22. Address computation

Learning objectives:

- hashing
- perfect hashing
- collision resolution methods: separate chaining, coalesced chaining, open addressing (linear probing and double hashing)
- deletions degrade performance of a hash table
- Performance does not depend on the number of data elements stored but on the load factor of the hash table.
- randomization: transform unknown distribution into a uniform distribution
- Extendible hashing uses a radix tree to adapt the address range dynamically to the contents to be stored; deletions do not degrade performance.
- order-preserving extendible hashing

Concepts and terminology

The term *address computation* (also *hashing*, *hash coding*, *scatter storage*, or *key-to-address transformations*) refers to many search techniques that aim to assign an *address* of a storage cell to any *key value* x by means of a formula that depends on x only. Assigning an address to x independently of the presence or absence of other key values leads to faster access than is possible with the comparative search techniques discussed in earlier chapters. Although this goal cannot always be achieved, address computation does provide the fastest access possible in many practical situations.

We use the following concepts and terminology (Exhibit 22.1). The *home address* a of x is obtained by means of a *hash function* h that maps the *key domain* X into the *address space* A [i.e. a = h(x)]. The address range is A = {0, 1, ... , m – 1}, where m is the number of storage cells available. The storage cells are represented by an array T[0 .. m – 1], the *hash table*; T[a] is the cell addressed by a ∊ A. T[h(x)] is the cell where an element with key value x is *preferentially* stored, but alas, not necessarily.

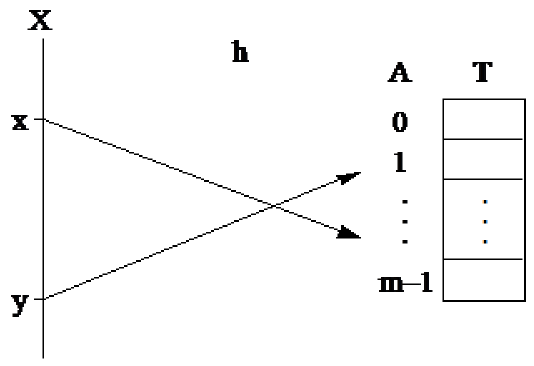

Exhibit 22.1: The hash function h maps a (typically large) key domain X into a (much smaller) address space A.

Each cell has a capacity of $b > 0$ elements; b stands for *bucket capacity*. The number n of elements to be stored is therefore bounded by $m \cdot b$. Two cases are usefully distinguished, depending on whether the hash table resides on disk or in central memory:

1. Disk or other secondary storage device: Considerations of efficiency suggest that a bucket be identified with a physical unit of transfer, typically a disk block. Such a unit is usually large compared to the size of an element, and thus $b > 1$.
2. Main memory: Cell size is less important, but the code is simplest if a cell can hold exactly one element (i.e. $b = 1$).

For simplicity of exposition we assume that $b = 1$ unless otherwise stated; the generalization to arbitrary b is straightforward.

The key domain X is normally much larger than the number n of elements to be stored and the number m of available cells T[a]. For example, a table used for storing a few thousand identifiers might have as its key domain the set of strings of length at most 10 over the alphabet {'a', 'b', ... , 'z', '0', ... , '9'}; its cardinality is close to 36^{10}. Thus in general the function h is many-to-one: Different key values map to the same address.

The content to be stored is a sample from the key domain: It is not under the programmer's control and is usually not even known when the hash function and table size are chosen. Thus we must expect *collisions*, that is, events where more than b elements to be stored are assigned the same address. *Collision resolution* methods are designed to handle this case by storing some of the colliding elements elsewhere. The more collisions that occur, the longer the search time. Since the number of collisions is a random event, the search time is a random variable. Hash tables are known for excellent average performance and for terrible worst-case performance, which, one hopes, will never occur.

Address computation techniques support the operations 'find' and 'insert' (and to a lesser extent also 'delete') in expected time O(1). This is a remarkable difference from all other data structures that we have discussed so far, in that the average time complexity does not depend on the number n of elements stored, but on the *load factor* $\lambda = n / (m \cdot b)$, or, for the special case $b = 1$: $\lambda = n / m$. Note that $0 \le \lambda \le 1$.

Before we consider the typical case of a hash table, we illustrate these concepts in two special cases where everything is simple; these represent ideals rarely attainable.

The special case of small key domains

If the number of possible key values is less than or equal to the number of available storage cells, h can map X one-to-one into or onto A. Everything is simple and efficient because collisions never occur. Consider the following example:

X = {'a', 'b', ... , 'z'}, A = {0, ... , 25}

h(x) = ord(x) – ord('a'); that is,

h('a') = 0, h('b') = 1, h('c') = 2, ... , h('z') = 25.

