hashing

a dictionary

- **Dictionary:**
  - Dynamic-set structure for storing items indexed using *keys*.
  - Supports operations Insert, Search, and Delete.
  - Applications:
    - Common database-like “formations”
    - Memory-management tables in operating systems.
    - Large-scale distributed systems
    - Cryptography
- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

hash table vs array

- A hash table is a generalization of an ordinary array.
- Ordinary array: store the element whose key is k in position k of the array.
- Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called **direct addressing**.
- We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.
- Given a key k, don’t just use k as the index into the array. Instead, compute a function of k, and use that value to index into the array. We call this function a hash function.

direct address tables

- Direct-address Tables are ordinary arrays.
- Facilitate direct addressing.
  - Element whose key is k is obtained by indexing into the k^{th} position of the array.
- Applicable when we can afford to allocate an array with one position for every possible key.
  - i.e. when the universe of keys U is small.

Scenario

- Maintain a dynamic set.
- Each element has a key drawn from a universe U = {0, 1, …., m-1} where m isn't too large.
- No two elements have the same key.

Direct-address table, or array, T[0…m-1]:

- Each slot, or position, corresponds to a key in U.
- If there's an element x with key k, then T[k] contains a pointer to x.
- Otherwise, T[k] is empty, represented by NIL.

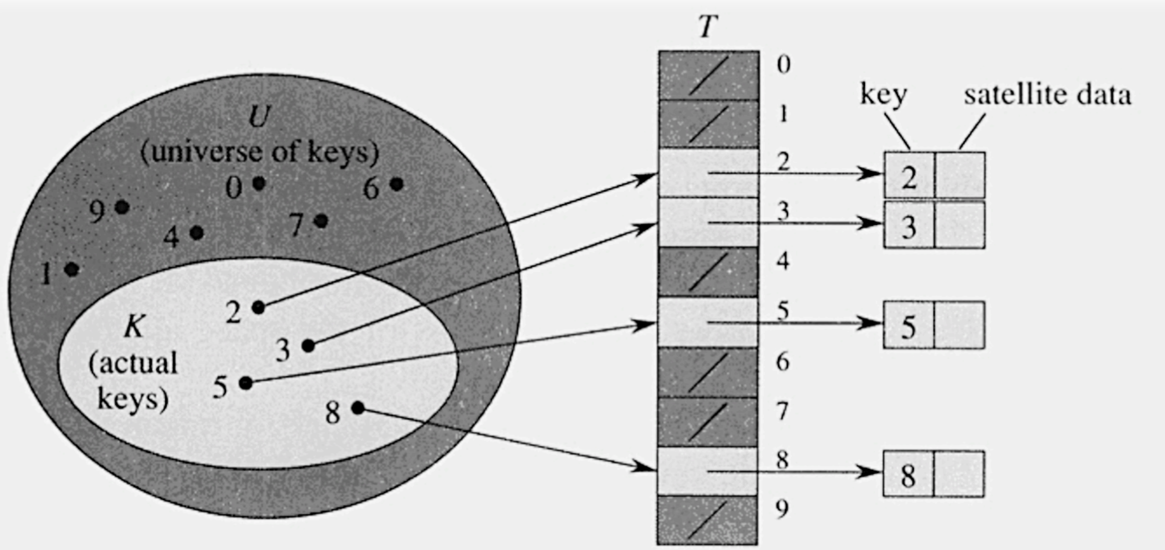


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

operations in O(1)

```
DIRECT-ADDRESS-SEARCH(T, k)
return T[k]

DIRECT-ADDRESS-INSERT(T, x)
T[key[x]] = x

DIRECT-ADDRESS-DELETE(T, x)
T[key[x]] = NIL
```

