Hash Functions and Hash Tables

A **hash function** h maps keys of a given type to integers in a fixed interval $[0, \dots, N-1]$. We call $h(x)$ **hash value** of x.

Examples:

- $h(x) = x \mod N$
  is a hash function for integer keys
- $h((x, y)) = (5 \cdot x + 7 \cdot y) \mod N$
  is a hash function for pairs of integers

$h(x) = x \mod 5$

	key	element
0		
1	6	tea
2	2	coffee
3		
4	14	chocolate

A **hash table** consists of:

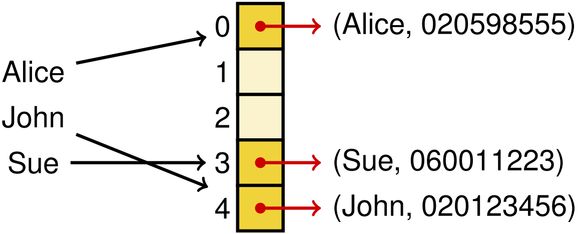
- hash function h
- an array (called table) of size N

The idea is to store item (k, e) at index $h(k)$.

Hash Tables: Example 1

Example: phone book with table size $N = 5$

- hash function $h(w) = (\text{length of the word } w) \mod 5$



- Ideal case: one access for find(k) (that is, $O(1)$).
- Problem: collisions
  - Where to store Joe (collides with Sue)?
- This is an example of a bad hash function:
  - Lots of collisions even if we make the table size N larger.

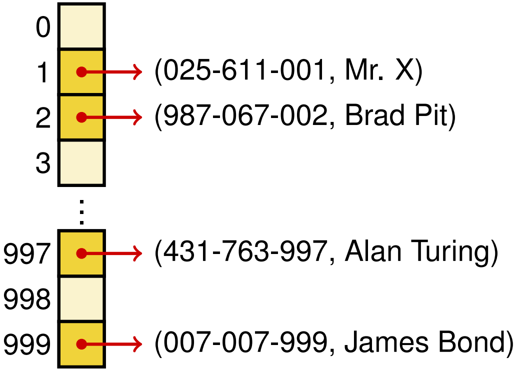
Hash Tables: Example 2

A dictionary based on a hash table for:

- items (social security number, name)
- 700 persons in the database

We choose a hash table of size $N = 1000$ with:

- hash function $h(x) =$ last three digits of x

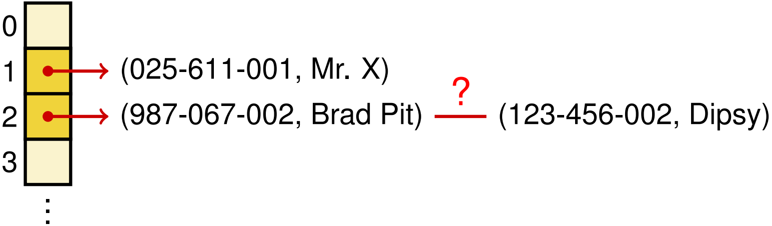


Collisions

Collisions

occur when different elements are mapped to the same cell:

- Keys k_1, k_2 with $h(k_1) = h(k_2)$ are said to collide.



(025-611-001, Mr. X)

(987-067-002, Brad Pit)

(123-456-002, Dipsy)

Different possibilities of handing collisions:

- chaining,
- linear probing,
- double hashing, . . .

Collisions continued

Usual setting:

- The set of keys is much larger than the available memory.
- Hence collisions are unavoidable.

How probable are collisions:

- We have a party with p persons. What is the probability that at least 2 persons have birthday the same day ($N = 365$).
- Probability for no collision:

$$\begin{aligned} q(p, N) &= \frac{N}{N} \cdot \frac{N-1}{N} \cdots \frac{N-p+1}{N} \\ &= \frac{(N-1)\cdot(N-2)\cdots(N-p+1)}{N^{p-1}} \end{aligned}$$

- Already for $p \geq 23$ the probability for collisions is > 0.5.

Hashing: Efficiency Factors

The efficiency of hashing depends on various factors:

- hash function
- type of the keys: integers, strings,...
- distribution of the actually used keys
- occupancy of the hash table (how full is the hash table)
- method of collision handling

The load factor α of a hash table is the ratio n/N, that is, the number of elements in the table divided by size of the table.

High load factor $\alpha \geq 0.85$ has negative effect on efficiency:

- lots of collisions
- low efficiency due to collision overhead

What is a good Hash Function?

Hash fuctions should have the following properties:

- Fast computation of the hash value ($O(1)$).
- Hash values should be distributed (nearly) uniformly:
  - Every has value (cell in the hash table) has equal probabilty.
  - This should hold even if keys are non-uniformly distributed.

The goal of a hash function is:

- 'disperse' the keys in an apparently random way

Example (Hash Function for Strings in Python)

We dispay python hash values modulo 997:

$$h(\text{‘a}') = 535 \quad h(\text{‘b}') = 80 \quad h(\text{‘c}') = 618 \quad h(\text{‘d}') = 163$$

$$h(\text{‘ab}') = 354 \quad h(\text{‘ba}') = 979 \quad \ldots$$

At least at first glance they look random.

Hash Code Map and Compression Map

Hash function is usually specified as composition of:

- **hash code map**: h_1 : keys $\rightarrow$ integers
- **compression map**: h_2 : integers $\rightarrow [0, \dots, N-1]$

The hash code map is appied before the compression map:

- $h(x) = h_2(h_1(x))$ is the composed hash function

The compression map usually is of the form $h_2(x) = x \mod N$:

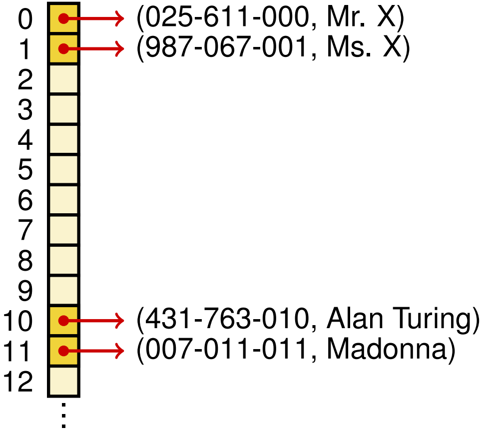
- The actual work is done by the hash code map.
- What are good N to choose? …see following slides

Compression Map: Example

We revisit the example (social security number, name):

- hash function $h(x) = x$ as number $\mod 1000$

Assume the last digit is always 0 or 1 indicating male/femal.



Then 80% of the cells in the table stay unused! Bad hash!

Compression Map: Division Remainder

A better hash function for 'social security number':

- hash function $h(x) = x$ as number $\mod 997$
- e.g. $h(025 - 611 - 000) = 025611000 \mod 997 = 409$

Why 997? Because 997 is a prime number!

- Let the hash function be of the form $h(x) = x \mod N$.
- Assume the keys are distributed in equidistance $\Delta < N$:

$$k_i = z + i \cdot \Delta$$

We get a collision if:

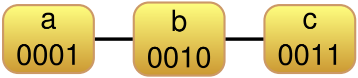
$$\begin{aligned} & k_i \mod N = k_j \mod N \\ \iff & z + i \cdot \Delta \mod N = z + j \cdot \Delta \mod N \\ \iff & i = j + m \cdot N \quad \text{(for some } m \in \mathbb{Z}\text{)} \end{aligned}$$

Thus a prime maximizes the distance of keys with collisions!

Hash Code Maps

What if the keys are not integers?

- **Integer cast**: interpret the bits of the key as integer.



a 0001

b 0010

c 0011

000100100011 = 291

What if keys are longer than 32/64 bit Integers?

- **Component sum**:
  - partition the bits of the key into parts of fixed length
  - combine the components to one integer using sum (other combinations are possible, e.g. bitwise xor, . . .)

