

Queues

A queue is a *linear* data structure that allows items to be added only to the rear of the queue and removed only from the front of the queue. Queues are *FIFO* structures: First-in First-out. They are used in operating systems to schedule access to resources such as a printer.

12.1 Queue Methods

The two standard modification methods are:

- `void enqueue(QueueType ob)`: insert the item at the *rear* of the queue
- `QueueType dequeue()`: delete and return the item at the *front* of the queue (sometimes called the first item).

A simple task with a queue is *echoing* the input (in the order it came): repeatedly insert into the queue, and then repeatedly dequeue.

12.2 Queue Implementation as Array

The natural approach to implementing a queue is, of course, an array. This suffers from the problem that as items are enqueued and dequeued, we reach the end of the array but are not using much of the start of the array.

The solution is to allow wrap-around: after filling the array, you start filling it from the front again (assuming these positions have been vacated). Of course, if there really are too many items in the queue, then this approach will also fail. This is sometimes called a *circular array*.

You maintain two markers for the two ends of the queue. The simplest is to maintain instance variables:

- `double data[]` stores the data
- `int count` records the number of elements currently in the queue, and `int capacity` the length of the array
- `int front` and `int rear` are such that: if `rear ≤ front`, then the queue is in positions `data[front] ... data[rear]`; otherwise it is in `data[front] ... data[capacity-1] data[0] ... data[rear]`

For example, the enqueue method is:

```

void enqueue(double elem)
{
    if (count == capacity)
        return;
    rear = (rear+1) % capacity ;
    data[rear] = elem ;
    count++;
}

```

Practice. As an exercise, provide the dequeue method.

12.3 Queue Implementation as Linked List

A conceptually simpler implementation is a linked list. Since we need to add at one end and remove at the other, we maintain *two* pointers: one to the front and one to the rear. The front will correspond to the head in a normal linked list (doing it the other way round doesn't work: why?).

12.4 Application: Simulation

There are two very standard uses of queues in programming. The first is in implementing certain searching algorithms. The second is in doing a simulation of a scenario that changes over time. So we examine the CarWash simulation (taken from Main).

The idea: we want to simulate a CarWash to gain some statistics on how service times etc. are affected by changes in customer numbers, etc. In particular, there is a single Washer and a single Line to wait in. We are interested in *how long on average it takes to serve a customer*.

We assume the customers arrive at random intervals but at a known rate. We assume the washer takes a fixed time.

So we create an artificial queue of customers. We don't care about all the details of these simulated customers: just their arrival time is enough.

for currentTime running from 0 up to end of simulation:

1. *toss coin to see if new customer arrives at currentTime;*
 if so, enqueue customer
2. *if washer timer expired, then set washer to idle*
3. *if washer idle and queue nonempty, then*
 dequeue next customer
 set washer to busy, and set timer
 update statistics

It is important to note a key approach to such simulations:

Wherever possible, you look ahead.

Thus when we “move” the Customer to the Washer, we immediately calculate what time the Washer will finish, and then update the statistics. In this case, it actually allows us to discard the Customer: the only pertinent information is that the Washer is busy.