Design and Analysis of Algorithms

CS3230

Lecture 7
Amortized Analysis

Warut Suksompong

Asymptotic notation for multiple parameters

- What does $O(m+n)$ or $O(mn)$ mean?

- For two functions $f(m,n)$ and $g(m,n)$, we say that $f(m,n) = O(g(m,n))$ if there exist constants c, m_0, n_0 such that
$$0 \leq f(m,n) \leq c \cdot g(m,n)$$
for all $m \geq m_0$ or $n \geq n_0$.

- Similar definitions for other asymptotic notations (but not our focus). See Exercise 3.1-8 of CLRS.

Amortized Analysis

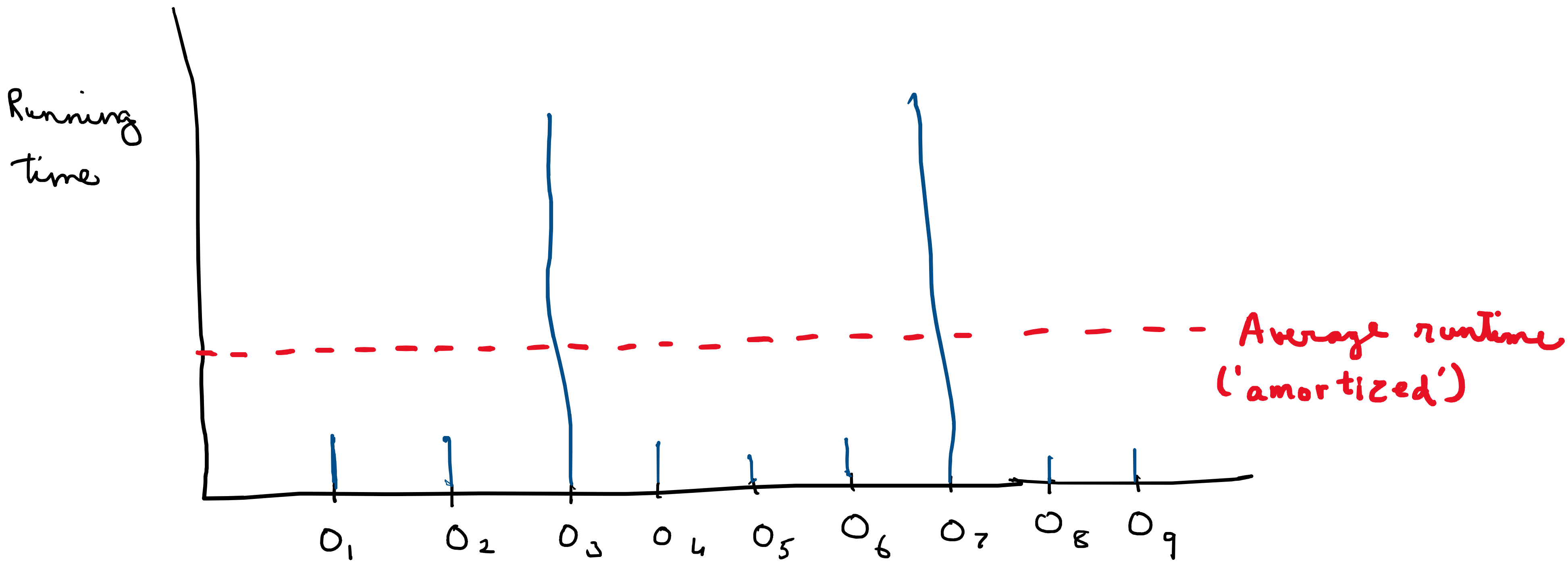
- There is a sequence of n operations $\underbrace{o_1, o_2, \ldots, o_n}$
- Let t(i) be the time complexity of i-th operation o_i
- Let T(n) be the time complexity of all n operations

$$T(n) = \sum_{i=1}^{n} t(i)$$
$$= n\, f(n)$$

f(n) = worst case time complexity of any of the n operations

Could be grossly wrong!

Amortized Analysis



Binary counter

A motivating example for amortized analysis

k-bit Binary Counter: How do we increment?

