These notes are currently revised each year by John Bullinaria. They include sections based on notes originally written by Martín Escardó and revised by Manfred Kerber. All are members of the School of Computer Science, University of Birmingham, UK.

- `MakeList(element, list)`, which puts an *element* at the top of an existing *list*.

Using those, our last example list can be constructed as

$$\texttt{MakeList}(3, \texttt{MakeList}(1, \texttt{MakeList}(4, \texttt{MakeList}(2, \texttt{MakeList}(5, \texttt{EmptyList}))))).$$

and it is clearly possible to *construct* any list in this way.

This *inductive* approach to data structure creation is very powerful, and we shall use it many times throughout these notes. It starts with the "*base case*", the `EmptyList`, and then builds up increasingly complex lists by repeatedly applying the "*induction step*", the `MakeList(element, list)` operator.

It is obviously also important to be able to get back the elements of a list, and we no longer have an item index to use like we have with an array. The way to proceed is to note that a list is always constructed from the first element and the rest of the list. So, conversely, from a non-empty list it must always be possible to get the first element and the rest. This can be done using the two *selectors*, also called *accessor methods*:

- `first(list)`, and
- `rest(list)`.

The selectors will only work for non-empty lists (and give an error or exception on the empty list), so we need a *condition* which tells us whether a given list is empty:

- `isEmpty(list)`

This will need to be used to check every list before passing it to a selector.

We call everything a list that can be constructed by the constructors `EmptyList` and `MakeList`, so that with the selectors `first` and `rest` and the condition `isEmpty`, the following relationships are automatically satisfied (i.e. true):

- `isEmpty(EmptyList)`
- `not isEmpty(MakeList(x, l))` (for any `x` and `l`)
- `first(MakeList(x, l))` = `x`
- `rest(MakeList(x, l))` = `l`

In addition to constructing and getting back the components of lists, one may also wish to *destructively* change lists. This would be done by so-called *mutators* which change either the first element or the rest of a non-empty list:

- `replaceFirst(x, l)`
- `replaceRest(r, l)`

For instance, with `l` = $[3, 1, 4, 2, 5]$, applying `replaceFirst(9, l)` changes `l` to $[9, 1, 4, 2, 5]$. and then applying `replaceRest([6, 2, 3, 4], l)` changes it to $[9, 6, 2, 3, 4]$.

We shall see that the concepts of *constructors*, *selectors* and *conditions* are common to virtually all abstract data types. Throughout these notes, we will be formulating our data representations and algorithms in terms of appropriate definitions of them.

lists as arrays. However, that can be problematic because lists are conceptually not limited in size, which means array based implementation with fixed-sized arrays can only approximate the general concept. For many applications, this is not a problem because a maximal number of list members can be determined a priori (e.g., the maximum number of students taking one particular module is limited by the total number of students in the University). More general purpose implementations follow a pointer based approach, which is close to the diagrammatic representation given above. We will not go into the details of all the possible implementations of lists here, but such information is readily available in the standard textbooks.

3.2 Recursion

We previously saw how iteration based on for-loops was a natural way to process collections of items stored in arrays. When items are stored as linked-lists, there is no index for each item, and *recursion* provides the natural way to process them. The idea is to formulate procedures which involve at least one step that invokes (or calls) the procedure itself. We will now look at how to implement two important *derived procedures* on lists, `last` and `append`, which illustrate how recursion works.

To find the last element of a list `l` we can simply keep removing the first remaining item till there are no more left. This *algorithm* can be written in *pseudocode* as:

```
last(l) {
   if ( isEmpty(l) )
      error('Error: empty list in last')
   elseif ( isEmpty(rest(l)) )
      return first(l)
   else
      return last(rest(l))
}
```


The running time of this depends on the length of the list, and is proportional to that length, since `last` is called as often as there are elements in the list. We say that the procedure has *linear time complexity*, that is, if the length of the list is increased by some factor, the execution time is increased by the same factor. Compared to the *constant time complexity* which access to the last element of an array has, this is quite bad. It does not mean, however, that lists are inferior to arrays in general, it just means that lists are not the ideal data structure when a program has to access the last element of a long list very often.

Another useful procedure allows us to *append* one list `l2` to another list `l1`. Again, this needs to be done one item at a time, and that can be accomplished by repeatedly taking the first remaining item of `l1` and adding it to the front of the remainder appended to `l2`:

```
append(l1,l2) {
   if ( isEmpty(l1) )
      return l2
   else
      return MakeList(first(l1),append(rest(l1),l2))
}
```

The time complexity of this procedure is proportional to the length of the first list, `l1`, since we have to call `append` as often as there are elements in `l1`.

This algorithm works by repeatedly splitting the array into two segments, one going from $left$ to mid, and the other going from $mid + 1$ to $right$, where mid is the position half way from $left$ to $right$, and where, initially, $left$ and $right$ are the leftmost and rightmost positions of the array. Because the array is sorted, it is easy to see which of each pair of segments the searched-for item x is in, and the search can then be restricted to that segment. Moreover, because the size of the sub-array going from locations $left$ to $right$ is halved at each iteration of the while-loop, we only need $\log_2 n$ steps in either the average or worst case. To see that this runtime behaviour is a big improvement, in practice, over the earlier linear-search algorithm, notice that $\log_2 1000000$ is approximately 20, so that for an array of size 1000000 only 20 iterations are needed in the worst case of the binary-search algorithm, whereas 1000000 are needed in the worst case of the linear-search algorithm.

With the binary search algorithm, it is not so obvious that we have taken proper care of the boundary condition in the while loop. Also, strictly speaking, this algorithm is not correct because it does not work for the empty array (that has size zero), but that can easily be fixed. Apart from that, is it correct? Try to convince yourself that it is, and then try to explain your argument-for-correctness to a colleague. Having done that, try to *write down* some convincing arguments, maybe one that involves a *loop invariant* and one that doesn't. Most algorithm developers stop at the first stage, but experience shows that it is only when we attempt to write down seemingly convincing arguments that we actually find all the subtle mistakes. Moreover, it is not unusual to end up with a better/clearer algorithm after it has been modified to make its correctness easier to argue.

It is worth considering whether linked-list versions of our two algorithms would work, or offer any advantages. It is fairly clear that we could perform a linear search through a linked list in essentially the same way as with an array, with the relevant pointer returned rather than an index. Converting the binary search to linked list form is problematic, because there is no efficient way to split a linked list into two segments. It seems that our array-based approach is the best we can do with the data structures we have studied so far. However, we shall see later how more complex data structures (trees) can be used to formulate efficient recursive search algorithms.

Notice that we have not yet taken into account how much effort will be required to sort the array so that the binary search algorithm can work on it. Until we know that, we cannot be sure that using the binary search algorithm really is more efficient overall than using the linear search algorithm on the original unsorted array. That may also depend on further details, such as how many times we need to performa a search on the set of n items – just once, or as many as n times. We shall return to these issues later. First we need to consider in more detail how to compare algorithm efficiency in a reliable manner.

so we can do something `n` times, or look for the `n`th item, etc. The classic example is the recursive factorial function:

```
factorial(int n) {
   if ( n == 0 ) return 1
   return n*factorial(n-1)
}
```

Another example, with two termination or base-case conditions, is a direct implementation of the recursive definition of *Fibonacci numbers* (see Appendix A.5):

```
F(int n) {
   if ( n == 0 ) return 0
   if ( n == 1 ) return 1
   return F(n-1) + F(n-2)
}
```

though this is an extremely inefficient algorithm for computing these numbers. Exercise: Show that the time complexity of this algorithm is $O(2^n)$, and that there exists a straightforward iterative algorithm that has only $O(n)$ time complexity. Is it possible to create an $O(n)$ recursive algorithm to compute these numbers?

In most cases, however, we won't need to worry about counting, because the relevant data structure has a natural end point condition, such as `isEmpty(x)`, that will bring the recursion to an end.

Chapter 7

Binary Search Trees

We now look at Binary Search Trees, which are a particular type of binary tree that provide an efficient way of *storing* data that allows particular items to be found as quickly as possible. Then we consider further elaborations of these trees, namely AVL trees and B-trees, which operate more efficiently at the expense of requiring more sophisticated algorithms.

7.1 Searching with arrays or lists

As we have already seen in Chapter 4, many computer science applications involve *searching* for a particular item in a collection of data. If the data is stored as an unsorted array or list, then to find the item in question one obviously has to check each entry in turn until the correct one is found, or the collection is exhausted. On average, if there are n items, this will take $n/2$ checks and in the worst case, all n items will have to be checked. If the collection is large, such as all items accessible via the internet, that will take too much time. We also saw that if the items are sorted before storing in an array, one can perform binary search which only requires $\log_2 n$ checks in the average and worst cases. However, that involves an overhead of sorting the array in the first place, or maintaining a sorted array if items are inserted or deleted over time. The idea here is that, with the help of binary trees, we can speed up the storing and search process without needing to maintain a sorted array.

7.2 Search keys

If the items to be searched are labelled by comparable *keys*, one can order them and store them in such a way that they are *sorted* already. Being 'sorted' may mean different things for different keys, and which key to choose is an important design decision.

In our examples, the search keys will, for simplicity, usually be integer numbers (such as student ID numbers), but other choices occur in practice. For example, the comparable keys could be words. In that case, comparability usually refers to the *alphabetical* order. If `w` and `t` are words, we write `w` $<$ `t` to mean that `w` precedes `t` in the alphabetical order. If `w` = *bed* and `t` = *sky* then the relation `w` $<$ `t` holds, but this is not the case if `w` = *bed* and `t` = *abacus*. A classic example of a collection to be searched is a dictionary. Each entry of the dictionary is a pair consisting of a word and a definition. The definition is a sequence of words and punctuation symbols. The search key, in this example, is the word (to which a definition is attached in the dictionary entry). Thus, *abstractly*, a dictionary is a sequence of

```
allsmaller(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) < v) and allsmaller(left(t),v)
                                          and allsmaller(right(t),v) )
}

allbigger(tree t, value v) {
   if ( isEmpty(t) )
      return true
   else
      return ( (root(t) > v) and allbigger(left(t),v)
                                          and allbigger(right(t),v) )
}
```

However, the simplest or most obvious algorithm is not always the most efficient. Exercise: identify what is inefficient about this algorithm, and formulate a more efficient algorithm.

7.9 Sorting using binary search trees

Sorting is the process of putting a collection of items in order. We shall formulate and discuss many sorting algorithms later, but we are already able to present one of them.

