Using lots of space to save lots of time.

Our desktop PCs have 32-bit integers; there are 2^{32} different integers in their range, or about 4×10^9 – 4 billion possible integers.

Suppose we have a sequence $A_{0..999}$ which contains integers; the problem is to find whether some integer x occurs in A or not.

Suppose we also have an array H which has 2^{32} elements. For each value y which does occur in A we put *true* in H_y; we put *false* in all the other elements.

> *we may have to play a trick with negative values of x to fool Java...*

To test whether integer x occurs in A we look at H_x; if we find *true* then x is in the sequence; if we find *false* then x is not in the sequence.

This is $O(1)$ – ***constant-time*** – searching, achieved at a huge space cost.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

*We can always save time, as in this case, by **pre-computing** all the answers and putting them into an array; then the cost of finding an answer is just the cost of looking in the array .. if we neglect the cost of setting-up the array of answers.*

Setting up H could be quick: it would be easy to build hardware which in a single memory cycle could flood the whole of H with *false*s, and then this $O(N)$ loop would put the *true*s in place:

```
for (k=m; k<n; k++)
  H[(long)A[k]&0xffffffffL]=true;
```

The arithmetical trickery in this example exploits the fact that we know that Java's `int`*s are 32 bits, and its* `long`*s are 64 bits.*

To look up an integer x:

```
H[(long)A[k]&0xffffffffL]
```

The space cost is huge, but just how huge?

Since we only have to store *true*s and *false*s in H, we could use a single bit per element; each byte of memory in our desktop PC has 8 bits, so we would need $2^{32}/8 = 2^{29} \approx 500$ megabytes.

> *At the time of writing memory is less than £2 a megabyte, so for about £1000 you can buy enough memory to hold array* H**.**
>
> *Java doesn't support bit-arrays, but there is no reason why it shouldn't.*

Here's $O(1)$-time code which would search for x in an array H of 2^{32} bits, represented by an array M of 2^{29} `byte`s:

```
M[x>>3]&(1<<(x&0x7))!=0;
  // x>>3 is (unsigned x)/8;
  // x&0x7 is (unsigned x)%8;
  // M[x>>3] picks a byte;
  // 1<<(x&0x7) picks a bit position;
  // & picks out the bit;
  // !=0 converts the answer to true or false
```


Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Constant-time searching of sequences of larger values using a similar technique, would be less practical, because there would be many more than 2^{32} possible values to be pre-indexed in *H*.

> *In some cases - e.g. strings - there is an infinite number of possible values, so we couldn't use this technique at all.*

In practice we have to be satisfied with something not quite so quick: ***hash addressing*** gives $O(1)$ performance and uses less space, but it may make more than a single comparison to find a value *x* in the sequence.

Hash addressing.

Hash addressing: index a table not with the key we are looking for, but with a ***hash key***: a number derived from the original key.

I assume a good deal of spare space – 1 megabyte, say – and the same sequence $A_{0..999}$ of 32-bit integers.

> *These days 1 megabyte isn't much memory: you'd easily offer it if that was the price you had to pay for fast $O(1)$ searching. Luckily, the price isn't that high.*

I assume also that we want to search A very often so that we won't be put off by setup costs, however high they turn out to be.

Hash addressing, (faulty) version 1 – a bit array Lb.

this version doesn't work, but it gets us closer to an understanding.

I assume that the machine and our compiler give us bit-addressing.

Suppose the spare megabyte holds an array *Lb* of bits: there is room for $2^{20} \times 8 = 2^{23}$ elements, so it will be impossible to give a unique entry to each element.

But we have only a thousand (about 2^{10}) integers to search, so there are many more elements of *Lb* than there are integers in *A*.

To enter or to look up an integer *x*, use $x \bmod (\text{size of } Lb)$: if there's a 1 at that position in *Lb* then *x* is in the sequence *A*; if there's a 0 then *x* isn't in the sequence *A*.

To initialise *Lb*, we must have some hardware which will flood it with 0s. Then we can insert the 1 bits, one at a time:

```
for (int k=0; k<1000; k++)
  Lb[A[k]&0x7fffff]=1;
```

and to look up an integer *x*:

```
Lb[x&0x7fffff]==1;
```

If there is a 1 in `Lb[x&0x7fffff]` then *A* contains an integer which shares its last 23 bits with *x*. ***But*** that number might not be *x* – it could be $x \pm 2^{24}$, $x \pm 2^{24} \pm 2^{25}$, ...

