Analysis of Algorithms (Fall 2013) Istanbul Technical University Computer Eng. Dept.

Chapter 17: Amortized Analysis

Course slides from Susan Bridges @MS State have been used in preparation of these slides.

Last updated: December 9, 2009

Purpose

- Understand amortized analysis, when it is used, and how it differs from the average-case analysis
- Be able to apply techniques of aggregate analysis, accounting method, and potential method to analyze operations on simple data structures

Outline

- Dynamic Tables
- Aggregate Analysis
- Accounting Method
- Potential Method

Amortized Analysis

- We examined ***worst-case***, ***average-case*** and ***best-case*** analysis performance
- In **amortized analysis** we care for the cost of one operation if considered *in a sequence of n operations*
  - In a sequence of n operations, some operations may be cheap, some may be expensive (actual cost)
  - The **amortized cost** of an operation equals the *total* cost of the n operations divided by n

Amortized Analysis

- Think of it this way:
  - You have a bank account with 1000TL and you want to go shopping and purchase some items
  - Some items you buy cost 1TL, some items you buy cost 100TL
  - You purchase 20 items in total, therefore the amortized cost of each purchase is 5TL

Amortized Analysis

- Purpose is to accurately compute the *total* time spent in executing a sequence of operations on a data structure
- Three different approaches:
  - aggregate method: brute force
  - accounting method: assign costs to each operation so that it is easy to sum them up while still ensuring that result is accurate
  - potential method: a more sophisticated version of the accounting method

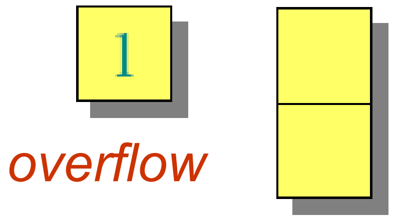
Dynamic Tables

How Large Should a Hash Table Be?

Goal: Make the table as small as possible, but large enough so that it will not overflow (or otherwise become inefficient)
Problem: What if we do not 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

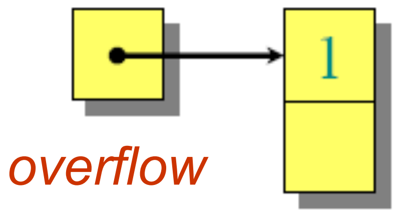
Example of a Dynamic Table

1. INSERT
2. INSERT



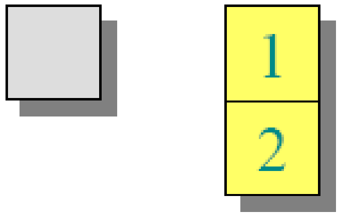
Example of a Dynamic Table

1. INSERT
2. INSERT



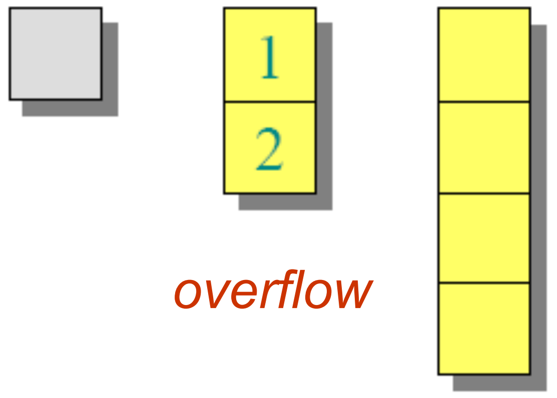
Example of a Dynamic Table

1. INSERT
2. INSERT



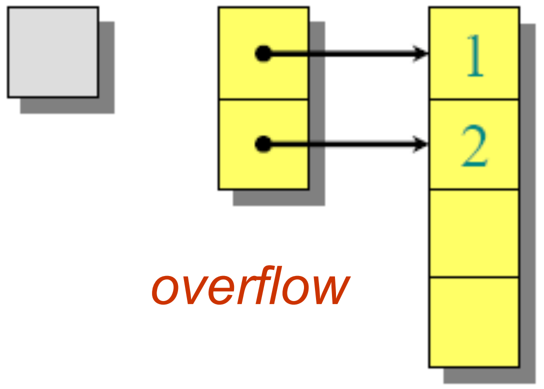
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



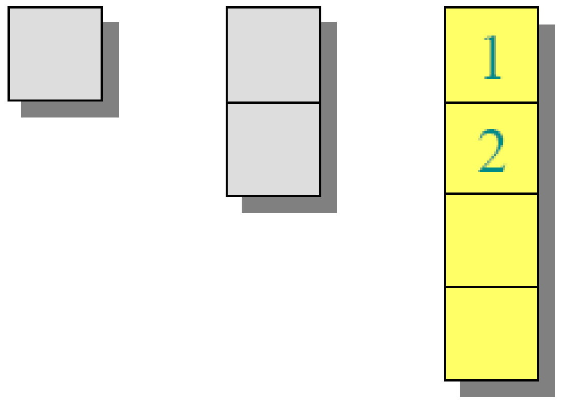
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