Since h is one-to-one, each key value x is implied by its address h(x). Thus we need not store the key values explicitly, as a single bit (present / absent) suffices:

```
var  T: array[0 .. 25] of boolean;

function member(x): boolean;
begin  return(T[h(x)])  end;
```

```
procedure insert(x);
begin  T[h(x)] := true  end;

procedure delete(x);
begin  T[h(x)] := false  end;
```

The idea of collision-free address computation can be extended to large key domains through a combination of address computation and list processing techniques, as we will see in the chapter "Metric data structures".

The special case of perfect hashing: table contents known a priori

Certain common applications require storing a set of elements that never changes. The set of reserved words of a programming language is an example; when the lexical analyzer of a compiler extracts an identifier, the first issue to be determined is whether this is a reserved word such as 'begin' or 'while', or whether it is programmer defined. The special case where the table contents are known a priori, and no insertions or deletions occur, is handled more efficiently by special-purpose data structures than by a general dictionary.

If the elements x_1, x_2, ... , x_n to be stored are known before the hash table is designed, the underlying key domain is not as important as the set of actually occurring key values. We can usually find a table size m, not much larger than the number n of elements to be stored, and an easily evaluated hash function h that assigns to each x_i a unique address from the address space {0, ... , m − 1}. It takes some trial and error to find such a *perfect hash function* h for a given set of elements, but the benefit of avoiding collisions is well worth the effort—the code that implements a collision-free hash table is simple and fast. A perfect hash function works for a static table only—a single insertion, after h has been chosen, is likely to cause a collision and destroy the simplicity of the concept and efficiency of the implementation. Perfect hash functions should be generated automatically by a program.

The following unrealistically small example illustrates typical approaches to designing a perfect hash table. The task gets harder as the number m of available storage cells is reduced toward the minimum possible, that is, the number n of elements to be stored.

Example

In designing a perfect hash table for the elements 17, 20, 24, 38, and 51, we look for arithmetic patterns. These are most easily detected by considering the binary representations of the numbers to be stored:

	5	4	3	2	1	0	bit position
17	0	1	0	0	0	1	
20	0	1	0	1	0	0	
24	0	1	1	0	0	0	
38	1	0	0	1	1	0	
51	1	1	0	0	1	1	

We observe that the least significant three bits identify each element uniquely. Therefore, the hash function h(x) = x mod 8 maps these five elements collision-free into the address space A = {0, ... , 6}, with m = 7 and two empty cells. An attempt to further economize space leads us to observe that the bits in positions 1, 2, and 3, with weights 2, 4, and 8 in the binary number representation, also identify each element uniquely, while ranging over the address space of minimal size A = {0, ... , 4}. The function h(x) = (x div 2) mod 8 extracts these three bits and assigns the following addresses:

X:	17	20	24	38	51
A:	0	2	4	3	1

A perfect hash table has to store each element explicitly, not just a bit (present/absent). In the example above, the elements 0, 1, 16, 17, 32, 33, ... all map into address 0, but only 17 is present in the table. The access function 'member(x)' is implemented as a single statement:

```
        return ((h(x) ≤ 4) cand (T[h(x)] = x));
```

The boolean operator 'cand' used here is understood to be the *conditional and*: Evaluation of the expression proceeds from left to right and stops as soon as its value is determined. In our example, h(x) > 4 suffices to assign 'false' to the expression (h(x) ≤ 4) and (T[h(x)] = x). Thus the 'cand' operator guarantees that the table declared as:

```
        var  T: array[0 .. 4] of element;
```

is accessed within its index bounds.

For table contents of realistic size it is impractical to construct a perfect hash function manually—we need a program to search exhaustively through the large space of functions. The more slack m – n we allow, the denser is the population of perfect functions and the quicker we will find one. [Meh 84a] presents analytical results on the complexity of finding perfect hash functions.

Exercise: perfect hash tables

Design several perfect hash tables for the content {3, 13, 57, 71, 82, 93}.

Solution

Designing a perfect hash table is like answering a question of the type: What is the next element in the sequence 1, 4, 9, ... ? There are infinitely many answers, but some are more elegant than others. Consider:

h	3	13	57	71	82	93	Address range
(x div 3) mod 7	1	4	5	2	6	3	[1 .. 6]
x mod 13	3	0	5	6	4	2	[0 .. 6]
(x div 4) mod 8	0	3	6	1	4	7	[0 .. 7]
if x = 71 then 4 else x mod 7	3	6	1	4	5	2	[1 .. 6]

Conventional hash tables: collision resolution

In contrast to the special cases discussed, most applications of address computation present the data structure designer with greater uncertainties and less favorable conditions. Typically, the underlying key domain is much larger than the available address range, and not much is known about the elements to be stored. We may have an upper bound on n, and we may know the probability distribution that governs the random sample of elements to be stored. In setting up a customer list for a local business, for example, the number of customers may be bounded by the population of the town, and the distribution of last names can be obtained from the telephone directory—many names will start with H and S, hardly any with Q and Y. On the basis of such information, but in ignorance of the actual table contents to be stored, we must choose the size m of the hash table and design the hash function h that maps the key domain X into the address space A= {0, ... , m – 1}. We will then have to live with the consequences of these decisions, at least until we decide to *rehash*: that is, resize the table, redesign the hash function, and reinsert all the elements that we have stored so far.

