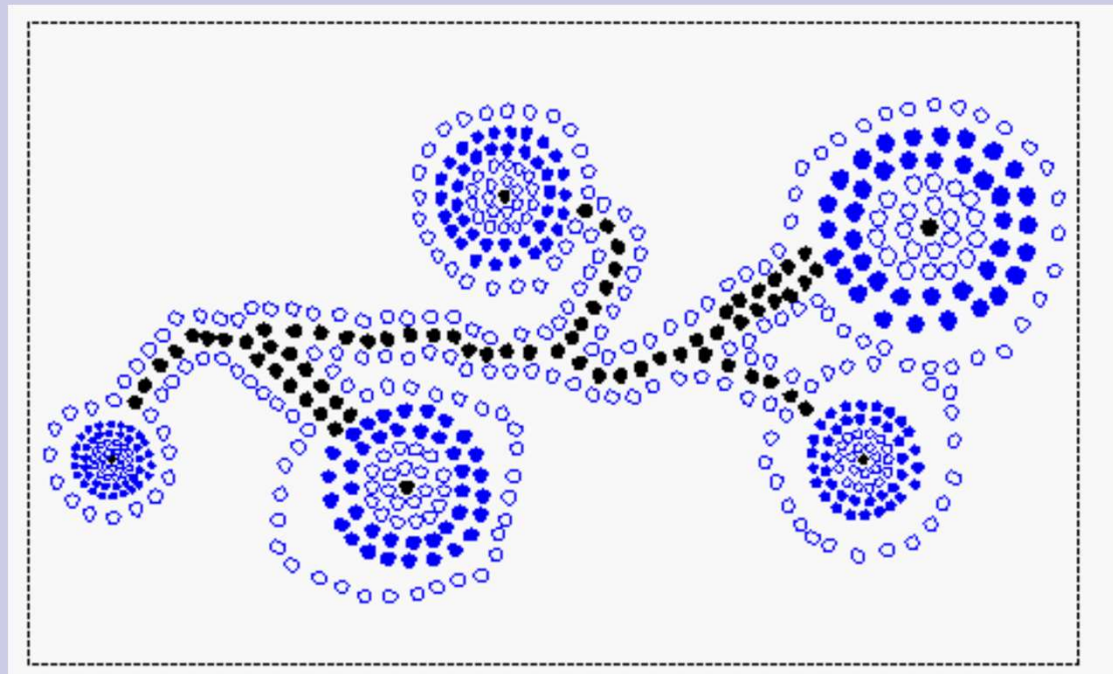


# Priority Queues

- Scheduling example
- The priority queue ADT
- Implementing a priority queue with a sequence
- Binary Heaps
- Insertion in a Heaps and Heapify



# Scheduling

- In a multi-user computer system, multiple users submit jobs to run on a single processor.
- We assume that the time required by each job is known in advance. Further, jobs can be preempted (stopped and resumed later)
- One policy which minimizes the average waiting time is SRPT (shortest remaining processing time).
- The processor schedules the job with the smallest remaining processing time.
- If while a job is running a new job arrives with processing time less than the remaining time of current job, the current job is preempted.

# Data Structure for SRPT

- We need to maintain the remaining processing time of the unfinished jobs at any point in time.
- We need to find the job with the shortest remaining processing time.
- When a job finishes we should remove it from our collection.
- When a new job arrives we need to add it to the collection.

# Priority Queues

- A priority queue is an *ADT*(*abstract data type*) for maintaining a set  $S$  of elements, each with an associated value called priority
- A PQ supports the following operations
  - **Insert**( $x$ ) insert element  $x$  in set  $S$  ( $S \leftarrow S \cup \{x\}$ )
  - **Minimum**() returns the element of  $S$  with smallest priority.
  - **Delete-min**() returns and removes the element of  $S$  with smallest priority.

# Priorities and Total Order Relations

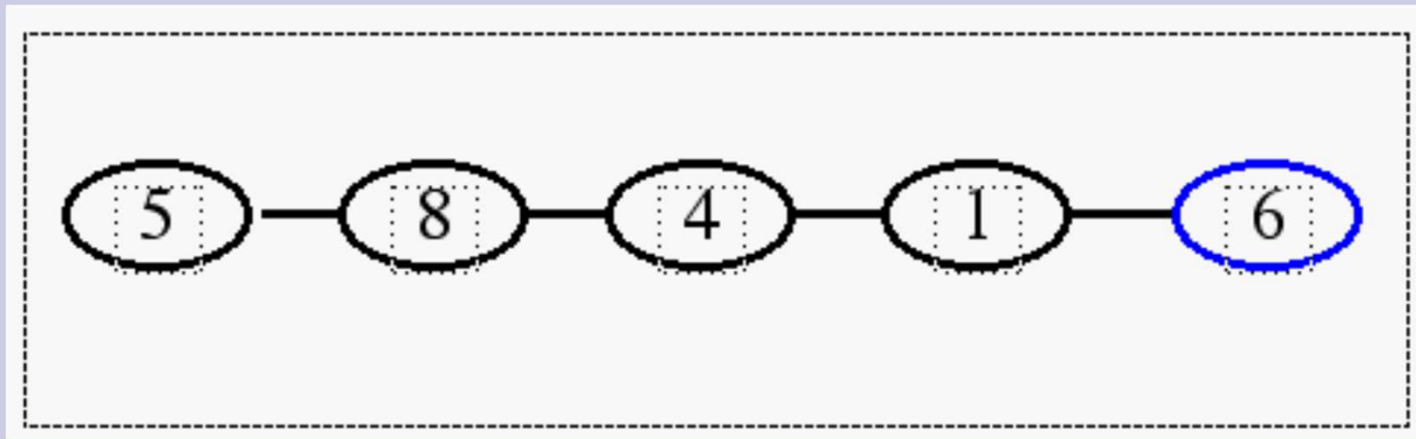
- A **Priority Queue** ranks its elements by **priority**.
- Every element has a priority. Priorities are not necessarily unique and are totally ordered.
- Total Order Relation, denoted by  $\leq$ 
  - Reflexive:**  $k \leq k$
  - Antisymmetric:** if  $k_1 \leq k_2$  and  $k_2 \leq k_1$ , then  $k_1 \leq k_2$
  - Transitive:** if  $k_1 \leq k_2$  and  $k_2 \leq k_3$ , then  $k_1 \leq k_3$

