Priority Queues

Table of contents

Introduction

You need to deposit a check at your bank, and so you go to the bank and there you see a single counter that says "Check Deposit". You of course pick that counter and stand at the back of the queue. You move slowly forward as the persons in front of you are serviced in a FIFO manner, until it's your turn. Since this is a FIFO queue, the earlier you arrive (your "priority"), the earlier you will receive service.

Your friend has to do the same, but at her bank, which is different than yours. Each customer gets a token with a number on it, and then waits in a nice Customer Service lounge with comfortable chairs until his or her number is called, which happens when one of the customer service agents is free. Note that there is no real queue in which people are standing, but there is an underlying queue which determintes who gets serviced next, the order being determined by the token number. The earlier you arrive, the smaller is the number on your token and the earlier you get serviced.

The differences between these two scenarios are just superficial - both systems behave exactly in the same way. In the first case, the "priority" is determined by your arrival order; in the second case, the "priority" is determined by a secondary "key" -- the token number -- which in turn depends on your arrival order. Both follow the FIFO principle. You can use a standard FIFO queue to model both the problems in exactly the same way. Also, note that once the priority is fixed, you cannot change that (let's assume that nobody else can cut in, or bribe in, while you're not looking).

Now, imagine that you want a system where the priority may depend on factors other than just the arrival time, and moreover, the priority may change at any time. In a queue of people waiting for a bus, a person with a small child infant may get higher priority and get on the bus before someone who may have gotten into the queue earlier. Or, let's say someone with much lower priority may pay a fee to "buy" higher priority, and jump ahead in line. This is actually quite different from a standard FIFO queue, in which the priority solely depends on the arrival time, and does not change until being removed from the queue. This is called a Priority Queue (PQ) because there is a priority that is not necessarily just the arrival order, and that it may change over time.

Note that a FIFO Queue is nothing but a Priority Queue with the priority or key directly proportional (or equal) to the arrival time; a LIFO Stack is also a Priority Queue with the property that the priority or key is inversely proportional to the arrival time.

As an example of what a priority queue is, imagine a sequence of integers that we're adding to a PQ, using

the value of the integer as the "key". After adding all the integers, let's remove these one by one from the PQ. There are two possibilities:

1. If a lower key represents higher priority, then the smallest integers would be removed first, followed by the smallest of the remaining ones, and so on. We will get the integers back in the non-decreasing order.
2. If a lower key represents lower priority, then the largest integers would be removed first, followed by the largest of the remaining ones, and so on. We will get the integers back in the non-increasing order.

We've just "discovered" another sorting algorithm - using a priority queue!

The interface looks just like a FIFO or LIFO, just that we explicitly specify a "priority" for each item, and we add an additional operation that allows us to change the priority of an item already in the PQ. The most basic operations are "add", "remove" and changePriority, and in addition we can have pretty much all the other operations that we had seen for Stack and Queue ADTs.

```
public interface PQueue {
    void add(Object item, Object key);
    Object remove();
    void changePriority(Object item, Object newKey);
}
```

Obviously, the "priority" Object must be Comparable, since we must be able to compare two different priorities together.

Here's an example of how to sort an array of integers using a priority queue. Assume that lower values for the key means that the object has higher priority, so will be removed earlier.

```
  /**
   * Sorts the array of integers.
   *
   * @param a the array of Integer objects to sort.
   */
  void sort(Integer[] a) {
      // Create an empty priority queue.
      PQ pq = new PQ();

      // Add all the integers in `a' to the PQ, using the integer value as
      // the key.
      for (int i = 0; i < a.length; i++)
        pq.add(a[i], a[i]);

      // Now extract the ones with the minimum key (hence highest priority)
      // from the PQ, and add it to the original array in sorted order.
      for (int i = 0; i < a.length; i++)
        a[i] = (Integer) PQ.remove();
  }
```

This will sort the array in non-decreasing order. How can you sort it in the non-increasing order? Two choices:

a. Make the priority queue understand that lower key means lower priority;

b. Sort first, and then reverse the array; or,
c. Add the Integer objects from the back.

```
for (int i = a.length-1; i >= 0; i--)
    a[i] = (Integer) PQ.remove();
```

Back to Table of contents

Implementation

The easiest implementation would use an array as the underlying container. The question is whether we would use an ordered/sorted or unordered/unsorted array (by the "key" of item) or not. Let's assume that the lower the key, the higher the priority (same as the non-decreasing order sorting example above).

Back to Table of contents

Unsorted array

If we use unsorted array, we don't care what the key is when adding the item, and simply add it to the end of the array (in the next available position in the array that is). Hence adding an item is independent of the number of items in the PQ, and takes constant time. Removing the next item however requires that we search for the lowest key value, and the remove it. Removing then requires searching at most `n` items in the PQ for the lowest key, and then may have to shift at most `n - 1` items to the left to fill up the gap left by the item removed. Even if we improved the search for the lowest key to $\log_2(n)$ using binary search, we still have to shift at most `n - 1` items to the left, so we don't gain much.

Note that removal is exactly the same as "Selection Sort"!

Worst case performance, given a PQ with `n` items in it

- **Adding**: 1 (or some other constant independent of `n`)
- **Removing**: `n` comparisons + `n - 1` shifts if we use linear search; or $\log_2(n)$ comparisons + `n - 1` shifts if we use binary search

Back to Table of contents

Sorted array

If we store the items in an array sorted by the key value (in non- increasing order), then the item with the smallest key is always in the last-used slot of the array, which removal a constant-time operation. Adding an item however takes much more time - we first have to find the proper insertion point, shift all the item from that point on right to create a gap, and then put the item in that gap. Searching for the insertion point (from the rear-end) requires at most `n` comparisons, following by at most `n` right-shifts to create a gap for the new item, and then a constant-time operation to put the item in the gap. We could modify binary search to find the insertion point in at most $\log_2(n)$ comparisons, but since we may still need at most `n` shifts, which doesn't help us much in the overall sense.

Note that adding looks is exactly the same as "Insertion Sort"!

Worst case performance, given a PQ with `n` items in it

- **Adding**: `n` comparisons + `n` shifts + 1 (or some other constant independent of `n`)
- **Removing**: 1 (or some other constant independent of `n`)

Back to Table of contents

Heap

This is the fun one! See Ch. 9 (9.1 - 9.6) for details.

Back to Table of contents