Maps and hash tables

Data Structures and Algorithms for Computational Linguistics III
(ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2021/22

version: 81e53c2 @2021-12-15

Hashing and hash-based data structure

- A *hash function* is a one-way function that takes a variable-length object, and turns it into a fixed-length bit string
- Most common applications of hash functions is the *map* (or *associative array*, or *dictionary*, or *symbol table*) data structure
- Maps are array-like data structures ($O(1)$ access/update) but can be indexed using arbitrary objects (e.g., strings)
- Hashing has many other applications
  - Database indexing
  - Cache management
  - Efficient duplicate detection
  - File signatures: verification against corrupt/tempered files
  - Password storage
  - Electronic signatures
  - As part of many other cryptographic algorithms/applications

Maps and sets

- Two common data structures that use hashing is sets and maps (Python `dict`)
- *set* abstract data type is based on the sets in mathematics: unordered collection without duplicates
- *map* abstract data type is a collection that allows indexing with almost any data type (Python `dict`s require immutable data types)
- Basic operations include

Sets:

- Check whether an object is in the set (`x in s`)
- Add an element to a set (`s.add(x)`)
- Remove an element from a set (`s.remove(x)`)

Maps:

- Retrieve the value of a key (`d[key]`)
- Associate a key with value (`d[key] = val`)
- Remove key–value pair (`del d[key]`)

Implementing sets and maps

	Check/retrieve	Add	Remove
Sorted array:	$O(\log n)$	$O(n)$	$O(n)$
Unsorted array:	$O(n)$	$O(1)$	$O(n)$
Skip list:	$O(\log n)$	$O(\log n)$	$O(\log n)$
Balanced search trees:	$O(\log n)$	$O(\log n)$	$O(\log n)$
Hash tables:	$O(1)$	$O(1)$	$O(1)$

A trivial array implementation

store each element i at index i (assuming non-negative integer keys for now)

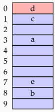

```
d[3] = 'a'
d[8] = 'b'
d[1] = 'c'
```

+ All operations are $O(1)$
- Cannot handle non-integer, non-negative keys
- Wastes a lot of memory if key values are spread across a wide range

Hash functions

- A hash function $h()$ maps a key to an integer index between 0 and m (size of the array)
- We use $h(k)$ as an index to an array (of size m)
- If we map two different key values to the same integer, a *collision* occurs
- The main challenge with implementing hash maps is to avoid and handle the collisions
- We can think of a hash function in two parts:
  - map any object (variable bit string) to an integer (e.g., 32 or 64 bit)
  - compress the range of integers to map size (m)

Compressing the hash codes

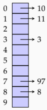



```
h(k)= lamda k: k % 10
d[3] = 'a'
d[8] = 'b'
d[1] = 'c'
d[10] = 'd'
d[97] = 'e'
d[40] = 'f' – collision
```

- An easy way to map any integer to range $[0, m]$ is to use modulo $m + 1$
- Good hash functions minimize collisions, but collisions occur
- Collisions has to be handled by a map data structure. Two common approaches:
  - Separate chaining
  - Open addressing

Separate chaining

or closed addressing

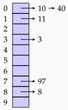



```
add(3)
add(8)
add(97)
add(11)
add(10)
```

- Each array element keeps a pointer to a secondary container (typically a list)
- When a collision occurs, add the item to the list,

Separate chaining

or closed addressing

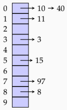



```
add(3)
add(8)
add(97)
add(11)
add(10)
add(40)
```

- Each array element keeps a pointer to a secondary container (typically a list)
- When a collision occurs, add the item to the list,

Separate chaining

or closed addressing

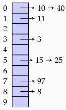



```
add(3)
add(8)
add(97)
add(11)
add(10)
add(40)
add(15)
```

- Each array element keeps a pointer to a secondary container (typically a list)
- When a collision occurs, add the item to the list,

Separate chaining

or closed addressing

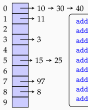



```
add(3)
add(8)
add(97)
add(11)
add(10)
add(40)
add(15)
add(25)
```

- Each array element keeps a pointer to a secondary container (typically a list)
- When a collision occurs, add the item to the list,

Separate chaining

or closed addressing




```
add(3)
add(8)
add(97)
add(11)
add(10)
add(40)
add(15)
add(25)
add(30)
```

- Each array element keeps a pointer to a secondary container (typically a list)
- When a collision occurs, add the item to the list,
- Why not just add to the head of the list?

Complexity of separate chaining

is it really O(1)?

0	→ 10 → 30 → 40
1	→ 11
2	
3	→ 3
4	
5	→ 15 → 25
6	
7	→ 97
8	→ 8
9	

- All operations require locating the element first
- Cost of locating an element include hashing (constant) + search in secondary data structure
- This means worst-case complexity is $O(n)$
- With a good hash function, the probability of a collisions is n/m: average bucket size is $O(n/m) = O(1)$ (if $m > n$)
- Expected complexity for all operations is $O(1)$

Load factor for separate chaining

- Load factor of a hash map is

$$\text{load factor} = \frac{\text{number of entries}}{\text{number of indices}}$$

- Low load factor means
  - better run time (fewer collisions)
  - more memory usage
- When load factor is over a threshold, the map is extended (needs rehash)
- Recommendation vary, but a load factor around 0.75 is considered optimal

Rehashing

Before: lf=9/10

0	1	2	3	4	5	6	7	8	9
10 → 30 → 40	11		3		15 → 25		97	8	

After: lf=9/13

0	1	2	3	4	5	6	7	8	9	10	11	12
	40	15	3	30		97		8		10	11	25

Open addressing (linear probing)

adding/accessing items

0	1	2	3	4	5	6	7	8	9
10	1		3				97	8	

```
add(3)
add(8)
add(97)
add(11)
add(10)
```

- During insertion, if there is a collision, look for the next empty slot, and insert
- During lookup, probe until there is an empty slot

Open addressing (linear probing)

adding/accessing items

0	1	2	3	4	5	6	7	8	9
10	1	40	3				97	8	

$h(40)$

```
add(3)
add(8)
add(97)
add(11)
add(10)
add(40)
```

- During insertion, if there is a collision, look for the next empty slot, and insert
- During lookup, probe until there is an empty slot

Open addressing (linear probing)

adding/accessing items

0	1	2	3	4	5	6	7	8	9
10	1	40	3	22			97	8	

$h(22)$

```
add(3)
add(8)
add(97)
add(11)
add(10)
add(40)
add(22)
```

- During insertion, if there is a collision, look for the next empty slot, and insert
- During lookup, probe until there is an empty slot

Open addressing (linear probing)

deleting items

0	1	2	3	4	5	6	7	8	9
10	1	40	3	22			97	8	

```
remove(40)
```

- We can locate an element as usual, and delete it

Open addressing (linear probing)

deleting items

0	1	2	3	4	5	6	7	8	9
10	1		3	22			97	8	

$h(40)$

```
remove(40)
```

- We can locate an element as usual, and delete it

Open addressing (linear probing)

deleting items

0	1	2	3	4	5	6	7	8	9
10	1		3	22			97	8	

$h(22)$

```
remove(40)
contains(22)
```

- We can locate an element as usual, and delete it
- However, this breaks probing: now $h(22)$ will point to an empty slot
- Rearranging the remaining items is complex & costly

Open addressing (linear probing)

deleting items

0	1	2	3	4	5	6	7	8	9
10	1	X	3	22			97	8	

$h(22)$

```
remove(40)
contains(22)
```

- We can locate an element as usual, and delete it
- However, this breaks probing: now $h(22)$ will point to an empty slot
- Rearranging the remaining items is complex & costly
- We insert a special value,
  - During lookup, treat it as full
  - During insertion, treat it as empty

Quadratic probing

- Linear probing tends to create clusters of items, especially if load factor is high (> 0.5)
- Quadratic probing provides some improvements
- Probe $(h(k) + i^2) \mod m$ for $i = 0, 1, \ldots$ until an empty slot is found
- If m is prime, and load factor is less than 0.5, quadratic probing is guaranteed to find an empty slot
- Although better than linear porobing, quadratic probing creates its own kind of clustering

Double hashing

- Similar to quadratic probing, probe non-linearly
- Instead of probing the next item, probe $(h(k) + i \times h'(k)) \mod m$ for $i = 0, 1, \ldots$ where $h'(k)$ another hash function
- A common choice is $h'(k) = q - (k \mod q)$ for a prime number $q < m$

Using a pseudo random number generator

- This method probes $(h(k) + i \times r_i) \mod m$ for $i = 0, 1, \ldots$ where r_i is the i^{th} number generated by a pseudo random number generator
- Pseudo random number generators generate numbers that are close to uniform. However given the same seed, the sequence is deterministic
- This approach is the most common choice for modern programming languages/environments
- This also avoids problems with inputs that intentionally generate hash collisions

Aside: hash DoS attacks

- A denial-of-service (DoS) attack aims to break or slow down an Internet site/service
- A particular attack (in 2003, but also 2011) made use of hash table implementation of popular programming languages
- Input to a web-based program is passed as key–value pairs, which are typically stored in a dictionary
- If one intentionally posts an input with a large number of colliding keys,
  - the hash table implementation needs to chain long sequences (separate chaining) or probe a large number of times (open addressing)
  - and eventually re-hash
- This increases expected to $O(1)$ time to worst-case complexity

Hash functions

and their properties

- A hash function *must be* consistent: if `a == b`, `h(a) == h(b)`
- A hash function should minimize collisions: values for h should be uniformly distributed
- A hash function should be fast to compute (...or maybe not – if you are using it for passwords)

Hash codes

- Earlier we suggested dividing the hash function into two
  - A has code that maps a variable-size object to an integer
  - A compression method that squeezes the integer value to hash table size
- A hash code avoid collisions: colliding hash codes are unavoidably mapped the same table address
- A naive approach is to truncate (e.g., take the most or least significant bits), or pad with an arbitrary pattern (if object is shorter than the hash code)
- This approach creates many collisions in real-world usage

Hash codes

xor or add

- A simple approach is based on
  - Bitwise add each k-bit segment of the memory representation of the object, ignoring the overflow: $h(x) = \sum_i x_i$
  - Similarly, one can use XOR instead of addition
- These methods meet the hash code requirement: if `a == b`, then `h(a) == h(b)`
- However, in practice, they create many collisions because of their associativity
  - `abc`, `bca` and `cba` all get the same hash code

Polynomial hash codes

- Polynomial hash codes are calculated using

$$h(x) = \sum_{i}^{n} x_i a^{n-i-1} = x_0 a^{n-1} + x_1 a^{n-2} + \ldots + x_{n-1}$$

- The important aspect is that now the function will produce different values with sequences with the same items in a different order
- The exact form is motivated by quick computation if rewritten as

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \ldots))$$

Cyclic-shift hash codes

- Instead of multiplying with powers of a constant, cyclic-shift hashing shifts some bits from one end of the other at each step in running sum
- Since bitwise operations are simple, this is a fast way of obtaining a non-associative valid has code

```
1010011001110100
1010111000110100
```

```python
def cyclic_shift(s):
  mask = 0xffff
  h = 0
  for ch in s:
    h = (h << 5 & mask) | (h >> 11)
    h += ord(ch)
  return h
```


A short divergence: cryptographic hash functions

- Hash functions has an important role in cryptography
- In cryptography, it is important to have hash functions for which it is difficult to find two keys with the same hash value
- There are a wide range of well-known hash functions (which are also available in most programming environments)
  - MD5
  - SHA-1
  - RIPEMD-160
  - Whirlpool
  - SHA-2
  - SHA-3
  - BLAKE2
  - BLAKE3
- These functions are designed for applications like digital fingerprinting, password storage,
- Computationally inefficient for use in data structures

Summary

- Hash functions are useful for implementing map ADT efficiently
- Hash functions have a wide range of other applications
- The main issue in implementing a hash function is avoiding collisions, and handling them efficiently when they occur
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 10)

Next:

- Algorithms on strings: pattern matching, edit distance, tries
- Reading: Goodrich, Tamassia, and Goldwasser (2013, chapter 13),Jurafsky and Martin (2009, section 3.11, or 2.5 in online draft)

Acknowledgments, credits, references

Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN: 9781118476734.

Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second edition. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.

blank

blank

blank

blank