Tutorial 06 - Table ADT 1, Hash Table

CS2040C Semester 2 2018/2019

By Jin Zhe, adapted from slides by Ranald+Si Jie, AY1819 S2 Tutor

Motivation

A motivating example

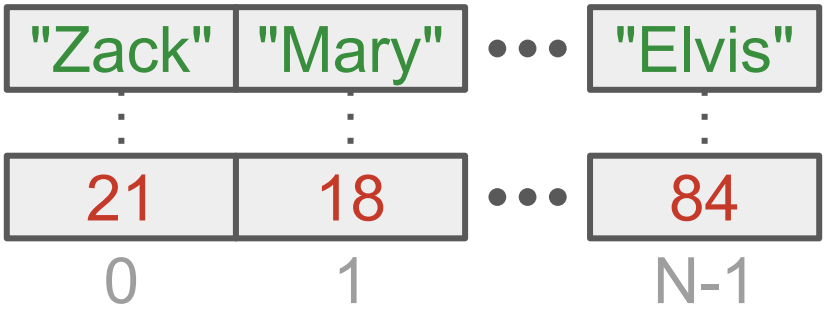
Suppose we have the following table (e.g. a spreadsheet) containing entries of people's name and age and we would like to capture it in a data structure which would support `search(name)`, `insert(name, age)` and `remove(name)` operations.

Name	Age
Zack	21
Mary	18
⋮	⋮
Elvis	84

Solution 1

Use two arrays

- One for storing `string` name
- One for storing `int` age
- The data entry for a person is represented by corresponding indexes in both arrays



Solution 1 analysis

- `search(name)`: Linear search the entire array for names. $O(N)$
- `insert(name, age)`: Just append to the back of arrays. $O(1)$
- `remove(name)`: Find the name, then remove elements at that index from both arrays. $O(N)$

Solution 2

- We know that we can do better if the array is sorted!
- Sorting two arrays while maintaining index correspondences is hairy, so we shall just use an array of (name, age) pairs and sort it based on first element of pair, i.e. the name

Solution 2 analysis

- `search(name)`: Binary search! $O(\log N)$
- `insert(name, age)`: Binary search lower bound then insert at rightful rank in the array. $O(N)$
- `remove(name)`: Binary search, then remove it from array. $O(N)$
- If given an array in the beginning, we pay an initial penalty of $O(N \log N)$ for sorting it

Can we do better?

Observations:

- It incurs $O(\log N)$ time to search — required by all 3 operations
- It incurs $O(N)$ time to shift array elements — incurred by `insert` and `remove`

Key idea

Can we somehow find a magical function which **maps each possible name to a fixed and unique array index** in $O(1)$? i.e. allows us to achieve random access in the array!

Realize with this we also no longer need to shift elements in the array during `insert` and `remove` since other names must always be fixed to their respective indexes.

Key idea

We call such a magical function a *hash function*!

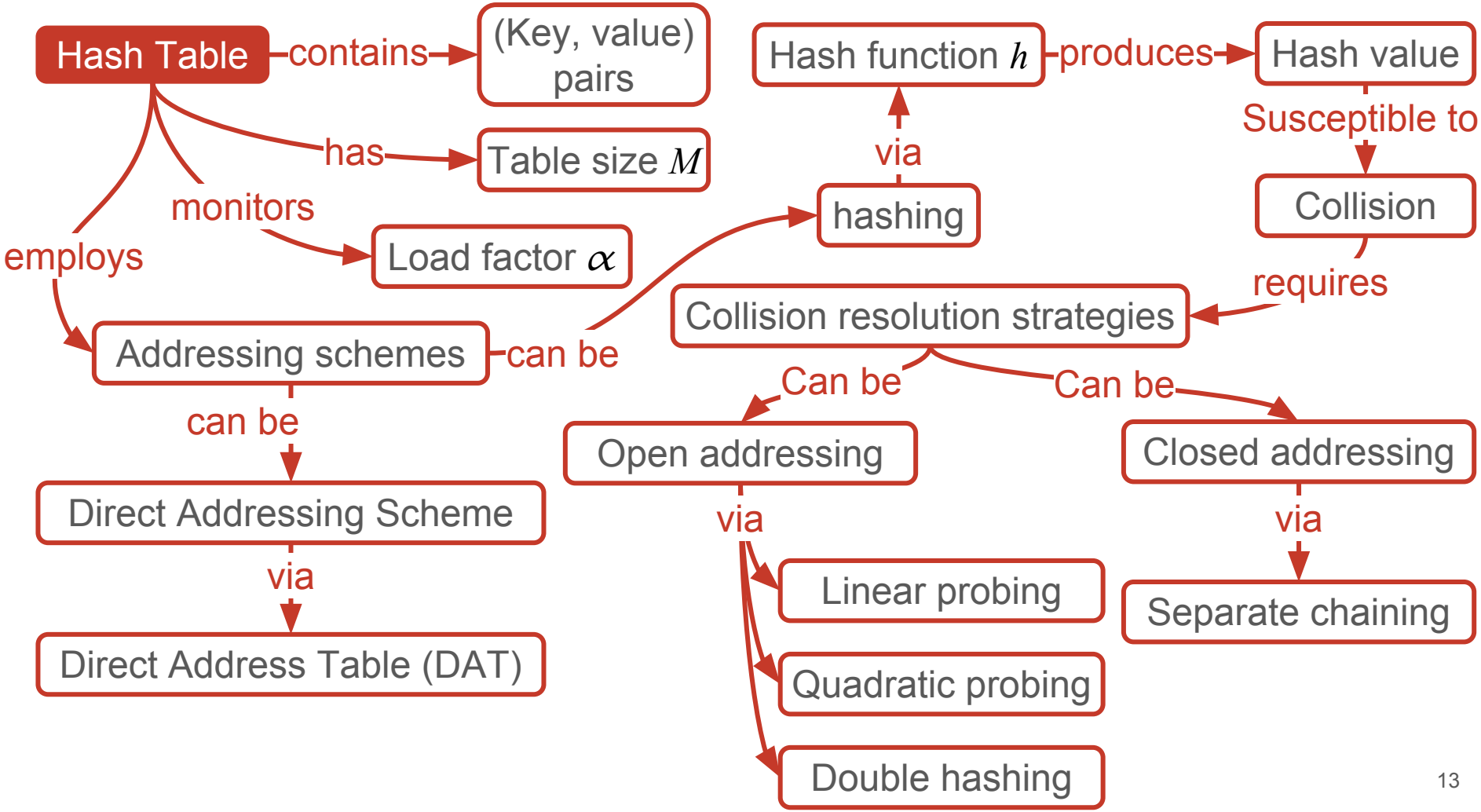
The data-structure it supports is called a *Hash Table*

Did you know?

The term "hash" offers a natural analogy with its non-technical meaning (to "chop" or "make a mess" out of something), given how hash functions scramble their input data to derive their output. In his research for the precise origin of the term, Donald Knuth notes that, while Hans Peter Luhn of IBM appears to have been the first to use the concept of a hash function in a memo dated January 1953, the term itself would only appear in published literature in the late 1960s, on Herbert Hellerman's Digital Computer System Principles, even though it was already widespread jargon by then.

Source: Wikipedia

Conceptual re*hash*


Hash Table
contains
(Key, value) pairs
has
Table size M
monitors
Load factor α
employs
Addressing schemes
can be
can be
Direct Addressing Scheme
via
Direct Address Table (DAT)
Hash function h
produces
Hash value
Susceptible to
Collision
via
hashing
requires
Collision resolution strategies
Can be
Can be
Open addressing
Closed addressing
via
via
Linear probing
Quadratic probing
Double hashing
Separate chaining

Other terminologies

- Hash Table is often also known as *Hashmap*, *Table*, *Map ADT* and *Dictionary*
- Hash value is also commonly referred to as *Hashcode*
- The act of calling hash function on a key is known as *hashing*
- Load factor is also referred to as *density*
- An index position in the Hash Table is commonly referred to as *table address*
- In closed addressing, a table address is sometimes also referred to as a *bucket* since it can contain multiple (key, value)

Table ADT

Common Table ADT operations

`search(v)`	Determine if `v` exists in the ADT or not. If it does return `true`, else return `false`.
`insert(v)`	Insert `v` into the ADT.
`remove(A)`	Remove `v` from the ADT.

We want a data structure implementation such that all three major Table ADT operations are done in $O(1)$

Hash function

A hash function $h(k)$ is one that maps an input key k (of any type) to a table address within a range $[0 .. M-1]$ where M is the *table size*.

Example:

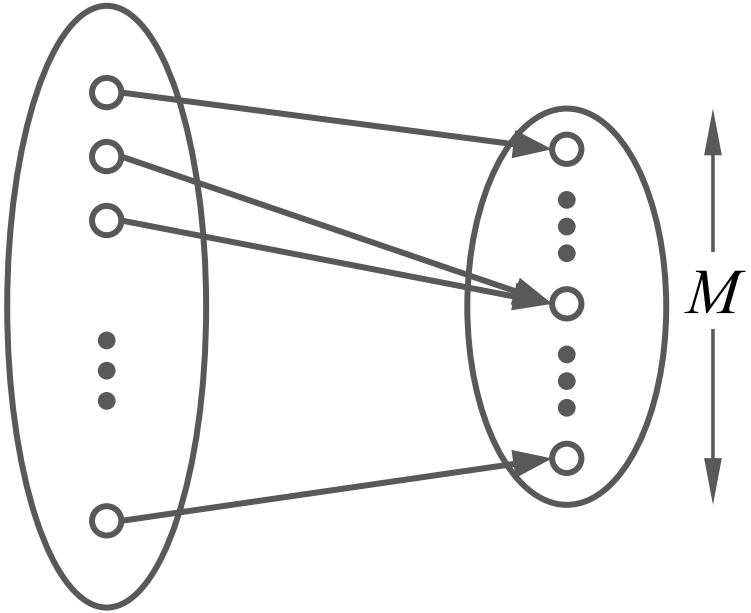
`h(2040)` → `6208`

`h("Hello")` → `39485`

Real life example: Postal Code

Hash function

$h: K \rightarrow \{0, \ldots, M-1\}$

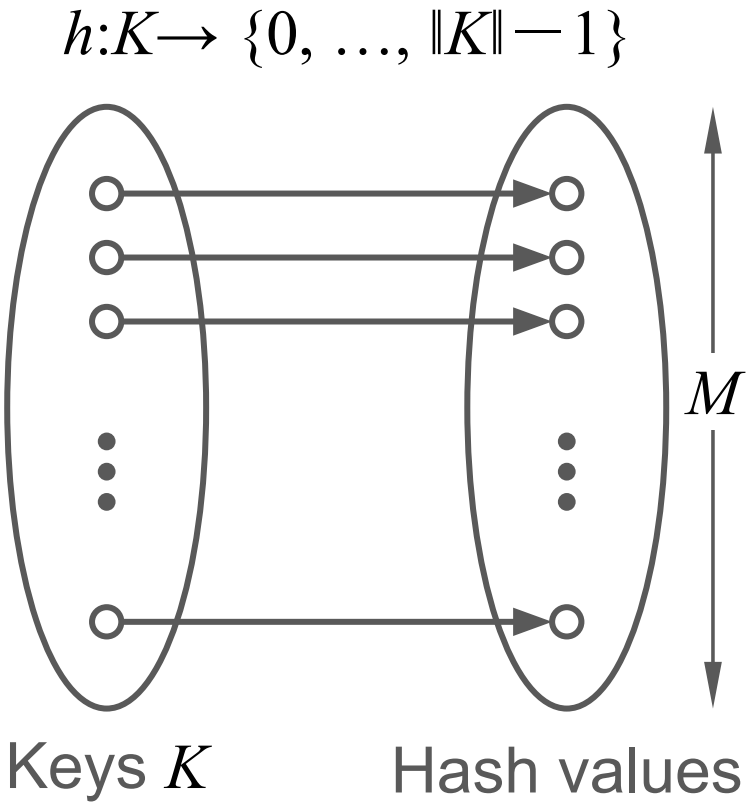


When we choose $M < \|K\|$, the hash function produces a many-to-one mapping.

Therefore collisions are inevitable **by design**.

Recall Pigeonhole Principle.

Hash function



A *perfect hash function* is one which achieves a one-to-one mapping (i.e. $M = \|K\|$), thereby preventing collisions. (See VisuAlgo Hashtable slide 4-4)

Realize this property is analogous to a Direct Address Table (DAT) which conceptually uses the identity function ($f: K \rightarrow K$) in place of a hash function.

Hash function

If perfect hash function and DAT prevents collision from occurring, why don’t we always use them?

- Perfect hash functions can only be derived if all possible keys are known beforehand. This is rarely the case!
- DATs can be extremely wasteful in terms of space. E.g. For keys that are positive 32-bit integers, a DAT would require ~8GB of memory, most of which are probably never used! :O
- DATs are only achievable for keys that have finite ranges! In practice, many keys have almost infinite ranges! E.g. for keys of type `string`, `float`, `double` etc. Therefore it’s often inevitable that $M < \|K\|$

Good Hash Function

In general, hash functions need to optimize for two **competing objectives**:

1. **Minimising table size** M: The range of possible hash values
2. **Minimising collision**: When 2 different keys are hashed to same table address

Good Hash Function

Minimize table size M because

- Large range of hashed values means more *space/memory* required
- If $M >$ range of keys, then many addresses in the table are ‘wasted’ because no key maps to them

Good Hash Function

Minimize hash value collisions because

- More hash collisions means more time required to determine if a key is in the table (main subroutine)
- Assuming constant table size, the most optimal distribution of keys (after hashing) over addresses is attained when collisions are minimized

Question 1 — Hash Table Basics

Finding a good hash function

A good hash function is essential for good Hash Table performance. It should be easy/efficient to compute and evenly distribute the possible keys.

For all the proposed hash functions on the following slides, comment on their flaw(s) (if any).

Assume load factor $\alpha =$ number of items N / Hash Table size $M = 0.3$ (i.e. low enough)

Question 1 Part 1

Hash Table Size M: 100

Keys `k`: Positive even integer

Hash Function `h(k)`: `k % 100`

Question 1 Part 1

Problems:

- Odd slots will *never* be utilized! Wasted half of the capacity!
  - Better `h(k)`: `(k / 2) % M`
- M is also not a prime! Why do we want primes?
  - Better `h(k)`: `(k / 2) % 97`

Self-read

Why we want table size M to be a prime

Informal proof adapted from this blog post:

- Suppose your hash function is perfect and produces raw hashed values (before `% M`) which are uniformly distributed. i.e. If a key is supplied at random, then every hashed value in possible output range is equally likely
- The entire space of possible raw hash values will contain many groups of numbers $\{x, 2x, 3x, 4x, 5x, 6x, \ldots\}$ which are multiples of a number x. i.e. x is the greatest common factor (GCF) between all the numbers in the group
- After `% M`, all these raw hash values will collide over k buckets, where $k = M / GCF(M, x)$.* E.g. For $M = 22$, $x = 4$, after `% M`, multiples of x will fall into $22/2 = 11$ buckets: $\{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}$

*Proof in CLRS chapter 31,theroem 31.20

Self-read

Why we want table size M to be a prime (cont'd)

- One way to prevent that is to choose a hashing scheme which does not generate raw hash codes that are multiples of one another but that would mean it's also a bad hash function which doesn't have a good distribution over its raw output range
- We observe what we really want is to force k to be equal to M so as to make each index in table a bucket! This is achieved by making $GCF(M, x)$ equal to 1! In other words, we force M to be *coprime* to x. Since x can be just about any number, we simply make sure that M is a large prime number!

Question 1 Part 2

Hash Table Size M: 1009

Keys `k`: Valid email addresses

Hash Function `h(k)`: `<sum of ASCII values of the last 10 characters> % 1009`

Recall: ASCII is a numerical representation of characters.

E.g. `'a' = 97; 'A' = 65; '#' = 35;`

Question 1 Part 2

Problem: Suffix of emails are largely the same ⇒ many emails would be hashed to the same values!

E.g. @u.nus.edu, @gmail.com etc.

Solution: Use standard string hashing formula (See VisuAlgo Hashtable 4-7) over the entire email address!

Question 1 Part 3

Hash Table Size M: 101

Keys `k`: $\in [0, 1000]$

Hash Function `h(k)`: `floor(k * random) % 101`,

where $0.0 \leq$ `random` ≤ 1.0

Note: `random` is floating point type so we have to `floor` the expression to ensure return value is an integer

Question 1 Part 3

Pros: Likely lead to optimal distribution of keys over table addresses

Cons:

- `random` will most likely be different every time for the same value of `k`. i.e. Non-deterministic hash values!
- We can insert a (key, value) quickly, but we cannot find it afterwards unless we linear search the table! :O

Hash Table in C++11

C++11 provides Hash Table data structure via `std::unordered_map`.

Built-in hash functions available for keys of primitive types

- Integer types (`int`/`long long`/`char`/`short`)
- Floating point types (`float`/`double`/`long double`)
- `string`

There is no need to custom define hash functions when our keys are of primitive types!

E.g. a Hash Table for `string` key, `int` value pairs is declared via:
`unordered_map<string, int>`

A caveat

However, keys that are hashing floating point types is **not recommended.**

Why?

Floating point numbers are prone to **precision errors**.

A small difference will lead to a completely different hash value!

Eg: 2.0000000000000000 vs 2.0000000000000001

Collision Resolutions

Open Addressing
Closed Addressing

Open vs Closed Addressing

Closed Addressing

Where the object will be slotted in is *completely* dependent on the hash function. i.e. Table address for a key is **fixed**

Open Addressing

Where the object will be slotted in depends on *other objects in the Hash Table*. i.e. Table address for a key is **variable**

Closed Addressing

Imagine there is a street with houses.

Some are empty, some are occupied.

Closed Addressing

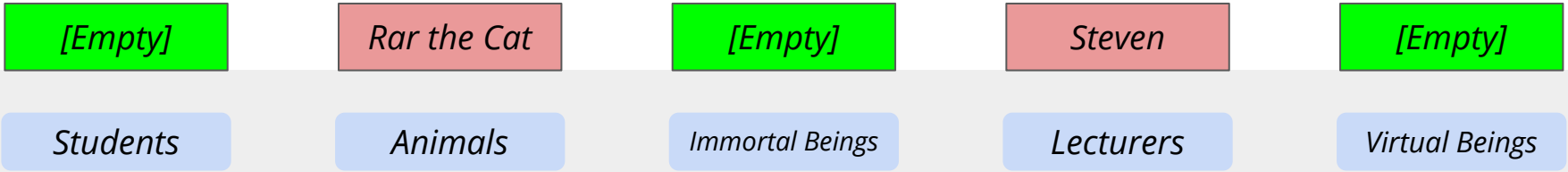
A new person “Jacq the Dino” comes along.

