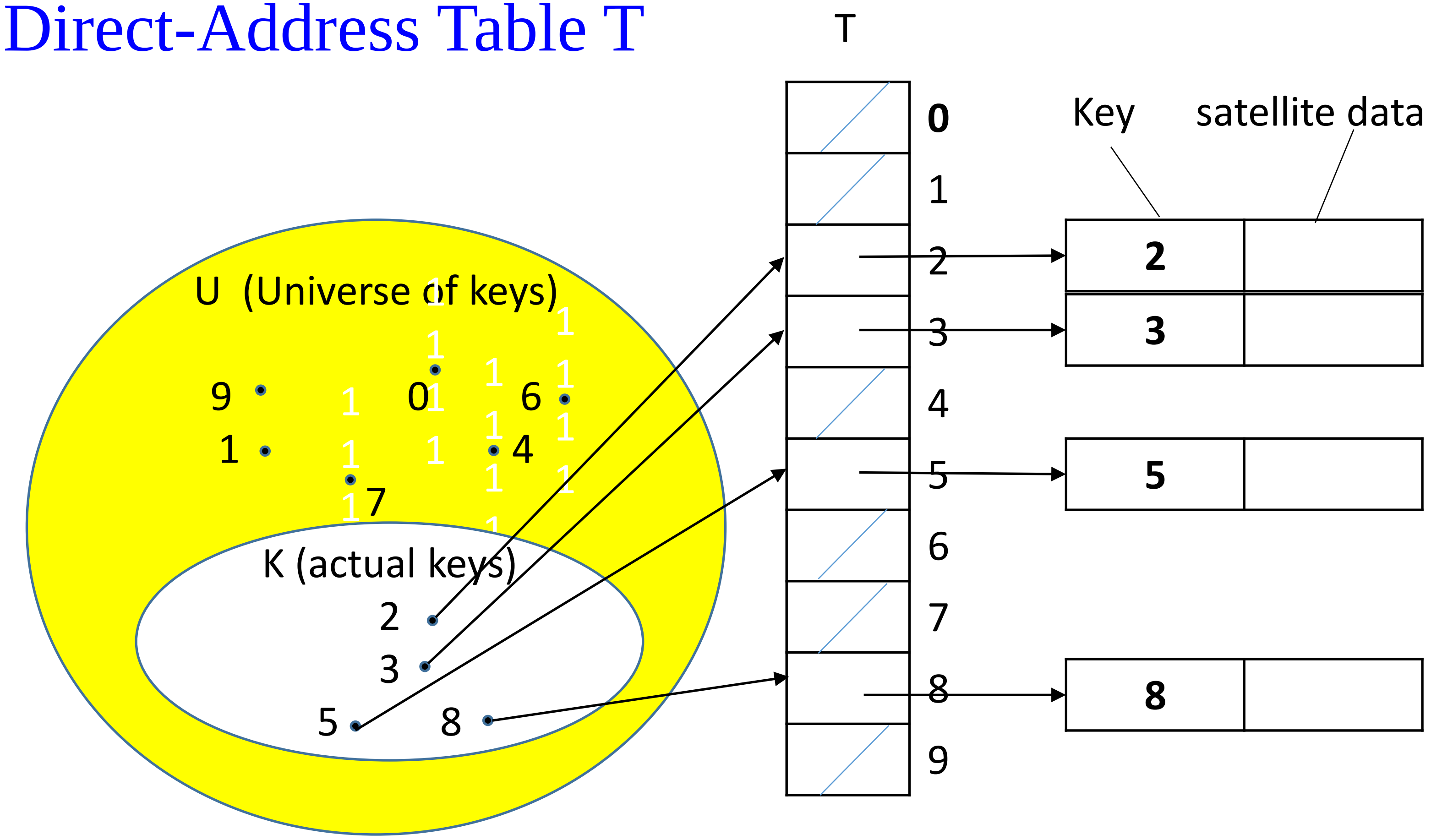
Chapter 00

Introducing Foundations

Contents

Hash Tables

Direct-Address Table T



The dictionary operations are:

```
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
```

Implement a dynamic set by a direct-address table T.
Each key in the universe U = {0, 1, 2, …, 9} corresponds to an index in table T. The set K = {2, 3, 5, 8} of actual keys determines the slots in the table that contains pointers to elements. The other slots contain / (NIL). Each of the dictionary operations is fast: only O(1) time is required.

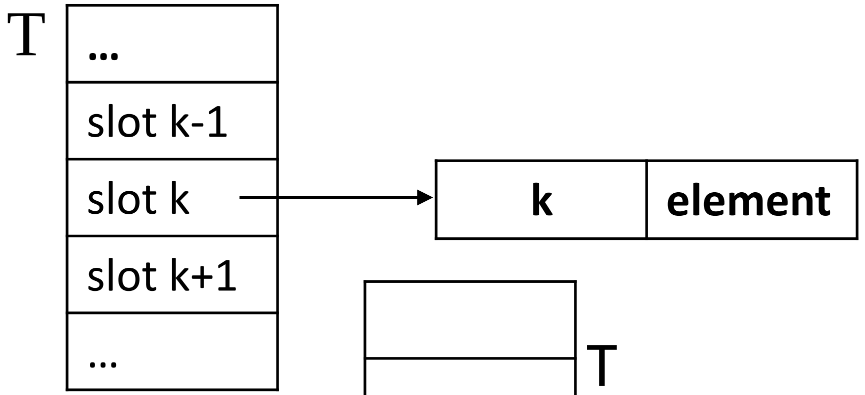
Direct-addressing

- The difficulty with direct addressing is:
  Storing a table T of size |U| is impractical, or impossible, for the large universe U.
- For storing a set K of keys in a table T of size |U|, m*ost of the space allocated for T would be is wasted,* if K $\ll$ U.
- For some applications
  - *to save space, store the object in the slot of the direct-address table T,* but NOT store the element's key and satellite data in an object external to the direct-address table T, with a pointer from a slot in the table to the object.
  - *Unnecessary to store the key field of an object in the slot*, if we have the index of the object in the table as its key.
  - Must have some *way to tell if the slot is empty* if the key is not stored.

Hash Tables

- For the direct-address table T:

  store an element with key k in slot k.

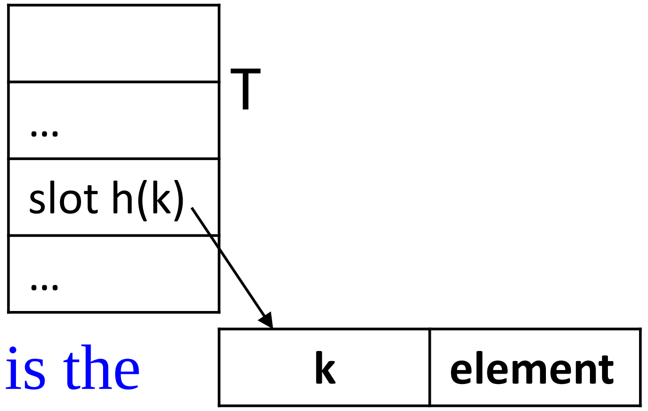


- For the hash table T,

  a hash function h computes the slot from the key k.

  store this element in slot h(k).

  An element with key k hashes to slot h(k), and h(k) is the hash value of key k.



- Define a hash function h mapping the universe U of keys into the slots (also called the positions) of a hash table T[0 .. m-1]:

  $h: U \rightarrow \{0, 1, \ldots, m-1\}$, $m << |U|$, and is defined by

  $h(k) = i$ for an $0 \leq i < m$.

