Amortized Analysis

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

For Algorithm Course

Outline

Basic Concepts

Motivation: given a **sequence** of operations, majority are cheap, but some rare might be expensive; thus a standard worst-case analysis might be overly pessimistic.

Basic idea: the cost of expensive operations can be “spread out” (amortized) to all operations. If the artificial amortized costs are still cheap, we get a tighter bound overall.

Amortized Analysis: A strategy to give a **tighter bound evenly** for a sequence of operations under **worst case** scenario.

Example: serving coffee in a bar

Amortized Analysis versus Average-Case Analysis

Amortized analysis differs from average-case analysis in:

Average-case analysis: average over all input, e.g., INSERTIONSORT algorithm performs well on “average” over all possible input even if it performs very badly on certain input.

Amortized analysis: average over operations, e.g., TABLEINSERTION algorithm performs well on “average” over all operations even if some operations use a lot of time.

- Probability is not involved;
- Guarantees the average performance of each operation in the worst case.

Types of Amortized Analyses

There are three common amortization arguments:

Aggregate Analysis: determine an upper bound $T(n)$ on the total cost of a sequence of n operations, and the average cost per operation is then $T(n)/n$ (referred as *amortized cost*).

Accounting Method: determine an amortized cost of each operation, different cost for different operations. Store "prepaid credit" for overcharge at early stage and pay for operations later in the sequence.

Potential Method: determine costs for operations, and maintain credit as the "potential energy" as a whole instead of associating the credit within individual objects.

Examples

Through out this lecture, we will continuously use three examples to illustrate the amortized methods:

Stack Operations: Push and pop elements from an empty stack;

Binary Counter: Count a series of numbers by binary flip flops;

Dynamic Table: A continuous storage array that could change size dynamically.

First Method: Aggregate Analysis

Compute the worst time $T(n)$ in total for a sequence of n operations. The *amortized cost* (average cost) per operation is $T(n)/n$ in the worst case.

- Cost $T(n)/n$ applies to each operation (There may be several types of operations)
- The other two methods may assign different amortized costs to different types of operation.

Example: Stack with Multipop Operations

There are two fundamental stack operations, each takes $O(1)$ time:

PUSH(S, x): push object x onto stack S.
POP(S): pop the top of stack S and returns the popped object.

Assign cost for each operation as **1**.

Time Complexity: The total cost of a sequence of n PUSH and POP operations is n, and the actual running time for n operations is $\Theta(n)$.

Multipop Operation

Now we add an additional stack operation MULTIPOP.

MULTIPOP(S, k): pop k top objects of stack S (or pop entire stack if it contains fewer than k objects).

ALGORITHM 1: MULTIPOP(S, k)

```
while S is not empty and k > 0 do
    POP (S);
    k ← k − 1;
```

The total cost of MULTIPOP is $\min\{|S|, k\}$.

A Sequence of Operations

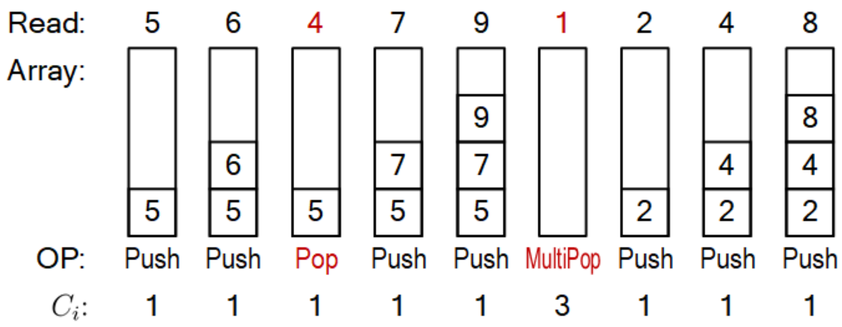
Consider a sequence of n POP, PUSH, and MULTIPOP operations on an initially empty stack.

ALGORITHM 1: Stack with MULTIPOP

Input : An array $A[1..n]$ of n elements and an integer k.
Output: Stack S.

```
for i = 1 to n do
    if A[i] ≥ A[i − 1] then
        PUSH(S, A[i]);
    else if A[i] ≤ A[i − 1] − k then
        MULTIPOP(S, k);
    else
        POP(S);
```


An Example Scenario



Cursory analysis: MULTIPOP(S, k) may take $O(n)$ time; thus,

$$T(n) = \sum_{i=1}^{n} C_i \leq n^2.$$

Cursory Analysis versus Tighter Analysis

In a sequence of operations, some operations may be cheap, but some operations may be expensive, say MULTIPOP(S, k).

However, the worst operation does not occur often. Therefore, the traditional worst-case *individual operation* analysis can give overly pessimistic bound.

Objective: For each operation we hope to assign an amortized cost $\widehat{C}_i$ to bound the actual total cost.

For any sequence of n operations, we have

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i.$$

Here, C_i denotes the actual cost of step i.

Tighter Analysis: Aggregate Technique

Basic idea: all operations have the same amortized cost $\frac{1}{n}\sum_{i=1}^{n}\widehat{C}_i$

Key observation: $\#Pop \le \#Push$; Thus, we have:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= \#Push + \#Pop \\ &\le 2 \times \#Push \\ &\le 2n \end{aligned}$$

Conclusion: on average, the MULTIPOP(S, k) step takes only $O(1)$ time rather than $O(k)$ time.

Another Example: Incrementing a Binary Counter

Consider a k-bit binary counter that counts upward from 0.

Use array $A[0, \cdots, k-1]$ of bits to record the count number.

A binary number x stored in the counter has its lowest-order bit in $A[0]$ and highest-order bit in $A[k-1]$, and

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

Initially, $x = 0$, $A[i] = 0$ for $i = 0, \cdots, k-1$.

An Example Scenario

Counter Value	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	Cost	Total Cost
0	0	0	0	0	0	0	0	0	**0**	**0**
1	0	0	0	0	0	0	0	1	**1**	**1**
2	0	0	0	0	0	0	1	0	**2**	**3**
3	0	0	0	0	0	0	1	1	**1**	**4**
4	0	0	0	0	0	1	0	0	**3**	**7**
5	0	0	0	0	0	1	0	1	**1**	**8**
6	0	0	0	0	0	1	1	0	**2**	**10**
7	0	0	0	0	0	1	1	1	**1**	**11**
8	0	0	0	0	1	0	0	0	**4**	**15**
9	0	0	0	0	1	0	0	1	**1**	**16**
10	0	0	0	0	1	0	1	0	**2**	**18**
11	0	0	0	0	1	0	1	1	**1**	**19**
12	0	0	0	0	1	1	0	0	**3**	**22**

Pseudo Code for Binary Counter

INCREMENT is used to add 1 (modulo 2^k) to the value in the counter.

ALGORITHM 3: INCREMENT(A)

```
i ← 0;
while  i ≤ k − 1 and A[i] = 1 do
    A[i] ← 0;
    i ← i + 1;
if i ≤ k − 1 then
    A[i] ← 1;
```

Consider a sequence of n operations that counts upward from 0:

ALGORITHM 4: BINARYCOUNTER

```
for i = 1 to n do
    INCREMENT(A);
```

Tighter Analysis: Aggregate Technique

Question: $T(n) \leq ?$

Cursory analysis: $T(n) \leq kn$ since an increment step might change all k bits.

Aggregate analysis: Basic operations: `flip(1→0)`, `flip(0→1)`

During a sequence of n INCREMENT operations:

$A[0]$ flips each time INCREMENT is called $\leftarrow n$ times;

$A[1]$ flips every other time $\leftarrow \lfloor n/2 \rfloor$ times;

$\cdots\ \cdots\ \cdots$

$A[i]$ flips $\lfloor n/2^i \rfloor$ times.

Tighter Analysis: Aggregate Technique (Cont.)

Thus,

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= 1+2+1+3+1+2+1+4+\cdots && \text{(add by row)} \\ &= \#flip(A[0]) + \#flip(A[1]) + \cdots + \#flip(A[k]) && \text{(add by column)} \\ &= n + \frac{n}{2} + \frac{n}{4} + \cdots \\ &\leq 2n \end{aligned}$$

Amortized cost of each operation: $O(n)/n = O(1)$.

Accounting Method

Basic idea: for each operation OP with actual cost C_{OP}, an amortized cost $\widehat{C_{OP}}$ is assigned such that for any sequence of n operations,

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}.$$

Intuition: If $\widehat{C_{op}} > C_{op}$, the overcharge will be stored as prepaid credit; the credit will be used later for the operations with $\widehat{C_{op}} < C_{op}$.

The requirement that $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i}$ is essentially credit never goes negative.

Example 1: Stack with MULTIPOP Operation

Example: For stack with MULTIPOP, assign amortized cost as:

Operation	Real Cost C_{op}	Amortized Cost $\widehat{C_{op}}$
PUSH	1	2
POP	1	0
MULTIPOP	$\min\{\lvert S\rvert, k\}$	0

Credit: the number of items in the stack.

Starting from an empty stack, any sequence of n_1 PUSH, n_2 POP, and n_3 MULTIPOP operations takes at most $T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C_i} = 2n_1$. Here $n = n_1 + n_2 + n_3$.

Note: when there are more than one type of operations, each type of operation might be assigned with different amortized cost.

Accounting Method: “Banker’s View”

Suppose you are renting a **"coin-operation"** machine, and are charged according to the number of operations.

Two payment strategies:

- Pay actual cost for each operation:
  say pay \$1 for PUSH, \$1 for POP, and \$$k$ for MULTIPOP.
- Open an account, and pay “average” cost for each operation:
  say pay \$2 for PUSH, \$0 for POP, and \$0 for MULTIPOP.

If “average” cost $>$ actual cost: the extra will be deposited as *credit*.

If “average” cost $<$ actual cost: credit will be used to pay actual cost.

Constraint: $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i$ for arbitrary n operations, i.e. you have enough credit in your account.

An Example Scenario

Read:	5	6	4	7	9	1	2	4	8
Array:	5	6 5	5	7 5	9 7 5		2	4 2	8 4 2
OP:	Push	Push	Pop	Push	Push	MultiPop	Push	Push	Push
C_i:	1	1	1	1	1	3	1	1	1
$\widehat{C}_i$:	2	2	0	2	2	0	2	2	2
Credit:	1	2	1	2	3	0	1	2	3

Example 2: Incrementing Binary Counter

Set amortized cost as follows:

OP	Real Cost C_{OP}	Amortized Cost $\widehat{C_{OP}}$
`flip(0→1)`	1	2
`flip(1→0)`	1	0

Key observation: $\#\mathit{flip}(0 \to 1) \geq \#\mathit{flip}(1 \to 0)$

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= \#\mathit{flip}(0 \to 1) + \#\mathit{flip}(1 \to 0) \\ &\leq 2\#\mathit{flip}(0 \to 1) \\ &\leq 2n \end{aligned}$$

Potential Technique: “Physicist’s View”

Basic idea: sometimes it is not easy to set $\widehat{C_{op}}$ for each operation OP directly.

Define a potential function as a bridge, i.e. we can assign a value to state rather than operation, and amortized costs are then calculated based on potential function.

Potential Function: $\Phi(S) : S \to R$, where S is state collection.

Amortized Cost Setting: $\widehat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1})$.

Potential Technique: "Physicist's View" (Cont.)

Then we have

$$\begin{aligned}\sum_{i=1}^{n}\widehat{C}_i &= \sum_{i=1}^{n}(C_i + \Phi(S_i) - \Phi(S_{i-1})) \\ &= \sum_{i=1}^{n} C_i + \Phi(S_n) - \Phi(S_0)\end{aligned}$$

Requirement: To guarantee $\sum_{i=1}^{n} C_i \le \sum_{i=1}^{n}\widehat{C}_i$, it suffices to assure

$\Phi(S_n) \ge \Phi(S_0)$.

Stack Example: Potential Changes

Potential Function: Let $\Phi(S)$ denote the number of items in stack.

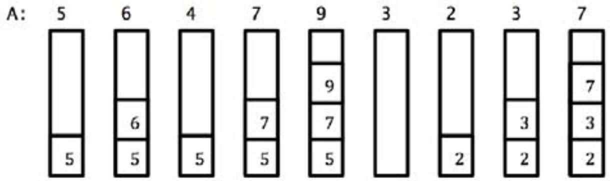
In fact, we simply use "credit" as potential.

State: Here state S_i refers to the STATE of the stack after the i-th operation.

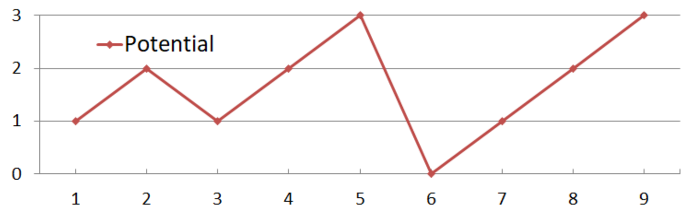
Correctness: $\Phi(S_i) \ge 0 = \Phi(S_0)$ for any i;

An Example Scenario

States of Stack S:



Polyline of Potential Function $\Phi(S_i)$:



Potential Function Technique: Amortized Cost Setting

Definition: $\Phi(S)$ denotes the number of items in stack;

PUSH: $\Phi(S_i) - \Phi(S_{i-1}) = 1$
$\widehat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 2$

POP: $\Phi(S_i) - \Phi(S_{i-1}) = -1$
$\widehat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 0$

MULTIPOP: $\Phi(S_i) - \Phi(S_{i-1}) = -\#Pop$
$\widehat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1}) = 0$

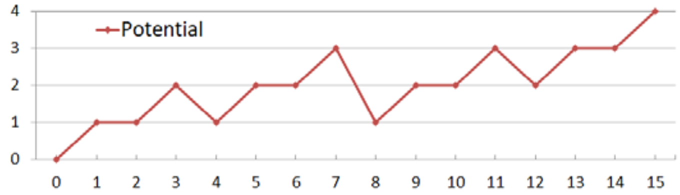
Thus, starting from an empty stack, any sequence of n_1 PUSH, n_2 POP, and n_3 MULTIPOP operations takes at most $T(n) = \sum_{i=1}^{n} C_i \le \sum_{i=1}^{n}\widehat{C}_i = 2n_1$. Here $n = n_1 + n_2 + n_3$.

Binary Counter

Definition: Set potential function as $\Phi(S) = \#1$ in counter

Counter Value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Cost	Total Cost
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	0	2	3
3	0	0	0	0	0	0	1	1	1	4
4	0	0	0	0	0	1	0	0	3	7
5	0	0	0	0	0	1	0	1	1	8
6	0	0	0	0	0	1	1	0	2	10
7	0	0	0	0	0	1	1	1	1	11
8	0	0	0	0	1	0	0	0	4	15

Polyline of Potential Function $\Phi(S)$:



Binary Counter (Cont.)

Definition: Set potential function as $\Phi(S) = \#1$ in counter;

At step i, the number of flips C_i is:

$$\begin{aligned}
C_i &= \#\mathit{flip}^{(i)}_{0\to1} + \#\mathit{flip}^{(i)}_{1\to0} = 1 + \#\mathit{flip}^{(i)}_{1\to0} \quad (\mathit{why?}) \\
\Phi(S_i) &= \Phi(S_{i-1}) + 1 - \#\mathit{flip}^{(i)}_{1\to0} \\
\widehat{C}_i &= C_i + \Phi(S_i) - \Phi(S_{i-1}) \\
&= 1 + \#\mathit{flip}^{(i)}_{1\to0} + 1 - \#\mathit{flip}^{(i)}_{1\to0} = 2
\end{aligned}$$

Thus we have

$$T(n) = \sum_{i=1}^{n} C_i \le \sum_{i=1}^{n} \widehat{C}_i = 2n$$

In other words, starting from 00....0, a sequence of n INCREMENT operations takes at most $2n$ time.

A Practical Problem

Suppose you are asked to develop a `C++` compiler.

`vector` is one of a `C++` class templates to hold a set of objects. It supports the following operations:

- `push_back`: to add a new object onto the tail;
- `pop_back`: to pop out the last object;

Recall that `vector` uses a **contiguous memory area** to store objects.

Question: How to design an efficient **memory-allocation strategy** for `vector`?

DYNAMICTABLE Problem

In many applications, we do not know in advance how many objects will be stored in a table.

Thus we have to allocate space for a table, only to find out later that it is not enough.

DYNAMIC EXPANSION: When inserting a new item into a full table, the table must be reallocated with a larger size, and the objects in the original table must be copied into the new table.

DYNAMIC CONTRACTION: Similarly, if many objects have been removed from a table, it is worthwhile to reallocate the table with a smaller size.

We will show a **memory allocation strategy** such that the amortized cost of insertion and deletion is $O(1)$, even if the actual cost of an operation is large when it triggers an expansion or contraction.

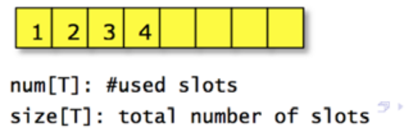
Table Expansion Operation

ALGORITHM 5: TABLE_INSERT(T, i)

```
if size[T] = 0 then
    allocate a table with 1 slot;
    size[T] = 1;
if num[T] = size[T] then
    allocate a new table with 2 × size[T] slots; //double size
    size[T] = 2 × size[T];
    copy all items into the new table;
    free the original table;
insert the new item i into T;
num[T] ← num[T] + 1;
```


Example: TABLEINSERT

An Example Dynamic Table T:



Consider a sequence of operations starting with an empty table:

ALGORITHM 6: TABLE_INSERT

```
Table T;
for i = 1 to n do
    TABLE_INSERT(T, i);
```


TABLEINSERT(1)

INSERT(1) | 1 | C_1=1

TABLEINSERT(2)

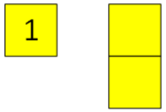
INSERT(1) | 1 | C_1=1

INSERT(2) | *overflow*

TABLEINSERT(2)

INSERT(1)

INSERT(2)

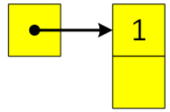


C_1=1

TABLEINSERT(2)

INSERT(1)

INSERT(2)

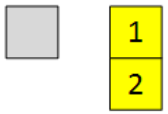


C_1=1

TABLEINSERT(2)

INSERT(1)

INSERT(2)



C_1=1

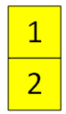
C_2=2

TABLEINSERT(3)

INSERT(1)

INSERT(2)

INSERT(3)



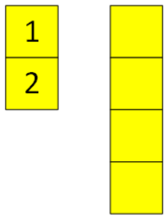
overflow

C_1=1

C_1=2

TableInsert(3)

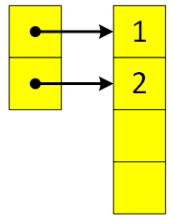
INSERT(1)
INSERT(2)
INSERT(3)



C_1=1
C_1=2

TableInsert(3)

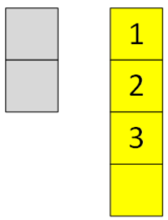
INSERT(1)
INSERT(2)
INSERT(3)



C_1=1
C_2=2

TableInsert(3)

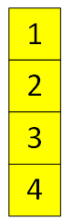
INSERT(1)
INSERT(2)
INSERT(3)



C_1=1
C_2=2
C_3=3

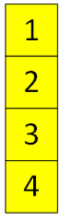
TableInsert(4)

INSERT(1)
INSERT(2)
INSERT(3)
INSERT(4)



