Data Structures

Array Lists

CS 225

Brad Solomon

February 2, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Course Policy Reminders

Please use the extension request form

Please email CS 225 admin

Post your art on #mp-art!

Learning Objectives

Discuss data variables for implementing array lists

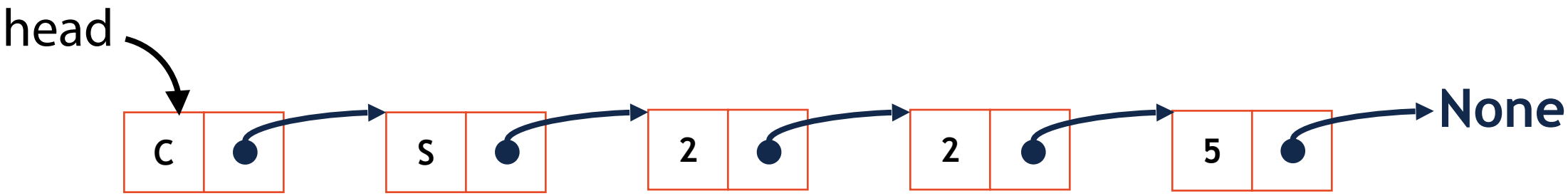
Review array list implementations

Discuss amortized analysis

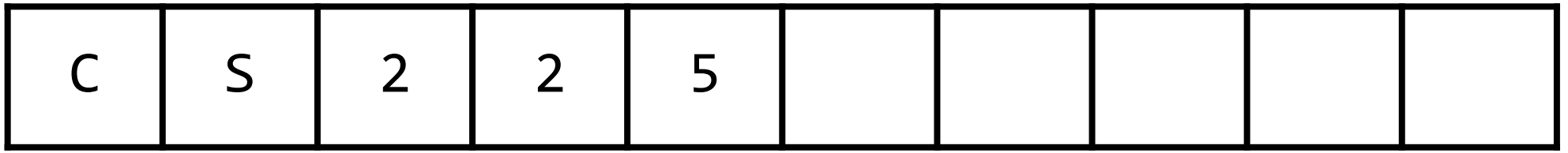
Consider extensions to lists

List Implementations

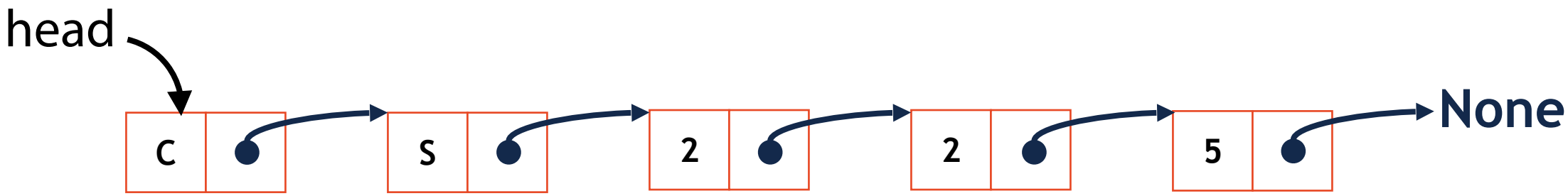
1. Linked List



2. Array List

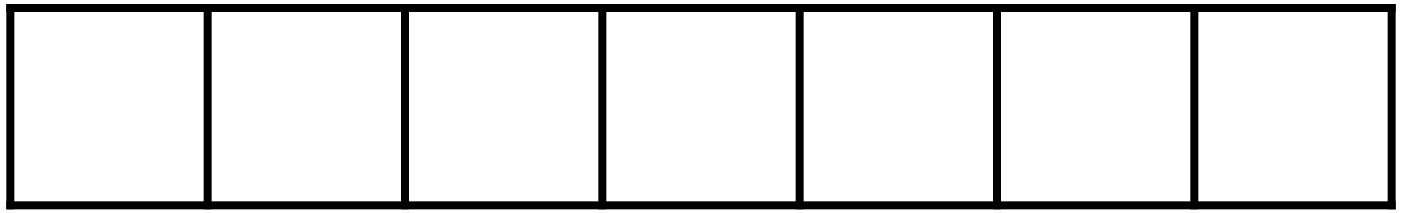


Linked List Runtimes



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

Array List



An array is allocated as continuous memory.

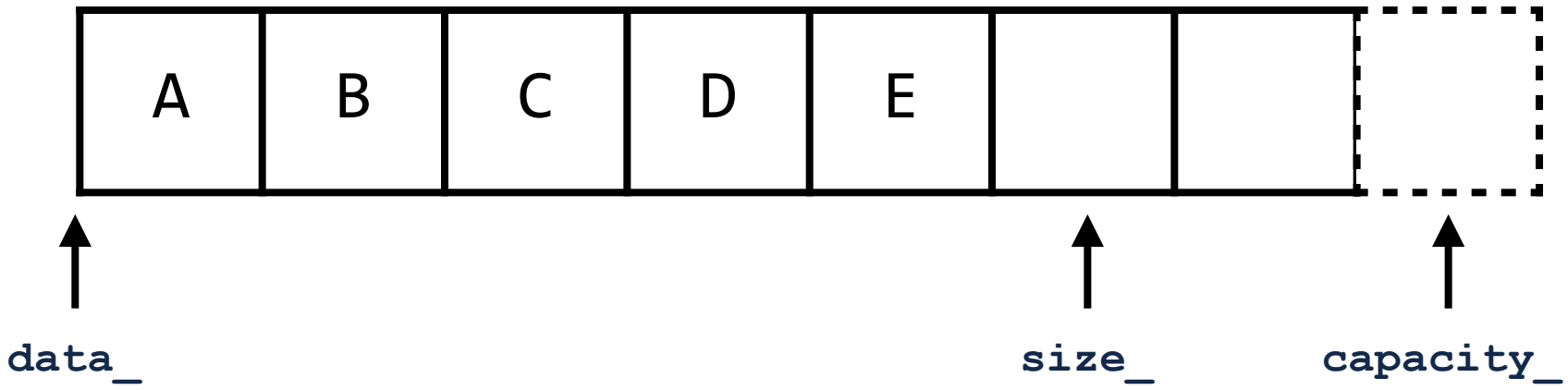
Three values are necessary for efficient array usage:

1)

2)

3)

Array List



In C++, vector is implemented as:

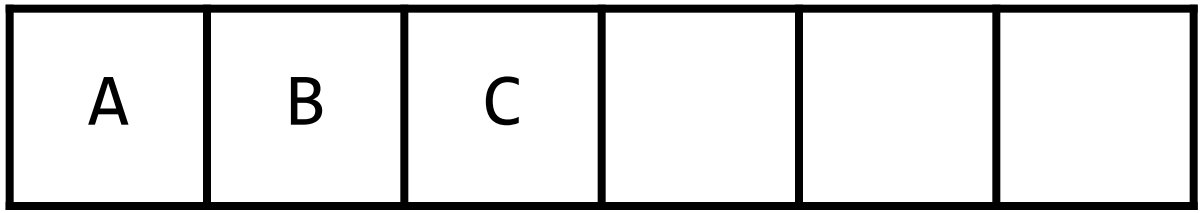
1) **Data:** Stored as a pointer to array start

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

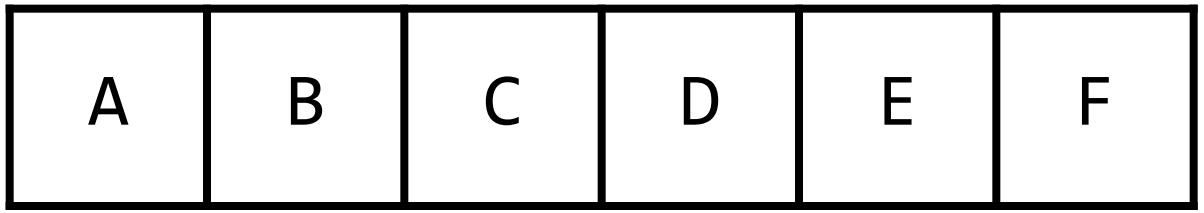
```
 1  #pragma once
 2
 3  template <typename T>
 4  class List {
 5  public:
…       /* --- */
25  private:
26    T *data_;
27
28    T *size_;
29
30    T *capacity_;
…
        /* --- */
    };
```



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

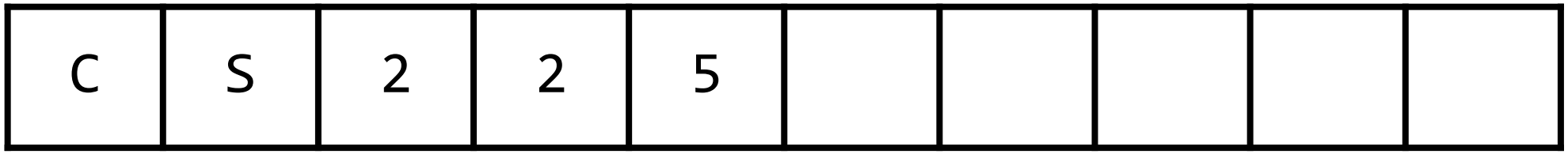
List.h

```
 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_;
 …
        /* --- */
    };
```

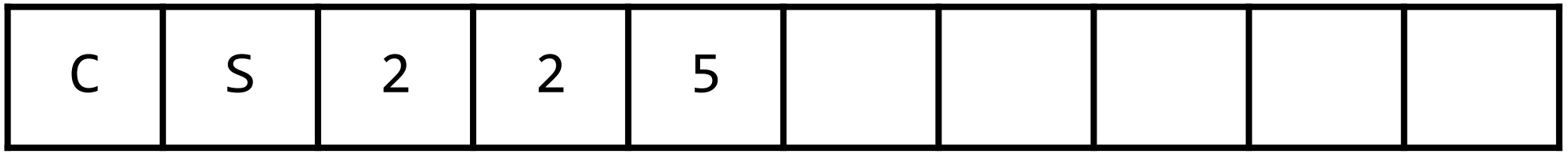


How do I know if I'm at capacity?

