Hash Tables and Hashing

CS 214, Fall 2019

Dictionary data structures we have seen, with lookup times and a special case

- (Balanced) binary search tree — $\mathcal{O}(\log n)$
- Sorted array — $\mathcal{O}(\log n)$
- List of associations — $\mathcal{O}(n)$

Dictionary data structures we have seen, with lookup times and a special case

- (Balanced) binary search tree — $\mathcal{O}(\log n)$
- Sorted array — $\mathcal{O}(\log n)$
- List of associations — $\mathcal{O}(n)$
- An array using keys (0, 1, …, $k-1$) as indices — $\mathcal{O}(1)$

The last of these is sometimes called "direct addressing"

A direct addressing example

Suppose we want to map digits to their names in English:

```
let digits = ['zero', 'one', 'two', 'three', 'four',
              'five', 'six', 'seven', 'eight', 'nine']

def get_digit_name(name: int?) -> str?:
    digits[name]
```

Non-direct addressing example: phone book

A phone book is a dictionary where the keys are names and the values are phone numbers

How can we use names (strings) as keys?

Non-direct addressing example: phone book

A phone book is a dictionary where the keys are names and the values are phone numbers

How can we use names (strings) as keys?

Let's map strings to small integer keys by using the value of the first character

The first-character hash

(bucket)	name	phone
(0)	`"Alice"`	555–1212
(1)	⊘	⊘
(2)	`"Carol"`	555–1214
(3)	⊘	⊘
	⋮	
(24)	`"Yves"`	555–1215
(25)	⊘	⊘

The first-character hash

(bucket)	name	phone
(0)	"Alice"	555–1212
(1)	⊘	⊘
(2)	"Carol"	555–1214
(3)	⊘	⊘
	⋮	
(24)	"Yves"	555–1215
(25)	⊘	⊘

What happens when we want to add Charles to the phonebook?

Hash collision!

The function that maps names to numbers is called a *hash function*:

$$h(\texttt{"Alice"}) = 0$$
$$h(\texttt{"Carol"}) = 2$$

Hash collision!

The function that maps names to numbers is called a *hash function*:

$$h(\texttt{"Alice"}) = 0$$
$$h(\texttt{"Carol"}) = 2$$

When the hash function gives the same value for two keys, that's called a *hash collision*:

$$h(\texttt{"Charles"}) = 2$$

Hash collision!

The function that maps names to numbers is called a *hash function*:

$$h(\texttt{"Alice"}) = 0$$
$$h(\texttt{"Carol"}) = 2$$

When the hash function gives the same value for two keys, that's called a *hash collision*:

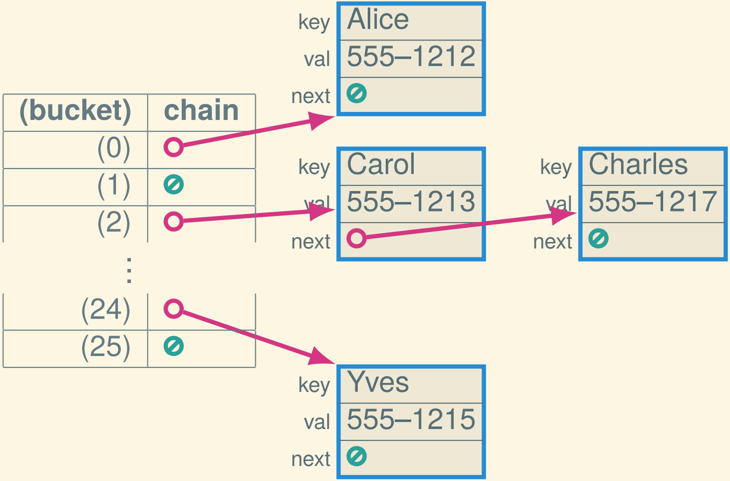
$$h(\texttt{"Charles"}) = 2$$

How do we resolve it?

Two solutions to hash collision

1. Store a linked list in each bucket (*separate chaining*)
2. Use the next free bucket instead (*open addressing*)

Separate chaining hash table



Open addressing hash table

(bucket)	name	phone
(0)	"Alice"	555–1212
(1)	⊘	⊘
(2)	"Carol"	555–1214
(3)	"Charles"	555–1217
(4)	⊘	⊘
⋮		
(24)	"Yves"	555–1215
(25)	⊘	⊘

What happens as the table fills up

- Separate chaining: the length of the chains is $\mathcal{O}(n)$
- Open addressing: the length of the scan is $\mathcal{O}(n)$

Thus, it's important to have enough buckets

Our hash function sucks

Using the first letter limits us to 26 buckets, but for a big phonebook we need more buckets

Our hash function sucks

Using the first letter limits us to 26 buckets, but for a big phonebook we need more buckets

Here's a better hash function:

Input: A string *str* and number of buckets *buckets*
Output: A hash code between 0 and $\textit{buckets} - 1$

$\textit{hash} \leftarrow 1$;

for each character *c* in *str* do
| $\textit{hash} \leftarrow 31 \times \textit{hash} + c$
end

return *hash* % *buckets*

What makes a good hash function?

Hash functions are big topic—what you need to know:

- deterministic (not random)
- uniform (not clustery)

Load

For good performance, we can't let the table get too full

One way to think of this is the *load factor*:

$$\text{load factor} = \frac{n}{k}$$

- n: number of entries
- k: number of buckets

Load

For good performance, we can't let the table get too full

One way to think of this is the *load factor*:

$$\text{load factor} = \frac{n}{k}$$

- n: number of entries
- k: number of buckets

For separate chaining, we should keep the load factor < 2

For open addressing, we should keep the load factor < 0.75

Resizing

When the load factor gets too high, we need to grow the table

- Requires rehashing everything!
- Doubles in size (like dynamic array)

Next time: Big-O notation