We use our hash function to determine where to place her.

[Empty]	Rar the Cat	[Empty]	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Closed Addressing

However, the house at the chosen location, is already occupied. :(

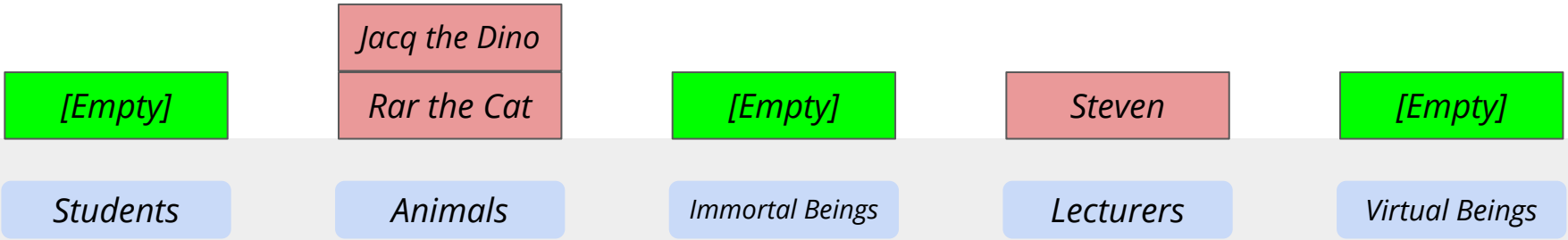


Closed Addressing

In **closed** addressing, instead of finding a new house…

We simply ‘build a new floor’ at the same position…

and place the element there.

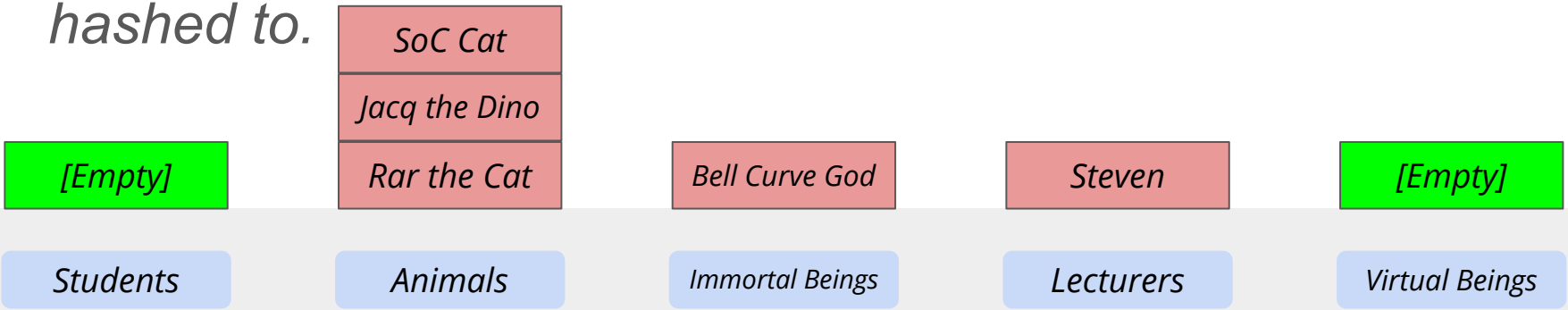


Closed Addressing

As we add more and more people, there are

- Some buckets with more people and
- some buckets with nobody

However, we know they are *always in the position they are hashed to.*

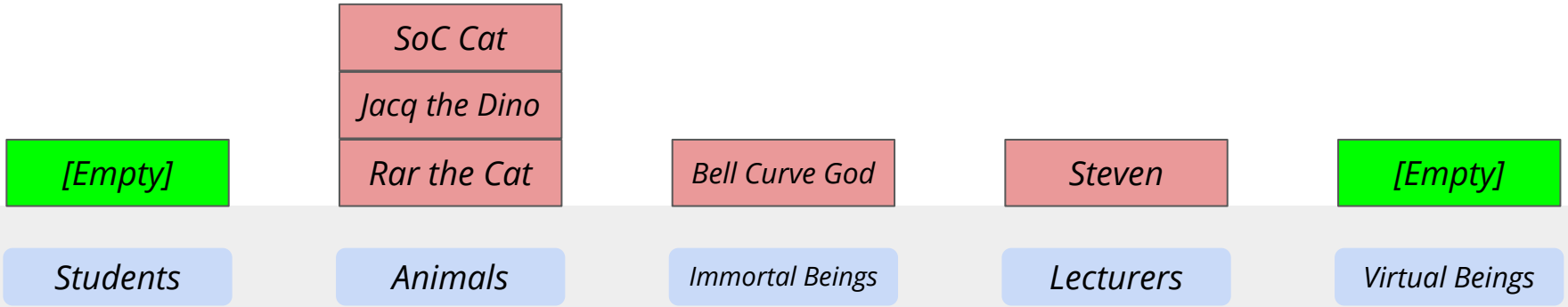


Closed Addressing

Implementation

We can use an *array of* vectors (or Singly/Doubly LL).

Each position will be a *vector (resizeable array)* so that it can accommodate more people if needed.



Closed Addressing

Implementation

Recall that finding an element in a vector is $O(\alpha)$.

Recall that deleting any element in a vector is also $O(\alpha)$.

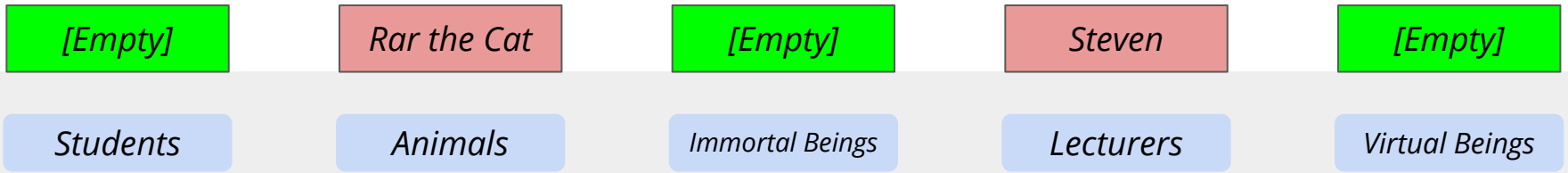
We will try to make α as small as we can.