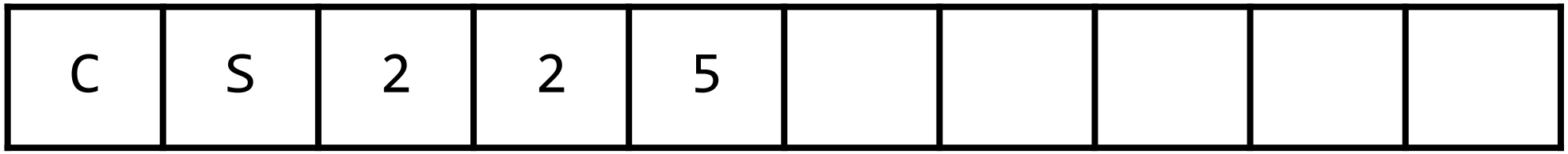
Array List: [ ]



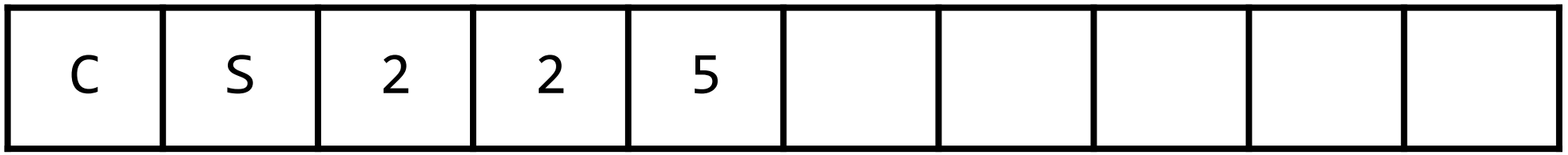
Array List: `insertFront(data)`



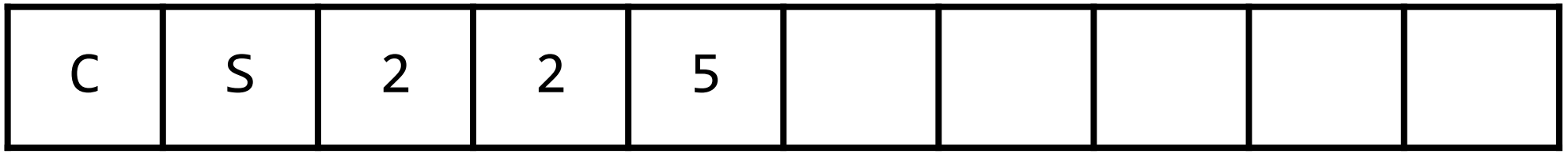
Array List: `insertBack(data)`



Array List: `insert(data, index)`



Array List: Not at capacity

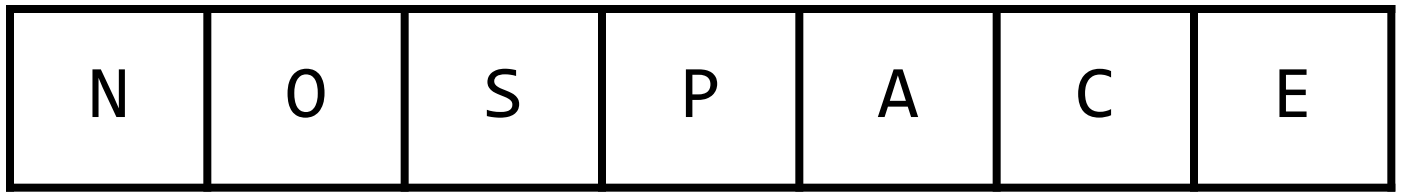


