

VE281

Data Structures and Algorithms

Hashing: Basics and Hash Function

Learning Objectives:

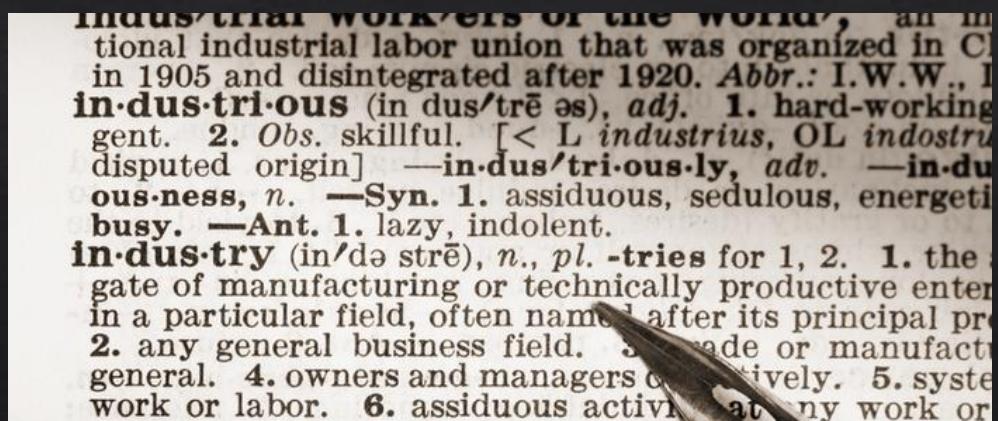
- Know the purpose of hashing
- Understand the collision problem
- Know how to design a good hash function

Outline

- ❖ Review of Dictionary
- ❖ Hashing Basics
- ❖ Hash Function

Dictionary

- ◊ How do you use a dictionary?
 - ◊ Look up a “word” and find its meaning.
- ◊ We also have an abstract data type called dictionary.
 - ◊ It is a collection of pairs, each containing a **key** and an **value**
(key , value)
 - ◊ **Important:** Different pairs have different keys.



Industrial Workers of the World, an international industrial labor union that was organized in Chicago in 1905 and disintegrated after 1920. *Abbr.: I.W.W., I.W.W.*

in-dus-tri-ous (in dus/trē əs), *adj.* 1. hard-working; diligent. 2. *Obs.* skillful. [< L *industrius*, OL *indostru-* disputed origin] —**in-dus/tri-ous-ly**, *adv.* —**in-dus-tri-ous-ness**, *n.* —**Syn.** 1. assiduous, sedulous, energetic, busy. —**Ant.** 1. lazy, indolent.

in-dus-try (in/də strē), *n.*, *pl. -tries* for 1, 2. 1. the gate of manufacturing or technically productive enterprise in a particular field, often named after its principal product. 2. any general business field. 3. trade or manufacture in general. 4. owners and managers collectively. 5. systematic work or labor. 6. assiduous activity at any work or

Dictionary

- ❖ Key space is usually more regular/structured than value space, so easier to search.
- ❖ Dictionary is optimized to quickly **add (key, value)** pair and **retrieve value** by key.

Methods

- ❖ **Value find(Key k)**: Return the value whose key is **k**. Return **Null** if none.
- ❖ **void insert(Key k, Value v)**: Insert a pair **(k, v)** into the dictionary. If the pair with key as **k** already exists, update its value.
- ❖ **Value remove(Key k)**: Remove the pair with key as **k** from the dictionary and return its value. Return **Null** if none.

Runtime for Array Implementation

Pair **Array[MAXSIZE]** :



- ❖ Unsorted array
 - ❖ **find()** $O(n)$
 - ❖ **insert()** $O(n)$: $O(n)$ to verify duplicate, $O(1)$ to put at the end
 - ❖ **remove()** $O(n)$: $O(n)$ to verify existence, $O(1)$ to exchange the “hole” with the last element

- ❖ Sorted array
 - ❖ **find()** $O(\log n)$: binary search
 - ❖ **insert()** $O(n)$: $O(\log n)$ to verify duplicate, $O(n)$ to insert
 - ❖ **remove()** $O(n)$: $O(\log n)$ to verify existence, $O(n)$ to remove

Can we do **find**, **insert**, and **remove** in $O(1)$ time?

