Today lesson is on Hash-tables.

Preface:

Difinition: Abstract Data Type - (from geeksforgeeks) ADT from now on.

ADT is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.
The difinition of ADT only mentions what operations are to be performed, but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It's called "abstract" because it gives an implementation independent view. The process of providing only the essentials and hiding the details is known as abstraction.
For example: stack, we can define next operations:
add - element will be added to the set.
remove - in LIFO order.
etc. There can be some implementation: by arrays, linked-list etc., but when we looks at a stack as ADT we don't care about the implementations, all we know is that there are some operations and that elements will be removed by LIFO order.

Map<K, V> myMap = new

TreeMap
HashMap
LinkedHashMap

ADT

Implementations

Dictionary - ADT

maintain set of items, each with a key

- Insert (item) overwrite any existing key
- delete (item)
- search (key) : return item with given key, or report doesnt exist.

note: its a little different then BST, in BST if we didnt find a key, we could find the next smaller or larger key: saccessor or predecessor, here we are not able to do so.

we are assuming that items have unique keys, no two items have the same key, and one way to enforce that is when you insert an item with an existing key it overwrite whatever key was there (Pyton, Java..)

One way to solve dictionaries is by balanced binary search trees as AUL - and we can do all of the operations on log(n) time (we ignore from the face that doing a search in AUL gives us more information)

This is one solution - it turns out that we can get better.

like we already saw, in the comparision model the best way to do sort is nlog(n) and search in log(n), then we saw in the RAM model - if we assume that our items are integers we can sort faster - sometims in linear time - here we will see how to search faster than log(n) - and we are going to get down to constant time. - with no assumptions, except maybe that the keys are integers. O(1) with high probability!

Python: dict

D[key] ~ search

D[key] = Val ~ insert

del D[key] ~ delete

→ item = (key, Value) Java too.

Motivations: (as a result in every modern programming language)

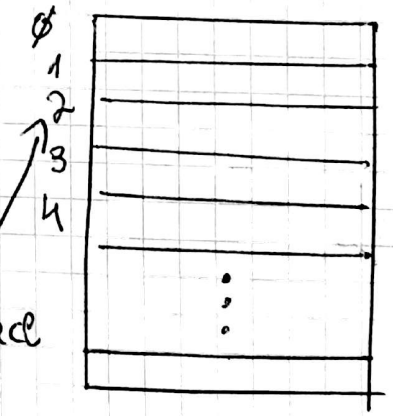
- docdist - comparing between documents. (problem) [c, c++, ruby, python, Java]
- databases (hashing, search trees)
- compilers & interpreters
- network router
  server
- substring search (editors)
- string commonalities
- file\dir syn chronization [e.g dropbox if you send a file, you check through hasing. if the file had changed]
- cryptography

Simple approach:

Direct-access table

store items in array indexd by key [integers keys] - item with index 2 - will be in second place

all empty slots will be null



2 problems: Badness

(1) Keys may not be nonneg integers!

(2) gigantic memory hog

Solution to (1): **prehash**

- maps keys to nonneg integers [string, object...]
- in theory keys are finite and discrete.
- in python, hash(x) is the prehash of x (if you give an integer as a key it returns the integer, and if you give a key it let say x so hash(x) is integer
- hash('\0B') = hash('\0\0C') = 64

ideally: hash(x) = hash(y) <=> x == y

--hash-- -> can implement hash method which tells python what to do, when you call hash of your object.

