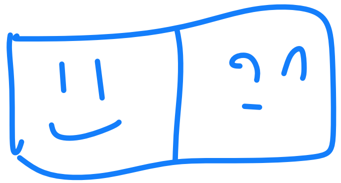
Data Structures

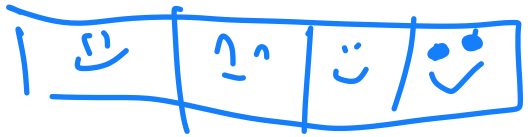
Array Lists

CS 225

Brad Solomon

February 2, 2026





UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Exam 1 (2/09 — 2/11)

Content through today

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam will be released on PL

Topics covered can be found on website

Register now

https://courses.engr.illinois.edu/cs225/exams/

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

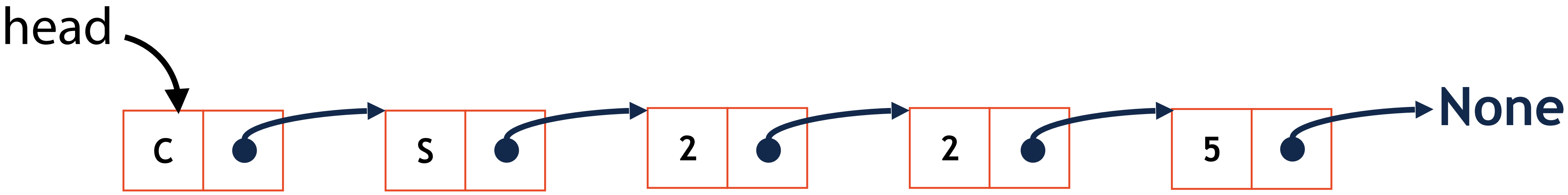
Review array list implementations

Discuss amortized analysis

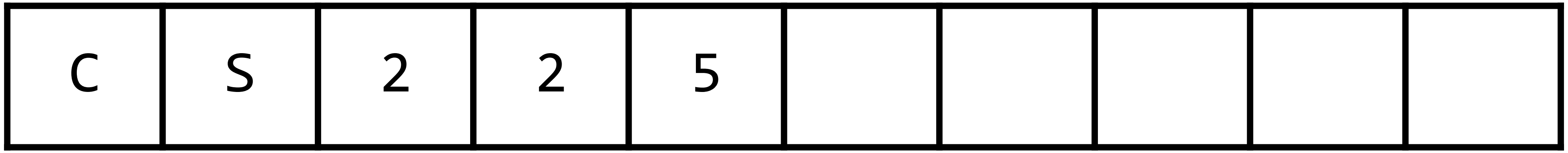
Consider extensions to lists

List Implementations

1. Linked List

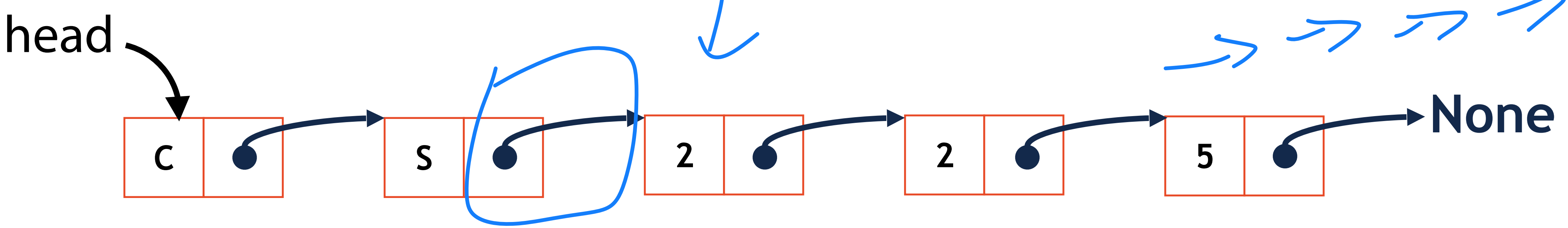


2. Array List



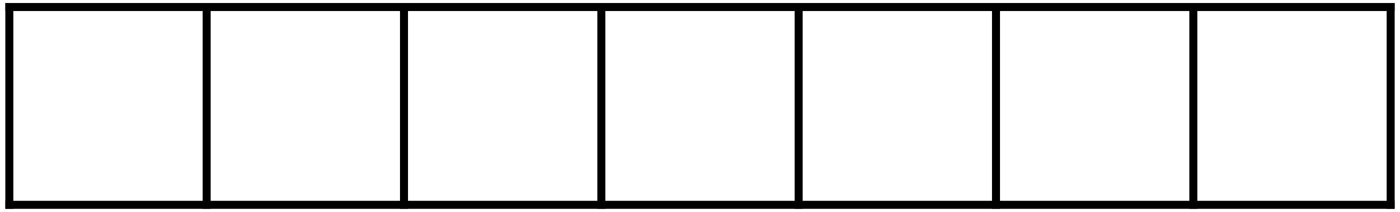
Linked List Runtimes

100th item



	@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.

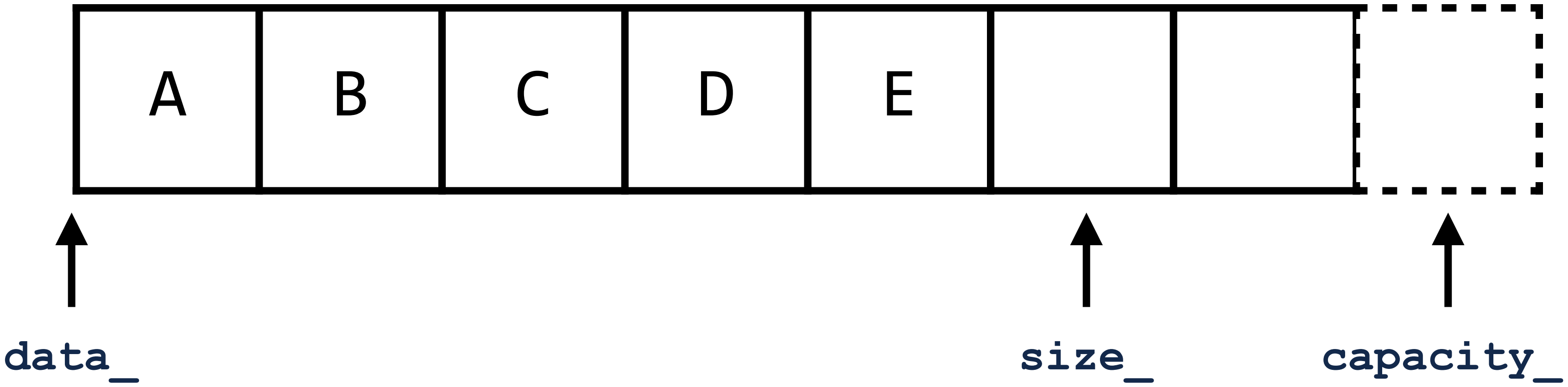
Three values are necessary for efficient array usage:

1) 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

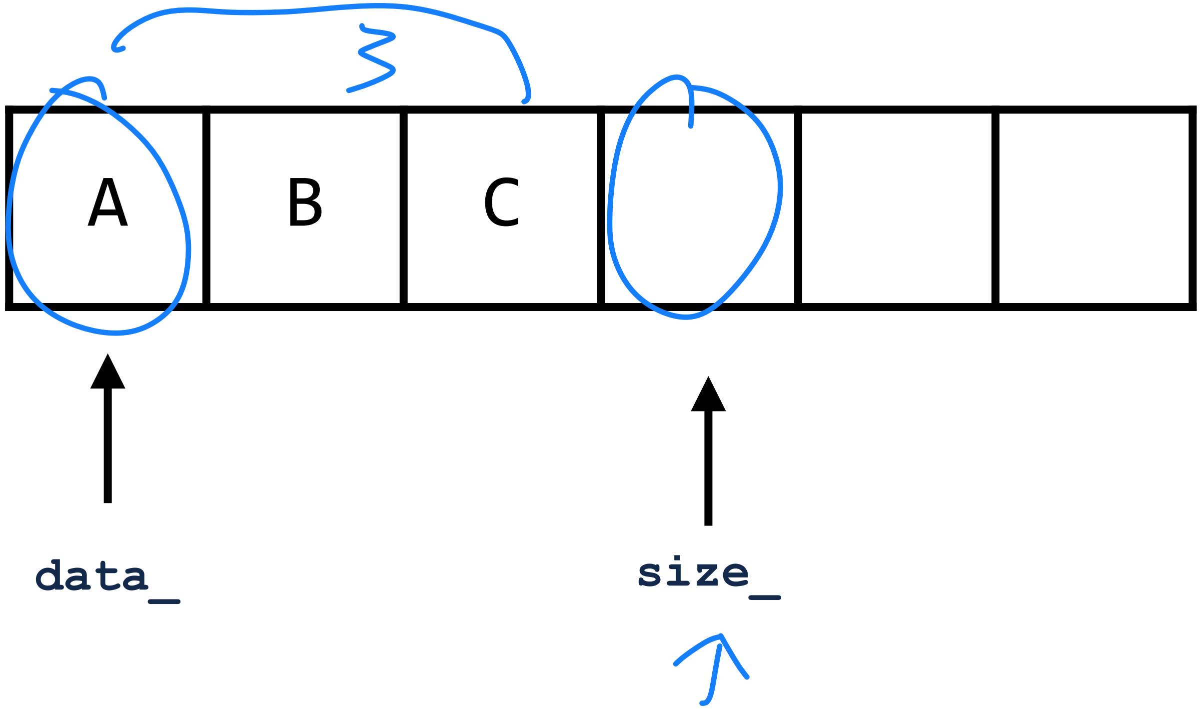
```
#pragma once

template <typename T>
class List {
public:
    /* --- */
private:
  T *data_;

  T *size_;

  T *capacity_;

    /* --- */
};
```



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

Reminder: C++ will handle sizeof calcs for us.

List.h

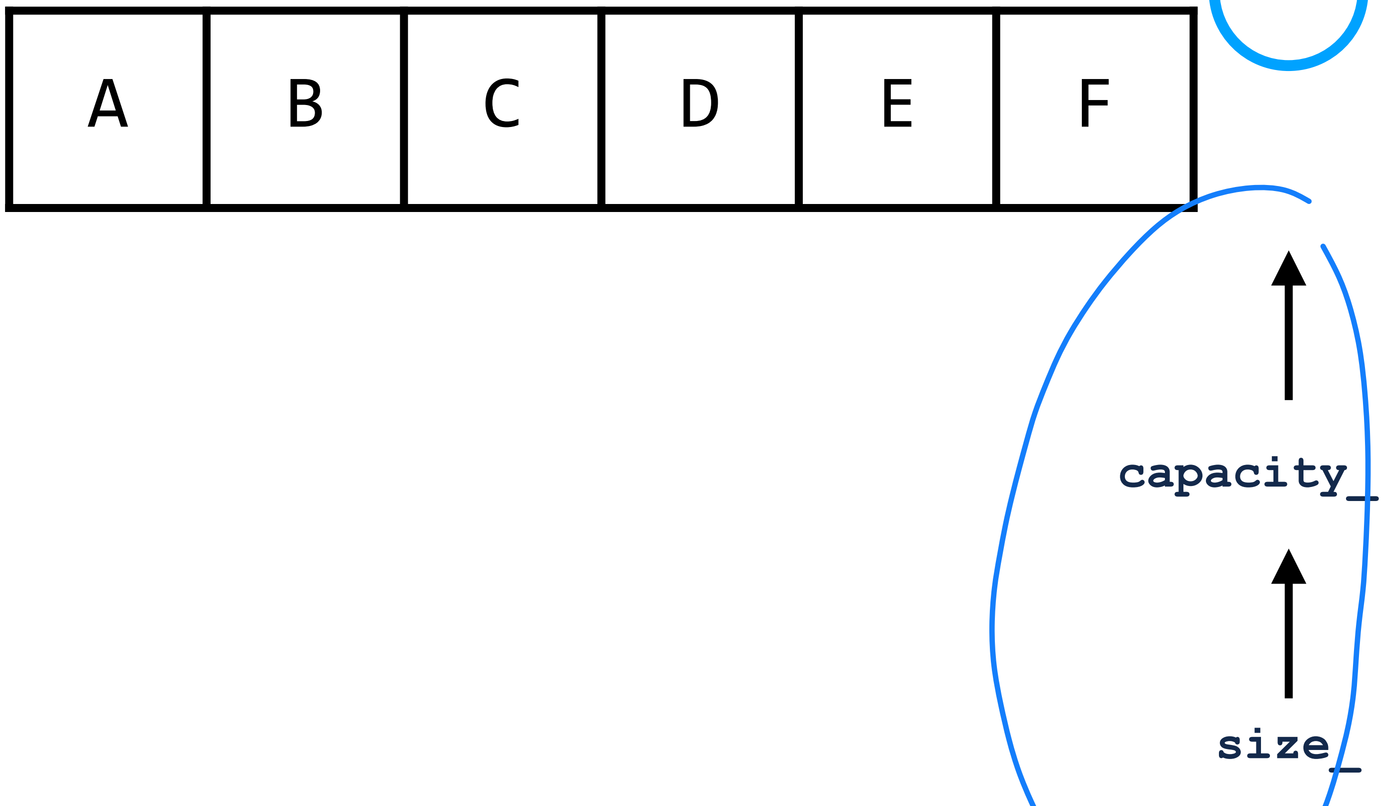
```
#pragma once

template <typename T>
class List {
public:
    /* --- */
private:
  T *data_;

  T *size_;

  T *capacity_;

    /* --- */
};
```



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

Array List: [ ]

pointer (data_)

+ i

C	S	2	2	5					

Random access at index i

↳ Data

↳ size

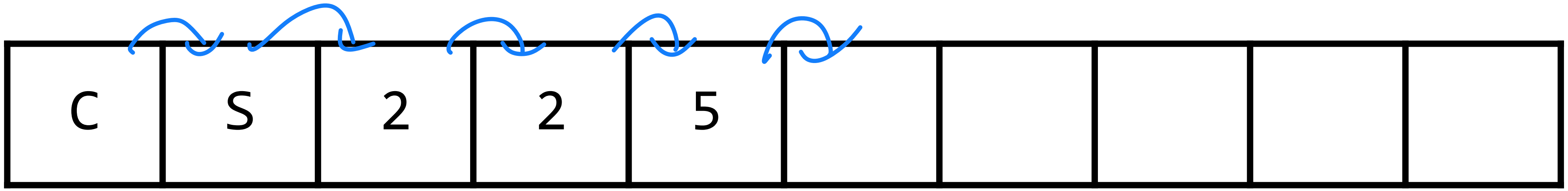
↳ capacity

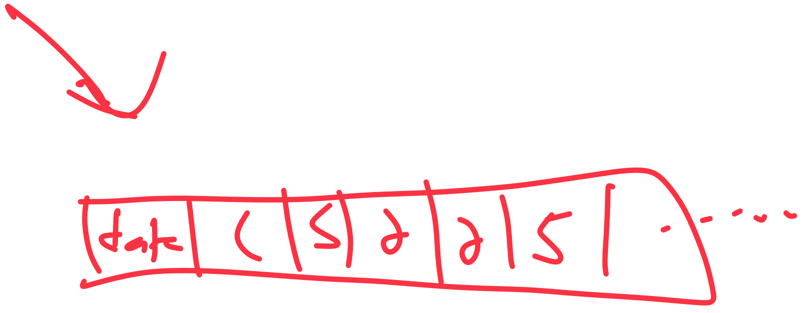
data_ + i

is item of interest

O(1)

Array List: `insertFront(data)`





1) Find position of interest $O(1)$
   ↳ position is data

2) Add new item ← $O(1)$
   ↳ move C, move S, ...
   ↳ Move all items after insert location ← $O(n)$

$O(n)$ overall

Array List: `insertBack(data)`

size pointer

C	S	2	2	5					

0) Make sure I can insert by checking if size != capacity $O(1)$

1) Insert data at size $O(1)$

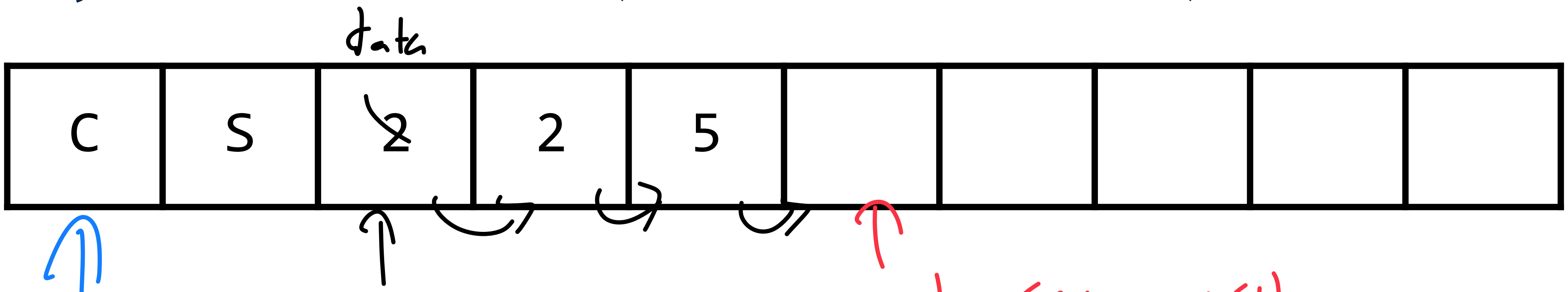
and check if full

2) size++; (so that we remember next insert loc)

$O(1)$

$O(1)$ to insertBack in array

Array List: `insert(data, index)`



Best case O(1)

Worst case O(n)

Like insert front

(1) find insert location

(2) Move all items past insert point

(3) Add data to new opening

Array List: Not at capacity

Join Code: 225

delete at size -1

O(1) del

size --

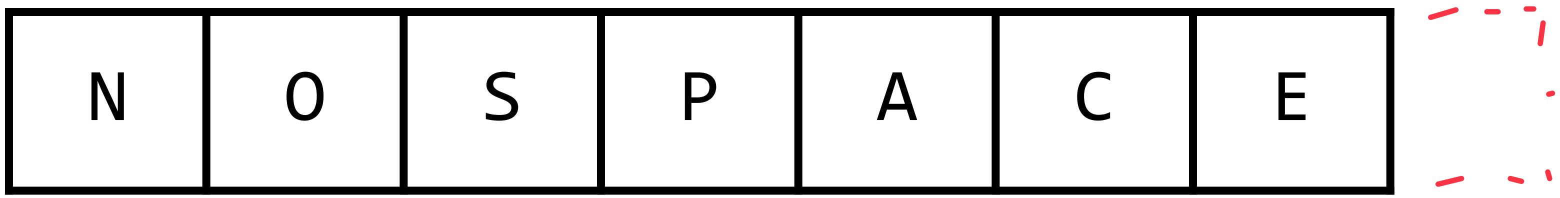
~~C~~	S	2	2	~~5~~					

O(1) to shift

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

Array List: addspace(data)

insert Back

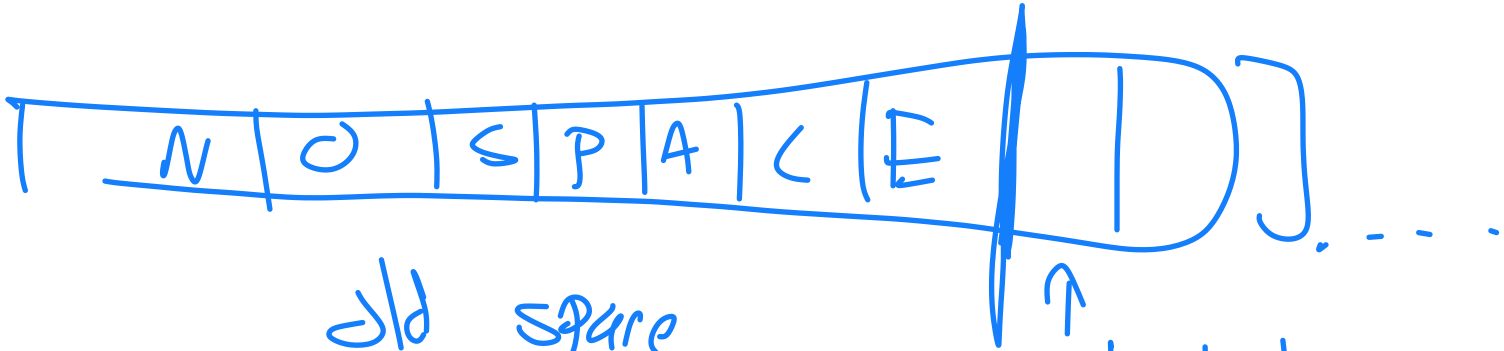


capacity

size

cannot modify capacity

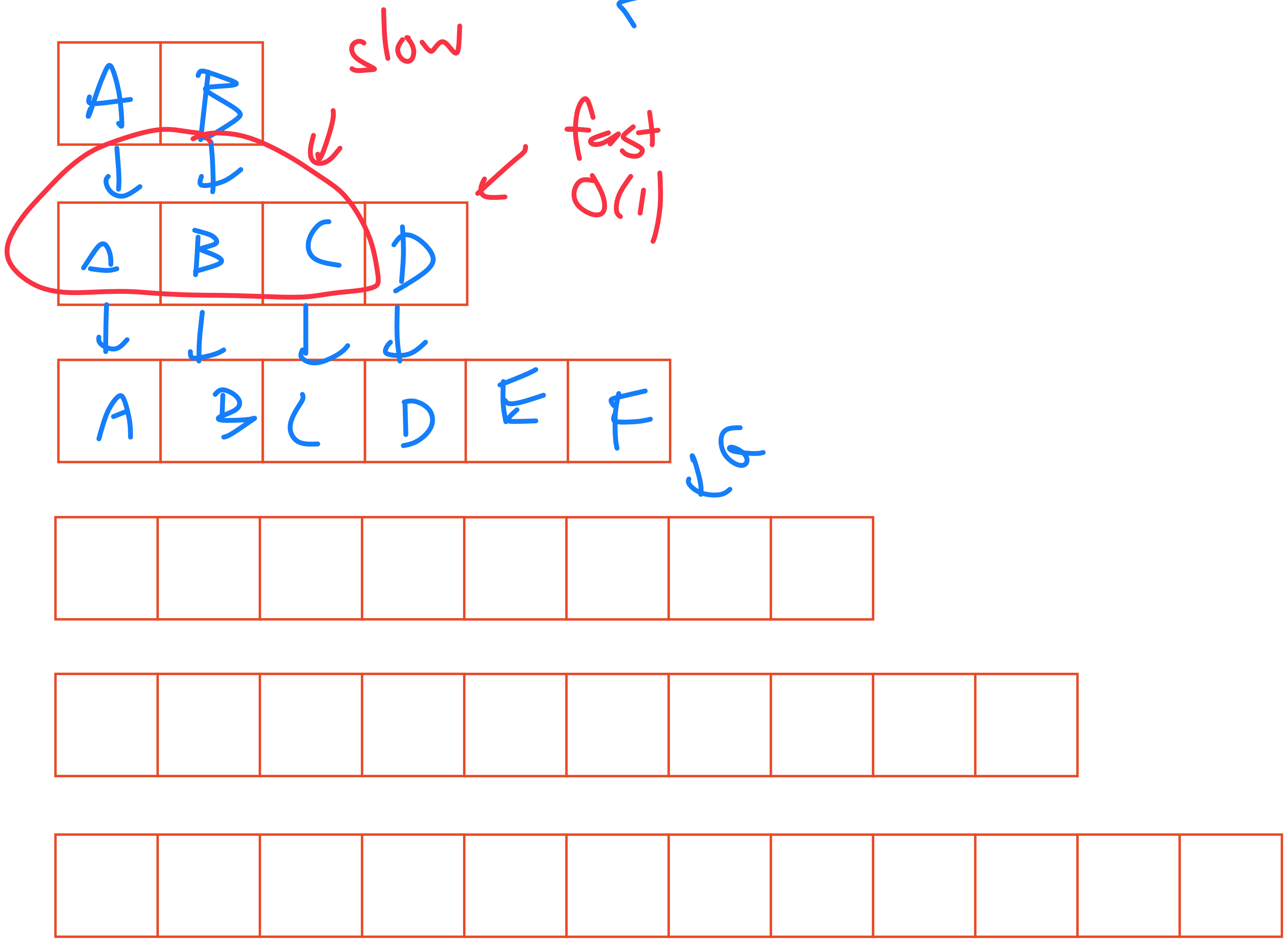
Instead must create new array



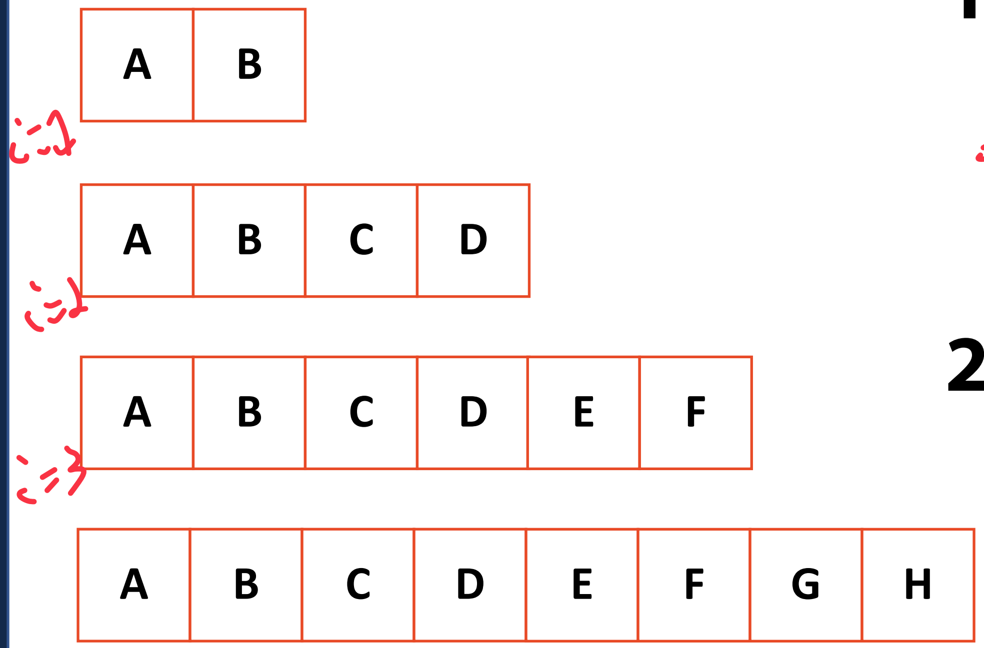
old space

+ at least one new

Resize Strategy: +2 elements every time



Resize Strategy: +2 elements every time



1) How many copy calls per reallocation?

For realloc i, $2i$ copies

2) Total reallocations for N objects?

Let k is # reallocs $\quad K = \lceil \frac{N}{2} \rceil$

Resize Strategy: +2 elements every time

A	B

A	B	C	D

A	B	C	D	E	F

A	B	C	D	E	F	G	H

1) How many copy calls per reallocation?

For reallocation i, 2i copy calls are made

2) Total reallocations for N objects?

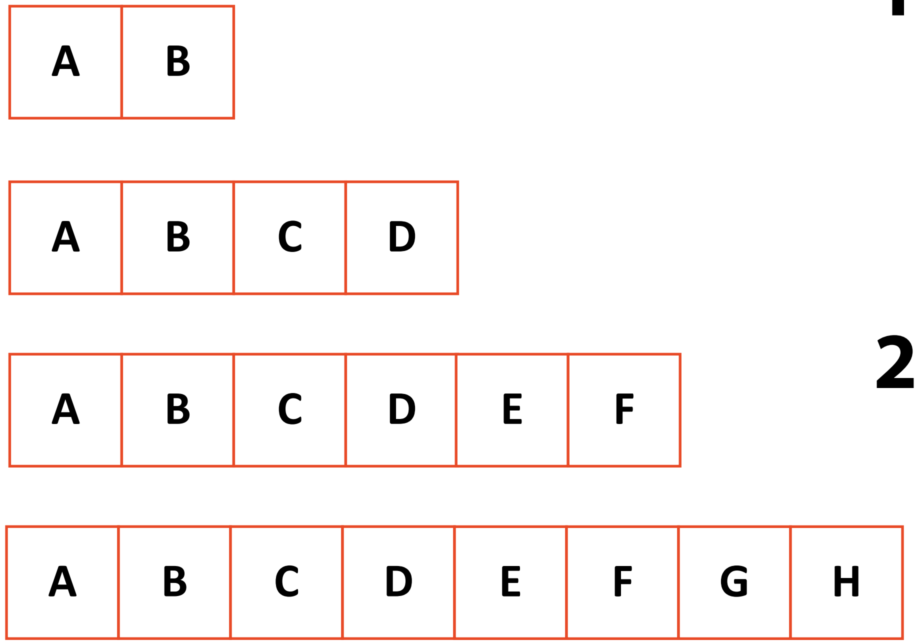
Let k be the number of reallocs, $k = \dfrac{N}{2}$

Total number of copy calls:

$$\sum_{i=1}^{k} 2i = k(k+1) = k^2 + k$$

plus in 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: O(n)

Look at runtime as average over N inserts

↳ Do a direct calc of the sum total work

$$\left(\frac{N^2 + 2N}{4}\right) / N \rightarrow \sim N$$

work per op

Big O: O(n)

2 types of inserts

1) | A | B |

The easy O(1) insert

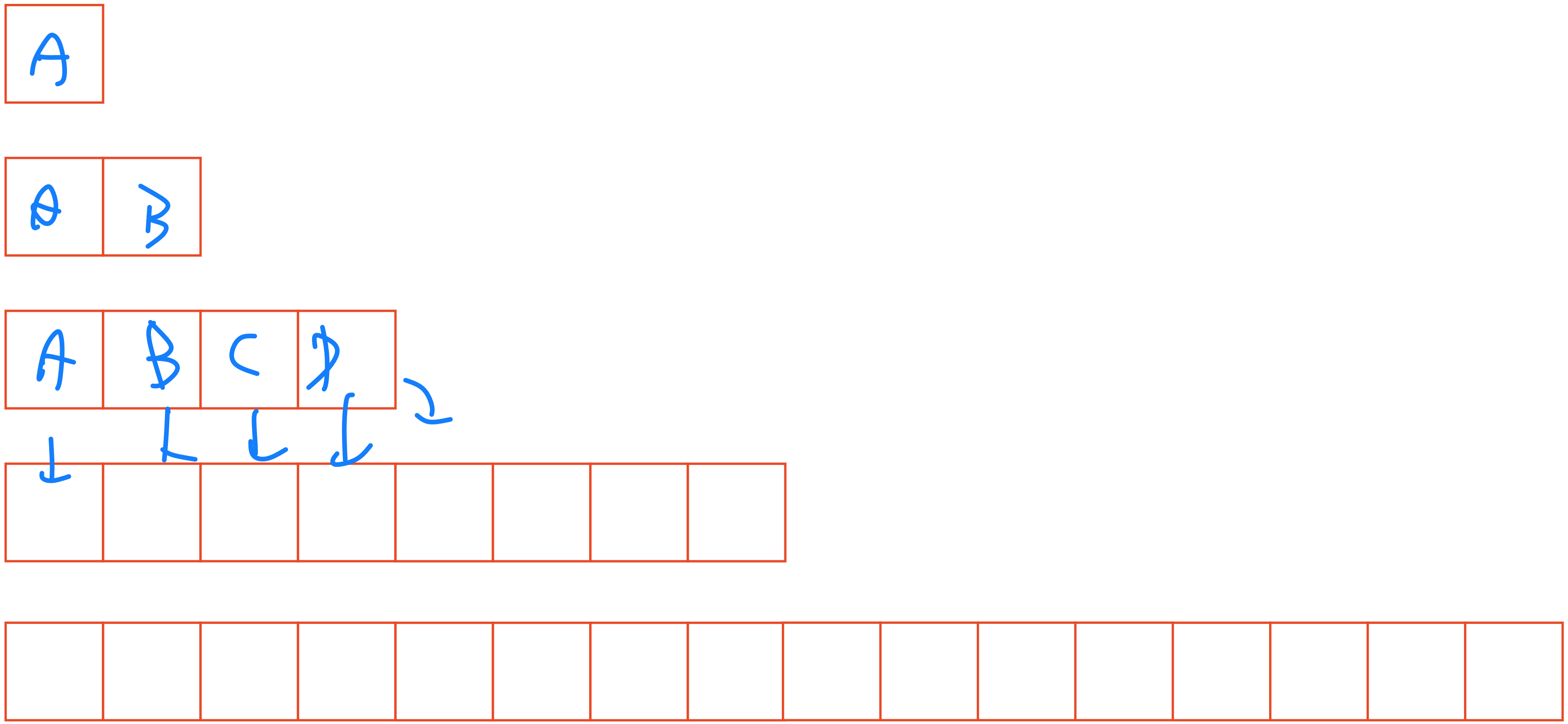
copy 2 items

2) | A | B | C |

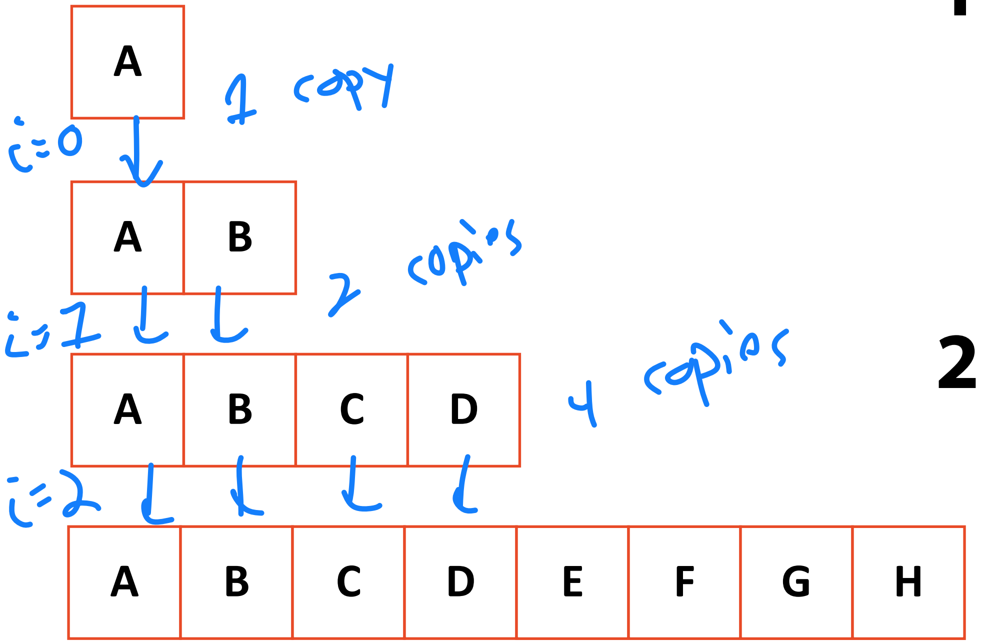
+ 1 new insert

O(n)

Resize Strategy: x2 elements every time



Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

for realloc i, 2^i copies

Many more copies per realloc

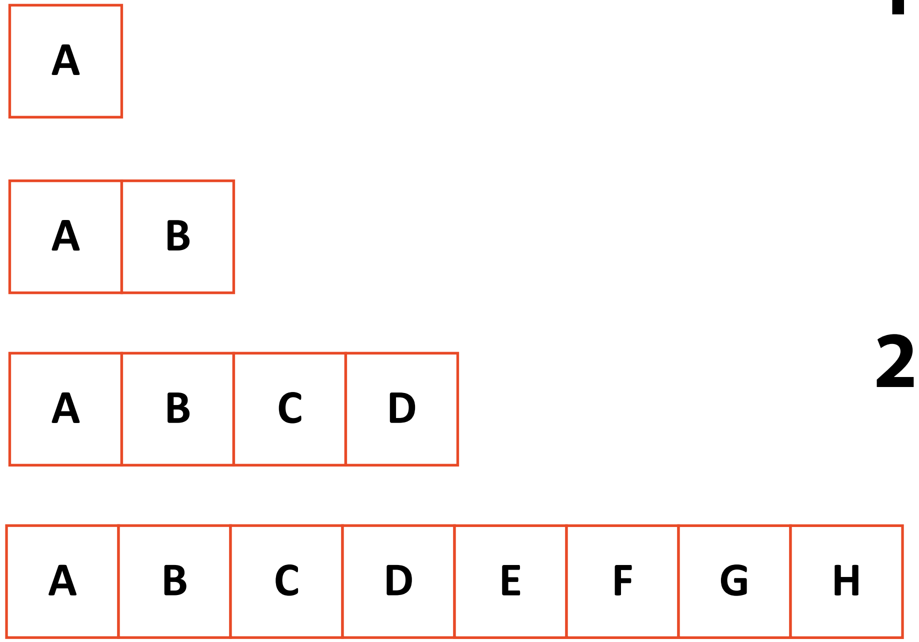
2) Total reallocations for N objects?

$n < 2^k \rightarrow k \approx \log_2 n$

Many fewer reallocations

Tradeoff

Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

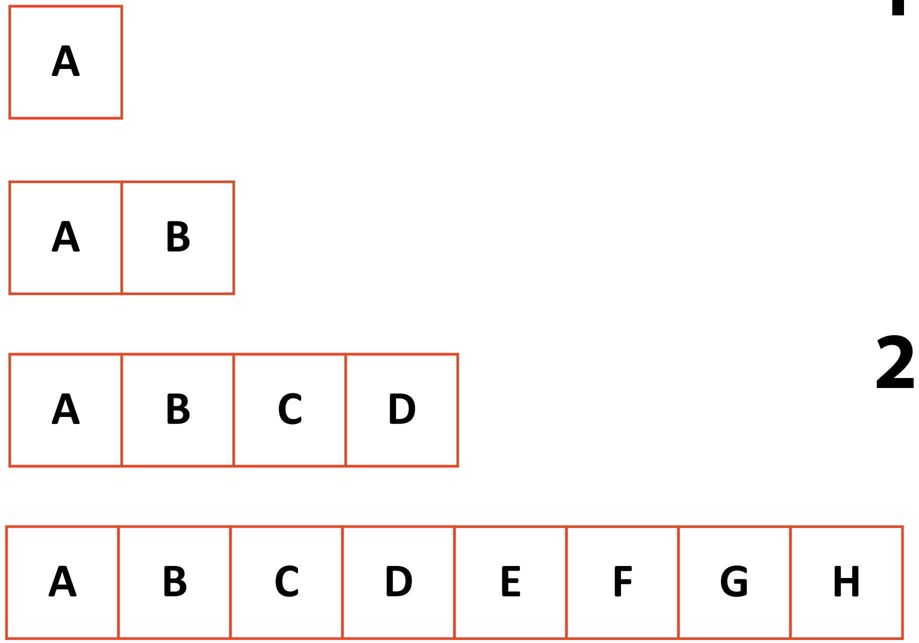
For reallocation i, 2^i copy calls are made

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 = $\lceil log_2 N \rceil$

plus in k

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: $2N - 1$

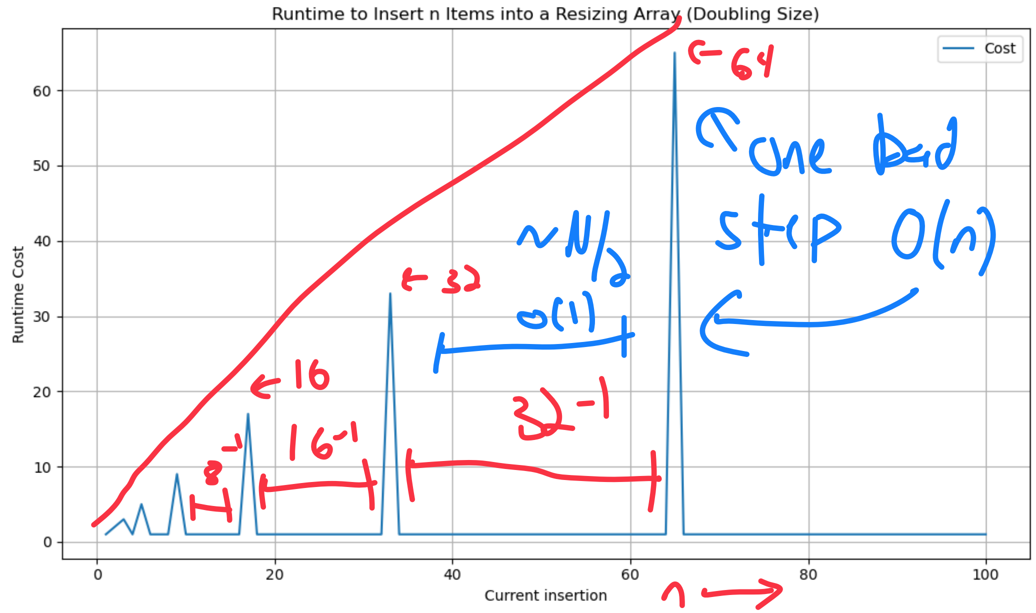
Amortized: $O(1)^*$

Precise total work over N calls

$$\frac{2N-1}{N} \cong 2 \text{ work per insert}$$

Big O: $O(n)$

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:

amortization

- 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

On exam 1

	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1) !!
Insert after **given** element ↳ a pointer	O(1)	O(n)
Remove after **given** element ↳ pointer ref to pointer	O(1)	O(n)
Insert at **arbitrary** location	O(n) Find O(1) insert O(n)	O(1) Find O(n) insert O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

* Special Cases?

Insert / Remove Front
O(1)

Insert / Remove Back *
O(1)
Not full

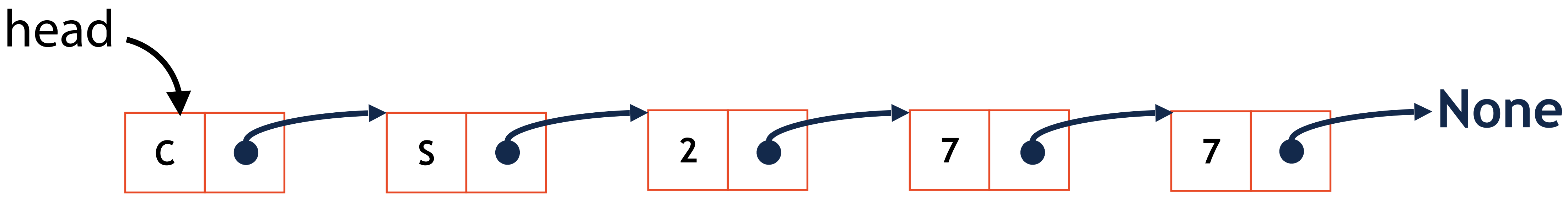
Thinking critically about lists: tradeoffs

The implementations shown are foundational (simple).

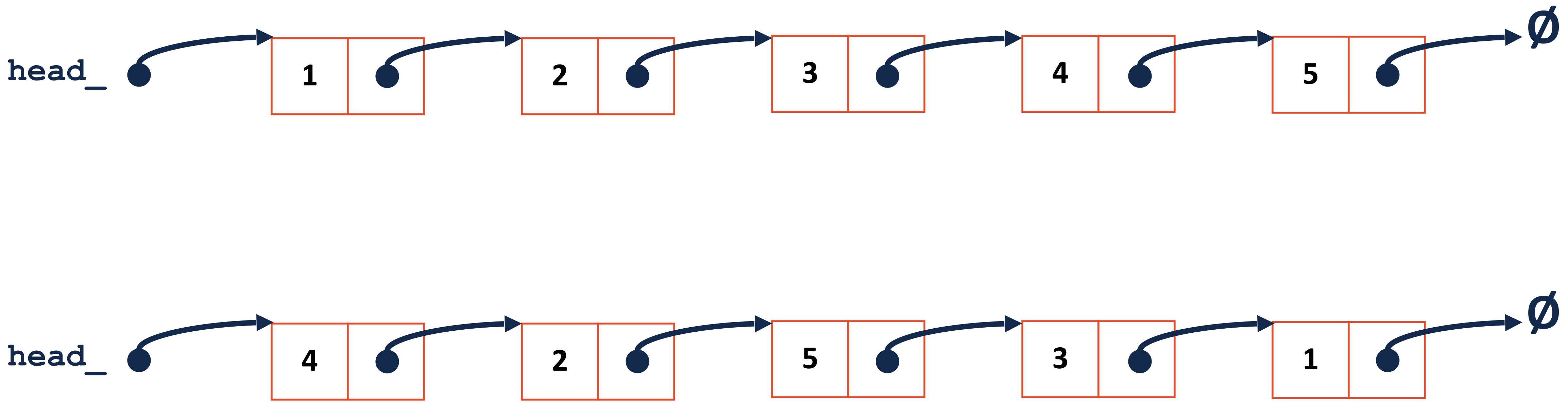
Can we make our lists better at some things? What is the cost?

Thinking critically about lists: tradeoffs

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



Thinking critically about lists: tradeoffs

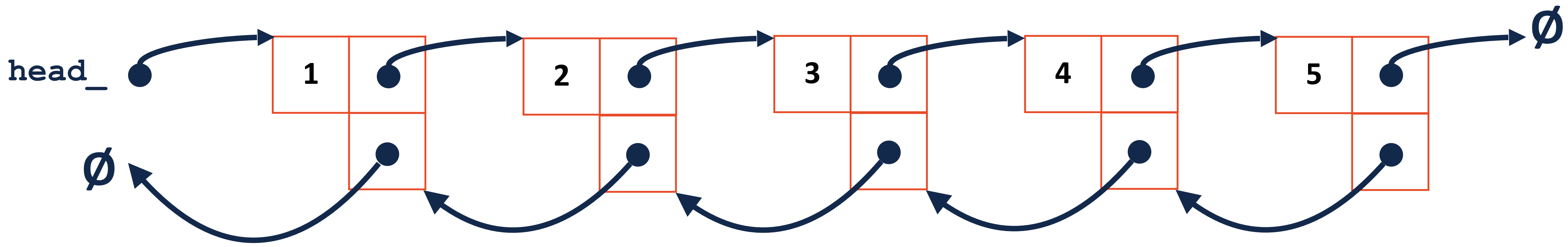


Thinking critically about lists: tradeoffs

2	7	5	9	7	14	1	0	8	3

0	1	2	3	5	7	7	8	9	14

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?