Back On The Chain Gang

Some thoughts on linked lists and elementary structures

by Julian Bucknall

One of the most ubiquitous data structures in the programming canon is the linked list. They crop up everywhere. Even when programming with Delphi and wielding its `TList`, the jack of all trades container, there is still room for a linked list. If you think back to October's *Algorithms Alfresco* on how to store graphs in memory, one of the implementations I used was an array of linked lists (all right, I admit it was a `TList` instead of an array, but if you think about it, a `TList` is just a class representation of an array of pointers). If you think back to January's column, the breadth-first traversal of a graph used a queue, and the queue I used was based on a linked list.

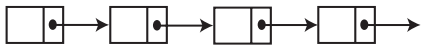
So linked lists are still important (we'll see some more definite reasons why in a minute). I dare say that a large proportion of my readers have coded up a linked list several times in the past, and are now wondering whether to continue reading about something learned at grandma's knee. Bear with me a while and we'll see whether I can teach you something new. If the worst comes to the worst, at least you'll get some reusable code at the end of it.

Day After Day

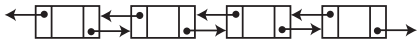
So what is a linked list? At its most basic, it is a chain of items or objects of some description, with each item containing a pointer pointing to the next item in the chain (a singly linked list, Figure 1) or with pointers to both the next and previous items in the chain (a doubly linked list, Figure 2). The reason they are so important is that adding or removing an item from a linked list is a constant time operation no matter how many items there are in the list or where the addition/deletion is taking place. Compare this with an array (or `TList` for that matter) where if you insert an item at the top of the array you must move all of the items down by one place to make room. Obviously this move takes longer the more items there are. Another great benefit of linked lists is they can grow to an arbitrarily large size, unlike an array, we don't have to allocate larger and larger contiguous blocks of memory and move the current data over every time; with a linked list the list grows incrementally in small steps, not in larger and larger strides.

Let's be a little more concrete here, build a singly linked list and see what we discover in the process. Firstly we must define the type of the item we shall be storing in the linked list. As a first stab people generally come up with the `TSimpleNode` record definition in Listing 1. The only field of note for our purposes is the next pointer `Next`; the remainder of the fields in the type definition are just our own variables we want to track in the linked list, here represented by a single field called `Data`. OK, pretty easy so far. The linked list itself is a variable of type `PSimpleNode`, initially set to `nil` to indicate an empty list. You can think of the linked list variable as being a pointer to the first item in the list. By the way, the items in the list are generally known as *nodes*, and that's the way we shall refer to them from now on.

How do we insert a new node? For a singly linked list there's a single basic possibility: insertion after a given node in the list (we do have to make an exception to the rule: adding a node before all the others in the list). In simple language we set the `Next` pointer in our new node to the node after the given node and we set the `Next` pointer of the given node to our new node. Similarly for deletion: the simplest possibility is deletion



➤ *Figure 1: A singly linked list.*



➤ *Figure 2: A doubly linked list.*

```
type
  PSimpleNode = ^TSimpleNode;
  TSimpleNode = packed record
    Next : PSimpleNode;
    Data : SomeType;
  end;
```

➤ *Listing 1:*
A singly linked list item.

after a node in the list (although we do have to make a special case for deleting the first node in the list). Here we set the given node's `Next` pointer to the node after the one we are about to delete: at that point the node to be deleted is unlinked from the list and we can dispose of it. Listing 2 has the details and Figures 3 and 4 show the steps involved in the general case in both these routines. Notice that both routines have the list parameter as a `var` parameter: it is entirely possible that the pointer to the linked list will be changed by both routines. Of course we could code all this inside a class and therefore hide this possibility inside the implementation. We won't do so *just* yet, though.

Traversing the list is pretty simple as well. We essentially walk

the list, going from node to node following the Next pointers, until we reach the nil node that signifies the end of the list. Listing 3 shows the simple loop required where the Process procedure is defined elsewhere and presumably will do something with the Data field of the node it is passed. Emptying a linked list uses a slight variation of this technique in order to make sure we don't refer to the Next field of the node after it is disposed.

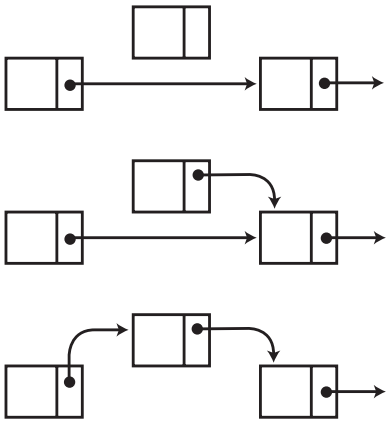
Now we've seen traversals, let's ask the question which may have popped into your mind a couple of paragraphs back. What if we want to add a node *before* another? How do we do it? The only way with a singly linked list is to traverse the list, looking for the node before which we want to add our new node. In the process of traversing the list we maintain a variable that points to the previous node (the parent, if you will). Once we find the node we were looking for we'll have the pointer to the previous node and we can just use the InsertAfter routine on this parent node. Listing 4 shows this technique with the InsertBefore routine. Notice the special code to cater for the case where you're adding a node before the first node (the parent is nil in this case). This routine isn't as fast as the InsertAfter routine I discussed above because it requires the linked list to be walked beforehand to find the parent of the given node. In general, if there is a possibility of inserting before a node, we'd use a doubly linked list instead.

Brass In Pocket

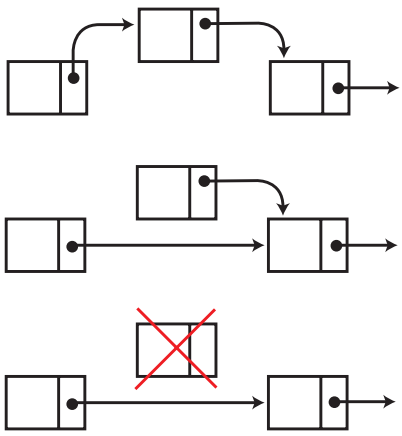
I'm quite sure that none of this so far is a surprise to you. However, look again at the code for the insertions and the deletion. Doesn't it strike you as somehow, you know, *messy*? It does to me. We have to have these special cases in both routines in order to cope with processing the first node in the list. We have to make sure that the linked list parameter is a var parameter in case it gets changed. Isn't there a better way? The answer is yes, by using dummy nodes.

In the case of a singly linked list we would use a single dummy node at the head of the linked list. The first node in the list that stores data will be the one pointed to by the head node's Next field. The end of the linked list is determined by a nil Next value, as before. We have to have a routine to initialize the list this time (before all we did was set the list variable to nil, now we have to allocate a dummy node to act as the head node) and one to destroy the list (we have to deallocate the head dummy node). If this isn't ringing alarm bells about writing a class to encapsulate all this then there's no hope I'm afraid!

➤ *Figure 3: Adding a node to a singly linked list.*



➤ *Figure 4: Deleting a node from a singly linked list.*



➤ *Listing 2: Insertion and deletion in a singly linked list.*

```
procedure InsertAfter(var aList : PSimpleNode; aNode : PSimpleNode;
  const aData : SomeType);
var NewNode := PSimpleNode;
begin
  {create new node, save data}
  New(NewNode);
  NewNode^.Data := aData;
  if aNode = nil then begin
    {insert at front}
    NewNode^.Next := aList;
    aList := NewNode;
  end else begin
    {insert after given node}
    NewNode^.Next := aNode^.Next;
    aNode^.Next := NewNode;
  end;
end;
procedure DeleteAfter(var aList : PSimpleNode; aNode : PSimpleNode);
var NewNode := PSimpleNode;
begin
  New(NewNode);
  NewNode^.Data := aData;
  if aNode = nil then begin
    {delete the first node}
    if aList <> nil then begin
      OldNode := aList;
      aList := OldNode^.Next;
      Dispose(OldNode);
    end;
  end else begin
    {delete after given node}
    OldNode := aNode^.Next;
    if OldNode <> nil then begin
      aNode^.Next := OldNode^.Next;
      Dispose(OldNode);
    end;
  end;
end;
```