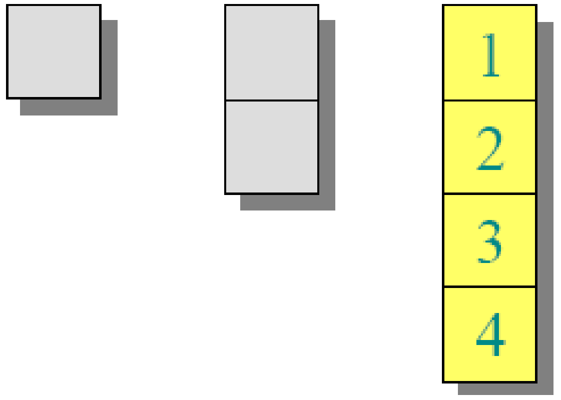
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



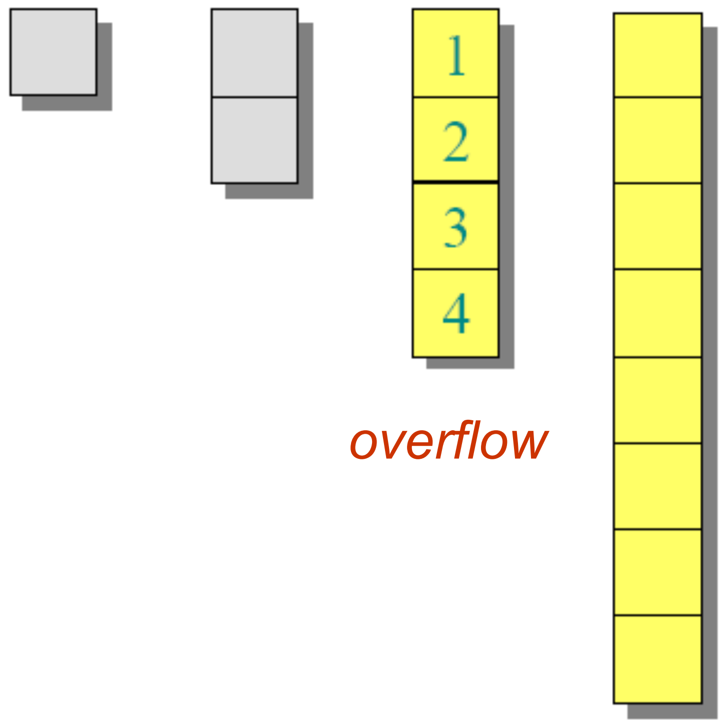
Example of a Dynamic Table

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



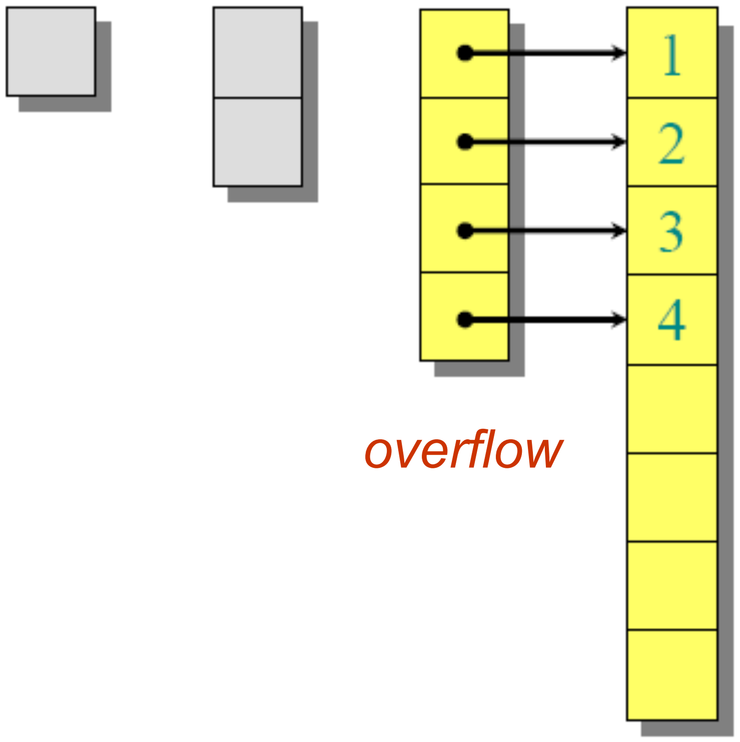
Example of a Dynamic Table

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



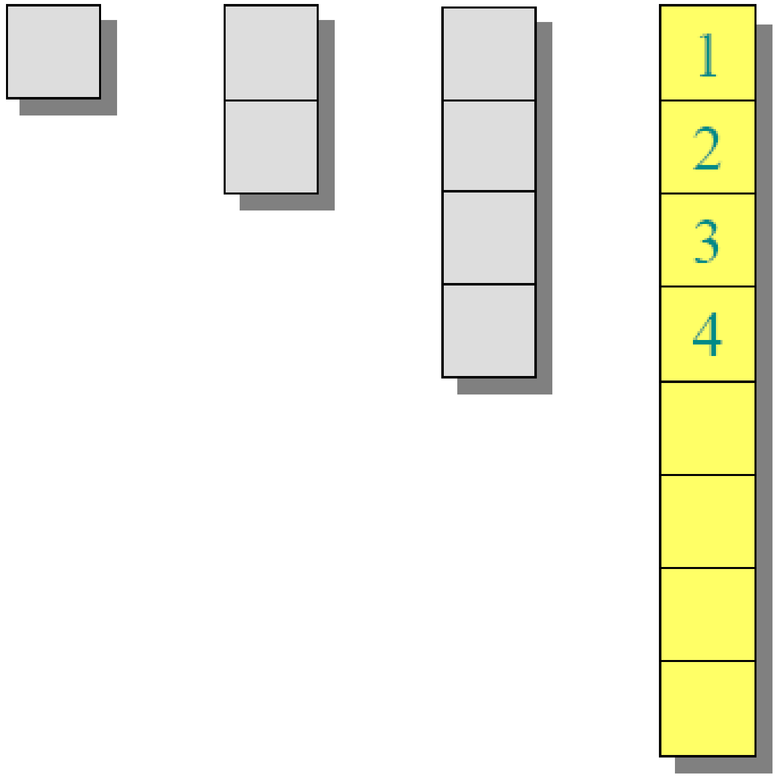
Example of a Dynamic Table

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



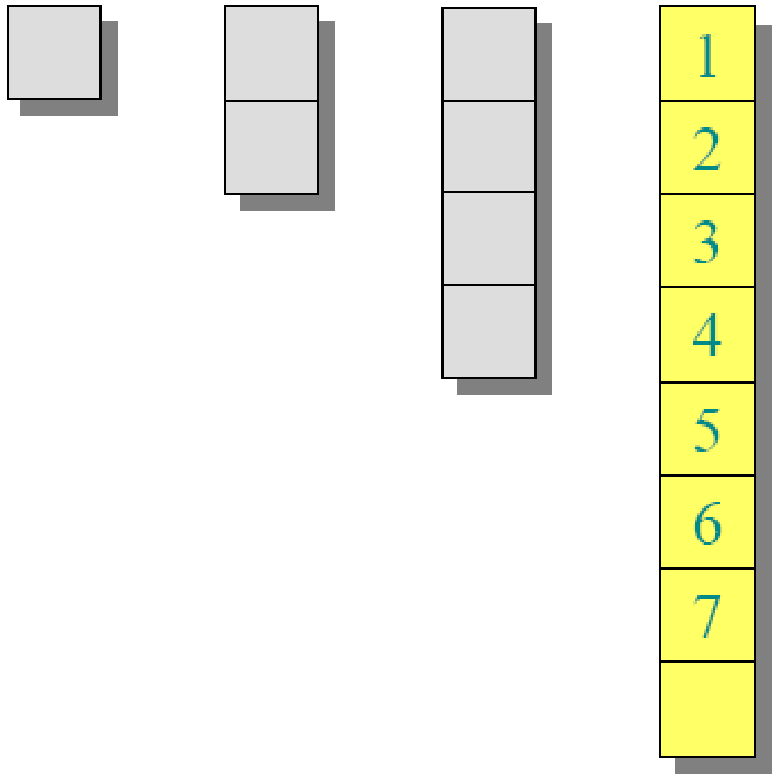
Example of a Dynamic Table

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
- 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 us see why…

Tighter Analysis

Let c_i = the cost of the i th insertion

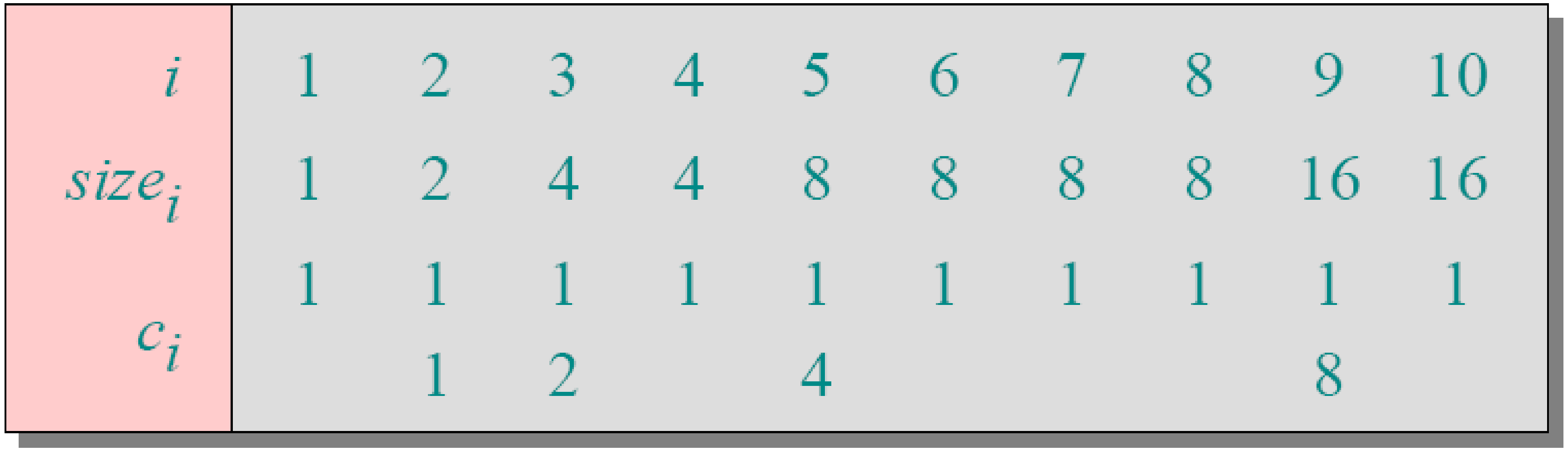
$$= \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2, \\ 1 & \text{otherwise} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	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 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	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 (cont.)

Cost of *n* 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

Amortized Analysis

- Is any strategy for analyzing a sequence of operations to show that
  - average cost per operation is small, even though a single operation within sequence might be expensive
- Does not involve probability even though we are taking averages! (differs from average-case analysis)
- Guarantees average performance of each operation in *worst case*

Types of Amortized Analysis

- Three most common techniques used in amortized analysis:
  - aggregate analysis
  - accounting method
  - potential method

Aggregate Analysis

- We determine an upper bound T(n) on total cost of sequence of *n* operations
- Average cost per operation is then T(n)/n
- We take average cost as amortized cost of each operation, so that all operations have same amortized cost

Accounting Method

- We determine an amortized cost of each operation
- When there is more than one type of operation, each type of operation may have a different amortized cost
- Accounting method overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure
- The credit is used later in the sequence to pay for operations that are charged less than they actually cost

Potential Method

- Is like accounting method in that we determine amortized cost of each operation and may overcharge operations early on to compensate for undercharges later
- Maintains the credit as “potential energy” of data structure as a whole instead of associating the credit with individual objects within the data structure

Note

- Charges assigned during an amortized analysis are for analysis pruposes only
- They need not and should not appear in the code
- If, for example, a credit is assigned to an object x when using the accounting method, there is no need to assign an appropriate amount to some attribute *credit*[*x*] in the code

Benefits of Amortized Analysis

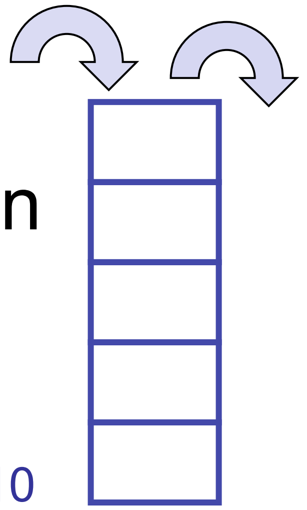
- The insight into a particular data structure gained by performing an amortized analysis can help in optimizing the design
- We will use the potential method to analyze a dynamically expanding and contracting table later on

Aggregate Analysis

Aggregate Analysis

- In aggregate analysis, we show that
  - For all *n*, a sequence of operations takes *worst-case* time T(*n*) in total
- In worst-case, average cost, or amortized cost, per operation is therefore T(n)/n
- Note
  - This amortized cost applies to each operation, even when there are several types of operations in the sequence
  - The other two methods (accounting and potential) may assign different amortized costs to different types of operations

Aggregate Analysis: Stack Operations



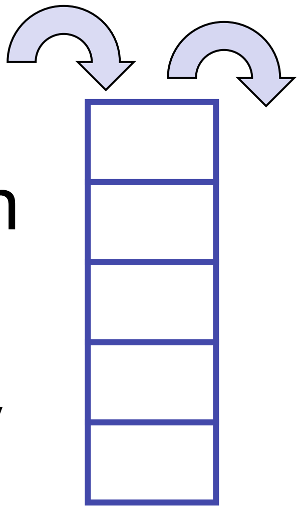
- Example: Analyze stacks that have been augmented with a new operation
- Operations are
  - Push(*S*, *x*): pushes object *x* onto stack *S*
  - Pop(*S*): pops top of stack *S*, returns popped object

  seen in Ch. 10 Elementary Data Structures

  - Multipop(*S*, *k*): pops the top *k* elements, or pops entire stack if it contains fewer than *k* objects

Aggregate Analysis: Stack Operations

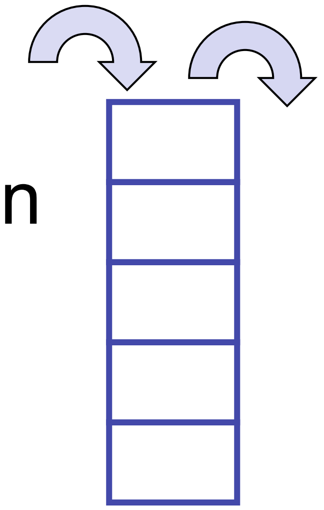
- Example: Analyze stacks that have been augmented with a new operation
- Operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise



```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and $k \neq 0$
2      do POP(S)
3          $k \leftarrow k - 1$
```

Aggregate Analysis: Stack Operations

- Example: Analyze stacks that have been augmented with a new operation

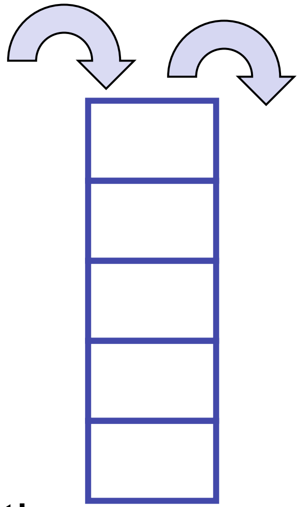


```
top ->23
      17
       6
      39
      10          top -> 10
      47                 47
      —                  —          —
```

Top 4 objects popped by MULTIPOP(S, 4)

MULTIPOP(S, 7) empties the stack because there are fewer than 7 objects remaining

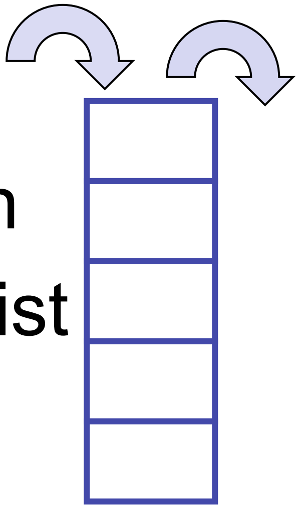
Aggregate Analysis: Stack Operations



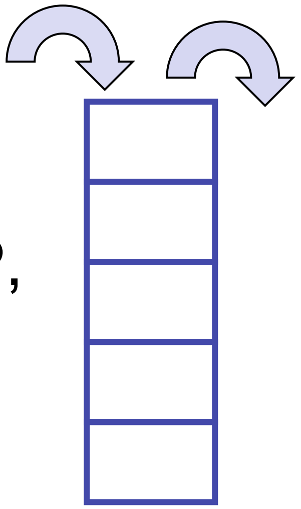
- Example: Analyze stacks that have been augmented with a new operation
- Running time of MULTIPOP(S, k) on a stack of s objects?
  - Actual running time linear in number of POP operations executed, and thus it suffices to analyze MULTIPOP in terms of abstract costs of 1 each for PUSH and POP
  - Number of operations of **while** loop is number min(s, k) of objects popped off stack
  - For each iteration of loop, one call made to POP in line 2
  - Thus, total cost of MULTIPOP is min(s,k), and actual running time is linear function of this cost

Aggregate Analysis: Stack Operations

- Example: Analyze stacks that have been augmented with a new operation
- Implement with either array or linked list
  - time for Push is O(1)
  - time for Pop is O(1)
  - time for Multipop is O(min(|S|,k))

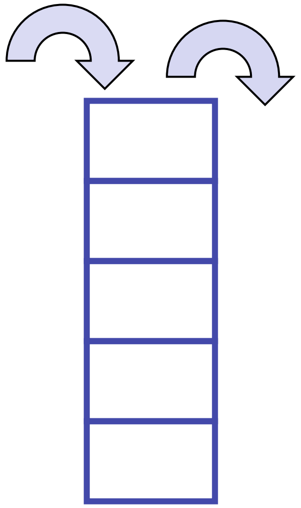


Aggregate Analysis: Stack Operations



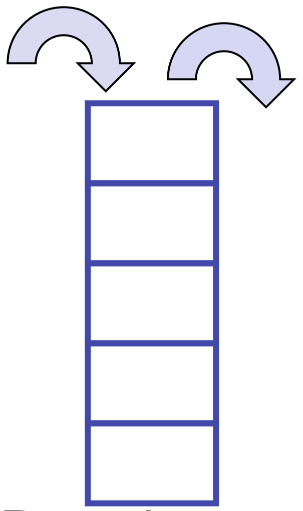
- Example: Analyze stacks that have been augmented with a new operation
- Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack
  - Worst-case cost of a MULTIPOP operation in the sequence is O(n), since stack size is at most n
  - Worst-case cost of any stack operation is therefore O(n), and hence a sequence of n operations costs O(n^2), since we may have O(n) MULTIPOP operations costing O(n) each
  - Although this analysis is correct, O(n^2) result, obtained by considering the worst-case cost of each operation individually, is not tight

Aggregate Analysis: Stack Operations



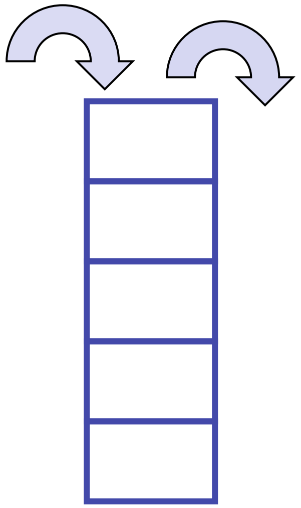
- Example: Analyze stacks that have been augmented with a new operation
- Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations
  - In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most O(n)
  - Why? Each object can be popped at most once for each time it is pushed
  - Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n

Aggregate Analysis: Stack Operations



- Example: Analyze stacks that have been augmented with a new operation
- Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations
  - For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of O(n) time
  - Average cost of an operation is O(n)/n = O(1)
  - In aggregate analysis, we assign the amortized cost of each operation to be the average cost
  - In this example, therefore, all three stack operations have an amortized cost of O(1)

Aggregate Analysis: Stack Operations



- Example: Analyze stacks that have been augmented with a new operation
- Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations
  - We emphasize again that although we have just shown that the average cost, and hence running time, of a stack operation is O(1), no probabilistic reasoning was involved
  - We actually showed a *worst-case* bound of O(n) on a sequence of n operations
  - Dividing this cost by n yielded the average cost per operation, or the amortized cost

Aggregate Analysis: Incrementing a Binary Counter

- Example: Implementing a k-bit binary counter that counts upward from 0
- We use an array A[0..k-1] of bits, where *length*[A]=k, as the counter
- A binary number x that is stored in the counter has its lowest-order bit in A[k-1], so that x = $\sum_{i=0}^{k-1} A[i] \cdot 2^i$
- Initially, x = 0, and thus A[i] = 0 for i= 0, 1, …, k-1
- To add 1 (modulo 2^k) to the value in the counter, we use the following procedure

Aggregate Analysis: Incrementing a Binary Counter

```
INCREMENT(A)
1   i ← 0
2   while i < length[A] and A[i] = 1
3       do A[i] ← 0
4           i ← i + 1
5   if i < length[A]
6       then A[i] ← 1
```

This procedure *resets* the first i-th sequence of 1 bits and *sets* A[i] equal to 1 (ex. 0$\underline{011}$ → 0100, 01$\underline{01}$ → 0110, $\underline{0111}$ → 1000)

Aggregate Analysis: Incrementing a Binary Counter

4-bit counter:

Counter value	COUNTER	Bits flipped (work T(n))
0	0 0 0 **0**	0
1	0 0 **0 1**	1
2	0 0 1 **0**	3
3	0 **0 1 1**	4
4	0 1 0 **0**	7
5	0 1 **0 1**	8
6	0 1 1 **0**	10
7	0 **1 1 1**	11
8	1 0 0 **0**	15
	$A_3A_2A_1A_0$	

highlighted are bits that flip at each increment

Aggregate Analysis: Incrementing a Binary Counter

- Example: Implementing a k-bit binary counter that counts upward from 0
- A naive approach says that a sequence of n operations on a n-bit counter needs O(n^2) work
- Each INCREMENT() takes up to O(n) time. n INCREMENT() operations can take O(n^2) time
- Amortized analysis with accounting method
- We show that amortized cost per INCREMENT() is only O(1) and the total work O(n)
- OBSERVATION: In example, T(n) (work) is never twice the amount of counter value (total # of increments)

Aggregate Analysis: Incrementing a Binary Counter

- Example: Implementing a k-bit binary counter that counts upward from 0
- Charge each 0 → 1 flip 2TL in line 6
- 1TL pays for the 0 → 1 flip in line 6
- 1TL is deposited to pay for the 1 → 0 flip later in line 3

- Therefore, a sequence of n INCREMENTS() needs

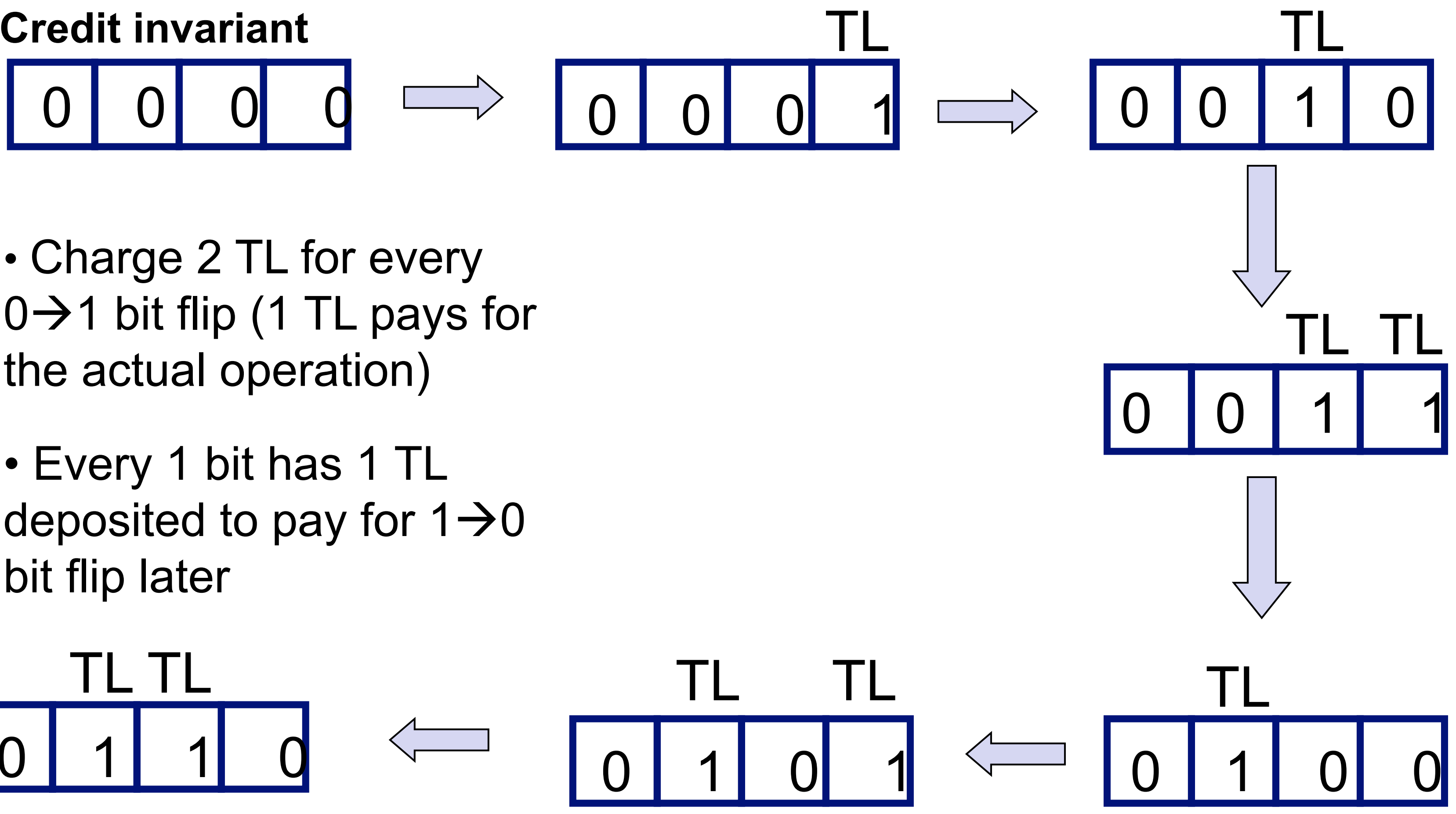
$$\boxed{T(n)= 2n \text{ TL}}$$

- Each INCREMENT() has an amortized cost of

$$\boxed{2n/n = O(1)}$$

Aggregate Analysis: Incrementing a Binary Counter

Credit invariant



- Charge 2 TL for every 0→1 bit flip (1 TL pays for the actual operation)
- Every 1 bit has 1 TL deposited to pay for 1→0 bit flip later

Types of Amortized Analysis

Three most common techniques used in amortized analysis:

- ***aggregate*** analysis
- ***accounting*** method
- ***potential*** method

- We have 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

Accounting Method

- We assign differing charges to different operations, with some operations charged more or less than they actually cost
- The amount we charge an operation is called its amortized cost
- When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as credit
- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost
- Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up
- This method is very different from aggregate analysis, in which all operations have the same amortized cost

Accounting Method

- Charge *i* th operation a fictitious ***amortized cost*** $\hat{c}_i$, where 1 TL 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

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

  for all *n* (where c_i: actual cost of *i* th operation)
- Thus, the total amortized costs provide an upper bound on the total true costs

Accounting Method: Stack Operations

- Recall the actual costs of the operations were

PUSH	1
POP	1
MULTIPOP	min(k,s) (k: argument to MULTIPOP, s: stack size)

- Let us assign the following amortized costs:

PUSH	2
POP	0
MULTIPOP	0

- Note: Amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable
- Here, all three amortized costs are O(1), although in general the amortized costs of the operations under consideration may differ asymtotically

Accounting Method: Stack Operations

- We will now show that we can pay for any sequence of stack operations by charging the amortized costs
- Suppose we use a 1 TL coin to represent each unit of cost
- We start with an empty stack
- Recall the analogy of Ch. 10 between the stack data structure and a stack of plates in a cafeteria
- When we push a plate on the stack, we use 1 TL to pay the actual cost of the push and are left with a credit of 1 TL (out of the 2 TL charged), which we can put on top of the plate
- At any point in time, every plate on the stack has a lira of credit on it

Accounting Method: Stack Operations

- The lira stored on the plate is prepayment for the cost of popping it from the stack
- When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack
- To pop a plate, we take the lira of credit off the plate and use it to pay the actual cost of the operation
- Thus, by charging the PUSH operation a little bit more, we do not need to charge the POP operation anything

Accounting Method: Stack Operations

- We do not need to charge MULTIPOP operations anything either
- To pop the first plate, we take the lira of credit off the plate and use it to pay the actual cost of the POP operation
- To pop a second plate, we again have a lira of credit on the plate to pay for the POP operation, and so on
- Thus, we have always charged enough up front to pay for MULTIPOP operations

Accounting Method: Stack Operations

- In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative
- Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost
- Since the total amortized cost is O(n), so is the actual cost

Accounting Method: Incrementing a Binary Counter

- Example: Analyze the INCREMENT operation on a binary counter that starts at zero
- As we observed earlier, running time of this operation is proportional to the number of bits flipped, which we will use as our cost for this example
- Let us once again use a lira coin to represent each unit of cost (the flipping of a bit in this example)

Accounting Method: Incrementing a Binary Counter

- Example: Analyze the INCREMENT operation on a binary counter that starts at zero
- For the amortized analysis, let us charge an amortized cost of 2 liras to set a bit to 1
- When a bit is set, we use 1 lira (out of the 2 liras charged) to pay for the actual setting of the bit, and we place the other lira on the bit as credit to be used later when we flip the bit back to 0
- At any point in time, every 1 in the counter has a lira of credit on it, and thus we do not need to charge anything to reset a bit to 0; we just pay for the reset with the lira coin on the bit

Accounting Method: Incrementing a Binary Counter

- Example: Analyze the INCREMENT operation on a binary counter that starts at zero
- The amortized cost of INCREMENT can now be determined
- The cost of resetting the bits within the **while** loop is paid for by the liras on the bits that are reset
- At most one bit is set, in line 6 of INCREMENT, and therefore the amortized cost of an INCREMENT operation is at most 2 liras
- The number of 1's in the counter is never negative, and thus the amount of credit is always nonnegative
- Thus, for n INCREMENT operations, the total amortized cost is O(n), which bounds the total actual cost.

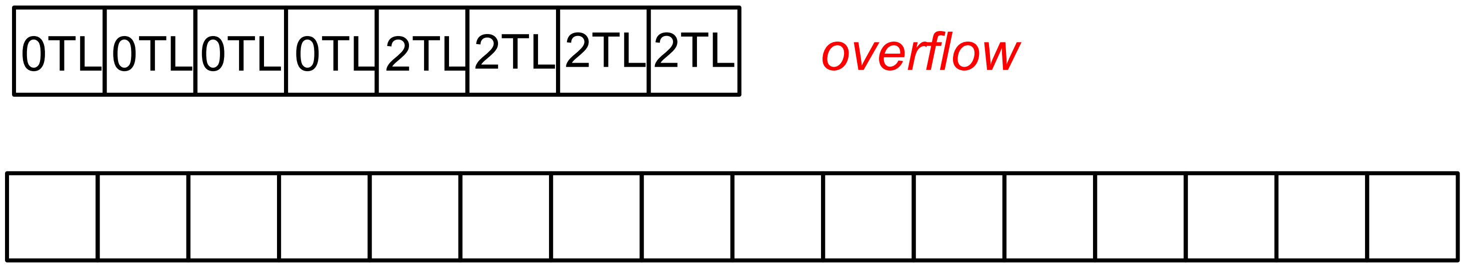
Accounting Analysis of Dynamic Tables

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

- 1TL pays for the immediate insertion
- 2TL is stored for later table doubling

When the table doubles, 1TL pays to move a recent item, and 1TL pays to move an old item

Example:



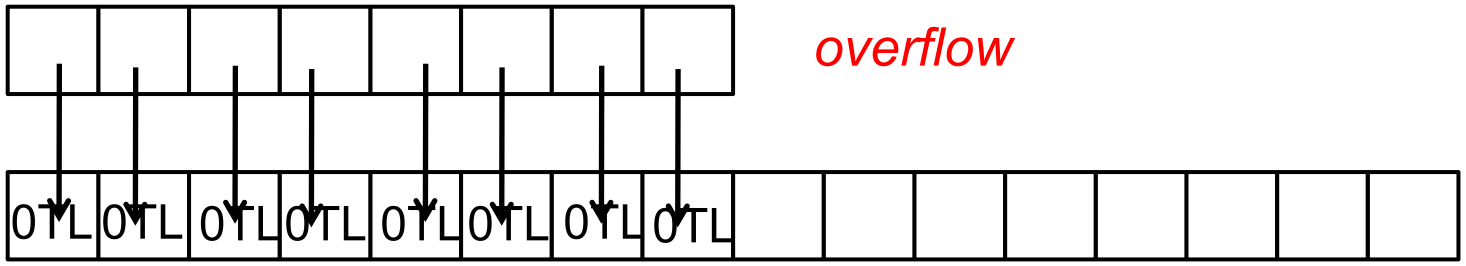
Accounting Analysis of Dynamic Tables

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

- 1TL pays for the immediate insertion
- 2TL is stored for later table doubling

When the table doubles, 1TL pays to move a recent item, and 1TL pays to move an old item

Example:



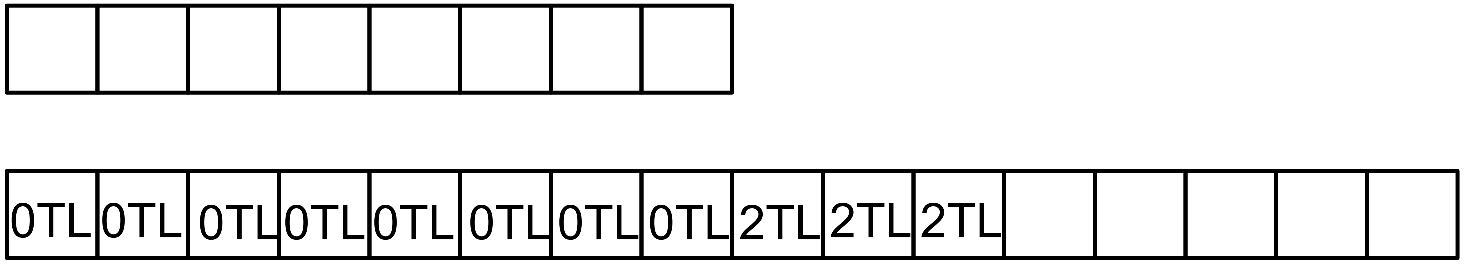
Accounting Analysis of Dynamic Tables

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

- 1TL pays for the immediate insertion
- 2TL is stored for later table doubling

When the table doubles, 1TL pays to move a recent item, and 1TL pays to move an old item

Example:



Accounting Analysis of Dynamic Tables (cont.)

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 actually, the first operation costs only \$2, not \$3

Potential Method

Potential Method

- Instead of representing prepaid work as credit stored with specific objects in the data structure
  - potential method of amortized analysis represents the prepaid work as "potential energy," or just "potential," that can be released to pay for future operations
- Potential is associated with data structure as a whole rather than with specific objects within data structure

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
- Cost of operation i is c_i
- Define a ***potential function*** $\Phi: \{D_i\} \to \mathbb{R}$, such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i
- ***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

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

- 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 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 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 True Costs

The total amortized cost of *n* operations is

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

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i \quad \text{since} \quad \Phi(D_n) \geq 0 \quad \text{and} \quad \Phi(D_0) = 0$$

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

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

Example:

●	●	●	●	●	●		

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

(

0TL	0TL	0TL	0TL	2TL	2TL		

accounting method)

Calculation of Amortized Costs

The amortized cost of the *i* th insertion is

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

Calculation of Amortized Costs

The amortized cost of the *i* th insertion is

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

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

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

Calculation of Amortized Costs

The amortized cost of the *i* th insertion is

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

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

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

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

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

Calculation

Case 1: *i* –1 is an exact power of 2

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

Calculation

Case 1: *i* –1 is an exact power of 2

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

Calculation

Case 1: *i* –1 is an exact power of 2

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

Calculation

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

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

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

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

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

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

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 Θ(*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

Summary

- 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

Summary

- Dynamic Tables
- Aggregate Analysis
- The Accounting Method
- The Potential Method