Data Structures

Array Lists

CS 225

Brad Solomon

September 8, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Course Policy Reminders

Request an extension? → Extension Request Form

Missed an exam?

cs225 admin

Want to email a professor about the class?

Post your art on #mp-art!

Learning Objectives

Discuss data variables for implementing array lists

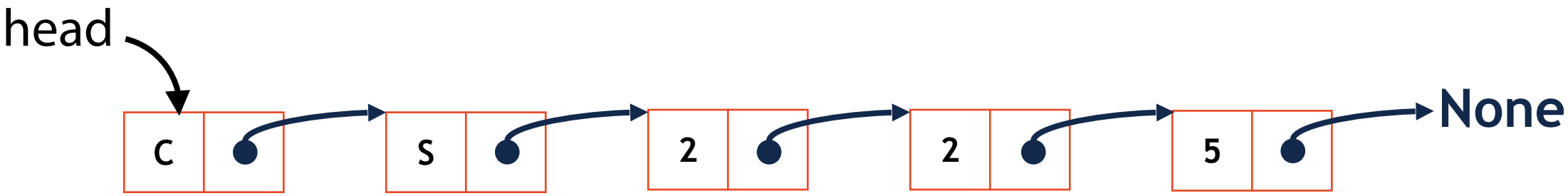
Review array list implementations

Discuss amortized analysis

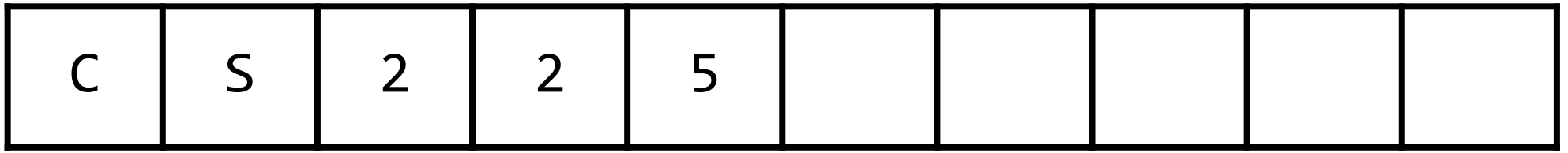
Consider extensions to lists

List Implementations

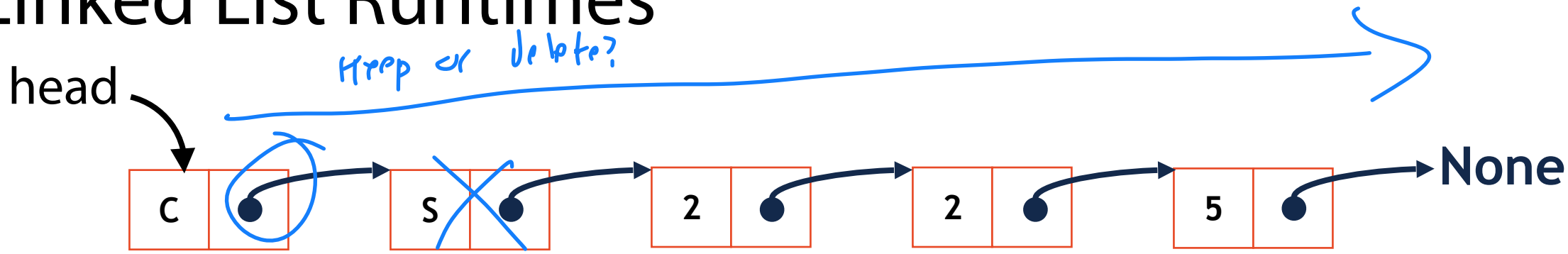
1. Linked List



2. Array List



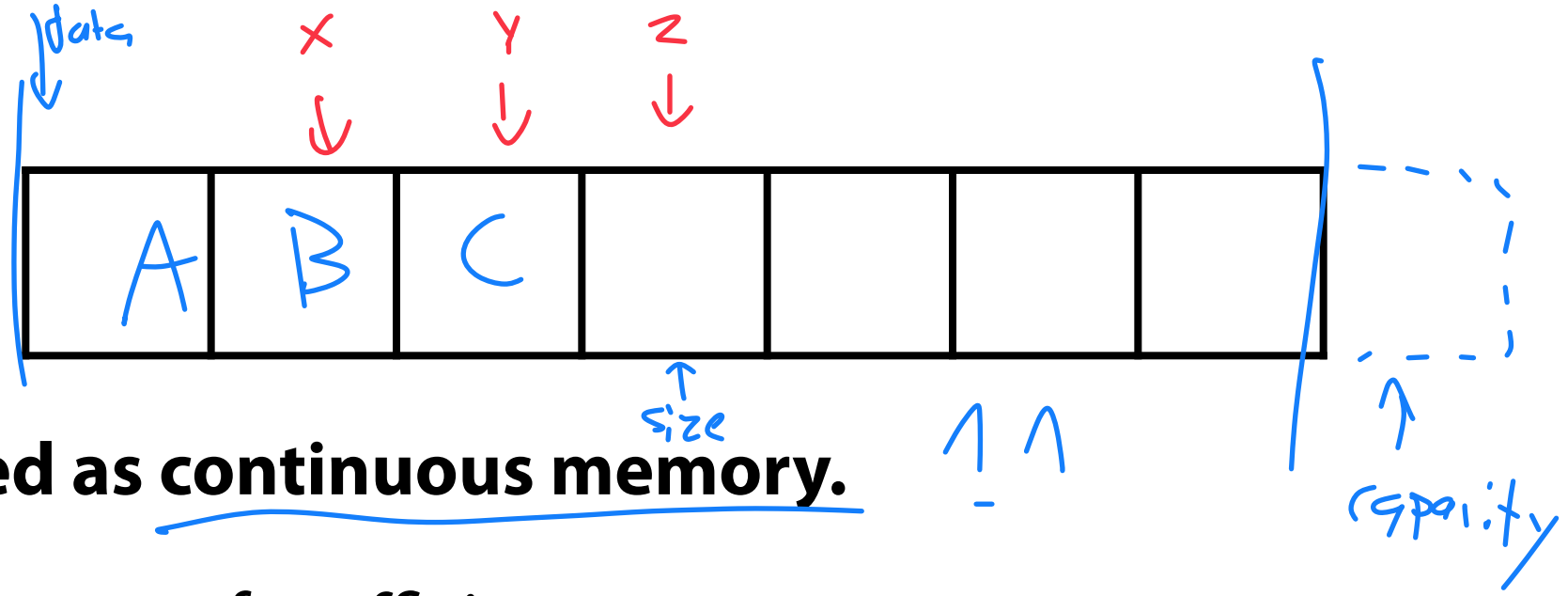
Linked List Runtimes



Easy to modify at known location

	@Front	@RefPointer	@Index
Insert	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Array List



4th thing - type of data

An array is allocated as continuous memory.

Three values are necessary for efficient array usage:

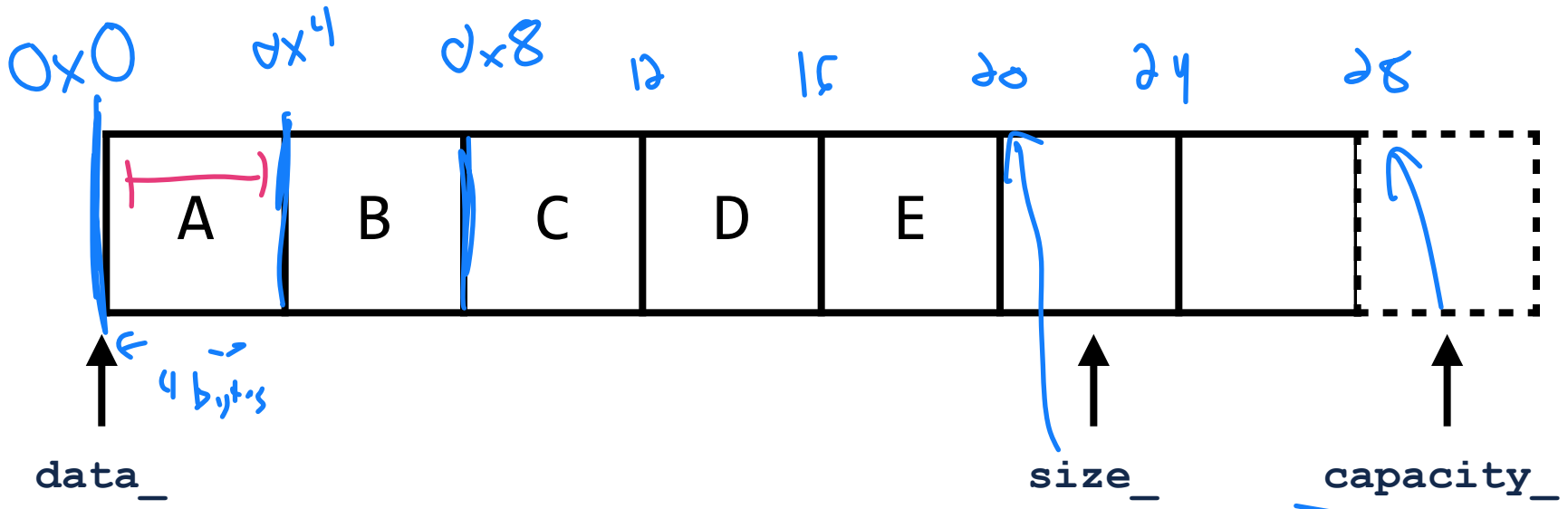
1) Data – the start pos of array (pointer)

2) size – the current # of items in my array

3) capacity – the total # of allocated spaces

Array List

↳ int



In C++, vector is implemented as:

These are pointers! Value as mem address

1) **Data:** Stored as a pointer to array start

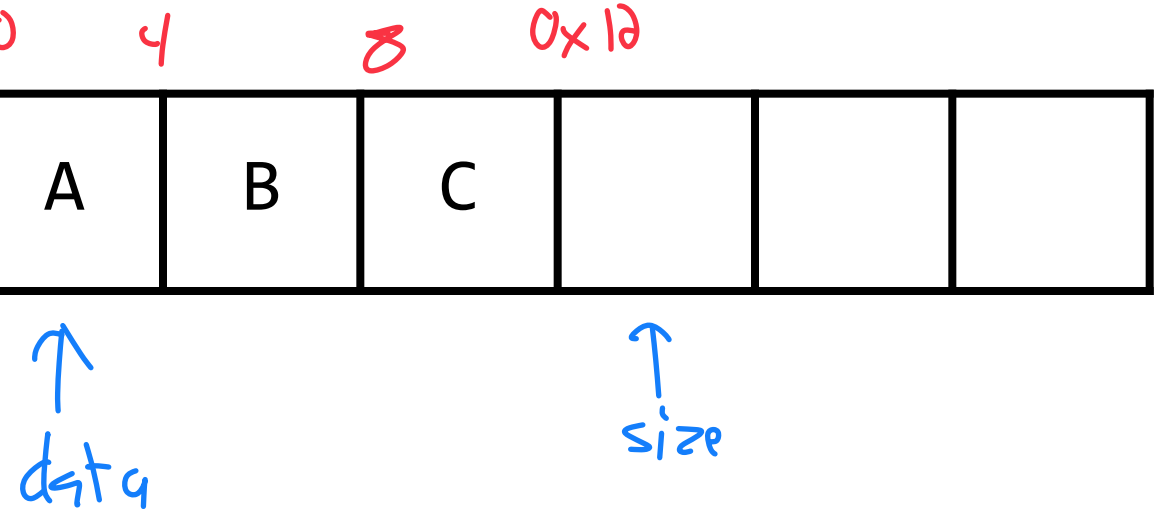
Will always be start of block

2) **Size:** Stored as a pointer to the next available space

3) **Capacity:** Stored as a pointer past the end of the array

List.h

```
 1  #pragma once
 2
 3  template <typename T>
 4  class List {
 5  public:
       /* --- */
…
25  private:
26    T *data_;
27
28    T *size_;
29
30    T *capacity_;
…
       /* --- */
    };
```



int *p

p++; ← This moved one memory address

If I want to know the number of items in the array: size - data = 3

int *

sizeof(type)

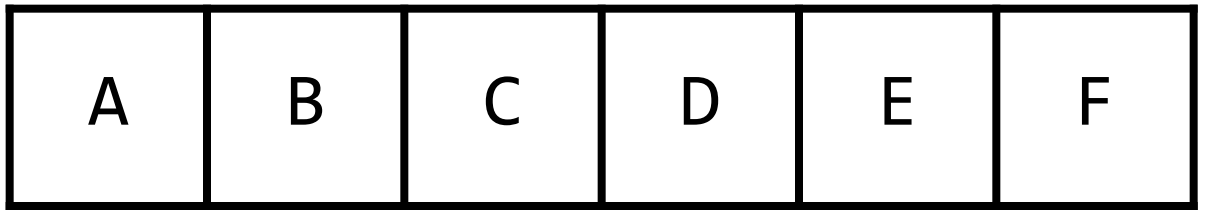
0x10 - 0x0

4

3-0 = 3

List.h

```
 1  #pragma once
 2
 3  template <typename T>
 4  class List {
 5  public:
 …      /* --- */
25  private:
26    T *data_;
27
28    T *size_;
29
30    T *capacity_;
 …
        /* --- */
    };
```



If ints

size == capacity

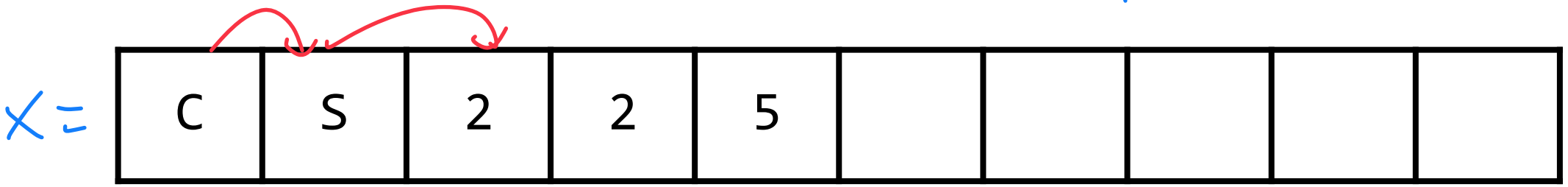
B/c pointer

How do I know if I'm at capacity? size_ == capacity_

Tiny bit faster!

Array List: [ ]

x[2]



data

sizeof(data type) x 2

data + sizeof(data type) . index

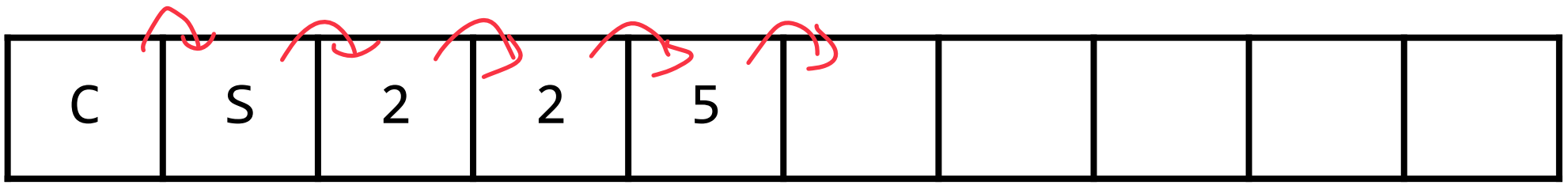
B/c we use pointers

item = data + index;

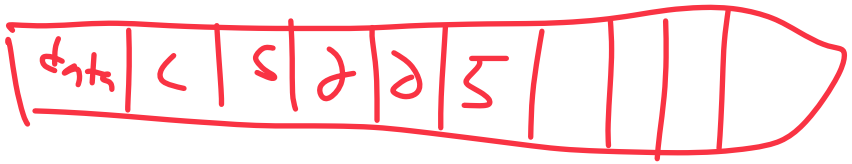
O(1) time

↳ array has efficient random access!

Array List: `insertFront(data)`



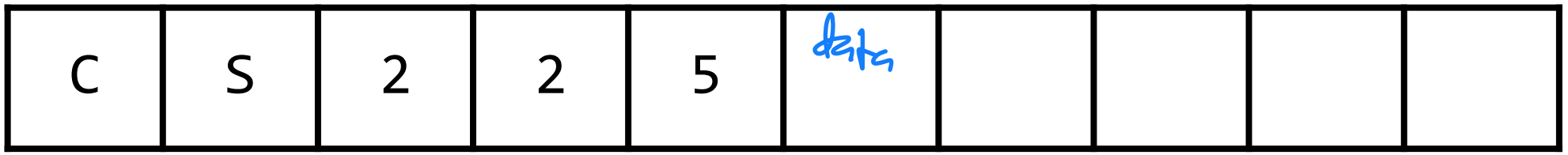
O(1) to look up front item



Because we must move all items

if n items in list, insert Font is O(n)

Array List: `insertBack(data)`



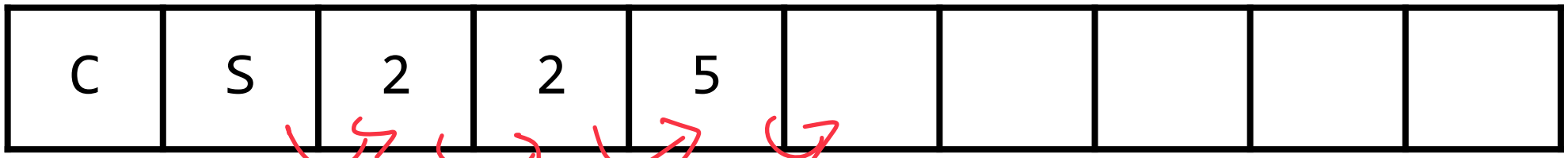
Insert item at size_

size++;

O(1)

Key caveat: only true if not at capacity

Array List: `insert(data, index)`



O(1)
: insert Front

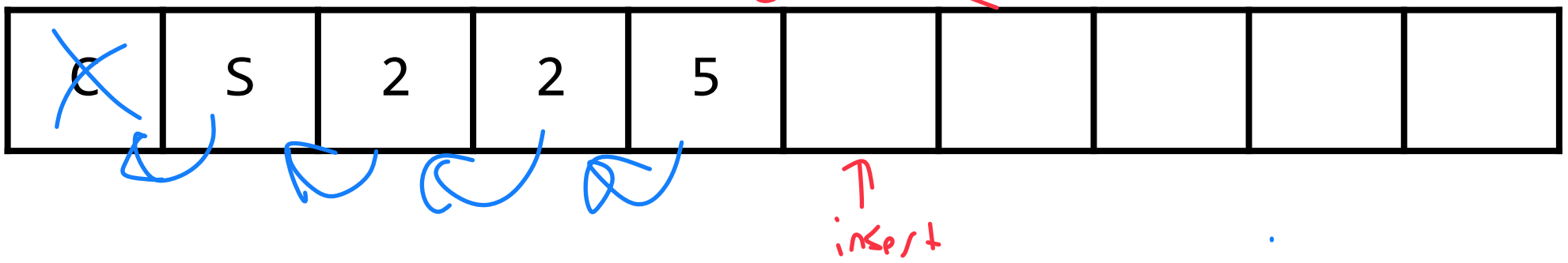
O(1)
: insert Back

(data, 0)

(data, 1)

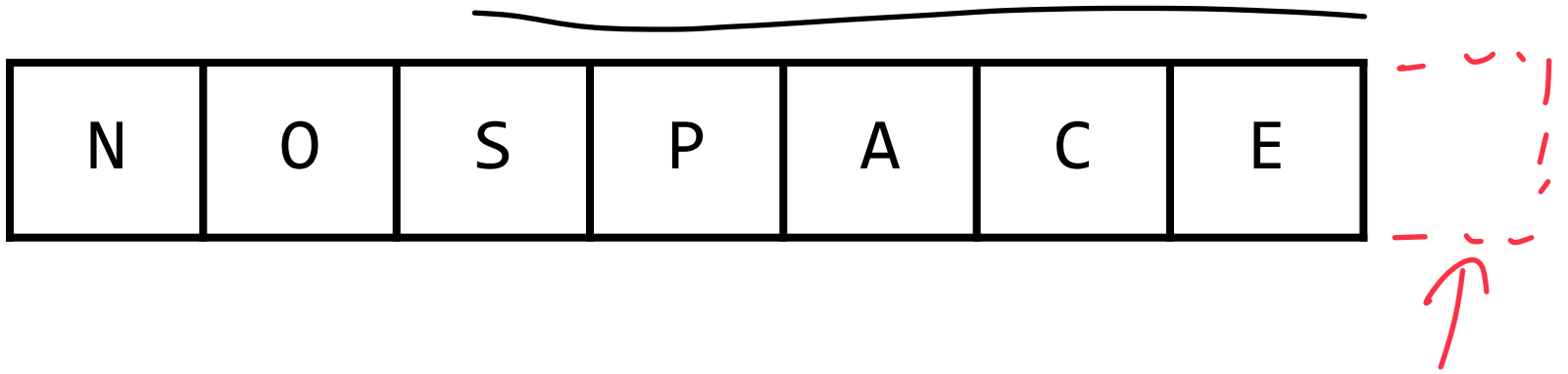
O(n)

Array List: Not at capacity

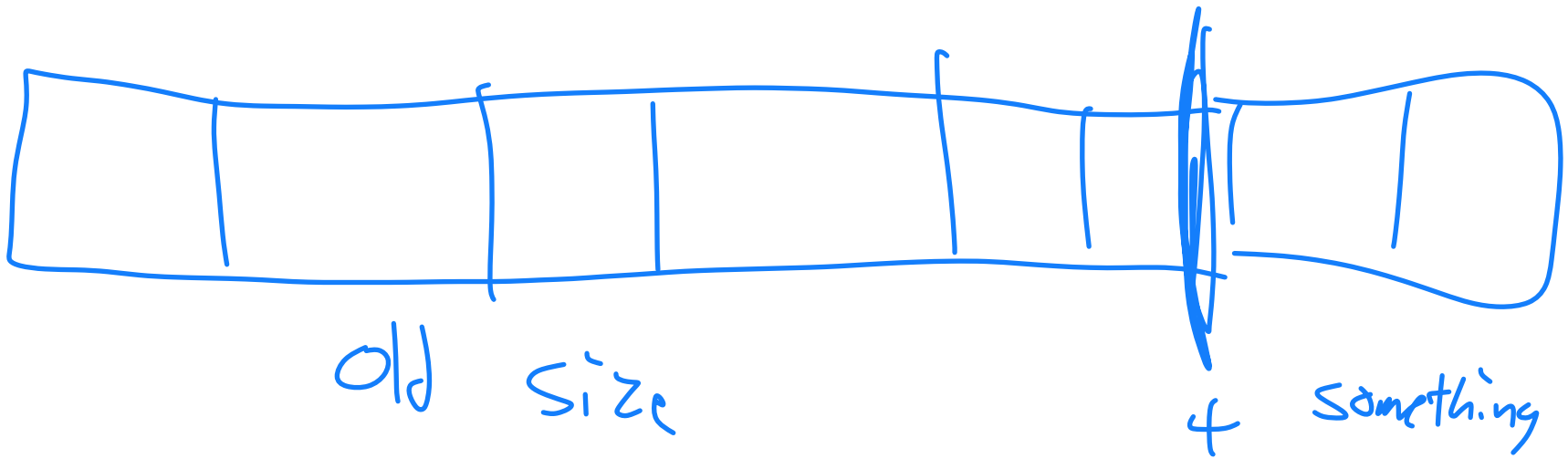


	@Front	@Back	@Index
Insert	O(n)	O(1)	O(n)
Delete	O(n)	O(1)	O(n)

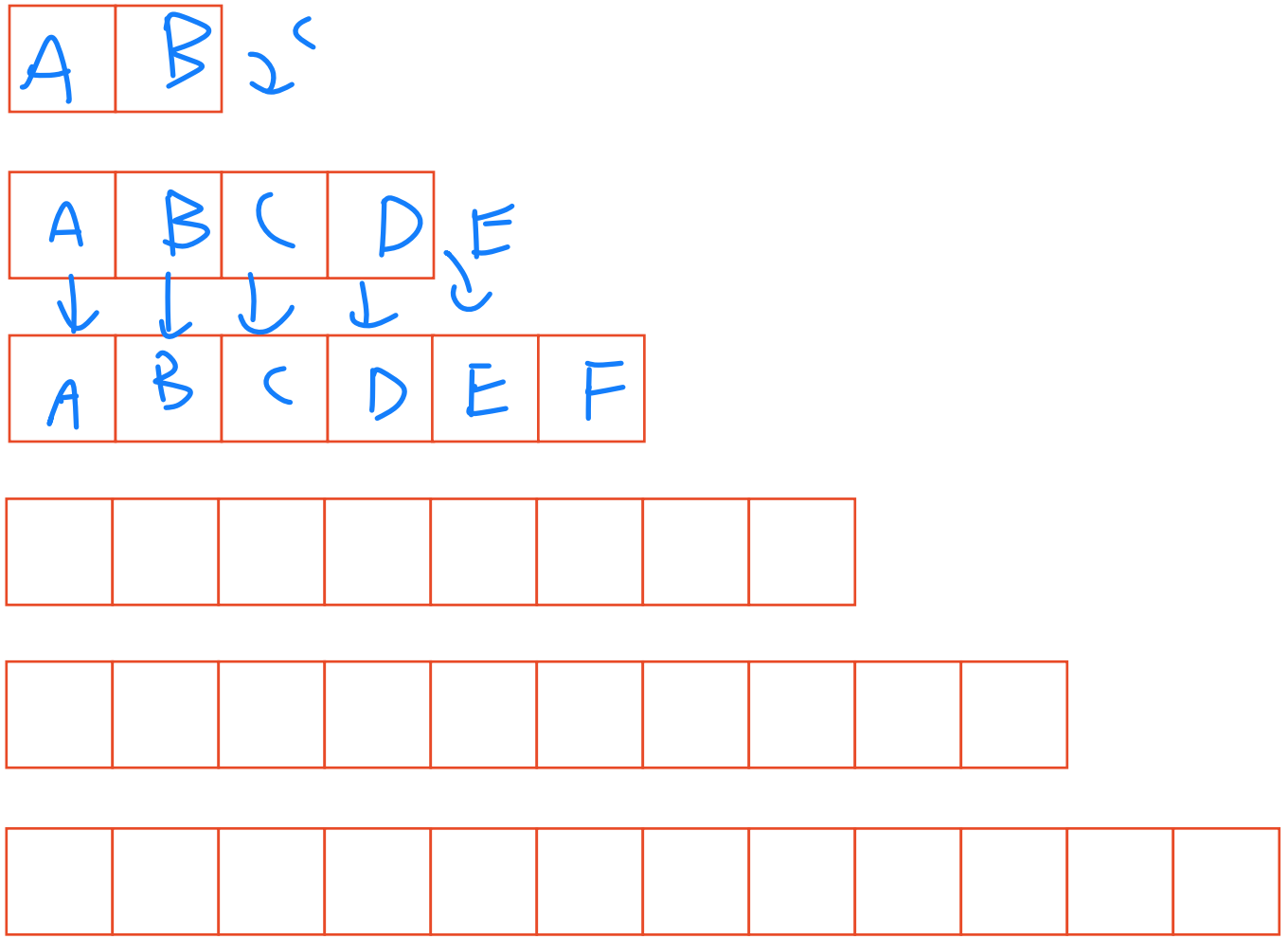
Array List: addspace(data)



