TUTORIAL 9

HASHING

Problem: Given some data of the form (k_i, v_i) where k_i is a key and v_i is a value, find some way of efficiently storing this data and operating on it.

We have seen a few ways to maybe do this, but all seem to miss the mark in some way or another:

1. Array: if keys are just an index, then inserting, searching and deleting are $O(1)$! But if they're not, it's $O(n)$...
2. BST: if keys are orderable, can reduce to $O(\log n)$. But if they aren't, this won't work...
3. Linked list: no better than an array or BST, so won't help.

Solution: "Turn" keys into some kind of index so we can use (1). i.e. have a function $h: \{\text{keys}\} \longrightarrow \{\text{indices}\}$ which is cheap to compute for keys, and then for some item (k, v), insert it at index $h(k)$ in some array of values $\Rightarrow$ hashing and hash tables.

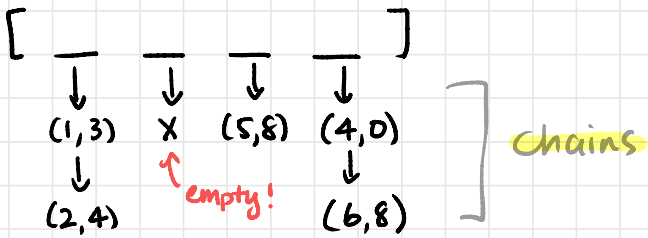
Expectation: If h is $O(1)$ always, then insert, search, delete are also $O(1)$. Sounds good, but...

Reality: Lots of keys
- $\Rightarrow$ lots of slots if we want h to be 1:1
- $\Rightarrow$ need lots of memory, which we might not have
- $\Rightarrow$ allow for $h(k_1) = h(k_2)$ for $k_1 \neq k_2$, i.e. a hash collision
- $\Rightarrow$ now we don't need as many slots!
- $\Rightarrow$ now we have to deal with collisions!

Collision: For two keys k_1 and k_2 with $h(k_1) = h(k_2)$, which one do we put in that array slot?

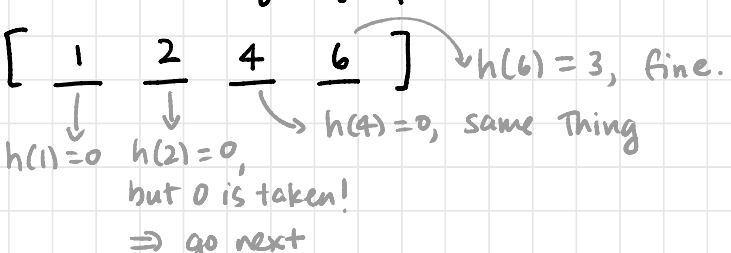
Resolution: A few ways to do this:

① **Separate chaining**: make each array slot a **linked list**, and for each group of keys with the same hash value, store them all as nodes in the slot's linked list.



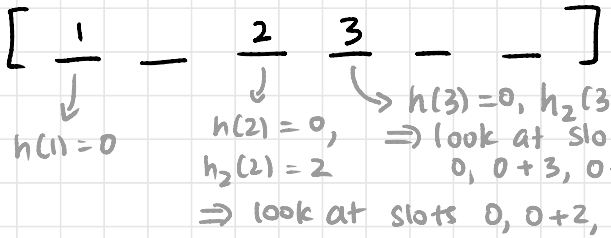
Pros: Theoretically can store infinitely many items!
Cons: If the chain is long, potentially bad complexity.

② **Linear Probing**: When a slot is already taken, look at the next slot, the next next slot, etc. until one is found. When searching / deleting, need to **probe** until the key is found instead of giving up after the first slot.



Pros: Mostly beyond the scope of 2521 (Cache locality)
Cons: Table can only store a fixed # of items before resize, clustering of keys over time

③ Double hashing: Generalisation of LP, because now rather than incrementing by 1 when looking for a slot, we increment by some amount determined by a secondary hash function h_2.



Pros: Same as LP, but with less clustering
Cons: Deletion is quite tricky to do "well"

2. Assume table has N slots, uses separate chaining for collision resolution with sorted chains per slot

(a) Inserting $k = 2N$ items
⇒ best case time complexity (#. key comparisons):
Why?

⇒ worst case complexity (#. key comparisons):
Why?

(b) Average search complexity (#. key comparisons)

* After the best case in (a):

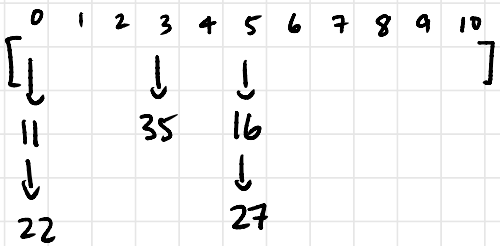
* After the worst case in (a):

3. Insert values into a hash table:

11 16 27 35 22 20 15 24 29 19 3

with $h(x) = x \bmod 11$ as the hash function

(a) Separate chaining:



(b) Linear probing:

0	1	2	3	4	5	6	7	8	9	10
11	22	24	35	15	16	27	29	19	20	3

(c) Double hashing, $h_2(x) = x \bmod 3 + 1$

0	1	2	3	4	5	6	7	8	9	10
11	3	22	35	15	16	27	24	19	20	29

6. 2521 slots, double hashing for collision resolution

$\Rightarrow$ suitable $h_2(x) =$

Why?