Later sections present some pragmatic advice on the choice of h; for now, let us assume that an appropriate hash function is available. Regardless of how smart a hash function we have designed, collisions (more than b elements share the same home address of a bucket of capacity b) are inevitable in practice. Thus hashing requires techniques for handling collisions. We present the three major *collision resolution* techniques in use: separate chaining, coalesced chaining, and open addressing. The two techniques called *chaining* call upon list processing techniques to organize overflowing elements. *Separate chaining* is used when these lists live in an overflow area distinct from the hash table proper; *coalesced chaining* when the lists live in unused parts of the table. *Open addressing* uses address computation to organize overflowing elements. Each of these three techniques comes in different variations; we illustrate one typical choice.

Separate chaining

The memory allocated to the table is split into a *primary* and an *overflow* area. Any overflowing cell or bucket in the primary area is the head of a list, called the *overflow chain*, that holds all elements that overflow from that bucket. Exhibit 22.2 shows six elements inserted in the order x_1, x_2, … . The first arrival resides at its home address; later ones get appended to the overflow chain.

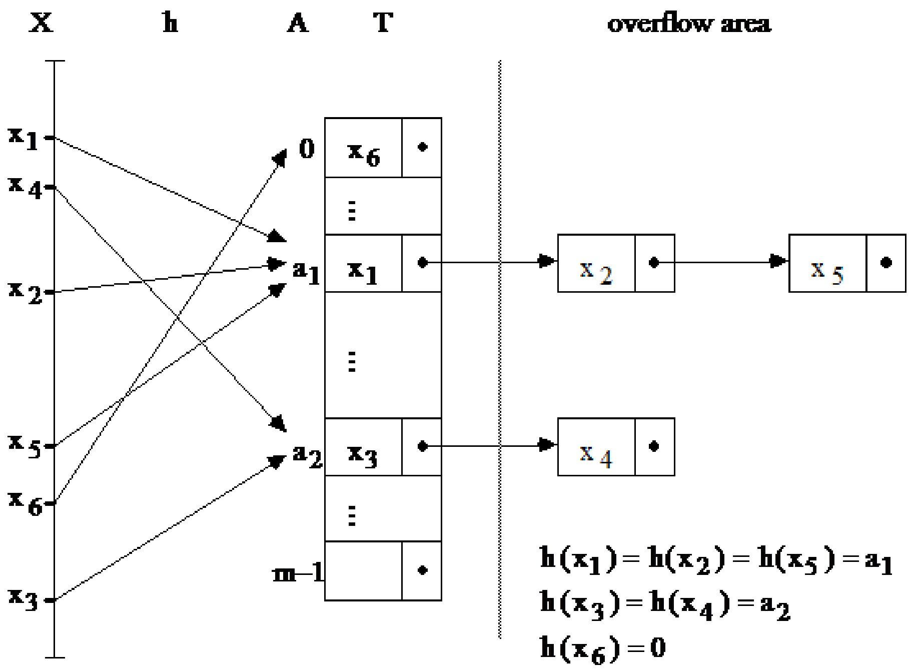

Exhibit 22.2: Separate chaining handles collisions in a separate overflow area.

Separate chaining is easy to understand: insert, delete, and search operations are simple. In contrast to other collision handling techniques, this hybrid between address computation and list processing has two major advantages: (1) deletions do not degrade the performance of the hash table, and (2) regardless of the number m of home addresses, the hash table will not overflow until the entire memory is exhausted. The size m of the table has a critical influence on the performance. If m « n, overflow chains are long and we have essentially a list processing technique that does not support direct access. If m » n, overflow chains are short but we waste space in the table. Even for the practical choice m ≈ n, separate chaining has some disadvantages:

- Two different accessing techniques are required.
- Pointers take up space; this may be a significant overhead for small elements.

- Memory is partitioned into two separate areas that do not share space: If the overflow area is full, the entire table is full, even if there is still space in the array of home cells. This consideration leads to the next technique.

