Analysis and Design of Algorithms

CS3230

Week 7

Amortized Analysis

Diptarka Chakraborty

Ken Sung

Announcement on Midterm

- Midterm is scheduled on 7th March (Saturday) 11:00-13:00
- Please reach the exam venue 15mins in advance
- Don’t forget to upload your temperature before coming for the exam
- Best of luck for your midterm

Details of the homeworks (change of plan)

- HW1 (24 Jan – 7 Feb)
  - Recurrence and Master theorem
- HW2 (7 Mar – 21 Mar)
  - Amortization
- HW3 (26 Mar – 10 Apr)
  - NP hardness

- Prog1 (14 Feb – 6 Mar)
  - Divide-and-conquer or linear sorting
- Prog2 (20 Mar – 10 Apr)
  - DP or Greedy

Syllabus (Change of Plan)

Week No.	Date	Topics	Lecturer
1	16-Jan-20	Reasoning and asymptotic analysis (analysis)	Ken
2	23-Jan-20	Recurrences and Master Theorem (analysis)	Ken
3	30-Jan-20	Divide-and-Conquer Algorithm (design)	Ken
4	6-Feb-20	Sorting Lower Bounds (analysis), Sorting in Linear Time (design)	Ken
5	13-Feb-20	Probabilistic Analysis, Randomized Algorithms --- Hashing and Quicksort (analysis)	Ken
6	20-Feb-20	Searching, Median Find, Order Statistics (i.e. Select) (design)	Ken
	27-Feb-20	Recess week	
7	5-Mar-20	Amortized Analysis (analysis) + mid-term test (7 Mar: 11am-1pm)	Diptarka
8	12-Mar-20	Dynamic Programming (design)	Diptarka
9	19-Mar-20	Greedy Algorithms (design)	Diptarka
10	26-Mar-20	Problem reductions and NP-completeness (analysis)	Diptarka
11	2-Apr-20	More NP-Completeness (analysis) + Approximation Algorithms (design)	Diptarka
12	9-Apr-20	Approximation Algorithms (continue) + Graph algorithms (design)	Diptarka
13	16-Apr-20	Summary and Revision	Ken/Diptarka
	18-Apr-20	Examination	

Why do we need “Amortized Analysis”?

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

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

May be grossly wrong!

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

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]	*Total Cost*
0	0	0	0	0	0	*0*
1	0	0	0	0	1	*1*
2	0	0	0	1	0	*3*
3	0	0	0	1	1	*4*
4	0	0	1	0	0	*7*
5	0	0	1	0	1	*8*
6	0	0	1	1	0	*10*
7	0	0	1	1	1	*11*
8	0	1	0	0	0	*15*
9	0	1	0	0	1	*16*
10	0	1	0	1	0	*18*
11	0	1	0	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 1:

Let $t(i) =$ no. of bit flips during ith increment

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

In the worst case, $t(i) = k$

→ $T(n) = O(n\,k)$

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

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

$f(0) = n$

$f(1) = n/2$

$f(2) = n/4$

$f(i) = n/2^i$

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

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

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 = $O(1)$

Question 1

When we say that an operation is amortized $\Theta(1)$, we mean that:

- n operations run in total $\Theta(n)$ time in the worst case.
- n operations run in total $\Theta(n)$ time in the expected case.
- n operations run in total $\Theta(n)$ time in the best case.
- n operations run in total $\Theta(1)$ in the expected case.

Solution

Answer: A

Amortized cost of $\Theta(1)$ means that the total cost divided by the number of operations is $\Theta(1)$. Hence the total cost is $\Theta(n)$.

Amortized analysis is done in the worst case.

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***!

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

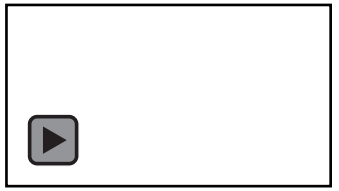
- In our last example, average cost per increment = $\frac{T(n)}{n} < 2 = O(1)$

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

Question 2

- 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 worst case running time of the two operations? (n is the size of the queue)

- 1) INSERT: O(1) time, EMPTY: O(1) time
- 2) INSERT: Θ(n) time, EMPTY: Θ(n) time
- 3) INSERT: O(1) time, EMPTY: Θ(n) time
- 4) INSERT: Θ(n) time, EMPTY: O(1) time



Solution

Answer: (3)

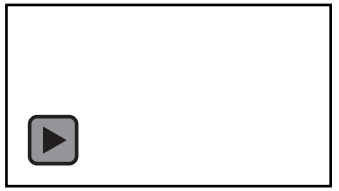
INSERT(x) takes $\Theta(1)$ time since we just insert x to the head of the queue.

EMTPY() requires us to delete all elements in the queue. It takes O(n) time.

Question 3

- 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 amortized running time of the two operations? (n is the size of the queue)

- 1) INSERT: O(1) time, EMPTY: O(1) time
- 2) INSERT: Θ(n) time, EMPTY: Θ(n) time
- 3) INSERT: O(1) time, EMPTY: Θ(n) time
- 4) INSERT: Θ(n) time, EMPTY: O(1) time



Solution

Answer: (1)

- Note that we can delete an element only if we insert it.
- If we INSERT n elements, EMPTY can delete n elements.

- The total running time of n operations is $\Theta(n)$.
- Hence, each operation takes $\Theta(1)$ amortized time.

Types of amortized analyses

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

- We have just seen an aggregate analysis.

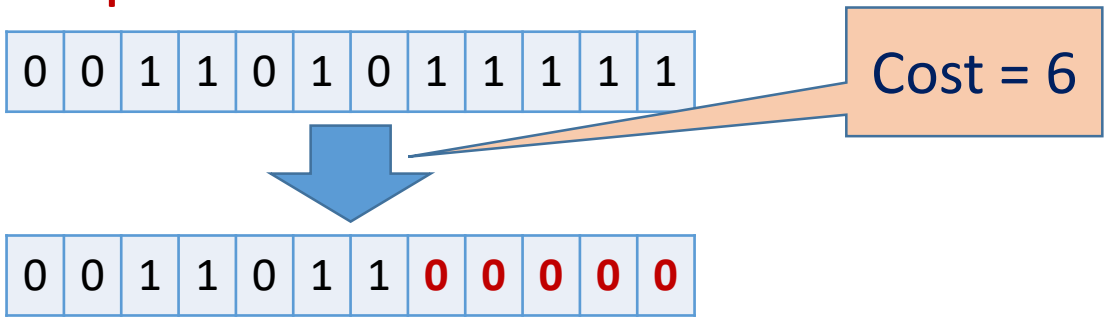
- The aggregate method, though simple, lacks the precision of the other two methods. In particular, the accounting and potential methods allow a specific ***amortized cost*** to be allocated to each operation.

Accounting (Banker's) method

- Charge *i* th operation a fictitious ***amortized cost*** *c(i)*, assuming $1 pays for 1 unit of work (time) e.g., bit-flips in the previous example.
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the ***bank*** for use by subsequent operations.
- The idea is *to impose an extra charge on inexpensive operations* and use it to pay for expensive operations later on.
- The bank balance ***must not go negative***!
- We must ensure that $\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$ for all n.
- Thus, **the total amortized costs provide an upper bound on the total true costs.**

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

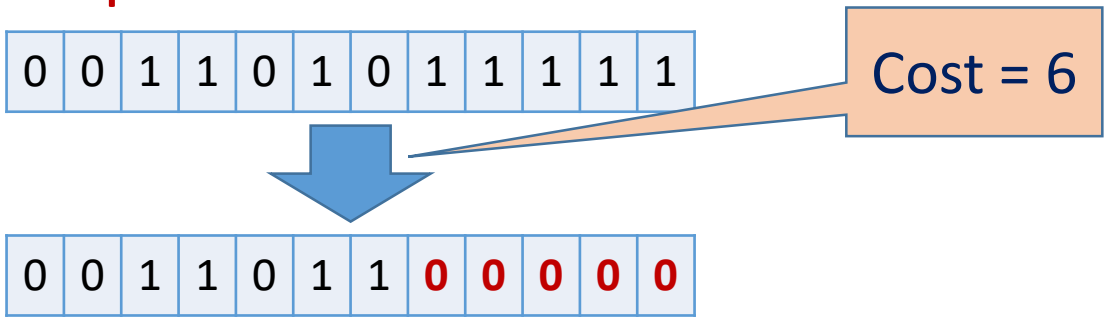
k-bit Binary Counter: how do we increment?

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	

```
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
```

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.

Why?
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 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**

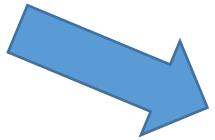
Accounting (Banker's) method

- Charge *i* th operation a fictitious ***amortized cost*** *c(i)*, assuming $1 pays for 1 unit of work (time) e.g., bit-flips in the previous example.
- This fee is consumed to perform the operation.
- Any amount not immediately consumed is stored in the ***bank*** for use by subsequent operations.
- The idea is *to impose an extra charge on inexpensive operations* and use it to pay for expensive operations later on.
- The bank balance ***must not go negative***!
- We must ensure that $\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$ for all n.
- Thus, **the total amortized costs provide an upper bound on the total true costs.**

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

Question 4

Who is the Master of Algorithms pictured below?

- Daniel Sleator
- Ron Rivest
- John Hopcroft
- Robert Tarjan

Solution

Robert Tarjan

Turing Award winner

- Linear time select (with Floyd and 3 others), least common ancestor, connected components, Fibonacci heap, splay trees.
- Many of the data structures analysed with amortized analysis.
- The term “amortized” is due to Daniel Sleator and Robert Tarjan.

Potential method

ϕ: potential function associated with the algorithm/data-structure

$\phi(i)$: Potential at the end of ith operation

Important conditions to be fulfilled by ϕ

$\phi(0) = 0$

$\phi(i) \geq 0$ for all i

$\Delta\phi_i$ = Potential difference

Amortized cost of ith operation $\stackrel{\text{def}}{=}$ Actual cost of ith operation $+ \left(\phi(i) - \phi(i-1)\right)$

Amortized cost of n operations $\stackrel{\text{def}}{=} \sum_i$ Amortized cost of ith operation

$=$ Actual cost of n operations $+ \left(\phi(n) - \phi(0)\right)$

$=$ Actual cost of n operations $+ \phi(n)$

$\geq$ Actual cost of n operations

Potential method

ϕ: potential function associated with the algorithm/data-structure

$\phi(i)$: Potential at the end of ith operation

Important conditions to be fulfilled by ϕ

$\phi(0) = 0$

$\phi(i) \geq 0$ for all i

$\Delta\phi_i$ = Potential difference

Amortized cost of ith operation $\stackrel{\text{def}}{=}$ Actual cost of ith operation $+ \left(\phi(i) - \phi(i-1)\right)$

Amortized cost of n operations $\geq$ Actual cost of n operations

If we want to show that ***actual cost*** of n operations is O(g(n)) then it suffices to show that ***amortized cost*** of n operations is O(g(n))

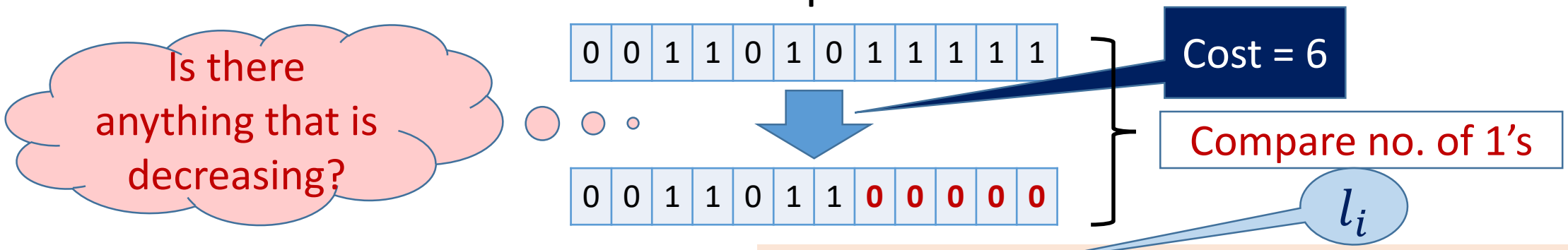
Potential method - recipe

- Try to select a suitable $\boldsymbol{\phi}$, so that for the costly operation, $\Delta\boldsymbol{\phi}_i$ is negative to such an extent that it *nullifies* or *reduces* the effect of actual cost.
- **Question**: How to find such a suitable potential function $\boldsymbol{\phi}$?

Try to view carefully the costly operation and see if there is some quantity that is "**decreasing**" during the operation.

