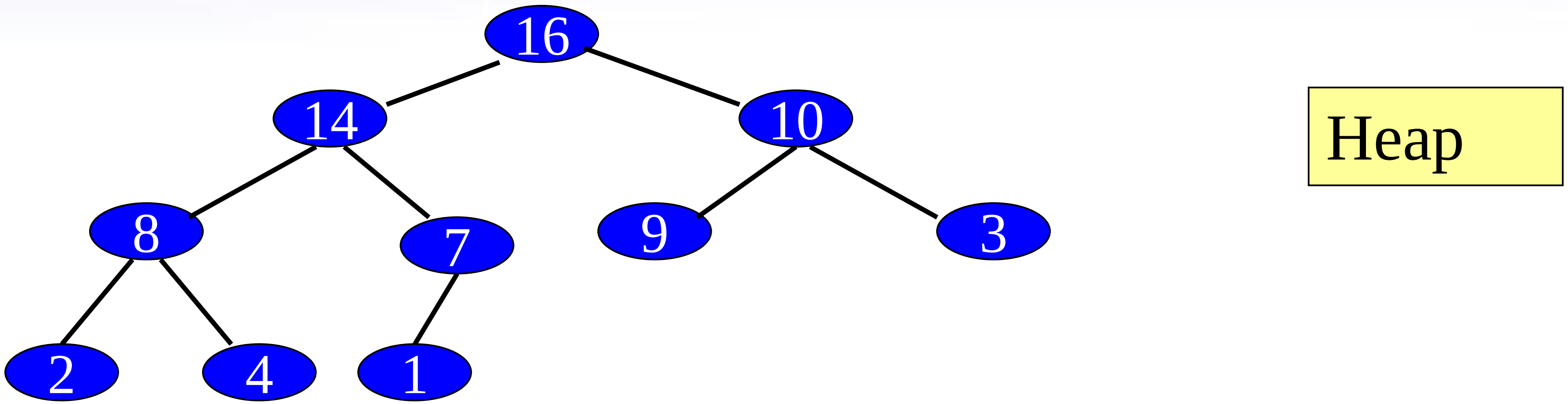
- A **Complete Binary Tree** with the **heap property.**
- **Heap Property:** The value of each node is greater than or equal to those of its children.
- Examples





Heaps (contd.)

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



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 : **A[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×*i***

RIGHT_CHILD(i) is the index of right child of ***i*** :***RIGHT_CHILD(i)*** = $2 \times i + 1$

Heap property

- For max-heaps (largest element at root),

  max-heap property: for all nodes i ,excluding the root,

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

- For min-heaps (smallest element at root),

  min-heap property: for all nodes i ,excluding the root,

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

The HEAPSORT Algorithm

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

Procedure HEAPSORT(A)

```
BUILD_HEAP[A]
for  i ← length[A] down to 2
        Exchange A[1] ↔ A[i]
        heap_size[A] ← heap_size[A]-1;
        HEAPIFY(A,1)
```

Procedure BUILD_HEAP (A)

```
heap_size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        HEAPIFY(A,i)
```

Procedure **HEAPIFY** (*A*,*i*)

```
l ← LEFT_CHILD (i);
r ← RIGHT_CHILD (i);
if l ≤ heap_size[A] and A[ l ] > A[ i ]
        then largest ← l;
        else largest ← i;
if r ≤ heap_size[A] and A[r] > A[largest]
        then largest ← r;
if largest ≠ i
        then exchange A[i] ↔ A[largest]
                HEAPIFY (A,largest)
```

Heap Algorithms

- **HEAPIFY**:

  To maintain heap property

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

- **BUILD_HEAP**

  To build heap from an unsorted input array

- **HEAPSORT**

  Sorts an array in place.

BUILD_HEAP

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

Procedure BUILD_HEAP (A)

1. $heap\_size[A] \leftarrow length[A]$
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. 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

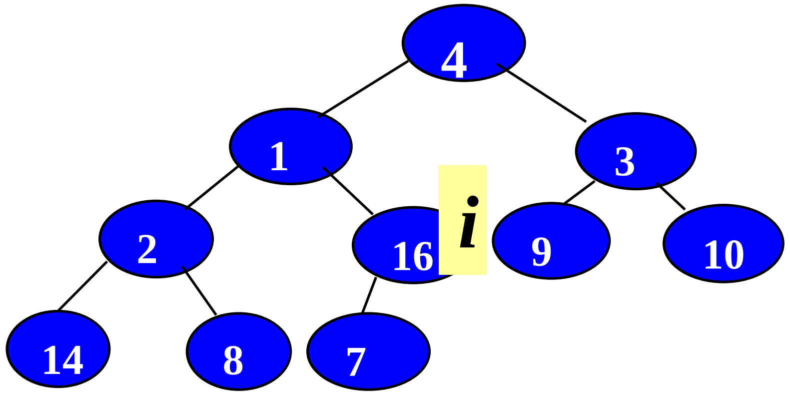
<u>Step1</u>

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

HEAPIFY(A,5)

A

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



HEAPIFY

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

$$\boxed{A[\textit{P}\text{ARENT}(i)] \geq A[i]}$$

- **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

Procedure **HEAPIFY** (A,i)

1. $l \leftarrow$ LEFT_CHILD (i);
2. $r \leftarrow$ RIGHT_CHILD (i);
3. if $l \leq$ heap_size[A] and A[l] > A[i]
4.   then largest $\leftarrow$ l;
5.   else largest $\leftarrow$ i;
6. if $r \leq$ heap_size[A] and A[r] > A[*largest*]
7.   then largest $\leftarrow$ r;
8. if *largest* $\neq$ i
9.   then exchange A[i] $\leftrightarrow$ A[*largest*]
10.   **HEAPIFY** (A,*largest*)

Example

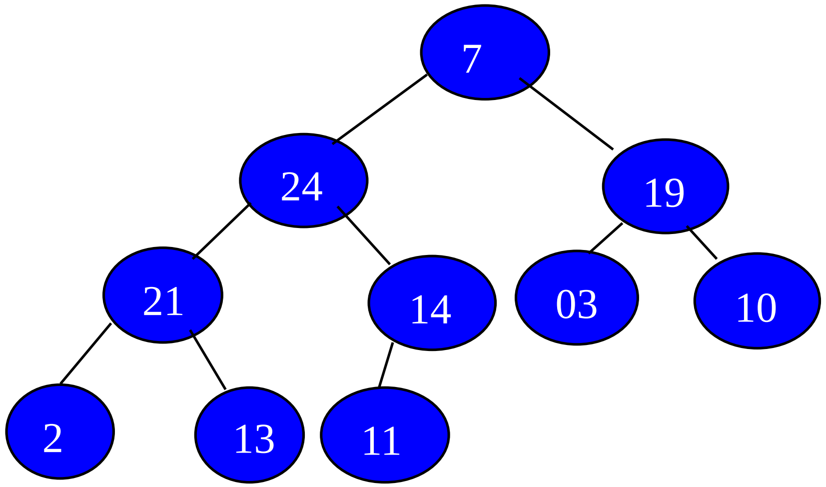
- You are given the following array.

A

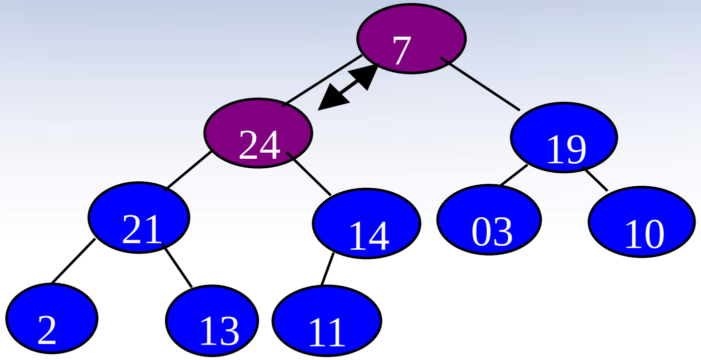
1	2	3	4	5	6	7	8	9	10
7	24	19	21	14	03	10	2	13	11

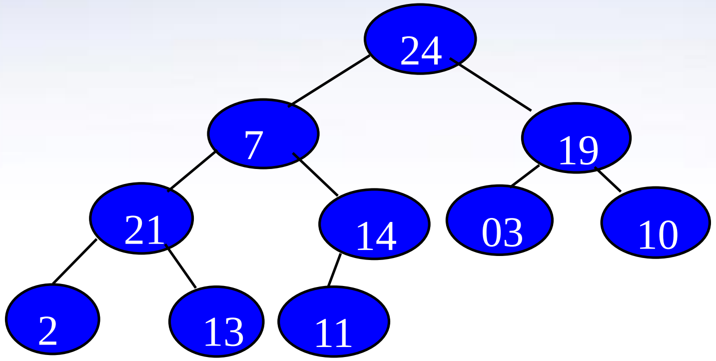
•Now we are going to maintain the heap property

•Drawing a heap would make our work easy

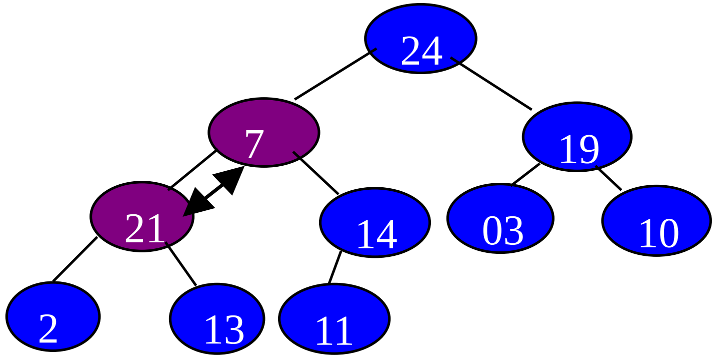


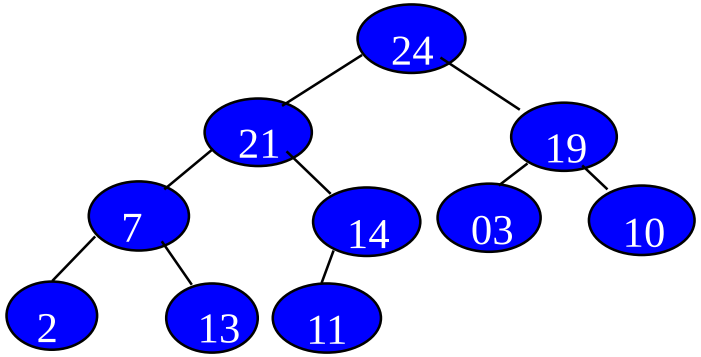
HEAPIFY (*A*,1)





HEAPIFY (*A*,2)

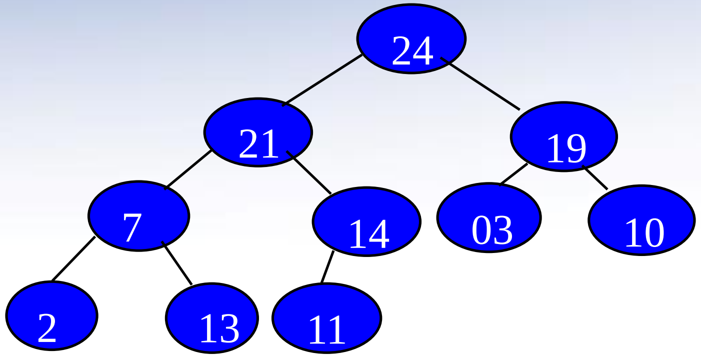


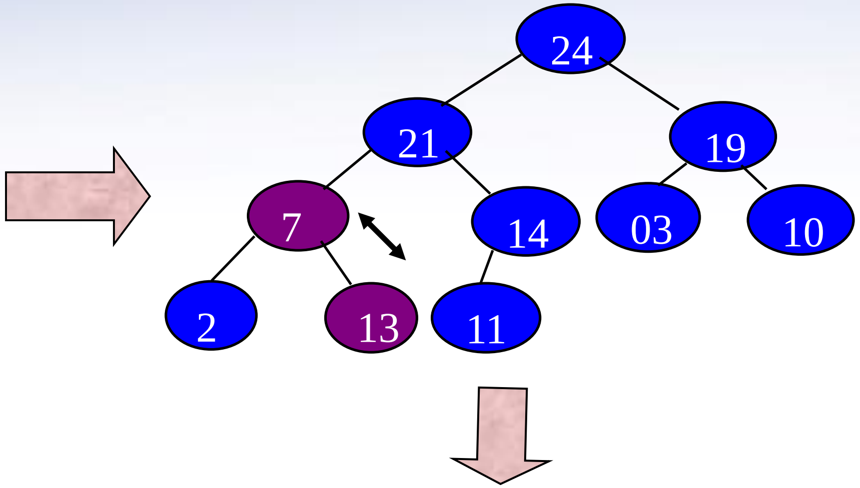


HEAPIFY (*A*,4)

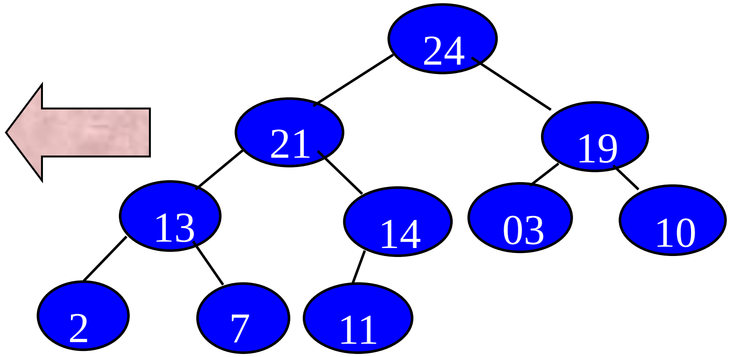
To be contd.

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





HEAPIFY (*A*,4)

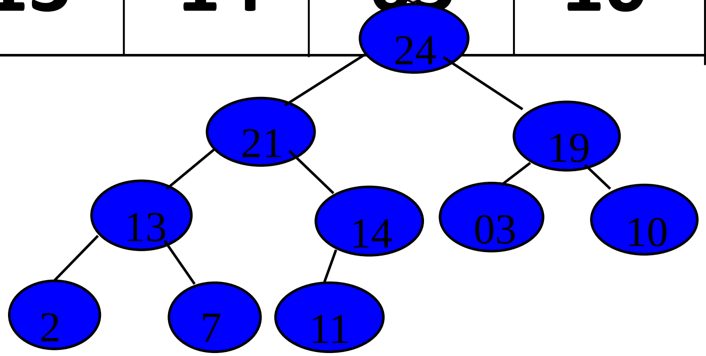


<u>Important point</u> Although we represent this process using a heap actually all the task in done on the input array

Resulting Heap

Array view of 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



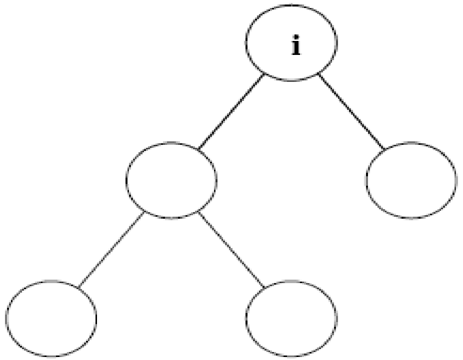
Analysis of Heapify Algorithm.

- The running time of MAX-HEAPIFY on a subtree of size n rooted at given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$, 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 $2n/3$-the worst case occurs when the last row of the tree is exactly half full-and the running time of MAX-HEAPIFY can therefore be described by the recurrence

$$\boldsymbol{T(n) \le T(2n/3) + \Theta(1).}$$

- The solution to this recurrence, by case 2 of the master theorem, is $\boxed{\boldsymbol{T(n) = O(\lg n)}.}$ Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as $O(h)$.

Recursive Analysis.

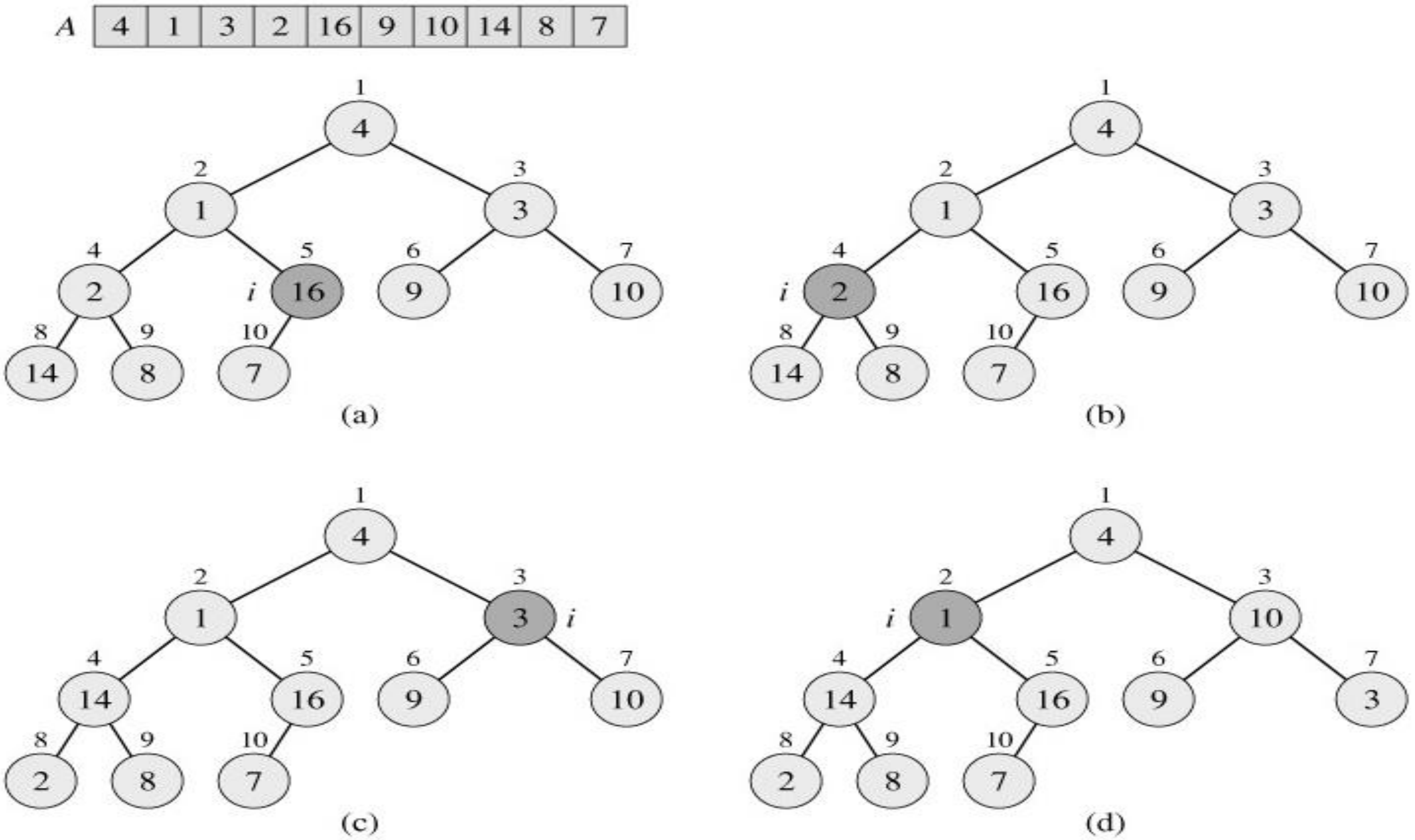


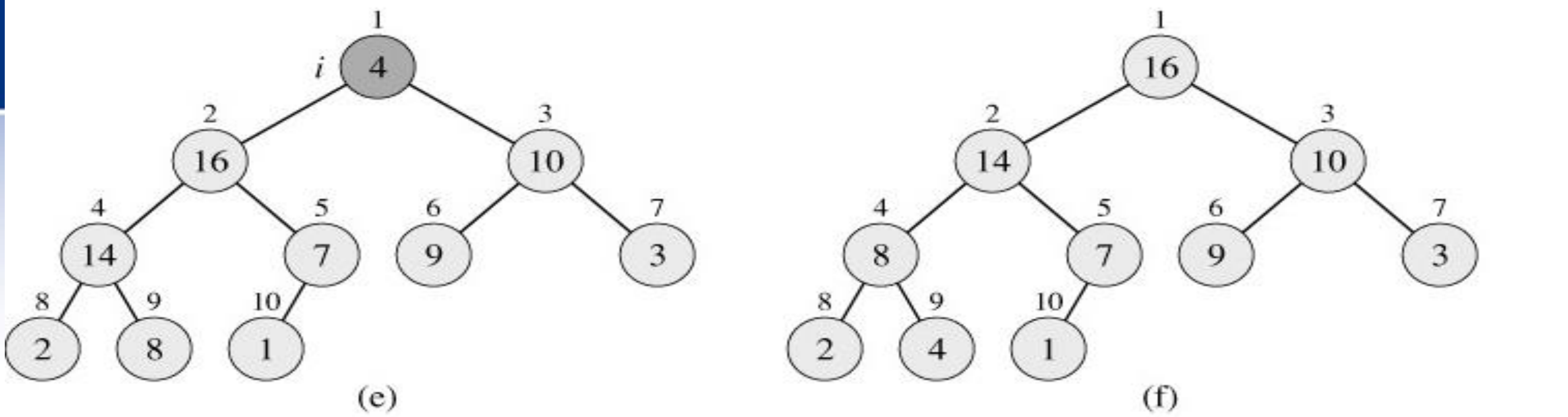
$$\text{n} = 5,\ n_{subtree} = \lfloor 2n/3 \rfloor = 3$$

Note that this expression has the maximum value when the lowest level of the heap is exactly half full.

$$\text{T(n)} = \text{T}(\lfloor\ 2\text{n}/3\ \rfloor) + \Theta(1)$$

Master: $a = 1$, $b = 3/2$, $f(n) = \Theta(1) = \Theta(n^{log_{3/2}1}) = \Theta(n^0) = \Theta(1)$,
Case ____________
$T(n) = \Theta(n^{log_{3/2}1} lgn) = \Theta(lgn)$.


A 4 1 3 2 16 9 10 14 8 7
(a)
(b)
(c)
(d)

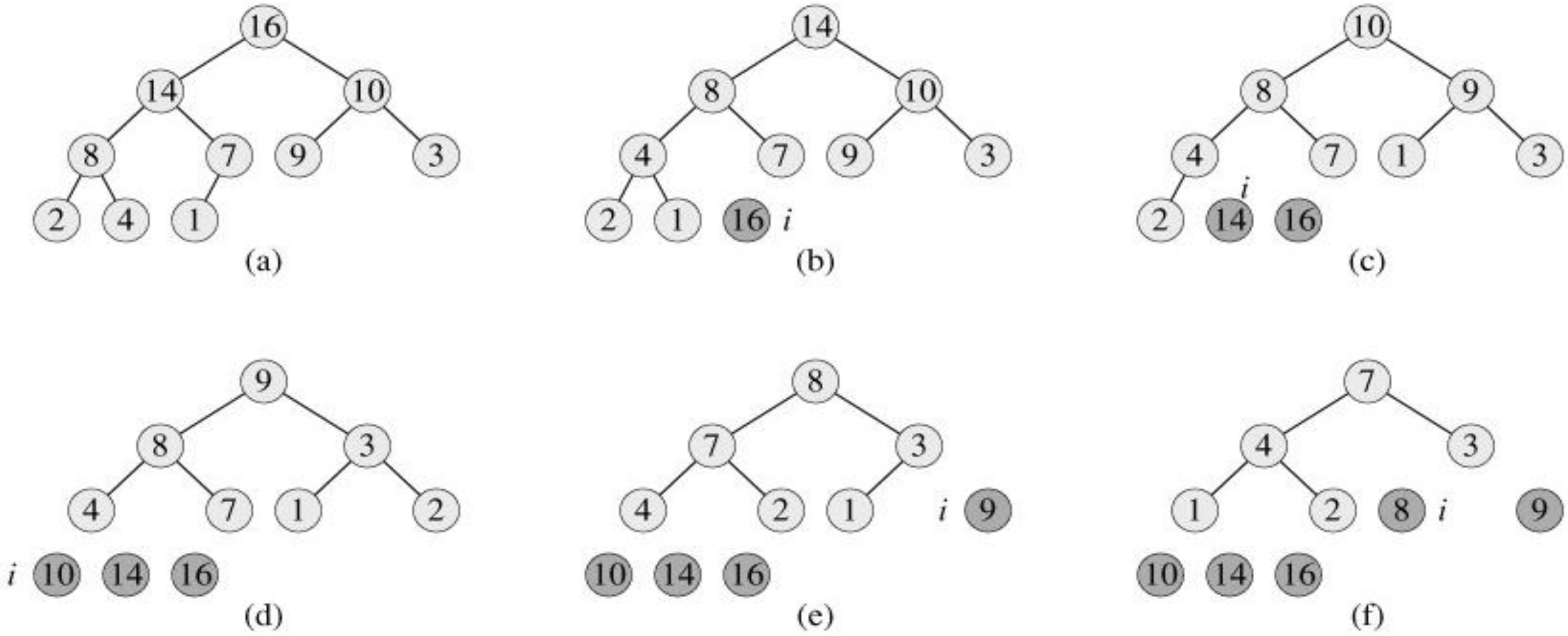


The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. *(a)* A 10-element input array *A* and the binary tree it represents. The figure shows that the loop index *i* refers to node 5 before the call MAX-HEAPIFY(*A*, *i*). *(b)* The data structure that results. The loop index *i* for the next iteration refers to node 4. *(c)*-*(e)* Subsequent iterations of the *for* loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. *(f)* The max-heap after BUILD-MAX-HEAP finishes.

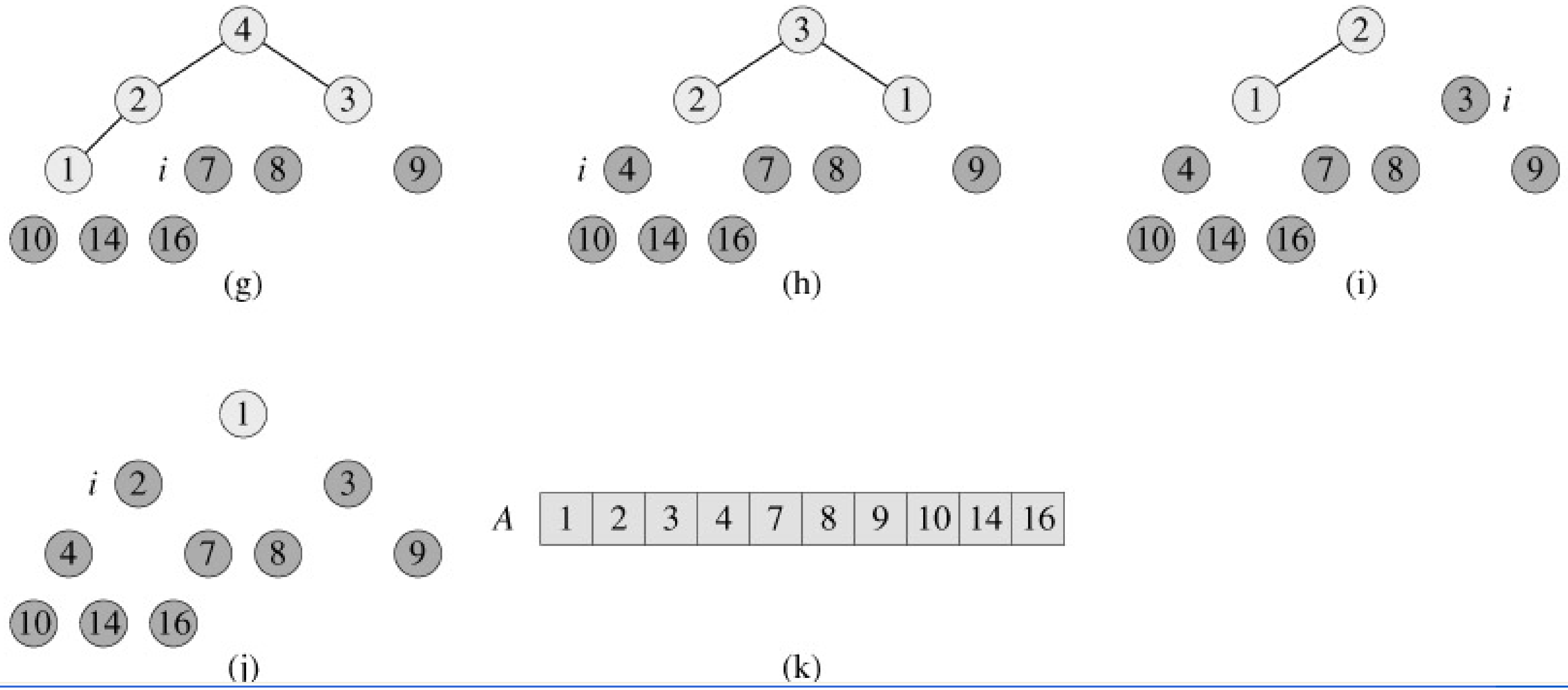
Analysis of Build Heap Algorithm.

- We can compute a simple upper bound on the running time of BUILD-HEAP as follows.
- Each call to **HEAPIFY costs $O(\lg n)$** time,
- and there are $O(n)$ such calls.
- Thus, **the running time is $\boxed{O(n \lg n)}$.**

The operation of HEAPSORT.



The operation of HEAPSORT.



HEAPSORT-complexity

Running Time:

- Step 1: BUILD_HEAP takes O(n) time
- Steps 2 to 5 : there are (n-1) calls to HEAPIFY which takes O(log n) time
- Therefore running time takes O (n log n)

Summary

- Complete binary Tree
- Heap property
- Heap
- Maintaining heap Property(HEAPIFY)
- Building Heaps
- HeapSort Algorithm

Questions ???

Thank You