

Best data use

We learn about stacks and queues very early on, but sometimes some neat optimisations are missed

In this issue...

▶ WHAT'S COVERED

One of the very first data structures we learn in programming is how to use a singly linked list to implement a stack and a queue. Even though the initial code we write will be efficient, and can be guaranteed to be efficient by mathematical reasoning, our very familiarity with the concepts blinds us to some clever performance optimisations.

Every now and then I see a post in one of the programming newsgroups I frequent where someone talks about their implementation of a data structure and then asks for other people's comments about their implementation.

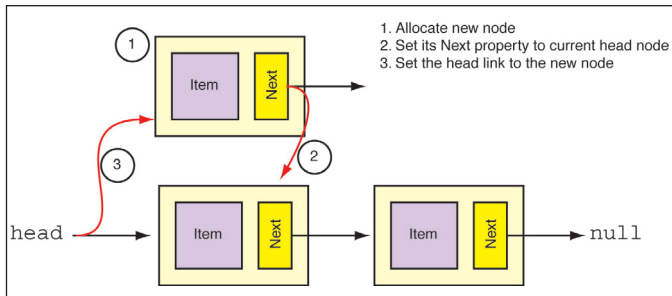
A recent one I read described a standard implementation of a queue. Although the queue was going to work just fine, there were a couple of very simple improvements that could have been made to boost performance and make the code more elegant. These enhancements would also make it easier to convert into a lock-free form for multi-threaded programming.

Let's take a look at building a stack and a queue using a linked list. I shall assume that we're using an object-oriented language, so you won't see any pointers here.

A stack has two main operations, traditionally called push and pop. The usual image we use is a stack of plates in a cafeteria: push adds a new plate to the top of the stack and pop removes the top one. When we implement the push operation we're going to be given an object of some description to add to the stack. We should allocate another object to hold it, a 'link object', which will also have a link to the next object in the stack. This link object is more generally known as a 'node'.

Nugget

One of the issues we still face with both the stack and the queue is memory management. The node is a small object: just an item and a next link. For each enqueue or push we will be allocating a new node, for every dequeue or pop we'll be implicitly marking a node as unused, or explicitly freeing it. Memory management for such small objects can be problematic: we could be spending quite a bit of time in the memory heap code. An optimisation that you could explore is to implement a free list of unused nodes as a specialised stack.



▲ Figure 1: Pushing a node onto a stack.

The first push merely saves the node internally to the stack. The second and subsequent pushes sets the link of the new node to the current list and saves the new linked list headed by this new node. Figure 1 shows the steps in pushing an item onto the stack.

Assuming that the push method will be passed an item to store on the stack, the algorithm goes like this: allocate a new node, set its 'item property' to the item we're given, set its 'next property' to the linked list already present in the stack (this could be null if there is no list yet), and then set the stack's linked list to this new node.

The pop operation

Now we've seen the push operation, let's take a look at the inverse operation, pop. There's a big wrinkle here that we should design for: what happens when pop is called with no items on the stack? In essence, there are two possibilities, one pretty

innocuous, but that could cause problems for the unobservant, and the other which is very vocal. The first option is to return null or nothing. This is simple, but it does impose on the user of the stack to understand and cater for the case that pop could return null. The second option is to throw an exception.

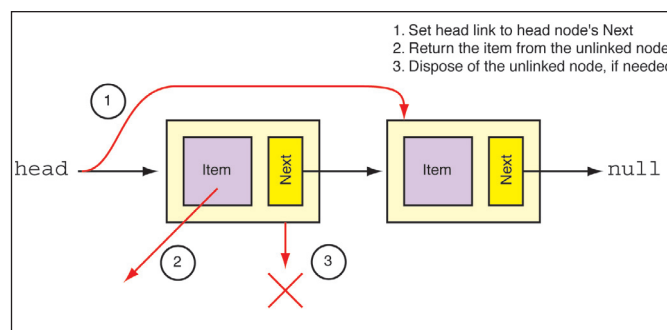
Personally, I prefer the first case for a couple of reasons. First it matches the way I program with stacks (it'll generally be a loop which I'll exit if there was nothing popped), and second it's less intrusive and more efficient than throwing an exception.

The algorithm for pop works like this (figure 2 shows the process): if there is nothing in the stack (the linked list will be null in this case), return null. Otherwise, save the node at the top of the linked list, set the linked list to the node's 'next property', and return the item in the node. If you're working in a non-garbage collected environment, there will be a couple of extra steps in order to de-allocate the node after you have finished with it, of course.

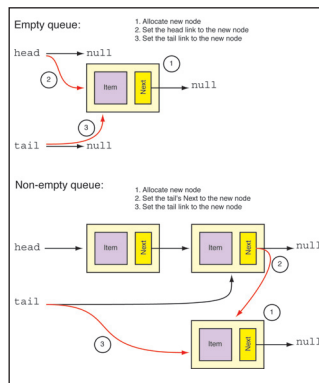
All in all, the stack is pretty easy to implement as a linked list, so let's take a look at the queue.

Form a queue

In a queue, we have to maintain links to both the front and end of the linked list, the front, or head, because that's where we'll be removing nodes, and the end, or



▲ Figure 2: Popping a node from the stack.



▲ **Figure 3: Dequeueing a node from the queue.**

tail, where we'll be adding nodes. Again, there's a traditional image we bear in mind: the supermarket queue. We join the queue for paying at the tail end of the queue, and the cashier processes the customers from the head.