Potential method on Binary Increment

- Take a close look on increment operation



- Actual cost of ith increment = 1 + Length of the longest suffix with all 1's

Actual cost of ith increment	$\Delta\phi_i$	Amortized cost of ith increment
$l_i + 1$	$-l_i + 1$	2

$\phi(i)$: No. of 1's in the counter after ith increment

Amortized cost of n increments $= 2n = O(n)$

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

Dynamic table

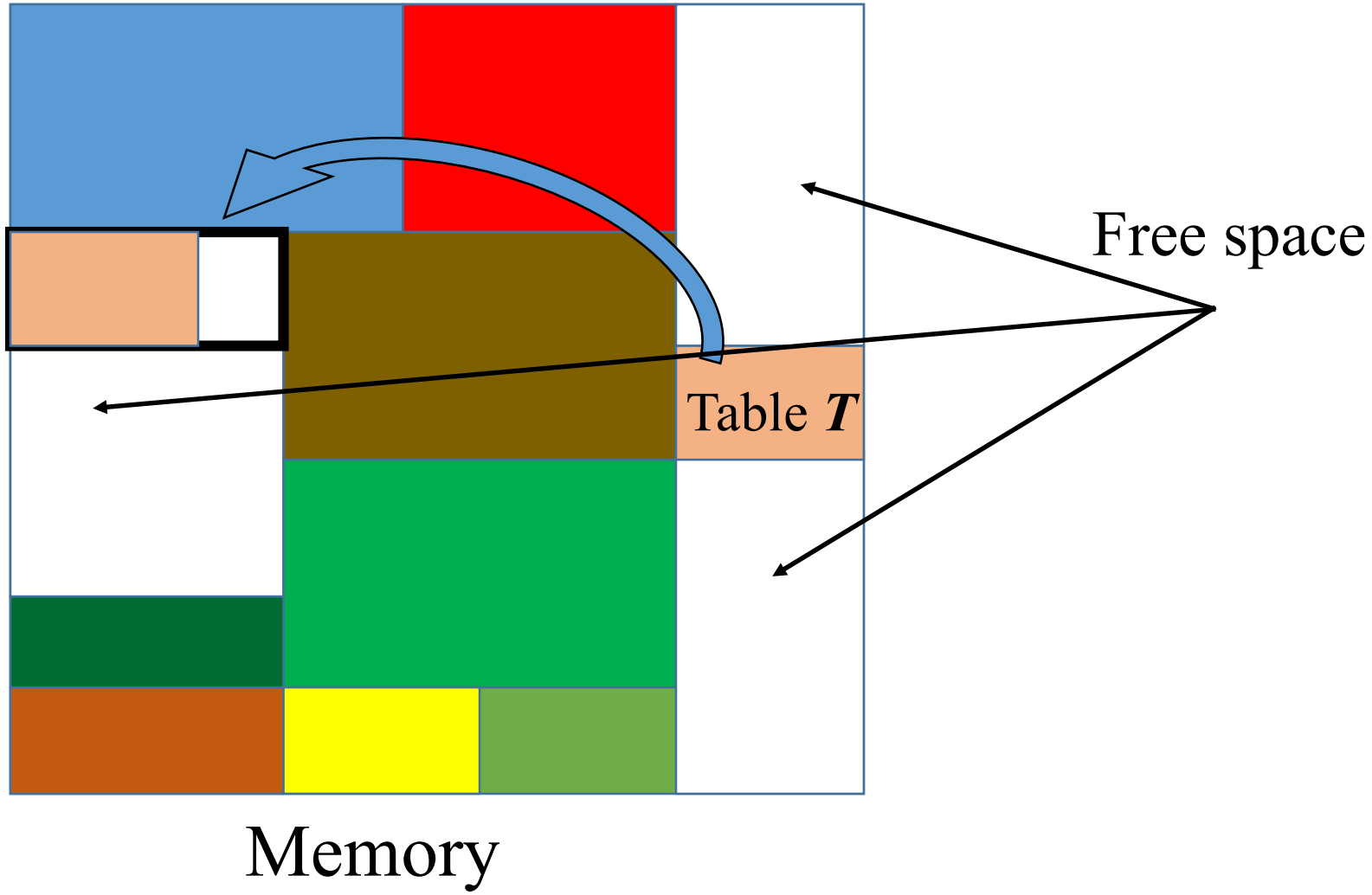
For Insertion only

How large should a table be?

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

Memory Management

What happens if more space is needed by T ?



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

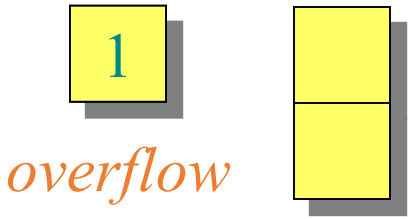
- n: number of elements in the table.
- **createTable**(k): A system-call that creates a table of size k and returns its pointer.
- **size**(T): the size of table T.
- **copy**(T,T'): copies the contents of table T into T'.
- **free**(T): free the space (return the space to OS) occupied by table T.

A 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;
```

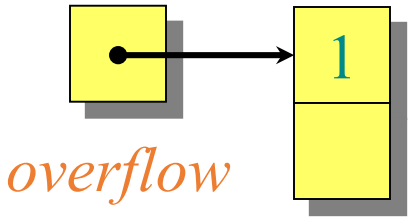
Example of a dynamic table

1. INSERT
2. INSERT



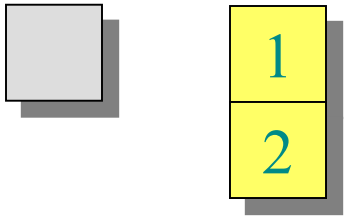
Example of a dynamic table

1. INSERT
2. INSERT



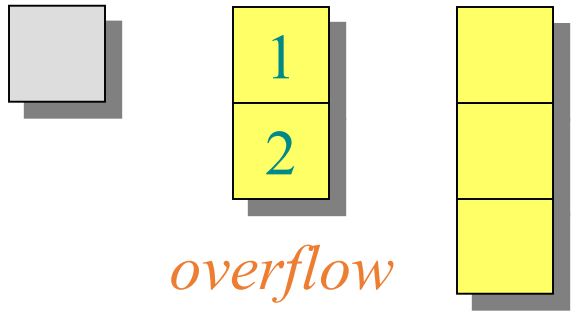
Example of a dynamic table

1. INSERT
2. INSERT



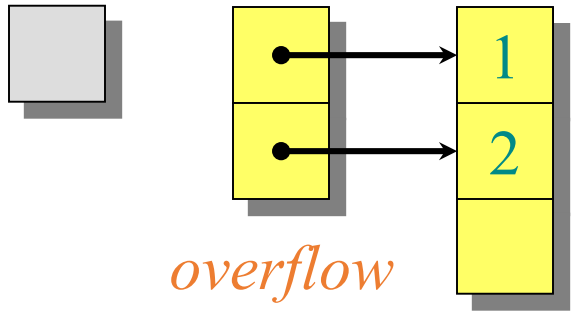
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



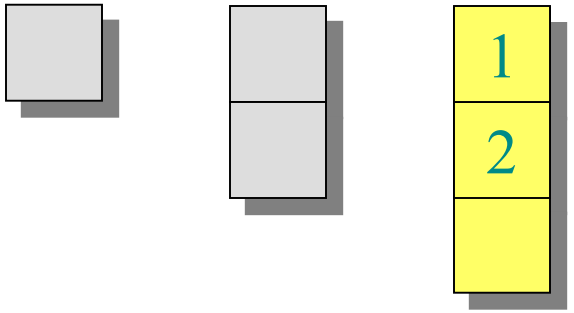
Example of a dynamic table

1. INSERT
2. INSERT
3. INSERT



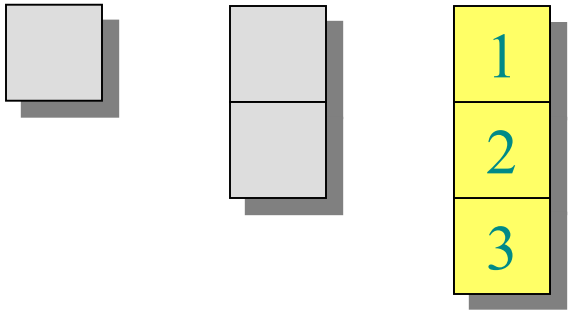
Example of a dynamic table

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;
```

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

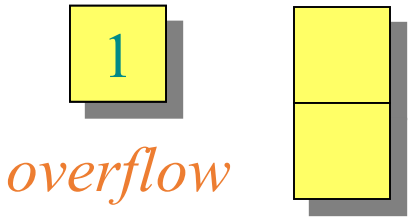
Idea: Every time table is full, create a new table of **double the size**

An efficient 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;
```

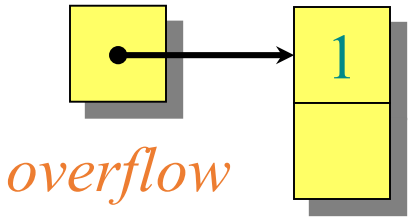
Example of an efficient dynamic table

1. INSERT
2. INSERT



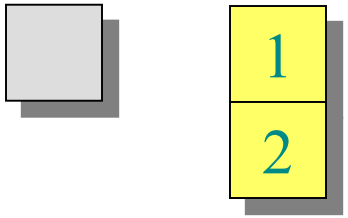
Example of an efficient dynamic table

1. INSERT
2. INSERT



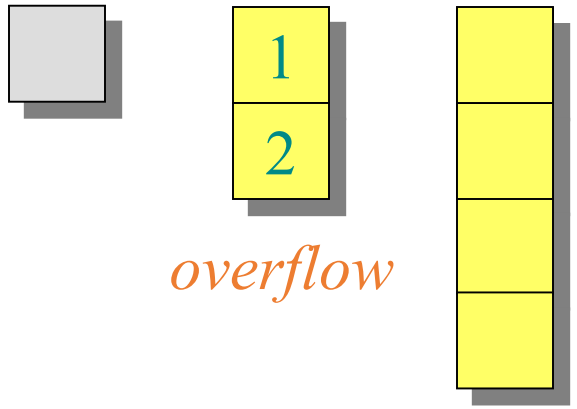
Example of an efficient dynamic table

1. INSERT
2. INSERT



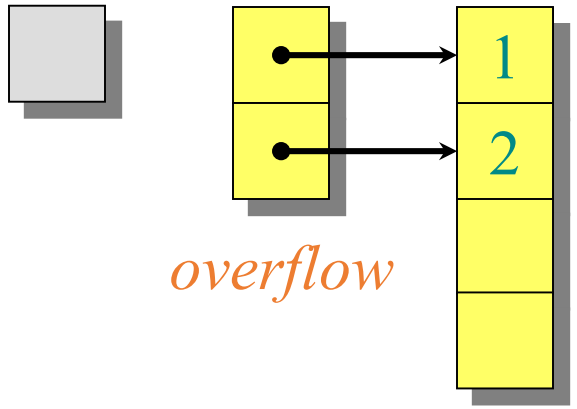
Example of an efficient dynamic table

1. INSERT
2. INSERT
3. INSERT



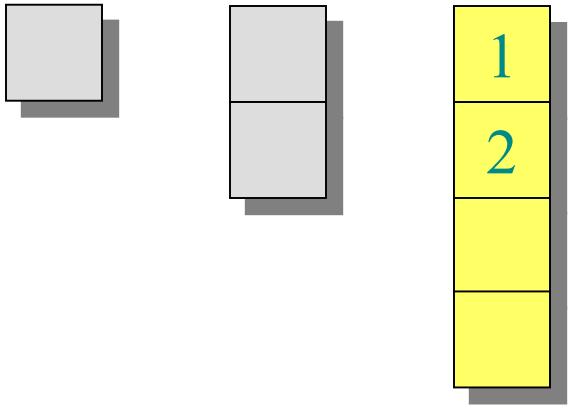
Example of an efficient dynamic table

1. INSERT
2. INSERT
3. INSERT



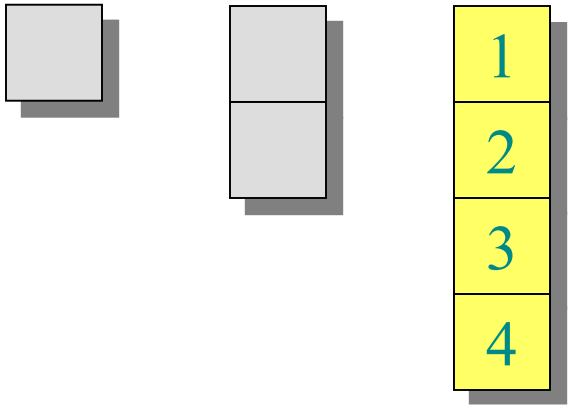
Example of an efficient dynamic table

1. INSERT
2. INSERT
3. INSERT



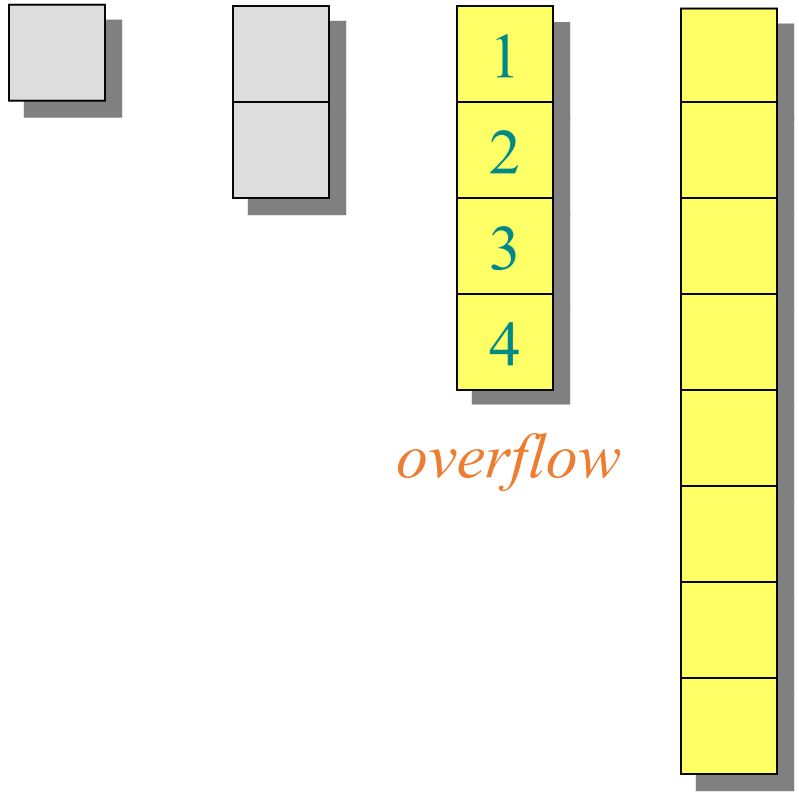
Example of an efficient dynamic table

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



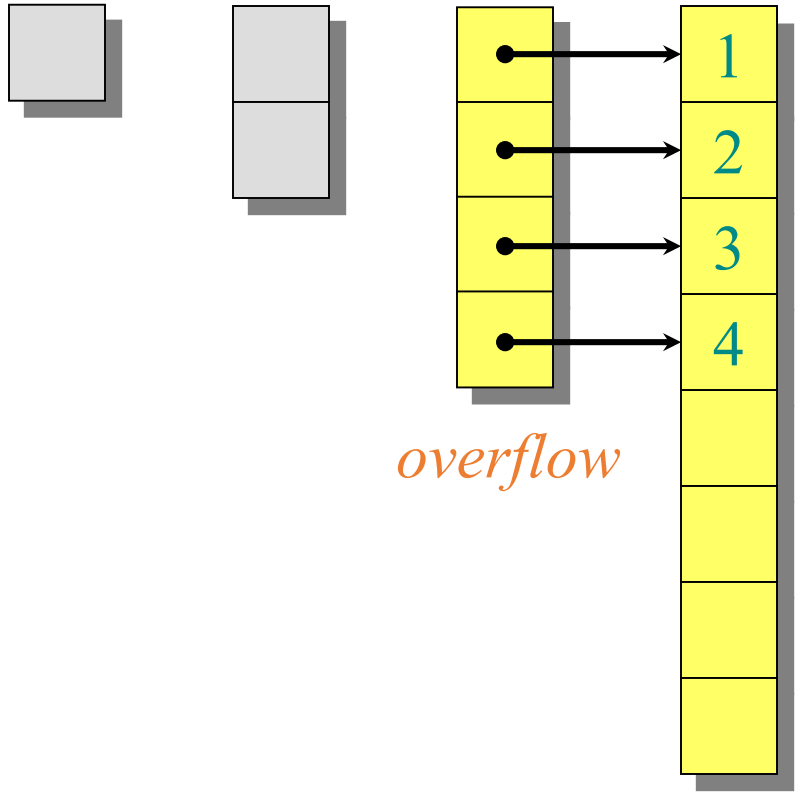
Example of an efficient dynamic table

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



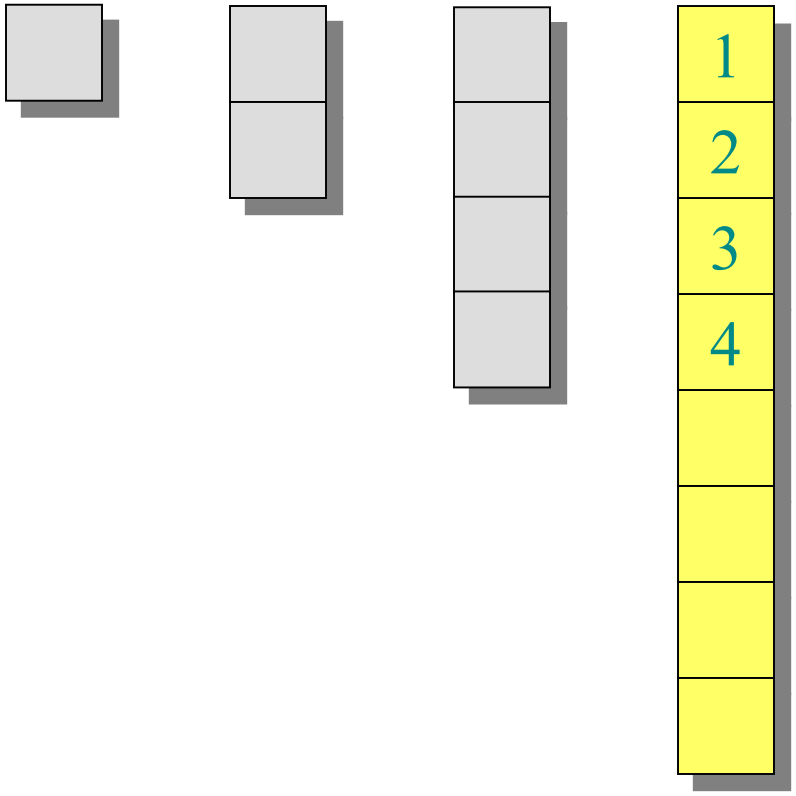
Example of an efficient dynamic table

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



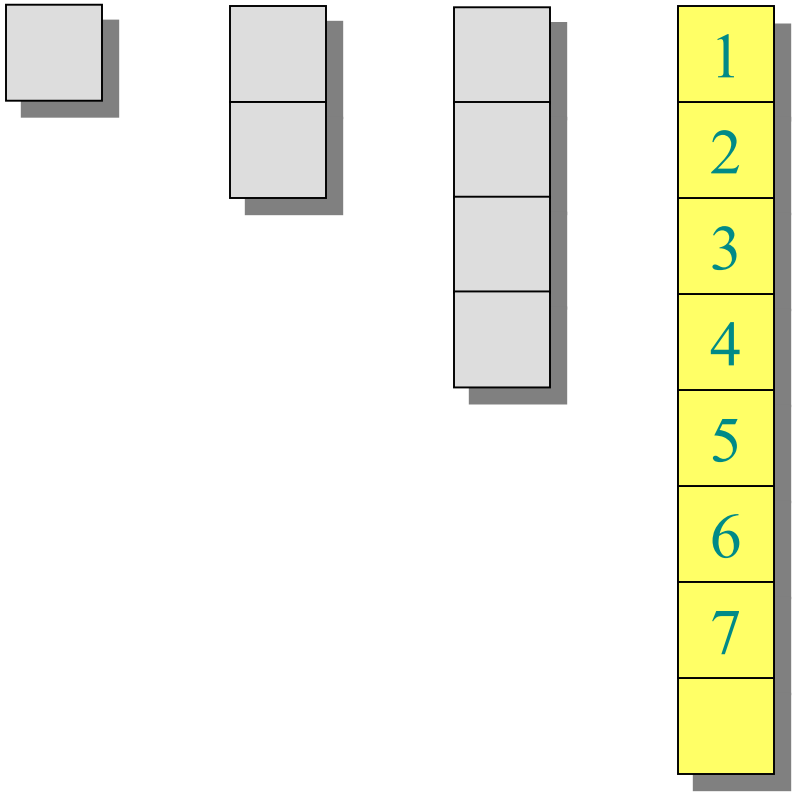
Example of an efficient dynamic table

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



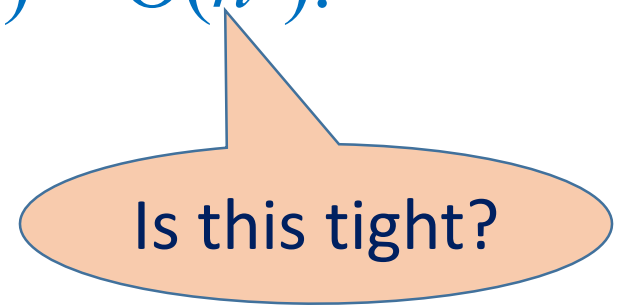
Example of an efficient dynamic table

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



Amortized analysis

- Observe, once the table is full, we create a table of double the size.
- It will take $\mathbf{O(1)}$ 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 whenever $\boldsymbol{n}$ 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

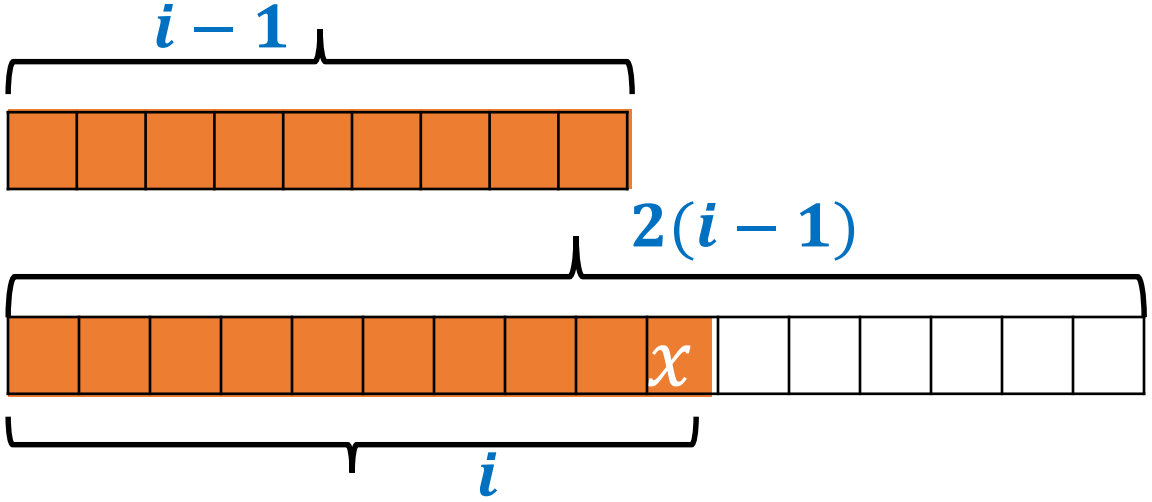
Cost of n insertions = $\sum_{i=1}^{n} t(i)$

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

$$\leq 3n$$

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

Potential method for Insert(x)



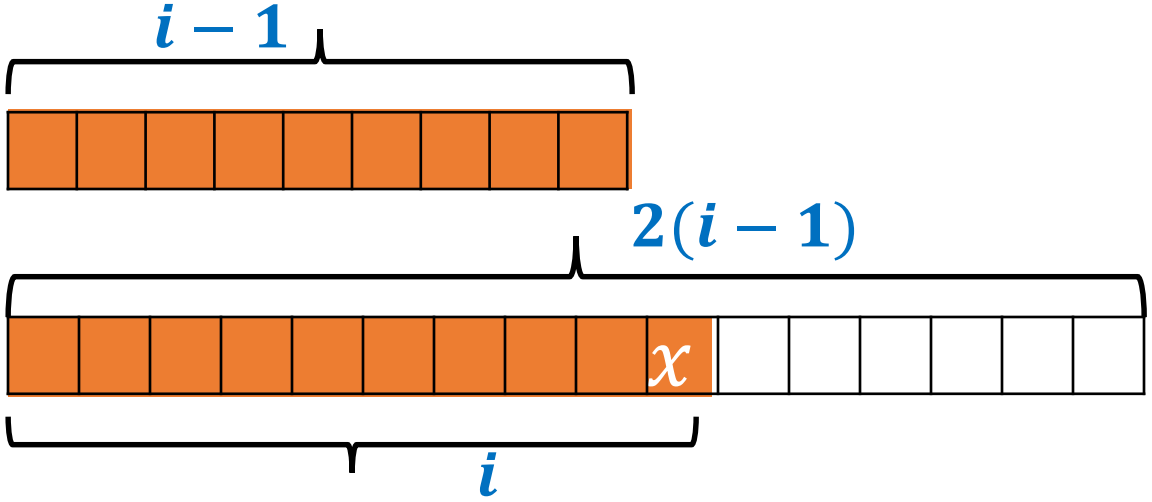
Before **Insert**(x)

After **Insert**(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	i		

Seems like everything including **size of T** has increased

Potential method for Insert(x)



Before **Insert**(x)

After **Insert**(x)

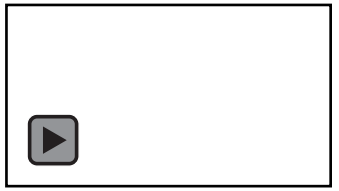
Operation **Insert**(x)	Actual Cost	$\Delta\phi_i$	Amortized Cost
Case 1: when table is not full	1		
Case 2: when table is already full	i		

What about (**– size of T)** ?

Question 5

- Does the function (- size of T) satisfy all the properties of a potential function ϕ ?

- 1) Yes
- 2) No

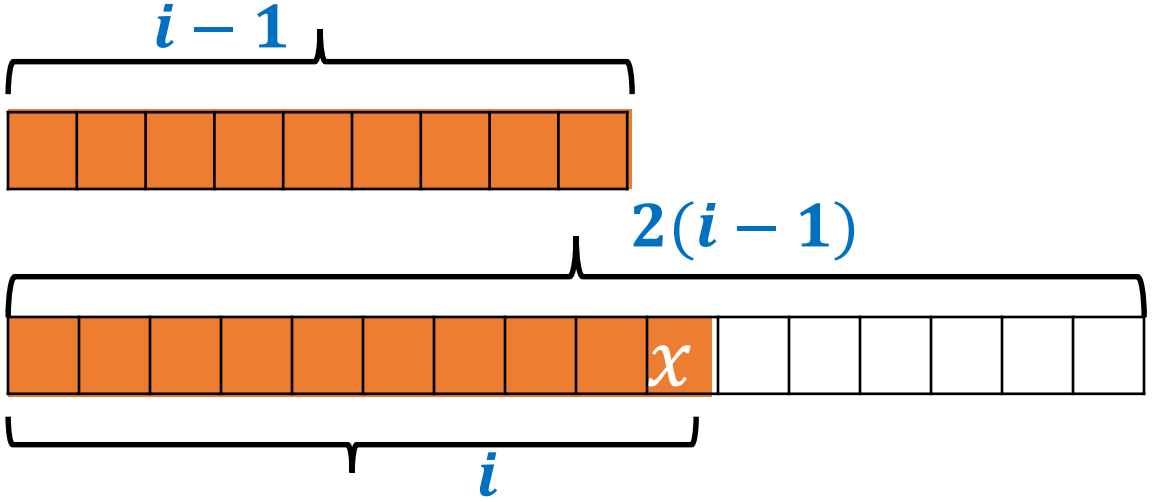


Solution

Answer: (2)

ϕ cannot be negative.

Potential method for Insert(x)



Before **Insert**(x)

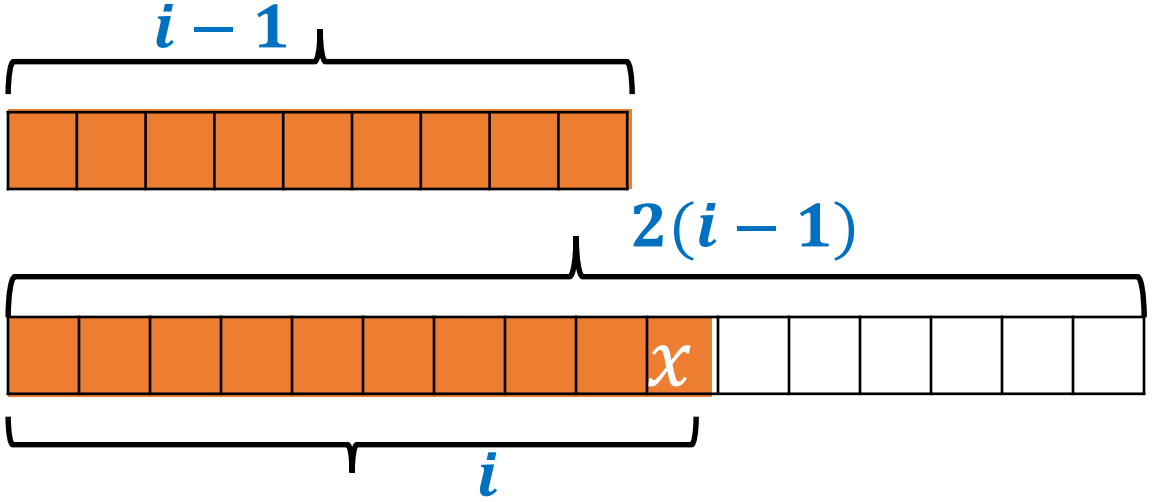
After **Insert**(x)

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	i		

What about (**– size of T**) ?

This has decreased ☺
However ϕ cannot be negative ☹

Potential method for Insert(x)



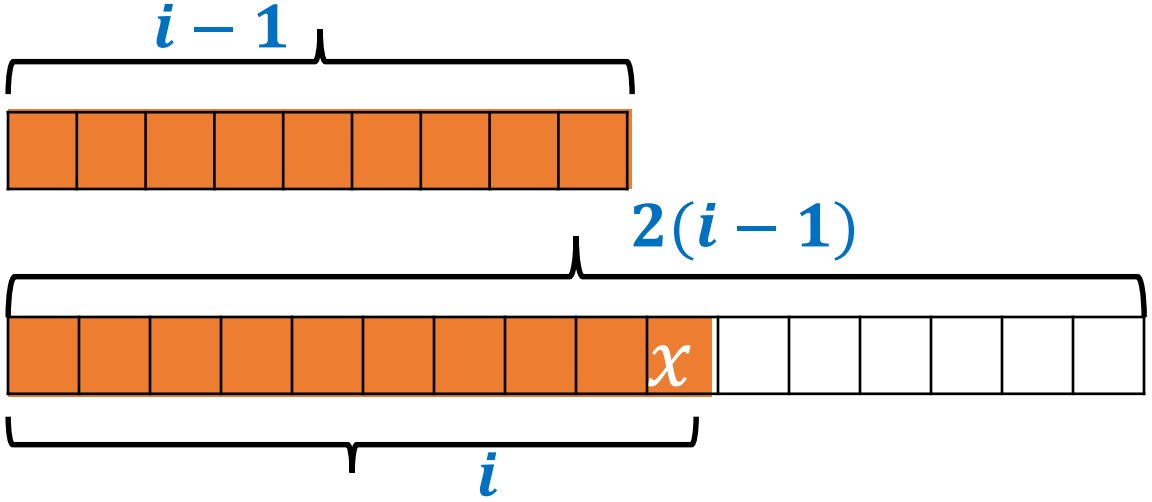
Before **Insert**(x)

After **Insert**(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	i		

May be add some quantity with (**– size of T**) to make it non-negative

Potential method for Insert(x)



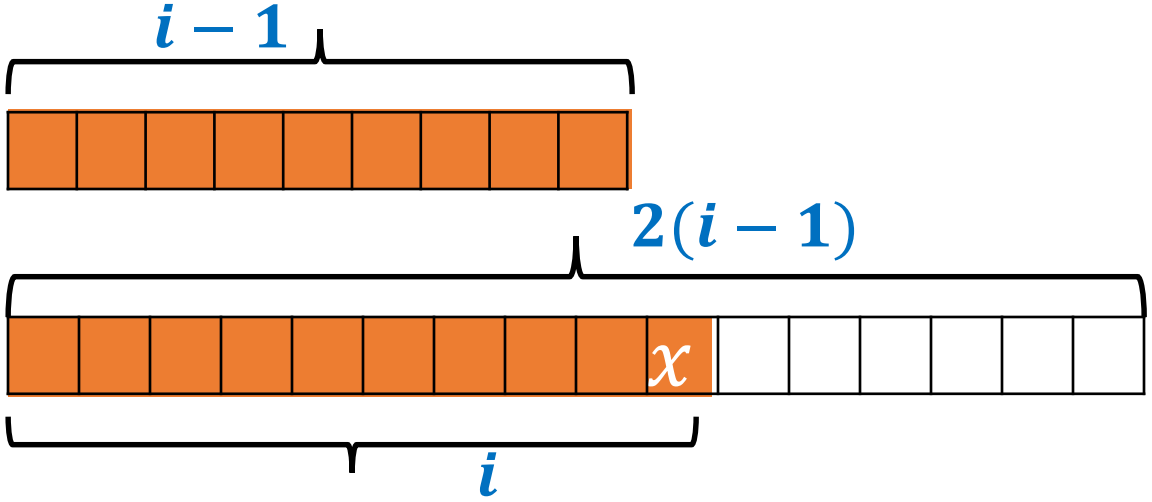
Before **Insert**(x)

After **Insert**(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	i		

Observation: Table **T** is always at least half-full

Potential method for Insert(x)



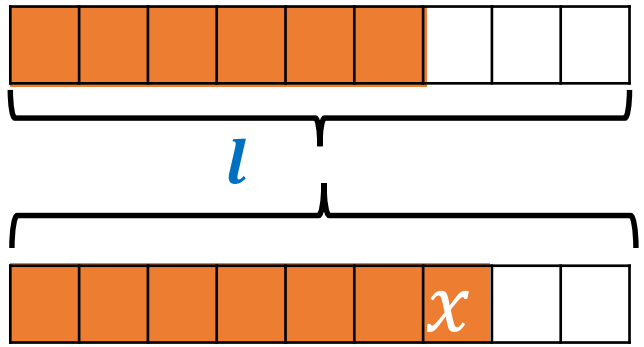
$\phi(i\text{-}1) = i - 1$ Before **Insert**(x)

$\phi(i) = 2$ After **Insert**(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	i	$3 - i$	3

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

Potential method for Insert(x)



$\phi(i\text{-}1) = 2(i-1) - l$

Before **Insert**(x)

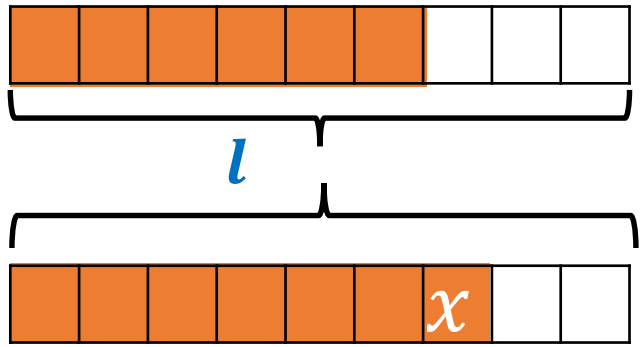
$\phi(i) = 2i - l$

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	i	$3 - i$	**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	i	$3 - i$	**3**