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	
0	0	0	0	0	0	
1	0	0	0	0	1	
2	0	0	0	1	0	
3	0	0	0	1	1	
4	0	0	1	0	0	
5	0	0	1	0	1	
6	0	0	1	1	0	
7	0	0	1	1	1	
8	0	1	0	0	0	
9	0	1	0	0	1	
10	0	1	0	1	0	
11	0	1	0	1	1	

Objective: Count total Bit flips (0 → 1, 1 → 0) during n increments

$T(n)$ = total no. of bit flips during n increments

Aim: To get a tight bound on $T(n)$

```
INCREMENT(A)
1.  i ← 0
2.  while i < length[A] and A[i] = 1 do
3.          A[i] ← 0          ⊳ flip 1→0
4.          i ← i + 1
5.  if  i < length[A]
6.      then  A[i] ← 1        ⊳ flip 0→1
```

k-bit Binary Counter: No. of bit flips

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	*Cost*
0	0	0	0	0	0	*0*
1	0	0	0	0	1	*1*
2	0	0	0	1	0	*2*
3	0	0	0	1	1	*1*
4	0	0	1	0	0	*3*
5	0	0	1	0	1	*1*
6	0	0	1	1	0	*2*
7	0	0	1	1	1	*1*
8	0	1	0	0	0	*4*
9	0	1	0	0	1	*1*
10	0	1	0	1	0	*2*
11	0	1	0	1	1	*1*

Objective: Count total Bit flips (0 → 1, 1 → 0) during n increments

$\boldsymbol{T(n)}$ = total no. of bit flips during $\boldsymbol{n}$ increments

Aim: To get a tight bound on $\boldsymbol{T(n)}$

Attempt 1:

Let $\boldsymbol{t(i)}$ = no. of bit flips during $\boldsymbol{i}$th increment

$$\boldsymbol{T(n) = \sum_{i=1}^{n} t(i)}$$

In the worst case, $\boldsymbol{t(i)} = k$

→ $\boldsymbol{T(n) = O(n\,k)}$

Is this a tight bound?

k-bit Binary Counter: No. of bit flips

Ctr	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*
5	0	0	1	**0**	**1**	*1*	*8*
6	0	0	1	1	**0**	*2*	*10*
7	0	**0**	**1**	**1**	**1**	*1*	*11*
8	0	1	0	0	**0**	*4*	*15*
9	0	1	0	**0**	**1**	*1*	*16*
10	0	1	0	1	**0**	*2*	*18*
11	0	1	**0**	**1**	**1**	*1*	*19*

Objective: Count total Bit flips (0 → 1, 1 → 0) during n increments

$T(n)$ = total no. of bit flips during n increments

Aim: To get a tight bound on $T(n)$

Attempt 2:

Let $f(i)$ = no. of times ith bit flips

$$T(n) = \sum_{i=0}^{k-1} f(i)$$

$f(0) = n$
$f(1) = n/2$
$f(2) = n/4$
$f(i) = n/2^i$

Much better than $O(nk)$ since $k \approx \log n$

→ $T(n) = n \sum_{i=0}^{k-1} 2^{-i} < 2n$

Amortized Analysis

- ***Amortized analysis*** is a 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.

- In our last example, average cost per increment = $\frac{T(n)}{n} < 2 = O(1)$

We say **amortized cost** of each increment = $O(1)$

Amortized Analysis

- ***Amortized analysis*** is a 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.

- Note, ***no probability is involved***!

Do not get confused with the average-case analysis

- An amortized analysis *guarantees* the *average performance* of each operation in the *worst case*.

- In our last example, average cost per increment = $\frac{T(n)}{n} < 2 = O(1)$

We say **amortized cost** of each increment = $O(1)$

Another Example: Queues

- Consider a queue with two operations:
  - *INSERT(x): Inserting* one element x and
  - *EMPTY(): Emptying* the queue, implemented by deleting all the elements one by one.
- What is the <u>worst case</u> running time for a sequence of n operations?

- Cost of a single INSERT: $\Theta(1)$
- Cost of a single EMPTY: $\Theta(n)$ (because at most n elements inserted)

Amortized Analysis: Queues

- An EMPTY is a sequence of DELETE’s where each DELETE removes one element from the front of the queue

- Notice: #DELETE’s $\leq$ #INSERT’s

- If there are k INSERT’s in the sequence, sum of cost of all the EMPTY’s is $\leq k$.

- Total cost: $\leq k + k = 2k \leq 2n$. Amortized cost is $O(1)$.

Types of Amortized Analyses

- Three common amortization arguments:
  - *Aggregate* method
  - *Accounting* method
  - *Potential* method

- We have just seen two examples of aggregate analysis.

- 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.

Accounting (Banker's) Method

- Charge ith operation a fictitious ***amortized cost*** *c(i)*.
- Idea is that the amortized cost *c(i)* is a fixed cost for each operation, while the true cost *t(i)* varies depending on when the operation is called.
- The amortized cost *c(i)* must satisfy:

$$\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i) \text{ for all } n$$

The total amortized cost provides an upper bound on the total true cost.

