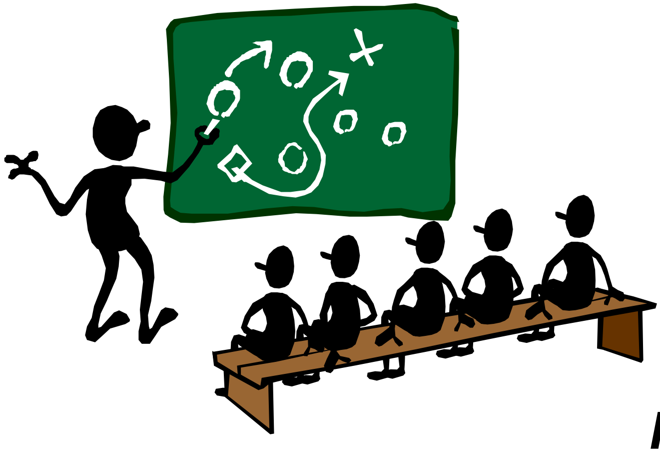
Algorithms – Chapter 11
Hash Tables



Juinn-Dar Huang

Professor

jdhuang@mail.nctu.edu.tw

September 2007

Rev. '08, '11, '12, '15, '16, '18, '19, '20, '21

Introduction

- Many applications require a dynamic set that supports only the dictionary operations – insert, search, delete
  - e.g., compiler
- A hash table is an effective way for this
- Under reasonable assumptions, the **expected** time to insert/search/delete an element in a hash table is **O(1)**
  - Cool! Isn't it?

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Direct-Address Tables

- Direct addressing
  - works well when the universe U of keys is small
  - U = {0, 1, …, m – 1} where U is not large
- Direct-address tables (Arrays)
  - assume no 2 elements have the same keys
  - implemented by an array T[0 .. m – 1]
    - in which each position, **slot**, corresponds to a key in U

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Operations of Direct-Addressing

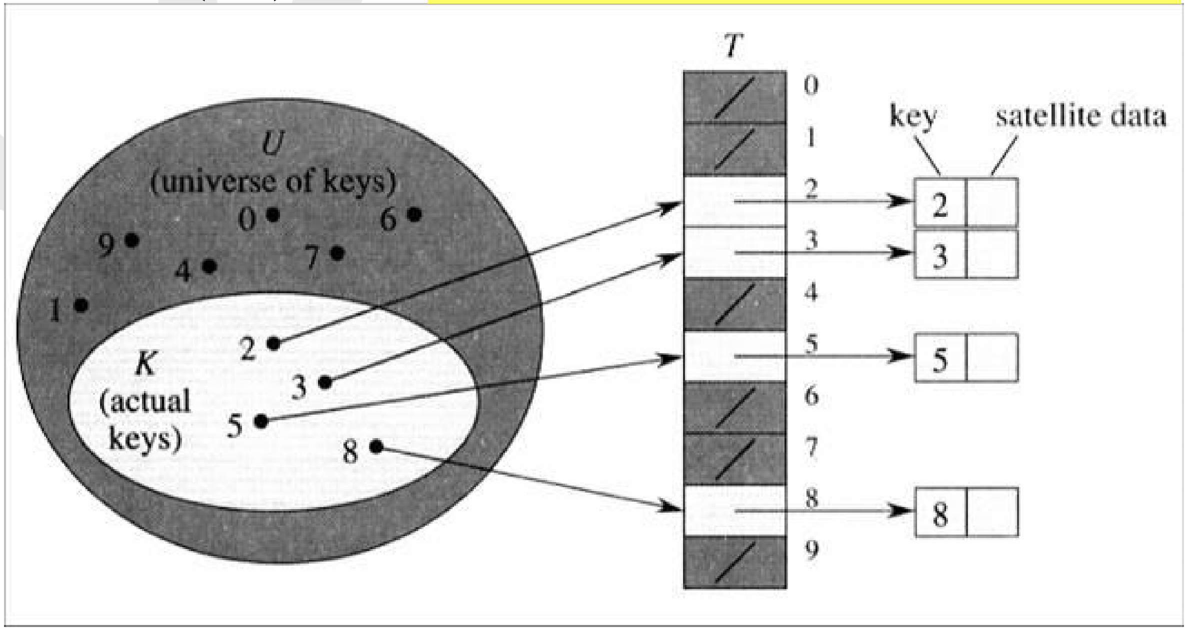
- Actually, operations on an array

DIRECTED_ADDRESS_SEARCH(T,k)
return $T[k]$

DIRECTED_ADDRESS_INSERT(T,x)
$$T[key[x]] \leftarrow x$$

DIRECTED-ADDRESS_DELETE(T,x)
$$T[key[x]] \leftarrow nil$$

O(1) for each operation (array operation in fact)



