Hash Tables

Hash functions and hash tables

Idea and Examples

Hash function details

Address Generation (Hash code + Compression code)

Collision Resolution

Linear probing

Quadratic probing

Double hashing

Idea

Hash tables are an example of a dictionary.

Data is stored and retrieved by use of a function of the key.

It is stored, but not sorted!

Example

Student records are stored in an array using a 7 digit student i.d. the index.

If the i.d. were used unmodified, the array would have to have enough room for 10,000,000 student records.

Instead, student i.d.'s are hashed to produce an integer between, say 1 and 100,000 which indexes into an array.

Problem

Since a possible 10,000,000 numbers are being compressed into just 100,000, how can we guarantee that no 2 i.d.'s end up stored in the same place?

Problem 1

Address Generation

Construction of the function $h(K_i)$

- **Simple to calculate**
- **Uniformly distribute the elements in the table**

Problem 2

Collision Resolution

The Idea:

$\forall$ key K_i

$h(K_i)$ = position of K_i in the table

$h(K_i) = pos$ pos: integer

$h(K_i) \neq h(K_g)$ $i \neq g$

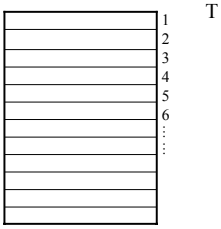
Search for key K_i

↓

$O(1)$!

↑

Insertion

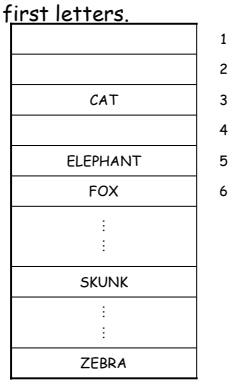


—Example —

The keys all have different first letters.

CAT, ELEPHANT, FOX, SKUNK, ZEBRA

h (CAT) = 3
h (ELEPHANT) = 5
h (FOX) = 6

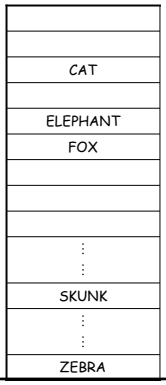


Problem

If we want to insert a key that doesn't have a different first letter

⇒

COLLISIONS



Problem

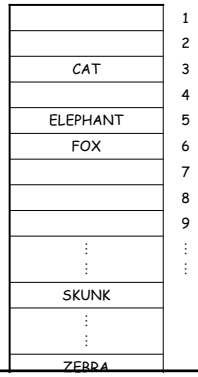
If we want to insert a key that doesn't have a different first letter

⇒ COLLISIONS

We want to insert:
CRICKET

h (CRICKET) = 3

position 3 is occupied



Definition:

load factor of an Hash Table

$$\alpha = \frac{n}{N}$$

→ # of elements
→ # of cells

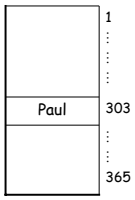
Collisions

What is the probability of a collision?

The Birthday Paradox

- 50 people (n = 50)
- h (person) = value of her birth date (from 1 to 365)

1 Jan. = 1
2 Jan. = 2 (N = 365)
⋮
31 Dec.=365



The load factor is $\alpha = \frac{50}{365}$ So the table is only ≈ 13.76 % full
But

The probability that 2 people have the same birthday turns out to be 0.97 !!!!

Address Generation

Hash code map:
h_1: keys → integers

Compression map:
h_2: integers → [0, **TableSize** - 1]

$$h(x) = h_2(h_1(x))$$

Hash Code Maps

Hash codes reinterpret the key as an integer. They should: 1. Provide good "spread"
2. Give the same result for the same key

Examples:

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Good in general, except for numeric and string keys
- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

Hash Code Maps

- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Code Maps (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
    $a_0\, a_1 \ldots a_{n-1}$
  - We evaluate the polynomial
    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1}z^{n-1}$
    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Hash Code Maps Cont.

- Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:
  - The following polynomials are successively computed, each from the previous one in $O(1)$ time
    $p_0(z) = a_{n-1}$
    $p_i(z) = a_{n-i-1} + zp_{i-1}(z)$
    $(i = 1, 2, \ldots, n-1)$
- We have $p(z) = p_{n-1}(z)$

Compression Maps

Compression Maps:

• Take the output of the hash code and compress it into the desired range.

• If the result of the hash code was the same, the result of the compression map should be the same.

•Compression maps should maximize "spread" so as to minimize collisions.

Compression Maps Examples

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - The reason has to do with number theory and is beyond the scope of this course

Compression Maps Examples

- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Address Generation
Some examples …

Address Generation (a)

N = size of the table
r = ⌈ log N ⌉

$h_1(x)$: integer cast

a) $h_2(h_1(x))$: = subset (of r bits) of $h_1(x)$

a.1) the r least significant bits

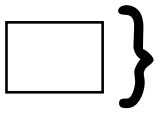
a.2) the r most significant bits

a.3) the central r bits

→ Simple to calculate

→ Doesn't guarantee a random distribution

—Example —



2^9

N = 2^9
r = 9

CHAT␣␣

Coding of letters

A	000001
B	000010
C	000011
⋮	
H	001000
⋮	
T	010010
⋮	
␣	100000

h(000011001000000001010010100000100000) =

a.2 000011001
All the animals that begin with "CH" hash to the same location.

a.1 000100000
All the animals of 4 (or less) characters hash to the same location.

Address Generation (b)

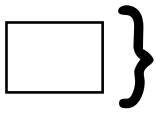
$h_1(x)$: integer cast

b) $h_2(h_1(x))$: sum of subset of bits of $h_1(x)$

→ Simple to calculate

→ More random than **a)**

—Example —



2^9

N = 2^9
r = 9

CHAT␣␣

Coding of letters

A	000001
B	000010
C	000011
⋮	
H	001000
⋮	
T	010010
⋮	
␣	100000

h(000011001000000001010010100000100000) =

b)

000011001 most significant
000101001 central
000100000 least significant
XOR 000010000

Address Generation (c)

$h_1(x)$: integer cast

c) $h_2(h_1(x))$: subset (of r bits) of $h_1(x)^2$

→ Multiplication is involved
→ More random than **a)** and **b)**

Address Generation (d)

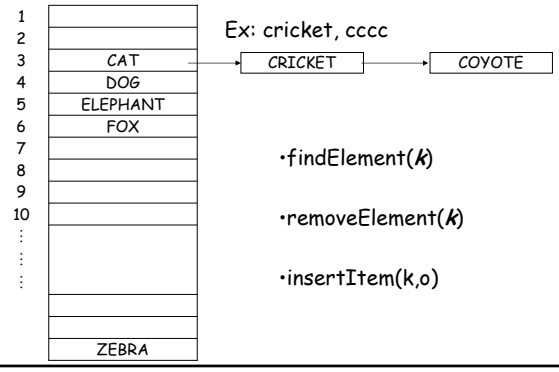
d) $h_2(h_1(x))$: = $h_1(x)$ MOD N

→ Division is involved!
→ Very random (if N is odd)

Collision Resolution

Collision Resolution

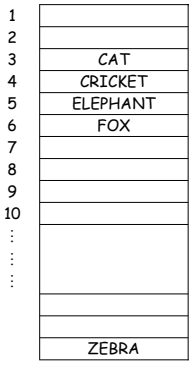
Separate Chaining



- findElement(*k*)
- removeElement(*k*)
- insertItem(k,o)

Collision Resolution (examples)

1. Open Addressing



→COYOTE
- h (COYOTE) = 3 OCCUPIED
- We consider 4 OCCUPIED
- We consider 5 OCCUPIED
- " 6 OCCUPIED
- " 7 FREE!

