Follow Me, Follow You

In which we look at skip lists, a useful but little known sorted structure

by Julian Bucknall

Last year I finally joined the Association of Computing Machinery (ACM). I'd been meaning to for a while and the existence of this column and its monthly deadline was prompting me to do more and better research for the more esoteric algorithms so that I could present them to you. After all, there are only so many beginning algorithms out there. Anyway, the ACM has academic papers you can download for free, and it wasn't long after I'd joined when I came across a variant of a skip list, a favorite structure of mine. Of course, if I'm to describe a variant, I must describe the original first and the skip list structure follows nicely from last month's *Algorithms Alfresco* column on linked lists.

So let's consider skip lists.

It's Gonna Get Better

With a doubly linked list, if we wanted to find a particular item in the list, we had to start at the beginning and walk the list, following the `Next` pointers one by one until we found the item we were looking for. A sequential search in other words. If the list was sorted, we could employ a binary search technique to minimize the amount of comparisons we were doing, but still we had to follow the `Next` pointers in the list. I showed last time that even with binary search in the worst case we had to follow *n* `Next` pointers for *n* nodes in the list. Is there any way we could do better than this?

William Pugh, in his 1990 paper *Skip Lists: A Probabilistic Alternative to Balanced Trees*, showed that there was.

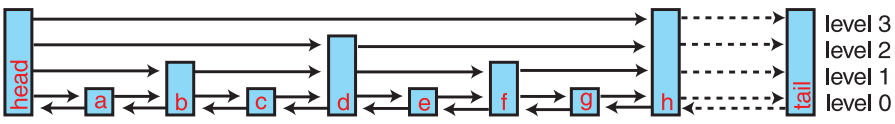
The paper's title already gives us some food for thought. *Probabilistic*? *Balanced Trees*? The latter I shall be covering in a later instalment of *Algorithms Alfresco*, the former we'll be dealing with in a moment.

What Pugh invented was a linked list, but one that was a little out of the ordinary to say the least. At its lowest level it is a doubly linked list, with a forward link to the next node in the list and a backward link to the previous. However, for this structure he made some nodes with another forward link that pointed to a node that was several nodes in front. This link *skipped* over a whole bunch of other nodes. He then made some of these nodes have yet another forward link that jumped even further ahead. And then again some of *these* nodes had another link that skipped over even more nodes. The structure looks a little like Figure 1. Notice that eventually all links end at the tail node, and that the head node is the start for all forward links at every level.

Of what use is this structure? If the items in the list were unsorted, it would be pretty useless. To find an item you'd still have to visit every single node and the extra links are worthless: they just take up space. On the other hand, if the items were sorted, you'd be able to jump over huge swathes of them, gradually taking smaller and smaller jumps, zeroing in on the item you're looking for. A bit like binary search in a way. We'll describe this process a little more rigorously in a moment.

So point one to recognize is that a skip list is a container that stores items in a sorted order.

➤ *Figure 1: A skip list.*



This makes the latter part of the title of Pugh's paper a little more understandable. Binary search trees (ie, binary trees that store items in sorted order) have a major problem. Inserting a bunch of items in sorted order causes the tree to break down into a linked list. Instead of being nice and bushy, the tree becomes long and spindly or twiggy. The good log *n* search times of the tree deteriorate into a linear process instead. To counteract this there are several varieties of balancing algorithms for insertion and deletion from binary trees, all designed to make them more bushy. But, of course, these balancing algorithms slow down those operations. Pugh was therefore comparing skip lists to balanced binary search trees to store sorted information.

Way Of The World

If you look again at Figure 1, you'll notice that there is a doubly linked list at level 0, a singly linked list at level 1 that skips over single nodes (ie, it links every second node), another singly linked list at level 2 that skips over three nodes (ie, it links every fourth node), and another singly linked list at level 3 that skips over seven nodes (ie, it links every eighth node). To find the node named *g* we could follow the link at level 2 from the head node to node *d*, then the link at level 1 to node *f*, and then the link at level 0 to get to node *g*. Hence, in theory, we only need to follow 3 links to get to that seventh node.

OK, so much for theory, what about in practice? How would we

find *g*? The algorithm works like this:

1. Set a variable called `LevelNumber` to the highest level of links in our skip list (we assume that we made a note of this through all our inserts and deletes as we built up the skip list).
2. Set a variable called `BeforeNode` to the dummy head node.
3. Compare the node reached by following the forward link at level `LevelNumber` from `BeforeNode` (call this node `NextNode`.)
4. If `NextNode` is the one we want, we're finished.
5. If `NextNode` is less than the one we want, then the latter must be beyond `NextNode`, so set `BeforeNode` to `NextNode`, and continue at step 3.
6. If `NextNode` is greater than the one we want, the node we want is in between `BeforeNode` and `NextNode`. We decrease `LevelNumber` by one (ie, we want to reduce the number of nodes we skip over).
7. If the `LevelNumber` is 0 or greater, we continue at step 3.
8. Otherwise the item we seek is not to be found in the skip list and, if we were to insert it, it would appear in between `BeforeNode` and `NextNode`.

Following this algorithm to find *g* we would start at level 3 and the head node. Follow the link at level 3 from the head node and we get to node *h*. We compare and *h* is greater than *g*. We therefore drop a level and start over. Follow the link at level 2 from the head node and we get to node *d*. Compare. *d* is less than *g* so we advance to node *d*. Follow the link at level 2 again and we get to *h*. Compare, it's larger, therefore we drop a level. Follow the link from *d* at level 1 and we reach *f*. This is smaller so we advance. Follow the link at level 1 and we reach *h* again which is greater. So we drop a level again and follow the link to finally reach *g*.

In doing so, we have followed 6 links and made 6 comparisons. This doesn't sound too hot; after all, if we were using a simple doubly linked list without a binary search we would have followed 7 links and made 7 comparisons.

```
type
  PslNode = ^TslNode;
  TslNodeArray = array [0..pred(MaxSkipLevels)] of PslNode;
  TslNode = packed record
    slnData : pointer;
    slnLevel: longint;
    slnPrev : PslNode;
    slnNext : TslNodeArray;
  end;
function TaaSkipList.Search(aItem : pointer) : PslNode;
var
  Level  : integer;
  Walker : PslNode;
  Temp   : PslNode;
  CmpResult : integer;
begin
  {initialize}
  Walker := FHead;
  Level := MaxLevel;
  {start zeroing in on the item we want}
  while (Level >= 0) do begin
    Temp := Walker^.slnNext[Level];
    if (Temp = FTail) then
      {pretend that the tail's data is greater than our item}
      CmpResult := 1
    else
      {compare the next node's data with our item}
      CmpResult := FCompare(Temp^.slnData, aItem);
    if (CmpResult = 0) then begin
      {if equal then we found the item}
      Result := Temp;
      Exit;
    end;
    if (CmpResult < 0) then begin
      {if less than, then advance the walker node}
      Walker := Temp;
    end else begin
      {if greater than, save the before node, drop down a level}
      dec(Level);
    end;
  end;
  {if we reach this point, the item is not in the skip list}
  Result := nil;
end;
```

➤ *Listing 1: Searching for an item in a skip list.*

However, Figure 1 makes an assumption that I said nothing about: that a link at level n+1 jumps a distance twice that of level n. But why should it? Why not three times as far, or four, or five? Indeed, that is the plan. In our skip list we shall jump four nodes at a time for level 1, 16 (ie, 4x4) for level 2, 64 (ie, 4^3) for level 3, and 4^n for level n.

The reason for choosing four as our multiplier is that we have to balance the need for jumping major distances at high levels versus the length of the slow level 0 search at the end. Four is a good compromise.

Listing 1 shows the search routine for a skip list, implementing this algorithm.

Deep In The Motherlode

The next question we should ask is, to how many levels should we allow the skip list to grow? Think about it for a moment. If we assume that an item we are storing in the skip list is a pointer (much as we did in February's column with linked lists) then nodes at level 0 are at least 12 bytes in size (one data pointer, one forward pointer, one backward pointer), nodes at level 1 are 16 bytes in size (now two forward pointers), at level 2 they're 20 bytes in size and so on. Hence at level n nodes are at least $4n$+12 bytes in size. If we go as far as level 16 (say) then the nodes at that level will be 76 bytes large. Admittedly, they'll be jumping forward 4^{16} nodes (4 billion) at that stage. Which of course should strike you forcibly between the eyes: the machines you and I program on just can't have that many nodes. Win32 allows applications much less than 4 billion *bytes* to play around in (let alone nodes). Hence we can safely limit the levels to a maximum of 16, numbered 0 to 15. At the top level we'll be jumping to the billionth node in front, and I doubt we'll ever get close to that. Note that I kept saying 'at least' in the discussion about node sizes. That's because each node will have to have an extra field: the size of the node or maybe the level for which the node is for. We could easily use a byte for this, but because of the way that 32-bit

processors work, and because of the way the Delphi heap manager works, it'll be just as easy, and better, to use a full integer (or `longint`).

I'm sure that some of you are wondering by now how on earth we are going to build this extremely regular structure through a series of random insertions and deletions. Well, here's the kicker: we don't. We finally come to the final puzzling word in Pugh's title: *Probabilistic*.

The cleverness of Pugh's algorithm is that he realized that it was impossible (or rather, much too long-winded and time-consuming) to build the regular structure, so he proposed building a structure that *on average* approximated to the regular structure. Look back at Figure 1 for an example. In the first part of this regular skip list we have 8 nodes, *a* to *h*. Ignoring *h* for the moment, one of these nodes is at level 2, two at level 1 and four at level 0. In other words, picking a node at random, we can calculate that it is at level 0 with probability 0.5, at level 1 with probability 0.25, at level 2 with probability 0.125 and so on.

Pugh's algorithm for insertion in a skip list replicates these probabilities, so that overall, there are approximately the right numbers of nodes at each level. The results of the algorithm are probabilistic. This means that, *on average*, the probabilistic skip list will work with the same efficiency as the fully 'regular' skip list: some nodes will take longer to find, some a shorter time, but, averaged out, the probabilistic skip list performs the same as its regular cousin.

Many Too Many

So, with this information, we can now describe the insertion algorithm. But first we need to describe how to create a skip list and what data we need to store about it. To create a skip list we allocate a node of level 15 for a dummy head node and a node of level 0 for a dummy tail node. All of the forward pointers in the head node are set to point to the tail node. The tail node's backward pointer is set to point to the head node. We also need to track the maximum level currently used by the skip list (we can set this to zero at creation time). Of course, with all this allocation and stuff, it's crying out for a class implementation, and we'll certainly write one, but we won't go there just yet.

The insertion algorithm can now be shown. There are a total of ten steps.

1. Perform the search algorithm to find the item we are about to insert, with one extra caveat. Every time we need to descend a level, store the value of `BeforeNode` before doing so. We'll end up with a set of values of `BeforeNode`, one for each level (since we've limited the number of levels, we can use a simple array for this, one node per level).
2. If the item is found, raise an error (we'll discuss why in a minute).
3. If the item is not found, we know between which two nodes we have to insert the item. Plus, we know that we reached level zero during the search.
4. Set a variable called `LevelNumber` to zero.
5. Using a random number generator, calculate a random number between 0 and 1.
6. If the number is less than 0.25, increment `LevelNumber`.
7. If `LevelNumber` is less than or equal to the current maximum level for the skip list (or 15) return to step 5.
8. If `LevelNumber` is greater than the current maximum level for the skip list, set this latter value to `LevelNumber`.
9. Create a node of level `LevelNumber` and set its data pointer to our item.
10. Now the fun stuff: we have to insert this node into the links at all levels up to `LevelNumber` (and that's why we stored all those values of `BeforeNode` during the search in step 1). This is merely applying the 'insert after' method for the doubly linked list at level 0, and for the singly linked lists at levels 1 to `LevelNumber` (for a discussion of this, see the February 1999 *Algorithms Alfresco* column).

