Chapter 42 - Queues

1. Queues

The origin of the type of data structure known as a queue is obvious - to anyone from the United Kingdom. What they call a *queue* is known as a *line* in the United States. In the UK, you queue for movie tickets. In the US, you stand in line for movie tickets. Queues are joined at the back and left from the front. Queue jumping (cutting in line) is forbidden.

The basic idea of a queue is that data can only be added to the back and can only be accessed from the front. This means that the chief property of a queue is to insure that data that is received first is processed first. Because of this, queues are said to be first in, first out (FIFO) data structures and are sometimes referred to as FIFO queues.

A single queue is ideal for regulating access to a resource in a way that insures that requests for that resource are filled in order of request. For example, a common job for a network operating system is to provide access to a printer for multiple network clients. Any client may print to the printer at any time. However, only one print job can be printed at a time. Network operating systems usually receive a print job from a client, then place that job at the back of a print queue. When the printer is idle, the job at the front of the print queue is removed from the queue and sent to the printer. In this way, all users of a network have equal access to a common resource, the printer. All users may tell the client software to print to the printer at the same time. When a client finishes creating a print job, that job is added to the network print queue where it waits for the print jobs ahead of it to finish printing so that it may be sent to the printer.

![Diagram of a single queue](image1)

Multiple queues are ideal for regulating access to a resource when different groups or types of requests have different priorities assigned to them. To use the network printer example, it is common for large institutions to share very expensive high speed and quality printers among hundreds of network clients at one time, assigning differing priorities to different types of jobs or users. Giving each priority level its own queue is one way to implement this. Here is an illustration using three queues, each representing a separate priority level.

![Diagram of multiple queues](image2)
In the preceding illustration of multiple print queues, print jobs sent from administration clients go to a queue reserved for them, as do the print jobs created by faculty clients and student clients. All three queues serve the same printer. In a typical case, administrative queue print jobs will be handled first, faculty jobs second and student jobs last. If there is a job in the administration queue, it will be given access to the printer the next time the printer becomes idle, i.e. finishes a print job or comes on line. Print jobs in the faculty queue will have to wait until all jobs in the administration queue are serviced before they will be allowed to print. Print jobs in the student queue will have to wait until all jobs in the administration queue and the faculty queue are serviced before they will be allowed to print. This arrangement works because there are seldom many administration print jobs and, while there are more faculty print jobs, there are many more student print jobs then generated by either of other groups. This arrangement would not work if the administration or faculty generated enough print jobs so that their queues were never empty.

In C++, queues are frequently implemented as dynamic data structures in the form of linked lists. A pointer variable named \textit{front} is used to point to the first node in the queue. A pointer variable named \textit{back} is used to point to the last node in the queue. Operations on the queue include the creation of the list structure, enqueuing (adding a new node), dequeuing (removing a node from the queue), a function to determine if the queue is empty and the disposal of any remaining nodes in the queue when the data structure passes from scope.

To implement a queue as a specialized dynamic list, a struct is needed. This will consist of the data fields and a pointer to the next node in the queue.

\begin{verbatim}
struct nodeType {
    dataType dataField1, dataField2, ... dataFieldn;
    nodeType *next;
};
\end{verbatim}

Creation of the list structure is very simple. The \textit{front} and \textit{back} pointers must be initialized to NULL to create the empty queue.

\begin{verbatim}
nodeType *front, *back;
front = back = NULL;
\end{verbatim}

The function to determine if the queue is empty is very simple. If the pointer \textit{front} is NULL, the queue is empty. If the queue is empty, the function should return true, else it should return false.

\begin{verbatim}
bool isEmpty() {
    if (front == NULL) return true;
    return false;
}
\end{verbatim}
Enqueuing is just adding a new node to the back of the linked list. If the queue is empty, \textit{front} and \textit{back} must simply be set to point to the new node. If the queue is not empty, the pointer to the next node of the last node must be set to point to the new node and \textit{back} must be set to point to the new node. The new node's pointer to the next node must always be set to point to NULL.

\begin{verbatim}
nodeType * temp;

temp = new nodeType();
temp->dataField1 = data;
temp->dataField2 = data;
::
temp->dataFieldn = data;
temp->next = NULL;

if (isEmpty()) front = temp;
else back->next = temp;
back = temp;
\end{verbatim}

Dequeuing requires that the data be preserved so that it can be used. This can be done using a variable of nodeType. This can be a new variable or the same dynamic node. If a new node variable is used, the original one must be deleted. In either event, \textit{front} must be updated to point to whatever the \textit{next} pointer of the dequeued node points to. Of course, an empty node cannot be dequeued and, if dequeuing produces an empty list, \textit{back} and \textit{front} must be set to NULL.

\begin{verbatim}
nodeType temp;
if (!isEmpty( )) {
    temp = *front;
    temp.next = NULL;
    nodeType tempPointer;
    tempPointer = front;
    front = front->next;
    delete tempPointer;
    if (isEmpty( )) front = back = NULL;
}
\end{verbatim}

Disposing of the queue is the same as any linked list.

\begin{verbatim}
nodeType *temp;
while (first != NULL) {
    temp = first;
    first = first->next;
    delete temp;
}
\end{verbatim}
Exercise 42.1
Using the multiple queue illustration on page 1 of this chapter, list the job numbers in
the order that they would be printed. Assume that no other jobs will be added to any of
the queues.

Programming Exercise 42.1
Using the following struct, create a print queue class with a queue created from a
dynamic linked list. The *dequeue* function should return a non-dynamic copy of the first
node with the *next* pointer set to NULL. If the queue is empty, the node returned should
have a job number of -1. Include in the class a void function that displays the information
contained in all nodes of the linked list that represents the queue.

```
struct printJob {
    int job, pages;
    printJob *next;
};
```

Programming Exercise 42.2
Using the struct and class from 42.1, create a simulation of the single queue problem.
The logic cycle of the simulation should:

1. display the activity of the printer
2. display the state of the queue
3. allow the user of the simulation to insert as many jobs each logic cycle as
   wished

A job will occupy the printer for a time based on the number of pages, with one logic
cycle equaling the time needed to print one page. Job numbers should be created by the
program, not entered by the user. The number of pages of each print job should be entered
by the simulation user.

Programming Exercise 42.3
Using the struct and class from 42.1, create a simulation of the multiple queue
problem. The logic cycle of the simulation should:

1. display the activity of the printer
2. display the state of all queues
3. allow the user of the simulation to insert as many jobs each logic cycle as
   wished into any of the three queues

A job will occupy the printer for a time based on the number of pages, with one logic
cycle equaling the time needed to print one page. Job numbers should be created by the
program, not entered by the user. The number of pages of each print job should be entered
by the simulation user.