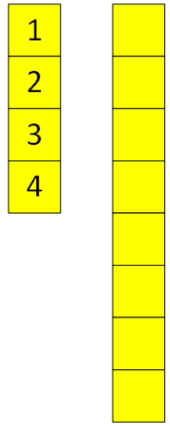
C_1=1
C_2=2
C_3=3
C_4=1

TABLEINSERT(5)



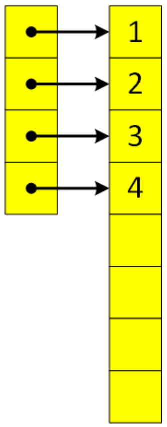
INSERT(1) C_1=1
INSERT(2) C_2=2
INSERT(3) C_3=3
INSERT(4) C_4=1
INSERT(5) *overflow*

TABLEINSERT(5)



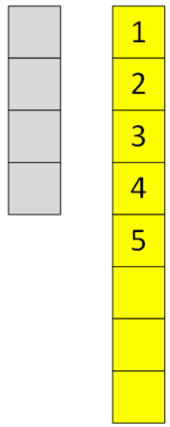
INSERT(1) C_1=1
INSERT(2) C_2=2
INSERT(3) C_3=3
INSERT(4) C_4=1
INSERT(5)

TABLEINSERT(5)



INSERT(1) C_1=1
INSERT(2) C_2=2
INSERT(3) C_3=3
INSERT(4) C_4=1
INSERT(5)

TABLEINSERT(5)



INSERT(1) C_1=1
INSERT(2) C_2=2
INSERT(3) C_3=3
INSERT(4) C_4=1
INSERT(5) C_5=5

TableInsert(6)

INSERT(1)	1	$C_1=1$
INSERT(2)	2	$C_2=2$
INSERT(3)	3	$C_3=3$
INSERT(4)	4	$C_4=1$
INSERT(5)	5	$C_5=5$
INSERT(6)	6	$C_6=1$

TableInsert(7)

INSERT(1)	1	$C_1=1$
INSERT(2)	2	$C_2=2$
INSERT(3)	3	$C_3=3$
INSERT(4)	4	$C_4=1$
INSERT(5)	5	$C_5=5$
INSERT(6)	6	$C_6=1$
INSERT(7)	7	$C_7=1$

TableInsert(8)

INSERT(1)	1	$C_1=1$
INSERT(2)	2	$C_2=2$
INSERT(3)	3	$C_3=3$
INSERT(4)	4	$C_4=1$
INSERT(5)	5	$C_5=5$
INSERT(6)	6	$C_6=1$
INSERT(7)	7	$C_7=1$
INSERT(8)	8	$C_8=1$

Cursory analysis: $O(n^2)$

Consider a sequence of operations starting with an empty table. Define C_i as the cost of the ith operation (elementary insertions or deletions),

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

Here $C_i = i$ when the table is full, since we need to perform 1 insertion, and copy $i-1$ items into the new table.

If n operations are performed, the worst-case cost of an operation will be $O(n)$. Thus, the total running time is $O(n^2)$. **Not tight!**

Tighter Analysis 1: Aggregate Method

Key Observation: **Table expansions are rare**.

The $O(n^2)$ bound is not tight since **table expansion** doesn't occur often in the course of n operations.

Specifically, **table expansion** occurs at the ith operation, where $i-1$ is an exact power of 2.

i	1	2	3	4	5	6	7	8	9	10	11	12
$Size_i$	1	2	4	4	8	8	8	8	16	16	16	16
C_i	1	2	3	1	5	1	1	1	9	1	1	1

We can decompose C_i as follows:

i	1	2	3	4	5	6	7	8	9	10	11	12
$Size_i$	1	2	4	4	8	8	8	8	16	16	16	16
C_i (insert)	1	1	1	1	1	1	1	1	1	1	1	1
C_i (copy)		1	2		4				8			

Total cost of n operations

The total cost of n operations is:

$$\begin{aligned}\sum_{i=1}^{n} C_i &= 1+2+3+1+5+1+1+1+9+1+\ldots \\ &= n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

Thus the amortized cost of an operation is 3.

In other words, the average cost of each TABLEINSERT operation is $O(n)/n = O(1)$.

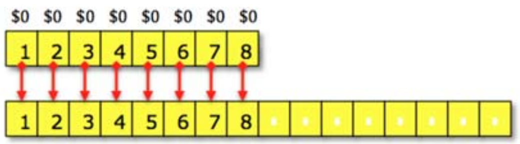
Tighter Analysis 2: Accounting Technique

For the i-th operation, an **amortized cost** $\widehat{C}_i = \$3$ is charged.

- \$1 pays for the insertion **itself**;
- \$2 is stored for **later table doubling**, \$1 for copying one of the recent $\frac{i}{2}$ items, \$1 for copying one of the old $\frac{i}{2}$ items.