The node values stored in a binary search tree can be printed in ascending order by recursively printing the left sub-tree, root and right sub-tree in the right order as follows:


```
printInOrder(tree t) {
   if ( not isEmpty(t) ) {
      printInOrder(left(t))
      print(root(t))
      printInOrder(right(t))
   }
}
```

Then, if the collection of items to be sorted is given as an array `a` of known size `n`, they can be printed in sorted order by the algorithm:

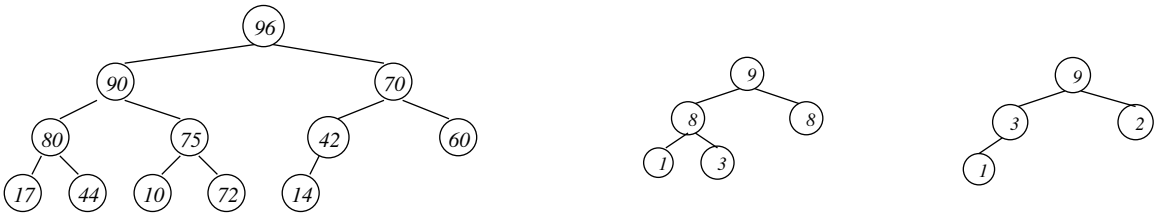
```
sort(array a of size n) {
   t = EmptyTree
   for i = 0,1,...,n-1
      t = insert(a[i],t)
   printInOrder(t)
}
```

which starts with an empty tree, inserts all the items into it using `insert`(`v`, `t`) to give a binary search tree, and then prints them in order using `printInOrder`(`t`). Exercise: modify this algorithm so that instead of printing the sorted values, they are put back into the original array in ascending order.

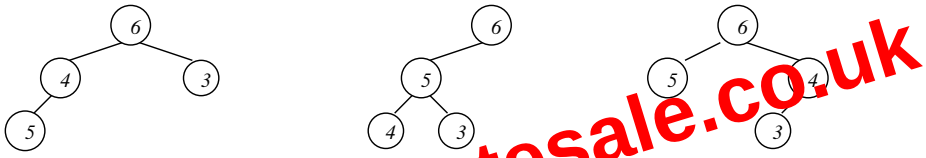
Alternatively, one could define a heap tree as a complete binary tree such that the priority of every node is higher than (or equal to) that of all its descendants. Or, as a complete binary tree for which the priorities become smaller along every path down through the tree.

The most obvious difference between a binary heap tree and a binary search trees is that the biggest number now occurs at the root rather than at the right-most node. Secondly, whereas with binary search trees, the left and right sub-trees connected to a given parent node play very different rôles, they are interchangeable in binary heap trees.

Three examples of binary trees that are valid heap trees are:



and three which are not valid heap trees are:



the first because $5 > 4$ violates the required priority ordering, the second because it is not perfectly balanced and hence not complete, and the third because it is not complete due to the node on the last level not being as far to the left as possible.

8.3 Basic operations on binary heap trees

In order to develop algorithms using an array representation, we need to allocate memory and keep track of the largest position that has been filled so far, which is the same as the current number of nodes in the heap tree. This will involve something like:

```
int MAX = 100     // Maximum number of nodes allowed
int heap[MAX+1]   // Stores priority values of nodes of heap tree
int n = 0         // Largest position that has been filled so far
```

For heap trees to be a useful representation of priority queues, we must be able to *insert* new nodes (or customers) with a given priority, *delete* unwanted nodes, and identify and remove the top-priority node, i.e. the *root* (that is, 'serve' the highest priority customer). We also need to be able to determine when the queue/tree is empty. Thus, assuming the priorities are given by integers, we need a constructor, mutators/selectors, and a condition:

```
insert(int p, array heap, int n)
delete(int i, array heap, int n)
int root(array heap, int n)
boolean heapEmpty(array heap, int n)
```

Identifying whether the heap tree is empty, and getting the root and last leaf, is easy:

array into a binary heap tree, and then the for loop moves each successive root one item at a time into the correct position in the sorted array:

```
heapSort(array a, int n) {
   heapify(a,n)
   for( j = n ; j > 1 ; j-- ) {
      swap a[1] and a[j]
      bubbleDown(1,a,j-1)
   }
}
```

It is clear from the swap step that the order of identical items can easily be reversed, so there is no way to render the Heapsort algorithm *stable*.

The average and worst-case time complexities of the entire Heapsort algorithm are given by the *sum* of two complexity functions, first that of `heapify` rearranging the original unsorted array into a heap tree which is $O(n)$, and then that of making the sorted array out of the heap tree which is $O(n\log_2 n)$ coming from the $O(n)$ bubble-downs each of which has $O(\log_2 n)$ complexity. Thus the overall average and worst-case complexities are both $O(n\log_2 n)$, and we now have a sorting algorithm that achieves the theoretical best worst-case time complexity. Using more sophisticated priority queues, such as Binomial or Fibonacci heaps, cannot improve on this because they have the same delete time complexity.

A useful feature of Heapsort is that if only the largest $m \ll n$ items need to be found and sorted, rather than all n, the complexity of the second stage is only $O(m\log_2 n)$, which can easily be less than $O(n)$ and thus reduce the whole algorithm to only $O(n)$.

9.11 Divide and conquer algorithms

All the sorting algorithms considered so far work on the whole set of items together. Instead, *divide and conquer* algorithms recursively split the sorting problem into more manageable sub-problems. The idea is that it will usually be easier to sort many smaller collections of items than one big one, and sorting single items is trivial. So we repeatedly split the given collection into two smaller parts until we reach the 'base case' of one-item collections, which require no effort to sort, and then merge them back together again. There are two main approaches for doing this:

Assuming we are working on an array `a` of size n with entries `a[0],...,a[n-1]`, then the obvious approach is to simply split the set of indices. That is, we split the array at item $n/2$ and consider the two sub-arrays `a[0],...,a[(n-1)/2]` and `a[(n+1)/2],...,a[n-1]`. This method has the advantage that the splitting of the collection into two collections of equal (or nearly equal) size at each stage is easy. However, the two sorted arrays that result from each split have to be *merged* together carefully to maintain the ordering. This is the underlying idea for a sorting algorithm called *mergesort*.

