The Priority Queue ADT

Modelling queues where items are selected in order of priority instead of how long they've been in the queue.

Many real-world queues are not FIFO:

- Hospital waiting lists (patients with critical illnesses will be placed towards the front of the queue).
- Air traffic control (airplanes with low fuel will be landed first).
- Access nodes forwarding packets in (e.g.) LTE networks (packets from voice calls preferred over buffered video).
- Manufacturing scheduling (jobs with closest due dates are preferred).

The element with the highest priority is the one which is removed next.

The Element

Items will now be stored with two pieces of data:

- The *value* – representing the original item
- The *key* – representing its priority value

Any data type will do for the keys, as long as we can compare them. By convention, lower keys represent higher priority elements.

```
class Element:
    def __init__(self, key, value):
        self._key = key
        self._value = value

    def __eq__(self, other):
        return self._key == other._key

    def __lt__(self, other):
        return self._key < other._key
```

The `==` operator now checks if priorities are equal, and `<` checks if an item is higher priority.

Methods

- add(key, value) – add a new element into the priority queue
- min() – return the value with the minimum key
- remove_min() – remove and return the value with the minimum key
- is_empty() – return True is there are no items in the queue
- length() – return the number of elements in the queue

Implementations

There are many different ways to implement priority queues using data structures and ADTs we've seen already.

- unsorted Python list
- unsorted DLL (doubly-linked list) or SLL (singly-linked list)
- sorted Python list
- sorted DLL or SLL
- binary search tree (assuming we can extend to allow duplicate items)

    - how would we make this extension?
- avl trees (assuming the extension above)

Unsorted Python List

- `add()` – append the value to the end of the list
    - when the list is full, Python will have to find more space and copy the elements over
    - O(1) on average
- `min()` – we have to do a linear search through the list
    - O(n)
- `remove_min()` – linear search, remove, rearrange space
    - O(n)
- `is_empty()` and `length()`
    - both O(1)

`min()` and `remove_min()` are important for priority queues, so O(n) is too slow.

Doubly-Linked List

- `add()` – add item at the head or tail
    - O(1)
- `min()` – search through list
    - O(n)
- `remove_min` – search, remove (though no rearranging space needed)
    - O(n)
- `is_empty()` and `length()`
    - both O(1)

Sorted Python List

We sort the list so that the highest priority item is at the end of the list, as removing from the end of a python list is more efficient.

- `add()` – find where the new item goes, shuffle everything that's after it down the list
  - O(n)
- `min()` – grab last item
  - O(1)
- `remove_min()` – remove last item, Python may decide to rearrange space
  - O(1) on average
- `is_empty()` and `length()`
  - both O(1)

Sorted DLL

Doesn't matter whether new items are added at the start or the end.

- `add()` – have to search to find where to put the new element
  - O(n)
- `min()` – update arrows
  - O(1)
- `remove_min()` – update arrows
  - O(1)
- `is_empty()` and `length()`
  - both O(1)

AVL Tree

Add requires searching down the tree to find where the item goes. At each node, we have a data structure (e.g. a list) containing all values of a given priority. Since we don't care which item of a given priority comes out first, we'll just be taking whatever item from that datastructure.

- `add()`
  - O(log(n))
- `min()` – find the minimum by going left
  - O(log(n))
  - Alternatively we could maintain a `minimum` pointer, which would

    - still require an O(log(n)) search every time we need to update the pointer
- `remove_min()` – find node, possibly remove it, rebalance
    - O(log(n))

Have to pay the cost of balancing now though.

Representing a Binary Tree with a List

We can number all the nodes in a binary tree, and map those numbers to elements in a list.

With a left-to-right, top-to-bottom numbering scheme, the following formula works for finding the addresses of the children or parent of a node:

- left(i) = 2i + 1
- right(i) = 2i + 2
- parent(i) = (i-1)//2

So we can hop around using list indices.

However, since the list must contain every possible position for the current depth, the size of the representation is defined only by the depth – there may be wasted space.

The Heap

- We want to keep the complexity of all operations to O(log(n)) or better.
- We want to keep the complexity of `min()` to O(1), as it is just a reporting method.

So the most import issue is storing the location of the element with the minimum keep.

It looks like we need a compromise between a fully sorted structure (which gives low access time) and an unsorted structure (which gives low update time).

Binary trees offer an upper bound of O(log(n)). We'll keep the minimum key element at the root of the tree – then each of its children must have a higher key.

We'll extend that requirement to all nodes – we don't care which child is which, but both children must have higher key.

Maintaining the PQ Data

When we remove the root node, we know that we will be moving the lower key of its two children up into the root position (as the new root must have the lowest key in the tree).

- Where do we add a new node?
- How do we do the rebalancing easily?

The Binary Heap

- Every node has lower key than its children
- Every level (except the last) must be complete
- The lowest level is always filled from the left

Adding to a Binary Heap

Now when we add a new node, we add it to the leftmost empty spot in the last level, since we know something must go there. We then compare it to its parent, swap them if they're ordered incorrectly, and repeat up the tree until the new element is in the right place.

```
Create element object
Add element in last position
Bubble element up heap
Update last position
Update heap size
```

Since each swap is O(1) and the maximum number of swaps is O(log(n)) – we only have to go up one path to the root and the tree is complete – adding is

O(log(n)).

Removing from a Binary Heap

```
Extract the root value
Copy the last element into the root
Remove the last node
Bubble root element down
Update last position
Update size

Bubble element down (recursive):
    if children
        target = child with min key
        if this key > target key
         […]
```

This is O(log(n)) for the same reason as adding – we're only working our way down one path, and we will have to go log(n) steps at most.

Represent the Tree with an Array-Based List

Updating the last position might still be expensive, as we'll have to crawl up the tree and down again. To overcome this, we'll represent the tree as a list.

- Root node is at index 0.
- Next item to be added is at index size
- Last item is at index size - 1
- left(i) = 2i + 1
- right(i) = 2i + 2
- parent(i) = (i - 1)//2