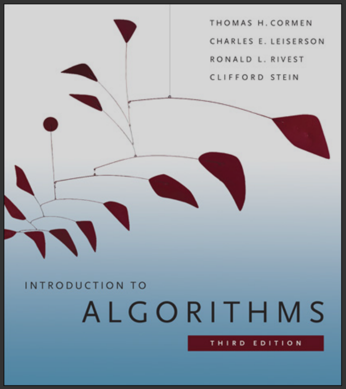


DATA STRUCTURES I, II, III, AND IV

I. *Amortized Analysis*
II. *Binary and Binomial Heaps*
III. *Fibonacci Heaps*
IV. *Union–Find*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

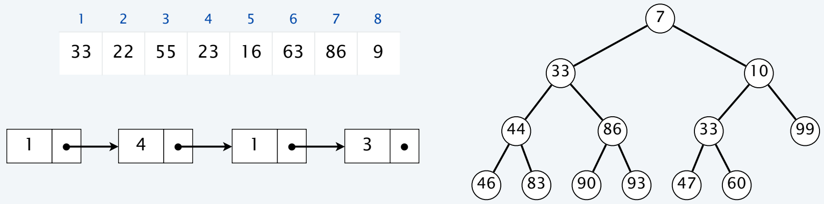
Last updated on 1/5/22 12:37 PM

Data structures

Static problems. Given an input, produce an output.
Ex. Sorting, FFT, edit distance, shortest paths, MST, max-flow, ...

Dynamic problems. Given a sequence of operations (given one at a time), produce a sequence of outputs.
Ex. Stack, queue, priority queue, symbol table, union–find, ...

Algorithm. Step-by-step procedure to solve a problem.
Data structure. Way to store and organize data.
Ex. Array, linked list, binary heap, binary search tree, hash table, ...



Appetizer

Goal. Design a data structure to support all operations in $O(1)$ time.

- INIT(n): create and return an initialized array (all zero) of length n.
- READ(A, i): return element i in array.
- WRITE($A, i, value$): set element i in array to $value$.

Assumptions.

true in C or C++, but not Java

- Can MALLOC an uninitialized array of length n in $O(1)$ time.
- Given an array, can read or write element i in $O(1)$ time.

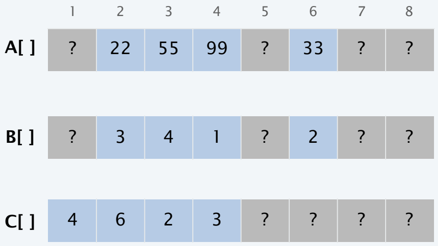
Remark. An array does INIT in $\Theta(n)$ time and READ and WRITE in $\Theta(1)$ time.

Appetizer

Data structure. Three arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k.

- $A[i]$ stores the current value for READ (if initialized).
- k = number of initialized entries.
- $C[j]$ = index of j^{th} initialized element for $j = 1, \ldots, k$.
- If $C[j] = i$, then $B[i] = j$ for $j = 1, \ldots, k$.

Theorem. $A[i]$ is initialized iff both $1 \le B[i] \le k$ and $C[B[i]] = i$.
Pf. Ahead.



	1	2	3	4	5	6	7	8
A[ ]	?	22	55	99	?	33	?	?
B[ ]	?	3	4	1	?	2	?	?
C[ ]	4	6	2	3	?	?	?	?

k = 4

A[4]=99, A[6]=33, A[2]=22, and A[3]=55 initialized in that order

Appetizer

```
INIT $(A, n)$
    $k \leftarrow 0$.
    $A \leftarrow$ MALLOC$(n)$.
    $B \leftarrow$ MALLOC$(n)$.
    $C \leftarrow$ MALLOC$(n)$.
```

```
READ $(A, i)$
    IF (IS-INITIALIZED $(A[i])$)
        RETURN $A[i]$.
    ELSE
        RETURN 0.
```

```
WRITE $(A, i, value)$
    IF (IS-INITIALIZED $(A[i])$)
        $A[i] \leftarrow value$.
    ELSE
        $k \leftarrow k + 1$.
        $A[i] \leftarrow value$.
        $B[i] \leftarrow k$.
        $C[k] \leftarrow i$.
```

```
IS-INITIALIZED $(A, i)$
    IF $(1 \le B[i] \le k)$ and $(C[B[i]] = i)$
        RETURN $true$.
    ELSE
        RETURN $false$.
```

Appetizer

Theorem. $A[i]$ is initialized iff both $1 \le B[i] \le k$ and $C[B[i]] = i$.

Pf. ⇒

- Suppose $A[i]$ is the j^{th} entry to be initialized.
- Then $C[j] = i$ and $B[i] = j$.
- Thus, $C[B[i]] = i$.

	1	2	3	4	5	6	7	8
A[ ]	?	22	55	99	?	33	?	?
B[ ]	?	3	4	1	?	2	?	?
C[ ]	4	6	2	3	?	?	?	?

k = 4

A[4]=99, A[6]=33, A[2]=22, and A[3]=55 initialized in that order

---

Appetizer

Theorem. $A[i]$ is initialized iff both $1 \le B[i] \le k$ and $C[B[i]] = i$.

Pf. ⇐

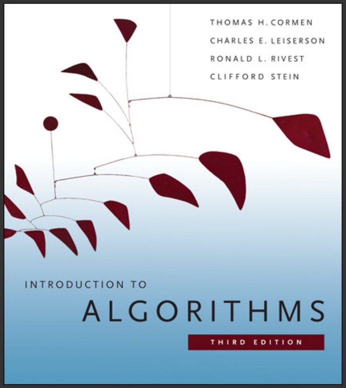
- Suppose $A[i]$ is uninitialized.
- If $B[i] < 1$ or $B[i] > k$, then $A[i]$ clearly uninitialized.
- If $1 \le B[i] \le k$ by coincidence, then we still can't have $C[B[i]] = i$ because none of the entries $C[1..k]$ can equal i. ▪

	1	2	3	4	5	6	7	8
A[ ]	?	22	55	99	?	33	?	?
B[ ]	?	3	4	1	?	2	?	?
C[ ]	4	6	2	3	?	?	?	?

k = 4

A[4]=99, A[6]=33, A[2]=22, and A[3]=55 initialized in that order

---



AMORTIZED ANALYSIS

- *binary counter*
- *multi-pop stack*
- *dynamic table*

Lecture slides by Kevin Wayne

http://www.cs.princeton.edu/~wayne/kleinberg-tardos

Last updated on 1/5/22 12:37 PM

Amortized analysis

Worst-case analysis. Determine worst-case running time of a data structure operation as function of the input size n.

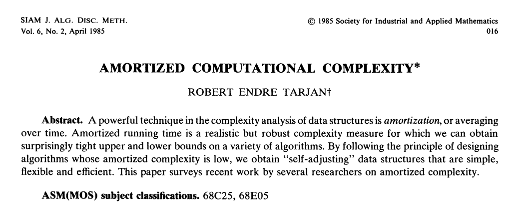
can be too pessimistic if the only way to encounter an expensive operation is when there were lots of previous cheap operations

Amortized analysis. Determine worst-case running time of a sequence of n data structure operations.

Ex. Starting from an empty stack implemented with a dynamic table, any sequence of n push and pop operations takes $O(n)$ time in the worst case.

Amortized analysis: applications

- Splay trees.
- Dynamic table.
- Fibonacci heaps.
- Garbage collection.
- Move-to-front list updating.
- Push–relabel algorithm for max flow.
- Path compression for disjoint-set union.
- Structural modifications to red–black trees.
- Security, databases, distributed computing, ...



SIAM J. ALG. DISC. METH.
Vol. 6, No. 2, April 1985

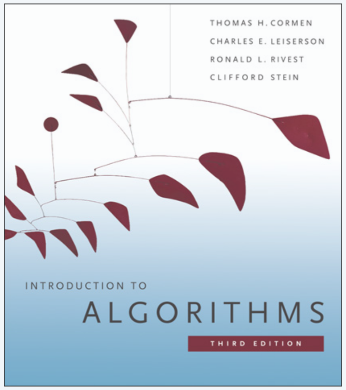
© 1985 Society for Industrial and Applied Mathematics
016

AMORTIZED COMPUTATIONAL COMPLEXITY*

ROBERT ENDRE TARJAN†

Abstract. 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.

