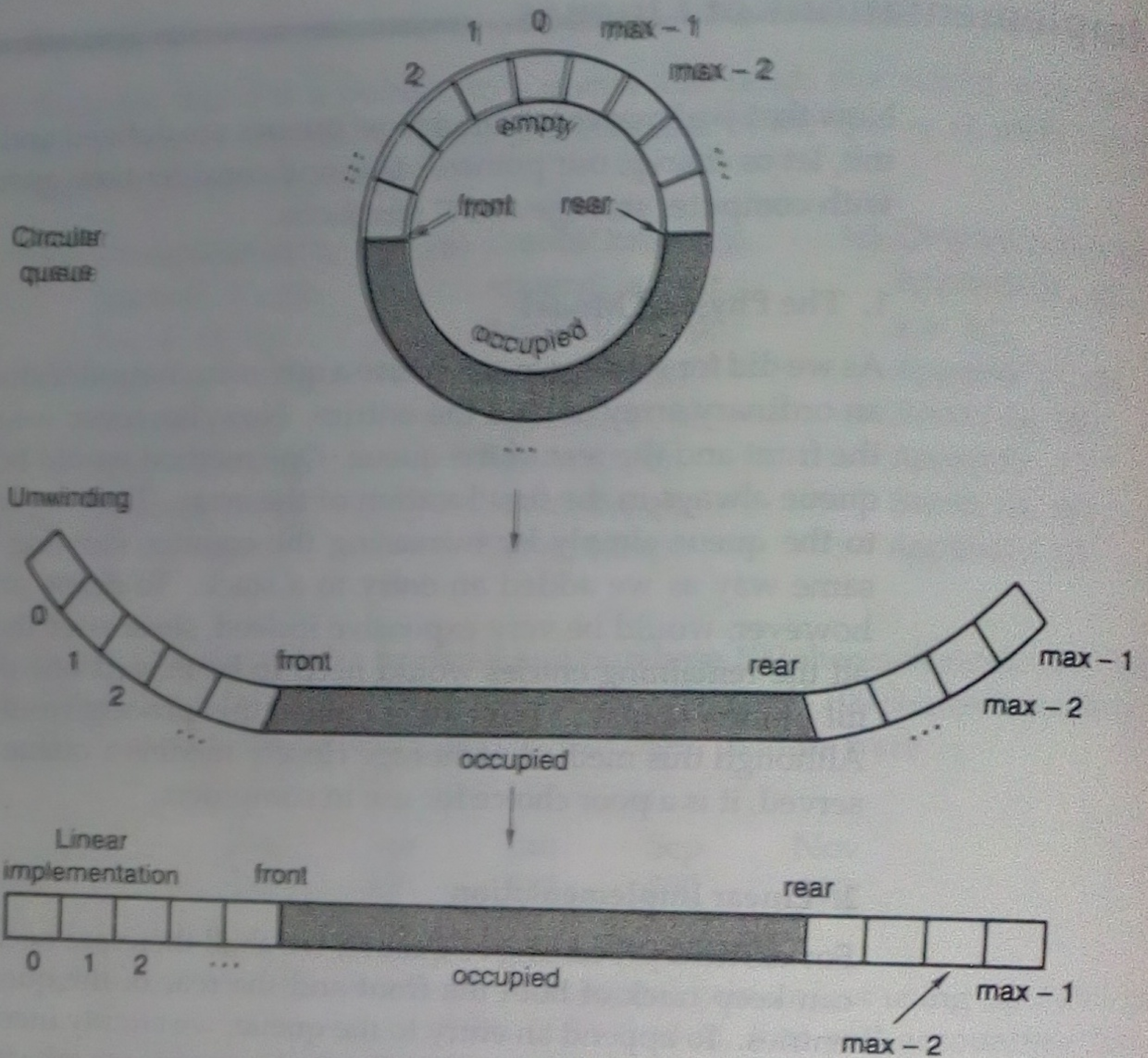

Figure 4.2. Queue in a circular array

3. Circular Arrays

In concept, we can overcome the inefficient use of space simply by thinking of the array as a circle rather than a straight line. See Figure 4.2. In this way, as entries are added and removed from the queue, the head will continually chase the tail around the array, so that the snake can keep crawling indefinitely but stay in a confined circuit. At different times, the queue will occupy different parts of the array, but we never need worry about running out of space unless the array is fully occupied, in which case we truly have overflow.

