CSC263 Week 5

Announcements

Assignment 1 marks out

class average 70.5%

median 75%

MIDTERM next week!!

MIDTERM:

- Asymptotic analysis (O, Ω)
- Runtime analysis
  (worst-case, best-case, average-case)
- Heaps
- Binary Search Trees
- AVL Trees
- Hashing

Short-answer questions, multiple choice
Example: insert/delete x into this heap/avl tree

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

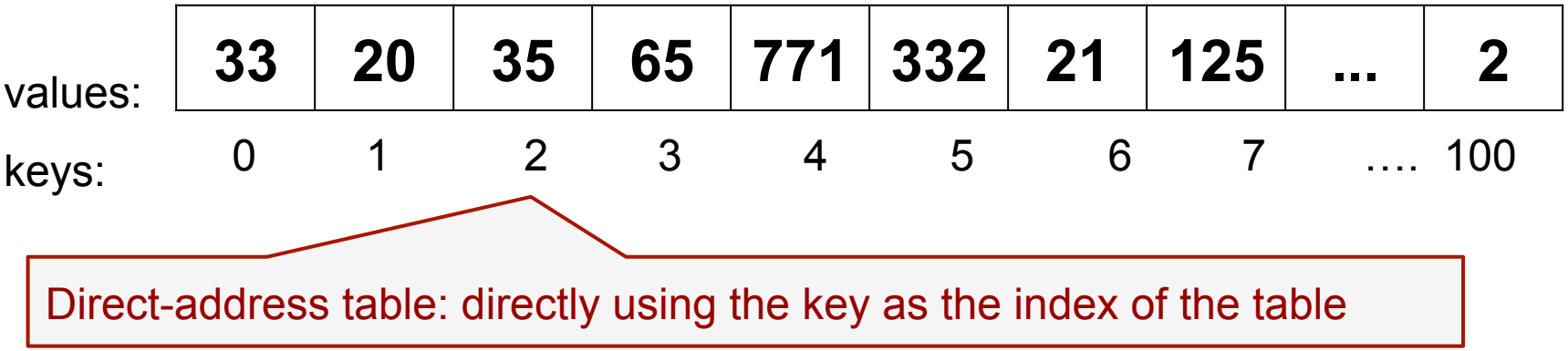
a fancy name for “array”...

Problem

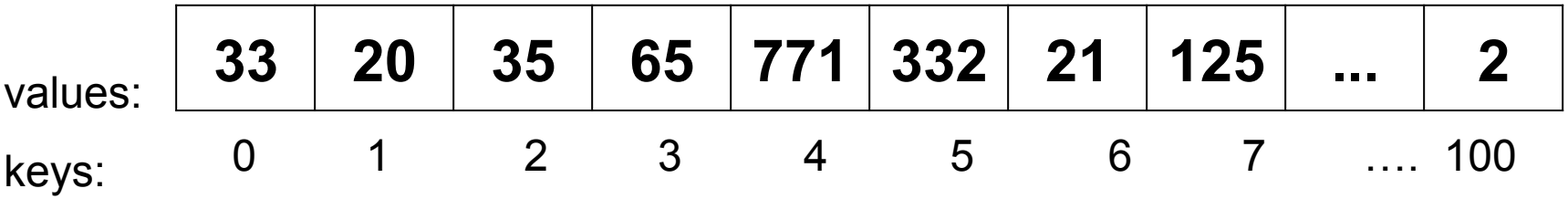
Read a grade file, keep track of number of occurrences of each grade (integer 0~100).

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?



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**.