The test doesn't look at the top 9 bits of *x*, so there are 2^9 other integers which might be signalled by that 1.

We can't use a bit-array.

Hash addressing, version 2 – L an array of integers.

this is the basis of a solution – but we shall meet some snags.

When we looked for x in *Lb* we got a 'miss' (0) or we get a 'hit' (1). A 'miss' meant that x is definitely not in A. A 'hit' meant that x *might* be in A.

We need to distinguish between 'accidental' hits – x shares a hash index with a number which is in A – and 'correct' hits – x really is in A.

Instead of storing a 1 or a 0 in *Lb*, I'm going to store an integer in L. I have a megabyte of space, so L will have $2^{20}/4 = 2^{18}$ elements – about 250 000.

L is still much larger than A.

We shall use the last 18 bits of x to index L.

We shall assume, for reasons which will become clear, that 0 doesn't occur in A.

we shall see later how to relax this requirement.

We zero-flood L as usual.

Then we insert the values from A:

```
for (k=0; k<1000; k++)
  L[A[k]&0x3ffff]=A[k];
```

To look up an integer x:

```
x!=0 && L[x&0x3ffff]==x;
```

Suppose that $x \neq y$ but $x \bmod 2^{18} = y \bmod 2^{18} = i$. Then either $L_i = x$ or $L_i = y$: it can't be both.

We have created the problem of 'false misses': if $L_i = y$ we shall look in L for x and find y, yet perhaps x really does belong to A.

We can fix the problem of false misses.

Richard Bornat
Dept of Computer Science
QUEEN MARY AND WESTFIELD COLLEGE UNIVERSITY OF LONDON

Aside: collisions are quite likely.

When two search keys share the same entry in the hash table we have a *collision*.

Collisions are surprisingly likely, even though L is large and A is small.

When n people meet there is a chance that there will be a pair with the same birthday: the chance is $1 - \left(364/365 \times 363/365 \times \ldots \times 365-n/365\right)$, and in a group of only 23 people there is more than a 50% chance that there's a shared birthday.

The chance that two elements of a thousand-element array A share the same low 18 bits is $1 - \left(2^{18}-1/2^{18} \times 2^{18}-2/2^{18} \times \ldots \times 2^{18}-1000/2^{18}\right)$: 85% chance of at least one such coincidence, according to my calculations, despite the fact that L has more than 250 spare elements for every one that is used!!

Hash addressing version 3: handling collisions by 'rehashing'.

When we insert an element of A into some hash table element L_i we have to be careful: we might find the element we want to use is already 'full'.

An element is 'full', in my simplified treatment, if it is non-zero.

When we insert elements into L we look in the next position when we find a full one:

```
for (k=0; k<1000; k++) {
  for (int i = A[k]&0x3fff;
       L[i]!=0 && L[i]!=A[k]; i=(i+1)&0x3fff) ;
  L[i]=A[k];
}
```

the 'wrap round' calculation makes sure that if/when i reaches the end of L, it starts again at the beginning.

the loop stops when $L_i = 0 \vee L_i = A_k$

there are bound to be lots of free positions, given my assumptions about the sizes of L and A.

When we look in *L*, we make sure we don't give up until we have seen an empty element:

```
if (x==0) return false; // no zeroes in L
else {
  for (int i = x&0x3fff;
       L[i]!=x; i=(i+1)&0x3fff)
    if (L[i]==0) return false;
  return true; // loop terminates when L[i]==x
}
```

That code does a 'hash' of the number *x* to give an index *i* of *L*; it then does a sequential search from that position to find if *x* has been entered into *L*.

To get $O(1)$ performance we must ensure that the length of the sequential search is independent of the size of the sequence *A*; to get fast $O(1)$ performance we must ensure that the sequential search is on average very short.

Exact analysis supports our gut feeling that if the size of *L* is much larger than the size of *A*, then the sequential search will be very short; the same analysis also shows that we don't need such a large array as *L* to get $O(1)$ search times.

Exact analysis of 'linear rehash'.

Weiss (p615) gives the formula, and refers us to Knuth for the analysis.

Let λ be the 'load factor': the proportion of our 'hash table' that is used. In the case of L and A above, λ is very small, about $1/250$.

λ is the probability that when we compute an index i, we find a cell which is full; the probability that we find it empty is therefore $1 - \lambda$. Naively, the average number of times we have to look to find an empty entry might be expected to be $1/(1-\lambda)$, which in our case is $250/249$, pretty close to 1.

That analysis works for small values of λ, but at larger values it breaks down. We get 'clustering': those entries which hash to i get confused with those which hash to i+1, and both those sets get confused with those which hash to i+2, and so on and on.

The average number of probes to make a *successful* search, say Knuth, Sedgewick and Weiss, is

$$\frac{1 + 1/(1-\lambda)}{2}$$

The average number of probes to make an *unsuccessful* search is

$$\frac{1 + 1/(1-\lambda)^2}{2}$$

Here's a tabulation of those two functions for various values of λ :

Load factor	successful search (average)	unsuccessful search (average)
0.004	1.002	1.004
0.1	1.056	1.117
0.2	1.125	1.281
0.3	1.214	1.520
0.4	1.333	1.889
0.5	1.500	2.500
0.6	1.750	3.625
0.7	2.167	6.056
0.8	3.000	13.000
0.9	5.500	50.500

Only at load factors about 0.6 and above does the length of *unsuccessful* searches become more than twice the length of successful searches.

At a load factor of 0.5 you can find if something is in the table (or not) in less than 3 probes on average: fast as binary chop on a 8-element table, and *the efficiency of hash addressing depends on the load factor* (the relative sizes of *A* and *L*) ***not*** *on the size of L or A.*

Worst-case analysis for hash addressing is pretty fierce mathematics. I shan't even glance at in this course.

Hash addressing, at low load factors, is $O(1)$ and binary chop is $O(\lg N)$.

So hash addressing can easily be made faster than binary chop.

The exact analysis supposes that the keys we are searching for are randomly distributed. In practice there may be regularities which will increase the size of clusters, and in practice those clusters can be broken up by 're-hashing' techniques which replace the sequential search shown above.

A nice feature of hash addressing: *you can insert new entries into the table at any time*, without disturbing things that are there already. (But watch that load factor!) In a binary chop table, you have to move half – or more – of the entries out of the way.

A not-very-nice feature of hash addressing: deleting entries is not so easy. You can't replace an entry with 'empty', because that would hide all the entries that are in its sequential search cluster. You need to be able to distinguish between 'full', 'empty' and 'deleted'.

Weiss has an example implementation (pp 567-572). You are expected to read and understand it.

If you aren't searching for integers, but (say) for strings, you may have to use a more interesting hash function.

> *making the size of the table a prime number p, treating the string as a large binary number S, and using* $S \bmod p$ *is a favourite trick of mine. That hash function takes account of all the bits in S. Weiss talks about this (pp 555-557).*
>
> *If you find that your keys don't give a nice spread of indices in the hash table – and often they don't – there are fancy 're-hashing' schemes which minimise the size of clusters. Weiss describes 'quadratic rehash' (pp565).*
>
> *But just remember that hash addressing is fast, even using the techniques shown in this lecture, provided only that you are careful to keep the load factor down.*

See Knuth for more than you imagine you might need to know about hash addressing.

When you do a second or a third-year project, don't forget hash addressing!

Key points

A hash key is a hash table index, derived from a database key.

The hash table contains all the database keys, used to disambiguate collisions, plus the corresponding data (or a reference to it).

Load factor is the ratio of hash table size to database size.

Hash addressing always costs a lot in space; if the table size is proportional to database size we may call it $O(N)$.

At load factors (<0.5) hash addressing gives $O(1)$ search time.

The linear rehash mechanism may cause secondary clustering; other rehash techniques reduce this.

Hash addressing has $O(N)$ setup time.

Insertion is difficult to cost, because of the possibility that the table may have to be resized: normally it's $O(1)$, but occasionally it will be $O(N)$, because a new table must be built.

Deletion costs $O(1)$, but the deleted entry can't be reclaimed until the table is rebuilt.