Announcements

- Weekly reading assignment:

  Chapter 12 & 13, CLRS

- Homework #4 Due on October 27

- Extra Help Sessions after class today and Thursday, 10/27/05

Dictionary Operations

Direct-Address-Search (T, k)

return $T[k]$

Direct-Address-Insert (T, x)

$T[key[x]] \leftarrow x$

Direct-Address-Delete (T, x)

$T[key[x]] \leftarrow NIL$

*** Each takes *O(1)* time in the worst case.**

Hash Tables

- An effective data structure for dictionaries
- Search an element takes $\Theta(n)$ time in the worst case and $O(1)$ time on average (direct addressing takes $O(1)$ time in the worst case)
- When the total number of keys stored are much less than total possible, it is better than direct-address tables (arrays)

Hashing

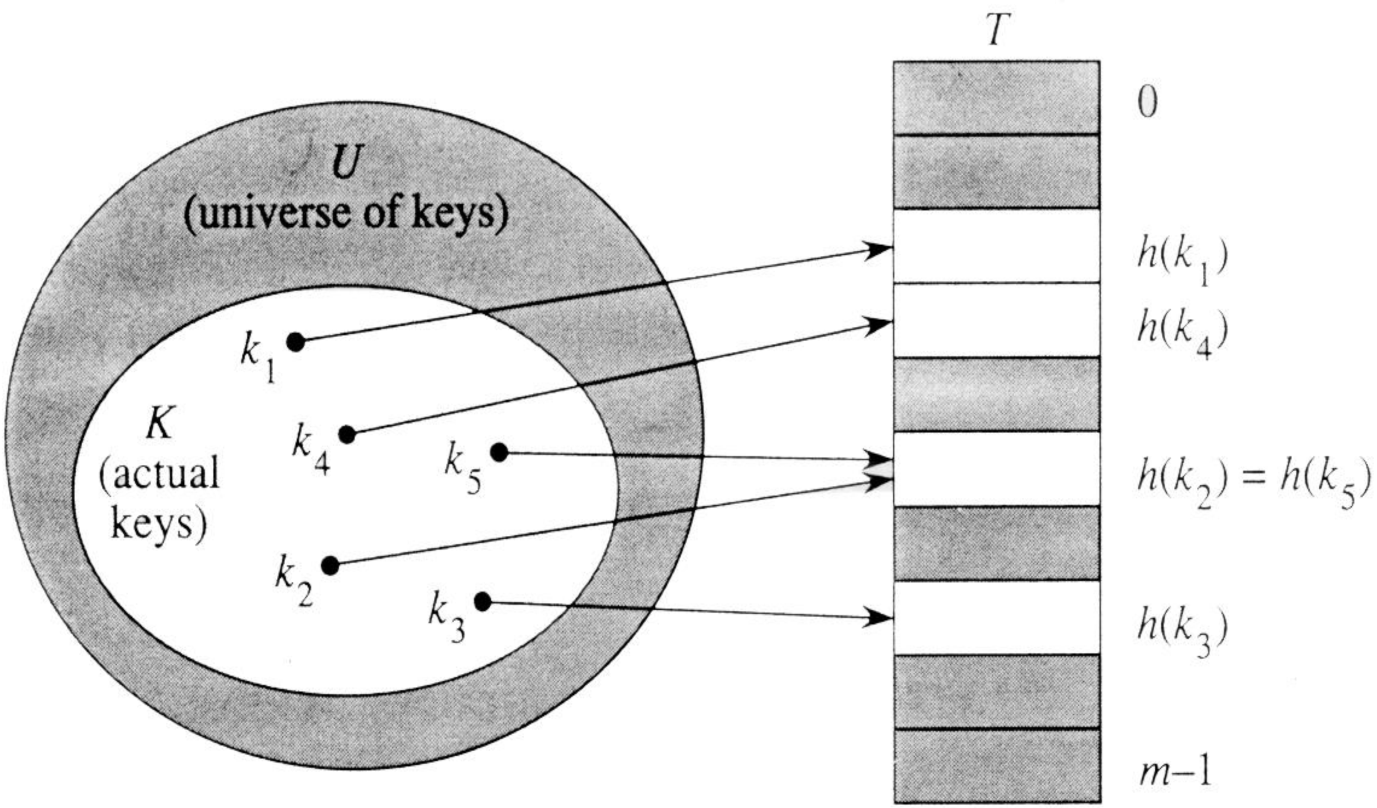

Figure 12.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

Definitions

- With hashing, the element is stored in slot $h(k)$; a "*hashing function*" h is used to compute the slot from the key k. Here h maps the universe U of keys into slots of a "*hash table*" $T[0,...m\text{-}1]$:

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

- An element with key k "*hashes*" to slot $h(k)$. $h(k)$ is the "*hash value*" of key k.
- *Collision*: two keys hash into the same slot

Hash Tables

- Great for finding things quickly
- Bad for finding minimum, maximum, etc. (priority queue is better for this)
- Issues:
  – Collision resolution
  – Design of hashing functions

Methods of Resolution

- Open Addressing
- Chaining: put all elements that hash to the same slot in a linked list.

Collision Resolution by Chaining

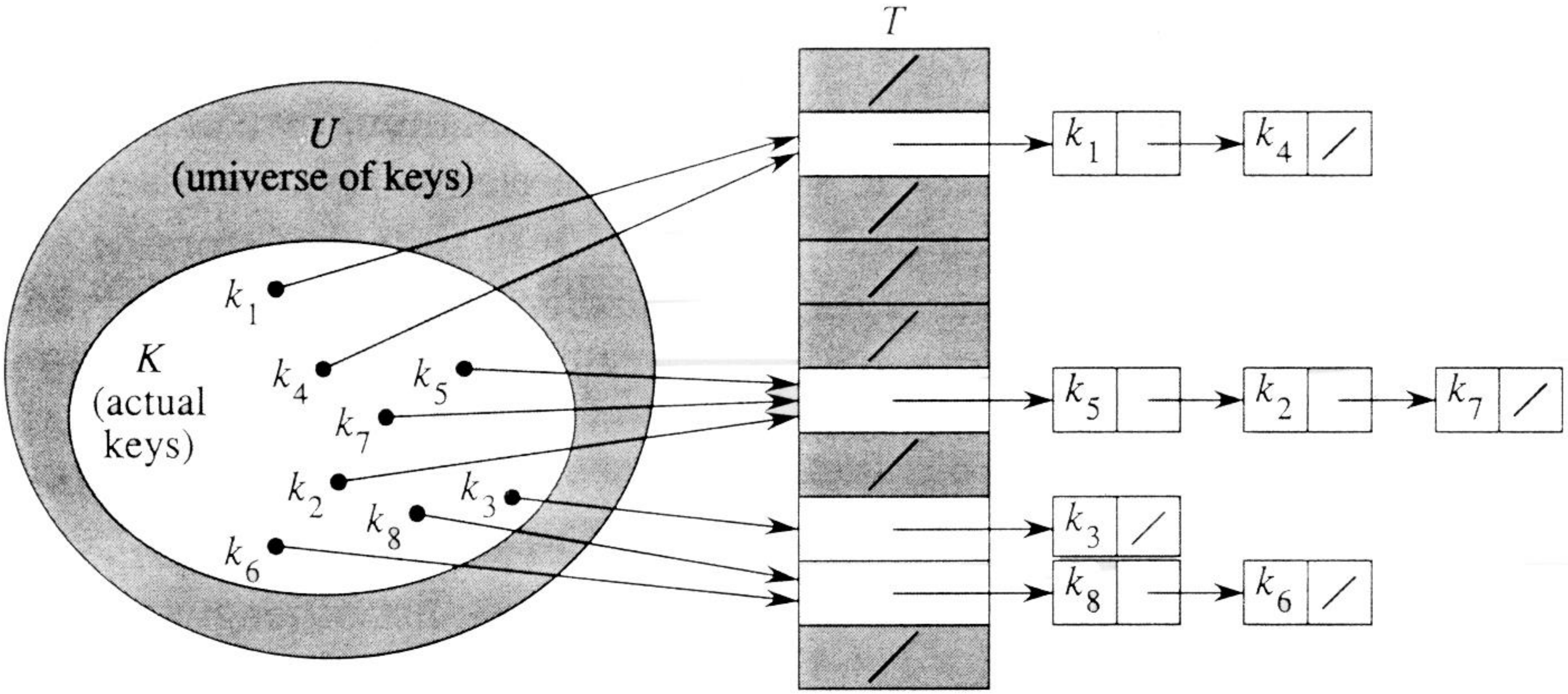

Figure 12.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Hashing with Chaining

Chained-Hash-Search (T, k)

search an element with key k in list $T[h(k)]$

Chained-Hash-Insert (T, x)

insert x at the head of list $T[h(key[x])]$

Chained-Hash-Delete (T, x)

delete x from the list $T[h(key[x])]$

*** Insertion takes $O(1)$ time in the worst case; deletion can also take $O(1)$ time. Searching's worst-case time is proportional to length of T.**

Analysis on Chained-Hash-Search

- *Load factor* $\alpha = n/m$ = average keys per slot, for m slots to store n elements
- Worst case: $\Theta(n)$ + time to compute $h(k)$
- Average case depends on how well h distributes the keys among m slots.
- Assume *simple uniform hashing* and $O(1)$ time to compute $h(k)$, time required to search an element with key k depends linearly on length of $T[h(k)]$.
- *Consider expected number of elements examined by search algorithm, i.e. the number of elements in* $T[h(k)]$ *that are checked to see if their keys* = k.

Costs of Search

- **Cost of an unsuccessful search** = $1+\alpha = \Theta(1+\alpha)$
  - 1 to access the slot & α expected time to search the list
- **Cost of a successful search** = $1+ \alpha/2 = \Theta(1+\alpha)$
  - 1 to access the slot & $\alpha/2$ expected time to search the list

Worst Case Time for Chained Hash Search

- **If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$. Searching takes constant time on average. Thus, all dictionary operations can be supported in $O(1)$ time on average.**

Good Hash Functions

- Satisfies (approximately) the assumption of *simple uniform hashing*, i.e. each element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.
- Regularity in key distribution should not affect uniformity.

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

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

An example is division method.

Division Method

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

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

- Don't pick certain values, such as $m = 2^p$

  Or hash won't depend on all bits of k.

- 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 fraction part of kA

- The value of m is not critical, typically a power of 2, $m = 2^p$ (A common implementation on most computers is given in Figure 12.4.)

For example, $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

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

$$= \lfloor 1000 * 0.018169\ldots \rfloor = 18$$

Multiplication Method

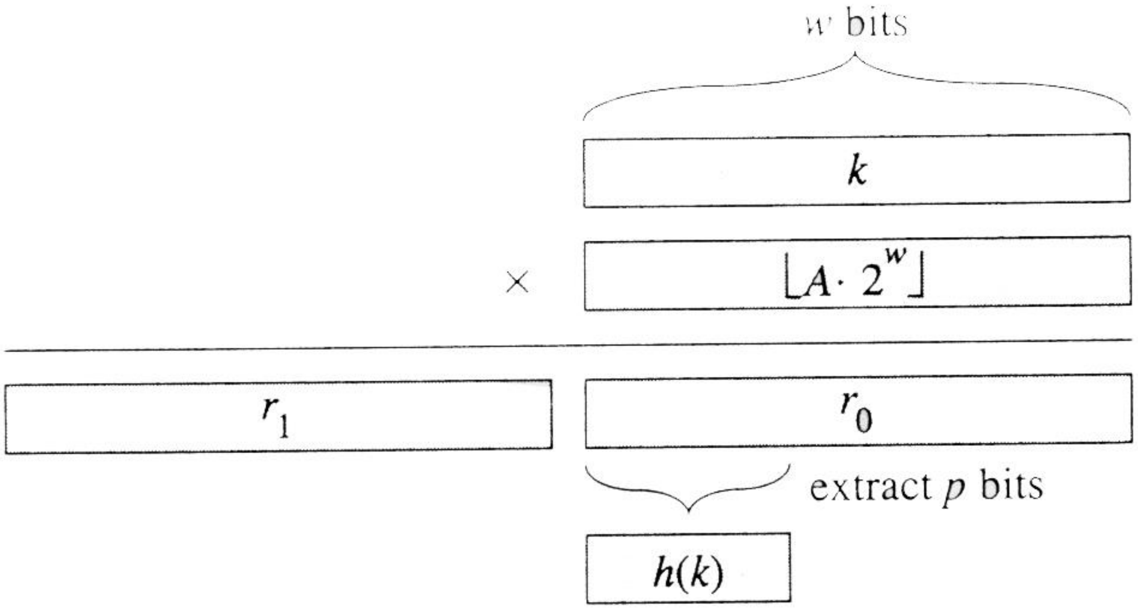

Figure 12.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $\lfloor A \cdot 2^w \rfloor$, where $0 < A < 1$ is a suitable constant. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

Another Example

- **Assume** $m = 8 = 2^3$**, and 7-bit words**
- **Think of the machine integer A as having a radix point on the left**
- $h(k)$ **takes fractional part of product, discard rest, shifts fractional part left and takes the shifted-out bits.**
- **Example: see one on the board**

Universal Hashing

- Choose the hashing function randomly s.t. it is independent of keys that will actually be stored.
- Select the hash function at random at run time from a carefully designed class of functions.
- Let H be a collection of hash functions that map a universe U of keys into the range $\{0, 1, \ldots, m-1\}$. H is said to be "*universal*", if for each pair of distinct keys, $x, y \in U$, number of hash functions for which $h(x)=h(y)$ is precisely $|H|/m$. The chance of a collision between 2 keys is exactly $1/m$.

Theorem

- **If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m, where $n \le m$, the expected number of collisions involving a particular key x is less than 1.**

(Proof) Let C_{yz} be a random variable that = 1, if $h(y) = h(z)$ and 0 otherwise. $E[C_{yz}] = 1/m$.

Let C_x be the total number of collisions involving key x in a hash table T of size m. Then,

$$E[C_x] = \sum_{y \in T\, y \neq x} E[C_{xy}] = (n-1)/m < \alpha.$$

Example of Universal Hashing

- The class H defined by

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

is a universal function, where the table size m is a prime, the key x is decomposed into bytes s.t. $x = \langle x_0, \ldots, x_r \rangle$ and $a = \langle a_0, \ldots, a_r \rangle$ denote a sequence of $r+1$ elements randomly chosen from the set $\{0, 1, \ldots, m-1\}$.

Open Addressing

- **All elements are stored in the hash table itself. Each table entry contains either an element of the dynamic set or NIL.**
- **There are no list/elements outside of table. The hash table fills up s.t. no further insertion can be made. The load factor never exceeds 1.**
- **To perform insertion, successively examine or *probe* the hash table till an empty slot is found. The sequence of probed positions depends on the key being inserted. We require every key k, the probe sequence $\langle h(k,0), h(k,1), \ldots, h(k, m-1)\rangle$ be a permutation of $\langle 0, 1, \ldots, m-1\rangle$.**

Hash-Insert (T, k)

$i \leftarrow 0$
repeat $j \leftarrow h(k, i)$
    if $T[j]$ = NIL
        then $T[j] \leftarrow k$
            return j
        else $i \leftarrow i + 1$
until $i = m$
Error “hash table overflow”

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
```

Linear Probing

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

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

  The initial probe position is $T[h'(k)]$, the next is $T[h'(k)+1]$, and so on upto $T[m]$, then wrap around.

- **Only m distinct probe sequences are used. Easy to implement, but suffers *"primary clustering"*. Long runs of occupied slots build up and increases the search time.**

Quadratic Probing

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

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

  The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depends on a quadratic function of the probe number i.

- 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 ***"secondary clustering"***.

Double Hashing

- h_1 and h_2 are auxiliary hash functions and the initial positions probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by amount $h_2(k)$, modulo m. The function has the form:

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

- The probe sequences depends two ways upon the key k, since both position and offset or both may vary. The value of $h_2(k)$ must be relatively prime to m. E.g., let m be a power of 2 and h_2 <u>be chosen always to return an odd integer</u>.