4.1 Definitions

In ordinary English, a queue is defined as a waiting line, like a line of people waiting to purchase tickets, where the first person in line is the first person served. For computer applications, we similarly define a ***queue*** to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. Queues are also called ***first-in, first-out lists***, or ***FIFO*** for short. See Figure 4.1.

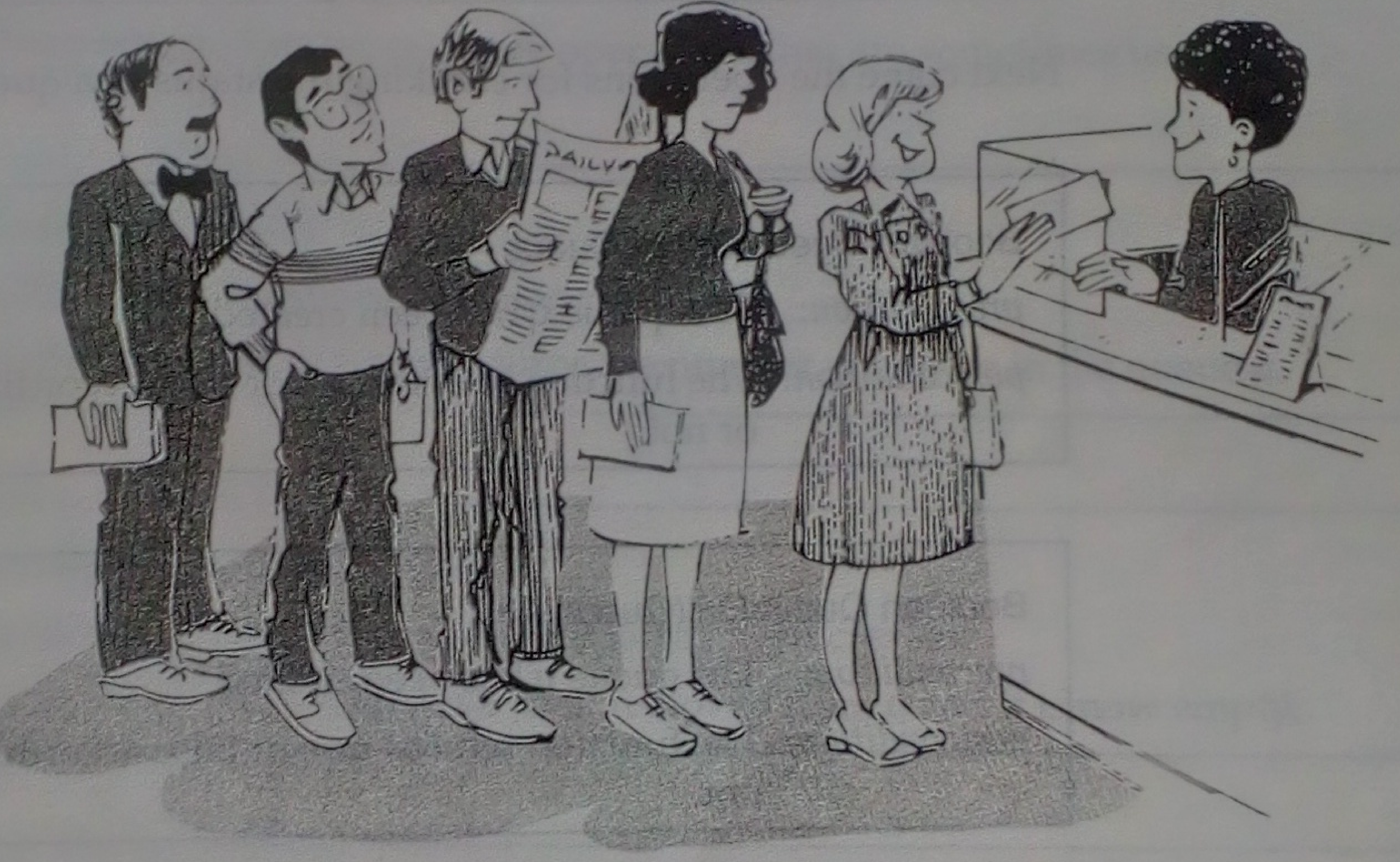

Figure 4.1. A queue

applications

Applications of queues are, if anything, even more common than are applications of stacks, since in performing tasks by computer, as in all parts of life, it is so often necessary to wait one's turn before having access to something. Within a computer system there may be queues of tasks waiting for the printer, for access to disk storage, or even, in a time-sharing system, for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue.

front and rear

The entry in a queue ready to be served, that is, the first entry that will be removed from the queue, we call the ***front*** of the queue (or, sometimes, the ***head*** of the queue). Similarly, the last entry in the queue, that is, the one most recently added, we call the ***rear*** (or the ***tail***) of the queue.

operations

To complete the definition of a queue, we must specify all the operations that it permits. We shall do so by listing the function name for each operation, together with the preconditions and postconditions that complete its specifications. As you read these specifications, you should note the similarity with the corresponding operations for a stack.

altern

The first step we must perform in working with any queue is to use the function CreateQueue to initialize it for further use:

void CreateQueue(Queue *q);

precondition: None.

postcondition: The queue q has been initialized to be empty.

Next come the operations for checking the status of a queue.

Boolean QueueEmpty(Queue *q);

precondition: The queue q has been created.

postcondition: The function returns true or false according as queue q is empty or not.

Boolean QueueFull(Queue *q);

precondition: The queue q has been created.

postcondition: The function returns true or false according as queue q is full or not.

The declarations for the fundamental operations on a queue come next.

void Append(QueueEntry x, Queue *q);

precondition: The queue q has been created and is not full.

postcondition: The entry x has been stored in the queue as its last entry.

void Serve(QueueEntry *x, Queue *q);

precondition: The queue q has been created and is not empty.

postcondition: The first entry in the queue has been removed and returned as the value of x.

The names *Append* and *Serve* are used for the fundamental operations on a queue to indicate clearly what actions are performed and to avoid confusion with the terms *alternative names*

we shall use for other data types. Other names, however, are very frequently used for these operations, terms such as *Insert* and *Delete* or the coined words *Enqueue* and *Dequeue*.

Note from the preconditions that it is an error to attempt to append an entry onto a full queue or to serve an entry from an empty queue. If we write the functions Append and Serve carefully, then they should return error messages when they are used incorrectly. The declarations, however, do not guarantee that the functions will catch the errors, and, if they do not, then they may produce spurious and unpredictable results. Hence the careful programmer should always make sure, whenever invoking a subprogram, that its preconditions are guaranteed to be satisfied.

There remain four more queue operations that are sometimes useful.

```
int QueueSize(Queue *q);
```
precondition: The queue q has been created.
postcondition: The function returns the number of entries in the queue q.

```
void ClearQueue(Queue *q);
```
precondition: The queue q has previously been created.
postcondition: All entries have been removed from q and it is now empty.

```
void QueueFront(QueueEntry *x, Queue *q);
```
precondition: The queue q has been created and is not empty.
postcondition: The variable x is a copy of the first entry in q; the queue q remains unchanged.

The final operation is not part of the strict definition of a queue, but it remains quite useful for debugging and demonstration.

```
void TraverseQueue(Queue *q, void (*Visit)(QueueEntry x));
```
precondition: The queue q has been created.
postcondition: The function Visit(QueueEntry x) has been performed for each entry in the queue, beginning with the entry at the front and proceeding toward the rear of q.

4.2 Implementations of Queues

Now that we have considered how queues are defined and the operations they admit, let us change our point of view and consider how queues can be implemented with computer storage and C functions.

1. The Physical Model

As we did for stacks, we can create a queue in computer storage easily by setting up an ordinary array to hold the entries. Now, however, we must keep track of both the front and the rear of the queue. One method would be to keep the front of the queue always in the first location of the array. Then an entry could be appended to the queue simply by increasing the counter showing the rear, in exactly the same way as we added an entry to a stack. To delete an entry from the queue, however, would be very expensive indeed, since after the first entry was served, all the remaining entries would need to be moved one position up the queue to fill in the vacancy. With a long queue, this process would be prohibitively slow. Although this method of storage closely models a queue of people waiting to be served, it is a poor choice for use in computers.

2. Linear Implementation

For efficient processing of queues, we shall therefore need two indices so that we can keep track of both the front and the rear of the queue without moving any entries. To append an entry to the queue, we simply increase the rear by one and put the entry in that position. To serve an entry, we take it from the position at the front and then increase the front by one. This method, however, still has a major defect. Both the front and rear indices are increased but never decreased. Even if there are never more than two entries in the queue, an unbounded amount of storage will be needed for the queue if the sequence of operations is

defect

```
Append, Append, Serve, Append, Serve, Append, ....
```

The problem, of course, is that, as the queue moves down the array, the storage space at the beginning of the array is discarded and never used again. Perhaps the queue can be likened to a snake crawling through storage. Sometimes the snake is longer, sometimes shorter, but if it always keeps crawling in a straight line, then it will soon reach the end of the storage space.

advantage

Note, however, that for applications where the queue is regularly emptied (such as when a series of requests is allowed to build up to a certain point, and then a task is initiated that clears all the requests before returning), at a time when the queue is empty, the front and rear can both be reset to the beginning of the array, and the simple scheme of using two indices and straight-line storage becomes a very efficient implementation.