Amortized Analysis

Heejin Park

Division of Computer Science and Engineering

Hanyang University

Contents

- **Introduction**
- **Aggregate analysis**
- **Accounting method**
- **Potential method**
- **Dynamic Table**

Introduction

- Amortized Analysis
  - We average the time required to perform a sequence of data-structure operations over all the operations performed.
  - Show that the average cost of an operation is small, even though a single operation within the sequence might be expensive.
    - average <u>over a sequence of operations</u>
  - probability is not involved (contrary to average case analysis)
  - Guarantees ***the average performance of each operation*** in the worst case

Introduction

- Three techniques for amortized analysis
  1. aggregate analysis
  2. accounting method
  3. potential method

- Examine these three methods using two examples
  - Stack with MULTIPOP operation
  - Incrementing binary counter

Contents

Aggregate analysis

- Aggregate analysis
  - A sequence of n operations takes *worst-case* time $T(n)$ in total.
  - Average cost per operation is $T(n)/n$ in the worst case
    = amortized cost
  - Amortized cost is the same to each operation
    - Even when there are several types of operations in the sequence

Aggregate analysis

- Example of stack operation
  - Stack operations
    - PUSH(S, x)
    - POP(S)
    - MULTIPOP(S, k)
  - PUSH and POP run in $O(1)$ time.
    - Thus the cost of each is 1.
  - Actual running time for n operations is $\Theta(n)$.
    - Total cost of a sequence of these n operations is n.

Aggregate analysis

- Example of stack operation
  - MULTIPOP(S, k)
    - Actual running time is linear in the number of POP operations actually executed.

MULTIPOP(S, k)

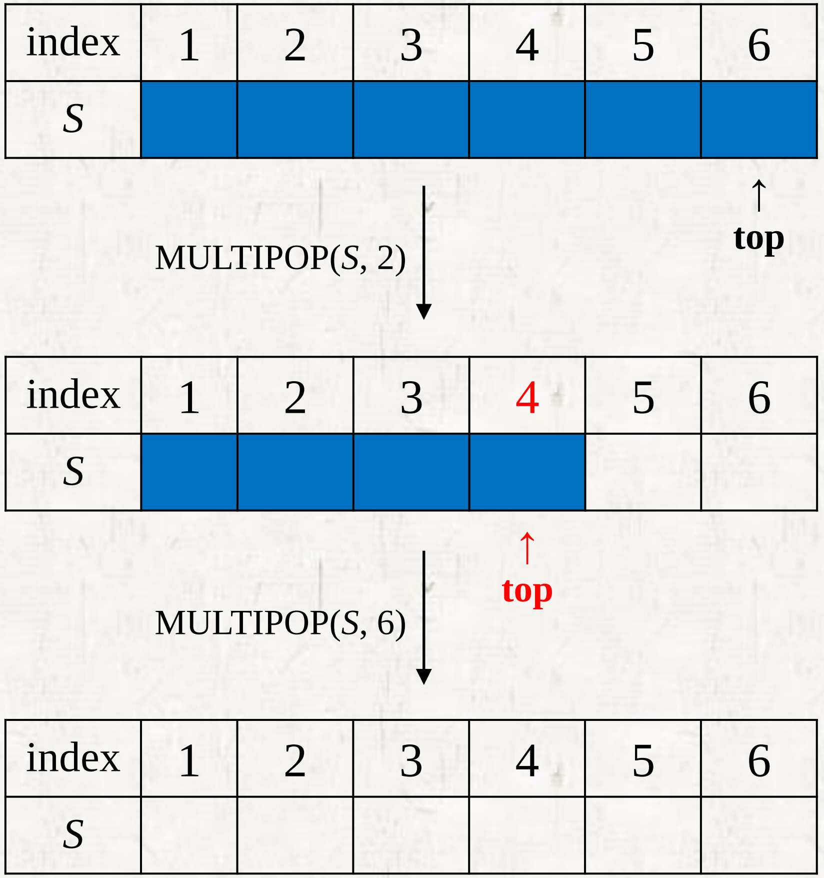
```
while not STACK-EMPTY(S) and k > 0
        POP(S)
        k = k - 1
```

$T(1) * k$

- So, cost of MULTIPOP(S, k) is $O(k)$.

Aggregate analysis

- Example of stack operation
  - MULTIPOP(S, k)
    - Remove the k top objects of stack
    - If objects of stack are less than k, it removes the objects in the stack.
      - So, stack is empty



Aggregate analysis

- Example of stack operation
  - Analysis of a sequence of n PUSH, POP and MULTIPOP operations
    - on an initially empty stack
  - Intuitive analysis of time complexity (wrong way)
    - The worst-case cost of one MULTIPOP: $O(n)$
    - Stack size: at most n

➔ Total cost : $O(n^2)$

- This cost isn't tight

Aggregate analysis

- Example of stack operation
  - Using Aggregate analysis
    - Can obtain a better upper bound from the entire sequence of *n* operations
  - Any sequence of *n* PUSH, POP and MULTIPOP operations
    - on an initially empty stack
    - [Push, push, pop, push, push, push, multipop(2), …]

= [Push, push, pop, push, push, push, {pop, pop}, …]

$$n \geq \#(\text{push}) \geq \#(\text{pop})$$

$$2n \geq \#(\text{push}) + \#(\text{pop})$$

➔ Total cost : O(*n*)

- Amortized cost is $O(n) / n = O(1)$

Aggregate analysis

- Example of incrementing a binary counter
  - Consider the problem of implementing a k-bit binary counter that counts upward from 0
    - Use an array $A[0..k\text{-}1]$ of bits

***A*[k-1]**	**…**	***A*[2]**	***A*[1]**	***A*[0]**

Aggregate analysis

- Example of incrementing binary counter
  - Cost of INCREMENT operation is proportional to the number of bits flip

```
INCREMENT (A)
i = 0
while i < A.length and A[i] == 1
    A[i] = 0            ← Bit flip, T(1)
    i = i + 1
if i < A.length
    A[i] = 1            ← Bit flip, T(1)
```

Aggregate analysis

- Example of incrementing binary counter
  - Cost of INCREMENT operation is proportional to the number of bits flipped.

Ex. INCREMENT(A)

Counter value	***A*[4]**	***A*[3]**	***A*[2]**	***A*[1]**	***A*[0]**	**cost**	**Total cost**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**1**	**1**	**1**

Aggregate analysis

- Example of incrementing binary counter
  - Cost of INCREMENT operation is proportional to the number of bits flipped.

Ex. INCREMENT(*A*)

Counter value	***A*[4]**	***A*[3]**	***A*[2]**	***A*[1]**	***A*[0]**	**cost**	**Total cost**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**1**	**1**	**1**
2	**0**	**0**	**0**	**1**	**0**	**2**	**3**

Aggregate analysis

- Example of incrementing binary counter
  - Cost of INCREMENT operation is proportional to the number of bits flipped.

Ex. INCREMENT(A)

Counter value	***A*[4]**	***A*[3]**	***A*[2]**	***A*[1]**	***A*[0]**	**cost**	**Total cost**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**1**	**1**	**1**
2	**0**	**0**	**0**	**1**	**0**	**2**	**3**
3	**0**	**0**	**0**	**1**	**1**	**1**	**4**

Aggregate analysis

- Example of incrementing binary counter
  - Cost of INCREMENT operation is proportional to the number of bits flipped.

Ex. INCREMENT(A)

Counter value	A[4]	A[3]	A[2]	A[1]	A[0]	cost	Total cost
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**1**	**1**	**1**
2	**0**	**0**	**0**	**1**	**0**	**2**	**3**
3	**0**	**0**	**0**	**1**	**1**	**1**	**4**
4	**0**	**0**	**1**	**0**	**0**	**3**	**7**

Aggregate analysis

- Example of incrementing binary counter
  - A single execution of INCREMENT takes time $\Theta(k)$ in the worst case
    - In which array A contains all 1s.

