Amortized Analysis

- Amortized Analysis is applicable for some special set of algorithms and solutions.
- We can not apply for this analysis for every solutions
- One input is changing the running time of another set of inputs
- In a sequence of n operations, some inputs are taking more time, other inputs are taking less time. If one input is affecting the running time of another set of operations or inputs, we go for amortized analysis.

- Amortized analysis is not average case analysis.
- In average case analysis we use probability, but probability is not involved in amortized analysis.
- But, we use the term average cost in amortized analysis.
- In amortized analysis, the time required to perform a sequence of data structure operation is average over all the operations performed.

There are three types of Amortized Analysis:

1. Aggregate Analysis

2. The Accounting Method

3. The Potential Method

Aggregate Analysis

In aggregate analysis for all n, a sequence of n operations takes worst case time T(n) in total. In the worst case the average cost or amortized cost per operation.

$$\text{Cost} / \text{Operation} = T(n) / n$$

Stack Example

A common example of aggregate analysis is a modified stack. Stacks are a linear data structure that have two constant-time operations. `push(element)` puts an element on the top of the stack, and `pop()` takes the top element off of the stack and returns it. These operations are both constant-time, so a total of n operations (in any order) will result in $O(n)$ total time.

Consider a stack S that has the following operations:

- Push(S, x) pushes an element x onto the stack S
- Pop(S) pops the top element x of S and returns x
- Multipop(S, k) pops the min(k, |S|) top elements of S

Running time: Push(S,x) is O(1), Pop(S) is O(1), Multipop(S,k) is O(min(k,|S|))

Now, a new operation is added to the stack. `multipop(k)` will either pop the top k elements in the stack, or if it runs out of elements before that, it will pop all of the elements in the stack and stop. The pseudo-code for `multipop(k)` would look like this:

```
multipop(k):
    while stack not empty and k > 0:
        k = k - 1
         stack.pop()
```

Looking at the pseudo-code, it's easy to see that this is not a constant-time operation. `multipop` can run for at most n times, where n is the size of the stack. So, the worst-case runtime for `multipop` is $O(n)$. So, in atypical analysis, that means that n `multipop` operations take $O(n^2)$ time.

However, that's not actually the case. Think about `multipop` and what it's actually doing. `multipop` cannot function unless there's been a push to the stack because it would have nothing to pop off. In fact, any sequence of n operations of `multipop`, `pop` and `push` can take at most $O(n)$ time. `multipop`, the only non-constant-time operation in this stack, can only take $O(n)$ time if there have also been n constant-time `push` operations on the stack. In the very worst case, there are n constant-time operations and just 1 operation taking $O(n)$ time.

For any value of n, any sequence of `multipop`, `pop`, and `push` takes $O(n)$ time. So, using aggregate analysis,

$$T(n)/n = O(n)/n = O(1)$$

So, this stack has an amortized cost of $O(1)$ per operation.

Binary Counter

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

```
Increment (A,k)
{
      int i = 0

      while ( i < k && A[ i ] == 1 )
      {
            A[ i ] = 0
            i = i + 1
      }

      If ( i < k)
      {
            A[ i ] = 1
      }
}
```

No of bits in the array = $k = 4$

No of Sequences = $n = 16 = 2^4 = 2^k$

Therefore, $k = \text{Log } n$

A[0] Flips n times

A[1] flips n/2 times

A[2] flips n/4 times

A[i] flips $n/2^i$ times

i=0,1,2,…..log n

For i>log n, bit A[i] never flips

(n=16,   log 16 base2=4)

$$\sum_{i=0}^{\lfloor log\, n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \times \sum_{i=0}^{\infty} \frac{1}{2^i} < n \times \frac{1}{1-\frac{1}{2}} < 2n$$

$2n = O(n)$

cost of performing n increment operation $=O(n)$
cost of one operation $=O(n)/n = O(1)$

Aggregate Analysis $= O(n) / n = O(1)$

Accounting Method

- Assign different charges to different operations.
- The amount of the charge is called amortized cost.
- amortized cost can be more or less than actual cost.
- When

  amortized cost > actual cost,

  the difference is saved in specific objects of the Data Structure as credits.
- The credits can be used by later operations whose

  amortized cost < actual cost.

• In aggregate analysis, all operations have same amortized costs.

Accounting Method

- suppose **actual cost** is c_i for the *i*th operation in the sequence, and **amortized cost** is c_i',
- $\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$
  - **amortized cost** is an upper bound of total **actual cost**.
  - holds for all sequences of operations.

- Current credit can never be negative

Accounting Method: Stack Operations

- Actual costs:
  - PUSH :1, POP :1, MULTIPOP: min(s,k).
- Let assign the following amortized costs:
  - PUSH:2, POP: 0, MULTIPOP: 0.

Accounting Method: Stack Operations

- PUSH(x)
  - 1 credit is used to pay the Push operation
  - The other credit is stored in object(x)

- POP(x) - MULTIPOP
  - Whenever an element is removed, its credit is used to pay for the POP operation

Accounting Method: Stack Operations

- At any point of time, the total of credits is the total number of elements in the stack. It cannot be negative
- Total cost<=The total amortized cost <=2n

Stack Operations

Operation	Amortized Cost	Actual Cost	Credit
Push (A, S)	2	1	1
Push (B, S)	2	1	2
Pop (S)	0	1	1
Push (C, S)	2	1	2
Pop (S)	0	1	1
Pop (S)	0	1	0
Push (D, S)	2	1	1
Pop (S)	0	1	0
Total Amortized Cost	8		

Amortize Analysis : Stack Operations

Analysis of the stack operation:

- Total amortized cost = (2+2+0+2+0+0+2+0)
- Total amortized cost = 8 (for 4 Push operations)
- Total amortized cost = 2*4 (for 4 Push operations)
- Total amortized cost = 2*n (for n Push operations)
- T(n) = 2n
- **T(n) = O(n)**

Accounting method: binary counter

Charge an amortized cost of \$2 to set a bit from 0 to 1 and \$0 to set a bit from 1 to 0

Whenever a bit is set from 0 to 1, use \$1 to pay the actual cost, and store another \$1 on the bit as credit.

When a bit is reset (1 to 0), the stored \$1 pays the actual cost.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Actual 1-->0 (1)	Amo 1-->0 (0)	Amo 0-->1 (2)	credit
0	0	0	0	0	0	0	0	0				
1	0	0	0	0	0	0	0	1			2	1
2	0	0	0	0	0	0	1	0	1	0	2	1
3	0	0	0	0	0	0	1	1			2	2
4	0	0	0	0	0	1	0	0	2	0	2	1
5	0	0	0	0	0	1	0	1			2	2
6	0	0	0	0	0	1	1	0	1	0	2	2
7	0	0	0	0	0	1	1	1			2	3
8	0	0	0	0	1	0	0	0	3	0	2	1
9	0	0	0	0	1	0	0	1			2	2
10	0	0	0	0	1	0	1	0	1	0	2	2
11	0	0	0	0	1	0	1	1			2	3
12	0	0	0	0	1	1	0	0	2	0	2	2
13	0	0	0	0	1	1	0	1			2	3
14	0	0	0	0	1	1	1	0	1	0	2	3
15	0	0	0	0	1	1	1	1			2	4
16	0	0	0	1	0	0	0	0	4	0	2	1

Total Amortized cost =32 (16 increments)

Total Amortized cost =2 x 16 (16 increments)

Total Amortized cost =2 x n (n increments)

T(n) = 2n

T(n) = O(n)

The Potential Method

- Same as accounting method: something prepaid is used later.
- Different from accounting method
  - The prepaid work not as credit, but as “potential energy”, or “potential”.

