CSC263 Winter 2020

Hash Tables

Week 5

Hash Table implementing ADT 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)**

worst-case **average-case**

Hash table Applications

- Python **dict** is implemented using hash table
- C++ **unordered_map**
- **Spell checkers**: just modify one letter of the wrong word at a time, and lookup a dictionary implemented by hash table. This naive algorithm works well because hash table lookup is so fast.
- Hashing is widely used in cryptography and integrity verification (e.g., checksum with MD5, SHA)
- Database indexing, cache...

Direct Address Table

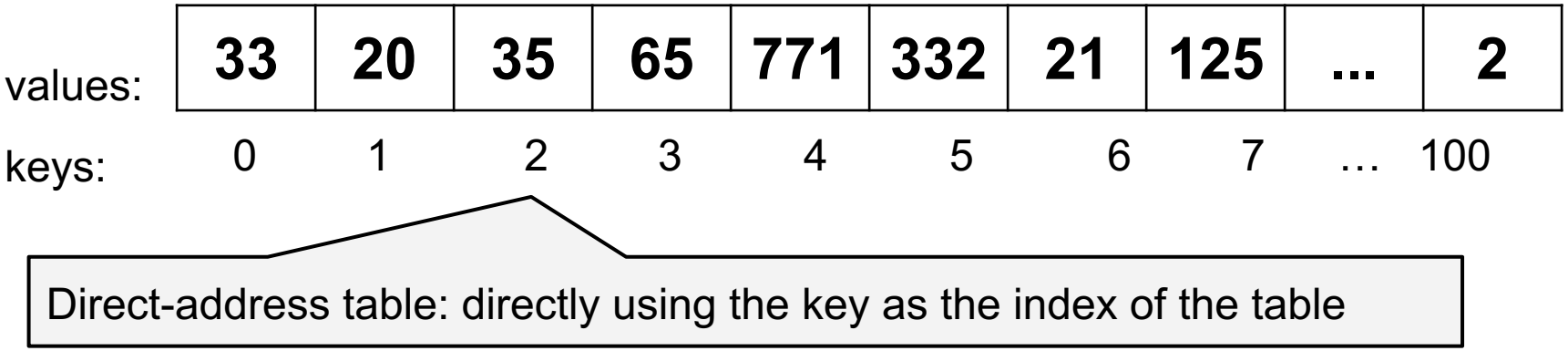
the formal term for ***array***

Example: Problem

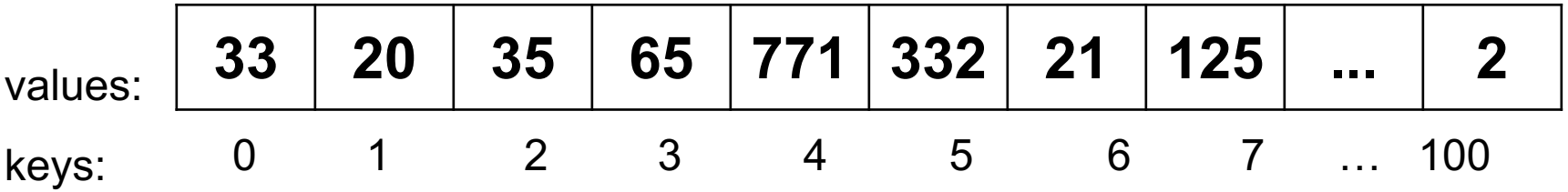
Read a grade file, keep track of number of occurrences of each grade (integer 0~100).

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

`Search, Insert, Delete` in O(1) time, worst-case.



The drawbacks of direct-address table?



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.