There are a couple of weird things happening in this algorithm that need a little further explanation. Steps 5, 6, 7 and 8 for example: what's all this about? Well, what's happening here is that we're calculating the level that this new node is to have (how big it is, essentially). In our skip list we have a skip factor of 4. In other words, $\frac{3}{4}$ of the nodes must be level 0 (or to put it another way $\frac{1}{4}$ of the nodes must be level 1 or greater), $\frac{3}{16}$ of the nodes level 1 (or to put it another way $\frac{1}{16}$ of the nodes must be level 2 or greater), $\frac{3}{64}$ of the nodes level 2 and so on. We also don't want to expand the maximum level of the skip list too much: it should only increase by one level at a time (think about the 'regular' skip list again: if only 3 items are inserted in the list, *a*, *b*, *c*, the maximum level is 1, whereas in our probabilistic skip list we could get a maximum level of 15 with a bit of luck). So that's what steps 5 through 8 are doing.

Step 2 also bears some explanation. What it basically says is that a skip list cannot have duplicate items (or rather items that compare equal). Why? Imagine a skip list just containing 42 nodes of value *a*, what does it mean to search for item *a*? Because of the nature of the skip list we'll jump over a whole bunch of them in the first step of the search algorithm to, say, the 35th. We found *a*! We didn't find the first one, or the last, but we did find one. Should we add a few steps to the algorithm to walk backwards until we don't find any more occurrences of *a* (and hence find the first one)? Or forwards for that matter. Some would say that we ought to add them in the order they arrived (so when we insert we should insert at the end of the list of duplicates, and when we search we should find the first). This just makes the algorithm messier and, in my view, the extra complexity is unnecessary.

Presumably, if we want to add duplicate items, we know how to differentiate them, otherwise they truly are the same item. If we can differentiate them then presumably the comparison function

should as well. Ergo, they are no longer duplicates.

Illegal Alien

Having talked about insertion, we need to talk about deletion of a node. This is fairly easy, if long winded. Let's imagine that we want to delete node *d* in Figure 1. If you look at it, we'll need to fix up three forward links (*d* is at level 2) and one backward link. Eventually, we'll have a link at level 2 between the head node and *h*, a link at level 1 between *b* and *f*, and two links between *c* and *e*, one forwards and one backwards. The algorithm goes like this.

1. Find the item we wish to delete by the usual method.

2. Assume we find it at level *i*. Save the node before the one we want to delete as the *ith* item in an array. Set `LevelNumber` to *i*, and the node before in `BeforeNode`.

3. Decrease `LevelNumber` by one.

4. If `LevelNumber` is negative, continue at step 7.

5. Starting at `BeforeNode`, follow the links on level `LevelNumber` until we reach the item again. As we walk the links keep a note of the parent of each node.

6. Store the node before the one we wanted in the array, set `BeforeNode` to this node. Continue at step 3.

7. We now have an array of prior nodes from level *i* down to 0. Perform the usual linked list 'delete after' operations on each level.

Step 5 is guaranteed to work (we are guaranteed to always find the item we want at every level) because a node at level *n* has a link at each level up to *n* pointing to it.

Undertow

Can we write our class now? Ordinarily, yes, but given the discussion we had last month about node managers, should we (can we?) do the same here? A smidgeon more difficult this time because different nodes are different sizes, but the answer is still yes to both questions. Recall the original requirement for a node manager: we were always allocating and freeing nodes of the same size, so we decided that allocating a batch of nodes and then doling them out as required would be faster than using the Delphi heap manager. Our node manager this time will have to track 16 different node sizes instead of one. The first node size is 16 bytes (one data pointer, two pointers, one forward and one back, and a level number), the second 20, the third 24, all the way up to the sixteenth node size of 76 bytes (of which we won't allocate all that many!). When we allocate or free a node we pass an extra parameter detailing which level the node is for. The node manager has an array of free lists, one for each node size (and also an array of pages lists) and each list is managed independently. The code in Listing 2 shows the node manager for skip lists. You can compare this to February's versions.

