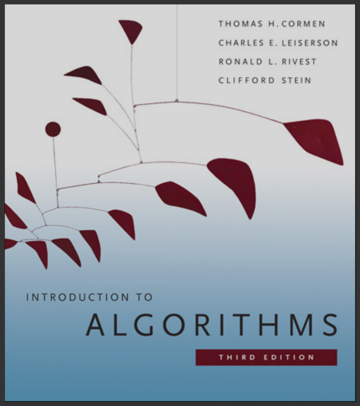


DATA STRUCTURES

- *amortized analysis*
- *binomial heaps*
- *Fibonacci heaps*
- *union-find*

Lecture slides by Kevin Wayne

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

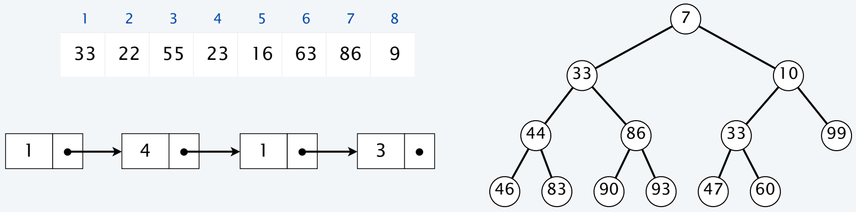
Last updated on Apr 8, 2013 6:13 AM

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 i^{th} element of array.
- WRITE($A, i, value$): set i^{th} element of 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 i^{th} element in $O(1)$ time.

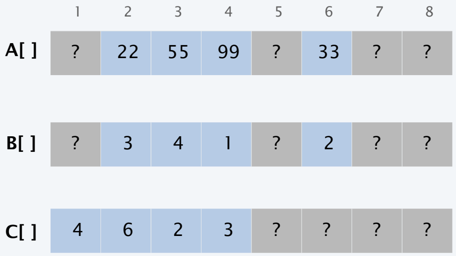
Remark. An array does INIT in $O(n)$ time and READ and WRITE in $O(1)$ time.

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 entry 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$(n)$.
$s \leftarrow$ MALLOC$(n)$.
```

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

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

```
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. $\Rightarrow$

- 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. $\Leftarrow$

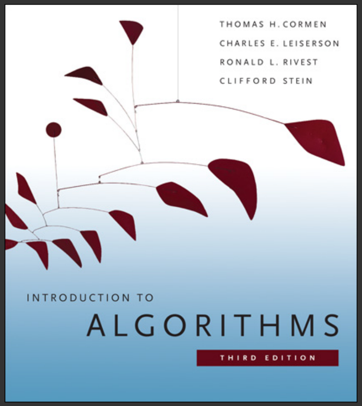
- Suppose $A[i]$ is uninitialized.
- If $B[i] < 1$ or $B[i] > k$, then $A[i]$ clearly uninitialized.
- If $1 \le B[i] \le k$ by coincidence, then we still can't have $C[B[i]] = i$ because none of the entries $C[1..k]$ can equal i. ▪

	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*
- *multipop stack*
- *dynamic table*

Lecture slides by Kevin Wayne

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

Last updated on Apr 8, 2013 6:13 AM

Amortized analysis

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

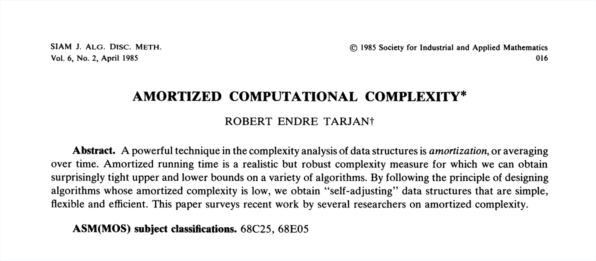
can be too pessimistic if the only way to encounter an expensive operation is if there were lots of previous cheap operations

Amortized analysis. Determine worst-case running time of a sequence of data structure operations as a function of the input size.

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, ...



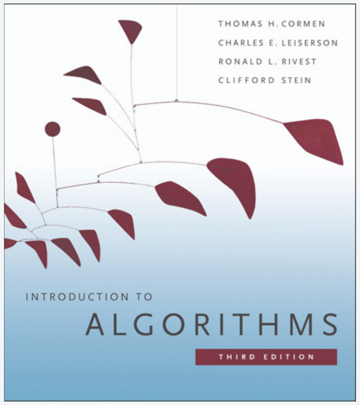
SIAM J. ALG. DISC. METH.
Vol. 6, No. 2, April 1985

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*
- *multipop stack*
- *dynamic table*

Binary counter

Goal. Increment a k-bit binary counter $(\bmod 2^k)$.

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 $O(nk)$ bits.

Pf. At most k bits flipped per increment. ▪

Aggregate method (brute force)

Aggregate method. Sum up sequence of operations, weighted by their cost.