➤ *Listing 3: Traversing a linked list.*

```
procedure TraverseList(aList : PSimpleNode);
begin
  while aList <> nil do begin
    Process(aList);
    aList := aList^.Next;
  end;
end;
```

Before we actually write such a class there's one more thing to consider. We started off by declaring a node record type to hold firstly the data we were interested in and secondly a pointer to the next node in the linked list. The second item is invariant, but the first depends on our particular application, or on the particular use we have at the moment. We could have one field, two or more. It seems hard to write a generic, reusable linked list class when we don't know ahead of time what we are going to store in it.

There are two solutions to this conundrum. The first one is to declare an ancestor node class that just consists of the `Next` pointer. Your data is then defined as a descendant of this class. In this case, you are responsible for allocating and deallocating the nodes, all the linked list wants are preallocated nodes whose `Next` pointers it can manipulate. It must be said that this is a slightly inelegant method since you are forced to declare descendants of the ancestor node class to store your data (what if the items you wanted to put in the linked list were instances of a class that you had no control over?).

The second solution, and I think the much better one, is to abstract out the data into the form of a typeless pointer. When we add an item to the linked list, we just present the linked list class with a pointer value (say a pointer to our data, or an object of ours on the heap) and let the linked list do the rest: allocating a node, setting the data, maintaining the links. This is the approach taken by my freeware EZDSL class library *[get it at ftp://ftp.turbopower.com/pub/misc/funcs/ezdsl301.exe Ed]*. This is a clean way around the problem since the user of the class doesn't have to know anything about the `Next` pointers, or reserve space for them, or create a special descendant of an ancestor class, etc.

Space Invader

This second solution has a corollary that is even more compelling. The nodes used by the class in this case are *always* eight bytes in size: a `Next` pointer and a `Data` pointer, both four bytes. So what? Well, when the linked list wants to store some data, it has to allocate the node first. To do this it would have to use the highly complex Delphi heap manager to allocate eight bytes. The heap manager has all sorts of fabulous code to manage chunks of memory and to allocate and free arbitrarily sized blocks for our use, and it manages all this complexity and functionality in an amazingly efficient manner. But we know that we only want eight byte blocks and we will always be dealing with eight byte blocks. Can we use this regularity to speed up the allocation and deallocation of our fixed sized nodes? The answer is, of course, yes: we allocate a batch of nodes from the Delphi heap manager inside the linked list object and dole them out as required. Of course we don't allocate a batch of nodes singly, we allocate instead 1,024 bytes, say, and view this large block as 128 separate nodes. If we require more than 128 nodes, then we allocate another 1,024 byte block to give us a further 128 nodes.

But here comes the interesting part. The large blocks we allocate we store in an internal linked list, and the nodes we split from them go into a free list, which is also a linked list. So, our linked list class will rely on linked lists itself to work more efficiently. Cute, eh?

What do I mean by a free list? This is a common construction in programming. What happens is that we have a set of items that we 'allocate' and 'free' and when freed will in all likelihood be reused at some stage. Delphi's heap manager uses a free list of deallocated memory blocks of differing sizes. Many database engines will have a free list of deleted records that can be reused. When we want to allocate an item we go to our free list and reuse one of the items on it. If all the items are the same, then the free list becomes a stack (at least on the surface, underneath it's still a linked list): to free an item we push it onto the free list, to allocate an item we pop the first one off the free list.

In our node allocation manager, there is a free list of nodes, initially set to `nil`. When we want to allocate a node the manager looks to the free list. If there are no nodes present (the free list is `nil`), the manager allocates a large chunk of memory from Delphi's heap manager, usually called a *page*. It then splits up this page into node sized pieces and pushes them all onto the free list. After this process, it can pop a node off the free list and return it to the consumer. When a node is freed, the node allocation manager just pushes it onto the free list. By 'push' I mean insert a node at the top of the list, and by 'pop' I mean delete the node at the top of the list.

