Hash Table

CS143: lecture 11

Byron Zhong, July 11

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Two $O(n \log n)$ algorithms: Tree, Heap
- There are a lot more sorting algorithms...

Sorting

Recap

- Three $O(n^2)$ algorithms: Selec ubble
- Two $O(n \log n)$ algorithms: Tr
- There are a lot more sorting a

Name	Best	Average	Worst
Quicksort	$n \log n$	$n \log n$	n^2
Merge sort	$n \log n$	$n \log n$	$n \log n$
In-place merge sort	—	—	$n \log^2 n$
Introsort	$n \log n$	$n \log n$	$n \log n$
Heapsort	$n \log n$	$n \log n$	$n \log n$
Insertion sort	n	n^2	n^2
Block sort	n	$n \log n$	$n \log n$
Timsort	n	$n \log n$	$n \log n$
Selection sort	n^2	n^2	n^2
Cubesort	n	$n \log n$	$n \log n$
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$
Bubble sort	n	n^2	n^2
Exchange sort	n^2	n^2	n^2
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)
Cycle sort	n^2	n^2	n^2
Library sort	$n \log n$	$n \log n$	n^2
Patience sorting	n	$n \log n$	$n \log n$
Smoothsort	n	$n \log n$	$n \log n$
Strand sort	n	n^2	n^2
Tournament sort	$n \log n$	$n \log n$	$n \log n$
Cocktail shaker	n	n^2	n^2

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Two $O(n \log n)$ algorithms: Tree, Heap
- There are a lot more sorting algorithms...
- ... we have time for one more.

Counting Sort

- Count the occurrences of every number
- Output each number as many times as it occurs in the original list

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9

Counts

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

Counting Sort

Output

Counts

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

Counting Sort

Output

2	2							

Counts

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

Counting Sort

Output

2	2	4	4					

Counts

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

Counting Sort

Output

2	2	4	4	6				

Counts

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

Counting Sort

Output

2	2	4	4	6	8			

Counts

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

Counting Sort

Output

2	2	4	4	6	8	9	9	9

Counts

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

Counting Sort

Complexity

1. Find the range of values: $O(n)$
2. Initialize array: $O(n)$
3. Scan the list to count: $O(n)$
4. Scan the counts to output: $O(n)$

$$O(n) + O(n) + O(n) + O(n) = O(n)$$

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:
  - Counts: $O(\text{Range})$ -- if the input is sparse, this can be a lot
  - Output: $O(n)$
- This is almost a Map!
  - Key: Integer
  - Value: Counts

Counting Sort

Limitations?

- Can we make this work with any value?
  - Sure, instead of having an array of integers, we can have an array of whatever values
- Can we make this work with any key?
  - Turn any key into an integer
  - Make the range of the integer reasonable

Hashing

Turning any value into an integer

- A *hash function* maps a key to an integer *deterministically*:
  - I.e. the same key is always turned into the same integer
  - Hash functions should run in $O(1)$ time
- There are good/bad choices for hash functions

Hashing

Example: 2-letter word dictionary

- Map 2-letter words to definitions:
  - Key: 2-letter words (string)
  - Value: definitions (string)

```
ah: used to express delight, relief, regret, or contempt
as: to the same degree or amount
at: used as a function word to indicate presence or occurrence in, on, or near
do: to bring to pass
go: to move on a course
ha: used especially to express surprise, joy, or triumph
he: that male one who is neither speaker nor hearer
hi: used especially as a greeting
...
```

- What hash function could we use to map keys to ints?

Hashing

Example: 2-letter word dictionary

- How many 2-letter words are there?
  - 26 * 26 = 676
- How to map words into [0, 676)?
  - Idea: map a-z: 0-25
  - then, first letter's number * 26 + second letter's number

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

  - $hash(\alpha\beta) = 26\alpha + \beta$
  - $hash(\text{go}) = 26 \cdot 6 + 14 = 170$

Hashing

Example: 2-letter word dictionary

- Example!

Hashing

Problem

- Can we extend this function to work for all words?
- https://en.wikipedia.org/wiki/Longest_word_in_English

Word	Letters
Longest chemical	189,819
Longest word in Merriam-Webster	45
Supercalifragilisticexpialidocious	34
Longest word in Shakespeare’s works	27

- $26^{27} = 160059109085386090080713531498405298176$

Hashing

Problem

- $26^{27} = 160059109085386090080713531498405298176$
- Too big for an array!
- Also, English has ~700,000 words; we only need a tiny fraction of these.
- Solution: Compress

Hashing

Compression

- Generally, hash functions do not care about its output range.
- We use a *compression function* to put the integer in the reasonable range [0,size)
- Common choice: modulus
  - a % b calculates the remainder of a divided by b
  - a % b always returns an int in the range [0, b)

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	
2	
3	
4	
5	
6	
7	7
8	
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	
2	
3	
4	
5	
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35

0	
1	41
2	
3	
4	
5	
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35
-

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35
- What if we try to insert 75?

0	
1	41
2	
3	
4	
5	35
6	
7	7
8	18
9	

Hashing

Compression example

- Keys: integer
- Table size: 10
- hash: itself
- compress: hash % 10
- insert: 7, 18, 41, 35
- What if we try to insert 75?

0		
1	41	
2		
3		
4		
5	35	**75**
6		
7	7	
8	18	
9		

Hashing

Collision

- Two different keys sometimes end up in the same slot
  - This is called a collision
- Collision has to happen if we have smaller array than the range of hash function
  - Hash function could produce the same integer for two different keys
  - Compression merges different hashes together
- All tables need to handle collision

Hashing

Handling Collision

1. Avoid collisions when possible:
    1. Pick a good hash function (e.g. `strlen` is a terrible hash function)
    2. Pick a good table size
2. When they arise (inevitably):
    1. Have a way to put collisions in a table.

Hashing

Picking a good hash function

- Minimize collision:
  - What is the worst possible hash function?
  - hash(k) = 1
  - What is the best possible hash function?
  - Every input maps to a distinct output, $f(x) = f(y) \implies x = y$
  - This is called *perfect* hashing. The two-letter hash function is a perfect hash function.

Hashing

Picking a good hash function (Example)

- If we want to hash UChicago students:
  - Use their birthdays
    - Month (Jan, Feb, Mar, ...)?
    - Age (0, 1, 2, ..., 100)?
    - Day of month (1, 2, 3, ..., 31)?
  - Use their first name
  - Use their last name
  - Use their student ID

Hashing

Picking a good hash function

- A good hash function should be:
  - fast
  - collision with (extremely) low probability
  - spreads out the keys
- CS284: Cryptography