Another approach would be to split the array in such a way that, at each stage, all the items in the first collection are no bigger than all the items in the second collection. The splitting here is obviously more complex, but all we have to do to put the pieces back together again at each stage is to take the first sorted array followed by the second sorted array. This is the underlying idea for a sorting algorithm called *Quicksort*.

We shall now look in detail at how these two approaches work in practice.

9.14 Summary of comparison-based sorting algorithms

The following table summarizes the key properties of all the comparison-based sorting algorithms we have considered:

Sorting Algorithm	Strategy employed	Objects manipulated	Worst case complexity	Average case complexity	Stable
Bubble Sort	Exchange	arrays	$O(n^2)$	$O(n^2)$	Yes
Selection Sort	Selection	arrays	$O(n^2)$	$O(n^2)$	No
Insertion Sort	Insertion	arrays/lists	$O(n^2)$	$O(n^2)$	Yes
Treesort	Insertion	trees/lists	$O(n^2)$	$O(n\log_2 n)$	Yes
Heapsort	Selection	arrays	$O(n\log_2 n)$	$O(n\log_2 n)$	No
Quicksort	D & C	arrays	$O(n^2)$	$O(n\log_2 n)$	Maybe
Mergesort	D & C	arrays/lists	$O(n\log_2 n)$	$O(n\log_2 n)$	Yes

To see what the time complexities mean in practice, the following table compares the typical run times of those of the above algorithms that operate directly on arrays:

Algorithm	128	256	512	1024	O1024	R1024	2048
Bubble Sort	54	221	881	3621	1285	5627	14497
Selection Sort	12	45	164	662	643	833	2497
Insertion Sort	15	69	286	1437	6	2200	4536
Heapsort	21	45	103	236	225	249	527
Quicksort	2	27	55	113	1131	1200	230
Quicksort2	6	12	24	57	1115	1191	134
Mergesort	18	36	88	188	166	170	409
Mergesort2	6	22	48	112	94	93	254

As before, arrays of the stated sizes are filled randomly, except O1024 that denotes an array with 1024 entries which are already sorted, and R1024 that denotes an array which is sorted in the *reverse* order. Quicksort2 and Mergesort2 are algorithms where the recursive procedure is abandoned in favour of Selection Sort once the size of the array falls to 16 or below. It should be emphasized again that these numbers are of limited accuracy, since they vary somewhat depending on machine and language implementation.

What has to be stressed here is that there is no 'best sorting algorithm' in general, but that there are usually good and bad choices of sorting algorithms *for particular circumstances*. It is up to the program designer to make sure that an appropriate one is picked, depending on the properties of the data to be sorted, how it is best stored, whether all the sorted items are required rather than some sub-set, and so on.

9.15 Non-comparison-based sorts

All the above sorting algorithms have been based on comparisons of the items to be sorted, and we have seen that we can't get time complexity better than $O(n\log_2 n)$ with comparison based algorithms. However, in some circumstances it is possible to do better than that with sorting algorithms that are not based on comparisons.

That is, we think of the strings as coding numbers in base 26.

Now it is quite easy to go from any *number* k (rather than a string) to a number from 0 to 10. For example, we can take the remainder the number leaves when divided by 11. This is the *C* or *Java* modulus operation `k % 11`. So our hash function is

$$h(X_1X_2X_3) = (\texttt{k}_1 * 26^2 + \texttt{k}_2 * 26 + \texttt{k}_3)\%11 = \texttt{k}\%11.$$

This *modulo* operation, and *modular arithmetic* more generally, are widely used when constructing good hash functions.

As a simple example of a hash table in operation, assume that we now wish to insert the following three-letter airport acronyms as keys (in this order) into our hash table: PHL, ORY, GCM, HKG, GLA, AKL, FRA, LAX, DCA. To make this easier, it is a good idea to start by listing the values the hash function takes for each of the keys:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA
$h(X_1X_2X_3)$	4	8	6	4	8	7	5	1	1

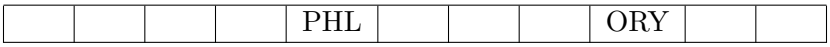
It is clear already that we will have hash collisions to deal with.

We naturally start off with an empty table of the required size, i.e. 11:

Clearly we have to be able to tell whether a particular location in the array is still empty, or whether it has already been filled. We can assume that there is a *unique* key or entry (which is *never* associated with a record) which denotes that the position has not been filled yet. However, for clarity, this key will not appear in the pictures we use.

Now we can begin inserting the keys in order. The number associated with the first item PHL is 4, so we place it at index 4, giving:

Next is ORY, which gives us the number 8, so we get:



Then we have GCM, with value 6, giving:

Then HKG, which also has value 4, results in our first collision since the corresponding position has already been filled with PHL. Now we could, of course, try to deal with this by simply saying the table is full, but this gives such poor performance (due to the frequency with which collisions occur) that it is unacceptable.

10.7 Strategies for dealing with collisions

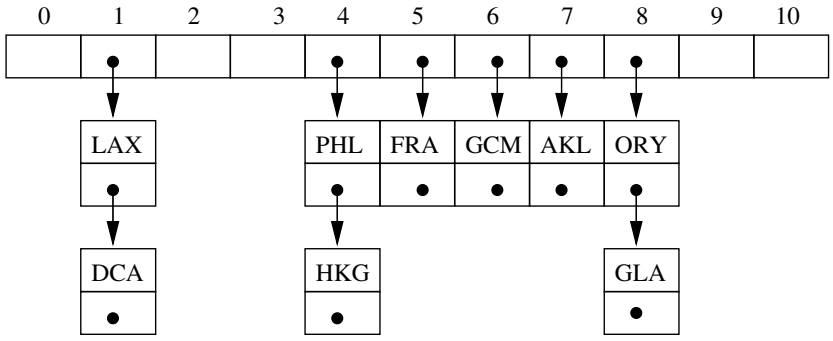
We now look at three standard approaches, of increasing complexity, for dealing with hash collisions:

Buckets. One obvious option is to reserve a two-dimensional array from the start. We can think of each column as a bucket in which we throw all the elements which give a particular result when the hash function is supplied, so the fifth column contains all the keys for which the hash function evaluates to 4. Then we could put HKG into the slot 'beneath' PHL, and GLA in the one beneath ORY, and continue filling the table in the order given until we reach:

0	1	2	3	4	5	6	7	8	9	10
	LAX			PHL	FRA	GCM	AKL	ORY		
	DCA			HKG				GLA		

The disadvantage of this approach is that it has to reserve quite a bit more space than will be eventually required, since it must take into account the likely maximal number of collisions. Even while the table is still quite empty overall, collisions will become increasingly likely. Moreover, when searching for a particular key, it will be necessary to search the entire column associated with its expected position, at least until an empty slot is reached. If there is an order on the keys, they can be stored in ascending order, which means we can use the more efficient binary search rather than linear search, but the ordering will have an overhead of its own. The average complexity of searching for a particular item depends on how many entries in the array have been filled already. This approach turns out to be slower than the other techniques we shall consider, so we shall not spend any more time on it, apart from noting that it does prove useful when the entries are held in slow external storage.

Direct chaining. Rather than reserving entire sub-arrays (the columns above) for keys that collide, one can instead create a linked list for the set of entries corresponding to each key. The result for the above example can be pictured something like this:



This approach does not reserve any space that will not be taken up, but has the disadvantage that in order to find a particular item, lists will have to be traversed. However, adding the hashing step still speeds up retrieval considerably.

We can compute the size of the average non-empty list occurring in the hash table as follows. With n items in an array of size m, the probability than no items land in a particular slot is $q(n,m) = (\frac{m-1}{m})^n$. So the number of slots with at least one item falling in it is

$$N(n,m) = m.\Big(1 - q(n,m)\Big) = m.\Big(1 - (\frac{m-1}{m})^n\Big)$$

and when searching for AKL we would know to continue beyond the exclamation mark. If, on the other hand, we are trying to *insert* a key, then we can ignore any exclamation marks and fill the position once again. This now does take care of all our problems, although if we do a lot of deleting and inserting, we will end up with a table which is a bit of a mess. A large number of exclamation marks means that we have to keep looking for a long time to find a particular entry despite the fact that the load factor may not be all that high. This happens if deletion is a frequent operation. In such cases, it may be better to re-fill a new hash table again from scratch, or use another implementation.

Search complexity. The complexity of open addressing with linear probing is rather difficult to compute, so we will not attempt to present a full account of it here. If λ is once again the load factor of the table, then a successful search can be shown to take $\frac{1}{2}(1 + \frac{1}{1-\lambda})$ comparisons on average, while an unsuccessful search takes approximately $\frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$. For relatively small load factors, this is quite impressive, and even for larger ones, it is not bad. Thus, the hash table time complexity for search is again constant, i.e. $O(1)$.

Clustering. There is a particular problem with linear probing, namely what is known as *primary* and *secondary clustering*. Consider what happens if we try to insert two keys that have the same result when the hash function is applied to them. Take the above example with hash table at the stage where we just inserted GLA:

			HKG	PHL		CDG	GLA	ORY		

If we next try to insert JFK we note that the hash function evaluates to 8 once again. So we keep checking the same locations we only just checked in order to insert GLA. This seems a rather inefficient way of doing this. This effect is known as *primary clustering* because the new key JFK will be inserted close to the previous key with the same primary position, GLA. It means that we get a continuous 'block' of filled slots, and whenever we try to insert any key which is sent into the block by the hash function, we will have to test all locations until we hit the end of the block, and then make such block even bigger by appending another entry at its end. So these blocks, or clusters, keep growing, not only if we hit the same primary location repeatedly, but also if we hit anything that is part of the same cluster. The last effect is called *secondary clustering*. Note that searching for keys is also adversely affected by these clustering effects.

10.9 Double Hashing

The obvious way to avoid the clustering problems of *linear probing* is to do something slightly more sophisticated than trying every position to the left until we find an empty one. This is known as *double hashing*. We apply a *secondary hash function* to tell us how many slots to jump to look for an empty slot if a key's *primary position* has been filled already.

Like the primary hash function, there are many possible choices of the secondary hash function. In the above example, one thing we could do is take the same number k associated with the three-character code, and use the result of integer division by 11, instead of the remainder, as the secondary hash function. However, the resulting value might be bigger than 10, so to prevent the jump looping round back to, or beyond, the starting point, we *first* take

An undirected graph is *connected* if every pair of vertices has a path connecting them. For directed graphs, the notion of connectedness has two distinct versions: We say that a digraph is *weakly connected* if for every two vertices A and B there is either a path from A to B or a path from B to A. We say it is *strongly connected* if there are paths leading both ways. So, in a weakly connected digraph, there may be two vertices i and j such that there exists no path from i to j.

A graph clearly has many properties similar to a *tree*. In fact, any tree can be viewed as a simple graph of a particular kind, namely one that is connected and contains no circles. Because a graph, unlike a tree, does not come with a natural 'starting point' from which there is a unique path to each vertex, it does not make sense to speak of parents and children in a graph. Instead, if two vertices A and B are connected by an edge e, we say that they are *neighbours*, and the edge connecting them is said to be *incident* to A and B. Two edges that have a vertex in common (for example, one connecting A and B and one connecting B and C) are said to be *adjacent*.

11.2 Implementing graphs

All the data structures we have considered so far were designed to hold certain information, and we wanted to perform certain actions on them which mostly centred around inserting new items, deleting particular items, searching for particular items, and sorting the collection. At no time was there ever a *connection* between all the items represented, apart from the order in which their keys appeared. Moreover, that connection was never something that was inherent in the structure and that we therefore tried to represent somehow – it was just a property that we used to store the items in a way which made sorting and searching quicker. Now, on the other hand, it is the connections that are the crucial information we need to encode in the data structure. We are *given* a structure which comes with specified connections, and we need to design an implementation that efficiently keeps track of them.

Array-based implementation. The first underlying idea for *array*-based implementations is that we can conveniently rename the vertices of the graph so that they are labelled by non-negative integer indices, say from 0 to $n-1$, if they do not have these labels already. However, this only works if the graph is given *explicitly*, that is, if we know in advance how many vertices there will be, and which pairs will have edges between them. Then we only need to keep track of which vertex has an edge to which other vertex, and, for weighted graphs, what the weights on the edges are. For unweighted graphs, we can do this quite easily in an $n \times n$ two-dimensional binary array `adj`, also called a *matrix*, the so-called *adjacency matrix*. In the case of weighted graphs, we instead have an $n \times n$ *weight matrix* `weights`. The array/matrix representations for the two example graphs shown above are then:

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	0	0	1	0	0
C	2	1	0	0	0	1
D	3	0	0	1	0	1
E	4	0	0	0	0	0

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	∞	4	∞
B	1	2	0	2	2	6
C	2	∞	3	0	2	1
D	3	∞	∞	∞	0	1
E	4	∞	∞	3	2	0

the algorithm finishes. However, *before the algorithm finishes*, $D[z]$ is the best *overestimate* we currently have of the distance from s to z. We initially have $D[s] = 0$, and set $D[z] = \infty$ for all vertices z other than the start node s. Then the algorithm repeatedly decreases the overestimates until it is no longer possible to decrease them further. When this happens, the algorithm terminates, with each estimate fully constrained and said to be *tight*.

Improving estimates. The general idea is to look systematically for *shortcuts*. Suppose that, for two given vertices u and z, it happens that $D[u] + \text{weight}[u][z] < D[z]$. Then there is a way of going from s to u and then to z whose total length is smaller than the current overestimate $D[z]$ of the distance from s to z, and hence we can replace $D[z]$ by this better estimate. This corresponds to the code fragment

```
if ( D[u] + weight[u][z] < D[z] )
   D[z] = D[u] + weight[u][z]
```

of the full algorithm given below. The problem is thus reduced to developing an algorithm that will systematically apply this improvement so that (1) we eventually get the tight estimates promised above, and (2) that is done as efficiently as possible.

Dijkstra's algorithm, Version 1. The first version of such an algorithm is not as efficient as it could be, but it is relatively simple and certainly correct. (It is always a good idea to start with an inefficient simple algorithm, so that the results from it can be used to check the operation of a more complex efficient algorithm.) The general idea is that, at each stage of the algorithm's operation, if an entry $D[u]$ of the array D has the minimal value among all the values recorded in D, then the overestimate $D[u]$ must actually be tight, because the improvement algorithm discussed above cannot possibly find a shortcut.

The following algorithm implements that idea:

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.
// Output: An array 'D' of distances as explained above.

// We begin by buiding the distance overestimates.

D[s] = 0   // The shortest path from s to itself has length zero.

for ( each vertex z of the graph ) {
   if ( z is not the start vertex s )
      D[z] = infinity    // This is certainly an overestimate.
}

// We use an auxiliary array 'tight' indexed by the vertices,
// that records for which nodes the shortest path estimates
// are ''known'' to be tight by the algorithm.

for ( each vertex z of the graph ) {
   tight[z] = false
}
```

```
Vertex D has minimal estimate, and so is tight.

Neighbour E has estimate unchanged.

        |A      B       C       D       E
--------+-----------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     no
pred.   |none   A       B       B       C

Vertex E has minimal estimate, and so is tight.

Neighbour C is already tight.
Neighbour D is already tight.

        |A      B       C       D       E
--------+-----------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     yes
pred.   |none   A       B       B       C

End of Dijkstra's computation.

A shortest path from A to E is:  A B C E.
```


Once it is clear what is happening at each stage, it is usually more convenient to adopt a shorthand notation that allows the whole process to be represented in a single table. For example, using a "*" to represent tight, the distance, status and predecessor for each node at each stage of the above example can be listed more concisely as follows:

```
Stage  |   A        B         C         D         E
-------+-----------------------------------------------
1      |  0        oo        oo        oo        oo
2      |  0 *      1   A     oo        4   A     oo
3      |  0 *      1 * A     3   B     3   B     7   B
4      |  0 *      1 * A     3 * B     3   B     4   C
5      |  0 *      1 * A     3 * B     3 * B     4   C
6      |  0 *      1 * A     3 * B     3 * B     4 * C

