

# Data Structures

## Stacks and Queues

CS 225

February 4, 2026

Brad Solomon



UNIVERSITY OF  
**ILLINOIS**  
URBANA - CHAMPAIGN

Department of Computer Science



# Revamp

*Agentic AI Lab*

*Intelligent by Design. Autonomous by Nature*

# About Us

Revamp is a new implementation lab at UIUC focused on building at the edge: Agentic AI Architecture and Autonomous Systems. We are a group of builders who architect and ship production-grade multi-agent workflows for real-world clients.

If you want to move past the theory and start engineering AI systems that actually work, this is where you belong.

We are looking for SWE's, Systems Designers, and Marketing wizards ready to build at scale.

Feel free to reach out at [revampuiuc@gmail.com](mailto:revampuiuc@gmail.com) if you have any questions



Interest Form



Application Form

# Exam 1 (2/09 — 2/11)

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/>

# 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

Consider extensions to lists (data structure tradeoffs)

Introduce the stack and the queue data structure

Introduce and explore iterators

# List Implementation



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

Special Cases:

insertFront

insertBack (not full)

# Thinking critically about lists: tradeoffs

The implementations shown are foundational (simple).

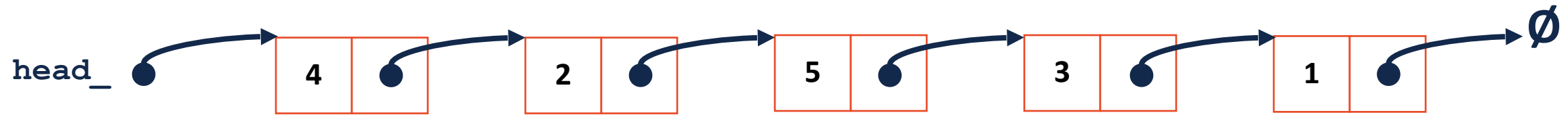
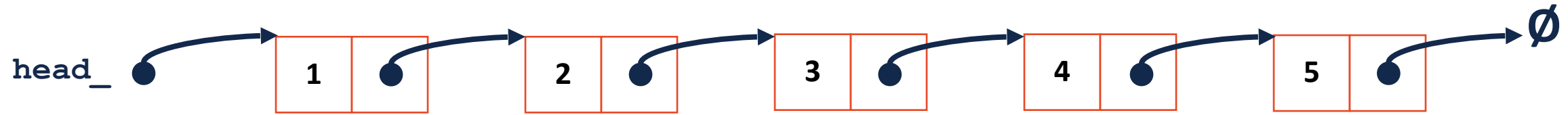
Can we make our lists better at some things? What is the cost?

# Thinking critically about lists: tradeoffs

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



# Thinking critically about lists: tradeoffs

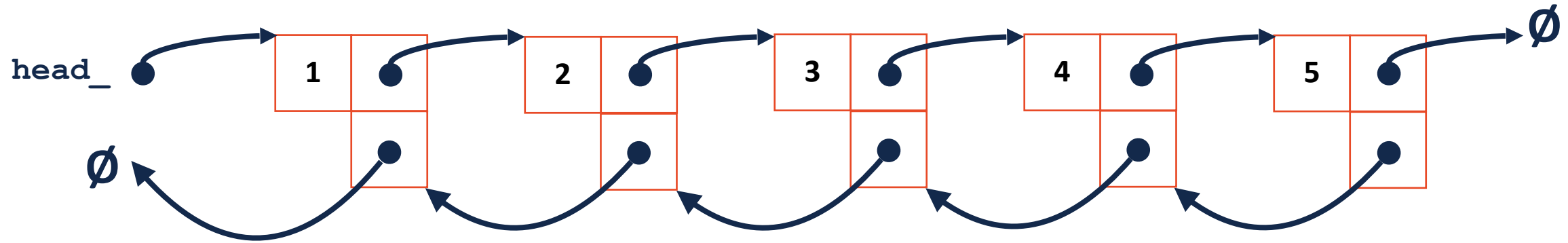


# Thinking critically about lists: tradeoffs

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

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?

# Stack Data Structure

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

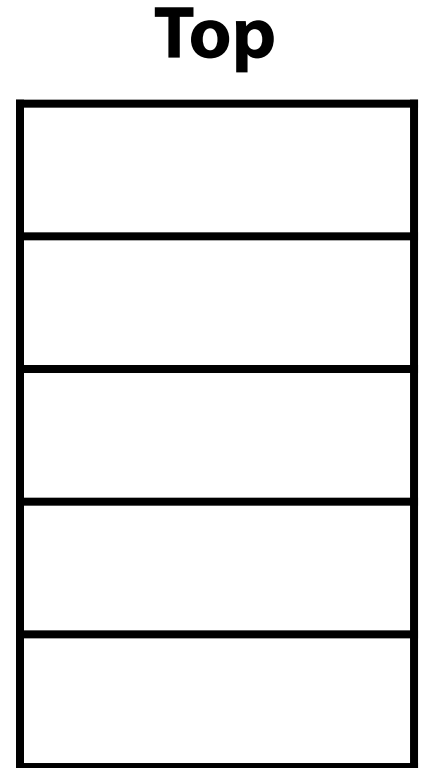
However you can only do three\* operations:

**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)
```

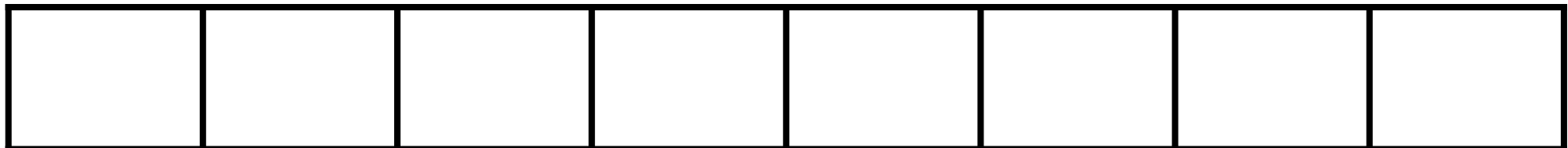


# Stack Data Structure

C++ has a built-in stack

Underlying implementation is vector or deque

```
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 ...

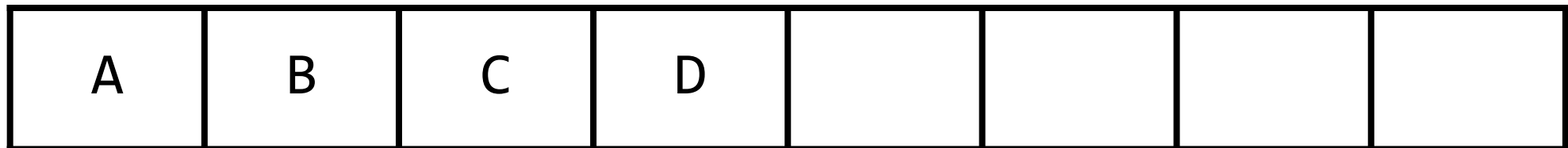


# Stack Data Structure

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

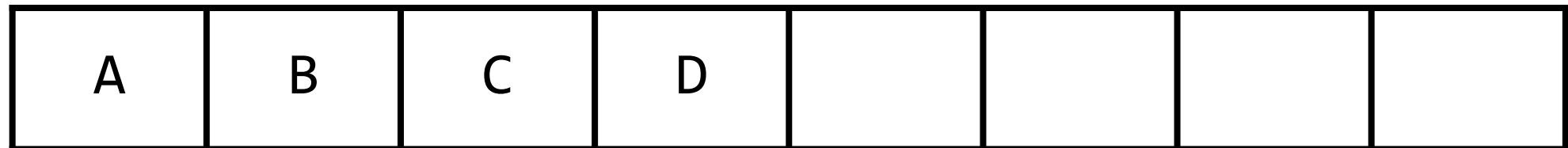
```
*size = X;
```

```
size++;
```



# Stack Data Structure

Pop() is equivalent to...

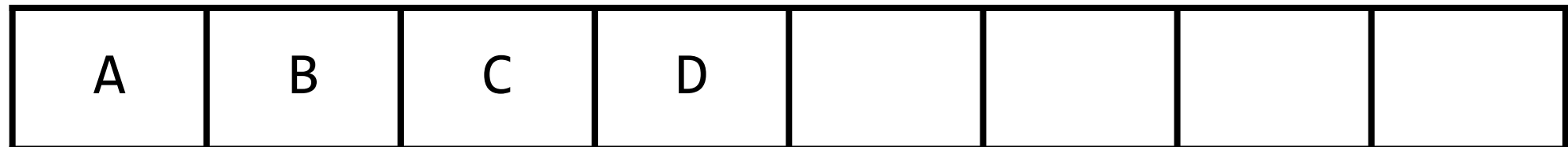


# Stack Data Structure

Pop() is equivalent to removeBack()

```
size--;
```

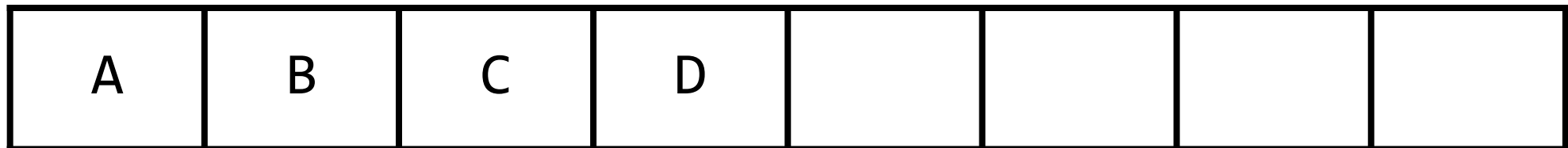
```
remove(size);
```



# Stack Data Structure

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

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



# Stack ADT

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



# Queue Data Structure

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

However you can only do three\* operations:

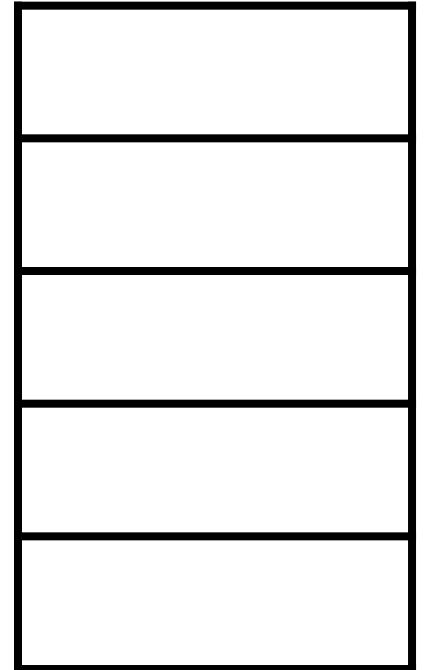
**Enqueue:** Put an item at the back of the queue

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

**Front:** Return the front item of the queue

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

**Front**



# 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?

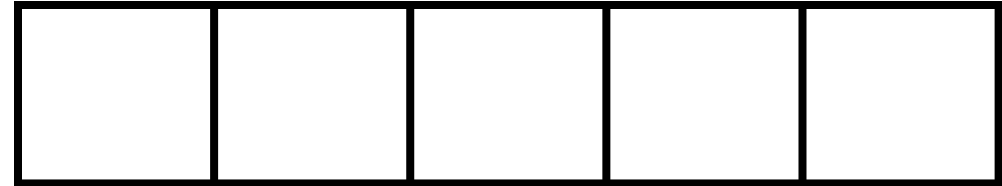


# Queue Data Structure

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 };
```

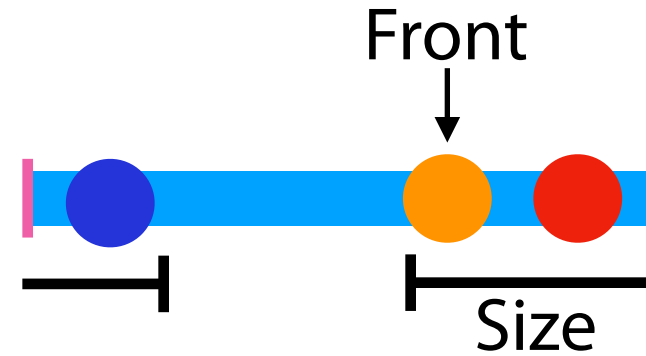
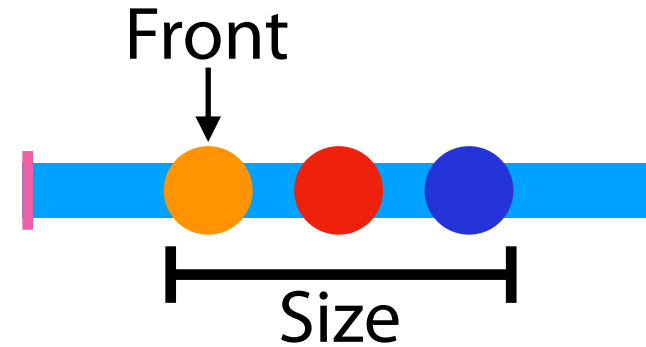


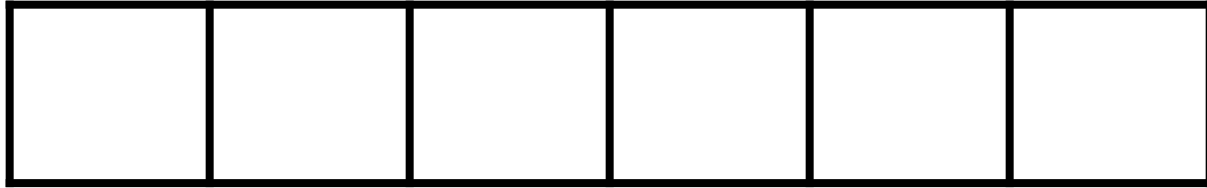
# (Circular) Queue Data Structure



## 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:**

```
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):** Add data to 'back' of queue

Insert D at index  **$(\text{size} + \text{front}) \% \text{capacity}$**

**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<int> q;
```

```
...
```

```
q.enqueue(D);
```

```
q.dequeue();
```

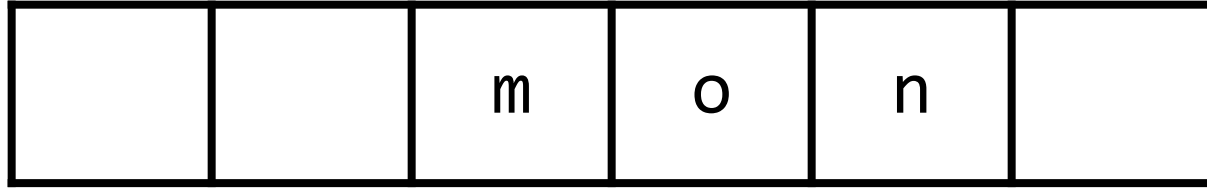
```
q.dequeue();
```

```
q.dequeue();
```

```
q.dequeue();
```

```
q.enqueue(E);
```

# Queue Data Structure: Resizing



```
Queue<char> q;
```

```
...
```

```
q.enqueue(d);
```

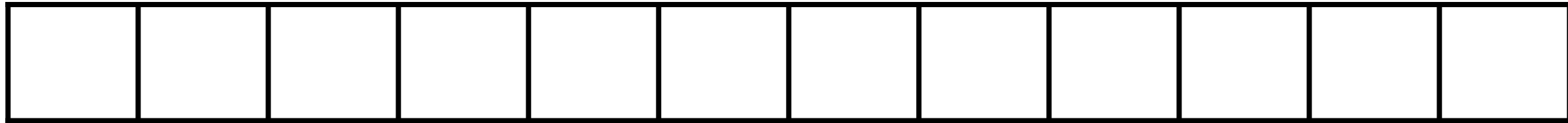
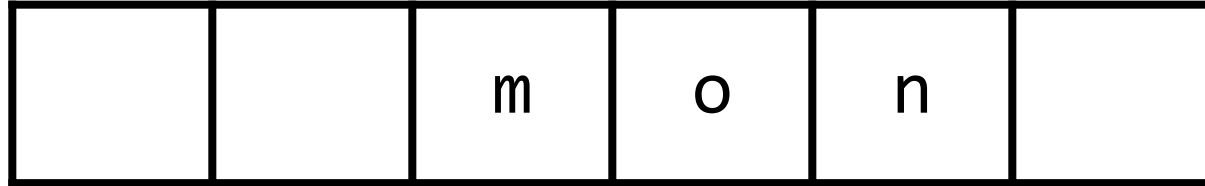
```
q.enqueue(a);
```

```
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);
```

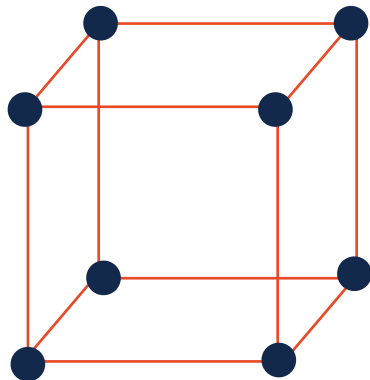
# Queue ADT



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

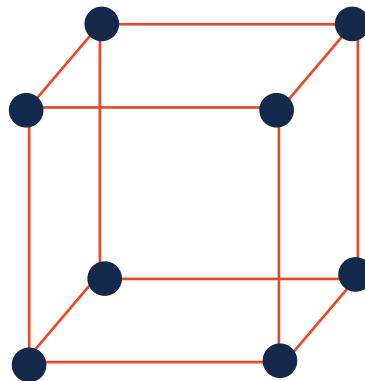
# Iterators

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



# Iterators

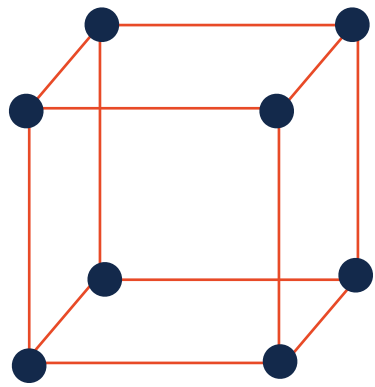
We want to be able to loop through all elements for any underlying implementation in a systematic way



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



```
1 Cube::Iterator start = myCube.begin();  
2  
3 while (it != myCube.end()) {  
4     std::cout << *it << " ";  
5     it++;  
6 }  
7
```

# 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:

```
1 template <class T>
2 class List {
3
4     class ListIterator : public
5     std::iterator<std::bidirectional_iterator_tag, T> {
6         public:
7
8             ListIterator& operator++();
9
10            ListIterator& operator--();
11
12            bool operator!=(const ListIterator& rhs);
13
14            const T& operator*();
15        };
16
17        ListIterator begin() const;
18
19        ListIterator end() const;
20    };
```

```
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:**

- 
- 

