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

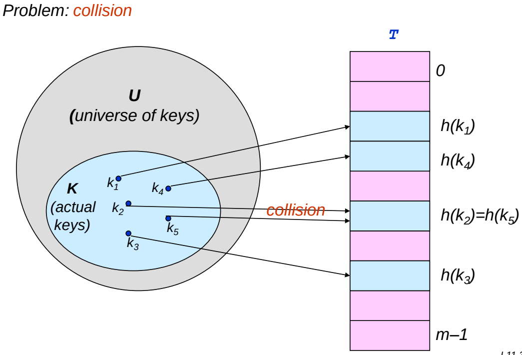
- Hash function h: Mapping from U to the slots of a hash table $T[0..m–1]$.

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

- 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.

Hash Functions

Problem: collision

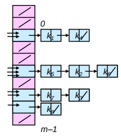


Issues with Hashing

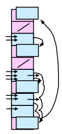
- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations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

Methods of Resolution

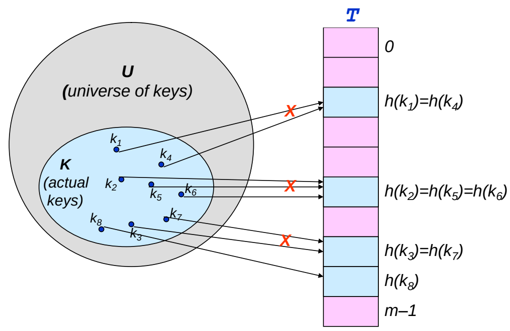
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



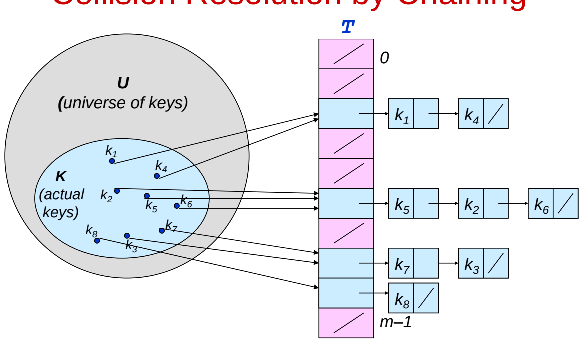
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



Hashing with 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 singly-linked lists. $O(1)$ with doubly-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.

Expected Cost – 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 when lists are doubly linked.
- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

L11.9

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:
    $\sum_{k:h(k) = j} P(k) = 1/m$ for $j = 0, 1, \ldots, m-1$.
  - An example is the division method.

L11.10

Keys as Natural Numbers

- Hash functions assume that the keys are natural numbers.
- When they are not, have to interpret them as natural numbers.
- Example: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, CLRS $= 67\cdot128^3 + 76\cdot128^2 + 82\cdot128^1 + 83\cdot128^0$
    $= 141{,}764{,}947$.

L11.11

Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed three methods:
  - Division method
  - Multiplication method
  - Universal hashing

L11.12

Division Method

- 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 $m=2^p$
  - Or hash won't depend on all bits of k.
- **Good choice for m:**
  - Primes, not too close to power of 2 (or 10) are good.

Multiplication Method

- If $0 < A < 1$, $h(k) = \lfloor m\,(kA \bmod 1)\rfloor = \lfloor m\,(kA - \lfloor kA\rfloor)\rfloor$

  where $kA \bmod 1$ means the fractional part of kA, *i.e.*, $kA - \lfloor kA\rfloor$.
- Disadvantage: Slower than the division method.
- Advantage: Value of m is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.
- Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

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

$= \lfloor 1000 \cdot 0.018169\ldots \rfloor = 18.$

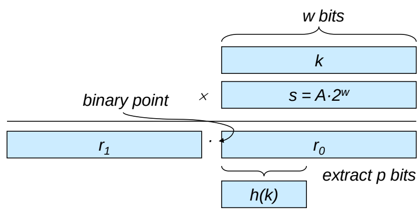
How to choose A?

- How to choose A?
  - The multiplication method works with any legal value of A.
  - But it works better with some values than with others, depending on the keys being hashed.
  - Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

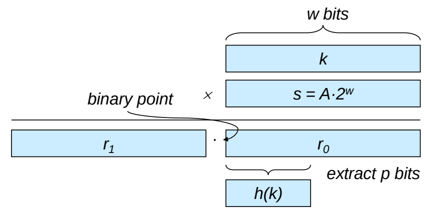
Multiplication Method. – Implementation

- 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.)
- Let $0 < s < 2^w$. (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 ($kA \bmod 1 = kA - \lfloor kA\rfloor$).
- We don't care about the integer part of kA.
  - So, just use r_0, and forget about r_1.

Multiplication Mthd – Implementation



- We want $\lfloor m\,(kA \bmod 1)\rfloor$. We could get that by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.
- But, we don't need to shift. Just take the p most significant bits of r_0.



Example:

- K = 123456, $p = 14$, $m = 2^{14} = 16384$, $w = 32$,
- Adapting Knuth's suggestion, A to be the fraction of the form $s/2^{32}$ that is close to $\approx (\sqrt{5} - 1)/2$, so $A = 2654435769 / 2^{32}$
- K*S = 327706022297664 = $(76300 * 2^{32}) + 17612864$
- $r_1 = 76300$, $r_0 = 17612864$,
- The 14 most significant bits of r_0 yield $h(k) = 67$.

Universal Hashing

- A malicious adversary who has learned the hash function chooses keys that all map to the same slot, giving worst-case behavior.
- Defeat the adversary using 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 $\{0, 1, \ldots, m-1\}$ 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 not in the table T, the expected length of the list that k hashes to is $\leq \alpha$.
- If k is in the table T, the expected length of the list that k hashes to is $\leq 1+\alpha$.

Proof:
$X_{kl} = I\{h(k)=h(l)\}$. $E[X_{kl}] = Pr\{h(k)=h(l)\} \leq 1/m$.
RV Y_k = no. of keys other than k that hash to the same slot as k. Then,

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

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

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

Example of Universal Hashing

When the table size m is a prime,
key x is decomposed into bytes s.t. $x = <x_0, \ldots, x_r>$,
and $a = <a_0, \ldots, a_r>$ denotes a sequence of $r+1$ elements randomly chosen from $\{0, 1, \ldots, m-1\}$,

The class H defined by

$H = \Upsilon_a \{h_a\}$ with $h_a(x) = \sum_{i=0 \text{ to } r} a_i x_i \bmod m$

is a universal function,

(but if some a_i is zero, h does not depend on all bytes of x and if all a_i are zero the behavior is terrible. See text for better method of universal hashing.)

Open Addressing

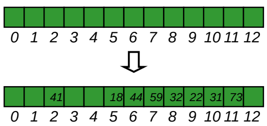
- An alternative to chaining for handling collisions.
- **Idea:**
  - Store all keys in the hash table itself. What can you say about α?
  - Each slot contains either a key or NIL.
  - To ***search*** for key k:
    - Examine slot $h(k)$. Examining a slot is known as a **probe**.
    - If slot $h(k)$ 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.
- **Advantages:** Avoids pointers; so can use a larger table.

Probe Sequence

- 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 extended to:
  - $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$
- $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

Ex: Linear Probing

- Example:
  - $h(x) = x \bmod 13$
  - h(x,i)=(h(x) + i) mod 13
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Operation Insert

- Act as though we were searching, and insert at the first NIL slot found.
- Pseudo-code for **Insert**:

```
Hash-Insert(T, k)
i ← 0
repeat j ← h(k, i)
        if T[j] = NIL
           then T[j] ← k
                return j
           else i ← i + 1
until i = m
error "hash table overflow"
```


Pseudo-code for Search

```
Hash-Search (T, k)
i ← 0
repeat j ← h(k, i)
        if T[j] = k
           then return j
        i ← i + 1
until T[j] = NIL or i = m
return NIL
```


Deletion

- Cannot just turn the slot containing the key we want to delete to contain NIL. Why?
  - We might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied.
- 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. (So, the Hash-Insert need to be modified.)
- **Disadvantage:** Search time is no longer dependent on α.
  - Hence, chaining is more common when keys have to be deleted.

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. (None of these can fulfill the assumption of uniform hashing.)

L11.34

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.
- Easy to implement, but suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
  - Hence, average search and insertion times increase.

L11.35

Ex: Linear Probing

- Example:
  - $\boldsymbol{h'(x)} = \boldsymbol{x} \bmod 13$
  - $h(x)=(h'(x)+i) \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

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

⇩

		41			*18*	*44*	*59*	*32*	*22*	*31*	*73*	
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*

L11.36

Quadratic Probing

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

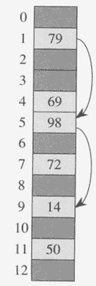
  key Probe number Auxiliary hash function
- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i.
- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same. $h(k_1,0) = h(k_2,0)$

L11.37

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. h_2 gives the remaining probes.
- Must have $h_2(k)$ relatively prime to m, so 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.



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

$m = 13$
$h_1(k) = k \bmod 13$
$h_2(k) = 1 + (k \bmod 11)$

$14 \equiv 1 \pmod{13}$
$14 \equiv 3 \pmod{11}$

Hashing with open addressing tool

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 $\alpha = n/m$.
- **Assumptions:**
  - Assume that the table never completely fills, so $n < m$ and $\alpha < 1$.
  - Assume uniform hashing.
  - No deletion.
  - In a successful search, each key is equally likely to be searched for.

Expected cost of an unsuccessful

Theorem:
Given an open-address hash table with $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$ assuming uniform hashing.

Proof:

Every probe except the last is to an occupied slot.

Let RV X = # of probes in an unsuccessful search.

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

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

$\Pr\{X \geq i\}$

$= \Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}$.

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

Proof – Contd.

$X \ge i$ 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.

$\Pr\{X \ge i\}$

$= \Pr\{A_1 \cap A_2 \cap \ldots \cap A_{i-1}\}$.

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

- $\Pr\{A_j | A_1 \cap A_2 \cap \ldots \cap A_{j-1}\} = (n-j+1)/(m-j+1)$.

$$\Pr\{X \ge i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \Lambda \frac{n-i+2}{m-i+2}$$
$$\le \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}.$$

Proof – Contd.

$$E[X] = \sum_{i=0}^{\infty} i \Pr\{X = i\}$$
$$= \sum_{i=0}^{\infty} i(\Pr\{X \ge i\} - \Pr\{X \ge i+1\})$$
$$= 1 \cdot \Pr\{X \ge 1\} - 1 \cdot \Pr\{X \ge 2\} + 2 \cdot \Pr\{X \ge 2\} - 2 \cdot \Pr\{X \ge 3\} + \Lambda$$
$$= 1 \cdot \Pr\{X \ge 1\} + \Pr\{X \ge 2\} + \Pr\{X \ge 3\} + \Lambda \qquad \textit{(C.25)}$$
$$= \sum_{i=1}^{\infty} \Pr\{X \ge i\}$$
$$\le \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \qquad \textit{(A.6)}$$

- If α is a constant, search takes $O(1)$ time.
- Corollary: Inserting an element into an open-address table takes $\le 1/(1-\alpha)$ probes on average.

Expected cost of a successful

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.
- By the previous corollary, 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.

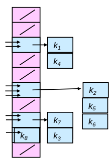
Proof – Contd.

Averaging over all n keys, average # of probes is given by

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}$$
$$= \frac{1}{\alpha}(H_m - H_{m-n})$$
$$\le \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Perfect Hashing

- If you know the n keys in advance (static), make a hash table with $O(n)$ size, and worst-case $O(1)$ lookup time!
- Just use two levels of hashing: A table of size n, then tables of size n_j^2.



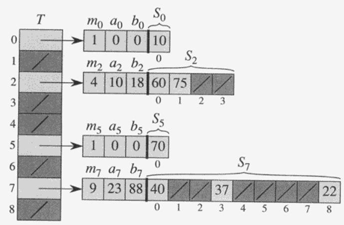


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

End of Chapter 11