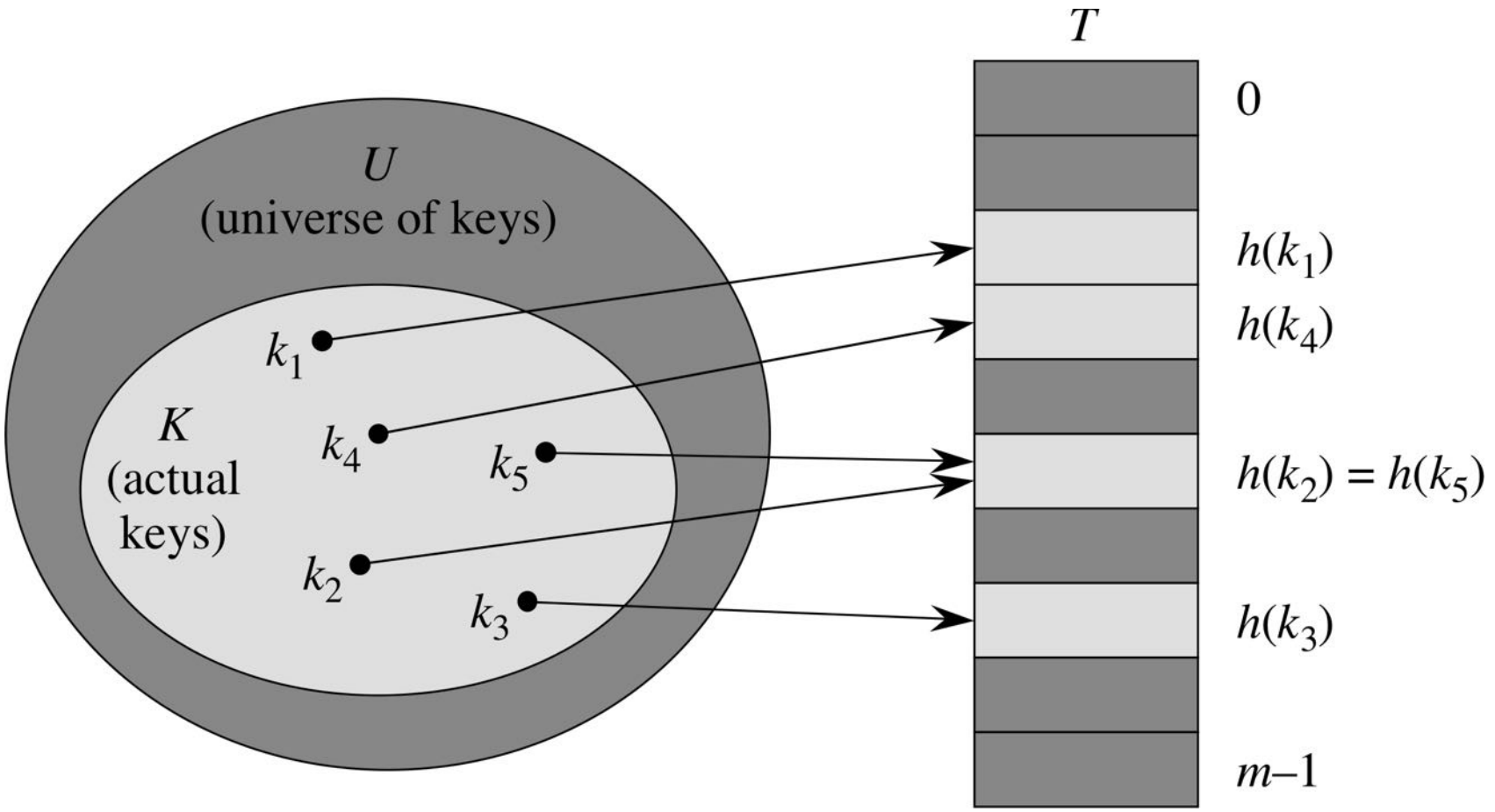
We need to map the **universe** of keys into a small number of **slots**.

A **hash function** does both!

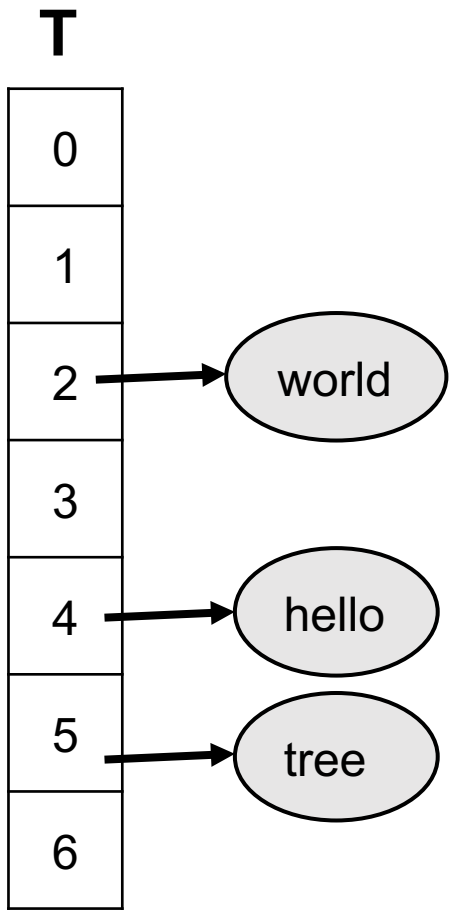
Definitions

Universe 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 deterministic function mapping **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.