Coalesced chaining

The chains that emanate from overflowing buckets are stored in the empty space in the hash table rather than in a separate overflow area (Exhibit 22.3). This has the advantage that all available space is utilized fully (except for the overhead of the pointers). However, managing the space shared between the two accessing techniques gets complicated.

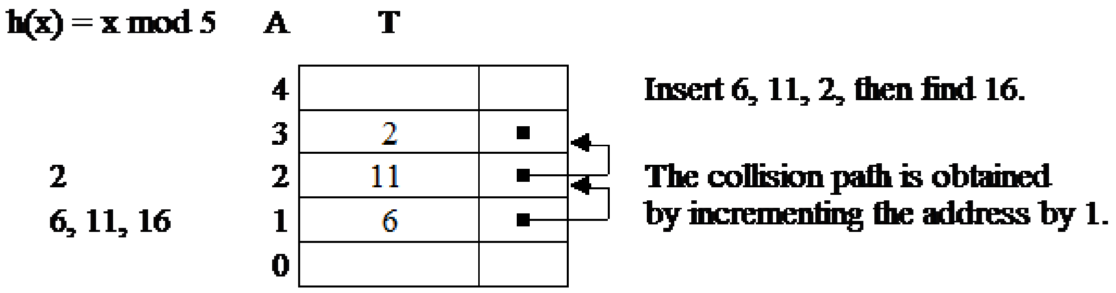
h(x) = x mod 5
A
T
4
3
2
1
0
2
11
6
2
6, 11, 16
Insert 6, 11, 2, then find 16.
The collision path is obtained by incrementing the address by 1.

Exhibit 22.3: Coalesced chaining handles collisions by building lists that share memory with the hash table.

The next technique has similar advantages (in addition, it incurs no overhead for pointers) and disadvantages; all things considered, it is probably the best collision resolution technique.

Open addressing

Assign to each element $x \in X$ a *probe sequence* $a_0 = h(x), a_1, a_2, \dots$ of addresses that fills the entire address range A. The intention is to store x preferentially at a_0, but if $T[a_0]$ is occupied then at a_1, 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

$a_{i+1} = (a_i + 1) \bmod m$ 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

$$a_0 = h(x),\ \ \delta = g(x) > 0,\ \ a_{i+1} = (a_i + \delta) \bmod m,\ \ m \text{ prime}$$

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

Two important important details must be solved:

- 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 δ, and the easiest way to guarantee this condition is to choose m prime.
- 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 Exhibit 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 λ, 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.

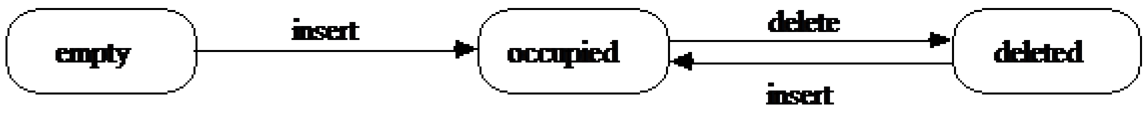

Exhibit 22.4: This state diagram describes possible life cycles of a cell: Once occupied, a cell will never again be as useful as an empty cell.

Exercise: hash table with deletions

Modify the program above to implement double hashing with deletions.

Choice of hash function: randomization

In conventional terminology, hashing is based on the concept of **randomization**. The purpose of randomizing is to transform an **unknown distribution** over the key domain X into a uniform distribution, and to turn consecutive samples that may be dependent into independent samples. This task appears to call for magic, and indeed, there is little or no mathematics that applies to the construction of hash functions; but there are commonsense observations worth remembering. These observations are primarily "don'ts". They stem from properties that sets of elements we wish to store frequently possess, and thus are based on *some* knowledge about the populations to be stored. If we assumed strictly nothing about these populations, there would be little to say about hash functions: an order-preserving proportional mapping of X into A would be as good as any other function. But in practice it is not, as the following examples show.

1. A Fortran compiler might use a hash table to store the set of identifiers it encounters in a program being compiled. The rules of the language and human habits conspire to make this set a highly biased sample from the set of legal Fortran identifiers. *Example:* Integer variables begin with I, J, K, L, M, N; this convention is likely to generate a cluster of identifiers that begin with one of these letters. *Example:* Successive identifiers encountered cannot be considered independent samples: If X and Y have occurred, there is a higher chance for Z to follow than for WRKHG. *Example:* Frequently, we see sequences of identifiers or statement numbers whose character codes form arithmetic progressions, such as A1, A2, A3, ... or 10, 20, 30, ...
2. All file systems require or encourage the use of naming conventions, so that most file names begin or end with one of just a few prefixes or suffixes, such as ···.SYS, ···.BAK, ···.OBJ. An individual user, or a user community, is likely to generate additional conventions, so that most file names might begin, for example, with the initials of the names of the people involved. The files that store this text, for example, are structured according to 'part' and 'chapter', so we are currently in file P5 C22. In some directories, file names might be sorted alphabetically, so if they are inserted into a table in order, we process a monotonic sequence.

   The purpose of a hash function is to break up all regularities that might be present in the set of elements to be stored. This is most reliably achieved by "hashing" the elements, a word for which the dictionary offers the following explanations: (1) from the French *hache*, "battle-ax"; (2) to chop into small pieces; (3) to confuse, to muddle. Thus, to approximate the elusive goal of randomization, a hash function destroys patterns, including, unfortunately, the order < defined on X. Hashing typically proceeds in two steps.

   1. Convert the element x into a number #(x). In most cases #(x) is an integer, occasionally, it is a real number $0 \le \#(x) < 1$. Whenever possible, this conversion of x into #(x) involves no action at all: The representation of x, whatever type x may be, is reinterpreted as the representation of the number #(x). When x is a variable-length item, for example a string, the representation of x is partitioned into pieces of suitable length that are "folded" on top of each other. For example, the four-letter word x = 'hash' is encoded one letter per byte using the 7-bit ASCII code and a leading 0 as 01101000 01100001 01110011 01101000. It may be folded to form a 16-bit integer by exclusive-or of the leading pair of bytes with the trailing pair of bytes:

      $$\begin{array}{rl} & 0110100001100001 \\ \text{xor} & \underline{0111001101101000} \\ & 0001101100001001 \end{array}$$ which represents $\#(x) = 27 \cdot 2^8 + 9 = 6921$.

      Such folding, by itself, is not hashing. Patterns in the representation of elements easily survive folding. For example, the leading 0 we have used to pad the 7-bit ASCII code to an 8-bit byte remains a zero regardless of x. If we had padded with a trailing zero, all #(x) would be even. Because #(x) often has the same representation as x, or a closely related one, we drop #() and use x slightly ambiguously to denote both the original element and its interpretation as a number.
   2. Scramble x [more precisely, #(x)] to obtain h(x). Any scrambling technique is a sensible try, as long as it avoids fairly obvious pitfalls. Rules of thumb:

- Each bit of an address h(x) should depend on all bits of the key value x. In particular, don't ignore any part of x in computing h(x). Thus $h(x) = x \bmod 2^{13}$ is suspect, as only the least significant 13 bits of x affect h(x).
- Make sure that arithmetic progressions such as Ch1, Ch2, Ch3, ... get broken up rather than being mapped into arithmetic progressions. Thus h(x) = x mod k, where k is significantly smaller than the table size m, is suspect.
- Avoid any function that cannot produce a uniform distribution of addresses. Thus $h(x) = x^2$ is suspect; if x is uniformly distributed in [0, 1], the distribution of x^2 is highly skewed.

A hash function must be fast and simple. All of the desiderata above are obtained by a hash function of the type:

h(x) = x mod m

where m is the table size and a prime number, and x is the key value interpreted as an integer.

No hash function is guaranteed to avoid the worst case of hashing, namely, that all elements to be stored collide on one address (this happens here if we store only multiples of the prime m). Thus a hash function must be judged in relation to the data it is being asked to store, and usually this is possible only after one has begun using it. Hashing provides a perfect example for the injunction that the programmer must think about the data, analyze its statistical properties, and adapt the program to the data if necessary.

Performance analysis

We analyze open addressing without deletions assuming that each address α_i 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 p_i 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: $p_i = \lambda^{i-1} \cdot (1 - \lambda)$. Let q_i 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: $q_i = \lambda^{i-1}$. q_i 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

$$U(\lambda) = \sum_{i=1}^{m} i \cdot p_i = \sum_{i=1}^{m} \sum_{j=i}^{m} p_j = \sum_{i=1}^{m} q_i = \sum_{i=1}^{m} \lambda^{i-1} \approx \frac{1}{1-\lambda}.$$

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:

$$S(\lambda) \approx \frac{1}{\lambda} \int_0^{\lambda} U(\mu)\, d\mu = \frac{1}{\lambda} \int_0^{\lambda} \frac{d\mu}{1-\mu} = \frac{1}{\lambda} \ln \frac{1}{1-\lambda}.$$

Exhibit 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 per cent, 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 per cent—unsuccessful searching is the killer!

Table 22.1: The average number of probes per search grows rapidly as the load factor approaches 1.