$A[k-1]$	…	$A[2]$	$A[1]$	$A[0]$	cost
1	…	1	1	1	-
0	…	0	0	0	k

  - Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

Aggregate analysis

- Example of incrementing binary counter
  - Aggregate Analysis
    - can tighten our analysis to yield a worst-case cost of $O(n)$ for a sequence of n INCREMENT operations
      - by observing that not all bits flip each time INCREMENT is called

Aggregate analysis

- Example of incrementing binary counter
  - Compute bit flip of Array A
    - flip of $A[0]$: n
    - flip of $A[1]$: $\left\lfloor \frac{n}{2} \right\rfloor$
    - flip of $A[2]$: $\left\lfloor \frac{n}{4} \right\rfloor$
  - The total number of flip in the sequence
    - $\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} n/2^i = 2n$

    ➔ Total cost $O(n)$

  - Amortized cost = $O(n)/n = O(1)$

	$A[3]$	$A[2]$	$A[1]$	$A[0]$
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
	↑	↑	↑	↑
	$n/8$	$n/4$	$n/2$	n

Aggregate analysis

- Running time
  - $O(n)$ time in total
- Amortized cost
  - $O(n) / n = O(1)$

Contents

- *Introduction*
- *Aggregate analysis*
- **Accounting method**
- **Potential method**
- **Dynamic Table**

Accounting method

- Accounting method
  - Amortized cost of an operation: assign differing charges to different operations, with some operations charged more or less than they actually cost
  - Credit
    - the difference between amortized cost and actual cost
    - can help pay for later operations whose amortized cost is less than their actual cost
  - Different operations may have different amortized costs, unlike aggregate analysis.

Accounting method

- Accounting method
  - choose amortized cost of operations, so that the total amortized cost provides an upper bound on the total actual cost.
    - c_i : actual cost of the ith operation
    - $\hat{c}_i$: amortized cost of the ith operation
    - $\sum_{i=0}^{n} \hat{c}_i \geq \sum_{i=0}^{n} c_i$, for all sequences of n operations
  - The total credit stored in the data structure is $\sum_{i=0}^{n} \hat{c}_i - \sum_{i=0}^{n} c_i$, and must be non-negative at all times.

Accounting method

- Example of stack operation
  - The actual costs of the operations
    - PUSH 1
    - POP 1
    - MULTIPOP $\min(k,s)$
  - The amortized costs of the operations
    - PUSH 2
    - POP 0
    - MULTIPOP 0

Accounting method

- Example of stack operation

index	1	2	3	4
S				

S	**1**	**2**	**3**	**4**
credit				

Accounting method

- Example of stack operation
  - PUSH

index	1	2	3	4
S				

↑
top

S	1	2	3	4
credit	**1**			

  - PUSH : actual cost 1 + prepaid credit 1
  - Amortized cost : actual cost + credit = 2

Accounting method

- Example of stack operation
  - PUSH

index	1	2	3	4
S				

↑
top

S	1	2	3	4
credit	1	1		

  - PUSH : actual cost 1 + prepaid credit 1
  - Amortized cost : actual cost + credit = 2

Accounting method

- Example of stack operation
  - PUSH

index	1	2	3	4
S				

↑
top

S	**1**	**2**	**3**	**4**
credit	**1**	**1**	**1**	

  - PUSH : actual cost 1 + prepaid credit 1
  - Amortized cost : actual cost + credit = 2

Accounting method

- Example of stack operation
  - POP

index	1	2	3	4
S				

↑
top

S	**1**	**2**	**3**	**4**
credit	**1**	**1**	**0**	

  - POP and MULTIPOP : pay credit 1
  - Amortized cost : actual cost - credit = 0

Accounting method

- Example of stack operation
  - PUSH

index	1	2	3	4
S				

↑
top

S	**1**	**2**	**3**	**4**
credit	**1**	**1**	**1**	

- PUSH : actual cost 1 + prepaid of credit 1
- Amortized cost : actual cost + credit = 2

Accounting method

- Example of stack operation
  - POP and MULTIPOP must execute after PUSH operation
    - Charging the PUSH operation a little bit more (= credit)

      So, credit pay actual cost of POP and MULTIPOP operation

  - The amount of credit is always nonnegative
    - Because the stack always has nonnegative objects.
    - Thus, the total amortized cost is an upper bound on the total actual cost

  - Total amortized cost : $O(n)$

  - Total actual cost : $O(n)$

Accounting method

- Example of incrementing a binary counter
  - The actual costs
    - Bit set ($0 \rightarrow 1$)   :   1
    - Bit reset ($1 \rightarrow 0$) :   1
  - The amortized costs
    - Bit set   :   2
    - Bit reset   :   0

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	**0**	**0**	**1**

A	***A*[0]**	***A*[1]**	***A*[2]**	***A*[3]**	***A*[4]**
credit	**1**				

Accounting method

- Example of incrementing binary counter

***A*[3]**	***A*[2]**	***A*[1]**	***A*[0]**
0	**0**	**0**	**1**
			0

A	***A*[0]**	***A*[1]**	***A*[2]**	***A*[3]**	***A*[4]**
credit	**0**				

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	**0**	**0**	**1**
0	**0**	**1**	**0**

A	***A*[0]**	***A*[1]**	***A*[2]**	***A*[3]**	***A*[4]**
credit	**0**	**1**			

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	**0**	**0**	**1**
0	**0**	**1**	**0**
0	**0**	**1**	**1**

A	***A*[0]**	***A*[1]**	***A*[2]**	***A*[3]**	***A*[4]**
credit	**1**	**1**			

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	**0**	**0**	**1**
0	**0**	**1**	**0**
0	**0**	**1**	**1**
			0

A	*A*[0]	*A*[1]	*A*[2]	*A*[3]	*A*[4]
credit	**0**	**1**			

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	**0**	**0**	**1**
0	**0**	**1**	**0**
0	**0**	**1**	**1**
		0	**0**

A	*A*[0]	*A*[1]	*A*[2]	*A*[3]	*A*[4]
credit	**0**	**0**			

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	0	0	1
0	0	1	0
0	0	1	1
	1	0	0

A	*A*[0]	*A*[1]	*A*[2]	*A*[3]	*A*[4]
credit	0	0	1		

Accounting method

- Example of incrementing binary counter

A[3]	*A*[2]	*A*[1]	*A*[0]
0	**0**	**0**	**1**
0	**0**	**1**	**0**
0	**0**	**1**	**1**
0	**1**	**0**	**0**
0	**1**	**0**	**1**

A	*A*[0]	*A*[1]	*A*[2]	*A*[3]	*A*[4]
credit	**1**	**0**	**1**		

Accounting method

- Example of incrementing binary counter
  - Bit reset must execute after bit set
    - Charging the bit set in credit

      So, credit pay for actual cost of reset operation

  - The amount of credit is always nonnegative

    Because the number of 1s in the counter never becomes negative

    Thus, the total amortized cost is an upper bound on the total actual cost

  - the total amortized cost : $O(n)$

  - the total actual cost : $O(n)$

Accounting method

- Amortized cost
  - $O(n)$ time in total
- Running time
  - $O(n)$ time in total

Contents

- *Introduction*
- *Aggregate analysis*
- *Accounting method*
- **~~Potential method~~**
- **Dynamic Table** (57n)

Potential method

- Potential method
  - Similar to an accounting method

    Credit → "potential energy" or just "potential"

  - The potential with the data structure as a whole rather than with specific objects within the data structure.

