set & hash map

spring 2024
data structures

Set

- ADT
- Stores **unique** values
    - No order necessary
- Static / Frozen Sets
    - Do not change after constructed.
    - Can only do existence check and enumerations on it.
- Dynamic sets
    - Allows insertion and deletion of elements
- Multisets
    - Special set where an element can appear multiple times on the set

```java
Set<String> s = new HashSet<>();

s.add("hey");
s.add("hey");
System.out.println(s);
```

Set

- Holds element without duplicates.
  - If you add an elements of an array into a set, it will remove the duplicates.
  - But the order might be broken (most of the time it is)
- You have a list of emails, you want to get unique elements.
- You want to check whether a user in a blacklist.
  - O(1)
- Or similarly you want to check whether a user is checked in.
- Get the *unique* number of things
  - You played 10 songs. But 3 of them are the same.
  - How many unique songs?
- It is like the set in mathematics
  - You can do set operations: union, differences, etc.

Hash map

- Also referred to as *dictionary*
- Key-value data storage
  - Efficient storage and retrieval of data based on a unique key
  - O(1) average time
    - for lookup, insertion, deletion
- Examples
  - Phone book
  - Capital cities
  - word count
  - caching

Hashing

- There is a **hash function**.
- Deterministic
  - A mathematical function that turns an input into a deterministic output
  - h(input) => Hash (hash value, hash code)
  - For the same input, you will always get the same output.
- Fast
  - Designed to be computed quickly
- Fixed size
  - The length of the output is the same regardless of the input.
  - Doesn’t matter if the input is “a” or “sf04fk204fk24g0k”, the output length will always be the same.
- Irreversible
  - There is no inverse to the hash function.
  - When you look at the hash, you cannot understand what the original input is.
- Unique mapping
  - This is not theoretically possible due to **pigeonhole principle**
  - But we try to do it as unique as possible

Hashing

- There are different hash functions
- We can also build our own
  - It doesn’t have to be good
- Good hash functions are constructed usually by mathematicians
  - Or cryptographs for *CHF* (Cryptographic Hash Functions)
  - For a hash function to be CHF, it needs to be **better** than general.
    - Has to satisfy certain conditions
    - MD5, Sha1 (old)
    - SHA256, SHA512 (new, standards)
    - Generally used for password storage

- Imagine a library
- Books are **data**
- Their names are **keys**
  - By using those keys, we reach the books.
- The librarian can create a **unique shelf number** based on the keys.
  - So, we put the keys into a hash function, and we get a unique value.
  - The hash function we will create will output a shelf number.
    - Ideally, a unique one.
- So, when we want to get the book,
  - We will write the name (key) of the book.
  - Hash function will calculate the shelf number and find the book and bring it to us.

Simple hash functions

- Let’s create our own hash function.
- Let’s go with the shelf example. If I have 100 shelves, I want the hash function to give me something between 1-100. If it gives me 101, I cannot use it because I don’t have that shelf!
- That is why we are going to control the output of the hash function.
  - We use *modular arithmetic* for it.
  - A number % 100 -> Output cannot be larger than 100.
  - So we control it.

- Let’s do an experiment.
- Imagine a key is the string
  - We can make book names again.
  - Let’s say we have 100 shelves
- We get each char, get the alphabetical index (a->0, b->1, …)
- Sum them up!
- Apparently, it will be larger than the shelf number.
  - We will mod 100 the output number to get a value between 0 and 100.

Collisions

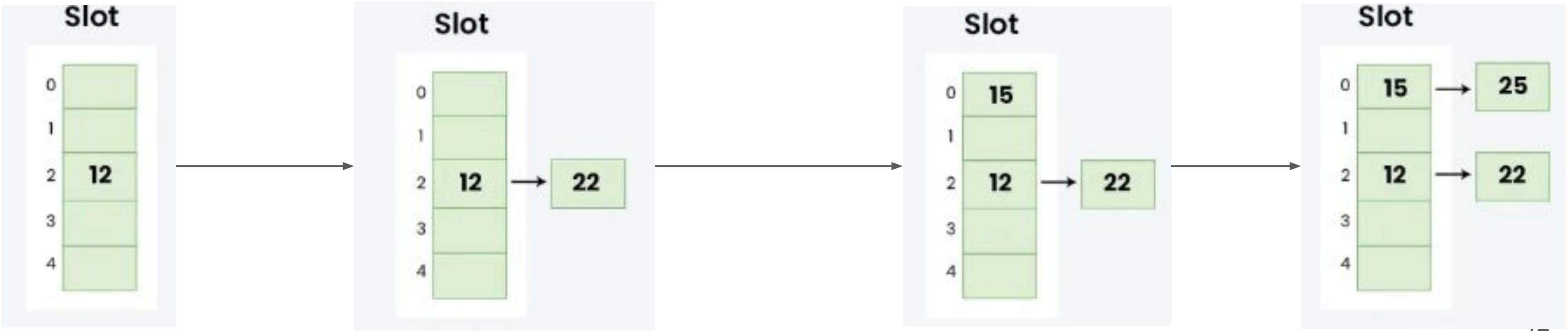
- Inevitable
- Three solutions:
  - Separate chaining
    - Uses linked-list. When multiple elements are hashed into the same index, we insert them into a singly-linked list called **chain**.
  - Open chaining
    - Linear probing
    - Quadratic probing
  - Double hashing

Separate chaining

- If two different elements have the **same** hash value, we store both in the same linked-list, one after the other.
- e.g.
- **Keys:** 12,22,15,25 → mod 5.
  - That means we have 5 slots. 0,1,2,3,4.

Separate chaining (12,22,15,25 → mod 5)

- We get 12 first.
  - Take 12 % 5 -> 2.
  - So we are going to write 12 into index 2.
- Get 22.
  - Get 22%5 -> 2 again.
  - We are creating a linkedlist to slot 2.
- Get 15.
  - 15 % 5 = 0
  - Add to slot 0.
- Get 25
  - 25 % 5 = 0
  - Create a linkedlist for slot 0 and add it.



Separate chaining

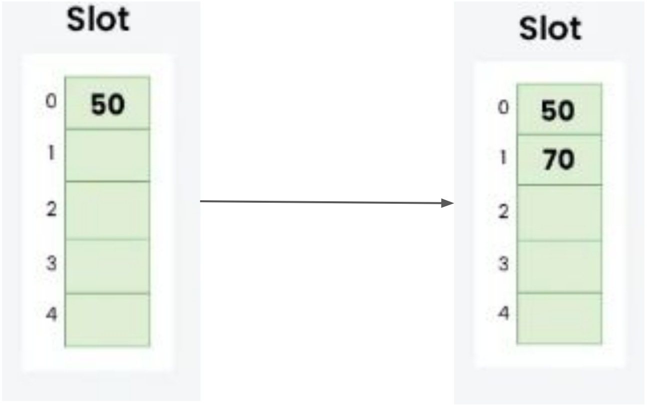
- Easy to implement
- Hash map doesn’t fill up.
- Worse performance compared to open chaining.
- Wasted space
  - Some slots are going to be empty.
- If the chain becomes too long, the complexity increases.
  - Remember, in a linked-list you need to traverse to the end.
  - It’s possible that the element is at the end of the linkedlist.

Open addressing (closed hashing)

- A method for handling collisions.
- Two ways: **Linear probing** and **quadratic probing**
  - Probing: Searching for the next available spot.
- Linear probing
  - When we want to add something, we go to the necessary slot.
  - If the slot is filled, we check for the next location.
    - rehash(key) = (key+1)%size
  - So, let's say our key is 12, and size is 5.
    - 12 % 5 = 2 -> check slot 2. if it is not empty;
    - (12+1) % 5 = 3 -> check slot 3. Do it until you see an empty slot.

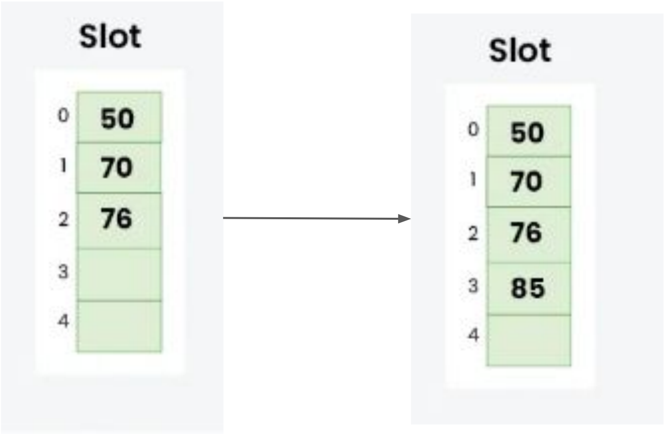
e.g. Linear probing

- Keys
  - 50, 70, 76, 85, 93
- Hash function
  - Mod 5.
- We take 50
  - 50 % 5 = 0 → add to slot 0.
- Take 70
  - 70 % 5 = 0 → add to slot 0
  - can’t do it, slot is not empty. rehash
    - 71 % 5 = 1 -> slot 1. if empty, put.



e.g. Linear probing (50,70,76,85,93)

- Get 76
  - 76 % 5 = 1 -> slot 1
  - Slot 1 is full.
    - 77 % 5 = 2 -> slot 2
    - Check slot 2. If empty, put it there.
- Get 85
  - 85 % 5 = 0 -> slot 0
  - Full, 86 % 5 = 1 -> slot 1.
  - Full, 87 % 5 = 2 -> slot 2
  - Full, 88 % 5 = 3 -> slot 3 → put it here.