Outline

- ❖ Review of Dictionary
- ❖ Hashing Basics
- ❖ Hash Function

Hashing: High-Level Idea

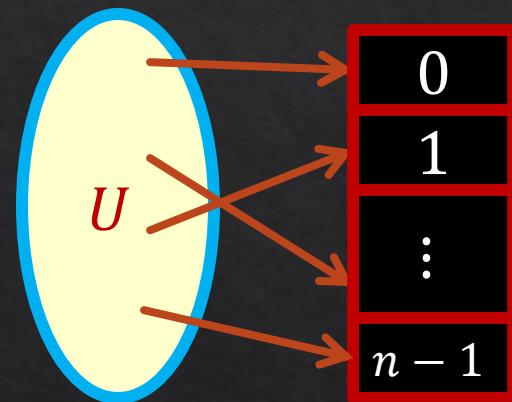
- ❖ **Setup:** A universe U of objects
 - ❖ E.g., All names, all IP addresses, etc.
 - ❖ Generally, very BIG!
- ❖ **Goal:** Want to maintain an evolving set $S \subseteq U$
 - ❖ E.g., 200 students, 500 IP addresses
 - ❖ Generally, of reasonable size.
- ❖ Naïve solutions
 1. Array-based solution (index by $u \in U$)
 - ❖ $\Theta(1)$ operation time, BUT $\Theta(|U|)$ space.
 2. Linked list-based solution:
 - ❖ $\Theta(|S|)$ space, BUT $\Theta(|S|)$ operation time.

Can we get the best of both solutions?

Hashing: High-Level Idea

❖ Solution:

- ❖ Pick an array A of n buckets.
 - ❖ $n = c|S|$: a small multiple of $|S|$.
- ❖ Choose a hash function $h: U \rightarrow \{0, 1, \dots, n - 1\}$
 - ❖ h is fast to compute.
 - ❖ The same key is always mapped to the **same** location.
- ❖ Store item k in $A[h(k)]$



❖ The array is called **hash table**

- ❖ An array of **buckets**, where each bucket contains items as assigned by a hash function.
- ❖ $h[k]$ is called the **home bucket** of key k .

Hashing Example

- ❖ Pairs are: (22,a), (33,b), (3,c), (73,d), (85,e)
- ❖ Hash table is **A[0 : 7]** and table size is **M = 8**
- ❖ Hash function is **h[key] = key/11**
- ❖ Every item with **key** is stored in the bucket **A[h(key)]**

(3,c)		(22,a)	(33,b)			(73,d)	(85,e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Question: What is the time complexity for
find(), **insert()**, and **remove()**?

O(1)

Achieves both fast runtime and efficient memory usage

What Can Go Wrong?

(3,c)		(22,a)	(33,b)			(73,d)	(85,e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- ❖ Where does (35, g) go?
- ❖ Problem: The home bucket for (35, g) is already occupied!
 - ❖ This is a “**collision**”.

Collision and Collision Resolution

- ❖ Collision occurs when the hash function maps two or more items—all having **different** search keys—into the **same** bucket.
- ❖ What to do when there is a collision?
 - ❖ **Collision-resolution scheme**: assigns distinct locations in the hash table to items involved in a collision.
- ❖ Two major schemes:
 - ❖ Separate chaining
 - ❖ Open addressing



Insight of Collision: Birthday Problem

- ❖ Consider n people with random birthdays (i.e., with each day of the year equally likely). What is the smallest n so that there is at least a 50% chance that two people have the same birthday?
 - A. 23
 - B. 57
 - C. 184
 - D. 367

Collision is inevitable!

Hash Table Issues

- ❖ Choice of the hash function.
- ❖ Collision resolution scheme.
- ❖ Size of the hash table and rehashing.

Outline

- ❖ Review of Dictionary
- ❖ Hashing Basics
- ❖ Hash Function

Hash Function Design Criteria

- ❖ Must compute a bucket for every key in the universe.
- ❖ Must compute the same bucket for the same key.
- ❖ Should be easy and quick to compute.
- ❖ Minimizes collision
 - ❖ Spread keys out evenly in hash table
 - ❖ **Gold standard:** **completely random hashing**
 - ❖ The probability that a randomly selected key has bucket i as its home bucket is $1/n$, $0 \leq i < n$.
 - ❖ Completely random hashing **minimizes** the likelihood of an collision when keys are selected at random.
 - ❖ However, completely random hashing is **infeasible** due to the need to remember the random bucket.

The hardest criterion



Bad Hash Functions

- ❖ Example: keys = phone number in China (11 digits)
 - ❖ $|U| = 10^{11}$
 - ❖ **Terrible** hash function: $h(key)$ = first 3 digits of key , i.e., area code
 - ❖ The keys are not spread out evenly. Buckets 010, 021 may have a lot of keys mapped to them, while some buckets have no keys.
 - ❖ **Mediocre** hash function: $h(key)$ = last 3 digits of key .
 - ❖ Still vulnerable to patterns in last 3 digits.

