CSC263 Week 5

Larry Zhang

http://goo.gl/forms/S9yie3597B

Announcements

PS3 marks out, class average 81.3%

Assignment 1 due next week.

Response to feedbacks -- tutorials

> “We spent too much time on working by ourselves, instead of being taught by the TAs.”
> We intended to create an “active learning” atmosphere in the tutorials which differs from the mostly “passive learning” atmosphere in the lectures. If that’s not working for you after all, let me know through the weekly feedback form and we will change!

Foreseeing February

Feb 10: A1 due

Feb 16: Reading week

Feb 16~25: Larry out of town

Tuesday Feb 24: Lecture by Michelle

Thursday Feb 26: Lecture at exceptional location **RW110**

Thursday Feb 26: 11am-1pm, 2pm-4pm - Pre-test office hour at BA5287, **4pm-6pm midterm**

Office hours while Larry's away

- ➔ Francois (MTWRF 1:30-2:30), Michelle (MW 10:30-12)
- ➔ **Please go to these office hours to have your questions answered! (or email Larry)**

Data Structure of the Week

Hash Tables

Hash table is for implementing Dictionary

	unsorted list	sorted array	Balanced BST	Hash table
Search(S, k)	O(n)	O(log n)	**O(log n)**	**O(1)**
Insert(S, x)	O(n)	O(n)	**O(log n)**	**O(1)**
Delete(S, x)	O(1)	O(n)	**O(log n)**	**O(1)**

average-case, and if we do it right

Direct address table

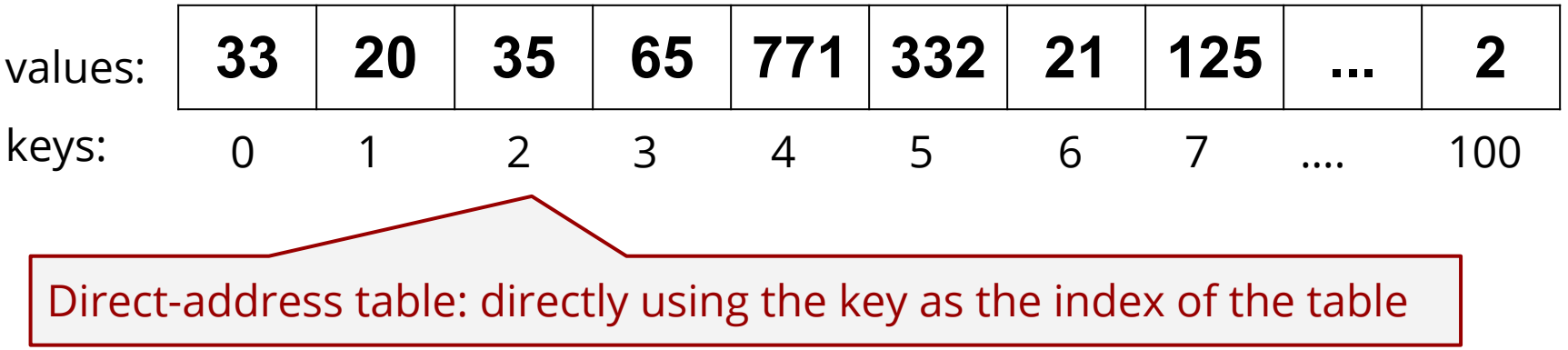
a fancy name for “array”…

Problem

Read a grade file, keep track of number of occurrences of each grade (integer 0~100).

The fastest way: create an array **T[0, ..., 100]**, where **T[i]** stores the number of occurrences of grade **i**.

Everything can be done in O(1) time, worst-case.



The drawbacks of direct-address table?

values:	33	20	35	65	771	332	21	125	...	2
keys:	0	1	2	3	4	5	6	7	....	100

Drawback #1: What if the keys are **not integers**? Cannot use keys as indices anymore!

We need to be able to **convert** any type of key to an **integer**.

