CS3230 Lecture 6 – (19-Feb-14)

"Amortized Analysis"

School *of* Computing

❑ **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 Mid-Term Quiz

CS3230 Mid-Term Quiz
Sat, 20-Sep, 10:00am – 11:30am
LT15 (Name starts with A-K)
LT19 (Name starts with L-Z)

There are 4 questions.
Answer ALL of them.

Open Book and open notes.
List of Topics (Uploaded to IVLE)

Sleator and Tarjan

Robert Tarjan and his students (most of them).
At a conference in 2008 to celebrate his 60th birthday.

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

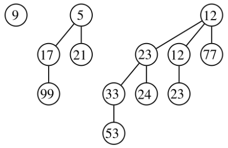
Robert Endre Tarjan

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

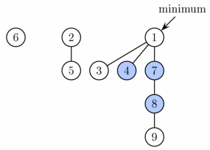
Amortized Analysis & Fibonacci Heaps

J. Vuillemin



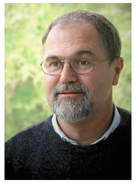
Binomial Heap
Elegant, simple
All in O(lg n)

R. E. Tarjan



Fibonacci Heap
Amortized

Beyond scope of CS3230.
May see these in CS5234.



Stefan Naehar
Kurt Mehlhorn, MPI

"We programmed them already. They are in LEDA."

Thank you.

Q & A

NUS National University of Singapore
School of Computing

CS3230 Lecture 6 – (19-Feb-2014)

NUS National University of Singapore
School of Computing

"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

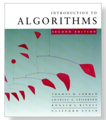
Given *any* sequence of *n* operations $O_1, O_2, \ldots, O_n$ on a data structure D where
D_k = d.s. after applying O_k with cost c_k on D_{k-1}.

$$D_0 \xrightarrow{O_1} D_1 \xrightarrow{O_2} D_1 \xrightarrow{O_3}$$

$$\cdots \xrightarrow{O_{k-1}} D_{k-1} \xrightarrow{O_k} D_k$$

Want to compute $C(n)/n$, where

$$C(n)/n = (c_1 + c_2 + c_3 + \ldots + c_n) / n = \frac{1}{n}\sum_{k=1}^{n} c_k$$

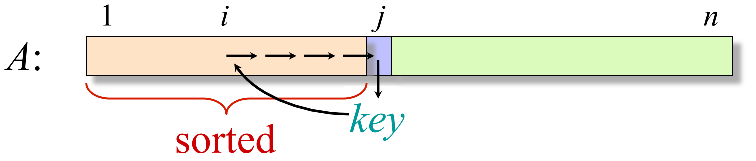


Consider: Insertion sort

INSERTION-SORT (A, n) $\triangleright$ $A[1 \,.\,.\, n]$
for $j \leftarrow 2$ **to** n
  do $key \leftarrow A[j]$
    $i \leftarrow j - 1$
    while $i > 0$ and $A[i] > key$
      do $A[i+1] \leftarrow A[i]$
        $i \leftarrow i - 1$
    $A[i+1] = key$

O_j = "inserts" $A[j]$ into sorted $A[1 \,..\, (j-1)]$

operation O_j with cost is c_j



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

Insertion Sort does $(n-1)$ "insert" operations
- Let O_j = "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**, $c_j = (j-1)$

Total (*worst-case*) *time* for Insertion Sort:

$$C(n) = (c_1 + c_2 + c_3 + \ldots + c_n)$$
$$= (0 + 1 + 2 + \ldots + (n-1)) \approx \mathbf{0.5(n^2)}$$

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

$$C(n)/n \approx \mathbf{0.5n}$$

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:

$$C(n) = (d_1 + d_2 + d_3 + \ldots + d_n)$$
$$= (0 + 1 + 2 + \ldots + (n-1)) \approx \mathbf{0.5(n^2)}$$

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

$$C(n)/n \approx \mathbf{0.5n}$$

This worst-case bound is **TIGHT**.
Eg: elements inserted into BST
in increasing order.

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

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

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

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

Example: Consider this sequence

Consider a sequence of operations with cost

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

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

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

Amortized Complexity A&E

Question:
How many of you keep your room clean and tidy ***all the time***?