ASM(MOS) subject classifications. 68C25, 68E05



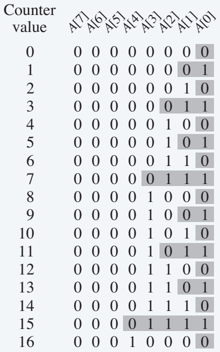
CHAPTER 17

AMORTIZED ANALYSIS

- binary counter
- multi-pop stack
- dynamic table

Binary counter

Goal. Increment a k-bit binary counter $(\text{mod } 2^k)$.
Representation. $A[j] = j^{th}$ least significant bit of counter.



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Cost model. Number of bits flipped.

Binary counter

Goal. Increment a k-bit binary counter (mod 2^k).

Representation. $A[j] = j^{th}$ least significant bit of counter.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n\,k)$ bits. ← overly pessimistic upper bound

Pf. At most k bits flipped per increment. ▪

Aggregate method (brute force)

Aggregate method. Analyze cost of a sequence of operations.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Binary counter: aggregate method

Starting from the zero counter, in a sequence of n INCREMENT operations:

- Bit 0 flips n times.
- Bit 1 flips $\lfloor n / 2 \rfloor$ times.
- Bit 2 flips $\lfloor n / 4 \rfloor$ times.
- …

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.

Pf.

- Bit j flips $\lfloor n / 2^j \rfloor$ times.
- The total number of bits flipped is $\displaystyle\sum_{j=0}^{k-1} \left\lfloor \frac{n}{2^j} \right\rfloor < n \sum_{j=0}^{\infty} \frac{1}{2^j}$

$$= 2n \quad ▪$$

Remark. Theorem may be false if initial counter is not zero.

Accounting method (banker's method)

Assign (potentially) different charges to each operation.

- D_i = data structure after i^{th} operation.
- c_i = actual cost of i^{th} operation.
- $\hat{c}_i$ = amortized cost of i^{th} operation = amount we charge operation i.
- When $\hat{c}_i > c_i$, we store credits in data structure D_i to pay for future ops; when $\hat{c}_i < c_i$, we consume credits in data structure D_i.
- Initial data structure D_0 starts with 0 credits.

can be more or less than actual cost

Credit invariant. The total number of credits in the data structure ≥ 0.

$$\sum_{i=1} \hat{c}_i - \sum_{i=1} c_i \geq 0$$

← our job is to choose suitable amortized costs so that this invariant holds

Accounting method (banker's method)

Assign (potentially) different charges to each operation.

- D_i = data structure after i^{th} operation.
- c_i = actual cost of i^{th} operation.
- $\hat{c}_i$ = amortized cost of i^{th} operation = amount we charge operation i.
- When $\hat{c}_i > c_i$, we store credits in data structure D_i to pay for future ops; when $\hat{c}_i < c_i$, we consume credits in data structure D_i.
- Initial data structure D_0 starts with 0 credits.

can be more or less than actual cost

Credit invariant. The total number of credits in the data structure ≥ 0.

$$\sum_{i=1} \hat{c}_i \; - \; \sum_{i=1} c_i \; \geq \; 0$$

Theorem. Starting from the initial data structure D_0, the total actual cost of any sequence of n operations is at most the sum of the amortized costs.

Pf. The amortized cost of the sequence of n operations is: $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$. ▪

credit invariant

Intuition. Measure running time in terms of credits (time = money).

Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).

increment

Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).
- Flip bit j from 1 to 0: pay for it with the 1 credit saved in bit j.

increment

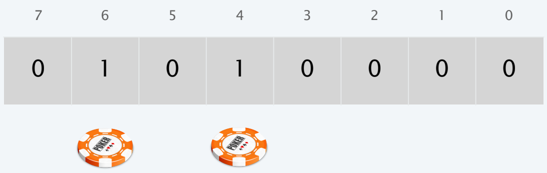
Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).
- Flip bit j from 1 to 0: pay for it with the 1 credit saved in bit j.



Binary counter: accounting method

Credits. One credit pays for a bit flip.
Invariant. Each 1 bit has one credit; each 0 bit has zero credits.

Accounting.

- Flip bit j from 0 to 1: charge 2 credits (use one and save one in bit j).
- Flip bit j from 1 to 0: pay for it with the 1 credit saved in bit j.

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.

Pf.

- Each INCREMENT operation flips at most one 0 bit to a 1 bit, so the amortized cost per INCREMENT ≤ 2.
- Invariant $\Rightarrow$ number of credits in data structure ≥ 0.
- Total actual cost of n operations $\le$ sum of amortized costs $\le 2n$. ▪

the rightmost 0 bit
(unless counter overflows)

accounting method theorem

Potential method (physicist's method)

Potential function. $\Phi(D_i)$ maps each data structure D_i to a real number s.t.:

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each data structure D_i.

Actual and amortized costs.

- c_i = actual cost of i^{th} operation.
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ = amortized cost of i^{th} operation.

our job is to choose
a potential function
so that the amortized cost
of each operation is low

Potential method (physicist's method)

Potential function. $\Phi(D_i)$ maps each data structure D_i to a real number s.t.:

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each data structure D_i.

Actual and amortized costs.

- c_i = actual cost of i^{th} operation.
- $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ = amortized cost of i^{th} operation.

Theorem. Starting from the initial data structure D_0, the total actual cost of any sequence of n operations is at most the sum of the amortized costs.
Pf. The amortized cost of the sequence of operations is:

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))\\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\\ &\ge \sum_{i=1}^{n} c_i \quad \blacksquare\end{aligned}$$

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

increment

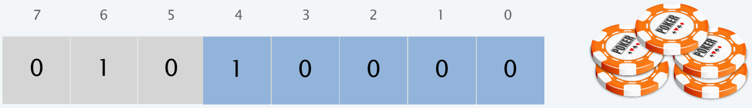
7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	1

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

increment



Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Binary counter: potential method

Potential function. Let $\Phi(D)$ = number of 1 bits in the binary counter D.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.

Pf.

- Suppose that the i^{th} INCREMENT operation flips t_i bits from 1 to 0.
- The actual cost $c_i \le t_i + 1$. ← operation flips at most one bit from 0 to 1 (no bits flipped to 1 when counter overflows)
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

  $\le c_i + 1 - t_i$ ← potential decreases by 1 for t_i bits flipped from 1 to 0 and increases by 1 for bit flipped from 0 to 1

  ≤ 2.
- Total actual cost of n operations $\le$ sum of amortized costs $\le 2n$. ▪

↑ potential method theorem

Famous potential functions

Fibonacci heaps. $\Phi(H) = 2\,trees(H) + 2\,marks(H)$

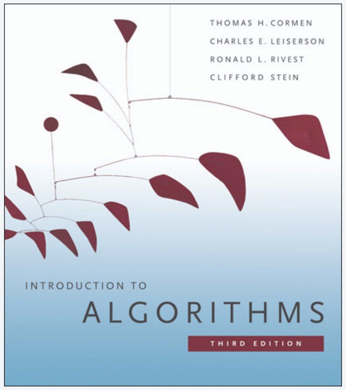
Splay trees. $\Phi(T) = \sum_{x \in T} \lfloor \log_2 size(x) \rfloor$

Move-to-front. $\Phi(L) = 2\,inversions(L, L^*)$

Preflow–push. $\Phi(f) = \sum_{v\,:\,excess(v) > 0} height(v)$

Red–black trees. $\Phi(T) = \sum_{x \in T} w(x)$

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red} \\ 1 & \text{if } x \text{ is black and has no red children} \\ 0 & \text{if } x \text{ is black and has one red child} \\ 2 & \text{if } x \text{ is black and has two red children} \end{cases}$$



SECTION 17.4

AMORTIZED ANALYSIS

- *binary counter*
- *multi-pop stack*
- *dynamic table*

Multipop stack

Goal. Support operations on a set of elements:

- PUSH(S, x): add element x to stack S.
- POP(S): remove and return the most-recently added element.
- MULTI-POP(S, k): remove the most-recently added k elements.

```
MULTI-POP(S, k)
FOR $i = 1$ TO $k$
    POP(S).
```

Exceptions. We assume POP throws an exception if stack is empty.

Multipop stack

Goal. Support operations on a set of elements:

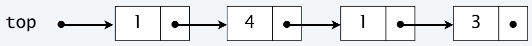
- PUSH(S, x): add element x to stack S.
- POP(S): remove and return the most-recently added element.
- MULTI-POP(S, k): remove the most-recently added k elements.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n^2)$ time.

overly pessimistic upper bound

Pf.

- Use a singly linked list.
- POP and PUSH take $O(1)$ time each.
- MULTI-POP takes $O(n)$ time. ▪



Multipop stack: aggregate method

Goal. Support operations on a set of elements:

- PUSH(S, x): add element x to stack S.
- POP(S): remove and return the most-recently added element.
- MULTI-POP(S, k): remove the most-recently added k elements.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf.

- An element is popped at most once for each time that it is pushed.
- There are $\leq n$ PUSH operations.
- Thus, there are $\leq n$ POP operations (including those made within MULTI-POP). ▪

Multipop stack: accounting method

Credits. 1 credit pays for either a PUSH or POP.
Invariant. Every element on the stack has 1 credit.

Accounting.

- PUSH(S, x): charge 2 credits.
  - use 1 credit to pay for pushing x now
  - store 1 credit to pay for popping x at some point in the future
- POP(S): charge 0 credits.
- MULTIPOP(S, k): charge 0 credits.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf.

- Invariant $\Rightarrow$ number of credits in data structure ≥ 0.
- Amortized cost per operation ≤ 2.
- Total actual cost of n operations $\le$ sum of amortized costs $\le 2n$. ▪

↑ accounting method theorem

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [Case 1: push]

- Suppose that the i^{th} operation is a PUSH.
- The actual cost $c_i = 1$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [Case 2: pop]

- Suppose that the i^{th} operation is a POP.
- The actual cost $c_i = 1$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$.

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [Case 3: multi-pop]

- Suppose that the i^{th} operation is a MULTI-POP of k objects.
- The actual cost $c_i = k$.
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$. ▪

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of elements currently on the stack.

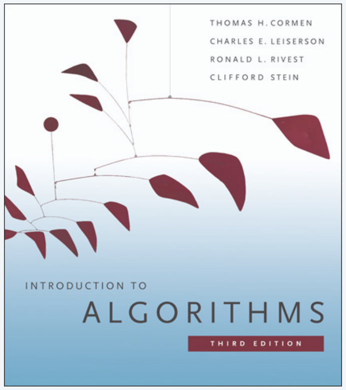
- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for each D_i.

Theorem. Starting from an empty stack, any intermixed sequence of n PUSH, POP, and MULTI-POP operations takes $O(n)$ time.

Pf. [putting everything together]

- Amortized cost $\hat{c}_i \leq 2$. ← 2 for push; 0 for pop and multi-pop
- Sum of amortized costs $\hat{c}_i$ of the n operations $\leq 2\,n$.
- Total actual cost $\leq$ sum of amortized cost $\leq 2\,n$. ▪

↑ potential method theorem



SECTION 17.4

AMORTIZED ANALYSIS

- *binary counter*
- *multi-pop stack*
- *dynamic table*

Dynamic table

Goal. Store items in a table (e.g., for hash table, binary heap).

- Two operations: INSERT and DELETE.
  - too many items inserted ⇒ expand table.
  - too many items deleted ⇒ contract table.
- Requirement: if table contains m items, then space = $\Theta(m)$.

