The Leader Of The Pack

Priority queues and heapsort

by Julian Bucknall

I was reviewing last month's article in preparation to writing this one, when I was suddenly struck by something. You know the bad photocopies of bad epigrams that some people put up on their cubicle walls (the ones that Scott Adams lampoons in *Dilbert*)? Well, I was suddenly struck by one I remember from way back when. It was 'Think Ahead', but this had a little typographic flair: the 'Think' and 'Ah' were in large type, but the remainder, 'ead', had to be squeezed into the remaining space in a very condensed type to fit. I suddenly felt the same way. To continue with my discussion of graphs and graph algorithms, I needed to use a priority queue and I think, to be fair, it would be best for all of us if I devoted some time to this interesting data structure.

So, dear reader, please excuse the slight digression from my graph series. This month we'll talk about the priority queue, which, if nothing else comes of our talks on graphs, you'll be able to use as a data structure in your code.

So, then, a priority queue. Something for queuing priorities?

First Cut Is The Deepest

Let's review quickly what an ordinary queue is. A queue is a data structure that has two main operations: add an item to the queue, and retrieve the oldest item from the queue. Generally the queue does not allow you to fiddle with other items in the queue: they're deemed to be inaccessible until they're at the front of the queue (that is, the oldest), nor is it allowable to insert an item in the queue at some random place, or at the front (ie, to make it the oldest). The queue is known as a FIFO structure: First In, First Out (compared to a stack, which is a LIFO structure: Last In, First Out).

In implementing a queue, we don't bother to give each item the date and time of its arrival in the queue just so that we can find the oldest: the queue keeps the items in order of their arrival. Just like the checkout at a supermarket, in fact. The customers wait patiently (sometimes) in line, until their turn with the checkout person arrives. They tag on to the end of the queue and are processed in order from the front. Queue jumpers are not allowed (at least not in my supermarket they're not).

All fine and dandy, and the queue is an important data structure in its own right. However, it has a limitation in that items are processed in the order of their arrival. Suppose we want to process items in some other order: in other words, a queue that still has the 'add an item' operation, but whose second operation is not 'retrieve the oldest', but 'retrieve the largest (or smallest)'. We want to replace the simplistic age ordering with another ordering criterion.

The Legend Of Xanadu

This new data structure is a priority queue. It has two main operations: add an item (as before), and retrieve the item with the largest priority (we assume that each item has an associated priority). What do I mean by *priority* in this context? Well, it can be anything. Classically, it's usually a numeric value that denotes the item's priority in some process. I'm thinking here of print queues in operating systems (or job queues, or threads in a multithreaded environment). Each print job is assigned a priority, a value that indicates how important that particular print job is. High priority print jobs would need to be processed before low priority print jobs. The operating system finishes off a particular print job, and then goes to the print queue and from it retrieves the print job with the highest priority. As work is done in the operating system, other print jobs get added to the print queue with various priorities.

Going back to the supermarket analogy, the deli counter in my supermarket operates a numbering scheme. You grab a number on a piece of paper from a dispenser and then wait around in some amorphous mass of customers until your number is called and you can go up and order your potato salad or whatever. This is almost the same as the traditional queue where the numbers represent how long you've been waiting, but note that we, the customers, don't have to wait in a nice orderly line like a standard queue. Because we have a priority (a number on a piece of paper) we can shuffle around just as we like.

Do note that the value we are using as 'priority' doesn't need to be a classic priority number in this vein. It can be any value, just so long as the queue is able to determine the item with the largest value. In other words, the values have to have some meaningful ordering. An example: the priority could be a name, and the ordering the standard alphabetic order. So the retrieval operation of getting the item with the largest priority would instead become getting the item earliest in alphabetic sequence (ie, A... before B... etc). Imagine the uproar at my deli counter if that was the definition of 'priority'! Certainly the Young family would never shop there. Still, when programming, the point is that we can select the retrieval ordering that suits our needs.

The Carnival Is Over

Enough chatter, let's consider how to code a priority queue. The

queue must be able to firstly store an arbitrary number of items, secondly add an item with associated priority to the queue, and thirdly identify and retrieve the item with the largest priority.

Traditionally, the first attribute (storing an arbitrary number of items) has been implemented by a linked list. This structure gives us an extensible container without too much overhead or too many efficiency constraints. Let's be different, however, and use a somewhat under-appreciated Delphi data structure: the `TList`. It can be grown, accessing items is fast, but efficiency suffers a little when deleting an item from the middle of the list. Since we won't be using this first design anyway (we're just illustrating the *concept* of a priority queue at the moment, I'll be revealing a better algorithm shortly), this last problem won't cause us too much heartache. So we'll use an internal `TList` object to store the items we get.

The next attribute (add an item to the queue) is easy with `TList`: just call the `Tlist`'s `Add` method. We'll take the view that the items we add to the priority queue will be objects of some description, with their priority as a property of the object. If we want the priority to be separate from the objects (or pointers, or integers, or whatever) we add to the priority queue then the details just become a little more involved (create a super-object that contains the original object and priority and add that to the `TList` instead) and will just obscure what we're trying to show.

The third attribute (finding the highest priority and returning the associated object, removing it from the priority queue in the process) is a teensy bit more involved but still quite simple. Essentially we iterate through the items in the `TList` and for each item we see whether its priority is larger than the largest priority we've found so far. If it is, we take note of the index of the item in the `TList` with this newer largest priority, and move on to the next item. After we've checked all of the items in the `TList`, we know which is the largest (we took note of its index) and so we just remove it from the `TList` and pass it back.

The code in Listing 1 shows this simple priority queue. It uses a comparison event that you set up in order to determine whether an item's priority is greater than another's. The priority queue therefore doesn't need to know how to compare priorities (and hence whether they're numbers or strings or something else): it just calls the comparison event, passing the two items whose priorities it needs to compare. Note also that the queue doesn't need to know *what* the items are, it just stores them, so we just declare the queue to use pointer variables and typecast as necessary.

Pretty good, I'd say. From a discussion of a concept to a working data structure in a couple of pages. Before we get too carried away, let's think about the efficiency of our design. Firstly, adding an item will get done in constant time (ignoring time lost due to growing the `TList`). In other words, adding an item to an empty queue or to a queue containing thousands of items will take roughly the same amount of time. In computer science speak, we say that the algorithm is O(1) (or big-Oh of 1), that is, no matter how many items there are, we consider the time taken to be constant.

Now let's look at the opposite operation: removing an item. Here, we need to read through *all* the items in the `TList` to find the one with the largest priority. The time taken for this operation with a queue with one item would obviously be less than the time for a queue with thousands of items. The time taken, in fact, is proportional to the number of items or, in computer science speak, O(N). When we remove an item, we replace it with the last item in the `TList`, so that we don't have to do any reorganization of the `TList`.

So we have a structure that implements a priority queue in which adding an item is an O(1)

➤ *Listing 1: Simple priority queue using TList to store items.*

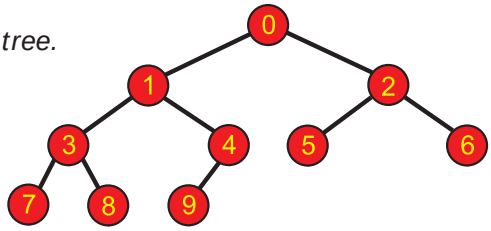
```
type
  TaaPriorityQueueA = class
    { Priority queue that's fast at insertion slow at retrieval}
    private
      pqCompare : TaaItemPriorityCompare;
      pqList    : TList;
    protected
      function pqGetCount : integer;
    public
      { Create priority queue}
      constructor Create(aCompareFn:
        TaaItemPriorityCompare);
      { Dispose: items remaining are NOT freed}
      destructor Destroy; override;
      { Add an item}
      procedure Add(aItem : pointer);
      { Remove and return item with largest priority}
      function Remove : pointer;
    end;
constructor TaaPriorityQueueA.Create(aCompareFn :
  TaaItemPriorityCompare);
begin
  inherited Create;
  pqCompare := aCompareFn;
  pqList := TList.Create;
end;
destructor TaaPriorityQueueA.Destroy;
begin
  pqList.Free;
  inherited Destroy;
end;
procedure TaaPriorityQueueA.Add(aItem : pointer);
begin
  pqList.Add(aItem);
end;
function TaaPriorityQueueA.Remove : pointer;
var
  Inx     : integer;
  PQCount : integer;
  MaxInx  : integer;
  MaxItem : pointer;
begin
  PQCount := pqList.Count;
  if (PQCount = 0) then
    Result := nil
  else if (PQCount = 1) then begin
    Result := pqList[0];
    pqList.Clear;
  end else begin
    MaxItem := pqList[0];
    MaxInx := 0;
    for Inx := 1 to pred(PQCount) do
      if (pqCompare(pqList[Inx], MaxItem) > 0) then begin
        MaxItem := pqList[Inx];
        MaxInx := Inx;
      end;
    Result := MaxItem;
    pqList.Delete(MaxInx);
  end;
end;
```

operation and removing it is an O(N) operation. For small numbers of items this structure is perfectly acceptable and usable.

