Lecture Overview

- Dictionaries and Python
- Motivation
- Hash functions
- Chaining
- Simple uniform hashing
- “Good” hash functions

Readings

CLRS Chapter 11.1, 11.2, 11.3.

Dictionary Problem

Abstract Data Type (ADT) maintains a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists
- assume items have distinct keys (or that inserting new one clobbers old)
- balanced BSTs solve in $O(\lg n)$ time per op. (in addition to inexact searches like nextlargest).
- goal: $O(1)$ time per operation.

Python Dictionaries:

Items are (key, value) pairs e.g. $d = \text{'algorithms': 5, \text{'cool': 42}}$

\begin{align*}
d.\text{items}() & \rightarrow [(\text{'algorithms'}, 5), (\text{'cool'}, 5)] \\
d[\text{'cool'}] & \rightarrow 42 \\
d[42] & \rightarrow \text{KeyError} \\
\text{\text{'cool'} in d} & \rightarrow \text{True} \\
42 \text{ in d} & \rightarrow \text{False}
\end{align*}

Python \texttt{set} is really \texttt{dict} where items are keys.
Motivation

Document Distance

• already used in

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

• new docdist7 uses dictionaries instead of sorting:

```python
def inner_product(D1, D2):
    sum = φ. φ
    for key in D1:
        if key in D2:
            sum += D1[key]*D2[key]

⇒ optimal Θ(n) document distance assuming dictionary ops. take O(1) time
```

PS2

How close is chimp DNA to human DNA?
= Longest common substring of two strings
  e.g. ALGORITHM vs. ARITHMETIC.
  Dictionaries help speed algorithms e.g. put all substrings into set, looking for duplicates
  - Θ(n^2) operations.
How do we solve the dictionary problem?

A simple approach would be a direct access table. This means items would need to be stored in an array, indexed by key.

<table>
<thead>
<tr>
<th>key</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>key</td>
<td>item</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Direct-access table

Problems:

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range ⇒ large space e.g. one key of $2^{256}$ is bad news.

2 Solutions:

Solution 1: map key space to integers.

- In Python: hash (object) where object is a number, string, tuple, etc. or object implementing — hash — Misnomer: should be called “prehash”
- Ideally, $x = y$ ⇔ hash($x$) = hash ($y$)
- Python applies some heuristics e.g. hash('\phiB') = 64 = hash('\phiC')
- Object’s key should not change while in table (else cannot find it anymore)
- No mutable objects like lists
Solution 2: hashing (verb from ‘hache’ = hatchet, Germanic)

- Reduce universe $U$ of all keys (say, integers) down to reasonable size $m$ for table
- idea: $m \approx n$, $n = |k|$, $k = $ keys in dictionary
- hash function $h: U \rightarrow \phi, 1, \ldots, m - 1$

![Diagram of hashing with keys and table]

**Figure 2:** Mapping keys to a table

- two keys $k_i, k_j \in K$ collide if $h(k_i) = h(k_j)$

**How do we deal with collisions?**

There are two ways

1. Chaining: TODAY
2. Open addressing: NEXT LECTURE
Chaining

Linked list of colliding elements in each slot of table

Figure 3: Chaining in a Hash Table

- Search must go through whole list \( T[h(key)] \)
- Worst case: all keys in \( k \) hash to same slot \( \implies \Theta(n) \) per operation

**Simple Uniform Hashing - an Assumption:**

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

\[
\begin{align*}
\text{let } n &= \# \text{ keys stored in table} \\
m &= \# \text{ slots in table} \\
\text{load factor } \alpha &= \frac{n}{m} = \text{average } \# \text{ keys per slot}
\end{align*}
\]

**Expected performance of chaining: assuming simple uniform hashing**

The performance is likely to be \( O(1 + \alpha) \) - the 1 comes from applying the hash function and access slot whereas the \( \alpha \) comes from searching the list. It is actually \( \Theta(1 + \alpha) \), even for successful search (see CLRS).

Therefore, the performance is \( O(1) \) if \( \alpha = O(1) \) i.e. \( m = \Omega(n) \).
Hash Functions

Division Method:

\[ h(k) = k \mod m \]

- \( k_1 \) and \( k_2 \) collide when \( k_1 = k_2 \mod m \) i.e. when \( m \) divides \( |k_1 - k_2| \)
- fine if keys you store are uniform random
- but if keys are \( x, 2x, 3x, \ldots \) (regularity) and \( x \) and \( m \) have common divisor \( d \) then use only \( 1/d \) of table. This is likely if \( m \) has a small divisor e.g. 2.
- if \( m = 2^r \) then only look at \( r \) bits of key!

**Good Practice:** A good practice to avoid common regularities in keys is to make \( m \) a prime number that is not close to power of 2 or 10.

**Key Lesson:** It is inconvenient to find a prime number; division slow.

Multiplication Method:

\[ h(k) = [(a \cdot k) \mod 2^w] \gg (w - r) \]

where \( m = 2^r \) and \( w \)-bit machine words and \( a = \) odd integer between \( 2(\cdot w - 1) \) and \( 2^w \).

**Good Practice:** \( a \) not too close to \( 2^{(w-1)} \) or \( 2^w \).

**Key Lesson:** Multiplication and bit extraction are faster than division.

![Figure 4: Multiplication Method](image-url)