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

Problem sets: How are students doing?
Lab assistant: hours, give us feedback
Forms of Naming

\[
L = [] \\
L[0] = \text{True} \\
\vdots \\
\text{if } L[35]: \ldots \\
\Theta(1)
\]

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

The ability to use arbitrary names and still be able to \text{insert}(key, value) \text{, delete}(key) \text{, and search}(key) in \Theta(1) time

```python
def count_frequency(word_list):
    D = {}
    for word in word_list:
        if word in D:
            D[word] += 1
        else:
            D[word] = 1

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

\Theta(n) \text{ is } \text{insert, search take } \Theta(1) = \text{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

Idea: define a "random-looking" function $h$ from set $U$ of keys to the set $\{0,1,\ldots,m-1\}$ 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 $\{0,1,\ldots,m-1\}$
(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: chaining (another approach, open addressing, next week)

Hash table \( T[0..m-1] \)

\( T[i] \) is a list of elements that have \( h(x) = i \)

\[ T[0] \] 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 Θ(n) time
  - (insert is still Θ(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 \( \lambda = \frac{n}{m} \), average keys / slot
- time to do search/delete: $\Theta(1 + \alpha)$
  \[ \text{compute } i = \pi(x) \]
  \[ \text{search list } \text{T}[i] \]

- $m = \mathcal{L}(n) \Rightarrow \alpha = O(1) \Rightarrow \Theta(1 + \alpha) = \Theta(1)$
- time to enumerate: $\Theta(m + n)$
  \[ \text{enumerate all } \text{T} \text{ in } \text{T} \]

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

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

- This almost addresses question 4, 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 $p > m$

pick $a \quad 0 < a < p$

pick $b \quad 0 < b < p$

let $h(x) = ((ax+b) \mod p) \mod m \quad \text{only if } p > m$

\[ \text{e.g. } \begin{cases} m = 1,000,000 \\ p = 1,000,003 \\ a = 314,159 \\ b = 271,828 \end{cases} \] 

\{ \text{can reuse } p, a, b \text{ for smaller } m \text{’s} \}

If keys are not integers, convert them first to integers

$x = "ATTC-CTAC"$ treat as a base-4 integer

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

Textbook describes other methods.