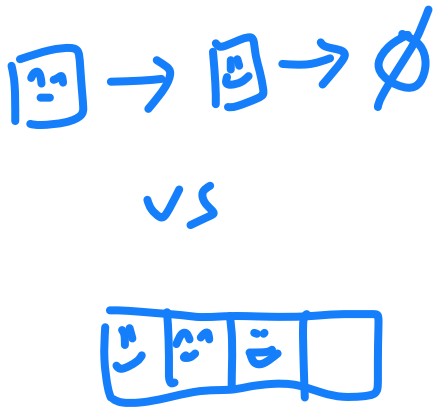
Data Structures

Array Lists

CS 225

Brad Solomon

September 4, 2026



Department of Computer Science

Post your art on #mp-art!

Learning Objectives

Discuss data variables for implementing array lists

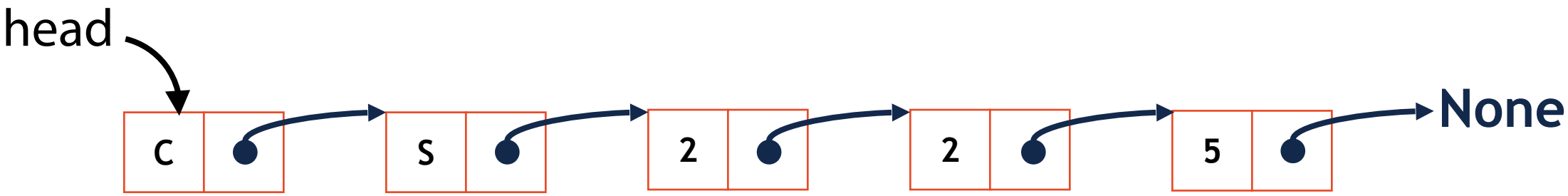
Review array list implementations

Discuss amortized analysis

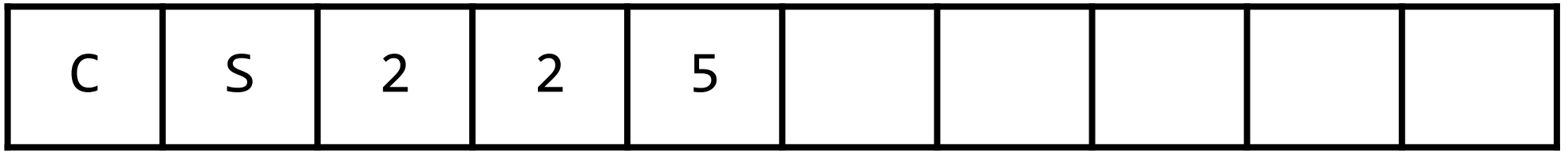
Consider extensions to lists

List Implementations

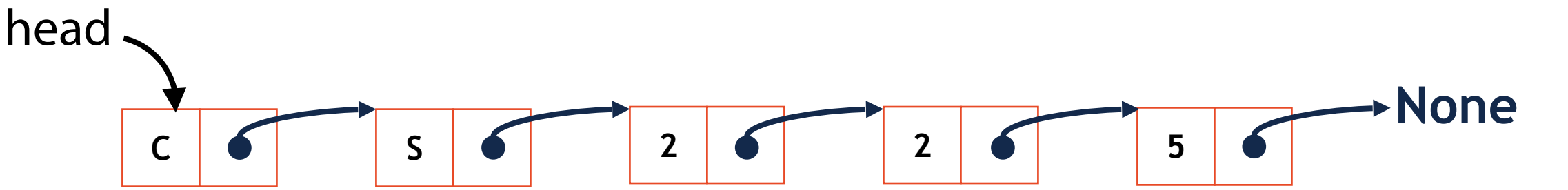
1. Linked List



2. Array List



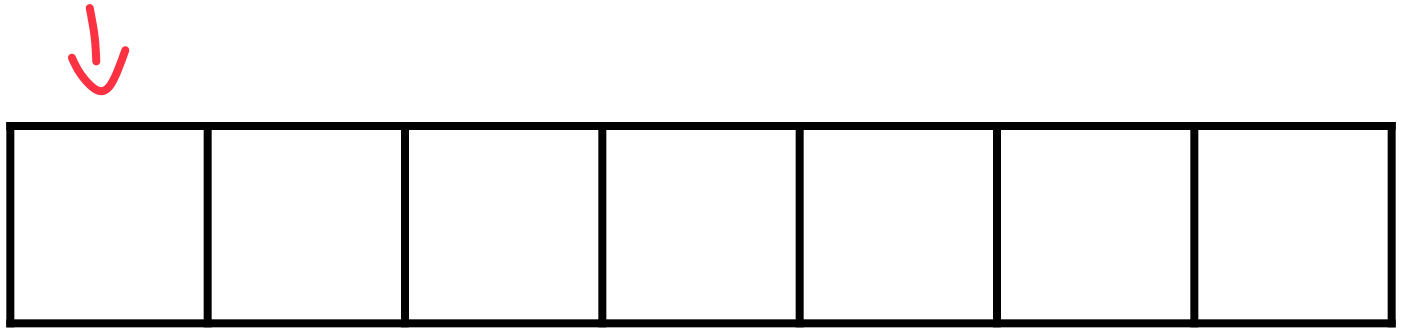
Linked List Runtimes



Random Access

	@Front	@RefPointer	@Index
			Some arbitrary index
Insert	O(1)	O(1)	O(n)
Delete	O(1)	O(1)	O(n)

Array List



An array is allocated as continuous memory.

Three values are necessary for efficient array usage:

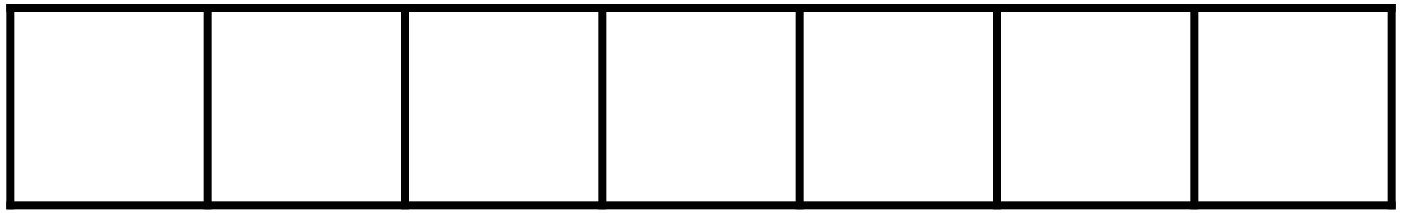
1) Start location of array

2) # of items in array (current) (size)

3) End of my allocated block (capacity / size limit)

4) size of item in array ⇔ homogenous (Data type)

Array List



An array is allocated as continuous memory.