A shortest path from A to E is:  A B C E.
```

Dijkstra's algorithm, Version 2. The time complexity of Dijkstra's algorithm can be improved by making use of a *priority queue* (e.g., some form of heap) to keep track of which node's distance estimate becomes tight next. Here it is convenient to use the convention that lower numbers have higher priority. The previous algorithm then becomes:

sparse with $e = O(n)$. That is, there are usually not many more edges than vertices, and in this case the time complexity for both priority queue versions is $O(n \log_2 n)$, which is a clear improvement over the previous $O(n^2)$ algorithm.

11.7 Shortest paths – Floyd's algorithm

If we are not only interested in finding the shortest path from one specific vertex to all the others, but the shortest paths between every pair of vertices, we could, of course, apply Dijkstra's algorithm to every starting vertex. But there is actually a simpler way of doing this, known as *Floyd's algorithm*. This maintains a square matrix 'distance' which contains the overestimates of the shortest paths between every pair of vertices, and systematically decreases the overestimates using the same *shortcut* idea as above. If we also wish to keep track of the routes of the shortest paths, rather than just their lengths, we simply introduce a second square matrix 'predecessor' to keep track of all the 'previous vertices'.

In the algorithm below, we attempt to decrease the estimate of the distance from each vertex s to each vertex z by going systematically via each possible vertex u to see whether that is a shortcut; and if it is, the overestimate of the distance is decreased to the smaller overestimate, and the predecessor updated:

```
// Store initial estimates and predecessors.

for ( each vertex s ) {
   for ( each vertex z ) {
      distance[s][z] = weight[s][z]
      predecessor[s][z] = s
   }
}

// Improve them by considering all possible shortcuts u.