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

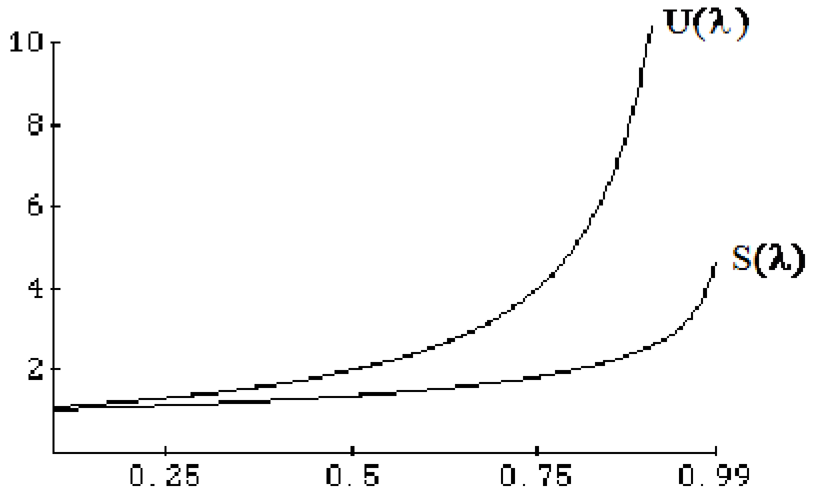

Exhibit 22.5: The average number of probes per search grows rapidly as the load factor approaches 1.

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.

Extendible hashing

In contrast to standard hashing methods, extendible forms of hashing allow for the dynamic extension or shrinkage of the address range into which the hash function maps the keys. This has two major advantages: (1) Memory is allocated only as needed (it is unnecessary to determine the size of the address range a priori), and (2) deletion of elements does not degrade performance. As the address range changes, the hash function is changed in such a way that only a few elements are assigned a new address and need to be stored in a new bucket. The idea that makes this possible is to map the keys into a very large address space, of which only a portion is active at any given time.

Various extendible hashing methods differ in the way they represent and manage a smaller *active address range* of variable size that is a subrange of a larger *virtual address range*. In the following we describe the method of extendible hashing that is especially well suited for storing data on secondary storage devices; in this case an address points to a physical block of secondary storage that can contain more than one element. An address is a bit string of maximum length k; however, at any time only a prefix of d bits is used. If all bit strings of length k are represented by a so-called *radix tree* of height k, the active part of all bit strings is obtained by using only the upper d levels of the tree (i.e. by cutting the tree at level d). Exhibit 22.6 shows an example for d = 3.

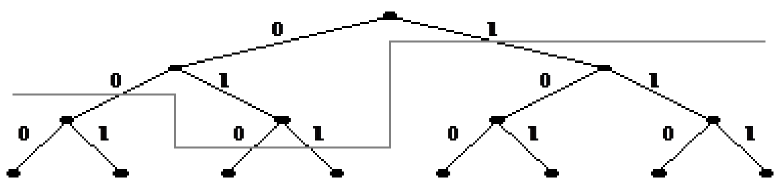

Exhibit 22.6: Address space organized as a binary radix tree.

The radix tree shown in Exhibit 22.6 (without the nodes that have been clipped) describes an active address range with addresses {00, 010, 011, 1} that are considered as bit strings or binary numbers. To each active node with address s there corresponds a bucket B that can store b records. If a new element has to be inserted into a full bucket B, then B is split: Instead of B we find two twin buckets B_0 and B_1 which have a one bit longer address than B, and the elements stored in B are distributed among B_0 and B_1 according to this bit. The new radix tree now has to point to the two data buckets B_0 and B_1 instead of B; that is, the active address range must be extended locally (by moving the broken line in Exhibit 22.6). If the block with address 00 overflows, two new twin blocks with addresses 000 and 001 will be created which are represented by the corresponding nodes in the tree. If the overflowing bucket B has depth d, then d is incremented by 1 and the radix tree grows by one level.

In extendible hashing the clipped radix tree is represented by a directory that is implemented by an array. Let d be the maximum number of bits that are used in one of the bit strings for forming an address; in the example above, d = 3. Then the directory consists of 2^d entries. Each entry in this directory corresponds to an address and points to a physical data bucket which contains all elements that have been assigned this address by the hash function h. The directory for the radix tree in Exhibit 22.6 looks as shown in Exhibit 22.7.

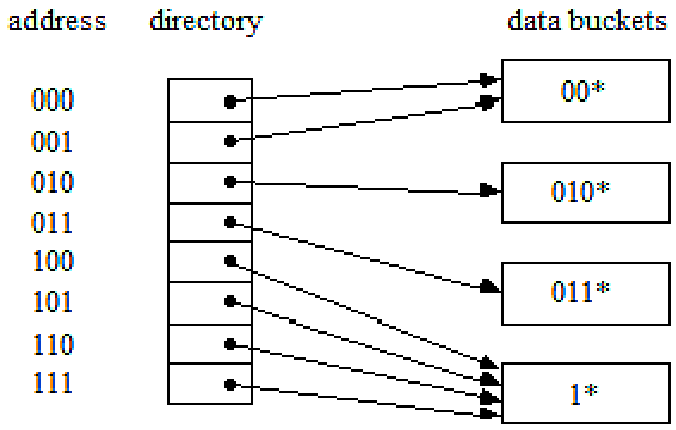

Exhibit 22.7: The active address range of the tree in Exhibit 22.6 implemented as an array.

The bucket with address 010 corresponds to a node on level 3 of the radix tree, and there is only one entry in the directory corresponding to this bucket. If this bucket overflows, the directory and data buckets are reorganized as shown in Exhibit 22.8. Two twin buckets that jointly contain fewer than b elements are merged into a single bucket. This keeps the average bucket occupancy at a high 70 per cent even in the presence of deletions, as probabilistic analysis predicts and simulation results confirm. Bucket merging may lead to halving the directory. A formerly large file that shrinks to a much smaller size will have its directory shrink in proportion. Thus extendible hashing, unlike conventional hashing, suffers no permanent performance degradation under deletions.

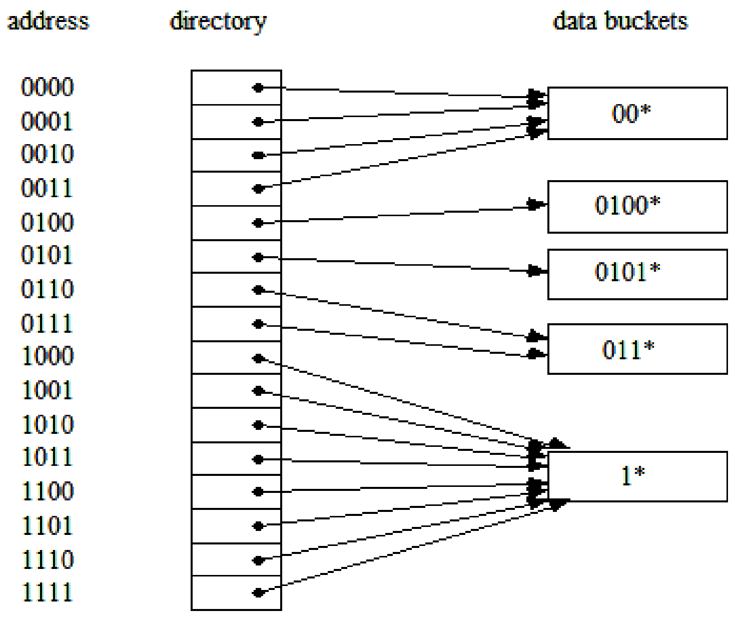

Exhibit 22.8: An overflowing bucket may trigger doubling of the directory.

A virtual radix tree: order-preserving extendible hashing

Hashing, in the usual sense of the word, destroys structure and thus buys uniformity at the cost of order. Extendible hashing, on the other hand, is practical without randomization and thus needs not accept its inevitable consequence, the destruction of order. A uniform distribution of elements is not nearly as important: Nonuniformity causes the directory to be deeper and thus larger than it would be for a uniform distribution, but it affects neither access time nor bucket occupancy. And the directory is only a small space overhead on top of the space required to store the data: It typically contains only one or a few pointers, say a dozen bytes, per data bucket of, say 1k bytes; it adds perhaps a few percent to the total space requirement of the table, so its growth is not critical. Thus extendible hashing remains feasible when the identity is used as the address computation function h, in which case data is accessible and can be processed sequentially in the order ≤ defined on the domain X.

When h preserves order, the word *hashing* seems out of place. If the directory resides in central memory and the data buckets on disk, what we are implementing is a virtual memory organized in the form of a radix tree of unbounded size. In contrast to conventional virtual memory, whose address space grows only at one end, this address space can grow anywhere: It is a virtual radix tree.

As an example, consider the domain X of character strings up to length 32, say, and assume that elements to be stored are sampled according to the distribution of the first letter in English words. We obtain an approximate distribution by counting pages in a dictionary (Exhibit 22.9). Encode the blank as 00000, 'a' as 00001, up to 'z' as 11011, so that 'aah', for example, has the code 00001 00001 01000 00000 ... (29 quintuples of zeros pad 'aah' to32letters). This address computation function h is almost an identity: It maps $\{' ', 'a', \dots , 'z'\}^{32}$ one-to-one into $\{0, 1\}^{160}$. Such an order-preserving address computation function supports many useful types of operations: for example, range queries such as "list in alphabetic order all the words stored from 'unix' to 'xinu' ".

