SLIIT

Discover Your Future

IT2070 – Data Structures and Algorithms

Lecture 08

Heap Sort Algorithms

U. U. Samantha Rajapaksha

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

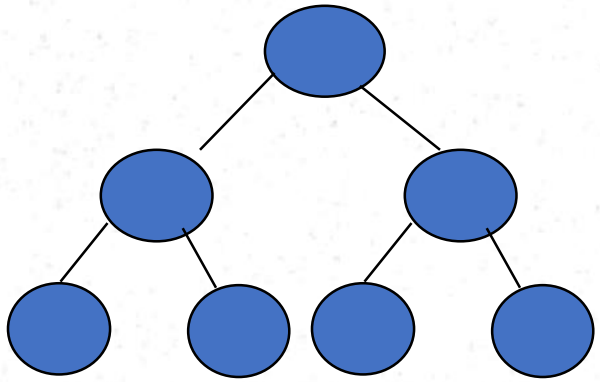
Samantha.r@slit.lk

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 $2^{h+1}-1$ nodes



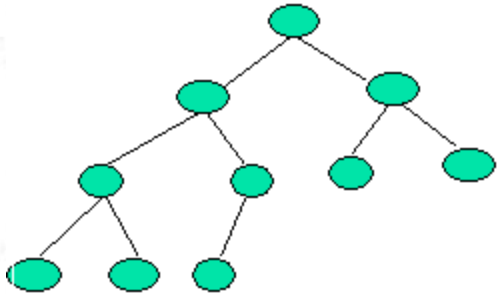
number of nodes = $2^{\text{height of tree}+1} - 1$

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

Height of a complete binary tree

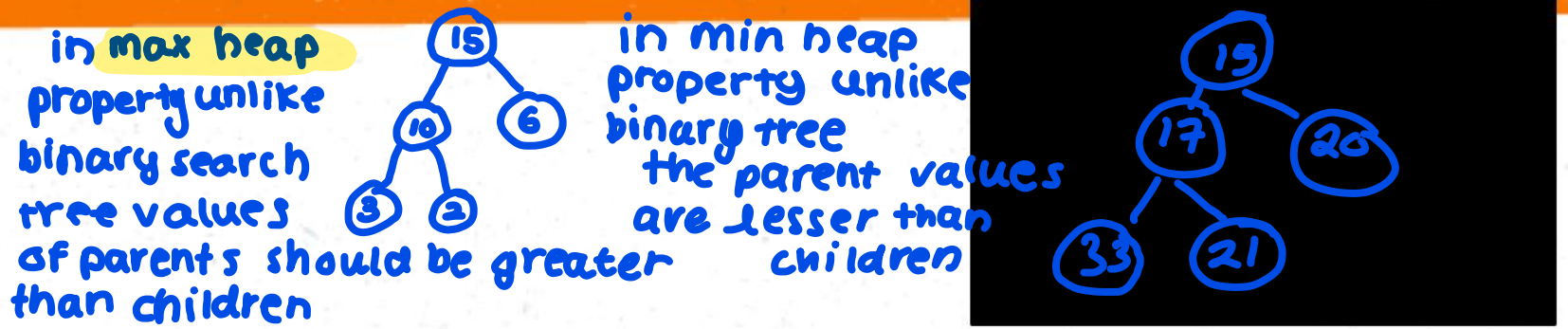
Height of a complete binary tree that contains ***n*** elements is $\lfloor \log_2(n) \rfloor$

Example



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

Heaps



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

1) **max heaps** and 2) **min heaps**

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

Definition!!!!!!!!!!!!!!

max-heap → max-heap property:

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

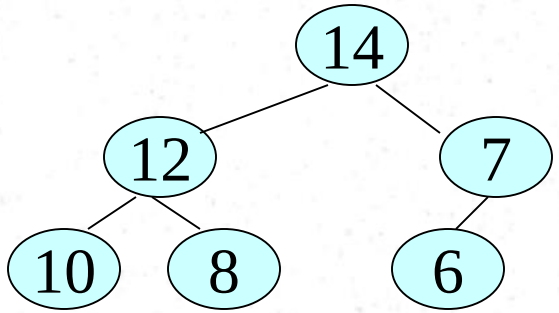
min-heap → min-heap property:

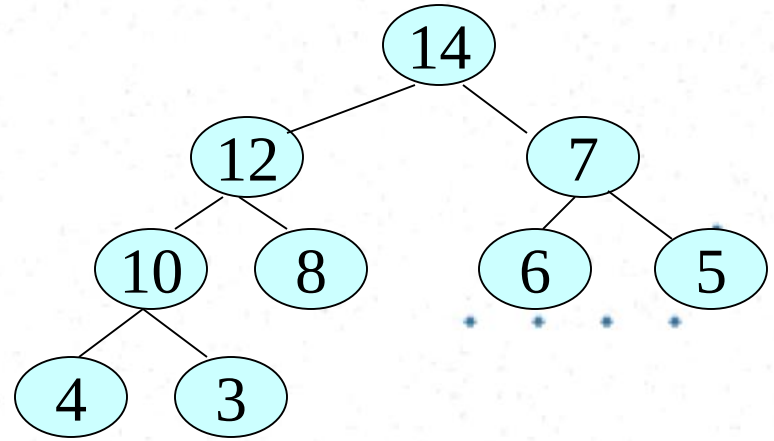
The value of each node is less than or equal to those of its children.