Potential method

- Potential method
  - With n operations,

    D_0 : an initial data structure

    D_i : the data structure that results after applying the ith operation to data structure D_{i-1}

    $\Phi(D_i)$: the potential associated with data structure D_i

  - Potential difference ($\Phi(D_i) - \Phi(D_{i-1})$)
    - positive

      The potential of the data structure increases
    - negative

      The decrease in the potential pays for the actual cost of the operation

Potential method

- Potential method
  - Amortized cost
    - $\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - The total amortized cost of the *n* operations
    - $\sum_{i=0}^{n} \hat{c}_i = \sum_{i=0}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1}))$
      $= \sum_{i=0}^{n} c_i + \Phi(D_n) - \Phi(D_0)$
  - We require $\Phi(D_i) \geq \Phi(D_0)$ for all *i*
    - so that $\sum_{i=0}^{k} \hat{c}_i \geq \sum_{i=0}^{k} c_i$ for all $1 \leq k \leq n$

Potential method

- Example of stack operation
  - Potential function Φ
    - the number of objects in the stack
    - $\Phi(D_0) = 0$
  - The stack D_i that results after the ith operation has nonnegative potential
    - $\Phi(D_i) \geq 0 = \Phi(D_0)$

Potential method

- Example of stack operation
  - Amortized cost analysis of each operation
    - PUSH operation
      - If the ith operation on a stack containing s objects is a PUSH operation,
        - $\Phi(D_i) - \Phi(D_{i-1}) = (s + 1) - s = 1$
        - So, the amortized cost is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
          $= 1 + (s + 1) - s$
          $= 2$
    - POP operation
      - If the ith operation on a stack containing s objects is a POP operation,
        - $\Phi(D_i) - \Phi(D_{i-1}) = (s - 1) - s = -1$
        - So, the amortized cost is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$
          $= 1 + (s - 1) - s$
          $= 0$

Potential method

- Example of stack operation
  - Amortized cost analysis of each operation
    - MULTIPOP(S,k) operation
      - If the ith operation on a stack containing s objects is a MULTIPOPoperation,
      - k' = min(k, s) : The number of objects to be popped off the stack
        - $\Phi(D_i) - \Phi(D_{i-1})$ = –min(k, s) = – k'
          The amortized cost is $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
          $= k' - k'$
          $= 0$

Potential method

- Example of stack operation
  - Amortized cost : $O(1)$
  - Total amortized cost : $O(n)$
  - Total actual cost : $O(n)$

3. Potential method

- Example of incrementing binary counter
  - Potential function Φ
    - The number of 1's in the array
    - b_i : The number of 1's in the counter after the *i*th INCREMENT operation
    - t_i : The number of bits reset in the *i*th INCREMENT operation
  - Actual cost of the operation
    - c_i is at most t_i +1

```
INCREMENT (A)
    i = 0
    while i < A.length and A[i] == 1
        A[i] = 0
        i = i + 1
    if i < A.length
        A[i] = 1
```

3. Potential method

- Example of incrementing a binary counter
  - Case of $b_i = 0$
    - the ith operation resets all k bits
    - $b_{i\text{-}1} = t_i = k$
  - Case of $b_i > 0$
    - $b_i = b_{i\text{-}1} - t_i + 1$
  - In either case
    - $b_i \leq b_{i\text{-}1} - t_i + 1$

Ex) 1111 -> 0000

Counter value	$A[k]$	…	$A[2]$	$A[1]$	$A[0]$	b_i
i-1	1	…	1	1	1	k
i	0	…	0	0	0	0

Ex) 0111 -> 1000

Counter value	$A[k]$	…	$A[2]$	$A[1]$	$A[0]$	b_i
i-1	0	…	1	1	1	k-1
i	1	…	0	0	0	1

3. Potential method

- Example of incrementing a binary counter
  - Potential difference
    - $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i\text{-}1}$
      $\leq (b_{i\text{-}1} - t_i + 1) - b_{i\text{-}1}$
      $= 1 - t_i$
  - Amortized cost
    - $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$
      $\leq (t_i + 1) + (1 - t_i) = 2$
      ➔ O(1)

3. Potential method

- Example of incrementing a binary counter
  - If the counter starts at zero, $\Phi(D_0) = 0$ and since $\Phi(D_i) \geq 0$ for all i
    - The total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost
    - The worst-case cost of n INCREMENT operations is $O(n)$

3. Potential method

- Example of incrementing a binary counter
  - If it does not start at zero
    - $b_0 \geq 0$, $b_n \leq k$ (k : the number of bits in the counter)
    - $\sum_{i=0}^{n} \hat{c}_i = \sum_{i=0}^{n} c_i + \Phi(D_n) - \Phi(D_0)$
      - $\sum_{i=0}^{n} c_i = \sum_{i=0}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0)$ ($\hat{c}_i \leq 2$ for all $1 \leq i \leq n$)

        $\leq \sum_{i=0}^{n} 2 - b_n + b_0$ ($\Phi(D_n) = b_n$, $\Phi(D_0) = b_0$)

        $= 2n - b_n + b_0$
    - The total actual cost is $O(n)$ ($b_0 \leq k$, $k = O(n)$)

Contents

785

Dynamic tables

- We do not always know in advance how many objects some applications will store in a table
  - insertion
    - So allocate space for a table and reallocate the table when new item is added.
  - deletion
    - Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size
  - Using amortized analysis, we shall show that the amortized cost of insertion and deletion is only $O(1)$

Aggregate analysis

- INSERT
  - When inserting an item into a full table, we can expand the table by allocating a new table with more slots than the old table had.
  - A common heuristic allocates a new table with **twice** as many slots as the old one.

Aggregate analysis

- INSERT
  - *T.table* : a pointer to the block of storage representing the table.
  - *T.num* : the number of items in the table
  - *T.size* : the total number of slots in the table.

Aggregate analysis

```
TABLE-INSERT(T, x)
    if T.size == 0
        allocate T.table with 1 slot
        T.size = 1
    if T.num == T.size
        allocate new-table with 2 * T.size slots
        insert all items in T.table into new-table
        free T.table
        T.table = new-table
        T.size = 2 * T.size
    insert x into T.table
    T.num = T.num +1
```

elementary insertion

expansion

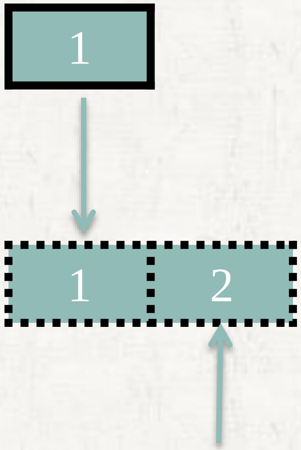
Aggregate analysis

- Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table.
  - If the current table has room for the new item, then cost $c_i = 1$.
  - If the current table is full, an expansion occurs, then $c_i = i$.
    - 1 for insert new item, i-1 for move for extend

1

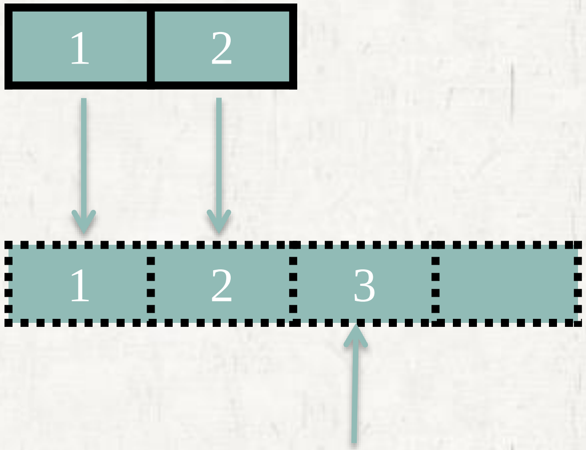
Aggregate analysis

- Let us analyze a sequence of *n* TABLE-INSERT operations on an initially empty table.
  - If the current table has room for the new item, then cost $c_i = 1$.
  - If the current table is full, an expansion occurs, then $c_i = i$.
    - 1 for insert new item, i-1 for move for extend



Aggregate analysis

- Let us analyze a sequence of *n* TABLE-INSERT operations on an initially empty table.
  - If the current table has room for the new item, then cost $c_i = 1$.
  - If the current table is full, an expansion occurs, then $c_i = i$.
    - 1 for insert new item, i-1 for move for extend



Aggregate analysis

- Let us analyze a sequence of *n* TABLE-INSERT operations on an initially empty table.
  - If the current table has room for the new item, then cost $c_i = 1$.

  - If the current table is full, an expansion occurs, then $c_i = i$.
    - 1 for insert new item, i-1 for move for extend

Aggregate analysis

- Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table.

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of } 2 \\ 1 & \text{otherwise} \end{cases}$$

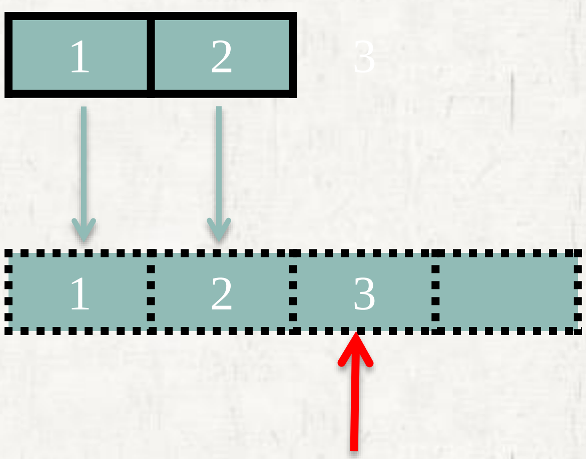
Aggregate analysis

- The total cost of n TABLE-INSERT operations Is therefore

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

Aggregate analysis

- Let us analyze a sequence of *n* TABLE-INSERT operations on an initially empty table.
  - For 1 to *n*, when item inserted in table, it's cost is 1.
    - It requires 1 * *n* = *n* cost.
    - It is expressed in under arrow.



$$\boxed{n} + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$$

Aggregate analysis

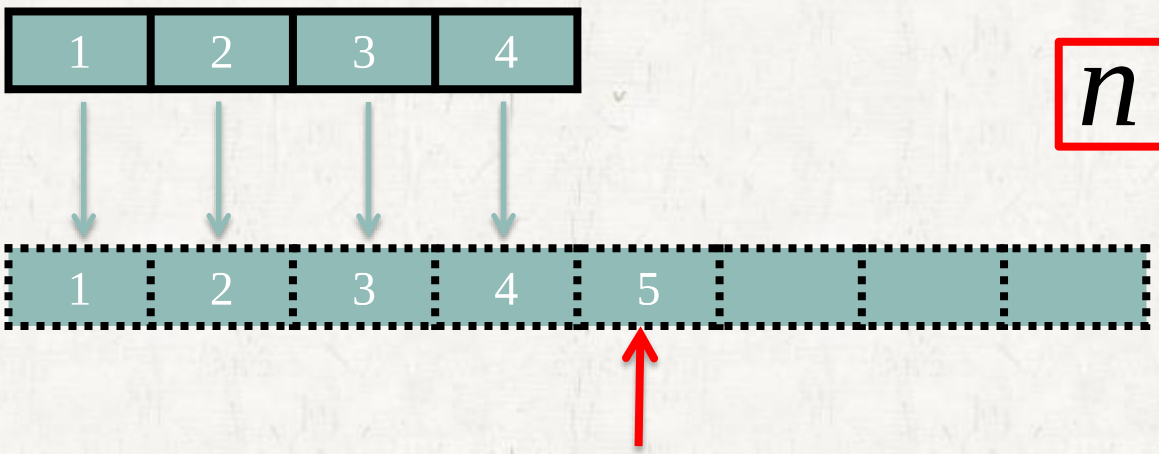
- Let us analyze a sequence of *n* TABLE-INSERT operations on an initially empty table.
  - For 1 to *n*, when item inserted in table, it's cost is 1.
    - It requires 1 * $n = n$ cost.
    - It is expressed in under arrow.

$$\boxed{n} + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$$

1	2	3	4

Aggregate analysis

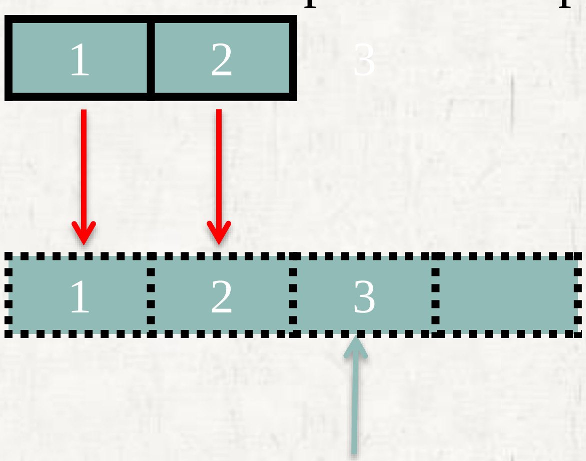
- Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table.
  - For 1 to n, when item inserted in table, it's cost is 1.
    - It requires $1 * n = n$ cost.
    - It is expressed in under arrow.



$$\boxed{n} + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$$

Aggregate analysis

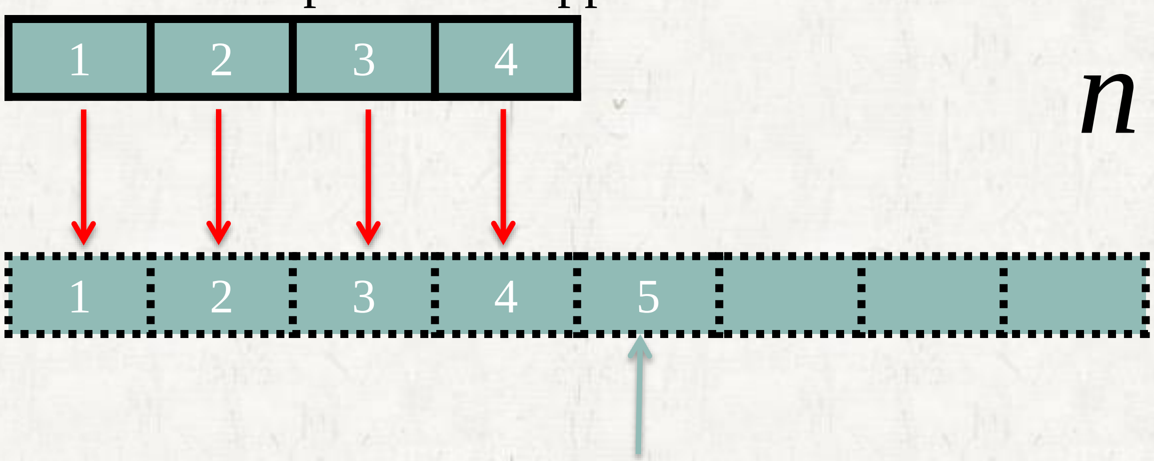
- Let us analyze a sequence of *n* TABLE-INSERT operations on an initially empty table.
  - When table size is exact power of 2, table expansion occur
    - 2^j insert is occurred.
    - And it occurred $\lfloor \lg n \rfloor$ times.
    - It is expressed in upper arrow



$$n + \boxed{\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j}$$

Aggregate analysis

- Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table.
  - When table size is exact power of 2, table expansion occur
    - 2^j insert is occurred.
    - And it occurred $\lfloor \lg n \rfloor$ times.
    - It is expressed in upper arrow



$$n + \boxed{\sum_{j=0}^{\lfloor \lg n \rfloor} 2^j}$$

Aggregate analysis

- The total cost of n TABLE-INSERT operations Is therefore

$$\begin{aligned}\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n\end{aligned}$$

- Since the total cost of n TABLE-INSERT operations is bounded by 3n, the amortized cost of single operation is at most 3 (3n / n)

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.

Accounting method

```
TABLE-INSERT(T, x)
    if T.size == 0
        allocate T.table with 1 slot
        T.size = 1
    if T.num == T.size
        allocate new-table with 2 * T.size slots
        insert all items in T.table into new-table
        free T.table
        T.table = new-table
        T.size = 2 * T.size
    insert x into T.table
    T.num = T.num +1
```

elementary insertion

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - There are two types of elementary insertion:
    - 6 insert all items in *T.table* into new-table
    - 10 insert *x* into *T.table*

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - 1 **cost** for line 10,
    - 2 **credit** for line 6.
    - Credit is used to move items when expansion is occured

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2		
0	0		

Credit for move

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2	3		
0	0	0		Credit for move

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2	3		
0	0	1		Credit for move

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2	3		
1	0	1		Credit for move

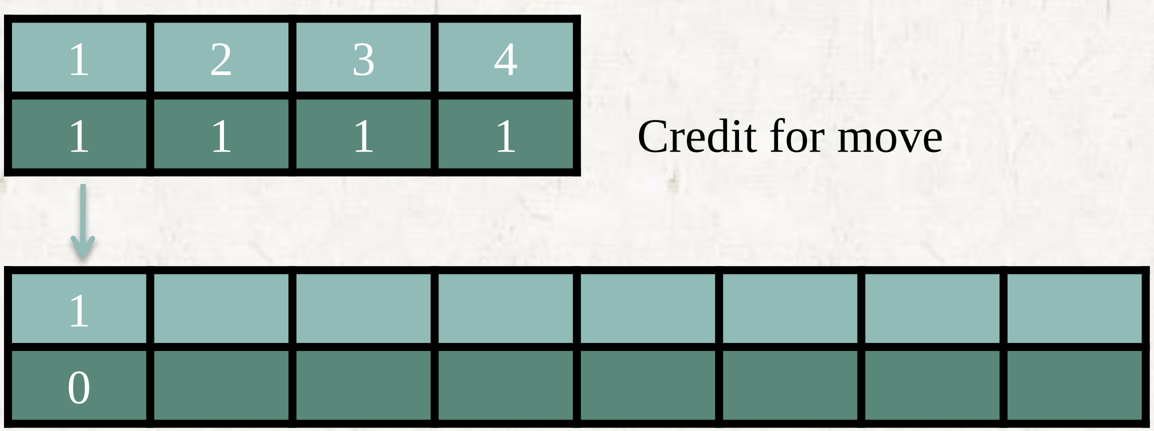
Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2	3	4	
1	1	1	1	Credit for move

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands



Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2	3	4
1	1	1	1

Credit for move

1	2						
0	0						

Accounting method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TABLE-INSERT operation should be 3.
  - each item pays for 3 elementary insertions:
    - inserting itself into the current table
    - moving itself when the table expands
    - moving another item that has already been moved once when the table expands

1	2	3	4
1	1	1	1

Credit for move

1	2	3	4				
0	0	0	0				

Potential method

- We can use the potential method to analyze a sequence of n TABLE-INSERT operations.
  - and we shall use it in Section 17.4.2 to design a TABLE-DELETE operation that has an O(1) amortized cost as well

Potential method

- We can use the potential method to analyze a sequence of n TABLE-INSERT operations.
  - and we shall use it in Section 17.4.2 to design a TABLE-DELETE operation that has an O(1) amortized cost as well

  - We start by defining a potential function Φ
    - 0 immediately after an expansion
    - table size by the time the table is full

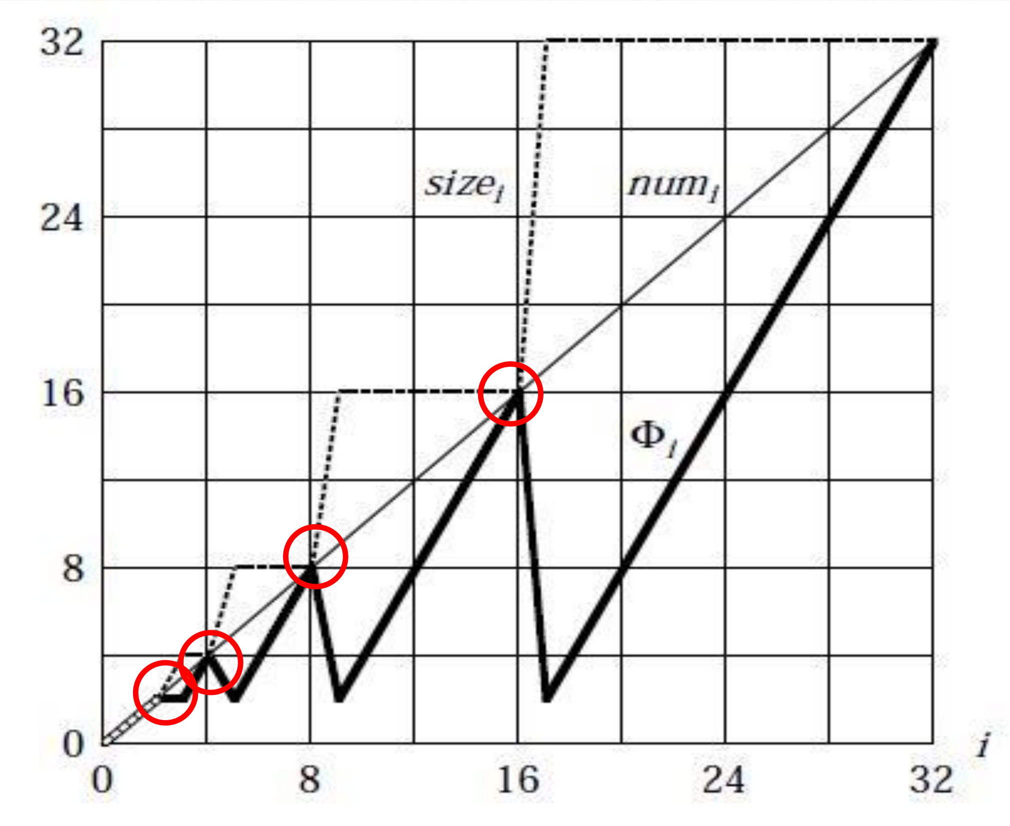
Potential method

- $\Phi(T) = 2*T.num - T.size$ (17.5)

- Immediately before an expansion, we have $T.num = T.size$ and thus $\Phi(T) = T.num$

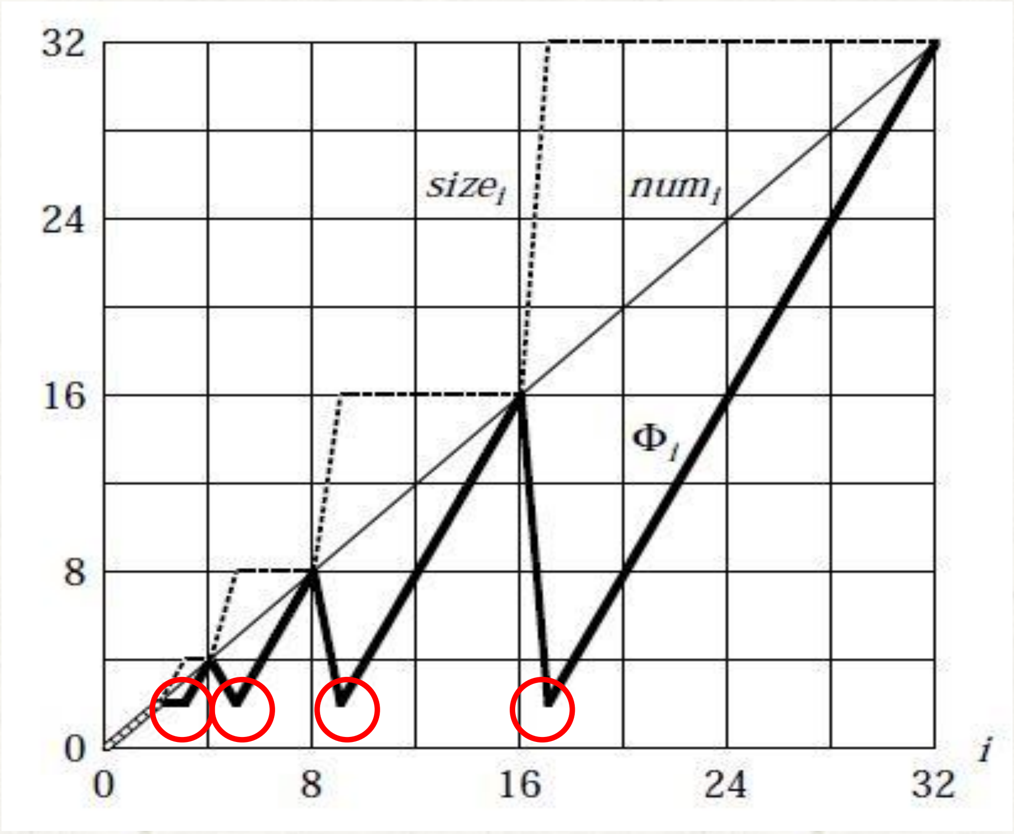
- $\Phi(T)$ is always nonnegative
  - The initial value of the potential is 0
  - and since the table is always at least half full, $T.num \geq T.size/2$

Potential method



- Before extention, $\Phi_i = num_i$

Potential method



- After extention, $\Phi_i = 0$ but immediately increased by 2

Potential method

- The amortized cost of the *i*th TABLE-INSERT operation
  - num_i : the number of items stored in the table after the *i*th operation
  - $size_i$: the total size of the table after the *i*th operation
  - Φ_i : the potential after the *i*th operation
  - $\hat{c}_i$: its amortized cost with respect to Φ

  - Initially, we have $num_0 = 0$, $size_0 = 0$, and $\Phi_0 = 0$.

Potential method

- The amortized cost of the *i*th TABLE-INSERT operation
  - If the *i*th TABLE-INSERT operation does not trigger an expansion, then we have $size_i = size_{i-1}$ and the amortized cost of the operation is
    - $\Phi(T) = 2*T.num - T.size$

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2*num_i - size_i) - (2*num_{i-1} - size_{i-1}) \\
&= 1 + (2*num_i - size_i) - (2*(num_i - 1) - size_i) \\
&= 3
\end{aligned}
$$

Potential method

- The amortized cost of the *i*th TABLE-INSERT operation
  - If the *i*th operation does trigger an expansion, then we have
    $size_i = 2 * size_{i-1}$
    $size_{i-1} = num_{i-1} = num_i - 1$
    $size_i = 2 * (num_i - 1)$.
    Thus, the amortized cost of the operation is

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2*num_i - size_i) - (2*num_{i-1} - size_{i-1}) \\
&= num_i + (2*num_i - 2*(num_i - 1)) - (2*(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3
\end{aligned}
$$

Table expansion and contraction

- TABLE-DELETE operation.
  - Table contraction is analogous to table expansion:
    - when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one

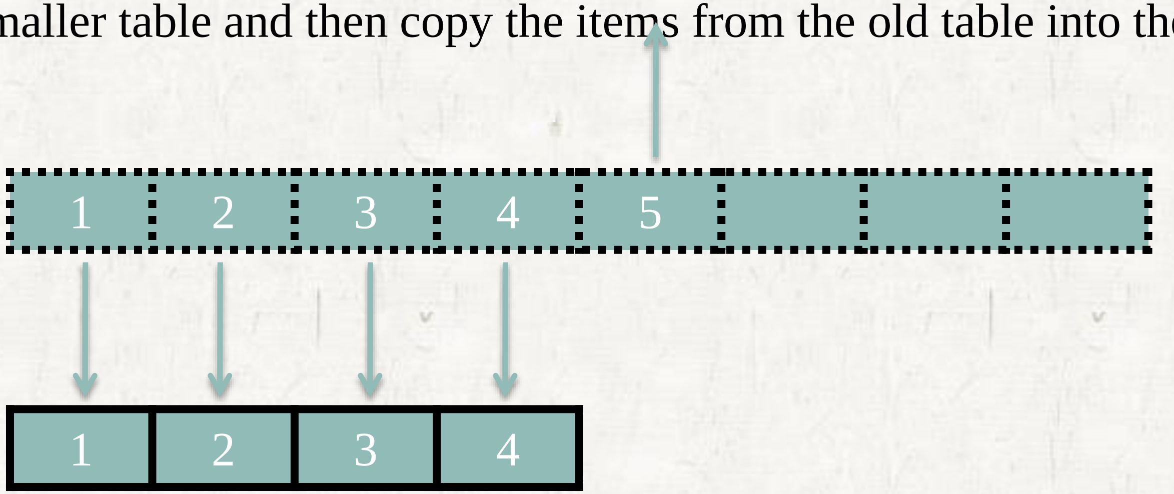


Table expansion and contraction

- TABLE-DELETE operation.
  - load factor : $\alpha(T) = T.num\ /\ T.size$

Table expansion and contraction

- TABLE-DELETE operation.
  - load factor : $\alpha(T) = T.num \,/\, T.size$
  - we would like to preserve two properties:
    - the load factor of the dynamic table is bounded below by a positive constant
    - the amortized cost of a table operation is bounded above by a constant.

Table expansion and contraction

```
TABLE-DELETE(T, x)
    if T.size == 0
        break
    if T.num == T.size / 2
        allocate new-table with T.size / 2 slots
        insert all items in T.table into new-table
        free T.table
        T.table = new-table
        T.size = T.size / 2
    delete x into T.table
    T.num = T.num – 1
```

Table expansion and contraction

- Table expansion and contraction
  - double the table size upon inserting an item into a full table
  - halve the size when a deleting an item would cause the table to become less than half full
  - This strategy would guarantee that the load factor of the table never drops below 1/2, but have **problem**

Table expansion and contraction

- Table expansion and contraction
  - We perform n operations on a table T , where n is an exact power of 2.
  - The first $n/2$ operations are insertions,
    - cost a total of $\Theta(n)$.
  - At the end of this sequence of insertions, $T.num = T.size = n/2$.
  - For the second $n/2$ operations, we perform the following sequence:
    - insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

1	….	n/2

Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

  - And insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

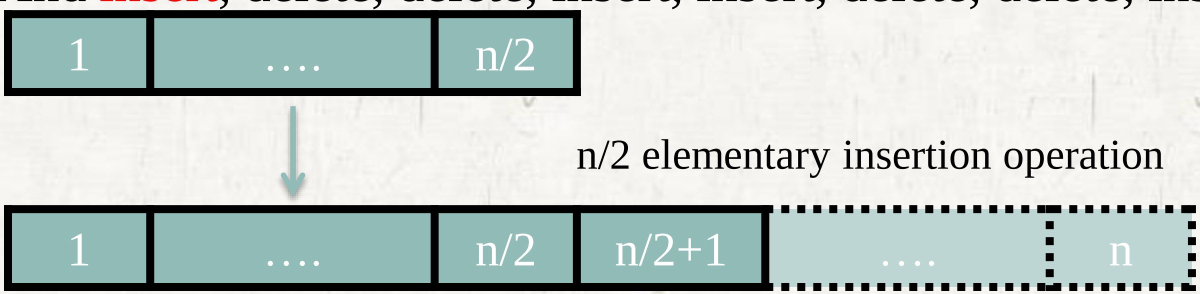


Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

  - And insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

  - And insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

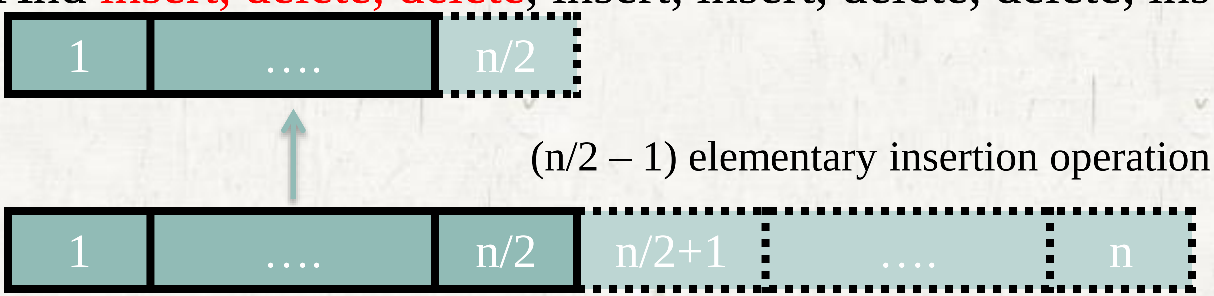


Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

1	….	n/2

  - And insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

1	….	n/2

Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

  - And insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .

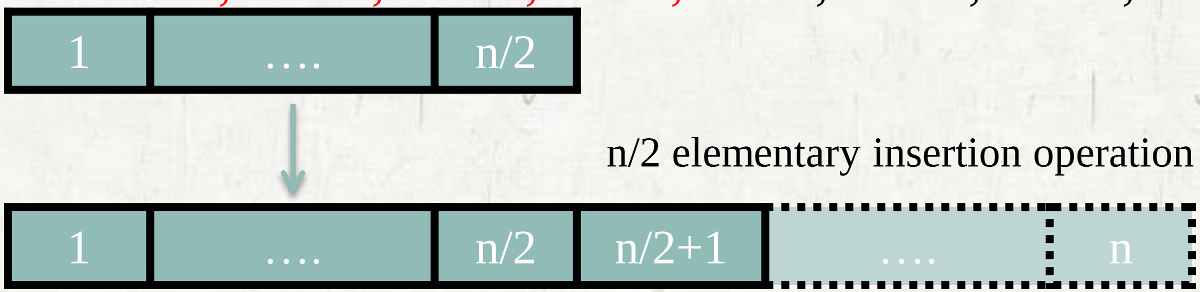


Table expansion and contraction

- Table expansion and contraction
  - First n/2 insertion.

1	….	n/2

  - And insert, delete, delete, insert, insert, delete, delete, insert, insert, . . .
    - n / 2 number of insertion occur in 2 operation!

Table expansion and contraction

- Table expansion and contraction
  - The cost of each expansion and contraction is $\Theta(n)$, and there are Θ (n) of them.
    - After n /2 th operation, cost of each 2 operation is n / 2
  - Thus, the total cost of the n operations is Θ (n^2), making the amortized cost of an operation $\Theta(n)$.

Table expansion and contraction

- Improve upon this strategy
  - Specifically, we continue to double the table size upon inserting an item into a full table,
  - but we halve the table size when deleting an item causes the table to become less than 1/4 full, rather than 1/2 full as before.
  - The load factor of the table is therefore bounded below by the constant 1/4.

Table expansion and contraction

- potential method to analyze the cost of a sequence of *n* TABLE-INSERT and TABLE-DELETE operations
  - Let us denote the load factor of a nonempty table T by $\alpha(T) = T.num / T.size$
  - Since for an empty table, $T.num = T.size = 0$ *and* $\alpha(T) = 1$
  - We shall use as our potential function

$$\Phi_i = \begin{cases} 2 * num_i - size_i & \text{if } \alpha(T) \geq 1/2 \\ size_i / 2 - num_i & \text{if } \alpha(T) < 1/2 \end{cases}$$

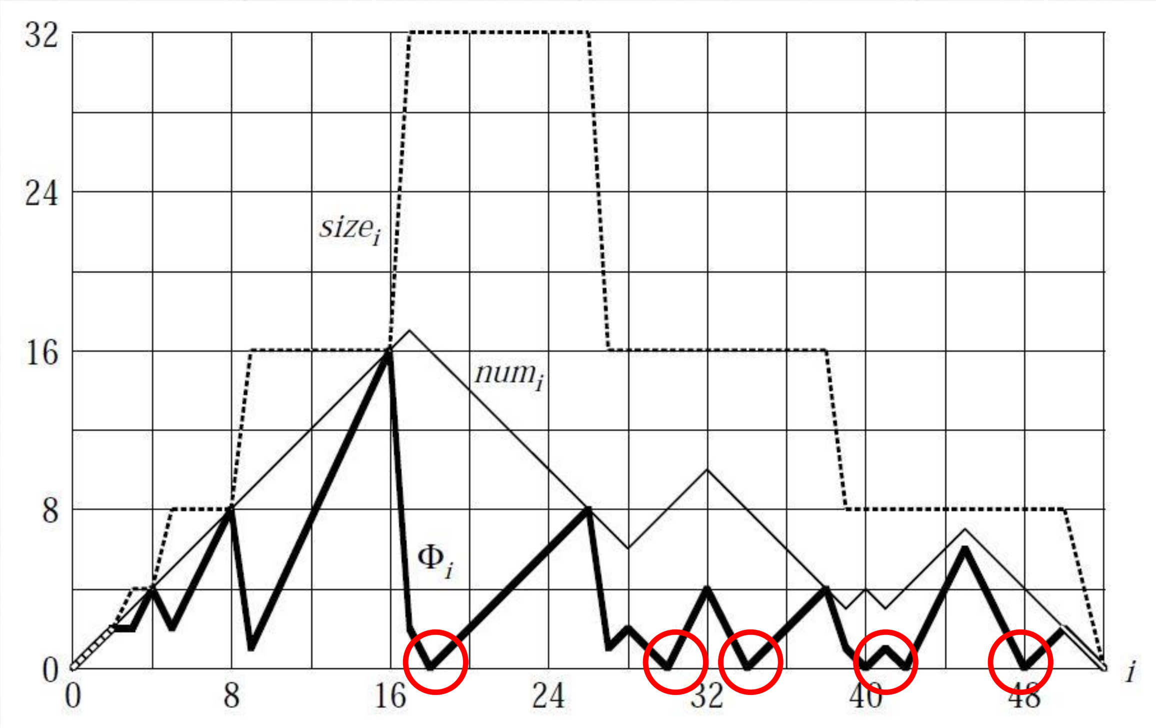
Table expansion and contraction

- potential method to analyze the cost of a sequence of *n* TABLE-INSERT and TABLE-DELETE operations

$$\Phi_i = \begin{cases} 2 * num_i - size_i & \text{if } \alpha(T) \geq 1/2 \\ size_i / 2 - num_i & \text{if } \alpha(T) < 1/2 \end{cases}$$

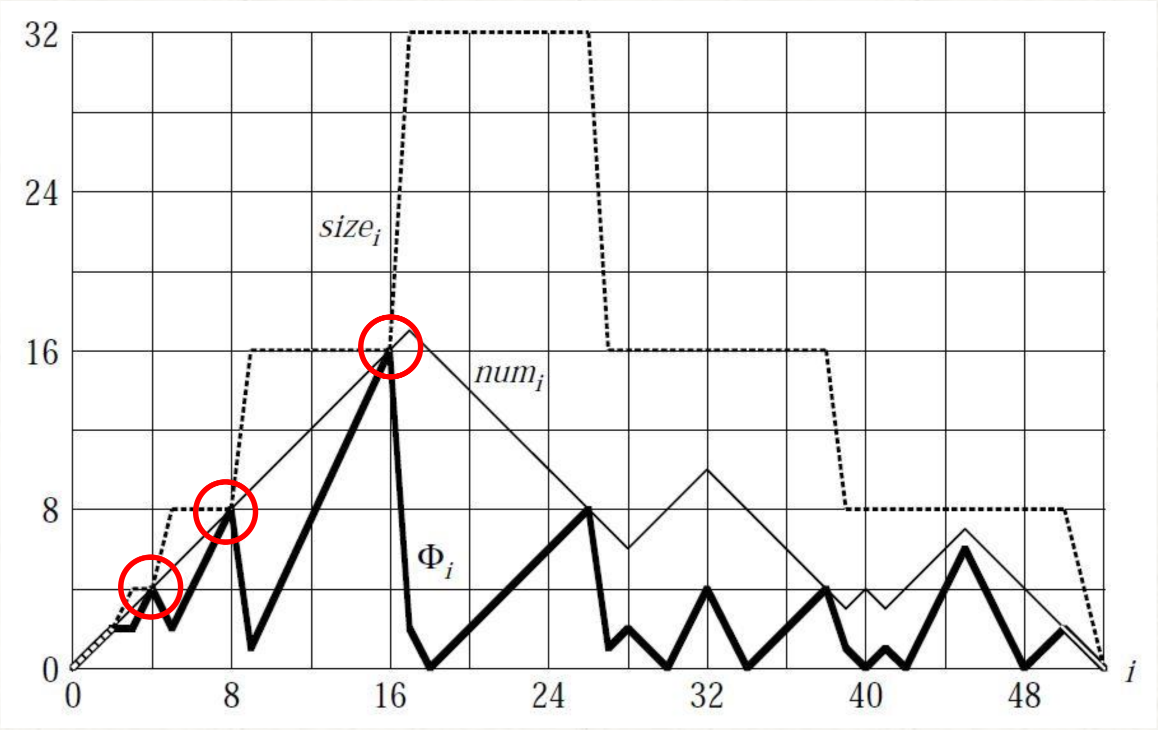
  - When $\alpha(T) = 1/4$, and if *i*th operation is deletion
    - Contraction is occur.
    - It need num_i potential

Table expansion and contraction



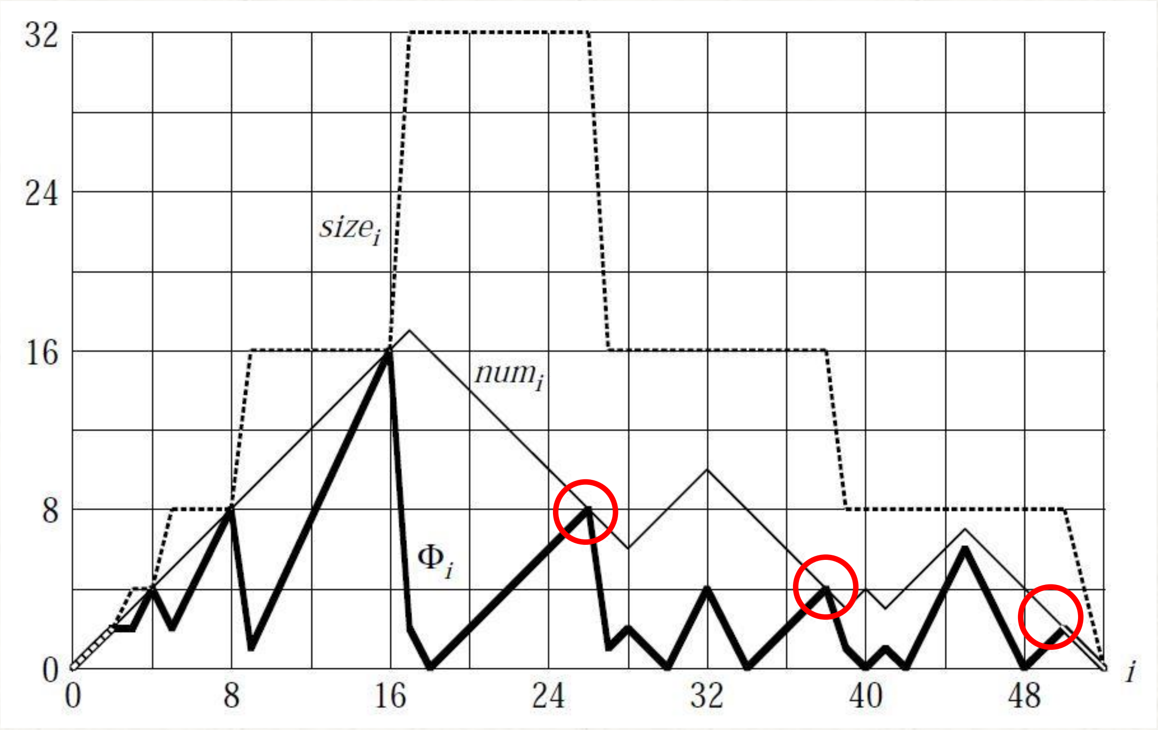
- When load factor is 1/2, the potential is 0.

Table expansion and contraction



- When the load factor is 1
  - we have T.*size* = *T.num*
    which implies Φ(*T*) = T.*num*
  - Thus the potential can pay for an expansion if an item is inserted.

Table expansion and contraction



- When the load factor is 1/4, we have *T.size = 4*T.num*, which implies Φ(*T*) = T.*num*
  - Thus the potential can pay for a contraction if an item is deleted.

Table expansion and contraction

- TABLE-INSERT and TABLE-DELETE
  - c_i : the actual cost of the *i*th operation
  - $\hat{c}_i$: its amortized cost with respect to Φ
  - num_i : the number of items stored in the table after the *i*th operation
  - $size_i$: the total size of the table after the *i*th operation
  - α_i : the load factor of the table after the *i*th operation
  - Φ_i : the potential after the *i*th operation
  - Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, *and* $\Phi_0 = 0$

Table expansion and contraction

- TABLE-INSERT
  - The analysis is identical to that for table expansion in Section 17.4.1 if $\alpha_{i-1} \geq 1/2$.
    - Whether the table expands or not, the amortized cost of the operation is at most 3

Table expansion and contraction

- TABLE-INSERT
  - If $\alpha_{i\ -1}$ <1/2, table cannot expand.
    - Then $size_i = size_i$-1
    - $num_{i-1} = num_i - 1$

      Then the amortized cost of the *i*th operation is

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$
$$= 1 + (size_i/2 - num_i) - (size_{i-1} / 2 - num_{i-1})$$
$$= 1 + (size_i/2 - num_i) - (size_i / 2 - (num_i - 1))$$
$$= 0$$

Table expansion and contraction

- TABLE-INSERT
  - If $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then
    - $\Phi_{i-1} = T.size / 2 - T.num$
    - $\Phi_i = 2 * T.num - T.size$
    - $num_i = num_{i-1} + 1 = \alpha_i * size_i$

Table expansion and contraction

- TABLE-INSERT
  - If $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then

    $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$

    $= 1 + (2*num_i - size_i) - (size_{i-1} / 2 - num_{i-1})$

    $= 1 + (2*(num_{i-1} + 1) - size_{i-1}) - (size_{i-1} / 2 - num_{i-1})$

    $= 3 * num_{i-1} - 3size_{i-1}/2 + 3$

    $= 3\alpha_{i-1}size_{i-1} - 3size_{i-1}/2 + 3$

    $< 3size_{i-1}/2 - 3size_{i-1}/2 + 3$

    $= 3$

  - Thus, the amortized cost of a TABLE-INSERT operation is at most 3.

Table expansion and contraction

- TABLE-DELETE
  - $num_i = num_{i-1} - 1$

  - If $\alpha_{i-1} < 1/2$
    - 1. operation causes the table to contract
    - 2. operation does trigger a contraction

Table expansion and contraction

- TABLE-DELETE
  - If $\alpha_{i-1} < 1/2$
    - 1. operation causes the table to contract
      - Then $size_i = size_{i-1}$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1} / 2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i / 2 - (num_i+1)) \\
&= 2
\end{aligned}$$

Table expansion and contraction

- TABLE-DELETE
  - If $\alpha_{i-1} < 1/2$
    - 2. operation does trigger a contraction
      - actual cost of the operation is $c_i = num_i + 1$
      - $size_i / 2 = size_{i-1}/4 = num_{i-1} = num_i + 1$
      - the amortized cost of the operation is

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1} / 2 - num_{i-1}) \\
&= (num_i + 1) + ((num_i + 1) - num_i) - ((2*num_i + 2) - (num_i + 1)) \\
&= 1
\end{aligned}
$$

Table expansion and contraction

- TABLE-DELETE
  - If $\alpha_{i\ -1} \geq 1/2$
    - It's amortize cost is constant.
    - If $\alpha_i < 1/2$, operation don't trigger contraction.

Table expansion and contraction

- In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is O(n).