Note that this `insert` algorithm does not increment the heap size `n` – that has to be done separately by whatever algorithm calls it.

Inserting a node takes at most $O(\log_2 n)$ steps, because the maximum number of times we may have to 'bubble up' the new element is the height of the tree which is $\log_2 n$.

8.5 Deleting a heap tree node

To use a binary heap tree as a priority queue, we will regularly need to delete the root, i.e. remove the node with the highest priority. We will then be left with something which is not a binary tree at all. However, we can easily make it into a complete binary tree again by taking the node at the 'last' position and using that to fill the new vacancy at the root. However, as with insertion of a new item, the heap tree (priority ordering) property might be violated. In that case, we will need to '*bubble down*' the new root by comparing it with both its children and exchanging it with the largest. This process is then repeated until the new root element has found a valid place. Thus, a suitable algorithm is:

```
deleteRoot(array heap, int n) {
   if ( n < 1 )
      error('Node does not exist')
   else {
      heap[1] = heap[n]
      bubbleDown(1,heap,n-1)
   }
}
```

A similar process can also be applied if we need to delete any other node from the heap tree, but in that case we may need to '*bubble up*' the shifted last node rather than bubble it down. Since the original heap tree is ordered, items will only ever need to be bubbled up or down, never both, so we can simply call both, because neither procedure changes anything if it is not required. Thus, an algorithm which deletes any node `i` from a heap tree is:

```
delete(int i, array heap, int n) {
   if ( n < i )
      error('Node does not exist')
   else {
      heap[i] = heap[n]
      bubbleUp(i,heap,n-1)
      bubbleDown(i,heap,n-1)
   }
}
```

The bubble down process is more difficult to implement than bubble up, because a node may have none, one or two children, and those three cases have to be handled differently. In the case of two children, it is crucial that when both children have higher priority than the given node, it is the highest priority one that is swapped up, or their priority ordering will be violated. Thus we have:

```
bubbleDown(int i, array heap, int n) {
   if ( left(i) > n )                    // no children
      return
   elseif ( right(i) > n )               // only left child
      if ( heap[i] < heap[left(i)] )
         swap heap[i] and heap[left(i)]
   else                                  // two children
      if ( heap[left(i)] > heap[right(i)] and heap[i] < heap[left(i)] ) {
          swap heap[i] and heap[left(i)]
          bubbleDown(left(i),heap,n)
       }
      elseif ( heap[i] < heap[right(i)] ) {
          swap heap[i] and heap[right(i)]
          bubbleDown(right(i),heap,n)
      }
   }
}
```

In the same way that the `insert` algorithm does not increment the heap size, this `delete` algorithm does not decrement the heap size `n` – that has to be done separately by whatever algorithm calls it. Note also that this algorithm does not attempt to be *fair* in the sense that if two or more nodes have the same priority, it is not necessarily the one that has been waiting longest that will be removed first. However, this factor could easily be fixed, if required, by keeping track of arrival times and using that in cases of equal priority.

As with insertion, deletion takes at most $O(\log_2 n)$ steps, because the maximum number of times it may have to bubble down or bubble up the replacement element is the height of the tree which is $\log_2 n$.

8.6 Building a new heap tree from scratch

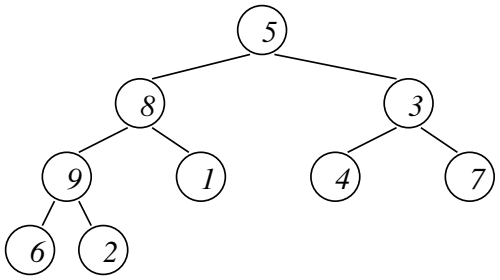
Sometimes one is given a whole set of n new items in one go, and there is a need to *build* a binary heap tree containing them. In other words, we have a set of items that we wish to *heapify*. One obvious possibility would be to insert the n items one by one into a heap tree, starting from an empty tree, using the $O(\log_2 n)$ '*bubble up*' based `insert` algorithm discussed earlier. That would clearly have overall time complexity of $O(n\log_2 n)$.

It turns out, however, that rearranging an array of items into heap tree form can be done more efficiently using '*bubble down*'. First note that, if we have the n items in an array `a` in positions $1, \ldots, n$, then all the items with an index greater than $n/2$ will be leaves, and not need bubbling down. Therefore, if we just bubble down all the non-leaf items `a[n/2],...,a[1]` by exchanging them with the larger of their children until they either are positioned at a leaf, or until their children are both smaller, we obtain a valid heap tree.

Consider a simple example array if items from which a heap tree must be built:

5	8	3	9	1	4	7	6	2

We can start by simply drawing the array as a tree, and see that the last 5 entries (those with indices greater than $9/2 = 4$) are leaves of the tree, as follows:



Then the rearrangement algorithm starts by bubbling down `a[n/2]` = `a[4]` = `9`, which turns out not to be necessary, so the array remains the same. Next `a[3]` = `3` is bubbled down, swapping with `a[7]` = `7`, giving:

5	8	7	9	1	4	3	6	2

Next `a[2]` = `8` is bubbled down, swapping with `a[4]` = `9`, giving:

5	9	7	8	1	4	3	6	2

Finally, `a[1]` = `5` is bubbled down, swapping with `a[2]` = `9`, to give first:

9	5	7	8	1	4	3	6	2

then swapping with `a[4]` = `8` to give:

9	8	7	5	1	4	3	6	2

and finally swapping with `a[8]` = `6` to give:

9	8	7	6	1	4	3	5	2

which has the array rearranged as the required heap tree.

Thus, using the above `bubbleDown` procedure, the algorithm to build a complete binary heap tree from any given array `a` of size `n` is simply:

```
heapify(array a, int n) {
   for( i = n/2 ; i > 0 ; i-- )
      bubbleDown(i,a,n)
}
```

The time complexity of this heap tree creation algorithm might be computed as follows: It potentially bubbles down $\lfloor n/2 \rfloor$ items, namely those with indices $1, \ldots, \lfloor n/2 \rfloor$. The maximum number of bubble down steps for each of those items is the height of the tree, which is $\log_2 n$, and each step involves two comparisons – one to find the highest priority child node, and one to compare the item with that child node. So the total number of comparisons involved is at most $(n/2).\log_2 n.2 = n\log_2 n$, which is the same as we would have by inserting the array items one at a time into an initially empty tree.

In fact, this is a good example of a situation in which a naive counting of loops and tree heights over-estimates the time complexity. This is because the number of bubble down steps

will usually be less than the full height of the tree. In fact, at each level as you go down the tree, there are more nodes, and fewer potential bubble down steps, so the total number of operations will actually be much less than $n\log_2 n$. To be sure of the complexity class, we need to perform a more accurate calculation. At each level i of a tree of height h there will be 2^i nodes, with at most $h-i$ bubble down steps, each with 2 comparisons, so the total number of comparisons for a tree of height h will on average be

$$C(h) = \sum_{i=0}^{h} 2^i(h-i) = 2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

The final sum converges to 2 as h increases (see Appendix A.4), so for large h we have

$$C(h) \approx 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} = 2^h.2 = 2^{h+1} \approx n$$

and the worst case will be no more than twice that. Thus, the total number of operations is $O(2^{h+1}) = O(n)$, meaning that the complexity class of `heapify` is actually $O(n)$, which is better than the $O(n\log_2 n)$ complexity of inserting the items one at a time.

8.7 Merging binary heap trees

Frequently one needs to *merge* two existing priority queues based on binary heap trees into a single priority queue. To achieve this, there are three obvious ways of merging two binary heap trees `s` and `t` of a similar size n into a single binary heap tree:

1. Move all the items from the smaller heap tree one at a time into the larger heap tree using the standard `insert` algorithm. This will involve moving $O(n)$ items, and each of them will need to be bubbled up at cost $O(\log_2 n)$, giving an overall time complexity of $O(n\log_2 n)$.

2. Repeatedly move the last items from one heap tree to the other using the standard `insert` algorithm, until the new binary tree `makeTree(0,t,s)` is complete. Then move the last item of the new tree to replace the dummy root "0", and bubble down that new root. How this is best done will depend on the sizes of the two trees, so this algorithm is not totally straightforward. On average, around half the items in the last level of one tree will need moving and bubbling, so that will be $O(n)$ moves, each with a cost of $O(\log_2 n)$, again giving an overall time complexity of $O(n\log_2 n)$. However, the actual number of operations required will, on average, be a lot less than the previous approach, by something like a factor of four, so this approach is more efficient, even though the algorithm is more complex.

3. Simply concatenate the array forms of the heap trees `s` and `t` and use the standard `heapify` algorithm to convert that array into a new binary heap tree. The `heapify` algorithm has time complexity $O(n)$, and the concatenation need be no more than that, so this approach has $O(n)$ overall time complexity, making it in the best general approach of all three.

Thus, the merging of binary heap trees generally has $O(n)$ time complexity.

