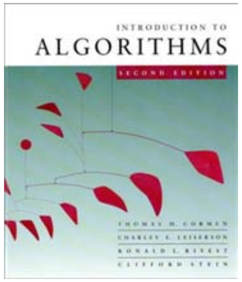
CS 5633 -- Spring 2005



Dynamic Tables

Carola Wenk

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

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).

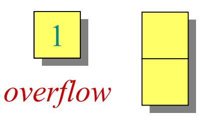
Problem: What if we don't know the proper size in advance?

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.

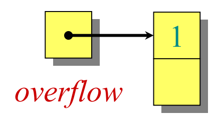
Example of a dynamic table

1. INSERT
2. INSERT



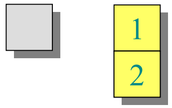
Example of a dynamic table

1. INSERT
2. INSERT



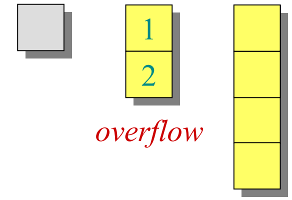
Example of a dynamic table

1. INSERT
2. INSERT



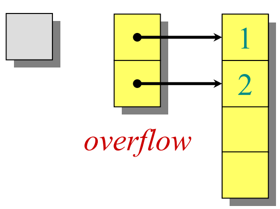
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



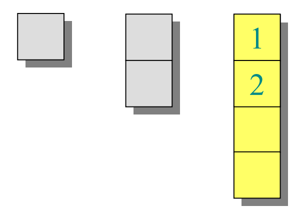
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



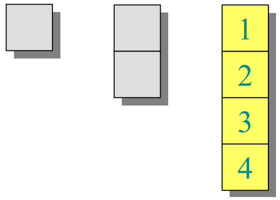
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



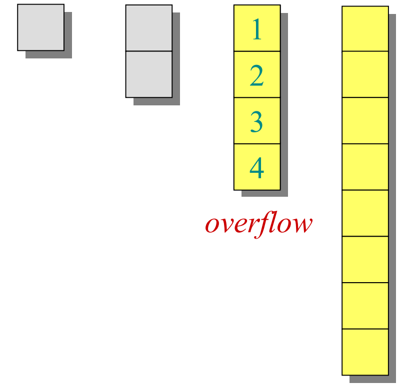
Example of a dynamic table

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



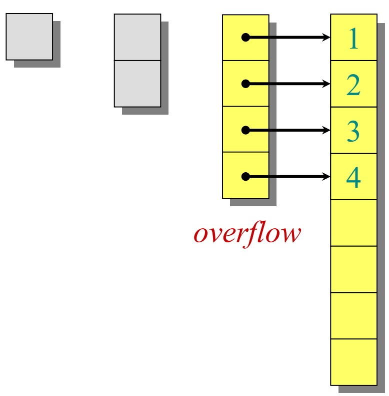
Example of a dynamic table

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



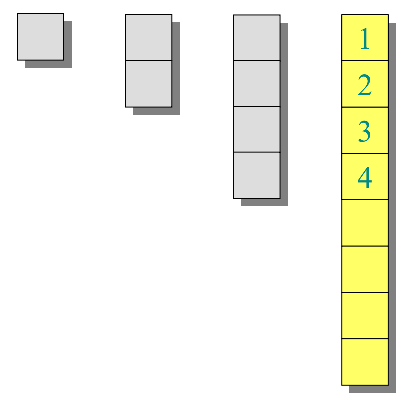
Example of a dynamic table

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



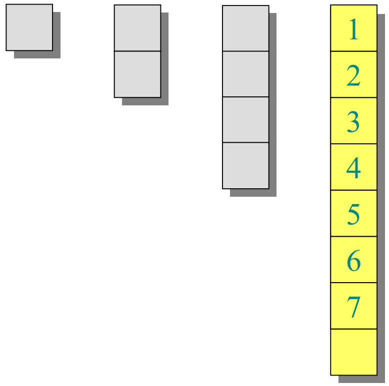
Example of a dynamic table

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 $O(n)$. Therefore, the worst-case time for n insertions is $n \cdot O(n) = O(n^2)$.

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

Let's see why.

Tighter analysis

Let $c_i =$ the cost of the i th insertion

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										

Tighter analysis

Let $c_i =$ the cost of the i th insertion

$= 1 +$ cost to double array size

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
	?	?	?	?	?	?	?	?	?	?

Tighter analysis

Let $c_i =$ the cost of the ith insertion

$= 1 +$ cost to double array size

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
	0	1	2	0	4	0	0	0	8	0

Tighter analysis

Let $c_i =$ the cost of the ith insertion

$= 1 +$ cost to double array size

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
	0	1	2	0	4	0	0	0	8	0
(sum)	1	2	3	1	5	1	1	1	9	1

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 $\Theta(n)/n = \Theta(1)$.

Amortized analysis

An ***amortized analysis*** is any strategy for analyzing a **sequence** of operations:

- compute the total cost of the sequence, OR
- amortized cost of an operation = average cost per operation, averaged over the number of operations in the sequence
- amortized cost can be small, even though a single operation within the sequence might be expensive

Amortized analysis

Even though we're taking averages, however, probability is not involved!

- 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.~~

Won't cover in class

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.

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, and
- 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.

- Thus, the total amortized costs provide an upper bound on the total true costs.

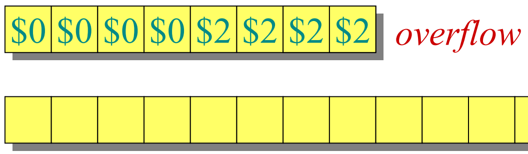
Accounting analysis of dynamic tables

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:



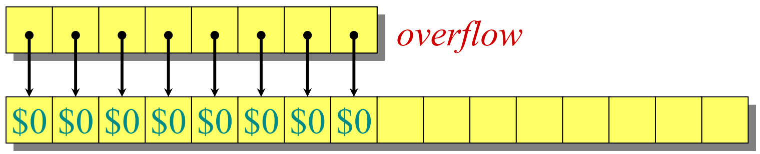
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith 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:



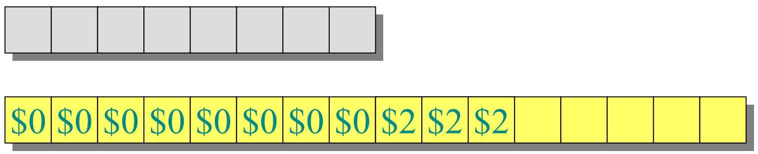
Accounting analysis of dynamic tables

Charge an amortized cost of $\hat{c}_i = \$3$ for the ith 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.

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.
- Different schemes may work for assigning amortized costs in the accounting method, sometimes yielding radically different bounds.