Indian Institute of Information Technology
Design and Manufacturing, Kancheepuram
Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 501 Advanced Data Structures and Algorithms

Instructor
N.Sadagopan
Scribe:
Pranjal Choubey
Renjith.P

Amortized Analysis

Objective: In this lecture, we shall present the need for amortized analysis, and case studies involving amortized analysis.

Motivation: Consider data structures Stack, Binomial Heap, Min-Max Heap; stack supports operations such as push, pop, multipush and multipop, and heaps support operations such as insert, delete, extract-min, merge and decrease key. For data structures with many supporting operations, can we look for an analysis which is better than classical asymptotic analysis. Can we look for a micro-level analysis to get a precise estimate of cost rather than worst case analysis.

1 Amortized Analysis

Amortized analysis is applied on data structures that support many operations. The sequence of operations and the multiplicity of each operation is application specific or the associated algorithm specific. Classical asymptotic analysis gives worst case analysis of each operation without taking the effect of one operation on the other, whereas amortized analysis focuses on a sequence of operations, an interplay between operations, and thus yielding an analysis which is precise and depicts a micro-level analysis.

Since many operations are involved as part of the analysis, the objective is to perform efficiently as many operations as possible, leaving very few costly operations (the time complexity is relatively more for these operations). To calculate the cost of an operation or the amortized cost of an operation, we take the average over all operations. In particular, worst case time of each operation is taken into account to calculate the average cost in the worst case. Some of the highlights of amortized analysis include (reader is advised to revisit this section after learning the techniques of amortized analysis);

- Amortized Analysis is applied to algorithms where an occasional operation is very slow, but most of the other operations are faster.
- In Amortized Analysis, we analyze a sequence of operations and guarantee a worst case average time which is lower than the worst case time of a particular expensive operation.
- Amortized analysis is an upper bound: it is the average performance of each operation in the worst case. Amortized analysis is concerned with the over all cost of a sequence of operations. It does not say anything about the cost of a specific operation in that sequence.
- Amortized analysis can be understood to take advantage of the fact that some expensive operations may pay for future operations by somehow limiting the number or cost of expensive operations that can happen in the near future.
- Amortized analysis may consist of a collection of cheap, less expensive and expensive operations, however, amortized analysis (due to its averaging argument) will show that average cost of an operation is cheap.
- This is different from average case analysis, wherein averaging argument is given over all inputs for a specific operation. Inputs are modeled using a suitable probability distribution. In amortized analysis, no probability is involved.

2 The Basics: the three approaches to amortized analysis

It is important to note that these approaches are for analysis purpose only. The underlying algorithm design is unaltered and the purpose of these micro-level analysis is to get a good insight into the operations being performed.

- **Aggregate Analysis:** Aggregate analysis is a simple method that involves computing the total cost $T(n)$ for a sequence of n operations, then dividing $T(n)$ by the number n of operations to obtain the amortized cost or the average cost in the worst case. For all operations the same amortized cost $T(n)/n$ is assigned, even if they are of different types. The other two methods may allow for assigning different amortized costs to different types of operations in the same sequence.

- **Accounting Method:** As part of accounting method we maintain an account with the underlying data structure. Initially, the account contains '0' credits (charges). When we perform an operation, we charge the operation, and if we over charge an operation, the excess charge will be deposited to the account as credit. For some operation, we may charge nothing, in such a case, we make use of charges available at credit. Such operations are referred to as *free* operations. Analysis ensures that the account is never at debit (negative balance). This technique is good, if for example, there are two operations O_1 and O_2 which are tightly coupled, then O_1 can be over charged and O_2 is free. Typically, the charge denotes the actual cost of that operation. The excess charge will be stored at objects (elements) of a data structure.
  In the accounting method, the amount charged for each operation type is the amortized cost for that type. As long as the charges are set so that it is impossible to go into debt, the amortized cost will be an upper bound on the actual cost for any sequence of operations. Therefore, the trick to successful amortized analysis with the accounting method is to pick appropriate charges and show that these charges are sufficient to allow payment for any sequence of operations.

- **Potential Function Method:** Here, the analysis is done by focusing on structural parameters such as the number of elements, the height, the number of property violations of a data structure. For an operation, after performing the operation, the change in the structural parameter is captured using a function which is maintained at the data structure. The function that captures the change is known as a *potential function*. As part of the analysis, we work with non-negative potential functions. If the change in potential is positive, then that operation is over charged and similar to accounting method, the excess potential will be stored at the data structure. If the change in potential is negative, then that operation is under charged which would be compensated by excess potential available at the data structure.
  Formally, let c_i denote the actual cost of the i^{th} operation and $\hat{c}_i$ denote the amortized cost of the i^{th} operation. If $\hat{c}_i > c_i$, then the i^{th} operation leaves some positive amount of credits, the credits $\hat{c}_i - c_i$ can be used up by future operations. And as long as

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \qquad (1)$$

  the total available credit will always be non negative, and the sum of amortized costs will be an upper bound on the actual cost.
  That is, the potential function method defines a function that maps a data structure onto a real valued non-negative number. In the potential method, the amortized cost of operation i is equal to the actual cost plus the increase in potential due to that operation:

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1} \qquad (2)$$

  From equation 1 and 2:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \phi_i - \phi_{i-1}) \qquad (3)$$

$$\sum_{i=1}^{n} \hat{c}_i = \langle \sum_{i=1}^{n} c_i \rangle + \phi_n - \phi_0 \qquad (4)$$

Since $\phi_0 = 0$ and $\phi_n \geq 0$, $\phi_n - \phi_0 \geq 0$,

$$\sum_{i=1}^{n} \hat{c}_i = \langle \sum_{i=1}^{n} c_i \rangle + \phi_n - \phi_0 \geq \sum_{i=1}^{n} c_i \tag{4}$$

Thus it follows that,

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

.

3 Case Studies

We shall present three case studies to explain each analysis technique in detail.

3.1 Stack

To begin with, we shall consider stack with two supporting operations, namely push() and pop(). Subsequently, our stack implementation shall consist of four operations, namely push(), pop(), multi-push() and multi-pop(). The operation multi-pop(k) fetches the top k elements of the stack if stack contains at least k elements. If stack contains less than k elements, then output the entire stack.

Stack with operations push() and pop()

Let $Y = (O_1, \ldots, O_y)$ be a sequence of operations performed on a stack such that each O_i is push/pop. We assume that pop is performed on an non-empty stack. For analysis, we need to fix n, the number of elements in the stack. Since Y is arbitrary, n must be carefully chosen, otherwise it may lead to incorrect analysis. We let n to denote the number of push operations in Y, i.e., the number of elements inserted through push operations. Clearly, the number of pop operations cannot exceed n. We shall now show that for any sequence of y operations, the amortized costs of push and pop is $O(1)$.

- `Aggregate Analysis :` Suppose in Y, there are l push operations, then the total cost is $\frac{l \cdot O(1) + (y-l) \cdot O(1)}{y} = O(1)$
  Thus, as per aggregate analysis, we assign the same cost to all operations, which is the average cost in worst case. Therefore, the cost of push and pop is $O(1)$ amortized.

- `Accounting method :` As part of accounting method, we need to assign credits to objects of stack S. Whenever we push an element x into S, we charge '2' credits. One credit will be used for pushing the element which is the actual cost for pushing x into S and the other credit is stored at x which would be used later. For pop operation, we charge nothing, i.e., pop operation is free. Although, the actual cost is '1', in our analysis we charge '0'. The excess credit '1' available at the element will be used when we perform a pop operation. Since pop is performed on a non-empty stack, the account will always be at credit. Thus, the amortised cost of push is $2 = O(1)$ and pop is $0 = O(1)$.

- `Potential Function Method:` The first thing in this method is to define a potential function capturing some structural parameter. A natural structural parameter of interest for stack is the number of elements. Keeping the number of elements as the potential function, we now analyze all three operations.

  1. Push
     $\hat{c}_{push} = c_{push} + \phi_i - \phi_{i-1}$

$=1 + x + 1 - x$, where x denotes the number of elements in S before push operation.
$=2$

2. Pop
$\hat{c}_{pop} = c_{pop} + \phi_i - \phi_{i-1}$
$=1 + x - 1 - x$
$=0$

Thus, as per potential function method, both push and pop are constant amortized. It is important to highlight the fact that, aggregate analysis focuses on the entire sequence of operations; the average cost in the worst case for the sequence, the average cost obtained is precisely the amortized cost. For all operations, the same amortized cost is assigned, unlike other two techniques. In accounting and potential function techniques, amortized cost is calculated for each distinct operation.

Stack with supporting operations push, pop and multi-pop

Let $Y = (O_1, \ldots, O_y)$ be a sequence of operations performed on a stack such that each O_i is push/pop/multi-pop. We assume that pop/multi-pop is performed on an non-empty stack. For analysis, we need to fix n, the number of elements in the stack. Since Y is arbitrary, n must be carefully chosen, otherwise it may lead to incorrect analysis. We let n to denote the number of push operations in Y, i.e., the number of elements inserted through push operations. Clearly, the number of elements popped out of pop and multi-pop operations cannot exceed n. We shall now show that for any sequence of y operations, the amortized costs of push, pop and multi-pop are $O(1)$.

- `Aggregate Analysis:` Suppose in a sequence of y operations, there are l_1 push, l_2 pop operations, and $(y - l_1 - l_2)$ multi-pop operations. The actual cost of push and pop is $O(1)$. The trivial actual cost of multi-pop is $O(n)$. A simple aggregate analysis tells us that amortized costs of all three operations are $O(n)$. That is,
$\frac{l_1 \cdot O(1) + l_2 \cdot O(1) + (y - l_1 - l_2) \cdot O(n)}{y}$
$\leq \frac{l_1 \cdot O(n) + l_2 \cdot O(n) + (y - l_1 - l_2) \cdot O(n)}{y}$
$= \frac{O(ny)}{y} = O(n)$
Thus, all three operations incur $O(n)$ amortized. We shall now present a tighter analysis using which we can show that all three operations can be performed in $O(1)$ amortized. Note that the cumulative cost of multi-pop operations in the sequence Y cannot exceed $O(1) \cdot (l_1 - l_2)$ as the total number of elements popped out over all multi-pops cannot be more than $(l_1 - l_2)$. Therefore,
$\frac{l_1 \cdot O(1) + l_2 \cdot O(1) + O(1) \cdot (l_1 - l_2)}{y}$
$= \frac{2l_1}{y}$. Since $y \geq l_1$,
$= O(1)$ amortized. Thus, as per aggregate analysis, all three operations incur $O(1)$ amortized costs. The worst case happens when there are $y = n+1$ operations with n push followed by one multi-pop(n). The cost in such a scenario is $\frac{n+n}{n+1} \leq 2 = O(1)$. Since aggregate analysis focuses on the average cost in the worst case, the above scenario maximizes the actual cost (numerator) with the minimum number of operations (denominator).

- `Accounting Method:` Similar to earlier discussion, we assign 2 credits for push and 0 credits for pop. Even for multi-pop, we charge nothing as sufficient credits are available at elements themselves. Thus, the amortised cost of push is $2 = O(1)$, pop is $0 = O(1)$ and multi-pop is $0 = O(1)$.

- `Potential Function Method:` As before, the potential function denotes the number of elements in the stack. The costs for push is 2 and pop is 0.
  Multi-pop(k)
  $\hat{c}_{mpop} = c_{mpop} + \phi_i - \phi_{i-1}$
  $=k + x - k - x$, where $x = |S|$.
  $=0$
  Thus, the amortized costs of all three operations are $O(1)$.

Stack with operations push, pop, multi-pop and multi-push

Let us fix n to be the number of elements inserted through push and multi-push operations.

- `Aggregate Analysis:` The actual cost of multi-push(k) is k, multi-pop(k') is k' and for the other two is constant. Therefore, the amortized cost is

  $$\frac{l_1 \cdot O(1) + l_2 \cdot O(1) + l_3 \cdot O(?) + (y - l_1 - l_2 - l_3) \cdot O(?)}{y}$$

  Note for multi-push() and multi-pop() the cost depends on k and k', moreover the value of k and k' varies for each multi-push and multi-pop operation. A trivial analysis tells us that the cost of multi-push and multi-pop cannot exceed n, and thus we get,

  $$\frac{l_1 \cdot O(1) + l_2 \cdot O(1) + l_3 \cdot O(n) + (y - l_1 - l_2 - l_3) \cdot O(n)}{y}$$

  $$\leq \frac{l_1 \cdot O(1) + l_2 \cdot O(1) + l_3 \cdot O(k') + (y - l_1 - l_2 - l_3) \cdot O(k)}{y}$$

  where $k' = \max(q \mid \text{ multi-pop}(q) \text{ in } Y)$ and $k = \max(r \mid \text{ multi-push}(r) \text{ in } Y)$

  $$\leq \frac{l_1 \cdot O(n) + l_2 \cdot O(n) + l_3 \cdot O(n) + (y - l_1 - l_2 - l_3) \cdot O(n)}{y}$$

  $= O(n)$.
  Surprisingly, the tighter analysis also reveals that the amortized cost is $O(n)$. Let K be number of elements inserted into the stack through push and multi-push operations, then the cost of pop and multi-pop cannot exceed K. Therefore, the amortized cost is $\frac{K+K}{y}$. For the worst case scenario, $K = O(n)$ and $y = O(1)$.
  That is, n elements are inserted into the stack through push and multi-push, then the cost of pop and multi-pop cannot exceed n.
  Amortized cost= $\frac{O(n)+O(n)}{y}$. For the worst case scenario to happen, y must be constant. Thus, we get, $O(n)$ amortized. The sequence (multi-push($\frac{n}{2}$), multi-push($\frac{n}{2}$)) has just 2 operations with cost $O(n)$. Thus, Amortized cost= $\frac{O(n)+O(n)}{2} = O(n)$. Therefore, as per aggregate analysis, all operations incur $O(n)$ amortized.

- `Accounting Method:` For push, assign 2 credits, and pop and multi-pop(k'), assign '0' credits. We assign $2k$ credits for multi-push(k). Out of $2k$ credits, k credits are used for multi-push operation and the other k credits are stored with the stack itself. Since $k = O(n)$, the cost of multi-push() is $O(n)$ amortized, whereas, for the other three it is $O(1)$ amortized.

- `Potential Function Method:` We shall analyze multi-push(k) and for the rest the analysis is same as before.
  $\hat{c}_{mpush} = c_{mpush} + \phi_i - \phi_{i-1}$
  $=k + x + k - x$
  $=2 \cdot k = O(n)$.

Note the similarity between potential function method and the accounting method. For push, the amortized cost is '2' which is also the credit assigned by the accounting method. The potential function, in particular,

the change in potential gives us the excess credit stored at each element of S. One can also use potential functions such as $2 \cdot |S|$ or $c \cdot |S|$, where c is a constant, and still obtain $O(1)$ amortized cost for the basic scheme. The excess potential will be simply stored at the data structure as credits. Similarly, in accounting method, if we charge, say 4 credits for push, then excess credits will be stored at elements.

3.2 Binary Counter

Consider a binary counter on k bits which is initially set to all 0's. The primitive operations are Increment, Decrement and Reset. Increment on a counter adds a bit '1' to the current value of the counter. Similarly, decrement on a counter subtracts a bit '1' from the current value of the counter. For example, if counter contains '001101', on increment, the content of the counter is '001110' and on decrement, the result is '001100'. The reset operation makes the counter bits all 0's. The objectives here are to analyze amortized costs of (i) a sequence of n increment operations, (ii) a sequence of n increment and decrement operations, (iii) a sequence of n increment, decrement and reset operations. For analysis, our primitive operation is flipping a bit '0' to '1' or vice versa.

Amortized analysis: a sequence of n increment operations

Given a binary counter on k-bits, we shall now analyze the amortized cost of a sequence of n increment operations. An illustration is given in Figure 1. A trivial analysis shows $O(k)$ for each increment and for n increment operation, the total cost is $O(nk)$ and the average cost is $O(k)$. We now present a micro-level analysis using which we show that the amortized cost is $O(1)$. Note that not all bits in the counter flip in each increment operation.

The 0^{th} bit flips in each increment and there are $\lfloor n \rfloor$ flips. The 1^{st} bit is flipped alternately and thus $\lfloor \frac{n}{2} \rfloor$ flips in total. The i^{th} bit is flipped $\lfloor \frac{n}{2^i} \rfloor$ times in total. It is important to note that, for $i > \lfloor \log n \rfloor$, bit $A[i]$ never flips at all. The total number of flips in a sequence of n increments is thus

$$\sum_{i=0}^{\lfloor \log n \rfloor} \lfloor \tfrac{n}{2^i} \rfloor \quad < n \sum_{i=0}^{\infty} \tfrac{1}{2^i} = 2n$$

The worst-case time for a sequence of n increment operations on an initially zero counter is therefore $O(n)$. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$. This completes the argument for aggregate analysis.

As part of `accounting method`, we assign '2' credits with each bit when it is flipped from 0 to 1. '1' credit will be used for the actual flip and the other '1' credit will be stored at the bit itself. When a bit is flipped from 1 to 0 in subsequent increments, it is done for free. The credit '1' stored at the bit '1' will actually pay for this operation. At the end of each increment, the number of 1's in the counter is the credit accumulated at the counter. The primitive operations are flipping a bit from 1 to 0 and 0 to 1. The former charges 2 credits which is $O(1)$ and the latter charges 0 credits which is also $O(1)$. Thus, the amortized cost of increment is $O(1)$.

For `potential function` method, the structural parameter of interest is the number of 1's in the counter. During i^{th} iteration all ones after the last zero is set to zero and the last zero is set to 1. For example, when increment is called on a counter with its contents being '11001111', the result is '11010000'. Let x denotes the total number of 1's before the i^{th} operation and t denote the number of ones after the last zero. At the end of i^{th} operation there will be $x - t + 1$ ones, t ones are changed to 0 and the last zero is changed to 1. Thus, the actual cost for the increment is $1 + t$. Note that the parameter t is a local variable introduced as part of micro-level analysis and does not affect the potential function defined already. Therefore, the amortized cost is

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 1: Source: CLRS, An 8-bit binary counter where the value goes from 0 to 16 by a sequence of 16 increment operations. Bits that flip to achieve the next value are shaded.

$\hat{c_i} = c_i + \phi_i - \phi_{i-1}$
$=1 + t + (x - t + 1) - x$
$=2$
$=O(1)$

Remark: Amortized analysis for a sequence of n decrement operations with base value of the counter all 1's, is similar to the analysis presented above and hence, a sequence of n decrements incur $O(1)$ amortized.

Amortized analysis: a sequence of n increment and decrement operations

- `Aggregate Analysis:` Suppose in a sequence $Y = (O_1, \ldots, O_y)$, there are l increments, then the average cost in the worst case is
  $\frac{l \cdot O(?) + (y - l) \cdot O(?)}{y}$
  The cost for increment/decrement varies depending on the configuration of the binary counter. However, for the worst case scenario with k being the size of the counter, the following scenario can happen. For example, $k = 8$ and for the current configuration 10000000, on increment, the configuration is changed to 01111111; the cost of this increment is $O(k)$. Similarly, when we perform decrement on counter with the configuration 01111111, then we incur $O(k)$ effort to obtain the configuration 10000000. Thus, amortized cost is
  $\frac{l \cdot O(k) + (y - l) \cdot O(k)}{y} = O(k)$
  Therefore, the amortized costs for increment and decrement in a sequence of n increments and decrements is $O(k) = O(\log n)$ amortized.
  `Worst case scenario:` Consider a sequence having $\frac{n}{2}$ increments followed by $\frac{n}{2}$ alternating increments and decrements. As far as aggregate analysis is concerned, the actual cost for the first half of the operations is $O(n)$, whereas the second half incurs $\frac{n}{2} \cdot O(k)$. Thus, the amortized cost is
  $\frac{O(n) + \frac{n}{2} \cdot O(k)}{\frac{n}{2} + \frac{n}{2}} = O(k) = O(\log n)$.

- `Accounting Method:` As part of accounting method, we assign 2 credits for increment and $2k = O(k)$ credits for decrement operations. As per our credit assignment scheme, we ensure for any configuration of the counter, if the value of the bit is '1' then there is a credit '1' associated with it which can be used later for increment/decrement operation. However, if the value of the bit is '0', then there is no credit associated with it. Therefore, decrement operations must be sufficiently charged so that it pays for the operation and also it places credit '1' at positions where the value is '1'. Due to this reason, we charge at most $2k$ credits for decrement; k credits for the operation and another k credit will be placed at the counter.
  Consider the following configuration; 10...010...0, let i be the position of the rightmost '1', then for decrement we need $2(i-1)$ credits : $(i-1)$ credits for flipping '0' to '1' and the other $(i-1)$ credits will be placed at the last $(i-1)$ 1's. Thus, the total credits assigned is $2(i-1) \leq 2k = O(k)$. The actual credits assigned depends on the configuration of the binary counter. Therefore, as per accounting method, a sequence of y increment and decrement operations incur $O(1)$ amortized for increment and $O(k) = O(\log n)$ amortized for decrement.

- `Potential function method` The function refers to the number of 1's in the counter. For increment operation, similar to the above section, let x and t denote the number of 1's in the counter and the number of 1's after the rightmost zero in the configuration. Thus, the amortised cost of increment is
  $\hat{c_i} = c_i + \phi_i - \phi_{i-1}$
  $=1 + t + (x - t + 1) - x$
  $=2$
  $=O(1)$
  For decrement, let x and t denote the number of 1's in the counter and the number of 0's after the rightmost one in the configuration. Thus, the amortised cost of decrement is
  $\hat{c_i} = c_i + \phi_i - \phi_{i-1}$
  $=1 + t + (x + t - 1) - x$
  $=2t$
  $=O(k)$
  Thus, the amortized cost of increment is $O(1)$ and that of decrement is $O(k) = O(\log n)$.

Amortized analysis: a sequence of n increment, decrement and reset operations

- `Aggregate Analysis:` Suppose in a sequence Y, there are l_1 increment, l_2 decrement and the rest are reset operations. Then, the amortized cost is
  $$\frac{l_1 \cdot O(?) + l_2 \cdot O(?) + (y - l_1 - l_2) \cdot O(?)}{y}$$
  Since we are interested in average cost in the worst case; the following scenario is one such example meeting our requirements.
  $$\frac{O(n) + l_2 \cdot O(k) + (y - l_1 - l_2) \cdot O(k)}{n} = O(k)$$
  An example scenario: a sequence of $\frac{n}{2}$ increments, followed by an alternating increment and decrement for $\frac{n}{2}$ times and finally a reset operation. The cost of the first $\frac{n}{2}$ increments is $O(n)$, the next set incurs $\frac{n}{2} \cdot O(k)$ and the reset incurs $O(k)$. The amortized cost is
  $\frac{O(n) + O(n) \cdot O(k) + O(k)}{n+1} = O(k)$. Therefore, for any input sequence increment, decrement and reset operations, the amortized costs of all three operations are $O(k) = O(\log n)$.

- `Accounting Method:` We charge '2' credits for increment, $2k$ credits for decrement, and '0' credits for reset. Since, in any configuration of the binary counter, the bit '1' has a credit associated with it, the reset operation is charged nothing. Amortized costs of increment is $O(1)$, decrement is $O(k) = O(\log n)$ and the reset is $O(1)$.

- `Potential Function Method:` Analysis of increment and decrement are same as before. For the reset operation;
  the actual cost is x, the number of 1's in the counter.
  $\hat{c_i} = c_i + \phi_i - \phi_{i-1}$
  $= x + 0 - x$
  $= O(1)$
  Thus, as per potential function based analysis, amortized costs of increment, decrement, and reset are $O(1), O(k), O(1)$, respectively.