Open Addressing

Imagine again there is a street with houses.

Some are empty, some are occupied.

And you have that same ‘hash function’ from just now too.



Open Addressing

Similarly, a new person “Jacq the Dino” comes along.

We determine where she should go, depending on the hash function.

[Empty]	Rar the Cat	[Empty]	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing

However, the house is already occupied :(

In open addressing, we cannot build another floor.

We thus need to find another (vacant) house for “Jacq the Dino”.

[Empty]	Rar the Cat	[Empty]	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing

There are several ways to do so:

- Go down the road *one by one* and find the first vacant house on the right. (**Linear Probing** aka LP)
- Wrap-around to the front if needed.

[Empty]	Rar the Cat	[Empty]	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing - Linear Probing

If we used Linear Probing, “Jacq the Dino” will end up here!

[Empty]	Rar the Cat	Jacq the Dino	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing - Linear Probing

Now lets see when “SoC Cat” comes.

Full...

[Empty]	Rar the Cat	Jacq the Dino	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing - Linear Probing

Now lets see when “SoC Cat” comes.

Full… full…

[Empty]	Rar the Cat	Jacq the Dino	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing - Linear Probing

Now lets see when “SoC Cat” comes.

Full… full… *still* full…

[Empty]	Rar the Cat	Jacq the Dino	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing - Linear Probing

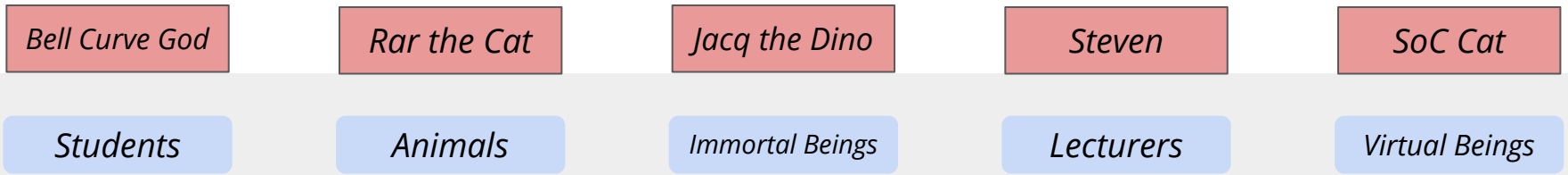
Now lets see when “SoC Cat” comes.

Full… full… *still* full… vacant!

Open Addressing - Linear Probing

Now when “Bell Curve God” arrives, he will end up there.

After probing a full cycle…

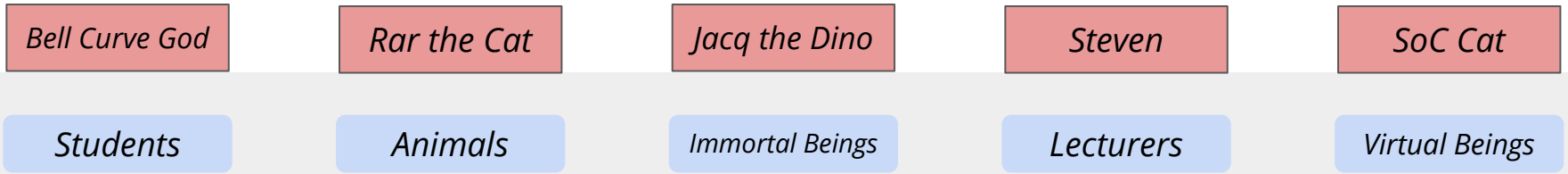


Open Addressing - Linear Probing

As you can see, the hash function *does not have much meaning anymore…*

It only serves to determine the **starting position**.

(This might complicate *some* cases where the key stores important information)



Open Addressing

Instead of checking adjacent houses, we can skip some houses based on a quadratic formula

- 1st house, 4th house, 9th house … etc
- **Quadratic Probing** aka QP

[Empty]	Rar the Cat	[Empty]	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

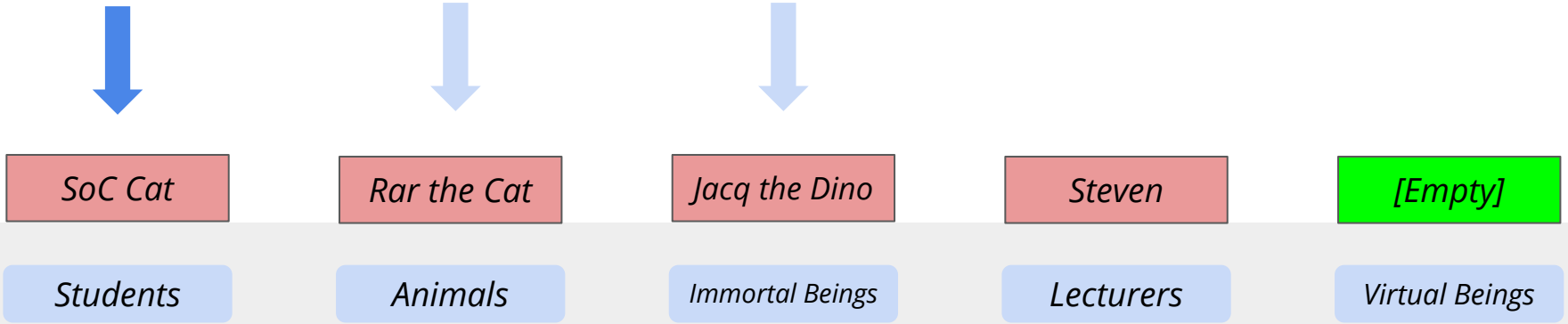
Open Addressing - Quadratic Probing

If we used Quadratic Probing, “Jacq the Dino” will still end up here!

[Empty]	Rar the Cat	Jacq the Dino	Steven	[Empty]
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

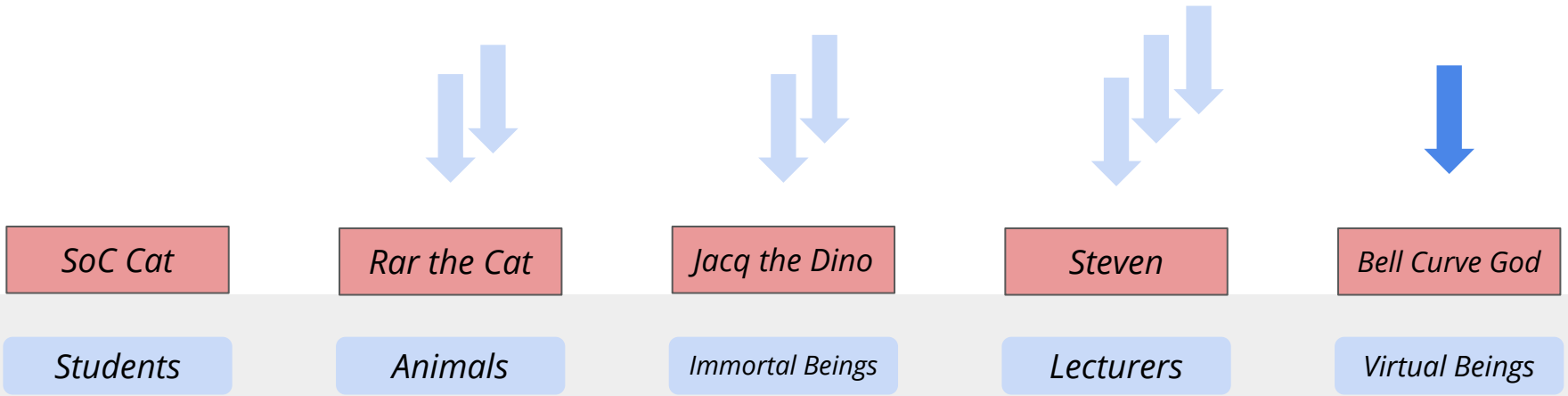
Open Addressing - Quadratic Probing

But when “SoC Cat” comes, he will end up… here.



Open Addressing - Quadratic Probing

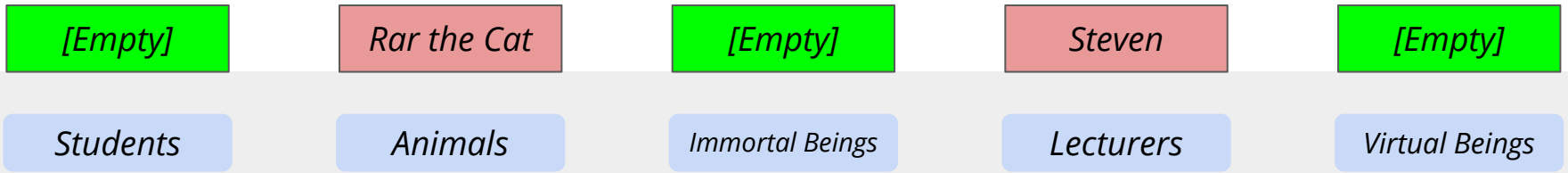
But when “Bell Curve God” comes, he will keep checking… and checking .. and end up at the last empty slot after **18** tries….



Open Addressing - Double Hashing

We can also vary the order to check, based on another hash!

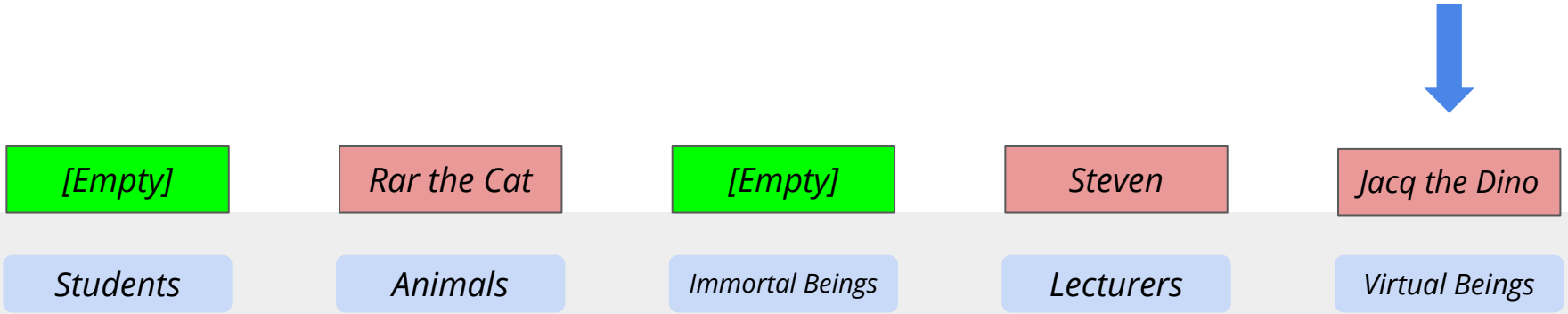
- Let **x** = hash2(*“Jacq the Dino”*) = **3**
- hash2(*“SoC Cat”*) = **2**
- hash2(*“Bell Curve God”*) = **4**



Open Addressing - Double Hashing

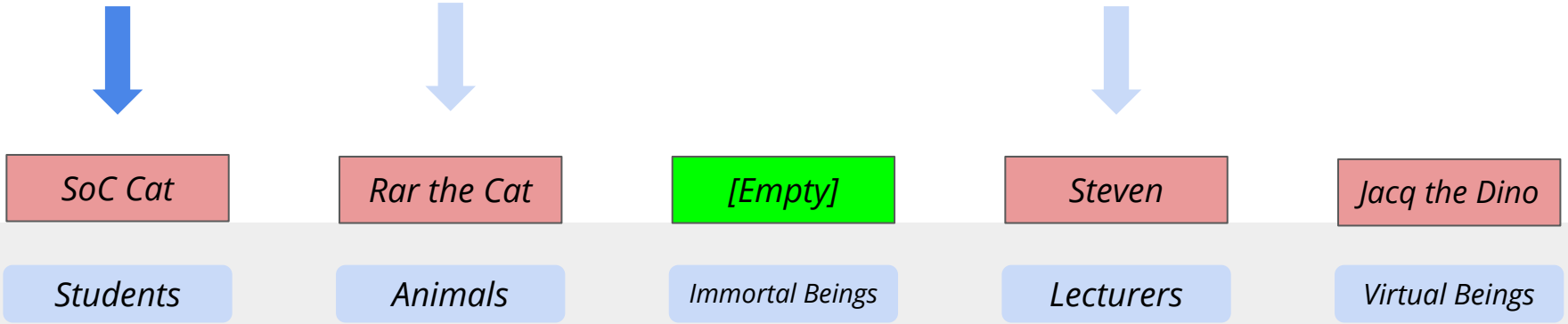
We can also vary the order to check, based on another hash!

