​Hashing and Chaining

#### MIT OCW: Erik Demaine

### Keywords:

Big O, log or lg, ADT, AVL

## Hash tables Data structures

### Intro

Usually called a dictionary or ADT (abstract data type). Maintain set of items, each item has a key. Operations you can perform:

1. insert(item): over writes an existing key.
2. delete(item)
3. search(key): return item with a given key.

So this is different from a BST. In BST if the key is not found, you can find the next larger and the next smaller. In this, you are not allowed to do so. Every item has a unique key. So inserting in item with any existing key will overwrite whatever key was there. One way to solve dictionaries, is to use Bounce Binary search trees or AVL trees.

So you can do all of the operations in:

O(log n) , time for AVL.

While we know, the best way to:

* sort is n log n
* search is log n

Turns out this time can be optimized even more to make it a constant time : O(1).

#### Hashing In Python.

the data type is for hashmap is 'dict'.

dict[key] : to look up a key. The analog of search

dict[key=]=val : insert. Analog of insert

del D[key] : delete

In python, an item is: item = (key, value)

### Uses of hashing and dictionary searching:

1. doc-dict: dic for counting words.

2. databases: every database uses either hashing or search trees.

3. every login passowrd table. every time user logs, hash table (dictionary) will run

4. when compiling you need a dictionary: compilers and interpreters

5. routers (I am guessing routing tables) use the same approach to see which devices are connected; servers.

6. substring search. Unix's grep uses the same logic to search.

7. Files and directory synchronization. Hashing to tell if a file changed.

8. Cryptrographic hash functions

### Approach:

1. Direct access table:

The simple approach. Just a regular table of arrays. Index of the arrays is the key. So, indexed by key. If your item was all just integers, this will work.

Problems:

1. Keys may not be integers.

2. Gigantic memory hog! If set of possible keys is huge, then you need one slot in

your array per key. That's a lot of memory (say for storing each item, you need a memory for say every integer item and memory for key, again an int).

Solution:

1. Prehashing: Maps keys to non negative integers. In python we have a

function hash. hash(x) is the prehash of x. Note: python still has some problems:

For example:

This hash() works well for integers. But has some issues with strings, say for example below. This should not happen, because ideally,

hash('/0B')= hash('1010C')= 64.

hash(x) = hash(y) iff x = y.

If you want to define a custom object you may use: \_\_hash\_\_

2. Hashing:

Reduce the universe of all possible keys (integers) to a reasonable size m for the table.

3. Chaining:

Linked list of colliding elements in all slots of hash table. The idea is that, for every possible collision, we store the keys in a list. Worst Case of a hash table is a linked list. Since in the worst case, a hash table of n elements will have all keys that point to the same slot. This will result in (3) i.e chaining. I.e the cost of linked list is the worst case of hashing.

This is essentially bad, but in practice it doesn't happen because of randomization.

### Simple Uniform Hashing