Original:

Expansion:



Tighter Analysis 2: Accounting Technique

Key observation: the credit never goes negative. In other words, the sum of amortized cost provides an upper bound of the sum of actual costs.

$$T(n) = \sum_{i=1}^{n} C_i = \leq \sum_{i=1}^{n} \widehat{C}_i = 3n.$$

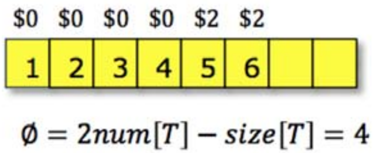
i	1	2	3	4	5	6	7	8	9	10	11	12
$Size_i$	1	2	4	4	8	8	8	8	16	16	16	16
C_i (insert)	1	1	1	1	1	1	1	1	1	1	1	1
C_i (copy)		1	2		4				8			
$\widehat{C}_i$	3	3	3	3	3	3	3	3	3	3	3	3
Credit	2	3	3	5	3	5	7	9	3	5	7	9

Tighter Analysis 3: Potential Function Technique

Basic idea: the **bank account** can be viewed as potential function of the dynamic set. More specifically, we prefer a potential function $\Phi : \{T\} \to R$ with the following properties:

- $\Phi(T) = 0$ immediately **after** an expansion;
- $\Phi(T) = size[T]$ immediately **before** an expansion; thus, the next expansion can be paid for by the potential.

A possibility: $\Phi(T) = 2 \times num[T] - size[T]$



$\Phi(T) = 2 \times num[T] - size[T]$: An Example

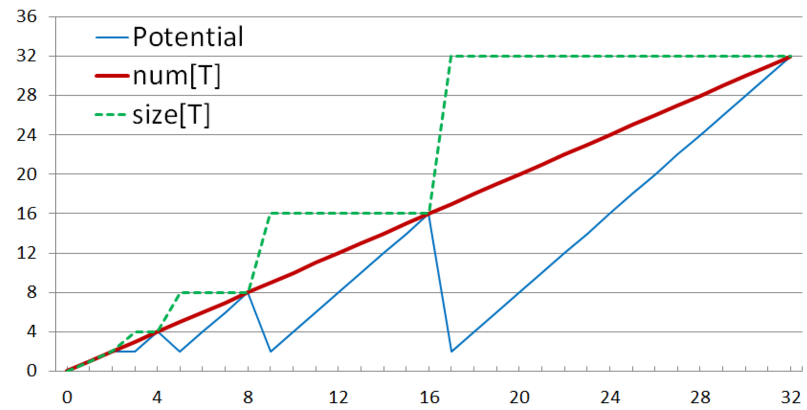


Figure: The effect of a sequence of n TABLEINSERT on $size_i$ (green), num_i (red), and Φ_i (blue).

Correctness of $\Phi(T) = 2 \times num[T] - size[T]$

Correctness: Initially $\Phi_0 = 0$, and it is easy to verify that $\Phi_i \geq \Phi_0$ since the table is always at least half full.

The **amortized cost** $\widehat{C}_i$ with respect to Φ is defined as:

$$\widehat{C}_i = C_i + \Phi(T_i) - \Phi(T_{i-1}).$$

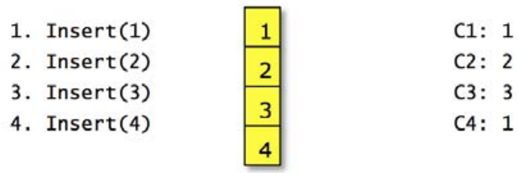
Thus $\sum_{i=1}^{n} \widehat{C}_i = \sum_{i=1}^{n} C_i + \Phi_n - \Phi_0$ is really an upper bound of the actual cost $\sum_{i=1}^{n} C_i$.

Calculate $\widehat{C}_i$ with respect to Φ

Case 1: the i-th insertion does not trigger an expansion

$size_i = size_{i-1}$ ($size_i$: the table size after the i-th operation.)
$num_i = num_{i-1} + 1$ (num_i: no. of items after the i-th operations)

$$\begin{aligned}
\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + 2 \\
&= 3
\end{aligned}$$



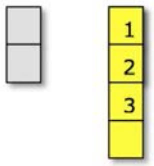
Calculate $\widehat{C}_i$ with respect to Φ

Case 2: the i-th insertion triggers an expansion

$$size_i = 2 \times size_{i-1}.$$
$$size_{i-1} = num_{i-1} = num_i - 1.$$

$$\begin{aligned}
\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= num_i + 2 - (num_i - 1) \\
&= 3
\end{aligned}$$

```
1. Insert(1)
2. Insert(2)
3. Insert(3)
```




```
C1: 1
C2: 2
C3: 3
```


TableDelete Operation

To implement TableDelete operation, it is simple to remove the specified item from the table, followed by a Contraction operation when the **load factor** (denoted as $\alpha(T) = \frac{num[T]}{size[T]}$) is small, so that the wasted space is not exorbitant.

Specifically, when the number of the items in the table drops too low, we allocate a new, smaller space, copy the items from the old table to the new one, and finally free the original table.

We would like the following two properties:

- The load factor is bounded below by a constant;
- The amortized cost of a table operation is bounded above by a constant.

Trial 1

Trial 1: load factor $\alpha(T)$ never drops below 1/2

A natural strategy is:

- To double the table size when inserting an item into a full table;
- To halve the table size when deletion causes $\alpha(T) < \frac{1}{2}$.

The strategy guarantees that load factor $\alpha(T)$ never drops below 1/2.

However, the amortized cost of an operation might be quite large.

An Example of Large Amortized Cost

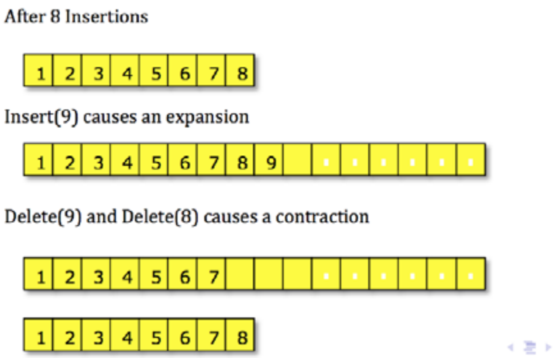
Consider a sequence of $n = 16$ operations:

- The first 8 operations: `I, I, I, ...`.
- The second 8 operations: `I, D, D, I, I, D, D, I`
- Repeat the `I, D, D, I` opertions $\cdots\cdots$

Note:

- After the 8-th `I`, we have $num_8 = size_8 = 8$.
- The 9-th `I` leads to a table expansion;
- The following two `D` lead to a table contraction;
- The following two `I` lead to a table expansion, and so on.

An Example of Large Amortized Cost



The expansion/contraction takes $O(n)$ time, and there are n of them.

Thus the total cost of n operations are $O(n^2)$, and the amortized cost of an operation is $O(n)$.

Trial 2

Trial 2: load factor $\alpha(T)$ never drops below 1/4

Another strategy is:

- To double the table size when inserting an item into a full table;
- To halve the table size when deletion causes $\alpha(T) < \frac{1}{4}$.

The strategy guarantees that load factor $\alpha(T)$ never drops below 1/4.

Amortized Analysis

We start by defining a potential function $\Phi(T)$ that is 0 immediately after an expansion or contraction, and builds as $\alpha(T)$ increases to 1 or decreases to $\frac{1}{4}$.

$$\Phi(T) = \begin{cases} 2 \times num[T] - size[T] & \texttt{if} \ \ \alpha(T) \geq \frac{1}{2} \\ \frac{1}{2} size[T] - num[T] & \texttt{if} \ \ \alpha(T) < \frac{1}{2} \end{cases}$$

Correctness: the potential is 0 for an empty table, and $\Phi(T)$ never goes negative. Thus, the total amortized cost of a sequence of n operations with respect to Φ is an upper bound of the actual cost.

Amortized Cost of TABLEINSERT

Case 1: $\alpha_{i-1} \geq \frac{1}{2}$ and no expansion

The amortized cost is:

$$\begin{aligned} \widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (2num_{i-1} - size_{i-1}) \\ &= 3 \end{aligned}$$

```
1. Insert(1)
2. Insert(2)
3. Insert(3)
4. Insert(4)
```

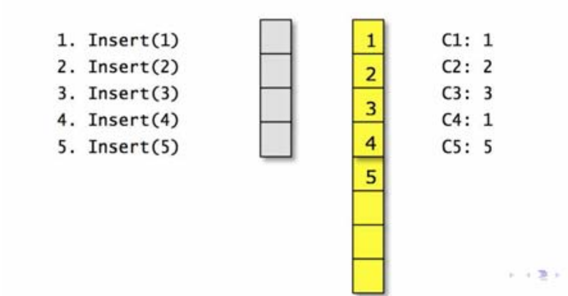

```
C1: 1
C2: 2
C3: 3
C4: 1
```

Amortized Cost of TABLEINSERT

Case 2: $\alpha_{i-1} \geq \frac{1}{2}$ and an expansion was triggered

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= num_{i-1} + 1 + (2(num_{i-1}+1) - 2size_{i-1}) - (2num_{i-1} - size_{i-1}) \\ &= 3 + num_{i-1} - size_{i-1} \quad \leftarrow num_{i-1} = size_{i-1} \\ &= 3\end{aligned}$$

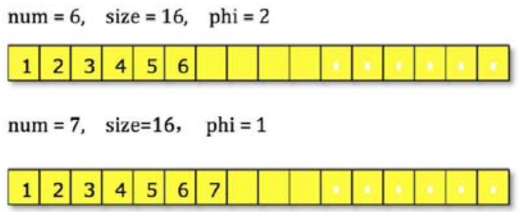


Amortized Cost of TABLEINSERT

Case 3: $\alpha_{i-1} < \frac{1}{2}$ and $\alpha_i < \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_i - (num_i - 1)) \\ &= 0\end{aligned}$$

