15451 Fall 2022

Amortized Analysis

Elaine Shi

Imagine some data structure

- Each operation takes **non-uniform** runtime
- An algorithm may make many calls to the data structure
- What matters: **average** cost per operation, also called **amortized** cost

Example: growing an array

- On item arrival, store in an array
- No prior knowledge of #items

What space should we preserve for the array?

- Allocate O(1) upfront
- Whenever full, double the size

What is the amortized cost?

std: vector

- constructor: vector<int> array
- push_back ← add new item
- pop_back ← deletion
- index into: array[i]

- Allocate O(1) upfront
- Whenever full, double the size
- Whenever ¼ loaded, half the size

This Lecture

- Learn how to do amortized analysis
- Design algorithms with good amortized runtime

Amortized algorithm design and analysis are useful in many applications!

Growing an array

- initialize(): allocates an empty table of size 1 ($n = 1$, $s = 0$)
- insert(): add a new element to the table (s++)
  - if $s = n$ then grow(),
  - add the new elem to array[s] (costs 1)
- grow(): double the size from n to $2n$, costs $2n$

Suppose at the end, there are m elements in the array.

What is the amortized cost of such an array?

Let N be the space allocated at the end of the day

$1 \to 2 \to 4 \to 8 \cdots \to \frac{N}{2} \to N$

$2 + 4 + 8 + \cdots + N \le 2N \le 4m$

amortized cost $= 4 + 1 = 5$

$\frac{N}{m} \le 2$

space reserved; actual # items

$N \ge m$, $N \le 2m$

Another example: Binary Counter

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

- Suppose each bit flipped costs 1
- Amortized cost of the binary counter?

$T = 2^n$

$1 + \frac{1}{2} + \frac{1}{4} \cdots \le 2$

$T + \frac{T}{2} + \frac{T}{4} + \cdots \frac{T}{2^{n-1}} \le 2T$

$\frac{2T}{T} = 2$

The Potential Method

- Another method of counting
- Sometimes makes analysis easier
  - e.g., for more complex algorithms

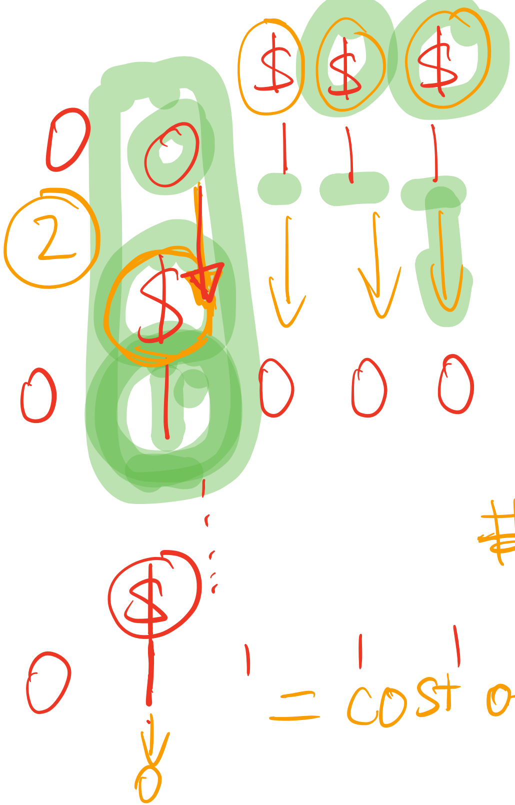
The Potential Method

Banker's view

- initially bank is empty
- every step:
  - put coins into bank
  - pay for the work using (part of) the coins in the bank

How many coins should we deposit per step, s.t. we never run out of coins?