Hash Functions

- ❖ Hash function ($h(key)$) maps key to buckets in two steps:
 1. Convert key into an integer in case the key is not an integer.
 - ❖ A function $t(key)$ which returns an integer value, known as **hash code**.
 2. **Compression map**: Map an integer (hash code) into a home bucket.
 - ❖ A function $c(hashcode)$ which gives an integer in the range $[0, n - 1]$, where n is the number of buckets in the table.
- ❖ In summary, $h(key) = c(t(key))$, which gives an index in the table.

Map Non-integers into Hash Code

- ❖ String: use the ASCII (or UTF-8) encoding of each char and then perform arithmetic on them.
- ❖ Floating-point number: treat it as a string of bits.
- ❖ Images, (viral) code snippets, (malicious) Web site URLs: in general, treat the representation as a bit-string, using all of it or **extracting** parts of it (i.e., www.sjtu.edu.cn).

Strings to Integers

- ❖ Simple scheme: adds up all the ASCII codes for all the chars in the string.
 - ❖ Example: $t(\text{"He"}) = 72 + 101 = 173.$
- ❖ Not good. Why?
 - ❖ Consider English words “post”, “pots”, “spot”, “stop”, “tops”.

Strings to Integers

- ❖ A better strategy: Polynomial hash code taking **positional** info into account.

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

where a is a constant.

- ❖ If $a = 33$, the hash codes for “post” and “stop” are

$$t(\text{post}) = 112 \cdot 33^3 + 111 \cdot 33^2 + 115 \cdot 33 + 116 = 4149734$$

$$t(\text{stop}) = 115 \cdot 33^3 + 116 \cdot 33^2 + 111 \cdot 33 + 112 = 4262854$$

Strings to Integers

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

- ◊ Good choice of a for English words: 31, 33, 37, 39, 41
 - ◊ What does it mean for a to be a **good** choice? Why are these particular values **good**?
 - ◊ Answer: according to statistics on 50,000 English words, each of these constants will produce less than 7 collisions.
- ◊ In Java, its **String** class has a built-in **hashCode()** function. It takes $a = 31$. Why?
 - ◊ Multiplication by 31 can be replaced by a shift and a subtraction for **better performance**: `31*i == (i << 5) - i`

Hash function criteria: Should be easy and quick to compute.

Compression Map

- ❖ Map an integer (hash code) into a home bucket.
- ❖ The most common method is by **modulo arithmetic**.

`homeBucket = c(hashcode) = hashcode % n`

where n is the **number of buckets** in the hash table.

- ❖ Example: Pairs are (22,a), (33,b), (3,c), (55,d), (79,e). Hash table size is 7.

	(22,a)	(79,e)	(3,c)		(33,b)	(55,d)
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Hashing by Modulo

- ❖ In practice, keys of an application tend to have a specific pattern.
 - ❖ For example, memory address in computer is multiple of 4.
- ❖ The choice of the hash table size n will affect the distribution of home buckets.

Hashing by Modulo

- ◊ Suppose the keys of an application are more likely to be mapped into even integers.
 - ◊ E.g., memory address is always a multiple of 4.
- ◊ When the hash table size n is an **even** number, **even** integers are hashed into **even** home buckets.
 - ◊ E.g., $n = 14$: $20\%14 = 6$, $32\%14 = 4$, $8\%14 = 8$
- ◊ The bias in the keys results in a bias toward the **even** home buckets.
 - ◊ All **odd** buckets are guaranteed to be empty.
 - ◊ The distribution of home buckets is not uniform!

Hashing by Modulo

- ❖ However, when the hash table size n is **odd**, even (or odd) integers may be hashed into both odd and even home buckets.
 - ❖ E.g., $n = 15$: $20\%15 = 5$, $32\%15 = 2$, $8\%15 = 8$
 $15\%15 = 0$, $3\%15 = 3$, $23\%15 = 8$
- ❖ The bias in the keys does not result in a bias toward either the odd or even home buckets.
 - ❖ Better chance of uniform distribution of home buckets.
- ❖ So **do not** use an even hash table size n .

Hashing by Modulo

- ❖ Similar **biased** distribution of home buckets happens in practice when the hash table size n is a multiple of small prime numbers.
- ❖ The effect of each prime divisor p of n **decreases** as p gets **larger**.
- ❖ Ideally, choose the hash table size n as a **large prime number**.