Theorem. Starting from an empty dynamic table, any intermixed sequence of n INSERT and DELETE operations takes $O(n^2)$ time.

overly pessimistic upper bound

Pf. Each INSERT or DELETE takes $O(n)$ time. ▪

Dynamic table: insert only

- When inserting into an empty table, allocate a table of capacity 1.
- When inserting into a full table, allocate a new table of twice the capacity and copy all items.
- Insert item into table.

insert	old capacity	new capacity	insert cost	copy cost
1	0	1	1	–
2	1	2	1	1
3	2	4	1	2
4	4	4	1	–
5	4	8	1	4
6	8	8	1	–
7	8	8	1	–
8	8	8	1	–
9	8	16	1	8
⋮	⋮	⋮	⋮	⋮

Cost model. Number of items written (due to insertion or copy).

Dynamic table: insert only (aggregate method)

Theorem. [via aggregate method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.

Pf. Let c_i denote the cost of the i^{th} insertion.

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

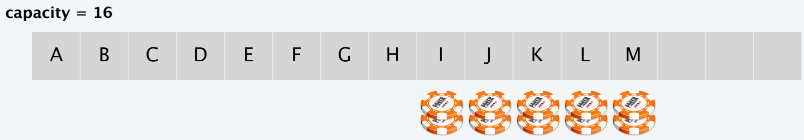
Starting from empty table, the cost of a sequence of n INSERT operations is:

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \quad \blacksquare \end{aligned}$$

Dynamic table demo: insert only (accounting method)

Insert. Charge 3 credits (use 1 credit to insert; save 2 with new item).
Invariant. 2 credits with each item in right half of table; none in left half.

insert N



Dynamic table: insert only (accounting method)

Insert. Charge 3 credits (use 1 credit to insert; save 2 with new item).

Invariant. 2 credits with each item in right half of table; none in left half.

slight cheat if table capacity = 1
(can charge only 2 credits for first insert)

Pf. [by induction]

- Each newly inserted item gets 2 credits.
- When table doubles from k to $2k$, $k / 2$ items in the table have 2 credits.
  - these k credits pay for the work needed to copy the k items
  - now, all k items are in left half of table (and have 0 credits)

Theorem. [via accounting method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.
Pf.

- Invariant $\Rightarrow$ number of credits in data structure ≥ 0.
- Amortized cost per INSERT $= 3$.
- Total actual cost of n operations $\le$ sum of amortized cost $\le 3n$. ▪

accounting method theorem

Dynamic table: insert only (potential method)

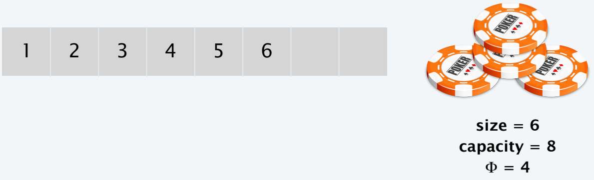
Theorem. [via potential method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.

Pf. Let $\Phi(D_i) = 2\, size(D_i) - capacity(D_i)$.

number of elements

capacity of array

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i. ← immediately after doubling $capacity(D_i) = 2\, size(D_i)$



Dynamic table: insert only (potential method)

Theorem. [via potential method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.

Pf. Let $\Phi(D_i) = 2\ size(D_i)\ -\ capacity(D_i)$.

number of elements

capacity of array

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Case 0. [first insertion]

- Actual cost $c_1 = 1$.
- $\Phi(D_1) - \Phi(D_0) = (2\ size(D_1) - capacity(D_1)) - (2\ size(D_0) - capacity(D_0))$
  $= 1$.
- Amortized cost $\hat{c}_1 = c_1 + (\Phi(D_1) - \Phi(D_0))$
  $= 1 + 1$
  $= 2$.

Dynamic table: insert only (potential method)

Theorem. [via potential method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.

Pf. Let $\Phi(D_i) = 2\ size(D_i)\ -\ capacity(D_i)$.

number of elements

capacity of array

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Case 1. [no array expansion] $capacity(D_i) = capacity(D_{i-1})$.

- Actual cost $c_i = 1$.
- $\Phi(D_i) - \Phi(D_{i-1}) = (2\ size(D_i) - capacity(D_i)) - (2\ size(D_{i-1}) - capacity(D_{i-1}))$
  $= 2$.
- Amortized cost $\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$
  $= 1 + 2$
  $= 3$.

Dynamic table: insert only (potential method)

Theorem. [via potential method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.

Pf. Let $\Phi(D_i) = 2\ size(D_i)\ -\ capacity(D_i)$.

number of elements

capacity of array

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

Case 2. [array expansion] $capacity(D_i) = 2\ capacity(D_{i-1})$.

- Actual cost $c_i = 1 + capacity(D_{i-1})$.
- $\Phi(D_i) - \Phi(D_{i-1}) = (2\ size(D_i) - capacity(D_i)) - (2\ size(D_{i-1}) - capacity(D_{i-1}))$
  $= 2 - capacity(D_i) + capacity(D_{i-1})$
  $= 2 - capacity(D_{i-1})$.
- Amortized cost $\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$
  $= 1 + capacity(D_{i-1}) + (2 - capacity(D_{i-1}))$
  $= 3$.

Dynamic table: insert only (potential method)

Theorem. [via potential method] Starting from an empty dynamic table, any sequence of n INSERT operations takes $O(n)$ time.

Pf. Let $\Phi(D_i) = 2\ size(D_i)\ -\ capacity(D_i)$.

number of elements

capacity of array

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \ge 0$ for each D_i.

[putting everything together]

- Amortized cost per operation $\hat{c}_i \le 3$.
- Total actual cost of n operations $\le$ sum of amortized cost $\le 3n$. ▪

potential method theorem

Dynamic table: doubling and halving

Thrashing.

- Insert: when inserting into a full table, double capacity.
- Delete: when deleting from a table that is ½-full, halve capacity.

Efficient solution.

- When inserting into an empty table, initialize table size to 1; when deleting from a table of size 1, free the table.
- Insert: when inserting into a full table, double capacity.
- Delete: when deleting from a table that is ¼-full, halve capacity.

Memory usage. A dynamic table uses $\Theta(n)$ memory to store n items.

Pf. Table is always between 25% and 100% full. ▪

Dynamic table demo: insert and delete (accounting method)

Insert. Charge 3 credits (1 to insert; save 2 with item if in right half).

Delete. Charge 2 credits (1 to delete; save 1 in empty slot if in left half).

Invariant 1. 2 credits with each item in right half of table.

Invariant 2. 1 credit with each empty slot in left half of table.

delete M

capacity = 16

A	B	C	D	E	F	G	H	I	J	K	L	M			

Dynamic table: insert and delete (accounting method)

Insert. Charge 3 credits (1 to insert; save 2 with item if in right half).

Delete. Charge 2 credits (1 to delete; save 1 in empty slot if in left half).

discard any existing or extra credits

Invariant 1. 2 credits with each item in right half of table. ← to pay for expansion

Invariant 2. 1 credit with each empty slot in left half of table. ← to pay for contraction

Theorem. [via accounting method] Starting from an empty dynamic table, any intermixed sequence of n Insert and Delete operations takes $O(n)$ time.

Pf.

- Invariants ⇒ number of credits in data structure ≥ 0.
- Amortized cost per operation ≤ 3.
- Total actual cost of n operations $\leq$ sum of amortized cost $\leq 3n$. ▪

↑ accounting method theorem

Dynamic table: insert and delete (potential method)

Theorem. [via potential method] Starting from an empty dynamic table, any intermixed sequence of n Insert and Delete operations takes $O(n)$ time.

Pf sketch.

- Let $\alpha(D_i) = size(D_i) \,/\, capacity(D_i)$.
- Define $\Phi(D_i) = \begin{cases} 2\,size(D_i) \; - \; capacity(D_i) & \text{if } \alpha(D_i) \geq 1/2 \\ \frac{1}{2}\,capacity(D_i) \; - \; size(D_i) & \text{if } \alpha(D_i) < 1/2 \end{cases}$
- $\Phi(D_0) = 0, \Phi(D_i) \geq 0.$ [a potential function]
- When $\alpha(D_i) = 1/2, \Phi(D_i) = 0.$ [zero potential after resizing]
- When $\alpha(D_i) = 1, \Phi(D_i) = size(D_i).$ [can pay for expansion]
- When $\alpha(D_i) = 1/4, \Phi(D_i) = size(D_i).$ [can pay for contraction]

…