Banker's view: Binary Counter



total coins in bank $\nearrow \phi_i$

3 $\quad \phi_i$

1 $\quad \phi_{i+1}$

coins you need to put in in time step i

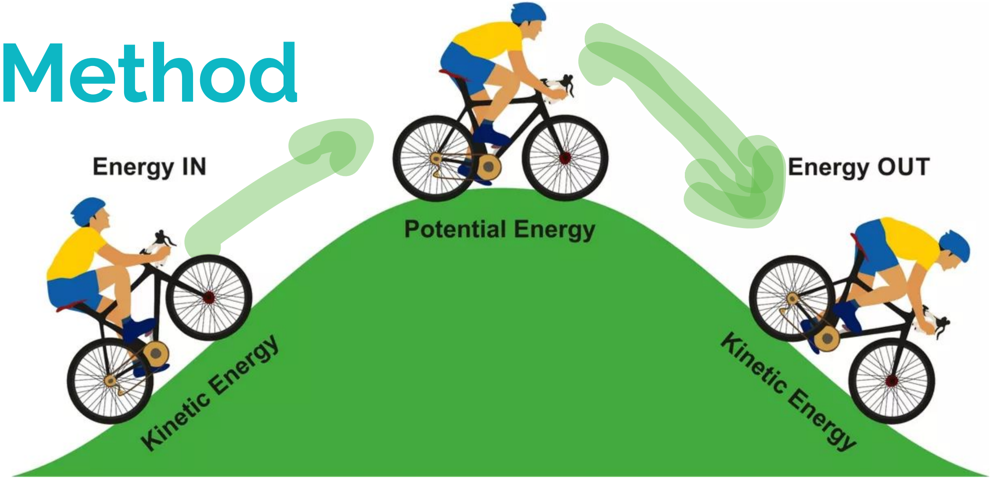
$= \text{cost of step } i + \phi_{i+1} - \phi_i$

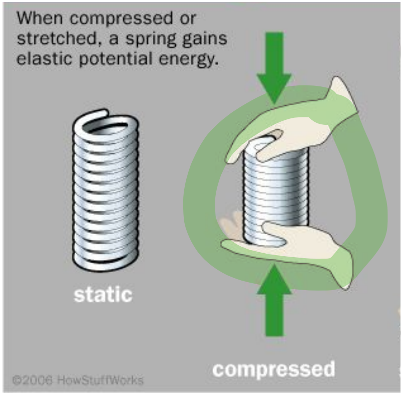
Banker's view: Binary Counter

- bank: 1 coin on each 1 bit
- every 0 ⇒ 1: deposit 2 coins
- every 1 ⇒ 0: use the coins on 1s to pay

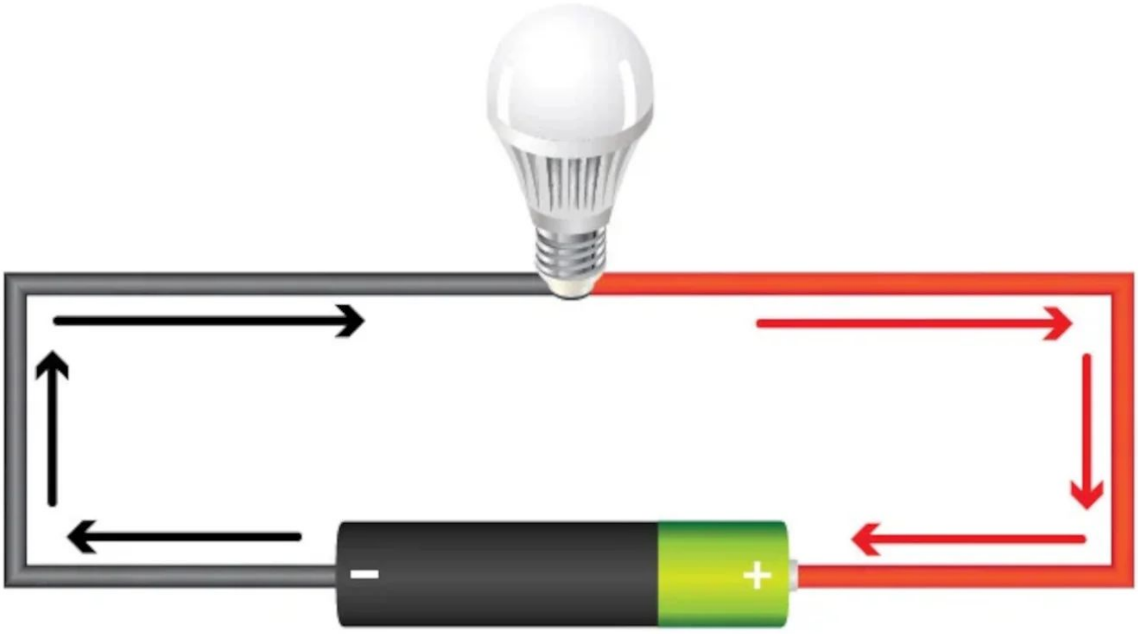
The Potential Method

Physist's view





Electric potential = voltage



The Potential Method

Physist's view

- Need to pay to build up potential
- Whenever the algorithm incurs some cost, we can pay for it using the potential