Accounting (Banker's) Method

- Typically, the fixed amortized cost $c(i)$ will be more than the true cost $t(i)$. Only occasionally, the true cost $t(i)$ will be larger than $c(i)$.

- The extra amount we pay for cheap operations early on can be thought of as **credit** paid in advance for the rare, expensive operations later!

- Analysis must ensure that there's always enough credit to pay for true cost.

- NOTE: Different operations can have different amortized costs.

Amortized Analysis of Queues

- For INSERT, set amortized cost to 2. (True cost is 1.)

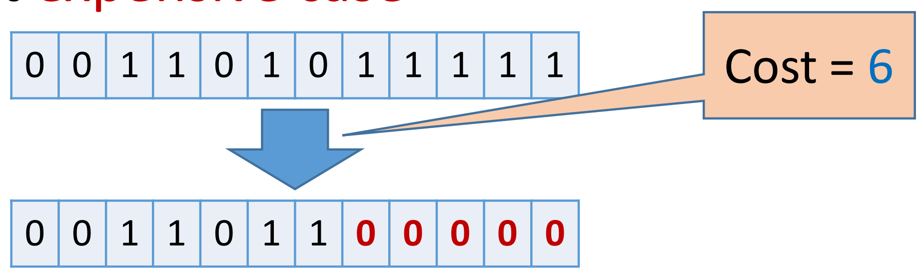
- For EMPTY, set amortized cost to 0. (True cost is size of queue.)

- Whenever an element is inserted, we pay an extra 1. This extra 1 can be used as credit that can pay for deleting it later.

- Total cost is at most 2*#INSERT $\leq 2n$.

Accounting Analysis of Binary Increment

- First identify the most expensive case



Observe, 5 bits are reset to 1 → 0, and only one bit is set to 0 → 1

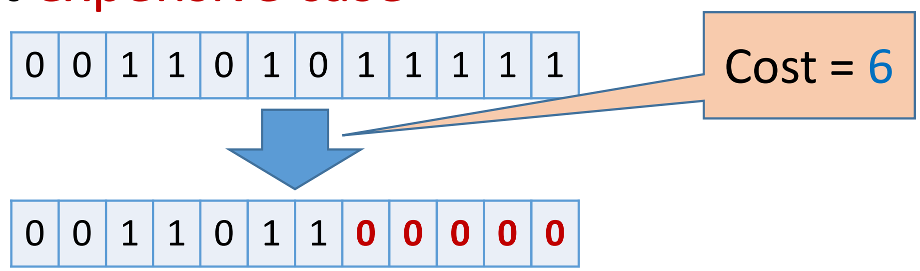
k-bit Binary Counter: How to increment?

Ctr	A[4]	A[3]	A[2]	A[1]	A[0]	
0	0	0	0	0	0	
1	0	0	0	0	1	
2	0	0	0	1	0	
3	0	0	0	1	1	
4	0	0	1	0	0	
5	0	0	1	0	1	
6	0	0	1	1	0	
7	0	0	1	1	1	
8	0	1	0	0	0	
9	0	1	0	0	1	
10	0	1	0	1	0	
11	0	1	0	1	1	

INCREMENT(A)
1. $i \leftarrow 0$
2. **while** $i < length[A]$ **and** $A[i] = 1$ **do**
3. $\quad A[i] \leftarrow 0$ ⊳ *flip* 1→0
4. $\quad i \leftarrow i + 1$
5. **if** $i < length[A]$
6. $\quad$ **then** $A[i] \leftarrow 1$ ⊳ *flip* 0→1

Accounting Analysis of Binary Increment

- First identify the most expensive case



Observe, 5 bits are reset to 1 → 0, and only one bit is set to 0 → 1

Can we do 1 → 0
“free of cost” ?

Accounting Analysis of Binary Increment

- Charge $2 for each 0 → 1

$1 pays for the actual bit setting.
$1 is stored in the bank.

At some point of time, this bit must have been set

- **Observation:** At any point, every 1 bit in the counter has $1 on its bank.
- Use that $1 as credit to pay for resetting 1 → 0. (reset is "*free*")

Example:

0 0 0 1$^{\$1}$ 0 1$^{\$1}$ 0

0 0 0 1$^{\$1}$ 0 1$^{\$1}$ 1$^{\$1}$ Amortized Cost = $2

0 0 0 1$^{\$1}$ 1$^{\$1}$ 0 0 Amortized Cost = $2

Accounting Analysis of Binary Increment

- **Invariant we need to keep:** Bank balance never drops below 0. Thus, the sum of the amortized costs provides an upper bound on the sum of the true costs.

Observation: At any point, every 1 bit in the counter has $1 on its bank.

