More Sorting Algorithms

Priority Queues, Heaps, and Heapsort

Priority Queue: Motivating Example

3 jobs have been submitted to a printer in the order A, B, C.

Consider the printing pool at this moment.

Sizes: Job A — 100 pages

Job B — 10 pages

Job C — 1 page

Average finish time with FIFO service:

(100+110+111) / 3 = 107 time units

Average finish time for shortest-job-first service:

(1+11+111) / 3 = 41 time units

FIFO = First In First Out

Priority Queue: Motivating Example

- The elements in the queue are printing jobs, each with the associated number of pages that serves as its priority
- Processing the shortest job first corresponds to extracting the smallest element from the queue
- Insert new printing jobs as they arrive

Want a queue capable of supporting two operations:

Insert and **Extract-Min**.

Priority Queue

A *Priority Queue* is an abstract data structure that supports two operations

- Insert: inserts the new element into the queue
- Extract-Min: removes and returns the smallest element from the queue

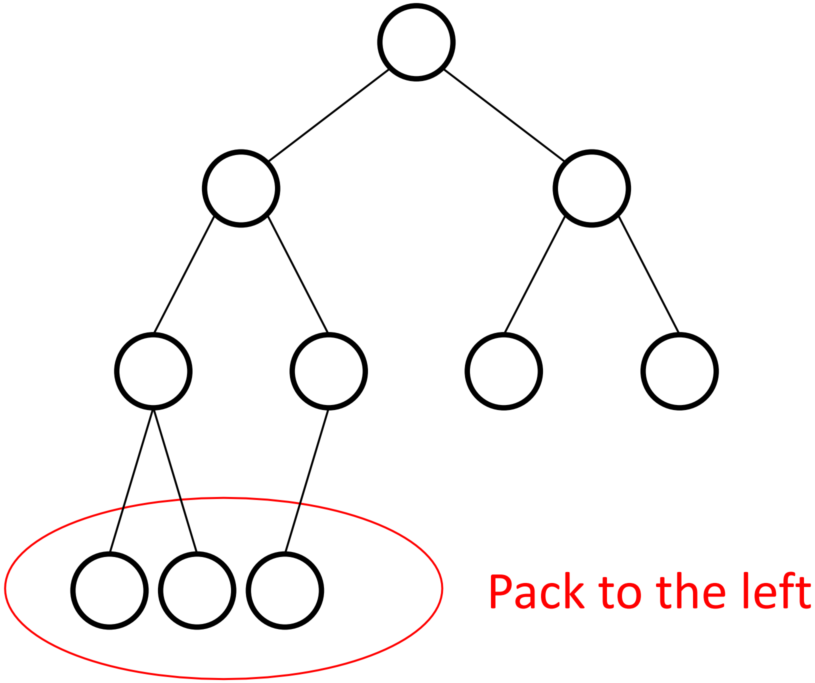
Possible Implementations

- unsorted list + a pointer to the smallest element
  - **Insert in $O(1)$ time**
  - **Extract-Min in $O(n)$ time**, since it requires a linear scan to find the new minimum
- sorted doubly linked list + a pointer to first element
  - **Insert in $O(n)$ time** (to insert at proper location)
  - **Extract-Min in $O(1)$ time**

Question

Is there any data structure that supports both these priority queue operations faster at the same time?
In $O(\log n)$ time for each?

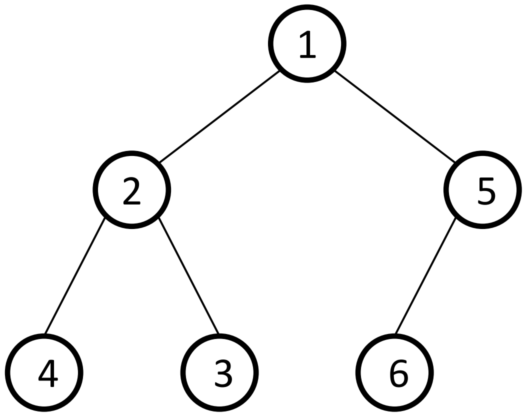
(Binary) Heap



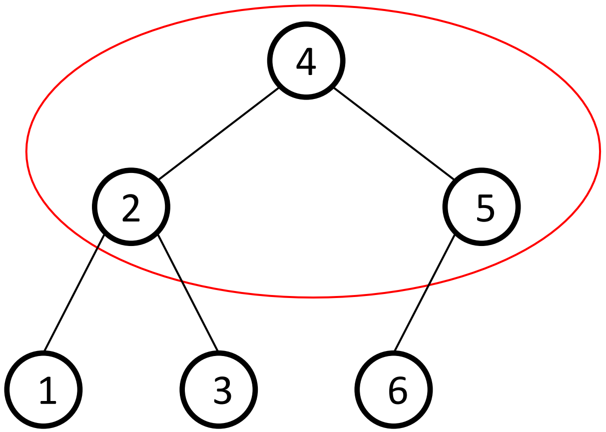
Heaps are “almost complete binary trees”

- All levels are full except possibly the lowest level
- If the lowest level is not full, then nodes must be packed to the left

Heap-order Property



A min-heap



Not a heap

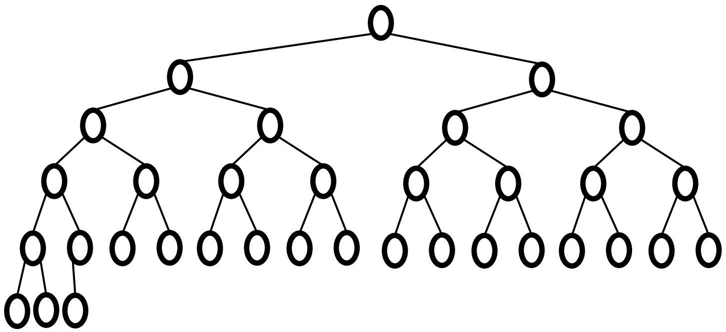
Heap-order property:

The value of a node is at least the value of its parent — Min-heap

Heap Properties

- If the heap-order property is maintained, we will show that heaps support the following operations efficiently (n is # elements in the heap):
  - **Insert in $O(\log n)$ time**
  - **Extract-Min in $O(\log n)$ time**

Heap Properties



Level $i =$	Nodes on level i	Nodes on or above level i
0	1	1
1	2	3
2	4	7
3	8	15
4	16	31
$h = 5$	3	34

A heap with height (deepest level) h has 2^i nodes on every level $i < h$.

=> A heap with height h has $2^h - 1$ nodes above level h.

=> A heap with height h has between 2^h and $2^{h+1} - 1$ nodes.

Consider a heap with an n elements and height h:

=> $2^h \leq n < 2^{h+1}$ => $h \leq \log_2 n < h + 1.$

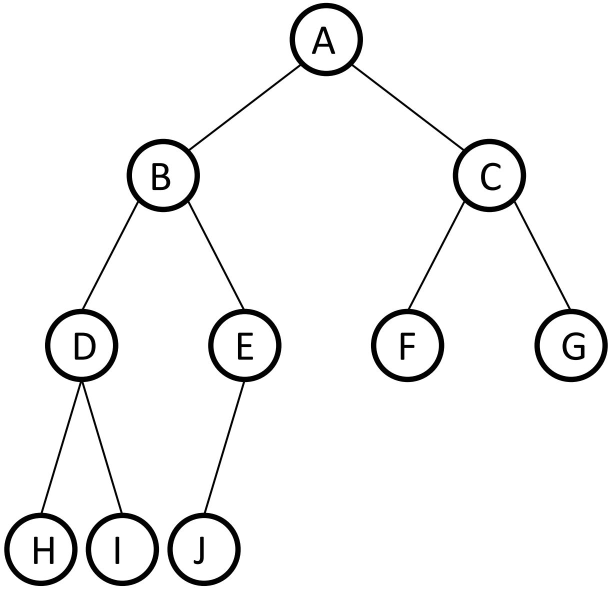
=> an n-element heap has height $\Theta(\log n)$.

Heap Properties

- If the heap-order property is maintained, we will show that heaps support the following operations efficiently
(n is # elements in the heap):
  - **Insert in $O(\log n)$ time**
  - **Extract-Min in $O(\log n)$ time**

- Structural properties
  - an n-element heap has height $\Theta(\log n)$.
  - Also, the structure is so regular, it can be represented in an array with no pointers required!!!

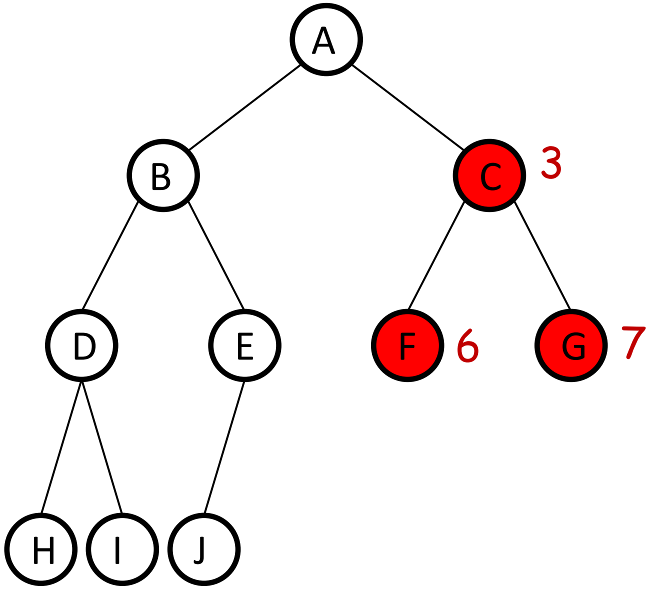
Array Implementation of Heap



1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

- The root is in array position 1
- For any element in array position i
  - The left child is in position $2i$
  - The right child is in position $2i + 1$
  - The parent is in position $\lfloor i/2 \rfloor$

Array Implementation of Heap



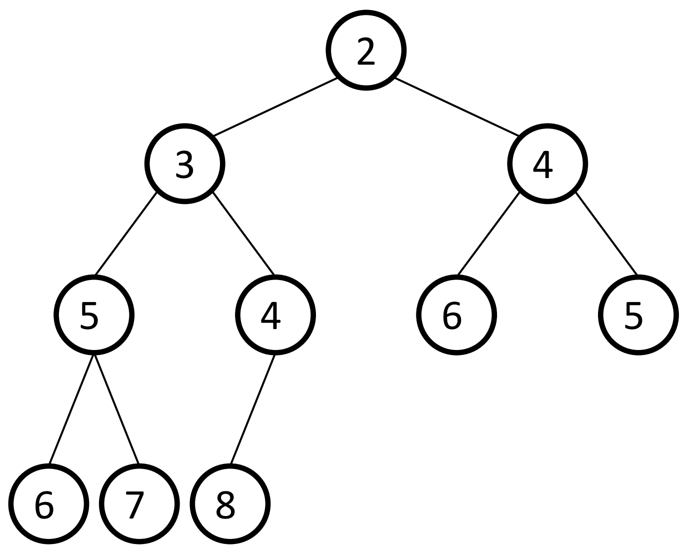
1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

Example: $\mathrm{C} = A[3]$.
C's children are
$F = A[2 \cdot 3] = A[6]$ and $G = A[2 \cdot 3 + 1] = A[7]$.
G's parent is $\mathrm{C} = A[3] = A[\lfloor 7/2 \rfloor]$.

- The root is in array position 1
- For any element in array position i
  - The left child is in position $2i$
  - The right child is in position $2i + 1$
  - The parent is in position $\lfloor i/2 \rfloor$
- We will draw the heaps as trees, with the understanding that an actual implementation will use simple arrays

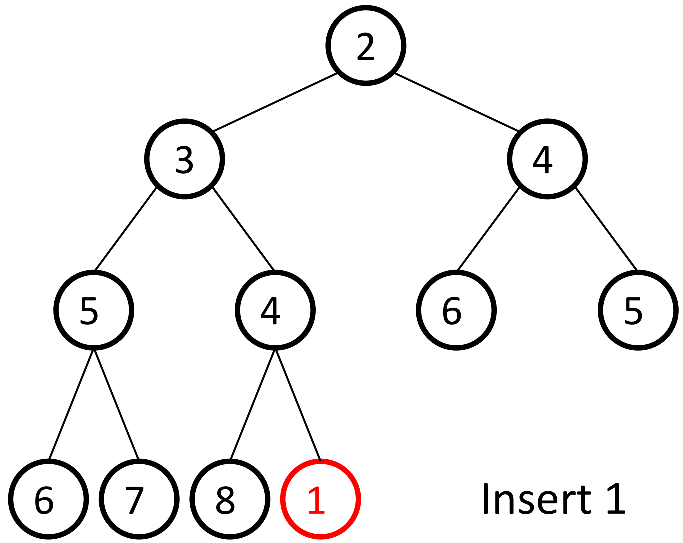
Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



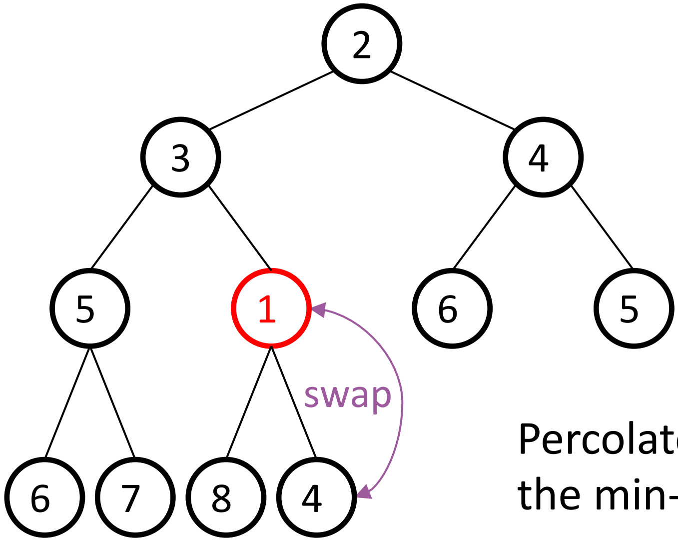
Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Insertion

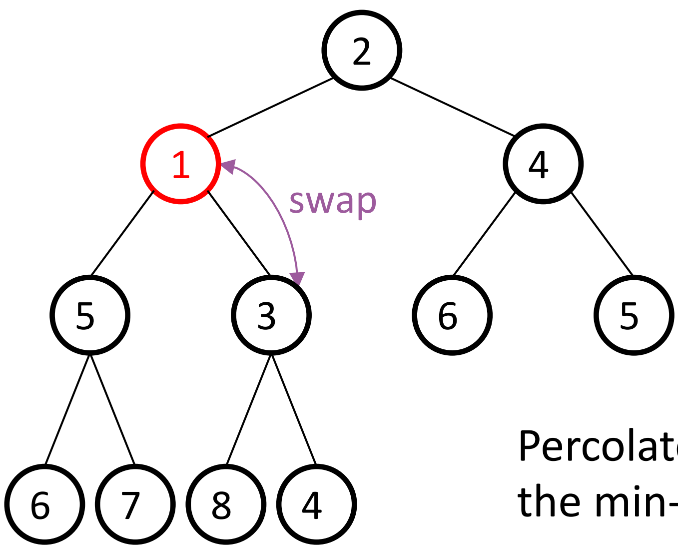
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Percolate up to maintain the min-heap property

Insertion

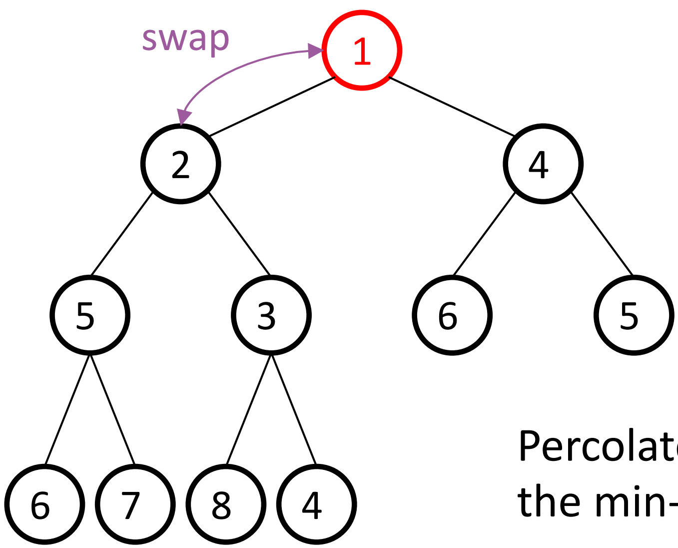
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Percolate up to maintain the min-heap property

Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.

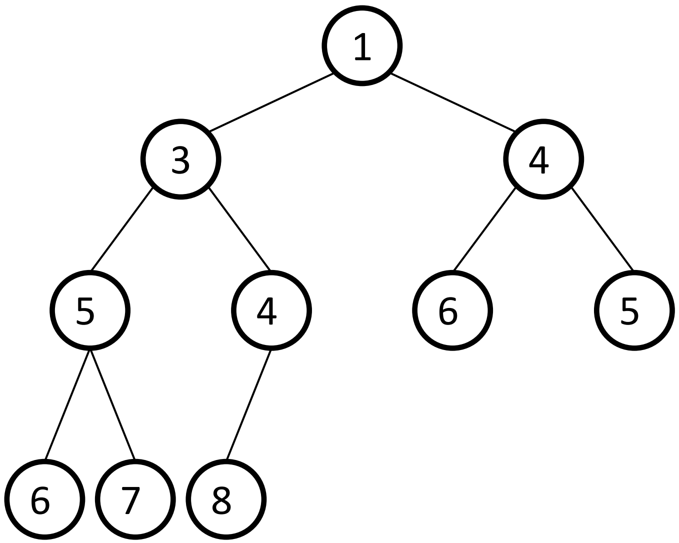


Percolate up to maintain the min-heap property

- Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element
- Time complexity $= O(\text{height}) = O(\log n)$

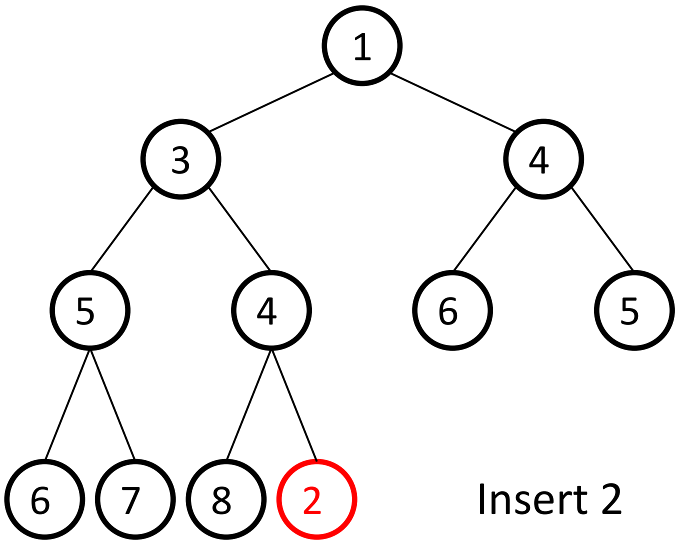
Insertion (2nd Example)

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



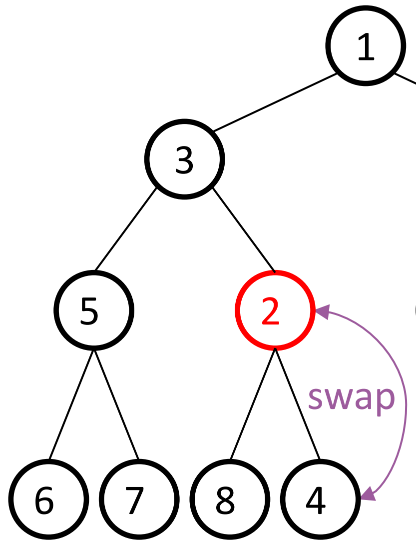
Insertion (2nd Example)

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Insertion (2nd Example)

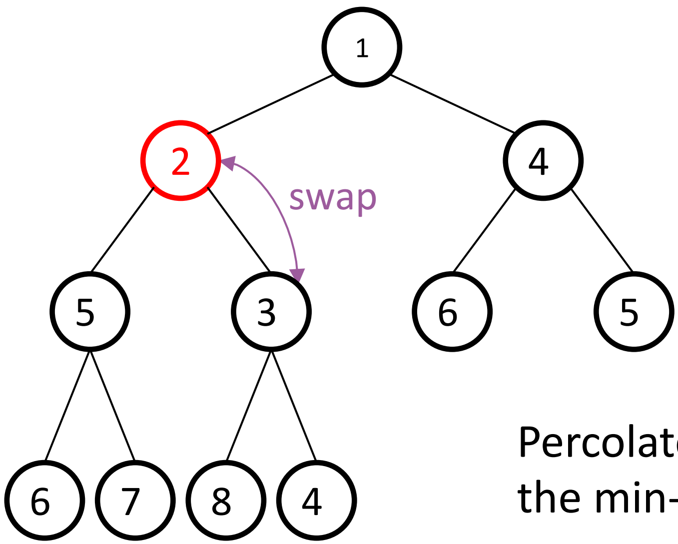
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Percolate up to maintain the min-heap property

Insertion (2nd Example)

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Percolate up to maintain the min-heap property

In this example, swapping stopped BEFORE reaching the top.

Insertion

Insert(x, i): Add item x to heap $A[1 \cdots i-1]$ creating heap $A[1 \cdots i]$

begin

$A[i] := x;$

$j = i;$

while $j > 1$ **and** $A[j] < A\left[\left\lfloor \frac{j}{2} \right\rfloor\right]$ **do**

// $A[j]$ is less than its parent

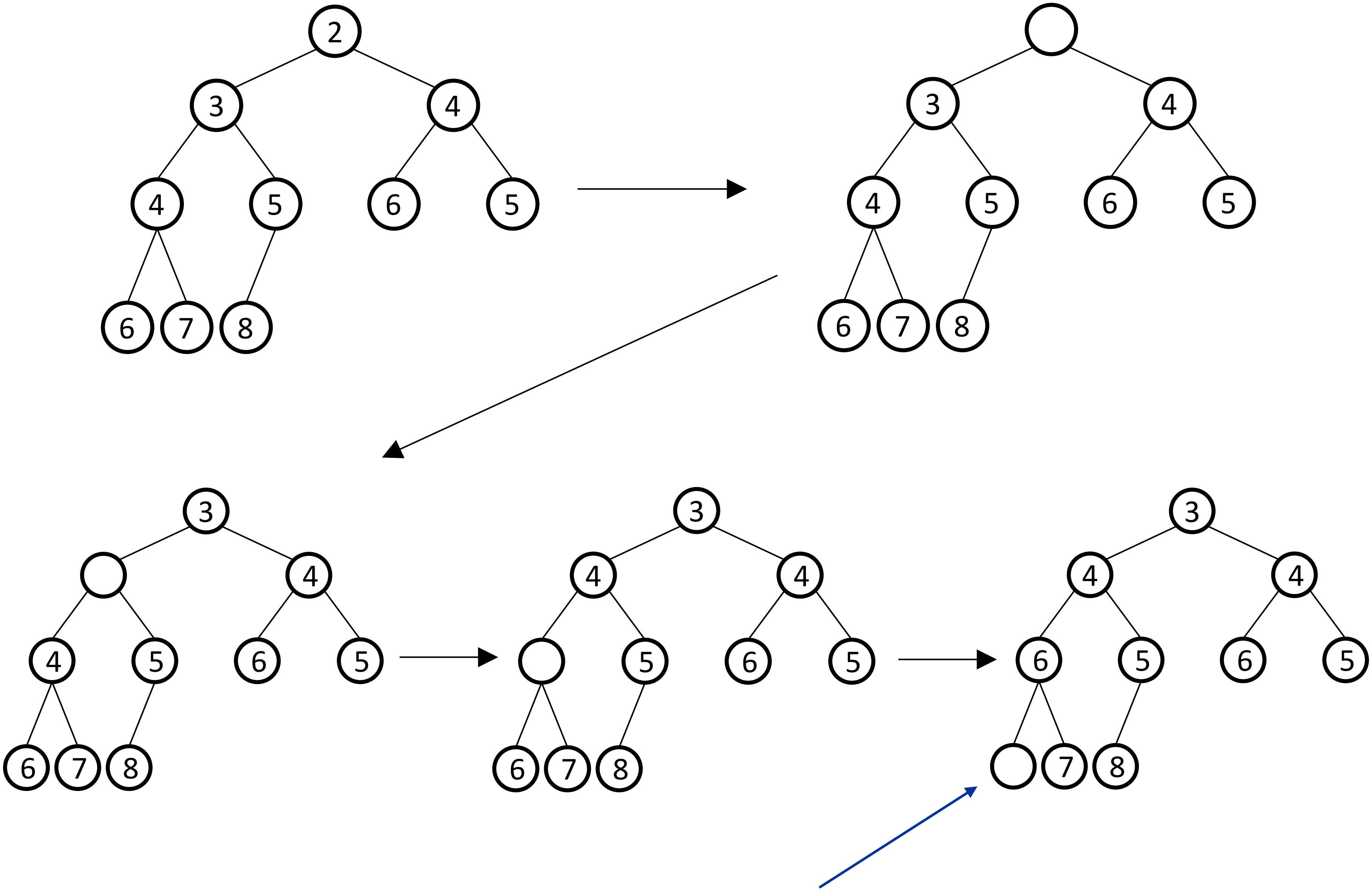
Swap $A[j]$ and $A\left[\left\lfloor \frac{j}{2} \right\rfloor\right]$; // Bubble Up

$j := \left\lfloor \frac{j}{2} \right\rfloor$

end

end

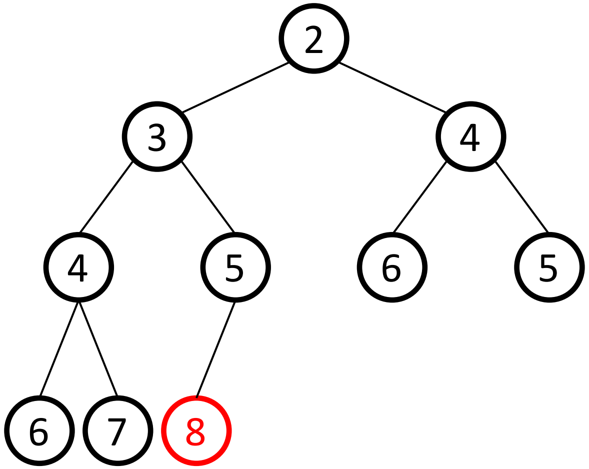
Extract-Min: First Attempt



Min-heap property preserved, **but completeness** not preserved!

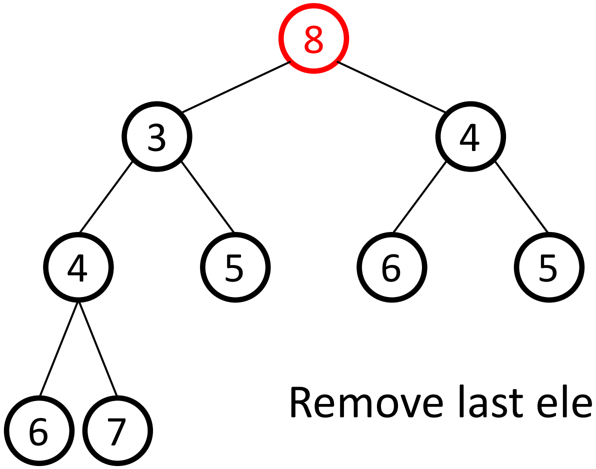
Extract-Min

- Copy the last element X to the root
  (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Extract-Min

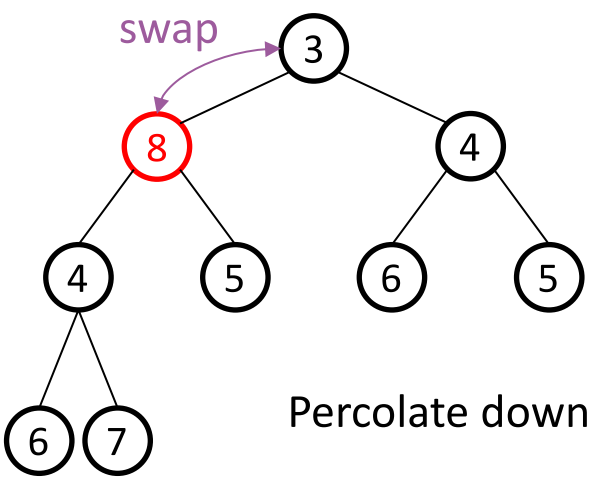
- Move the last element X to the root
(i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Remove last element and copy its value to the root

Extract-Min

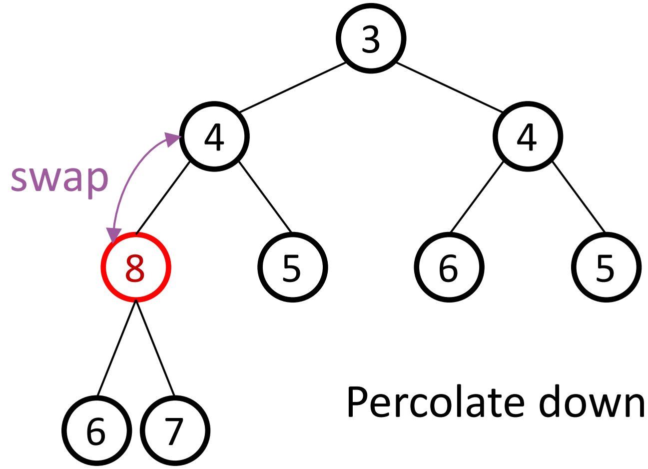
- Copy the last element X to the root
(i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling) down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Percolate down to maintain min-heap property

Extract-Min

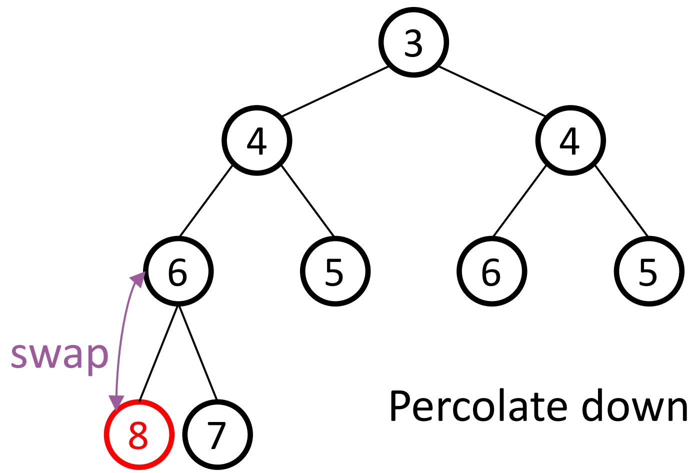
- Copy the last element X to the root
(i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Percolate down to maintain min-heap property

Extract-Min

- Copy the last element X to the root
  (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Percolate down to maintain min-heap property

- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing X (with respect to its children)
- Time complexity $= O(\text{height}) = O(\log n)$

Extract-Min

Extract-Min(i): Remove (smallest) item $A[1]$ in Heap and make $A[1 \cdots i-1]$ a Heap of remaining elements.
Empty array cells will contain an ∞ as an empty flag.

```
begin
    Output(A[1]);
    Swap A[1] and A[i]; A[i] := ∞; j := 1; // Remove smallest
    l := A[2j]; r := A[2j + 1];
    while A[j] > min(A[l], A[r]) do
        // if A[j] larger than a child, swap with min child
        if l < r then
            Swap A[j] with A[2j]; j := 2j;
        else
            Swap A[j] with A[2j + 1]; j := 2j + 1;
        end
        l := A[2j]; r := A[2j + 1];
    end
End:
```

Heapsort

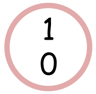
- **Build a binary heap of n elements**
  - the minimum element is at the top of the heap
  - insert n elements one by one $\Rightarrow O(n \log n)$
    (A more clever approach can do this in $O(n)$ time.)

- **Perform n Extract-Min operations**
  - the elements are extracted in sorted order
  - each Extract-Min operation takes $O(\log n)$ time
    $\Rightarrow O(n \log n)$

- Total time complexity: $O(n \log n)$

Heapsort example: Insert(10)

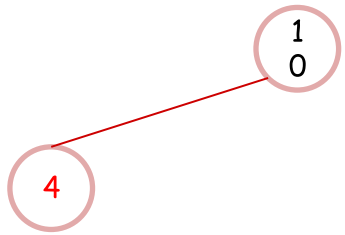
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(4)

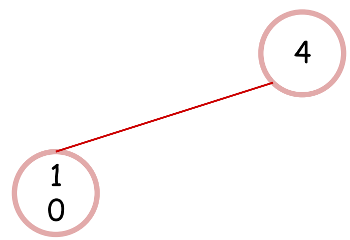
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(4)

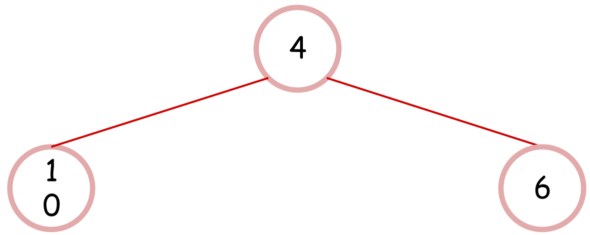
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(6)

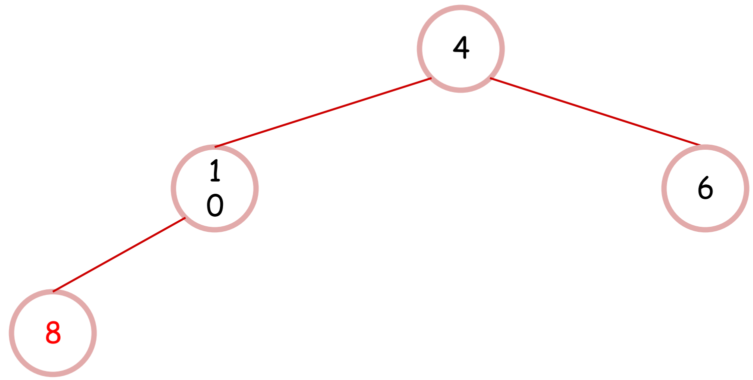
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(8)

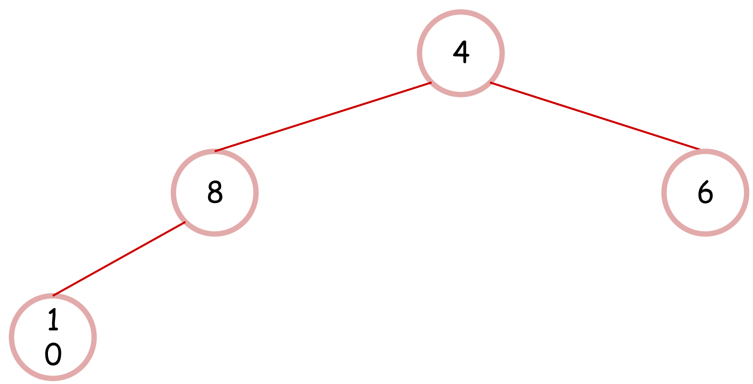
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(8)

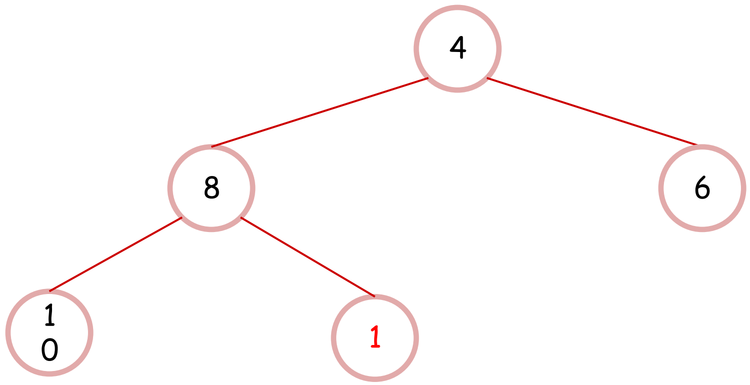
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(1)

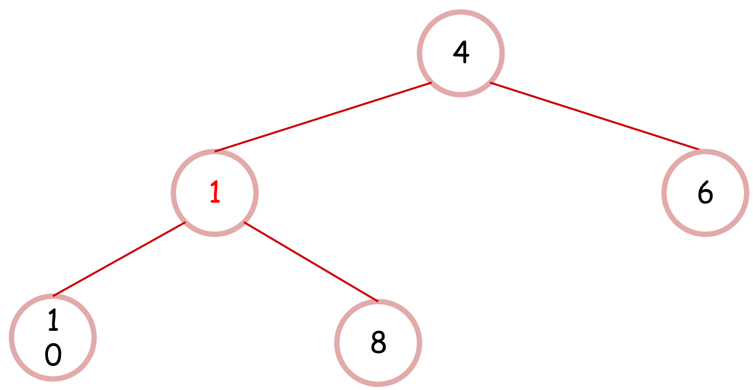
Input	10	4	6	8	1	3	2	5	7	9



Outpu t										

Heapsort example: Insert(1)

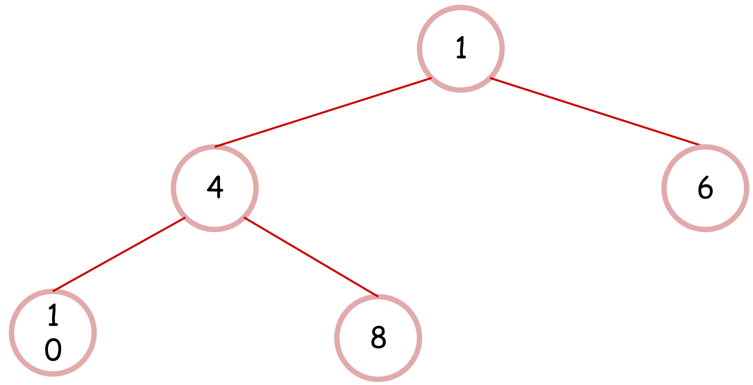
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(1)

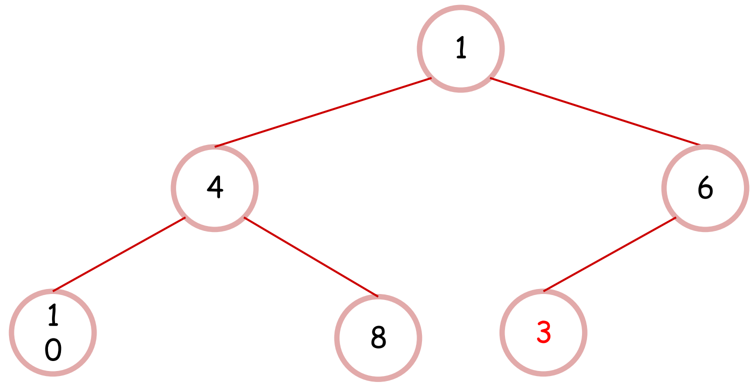
Input	10	4	6	8	1	3	2	5	7	9



Outpu t										

Heapsort example: Insert(3)