# Comparators

- The most general and reusable form of a priority queue makes use of **comparator** objects.
- Comparator objects are external to the keys that are to be compared and compare two objects.
- When the priority queue needs to compare two keys, it uses the comparator it was given to do the comparison.
- Thus a priority queue can be general enough to store any object.
- The comparator ADT includes:
  - isLessThan(a, b), isLessThanOrEqualTo(a,b),
  - isEqualTo(a, b), isGreaterThan(a,b),
  - isGreaterThanOrEqualTo(a,b), isComparable(a)

# Implem. with Unsorted Sequence

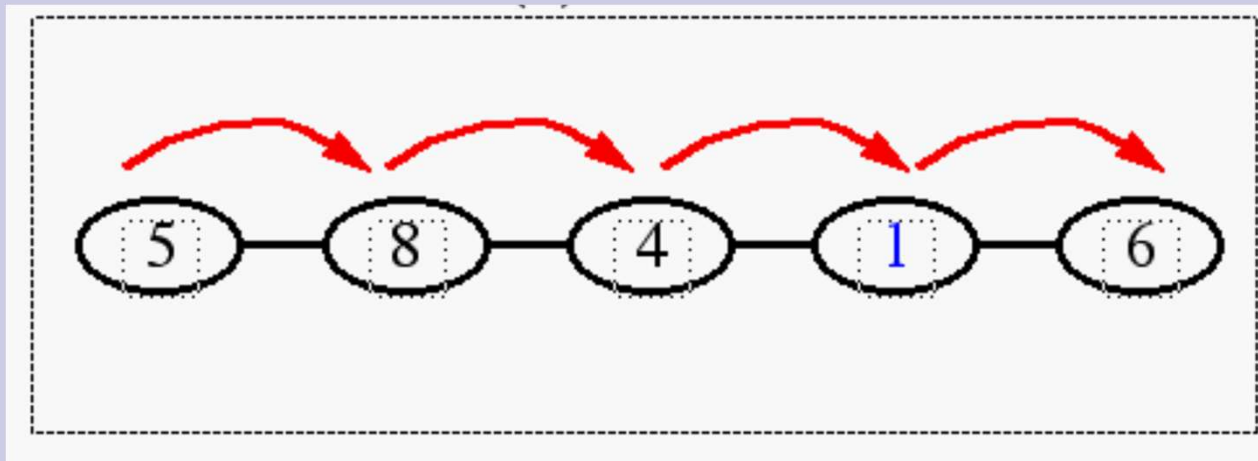
- The items are pairs (priority, element)
- We can implement `insert()` by using `insertLast()` on the sequence. This takes  $O(1)$  time.



- However, because we always insert at the end, irrespective of the key value, our sequence is not ordered.

# Unsorted Sequence (contd.)

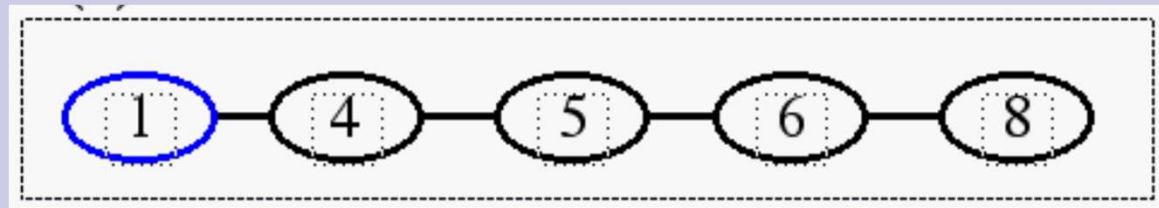
- Thus, for methods such as `minimum()`, `delete-min()` we need to **look at all the elements** of  $S$ . The worst case time complexity for these methods is  $O(n)$ .



