**Motivation**

A queue is an ADT (abstract data type) that stores items first-in first-out. When is this not the case?

- The president enters a bank. You'd probably serve him first.
- Immigration lines. Citizens and folks with special cards get priority.
- Flights at an airport. Air Force One would probably jump the queue.
- Immigration lines. Citizens and folks with special cards get priority.

We'd like a data type that allows us to repeatedly remove the highest priority element, where priority doesn't have to be the same as "age in the queue".

A BST would work, but it has more overhead than we need, because it allows us access to arbitrary items, whereas we'll only need access to the highest priority item, where priority doesn't have to be the same as "age in the queue".

A queue no longer works because it's not general enough.

Our goal will be to get O(1) access to the max priority element.

A priority queue stores a collection of comparable elements such that the higher priority element is 'at the front'. Operations are:

- insert(element) aka offer --- add element to the queue
- removeMin() aka poll --- remove the highest priority element
- findMin() aka peek --- return but do not remove the highest priority element
- size() --- return the number of elements in the priority queue
- isEmpty() --- return whether the priority queue is empty
- traverse() / iterate() --- iterate over all elements in the priority queue

We want all of these operations to be O(1) except for insert and removeMin, which we want in O(log n) time.
Implementing PQ's

A priority queue stores a collection of comparable elements such that the highest priority element is "at the front". Operations are:
- `insert( element )` aka `offer` --- add element to the queue
- `removeMin()` aka `poll` --- remove the highest priority element
- `findMin()` aka `peek` --- return but do not remove the highest priority element
- `size()`
- `isEmpty()`
- `traverse()` / `iterate()`

**Idea #1:** Implement PQ using an unordered array as backing storage

Then what would be the running time of the operations be?
- `insert( element )` --- $O(1)$ (just add it to the end)
- `removeMin()` --- $O(n)$ (takes $O(n)$ time to find min in unordered list)
- `findMin()` --- $O(n)$ (takes $O(n)$ time to find min in unordered list)
- `size()` --- $O(1)$
- `isEmpty()` --- $O(1)$
- `traverse()` / `iterate()` --- $O(n)$ if do not care about order, longer if you do.

**Idea #2:** Implement PQ using an ordered array / LL as backing storage

Then what would be the running time of the operations be?
- `insert( element )` --- $O(n)$ (to find the correct location for the element)
- `removeMin()` --- $O(n)$ if an array due to shifting, $O(1)$ if LL
- `findMin()` --- $O(1)$ (it's the front element)
- `size()` --- $O(1)$
- `isEmpty()` --- $O(1)$
- `traverse()` / `iterate()` --- $O(n)$

**Idea #3:** Implement PQ using a balanced binary search tree as backing storage

Then what would be the running time of the operations be?
- `insert( element )` --- $O(\log n)$
- `removeMin()` --- $O(\log n)$
- `findMin()` --- $O(\log n)$
- `size()` --- $O(1)$
- `isEmpty()` --- $O(1)$
- `traverse()` / `iterate()` --- $O(n)$

---

**Binary Heap**

Examples:

A binary heap is an array-based data structure that combines a queue and a binary search tree.

A binary heap is a complete binary tree where each node is completely filled, and each node's key is less than or equal to the keys of its children.

A binary search tree is a data structure that combines a queue and a binary search tree.

A binary search tree is:
- a complete binary tree where each level is completely filled, and each node's key is less than or equal to the keys of its children.
- with the heap order property.

Examples:

- For all nodes n, the value/key of n is <= value/key of its children.
- It's not necessarily a complete binary tree.

**Implementation**

- Using a balanced binary search tree as backing storage.
- Using a unordered array as backing storage.

---

**Implementing PQ's**
A binary heap is an array-based data structure that combines a queue and a binary search tree. A binary heap is:

- a complete binary tree (a binary tree where each level is completely filled, except for possibly the last, which is filled from left-to-right.)
- with the heap order property:
  - for all nodes n, the value/key of n is <= value/key of its children.

The nodes of a heap are stored in an array in level-order (as produced by BFS.) Place the root node at index 1 (NOT index 0 this time!)

To insert an element into a PQ's heap, we add it as a leaf in the next open spot (which happens to be the size+1 index.) Also add the element at position 0.

Priority Queue Insertion with Heap

Priority Queue Determination with Heap

Priority Queue Determination with Heap