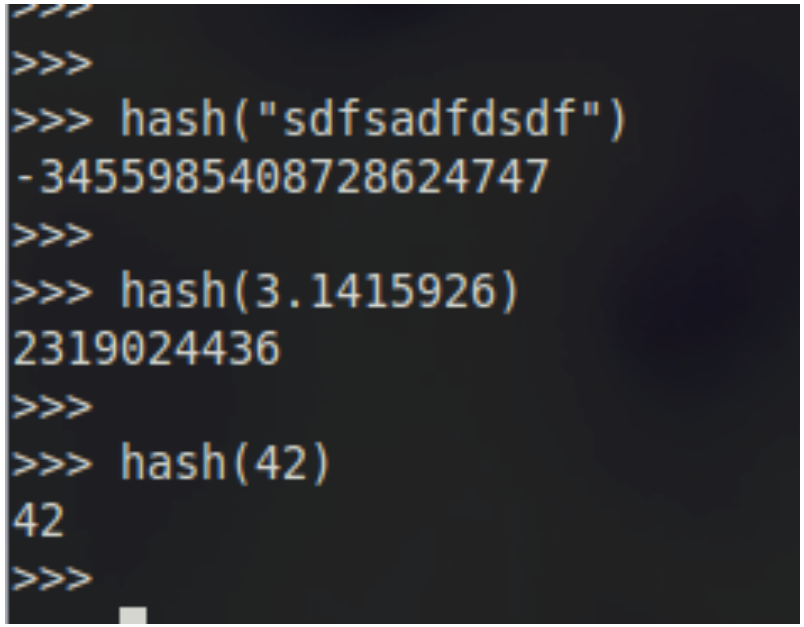
Drawback #2: What if the grade 1,000,000,000 is allowed? Then we need an array of size **1,000,000,001**! Most space is **wasted**.

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).

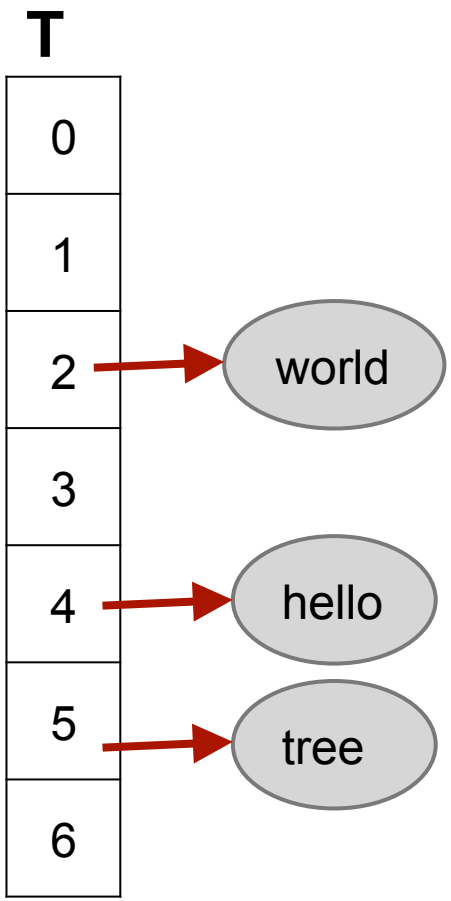
Definitions

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, ***h(k)*** is the index in T where the key ***k*** is stored.

Example: A hash table with M = 7



Insert(“hello”)
assume h(“hello”) = 4

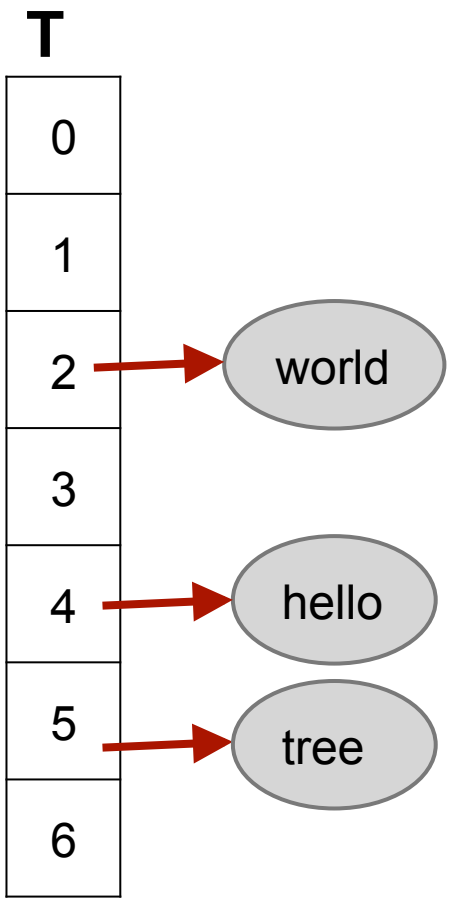
Insert(“world”)
assume h(“world”) = 2

Insert(“tree”)
assume h(“tree”) = 5

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

What’s new potential issue?

Example: A hash table with M = 7



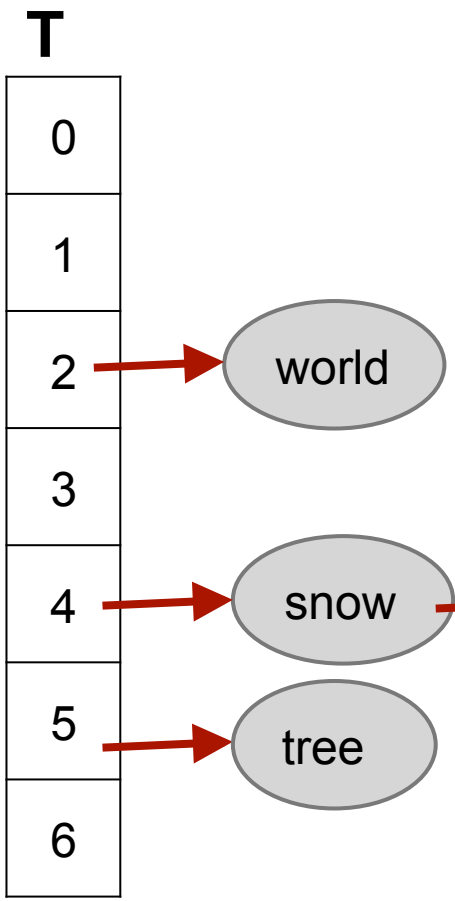
What if we Insert("snow"), and h("snow") = 4?

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?

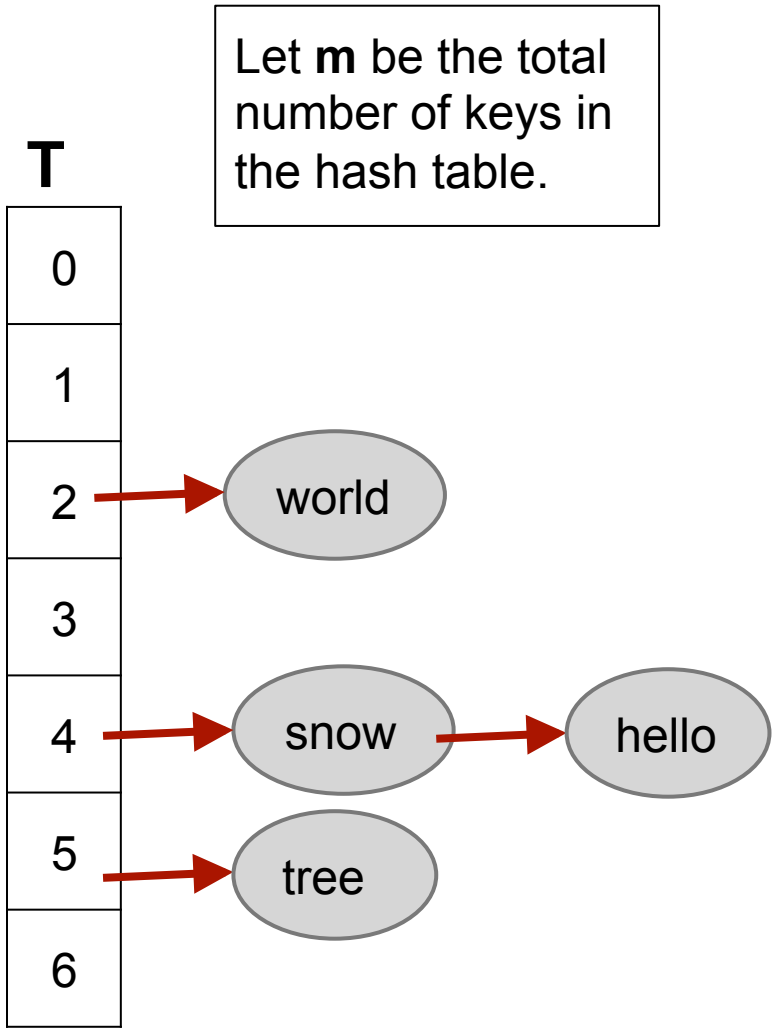
Then we have a **collision**.

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



- **Search(k):**
  - Search k in the linked list stored at **T[h(k)]**
  - Worst-case **O(length of chain),**
  - Worst length of chain: **O(m)** (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(m) 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.
(More on this soon!)

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

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(1010010100001**010**) = **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

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"

The division method

The division method

h(k) = k mod 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** better be a **prime number**

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**" (coming up next!)

➔ Achieve constant sized chains in expectation by choosing randomly from this set of hash functions. (choose a,b randomly)

On Heuristics

- These methods can be good in practice but there is NO GUARANTEE.

- If the hash function h is chosen in advance, there will be sets S that hash very badly (and thus all operations will be inefficient.)

So what else can we do?

Use randomness!!

Randomness

(noun)

A random sequence of events, symbols or steps has no order and does not follow an intelligible pattern or combination

Randomness

- Randomness is a wonderful resource!
- It allows us to fool adversaries.
- It can give faster and simpler algorithms
- MANY applications including: cryptography, data privacy, statistics

Universal Hashing

- Use randomization to achieve good expected behavior of hashing with chaining for any subset ***S*** of ***U***

- **Idea:** Pick a hash function h from $\mathcal{H}$ at random, where $\mathcal{H}$ is a "nice" family of hash functions H so that:

  for any ***S***, the expected number of collisions is constant.

Definitions

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

U = [0,1,…,N-1]

Range = [0,1,…,M-1]

Hash functions h: maps [0,1,…,N-1] to [0,1,…,M-1]

Let S be the (unknown) subset of U that is getting mapped to [0,1,..,M-1]

Let |S| = m, and let m/M = α be the load factor
(typically we want to choose M ~ m)

Universal Hashing

A family $\mathcal{H}$ of hash functions from [0,1,..,N-1] to [0,1,…,M-1] is d-**universal** if for all j,k in [0,1,..,N-1], j ≠ k

$$\Pr_{h \text{ in } \mathcal{H}}[\, h(j) = h(k)] \leq d/M$$

****Think of d = 1 (or maybe 2)**

Equivalently let $X_{jk} = 1$ if h(j)=h(k) and 0 otherwise.
Then $\mathcal{H}$ is universal if for all j ≠ k: $E_h [X_{jk}] \leq d/M$

Universal Hashing: why is d-universal good enough?

Theorem

Let $0 \le j \le N-1$, and let **S** be a subset of **[0,..,N-1]**, $|S|=m$. Then

$E_{h \text{ in } \mathcal{H}}$[# collisions between j and **S**] $\le dm/M$

[The number of collisions between j and **S** is the number of items in **S** that map to h(j)]

Theorem tells us that each j, the expected chain length of bucket containing j is at most $1+dm/M$
If we pick M so that $m/M = O(1)$, this is constant!

Universal hashing

Proof of Theorem.

Let $C_h(j,S)$ = # collisions between j and S

Let X_{jk} be the indicator random variable that is

1 if h(j)=h(k) and 0 otherwise

Let $C_h(j,S) = \Sigma_{k \text{ in } S} X_{jk}$

Then:

$$E_h[C_h(j,S) = \Sigma_{k \text{ in } S} E_h[X_{jk}] \leq \Sigma_{k \text{ in } S} 1/M = md/M$$

Designing a universal family $\mathcal{H}$

Example 1. The set of all functions from
U=[0,…,N-1] to [0,..,M-1] is a universal family.