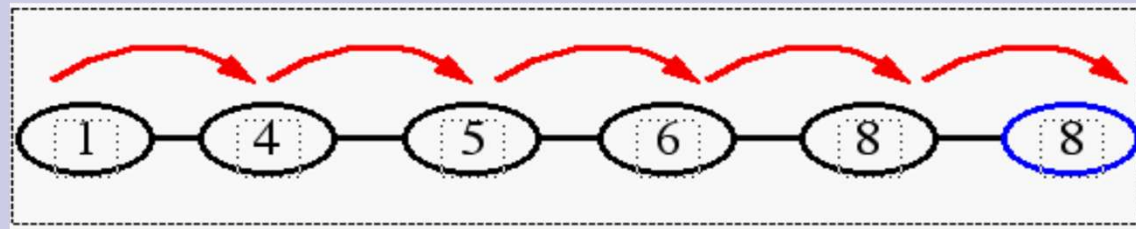


# Implem. with Sorted Sequence

- Another implementation uses a sequence  $S$ , sorted by increasing priorities.
- `minimum()` and `delete-min()` take  $O(1)$  time.



- However, to implement `insert()`, we must now scan through the entire sequence in the worst case. Thus `insert()` runs in  $O(n)$  time.



# Priority Queues

- Applications:

- job scheduling shared computing resources (Unix)
  - Event simulation
  - As a building block for other algorithms
- A Heap *can* be used to implement a PQ

# (Binary) Heaps

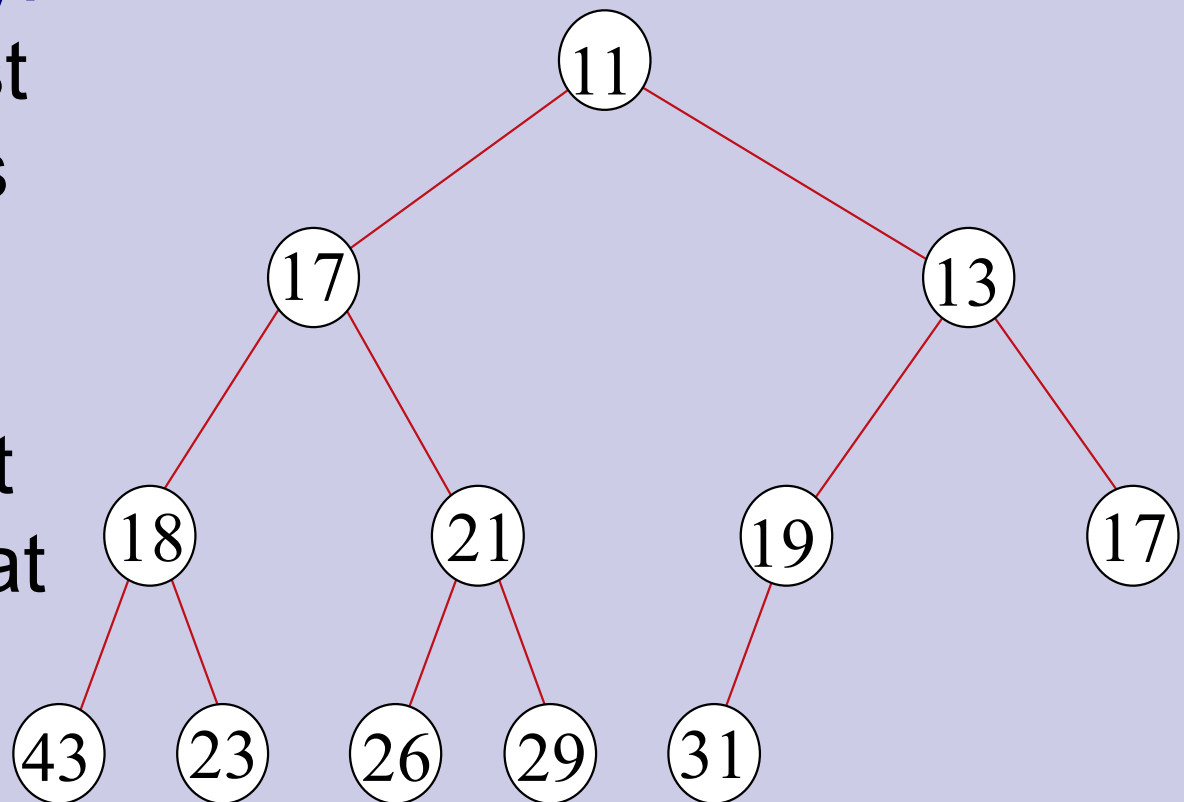
A binary tree that stores priorities (or priority-element) pairs at nodes

## Structural property:

All levels except last are full. Last level is left-filled.

