CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1]with huge thanks to Anna Bretscher and Albert Lai

introduction

Today we begin studying how to calculate

- the total time of
- a sequence of operations as a whole

As opposed to what?

Sum of worst case times of each individual operations separately

multi-pop stack

As an example consider multi-pop stack operations:

- `push(`x`)`:
  - time complexity: $\Theta(1)$
- `pop()`:
  - time complexity: $\Theta(1)$
- `multipop(`k`)`:
  - `pop()` up to k times
  - time complexity: $\Theta(k)$

Start from empty and perform m operations. What is the total time?

multi-pop stack: naïve cost analysis

Start from empty and perform n operations. What is the total time?

1. each operation:

2. if stack size close to n:

3. total is:

multi-pop stack: naïve cost analysis

Start from empty and perform n operations. What is the total time?

1. each operation: $O(k)$ time for multipop(k)
2. if stack size close to n: $O(n)$ time
3. total is: $O(n^2)$ for n operations

But can this actually happen?

multi-pop stack: better cost analysis

Starting from empty, perform n operations:

1. at most n `pushes`
2. cannot `pop` / `multipop` more than what has been `pushed`
3. all `pops` and `multipops` together: at most n pops
4. total: n operations take $\mathcal{O}(n)$ time in the worst case

amortized time

Idea:

- if n operations take $\mathcal{O}(n)$ total time in the worst case, then
- each operation takes $\mathcal{O}(1)$ **amortized** time

amortized time

Idea:

- if n operations take $\mathcal{O}(n)$ total time in the worst case, then
- each operation takes $\mathcal{O}(1)$ **amortized** time

In general:

- if n operations take $\mathcal{O}(f(n))$ total time in the worst case, then
- each operation takes $\mathcal{O}(f(n)/n)$ **amortized** time

amortization method #0: aggregate

Aggregate method:

- what we just saw with multi-pop stacks
- make an observation / argument about overall number of steps in n operations
- usually examine how different operations depend on each other
- divide total steps by the number of opearations

amortization method #1: accounting

Accounting method:

Using our multi-pop stacks example, consider:

- each operation receives 2 dollars
- `push` and `pop` spend 1 dollar
- `multipop(`k`)` spends the number of items actually popped → min(k, size)
- if leftover after operation: save for future
- if not enough for operation: spend from savings

amortization method #1: accounting

Accounting method:

Using our multi-pop stacks example, consider:

- each operation receives 2 dollars
- `push` and `pop` spend 1 dollar
- `multipop(`k`)` spends the number of items actually popped
- if leftover after operation: save for future
- if not enough for operation: spend from savings

Only works if:

amortization method #1: accounting

Accounting method:

Using our multi-pop stacks example, consider:

- each operation receives 2 dollars
- `push` and `pop` spend 1 dollar
- `multipop(`k`)` spends the number of items actually popped
- if leftover after operation: save for future
- if not enough for operation: spend from savings

Only works if: always have "enough" to pay

1. Prove invariant: amount ≥ 0
2. Conclude: each operation takes $\mathcal{O}(2)$ amortized time (i.e., what it receives).

Accounting method: multipop example

Prove invariant: $amount \geq size$.

1. Initially: $amount = size = 0$
2. `push`:
   - Assume $amount \geq size$ before `push`
   - $amount' = amount + receive - spend = amount + 1$
   - $size' = size + 1$
   - $\therefore amount' \geq size'$
3. pop:
   - Assume $amount \geq size$ before pop
   - $amount' = amount + receive - spend = amount + 1$
   - $size' = size - 1$
   - $\therefore amount' \geq size'$

→ pay \$1 for push, save \$1 = \$2 total (2) (1)

→ pay \$1 saved for pop = \$0 total

Accounting method: multipop example

Prove invariant: $amount \geq size$.

4. `multipop`:
   - Assume $amount \geq size$ before `multipop`
   - Let k be the number of items popped.
   - $amount' = amount + receive - spend^{(k)} = amount + 2 - k$
   - $size' = size - k$
   - $\therefore amount' \geq size'$

pay $k \times \$1$ saved for multipop
$= \$0$ total

Finally, note that $size \geq 0$ and therefore $amount \geq 0$ is an invariant.

multipop example: potential function

Formally:

D_i : Data structure at ith operation

- Define a potential function $\Phi(D_i)$ = stack size after i operations
  number of elements stored at Data Structure i
- Let t_i = time(operation i)
  [real time complexity]
- Let $t = \sum_{i=1}^{n}$ total time of n operations
- Let $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1})$
  $ saved in step i

Then:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \left[t_i + \phi(D_i) - \phi(D_{i-1}) \right]$$

$$= \sum_{i=1}^{n} t_i + \underbrace{\phi(D_n) - \phi(D_0)}_{\geq 0} \geq \sum_{i=1}^{n} t_i$$

Thus, we can use $\mathcal{O}(a_i)$ as amortized time upper bound.

$$\sum_{\text{operations}} \text{amortized cost} \geqslant \sum_{\text{operations}} \text{actual cost}$$

multipop example: amortized time

Let:

- $\Phi(D_i) =$ stack size after i operations
- $t_i =$ time(operation i)
- $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1})$
- $s =$ stack size before i^{th} operation

Then:

- `push`:
- `pop`:
- `multipop(`k`)`:

Conclusion: each amortized time is in $\mathcal{O}(1)$.

amortized time: in general

- Define $\Phi(D_i)$:
  potential function for data structure D after i operations
- Prove $\Phi(D_n) \geq \Phi(D_0)$ for all $n \geq n_0$ sequences of operations
- Let $t_i = \text{time(operation } i)$
- Then $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1})$ is amortized time
  - can be different for different operations

expandable arrays / dynamic arrays / array lists ...

Data structure:

- usual array operations:
  - `get(`i`)`: read $A[i]$ for $0 \leq i < size(A)$
  - `set(`i`, `x`)`: write $A[i] := x$ for $0 \leq i < size(A)$
  - `size()`: return size of A : current number of elements in A
- but size can grow
  - `add(`x`)`:
    - write x at the end of A, if there is space
    - if A is full, double capacity and copy all elements before adding x
- examples in your favourite programming languages?

expandable array: add

```
dynamic_array {
   int capacity;    // capacity / length of arr
   int size;        // current number of elements
   T* arr;          // array of elements (of type T)
}

add(x):
0. if size = capacity:
1.    capacity := 2 * capacity
2.    newArr := new array of length capacity
3.    copy elements of arr into newArr
4.    arr := newArr
5. arr[size++] := x
```

expandable arrays: amortized time idea

- `get`, `set`, `size`: receive \$1, spend \$1.
- `add`: receives \$3.
- if need to double capacity and copy:
  - since last copying, *capacity*/2 cells have \$2 saved each
  - so \$*capacity* saved in total
  - enough to copy

expandable arrays: amortized time

Invariant: $\mathit{capacity} \leq 2 * \mathit{size}$
Proof: exercise.

Define potential $\Phi(D) = 2 * \mathit{size} - \mathit{capacity}$.
Then $\Phi(D_i) \geq 0$ for all i.

Prove: $\Phi(D_n) - \Phi(D_0) \geq 0$ for all sequences of n operations.

Then can compute amotrized time as $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1})$.

expandable arrays: amortized time

- `get`, `set`, `size`: do not change `size` nor `capacity`, $\mathcal{O}(1)$.
- `add`, if no copying: (1 element)

$$a_i = 1 + [2(\text{size}+1) - \text{capacity}] - (2\,\text{size} - \text{capacity}) = 3$$

$[2(\text{size}+1) - \text{capacity}]$: $\phi(D_i)$; $(2\,\text{size} - \text{capacity})$: $\phi(D_{i-1})$

- `add`, if copying:

$$a_i = \text{capacity}+1 + [2(\text{size}+1) - 2\,\text{capacity}] - (2\,\text{size} - \text{capacity})$$
$$= 3$$

Therefore `add`'s amortized time is $\mathcal{O}(3)$.