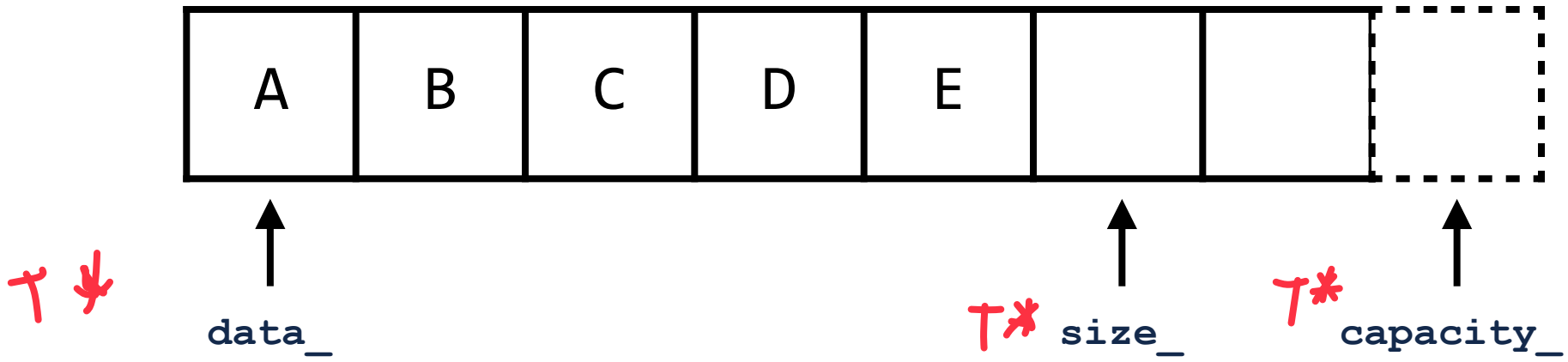
Three values are necessary for efficient array usage:

1) The start location of the array

2) The number of items currently stored in the array

3) The maximum capacity of the array

Array List



In C++, vector is implemented as:

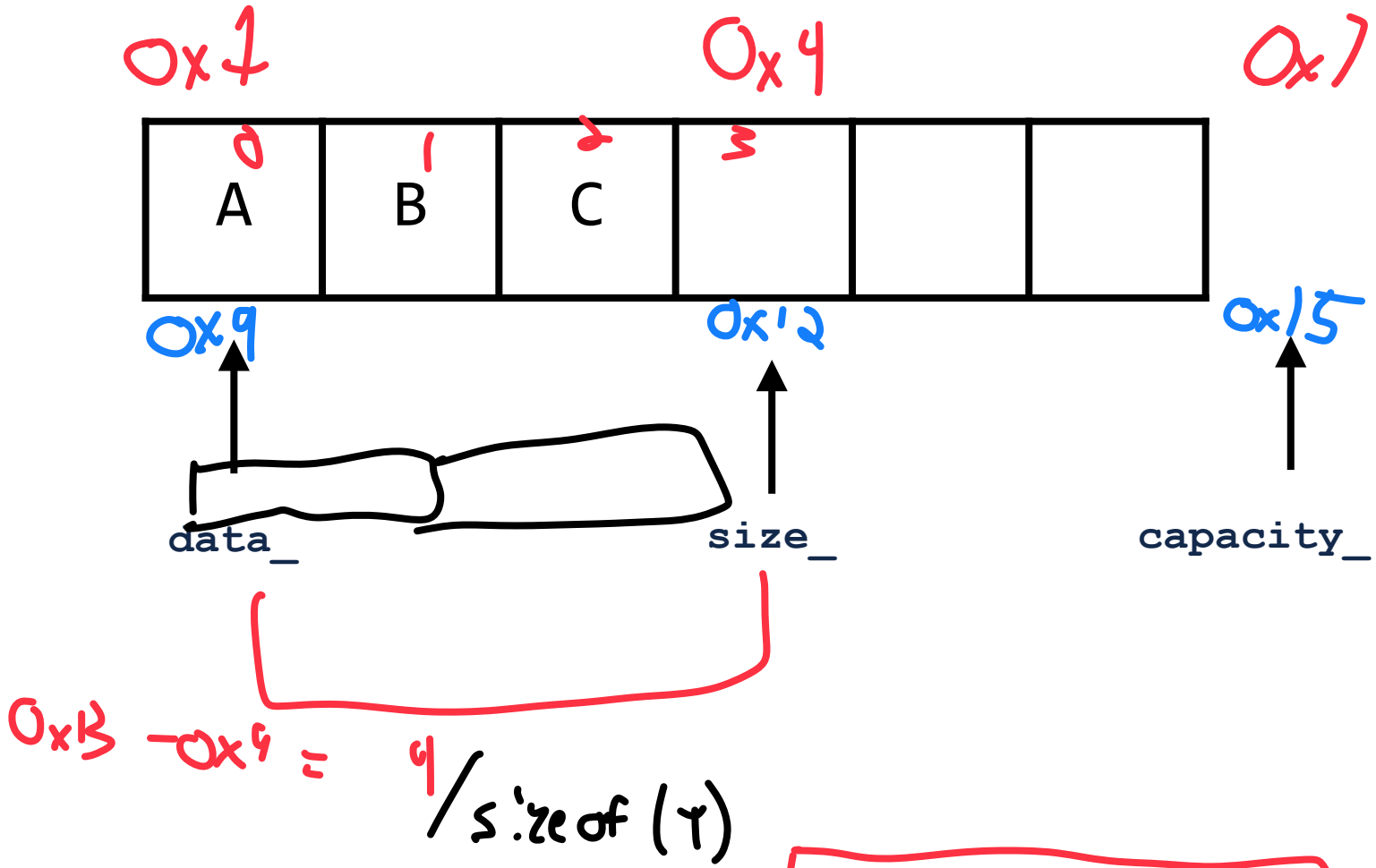
1) **Data:** Stored as a pointer to array start

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_;
…
        /* --- */
    };
```



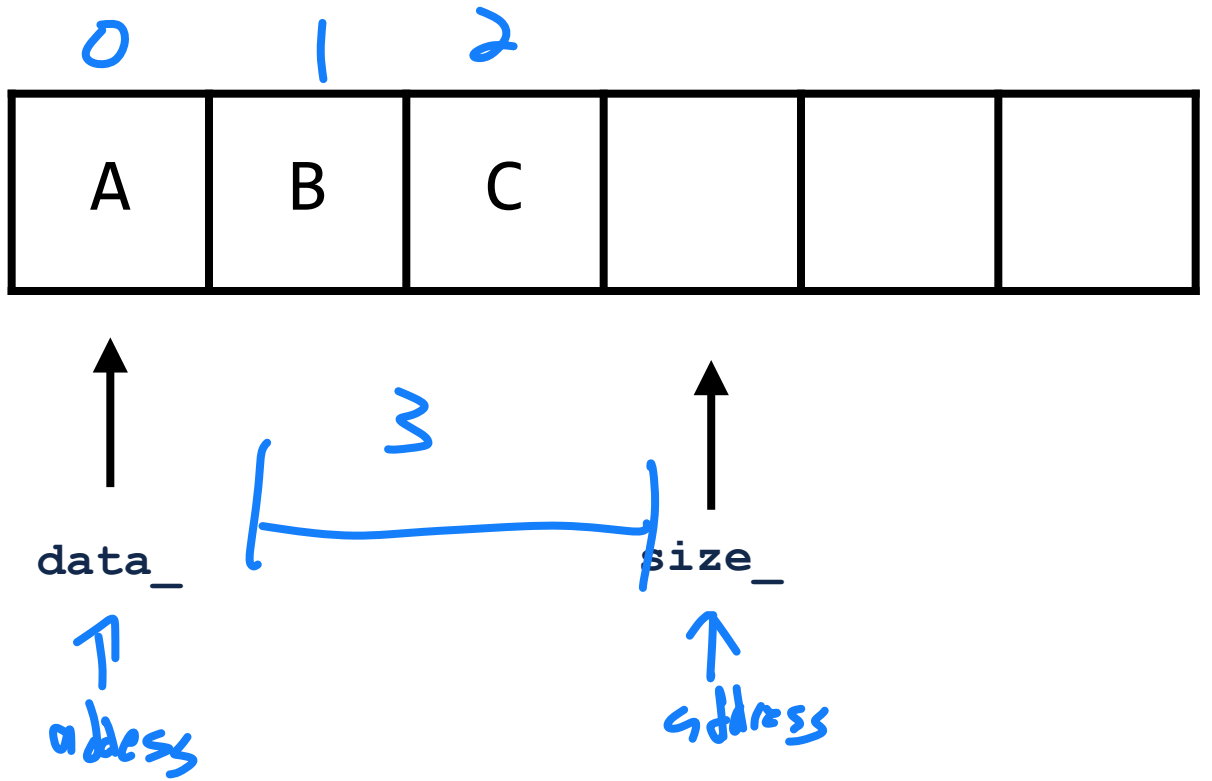
If I want to know the number of items in the array: size_ - data_

↳ sizeof(T) is needed?

(ft pointer math does this for you!

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_;
…
        /* --- */
    };
```