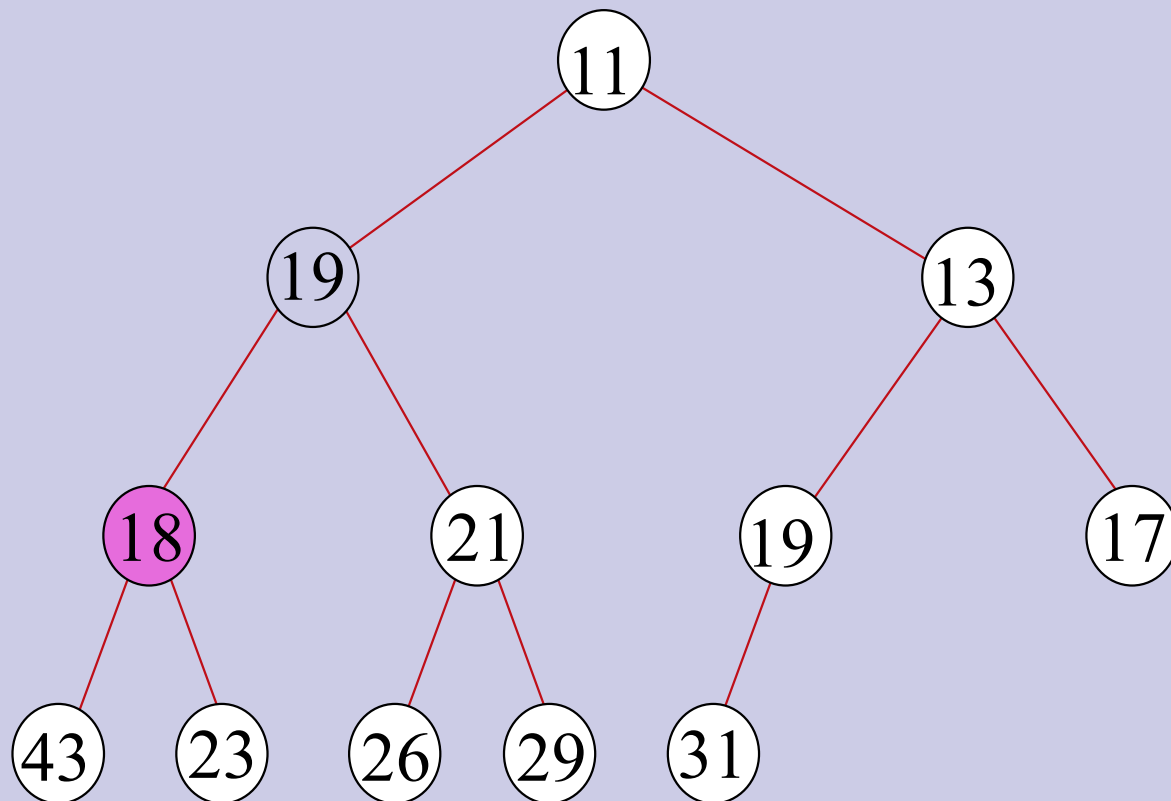
## Heap property:

Priority of node is at least as large as that of its parent.



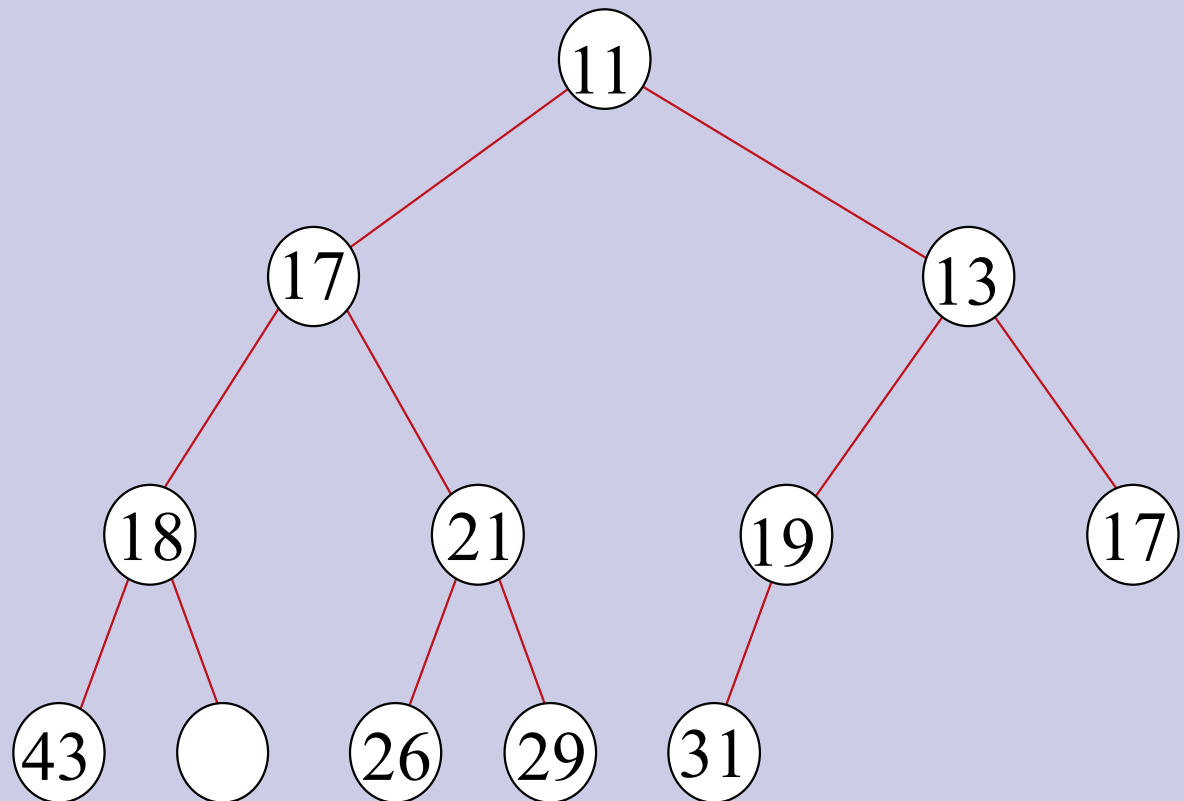
# Examples of non-Heaps

- Heap property violated



# Example of non-heap

- Last level not left-filled



# Finding the minimum element

- The element with smallest priority always sits at the root of the heap.
- This is because if it was elsewhere, it would have a parent with larger priority and this would violate the heap property.
- Hence `minimum()` can be done in  $O(1)$  time.