The Amortized cost c_i' of the i th operation with respect to potential function Φ is defined by :

$$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

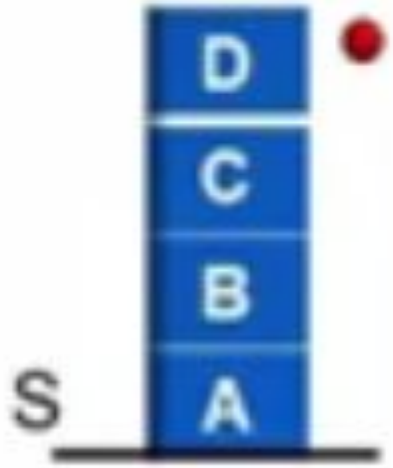
ie (actual cost + potential change)

where

- ❑ c_i' is Amortized cost of the i th operation .
- ❑ c_i is Actual cost of the i th operation .
- ❑ D_i is Data structure.
- ❑ $\Phi(D_i)$ is called the potential of D_i.

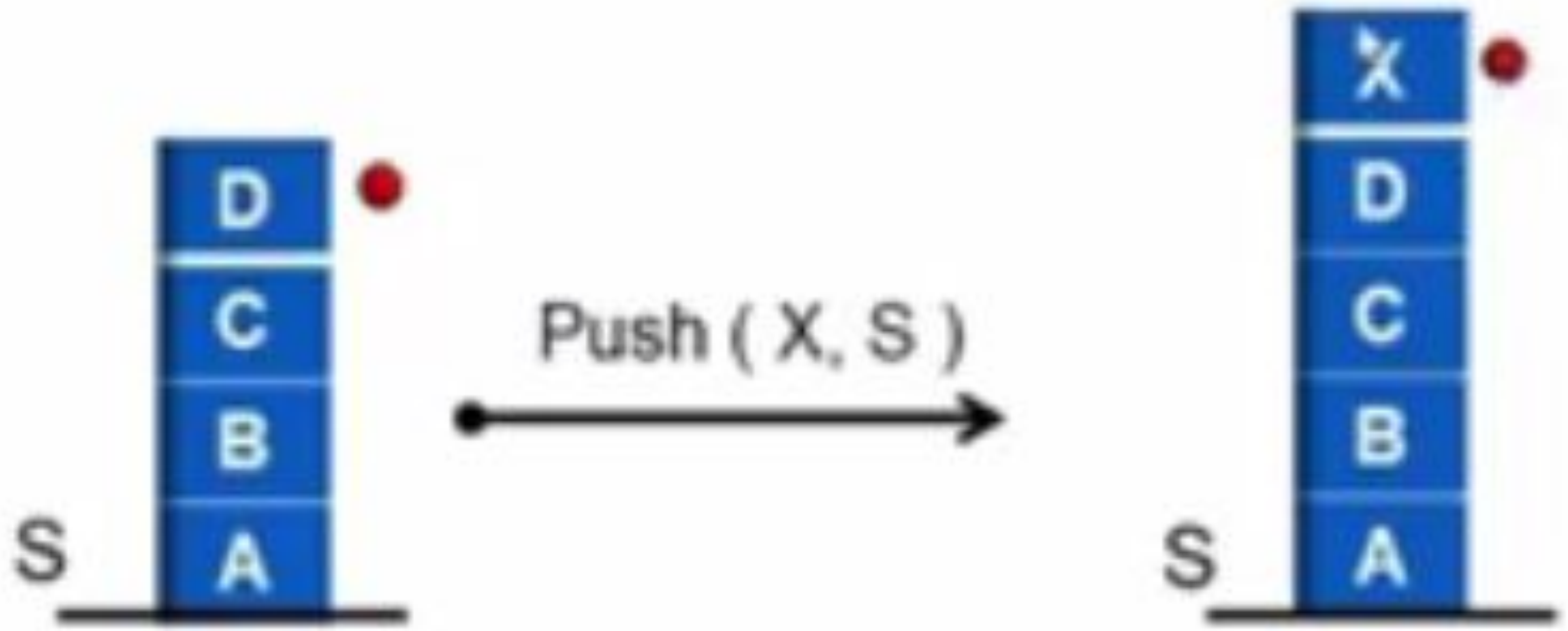
- Goal :
    - Find worst case time, T(n), for n operations

