Tutorial 06 - Table ADT, Hash Table

CS2040S Semester 1 2023/2024

By Wu Biao, adapted from previous slides

Set real display name

https://pollev.com/rezwanarefin430

Motivation

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(v)`	Remove `v` from the ADT.

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

A motivating example

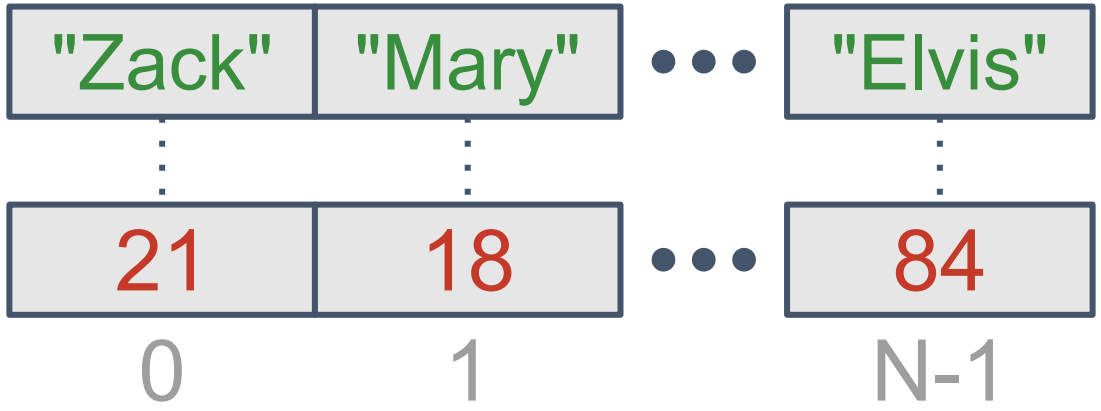
Suppose we have the following table (e.g. a spreadsheet) containing entries of people's name and age and we would like to capture it in a data structure which would support `search(name)`, `insert(name, age)` and `remove(name)` operations.

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.

Hash Function Basics

Hash Function

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

Example:

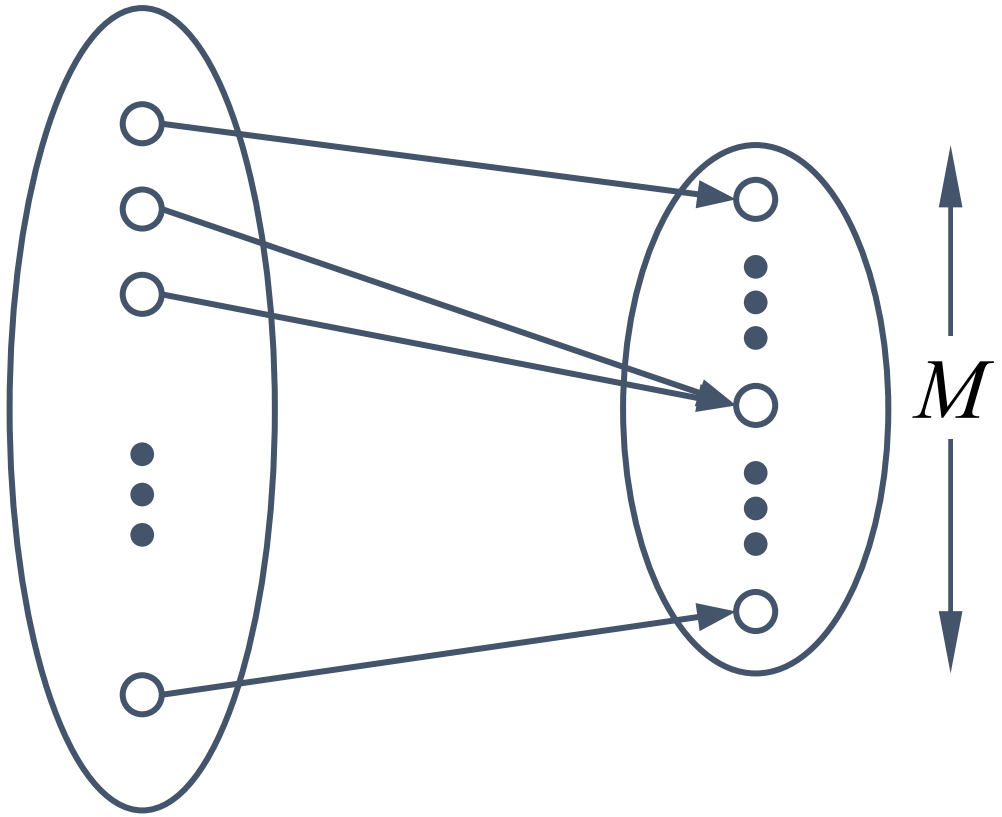
`h(2040)` → `6208`

`h("Hello")` → `39485`

Real life example: Postal Code

Hash Function

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



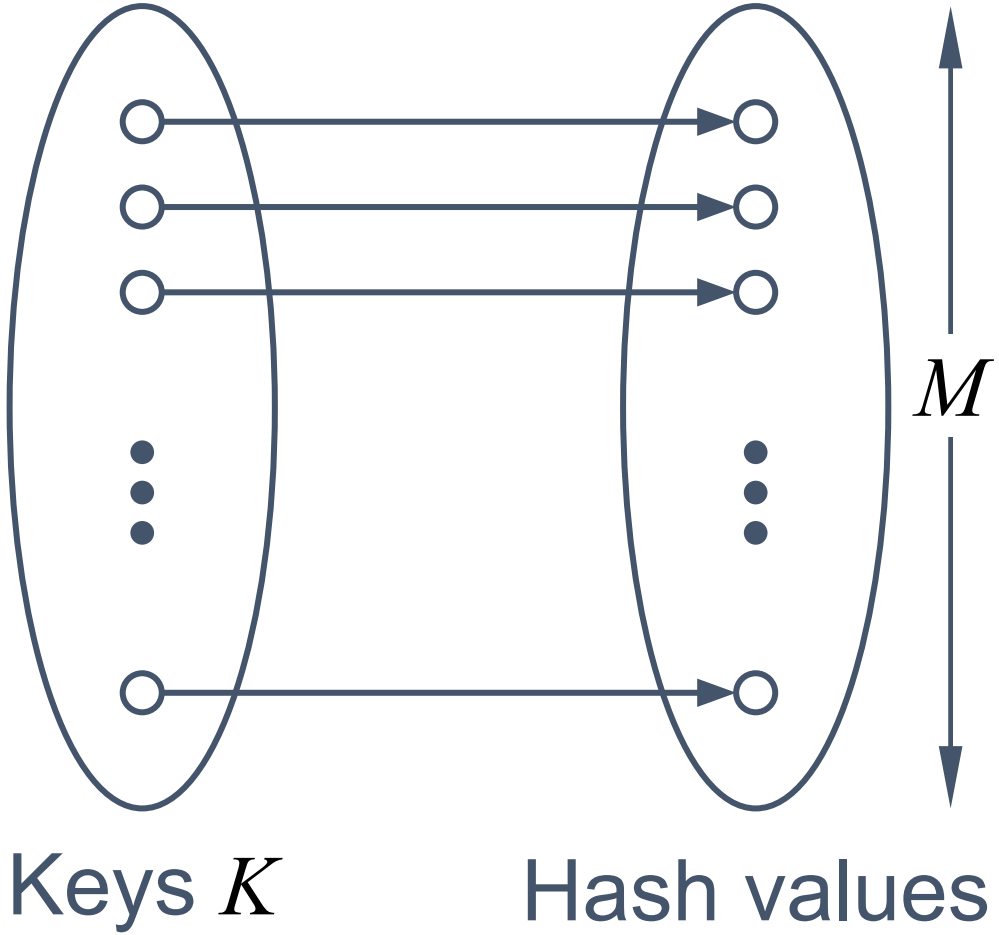
When we choose $\mathbf{M} > \|\mathbf{K}\|$, the hash function produces a many-to-one mapping.

Therefore collisions are inevitable **by design**.

Recall Pigeonhole Principle.

Hash Function

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



A perfect hash function is one which achieves a one-to-one mapping (i.e. $\mathbf{M = \|K\|}$), thereby preventing collisions.

Realize this property is analogous to a Direct Address Table (DAT) which conceptually uses the identity function $\mathbf{f(k) = k}$ 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 $\mathrm{M} >$ 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

Tutorial: Hash Table Basics

Question 1: Choose the best hash function

1. int index = (rand() * (key[0] - ’A’)) % N;

2. int index = (key[0] - ’A’) % N;

3. int index = hash function(key) % N; // hash_function from visualgo

Question 1: Choose the best hash function

1. int index = (rand() * (key[0] - 'A')) % N;

   Not a *function.* Same key may give different hash.

2. int index = (key[0] - 'A') % N;

3. int index = hash function(key) % N; // hash_function from visualgo

Question 1: Choose the best hash function

1. int index = (rand() * (key[0] - 'A')) % N;

   Not a *function.* Same key may give different hash.

2. int index = (key[0] - 'A') % N;

   May not be uniformly distributed. Doesn't use the entire key. What if keys are NUS student numbers?

3. int index = hash function(key) % N; // hash_function from visualgo

Question 1: Choose the best hash function

1. int index = (rand() * (key[0] - 'A')) % N;

   Not a *function.* Same key may give different hash.

2. int index = (key[0] - 'A') % N;

   May not be uniformly distributed. Doesn't use the entire key. What if keys are NUS student numbers?

3. int index = hash function(key) % N; // hash_function from visualgo

   Deterministic. Uses entire key. Has good distribution.

Question 2: Comment on Hash Functions

(1) M = 100. The keys are N = 50 positive even integers in the range of [0, 10^4]. The hash function is h(key) = key % 100.

Question 2: Comment on Hash Functions

(1) M = 100. The keys are N = 50 positive even integers in the range of [0, 10^4]. The hash function is h(key) = key % 100.

Comments:

- h(key) is always even.
  - So half of the table will be unused.
  - Wasted memory.
  - More collisions for not utilizing the whole memory.
- Modulo 100 only uses last two digits.
  - Entire key is not used.
  - May result is many collisions.

Why is prime modulo better?

- Suppose **N** is composite. Consider a factor **p** of **N**.
- All multiples of **p** are also hashed to values that are also multiple of **p**!
- **Example:**
  - In Java, the default hashCode() function is memory address of the Object. Suppose your Object is an int (32 bits = 4 bytes). So the hashCode() is always a multiple of 4.
  - If you write a hash table with modulo 12, then only indices 0, 4, 8 will be used!
  - All other indices: 1, 2, 3, 5, 6, 7, 9, 10, 11 are unused!

Question 2: Comment on Hash Functions

(2) M = 100. The keys are N = 50 positive integers in the range of [0, 10^4].

The hash function is h(key) = floor(sqrt(key)) % 100.

Question 2: Comment on Hash Functions

(2) M = 100. The keys are N = 50 positive integers in the range of $[0, 10^4]$.

The hash function is h(key) = floor(sqrt(key)) % 100.

Comments:

- Not uniformly distributed.
  - Only keys 0, 10000 mapped to 0.
  - Only [1…3] mapped to 1.
  - …
  - [9801…9999] mapped to 99.
- Sqrt is computationally expensive.

Question 2: Comment on Hash Functions

(3) M = 101. The keys are N = 50 positive integers in the range of [0, 10 000]. The hash function is h(key) = floor(key * random) % 101, where 0.0 ≤ random ≤ 1.0.