Max-heaps are used in heapsort algorithm

Heaps

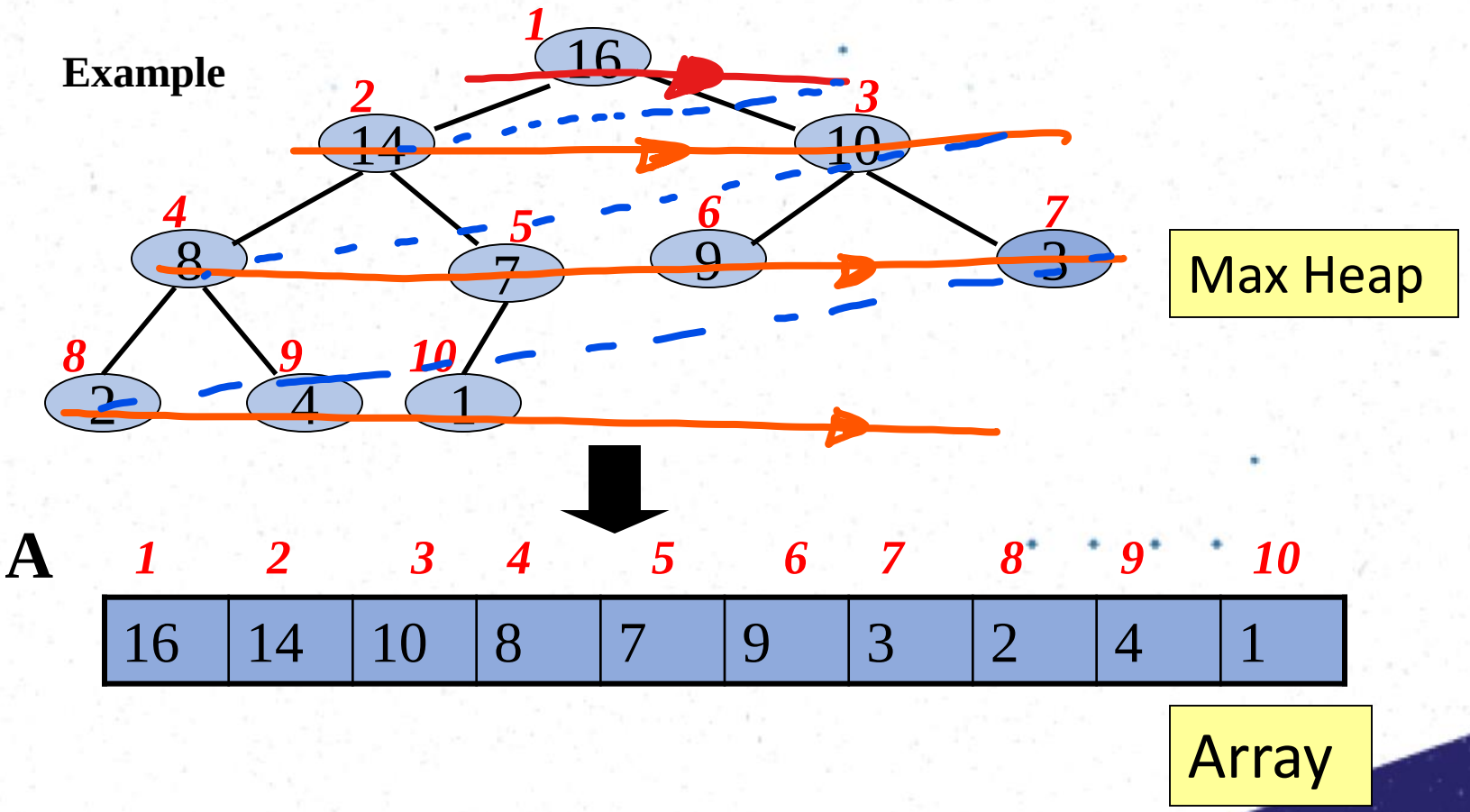
Complete Binary Tree with the **max-heap property - examples**





Heaps (contd.)

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

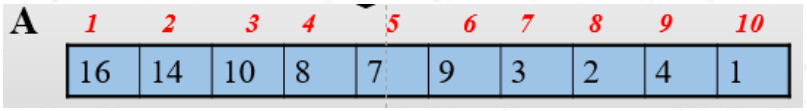


Heaps (contd.)

steps to follow to apply maxheap algorithm (To the Tree diagram)
1.Arrange the given data set into a tree(according to the CBT method -left to right)
2.arrange the tree according to the Max heap algorithm(Heapify)
3.Apply the swapping according to the heapsort Algorithm

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

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



Given node with index *i*,

in a complete binary tree the parent node and left, right child nodes are defined as follows

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

  PARENT(i) = $\lfloor i/2 \rfloor$ ← a parent of any value in the array can be taken by this

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

  LEFT_CHILD(i) = **2×*i*** → left child of index 5 = 5×2 = 10

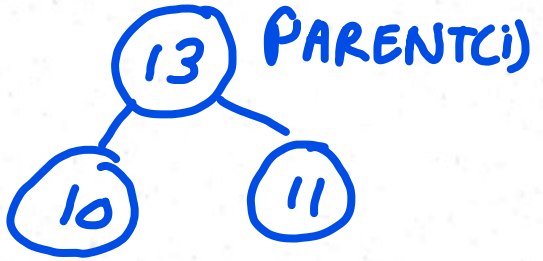
- ***RIGHT_CHILD(i)*** is the index of right child of ***i***;

  RIGHT_CHILD(i) = 2×*i* +1 → Right child of index 5 = 5×2+1 = 11 no index 11 ∴ no Right child

example :- parent of node 5 = $\frac{5}{2}$ = 2 ∴ parent of node 5 with value 7 is node 2 with value 14

∴ left child is node 10 with value 1

Heap Algorithms



- **MAX_HEAPIFY:**

  To maintain max-heap property

here we take 1 node to the correct place

the value of parent should be greater than child

$$A[PARENT(i)] \geq A[i]$$

here we take all the parent nodes to correct place

- **BUILD_MAX_HEAP**

  To build max-heap from an unsorted input array

- **HEAPSORT**

  Sorts an array in place.

MAX_HEAPIFY

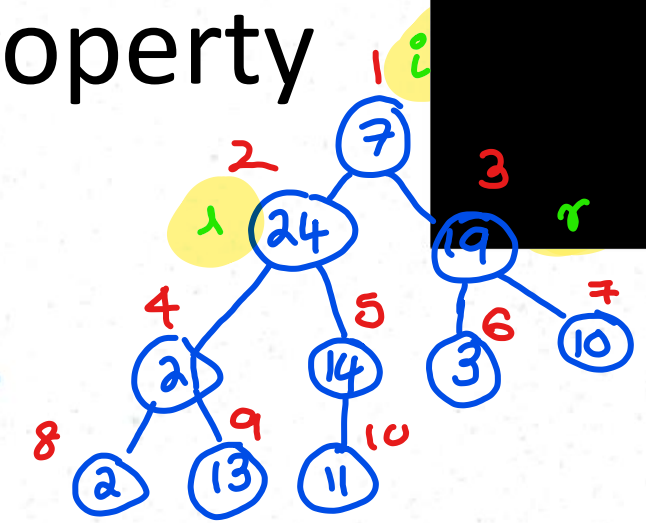
- The MAX_HEAPIFY algorithm checks the heap elements for violation of the heap property and restores max-heap property;

$$A[PARENT(i)] \geq A[i]$$

- **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*)

1. *l* = LEFT(*i*);
2. *r* = RIGHT(*i*);
3. if *l* ≤ A.heap_size and A[*l*] > A[*i*]
4. largest = *l*;
5. else largest = *i*;
6. if *r* ≤ A.heap_size and A[*r*] > A[*largest*]
7. largest = *r*;
8. if *largest* ≠ *i*
9. exchange A[*i*] with A[*largest*]
10. **MAX_HEAPIFY** (A,*largest*)

left child value

right child value

doesn't this count as the size of the array?

size of whole heap

value of l > value of i

largest is now l

the value we assing as largest

if largest is not i then interchange

Example

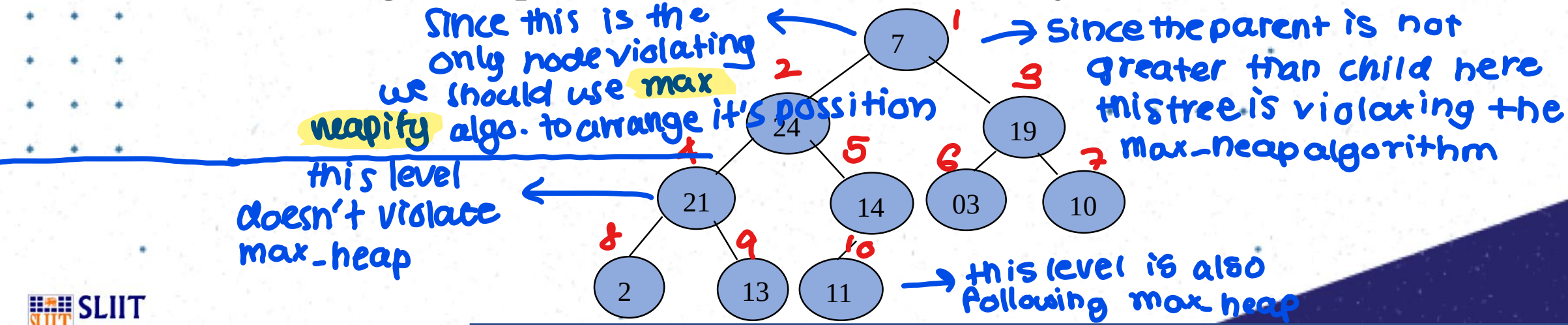
You are given the following array

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

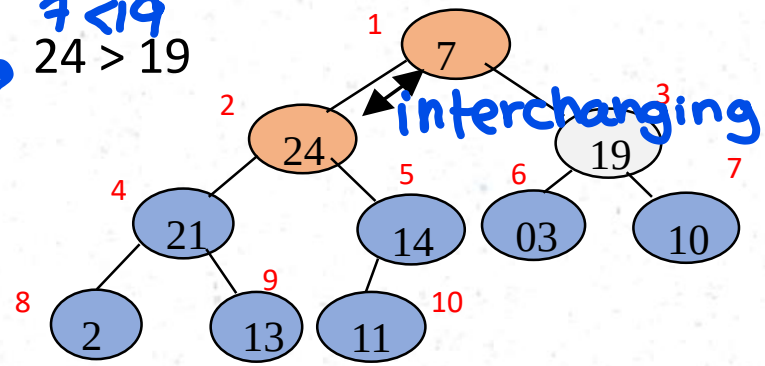
Drawing a heap would make our work easy



MAX_HEAPIFY (*A*,1)

we apply max-heapify algorithm to the whole array

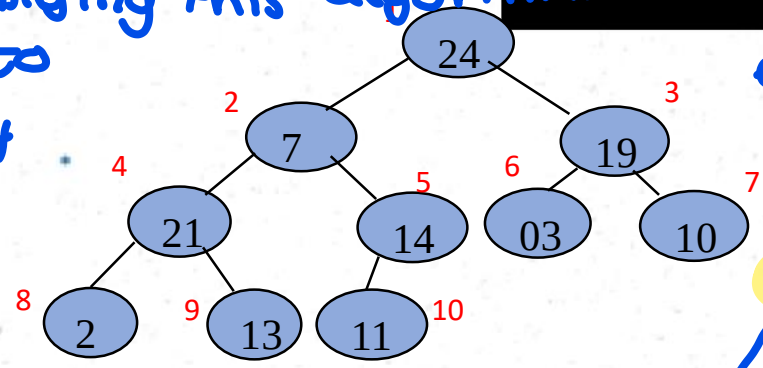
index which we are applying this algorithm to whole array