# Height of a heap

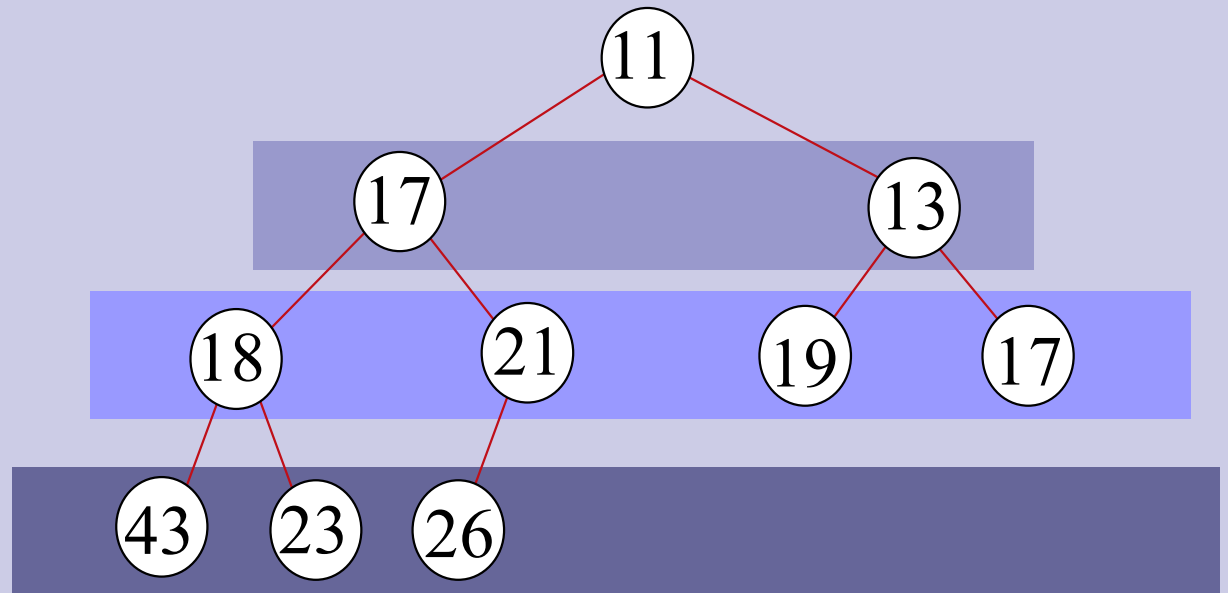
- Suppose a heap of  $n$  nodes has height  $h$ .
- Recall: complete binary tree of height  $h$  has  $2^{h+1}-1$  nodes.
- Hence  $2^h-1 < n \leq 2^{h+1}-1$ .
- $n = \lfloor \log_2 h \rfloor$

# Implementing Heaps

**Parent** ( $i$ )  
return  $\lfloor i/2 \rfloor$

**Left** ( $i$ )  
return  $2i$

**Right** ( $i$ )  
return  $2i+1$



	1	2	3	4	5	6	7	8	9	10
A	11	17	13	18	21	19	17	43	23	26
Level:	0	1		2				3		

**Heap property:**  $A[\text{Parent}(i)] \leq A[i]$



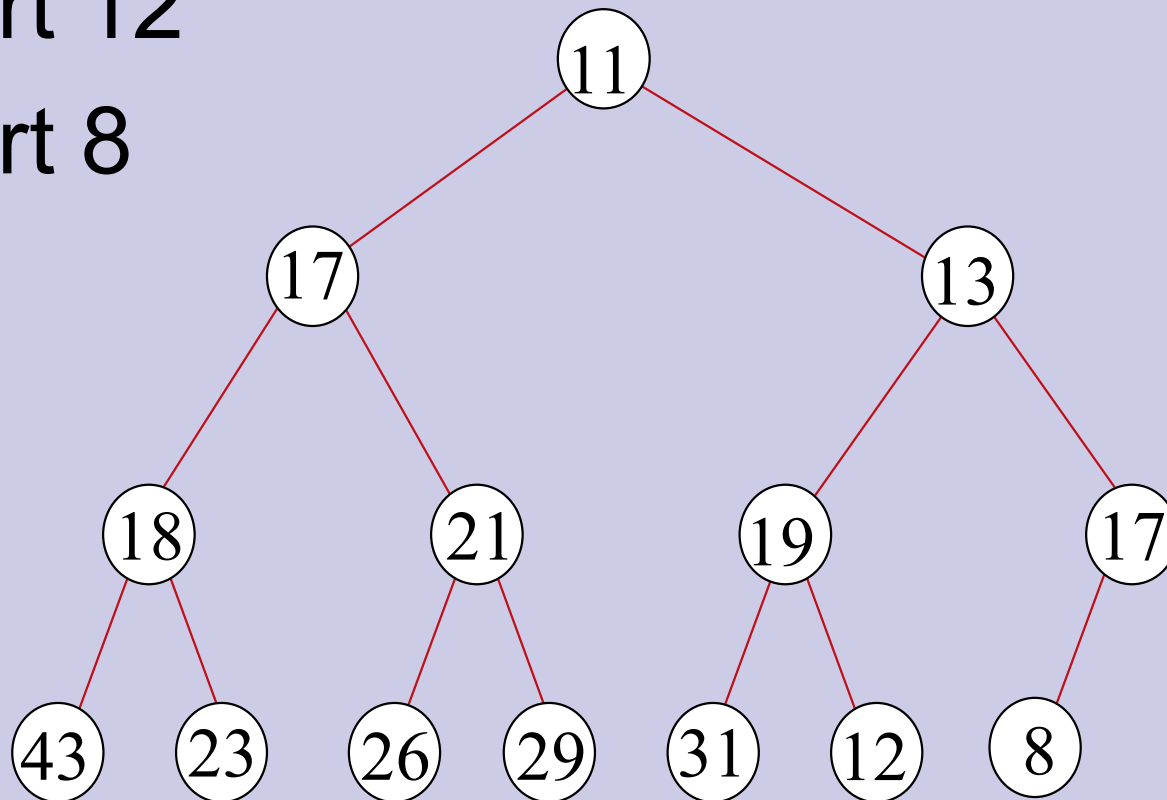
# Implementing Heaps (2)

