Final thoughts on arrays (& lists)

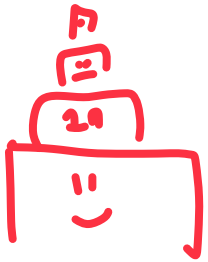
Last content on exam 1

Data Structures + Stacks and Queues

CS 225

Brad Solomon

September 9, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

workshop + info meeting

DEVELOP AN APP

WITH SIGMOBILE{ }

SEP 14

6PM TIL 8PM

COME TO **SIEBEL CS, ROOM 2406**
FROM **6-8PM** ON **SEP 14**

NO EXPERIENCE NEEDED!

BEGINNERS WELCOME

ARE YOU INTERESTED IN...

- Systems design
- App development
- Developing projects

WHAT IS SIGMOBILE?

We are an RSO associated with ACM focused on all things regarding app/mobile development

Join the Discord!

Learning Objectives

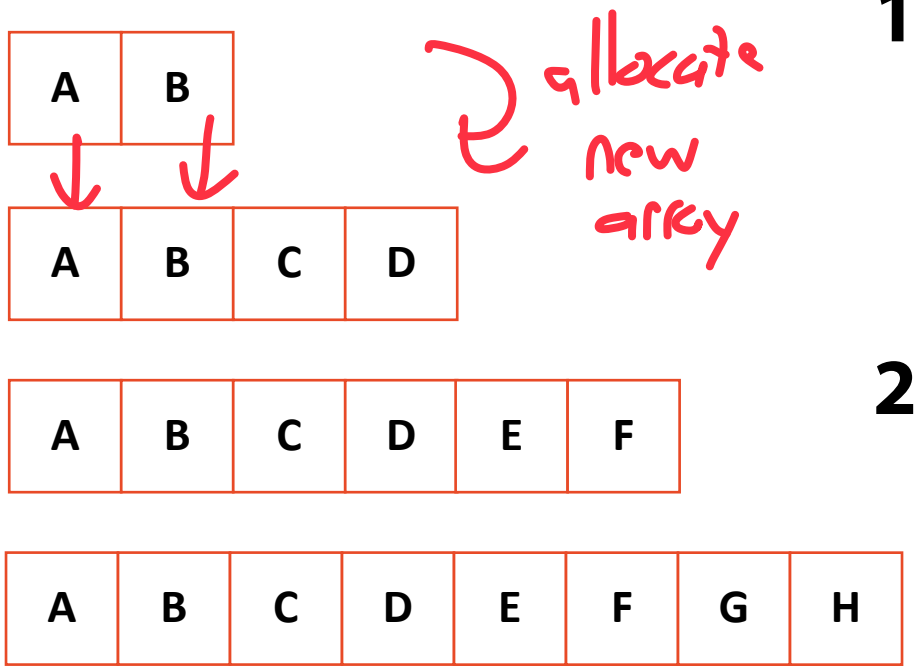
Finalize resize implementation for an array list

Consider extensions to lists (data structure tradeoffs)

Introduce the stack and the queue data structure

Introduce and explore iterators

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

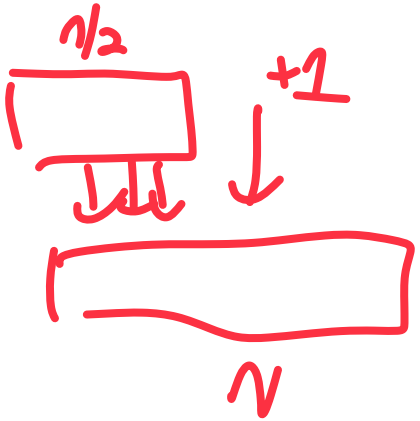
Amortized: ~N ops per insert

Precise total work over N calls

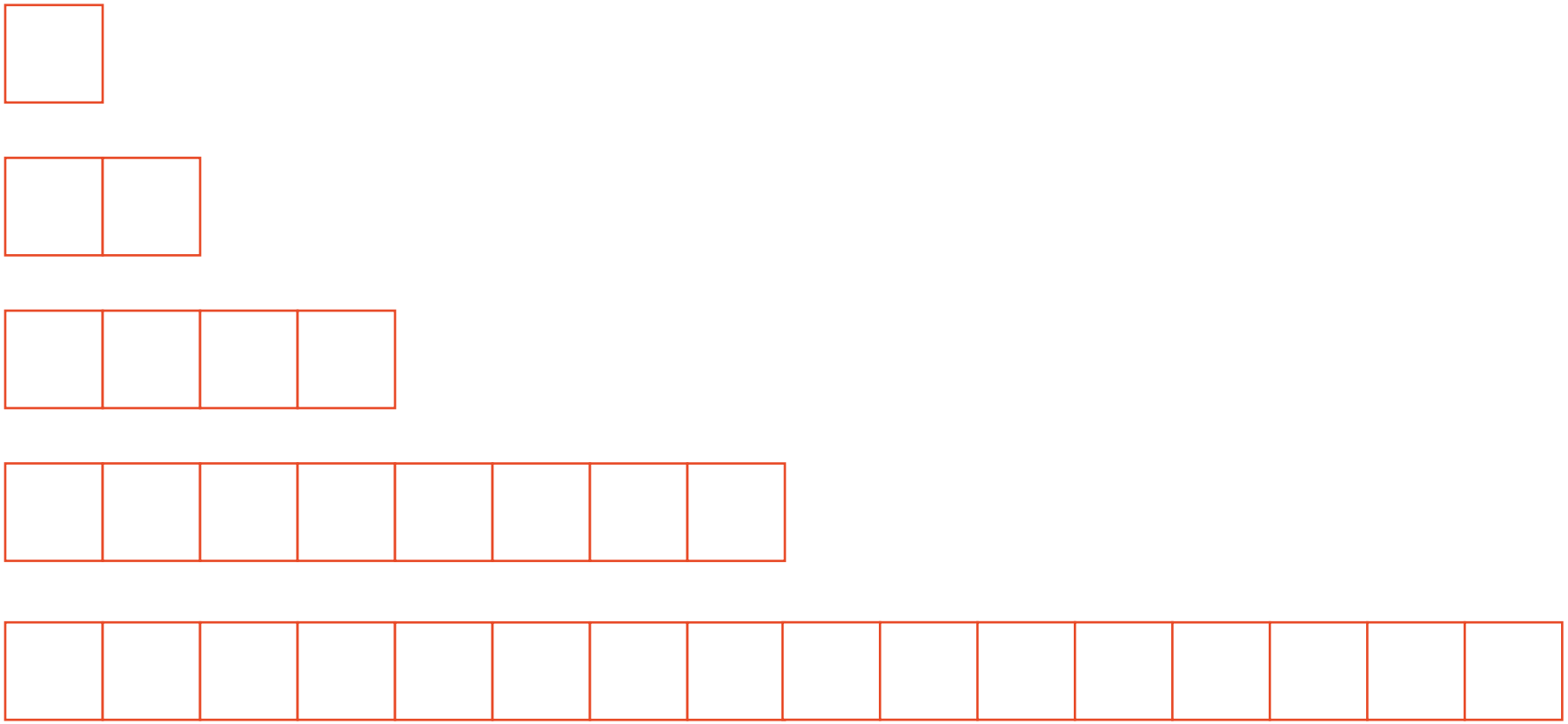
Big O: O(N)

Upperbound on worst case

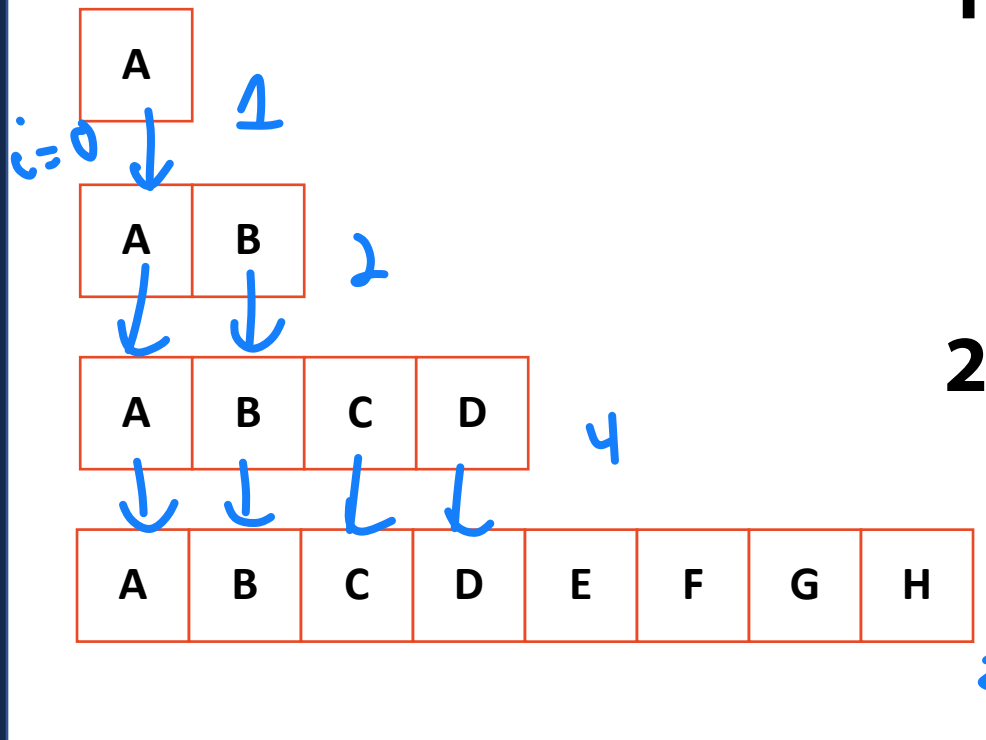
↳ I have to reallocate



Resize Strategy: x2 elements every time



Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

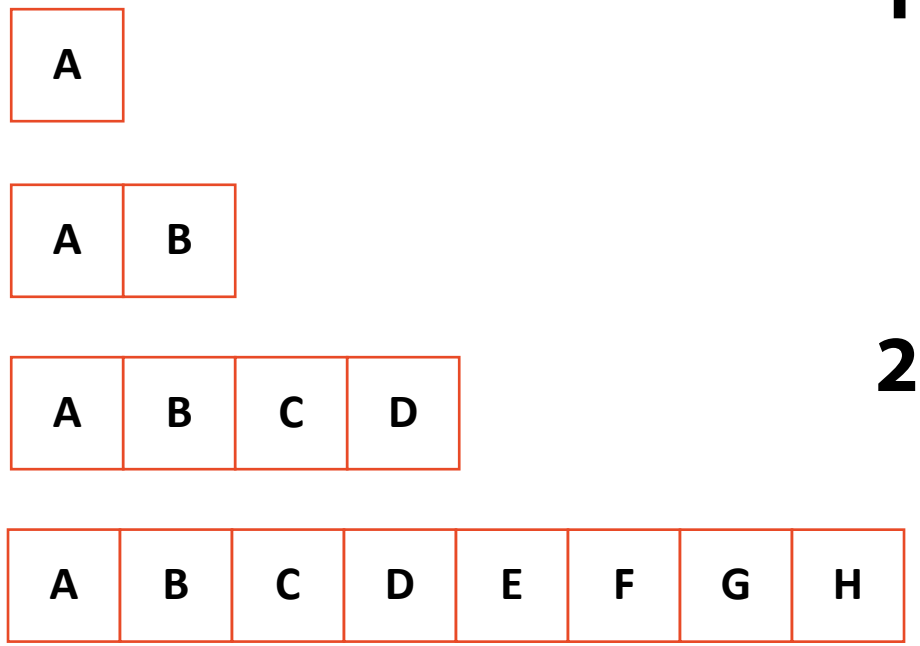
for realloc i , 2^i

2) Total reallocations for N objects?