If the two binary heap trees are such that very few moves are required for the second approach, then that may look like a better choice of approach than the third approach. However, `makeTree` will itself generally be an $O(n)$ procedure if the trees are array-based, rather than pointer-based, which they usually are for binary heap trees. So, for array-based similarly-sized binary heaps, the third approach is usually best.

If the heap trees to be merged have very different sizes n and $m < n$, the first approach will have overall time complexity $O(m\log_2 n)$, which could be more efficient than an $O(n)$ approach if $m \ll n$. In practice, a good general purpose merge algorithm would check the sizes of the two trees and use them to determine the best approach to apply.

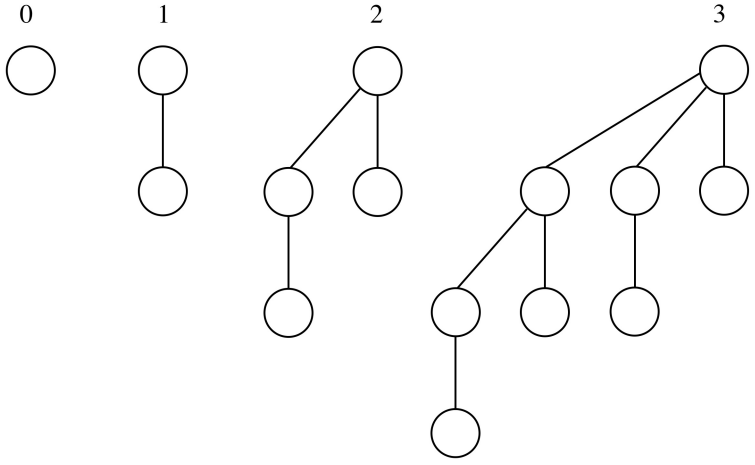
8.8 Binomial heaps

A *Binomial heap* is similar to a binary heap as described above, but has the advantage of more efficient procedures for insertion and merging. Unlike a binary heap, which consists of a single binary tree, a binomial heap is implemented as a collection of binomial trees.

Definition. A *binomial tree* is defined recursively as follows:

- A binomial tree of order 0 is a single node.
- A binomial tree of order k has a root node with children that are roots of binomial trees of orders $k-1$, $k-2$, ..., 2, 1, 0 (in that order).

Thus, a binomial tree of order k has height k, contains 2^k nodes, and is trivially constructed by attaching one order $k-1$ binomial tree as the left-most child of another order $k-1$ binomial tree. Binomial trees of order 0, 1, 2 and 3 take the form:



and it is clear from these what higher order trees will look like.

A *Binomial heap* is constructed as a collection of binomial trees with a particular structure and node ordering properties:

- There can only be zero or one binomial tree of each order.
- Each constituent binomial tree must satisfy the priority ordering property, i.e. each node must have priority less than or equal to its parent.

The structure of such a heap is easily understood by noting that a binomial tree of order k contains exactly 2^k nodes, and a binomial heap can only contain zero or one binomial tree of each order, so the total number of nodes in a Binomial Heap must be

$$n = \sum_{k=0}^{\infty} b_k 2^k \qquad b_k \in [0, 1]$$

where b_k specifies the number of trees of order k. Thus there is a one-to-one mapping between the binomial heap structure and the standard binary representation of the number n, and since the binary representation is clearly unique, so is the binomial heap structure. The maximum number of trees in a heap with n nodes therefore equals the number of digits when n is written in binary without leading zeros, i.e. $\log_2 n + 1$. The heap can be stored efficiently as a linked list of root nodes ordered by increasing tree order.

The most important operation for binomial heaps is *merge*, because that can be used as a sub-process for most other operations. Underlying that is the merge of two binomial trees of order j into a binomial tree of order $j + 1$. By definition, that is achieved by adding one of those trees as the left most sub-tree of the root of the other, and preservation of the priority ordering simply requires that it is the tree with the highest priority root that provides the root of the combined tree. This clearly has $O(1)$ time complexity. Then merging two whole binomial heaps is achieved by merging the constituent trees whenever there are two of the same order, in a sequential manner analogous to the addition of two binary numbers. In this case, the $O(1)$ insert complexity will be multiplied by the number of trees, which is $O(\log_2 n)$, so the overall time complexity of merge is $O(\log_2 n)$. This is better than the $O(n)$ complexity of merging binary heaps that can be achieved by concatenating the heap arrays and using the $O(n)$ heapify algorithm.

Insertion of a new element into an existing binomial heap can easily be done by treating the new element as a binomial heap consisting of a single node (i.e., an order zero tree), and merging that using the standard merge algorithm. The average time complexity of that *insert* is given by computing the average number of $O(1)$ tree combinations required. The probability of needing the order zero combination is 0.5, the probability of needing a second combination is 0.5^2, and the third is 0.5^3, and so on, which sum to one. So insertion has $O(1)$ overall time complexity. That is better than the $O(\log_2 n)$ complexity of insertion into a standard binary heap.