for ( each vertex u ) {
   for ( each vertex s ) {
      for ( each vertex z ) {
         if ( distance[s][u]+distance[u][z] < distance[s][z] ) {
            distance[s][z] = distance[s][u]+distance[u][z]
            predecessor[s][z] = predecessor[u][z]
         }
      }
   }
}
```


As with Dijkstra's algorithm, this can easily be adapted to the case of non-weighted graphs by assigning a suitable weight matrix of 0s and 1s.

The time complexity here is clearly $O(n^3)$, since it involves three nested for loops of $O(n)$. This is the same complexity as running the $O(n^2)$ Dijkstra's algorithm once for each of the n possible starting vertices. In general, however, Floyd's algorithm will be faster than Dijkstra's, even though they are both in the same complexity class, because the former performs fewer

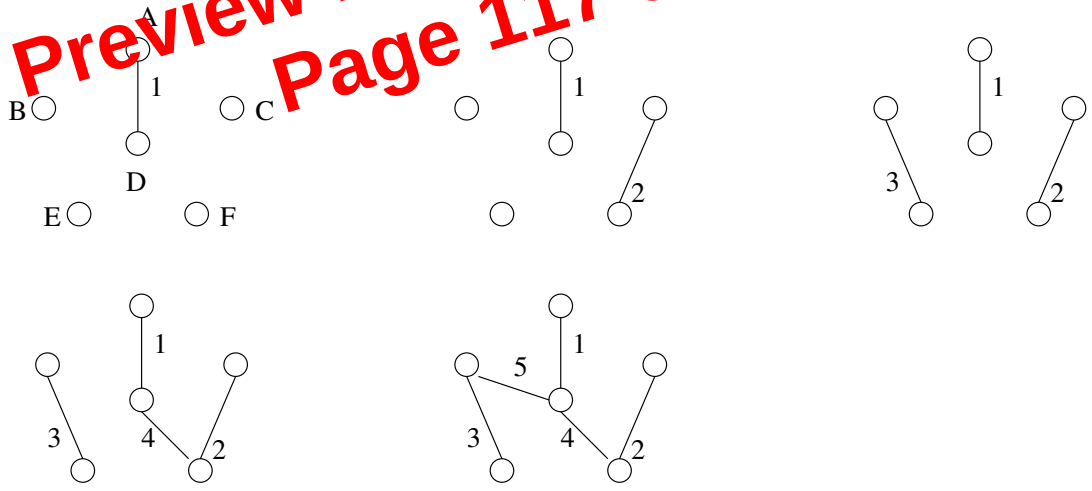
be a minimal spanning tree. Now we can repeat this process until we have replaced all the edges in X_1 that are not in Y, and we end up with the minimal spanning tree $X_n = Y$, which completes the proof that Y is a minimal spanning tree.

The time complexity of the standard Prim's algorithm is $O(n^2)$ because at each step we need to choose a vertex to add to S, and then update the `closest` array, not dissimilar to the simplest form of Dijkstra's algorithm. However, as with Dijkstra's algorithm, a *Binary* or *Binomial heap* based priority queue can be used to speed things up by keeping track of which is the minimal weight vertex to be added next. With an adjacency list representation, this can bring the complexity down to $O((e+n)\log_2 n)$. Finally, using the more sophisticated *Fibonacci heap* for the priority queue can improve this further to $O(e+n\log_2 n)$. Thus, using the optimal approach in each case, Prim's algorithm is $O(n\log_2 n)$ for sparse graphs that have $e = O(n)$, and $O(n^2)$ for highly connected graphs that have $e = O(n^2)$.

Just as with Floyd's versus Dijkstra's algorithm, we should consider whether it eally is necessary to process every vertex at each stage, because it could be sufficient to only check actually existing edges. We therefore now consider an alternative edge-based strategy:

Kruskal's algorithm – A greedy edge-based approach. This algorithm does not consider the vertices directly at all, but builds a minimal spanning tree by considering and adding edges as follows: Assume that we already have a collection of edges T. Then from all the edges not yet in T, choose one with minimal weight such that its addition to T does not produce a circle, and add that to T. If we start with T being the empty set, and continue until no more edges can be added, a minimal spanning tree will be produced. This approach is known as *Kruskal's algorithm*.

For the same graph as used for Prim's algorithm, this algorithm proceeds as follows:



In practice, Kruskal's algorithm is implemented in a rather different way to Prim's algorithm. The general idea of the most efficient approaches is to start by sorting the edges according to their weights, and then simply go through that list of edges in order of increasing weight, and either add them to T, or reject them if they would produce a circle. There are implementations of that which can be achieved with overall time complexity $O(e\log_2 e)$, which is dominated by the $O(e\log_2 e)$ complexity of sorting the e edges in the first place.

This means that the choice between Prim's algorithm and Kruskal's algorithm depends on the connectivity of the particular graph under consideration. If the graph is sparse, i.e. the

number of edges is not much more than the number of vertices, then Kruskal's algorithm will have the same $O(n\log_2 n)$ complexity as the optimal priority queue based versions of Prim's algorithm, but will be faster than the standard $O(n^2)$ Prim's algorithm. However, if the graph is highly connected, i.e. the number of edges is near the square of the number of vertices, it will have complexity $O(n^2\log_2 n)$ and be slower than the optimal $O(n^2)$ versions of Prim's algorithm.

11.9 Travelling Salesmen and Vehicle Routing

Note that all the graph algorithms we have considered so far have had *polynomial time complexity*. There are further graph based problems that are even more complex.

Probably the most well known of these is the *Travelling Salesman Problem*, which involves finding the shortest path through a graph which visits each node precisely once. There are currently no known polynomial time algorithms for solving this. Since only algorithms with *exponential complexity* are known, this makes the Travelling Salesman Problem difficult even for moderately sized n (e.g., all capital cities). Exercise: write an algorithm in pseudocode that solves the Travelling Salesman Problem, and determine its time complexity.

A variation of the shortest path problem with enormous practical importance in transportation is the *Vehicle Routing Problem*. This involves finding a series of routes to service a number of customers with a fleet of vehicles with minimal cost, where that cost may be the number of vehicles required, the total distance covered, or the total driver time required. Often, for practical instances, there are conflicts between the various objectives, and there is a *trade-off* between the various costs which have to be balanced. In such cases, a *multi-objective optimization* approach is required which returns a *Pareto front* of non-dominated solutions, i.e. a set solutions for which there are no other solutions which are better on all objectives. Also, in practice, there are usually various *constraints* involved, such as fixed delivery time-windows, or limited capacity vehicles, that must be satisfied, and that makes finding good solutions even more difficult.

Since exact solutions to these problems are currently impossible for all but the smallest cases, *heuristic* approaches are usually employed, such as *evolutionary computation*, which deliver solutions that are probably good but cannot be proved to be optimal. One popular approach is to maintain a whole population of solutions, and use simulated evolution by natural selection to iteratively improve the quality of those solutions. That has the additional advantage of being able to generate a whole Pareto front of solutions rather than just a single solution. This is currently still a very active research area.

Appendix A

Some Useful Formulae

The symbols a, b, c, r and s represent real numbers, m and n are positive integers, and indices i and j are non-negative integers.

A.1 Binomial formulae

$(a+b)^2 = a^2 + 2ab + b^2$
$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$

$(a+b)(a-b) = a^2 - b^2$
$(a+b)^4 = a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4$

A.2 Powers and roots

$a^0 = 1$
$a^{-r} = 1/(a^r)$
$a^r a^s = a^{r+s}$
$a^s b^s = (ab)^s$
$(a^r)^s = a^{rs} = a^{sr} = (a^s)^r$

$a^1 = a$
$a^{1/n} = \sqrt[n]{a}$
$a^r/a^s = a^{r-s}$
$a^s/b^s = (a/b)^s$

and the following are special cases of the above:

$\sqrt[n]{a}\,\sqrt[n]{b} = \sqrt[n]{ab}$
$\sqrt[n]{a}/\sqrt[n]{b} = \sqrt[n]{a/b}$

$a^{m/n} = \sqrt[n]{a^m} = \sqrt[n]{a}^m$
$a^{-(m/n)} = 1/(\sqrt[n]{a^m}) = 1/(\sqrt[n]{a})^m$

A.3 Logarithms

Definition: The *logarithm of c to base a*, written as $\log_a c$, is the real number b satisfying the equation $c = a^b$, in which we assume that $c > 0$ and $a > 1$.

There are two special cases worth noting, namely $\log_a 1 = 0$, since $a^0 = 1$, and $\log_a a = 1$, since $a^1 = a$. From the definition, we immediately see that:

$$a^{\log_a c} = c \qquad \text{and} \qquad \log_a a^b = b$$

and we can move easily from one base a to another a' using:

$$\log_{a'} b = \log_{a'} a * \log_a b.$$

Using the above formula, the time complexity k is computed as follows:

$$k = \sum_{i=0}^{n-1} \sum_{j=0}^{i} 3 = \sum_{i=0}^{n-1} (i+1)3 = 3 \sum_{i=0}^{n-1} (i+1) = 3 \sum_{i=1}^{n} i = 3 \frac{n(n+1)}{2}.$$

Two more sums that often prove useful when computing complexities are:

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots = 2$$

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 0 + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots = 2$$

because they can be truncated at large n to give:

$$\sum_{i=0}^{n} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^n} \simeq 2 = O(1)$$

$$\sum_{i=0}^{n} \frac{i}{2^i} = 0 + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots + \frac{n}{2^n} \simeq 2 = O(1)$$

which are needed for computing the complexity of heap tree creation and certain special cases of quicksort.

A.5 Fibonacci numbers

The sequence of *Fibonacci numbers* F_n is defined recursively by

$$F_n = F_{n-1} + F_{n-2}$$

with the base values $F_0 = 0, F_1 = 1$. Thus the sequence begins 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...