- Notice the implicit tree links; children of node  $i$  are  $2i$  and  $2i+1$
- Why is this useful?
  - In a binary representation, a multiplication/division by two is left/right shift
  - Adding 1 can be done by adding the lowest bit

# Insertion in a Heap

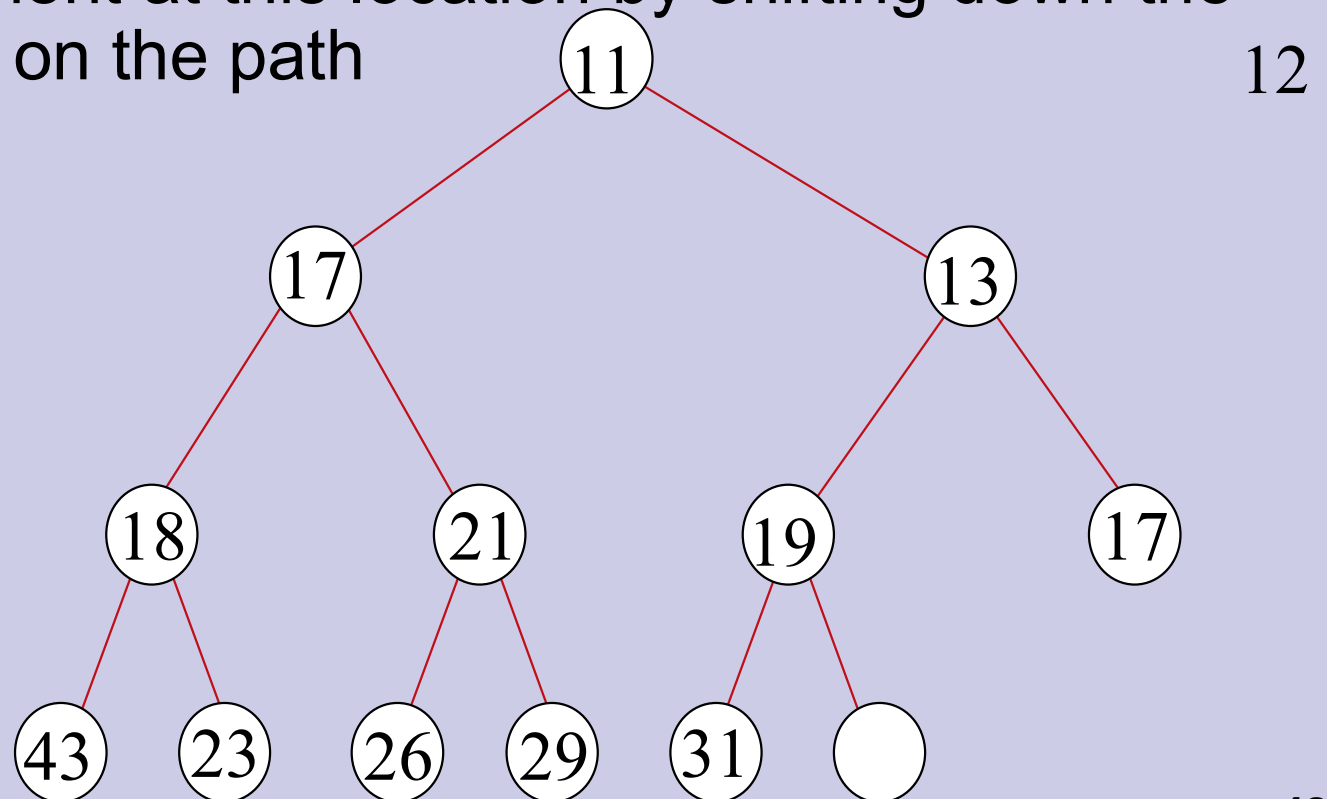
□ Insert 12

□ Insert 8



# Another View of Insertion

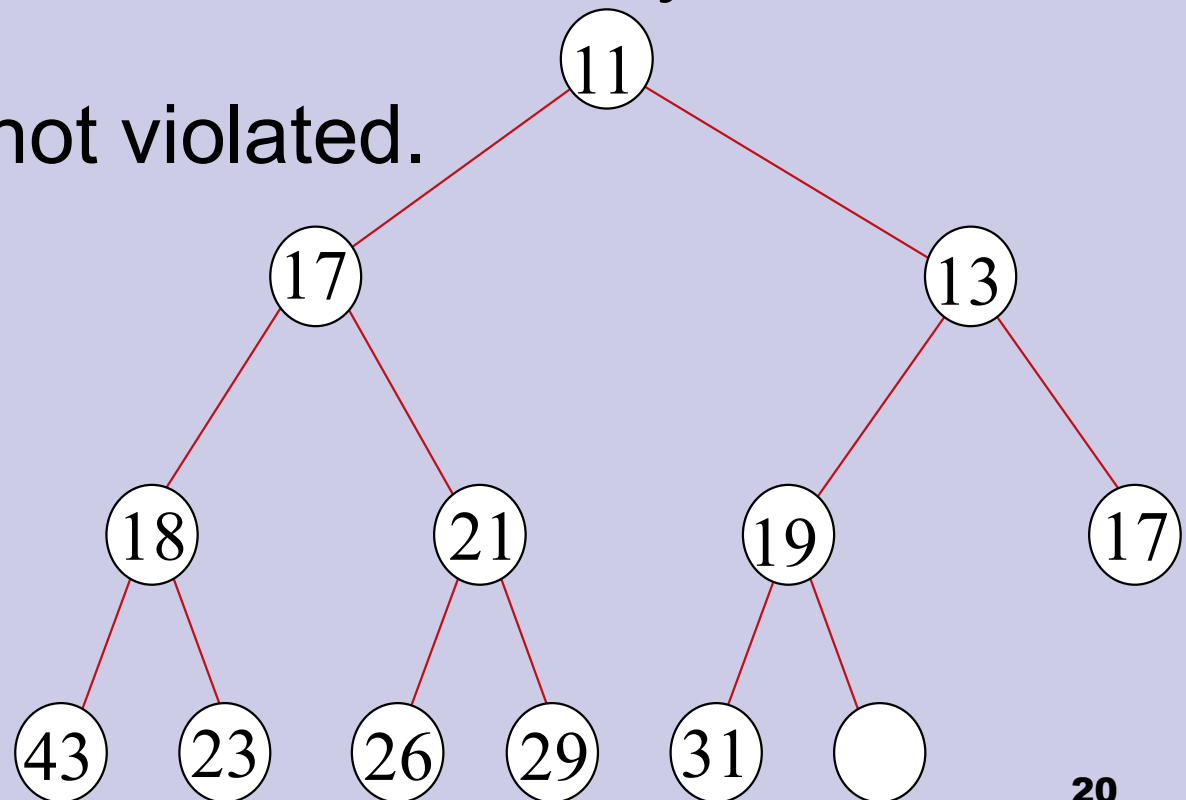
- Enlarge heap
- Consider path from root to inserted node
- Find topmost element on this path with higher priority than that of inserted element.
- Insert new element at this location by shifting down the other elements on the path



# Correctness of Insertion

- The only nodes whose contents change are the ones on the path.
- Heap property may be violated only for children of these nodes.
- But new contents of these nodes only have lower priority.
- So heap property not violated.

