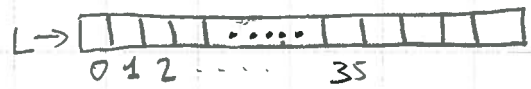
6.006 Lecture 5: Hashing I

- Naming data, fast docdist, direct addressing
- Hash functions - general idea
- 3 questions in the design of hash tables
- Chaining
- Analysis, simple uniform hashing analysis
- ~~An~~ Example of good hash functions

- Problem sets: How are students doing?
- Lab assistant: hours, give us feedback

Forms of Naming

```
L=[]
L[0] = True          Θ(1)
⋮
if L[35]: ...        Θ(1)
```



address(L[35]) =
address(L) + 35 · width of elt

Naming elements of a list (array) using integers is very efficient, if the range of integers is not too large.

The ability to use <u>arbitrary names</u> and still be able to insert(key, value), delete(key), search(key) is very useful: in $\Theta(1)$ time

```
def count_frequency(word_list):
    D={}                Python dictionary ≡ hash table
search → for word in word_list:
            if word in D: D[word] += 1   ← search, insert
            else: D[word]=1              ← insert
```

```
def inner_product(D1, D2):
    sum = 0
    for word in D1:
        if word in D2:
            sum += D1[word] * D2[word]
```

$\Theta(n)$ if insert, search take $\Theta(1)$ ⇒ optimal

Hash Functions

Keys (names) that are natural for the application are not always small integers:

- set of length-20 strings over alphabet A, T, G, C
- credit-card #'s
- English words

PS2

Idea: define a "random-looking" function h from set U of keys to the set $\{0, 1, \cdots m-1\}$ of indices of table T, store (x, v) in $T[h(x)]$

Such a function is built into Python: hash(x) gives a 32-bit integer, and hash(x) % m gives a number in $\{0, 1, \cdots m-1\}$ (x must be immutable; more in recitation)

Three questions

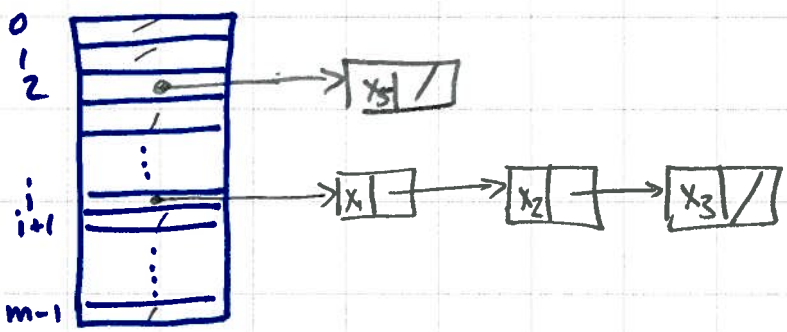
1. How to choose m?
2. How to compute $h(x)$?
3. How to insert both x and y if $h(x) = h(y)$? (collision)

One approach for dealing with collisions: <u>chaining</u>

(anothe approach, open addressing, next week)

Hash table $T[0..m-1]$

$T[i]$ is a <u>list</u> of elements that have $h(x)=i$



$h(x_5)=2$

$h(x_1)=h(x_2)=h(x_3)=i$

$T[i]$ can be a Python list (array that Python grows dynamically) or a linked list.

Analysis of Hashing with Chaining

n items in a table of size m

worst-case: all hash to position i

$T[i]$ has length n

search, delete takes $\Theta(n)$ time

(insert is still $\Theta(1)$)

Let's demand less: expected time under an assumption

Simple Uniform Hashing Assumption

Each key is equally likely to hash to any slot, independent of where other keys are hashed to

Load Factor $\alpha = n/m$ = average keys/slot

- time to do search/delete: $\Theta(1+\alpha)$

  compute $i=h(x)$ — search list $T[i]$

- $m=\Omega(n) \Rightarrow \alpha=O(1) \Rightarrow \Theta(1+\alpha)=\Theta(1)$
- time to enumerate: $\Theta(m+n)$

  enumerate T — enumerate all the lists in T

- $m=O(n) \Rightarrow \Theta(m+n)=\Theta(n)$

- We want both $m=\Omega(n)$ and $m=O(n)$ so we aim for $m=\Theta(n)$, ~~say~~ say $\frac{n}{4} \leq m \leq 4n$

- This almost addresses question (1), but next time we'll see how to resize dynamically

Computing $h(x)$ (question ②)

Lots of ways. Here's a good one for integer x's (large integers)

Let p be a prime $p \geq m$

pick a $0 < a < p$

pick b $0 < b < p$

let $h(x) = ((ax+b) \bmod p) \underbrace{\bmod m}_{\text{only if } p > m}$

e.g $m = 1{,}000{,}000$

$$\left.\begin{array}{l} p = 1{,}000{,}003 \\ a = 314{,}159 \\ b = 271{,}828 \end{array}\right\} \text{can reuse } p, a, b \text{ for smaller } m\text{'s}$$

If keys are not integers, convert them first to integers

x="ATTGCTAC" treat as a base-4 integer

x="Boston" treat as a base-52 integer
or base-26
or base 128...

Textbook describes other methods.