Databases and keys

A database stores records with various attributes.

The database can be represented as a table, where each row is a record, and each column is an attribute.

Number	Name	Dept	Alias
20090612	오재훈	산디과	alpha0401
20100202	강상익	무학	scala
20100311	손호진	무학	python_is_great

row

column

Databases often designate one attribute as the key. The key has to be unique—every key appears on only one row. A table with keys is a keyed table.

We want to find records (rows) by key, so the keyed table is a map: key $\rightarrow$ record.

Integer keys

Let's make a keyed table of all the students in the class, with the student number as the key.

```
case class Student(name: String, id: Int,
                   dept: String, alias: String)
```

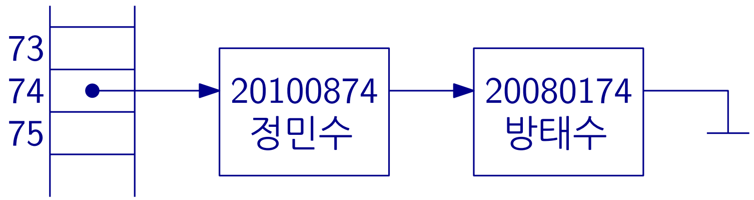
Using an array with 100 slots, we can use the last two digits of the student number as the index.

But the last two digits are not unique — we have collisions:

Number	Name	Dept	Alias
20100874	정민수	무학	ubuntu
20080174	방태수	산디과	apple

Chaining

Chaining: Each slot is actually a linked list of (key, value) pairs stored in this slot. (We need the key!)



To search for a key 20080174, we access the table at index 74, and then search through the linked list.

Analysis

We assume the hash function is good: It should distribute the items on the slots uniformly.

Analysis of hash tables assumes that the hash function is random: Each slot is equally likely to be chosen. The choices for two different items are independent.

Consider insertion/deletion/searching an item x. The running time is proportional to the length of the chain for x.

This is equal to the number of items y for which $h(y) = h(x)$. For given y, this happens with probability $1/N$. The expected value for all y is n/N.
Here n is the number of items, and N is the table size.

Load factor: The load factor λ of a hash table is n/N.
Running time is $O(\lambda)$.

Open addressing

We could make the data structure much more compact if we could avoid the linked lists and store all data in the table.

Open addressing: allow to store items at a slot different from its hash code.
Closed addressing: items must be stored at the slot given by its hash code: chaining.

Easiest form of open addressing: Linear probing.
Start at the slot given by the hash code.
If it is already in use, try the next, and continue until a free slot is found.

Linear probing

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

insert: 89 18

Linear probing

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

insert: 89 18 49

Linear probing

0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

insert: 89 18 49 58

Linear probing

0	49
1	58
2	9
3	
4	
5	
6	
7	
8	18
9	89

Find operation: Need to search sequentially until key found or empty slot found.

insert: 89 18 49 58 9

Linear probing

0	49
1	
2	9
3	
4	
5	
6	
7	
8	18
9	89

Find operation: Need to search sequentially until key found or empty slot found.

Delete operation: Slot is marked as available (can be reused at insertion, but is not the same as an empty slot).

insert: 89 18 49 58 9 delete 58

Analysis of linear probing

How far do we have to search to insert a new item?

Simplified analysis: Let's assume all slots are filled with equal and independent probability. So each slot is filled with probability $\lambda = n/N$.

The expected number of probes (slots considered) until we find a free slot is $1/(1-\lambda)$.

The load factor λ ranges from 0 (empty hash table) to 1 (completely full hash table). When it approaches 1, the hash table becomes very inefficient, and needs to be enlarged.