@Front **@Back** **@Index**

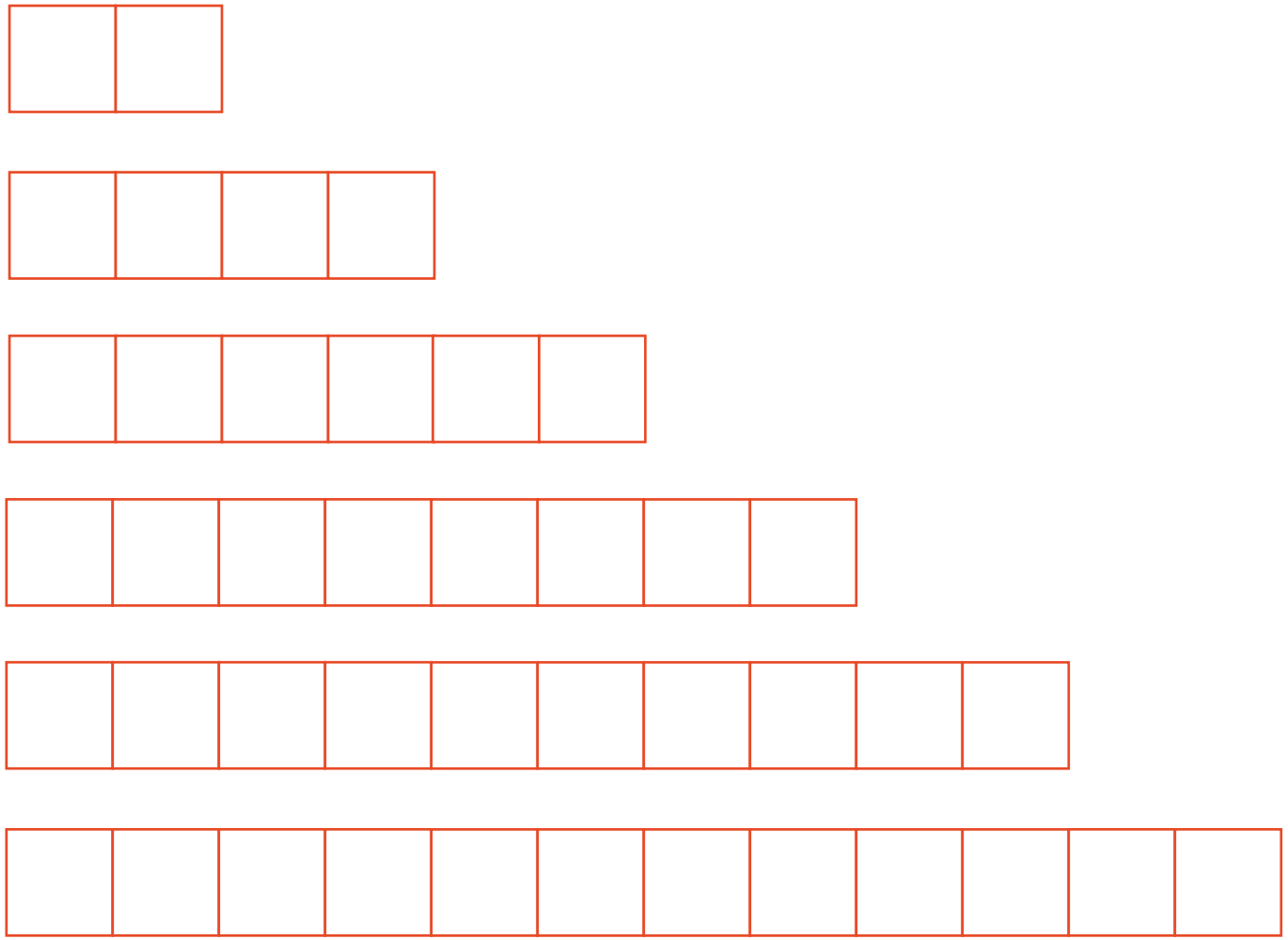
Insert

Delete

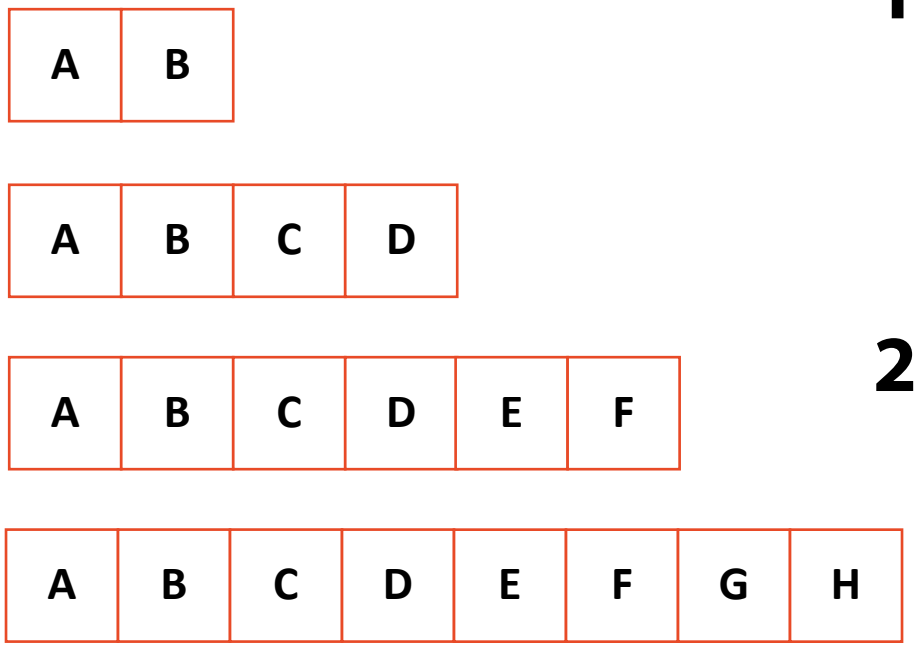
Array List: addspace(data)



Resize Strategy: +2 elements every time



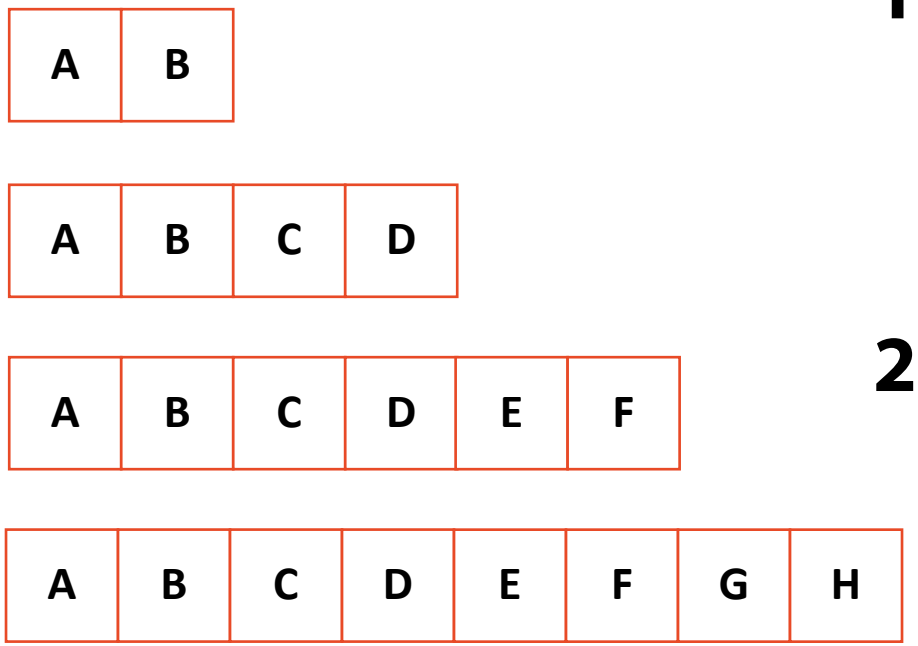
Resize Strategy: +2 elements every time



1) How many copy calls per reallocation?

