CS2351
Data Structures

Lecture 21:

Amortized Analysis

About this lecture

- Given a data structure, amortized analysis studies in a sequence of operations, the average time to perform an operation
- Introduce amortized cost of an operation
- Three Methods for the Same Purpose

  (1) Aggregate Method

  (2) Accounting Method

  (3) Potential Method (see textbook)

Super Stack

- Your friend has created a super stack, which, apart from Push/Pop, supports:

  Super-Pop(k): pop top k items

- Suppose Super-Pop never pops more items than current stack size

- The time for Super-Pop is O(k)
- The time for Push/Pop is O(1)

Super Stack

- Suppose we start with an empty stack, and we have performed n operations
  - But we don't know the order

Questions:

- Worst-case time of a Super-Pop ?

  Ans. $O(n)$ time [why?]
- Total time of n operations in worst case ?

  Ans. $O(n^2)$ time [correct, but not tight]

Super Stack

- Though we don't know the order of the operations, we still know that:
  - There are at most n Push/Pop

    → Time spent on Push/Pop = $O(n)$
  - # items popped by all Super-Pop cannot exceed total # items ever pushed into stack

    → Time spent on Super-Pop = $O(n)$

So, total time of n operations = $O(n)$!!!

Amortized Cost

- So far, there are no assumptions on n and the order of operations. Thus, we have:

For any n and any sequence of n operations,
worst-case total time = $O(n)$

- We can think of each operation performs in average $O(n) / n = O(1)$ time

  → amortized cost = $O(1)$ per operation
  (or, each runs in amortized $O(1)$ time)

Amortized Cost

- In general, we can say something like:
  - OP_1 runs in amortized $O(x)$ time
  - OP_2 runs in amortized $O(y)$ time
  - OP_3 runs in amortized $O(z)$ time

Meaning:

For any sequence of operations with

$\#OP_1 = n_1$, $\#OP_2 = n_2$, $\#OP_3 = n_3$,

worst-case total time $= O(n_1 x + n_2 y + n_3 z)$

Binary Counter

- Let us see another example of implementing a k-bit binary counter
- At the beginning, count is 0, and the counter will be like (assume k = 5):

0	0	0	0	0

which is the binary representation of the count

Binary Counter

- When the counter is incremented, the content will change
- Example: content of counter when:

0	0	1	0	1

→

0	0	1	1	0

count = 5 cost = 2 count = 6

- The cost of the increment is equal to the number of bits flipped

Binary Counter

Special case:

When all bits in the counter are 1, an increment resets all bits to 0

1	1	1	1	1

0	0	0	0	0

count = MAX   cost = k   count = 0

- The cost of the corresponding increment is equal to k, the number of bits flipped

Binary Counter

- Suppose we have performed n increments

Questions:

- Worst-case time of an increment ?

  Ans. $O(k)$ time

- Total time of n operations in worst case ?

  Ans. $O(nk)$ time [correct, but not tight]

Binary Counter

Let us denote the bits in the counter by

$$b_0, b_1, b_2, \ldots, b_{k-1},$$

starting from the right

b_4	b_3	b_2	b_1	b_0

Observation:

b_i is flipped only once in every 2^i increments

Precisely, b_i is flipped at x^{th} increment $\Leftrightarrow$ x is divisible by 2^i

Amortized Cost

- So, for n increments, the total cost is:

$$\sum_{i=0 \text{ to } k} \lfloor n / 2^i \rfloor$$

$$\leq \sum_{i=0 \text{ to } k} (n / 2^i) < 2n$$

- By dividing total cost with #increments,

➔ amortized cost of increment = $O(1)$

Aggregate Method

- The computation of amortized cost of an operation in super stack or binary counter follows similar steps:

1. Find total cost (thus, an "aggregation")
2. Divide total cost by #operations

This method is called Aggregate Method

Accounting Method

- In real life, a bank account allows us to save our excess money, and the money can be used later when needed
- We also have an easy way to check the savings

- In amortized analysis, the accounting method is very similar …

Accounting Method

- Each operation pays an amortized cost
  - if amortized cost $\geq$ actual cost, we save the excess in the bank
  - Else, we use savings to help the payment
- Often, savings can easily be checked from the objects in the current data structure

Lemma: For a sequence of operations, if we have enough to pay for each operation,

total actual cost $\leq$ total amortized cost

Super Stack (Take 2)

- Recall that apart from Push/Pop, a super stack, supports:

  Super-Pop(k): pop top k items in k time

- Let us now assign the amortized cost for each operation as follows:

  Push = \$2

  Pop or Super-Pop = \$0

Super Stack (Take 2)

Questions:

- Which operation “saves money to the bank“ when performed?
- Which operation “needs money from the bank“ when performed?
- How to check the savings in the bank ?

