

# DSC40B: Theoretical Foundations of Data Science II

## Lecture 9: *Hashing and Hash table*

Instructor: Yusu Wang

# Recall: Set operations

---

- ▶ Imagine you are maintaining a database indexed by some keys (real values), and you hope to support the following operations:

- ▶ Search
- ▶ Maximum
- ▶ Minimum
- ▶ Successor
- ▶ Predecessor

Using balanced BST, all these operations can be done in  $O(\lg n)$  time

- ▶ Insert
- ▶ Delete
- ▶ Extract-Max
- ▶ Increase-key



# Dictionary operations

---

- ▶ Given a universe of elements  $U$ 
  - ▶ these elements may not be numbers
- ▶ Need to store some keys
- ▶ Need to perform the following operations for keys
  - ▶ Search (membership queries)
  - ▶ Insert
  - ▶ Delete

We can use balanced BST for this purpose if keys can be compared.

But we can have even lighter weight data structure to handle these



# Today

---

- ▶ Hashing in general
- ▶ Hash table
  - ▶ For dictionary operations



---

# Hashing



# Hashing

---

- ▶ An important idea used commonly in practice
- ▶ Many uses:
  - ▶ Fast queries on a large data set.
  - ▶ Verifying message integrity.
  - ▶ Identify if file has changed in version control.



# Hash function

---

- ▶ Mathematically, a hash function is simply a function
  - ▶  $f: U \rightarrow X$  from one set to another
- ▶ To make it useful, in practice,
  - ▶ It often maps some potentially large or complex object to a much smaller and simpler “fingerprint” or “signature”
  - ▶ One also wants the mapping to be “uniform” and cause few “collisions”.
  - ▶ Note: a hash function needs to be deterministic!
    - ▶ Hashing the same object twice, we should get the same answer



# Some examples

---

- ▶ A cryptographic hash function:

- ▶ maps data of arbitrary size into an often fixed-sized output of much smaller size
- ▶ e.g, MD5: maps it to a 128-bit value
- ▶ hard to “reverse engineer” input from hash.
- ▶ two similar input could and should lead to very different hash values

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

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





# A cryptographic hash function

---

## ▶ 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.

## ▶ What do I do?

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



# Some examples

---

- ▶ **Want to place images into 100 bins.**
  - ▶ How do we decide which bin an image goes into?
  - ▶ Hash function!
  - ▶ Takes in an image.
  - ▶ Outputs a number in  $\{1, 2, \dots, 100\}$



---

# Hash Table



# Dictionary operations

---

- ▶ Given a universe of elements  $U$
- ▶ Need to store some keys
- ▶ Need to perform the following operations for keys
  - ▶ Insert
  - ▶ Search
  - ▶ Delete
- ▶ In other words, the key operation is **membership queries** (search), but also allow dynamic updates (insert, delete).



# Approach 0

---

- ▶ Use an array to organize all the keys
  - ▶ We could pre-sort the array.
    - ▶ Preprocessing time:
  - ▶ Search:
  - ▶ Insert / Delete:



# Approach 1

---

- ▶ Organize all keys in a doubly-linked list
  - ▶ Pre-processing: **None**
  - ▶ Insert:
  - ▶ Delete
  - ▶ Search:



# Approach 2

---

- ▶ **Organize all keys in a balanced BST**
  - ▶ Search:
  - ▶ Insert:
  - ▶ Delete



# Approach 3

---

- ▶ Direct address tables
- ▶ Suppose we know that all the keys we ever care are from 0 to  $N = 99,999$ 
  - ▶ e.g, we are querying for zipcodes
- ▶ Open an array  $A$  of size  $N$ 
  - ▶ If a zipcode  $z$  is in, set  $A[z] = 1$ , and 0 otherwise
  - ▶ Given a query zipcode, say 22330, simply return  $A[22330]$
- ▶ Search:
- ▶ Insert:
- ▶ Delete:

What's the problem with this approach?

Not practical for if the size of universe  $U$  (which is  $N$  in this example) is huge!





# Use Hash Table!

---

- ▶  $U$  : universe
- ▶  $T[0 \dots m - 1]$  : a hash table of size  $m$ 
  - ▶  $m \ll |U|$
  - ▶ usually, we choose  $m$  to be around the size of data we will see
- ▶ Hash functions
  - ▶  $h: U \rightarrow \{0, 1, \dots, m - 1\}$ 
    - ▶ i.e,  $h$  maps each element in the universe to an index in the hash-table
- ▶  $h(k)$  is called the **hash value** of key  $k$ .
  - ▶ Given a key  $k$ , we will store it in location  $h(k)$  of hash table  $T$ ,
    - ▶ i.e, store it at  $T[h(k)]$



# Collision

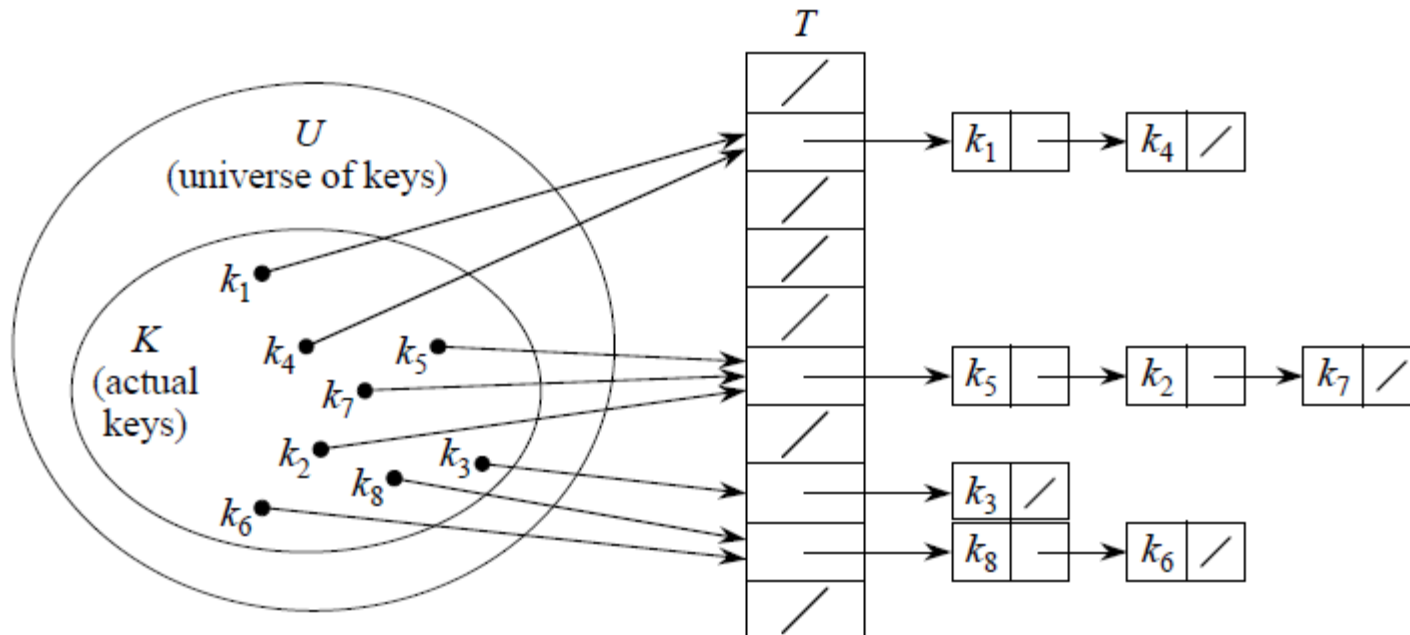
---

- ▶ Since the size of hash table is smaller than the universe:
  - ▶ Multiple keys may hash to the same slot.
  - ▶ A **collision** happens when  $h(x) = h(y)$  for  $x \neq y \in U$
- ▶ How to handle collisions?
  - ▶ Chaining
  - ▶ Open addressing
  - ▶ ...



# Collision Resolved by Chaining

---



- ▶  $T[j]$ : a pointer to the head of the linked list of all stored elements that hash to  $j$
  - ▶ Nil otherwise
- 



# A simple example

---

- ▶  $U$ : all positive integers  $\mathbb{N}$
- ▶ Hash table  $T[0, \dots, 10]$
- ▶ Hash function:  $h: \mathbb{N} \rightarrow [0, \dots, 10]$ 
  - ▶  $h(x) = x \bmod 11$



# Dictionary operations

---

- ▶ Chained-Hash-Insert( $T, x$ )

- ▶ Insert  $x$  at the head of list  $T[h(x)]$

$O(1)$

- ▶ Chained-Hash-Search( $T, x$ )

- ▶ Search for an element with key  $x$  in list  $T[h(x)]$

$O(\text{length}(T[h(x)]))$

- ▶ Chained-Hash-Delete( $T, x$ )

- ▶ Delete  $x$  from the list  $T[h(x)]$

$O(\text{length}(T[h(x)]))$



# Good Hash Function

---

- ▶ Performance of Hash table operations

- ▶ depend on the number of elements hashed to each slot in hash table.

- ▶ Intuitively,

- ▶ A good hash function should spread elements into the hash table uniformly
  - ▶ If there are  $n$  elements in the input data and  $m$  slots
    - ▶ then ideally there should be  $\frac{n}{m}$  elements in each hash table slot.



# Average case analysis

---

- ▶  $n$ : # elements in the table
- ▶  $m$ : size of table (# slots in the table)
- ▶ **Load factor:**
  - ▶  $\alpha = \frac{n}{m}$  : average number of elements per linked list
  - ▶ Intuitively the optimal time needed
- ▶ Individual operation can be slow ( $\Theta(n)$  worst case time)
  - ▶ *Under certain assumption of the distribution of keys*, analyze expected performance.



