Learning Objectives

Introduce the stack and the queue data structure
<table>
<thead>
<tr>
<th></th>
<th>Singly Linked List</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert/Remove at <strong>front</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <strong>given</strong> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <strong>given</strong> element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insert at <strong>arbitrary</strong> location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remove at <strong>arbitrary</strong> location</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:
Thinking critically about lists: tradeoffs

I want a list that can add and remove in $O(1)$. I am willing to make random access impossible to do so.
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

The **call stack** is a key concept for understanding recursion.

```
template <typename T>
typename List<T>::ListNode * & List<T>::_index(unsigned index, ListNode * & root){
    if (index == 0){ return root; }
    if (root == nullptr){ return root; }
    return _index(index - 1, root -> next);
}
```
Stack Data Structure

C++ has a built-in stack

Underlying implementation is vector or deque

```cpp
#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();
    stack.push(2);
    stack.push(1);
    stack.push(3);
    stack.push(5);
    stack.pop();
    stack.push(9);
}
```
Stack ADT

[Order]:

[Implementation]:

[Runtime]:
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?

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

An array list implementation is better in practice — why?
## Engineering vs Theory Efficiency

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

(Care of [https://gist.github.com/hellerbarde/2843375](https://gist.github.com/hellerbarde/2843375))
Queue Data Structure

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

template <typename T>
class Queue {
  public:
    void enqueue(T e);
    T dequeue();
    bool isEmpty();
  
  private:
    T *items_;
    unsigned capacity_;
    unsigned size_;    
    unsigned 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);
#include <queue>

namespace MyNamespace {

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

    private:
      T *items_; // Front
      unsigned capacity_; // Size
      unsigned size_; // Size
      unsigned front_; // Front
  };

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

}

```cpp
#include <queue>

namespace MyNamespace {

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

    private:
      T *items_; // Front
      unsigned capacity_; // Size
      unsigned size_; // Size
      unsigned front_; // Front
  };

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

} // namespace MyNamespace
```
Queue Data Structure: Resizing

```
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);
```
Queue ADT

• [Order]:

• [Implementation]:

• [Runtime]: