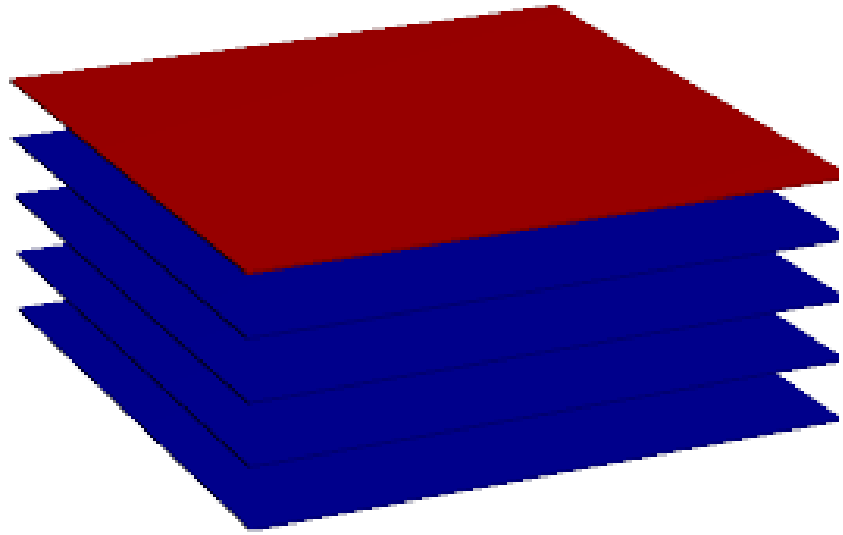
CI583: Data Structures and Operating Systems
Queues and priority queues



The queue

A queue is a collection with `insert` and `remove` methods, where `remove` returns the element that has been in the queue the longest. The methods are often called `enque` and `deque`.

This method of access is called FIFO – first in, first out.

The queue

If we implement a queue using arrays, we need to keep references to the front and back of the queue. In this way, we know where to insert new elements, and from whence to remove the oldest element.

If our implementation uses lists, we need to keep a reference to the front of the queue, because this won't be the same thing as the head of the list.

The queue

```
1  class ListItem {
2    int data;
3    ListItem next;
4  }
5  class QueueList {
6    ListItem head;
7    int length=0;
8    public void insert(int e) {
9      head = new ListItem(e, head);
10     length++;
11   }
12   //...
13 }
```

The queue

```
1  class QueueList {
2    //...
3    public int remove() {
4      ListItem front = head;
5      for(int i=0;i<length;i++) {
6        front = front.next;
7      }
8      length--;
9      return front.data;
10   }
11 }
```

The queue

What order is `remove`? Can we improve on that?

```
1  public int remove() {
2      ListItem front = head;
3      for(int i=0;i<length;i++) {
4        front = front.next;
5      }
6      length--;
7      return front.data;
8    }
```

The queue

We can, by keeping track of the head of the queue. The easiest way to do that is by using a doubly-linked list. This is a list where we can navigate to the previous element, as well as to the next.

Then, we remove an item just switching the reference to the front of the queue up by one.

The doubly-linked list

```
1  class DListItem {
2    int data;
3    DListItem previous;
4    DListItem next;
5    //...
6  }
7  class QueueList2 {
8    DListItem head;
9    DListItem front;
10   public void insert(int e) {
11     DListItem newHead = new DListItem(e, head);
12     head.setPrevious(newHead);
13     head = newHead;
14   }
15   //...
16 }
```

The doubly-linked list

Now `remove` is $O(1)$, a big saving for long queues!

```
1     //...
2     public int remove() {
3       DListItem oldFront = front;
4       front = front.previous;
5       return oldFront.data;
6     }
7   }
```

The dequeue

There are several important variations on the queue, one of which is the double-ended queue, or deque (pronounced *deck*, to distinguish it from the `dequeue` method).

We can remove from either end of a deque, so we can use them *stack-wise* or *queue-wise*.

The priority queue

The next variation on the queue is the priority queue, in which elements with the highest "priority", whatever we choose to mean by that, are at the front of the queue.

Hopefully you can see that this data structure would be useful for any sort of scheduling task, such as maintaining a queue of messages or of threads within an operating system.

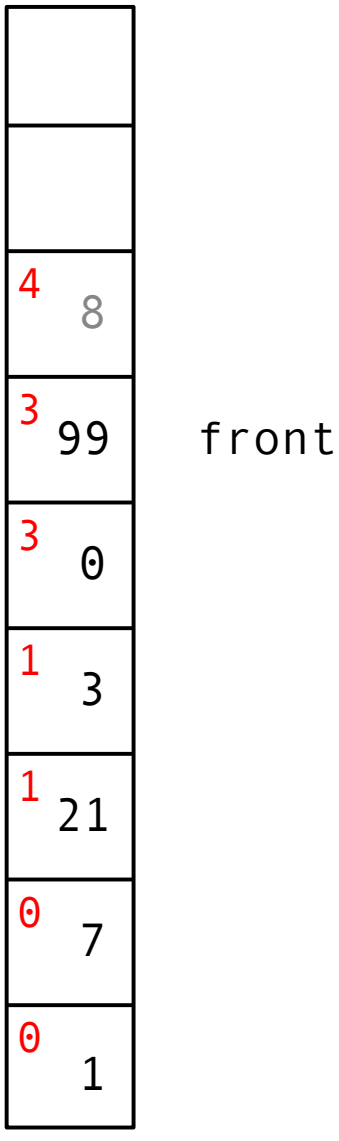
New elements need to be inserted according to their priority, so the order of the `insert` method is no longer $O(1)$.

Visualising a priority queue

Elements are ordered by priority, as well as the order in which they were added to the queue.

When storing the data in an array, we move the position of `front` as elements are added and removed.

Inserting an element is now $O(n)$ since the new element might have a lower priority than anything currently in the queue.



Higher levels of abstraction

Each of the ADTs we've seen this week is an abstraction for a particular problem, such as maintaining a list of jobs that must be completed in FIFO order.

The problems in question could be solved using the basic collection data types, but encapsulating the problem in the data type is a powerful form of abstraction. We encode the problem (e.g. the problem of managing a queue) in the data type.

In later weeks we will look at more sophisticated ADTs that encapsulate different problems. For instance, the binary search tree can be thought of as encoding the algorithm for binary search directly in the data type.

Next week

Next week we will be looking at trees and using them to implement heaps in the heapsort algorithm.