COMP90038
Algorithms and Complexity

Lecture 13: Priority Queues, Heaps and Heapsort

(with thanks to Harald Søndergaard)

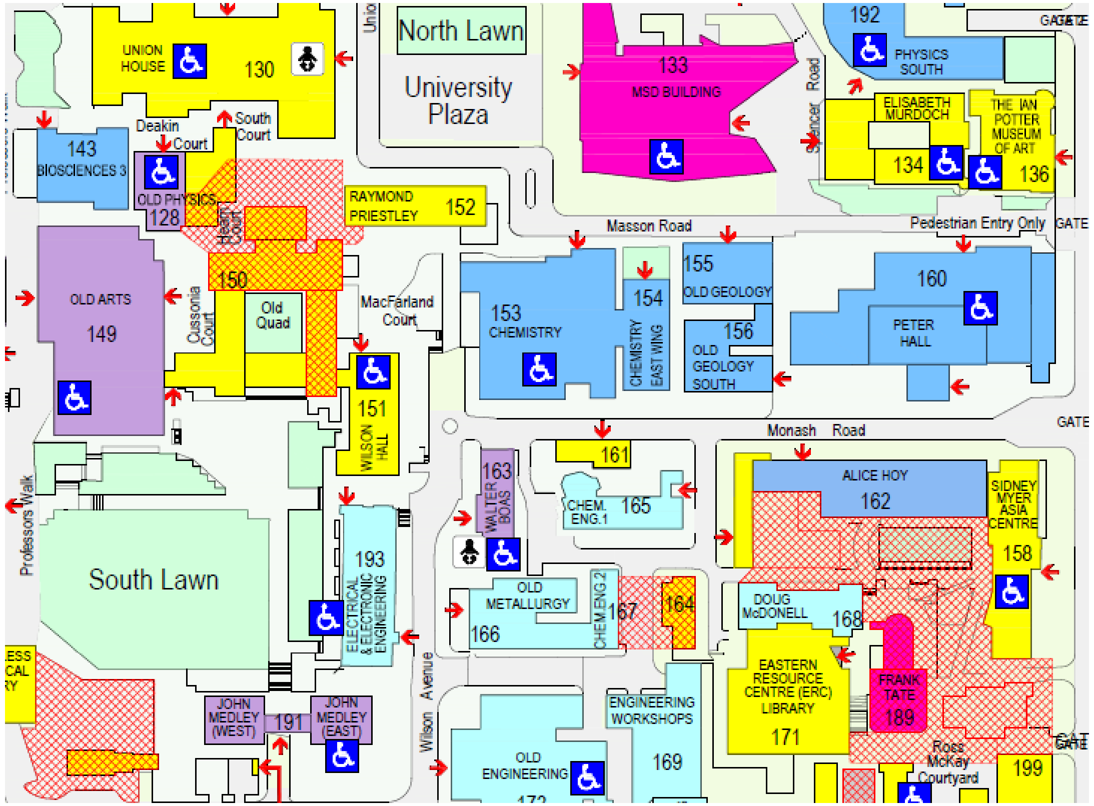
Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

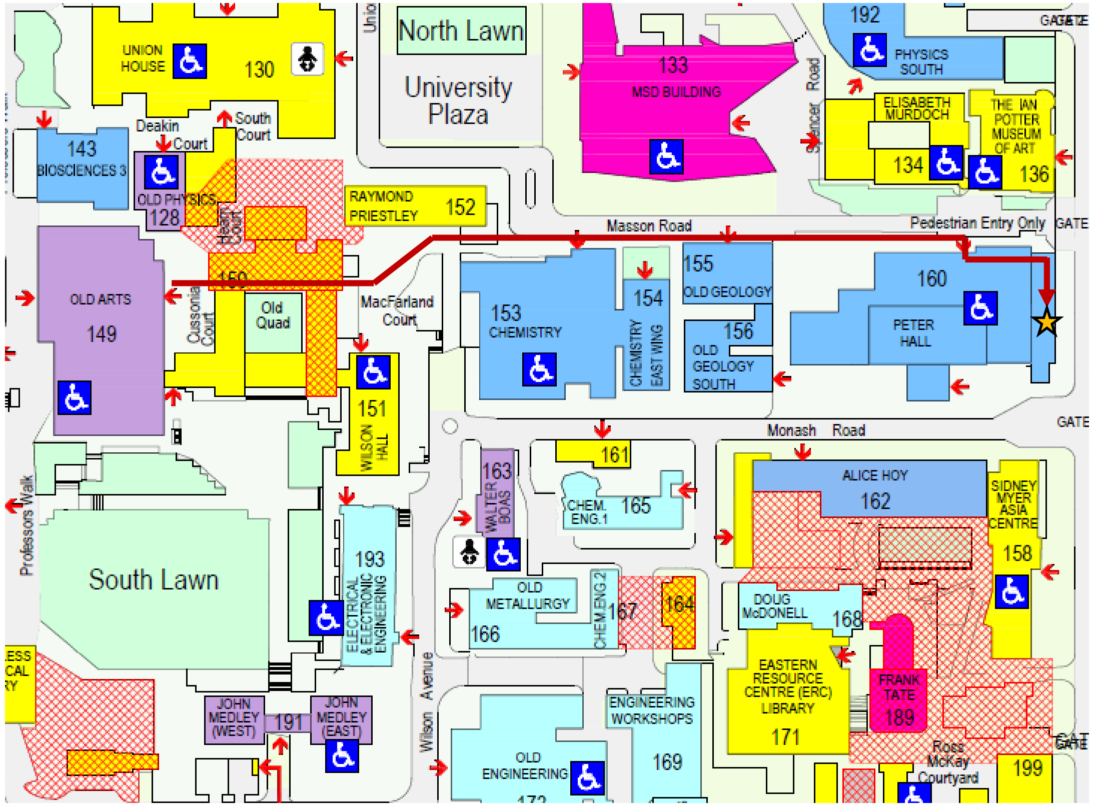
Where to find me?