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

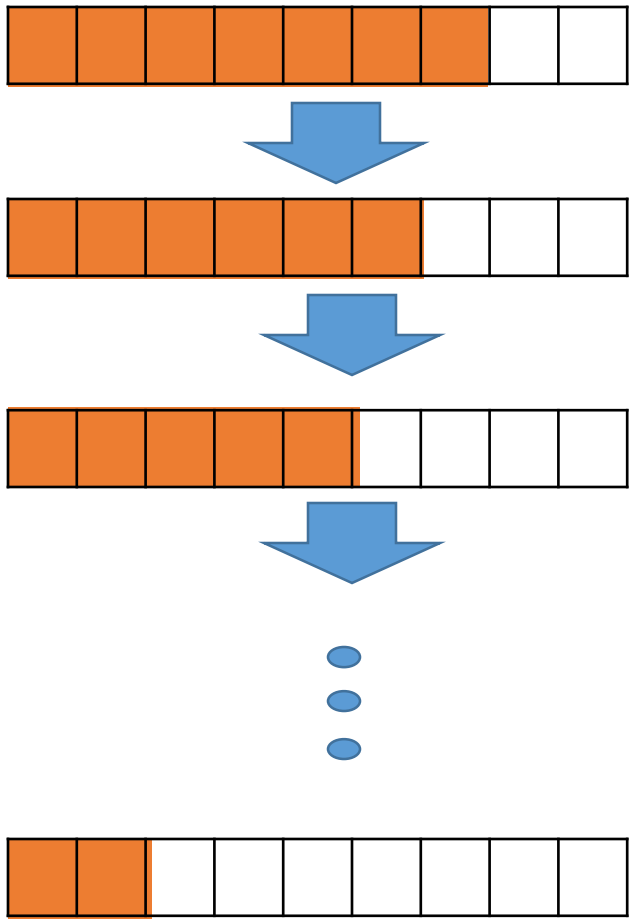
Actual cost of n insertions $= O(n)$

Accounting analysis (Exercise)

- How can you charge each non-overflow case so that it nullifies expensive overflow case?
- Suppose overflow happens in ith insertion. So the next overflow happens during 2ith insertion.
- Compare no. of insertions between these two overflow cases and no. of items needs to be copied during 2ith iteration.

Important: Don't forget to show that bank balance never falls **below 0**

Dynamic table for deletion only (Exercise)



Idea: Every time table is **half-free**, create a new table of **half the size**

Wastage of space !!

Exercise: Analyze cost of n deletions using amortized analysis (by all three methods).

Conclusions

- 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.
- 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. Arnab Bhattacharya and Prof. Wing-Kin Sung