# Simple uniform hashing assumption

---

- ▶ Simple uniform hashing assumption:
  - ▶ any given element is equally likely to hash into any of the  $m$  slots in  $T$
- ▶ Let  $n_j$  be length of list  $T[j]$ 
  - ▶  $n = n_0 + n_1 + \cdots + n_{m-1}$
  - ▶ Under simple uniform hashing assumption:
    - ▶ expected value  $E[n_j] = \alpha = \frac{n}{m}$

Why?





# Why

---

- ▶ Let  $\{k_1, k_2, \dots, k_n\}$  be the set of keys
- ▶ Goal: Estimate  $E(n_j)$
- ▶ Let  $X_i = 1$  if  $h(k_i) = j$   
0 otherwise
- ▶ Note:  $n_j = \sum_{i=1}^n X_i$  !
- ▶  $E[X_i] = \Pr[h(k_i) = j] = \frac{1}{m}$
- ▶ Hence  $E[n_j] = E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i] = \sum_{i=1}^n \frac{1}{m} = \frac{n}{m}$



# Expected running time

---

- ▶ Under Simple uniform hashing assumption

- ▶ Search:

- ▶ Expected time:  $ET(n) = \Theta(1 + \frac{n}{m})$

- ▶ (worst case time  $T(n) = \Theta(n)$ )

- ▶ Insert:

- ▶  $T(n) = \Theta(1)$

- ▶ Delete:

- ▶ Expected time:  $ET(n) = \Theta(1 + \frac{n}{m})$

- ▶ (worst case time  $T(n) = \Theta(n)$ )

Key message:

if the load factor  $\alpha = \frac{n}{m} = \Theta(1)$

then  $\Theta(1)$  expected time for these operations!



# Hashing in Python

---

- ▶ `dict` and `set` implement hash tables

- ▶ Querying is done using `in`:

```
>>> # make a set
```

```
>>> L = {3, 6, -2, 1, 7, 12}
```

```
>>> 1 in L # Theta(1)
```

```
False
```

```
>>> 7 in L # Theta(1)
```

```
True
```



---

# Some examples of using hash tables and downsides of hash tables



# Recall the movie problem

---

## ► The Movie problem

- Input: Given a list of length of movies available, stored in array *movies*, and a flight duration  $D$
- Output: Return two movies whose total length =  $D$ ; **None** otherwise.



# Recall

---

- ▶ The naïve algorithm solves it in  $\Theta(n^2)$  time
- ▶ But earlier, we mentioned we can do better by
  - ▶ First sorting the array of movie lengths
  - ▶ Then check for each movie  $x$ , whether there exists one whose length is  $D - \text{length}(x)$
  - ▶ Worst case time complexity  $T(n) = \Theta(n \lg n)$



# New approach via Hashing

---

- ▶ Use Hash table, and frame this as a membership query problem

```
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]  
    return None
```

Expected time?  
 $\Theta(n)$

# Another example

---

## ▶ Anagrams

- ▶ Two strings  $w_1$  and  $w_2$  are **anagrams** if the letters of  $w_1$  can be permuted to make  $w_2$ .
  - ▶ E.g, “**eat**” and “**tea**”, or “**listen**” and “**silent**”

## ▶ The *Anagrams problem*

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





# Using Hash table

---

## ► Observation:

- two strings are anagrams if their sorted lists are equal
  - `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)  
    return False
```

Expected time?  
 $\Theta(n)$



# Hashing Downsides

---

- ▶ Only support dictionary queries
  - ▶ i.e, membership queries + insert / delete
  - ▶ Example 1: cannot be used to query for the two movies whose total time is **closest** to  $D$
  - ▶ Example 2: cannot be used for performing a range query in a list of numbers
    - ▶ Say report the number of numbers fall within range  $[a, b]$



# Hashing Downsides

---

- ▶ Only support dictionary queries
  - ▶ i.e, membership queries + insert / delete
- ▶ No locality: similar items map to very different bins
  - ▶ Necessarily so for the performance of hashing in most cases!
  - ▶ But, in practice, we often query similar objects continuously
    - ▶ May result in many cache misses, slow



# Summary

---

## ▶ Hashing

- ▶ Very useful idea
- ▶ Generates a small signature for a potentially large complex object

## ▶ Hash Table

- ▶ Very practical data structure for **dictionary operations**
  - ▶ Very efficient for these operations, can be constant expected time if the load factor  $\alpha = \frac{n}{m} = \Theta(1)$
- ▶ Especially when the number of keys necessary is much smaller than the size of universe where input objects could come from!
- ▶ In practice, need to choose hash functions properly
  - ▶ There exist intelligent hashing schemes



---

FIN