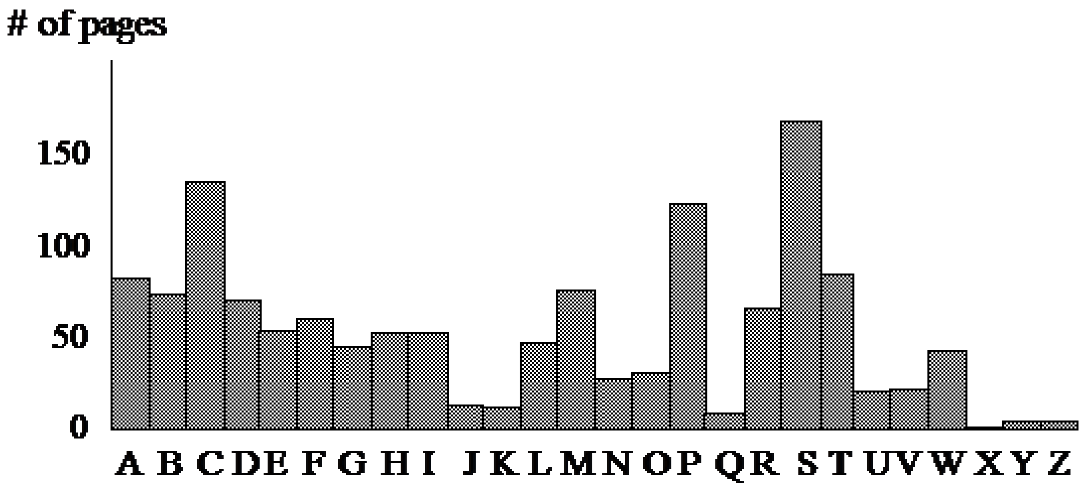

Exhibit 22.9: Relative frequency of words beginning with a given letter in Webster's dictionary.

If there is one page of words starting with X for 160 pages of words starting with S, this suggests that if our active address space is partitioned into equally sized intervals, some intervals may be populated 160 times more densely than others. This translates into a directory that may be 160 times larger than necessary for a uniform distribution, or, since directories grow as powers of 2, may be 128 or 256 times larger. This sounds like a lot but may well be bearable, as the following estimates show.

Assume that we store 10^5 records on disk, with an average occupancy of 100 records per bucket, requiring about 1000 buckets. A uniform distribution generates a directory with one entry per bucket, for a total of 1k entries, say 2k or 4k bytes. The nonuniform distribution above requires the same number of buckets, about 1,000, but generates a directory of 256k entries. If a pointer requires 2 to 4 bytes, this amounts to 0.5 to 1 Mbyte. This is less of a memory requirement than many applications require on today's personal computers. If the application warrants it (e.g. for an on-line reservation system) 1 Mbyte of memory is a small price to pay.

Thus we see that for large data sets, extendible hashing approximates the ideal characteristics of the special case we discussed in this chapter's section on “the special case of small key domains”. All it takes is a disk and a central memory of a size that is standard today but was practically infeasible a decade ago, impossible two decades ago, and unthought of three decades ago.

Exercises and programming projects

1. Design a perfect hash table for the elements 1, 10, 14, 20, 25, and 26.
2. The six names AL, FL, GA, NC, SC and VA must be distinguished from all other ordered pairs of uppercase letters. To solve this problem, these names are stored in the array T such that they can easily be found by means of a hash function h.

```
type addr = 0 .. 7;
pair = record c1, c2: 'A' .. 'Z' end;
var T: array [addr] of pair;
```

(a) Write a

```
function h (name: pair): adr;
```

which maps the six names onto different addresses in the range 'adr'.

(b) Write a

```
procedure initTable;
```

which initializes the entries of the hash table T.

(c) Write a

```
 function member (name: pair): boolean;
```

which returns for any pair of uppercase letters whether it is stored in T.

3. Consider the hash function h(x) = x mod 9 for a table having nine entries. Collisions in this hash table are resolved by coalesced chaining. Demonstrate the insertion of the elements 14, 19, 10, 6, 11, 42, 21, 8, and 1.
4. Consider inserting the keys 14, 1, 19, 10, 6, 11, 42, 21, 8, and 17 into a hash table of length m = 13 using open addressing with the hash function h(x) = x mod m. Show the result of inserting these elements using
   (a) Linear probing.
   (b) Double hashing with the second hash function g(x) = 1 + x mod (m+1).
5. Implement a dictionary supporting the operations 'insert', 'delete', and 'member' as a hash table with double hashing.
6. Implement a dictionary supporting the operations 'insert', 'delete', 'member', 'succ', and 'pred' by order-preserving extendible hashing.