Claim: After **i** increments, the amount of money in the bank is the number of 1's in the binary representation of **i**.

Accounting Analysis of Binary Increment

- **Claim**: After i increments, the amount of money in the bank is the number of 1's in the binary representation of i.
- **Proof**:
  - Every time we set a bit 0 → 1, we pay \$2.
  - \$1 is used to flip the bit while \$1 is stored in bank.
  - Every time we reset a bit from 1 → 0, we use \$1 from bank to flip the bit.
  - Hence, the amount of money in the bank is the number of 1's in the binary representation of i.

Accounting Analysis of Binary Increment

- Since the number of 1's in the binary representation of i is non-negative, previous slide shows that the bank balance is always non-negative.

- Conclusion:
  - Amortized cost for each increment = 2 = $O(1)$
  - Amortized cost for n increments = $2n = O(n)$
  - Actual cost for n increments = $O(n)$

Actual cost ≤ Amortized cost

Potential Method

$\boldsymbol{\phi}$: Potential function associated with the algorithm/data-structure (as opposed to with specific object in data-structure)

$\boldsymbol{\phi(i)}$: Potential at the end of $\boldsymbol{i}$th operation

$\boldsymbol{c_i}$: Amortized cost of $\boldsymbol{i}$th operation

$\boldsymbol{t_i}$: True cost of $\boldsymbol{i}$th operation

Key relation: $\boldsymbol{c_i = t_i + \phi(i) - \phi(i-1)}$

Adding up the above relation for $\boldsymbol{i = 1, 2, \dots, n}$:

$$\boldsymbol{c_1 + c_2 + \cdots + c_n \geq t_1 + t_2 + \cdots + t_n + \phi(n) - \phi(0)}$$

Potential Method

Key relation: $c_i = t_i + \phi(i) - \phi(i-1)$

$$c_1 + c_2 + \cdots + c_n = t_1 + t_2 + \cdots + t_n + \phi(n) - \phi(0)$$

Typically, we define $\phi(0) = 0$. So as long as $\phi(n) \geq 0$ for all n, the amortized cost is an upper bound of the true cost:

$$c_1 + c_2 + \cdots + c_n \geq t_1 + t_2 + \cdots + t_n$$

Potential Method on Queue

Potential function: $\boldsymbol{\phi(i)}$ = number of elements in the queue after $\boldsymbol{i}$-th operation

We have $\boldsymbol{\phi(0) = 0}$. Moreover, $\boldsymbol{\phi(n) \geq 0}$ for all $\boldsymbol{n}$.

Amortized cost for INSERT:

$$\boldsymbol{c_i = t_i + \phi(i) - \phi(i-1) = 1 + 1 = 2}$$

Amortized cost for EMPTY (suppose there are $\boldsymbol{k}$ elements in the queue):

$$\boldsymbol{c_i = t_i + \phi(i) - \phi(i-1) = k + 0 \ - k = 0}$$

For binary increment, potential function is the number of 1-bits (see CLRS for analysis)

Dynamic table

For Insertion only

How large should a table be?

- **Goal:** Make the table as small as possible, but large enough so that it won't overflow (otherwise becomes inefficient).
- **Problem:** What if we don't know the proper size in advance?

How large should a 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 (via **`malloc`** or **`new`**) a new, larger table. Move all items from the old table into the new one, and free the storage for the old table.

- Dynamic tables are implemented as ArrayList in Java or std:vector in C++.

Some Notations

- n: number of elements in the table.
- **createTable**(k): A system-call that creates a table of size k and returns its pointer.
- **size**(T): the size of table T.
- **copy**(T,T'): copies the contents of table T into T'.
- **free**(T): free the space (return the space to OS) occupied by table T.

A <u>trivial</u> way to perform Insert($\boldsymbol{x}$)

If ($\boldsymbol{n = 0}$)

    $\boldsymbol{T} \leftarrow$ **createTable**($\mathbf{1}$);

Else

// Table is full

    If($\boldsymbol{n} =$ **size**($\boldsymbol{T}$))

    { $\boldsymbol{T'} \leftarrow$ **createTable**($\boldsymbol{n + 1}$);

        copy($\boldsymbol{T}$,$\boldsymbol{T'}$);

        free($\boldsymbol{T}$);

        $\boldsymbol{T} \leftarrow \boldsymbol{T'}$

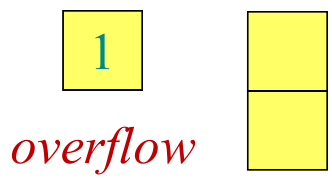
    }

Insert $\boldsymbol{x}$ into $\boldsymbol{T}$;

$\boldsymbol{n} \leftarrow \boldsymbol{n + 1}$;

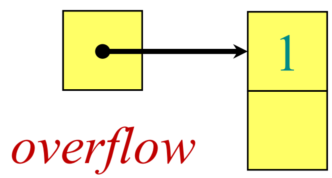
Example of a Dynamic Table

1. INSERT
2. INSERT



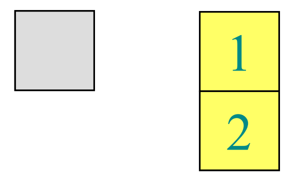
Example of a Dynamic Table

1. INSERT
2. INSERT



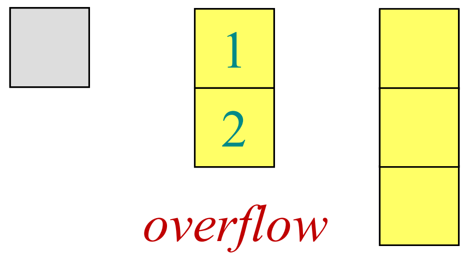
Example of a Dynamic Table

1. INSERT
2. INSERT



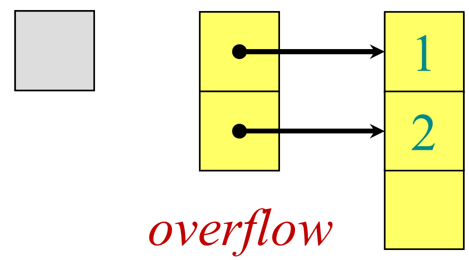
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



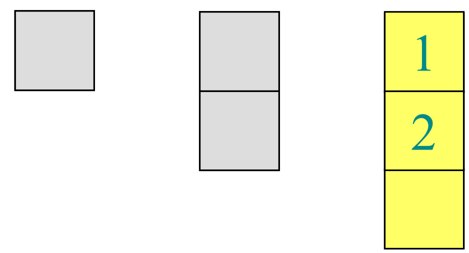
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



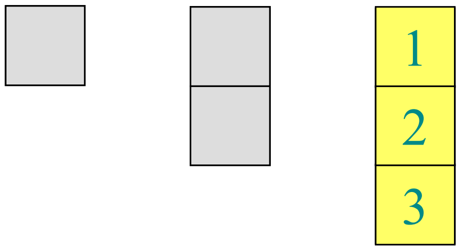
Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



Example of a Dynamic Table

1. INSERT
2. INSERT
3. INSERT



A trivial way to perform Insert(x)

If ($n = 0$)
    T ← createTable(1);
Else
    If(n = size(T))   // Table is full
    { T' ← createTable($n+1$);
        copy(T,T');
        free(T);
        T← T'
    }
Insert x into T;
n← $n+1$;

Idea: Every time table is full, create a new table of **double the size**

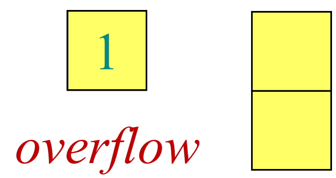
Time complexity of n insertions : $O(n^2)$

Proposed way to perform Insert(x)

```
If (n = 0)
    T ← createTable(1);
Else
    If(n = size(T))    // Table is full
    {   T' ← createTable( 2n);
        copy(T,T');
        free(T);
        T ← T'
    }
Insert x into T;
n ← n + 1;
```

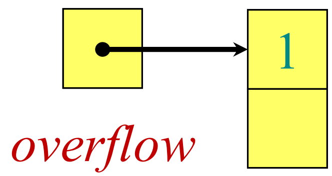
Example of an Efficient Dynamic Table

1. INSERT
2. INSERT



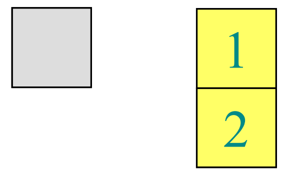
Example of an Efficient Dynamic Table

1. INSERT
2. INSERT



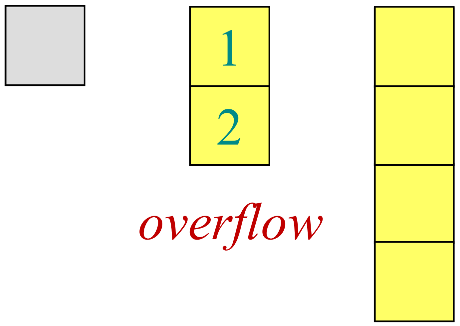
Example of an Efficient Dynamic Table

1. INSERT
2. INSERT



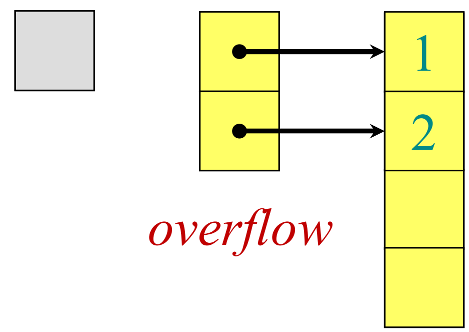
Example of an Efficient Dynamic Table

1. INSERT
2. INSERT
3. INSERT



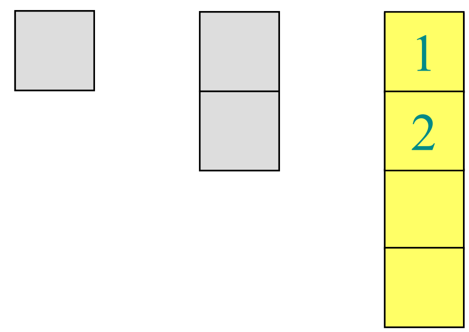
Example of an Efficient Dynamic Table

1. INSERT
2. INSERT
3. INSERT



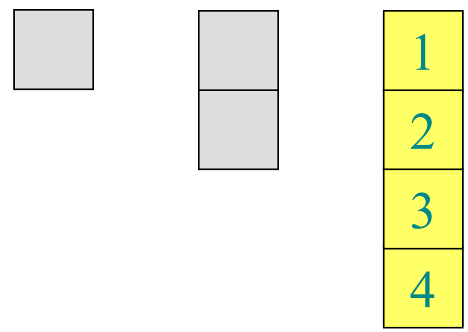
Example of an Efficient Dynamic Table

1. INSERT
2. INSERT
3. INSERT



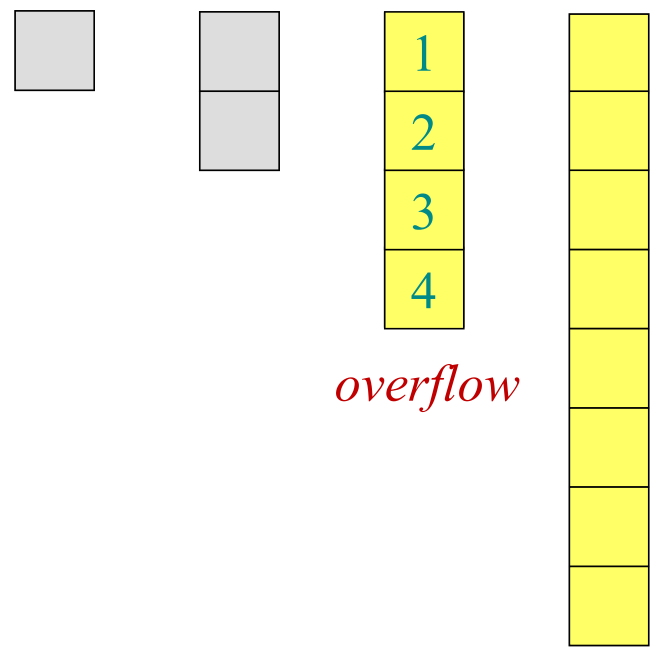
Example of an Efficient Dynamic Table

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



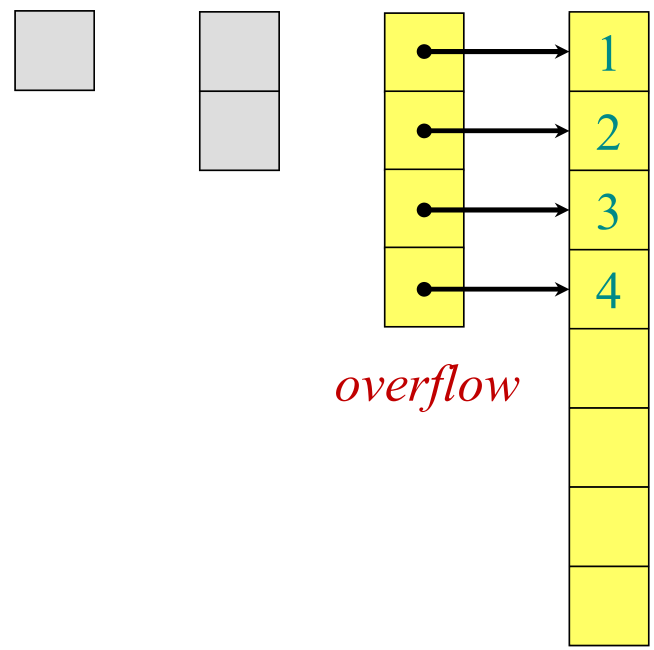
Example of an Efficient Dynamic Table

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



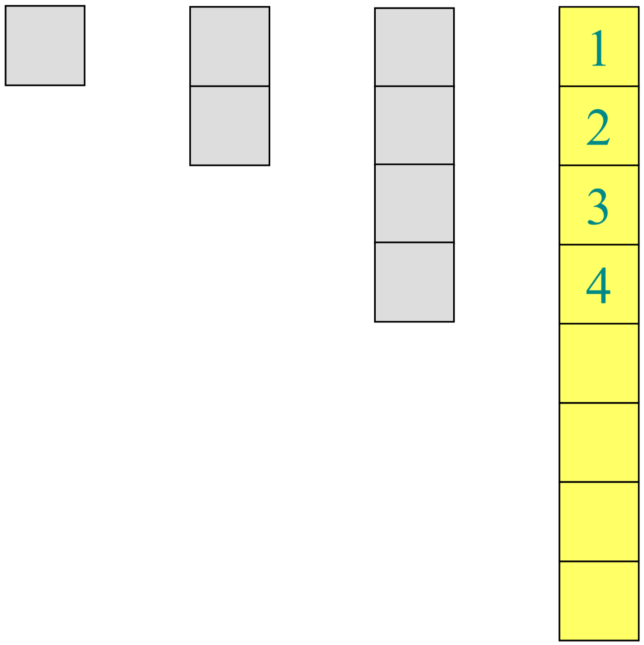
Example of an Efficient Dynamic Table

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



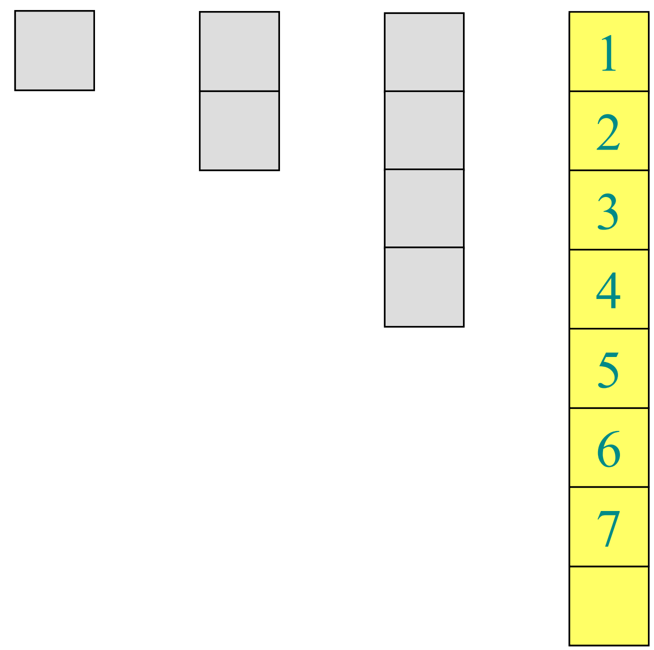
Example of an Efficient Dynamic Table

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



Example of an Efficient Dynamic Table

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 $O(n)$.

- So, the worst-case time for n insertions is $n \cdot O(n) = O(n^2)$.

Is this tight?

Amortized Analysis

- Observe, once the table is full, we create a table of double the size.
- It will take **O(1)** time for next many insertions (filling up empty slots) until overflow happens.
- So the heavy operation (copying the table into new table) will occur only when $\boldsymbol{n-1}$ is a power of $\mathbf{2}$.

Aggregate Method

Let $t(i) =$ the cost of the i th 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**
$t(i)$	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
		1	**2**		**4**				**8**	

Cost for i th insert

Cost for copying due to overflow

Aggregate Method

$$= 2^{\lfloor \log(n-1) \rfloor + 1} - 1$$
$$\leq 2(n-1)$$

(Geometric series)

Cost of n insertions $= \sum_{i=1}^{n} t(i)$

$$\leq \text{n} + \boxed{\sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j}$$