How much should we pay per step, s.t. there is always enough potential to pay for the algorithm's cost?

amortized cost of step i

Cost of the ith operation

Potential at the end of time step i

$$ac_i = c_i + \phi_i - \phi_{i-1}$$

(amortized cost) = (actual cost) + (change in potential).

$$\phi_1 - \phi_0 + \phi_2 - \phi_1 + \phi_3 - \phi_2 + \phi_4 - \phi_3 - \cdots - \phi_n$$

$$ac_i = c_i + \phi_i - \phi_{i-1}$$

(amortized cost) = (actual cost) + (change in potential).

Summing both sides

$$\sum_i ac_i = \sum_i (c_i + \phi_i - \phi_{i-1}) = \phi_n - \phi_0 + \sum_i c_i$$

$$ac_i = c_i + \phi_i - \phi_{i-1}$$

(amortized cost) = (actual cost) + (change in potential).

Summing both sides

$$\sum_i ac_i = \sum_i (c_i + \phi_i - \phi_{i-1}) = \phi_n - \phi_0 + \sum_i c_i$$

$$\sum_i c_i = \sum_i ac_i + \phi_0 - \phi_n$$

Potential analysis: growing a table

space reserved → n, #items → s

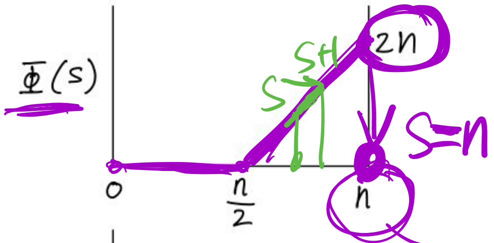
$$\Phi(n, s) = \begin{cases} 0 & \text{if } s \le \frac{n}{2} \\ 4(s - \frac{n}{2}) & \text{otherwise} \end{cases}$$

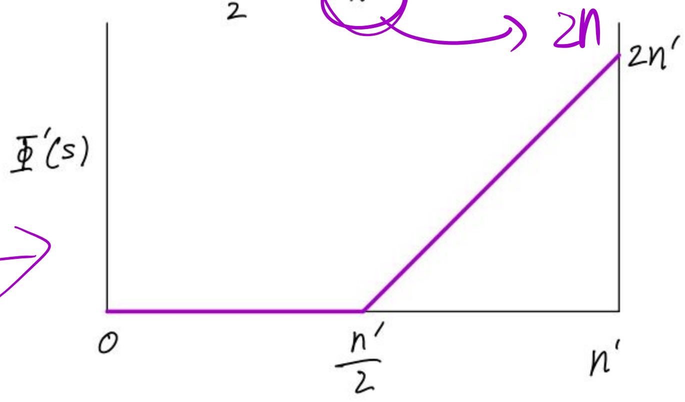
whenever I don't need to grow:

ac = 1 + 4 = 5

whenever I need to grow:

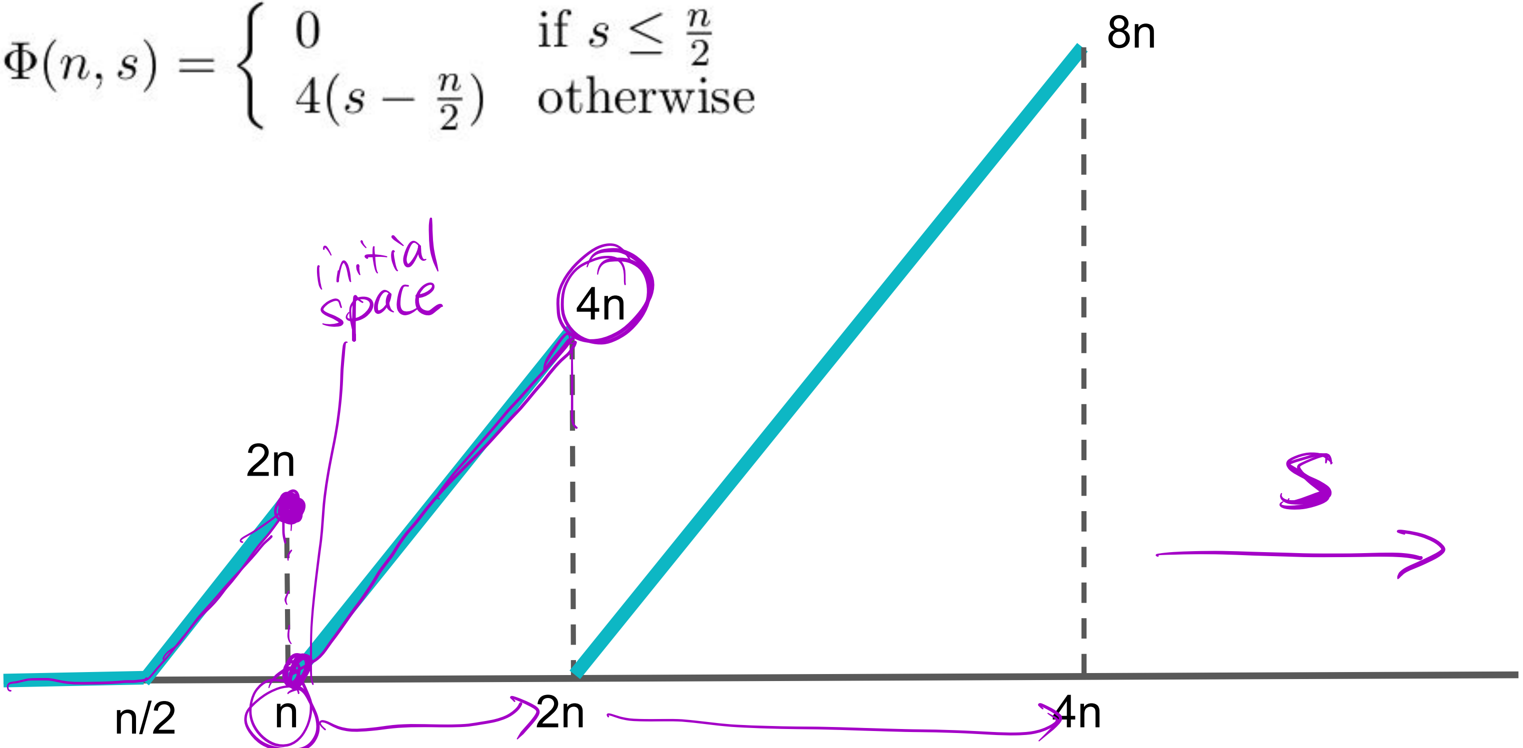
2n+1 −2n = 1





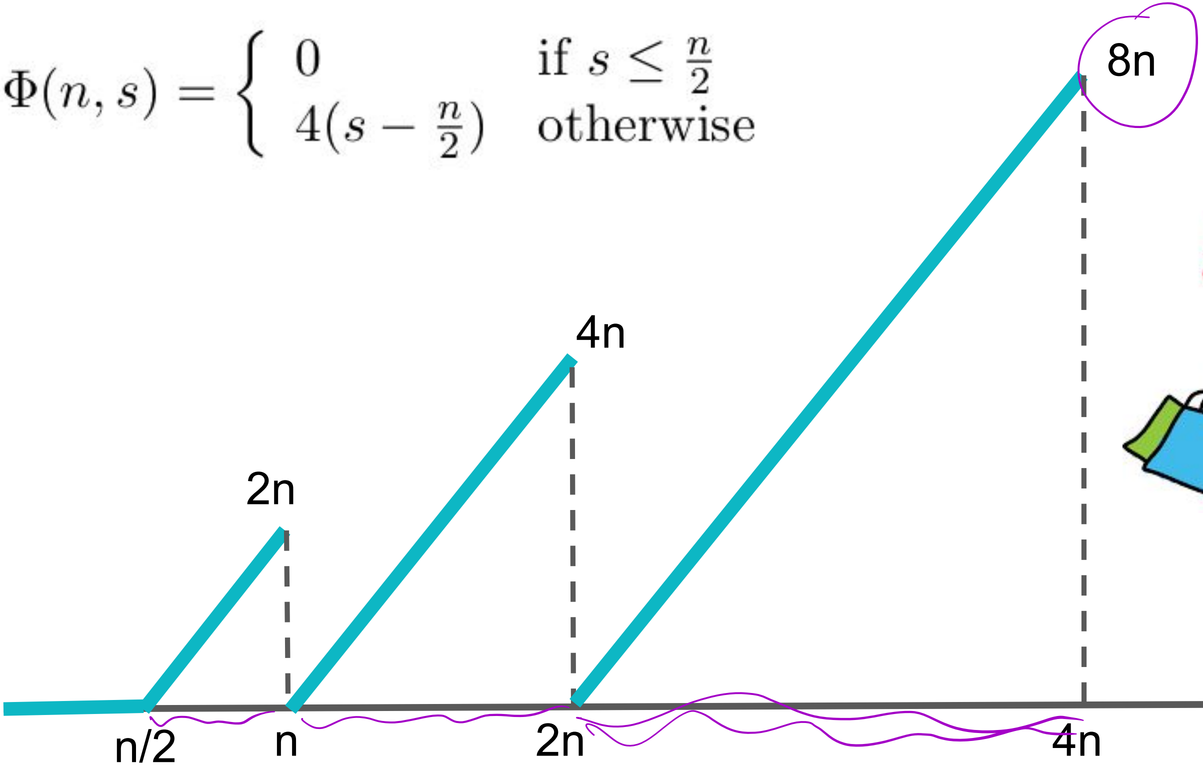
Potential analysis: growing a table

$$\Phi(n,s) = \begin{cases} 0 & \text{if } s \leq \frac{n}{2} \\ 4(s - \frac{n}{2}) & \text{otherwise} \end{cases}$$



Potential analysis: growing a table

$$\Phi(n, s) = \begin{cases} 0 & \text{if } s \leq \frac{n}{2} \\ 4(s - \frac{n}{2}) & \text{otherwise} \end{cases}$$



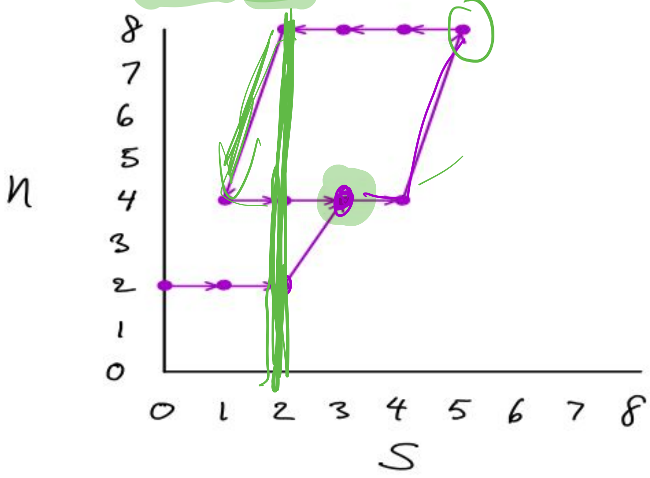
Potential analysis: growing and shrinking a table

- initialize(): allocates an empty table of size 2 (n = 2, s = 0)
- insert(): add a new element to the table (s++)
  - if s = n then grow(),
  - add the new elem to array[s] (costs 1)
- delete(): delete the last elem from table (s--)
  - if s = n/4 and n>=4 then shrink()
  - delete last elem (costs 1)
- grow(): double the size from n to 2n, costs 2n
- shrink(): change the size of table to n/2. Costs n.

$n \to n/2$

Value of n depends not just on s, but also the history

$(0,2),(1,2),(2,2),(3,4),(4,4),(5,8),(4,8),(3,8),(2,8),(1,4),(2,4),(3,4)$



$$\Phi(n, s) := 4\left|s - \frac{n}{2}\right|$$

$n = 2$

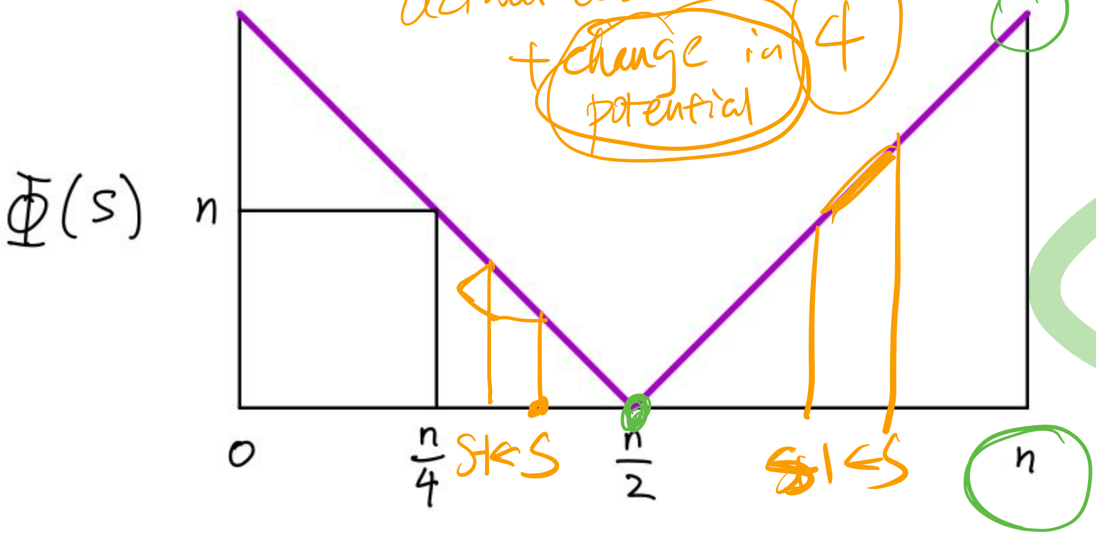
$s = 0$
$n = 2$

$\phi(n, s) = 4$

$\phi_0 = 4$

actual cost + change in potential

4



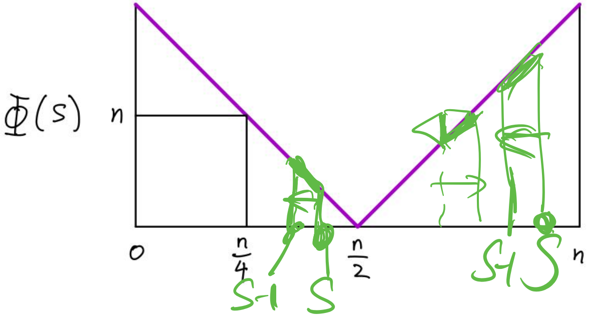
Theorem: total cost of N insertions and deletions is at most 5N + 4.

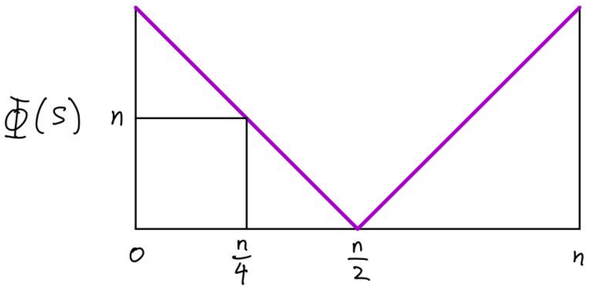
Insertion: 1 + 4 = 5

(1: actual cost; 4: change in potential)

$\phi_0 = 4$

Deletion: 1 + 4 = 5




$\Phi(s)$
n
0
$\frac{n}{4}$
$\frac{n}{2}$
n

The dictionary data structure

dict

- insert(key, val)
- search(key)

Next lecture: splay tree

Today: a hierarchical data structure that is almost as good as splay tree.

- A sorted array can be searched in **O(log n)** time
- Unfortunately insertion is slow

static data structure : search ( )

dynamic + insert ( )

```
A0:   [5]
A1:   [4,8]
A2:   empty
A3:   [2, 6, 9, 12, 13, 16, 20, 25]
```

- Search: binary search in each level

$O(\log n \cdot \log n) = O(\log^2 n)$

A0: [5]

A1: [4, 8]

A2: empty

A3: [2, 6, 9, 12, 13, 16, 20, 25]

- Now we want to insert 10

After inserting 10

```
A0:  empty
A1:  empty
A2:  [4, 5, 8, 10]
A3:  [2, 6, 9, 12, 13, 16, 20, 25]
```

What's the amortized cost of insertion?

This is a paradigm for compiling any static data structure into a dynamic one

JOURNAL OF ALGORITHMS 1, 301–358 (1980)

Decomposable Searching Problems
I. Static-to-Dynamic Transformation*

JON LOUIS BENTLEY† AND JAMES B. SAXE

Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213

Received October 29, 1979; revised April 15, 1980