$N \leq 2^k$ (k = # times realloc)

↳ $k = \log_2 N$

Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

For reallocation i, 2^i copy calls are made

vs $2 \cdot i$

2) Total reallocations for N objects?

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

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

plug in k

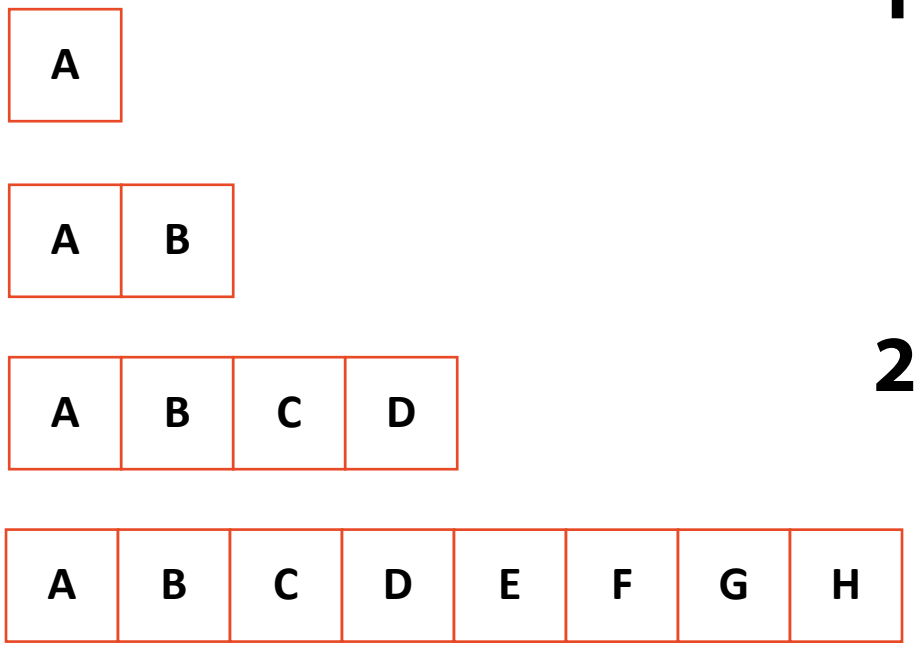
vs $k = N/2$ *

$2^{\log_2 N + 1} - 1 = 2 \cdot 2^{\log_2 N} - 1 \simeq 2N - 1$

Tradeoff in action!

x2 less frequent reallocations each reallocation is larger

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

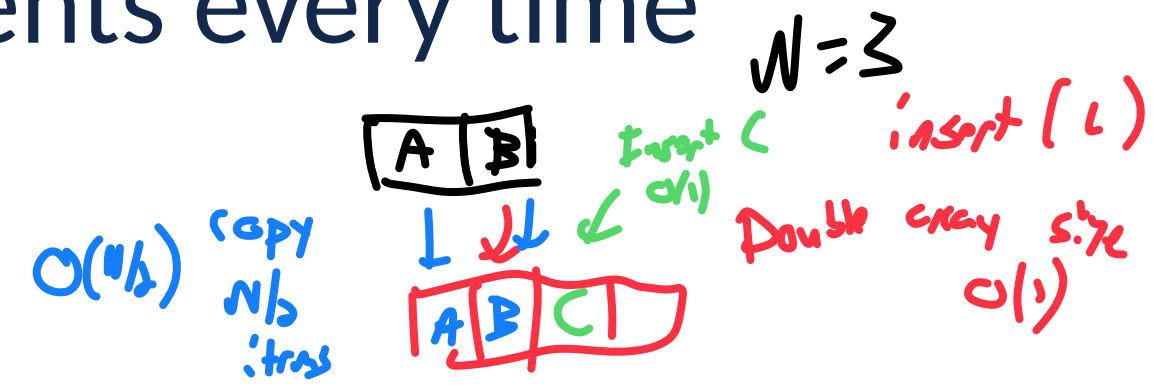
Total copies for n inserts: $2N - 1$

Amortized:

Big O:

Resize Strategy: x2 elements every time

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



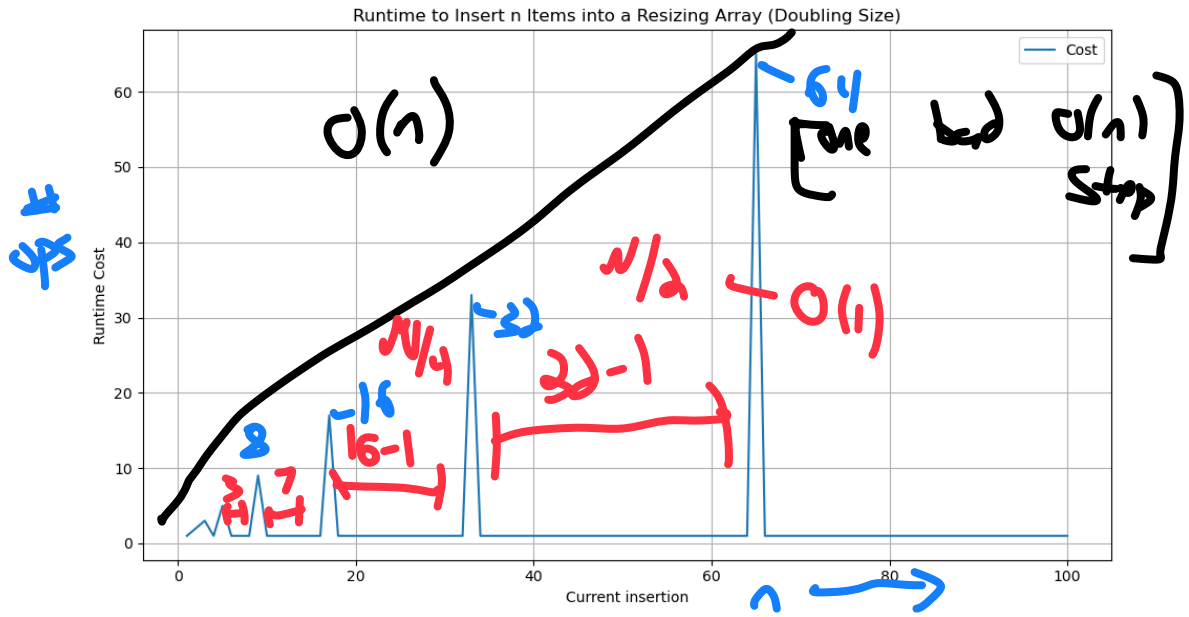
Amortized: $O(1)^*$

Big O: $O(n)$

Precise total work over N calls

$2N-1 / N$ = # work per insert is ~2

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

infrequent

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	O(n)	O(1) ☺
Insert after **given** element w/ ref to pointer	O(1) ☺	O(n)
Remove after **given** element w/ 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) find O(1) del O(n)	O(1) find O(n) remove O(n)
Search for an input **value**	O(n)	O(n)

Special cases:

insert/Remove front O(1)

insert/Remove Back O(1)*

* array not full

List Implementation

	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1)
Insert after **given** element	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

Special Cases: insertFront insertBack (not full)

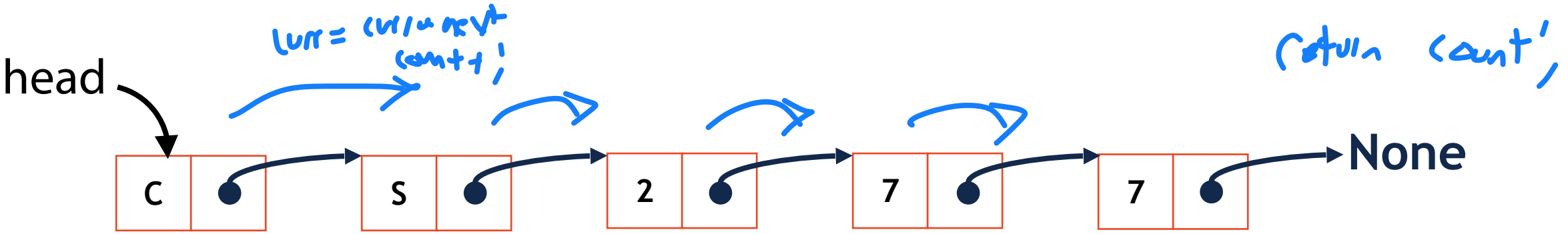
Thinking critically about lists: tradeoffs

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: O(n) → O(1)



List
↳ ListNode* head
↳ int size;
↳ modifying insert / remove
size++ / size--

ListNode
↳ T & data
↳ ListNode * next;

Goal: O(n) → O(1) size()

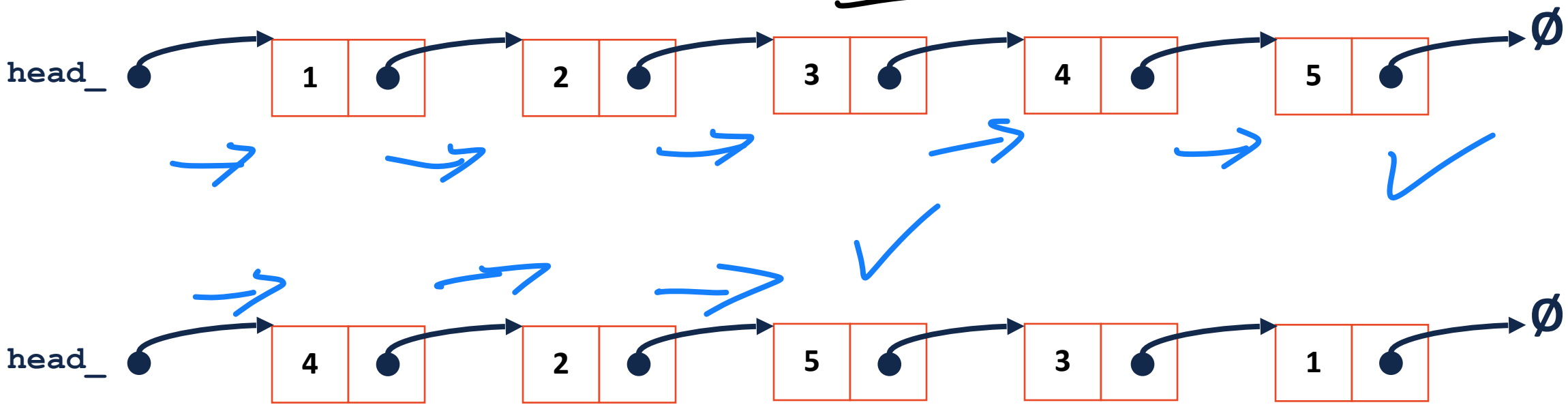
Cost: One single int in memory
↳ An O(1) increase in insert & remove