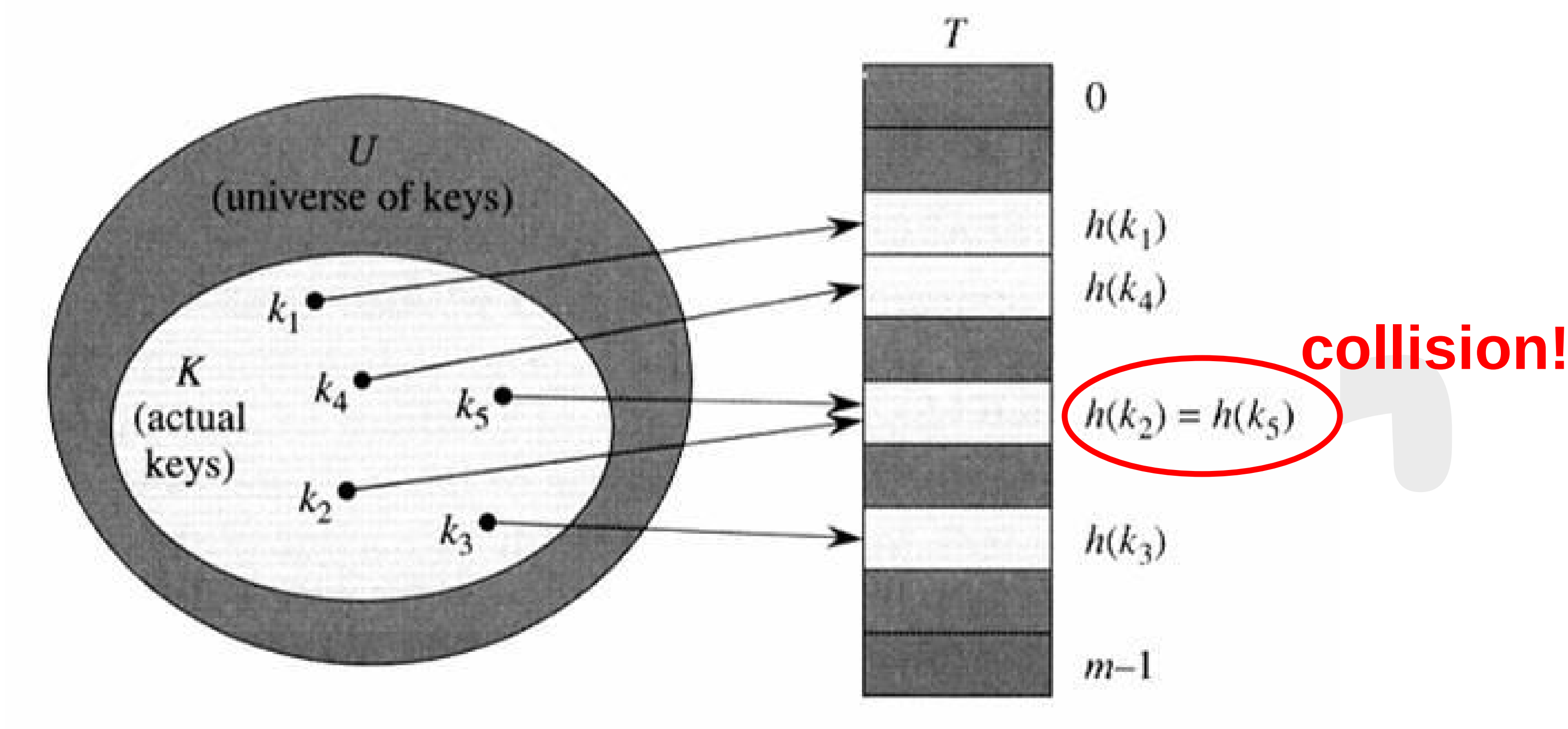
Hash Tables

- However, how if U is large?
  - i.e., a table (array) of size |U| may be impractical, or even impossible
- Moreover, K may be so small relative to U
  - i.e., most of space allocated for T is wasted
  - e.g., identifiers used in a program vs. all valid identifiers
- When |K| << |U|, using a hash table can
  - reduce the space requirement to **Θ(|K|)**
  - keep the search time take still **O(1)** **on average**

Hash Functions

- With direct addressing
  - – an element with key k is stored in slot k
- With hashing
  - – an element with key k is stored in slot h(k)
  - – i.e., a hash function h is used to compute the slot
- A hash function h maps the universe U of keys into the slots of a hash table T[0 .. m – 1]
  - – i.e., $h : U \rightarrow \{ 0, 1, \ldots, m - 1 \}$, where $|U| >> m$
  - – an element with key k hashes to slot h(k)
  - – or, h(k) is the hash value of key k

Illustration



$|U| \gg |T| = m$; and

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

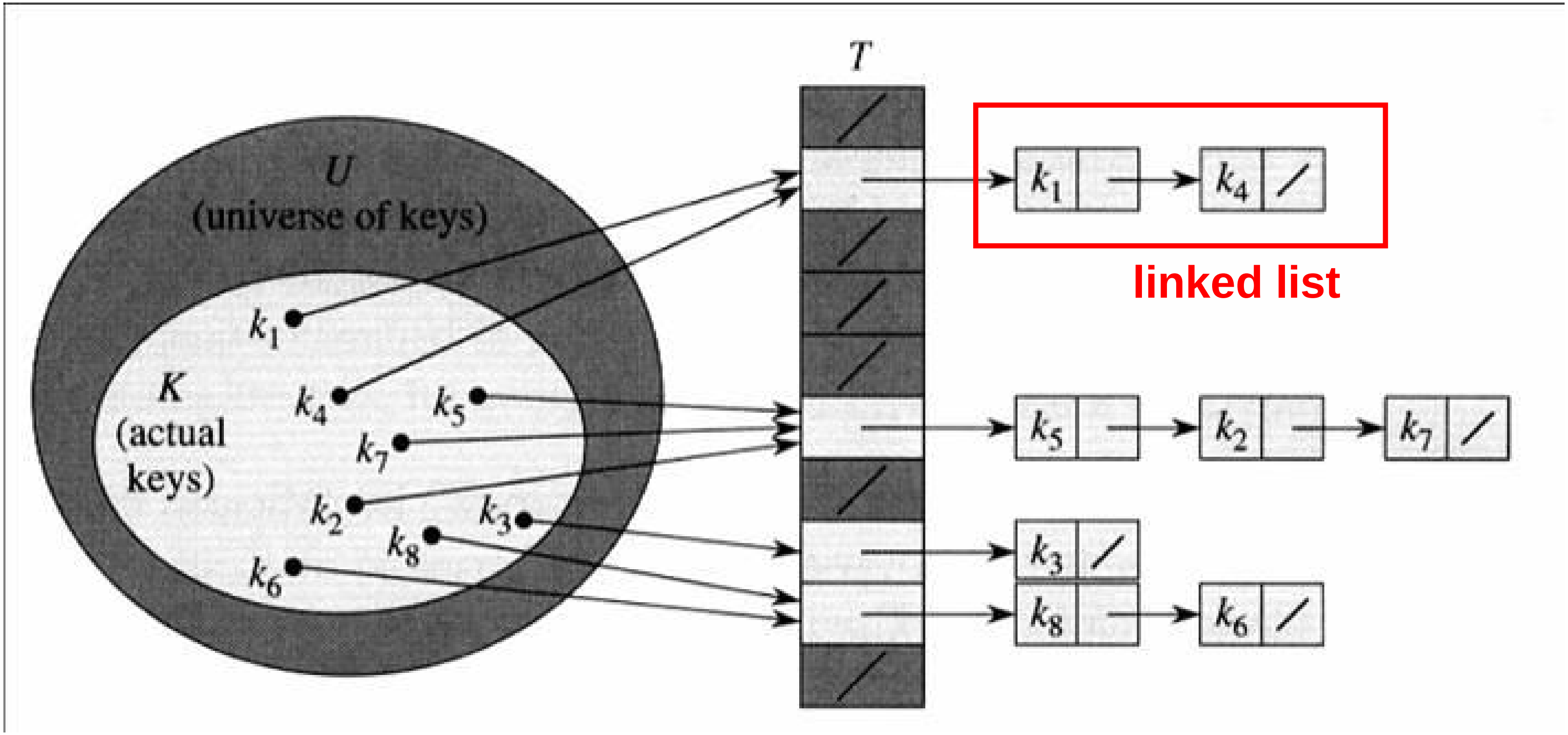
Collisions

- Problem: 2 keys may hash to the same slot
  - ➔ collision

Tactics

- Find a “good” hash function to avoid collisions
  - however, since |U| > m, it’s impossible to absolutely avoid collisions
  - well, then minimize collisions
- Methods to resolve collisions
  - chaining
  - open addressing (discussed later)

Chaining (1/2)


T
U
(universe of keys)
K
(actual
keys)
k1
k4
k5
k7
k2
k3
k8
k6
linked list

Chaining (2/2)

- In chaining, all the elements that hash to the same slot are put in a linked list

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 $T[h(key[x])]$

Time Complexity for Insert/Delete

Time complexity

- INSERT
  - O(1)
- DELETE
  - O(1) if the lists are doubly linked
- How about SEARCH?

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Time Complexity for Search

- Given a hash table T with m slots that stores n elements
  - **load factor α** for T is defined as n/m
  - the average number if elements stored in a slot

- Worst-case time complexity
  - all n keys hash to the same slot ➔ $\Theta(n)$
  - extremely unlikely to happen

Simple Uniform Hashing

- Average-case time complexity
  - the performance depends on how well the hash function distribute the keys
- Assumptions of simple uniform hashing
  - any element is equally likely to hash into any of the m slots
  - the hashing result is independent of where any other element has hashed to

Average-Case Time Complexity

- For $j = 0, 1, \ldots, m-1$, the length of the list $T[j]$ is denoted by n_j
  - $n = n_0 + n_1 + \ldots + n_{m-1}$
- The average value of n_j is $E[n_j] = \alpha = n/m$
- Assume h(k) can be computed in O(1)

- Two cases for a search
  - unsuccessful search (key not found)
  - successful search (key found)

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Unsuccessful Search

- In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing
  - to compute h(k) ➔ $\Theta(1)$
  - to search to the end of T[h(k)] ➔ $E[n_{h(k)}] = \alpha$

if n = O(m) then
$\Theta(1+\alpha)$ ➔ $\Theta(1)$

Successful Search

- In a hash table in which collisions are resolved by chaining, a successful search takes expected time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing
  - assume the key being searched is equally likely to be any of the *n* keys stored in the table
  - to find x, # of elements examined is (1 + # of elements appear before x in x's list)
  - elements before x in the list are inserted after x is inserted

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Time Complexity Analysis (1/2)

- Let x_i denote the ith element inserted into the table
- Let k_i = key[x_i]
- Define the random variable X_{ij} = I{ $h(k_i) = h(k_j)$ }, $i < j$
  - in simple uniform hashing, $Pr\{ h(k_i) = h(k_j) \} = 1/m \Rightarrow E[X_{ij}] = 1/m$

Time Complexity Analysis (2/2)

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

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

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

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

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

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

$$\Theta\left(2+\frac{\alpha}{2}-\frac{\alpha}{2n}\right)=\Theta(1+\alpha)$$

if n = O(m) then

$\Theta(1+\alpha) \rightarrow \Theta(1)$

Hash Tables

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Hash Functions

- What makes a good hash function?
  - satisfy (approximately) the assumption of simple uniform hashing
- If the distribution is known
  - e.g., keys are random real numbers k independently and uniformly distributed in the range $0 \le k < 1$
  - a hash function can be easily obtained; e.g., $h(k) = \lfloor km \rfloor$

Keys as Natural Numbers

- Most hash functions assume that the universe of keys is N
- If keys are not natural numbers
  ➔ need a mapping method
- Example
- the ASCII string “pt” can be interpreted as 112*128+116 = 14452

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Hash Function – Division

- Map a key k into one of m slots by taking the remainder of k divided by m
- That is, the hash function is defined as
  - $h(k) = k \bmod m$
- Avoid certain values of m
  - e.g., m should not be a power of 2
  - since if $m = 2^p$, $h(k)$ is just the p lowest-order bits of k
- It's better to make hash function depend on all the bits of the key!
- A prime not too close to 2^p is often a good choice for m

Hash Function – Multiplication (1/2)

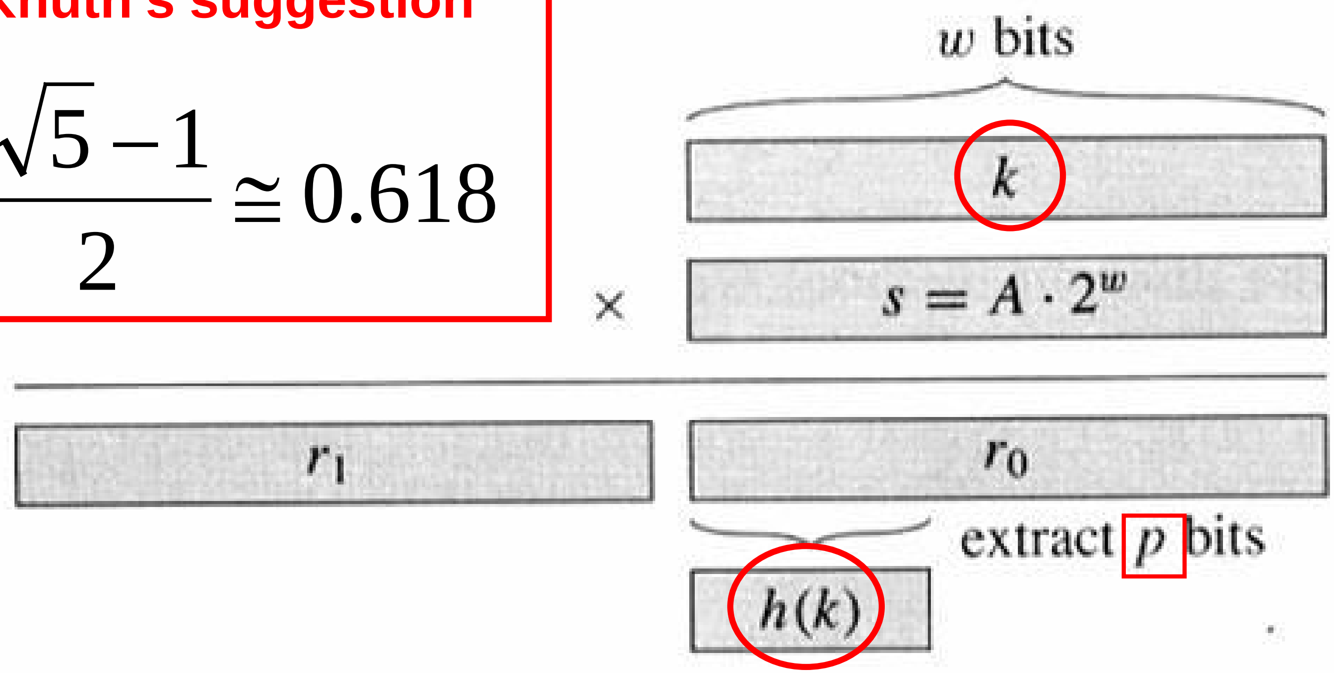
- Multiplication method
  - multiply the key k by a constant A, where $0 < A < 1$
  - extract the fractional part of kA ➔ f, where $0 \le f < 1$
    - i.e., $f = kA - \lfloor kA \rfloor$
  - $h(k) = \lfloor fm \rfloor$
- An advantage of this method is
  - the value of m is not critical
    - typically, m is selected as a 2^p
  - multiplication vs. division

Hash Function – Multiplication (2/2)

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Knuth's suggestion

$$m = 2^{\boxed{p}}, A = \frac{\sqrt{5}-1}{2} \cong 0.618$$



Open Addressing

- Open addressing
  - all elements are stored in the hash table itself
  - each entry contains either an element or NIL
  - no elements are stored outside the table (not like chaining)
  - the load factor α can never exceed 1

- Instead of following pointers (in chaining), we compute the sequence of slots to be examined
  - another way to resolve collisions

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Element Insertion

- To insert an element
  - the hash table is successively probed until an empty slot is found
- Probing in a fixed order starting with 0
  - actually append an element ➔ $\Theta(n)$ time
- Instead, the sequence of positions probed depends on the key
  - $h : U \times \{ 0, 1, \ldots, m-1 \} \rightarrow \{ 0, 1, \ldots, m-1 \}$
  - for every key k, the probe sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ MUST be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Insertion Procedure

Hash-Insert(T, k)

$i \leftarrow 0$

repeat $\boxed{j \leftarrow h(k, i)}$

if $T[j] = \text{NIL}$

then $T[j] \leftarrow k$

return j

else $i \leftarrow i + 1$

until $i = m$

error “hash table overflow”

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Search Procedure

Hash-Search(T, k)
$i \leftarrow 0$
repeat $\boxed{j \leftarrow h(k, i)}$
    if $T[j] = k$
        then return j
    $i \leftarrow i + 1$
until $T[j] = \text{NIL}$ or $i = m$
return NIL

Question: How about deletion?

Probing Methods

- How to generate a probe sequence?
  - i.e., how to implement
    $h : U \times \{ 0, 1, \ldots, m–1 \} \rightarrow \{ 0, 1, \ldots, m–1 \}$?

- Methods
  - linear probing
  - quadratic probing
  - double hashing

Linear Probing

- Given an ordinary hash function
  - $h' : U \to \{ 0, 1, \ldots, m-1 \}$
  - referred as auxiliary hash function
- Linear probing

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

- Drawbacks
  - only m distinct probe sequences
  - primary clustering

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Quadratic Probing

- Quadratic probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, c_2 \neq 0$$

  - c_1, c_2 and m should be carefully selected
  - one good way: $c_1 = c_2 = 0.5$, m is a power of 2 (Check Problem 11-3)
- No primary clustering issue
- However
  - only m distinct probe sequences as well
  - secondary clustering
    - if $h(k_1, 0) = h(k_2, 0)$, k_1 and k_2 have the same probe sequence

Double Hashing

- Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

- h_2 must be relatively prime to m
  - e.g., let m is a power of 2 and h_2 always produce odd numbers
- Relax clustering problem
  - m^2 distinct probe sequences
  - k_1 and k_2 have the same probe sequence only if $(h_1(k_1), h_2(k_1)) = (h_1(k_2), h_2(k_2))$
  - better than linear and quadratic probing

Analysis – Unsuccessful Search (1/2)

- Uniform hashing
  - each key is equally likely to have any of the m! permutations of < 0, 1, …, m–1 > as its probe sequence
- Given an open-addressing hash table with load facor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most **$1/(1 - \alpha)$**
  - uniform hashing is assumed
- Definition
  - random variable X as the number of probes in an unsuccessful search

Analysis – Unsuccessful Search (2/2)

$$\Pr\{X \geq 1\} = 1$$

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1} \quad \text{, for } 2 \leq i \leq n+1$$

$$\Pr\{X \geq i\} = 0 \quad \text{, for } i > n+1$$

$$E[X] = \boxed{\sum_{i=1}^{n+1} i \cdot \Pr\{X = i\} = \sum_{i=1}^{n+1} \Pr\{X \geq i\}}$$

$$\leq \sum_{i=1}^{n+1} \alpha^{i-1} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i}$$

$$= \boxed{\frac{1}{1-\alpha}}$$

Analysis – Element Insertion

- Inserting a key
  - requires an unsuccessful search first
  - places the key into the first empty slot found
- On average, at most $1/(1 - \alpha)$ probes are expected

Juinn-Dar Huang
jdhuang@mail.nctu.edu.tw

Analysis – Successful Search (1/2)

- A search for a key k follows the same probe sequence as was followed when k was inserted
- If k was the (i+1)th key inserted into the hash table, the expected number of probes for k is at most
  - $1/(1 - \alpha) = 1/(1 - i/m) = m/(m - i)$

Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Analysis – Successful Search (2/2)

- Average number of probes in a successful search

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

$$\leq \frac{1}{\alpha}(\ln m - \ln(m-n))$$

$$= \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$