- My office is at the Peter Hall building (Room G.83)



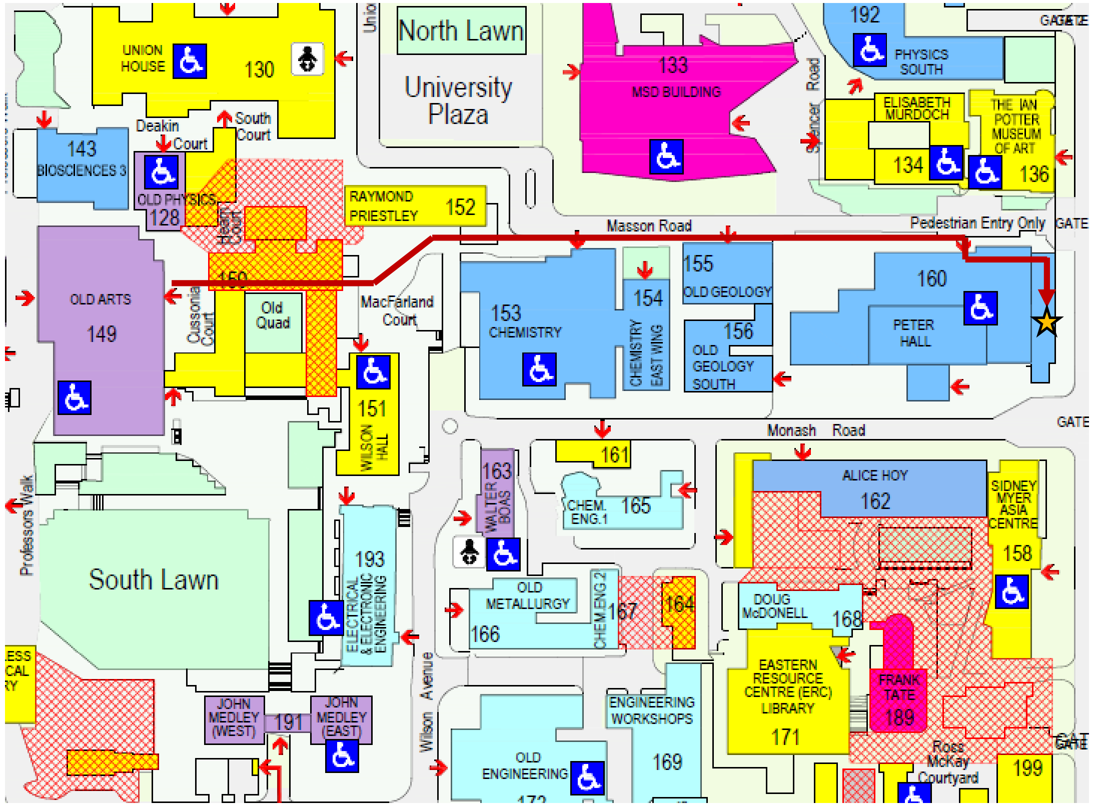
Where to find me?

- My office is at the Peter Hall building (Room G.83)



Where to find me?

- My office is at the Peter Hall building (Room G.83)



- Consultation hours:
  - Wednesdays 10:00am-11:00am
  - By appointment on Monday/Friday (limited slots)

Heaps and Priority Queues

- The **heap** is a very useful data structure for **priority queues**, used in many algorithms.
- A priority queue is a **set** (or **pool**) of elements.
- An element is injected into the priority queue together with a **priority** (often the key value itself) and elements are ejected according to priority.
- We think of the heap as a **partially ordered binary tree**.
- Since it can easily be maintained as a **complete** tree, the standard implementation uses an array to represent the tree.

The Priority Queue

- As an abstract data type, the priority queue supports the following operations on a “pool” of elements (ordered by some linear order):

  - **find** an item with maximal priority
  - **insert** a new item with associated priority
  - test whether a priority queue is empty
  - **eject** the **largest** element

- Other operations may be relevant, for example:

  - **replace** the maximal item with some new item
  - **construct** a priority queue from a list of items
  - **join** two priority queues

Some Uses of Priority Queues

- **Job scheduling** done by your operating system. The OS will usually have a notion of "importance" of different jobs.
- (Discrete event) **simulation** of complex systems (like traffic, or weather). Here priorities are typically event times.
- Numerical computations involving floating point numbers. Here priorities are measures of computational "error".

- Many sophisticated algorithms make essential use of priority queues (Huffman encoding and many shortest-path algorithms, for example).

Stacks and Queues as Priority Queues

- Special instances are obtained when we use **time** for priority:
  - If "large" means "late" we obtain the **stack**.
  - If "large" means "early" we obtain the **queue**.

Possible Implementations of the Priority Queue

- Assume priority = key.

	INJECT(e)	EJECT()
Unsorted array or list		
Sorted array or list		
Heap	$O(\log n)$	$O(\log n)$

- How is this accomplished?

The Heap

- A **heap** is a complete binary tree which satisfies the **heap condition**:

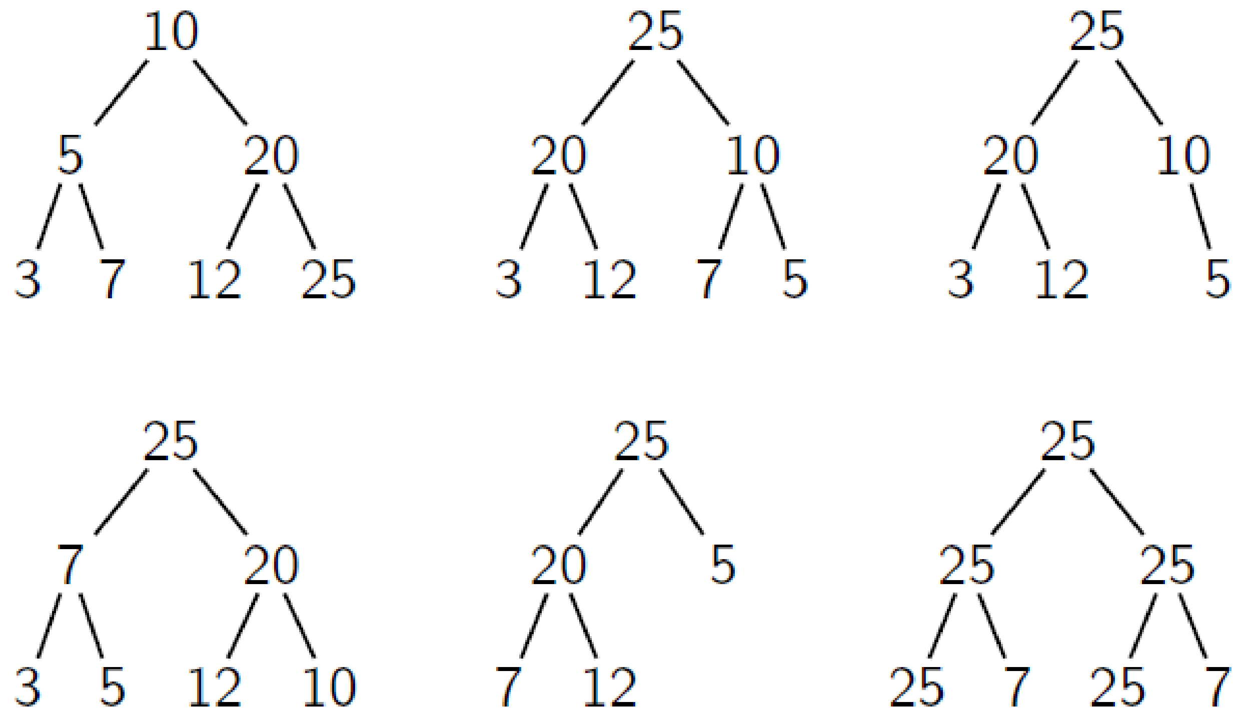
  Each child has a priority (key) which is no greater than its parent's.

- This guarantees that the root of the tree is a maximal element.

- (Sometimes we talk about this as a **max-heap** – one can equally well have min-heaps, in which each child is no smaller than its parent.)

Heaps and Non-Heaps

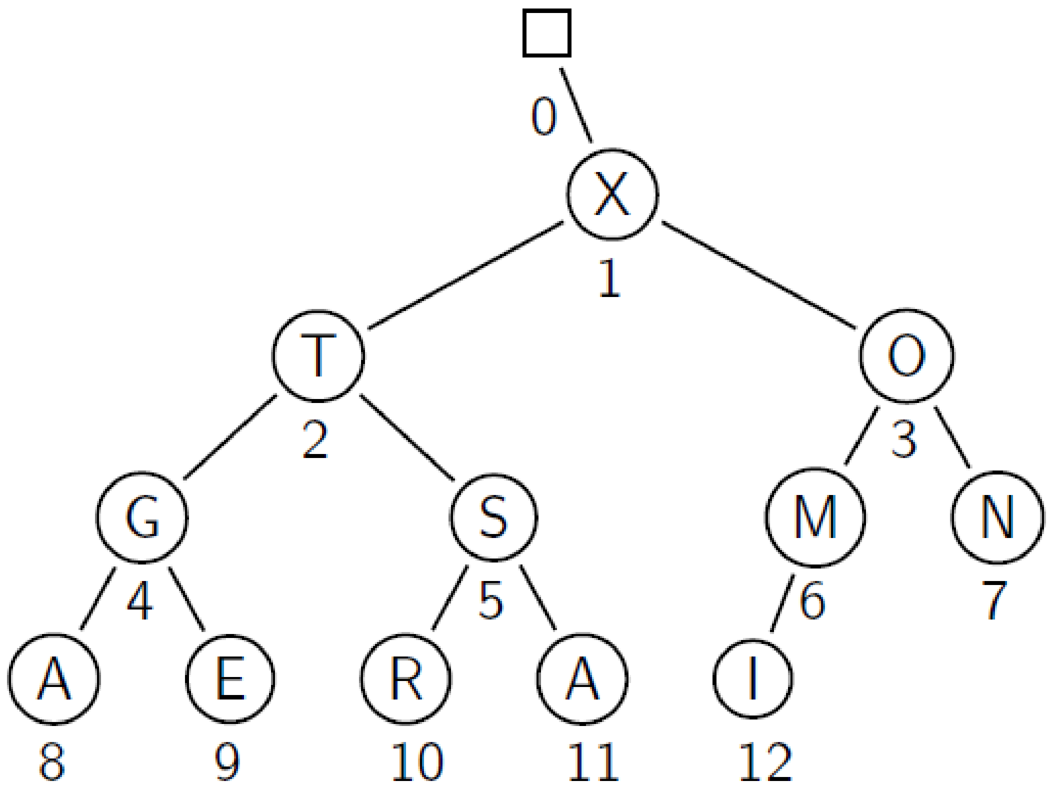
- Which of these are heaps?



Heaps as Arrays

- We can utilise the completeness of the tree and place its elements in level-order in an array *H*.

- Note that the children of node i will be nodes $2i$ and $2i + 1$.



