COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 11 AMORTIZED ANALYSIS

ALPTEKİN KÜPÇÜ

Based on slides of Jennifer Welch and Cevdet Aykanat

AMORTIZED ANALYSIS

- **Key Point: The time required to perform a sequence of data structure operations is averaged over all operations performed**
- **Amortized analysis can be used to show that**
  - The average cost of an operation is small
    - if one averages over a sequence of operations
  - even though a single operation might be expensive
- **Goal: Accurately compute the total time spent in executing a sequence of operations on a data structure**

HEAP EXAMPLE

- **Some algorithms involve repeated calls to one or more data structures**
- **Example: Heapsort**
  - repeatedly insert keys into a heap
  - repeatedly remove the smallest key from the heap
- **When analyzing the running time of the overall algorithm, need to sum up the time spent in all the calls to the data structure**
- ***When different calls take different times, how can we accurately calculate the total time?***

HEAP EXAMPLE

- **Each of the *n* calls to insert into the heap operates on a heap with at most *n* elements**
- **Inserting into a heap with *n* elements takes $O(\log n)$ time**
- **So total time spent for *n* insertions is $O(n \log n)$**
- **But this is an over-estimate! (remember BUILDHEAP() analysis)**
  - Different insertions take different amounts of time
  - Many of the insertions are on significantly smaller heaps
  - Tighter analysis shows that it takes $O(n)$ time only

AMORTIZED ANALYSIS

- **Aggregate method:** **Assigns the same amortized cost to each operation by calculating total cost of n operations / n**
- **Accounting method:** **Assigns costs to each operation so that it is easy to sum them up while still ensuring that result is accurate. May assign different amortized costs to different types of operations.**
- **Potential method:** **A more sophisticated version of the accounting method. Keeps a potential function.**

AMORTIZED ANALYSIS VS. AVERAGE-CASE ANALYSIS

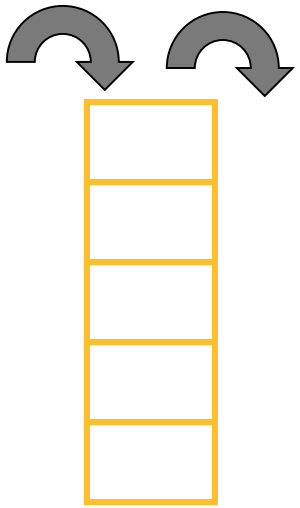
- **Amortized analysis does not use any probabilistic reasoning**
  - Average-case analysis must assume a probability distribution over the inputs
- **Amortized analysis guarantees the average performance of each operation in the worst-case**
  - Considering worst-case performance of a series of operations, what is the average per operation?

AGGREGATE METHOD

- **Show that sequence of n operations takes worst-case time T(n) in total for all n**
- **The amortized cost (average cost in the worst case) per operation is therefore $^{T(n)}/_{n}$**
- **This amortized cost applies to each operation**
  - Even when there are several types of operations in the sequence
    - e.g., if a sequence of insertions and deletions are performed, both types of operations are assigned the same amortized cost, even if they have different asymptotic complexities.

EXAMPLE #1: STACK

- **Stack Operations:**
  - Push(*S*,*x*)
  - Pop(*S*)
  - Multipop(*S*,*k*) - pop the top *k* elements
    - **while** STACK-NOT-EMPTY(S) and *k*>0 **do**
    - POP(S)
    - *k*=*k*-1



- **Implement with either array or linked list. Costs?**
  - Time for Push is O(1)
  - Time for Pop is O(1)
  - Time for Multipop is O(min(|*S*|,*k*))

EXAMPLE #1: STACK

- **Let us analyze a sequence of n POP, PUSH, and MULTIPOP operations on an initially empty stack**
- **The worst case of a MULTIPOP operation in the sequence is O(n), since the stack size is at most n**
- **Hence, a sequence of n operations costs a total of $O(n^2)$**
  - We may have **n MULTIPOP** operations each O(n)
- **The analysis is correct**
  - But not tight
- **We can obtain a better bound by using aggregate method of amortized analysis**

AGGREGATE METHOD FOR STACK

- **Key idea: total number of elements popped in the entire sequence is at most the total number of elements pushed.**
  - for all Pop calls, including the ones within Multipop
- **Maximum number of Push operations is *n***
- **So total time for entire sequence is *O(n)***
- **And amortized cost per operation is *O(n)/n = O(1)***
- ***Much tighter analysis, a huge difference in terms of cost***

EXAMPLE #2: *K*-BIT BINARY COUNTER

- **Counter Operation:**
  - increment(*A*) - add 1 (initially 0)
- **Implementation:**
  - *k*-element binary array
  - use grade school ripple-carry algorithm
  - INCREMENT(A) – Cost?
    - i ← 0
    - while i<k and A[i]=1 do
    - A[i] ← 0 (flip, reset)
    - i ← i+1
    - if i<k then
    - A[i] ← 1 (flip, set)
  - Worst-case O(*k*) algorithm
  - *n* increments will take O(*kn*) time
    - NOT TIGHT AGAIN

EXAMPLE #2: *K*-BIT BINARY COUNTER

Counter value	[7]	[6]	[5]	[4]	[3]	[2]	[1]	[0]	Incre cost	Total cost
0	0	0	0	0	0	0	0	**0**		
1	0	0	0	0	0	0	**0**	**1**	1	1
2	0	0	0	0	0	0	1	**0**	2	3
3	0	0	0	0	0	**0**	**1**	**1**	1	4
4	0	0	0	0	0	1	0	**0**	3	7
5	0	0	0	0	0	1	**0**	**1**	1	8
6	0	0	0	0	0	1	1	**0**	2	10
7	0	0	0	0	**0**	**1**	**1**	**1**	1	11
8	0	0	0	0	1	0	0	**0**	4	15
9	0	0	0	0	1	0	**0**	**1**	1	16
10	0	0	0	0	1	0	1	**0**	2	18
11	0	0	0	0	1	0	1	1	1	19

Bits that flip to achieve the next value are shaded

AGGREGATE METHOD FOR *K*-BIT COUNTER

- **In a sequence of *n* operations, not all of them will cause all *k* bits to flip**
  - bit 0 flips with every increment
  - bit 1 flips with every 2nd increment
  - bit 2 flips with every 4th increment
  - bit i flips with every 2^i-th increment
- **Total number of bit flips in *n* increment operations is**
  - $n + n/2 + n/4 + \ldots + n/2^k < 2n$
- **So total cost of the sequence is *O(n)***
- **Amortized cost per operation is *O(n)/n = O(1)***
- **Again, much tighter analysis makes a huge difference in terms of calculated cost**

ACCOUNTING METHOD

- **Assign an *amortized cost* to each operation**
- **Assignment must ensure that the sum of all the amortized costs in a sequence is at least the sum of all the actual costs**
  - remember, we want an upper bound on the total cost of the sequence (worst-case value)
- **How to ensure this property?**

ACCOUNTING METHOD

- **For each operation in the sequence:**
  - if amortized cost > actual cost then store difference as credit with an object in the data structure
  - if amortized cost < actual cost then use the stored credits to make up the difference
- **The total credit associated with the data structure must be non-negative at all times!**
  - Since it represents the amount by which the total amortized cost incurred so far exceed the total actual cost incurred so far

ACCOUNTING METHOD VS. AGGREGATE METHOD

- **Aggregate method:**
  - First analyze entire sequence
  - Then calculate amortized cost per operation
- **Accounting method:**
  - First assign amortized cost per operation
  - Check that amortized costs are valid (always non-negative credits)
  - Then compute cost of entire sequence of operations

ACCOUNTING METHOD FOR STACK

- **Assign the following amortized costs:**
  - Push: 2
  - Pop: 0
  - Multipop: 0

- **For *Push*, actual cost is 1. Store the extra 1 as a credit, associated with the pushed element.**
- **Pay for each popped element (either from *Pop* or *Multipop*) using the associated credit**

ACCOUNTING METHOD FOR STACK

- **We start with an empty stack of plates**
- **When we push a plate on the stack**
  - We use $1 to pay the actual cost of the push operation
  - We put a credit of $1 on top of the pushed plate
- **At any time point, every plate on the stack has a $1 credit on it**
  - It is a pre-payment for the cost of ***popping*** it
- **In order to *pop* a plate from the stack**
  - We take $1 of credit off the plate
  - And use it to pay the actual cost of the ***pop*** operation
  - Since the assigned amortized cost is 0

