

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 1

Warmup

Exercise

- ▶ How fast can we query/insert with these data structures?

	Query	Insert
Unsorted linked list		
Unsorted array		
Sorted array		
BST		

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 2

Direct Address Tables

Counting Frequencies

- ▶ How many times does each age appear?

PID	Name	Age
A1843	Wan	24
A8293	Deveron	22
A9821	Vinod	41
A8172	Aleix	17
A2882	Kayden	4
A1829	Raghu	51
A9772	Cui	48
⋮	⋮	⋮

Exercise

What data structure would you use to store the age counts?

Direct Address Tables

- ▶ Idea: keep an **array** `arr` of length, say, 125.
- ▶ Initialize to zero.
- ▶ If we see age x , increment `arr[x]` by one.

Building the Table

```
# loading the table  
table = np.zeros(125)  
  
for age in ages:  
    table[age] += 1
```

- ▶ Time complexity if there are n people?

Query

```
# query: how many people are 55?  
print(table[55])
```

- ▶ Time complexity if there are n people?

Counting Names

- ▶ How many times does each name appear?

PID	Name	Age
A1843	Wan	24
A8293	Deveron	22
A9821	Vinod	41
A8172	Aleix	17
A2882	Kayden	4
A1829	Raghu	51
A9772	Cui	48
⋮	⋮	⋮

Downsides

- ▶ DATs are **fast**.
- ▶ What are the downsides of DATs?
- ▶ Could we use a DAT to store:
 - ▶ zip codes?
 - ▶ phone numbers?
 - ▶ credit card numbers?
 - ▶ names?

Downsides

- ▶ Things being stored must be integers, or convertible to integers
 - ▶ why? valid array indices
- ▶ Must come from a small range of possibilities
 - ▶ why? memory usage. example: phone numbers

Hash Tables

- ▶ Insight: anything can be “converted” to an integer through **hashing**.
- ▶ But not uniquely!
- ▶ Hash tables have many of the same advantages as DATs, but work more generally.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 3

Hashing

Hashing

- ▶ One of the most important ideas in CS.
- ▶ Tons of uses:
 - ▶ Verifying message integrity.
 - ▶ Fast queries on a large data set.
 - ▶ Identify if file has changed in version control.

Hash Function

- ▶ A **hash function** takes a (large) object and returns a (smaller) “fingerprint” of that object.
- ▶ Usually the fingerprint is a number, guaranteed to be in some range.

How?

- ▶ Looking at certain bits, combining them in ways that *look* random (but aren't!)

Hash Function Properties

- ▶ Hashing same thing twice returns the same hash.
- ▶ Unlikely that different things have same fingerprint.
 - ▶ But not impossible!

Collisions

- ▶ Hash functions map objects to numbers in a defined range.
 - ▶ Example: given image, return number in $[0, 1, 2, \dots, 1024]$
- ▶ There will be two images with the same hash.
 - ▶ **Pigeonhole principle**: if there are n pigeons, $< n$ holes, there will a hole with ≥ 2 pigeons.
- ▶ **Collision**: two objects have the same hash

“Good” Hash Functions

- ▶ A good hash function tries to minimize collisions.

Hashing in Python

- ▶ The `hash` function computes a hash.

```
>>> hash("This is a test")
```

```
-670458579957477203
```

```
>>> hash("This is a test")
```

```
-670458579957477203
```

```
>>> hash("This is a test!")
```

```
1860306055874153109
```

MD5

- ▶ MD5 is a **cryptographic** hash function.
 - ▶ Hard to “reverse engineer” input from hash.

- ▶ Returns a *really large* number in hex.

a741d8524a853cf83ca21eabf8cea190

- ▶ Used to “fingerprint” whole files.

Example

```
> echo "My name is Akbar" | md5
a741d8524a853cf83ca21eabf8cea190
> echo "My name is Akbar" | md5
a741d8524a853cf83ca21eabf8cea190
> echo "My name is Akbar!" | md5
f11eed2391bbd0a5a2355397c089fafd
```

Example

```
> md5 slides.pdf  
e3fd4370fda30ceb978390004e07b9df
```

Why?