I Like It

But, and this is a fairly big but, we can easily do better. I'm sure you can think of one efficiency improvement straight away: maintain the `TList` in priority order, keep it sorted. If you think about it, this means we shift the work of the queue from item removal to item insertion. When we add an item we find its correct place inside the `TList`, which is after all the items with lower priority and before all those with higher priority. If we do this extra work during the add phase, the `TList` will have all the items in priority order and hence, when we remove an item, all we need to do is to delete the last item. Pretty simple, huh? In fact, removal becomes an O(1) operation (we know exactly where the item with the largest priority is, it's at the end, so removing it doesn't depend on how many items there are).

Calculating the time required for insertion in this sorted `TList` is a little more involved. The simplest way to think of how it's done is to think of it as an insertion sort (see *Algorithms Alfresco* for September 1998): grow the `TList` by one item, and then move items along by one into the spare hole (like beads on a thread) starting from the end of the `TList`. You stop when you reach an item that has a priority less than the one you are trying to insert. You then have a 'hole' in the `TList` where you can put the new item. If you think about this for a moment, on average you'd move *n*/2 items for *n* items in the `TList`. Hence insertion is an O(N) operation (the time taken is again proportional to the number of items in the queue), although with this improvement the time taken would be somewhat less than the previous implementation. Listing 2 shows how these two operations are coded with this kind of internal structure.



➤ *Figure 1: A complete binary tree.*

We've now moved from fast insertion and slow deletion to slow insertion and fast deletion. Can we do better? Yes, we can, by abandoning the `TList` and moving to another data structure entirely: the binary search tree. Here, insertions and deletions are both O(log N) operations, in other words the time taken for both item insertion and deletion are proportional to the logarithm of the number of items in the structure. I won't go into this structure now because the binary tree suffers from one big problem: you need to worry about balancing it to maintain its insertion and deletion time properties. There are a few balancing algorithms out there (eg red-black, AVL and splaying) and they deserve articles of their own. Of course I needn't worry too much about them here because I have a better structure up my sleeve anyway.

With A Little Help From My Friends

In a binary search tree, the nodes are arranged so that for every node, it is greater than its left child and less than its right child. This is known as strict ordering. There is a lesser ordering called the heap property that can be applied to a binary tree which just states that any node in the tree must be greater than both its children. Note that the heap property does not state that the left child is less than the right child (for then the heap property becomes a strict ordering again), but just that the parent is greater than both its children. There's one more thing to the heap property: the tree to which it applies must be *complete*. A binary tree is called complete when all its levels are full, except for possibly the last. A complete tree is as balanced as it can be. Figure 1 shows an example.

So how does this help us in our quest for the perfect priority queue structure? Well, it turns out that the insertion and deletion operations on a binary tree with the heap property are O(logN), but they are significantly faster than the same operations in a binary search tree. (By the way, let's shorten 'binary tree with the heap property' into 'heap', it'll save on ink. Don't confuse this heap with Delphi's memory heap though.) This is one instance where the big-O notation falls short, it doesn't give any quantitative feel for which of two operations with the same big-O value is actually *faster*.

Anyway, how do we insert and remove from a heap? To do an insertion we perform a *bubble-up* operation, with removal we perform a *trickle-down* operation. Let's see what these cute terms actually mean.

Insertion first. To insert an item into a heap, we add it to the end of the heap (in the only place that continues to maintain the completeness attribute, in Figure 1 that would be the right child of the 4 node). At this point the heap property of the tree may be violated (the new node may be larger than

➤ *Listing 2: Priority queue Add and Remove using a sorted TList.*

```
procedure TaaPriorityQueueB.Add(aItem : pointer);
var Inx : integer;
begin
  pqList.Count := pqList.Count + 1; {increment number of items in the list}
  Inx := pqList.Count - 2;          {find where to put our new item}
  while (Inx >= 0) and (pqCompare(pqList[Inx], aItem) > 0) do begin
    pqList[Inx+1] := pqList[Inx];
    dec(Inx);
  end;
  pqList[Inx+1] := aItem;  {put it there}
end;
function TaaPriorityQueueB.Remove : pointer;
begin
  Result := pqList.Last;
  pqList.Count := pqList.Count - 1;
end;
```

its parent), so we need to patch the tree up. If this new child node is greater than its parent, swap it with the parent. Similarly our new node may also be greater than its new parent, so it needs to be swapped again. In this manner we continue working our way up the heap until we reach a point where our new node is no longer greater than its parent, or we've reached the root of the tree. We've now ensured that all nodes are greater than both their children again and the heap property has been restored. As you can see, we *bubble-up* the new node until it reaches its correct place (the root or just under a node that is larger than it).

If you think about it, the heap property ensures that the largest item is at the root (if it wasn't, it would have a parent smaller than itself, hence the heap property has been violated). We can therefore move onto removal: the item we want is at the root. The theory would seem to indicate that we delete the root node, passing the item back as a result, but this would leave us with two separate subtrees: an utter violation of the completeness attribute of our heap. Instead, we replace the root node with the last node of the heap and thereby ensure that the tree remains complete. But, again, we've probably violated the heap property: the new root may be smaller than one or both of its children. So we find the larger of the node's two children and swap it over. Again, this new position may violate the heap property, so we verify whether it's again smaller than one (or both) of its children, and repeat the process. Eventually, we'll find that the node has sunk or *trickled down* to a level where it is greater than both its children, or it's now a leaf with no children. Either way we've again restored the heap property.

Pretty simple, huh? Well, unfortunately using a tree like this is pretty wasteful of space. For every node we have to maintain three pointers: one for each child, so that we can trickle down the tree, and one for the parent, so that we can bubble up. Every time we swap nodes around we run the risk of having to update umpteen different pointers for numerous nodes. So, the usual trick is to leave the nodes where they are and just swap the items around inside the nodes instead.

Poetry In Motion

There is, however, a simpler way. A complete binary tree can be easily represented by an array. Look at Figure 1 again. Notice how the node numbers are sequential and have been allocated to the nodes from top to bottom, level by level, and from left to right on each level. This top/bottom, left/right annotation provides a nice mapping from node to element number in an array. There are no 'holes' in the mapping: the fact that the heap is a complete tree helps there. Now have a look at the numbers of the children for each node. The children for node 0 are 1 and 2 respectively. The children for node 3 are 7 and 8, for node 5 they are 11 and 12. Notice any pattern? In fact the

```
procedure TaaPriorityQueue.Add(aItem : pointer);
begin
  {add extra space at the end of the queue}
  pqList.Count := pqList.Count + 1;
  {now bubble it up as far as it will go}
  pqBubbleUp(pred(pqList.Count), aItem);
end;
procedure TaaPriorityQueue.pqBubbleUp(
  aFromInx : integer; aItem : pointer);
var ParentInx : integer;
begin
  { while item under consideration is larger than its
    parent,swap it with its parent and continue from its
    new position. NB: the parent for child at index N is at
    (N-1) div 2 }
  ParentInx := (aFromInx - 1) div 2;
  {while item has a parent and it's greater than parent...}
  while (aFromInx > 0) and
    (pqCompare(aItem, pqList[ParentInx]) > 0) do begin
    {move our parent down the tree}
    pqList[aFromInx] := pqList[ParentInx];
    aFromInx := ParentInx;
    ParentInx := (aFromInx - 1) div 2;
  end;
  {store our item in the correct place}
  pqList[aFromInx] := aItem;
end;
procedure TaaPriorityQueue.pqTrickleDown(aFromInx :
  integer; aItem : pointer);
var
  ChildInx  : integer;
  ListCount : integer;
begin
  { while item under consideration is smaller than one of
    its children, swap it with larger child and continue
    from its new position. NB: children for parent at index
    N are at (2N+1) and 2N+2 }
  ListCount := pqList.Count;
  {calculate left child index}
  ChildInx := succ(aFromInx * 2);
  {while there is at least a left child...}
  while (ChildInx < ListCount) do begin
    { if there is a right child, calculate the index of the
      larger child}
    if (succ(ChildInx) < ListCount) and
      (pqCompare(pqList[ChildInx], pqList[succ(ChildInx)])
      < 0) then inc(ChildInx);
    {if item is >= larger child, we're done }
    if (pqCompare(aItem, pqList[ChildInx]) >= 0) then
      Break;
    { otherwise move the larger child up the tree, and move
      our item down the tree and repeat}
    pqList[aFromInx] := pqList[ChildInx];
    aFromInx := ChildInx;
    ChildInx := succ(aFromInx * 2);
  end;
  {store our item in the correct place}
  pqList[aFromInx] := aItem;
end;
function TaaPriorityQueue.Remove : pointer;
begin
  Result := pqList[0];  {return the item at the root}
  { replace the root with the child at the lowest, rightmost
    position, and shrink the list}
  pqList[0] := pqList.Last;
  pqList.Count := pqList.Count - 1;
  {now trickle down the root item as far as it will go}
  if (pqList.Count > 0) then
    pqTrickleDown(0, pqList[0]);
end;
```

➤ *Listing 3: Priority queue Insert and Remove using the heap algorithm.*

children for node *n* are nodes 2*n*+1 and 2*n*+2, and the parent for node n is |(*n*-1)/2| (where the || operator means take the integer part of the bit in between the bars: in Delphi we usually replace the / operator with the `div` operator, and the result is always an integer). Suddenly we have a simple way of implementing a heap with an array and are able to work out where the children and parent of a node are in the array. And furthermore, *we can use a* `TList` *again!* Listing 3 shows this new code for insert and remove. Rather than describe what's going on, I leave you to follow the description of bubble-up and trickle-down and check the code and comments yourself.

In the literature, nodes are usually counted from one instead. This makes the arithmetic a little easier: node *n*'s children are at 2*n* and 2*n*+1 and its parent is at |*n*/2|. We'll stick with our original formulae though, since we're using `TList`.

Having obtained a heap implementation of a priority queue, we can observe that the heap can in fact be used as a sorting algorithm: add a bunch of items all in one go to the heap and then pick them off one by one in the correct order. (Note that, as written, the items are picked off in reverse order, largest first, but with a quick change of the compare method, we can get them in ascending order instead. In fact, the heap we have developed so far is known as a max-heap, the largest is picked off first, and one that works in reverse order is known as a min-heap, the smallest is picked off first.) Sorting with heaps, or more strictly with the heap algorithm, is known as... heapsort. If you remember from the September 1998 *Algorithms Alfresco* column, heapsort was one of the sorts I didn't discuss, so let's take the opportunity to deal with it now and introduce it a little more formally.

Walkin' Back To Happiness

Let's play around with the heap, at least with regard to sorting. The algorithm I've given so far is this: assume we have a min-heap, add all the items to it, and then retrieve them one by one. If the items were held in an array (or a `TList`) in the first place, this algorithm would mean that all the items would be copied from one array to another, and then copied back. Is there any way to order the items into a heap *in situ* without having to have a separate work array to hold them? In other words, can we make an existing array into a heap by 'applying' a heap structure to it? Amazingly enough, we can, and furthermore we can do so in O(N) time, rather than the O(NlogN) time required by adding the items one by one to a separate heap. We use an algorithm called Floyd's algorithm.

We start out with the parent of the rightmost child node (ie, the node furthest to the right on the last level of the heap). Apply the trickle-down algorithm to this parent. Select the node to the left of the parent on the same level (it'll be a parent as well, of course). Apply the trickle-down algorithm again. Keep on moving left, applying the trickle-down algorithm, until you run out of nodes. Move up a level, to the rightmost node. Continue the same process from right to left, going up level by level, until you reach the root node. At this point the array has been ordered into a heap and we could start peeling off items one by one in the usual manner. Listing 4 extends our heap class to enable another `TList` created externally to replace the one auto-created by the class, and in doing so Floyd's algorithm is applied.

Having ordered an array into a heap, what then? Peeling off the items one by one still means we need somewhere to put them in sorted order, presumably some auxiliary array. Or does it? Think about it for a moment. If we peel off

the largest item, the heap size reduces by one, *leaving space at the end for the item we just obtained*. In fact, the algorithm to remove an item from a heap requires the lowest, rightmost node to be copied to the root before being trickled down, so all we need to do is to swap the root with the lowest, rightmost node, reduce the count of items in the heap, and then apply the trickle-down algorithm. Continue doing this until we run out of items in the heap. What's left is the items sorted in the original array.

