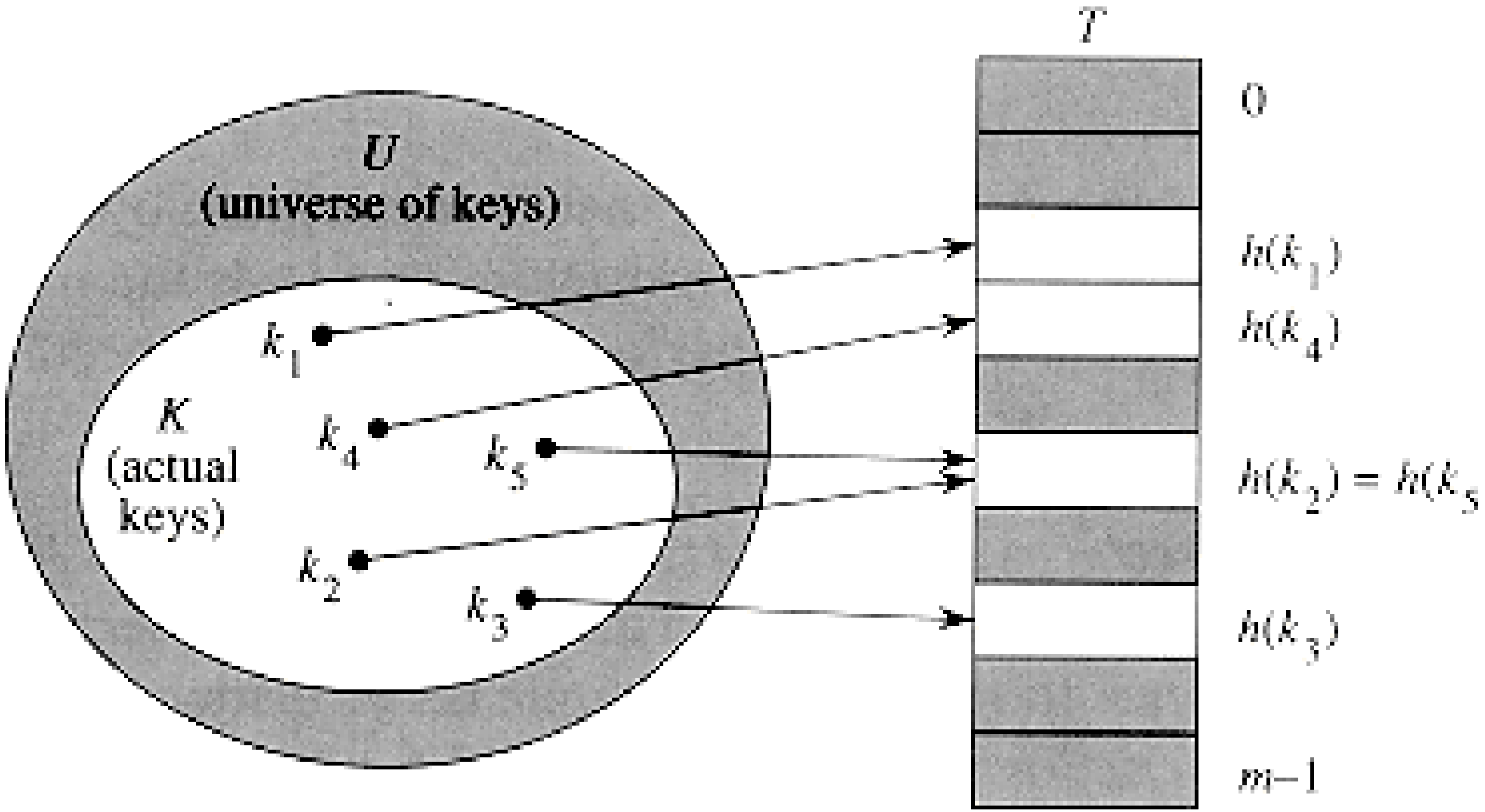
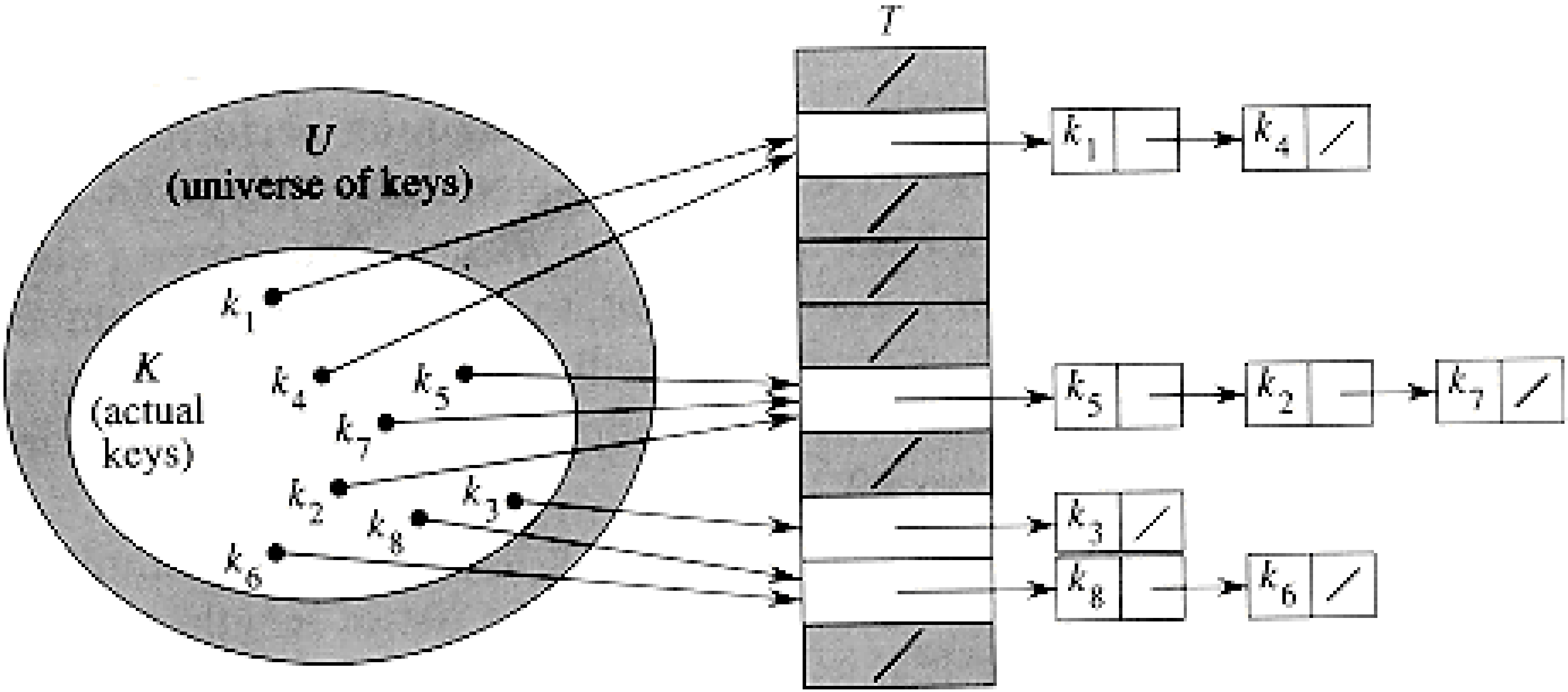
# Hash table