4. Implementation of Circular Arrays

Our next problem is to implement a circular array as an ordinary linear (that is, straight-line) array. To do so, we think of the positions around the circle as numbered from 0 to MAX – 1, where MAX is the total number of entries in the circular array, and to implement the circular array, we use the same-numbered entries of a linear array. Then moving the indices is just the same as doing modular arithmetic: When we increase an index past MAX – 1, we start over again at 0. This is like doing arithmetic on a circular clock face; the hours are numbered from 1 to 12, and if we add four hours to ten o'clock, we obtain two o'clock.

modular arithmetic

Perhaps a good human analogy of this linear representation is that of a priest serving communion to people kneeling at the front of a church. The communicants

do not move until the priest comes by and serves them. When the priest reaches the end of the row, he returns to the beginning and starts again, since by this time a new row of people have come forward.

5. Circular Arrays in C

In C, we can increase an index i by 1 in a circular array by writing

```
if (i >= MAX – 1)
    i = 0;
else
    i++;
```

or even more easily (but perhaps less efficiently at run time since it uses division) by using the % operator:

```
i = (i + 1) % MAX;
```

6. Boundary Conditions

Before writing formal algorithms to add to and delete from a queue, let us consider the boundary conditions, that is, the indicators that a queue is empty or full. If there is exactly one entry in the queue, then the front index will equal the rear index. When this one entry is removed, then the front will be increased by 1, so that an empty queue is indicated when the rear is one position before the front. Now suppose that the queue is nearly full. Then the rear will have moved well away from the front, all the way around the circle, and when the array is full the rear will be exactly one position before the front. Thus we have another difficulty: The front and rear indices are in exactly the same relative positions for an empty queue and for a full queue! There is no way, by looking at the indices alone, to tell a full queue from an empty one. This situation is illustrated in Figure 4.3.

empty or full?

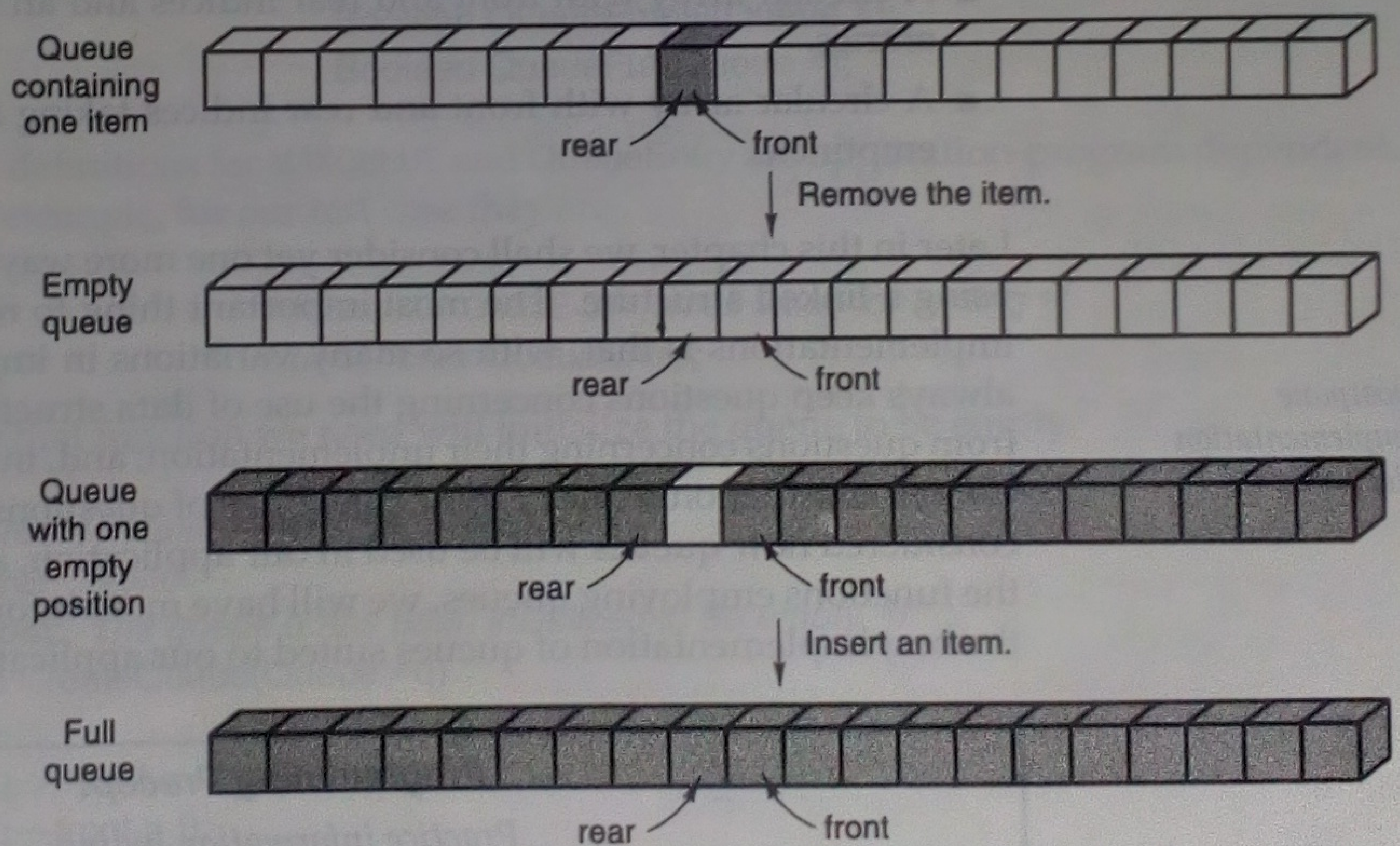

Figure 4.3. Empty and full queues

7. Possible Solutions

1. empty position

2. flag

3. special values

There are at least three essentially different ways to resolve this problem. One is to insist on leaving one empty position in the array, so that the queue is considered full when the rear index has moved within two positions of the front. A second method is to introduce a new variable. This can be a Boolean variable that will be used when the rear comes just before the front to indicate whether the queue is full or not (a Boolean variable to check emptiness would be just as good) or an integer variable that counts the number of entries in the queue. The third method is to set one or both of the indices to some value(s) that would otherwise never occur in order to indicate an empty (or full) queue. If, for example, the array entries are indexed from 0 to MAX−1, then an empty queue could be indicated by setting the rear index to −1.

8. Summary of Implementations

To summarize the discussion of queues, let us list all the methods we have discussed for implementing queues.

- The physical model: a linear array with the front always in the first position and all entries moved up the array whenever the front is deleted. This is generally a poor method for use in computers.
- A linear array with two indices always increasing. This is a good method if the queue can be emptied all at once.
- A circular array with front and rear indices and one position left vacant.
- A circular array with front and rear indices and a Boolean variable to indicate fullness (or emptiness).
- A circular array with front and rear indices and an integer variable counting entries.
- A circular array with front and rear indices taking special values to indicate emptiness.

postpone implementation decisions

Later in this chapter, we shall consider yet one more way to implement queues, by using a linked structure. The most important thing to remember from this list of implementations is that, with so many variations in implementation, we should always keep questions concerning the use of data structures like queues separate from questions concerning their implementation; and, in programming we should always consider only one of these categories of questions at a time. After we have considered how queues will be used in our application, and after we have written the functions employing queues, we will have more information to help us choose the best implementation of queues suited to our application.

Programming Precept

Practice information hiding:
Separate the application of data structures from their implementation.

4.3 Circular Queues in C

Next let us write C functions for implementation of a queue. It is clear from the last section that a great many implementations are possible, some of which are but slight variations on others. Let us therefore concentrate on only one implementation, leaving the others as exercises. The implementation in a circular array which uses a counter to keep track of the number of entries in the queue both illustrates techniques for handling circular arrays and simplifies the programming of some of the operations. Let us therefore work only with this implementation.

We shall take the queue as stored in an array indexed with the range

0 to MAXQUEUE − 1

and containing entries of a type QueueEntry. The variables front and rear will point to appropriate positions in the array. The variable count is used to keep track of the number of entries in the queue. The file queue.h contains the structure declaration for a queue and the prototypes associated with queues:

type queue

```
typedef struct queue {
    int count;
    int front;
    int rear;
    QueueEntry entry[MAXQUEUE];
} Queue;
void Append(QueueEntry, Queue *);
void CreateQueue(Queue *);
void Serve(QueueEntry *, Queue *);
int QueueSize(Queue *);
Boolean QueueEmpty(Queue *);
Boolean QueueFull(Queue *);
```

The definitions for MAXQUEUE and QueueEntry are application-program dependent. For example, for our test case they are

```
#define MAXQUEUE 3/* small value for testing */
typedef char QueueEntry;
```

The first function we need will initialize the queue to be empty.

```
/* CreateQueue: create the queue.
   Pre:  None.
   Post: The queue q has been initialized to be empty. */
void CreateQueue(Queue *q)
{
    q->count = 0;
    q->front = 0;
    q->rear = -1;
}
```