Creating a whole new binomial heap from scratch can be achieved by using the $O(1)$ insert process for each of the n items, giving an overall time complexity of $O(n)$. In this case, there is no better process, so heapify here has the same time complexity as the *heapify* algorithm for binary heaps.

Another important heap operation in practice is that of updating the heap after increasing a node priority. For standard binary heaps, that simply requires application of the usual bubble-up process with $O(\log_2 n)$ complexity. Clearly, a similar process can be used in binomial heaps, and that will also be of $O(\log_2 n)$ complexity.

The highest priority node in a binomial heap will clearly be the highest priority root node, and a pointer to that can be maintained by each heap update operation without increasing the complexity of the operation. Serving the highest priority item requires deleting the highest priority node from the order j tree it appears in, and that will break it up into another binomial heap consisting of trees of all orders from 0 to $j - 1$. However, those trees can easily be merged back into the original heap using the standard merge algorithm, with the

standard merge complexity of $O(\log_2 n)$. Deleting non-root nodes can also be achieved with the existing operations by increasing the relevant node priority to infinity, bubbling-up, and using the root delete operation, again with $O(\log_2 n)$ complexity overall. So, the complexity of *delete* is always $O(\log_2 n)$.

Exercise: Find pseudocode versions of the merge, insert and delete algorithms for binomial heaps, and see exactly how their time complexities arise.

8.9 Fibonacci heaps

A *Fibonacci heap* is another collection of trees that satisfy the standard priority-ordering property. It can be used to implement a priority queue in a similar way to binary or binomial heaps, but the structure of Fibonacci heaps are more flexible and efficient, which allows them to have better time complexities. They are named after the *Fibonacci numbers* that restrict the tree sizes and appear in their time complexity analysis.

The flexibility and efficiency of Fibonacci heaps comes at the cost of more complexity: the trees do not have a fixed shape, and in the extreme cases every element in the heap can be in a separate tree. Normally, the roots of all the trees are stored using a *circular doubly linked list*, and the children of each node are handled in the same way. A pointer to the highest priority root node is maintained, making it trivial to find the highest priority node in the heap. The efficiency is achieved by performing many operations in a *lazy* manner, with much of the work postponed for later operations to deal with.

Fibonacci heaps can easily be *merged* with $O(1)$ complexity by simply concatenating the two lists of root nodes, and then *insertion* can be done by merging the existing heap with a new heap consisting only of the new node. By inserting n items one at a time, a whole heap can be created from scratch with $O(n)$ complexity.

Obviously, at some point, order needs to be introduced into the heap to achieve the overall efficiency. This is done by keeping the number of children of all nodes to be at most $O(\log_2 n)$, and the size of a subtree rooted in a node with k children is at least F_{k+2}, where F_k is the kth Fibonacci number. The number of trees in the heap is decreased as part of the *delete* operation that is used to remove the highest priority node and update the pointer to the highest priority root. This delete algorithm is quite complex. First it removes the highest priority root, leaving its children to become roots of new trees within the heap, the processing of which will be $O(\log_2 n)$. Then the number of trees is reduced by linking together trees that have roots with the same number of children, similar to a Binomial heap, until every root has a different number of children, leaving at most $O(\log_2 n)$ trees. Finally the roots of those trees are checked to reset the pointer to the highest priority. It can be shown that all the required processes can be completed with $O(\log_2 n)$ average time complexity.

For each node, a record is kept of its number of children and whether it is marked. The mark indicates that at least one of its children has been separated since the node was made a child of another node, so all roots are unmarked. The mark is used by the algorithm for increasing a node priority, which is also complex, but can be achieved with $O(1)$ complexity. This gives Fibonacci heaps an important advantage over both binary and binomial heaps for which this operation has $O(\log_2 n)$ time complexity.

Finally, an arbitrary node can be deleted from the heap by increasing its node priority to infinity and applying the delete highest priority algorithm, resulting in an overall time complexity of $O(\log_2 n)$.

Exercise: Find pseudocode versions of the various Fibonacci heap operations, and work out how Fibonacci numbers are involved in computing their time complexities.

8.10 Comparison of heap time complexities

It is clear that the more complex Binomial and Fibonacci Heaps offer average time complexity advantages over simple Binary Heap Trees. The following table summarizes the average time complexities of the crucial heap operations:

Heap type	Insert	Delete	Merge	Heapify	Up priority
Binary	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	$O(n)$	$O(\log_2 n)$
Binomial	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(n)$	$O(\log_2 n)$
Fibonacci	$O(1)$	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(1)$

Obviously it will depend on the application in question whether using a more complicated heap is worth the effort. We shall see later that Fibonacci heaps are important in practice because they are used in the most efficient versions of many algorithms that can be implemented using priority queues, such as *Dijkstra's algorithm* for finding shortest routes, and *Prim's algorithm* for finding minimal spanning trees.

Chapter 9

Sorting

9.1 The problem of sorting

In computer science, 'sorting' usually refers to bringing a set of items into some well-defined order. To be able to do this, we first need to specify the notion of *order* on the items we are considering. For example, for numbers we can use the usual numerical order (that is, defined by the mathematical 'less than' or '$<$' relation) and for strings the so-called *lexicographic* or *alphabetic* order, which is the one dictionaries and encyclopedias use.

Usually, what is meant by *sorting* is that once the sorting process is finished, there is a simple way of 'visiting' all the items in order, for example to print out the contents of a database. This may well mean different things depending on how the data is being stored. For example, if all the objects are sorted and stored in an array a of size n, then

```
for i = 0,...,n-1
   print(a[i])
```

would print the items in ascending order. If the objects are stored in a linked list, we would expect that the first entry is the smallest, the next the second-smallest, and so on. Often, more complicated structures such as binary search trees or heap trees are used to sort the items, which can then be printed, or written into an array or linked list, as desired.

Sorting is important because having the items in order makes it much easier to *find* a given item, such as the cheapest item or the file corresponding to a particular student. It is thus closely related to the problem of *search*, as we saw with the discussion of binary search tress. If the sorting can be done beforehand (*off-line*), this enables faster *access* to the required item, which is important because that often has to be done on the fly (*on-line*). We have already seen that, by having the data items stored in a sorted array or binary search tree, we can reduce the average (and worst case) complexity of searching for a particular item to $O(\log_2 n)$ steps, whereas it would be $O(n)$ steps without sorting. So, if we often have to look up items, it is worth the effort to sort the whole collection first. Imagine using a dictionary or phone book in which the entries do not appear in some known logical order.

It follows that sorting algorithms are important tools for program designers. Different algorithms are suited to different situations, and we shall see that there is no 'best' sorting algorithm for everything, and therefore a number of them will be introduced in these notes. It is worth noting that we will be far from covering *all* existing sorting algorithms – in fact, the field is still very much alive, and new developments are taking place all the time. However,

the general strategies can now be considered to be well-understood, and most of the latest new algorithms tend to be derived by simply tweaking existing principles, although we still do not have accurate measures of performance for some sorting algorithms.

9.2 Common sorting strategies

One way of organizing the various sorting algorithms is by classifying the underlying idea, or 'strategy'. Some of the key strategies are:

enumeration sorting Consider all items. If we know that there are N items which are smaller than the one we are currently considering, then its final position will be at number $N + 1$.

exchange sorting If two items are found to be out of order, exchange them. Repeat till all items are in order.

selection sorting Find the smallest item, put it in the first position, find the smallest of the remaining items, put it in the second position ...

insertion sorting Take the items one at a time and insert them into an initially empty data structure such that the data structure continues to be sorted at each stage.

divide and conquer Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted 'parts' back together in a way that preserves the sorting.

All these strategies are based on *comparing* items and then rearranging them accordingly. These are known as comparison-based sorting algorithms. We will later consider other non-comparison-based algorithms which are possible when we have specific prior knowledge about the items that can occur, or restrictions on the range of items that can occur.

The ideas above are based on the assumption that all the items to be sorted will fit into the computer's internal memory, which is why they are often referred to as being *internal sorting algorithms*. If the whole set of items cannot be stored in the internal memory at one time, different techniques have to be used. These days, given the growing power and memory of computers, external storage is becoming much less commonly needed when sorting, so we will not consider *external sorting algorithms* in detail. Suffice to say, they generally work by splitting the set of items into subsets containing as many items as can be handled at one time, sorting each subset in turn, and then carefully merging the results.

9.3 How many comparisons must it take?

An obvious way to compute the *time complexity* of sorting algorithms is to count the number of comparisons they need to carry out, as a function of the number of items to be sorted. There is clearly no general *upper bound* on the number of comparisons used, since a particularly stupid algorithm might compare the same two items indefinitely. We are more interested in having a *lower bound* for the number of comparisons needed for the best algorithm in the worst case. In other words, we want to know the minimum number of comparisons required