#ASLI ENGINEERING

Implementing Hash Maps

SWIPE

BY

ARPIT BHAYANI

Implementing Hash Maps

A common use of Hash Tables is to build Hash Maps, a powerful data structure that allows us to store key value pairs and retrieve them by key when needed

* There is no lookup or deletion by Value

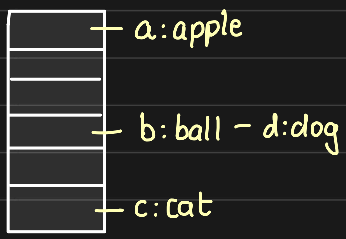
a ⟶ apple
b ⟶ ball
c ⟶ cat
d ⟶ dog

Keys and values both have application context

Hence, we need to store

- the application keys in the Hash Table
- the hash keys along with the key
    ↳ avoid computing $fn(k) = h$

Hash Table



```
struct Pair {
    void * key;        ⟵ hold key of any type
    int hash_key;      ⟵ hash stored to avoid re-computation
    void * value;      ⟵ holds value of any type
}
```

Implementation detail : if we support generic key (void *) how would we compare two such keys?

hence, we would need a custom comparator for the type

Implementation detail: when we delete a key from the hash table it may be our responsibility to clean them up [manual memory management]

Hence, as part of robust implementation we would need comparator function and destructor function

↓ for key

↙ for key ↘ for value

With Hash Map, we never care about the value

all accesses are key-based

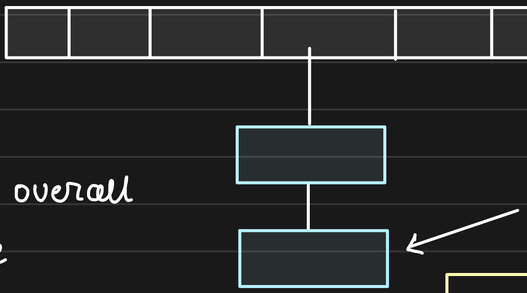
↓

insert, update, delete, lookup

Implementation Detail

Putting the same key, again should not raise an error, instead it should be same as update

Implementing HashMap with Hash Table with Chaining



HashMap overall will have

1. array
2. # size
3. # keys
4. comparator function
5. destructor func (key)
6. destructor func (value)

↑

We will need to invoke the destructor function (if provided) when key/value is deleted

```
struct node {
    int32 hash_key;
    void * key;
    void * value;
    struct node * next;
}
```

application key

hash of the key to avoid re-computation

value to be stored against the key

Implementation Detail

lookup function would return the value for the provided key, and NULL if it does not exist

Implementation Detail

To avoid duplicate keys, we always check and insert

Approach 1: if key present, do not insert at all

Approach 2: if key present, delete old key and re-insert

↳ advantage: it would bring the key at the head of the list

Approach 3: if key present, insert the key without deleting

↳ multiple instances of the same key → space infficient

↙ lookup would return the first match

↓ delete becomes expensive [we have to delete all instance of the same key]

↓ fast inserts

Implementing HashMap with Hash Table with Open Addressing

```
struct slot {
    bool is_empty;
    bool is_deleted;
    int32 hash_key;
    void * key;
    void * value;
}
```

marks if slot is empty

marks if key is soft deleted

hash of the key to avoid re-computation

application key

application value

HashMap overall will have

1. array
2. # size
3. # used slots
4. # active keys
5. comparator function
6. destructor func (key)
7. destructor func (value)

Implementation Detail

During insert, lookup and delete when we find the matching hash key we need to explicitly compare the keys to check its existence.

Implementation Detail

for both key and value

Destructors should be invoked only when we are hard deleting the key