# Data Structures

# Stacks and Queues

CS 225

Brad Solomon & G Carl Evans

September 8, 2023

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

# Reminder: Labs are due on Sunday

# Learning Objectives

Introduce the stack and the queue data structure

Introduce and explore iterators

# 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:

unsorted

| 2 | 7 | 5 | 9 | 7 | 14 | 1 | 0 | 8 | 3 |
|---|---|---|---|---|---|---|---|---|---|

Sorted array :: (search)

| 0 | 1 | 2 | 3 | 5 | 7 | 7 | 8 | 9 | 14 |
|---|---|---|---|---|---|---|---|---|---|

↳ time to sort ↳ some D.S. don't use sorting well

# Thinking critically about lists: tradeoffs

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

↳ can only add (remove) access at specific place

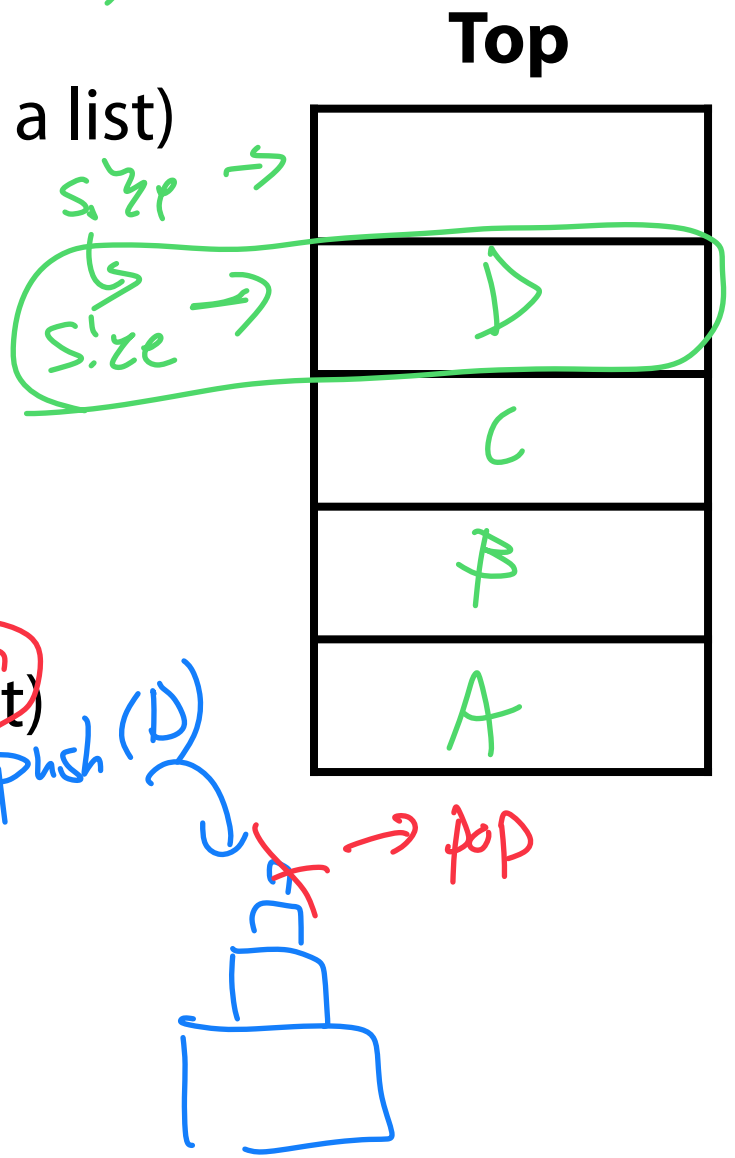
# Stack Data Structure

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)

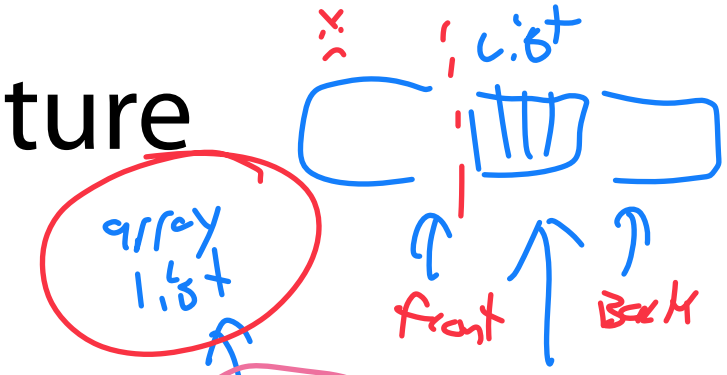



```
push(3); push(5); pop(); push(2)
```

# Stack Data Structure

C++ has a built-in stack

Underlying implementation is vector or deque




↳ Best case array is insert/remove back!

