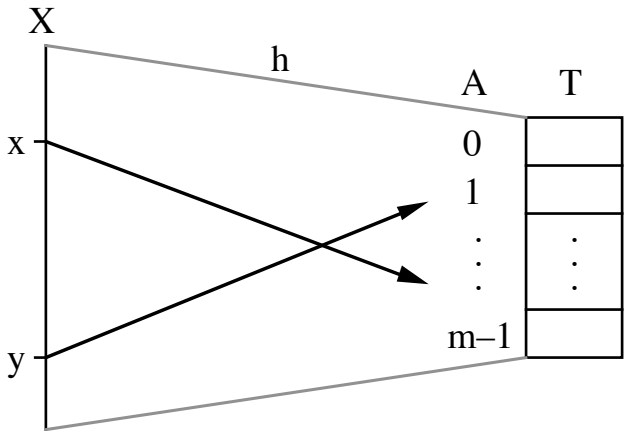
Ch 6 Randomized algorithms

Ecclesiastes:
.. and saw under the sun,
that the race is not to the swift,
nor the battle to the strong,
neither yet bread to the wise,
nor yet riches to men of understanding,
nor yet favor to men of skill;
but time and chance
happeneth to them all.

6.1 Performance analysis of hashing with open addressing



Hashing was the first prominent use of randomization in computing (the second one was quicksort). A sample of items (keys, think of names of people) x, y, .. drawn from a large key domain X (think of all character strings of a fixed length) must be stored in a table (think of a telephone directory) with a much smaller address space A[0 .. m-1]. The purpose of the hash function h: X -> A is to transform a possibly unknown probability distribution over X into a uniform distribution over A. Since h is chosen before the values to be stored are known, collisions are inevitable: $x \neq y$ yet $h(x) = h(y)$. Among various collision resolution techniques we consider **open addressing.**

Assign to each element $x \in X$ a *probe sequence* $a0 = h(x), a1, a2, \ldots$ of addresses that fills the entire address range A. The intention is to store x preferentially at a0, but if T[a0] is occupied then at a1, and so on, until the first empty cell is encountered along the probe sequence. The occupied cells along the probe sequence are called the *collision path* of x - note that the collision path is a prefix of the probe sequence. If we enforce the *invariant*:

If x is in the table at T[a] and if i precedes a in the probe sequence for x,
then T[i] is occupied.

the following fast and simple loop that travels along the collision path can be used to search for x:

a := h(x);
while T[a] ≠ x and T[a] ≠ empty do a := (next address in probe sequence);

Let us work out the details so that this loop terminates correctly and the code is as concise and fast as we can make it. The probe sequence is defined by formulas in the program (an example of an implicit data structure) rather than by pointers in the data as is the case in coalesced chaining.

Example: Linear probing
$a_{i+1} = (ai + 1) \bmod m$ is the simplest possible formula. Its only disadvantage is a phenomenon called *clustering*. Clustering arises when the collision paths of many elements in the table overlap to a large extent, as is likely to happen in linear probing. Once elements have collided, linear probing will store them in consecutive cells. All elements that hash into this block of contiguous occupied cells travel along the same collision path, thus lengthening this block; this in turn increases the probability that future elements will hash into this block. Once this positive feedback loop gets started, the cluster keeps growing.

Double hashing is a special type of open addressing designed to alleviate the clustering problem by letting different elements travel with steps of different size. The probe sequence is defined by the formulas

$a0 = h(x),\ d = g(x) > 0,\ ai+1 = (ai + d) \bmod m,\ m$ prime

g is a second hash function that maps the key space X into [1 .. m – 1].

Two important important details must be solved:
1. The probe sequence of each element must span the entire address range A. This is achieved if m is relatively prime to every step size d, and the easiest way to guarantee this condition is to choose m prime.
2. The termination condition of the search loop above is: T[a] = x or T[a] = empty. An unsuccessful search (x not in the table) can terminate only if an address a is generated with T[a] = empty. We have already insisted that each probe sequence generates all addresses in A. In addition, we must guarantee that the table contains at least one empty cell at all times - this serves as a sentinel to terminate the search loop.