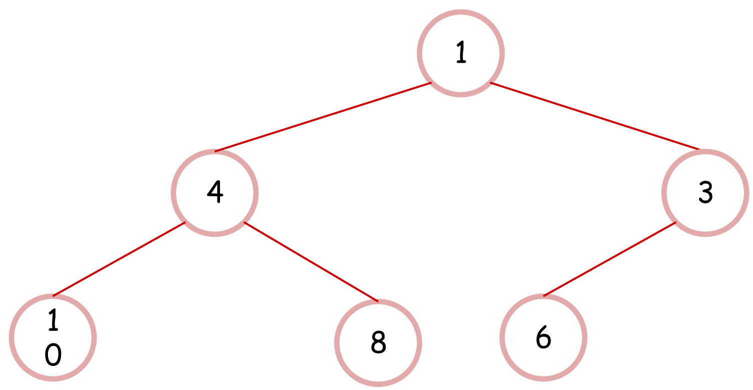
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(3)

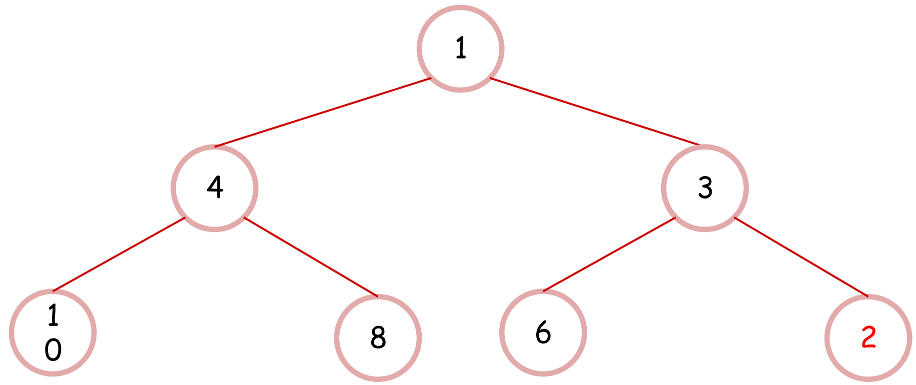
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example

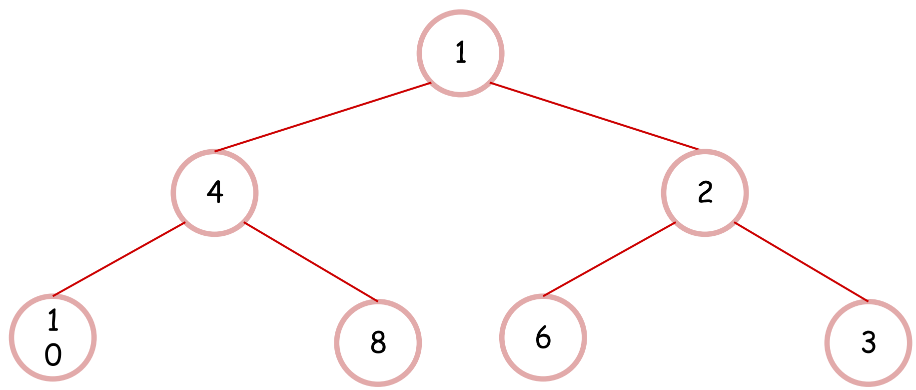
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example

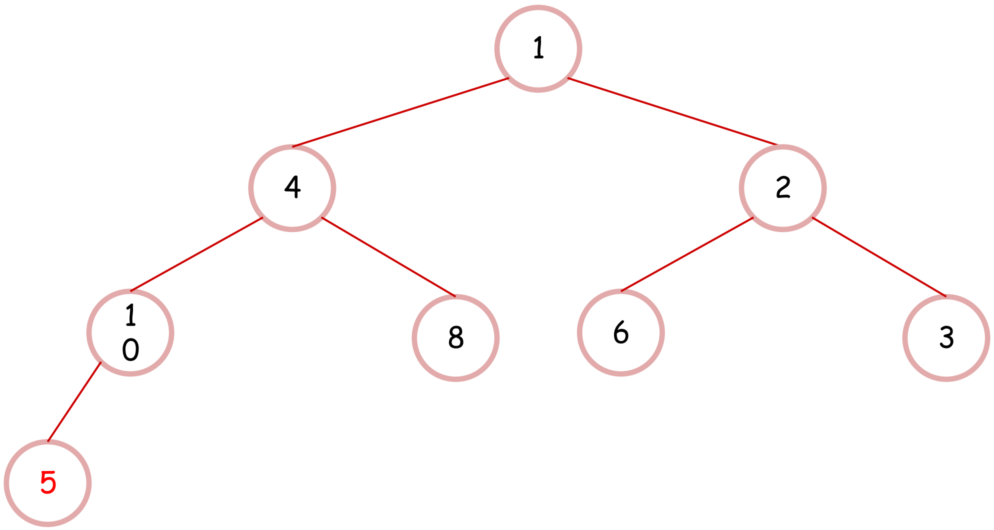
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(5)

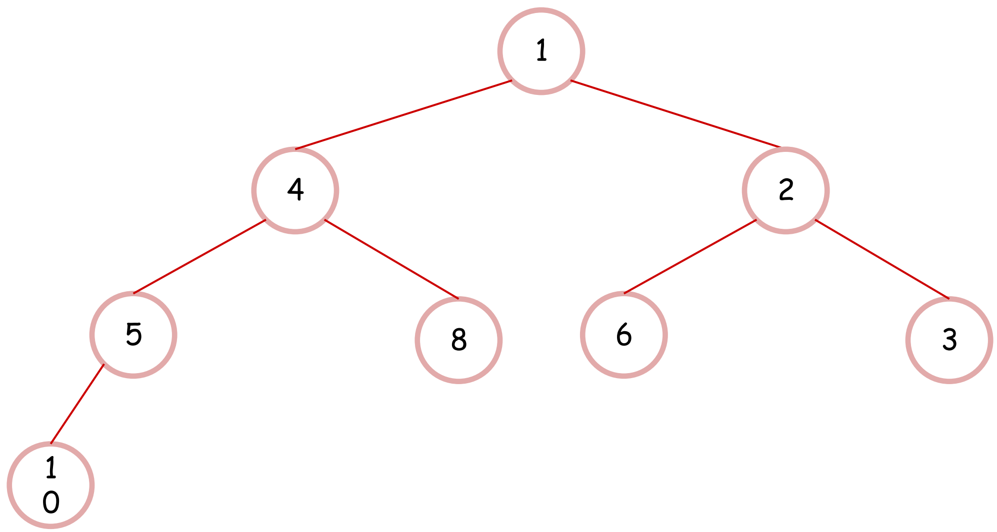
Input	10	4	6	8	1	3	2	5	7	9



Outpu t										

Heapsort example: Insert(5)

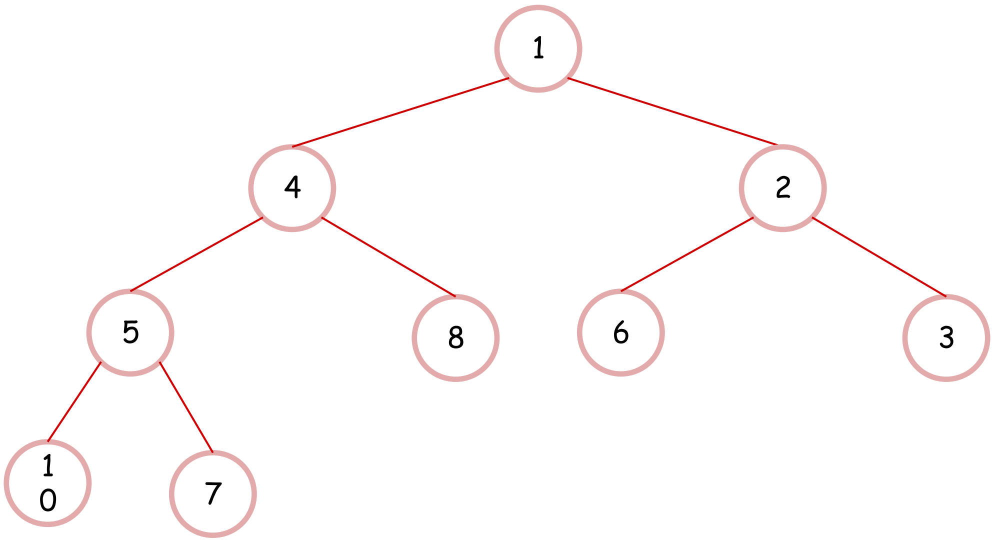
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(7)

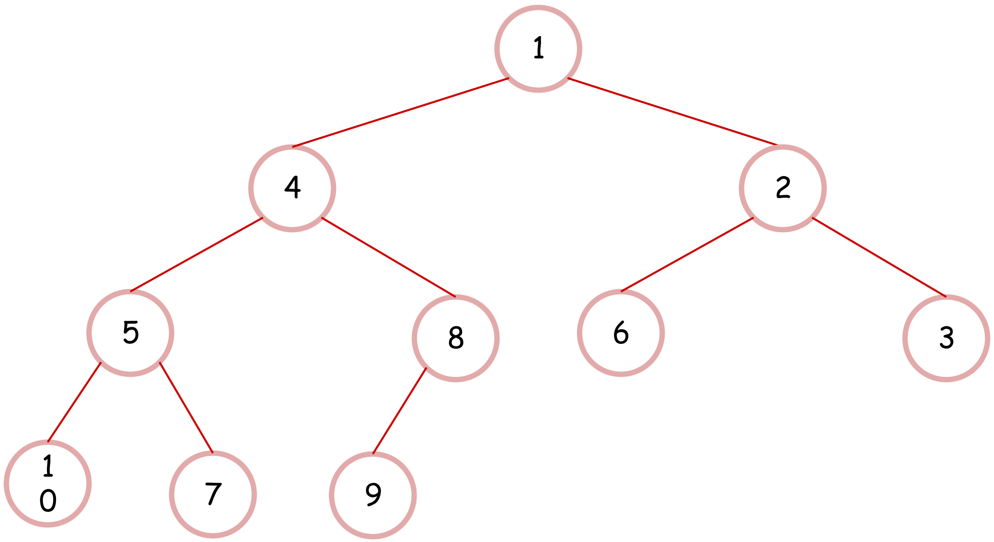
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Insert(9)

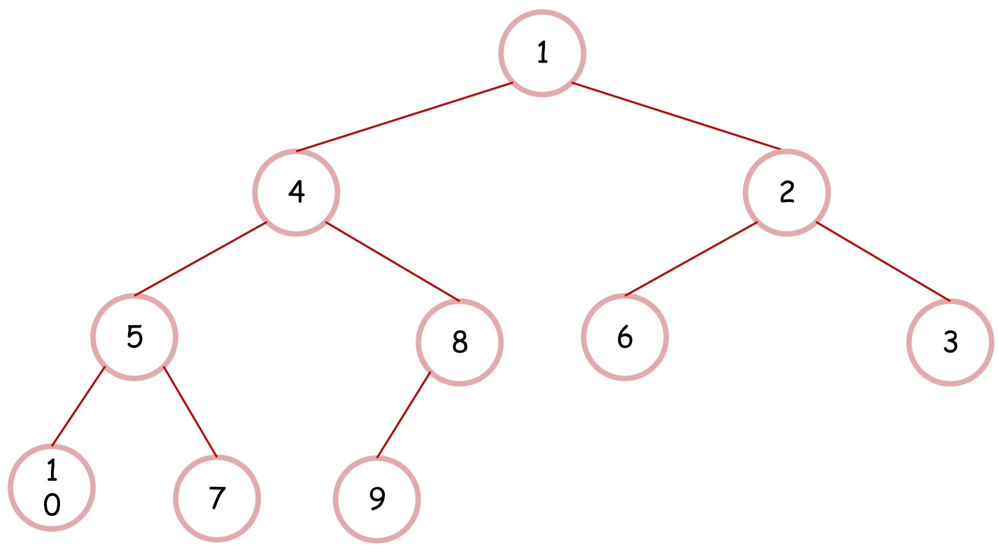
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Now extract the items one at a time

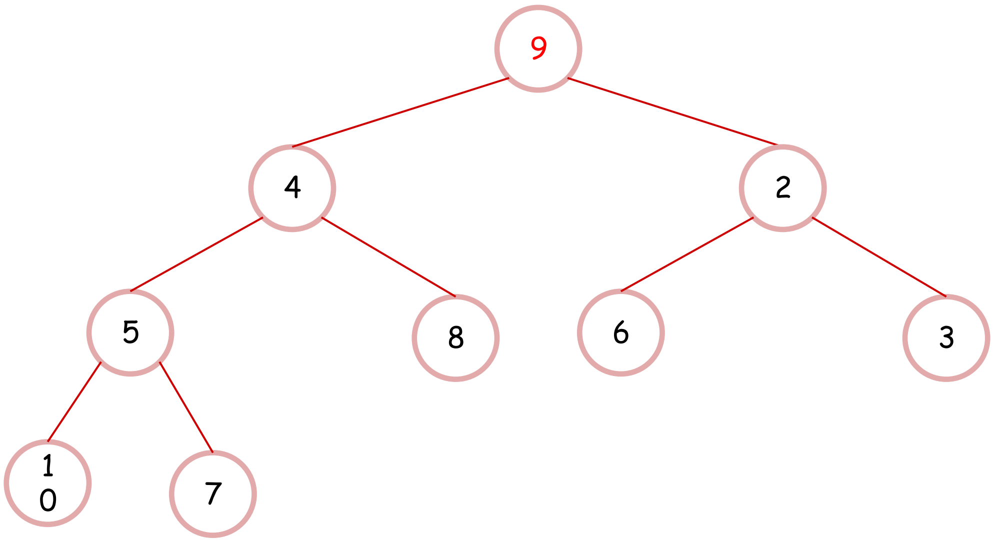
Input	10	4	6	8	1	3	2	5	7	9



Output										

Heapsort example: Extract-Min( )

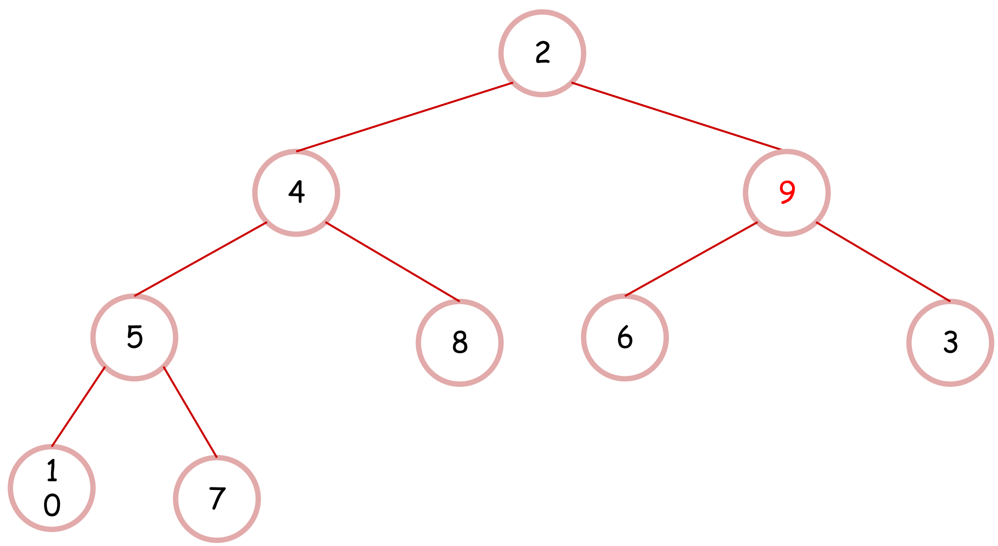
Input	10	4	6	8	1	3	2	5	7	9



Outpu t	1									

Heapsort example

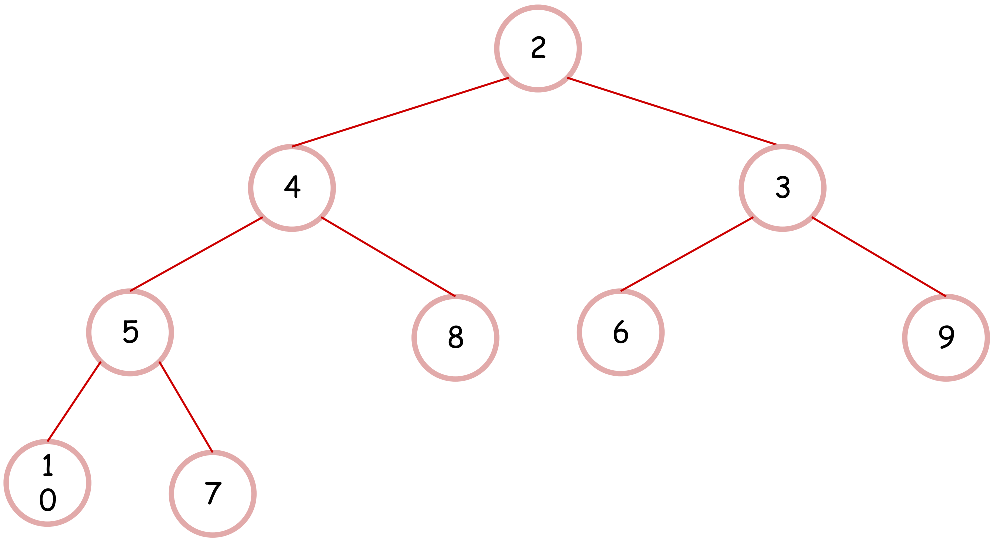
Input	10	4	6	8	1	3	2	5	7	9



Outpu t	1									

Heapsort example

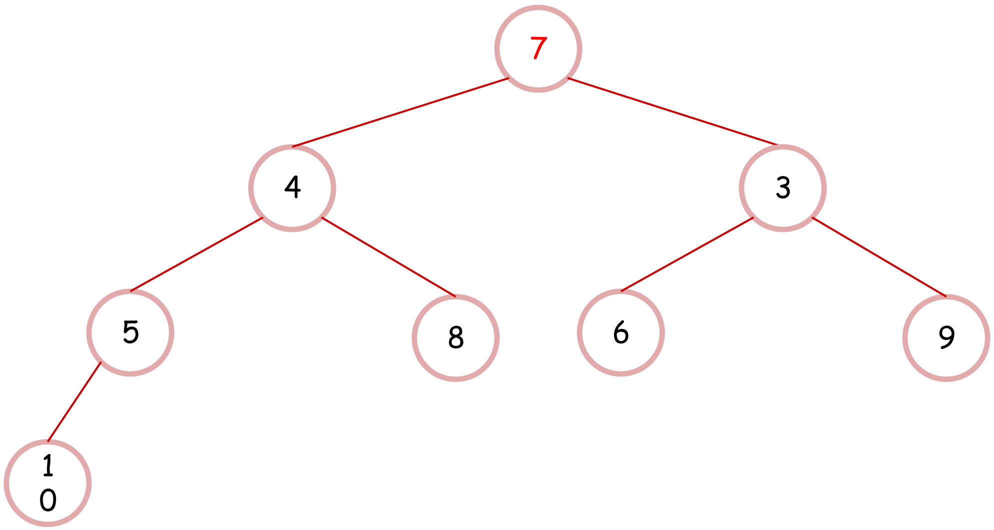
Input	10	4	6	8	1	3	2	5	7	9



Output	1									

Heapsort example: Extract-Min( )

