Introduction to Algorithms

CS 430

Lecture 15-16

ILLINOIS INSTITUTE OF TECHNOLOGY

By Lan Yao

Outlines

- Amortized Analysis
  - Aggregate Analysis
  - Accounting Analysis
  - Potential Analysis
  - More Examples

What is Amortized Analysis?

- Not an algorithm;
- Do NOT need to or should NOT appear in the code;
- It is the average cost over a sequence of operations on a data structure;
- It could be small although a single operation may be expensive;
- It is not the cost for average case and does not involve probability analysis;

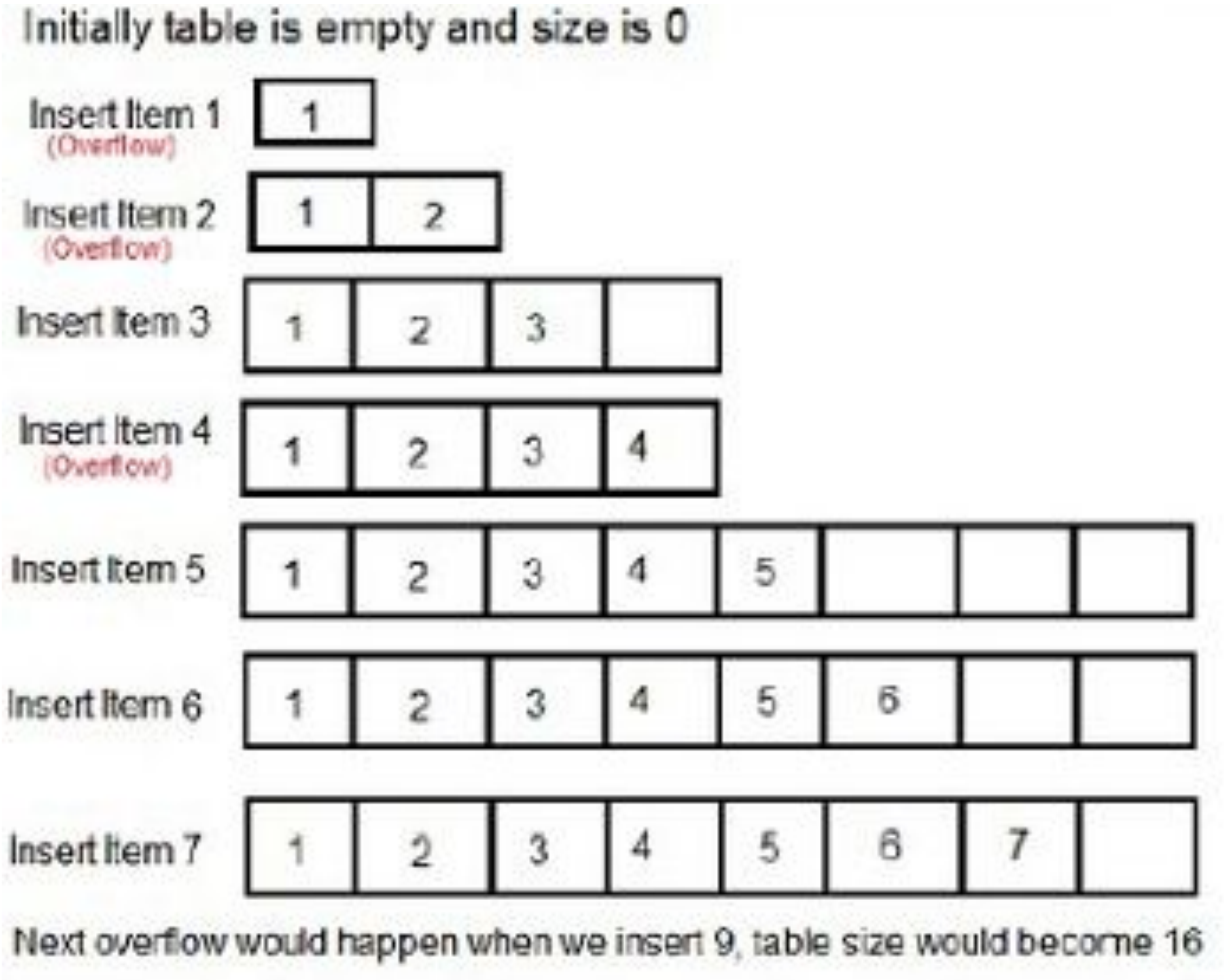
What is Amortized Analysis?