So in our implementation we shall have to maintain two links: the first is the linked list itself, just as in the case of the stack, the second is a link to the final node in the linked list, the furthest from the head.

As there are two links to take care of, the algorithms for enqueue and dequeue (the queue equivalents of push and pop) are a little more complicated.

For enqueue (figure 3), the first time an item is enqueued is going to be a little different than from other times. When the queue is empty, both the head (the linked list itself) and the tail will be null. We'll just use the nullness of the tail as an indicator of

whether the queue is empty or not. So, if the tail is null, we allocate a new node, set its 'item property' to the item we're given, set its 'next property' to null, and then set both the head and tail properties to the new node.

Otherwise, when the tail is not null, we allocate a new node, set its 'item property' to the item we are given, set its 'next property' to null (just as before). Now comes the fun bit: we set the tail's 'next property' to our new node, and then set the tail to our new node.

Dequeue, on the other hand, is replete with special cases (figure 4). There are three: the queue is empty (not shown), the queue has exactly one node, and the queue has more than one node.

The easiest case is when the queue is empty: we merely return null for the item. No changes to the queue itself need to be done.

Let's now look at the case where there is more than one node in the queue (we'll see that the final case is a special case of this scenario). In essence, the operations are exactly the same as for the stack's pop method: save the node at the top of the linked list, set the linked list to the node's 'next property', and return the item in the node. If there were more than one node in the linked list, we wouldn't have to do anything else; the tail link would still be valid.

If, however, there was only one node in the list, removing the one and only node would make the tail link invalid since it points to

a node that was no longer in the list. So in this case we would have to set the tail link to null (the head link would automatically be set to null, just as in the pop case). And how do we know if the linked list only has one node? Easy: both the head and the tail links will be equal when we start the dequeue operation.

At this point, let's step back a moment and look at the complexities of each method. Push has only one case: you do the same no matter how many nodes are in the linked list. Pop has two cases: the stack is either empty or it isn't. Enqueue has two cases: the container is either empty or not empty. Dequeue has three cases: empty, one node, and more than one node.

Making it simple

Naturally, the question poses itself: is there a way to reduce the complexity of the queue's operations? To make it as simple as the stack?

Well, if we could guarantee that the queue always has at least one node in it, we certainly could. It would never be empty, which would remove a case for both enqueue and dequeue. However, we can't always guarantee that the queue would have at least one node, of course: we don't know, when implementing the queue, how it's going to be used. But even if we can't guarantee for the queue, we can assure it for the linked list inside the queue. This linked list is hidden from the

Nugget

Another strategy to explore to minimize the allocation of lots of little node objects is to make the node a structure (a record) instead of a class, and then allocate a large array of nodes (call this a block). Then push all these nodes onto a free list, and then use the free list for allocating nodes.

This strategy is a little more complex than a simple free list since you'll find that you have to manage one or more blocks, again in some kind of linked list, but a benefit of this algorithm is twofold.

The first is, depending on how you tweak things, the nodes you use will generally all be in the same memory block. The operating system will not have to swap memory pages in and out as you refer to nodes. The second benefit is that you can refer to the nodes by index and not by reference, which may similarly improve performance slightly. ■

outside world, so you can force it to have at least one node when you construct the queue instance.

So, in the queue's constructor, you would allocate a node, set both its item and next properties to null, and then set the head and tail links to this dummy node. From this point the linked list will never be empty.

Of course, we now have to redefine when the queue itself is empty (this dummy node doesn't count for the user of the queue since it contains no items and will never be dequeued). I'm sure you can see that the queue is empty if the head and tail links are equal.

Now, with this dummy node present, when we enqueue we set the tail's 'next property' to our new node, and then set the tail to our new node. When we dequeue, if the head equals the tail, we return null; otherwise we save the head's 'next node' (remember the head itself is a dummy node), set the head's 'next property' to this saved node's 'next link'.

And there you are, with the use of a dummy node, we've simplified the queue's operations quite considerably: in the case of enqueue by removing an if statement completely, and for dequeue by removing one of the two if statements. ■

Julian M Bucknall has worked for many major companies and is now CTO for Developer Express. feedback@pcplus.co.uk

Spotlight on... Removing the head link

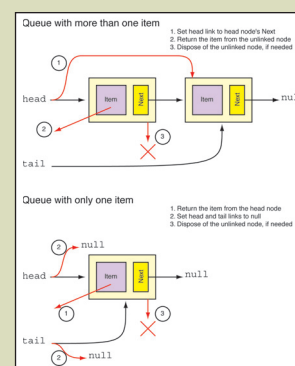
One improvement you can make to the standard queue algorithm that involves a fun little trick is to remove the head link altogether and just have one link to update as we enqueue and dequeue nodes from the queue.

This seems impossible: after all the head link is the linked list itself. The trick is to use a circular linked list instead of a linear list. When the queue is first constructed, we allocate our dummy node. Instead of setting its 'next property' to null as we did before, we set it to itself to form a circular list. We call this dummy node the tail. To determine if the queue is empty,

we check to see if the tail's 'next link' is equal to the tail.

Now to enqueue a new node, we do pretty much exactly the same as we did before: set the 'next link' of the new node to the 'next link' of the tail node (in the standard queue algorithm, this was null), set the tail's 'next link' to our new node, and then set the tail link to the new node.

To dequeue the node from the head we have to find it first. Since the linked list is circular, the head node is the node that is pointed to by the tail node's 'next link'. Now that we have the head node, we can proceed as before. ■



▲ **Figure 4: Enqueueing a new node into a queue.**