- Push Operation
- Pop Operation
- Multipop Operation



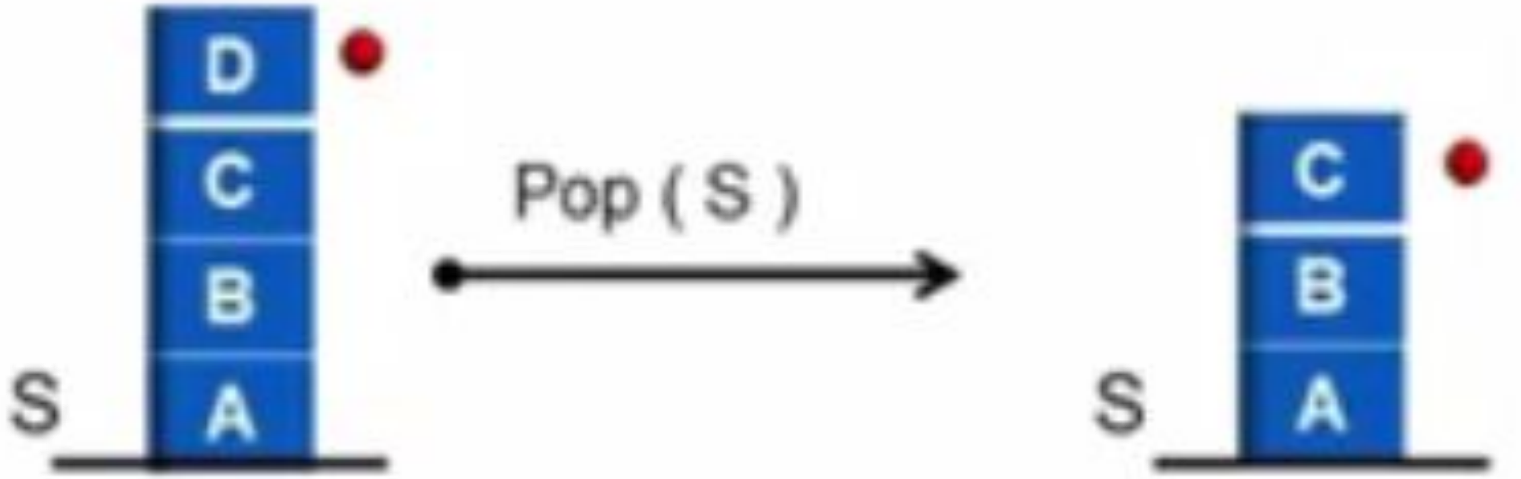
❑ Push Operation :

- Push (X, S) has complexity O(1)



Pop Operation :

- Pop (S) returns popped object, complexity is O(1)