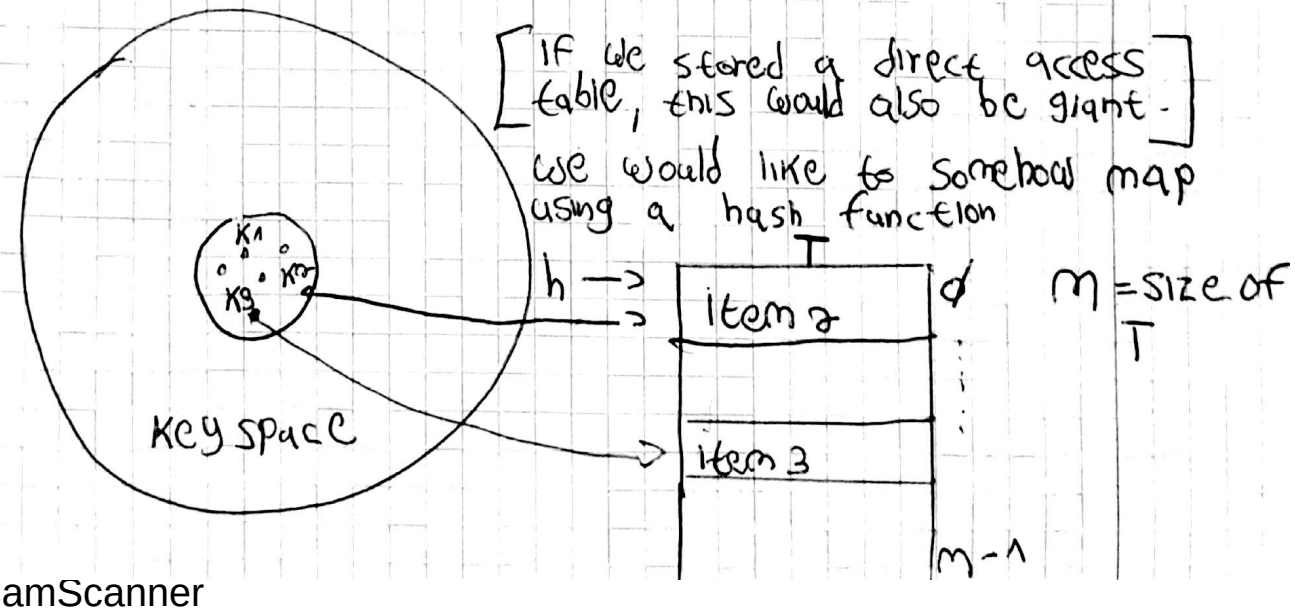
prehash must not change! Dont mess with it.

[for example a list which is a mutable object can't be in a hash table because if you append objects to the list it would get another value]

The more interesting problem is reducing space

Solution to (2): **hashing** - to hash means cut into pieces and mix around.

- reduce universe of all keys (integers) down to reasonable size m for table



Idea: $m = \Theta(n)$ (5)

n - number of keys in the dictionary (right now)
That is if we want to store n items will we use $2n$, $3n$ space, but not much more.

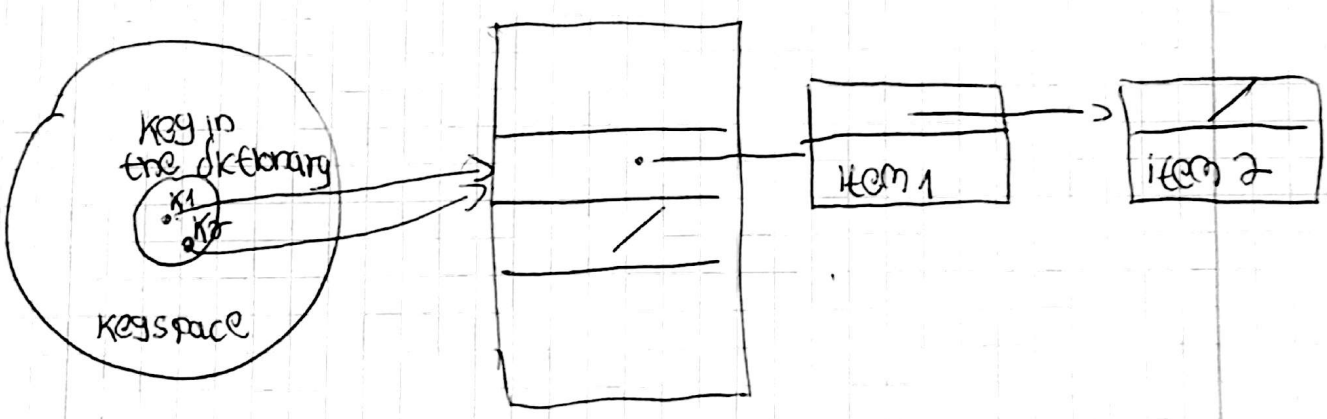
> By difinition hashing function $h: [0, 1, \ldots m-1]$ hashing function needs to know what m is.

Collision: $h(k_i) == h(k_j)$, but $k_i \neq k_j$

and then there gonna be chainning.

Chaining: linked list of colliding elements

- in each slot of hash table



- Some slots can be null. Worst case all of the keys are mapped to the same slot by h(key)
Worst case $\Theta(n)$ [the reason that it actually work is randomization]

Simple uniform hashing: [false assumption]
each key is equally likely to be hashed to any slot of the table independent of where other keys hashing [key one goes to random place, no matter what place key two goes to random place.