There's one small problem, though, with the skip list node manager that's not so readily apparent with the linked list managers from last month. The problem is one of thrashing and becomes obvious when you have millions of nodes. Win32 will utilize a swap file when real memory gets scarce, in other words, memory pages will get swapped to and from disk. The thing about the skip list node manager is that neighboring nodes in the skip list will be from different memory pages. If you sequentially walk the skip list from start to finish, you will come across nodes of different sizes (and hence from different memory pages) as you go, causing page swaps to occur. There's not a lot we can do about this (and indeed, if we are using millions of nodes, the items for those nodes will be on different memory pages anyway) apart from using the standard Delphi heap manager (which also will produce nodes that will suffer from thrashing in low memory conditions). The code on the diskette allows you to switch heap management from the node manager to the Delphi heap manager.

➤ *Listing 2: The node manager for skip lists.*

```
const
  NodeSize : array [0..pred(MaxSkipLevels)] of integer =
    (16, 20, 24, 28, 32, 36, 40, 44,
     48, 52, 56, 60, 64, 68, 72, 76);
type
  PslnmPage = ^TslnmPage;
  TslnmPage = packed record
    slnmpNext  : PslnmPage;
    slnmpSize  : longint;
    slnmpNodes : TByteArray;
  end;
var
  slnmFreeList : TslNodeArray; {a free list per node size}
  slnmPageList : PslnmPage;
procedure slnmFreeNode(aNode : PslNode; aLevel : integer);
begin
 {$IFDEF UseNodeManager}
  {add the node to the top of the correct free list}
  aNode^.slnNext[0] := slnmFreeList[aLevel];
  slnmFreeList[aLevel] := aNode;
 {$ELSE}
  FreeMem(aNode, NodeSize[aLevel]);
 {$ENDIF}
end;
procedure slnmAllocPage(aLevel : integer);
var
  NewPage : PslnmPage;
  i       : integer;
  PageSize: integer;
  Offset  : integer;
begin
  {get a new page}
  PageSize := sizeof(pointer) + {the slnmpNext field}
    sizeof(longint) + {the slnmpSize field}
    (PageNodeCount * NodeSize[aLevel]); {the nodes}
  GetMem(NewPage, PageSize);
  NewPage^.slnmpSize := PageSize;
  {add it to the current list of pages}
  NewPage^.slnmpNext := slnmPageList;
  slnmPageList := NewPage;
  {add all the nodes on the page to the free list}
  Offset := 0;
  for i := 0 to pred(PageNodeCount) do begin
    slnmFreeNode(@NewPage^.slnmpNodes[Offset], aLevel);
    inc(Offset, NodeSize[aLevel]);
  end;
end;
function slnmAllocNode(aLevel : integer) : PslNode;
begin
 {$IFDEF UseNodeManager}
  {if the free list is empty, allocate a new page of nodes}
  if (slnmFreeList[aLevel] = nil) then
    slnmAllocPage(aLevel);
  {return the first node on the free list}
  Result := slnmFreeList[aLevel];
  slnmFreeList[aLevel] := Result^.slnNext[0];
 {$ELSE}
  GetMem(Result, NodeSize[aLevel]);
 {$ENDIF}
  Result^.slnLevel := aLevel;
end;
```

```
function TaaSkipList.Delete : pointer;
var
  i, Level    : integer;
  Temp        : PslNode;
  BeforeNodes : TslNodeArray;
begin
  {search for the item and create the BeforeNodes array}
  slSearchPrim(FCursor^.slnData, BeforeNodes);
  {the only valid before nodes are from the skip list's
   maximum leveldown to this node's level; we need to get
   the before nodes for the others}
  Level := FCursor^.slnLevel;
  if (Level > 0) then begin
    for i := pred(Level) downto 0 do begin
      BeforeNodes[i] := BeforeNodes[i+1];
      while (BeforeNodes[i]^.slnNext[i] <> FCursor) do
        BeforeNodes[i] := BeforeNodes[i]^.slnNext[i];
    end;
  end;
  {patch up the links on level 0 - doubly linked list}
  BeforeNodes[0]^.slnNext[0] := FCursor^.slnNext[0];
  FCursor^.slnNext[0]^.slnPrev := BeforeNodes[0];
  {patch up the links on the other levels - all singly
   linked lists}
  for i := 1 to Level do begin
    BeforeNodes[i].slnNext[i] := FCursor^.slnNext[i];
  end;
  {reset cursor, dispose of the node}
  Result := FCursor^.slnData;
  Temp := FCursor;
  FCursor := FCursor^.slnNext[0];
  slnmFreeNode(Temp, Level);
  {we now have one less node in the skip list}
  dec(FCount);
end;
```

```
procedure TaaSkipList.Insert(aItem : pointer);
var
  i, Level    : integer;
  NewNode     : PslNode;
  BeforeNodes : TslNodeArray;
begin
  {search for the item and create the BeforeNodes array}
  if slSearchPrim(aItem, BeforeNodes) then
    raise Exception.Create(
      'TaaSkipList.Insert: duplicate item');
  {calculate the level for the new node}
  Level := 0;
  while (Level <= MaxLevel) and (Random < 0.25) do
    inc(Level);
  {if we've gone beyond the maximum level, save it}
  if (Level > MaxLevel) then
    inc(FMaxLevel);
  {allocate the new node}
  NewNode := slnmAllocNode(Level);
  NewNode^.slnData := aItem;
  {patch up the links on level 0 - a doubly linked list}
  NewNode^.slnPrev := BeforeNodes[0];
  NewNode^.slnNext[0] := BeforeNodes[0].slnNext[0];
  BeforeNodes[0].slnNext[0] := NewNode;
  NewNode^.slnNext[0]^.slnPrev := NewNode;
  {patch up the links on the other levels - all singly
   linked lists}
  for i := 1 to Level do begin
    NewNode^.slnNext[i] := BeforeNodes[i].slnNext[i];
    BeforeNodes[i].slnNext[i] := NewNode;
  end;
  {we now have one more node in the skip list}
  inc(FCount);
end;
```

