# Priority queues

# Priority queues?

A priority queue is a data structure.

# Data structures

$\boxed{\text{Data structure = data + operations on it}}$

Data:

- Normally structured as an ID plus additional data. The ID is also called a key.
- We usually do not mention the additional data. That is, elements are referred to simply as ID, but they are actually (ID, data) or (ID, reference to data).
- The ID is often from an ordered universe (has an order), e.g., int, float, String.

Operations:

- The properties of the data structure are defined by the provided operations and their runtime performance.
- The goals are flexibility and efficiency (which are usually conflicting objectives).

# Priority queues

## Data:

- **Element** = key (ID) from an ordered universe, possibly with additional data.

## Core operations (max-version of priority queue):

- **Q.Extract-Max**: Returns the element with the largest key in the priority queue Q (an arbitrary such element if multiple have the same key). The element is removed from Q.
- **Q.Insert(e: element)**: Adds the element *e* to the priority queue Q.

**Note:** We can sort using these operations.

n×Insert

n×Extract-Max

# Priority queues

## Additional operations:

- **Q.Increase-Key(r: reference to an element in Q, k key)**: Changes the key to max{k, old key} for the element referenced by *r*.
- **Q.Build(L: list of elements)**: Builds a priority queue containing the elements in list *L*.

## Trivial operations for all data structures:

- **Q.CreateNewEmpty(), Q.RemoveEmpty(), Q.IsEmpty()?**

(These will not be mentioned going forward.)

## Implementation via heaps

A possible implementation: Use the heap structure from Heapsort.

[Note: The array version of heaps requires a known maximum size n of the queue. Alternatively, the array can be replaced by an extendible array, such as java.util.ArrayList in Java or lists in Python. The heap tree can also be implemented using pointers/references.]

We already have:

- **Extract-Max:** Essentially the same as the second phase of Heapsort – remove the root, move the last leaf up as the new root, and call Heapify.
  **Runtime:** O(log n).

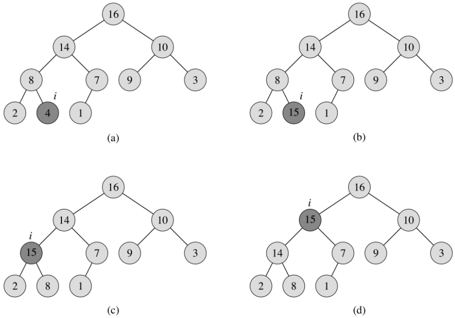
- **Build:** Use **Heapify** repeatedly in a bottom-up manner.
  **Runtime:** O(n).

Missing:

- Insert
- Increase-Key

# Increase Key

1. Change the key for the element.
2. Restore heap order: as long as the element is greater than its parent, swap places with it.




**Runtime:** The height of the tree, i.e.O(log n).

# Insert

1. **Insert** the new element at the end ($\Rightarrow$ heap property is maintained).
2. Restore heap order exactly as in **Increase-Key**: as long as the element is greater than its parent, swap places with it.

Runtime: The height of the tree, i.e., O(log n).

# Different implementations of priority queues

| | *Heap* | Unsorted list | | Sorted list | |
|---|---|---|---|---|---|
| EXTRACT-MAX | $O(\log n)$ | $O(n)$ | go through the whole list | $O(1)$ | take the first |
| BUILD | $O(n)$ | $O(1)$ | | $O(n \log n)$ | sort using fx Quicksort |
| INCREASE-KEY | $O(\log n)$ | $O(1)$ | | $O(n)$ | |
| INSERT | $O(\log n)$ | $O(1)$ | | $O(n)$ | |

The above operations are for max-priority queues. It is, of course, easy to create min-priority queues with the operations Extract-Min, Build, Decrease-Key, and Insert, simply by reversing all inequalities between keys in the definitions and algorithms.