

Q: *What is the most important data structure technique in your research in Yahoo?*

A: *Hashing, hashing, and hashing*

— Udi Manber

Chief Scientist at Yahoo.com

Responding to a question (SODA 2001)

“Random number generation is too important to be left to chance.”

— Robert R. Coveyou (1915–1996)

Lecture XI HASHING

Hashing is a practical technique to implement a dictionary. Its space usage is linear $\mathcal{O}(n)$ which is optimal. This is fundamentally an application of the array data structure. Like the humble array data structure, hashing is relatively simple and easy to implement. It can be highly efficient when correctly implemented. Every practitioner ought have some hashing knowledge under his or her belt.

Traditional analysis of hashing requires probabilistic assumptions in order to prove that its search time is also optimal, $\Theta(1)$. In modern setting, such probabilistic assumptions are reduced or completely removed in various ways. We look at variants and extensions of the basic hashing framework, including universal hashing, perfect hashing, and extendible hashing.

Hash is one of the oldest and most widely used data structures in computer science. The first paper¹ on hashing was by Dumeyin in 1956. Peterson [8] is another early paper. The survey of Robert Morris (1968) mark the first time that the term “hashing” appeared in publication; this paper also introduce methods beyond linear probing. Knuth [6] surveys the early history and the basic techniques in hashing. The modern approach to hashing began with the introduction of universal hashing by Carter and Wegman [1] in 1977.

§1. Elements of Hashing

¶1. Recall that a dictionary (¶III.7) is an abstract data type that stores a set of items under three basic operations: `lookUp`, `insert` and `delete`. Each item is a pair $(Key, Data)$. We will assume that the items have distinct keys.

- `insert(Item)`: returns a pointer to the location of the inserted item. Return `nil` if insertion fails.
- `lookUp(Key)`: returns a pointer to the location of an item with this key. If no such item exists, return `nil`.

¹Arnold I. Dumeyin, *Computers and Automation*, 5(12), Dec 1956.

these operations do not depend on an ordering on keys!

- **delete**(*Pointer*): removes item referenced by the pointer. This *Pointer* may be obtained from a prior **lookUp**.

The only property of keys we rely on is whether two keys are equal or not. In particular, we do not want to sort keys (use decisions based on whether one key is larger or smaller than another). There are two important special cases of dictionaries: if a dictionary supports insertions and lookups, but not deletions, we call it a **semi-dynamic dictionary**. If it supports only lookups, but not insertions or deletions, it is called a **static dictionary**. For instance, conventional books such as lexicons, encyclopedias, and phone directories, are static dictionaries for ordinary users.

¶2. **Example:** An everyday illustration of hashing is your (non-electronic) personal address book. Each item is a pair of the form (*name, address&data*). Let us allocate 26 pages, one for each letter of the alphabet. We store each item in the page allocated to the first letter of its name component. E.g., (*Yap, 111PrivetDrive*) will be stored in page 25 for the letter 'Y'. To lookup a given name, we just do a linear search of the page allocated to the first letter of the name. Deletion is done by marking an item as deleted. If a page allocated to a letter is filled up, additional entries may be placed in an overflow area. To describe this address book in the hashing framework, we say that each name x is “hashed” to its first letter which is denoted by $h(x)$. So h is a “hash function”. Of course, this simple hash function is not a good one because some pages are likely to be under-populated while others are over-populated.

¶3. **Example:** Let us consider a concrete case of storing and looking up the following set of 25 words:

```
boolean break case char class const continue do double else
for float if int import long new private public return
static switch this void while
```

Let K be the above set of words. The reader will recognize K as a subset² of the key words in the **Java** language. A **Java** compiler will need to recognize these key words, and so this example is almost realistic. It illustrates a static dictionary problem. Let us store K into an array $T[0..28]$. Assume we have a method to convert each key word $x \in K$ into a hash value, which is an integer $h(x)$ between 0 and 28. Thus h is a function from K to $\mathbb{Z}_{29} = \{0, 1, \dots, 28\}$. The idea is that we want to store x in the entry $T[h(x)]$. For example, suppose $h(x)$ simply returns the “sort code” of first letter of x (the sort code of **a** is 1, **b** is 2, **z** is 26, etc. The problem is that two keywords may have the same hash value: for instance, the hash values of **do** and **double** are both 4 and we cannot store both these key words in $T[4]$. This is called a conflict. We must resolve the conflict somehow (or change the hash function). One solution is to simply find the next slot after $T[4]$ that is available and storing it there. How does this decision affect the lookup method?

¶4. **Three simple solutions to the Dictionary Problem.** A good way to understand hashing is to first consider three straightforward methods of implementing a dictionary:

²The full set has 50 keywords (circa 2010). The longest keyword is **synchronized** with 12 characters.

- (a) As a linked list
- (b) As an array of size u
- (c) As a binary search tree

Using linked lists, we use $\Theta(n)$ space to store the n keys in K , but the time to lookup a key is $\Theta(n)$ in the worst or expected case. The space is optimal but the time is considered too slow, even for moderate n . Assuming the keys come from an universe U of size u , we can use a table (i.e., array) of size u . Then we simply store the data associated with key k in the k th entry of the table. The space is $\Theta(u)$ and the time for each dictionary operation is $\mathcal{O}(1)$. This time is optimal but the space usage is suboptimal (especially if $u \gg n$). Finally, if we use binary search trees, we can achieve $\mathcal{O}(n)$ space with $\mathcal{O}(\log n)$ time for all operations. Of course this solution must exploit an ordering of keys.

In the description of each method, we acted as if we were only storing keys in our dictionary. In applications, we are typically storing items, namely, key-data pairs. It is assumed that the data part is stored along side the key part. Modifying the above methods to account for the data part is routine. So, to focus on the underlying algorithms, we prefer to ignore the data part. Henceforth we continue this expedient of pretending that we are only storing “naked keys” without associated data.

¶5. The Hashing Framework. The hashing approach to dictionaries can be regarded as a modification of the simple array solution (method (b) above). Since array indexing is considered extremely fast, we want to use a “simulated index” into the array for lookup, insertion or deletion. This simulated index of a key is just the hash value of the key. *The goal is to implement dictionaries in which both time and space are both optimal: $\mathcal{O}(1)$ time and $\mathcal{O}(n)$ space.* Traditionally, time is $\mathcal{O}(1)$ in an expected sense. But we shall demonstrate an important case where time can even be worst case $\mathcal{O}(1)$.

The following notations will be used throughout this chapter.

Let U be the **universe**, and all keys are elements of U . U is sometimes called **key space**. Our goal is to maintain a set $K \subseteq U$ of keys of size $n = |K|$. Also, let $u = |U|$. It is important to realize that U is fixed while K is a dynamic set whose membership can change over time.

(H1) *The first premise of hashing is*

$$n = |K| \ll |U| = u. \quad (1)$$

For example, let $U = \{0, \dots, 9\}^9$ represent the set of all possible social security numbers in the USA. If a personnel database in a company uses social security numbers as keys then the number n of actual keys is much less than $u = 10^9$. Thus the first premise of hashing is satisfied. On the other hand, the premise may fail if the database is used by the US Internal Revenue Service (IRS) to represent all tax payers.

The basic hashing scheme begins with an array $T[0..m-1]$ of size m . Call T the (primary) **hash table**. Although this looks like the original solution with array of size u (method (b))

above), our first premise of hashing (H1) precludes m to be anywhere close to u . Indeed, we typically aim for optimal space, i.e., $m = O(n)$. We stress that there is no prescribed relationship between m and n . In particular, both $m \geq n$ and $m < n$ could be useful, depending on the particular hashing technique used. Each entry in this table is called a **slot** (or bucket). The key of an item is used to compute an index into the hash table. So another key element of hashing is the use of a **hash function**

$$h : U \rightarrow \mathbb{Z}_m$$

*Elements of
hashing: U, K, T, h*

*Parameters of
hashing: u, n, m*

from U to array indices. Recall that $\mathbb{Z}_m = \{0, 1, \dots, m-1\}$. Observe that the domain of h is U and not K , even in the static case. We say a key k is **hashed** to the **hash value** $h(k)$. To search for an item x with key k , we begin our search by examining the entry $T[h(k)]$. Following method (b) above, we could try to store x in $T[h(k)]$; but many hashing variations uses $T[h(k)]$ only as an indirection to the eventual location of x .

(H2) The second premise of hashing is that the hash function $h(k)$ can be evaluated quickly. In complexity analysis, we assume evaluation takes $\mathcal{O}(1)$ time. (2)

Two keys $k, k' \in U$ **collide** if $k \neq k'$ but $h(k) = h(k')$. If no pair in K collide, we could of course simply store $k \in K$ in slot $T[h(k)]$. Under assumption (H1), collisions are unavoidable except in the static case where K is fixed (see below). But in general we need to deploy some **collision resolution scheme**. Different collision resolution schemes give rise to different flavors of hashing. Sometimes, collisions are called **conflicts** and we use them interchangeably.

Consider a hash function $h : U \rightarrow \mathbb{Z}_m$ and $K \subseteq U$ a set of keys. We say h is **perfect for** K (or, K -perfect) if

$$(\forall j \in \mathbb{Z}_m) |h^{-1}(j) \cap K| \leq 1,$$

i.e., there are no collisions for keys in K . Of course, K -perfect functions are possible only if $|K| \leq m$. If we do not enforce any relation between $|K|$ and m , then the best we can hope for is that h distributes the set K evenly among the slots. That is, for all $i, j \in \mathbb{Z}_m$, we want the size of the sets $h^{-1}(i) \cap K$ and $h^{-1}(j) \cap K$ to be approximately equal: say h is **K -equidistributed** if they differ by at most one: for all $i, j \in \mathbb{Z}_m$,

$$|(|h^{-1}(i) \cap K| - |h^{-1}(j) \cap K|)| \leq 1. \quad (3)$$

Here are two simple equidistributed functions from \mathbb{Z}_v to \mathbb{Z}_m where $v \geq m$:

- $g_m(x) := x \bmod m$
- $g_{m,v}(x) := \lfloor mx/v \rfloor$.

For instance, if $v = 5$ and $m = 2$, then $g_2[0, 1, 2, 3, 4] = [0, 1, 0, 1, 0]$ and $g_{2,5}[0, 1, 2, 3, 4] = [0, 0, 0, 1, 1]$. Clearly, both these functions are equidistributed for $K = \mathbb{Z}_v$.

To summarize: in hashing, the fundamental decision of the algorithm designer is to choose a hash function $h : U \rightarrow \mathbb{Z}_m$. Here, U is given in advance but m is a design decision that is based on other parameters such as the maximum number of items that will be in the dictionary at any given moment. The second major decision is the choice of a collision resolution strategy.

¶6. **Practical Construction of Hash Functions.** A common response to the construction of hash functions is³ to “do something really complicated and mysterious”. E.g., $h(x) = (\lfloor \sqrt{x} \rfloor^3 - 5x^2 + 17)3 \bmod m$. Unfortunately, such schemes inevitably fail to perform as well as two simple and effective methods. Following Knuth [6], we call them the division and multiplication methods, respectively.

(A) Division method: The simplest is to treat a key k as an integer, and to define

$$h(k) = k \bmod m.$$

So choosing a hash function here amounts to selecting m . There is an obvious pitfall to avoid when choosing m : assuming k a d -ary integer, then it is a bad idea for m to be a power of d because if $m = d^\ell$ then $h(k)$ is simply the low order ℓ digits of k . This is not considered good as we would like h to depend on all the digits of k . For example, if k is a sequence of ASCII characters then k can be viewed as a d -ary integer where $d = 128$. Since d is a power of 2 here, it is also a bad idea for m to be a power of 2. Usually, choosing m to be a prime number is a good idea. If we have some target value for m (say, $m \sim 2^{16} = 65536$), then we may choose m to be a prime close to this target (e.g., $m = 65521$ or $m = 65537$).