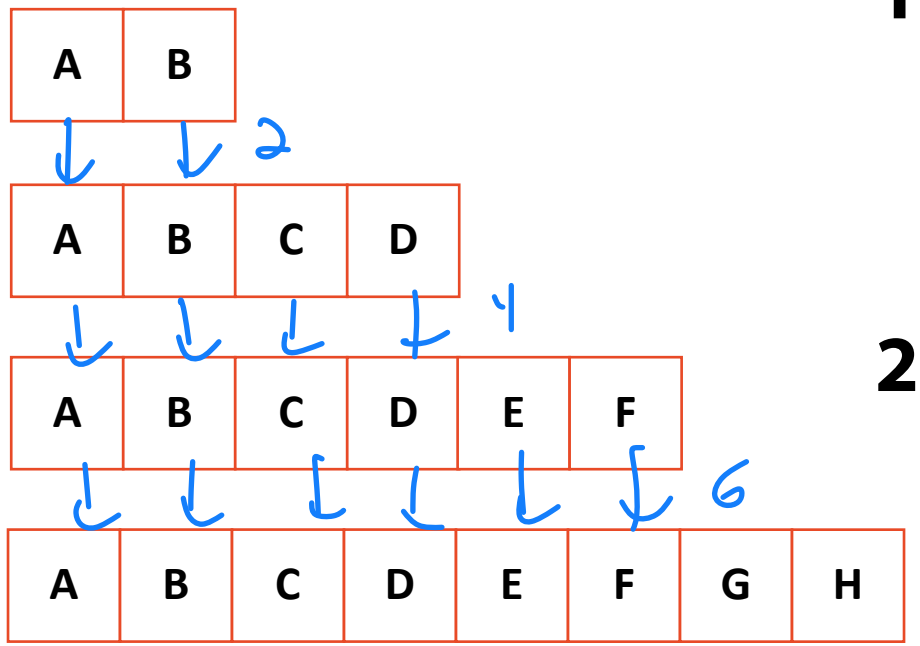
We must allocate New memory



Resize Strategy: +2 elements every time



Resize Strategy: +2 elements every time



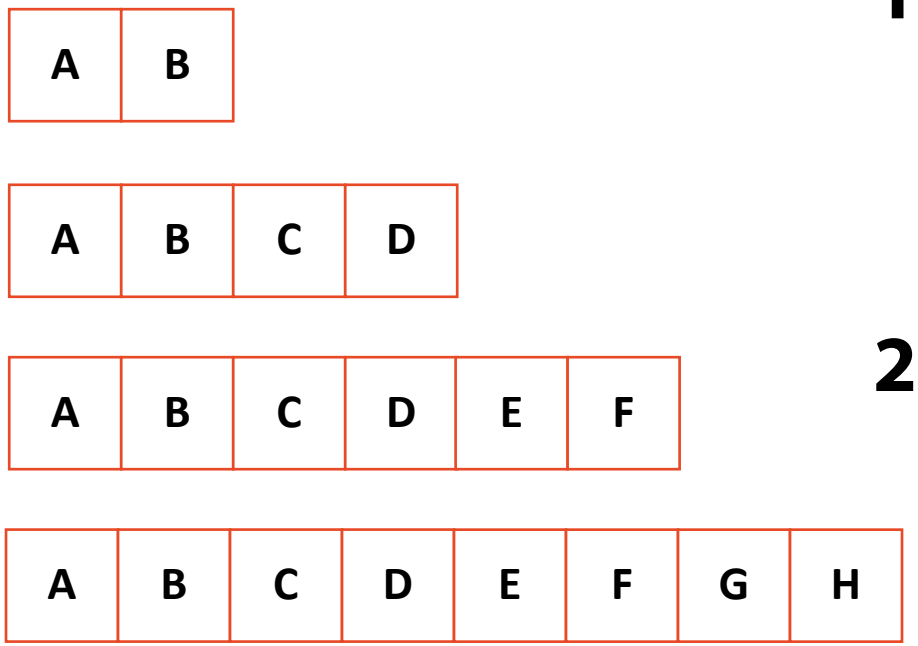
1) How many copy calls per reallocation?

For i^{th} reallocation, $2i$ copies

2) Total reallocations for N objects?

$N/2$ reallocations

Resize Strategy: +2 elements every time



1) How many copy calls per reallocation?

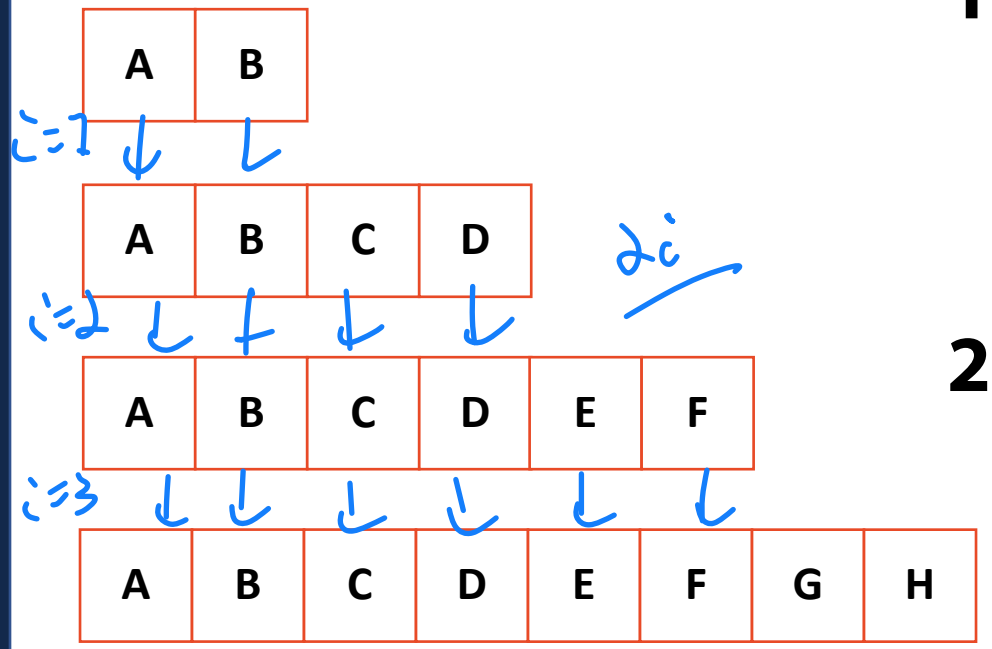
For reallocation i, 2i copy calls are made

2) Total reallocations for N objects?

