

# **Applied Algorithms**

## **CSCI-B505 / INFO-I500**

### **Lecture 7.**

### **Amortized Analysis - 2**

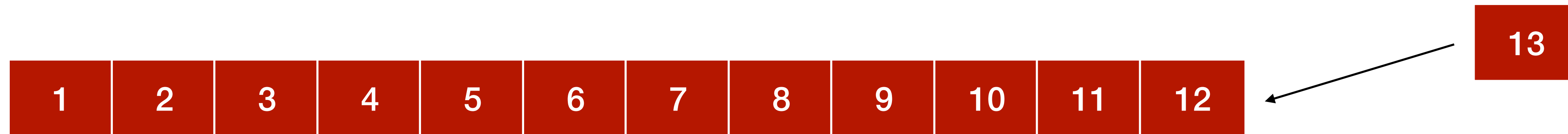
- Dynamic Array Allocation
- Amortized Dictionary Data Structure

# Dynamic Arrays

- Remember array is a contiguous block in memory.
- Thus, its size should be definite at the time of creation.
- However, the size of an array can change frequently !
- Therefore, the **dynamic arrays**, whose size can be altered at run time is of our interest. *Notice that many modern programming languages make use of it.*
- **At the end we will see that, theoretically it is no different then static arrays !?**

# Dynamic Array Allocation

Main Question: What to do when array is full, in other words, all cells are occupied, and we need to add a new element to this full array.



Allocate a larger size in the memory



Move **old** elements to the new array



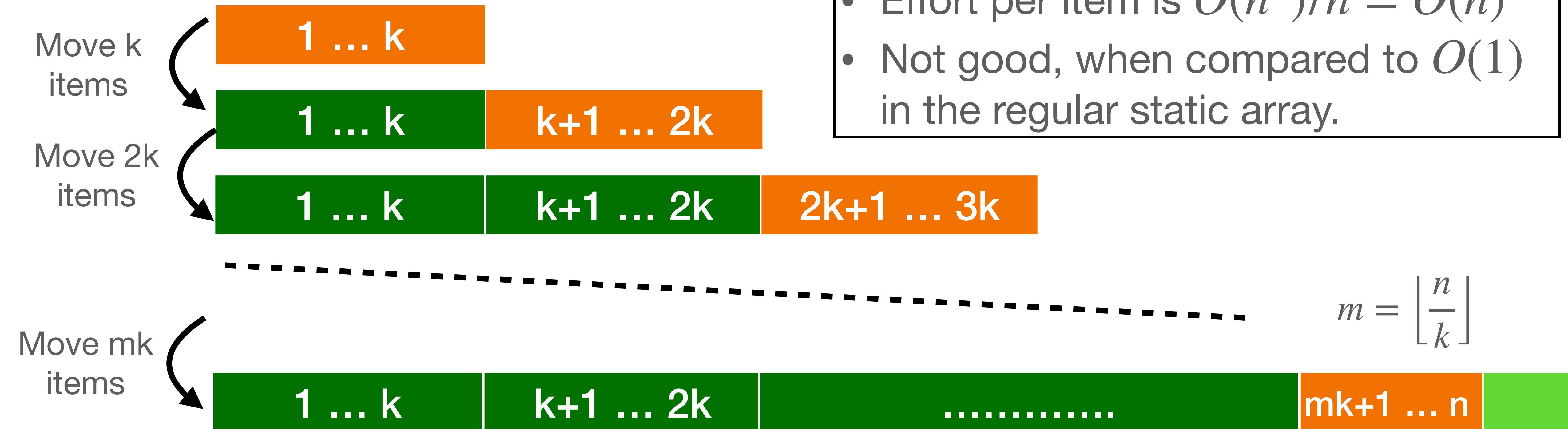
Append the **new** item to the new array



# Dynamic Array Allocation

Let's consider incrementing the size by a fixed amount each time we need to resize.

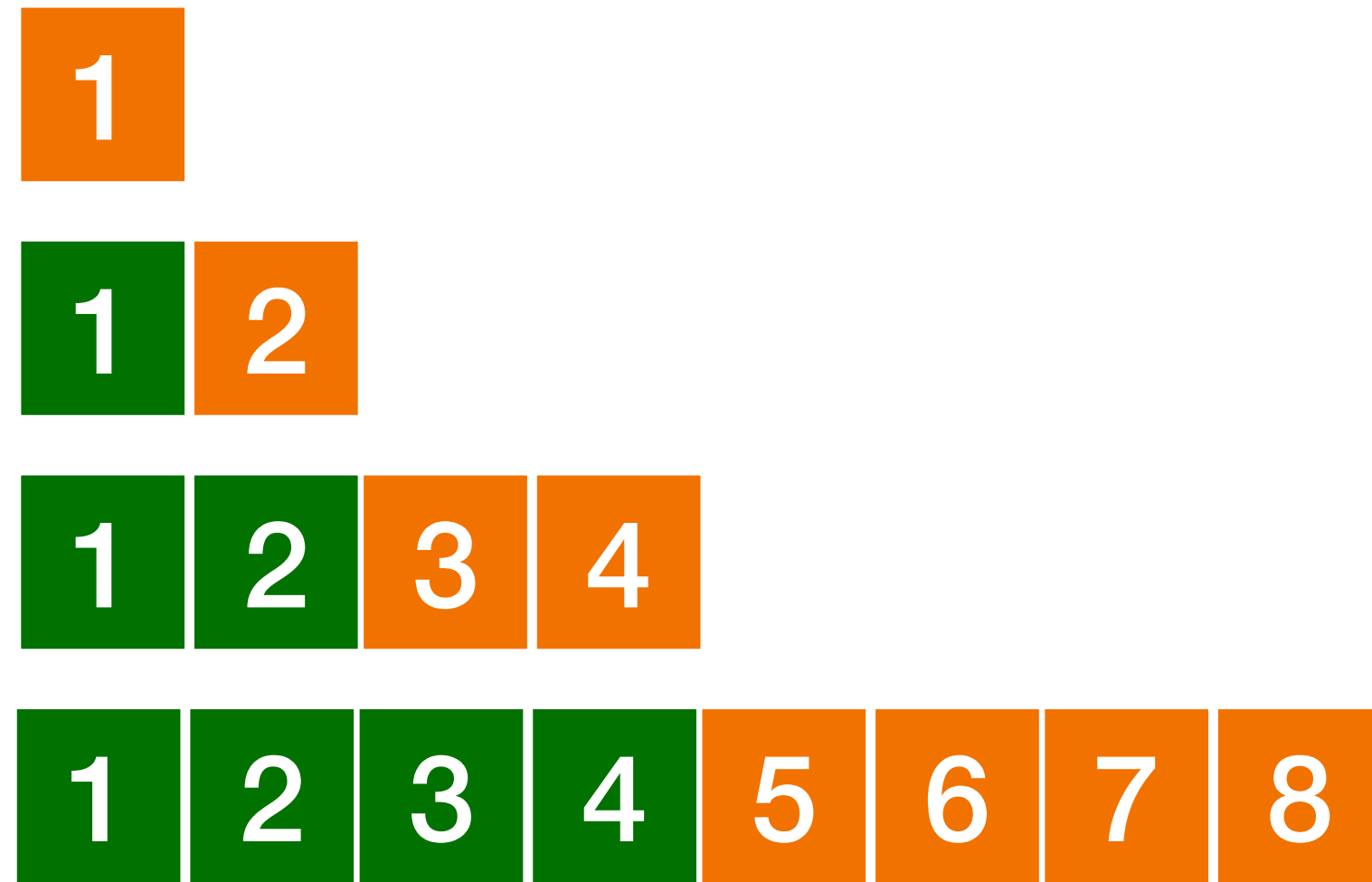
- Effort per item is  $O(n^2)/n = O(n)$
- Not good, when compared to  $O(1)$  in the regular static array.



$$k(1 + 2 + \dots + m) = \frac{k \cdot m \cdot (m + 1)}{2} \in O(k \cdot m^2) \rightarrow O(n^2)$$

# Dynamic Array Allocation

Let's consider **doubling** the current size each time we need to resize.



- Now, effort per item is  $O(n)/n = O(1)$
- Same with the regular static array
- But, where did our extra movements go? !
  - In the hidden constant of  $O(1)$ , which is 3 on the dynamic case ?



$$(1 + 2 + \dots + 2^k) = 2^{k+1} - 1 \in O(2^k) \rightarrow O(n)$$

$$k = \lfloor \log n \rfloor \rightarrow 2^k \approx n$$

# Dynamic Array Allocation

## Aggregate analysis

insert	old capacity	new capacity	insert cost	copy cost
1	0	1	1	–
2	1	2	1	1
3	2	4	1	2
4	4	4	1	–
5	4	8	1	4
6	8	8	1	–
7	8	8	1	–
8	8	8	1	–
9	8	16	1	8
⋮	⋮	⋮	⋮	⋮

$\lceil \log n \rceil \leftarrow \bullet$

- $c_i$  is the cost of  $i^{th}$  insertion to the array
- If  $i = 2^k + 1$ , for some  $k > 0$ , then  $c_i = i$ .
  - Else,  $c_i = 1$ .

$$\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lceil \log n \rceil} 2^j$$

$$< n + 2n$$

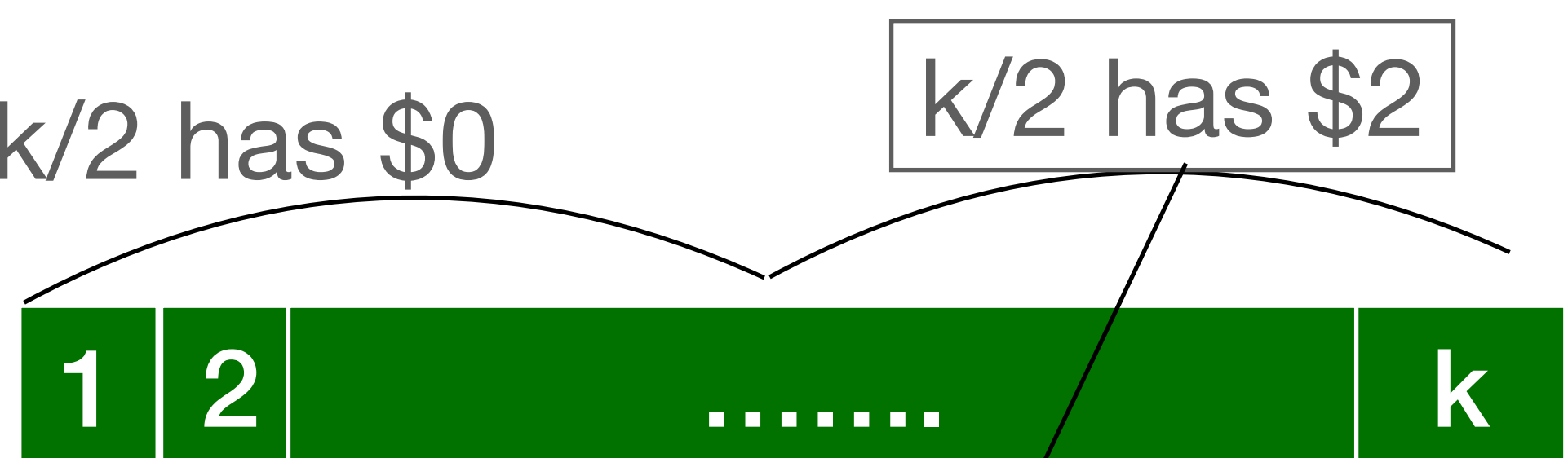
$$< 3n$$

- The cost per item is less than  $\frac{3n}{n} = 3$
- Therefore,  $O(1)$ .

# Dynamic Array Allocation

## Accounting Method

\$3 per each insert will guarantee to avoid bankruptcy.



If half of the elements have \$2 on them, it is enough to copy the old items. Since regular \$1 cost will be required per each, \$3 will surely avoid bankruptcy.

\$k is used to copy k elements



Should have \$2 left on it after **insertion**



# Dynamic Array Allocation

## Potential Method

$$\phi(D_i) = 2 \cdot \underset{\text{\# of elements}}{size(D_i)} - \underset{\text{Total space of the array}}{capacity(D_i)} \qquad \phi(D_0) = 0$$

$\forall i, \phi(D_i) \geq 0$ , since after each resize capacity is 2 times the number of elements.

**Case 1: There is space in the capacity, so no resize is required.**

- Actual cost:  $c_i = 1$
- $\phi(D_i) - \phi(D_{i-1}) = 2 \cdot size(D_i) - capacity(D_i) - 2 \cdot size(D_{i-1}) + capacity(D_{i-1}) = 2$ , since
$$size(D_i) = size(D_{i-1}) + 1$$
$$capacity(D_i) = capacity(D_{i-1})$$
- Amortized cost :  $\hat{c}_i = 1 + 2 = 3$

# Dynamic Array Allocation

## Potential Method

$$\phi(D_i) = 2 \cdot \underset{\text{\# of elements}}{size(D_i)} - \underset{\text{Total space of the array}}{capacity(D_i)} \qquad \phi(D_0) = 0$$

$\forall i, \phi(D_i) \geq 0$ , since after each resize capacity is 2 times the number of elements.

**Case 2: There is NO space in the capacity, so resizing is required.**

- Actual cost:  $c_i = 1 + capacity(D_{i-1})$
- $$\begin{aligned} \phi(D_i) - \phi(D_{i-1}) &= 2 \cdot size(D_i) - capacity(D_i) - 2 \cdot size(D_{i-1}) + capacity(D_{i-1}) \\ &= 2 - capacity(D_{i-1}), \end{aligned}$$

since  $size(D_i) = size(D_{i-1}) + 1$  and  $capacity(D_i) = 2 \cdot capacity(D_{i-1})$
- Amortized cost :  $\hat{c}_i = 1 + capacity(D_{i-1}) + 2 - capacity(D_{i-1}) = 3$

# Dynamic Array Allocation

- What if we want to support resize on delete operation ?
- When the number of items in the array become less, we shrink the array.
- What would be a good strategy?
  - Halve the array when items become  $1/2$  ???

# Amortized Dictionary Data Structure

*Dictionary structures are always in the heart of computing.*

*We have many alternatives. Here is one of those ...*

Problem: Given  $n$  items, provide an efficient way to support search and update.

Main idea:

- Instead of a single list, maintain a collection of arrays, say  $A_0, A_1, \dots, A_{k-1}$
- Array  $A_i$  has either exactly  $2^i$  elements or empty with zero elements.
- Each array is sorted.
- There is no relation or order between the arrays.

Depending on the number  $n$ , how will we decide on the number of arrays ( $k=?$ ), and how will we decide which ones will be empty and full ?

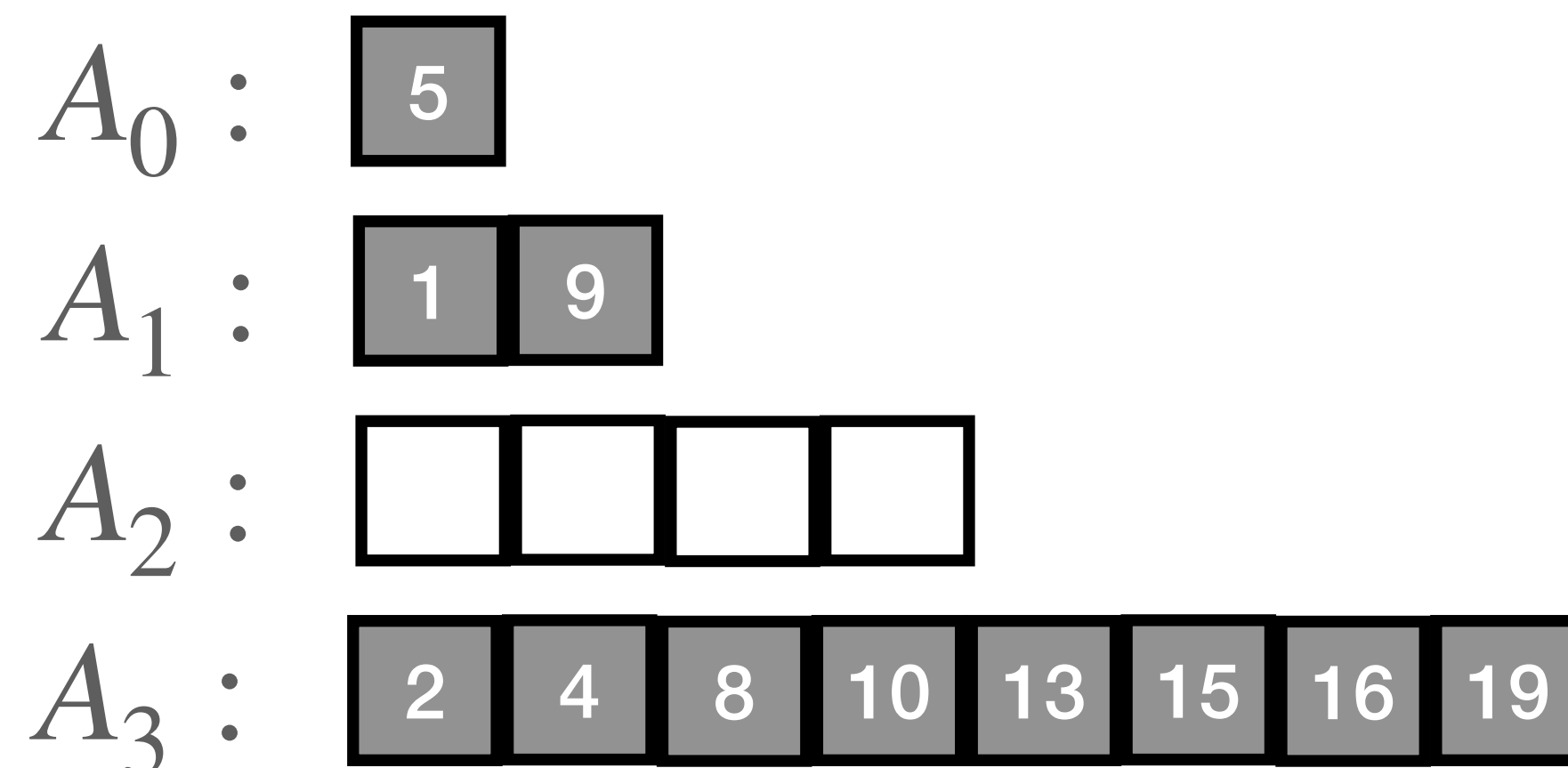
# Amortized Dictionary Data Structure

Depending on the number  $n$ , how will we decide on the number of arrays ( $k=?$ ), and how will we decide which ones will be empty and full ?

Each integer can be written as the sum of the powers of 2.  
Actually, this is its binary representation !

$(n = 11) \rightarrow n = 1011$  indicates  $n=8.1+4.0+2.1+1.1$

**4 arrays  $A_3, A_2, A_1, A_0$  with sizes 8,4,2,1, respectively.  $A_2$  is empty.**

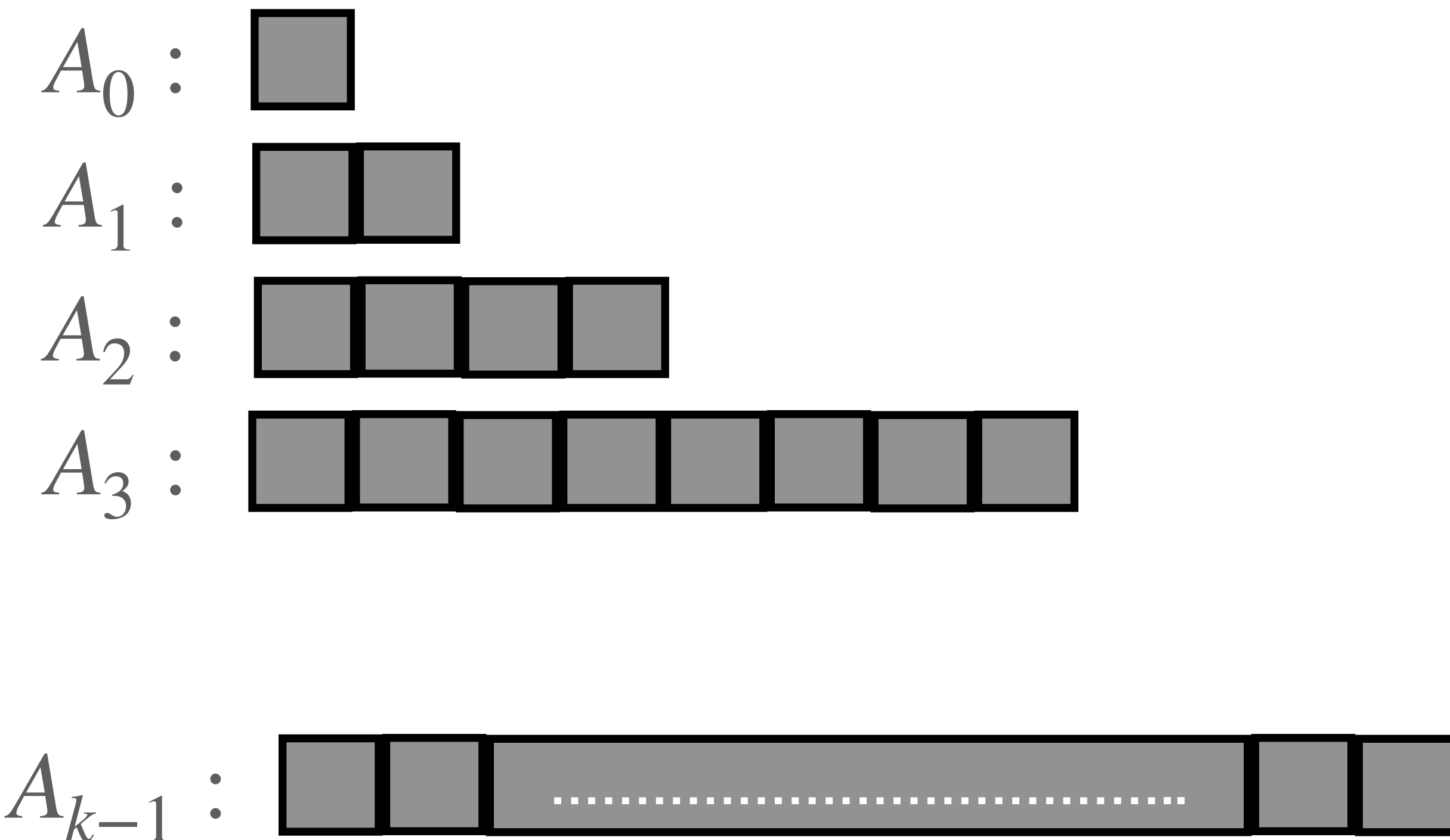


- Given  $n$ , we maintain  $\lceil \log(n + 1) \rceil$  arrays.
- Each has its corresponding size.
- The ones with a 1 bit are full, others empty