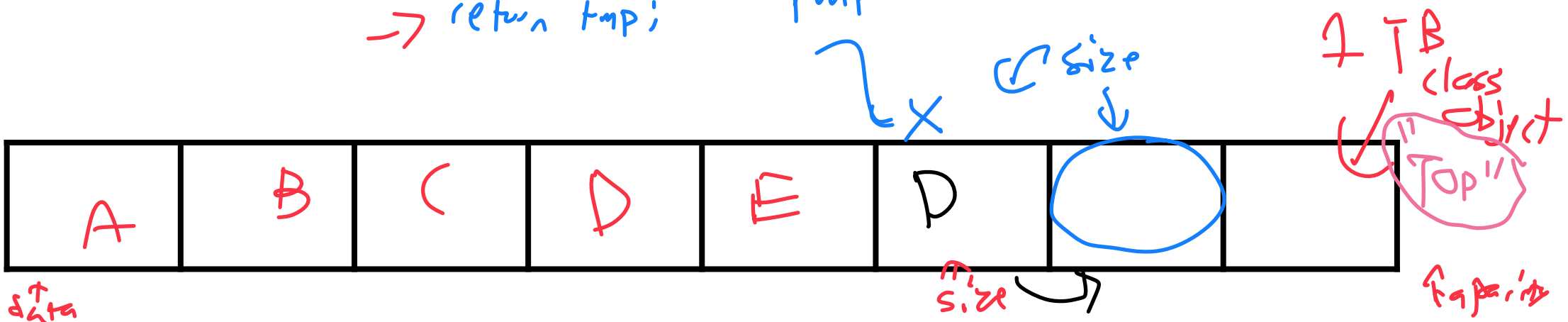
1) InsertBack(D) / Push()

↳ *size = D;
size++;

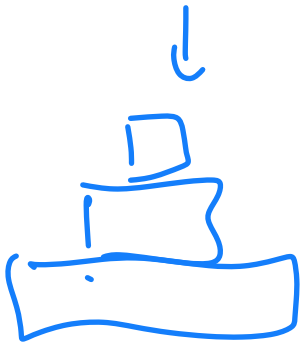
2) RemoveBack() / Pop()

size--;
T tmp = *size;
→ remove size; // *size = null
→ return tmp;

```
1  #include <stack>
2  int main() {
3    stack<int> stack;
4
5    stack.push(3);
6    stack.push(8);
7    stack.push(4);
8    stack.pop();
9    stack.push(7);
10   stack.pop();
11
12   stack.pop();
13   stack.push(2);
14   stack.push(1);
15   stack.push(3);
16   stack.push(5);
17
18   stack.pop();
19   stack.push(9);
20 }
```

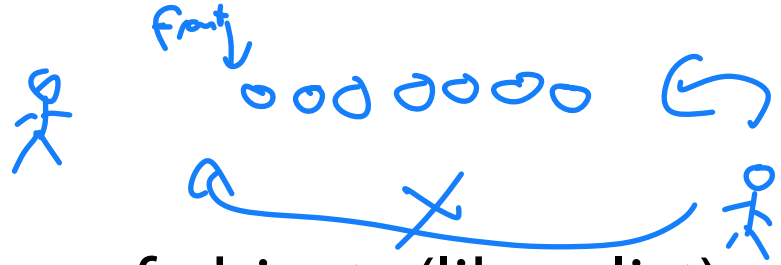


# Stack ADT



- [Order]: last in first out (LIFO)
- [Implementation]: vector / deque ← C++
  LL or array list
- [Runtime]: push() O(1)
  pop() O(1)