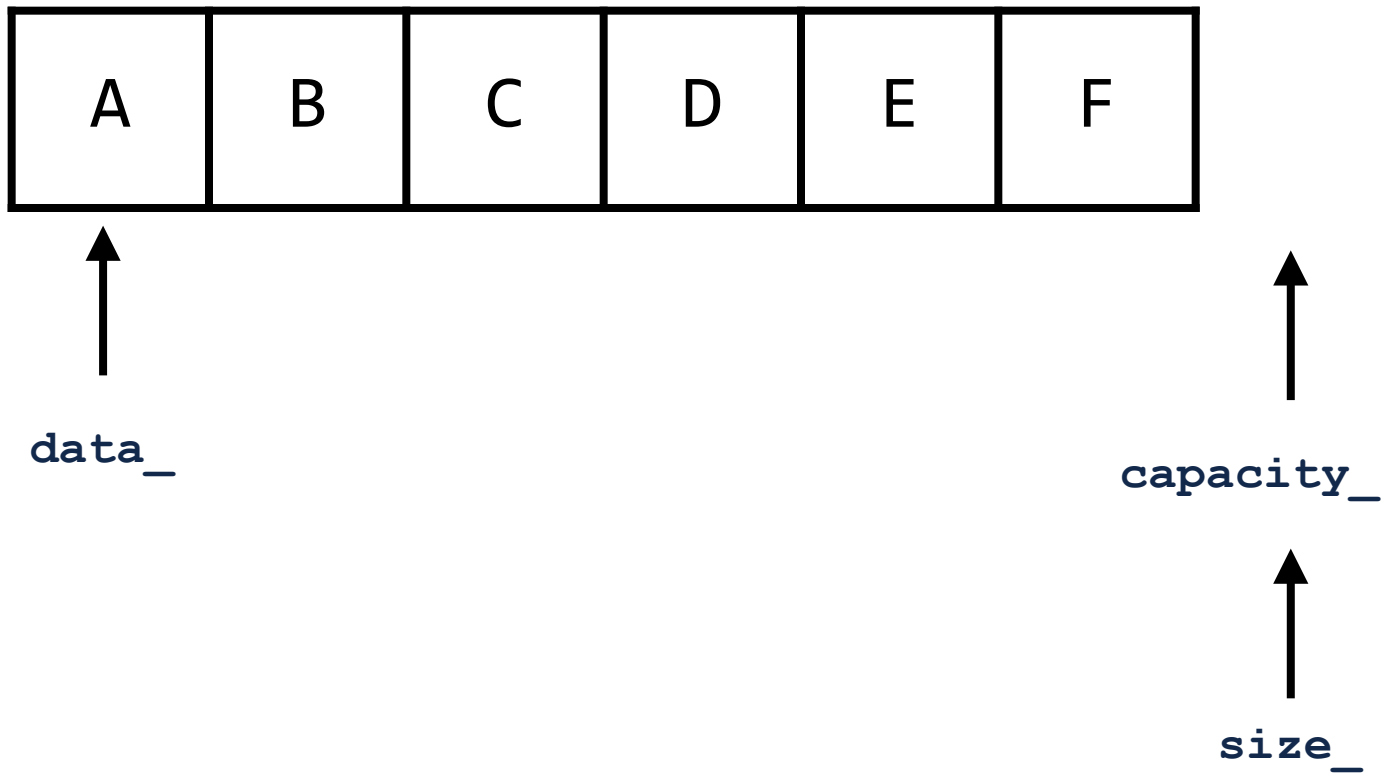
If I want to know the number of items in the array: **size_ - data_**

↳ why ptr (p is type T*) works!

p = p+1 ↳ sizeof(T)

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_;
 …
       /* --- */
   };
```

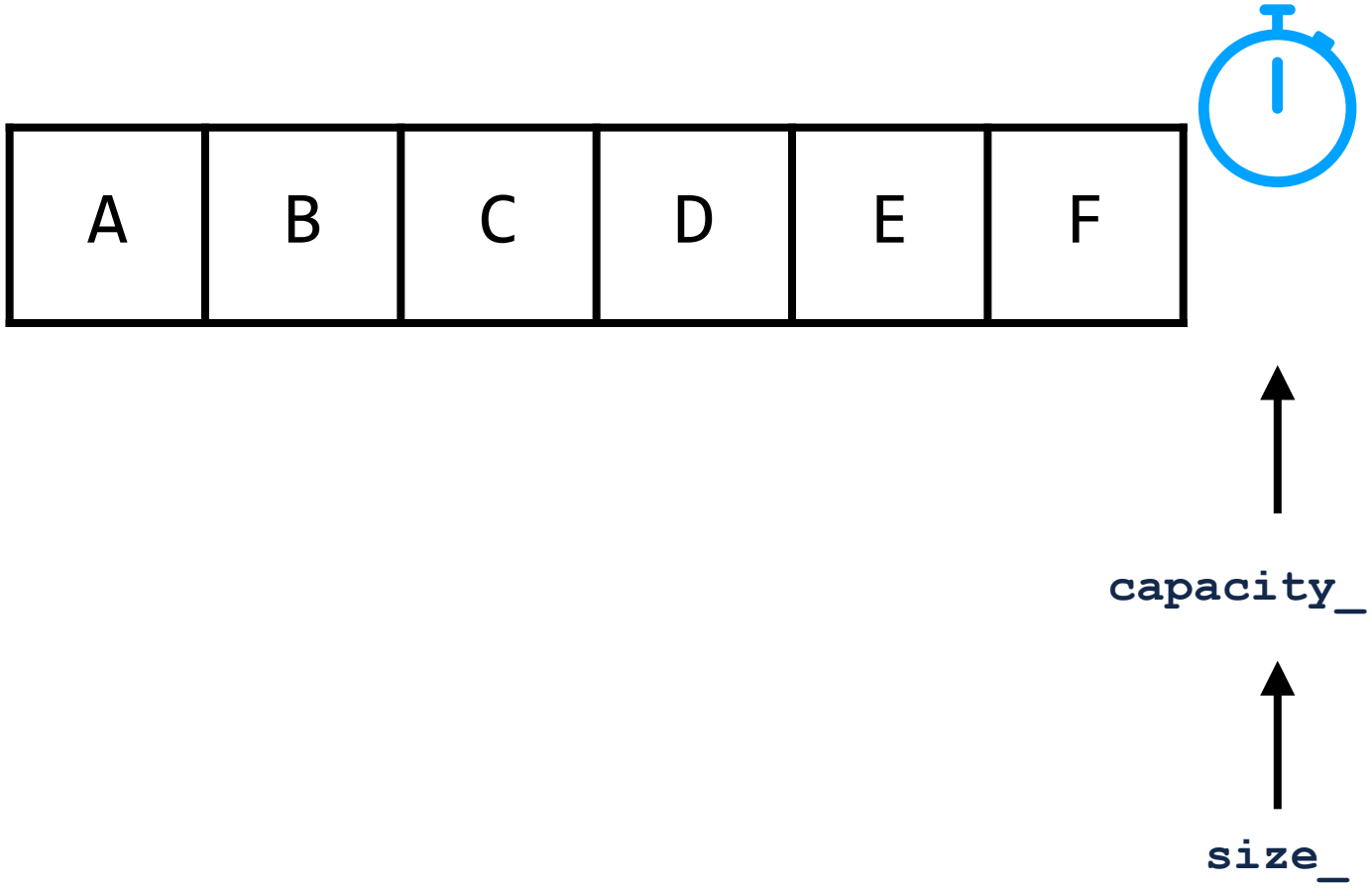


How do I know if I'm at capacity? capacity_ - size_ = 0

capacity_ == size_

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_;
 …
        /* --- */
    };
```



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

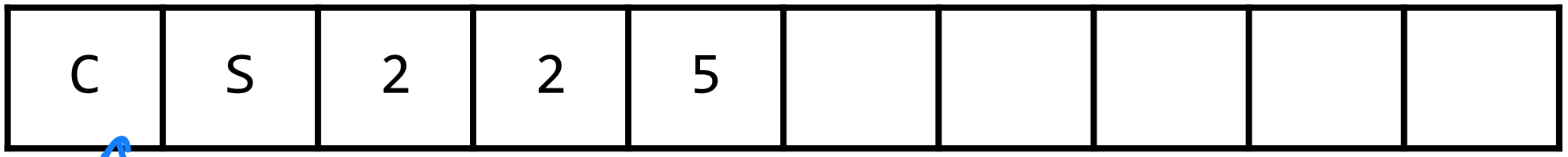
Array List: [ ]

L[i]

i = 2

L[i] = 10

x = L[i]



↳ data_

↳ size_

↳ capacity_

data

2 away

return type is T*

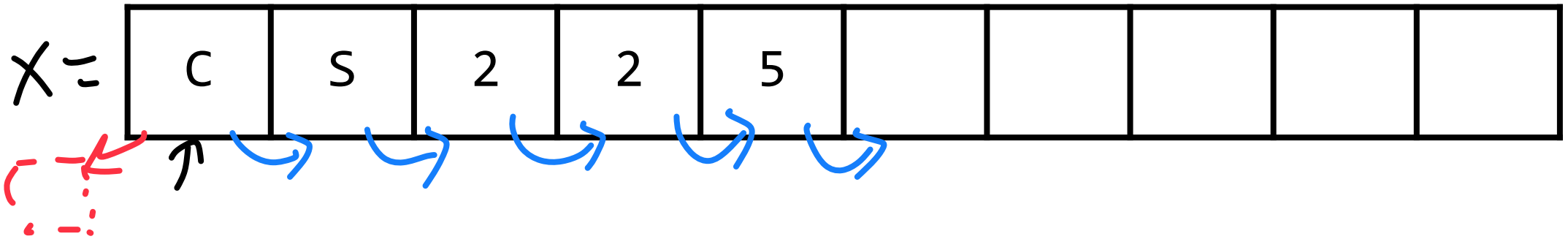
return data_ + i;

if return type T or T&

↳ *(data_ + i)

Random access! O(1)

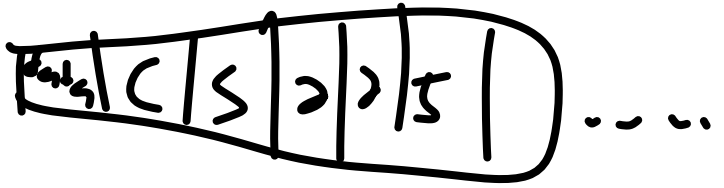
Array List: `insertFront(data)`



1) Move everything over by one address O(n)

2) Add data to index 0 O(1)

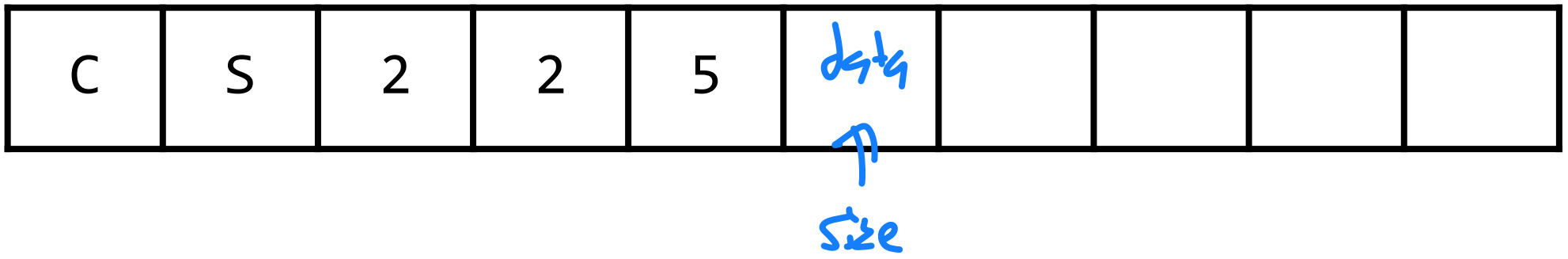
O(n) overall



Why not claim mem address data -1?

↳ We don't own and cannot request memory before or after array

Array List: `insertBack(data)`



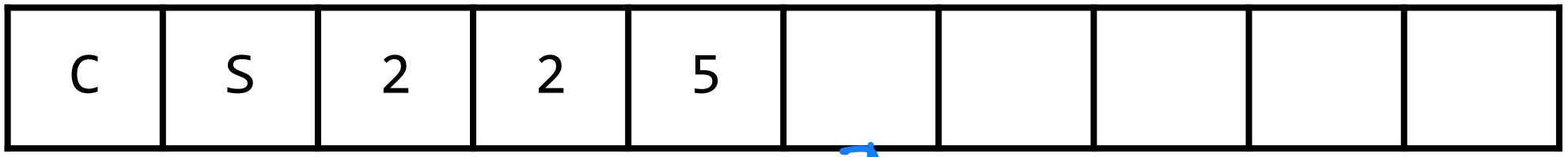
1) Check if size == capacity → resize Then insert

2) size points to next available space!

↳ *size_ = data;

3) size++;

Array List: `insert(data, index)`



insert front

O(n)

insert Back

O(1) * not at capacity

insert(data, 0)

O(n)

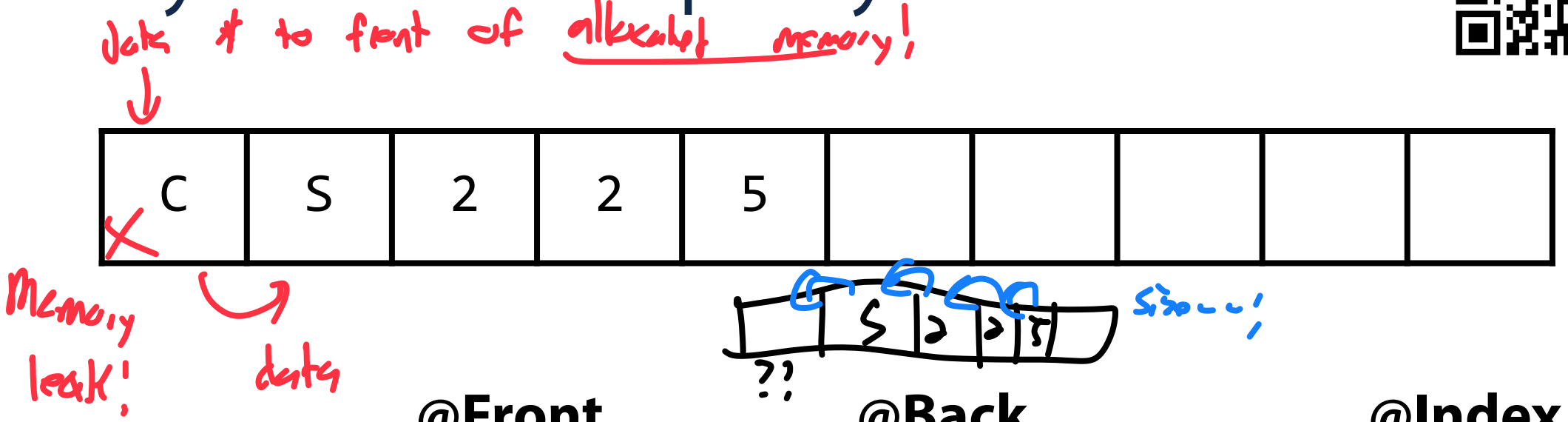
1) Shift all values at & after index

2) Insert data @ index

3) size++;

Array List: Not at capacity

Join Code: 225



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

Array List: addspace(data)

