BITS Pilani

Hyderabad Campus

Data Structures and Algorithms (10)

Queue

“Queue” is the fancy name for waiting in line

Problem – Queue in Hospital

Patients waiting for help in Emergency Room

Give priority to

- ✓ Severely wounded
- ✓ Bleading
- ✓ …
- ✓ the ones with crash !!!

Problem - Queue in Operating System

Processes waiting for services

Give priority to

- ✓ I/O bound
- ✓ Interrups
- ✓ Eg. small jobs(1page print) may be given priority over large jobs (100pages) …

Priority Queue

- ✓ Priority Queue is a data structure allowing at least the following two operations:
  - ➢ insert same like enqueue in nonpriority queues
  - ➢ deleteMin (/deleteMax) is the priority queue equivalent of queue's dequeu operation (i.e. find and remove the element with minimum (/maximum) priority)
- ✓ Efficient implementation of priority queue ADTs
- ✓ Uses and implementation of priority queues

Priority Queues

Queue:

- First in, first out
- First to be removed: First to enter

Priority Queue:

- First in, highest (/lowest) priority element out
- First to be removed: element with highest (/lowest) priority

- Operations: *Insert(), Remove-top()*

Applications

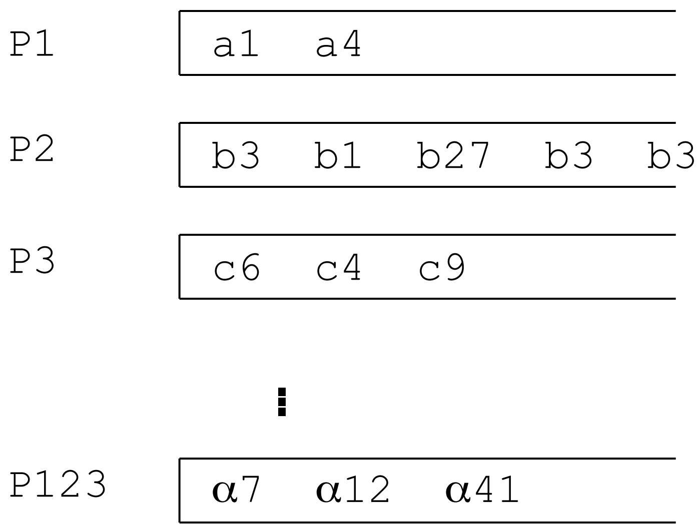
- Process scheduling
  - Give CPU resources to most urgent task
- Communications
  - Send most urgent message first
- Event-driven simulation
  - Pick next event (by time) to be simulated

Priority Queue implementations

- ✓ Number of different priority categories is **known**
- ✓ One queue for each priority

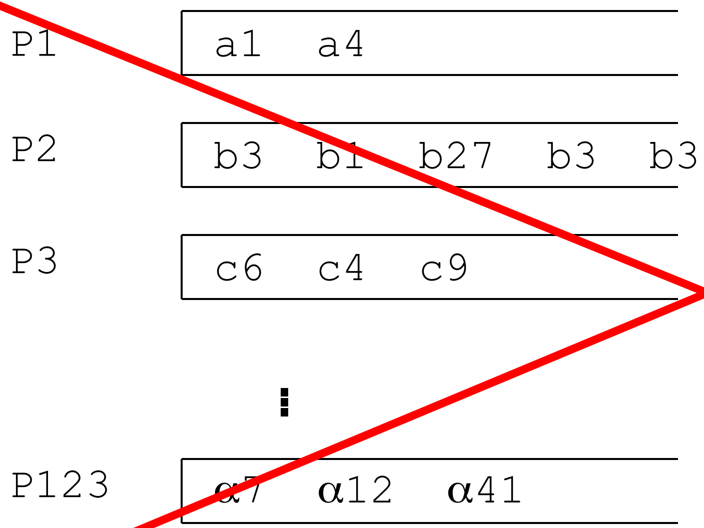
Algorithm

- ✓ **enqueue**
  - put in to proper queue
- ✓ **dequeue**
  - get from P1 first
  - then get from P2
  - then get from P3
  - …
  - then get from P123



Priority Queue implementations

- ✓ Number of different priority categories is **unknown**
- ✓ One queue for each priority



Algorithm

enqueue

- put in to proper queue

dequeue

- get from P1 first
- then get from P2
- then get from P3
- …
- then get from P123

Types of priority queues

- Ascending priority queue
  - Removal of minimum-priority element
  - *Remove-top():* Removes element with min priority
- Descending priority queue
  - Removal of maximum-priority element
  - *Remove-top():* Removes element with max priority

Generalizing queues and stacks

- ✓ Priority queues generalize normal queues and stacks
- ✓ Priority set by time of insertion
- ✓ Stack: Descending priority queue
- ✓ Queue (normal): Ascending priority queue

Priority Queue implementation(2)

Sorted linked-list, with head pointer

- Insert()
  - Search for appropriate place to insert
  - O(n)
- Remove()
  - Remove first in list
  - O(1)

Priority Queue implementation(3)

Unsorted linked-list, with head pointer

- Insert()
  - Insert at the end of linked list
  - O(1)

- Remove()
  - Search for the element (e) with min or max priority
  - Remove element (e)
  - O(n)

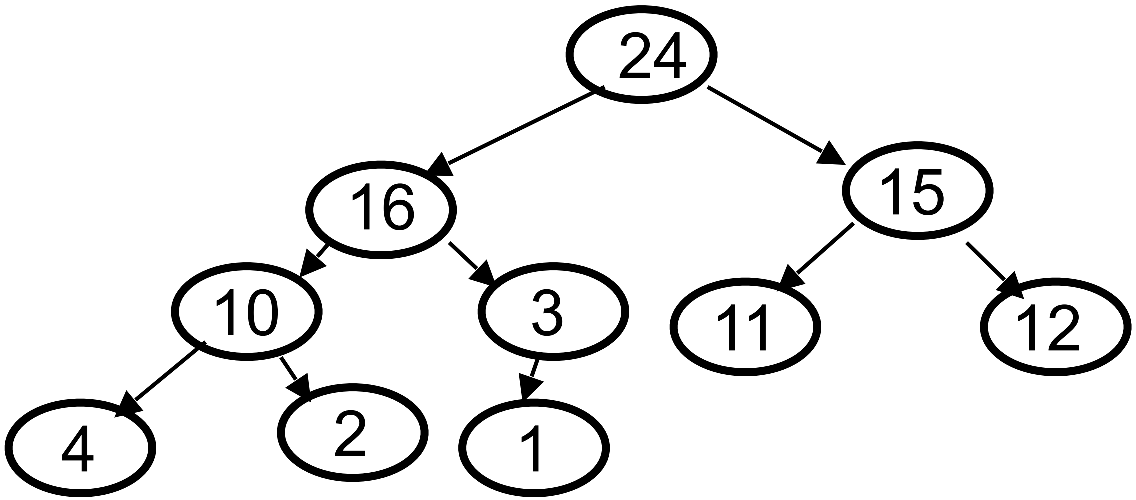
Priority Queue implementation(3)

Heap: Almost-full binary tree with heap property

- Almost full:
  - Balanced (all leaves at max height *h* or at *h-1*)
  - All leaves to the left

- Heap property: Parent >= children (descending)
  - True for all nodes in the tree
  - Note this is very different from binary search tree (BST)