Let CTR_k = "CLEAN-AND-TIDY-ROOM" op on the k^{th} 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*

Amortized Complexity A&E

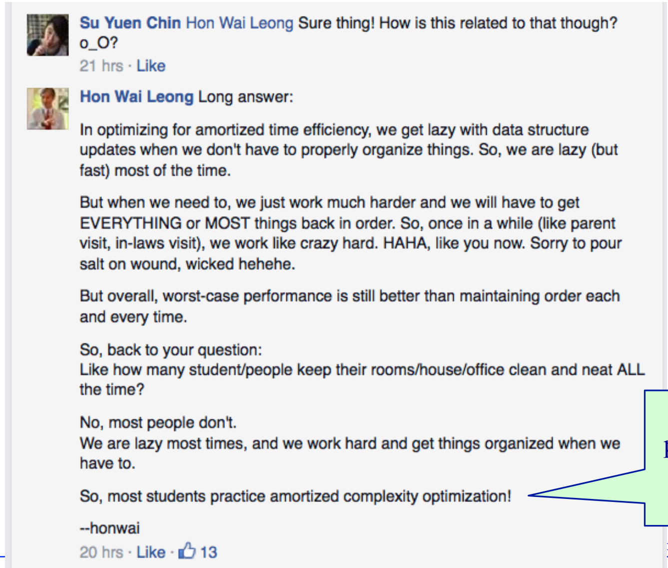
On Sat, 13-9-2014 this appeared on my FB wall.

Amortized Complexity A&E

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

Me think… Amortized Complexity in real life

Amortized Complexity A&E



All students practise amortized complexity optimization

Example: Consider this sequence

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

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

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

When does worst-case happen?

Consider a sequence of operations with cost

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

Worst case happens here

Most operations only cost 1

Worst-case: (Upper bound W_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**	**10**

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

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

Worst case

We will show ***later*** that

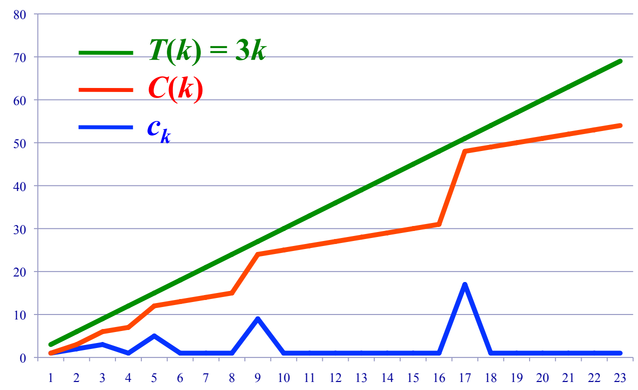
$C(n) \le 3n$ for all n

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

- ❖ It give average $\leq 0.5n$

Amortized complexity analysis (*shown later*)