T
0
U
(universe of keys)
h(k1)
h(k4)
K
(actual
keys)
k1
k4
k5
k2
k3
h(k2) = h(k5)
h(k3)
m–1

# Collision resolution by chaining



# Collision resolution by chaining

operations on a hash table $T$

insert($T$,$x$)

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

search($T$,$k$)

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

delete($T$,$x$)

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

Running time

| | |
|---|---|
| insert : | $O(1)$ |
| search: | proportional to the length of the list |
| delete: | ( ***if the lists are singly linked*** ) |
| | proportional to the length of the list |

***if the lists are doubly linked and***
***when we know position:*** $O(1)$

# Open addressing

store the records directly within the array

**probing**: search through alternate locations in the array (the probe sequence)

**Collisions** (solutions)

- linear probing

  the interval between probes is fixed - often at 1.
- quadratic probing

  the interval between probes increases proportional to the hash value (the interval increase linearly)
- double hashing

  the interval between probes is computed by another hash function

# Open addressing

## Formal:

hash function is defined as follows:

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

the ***probe sequence***

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

*important:* acces every hash-table position

Assume that *(for the next examples)*

- each entry contains either a key or $\perp$

# Open addressing: linear probing

Given hash function $h'$: $U \rightarrow \{0, 1, \ldots, m - 1\}$

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

Slot probed: $\boldsymbol{T[h'(k)]}$, $T[h'(k) + 1]$, … $T[m - 1]$,
$T[0]$, $T[1]$, . . . , until $T[h'(k) - 1]$.

Problem : ***primary clustering***

long runs of occupied slots build up, increasing the average search time.

**Example**

Consider keys: 53, 151, 54, 55, 56

illustrate their positioning in an initially empty hash table,

when m = 97 and $h'(k)=k \bmod m$

# Open addressing: quadratic probing

Given hash function $h'$: $U \rightarrow \{0, 1, . . . , m - 1\}$,

$$\boldsymbol{h(k,i) = (h'(k) + c1*i + c2*i^{2}) \bmod m}$$

$c1$ and $c2$ <> 0 are auxiliary constants,

and $i = 0, 1, . . . , m - 1$.

Problem: ***secondary clustering***

if two keys have the same initial probe position,

then their probe sequences are the same:

$$h(k1, 0) = h(k2, 0) \Rightarrow h(k1, i) = h(k2, i).$$

# Open addressing: double hashing

Given hash functions

$h1, h2: U \rightarrow \{0, 1, . . . , m - 1\},$

$$\boldsymbol{h(k,i) = (h1(k) + i*h2(k)) \bmod m}$$

- *h*1 and *h*2 - auxiliary hash func.

Remark:

one of the best methods for open addressing

# Open addressing: double hashing

**Choosing h1 and h2**

if $m$ and $h2(k)$ have greatest common divisor $d > 1$ for some key $k$, then a search for key $k$ would examine only $(1/d)$th of the hash table.

$h2(k)$ - relatively prime to the hash-table size $m$

Convenient ways to ensure this condition:

- $m$ be a power of 2
  design $h2$ so that it always produces an odd number
- let $m$ be prime
  and design $h2$ so that it always returns a positive integer less than $m$.

Example:

choose $m$ prime

$h1(k) = k \bmod m$ ,

$h2(k) = 1 + (k \bmod m')$,

where $m'$ slightly less than $m$ (say, $m - 1$ or $m - 2$).

# Open addressing

**Write subalg. for search, insert, delete .**

**Delete**

move the data

mark position with a special value DELETED

- modify SEARCH

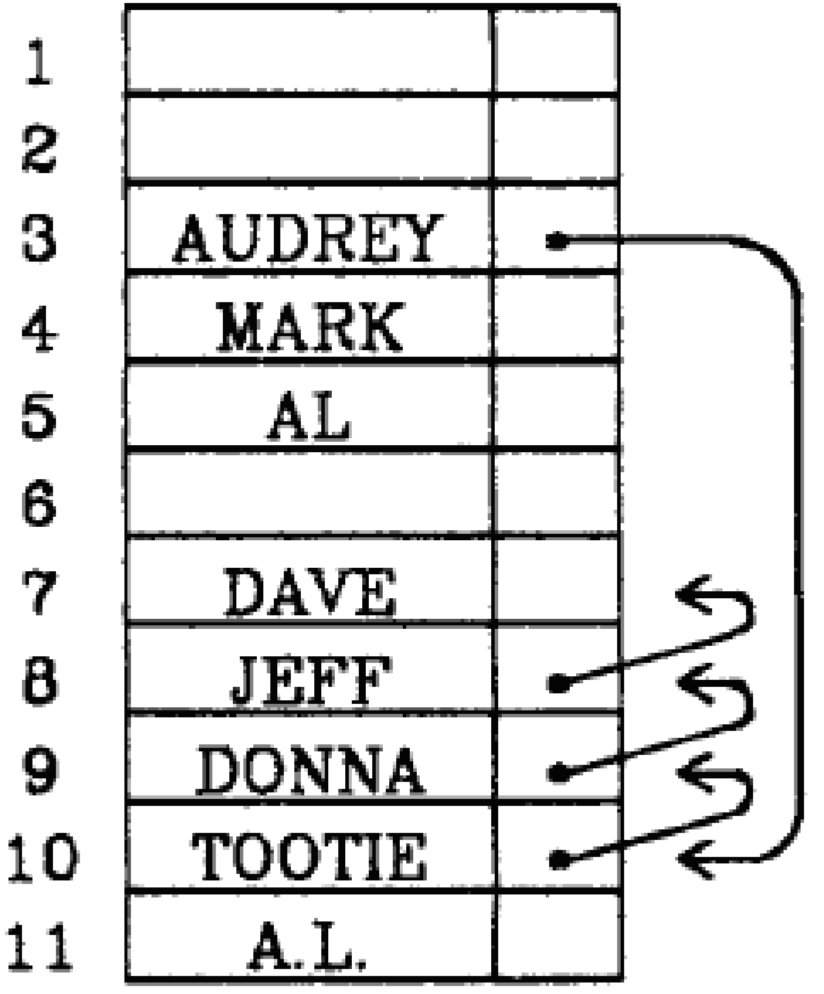
  so that it keeps on looking when it sees the value DELETED,

- modify INSERT

  would treat DELETED slot as if it were empty (a new key can be inserted)

# Coalesced hashing

| | h1 |
|---|---|
| A.L. | 11 |
| AUDREY | 3 |
| AL | 5 |
| TOOTIE | 3 |
| DONNA | 10 |
| MARK | 4 |
| JEFF | 10 |
| DAVE | 9 |



# Hash function

- the universe of keys is a subset of

  $N = \{0,1,2, . . .\}$
- if the keys are not natural numbers - interpret them as natural numbers

Example:

a character string

consider successive ASCII codes

# Method for Creating Hash Function

maps the universe *U* of keys

into the slots of a ***hash table*** *T*

1. *The division method.*
2. *The multiplication method.*
3. *Universal hashing.*

# Building hash function: division method

$h(k) = k \bmod m$

**0-based arrays**

experiments =>

good values for $m$ are

prime not too close to exact powers of 2

# Building hash function: multiplication method

**The multiplication method**

h(k) = floor( m * frac(k * A))

where

m - hash table size

A - constant in the range 0< A <1

Remark:

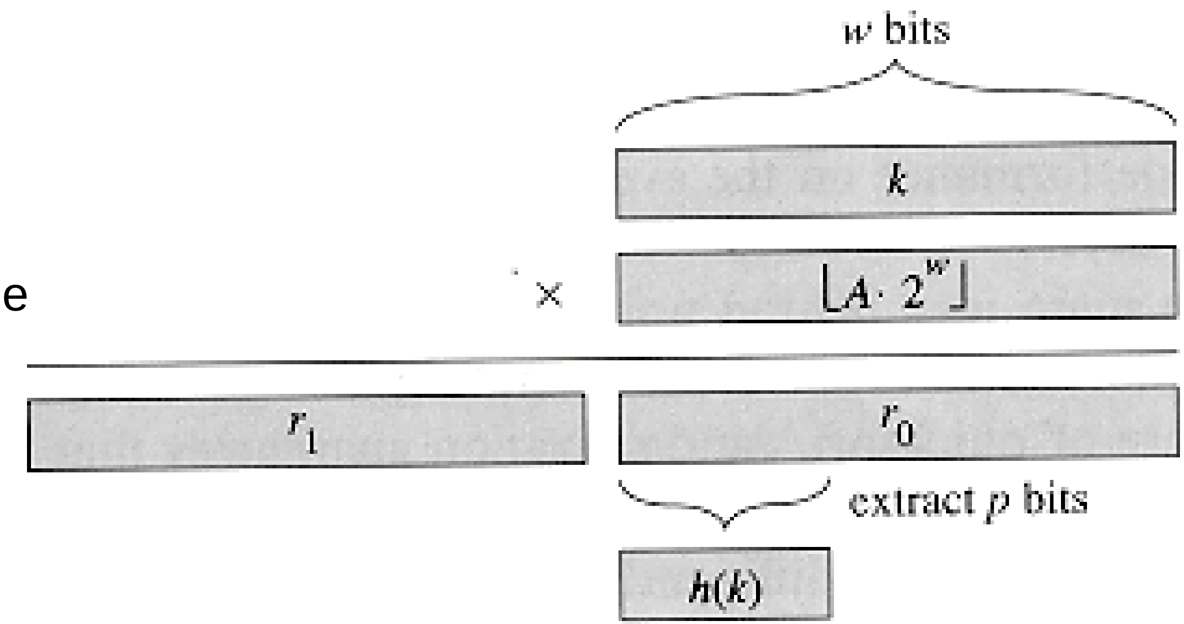
• the value of *m* is not critical

Easy implementation:

restrict $A = s/2^w$,

*w*=machine word size

$m = 2^p$ for some integer $p$



# Building hash function: multiplication method

**The multiplication method**

***good value for A (experimental)***

$$A \approx \frac{\sqrt{5}-1}{2} \approx 0.6180339887$$

**Donald Knuth**, *The Art of Computer Programming* , **1968**

Numeric example:

$k$ = 123456

$m$ = 10000

$A$ = 0.6180339887

$h(k)$ = floor(41.151…) = 41

$k$ = 50

$h(k)$ = 9016

# Hash function

| | Multiplication Method | Division Method |
|---|---|---|
| m | 1000 | 1000 |
| A | 0.618033988749895 | |
| *key* | *h(key)= floor(m * frac(key * A))* | *h(key) = key mod* 1000 |
| 123456 | 4 | 456 |
| 123459 | 858 | 459 |
| 123496 | 725 | 496 |
| 123956 | 21 | 956 |
| 129456 | 208 | 456 |
| 193456 | 383 | 456 |
| 923456 | 195 | 456 |



**the value of *m* is not critical**

# Building hash function: universal hashing

**Universal hashing**: refers to selecting a hash function at random from a family of hash func. with a certain property

**universal class of hash functions**

Let H be a finite collection of

hash functions that map : U ➔ {0,1, . . . , m - 1}

Such a collection is said to be **universal**

if for each pair of distinct keys x, y $\in$ U,

the nr. of hash functions for which h(x) = h(y) is at most **|H|/m**

---

With a function h chosen uniformly at random from H, the chance of a collision between x and y, where x <> y, is less than 1/m.

$$P(h(x) = h(y)) \leq \frac{1}{m}$$

# Building hash function: universal hashing

Example:

m – the size of hash table, prime

key x: decompose a key x into r *bytes*

$$x = \langle x_1, x_2, \ldots, x_r \rangle$$

with: $x_i <= m$

**hash function**:

$$h_a(x) = \sum_{i=1}^{r} a_i * x_i \mod m$$

$\langle a_1, a_2, \ldots, a_r \rangle$ is a fixed sequence of random numbers

$a_i \in \{0, \ldots, m-1\}$

**universal class of hash functions**

$$H = \bigcup_a h_a$$

union taken over all possible ***a***-s

- $m^r$ members
- can be shown to be universal

# Building hash function: universal hashing

**Universal hashing**: refers to selecting a hash function at random from a family of hash func. with a certain property

Useful for algorithms that need multiple hash functions

ex.: rehashing

the data structure needs to be rebuilt

if too many collisions occur

# Hash function

- perfect hash function
  - injective: maps distinct elements with no collisions
  - it is too expensive to compute it for every input

➔ build a hash function to minimize collisions

**good hash function**

In practice:

- use **heuristic** information to create a hash function that is likely to perform well

Choose between:

- simple and fast, but have a high number of collisions;
- more complex functions, with better quality, but take more time to calculate

# Good hash function

A **good hash function** satisfies

the assumption of **simple uniform hashing**

- a key x is equally likely to hash to any of the *m* slots

$P(h(x)=j) = 1/m \quad , \text{for any } j=0,\ldots,m-1$

---

- each bucket is equally likely to be occupied

- probability that two keys map to the same slot is 1/m

$$P(h(x)=h(y)) = \frac{1}{m}$$

**x, y - independent random variable**

# Good hash function

Need: qualitative information about P

**uniform distributed keys**

Example:

- keys are random real numbers independently and uniformly distributed in the range [0,1).

  $h(k) = [k * \boldsymbol{m}]$

  satisfies the simple uniform hashing property

- keys are random integers independently and uniformly distributed in the range 0 to N−1

  where N much larger than m

  $h(k) = k \bmod \boldsymbol{m}$

  satisfies the simple uniform hashing property

# Hash function

Need: qualitative information about P **not uniformly distributed keys**

**Special-purpose hash function**

- exceptionally good for a specific kind of data

  no performance on data with different distribution

Example (1)

input data: file names such as FILE0000.CHK, FILE0001.CHK, FILE0002.CHK, etc., with mostly sequential numbers.

- extracts the numeric part **k** of the file name ***fn***

  *h(**fn**) = numeric_part(**fn**) mod m*

# Hash function

Example (2)

input data: text in any natural language

has highly non-uniform distributions of characters, and character pairs, very characteristic of the language

- string
- variable length data

it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way

```
Example of hash function:
Function HashMultiplicative(strKey) {
    hash = INITIAL_VALUE;
    for i = 1, length(strKey) do
      hash = M * hash + strKey [i]
    endfor
    return hash % TABLE_SIZE;
}
```

**D. Bernstein,**
**`comp.lang.c`, (1991 ?)**

INITIAL_VALUE = 5381
M = 33

**B. Kernighan, D. Ritchie,**
***The C Programming Language* , 1978**

INITIAL_VALUE = 0
M = 31

# Hash function

Example (3)

input data: an unchanging dictionary
(text in a natural language)

If the dictionary is unchanging, you might want to consider perfect hashing;

- for a given dataset you can guarantee that there will be no collisions

# Hash function

Example (4)

assume

input data: three-letter words

formed with any of a set of char extended ASCII code

perfect hashing

- h(str) = ASCIIcode(str[0]) * $256^2$
  + ASCIIcode(str[1]) * $256^1$
  + ASCIIcode(str[2])
- ASCIIcode(str[i]) : values from range 0..255
- hash table of size $3^{256}$ **?!**

# Hash table and hash function in programming languages

Java

HashMap

– Hash table based implementation of the Map interface

HashSet

– implements the Set interface, backed by a hash table

# Hash in programming languages

## Java Object

- public int hashCode()

*As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)*

- public boolean equals(Object obj)

  if two objects are equal then they must return same hash code

  - that is compared by equal() of that class

# Hash in programming languages

The java.lang.String hash function

Given: s of java.lang.String

h(s)= s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

- uses arithmetic int

where s[i] is the ith character of the string,

n is the length of the string

^ indicates exponentiation.

(The hash value of the empty string is zero.)

# Hash tables in programming languages

- STL map: Associative key-value pair held in balanced binary tree structure
  - usually a red-black tree

New in C++ 11

- unordered_map

Some implementations

- hash_map was a common extension provided by many library implementations

# Hash table – performance analysis

Hash function assumption: computed in $\Phi(1)$

Load factor: $\alpha = n / m$

- n = number of entries used
- m = hash table size

Hash table and $\alpha$

$\alpha = 1$ : table is full ➔ increase the table

**rehash** !!

$\alpha < 1$

**$\alpha$ from 0 to 0.75** **(up to 0.7, about 2/3 full)**

- With a good hash function, the average lookup cost is nearly constant

Beyond that point, the prob. of collisions and the cost of handling them increases.

**a low load factor**

- wasted memory
- not necessarily any reduction in search cost

# Performance : collision resolution by chaining

under simple uniform hashing

average complexity for operations: Φ (1)

**add -** Φ (1)

**search:**

**Theorem**:

under simple uniform hashing an unsuccessful search takes Φ (1 + $\alpha$ ) time on the average.

**Theorem**:

under simple uniform hashing a successful search takes

Φ(1 + $\alpha$ ) time on the average.

**delete**: search (the list) and remove

# Performance : collision resolution by open addressing

under the assumption of simple uniform hashing and constant $\alpha$

average complexity for operations: $\Phi$ (1)

**Theorem**:

under the assumption of uniform hashing

with load factor $\alpha = n/m < 1$

the expected number of probes

- in an unsuccessful search is at most $1/(1-\alpha)$,
- for operation add is at most $1/(1-\alpha)$
- in a successful search is at most $1/\alpha * \log(1/(1-\alpha))$

# Collision resolution by chaining. Variations.

we can use: list

other data structure

- sorted list
- doubly linked list
- self-balancing tree
  - worst-case time O(log n)
  - extra memory and extra design
    - if long delays must be avoided at all costs
      e.g. in a real-time application

.

# Collision resolution. Variations

two-level hashing

- first level

  use a hashing function chosen from a family of universal hash functions.

- second level

  Use (small) secondary table $S_j$ with an associated hash function hj

…

# Hash, chaining

Hash, chaining.

- representation

  use : (semistatic) array,

  singly linked nodes for chaining

  elements are TKey (…TElement)

- add an element to the hash

  pseudocode

Assume: there is a hashFunc: TKey -> {0,…,m-1}

external to element and hash table

# Open addressing

Consider inserting the keys

$$31, 60, 5, 29, 18, 16, 17$$

into a hash table of length $m = 11$

using open addressing with the primary hash function

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

Illustrate the result of inserting these keys by using

- linear probing
- quadratic probing with $c1 = 0$ and $c2 = 1$
- double hashing with $h2(k) = 1 + (k \bmod (m - 1))$

# Hash, chaining

Hash, chaining

Consider the keys to be inserted in a hash are the next 26 small letters:

a,b,c,d,e,f,g,h,I,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z

Insert the keys

j, k, l, m, n, u, v, w, a, b, c

into a hash table of length m=11

Describe the steps and illustrate the result

1. a
2. b
3. c
4. d
5. e
6. f
7. g
8. h
9. I
10. j
11. k
12. l
13. m
14. n
15. o
16. p
17. q
18. r
19. s
20. t
21. u
22. v
23. w
24. x
25. y
26. z