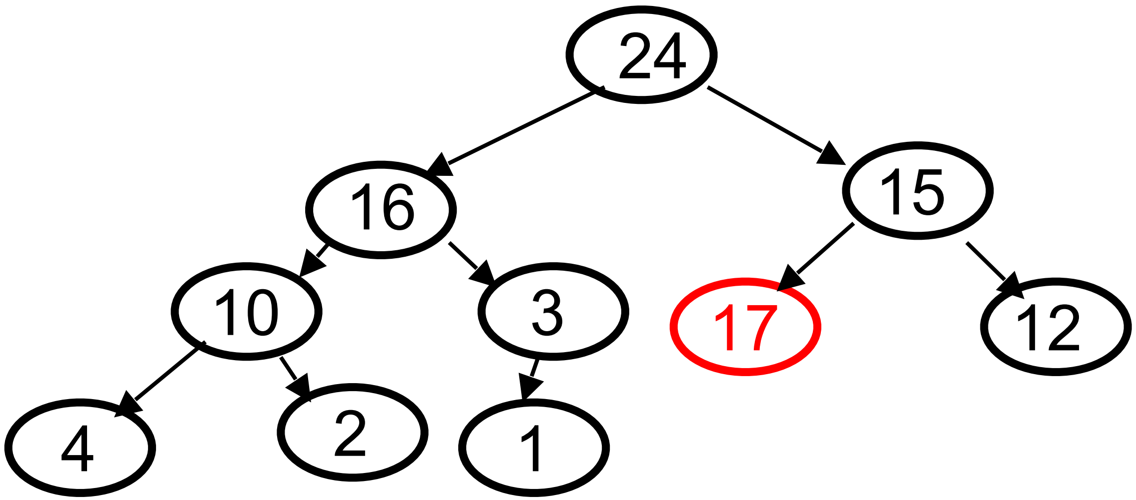
Heap Examples



Heap or not?

Heap

Heap Examples



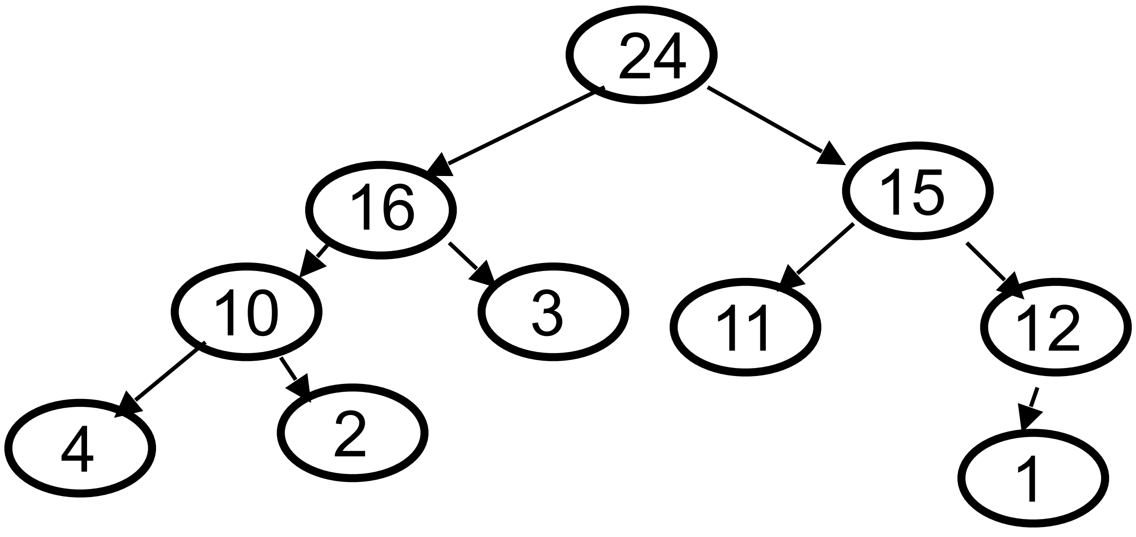
Heap or not?

Not Heap

(does not maintain heap property)

(17 > 15)

Heap Examples



Heap or not?

Not Heap

(balanced, but leaf with priority 1 is not in place)

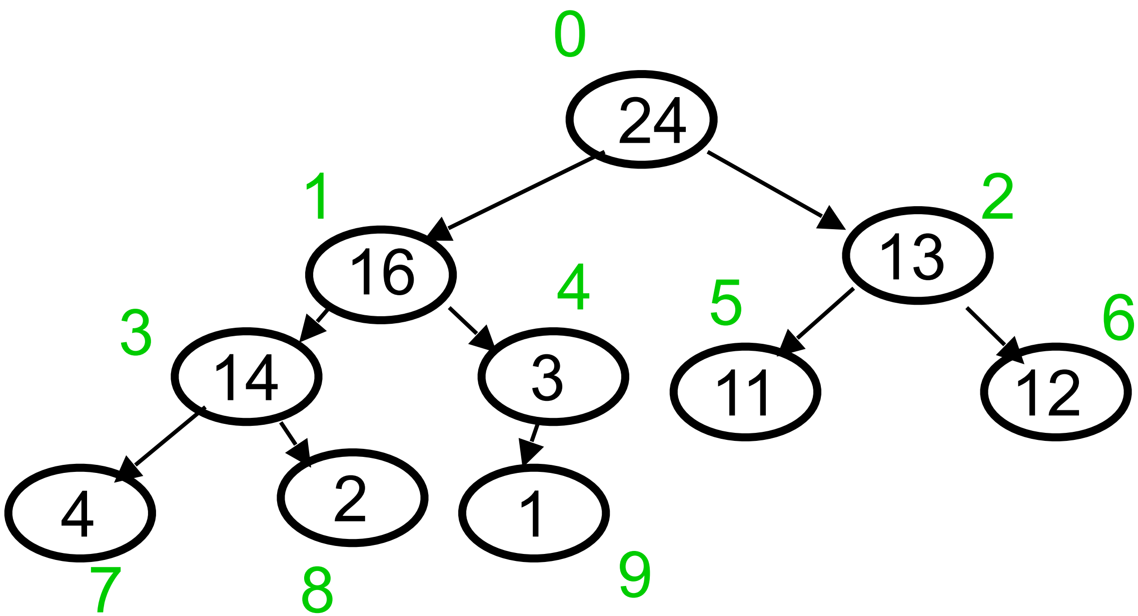
Representing heap in an array

Representing an almost-complete binary tree

- ✓ For parent in index i (i >= 0)
  - – Left child in: $i*2 + 1$
  - – Right child in: $i*2 + 2$

- ✓ From child to parent:
  - – Parent of child c in: $\lfloor (c-1)/2 \rfloor$

Example



In the array:

	24	16	13	14	3	11	12	4	2	1
Index:	0	1	2	3	4	5	6	7	8	9

Heap property

- Heap property: parent priority >= child
  - For all nodes
- Any sub-tree has the heap property
  - Thus, root contains max-priority item
- Every path from root to leaf is descending
  - This does not mean a sorted array
  - In the array:

24 16 13 14 3 11 12 4 2 1

Maintaining heap property (*heapness*)

- Remove-top():
  - ✓ Get root
  - ✓ Somehow fix heap to maintain heapness

- Insert()
  - ✓ Put item somewhere in heap
  - ✓ Somehow fix the heap to maintain heapness

Remove-top(array-heap h)

```
if h.length = 0 // num of items is 0
      return NIL
t     ← h[0]
h[0]← h[h.length-1] // last leaf
h.length ← h.length - 1
heapify_down(h, 0) // see next
return t
```

Heapify_down()

Takes an almost-correct heap, fixes it

Input: root to almost-correct heap, r

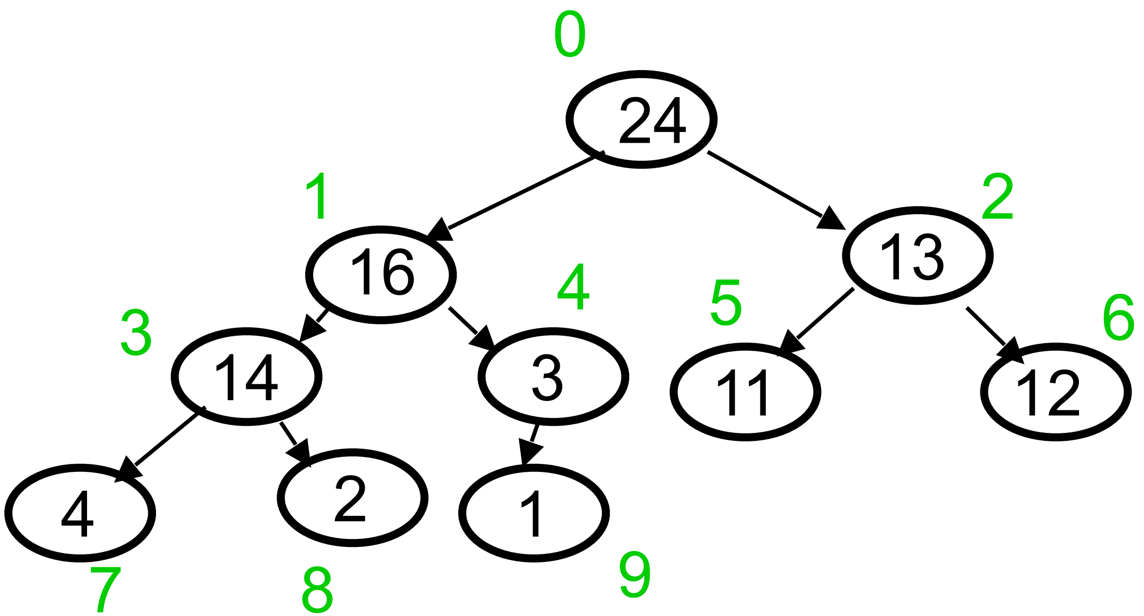
Assumes: Left subtree of r is heap

Right subtree of r is heap

but r maybe < left or right roots

- ✓ **Key operation**: interchange r with largest child.
- ✓ Repeat until in right place, or leaf.

Remove-top() example:

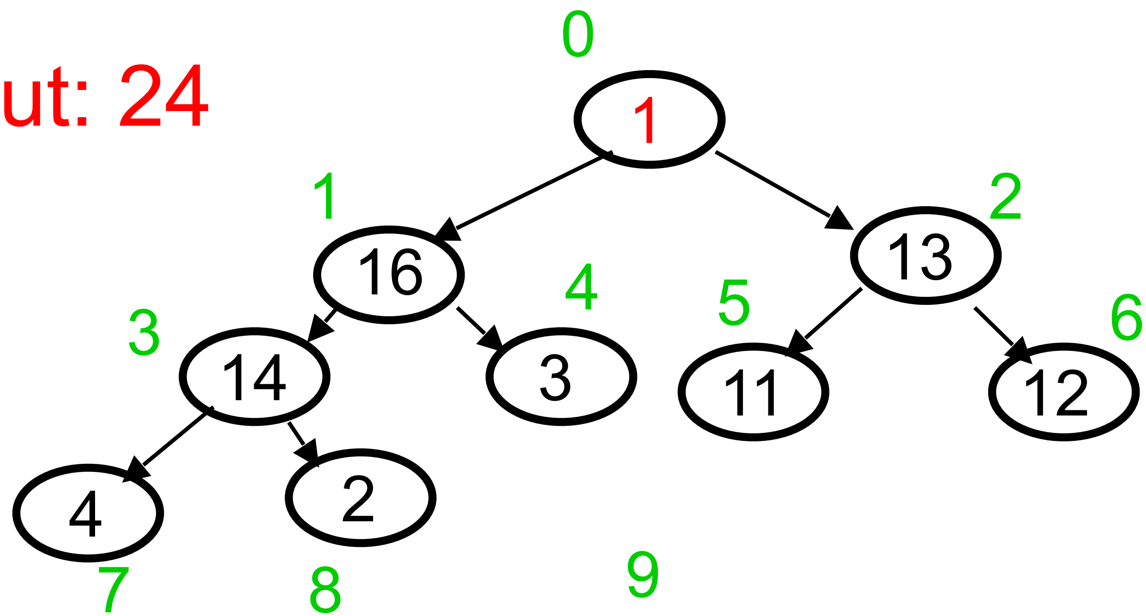


In the array:

	24	16	13	14	3	11	12	4	2	1
Index:	0	1	2	3	4	5	6	7	8	9

Remove-top() example:

Out: 24

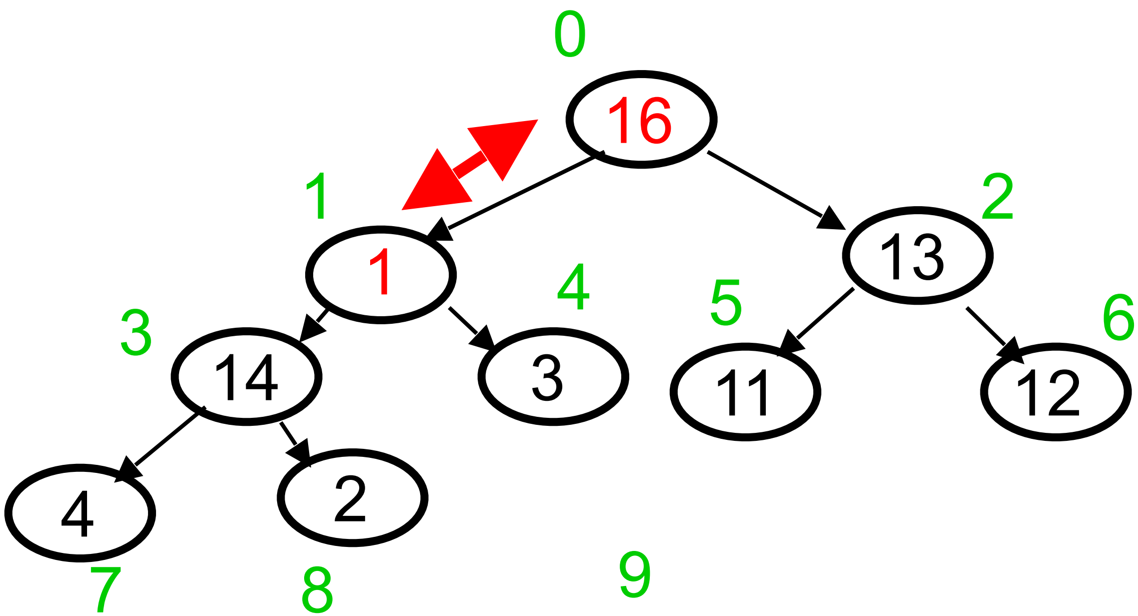


In the array:

1 16 13 14 3 11 12 4 2

Index: 0 1 2 3 4 5 6 7 8 9

Remove-top() example:

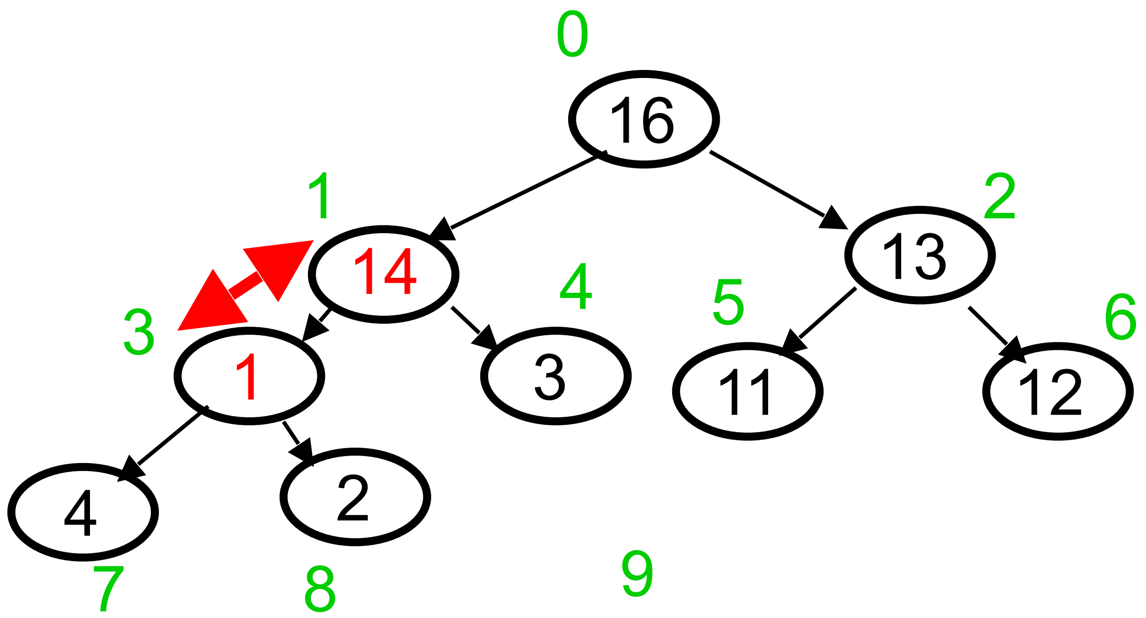


In the array:

16 1 13 14 3 11 12 4 2

Index: 0 1 2 3 4 5 6 7 8 9

Remove-top() example:

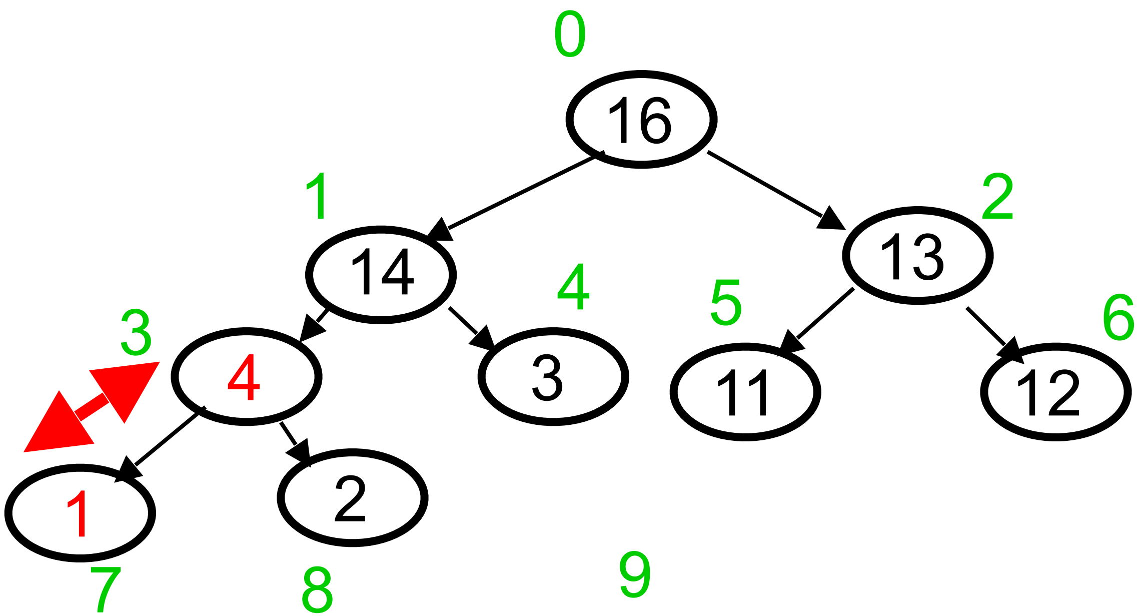


In the array:

16 14 13 1 3 11 12 4 2

Index: 0 1 2 3 4 5 6 7 8 9

Remove-top() example:



In the array:

16 14 13 4 3 11 12 1 2

Index: 0 1 2 3 4 5 6 7 8 9

Heapify_down(heap-array h, index i)

```
l ← LEFT(i)   // 2*i+1
r ← RIGHT(i) // 2*i+2
if l < h.length // left child exists
    if h[l] > h[r]
        largest ← l
    else largest ← r
if h[largest] > h[i] // child > parent
    swap(h[largest],h[i])
    Heapify_down(h,largest) // recursive
```

Remove-top() Complexity

- ✓ Removal of root – O(1)
- ✓ Heapify_down() – O(height of tree)

  O(log n)

- ➢ Remove-top() - O(log n)

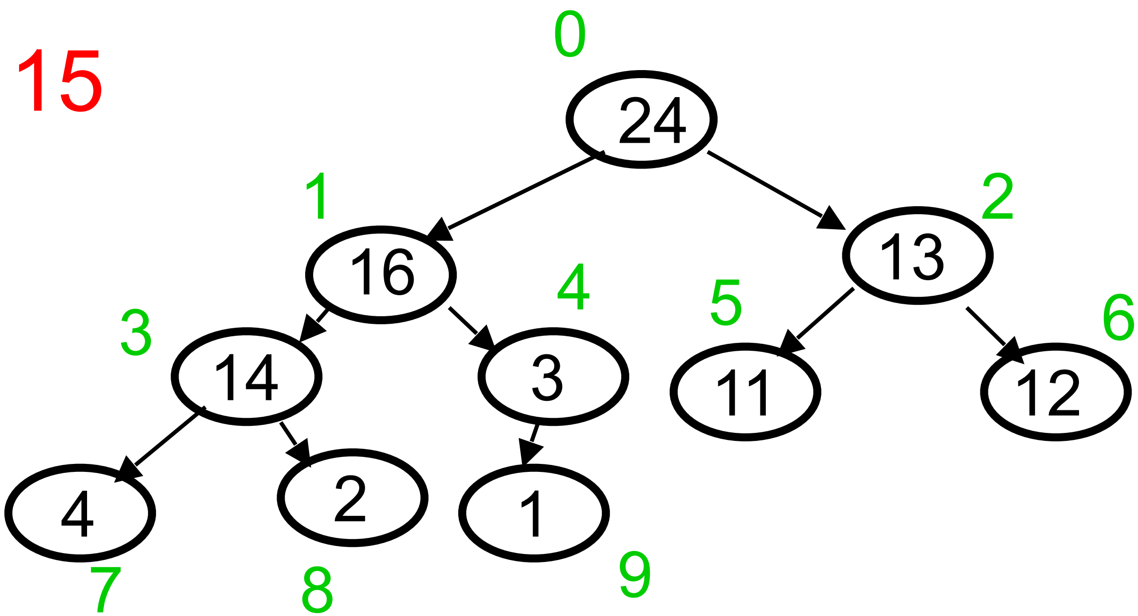
Insert(heap-array h, item t)

- ✓ Insertion works in a similar manner
- ✓ We put the new element at the end of array
- ✓ Exchange with ancestors to maintain heapness
  - ➢ If necessary.
  - ➢ Repeatedly.

```
h[h.length] ← t
h.length ← h.length + 1
Heapify_up(h, h.length) // see next
```

Insert() example:

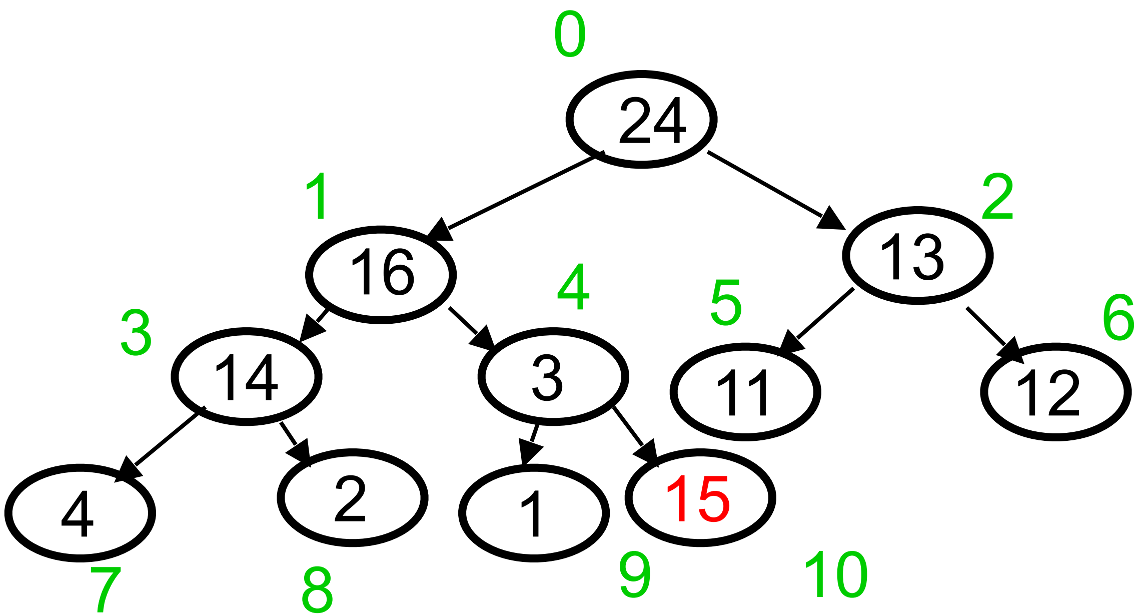
In: 15



In the array:

	24	16	13	14	3	11	12	4	2	1
Index:	0	1	2	3	4	5	6	7	8	9

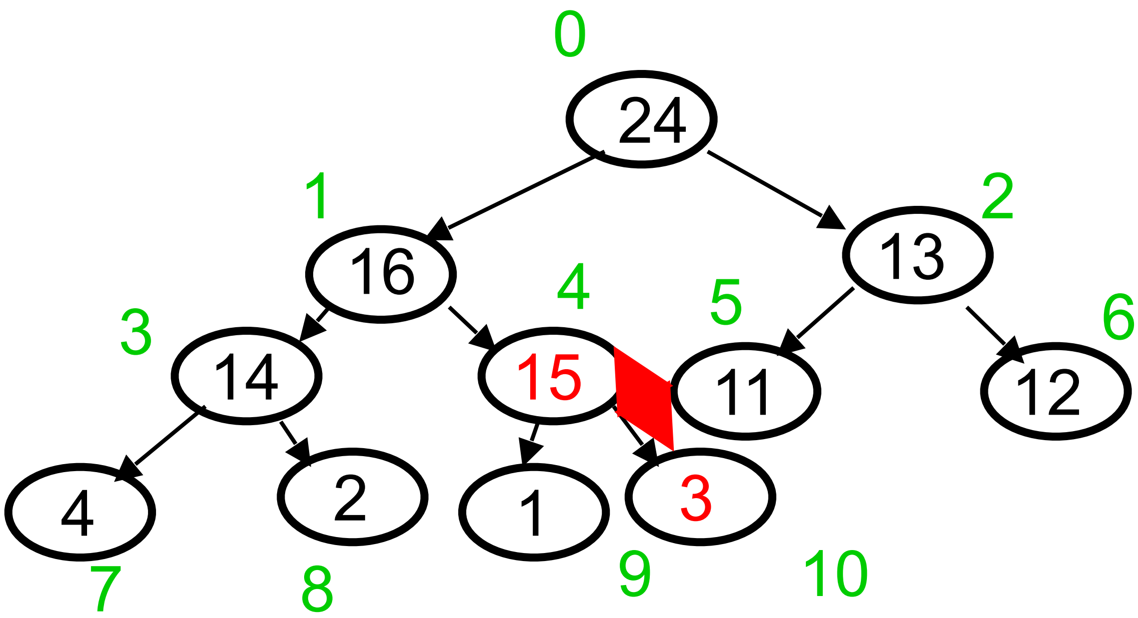
Insert() example:



In the array:

	24	16	13	14	3	11	12	4	2	1	15
Index:	0	1	2	3	4	5	6	7	8	9	10

Insert() example:



In the array:

24 16 13 14 15 11 12 4 2 1 3

Index: 0 1 2 3 4 5 6 7 8 9 10

Heapify_up(heap-array h, index i)

```
p ← PARENT(i)   // floor( (i-1)/2 )
if p < 0
    return       // we are done
if h[i] > h[p]  // child > parent
    swap(h[p],h[i])
    Heapify_up(h,p) // recursive
```

Insert() Complexity

- ✓ Insertion at end – O(1)
- ✓ Heapify_up() – O(height of tree)
  O(log n)

- ✓ Insert() - O(log n)

Priority queue as heap as binary tree in array

- ✓ Complexity is O(log n)
  - ➢ Both *insert()* and *remove-top()*
- ✓ Must pre-allocate memory for all items
- ✓ Can be used as efficient sorting algorithm
- ✓ Heapsort()

Heapsort(array a)

```
h ← new array of size a.length
for i←1 to a.length
   insert(h, a[i])        // heap insert
i ← 1
while not empty(h)
   a[i] ← remove-top(h) // heap op.
   i ← i+1
```

Complexity: O(n log n)

Building a heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(A)