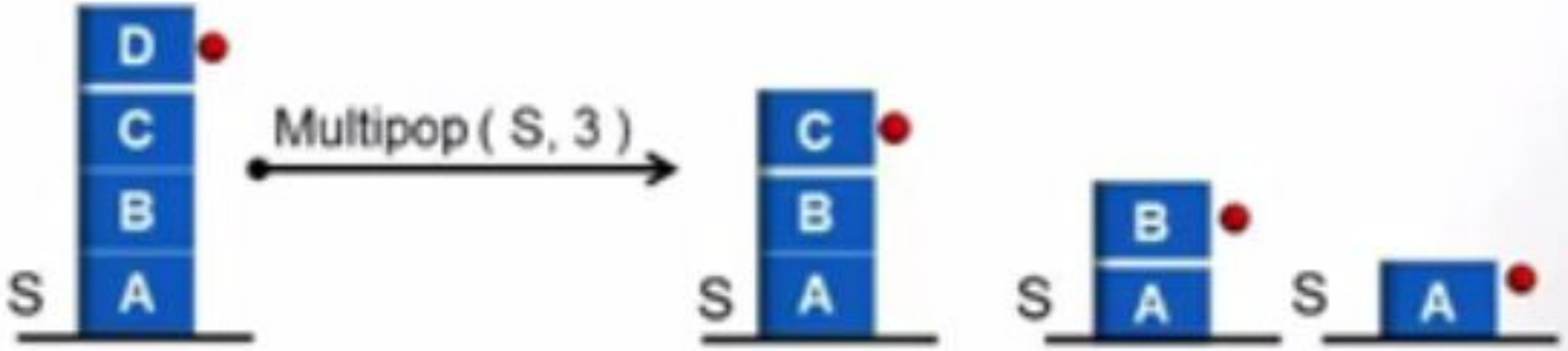


Multipop Operation :

Multipop (S , k)

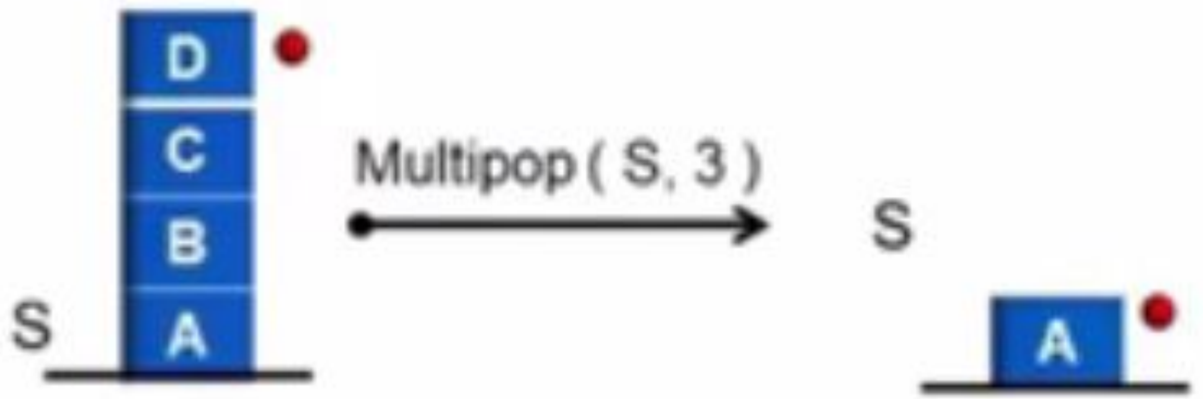
```
1. While not Stack_Empty (S) and k != 0
    2. do Pop ( S )
    3.     k ← k -1
```

- Complexity is min(S , k) where s is the stack size



Multipop Operation :

- Complexity is min(S , k) where s is the stack size.

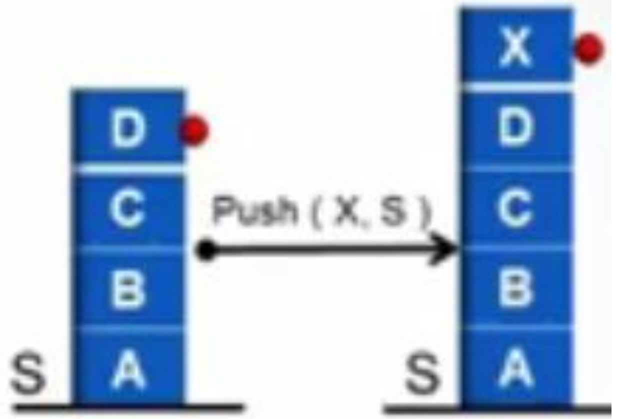


- Complexity is : min(S, k) = min(4, 3) = 3

Amortize Analysis : Stack Operations

- Potential for a stack is the number of objects in the stack.
- So $\Phi(D_0) = 0$, and $\Phi(D_i) \geq 0$