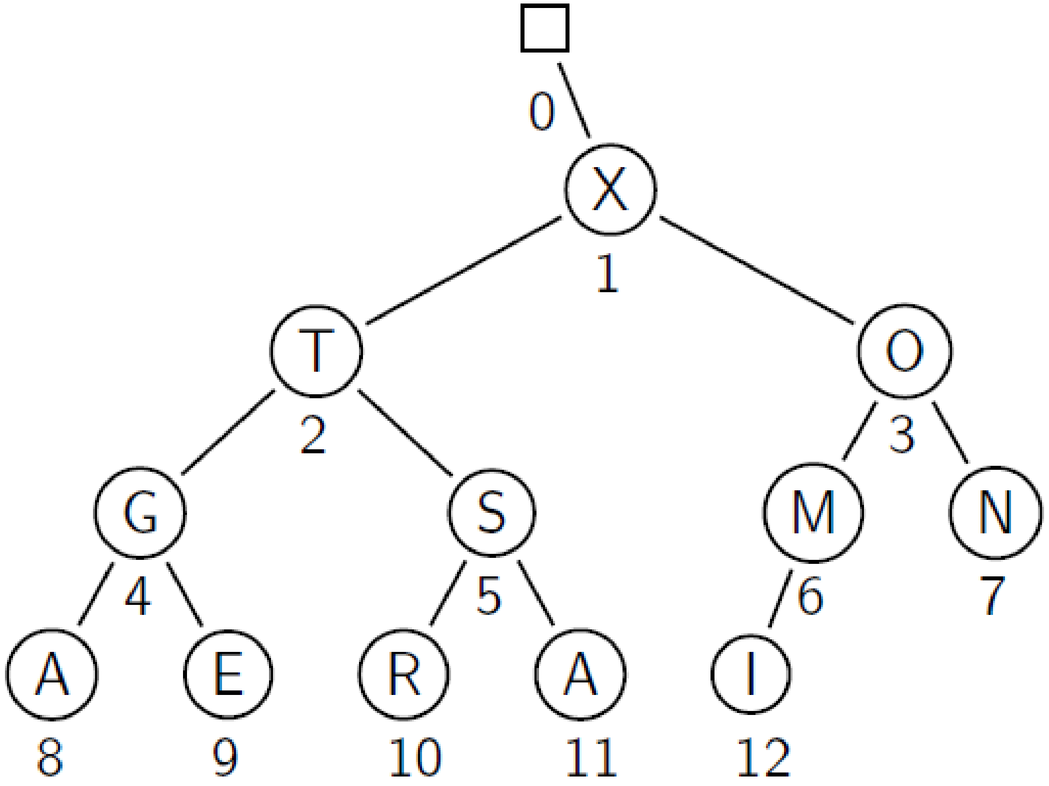
H:

0	1	2	3	4	5	6	7	8	9	10	11	12
	X	T	O	G	S	M	N	A	E	R	A	I

Heaps as Arrays

- This way, the heap condition is very simple:

- For all $i \subset \{0,1,\ldots,n\}$, we must have **H[i] ≤ H[i/2]**.



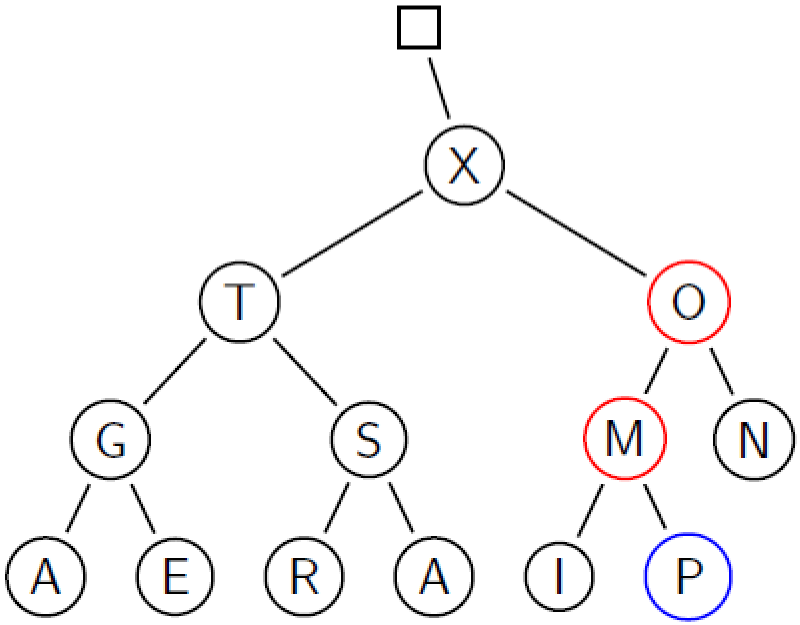
H:													
		X	T	O	G	S	M	N	A	E	R	A	I
	0	1	2	3	4	5	6	7	8	9	10	11	12

Properties of the Heap

- The root of the tree $H[1]$ holds a maximal item; the cost of EJECT is $O(1)$ plus time to restore the heap.
- The height of the heap is $\lfloor \log_2 n \rfloor$.
- Each subtree is also a heap.
- The children of node i are $2i$ and $2i+1$.
- The nodes which happen to be parents are in array positions 1 to $\lfloor n/2 \rfloor$.
- It is easier to understand the heap operations if we think of the heap as a tree.

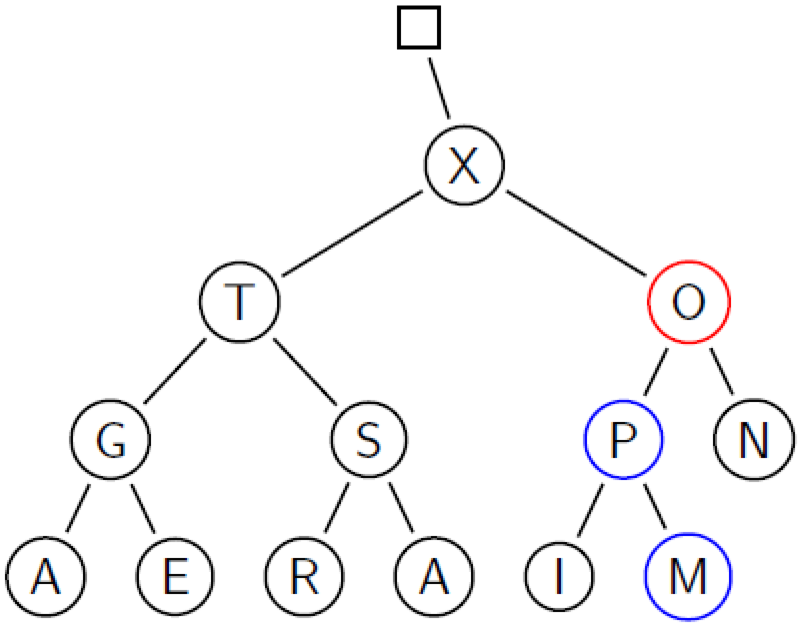
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller:



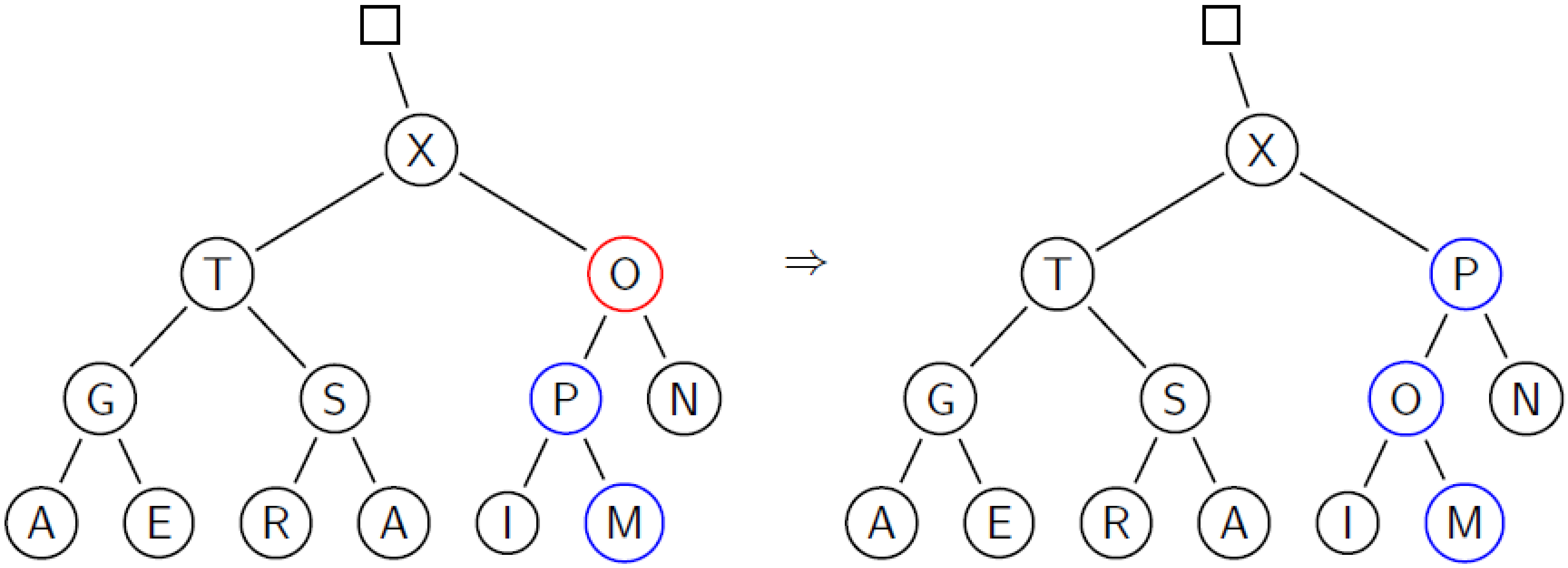
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller:



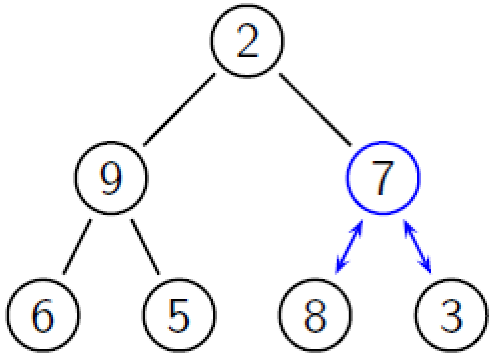
Injecting a New Item

- Place the new item at the end; then let it "climb up", repeatedly swapping with parents that are smaller:



Building a Heap Bottom-Up

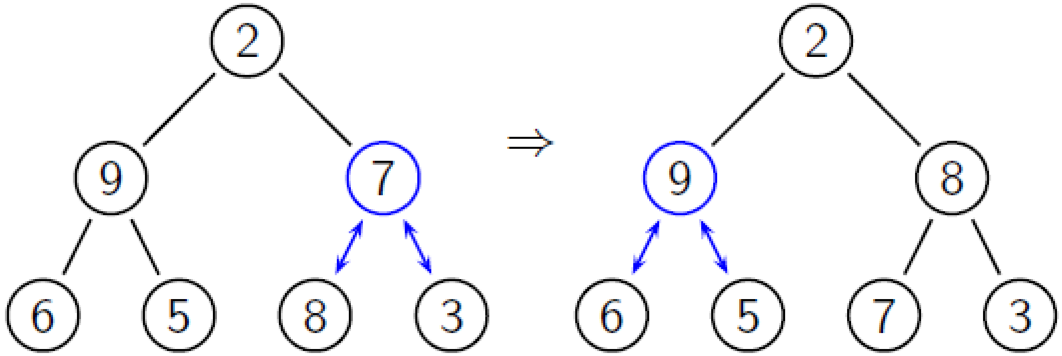
- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. The construction cost will be *n* log *n*. But there is a better way:



- Start with the last parent and move backwards, in level-order. For each parent node, if the largest child is larger than the parent, swap it with the parent.

Building a Heap Bottom-Up

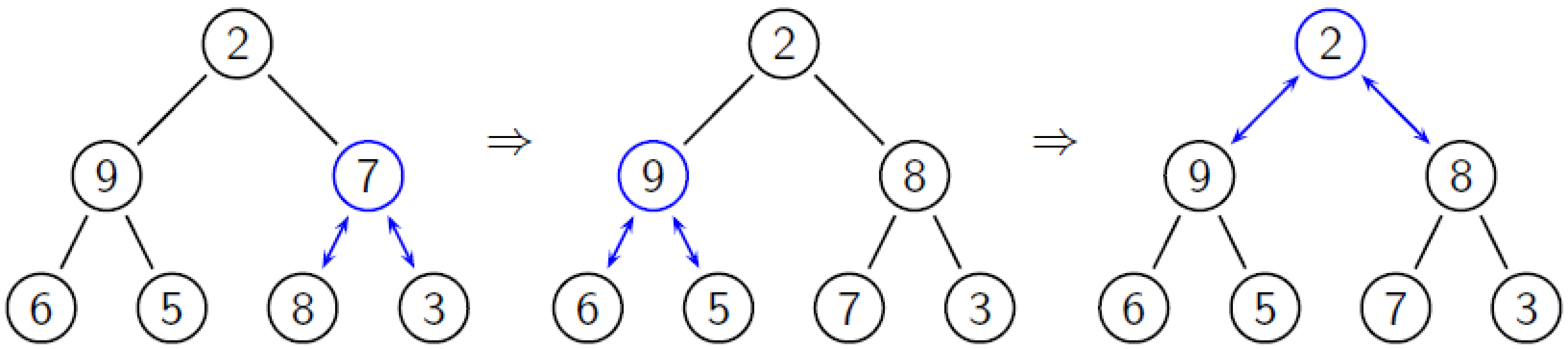
- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. The construction cost will be $n \log n$. But there is a better way:



- Start with the last parent and move backwards, in level-order. For each parent node, if the largest child is larger than the parent, swap it with the parent.

Building a Heap Bottom-Up

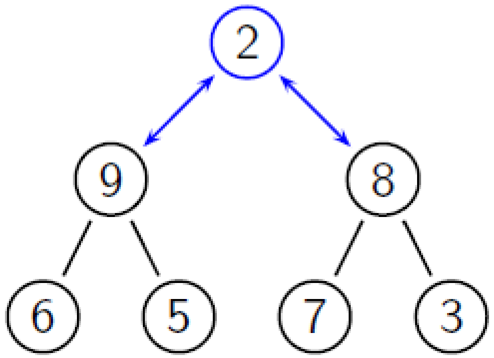
- To construct a heap from an arbitrary set of elements, we can just use the inject operation repeatedly. The construction cost will be *n* log *n*. But there is a better way:



- Start with the last parent and move backwards, in level-order. For each parent node, if the largest child is larger than the parent, swap it with the parent.

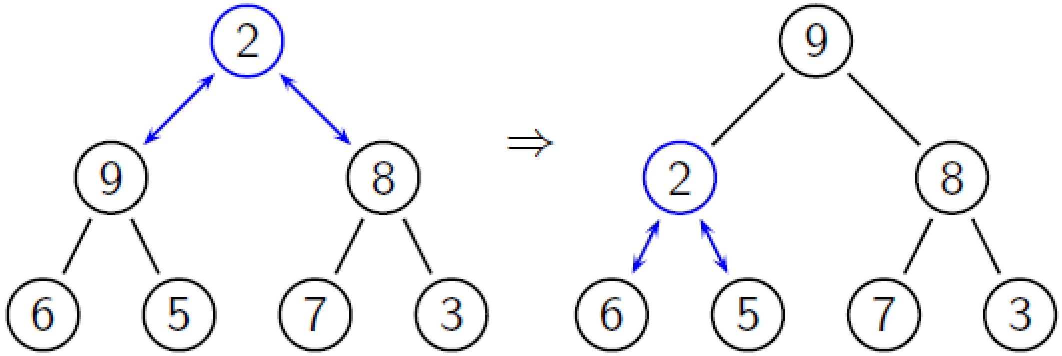
Building a Heap Bottom-Up: Sifting Down

- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



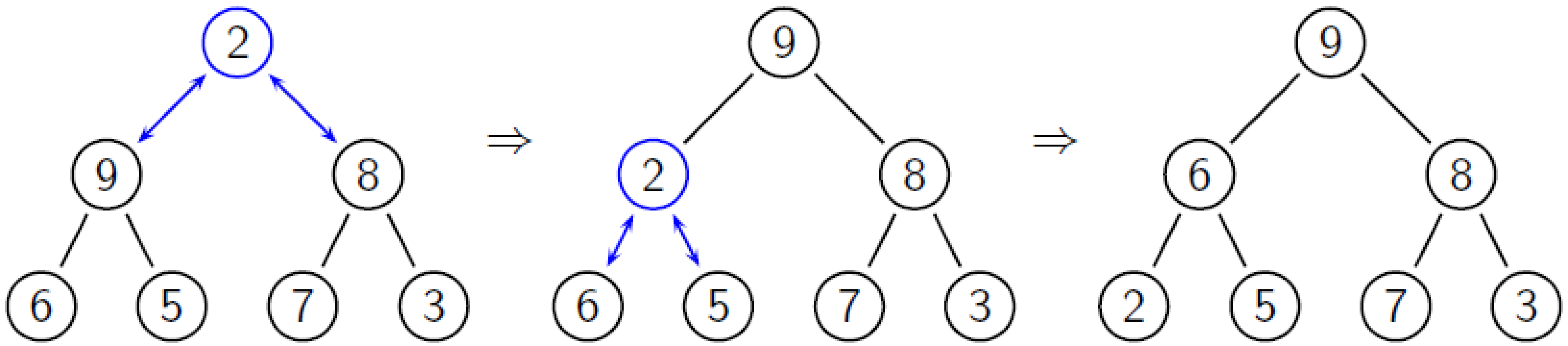
Building a Heap Bottom-Up: Sifting Down

- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



Building a Heap Bottom-Up: Sifting Down

- Whenever a parent is found to be out of order, let it "sift down" until both children are smaller:



Turning $H[1]$ … $H[n]$ into a Heap, Bottom-Up

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i
    v ← H[k]
    heap ← False
    while not heap and 2 × k ≤ n do
        j ← 2 × k                                  ▷ j is k's left child
        if j < n then
            if H[j] < H[j + 1] then
                j ← j + 1                          ▷ j is k's largest child
        if v ≥ H[j] then
            heap ← True
        else                                       ▷ Promote H[j]
            H[k] ← H[j]
            k ← j
    H[k] ← v
```

Analysis of Bottom-Up Heap Creation

- For simplicity, assume the heap is a full binary tree: $n = 2^{h+1} - 1$. Here is an upper bound on the number of "down-sifts" needed (consider the root to be at level \$h\$, so leaves are at level 0):

$$\sum_{i=1}^{h} \sum_{\text{nodes at level } i} i \quad = \quad \sum_{i=1}^{h} i \cdot 2^{h-i} \quad = \quad 2^{h+1} - h - 2$$

- The last equation is easily proved by mathematical induction.

- Note that $2^{h+1} - h - 2 < n$, so we perform at most a linear number of down-sift operations. Each down-sift is preceded by two key comparisons, so the number of comparisons is also linear.

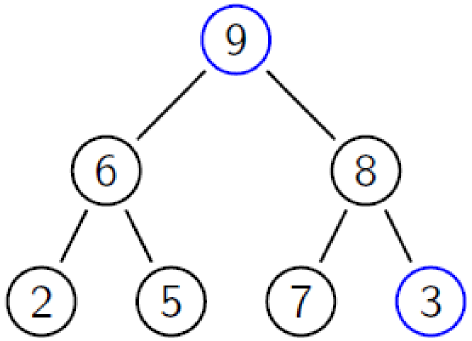
- Hence we have a **linear-time** algorithm for heap creation.

Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item *z* in the heap, and then let *z* "sift down" to its proper place.

- After this, the last element (here shown in green) is no longer considered part of the heap, that is, *n* is decremented.

- Clearly ejection is $O(\log n)$.

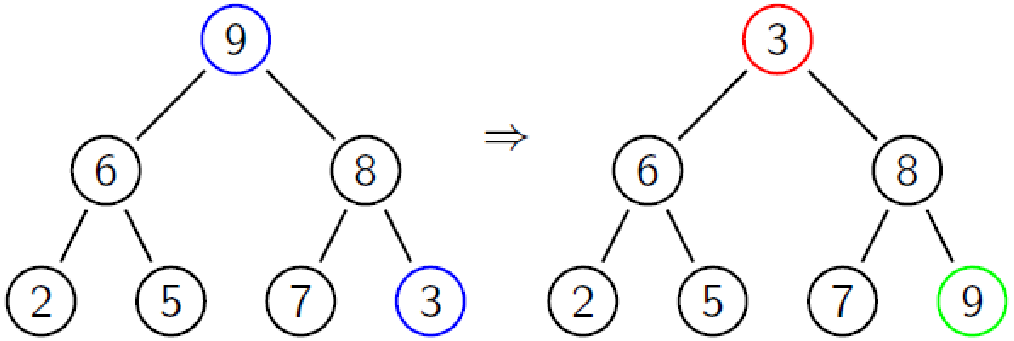


Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item *z* in the heap, and then let *z* "sift down" to its proper place.

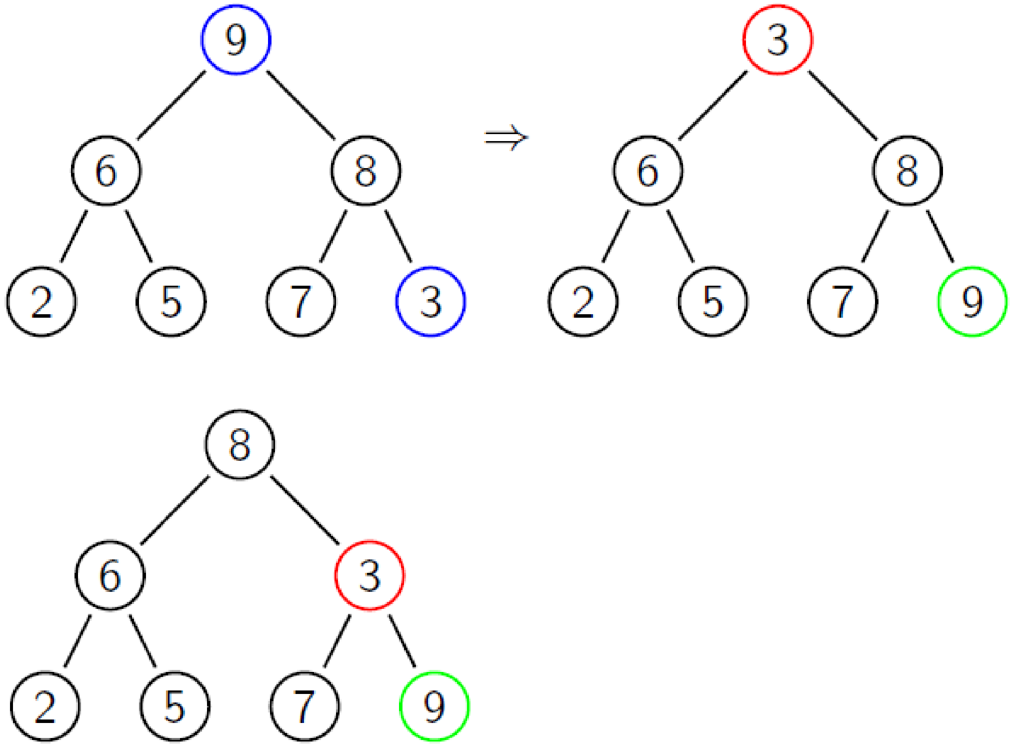
- After this, the last element (here shown in green) is no longer considered part of the heap, that is, *n* is decremented.

- Clearly ejection is $O(\log n)$.



Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item *z* in the heap, and then let *z* "sift down" to its proper place.
- After this, the last element (here shown in green) is no longer considered part of the heap, that is, *n* is decremented.
- Clearly ejection is $O(\log n)$.

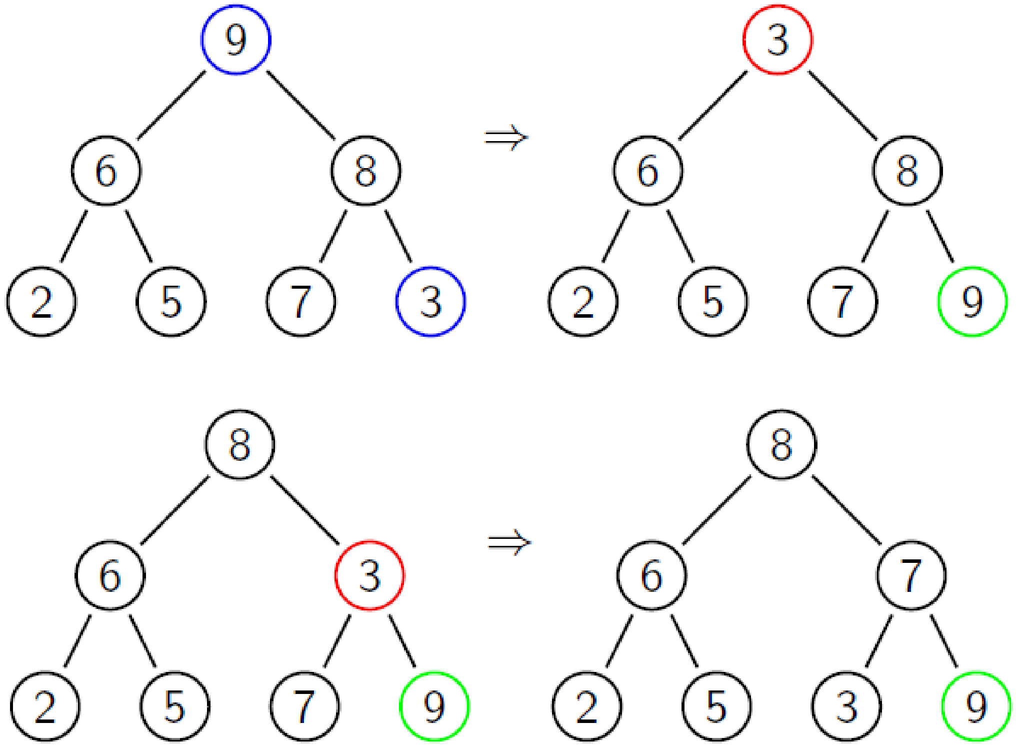


Ejecting a Maximal Element from a Heap

- Here the idea is to swap the root with the last item *z* in the heap, and then let *z* "sift down" to its proper place.

- After this, the last element (here shown in green) is no longer considered part of the heap, that is, *n* is decremented.

- Clearly ejection is $O(\log n)$.



Exercise: Build and Then Deplete a Heap

- First build a heap from the items S, O, R, T, I, N, G.

- Then repeatedly eject the largest, placing it at the end of the heap.

Exercise: Build and Then Deplete a Heap

- First build a heap from the items S, O, R, T, I, N, G.
- Then repeatedly eject the largest, placing it at the end of the heap.
- Anything interesting to notice about the tree that used to hold a heap?

Heapsort

- Heapsort is a Θ(n log n) sorting algorithm, based on the idea from this exercise.

- Given an unsorted array $H[1] \ldots H[n]$:

- **Step 1:** Turn *H* into a heap.
- **Step 2:** Apply the eject operation *n*-1 times.

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

Stage 2 (maximum deletions)

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

Stage 2 (maximum deletions)

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

Stage 2 (maximum deletions)

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

Stage 2 (maximum deletions)

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8
2 **9** 8 6 5 7
2 9 8 6 5 7
9 **2** 8 6 5 7
9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 $\underline{8}$

2 **9** 8 $\underline{6\ \ 5}$ 7

2 $\underline{9\ \ 8}$ 6 5 7

9 **2** 8 $\underline{6\ \ 5}$ 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 $\underline{7}$

7 6 8 2 5 | **9**

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8
2 **9** 8 6 5 7
2 9 8 6 5 7
9 **2** 8 6 5 7
9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7
7 6 8 2 5 | **9**
8 6 7 2 5 | **9**

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5 | **9**

5 6 7 2 | **8** **9**

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5 | **9**

5 6 7 2 | **8** **9**

7 6 5 2 | **8** **9**

2 6 5 | **7** **8** **9**

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5 | **9**

5 6 7 2 | **8** **9**

7 6 5 2 | **8** **9**

2 6 5 | **7** **8** **9**

6 2 5 | **7** **8** **9**

5 2 | **6** **7** **8** **9**

Heapsort

Stage 1 (heap construction)

2 9 **7** 6 5 8

2 **9** 8 6 5 7

2 9 8 6 5 7

9 **2** 8 6 5 7

9 6 8 2 5 7

Stage 2 (maximum deletions)

9 6 8 2 5 7

7 6 8 2 5 | **9**

8 6 7 2 5 | **9**

5 6 7 2 | **8** **9**

7 6 5 2 | **8** **9**

2 6 5 | **7** **8** **9**

6 2 5 | **7** **8** **9**

5 2 | **6** **7** **8** **9**

5 2 | **6** **7** **8** **9**

2 | **5** **6** **7** **8** **9**

2 **5** **6** **7** **8** **9**

Properties of Heapsort

- On average slower than quicksort, but stronger performance guarantee.
- Truly in-place.
- Not stable.

Next lecture

- Transform-and-Conquer
  - Pre-sorting (Levitin Section 6.1)