Hash Table



Example: A hash table with m = 7



Insert(“hello”)
assume h(“hello”) = 4

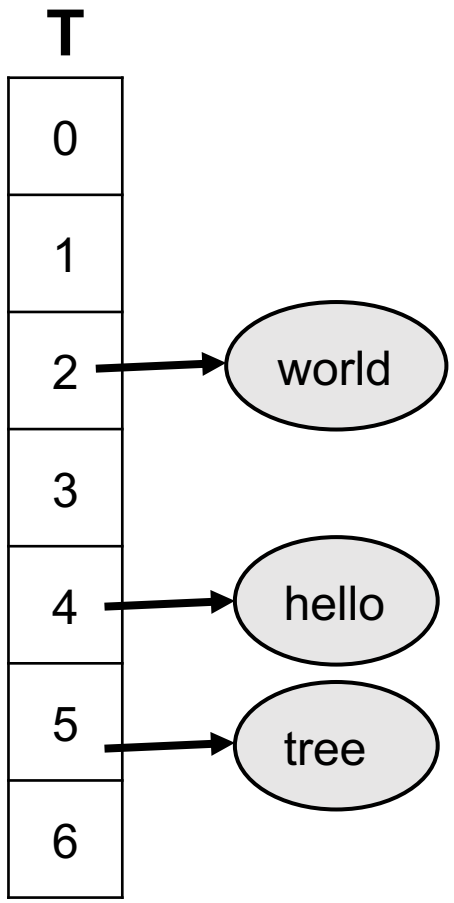
Insert(“world”)
assume h(“world”) = 2

Insert(“tree”)
assume h(“tree”) = 5

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

What’s the concern?

Example: A hash table with m = 7

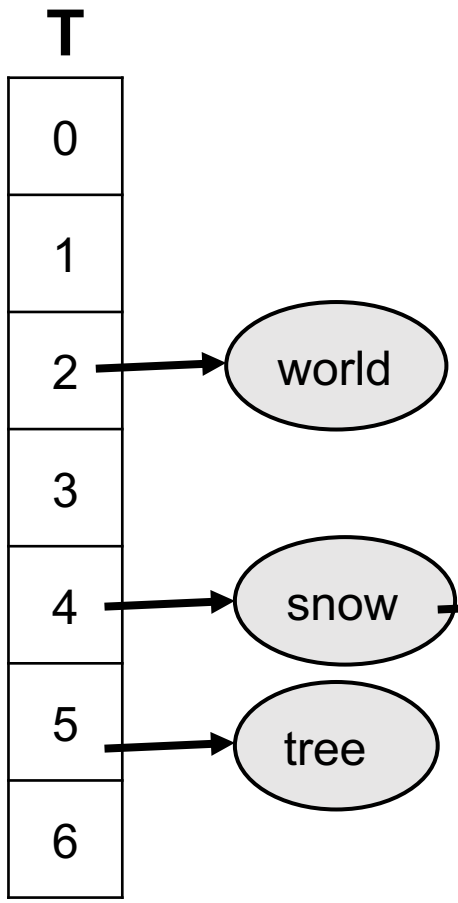


What if we Insert("snow"), and h("snow") = 4?

Collision Two distinct keys hash to the same location.

One way to resolve collisions is **Chaining**

Example: A hash table with m = 7



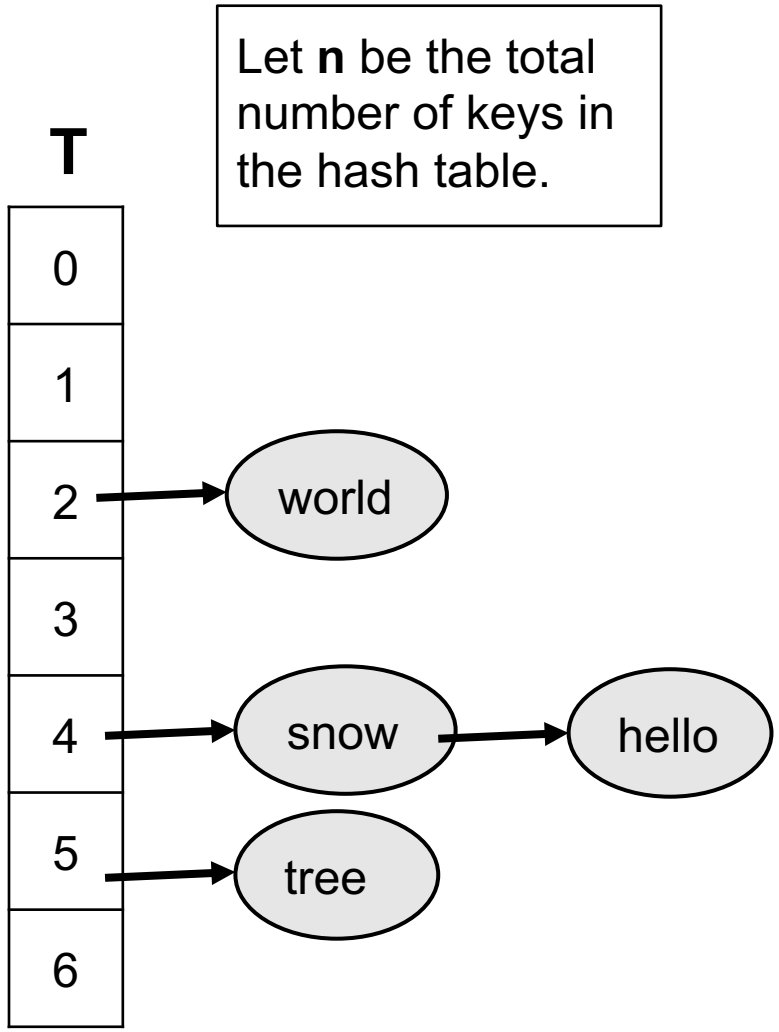
What if we Insert("snow"), and h("snow") = 4?