1001010 | 0010111 | 0110000

1001010 + 0010111 + 0110000 = 74 + 23 + 48 = 145

Hash Code Maps, continued

Other possible hash code maps:

- Polynomial accumulation:
  - partition the bits of the key into parts of fixed length

    $$a_0 a_1 a_2 \dots a_n$$

  - take as hash value the value of the polynom:

    $$a_0 + a_1 \cdot z + a_2 \cdot z^2 \dots a_n \cdot z^n$$

  - especially suitable for strings (e.g. $z = 33$ has at most 6 collisions for 50.000 english words)
- Mid-square method:
  - pick m bits from the middle of x^2
- Random method:
  - take x as seed for random number generator

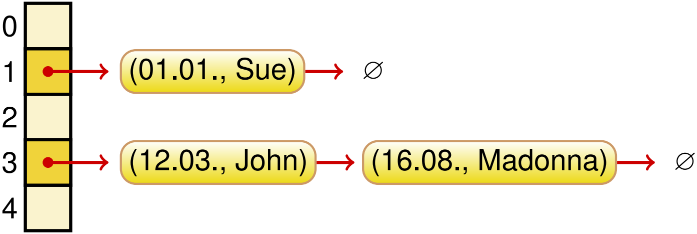
Collision Handling: Chaining

Chaining: each cell of the hash table points to a linked list of elements that are mapped to this cell.

- colliding items are stored outside of the table
- simple but requires additional memory outside of the table

Example: keys = birthdays, elements = names

- hash function: $h(x) = (\text{month of birth}) \mod 5$



Worst-case: everything in one cell, that is, linear list.

Collision Handling: Linear Probing

Open addressing:

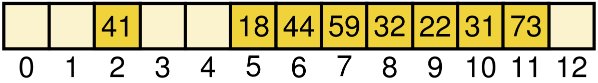
- the colliding items are placed in a different cell of the table

Linear probing:

- colliding items stored in the next (circularly) available cell
- testing if cells are free is called ‘probing’

Example: $h(x) = x \mod 13$

- we insert: 18, 41, 22, 44, 59, 32, 31, 73



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

Colliding items might lump together causing new collisions.

Linear Probing: Search

Searching for a key k (findElement(k)) works as follows:

- Start at cell $h(k)$, and probe consecutive locations until:
    - an item with key k is found, or
    - an empty cell is found, or
    - all N cells have been probed unsuccessfully.

```
findElement($k$):
    $i = h(k)$
    $p = 0$
    while $p < N$ do
        $c = A[i]$
        if $c == \varnothing$ then return No_Such_Key
        if $c.key == k$ then return $c.element$
        $i = (i + 1) \mod N$
        $p = p + 1$
    return No_Such_Key
```

Linear Probing: Deleting

Deletion remove(k) is expensive:

- Removing 15, all consecutive elements have to be moved:

		15	2	3	4	5	6	7				
0	1	2	3	4	5	6	7	8	9	10	11	12

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

To avoid the moving we introduce a special element Available:

- Instead of deleting, we replace items by Available (A).

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

- From time to time we need to ‘clean up’:
  - remove all Available and reorder items

Linear Probing: Inserting

Inserting insertItem(k, o):

- Start at cell $h(k)$, probe consecutive elements until:
  - empty or Available cell is found, then store item here, or
  - all N cells have been probed (table full, throw exception)

		A	16	17	4	A	6	7				
0	1	2	3	4	5	6	7	8	9	10	11	12

Example: insert(3) in the above table yields ($h(x) = x \mod 13$)

		A	16	17	4	3	6	7				
0	1	2	3	4	5	6	7	8	9	10	11	12

Important: for findElement cells with Available are treated as filled, that is, the search continues.

Linear Probing: Possible Extensions

Disadvantages of linear probing:

- Colliding items lump together, causing:
  - longer sequences of probes
  - reduced performance

Possible improvements/ modifications:

- instead of probing successive elements, compute the i-th probing index h_i depending on i and k:

$$h_i(k) = h(k) + f(i, k)$$

Examples:

- Fixed increment c: $h_i(k) = h(k) + c \cdot i$.
- Changing directions: $h_i(k) = h(k) + c \cdot i \cdot (-1)^i$.
- Double hashing: $h_i(k) = h(k) + i \cdot h'(k)$.

Double Hashing

Double hashing uses a secondary hash function $d(k)$:

- Handles collisions by placing items in the first available cell

  $$h(k) + j \cdot d(k)$$

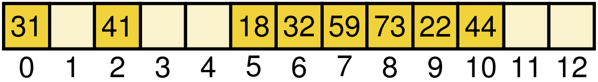
  for $j = 0, 1, \ldots, N-1$.
- The function $d(k)$ always be > 0 and $< N$.
- The size of the table N should be a prime.

Double Hashing: Example

We use double hashing with:

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

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



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

Performance of Hashing

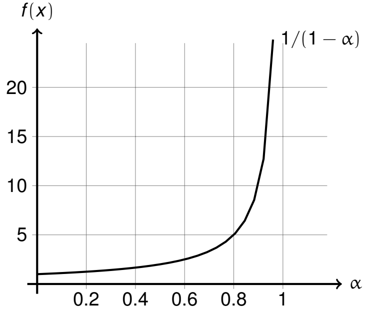
In worst case insertion, lookup and removal take $O(n)$ time:

- occurs when all keys collide (end up in one cell)

The load factor $\alpha = n/N$ affects the performace:

- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes is:

$$1/(1-\alpha)$$



Performance of Hashing

In worst case insertion, lookup and removal take $O(n)$ time:

- occurs when all keys collide (end up in one cell)

The load factor $\alpha = n/N$ affects the performace:

- Assuming that the hash values are like random numbers, it can be shown that the expected number of probes is:

$$1/(1-\alpha)$$

In practice hashing is very fast as long as $\alpha < 0.85$:

- $O(1)$ expected running time for all Dictionary ADT methods

Applications of hash tables:

- small databases
- compilers
- browser caches

Universal Hashing

No hash function is good in general:

- there always exist keys that are mapped to the same value

Hence no single hash function *h* can be proven to be good.

However, we can consider a set of hash functions *H*.
(assume that keys are from the interval $[0, M-1]$)

We say that *H* is universal (good) if for all keys $0 \leq i \neq j < M$:

$$\text{probability}(h(i) = h(j)) \leq \frac{1}{N}$$

for *h* randomly selected from *H*.

Universal Hashing: Example

The following set of hash functions H is universal:

- Choose a prime p betwen M and $2 \cdot M$.
- Let H consist of the functions

$$h(k) = ((a \cdot k + b) \mod p) \mod N$$

for $0 < a < p$ and $0 \leq b < p$.

Proof Sketch.

Let $0 \leq i \neq j < M$. For every $i' \neq j' < p$ there exist unique a, b such that $i' = a \cdot i + b \mod p$ and $j' = a \cdot i + b \mod p$. Thus every pair (i', j') with $i' \neq j'$ has equal probability. Consequently the probability for $i' \mod N = j' \mod N$ is $\leq \frac{1}{N}$. $\square$

Comparison AVL Trees vs. Hash Tables

Dictionary methods:

	search	insert	remove
AVL Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
Hash Table	$O(1)$ [1]	$O(1)$ [1]	$O(1)$ [1]

[1] expected running time of hash tables, worst-case is $O(n)$.

Ordered dictionary methods:

	closestAfter	closestBefore
AVL Tree	$O(\log_2 n)$	$O(\log_2 n)$
Hash Table	$O(n + N)$	$O(n + N)$

Examples, when to use AVL trees instead of hash tables:

1. if you need to be sure about worst-case performance
2. if keys are imprecise (e.g. measurements),
   e.g. find the closest key to 3.24: closestTo(3.72)