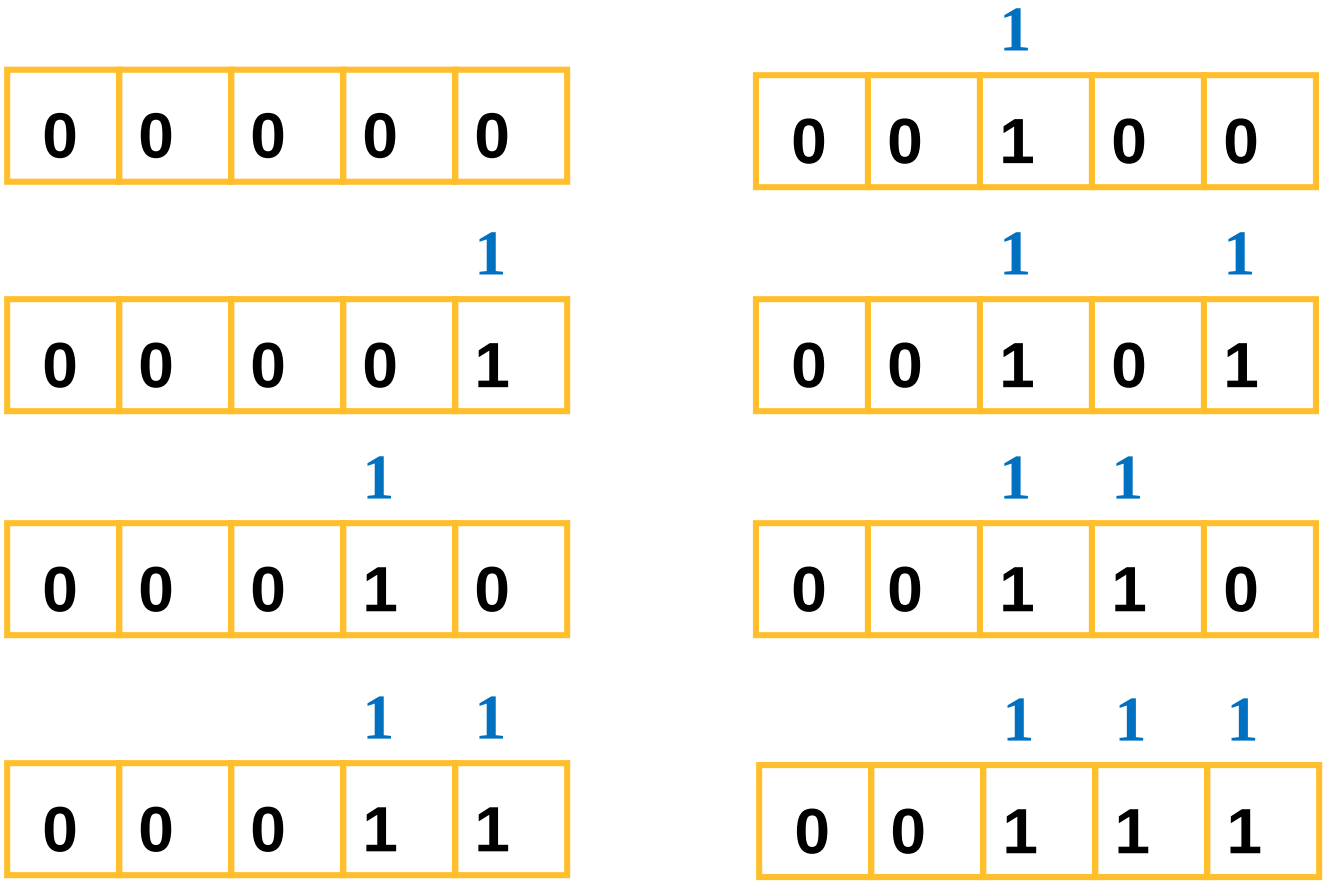
ACCOUNTING METHOD FOR STACK

- **Thus by charging the push operation a little bit more, we do not need to charge anything for the pop & multipop operations**
- **We have ensured that the amount of credits is always non-negative**
  - Since each plate on the stack always has $1 of credit
  - And the stack always has a non-negative number of plates
- **Thus, for any sequence of n push, pop, multipop operations the total amortized cost is an upper bound on the total actual cost**
- **Each amortized cost is *O(1)***
- **So total cost of entire sequence of *n* operations is *O(n)***

ACCOUNTING METHOD FOR *K*-BIT COUNTER

- **Assign amortized cost for *increment* operation to be 2.**
- **Actual cost is the number of bits flipped:**
  - a series of 1's are reset to 0
  - then a 0 is set to 1
- **Idea: $1 is used to pay for the actual cost of flipping a 0 to 1. The extra $1 credit is stored with the bit to pay for the flip back to 0.**

ACCOUNTING METHOD FOR *K*-BIT COUNTER



ACCOUNTING METHOD FOR *K*-BIT COUNTER

- **All changes from 0 to 1 are paid using $1 out of $2 allocated budget.**
  - There only one such change
  - The remaining $1 is stored as credit
- **All changes from 1 to 0 are paid using previously stored credit.**
- **Credits are always non-negative.**
- **Amortized time per operation is *O(1)***
- **Total cost of sequence is *O(n)***

