Hashing with Chaining

Dictionaries as an Abstract Data Type

→ Maintains set of items, each with a key

Operations

↳ Insert(item): (overwrite existing key)

↳ Delete(item)

↳ Search(key)

↳ Returns item if found or raises dict.KeyError (KeyError in Python)

Python Dictionary

D[key] ← Search

D[key] = val ← Insert

del D[key] ← delete

item = (key, value)

Simple Approach to Dictionary

Direct Access Tables

← Store items in the array at x across

and null if the normal index is empty

↳ An array, where the index are the keys in the key

↳ In python, the keys in the array are restricted to only nonnegative ints 0…n-1

Cons

① Keys may not be nonnegative ints

② Memory Hog

Solution to Integer Problem

① Prehashing

→ Maps keys to nonnegative integers (maybe you have a string or an object, etc. How do you map these things to some integer)

Theory: keys are finite and discrete. e.g. → Anything on the computer can be represented by a string of bits, so a string of bits represent an integer

In-Practice

→ Python's hash(x) function is the prehash of x, that maps any hashable object in Python to an integer

↳ In Python, it may not necessarily be a perfect hash.

↳ hash(1) == hash(2)

→ Ideally, you only want the hash(x) to return the hash(y) when they are the same thing

→ __hash__ is a built-in that can be overwritten. The default is ID (which was your object's address in memory)

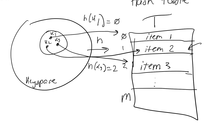
↳ This is subject to change overtime, e.g. if you're item moves places in memory.

↳ It has to be implemented such that your items don't change overtime

Solution to ② & Chaining

→ Reduce universe $\mathcal{U}$ of all keys (usually) down to reasonable size m for table

Hash table



→ We want m to be the size of n

⇓

We want the table to be roughly the size of the number of keys in the dictionary ⇒ $\Theta(n) = m$

→ If we want to store n things, then it will be linear, using maybe n or 2n or 3n space

Problems

→ The hash table is much more smaller than the set of all possible keys in the $\mathcal{U}$, meaning that 2 keys that map to the same spot in the hash table (known as a collision)

Simple Uniform Hashing

→ Each key is equally likely to be hashed to any slot of the table, independent of where other keys hash

Two Probabilistic Assumptions

① Uniformity – If you take some key in the space of m that you want to store, the hash function maps it to a uniform random

↳ e.g. each slot is equally likely to be hashed to

② Independence → Uniformity is true for all keys

↳ If key 1 maps to a uniform random place, then no matter where it goes, key 2 also maps to a uniform random place.

↳ This isn't technically true

Analysis of Hashing w/Chaining

→ We want to know the expected length of a chain

↳ If we have n keys mapped to m slots in a table, then α is the expected length of a chain

Expected length of Chain = n/m

↳ The chance of a key going to a particular slot is 1/m. Since they are all independent, you add them as 1/m + 1/m … 1/m for n keys, making it n/m

n/m = α ← The "Load Factor"

if n = m then length of a chain is 1…if it's 10, then you have to times as many elements as you do slots… This may seem like a lot, but it's still linear so fine.

O(1) if m ≈ Θ(n)

Running Time

→ You look up your hash in the table and traverse the list until you find your element. If the element doesn't exist, you have to traverse the entire list.

Running Time = O(1 + chain)

O(1 + α)

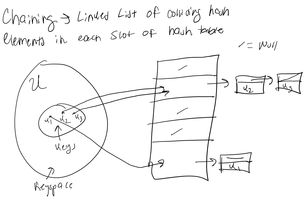
* Assume Hash Function takes const of time

* Following Point of takes const time

Collisions/Chaining

Collisions → $h(k_i) = h(k_j)$ but $k \neq j$

Chaining → Linked List of colliding hash elements in each slot of hash table



↳ You store keys at their respective slots in the hash table as linked lists

↳ If there are other keys already mapped to a particular slot, then you chain them on as the next node in list

Complexity

Worst case is $\Theta(n)$ because if you happen to store n keys that all map to the same slot, then you will only have a linear list of n items.

In Practice hashing works well because the hash function randomly computes your keys, giving you a nice distribution, where elements in the table typically only take $\Theta(1)$ to access

Hash functions

① Division Method

$h(k) = k \bmod n$

↳ Gives you a number between m and n-1

↳ Choose a prime number for m so there aren't too many common factors

↳ Don't choose numbers like powers of 2 or 10 because these are common

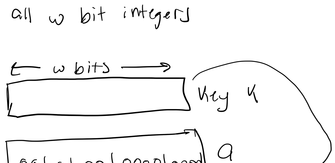
→ Not great

② Multiplication Method

$h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w-r)$

↳ We assume that we are in a w bit machine

↳ We take a a random integer among all w bit integers



↳ Results: key k that is w bits long

→ Multiplying two words gives you a double.

$\lfloor (a \cdot k) \bmod 2^w \rfloor >> (w-r)$

$m = 2^r$, where r is a number between 0 and n-1?

→ The intuition is that doing an or these crazy operation will shift the bits significantly enough to randomize the number

* A should be odd and not close to a power of 2

③ Universal Hashing

$h(k) = \lfloor (ak + b) \bmod p \rfloor \bmod m$

key ← random ∈ {1…p-1}

Prime > |U|

Worst Case

For keys $k_1 \neq k_2$:

Probability $\{h(k_1) = h(k_2)\} = 1/m$

↳ Probability of two keys colliding is 1/m over a and b