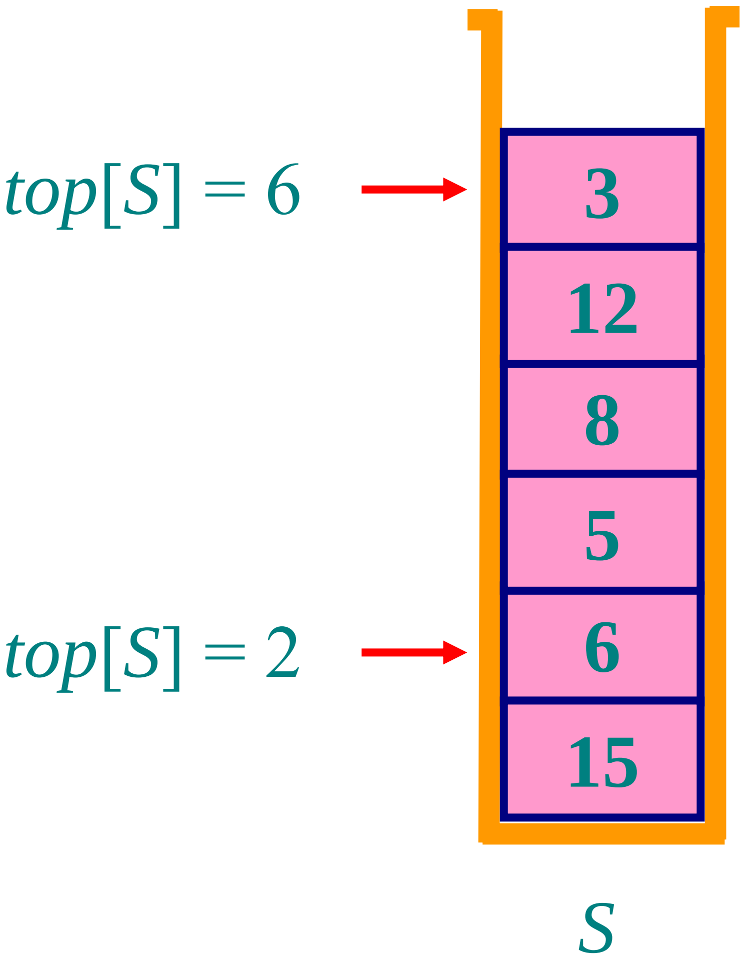
Data Structures and Algorithm

Xiaoqing Zheng

zhengxq@fudan.edu.cn

MULTIPOP



MULTIPOP(S, x)

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

MULTIPOP$(S, 4)$

Analysis of MULTIPOP

A sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack.

- The worst-case cost of a MULTIPOP operation is $O(n)$
- The worst-case time of any stack operation is therefore $O(n)$
- A sequence of n operations costs is $O(n^2)$

This bound is not tight!

Amortized analysis

- An amortized analysis is any strategy for analyzing a sequence of operations to show that the ***average cost per operation*** is small, even though a single operation within the sequence might be expensive.
- Even though we're taking averages, however, ***probability*** is not involved!
- An amortized analysis ***guarantees*** the average performance of each operation in the worst case.

Types of amortized analyses

Three common amortization arguments:

- ***Aggregate method***
- ***Accounting method***
- ***Potential method***

The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific ***amortized cost*** to be allocated to each operation.

Aggregate analysis

- We show that for all n, a sequence of n operation takes worst-case time $T(n)$ in total.
- Hence, the average cost, or amortized cost, per operation is $T(n)/n$

MULTIPOP (aggregate method)

- Each object can be popped at most once for each time it is pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n.
- Any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. The average cost of an operation is $T(n)/n = O(1)$

8-bit binary counter

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

Incrementing a binary counter

An array $A[0 \ldots k-1]$ of bit, where $length[A] = k$, as the counter.

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

INCREMENT(A)

1. $i \leftarrow 0$
2. **while** $i < length[A]$ **and** $A[i] = 1$
3.   **do** $A[i] \leftarrow 0$
4. $i = i + 1$
5. **if** $i < length[A]$
6.   **then** $A[i] \leftarrow 1$

Binary counter (aggregate method)

For $i = 0, 1 \ldots, \lfloor \lg n \rfloor$, bit $A[i]$ flips $\lfloor n / 2^i \rfloor$ times in a sequence of n INCREMENT operation on an initially zero counter.

