CS 5321: Advanced Algorithms – Amortized Analysis of Data Structures

Ali Ebnenasir
Department of Computer Science
Michigan Technological University

Motivations

- Why amortized analysis and when?
- Suppose you have a linked list of sorted elements
  - How difficult is it to find min/max?
- If you do not allow insert/delete operations on your data structure, then you can count on the efficiency of the *find* operation
- What happens if you allow insert/delete?
  - While maintaining the order
  - Without concern for the order

Motivation – cont'd

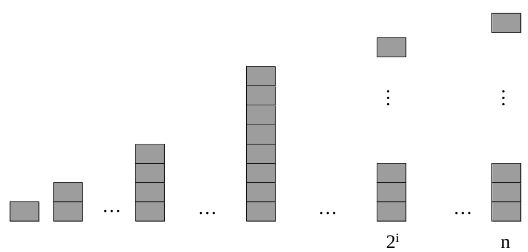
- Insertion/deletion while maintaining the order
  - Find min/max remains cheap, whereas insertion/deletion is expensive
- Insertion/deletion without maintaining the order
  - Find min/max becomes expensive, whereas insertion/deletion is cheap
- So, what?
  - Well, if we allow cheap insertion/deletion, and periodically fix things up, then, even if the cost of fix-up is high, we calculate an *amortized* cost for a sequence of find/insertion/deletion
    - Assumption: frequency of insertion/deletion is low

Motivating Example

- Consider a stack with size k, where initial value of k is 0
- Once the stack is full, allocate another stack with size 2k, and copy all the k elements to the new stack
- What is the *amortized* cost of performing a sequence of n operations?
  - Cost of push and pop is O(1)

Motivating Example – cont'd

- After the i-th expansion, 2^{i-1} elements must be copies to the new stack
- What is the total worst case cost for a sequence of n operations?



Motivating Example: Calculating Amortized Cost

- How many times do we double the size?
  - Log n
- What is the cost of each expansion?
  - 2^{i-1}
- What is the total cost of expansion?
  - $\sum_{i=1}^{\log n} 2^{i-1} = 2^{(\log n)} -1 = n -1$
- What is the total cost?
  - $n + n - 1 < 2n$
  - Thus, we have O(n) total cost for a sequence of n operations
  - Amortized cost = O(1)

Problem Statement

- Characterize and analyze the average cost of a specific operation *over any sequence* of n operations on a *data structure*
  - Even if there are different operations, the amortized cost is the same for all of them
- What is the difference with the average case complexity?
  - Average case:
    - over a probability distribution of input
  - Amortized:
    - no probability is involved; average for a sequence of operations
  - Worst case:
    - time complexity for the worst case input

Techniques

- Aggregate
- Accounting
- Potential

Aggregate Analysis - I

- T(n): worst-case cost for n operations
- Amortized: T(n) / n
- Example: Stack
- Operations:
  - PUSH(S, x): O(1) each; O(n) for any sequence of n operations
  - POP(S): O(1) each; O(n) for any sequence of n operations
  - MULTIPOP(S, k)

Aggregate Analysis – Stack Example

MULTIPOP(S, k)
while S is not empty and $k > 0$
  do POP(S)
    $k \leftarrow k - 1$

- Linear time in the number of POPs
- Number of iterations of while loop?
  - min(s, k), where s is the no. of elements on stack
- Worst case cost of MULTIPOP?
  - O(n)
- Thus, worst case cost of a sequence of n operations?
  - $O(n^2)$

Aggregate Analysis - Stack Example

- Can we have n consecutive MULTIPOPs?
- Can we think of a tighter bound than $O(n^2)$?
- How many times each object can be popped in a sequence of n operations?
  - n. why?
- Thus average over n operations is O(1) per operation
- Aggregate analysis summary
  1. Calculate the worst case cost for a sequence of n operations
  2. Take the average

Aggregate Analysis – Binary Counter

0000
000<u>1</u>
00<u>1</u>0
00<u>11</u>
0100
…
1111

INCREMENT(A, k)
$i \leftarrow 0$
while $i < k$ and $A[i] = 1$
  do $A[i] \leftarrow 0$
    $i \leftarrow i + 1$
if $i < k$
  then $A[i] \leftarrow 1$

- k is the number of bits

Aggregate Analysis – Binary Counter

- k-bit binary counter
- A[k-1] A[k-2] …. A[2] A[1] A[0]
- Count upward from 0
- Increment: add 1 in mod 2^{k-1}
- Worst case cost of an increment?
  - Number of bit flips; O(k)
- Worst case cost of a sequence of n increments
  - O(nk)
- Does all bits flip for every increment?

Aggregate Analysis – Binary Counter

Bit	**flip frequency**	**times in n increments**
0	every time	n
1	½ the time	$\lfloor n/2 \rfloor$
2	¼ the time	$\lfloor n/4 \rfloor$
…	…	…
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
…	…	…
k-1	$1/2^{k-1}$the time	$\lfloor n/2^{k-1} \rfloor$
i>= k	never	0

total no. of flips = $n \cdot \sum_{i=1}^{\log n} (1/2)^i <= 2n$ (why?)

- Worst case cost over the sequence = O(n)
- Amortized cost = O(1)

Accounting Method

Accounting Method - I

- Taking the average does not seem to reflect the individual cost of each operation!
- Allocate some charge for each operation
  - Some maybe charged more than actual cost
  - Some maybe charged less than actual cost
- Amortized cost = the amount we charge
- *Credit* = amortized cost – actual cost
- We can use the credit to compensate the operations whose amortized cost is less than actual cost
- Requirements:
  - Always (*Credit* >=0) holds
  - I.e., *the amortized cost must be an upper bound of the actual cost*
- How does this method differ from the aggregate method?
  - Amortized costs of different operations may not be the same

Accounting Method - II

- What does it mean if (*Credit* < 0)?
  - Undercharging an operation with the promise of repaying the account later
  - Amortized cost cannot be an upper bound
- Let c_i be the actual cost of operation *i*
- Let ac_i be the amortized cost of operation *i*
- Thus, for all sequences of *n* operations, we should have

$$\sum_{i=1}^{n} ac_i - \sum_{i=1}^{n} c_i >= 0$$

  - I.e., the total credit associated with the data structure should be nonnegative at all times

Accounting Method – Stack Example

- For each PUSH consider $2 amortized cost
  - $1 actual cost of PUSH
  - $1 prepayment for POP later on
- What should be the amortized cost of POP and MULTIPOP?
  - Both $0. (Why?)
- Can *Credit* ever become negative?
- What is the amortized cost for n operations?
  - O(n)
- What is the actual cost?

Accounting Method – Stack Example

- What if we have a stack with size k and we take a backup after every k operations?
- How would you assign an amortized cost to each stack operation?
  – PUSH → \$2 Why?
  – POP → \$2 Why?
  – MULTIPOP → \$0 Why?
- Could *Credit* go negative?

Accounting Method – Binary Counter

- How much should we dedicate for setting a bit to 1?
  – \$2 amortized cost of setting a bit to 1. why?
- Should we charge anything for resetting back to 0?
- Given a value in the counter, how can you calculate the current available credit?
- What does this mean?
  – Credit is always nonnegative
- What is the amortized cost of n increments?
  – O(n) why?

Accounting Method vs. Aggregate

- Aggregate considers a uniform cost for all operations
- Accounting considers separate costs for different operations (associates credit to each operation)

Potential Method

Potential Method - I

- In the accounting method, the credit associated to each object can only be used to repay for the same object
- What if we accumulate the credit as a whole to repay for any operation we want?
- This way we have more flexibility
- Accumulated credit = potential of the data structure
- Let D_i be data structure after operation *i*
- Let D_0 be the initial data structure
- Let c_i be the actual cost of operation *i*
- Let ac_i be the amortized cost of operation *i*

Potential Method - II

- Define a potential function
- $\Phi: D_i \rightarrow R$, where
  - R is the set of real numbers and $\Phi(D_i)$ is the potential associated to data structure D_i
- Added potential due to i-th operation is

$$\Phi(D_i) - \Phi(D_{i-1})$$

- Thus, we have $ac_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  $\sum_{i=1}^{n} ac_i = \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} \Phi(D_i) - \Phi(D_{i-1})$
  $= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$ Why?
- What does this say about Φ?
- How can we guarantee that the amortized cost is an upper bound for the actual cost?
  - In each step we should have $\Phi(D_i) >= \Phi(D_0)$

Potential Method - Stack

- How should we define the potential function for stack operations?
- Potential function Φ: number of items in stack
  - i.e., sum of credits of all items in the accounting method
- What is $\Phi(D_0)$?
- Does this always $\Phi(D_i) >= \Phi(D_0)$ hold? Why?

Potential Method - Stack

Operation	c_i	$\Phi(D_i) - \Phi(D_{i-1})$	ac_i
PUSH	1	$(\lvert s\rvert+1) - \lvert s\rvert = 1$	2
POP	1	$(\lvert s\rvert-1) - \lvert s\rvert = -1$	0
MULTIPOP	k'	$(\lvert s\rvert - k') - \lvert s\rvert = -k'$	0

where $k' = \min(k, |s|)$ and $|s|$ is the number of items in stack

- Is amortized cost an upper bound of actual cost?
- What is the amortized cost of a sequence of n operations?
- I leave the binary counter for you as an exercise!

Dynamic Tables

Dynamic Tables - I

- Need a table (e.g., hash table), but do not know how many objects will be stored
- Expand the size of the table when needed
- Contract the table when few items exist
- Can we have an amortized cost O(1) per operation?
- *Constraint*:
  - the unused space is always less than or equal to a fraction of the allocated space
    - In this case, not more than half

Dynamic Tables - II

- Let *size* be the size of the table
- Let *num* be the number of items stored
- Load factor α = *num/size*
- if *size* = 0 then *num* will be 0; then define α = 1
- Assume we *only insert new objects*
- Double the size of the table once it is full
- Constraint: ensure α >= 1/2
- *Elementary insertion*: insertions that do not need expansion

Dynamic Tables - III

TABLE-INSERT(T, x)
if $size[T] = 0$
  then allocate $table[T]$ with 1 slot
    $size[T] \leftarrow 1$
if $num[T] = size[T]$ ▷ expand?
  then allocate *new-table* with $2 \cdot size[T]$ slots
    insert all items in $table[T]$ into *new-table* ▷ $num[T]$ elem insertions
    free $table[T]$
    $table[T] \leftarrow$ *new-table*
    $size[T] \leftarrow 2 \cdot size[T]$
insert x into $table[T]$ ▷ 1 elem insertion
$num[T] \leftarrow num[T] + 1$

Initially, $num[T] = size[T] = 0$.

Aggregate Analysis

- What is the actual cost of i-th operation (denoted c_i)?
  - If i-1 is a power of 2 then $c_i = i$
  - Otherwise, $c_i = 1$
- Total cost = $\sum_{i=1}^{n} c_i = n + \sum_{j=1}^{\log n} 2^j < 3n$
- Amortized cost per operation = 3

Accounting Analysis

- Consider $3 for each object. Why? (remember we only insert)
  - $1 for insertion
  - $1 for moving
  - $1 for moving an element that has been moved once. Why?
- Assume we have m items and just expanded the table
- The existing m items pay for their move, but can they pay for the next move?

Potential Method - I

- Define a potential function for a table T
  - $\Phi(T) = 2\ \text{num}(T) - \text{size}(T)$
- Initial state?
  - $\Phi(T) = 0$
- Just before expansion?
  - num = size, thus $\Phi(T)$ = num
- Just after expansion?
  - num = ½ size, thus $\Phi(T) = 0$
- Is $\Phi(T) >= 0$ always true?
  - num >= ½ size is always true, thus $\Phi(T) >= 0$ always holds

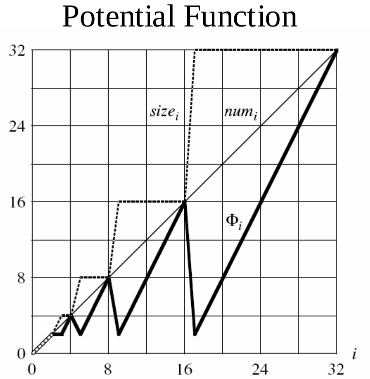
Potential Method - II

- Let num_i be the number of elements in the table after i-th operation
- Let $size_i$ be the size of the table after i-th operation
- Let Φ_i be the potential of the table after i-th operation
- Consider possible two cases after i-th operation:
- No expansion:
  - $num_i = num_{i-1} + 1$
  - $size_i = size_{i-1}$
  - $c_i = 1$
  - $ac_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$
  - $ac_i = 3$

Potential Method - III

- Expansion
  - $num_i = num_{i-1} + 1$
    - $size_{i-1} = num_{i-1} = num_i - 1$
  - $size_i = 2 \cdot size_{i-1}$
  - $c_i = num_i$
  - $ac_i = c_i + \Phi_i - \Phi_{i-1}$
    $= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$
    $= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2 \cdot (num_i - 1) - (num_i - 1))$
  - $ac_i = 3$
- In the case of only insertion, amortized cost per operation: O(1)

Expansion – Potential Function



Expansion & Contraction - I

- What if we want to free some unused space when the load factor goes below some threshold?
- α drops too low, contract the table
  - Allocate smaller table
  - Copy all items to the new table
- Constraint:
  - *A constant lower bound for α, i.e., load factor cannot go below a factor of the table size*
- Cost of elementary insertions and deletions: O(1)

Expansion & Contraction - II

- Strategy
  - Expand when α =1
  - Contract when α is going below ½
    - I.e., ½ =< α <=1
- What is the potential problem with this strategy?
  - Thrashing could occur!
  - Insert and delete right on the boundary of ½
- How do we fix this strategy?
  - ¼ =< α <=1
  - Double the size when α =1; after expansion α = ½
  - Halve the size when α = ¼ ; after contraction α = ½

Expansion & Contraction - III

- How do we ensure that we have enough potential to pay for expansion/contraction? potential function?
- $\Phi(T) =$
  - $2 \cdot num[T] - size[T]$ if $\alpha \geq 1/2$
  - $size[T]/2 - num[T]$ if $\alpha < 1/2$
- Is $\Phi(T)$ >= 0?
  - T is empty $\Rightarrow \Phi = 0$
  - $\alpha \geq$ ½ $\Rightarrow num[T] \geq$ ½ size[T] $\Rightarrow \Phi(T)$ >= 0
  - $\alpha <$ ½ $\Rightarrow num[T] <$ ½ $size[T]$ $\Rightarrow \Phi(T)$ >= 0
- Intuitively, $\Phi(T)$ measures how far from $\alpha = 1/2$ we are
  - α = ½ $\Rightarrow \Phi(T) = 2 \cdot num - 2 \cdot num = 0$
  - α = 1 $\Rightarrow \Phi(T) = 2 \cdot num - num = num$
  - α = ¼ $\Rightarrow \Phi(T) = size/2 - num = 4 \cdot num/2 - num = num$

Expansion & Contraction - III

- Cases to be analyzed for amortized cost per operation
  - Insertion/deletion
  - $\alpha \geq$ ½ or $\alpha <$ ½
  - Expansion/contraction
- <u>Case 1</u>: i-th operation is a <u>*delete*</u> operation
  - Notation: α_i denotes α after i-th operation
  - Subcase 1-1: If $\alpha_{i-1} < 1/2$ and $\alpha_i < 1/2$

Expansion & Contraction - Deletion

- Subcase 1-1 (Delete): $\alpha_{i-1} <$ ½ and $\alpha_i <$ ½
  - No contraction

    $ac_i = 1 + (size_i /2 - num_i) - (size_{i-1}/2 - num_{i-1})$

    $= 1 + (size_i /2 - num_i) - (size_i /2 - (num_i +1))$

    $= 2$
  - Contraction

    $ac_i = (num_i +1) + (size_i /2 - num_i) - (size_{i-1} /2 - num_{i-1})$

    » we have $size_i /2 = size_{i-1} /4 = num_{i-1} = num_i +1$

    $ac_i = (num_i +1) + ((num_i +1) - num_i) - ((2 \cdot num_i +2) - (num_i +1))$

    $= 1$

Expansion & Contraction - Deletion

- Case 2 (Delete): If $\alpha_{i-1} \geq 1/2$, then no contraction will be needed
- Subcase 2-1: $\alpha_i \geq 1/2$

  $ac_i = 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$

  $= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_i +2 - size_i)$

  $= -1$ (Does it make sense?)
- Subcase 2-2: $\alpha_i <$ ½
  - thus we have $num_i =<$ ½ $\cdot\, size_i - 1$

    $ac_i = 1 + (size_i /2 - num_i) - (2 \cdot num_{i-1} - size_{i-1})$

    $= 1 + (size_i /2 - num_i) - (2 \cdot num_i +2 - size_i)$

    $= -1 + 3/2 \,.\, size_i - 3 \,.\, num_i$

    $=< -1 + 3/2 \,.\, size_i - 3 \,.\, (½ \cdot size_i - 1) = 2$
- Therefore, amortized cost is =< 2

Expansion & Contraction - Insertion

- $\alpha_{i-1} \geq ½$, same analysis as before; $ac_i = 3$
- $\alpha_{i-1} < ½$, *no expansion*
- If $\alpha_{i-1} < ½$ and $\alpha_i < ½$

$ac_i = c_i + \Phi_i - \Phi_{i-1}$

$= 1 + (size_i /2 - num_i) - (size_{i-1} /2 - num_{i-1})$

$= 1 + (size_i /2 - num_i) - (size_i /2 - (num_i - 1))$

$= 0$

Expansion & Contraction - Insertion

- If $\alpha_{i-1} < ½$ and $\alpha_i >= ½$

$ac_i = 1 + (2 \cdot num_i - size_i) - (size_{i-1} /2 - num_{i-1})$

$= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1} /2 - num_{i-1})$

$= 3 \cdot num_{i-1} - 3/2 \,.\, size_{i-1} + 3$

$= 3 \cdot \alpha_{i-1} \,.\, size_{i-1} - 3/2 \,.\, size_{i-1} + 3$

< 3 (Why?)

- Therefore, amortized cost < 3