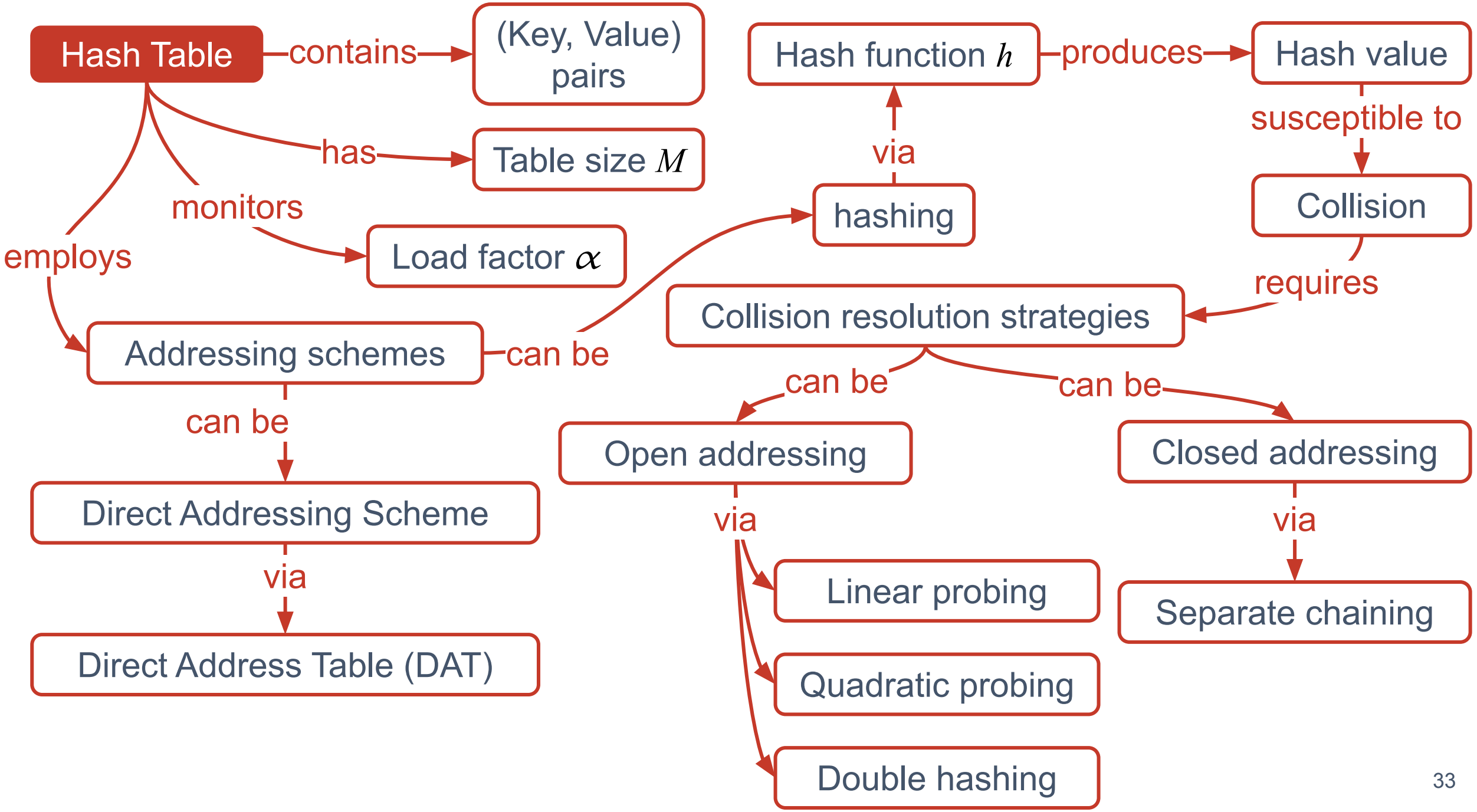
Question 2: Comment on Hash Functions

(3) M = 101. The keys are N = 50 positive integers in the range of [0, 10 000]. The hash function is h(key) = floor(key * random) % 101, where 0.0 ≤ random ≤ 1.0.

Comments:

- Not a function.
- Same key may give different hash every time.

Hash Table Basics


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
requires
via
hashing
Collision resolution strategies
can be
can be
Open addressing
via
Linear probing
Quadratic probing
Double hashing
Closed addressing
via
Separate chaining

Addressing Scheme: Direct Addressing

- If keys are integers and their range is small, we can use **i**-th index of an array to store values of key **i**.
- You have already used direct addressing table without naming it:
  - Counting Sort frequency table: Key = element, Value = frequency of the element.

Addressing Scheme: Hashing

- If direct addressing is not possible, we address using the hash value.
- Two modes of addressing using the hash:
  - **Open addressing:** Position of a key in hash table may depend on other objects.
  - **Closed addressing:** Position of a key in hash table is determined by the key.

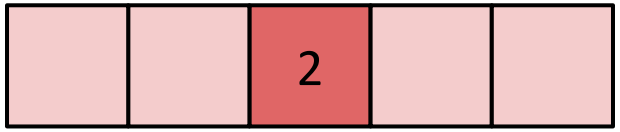
Open Addressing

- In open addressing, we first hash the **key**, say **h**.
- If index **h** is occupied, we iteratively try indices **h+f(1)**, **h+f(2)**, **h+f(3)**, … etc for a function **f**.
  - This sequence **h, h+f(1), h+f(2)**, … is called the probe sequence.
- Some **f** have special names:
  - Linear probing: **f(i) = i.**
  - Quadratic probing: **$f(i) = i^2$.**
  - Double hashing: **f(i) = i * hash2(key)**.

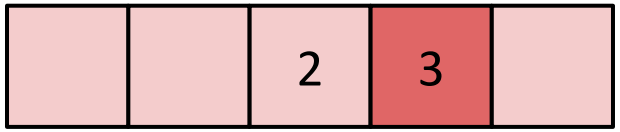
Open Addressing: Insertion

- Calculate hash **h** of the **key**.
- Follow the probe sequence and put the key at the first empty place.

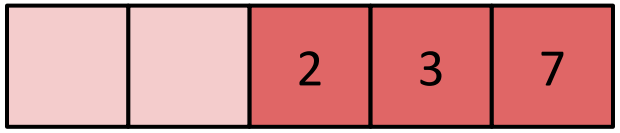
Insert(2):



Insert(3):



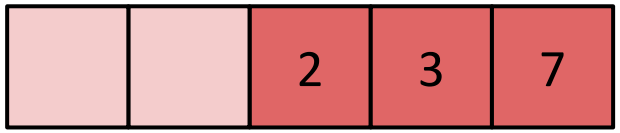
Insert(7):



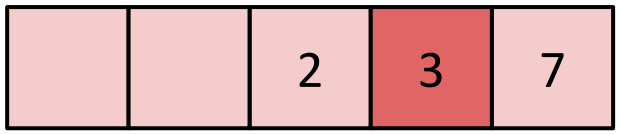
Open Addressing: Search

- Calculate hash **h** of the **key**.
- Follow the probe sequence until you find the key or an empty space.

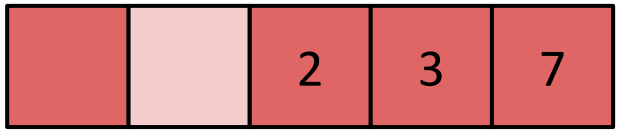
Search(7):



Search(3):



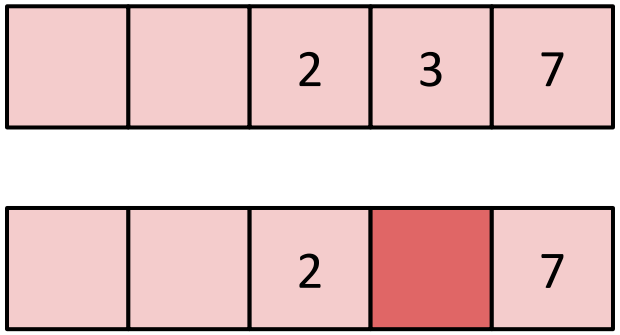
Search(12):



Open Addressing: Remove

- Calculate hash **h** of the **key**.
- Follow the probe sequence until you find the key.
- Remove the key.

Remove(3):

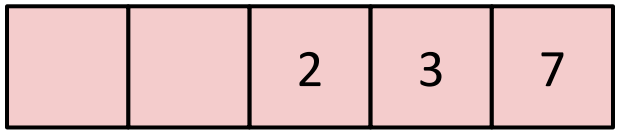


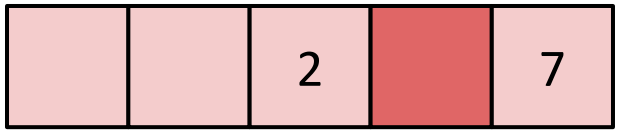
Open Addressing: Remove

- Calculate hash **h** of the **key**.
- Follow the probe sequence until you find the key.
- Remove the key.

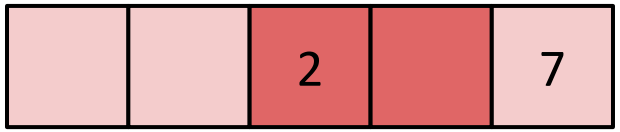
Doesn't work.
Broke previous operations.

Remove(3):





Search(7):

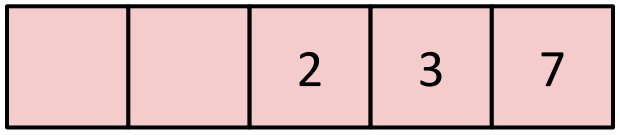


7 not found

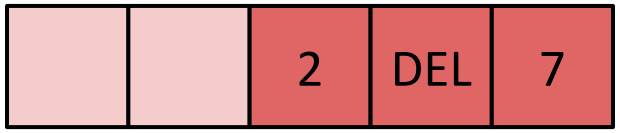
Open Addressing: Remove

- Calculate hash **h** of the **key**.
- Follow the probe sequence until you find the key.
- ~~Remove the key.~~ Replace key with an special symbol DEL.
- Modify previous operations to ignore DEL.

Remove(3):



Search(7):



7 is found

Linear/Quadratic/DoubleHashing Comparison

Linear Probing:

- Forms clusters (called primary clusters).
- The most cache efficient.
  - When you access some memory, a lot of address around that element is loaded into cache. Accessing the cache is almost free compared to memory access.
  - So even though linear probing may create clusters, it is compensated by cache efficiency.
  - In fact most trading companies will use some self implemented Linear Probing hash table variant.

Linear/Quadratic/DoubleHashing Comparison

Quadratic Probing:

- Solved the problem of primary clustering.
- But it may creates clusters along the probing path.
- Which in quadratic probing case is a bigger problem …
  - **h , h+1^2, h+2^2, h+3^2, … modulo P** can have cycles.
  - It is possible that the hash table has empty spaces, but the quadratic probing will never find it!
  - Only guarantee is, if less than half the table is filled, and **P** is a prime, then quadratic probing will at least find a free spot.
  - Always need to ensure more than half of the table is free!
- Cache inefficient than linear probing. Random location accesses.

Linear/Quadratic/DoubleHashing Comparison

DoubleHashing Probing:

- The hash function needs to satisfy some conditions:
  - If hash2 ever returns 0, then we are in trouble.
  - If hash2 returns 1, then it is same as linear probing, and will form primary clusters.
- We generally want hash2 > 1.
- A good choice is **hash2(key) = P' - key % P'** where **P'** is a prime **< P.**
- It is hard to get either primary or secondary clusters here.
- Not cache efficient.

Closed Addressing: Separate Chaining

- For each bucket, create a linked list.
- Insert:
  - Find the correct bucket.
  - Go through linked list and check if **key** is already present.
  - If not, then add the **key** to the back of the linked list.
- Search:
  - Find the correct bucket.
  - Search in the linked list.
- Remove:
  - Find the correct bucket.
  - Remove from linked list.

Closed Addressing: Pros and Cons

- MOST cache inefficient.
  - Nodes in the LinkedList are heap allocated.
  - Each node is in a completely random location.
- Good complexity: on average* **O(1 + α)** per operation.
- Worst case is still O(N).
  - Due to the reason in *. Also if there are many collisions.
- Worst case can be made (ignoring *) O(log N) by using a binary search tree instead of linked list for each bucket.
  - Java does this, if it detect too much collision.

* In practice this is amortized complexity. When the **α** becomes too big, number of buckets is increased to make is smaller. That requires rehashing all the elements in the hash table. So it is possible that a single operation is taking **O(N)**. But over **N** operations, the expected total complexity is **O((1 + α) N)**.

Question 3: Hashing or No Hashing

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 3.1

- 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 3.1

How many Hash Tables do we need?

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

Question 3.1

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

- Hash Table: Name to age
  - Key: `String` name
  - Value: `Integer` age
- DAT: age to names
  - Key: `Integer` age
  - Value: `ArrayList<String>` names

Question 3.1

What's the hash function?

- Name to age: Standard string hashing.
- 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: Not hashing, no collision resolution is needed.
  - We do have a "chain" of sorts in this solution. But the chain itself is the value we are tracking. The chain is not due to some collision resolution.

Question 3.2

- 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 (possibly multiple) person's **full name and age** given **last-name** (non-distinct).

Question 3.2

How many Hash Tables do we need?

- Just 1

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

- Key: `String` last name
- Value: `ArrayList<Pair<String, Integer>>` (full name, age) pairs

Question 3.2

What is the hash function?

- h(last name) using standard string hashing.

What collision resolution?

- Separate chaining for people with same hash of last_name.
- Note that there are *two* chains here.
  - The value in (key, value) itself is a chain: contains persons with same last names.
  - When there is a hash collision in the key (eg. last name), then the hash table will have another chain of these (key, value) pairs.

Question 3.3

- A grades management program stores a student's **index number** and his/her **final score** in a module.
- There are 100000 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.

Question 3.3

Should we use a hash table? Map score to lits of indices of students who got that score. NO.

- Problem: **float** data should not be hashed. Due to precision issues, it is possible that you have a **2** and **2.00000000000001**. But both will get very different hashes.
  - If **float** data is fixed precision, we may convert **float** to **int** first. Eg. **65.5 -> 655**.
- Problem: Hash table has no order!
  - To retrieve all scores at least 65.5, we have to check *every* bucket.
  - After retrieving entries in all buckets we still need to sort them to prepare the report!
- We should really be using some data structure that preserves order!

Question 4: Hash Table Review

Already covered

Question 5: Hash Table Discussion

(1) Any difference between List and Table?

Question 5: Hash Table Discussion

(1) Any difference between List and Table?

Possible answer: In List we control where an element should go. In Table we do not; the table internally chooses a place from where it can efficiently retrieve the element later.

Question 5: Hash Table Discussion

(2) Design a perfect hash function for the names of Steven's current family members: {"Steven Halim", "Grace Suryani Halim", "Jane Angelina Halim", "Joshua Ben Halim", "Jemimah Charissa Halim"}.

Question 5: Hash Table Discussion

(2) Design a perfect hash function for the names of Steven's current family members: {"Steven Halim", "Grace Suryani Halim", "Jane Angelina Halim", "Joshua Ben Halim", "Jemimah Charissa Halim"}.

Possible answer: Take second letter of the name (Note: that these are distinct: t, r, a, o, e). Then map these characters to [0…4].

Question 5: Hash Table Discussion

(3) Compare the collision resolution techniques.

We already did this before.

Question 5: Hash Table Discussion

(4) Which data structure would you use if you need many additions, many removals, and many requests of the data in sorted order?

Definitely not hash table, because it doesn't preserve order.
List can be kept sorted, but then insertions or removals will be expensive.
Heap has *some sort of* order, but cannot produce entire data set in sorted order efficiently.
The proper data structure will be taught later in the course!

Break, Attendance, Questions

Hands-On Session

Java HashSet and HashMap

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashSet.html
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html

https://visualgo.net/training?diff=Medium&n=5&tl=5&module=hashtable

PS4 Discussion

PS4A: /swaptosort

- Given a reverse sorted array **A.**
- You have a list of **(a, b)** pairs.
- Pair **(a, b)** means that you are allowed to **swap(A[a], A[b])**.
- Can you sort the array using only allowed operations?

PS4A: /swaptosort

- Where should **A[i]** go in the final array?
  - **n - i + 1**. (Remember **A** is reverse sorted).
- If you need to swap indices a and b, there must be a sequence of valid swaps: **(a, x), (x, y), (y, z), …., (*, w), (w, b)**.
- Which data structure we've learnt let us query if a and b are reachable like this?
  - Union Find.
  - All swappable pairs should be in the same component.

PS4B: /kaploeb

- Using a hashmap, keep track of the total time each individual and the number of laps they have completed.
- Afterwards, just loop through the key-value pairs, getting the runners if they completed the race and sort them by their completion time, tie break by start number.
- Read floating point numbers as integers **ab.cd**.
  - **COMMON MISTAKE:** value of ab.cd should not be abcd.
  - There are **60** seconds in a minute, not **100**!
  - **ab.cd -> ab * 60 + cd.**

Lab: /bokforing

Lab: /bokforing

Hints will be provided at 5 min intervals.

5 min:

10 min:

15 min:

Lab: /bokforing

Hints will be provided at 5 min intervals.

5 min: Indices are small. Can use a DAT to store wealth.

10 min:

15 min:

Lab: /bokforing

Hints will be provided at 5 min intervals.

5 min: Indices are small. Can use a DAT to store wealth.

10 min: You cannot reset every person's wealth individually on a RESET. What if there are many consecutive RESET?

15 min:

Lab: /bokforing

Hints will be provided at 5 min intervals.

5 min: Indices are small. Can use a DAT to store wealth.
10 min: You cannot reset every person's wealth individually on a RESET. What if there are many consecutive RESET?
15 min: Keep track of a default wealth, and list of persons whose wealth has been changed since last RESET. On a RESET, you only need to change wealth of these persons!

- Since total number of updates is **O(Q)**, the total complexity of all RESET is **O(Q)**. So each RESET is amortized **O(1)**.

Thank You!

Anonymous Feedback:
https://forms.gle/MkETeXdUT53Vhh896