- ❖ 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

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

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

```
INCREMENT(A)
   i ← 0
   while i < length[A] and A[i] = 1
       do A[i] ← 0          ▷ reset a bit
          i ← i + 1
   if i < length[A]
       then A[i] ← 1        ▷ 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 $(c_1 + c_2 + \ldots + c_k)$

Worst-case analysis

Consider a sequence of n insertions. 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*	*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*

Total cost of n operations	
A[0] flipped every op	n
A[1] flipped every 2 ops	$n/2$
A[2] flipped every 4 ops	$n/2^2$
A[3] flipped every 8 ops	$n/2^3$
… … … …	…
A[i] flipped every 2^i ops	$n/2^i$

Tighter analysis (continued)

Cost of n increments $= \sum_{i=1}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor$

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

$= \Theta(n)$

Wow!
Only 2 bits flipped per increment operation (*amortized*).

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 $Q(k)$

Let's Review Amortized Analysis

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.

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.

A First Example: Binary Counter

Amortized Analysis (Accounting Method)

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:

0 0 0 $1^{\$1}$ 0 $1^{\$1}$ 0

0 0 0 $1^{\$1}$ 0 $1^{\$1}$ $1^{\$1}$ **Cost = \$2**

0 0 0 $1^{\$1}$ $1^{\$1}$ 0 0 **Cost = \$2**

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*

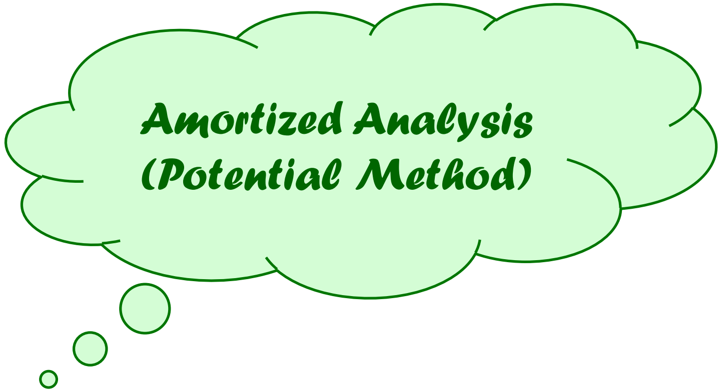
- **When Incrementing,**
  - **Amortized cost for line 3 = \$0** **// use savings**
  - **Amortized cost for line 6 = \$2** **// cost for setting bit**
- **Amortized cost for INCREMENT(A) = \$2**
- **Amortized cost for n INCREMENT(A) = \$2n =$O(n)$**

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
$bank_i$	1	1	2	1	2	2	3	1	2	2

A First Example: Binary Counter



Potential method

IDEA: View the bank account as the potential energy (*à la* physics) of the dynamic set.

Framework:

- Start with an initial data structure D_0.
- Operation i transforms D_{i-1} to D_i.
- The cost of operation i is c_i.
- Define a ***potential function*** $\Phi : \{D_i\} \to \mathrm{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i.
- The ***amortized cost*** $\hat{c}_i$ with respect to Φ is defined to be $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

Understanding potentials

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

- If $\Delta\Phi_i > 0$, then $\hat{c}_i > c_i$. Operation i stores work in the data structure for later use.
- If $\Delta\Phi_i < 0$, then $\hat{c}_i < c_i$. The data structure delivers up stored work to help pay for operation i.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

Summing both sides.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

The series telescopes.

Amortized costs bound the true costs

The total amortized cost of n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$
$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$
$$\geq \sum_{i=1}^{n} c_i$$

since $\Phi(D_n) \geq 0$ and $\Phi(D_0) = 0$.

Potential analysis of INCREMENT

Define the potential of the counter after the i^{th} operation by $\Phi(D_i) = b_i$, the number of 1's in the counter after the i^{th} operation.

Note:

- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i.

Example:

0 0 0 1 0 1 0 $\Phi = 2$

(**0 0 0 1**$^{\$1}$ **0 1**$^{\$1}$ **0** Accounting method)

Calculation of amortized costs

Assume ith INCREMENT resets t_i bits (in line 3).
Actual cost $c_i = (t_i + 1)$
Number of 1's after ith operation: $b_i = b_{i-1} - t_i + 1$

The amortized cost of the ith INCREMENT is

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

Therefore, n INCREMENTs cost $\Theta(n)$ in the worst case.

Second Example:

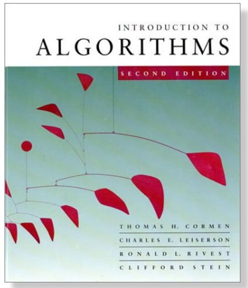
Dynamic Tables

Thank you.

School of Computing

Introduction to Algorithms

6.046J/18.401J

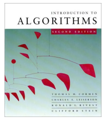


LECTURE 6

Amortized Analysis

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

Prof. Charles E. Leiserson



Example 2: Dynamic Table

Dynamic Table & Appl
(Expands & Contracts,
On Demand)

Slide added by LeongHW (Nov 2013)

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

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

Slide added by LeongHW (Nov 2013)

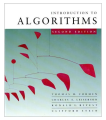
ArrayList API and impl (2)

Code for ensureCapacityInternal

```
/**
 * Increases the capacity of this <tt>ArrayList</tt> instance, if
 * necessary, to ensure that it can hold at least the number of elements
 * specified by the minimum capacity argument.
 *
 * @param   minCapacity   the desired minimum capacity
 */
public void ensureCapacity(int minCapacity) {
    if (minCapacity > 0)
        ensureCapacityInternal(minCapacity);
}

private void ensureCapacityInternal(int minCapacity) {
    modCount++;
    // overflow-conscious code
    if (minCapacity - elementData.length > 0)
        grow(minCapacity);
}
```

Slide added by LeongHW (Nov 2013)



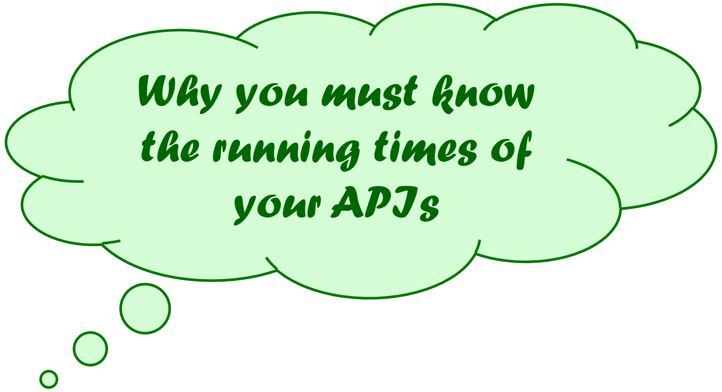
ArrayList API and impl (3)

Code for grow

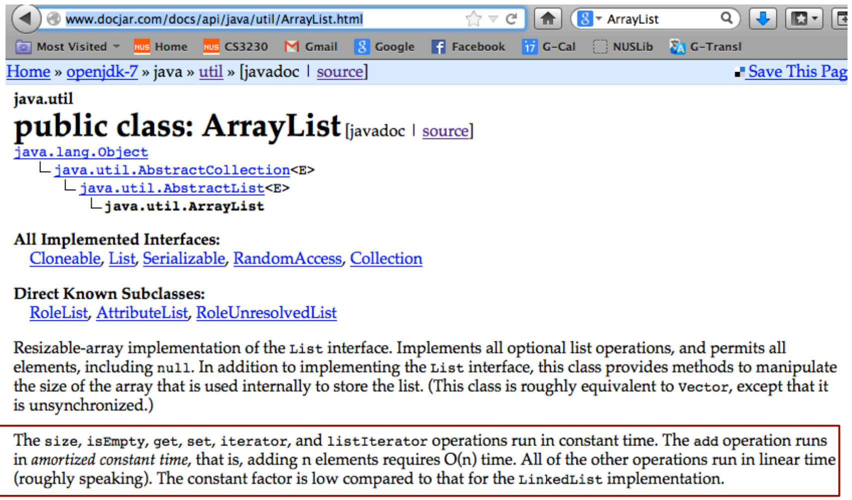
```
/**
 * Increases the capacity to ensure that it can hold at least the
 * number of elements specified by the minimum capacity argument.
 *
 * @param minCapacity the desired minimum capacity
 */
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

Slide added by LeongHW (Nov 2013)

Side-Track:



Important to know your APIs well


www.docjar.com/docs/api/java/util/ArrayList.html

Most Visited | Home | CS3230 | Gmail | Google | Facebook | G-Cal | NUSLib | G-Transl

Home » openjdk-7 » java » util » [javadoc | source] Save This Pag

java.util
public class: ArrayList [javadoc | source]

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList

All Implemented Interfaces:
Cloneable, List, Serializable, RandomAccess, Collection

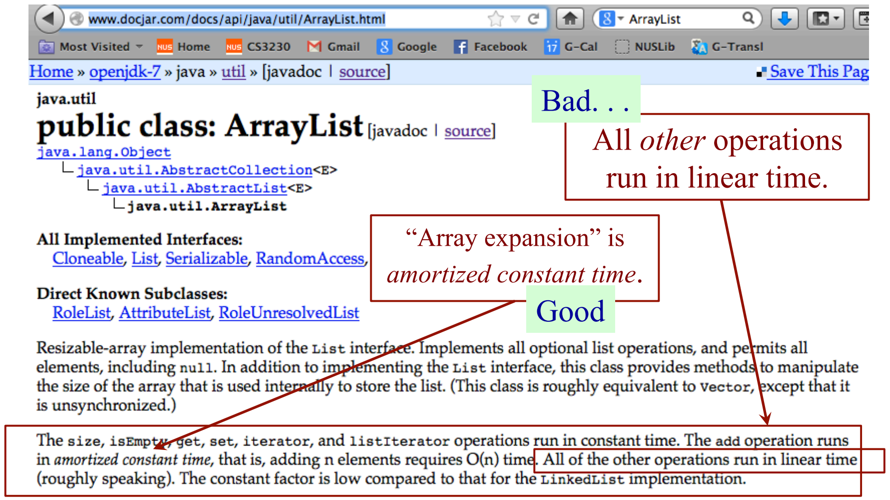
Direct Known Subclasses:
RoleList, AttributeList, RoleUnresolvedList

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.

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

Important to know your APIs well


www.docjar.com/docs/api/java/util/ArrayList.html

Most Visited | Home | CS3230 | Gmail | Google | Facebook | G-Cal | NUSLib | G-Transl

Home » openjdk-7 » java » util » [javadoc | source] Save This Pag

java.util
public class: ArrayList [javadoc | source]

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList

All Implemented Interfaces:
Cloneable, List, Serializable, RandomAccess,

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.