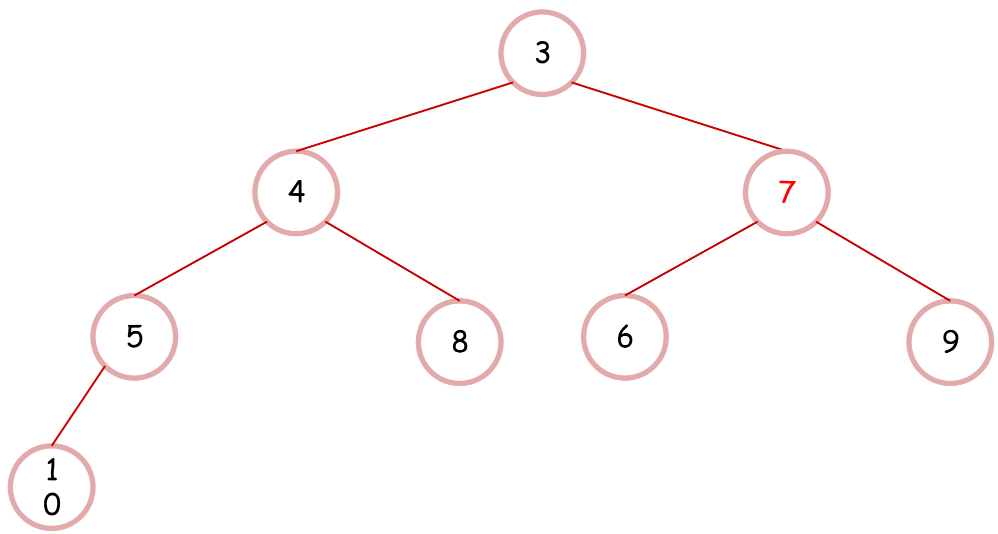
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2								

Heapsort example

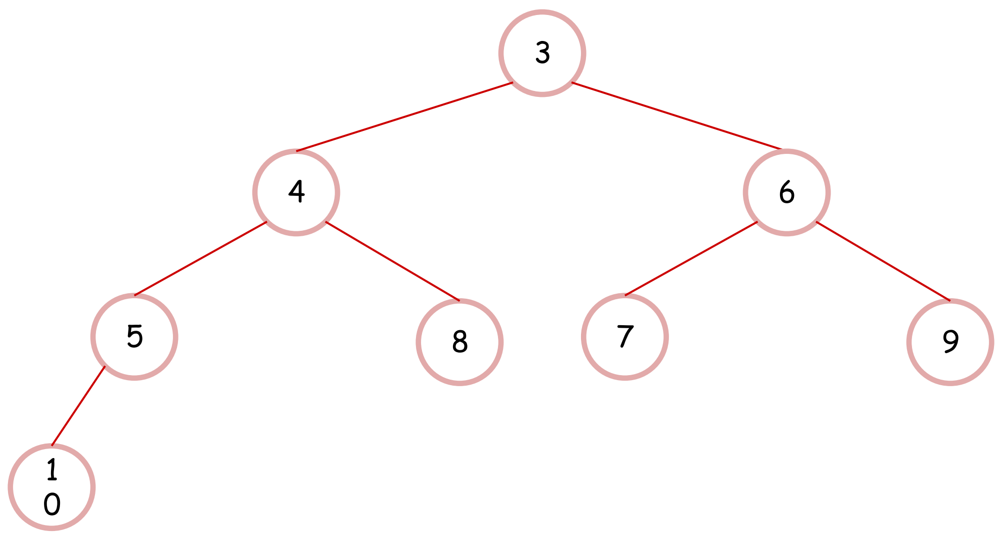
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2								

Heapsort example

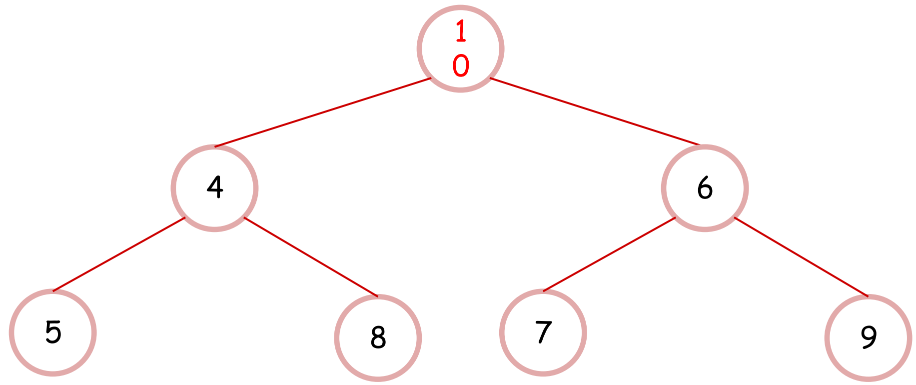
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2								

Heapsort example: Extract-Min( )

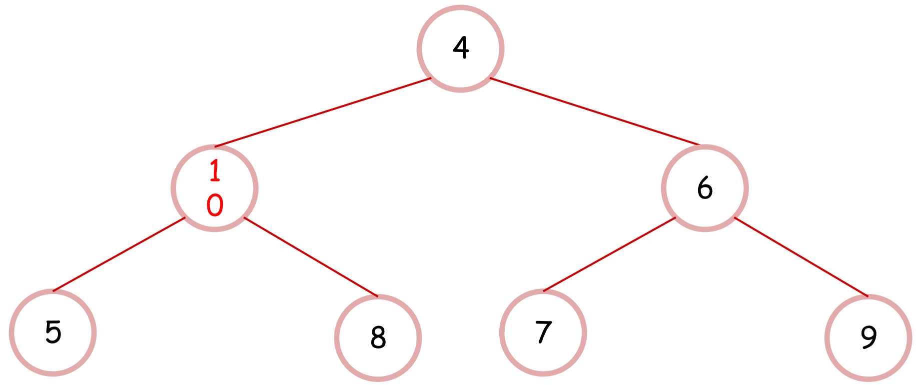
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3							

Heapsort example

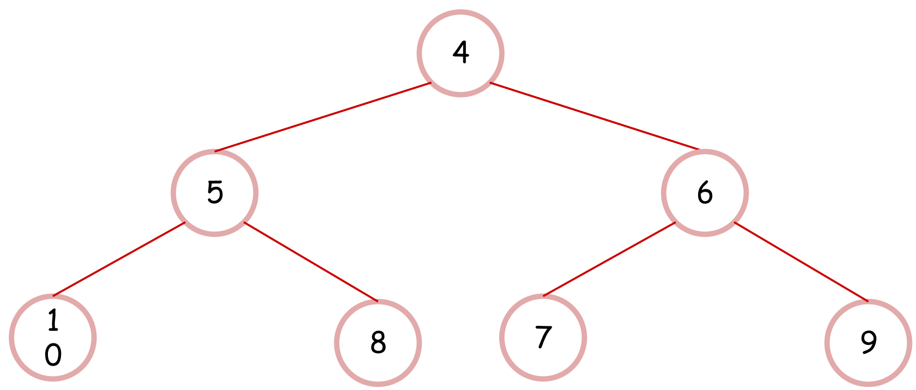
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3							

Heapsort example

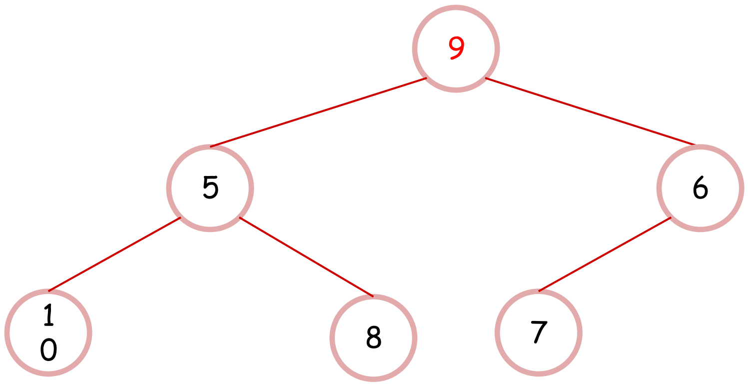
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3							

Heapsort example: Extract-Min( )

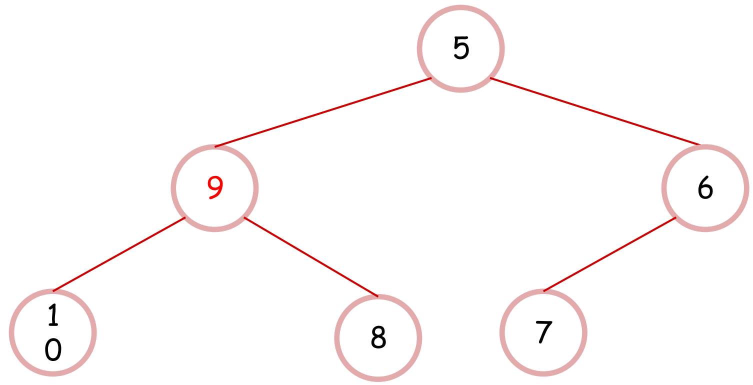
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4						

Heapsort example

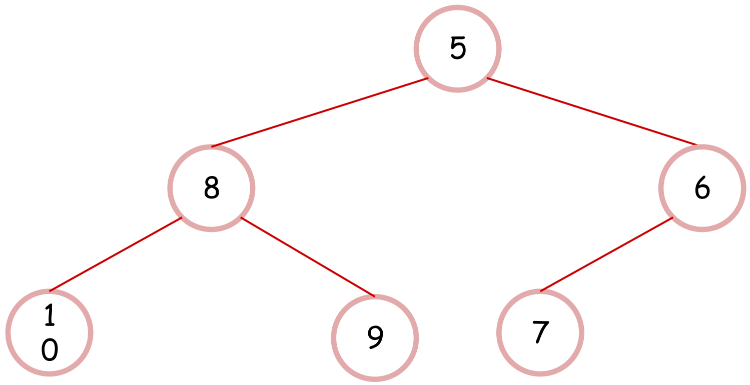
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4						

Heapsort example

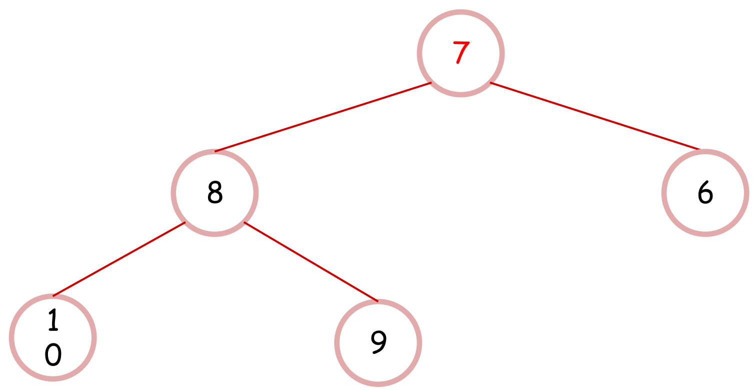
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4						

