3-2 Amortized Analysis

Hengfeng Wei

hfwei@nju.edu.cn

October 08, 2018

Robert Tarjan

John Hopcroft

For fundamental achievements
in the design and analysis of algorithms and data structures.

— *Turing Award, 1986*

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

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.

"Amortized Computational Complexity", 1985

Amortized analysis is

an algorithm analysis technique for

analyzing a sequence of operations

irrespective of the input to show that

the average cost per operation is small, even though

a single operation within the sequence might be expensive.

By averaging the cost per operation over a worst-case sequence,

amortized analysis can yield a time complexity that is

more robust than average-case analysis, since

its probabilistic assumptions on inputs may be false,

and more realistic than worst-case analysis, since it may be

impossible for every operation to take the worst-case time,

as occurs often in manipulation of data structures.

Dynamic Tables

(I) Summation Method

(II) Accounting Method

(III) Potential Method

(I) Summation Method

(II) Accounting Method

(III) Potential Method

Dynamic Tables

"Move-to-Front" List

Splay Tree

The Summation Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

The Summation Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$\boxed{\forall i,\ \hat{c}_i = \frac{\left(\sum_{i=1}^{n} c_i\right)}{n}}$$

The Summation Method for Array Doubling

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

The Summation Method for Array Doubling

On any sequence of n $\textsc{Insert}$s on an initially empty array.

$$
\begin{array}{rcccccccccc}
o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\
c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1
\end{array}
$$

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

$$
\begin{array}{ccccccccccc}
o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\
c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1
\end{array}
$$

$$
c_i = \begin{cases} (i-1)+1 = i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{o.w.} \end{cases}
$$

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

$$
\begin{array}{ccccccccccc}
o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\
c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1
\end{array}
$$

$$
c_i = \begin{cases} (i-1)+1 = i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{o.w.} \end{cases}
$$

$$
\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lceil \lg n \rceil - 1} 2^j = n + (2^{\lceil \lg n \rceil} - 1) \le n + 2n = 3n
$$

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

$$
\begin{array}{lcccccccccc}
o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\
c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1
\end{array}
$$

$$
c_i = \begin{cases} (i-1)+1 = i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{o.w.} \end{cases}
$$

$$
\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lceil \lg n \rceil - 1} 2^j = n + (2^{\lceil \lg n \rceil} - 1) \le n + 2n = 3n
$$

$$
\boxed{\forall i,\ \hat{c}_i = 3}
$$

The Summation Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

The Summation Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$\boxed{\forall i,\ \hat{c}_i = \frac{\left(\sum_{i=1}^{n} c_i\right)}{n}}$$

The Summation Method for Array Doubling

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

$$
\begin{array}{lcccccccccc}
o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\
c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1
\end{array}
$$

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

$$\begin{array}{ccccccccccc} o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\ c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1 \end{array}$$

$$c_i = \begin{cases} (i-1)+1 = i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{o.w.} \end{cases}$$

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

o_i :	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8	o_9	o_{10}
c_i :	1	2	3	1	5	1	1	1	9	1

$$c_i = \begin{cases} (i-1)+1 = i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{o.w.} \end{cases}$$

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lceil \lg n \rceil - 1} 2^j = n + (2^{\lceil \lg n \rceil} - 1) \le n + 2n = 3n$$

The Summation Method for Array Doubling

On any sequence of n INSERTs on an initially empty array.

$$\begin{array}{lcccccccccc} o_i: & o_1 & o_2 & o_3 & o_4 & o_5 & o_6 & o_7 & o_8 & o_9 & o_{10} \\ c_i: & 1 & 2 & 3 & 1 & 5 & 1 & 1 & 1 & 9 & 1 \end{array}$$

$$c_i = \begin{cases} (i-1)+1 = i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{o.w.} \end{cases}$$

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lceil \lg n \rceil - 1} 2^j = n + (2^{\lceil \lg n \rceil} - 1) \le n + 2n = 3n$$

$$\boxed{\forall i,\ \hat{c}_i = 3}$$

The Accounting Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$a_1, a_2, \ldots, a_n$$

The Accounting Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$a_1, a_2, \ldots, a_n$$

$$\boxed{\hat{c_i} = c_i + a_i,\ a_i >=< 0}$$

Amortized Cost $=$ Actual Cost $+$ Accounting Cost

The Accounting Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$a_1, a_2, \ldots, a_n$$

$$\boxed{\hat{c}_i = c_i + a_i, \; a_i >=< 0}$$

Amortized Cost $=$ Actual Cost $+$ Accounting Cost

$$\forall n, \; \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i$$

The Accounting Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$a_1, a_2, \ldots, a_n$$

$$\boxed{\hat{c}_i = c_i + a_i, \; a_i >=< 0}$$

Amortized Cost = Actual Cost + Accounting Cost

$$\forall n, \; \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i \implies \boxed{\forall n, \; \sum_{i=1}^{n} a_i \ge 0}$$

The Accounting Method

$$o_1, o_2, \ldots, o_n$$

$$c_1, c_2, \ldots, c_n$$

$$a_1, a_2, \ldots, a_n$$

$$\boxed{\hat{c}_i = c_i + a_i, \; a_i >=< 0}$$

Amortized Cost = Actual Cost + Accounting Cost

$$\forall n, \; \sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \hat{c}_i \implies \boxed{\forall n, \; \sum_{i=1}^{n} a_i \ge 0}$$

Key Point: Put the accounting cost on specific objects.

The Accounting Method for Array Doubling

$Q : \hat{c}_i = 3$ *vs.* $\hat{c}_i = 2$

The Accounting Method for Array Doubling

$$Q : \hat{c}_i = 3 \textit{ vs. } \hat{c}_i = 2$$

$$\hat{c}_i = 3 = \underbrace{1}_{\text{insert}} + \underbrace{1}_{\text{move itself}} + \underbrace{1}_{\text{help move another}}$$

The Accounting Method for Array Doubling

$$\color{red}{Q} : \hat{c}_i = 3 \text{ vs. } \hat{c}_i = 2$$

$$\hat{c}_i = 3 = \underbrace{1}_{\text{insert}} + \underbrace{1}_{\text{move itself}} + \underbrace{1}_{\color{red}{\text{help move another}}}$$

	$\hat{c}_i$	c_i *(actual cost)*	a_i *(accounting cost)*
INSERT *(normal)*	3	1	2
INSERT *(expansion)*	3	$1+t$	$-t+2$

Simulating a queue Q using two stacks S_1, S_2 (Problem $\mathbb{E}3$)

```
procedure ENQ(x)
    Push(S_1, x)

procedure DEQ()
    if S_2 = ∅ then
        while S_1 ≠ ∅ do
            Push(S_2, Pop(S_1))
    Pop(S_2)
```

The Summation Method for Queue Simulation

$$\frac{\left(\sum_{i=1}^{n} c_i\right)}{n}$$

The Summation Method for Queue Simulation

$$\frac{\left(\sum_{i=1}^{n} c_i\right)}{n}$$

The operation sequence is *NOT* known.

The Accounting Method for Queue Simulation

item:	Push into S_1	Pop from S_1	Push into S_2	Pop from S_2
	1	1	1	1

The Accounting Method for Queue Simulation

item:	Push into S_1	Pop from S_1	Push into S_2	Pop from S_2
	1	1	1	1

$$\hat{c}_{\text{ENQ}} = 3$$

$$\hat{c}_{\text{DEQ}} = 1$$

The Accounting Method for Queue Simulation

item:	Push into S_1	Pop from S_1	Push into S_2	Pop from S_2
	1	1	1	1

$$\hat{c}_{\text{ENQ}} = 3$$
$$\hat{c}_{\text{DEQ}} = 1$$

$$\sum_{i=1}^{n} a_i \geq 0$$

The Accounting Method for Queue Simulation

item:	Push into S_1	Pop from S_1	Push into S_2	Pop from S_2
	1	1	1	1

$$\hat{c}_{\text{ENQ}} = 3$$
$$\hat{c}_{\text{DEQ}} = 1$$

$$\sum_{i=1}^{n} a_i \ge 0 \Longleftarrow \sum_{i=1}^{n} a_i = \#S_1 \times 2$$

The Accounting Method for Queue Simulation

$$\hat{c}_{\text{ENQ}} = 3$$
$$\hat{c}_{\text{DEQ}} = 1$$

$$\#S_1 = t$$

	$\hat{c}_i$	c_i *(actual cost)*	a_i *(accounting cost)*
ENQUEUE	3	1	2
DEQUEUE ($S_2 = \emptyset$)	1	1	0
DEQUEUE ($S_2 \neq \emptyset$)	1	$1 + 2t$	$-2t$

“Move-to-Front” (MTF) List

What work are you proudest of?

What work are you proudest of?

Proudest? It's hard to choose.

What work are you proudest of?

Proudest? It's hard to choose.

I like the ***self-adjusting search tree*** *data structure that Danny Sleator and I developed.*

Self-Adjusting Binary Search Trees

DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

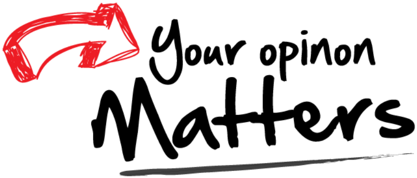
AT&T Bell Laboratories, Murray Hill, NJ

Abstract. The *splay* tree, a self-adjusting form of binary search tree, is developed and analyzed. The binary search tree is a data structure for representing tables and lists so that accessing, inserting, and deleting items is easy. On an n-node splay tree, all the standard search tree operations have an amortized time bound of $O(\log n)$ per operation, where by "amortized time" is meant the time per operation averaged over a worst-case sequence of operations. Thus splay trees are as efficient as balanced trees when total running time is the measure of interest. In addition, for sufficiently long access sequences, splay trees are as efficient, to within a constant factor, as static optimum search trees. The efficiency of splay trees comes not from an explicit structural constraint, as with balanced trees, but from applying a simple restructuring heuristic, called *splaying*, whenever the tree is accessed. Extensions of splaying give simplified forms of two other data structures: lexicographic or multidimensional search trees and link/cut trees.

"Self-Adjusting Binary Search Trees", JACM, 1985

Self-Adjusting Binary Search Trees


THANK YOU



Office 302

Mailbox: H016

hfwei@nju.edu.cn