What is the problem??

Designing a universal family $\mathcal{H}$

Example 1. The set of all functions from
U=[0,…,N-1] to [0,..,M-1] is a universal family.

What is the problem??

This universal family is WAY too large.
Just to write down one **h in** $\mathcal{H}$ takes
time >> N > M > m
BUT we want to run in time O(log m) or O(1)

Designing a universal family $\mathcal{H}$

Example 2. Let U=[0,…,N-1], where N is a prime p, and M divides p-1

Let $\mathcal{H}$ = { $\mathbf{h_a}$ | a=1,2,…,p-1 }

where $h_a(x) = (ax \bmod p) \bmod M$

Theorem.

$Pr_{\mathbf{h}}$ [**h**(j) = **h**(k)] ≤ 2/M (so 2-universal)

Designing a universal family $\mathcal{H}$

Let U=[0,…,N-1], where N is a prime p, and M divides p-1
Let $\mathcal{H}$ = { $\mathbf{h}_a$ | a=1,2,…,p-1 }
where $h_a(x)$ = (ax mod p) mod M

Theorem.
$Pr_{\mathbf{h}}$ [$\mathbf{h}(j) = \mathbf{h}(k)$] ≤ 2/M

Fact 1. For p prime, ax mod p is 1-1, onto for all a
Fact 2. For all z≠0, for all i: Pr_a [az mod p = i] = 1/p

Designing a universal family $\mathcal{H}$

Fact 1. For p prime, for all a, (ax mod p) is bijective

Fact 2. For all $z \neq 0$, for all i: Pr_a [az mod p = i] = 1/p

Example: p=11

x	0	1	2	3	4	5	6	7	8	9	10
a=1	0	1	2	3	4	5	6	7	8	9	10
a=2	0	2	4	6	8	10	1	3	5	7	9
a=3	0	3	6	9	1	4	7	10	2	5	8
a=4	0	4	8	1	5	9	2	6	10	4	7
a=5	0	5	10	4	9	3	8	2	7	1	6
a=6	0	6	1	7	2	8	3	9	4	10	5
a=7	0	7	3	10	6	2	9	5	1	8	4
a=8	0	8	5	2	10	7	4	1	9	6	3
a=0	0	9	7	5	3	1	10	8	6	4	2
a=10	0	10	9	8	7	6	5	4	3	2	1

Designing a universal family $\mathcal{H}$

The facts tells us that all columns (except the first) of the matrix below are different permutations of [1…p-1].

Example: p=11

x	0	1	2	3	4	5	6	7	8	9	10
a=1	0	1	2	3	4	5	6	7	8	9	10
a=2	0	2	4	6	8	10	1	3	5	7	9
a=3	0	3	6	9	1	4	7	10	2	5	8
a=4	0	4	8	1	5	9	2	6	10	4	7
a=5	0	5	10	4	9	3	8	2	7	1	6
a=6	0	6	1	7	2	8	3	9	4	10	5
a=7	0	7	3	10	6	2	9	5	1	8	4
a=8	0	8	5	2	10	7	4	1	9	6	3
a=0	0	9	7	5	3	1	10	8	6	4	2
a=10	0	10	9	8	7	6	5	4	3	2	1

Designing a universal family $\mathcal{H}$

Fact 1. For p prime, for all a, (ax mod p) is bijective
Fact 2. For all $z \neq 0$, for all i: Pr_a [az mod p = i] = 1/p

Say p-1 = cM (M divides p-1). Then:

Pr_h [h(x)=h(y)] (y<x<p, maps to something < M-1)
= Pr_a [(ax mod p)mod M = (ay mod p)mod M]
= Pr_a [(a(x-y)mod p)mod M =0]
= Pr_a [a(z) mod p = 0 or M or 2M or ..or cM] ($z \neq 0$)
≤ 1/p (c+1) [by Fact 2]
≤ 1/p (2p/M)
= 2/M

Summary: Universal Hashing

- Start with unknown S
- Randomly pick one hash function, h, from a small, efficient universal hash family and use h to map S
- The expected chain length will be constant!
- **Very important:** this expectation is for all S, over the random choice of h

Open addressing

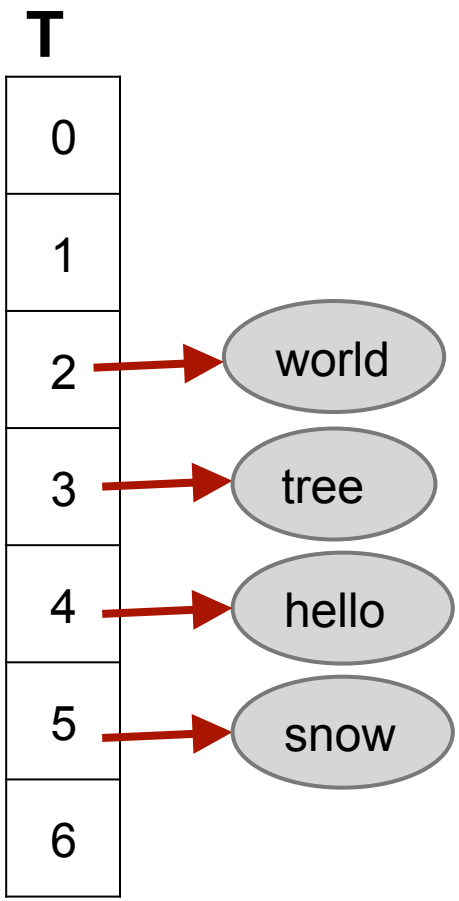
another way of resolving **collisions**
other than chaining

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, ...`



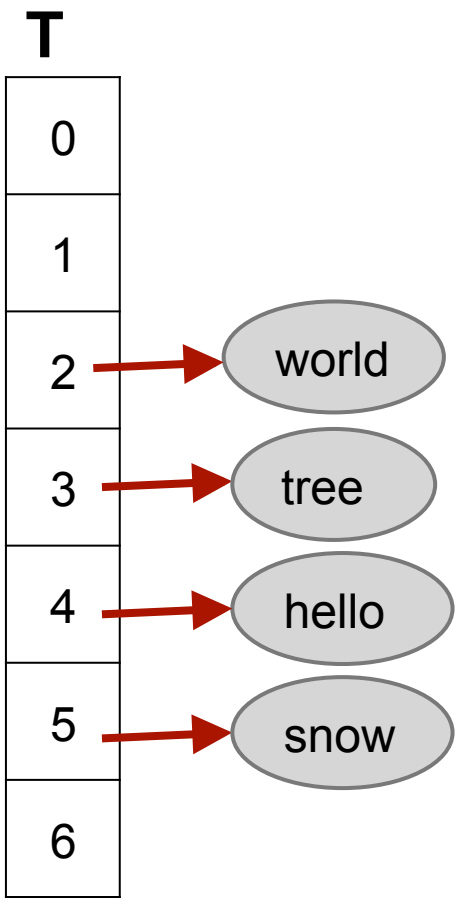
Insert("hello")
assume h("hello") = 4

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 $i=0,1,2,...$

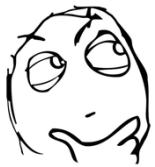
Now the jumps almost look like random, the jump-step ($h_2(k)$) is different for different **k**, which helps avoiding clustering upon collisions, therefore avoids long runs *(each one has their own way of jumping, so no crowd)*.

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.

Hashing is one of the most important ideas in Computer Science!!

What you have seen today is just the tip of the iceberg!!

- Perfect hashing
- Cuckoo hashing
- Bloom filter
- Fast string search algorithm
- Geometric hashing
- Cryptography: authentication, message fingerprinting, digital signatures
- Complexity: approximate counting, recycling random bits, interactive proofs

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**: expected time O(1+α) for search, insert and delete when h chosen randomly from a universal hash family

Next week

➔Randomized algorithms