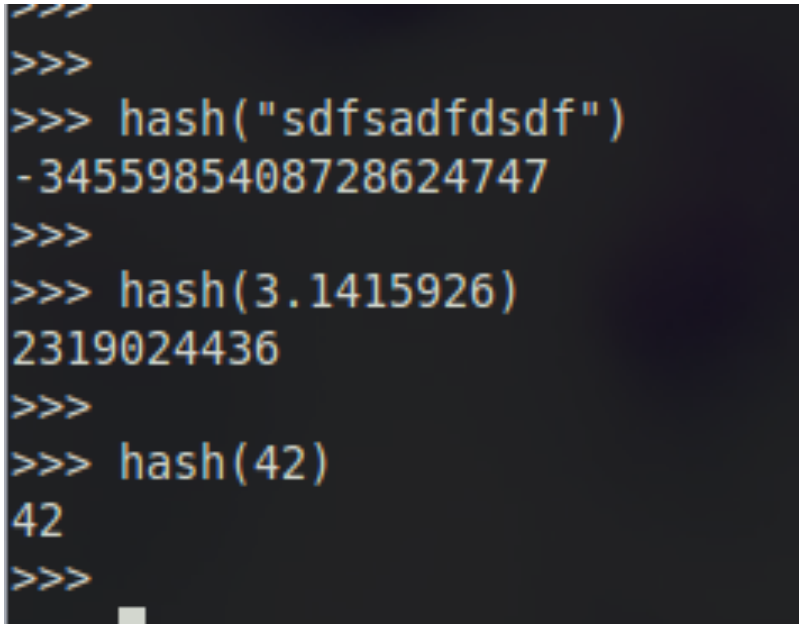
Drawback #2: What if the grade 1,000,000,000 is allowed? Then we need an array of size **1,000,000,001**! Most space is **wasted**.

We need to map the **universe** of keys into a small number of **slots**.

A **hash function** does both!

An unfortunate naming confusion

Python has a built-in “**hash()**” function



By our definition, this “**hash()**” function is not really a **hash function** because it only does the first thing (convert to integer) but **not** the second thing (map to a small number of slots).

Definitions

Universe of keys *U*, the set of all possible keys.

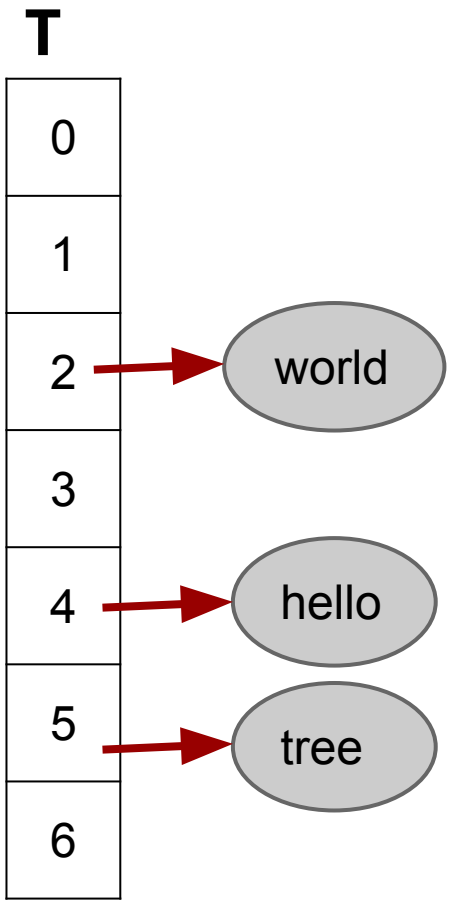
Hash Table *T*: an array with ***m*** positions, each position is called a "**slot**" or a "**bucket**".

Hash function *h*: a functions maps ***U*** to **{0, 1, ..., m-1}**

in other words, ***h(k)*** maps any key ***k*** to one of the ***m*** buckets in table ***T***.

in yet other words, in array ***T***, ***h(k)*** is the the index at which the key ***k*** is stored.

Example: A hash table with m = 7



Insert(“hello”)
assume h(“hello”) = 4

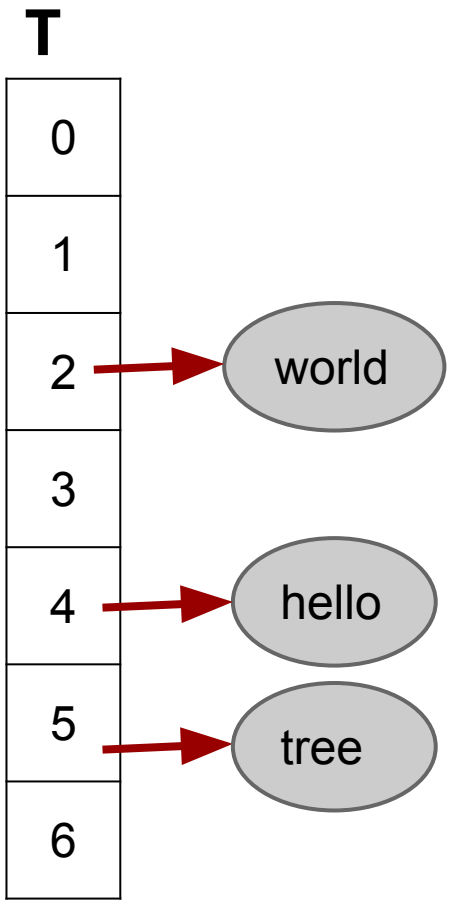
Insert(“world”)
assume h(“world”) = 2

Insert(“tree”)
assume h(“tree”) = 5

Search(“hello”)
return **T[h(“hello”)]**

What’s new potential issue?

Example: A hash table with m = 7



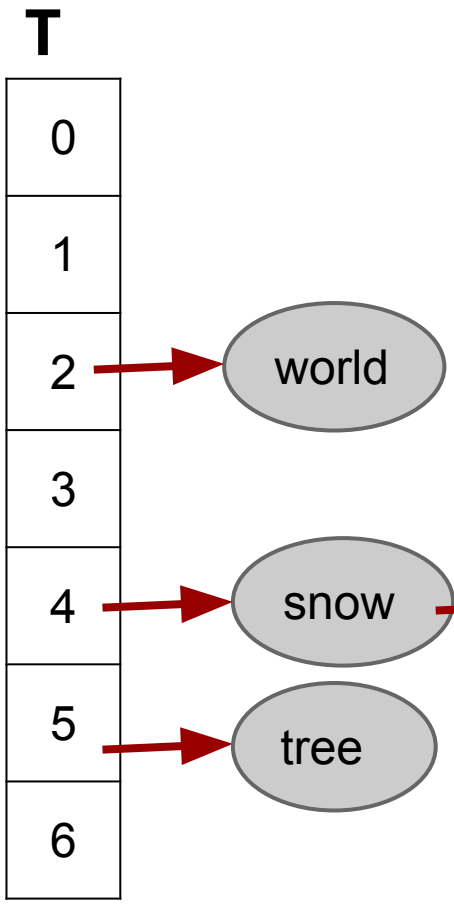
What if we Insert("snow"), and h("snow") = 4?

Then we have a **collision**.

One way to resolve collision is

Chaining

Example: A hash table with m = 7



What if we Insert("snow"), and h("snow") = 4?

Then we have a **collision**.

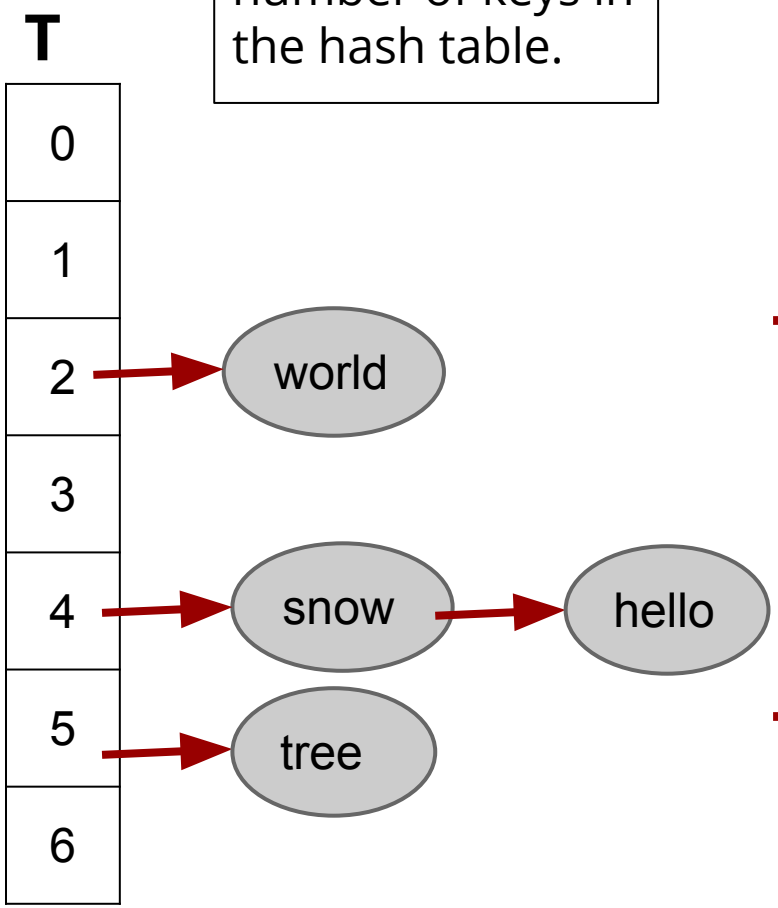
Store a **linked list** at each bucket, and insert new ones at the **head**

One way to resolve collision is

Chaining

Hashing with chaining: Operations

Let **n** be the total number of keys in the hash table.



- **Search(k):**
  - Search k in the linked list stored at **T[h(k)]**
  - Worst-case **O(length of chain),**
  - Worst length of chain: **O(n)** (e.g., all keys hashed to the same slot)
- **Insert(k):**
  - Insert into the linked list stored at **T[h(k)]**
  - Need to check whether key already exists, still takes **O(length of chain)**
- **Delete(k)**
  - Search k in the linked list stored at **T[h(k)]**, then delete, **O(length of chain)**

Hashing with chaining operations, worst-case running times are O(n) in general. Doesn't sound too good.

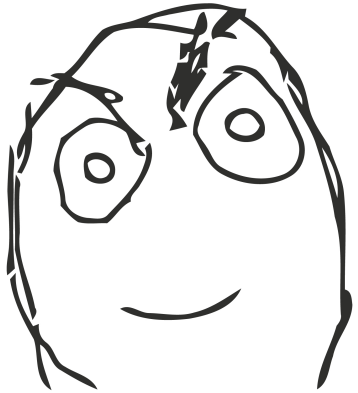
However, in practice, hash tables work really well, that is because

➔ The worst case almost never happens.

➔ **Average case** performance is **really** good.

In fact, Python "**dict**" is implemented using hash table.

Average-case analysis: Search in hashing with chaining



Assumption: Simple Uniform Hashing

Every key **k** ∈ **U** is **equally likely** to hash to any of the **m** buckets.

For any key **k** and any bucket **j**

$$\Pr(h(k) = j) = \frac{1}{m} = \sum_{k \in U : h(k) = j} \Pr(k)$$

Given a key k, each of the **m** slots is equally likely to be hashed to, therefore **1/m**

Out of all keys in the universe, **1/m** of the keys will hash to the given slot ***j***

Let there be **n** keys stored in a hash table with **m** buckets.

Let L_j be the length of the linked list at slot j

$$\text{then } \sum_{j=0}^{m-1} L_j = n$$

Let random variable **N(k)** be the number of elements examined during search for **k**, then average-running time is basically (sort-of) **E[N(k)]**

$$E[N(k)] = \sum_{k \in U} \Pr(k) \cdot N(k)$$

$$= \sum_{j=0}^{m-1} \sum_{k \in U : h(k) = j} \Pr(k) \cdot N(k)$$

$$\leq \sum_{j=0}^{m-1} \sum_{k \in U : h(k) = j} \Pr(k) \cdot L_j$$

$$= \sum_{j=0}^{m-1} \frac{1}{m} \cdot L_j$$

$$= \frac{1}{m} \sum_{j=0}^{m-1} L_j \quad = \frac{n}{m}$$

Dividing the universe into **m** parts

N(k) <= Lj, at most examine all elements in the chain

$$\Pr(h(k) = j) = \frac{1}{m}$$

$$\frac{1}{m} = \sum_{k \in U : h(k) = j} \Pr(k)$$

$$\sum_{i=0}^{m-1} L_i = n$$

$$E[N(k)] \leq \frac{n}{m}$$

$$\text{Let } \alpha = \frac{n}{m}$$

and call it the **load factor**
(average number of key per bucket, i. e., the average length of chain)

Add 1 step for the hashing **h(k)**,

then the average-case running time for Search is in at most **1+α** (O(1+α))

By a bit more proof, we can show that it's actually **Θ(1+α)**

A bit more proof: average-case runtime of a successful search (after-class reading)

Assumption: k is a key that **exists** in the hash table

The number of elements examined during search for a key k

= 1 + number of elements before x in chain

The successful comparison when found k

The comparisons that return false

= 1 + number of keys that hash samely as k and are inserted after k

so it's **before** x in the chain (we insert at the head)

so in the **same** chain as x

Proof continued…

Let k1, k2, k3, …, kn be the order of insertion

Define $X_{ij} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$

$$E[X_{ij}] = \frac{1}{m}$$

because simple uniform hashing

then, the expectation

E[number of keys that hash **samely** as a key k and are inserted **after** k]

$$= E\left[\frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} X_{ij}\right] = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} E[X_{ij}] = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{m}$$

average over all keys ki

sum over all keys kj inserted **after** ki

$$= \frac{1}{nm}\sum_{i=1}^{n}(n-i) = \cdots = \frac{\alpha}{2} - \frac{\alpha}{2n}$$

So overall, average-case runtime of successful search: $1 + E[\text{number} \ldots] = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$

$\Theta(1+\alpha)$ $\Theta(1 + n/m)$

If **n < m**, i.e., more slots than keys stored, the running time is **Θ(1)**

If **n/m** is in the order of a constant, the running time is also **Θ(1)**

If **n/m** of higher order, e.g., **sqrt(n),** then it's not constant anymore.

So, in practice, choose m wisely to guarantee constant average-case running time.

We made an important assumption…

Simple Uniform Hashing

Can we really get this for real?

Difficult, but we try to be as close to it as possible.

Choose good hash functions => Thursday

CSC263 Week 5

Thursday

Announcements

Don’t forget office hours (A1 due next week)

Thu 2-4pm, Fri 2-4pm, Mon 4-5:30pm

or anytime when I’m in my office

New question in our **Weekly Feedback Form**:

What would make the slides awesome for self-learning?

What features would you like to have, so that you don’t need to go to lectures anymore? Feel free to be creative and unrealistic.

http://goo.gl/forms/S9yie3597B

New “**tips of the week**” updated as usual.

Recap

- **Hash table**: a data structure used to implement the Dictionary ADT.
- **Hash function h(k)**: maps any key k to {0, 1, …, m-1}
- Hashing with **chaining**: average-case O(1+α) for search, insert and delete, assuming **simple uniform hashing**

Simple Uniform Hashing

All keys are **evenly** distributed to the **m** buckets of the hash table, so that the lengths of chains at each bucket are the same.

➔ Think about inserting English words from a document into the hash table

We **cannot** really **guarantee** this in practice, we don't really the distribution from which the keys are drawn.

➔ e.g., we cannot really tell which English words will actually be inserted into the hash table before we go through the whole document.

➔ so there is **no way** to choose a hash function **beforehand** that guarantees all chains will be equally long (simple uniform hashing).

So what can we do?

We use some **heuristics**.

Heuristic

(noun)

A method that works in practice
but you don’t really know why.

First of all

Every object stored in a computer can be represented by a **bit-string** (string of 1's and 0's), which corresponds to a (**large**) **integer**, i.e., any type of key can be converted to an **integer** easily.

So the only thing a **hash function** really needs to worry about is how to **map** these large integers to a small set of integers **{0, 1, …, m-1}**, i.e., the buckets.

What do we want to have in a hash function?

Want-to-have #1

h(k) depends on **every bit** of **k**,

so that the differences between different **k**'s are fully considered.

h(k) = lowest 3-bits of k
e.g.,
h(1010010100010**10**) = **2**

bad

h(k) = sum of all bits
e.g.,
h(1010010100010**10**) = **6**

a little better

Want-to-have #2

h(k) "**spreads out**" values, so all buckets get something.

Assume there are m = 263 buckets in the hash table.

h(k) = k mod 2

bad

because all keys hash to either bucket 0 or bucket 1

h(k) = k mod 263

better

because all buckets could get something

Want-to-have #3

h(k) should be **efficient to compute**

h(k) = solution to the PDE ***$^%** with parameter k

Yuck!

h(k) = k mod 263

better

1. h(k) depends on every bit of k
2. h(k) “spreads out” values
3. h(k) is efficient to compute

In practice, it is difficult to get all three of them, ...

but there are some **heuristics** that work well

The division method

The division method

h(k) = k mod m

h(k) is between 0 and m-1, apparently

Pitfall: sensitive to the value of **m**

➔ if m = 8, …

- h(k) just returns the lowest 3-bits of k

➔ so **m** better be a **prime number**

- That means the size of the table better be a prime number, that's kind-of restrictive!

A variation of the division method

h(k) = (ak + b) mod m

where a and b are constants that can be picked

Used in "**Universal hashing**" (see textbook 11.3.3 if interested)

➔ achieve simple uniform hashing and fight malicious adversary by choosing randomly from a set of hash functions.

The multiplication method

The multiplication method

$$h(k) = \lfloor m \cdot (kA \mod 1) \rfloor$$

x mod 1 returns the **fractional** part of **x**

with “magic constant” 0 < A < 1

like, A = 0.45352364758429879433234

We “mess-up” **k** by multiplying **A**, take the fractional part of the “mess” (between **0** and **1**), then multiply **m** to make sure the result is between **0** and **m-1**.

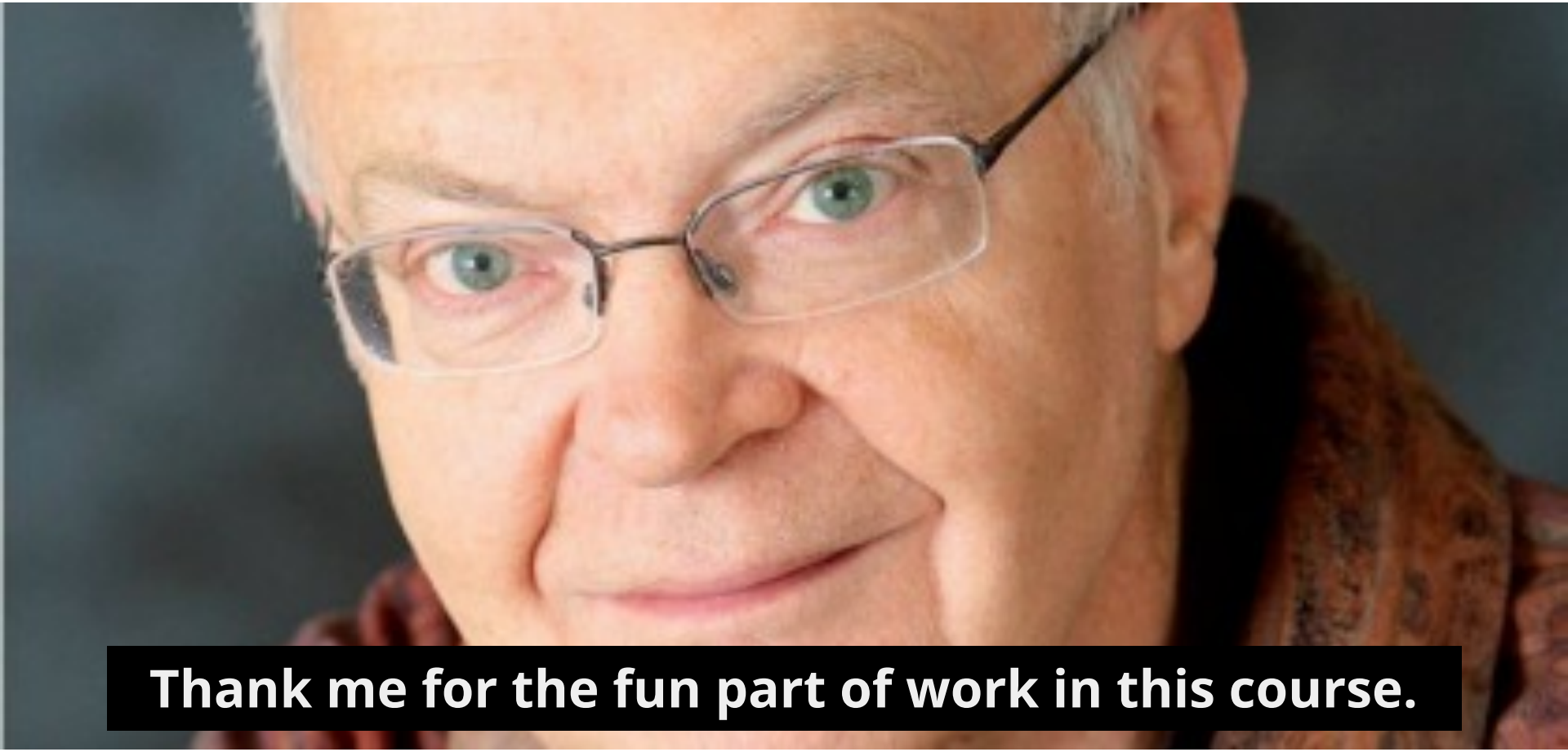
Tends to evenly distribute the hash values, because of the “mess-up”.
Not sensitive to the value of **m**, unlike division method

Magic A suggested by **Donald Knuth**: $A = \frac{\sqrt{5}-1}{2} = 0.618\ldots$

Donald Knuth

The “father of analysis of algorithms”

Inventor of LaTeX



Summary: hash functions

Hash

(noun)

a dish of cooked meat cut into small pieces and cooked again, usually with potatoes.

(verb)
make (meat or other food) into a hash

"The spirit of hashing"

Open addressing

another way of resolving **collisions**
other than chaining

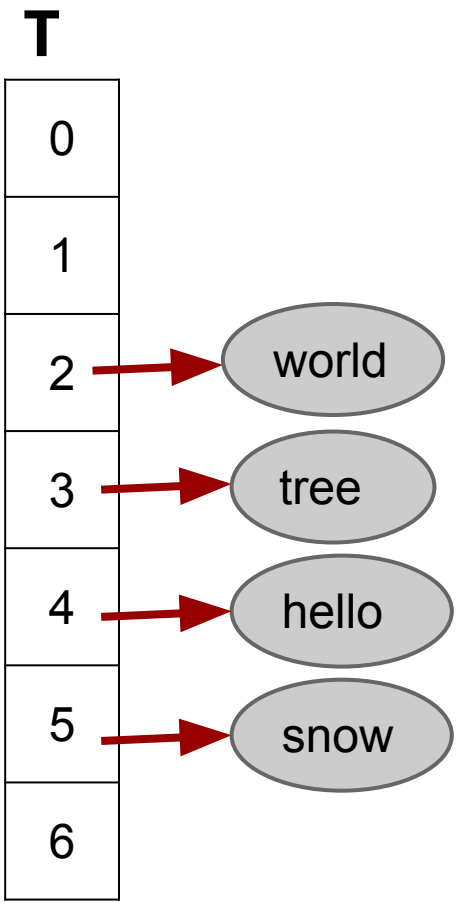
Open addressing

- ➔ There is no chain
- ➔ Then what to do when having a **collision**?
  - ◆ Find **another bucket** that is **free**
- ➔ How to find another bucket that is free?
  - ◆ We **probe**.
- ➔ How to probe?
  - ◆ **linear** probing
  - ◆ **quadratic** probing
  - ◆ **double hashing**

Linear probing

Probe sequence:

`(h(k) + i) mod m, for i=0,1,2, ...`



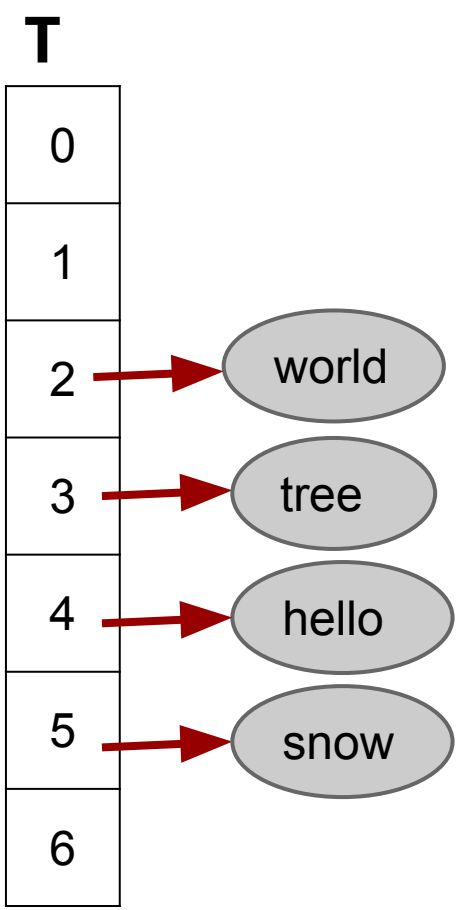
Insert("hello")
assume h("hello") = 4

Insert("world")
assume h("world") = 2

Insert("tree")
assume h("tree") = 2
probe 2, 3 ok

Insert("snow")
assume h("snow") = 3
probe 3, 4, 5 ok

Problem with linear probing



Keys tend to **cluster**, which causes **long runs** of probing.

Solutions: Jump **farther** in each probe.

before: h(k), h(k)+1, h(k)+2, h(k)+3, ...

after: h(k), h(k)+1, h(k)+4, h(k)+9, ...

This is called quadratic probing.

Quadratic probing

Probe sequence

$(h(k) + c_1 i + c_2 i^2) \bmod m$, for $i=0,1,2,...$

Pitfalls:

- ➔ Collisions still cause a milder form of **clustering**, which still cause **long runs** *(keys that collide jump to the same places and form crowd)*.
- ➔ Need to be careful with the values of c_1 and c_2, it could jump in such a way that some of the buckets are never **reachable**.

Double hashing

Probe sequence:

$(h_1(k) + ih_2(k)) \bmod m$, for $i=0,1,2,...$

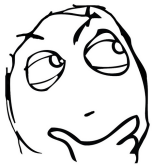
Now the jumps almost look like random, the jump-step ($h_2(k)$) is different for different **k**, which helps avoiding clustering upon collisions, therefore avoids long runs *(each one has their own way of jumping, so no crowd)*.

Performance of open addressing

Assuming simple uniform hashing, the average-case **number of probes** in an **unsuccessful** search is 1/(1-α).

For a **successful** search it is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

In both cases, assume α < 1



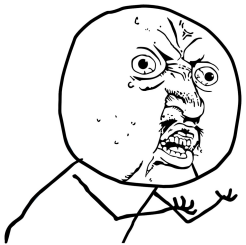
Open addressing cannot have α > 1. Why?

How exactly to do Search, Insert and Delete work in an open-addressing hash table?

Will see in this week's tutorial.

Next week

➔ Randomized algorithms



http://goo.gl/forms/S9yie3597B