Sorting Algorithms

Lecture 7: 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

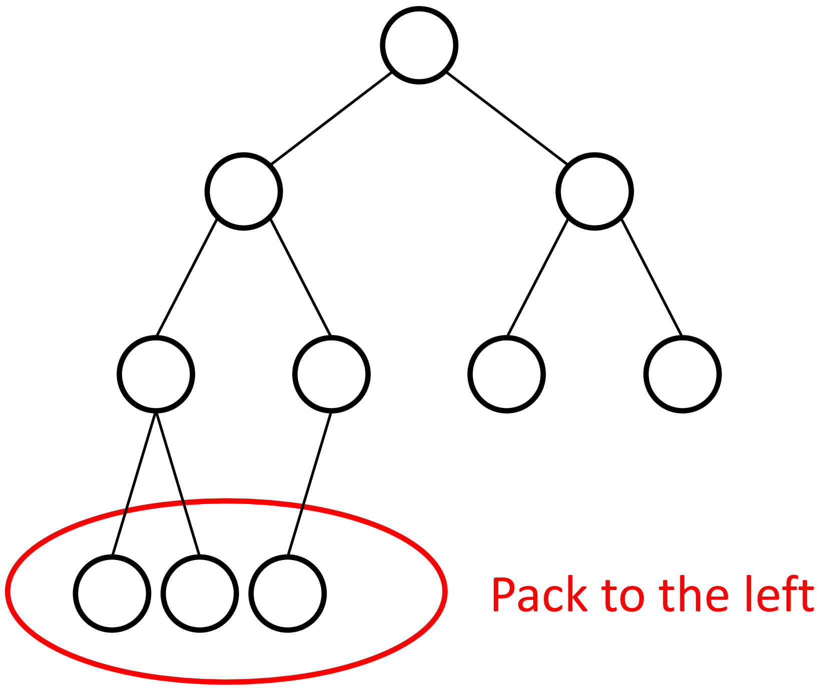
- sorted (circular) array
  - **Insert in $O(n)$ time**
  - **Extract-Min in $O(1)$ time**

- sorted doubly linked list
  - **Insert in $O(n)$ time**
  - **Extract-Min in $O(1)$ time** *(given pointer to minimum-item)*

Question

Is there any data structure that supports both these priority queue operations in $O(log\ n)$ time?

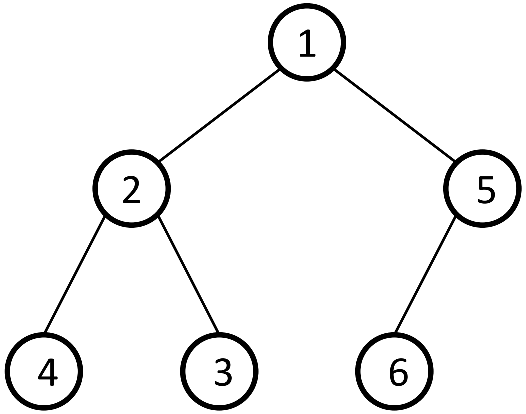
(Binary) Heap



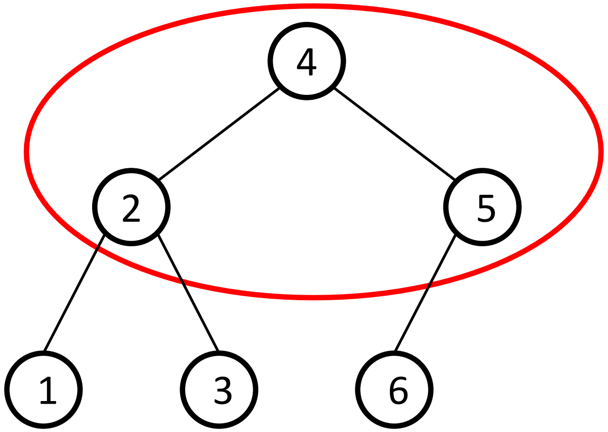
Heaps are "almost complete binary trees"

- 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:

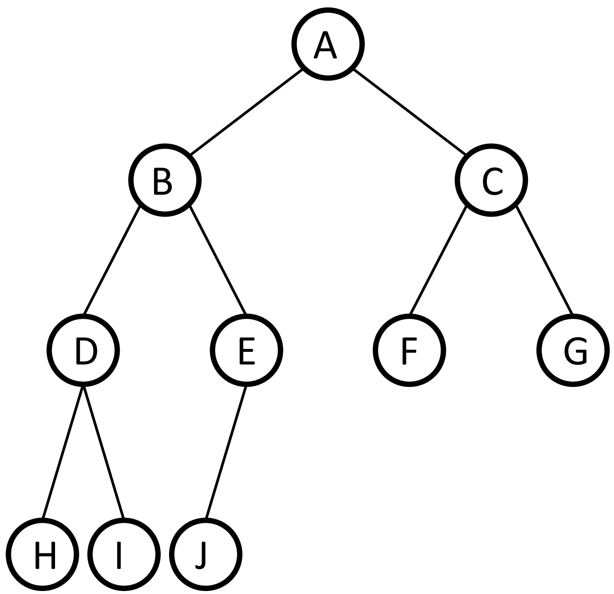
The value of a node is at least the value of its parent — Min-heap

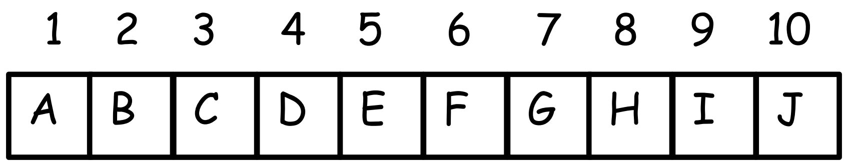
Heap Properties

- 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**

- Structural properties
  - Fact from Discrete Math
    A heap of height h has between 2^h to $2^{h+1} - 1$ nodes.
  - If $2^h \leq n < 2^{h+1}$ => $h \leq \log_2 n < h + 1$
    => 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!!!

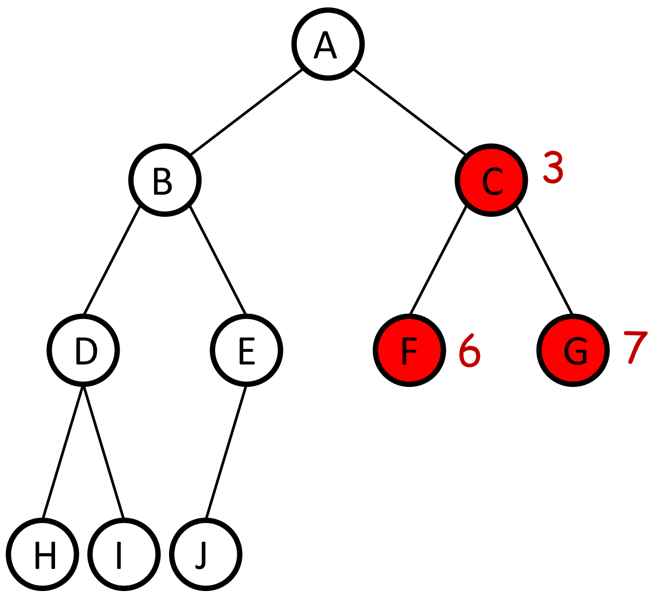
Array Implementation of Heap





- 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



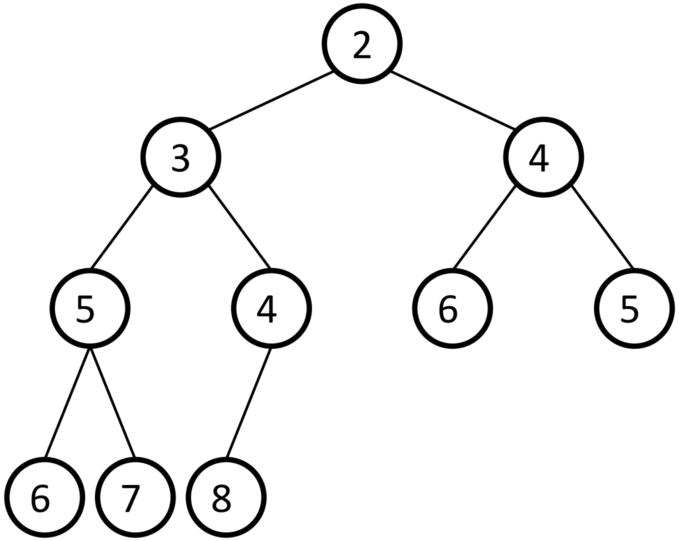
1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