The total number of flips in the sequence is thus

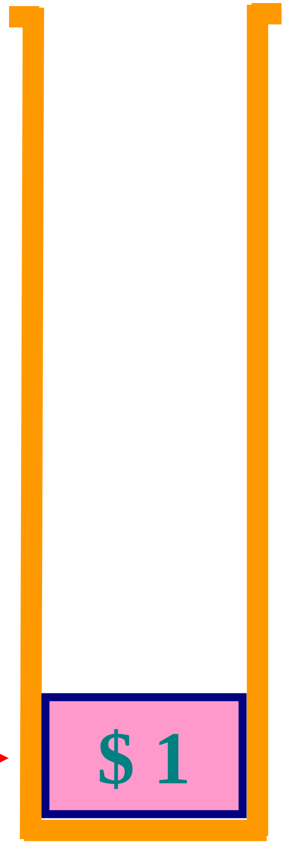
$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$
$$= 2n$$
$$= O(n)$$

The amortized cost per operation is

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

MULTIPOP (accounting method)

PUSH



top[*S*] = 1

S

$ 1

Actual costs

PUSH	1,
POP	1,
MULTIPOP	*min*(*k*, *s*).

Amortized costs

PUSH	2,
POP	0,
MULTIPOP	0.

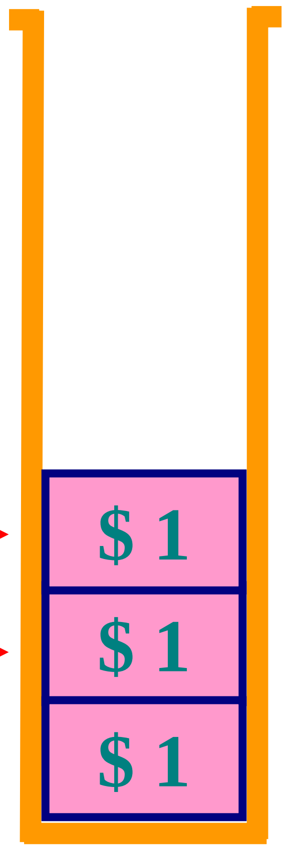
MULTIPOP (accounting method)

PUSH

PUSH

top[*S*] = 3

top[*S*] = 2



S

Actual costs

PUSH	1,
POP	1,
MULTIPOP	*min*(*k*, *s*).

Amortized costs

PUSH	2,
POP	0,
MULTIPOP	0.

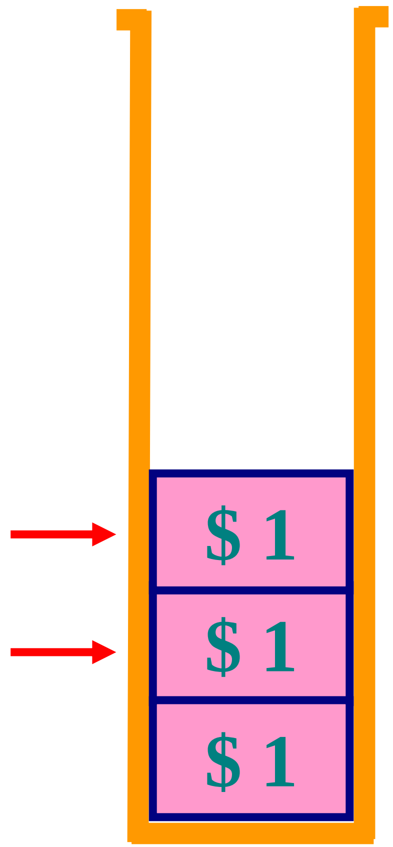
$ 1 + $ 1 + $ 1

MULTIPOP (accounting method)

POP

top[*S*] = 3

top[*S*] = 2



S

Actual costs

PUSH	1,
POP	1,
MULTIPOP	*min*(*k*, *s*).

Amortized costs

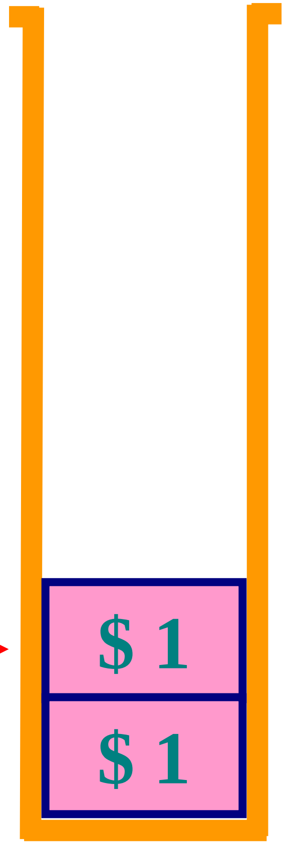
PUSH	2,
POP	0,
MULTIPOP	0.

$ 1 + $ 1 + $ 1 + $ 1

MULTIPOP (accounting method)

MULTIPOP 2

$top[S] = 2$



$top[S] = 0$

S

Actual costs

PUSH	1,
POP	1,
MULTIPOP	$min(k, s)$.

Amortized costs

PUSH	2,
POP	0,
MULTIPOP	0.

$ 1 + $ 1 + $ 1 + $ 1 + $ 1 + $ 1

Accounting method

- We assign ***differing charges*** to different operations, with some operations charged more or less than they actually cost.
- The amount we charge an operation is called its ***amortized cost***.
- When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as ***credit***.

Accounting method

- We denote the actual cost of the *i*th operation by c_i and the amortized cost of the *i*th operation by $\widehat{c_i}$, we require

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$$

Why?

Binary counter (accounting method)

Amortized costs	
Set a bit to 1	2,
Flip the bit back to 0	0.

Potential method

- For each $i = 1, 2, \ldots, n$, we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}.
- A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with data structure D_i.
- A ***amortized cost*** $\widehat{c}_i$ of ith operation with respect to potential function is Φ defined by

$$\widehat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Potential method

- The total amortized cost of the n operation is

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

- Define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

MULTIPOP (potential method)

We define the potential function Φ on a stack to be the number of objects in the stack.

- **PUSH** operation

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

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

- **MULTIPOP**(S, k) operation and $k' = min(k, s)$

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

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k' = 0$$

- **POP** operation and $k' = min(1, s)$

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

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k' = 0$$

Binary counter (potential method)

- We define the potential of counter after the ith INCREMENT operation to be b_i, the number of 1's in the counter after the ith operation.
- Suppose that the ith INCREMENT operation reset t_i bit.

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i\end{aligned}$$

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

8-bit binary counter

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

$$\begin{aligned}
&\Phi(D_i)-\Phi(D_{i-1}) \\
&= (b_{i-1}-t_i+1)-b_{i-1} \\
&= (2-1+1)-2 \\
&= 0
\end{aligned}$$

$$\begin{aligned}
\hat{c}_i &= c_i+\Phi(D_i)-\Phi(D_{i-1}) \\
&= 2+0 \\
&= 2
\end{aligned}$$

8-bit binary counter

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**

$$\begin{aligned}
&\Phi(D_i) - \Phi(D_{i-1}) \\
&= (b_{i-1} - t_i + 1) - b_{i-1} \\
&= (3 - 3 + 1) - 3 \\
&= -2
\end{aligned}$$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 4 - 2 \\
&= 2
\end{aligned}$$

Binary counter (potential method)

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c_i} - \Phi(D_n) + \Phi(D_0)$$

$$= \sum_{i=1}^{n} 2 - b_n + b_0$$

$$= 2n - b_n + b_0$$

$$= O(n)$$

Hash tables by chaining

Expected time to search for a record with a given key

$$\Theta(1+\alpha)$$

apply hash function and access slot

search the list

Expected search time $= \Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.

$\alpha = n/m =$ average number of keys per slot.

Hash tables by open-addressed

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Theorem. Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most .

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

How large should a hash table be?

Goal: Make the table as small as possible, but large enough so that it won't overflow (or otherwise become inefficient).

Problem: What if we don't know the proper size in advance?

Solution: ***Dynamic tables***.

IDEA: Whenever the table overflows, "grow" it by allocating a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

Example of a dynamic table

1. **INSERT**

1

Example of a dynamic table (cont.)

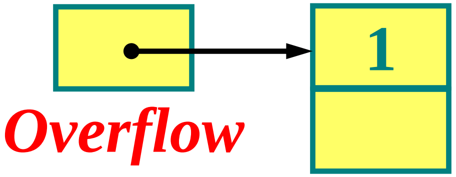
1. INSERT
2. INSERT

1

Overflow

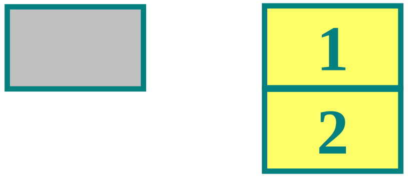
Example of a dynamic table (cont.)

