Priority Queues

A **priority queue** is a data structure.

> A **data structure** consists of data and the operations performed on it. The data is usually structured with an ID (also known as a key) and additional data. The ID often comes from an ordered universe, such as int, float, or String.

The properties of a data structure are defined by the operations it offers and their runtimes. The goals are flexibility and efficiency, which are often conflicting.

Think of a data structure as an **API** for accessing a collection of data.

- **Level 1:** The offered operations (in Java: an interface).
- **Level 2:** A concrete implementation of the offered operations (in Java: a class that implements the interface). A given set of operations (level 1) can have many different implementations (level 2), often with different runtimes. This course catalogs data structures (level 1) with broad applications and efficient implementations (level 2).

Priority Queues, Level 1

Data: Element = key (ID) from an ordered universe, plus additional data.

Central 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 there are multiple equal ones). 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` followed by `n` × `Extract-Max`.

Extra 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 the list `L`.

Trivial operations for all data structures (will not be mentioned further):

- `Q.CreateNewEmpty()`
- `Q.DestructEmpty()`
- `Q.IsEmpty?()`

Implementation via Heaps

One possible implementation (level 2) is using the heap structure from Heapsort. We already have:

- `Extract-Max`: Essentially what is used during the second part of Heapsort – remove root, move last leaf up as root, call `Heapify`. Runtime: $O(\log n)$.
- `Build`: Use `Heapify` repeatedly bottom-up. Runtime: $O(n)$.

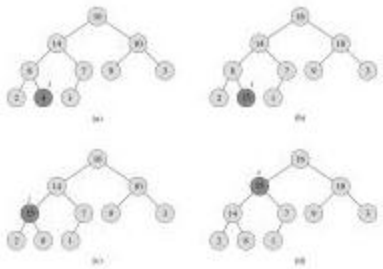
Missing: `Insert` and `Increase-Key`.

Note: In Java, the array version of heaps requires a known maximum for the size of the queue. Alternatively, the array can be replaced by an extendible array, e.g., `java.util.ArrayList` in Java or lists in Python. One can also implement the heap tree via pointers/references, as we do with search trees later.

Increase-Key

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

Here's a visual example of how `Increase-Key` works:



The image above shows a binary tree undergoing the `Increase-Key` operation. It illustrates how the value of a node is increased and then percolated upwards by swapping with its parent until the heap property is restored.

Insert

1. Insert the new element last (⇒ heap shape in order).
2. Restore heap order exactly as in `Increase-Key`: as long as element is larger than parent, swap positions with this.

Runtime: Height of the tree, i.e., $O(\log n)$.

Different Implementations of Priority Queues

Same level 1, different level 2:

Operation	Heap	Unsorted List	Sorted List
Extract-Max	$O(\log n)$	$O(n)$	$O(1)$
Build	$O(n)$	$O(1)$	$O(n \log n)$
Increase-Key	$O(\log n)$	$O(1)$	$O(n)$
Insert	$O(\log n)$	$O(1)$	$O(n)$

The above collection of operations (level 1) is for max-priority queues. It is naturally easy to make min-priority queues with the operations `Extract-Min`, `Build`, `Decrease-Key`, `Insert`, simply by reversing all inequalities between keys in definitions and algorithms.