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

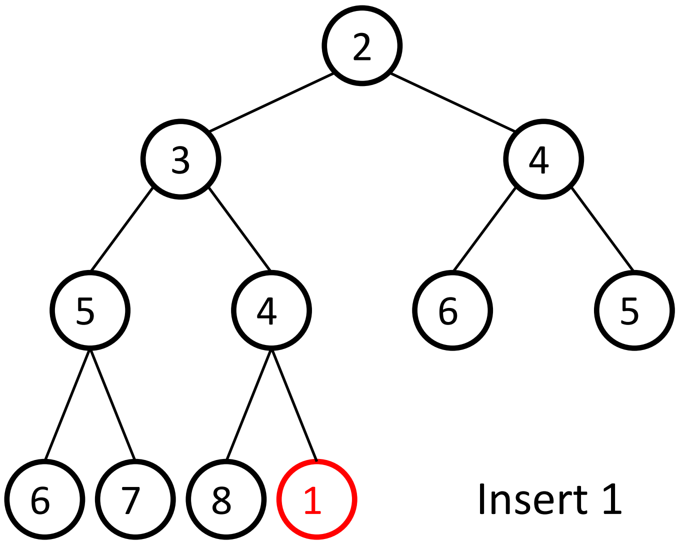
Insertion

- 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

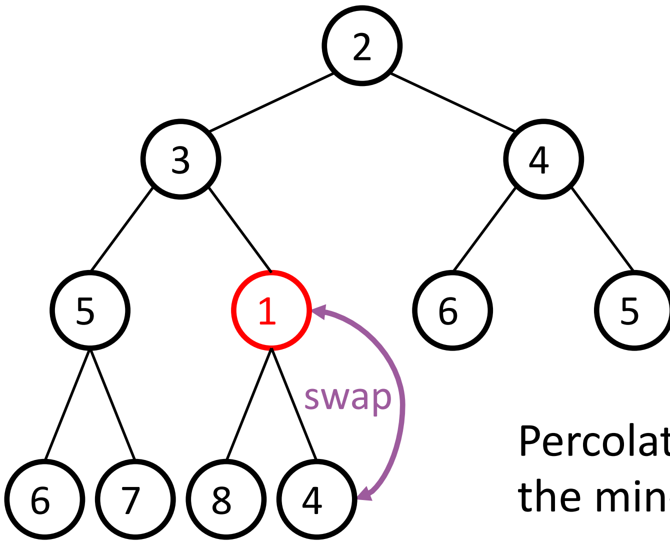
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
    - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Insert 1

Insertion

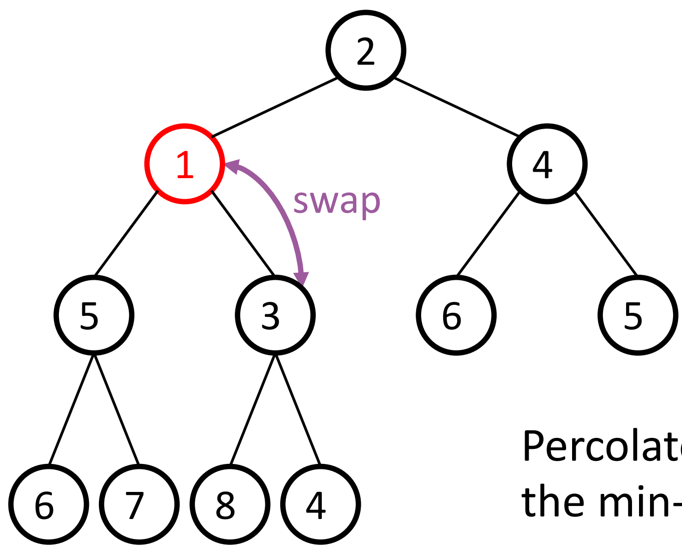
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Percolate up to maintain the min-heap property

Insertion

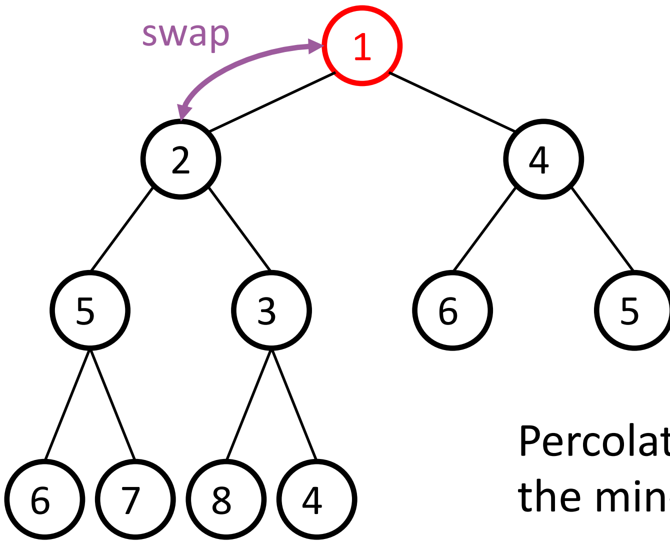
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



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.

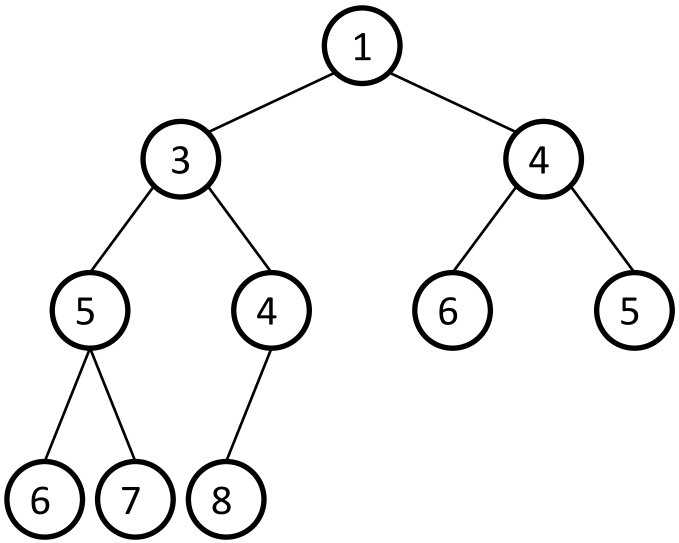


Percolate up to maintain the min-heap property

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

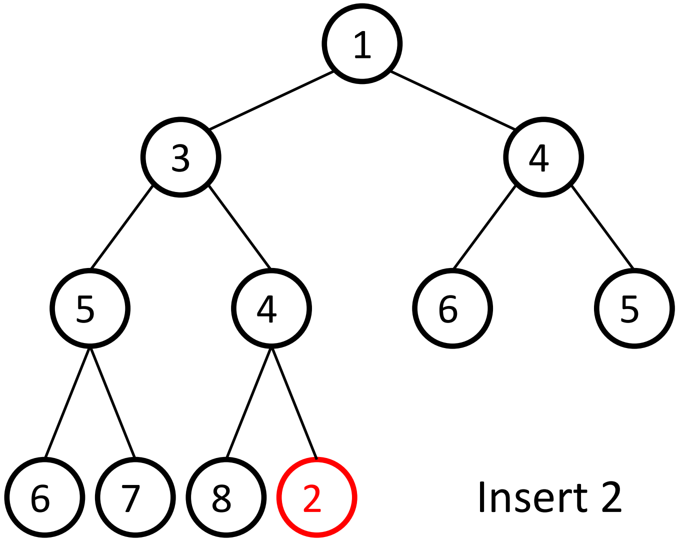
Insertion (2nd Example)

- 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.



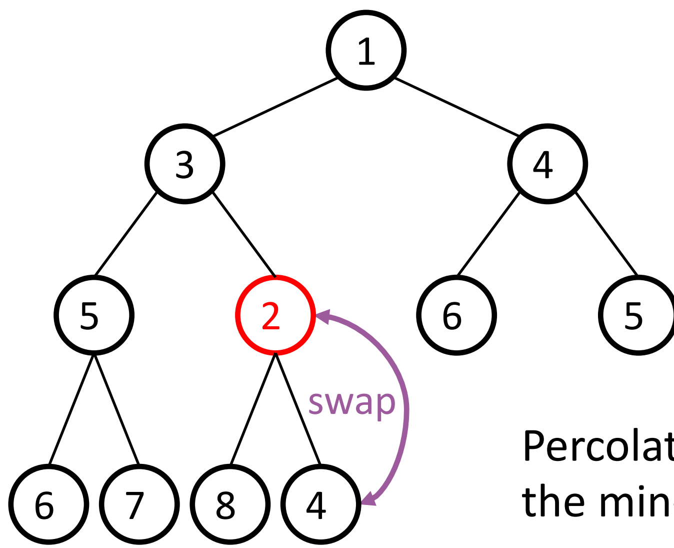
Insertion (2nd Example)

- 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)

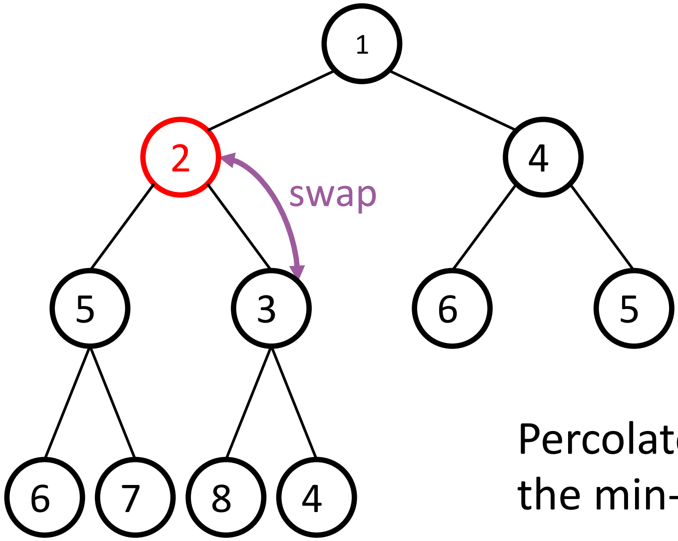
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