- Then we use the second hash value **x**, to vary our steps!
- Checks: base, base + **x**, base + 2**x**, … etc



Open Addressing - Double Hashing

“SoC Cat” with **x** = 2



Open Addressing - Double Hashing

"Bell Curve God" with **x** = 4

But his base slot is already empty.

SoC Cat	*Rar the Cat*	*Bell Curve God*	*Steven*	*Jacq the Dino*
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

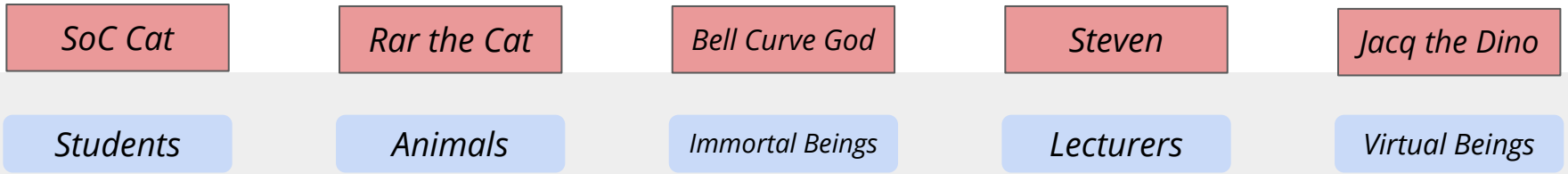
Open Addressing

Implementation

We just need an *array* of elements.

However, the usual practice is to declare much more space than we need. (In general, >= 4 times :D)

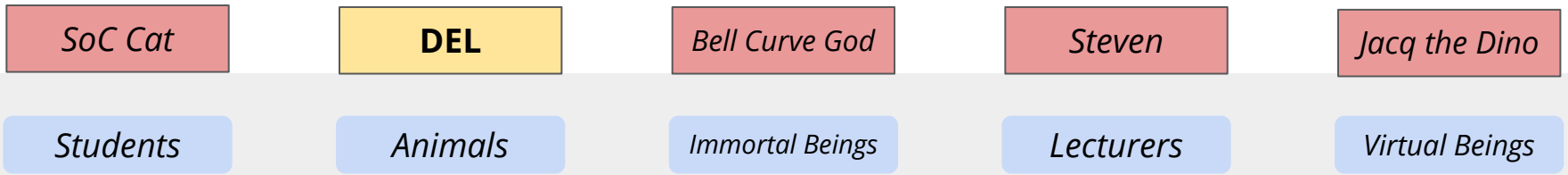
Reduce time associated with repeatedly probing.



Open Addressing

Implementation

When deleting elements, we need to flag it as deleted instead.



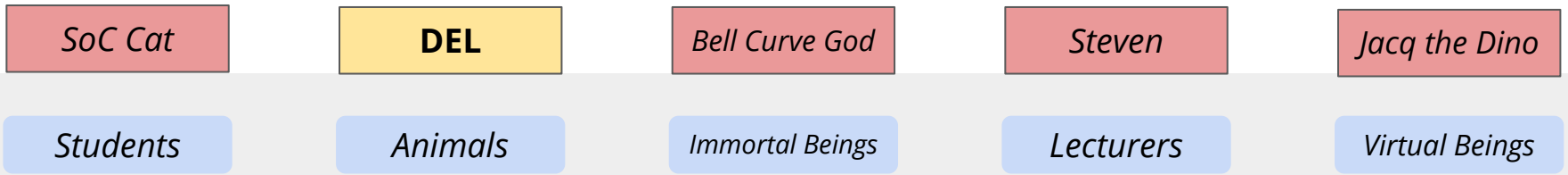
Open Addressing

Implementation

To **find** an element, we follow the same *path* we took when we add the element.

Continue until we find the element…

Or until we see an empty slot.



Open Addressing

Implementation

During ***find*** operation,

we cannot treat ‘deleted’ elements as empty.

Why?

SoC Cat	DEL	Bell Curve God	Steven	Jacq the Dino
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Open Addressing

Implementation

Why?

Imagine if we want to find “SoC Cat” below.

We will stop at the first index if we don’t check beyond the deleted marker.

SoC Cat	**DEL**	*Bell Curve God*	*Steven*	*Jacq the Dino*
Students	*Animals*	*Immortal Beings*	*Lecturers*	*Virtual Beings*

Open Addressing

Implementation

During ***insert*** operation,

We **can** overwrite the deleted markers.

Does not affect our ***find*** operation.

SoC Cat	Tom the Cat	Bell Curve God	Steven	Jacq the Dino
Students	Animals	Immortal Beings	Lecturers	Virtual Beings

Question 2 — Hash Table Basics

Key-Value Mappings

Hashing or No Hashing

Hash Table is a Table ADT that allows for `search(v)`, `insert(new-v)`, and `delete(old-v)` operations in $O(1)$ average-case time, if properly designed.

However, it is not without its limitations. For each of the cases on the following slides, state if Hash Table can be used. If not possible to use Hash Table, explain why is Hash Table not suitable for that particular case. If it is possible to use Hash Table, describe its design, including:

1. The `<Key, Value>` pair
2. Hashing function/algorithm
3. Collision resolution (OA: LP/QP/DH or SC; give some details)

Question 2 Part 1

- A mini-population census conducted in neighbourhood
- No two person share the same name
- Two or more people can share the same age
- Assume age is an integer within $[0..150]$
- Only interested in storing every person's **name and age**
- Operations to support:
  - Retrieve **age given name**
  - Retrieve **list of names given age** (in any order)

Question 2 Part 1

How many Hash Tables do we need?

1. Name to age
2. Age to list of names

Question 2 Part 1

What's the key-value pair and their types?

- Hash Table: Name to age
  - Key: `<string>` name
  - Value: `<int>` age
- DAT: age to names
  - Key: `<int>` age
  - Value: `<vector<string>>` names

Question 2 Part 1

What's the hash function?