12

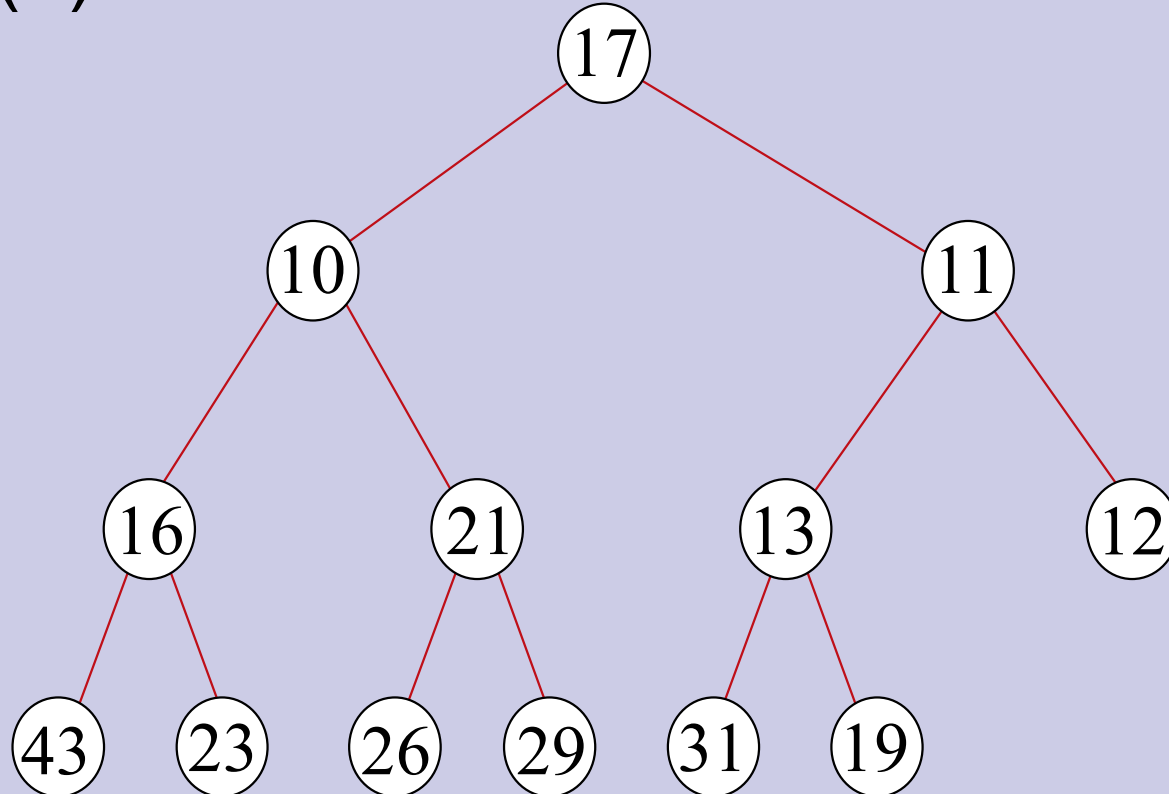


# Heapify

- $i$  is index into the array  $A$
- Binary trees rooted at  $\text{Left}(i)$  and  $\text{Right}(i)$  are heaps
- But,  $A[i]$  might be smaller than its children, thus violating the heap property
- The method **Heapify** makes binary tree rooted at  $i$  a heap by moving  $A[i]$  down the heap.

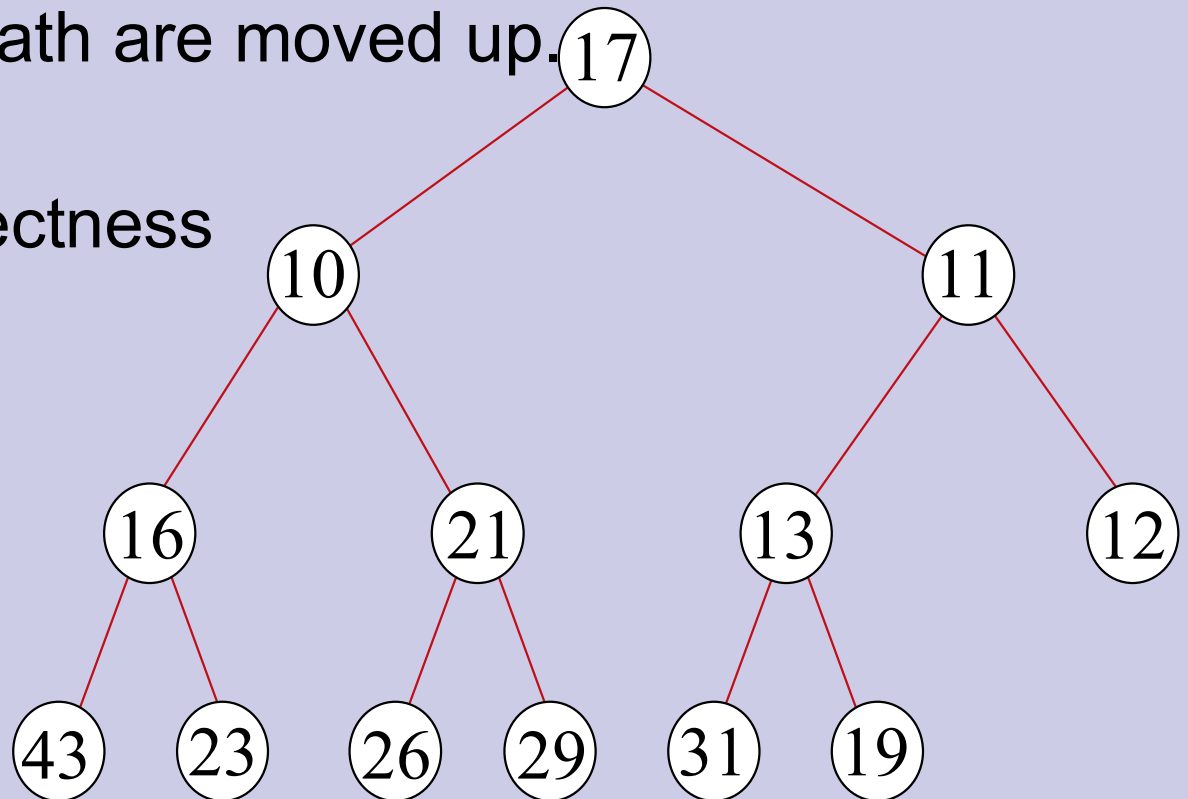
# Heapify

- Heap property violated at node with index 1 but subtrees rooted at 2, 3 are heaps.
- `heapify(1)`



# Another View of Heapify

- Heapify(i) traces a path down the tree.
- Last node on path (say j) has both  $A[\text{left}(j)]$ ,  $A[\text{right}(j)]$  are larger than  $A[i]$
- All elements on path have lower priority than their siblings.
- All elements on this path are moved up.
- $A[i]$  goes to location j.
- This establishes correctness



# Running time Analysis

- A heap of  $n$  nodes has height  $O(\log n)$ .
- While **inserting** we might have to move the element all the way to the top.
- Hence at most  $O(\log n)$  steps required.
- In **Heapify**, the element might be moved all the way to the last level.
- Hence **Heapify** also requires  $O(\log n)$  time.