7 < 24
7 < 19
24 > 19

then check 9

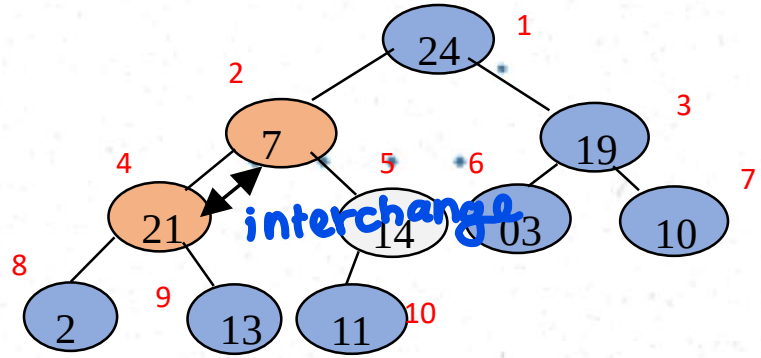
interchanging 7 and 24



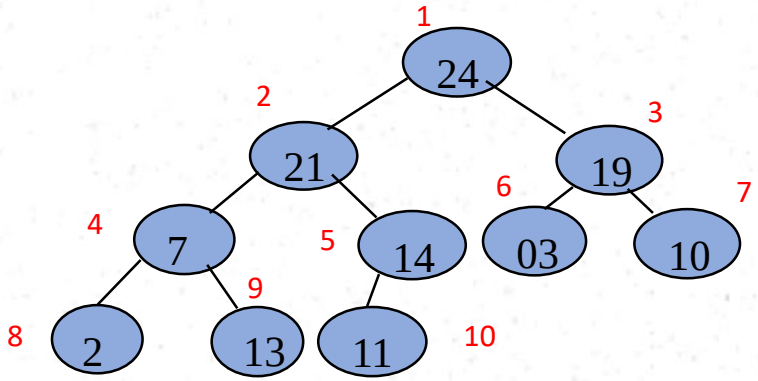
even now,
7 < 21
7 < 14
21 > 14

therefore again apply max-heapify

MAX_HEAPIFY (*A*,2)



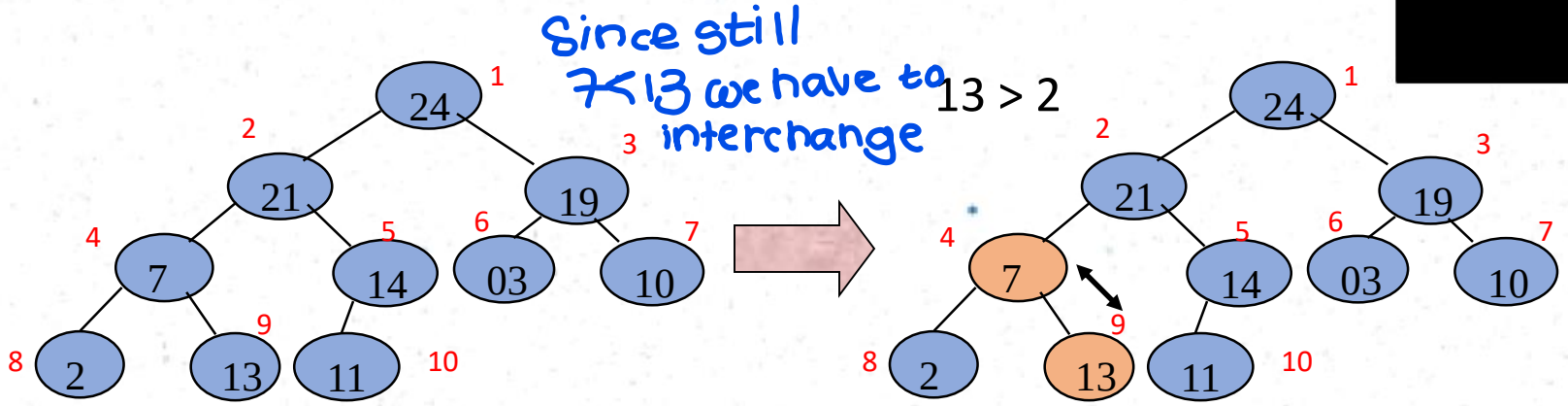
21 > 14



MAX_HEAPIFY (*A*,4)

To be contd.

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

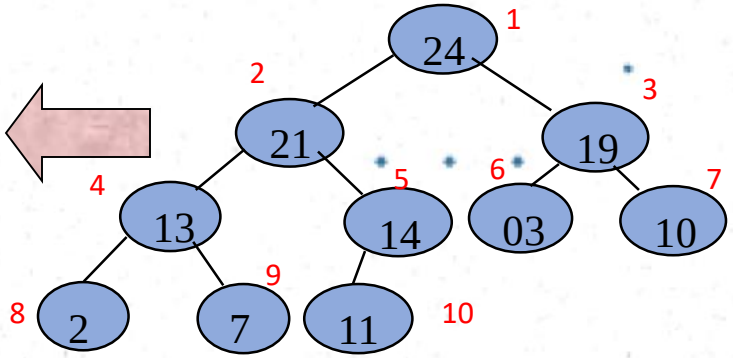


MAX_HEAPIFY (*A*,4)

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

using the tree

on this array of content



Resulting Heap

Array view of MAX_HEAPIFY Algorithm

this the array view before calling max.heapify

7	24	19	21	14	03	10	02	13	11

this is after one max-heapify is called

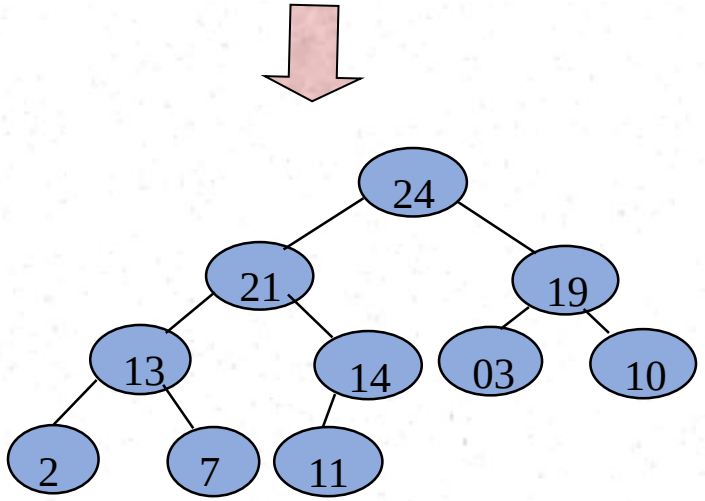
24	7	19	21	14	03	10	02	13	11

after 2 →

24	21	19	07	14	03	10	02	13	11

after 3 →

24	21	19	13	14	03	10	02	07	11



if you go through the code we know from line number 1–9 the algorithm doesn't consider about the size of the array (we just exchange and compare the neighbouring nodes) but the last 10th line in the previous case would have to execute 3 times for the node 7 to be pushed to it's correct position. That is because we know the height of the tree

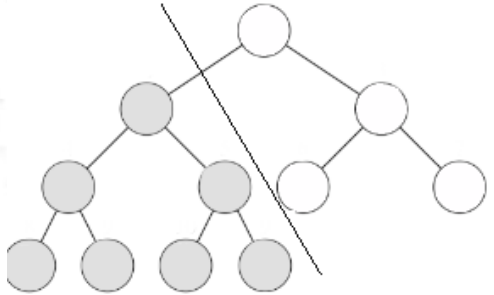
Analysis of Heapify Algorithm.

∴ the height of the tree is needed for the last line to execute

So if we say the time for line 19 is K then the total time would be 3K

like that here the constant time is 0

- The running time of MAX-HEAPIFY on a subtree of size n rooted at given node i is the $\Theta(1)$ 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 $O(h)$.

$O \times h = O(h)$

the hieght of a complete binary tree $\} = \log_2 n$

The solution to this recurrence is

Running time for max-heapify algorithm $T(n) = O(\lg n)$

$\therefore T(n) = O(\log_2 n)$

BUILD_HEAP

we know in a tree/heap not just the root but many element within the tree might not obey this max-heapify algorithm ∴ this algorithm is used to rebuild the whole tree

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 = ⌊A.length/2⌋ 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

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

this equation helps us to get the last parent node of the array

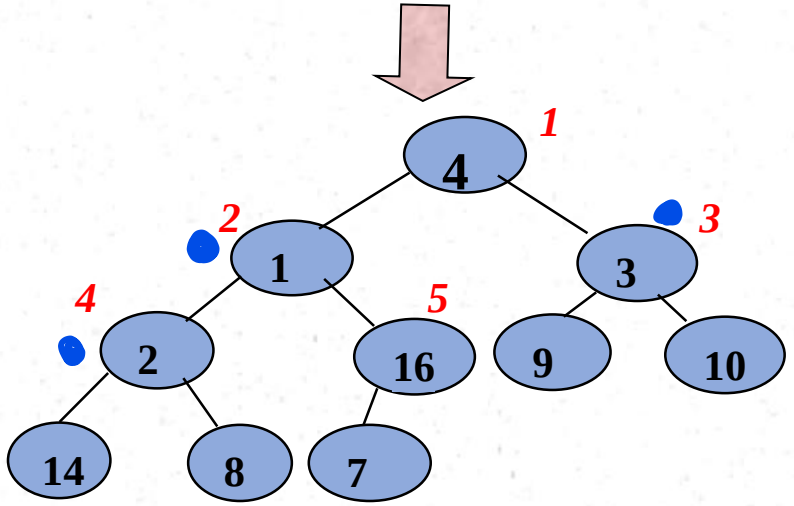
MAX_HEAPIFY(A,5)

we call max_heapify until $i = 5$

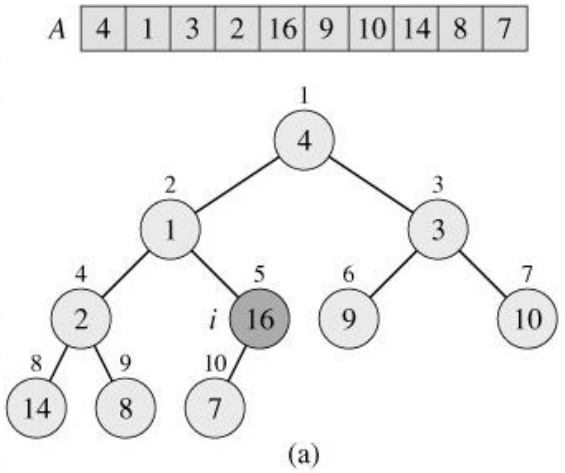
MAX_HEAPIFY(A,5)
MAX_HEAPIFY(A,3)
MAX_HEAPIFY(A,2)
MAX_HEAPIFY(A,1)

- the nodes which has the issues of not following max-heapify

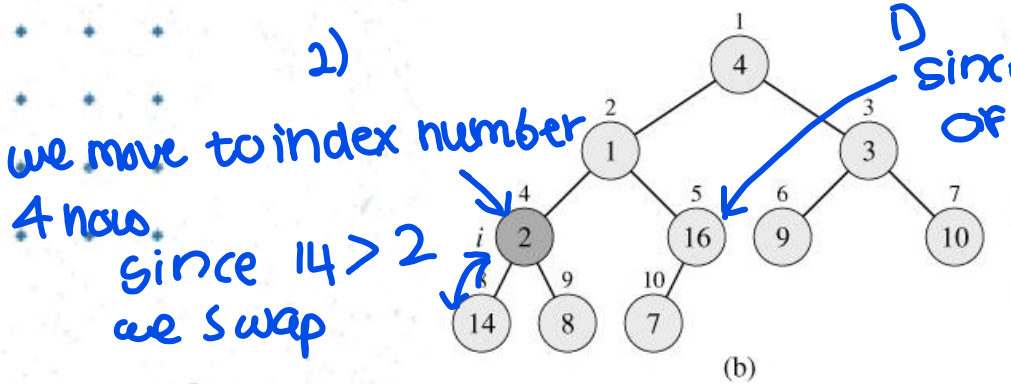
A	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	**16**	9	10	14	8	7



Solution (Contd.)



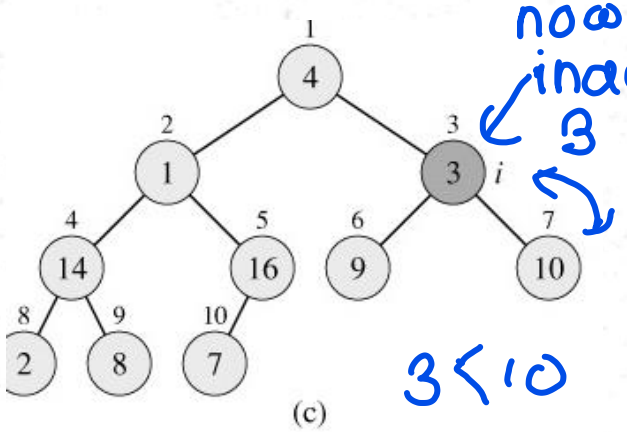
loop index *i* refers to node 5



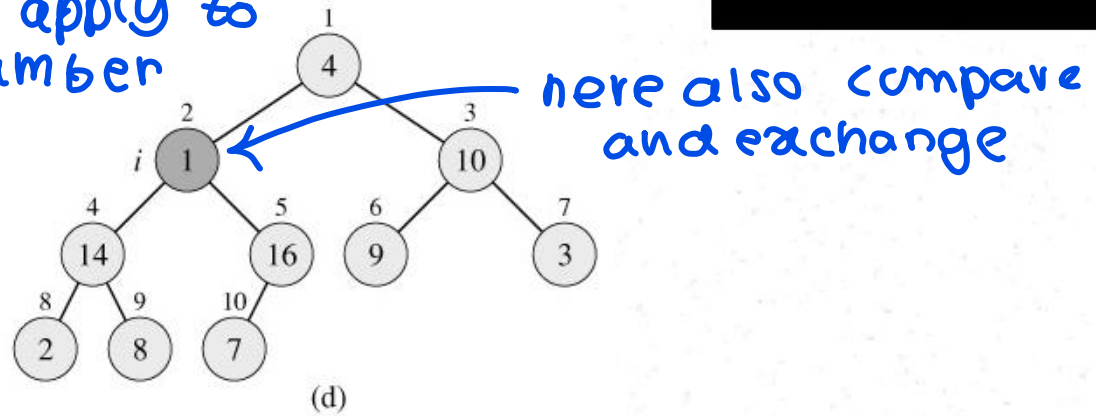
2) we move to index number 4 now. since 14 > 2 we swap

1) since index number 5 value is > than values of index 10 there is no exchange needed to be done

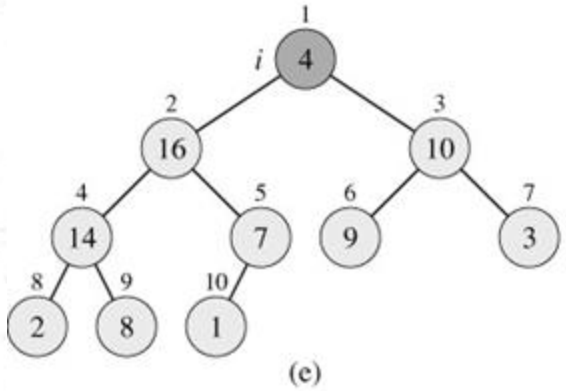
loop index *i* for the next iteration refers to node 4



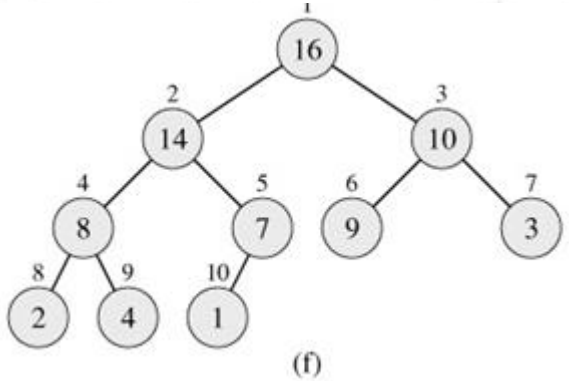
loop index *i* refers to node 3



loop index *i* refers to node 2



loop index *i* refers to node 1



max-heap after BUILD-MAX-HEAP finishes.

Analysis of Build Max Heap Algorith

here we call max-heapify to all parent nodes

in build max heap the height varies because we don't know where might the last parent node is

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.

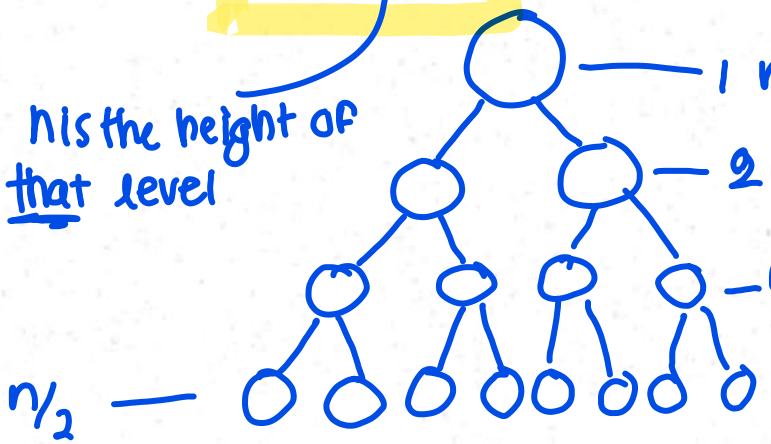
- *n*-element heap has height $\lfloor \lg n \rfloor$

and

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

h is the height of that level

n/16 roughly



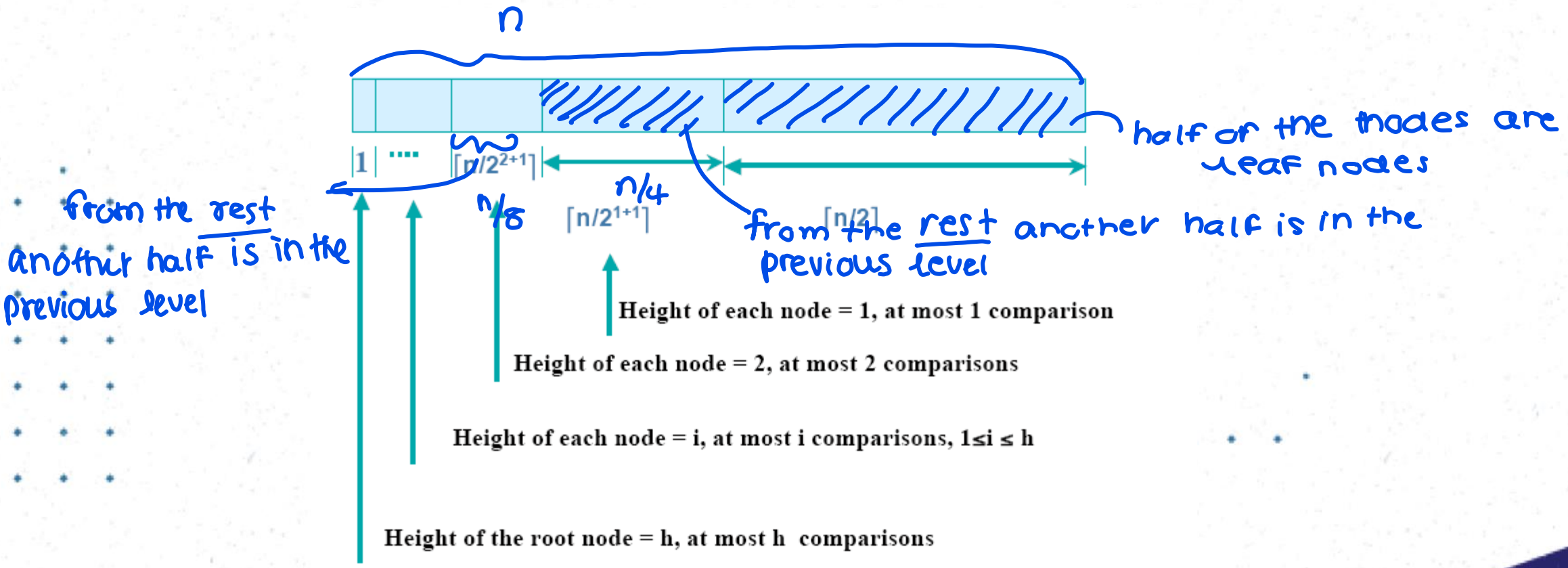
$n/2^{3+1}$

$n/2^{2+1}$

usually if you have n number of nodes in the tree then you have n/2 number of nodes in the last level

$n/4 = n/2^{1+1}$

Analysis of Build Max Heap Algorith



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:

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

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:

running time of build-heap

finnally after simplified

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

- 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)
```

now we have a build max heap array with all the parent nodes in the correct places

we exchange A[1] the root with the last element in array

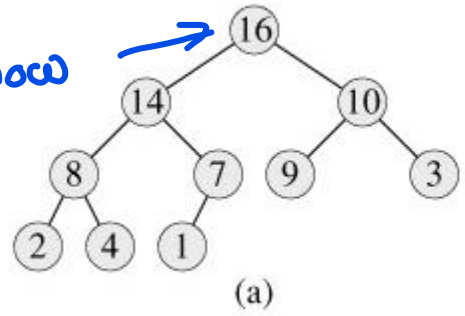
next we can sort it

The operation of HEAPSORT.

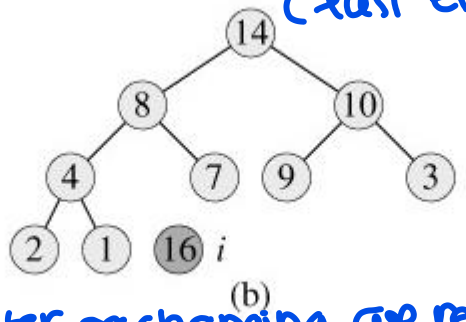
this happens after BUILD-MAX-HEAP

we know now A[1] is definitely the largest of the heap

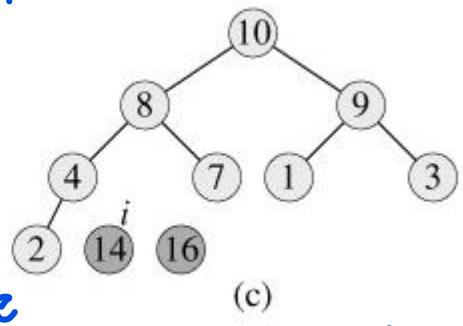
exchange A[1] with A[i]
A[1] is exchanged with A[i] (last element in tree)



(a)

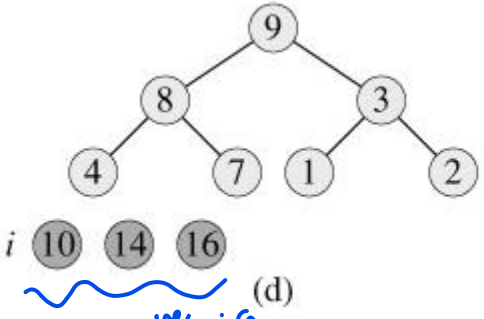


(b)



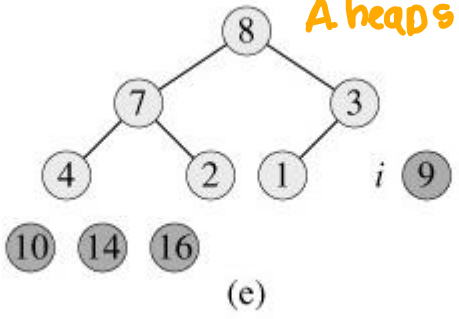
(c)

after exchanging we remove the last element from our tree (size of tree is reduced)
A.heapsize = A.heapsize − 1

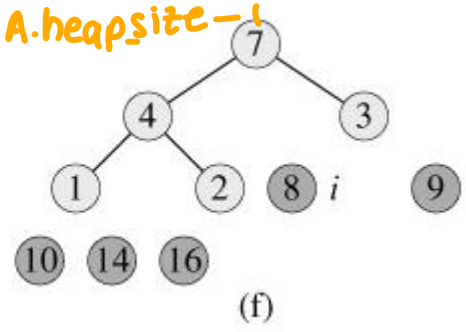


(d)

now here Sorting is happening

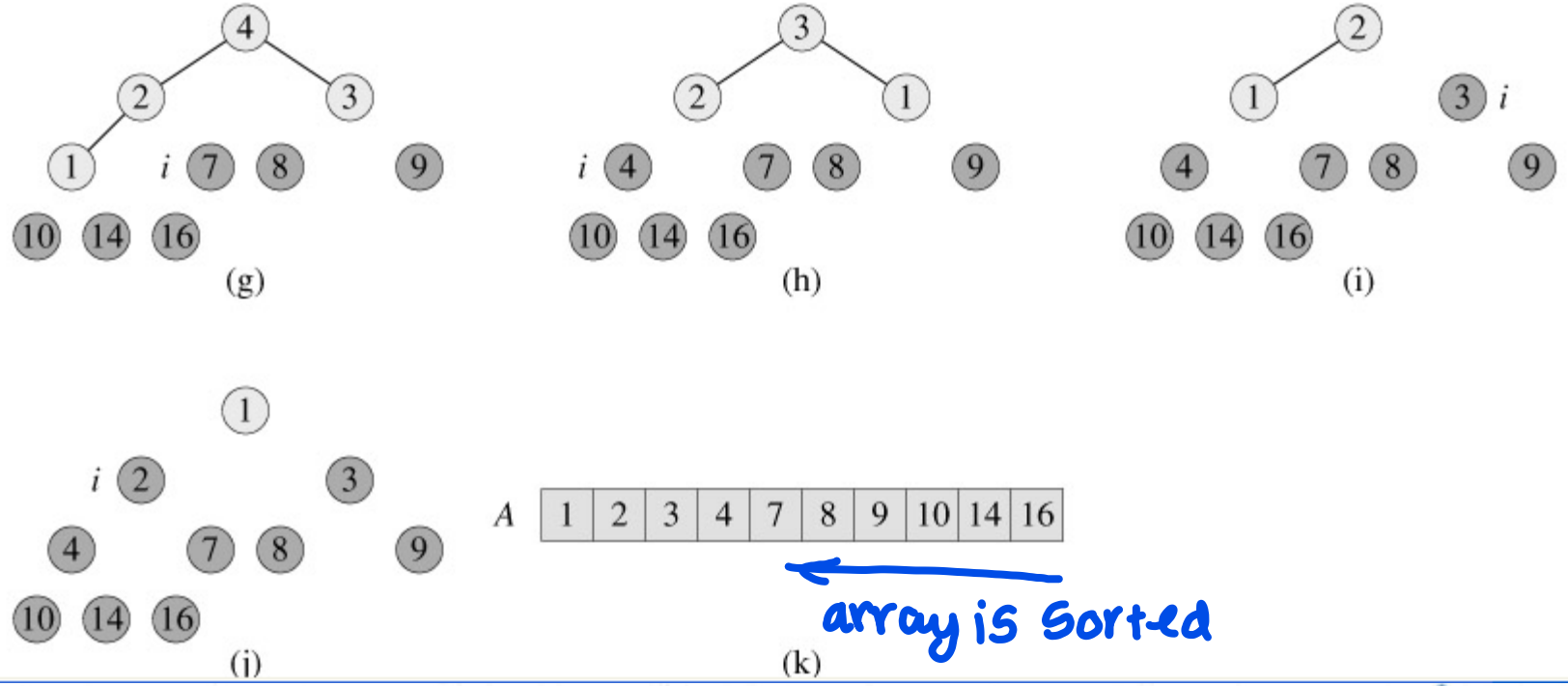


(e)



(f)

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)

Running time of heapsort algorithm

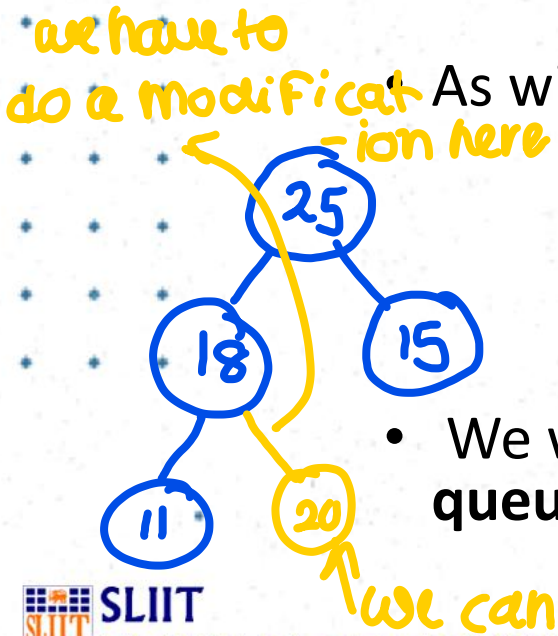
MAX_HEAPIFY → O(log n)
BUILD MAX HEAP → O(n)
HEAP SORT → O(n log n)

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 – based on max heaps
  - min-priority queues – " " min "



- 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.

Operation done by max_priority queues

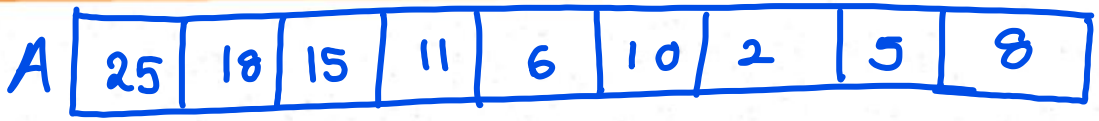
1) • INSERT(*S*, *x*) inserts the element *x* into the set *S*. This operation could be written as $S = S \cup \{x\}$.

2) • EXTRACT-MAX(*S*) removes and returns the element of *S* with the largest key. → remove the max value

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.



1 HEAP_EXTRACT_MAX

Check example lecture 8 Heap_extract_max we can remove the maximum value

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])

we consider the size of our heap is n,

```
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
```

time is constant

this running time is O(logn)

→ after ignoring the constants

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.