2) Total reallocations for N objects?

Resize Strategy: +2 elements every time



1) How many copy calls per reallocation?

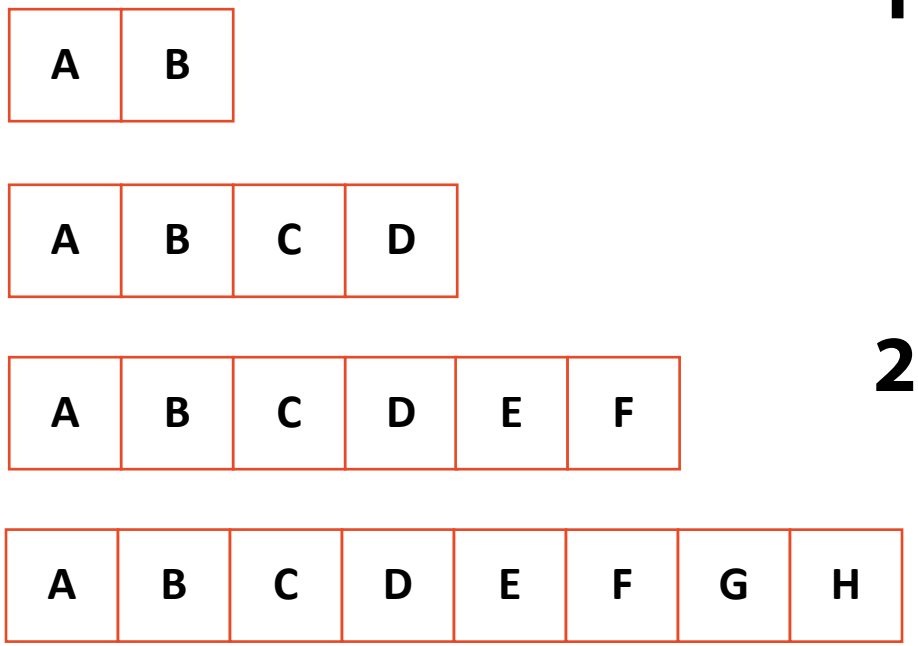
For reallocation i, 2i copy calls are made

2) Total reallocations for N objects?

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

Total number of copy calls:

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: $\frac{N^2 + 2N}{4}$

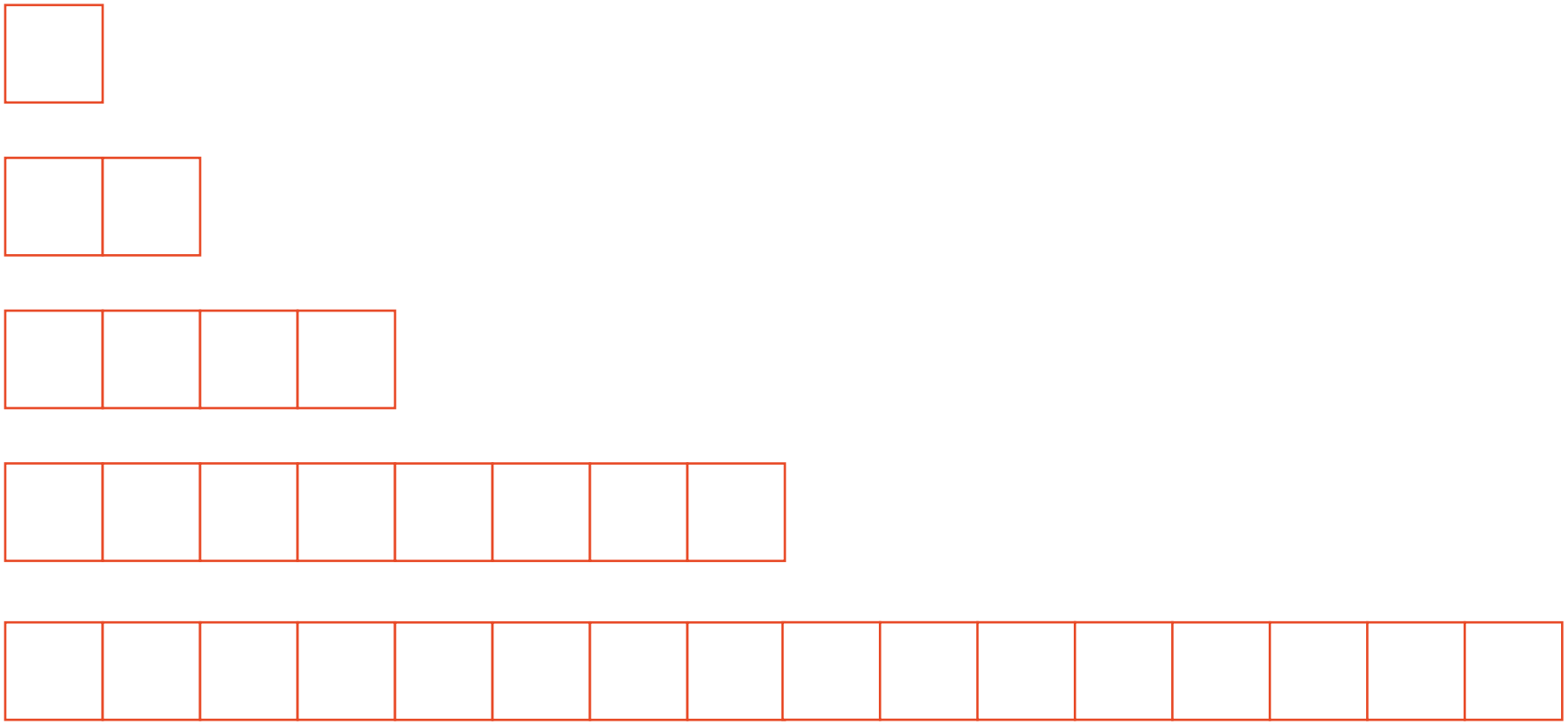
Resize Strategy: +2 elements every time

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

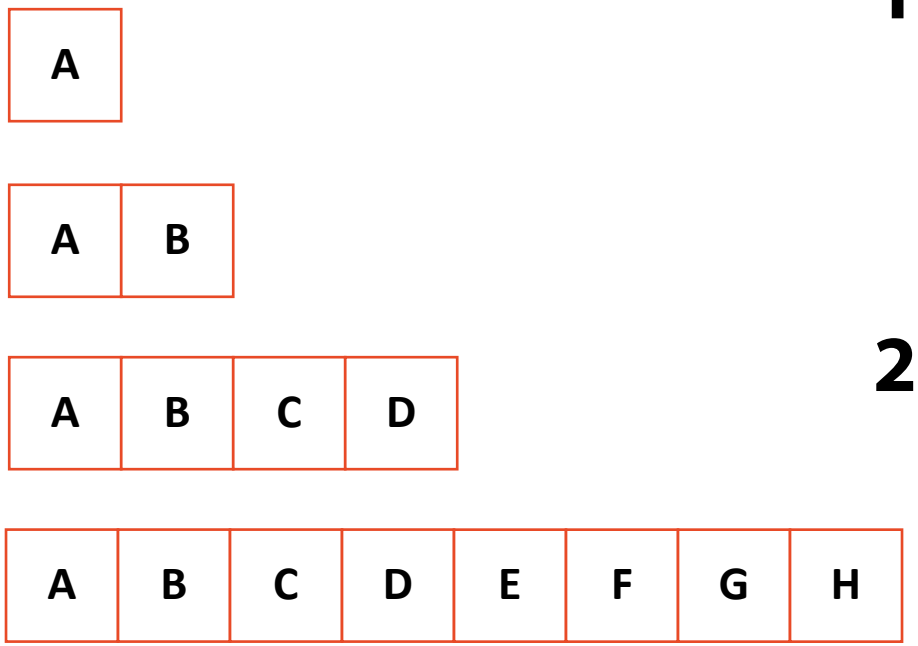
Amortized:

Big O:

Resize Strategy: x2 elements every time



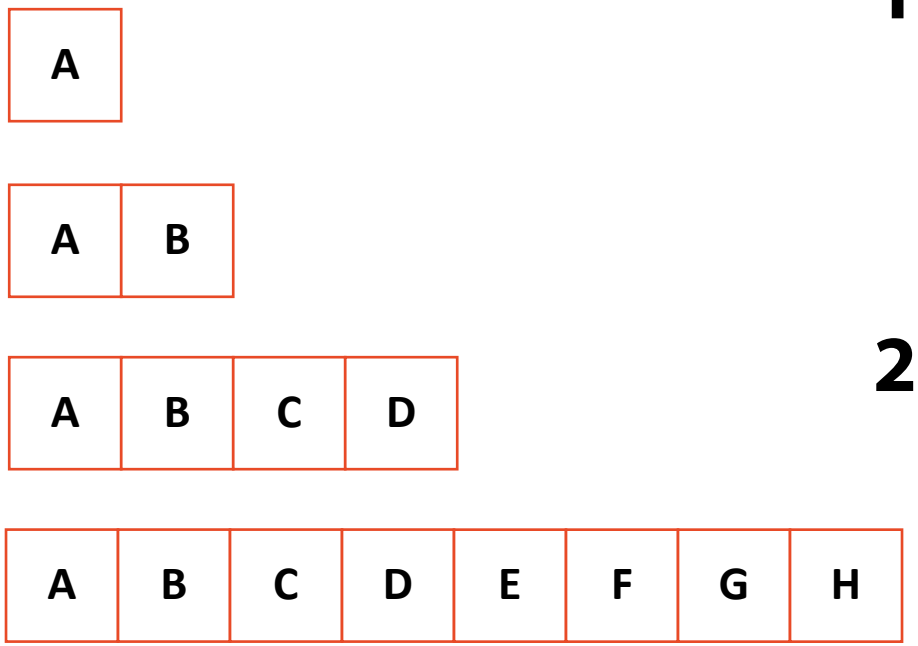
Resize Strategy: x2 elements every time



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?

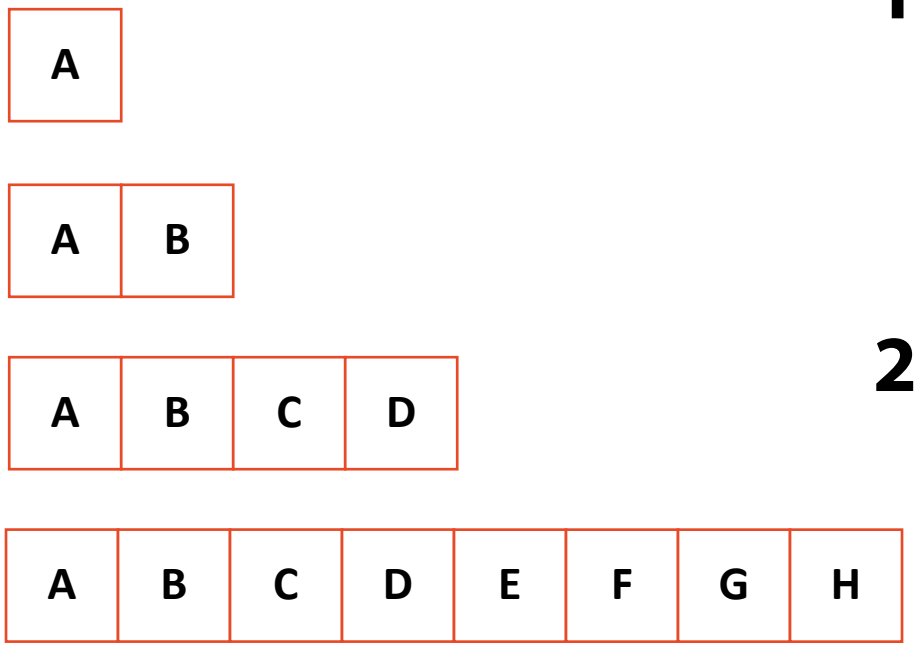
For reallocation i, 2^i copy calls are made

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

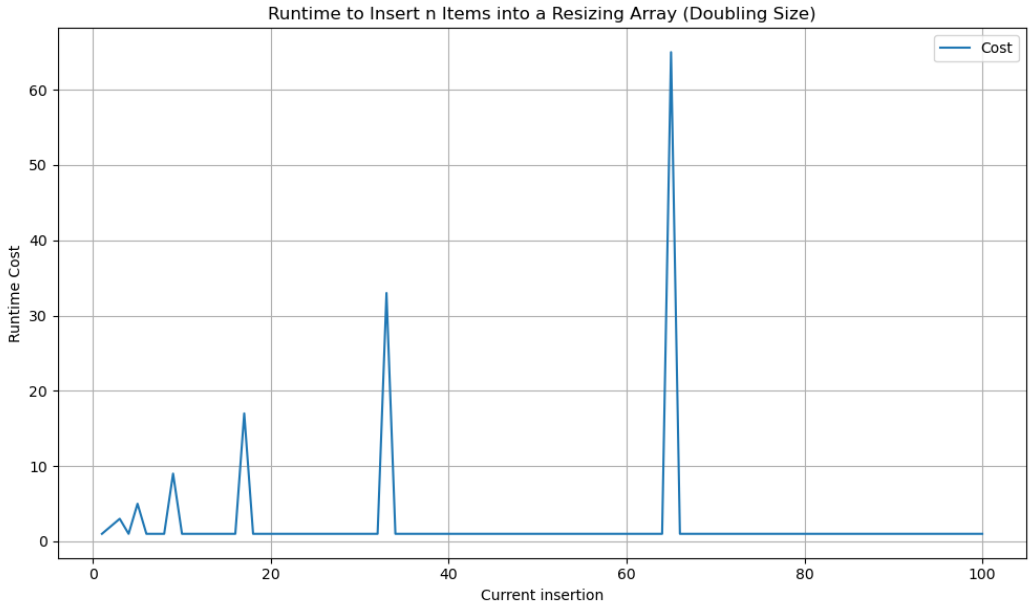
Total copies for n inserts: $2n - 1$

Amortized:

Precise total work over N calls

Big O:

Upperbound on worst case



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**		

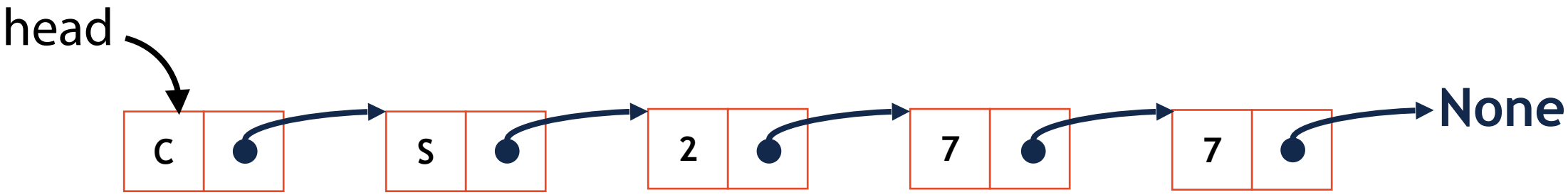
Thinking critically about lists: tradeoffs

The implementations shown are foundational (simple).