The following declarations and procedures implement double hashing. We assume that the comparison operators = and ≠ are defined on X, and that X contains a special value 'empty', which differs from all values to be stored in the table. For example, a string of blanks might denote 'empty' in a table of identifiers. We choose to identify an unsuccessful search by simply returning the address of an empty cell.

```
const       m = … ;  { size of hash table - must be prime! }
            empty = … ;
type        key = … ;  addr = 0 .. m – 1;  step = 1 .. m – 1;
var         T: array[addr] of key;
            n: integer;  { number of elements currently stored in T }
function h(x: key): addr;  { hash function for home address }
function g(x: key): step;  { hash function for step }

procedure init;
var  a: addr;
begin
    n := 0;
    for a := 0 to m – 1 do  T[a] := empty
end;

function find(x: key): addr;
var  a: addr;  d: step;
begin
    a := h(x);  d := g(x);
    while  (T[a] ≠ x) and (T[a] ≠ empty)  do  a := (a + d) mod m;
    return(a)
end;

function insert(x: key): addr;
var  a: addr;  d: step;
begin
    a := h(x);  d := g(x);
    while  T[a] ≠ empty  do  begin
        if  T[a] = x  then  return(a);
        a := (a + d) mod m
    end;
    if  n < m – 1  then  { n := n + 1;  T[a] := x }  else  err-msg('table is full');
    return(a)
end;
```

Deletion of elements creates problems, as is the case in many types of hash tables. An element to be deleted cannot simply be replaced by 'empty', or else it might break the collision paths of other elements still in the table - recall the basic invariant on which the correctness of open addressing is based. The idea of rearranging elements in the table so as to refill a cell that was emptied but needs to remain full is quickly abandoned as too complicated - if deletions are numerous, the programmer ought to choose a data structure that fully supports deletions, such as balanced trees implemented as list structures. A limited number of deletions can be accommodated in an open address hash table by using the following technique.
At any time, a cell is in one of three states:
- empty (was never occupied, the initial state of all cells)
- occupied (currently)
- deleted (used to be occupied but is currently free)

A cell in state 'empty' terminates the find loop; a cell in state 'empty' or in state 'deleted' terminates the insert loop. The state diagram shown in Fig. 22.4 describes the transitions possible in the lifetime of a cell. Deletions degrade the performance of a hash table, because a cell, once occupied, never returns to the virgin state 'empty' which alone terminates an unsuccessful find. Even if an equal number of insertions and deletions keeps a hash

table at a low load factor l, unsuccessful finds will ultimately scan the entire table, as all cells drift into one of the states 'occupied' or 'deleted'. Before this occurs, the table ought to be rehashed; that is, the contents are inserted into a new, initially empty table.

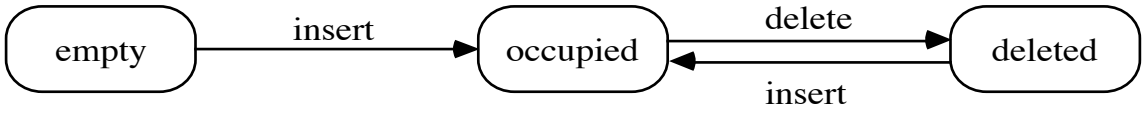


Figure 22.4: This state diagram describes possible life cycles of a cell:
Once occupied, a cell will never again be as useful as an empty cell.

Exercise: Hash table with deletions
Modify the program above to implement double hashing with deletions.

Performance analysis
We analyze open addressing without deletions assuming that each address αi is chosen independently of all other addresses from a uniform distribution over A. This assumption is reasonable for double hashing and leads to the conclusion that the average cost for a search operation in a hash table is O(1) if we consider the load factor λ to be constant. We analyze the average number of probes executed as a function of λ in two cases: $U(\lambda)$ for an unsuccessful search, and $S(\lambda)$ for a successful search.
Let pi denote the probability of using *exactly* i probes in an unsuccessful search. This event occurs if the first (i – 1) probes hit occupied cells, and the i-th probe hits an empty cell: $pi = \lambda^{i-1} \cdot (1 - \lambda)$. Let qi denote the probability that *at least* i probes are used in an unsuccessful search; this occurs if the first i – 1 inspected cells are occupied: $qi = \lambda^{i-1}$. qi can also be expressed as the sum of the probabilities that we probe exactly j cells, for j running from i to m. Thus we obtain

The number of probes executed in a successful search for an element x equals the number of probes in an unsuccessful search for the same element x before it is inserted into the hash table. [Note: This holds only when elements are never relocated or deleted]. Thus the average number of probes needed to search for the i-th element inserted into the hash table is $U((i - 1) / m)$, and $S(\lambda)$ can be computed as the average of $U(\mu)$, for μ increasing in discrete steps from 0 to λ. It is a reasonable approximation to let μ vary continuously in the range from 0 to λ:

Figure 22.5 suggests that a reasonable operating range for a hash table keeps the load factor λ between 0.25 and 0.75. If λ is much smaller, we waste space, if it is larger than 75%, we get into a domain where the performance degrades rapidly. Note: If all searches are successful, a hash table performs well even if loaded up to 95% - unsuccessful searching is the killer!

