The functions for adding to and deleting from a queue follow our preceding discussion closely. Notice that we guard against a violation of the preconditions and invoke the Error function when necessary.

```
/* Append: append an entry to the queue.
   Pre:  The queue q has been created and is not full.
   Post: The entry x has been stored in the queue as its last entry.
   Uses: QueueFull, Error. */
void Append(QueueEntry x, Queue *q)
{
    if (QueueFull(q))
        Error("Cannot append an entry to a full queue.");
    else {
        q->count++;
        q->rear = (q->rear + 1) % MAXQUEUE;
        q->entry[q->rear] = x;
    }
}
```

```
/* Serve: remove the first entry in the queue.
   Pre:  The queue q has been created and is not empty.
   Post: The first entry in the queue has been removed and returned as the value of x.
   Uses: QueueEmpty, Error. */
void Serve(QueueEntry *x, Queue *q)
{
    if (QueueEmpty(q))
        Error("Cannot serve from an empty queue.");
    else {
        q->count--;
        *x = q->entry[q->front];
        q->front = (q->front + 1) % MAXQUEUE;
    }
}
```

The three functions concerning the size of the queue are all easy to write in this implementation.

```
/* QueueSize: return the number of entries in the queue.
   Pre:  The queue q has been created.
   Post: The function returns the number of entries in the queue q. */
int QueueSize(Queue *q)
{
    return q->count;
}
```

```
/* QueueEmpty: returns non-zero if the queue is empty.
   Pre:  The queue q has been created.
   Post: The function returns non-zero if the queue q is empty, zero otherwise. */
Boolean QueueEmpty(Queue *q)
{
    return q->count <= 0;
}
```

```
/* QueueFull: returns non-zero if the queue is full.
   Pre:  The queue q has been created.
   Post: The function returns non-zero if the queue is full, zero otherwise. */
Boolean QueueFull(Queue *q)
{
    return q->count >= MAXQUEUE;
}
```

Note that the queue is specified as a pointer in each of these functions, even though it is not modified by any of them. This is a concession to efficiency that saves the time required to make a new local copy of the entire queue each time one of the functions is evaluated.

The functions ClearQueue, QueueFront, and TraverseQueue will be left as exercises.

4.6 Linked Queues

In contiguous storage, queues were significantly harder to manipulate than were stacks, and even somewhat harder than simple lists, because it was necessary to treat straight-line storage as though it were arranged in a circle, and the extreme cases of full queues and empty queues caused difficulties. It is for queues that linked storage really comes into its own. Linked queues are just as easy to handle as are linked stacks. We need only keep two pointers, front and rear, that will point, respectively, to the beginning and the end of the queue. The operations of insertion and deletion are both illustrated in Figure 4.9.