- ▶ I release a piece of software.
- ▶ I host it on Google Drive.
- ▶ Someone (Google, US Gov., etc.) decides to insert extra code into software to spy on users.
- ▶ You have no way of knowing.

Why?

- ▶ I release a piece of software & **publish the hash.**
- ▶ I host it on Google Drive.
- ▶ Someone inserts extra code.
- ▶ You download the software and hash it. If hash is different, you know the file has been changed!

Another Use: De-duplication

- ▶ Building a massive training set of images.
- ▶ Given a new image, is it already in my collection?
- ▶ Don't need to compare images pixel-by-pixel!
- ▶ Instead, compare **hashes**.

Hashing for Data Scientists

- ▶ Don't need to know much about *how* the hash function works.
- ▶ But should know how they are used.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 4

Hash Tables

Membership Queries

- ▶ **Given:** a collection of n numbers and a target t .
- ▶ **Find:** determine if t is in the collection.

Goal

- ▶ DATs are fast, but won't work for things that aren't numbers in a small range.
- ▶ Idea: hash objects to numbers in a small range, use a DAT.
- ▶ But must deal with collisions.

Hash Tables

- ▶ Pick a table size m .
 - ▶ Usually $m \approx$ number of things you'll be storing.
- ▶ Create hash function to turn input into a number in $\{0, 1, \dots, m - 1\}$.
- ▶ Create DAT with m bins.

Example

`hash('hello')` == 3

`hash('data')` == 0

`hash('science')` == 4

0 1 2 3 4 ... $m - 1$

Collisions

- ▶ The **universe** is the set of all possible inputs.
- ▶ This is usually much larger than m (even infinite).
- ▶ Not possible to assign each input to a unique bin.
- ▶ If $\text{hash}(a) == \text{hash}(b)$, there is a **collision**.

Example

```
hash('hello') == 3  
hash('data') == 0  
hash('san diego') == 3
```

0 1 2 3 4 ... $m - 1$

Chaining

- ▶ Collisions stored in same bin, in linked list.
- ▶ **Query:** Hash to find bin, then linear search.



The Idea

- ▶ A good hash function will utilize all bins evenly.
 - ▶ Looks like uniform random distribution.
- ▶ If $m \approx n$, then only a few elements in each bin.
- ▶ As we add more elements, we need to add bins.

Average Case

- ▶ n elements in table.
- ▶ m bins.
- ▶ Assume elements placed randomly in bins¹.
- ▶ Expected bin size:

¹Of course, they are placed deterministically.

Average Case

- ▶ n elements in table.
- ▶ m bins.
- ▶ Assume elements placed randomly in bins¹.
- ▶ Expected bin size: n/m

¹Of course, they are placed deterministically.

Analysis

- ▶ Query:
 - ▶ Time to find correct bin:
 - ▶ Expected number of elements in the bin:
 - ▶ Time to perform linear search:
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin:
 - ▶ Time to perform linear search:
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search:
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total:

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total: $\Theta(1 + n/m)$

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total: $\Theta(1 + n/m)$
 - ▶ We usually guarantee $m = \Theta(n)$

Analysis

- ▶ Query:
 - ▶ Time to find correct bin: $\Theta(1)$
 - ▶ Expected number of elements in the bin: n/m
 - ▶ Time to perform linear search: $\Theta(n/m)$
 - ▶ Total: $\Theta(1 + n/m)$
 - ▶ We usually guarantee $m = \Theta(n)$
 - ▶ Expected time: $\Theta(1)$.

Worst Case

- ▶ Everything hashed to same bin.
 - ▶ Really unlikely!
- ▶ Query:
 - ▶ $\Theta(1)$ to find bin
 - ▶ $\Theta(n)$ for linear search.
 - ▶ Total: $\Theta(n)$.

Exercise

What is the worst case time complexity of inserting an element into a hash table that uses chaining with linked lists?