λ	0.25	0.5	0.75	0.9	0.95	0.99
$U(\lambda)$	1.3	2.0	4.0	10.0	20.0	100.0
$S(\lambda)$	1.2	1.4	1.8	2.6	3.2	4.7

Figure 22.5: The average number of probes per search grows rapidly as the load factor approaches 1.

Thus the hash table designer should be able to estimate n within a factor of 2 - not an easy task. An incorrect guess may waste memory or cause poor performance, even table overflow followed by a crash. If the programmer becomes aware that the load factor lies outside this range, she may rehash - change the size of the table, change the hash function, and reinsert all elements previously stored.

6.2 Communication protocol for file verification

Consider a server X and a client Y that communicate over a thin communication line. X stores a file that we also call X, Y stores a file Y, and the question to be answered is: $X = Y$ or $X \neq Y$?
Depending on context, $X = x1, x2, .., xn$ and $Y = y1, y2, .., yn$ are considered as bitstrings or as integers.

It is evident that, in general, any deterministic file verification protocol must transmit at least n bits between X and Y. If the protocol's answer is $X = Y$ although some bit xi is ignored and not compared to its mate yi, an adversary could set the value of bit xi to obtain the state $X \neq Y$ and thereby invalidate the protocol's answer to the file identity question. We present a randomized protocol that "solves" the file verification problem by transmitting only O(log n) bits, at the cost of incurring a negligible risk of making the wrong decision. We assume the communication is error free. This is a realistic assumption since the messages to be transmitted turn out to be so short that error correcting codes can well be afforded.

Randomized file verification protocol:
1) X chooses a prime p at random in $[1 .. n^2]$ with uniform probability
2) X computes $s = X \bmod p$
3) X sends (p, s) to Y
4) X computes $r = Y \bmod p$
5) Y's decision procedure: $s \neq r \rightarrow X \neq Y$; $s = r \rightarrow X = Y$

Observations about these steps:
1) Prime number theorem: (number of primes $\leq N$) $\sim N / \ln N$
   Applied to our case: (number of primes in $[1 .. n^2]$) $\sim n^2 / \ln n^2$
3) Both p and s are in $[1 .. n^2]$, hence the size of the message (p, s): $| p, s | \leq 2 \log n^2 = 4 \log n$
   I.e., instead of sending n bits, we only send 4 log n bits.
5) Consider the 4 cases of the decision process: $s \neq r \rightarrow X \neq Y$; $s = r \rightarrow X = Y$
   $X = Y$ & $s = r$ and $X \neq Y$ & $s \neq r$ give the correct answer
   $X = Y$ & $s \neq r$ cannot occur under the assumption of error free communication
   $X \neq Y$ & $s = r$ is a possible incorrect result - what is the probability of error?

Let's compute the probability of the event $X \neq Y$ & $s = r$, leading to the the erroneous conclusion $X = Y$.
Call the event $X \bmod p = Y \bmod p$: "p is a bad prime".
It occurs iff $| X - Y |$ is a multiple of p, in other words, iff p divides $| X - Y |$.
How many "bad primes" are there, i.e. how many distinct prime factors can $| X - Y |$ have?

Lemma: $| X - Y |$ has $\leq n-1$ distinct prime factors.

Pf of lemma:
Observe: $X < 2^n$, $X < 2^n$, and hence $| X - Y | < 2^n$.
Let $| X - Y | = p_1^{e1} p_2^{e2} ... p_k^{ek}$,
where $e_j > 0$ and the distinct prime factors have been ordered as $p_1 < p_2 < ... < p_k$.
Assume $k \geq n$ and derive a contradiction:
If $k \geq n$ then $| X - Y | = p_1^{e1} p_2^{e2} ... p_k^{ek} \geq p_1 p_2 ... p_k \geq p_1 p_2 ... p_n > 1 \cdot 2 \cdot ... \cdot n = n!$
Thus, we derive $n! < | X - Y | < 2^n$, but $n! > 2^n$ for $n > 3$. This contradiction proves the lemma.

The conclusion $| X - Y |$ has $\leq n-1$ distinct prime factors can be expressed as
"there are at most n -1 bad primes out of $n^2 / \ln n^2$ ".
Thus, the probability of picking a bad prime, i.e. the error probability, is
$e < (n-1) / n^2 / \ln n^2 < \ln n^2 / n = 2 \ln n / n$

The error probability e tends to 0 rapidly with growing n.
Ex: For $n = 10^{12}$ we have $| p, s | = 160$ and $e < 10^{-10}$ - an extremely short message of 160 bits produces a negligible error rate.