POTENTIAL METHOD

- **Accounting method represents prepaid work as credit stored with specific objects in the data structure.**
- **Potential method represents the prepaid work as potential energy (or just potential) that can be released to pay for the future operations.**
- **The potential is associated with the data structure as a whole rather than with specific objects within the data structure.**

POTENTIAL METHOD

- **Define potential function Φ which maps any state of the data structure to a real number**
- **Notation:**
  - D_0 : initial state of data structure
  - D_i : state of data structure after i^{th} operation
  - c_i : actual cost of i^{th} operation
  - m_i : amortized cost of i^{th} operation

POTENTIAL METHOD

- **Define amortized cost of i^{th} operation:**
  - $m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - actual cost plus the change in the potential from previous state to current state
- **Sum of all the amortized costs is sum of all the actual costs plus $\Phi(D_n) - \Phi(D_0)$**
  - Must ensure potential difference is non-negative

POTENTIAL METHOD

- **Define amortized cost of i^{th} operation:**
  - $m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - actual cost plus the change in the potential from previous state to current state
- **Sum of all the amortized costs is sum of all the actual costs plus $\Phi(D_n) - \Phi(D_0)$**
  - Must ensure potential difference is non-negative
  - However, $\Phi(D_n) \geq \Phi(D_0)$ should hold for all possible **n** since, in practice, we do not always know **n** in advance
  - Hence if we require that $\Phi(D_i) \geq \Phi(D_0)$, for all **i** then we ensure that we **pay in advance** (as in the accounting method)
  - Usually Φ is defined so that
    - $\Phi(D_0) = 0$
    - $\Phi(D_i) \geq 0$

POTENTIAL METHOD FOR STACK

- **$\Phi(D_i)$: Number of elements in the stack after the i^{th} operation**
- **Check:**
  - $\Phi(D_0) = 0$, since stack is initially empty
  - $\Phi(D_i) \geq 0$, since a stack cannot have a negative number of elements
- **Next calculate amortized cost of each operation…**

POTENTIAL METHOD FOR STACK

- **If i^{th} operation is a Push and stack has s elements:**
  - $m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
    $= 1 + (s + 1) - s$
    $= 2 = O(1)$
- **If i^{th} operation is a Pop and stack has s elements:**
  - $m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
    $= 1 + (s - 1) - s$ OR $= 1 + 0 - 0$ (for empty stack)
    $= 0 = O(1)$ $= 1 = O(1)$
- **If i^{th} operation is a Multipop(k) and stack has s elements:**
  - $m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Let $x = \min(s,k)$
    $= x + (s - x) - s$ OR $= 1 + 0 - 0$ (for empty stack)
    $= 0 = O(1)$ $= 1 = O(1)$

POTENTIAL METHOD FOR STACK

- **All operations have *O(1)* amortized cost**
- **So total cost of the sequence is *O(n)***
- **Different potential functions may yield different amortized costs, which are still upper bounds for the actual costs**
  - The best potential function to use depends on the desired time bounds

POTENTIAL METHOD FOR *K*-BIT COUNTER

- **$\Phi(D_i)$: Number of 1's in the counter after the i^{th} *increment* operation**
- **Check:**
  - $\Phi(D_0) = 0$, since counter is initially all 0's
  - $\Phi(D_i) \geq 0$, since a counter cannot have a negative number of 1's
- **Next calculate amortized cost of the increment operation…**

POTENTIAL METHOD FOR *K*-BIT COUNTER

- Let b = number of 1's just before i^{th} operation
- Let x = number of 1's that are changed to 0 in i^{th} operation
- $m_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

  $= (x+1) + (b-x+1) - b$

  $= 2$

x 1's are changed to 0 and one 0 is changed to 1

- All operations have $O(1)$ amortized cost
- So total cost of the sequence is $O(n)$
- *Exercise: what if the counter does not start at 0 (i.e., $\Phi(D_0) \neq 0$) ?*

AMORTIZED ANALYSIS

- **Average-case analysis:**
  - Assume an input probability distribution
  - Calculate average-case running time for one algorithm (operation)
- **Randomized algorithms:**
  - Use random coins within the algorithm (not during analysis)
  - Calculate expected running time
  - No particular input distribution assumed
- **Amortized analysis:**
  - No involvement of probability
  - Average performance on a sequence of operations, even when some operation is individually expensive
  - Guarantee average performance of each operation among the sequence in the worst case