**Queue**

Queues follow the First in First Out or FIFO method. The data added first is removed first as well, while new data is added to the end of the queue. This means, unlike stacks, we need to keep track of two positions, the head, from where we will remove items, and the tail, where we will add items. We will be looking at the two primary functions for a queue, Enqueue, which will take some data x and add it to the queue, and Dequeue, which will remove the data at the head of the queue.

There are a few types of queues. First, we will look at the Basic Queue. For each queue, the pink and green cells indicate the positions of the head and tail respectively, blue cells indicate both the head and the tail are there. Easily visible values are currently in the queue, while the ones that are greyed out are older values that have been ‘removed’. We did not use such colours with stack, even though memory works the same way there as well, because it was more difficult to show since it kept going back and forth.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| 50 |  |  |  |  |  |
| 50 | 30 | 40 | 60 |  |  |
| 50 | 30 | 40 | 60 | 70 |  |
| 50 | 30 | 40 | 60 | 70 |  |
| 50 | 30 | 40 | 60 | 70 | 80 | Outside Queue |

For the Basic Queue, both the head and the tail will simply keep moving forward till the end of the queue. The first thing we need to solve is how to check when our queue is full. An easy way to do this is to simply check when tail = qMax, where qMax is the size of our queue. Until tail = qMax, we can keep adding items. Another problem is how to check when the queue is empty, meaning there are no more items that can be removed. From the cells above, it is obvious that this occurs when head = tail. Using this information, we can write our two primary functions.

void enqueue (Q, x, &tail, qMax)  
{  
 if (tail == qMax) *//Error. Queue is full.* else  
 {  
 Q[tail] = x;  
 tail++;  
 }  
}  
  
int dequeue (Q, &head, tail)  
{  
 if (head == tail) *//Error. Queue is empty.* else  
 {  
 cout<<Q[head];  
 head++;  
 }  
}

PSEUDOCODE

Note: There are some implementations of queues where the dequeue function only increments head. The item at head is displayed with a separate function.

There are a few more functions we could write for the basic queue:

* bool isEmpty() //head = tail
* int front() //return head element
* bool isFull() //tail = qMax
* int size() //tail – head

A major problem with the basic queue is that it can only be used once, since both head and tail keep moving forwards. This is a huge waste of memory. To solve this, we use circular queues. The major difference here is, when either head or tail has reached the end of the queue, it will circle back around to the front. Note that if there is a value that is difficult to see (has been removed) at the position of tail, the next input will overwrite that value.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 50 | 30 | 60 | 70 | 80 |  |
| 50 | 30 | 60 | 70 | 80 | 20 |
| 10 | 12 | 16 | 70 | 80 | 20 |
| 10 | 12 | 16 | 21 | 80 | 20 |
| 10 | 12 | 16 | 21 | 80 | 20 |
| 10 | 12 | 16 | 21 | 80 | 20 |
| 10 | 12 | 16 | 21 | 80 | 20 |
| 10 | 12 | 16 | 21 | 80 | 20 |

There are two problems here. First, how do we take head and tail back to the front. This is as simple as using mod values. Instead of incrementing head and tail, if we do something like head = (head+1)%qMax, their value will be set to 0 again when they reach the length of the queue. For qMax = 6, head = (head+1)%6. The second problem is how to tell when the queue is full and when it is empty. From the queues above, we can see that in both cases, head = tail. Thus, we need another variable to keep track of how many elements are currently in the queue.

void enqueue (Q, x, &tail, qMax, &count)  
{  
 if (count == qMax) *//Error. Queue is full.* else  
 {  
 Q[tail] = x;  
 tail = (tail+1)%qMax;  
 count++;  
 }  
}  
  
int dequeue (Q, qMax, &head, &count)  
{  
 if (count == 0) *//Error. Queue is empty.* else  
 {  
 cout<<Q[head];  
 head = (head+1)%qMax;  
 count--;  
 }  
}

PSEUDOCODE

Accordingly, our other functions are:

* bool isEmpty() //count = 0
* int front() //return head element
* bool isFull() //count = qMax
* int size() //count

In real life, queues are used in a lot of places, including printers, keyboard buffers and computer instruction execution.

There is another scenario for queues. Take a computer executing some program that hangs. We can override the program and force it to end, thus breaking out of queue and forcing the computer to execute our instruction before the hung program’s ones. This makes use of something called a priority queue, where data with higher priority goes first. This will be discussed later.