N	O	S	P	A	C	E

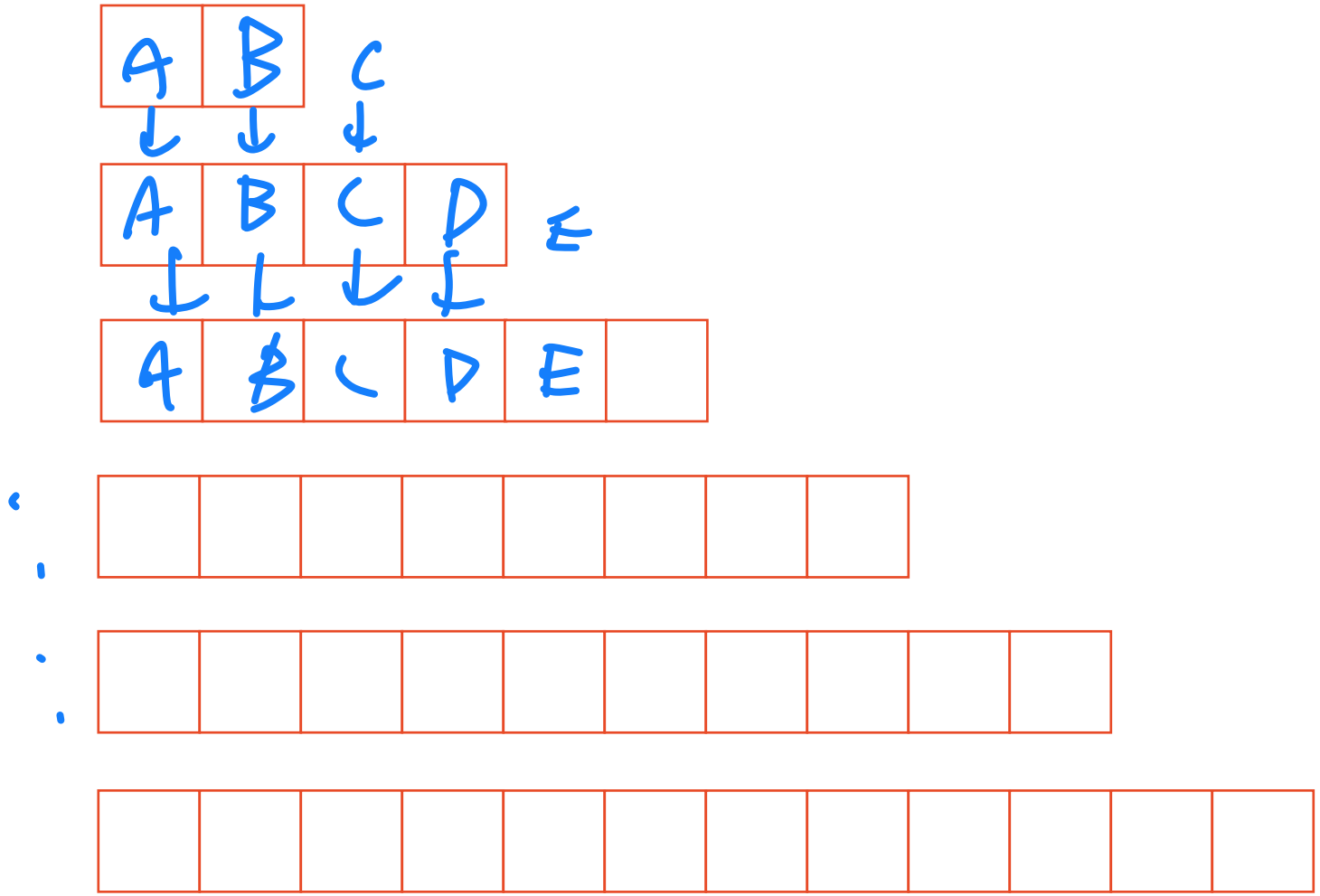
size == capacity

We cannot assume ownership of next Mem address!

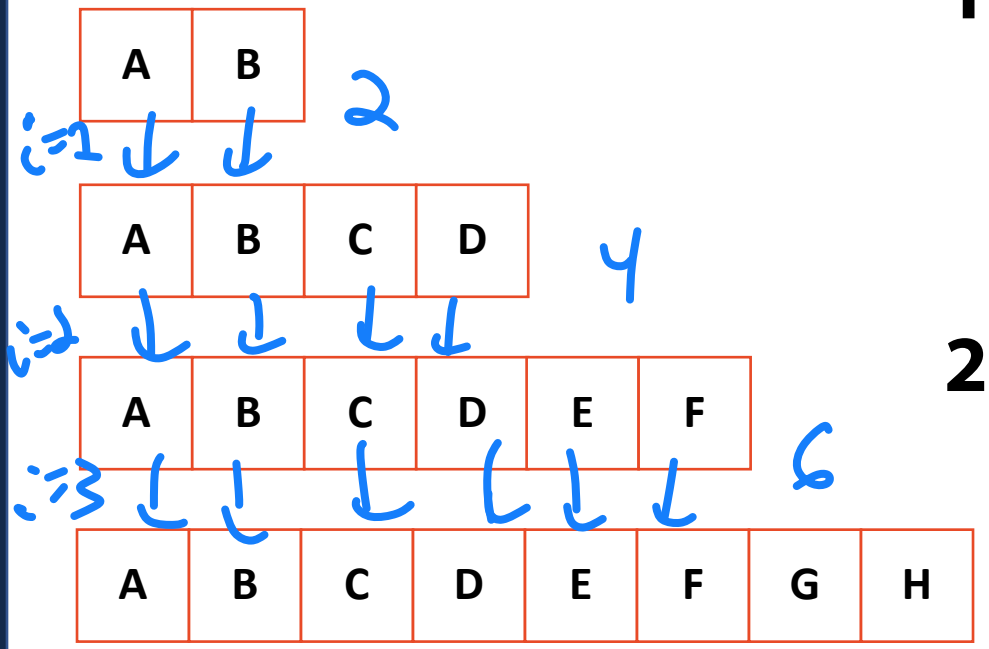
↳ We Must allocate new list

↳ and copy items!

Resize Strategy: +2 elements every time



Resize Strategy: +2 elements every time



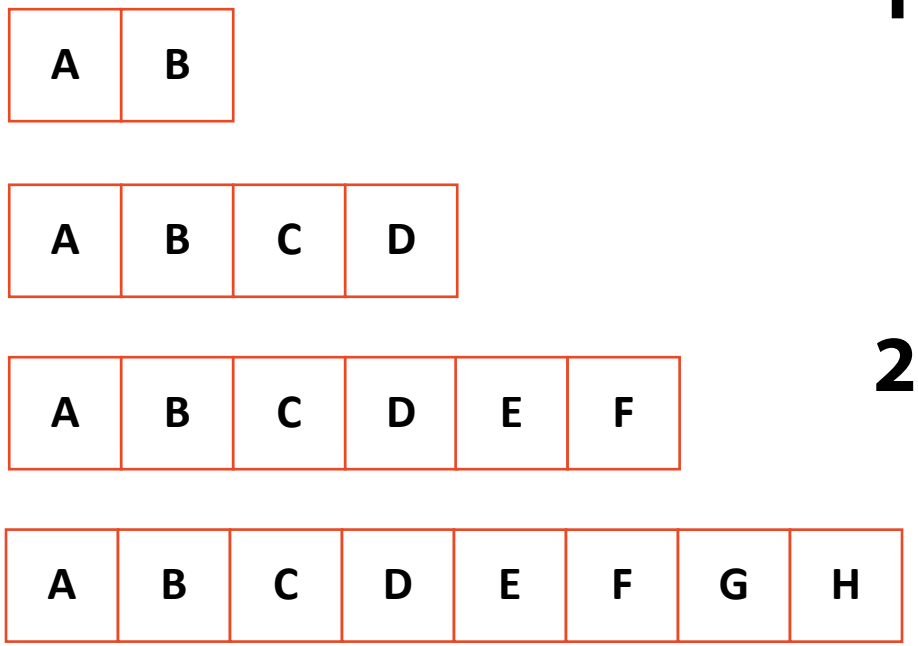
1) How many copy calls per reallocation?

for all i, 2^i copies

2) Total reallocations for N objects?

N/2 times to add total of N capacity

Resize Strategy: +2 elements every time



1) How many copy calls per reallocation?

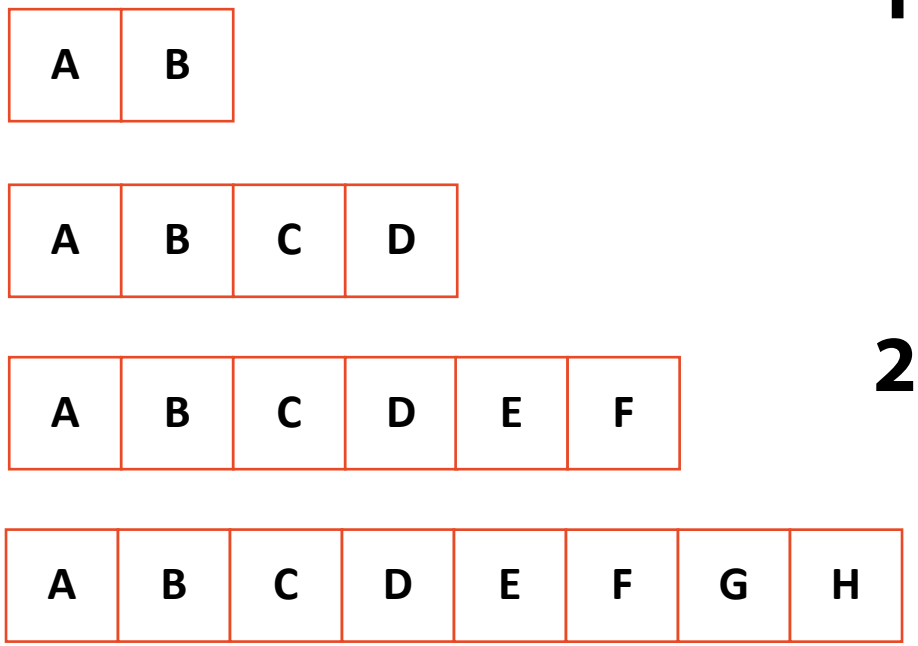
For reallocation i, 2i copy calls are made

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, $k = \frac{N}{2}$

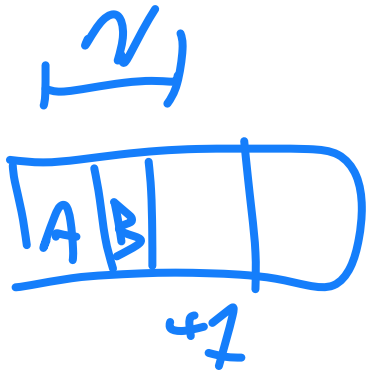
Total number of copy calls: $\sum_{i=1}^{k} 2i = k(k+1) = k^2 + k$

… For N objects: $\frac{N^2 + 2N}{4}$

Total # of operations done to add N items

Resize Strategy: +2 elements every time

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



This is Bad!

Amortized: $\approx N$

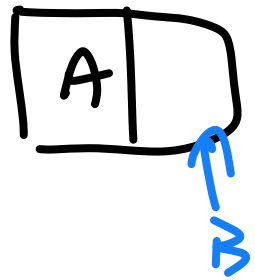
Look at total runtime / # of inserts

"Average performance over N inserts"

$\frac{N^2+2N}{4} / N \approx N$ ops per insert

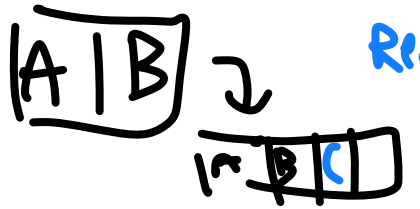
Big O: $O(n)$

2 types of insertions

1) 

Open position at end of array

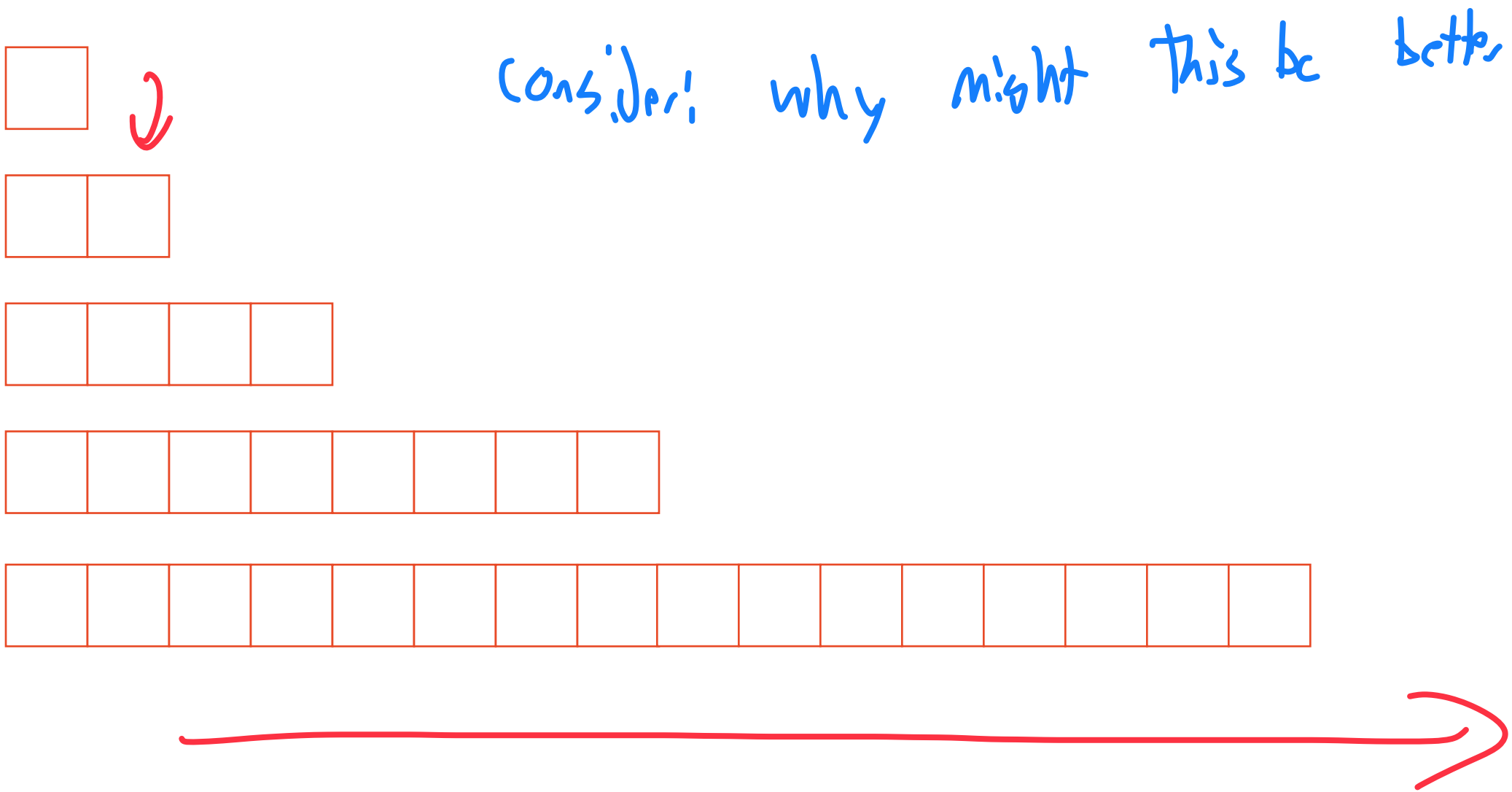
$O(1)$ add to ending

2) 

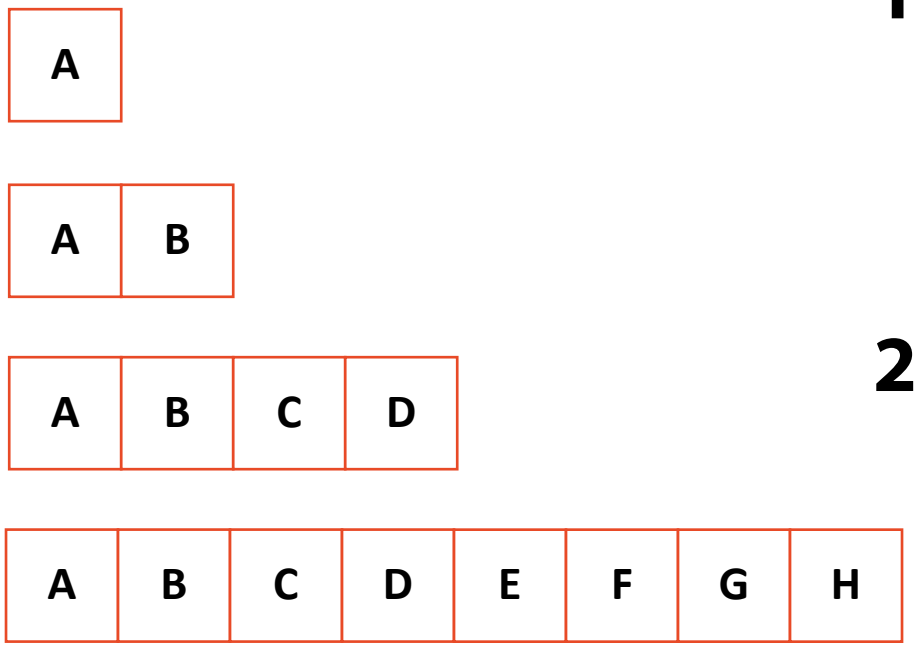
Reallocation insert

$O(n)$ has to copy all the values

Resize Strategy: x2 elements every time



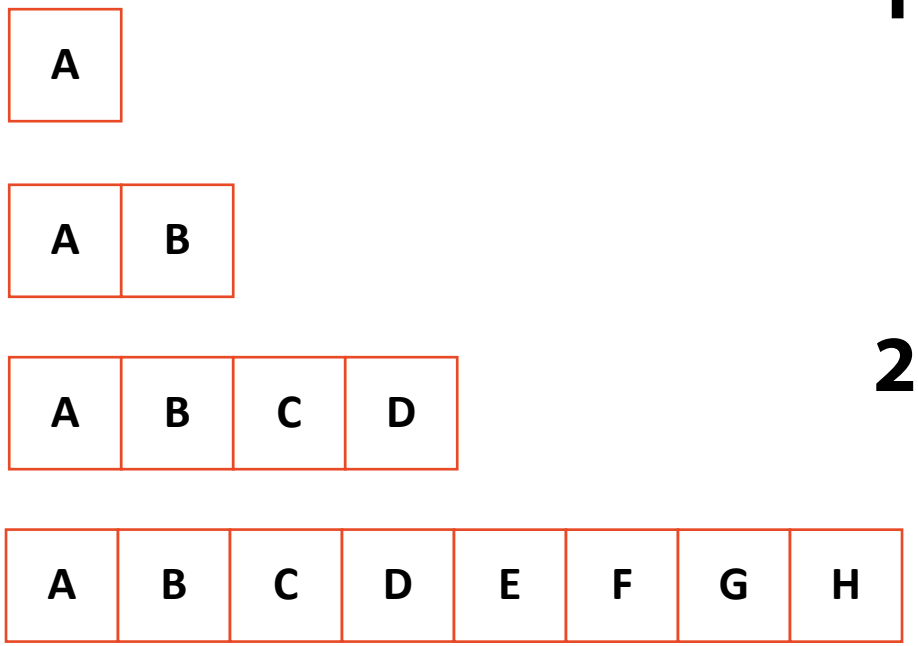
Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

2) Total reallocations for N objects?

Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

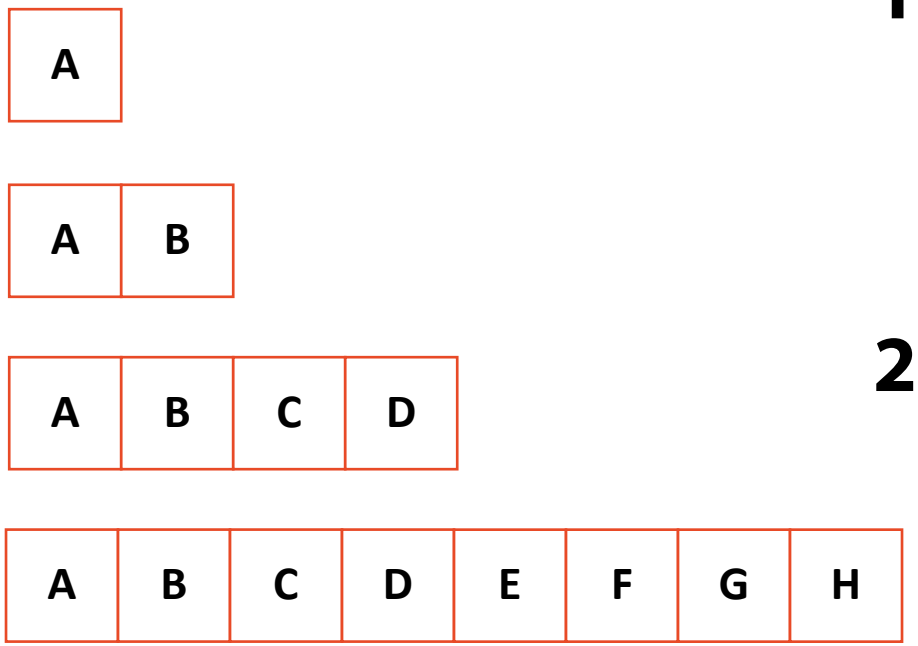
For reallocation i, 2^i copy calls are made

2) Total reallocations for N objects?

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

Total number of copy calls:

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:

Resize Strategy: x2 elements every time

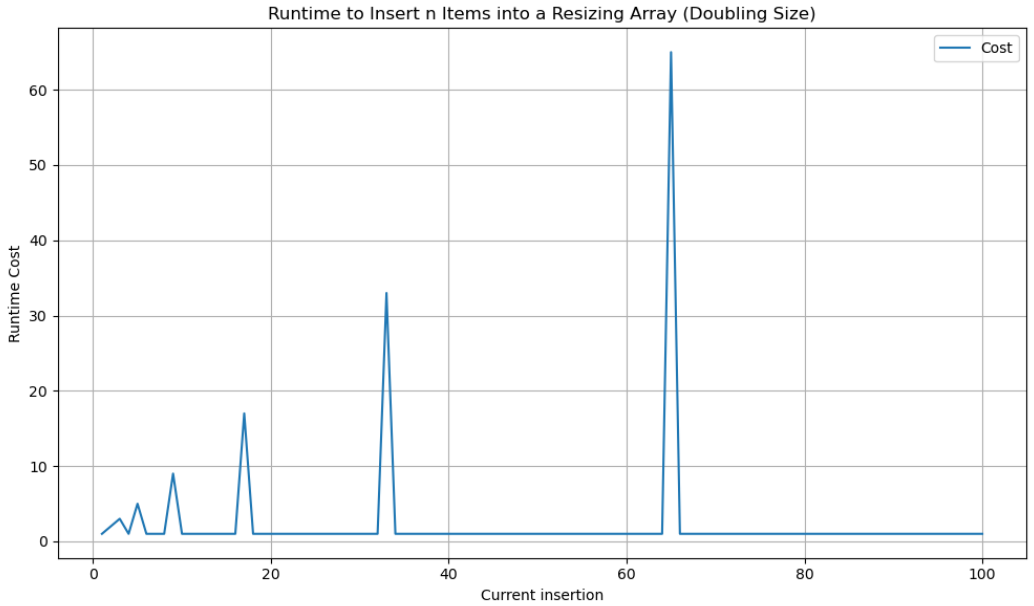
Total copies for n inserts: $2N - 1$

Amortized:

Precise total work over N calls

Big O:

Upperbound on worst case



Distinguishing amortized vs Big O

1) Every insert operation has a Big O of O(N)

2) By totaling up the total work over N inserts, we see a different story:

- The 'bad' insert steps happen roughly once every n inserts

Ex: n = 2, 4, 8, 16, 32, …

- The amount of work in the bad insert also grows linearly

- All other inserts are 'good' inserts each taking O(1) time

When viewed in the lens of 'total work over N inserts' we get O(1) *

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**		

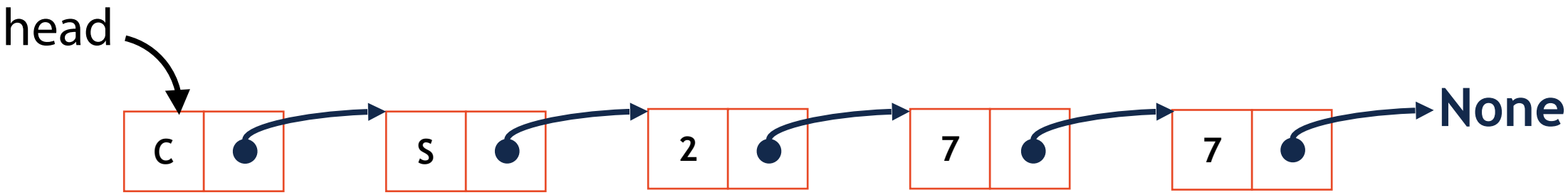
Thinking critically about lists: tradeoffs

The implementations shown are foundational (simple).

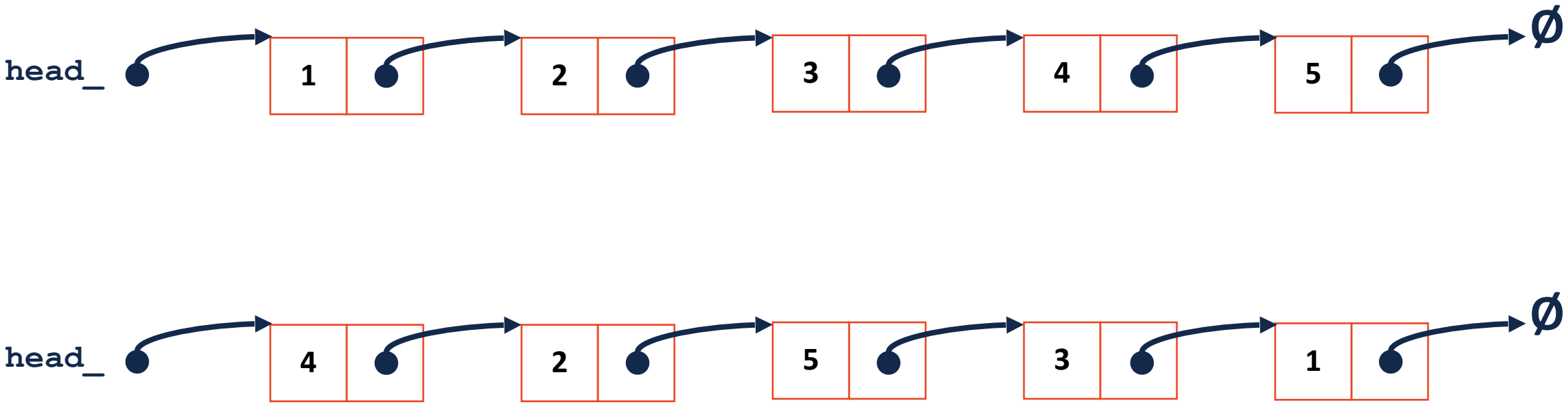
Can we make our lists better at some things? What is the cost?

Thinking critically about lists: tradeoffs

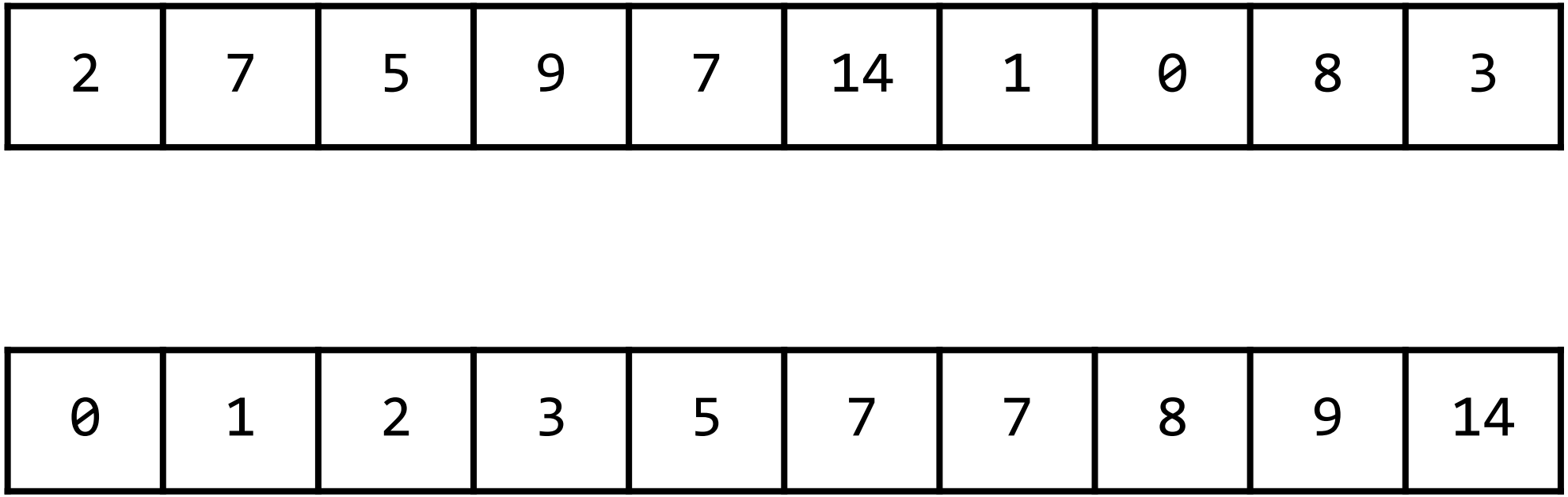
Getting the size of a linked list has a Big O of:



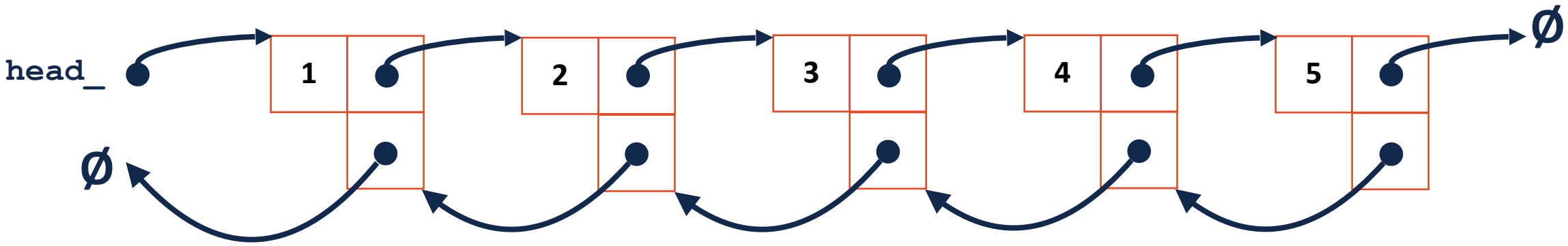
Thinking critically about lists: tradeoffs



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?