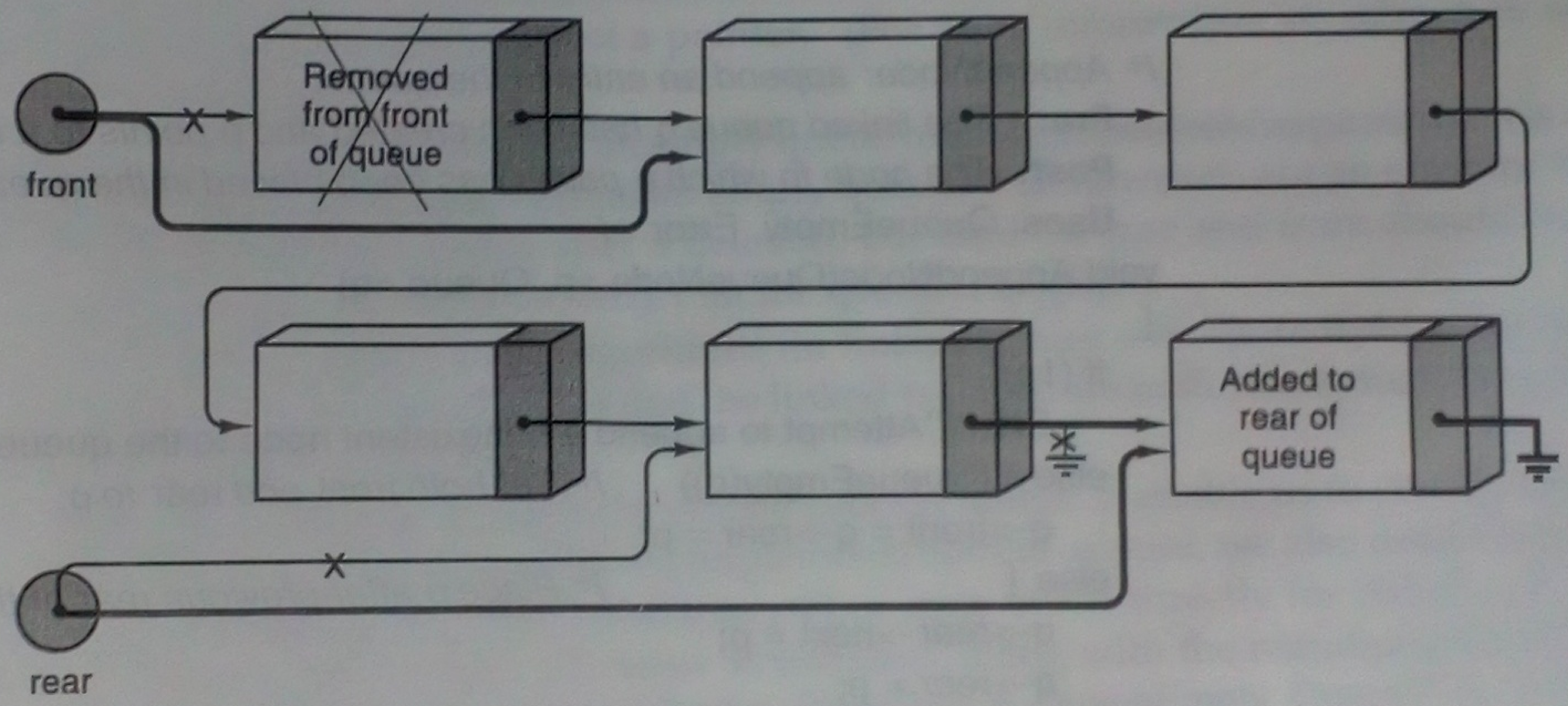

Figure 4.9. Operations on a linked queue

For all queues, we denote by queueentry the type designating the items in the queue. For linked queues, the structure of a node can then be declared as follows, in close analogy to what we have already done for stacks.

type queue

```
typedef char QueueEntry;
typedef struct queuenode {
    QueueEntry info;
    struct queuenode *next;
} QueueNode;
```

type queue

```
typedef struct queue {
    QueueNode *front;
    QueueNode *rear;
} Queue;
```

initialize

A queue should be initialized to be empty with the function:

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

Let us next turn to functions that process queue *nodes*. These will be considered private to the implementation, since the specifications for abstract queues specify only operations on entries, not nodes. First, to add a node p to the rear of a queue, we write:

```
/* AppendNode: append an entry to the queue.
   Pre:  The linked queue q has been created and p points to a node not already in q.
   Post: The node to which p points has been placed in the queue as its last entry.
   Uses: QueueEmpty, Error. */
void AppendNode(QueueNode *p, Queue *q)
{
    if (!p)
        Error("Attempt to append a nonexistent node to the queue.");
    else if (QueueEmpty(q))    /* Set both front and rear to p.                */
        q->front = q->rear = p;
    else {                     /* Place p after previous rear of the queue.    */
        q->rear->next = p;
        q->rear = p;
    }
}
```

Note that this function includes error checking to prevent the insertion of a nonexistent node into the queue. The cases when the queue is empty or not must be treated separately, since the addition of a node to an empty queue requires setting both the front and the rear to the new node, whereas addition to a nonempty queue requires changing only the rear.

To remove a node from the front of a queue, we use the following function:

```
/* ServeNode: remove the first entry in the queue.
   Pre:  The linked queue q has been created and is not empty.
   Post: The first node in the queue has been removed and parameter p points to this
         node.
   Uses: QueueEmpty, Error. */
void ServeNode(QueueNode **p, Queue *q)
{
   if (QueueEmpty(q))
      Error("Attempt to delete a node from an empty queue.");
   else {
      *p = q->front;               /* Pull off the front entry of the queue.         */
      q->front = q->front->next;  /* Advance front of queue to the next node.       */
      if (QueueEmpty(q))           /* Is the queue now empty?                        */
         q->rear = NULL;
   }
}
```

The ** in front of the argument p indicates that the function expects 'a pointer to a pointer.' That is, the function receives a reference to a pointer; this is also commonly referred to as 'the address of a pointer.' (For more information on references see Appendix C.)

Again the possibility of an empty queue must be considered separately. It is an error to attempt deletion from an empty queue. It is, however, not an error for the queue to become empty after a deletion, but then the rear and front should both become NULL to indicate clearly that the queue is empty.

simplicity

If you compare these algorithms for linked queues with those needed for contiguous queues, you will see that the linked versions are both conceptually simpler and easier to program.

implementations

The functions we have developed process nodes; to enable us to change easily between contiguous and linked implementations of queues, we also need versions of functions Append and Serve that will process *entries* directly for linked queues. We leave writing these functions as exercises, along with the remaining functions for processing queues: CreateQueue, ClearQueue, QueueEmpty, QueueFull, QueueSize, QueueFront, and QueueFrontNode.