Set and Map ADT
Hash tables

Lecture 13

by Marina Barsky

Modeling dictionaries

sensacionalista *adj.* sensational
sensato *adj.* sensible
sensibilidad *n.* sensibility
sensibilizar *v.* sensitize
sensible *adj.* sensitive
sensiblero *adj.* maudlin
sensor *n.* sensor
sensorial *adj.* sensory
sensual *adj.* sensual
sensualidad *n.* sensuality
sentar *v.t.* sit
sentencia *n.* sentence
sentenciar *v.* sentence
sentencioso *adj.* sententious
sentido *n.* sense
sentimental *adj.* sentimental
sentimiento *n.* feeling
sentir *v.* feel
sentirse atrapado *v.* trammel
seña *n.* sign
señal *n.* signal
señalación *n.* pointing
señalar *v.* pinpoint

sepultura *n.* sepulture
sepulturero *n.* undertaker
sequía *n.* drought
séquito *n.* entourage
ser incapaz *adj.* unable
ser mayor que *v.* outweigh
ser mujeriego *v.* womanize
ser suficiente *v.* suffice
ser *v.* be
serenidad *n.* serenity
sereno *adj.* serene
seres queridos *adj.* nearest
series *n.* series
serio *adj.* serious, earnest
sermón *n.* sermon
sermonear *v.* preach
serpentear *v.* crankle, wriggle
serpentina *n.* streamer
serpentino *adj.* serpentine
serpiente *n.* snake
serrar *v.* saw
serrín *n.* sawdust
servicio *n.* service

- Dictionary is a collection of pairs
- Each pair has a key and the value associated with this key
- The most important functionality of a dictionary is the search of a value by a given key

Let’s assume for the simplicity of the discussion that each key is unique: no duplicate keys are allowed

Abstract Data Type: Map (Dictionary, Associative Array)

Specification

Dictionary is an Abstract Data Type which stores a collection of (key, value) pairs and supports the following operations:

- → **Add (*k*, *e*)** - adds element *e* to the collection and associates it with key *k*
- → **Remove (k)** - removes element with key *k* from the collection
- → **Get (k)** - returns the element associated with key *k*
- → **Contains (k)** - returns *True* if there is an element associated with the key *k*. Returns *False* otherwise

The main functionality is to quickly locate a value given the key

Which data structure to use to implement Dictionary ADT?

Main goal: **locate the element by the key**

- *Linked List, Array* - *N* elements are **unsorted** – search requires **O(*N*)** time
- Sorted array - *N* elements are **sorted – O(log *N*)** binary search

It doesn't seem like we can do much better

Searching in time O(1)

- How about **O(1)**, that is, **constant-time search**?
- We **can** do it **if** we store data in an array organized in a particular way

“Hash is a food, especially meat and potatoes, chopped and mixed together; a confused mess “ (en.wiktionary.org/wiki/hash)

The idea of

Hashing

Example 1: First repeating character

Input: String S of length N

Output: first repeating character (if any) in S

- The obvious $O(N^2)$ solution:

  for each character in order:

      check whether that character is repeated

Example 1: First repeating character

Input: String *S* of length *N*

Output: first repeating character (if any) in *S*

a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111

The number of all possible characters is 256 (ASCII characters)

- We create an array *H* of size 256 and initialize it with all zeros
- For each input character *c* go to the corresponding slot *H*[*c*] and set count at this position to 1
- Since we are using arrays, it takes constant time for reaching any location
- Once we find a character for which counter is already 1 – we know that this is the one which is repeating for the first time

Example 1: First repeating character

Input: String *S* of length *N*

Output: first repeating character (if any) in *S*

cabare

a	97	1
b	98	1
c	99	1
d	100	
e	101	
f	102	
g	103	
h	104	
i	105	
j	106	
k	107	
l	108	
m	109	
n	110	
o	111	

Run-time O(N)

- Because the total number of all possible keys is small (256), we were able to **map each key** (character) **to a single memory location**
- The key tells us precisely where to look in the array!