Hash Tables

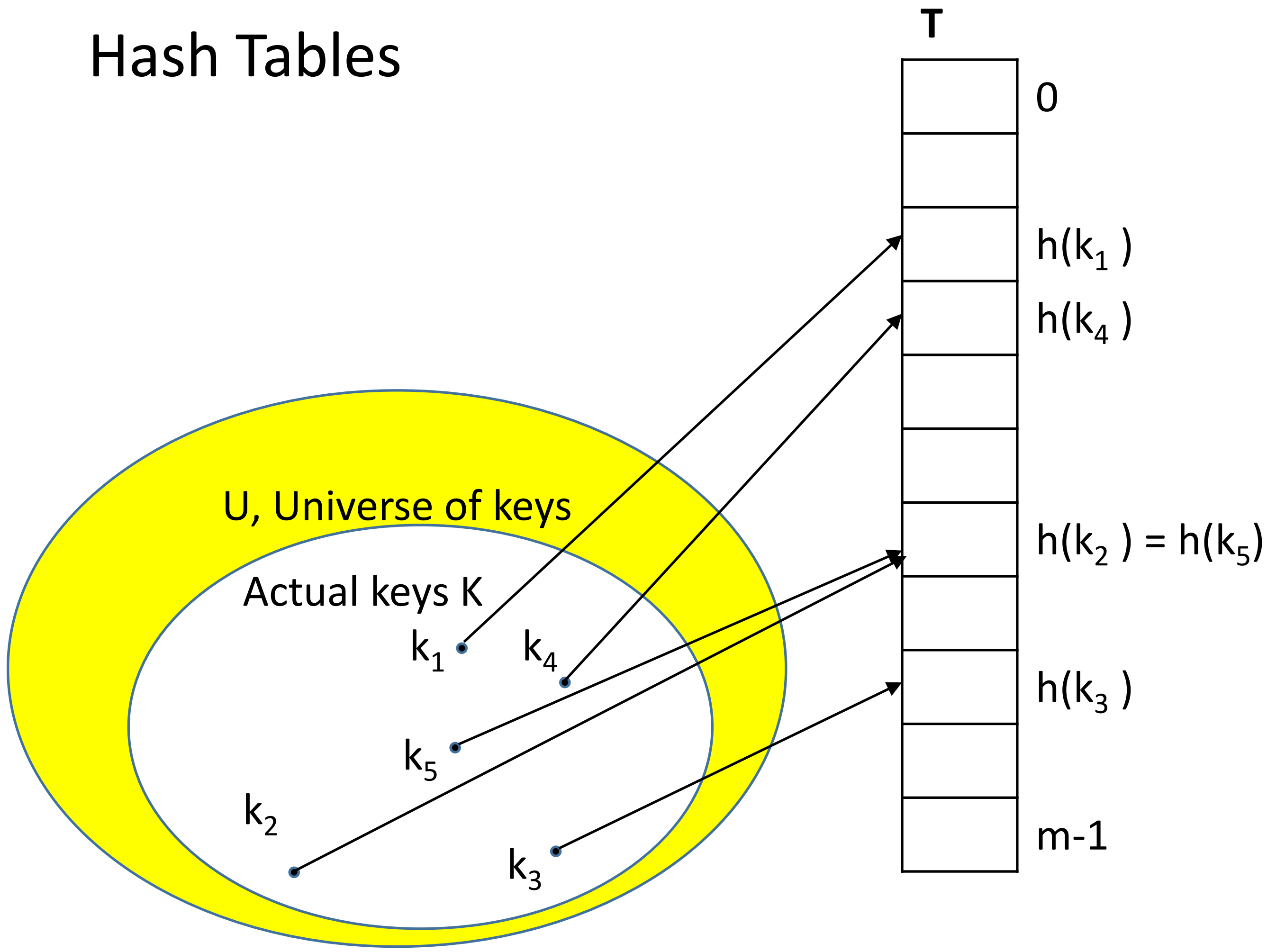


Figure Hash Table: Using a hash function h to map keys to hash-table slots. Both keys $h(k_2)$ and $h(k_5)$ map to the same slot, so they collide.

Hash Tables vs Direct-Address Tables Comparison

- Let K be the set of keys. Let U be the universe of all possible keys. When $K \ll U$, the storage requirements for
  - a direct-address table is $\theta(m)$, with the table size $m \cong |U|$.
  - the hash table is $\theta(|K|)$,
  - $\theta(|K|) << \theta(m)$.

- Searching for an element in
  - the hash table requires the average time $O(1)$.
  - the direct-address table requires the worst-case time $O(1)$.

Hash Tables vs Direct-Address Tables

Example: Encode the location of a computer on the Internet:

- use a 32-bit-IP (Internet protocol) address,
  - usually shown broken down into four 8-bit fields, such as 128.32.168.80.
- For table (direct) addressing,
  - achieve fast lookup times
    - if the records are maintained in an array indexed by IP address.
  - Waste memory space:
    - the array would have $2^{32} \approx 4 * 10^9$ entries, the vast majority of them blank.

Example: for reducing the storage requirements:

- Suppose no more than 251 IP (Internet protocol) addresses.
- Let N be the set of all 251 IP addresses (disregarding dots).
- Define a function Hash : N → {0, 1, … 250} by
  - Hash(n) = n mod 251, for every IP address n ∈ N, where
  - n mod 251 = n – 251* $\frac{n}{251}$, and $\frac{n}{251}$ is integer division.
    [r = x – y *q]
  - Place the record with IP address n in the position of Hash(n).
- For example, Hash(128.32.168.80) = 1283216880 (mod 251)

  = 1283216880 - (251 * 5112417) = 213
  - The record for the IP address 128.32.168.80 is located in position 213 of a table.

Hash Table

Hash	IP Addresses
0	
1	
2	
…	
171	73.102.177.154
172	73.102.177.405
…	
212	
213	128.32.168.80
214	
…	
250	
251	

Hash(73.102.177.154) = 171
Hash(73.102.177.405) = 171
They collide.

Hash Table

- A hash function h must be deterministic
  - h always produces the same output h(k) for the given input k.
- Impossible to Avoid collisions is impossible
  - since |U| > m, there must be two keys that have the same hash value.
- Minimize the number of collisions,
  - if a well-designed h appeared to be "random".
- Need a method for resolving the collisions that do occur.

Hash Table

Collision Resolution

- Any hashing scheme have a *collision resolution mechanism*.
- The two principal versions of hashing have different collision resolution mechanism:
  - *open hashing* (also called separate chaining, or chaining) and
  - *closed hashing* (also called open addressing).

- Various collision resolution methods are used, if collisions occur.

Hash Table

Linear Probing - a simple collision resolution method:

Let the table's size be m. Define a Hash() function.

- To place a record with IP address n in the position of Hash(n) = k, where $0 \le k < m$,
  - if the position k (=Hash(n)) is occupied,
    - search downward from this position k through m-1 to place the record in the first empty position found;
    - go back up to the beginning of the table from 0 to k -1 if necessary.
- To search a record in the table from its IP address n,
  - compute Hash(n) and
  - search downward from this position to find the record with IP address n.
    - Search may go from Hash(n) through m-1 and then from 0 to Hash(n) – 1.
- Quite an efficient way to store and locate records, when collisions are not many.

Collision Resolution by Chaining (or called Open Hashing)

