Lecture 2: Priority Queues, Heaps, and Heapsort
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
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:

\[
\frac{100 + 110 + 111}{3} = 107 \text{ time units}
\]
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:

\[
\frac{(100+110+111)}{3} = 107 \text{ time units}
\]

Average finish time for shortest-job-first service:

\[
\frac{(1+11+111)}{3} = 41 \text{ time units}
\]
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.
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.

A queue capable of supporting two operations: Insert and Extract-Min?
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 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

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

Question

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

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

Heap-order property:

- A min-heap
- Not a heap
If the heap-order property is maintained, heaps support the following operations efficiently (assume there are \( n \) elements in the heap):

- **Insert** in \( O(\log n) \) time
- **Extract-Min** in \( O(\log n) \) time
If the heap-order property is maintained, heaps support the following operations efficiently (assume there are $n$ elements in the heap)

- **Insert** in $O(\log n)$ time
- **Extract-Min** in $O(\log n)$ time

**Structure properties**

- A heap of height $h$ has between $2^h$ to $2^{h+1} - 1$ nodes. Thus, an $n$-element heap has height $\Theta(\log n)$. 
If the heap-order property is maintained, heaps support the following operations efficiently (assume there are $n$ elements in the heap)

- **Insert** in $O(\log n)$ time
- **Extract-Min** in $O(\log n)$ time

**Structure properties**

- A heap of height $h$ has between $2^h$ to $2^{h+1} - 1$ nodes. Thus, an $n$-element heap has height $\Theta(\log n)$.
- The structure is so regular, it can be represented in an array and no links are necessary !!!
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 $\lfloor i/2 \rfloor$

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.

```
2
3 4
5 4 6 5
7 6 8
```
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.

```
  2
 /   \
3     4
 /  \
5  6  5
 / \
6 7  8 Insert 1
```

Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element

Time complexity = \( O(\log n) \)
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.

![Diagram of a min-heap with nodes 1, 2, 3, 4, 5, 6, 7, 8, and arrows indicating movement during insertion and percolation.]

Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element.

Time complexity = $O(\log n)$
- 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.

```
2
1 4
5 3 6 5
7 6 8 4
swap
```

Percolate up to maintain the min-heap property
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.

Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element
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.

![Diagram of a min-heap](image)

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

Min-heap property preserved, but completeness not preserved!

Lecture 2: Priority Queues, Heaps, and Heapsort

Sorting and Searching

15 / 24
Min-heap property preserved, but completeness not preserved!
Extract-Min

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

```
2
3 4
4 5 6 5
76 8
```

Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children).

Time complexity = \(O(\text{height}) = O(\log n)\)
- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.

Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children).

Time complexity = $O(\log n)$
Copy the last element to the root (i.e., overwrite the minimum element stored there)

Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.

Percolate down to maintain the min-heap property
 Extract-Min

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

Percolate down to maintain the min-heap property

Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)

Time complexity = \(O(\log n)\)
Extract-Min

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

```
3
4 4
6 5 6 5
7 8
```

Percolate down to maintain the min-heap property

Swap

- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)
Extract-Min

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

Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)
- Time complexity = $O(\text{height}) = O(\log n)$
Build a binary heap of $n$ elements
- the minimum element is at the top of the heap
Heapsort

- Build a binary heap of $n$ elements
  - the minimum element is at the top of the heap

- Perform $n$ Extract-Min operations
  - the elements are extracted in sorted order
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
Build a binary heap of $n$ elements
- the minimum element is at the top of the heap
- insert $n$ elements one by one
  $\implies O(n \log n)$
  (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
  $\implies O(n \log n)$
Build a binary heap of $n$ elements
- the minimum element is at the top of the heap
- insert $n$ elements one by one
  $\implies O(n \log n)$
  (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
  $\implies O(n \log n)$

Total time complexity: $O(n \log n)$
A **Priority queue** is an abstract data structure that supports two operations: **Insert** and **Extract-Min**.
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.
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.
Sometimes priority queues need to support another operation called **Decrease-Key**

- **Decrease-Key**: decreases the value of one specified element
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
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?
For some algorithms, there are other desirable Priority Queue operations, e.g., *Delete* an arbitrary item and *Meld*ing 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*. 