Super Stack (Take 2)

- Does our bank have enough to pay for each Super-Pop operation?

Ans. When Super-Pop is performed, each popped item donates its corresponding \$1 to help the payment

➔ Enough \$\$ to pay for each Super-Pop

Super Stack (Take 2)

Conclusion:

- Amortized cost of Push = 2
- Amortized cost of Pop/Super-Pop = 0

Meaning:

For any sequence of operations with

Push = n_1, # Pop = n_2, # Super-Pop = n_3,

total actual cost $\leq 2n_1$

Binary Counter (Take 2)

- Let us use accounting method to analyze increment operation in a binary counter, whose initial count = 0

0	0	0	0	0

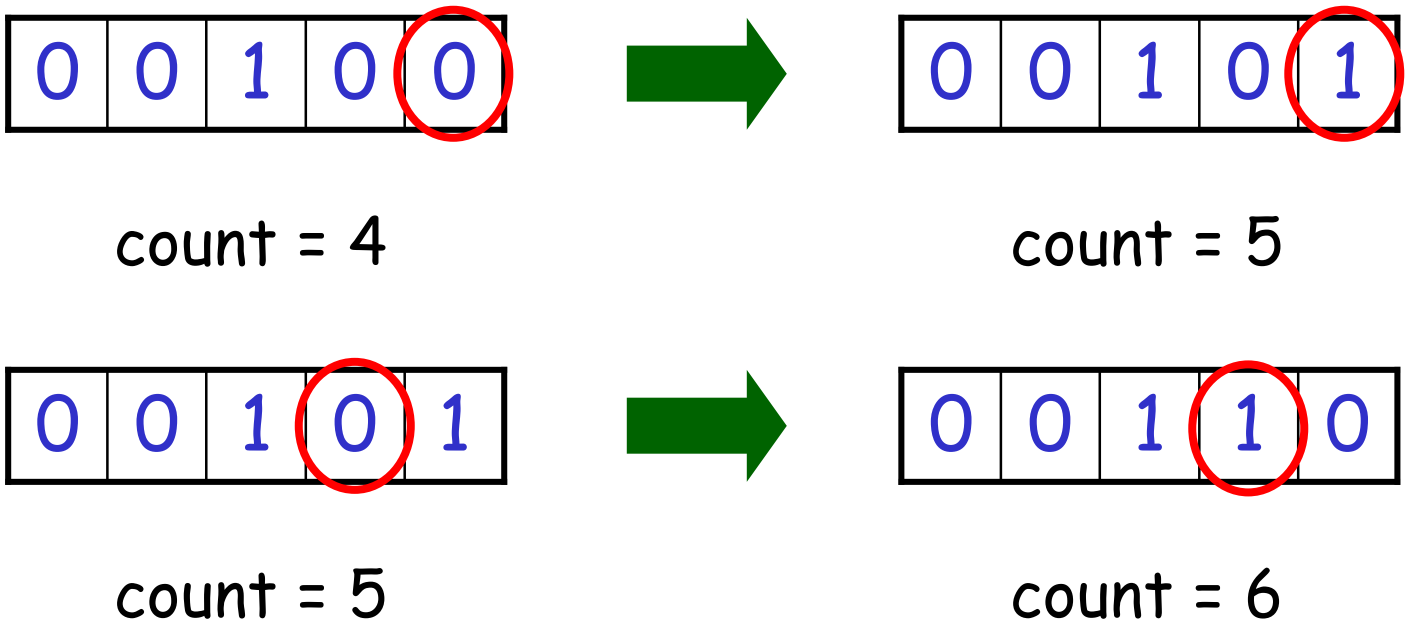
- We assign amortized cost for each increment = \$2

- Recall: actual cost = #bits flipped

Binary Counter (Take 2)

Observation: In each increment operation, at most one bit is set from 0 to 1 (whereas the following bits are set from 1 to 0).

E.g.,



Binary Counter (Take 2)

Lemma: Savings = # of 1's in the counter

Proof: By induction

To show amortized cost = \$2 is enough,

- we use \$1 to pay for flipping some bit x from 0 to 1, and store the excess \$1
- For other bits being flipped (from 1 to 0), each donates its corresponding \$1

➔ Enough to pay for each increment

Binary Counter (Take 2)

Conclusion:

- Amortized cost of increment = 2

Meaning:

For n increments (with initial count = 0)

total actual cost $\leq 2n$

Question: What's wrong if initial count $\neq 0$?

Accounting Method (Remarks)

- In contrast to the aggregate method, the accounting method may assign different amortized costs to different operations
- Another thing: To help the analysis, we usually link each excess \$ to a specific object in the data structure (such as an item in a stack, or a bit in a binary counter)

  → called the credit stored in the object