- **Amortized cost of Push:**



  - Potential change:

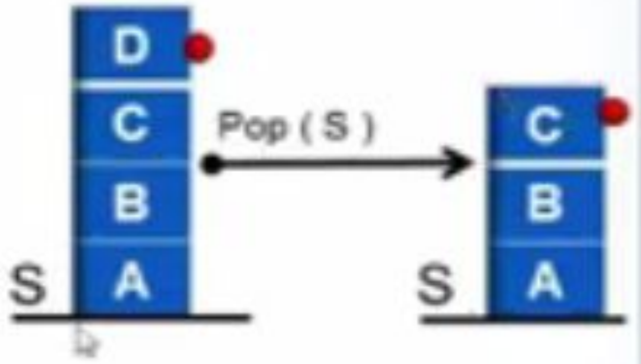
$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1.$$

  - Amortized cost :

$$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + 1 = 2$$
$$c_i' = O(1)$$

Amortize Analysis : Stack Operations

- Potential for a stack is the number of objects in the stack.
- So $\Phi(D_0) = 0$, and $\Phi(D_i) \geq 0$
- **Amortized cost of Pop:**



  - Potential change:

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

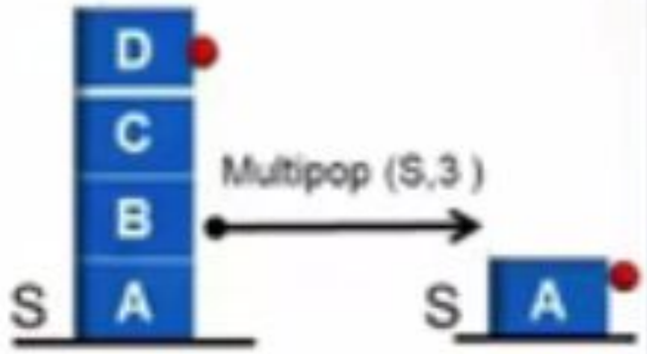
  - Amortized cost :

$$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= 1 + (-1) = 0$$
$$c_i' = O(1)$$

Amortize Analysis : Stack Operations

- Potential for a stack is the number of objects in the stack.
- So $\Phi(D_0) = 0$, and $\Phi(D_i) \geq 0$

- **Amortized cost of Multipop (S,k):**



  - Potential change:

$$\Phi(D_i) - \Phi(D_{i-1}) = (s\text{-k}) - s = \text{-k}.$$

  - Amortized cost :

$$c_i' = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
$$= k + (-k) = 0$$
$$\mathbf{c_i' = O(1)}$$

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Ø (no of ones)	Ci (no of bits flipped)	Amortized cost Ci+Pi-Pi-1
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1	1+1-0 =2
2	0	0	0	0	0	0	1	0	1	2	2+1-1=2
3	0	0	0	0	0	0	1	1	2	1	2
4	0	0	0	0	0	1	0	0	1	3	2
5	0	0	0	0	0	1	0	1	2	1	2
6	0	0	0	0	0	1	1	0	2	2	2
7	0	0	0	0	0	1	1	1	3	1	2
8	0	0	0	0	1	0	0	0	1	4	2
9	0	0	0	0	1	0	0	1	2	1	2
10	0	0	0	0	1	0	1	0	2	2	2
11	0	0	0	0	1	0	1	1	3	1	2
12	0	0	0	0	1	1	0	0	2	3	2
13	0	0	0	0	1	1	0	1	3	1	2
14	0	0	0	0	1	1	1	0	3	2	2
15	0	0	0	0	1	1	1	1	4	1	2
16	0	0	0	1	0	0	0	0	1	5	2

The Total Amortized cost of 16 operations = 32

The Total Amortized cost of 16 operations = 2 x 16

The Total Amortized cost of n operations = 2 x n

Therefore, asymptotically, the total Amortized cost of n operations = O(n)