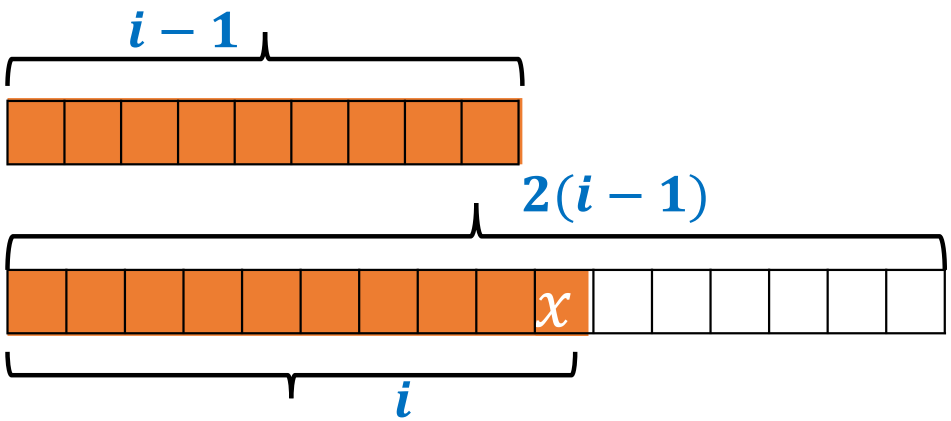
$$\leq 3n$$

Thus, the average cost of each insertion in dynamic table is $=$ O(n)/n $=$ O(1).

Accounting Method

- Charge \$3 for each insertion
  - \$1 for the insertion into the current table
  - \$1 for moving when the table expands
  - \$1 for moving another item (which was moved before) when the table expands
- Suppose the table has size m after an expansion, so it currently holds $m/2$ elements. Consider the next insertion.
  - \$1 to pay for the insertion itself
  - \$1 stored for moving this item when the table expands
  - \$1 stored for moving one of the existing $m/2$ items when the table expands

Potential Method for Insert($\boldsymbol{x}$)



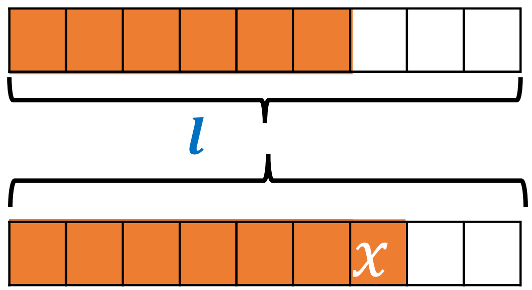
$\phi(i\text{-}1) = i - 1$ Before **Insert**($\boldsymbol{x}$)

$\phi(i) = 2$ After **Insert**($\boldsymbol{x}$)

Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1		
Case 2: when table is already full	$\boldsymbol{i}$	$\mathbf{3} - \boldsymbol{i}$	$\mathbf{3}$

$$\phi(\boldsymbol{i}) = \mathbf{2}\boldsymbol{i} - \boldsymbol{size}(\boldsymbol{T})$$

Potential Method for Insert(x)



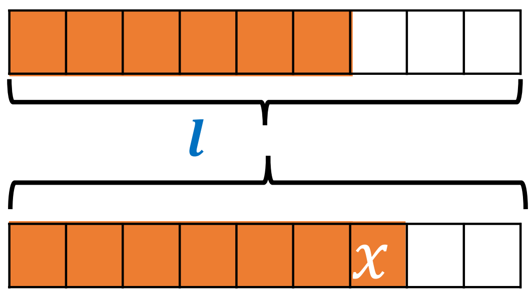
$\phi(i\text{-}1) = 2(i-1) - l$ Before **Insert**(x)

$\phi(i) = 2i - l$ After **Insert**(x)

Operation Insert(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1	$\mathbf{2}$	$\mathbf{3}$
Case 2: when table is already full	$\boldsymbol{i}$	$\mathbf{3 - \boldsymbol{i}}$	$\mathbf{3}$

$$\phi(i) = 2i - size(T)$$

Potential Method for Insert(x)



$\phi(i\text{-}1) = 2(i-1) - l$ — Before **Insert**(x)

$\phi(i) = 2i - l$ — After **Insert**(x)

Operation Insert(x)	**Actual Cost**	$\Delta\phi_i$	**Amortized Cost**
Case 1: when table is not full	1	**2**	**3**
Case 2: when table is already full	$\boldsymbol{i}$	$\boldsymbol{3-i}$	**3**

Amortized cost of n insertions $= 3n = O(n)$

Actual cost of n insertions $= O(n)$

Conclusion

- Amortized costs can provide a clean abstraction of data-structure performance.
- Amortized analysis can be performed using all 3 methods: aggregate method, accounting method, potential method.
  - But each method has some situations where it is arguably the simplest or most precise.
  - Choice of potential function can be somewhat tricky, and you sometimes need to play around with different options.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.

Acknowledgement

- The slides are modified from
  - The slides from Prof. Surender Baswana
  - The slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - The slides from Prof. Diptarka Chakraborty
  - The slides from Prof. Arnab Bhattacharyya