Thinking critically about lists: tradeoffs

5

Sorted linked list

A BAD trade off



Gain: faster search on sorted

↳ Problem: Linked list has no random access

Cost: Insert always O(n) now

A situation where you needed to filter a list in smallest → largest

Thinking critically about lists: tradeoffs

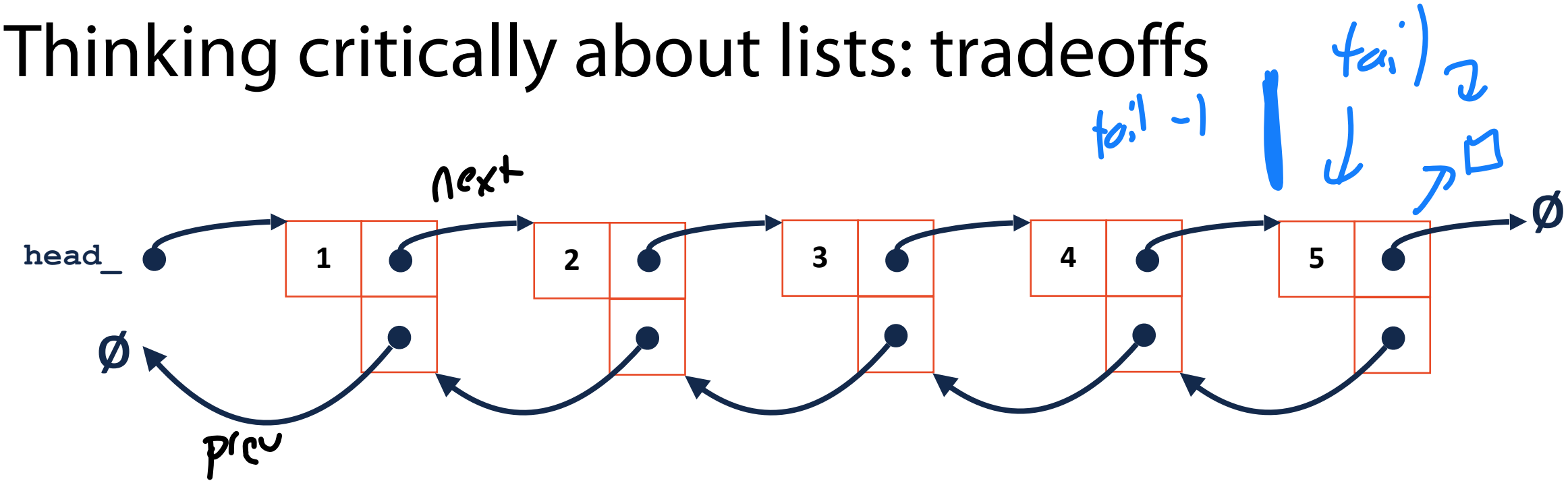
2	7	5	9	7	14	1	0	8	3

0	1	2	3	5	7	7	8	9	14

Gain: O(log n) search

Cost: No O(1) insertBack

Thinking critically about lists: tradeoffs



Gain: ???

Cost: 1 pointer per node

Combination of tail & prev lets us removeBack() O(1)!

Thinking critically about lists: tradeoffs

As we progress in the class, we will see that $O(n)$ isn't very good.

Take searching for a specific list value:

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

Can we make a 'list' that is O(1) to insert and remove?

↳ special case exceptions!

Stack Data Structure

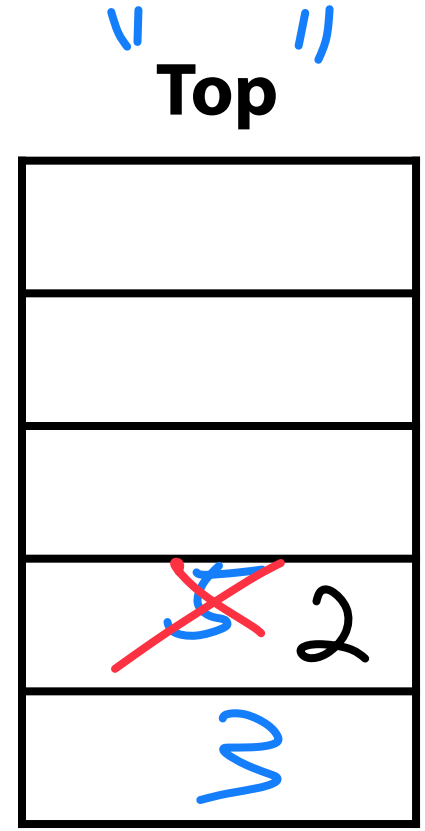
A **stack** stores an ordered collection of objects (like a list)

However you can only do three* operations:

Push: Put an item on top of the stack

Pop: Remove the top item of the stack

Top: Return the top item of the stack




```
push(3); push(5); pop(); push(2)
```

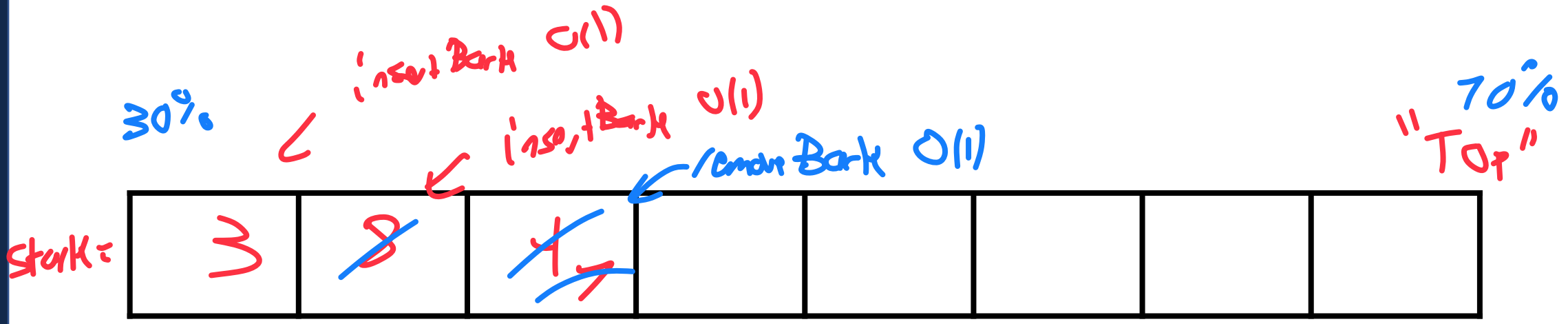
Stack Data Structure

C++ has a built-in stack

Underlying implementation is vector or deque

Is the 'top' the first or last item in an array?

```
1  #include <stack>
2  int main() {
3    stack<int> stack;
4    stack.push(3);
5    stack.push(8);
6    stack.push(4);
7    stack.pop();
8    stack.push(7);
9    stack.pop();
10   stack.pop();
11 }
```



Stack Data Structure

Push(X) is equivalent to …

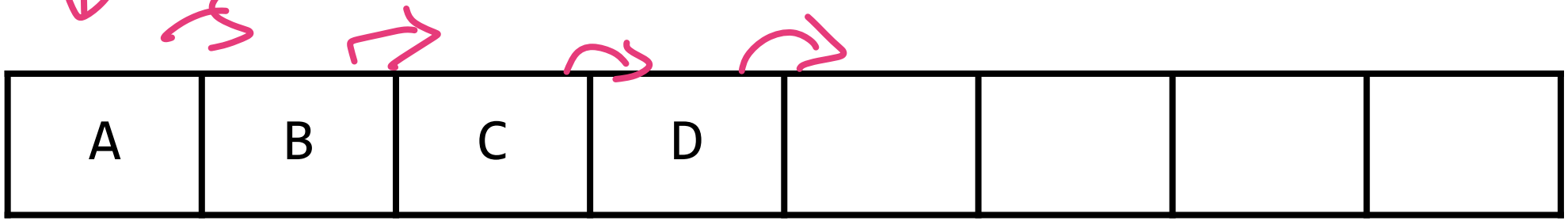
After class Q: why not "front as top"

add( )

X

Worst case would not best case!

O(n) to move to the spot



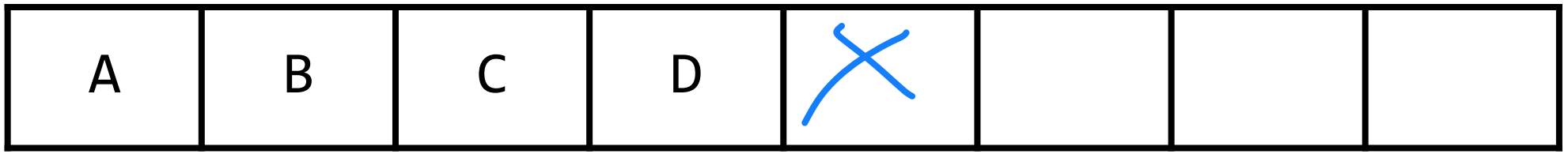
Stack Data Structure

`Push(X)` is equivalent to `insertBack(X)`

```
*size = X;
size++;
```

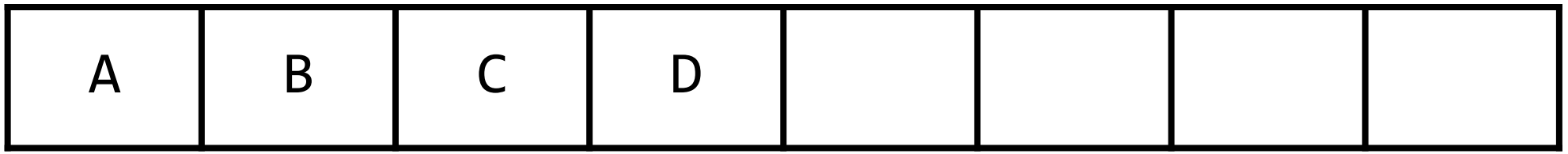
O(1)

size



Stack Data Structure

Pop( ) is equivalent to…



Stack Data Structure

`Pop()` is equivalent to `removeBack()`

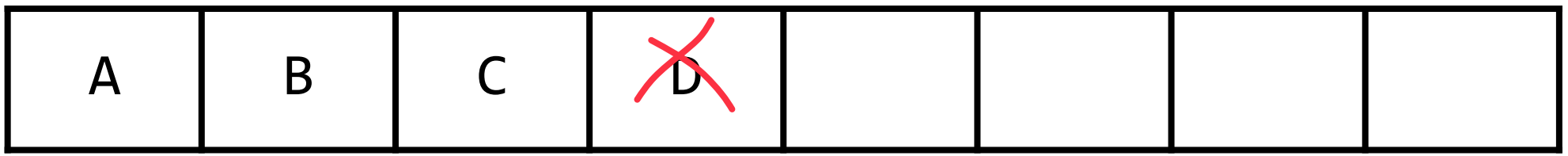
```
size--;
```

O(1)

```
remove(size);
```

O(1)

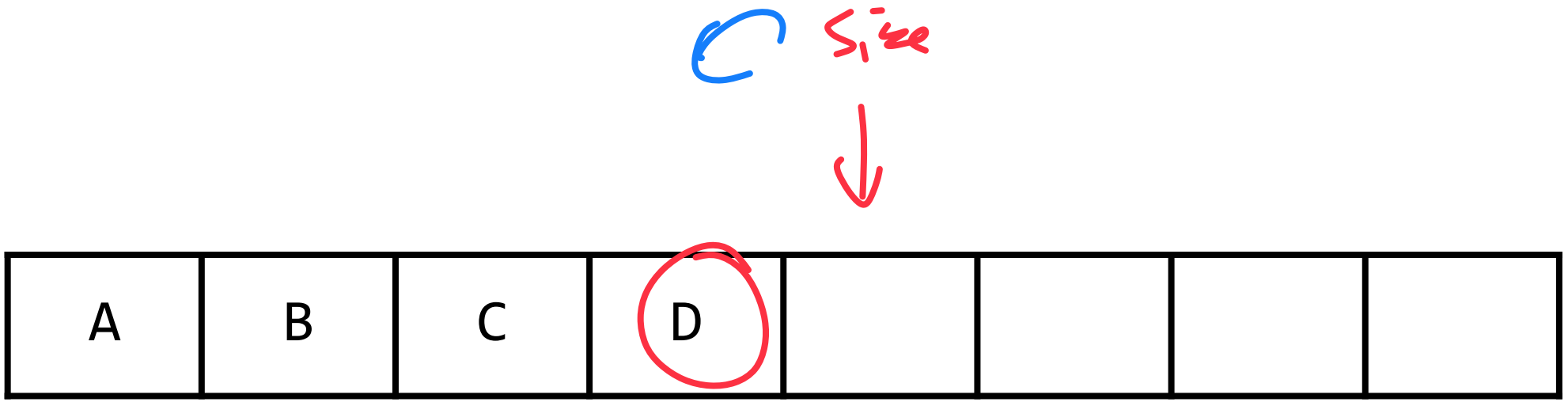
size



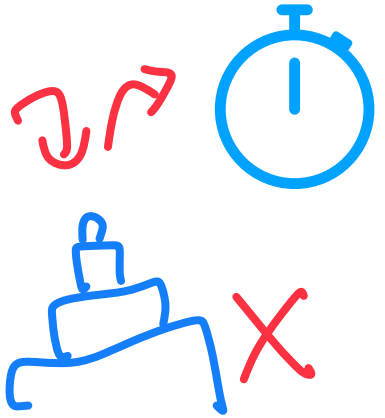
Stack Data Structure

Top() is tricky — remember size points to next available space!

```
return *(size - 1);
```



Stack ADT



- [Order]: last in first out (Lifo)
- [Implementation]: As array (top is last item)
  As linked list (first item)
- [Runtime]: O(1)* always for all ops

* array not full

Queue Data Structure

A **queue** stores an ordered collection of objects (like a list)

However you can only do three* operations:

Enqueue: Put an item at the back of the queue

Dequeue: Remove the front item of the queue

Front: Return the front item of the queue

```
enqueue(3); enqueue(5); dequeue(); enqueue(2)
```

Queue Data Structure

The queue is a **first in — first out** data structure (FIFO)

What data structure excels at removing from the front?

Can we make that same data structure good at inserting at the end?

Queue Data Structure

The C++ implementation of a queue is also a vector or deque — why?

Engineering vs Theory Efficiency

	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💓
Branch mispredict	5 seconds	Yawn 😯
L2 cache reference	7 seconds	Long yawn 😯😯😯
Mutex lock/unlock	25 seconds	Make coffee ☕
Main memory reference	100 seconds	Brush teeth
Compress 1K bytes	50 minutes	TV show 📺
Send 2K bytes over 1 Gbps network	5.5 hours	(Brief) Night's sleep 🛌
SSD random read	1.7 days	Weekend
Read 1 MB sequentially from memory	2.9 days	Long weekend
Read 1 MB sequentially from SSD	11.6 days	2 weeks for delivery 📦
Disk seek	16.5 weeks	Semester
Read 1 MB sequentially from disk	7.8 months	Human gestation 🐣
Above two together	1 year	🌍☀️
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓

(Care of https://gist.github.com/hellerbarde/2843375)

Engineering vs Theory Efficiency

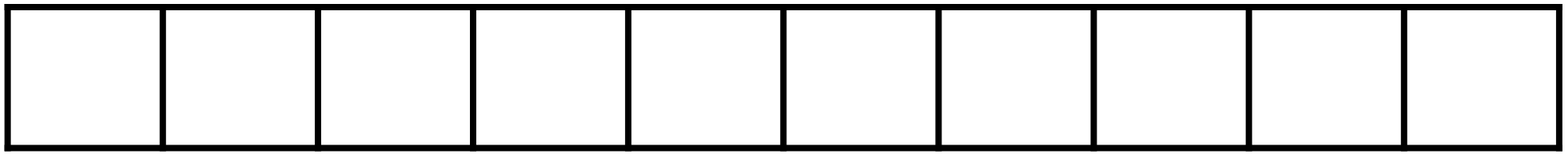
	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💓
Main memory reference	100 seconds	Brush teeth
SSD random read	1.7 days	Weekend
Disk seek	16.5 weeks	Semester
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓

(Care of https://gist.github.com/hellerbarde/2843375)

Queue Data Structure

```
q.enqueue(8);
q.enqueue(4);
q.dequeue();
```

What do we need to track to maintain a queue with an array list?

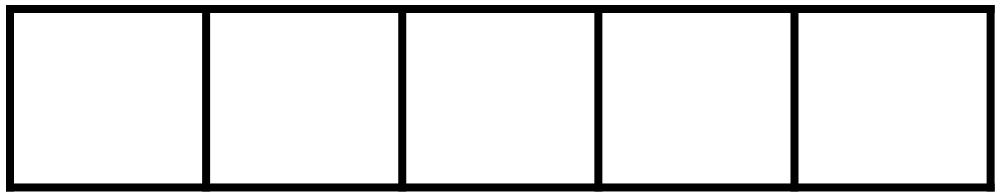


Queue Data Structure

Unlike the array list, it is easier to implement a Queue using unsigned ints

Queue.h

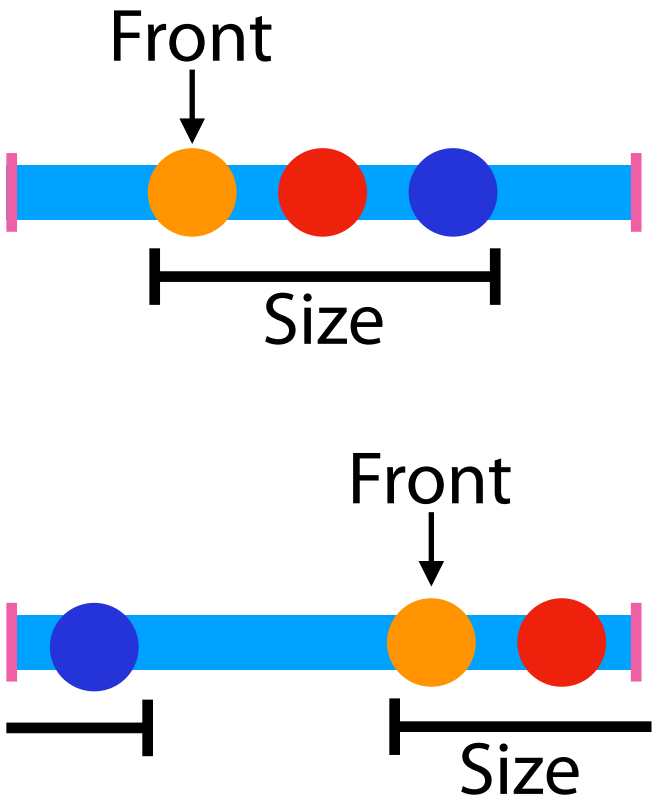
```
 1  #pragma once
 2
 3  template <typename T>
 4  class Queue {
 5    public:
 6      void enqueue(T e);
 7      T dequeue();
 8      bool isEmpty();
 9
10    private:
11      T *data_;
12      unsigned size_;
13      unsigned capacity_;
14      unsigned front_;
15  };
```

