Introduction to Queues

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.

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 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, with multitasking, 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 task, which must be done in turn by keeping them in a queue.

Queue operations. Indeed, we need to perform two operations with queues: append (or add), put an entry to the queue, and serve (or delete), remove an entry from the queue. The entry in a queue ready to be served, that is, the first entry that will be removed the queue, is called 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, is called the rear (or the tail) of the queue.

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 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 append, append, serve, append, serve, append, serve, append, …

The problem is that as the queue moves down the array, the storage space at the beginning of the array is discarded and never used again.

Circular arrays. In concept, we can overcome the inefficient use of space simply by thinking of the array as a circular
rather than a straight line. In this way, as entries are added to and removed from the queue, the head will continually chase the tail around the array. 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.

Implementation of circular arrays. Our next problem is to implement a circular array as an ordinary linear 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 same numbered entries of a linear array. Then moving 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.
In C (or C++) we can increase an index \( i \) by 1 in a circular array by writing

\[
\text{If } (i >= \text{MAX}-1) \\
\quad \text{i}=0; \\
\text{else} \\
\quad \text{i}++; \\
\]

Or even more easily by using the \( \% \) operator:

\[ i=(i+1)\%\text{MAX}; \]

Before writing formal algorithm to add to or 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 a 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 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 then the array is full the rear will be exactly one position behind the front. Thus, we have another difficulty: the front and rear indices are in exactly same relative positions for an empty queue and for a full queue! This situation is illustrated in figure.
The following program illustrates the queue operations.

```cpp
#include <iostream>
#include <conio>

const int MAXQUEUE = 3;

typedef char item_type;
typedef struct queue_tag{
    int front;
    int rear;
    item_type entry[MAXQUEUE];
} queue_type;

queue_type queue;
item_type item, value;

void AddQueue(item_type);
void DeleteQueue();
void Initialize(queue_type);
bool Full();
bool Empty();

int main()
{
    queue.front = 0;
    queue.rear = -1;
    int tag, flag = 1;
    while(flag != 0)
    {
        cout << "Enter 1 to add to the queue or 0 to delete from the queue: ";
        cin >> tag;
        cout << endl;
        switch(tag)
        {
        case 1:
            cout << "Enter item: ";
            cin >> item;
            cout << "\n\n";
            AddQueue(item);
            break;

        case 0:
            DeleteQueue();
            break;

        default:
            cout << "Incorrect value entered. Enter new value.\n\n";
            break;
        }
        cout << "Enter 1 if you want to continue or 0 if you want to exit: ";
        cin >> flag;
        cout << endl;
    }
    return 0;
}
```

getch();
return 0;

void AddQueue(item_type item)
{
   if (Full() == true)
      cout << "Queue is full!!!\n\n";
   else
      queue.rear = (queue.rear + 1) % MAXQUEUE;
      queue.entry[queue.rear] = item;
}

void DeleteQueue()
{
   if (Empty() == true)
      cout << "Queue is empty!!!\n\n";
   else
      value = queue.entry[queue.front];
      cout << value << '\n';
      if (queue.front != queue.rear)
         queue.front = (queue.front +1)%MAXQUEUE;
      else
         queue.front = 0;
         queue.rear = -1;
}

bool Empty()
{
   if (queue.rear <= -1)
      return true;
   else
      return false;
}

bool Full()
{
   if ((queue.rear == queue.front -1 &&
      queue.rear > -1) ||
      (queue.rear == MAXQUEUE -1 &&
      queue.front == 0))
    return true;
else
    return false;
}