*What do you think about the construction cost?*

# Amortized Dictionary Data Structure

Searching for a key on this dictionary which maintains  $n$  keys in total ?



$$k = \lceil \log(n + 1) \rceil$$

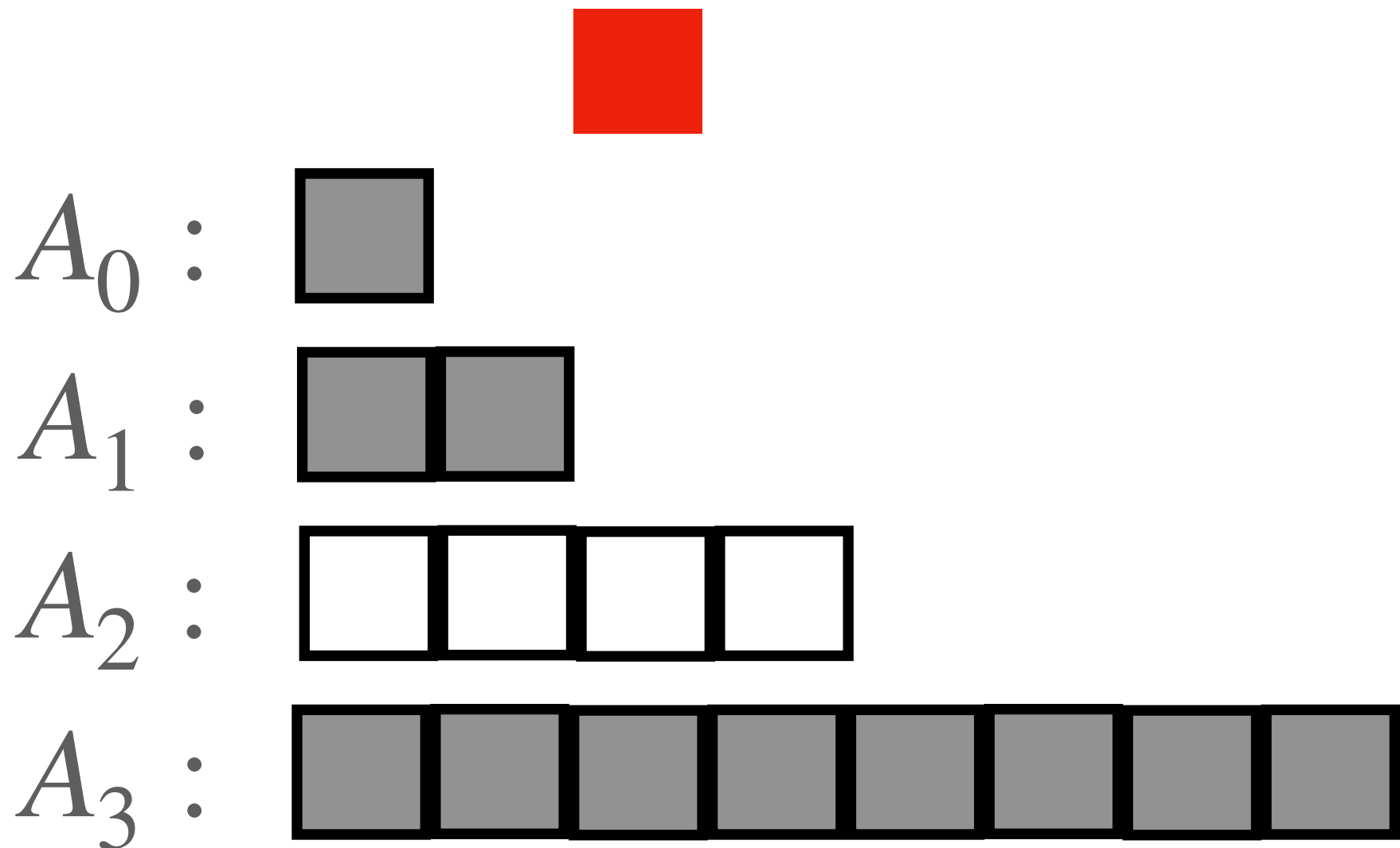
- Investigate each array one-by-one.
- We have  $k = \lceil \log(n + 1) \rceil$  arrays.
- Search on a sorted array of  $t$  elements is  $O(\log t)$  via binary search.
- Longest array size  $\leq n$ .
- At most  $k$  arrays will be searched.
- Each search is  $O(\log n)$  time.

**Then, overall cost of search is**

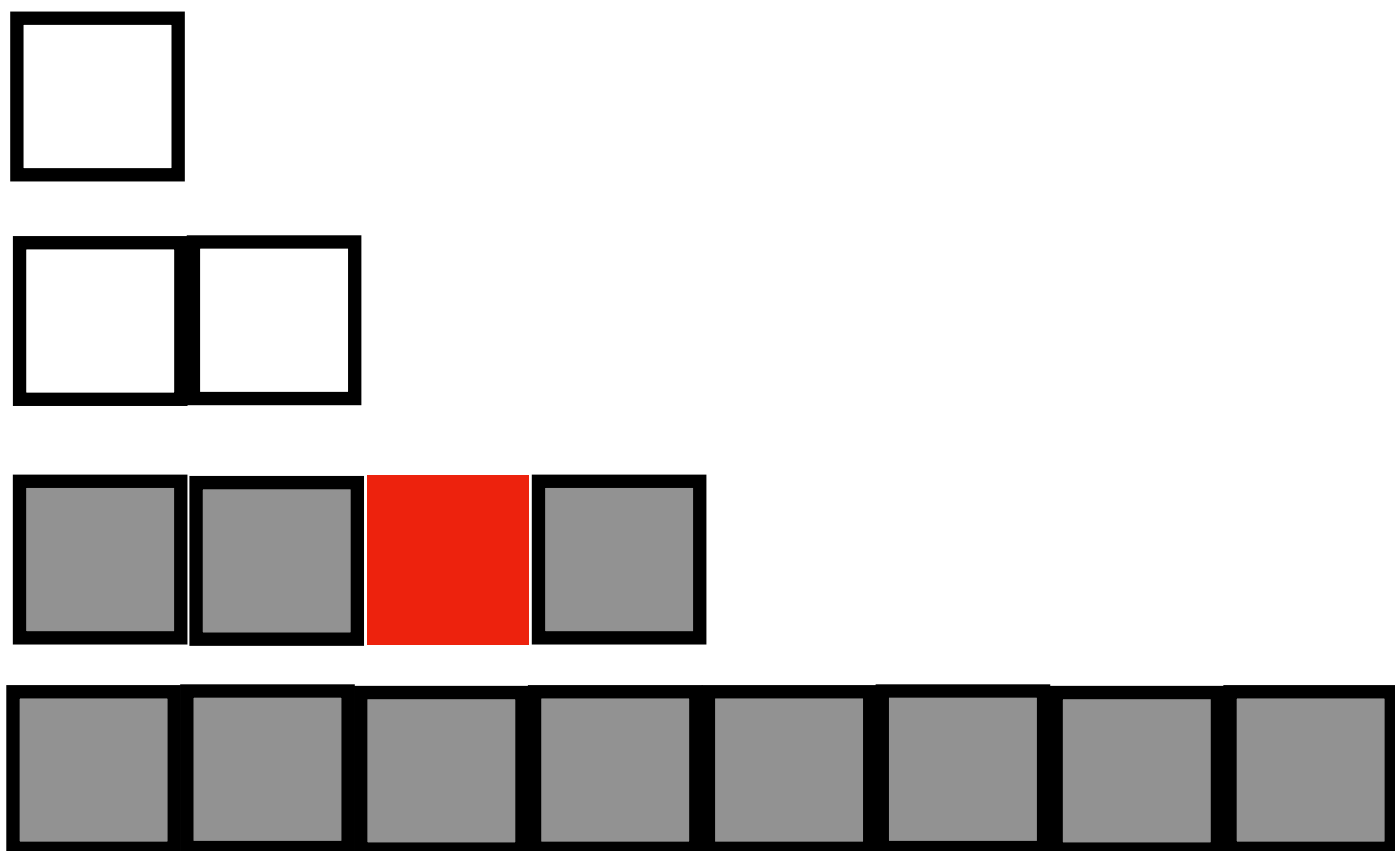
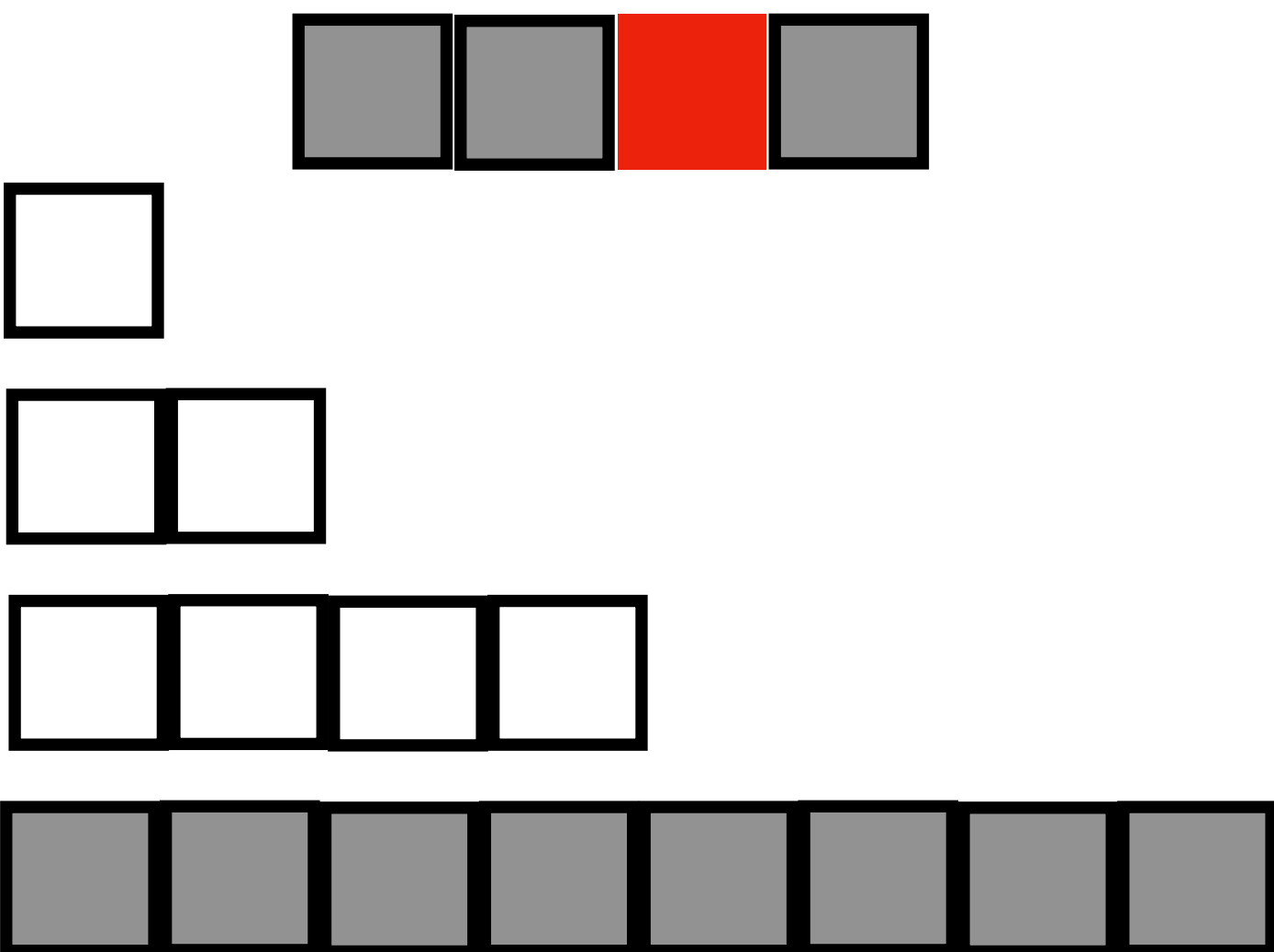
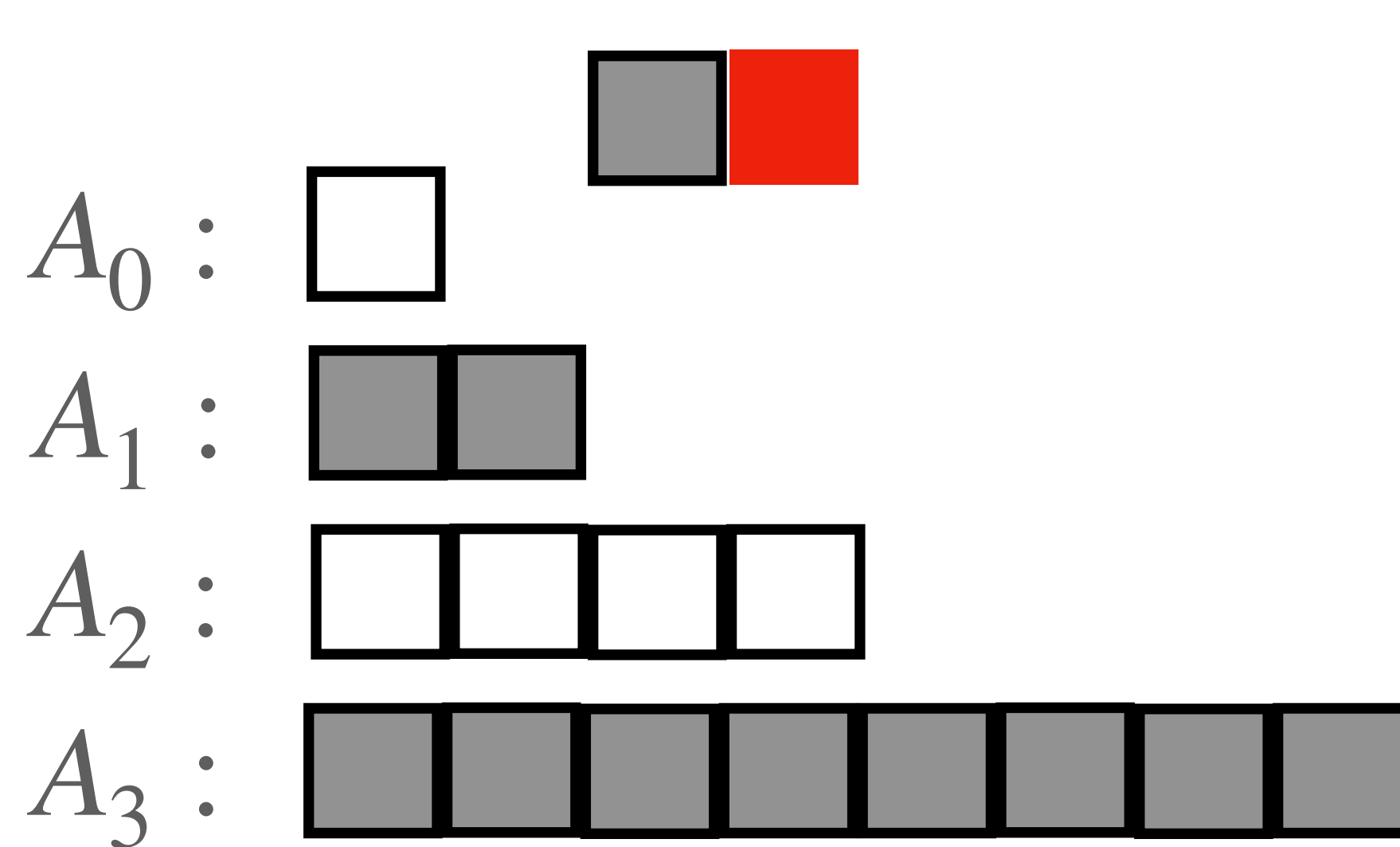
$$k \cdot \log n \in O(\log^2 n)$$