# Queue Data Structure

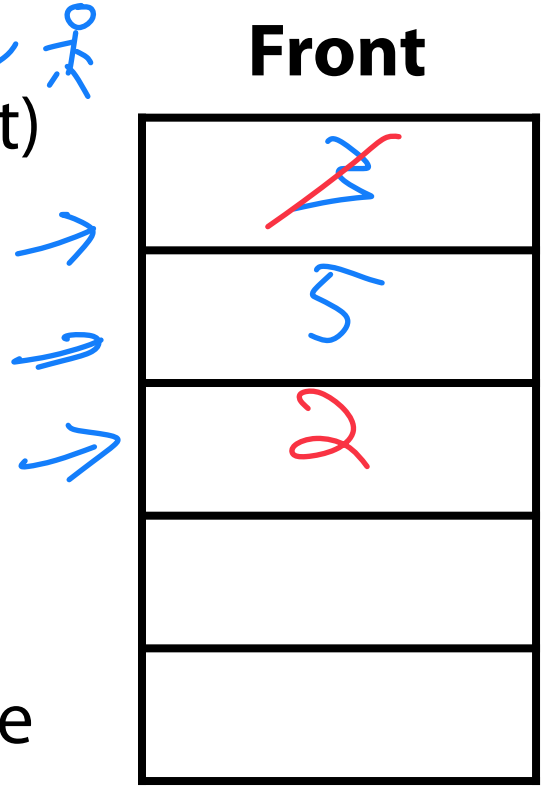



A **queue** stores an ordered collection of objects (like a list)

However you can only do two operations:

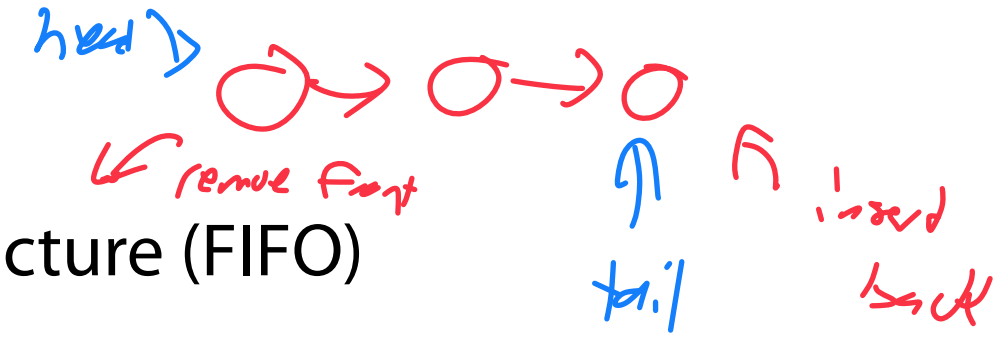
**Enqueue**: Put an item at the back of the queue

**Dequeue**: Remove and return the front item of the queue




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

# Queue Data Structure



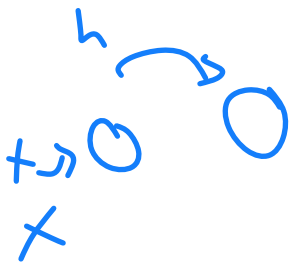

The queue is a **first in — first out** data structure (FIFO)

What data structure excels at removing from the front?

Linked List

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

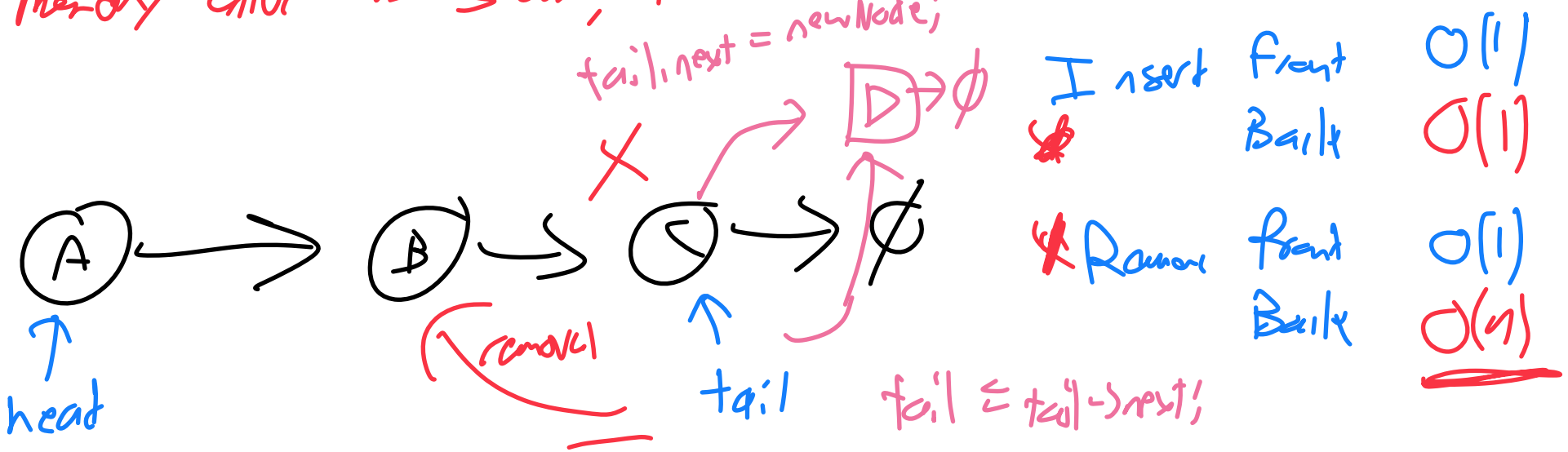
↳ By adding a tail pointer to last node




# Queue Data Structure

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

1) the array uses continous memory the LL does not!
2) LL is more prone to memory leaks
   ↳ LL has many more pointers 1 per object
3) memory allocation is slow, do it once




| | | |
|---|---|---|
| Insert | Front | O(1) |
| | Back | O(1) |
| Remove | Front | O(1) |
| | Back | O(n) |

# 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)

# Queue Data Structure

Array [ Data, size, capacity ]

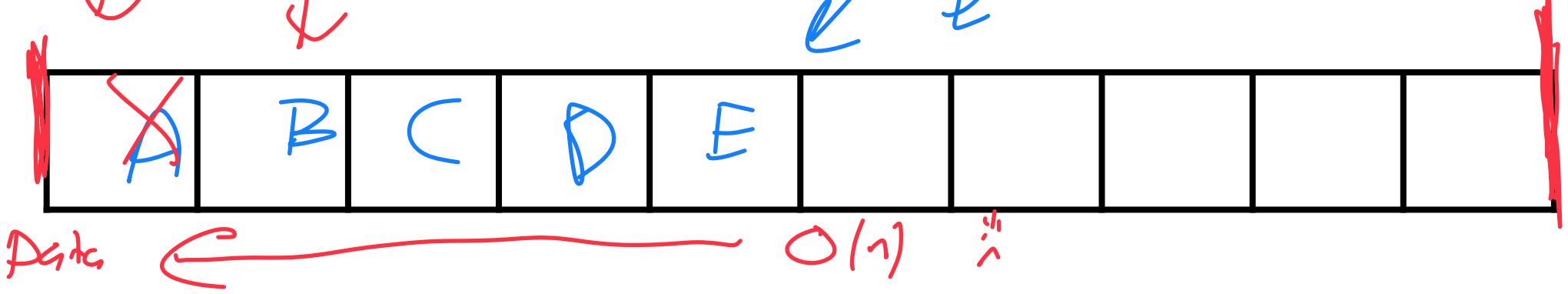
What do we need to track to maintain a queue with an array list?

1) Let's make size / capacity / front unsigned integers

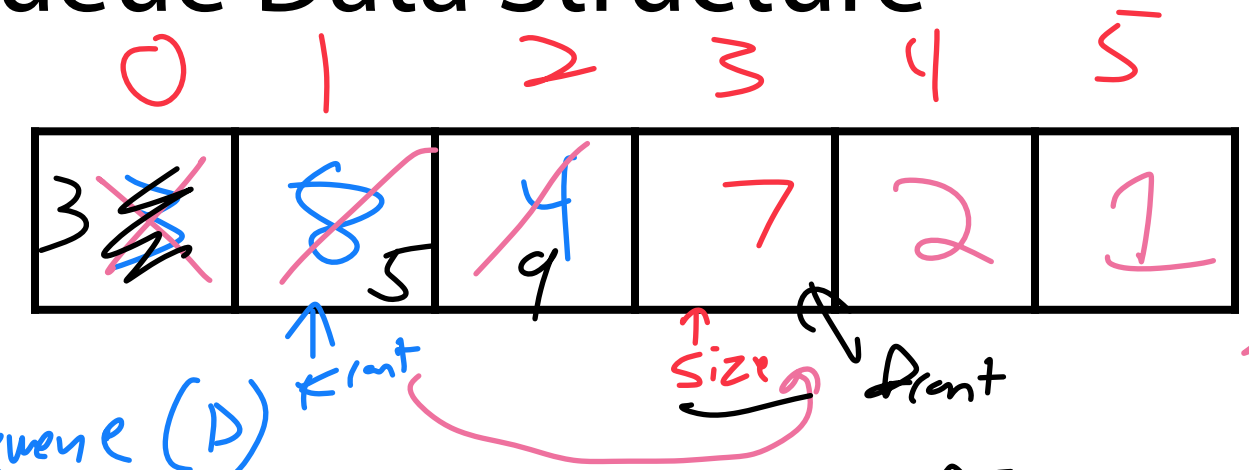
Front

Shift or leave back on remove

size++



# Queue Data Structure




Enqueue (D)
↳ add D at location (size + front) % capacity
↳ size ++; % capacity

Dequeue()
↳ remove & return at location front
↳ front ++; % capacity

Size: 0 1 2 3

Front: 0 1

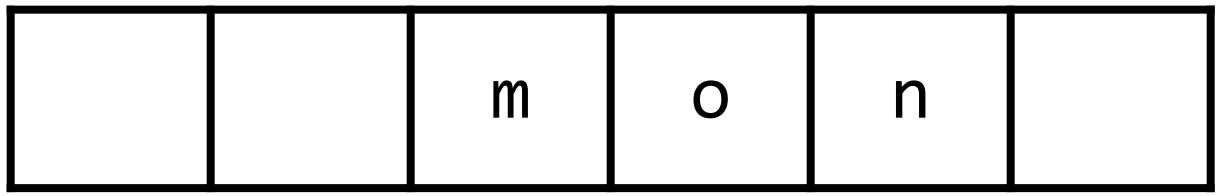
```
Queue<int> q;
q.enqueue(3);
q.enqueue(8);
q.enqueue(4);
q.dequeue();
q.enqueue(7);
q.dequeue();
q.dequeue();
q.enqueue(2);
q.enqueue(1);
q.enqueue(3);
q.enqueue(5);
q.dequeue();
q.enqueue(9);
```

Enque (x):

**After Class Note:**
**I made a big mistake here!**
**Can you spot it before Monday?**

Capacity: 6

# Queue Data Structure: Resizing




reallocate

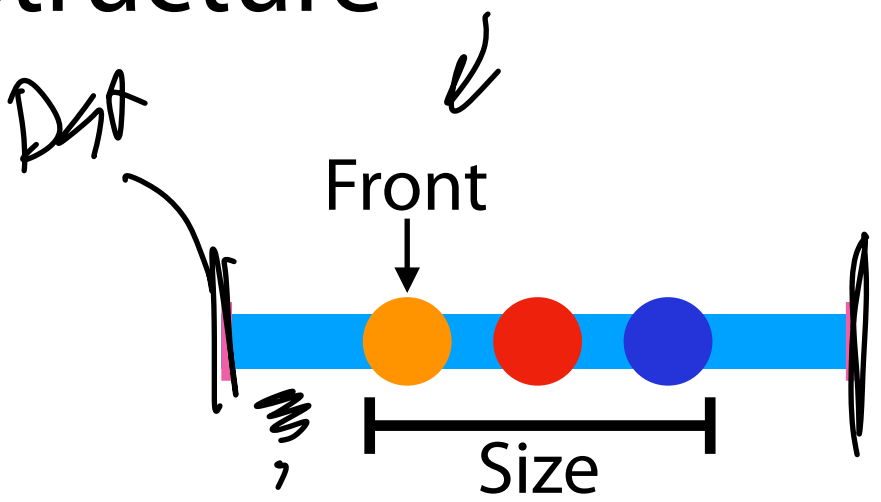
Do an Madam!

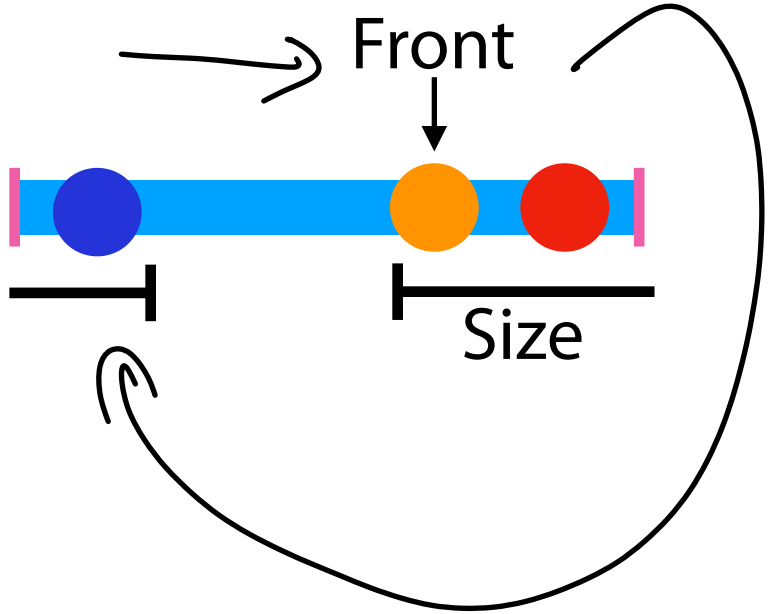
```
Queue<char> q;
…
q.enqueue(m);
q.enqueue(o);
q.enqueue(n);
…
q.enqueue(d);
q.enqueue(a);
q.enqueue(y);
q.enqueue(i);
q.enqueue(s);
q.dequeue();
q.enqueue(h);
q.enqueue(a);
```

# (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 };
```



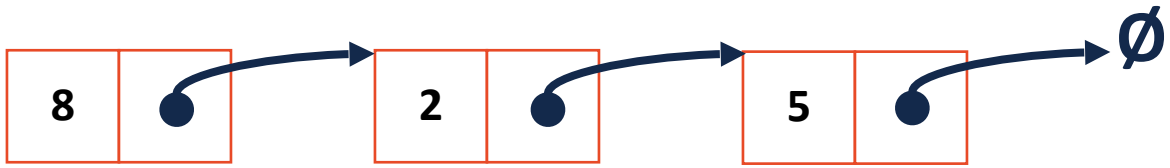



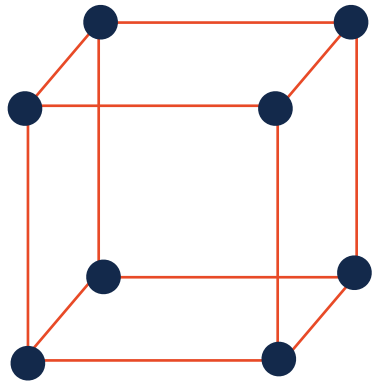
# Queue ADT

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

# Iterators

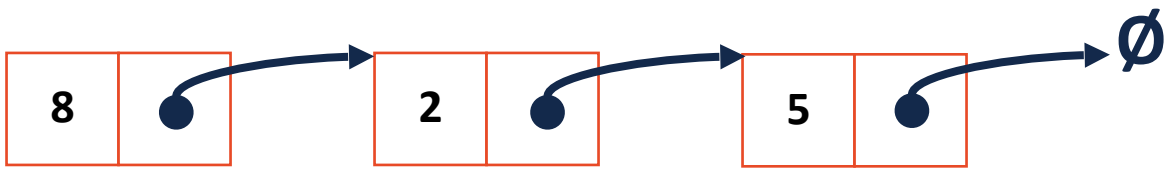
We want to be able to loop through all elements for any underlying implementation in a systematic way

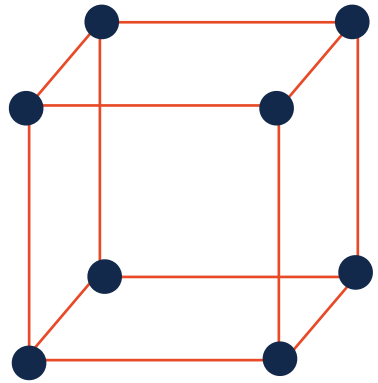


# Iterators

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



| Cur. Location | Cur. Data | Next |
|---|---|---|
| `ListNode *` | | |
| `index` | | |
| `(x, y, z)` | | |

# 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

Future assignments will have you write custom iterators:

```
 1 template <class T>
 2 class List {
 3
 4     class ListIterator : public
   std::iterator<std::bidirectional_iterator_tag, T> {
 5       public:
 6
 7         ListIterator& operator++();
 8
 9         ListIterator& operator--()
10
11         bool operator!=(const ListIterator& rhs);
12
13         const T& operator*();
14     };
15
16     ListIterator begin() const;
17
18     ListIterator end() const;
19 };
```

# stlList.cpp

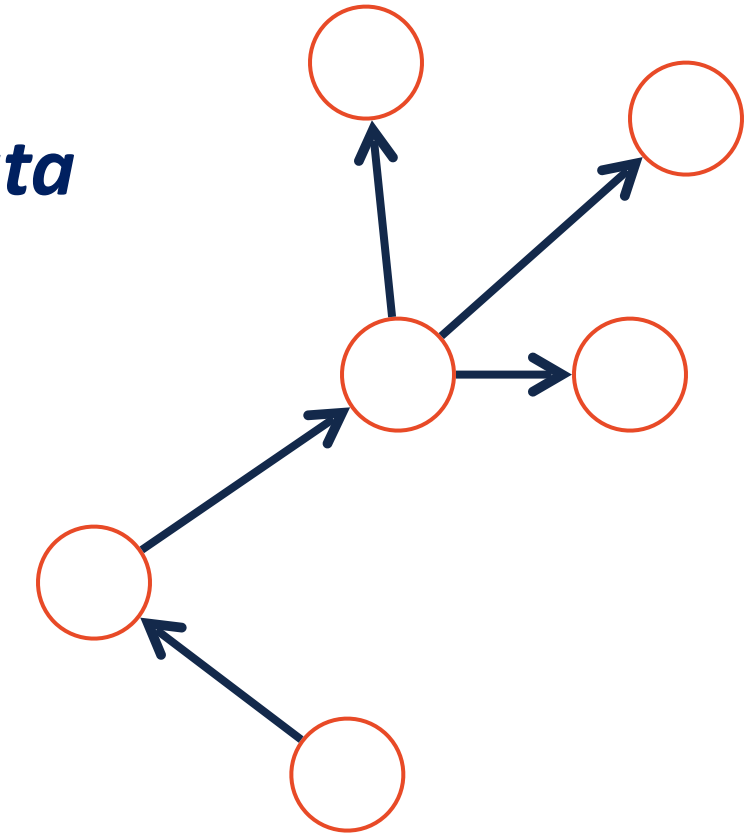
```
1  #include <list>
2  #include <string>
3  #include <iostream>
4
5  struct Animal {
6    std::string name, food;
7    bool big;
8    Animal(std::string name = "blob", std::string food = "you", bool big = true) :
9      name(name), food(food), big(big) { /* nothing */ }
10 };
11
12 int main() {
13   Animal g("giraffe", "leaves", true), p("penguin", "fish", false), b("bear");
14   std::vector<Animal> zoo;
15
16   zoo.push_back(g);
17   zoo.push_back(p);   // std::vector's insertAtEnd
18   zoo.push_back(b);
19
20   for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
21     std::cout << (*it).name << " " << (*it).food << std::endl;
22   }
23
24   return 0;
25 }
```

```
1
2  std::vector<Animal> zoo;
3
4
5  /* Full text snippet */
6
7    for ( std::vector<Animal>::iterator it = zoo.begin(); it != zoo.end(); ++it ) {
8      std::cout << (*it).name << " " << (*it).food << std::endl;
9    }
10
11
12 /* Auto Snippet */
13
14   for ( auto it = zoo.begin(); it != zoo.end; ++it ) {
15     std::cout << animal.name << " " << animal.food << std::endl;
16   }
17
18 /* For Each Snippet */
19
20   for ( const Animal & animal : zoo ) {
21     std::cout << animal.name << " " << animal.food << std::endl;
22   }
23
24
25
```

# Trees

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

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



**A tree is:**

- 
- 