CS 3110

Amortized Analysis

Nate Foster

Spring 2020

Today's music: "Money, Money, Money" by ABBA

CLICKER QUESTION 1

Review

Current topic: Efficiency

- Big Oh
- Hash tables (and mutability)

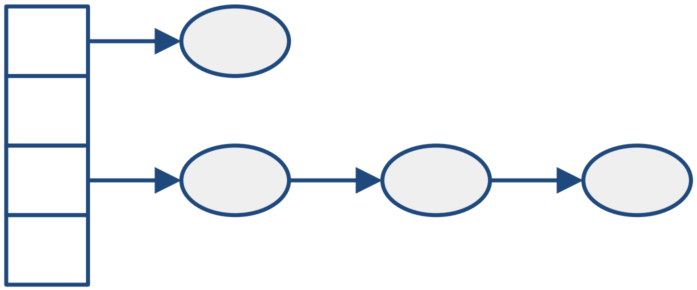
Today:

- Amortized analysis

REVIEW OF HASH TABLES

Hash table: chaining

```
type ('k, 'v) t = {
  mutable buckets
    : ('k * 'v) list array
}
```



Implementation of operations

- Insert (k, v):
  - Hash k to find bucket b
  - Search through b to delete any previous binding of k (to maintain RI)
  - Mutate bucket to add new binding of k
- Find k:
  - Hash k to find bucket b
  - Search through b to find binding of k
- Remove k:
  - Hash k to find bucket b
  - Search through b to delete any binding of k

…every operation requires search through bucket

…efficiency depends on bucket length

Load factor

Load factor = average bucket length = α

(# bindings in hash table) / (# buckets in array)

- # bindings not under implementer's control
- # buckets is
- When load factor gets above some constant, make array bigger
  - Which makes load factor smaller
  - Then redistribute keys across bigger array

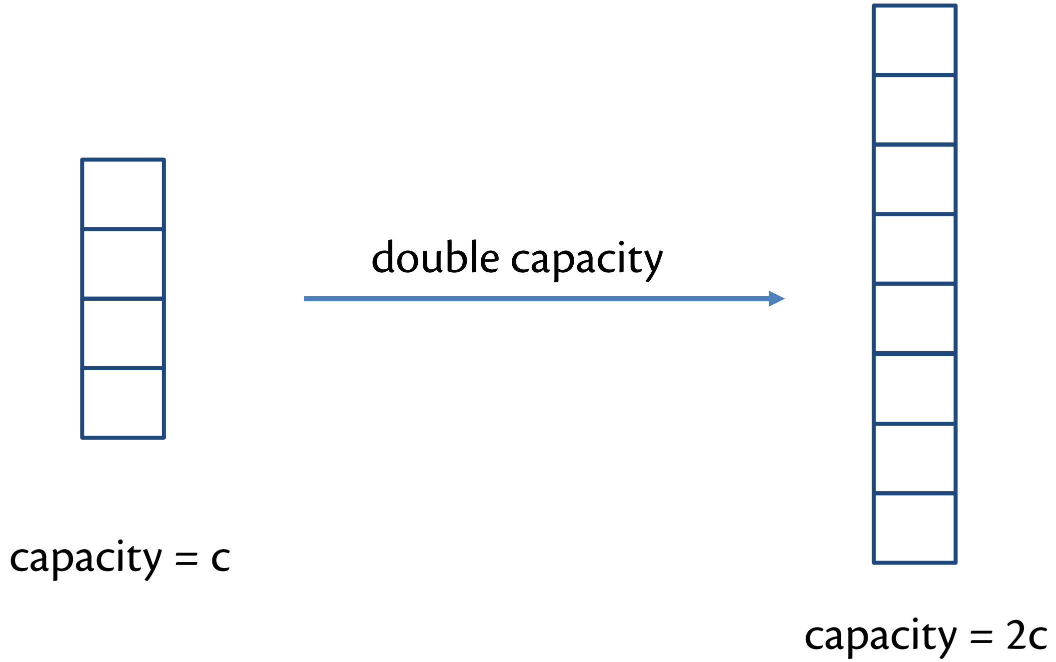
CLICKER QUESTION 2

Rehashing

- If load factor ≥ 2.0 then:
  - double array size
  - rehash elements into new buckets
  - thus bringing load factor back to around 1.0
- Both OCaml **`Hashtbl`** and **`java.util.HashMap`** do this
- Efficiency:
  - find, and remove: expected O(1)
  - insert: O(n), because it can require rehashing all elements
  - but we wanted O(1)…

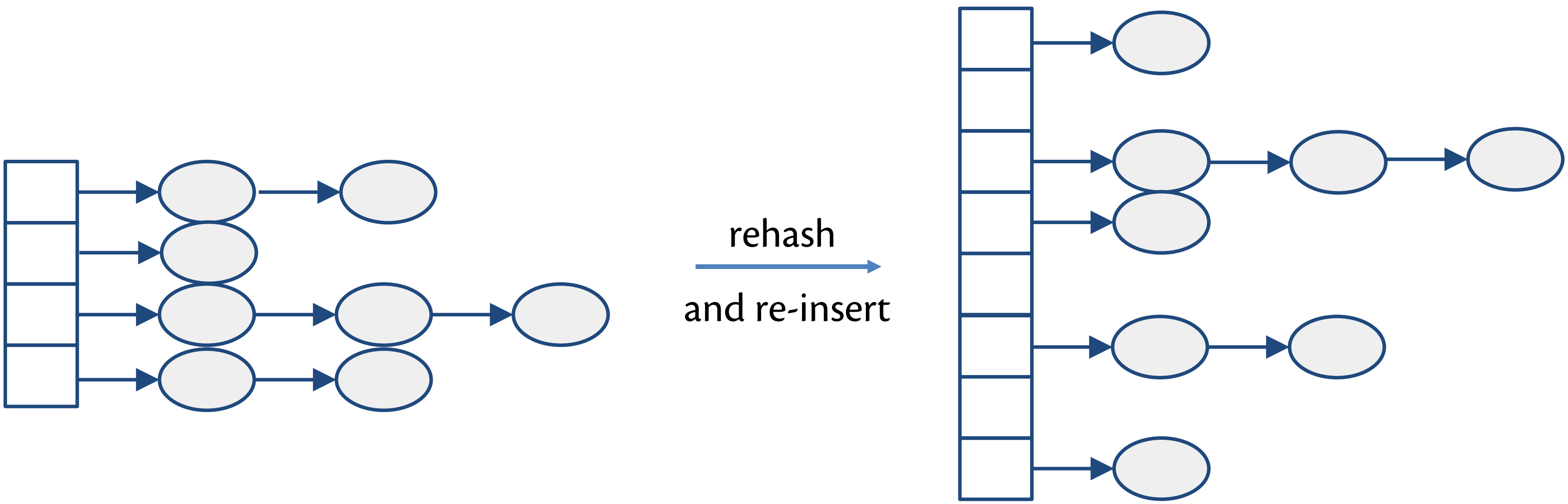
AMORTIZED ANALYSIS

Hash table resize



Cost = O(n) = O(2c)

Hash table resize



Expected cost = 2n O(1)= O(n)

Total cost to resize

Expected cost = O(n) + O(n) = O(n)

Suppose the hidden constant is r

- r = x + y + z
- x is cost to allocate
- y is cost to hash
- z is cost to insert

Let's call that $r

Saving money

on insert

save $r

on resize

spend $r·n

Bank account balance

Capacity	Bindings	Load factor α	Balance
16	16	1	$0

Bank account balance

Capacity	Bindings	Load factor α	Balance
16	16	1	\$0
Insert 16 bindings			
16	32	2	\$16r

Bank account balance

<table>
<tr><th>Capacity</th><th>Bindings</th><th>Load factor α</th><th>Balance</th></tr>
<tr><td>16</td><td>16</td><td>1</td><td>$0</td></tr>
<tr><td colspan="4" align="center">Insert 16 bindings</td></tr>
<tr><td>16</td><td>32</td><td>2</td><td>$16r</td></tr>
<tr><td colspan="4" align="center">Resize and rehash</td></tr>
<tr><td>32</td><td>32</td><td>1</td><td>-$16r</td></tr>
</table>

Let’s double the amount we save: $2rn

Bank account balance

Capacity	Bindings	Load factor α	Balance
16	16	1	\$0
Insert 16 bindings			
16	32	2	\$32r

Bank account balance

Capacity	Bindings	Load factor α	Balance
16	16	1	\$0
	Insert 16 bindings		
16	32	2	\$32r
	Resize and rehash		
32	32	1	\$0

Bank account balance

Capacity	Bindings	Load factor α	Balance
16	16	1	\$0
Insert 16 bindings			
16	32	2	\$32r
Resize and rehash			
32	32	1	\$0
Insert 32 bindings			
32	64	2	\$64r
Resize and rehash			
64	64	1	\$0

Budgeting

Mon

Tue

Wed

Thur

Fri

Budgeting

- Key idea is to analyze worst-case efficiency of
  - sequence of operations
  - not individual operations
- Rare expensive operations paid for by common inexpensive operations

Hash table efficiency

- find, and remove: expected O(1)
- insert: expected O(1), because rehashing can be paid for with amortization

TWO-LIST QUEUES

Two-list queues [lec 7]

abstract:

1	2	3	4	5	6	7	8

concrete:

front:

1	2	3

(enqueued since front last emptied)

back:

8	7	6	5	4

(recently enqueued)

Two-list queues: AF+RI

- Rep type:
  - front of queue: list, stored in order
  - back of queue: list, stored in reverse order
- RI: if front is empty then back is empty

Two-list queues: efficiency

- Peek: head of front O(1)
- Enqueue: cons onto back O(1)
  - But if completely empty, cons onto front instead to maintain RI O(1)
- Dequeue: tail of front O(1)
  - If front becomes empty, reverse back and make it the front to maintain RI O(n)

Amortized analysis

on enqueue (back)

save $1

on reverse

spend $n

Bank account balance

Front length	Back length	Balance
0	0	\$0
Enqueue 1 element		
1	0	\$0
Enqueue 9 elements		
1	9	\$9
Dequeue 1 element		
0	9	\$9
Reverse back and make it front		
9	0	\$0
Dequeue 9 elements		
0	0	\$0

KEY IDEAS OF AMORTIZED ANALYSIS

Amortized analysis

- *Amortize:* put aside money at intervals for gradual payment of debt [Webster's 1964]
- In efficiency analysis:
  - Pay extra "money" for some operations as a credit
  - Use that credit to pay higher cost of some later operations
  - a.k.a. *banker's method* and *accounting method*
- Invented by Sleator and Tarjan (1985)

Robert Tarjan

b. 1948

Turing Award Winner (1986) with Prof. John Hopcroft

For fundamental achievements in the design and analysis of algorithms and data structures.

Cornell CS faculty 1972-1973

Upcoming events

- [Friday] Project Teams Due
- [Next week] CS 3110 goes virtual
- [Next Monday] R0 due
- [Next Wednesday] MS0 due

This is money.

THIS IS 3110