Bad. . .

All *other* operations run in linear time.

"Array expansion" is *amortized constant time*.

Good

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

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

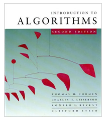
Not all online help gives running time.

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;



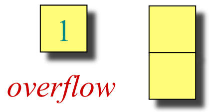
Dynamic Tables

First consider only Insertion

Slide added by LeongHW (Nov 2013)

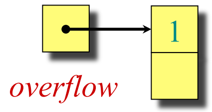
Example of a dynamic table

1. INSERT
2. INSERT



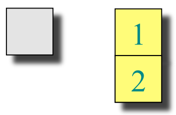
Example of a dynamic table

1. INSERT
2. INSERT



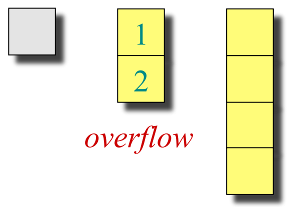
Example of a dynamic table

1. INSERT
2. INSERT



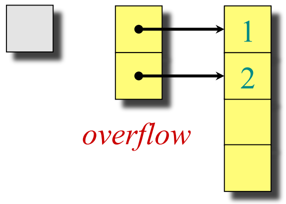
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



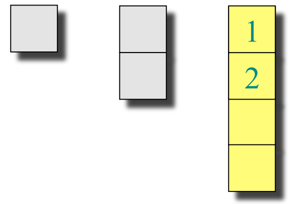
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



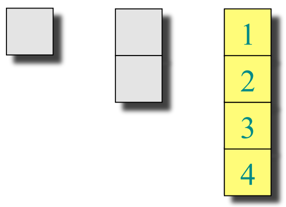
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



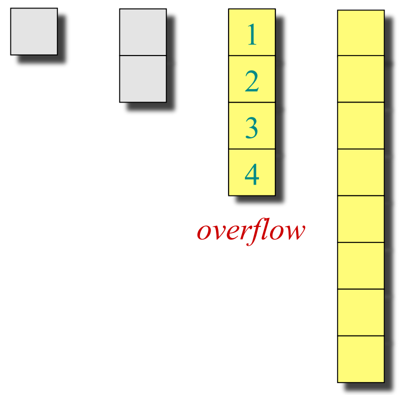
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT



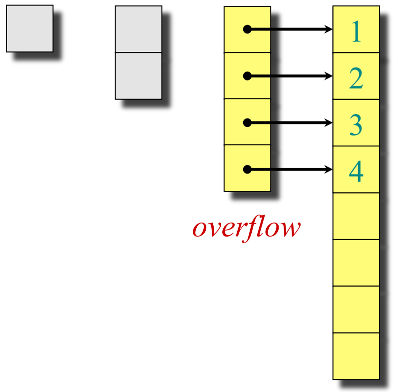
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



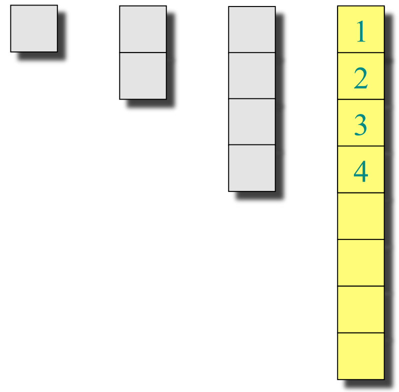
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



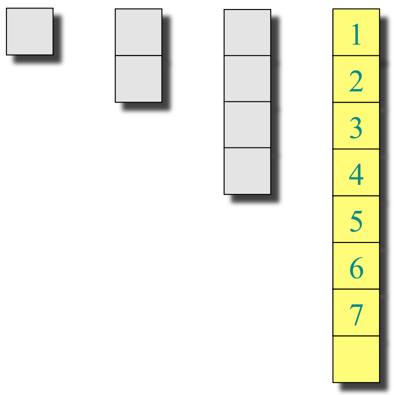
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



Example of a dynamic table

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)

Amortized Analysis: Aggregate Method

Slide added by LeongHW (Nov 2013)

Tighter analysis

Let c_i = the cost of the ith 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	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

Tighter analysis

Let c_i = the cost of the ith 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	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Tighter analysis (continued)

$$\text{Cost of } n \text{ insertions} = \sum_{i=1}^{n} c_i$$

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

$$\le 3n$$

$$= \Theta(n).$$

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

Recall from Introduction…

Slide added by LeongHW (Nov 2013)

Example: Consider this sequence

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

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

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

Slide added by LeongHW (Nov 2013)

Using Amortized Analysis…

Consider a sequence of operations with cost

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

Worst case

We will show ***later*** that

$C(n) \le 3n$ for all n

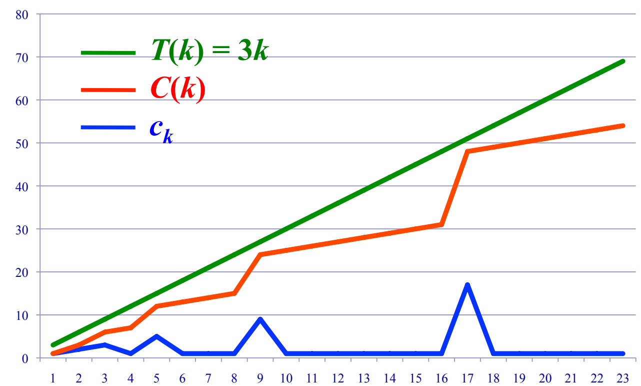
Amortized cost per operation = $C(n)/n = 3$

Amortized Time is Constant $\Theta(1)$

Slide added by LeongHW (Nov 2013)

Graphical demonstration

- For now, I only give a graphical demo



Slide added by LeongHW (Nov 2013)

Dynamic Tables (Insert)

Slide added by LeongHW (Nov 2013)

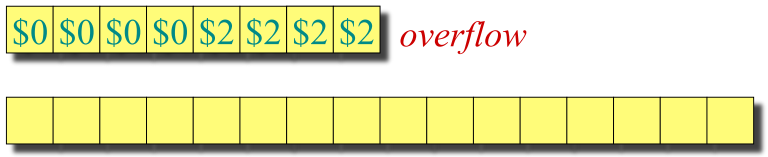
Accounting analysis of dynamic tables

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:



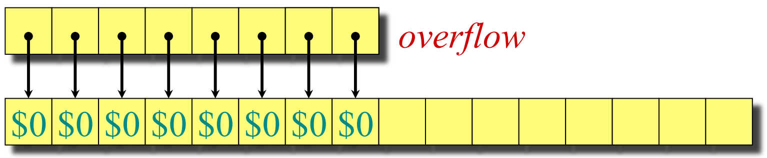
Accounting analysis of dynamic tables

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:



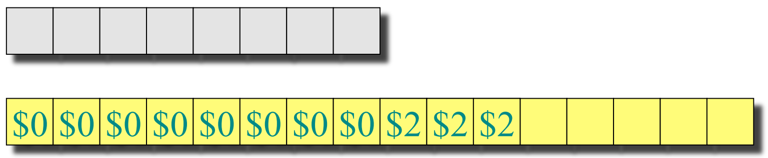
Accounting analysis of dynamic tables

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:



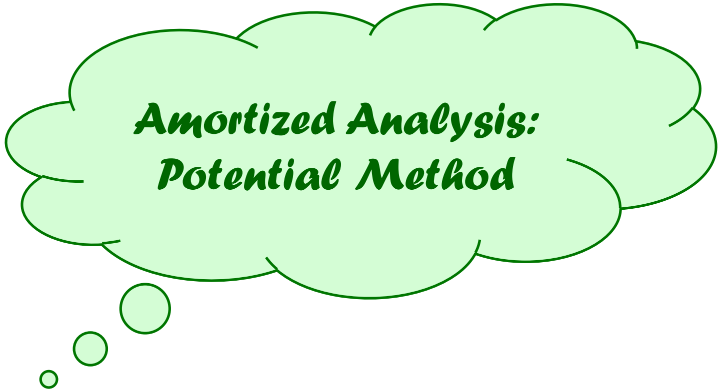
Accounting analysis (continued)

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	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 (Insert)



Slide added by LeongHW (Nov 2013)

Potential analysis of table doubling

Define the potential of the table after the ith insertion by $\Phi(D_i) = 2i - 2^{\lceil \lg i \rceil}$. (Assume that $2^{\lceil \lg 0 \rceil} = 0$.)

Note: In the text [CLRS]
$\Phi(T) = 2num[T] - size[T]$
(the same thing)

Note:
- $\Phi(D_0) = 0$,
- $\Phi(D_i) \geq 0$ for all i.

Example:

•	•	•	•	•	•		

$\Phi = 2 \cdot 6 - 2^3 = 4$

(

\$0	\$0	\$0	\$0	\$2	\$2		

accounting method)

Calculation of amortized costs

The amortized cost of the ith insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

Calculation of amortized costs

The amortized cost of the ith insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise;} \end{cases}$$

$$+ \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)$$

Calculation of amortized costs

The amortized cost of the ith insertion is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise;} \end{cases}$$

$$+ \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)$$

$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise;} \end{cases}$$

$$+ 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}.$$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\hat{c}_i = i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

Eg: $i = 9, (i-1) = 8 = 2^3$

$$2^{\lceil \lg i \rceil} = 2^{\lceil \lg 9 \rceil} = 2^{\lceil 3.17 \rceil} = 16 = 2(i-1)$$

$$2^{\lceil \lg (i-1) \rceil} = 2^{\lceil \lg 8 \rceil} = 8 = (i-1)$$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\hat{c}_i = i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$
$$= i + 2 - 2(i-1) + (i-1)$$

Eg: $i = 9, (i-1) = 8 = 2^3$

$$2^{\lceil \lg i \rceil} = 2^{\lceil \lg 9 \rceil} = 2^{\lceil 3.17 \rceil} = 16 = 2(i-1)$$

$$2^{\lceil \lg (i-1) \rceil} = 2^{\lceil \lg 8 \rceil} = 8 = (i-1)$$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1\end{aligned}$$

Eg: $i = 9,\ (i-1) = 8 = 2^3$

$2^{\lceil \lg i \rceil} = 2^{\lceil \lg 9 \rceil} = 2^{\lceil 3.17 \rceil} = 16 = 2(i-1)$

$2^{\lceil \lg (i-1) \rceil} = 2^{\lceil \lg 8 \rceil} = 8 = (i-1)$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Eg: $i = 9,\ (i-1) = 8 = 2^3$

$2^{\lceil \lg i \rceil} = 2^{\lceil \lg 9 \rceil} = 2^{\lceil 3.17 \rceil} = 16 = 2(i-1)$

$2^{\lceil \lg (i-1) \rceil} = 2^{\lceil \lg 8 \rceil} = 8 = (i-1)$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\hat{c}_i = 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil}$$

Eg: $i = 7,\ (i-1) = 6 \neq 2^k$

$2^{\lceil \lg i \rceil} = 2^{\lceil \lg (i-1) \rceil} = 2^{\lceil \lg 7 \rceil} = 2^{\lceil \lg 6 \rceil} = 8$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3 \qquad \text{(since } 2^{\lceil \lg i \rceil} = 2^{\lceil \lg (i-1) \rceil}\text{)}\end{aligned}$$

Eg: $i = 7,\ (i-1) = 6 \neq 2^k$

$2^{\lceil \lg i \rceil} = 2^{\lceil \lg (i-1) \rceil} = 2^{\lceil \lg 7 \rceil} = 2^{\lceil \lg 6 \rceil} = 8$

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

Calculation

Case 1: $i - 1$ is an exact power of 2.

$$\begin{aligned}\hat{c}_i &= i + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= i + 2 - 2(i-1) + (i-1) \\ &= i + 2 - 2i + 2 + i - 1 \\ &= 3\end{aligned}$$

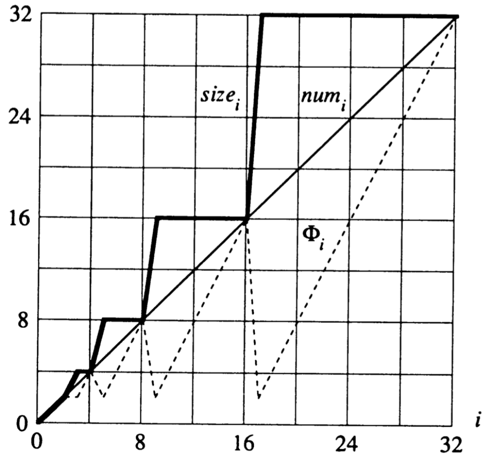
Case 2: $i - 1$ is *not* an exact power of 2.

$$\begin{aligned}\hat{c}_i &= 1 + 2 - 2^{\lceil \lg i \rceil} + 2^{\lceil \lg (i-1) \rceil} \\ &= 3\end{aligned}$$

Therefore, n insertions cost $\Theta(n)$ in the worst case.

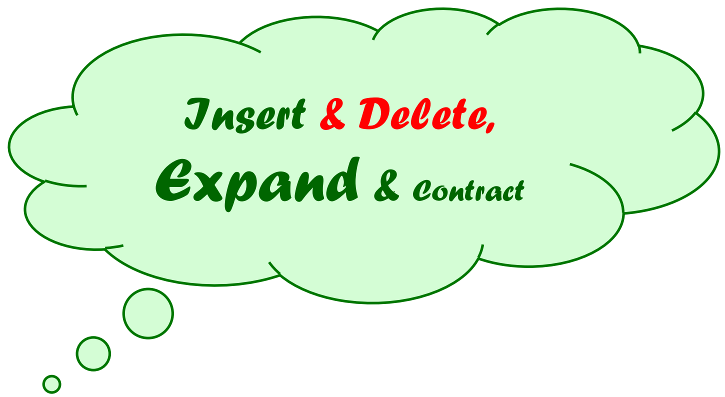
Exercise: Fix the bug in this analysis to show that the amortized cost of the first insertion is only 2.

Graphical Illustration



Slide added by LeongHW (Nov 2013)

Dynamic Tables



Slide added by LeongHW (Nov 2013)

When to Contract the table?

Solution 1: for Insert & Delete

Table overflows Double the table size
Table < ½ full 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

Slide added by LeongHW (Nov 2013)

When to Contract the table?

Solution 2: for Insert & Delete

Table overflows ➔ Double the table size
Table < ¼ full ➔ Halve the table size

This allows the table to save up some credit when $\alpha(T)$ is between ½ to ¼
To pay for moving old items to contracted table

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

Slide added by LeongHW (Nov 2013)

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.

Slide added by LeongHW (Nov 2013)

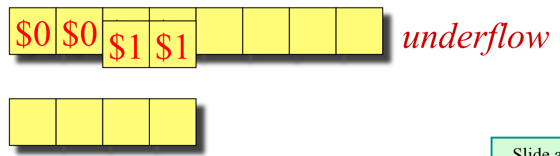
Accounting: Insert & Delete

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)

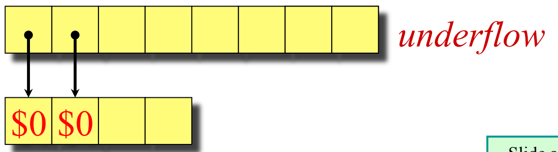
Accounting: Insert & Delete

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)

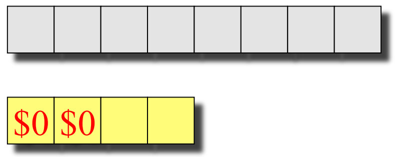
Accounting: Insert & Delete

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)

Accounting Method

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.

Slide added by LeongHW (Nov 2013)

Insert & Delete : Potential Method

Need new Potential function

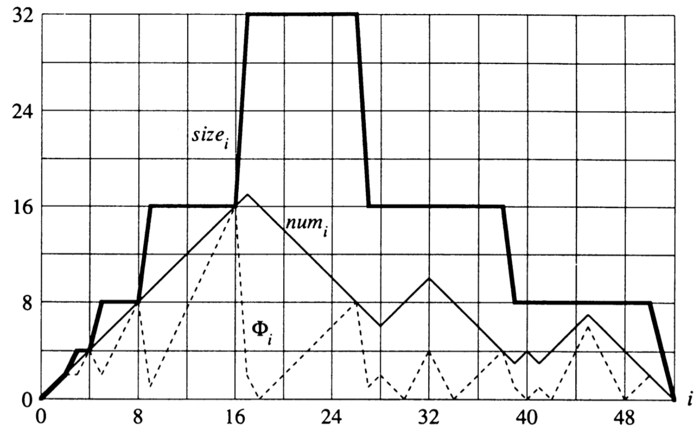
$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2 , \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2 \end{cases}$$

After expansion/contraction,
$\alpha(T) = ½$ and $\Phi(T) = 0$.

After that, $\Phi(T)$ increases
when $\alpha(T)$ increases from ½ to 1, or
when $\alpha(T)$ decreases from ½ to ¼

Slide added by LeongHW (Nov 2013)

Graphical Illustration



Slide added by LeongHW (Nov 2013)

Insert & Delete : Potential Method

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2 \text{ ,} \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2 \end{cases}$$

Work out all the details yourself.

Prove amortized cost for any Insert ≤ 3,
amortized cost for any Delete ≤ 2.

Refer to [CLRS] Ch.17.4, pp. 463—471.

Slide added by LeongHW (Nov 2013)

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.