CSCB63 – Design and Analysis of Data Structures

Anya Tafliovich[1]

[1] based on notes by 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 the worst case times of
- each individual operation 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: $\Theta(k)$ worst case

Start from empty and perform n 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: `multipop(`k`)` takes $\mathcal{O}(k)$ time
2. if stack size close to n: `multipop(`n`)` takes $\mathcal{O}(n)$ time
3. total is: n operations take $\mathcal{O}(n^2)$ time

But can this actually happen?

- if `multipop` happens rarely, don't have n expensive operations
- if `multipop` happens often, stack size won't grow to n

So either way, we won't get $\mathcal{O}(n^2)$!

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

amortised time

Idea:

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

In general:

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

amortisation 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 operations

amortisation 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 $\min(k, size)$ dollars
- 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 $\mathcal{O}(2)$ amortised time (i.e., what it receives).

Accounting method: multipop example

Prove invariant: $\mathit{amount} \geq \mathit{size}$.

1. Initially: $\mathit{amount} = 0 = \mathit{size}$
2. push:
   - Assume $\mathit{amount} \geq \mathit{size}$ before push
   - $\mathit{amount}' = \mathit{amount} + \mathit{credit}(\mathtt{push}) - \mathit{cost}(\mathtt{push}) = \mathit{amount} + 2 - 1 = \mathit{amount} + 1$
   - $\mathit{size}' = \mathit{size} + 1$
   - $\therefore \mathit{amount}' \geq \mathit{size}'$
3. pop:
   - Assume $\mathit{amount} \geq \mathit{size}$ before pop
   - $\mathit{amount}' = \mathit{amount} + \mathit{credit}(\mathtt{pop}) - \mathit{cost}(\mathtt{pop}) = \mathit{amount} + 2 - 1 = \mathit{amount} + 1$
   - $\mathit{size}' = \mathit{size} - 1$
   - $\therefore \mathit{amount}' \geq \mathit{size}'$

Accounting method: multipop example

Prove invariant: $\mathit{amount} \geq \mathit{size}$.

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

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

multipop example: potential function

Formally:

- Define a <u>potential</u> function $\Phi(D_i) =$ stack size after i operations
- Let $t_i = \text{time}(\text{operation } i)$
- Let $t = \sum_{i=1}^{n} t_i$ total time of n operations
- Let $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1})$

Then:

$$\begin{aligned}
\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} t_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= t + \Phi(D_n) - \Phi(D_0) \\
&\geq t & \text{since } \Phi(D_n) - \Phi(D_0) \geq 0
\end{aligned}$$

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

multipop example: amortised time

Recall:

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

Then:

- `push`: $a_i = 1 + (\Phi(D_{i-1}) + 1) - \Phi(D_{i-1}) = 2$
- `pop`: $a_i = 1 + (\Phi(D_{i-1}) - 1) - \Phi(D_{i-1}) = 0$
- `multipop(`k`)`: $a_i = j + (\Phi(D_{i-1}) - j) - \Phi(D_{i-1}) = 0$ where $j = \min(k, \Phi(D_{i-1}))$

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

amortised 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 amortised 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 some type T)
}

add(x):
if arr is empty:
  arr := new array of length 1
if size = capacity:
  capacity := 2 * capacity
  newArr := new array of length capacity
  copy elements of arr into newArr
  arr := newArr
arr[size++] := x
```

expandable arrays: amortised 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: amortised time

Invariant: $\textit{capacity} \leq 2 * \textit{size}$

Initially: $\textit{capacity} = 0 = \textit{size}$.

Let *capacity* and *size* be values before some operation, and $\textit{capacity}'$ and $\textit{size}'$ be corresponding values after the operation. Assume $\textit{capacity} \leq 2 * \textit{size}$.

- `get()`, `size()`, `set(i, x)`: no change
- `add(x)`:
  - if $\textit{size} < \textit{capacity}$:
    $\textit{capacity}' = \textit{capacity} \leq 2 * \textit{size} < 2 * (\textit{size} + 1) = 2 * \textit{size}'$
  - if $\textit{size} = \textit{capacity}$:
    $\textit{capacity}' = 2 * \textit{capacity} = 2 * \textit{size} = 2 * (\textit{size}' - 1) < 2 * \textit{size}'$

expandable arrays: amortised time

Invariant: $capacity \leq 2 * size$

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

Prove: $\Phi(D_n) - \Phi(D_0) \geq 0$ for all sequences of $n \geq n_0 = 0$ operations.

$$\Phi(D_n) - \Phi(D_0) = \Phi(D_n) - (2 * 0 - 0) = \Phi(D_n) \geq 0$$

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

expandable arrays: amortised time

Let s, c be array size and capacity before the i^{th} operation.

- `get()`, `size()`, `set(i, x)`: do not change `size` nor `capacity`, $\mathcal{O}(1)$.

- `add`, if no copying:
  $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (2*(s+1)-c) - (2*s-c) = 3$

- `add`, if copying:

$$
\begin{aligned}
a_i &= t_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= (c+1) + (2*(s+1) - 2*c) - (2*s - c) \\
&= 3
\end{aligned}
$$

Therefore amortised time is $\mathcal{O}(1)$.