And that is heapsort. Heapsort is important for a couple of reasons. It's an O(NlogN) algorithm, so it's fast. Heapsort is also just as fast in both the general case and the worst case. Compare this with quicksort: in the general case quicksort is faster, but quicksort can easily be tripped up by an already sorted set of items, causing it to crawl: it becomes an $O(N^2)$ algorithm, unless we apply some algorithm improvements. In this worst case, heapsort is better.

Listing 5 shows the heapsort routine. There's something I didn't point out before which I should. If you are using a max-heap, you retrieve the items in reverse order, from largest to smallest. However, if you are using a max-heap to do a heapsort, you get the items sorted in ascending order, *not* in reverse order. With a min-heap you'd peel items off in ascending order, but you'd heapsort in descending order. Something to be aware of.

```
procedure TaaPriorityQueue.pqMakeIntoHeap;
var
  Inx : integer;
begin
  { starting from lowest, rightmost parent, trickle down and then continue with
    rest of parents from right to left, bottom to top. Rightmost parent is
    parent of last item. This is ((count-1)-1) div 2 }
  for Inx := ((pqList.Count - 2) div 2) downto 0 do
    pqTrickleDown(Inx, pqList[Inx]);
end;
```

➤ *Listing 4: Floyd's algorithm.*

Reach Out I'll Be There

Having been briefly diverted onto heapsort, we should return to priority queues and finish off this article. Before we made our detour, we had written a priority queue based on a max-heap. Our original specification for the priority queue had defined two basic operations: adding an item to the queue, and removing the one with the largest priority. We've seen that the heap is very efficient at both these operations. Are there any more operations that make sense for the priority queue? Here's a good one: changing the priority of an item.

Right from the start, this is difficult. Consider it for a moment. The priority queue class would be given an item that's already in the queue somewhere (one that presumably has just had its priority changed), and the queue must restore the heap property. How on earth do we find the item in the queue? This is one place where the 'loose' sorting in the queue works against us. In a binary search tree we could use a binary search with the old priority and find the item quite easily. But in a heap? We could obviously use a sequential search (start at the beginning of the array and go through it item by item looking for the one we want) but that's pretty slow.

OK, leave the difficult bit for a moment and assume we have a way of efficiently finding the item

➤ *Listing 5: Heapsort.*

```
procedure AAHeapSort(var aItemArray : TList;
  aLeft, aRight : integer; aLessThan : TaaLessFunction);
var
  Inx      : integer;
  FromInx  : integer;
  ChildInx : integer;
  ListCount: integer;
  Item     : pointer;
begin
  {if there's nothing to do, do it}
  ListCount := aRight - aLeft + 1;
  if (ListCount <= 1) then
    Exit;
  {first, turn array into a heap; this is complicated by the
   fact that all our indexes are offset by aLeft}
  for Inx := ((ListCount - 2) div 2) downto 0 do begin
    Item := aItemArray[Inx+aLeft];
    FromInx := Inx;
    ChildInx := succ(FromInx * 2);
    {while there is at least a left child...}
    while (ChildInx <= aRight-aLeft) do begin
      {if there is a right child, calculate the index of
       the larger child}
      if (succ(ChildInx) <= aRight-aLeft) and
         aLessThan(aItemArray[ChildInx+aLeft],
         aItemArray[succ(ChildInx)+aLeft]) then
        inc(ChildInx);
      {if item is >= larger child, we're done}
      if not aLessThan(Item,
        aItemArray[ChildInx+aLeft]) then
        Break;
      {otherwise move the larger child up the tree, and move
       our item down the tree and repeat}
      aItemArray[FromInx+aLeft] :=
        aItemArray[ChildInx+aLeft];
      FromInx := ChildInx;
      ChildInx := succ(FromInx * 2);
    end;
    {store our item in the correct place}
    aItemArray[FromInx+aLeft] := Item;
  end;
  {if there are only two items, above will have sorted them}
  if (ListCount = 2) then
    Exit;
  {now progressively pop off largest element and reduce heap
   size by one, storing largest element in vacated space;
   again this is complicated by the fact that all our
   indexes are offset by aLeft}
  while (ListCount > 1) do begin
    {save last item (we'll pretend it's at the root),
     replace it with the root item (ie the largest)}
    Item := aItemArray[aRight];
    aItemArray[aRight] := aItemArray[aLeft];
    {reduce the size of the heap}
    dec(ListCount);
    dec(aRight);
    {trickle down from the root}
    FromInx := 0;
    ChildInx := succ(FromInx * 2);
    {while there is at least a left child...}
    while (ChildInx <= aRight-aLeft) do begin
      {if there is a right child, calculate the index of
       the larger child}
      if (succ(ChildInx) <= aRight-aLeft) and
        aLessThan(aItemArray[ChildInx+aLeft],
        aItemArray[succ(ChildInx)+aLeft]) then
        inc(ChildInx);
      {if our item is >= the larger child, we're done}
      if not aLessThan(Item,
        aItemArray[ChildInx+aLeft]) then
        Break;
      {otherwise move the larger child up the tree, and move
       our item down the tree and repeat}
      aItemArray[FromInx+aLeft] :=
        aItemArray[ChildInx+aLeft];
      FromInx := ChildInx;
      ChildInx := succ(FromInx * 2);
    end;
    {store our item in the correct place}
    aItemArray[FromInx+aLeft] := Item;
  end;
end;
```

in the queue. Once we do have its position, we look at its parent. Is the item's new priority greater than its parent's? If so, we bubble the item up the heap. If not, we look at its children. If it's smaller than one or both of its children we trickle it down the heap. That's the easy part done! And it's efficient to boot: because it uses the bubble-up and trickle-down routines, it's an O(logN) operation. So, dragging our feet as we do so, we have to consider the first bit: how do we efficiently find the item in the heap?

Obviously we have to do something else, store some extra information somewhere, have some kind of routine or data structure to help us. What we do is create what's called an indirect heap. Instead of arranging the actual items in a heap, we arrange 'pointers' to those items instead, in other words, provide a level of indirection, and then store the items somewhere else. All well and good, but haven't we just removed the problem somewhere else? We still have to find a given item after all. We could store the items in a hash table (see my articles in *The Delphi Magazine* for February and March 1998) or a sorted list of some kind, but both these would require some kind of key to go with the item. However, using a hash table would be a valid technique under these circumstances.

Another standard way is to keep the items externally to the priority queue in an array. The queue is told where the array is (and hence can get at the items) and the methods for the queue then use the number of each item in that array, instead of the item itself. The priority queue can then maintain an array of 'pointers' into the heap, indexed by this item number. I'll admit that this algorithm doesn't sit well with me since the priority queue has absolutely no control over the external list being modified, sorted or whatever.

The algorithm we'll show uses handles to items. When you add an item to the priority queue, you're passing control of that item to the queue. In return, you'll be given a handle by which you will henceforth refer to that item. When you remove the item with the largest priority from the queue, its handle will be destroyed. You can use the handle to delete an item from the queue (wherever it may appear in the queue) and the handle will be deleted as well. You can also use the handle to replace any item in the queue (if you want to change an item's priority, change the priority inside the item and use the item's handle to replace the item, *in situ* as it were). If your items are records or objects, you can store the handle to that item inside the item itself alongside its priority and that way you'll be able to keep track of the items and their handles (of course, you would be responsible for maintaining the handle value inside the item: the queue knows nothing about the internal structure of an item, remember). And the type of a handle? It can be anything really, but traditionally it's usually a (disguised) typeless pointer to make

sure you don't do some arithmetic with it. So that's what we'll do as well.

If you look at the code on this month's disk, you'll see that internally the queue maintains a linked list of nodes. Each node contains an item and an index into the heap, and a handle used by the queue's user is actually a pointer to one of these nodes. The heap algorithm now manages a heap of handles and it has to be careful to dereference a handle whenever the item is needed for comparison, or the heap index needs to be updated.

Out Of Time

With this final extended priority queue class, we come to the end of this month's column. We've come a long way: from a simplistic priority queue that allows addition of items and their removal in priority order (with one or the other operation inefficiently coded), to a sophisticated version that performs these two operations efficiently together with two extra operations: changing or deleting any item in the priority queue. Again, these use efficient algorithms. We'll use the latter priority queue in our next foray into graphs.

Julian Bucknall is thoroughly prioritized now he's married, but still enjoys those thoroughly terrific top ten tunes from his youth.