BM267 - Introduction to Data Structures

9. Heapsort

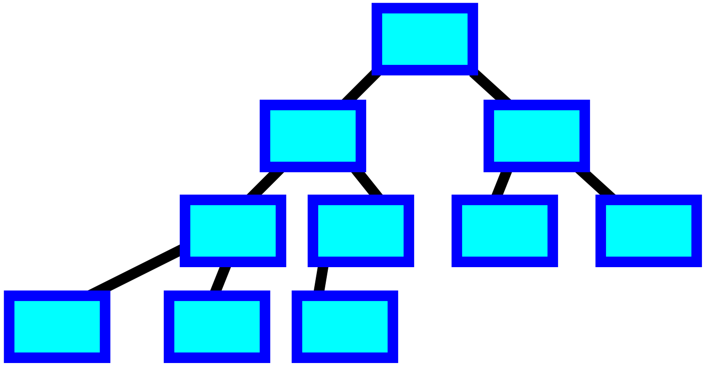
Ankara University
Computer Engineering Department
Bulent Tugrul

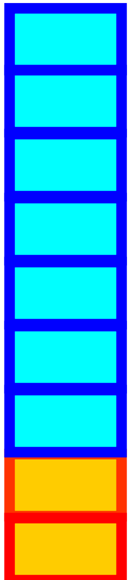
(Binary) Heap Structure

The **heap data structure** is an array organized as a **almost complete binary tree (which is always balanced)**.

The tree is completely filled (except possibly for the right side of the lowest level)

Which means, if N is the number of heap elements, the first N array elements are always full.

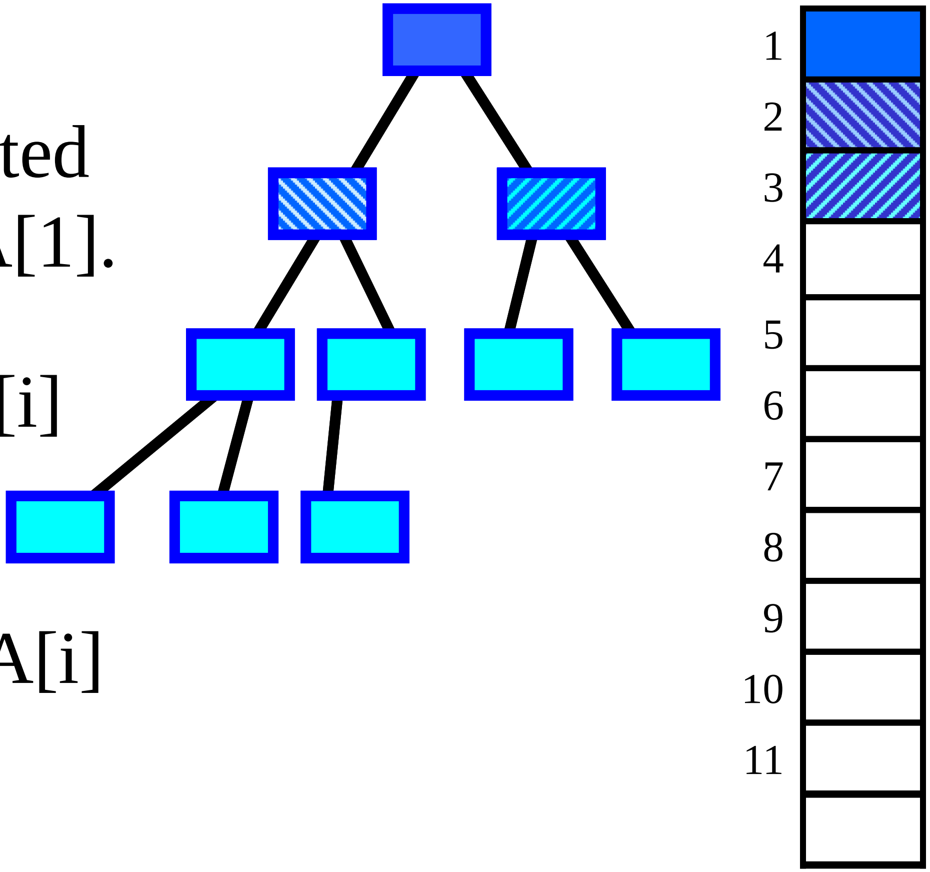




(Binary) Heap Structure

Structural Properties:

- The root of the tree is located in the first array element A[1].
- The left subtree of node A[i] is located in A[2i]
- The right subtree of node A[i] is located in A[2i+1]



(Note the 1-based notation for array indices)

(Binary) Heap Structure

More structural Properties:

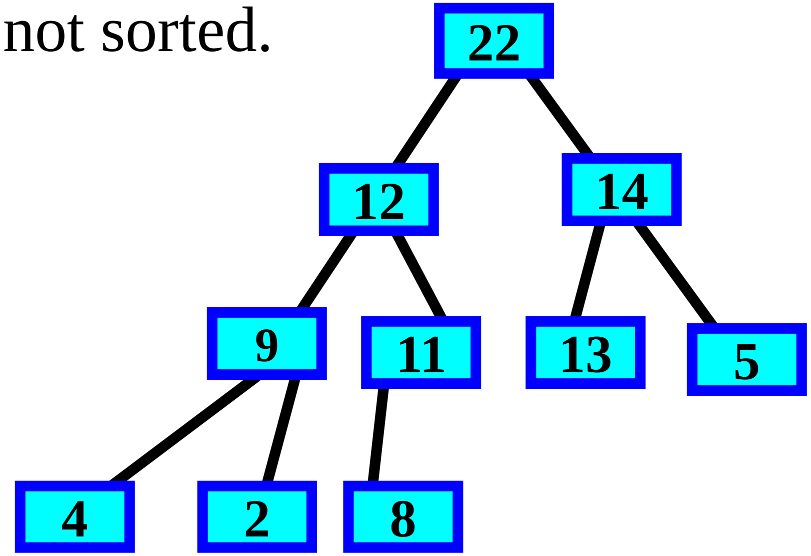
- **Left** subtrees are rooted on **even** numbered array elements,
- **Right** subtrees are rooted on **odd** numbered array elements.
- Parent of node i is in node A[$\lfloor i / 2 \rfloor$].
- Heap with N elements has height = $\lfloor \mathbf{log_2}\ N \rfloor$.

($\lfloor x \rfloor$ denotes truncation to integer.)

(Binary) Heap Structure

The tree nodes are located in the array in a top-down, left-to-right traversal order.

- A[1] contains the largest element;
- Elements in A[2] .. A[N] are not sorted.



22	**12**	**14**	**9**	**11**	**13**	**5**	**4**	**2**	**8**			
1	2	3	4	5	6	7	8	9	10	11	12	..

(Binary) Heap Structure

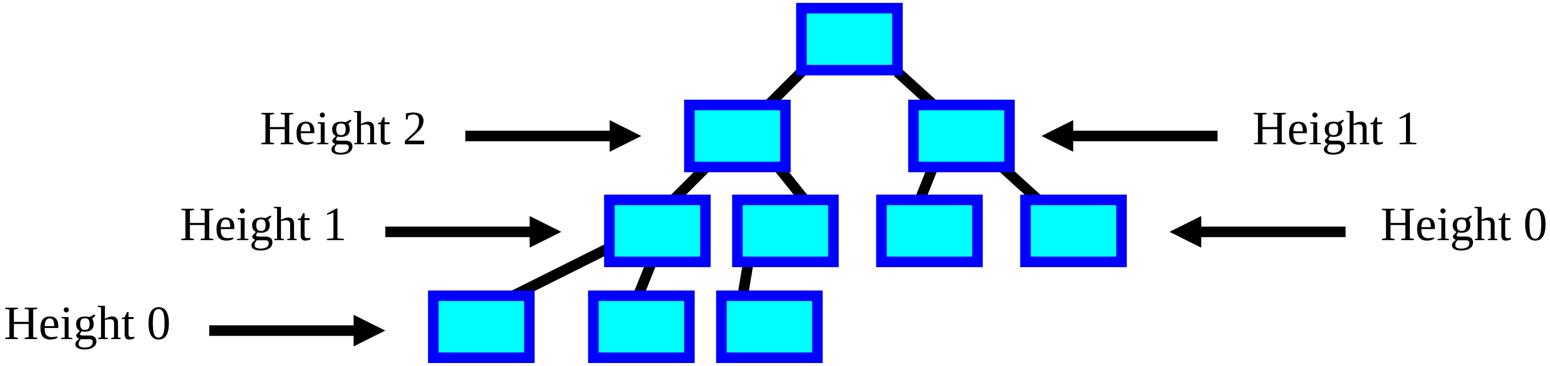
Content Properties: A heap must satisfy either of:

- **Max-Heap property**: A[parent(i)] ≥ A[i]
  - The value at node **i** cannot be greater than its parent.
  - Which means that the value at the root has the largest value currently in the heap.
- **Min-Heap property**: A[parent(i)] ≤ A[i]
  - The value at node **i** cannot be smaller than its parent.
  - Which means that the value at the root has the minimum value currently in the heap.

(Binary) Heap Structure

Some terminology:

- **Height of a node:** the number of segments of a downward path to the farthest leaf node.
- **Height of the tree:** the height of the root node.



Heap Operations

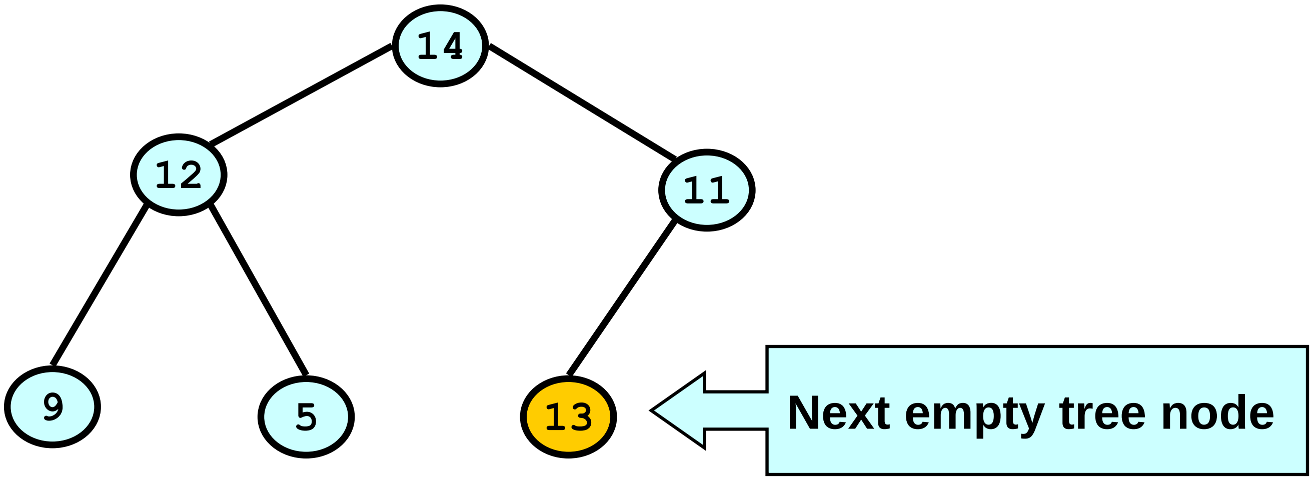
MaxHeapInsert( )	Adds an item to the heap
MaxHeapify( )	Maintains the heap property.
BuildMaxHeap( )	Converts an unordered array into a heap
HeapSort( )	Sorts the array in place
HeapExtractMax( )	Removes the item at the root

MaxHeapInsert( ) - Insert an item into the heap

Given: A heap of size M, and a new element.

Operation: Insert the new element into next available slot.

14	12	11	9	5	13							
1	2	3	4	5	6	7	8	9	10	11	12	..



Next empty tree node

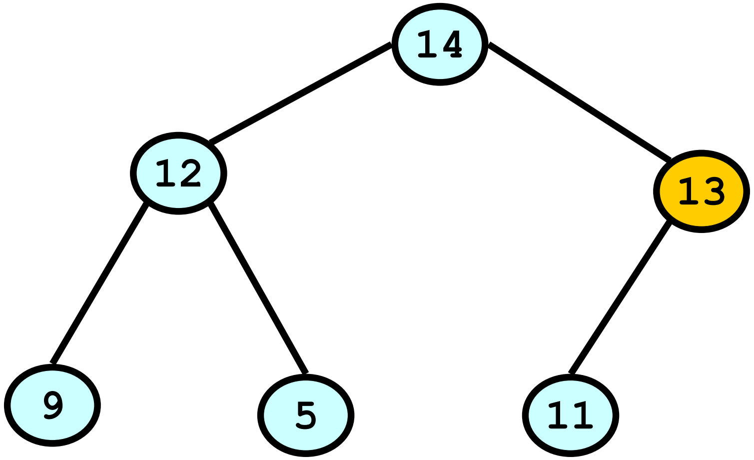
MaxHeapInsert( )

Given: A heap of size N, and a new element.

Operation:

- Insert new element into next available slot.
- Bubble up until the heap property is established

14	12	11	9	5	13							
1	2	3	4	5	6	7	8	9	10	11	12	..



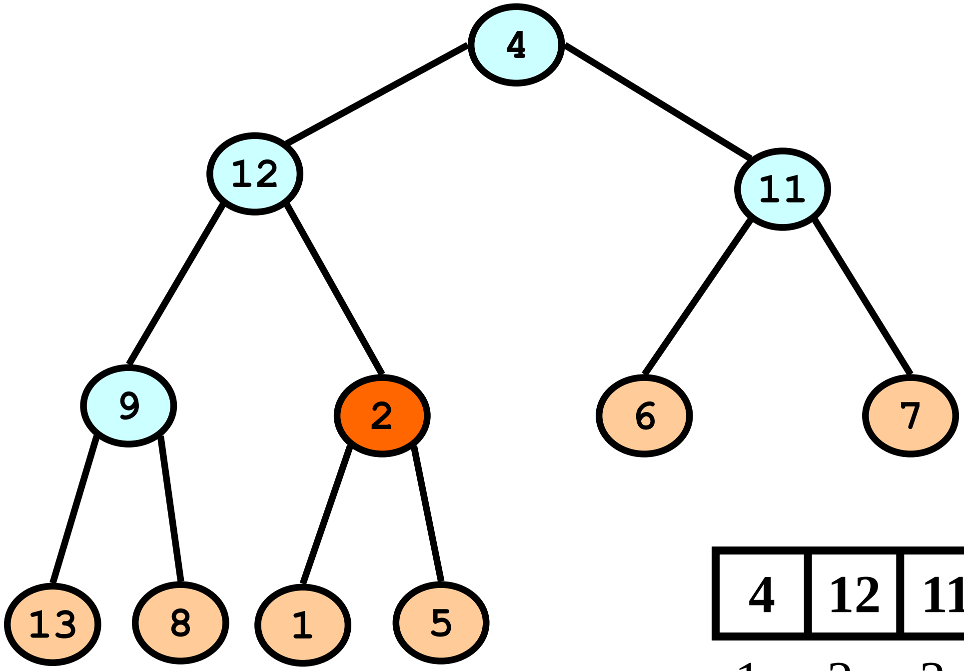
Now the heap is of size N+1

O($\log_2$ N) operations

MaxHeapify( ) - Combine heaps with the parent

Given: Node i, whose children at nodes 2i and 2i+1 already heapified.

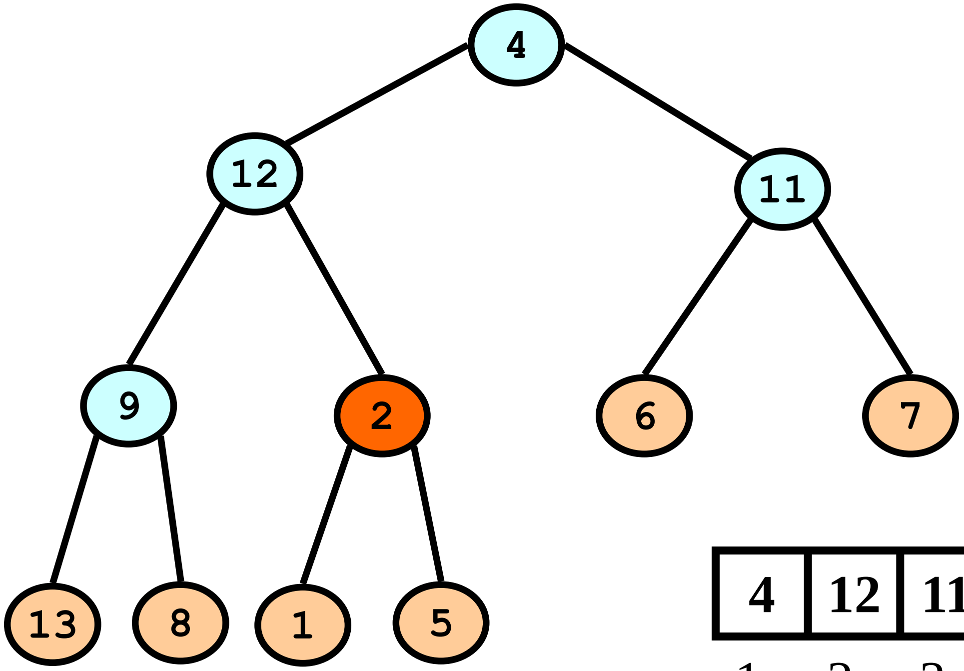
Operation: Let the value of A[i] float down until the node A[i] becomes a heap.

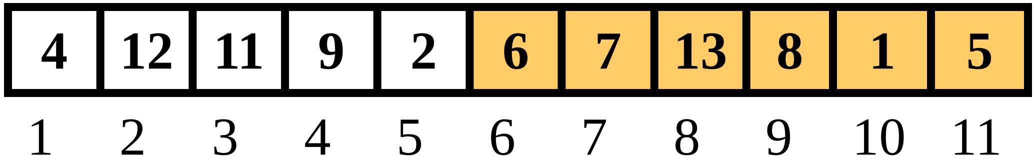


4	**12**	**11**	**9**	**2**	**6**	**7**	**13**	**8**	**1**	**5**
1	2	3	4	5	6	7	8	9	10	11

MaxHeapify( ) - Combine heaps with the parent

If the heap property is not satisfied, exchange the child node with the largest value and A[i].

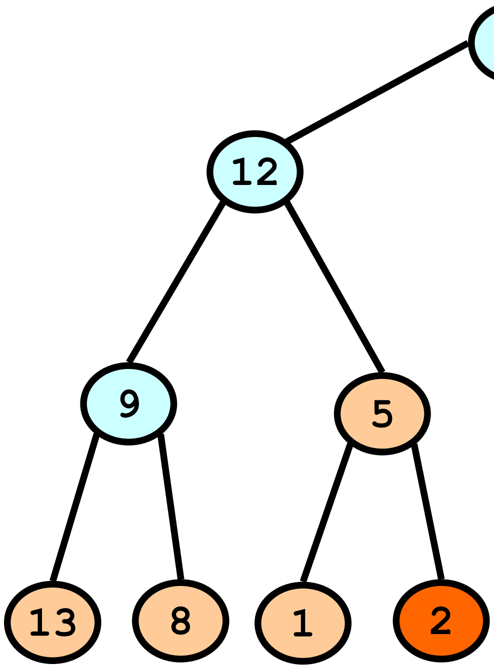




4	12	11	9	2	6	7	13	8	1	5
1	2	3	4	5	6	7	8	9	10	11

MaxHeapify( ) - Combine heaps with the parent

If the heap property is not satisfied at the selected child node, repeat the process.



O(log N) operations.

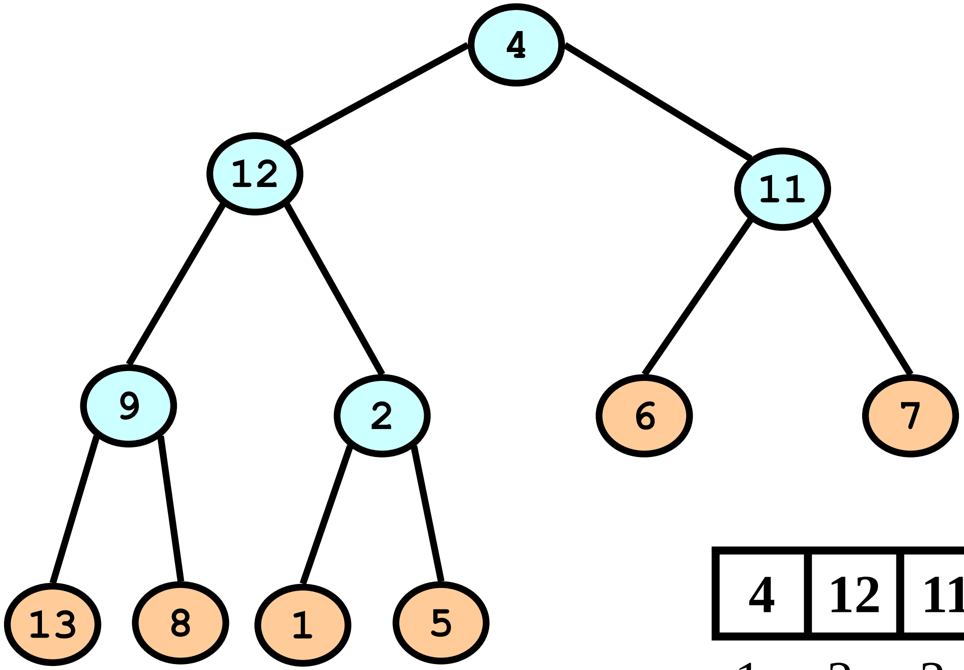
4	**12**	**11**	**9**	**2**	**6**	**7**	**13**	**8**	**1**	**5**
1	2	3	4	5	6	7	8	9	10	11

BuildMaxHeap( ) - Convert an array into a heap

Given: An unordered array of size N.

Operation:

- Convert each node in the tree into a heap, bottom-up.
- Nodes A[$\lfloor N/2 \rfloor$+1]..A[N]) are already heaps of size 1.
- Convert nodes A[$\lfloor N/2 \rfloor$] .. 1 into heaps, using **MaxHeapify(i)**



O(N) **MaxHeapify** calls

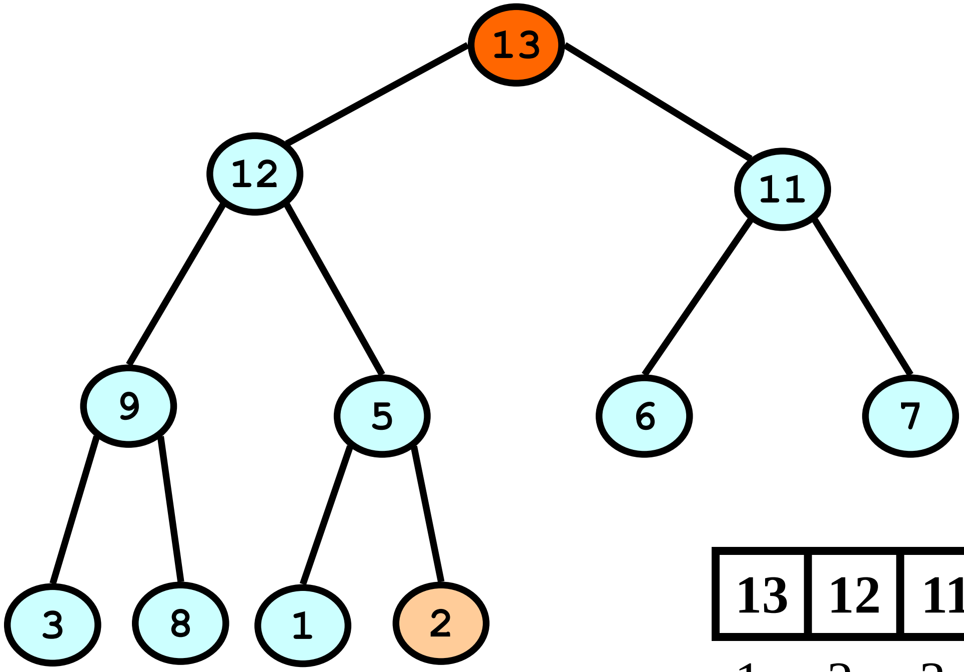
4	**12**	**11**	**9**	**2**	**6**	**7**	**13**	**8**	**1**	**5**
1	2	3	4	5	6	7	8	9	10	11

HeapExtractMax( ) - Remove the item at the root

Given: A heap of size N,

Operation:

- Exchange root with rightmost leaf.

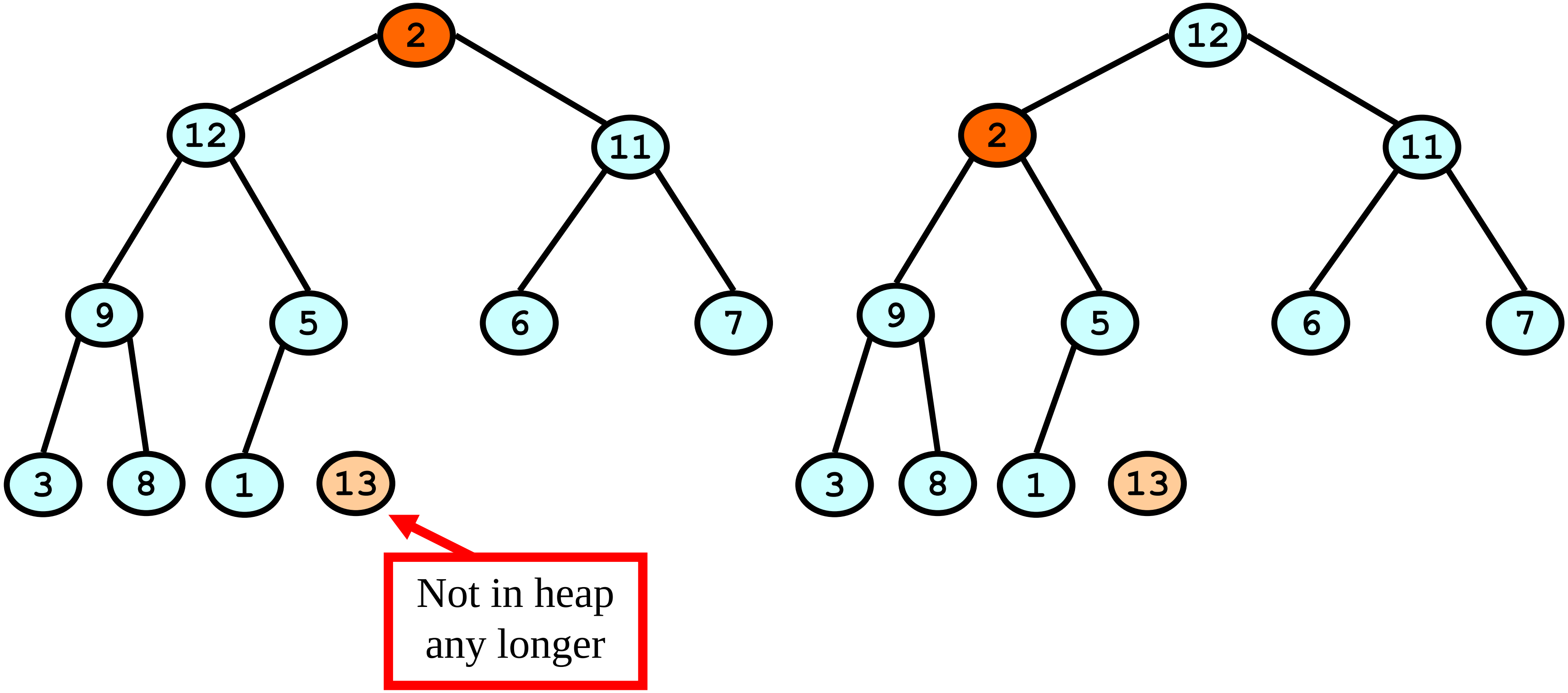


13	**12**	**11**	**9**	**5**	**6**	**7**	**3**	**8**	**1**	**2**
1	2	3	4	5	6	7	8	9	10	11

HeapExtractMax( ) - Remove the item at the root

Let the value of A[1] float down until the heap property is established.

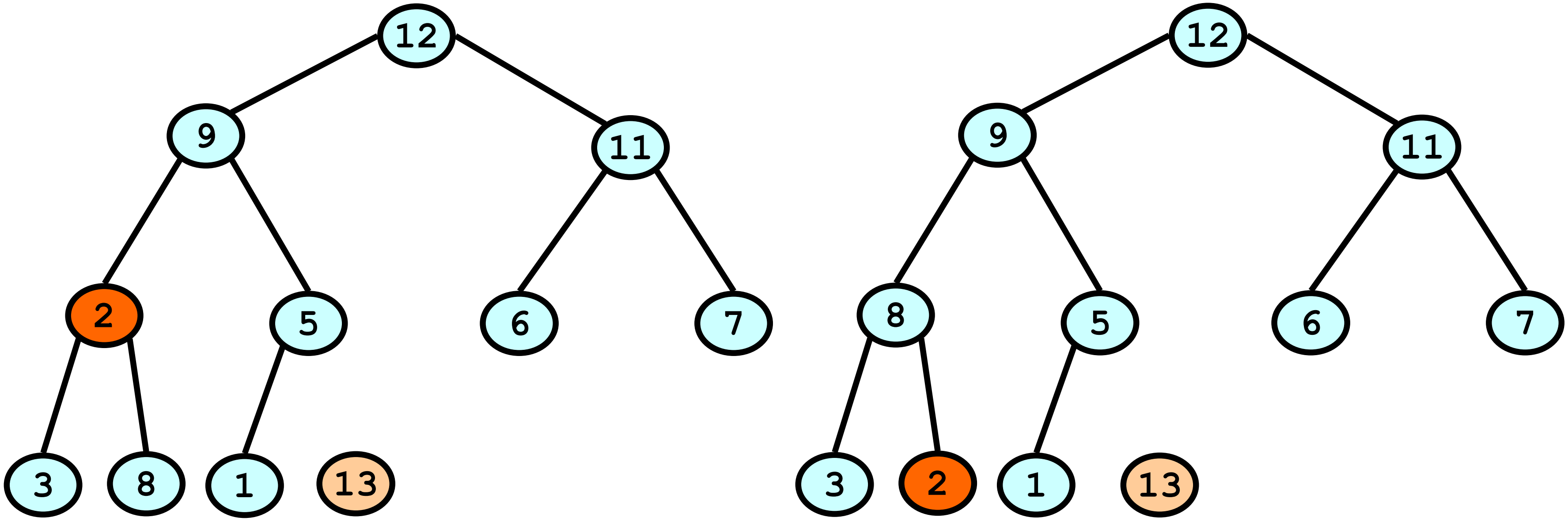
Proceed in the direction of the child node with the larger value.



Not in heap any longer

HeapExtractMax( ) - Remove the item at the root

Proceed in the direction of the child node with the larger value.



Heap property satisfied.

O(log N) operations.

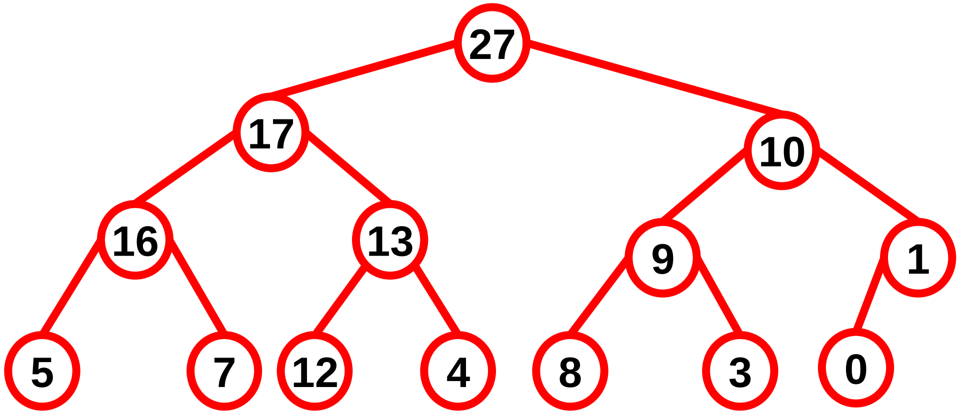
Heapsort( )

Illustrate the operation of Heapsort(A) on the array whose element values are:

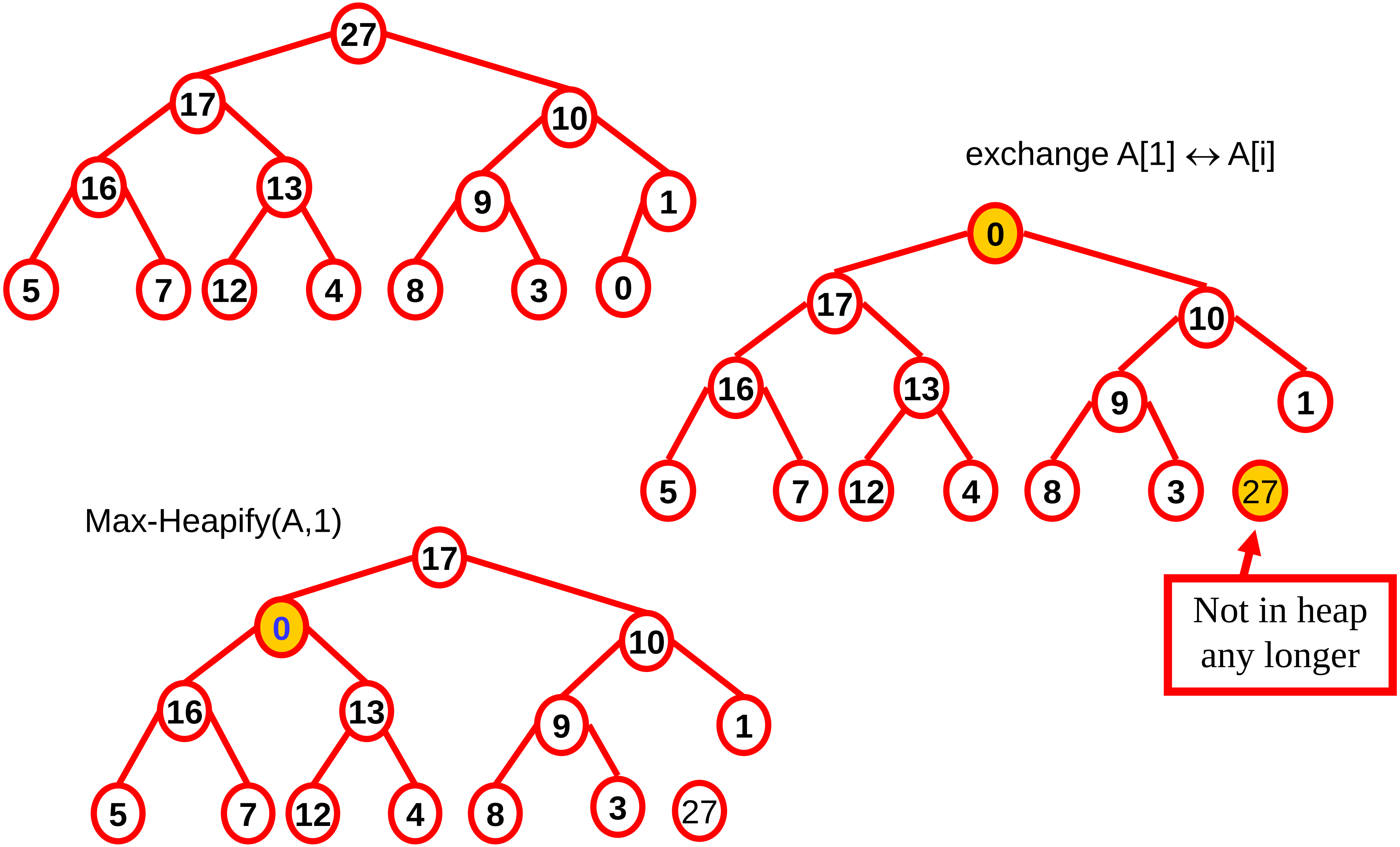
27	**17**	**10**	**16**	**13**	**9**	**1**	**5**	**7**	**12**	**4**	**8**	**3**	**0**
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Note: Since there a 14 nodes, and lg 8 = 3 < log 14 < log 16 = 4,

- there are 4 levels (0,1,2 and 3) in the tree,
- The height of the tree is 3.



Heapsort

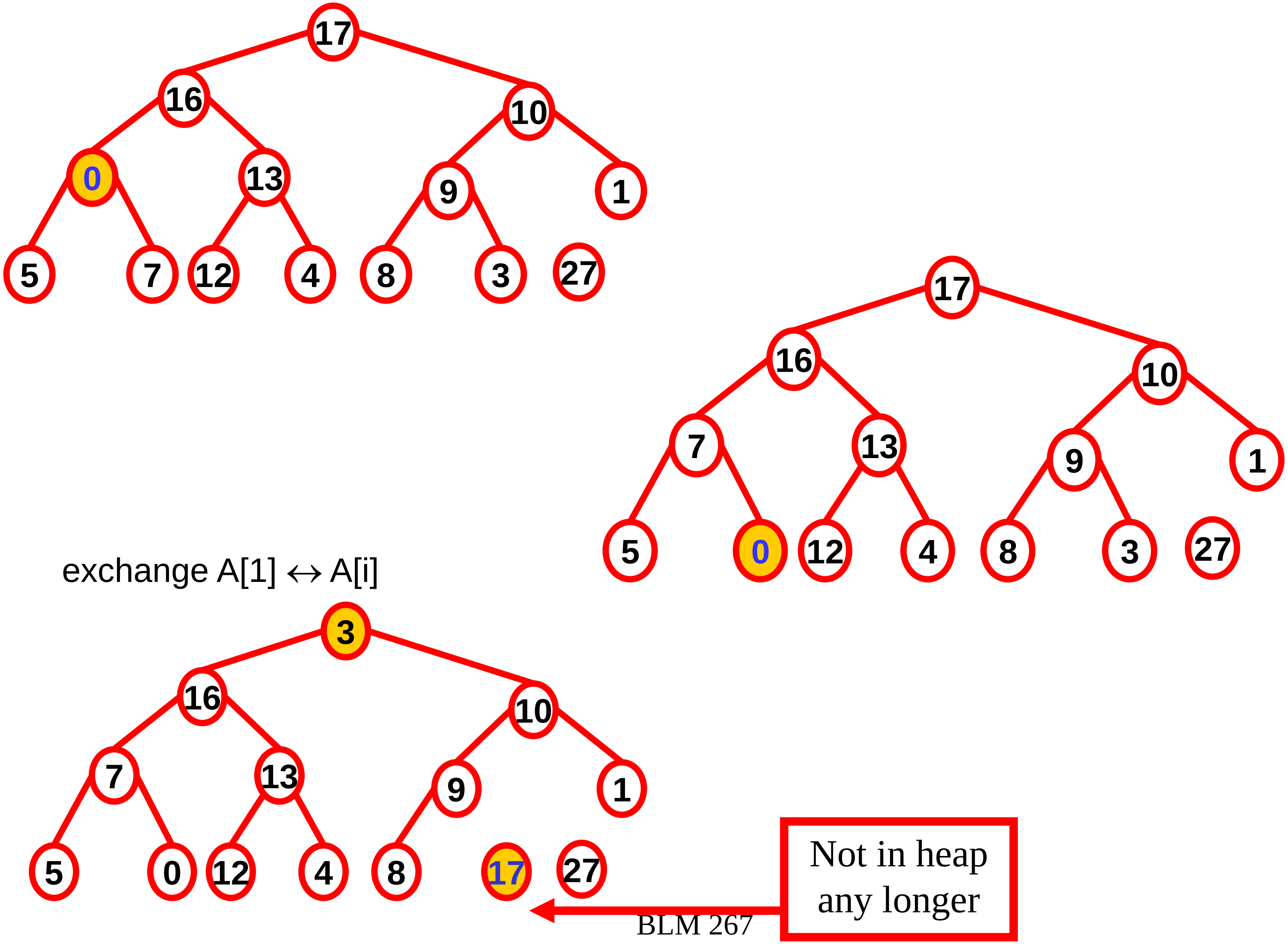


exchange A[1] ↔ A[i]

Not in heap any longer

Max-Heapify(A,1)

Heapsort

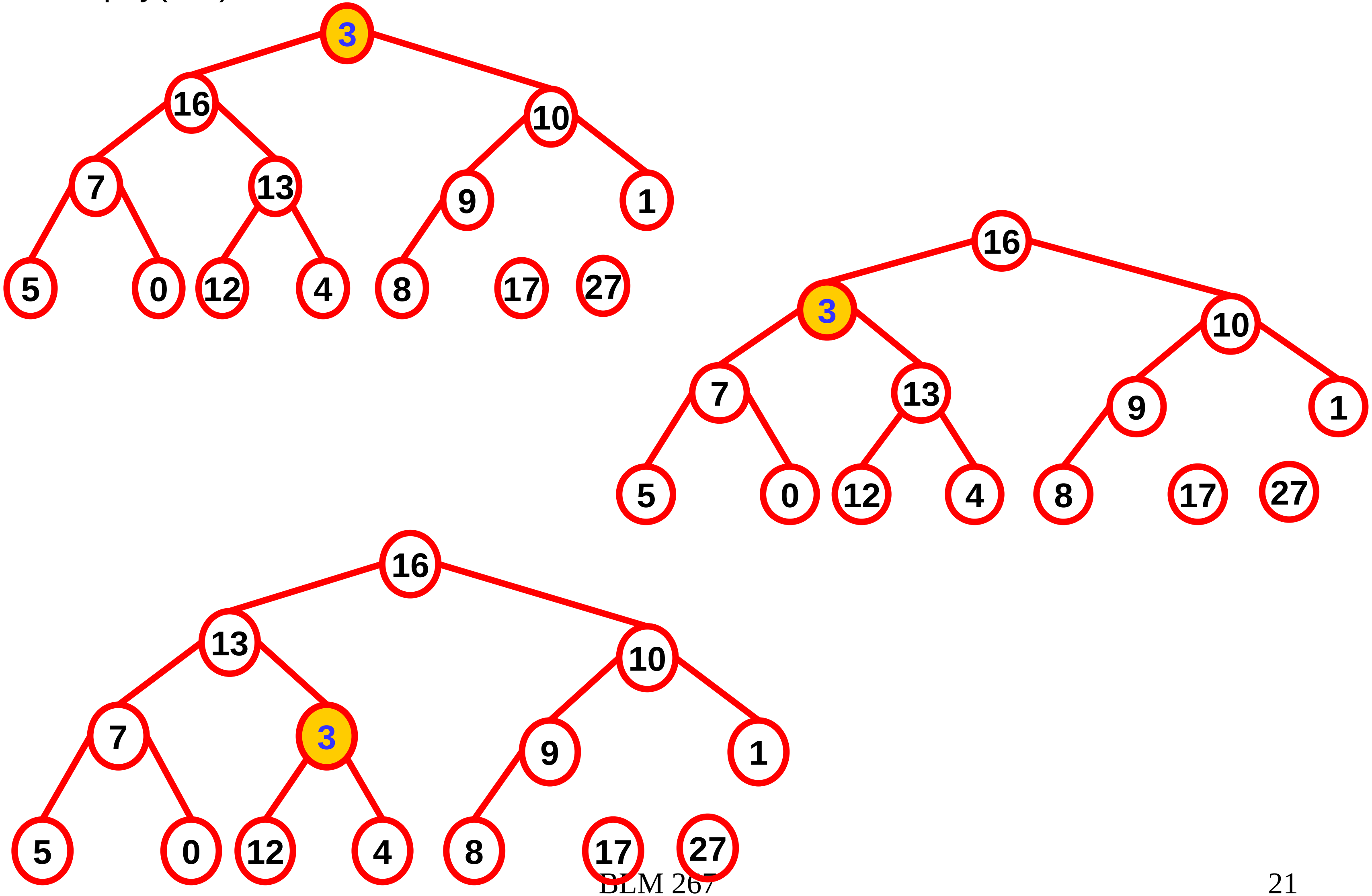


exchange A[1] ↔ A[i]

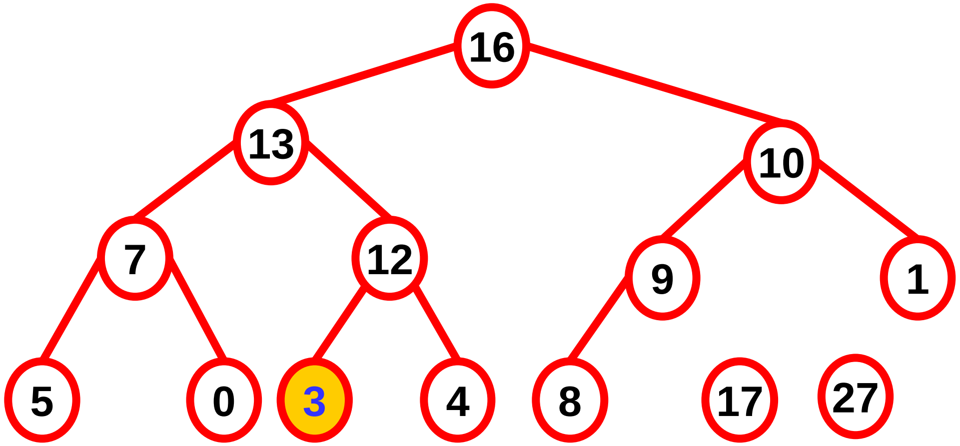
Not in heap any longer

Heapsort

Max-Heapify(A,1)



Heapsort



Heap property is satisfied for the first N-2 elements.

Last two elements are in their (sorted) place.

Perform N-1 extract-max operations during sort.

O(N log N).

No extra storage.

Heapsort - Analysis

- Here’s how the heapsort algorithm starts:

  `heapify the array;`

- Heapifying the array: we add each of **n** nodes
  - Each node has to float up, possibly as far as the root
  - Since the binary tree is perfectly balanced, sifting up a single node takes `O(log n)` time
  - Since we do this `n` times, heapifying takes `n*O(log n)` time, that is, `O(n log n)` time

Heapsort - Analysis

- Here’s the rest of the algorithm:

```
while the array isn’t empty {
    remove and replace the root;
    heapify the new root node;
}
```

- We do the while loop `n` times (actually, `n-1` times), because we remove one of the `n` nodes each time
- Removing and replacing the root takes `O(1)` time
- Therefore, the total time is `n`. How long does the **heapify( )** operation take?

Heapsort - Analysis

- To heapify the root node, we have to follow *one path* from the root to a leaf node (and we might stop before we reach a leaf)
- The binary tree is perfectly balanced
- Therefore, this path is O(log n) long
  - And we only do O(1) operations at each node
  - Therefore, heapify( ) takes O(log n) times
- Since we heapify inside a while loop that we do n times, the total time for the while loop is

  n*O(log n), or O(n log n)

Heapsort - Analysis

- Here’s the algorithm again:

```
heapify the array;
while the array isn’t empty {
    remove and replace the root;
    reheap the new root node;
}
```

- We have seen that heapifying takes O(n log n) time
- The while loop takes O(n log n) time
- The total time is therefore O(n log n) + O(n log n)
- Which is equivalent to O(n log n) time

Priority Queue

- **Problem:**
- Maintain a dynamically changing set S so that every element in S has a ***priority*** (***key***) k.
- Allow efficiently reporting the element with maximal priority in S.
- Allow the priority of an element in S to be increased.

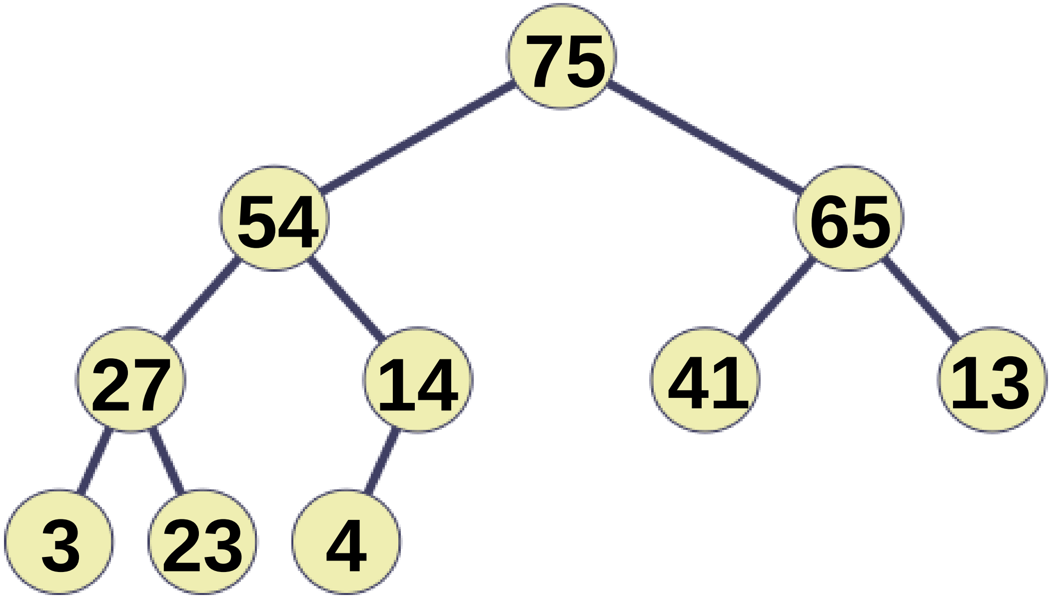
Priority Queue

- A ***(max-)priority queue*** supports the following operations:
  - **Insert(*S*, *x*, *k*):** Insert element *x* into *S* and give it priority *k*.
  - **Delete(*S*, *x*):** Delete element *x* from *S*.
  - **Find-Max(*S*):** Report the element with maximal priority in *S*.
  - **Delete-Max(*S*):** Report the element with maximal priority in *S* and remove it from *S*.
  - **Change-Priority(*S*, *x*, *k*):** Change the priority of *x* to *k*.

Binary Heap as Priority Queue

- Binary heaps are binary trees that satisfy the following ***heap property***:

- For every node v with parent u, let k_v and k_u be the priorities of the elements stored at v and u.
- Then $\mathbf{k_v \leq k_u}$.



Heaps, Priority Queues

- ***Priority queues*** support operations:
  - Insert, Delete, and Increase-Key
  - Find-Max and Delete-Max
- ***Binary heaps*** are priority queues that support the above operations in $O(\lg n)$ time.
- ***We can sort using a priority queue.***
- ***Heapsort***:
  - Sorts using the priority-queue idea
  - Takes $O(n \lg n)$ time (as Mergesort)
  - Sorts in place (as Insertion Sort)