Let k be the number of reallocs, $k = \frac{N}{2}$

Total number of copy calls: $\sum_{i=1}^{k} 2i = k(k+1) = k^2 + k$

Resize Strategy: +2 elements every time



1) How many copy calls per reallocation?

For reallocation i, 2i copy calls are made

2) Total reallocations for N objects?

Let k be the number of reallocs, $k = \frac{N}{2}$

Total number of copy calls: $\sum_{i=1}^{k} 2i = k(k+1) = k^2 + k$

... For N objects: $\frac{N^2 + 2N}{4}$

Resize Strategy: +2 elements every time

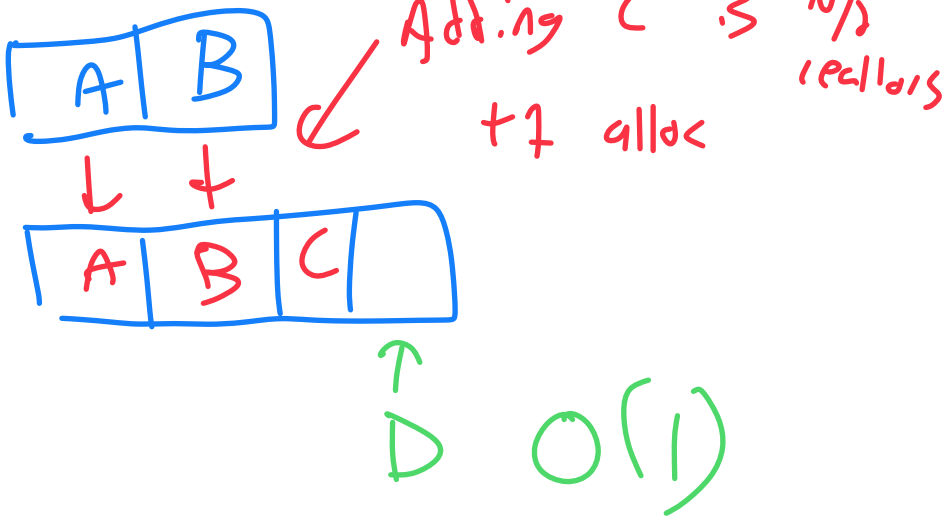
Total copies for N inserts: $\frac{N^2 + 2N}{4}$

Amortized: We inserted N items
↳ total amount of work averaged over input

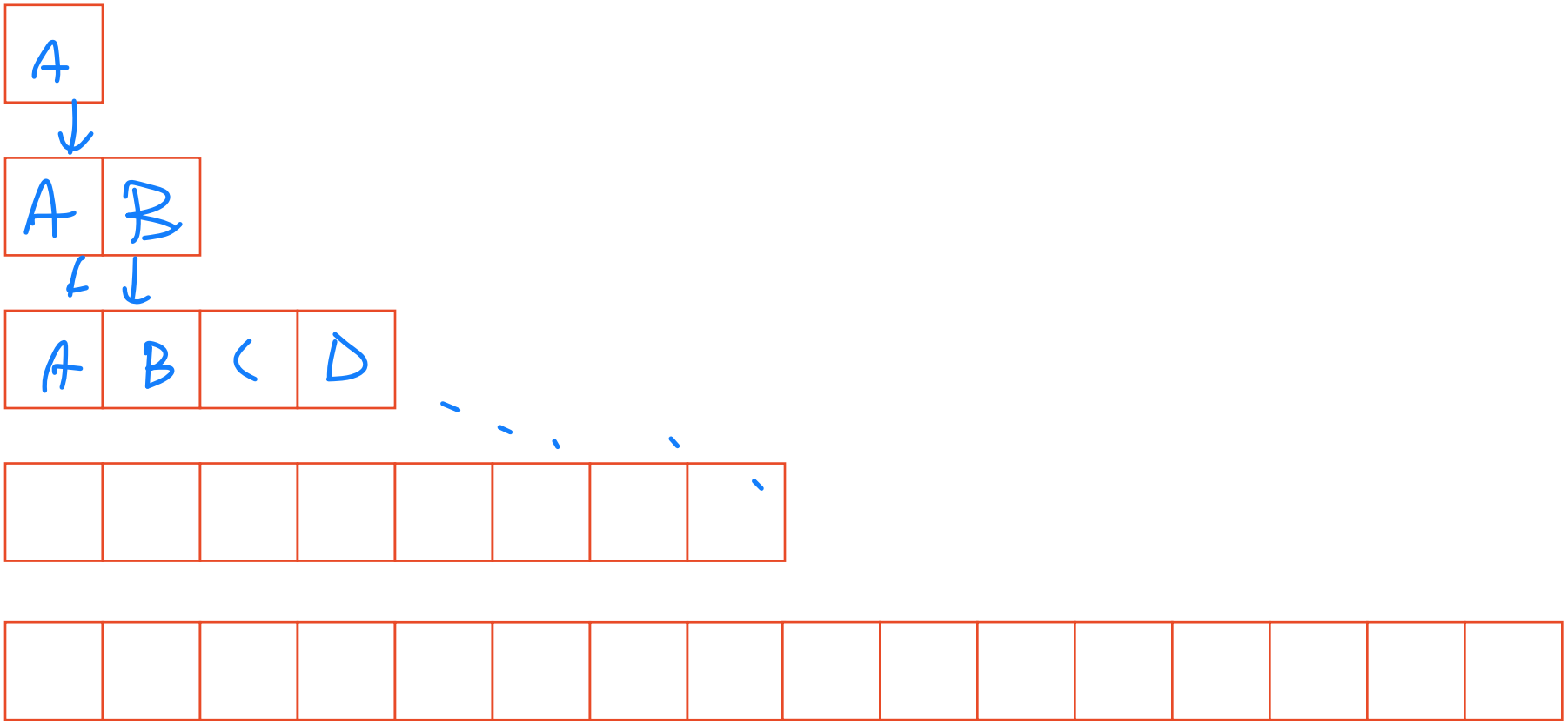
$\left(\frac{N^2 + 2N}{4}\right) / N \Rightarrow \sim N$ work per insert