➤ *Listing 3: Insertion into and deletion from a skip list.*

So we can now finally write our skip list class. I won't show you all the code here; that's what the diskette is for, after all. However, Listing 3 does show the insertion and deletion code since that's the most interesting bit, the rest being just housekeeping. The `slSearchPrim` routine (which is not shown) merely performs a search for an item based on the code in Listing 1 and builds up a `BeforeNodes` array of prior nodes as it drops from level to level.

Throwing It All Away

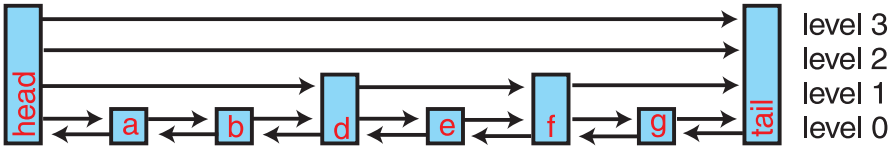
Having described the standard probabilistic skip list, let's have a look at the variant I found in a paper called *Deterministic Skip Lists* by Munro, Papdakis and Sedgewick. They describe two variants but I'll be showing only one, the one they called the 1-2 skip list.

In Pugh's skip list, he got around the problem of not being able to create the regular skip list by providing the approximate balance between node sizes in a probabilistic way. In a 1-2 skip list, Munro, Papdakis and Sedgewick provide the approximate balance in an algorithmic way. In their 1-2 skip list, they relax the regular rule slightly by saying that between any two nodes at level *n* there must be either one or two nodes at level *n*-1 (obviously this doesn't apply to level 0 nodes). Hence the name. The regular skip list in Figure 1 can be viewed as a 1-2 skip list because in between two nodes at a particular level there is exactly one node at the next level down.

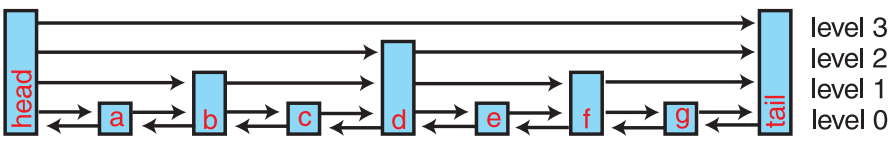
The search algorithm for a 1-2 skip list remains the same. After all, it doesn't depend on how the skip list was constructed, it just expects the nodes to be sorted and assumes that some of them are larger than others in order that it can do the rapid skips. It's only during insertion and deletion that we need worry about maintaining this 1-2 rule.

Let's take insertion first. How do we maintain the rule? Firstly we search for where the node should be inserted. Create a level 0 node at that point. This may cause the rule to be broken, since there may now be three level 0 nodes in a row. No problem. Grow the middle node of the three to a level 1 node. This may, in its turn, cause a violation of the rule at level 1. Again no problem, just grow the middle node of those level 1 nodes to level 2. And so on and so forth. Eventually the grown node will not cause a violation of the rule and everything is all right again. I'm sure you can see that we may have grown the maximum level of the skip list if we manage to go all the way up. Inserting *c* into the 1-2 skip list of Figure 2 will cause the *b* node to grow to level 1, which in turn causes the *d* node to grow to level 2. Figure 3 is the result.

➤ *Figure 2: A 1-2 skip list.*



➤ *Figure 3: Resulting 1-2 skip list after inserting c.*



Those of you who know about 2-3 binary search trees will spot a similarity here: this is the skip list equivalent of insertion in a 2-3 tree. For the rest of you, we'll be talking about them in a future article on binary trees.

From this small description we have a couple of problems to consider. The first is obvious: growing the node. Actually this won't be too bad with our node manager concept: save the data pointer, free the node that's now too small and allocate one that's one level larger and restore the data pointer.

The second problem is a little more intricate. If we have grown a node, we have to patch up the linked list at the level of the grown node. To do this we need to know the prior node at that level. That's not too bad, after all we were doing similar things for the probabilistic skip list insertion. The problem, though, seems to be compounded: after growing a node and patching the linked list at that level we need to count the number of consecutive nodes at that level. For example, after inserting *c* in Figure 2 we need to count the number of consecutive level 0 nodes. Ignoring for the moment that we are at level 0 (and hence have back pointers) we need to count from *a* onwards. But how do we get to *a* to start the count? We know where *b* is, but we have no idea how to get to *a* (or even if it's there).

The way round this is to store all the before nodes at each level as we are searching for where to insert the item. To count the number of consecutive nodes after growing a node to a given level, get the before node for the *next* level and then count the nodes from that point at the level we're interested in. For our example of inserting *c*, we'd find the before node at level 1 (the head node) and then count the nodes from there for the level 0 nodes, *a b c*. We grow *b* to level 1. Get the before node for level 2 (the head node again) and count the level 1 nodes, *b d f*. We grow *d* to level 2. Get the before node for level 3 (the head node yet again for our simple example). And then count the nodes at level 2 from there, *d*. We're done since we verified the rule at all levels.

Errata

I was just completing this month's column when I had two email messages from Father Christmas. He was forwarding a couple of messages from readers of December 1998's column who'd tried to use my implementation of the shuffle algorithm. They'd found the same bug and Father Christmas had annotated each forwarded message with an expressive Ha!

The shuffle routine presented in that article declared a local variable on top of an untyped parameter by using the absolute keyword:

```
A : PByteArray absolute aArray;
```

Well, both Peter Below of TeamB and Simon Mentha discovered that I'd blown it good and proper. I couldn't even blame the bug on the Man in Red either, since I'd expressly said in the article that I'd coded it. Anyway, the type of A was supposed to be TByteArray, not PByteArray. Although the routine would compile, it would never run; at least not without a vile Access Violation. I looked back on my test programs and discovered that I'd done the unforgivable: I'd 'tidied up' the code after testing it, introduced the bug and never retested. Argh! Sorry about that.

To make matters worse, Simon pointed out that the routine didn't produce a very random looking shuffle. Another bug which I fixed, the routine didn't follow the algorithm I'd outlined. All in all, I'm afraid this was not one of my better efforts. The fully tested and debugged routine is on this month's disk.

Now Simon can go back to writing his program to select the six winning lottery numbers!

I'd like to thank them both for their messages and encourage you to email me or our Esteemed Editor with thoughts, requests and bug reports about *Algorithms Alfresco.*

Just A Job To Do

Deletion is a little more complex. We find the node to delete. Delete it. This may cause a violation of the rule in two ways. Firstly we may delete a larger node that is separating two sequences of lower nodes. Imagine that nodes *a*, *b*, *d* and *e* are all level 0 nodes and node *c* is a level 1 node. We delete node *c*. This leaves us with four level 0 nodes in sequence, a violation of the 1-2 rule and one of them must be grown (note that this growing node will *not* cause a violation of the 1-2 rule since, in essence, it's replacing a node of that level, its neighbor, which has just been deleted).

Secondly we may delete a smaller node that is separating two larger nodes. Imagine nodes *a* and *c* are level 1 nodes and node *b* is a level 0 node separating them, which we delete. The 1-2 rule does not allow two nodes at higher levels to be next to each other, so one of them must be shrunk. This may in turn cause a violation of the rule (three in a row) so we need to grow one of them in a pattern that you can recognize from the insertion case.

The nice node manager that we used in the probabilistic skip list isn't so useful here. Our skip factor has reduced from 4 to 2 (or maybe 3). The maximum number of possible node levels is twice as many, from 16 to 32. And although having a node manager managing 32 different free lists is not impossible, it's just a lot more complex. We could reduce the number of free lists, at the expense of some wastage, by having level 0 and 1 nodes the same size, and level 2 and 3 nodes the same size and so on, halving the number of free lists.

The advantage of the 1-2 skip list over its probabilistic sibling is that

its behavior is much more stable. After all if you think about it in a hand-waving fashion (ie, I'm not going to provide formulae!), Pugh's original skip list only guarantees that *on average* things will work nicely. Given that we are logically throwing dice here, we may have a run of bad luck and have a long run of nodes at the same level causing search times to increase. If, horror, we have a bad random number generator, the efficiency of the skip list will deteriorate badly. The 1-2 skip list guarantees better behavior and a more regular look at the expense of a much deeper skip list and a more complicated algorithm for insertion and deletion.

However, I must say that this is probably only a theoretical advantage. I've seen skip lists as described by Pugh in several data structures books. I have yet to see the 1-2 skip list so described.

That's All

My own tests have shown that skip lists are a good alternative to balanced binary trees for storing sorted data. However, since their behavior can only be described statistically rather than rigorously, they are not necessarily the structure of choice in a general sense. After all, the implementers of the C++ Standard Template Library for various compilers usually code their associative maps as red-black binary search trees. Having said that, I certainly would use skip lists in situations where I needed to iterate through the items in sorted order, or where I didn't have too many deletions.

And with that we come to the end of another *Algorithms Alfresco* article. Thanks for reading!

Julian Bucknall programs, acts, goes to the gym but doesn't do much skipping. Having lived in the mountains for 6 years, he'd now like a home by the sea. He can be reached at julianb@turbopower.com