➤ *Listing 4: Insertion in a linked list before another.*

```
procedure InsertBefore(var aList : PSimpleNode; aNode : PSimpleNode;
  const aData : SomeType);
var
  NewNode  : PSimpleNode;
  WalkNode : PSimpleNode;
  Parent   : PSimpleNode;
begin
  {create new node, save data}
  New(NewNode);
  NewNode^.siData := aData;
  {find the given node}
  Parent := nil;
  WalkNode := aList;
  while (WalkNode <> nil) and (WalkNode <> aNode) do begin
    Parent := WalkNode;
    WalkNode := WalkNode^.Next;
  end;
  {if Parent is nil, insert before the first node}
  if Parent = nil then begin
    NewNode^.Next := aList;
    aList := NewNode;
  end else begin
    {otherwise insert after Parent}
    NewNode^.Next := Parent^.Next;
    Parent^.Next := NewNode;
  end;
end;
```

```
type
  PsnmPage = ^TsnmPage;
  TsnmPage = packed record
    snmpNext  : PsnmPage;
    snmpNodes : array [0..pred(PageNodeCount)] of TsllNode;
  end;
var
  snmFreeList : PsllNode;
  snmPageList : PsnmPage;
procedure snmFreeNode(aNode : PsllNode);
begin
  {add the node to the top of the free list}
  aNode^.sllnNext := snmFreeList;
  snmFreeList := aNode;
end;
procedure snmAllocPage;
var
  NewPage : PsnmPage;
  i       : integer;
begin
  {get a new page}
  New(NewPage);
  {add it to the current list of pages}
  NewPage^.snmpNext := snmPageList;
  snmPageList := NewPage;
  {add all the nodes on the page to the free list}
  for i := 0 to pred(PageNodeCount) do
    snmFreeNode(@NewPage^.snmpNodes[i]);
end;
function snmAllocNode : PsllNode;
begin
  {if the free list is empty, allocate a new page of nodes}
  if (snmFreeList = nil) then
    snmAllocPage;
  {return the first node on the free list}
  Result := snmFreeList;
  snmFreeList := Result^.sllnNext;
end;
```

➤ *Listing 5: The singly linked node allocation manager.*

Listing 5 shows the code in the node allocation manager. The only 'scary' stuff in this code, and it's not *that* scary, is the partitioning of the page into the smaller node sized pieces, but if you look closely, all it does is to add each node in the page to the free list by passing its address to the free node routine.

By the way, just to show that it's worth going to all the trouble of this node allocation manager, my tests have shown that it is at least twice as fast as the Delphi 4 heap manager over a full cycle of allocations and deallocations of millions of nodes.

Talk Of The Town

At this point we can start to write a generic singly linked list class. We know how to create an efficient node allocation manager, we can insert nodes into the list, delete nodes from the list and so on. We know we need to create a dummy node at the head in order to make the insertion and deletion code more efficient. Furthermore we can abstract out the data from the mechanics of the linked list itself. Are we really there yet? Not quite. If you look back at the original insertion and deletion code, we were calling the routines with reference to a particular node. This is a little difficult when we are trying to remove the need for the user of the linked list to know about nodes.

What we shall do instead is to introduce the concept of a cursor. A cursor is a gadget that points somewhere inside the linked list. We can set the cursor at the start of the linked list, before all of the nodes, and we can move this cursor forwards through the list. We can also examine the data at the cursor. Because of the way the singly linked list works we cannot move backwards (there are no backward links, remember: this will have to wait for the doubly linked list). We should be able to test to see if the cursor is before all nodes or after all the nodes (akin to beginning of file or end of file when we are talking about databases).

Now, having mocked up a quick design, we can write the singly linked list class. Rather than print all the code in the magazine and bulk out the article, please have a look at this month's disk for all the gory details.

Mystery Achievement

One of the questions that periodically comes up is how to sort a linked list. You may not believe it but it is possible to sort a singly linked list. I'll admit that when I first considered this I was of the opinion that there wasn't enough information in a singly linked list to do so: I thought that backward links were required. Of course I was wrong (not the first time!), we just need several pointers or cursors into the list.

What we'll do to sort a linked list is to perform an insertion sort (refer back to September 1998's *Algorithms Alfresco* for details). We use two cursors into the linked list: one cursor will move through the entire list, node by node, and the node it is pointing to will be the node we are trying to insert into the proper sequence. The other cursor will be used to walk through the currently sorted nodes, trying to find the correct spot for the node we're trying to insert. Think back to the card metaphor we were using in September. Deal yourself a hand of cards. Starting from the left take out the next card and try and place it in order in the set of cards prior to it that are already sorted. Unlike before, to find the correct spot, we'll always start at the beginning of the list and work our way forward until we get to a node that's 'greater' than the one we are trying to insert (or, indeed, until we reach the node itself, at which point we shall assume that it is in the correct place). Because the data in our nodes is an untyped pointer, we shall have to use the usual trick of supplying a function to compare two data items in order to perform the sort. The function returns an integer less than 0 if the first data item is less than the second, 0 if they are equal, or an integer greater than 0 if the first is greater than the second. Listing 6 has the details.

Once we have a linked list in sorted order, it would be nice to keep it so as we insert new items. The usual algorithm is simple. Essentially it's a sequential search for the correct position from the start of the linked list, comparing each node we encounter against the one we are trying to insert until we find the correct place to insert it, basically what we'd been doing in the middle of the insertion sort. If, however, the cost of comparing two nodes is large compared with moving from node to node (for

example, comparing two strings takes far longer than moving from one node to another), then we can try a different tack: a binary search.

Those of you who've used binary search before are now probably wondering how on earth we can perform a binary search on a linked list. Well, settle back and find out. A binary search usually works like this. Assume we have a set of items in sorted order and we can access each of them efficiently by a random access method (for example we have an array of strings in alphabetic order). Now the object of the exercise is to find an item we're given in the set. Get the item at the midpoint of the set and compare it with our given item. If it's equal we're done. If the midpoint item is less than ours, we perform the exact same steps on the latter half of the set, since that's obviously where the item will be found. If, however, the midpoint is greater than our item, then we perform the same steps on the first half of the set. Every time we make a comparison, we essentially find the item or we halve the set.

So how do we do this for a linked list? Well, we must make a small assumption first: we know the number of items in the set. Not too large an imposition, I'm sure you'll agree, and in fact the linked list class on the diskette maintains the count of items through insertions and deletions. Using this we can now describe binary search on a linked list.

1. Store the dummy head node in a variable `BeforeList`.

2. Calculate the number of items in the list, store in variable `ListCount`.

3. If `ListCount` is zero, insert item after `BeforeList`.

4. Calculate half of `ListCount`, rounded up if need be, store in `MidPoint`.

5. Move from `BeforeList` through the list, counting `MidPoint` nodes.

6. Compare this node (the `MidPoint`*th* node) against the item we have.

7. If equal, insert the item after this node.

8. If the node is less than our item, store this node in `BeforeList`, subtract `MidPoint` from `ListCount`, go back to step 3.

9. If the node is greater than our item, store `MidPoint-1` into `ListCount`, go back to step 3.

As you can see we are logically subdividing the linked list up into smaller and smaller sublists, reducing the number by half each time. Suppose in our search we use the maximum number of subdivisions: the worst case scenario. If there are *n* nodes in the list, we will follow approximately *n* links (ie, $n/2 + n/4 + n/8 + \ldots$, which in the limiting case is equal to *n*) and make approximately log *n* comparisons. Compare that with the sequential search method where in the worst case we'll follow *n* links and make *n* comparisons; but on average, we'll follow *n*/2 links and make *n*/2 comparisons. So, if the average time to make a comparison is much larger than the time to follow a link then we'll do much better to perform a binary search. Because of the way we have structured our linked list class, all comparisons are performed through at least one function call; hence, we'll find that a binary search is almost always better. I leave the implementation of the above algorithm as an exercise for the reader.

Precious

As we've already alluded to above, one great use for a singly linked list is for a stack. A stack is a container to which we add items and from which we can remove them in reverse order (a last in, first out, or LIFO container). The operations are usually called *push* for the add operation and *pop* for the remove operation. The stack is just a singly linked list where we only add nodes before the first node and remove the first node. Listing 7 shows the basic stack class.

We can now easily set up a queue class as well. A queue is a container to which we can add items and from which we can remove them in the order in which they were originally added (basically the oldest item first). This is known as a first in, first out (FIFO) queue. Unlike a stack, the queue's two main operations don't seem to have standard names. We'll use *enqueue* for adding an item to the end of the queue and *dequeue* for removing the oldest from the head of the queue (some books just use *put* and *get*, for example). Since we're adding items to the end of the queue, we make use of a pointer pointing there; this is usually known as the tail pointer. Unlike the head pointer which is a separate node altogether that

➤ *Listing 6: Insertion sorting a linked list.*

```
procedure TaaSingleList.Sort(aCompare : TaaCompareFunction);
var
  Walker : PsllNode;
  Temp   : PsllNode;
  WalkerParent : PsllNode;
  TempParent   : PsllNode;
begin
  {if there are zero (or one) items list is already sorted}
  if (Count <= 1) then
    Exit;
  {perform an insertion sort from the second item onwards}
  WalkerParent := FHead^.sllnNext;
  Walker := WalkerParent^.sllnNext;
  while (Walker <> nil) do begin
    {find where walker item should be in sorted list to its
     left: walk the sorted sublist making a note of parent
     as we go so we can insert properly. Note the loop below
     will terminate in the worst case by the walker node
     itself: we won't run off the end of the list}
    TempParent := FHead;
    Temp := TempParent^.sllnNext;
    while (aCompare(Temp^.sllnData, Walker^.sllnData) < 0)
      do begin
      TempParent := Temp;
      Temp := TempParent^.sllnNext;
    end;
    {did we find the walker node? If so, it's in the right
      place so move the walker's parent on by one link}
    if (Temp = Walker) then
      WalkerParent := Walker
    {otherwise, move the walker node into the correct
     place in the sorted sublist; leave the walker's parent
     where it is}
    else begin
      {disconnect the walker node}
      WalkerParent^.sllnNext := Walker^.sllnNext;
      {connect the walker node in the correct place}
      Walker^.sllnNext := Temp;
      TempParent^.sllnNext := Walker;
    end;
    {set the walker node}
    Walker := WalkerParent^.sllnNext;
  end;
end;
```

```
constructor TaaStack.Create;
begin
  inherited Create;
  {allocate a head node}
  FHead := snmAllocNode;
  FHead^.sllnNext := nil;
  FHead^.sllnData := nil;
end;
destructor TaaStack.Destroy;
begin
  Clear;
  snmFreeNode(FHead);
  inherited Destroy;
end;
procedure TaaStack.Clear;
var Temp : PsllNode;
begin
  Temp := FHead^.sllnNext;
  while (Temp <> nil) do begin
    FHead^.sllnNext := Temp^.sllnNext;
    snmFreeNode(Temp);
    Temp := FHead^.sllnNext;
  end;
  FCount := 0;
end;
function TaaStack.Pop : pointer;
var Temp : PsllNode;
begin
  if (Count = 0) then
    raise Exception.Create('TaaStack.Pop: stack is empty');
  Temp := FHead^.sllnNext;
  Result := Temp^.sllnData;
  FHead^.sllnNext := Temp^.sllnNext;
  snmFreeNode(Temp);
  dec(FCount);
end;
procedure TaaStack.Push(aItem : pointer);
var Temp : PsllNode;
begin
  Temp := snmAllocNode;
  Temp^.sllnData := aItem;
  Temp^.sllnNext := FHead^.sllnNext;
  FHead^.sllnNext := Temp;
  inc(FCount);
end;
```

➤ *Listing 7: A stack class using a linked list.*

holds the linked list together, the tail pointer just points to the final node in the linked list, akin to a cursor into the list. The disk contains such a queue class, and Listing 8 just shows the enqueue and dequeue methods.

Having touted all the great benefits of a linked list there is one operation for which it is not well designed. That operation is the *retrieve the nth item* operation; for instance, get the 100th item. For an array, this particular operation is very fast (it usually just reduces to a simple address calculation; the address of the 100th item is the start address of the array plus 99 times the size of each item), but for a linked list the only way to do it is to walk the list, counting the nodes (in other words, start at the first node and follow the next pointers 99 times). We did this in the binary search routine, if you recall. It is up to us to recognize these essential differences between arrays and linked lists and use the appropriate one for the particular part of our application.

Thin Line Between...

Having spent some time discussing singly linked lists, we now move onto doubly linked lists. Here, in addition to the link to the next node in the list, each node has a link, or pointer, to the previous node in the list. Whereas in the singly linked list inserting a node required a single link to be broken and two links to be set (the one

```
function TaaQueue.Dequeue : pointer;
var Temp : PsllNode;
begin
  if (Count = 0) then
    raise Exception.Create('TaaQueue.Dequeue: queue is empty');
  Temp := FHead^.sllnNext;
  Result := Temp^.sllnData;
  FHead^.sllnNext := Temp^.sllnNext;
  snmFreeNode(Temp);
  dec(FCount);
  {if we've managed to empty the queue, the tail pointer is now
   invalid, so reset it to point to the head node}
  if (Count = 0) then
    FTail := FHead;
end;
procedure TaaQueue.Enqueue(aItem : pointer);
var Temp : PsllNode;
begin
  Temp := snmAllocNode;
  Temp^.sllnData := aItem;
  Temp^.sllnNext := nil;
  {add the new node to the tail of the list and make sure the tail
   pointer points to the newly added node}
  FTail^.sllnNext := Temp;
  FTail := Temp;
  inc(FCount);
end;
```

➤ *Listing 8: Enqueue and dequeue using a linked list.*

from the previous node to the new one and the one from the new one to the next), with a doubly linked list, two links will be broken and four will be set. The four are: from the previous node to the new one and *vice versa*, and from the new node to the next one and *vice versa*.

Just like in the singly linked list, it makes sense to have a dummy head node, but this time it also makes sense to have a dummy tail node at the end of the linked list. The main reason is that with a doubly linked list we can efficiently implement an `InsertBefore` type operation, and if we had a dummy tail node, we'd get rid of a special case again (the 'I want to insert this new node at the end of the list' case).

To create a doubly linked list class, we'd go through the same steps as for the singly linked list case. Our nodes this time will be 12 bytes in size: a `Next` pointer, a `Prev` pointer (for the link to the previous node) and a `Data` pointer, each four bytes. Hence our node allocation manager will have to manage 12-byte nodes instead of our original 8-byte nodes for the singly linked list case, and it still makes sense, for all the same reasons as before, to have such a manager.

Note also that this time our internal cursor can be placed at the end of the list, after all the nodes, and we can also move the cursor backwards through the list. Again, I'll leave the code for the doubly linked list class to the diskette.

Sorting a doubly linked list seems more complicated: after all, we've just seen how easy it is for the singly linked list case and now we have to worry about all of these pesky pointers to the previous nodes. Actually in practice we

```
WalkerParent := FHead;
Walker := WalkerParent^.dllnNext;
while (WalkerParent <> FTail) do begin
  Walker^.dllnPrev := WalkerParent;
  WalkerParent := Walker;
  Walker := WalkerParent^.dllnNext;
end;
```

➤ *Listing 9: Patching up the links after sorting a doubly linked list.*

cheat a little and just use the singly linked list sort algorithm without worrying at all about the previous pointers. That operation gets the linked list sorted in a singly linked fashion. At that point we clean up and make one pass through the entire linked list and patch up all the previous links. Listing 9 shows the algorithm; it's very simple and just requires us to walk through the list maintaining a `Parent` pointer and patching the current node's `Prev` link to point to it.

Show Me

We've now seen a singly linked list and a doubly linked list. Are there any other types of linked list? One of the more common ones is a circular linked list. This is a linked list where the last node's next link points to the first node. In effect the list forms a closed loop and there is no 'first' node or 'last' node. No matter where you are, if there are *n* nodes in the list, you can follow the next links *n* times and arrive back at where you were. There are two types of circular linked list: the first has nodes with next pointers only (a circular singly linked list) and the second has nodes with both forward and backward pointers (a circular doubly linked list). I won't provide any code for a circular linked list as I'm sure you will be able to write one fairly quickly using the code for the singly or doubly linked list.

Another type of linked list is the multilist. With a singly linked list we have one next link per node. With a multilist we have two or more next links per node. Why? To maintain several different orderings of the items. Suppose the items were customer records: we could create a multilist where the first set of links maintained the records in name order and the second set of links maintained the records in Social Security number order. Again a multilist can come in a couple of different flavors: singly linked and doubly linked. In fact, if you think about it, the doubly linked list is itself a multilist. In practical applications, it's not really necessary to write a proper multilist (ie, with a node that has the required number of forward links and backward links), we can get the same effect by using two or more simple linked lists encapsulated inside a class. In general, it would be better to use other data structures to maintain the ordered sequences since insertion or deletion from a binary search tree (for an example) would be faster than insertion or deletion from a sorted linked list.

There is one particular optimization we can do with an unsorted linked list to improve access times to individual items. Imagine that we have a linked list. The application we are writing requires a find operation on the linked list to find a particular item (presumably we've written a routine which will get called for each item in the list, and it will return true if the item is the one we want). A simple sequential search, in other words. An example is a linked list of currencies for example, dollars, pounds, francs, whatever, and we need to find the marks item. Suppose, furthermore, that your application (or the people using it) tends to find two or three currencies more often than others. It would make sense to reorganize the linked list so that these currencies were at the front: the sequential search would complete faster in that case.

With a linked list, this kind of optimization is extremely easy to do. When we find the item we are looking for, we delete the item from its current position and reinsert it at the front of the list (of course, if it is already there, we do nothing). Note that we don't have to deallocate and reallocate the node: we can reuse the one the item is attached to quite easily. Because insertion and deletion in a linked list is a constant time operation we won't, in practice, notice these rearrangements. We will however notice the fact that the find operation completes much more quickly.

This latter optimization is traditionally used in hash tables, at least the type where collisions are resolved by adding the colliding items to a linked list at each bucket (see my column on hash tables from the March 1998 issue of *The Delphi Magazine*, where I mention this algorithm). Since the items in a hash table are not sorted *per se*, this optimization trick with linked lists makes perfect sense since a sequential search has to be made along a bucket's linked list to find a particular item.

Don't Get Me Wrong

That's about it for now for linked lists. We've looked at singly and doubly linked lists, inserting and deleting items in them. We discussed how to abstract out the requirements for a linked list so that we could write a generic linked list class that would work for any type of item we wanted to use. We reviewed how to sort a linked list and how to perform a binary search on a linked list (something usually reserved for arrays or array-like containers). We showed how to use linked lists to implement a stack and a queue, two other useful containers. Next time we'll look at another type of linked list, the skip list.

Julian Bucknall's left arm nearly became unlinked from his shoulder over Christmas, and some of this article and code was written one-handed. He thanks his wife, Donna, for looking after him whilst his arm was strapped up. You can email him at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.