- Get 93
    - 93 % 5 = 3 → Slot 3.
    - Full, 94 % 5 = 4 -> slot 4
    - Slot 4 -> ok.

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Hash function
  - Mod 10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Example (Linear Probing)

- Keys
    - 89, 18, 49, 58, 69
- Get 89
    - 89 % 10 = 9
    - Slot 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Get 89
  - 89 % 10 = 9
  - Slot 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Get 89
  - 89 % 10 = 9
  - Slot 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Get 18
  - 18 % 10 = 8
  - Slot 8

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Get 49
  - 49 % 10 = 9
  - Slot 9
  - Full, so rehash
  - (49 + 1) % 10 = 0
  - Slot 0

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Get 58
  - 58 % 10 = 8
  - Slot 8 is full, rehash.
  - 59 % 10 = 9
  - Slot 9 is full, rehash
  - 60 % 10 = 0
  - Slot 0 is full, rehash
  - 61 % 10 = 1
  - Slot 1

0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

Example (Linear Probing)

- Keys
  - 89, 18, 49, 58, 69
- Get 69
  - 69 % 10 = 9
  - Slot 9 is full.
  - 70 % 10 = 0
  - Slot 0 is full
  - 71 % 10 = 1
  - slot 1 is full
  - 72 % 10 = 2
  - slot 2

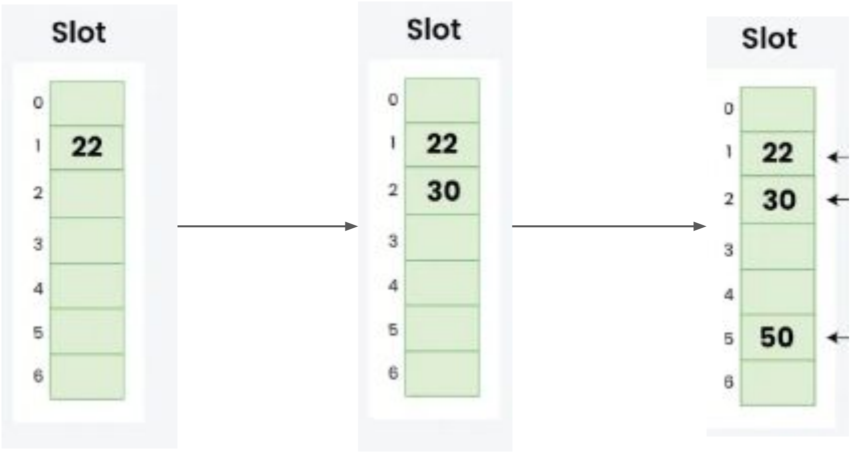
0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

quadratic probing

- Uses *mid-square* method.
- We look for **i^2** th slot for the **i’th** operation.
- We do the hashing again.
    - if slot (hash(x) % s) is full, we rehash: h(x) = (x + i^2) % s

Q. probing (22,30,50)

- Keys
  - 22,30,50
- Hash function
  - mod 7
- Get 22
  - 22 % 7 = 1 -> slot 1
- Get 30
  - 30 % 7 = 2 -> slot 2
- Get 50
  - 50 % 7 = 1 -> slot 1
  - Slot 1 is full.
  - Rehash: (50 + 1*1) % 7 = 2 -> slot 2 is full
  - Rehash: (50 + 2*2) % 7 = 5 -> slot 5.



Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Hash function
  - Mod 10

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Get 89
  - 89 % 10 = 9
  - Slot 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Get 89
  - 89 % 10 = 9
  - Slot 9

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Get 18
  - 18 % 10 = 8
  - Slot 8

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Get 49
  - 49 % 10 = 9
  - Slot 9, full.
  - (49 + 1*1) % 10 = 0
  - Slot 0

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Get 58
  - 58 % 10 = 8
  - Slot 8, full.
  - (58 + 1*1) % 10 = 9
  - Slot 9 is full
  - (58 + 2*2) % 10 = 2
  - Slot 2

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Q. probing

- Keys
  - 89, 18, 49, 58, 69
- Get 69
  - 69 % 10 = 9
  - Slot 9, full.
  - (69 + 1*1) % 10 = 0
  - Slot 0 is full
  - (69 + 2*2) % 10 = 3
  - Slot 3

0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89