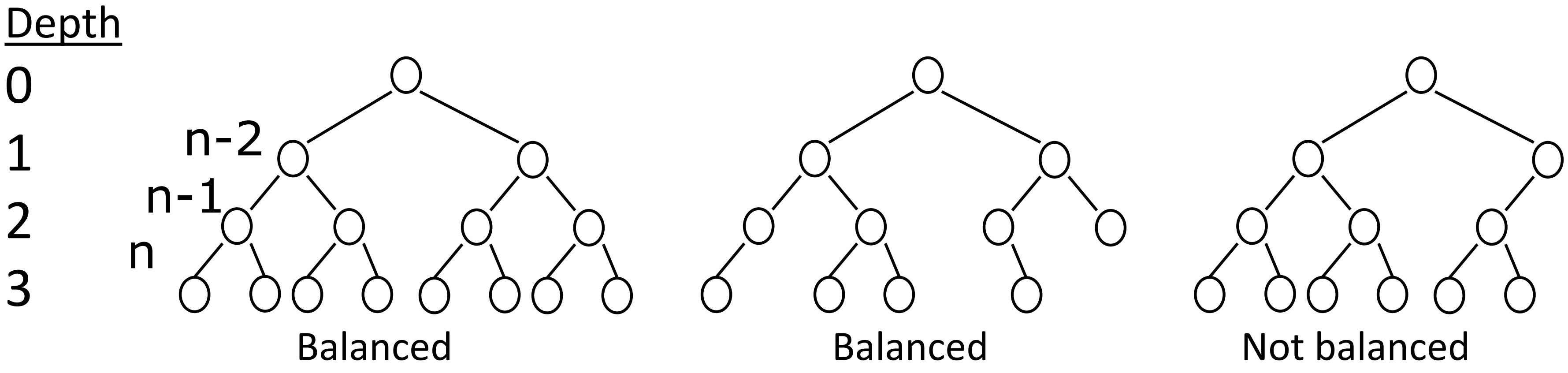
Heap Structures

Heapsort
Priority Queues

Prof.Dr.Mehmet Cudi Okur

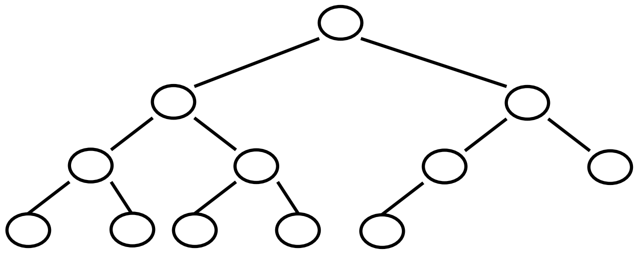
Reminder : Balanced binary trees

- Recall:
  - The depth of a node is its distance from the root
  - The depth of a tree is the depth of the deepest node
- A binary tree of depth n is balanced if all the nodes at depths 0 through n-2 have two children

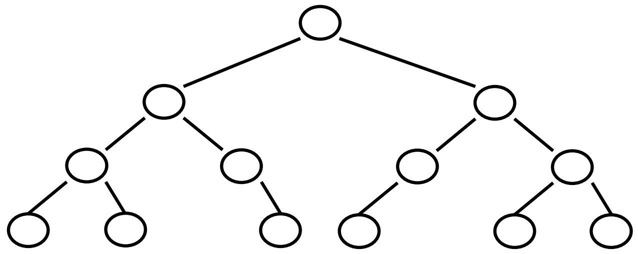


Left-justified Binary Trees

- A balanced binary tree is left-justified if:
  - all the leaves are at the same depth, or
  - all the leaves at depth n+1 are to the left of all the nodes at depth n



Left-justified

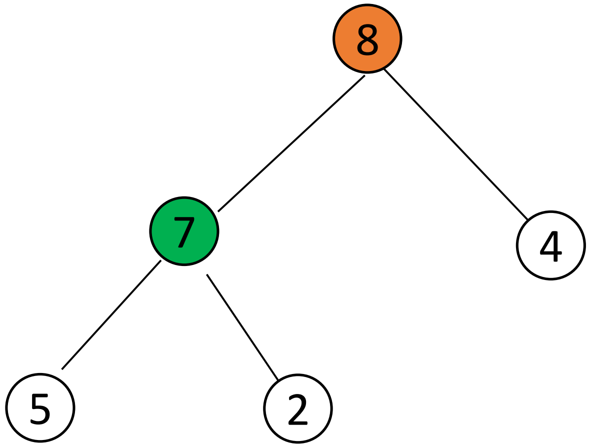


Not left-justified

The Heap Data Structure

- A heap is a binary tree with the following two properties:
  - Structural property: all levels are full, except possibly the last one, which is filled from left to right (left-justified)
  - Order (Maxheap) property : for any node x

$$\text{ValueParent}(x) \geq \text{Value}(x)$$



Heap

From the heap property, it follows that:
"The root has the maximum value in the heap!"

Heap Types : Assume node values are in A[ ]

- Max-heaps (largest element at root, also called binary-heap) have the *max-heap property:*
  - for all nodes i, excluding the root:

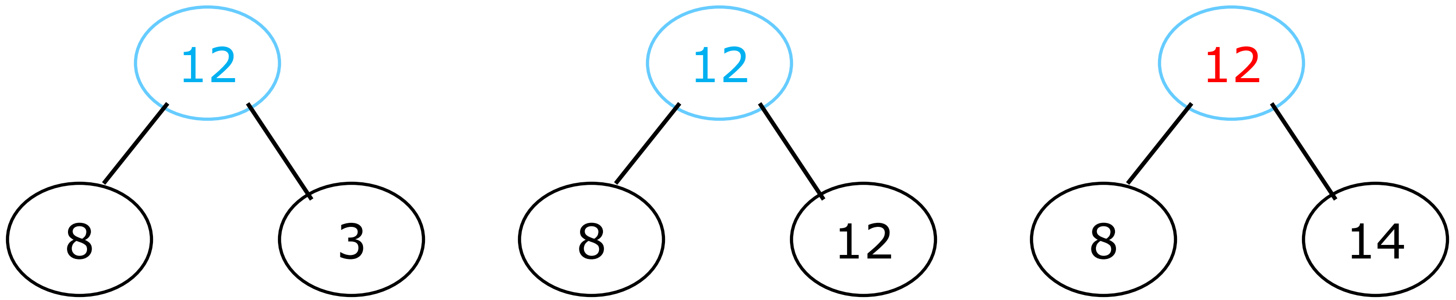
$$A[\mathrm{PARENT}(i)] \geq A[i]$$

- Min-heaps (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

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

The Heap Property: Explanation

- A node has the heap property (Max-heap) if the value in the node is as large as or larger than the values in its children



Blue node has heap property

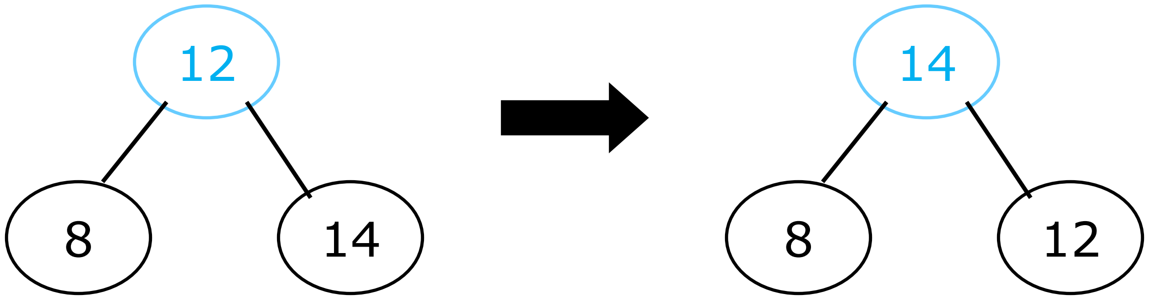
Blue node has heap property

Red node does not have heap property

- A binary tree is a heap if *all* nodes in it have the heap property

Restoring Heap Property : Move Up

- Given a node that does not have the heap property, we can give it the heap property by exchanging its value with the value of the larger child (Move-up the child or push down the parent)



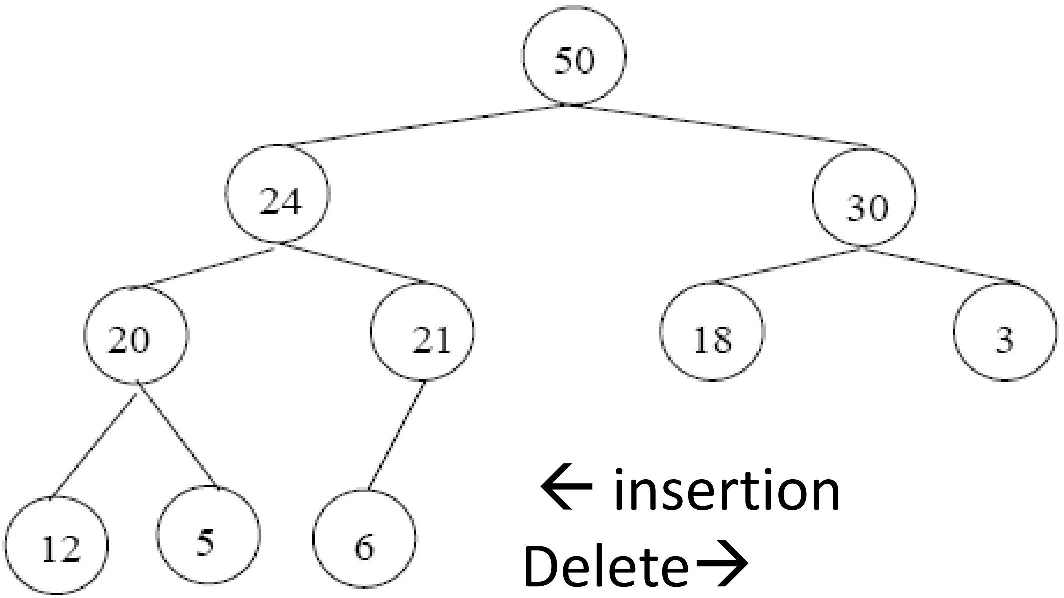
Blue node does not have heap property

Blue node has heap property

- This is sometimes called sifting (shifting) up
- Notice that now , the child may have *lost* the heap property

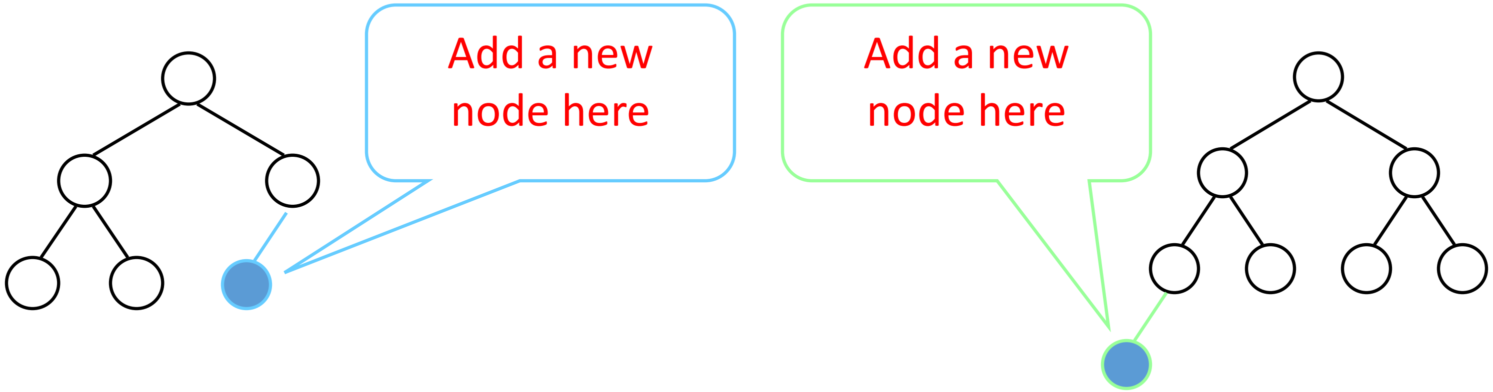
Adding/Deleting Nodes

- New nodes are always inserted at the bottom level: left to right
- Nodes are removed from the bottom level: right to left



Constructing a Heap : Heapify

- A tree consisting of a single node is a heap by definition
- We construct a heap by adding nodes one at a time:
  - Add the node just to the right of the rightmost node in the deepest level
  - If the deepest level is full, start a new level
- Examples:

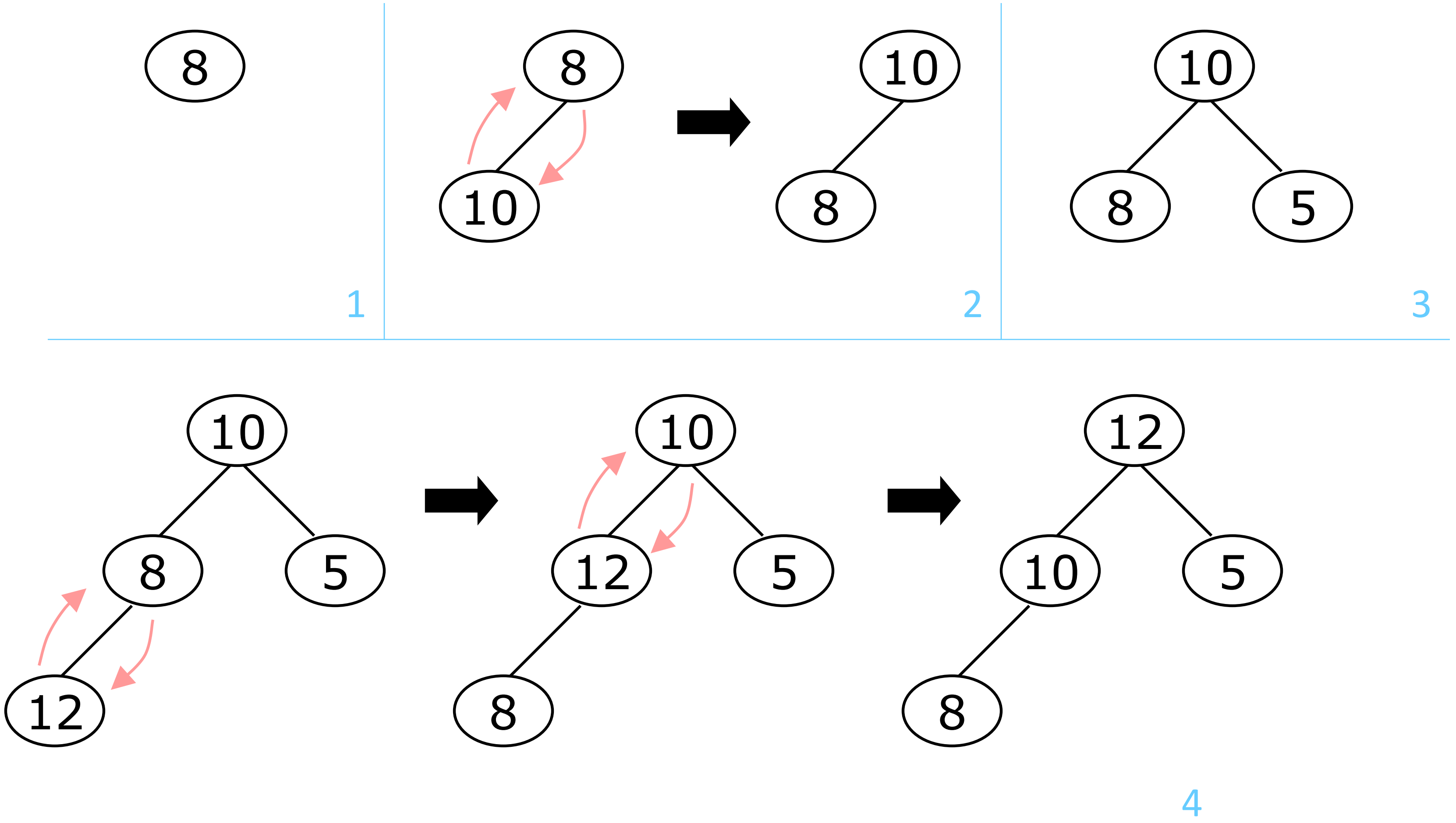


Constructing a Heap

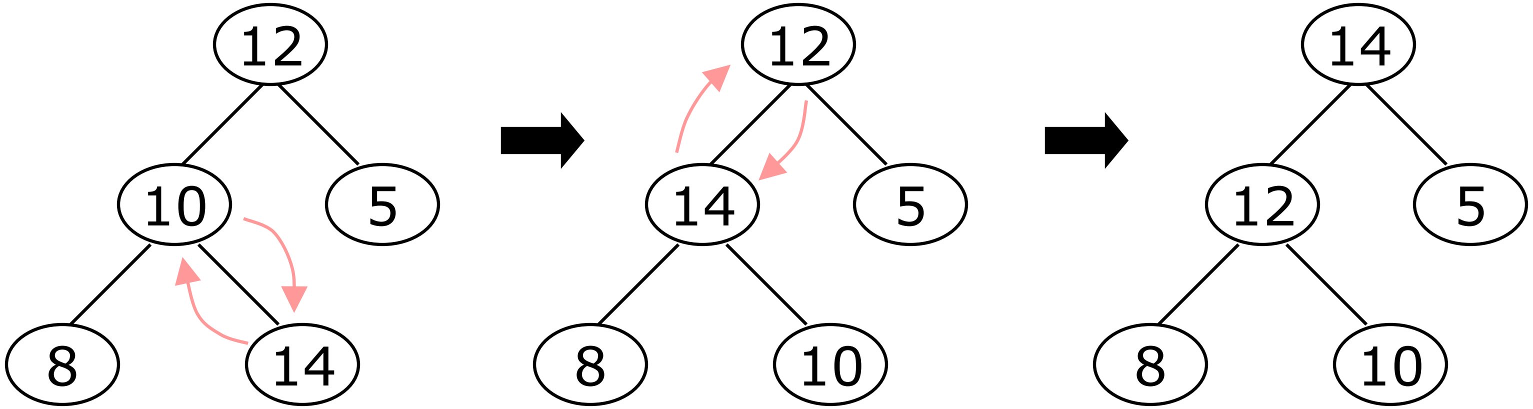
- Each time we add a node, we may destroy the heap property of its parent node
- To fix this, we apply shift up
- But each time we shift up, the value of the topmost node in the shift may increase, and this may destroy the heap property of *its* parent node
- We repeat the shifting up process, moving up in the tree, until either
  - We reach nodes whose values don't need to be swapped (because the parent is *still* larger than both children), or
  - We reach the root

Constructing a Heap : Example

Construct a heap(Maxheap) by Inserting in this order : 8,10,5,12.



Constructing a Heap : Insert a new element to the heap :14

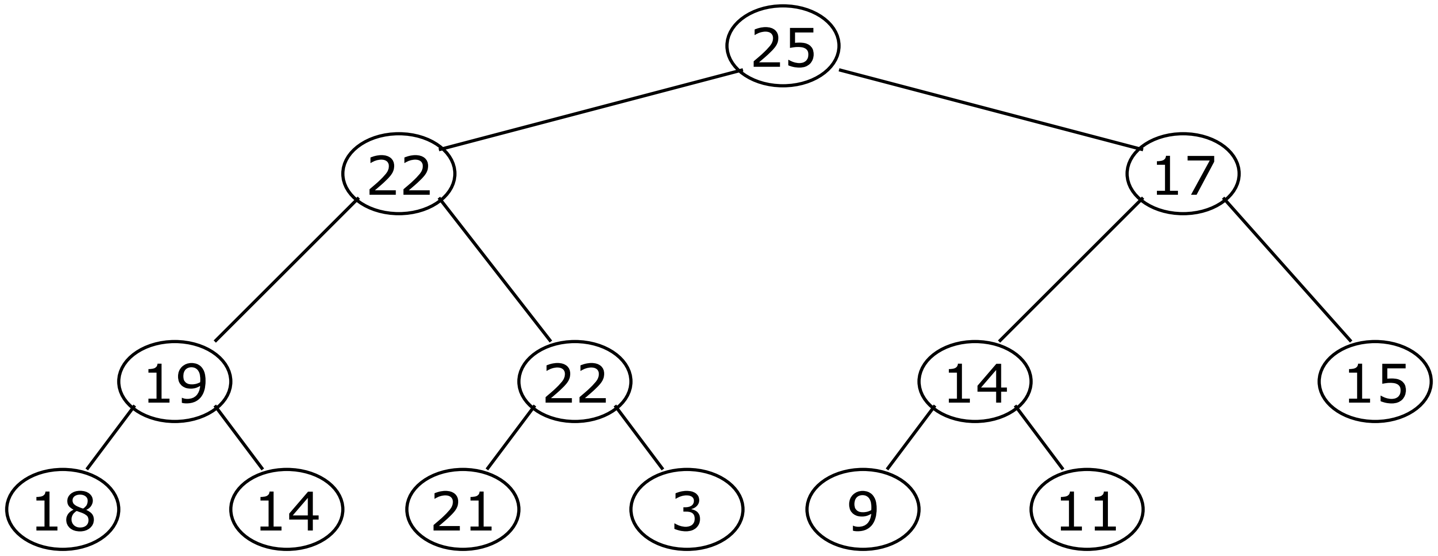


- The node containing 8 is not affected because its parent gets larger, not smaller

- The node containing 5 is not affected because its parent gets larger, not smaller
- The node containing 8 is still not affected because, although its parent got smaller, its parent is still greater than it was originally

A Sample Maxheap

- Here's a sample binary tree after it has been heapified

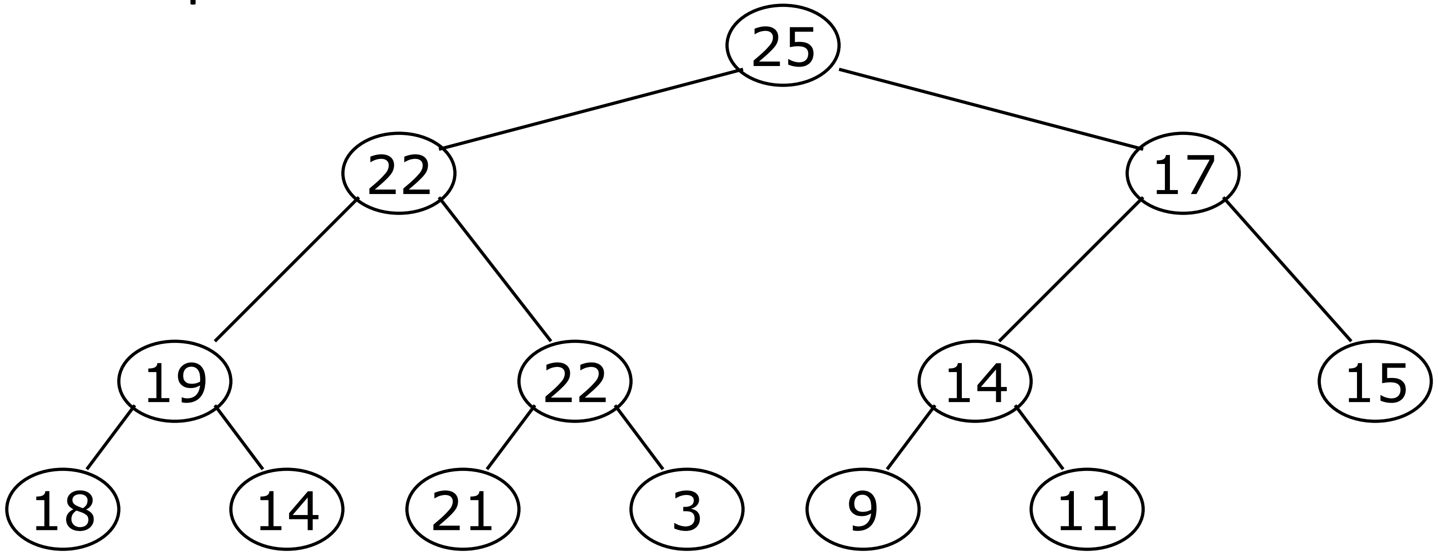


- Heapifying does *not* change the shape of the binary tree; the tree is balanced and left-justified, because it started out that way
- The height of a heap tree is h= $\lfloor \log n \rfloor$, n=Number of nodes
- Access time to the element with highest value is constant: O(1).

Mapping a Heap Into an Array

Heap trees are stored (represented) using an array A.

Example : Maxheap



1	2	3	4	5	6	7	8	9	10	11	12	13
25	22	17	19	22	14	15	18	14	21	3	9	11

- Node values of the heap are now the elements of an array such that :
  - The left child of index i is at index $2*i$
  - The right child of index i is at index $2*i+1$
    - Example: the children of node 3 (17) are at 6 (14) and at 7 (15)
  - For any element in array position i :
    - the parent is in position $\lfloor i/2 \rfloor$.

Mapping a Heap Into an Array : Minheap example

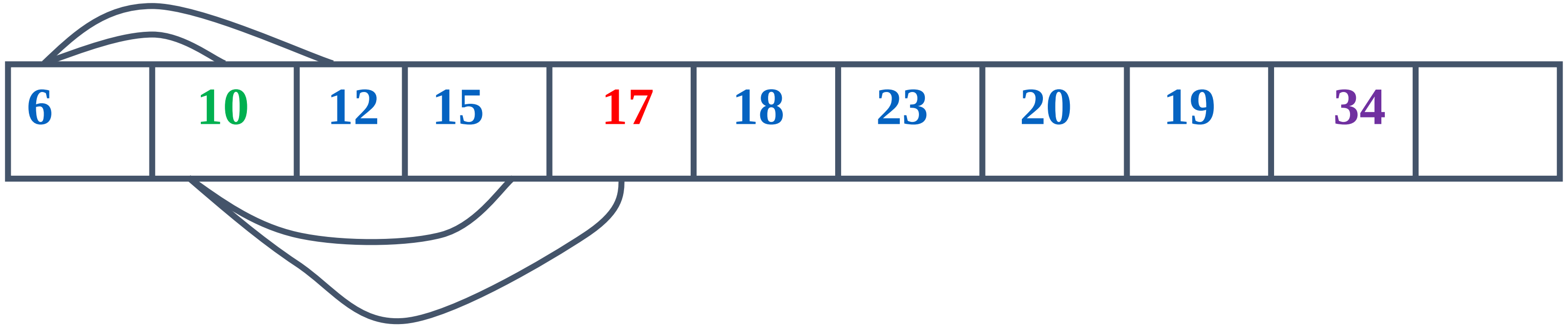
The node and index relations are the same for minheaps and maxheaps:

Minheap: For every subtree,the smallest element is always at the root.

Root – A[1]

Left Child of A[i]	-	A[2i]
Right child of A[i]	-	A[2i+1]
Parent of A[i]	-	A[$\lfloor i/2 \rfloor$]

Mapping a Minheap Into an Array : Example



Consider 17:

position in the array : 5.

parent is at position floor(5/2) = 2 (10)

left child is at position 5*2 = 10 (34)

right child - position 2*5 + 1 = 11 (empty.)

Operations on Heaps

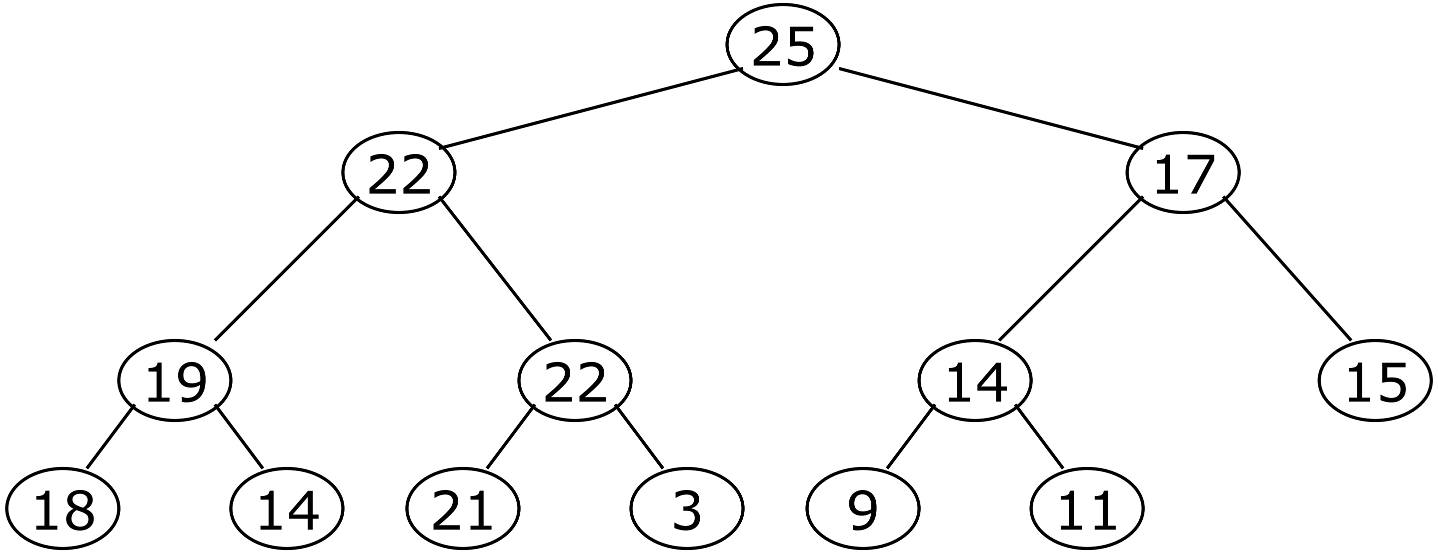
- Extract (Remove) root (max/min value) from the heap
- Maintain/Restore the heap property
  - HEAPIFY
- Create a heap from an unordered array
  - BUILD-HEAP
- Sort an array in place
  - HEAPSORT
- Priority queue applications

In the following parts we are going to consider maxheaps

All operations are also valid for minheaps.

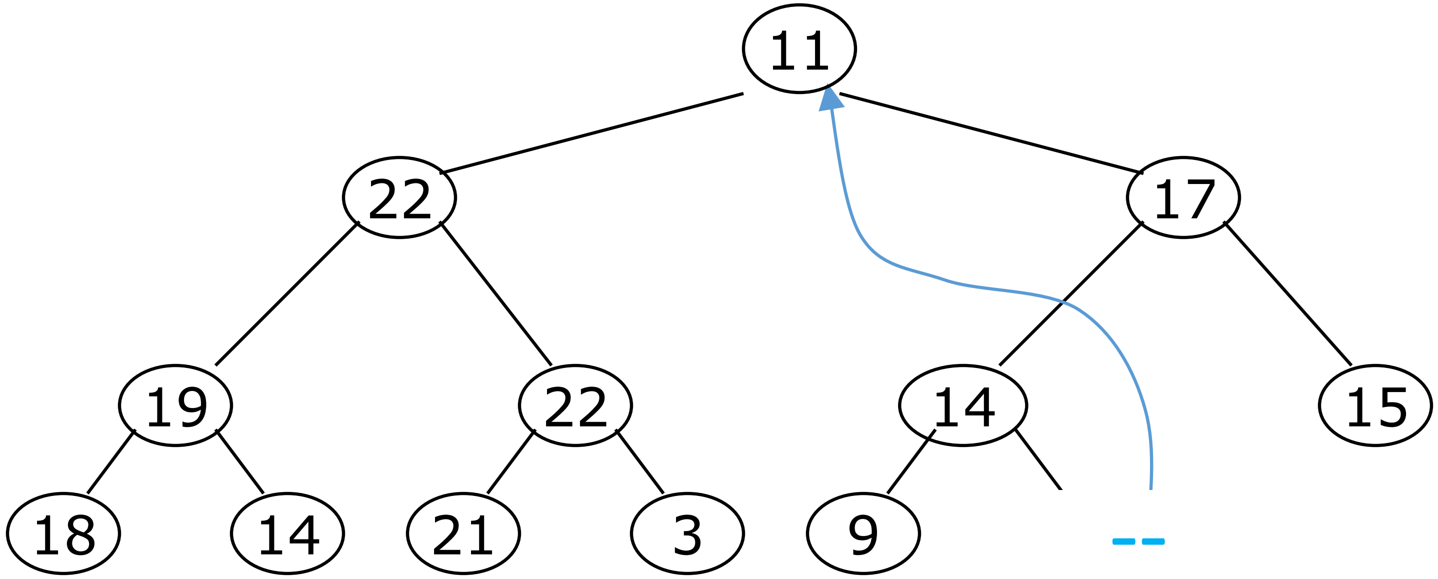
Removing the root

Notice that the largest value is always in the root
Suppose we *discard the* root:



Removing the root

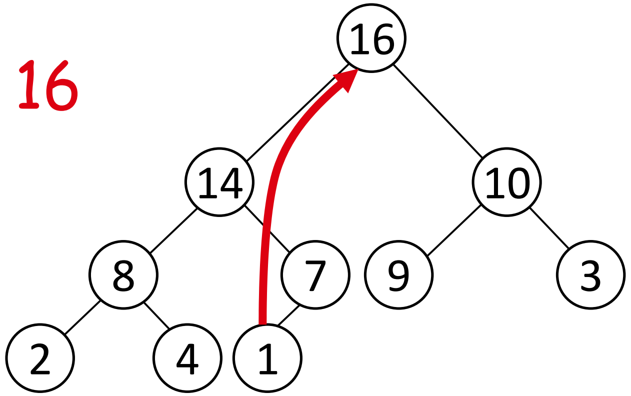
- How can we fix the binary tree so it is once again a *balanced and left-justified heap?*
- Solution: remove the rightmost leaf at the deepest level and use it for the new root and heapify.

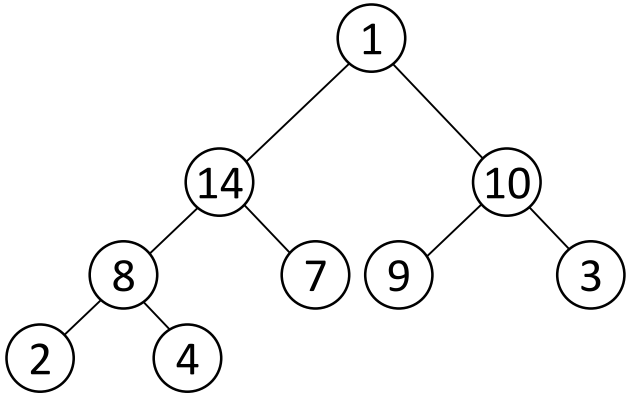


Example: EXTRACT-MAX and Heapify

Remove the root(16),move up the last element(1) to the root .Heapify the tree again.

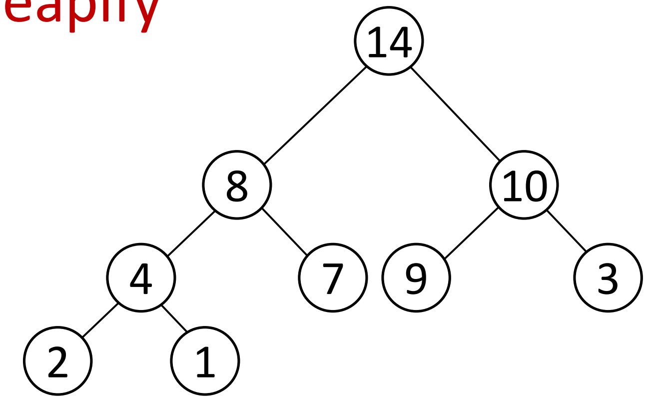
max = 16





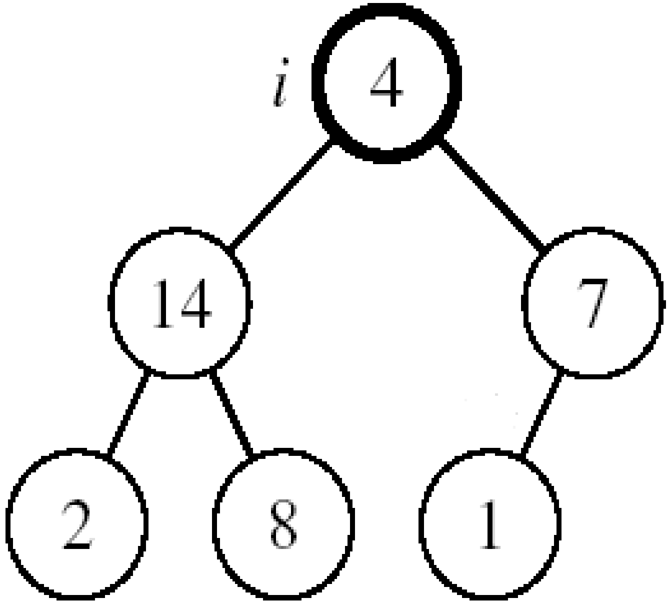
Heap size is decreased by one

Restore the maxheap: Heapify



Max-Heapify : Recursive Pseudocode

- $A[i]$ may be smaller than its children



//Restores max-heap property in an array A

Algorithm MAX-HEAPIFY(A, i, n)

  $l \leftarrow$ LEFT(i)   // Left index

  $r \leftarrow$ RIGHT(i)   //Right index

  if $l \leq n$ and $A[l] > A[i]$

    then $largest \leftarrow l$

    else $largest \leftarrow i$

  if $r \leq n$ and $A[r] > A[largest]$

    then $largest \leftarrow r$

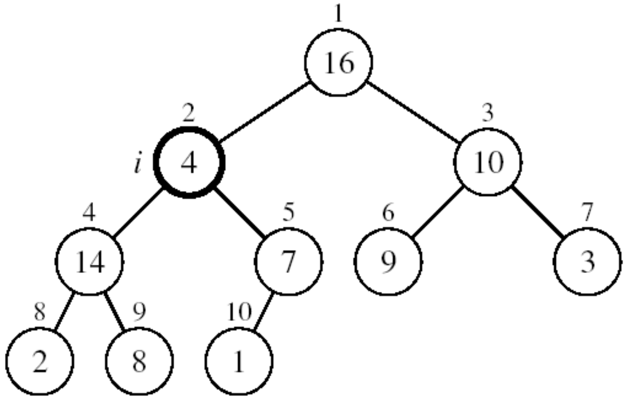
  if $largest \neq i$

   then swap ($A[i]$,$A[largest]$)

  MAX-HEAPIFY(A, $largest$, n) //Recursive call

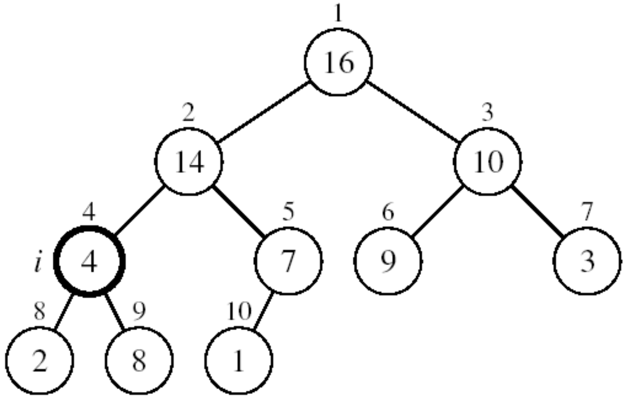
Example : Heapify (max-heapify)

MAX-HEAPIFY(A, 2, 10)



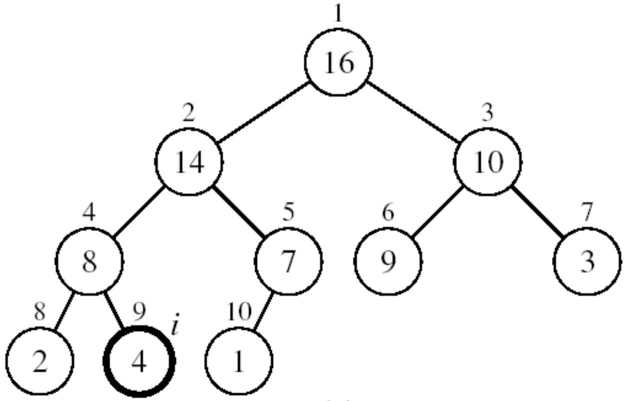
A[2] violates heap property
Largest index: l=4

A[2] ↔ A[4]



A[4] violates heap property
Largest index : r=9

A[4] ↔ A[9]



Heap property restored

MAX-HEAPIFY Complexity

- How many comparisons are needed in the worst case ?

It traces a path from the root to a leaf (longest path length: d)
At each level, it makes exactly 2 comparisons
Total number of comparisons is $2d$
Running time is $O(d)$ or $O(lgn)$

→ Running times are determined by the height of the heap tree.

→The complexity of MAX-HEAPIFY is $O(logn)$

or in terms of the height of the heap tree : $O(h)$

Building a Heap Using Max-Heapify

- Convert an array $A[1 \ldots n]$ into a max-heap array
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1),\ldots ,n]$ are leaves
  Ex: n=10, leaves start at node : (10/2)+1 = 6
- We can use max-heapify function to make a max heap out of the array

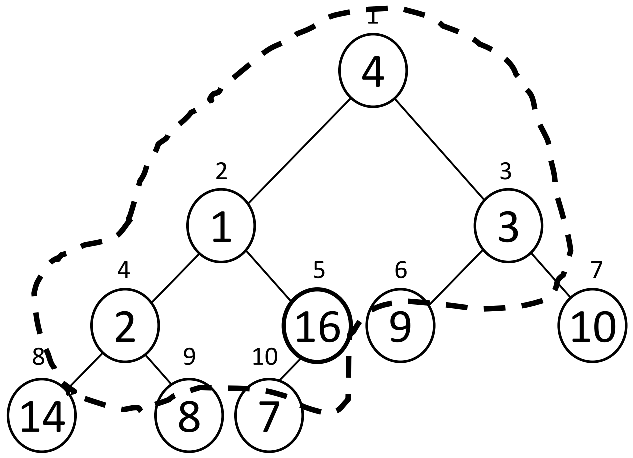
→Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

ALGORITHM BUILD-MAX-HEAP(A)

n = length[A]

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

do MAX-HEAPIFY(A, i, n) //call



A:

4	1	3	2	16	9	10	14	8	7

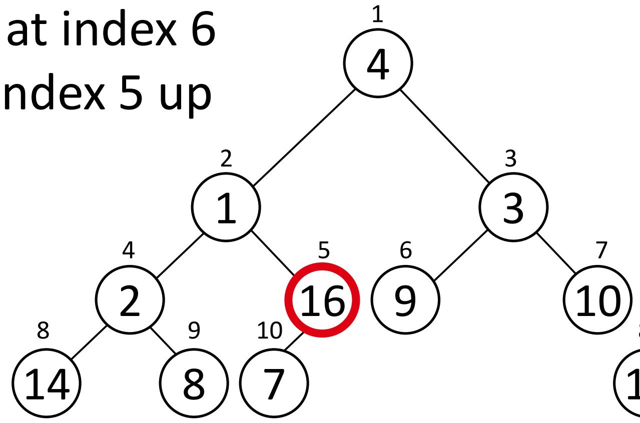
Example: A

4	1	3	2	16	9	10	14	8	7

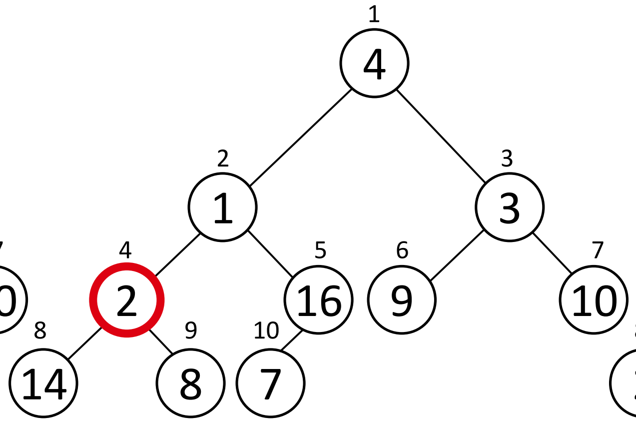
Leaves start at index 6

Move from index 5 up

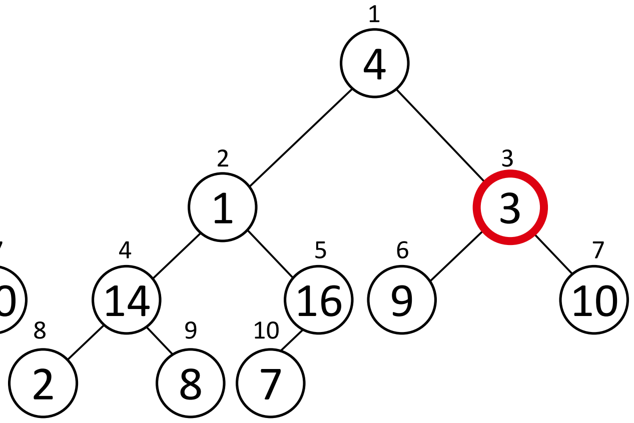
$i = 5$



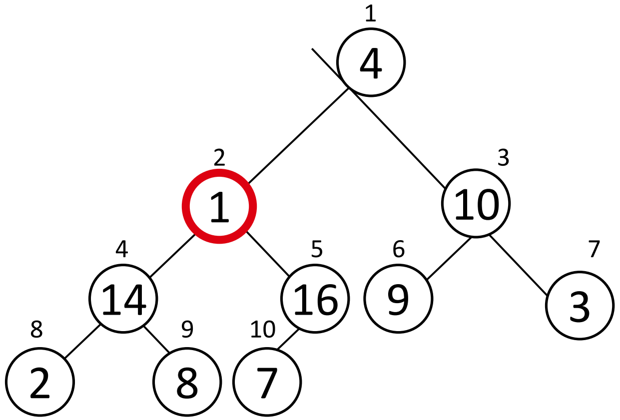
$i = 4$



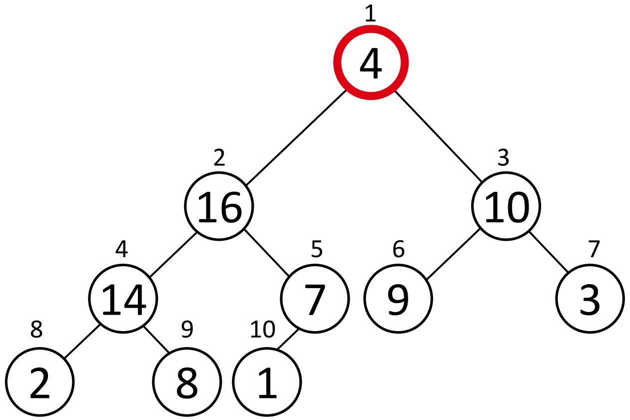
$i = 3$



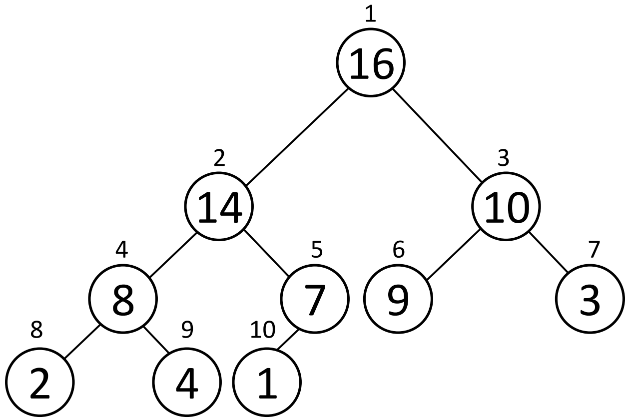
$i = 2$



$i = 1$



Maxheap



Complexity of BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

n = length[A]

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

do MAX-HEAPIFY(A, i, n) $\quad O(\log n)$ $\Big\} \; O(n)$

$\Rightarrow$Running time: n logn

Complexity ~ $O(n \log n)$

This is not an asymptotically tight upper bound.

It can be shown that the actual complexity is lesser : O(n).

A Different Sort Method : Heapsort

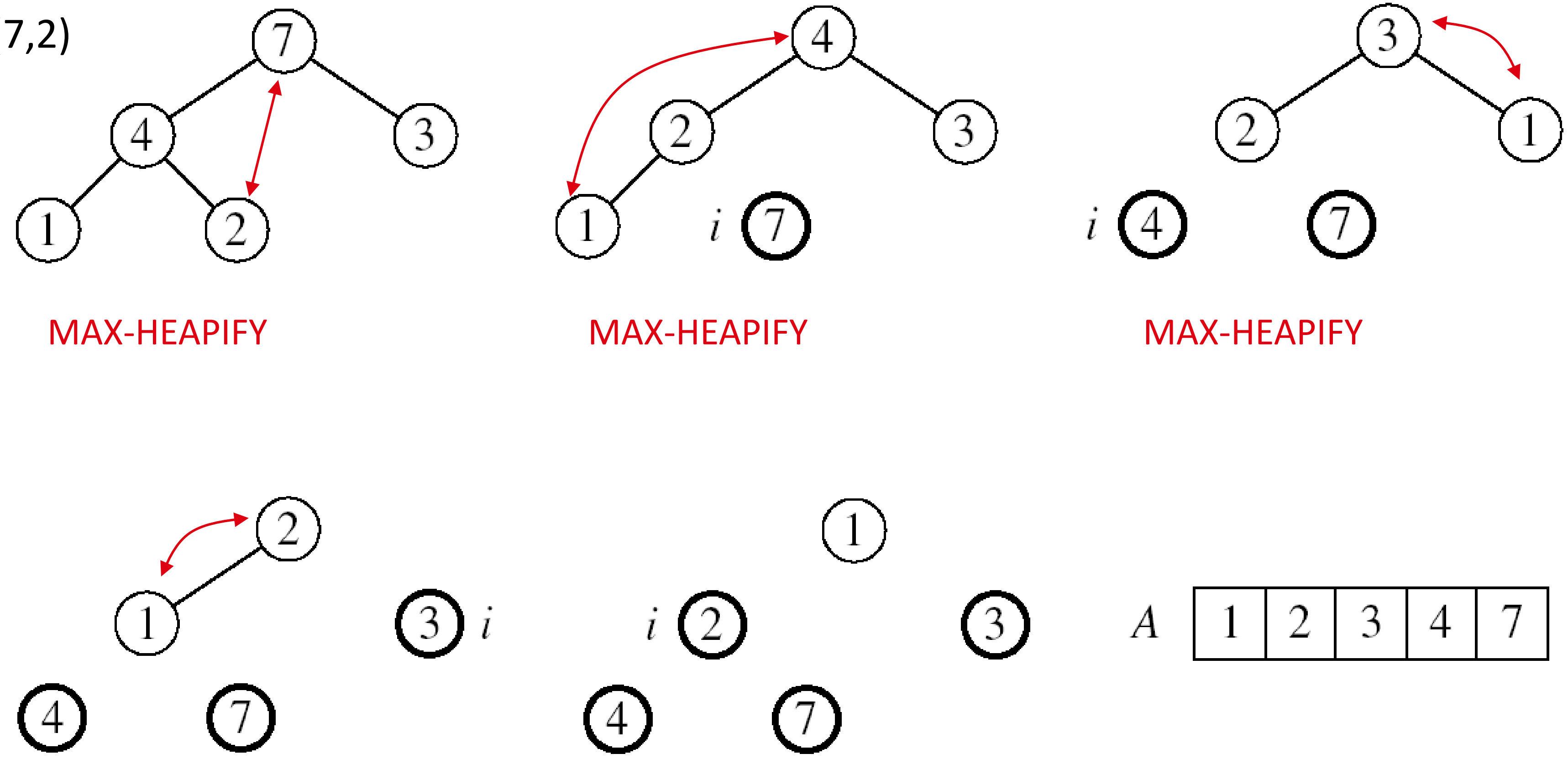
- Goal : Sort an array using heap representations
- Informal procedure:
  - Build a max-heap from the array
    - Swap the root (the maximum element) with the last element in the array
    - "Discard" this last node by decreasing the heap size
    - Call MAX-HEAPIFY on the new root
    - Repeat this process until only one node remains

Heapsort : The Method

```
heapify the array;    //Initial operation
while the array is not empty
 {
    remove and replace the root;
    heapify the new root node;
 }
```

Example: A=[7, 4, 3, 1, 2]

Swap(7,2)



MAX-HEAPIFY

MAX-HEAPIFY

MAX-HEAPIFY

MAX-HEAPIFY

Result : Sorted array

Heap Sort Algorithm

Algorithm HEAPSORT*(A)*

BUILD-MAX-HEAP(A) //Now A is a maxheap

for i ← length[A] **downto** 2

do swap($A[1] \boxed{\leftrightarrow} A[i]$)

MAX-HEAPIFY(A, 1, i - 1) //Call Heapify

Tracing Heapsort: Removing and Replacing the Root

- The “root” is the first element in the array
- The “rightmost node at the deepest level” is the last element
- Swap them...

0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

- Pretend that the last element in the array no longer exists—that is, the “last index” is now 11.

Tracing Heapsort : Reheap and repeat

- Reheap the root node (index 0, containing 11)...

0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
22	22	17	19	21	14	15	18	14	11	3	9	25

0	1	2	3	4	5	6	7	8	9	10	11	12
9	22	17	19	22	14	15	18	14	21	3	22	25

- ...And again, remove and replace the root node
- Remember, though, that the "last" array index is changed
- Repeat until the last becomes first, and the array is sorted!

Heap Sort Analysis

- Here's how the algorithm starts: Initial creation of the heap: Initial Build-Max-Heap
- Converting the array to maxheap: we add each of n nodes
  - Each node has to be moved up, possibly as far as the root
    - Since the binary tree is perfectly balanced, sifting up a single node takes O(log n) time (Worst case)
  - Since we do this n times, heapifying takes n*O(log n) time, that is,

  O(n log n) time

Heap Sort Analysis- Reheap

- Here’s the rest of the algorithm:

```
while the array isn't empty {
    remove and replace the root;
    reheap 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 however long the heapify (reheap) method takes.

Heap Sort Analysis-Reheap

- To reheap 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)
  - And we only do O(1) operations at each node
  - Therefore, reheaping takes O(log n) times
- Since we reheap 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)

Heap Sort Analysis – Total Complexity

- Here's the algorithm again:

```
Build maxheap ;
while the array isn't empty {
    remove and replace the root;
    reheap the new root node;   //Heapify each time
}
```

- We have seen that initial build maxheap takes $O(n \log n)$ time
- The while loop takes $O(n \log n)$ time

→The total time is therefore

$$T(n) = (n \log n) + (n \log n)$$
$$= O(n \log n)$$

Heapsort Comments

- It is an O(n log n) algorithm : An efficient sort method
- Detailed analysis shows that, the average case for heapsort is poorer than quick sort.
  - However Quicksort's worst case is far worse : $O(N^2)$.
- An average case analysis of heapsort is very complicated, but empirical studies show that there is little difference between the average and worst cases.
  - Heapsort usually takes about twice as long as quicksort.
  - On average, it is more costly, but it avoids the possibility of $O(N^2)$.

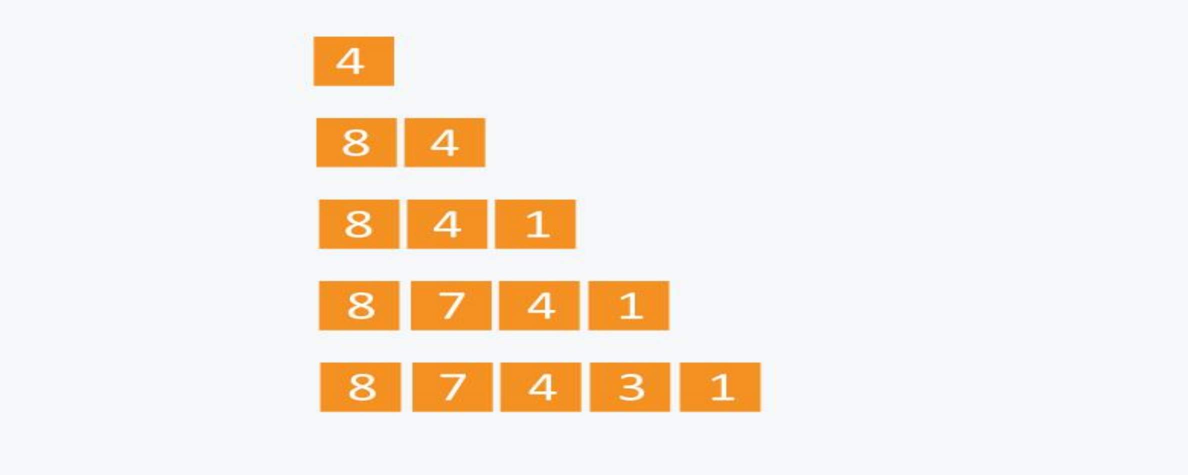
"Heapsort is a *really cool* algorithm!"

Poriority Queues

- A Priority Queue is similar to a simple queue, but the logical order of elements in the priority queue depends on their priority values.
- The element with highest priority is moved to the front of the queue and the one with lowest priority is moved to the back of the queue.

Example: Let's say we have an array with the following priority values:

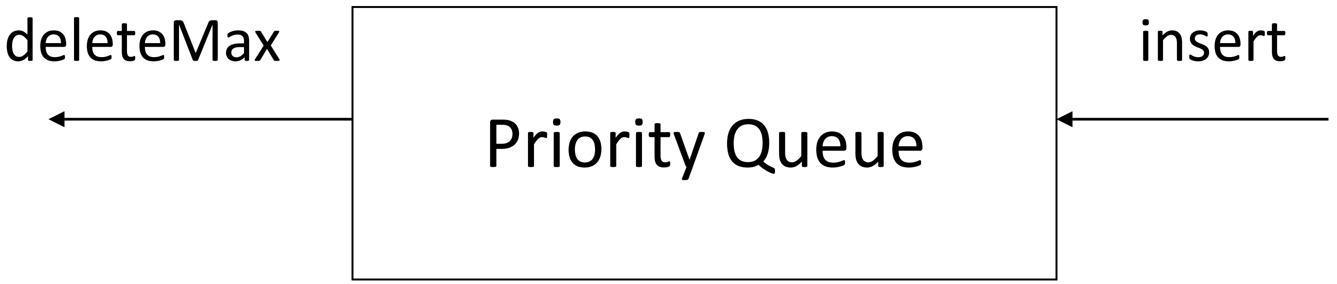
{4, 8, 1, 7, 3} . Enqueue order will be as folows:



Priority Queues

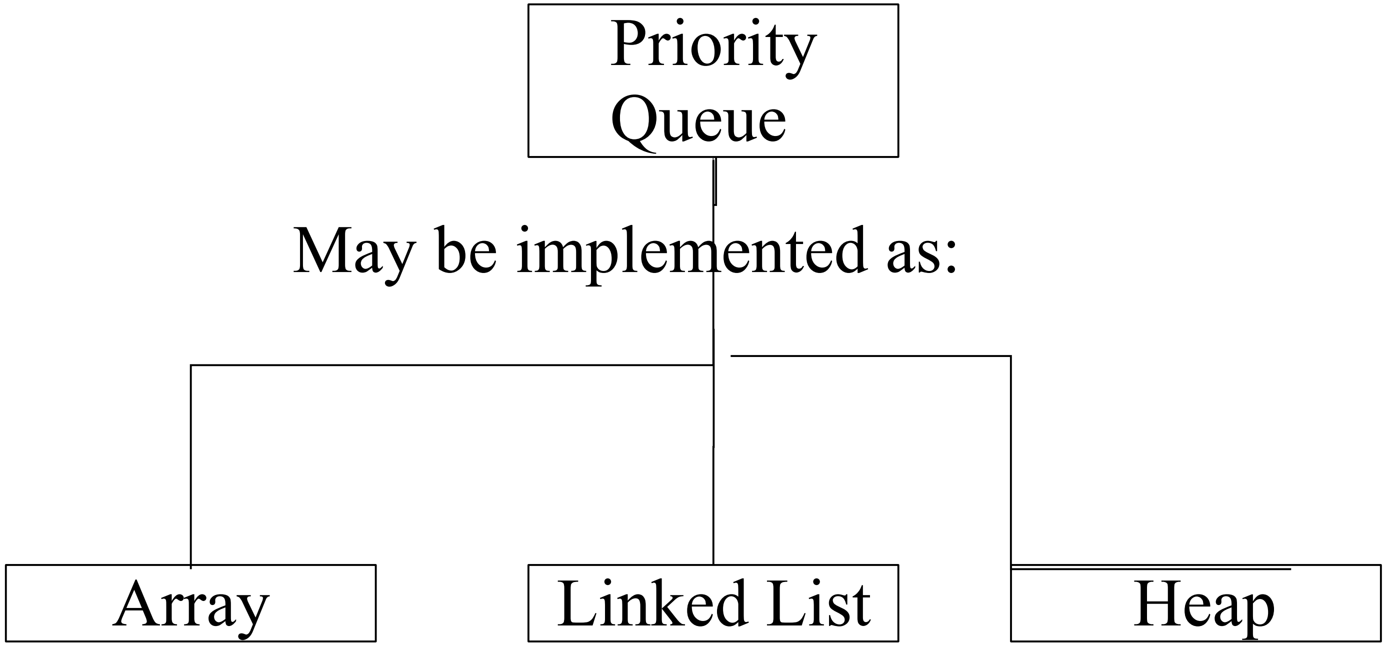
- Priority queue data structure supports two basic operations: insert a new item and remove the maximum (or minimum) item.
- In a *priority queue* items must leave from the front, but they enter(Insertion) the queue on the basis of a priority value.

→The queue is kept *sorted* by this value.



Heap is a convenient data structure to implement priority queues.

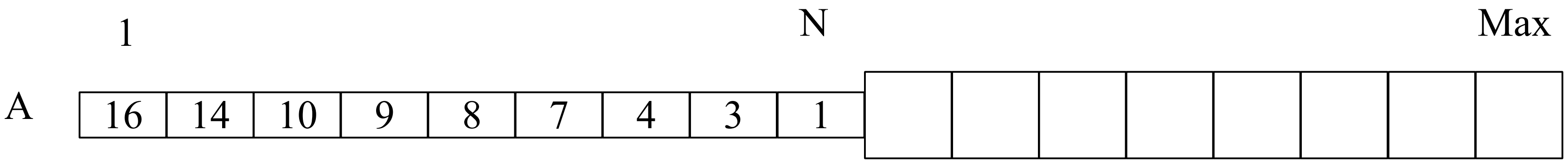
How to Implement Priority Queues?



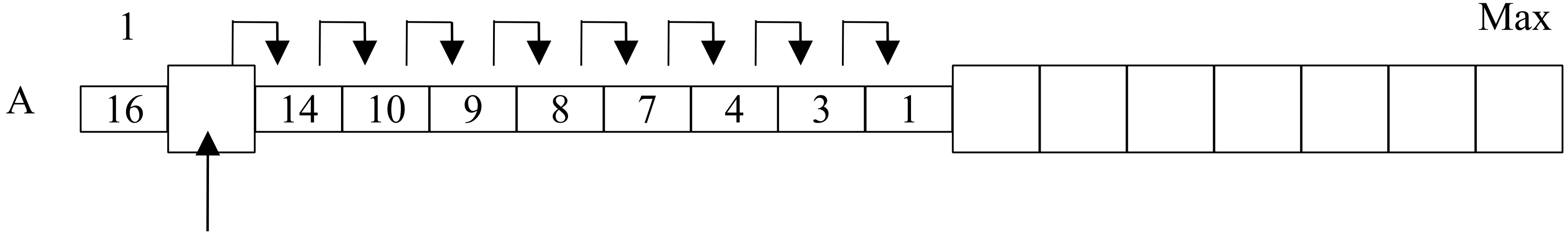
Array Implementation of Priority Queues

Suppose items with priorities 16, 14, 10, 9, 8, 7, 4, 3, 1
are to be stored in a priority queue.

Array implementation:



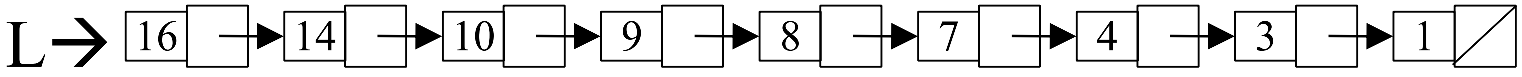
Suppose an item with priority 15 has to be added: Many shift operations!



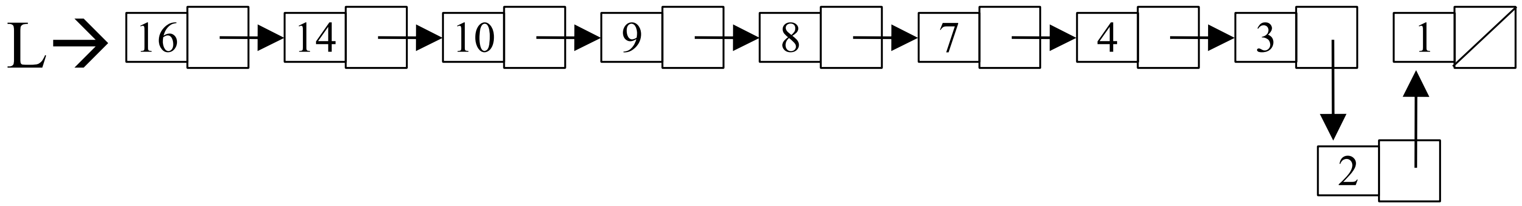
15

Thus inserting takes *O(N)* time: linear

Linked List Implementation of Priority Queues



Suppose an item with priority 2 is to be inserted:



Only *O(1)* (constant) pointer changes required, but it takes *O(N)* pointer traversals to find the location for insertion.

Wanted: a data structure for PQs that can be both searched and updated in better than *O(N)* time.

Heap Implementation of Priority Queues

- The (binary) heap is the classic dynamic method used to implement priority queues.
- When a priority queue is implemented using a heap, the worst-case times for both insert and removeMax are logarithmic.
- Array based heap implementation is the common method for priority queue representation.

Operations on Priority Queues

- Max-priority binary heap queues support the following operations:
  - INSERT(Q, x): inserts element x into Q
  - EXTRACT-MAX(Q): removes and returns the element of Q with largest key
  - MAXIMUM(Q): returns element of Q with largest key
  - INCREASE-KEY(Q, x, k): increases the value of element x's key to k (Assume $k \geq x$'s current key value)

Extracting Maximum Element

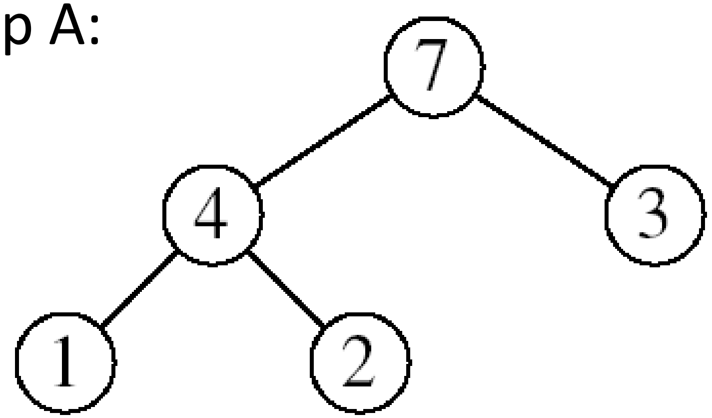
Goal:

- Return the largest element of the heap without removing it.

Algorithm HEAP-MAXIMUM(A)

return $A[1]$

Heap A:



Heap-Maximum(A) returns 7

Complexity: $O(1)$

Extract Max and Heapify the Queue

Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Algorithm:

- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1.

Extract Max and Heapify the Queue

//Priority queue delete max operation

Algorithm HEAP-EXTRACT-MAX(A, n)

if $n < 1$

then error "heap underflow"

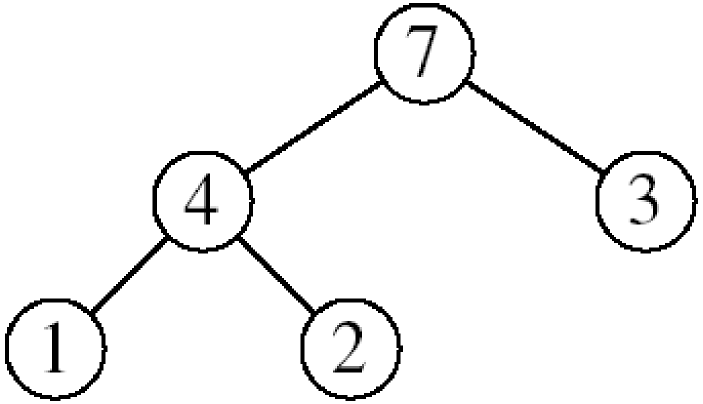
$max \leftarrow A[1]$

$A[1] \leftarrow A[n]$

MAX-HEAPIFY(A, 1, n-1) // remakes heap

Complexity : $O(\log n)$

return max

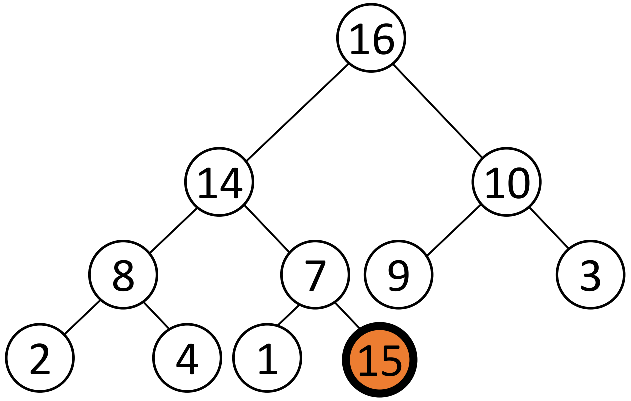


Inserting into a Priority Queue:Max-heap-insert

- Goal:
  - Inserts a new element into a priority queue
- Idea:
  - Expand the max-heap with a new element. Start from the rightmost position.
  - If the max-heap property does not hold anymore: Heapify.

→Traverse a path toward the root to find the proper place for the newly increased key.

Example: Insert 15.



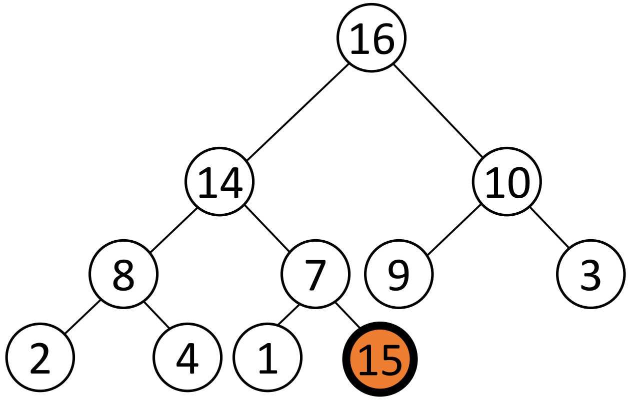
Implementing Priority Queues : Heap Insert

```
//Inserts key into maxheap A
HeapInsert(A, key)
{
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1  AND  A[Parent(i)] < key)
    {
        A[i] = A[Parent(i)];
        i = Parent(i);
    }
    A[i] = key;
}
```

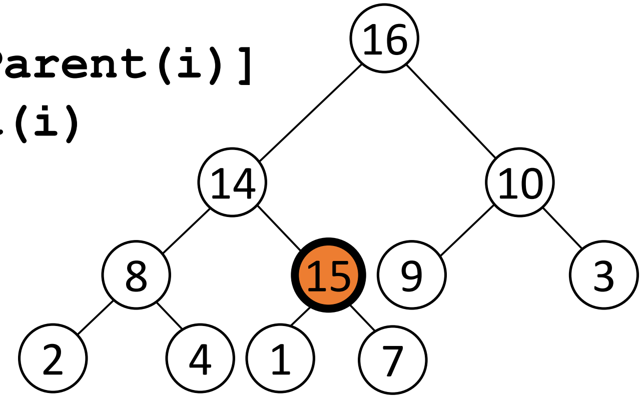
Example: Priority Queue(MAX-HEAP) INSERT

`heap_size[A]=10,i=11`

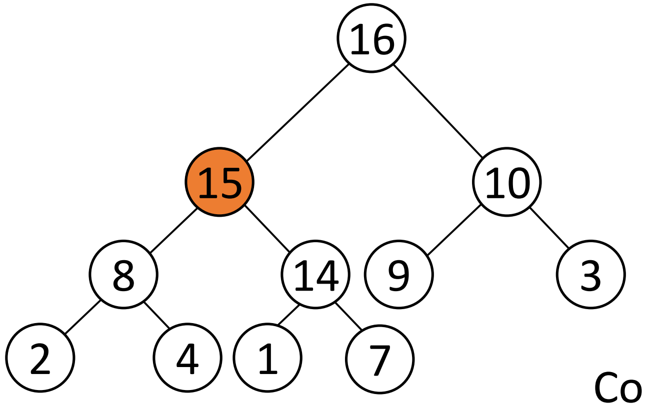
Insert 15 .Restore heap property.



`A[i] = A[Parent(i)]`
`i = Parent(i)`



The restored heap containing
the newly inserted element



Complexity: $O(\log n)$

Priority Queues in Practice

- CPU process queues
- Interrupt handling (If different interrupts have different priorities)
- Print queues
- Event-driven simulations (traffic flows)
- VLSI design (channel routing, pin layout)
- Artificial intelligence search algorithms
- Graph algorithms like Dijkstra’s shortest path algorithm,Minimum Spanning Trees…
- ……………………………..

Maxheapify C++

```
void max_heapify (int Arr[ ], int i, int N)
{   int left = 2*i                    //left child
    int right = 2*i +1           //right child
    if(left<= N and Arr[left] > Arr[i] )
         largest = left;
    else
        largest = i;
    if(right <= N and Arr[right] > Arr[largest] )
        largest = right;
    if(largest != i )
    {   swap (Ar[i] , Arr[largest]);
        max_heapify (Arr, largest,N);
    }
 }
```

Build_maxheap C++

```
void build_maxheap (int Arr[ ])
{
    for(int i = N/2 ; i >= 1 ; i-- )
    {
        max_heapify (Arr, i) ;
    }
}
```

Heapsort C++

```
void heap_sort(int Ar[ ])
{
    int heap_size = N;
    build_maxheap(Arr);
    for(int i = N; i>=2 ; i-- )
    {
        swap|(Arr[ 1 ], Arr[ i ]);
        heap_size = heap_size-1;
        max_heapify(Arr, 1, heap_size);
    }
}
```

Extract maximum : C++

```
int extract_maximum (int Arr[ ])
{
    if(length == 0)
    {
cout<< “Can’t remove element as queue is empty”;
        return -1;
    }
    int max = Arr[1];
    Arr[1] = Arr[length];
    length = length -1;
    max_heapify(Arr, length);
    return max;
}
```