#ASLI ENGINEERING

Chaining in Hash Tables

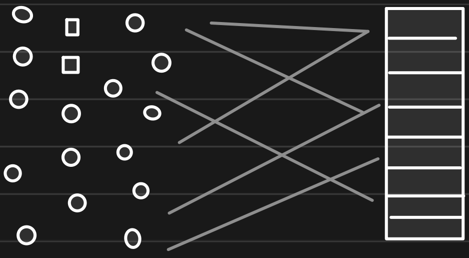
SWIPE

BY

ARPIT BHAYANI

Resolving conflicts through chaining

Conflicts are inevitable!



$$\underbrace{\begin{array}{l} key_1 \longrightarrow f \longrightarrow h_1 \\ key_2 \longrightarrow f \longrightarrow h_1 \end{array}}$$

Multiple keys can produce same hash key upon hashing

So, how can we store multiple keys in the same slot?

The two classical ways of achieving this

1. Chaining
2. Open addressing

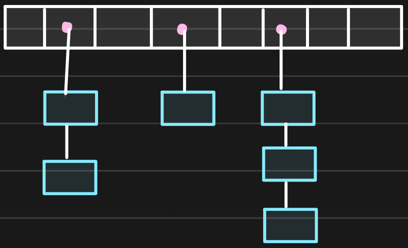
Chaining

We put colliding keys in a data structure that hold them well.

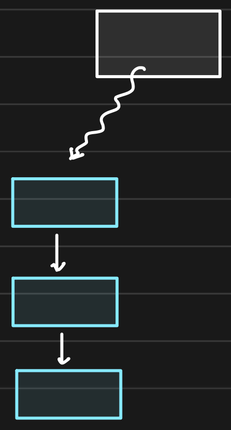
Most common implementation → Linked list

Core idea:

Form a chain of keys that hash to the same slot



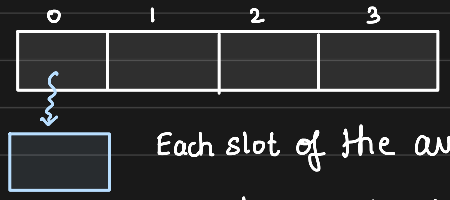
Chaining with linked lists



The core set of operations we need are

1. add a new key to the linked list
2. check if the key is present in the list
3. remove a key from the list

Simplest implementation: Singly linked list



Array of linked list

Each slot of the array contains the pointer to the head of the linked list

```
struct slot {
    struct node * head;
}
```

Each node of the list contains

1. pointer to the actual key
2. pointer to the next node of the list

```
struct node {
    void * key;
    struct node * next;
}
```

ARPIT BHAYANI

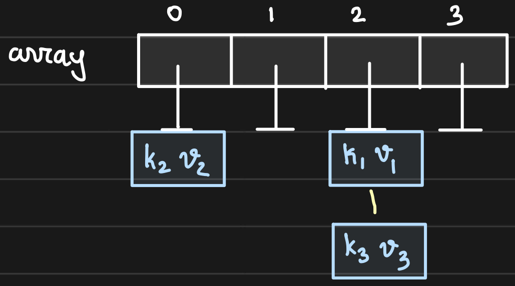
Hash Table Operations

1. Adding a key

$PUT(k_1, v_1)$

$PUT(k_2, v_2)$

$PUT(k_3, v_3)$



Given a key and a value, we

1. pass the key through the hash function and get index i
2. create a new linked list node with k,v
3. add it to the chain present at index i

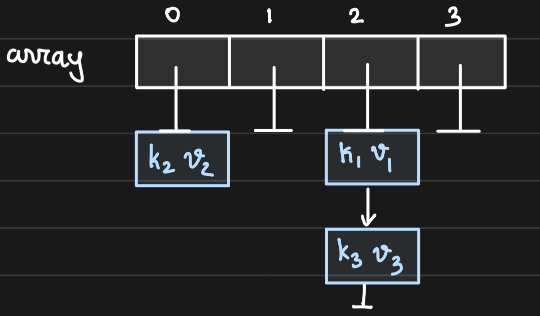
Possible implementations

1. insertion can always happen at the head ↗ fast
2. insertion can happen always at the tail ↗ fast
3. insertion can happen as per the sort order
   ↳ linear iteration

eg: a ⇝ ant → apple → atom

ARPIT BHAYANI

2. Delete a key



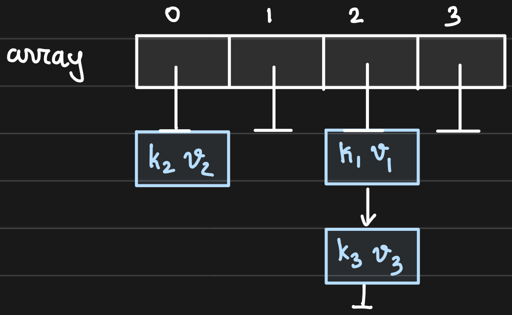
DELETE K_2 , K_1 , K_3

Delete operation is simple

1. reach the slot in O(1)
2. iterate through the list and find node → key == k_1
3. while iterating keep track of prev node
4. adjust the pointers
5. delete the intended node

* Enure pointers are handled and adjusted

3. Lookup a key

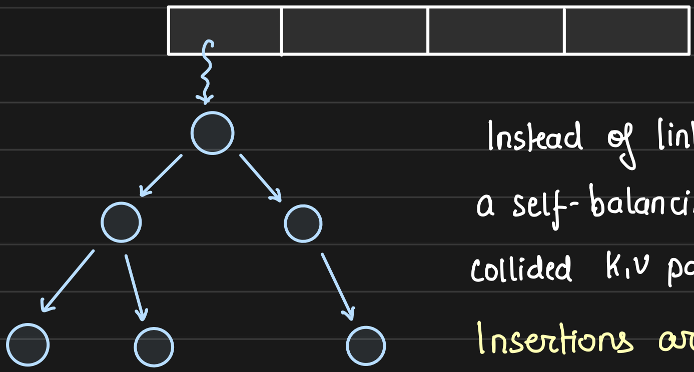


GET K_1 , K_2 , K_3

lookups are similar to delete

1. reach the slot in O(1)
2. linearly iterate through the list until we find node ⟶ key == k_1

Other data structures for chaining



Instead of linked list, we can use a self-balancing binary trees to store collided K,V pairs.

Insertions are not O(1) but lookups are O(h)

We can use search trees for chaining when we are expecting large number of collisions

ARPIT BHAYANI