CS3230 Lecture 6

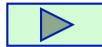
"Amortized Analysis"



- **□** Lecture Topics and Readings
 - Amortized Analysis

[CLRS]-C17

♦ Introduction



- ◆ Binary Increment
- **◆** Dynamic Tables

It pays to analyze in detail.

Better analysis may come from
a different point of view

(CS3230 Outline) Page 1

CS3230 Mid-Term Quiz

CS3230 Mid-Term Quiz

Sat, 07-Mar, 10:00am – 11:30am

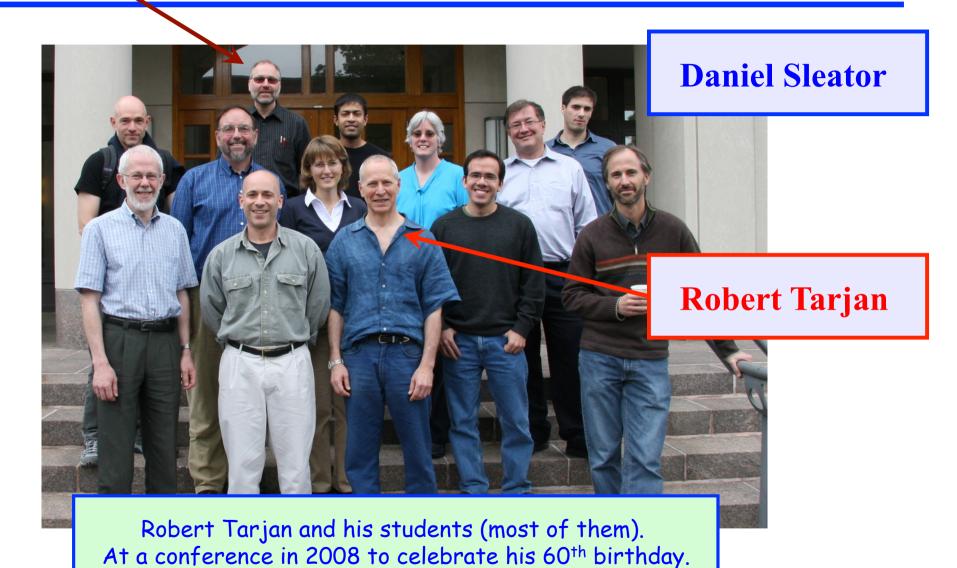
Venue: LT19 & SR3

(detail room assignment coming later)

IMPORTANT:

If you have VALID, GOOD reasons for a clash, email instructors ASAP giving details of your valid good reasons.

Sleator and Tarjan



(CS3230 Outline) Page 3

Bob Tarjan's landmark paper...

□...one of many important papers...

SIAM. J. on Algebraic and Discrete Methods, 6(2), 306-318. (13 pages)

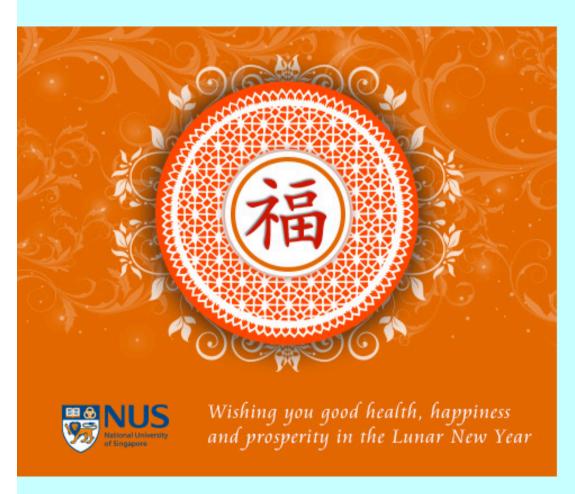
Amortized Computational Complexity

Robert Endre Tarjan

A powerful technique in the complexity analysis of data structures is amortization, or averaging over time. Amortized running time is a realistic but robust complexity measure for which we can obtain surprisingly tight upper and lower bounds on a variety of algorithms. By following the principle of designing algorithms whose amortized complexity is low, we obtain "self-adjusting" data structures that are simple, flexible and efficient. This paper surveys recent work by several researchers on amortized complexity.

Copyright © 1985 © Society for Industrial and Applied Mathematics

CNY Greetings from CS3230 Staff



Dear All in C53230,

Wishing you a Happy, Healthy and Rewarding Chinese New Year of the Goat, 2015 (you can still enjoy it, if you are not Chinese).

Have a good 1-week break. Come back from the break refreshed and ready for more C53230 fun.

From: Hon Wai (and the CS3230 Staff)

As we celebrate NUS'110th year of founding in 2015, we invite you to click on the logo for more information on NUS110 celebrations.



Thank you.





CS3230 Lecture 6 –

"Amortized Analysis"



- **□** Lecture Topics and Readings
 - * Amortized Analysis

[CLRS]-C17

♦ Introduction



- ◆ Binary Increment
- **◆** Dynamic Tables

It pays to analyze in detail.

Better analysis may come from
a different point of view

Data Structure Operations

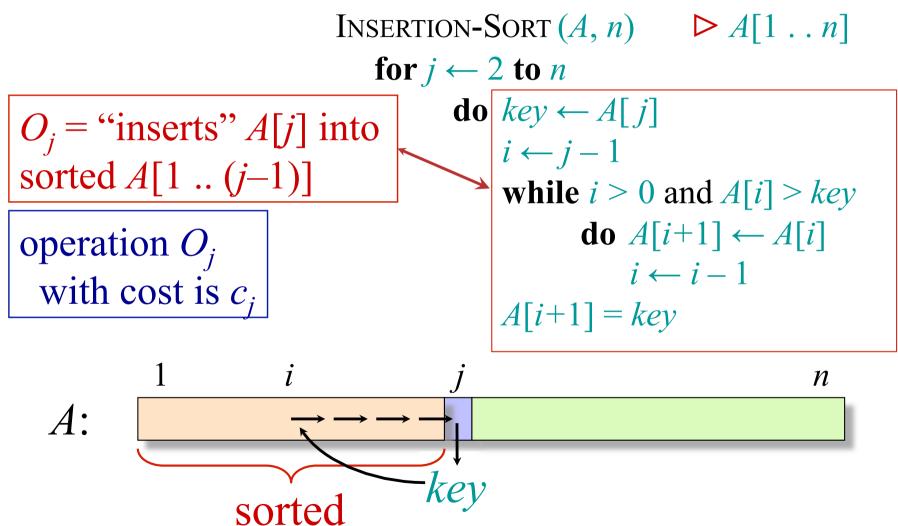
Given any sequence of n operations $O_1, O_2, ..., O_n$ on a data structure D where

 $D_k = \text{d.s.}$ after applying O_k with cost c_k on D_{k-1} .

Want to compute
$$C(n)/n$$
, where $C(n)/n = (c_1 + c_2 + c_3 + ... + c_n)/n = \frac{1}{n} \sum_{k=1}^{n} c_k$



Consider: Insertion sort



Insertion Sort: (n-1) "insert" ops

Insertion Sort does (n-1) "insert" operations

Let $O_i =$ "insert" A[j] into sorted A[1 ... (j-1)]

Let c_j = the cost of operation O_j

Consider a worst-case "insert" sequence:

In the worst-case, $c_i = (j-1)$

Total (worst-case) time for Insertion Sort:

$$C(n) = (c_1 + c_2 + c_3 + \dots + c_n)$$

= $(0 + 1 + 2 + \dots + (n-1)) \approx 0.5(n^2)$

Average cost of "insert" op. (worst-case):

$$C(n)/n \approx 0.5n$$

This worst-case bound is **TIGHT**. Eg: when A[1..n] reversed-sorted.

Insert into Binary Search Tree

Do *n* Insert ops on an *initially empty* BST j^{th} Insert operation inserts into BST (of size (j-1)) Let d_j be the cost of j^{th} INSERT operation

Consider a worst-case Insert sequence In the worst-case, $d_j = (j-1)$

Total (worst-case) *time* for *n* INSERT's:

$$C(n) = (d_1 + d_2 + d_3 + \dots + d_n)$$

= $(0 + 1 + 2 + \dots + (n-1)) \approx 0.5(n^2)$

Average cost of Insert op. (worst-case):

$$C(n)/n \approx 0.5n$$

This worst-case bound is **TIGHT**.

Eg: elements inserted into BST in increasing order.

Hon Wai Leong, NUS

Copyright © 2005-13



Motivation for Amortized C

For Insertion Sort, Insert into BST,

- * worst-case upper-bound is TIGHT
- ❖ Achieved when array is reversed-sorted and when elements inserted in increasing order.

But, sometimes,

- ❖ simple worst-case upper-bound is **NOT TIGHT**
- ❖ May over-estimate the actual running time

Example: Consider this sequence

Consider a sequence of operations with cost

$$c_k = \begin{cases} k & \text{if } (k-1) \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Example: Consider this sequence

Consider a sequence of operations with cost

$$c_k = \begin{cases} k & \text{if } (k-1) \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Question:

How many of you keep a "Clean and Tidy Room" *all the time*?

Let CTR_k = "Clean-and-Tidy-Room" op on the kth day.

Yes, we know that CTR_k is "expensive" operation!

Answer:

Most of us a lazy most of the time.

But once in a while, must do a lot of cleanup



Su Yuen Chin

19 hrs · Singapore · 🤼

This just happened. Me wake up, all blurry-eyed, chill in bed... open phone, see whatsapp. Message from mum? Oh ok. Open message

"Hi dear, r u home? I'm on way to Singapore and will be arriving at 5pm"

pinches face and stares at message again OMG!! THIS IS REAL!!!

instantly transforms into high speed cleaning robot!

Like · Comment · Share

Eugene Wee and 23 others like this.

On Sat, 13-9-2014 this appeared on my FB wall.



Su Yuen Chin

19 hrs · Singapore · 1

This just happened. Me wake up, all blurry-eyed, chill in bed... open phone, see whatsapp. Message from mum? Oh ok. Open message

"Hi dear, r u home? I'm on way to Singapore and will be arriving at 5pm"

pinches face and stares at message again OMG!! THIS IS REAL!!!

instantly transforms into high speed cleaning robot!

Like · Comment · Share

Eugene Wee and 23 others like this.

On Sat, 13-9-2014 this appeared on my FB wall.



Henry Bai Shuyong Why must clean when your mother come?

19 hrs · Like · 1 1



Su Yuen Chin Henry Bai Shuyong Ya hor, my place so clean already right! (4) unfortunately still fails my mum's standards (2)

19 hrs · Edited · Like · ₺ 2



Hon Wai Leong SuYuen;

This is so funny, yet so common.

Can I use this for class? When I teach amortized complexity analysis...

--honwai

19 hrs · Like · ₼ 4

Me think...
Amortized
Complexity
in real life!!

Analysis) Page 12



Su Yuen Chin Hon Wai Leong Sure thing! How is this related to that though? o_O?

21 hrs · Like



Hon Wai Leong Long answer:

In optimizing for amortized time efficiency, we get lazy with data structure updates when we don't have to properly organize things. So, we are lazy (but fast) most of the time.

But when we need to, we just work much harder and we will have to get EVERYTHING or MOST things back in order. So, once in a while (like parent visit, in-laws visit), we work like crazy hard. HAHA, like you now. Sorry to pour salt on wound, wicked hehehe.

But overall, worst-case performance is still better than maintaining order each and every time.

So, back to your question:

Like how many student/people keep their rooms/house/office clean and neat ALL the time?

No, most people don't.

We are lazy most times, and we work hard and get things organized when we have to.

So, most students practice amortized complexity optimization!

--honwai

20 hrs · Like · 13

All students practise amortized complexity optimization

nalysis) Page 13

Example: Consider this sequence

Consider a sequence of operations with cost

$$c_k = \begin{cases} k & \text{if } (k-1) \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

When does worst-case happen?

Consider a sequence of operations with cost

$$c_k = \begin{cases} k & \text{if } (k-1) \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$
Most operations only cost 1

Worst-case: (Upper bound W_k) $c_k \le k = W_k$ for all k

$$c_k \le k = W_k$$
 for all k

Worst case is *expensive* (but not often) most operations are cheap (cost=1)

If we bound each operation by W_k

Consider a sequence of operations with cost

k	1	2	3	4	5	6	7	8	9	10
c_k	1	2	3	1	5	1	1	1	9	1
W_k	1	2	3	4	5	6	7	8	9	1 10

	1	1	1	1	1	1	17 17	1	1	1	
k c_k W_k	11	12	13	14	15	16	17	18	19	20	

If we use worst-case bound

Consider a sequence of operations with cost

$$c_k = \begin{cases} k & \text{if } (k-1) \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Worst-case: (Upper bound W_k)

$$c_k \le k = W_k$$
 for all k

Total time for *n* operations

$$C(n) \le (1+2+3+\ldots+n) \approx 0.5(n^2)$$

Average cost per operation is

$$C(n)/n \approx 0.5n$$

Using Amortized Analysis...

Consider a sequence of operations with cost

$$c_k = \begin{cases} 1 + 2^j & \text{if } (k - 1) = 2^j \\ 1 & \text{otherwise} \end{cases}$$

We will show *later* that

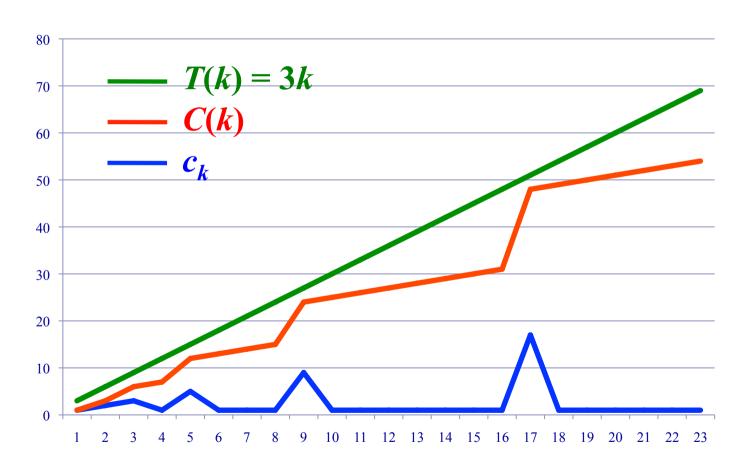
We will show *later* th
$$C(n) \leq 3n \quad \text{for all } n$$
Amortized cost per or

Amortized cost per operation = C(n)/n = 3

Amortized Time is Constant $\Theta(1)$

Graphical demonstration

☐ For now, I only give a graphical demo



Summary: For this example

Worst-case bound over-estimates

Amortized complexity analysis (shown later)

 \bullet Give average cost per operation ≤ 3

Sometimes, the worst-case time of an operation does not accurately capture the worst-case time of a *sequence of operation*.

Different Complexity Analysis

☐ Worst-Case Time:

* Worst-case of any operation O_k over all possible instances. (Worst-case *actual time* of c_k for any k).

□ Average-case Time:

❖ Average-case of each operation over all instances taken from certain probability distribution.

☐ Amortized Time:

Look at worst-case sequence of operations, and ask what is the *average cost per operation* for this worst-case sequence.

Motivation!

- Amortized analysis *guarantees* the average cost per operation in the *worst-case* sequence of operation.
- □ Even though a *bad single operation* may be *expensive*, the *average cost* may be small.
- ☐ Even in a worst-case sequence of operations, the *worst-case performance* does not happen very often

A First Example: Binary Counter



Incrementing a Binary Counter

 \square k-bit Binary Counter: A[0..k-1]

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^{i}$$

```
INCREMENT(A)
```

- 1. $i \leftarrow 0$
- 2. while i < length[A] and A[i] = 1
- 3. $\operatorname{do} A[i] \leftarrow 0$ > reset a bit 4. $i \leftarrow i + 1$ 5. $\operatorname{if} i < \operatorname{length}[A]$

- **then** $A[i] \leftarrow 1$ \triangleright set a bit

k-bit Binary Counter

Ctr	A[4]	A[3]	A[2]	A [1]	A[0]	c_k	C(k)
0	0	0	0	0	0		
1	0	0	0	0	1	1	1
2	0	0	0	1	0	2	3
3	0	0	0	1	1	1	4
4	0	0	1	0	0	3	7
5	0	0	1	0	1	1	8
6	0	0	1	1	0	2	10
7	0	0	1	1	1	1	11
8	0	1	0	0	0	4	15
9	0	1	0	0	1	1	16
10	0	1	0	1	0	2	18
11	0	1	0	1	1	1	19
12	0	1	1	0	0	3	22

Let c_k be the # of bits flipped during the k^{th} Increment

And
$$C(k)$$
 be the $(c_1 + c_2 + ... + c_k)$

Worst-case analysis

Consider a sequence of n increments. In **worst-case**, number of bit flipped during increment is k. Thus, worst-case *total time* for n insertions is $C(n) \le n \cdot k = O(n \cdot k)$.

But it is **WRONG** to say that $C(n) = \Theta(n \cdot k)$.

In fact, the worst-case cost for n insertions is only $\Theta(n) \ll \Theta(n \cdot k)$.

Let's see why.

Tighter analysis

Ctr	A[4]	A[3]	A[2]	A [1]	A[0]	c_k	C(k)
0	0	0	0	0	0		0
1	0	0	0	0	1	1	1
2	0	0	0	1	0	2	3
3	0	0	0	1	1	1	4
4	0	0	1	0	0	3	7
5	0	0	1	0	1	1	8
6	0	0	1	1	0	2	10
7	0	0	1	1	1	1	11
8	0	1	0	0	0	4	15
9	0	1	0	0	1	1	<i>16</i>
10	0	1	0	1	0	2	18
11	0	1	0	1	1	1	19
12	0	1	1	0	0	3	22

Total cost of *n* operations

Tighter analysis (continued)

Cost of *n* increments

ements =
$$\sum_{i=1}^{\infty} \left[\frac{1}{2^i} \right]$$

Wow!

Only 2 bits flipped per increment operation (amortized).

$$< n \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n$$

$$=\Theta(n)$$

Thus, the average cost of each increment operation is $\Theta(n)/n = \Theta(1)$.

More precisely, the average cost of each increment operation is $\leq 2n/n = 2$

Look-back: After we know...

Ctr	A[4]	A[3]	A[2]	A [1]	A[0]	c_k	C(k)
0	0	0	0	0	0		0
1	0	0	0	0	1	1	1
2	0	0	0	1	0	2	3
3	0	0	0	1	1 🔻	1	4
4	0	0	1	0	0	3	7
5	0	0	1	0	1 <	1	8
6	0	0	1	1	0	2	10
7	0	0	1	1	1 <	1	11
8	0	1	0	0	0	4	15
9	0	1	0	0_	1	1	16
10	0	1	0	1	0	2	18
11	0	1	0	1	1	1	19
12	0	1	1	0	0	3	22

We know:

2 bits flipped per increment

Savings: (for

future operations)

Expensive ops:

(use savings)

A-problem: (for those interested)

Did you see the pattern?

$$Q(1) = (1)$$

$$Q(2) = (1 \ 2 \ 1) = (1) \ 2 \ (1)$$

$$Q(3) = (1 \ 2 \ 1 \ 3 \ 1 \ 2 \ 1) = (1 \ 2 \ 1) \ 3 \ (1 \ 2 \ 1)$$

$$Q(4) = Q(3) 4 Q(3)$$

Solve for Q(k)



Amortized analysis

An *amortized analysis* is any strategy for analyzing a sequence of operations to show that the average cost per operation is small, even though a single operation within the sequence might be expensive.

• An amortized analysis *guarantees* the *average performance* of each operation in the *worst case*.

Types of amortized analyses

Three common amortization arguments:

- the *aggregate* method,
- the *accounting* method,
- the *potential* method.

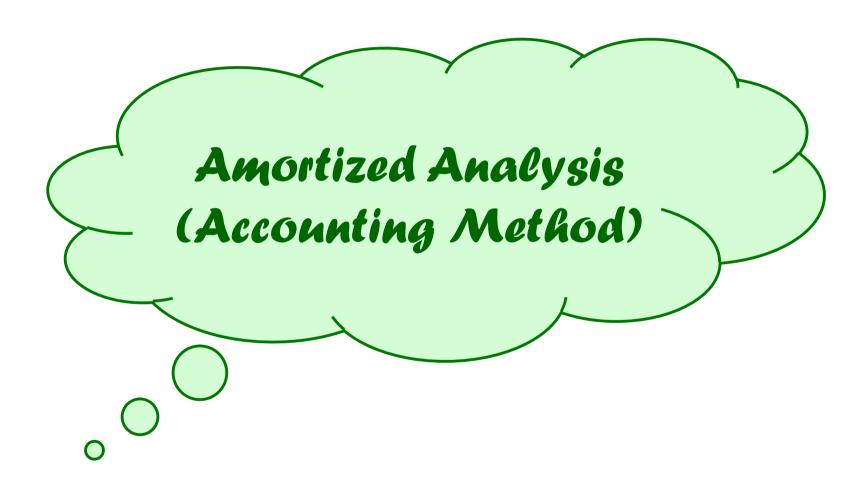
Will not cover potential method in Spring 2015 Sem

We've just seen an aggregate analysis.

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific *amortized cost* to be allocated to each operation.

(CS3230: Amortized Analysis) Page 33

A First Example: Binary Counter



Accounting method

- Charge *i* th operation a fictitious *amortized cost* \hat{c}_i , where \$1 pays for 1 unit of work (*i.e.*, time).
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the *bank* for use by subsequent operations.
- The bank balance must not go negative! We must ensure that

$$\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

for all n.

• Then, the total amortized costs provide an upper bound on the total true costs.

Accounting analysis of INCREMENT

Charge an amortized cost of \$2 every time a bit is set from 0 to 1

- \$1 pays for the actual bit setting.
- \$1 is stored for later re-setting (from 1 to 0).

At any point, every 1 bit in the counter has \$1 on it... that pays for resetting it. (reset is "free")

Example:

(CS3230: Amortized Analysis) Page 36

Cost = \$2

0 0 0 1 1 1 1 1 0

Incrementing a Binary Counter

```
INCREMENT(A)

1. i \leftarrow 0

2. while i < length[A] and A[i] = 1

3. do A[i] \leftarrow 0 > reset a bit

4. i \leftarrow i + 1

5. if i < length[A]

6. then A[i] \leftarrow 1 > set a bit
```

- **□** When Incrementing,
 - **❖** Amortized cost for line 3 = \$0 // use savings
 - **Amortized cost for line** 6 = \$2 // **cost for setting bit**
- \square Amortized cost for Increment(A) = \$2
- \square Amortized cost for n Increment(A) = \$2n = O(n)

(CS3230: Amortized Analysis) Page 37

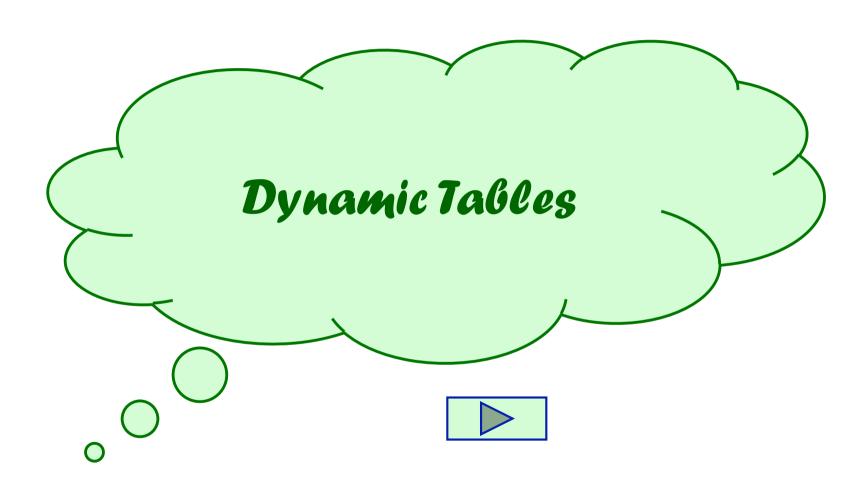
Accounting analysis (continued)

Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

i	1	2	3	4	5	6	7	8	9	10
#1 's	1	1	2	1	2	2	3	1	2	2
c_i	1	2	1	3	1	2	1	4	1	2
\hat{c}_i	2	2	2	2	2	2	2	2	2	2
i $\#1$'s c_i \hat{c}_i $bank_i$	1	1	2	1	2	2	3	1	2	2

(CS3230: Amortized Analysis) Page 38

Second Example:

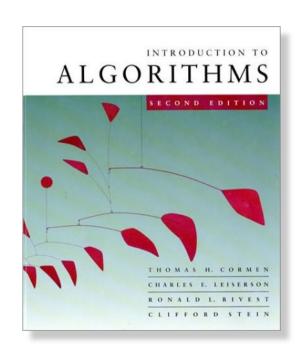


Thank you.



(CS3230: Amortized Analysis) Page 40

Introduction to Algorithms 6.046J/18.401J



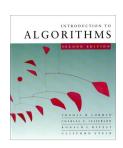
LECTURE 6

Amortized Analysis

- Dynamic tables
- Aggregate method
- Accounting method
- Potential method

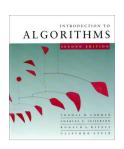
Prof. Charles E. Leiserson

With additional slides added by LeongHW (2013-2014)



Example 2: Dynamic Table





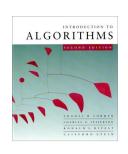
How large should a hash table be?

Goal: Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

Problem: What if we don't know the proper size in advance?

Solution: Dynamic tables.

IDEA: Whenever the table overflows, "grow" it by allocating (via **malloc** or **new**) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.



Dynamic Arrays

Used in: Java ArrayList and C++ Vector

Check its API for Java ArrayList:

http://www.docjar.com/html/api/java/util/ArrayList.java.html

especially API and code for methods add, ensureCapacityInternal, grow

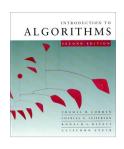
We illustrate with this

Check its API for C++ Vector:

http://www.cplusplus.com/reference/vector/vector/

especially API and code for methods push back, insert, resize

Slide added by LeongHW (Nov 2013)



ArrayList API and impl (1)

Code for add

```
404
          /**
           * Appends the specified element to the end of this list.
405
406
           * @param e element to be appended to this list
407
           * @return <tt>true</tt> (as specified by {@link Collection#add})
408
409
           */
          public boolean add(E e) {
410
              ensureCapacityInternal(size + 1); // Increments modCount!!
411
412
              elementData[size++] = e;
413
              return true;
414
```

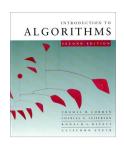


ArrayList API and impl (2)

Code for ensureCapacityInternal

```
171
          /**
172
           * Increases the capacity of this <tt>ArrayList</tt> instance, if
           * necessary, to ensure that it can hold at least the number of elements
173
174
           * specified by the minimum capacity argument.
175
                      minCapacity the desired minimum capacity
176
           * @param
177
           * /
178
          public void ensureCapacity(int minCapacity) {
              if (minCapacity > 0)
179
180
                  ensureCapacityInternal(minCapacity);
181
          }
182
183
          private void ensureCapacityInternal(int minCapacity) {
184
              modCount++;
185
              // overflow-conscious code
186
              if (minCapacity - elementData.length > 0)
                  grow(minCapacity);
187
188
```

Slide added by LeongHW (Nov 2013)



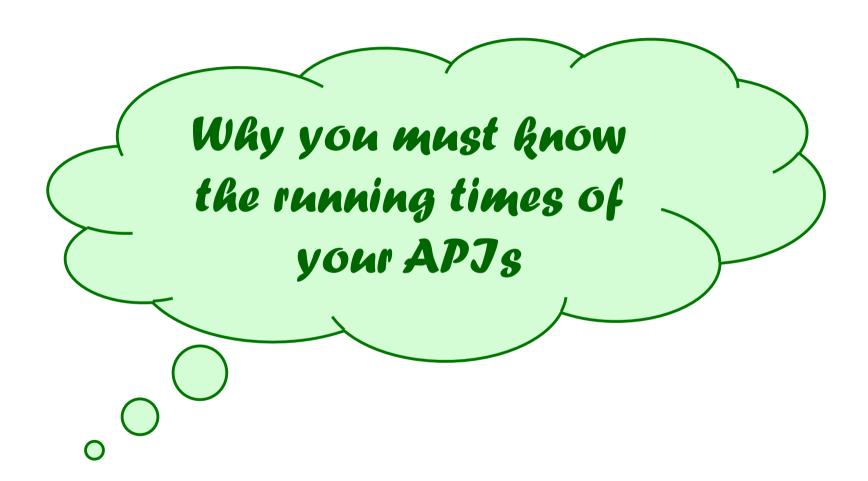
ArrayList API and impl (3)

Code for grow

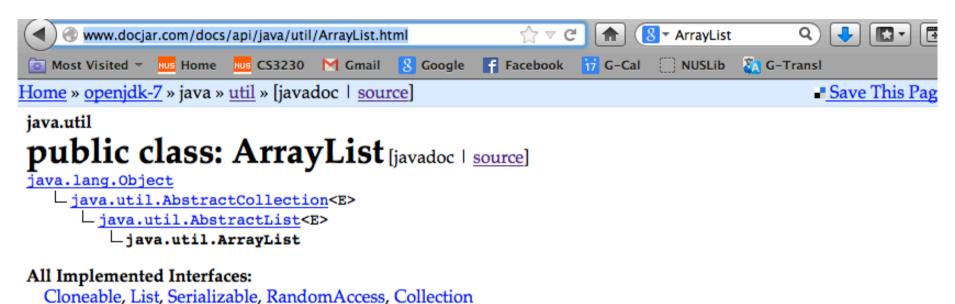
```
198
          /**
199
           * Increases the capacity to ensure that it can hold at least the
           * number of elements specified by the minimum capacity argument.
200
201
           * @param minCapacity the desired minimum capacity
202
           */
203
204
          private void grow(int minCapacity) {
              // overflow-conscious code
205
206
              int oldCapacity = elementData.length;
207
              int newCapacity = oldCapacity + (oldCapacity >> 1);
              if (newCapacity - minCapacity < 0)
208
209
                  newCapacity = minCapacity;
210
              if (newCapacity - MAX ARRAY SIZE > 0)
                  newCapacity = hugeCapacity(minCapacity);
211
212
              // minCapacity is usually close to size, so this is a win:
213
              elementData = Arrays.copyOf(elementData, newCapacity);
214
```

Slide added by LeongHW (Nov 2013)

Side-Track:



Important to know your APIs well



Direct Known Subclasses:

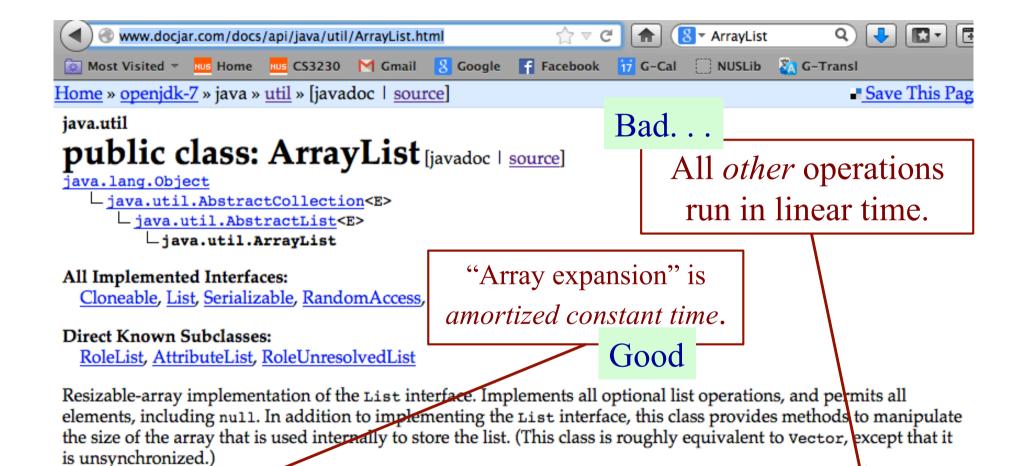
RoleList, AttributeList, RoleUnresolvedList

Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to vector, except that it is unsynchronized.)

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

l Analysis) Page 9

Important to know your APIs well



The size, is Empty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

(CS3230: Amortized Analysis) Page 10

Wrong use of API (unwittingly)

- ☐ In Spr-2014 PA1-ABC, you used **ArrayList**
 - To get the rank (after sorting),
 you call the API: indexOf()
 - To check for a student,
 you call the API: contains()

What's Good:

Easy and fast to do!

Done in no time at all.

What's FATAL:

TLE for task C too slow (linear search)!

Solution: Code a $O(\lg n)$ binary search API

(CS3230: Amortized Analysis) Page 11

Moral of the Story

"Theorem":

Just because the API is available in the library (& it's so easy to call it – just one loc) does NOT mean you should always call it.

Especially, if you want your code to scale.

It is important to know your APIs well. Remember *Analysis*! Analyze and know the running times of the APIs before you use them.

Careful with Online Help

Not all online help gives running time.

Official doc for Java ArrayList:

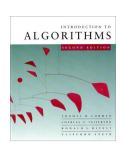
* http://www.docjar.com/docs/api/java/util/ArrayList.html

Good! Gives running times of all APIs.

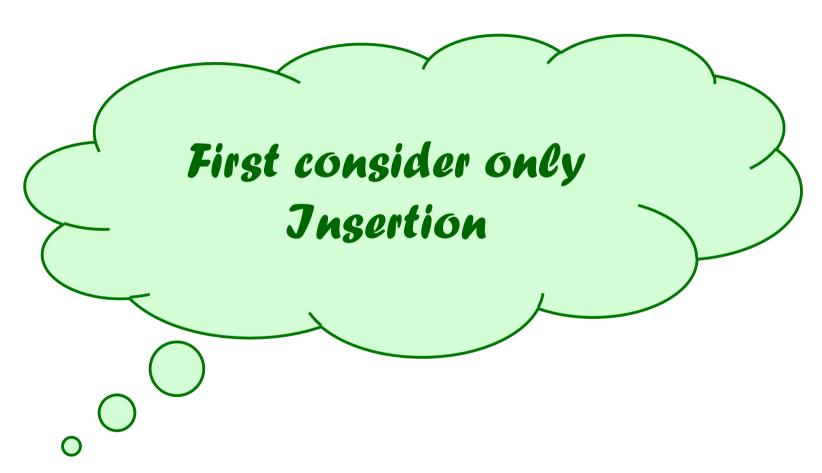
Some help resources on Java ArrayList:

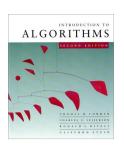
- * http://developer.android.com/reference/java/util/ArrayList.html
- * http://www.tutorialspoint.com/java/java_arraylist_class.htm

Don't show running times of API (not their focus) Good for novice and beginners;

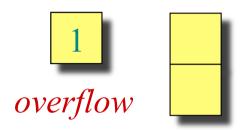


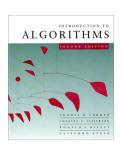
Dynamic Tables



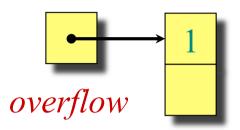


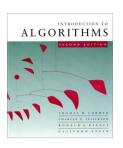
- 1. Insert
- 2. Insert





- 1. Insert
- 2. Insert

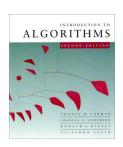




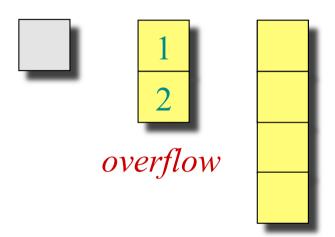
- 1. Insert
- 2. Insert

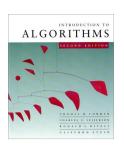


1 2

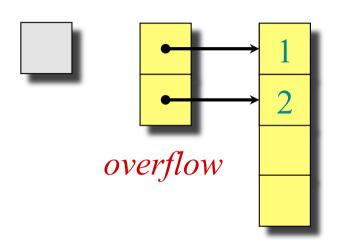


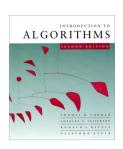
- 1. Insert
- 2. Insert
- 3. Insert



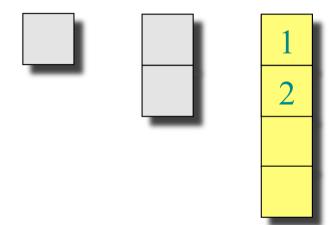


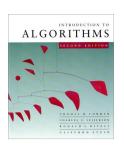
- 1. Insert
- 2. Insert
- 3. Insert



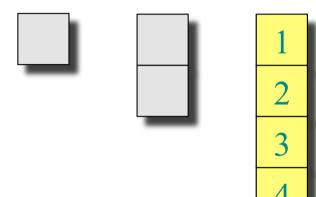


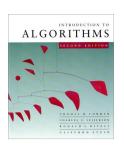
- 1. Insert
- 2. Insert
- 3. Insert



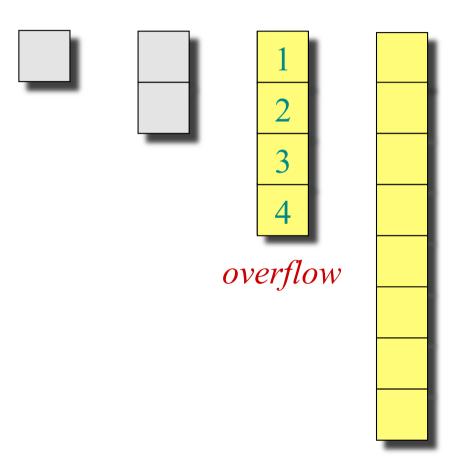


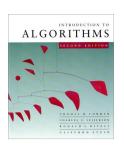
- 1. Insert
- 2. Insert
- 3. Insert
- 4. Insert



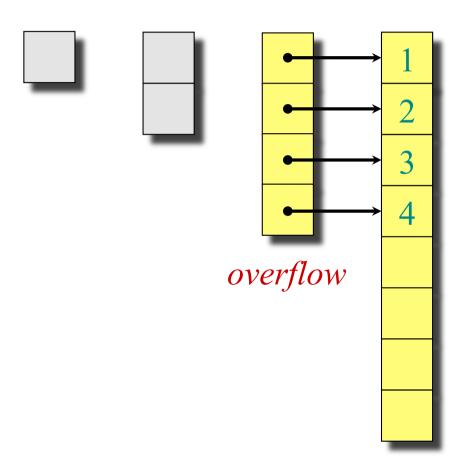


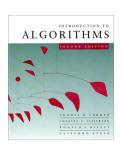
- 1. Insert
- 2. Insert
- 3. Insert
- 4. Insert
- 5. Insert



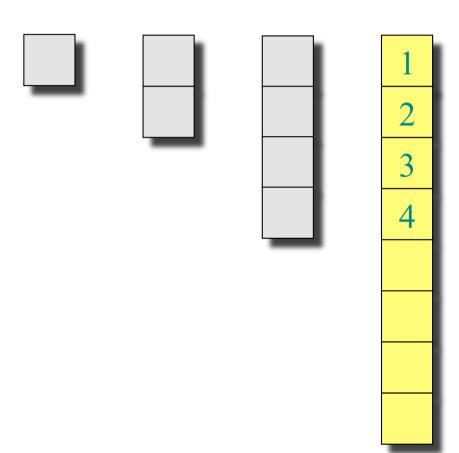


- 1. Insert
- 2. Insert
- 3. Insert
- 4. Insert
- 5. Insert



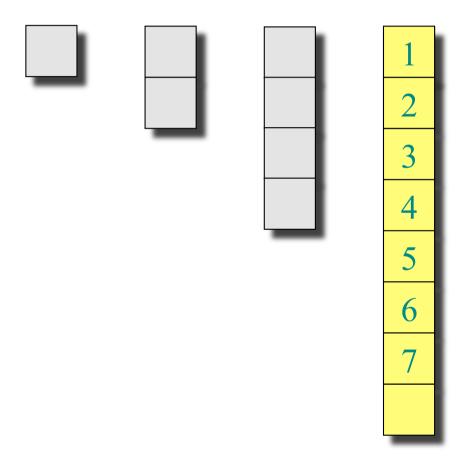


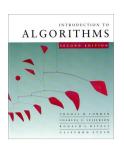
- 1. Insert
- 2. Insert
- 3. Insert
- 4. Insert
- 5. Insert





- 1. Insert
- 2. Insert
- 3. Insert
- 4. Insert
- 5. Insert
- 6. Insert
- 7. Insert



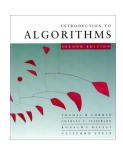


Worst-case analysis

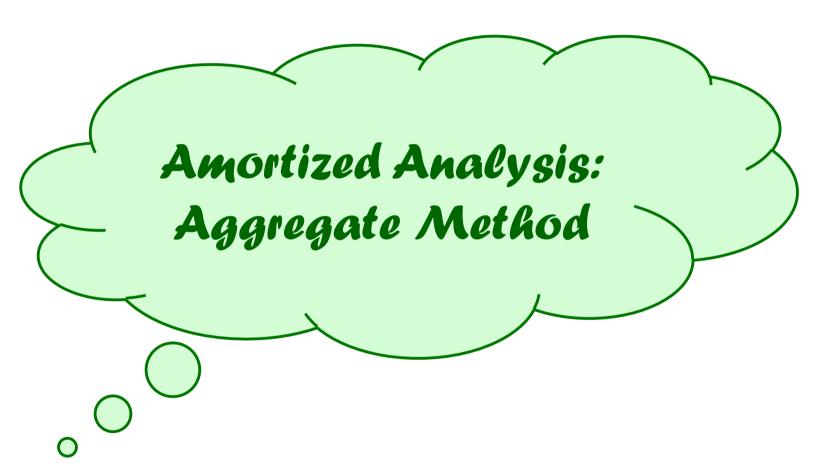
Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

WRONG! In fact, the worst-case cost for n insertions is only $\Theta(n) \ll \Theta(n^2)$.

Let's see why.



Dynamic Tables (Insert)





Tighter analysis

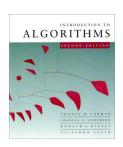
Let c_i = the cost of the *i* th insertion = $\begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$



Tighter analysis

Let c_i = the cost of the *i* th insertion = $\begin{cases} i & \text{if } i-1 \text{ is an exact power of 2,} \\ 1 & \text{otherwise.} \end{cases}$

i	1	2	3	4	5	6	7	8	9	10161
size _i	1	2	4	4	8	8	8	8	16	16
C.	1	1	1	1	1	1	1	1	1	1
i		1	2		4				8	



Tighter analysis (continued)

Cost of *n* insertions =
$$\sum_{i=1}^{n} c_{i}$$

$$\leq n + \sum_{j=0}^{\lfloor \lg(n-1) \rfloor} 2^{j}$$

$$\leq 3n$$

$$= \Theta(n).$$

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

Recall from Introduction...



Slide added by LeongHW (Nov 2013)

(CS3230: Amortized Analysis) Page 31

Example: Consider this sequence

Consider a sequence of operations with cost

$$c_k = \begin{cases} k & \text{if } (k-1) \text{ is a power of 2} \\ 1 & \text{otherwise} \end{cases}$$

Slide added by LeongHW (Nov 2013)

(CS3230: Amortized Analysis) Page 32

Using Amortized Analysis...

Consider a sequence of operations with cost

$$c_k = \begin{cases} 1 + 2^j & \text{if } (k - 1) = 2^j \\ 1 & \text{otherwise} \end{cases}$$

We will show *later* that

We will show *later* th
$$C(n) \leq 3n \quad \text{for all } n$$

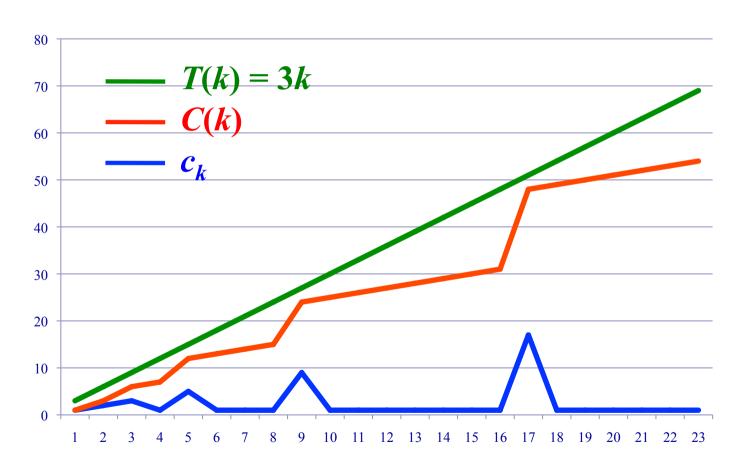
Amortized cost per operation = C(n)/n = 3

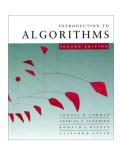
Amortized Time is Constant $\Theta(1)$

(CS3230: Amortized Analysis) Page 33

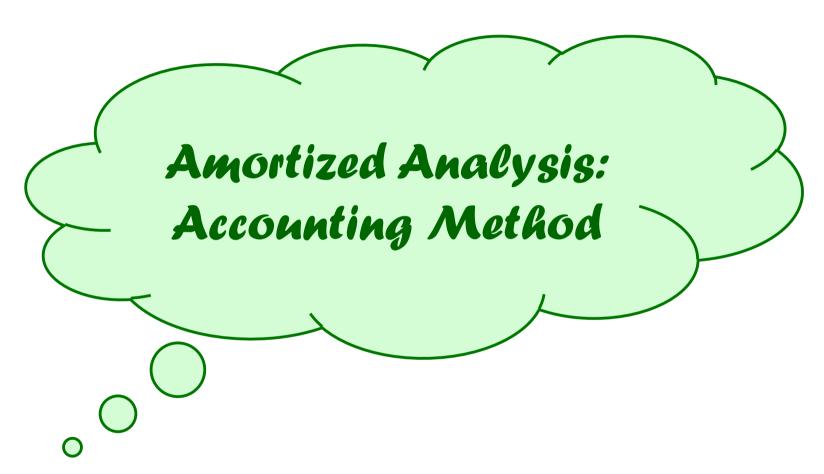
Graphical demonstration

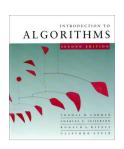
☐ For now, I only give a graphical demo





Dynamic Tables (Insert)





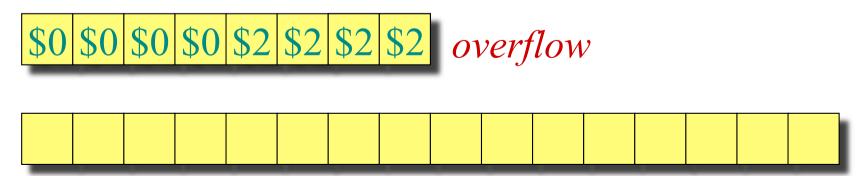
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the *i*th insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:





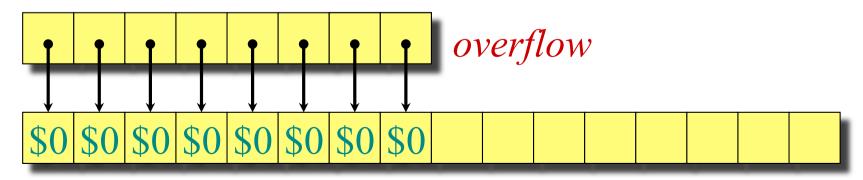
Accounting analysis of dynamic tables

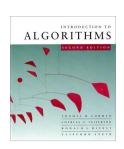
Charge an amortized cost of $\hat{c}_i = \$3$ for the *i*th insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:





Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the *i*th insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

Example:





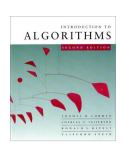


Accounting analysis (continued)

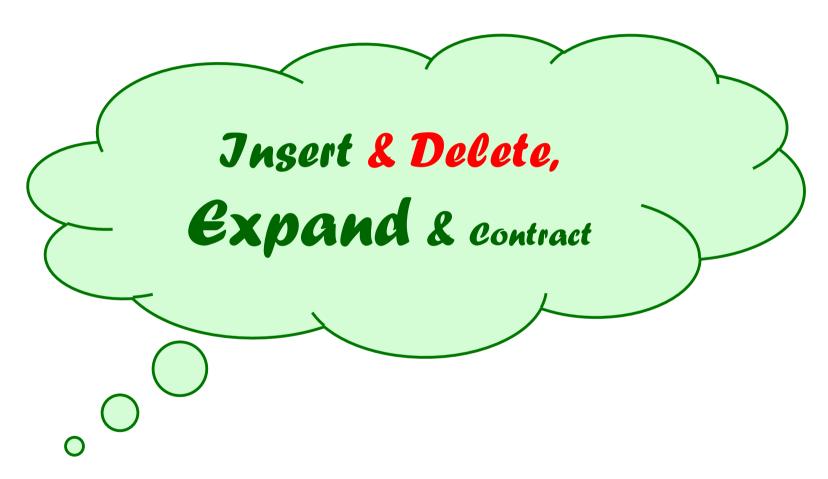
Key invariant: Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

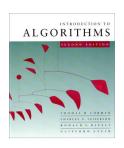
i $size_i$ c_i \hat{c}_i $bank_i$	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
\hat{c}_i	2*	3	3	3	3	3	3	3	3	3
bank _i	1	2	2	4	2	4	6	8	2	4

^{*}Okay, so I lied. The first operation costs only \$2, not \$3.



Dynamic Tables





When to Contract the table?

Solution 1: for Insert & Delete

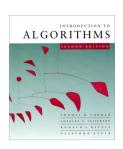
Table overflows
Table < ½ full

Double the table size Halve the table size

NO! Bad idea! WHY

Can cause trashing.

Table Size = 8; If we perform I, D, D, I, I, D, D, I, I, D, D



When to Contract the table?

Solution 2: for Insert & Delete

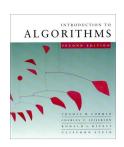
Table overflows

Double the table size

Table $< \frac{1}{4}$ full \rightarrow Halve the table size

This allows the table to save up some credit when $\alpha(T)$ is between $\frac{1}{2}$ to $\frac{1}{4}$ To pay for moving old items to contracted table

Define load factor $\alpha(T)$ $\alpha(T) = num[T] / size[T]$



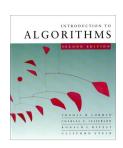
Insert & Delete: with Accounting Method

No change to Insert operation

Insert: amortized cost of $\hat{c}_i = \$3$ (no change)

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

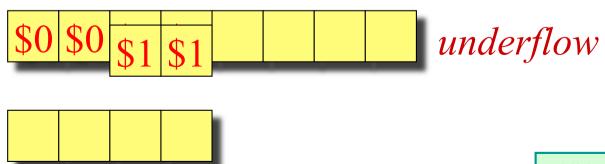


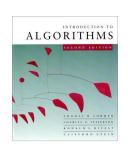
Delete: amortized cost of $\hat{c}_i = \$2$

- \$1 pays for the immediate deletion.
- \$1 stored in deleted cell, for later table halving.

When the table contracts, \$1 pays to copy the old item into new contracted table.

Example:



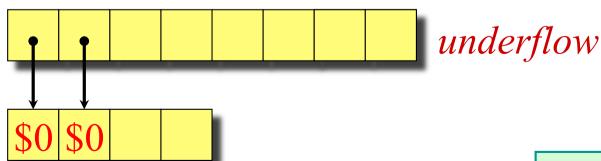


Delete: amortized cost of $\hat{c}_i = \$2$

- \$1 pays for the immediate deletion.
- \$1 stored in deleted cell, for later table halving.

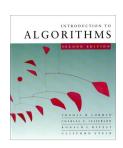
When the table contracts, \$1 pays to copy the old item into new contracted table.

Example:



Slide added by LeongHW (Nov 2013)

October 31, 2005

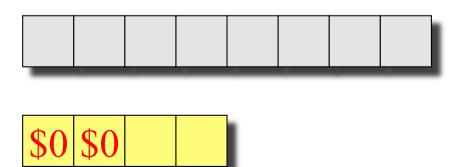


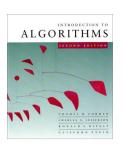
Delete: amortized cost of $\hat{c}_i = \$2$

- \$1 pays for the immediate deletion.
- \$1 stored in deleted cell, for later table halving.

When the table contracts, \$1 pays to copy the old item into new contracted table.

Example:





Insert: amortized cost of $\hat{c}_i = \$3$

Delete: amortized cost of $\hat{c}_i = \$2$

Therefore, *n* insert/delete cost $\Theta(n)$ in worst-case.

Therefore, *n* insert/delete cost $\leq 3n$ in worst-case.

Yes, we do "lose" *some* of the credits saved. (*Where*?) But, still amortized constant time.



Conclusions

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.