Heapsort example: Extract-Min( )

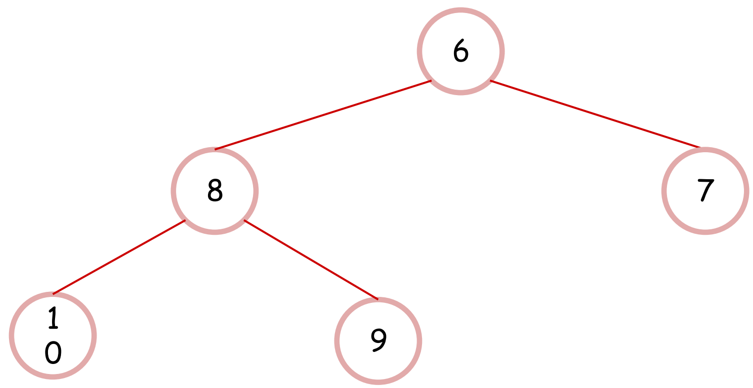
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4	5					

Heapsort example

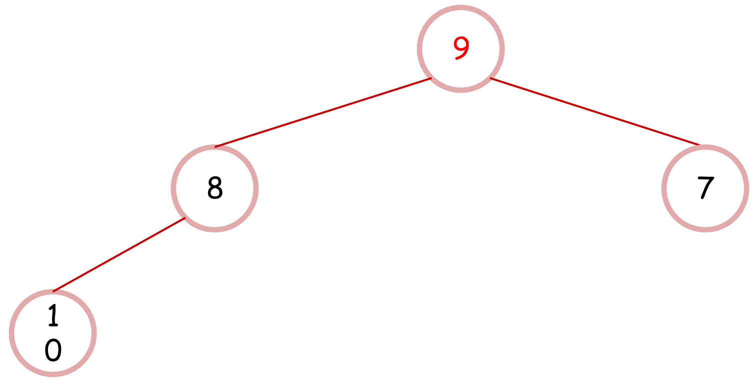
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4	5					

Heapsort example: Extract-Min( )

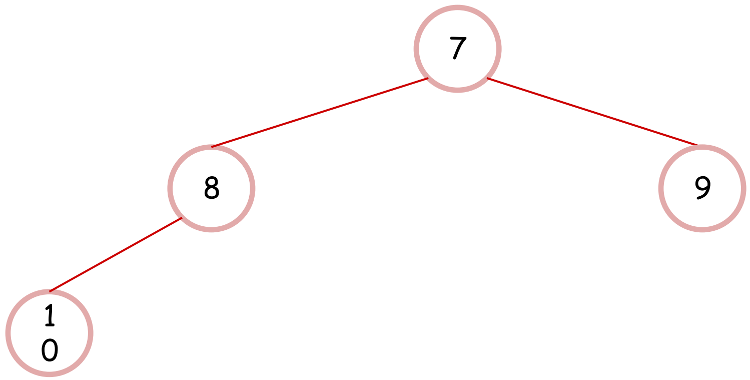
Input	10	4	6	8	1	3	2	5	7	9



Outpu t	1	2	3	4	5	6				

Heapsort example

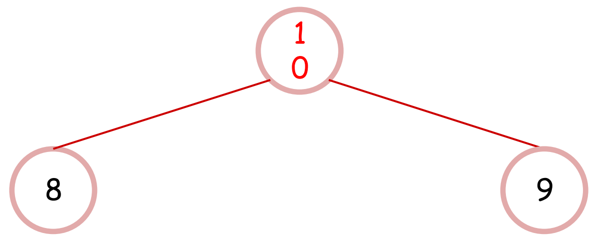
Input	10	4	6	8	1	3	2	5	7	9



Outpu t	1	2	3	4	5	6				

Heapsort example: Extract-Min( )

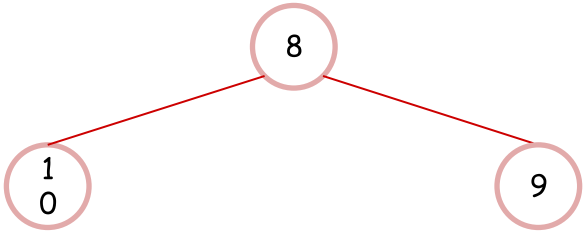
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4	5	6	7			

Heapsort example

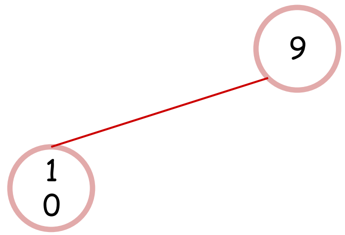
Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4	5	6	7			

Heapsort example: Extract-Min( )

Input	10	4	6	8	1	3	2	5	7	9



Output	1	2	3	4	5	6	7	8		

Heapsort example: Extract-Min( )

Input	10	4	6	8	1	3	2	5	7	9

10

Output	1	2	3	4	5	6	7	8	9	

Heapsort example: Extract-Min( )

Input	10	4	6	8	1	3	2	5	7	9

SORTED

Output	1	2	3	4	5	6	7	8	9	10

Summary

- *A Priority queue* is an abstract data structure that supports two operations: Insert and Extract-Min.

- If priority queues are implemented using heaps, then these two operations are supported in $O(\log n)$ time.

- Heapsort takes $O(n\log n)$ time, which is as efficient as merge sort and quicksort.

Exercise on merging k sorted arrays

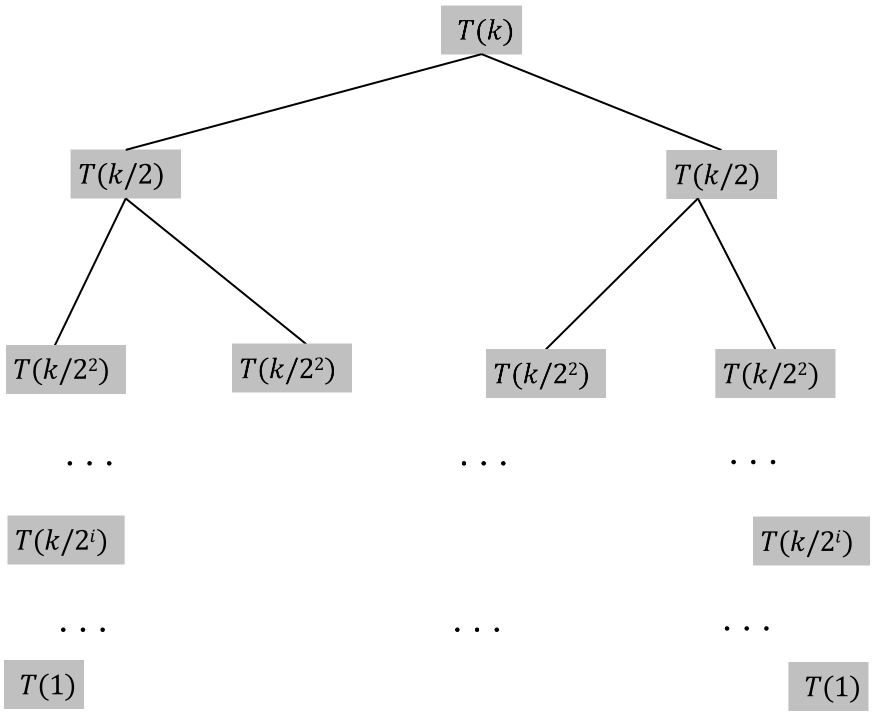
- Suppose that you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements
- First strategy: Recall the procedure for merging two sorted arrays used in the "combine" step of merge-sort. Using this procedure, we merge the first two arrays, then merge in the third, then merge in the fourth, and so on. Analyze the worst-case running time of this algorithm, in terms of k and n.
- The cost of merging two sorted arrays of size n into an array of size $2n$ is $2n$. So the first merge step takes $2n$, the second step $3n$ and so on. The final step takes kn. The total running time is $2n + 3n + \cdots + kn = O(k^2n)$.

Exercise on merging k sorted arrays (D&C)

- Suppose that you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements
- Second strategy: Design a more efficient solution using divide and conquer, and analyze its running time.
- Divide recursively k sorted arrays into two parts, each with $k/2$ arrays. When the subproblems have been solved, we get two sorted arrays of size $kn/2$ to merge.
- The merge step has cost kn. The recurrence is $T(k) = 2T(k/2) + kn$. So $T(k) = O(kn \log k)$.

Recursion Tree merging k sorted arrays

$$T(k) = 2T\left(\frac{k}{2}\right) + kn, T(1) = 1$$



Lv	#pr	work/pr	work/lv
0	1	kn	kn
1	2	$kn/2$	kn
2	2^2	$kn/2^2$	kn
i	2^i	$kn/2^i$	kn
$\log_2 k$	$2^{\log_2 k} = k$	1	k

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

Exercise on merging k sorted arrays (Heaps)

- Suppose that you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements
- Third strategy: Design another efficient solution based on the min-heap implementation of priority queues.
- Insert the first element of each array into an empty min-heap. Apply extract-min to get the smallest item in the min-heap, followed by inserting the next item of the array that the previous item belongs to. Repeat doing this until all items have been inserted into and extracted from the min-heap.
- Since, at any time, the size of the min-heap is at most k, each min-heap operation takes $O(\log k)$. Each of the kn items is being inserted and extracted exactly once, so the total running time is $O(kn \log k)$.

New Operation

Sometimes priority queues need to support another operation

called Decrease-Key

- Decrease-Key: decreases the value of one specified element
- Decrease-Key is used in later algorithms, e.g., in Dijkstra's algorithm for finding Shortest Path Trees

Question

How can heaps be modified to support Decrease-Key in $O(\log n)$ time?