Hash Table

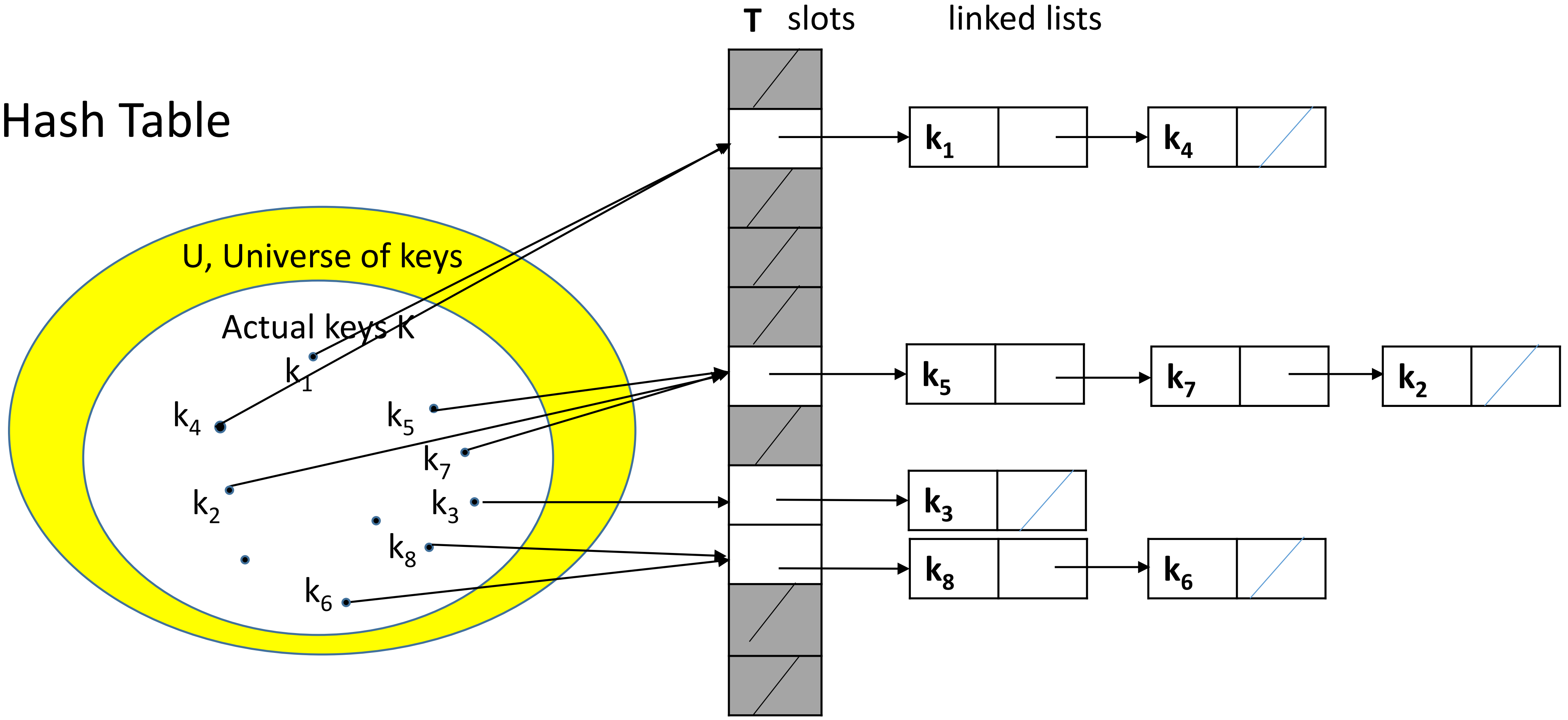


Figure 4: Collision resolution by chaining: Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$, where h() is a defined hash-function.

Collision Resolution by Chaining (or called Open Hashing)

For collisions are resolved by chaining

- In chaining, store all the elements that hash to the same slot (position) into the same linked list. The slot contains a pointer to the head of the list of these elements, otherwise, the slot contains a NIL. (See figure in the previous slide.)
- Let x be an element with key k. When collisions are resolved by chaining, the dictionary operations on a hash table T are:
  - Chained-Hash-Insert(T, x)

    Insert x at the head of the list T[h(key[x])]
  - Chained-Hash-Search(T, k)

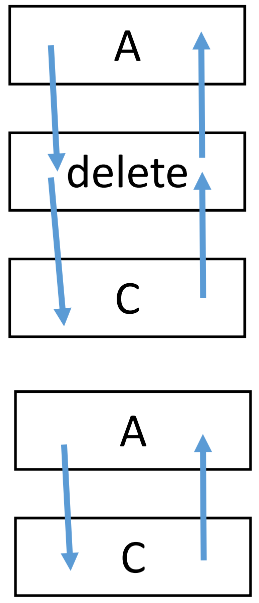
    Search for an element with key k in the list T[h(k)]
  - Chained-Hash-Delete(T, x)

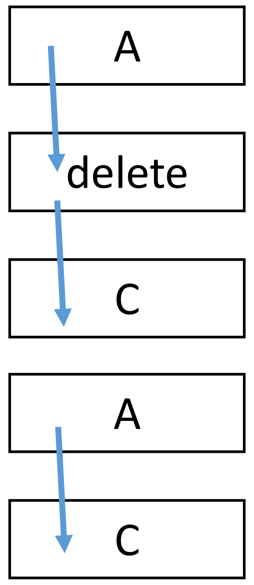
    Delete x from the list T[h(key[x])]

Hash Table

Time Efficiency for the Collision Resolution by Chaining

- The worst-case running time
  - for inserting an element x, is O(1).
  - for searching an element x, is proportional to the length L for the list; and is O(L).
- For deleting an element x,
  - if the lists are doubly linked, it requires O(1) time.
  - If the lists are singly linked, the deletion and searching have essentially the same running time
    - First find x in the list T[h(key[x])], so that the next link of x's predecessor can be properly set to splice x out.





Analysis of Hashing with Chaining (or called Open Hashing)

- How well does hashing with chaining perform?
- How long does it take to search for an element with a given key?

- Define the load factor α of the hash table T as follows.
  - Let the table T with m slots have n elements. The average number of elements per slot is

    $\alpha\ (or\ \alpha_T) = \frac{n}{m}$, which can be <, =, or >1,

    staying fixed, as n and m go to infinity.
- The worst-case behavior of hashing with chaining is terrible:
  - All n keys hash to the same slot, creating a list of length n.
    - The worst-case for the searching is θ(n) plus
    - the time for computing the hash function.
    - It is no better than if use one linked list for all the elements.
  - Thus, hash tables are not used for their worst-case performance.

Analysis of Hashing with Chaining (or called Open Hashing)

- The average-case performance of hashing depends on
  - how well the hash function h distributes the set of keys to be stored among the m slots, on the average.
  - Under the assumption of *simple uniform hashing*:
    - any given element is
      - *equally* likely to hash into any of the m slots,
      - *independent of* where any other element has hashed to.
  - Assume: the hash value h(k) can be computed in O(1) time.
  - Assume: the time required to search for an element with key k depends linearly on the length of the list T[h(k)], and the time to access slot h(k) (says, required O(1) for access slot).
- Consider the expected number of elements in the list T[h(k)] that are checked to see if any element have a key equal to k using a search algorithm. We have two cases:
  - Case I: the search is unsuccessful: no element in the table has key k.
  - Case II: the search successfully finds an element with key k.

Analysis of Hashing with Chaining (or called Open Hashing)

Theorem 12.1 (an unsuccessful search)

Under the assumption of simple uniform hashing, in a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\theta(1 + \alpha)$.

Proof: Under the assumption of simple uniform hashing, any key k is equally likely to hash to any of the m slots.

The average unsuccessful search time for a key k is

- the average time to search to the end of the list T[h(k)], where the average length of the list is the load factor $\alpha = \frac{n}{m}$.
- Thus, the expected number of elements examined in an unsuccessful search is α, and
- Total time required is $\theta(1 + \alpha)$, including $\theta(1)$, the time for computing h(k).

Analysis of Hashing with Chaining (or called Open Hashing)

Theorem 12.2 (a successful search)

Under the assumption of simple uniform hashing, in a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\theta(1 + \alpha)$.

Proof:

Theorem 12.2

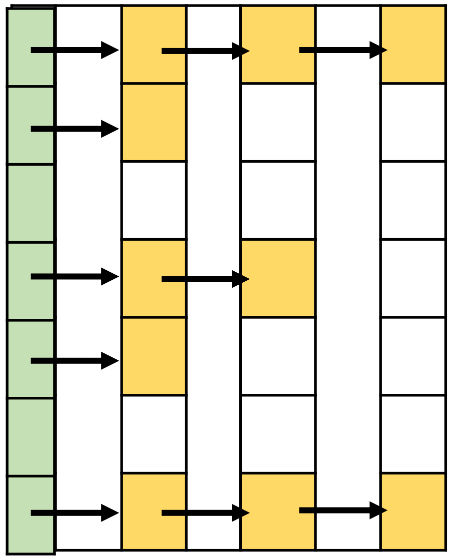
Under the assumption of simple uniform hashing, in a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\theta(1 + \alpha)$.

Proof: Assume that:

1. The key/element being searched for is equally likely to be any of the n keys/elements stored in the table.
2. The Chained-Hash-Insert procedure inserts a new element at the end of the list instead of the front.

   The average successful search time is the same whether new elements are inserted at the front of, or the end of the list.

   The expected number of elements examined during a successful search is 1 more than the number of elements examined when the sought-for the element was inserted (since every new element goes at the end of the list).



Theorem 12.2

Under the assumption of simple uniform hashing, in a hash table in which collisions are resolved by chaining, a successful search takes average-case time $\theta(1 + \alpha)$.

Proof: (continued)

In a successful search, the expected number of elements examined over the n items in the table of size m is

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

where $\frac{i-1}{m}$ is the expected length of the list to which the ith element is added.

$$\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{i-1}{m} \right) &= \frac{n}{n} + \frac{1}{nm}\sum_{i=1}^{n}(i - 1) \\ &= 1 + \frac{1}{nm}\left(\frac{(n-1)n}{2}\right) = 1 + \frac{n^2}{2nm} + \frac{n}{2nm} \\ &= 1 + \frac{\alpha}{2} - \frac{1}{2m}\end{aligned}$$

Thus, the total time required for a successful search (including the time for computing the hash function) is $\theta(2 + \frac{\alpha}{2} - \frac{1}{2m}) \approx \theta(1 + \alpha)$.

Analysis of Hashing with Chaining (or called Open Hashing)

What does this analysis mean?

- Searching takes constant time on average.
  - If the number m of hash-table slots is at least proportional to the number n of elements in the table, then we have n = O(m) and, consequently, $\alpha = \frac{n}{m} = \frac{O(m)}{m} = \frac{c_0 m + c_1}{m} = O(1)$.

- All dictionary operations can be supported in O(1) time on average when the lists are doubly linked. since
  - insertion takes O(1) worst-case time,
  - deletion takes O(1) worst-case time when the lists are doubly linked, and
  - searching takes constant time on average.

Hash Functions – What makes a good hash function?

- Let U be the large set of possible keys and let m be the size of a table. Assume that $U \gg m$.
- A hash function h: $U \rightarrow \{0, 1, ..., m-1\}$, converting a key into the hash value (referred to as an array index).
- What makes a good hash function?
- What do we do if two or more distinct keys have the same hash value (a collision)?

Hash Function – What makes a good hash function?

- What makes a good hash function?
- A good hash function is easy to compute and has minimal collisions.
- A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
    - For a random key, the probability of each hash value would be equally likely.
    - Assure keys with their records are distributed evenly throughout the hash table
    - This assurance
        - depends on what the distribution of possible key values is.
        - This distribution may not be known ahead of time.

Hash Functions – What makes a good hash function?

- A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots.
  - Assume: Let P be a probability distribution. The probability P(k) is that the key k is drawn independently from U.
    Then, the assumption of simple uniform hashing is that
    $$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ for } 0 \le j < m. \qquad \text{….. 12.1}$$
  - It is generally not possible to check this condition, since P is usually unknown. P is rarely known.
  - Suppose keys are known to be random real numbers k's independently and uniformly distributed in the range $0 \le k < 1$. that is, the probability distribution *P(k)* of each hash value would be equally likely. In this case, the hash function

    $$h(k) = \lfloor km \rfloor$$

    can be shown to satisfy equation 12.1, the condition of simple uniform hashing.

Hash Functions – What makes a good hash function?

Interpreting keys as natural numbers

- Assume that the universe of keys is the set N = {0, 1, 2, …} of natural numbers.
- Find a way to interpret keys as large natural numbers, if they are not natural numbers. Then, compute the hash value.
- Example:
  - If a key is a character string, then interpret it as an integer expressed in suitable radix notation.
  - Let interpret pt as the pair of decimal integers (112, 116), according to the ASCII character set. Then, expressed as a radix –128 integer, pt becomes

    (112*128) + 116 = 14452.
  - opt can be computed (((111*128) + 112)*128) + 116 = 1833076, using Horner Rule.

Example of Hash for Character Strings:

If K is a character string $c_0\ c_1\ \ldots\ c_{n-1}$, then, as an unsophisticated option, use

$$h(K) = \left(\sum_{i=0}^{n-1} ord(c_i)\right) \bmod m,$$

where m is a given table size, and $ord(c_i)$ *is the ordinal value of* c_i.

For a long character string K, a better option is to compute h(K) by using Horner's rule to calculate the result piecewise:

```
h ← 0;
for (i ← 0 to n - 1) do {
        h ← (h * C + ord(c_i))mod m; }
```

where C is a constant larger than every ord(c_i). e.g., C can be 128 if ASCII codes are used. As an example, when i = 2, h($c_0c_1c_2$) is

$((ord(c_0) \bmod m * C + ord(c_1)) \bmod m * C + ord(c_2)) \bmod m$

Hash Functions – What makes a good hash function?

Modular Hashing – the Division Method for creating hash functions:

- The modular hashing method is:
  - Map a key k into one of the m slots by taking the remainder of k divided by m. The hash function is h(k) = k mod m.
- Good values for m are primes not too close to exact powers of 2
- Choosing m to be closer to a power of two, 2^p, could cause problems.
  - If m = 2^p, h(k) is *the p number of lowest-order bits of k*. High frequency of collision occurrences.
  - An example: let m = 2^4. If k =13 (**1101**), 29 (1**1101**), 45 (10**1101**), 61 (11**1101**) …, then the hash value h(k) = 1101_2 is just the 4 lowest –order bits of k. In comparison with m = 13, then their hash value h(k) would be 0000, 0011, 1011, 1001, … respectively.
- Avoid choosing m = 10^p, if the application deals with decimal numbers as keys.

Hash Functions – What makes a good hash function?

The division method for creating hash functions

- An example of a good prime value m, not too close to exact powers of 2.
  - Suppose the character code has 8 bits, and let n = 2000 character strings. We could allocate a hash table of size m = 701 if we don't mind examining an average of 3 elements in an unsuccessful search. 701 is a prime and $701 \cong \frac{2000}{3}$ but not any power of 2.
  - For any integer key k, the hash function would be h(k) = k mod 701.

Hash Functions – What makes a good hash function?

The significant bits for creating hash functions

- Two obvious hash functions: either the first (most significant) or last (least significant) p bits of the key k.
- Assume that the table size is 2^p . (i.e., m = 2^p) --- for ease of implementing the h(k) function
- Let k be an n-bit integer.
- Compute these hash functions as follows:
- The hash function h(k) = $\lfloor \frac{k}{2^{n-p}} \rfloor$
  - the hash value of the first p bits of k. (quotient).
- The hash function h(k) = k mod 2^p
  - the hash value of the last p bits of k. (remainder).
- Time for computing these hash functions is fast.
- They are bad hash functions, especially for strings.

  e.g., h(xbcde) = h(ybcde) = h(zbcde)

k = **11 1011** 0011 n=10bits
p = 4 . $2^4 = 1\,0000.$
$2^{n-p} = 2^6 = 100\,0000.$
Then **11 10**11 0011 / 100 0000
q = **1110** and r = 11 0011

k = 11 1011 0011 n=10bits
p = 4 . $2^p = 2^4 = 1\,0000.$
k mod $2^p = 0011.$
Then 11 1011 **0011** / 1 0000
q = 11 1011 and r = **0011**

Hash Functions – What makes a good hash function?

Consider the bits in middle for improving the significant bits method.

The Multiplication method for creating hash functions

- This method involves two steps:
  - Multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA via $(kA - \lfloor kA \rfloor)$. ---$(3210*0.62 - \lfloor 3210*0.62 \rfloor) = 0.2$
  - Compute $\lfloor m(kA - \lfloor kA \rfloor) \rfloor$. --- $\lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor 128*0.2 \rfloor = 25$
  - Then the hash function is $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m(kA \bmod 1) \rfloor$

    where "k A mod 1" means the fractional part of k A,

    i.e., $kA \bmod 1 = kA - 1*\lfloor kA \rfloor = kA - \lfloor kA \rfloor$.

    --- $\lfloor m(kA \bmod 1) \rfloor = \lfloor 128(3210*0.62 \bmod 1) \rfloor = \lfloor 128(1990.20 \bmod 1) \rfloor = 128*0.20 = 25$

- ~~If $A = 1/2^n$ and $m = 2^p$, it is easy to compute h(k) is bits n thru n + p.~~

Hash Functions – What makes a good hash function?

The Multiplication method for creating hash functions

- That the value of m is not critical is an advantage of the multiplication method.
- Choose $m = 2^p$ for some integer p.
- Suppose that the machine's word size is w bits and a key k fits into a single word, the w-bit representation of the key k.
- Restrict A to be a fraction of the form $s / 2^w$ where s is an integer in the range $0 < s < 2^w$. i.e., $s = \lfloor A * 2^w \rfloor$.
- First multiply k by the w-bit integer $\lfloor A * 2^w \rfloor$, $0 < A < 1$
- The result is a 2w-bit value $r_1 2^w + r_0$, where
  - r_1 is the high-order word of the product and
  - r_0 is the low-order word of the product.
- The desired p-bit hash value consists of the p most significant bits of r_0.

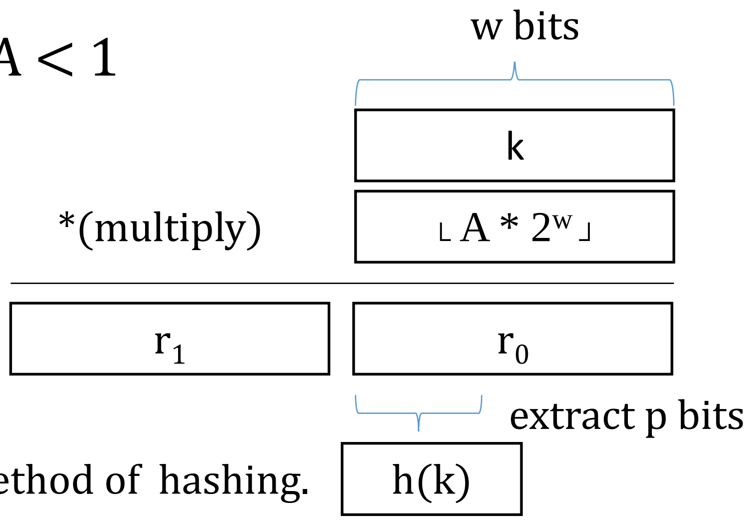


Fig: The multiplication method of hashing.

$$h(k) = \left\lfloor \frac{k}{2^{n-p}} \right\rfloor \quad \text{where } k = r_0,\ n = 32,$$

Hash Functions – What makes a good hash function?

The Multiplication method for creating hash functions

- Knuth suggests the choice of A as $A \cong (\sqrt{5} - 1)/2 = 0.6180339887\ldots$ (the golden ratio) seems to work reasonably well.
- As an example, $k = 123456$, $p = 14$, $m = 2^{14} = 16384$, and $w = 32$.
- Adapting Knuth's suggestion, choose an A to be the fraction of the form $s / 2^{32}$ where $0 < s < 2^{32}$ and $A \cong (\sqrt{5} - 1)/2$ so that $A = 2654435769 / 2^{32}$, where 2^{32} is 4,294,967,296.
- Then, $k*s = 327706022297664 = (76300 * 2^{32}) + 17612864$.
- Therefore, $r_1 = 76300$ and $r_0 = 17612864 < 2^{32}$.
- The p (=14) most significant bits of r_0 yield the value $h(k) = \left\lfloor \frac{17612864}{2^{32-14}} \right\rfloor = \left\lfloor \frac{17612864}{262144} \right\rfloor = 67$.

Hash Functions – What makes a good hash function?

The Multiplication method for creating hash functions

- Knuth suggests the choice of A as $A \cong (\sqrt{5} - 1)/2 = 0.6180339887\ldots$ (the golden ratio) seems to work reasonably well.
- Example: $k = 123456$, $m = 10000 < 2^{14}$, and $A = 0.6180339887\ldots$, then

$$
\begin{aligned}
h(k) &= \lfloor m(k\,A \bmod 1) \rfloor \\
&= \lfloor 10000(123456 * 0.61803\ldots \bmod 1) \rfloor \\
&= \lfloor 10000(76300.0041151\ldots \bmod 1) \rfloor \\
&= \lfloor 10000\,(0.0041151\ldots) \rfloor \\
&= \lfloor 41.151 \rfloor \\
&= 41.
\end{aligned}
$$

If $m = 2^{14}$, then $h(k) = 67$

Hash Functions – What makes a good hash function?

Universal hashing

- An ideal hash function?
- For any hash function h there exists a "bad set of keys", which says n keys, chosen by the malicious adversary, that all hash to the same slot, yielding an average retrieval time of Θ(n).
    - For these n number of keys, with collision resolution by chaining in an initially empty table with m slots, it takes Θ(n) steps to handle any sequence of n inserts, deletes, and searches containing O(m) inserts.
    - This would cost more than searching through the linked list due to hash function computations.
- The solution is to choose a hash function *randomly* from a family of hash functions, instead of using a fixed hash function, for which an adversary can always find a bad set of keys.
- The **universal hashing** approach is to choose the hash function ***randomly*** in a way that is **independent** of the keys that are actually going to be stored.

Hash Functions – What makes a good hash function?

Universal hashing

- An ideal hash function?
- The **universal hashing** approach is to choose the hash function ***randomly*** in a way that is **independent** of the keys that are actually going to be stored.
- The main idea behind universal hashing is to select the hash function at random at run time from a carefully designed class of functions.
- The probability of a collision between any two keys is provably 1/m, where m is the size of the table.
- Implementing universal hash functions necessarily involves randomization.
  - The first approach: Select the hash function at random at run time from a carefully designed class of functions (for yielding the desired property).
  - The second approach: Randomize the hash function itself.

Hash Functions – What makes a good hash function?

Universe hashing

- Randomization for these two approaches
  - Choose the hash function *randomly* in a way that is *independent* of the keys that are actually going to be stored.
  - guarantees good average-case performance, no matter what keys are provided as input.
  - guarantees that no single input always evoke worst-case behavior.
- Any fixed hash function is vulnerable to the worst-case behavior (all keys hash to the same slot).
- Universal hashing is the only effective way to improve the situation.

A Modular Hash Function for Character Strings:

If K is a character string $c_0\ c_1\ \ldots\ c_{n-1}$, then, as a very unsophisticated option, use

$$h(K) = \left(\sum_{i=0}^{n-1} ord(c_i)\right) \bmod m,$$

where m is the size of a given table. $ord(c_i)$ *is the ordinal value of* c_i

For a long character string, a better option is to compute h(K) as follows: (using Horner's rule to calculate the result piecewise)

```
h ← 0;
for (i ← 0 to n – 1) do {
    h ← (h * C + ord($c_i$)) mod m; }
```

where C is a constant larger than every ord(c_i). e.g., C can be 128 if ASCII codes are used.

Hash Functions – What makes a good hash function?

A Universal Hash Function for Strings

- Consider again the modular hash function for strings
- A way to randomize this hash function is to randomize the value of constant C.
- Use a simple pseudo-random number generator for this purpose.

```
hash(str, m)
    h ← 0; C ← 31415; x ← 27183;
    for (i ← 0 to n – 1) do {
        h ← (h * C + ord($c_i$)) mod m;
        C ← (C+x) mod (m-1)}
    if (h < 0)  return h + m
    else return h;
```

Hash Functions – What makes a good hash function?

A Universal Hash Function for Strings

- Use a simple pseudo-random number generator for this purpose.

```
hash(str, m)
    h ← 0; C ← 31415; x ← 27183;
    for (i ← 0 to n – 1) do {
        h ← (h * C + ord(c_i)) mod m;
        C ← (C+x) mod (m-1)}
    if (h < 0)  return h + m
    else return h;
```

- This idea can be extended to integers by multiplying each bye by a random coefficient in the same manner.

- It can shown that this method does produce a universal hash function under the assumption that the coefficients are truly random.

Hash Functions – What makes a good hash function?

A Universal Hash Function for Integers

- Let p be a large prime number such that every k key value is between 0 and p -1.
- Let a and b be integers smaller than p with a positive and be nonnegative.
- If a and b are selected randomly, then the hash function

$$H_{a.b}\ (k) = (((\ ak + b)\ \text{mod}\ p)\text{mod}\ m) \qquad \ldots.\ \text{I}$$

is universal.