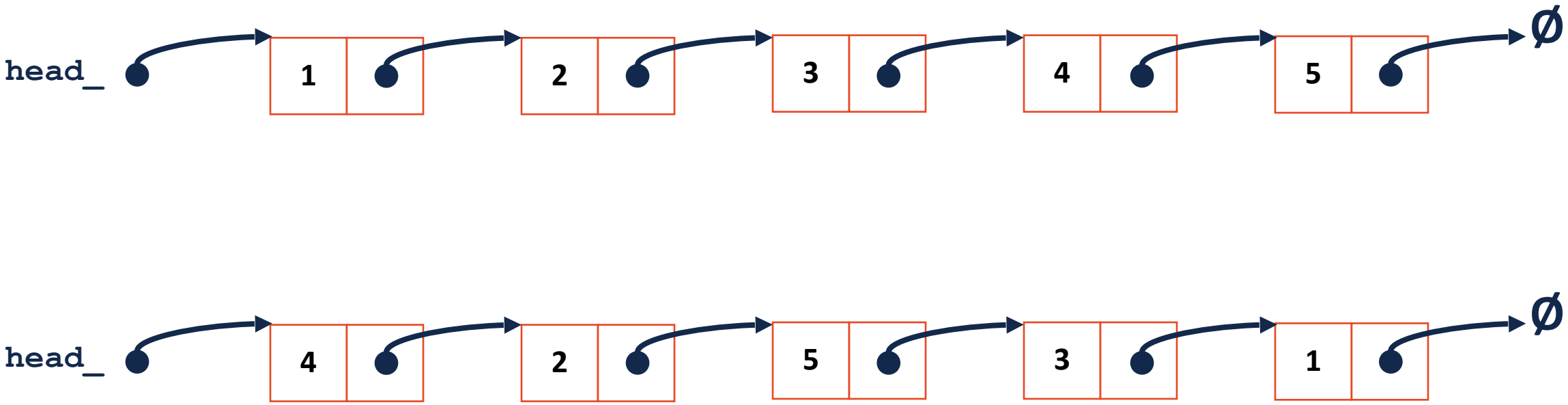
Can we make our lists better at some things? What is the cost?

Thinking critically about lists: tradeoffs

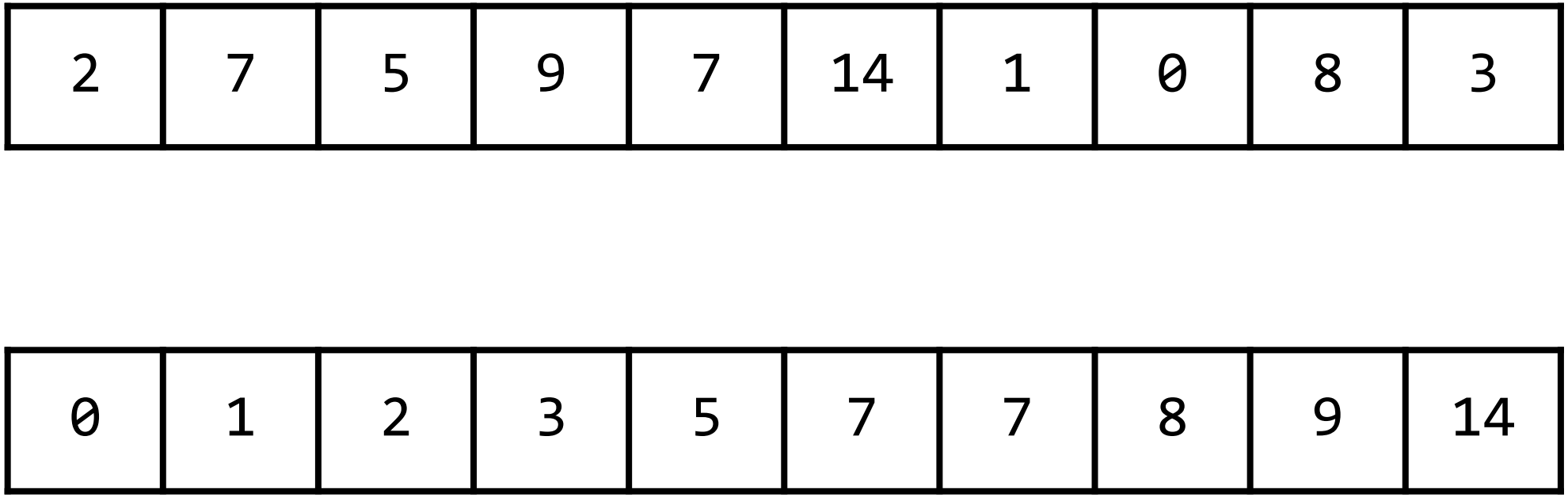
Getting the size of a linked list has a Big O of:



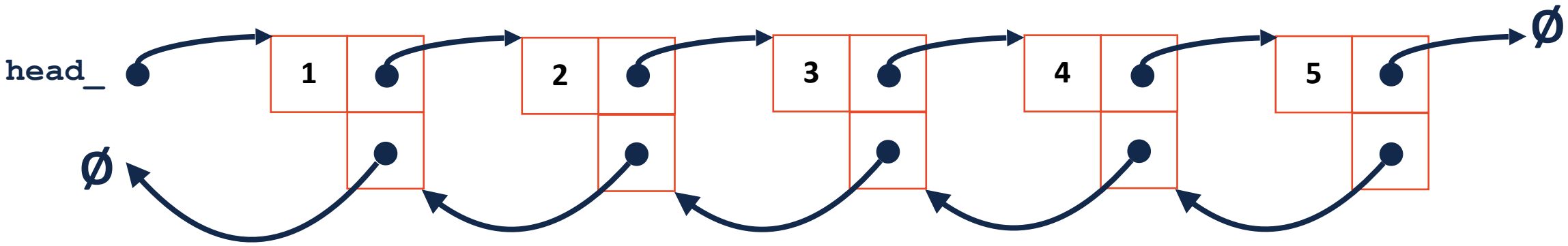
Thinking critically about lists: tradeoffs



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?