Real behavior of linear probing

Unfortunately, the probabilities are not independent:

Experiment 1: Fill each slot with probability $\lambda = 0.7$:

```
#### ## ### ## ### #####   # #### ####  ### #  ## ####  ###  #  # ####   ##  #####  # # # ##### ##
```

Average number of probes: 2.4

Experiment 2: Insert $\lambda * 100 = 70$ items with linear probing:

```
####  ###########  #  #  #######   ##  #####    #####  ## ###### ################# #  #   #  ### ###
```

Average number of probes: 4.4

Same with $\lambda = 0.9$: 6.9 versus 24.0

```
############ ### ####### ################## ############### ############### ## ######## # ###### ##
############################   # ###########    #############  #####################################
```

$\lambda = 0.5$	2.0	2.5
$\lambda = 0.7$	3.3	6.0
$\lambda = 0.9$	10.0	49.5
$\lambda = 0.95$	20.0	182.1
$\lambda = 0.99$	100.0	1750.5

$N = 10000$, and repeating 1000 times.

Linear probing causes clustering in the hash table.

Real analysis of linear probing

Assuming that the hash function behaves randomly, the expected number of probes for an insertion (or unsucessful search) is (for $N \to \infty$):

$$\frac{1}{2}\Big(1 + \frac{1}{(1-\lambda)^2}\Big)$$

Linear probing works very well when the hash function is good and the load factor λ is small, say $\lambda \leq 0.5$.

Linear probing is more sensitive to bad hash functions than chaining.

Load factor includes items that have been deleted! When there are too many deleted items, we need to rehash the table.

Hash functions

We typically use two functions:

Hash code

h_1 : keys $\rightarrow$ integers

Compression function

h_2 : integers $\rightarrow [0, N-1]$

Index in hash table is computed as $h_2(h_1(key))$.

Ideally, the hash function should map keys uniformly at random to an index into the hash table.

Resizing hash tables: We change the compression function only, and then need to rehash all elements.

Compression functions

Hash codes and compression functions are a bit of a black art. It is easy to mess up.

An obvious compression function is $h_2(x) = x \bmod N$.

It only works well if N is a prime number.

A better compression function is

$$h(x) = ((ax + b) \bmod p) \bmod N,$$

where a, b, and p are positive integers, p is a large prime, and $p \gg N$. N does not need to be prime.

Hash Codes

A good hash code for strings:

```
def hashCode(key: String): Int = {
  var hash = 0
  for (ch <- key)
    hash = (127 * hash + ch) % 16908799
  hash
}
```

Mix up the bits

Each character has different effect.

Bad hash codes:

- Sum up the codes of the letters (too small, and anagrams collide).
- Take the first three letters ("pre" is common, "xzq" never occurs).

Why is the function above good? Because it works in practice...

Hash codes and equality

Scala `HashSet` and `HashMap` compute a hash code by calling the element's `##` method.
Every Scala object implements `##`.

`HashSet` and `HashMap` only work correctly if the following "contract" is observed:
If `obj1 == obj2` then `obj1.## == obj2.##`.

This is true for all standard types, but needs to be done by the programmer for new types!

The default implementation of `##` simple returns the memory address of the object on the heap.

Mutable keys are dangerous! If you change a key in the hash table, you cannot find it anymore.

Alternatives to linear probing

Quadratic probing: Try slots $i + j^2$, for $j = 0, 1, 2, \ldots$.

Secondary hash function: Try slots $i + jd(k)$, where $d(k)$ is a secondary hash function.

The details are tricky, because we need to make sure that the probing will find an empty slot.
Before you implement this, read a good book!

Cuckoo hashing

Hashing with guaranteed constant search time!

We need two independent hash functions $h(x)$ and $g(x)$.

To insert item x, check if slot $h(x)$ or $g(x)$ is empty. If so, insert the item there.

Otherwise, let y be the item at position $g(x)$. Insert x at slot $g(x)$, and move y to its other possible slot.

To find an item, we only need to check two slots!

If the load factor is small enough ($\lambda \approx 1/3$), then insertions take expected constant time.

Hash table or binary search tree?

Hash tables do not support order on the items.

Hashing is fast if the hash function can be computed quickly.

Typical applications of hashing:

- symbol tables (in a compiler etc.),
- small databases,
- remembering positions (in a game tree),
- caching data (in a browser etc.),
- dictionaries.