This method of storing keys in the array is called ***direct addressing:*** **store key *k* in position *k* of the array**

Example 2: First repeating **number**

Input: Array *A* containing *N* **integers**
Output: first repeating number (if any) in *A*

- ➢ This very similarly looking problem is more difficult to solve with *direct addressing*
- ➢ The total number of all possible integers is 2,147,483,647. This is the universe of all possible keys - thus the size of the array
- ➢ What if we have only 25 integers to store? Impractical
- ➢ What if the keys are floats/strings/objects? Impossible
- ➢ For these cases we use a technique of *hashing*: we convert **each key into a number** using a *hash function*

Intuition: hashing inputs

- Suppose we were to come up with a “magic function” that, given a key to search for, would tell us the exact location in the array such that
  - If key is in that location, it’s in the array
  - If key is not in that location, it’s not in the array

- This function would have no other purpose
- If we look at the function’s inputs and outputs, the connection between them won’t “make any sense”
- This function is called a ***hash function*** because it “makes hash” of its inputs

Case study: hashing students

- Suppose we want to store student objects in the array
- For each student we apply the following ***hash function***:

hashCode(Student) =

length (Student.lastName)

This gives us the following values:

- hashCode('Chan')=4
- hashCode('Yam')=3
- hashCode('Li')=2
- hashCode('Jones')=5
- hashCode('Taylor')=6

Array of students: *hash table*

- We place the students into array slots which correspond to the computed hash values:
  - hashCode('Chan')=4
  - hashCode('Yam')=3
  - hashCode('Li')=2
  - hashCode('Jones')=5
  - hashCode('Taylor')=6

0	
1	
2	Li
3	Yam
4	Chan
5	Jones
6	Taylor
7	

Good hash function: length of the last name

- Our hash function is easy to compute
- An array needs to be of size 18 only, since the longest English surname, Featherstonehaugh (Guinness, 1996), is only 17 characters long
- We waste a little bit of space with entries 0,1 of the array, which do not seem to be ever occupied. But the waste is not that bad either

0	
1	
2	Li
3	Yam
4	Chan
5	Jones
6	Taylor
7	

Bad hash function: length of the last name

- ➢ Suppose we have a new student: Smith
  - ○ hashValue('Smith')=5

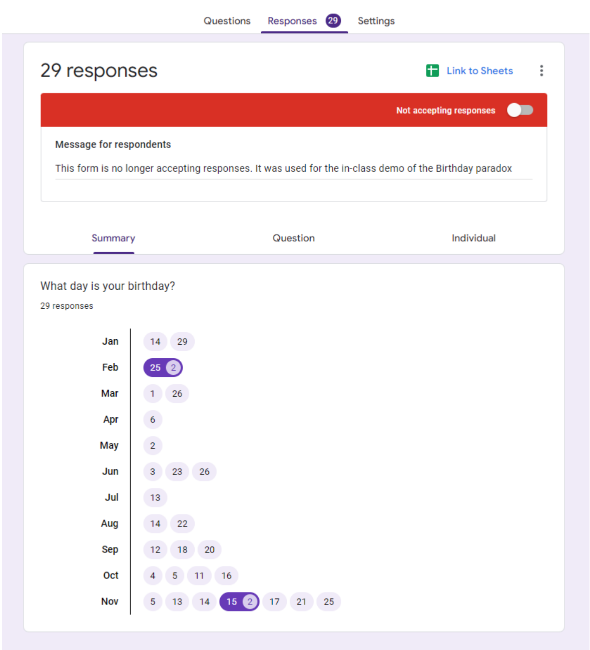
- ➢ When several values are hashed to the same slot in the array, this is called a **collision**

- ➢ Now what?

0	
1	
2	Li
3	Yam
4	Chan
5	Jones
6	Taylor
7	

Looking for a good hash function: day of birth?

- What about the day of birth?
  - We know that this would be 365 (366) possible values, so we can have an array of size 366
  - The birth day of each student is randomly distributed across this range, and this hash function is easy to compute

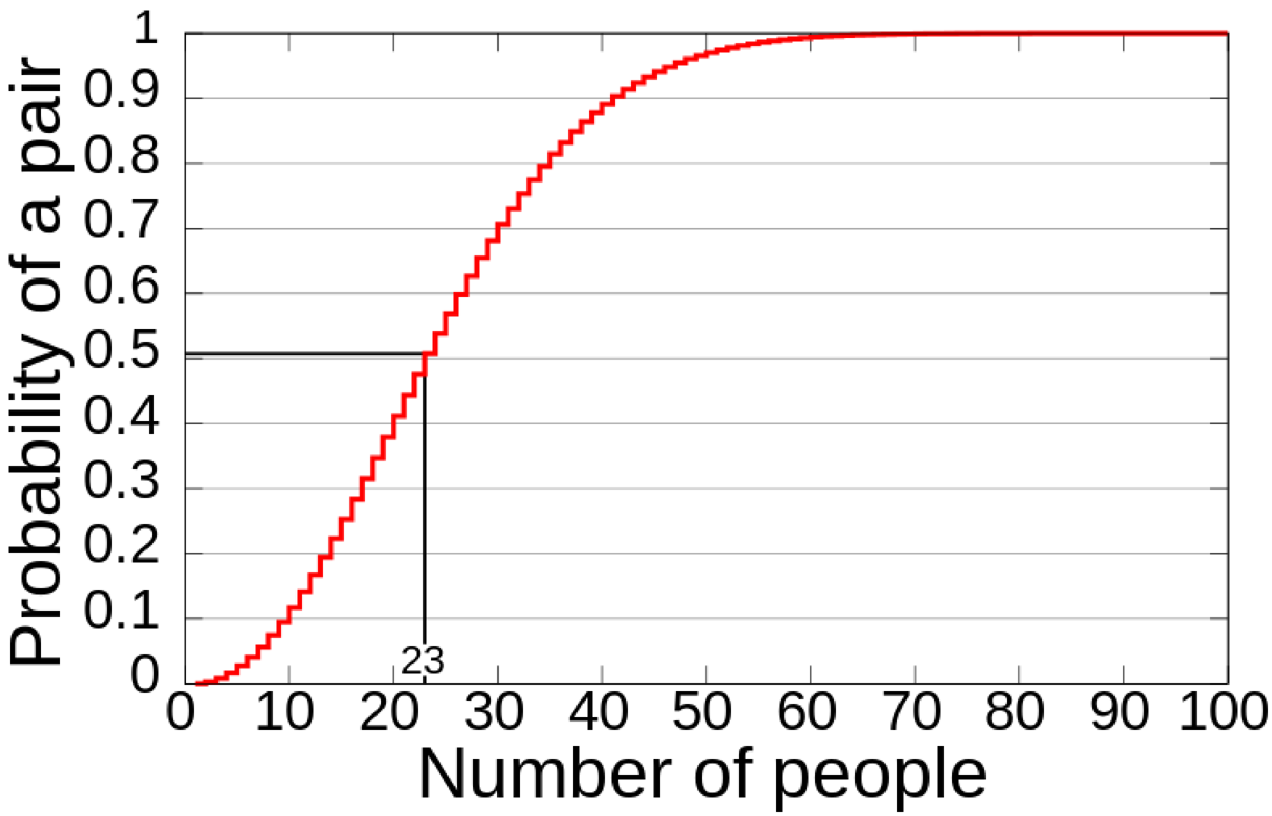


Experiment:
https://forms.gle/BCNWDHvUfRuCpr5p6

Birthday paradox

- For a college with only n=24 students, the probability that any 2 of them were born on the same day is > 0.5
- Let's approximate this probability:
  - The probability of any two people not having the same birthday is:

    p =364/365
  - The number of possible student pairs is $\binom{n}{2} = n(n-1)/2 = 276$
  - The probability for n students of not having birthday on the same date is $p^{n(n-1)/2}$. For 24 students this gives: $(364/365)^{276} \approx 0.47$.
  - Then the probability of finding a pair of students colliding on their birthday is 1.00 - 0.47 = 0.53 !
- This is called *a* *birthday paradox*



http://commons.wikimedia.org/wiki/File:Birthday_Paradox.svg

In search for a perfect hash function

A perfect *hash function* is a function that:

1. When applied to an Object, returns a number
2. When applied to *equal* Objects, returns the *same* number for each
3. When applied to *unequal* Objects returns *different* numbers for each, preventing collisions.
4. The numbers returned by hash function are *evenly* distributed between the range of the positions in the array
5. We also require for our hash function to be *efficiently* computable

non-random inputs → random numbers?

In search for a perfect hash function

- How to come up with this perfect hashing function?
- In general – there is no such magic function
  - In a few specific cases, where all the possible values are known in advance, it is possible to define a perfect hash function. For example hashing objects by their SSN numbers. But this will require an array to be of size 10^9
- It seems that **collisions are essentially unavoidable**. That means that we cannot guarantee that the hash of a key will bring us into the exact position where this key is located

- What is the next best thing?
  - A perfect hash function would have told us exactly where to look
  - However, the best we can do is a function that tells us **in what area of an array to *start* looking**!

Back to students:
Hashing names by summing up their character values

- It seems like a good idea to map each student surname into a number by adding up the ranks (or ASCII codes) of letters in this surname.

$$\text{hashCode (S)} = \sum_{i=0}^{len(S)} rank(S[i])$$

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

What a great hash function!

$$\text{hashCode (S)} = \sum_{i=0}^{len(S)} rank(S[i])$$

- hashCode('Chan')=3+8+1+14=26
- hashCode('Yam')=24+1+13=38
- hashCode('Li')=12+9=21
- hashCode('Jones')=10+15+14+5+18=62
- hashCode('Taylor')=19+1+24+12+15+17=88

- hashCode('Smith')=18+13+9+19+8=67

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

Still a lot of collisions!

$$\text{hashCode (S)} = \sum_{i=0}^{len(S)} rank(S[i])$$

- Not only hashCode('Yam')=hashCode('May')
- But hashCode('Chan')= hashCode('Lam') !

The function takes into account the value of each character in the string, but **not the order of characters**

Polynomial hashing scheme

➢The summation is not a good choice for sequences of elements **where the order has meaning**

➢Alternative: choose *A*≠1, and use a hash function for string *S* of length *N*:

$$hashCode(S) = \sum_{i=0}^{N-1} S[i] \cdot A^{N-1-i} =$$

$$S[0] \cdot A^{N-1} + S[1] \cdot A^{N-1-1} + S[2] \cdot A^{N-1-2} + \cdots + S[N-1] \cdot A^{N-1-(N-1)}$$

➢This is a polynomial of degree *N* for *A*, and the elements (characters) of the String are the coefficients of this polynomial

a	1
b	2
c	3
d	4
e	5
f	6
g	7
h	8
i	9
j	10
k	11
l	12
m	13
n	14
o	15
p	16
r	17
s	18
t	19
u	20
v	21
w	22
x	23
y	24
z	25

Example: polynomial hashing

$$hashCode(S) = \sum_{i=0}^{N-1} S[i] \cdot A^{N-1-i} =$$

$$S[0] \cdot A^{N-1} + S[1] \cdot A^{N-1-1} + S[2] \cdot A^{N-1-2} + \cdots + S[N-1] \cdot A^{N-1-(N-1)}$$

S_1 = 'Yam'

S_2 = 'May'

A = 31

$hashCode(S_1) = 24*31^2 + 1*31^1 + 13*31^0 = 23108$

$hashCode(S_2) = 13*31^2 + 1*31^1 + 24*31^0 = 12548$

➢Instead of using the *summation* of all character values, the polynomial hash function introduces interactions between different bits of successive characters that will provoke or spread randomness of the result

How to compute polynomial of degree *N* in time O(*N*)

Horner's method:

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$
$$= a_0 + x\bigg(a_1 + x\Big(a_2 + x\big(a_3 + \cdots + x(a_{n-1} + x\, a_n)\cdots\big)\Big)\bigg)$$

Let x=31, and a_0 … a_n represent n+1 characters of string S:

```
public int hashCode(){
    int hash=0;
    for (int i=0; i< length(); i++)
        hash=hash*31+S[i];
    return hash;
}
```

Java String *hashCode()*

- Polynomial hashing is quite good: for different strings it returns mostly different values which are well spread over the range of all possible integers
- This hash function is also very efficient, since we need only *n = length()* steps to compute it

Implemented inside the String class

```
public int hashCode(){
    int hash=0;
    for (int i=0; i< length(); i++)
        hash=hash*31+S[i];
    return hash;
}
```

That is ~how *hashCode()* is implemented inside Java *String* class

```
hash*31 = hash<<5 - hash
```

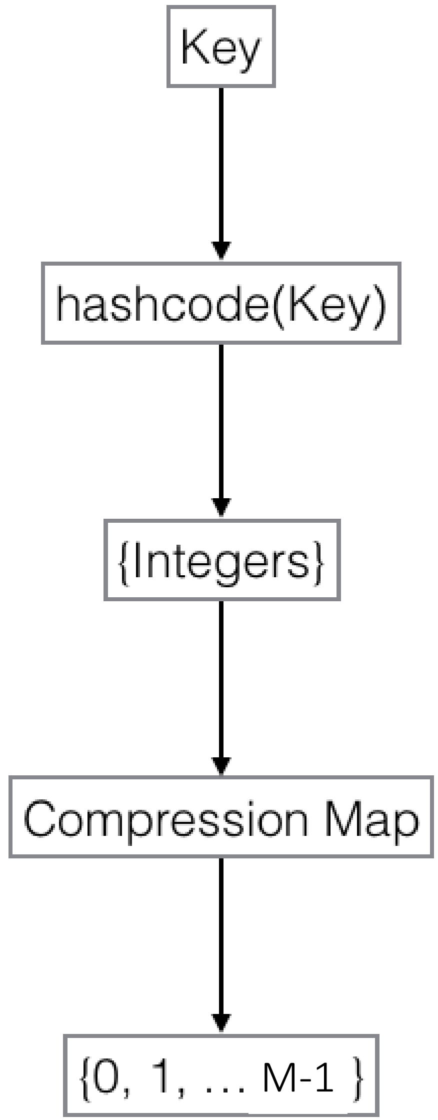
Reducing the range of *hashCode* to the capacity of the array

- The output of a good hash function is a number ~randomly distributed over the range of **all** integers.
  - But we need to store our objects in the array of size ***M***
- Step 2: **compression mapping**
  - Converting integers in range ~ [0,400000000] to integers in range [0, *M*]
  - The simplest way to do it: |*hashCode*| MOD *M*
  - In practice, the MAD (Multiply Add and Divide) method:

    |(*A**hashCode+*B*) MOD *M*|

    The best results when *A*, *B* and *M* are primes

Full hashing



Hashing Students to 7 slots

➔ Applying polynomial hashing:

hashCode('Taylor')=-880692189
hashCode('Yam')=119397
hashCode('Li')=345
hashCode('Lee')=107020
hashCode('Lam')=106904
hashCode('Roy')=113116

➔ Applying compression mapping:
|(11*hashCode+13) MOD 7|

arrayIndex('Taylor')=6
arrayIndex('Yam')=2
arrayIndex('Li')=4
arrayIndex('Lee')=5
arrayIndex('Lam')=3
arrayIndex('Roy')=1

0	
1	Roy
2	Yam
3	Lam
4	Li
5	Lee
6	Taylor

No more collisions?

- Does Java *hashCode* **always** produce different hash code for different strings?

  The answer is **NO**.

  If you run the code in the box, you will find out that

  - The words *Aa* and *BB* have the same *hashCode*
  - Words *variants* and *gelato* hash to the same value
  - …

```
public static void main(String [] args) {
    String [] words=new String[6];
    words[0]="Aa";
    words[1]="BB";
    words[2]="variants";
    words[3]="gelato";
    words[4]="misused";
    words[5]="horsemints";

    for(int i=0;i<6;i++)   {
      System.out.print("Hash code of "+words[i]+": ");
      System.out.println(words[i].hashCode());
    }
}
```

- We have to be prepared to deal with **collisions**, since they **are unavoidable**