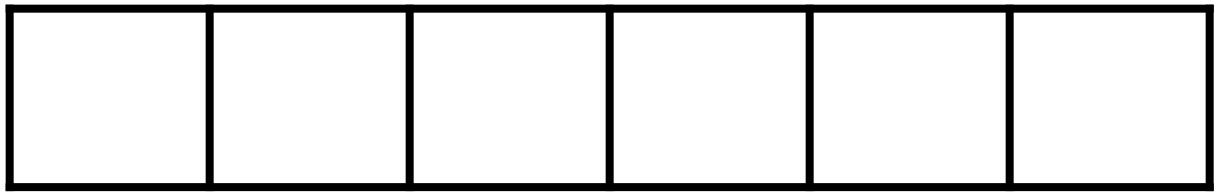


(Circular) Queue Data Structure

Queue.h

```
1  #pragma once
2
3  template <typename T>
4  class Queue {
5    public:
6      void enqueue(T e);
7      T dequeue();
8      bool isEmpty();
9
10   private:
11     T *data_;
12     unsigned capacity_;
13     unsigned size_;
14     unsigned front_;
15 };
```





Enqueue(D):

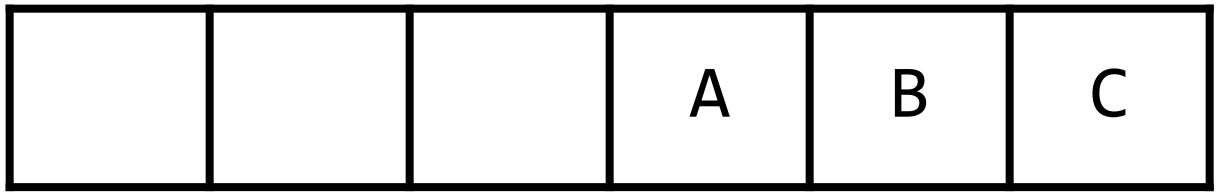
Dequeue():

Size:

Front:

Capacity:

```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```




```
Queue<int> q;
...
q.enqueue(D);
q.dequeue();
q.dequeue();
q.dequeue();
q.dequeue();
q.enqueue(E);
```

Enqueue(D): Add data to ‘back’ of queue

Insert D at index **(size+front) % capacity**

size++ (as long as size != capacity)

Dequeue(): Remove data at index front

front = **(front+1) % capacity**

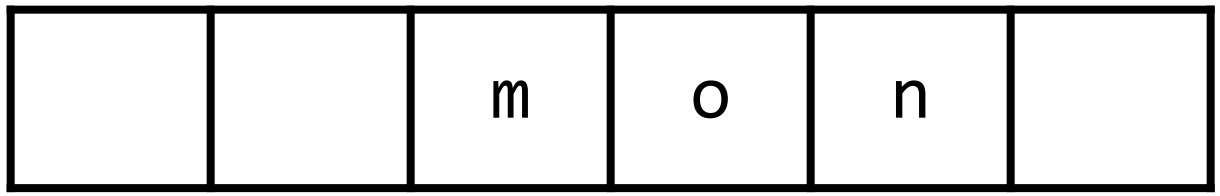
size-- (as long as size != 0)

Size: 3

Front: 3

Capacity: 6

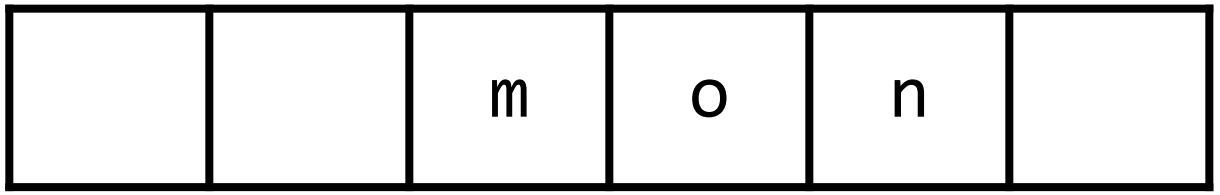
Queue Data Structure: Resizing

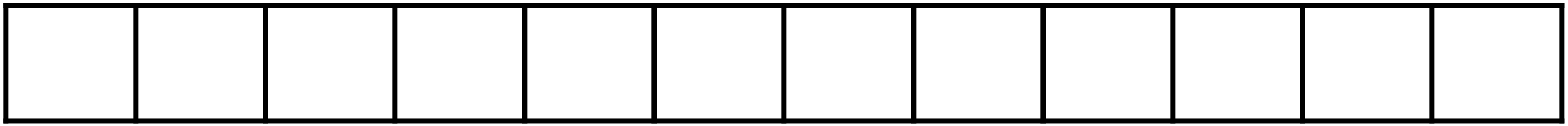



```
Queue<char> q;
…
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
```

Queue Data Structure: Resizing

```
Queue<char> q;
…
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
```





Queue ADT

- [Order]:
- [Implementation]:
- [Runtime]: