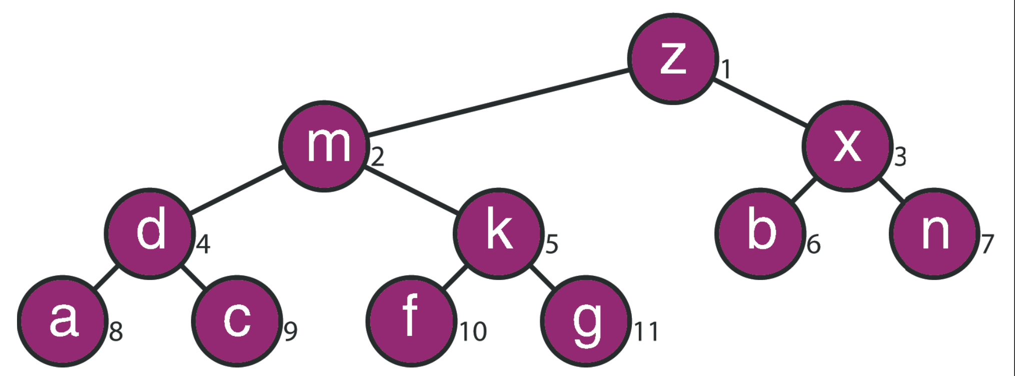
Binary heaps

Quickly find the largest item in a collection by assigning priority values within the binary heap



▲ **Figure 1: A binary heap.**

In this issue...

▶ WHAT'S COVERED

When you talk about a 'heap', people usually think of memory allocation. A 'binary heap' is not that kind of heap. It's a quite remarkable and simple data structure that has great power. It's a container of objects into which you can add objects, and from which you can easily and quickly pull out the largest object. It can complete both operations very quickly.

The binary heap is one of the best kept secrets in computer science. Oh sure, people have heard of stacks and queues, binary trees and hash tables, and some will argue about the merits of a red-black tree versus an AVL tree, but when you mention a 'heap', people automatically think of memory allocation and deallocation.

A binary heap, on the other hand, is a collection of items, each associated with a value, usually called a 'priority', from which you can quickly remove the item with the largest priority. From this rather trivial-sounding data structure, we can derive a simple – yet fast – sorting algorithm (known as 'heapsort'), maintain a list of prioritised tasks (a 'priority queue') and even manage chunks of reusable memory.

Here, the two binary heap operations that we're really interested in are adding a new item to the heap – call this the 'Insert()' method – and removing the largest item from the heap – call this 'RemoveMaximum()'. We should optimise performance for these two operations.

The question we have to answer is this: how quickly is quickly? The faster we want an operation to occur, the more compromises there will be. For this implementation, we'll state that inserting an item and removing the maximum should be O(log(n)). That is, the time taken for each of these operations should be proportional to the logarithm of the number of items in the heap, with as low a constant of proportionality as we can get.

Preparation out of the way, let's start implementing a binary heap. As a basic definition, we need to assume the following:

1 For every node in the tree, the item in the node is larger than the items in its children.
2 Every level in the tree, apart from the last, is complete.
3 All nodes in the lowest level of the tree are to the left. The last level of the tree is filled from the left to the right. This is known as 'left-complete'.

The diagram at the top left of this page shows such a binary heap.

Nugget

Since the binary heap algorithm enables you to find the largest item, and since it can be implemented using an array, we can easily modify it to sort an array. First, make the array into a heap using Floyd's algorithm and keep on extracting the largest item. Since every removal leaves an empty element at the end of the array and makes the array logically shorter, keep putting the largest item at the end of the array. ■

There are four levels: the top three are full or complete and the bottom level is left-complete.

A new node would have to occupy the left child of the 'b' node (obviously, there might be some tree rearrangements to be done in this case).

Identifying the largest item in the binary heap is easy: it's the root of the tree. Removing it, though, is problematic, since deleting the node gives us two halves of a tree, which is pretty useless. We need to patch up the tree somehow, so that the heap properties are satisfied again.

Top of the tree

Since we know how the structure of the heap should look if there were one fewer node, we just move the last node of the last level (the rightmost) to the root of the tree. The structure (that is, rules 2 and 3 as we defined earlier) is now valid again, but rule 1 is likely to be invalid since we're moving a small item from the bottom of the tree to the top.

So, let's fix rule 1 again. For this we use an algorithm called 'trickle-down'. Look at the two children of the new root.

Select the larger item, and if it's larger than that at the root, swap it with the item at the root. The root now obeys rule 1. Move down to the child node that participated in the swap and perform the same operation. Select the larger child item and swap it with the item, if it's greater. Continue moving down the tree until the node we reach no longer has any children or until the item in the node is larger than both its children.

Figure 2, on the right, shows this trickle-down process step by step, as we remove 'z' (the maximum) from Figure 1 and replace it with 'g'.

The key question to consider now is how quickly the operation we've called RemoveMaximum

Nugget

It's impossible to investigate binary heaps for any length of time without coming across the name Robert W Floyd. Floyd was a researcher in the burgeoning field of computer science in the late 1950s and 1960s, alongside Donald Knuth. He's one of the inventors of heapsort (1962, then known as treesort), and is known for devising a fast way to take an array and turn it into a heap. He was also responsible for the trickle-down tweak, among other things.

Spotlight on… Making remove operations faster

There is a simple tweak we can do to the trickle-down operation to make it quicker. This was first thought of by Robert W Floyd.

When we remove the largest item, we replace it with an item on the lowest level (the rightmost one on that level). It's likely that the item we're swapping to the root of the tree is one of the smallest in the binary heap and will return to the bottom level, or at least close to it.

Since the item is likely to be small, we'll swap it with the larger of its children every time until we get to the bottom level. In other words, instead of checking whether the larger child is also larger than the item, we'll assume that this is so. Of course, we may overshoot and bring the item lower than it should be.

No problem: we'll merely bubble-up the item from where it ends up. Because it's one of the smallest items in the tree anyway, it's likely that we may not need to do much to get the result we want out of the heap.

managed to go about its work. The initial removal of the largest item and the moving of the last item to the root were simple constant time operations that didn't depend on the number of items. (We'll see later how to identify the last node immediately.) With those out of the way, the main time spent in the operation was undoubtedly the trickle-down process. In a worst-case scenario, we'll have to trickle-down all the way to the bottom level.

How far is that? The number of levels in a complete binary tree, such as we have with a binary heap, is log(n). So the time taken by RemoveMaximum() is going to be proportional to log(n).

Note that at each level we have to do at least two comparisons and that, in general, it's the comparisons that take the time. As with all programming, the scale of the problem won't become clear until you scale up from simple experiments like these, but it's worth taking the time to optimise wherever you can, avoiding the moment where your program will only work as something to do while awaiting the heat death of the sun.

Bubbling up

Let's look at Insert() now. We know how to insert the node in order to solve the structure rules: add it as the last node on the lowest level (or start a new level, if the last level is already full). But again, rule 1 will probably need fixing up during development. This time, we use an algorithm called 'bubble-up'.

Look at the parent of the newly inserted item. If it's smaller, swap it with the new item and then move up the tree. Repeat the same process at this higher level. Continue either until you reach the root, or until the item at the parent of the current node is greater than the item we're bubbling up.

You'll find a graphical version of this on the right (Figure 3), taken step by step as we add the 'z' back into the mix.

How fast is Insert()? Again, assuming that adding the actual node doesn't depend on the number of nodes present, all the action is with the bubble-up process. If we assume the worst-case scenario again, the item may have to bubble all the way up to the root, through log(n) levels. So we can state that Insert() will also take log(n) time. (This time we only need to do one comparison per level, so we can say that Insert() will be roughly twice as fast as our original attempt, thanks to not having to go round the houses to sort out the heap.

Now that we have an optimised pair of operations to play with, let's think about the in-memory representation of this data structure. The first, most obvious, solution is to mimic the abstract tree: declare a node type with links to one parent and two children, and a reference to an item. That's a lot of overhead: three references to other nodes, making it a workable, but hardly efficient, method.

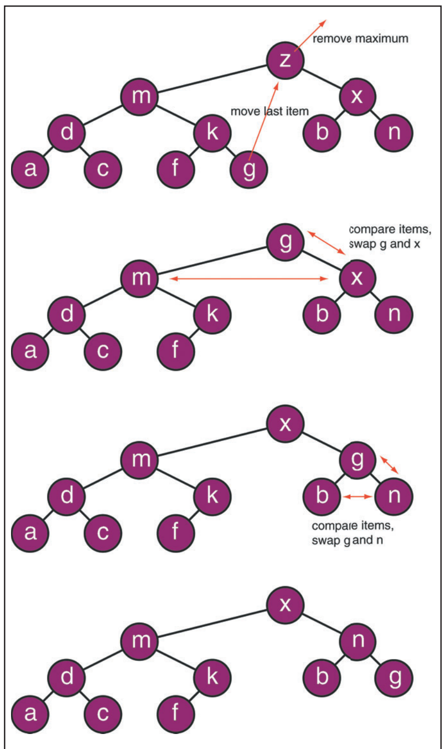
Can we do better? Of course. Consider this. Number all the nodes from the root, level by level, from left to right on each level. So the root is node one, its children are nodes two and three, their children are four and five, and then six and seven. You'll see these numbers back in Figure 1 on the opposite page.

The next level has nodes numbered from eight to 15 and so on. Notice that, because of the definition of a binary tree (rules 2 and 3, in particular), there are no gaps in this numbering sequence. In fact, we could use an array to hold the items.

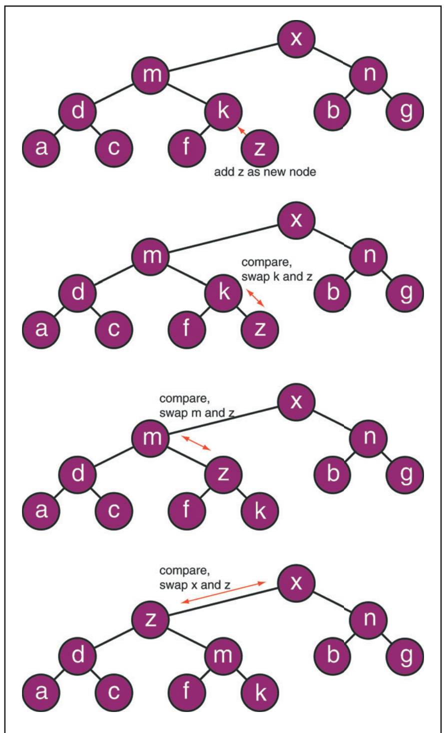
There's still an issue to solve before we can consider our job done, however. We need to find the parent and children of a node. Look closely at the numbering and you'll see a pattern. The children of node 'x' are at 2x and 2x+1, and the parent is at node x/2 (using integer division, so the remainder is discarded). ■

Julian M Bucknall has worked for many companies, ranging from TurboPower to Microsoft, and is now CTO for Developer Express.

Stumped by a particular piece of theory? Email us with subjects you'd like to see explained in detail on these pages. feedback@pcplus.co.uk



▲ **Figure 2: 'RemoveMaximum()', step by step.**



▲ **Figure 3: 'Insert()', step by step.**