Hash Functions – What makes a good hash function?

Universe hashing

- Let U be a universe of keys, and let *H* be a finite collection of hash functions mapping U to {0, 1, 2, …, m-1}. This collection is said to be universal if for each pair of distinct keys x, y ∈ U, the number of hash functions h ∈ *H* for which h(x) = h(y) is precisely $\frac{|H|}{m}$.
  - If h is chosen uniformly at random from *H*, the probability of a collision between x and y is:

$$\frac{\#\ functions\ that\ map\ x\ and\ y\ to\ the\ same\ slot}{Total\ \#\ of\ functions} = \frac{\frac{|H|}{m}}{|H|} = \frac{1}{m}.$$

  - i.e., with a hash function randomly chosen from *H*, the chance of a collision between x and y when x ≠ y is exactly $\frac{1}{m}$, which is exactly the chance of collision if h(x) and h(y) are randomly chosen from the set {0, 1, 2, …, m-1}.

Hash Functions – What makes a good hash function?

Universe hashing:

The following theorem shows that a universal class of hash functions gives good average-case behavior

Theorem 12.3: If h is chosen from a universal collection H of hash functions and is used to hash n keys into table T of size m, where $n \leq m$, then the expected number E of collisions involving a particular key x is less than 1. In other words,

$$E[\#\text{ of collisions with } x] < \frac{n}{m},$$

where $\frac{n}{m} = \alpha$ = load factor of hash table

Proof:

Hash Functions – What makes a good hash function?

Universe hashing:

Theorem 12.3: If h is chosen from a universal collection H of hash functions and is used to hash n keys into table T of size m, where n ≤ m, then the expected number of collisions involving a particular key x is less than 1.

Proof: For each pair of distinct keys y, z, let a random variable, c_{yz} = 1 if h(y) = h(z) (i.e., if y and z collide using h), and c_{yz} = 0 otherwise. Since, with a hash function randomly chosen from *H*, the chance of a collision between y and z when y ≠ z is exactly $\frac{1}{m}$, then the expected value of the random variable c_{yz} is $E[c_{yz}] = \frac{1}{m}$.

Let C_x be the total number of collisions involving key x in a hash table T of size m containing n keys. Then, the expectation of collisions involving x,
$E[C_x] = \sum_{y\in T, y\neq x} E[C_{xy}] = \frac{n-1}{m}$, using E[X + Y] = E[X} + E[Y].
Since n ≤ m, we have $E[C_x] < 1$.

Hash Functions – What makes a good hash function?

Universe hashing

Design/construct a universal class of hash functions.

1. Choose a prime m to be the table size.
2. Decompose a key x into r + 1 bytes (i.e., r + 1 characters, or fixed-width binary substrings), such that x = < x_0 , x_1 , …, x_r> where x_i ∈ {0, 1, 2, …, m-1} the only requirement that the maximum value of a byte should be less than m.
3. Pick randomly a = < a_0 , a_1 , …, a_r>, where a_i ∈ {0, 1, 2, …, m-1}, a sequence of r + 1 elements chosen randomly from the set {0, 1, 2, …, m-1}.
4. Define $h_a \in H$ as (x) = $\sum_{i=0}^{r} a_i\, x_i \bmod m$. ……….. 12.3

With this definition, $H = \bigcup_a \{ha\}$ has m^{r+1} members. ……….. 12.4

(i.e., the total number of hash functions in the family is m^{r+1} since each of the r+1's has m possible values.)

Theorem 12.4: The class *H* defined by equation 12.3 and 12.4 is a universal class of hash function.

Hash Functions – What makes a good hash function?

Universe hashing

Theorem 12.4: The class *H* defined by equation 12.3 and 12.4 is a universal class of hash function.

Proof: Consider any pair of distinct keys x, y. Assume that $x_0 \neq y_0$. For any fixed values of a_0, a_1 , …, a_r, there is exactly one value of a_0 that satisfies the equation h(x) = h(y); this a_0 is the solution to

$$a_0 (x_0 - y_0) \equiv - \sum_{i=0}^{r} a_i (xi - yi) \pmod{m}.$$

Since m is prime, the nonzero quantity $x_0 - y_0$ has a multiplicative inverse modulo m, and thus there is a unique solution for a_0 mod m. Therefore each pair of keys x and y collides for exactly m^r value of a, since they collide exactly once for each possible value of $< a_1 , \ldots, a_r >$ (i.e., for the unique value of a_0 noted above). Since there are m^{r+1} possible values for the sequence a, keys x and y collide with probability exactly $\frac{m^r}{m^{r+1}} = \frac{1}{m}$. Therefore, *H* is universal.

Hash Functions – What makes a good hash function?

Universe hashing - Example: Hashing IP Addresses

Let Universe U contain IP addresses. Each IP address is a 32-bit 4-tuple $< x_1, x_2, x_3, x_4 >$ where $x_i \in \{0, \ldots, 255\}$.

Let the size of table T be a prime number m (e.g., m = 997 if we need to store 500 IPs).

Define h_a for the 4-tuple $a = < a_1, a_2, a_3, a_4 >$ where $a_i \in \{0, \ldots, m-1\}$.

$$h_a : \text{IP address} \rightarrow \text{Slot}$$

Question: Using the above formulation of H, compute which slot IP address $< x_1, x_2, x_3, x_4 >$ hashes to?

Ans: Using the following equation which requires constant space and time: $h_a(x_1, x_2, x_3, x_4) = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \pmod{m}$

Hash Functions – What makes a good hash function?

Perfect Hashing

So far, we have dealt with the expected value, but what if we need to achieve a worst-case performance with the search time O(1). This is where Perfect Hashing comes in, but there is a price to pay; the table must be static. Static tables suit certain applications that are not dynamic such as symbol tables and files on a CD.

Idea

In (static) perfect hashing to achieve a worst-case performance, use a two-step hashing scheme similar to the double hashing scheme in open addressing.

A two-step scheme with universal hashing at each step such that there are no collisions at step 2. If items hash to slot i, then a table at slot i is used that has size, giving sparsity which allows us to easily find hash functions that would not cause collisions at step 2 (collisions may occur at step 1).

A search here takes O(1) time in the worst case for any key

Collision Resolution

- *open hashing* (also called separate chaining, or chaining) and
- ***closed hashing* (also called *open addressing*).**

Open Addressing (Closed Hashing)

In open addressing,

- Store all keys (or elements) in the hash table itself without the use of linked lists.
  - That is, each table entry contains either an element of the dynamic set or NIL.
- Table size m must be at least as large as the number of keys n.
- Hash table can “fill up” until no further insertion can be made; the load factor α can never exceed 1.
- Advantage of open addressing is that no pointers are used.

Open Addressing (Closed Hashing)

In open addressing,

- For insertion, examine (called *probe*) successively the hash table until an empty slot for storing the key is found.
- The sequence of positions probed *depends upon the key being inserted*.
- To determine which slots to probe, *extend the hash function* to include the probe number (starting from 0) as a second input. The hash function becomes

  $$h: U \times \{0, 1, 2, \ldots, m-1\} \rightarrow \{0, 1, 2, \ldots, m-1\}$$

- With open addressing, for every key k, the probe sequence

  $$\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$$

  is required to be a permutation of $\langle 0, 1, 2, \ldots, m-1 \rangle$. Every hash-table position is eventually considered as a slot for a new key as the table fills up.

Open Addressing (Closed Hashing)

- Assume that the elements in the hash table T are keys with no satellite information; the key k is identical to the element containing key k. Each slot contains either a key or NIL (if the slot is empty). Let h: U x {0, 1, 2, …, m-1} → {0, 1, 2, …, m-1}.

Hash-Insert(T, k)

```
    i ← 0
    repeat  j ← h(k, i)
            if (T[j] == NIL) //or NIL || DELETED if deletion is included
            then {T[j] ← k
                  return j; }
            else i++;
    until i = m;
error “hash table overflow”
```

Open Addressing (Closed Hashing)

- The procedure Hash-Search takes as input a hash table T and a key k, returning j if slot j is found to contain key k, or NIL if key k is not present in table T. Let h: U x {0, 1, 2, …, m-1} → {0, 1, 2, …, m-1}.

```
Hash-Search(T, k)
    i  ← 0
    repeat  j ← h(k, i)
            if (T[j]  ==  k)
                then return j;
            i++;
    until T[j] == NIL or i  == m; //use NIL || DELETED
    return NIL
```

Open Addressing (Closed Hashing)

- Deletion from an open-address hash table is difficult.
- Deleting a key from slot i :
  - Cannot simply mark this slot as empty by storing NIL in it.
    - This makes it impossible to retrieve any key k during whose insertion we had probed slot i and found it occupied.
  - A solution is to mark the slot by storing in it the special value DELETED instead of NIL.
- Modify the procedure Hash-Search to keep on looking when the value DELETED is encountered, while Hash-Insert inserts a new key if a slot has DELETED for if it were empty.
- With these, the search times are no longer dependent on the load factor α, and
  - for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Insert(h(k', i) = i

…

Insert(h(k, i) = i) $\neq nil$

…

InsertInsert(h(k, j) = j)

Delete(h(k', i) = i. Nil

Delete(h(k, i) = i) not found!

Open Addressing (Closed Hashing)

- In our analysis, we make the assumption of *uniform hashing*: Assume that each key considered is equally likely to have any of the m! permutations of {0, 1, 2, …, m-1} as its probe sequence.
- Uniform hashing generalizes the notion of simple uniform hashing to the situation in which the hash function produces not just a single number, but a whole probe sequence.
- True uniform hashing is difficult to implement. In practice, suitable approximations (such as double hashing) are used.

Open Addressing (Closed Hashing)

- Three commonly used techniques for computing the probe sequences required for open addressing:
    - linear probing,
    - quadratic probing, and
    - double hashing.
- These techniques guarantee that the probe sequence <h(k, 0), h(k, 1), …, h(k, m-1)> is a permutation of <0, 1, 2, …, m-1> for each key k.
- None of them fulfills the assumption of *uniform hashing*, since none of them is capable of generating more than m^2 different probe sequences (instead of the m! that uniform hashing requires).
- Double hashing has the greatest number of probe sequences and seems to give the best results.

Open Addressing (Closed Hashing)

Linear Probing

- Given an ordinary hash function $h': U \rightarrow \{0, 1, 2, \ldots, m-1\}$, the method of *linear probing* uses the hash function

  $h(k, i) = (h'(k) + i) \bmod m$ for $i = 0, 1, 2, \ldots, m-1$.

- Given key k, the first slot probed is $T[h'(k)]$. The next probe slot $T[h'(k) + 1]$, and so on up to the slot $T[m-1]$. Then, wrap around to slots $T[0], T[1], \ldots$, until we finally probe slot $T[h'(k) - 1]$.

- Items are inserted in the first empty slot with an address equal (if $i = 0$) or greater than (if $0 < i < m$) to the hashed address (wrapping around at the end of the table).
- To search an item, start at the hash address and continue to search each succeeding address until encountering a match or an empty slot.

Open Addressing (Closed Hashing)

Linear Probing

- Given an ordinary hash function $h': U \rightarrow \{0, 1, 2, \ldots, m-1\}$, the method of *linear probing* uses the hash function

  $$h(k, i) = (h'(k) + i) \bmod m \qquad \text{for } i = 0, 1, 2, \ldots, m-1.$$

- Deleting an item is difficult.
- Cannot just simply remove the item to be deleted.
- A solution is to replace the item with a sentinel "DELETED" that does not match any key and can be replaced by another item later on.
- Another solution is to rehash all items between the deleted item and the next empty space.
- Since the initial probe position determines the entire probe sequence, only m distinct probe sequence are used with linear probing.

Open Addressing (Closed Hashing)

Linear Probing

- Linear probing is easy to implement.
- It suffers from a problem known as *primary clustering*.
- Long runs of occupied slots build up, increasing the average search time.
- If the table has n = m/2 keys, where every even-indexed slot is occupied and every odd-indexed slot is empty, then the average unsuccessful search takes 1.5 probes. (1 + n/m) = (1 + (m/2)/m) = 1 + 0.5 = 1.5
- If the first n = m/2 locations are the ones occupied, the average number of probes increases to about n/4 = m/8.
- Clusters are likely to arise, since if an empty slot is preceded by i full slots, then the probability that the empty slot is the next one filled is (i+1)/m, compared with a probability of 1/m if the preceding slot was empty.
- Thus, runs of occupied slots tend to get longer, and linear probing is not a very good approximation to uniform hashing.

Open Addressing (Closed Hashing)

Quadratic Probing This alleviates the clustering problem by skipping slots.

- Uses a hashing function of the form

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

  where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $0 \leq i < m$.

- The initial position probed is T[h'(k)]; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i.
- This method works much better than linear probing, but to make full use of the hash table, the values of c_1, c_2 and m are constrained.
- If two keys have the same initial probe position, then their probe sequences are the same, since $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = h(k_2, i)$. This leads to a milder form of clustering, called *secondary clustering.*

  $h'(k) + c_1 i + c_2 i^2$
  $= h'(k')+c_1 i+c_2 i^2$

- As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

Open Addressing (Closed Hashing)

Double hashing

- Uses a hashing function of the form

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

  where h_1 and h_2 are auxiliary hash functions, and $0 \leq i < m$.

- The $h_2(k)$ value must be relatively prime to m for the *entire* hash table to be searched, i.e., the sequence to include all possible addresses.
  - Otherwise, if $GCD(h_2(k), m) = d > 1$ for some key k, then a search for key k would examine (1/d)th of the hash table.
- A convenient way to ensure this condition is to let m be a power of 2 and to design h_2 so that it always returns and produces an odd number. Another way is to let m be prime and to design h_2 so that it always returns a positive integer less than m.
- Example: choose m prime and let

  $$h_1(k) = k \bmod m;\ h_2(k) = 1 + (k \bmod m'),$$

  where m' is chosen to be slightly less than m (says, m -1 or m-2).

Open Addressing (Closed Hashing)

Double hashing

- Uses a hashing function of the form

  $h(k, i) = (h_1(k) + i\, h_2(k)) \bmod m$, where h_1 and h_2 are auxiliary hash functions, and $0 \le i < m$.

- Example: Let $k = 123456$, $m = 701$, and $m' = 700$.

  Define $h_1(k) = k \bmod m$; and $h_2(k) = 1 + (k \bmod m')$.

  Then $h_1(k) = 123456 \bmod 701 = 80$ and $h_2(k) = 1 + (123456 \bmod 700) = 257$.

  So, the first probe is to position 80, and then every 257th slot (mod m) is examined until the key is found or every slot is examined.

- Double hashing improves over linear or quadratic probing in that $\Theta(m^2)$ probe sequences are used, rather than $\Theta(m)$, since each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence, and as we vary the key, the initial probe position $h_1(k)$ and the offset $h_2(k)$ may vary independently.

- As a result, the performance of double hashing appears to be very close to the performance of the "idea" scheme of uniform hashing.

- The drawback is that we cannot delete items by rehashing, as in linear probing.

- Use a sentinel for deleting an item from the slot.

Open Addressing (Closed Hashing)

Double hashing

- Uses a hashing function of the form

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

  where h_1 and h_2 are auxiliary hash functions.

- The initial position probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$, modulo m. Thus the probe sequence depends in two ways upon the key k, since the initial probe position, the offset, or both, may vary.

- The following figure gives an example of insertion by double hashing.

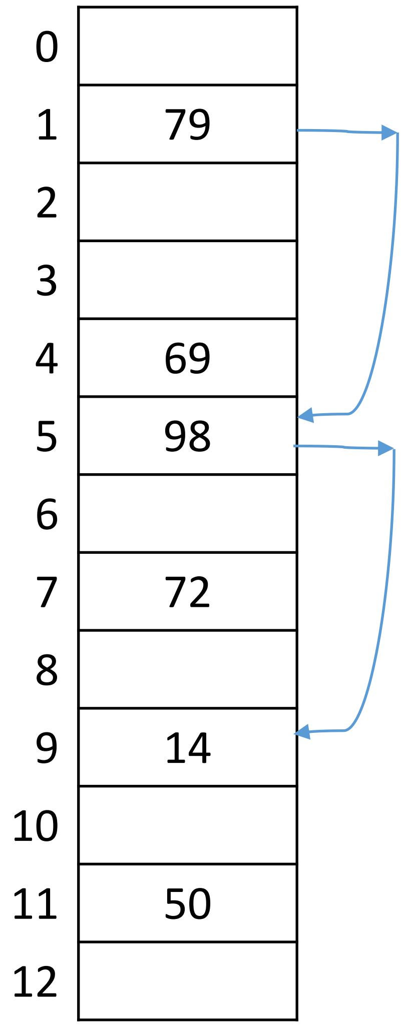


Figure: Insertion by double hashing $h(k, i) = (h_1(k) + i\, h_2(k))$ mod m.
Choose a hash table of size m = 13,
$h_1(k)$ = k mod 13 and $h_2(k)$ = 1 + (k mod 11).

79 ≡ 1 mod 13. 79 ≡ 8 mod 11.
h(79) = (1 + 0*(1+8)) mod 13 = 1.
Insert 79 in slot 1.

98 ≡ 7 mod 13. 98 ≡ 10 mod 11. Then
(7 + 0* (1+10)) mod 13 = 7. 72 is in slot 7.
(7 + 1* (1+10)) mod 13 = 5. Insert 98 in slot 5.

14 ≡ 1 mod 13 and 14 ≡ 3 mod 11.
(1 + 0*(1 + 3)) mod 13 = 1, which has 79.
(1 + 1*(1 + 3)) mod 13 = 5, which has 98.
(1 + 2*(1 + 3)) = 9. Insert the key 14 into empty slot 9, after slots 1 and 5 have been examined and found to be already occupied.

Analysis of Open-address Hashing

- Consider the analysis of open addressing in terms of the load factor α of the hash table, as n and m go to infinity. If n elements (i.e., keys) are stored in a table with m slots, the average number of elements per slot is $\alpha = \frac{n}{m}$.
- With open addressing, each slot has at most one element, and thus, n $\leq$ m implies $\alpha \leq 1$. That is, $\alpha = \frac{n}{m} \leq 1$.
- Assume that uniform hashing is used.
- Ideally, the probe sequence <h(k, 0), h(k, 1), …, h(k, m-1)> for each key k is equally likely to be any permutation of <0, 1, …, m-1>. That is, each possible probe sequence is equally likely to be used as the probe sequence for an insertion or a search.
- A given key has a unique fixed probe sequence associated with it. That is, considering the probability distribution on the space of keys and the operation of the hash function on the keys, each possible probe sequence is equally likely.

Analysis of Double Hashing

- Given an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$.
- When collisions are resolved by double hashing,
    - the expected number of probes in a successful search is at most $\frac{1}{\alpha}\ln(\frac{1}{1-\alpha}) + \frac{1}{\alpha}$, assuming uniform hashing and assuming that each key in the table is equally to be searched for; and
    - and the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.
- It is a big improvement over linear probing and quadratic probing.
- Double hashing allows us to achieve the same performance with a much smaller table.

Analysis of Open-address Hashing

- Let analyze the expected number of probes for hashing with open addressing under the assumption of uniform hashing, beginning with an analysis of the number of probes made in an unsuccessful search.
- Theorem 12.5: Given an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{(1-\alpha)}$, assuming uniform hashing.
- Comment: If α is a constant, Theorem 12.5 predicts that an unsuccessful search runs in O(1) time. For example, if the hash table is half full, the average number of probes in an unsuccessful search is $\frac{1}{(1-.5)} = 2$. If it is 90% full, the average number of probes is $\frac{1}{(1-.9)} = 10$.
- Corollary 12.6: Inserting an element into an open-address hash table with load factor α requires at most $\frac{1}{(1-\alpha)}$ probes on average, assuming uniform hashing.

Analysis of Open-address Hashing

- Computing the expected number of probes for a successful search requires a little more work.
- Theorem 12.7: Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most $\frac{1}{\alpha} \ln \frac{1}{(1-\alpha)} + \frac{1}{\alpha}$, assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.
- Comment: If the hash table is half full, the expected number of probes is less than 3.387. If the hash table is 90% full, the expected number of probes is less than 3.670.

Summary:

A hash function needs to satisfy somewhat conflicting requirements:

- A hash table's size should not be excessively large compared to the number of keys, but not jeopardize the implementation's time efficient.
- A hash function needs to distribute keys among the cells of the hash table as evenly as possible.
- A hash function has to be easy to compute.

The efficiency of searching depends on

- the lengths of the linked lists, which, in turn,
  - depend on the dictionary and table sizes, as well as
- the quality of the hash function.

- If the hash function evenly distributes n keys among m cells of the hash table, each list will be about n/m keys long.
  - The ratio **α** = n/m, called the load factor of the hash table, plays a crucial role in the efficiency of hashing.

- The average number of pointers (chain links) inspected
  - in successful searches, $S \approx 1 + \frac{\alpha}{2}$, and
  - unsuccessful search, $U = \boldsymbol{\alpha}$, (7.4)

  under the standard assumptions of searching for a randomly selected element and a hash function distributed keys uniformly (evenly) among the table's cells.
- They are almost identical to searching sequentially in a linked list;
- What we have gained by hashing is a reduction in average list size by a factor of the size of the hash table m.

The two other dictionary operations – insertion and deletion – are almost identical to searching.

- Insertions are normally done at the end of a list.
- Deletion is performed by searching for a key to be deleted and then removing it from its list.

Hence, the efficiency of these operations is identical to that of searching, and they are all Θ(1) in the average case if the number of keys n is about equal to the hash table's size m.

Closed Hashing (Open Addressing)

In closed hashing,

- all keys are stored in the hash table itself without the use of linked lists.
- This implies that the table size m must be at least as large as the number of keys n).

The mathematical analysis of linear probing is much more difficult problem than that of separate chaining.

For the linear probing approach,

- the average number of times the search algorithm for a key must access the hash table with the load factor **α** in successful and unsuccessful searches is, respectively.

$$S \approx \frac{1}{2}\left(1 + \frac{1}{1 - \propto}\right) \text{ and } U \approx \frac{1}{2}\left(1 + \frac{1}{(1 - \propto)^2}\right) \qquad \ldots (7.5)$$

(and the accuracy of these approximations increases with larger sizes of the hash table).

- These numbers are surprisingly small even for densely populated tables, i.e., for large percentage values of **α**:

$\boldsymbol{\alpha}$	$\frac{1}{2}\left(1+\frac{1}{1-\propto}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\propto)^2}\right)$
50%	**1.5**	**2.5**
75%	**2.5**	**8.5**
90%	**5.5**	**50.5**

It is worthwhile to compare the main properties of hashing with balanced search trees – its principal competitor for implementing dictionaries.

- Asymptotic time efficiency
- Ordering preservation
- Application of Hashing

- Asymptotic time efficiency

  With hashing, searching insertion and deletion can be implemented to take Θ(1) time on the average but Θ(n) time in the very unlikely worst case. For balanced search trees, the average time efficiencies are Θ(log n) for both the average and worst cases.

- Ordering preservation

  Unlike balanced search trees, hashing does not assume existence of key ordering and usually does not preserve it.

  This makes hashing less suitable for applications that need to iterate over the keys in order or require range queries such as counting the number of keys between some lower and upper bounds.

- Application of Hashing
    - Hashing has found by IBM researchers, many important applications. It becomes a standard technique for storing a symbol table – a table of a computer program's symbols generating during compilation.
    - Hashing is quite handy for such AI applications as checking whether positions generated by a chess-playing computer program have already been considered.
    - With some modifications, it has also proved to be useful for storing very large dictionaries on disks; this variation of hashing is called *extendible hashing*. Accordingly, a location computed by a hash function in extendible hashing indicates a disk address of a *bucket* that can hold up to b keys. When a key's bucket is identified all its keys are read into main memory and then searched for the key is question.

End of Chapter 0

☺