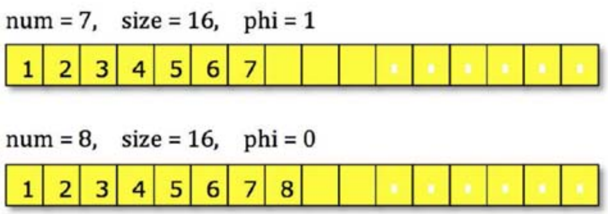


Amortized Cost of TABLEINSERT

Case 4: $\alpha_{i-1} < \frac{1}{2}$ but $\alpha_i \geq \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + (2num_i - size_i) - (\frac{1}{2}size_i - (num_i - 1)) \\ &= 1 + 0 - 1 = 0 \quad \leftarrow size_i = 2num_i\end{aligned}$$

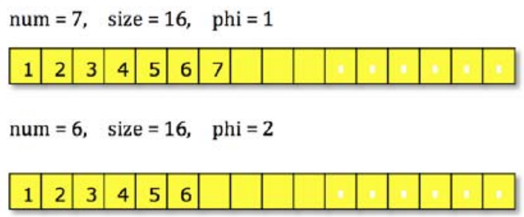


Amortized Cost of TABLEDELETE

Case 1: $\alpha_{i-1} < \frac{1}{2}$ and no contraction

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + (\frac{1}{2}size_{i-1} - (num_{i-1} - 1)) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 2\end{aligned}$$



Amortized Cost of TABLEDELETE

Case 2: $\alpha_{i-1} < \frac{1}{2}$ and a contraction was triggered

The amortized cost is:

$$\begin{aligned}
\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= num_i + 1 + (\tfrac{1}{2}size_i - num_i) - (\tfrac{1}{2}size_{i-1} - num_{i-1}) \\
&= num_{i-1} + (\tfrac{1}{4}size_{i-1} - (num_{i-1} - 1)) - (\tfrac{1}{2}size_{i-1} - num_{i-1}) \\
&= 1 + num_{i-1} - \tfrac{1}{4}size_{i-1} \qquad \leftarrow num_{i-1} = \tfrac{1}{4}size_{i-1} \\
&= 1
\end{aligned}$$

num=4, size=16, phi=4

1	2	3	4												

num=3, size=8, phi=1

1	2	3					

Amortized Cost of TABLEDELETE

Case 3: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i \geq \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}
\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + (2(num_{i-1} - 1) - size_{i-1}) - (2num_{i-1} - size_{i-1}) \\
&= -1
\end{aligned}$$

num = 10, size = 16, phi = 4

1	2	3	4	5	6	7	8	9	10						

num = 9, size = 16, phi = 2

1	2	3	4	5	6	7	8	9							

Amortized Cost of TABLEDELETE

Case 4: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i < \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}
\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (\tfrac{1}{2}size_i - num_i) - (2num_{i-1} - size_{i-1}) \\
&= 1 + (\tfrac{1}{2}size_{i-1} - (num_{i-1} - 1)) - (2num_{i-1} - size_{i-1}) \\
&= 2 + \tfrac{3}{2}size_{i-1} - 3num_{i-1} \\
&= 2 \qquad \leftarrow size_{i-1} = 2num_{i-1}
\end{aligned}$$

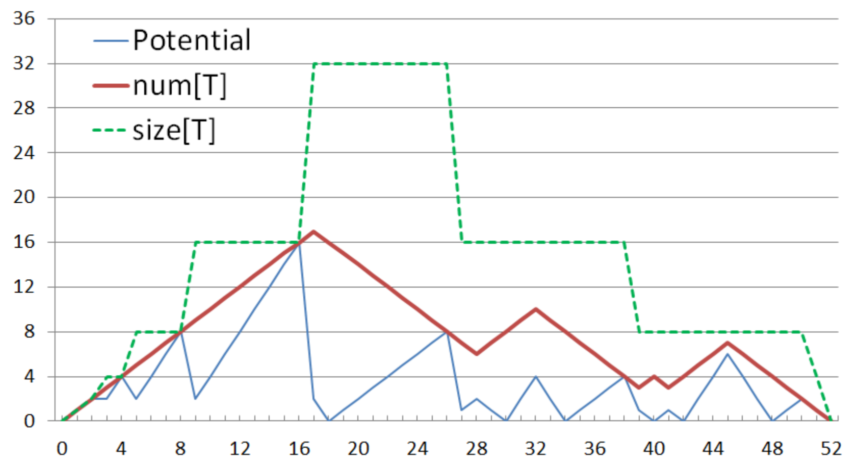
num = 8, size = 16, phi = 0

1	2	3	4	5	6	7	8								

num = 7, size = 16, phi = 1

1	2	3	4	5	6	7									

An Example Polyline of Φ_i



Conclusion

Since the amortized cost of each operation is bounded above by a constant, Starting with an empty table:

- a sequence of n TABLEINSERT operations cost $O(n)$ time in the worst case.
- the actual cost of any sequence of n TABLEINSERT and TABLEDELETE operations is still $O(n)$ in the worst case.

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.

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