- Name to age: `h(name)` using standard hashing for `string` method (see here)
- Age to list of names: Direct Addressing Table $M = 151$

What collision resolution?

- Name to age: SC or OA (LP, QP, DH)
- Age to list of names: Don't need because using DAT! :O

Question 2 Part 2

- A *much larger* population census is conducted across the country
- No two person share the same name
- Two or more people can share the same age
- Age is an integer within $[0..150]$
- Only interested in storing every person's **name and age**
- Operation to support
  - Retrieve **name(s)** of people who have age ≥ 17 years

Question 2 Part 2

How many Hash Tables do we need?

- Just 1 :)
- We can still use DAT like in Part 1

What's the key, value pair and their types?

- Key: `<int>` age
- Value: `<vector<string>>` names

Question 2 Part 2

What's the hash function?

- None needed since we are using direct addressing!

What collision resolution?

- SC since we are appending people's names at the back of the vector

Question 2 Part 2

Retrieve **name(s)** of people who have age ≥ 17 years

For each age $x \geq 17$, i.e. $x \in \{17, 18, 19, \ldots 150\}$

- Get hashcode $h(x)$ to obtain table address
- For each name in vector of names at address, add to output

Complexity: $O(N)$ where N is the number of names in the table

Question 2 Part 3

- A *different* population census is conducted across the country
- Only interested in storing every person's (distinct) **full name** (first name + last name) **and age**.
- Operation to support:
  - Retrieve person's **full name and age** given **last-name** (non-distinct)

Question 2 Part 3

How many Hash Tables do we need?

- Just 1

What's the key, value pair and their types?

- Key: `<string>` last name
- Value: `<vector<pair<string, int>>>` (full name, age) pairs

Question 2 Part 3

What's the hash function?

- `h(last name)` using standard string hashing

What collision resolution?

- SC for people with the same last name

Self-read

Question 2 Part 4

- A grades management program stores a student's **index number** and his/her **final score** in a module
- There are 1,000,000 students, each scoring final score in $[0.0, 100.0]$
- Store all the student's performance
- Operation to support:
  - Print the list of students **in ranking order** who scored *more than 65.5*

Self-read

Question 2 Part 4

Should we still use a hash table for this?

i.e.

- Key: `<float>` score
- Value: `<int>` index number

Not really… There are 2 issues here

Self-read

Question 2 Part 4

Issue 1:

The key is a floating point which will suffer from *floating point precision error*.

If the precision is fixed, say for instance 1 decimal point, we may convert the float to an integer and use it as the key

E.g: 65.5 → 655

Self-read

Question 2 Part 4

Issue 2:

We cannot avoid looping through every single table address and check their respective list of key-value pairs, and then finally sort qualifying key-value pairs. $O(M+N+N\ log\ N)$

Recall that hashing is called hashing because hash functions mess up the order of the data? This is the price we have to pay to support all the good stuff that comes with Hash Tables! Therefore, Hash Tables are not designed for ranked-retrieval problems. We will have to use data-structures to be discussed next week

Question 3 — Basic Hash Table Stuffs

Question 3

Quick check: Let's review all 4 modes of Hash Table (use the Exploration mode of https://visualgo.net/en/hashtable). During the tutorial session, the tutor will randomize the Hash Table size M, the selected mode (LP, QP, DH, or SC), the initial keys inside, and then ask student to `Insert(random-integer)`, `Remove(existing-integer)`, or `Search(integer)` operations. This part can be skipped if most students are already comfortable with the basics.

This part is open ended, up to the tutor.

Question 4 — Hash Table Discussion

Table ADT
Perfect Hash Function
Best Collision Resolution

Question 4 Part 1 — Table V.S. List ADT

What is/are the main difference(s) between List ADT basic operations (recap) versus Table ADT basic operations (recap)?

- Relative **ordering**
  - Important to List ADT
  - Unimportant to Table ADT
- **Mapping** between index to value
  - Important to Table ADT.
  - Unimportant to List ADT. i.e. Inserting at a non-boundary index will change the mapping for remaining items down the list

Self-read

Question 4 Part 2 — Perfect Hash Function

At https://visualgo.net/en/hashtable?slide=4-4, Steven mentions about Perfect Hash Function.

Come up with a **minimal perfect hash function** for these 5 names into index [0, 4] without collision.

- Steven Halim
- Grace Suryani Halim
- Jane Angelina Halim
- Joshua Ben Halim
- Jemimah Charissa Halim

Self-read

Question 4 Part 2 — Perfect Hash Function

One method (among many others) is to just use first character of the second word as the key

- Steven **H**alim
- Grace **S**uryani Halim
- Jane **A**ngelina Halim
- Joshua **B**en Halim
- Jemimah **C**harissa Halim

Self-read

Question 4 Part 2 — Perfect Hash Function

Another method: `(<Number of letters in the first word> - 2 * <Number of words>) % 5`

- **Steven** Halim : `(6) - 2*(2) = 2 % 5 = 2`
- **Grace** Suryani Halim : `(5) - 2*(3) = -1 % 5 = 4`
- **Jane** Angelina Halim : `(4) - 2*(3) = -2 % 5 = 3`
- **Joshua** Ben Halim : `(6) - 2*(3) = 0 % 5 = 0`
- **Jemimah** Charissa Halim : `(7) - 2*(3) = 1 % 5 = 1`

Question 4 Part 2 — Game of Collisions

Thus far, which collision resolution technique is better (in your opinion or Google around): One of the Open Addressing technique (LP, QP, DH) or the Separate Chaining technique?

Open Addressing (LH, QP, DH)

- Require deleted markers
- Inefficient if there are many deletions and insertions

Separate Chaining (SC)

- Unable to fully utilize unused addresses/buckets

Self-read

Question 4 Part 2 — Game of Collisions

Open Addressing (LH, QP, DH)

- Can rehash every 2^K operations (optional)
  - Why? Similar to vector's doubling
- More cache friendly (esp LP)
  - Spatial locality

Separate Chaining (SC)

- Easy to get obtain elements with same hash value because they are all in the same bucket

Self-read (optional)

Actual Implementations

GNU C++

- unordered_map uses Separate Chaining (see here)
- Linear space to number of elements inside

Java

- Also uses Separate Chaining (see here)
- Optimized to use Binary Search Tree within each slot if there are too many collisions

Python

- Open Addressing: hybrid of linear and *pseudo-random* probing (see here)