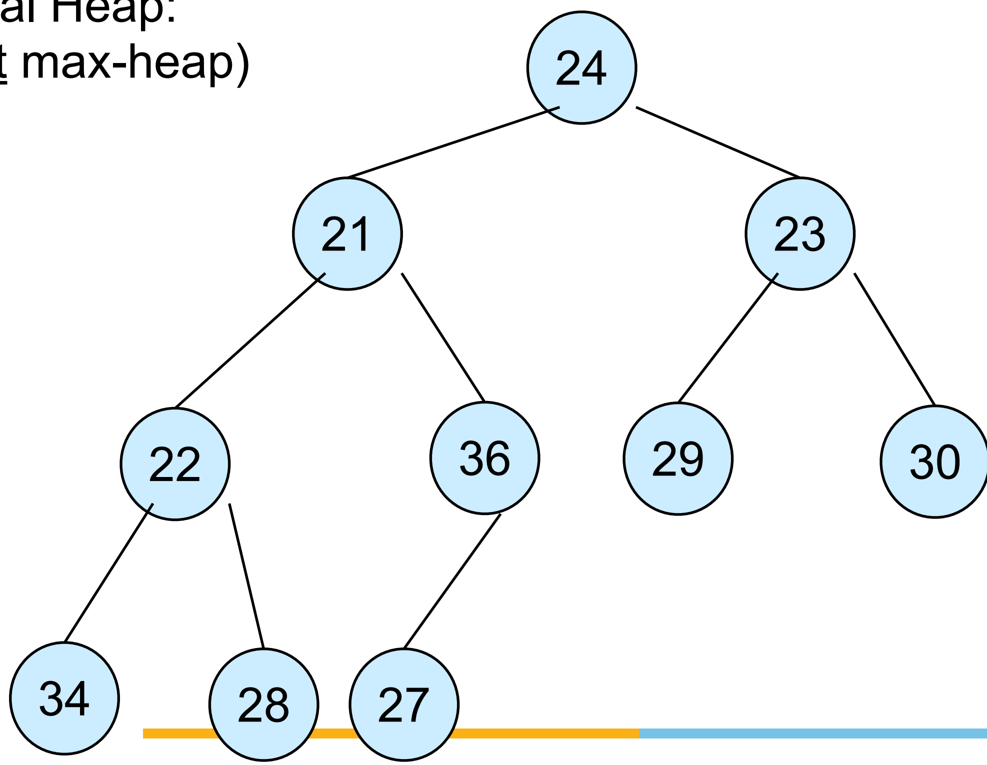
1. $heap\text{-}size[A] \leftarrow length[A]$
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3.   **do** $MaxHeapify(A, i)$

BuildMaxHeap – Example

Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap:
(not max-heap)



Data Structure Binary Heap

- Array viewed as a nearly complete binary tree.
  - Physically – linear array.
  - Logically – binary tree, filled on all levels (except lowest.)
- Map from array elements to tree nodes and vice versa
  - Root – $A[1]$
  - Left[i] – $A[2i]$
  - Right[i] – $A[2i+1]$
  - Parent[i] – $A[\lfloor i/2 \rfloor]$
- length[A] – number of elements in array A.
- heap-size[A] – number of elements in heap stored in A.
  - heap-size[A] ≤ length[A]

Heap Property (Max and Min)

- Max-Heap
  - For every node excluding the root, value is at most that of its parent: $A[parent[i]] \geq A[i]$
- Largest element is stored at the root.
- In any subtree, no values are larger than the value stored at subtree root.

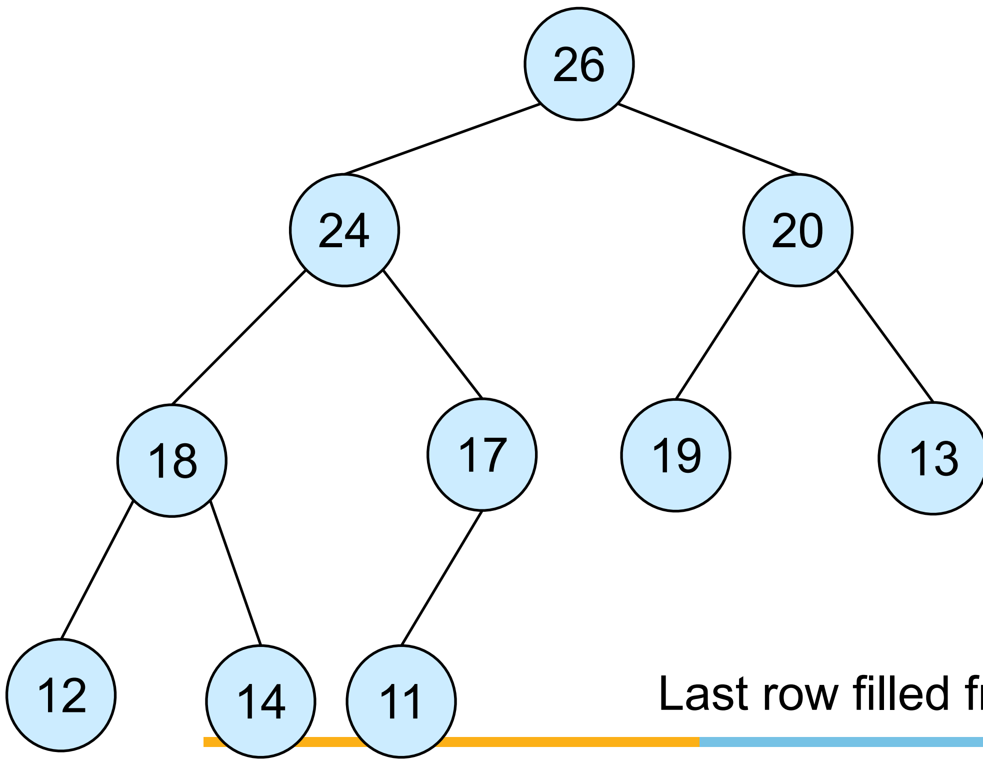
- Min-Heap
  - For every node excluding the root, value is at least that of its parent: $A[parent[i]] \leq A[i]$
- Smallest element is stored at the root.
- In any subtree, no values are smaller than the value stored at subtree root

Heaps – Example

26	24	20	18	17	19	13	12	14	11
1	2	3	4	5	6	7	8	9	10

Max-heap as an array.

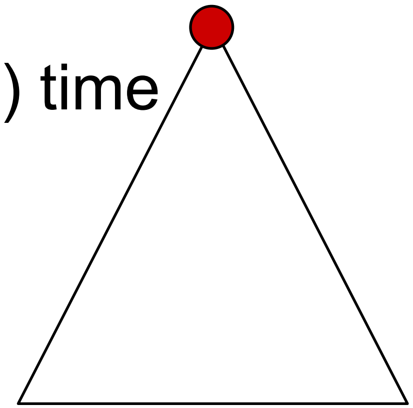
Max-heap as a binary tree.



Last row filled from left to right.

Height

- *Height of a node in a tree*: the number of edges on the longest simple downward path from the node to a leaf.
- *Height of a tree*: the height of the root.
- Height of a heap: $\lfloor \lg n \rfloor$
  - Basic operations on a heap run in $O(\lg n)$ time

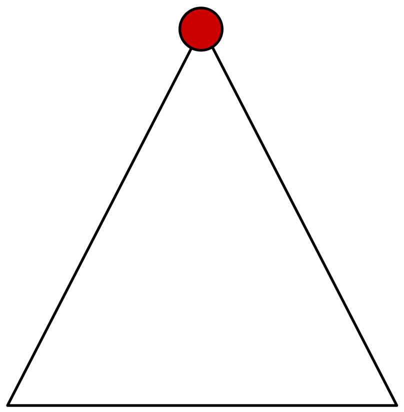


Heaps in Sorting

- Use max-heaps for sorting.
- The array representation of max-heap is not sorted.
- Steps in sorting
  - Convert the given array of size *n* to a max-heap (*BuildMaxHeap*)
  - Swap the first and last elements of the array.
    - Now, the largest element is in the last position – where it belongs.
    - That leaves *n* – 1 elements to be placed in their appropriate locations.
    - However, the array of first *n* – 1 elements is no longer a max-heap.
    - Float the element at the root down one of its subtrees so that the array remains a max-heap (MaxHeapify)
    - Repeat step 2 until the array is sorted.

Heap Characteristics

- *Height* $= \lfloor \lg n \rfloor$
- No. of *leaves* $= \lceil n/2 \rceil$
- No. of nodes of height h $\leq \lceil n/2^{h+1} \rceil$



Prove that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in an n element heap.

Proof By induction on h.

Basis: Show that it's true for $h = 0$ (i.e., that # of leaves $\leq \lceil n/2^{h+1} \rceil = \lceil n/2 \rceil$).
In fact, we'll show that the # of leaves $= \lceil n/2 \rceil$.

The tree leaves (nodes at height 0) are at depths H and $H - 1$. They consist of

- all nodes at depth H, and
- the nodes at depth $H - 1$ that are not parents of depth-H nodes.

Let x be the number of nodes at depth H—that is, the number of nodes in the bottom (possibly incomplete) level.

Note that $n - x$ is odd, because the $n - x$ nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

To prove the base case, we must consider separately the case in which n is even (x is odd) and the case in which n is odd (x is even).

Note that at any depth $d < H$ there are 2^d nodes, because all such tree levels are complete.

- If x is even, there are $x/2$ nodes at depth $H-1$ that are parents of depth H nodes, hence $2^{H-1} - x/2$ nodes at depth $H-1$ that are not parents of depth-H nodes. Thus,

$$\begin{aligned}
\text{total \# of height-0 nodes} &= x + 2^{H-1} - x/2 \\
&= 2^{H-1} + x/2 \\
&= (2^H + x)/2 \\
&= \lceil (2^H + x - 1)/2 \rceil \qquad \text{(because } x \text{ is even)} \\
&= \lceil n/2 \rceil \,.
\end{aligned}$$

($n = 2^H + x - 1$ because the complete tree down to depth $H-1$ has $2^H - 1$ nodes and depth H has x nodes.)

- If x is odd, by an argument similar to the even case, we see that

$$\begin{aligned}
\text{\# of height-0 nodes} &= x + 2^{H-1} - (x+1)/2 \\
&= 2^{H-1} + (x-1)/2 \\
&= (2^H + x - 1)/2 \\
&= n/2 \\
&= \lceil n/2 \rceil \qquad \text{(because } x \text{ odd} \Rightarrow n \text{ even)} .
\end{aligned}$$

Inductive step: Show that if it's true for height $h-1$, it's true for h.

Let n_h be the number of nodes at height h in the n-node tree T.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes.

We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - n_0 = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

Note that the nodes at height h in T would be at height $h-1$ if the leaves of the tree were removed—that is, they are at height $h-1$ in T'. Letting n'_{h-1} denote the number of nodes at height $h-1$ in T', we have

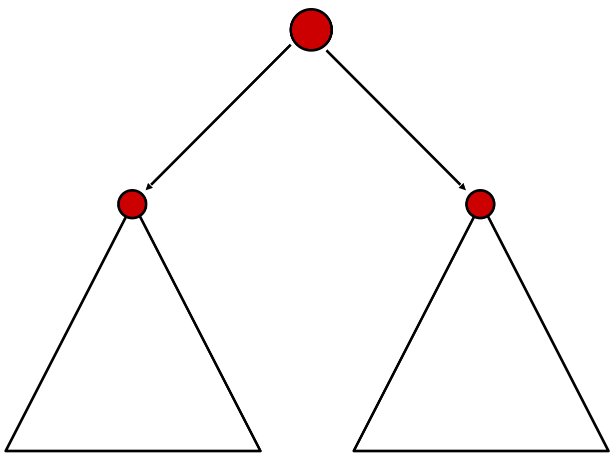
$$n_h = n'_{h-1} .$$

By induction, we can bound n'_{h-1}:

$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lfloor n/2 \rfloor / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil .$$

Maintaining the heap property

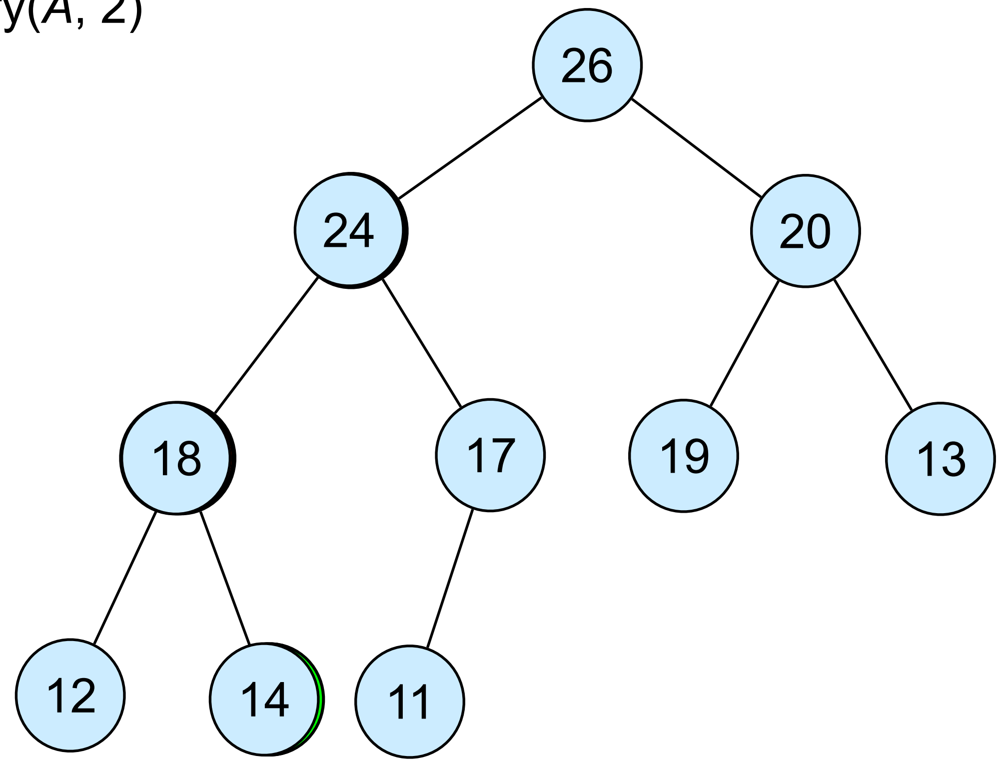
- Suppose two subtrees are max-heaps, but the root violates the max-heap property.



- Fix the offending node by exchanging the value at the node with the larger of the values at its children.
  - May lead to the subtree at the child not being a heap.
- Recursively fix the children until all of them satisfy the max-heap property.

MaxHeapify – Example

MaxHeapify(A, 2)



Procedure MaxHeapify

<u>*MaxHeapify(A, i)*</u>

1. $l \leftarrow$ left(i)
2. $r \leftarrow$ right(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. *MaxHeapify*(A, *largest*)

<u>Assumption:</u>
Left(i) and Right(i) are max-heaps.

Running Time for MaxHeapify

MaxHeapify(A, i)

1. $l \leftarrow$ left(i)
2. $r \leftarrow$ right(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. *MaxHeapify*(A, *largest*)

Time to fix node i and its children = **Θ(1)**

PLUS

Time to fix the subtree rooted at one of i's children = T(size of subree at *largest*)

Building a heap

- Use *MaxHeapify* to convert an array *A* into a max-heap.
- How?
- Call MaxHeapify on each element in a bottom-up manner.

BuildMaxHeap(A)

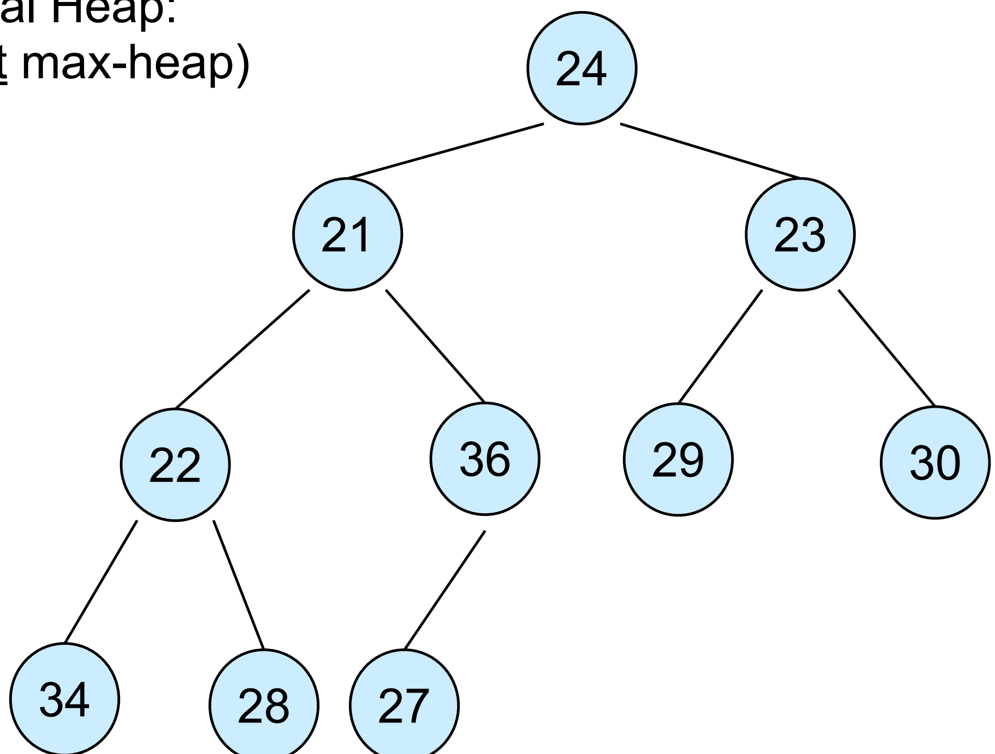
1. *heap-size*[*A*] ← *length*[*A*]
2. **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1
3. **do** *MaxHeapify*(*A*, *i*)

BuildMaxHeap – Example

Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap:
(<u>not</u> max-heap)



BuildMaxHeap – Example

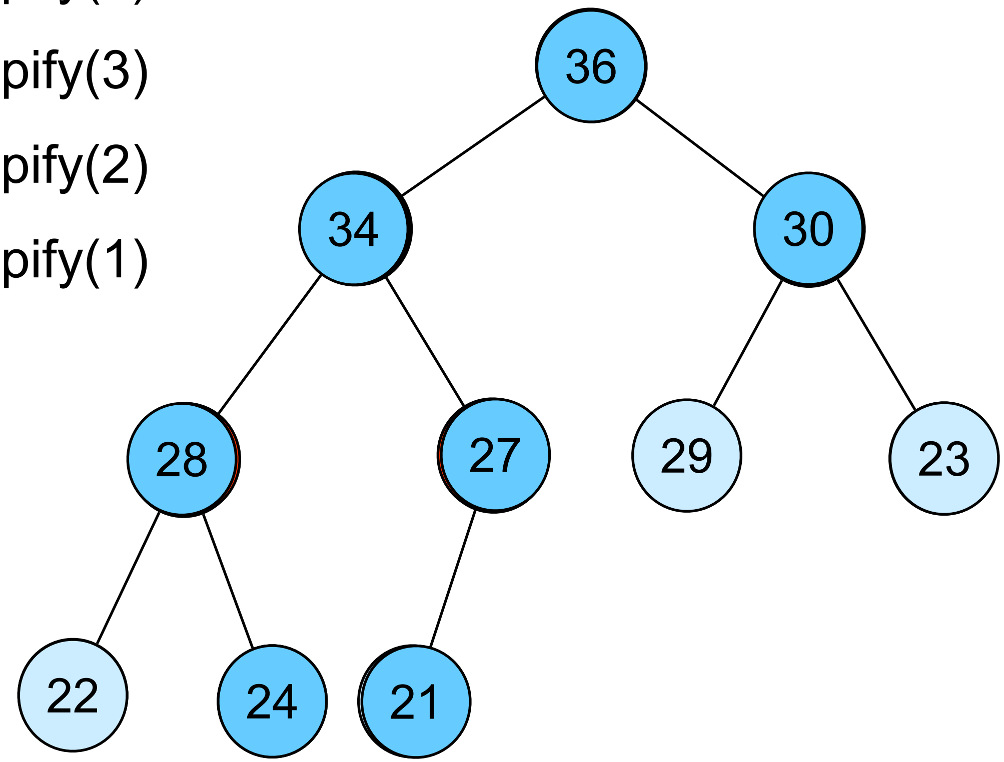
MaxHeapify($\lfloor 10/2 \rfloor = 5$)

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)



Correctness of *BuildMaxHeap*

- Loop Invariant: At the start of each iteration of the **for** loop, each node $i+1$, $i+2$, …, n is the root of a max-heap.
- Initialization:
  - Before first iteration $i = \lfloor n/2 \rfloor$
  - Nodes $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2$, …, n are leaves and hence roots of max-heaps.
- Maintenance:
  - By LI, subtrees at children of node i are max heaps.
  - Hence, MaxHeapify(i) renders node i a max heap root (while preserving the max heap root property of higher-numbered nodes).
  - Decrementing i reestablishes the loop invariant for the next iteration.

Running Time of *BuildMaxHeap*

- Loose upper bound:
  - Cost of a *MaxHeapify* call × No. of calls to *MaxHeapify*
  - $O(\lg n) \times O(n) = O(n\lg n)$
- Tighter bound:
  - Cost of a call to *MaxHeapify* at a node depends on the height, h, of the node – $O(h)$.
  - Height of most nodes smaller than n.
  - Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - No. of nodes of height h is $\lceil n/2^{h+1} \rceil$

Running Time of *BuildMaxHeap*

T(*BuildMaxHeap*) =

$$\sum_{h=0}^{\lfloor \lg n \rfloor} (NumberOfNodesAtHeight(h)) O(h)$$

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

$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$$

$$= O(n)$$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}$$

$$\le \sum_{h=0}^{\infty} \frac{h}{2^h}$$

$$= \frac{1/2}{(1-1/2)^2}$$

$$= 2$$

Can build a heap from an unordered array in linear time

Tighter Bound for *T*(*BuildMaxHeap*)

Thank You!!