# Amortized Dictionary Data Structure

INSERTING a new key



- Put new element into array  $H$  of size 1.
- $i = 0$
- Check  $A_i$ . If empty, copy  $H$  to  $A_i$  and stop. Else,  $H \leftarrow \text{merge}(A_i, H)$  and  $i \leftarrow i + 1$  and repeat.

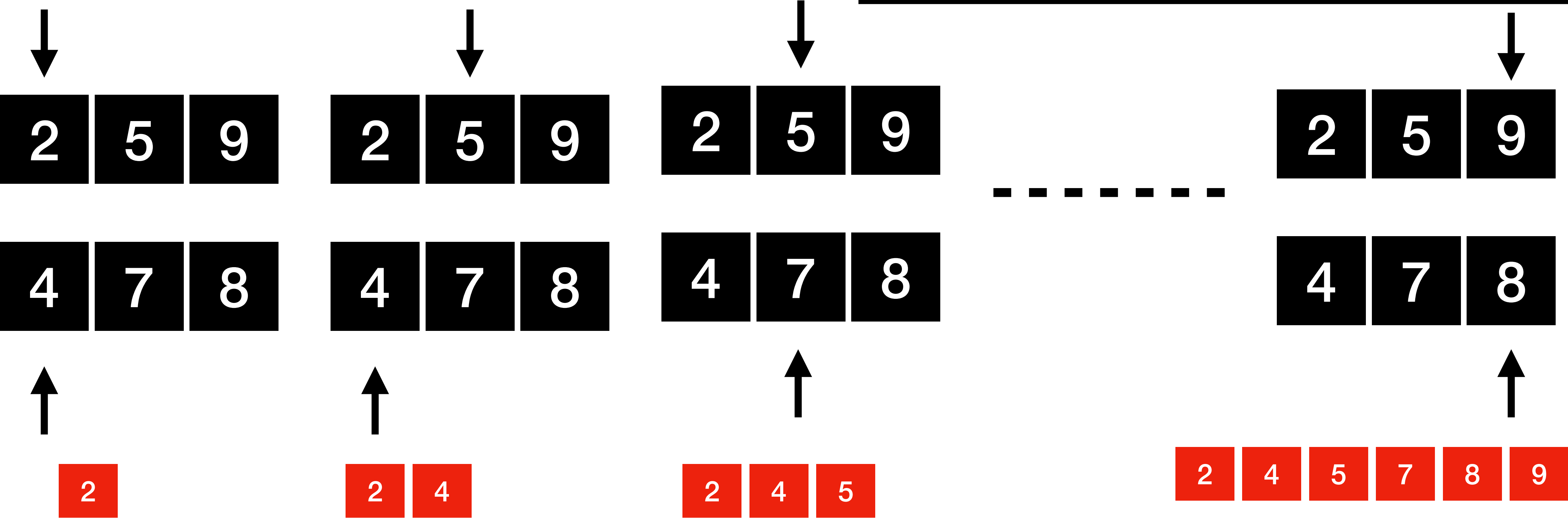


# Amortized Dictionary Data Structure

INSERTING a new key

Merging two sorted list, each with size  $\ell$  requires less than  $2\ell$  comparisons !

- Put new element into array  $H$  of size 1.
- $i = 0$
- Check  $A_i$ . If empty, copy  $H$  to  $A_i$  and stop. Else,  $H \leftarrow \text{merge}(A_i, H)$  and  $i \leftarrow i + 1$  and repeat.





# Amortized Dictionary Data Structure

INSERTING a new key

- Worst case: We visit and merge all arrays in the dictionary, e.g.  $n = 2^k - 1$  elements in the dictionary for some  $k$ , and we are adding the  $2^k$ th element
- What will be the cost of this worst case?
- Merging two sorted list, each with size  $\ell$  requires less than  $2\ell$  comparisons !
- Therefore,  $C = 2 + 4 + 8 + \dots + 2^k = 2^{k+1} - 1$ . Since  $k \in O(\log n)$ ,  $C \in O(n)$ .

Once such a worst case happens, can it appear repeatedly ?

NO!

So, regular worst case analysis is not tight! We can try an amortized approach by computing the cost of , say  $t$ , consecutive insert operations.

# Amortized Dictionary Data Structure

INSERTING a new key has  $O(\log n)$  amortized complexity.

The merge cost of  $A_i$  is at most  $2 \cdot 2^i$  as merging two list each with  $\ell$  costs  $2\ell$ .

During  $n$  insertion operations,

- $A_0$  will be subject to merge  $n/2$  times with a cost of 2.
- $A_1$  will be subject to merge  $n/4$  times with a cost of 4.
- $A_2$  will be subject to merge  $n/8$  times with a cost of 8.
- Totally  $O(\log n)$  arrays will be subject to merge each with  $O(n)$  cost.
- **Therefore, this makes total cost  $O(n \log n)$  for  $n$  insertions, which makes amortized cost of insertion  $O(\log n)$ .**

*This is exactly the same with the binary counter amortized analysis with one difference as the cost of flipping  $k^{th}$  bit is  $2^k$  instead of a constant 1 unit.*

# Amortized Dictionary Data Structure

## DELETING a key

- Assume we will be deleting an item from the array  $A_i$  that includes  $2^i$  elements.
- Split  $A_i$  into small arrays of length  $1, 2, 4, \dots, 2^{i-1}$ . Notice that  $1 + 2 + 4 + \dots + 2^{i-1} = 2^i - 1$ , which is exactly the number of remaining elements in  $A_i$ . Delete all items from  $A_i$ .
- For each of these small arrays, insert it into the dictionary again. Insert operation starts with the corresponding list length, i.e., small array of size 1, start with  $A_0$ , size 2 start with  $A_1$ , and continue accordingly.

There can be at most  $\log n$  small arrays after deleting an element. The amortized cost of insertion process is  $O(\log n)$  as we showed previously. So the cost of deleting an element in the worst case is  $O(\log^2 n)$  with the proposed method.

*There might be other ways of deletion as well ?*

# Reading assignment

- Read chapter 17 Amortized Analysis from Cormen and also related chapters from other text books or resources on the internet.
- Next week we will study recursions and divide-and-conquer type algorithms.