Growing the Hash Table

- ▶ Insertions take $\Theta(1)$ **unless** the hash table needs to grow.
- ▶ We need to ensure that $m \leq c \cdot n$.
 - ▶ Otherwise, too many collisions.
- ▶ If we add a bunch of elements, we'll need to increase m .
- ▶ Increasing m means allocating a new array, $\Theta(m) = \Theta(n)$ time.

Main Idea

Hash tables support constant (expected) time insertion and membership queries.

Dictionaryes

- ▶ Hash tables can also be used to store (key, value) pairs.
- ▶ Often called **dictionaries** or **associative arrays**.

Hashing in Python

- ▶ `dict` and `set` implement hash tables.
- ▶ Querying is done using `in`:

```
>>> # make a set
>>> L = {3, 6, -2, 1, 7, 12}
>>> 4 in L # Theta(1)
False
>>> 7 in L # Theta(1)
True
```

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 5

Fast Algorithms with Hash Tables

Faster Algorithms

- ▶ Hashing is a super common trick.
- ▶ The “best” solution to interview problems often involves hashing.

Example 1: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ Find two whose durations sum to **exactly** D .

Recall: Previous Solutions

- ▶ Brute force: $\Theta(n^2)$.
- ▶ Sort, use sorted structure: $\Theta(n \log n) + \Theta(n)$.
- ▶ Theoretical lower bound: $\Omega(n)$?
- ▶ Can we speed this up with hash tables?

Idea

- ▶ To use hash tables, we want to frame problem as a **membership query**.

Example

- ▶ Suppose flight is 360 minutes long.
- ▶ Suppose first movie is fixed: 120 minutes.
- ▶ Is there a movie lasting $(360 - 120) = 140$ minutes?

```
def optimize_entertainment_hash(times, D):
    hash_table = dict()
    for i, time in enumerate(times):
        hash_table[time] = i

    for i, time in enumerate(times):
        target = D - time
        if target in hash_table:
            return i, hash_table[target]
```

Example 2: Anagrams

Definition

Two strings w_1 and w_2 are **anagrams** if the letters of w_1 can be permuted to make w_2 .

Examples

- ▶ abcd / dbca
- ▶ listen / silent
- ▶ sandiego / doginsea

Problem

- ▶ Given a collection of n strings, determine if any two of them are anagrams.

Exercise

Design an efficient algorithm for solving this problem. What is its time complexity?

Solution

- ▶ We need to turn this into a **membership query**.
- ▶ **Trick:** two strings are anagrams iff

`sorted(w_1) == sorted(w_2)`

```
def any_anagrams(words):  
    seen = set()  
    for word in words:  
        w = sorted(word)  
        if w in seen:  
            return True  
        else:  
            seen.add(w)
```


Hashing **Downsides**

- ▶ Problem must involve **membership query**.

Example: The Movie Problem

- ▶ You're on a flight that will last D minutes.
- ▶ You want to pick two movies to watch.
- ▶ Find two whose added durations is **closest** to D .

Hashing **Downsides**

- ▶ No locality: similar items map to different bins.
- ▶ There is no way to quickly query entry closest to given input.

Example: Number of Elements

- ▶ Given a collection of n numbers and two endpoints, a and b , determine how many of the numbers are contained in $[a, b]$.
- ▶ Not a membership query.
- ▶ Idea: **sort** and use modified binary search.

DSC 40B

Theoretical Foundations II

Lecture 9 | Part 6

Hash Table Drawbacks

Hashing **Downsides**

- ▶ No locality: similar items map to different bins.
- ▶ But we often want similar items at the same time.
- ▶ Results in many **cache misses**, **slow**.

Hashing **Downsides**

- ▶ Memory overhead.

Hash Tables vs. BSTs

- ▶ Hash Table: $\Theta(1)$ insertion, query (expected time).
- ▶ BST: $\Theta(\log n)$ insertion, query (if balanced).
- ▶ Why ever use a BST?

Hash Tables vs. BSTs

- ▶ Hash tables keep items in arbitrary order.
- ▶ Example: how many elements are in the interval $[3, 23]$?
- ▶ Example: what is the min/max/median?
- ▶ BSTs win when order is important.