Under this assumption we can analize hashing with chaining

Analysis:

- expected length of chain for n keys, m slots. $= n/m$. The probability for each key is $1/m$, $1/m + 1/m + \dots$ (n times) $= n/m = \alpha$ = load factor of the table

As long as $m = \Theta n \rightarrow \alpha$ is constant.

$= \Theta 1$ running time $= O(1 + |\text{chain}|) = O(1+\alpha)$

[We write 1 because α, can be smaller than 1, and we always have the cost of computing the hash function!

Assume Simple Uniform Hashing → As long as α is constant $m = \Theta n$ → Constant running time all of operations

That is not really why hasing works → It's an intuition why hashing works

How do we construct h? 3 hash functions The first and the second are bad and the third is good, but the first 2 function most of the time ok!

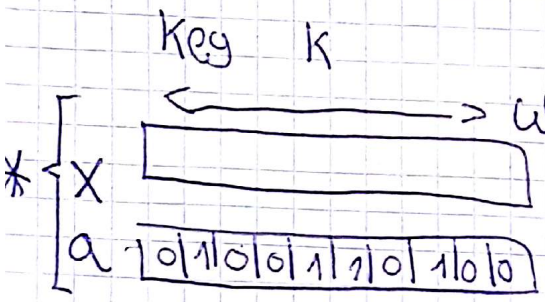
(1) DIVISION method $h(k) = k \bmod m$

There can be a lot of situations, where it not works good especially when m and k have a common factor e.g m is even and k always even - we use half of the table. Most of the time works good if m is prime and m not very close to a power of 2 or ten.

(2) multiplication method

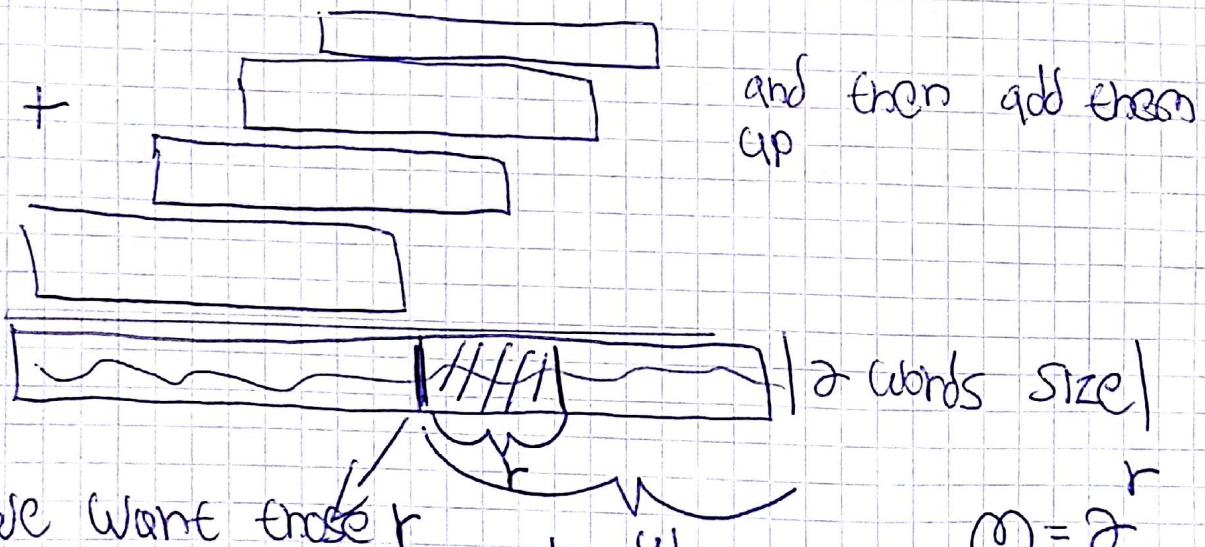
$$h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w-r)$$

- assuming w bit machine



we take random integer a among all w bit long integers

- we have 4 times 1 in a and thus we have 4 copies of k



we want those r bits in range $0 - m - 1$

$m = 2^r$

In the middle all of the randomization happen ③
It works because this is hashing - mixing up numbers add numbers etc...

a better be odd, and sould not be close to power of two 2^r 2^{r-1} ⇒ theoreti calls this can be bad

③ Universal hashing $h(k) = [(ak+b) \bmod p] \bmod m$

key

random $\in \{0, \dots p-1\}$

to make it between $0 \sim m-1$

prime $> |a|$ universe

for worst case keys

$k_1 \neq k_2$:

$Pr\{h(k_1) == h(k_2)\} = 1/m$

probability ⇓

That is that if we choose to use in universal hashing the probability that will be collisions is $1/m$ that is in the worst case (because a, b are random) → that is the best case for us.

Crucial thing that is missing is: What should m be?

If it will be too small, eventually we will insert more and more keys and α will be too big

If it will be big - kind of wasteful the whole idea is to avoid having one slot for every possible key

How to choose m?

- Want $m = \Theta(n)$
  $\Omega(n)$ → α constant
  $O(n)$ → space linear
  $\Rightarrow \alpha = \Theta(1)$

Idea: start small: $m = 8$

grow/shrink as necessary

- If $n \geq m$: grow table

Grow table: $m \to m'$

- make table of size m'
- build new hash fn prime
- rehash

```
for each item in T:
    T'.insert(item)
```

time

$\Theta(n + m + m') = \Theta(n)$

list / slots / new table

The size of the new table $T' \to m'$ have to update in the h'.

$m' = 2m$.

$m' = m+1$ Wrong answer

Cost of inserts if we choose to increment m by 1

Cost of m insertions:

$\Theta(1+2+3+4+\dots n)$ n insertions

each in cost of $\Theta n = \Theta(n^2)$

$\Rightarrow (1, 2, 4, 8, 16, \dots n)$ = geomerc series Θn

(can time we double n)

This is data structure that suppose to be constant time for operation, here in insertion when we need to add and the table is full $\to \Theta 2n$

log n of them are bad, but in the rest you don't do anything. $\to$ this is idea that we call

TABLE DOUBLING

Amortization:

-operton takes "$T(n)$ amortized" IF K operations take $\leq K \cdot T(n)$ time

think of meaning "$T(n)$ on average", where the average over all operations

In table doubling
So the amortized running is beta 1.

k insertions take $\Theta(k)$ time that means $\Theta(1)$ amortized / insert [you dont care that some insertions are expensive, most of the time is $\Theta 1$]

Also: k inserts & deletes take Θk

Problem: suppose we have n insertions, m became bigger and then we delete all of the keys → So now n=0, m = max value of n how can we fix that?

(1) if $m = n/2$ then shrink → $m/2$

SLOW: 2^k insert $\longleftrightarrow$ Delete $2^k + 1$ $= \Theta n$ per operation

[e.g let say we have 8 size of m then we insert 1 key, double size of $m' = 16$, $n = 9$ now we delete 1 key $m' = 16$, $n = 8$ and then shink again]

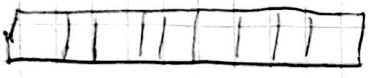
(2) if $m = \frac{n}{4}$ then shrink → $m/2$

maintain the invariant $n \leq m \leq 4n$

amortized time → $\Theta(1)$

Side effect → python lists

python lists also known as resizeable arrays

can  random access - constant
append - constant

deleting first item linear time. - moving all other items.

So in fact list.append
list.pop
constant amortized!

There is a way to get rid of the amortized, for people who work in real time systems

String Matching:

given two strings s & t: does s occur as a substring of t?

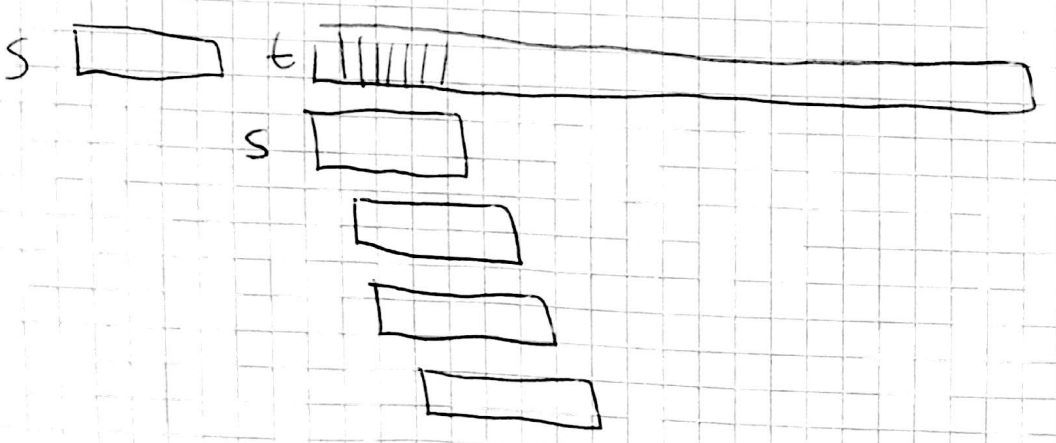
s = 'G.OOG' t = INBOX

Another application s is what we type in google and t is the intire web.

Simple algorithm

python code:

```
any (s==t [i:i+len(s)]
     for i in range(len(t)-len(s))
```



Time: $\Theta(|s| \cdot (|t| - |s|)) = O(|s| - |t|)$

How can we use Hashing to get linear time $O(|s|+|t|)$ guaranteed.

Rolling hash ADT:

- r.append(c): add char. c to the end of x
- r.skip(c) delete first char of x (assuming it is in c)

r maintains a string x

- r(): hash value of x = h(x)

Karp-Rabing algorithm:

```
for c in s: rs.append(c)
rs gives a hash value of s
for c in t[:len(s)]:
        rt.append(c)

if rs() == rt(): ...

for i in range(len(s), len(t)):
    rt.skip(t[i-len(s)])
    rt.append(t[i])
    if rs() == rt(): (Potentially can be equals)
        Check whether
        s == t[i-len(s)+1 : i+1]
        If equals found match
```

O(s)

else - happens with probability $\leq \frac{1}{|S|} \Rightarrow$

$O(1)$ expected time

$O(|s|+|t| + \#match \cdot |s|)$

rs represents hash value of s. $h(s)$

How to build ADT in order to spend constant time for each one of the operations?

division method

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

m: random prime $\geq |S|$

[The bigger m is, the higher probability complexity will be as we analyzed.]

r.append(c): shift by one = multiply by a

$q \rightarrow q \cdot a + ord(c)$

treat x as a multidigit number q in base a - alphabet size (asci = 256)

r.skip()

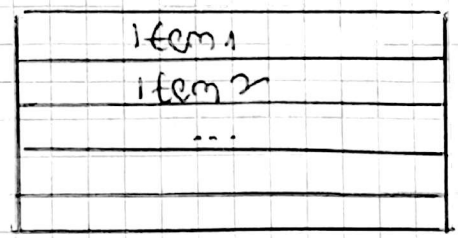
$q \rightarrow q - a^{|u|-1} \cdot c \cdot a$

Hashing:

Open addressing.
Uniform hashing analysis
Cryptographic hashing

Open addressing:

This is another approach to dealing with collisions.
no chaining



At most one per slot.

$m \geq n$

we don't have linked list we can't arbitrarily increase the storage of a slot using a chain

m - number of slots
n " " elements (keys)

Probing: (try to insert a key to a computed index of slot and if we fail - try to use a slightly different hash for the key)

Hash function: specified order of slots to probe for a key (for insert/search/delete)

hash function $h : U \times \{0,1,\dots m-1\} \to \{0,1,\dots m-1\}$ ← 2 arguments

U ← universe of keys; trial count

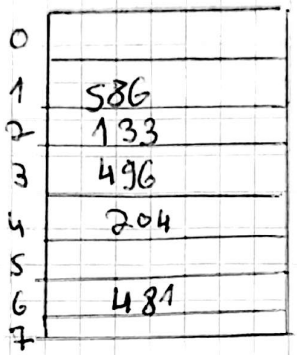
$h(k,1), h(k,2), \dots h(k,m-1)$ ← vector

arbitrary key k

to be a permutation of $\{0,1,\dots m-1\}$

In order to get access for the entirety of our hash table. Even if we have 1 slot left we want to be able to insert to this slot through our hash function.

T- Table



Insert 586 $h(586, 1) = 1$

As long as the table is empty the insertion is successful.

$h(481, 1) = 6$ ... etc

Insert (496)

$h(496, 1) = 4$ check of T index 4 =! null

fail and first probe is a fail

$h(496, 2) =$ check = 586 = fail

$h(496, 3) = 3$ success

None: empty slot

(flag)

Insert (K, V)

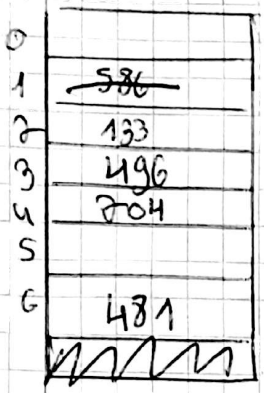
-Keep probing until an empty slot is found.

Insert item when found.

Search (k) As long as the slot encounters are occupied by keys ≠ k. Keep probing until you either encounter k or find an empty slot (you use in the same probe sequence that used for insertion.)

delete 586 :

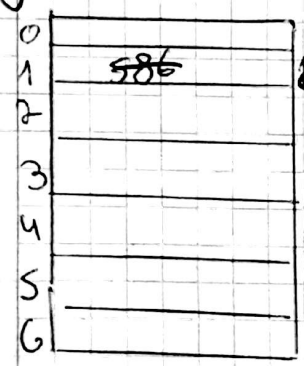
Problem.



- None -> According to the search ~~algorithm~~ algorithm search 496 is not exist!

How can we fix it?

- Replace deleted item with with DeleteMe flag (diff from none)



And the Psado code still correct.

Insert treats deleteMe the same as None but search keep koing.

We can coding this up with an array structure, is fairly straightforward

Complexity analize

Probing strategies:

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

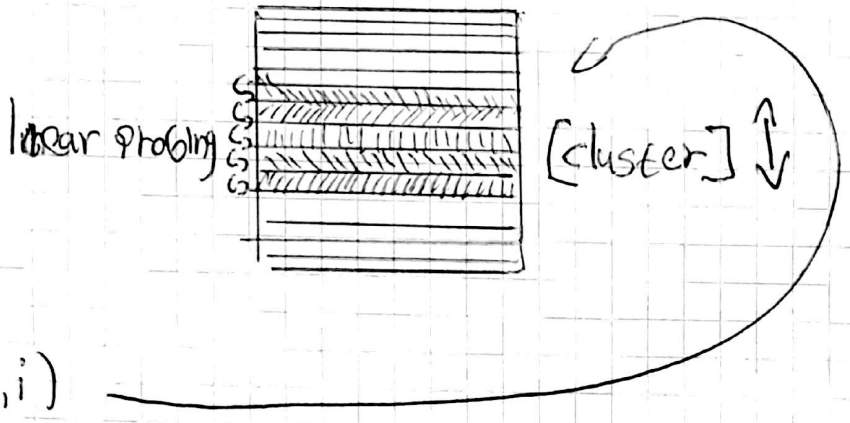
(h': ordinary hash function)

That is satisfy the permutation argument?

Yes Permutation ✓

Cluster - notion of clustering is a possibility of start getting consecutive groups of occupied slots, with keep growing

Assumption using linear probing can lead to losing average constant time



In terms of $0.01 < \alpha = \frac{n}{m} < 0.99$ $\ominus(\log n)$ size probability using linear probing can lead to $\ominus \log n$ size of insert, search which means losing the constant time advantage of hash tables [the problem is not the hash function the problem is the aspect of linearity.

Double Hashing:

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

First we have to figure out how we guarantee a permutation?

If $h_2(k)$, m relatively prime $\Rightarrow$ permutation

$m = 2^r$, $h_2(k)$ for all k is odd

Uniform Hashing Assumption X Simple Uniform Hashing

Each key is equally likely to have any one of the $m!$ permutations as it probe sequence.

[Anyone found an exact hash function that guarantee exact probability, but very close to it]

$\alpha = \frac{n}{m}$ cost of operations such as search, etc

$\leq \frac{1}{1-\alpha}$ We have to ensure that α is no more than 0.5

number of empty slots $\frac{m-n}{m} =$ $1-\alpha$ -> in term of prob $\frac{1}{1-\alpha}$

It's means that in order to preserve the size of α (no more than 0.5) we have to implement a resize function

In another way, than the chaining approach

Open addressing takes less memory -> because we don't use pointers!

Cryptographic hashes [Bonus]

Password Storage

One-way: given $h(x)$ it is very hard to find x s.t. $h(x) = a$

/etc/password : devadas : x12467
↓ login name ↓ :
: ↓

If we want to make sure that even the sys-admin doesn't know our password,

So in the file that have the password there is the login name and besides to it the hashed value of the password
So when I log in and type the password the system calcalate through the hash function that the password is right.

$h(x') = h(x)$
↑ typed in password

problems collisions...