Collision Two distinct keys hash to the same location.

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

One way to resolve collisions is **Chaining**

Hashing with Chaining: Operations



`Search(k)`

- Search k in the linked list stored at **T[h(k)]**
- Worst-case **O(length of chain),**

`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)**

How bad can “length of chain” be?

O(n)

i.e. all keys hash to the same slot

Hashing with Chaining

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.

`Search` in Hashing with Chaining

Average-case analysis

Simple Uniform Hashing Assumption

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

Intuition: If we insert n keys into the m slots with simple uniform hashing, the chains at every slot will have on average the same length, so the **average length** of each chain would be

$$\frac{\#\text{keys}}{\#\text{slots}} = \frac{n}{m}$$

Simple Uniform Hashing Assumption

formalized

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

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

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

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

This is **independent** of all other keys $k \neq k'$.

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

Average-case Running Time

$$E[N(k)] \leq \sum_{k' \in T} \Pr\big(h(k) = h(k')\big)$$

$$\leq 1 + \sum_{k' \in T} \Pr\big(h(k) = h(k')\big) \qquad k \neq k'$$

$k \in T$

Every key in T which collided

$$\leq 1 + \frac{n}{m}$$

$$E[N(k)] \leq 1 + \frac{n}{m}$$

Load factor $\alpha = \frac{n}{m}$

average number of keys per bucket, i.e., the average length of chain

Add O(1) steps for calculating **h(k)**, accessing **T**

Average-case running time for `Search`
is in at most $1 + \alpha$ ($O(1 + \alpha)$)
(it's an upper bound since we also consider the case of unsuccessful search, i.e. have to go through the chain)

With a little more work, we can show that it is actually **$\Theta(1+\alpha)$**

Bonus Proof
Average-case Runtime of a Successful Search

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

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]

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

average over all keys ki

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

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

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 of the order of a constant, the running time is still **Θ(1)**

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

In practice

- choose m (size of the table) wisely to guarantee constant average-case running time
- grow/shrink the table dynamically

7 min

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

Choose a good hash function

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 book into the hash table

We **cannot** really **guarantee** this in practice, we don't know 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 nobody really knows why.

First of all: Converting key to integer

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

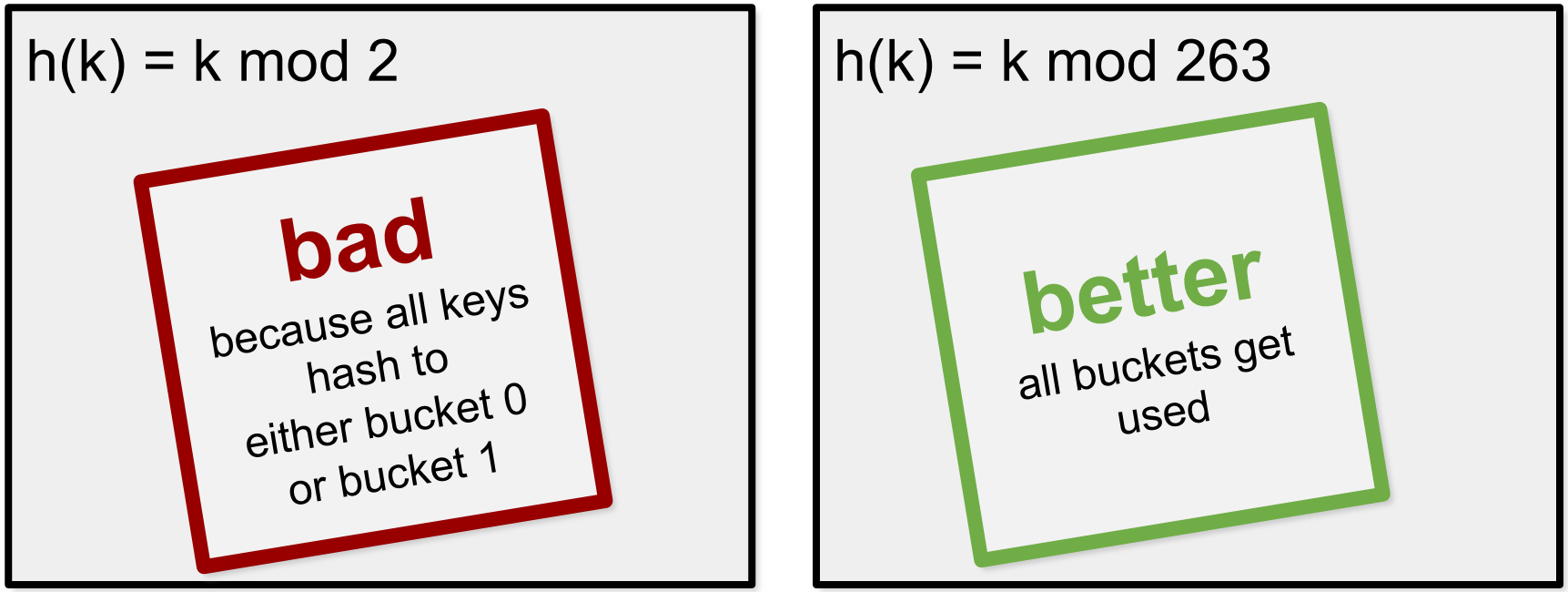
h(k) = sum of all bits
e.g.,
h(1010010100010**10**) = **6**

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.



Want-to-have #3

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

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

Terrible!

h(k) = k mod 263

better

A good Function

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

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

h(k) is between 0 and m-1

Pitfall: Sensitive to the value of m

- if m = 8: h(k) just returns the lowest 3-bits of k
- so m is preferably a prime number other than 2
  That means the size of the table better be a prime number,
  that's kind-of restrictive!

A Variation of the Division Method

$$h(k) = (ak + b) \bmod m$$

where a and b are constants picked randomly

Used in "**Universal hashing**" (see CLRS 11.3.3)

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

The Multiplication Method

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

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

with constant 0 < A < 1

e.g. 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**.

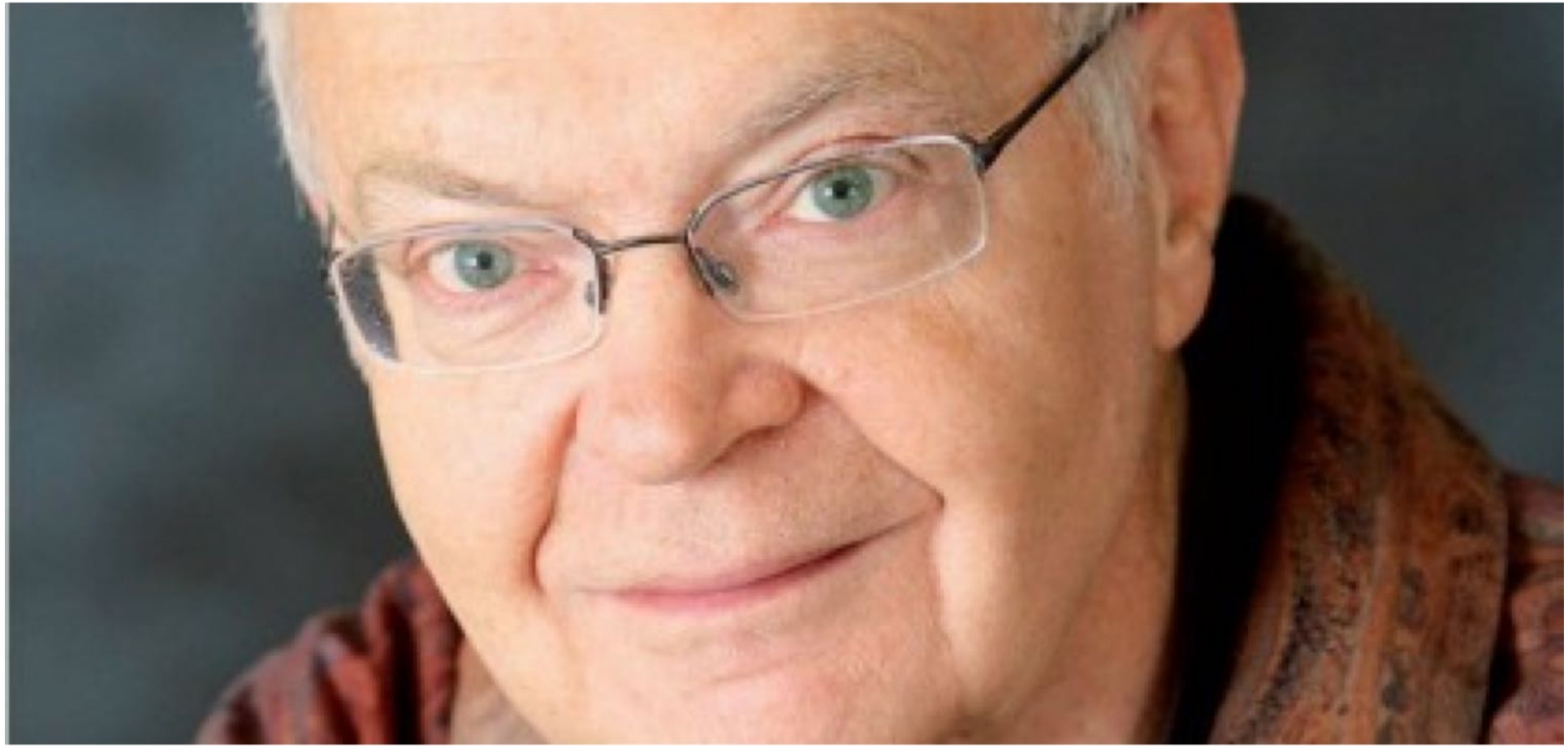
Tends to evenly distribute the hash values, because of the "mess-up". Not sensitive to the value of **m**.

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

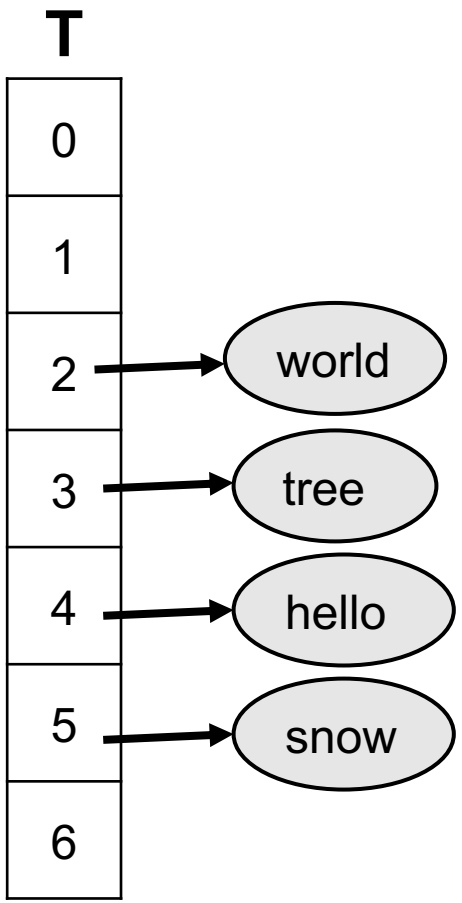
- There is no chain
- Then what to do when having a collision?
  - Find **another bucket** that is **free**
- How to find another bucket that is free?
  - We **probe**.
- How to probe?
  - **linear** probing
  - **quadratic** probing
  - **double hashing**

Requirement

For every key **k** the **probe sequence** is a **permutation** of <0,1,…,m-1>.

Linear probing

Probe sequence h(k,i) = **(h(k) + i) mod m**, for i=0,1,2, ...



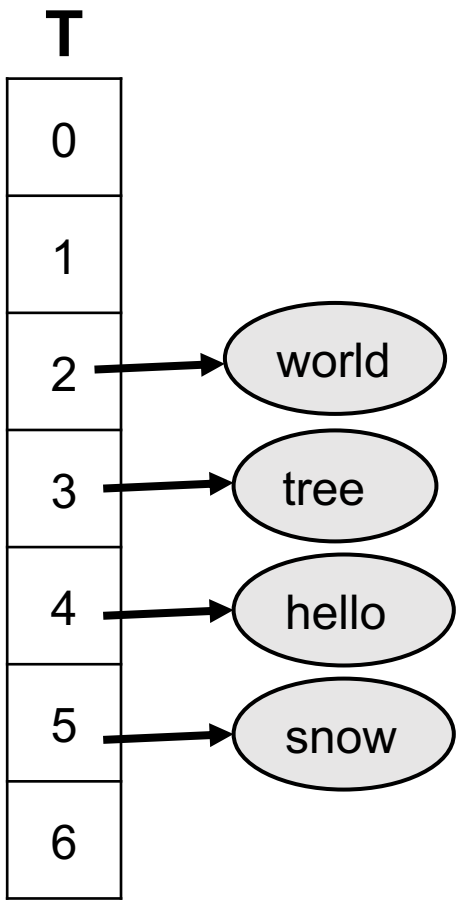
Insert("hello")
assume h("hello") = 4

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



Primary Clustering

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

Solution

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

Pitfalls

- Collisions still cause a milder form of clustering, i.e. **secondary clustering**, which still cause **long runs** (keys that collide jump to the same places and form cluster)
- 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(k,i) = $(h_1(k) + ih_2(k)) \bmod m$, for i=0,1,2,...

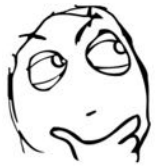
Now the jumps almost look like random, the jump-step ($h_2(k)$) is different for different **k**, which helps avoiding clustering upon collisions, therefore avoids long runs (each one has their own way of jumping, so no clustering).

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

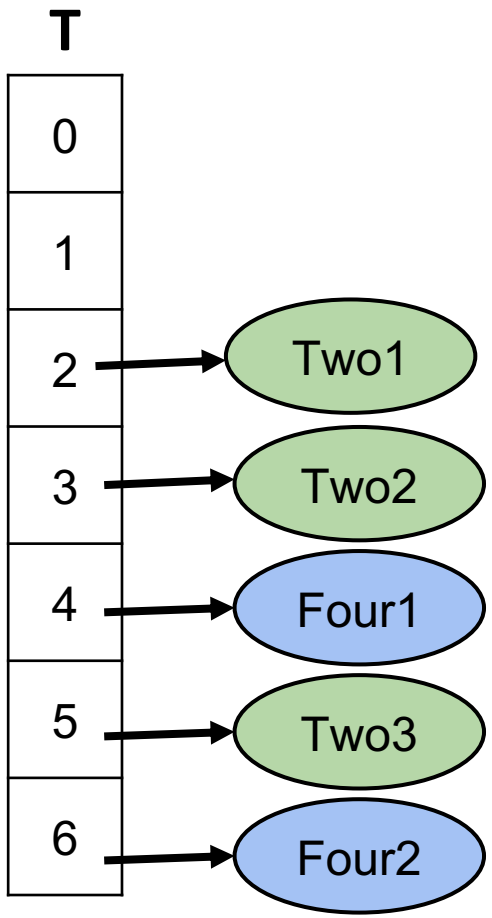


Open addressing cannot have α > 1. Why?

Search, Insert, Delete

in an open-addressing hash table

Insert (assume linear probing)



Insert(“Four2”)

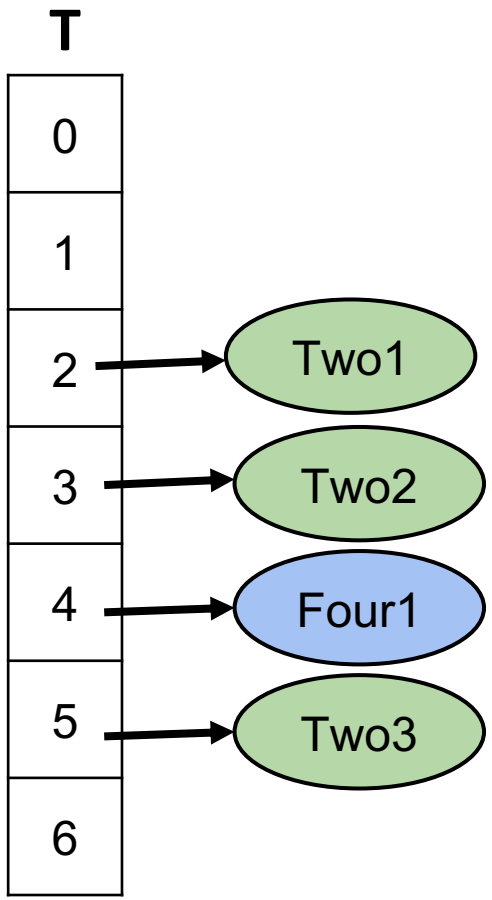
check T[**h(“Four2”)=4**]: occupied

check T[**h(“Four2”)+1=5**]: occupied

check T[**h(“Four2”)+2=6**]: free slot!

Insert there

Search (assume linear probing)



Search(k = "Two3"):

check T[**h("Two3")=2**]: not yet

check T[**h("Two3")+1=3**]: not yet

check T[**h("Two3")+2=4**]: not yet

check T[**h("Two3")+3=5**]: **found!**

Search(k="Four2"):

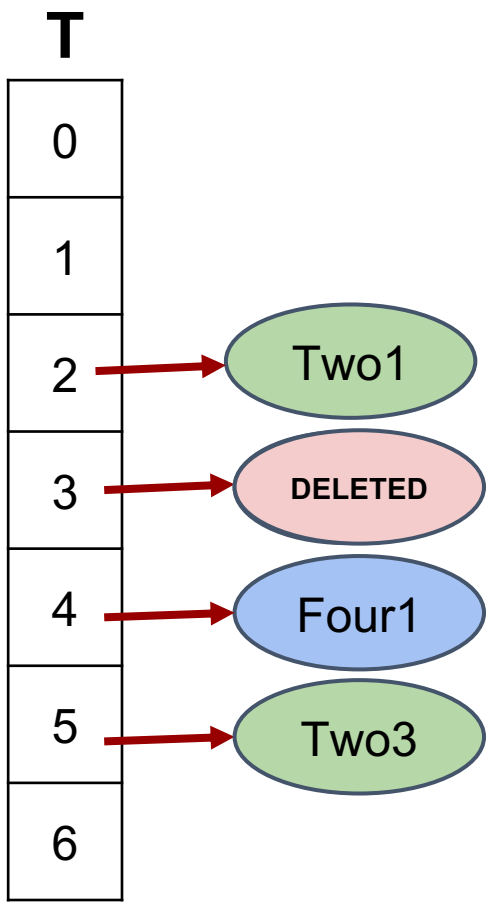
check T[**h("Four2")=4**]: not found

check T[**h("Four2")+1=5**]: not found

check T[**h("Four2")+2=6**]: a **free slot**!

Can **give up** now, **"Four2"** is NOT there

Delete (assume linear probing)



Delete("Two2")

check T[**h("Two2")=2**]: not yet
check T[**h("Two2")+1=3**]: found
Just delete it?

No, if T[3] becomes a free slot, next time **Search(Two3)** will give up by mistake.

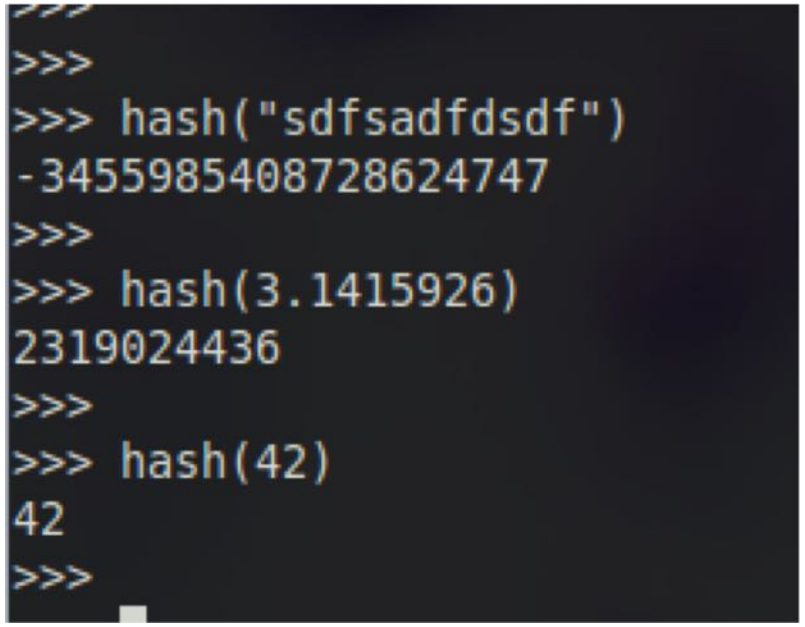
Solution: let T[3] store a special node called "**deleted**", and Search() doesn't give up when seeing "deleted". And "deleted" can be used for insertion

more details about open-addressing

in the Tutorial

An unfortunate Naming Confusion

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



By the 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).

One more thing to know

The **coolest** hash table so far.

Cuckoo Hashing

https://en.wikipedia.org/wiki/Cuckoo_hashing

Invented by Rasmus Pagh and Flemming Friche Rodler in 2001.

Simple algorithm with provable **worst-case O(1)** lookup time.

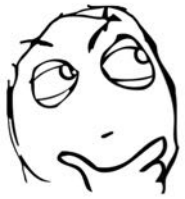
Reflection

Both BST and Hash Table both implement the Dictionary ADT, and they are both invented by smart people.

This is a rare case where one data structure is better than the other in **every** aspect.

But really? Is that true?

Balanced BST	Hash table
O(log n)	O(1)
O(log n)	O(1)
O(log n)	O(1)



What can BST do better than Hash Tables?

Today we learned

- Hash tables: fast lookup
- Choose good hash functions

Next week

- Amortized analysis
- QuickSort