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)

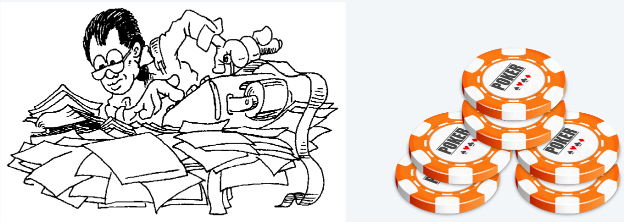
Assign different charges to each operation.

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

can be more or less than actual cost

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

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



Accounting method (banker's method)

Assign different charges to each operation.

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

can be more or less than actual cost

Key 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 operations is: $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$. ▪

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

Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each bit that is set to 1 has one credit.

Accounting.

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

increment

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

Binary counter: accounting method

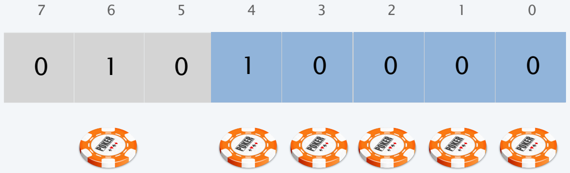
Credits. One credit pays for a bit flip.

Invariant. Each bit that is set to 1 has one credit.

Accounting.

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

increment



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

Binary counter: accounting method

Credits. One credit pays for a bit flip.

Invariant. Each bit that is set to 1 has one credit.

Accounting.

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

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

Binary counter: accounting method

Credits. One credit pays for a bit flip.
Invariant. Each bit that is set to 1 has one credit.

Accounting.

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

Theorem. Starting from the zero counter, a sequence of n INCREMENT operations flips $O(n)$ bits.
Pf. The algorithm maintains the invariant that any bit that is currently set to 1 has one credit $\Rightarrow$ number of credits in each bit ≥ 0. ▪

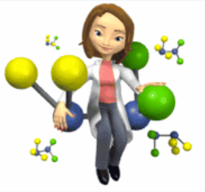
Potential method (physicist's method)

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

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



Potential method (physicist's method)

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

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

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

Binary counter: potential method

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

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

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

increment

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

Binary counter: potential method

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

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

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

Binary counter: potential method

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

- $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ 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 $c_i \leq t_i + 1$. ← operation sets one bit to 1 (unless counter resets to zero)
- The amortized cost $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

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

Famous potential functions

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

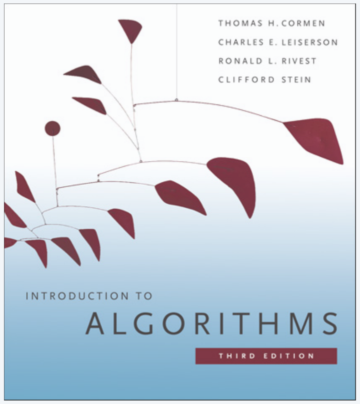
Splay trees. $\Phi(T) = \sum_{x \in T} \lfloor \log_2 size(x) \rfloor$

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

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*
- *multipop stack*
- *dynamic table*

Multipop stack

Goal. Support operations on a set of n elements:

- Push(S, x): push object x onto stack S.
- Pop(S): remove and return the most-recently added object.
- MultiPop(S, k): remove the most-recently added k objects.

```
MultiPop (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 n elements:

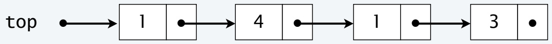
- Push(S, x): push object x onto stack S.
- Pop(S): remove and return the most-recently added object.
- MultiPop(S, k): remove the most-recently added k objects.

Theorem. Starting from an empty stack, any intermixed sequence of n Push, Pop, and MultiPop operations takes $O(n^2)$ time.

overly pessimistic upper bound

Pf.

- Use a singly-linked list.
- Pop and Push take $O(1)$ time each.
- MultiPop takes $O(n)$ time. ▪



Multipop stack: aggregate method

Goal. Support operations on a set of n elements:

- Push(S, x): push object x onto stack S.
- Pop(S): remove and return the most-recently added object.
- MultiPop(S, k): remove the most-recently added k objects.

Theorem. Starting from an empty stack, any intermixed sequence of n Push, Pop, and MultiPop operations takes $O(n)$ time.

Pf.

- An object is popped at most once for each time it is pushed onto stack.
- There are $\leq n$ Push operations.
- Thus, there are $\leq n$ Pop operations (including those made within MultiPop). ▪

Multipop stack: accounting method

Credits. One credit pays for a push or pop.

Accounting.

- PUSH(S, x): charge two credits.
  - use one credit to pay for pushing x now
  - store one credit to pay for popping x at some point in the future
- No other operation is charged a credit.

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

Pf. The algorithm maintains the invariant that every object remaining on the stack has 1 credit $\Rightarrow$ number of credits in data structure ≥ 0. ▪

Multipop stack: potential method

Potential function. Let $\Phi(D)$ = number of objects 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 MULTIPOP 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 objects 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 MULTIPOP 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

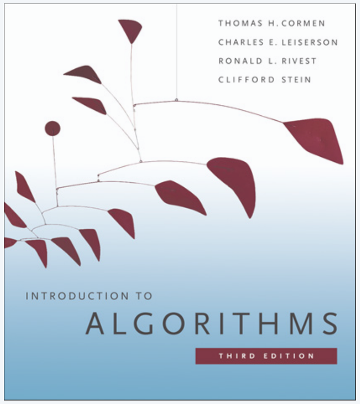
Potential function. Let $\Phi(D)$ = number of objects currently on the stack.

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

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

Pf. [Case 3: multipop]

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



SECTION 17.4

AMORTIZED ANALYSIS

- *binary counter*
- *multipop 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. A single INSERT or DELETE takes $O(n)$ time. ▪

Dynamic table: insert only

- Initialize table to be size 1.
- INSERT: if table is full, first copy all items to a table of twice the size.

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

Cost model. Number of items that are copied.

Dynamic table: insert only

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: insert only

Dynamic table: insert only

Accounting.

- INSERT: charge 3 credits (use 1 credit to insert; save 2 with new item).

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

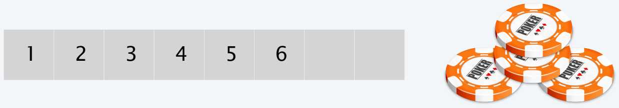
Pf. The algorithm maintains the invariant that there are 2 credits with each item in right half of table.

- When table doubles, one-half of the items in the table have 2 credits.
- This pays for the work needed to double the table. ▪

Dynamic table: insert only

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

Pf. Let $\Phi(D_i) = 2\ \underbrace{size(D_i)}_{\text{number of elements}} - \underbrace{capacity(D_i)}_{\text{capacity of array}}$.



Dynamic table: insert only

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

Pf. Let $\Phi(D_i) = 2\ \underbrace{size(D_i)}_{\text{number of elements}} - \underbrace{capacity(D_i)}_{\text{capacity of array}}$.

Case 1. [does not trigger expansion] $size(D_i) \le capacity(D_{i-1})$.

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

Case 2. [triggers expansion] $size(D_i) = 1 + capacity(D_{i-1})$.

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

Dynamic table: doubling and halving

Thrashing.

- Initialize table to be of fixed size, say 1.
- INSERT: if table is full, expand to a table of twice the size.
- DELETE: if table is ½-full, contract to a table of half the size.

Efficient solution.

- Initialize table to be of fixed size, say 1.
- INSERT: if table is full, expand to a table of twice the size.
- DELETE: if table is ¼-full, contract to a table of half the size.

Memory usage. A dynamic table uses $O(n)$ memory to store n items.
Pf. Table is always at least ¼-full (provided it is not empty). ▪

Dynamic table: insert and delete

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

Pf.

- In between resizing events, each INSERT and DELETE takes $O(1)$ time.
- Consider total amount of work between two resizing events.
  - Just after the table is doubled to size m, it contains $m / 2$ items.
  - Just after the table is halved to size m, it contains $m / 2$ items.
  - Just before the next resizing, it contains either $m / 4$ or $2\,m$ items.
  - After resizing to m, we must perform $\Omega(m)$ operations before we resize again (either ≥ m insertions or ≥ $m / 4$ deletions).
- Resizing a table of size m requires $O(m)$ time. ▪

Dynamic table: insert and delete

Dynamic table: insert and delete

Accounting.

- INSERT: charge 3 credits (1 credit for insert; save 2 with new item).
- DELETE: charge 2 credits (1 credit to delete, save 1 in emptied slot).

discard any existing credits

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

Pf. The algorithm maintains the invariant that there are 2 credits with each item in the right half of table; 1 credit with each empty slot in the left half.

- When table doubles, each item in right half of table has 2 credits.
- When table halves, each empty slot in left half of table has 1 credit. ▪

Dynamic table: insert and delete

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 $\alpha(D_i) = size(D_i) \,/\, capacity(D_i)$.

$$\Phi(D_i) = \begin{cases} 2\,size(D_i) \;-\; capacity(D_i) & \text{if } \alpha \geq 1/2 \\ \frac{1}{2}\,capacity(D_i) \;-\; size(D_i) & \text{if } \alpha < 1/2 \end{cases}$$

- When $\alpha(D) = 1/2$, $\Phi(D) = 0$. [zero potential after resizing]
- When $\alpha(D) = 1$, $\Phi(D) = size(D_i)$. [can pay for expansion]
- When $\alpha(D) = 1/4$, $\Phi(D) = size(D_i)$. [can pay for contraction]

  …