1. INSERT
2. INSERT



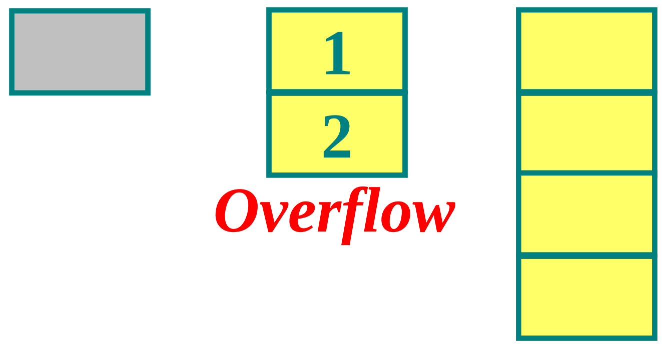
Example of a dynamic table (cont.)

1. INSERT
2. INSERT



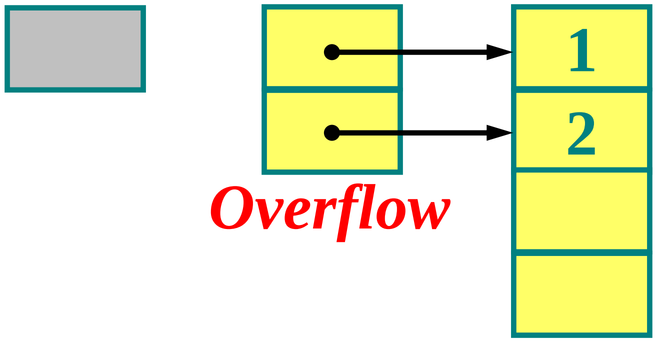
Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT



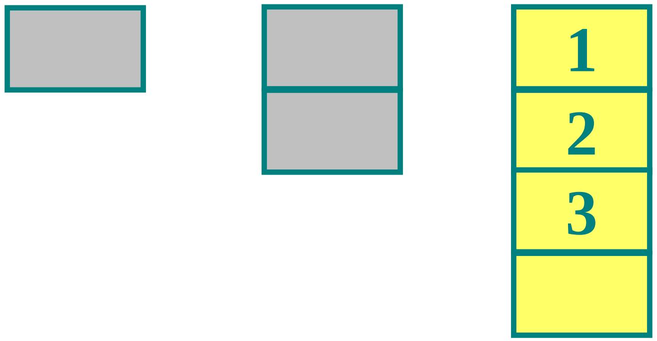
Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT



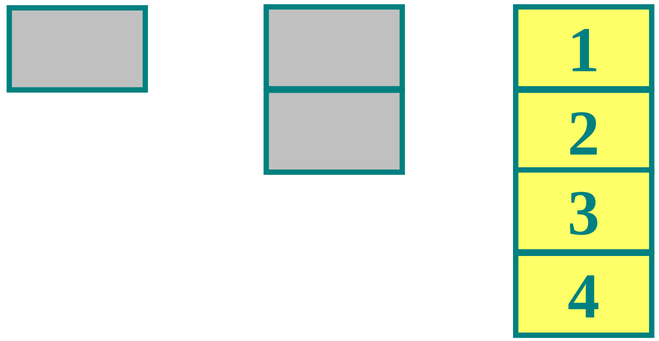
Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT



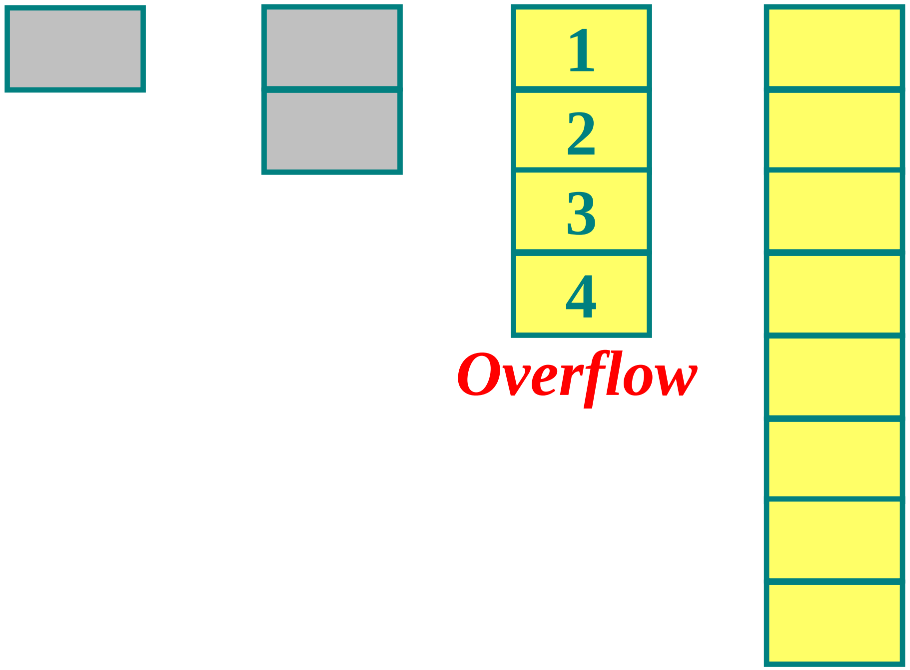
Example of a dynamic table (cont.)

1. **INSERT**
2. **INSERT**
3. **INSERT**
4. **INSERT**



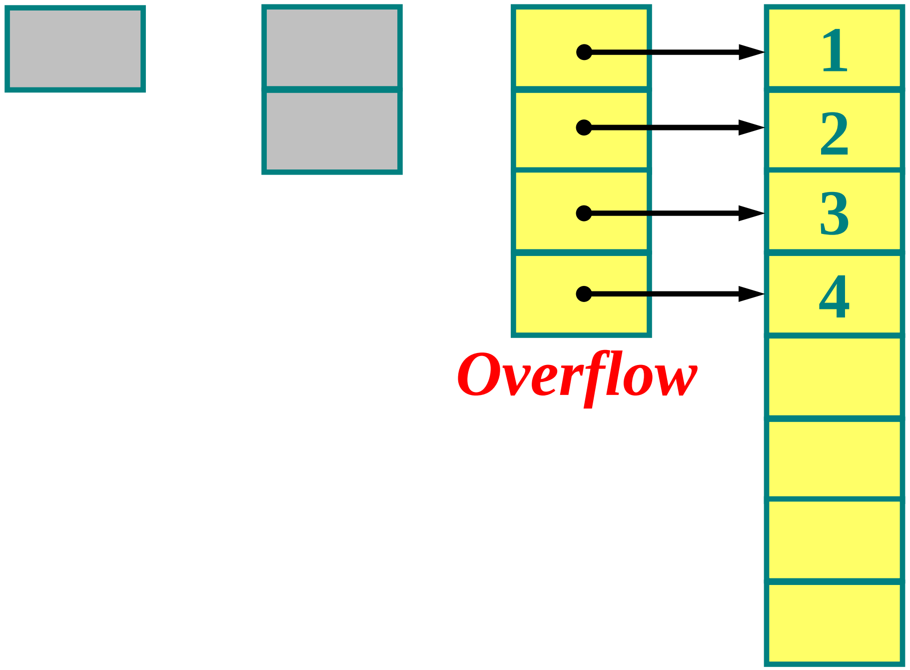
Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



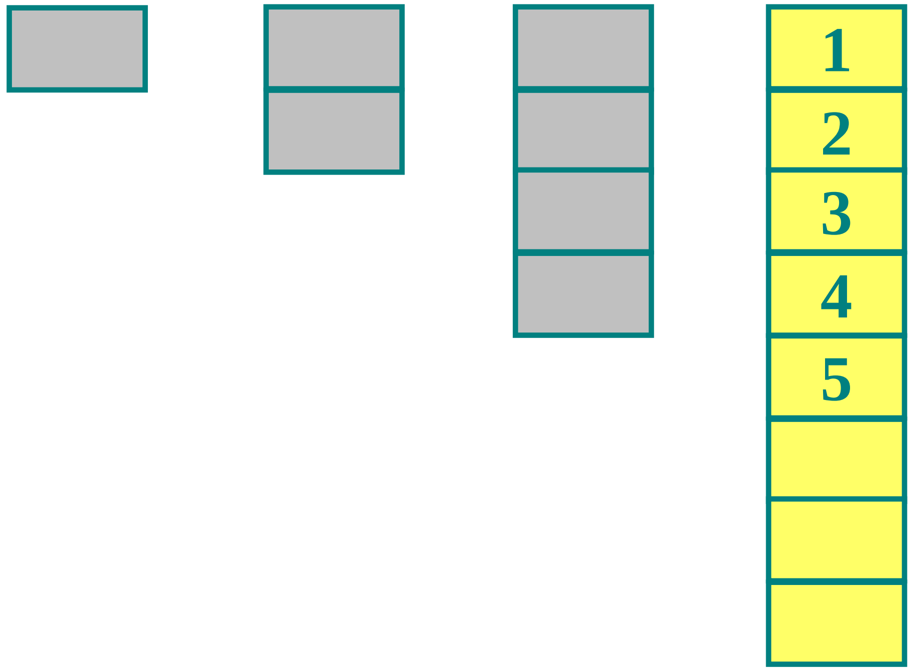
Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



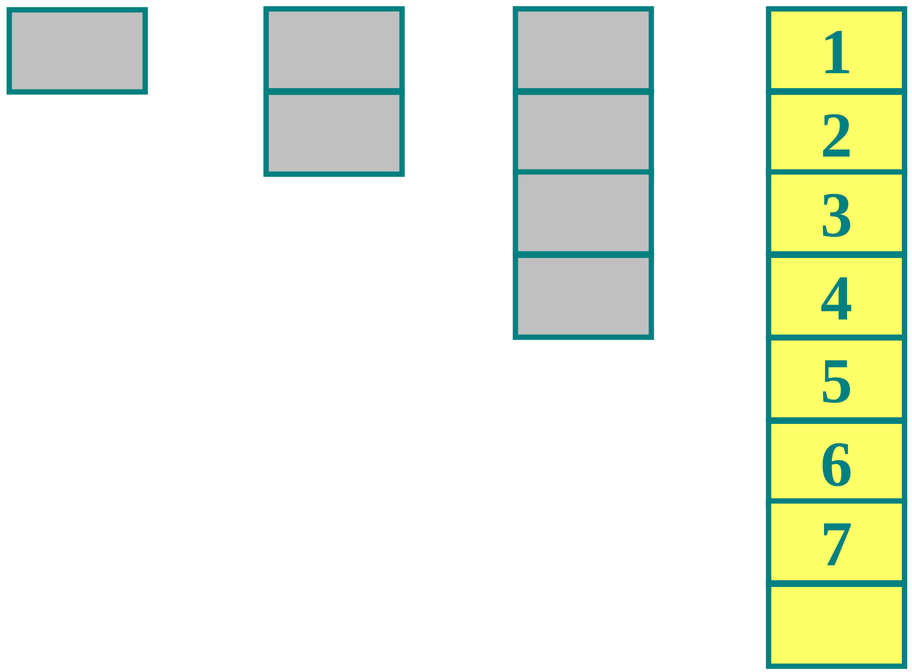
Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



Example of a dynamic table (cont.)

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Worst-case analysis

Consider a sequence of n insertions. The worst-case time to execute one insertion is $\Theta(n)$. Therefore, the worst-case time for n insertions is $n \cdot \Theta(n) = \Theta(n^2)$.

This bound is not tight! In fact, the worst-case cost for n insertions is only $\Theta(n)$.

Let's see why.

Tighter analysis

Let c_i = the cost of the ith insertion

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

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

Tighter analysis (cont.)

Let c_i = the cost of the ith insertion

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

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1	1	1	1	1	1	1	1	1
		1	2		4				8	

Tighter analysis (aggregate method)

The total cost of n TABLE-INSERT operations is therefore

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

Thus, the average cost of each dynamic-table operation is $\Theta(n)/n = \Theta(1)$.

Accounting analysis of dynamic tables

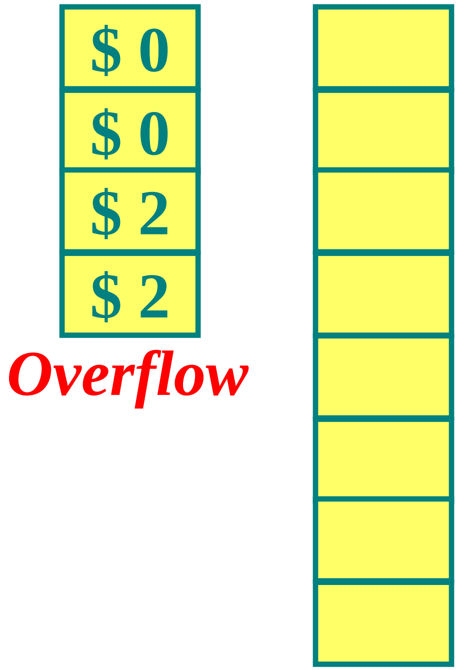
Charge an amortized cost of $\hat{c}_i = \$3$ for the ith insertion.

- \$1 pays for the immediate insertion.
- \$2 is stored for later table doubling.

When the table doubles, \$1 pays to move a recent item, and \$1 pays to move an old item.

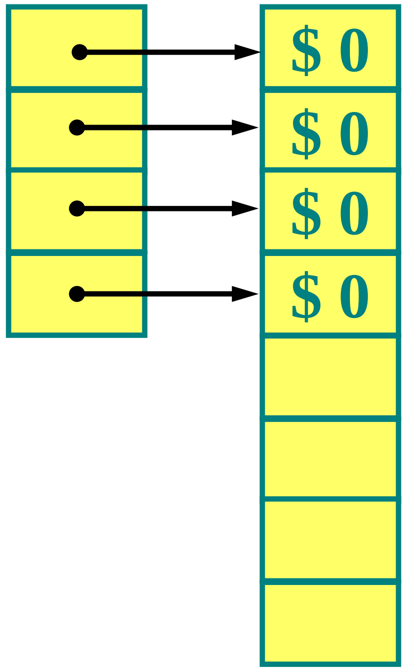
Accounting analysis of dynamic tables

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



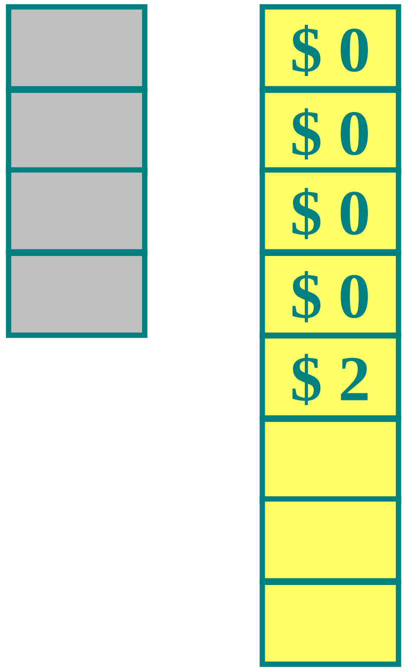
Accounting analysis of dynamic tables

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT



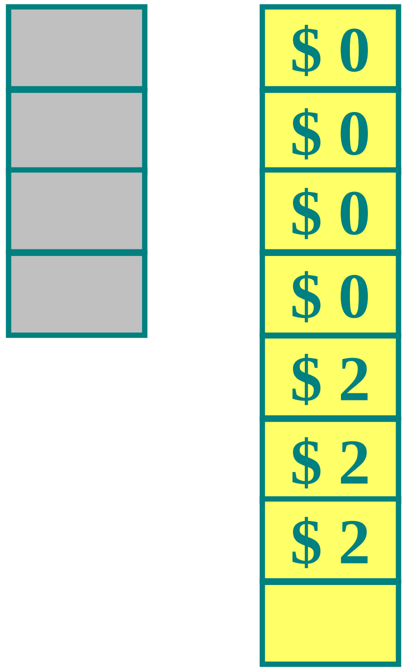
Accounting analysis of dynamic tables

1. **INSERT**
2. **INSERT**
3. **INSERT**
4. **INSERT**
5. **INSERT**



Accounting analysis of dynamic tables

1. INSERT
2. INSERT
3. INSERT
4. INSERT
5. INSERT
6. INSERT
7. INSERT



Dynamic table (potential method)

We start by defining a ***potential function*** Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full. The function

$$\Phi(T) = 2 \cdot num[T] - size[T]$$

- num_i denote the number of items stored in the table after the ith operation
- $size_i$ denote the total size of the table after the ith operation
- Φ_i denote the potential after the ith operation

Dynamic table (potential method)

- If the *i*th TABLE-INSERT operation does not trigger an expansion, the amortized cost is

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

- If the *i*th TABLE-INSERT operation triggers an expansion, the amortized cost is

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

Table expansion and contraction

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 form the old table into the new one.

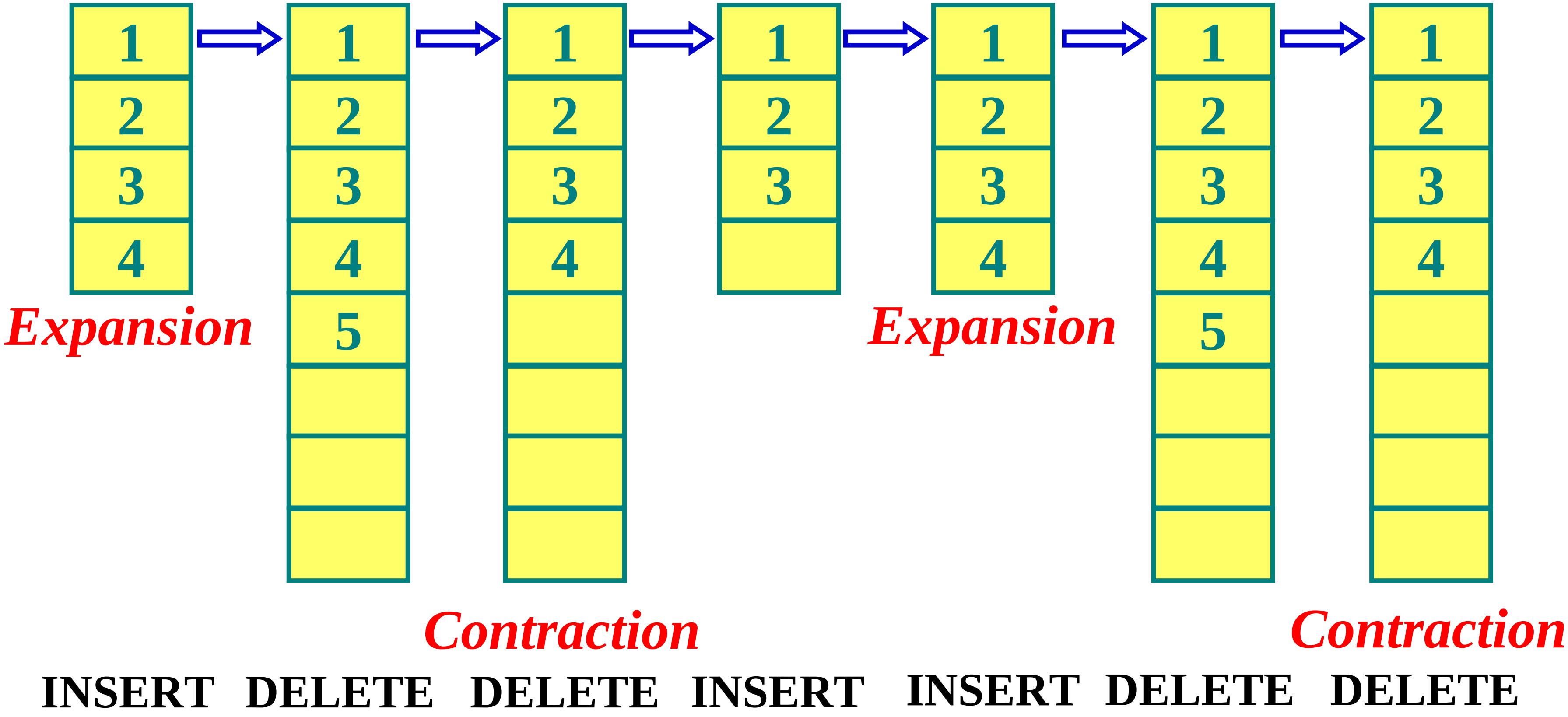
A natural strategy for expansion and contraction is to ***double*** the table size when an item is inserted into a full table and ***halve*** the size when a deletion would cause the table to become less than half full.

This strategy cause the amortized cost of an operation to be quite large.

Problem of natural strategy

We perform the following sequence.

I, D, D, I, I, D, D, I, I, …



Improved strategy

- Double the table size when an item is inserted into a full table
- Halve the table size when a deletion causes the table to become less than 1/4 full
- Potential function

$$\Phi(T) = \begin{cases} 2 \cdot num[T] - size[T] & \text{if } \alpha(T) \geq 1/2 \\ size[T]/2 - num[T] & \text{if } \alpha(T) < 1/2 \end{cases}$$

Any question?

Xiaoqing Zheng
Fundan University