– If the upper bound on cost of a sequence of operations is T(n)--worst case, then T(n)/n is the amortized cost of each operation;
– In amortized analysis, all operations have the same amortized cost as T(n)/n, although their accurate cost may differ.
– why? It helps to improve your algorithm design.

What is Amortized Analysis?

- ex. Insertion to a dynamic array.
  - items can be inserted at a given index--cost is O(1);
  - if that index is not present in the array, it has to double the size of the array then inserts the element if the index is present--the cost is not a constant;

- ex. Insertion to a dynamic array.



- ex. Insertion to a dynamic array.
  - If ci is the cost to insert ith element,

So $ci = 1 + \begin{cases} i-1, \text{ if } i-1 \text{ is power of } 2 \\ 0, \qquad\qquad \text{Otherwise} \end{cases}$

$$\frac{\sum_{i=1}^{n} c_i}{n} \le \frac{n + \sum_{j=1}^{\lfloor log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

$$2^0 + 2^1 + 2^2 + \ldots + 2^a = 2^{a+1} - 1$$

$$\Rightarrow \quad 2^1 + 2^2 + \ldots + 2^{\lfloor \lg(n-1) \rfloor} = 2^{\lfloor \lg(n-1) \rfloor + 1} - 1 - 1 = 2^{\lfloor \lg(n-1) \rfloor + 1} - 2 = 2(n-1) - 2 = O(n)$$

Aggregate Analysis

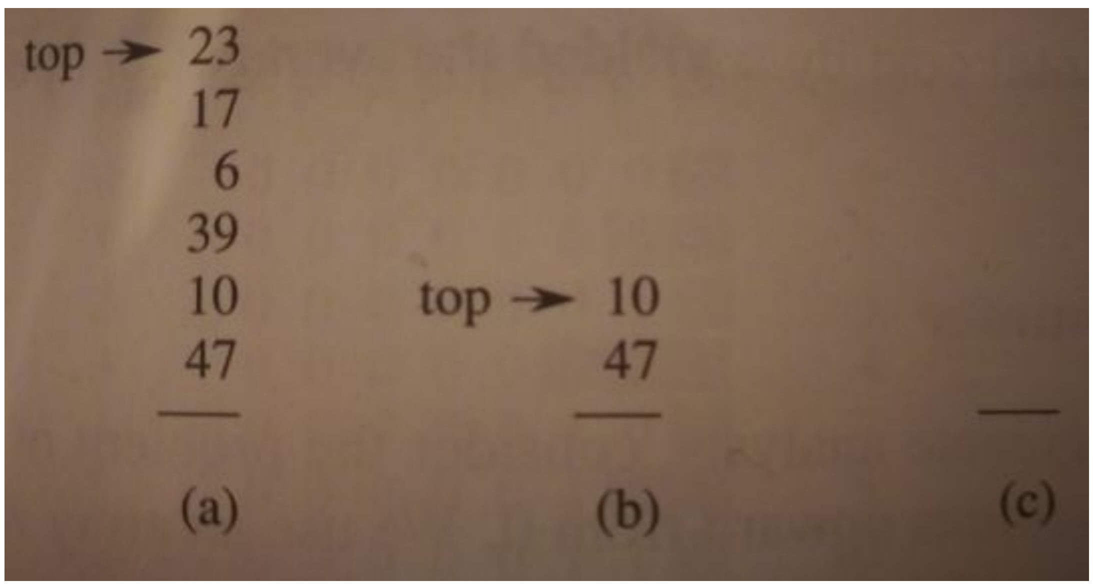
- the amortized cost =T(n)/n and it applies to any operation in n-operations sequence.
  - operations may be in different types.
- T(n) is the worst case cost.
- what are different types of operations?
  - insertion, deletion, selecting a single element, selecting a subarray of elements......

Aggregate Analysis

- Ex#1. Stack operations
  - pop, push, what else?
  - PUSH (S,x)--pushes x onto stack S;--O(1)
  - POP (S)-- pops and returns the top of S;--O(1)
  - the cost of a sequence of n PUSH or POP operations is O(n).
  - add a new operation: Multipop(S,k)--pops k top elements from S if the size of S>=k; otherwise pops all elements.

Aggregate Analysis

- Ex#1. Stack operations
  - Multipop(S,k)--pops k top elements from S if the size of S>=k; otherwise pops all elements.
  - Multipop(S,4), Multipop(S,3)



Aggregate Analysis

- algorithm and running time

```
MULTIPOP(S,k)
while not STACK-EMPTY(S) and k≠0
   do POP(S)
      k←k-1
```

Accurate running is:

Running time for a POP operation is c;

running time for Multipop (S,k)=c*min{s,k}, where s is the size of S.

- what is the cost of a sequence of n POP, PUSH and Mulitpop?

  – the size of stack is n;
  – worst case:
    - the cost of a Multipop is at most n= O(n);
    - n multipop operations in the sequence;
    - entire cost is O(n*n)=O(n2)

  Is $O(n^2)$ correct?

  Correct, but NOT TIGHT.

- interaction between operations:
  - push once, pop at most once
  - the number of times that POP, including Multipops is executed is at most = # PUSH
  - # PUSH is at most n;
  - For any n, the cost of a sequence of n POP, PUSH or Multipop is O(n).
- Amortized cost=O(n)/n=O(1)

Aggregate Analysis

- Ex#2. Binary Counter
  - A[0..k-1] is an array denoting a k-bit binary counter that counts up from 0;
  - counts up: add 1 to the value of counter;
  - A[0] stores the lowest value place of the counter and A[k-1] stores the highest value place.
  - x is the counting value.
  - for example, if x=11, then A[]=(11010000000...00)

A[0]   A[k-1]

Aggregate Analysis

- Ex#2. Binary Counter
  - count up by 1
    for 16 times
  - Adding 1 to A[i] is to flip it.
    if A[i] is 1, it yields a carry
    to A[i+1]. Then it goes
    iteratively.

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

- Algorithm of Binary Counter--count up by 1

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

- Amortized analysis on binary counter
  - worst case: each increment costs O(k);
  - a sequence of n Increment operations: O(nk);
  - correct, but not tight!

Consider the worst case.
if an operation causes
k flips, the operation
following it will cause 1 flip.

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

- A[0] flips every time. It flips n times at most.
- A[1] flips every other time. It flips $\left\lfloor \frac{n}{2} \right\rfloor$ times at most.

- A[2] flips $\left\lfloor \frac{n}{4} \right\rfloor$ times at most.
- for i=0,1,...,$\lfloor \lg n \rfloor$, A[i] flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ times.
- for i>$\lfloor \lg n \rfloor$, those A[i] do not need to flip at all.
  – each operation is an increment by 1, so that n operations is making n as the value of A, which requires $\lfloor \lg n \rfloor$ bits as a binary for n.

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

- A[2] flips $\left\lfloor \frac{n}{4} \right\rfloor$ times at most.

for i=0,1,..., $\lfloor \lg n \rfloor$ A[i] flips $\left\lfloor \frac{n}{2^i} \right\rfloor$ times.

- the total number of flips in A is:

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

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

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

Accounting Method

- definitions
  - for different operations, accounting method "charges" differently from their actual costs, less or more. These chargers are **amortized cost**.
  - when amortized cost>actual cost, **credit=amortized cost-actual cost**;
  - **credit is stored for future use when amortized cost<actual cost**.

Accounting Method

- definitions (cont'd)
  - How to assign amortized cost?
    - it must show that in worst case, the average cost is small;
    - the total amortized cost must be an upper bound on actual cost.
    - $\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$, if there are n operations in a sequence.
    -
    - total credit stored in the data structure is:

      (otherwise amortized cost will not be upper on actual cost)

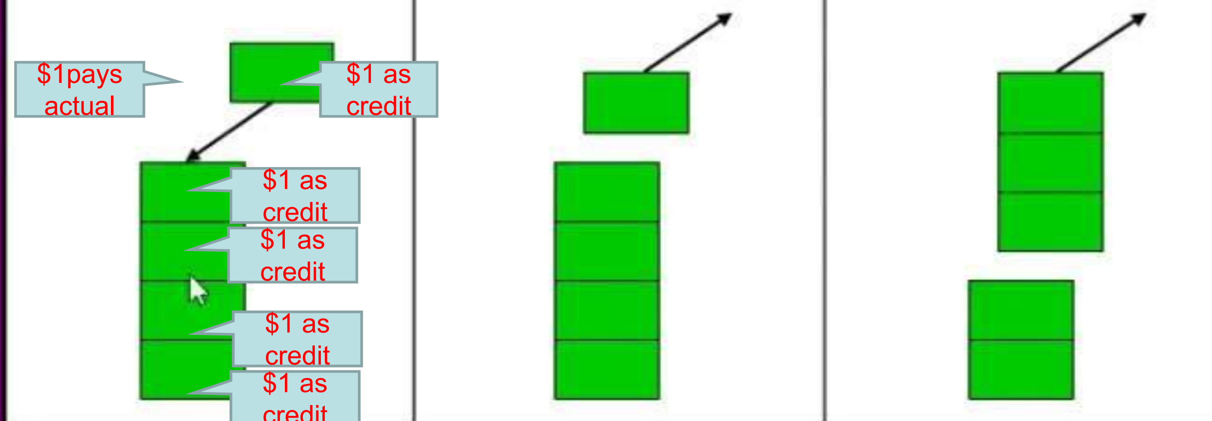
$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0$$

Ex#1: stack operations

actual cost:	PUSH	1
	POP	1
	Multipop	min(k, s)
amortized cost:	PUSH	2
	POP	0
	Multipop	0

Amortized Analysis

….. Accounting Method: Stack Example



3 ops:	Push(S,x)	Pop(S)	Multi-pop(S,k)
•Actual cost:	1	1	$\min(\lvert S\rvert,k)$
•Amortized cost:	2	0	0

Push(S,x) pays for possible later pop of x.

- amortized cost analysis
  - each object in the stack has \$1 of credit on it;
  - the total credit for a stack is nonnegative= the number of objects in the stack after a sequence of n operations=O(n);

- Ex#2. Binary Counter

actual cost: flip 1

amortized cost: set a bit to 1 : 2

reset a bit to 0 : 0

actual cost

amortized cost

Binary Counter	Step Cost	Bank Total
0 0 0 0 0		
0 0 0 0 1	$1	$1
0 0 0 1 0	$2	$1
0 0 0 1 1	$1	$2
0 0 1 0 0	$3	$1
0 0 1 0 1	$1	$2
0 0 1 1 0	$2	$2
0 0 1 1 1	$1	$3
0 1 0 0 0	$4	$1
0 1 0 0 1	$1	$2
... ...		

every 1 has a 1 dollar of credit on it

- amortized analysis

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

do not need extra payment. Each 1 has $1 credit.

Pay $2. But in each Increment, at most 1 bit is set.

The total amortized cost is nonnegative;
In each Increment, the amortized cost is at most $2;
The total amortized cost is at most 2*n=O(n).

Potential Method

- Definitions
  - the overpaid work is stored as “potential”;
  - it is associated with the entire data structure other than a single object;
  - it can be released for future operation;

  (decreases after an operation)

Potential Method

- Definitions (cont'd)
  - c_i: the actual cost of the ith operation; D_i: the data structure after i^{th} operation to D_{i-1}; $\varphi(D_i)$ is the potential associated with D_i;
  - $\hat{c}_i$ is the amortized cost of the i^{th} operation and is defined as: $\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$
  - the total amortized cost is:

$$\sum_{i=0}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \phi(D_i) - \phi(D_{i-1})] = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

Key point: how to define φ?

Potential Method

- Definitions (cont'd)
  - the total amortized cost is:

$$\sum_{i=0}^{n} \hat{c}_i = \sum_{i=1}^{n} [c_i + \phi(D_i) - \phi(D_{i-1})] = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

To define φ:

$\varphi(D_n) >= \varphi(D_0)$, so that total amortized cost is the upper bound to actual cost; (we'd better let all $\varphi(D_i) >= \varphi(D_0)$ and $\varphi(D_0)=0$)

All φs that follow the above codes are applicable for amortized analysis, but may yield different upper bounds.

- Ex#1. Stack problem

Amortized Analysis: Stack Example

Potential method:

Let the potential of a stack be the *number of elements* in the stack.

worst case

Actual cost	
PUSH	1
POP	1
MULTIPOP	min(k, S)

+

Potential diff
+1
-1
-min(k,S)

=

Amortized cost	
PUSH	2
POP	0
MULTIPOP	0

For the worst case, amortized cost of a sequence of n operations=n*2=O(n)

- Ex#2. Binary Counter

- t_i is the number of reset bits in the i^{th} operation;
- $c_i <= t_i + 1$; ;

$$\left.\begin{aligned} &if\ b_i = 0,\ \ then\ b_{i-1} = k = t_i; \\ &if\ b_i > 0,\ \ then\ b_i = b_{i-1} - t_i + 1; \end{aligned}\right\} b_i \le b_{i-1} - t_i + 1$$

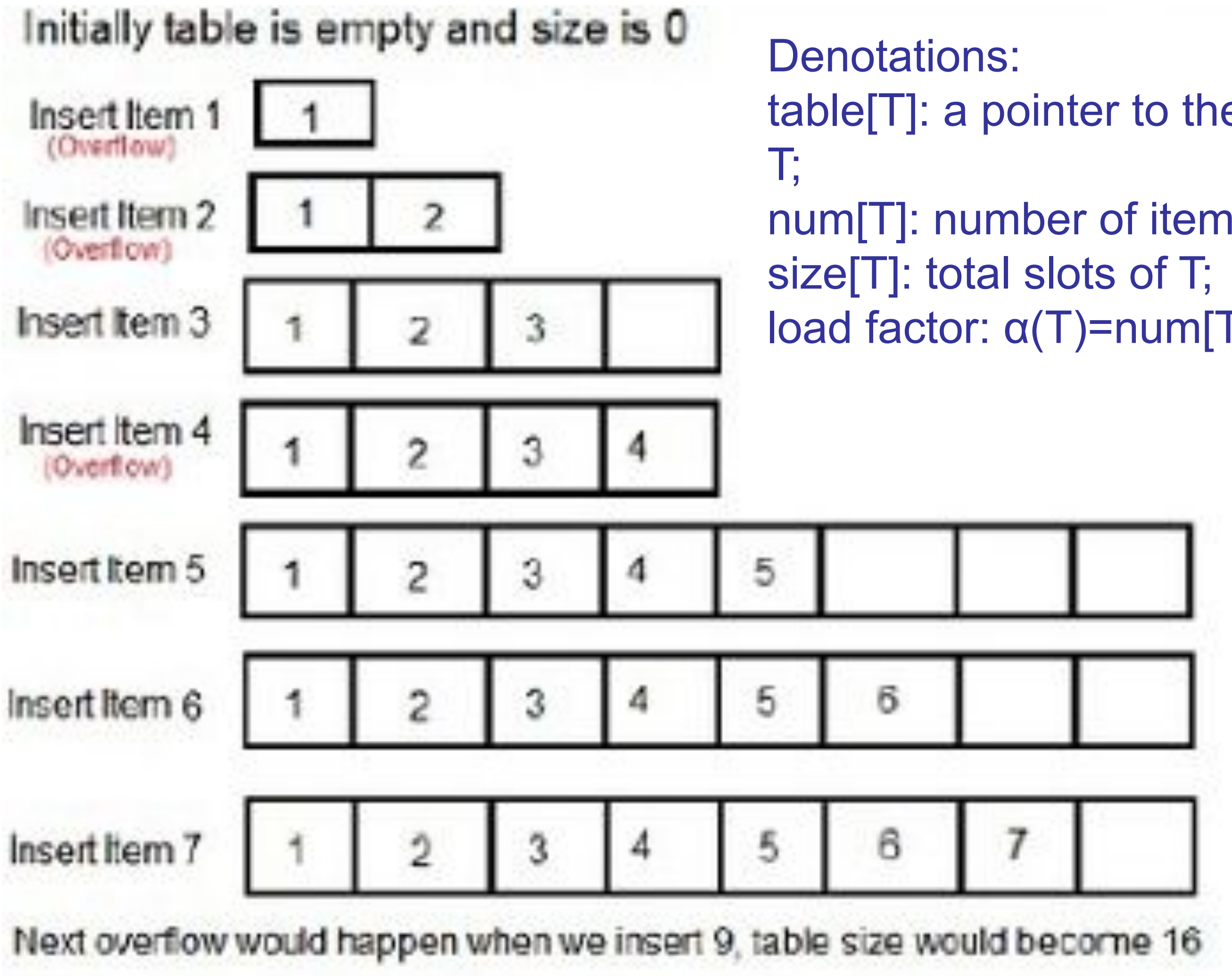
- we have

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = c_i + b_i - b_{i-1} \le c_i + b_{i-1} - t_i + 1 - b_{i-1}$$
$$\le c_i - t_i + 1 \le t_i + 1 - t_i + 1 = 2$$

- the worst case amortized cost=n*2=O(n).

More Examples

Dynamic Tables



Denotations:
table[T]: a pointer to the entry of table T;
num[T]: number of items in T;
size[T]: total slots of T;
load factor: α(T)=num[T]/size[T]>=1/2

- Algorithm of insertion

```
TABLE-INSERT(T, x)
if size (T)=0
   then    allocate table (T) with 1 slot
           size (T)←1
if num[T]=size[T]
   then    allocate new table with 2*size[T] slots
           insert all old items into the new table
           release table[T]
           table[T]←new table
           size[T]=2*size[t]
insert x into table[T]
num[T]←num[T]+1
```

- amortized analysis--aggregate analysis
  - for one operation, the worst-case cost is n, and the total cost is $O(n^2)$--correct but not tight.

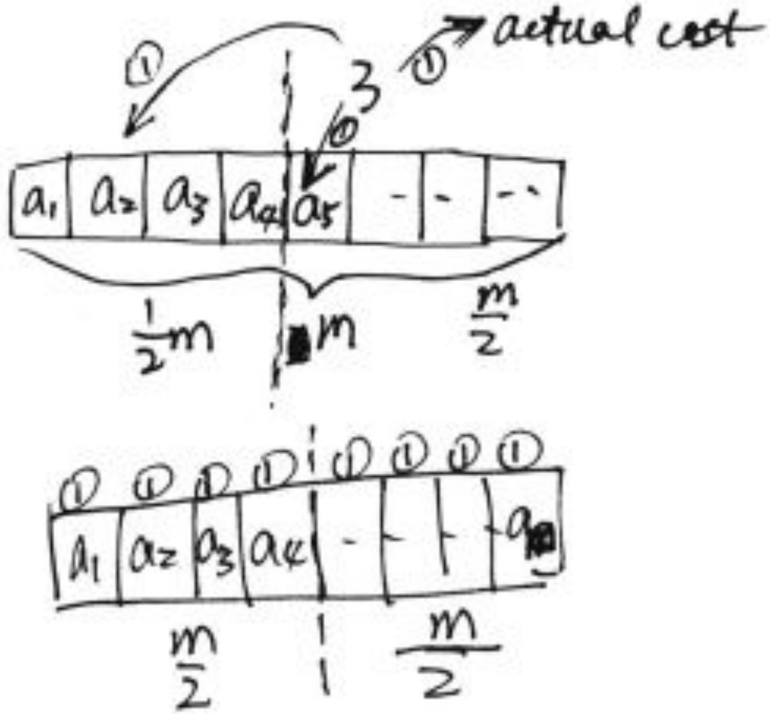
So $ci = 1 + \begin{cases} i-1, \ if \ i-1 \ is \ power \ of \ 2 \\ 0, \quad Otherwise \end{cases}$

$$\frac{\sum_{i=1}^{n} c_i}{n} \leq \frac{n + \sum_{j=1}^{\lfloor log_2(n-1) \rfloor} 2^j}{n} = \frac{3n}{n} = 3$$

$$2^0 + 2^1 + 2^2 + \ldots + 2^a = 2^{a+1} - 1$$

$$\Rightarrow \quad 2^1 + 2^2 + \ldots + 2^{\lfloor \lg(n-1) \rfloor} = 2^{\lfloor \lg(n-1) \rfloor + 1} - 1 - 1 = 2^{\lfloor \lg(n-1) \rfloor + 1} - 2 = 2(n-1) - 2 = O(2n)$$

- amortized analysis--accounting method



Amortized cost:
insertion: 3
copy: 0

The total credit at any moment is in [0,n]=O(n)

- amortized analysis--potential method
  - $\varphi(T)=2*num(T)-size(T)$;

    (when $2*num(T)=size(T)$, $\varphi(T)=0$)
  - $\alpha(T)>=1/2$, then $\varphi(T)$ is nonnegative;
  - total amortized cost is the upper bound of actual cost;

- when the ith insertion does not trigger an expansion:

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

- when it does:

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

Deletion

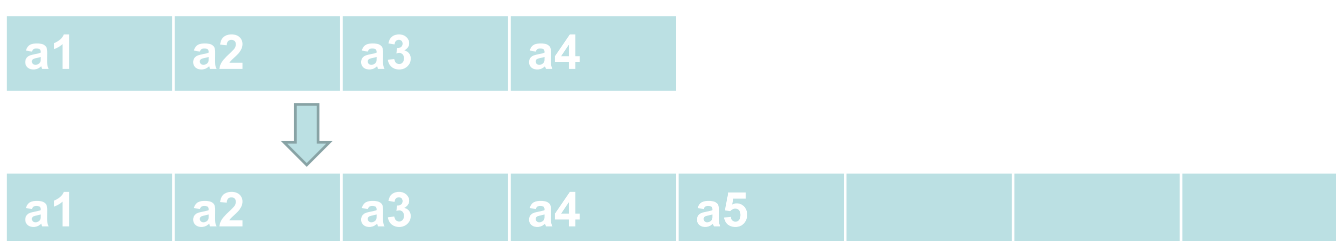
- Contraction
  - To keep load factor >=1/2, we contract the table to half when num_i is half of $size_i$?
  - example: when n=8, we conduct the following operations:

    I,I,I,I,I,D,D,I

Deletion

- Contraction
  - To keep load factor >=1/2, we contract the table to half when num_i is half of $size_i$?
  - example: when n=8, we conduct the following operations:

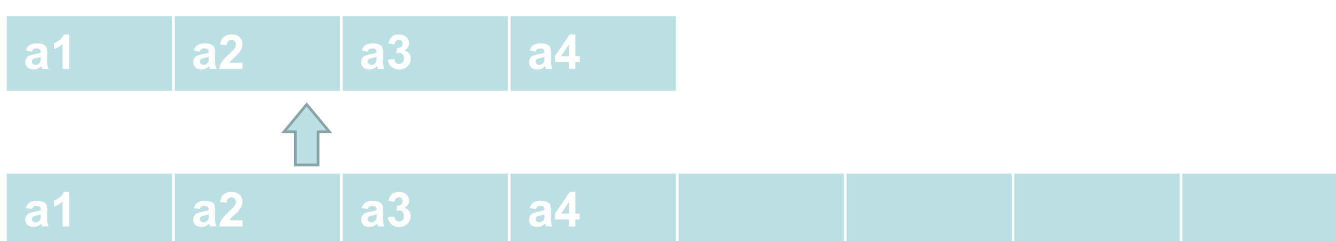
    I,I,I,I,I,D,D,I,



Deletion

- Contraction
  - To keep load factor >=1/2, we contract the table to half when num_i is half of $size_i$?
  - example: when n=8, we conduct the following operations:

    I,I,I,I,I,D,D,I



Deletion

- Contraction
  - To keep load factor >=1/2, we contract the table to half when num_i is half of $size_i$?
  - example: when n=8, we conduct the following operations:

    I,I,I,I,I,D,D,I

a1	a2	a3	

Deletion

- Contraction
  - To keep load factor >=1/2, we contract the table to half when num_i is half of $size_i$?
  - example: when n=8, we conduct the following operations:

    I,I,I,I,I,D,D,I

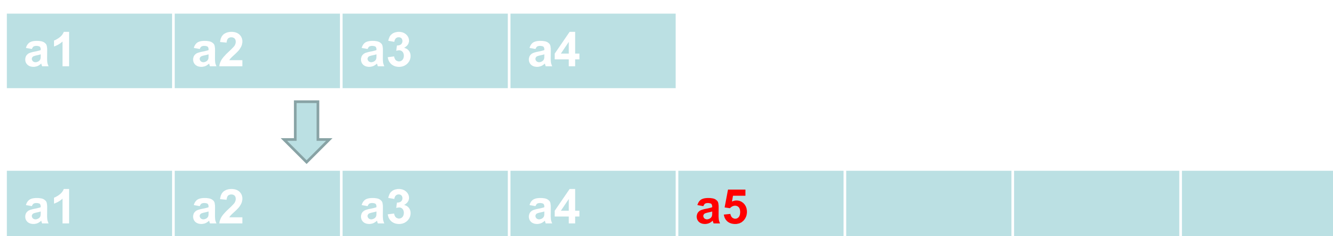
a1	a2	a3	a4

Deletion

- Contraction
  - To keep load factor >=1/2, we contract the table to half when num_i is half of $size_i$?
  - example: when n=8, we conduct the following operations:

I,I,I,I,I,D,D,I,I

How to fix it?



Actual cost is $O(n/2*n/2)=O(n^2)$

- Solution: let load factor be 1/4 when contraction occurs.
- amortized analysis--potential method
  - $\varphi_i = \begin{cases} 2*num_i - size_i, & \alpha_i >= 1/2 \\ size_i/2 - num_i & \alpha_i < 1/2 \end{cases}$
- for insertion:

if $\alpha_{i-1} >= 1/2$, $\hat{c}_i = 3$

if $\alpha_{i-1} < 1/2$, and $\alpha_i < 1/2$,

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + size_i/2 - num_i - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + size_i/2 - num_i - [size_i/2 - (num_i - 1)]$$
$$= 0$$

$\alpha_i >= 1/2$,

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + 2num_i - size_i - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + 2(num_{i-1} + 1) - size_{i-1} - (size_{i-1}/2 - num_{i-1})$$
$$= 3num_{i-1} - \frac{3}{2}size_{i-1} + 3 = 3\alpha_{i-1}size_{i-1} - \frac{3}{2}size_{i-1} + 3$$
$$< 3 \times \frac{1}{2}size_{i-1} - \frac{3}{2}size_{i-1} + 3 = 3$$

- amortized analysis--potential method
  - φ_i= 2*num_i-$size_i$, α_i>=1/2
    $size_{i/}2$-num_i α_i<1/2
- for deletion:

if α_{i-1}>=1/2, the cost is a constant;

if α_{i-1}<1/2, and does not contract,

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = 1 + size_i/2 - num_i - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + size_i/2 - num_i - [size_i/2 - (num_i + 1)]$$
$$= 2$$

contracts,

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1}) = num_i + 1 + size_i/2 - num_i - (size_{i-1}/2 - num_{i-1})$$
$$= num_i + 1 + num_i + 1 - num_i - [(2num_i + 2) - (num_i + 1)]$$
$$= 1$$