Puzzle. Why does the choice of base (modulus m) matter? If k is viewed abstractly as an integer, it has no base. And if $m = d^\ell$, and k is viewed in some other base different from d , then the problem seems to go away. But we haven't done anything! The answer lies in the extra-logical properties arising from how keys are represented and manipulated in practice. To avoid the base issues, we choose m to be prime. Alternatively, is the universe U of hashing really structure-free? The fact that we avoid hash functions modulo powers of 10 is a hint that in practice, U have informal structures. For instance, if U is a set of strings over some alphabet of size d then choosing a hash table of size some power of d is a bad idea. Why? Presumably keys in this universe might be biased by the d -ary structure of keys. For instance, if keys are variable names in a user program in which groups of names tend to share common suffixes, then the mod- m hashing function (when m is a power of d) may hash every key in a group into the same slot.

(B) Multiplication method. Let $0 < \alpha$ be an irrational number. Then define

$$h(k) = \lfloor ((k \cdot \alpha) \bmod 1) \cdot m \rfloor = \lfloor \{k \cdot \alpha\} \cdot m \rfloor. \quad (4)$$

where $\{x\} = x - \lfloor x \rfloor = (x \bmod 1)$ denotes the fractional part of x . Note that in this formula, substituting $\alpha - k$ for α (for any integer k) does not affect the hash function. A good choice for α turns out to be $\alpha = \phi$, the golden ratio. Numerically,

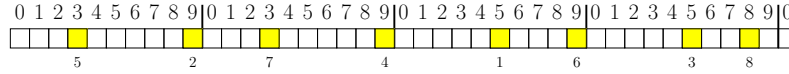
$$\phi = (1 + \sqrt{5})/2 = 1.61803\ 39887\ 49894\ 84820\ \dots$$

Remark that since $\phi > 1$, we might as well use $\alpha = \phi - 1 = 0.61803\dots$ in our calculations. With this choice, and for $m = 41$, we have $h(1) = \lfloor 25.339 \rfloor = 25$, $h(2) = \lfloor 9.678 \rfloor = 9$, $h(3) = \lfloor 35.018 \rfloor = 35$, etc. The next few values of $h(k)$ are shown in Table 1 and visualized in accompanying figure.

These numbers begin to reveal their secrets when we visualize their distribution. The choice $\alpha = \phi$ has an interesting theoretical basis, related to a remarkable theorem of Vera Turán Sós [6, p. 511] which we quote:

³Note that the same impulse arise when asked to design random number generators (§VIII.3).

k	1	2	3	4	5	6	7	8
$\{k\phi\}$.618	.236	.854	.472	.090	.708	.326	.944
$h(k)$	25	9	35	19	3	29	13	38

Table 1: Multiplication method using $\alpha = \phi$ and $m = 41$ Figure 1: Multiplicative Method: first 8 slots modulo $m = 41$

Theorem 1 (Three Distance Theorem) Let α be an irrational number. Consider the $n+1$ subsegments formed by placing the n numbers

$$\{\alpha\}, \{2\alpha\}, \dots, \{n\alpha\} \quad (5)$$

in the unit interval $[0, 1]$. Then there are at most three different lengths among these $n+1$ subsegments. Furthermore, the next point $\{(n+1)\alpha\}$ lies in one of the largest subsegment.

The proof of this theorem uses continued fraction. It is evident that if $\{\alpha\}$ is very close to 0 or 1, then the ratio of the lengths of the largest to the smallest subsegments will be large. Hence it is a good idea to choose α so that $\{\alpha\}$ is closer to $1/2$ than to 0 or 1. It turns out that the choice $\alpha = \phi = 1.61803\dots$ gives the most evenly distributed subsegment lengths.

Knuth [6, p. 509] proposes to implement (4) as follows. Suppose we are using machine arithmetic of a computer. Most modern machines uses computer words with w bits where $w = 32$ (or 64 or 128, etc). If we are designing the hash function, we have freedom to choose m , the size of the hash table. To exploit machine arithmetic, let us choose m so that $m = 2^\ell$ for some $1 < \ell \leq w$. We may choose α so that it satisfies $0 < \alpha < 1$. This determines an integer $0 < A < 2^w$ such $A/2^w$ is the largest w -bit binary fraction that is less than α . In other words, $A/2^w < \alpha < (A+1)/2^w$.

For instance, when $\alpha = \phi - 1$ (ϕ is the Golden Ratio) and $w = 32$ then $A = 2,654,435,769$.

Thus we have $\alpha = (A + \varepsilon)2^{-w}$ for some $0 < \varepsilon < 1$. Then $\{k\alpha\} = (A + \varepsilon)k2^{-w} - n$ for some $n \in \mathbb{N}$. Indeed, if $k < 2^w$, then $Ak \pmod{2^w}$ is just the lower w -bits of Ak .

Finally, $h(k) = \lfloor m\{k\alpha\} \rfloor = \lfloor (A + \varepsilon)mk2^{-w} \rfloor - mn$. Assuming $mk \leq 2^w$, it follows that $h(k) = \lfloor m\{k\alpha\} \rfloor = (A + \varepsilon)mk2^{-w} - mn$. Since we chose $m = 2^\ell$, it follows that $h(k) = 2^{\ell-w}k(A + \varepsilon)$. The proof of the following is left as an Exercise:

CLAIM: $h(k)$ is equal to $2^{k-w}Ak$.

In hardware, division is several times slower than multiplication. So we expect the division method to be somewhat slower than the multiplication method.

¶7. Remarks. A very common hashing situation is where U is a variable length string (we do not like to place any *a priori* bound on the length of the string. Assuming each character is byte-size, we may take $U = \mathbb{Z}_{256}^*$ (an infinite key space). The exercises give a practical way to

generate hash keys for this situation. In general, we can view each character as the coefficients of a polynomial $P(X)$ and we can evaluate this polynomial at some $X = a$ to give a hash code.

EXERCISES

Exercise 1.1: Please continue filling in the entries in Table 1. What is the first $k > 8$ when you get a collision? \diamond

Exercise 1.2: Suppose you want to construct a table $T[0..m-1]$ of minimum size m such that the multiplication method (with $\alpha = \phi$) will not have any collision for the first 100 entries. Experimentally determine this m . \diamond

Exercise 1.3: Consider the choice of m in the division method for hashing functions.

- (a) Why is it a bad idea to use an even number m in the division method?
- (b) Suppose keys are decimal numbers (base 10) and m is divisible by 3. What can you say about $k \bmod m$ and $k' \bmod m$ where the keys k, k' are different by a permutation of some (decimal) digits. \diamond

Exercise 1.4: (a) Compute the sequence $\{\alpha\}, \{2\alpha\}, \dots, \{n\alpha\}$ for $n = 10$ and $\alpha = \phi$ (= the golden ratio $(1 + \sqrt{5})/2 = 1.618\dots$). You may compute to just 4 decimal positions using any means you like.

(b) Let

$$\ell_0 > \ell_1 > \ell_2 > \dots$$

be the new lengths of subsegments, in order of their appearance as we insert the points $\{n\phi\}$ (for $n = 0, 1, 2, \dots$) into the unit interval. For instance, $\ell_0 = 1, \ell_1 = 0.61803, \ell_2 = 0.38197$. Compute ℓ_i for $i = 0, \dots, 10$. HINT: You have to insert over 50 points to get 10 distinct lengths, so you may want to consider writing a program to do this.

(c) Using the multiplication method with $\alpha = \phi$, please insert the following set of 16 keys into a table of size $m = 10$. Treat the keys as integers by treating the letters A, B, ..., Z as 1, 2, ..., 26, with the rightmost position having a value of 1, the next position with value 26, the third with value $26^2 = 676$, etc. Thus AND represents the integer $(1 \times 26^2) + (14 \times 26) + (4 \times 1) = 1044$. This is sometimes called the **26-adic** notation. To resolve collision, use separate chaining.

AND, ARE, AS, AT, BE, BOY, BUT, BY, FOR, HAD,
HER, HIS, HIM, IN, IS, IT

We just want you to display the results of your final hashing data structure.

(d) Use the division method on the same set of keys as (c), but with $m = 17$. \diamond

Exercise 1.5: This question assumes knowledge of Java. Consider the following definition of a generic hash table interface:

```
public interface HashTable <T> {
    public void insert (T x);
    public void remove (T x);
    public boolean contains (T x);
} //class
```

Please criticize this design. ◇

Exercise 1.6: Let K be the following set of 40 keys

A, ABOUT, AN, AND, ARE, AS, AT, BE, BOY, BUT,
BY, FOR, FROM, HAD, HAVE, HE, HER, HIS, HIM, I,
IN, IS, IT, NOT, OF, ON, OR, SHE, THAT, THE,
THEY, THIS, TO, WAS, WHAT, WHERE, WHICH, WHY, WITH, YOU

Experimentally find some simple hash functions to hash K into $T[0..m-1]$, where m is chosen between 50 and 60. Your goal is to minimize the maximum size of a bucket (a bucket is the set of keys that are hashed into one slot). (You need not be exhaustive – but report on what you tried before picking your best choice.)

- (a) Use a division method.
- (b) Use the multiplication method with $\alpha = \phi$.
- (c) Invent some other hashing rule not covered by the multiplication or division methods. ◇

Exercise 1.7: (Pearson [7]) A common hashing situation is the following: given a fixed alphabet $V = \mathbb{Z}_2^n$, we want to hash from $U = V^*$ to V . In practice, we may regard $U = \cup_{i=0}^s V^i$ for some large value of s . Typically, $n = 8$ (so V is a byte-size alphabet). Let $T : V \rightarrow V$ be stored as an array. Then we have a hash function h_T computed by the following:

HASH(w):
 Input: $w = w_1w_2 \cdots w_n \in \Sigma^*$.
 Output: hash value in $h(w) \in \Sigma$.
 1. $v \leftarrow 0$.
 2. for $i \leftarrow 1$ to n do
 3. $v \leftarrow T[v \oplus w_i]$.
 4. Return(v).

In line 3, $v \oplus w_i$ is viewed as a bitwise exclusive-or operation.

- (a) Show that if $d(w, w') = 1$ then $h(w) \neq h(w')$. Here, $d(w, w')$ is the Hamming distance (the number of symbols in w, w' that differ).
- (b) Use fact (a) to give a probe sequence $h(w, i)$ (where $i = 1, 2, \dots, N$) such that $(h(w, 1), h(w, 2), \dots, h(w, N))$ will cycle through all values of Σ .
- (c) Suppose $T[i] = i$ for all i . What does this hash function compute?
- (d) Suppose T is a random permutation of V . Show that h_T is not universal. HINT: consider the case $n = 1$ and $s = 3$. There are two choices for T . Find $x \neq y$ such that $\Pr\{h_T(x) = h_T(y)\} > 1/2$. ◇

Exercise 1.8: Here is an alternative and common solution in the hash function for the previous question.

HASH(w):
 Input: $w = w_1w_2 \cdots w_n \in \Sigma^*$.
 Output: hash value in $h(w) \in \Sigma$.
 $v \leftarrow 0$.
 for $i \leftarrow 1$ to n do
 $v \leftarrow (v + w_i) \bmod N$.
 Return(v).

Discuss the relative merits of the two methods (such as the efficiency of evaluating the hash function). \diamond

END EXERCISES

§2. Collision Resolution

We have outlined the hashing framework. This includes fundamental assumptions (H1 and H2) as basis for evaluating any concrete hashing scheme. Any such scheme must choose a primary hash function $h : U \rightarrow \mathbb{Z}_m$ to map keys into slots in a hash table $T[0..m-1]$. Collisions are inevitable because of (H1). Because of this, we do not simply store a key $k \in U$ in its slot $T[h(k)]$, but must treat $T[h(k)]$ as an entry point into some auxiliary search structure in which collisions are resolved. The two basic methods of resolving collisions are called **chaining** and **open addressing**.⁴

§8. Chaining Schemes. There are two variants of chaining. The simplest variant is called **separate chaining**. Here each table slot $T[i]$ is used as the header of a linked list of items. The linked list is called a chain or bucket. An inserted item with key k will be put at the head of the chain of $T[h(k)]$. Note that this scheme assumes some dynamic memory management (perhaps provided by the operating system), so that nodes in the linked list can be allocated and freed. The associated algorithms in this case are the obvious ones from list processing.

See Figure 2(a) for an example of separate chaining. The keys are inserted into the table of size 8 in the following order: ABE, BEV, ART, EARL, CATE. The hashing function $h(x)$ simply takes the first letter of each name and maps A to 1, B to 2, etc.

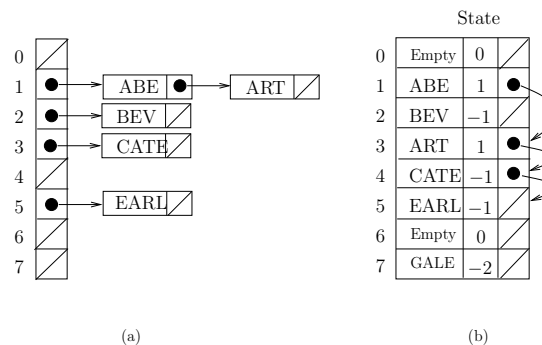


Figure 2: Chaining: (a) separate (b) coalesced

A more sophisticated variant is called **coalesced chaining** (see [6, p.513]). Here each slot $T[i]$ is potentially the node of some chain, and all nodes are allocated from the hash table T . In this way, we avoid the dynamic memory management required in separate chaining. Figure 2(b) illustrates a concrete scheme for coalesced chaining: each slot $T[i]$ has three fields with these meaning:

⁴Note that chaining is sometimes called “open hashing”, and open addressing is sometimes called **closed hashing**. The latter is unfortunate because having applying the qualifiers “open” and “closed” for the same concept is apt to confuse.

1. $T[i].\text{key}$ which stores a key (element of U).
2. $T[i].\text{next}$ which stores an element of \mathbb{Z}_m .
3. $T[i].\text{state}$ stores a value in $\{-2, -1, 0, 1, 2\}$ where $\text{state} = 0$ indicates the ORIGINAL state, $|\text{state}| = 1$ indicates OCCUPIED, $|\text{state}| = 2$ indicates DELETED. Initially, $\text{state} = 0$ but once a slot has been used, it never reverts to this state again. Moreover, $\text{state} < 0$ marks the slot as the END_OF_CHAIN, while $\text{state} > 0$ marks the MIDDLE_OF_CHAIN.

Based on the above interpretation, we deduce that $T[i]$ is a node in a chain iff $T[i].\text{state} \in \{-1, 1\}$. Moreover, if $T[i].\text{state} = 1$, then the next node in the chain is $T[T[i].\text{next}]$. Nodes are in one of three basic states: ORIGINAL, OCCUPIED and DELETED. At first blush, we might feel that only two basic states are needed – OCCUPIED and UNOCCUPIED. We invite the reader to see why we do need three states.

The reader should study the table in Figure 2(b) closely. The dictionary currently holds 5 keys: ABE, ART, BEV, CATE, EARL (as in the separate chaining data structure Figure 2(a)). The A-chain is (ABE, ART, CATE), not (ABE, ART, CATE, EARL). Similarly, the C-chain is (ART, CATE) and not (ART, CATE, EARL). Note the “coalescing” of the A-chain with the C-chain. You can also see some historical information: the key GALE had been inserted and then deleted. We also know that there is at least one earlier deletion (what is the initial letter of this deleted key?).

We also maintain a global variable n which is the number of keys currently in the hash table. Initially, $n = 0$ and $T[i].\text{next}$ is EMPTY for all i .

¶9. **To lookup a key** k , we first check $T[h(k)].\text{key} = k$. In general, suppose we have just checked $T[i].\text{key} = k$ for some index i . If this check is positive, we have found k and return i with success. If not, and $T[i].\text{next} = -1$ (END_OF_LIST), we return a failure value. Otherwise, we let $i = T[i].\text{next}$ and continue the search.

¶10. **To insert a key** k , we first check to see if the n number of items in the table has reached the maximum value m . If so, we return a failure. Otherwise, we perform a lookup on k as before. If k is found, we also return a failure. If not, we must end with a slot $T[i]$ where $T[i].\text{next} = -1$. In this case, we continue searching from i for the first j that does not store any keys (i.e., $T[j].\text{next}$ is either EMPTY or DELETED). This is done sequentially: $j = i + 1, i + 2, \dots$ (where the index arithmetic is modulo m). We are bound to find such a j . Then we set $T[i].\text{next} = j$, $T[j].\text{next} = -1$, $T[j].\text{key} = k$ and increment n . We may return with success.

¶11. **What about deletion?** We look for the slot i such that $T[i].\text{key} = k$. If found, we set $T[i].\text{key} = \text{DELETED}$. Otherwise deletion failed. Note the importance of distinguishing DELETED entries from EMPTY ones. When an empty slot is first used, it becomes “occupied”. It remains occupied until DELETED. Deleted slots can become occupied again, but they never become EMPTY. Another remark is that this method is called coalesced chaining for a good reason: chains in the separate chaining method can be combined into one chain using this scheme.

¶12. **Correctness and Coalesced List Graphs.** To understand the coalesced hashing algorithms, it is useful to look more closely at the underlying graph structures. They are just digraphs in which every node has outdegree at most 1; we may call them **coalesced list graphs**. Nodes with outdegree 0 are called **sinks**. We can also have cycles in such a graph. See Figure 3 for such a graph. The **components** of a coalesced list are just the set of nodes in the connected components in the corresponding undirected graph. There are two kinds of components: those with a unique sink and those with a unique cycle. Attached to each sink or cycle is a collection of trees.

Can coalesced hashing lead to cycles? How does coalescing occur?

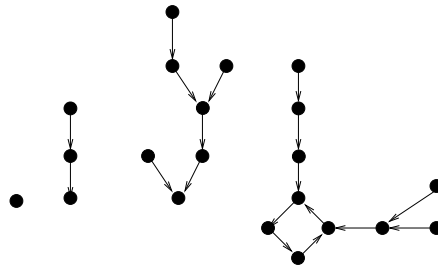


Figure 3: Coalesced List Graphs

¶13. **Open Addressing Schemes.** Like coalesced chaining, open addressing schemes store all keys in the table T itself. However, we no longer explicitly store pointers (the **next** field in coalesced chaining). Instead, for key k , we need to generate an infinite sequence of hash table addresses:

$$h(k, 0), h(k, 1), h(k, 2), \dots \quad (6)$$

This is called the **probe sequence** for k , and it specifies that after the i th unsuccessful probe, we next search in slot $h(k, i)$. In practice, the sequence (6) is cyclic: for some $1 \leq m' \leq m$, $h(k, i) = h(k, i + m')$ for all i . Ideally, we want $m' = m$ and the sequence $(h(k, 0), h(k, 1), \dots, h(k, m - 1))$ to be a permutation of \mathbb{Z}_m . This ensures that we will find an empty slot if any exists. In open addressing, as in coalesced chaining, we need to classify slots as **EMPTY**, **OCCUPIED** or **DELETED**.

There are three basic methods for producing a probe sequence:

Linear Probing This is the simplest:

$$h(k, i) = h_1(k) + i \pmod{m}$$

where h_1 is the usual hash function. One advantage (besides simplicity) is that this probe sequence will surely find an empty slot if there is one. A maximally contiguous sequence of occupied slots is called a **cluster**. A big cluster will be bad for insertions since it means we may have to traverse its length before we can insert a new key. Assuming a uniform probability of hashing to any slot, the probability of hitting a particular cluster is proportional to its length. Worse, insertion grows the length of a cluster – it grows by at least one but may grow by more when two adjacent clusters are joined. Thus, larger clusters have a higher probability of growing. Similarly, a maximal sequence of deleted and occupied slots forms a cluster for lookups. This phenomenon is known as **primary clustering**; it is similar to the tendency to see a cluster of nearly empty buses, arriving in quick succession after a long wait at the bus stop (see Exercise).

Quadratic Probing Here, the i th probe involves the slot

$$h(k, i) = h_1(k) + ai^2 + bi \pmod{m}$$

for some integer constants a, b . For reference, let “simple quadratic probing” refer to the case where $a = 1, b = 0$. There is a simple efficient method to compute successive probes: note that the difference $\Delta(i) := h(k, i+1) - h(k, i) = a((i+1)^2 - i^2) + b((i+1) - i) = a(2i+1) + b$. Moreover, $\Delta(i+1) - \Delta(i) = 2a$. Suppose we maintain the pair of variables D, V . These are initialized to the values b and $h(k)$. In each iteration, we update these variables as follows:

$$D \leftarrow D + 2a; \quad V \leftarrow V + D.$$

The the $V = h(k, i)$ after the i th iteration.

Using quadratic probing, we avoid primary clustering but there is a possibility of missing available slots in our probe sequence unless we take special care in our design of the probe sequence.

¶14. Example. Let us show how quadratic probing can miss an empty slot. Let the table size be $m = 3$, and hash function $h(x) = x \bmod 3$. Suppose the table contains $x = 0$ and $x = 1$. If we insert $x = 3$, then $h(x) = 0$. Then quadratic probing will look at $i^2 = 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, \dots$. Then $(i^2 \bmod 3) = 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, \dots$. Let us prove that $(i \bmod 3)$ is NEVER equal to 2: if $i \bmod 3 = 0$ then $i^2 \bmod 3 = 0$. if $i \bmod 3 = 1$ then $i^2 \bmod 3 = 1$. if $i \bmod 3 = 2$ then $i^2 \bmod 3 = 1$.

But it is interesting to note that this situation can be controlled to some extent: suppose the table size m is prime. CLAIM: if the load factor $\alpha = n/m$ is at most $1/2$, then quadratic probing will always find an empty slot. Thus, as long as the table is less than half-full, we are guaranteed to find an empty slot using this scheme.

Double Hashing Here, we use another auxiliary (ordinary) hash function $h_2(k)$.

$$h(k, i) = h_1(k) + i \cdot h_2(k) \pmod{m}.$$

To ensure that the probe sequence will visit every slot, it is sufficient to ensure that $h_2(k)$ is relatively prime to m . For example, this is true if m is prime and $h_2(k)$ is never a multiple of m . Other variants of double hashing can be imagined.

Note that both quadratic and double hashing are generalizations of linear probing.

EXERCISES

Exercise 2.1: In the traditional (paper) address book, what method is used to resolve collision? \diamond

Exercise 2.2: Recall our scheme for coalesced chaining as represented by Figure 2(b).

(a) We claim that the state -2 is unnecessary. What does it take to implement this? Be careful!

(b) Describe a sequence of insertions and deletions that produces the table shown in Figure 2(b).

(c) How can we combine the two fields $T[i].\text{state}$ and $T[i].\text{next}$ into one? \diamond

Exercise 2.3: In the separate chaining method, we have a choice about how to view the slot $T[i]$. Assume that each node in the chain has the form $(item, next)$ where $next$ is a pointer to the next node.

- (i) The slot $T[i]$ can simply be the first node in the chain (and hence stores an item).
- (ii) An alternative is for $T[i]$ to only store a pointer to the first node in the chain. Discuss the pros and cons of the two choices. Assume that an item requires k words of storage and a pointer requires ℓ word of storage. Your discussion may make use of the parameters k, ℓ and the load factor α . \diamond

Exercise 2.4: In coalesced hashing, you may be unable to insert a new key even when the table is not full. Illustrate this situation by giving a sequence of insertions and deletions into an initially empty small hash table (with only 3 slots). HINT: Use keys like Ann, Bob, Bill, Carol, etc. For simplicity, let the hash function look only at the first letter, $h(\text{Ann}) = 0$, $h(\text{Bob}) = h(\text{Bill}) = 1$, $h(\text{Carol}) = 2$, etc. \diamond

Exercise 2.5: T/F (Justify in either case)

- (a) In coalesced chaining, deleted slots can only be reoccupied by values with a fixed hash value.
- (b) Searching a key in coalesced chaining is never slower than the corresponding search in linear hashing (assume $h(x, i) = h(x) + i$ for linear hashing probe sequence)
- (c) In coalesced chaining, we may be unable to insert a new key even though the current number of keys is less than m (= number of slots). \diamond

Exercise 2.6: In quadratic hashing, we can avoid multiplications when computing successive addresses in the probe sequence. Show how to do this, *i.e.*, from $h(k, i)$, show how to derive $h(k, i + 1)$ by additions alone. \diamond

Exercise 2.7: Show that in double hashing, if $h_2(k)$ is relative prime to m , then all slots will eventually be probed. \diamond

Exercise 2.8: Buses start out at the beginning of a day by being evenly spaced out, say distance L apart. Let us assume that the bus route is a loop and the distance between bus i and bus $i + 1$ is $g_i \geq 0$ (the i th gap). So initially $g_i = L$. Each time a bus picks up passengers, it is more likely that the immediately following bus will have fewer or no passengers to pick up. The bus behind will therefore close up upon the first bus, forming a cluster. Moreover, the larger a cluster, the more likely the cluster will grow. In this way, the bus clustering phenomenon has similarities to the primary clustering phenomenon of hashing.

- (i) Do a simulation or analytical study of the evolution of the gaps g_i over time, assuming that the probability of passengers joining bus i is proportional to g_i , and this contributes proportionally to the slow down of bus i (so that g_{i-1} will decrease and g_{i+1} will increase). [You need not handle the case of the g_i 's going negative.]
- (ii) Let us say that two consecutive buses belong to the same cluster if their distance is $< L/2$. The size of a cluster is the distance between the leading bus and the last bus in its cluster, and the intercluster gap is defined as before. Unlike part (i), we need not worry about a bus overtaking another bus since they belong to the same cluster. So we may interpret g_i as the i th gap, but as the gap in front of the i th bus. \diamond

END EXERCISES

§3. Simplified Analysis of Hashing

¶15. We now perform the “traditional” analysis of the complexity of hashing. Notice that `delete` is $\Theta(1)$ in these methods and so the interest is in `lookUp` and `insert`. Note that an `insert` is preceded by a `lookUp`; only if this `lookUp` is unsuccessful can we then insert the new item. The actual insertion takes $\Theta(1)$ time. Hence it suffices to analyze the `lookUp`. In our analysis, the **load factor** defined as

$$\alpha := n/m$$

will be critical. Note that α will be ≤ 1 for open addressing and coalesced chaining but it is unrestricted for separate chaining.

We make several simplifying assumptions:

- **Random Key Assumption (RKA)**: it is assumed that every key in U is equally likely to be used in a lookup or an insertion. For deletion, it is assumed that every key in the current dictionary is equally likely to be deleted.
- **Perfect Hashing Assumption (PHA)**: This says our hash function is equidistributed in the sense of equation (3). Combined with (RKA), it means each lookup key k is equally likely to hash to any of the m slots. This is the best possible behavior to be expected from our hash function. So it is important to understand what we can expect under this condition.
- **Uniform Hashing Assumption (UHA)**: this assumption is about the probe sequence (6) in open addressing. We assume that the probe sequence (6) is cyclic and generates a permutation of \mathbb{Z}_m . Moreover, a random key k in U is equally likely to generate any of the $m!$ permutations of \mathbb{Z}_m .

Theorem 2 (RKA+PHA) *Using separate chaining for collision resolution, the average time for a `lookUp` is $\mathcal{O}(1 + \alpha)$.*

Proof. In the worst case, a `lookUp` of a key k needs to traverse the entire length $L(k)$ of its chain. By (RKA), the expected cost is $\mathcal{O}(1 + \bar{L})$ where \bar{L} is the average of $L(k)$ over all $k \in U$. The assumption (PHA) implies that \bar{L} is at most $n/m = \alpha$. To see this:

$$\begin{aligned} \bar{L} &= \frac{1}{u} \sum_{k=1}^u L(k) \\ &= \frac{1}{u} \sum_{j=1}^m \left(\sum_{k \in U: h(k)=j} L(k) \right) \\ &\leq \frac{1}{u} \sum_{j=1}^m \left(1 + \frac{u}{m} \right) L_j \quad (\text{by (PHA) and rewriting } L(k) \text{ as } L_{h(k)}) \\ &= \left(\frac{1}{u} + \frac{1}{m} \right) \sum_{j=1}^m L_j \\ &= \left(\frac{n}{u} + \frac{n}{m} \right) \\ &< 2\alpha. \end{aligned}$$

Q.E.D.

In order to ensure that this average time is $\mathcal{O}(1)$, we try to keep the load factor bounded in an application.

Let us analyze the average number of probes in a `lookUp` under open hashing. Recall that in this setting, when we lookup a key k , we compute a sequence of probes into $h(k, 1), h(k, 2), \dots$ until we find the key we are looking for, or we find a slot that is unoccupied. These two cases corresponds to a **successful** and an **unsuccessful lookup**, respectively. The average time for a lookup is just the number of probes made before we determine either success or otherwise. It is also easy to see that the average number of probes in an unsuccessful lookup will serve as an upper bound for the average number probes in a successful lookup.

Theorem 3 (UHA) *Using open addressing to resolve collisions, the average number of probes for an unsuccessful `lookUp` is less than*

$$\frac{1}{1 - \alpha}.$$

Proof. Clearly the expected number of probes is

$$\bar{T} = 1 + \sum_{i=1}^{\infty} i p_i$$

where p_i is the probability of making exact i probes into occupied slots. (The term “1+” in this expression accounts for the final probe into an unoccupied slot, at which point the `lookUp` procedure terminates.) But if q_i is the probability of making at least i probes into occupied slots, then we see that

$$\bar{T} = 1 + \sum_{i=1}^{\infty} i(q_i - q_{i+1}) = 1 + \sum_{i=1}^{\infty} q_i.$$

Note that $q_1 = n/m = \alpha < 1$. The assumption (UHA) implies that $q_2 = \frac{n(n-1)}{m(m-1)} < \alpha^2$. In general,

$$q_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} < \alpha^i.$$

Hence $\bar{T} < 1 + \sum_{i=1}^{\infty} \alpha^i = 1/(1 - \alpha)$.

Q.E.D.

Note that $\bar{T} \rightarrow \infty$ as $\alpha \rightarrow 1$. In order to achieve $\bar{T} = \mathcal{O}(1)$, we need to ensure that α is bounded away from 1, say $\alpha < 1 - \varepsilon$ for some constant $\varepsilon > 0$. For instance $\varepsilon = 1/2$ ensures $\bar{T} < 2$. Since all keys are stored in the table T , we often say that open addressing schemes uses no auxiliary storage (in contrast to separate chaining). Nevertheless, if α is bounded away from 1, some of the slots in T are really auxiliary storage.

EXERCISES

Exercise 3.1: Show that the average time to perform a successful lookup under the chaining scheme is $\Theta(1 + \alpha)$. \diamond

END EXERCISES

§4. Universal Hash Sets

¶16. The classical analysis of hashing depends on the random key assumption (RKA) and perfect hashing assumption (PHA). To get around this, a fundamentally new hashing idea was proposed by Carter and Wegman [1] in 1977. A really handy notation in this setting is the following: for any two sets U and V , let

$$[U \rightarrow V] \quad (7)$$

denote the set of all functions from U to V . This is generally a large set, and we are interested in subsets $H \subseteq [U \rightarrow V]$. If the functions in $[U \rightarrow V]$ are called “hash functions”, then H is called a “hash set”. We call H a **universal hash set** if for all $x, y \in U$, $x \neq y$,

$$|\{h \in H : h(x) = h(y)\}| \leq \frac{|H|}{m}. \quad (8)$$

We intend to use H by “randomly” picking an element from H and using it as our hashing function in our hashing scheme. Of course, we still need to use some collision resolution method such as chaining or open addressing methods.

So H is the sample space Ω

We employ the useful Kronecker “ δ -notation” from [1]. For $h \in [U \rightarrow \mathbb{Z}_m]$ and $x, y \in U$, define

$$\delta_h(x, y) := \begin{cases} 1 & \text{if } x \neq y \text{ and } h(x) = h(y) \\ 0 & \text{else.} \end{cases} \quad (9)$$

Thus $\delta_h(x, y)$ is the indicator variable for the x, y conflict event. We can replace any of h, x, y in this notation by sets: if $H \subseteq [U \rightarrow \mathbb{Z}_m]$ and $X, Y \subseteq U$ then

$$\delta_H(X, Y) = \sum_{h \in H} \sum_{x \in X} \sum_{y \in Y} \delta_h(x, y).$$

Variations such as $\delta_H(x, Y)$ or $\delta_h(X, Y)$ have the obvious meaning. So H is universal means $\delta_H(x, y) \leq |H|/m$ for all $x, y \in U$.

¶17. **Motivation.** In the following we let \mathbf{h} denote a uniformly random function in H . This means that for all $h \in H$, $\Pr\{\mathbf{h} = h\} = 1/|H|$. Let us first see why universality is a natural definition. It is easy to see that

$$\Pr\{\mathbf{h}(x) = \mathbf{h}(y)\} = \frac{|\{h \in H : h(x) = h(y)\}|}{|H|}.$$

This makes no assumptions about H . But if $x \neq y$, H is universal if and only if the last expression is $\leq 1/m$. This shows:

Lemma 4 H being universal is equivalent to

$$\Pr\{\mathbf{h}(x) = \mathbf{h}(y)\} \leq \frac{1}{m} \quad (10)$$

whenever $x \neq y$.

Note that $\Pr\{\mathbf{h}(x) = \mathbf{h}(y)\}$ is just the expected number of collisions involving x, y , $\mathbb{E}[\delta_{\mathbf{h}}(x, y)]$. Our lemma says that this expectation is at most $1/m$, which is as good as you can get with m slots. This is the assumption of traditional hashing theory (RKA+PHA) (see §3). But this is now achieved by construction rather than by assumption. The random key assumption (RKA) says that we are interested in analyzing \mathbf{k} , a random key in U , i.e., $\Pr\{\mathbf{k} = k\} = 1/u$ for any $k \in U$. Combined with the perfect hashing assumption (PHA),

$$\Pr\{h(\mathbf{k}) = i\} = 1/m \quad (11)$$

for any $i = 0, \dots, m-1$. So we have replaced the randomness assumption about keys in equation (11) by a randomness about hashing functions in equation (10). The latter assumption is better because in hashing applications, the algorithm designer choose the hash function, and preferably, imposes no condition on the set of keys to be inserted or searched. This is what universal hashing achieves.

The following theorem shows that universal hash sets gives us the “expected” behavior:

Theorem 5 Let $H \subseteq [U \rightarrow \mathbb{Z}_m]$ be a universal hash set and \mathbf{h} be a uniformly random function in H . For any subset $K \subseteq U$ of n keys, and for any $x \in K$, the expected number of collisions of \mathbf{h} involving x is $< n/m = \alpha$.

Proof. Recall the conflict indicator variable $\delta_{\mathbf{h}}(x, y)$ in (9). We have $\mathbb{E}[\delta_{\mathbf{h}}(x, y)] = \Pr\{\delta_{\mathbf{h}}(x, y) = 1\} \leq 1/m$. The expected number of collisions involving $x \in K$ is given by

$$\begin{aligned} \mathbb{E}[\delta_{\mathbf{h}}(x, K)] &= \mathbb{E}\left[\sum_{y \in K, y \neq x} \delta_{\mathbf{h}}(x, y)\right] \\ &= \sum_{y \in K, y \neq x} \mathbb{E}[\delta_{\mathbf{h}}(x, y)] \\ &= \frac{n-1}{m} < \alpha. \end{aligned}$$

Q.E.D.

¶18. **Generalization of Universality.** One direction to generalize universality is to replace (10) by

$$\Pr\{\mathbf{h}(x) = \mathbf{h}(y)\} \leq \varepsilon \quad (12)$$

for any fixed $\varepsilon > 0$. Such a hash set is called **almost ε -universal** by Stinson. Here, we generalize in a different direction. If $h : U \rightarrow V$ and $x_1, \dots, x_t \in U$ then we use the notation

$$h(x_1, \dots, x_t) = (y_1, \dots, y_t)$$

to mean $h(x_i) = y_i$ for all $i = 1, \dots, t$.

We say the set $H \subseteq [U \rightarrow V]$ is **strongly t -universal** ($t \in \mathbb{N}$) if for all $\{x_1, \dots, x_t\} \in \binom{U}{t}$, and all $y_1, \dots, y_t \in V$ (the y 's are not necessarily distinct),

$$|\{h \in H : h(x_1, \dots, x_t) = (y_1, \dots, y_t)\}| \leq \frac{|H|}{m^t}. \quad (13)$$

Alternatively, if \mathbf{h} is a random function in H , then (13) is equivalent to

$$\Pr\{\mathbf{h}(x_1, \dots, x_t) = (y_1, \dots, y_t)\} \leq \frac{1}{m^t}. \quad (14)$$

When $t = 2$, we simply call H a **strongly universal** hash set.

Theorem 6 If $H \subseteq [U \rightarrow \mathbb{Z}_m]$ is strongly universal, then it is universal.

Proof. Let $x \neq y \in U$ and \mathbf{h} be a random function of H .

$$\begin{aligned} \Pr\{\mathbf{h}(x) = \mathbf{h}(y)\} &= \sum_{i=0}^{m-1} \Pr\{\mathbf{h}(x) = \mathbf{h}(y) = i\} \\ &\leq \sum_{i=0}^{m-1} \frac{1}{m^2}, \quad (\text{by 2-universality, (14)}) \\ &= 1/m. \end{aligned}$$

By lemma 4, this implies the universality of H .

Q.E.D.

The converse is false: consider the set $S_U \subseteq [U \rightarrow U]$ of permutations of U . Thus $|S_U| = u!$ and for all $x \neq x'$,

$$|\{h \in S_U : h(x) = h(x')\}| = 0.$$

Thus S_U is universal. But for all $y, y' \in U$,

$$|\{h \in S_U : h(x, x') = (y, y')\}| = \begin{cases} 0 & \text{if } y \neq y', \\ (u-2)! & \text{else.} \end{cases}$$

So S_U is not 2-universal, since $(u-2)! > |S_U|/u^2$. But S_U is rather close to being 2-universal, and we will find it advantageous to modify the definition of t -universality so that S_U is considered 2-universal (Exercise).

¶19. On the Definition of Universality. Carter and Wegman show that their definition of universal hash sets is essentially the best possible.

Lemma 7 For all H , there exists $x, y \in U$ such that

$$\delta_H(x, y) > |H| \left(\frac{1}{m} - \frac{1}{u} \right).$$

Proof. First, fix $f \in H$ and let $U = \uplus_{i=0}^{m-1} U_i$ where $U_i = f^{-1}(i)$ ($i \in \mathbb{Z}_m$). Let $u_i = |U_i|$. Then

$$\delta_f(U_i, U_j) = \begin{cases} u_i(u_i - 1) & \text{if } i = j \\ 0 & \text{else.} \end{cases}$$

Hence

$$\delta_f(U, U) = \sum_i \sum_j \delta_f(U_i, U_j) = \sum_i \delta_f(U_i, U_i) = \sum_{i=0}^{m-1} u_i(u_i - 1).$$

It is easily seen that the expression $E(u_0, \dots, u_{m-1}) = \sum_{i=0}^{m-1} u_i(u_i - 1)$ is minimized when $u_i = u/m$ for all i (Exercise). Hence

$$\delta_f(U, U) \geq \sum_{i=0}^{m-1} \frac{u}{m} \left(\frac{u}{m} - 1 \right) = u^2 \left(\frac{1}{m} - \frac{1}{u} \right).$$

Hence

$$\delta_H(U, U) \geq |H| u^2 \left(\frac{1}{m} - \frac{1}{u} \right). \quad (15)$$

But

$$\delta_H(U, U) = \sum_{x \in U} \sum_{y \in U} \delta_H(x, y). \quad (16)$$

There are u^2 choices of x, y in (16). From (15), it follows that at least one of these choices will satisfy the lemma. **Q.E.D.**

This shows that, in general, the right hand side of (10) cannot be replaced by $\frac{1}{m} - \varepsilon$, for any constant $\varepsilon > 0$.

EXERCISES

Exercise 4.1: Student Quick claims out the universal hash set approach still does not overcome the problem of bad behavior for specialized sets $K \subseteq U$. That is, for any $h \in H$, we can still find a K that causes h to behave badly. Do you agree? \diamond

Exercise 4.2: Quick Search Company has implemented a dictionary data structure using universal hashing. You are a hacker who wants to make the boss of Quick Search Company (QSC) look bad, by making its dictionary operations slow. You can read all files (data, source code, etc) of the company, but you may not modify any file directly. You are also a legitimate user (employee of QSC?) who is allowed to enter new items into the dictionary. The dictionary is designed for 10,000 records (and will not accept more). It is currently half full. Discuss how you can accomplish your evil goals. What can the Quick Search Company do to avoid such kind of attacks? \diamond

Exercise 4.3: In the practical usage of a universal hash set H , suppose that after the choice of an $h_1 \in H$, the system administrator may find that the current set K of keys is causing suboptimal performance. The idea is that he should now discard h_1 and pick randomly another $h_2 \in H$ and re-insert all the keys in K . Give some guidelines about how to do this. E.g., how and when do you decide that K is causing suboptimal performance? \diamond

Exercise 4.4: Suppose we modify the definition of “ t -universality” of H to mean that for all $\{x_1, \dots, x_t\} \in \binom{U}{t}$, and all $y_1, \dots, y_t \in V$,

$$|\{h \in H : h(x_1, \dots, x_t) = (y_1, \dots, y_t)\}| \leq \frac{|H|}{m(m-1) \cdots (m-t+1)}.$$

- (a) What are the advantages of this definition?
- (b) Suppose we also modify the definition of universality of H to mean

$$|\{h \in H : h(x) = (y)\}| \leq \frac{|H|}{m-1}.$$

Show that 2-universality (in this modified sense) implies modified universality. Are there any some disadvantage in this definition? \diamond

END EXERCISES

§5. Construction of Universal Hash Sets

¶20. So far, we have only defined the concept of universal hash sets. We have not shown they exist! It is actually trivial to show their existence: just choose H to be the set $[U \rightarrow \mathbb{Z}_m]$. This H is universal (Exercise). It is unfortunately this choice is not useful: to use H , we intend to pick a random function h from H and use it as our hashing function. To “pick an h in U ” effectively, we need a “compact and effective representation” of each element of $[U \rightarrow \mathbb{Z}_m]$. If $H = [U \rightarrow \mathbb{Z}_m]$, this would require $\lg |H| = |U| \lg m$ bits. Since $u = |U|$ is very large by our fundamental assumption (H1), this is infeasible. It would also defeat an original motivation to use hashing in order to avoid $\Omega(u)$ space. Second, to use $h \in H$ as a hash function, each h must be easy to compute by assumption (H2). But not all functions in $[U \rightarrow \mathbb{Z}_m]$ have this property. Let us summarize our requirements on H :

- $|H|$ be moderate in size (typically $u^{\mathcal{O}(1)}$).
- There is an effective method to specify or name each member of H , and to randomly pick members of H .
- Each $h \in H$ must be easy to compute.

The latter two properties can be coupled together as follows: we can write $H = \{h_i : i \in I\}$ for some index set I , and there is a fixed universal program $M(\cdot, \cdot)$ such that, given an index $i \in I$, and $x \in U$, $M(i, x) = h_i(x)$. Thus $i \in I$ can be viewed as the “program” to compute h_i and M is the interpreter; the **program size** of H may be defined to be $\log |I|$. The interpreter $M(i, x)$ is efficient in that it takes $O(1)$ operations for any $(i, x) \in I \times U$. These “operations” are normally polynomial-time algebraic operations, with I and U viewed as suitable algebraic structures like finite fields, groups, etc. We next construct some universal hash sets that satisfy these requirements.

What are finite fields? They are not as unfamiliar as they sound: for instance, take $\mathbb{Z}_m = \{0, 1, \dots, m-1\}$, the set of integers modulo m . We know how to add, subtract and multiply modulo m . If m is a prime number, then we can also divide by a non-zero value (but the division algorithm is a bit less obvious). Any set F for which these four arithmetic operations are defined is called a **field**. For instance, the rational numbers \mathbb{Q} and real numbers \mathbb{R} are fields. But \mathbb{Z} is not a field because it lacks division. Of course, these sets are not finite. But \mathbb{Z}_p is a finite field for any prime p . Besides \mathbb{Z}_p , it turns out that for any prime power $q = p^n$ there is a finite field $GF(q)$ with exactly q elements. You might guess that $GF(q)$ is just \mathbb{Z}_q , with the usual modulo q arithmetic. Alternatively, you might guess that $GF(q)$ is just $GF(p)^n$, with componentwise modulo p arithmetic. Unfortunately, neither is the case. Here “GF” stands for Galois Field.

*Hey, I know one finite field!
 $\mathbb{Z}_2 = \{0, 1\}$*

¶21. **A Class of Universal Hash Sets.** Fix a finite field F with q elements. Typically, $F = \mathbb{Z}_q$ where q is prime. We are interested in hash functions in $[U \rightarrow F]$ where

$$\begin{aligned} U &= \underbrace{F \times F \times \cdots \times F}_r \\ &= F^r \end{aligned}$$

for any fixed $r \geq 1$. If $a = \langle a_0, a_1, \dots, a_r \rangle \in F^{r+1}$, we define the hash function

$$\begin{aligned} h_a &: U \rightarrow F \\ h_a(x) &= a_0 + \sum_{i=1}^r a_i x_i \end{aligned}$$

where $x = \langle x_1, \dots, x_r \rangle \in U$. Set

$$H_q^r := \{h_a : a \in F^{r+1}\} \quad (17)$$

so that $|H| = q^{r+1}$.

Hashing for ASCII Code Consider the case $F = \mathbb{Z}_2$ and $H = H_2^8$. So $h \in H$ is a hash function from $\mathbb{Z}_2^8 \rightarrow \mathbb{Z}_2$, i.e., it maps a byte to a binary value. Suppose $a = \langle 1, 0, 0, 0, 0, 1, 1, 1 \rangle$. View $x \in \mathbb{Z}_2^8$ as its ASCII code... .. incomplete...

Theorem 8 The set H_q^r is 2-universal. More precisely, if \mathbf{h} is a random function in H_q^r then

$$\Pr\{\mathbf{h}(x) = i, \mathbf{h}(y) = j\} = \frac{1}{q^2}$$

for all $x, y \in K$, $x \neq y$, and $i, j \in F$.

Proof. First write x and y as $x = \langle x_1, \dots, x_r \rangle$ and $y = \langle y_1, \dots, y_r \rangle$. Since $x \neq y$, we may, without loss of generality, assume $x_1 \neq y_1$. CLAIM: for any choice of a_2, \dots, a_r and $0 \leq i, j < m$, there exists unique a_0, a_1 such that if $a = \langle a_0, a_1, \dots, a_r \rangle$ then

$$h_a(x) = i, \quad h_a(y) = j. \quad (18)$$

To see this, note that (18) can be rewritten as

$$\begin{bmatrix} x_1 & 1 \\ y_1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} i - \sum_{\ell=2}^r a_\ell x_\ell \\ j - \sum_{\ell=2}^r a_\ell y_\ell \end{bmatrix}.$$

The right-hand side is a constant since we have fixed i, j and a_2, \dots, a_r , and x, y are given. The 2×2 matrix M on the left-hand side is non-singular because $x_1 \neq y_1$. Hence we may multiply both sides by M^{-1} , giving a unique solution for a_0, a_1 . This proves our CLAIM. There are q^{r-1} choices for a_2, \dots, a_r . It follows that there are exactly q^{r-1} functions in H such that (18) is true. Therefore,

$$\Pr\{\mathbf{h}(x) = i, \mathbf{h}(y) = j\} = \frac{q^{r-1}}{|H|} = \frac{1}{q^2}.$$

Q.E.D.

Thus H_q^r in (17) is universal.

We can increase the range of universal hash functions by forming its Cartesian products. For example, if $H \subseteq [U \rightarrow V]$ is universal, we can view H^2 as a subset of $[U \rightarrow V^2]$ where $h = (h_1, h_2) \in H^2$ can be viewed as the function $h(x) = (h_1(x), h_2(x)) \in V^2$. Clearly,

$$\Pr\{\mathbf{h}(x, y) = (i, j)\} \leq \Pr\{h_1(x, y) = (i, j)\} \Pr\{h_2(x, y) = (i, j)\} \leq m^{-4} = |V^2|^{-2}.$$

showing that H^2 is still universal.

¶22. **Example:** Consider a typical application where $U = \{0, \dots, 9\}^9$ is the set of social security numbers. We wish to construct a dictionary (=database) in which $n = 50,000$ (e.g., n is an upper bound for the number enrolled students at Universal University). Our problem is to choose an m such that $\alpha = n/m$ is some small constant, say

$$1 < \alpha < 10. \quad (19)$$

The motivation for $\alpha < 10$ is to bound the expected size of a chain which, according to theorem 5, is bounded by α . The motivation of $\alpha > 1$ is to limit the pre-allocated amount of storage (which is the table $T[0..m-1]$) to less than n . Note that U and n are given *a priori*.

Solution: We reduce this problem to the construction of a universal hash set of the form (17). Let us assume q is a prime. First of all, note that q should be somewhere between 5,000 and 50,000. We also need to choose r so that each $k \in U$ is viewed as an r -tuple $\langle k_1, \dots, k_r \rangle$. For this purpose, we divide the 9 digits in k into $r = 3$ blocks of 4, 4, 1 digits (respectively). E.g., $k = 123456789$ is viewed as the triple $\langle 1234, 5678, 9 \rangle$. Let q be the smallest prime larger than 10^4 , i.e., $q = 10007$. Hence $\alpha = 50000/10007 \approx 5$. Note that even though k_3 is never more than 9, it did not affect our application of theorem 5: the result does not depend on the choice of K ! This method can be generalized.

¶23. **Strongly t -Universal Sets.** For any $t \geq 2$, we can construct strongly t -universal hash set as follows: let $F = \mathbb{Z}_q$ and $U = F^t$ and $a \in F^{t+1}$, let $h_a : U \rightarrow F$ be defined by

$$h_a(X) = \sum_{i=0}^t a_i X^i$$

where $a = (a_0, \dots, a_t)^T$. Then if for all $x = (x_1, \dots, x_t)^T, y = (y_1, \dots, y_t)^T \in F^t$, we have that

$$h_a(x) = y$$

iff

$$V(x)a = y$$

where

$$V(x) = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^t \\ 1 & x_2 & x_2^2 & \cdots & x_2^t \\ \vdots & & & \ddots & \\ 1 & x_t & x_t^2 & \cdots & x_t^t \end{bmatrix}$$

is the Vandermonde Matrix. Assuming that x_1, \dots, x_t are t distinct values, $V(x)$ is nonsingular.

... to be continued.

¶24. **Weighted Universal Hash Sets.** Consider the following situation. Let U, V, W be three finite sets. Suppose

$$H \subseteq [U \rightarrow V]$$

is a universal hash set, and

$$g : V \rightarrow W$$

is an equidistributed hash function. This means

$$|\{x \in V : g(x) = i\}| \leq \lceil |V|/|W| \rceil.$$

For instance, let $W = \mathbb{Z}_m$ and g is the modulo m function, $g(x) = x \bmod m$. Let

$$g \circ H := \{g \circ h : h \in H\}$$

where $(g \circ h)(x) = g(h(x))$ denotes function composition. Under what condition is $g \circ H$ universal?

Before proceeding, we need a clarification: it may happen that there exists hash functions $h \neq h'$ such that $g \circ h = g \circ h'$. When this happens, we get $|g \circ H| < |H|$. In the following, we shall assume

$$|g \circ H| = |H|.$$

To allow this to hold without restriction, we must interpret $g \circ H$ as a multiset. Formally, a **multiset** is a pair (S, μ) where $\mu : S \rightarrow \mathbb{N}$ assigns a **multiplicity** $\mu(x)$ to each $x \in S$. We usually simply refer to S as the “multiset” with μ implicit. We shall generalize this further and allow $\mu(x)$ to be any non-negative real number. In this case, we call S a **weighted set**. For any set $X \subseteq S$, write $\mu(X)$ for $\sum_{x \in X} \mu(x)$. It is obvious that our concept of universality extends naturally to weighted set of functions: a weighted set $H \subseteq [U \rightarrow V]$ is **universal** if for all $x, y \in U$, $x \neq y$,

$$\mu(\{h \in H : h(x) = h(y)\}) \leq \frac{\mu(H)}{m}.$$

We use a weighted universal set H by picking a “random” function \mathbf{h} in H : this means for any $h \in H$, $\Pr\{\mathbf{h} = h\} = \mu(h)/\mu(H)$.

EXERCISES

Exercise 5.1: What does it mean for H to have “compact description and constant time evaluation”? HINT: think of the H ’s you know – why can’t they be just any arbitrary set of functions? This is a conceptual question; use your general understand of algorithms and computation to formalize this idea. \diamond

Exercise 5.2: (a) Is the set $H_0 = [U \rightarrow \mathbb{Z}_m]$ universal? 2-universal? Useful as a universal hash set?

(b) Is the set $H_U \subseteq [U \rightarrow U]$ of permutations on U universal? 2-universal? Useful as a universal hash set? \diamond

Exercise 5.3: For universal hash sets $H \subseteq [U \rightarrow \mathbb{Z}_m]$ and $K \subseteq U$ of size n , prove the following:

- (a) If $n = m$, the expected size of the largest bucket is less than $\sqrt{n} + \frac{1}{2}$.
- (b) If $n = 2m^2$, with probability $> 1/2$, every bucket receives an element. \diamond

Exercise 5.4: Consider the universal hash set $g \circ H$ above. Suppose $|F| = q$ and $m_1 = (q \bmod m)$. Give an exact expression for the cardinality of $\delta_H(x, y)$ for $x, y \in F$ in terms of m, q, m_1 . \diamond

Exercise 5.5: (Carter-Wegman) Suppose we modify the multiset H_g by omitting those functions in $h_{a,b} \in g \circ H$ where $b \neq 0$. Let \hat{H}_g be this new class. In other words, \hat{H}_g has all functions of the form $h_a(x) = g(ax)$. Show that $\delta_{\hat{H}_g}(x, y) \leq 2|\hat{H}_g|/m$. That is, the class is “universal within a constant factor of 2”. \diamond

Exercise 5.6: Suppose we define \hat{H}_q^r similarly to H_q^r , except that we fix $a_0 = 0$. Hence $|\hat{H}_q^r| = q^r$.

(a) Show that theorem 8 fails for \hat{H}_q^r .

(b) Show that \hat{H}_q^r is still universal. \diamond

Exercise 5.7: Consider the example above in which we choose to interpret a social security number as a triple $\langle k_1, k_2, k_3 \rangle$ where the 9 digits are distributed among k_1, k_2, k_3 in the proportions 4 : 4 : 1. Can I choose the proportion 3 : 3 : 3? What are the new freedoms I get with this choice? HINT: what other m 's are now available to me? How close can α get to 10? \diamond

Exercise 5.8: Generalize the above methods for construct t -universal hash sets for any $t \in \mathbb{N}$. \diamond

Exercise 5.9: Let $U = [1..t]^s$ for integers $t, s \geq 2$ and let n be given. What is a good way to construct a universal hash set H of functions from U to \mathbb{Z}_m , where m is chosen to satisfy $0.5 < \alpha = n/m < t$. NOTE: t is typically small, e.g., $t = 10, 26, 128, 256$. You may use the fact (Bertrand's postulate) that for any $n \geq 1$, there is a prime number p satisfying $n < p \leq 2n$. \diamond

Exercise 5.10: Let $H \subseteq [U \rightarrow V]$ be universal, and $g : V \rightarrow W$ be an equidistributed function. Define the multiset

$$g \circ H := \{g \circ h : h \in H\}.$$

Let $|H| = h, |U| = u, |V| = v, |W| = w$. Then $g \circ H$ is universal under either one of the following conditions:

(i) H is 2-universal and v divides w .

(ii) $v > w$ and $h \geq \frac{v^2(v-1)}{v-w}$. (For instance, if $v > w$ and $h \geq v^3$.) \diamond

END EXERCISES

§6. Optimal Static Hashing

¶25. Recall (§1) that a static dictionary is one that supports lookups, but no insertion or deletions. The question arises: for any set $K \subseteq U$, can we find hashing scheme that has worst-case $\mathcal{O}(1)$ access time and $\mathcal{O}(|K|)$ space? An elegant affirmative answer is provided by Fredman, Komlós and Szemerédi [5].

For brevity, call it the **optimal hashing problem**, since $\mathcal{O}(|K|)$ is optimal space and the worst-case $\mathcal{O}(1)$ is optimal time. The consideration of worst-case time here is to be contrasted to the expected time bounds of traditional hashing analysis (see §3). Also, the combination of small space with $\mathcal{O}(1)$ worst case time is necessary since we can otherwise obtain $\mathcal{O}(1)$ worst case time trivially, by using space $\mathcal{O}(|U|)$ and hashing each k into its own slot.

The following basic setup will be used in our analysis: assume $U = \mathbb{Z}_p$ for some prime p , and let $K \subseteq U$, $|K| = n$ be given. We want to define a hash function $h : U \rightarrow \mathbb{Z}_m$ with certain properties that are favorable to K . It is assumed that $u = |U| > m$. For any $k \in \mathbb{Z}_p$ and $x \in U$,

$$h_{k,m}(x) = ((kx \bmod p) \bmod m).$$

We write $h_k(x)$ instead of $h_{k,m}(x)$ when m is understood. We avoid $k = 0$ in the following, since $h_0(x) = 0$ for all x . For any $k \in \mathbb{Z}_p \setminus \{0\}$ and $i \in \mathbb{Z}_m$, define the i th bin to be $\{x \in K : h_k(x) = i\}$, and let its size be

$$b_k(i) := |\{x \in K : h_k(x) = i\}|.$$

635 Note that the number of pairs $\{x, y\}$ that collide in the i th bin is $\binom{b_k(i)}{2}$. We have the following
636 bound:

Lemma 9

$$\sum_{k=1}^{p-1} \sum_{i=0}^{m-1} \binom{b_k(i)}{2} < \frac{pn^2}{2m}.$$

Proof. The left-hand side counts the number of pairs

$$(k, \{x, y\}) \in \mathbb{Z}_p^+ \times \binom{K}{2}$$

637 such that $h_k(x) = h_k(y)$. Let us count this in another way: we say that $k \in \mathbb{Z}_p$ “charges” the
638 pair $\{x, y\} \in \binom{K}{2}$ if $h_k(x) = h_k(y)$. The k ’s that charge $\{x, y\}$ satisfies

$$\begin{aligned} (xk \bmod p) - (yk \bmod p) &\equiv 0 \pmod{m}, \\ (x - y)k \bmod p &\equiv 0 \pmod{m}, \\ (x - y)k \bmod p &\in S := \{m, 2m, \dots, \left\lfloor \frac{p-1}{m} \right\rfloor m\}. \end{aligned}$$

But for each element jm in the set S above, there is a unique $k (= jm(x - y)^{-1})$ such that $(x - y)k \bmod p = jm$. Hence the number of k ’s that charge $\{x, y\}$ is

$$|S| = \left\lfloor \frac{p-1}{m} \right\rfloor.$$

Thus the total number of charges, summed over all $\{x, y\} \in \binom{K}{2}$ is

$$\binom{n}{2} \left\lfloor \frac{p-1}{m} \right\rfloor < \frac{n^2 p}{2m}.$$

639

Q.E.D.

Corollary 10 (i) There exists a $k \in \mathbb{Z}_p^+$ such that

$$\sum_{i=0}^{m-1} \binom{b_k(i)}{2} < \frac{n^2}{2m}.$$

(ii) There are at least $p/2$ choices of $k \in \mathbb{Z}_p^+$ such that

$$\sum_{i=0}^{m-1} \binom{b_k(i)}{2} < \frac{n^2}{m}.$$

We have an immediate application. Choosing $m = n^2$, corollary 10(i) says that there is a k such that

$$\sum_{i=0}^{m-1} \binom{b_k(i)}{2} < 1. \quad (20)$$

This means for each $i \in \mathbb{Z}_m$, $\binom{b_k(i)}{2} = 0$ and hence $b_k(i) = 0$ or 1. This means h_k is a perfect hash function for K .

¶26. The FKS Scheme. We now describe the FKS scheme [5] to solve the optimal hashing problem. This scheme is illustrated in figure 4.

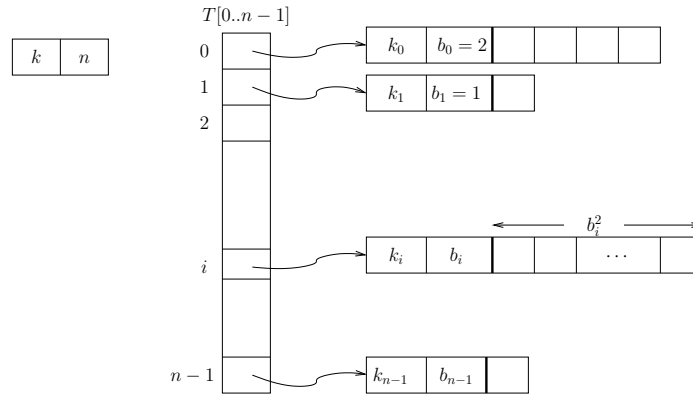


Figure 4: FKS Scheme

There are two global variables k, n and these are used to define the primary hash function,

$$\tilde{h}(x) = ((xk \bmod p) \bmod n). \quad (21)$$

There is a main hash table $T[0..n-1]$. The i th entry $T[i]$ points to a secondary hash table that has two parameters k_i, b_i and these define the secondary hash functions

$$h_{(i)}(x) = ((xk_i \bmod p) \bmod b_i^2). \quad (22)$$

We shall choose b_i to be the size of the i th bin,

$$b_i = |\{x \in K : \tilde{h}(x) = i\}|.$$

Hence, according the remark above, we could choose k_i in (22) so that (20) holds, and so $h_{(i)}$ is a perfect hash function.

How much space does the FKS scheme take? The primary table takes $n + 2$ cells (the “+2” is for storing the values n and k). The secondary tables use space

$$\sum_{i=0}^{n-1} (2 + b_i^2) = 2n + \sum_{i=0}^{n-1} b_i^2. \quad (23)$$

According to corollary 10(i), we can choose the key k in the primary hash function (21) such that

$$\sum_{i=0}^{n-1} \binom{b_i}{2} < \frac{n^2}{m} = n \quad (24)$$

($m = n$). Thus (24) implies $\sum_{i=0}^{n-1} b_i(b_i - 1) < 2n$ and hence

$$\sum_{i=0}^{n-1} b_i^2 < 2n + \sum_{i=0}^{n-1} b_i = 3n.$$

This, combined with (23), implies the secondary tables use space $5n$. The overall space usage is therefore less than

$$n + 2 + 5n = 6n + 2.$$

¶27. **Constructing a FKS Solution.** Given p and K , how do we find the keys k and k_0, \dots, k_{n-1} specified by the FKS scheme? For simplicity, assume that each arithmetic operation is constant time in the following analysis.

A straightforward way is to search through \mathbb{Z}_p to find a primary hash key k . Checking each k to see if corollary 10(i) is fulfilled takes $\mathcal{O}(n)$ time. Since there are p keys, this is $\mathcal{O}(pn)$ time. To find a suitable secondary k_i for each i takes another $\mathcal{O}(pb_i)$ time; summing over all i 's, this is $\mathcal{O}(pn)$ time. So the overall time is $\mathcal{O}(pn)$.

Since p can be very large relative to n , this solution is sometimes infeasible. If we use a bit more space (but still linear), we can use corollary 10(ii) to give a randomized method of construction (Exercise). We next present a deterministic time solution.

¶28. **Prime Sieve.** We take a short detour to consider the classical Sieve of Eratosthenes to find all primes less than some given n : We use a Boolean array $B[2..n-1]$ initialized to true.

```

For each  $i \leftarrow 2$  to  $n-1$     ◁ Outer Loop
  If  $B[i]$ 
    Output  $i$  as prime.
  (A) If  $(i^2 \leq n)$ 
    For  $j \leftarrow 2$  to  $n/i$     ◁ Inner Loop
       $B[ij] \leftarrow \text{false}$ 

```

Each inner loop takes $\mathcal{O}(1)$ time if i is non prime, and $\mathcal{O}(n/i)$ time if i is prime. Summing over all primes p , the algorithm takes $\mathcal{O}(n \left(\sum_p 1/p \right))$. The summation over the p 's is clearly at most $H_n = \mathcal{O}(\log n)$. So the complexity is $\mathcal{O}(n \log n)$. Actually, it is known in Number theory that $\sum_{p < n} 1/p = \ln \ln n + \mathcal{O}(1)$. So the cost is actually $\mathcal{O}(n \lg \lg n)$.

Notice the test in Line (A) to avoid the inner loop if $i > \sqrt{n}$. Why is this justified? In some applications, including the one to be described next, we should write the Prime Sieve as a “co-routine” which, after an initialization, can be repeatedly called to yield the next prime. Thus co-routines are routines with state information that is preserved between calls.

The solution uses a simple trick to reduce the size of the universe. We need a useful fact from number theory. Let $\vartheta(x) := \ln \left(\prod_{q \leq x} q \right)$ be the natural logarithm of the product of all primes q not exceeding x . Then a result of Rosser and Schoenfeld [9] says

$$0.31n < \vartheta(n) < 1.02n. \quad (25)$$

for all $n \geq 2$. Moreover, using the **sieve of Eratosthenes** (276–194 B.C.), we can produce a list of all the primes not exceeding n in time $O(n \lg \lg n)$ on a Real RAM (see, e.g., [10, p. 112]).

Lemma 11 *There exists a prime $q \leq 2n^2 \lg p$ that for all $x, y \in K$,*

$$x \neq y \quad \Rightarrow \quad (x \bmod q) \neq (y \bmod q). \quad (26)$$

This q can be found in $O((n^3 \lg p)/\lg(n \lg p))$ algebraic operations on elements of U .

Proof. Note that q satisfies (26) iff q does not divide $x - y$. If

$$N := \prod_{x, y} |x - y|$$

where $\{x, y\}$ range over $\binom{K}{2}$, then we are looking for a q that does not divide N . If $\vartheta(x) > N$, then there is some prime $q \leq x$ that does not divide N . It is therefore sufficient to see that

$$\begin{aligned} \vartheta(2n^2 \lg p) &> 0.62n^2 \lg p \quad (\text{by (25)}) \\ &> \binom{n}{2} \lg p \\ &> N. \end{aligned}$$

We now show that q can be found in $O(n^3 \lg p)$ operations: Assume $p > 2n^2 \lg p$, otherwise we can let q be equal to p . Use the sieve of Eratosthenes to list all the primes not exceeding $2n^2 \lg p$ in time $O(n^2 \lg p \lg \lg p)$. Discard primes q in the list that are less than n . For $q \geq n$, we check that q fulfills the conditions of (26).

This can be done in time $\mathcal{O}(n)$ as follows: first, initialize a Boolean array $V[0..q-1]$ to **false**. Then for each $k \in K$, we check if $V[k \bmod q]$ equals **false**; if so, we set $V[k \bmod q] \leftarrow \text{true}$; otherwise, we have found a conflict and we can reject this q . Eventually, we will find a suitable prime q . By the Prime Number Theorem, there are $O((n^2 \lg p)/\lg(np))$ such q 's to check. Hence the overall time is $O((n^3 \lg p)/\lg(np))$. **Q.E.D.**

Two remarks about this lemma: (1) Instead of $\vartheta(x)$, we could also use the prime number function $\pi(x)$ which count the number of primes less than x . Unfortunately, this gives a slightly weaker bound using the above argument. (2) Note that we take $O(n)$ steps to check if a q is suitable. One might imitate the proof of first part of the lemma, and checking if q divides N . This takes essentially quadratic in n since N is a product of up to $n^2 \log p$ many factors.

Theorem 12 *For any subset $K \subseteq \mathbb{Z}_p$, $n = |K|$, there is a hashing scheme to store K in $\mathcal{O}(n)$ space and with $\mathcal{O}(1)$ worst case lookup time. This scheme can be constructed deterministically in time*

$$\mathcal{O}(n^3 \lg p).$$

Proof. If $p < 2n^2 \lg p$, then we can use the FKS scheme for this problem. As noted above, the straightforward method to construct the FKS scheme takes time $\mathcal{O}(pn) = \mathcal{O}(n^3 \lg p)$, achieving our stated bound.

So assume $p \geq 2n^2 \lg p$. Use the sieve of Eratosthenes to list all the primes not exceeding $2n^2 \lg p$ in time $O(n^2 \lg p \lg \lg p)$. Discard primes q in the list that are less than n . For $q \geq n$, we check that q fulfills the conditions of the preceding lemma.

This can be done in time $\mathcal{O}(n)$ as follows: first, initialize a table $V[0..q-1]$ to 0. Then for each $k \in K$, we check if $V[k \bmod q]$ equals 0; if so, we set $V[k \bmod q] \leftarrow 1$; otherwise $V[k \bmod q] = 1$ and we have found a conflict and we can reject this q . Eventually, we will find a suitable prime q . The time taken is $\mathcal{O}(n^3 \lg p)$ since there are $\mathcal{O}(n^2 \lg p)$ such q 's.

We now construct a FKS scheme for the set of keys

$$K' = \{k \bmod q : k \in K\}$$

viewed as a subset of the universe \mathbb{Z}_q . The only difference is that, in the secondary tables, in the slot for key $k' \in K'$, we store the original value $k \in K$ corresponding to k' .

The straightforward method of constructing this scheme is $\mathcal{O}(qn)$ which is within our stated bound. To lookup a key k^* , we first compute $k' = k^* \bmod q$, and then use the FKS scheme to lookup the key k' . Searching for k' will return the key $k \in K$ such that $k \bmod q = k'$. Then k^* is in K iff if $k^* = k$. **Q.E.D.**

¶29. Bit Complexity Model. We can convert the above results into the bit complexity model. First, we have assumed $\mathcal{O}(1)$ space for storing each number in $U = \mathbb{Z}_p$. In the big complexity model, we just need to multiply each space bound by $\lg p$. As for time, each arithmetic operation that we have assumed is constant time really involves $\lg p$ bit numbers, and each uses

$$\mathcal{O}(\lg p \lg \lg p \lg \lg \lg p)$$

bit operations. Again, multiplying all our time bounds by this quantity will do the trick.

¶30. Bloom Filters Given a set $K \subseteq U$ of integers, hashing provides a data structure $D(K)$ that can answer membership queries, is $x \in K$? The minimum space for $D(K)$ if we want exact answers is $\log \binom{|U|}{|K|} = \log \binom{u}{n} = n \log u$ (check).

But suppose you are allowed $\varepsilon > 0$ probability of error, but only allowing false positives. I.e., if it is in the set, you must say YES. But if it not in the set, you might erroneously say YES. Bloom filters provide a solution with space $s = n \log(1/\varepsilon)$.

Why is this reasonable? Imagine K is so huge that you keep it in a cloud database. But you have a local Bloom filter (on your laptop) to answer $x \in K$. If yes, you will then go to the cloud to retrieve the data associated with x . Because of false positives, you may get nothing. But this wasted effort has ε probability of occurring.

HOW DOES IT WORK? Choose $k = k(\varepsilon) > 1$ (usually a small constant less than 5). Choose k random hash functions h_1, \dots, h_k from $[U \rightarrow \{1, \dots, s\}]$. Let $A[1..s]$ be a bit array, initially all 0. For each $x \in K$, we compute $h_i(x)$ for $i = 1, \dots, k$, and set $A[h_i(x)] = 1$ for each i .

To answer if $x \in K$, we check if $A[h_i(x)] = 1$ for all $i = 1, \dots, k$. If so, answer YES, else NO. Clearly, YES may be wrong, but NO is never wrong.

Compute k necessary to achieve the said performance.

EXERCISES

Exercise 6.1: Construct a FKS scheme for the following input: $p = 31$, $K = \{2, 4, 5, 15, 18, 30\}$. \diamond

Exercise 6.2: Construct a FKS scheme for the 40 common English words in §1 (Exercise 1.6). \diamond

Exercise 6.3: In many applications, the key space U comes with some specific structure. Suppose $U = \mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_r}$ where n_1, \dots, n_r are pre-specified. In a certain transaction processing application, we have $(n_1, \dots, n_r) = (2, 9, 4, 9, 5)$. Construct a FKS scheme for this application. \diamond

Exercise 6.4: When students are asked to prove a subquadratic time bound on the Prime Sieve, they produced the following answers:
 (i) $O(n^{3/2})$
 (ii) $O(n^2/\log n)$
 Please reverse-engineer to figure out their (correct) reasoning in these answers. \diamond

Exercise 6.5: Show that the expected time to construct the above hashing scheme for any given K is $\mathcal{O}(n^2)$. That is, find the values $k, k_0, \dots, k_{n-1}, b_0, \dots, b_{n-1}$ in expected $\mathcal{O}(n)$ time. \diamond

Exercise 6.6: Justify the test in Line (A) in the Prime Sieve Algorithm. \diamond

Exercise 6.7: The above $\mathcal{O}(pn)$ deterministic time algorithm for constructing the FKS scheme was only sketched. Please fill this in the details. Program this in a programming language of your choice. \diamond

Exercise 6.8: Lemma 11 shows that the prime q that satisfy (26) is bounded by $2n^2 \lg p$. What is the best upper bound on q if you used the prime number function $\pi(x)$ instead of $\vartheta(x)$? Note that $\pi(x)$ counts the number of primes less than x . \diamond

END EXERCISES

§7. Perfect Hashing

¶31. Let $h : U \rightarrow V$ and $K \subseteq U$. We said (§1) h is perfect for K if for all $i, j \in V$, we have $|b_i - b_j| \leq 1$ where $b_i = |h^{-1}(i) \cap K|$. In the literature, this definition is further restricted to the case $|K| \leq |V|$. In this case, we have $b_i = 0$ or $b_i = 1$. In this section, we assume this restriction. If h is perfect for K and $|K| = |V|$, then we say h is **minimal perfect**. A comprehensive survey of perfect hashing may be found in [2].

Following Mehlhorn, we say a set $H \subseteq [U \rightarrow V]$ is (u, v, n) -**perfect** if $|U| = u, |V| = v$ and for all $K \subseteq \binom{U}{n}$, there is a $h \in H$ that is perfect for K . Extending this notation slightly, we say H is $(u, v, n; k)$ -**perfect** if, in addition, $|H| = k$. Such a set H can be represented as $k \times u$ matrix M whose entries are elements of V . Each row of M represents a function in H . Moreover, if M' is the restriction of M to any n columns, there is a row of M' whose entries are all distinct.

Let us give a construction for such a matrix based on the theory of finite combinatorial planes. Let F_q be any finite field on q elements. Let M be a $(q+1) \times q^2$ matrix with entries in F_q . The rows of M are indexed by elements of $F \cup \{\infty\}$ and the columns of M are indexed by elements of F^2 . Let $r \in F \cup \{\infty\}$ and $(x, y) \in F^2$. The $(r, (x, y))$ -th entry is given by

$$M(r, (x, y)) = \begin{cases} xr + y & \text{if } r \neq \infty \\ x & \text{else.} \end{cases}$$

It is easy to see that for any two columns of M , there is exactly one row at which these two columns have identical entries. It easily follows:

Theorem 13 *If $q+1 > \binom{n}{2}$ then M represents a $(q^2, q, n; q+1)$ -perfect set of hash function.*

finiteplanes

We consider lower bounds on $|H|$ for perfect families.

Theorem 14 (Mehlhorn) *If H is (u, v, n) -perfect then*

- (a) $|H| \geq \binom{u}{n} \left(\frac{u}{v}\right)^2 \binom{v}{n}.$
 (b) $|H| \geq \frac{\log u}{\log v}.$

EXERCISES

Exercise 7.1: Let $m \geq n \geq 1$. What is the probability that a random function in $[\mathbb{Z}_n \rightarrow \mathbb{Z}_m]$ is perfect? Compute this probability if $m = 13, n = 10$. Or if $m = n = 10$? \diamond

Exercise 7.2: Compare the relative merits of the FKS scheme and the scheme in theorem 13 for constructing perfect hash functions. What are the respective program sizes in these two schemes? \diamond

Exercise 7.3: Let $x = (x_1, \dots, x_n)$ be a vector of real numbers. Let $f(x) = \prod_{i=1}^n x_i$ and $g(x) = \sum_{i=1}^n x_i$. We want to maximize $f(x)$ subject to $g(x) = c$ (for some constant $c > 0$) and also $x_i \geq 0$ for all i . HINT: a necessary condition according to the theory of Lagrange multipliers is that $\nabla f = \lambda \nabla g$ for some real number λ . Why is this also sufficient? \diamond

END EXERCISES

§8. Extendible Hashing

¶32. So far, all our hashing methods are predicated upon some implicit upper bound for our dictionary. The only method that can accommodate unbounded dictionary size is hashing with separate chaining, but as the average chain length increases, the effectiveness of this method also breaks down. Extendible hashing [3] is a technique to overcome this handicap of conventional hashing. It can also be an alternative to B -trees, which are extensively used in database management.

But before we consider extendible hashing, we should mention a simple method to overcome the fixed upper limit of a hashing data structure. Each time the upper limit L of a hashing structure is reached, we can simply reorganize the data structure into one with twice the limit, $2L$. This reorganization takes $O(L)$ time, and hence the amortized cost of this reorganization is $O(1)$ per original insertion. By the same token, if the number of keys is sufficiently small, we can reorganize the hash data structure into one whose limit is $L/2$. To avoid the phenomenon of trashing at the boundaries of these limits, it is not hard to introduce hysteresis behavior (Exercise).

Extendible hashing has a two-level structure comprising a **directory** and a variable set of **pages**. The directory is usually small enough to be in main memory while the pages store items and are kept in secondary memory. See figure 5 for an illustration.

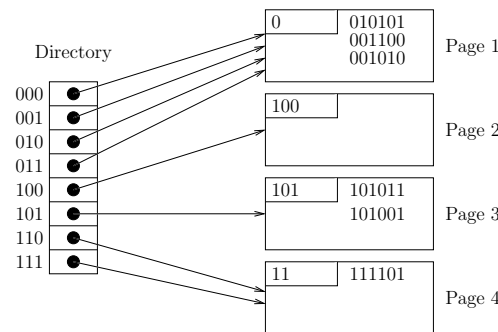


Figure 5: Extendible Hashing data structure: some hash values in the pages represent items stored under that hash value

We postulate a hash function of the form

$$h : U \rightarrow \{0, 1\}^L$$

for some $L > 1$. All pages have the same size, say, accommodating B items. Each page has its own **prefix** which is a binary string of length at most L . An item with key k will be stored in the page whose prefix p is a prefix of $h(k)$. For instance, in page 1 of figure 5, we store three items (as represented by the hash values of their keys: 010101, 001100 and 001010). The **depth** of the page is the length of its prefix. The **depth of the directory**, denoted by d , is the maximum depth of the pages. We require that the collection of page prefixes forms a prefix-free code. Recall (§IV.1, Huffman code) that a set of strings is a prefix-free code if no string in the set is a prefix of another. For instance, in figure 5, the prefix of each page is shown in the top left corner of the page; these prefixes form the prefix-free code

0, 100, 101, 11.

A directory of depth d is an array of size 2^d , where the entry $T[i]$ is a pointer to the page whose prefix is a prefix of the binary representation of i . So if a page has prefix of depth $d' \leq d$ then there will be $2^{d-d'}$ pointers pointing to it.

The actual storage method within a page is somewhat independent of extendible hashing method. For instance, any hashing scheme that uses a fixed size table but no extra storage will do (e.g., coalesced chaining or open addressing schemes). Search times for extendible hashing thus depends on the chosen method for organizing pages. It can be shown that the expected number of pages to store n items is about $n(B \ln 2)^{-1}$. This means that the expected load factor is $\ln 2 \approx 0.693$.

Knuth [6] is the basic reference on the classical topics in hashing. The article [4] considers minimal perfect hash functions for large databases.

EXERCISES

- Exercise 8.1:** (a) Show that in the worst case, the rules we have given above for increasing or decreasing the maximum size of a hashing data structure does not have $O(1)$ amortized cost for insertion and deletion.
 (b) Modify the rules to ensure amortized $O(1)$ time complexity for all dictionary operations. \diamond

END EXERCISES

References

- [1] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Computer and System Sciences*, 18:143–154, 1979.
- [2] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theor. Computer Science*, 182:1–143, 1997.
- [3] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing – a fast access method for dynamic files. *ACM Trans. on Database Systems*, 4:315–344, 1979.
- [4] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *J. ACM*, 35(1):105–121, 1992.
- [5] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31:538–544, 1984.
- [6] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Boston, 1972.
- [7] P. K. Pearson. Fast hashing of variable-length text strings. *Comm. of the ACM*, 33(6):677–680, 1990.
- [8] W. W. Peterson. Addressing for random access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957. Early major paper on hashing –perhaps the second paper on hashing? See Knuth v.3.
- [9] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math.*, 6:64–94, 1962.
- [10] C. K. Yap. *Fundamental Problems of Algorithmic Algebra*. Oxford University Press, 2000.