Amortized Analysis

Inge Li Gørtz

CLRS Chapter 17

Today

- Amortized analysis
  - Multipop-stack
  - Incrementing a binary counter
  - Dynamic tables

Dynamic tables

- Problem. Have to assign size of table at initialization.
- Goal. Only use space $\Theta(n)$ for an array with n elements.
- Applications. Stacks, queues, hash tables,….

- Can insert and delete elements at the end.

Dynamic tables

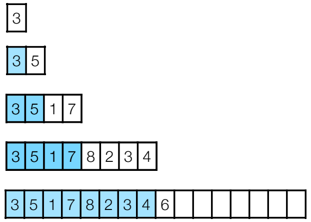
- First attempt.
  - Insert:
    - Create a new table of size n+1.
    - Move all elements to the new table.
    - Delete old table.
  - Size of table = number of elements

- Too expensive.
  - Have to copy all elements to a new array each time.
  - Insertion of N elements takes time proportional to: $1 + 2 + \cdots\cdots + n = \Theta(n^2)$.

- Goal. Ensure size of array does not change to often.

Dynamic tables

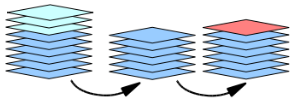
- Doubling. If the array is full (number of elements equal to size of array) copy the elements to a new array of double size.



- Consequence. Insertion of n elements take time:
  - n + number of reinsertions = n + 1 + 2 + 4 + 8 + ···· + $2^{\log n}$ < 3n.
  - Space: Θ(n).

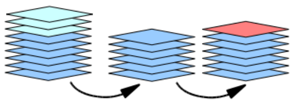
Example: Stack with MultiPop

- Stack with MultiPop.
  - Push(e): push element e onto stack.
  - MultiPop(k): pop top k elements from the stack
- Worst case: Implement via linked list or array.
  - Push: O(1).
  - MultiPop: O(k).
- Can prove total cost is no more than 2n.



Stack: Aggregate Analysis

- Amortized analysis. Sequence of n Push and MultiPop operations.
  - Each object popped at most once for each time it is pushed.
  - #pops on non-empty stack ≤ #Push operations ≤ n.
  - Total time O(n).



Binary counter

Amortized Analysis

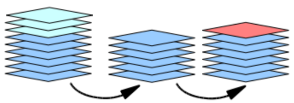
- Amortized analysis.
  - Average running time per operation over a *worst-case* sequence of operations.
- Methods.
  - Summation (aggregate) method
  - Accounting (tax) method
  - Potential method

Summation (Aggregate) method

- Summation.
  - Determine total cost.
  - Amortized cost = total cost/#operations.
- Analysis of doubling strategy (without deletions):
  - Total cost: $n + 1 + 2 + 4 + ... + 2^{\log n} = \Theta(n)$.
  - Amortized cost per insert: $\Theta(1)$.

Stack: Aggregate Analysis

- Amortized analysis. Sequence of n Push and MultiPop operations.
  - Each object popped at most once for each time it is pushed.
  - #pops on non-empty stack ≤ #Push operations ≤ n.
  - Total time O(n).
- Amortized cost per operation: 2n/n = 2.

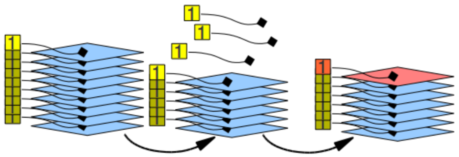


Accounting method

- Accounting/taxation.
  - Assign a cost to each type of operation that is different from actual cost.
  - c_i = cost of operation i
  - $\hat{c}_i$ = amortized cost of operation i
  - Need: $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$ for *any* sequence of n operations.
  - If $\hat{c} = c + x$, where $x > 0$: Assign the extra credit with elements in the data structure.
  - If $\hat{c} = c - x$, where $x > 0$: Use x credits stored in the data structure.

Stack: Accounting Method

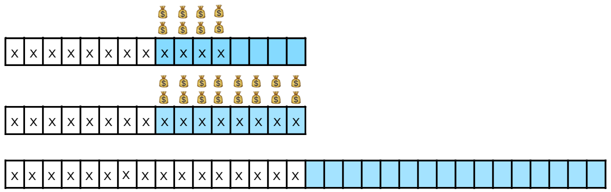
- Amortized analysis. Sequence of n Push and MultiPop operations.
  - Pay 2 credits for each Push.
  - Keep 1 credit on each element on the stack.
- Amortized cost per operation:
  - Push: 2
  - MultiPop: 1 (to pay for pop on empty stack).



Binary counter

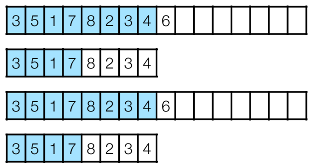
Dynamic Tables: Accounting Method

- Analysis: Allocate 2 credits to each element when inserted.
  - All elements in the array that is beyond the middle have 2 credits.
  - Table not full: insert costs 1, and we have 2 credits to save.
  - table full, i.e., doubling: half of the elements have 2 credits each. Use these to pay for reinsertion of all in the new array.
  - Amortized cost per operation: 3.



Dynamic tables with deletions

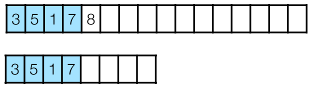
- Halving (first attempt). If the array is half full copy the elements to a new array of half the size.



- Consequence. The array is always between 50% and 100% full. **But** risk to use too much time (double or halve every time).

Dynamic tables

- Halving. If the array is a quarter full copy the elements to a new array of half the size.



- Consequence. The array is always between 25% and 100% full.

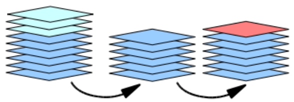
Potential method

- Potential method. Define a potential function for the data structure that is initially zero and always non-negative.
- Prepaid credit (potential) associated with the data structure (money in the bank).
- Ensure there is always enough "money in the bank" (non-negative potential).
- Amortized cost $\hat{c}_i$ of an operation: actual cost c_i plus change in potential.
  - $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
- Thus:

$$\sum_i^m \hat{c}_i = \sum_i^m \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right)$$

Stack: Potential Method

- Amortized analysis. Sequence of n Push and MultiPop operations.
  - $\Phi(S) =$ #elements on the stack
  - $S_i =$ stack after ith operation
  - $\hat{c}_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$



- Amortized cost per operation:
  - Push: $\hat{c}_i = 1 + \Phi(S_i) - \Phi(S_{i-1}) = 1 + (s+1) - s = 2$
  - Pop: $\hat{c}_i = 1 + \Phi(S_i) - \Phi(S_{i-1}) = 1 + (s-1) - s = 0$
  - Multipop(k): $\hat{c}_i = 1 + \Phi(S_i) - \Phi(S_{i-1}) = k + (s-k) - s = 0$

Binary Counter

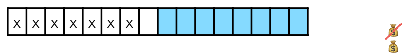
- Amortized analysis. Sequence of n increments.
  - $\Phi(B) =$ #1's in the counter
  - $B_i =$ binary counter after ith operation
  - $\hat{c}_i = c_i + \Phi(B_i) - \Phi(B_{i-1})$
- Amortized cost per increment: $t_i =$ #1's flipped to 0 in the ith operation.
  - $c_i = t_i + 1$
  - $\Phi(B_i) - \Phi(B_{i-1}) = -t_i + 1$
  - $\hat{c}_i = t_i + 1 - t_i + 1 = 2$

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Inserting when less than half full and still less than half full after insertion:



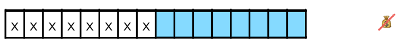
n = 7, L = 16

- amortized cost = 1 + - 💰 = 0

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Inserting when less than half full before and half full after:

n = 8, L = 16

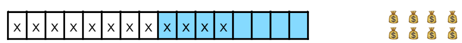


- amortized cost = 1 + - 💰 = 0

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Inserting when at least half full, but not full:

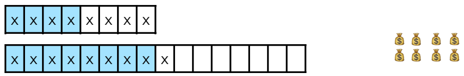
n = 12, L = 16



- amortized cost = 1 + 💰💰 = 3

Dynamic tables

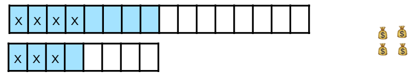
- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Inserting in full table and doubling n = 9, L = 16



- amortized cost = 9 + -💰💰💰💰💰💰 = 3

Dynamic tables

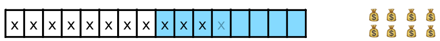
- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Deleting in a quarter full table and halving n = 3, L = 8



- amortized cost = 3 + -💰💰💰 = 0

Dynamic tables

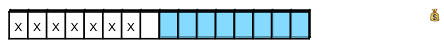
- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Deleting when more than half full (still half full after): n = 11, L = 16



- amortized cost = 1 + -💰💰 = -1

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Deleting when half full (not half full after): n = 7, L = 16

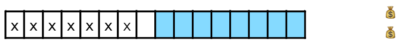


- amortized cost = 1 + 💰 = 2

Dynamic tables

- Doubling. If the table is full (number of elements equal to size of array) copy the elements to a new array of double size.
- Halving. If the table is a quarter full copy the elements to a new array of half the size
- Potential function.
  - $\Phi(D_i) = \begin{cases} 2n - L & \text{if } T \text{ at least half full} \\ L/2 - n & \text{if } T \text{ at less than half full} \end{cases}$
  - L = current array size, n = number of elements in array.
- Deleting in when less than half full (but still a quarter full after):

n = 7, L = 16



- amortized cost = 1 + 💰 = 2

Potential Method

- Summary:
  1. Pick a potential function, Φ, that will work (art).
  2. Use potential function to bound the amortized cost of the operations you're interested in.
  3. Show $\Phi(D_i) \geq 0$ for all i.
- Techniques to find potential functions: if the actual cost of an operation is high, then decrease in potential due to this operation must be large, to keep the amortized cost low.