Big O: Time for one insert

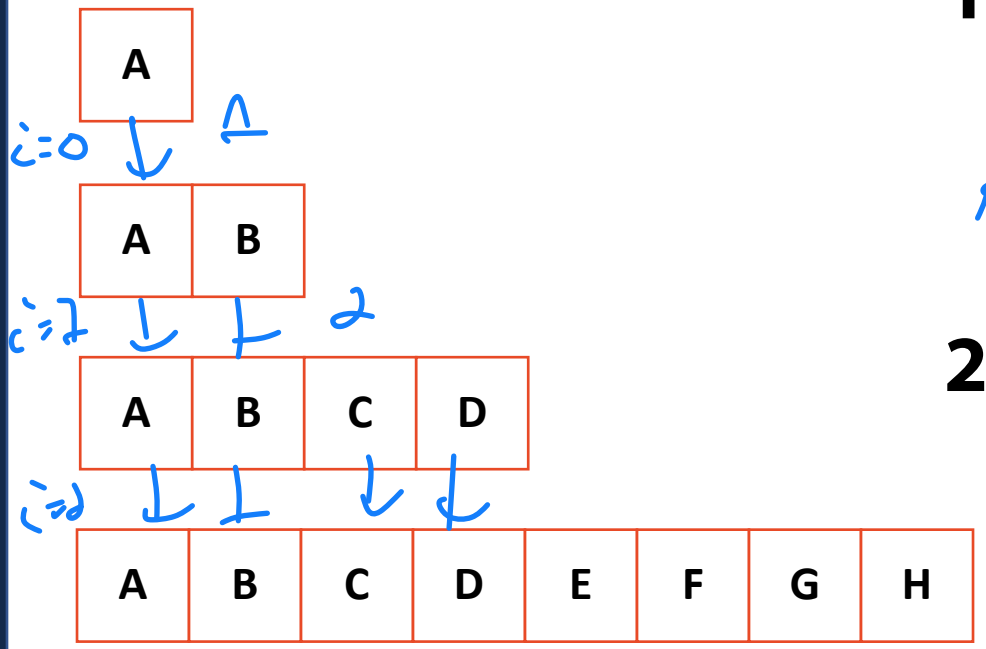
O(n)



Resize Strategy: x2 elements every time



Resize Strategy: x2 elements every time



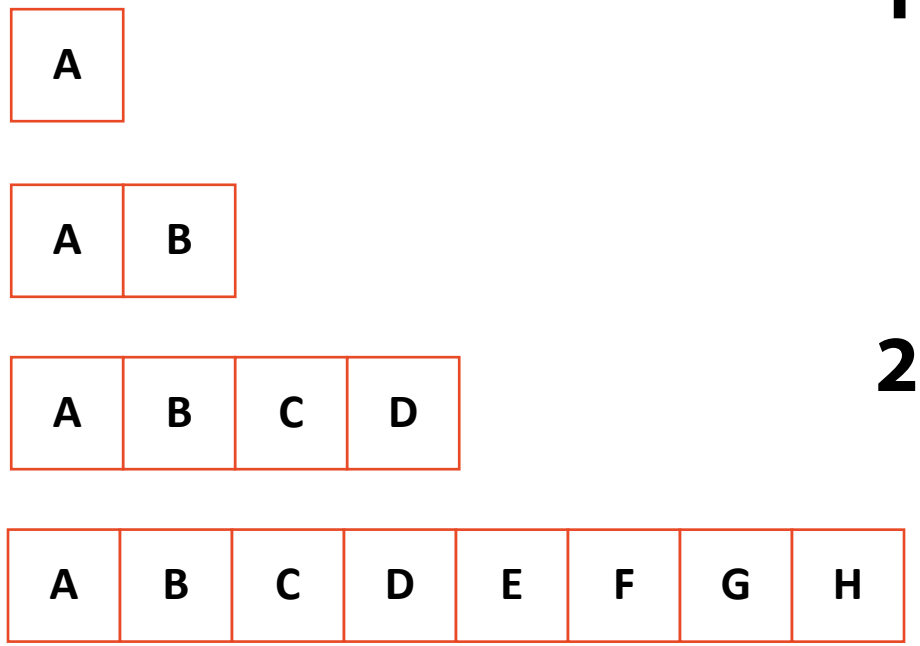
1) How many copy calls per reallocation?

At realloc i, 2^i copies are made

2) Total reallocations for N objects?

$N \leq 2^K \rightarrow K = \lceil \log_2 N \rceil$

Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

For reallocation i, 2^i copy calls are made

Many more copies than +2 per realloc

2) Total reallocations for N objects?

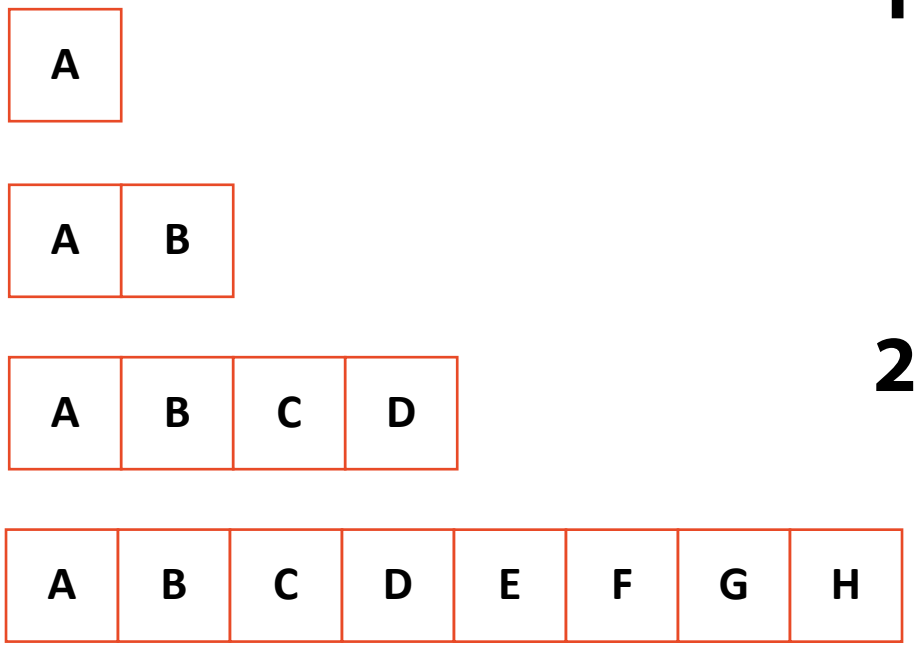
k = final realloc needed = $\lceil log_2 n \rceil$

realloc less

Total number of copy calls:

Continue on Wednesday

Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

For reallocation i, 2^i copy calls are made

2) Total reallocations for N objects?

k = final realloc needed = $\lceil log_2 n \rceil$

Total number of copy calls: $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

… For N objects: $2n - 1$

Resize Strategy: x2 elements every time

Total copies for n inserts: $2n - 1$

Amortized:

Big O:

List Implementation

	Singly Linked List	Array
Look up **arbitrary** location		
Insert after **given** element		
Remove after **given** element		
Insert at **arbitrary** location		
Remove at **arbitrary** location		
Search for an input **value**		

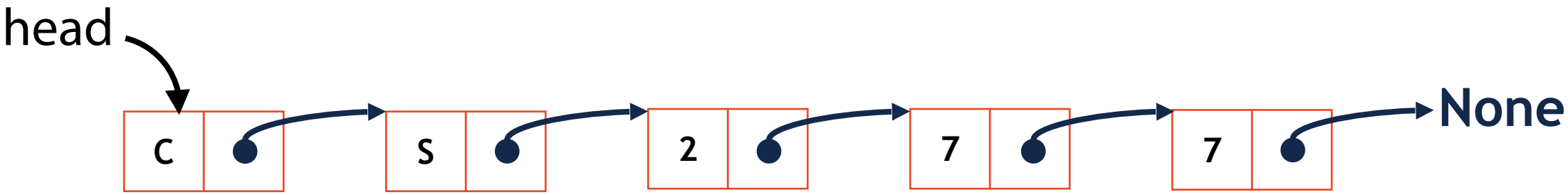
Thinking critically about lists: tradeoffs

The implementations shown are foundational (simple).

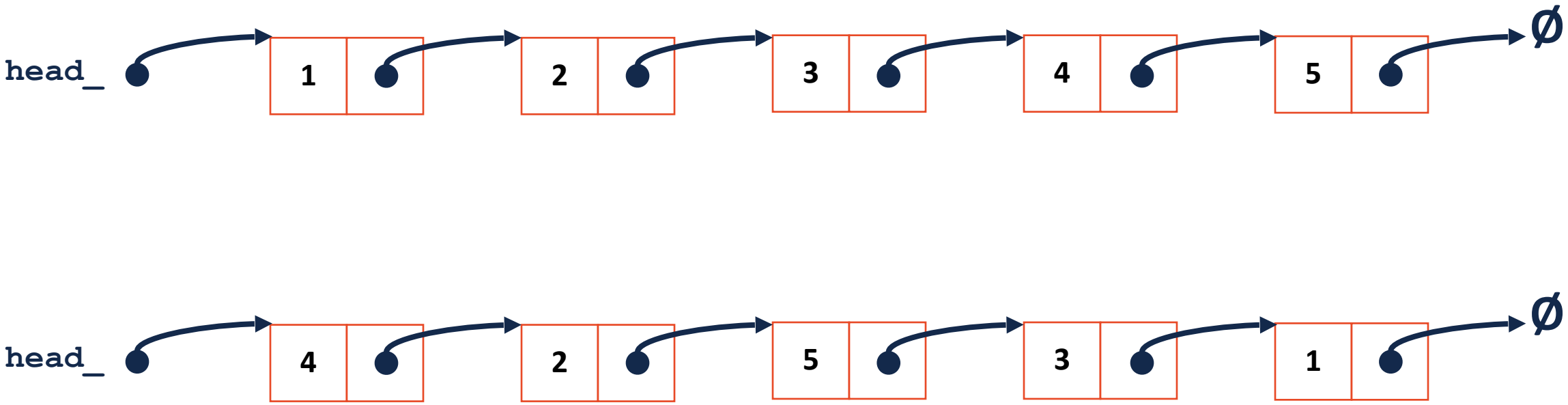
Can we make our lists better at some things? What is the cost?

Thinking critically about lists: tradeoffs

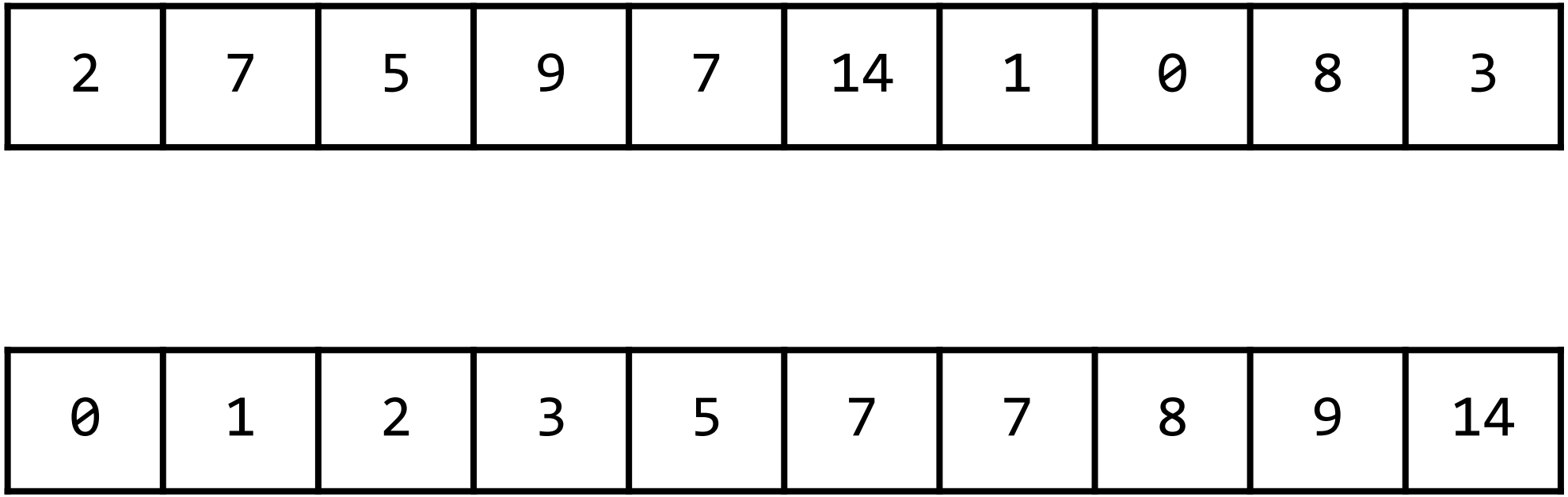
Getting the size of a linked list has a Big O of:



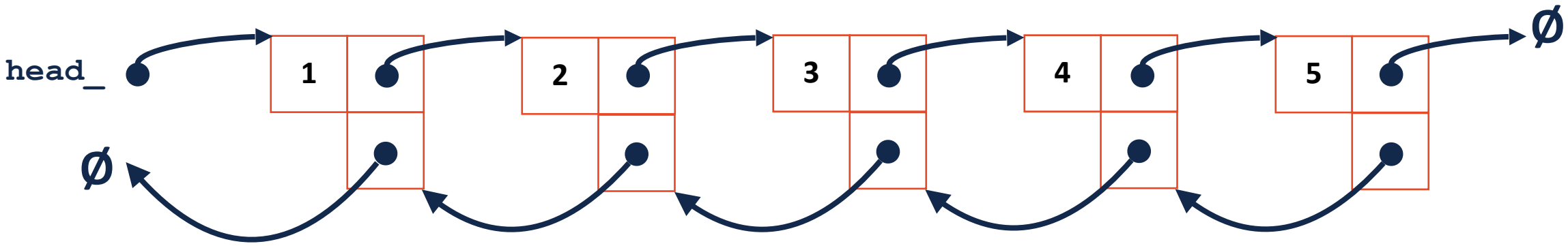
Thinking critically about lists: tradeoffs



Thinking critically about lists: tradeoffs



Thinking critically about lists: tradeoffs



Thinking critically about lists: tradeoffs

When we discuss data structures, consider how they can be modified or improved!

Next time: Can we make a 'list' that is O(1) to insert and remove? What is our tradeoff in doing so?