Linear Probing

Collision Resolution (1)
Linear Probing

$$\underbrace{h(K_i)}_{h_0(K_i)}, \underbrace{h(K_i)+1}_{h_1(K_i)}, \underbrace{h(K_i)+2}_{h_2(K_i)}, \underbrace{h(K_i)+3}_{h_3(K_i)} \ldots.$$

Let $h_0(K_i) = h(K_i)$

$$h_j(K_i) = [h(K_i) + j] \bmod N$$

Search with Linear Probing

- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

```
Algorithm findElement(k)
    i ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return NO_SUCH_KEY
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return NO_SUCH_KEY
```

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called **AVAILABLE**, which replaces deleted elements
- removeElement(k)
  - We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item *AVAILABLE* and we return element o
  - Else, we return *NO_SUCH_KEY*
- insert Item(k, o)
  - We throw an exception if the table is full
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - **A cell i is found that is either empty or stores AVAILABLE, or**
    - **N cells have been unsuccessfully probed**
  - We store item (k, o) in cell i

Performances of Linear Probing

Search: Average number of probes ....

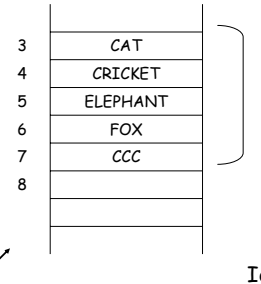
$C(\alpha)$

Experimental results for a hash table with load factor α

α=n/N	$C(\alpha)$
0.1 (10%)	1.06
0.5 (50%)	1.50
0.75 (75%)	2.50
0.9 (90%)	5.50

Problem

with Linear Probing: PRIMARY CLUSTERING



h (COYOTE) = 3
h_1 (COYOTE) = 4
h_2 = 5
h_3 = 6
h_4 = 7
h_5 = 8 !

Here we are using as address generation the integer corresponding to the first letter

Idea:
Use a non-linear probe

Collision Resolution (2)

Quadrating Probing

$h(k_i)$, $h(k_i)+1$, $h(k_i)+4$, $h(k_i)+9$, ...
$h_0(k_i)$ $h_1(k_i)$

$$h_j(k_i) = [h(k_i) + j^2] \ mod \ N$$

N: prime

→ *mod* is hard to calculate

→Visits only half of the table

but...

Performances of Quadratic Probing

Experimental results for a hash table with load factor α

Search

α = n/N	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.44
0.75 (75%)	1.99
0.9 (90%)	2.79

Problem

with non linear Probing: SECONDARY CLUSTERING

Two keys that hash to the same place follow the same collision path

Idea:

Double Hashing

Collision Resolution

Ex: **Open Adressing: (3) Double Hashing**

$h(k_i)$, $h(k_i)+h'(k_i)$, $h(k_i)+2h'(k_i)$, $h(k_i)+3h'(k_i)$, ...

(h_0, h_1, h_2, h_3)

$h_j(k_i) = [h(k_i) + j \cdot h'(k_i)] \bmod N$

OR

Ex:

$h(k_i)$, $h(k_i)+h'(k_i)$, $h(k_i)+4\,h'(k_i)$, $h(k_i)+9\,h'(k_i)$, ...

$h_j(k_i) = [h(k_i) + j^2 \cdot h'(k_i)] \bmod N$

➡ Choice of $h(\cdot)$

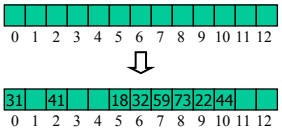
Choice of secondary hashing function $h'(\,)$

Example of Double Hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$

• Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



Another Example of Double Hashing

$h(k_i) = k_i \bmod N$

$h'(k_i) = k_i \operatorname{div} N$

N prime!

Performances of Double Hashing

Experimental results for a hash table with load factor α

Search

α = n/N	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.38
0.75 (75%)	1.83
0.9 (90%)	2.55

Performance of Hashing: Summary

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ affects the performance of a hash table
- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is approximately
  $1 / (1 - \alpha)$
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches