Amortization!

Data Structures

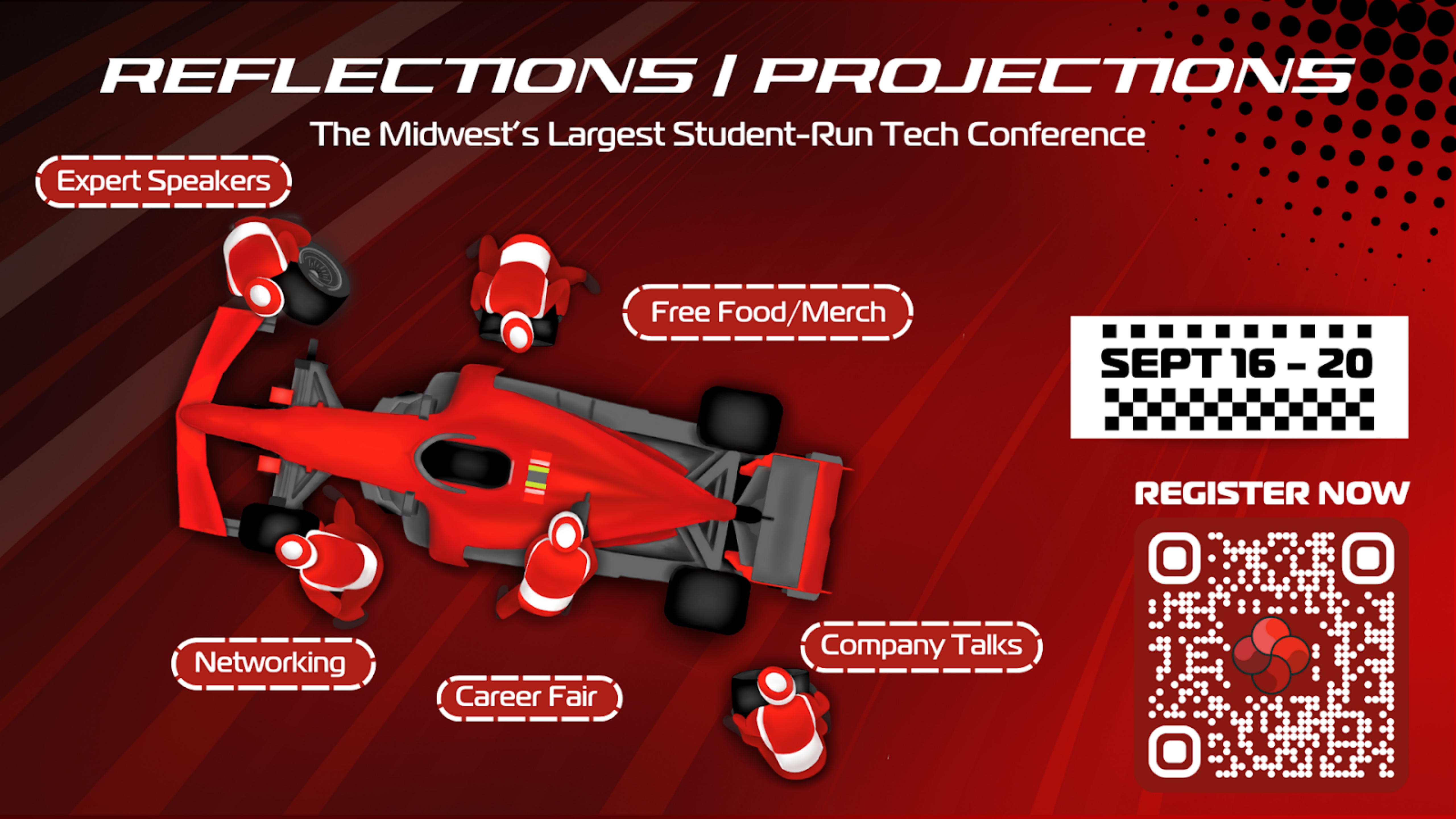
Final Array Theory & Stacks and Queues

Tradeoffs

CS 225
Brad Solomon

September 10, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science


REFLECTIONS | PROJECTIONS
The Midwest's Largest Student-Run Tech Conference
Expert Speakers
Free Food/Merch
SEPT 16 - 20
REGISTER NOW
Networking
Career Fair
Company Talks

Exam 1 (9/17 — 9/19)

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

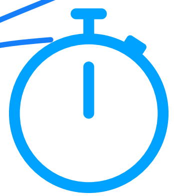
Register now

https://courses.engr.illinois.edu/cs225/fa2025/exams/

Preparing for Exams

Make sure you understand the coding assignments

Review lecture slides — especially review slides!



Take a look at 'staff notes' — added to website for past lectures

Do the practice exam before watching practice exam solution video

Learning Objectives

Discuss amortized analysis

Consider extensions to lists (data structure tradeoffs)

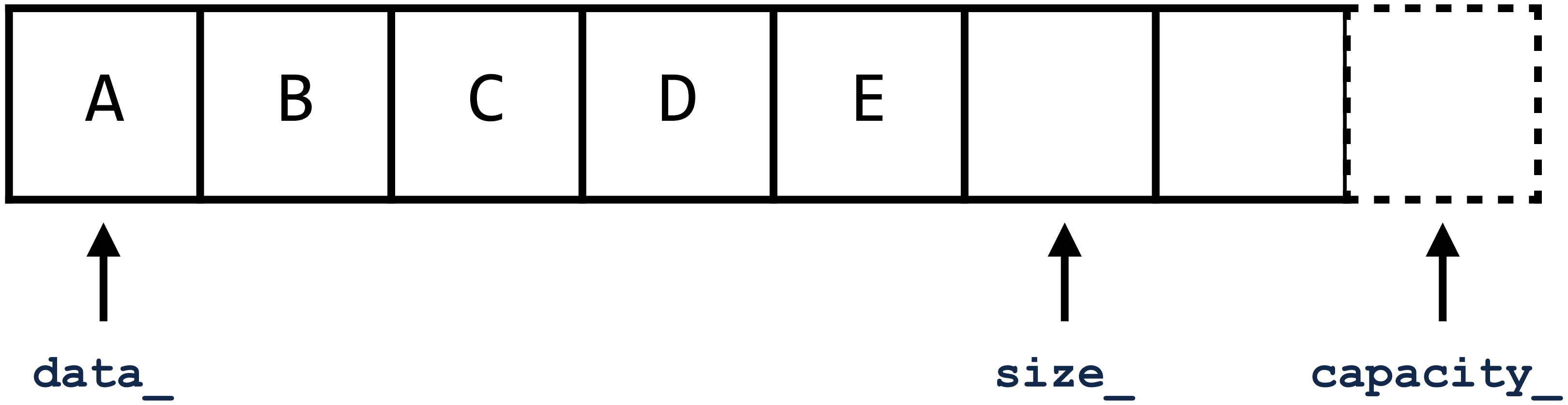
Introduce the stack and the queue data structure

Introduce and explore iterators

on exam

not on exam

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

Array List: Not at capacity

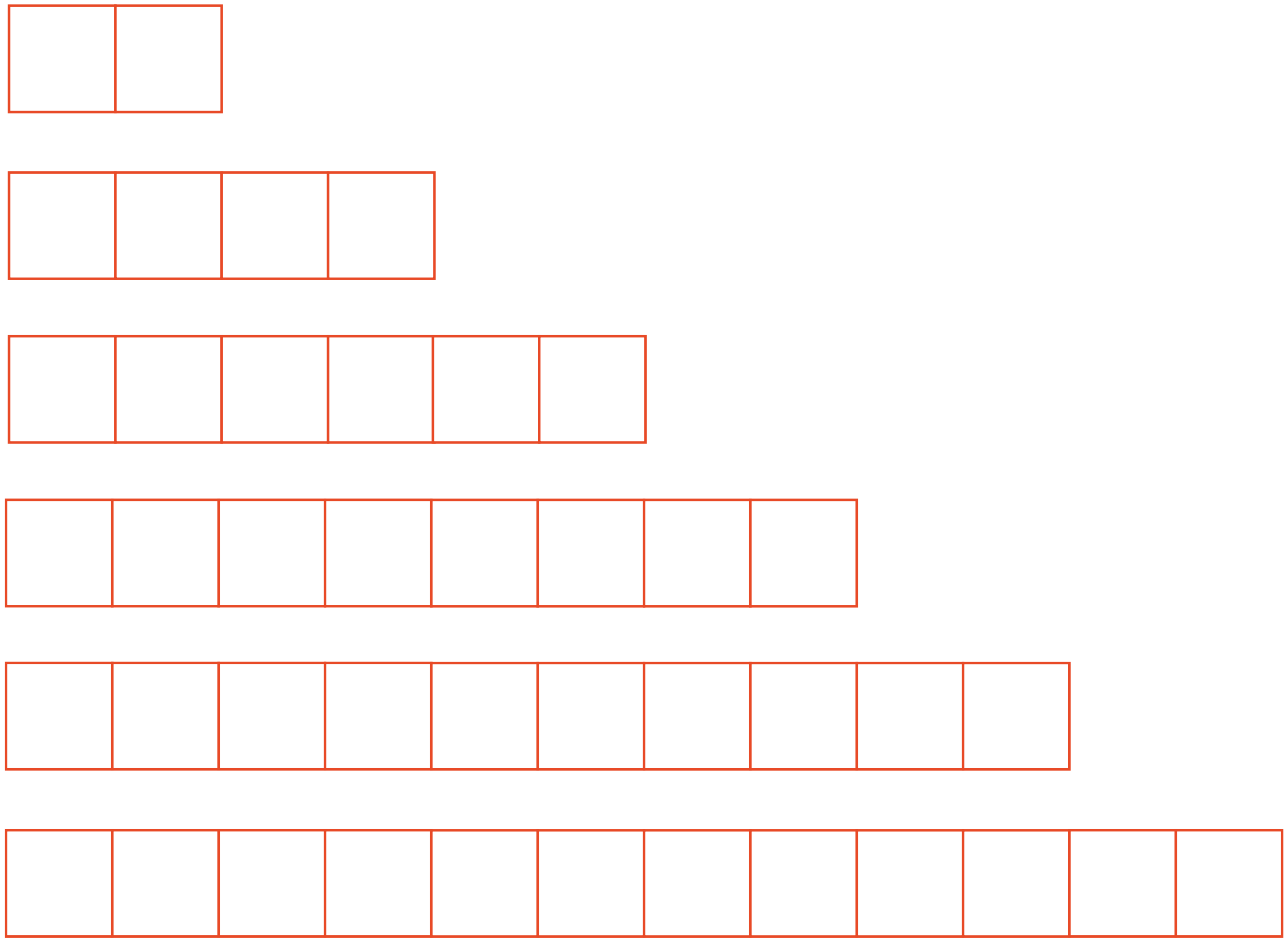
C	S	2	2	5					

Finding values is fast O(1)

Modifying (moving all values) slow O(n)

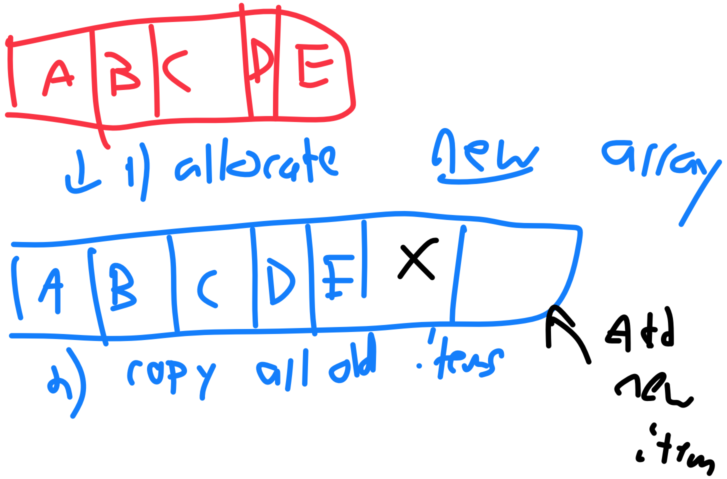
	@Front	@Back	@Index
Insert	O(n)	O(1)	O(n)
Delete	O(n)	O(1)	O(n)

Resize Strategy: +2 elements every time



Resize Strategy: +2 elements every time

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



Amortized:

Precise total work over N calls

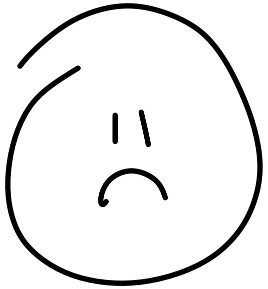
→ How much total work "cost" for a single insert?

Single insert is total/N ⇒ ~N copies per insert

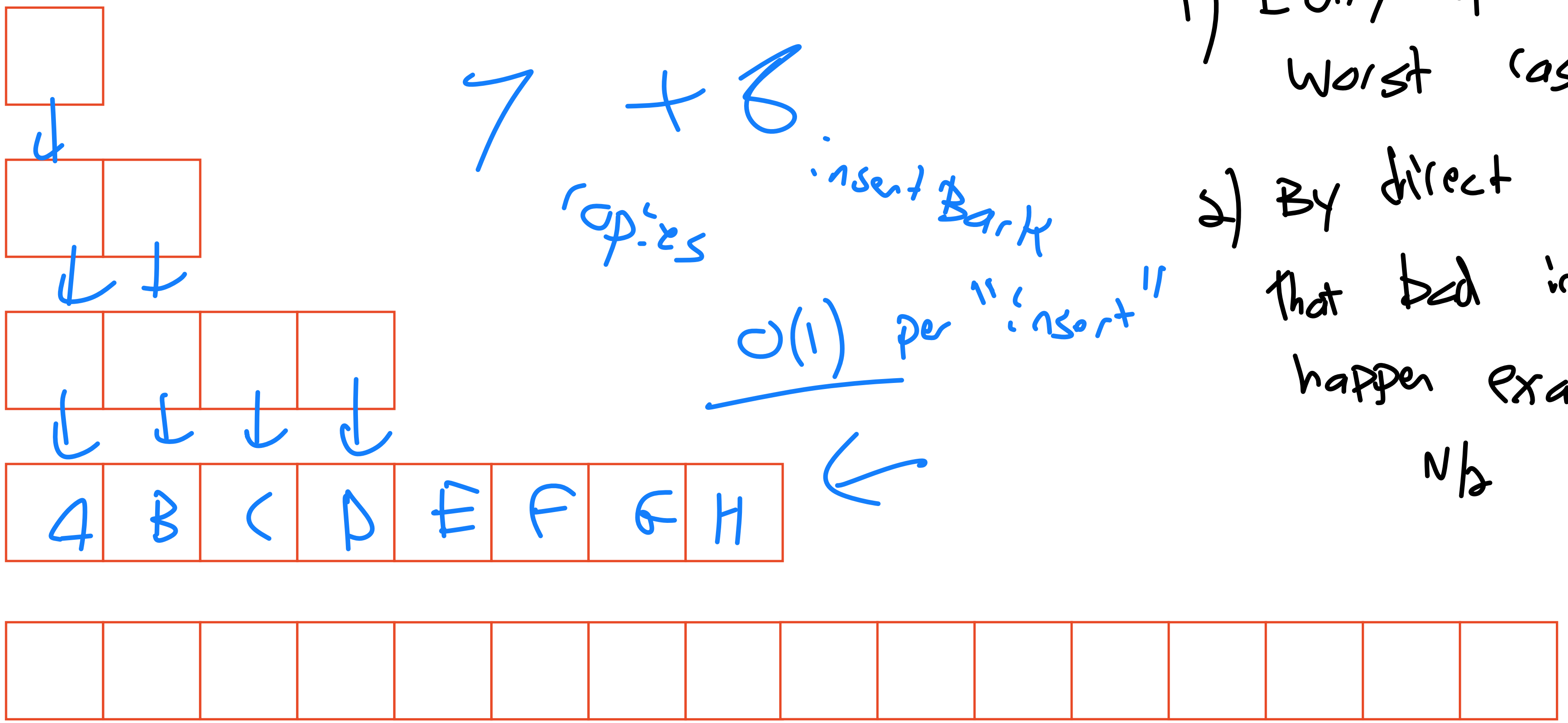
Big O:

Upperbound on worst case

O(N) for N items in list



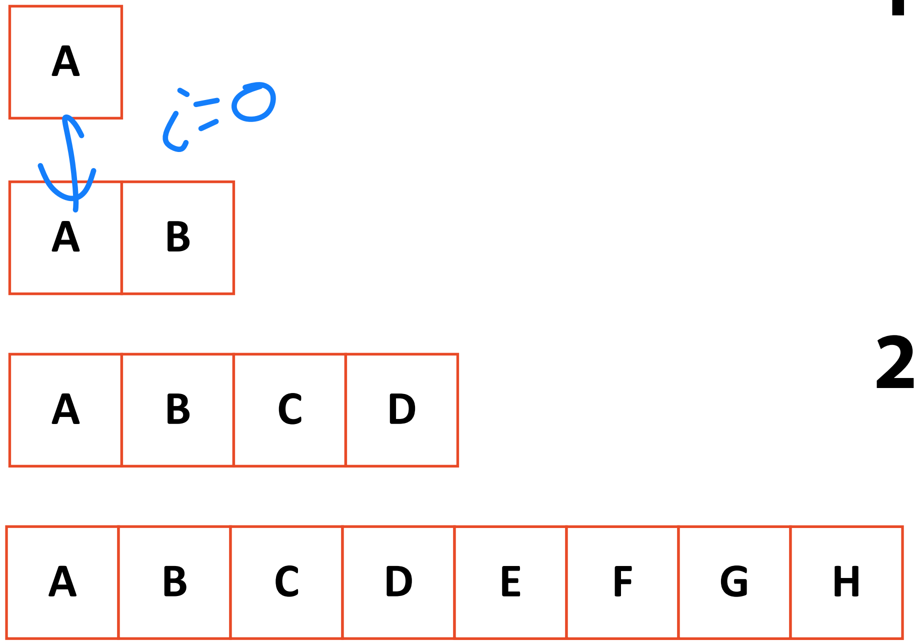
Resize Strategy: x2 elements every time



1) Every operation has worst case O(n)

2) By direct calcs we show that bad inserts [O(n)] happen exactly once after N/2 "good" [O(1)] inserts

Resize Strategy: x2 elements every time



1) How many copy calls per reallocation?

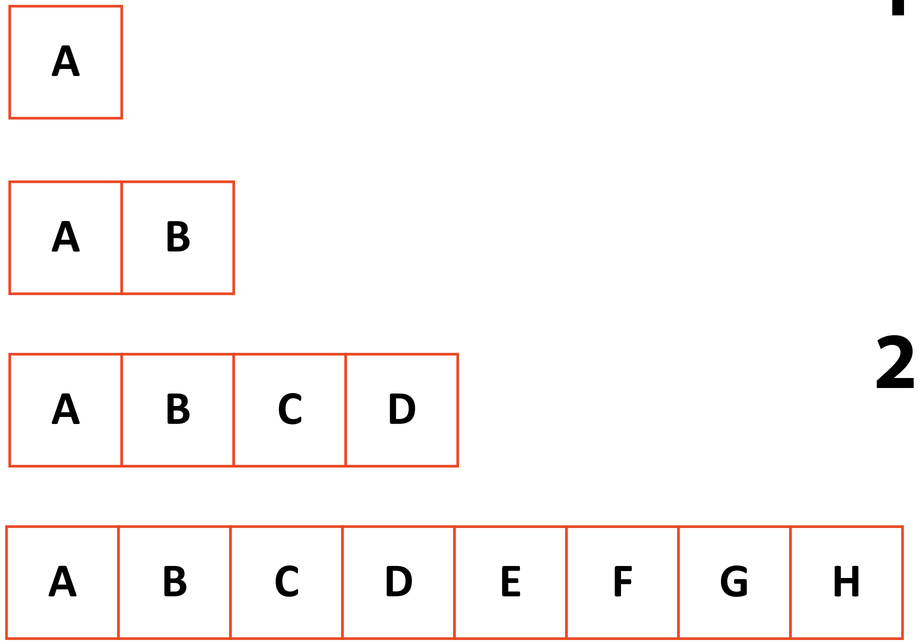
For reallocation i, 2^i copy calls are made

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 = $\lceil log_2 n \rceil$

sum of all reallocations

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

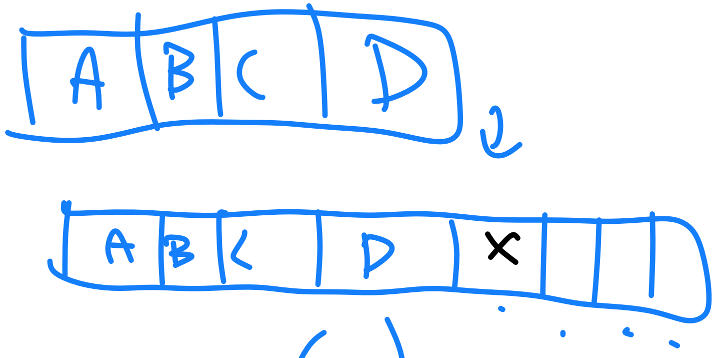
copies per realloc

… For N objects: $2n - 1$

$2^{\log_2 N + 1}$

Resize Strategy: x2 elements every time

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



amortized $O(1)^*$

Amortized: ~2 copies per insert

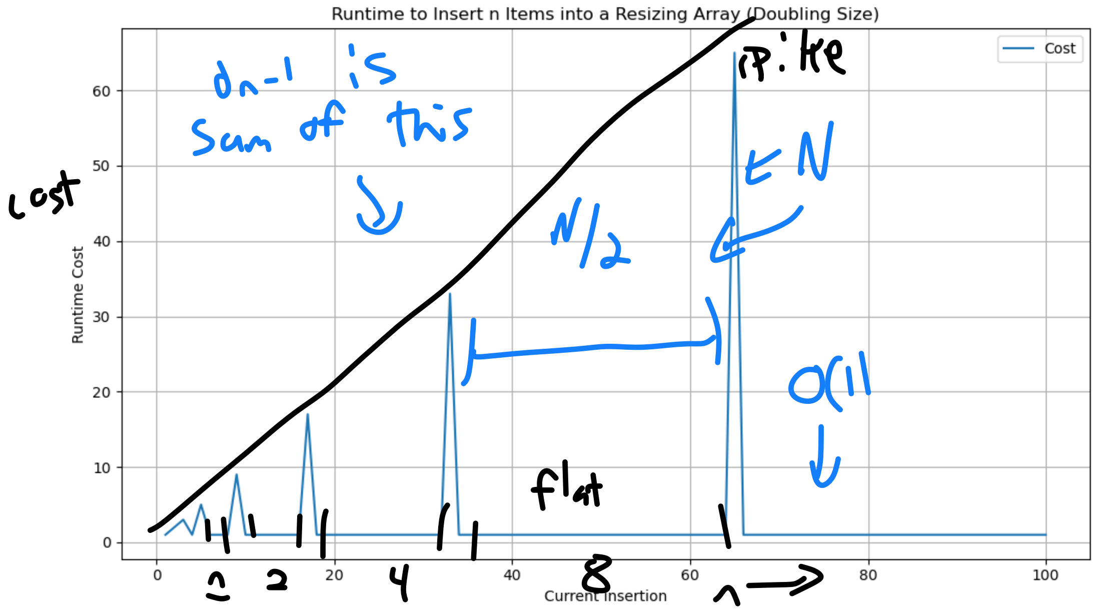
Big O: $O(1)$

Precise total work over N calls

Upperbound on worst case

$\frac{2n-1}{N}$ to get expected work for each insert

~2



List Implementation

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

Special cases:

insert Front (head)
remove

insert Back when not full
remove

Thinking critically about lists: tradeoffs

The implementations shown are foundational (simple).

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

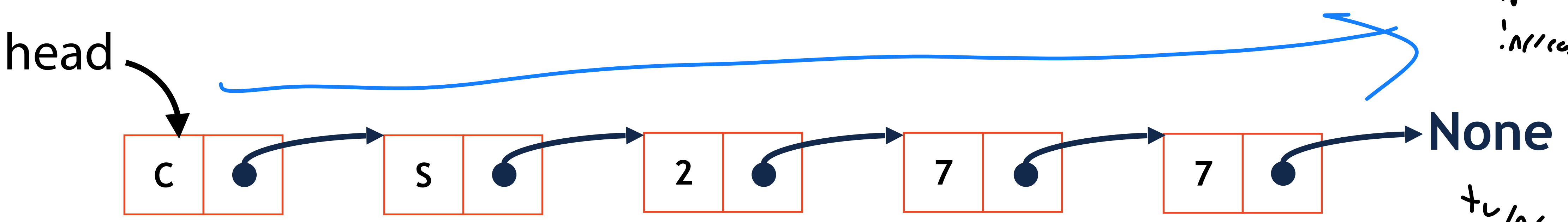
Thinking critically about lists: tradeoffs

Getting the size of a linked list has a Big O of: O(n) → O(1)

constant work

increase to insert, remove

turns O(n) into O(1)



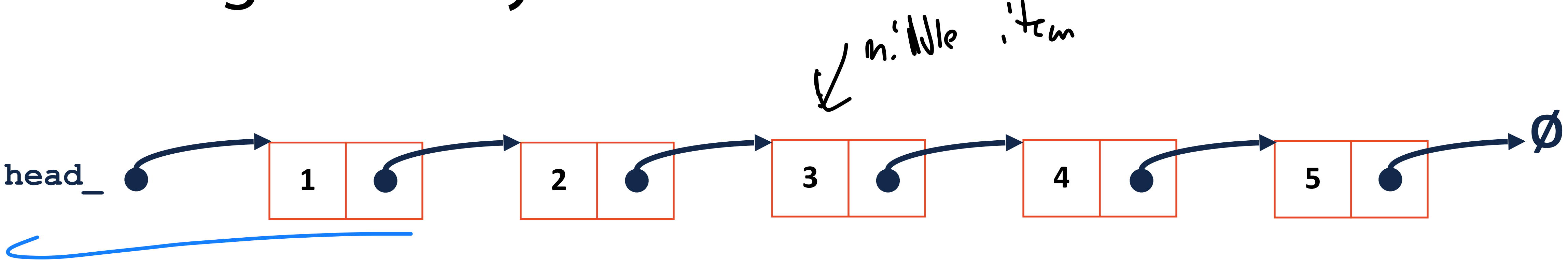
count++ →

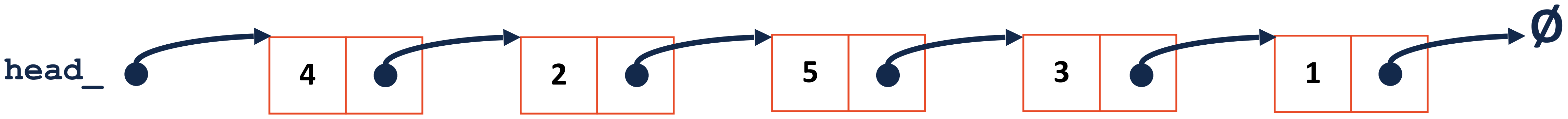
Added to LinkedList class
↳ unsigned size_;

insert
↳ size_++;

remove
↳ size_--;

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

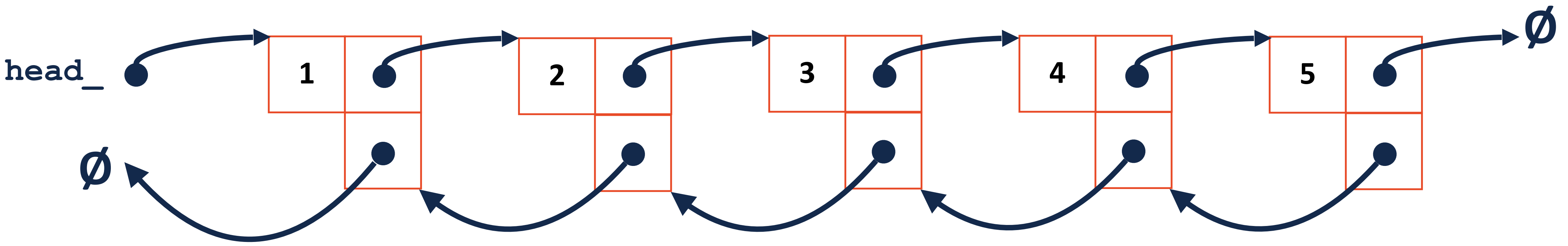
Benefit!!

Search $O(n)$ to $O(\log n)$

Cost

insertBack doesn't exist

Thinking critically about lists: tradeoffs



Doubly linked list!

Cost: storage cost!

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?

Possible if we remove our ability to do random access

Stack Data Structure

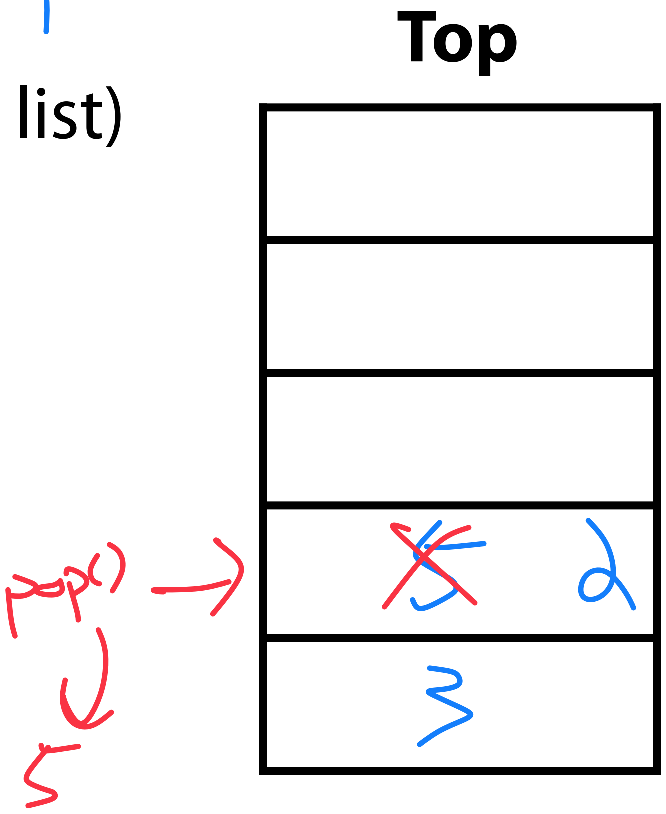
A **stack** stores an ordered collection of objects (like a list)

However you can only do two* operations:

Push: Put an item on top of the stack

Pop: Remove the top item of the stack (and return it)

Top: Look at top item



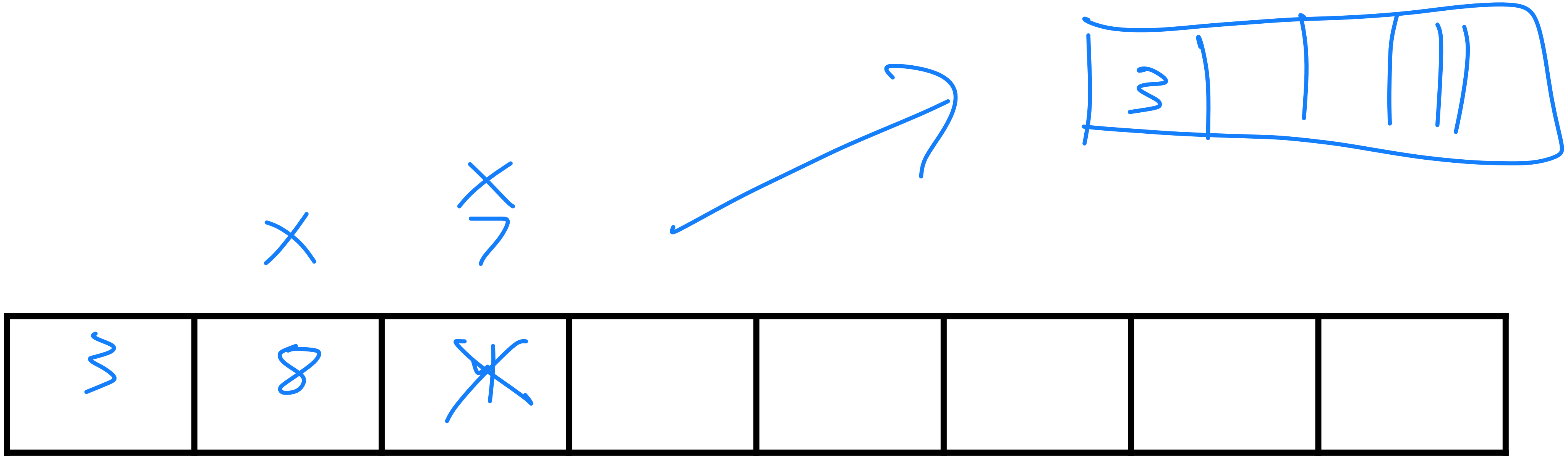

```
push(3); push(5); pop(); push(2)
```

Stack Data Structure

```
#include <stack>
int main() {
  stack<int> stack;
  stack.push(3);
  stack.push(8);
  stack.push(4);
  stack.pop();
  stack.push(7);
  stack.pop();
  stack.pop();
}
```

C++ has a built-in stack

Underlying implementation is vector or deque



Stack Data Structure

`Push(X)` is equivalent to …

A	B	C	D				

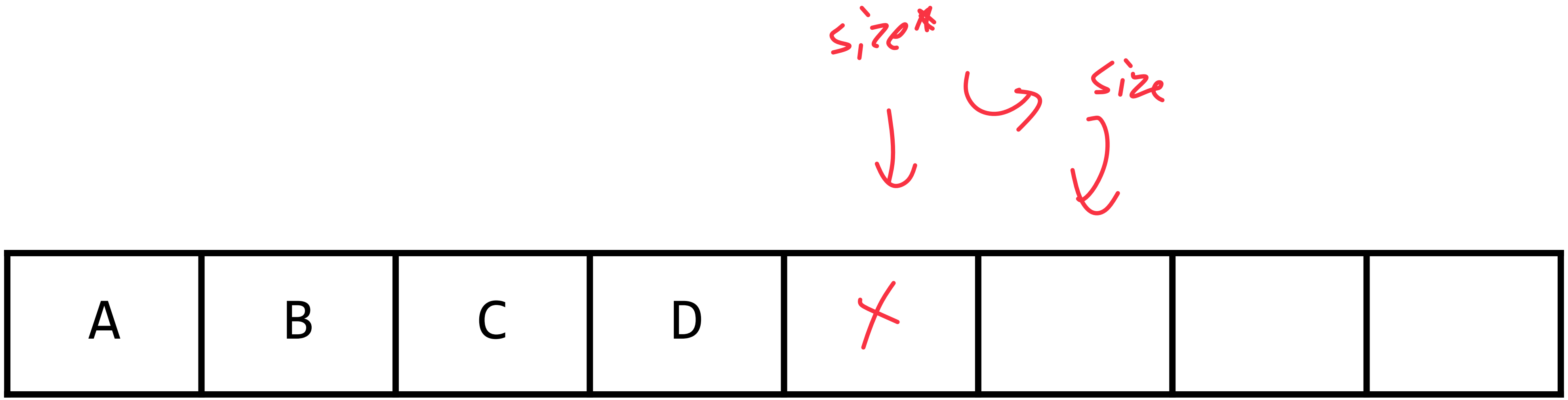
Stack Data Structure

O(1)

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

```
*size = X;

size++;
```



Stack Data Structure

Pop( )   is equivalent to…

A	B	C	D				

Stack Data Structure

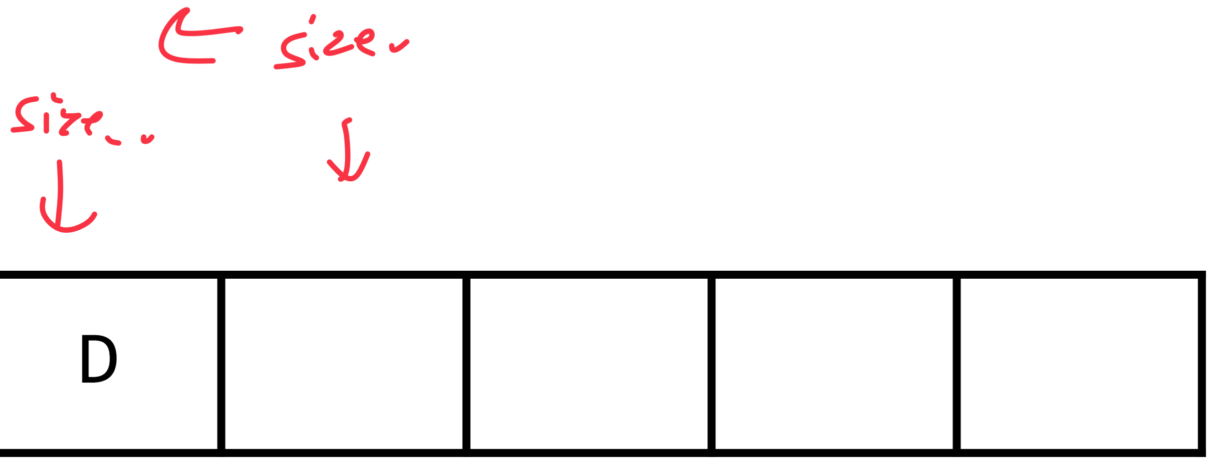
Pop() is equivalent to removeBack()

O(1)

```
size--;        O(1)

T tmp = *size; O(1)

return tmp;    O(1)
```

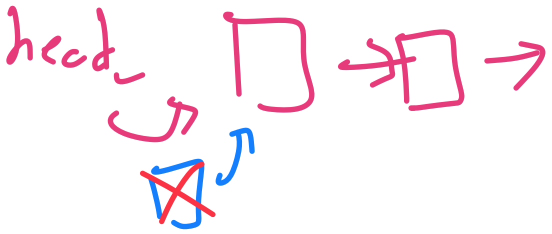


Stack ADT

- [Order]: Last in first Out (LIFO)

- [Implementation]: Trivially as an array!

Could I do this as a Linked List? Yes! insert/remove Front



This can't be full

- [Runtime]: $O(1)^*$

as long as array not full

Queue Data Structure

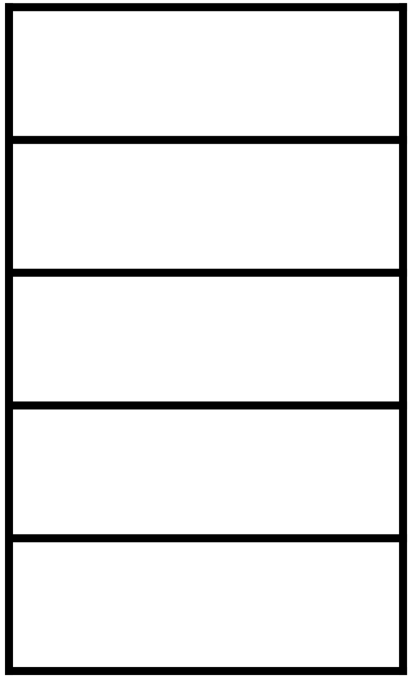
A **queue** stores an ordered collection of objects (like a list)

However you can only do two* operations:

Enqueue: Put an item at the back of the queue

Dequeue: Remove and return the front item of the queue

Front



```
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

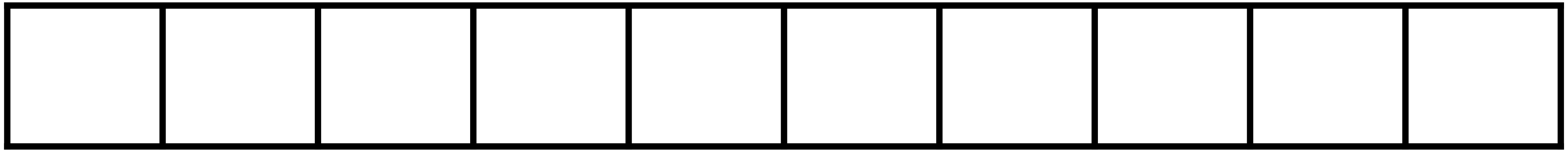
	Time x1 billion	Like
L1 cache reference	0.5 seconds	Heartbeat 💓
Main memory reference	100 seconds	Brush teeth
SSD random read	1.7 days	Weekend
Disk seek	16.5 weeks	Semester
Send packet CA->Netherlands->CA	4.8 years	Ph.D. 🎓

(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

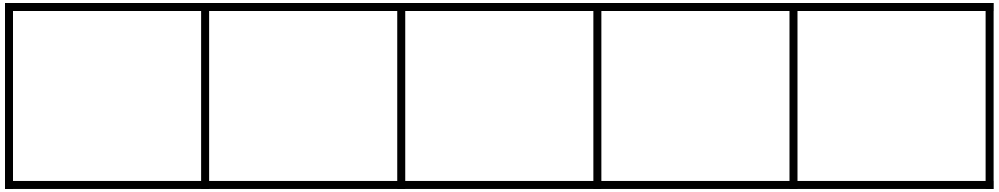
Unlike the array list, it is easier to implement a Queue using unsigned ints

Queue.h

```
#pragma once

template <typename T>
class Queue {
  public:
    void enqueue(T e);
    T dequeue();
    bool isEmpty();

  private:
    T *data_;
    unsigned size_;
    unsigned capacity_;
    unsigned front_;
};
```



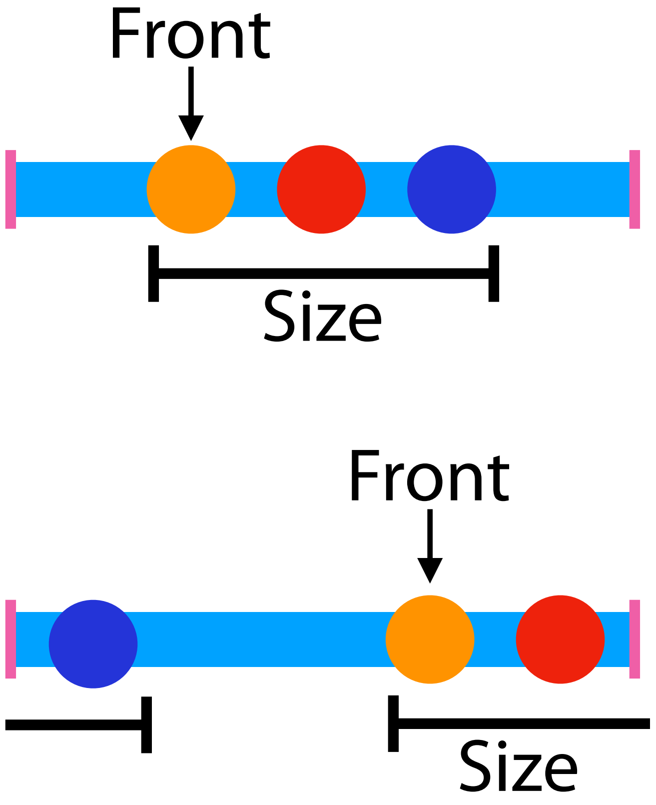
(Circular) Queue Data Structure

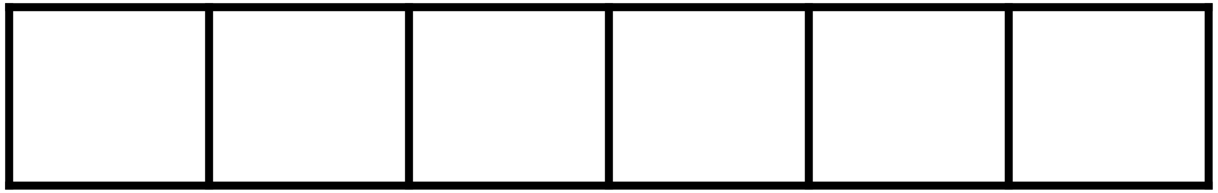
Queue.h

```
#pragma once

template <typename T>
class Queue {
  public:
    void enqueue(T e);
    T dequeue();
    bool isEmpty();

  private:
    T *data_;
    unsigned capacity_;
    unsigned size_;
    unsigned front_;
};
```





Enqueue(D):

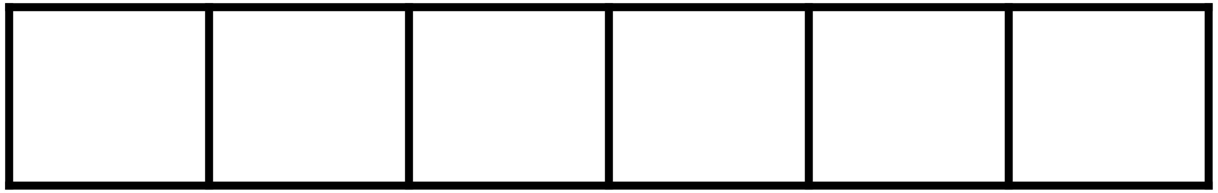
Dequeue():

Size:

Front:

```
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);
```

Capacity:



Enqueue(D): Insert @ (size+front) % capacity
size++ until size == capacity

Dequeue(): Remove @front
front = (front+1) % capacity
size--

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);
```

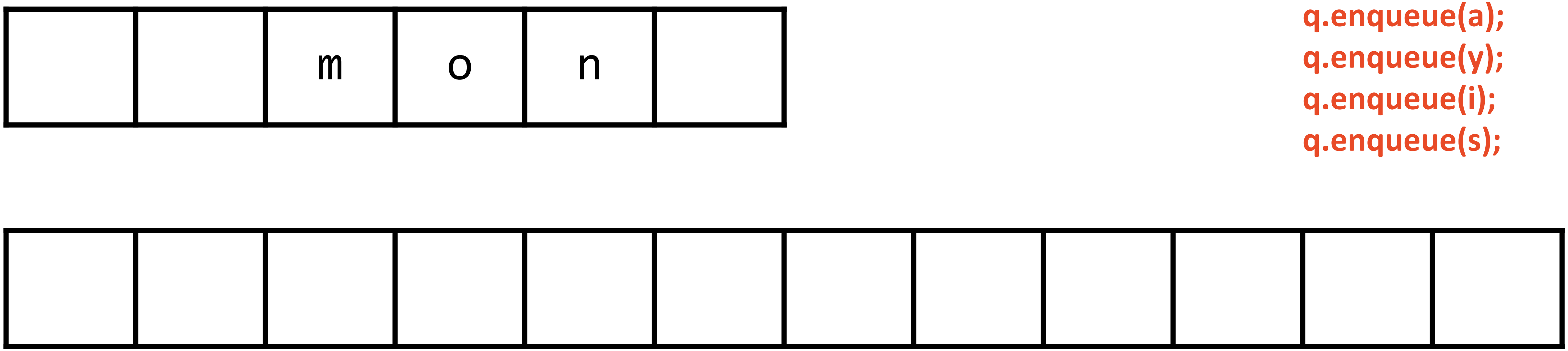
Queue Data Structure: Resizing

		m	o	n	

```
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

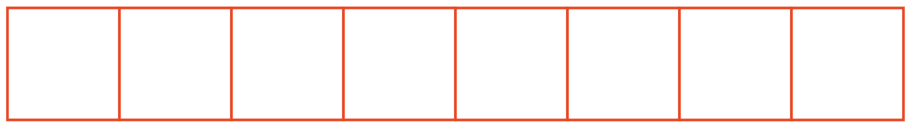
- [Order]:

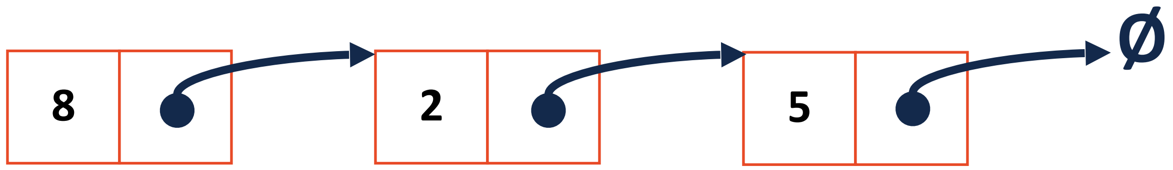
- [Implementation]:

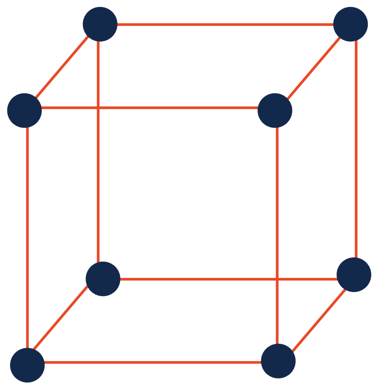
- [Runtime]:

Iterators

We want to be able to loop through all elements for any underlying implementation in a systematic way

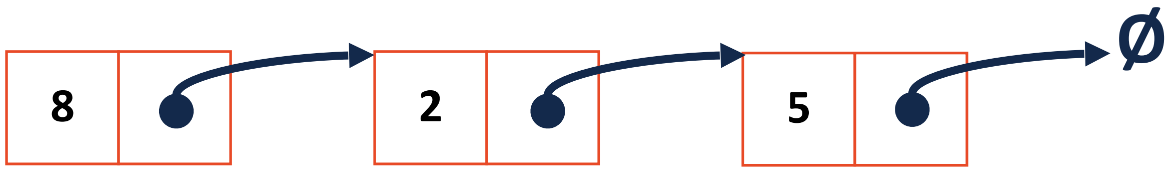


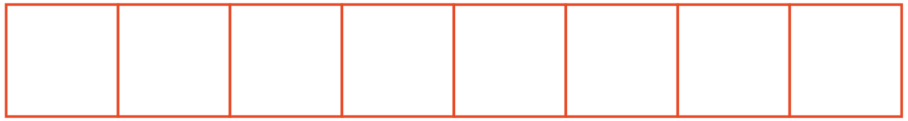


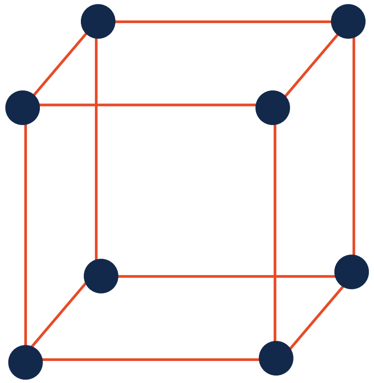


Iterators

We want to be able to loop through all elements for any underlying implementation in a systematic way



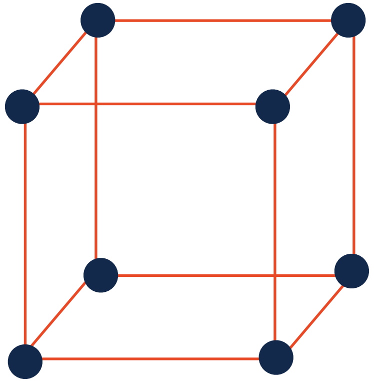




Cur. Location	Cur. Data	Next
`ListNode * curr`		
`unsigned index`		
`Some form of (x, y, z)`		

Iterators

Iterators provide a way to access items in a container without exposing the underlying structure of the container



```
Cube::Iterator start = myCube.begin();

while (it != myCube.end()) {
     std::cout << *it << " ";
     it++;
}

```

Iterators

For a class to implement an iterator, it needs two functions:

```
Iterator begin()
```

```
Iterator end()
```

Iterators

The actual iterator is defined as a class **inside** the outer class:

1. It must be of base class **std::iterator**

2. It must implement at least the following operations:

```
Iterator& operator ++()

const T & operator *()

bool operator !=(const Iterator &)
```

Iterators

Here is a (truncated) example of an iterator:

```
template <class T>
class List {

    class ListIterator : public std::iterator<std::bidirectional_iterator_tag, T> {
      public:

        ListIterator& operator++();

        ListIterator& operator--()

        bool operator!=(const ListIterator& rhs);

        const T& operator*();
    };

    ListIterator begin() const;

    ListIterator end() const;
};
```

stlList.cpp

```
#include <list>
#include <string>
#include <iostream>

struct Animal {
  std::string name, food;
  bool big;
  Animal(std::string name = "blob", std::string food = "you", bool big = true) :
    name(name), food(food), big(big) { /* nothing */ }
};

int main() {
  Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
  std::vector<Animal> zoo;

  zoo.push_back(g);
  zoo.push_back(p);   // std::vector’s insertAtEnd
  zoo.push_back(b);

  for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }

  return 0;
}
```

```
std::vector<Animal> zoo;


/* Full text snippet */

  for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
    std::cout << (*it).name << " " << (*it).food << std::endl;
  }


/* Auto Snippet */

  for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
    std::cout << animal.name << " " << animal.food << std::endl;
  }

/* For Each Snippet */

  for ( const Animal & animal : zoo ) {
    std::cout << animal.name << " " << animal.food << std::endl;
  }
```

Trees

"The most important non-linear data structure in computer science."

- David Knuth, The Art of Programming, Vol. 1

A tree is:

-
-