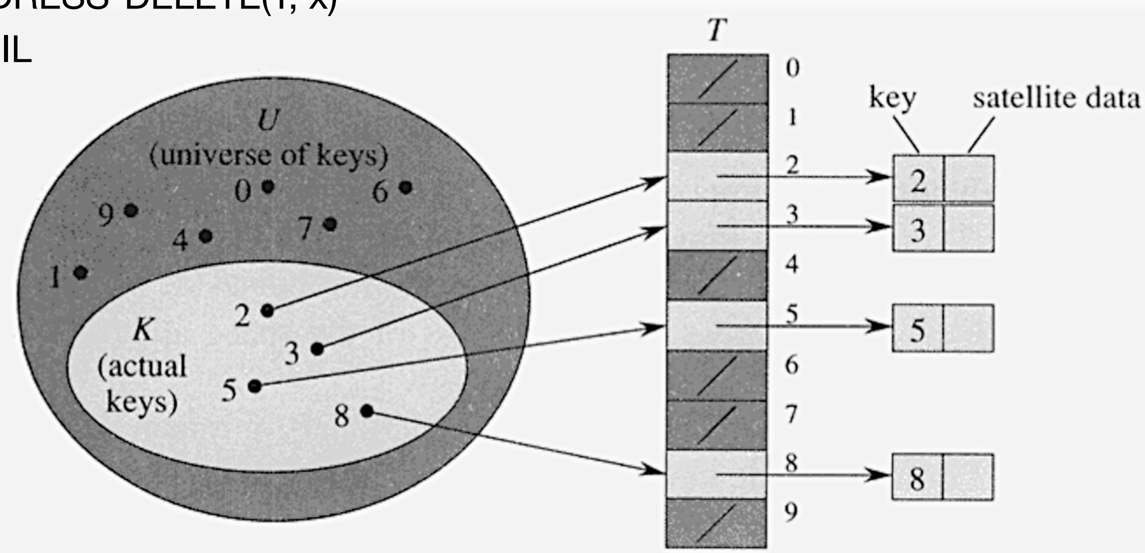


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

hash tables

Notation:

- U – Universe of all possible keys.
- K – Set of keys actually stored in the dictionary.
- $|K| = n$.

When U is very large,

- Arrays are not practical.
- $|K| << |U|$

Use a table of size proportional to $|K|$ – The hash tables.

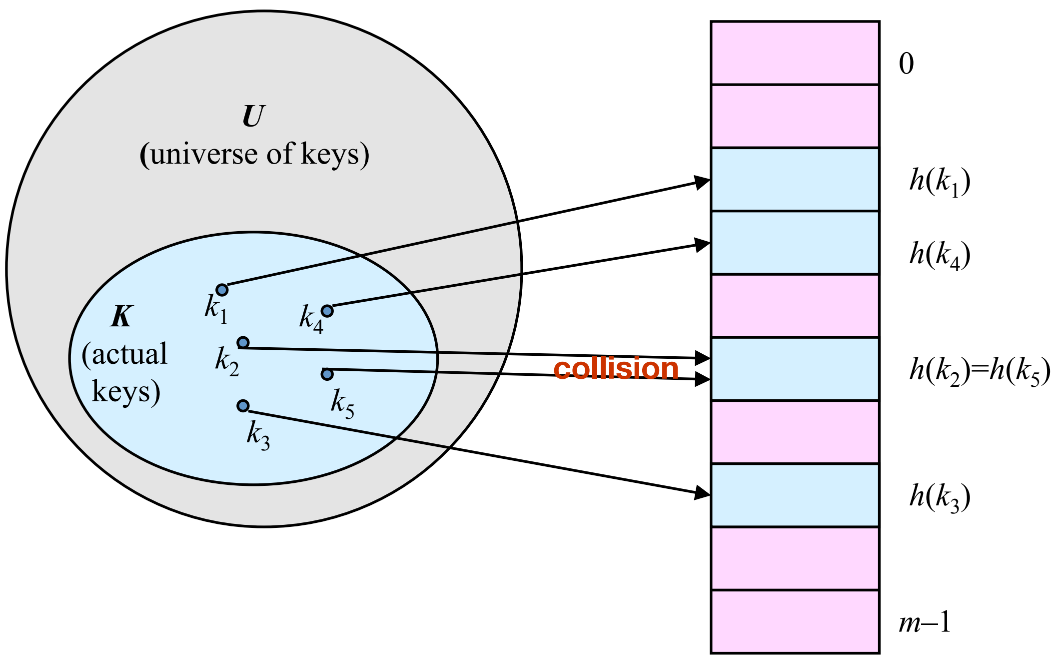
- However, we lose the direct-addressing ability.
- Define functions that map keys to slots of the hash table.

hashing

- Hash function h: Mapping from U to the slots of a hash table $T[0..m–1]$

$$h : U \rightarrow \{0,1,\ldots, m-1\}$$

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps or “hashes” to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

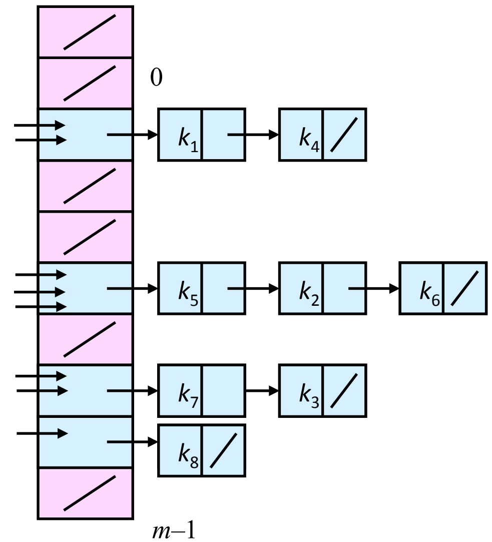


issues

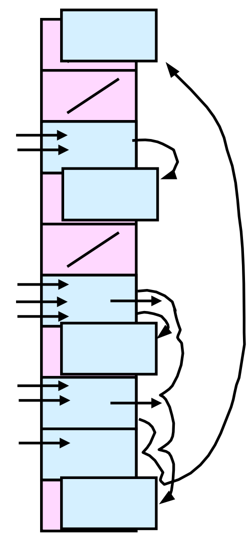
- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - Avoiding collisions is practically impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$.

resolving collisions

- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



hashing by chaining

Dictionary Operations:

- Chained-Hash-Insert (*T, x*)
  - Insert *x* at the head of list *T*[*h*(*key*[*x*])]
  - Worst-case complexity – *O*(1)
- Chained-Hash-Delete (*T, x*)
  - Delete *x* from the list *T*[*h*(*key*[*x*])].
  - Worst-case complexity: proportional to length of list with single-linked lists.
- Chained-Hash-Search (*T, k*)
  - Search an element with key *k* in list *T*[*h*(*k*)].
  - Worst-case complexity: proportional to length of list.

analysis

- Load factor $\alpha = n/m$ = average keys per slot.
  m – number of slots.
  n – number of elements stored in the hash table.

- Worst-case complexity: $\Theta(n)$ + time to compute $h(k)$.

- Average depends on how h distributes keys among m slots.

Assume

- *Simple uniform hashing*.

  Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to

- $O(1)$ time to compute $h(k)$

expected cost of unsuccessful search

Theorem:
An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- Any key not already in the table is equally likely to hash to any of the *m* slots.
- To search unsuccessfully for any key *k*, need to search to the end of the list $T[h(k)]$, whose expected length is α.
- Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

expected cost of successful search

Theorem:
A successful search takes expected time $\Theta(1+\alpha)$.

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the *n* elements in the table.
- The number of elements examined during a successful search for an element *x* is 1 more than the number of elements that appear before *x* in *x*'s list.
  - These are elements inserted <u>after</u> *x* was inserted
- Goal:
  - Find the average, over the *n* elements *x* in the table, of how many elements were inserted into *x*'s list after *x* was inserted.

Proof (cont'd):

- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.
- Define random variables $X_{ij} = \delta\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple **uniform** hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$

  $\Rightarrow E[X_{ij}] = 1/m$.
- Expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

number of elements inserted after x_i into the same slot as x_i.

Proof (cont'd):

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right)$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

expected total time for a successful search =
time to compute hash function + time to search
= ***O*(2+α/2 – α/2*n*) = *O*(1+ α)**

interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1) \Rightarrow$ Searching takes constant time on average.
- Insertion is $O(1)$ in the worst case.
- Deletion takes $O(1)$ worst-case time
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

good hash functions

- Satisfy the assumption of ***simple uniform hashing.***
  - Not possible to satisfy the assumption in practice

- Often use heuristics, based on the domain of the keys, to create a hash function that performs well

- Regularity in key distribution should not affect uniformity. Hash value should be independent of any patterns that might exist in the data:

  e.g. each key is drawn independently from U according to a probability distribution P:

$$\sum_{k:h(k)=j} P(k) = 1/m \quad \text{for } j = 0, 1, \ldots, m-1.$$

division

- Map a key k into one of the m slots by taking the remainder of k divided by m. That is,

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

- Example: $m = 31$ and $k = 78 \Rightarrow h(k) = 16$

- Advantage: Fast, since requires just one division operation.

- Disadvantage: Have to avoid certain values of m.
  - Don't pick certain values, such as $m=2^p$

- Good choice for m:
  - Primes, not too close to power of 2 (or 10) are good.

multiplication

- If $0 < A < 1$, $h(k) = \lfloor m\ (kA \bmod 1) \rfloor$
- Advantage: value of m is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.
- Disadvantage: slower than the division method.
- Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

  $h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor = \lfloor 1000 \cdot 0.018169\ldots \rfloor = 18$

$h(k) = \lfloor m\ (kA \bmod 1) \rfloor$
$0 < A < 1$
$m=2^p$

example

- Let m=8 (implies p=3: $m = 2^p$), k=21 and w=5: the word size of the machine (in bits)

Pick an s: $\mathbf{0 < s < 2^w}$ => $0 < s < 2^5$; choose s = 13
Calculate A as: $\mathbf{A = s/2^w}$ => $A = 13/2^5$ => A = 13/32
kA = 21 x 13/32 = 273/32 = 8.53125…
$kA \bmod 1$ = .53125… => $m(kA \bmod 1)$ = 8 x 0.53125… = 4.25
$\lfloor m(kA \bmod 1) \rfloor = 4$

You may as well just calculate $\boldsymbol{k \cdot s = r_1 \cdot 2^w + r_0}$
$k \cdot s = 21 \times 13 = 273 = 8 \cdot 2^5 + 17$ => ($r_1 = 8$, $r_0 = 17$)
Writing r_0 in w = 5 bits: 10001
Take p=3 most significant bits: 100 in binary = 4 in decimal
Therefore: h(k) = 4

multiplication implementation

- Choose $m = 2^p$, for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word (k takes w bits.)
- Choose an s: $0 < s < 2^w$ (s takes w bits)
- Restrict A to be of the form $s/2^w$.
- Let $k \times s = r_1 \cdot 2^w + r_0$
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($kA \bmod 1$)
- r_0 in w bits
- Use the p most important bits of r_0.

Universal Hashing

- Use a different random hash function each time
- Ensure that the random hash function is independent of the keys that are actually going to be stored.
- Ensure that the random hash function is “good” by carefully designing a class of functions to choose from.
  - Design a **universal** class of functions.

universal set of hash functions

- A finite collection of hash functions ***H*** that map a universe ***U*** of keys into the range $\{0, 1, \dots, m-1\}$ is ***"universal"*** if, for each pair of distinct keys $(k, l) \in U$,

  the number of hash functions $h \in H$ for which $h(k)=h(l)$ is no more than $|H|/m$

- The chance of a collision between two keys is the $1/m$ chance of choosing two slots randomly & independently.

- Universal hash functions give good hashing behavior.

cost of universal hashing

Theorem:
Using chaining and universal hashing on key *k*:

- If *k* is <u>not</u> in the table T, the expected length of the list that *k* hashes to is ≤ α.
- If *k* is in the table T, the expected length of the list that *k* hashes to is ≤ $1+\alpha$.

Proof:

$X_{kl} = \delta\{h(k)=h(l)\}$. $E[X_{kl}] = \Pr\{h(k)=h(l)\} \le 1/m$.

Y_k = number of keys other than *k* that hash to the same slot as *k*. Then,

$$Y_k = \sum_{l \in T \wedge l \neq k} X_{kl}, \text{ and } E[Y_k] = \mathrm{E}\left[\sum_{l \in T \wedge l \neq k} X_{kl}\right] = \sum_{l \in T \wedge l \neq k} E[X_{kl}] \le \sum_{l \in T \wedge l \neq k} \frac{1}{m}$$

$$\text{If } k \notin T, \text{exp. length of list} = E[Y_k] \le n/m = \alpha.$$

$$\text{If } k \in T, \text{exp. length of list} = E[Y_k] + 1 \le (n-1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha.$$

Open Addressing

An alternative to chaining for handling collisions.

Idea:

- Store all keys in the hash table.
- Each slot contains either a key or NIL => load factor $\alpha \leq 1$
- To ***search*** for key *k*:

  Examine slot h(k). Examining a slot is known as a **probe**.

  - If slot $h(k)$ contains key k, the search is successful.
  - If the slot contains NIL, the search is unsuccessful.

  There's a third possibility: **slot *h*(*k*) contains a key that is not *k***.

  - Compute the index of some other slot, based on k and which probe we are on.
  - Keep probing until we either find key k or we find a slot holding NIL.

probe sequence

- The sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.
- The hash function is,

$$\boldsymbol{h} : \boldsymbol{U} \times \underbrace{\{\mathbf{0, 1, \ldots,} \boldsymbol{m} \mathbf{- 1}\}}_{\textit{probe number}} \rightarrow \underbrace{\{\mathbf{0, 1, \ldots,} \boldsymbol{m} \mathbf{- 1}\}}_{\textit{slot number}}$$

- $\langle h(k,0), h(k,1),\ldots,h(k,m-1)\rangle$ should be a permutation of $\langle 0, 1,\ldots, m-1\rangle$.

example: *linear probing*

h(x) = x mod 13

h(x,i) = (h(x) + i) mod 13

Insert the keys: 18, 41, 22, 44, 59, 32, 31, 73

x = 18 => h(18) = 5 => h(18,0) = **5**

0	1	2	3	4	5	6	7	8	9	10	11	12
					18							

x = 41 => h(41) = 2 => h(41,0) = **2**

		41			18							

x = 22 => h(22) = 9 => h(22,0) = **9**

		41			18				22			

x = 44 => h(44) = 5 => h(44,0) = 5, h(44,1) = **6**

		41			18	44			22			

x = 59 => h(59) = 7 => h(59,0) = **7**

		41			18	44	59		22			

x = 32 => h(32) = 6 => h(32,0) = 6, h(32,1) = 7, h(32,2) = **8**

		41			18	44	59	32	22			

x = 31 => h(31) = 5 => h(31,0) = 5, h(31,1) = 6, h(31,2) = 7, h(31,3) = 8, h(31,4) = 9, h(31,5) = **10**

		41			18	44	59	32	22	31	73	

x = 73 => h(73) = 8 => h(73,0) = 8, h(73,1) = 9, h(73,2) = 10, h(73,3) = **11**

operation Insert

Act as though we were searching, and insert at the first NIL slot found.

<u>Hash-Search (T, k)</u>

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3.   **if** $T[j] = k$
4.   **then return** j
5. $i \leftarrow i + 1$
6. **until** $T[j] =$ NIL **or** $i = m$
7. **return** NIL

Hash-Insert(T, k)

1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3.   **if** $T[j] =$ NIL
4.   **then** $T[j] \leftarrow k$
5.   **return** j
6.   **else** $i \leftarrow i + 1$
7. **until** $i = m$
8. **error** “hash table overflow”

deletion

- Cannot just turn the slot containing the key we want to delete to contain NIL. **Why?**
- Use a special value DELETED instead of NIL when marking a slot as empty during deletion.
  - ***Search*** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - ***Insert*** should treat DELETED as though the slot were empty, so that it can be reused.

computing probe sequences

- The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1\rangle$ as its probe sequence.
- It is hard to implement true uniform hashing.
  - Approximate with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1\rangle$.
- Some techniques:
  - Use ***auxiliary hash functions***.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.
  - Can't produce all $m!$ probe sequences.

linear probing

$$h(k, i) = (h'(k)+i) \bmod m$$

key *probe number* *auxiliary hash function*

- The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
  - Hence, only *m* distinct probe sequences are possible.

- Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Hence, average search and insertion times increase.

Quadratic Probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \qquad c_1 \neq c_2$$

key *probe number* *auxiliary hash function*

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number *i.*
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

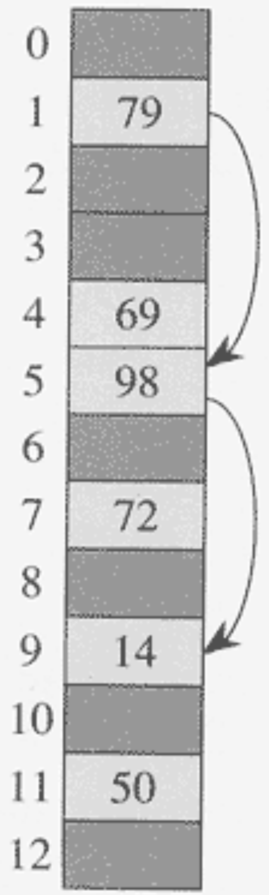
Double Hashing

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

key *probe number* *auxiliary hash functions*

- Two auxiliary hash functions.
  - h_1 gives the initial probe $T[h_1(k)]$. h_2 gives the remaining probes.
- $h_2(k)$ must be such that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
  - Choose m to be a power of 2 and have $h_2(k)$ always return an odd number, or,
  - Let m be prime, and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences.
  - One for each possible combination of $h_1(k)$ and $h_2(k)$.
  - Close to the ideal uniform hashing.

Figure from Intro to Algorithms, 3rd edition, MIT Press, p.273



$k = 14 \Rightarrow h_1(k)=1,\ h_2(k)=4$

$h(k,i) = (h_1(k) + i\ h_2(k)) \bmod m$

i=0=> h(k,i) = 1 mod 13=1 (occupied)

i=i+1=>i=1=>h(k,i)=(1+4) mod 13=5 (occupied)

i=i+1=>i=2=>h(k,i)=(1+8) mod 13=9 (it was NIL)

Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Analysis of Open-address Hashing

- Analysis is in terms of load factor α.
- **Assumptions:**
  - Assume that the table never fills completely, so $n < m$ and $\alpha < 1$.
  - Assume uniform hashing.
  - In a successful search, each key is equally likely to be searched for.

cost of unsuccessful search

Theorem:
The expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$.

Proof:

Every probe except the last is to an occupied slot.

Let a random variable X = # of probes in an unsuccessful search.

$X \geq i$ iff probes 1, 2, …, $i-1$ are made to occupied slots

Let A_i = event that there is an ith probe, to an occupied slot.

$\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\} = \Pr\{A_1\}\Pr\{A_2 \mid A_1\} \Pr\{A_3 \mid A_2 \cap A_1\} \ldots \Pr\{A_{i-1} \mid A_1 \cap \ldots \cap A_{i-2}\}$

There are n elements and m slots, therefore $\Pr\{A_1\}=n/m$.

Also $\Pr\{A_2 \mid A_1\} = (n-1)/(m-1)$, and in general $\Pr\{A_j \mid A_1 \cap A_2 \cap \ldots \cap A_{j-1}\} = (n-j+1)/(m-j+1)$

Therefore, for the i-th probe, $\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$.

And asymptotically,

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

cost of successful search

Theorem:
The expected number of probes in a successful search in an open-address hash table is at most (1/α) ln (1/(1–α)).

Proof:

- A successful search for a key *k* follows the same probe sequence as when *k* was inserted.
- If *k* was the (*i*+1)st key inserted, then α equaled *i*/*m* at that time.
- The expected number of probes made in a search for *k* is at most 1/(1–*i*/*m*) = *m*/(*m*–*i*).
- This is assuming that *k* is the (*i*+1)st key. We need to average over all *n* keys.
- Averaging over all n keys, the average number of probes is given by,

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\leq\frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-a}$$

$$\int_{m}^{n+1} f(x)\,dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m-1}^{n} f(x)\,dx$$ *(Appendix A.12, p.1154)*

Perfect Hashing

- Start with hashing similar to chaining
- Instead of making a linked list of keys hashing to slot j, we use a secondary hash table S_j with an associated hash function h_j. Careful choice of h_j to guarantee zero collisions
- To avoid collisions in secondary hashing, even with “ideal” hashing functions, S_j must be large enough (related to the number of keys in slot j): $m_j = n_j^2$ (probability of collision < ½)
- This means we use use two sets of universal hash functions, for the two levels of hashing

Figure from Intro to Algorithms, 3rd edition, MIT Press, p.278

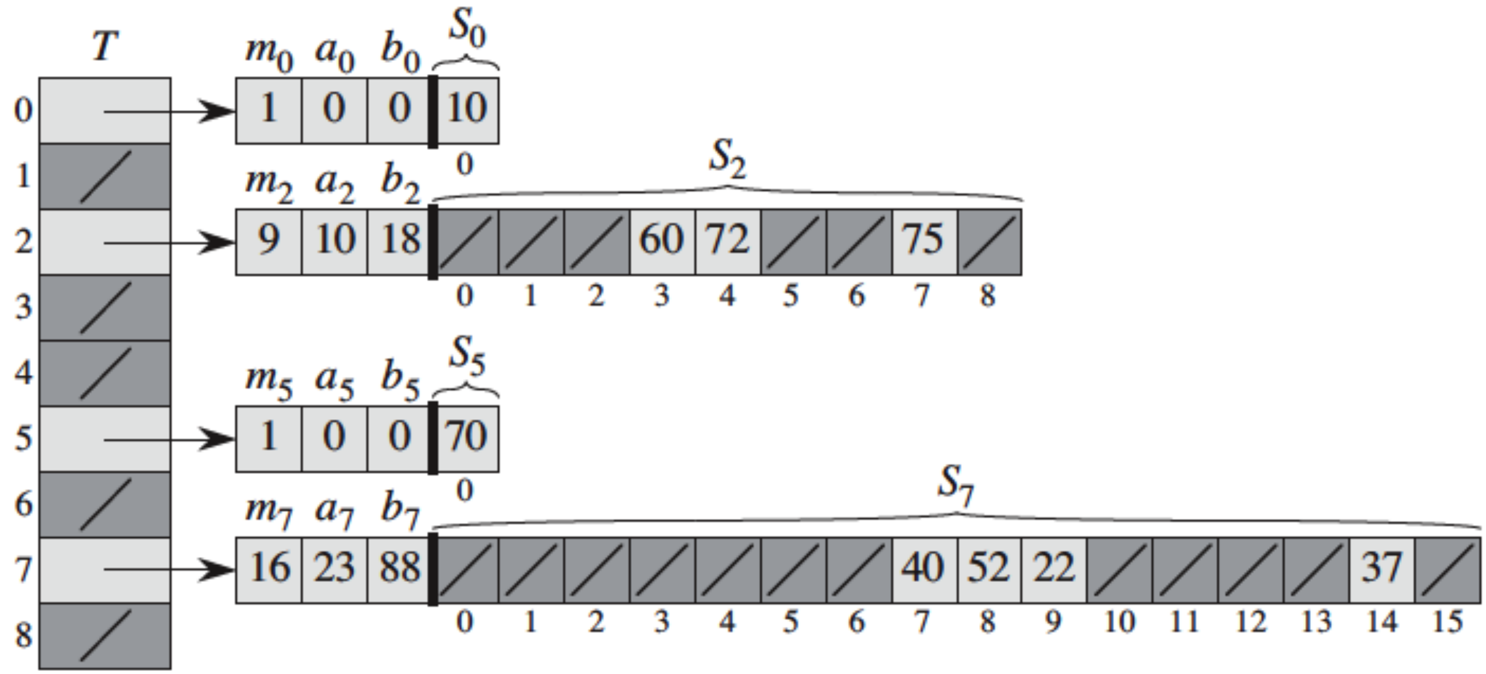


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, and so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2. No collisions occur in any of the secondary hash tables, and so searching takes constant time in the worst case.