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

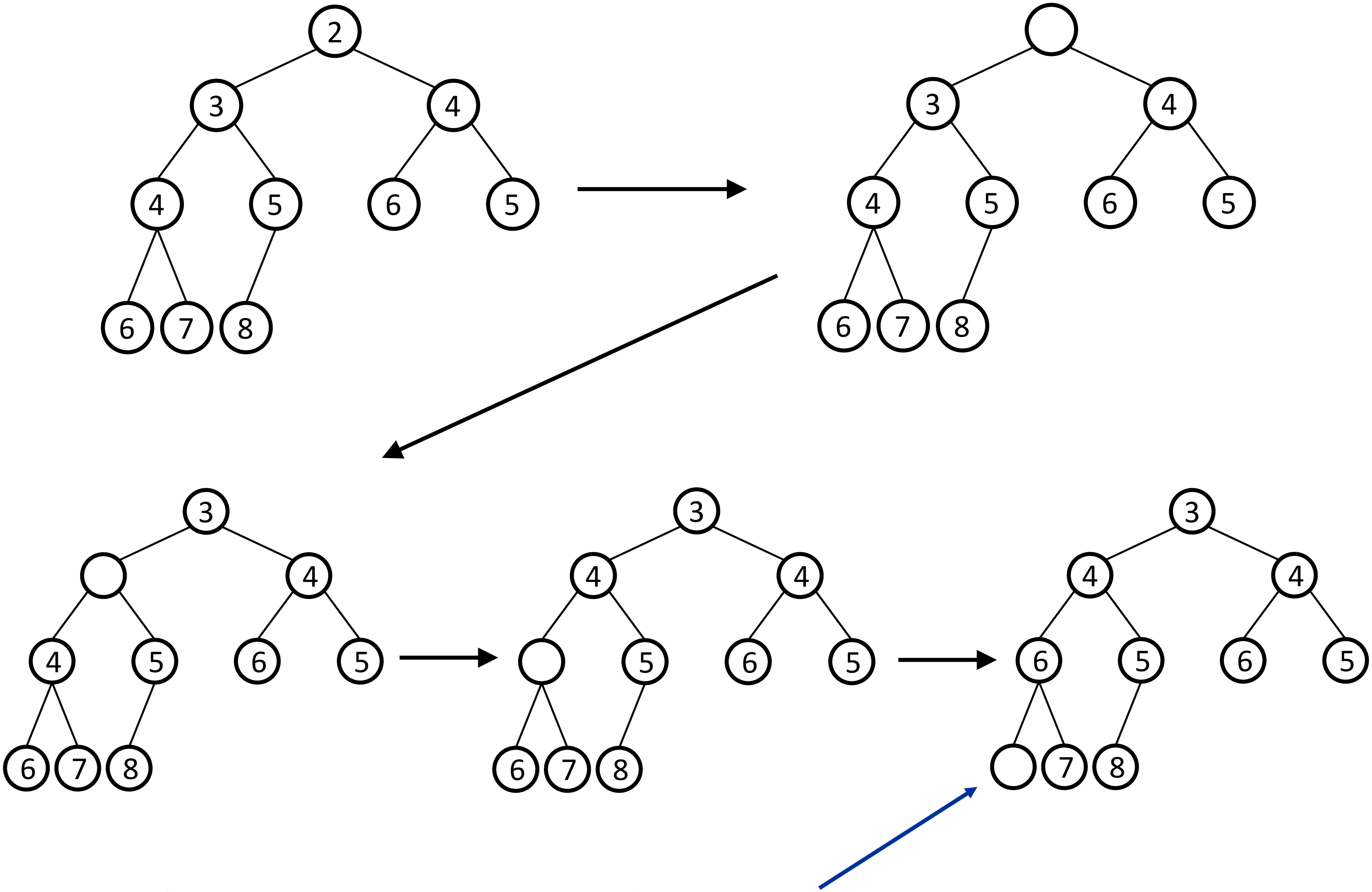
In this example, swapping stopped BEFORE reaching the top.

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 $A[j] < A[\lfloor \frac{j}{2} \rfloor]$ **and** $j > 1$ **do**
    // $A[j]$ is less than its parent
    Swap $A[j]$ and $A[\lfloor \frac{j}{2} \rfloor]$; // Bubble Up
    $j = \lfloor \frac{j}{2} \rfloor$
  end
end

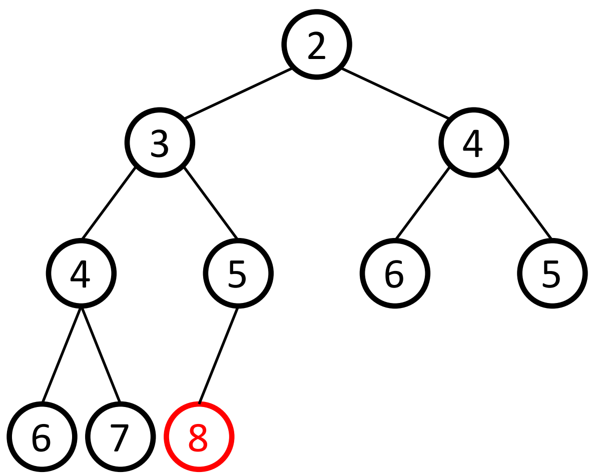
Extract-Min: First Attempt



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

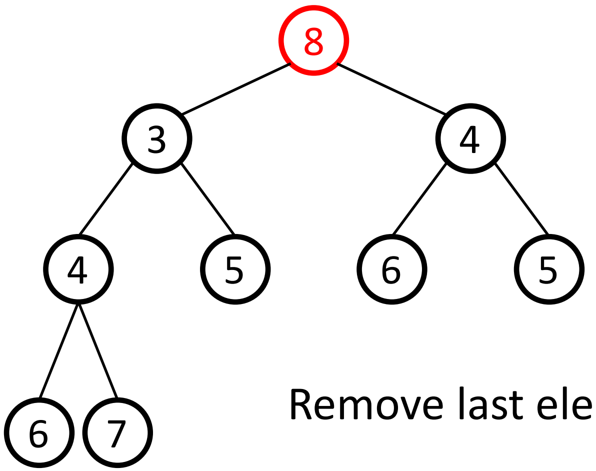
Extract-Min

- 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

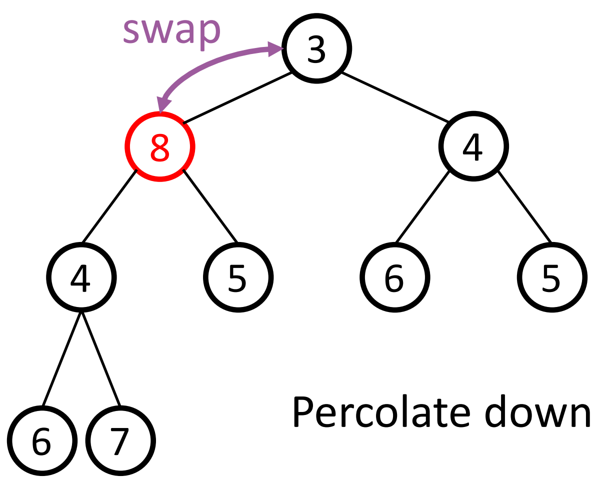
- Copy the last element X to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Remove last element and copy its value to the root

Extract-Min

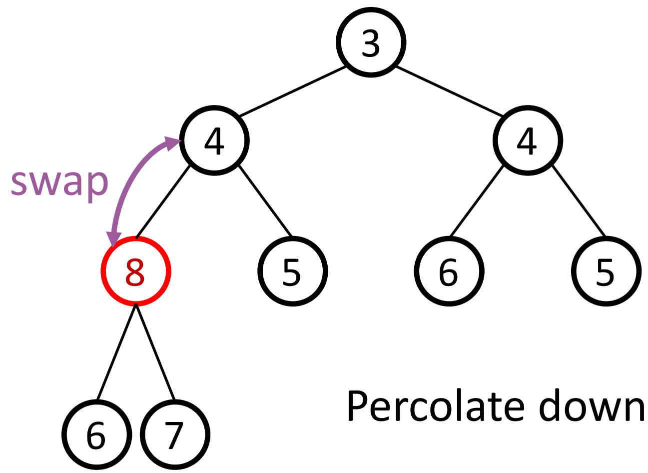
- Copy the last element X to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling) down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Percolate down to maintain min-heap property

Extract-Min

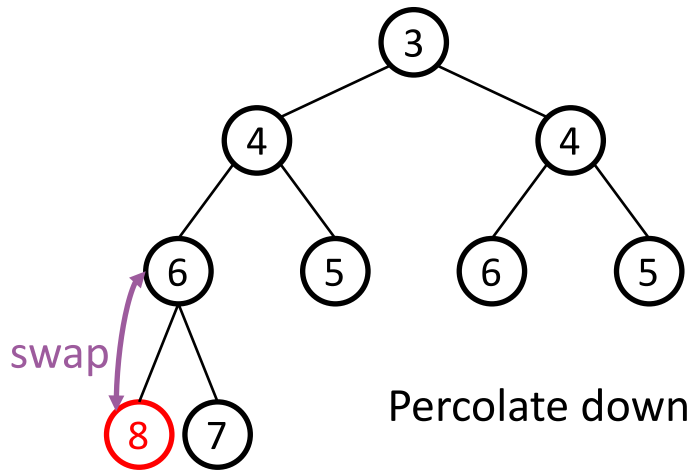
- Copy the last element X to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolating (or bubbling down): if the element is larger than either of its children, then interchange it with the smaller of its children.



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 $A[i]$; $A[i] = \infty$; $j = 1$; // Remove smallest
  $l = A[2j]; r = A[2j+1]$;
  while $A[j] > \min(l, r)$ **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: $O(n \log n)$

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.

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?

Going Further

- For some algorithms, there are other desirable Priority Queue operations, e.g., *Delete* an arbitrary item and *Melding* or taking the union of two priority queues

- There is a tradeoff between the costs of the various operations. Depending upon where the data structure is used, different priority queues might be better.

- Most famous variants are *Binomial Heaps* and *Fibonacci Heaps*