CS102
C++ Stacks & Queues

Prof Tejada

Stacks/Queues

- Templated lists are good for storing generic sequences of items, but they can be specialized to form other useful structures
- What if we had a List, but we restricted how insertion and removal were done?
  - *Stack:* only ever insert/remove from the front of the list
  - *Queue:* only ever insert at the back and remove from the front of the list

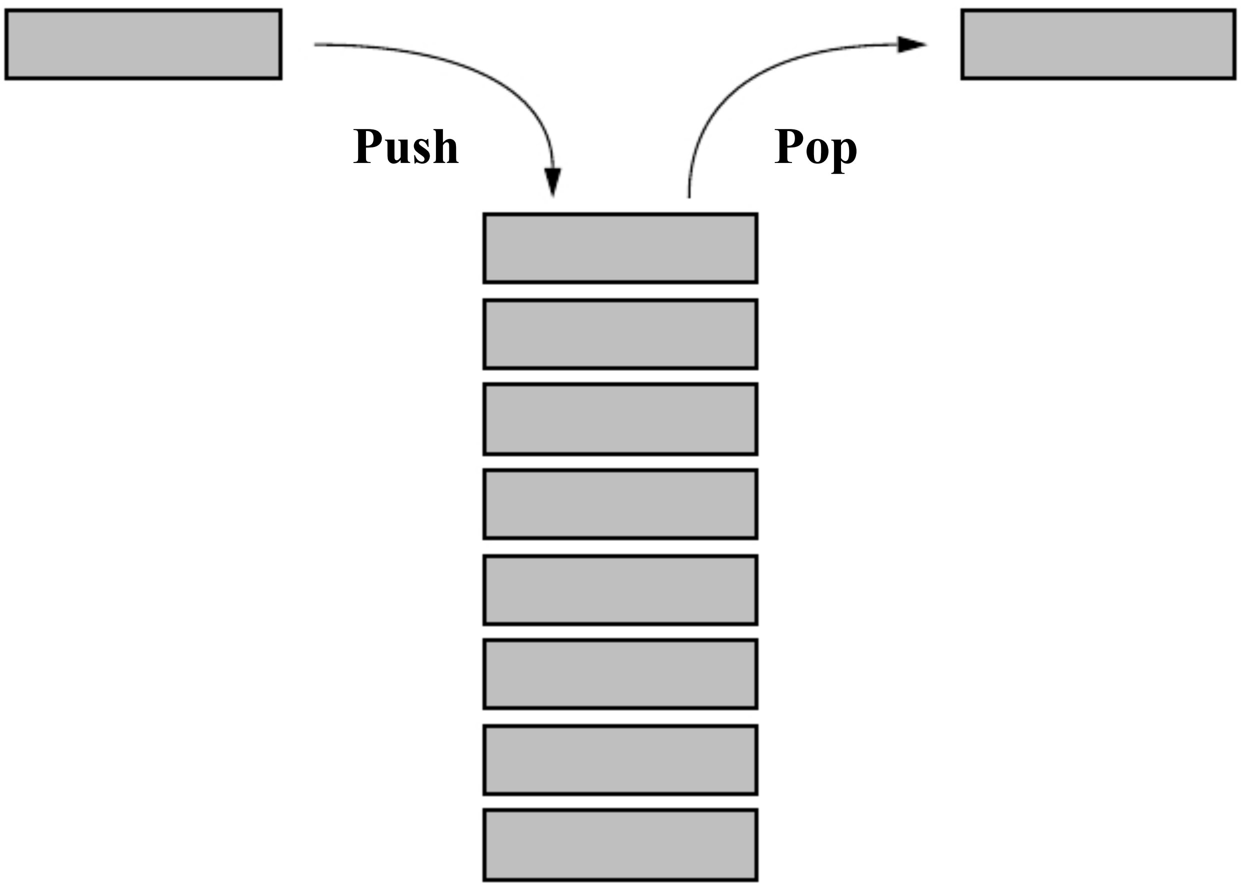
Stacks

- ***Stack:*** **a list of items where insertion and removal only occurs at one end of the list**
  - **Examples**
    - **A spring-loaded plate dispenser at a buffet**
    - **A stack of boxes where you have to move the top one to get to ones farther down**
    - **A PEZ dispenser**
- **Stacks are *LIFO (Last In, First Out)***
  - **Items at the top of the stack are the newest**
  - **Items at the bottom of the stack are the oldest**

The Stack ADT

- What member data does a Stack need?
  - A list of items
  - A length
  - A maximum size (optional)
- What member functions does a Stack have?
  - *push(item)* - Add an item to the top of the Stack
  - *pop()* - Remove the top item from the Stack
  - *top()* - Get a reference to the top item on the Stack (don’t remove it though!)
  - *size()* - Get the number of items in the Stack
  - *empty()* - Check if the Stack is empty

The Stack ADT



Stack Declaration

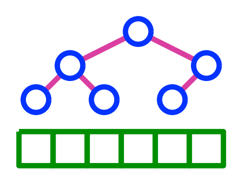
- What does the interface for a Stack look like?

```
template <typename T>
class Stack
{
  public:
    Stack();
    ~Stack();
    int  size() const;
    void  push(const T& value);
    void  pop();
    T& top();
    bool  empty() const;
};
```

Stack Declaration

- How would you build a Linked List-based Stack?

```
template <typename T>
class Stack
{
  private:
    Node<T>*  head;
    int  length;
  public:
    Stack();
    ~Stack();
    int  size()  const;
    void  push(const  T&  value);
    void  pop();
    T&  top();
    bool  empty()  const;
};
```



Stack Declaration

- **How would you build a Linked List-based Stack?**
  - **You could also back the Stack with a vector**

```
template  <typename  T>
class  Stack
{
    private:
        T*   date;  //could  also   be   vector<T>
        int   length;
    public:
        Stack();
        ~Stack();
        int   size()  const;
        void   push(const  T&  value);
        void   pop();
        T&  top();
        bool   empty()  const;
};
```

Stack Examples

- Reversing the letters in a string

```
int main() {
    Stack<char> s;

    string word;
    cout << "Enter a word: ";
    getline(cin,word);

    for(int i=0; i < word.size(); i++) {
        s.push(word.at(i));
    }
    while(!s.empty()) {
        cout << s.top();
        s.pop();
    }
}
```

Stack Examples

- How would you check that this string has equal numbers of opening/closing parentheses?

  ((((3*4 + 1)   5) + 6 * (2-3) + 4   (1/5)) + 1) + 2

- If you see a "(", use a *push*
- If you see a ")", use a *pop*
- At the end of the string, your stack should be completely empty!

Stack Examples

- The *call stack* isused to keep track of function calls in C++
- What happens when your code calls a function?
- What happens when you return from a function?
- How does your code keep track of which line it should return to when a function ends?

```
void  A(int  x,int  y)
{
    int  m=0;
    B(x);
}
```

```
void  B(int   x)
{
    int  n=0;
    C();
}
```

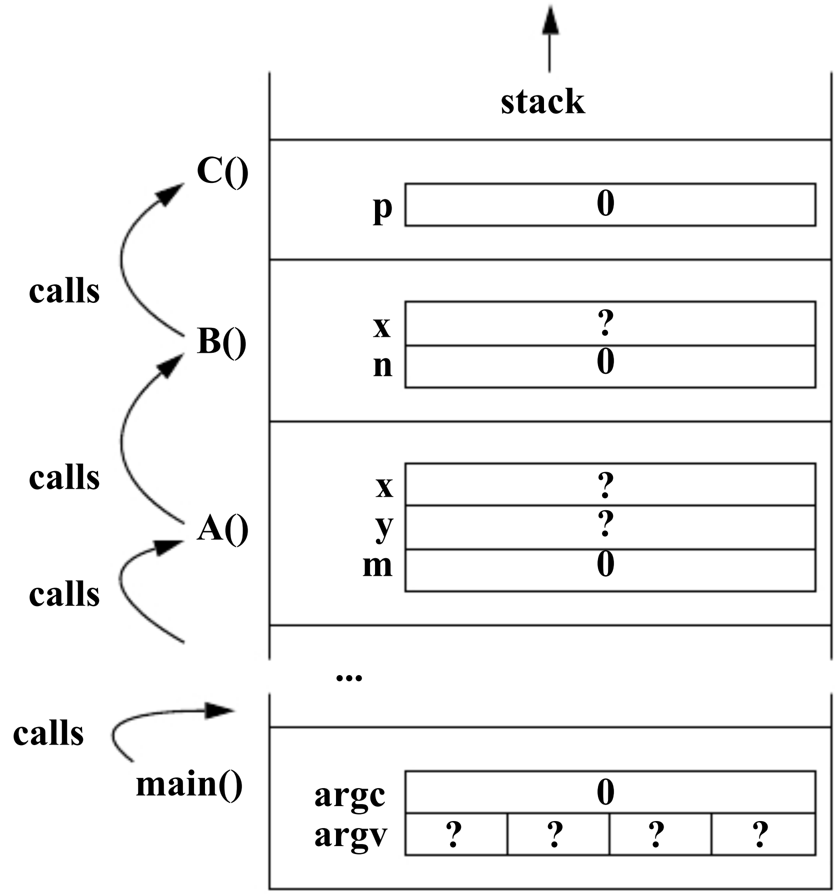
```
void  C()
{
    int   p=0;
    cout   <<   p;
}
```

The Call Stack

- What happens when a function is called?
  - The address of the next line of code is pushed onto the stack (one line past the function call)
  - A placeholder is put on the stack for the function's return type
  - Execution jumps to the function's code
  - All arguments to the function go on the stack
  - The function begins executing
  - All local variables to the function are pushed onto the call stack

The Call Stack

```
void C()
{
    int p=0;
    cout << p;
}

void B(int x)
{
    int n=0;
    C();
}

void A(int x,int y)
{
    int m=0;
    B(x);
}
```



The Call Stack

- What happens when a function returns?
  - The return value is copied back into the placeholder that we made for it
  - All local arguments and variables are popped off of the stack
    - This is why we call them *stack variables*
  - The return value is popped off the stack and assigned to a variable (if need be)
  - The address of the next line of code is popped off the stack and executed

- Inception is a perfect example of how the call stack works
  - Dreams = Functions
  - Dreaming = Calling a function
  - Waking up = Returning from a function

Other Stack Details

- How should you implement a Stack?
  - Back it with an array
  - Back it with a vector
  - Back it with a linked list
  - Inherit from linked list
  - Which is best?

- Stack Error Conditions
  - *Stack Underflow:* the name for the condition where you call *pop* on an *empty* Stack
  - *Stack Overflow:* the name for the condition where you call *push* on a *full* Stack (a stack that can't grow any more)

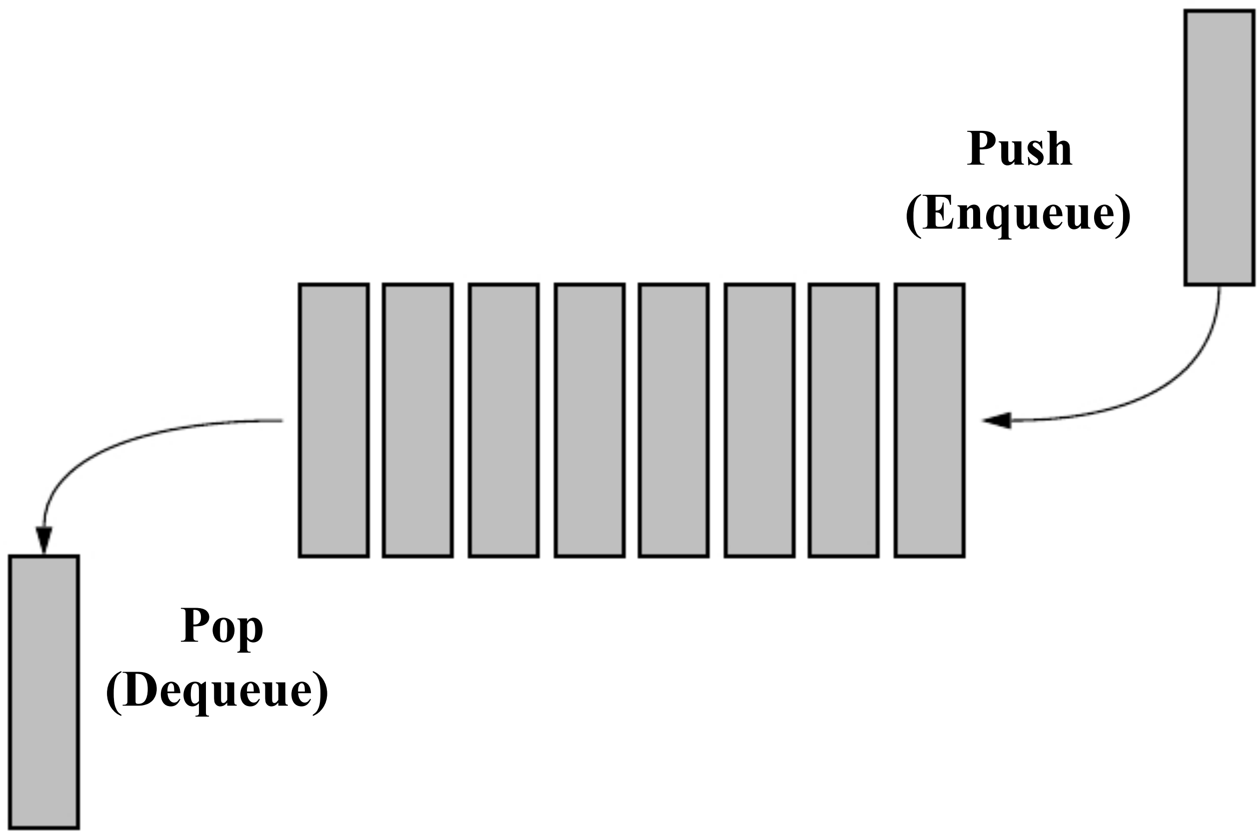
Queues

- ***Queue:*** **a list of items where insertion only occurs at the back of the list and removal only occurs at the front of the list**
  - **Like waiting in line for a cashier at a store**
- **Queues are *FIFO (First In, First Out)***
  - **Items at the back of the queue are the newest**
  - **Items at the front of the queue are the oldest**
  - **Elements are processed in the order they arrive**

The Queue ADT

- What member data does a Queue have?
  - A list of items
  - A length
  - A maximum size (optional)
- What member functions does a Queue have?
  - *push(item):* add an item to the back of the Queue
  - *pop():* remove the front item from the Queue
  - *front()/back():* get a reference to the front or back item of the Queue (don’t remove it though!)
  - *size():* get the number of items in the Queue
  - *empty():* check if the Queue is empty

The Queue ADT



Queue Declaration

- What does the interface for a Queue look like?

```
template <typename T>
class Queue
{
  public:
    Queue();
    ~Queue();
    int size() const;
    void push(const T& value); //enqueue
    void pop(); //dequeue
    T& front();
    T& back();
    bool empty() const;
};
```

Queue Declaration

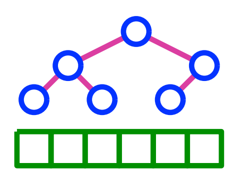
- What does a Linked List-based Queue look like?

```
template <typename T>
class Queue
{
  private:
    Node<T> *head, *tail;
    int length;
  public:
    Queue();
    ~Queue();
    int size() const;
    void push(const T& value); //enqueue
    void pop(); //dequeue
    T& front();
    T& back();
    bool empty() const;

};
```

Queue Declaration

- What does an array-based Queue look like?
  - You could also back the Queue with a vector

```
template <typename T>
class Queue
{
  private:
    T*  data; //could also be vector<T>
    int length;
  public:
    Queue();
    ~Queue();
    int  size() const;
    void push(const T& value);  //enqueue
    void pop(); //dequeue
    T&  front();
    T&  back();
    bool empty() const;

};
```



Queue Examples

- **How does a printer work?**
  - **Multiple print jobs are sent in**
  - **Click "Print" on the computer is much faster than actually printing (build a backlog)**
  - **Each job is processed in the order it's received *(FIFO)***
- **Why wouldn't you use a "Print Stack" instead of a "Print Queue"?**

Other Queue Examples

- Computer processor serving threads
- Serving customers at a restaurant (in the order they were seated)
- Valets parking cars at a busy restaurant (park them in the order they arrived)
- Anything that involves elements "waiting in line"
- How do you organize your closet?

Other Queue Details

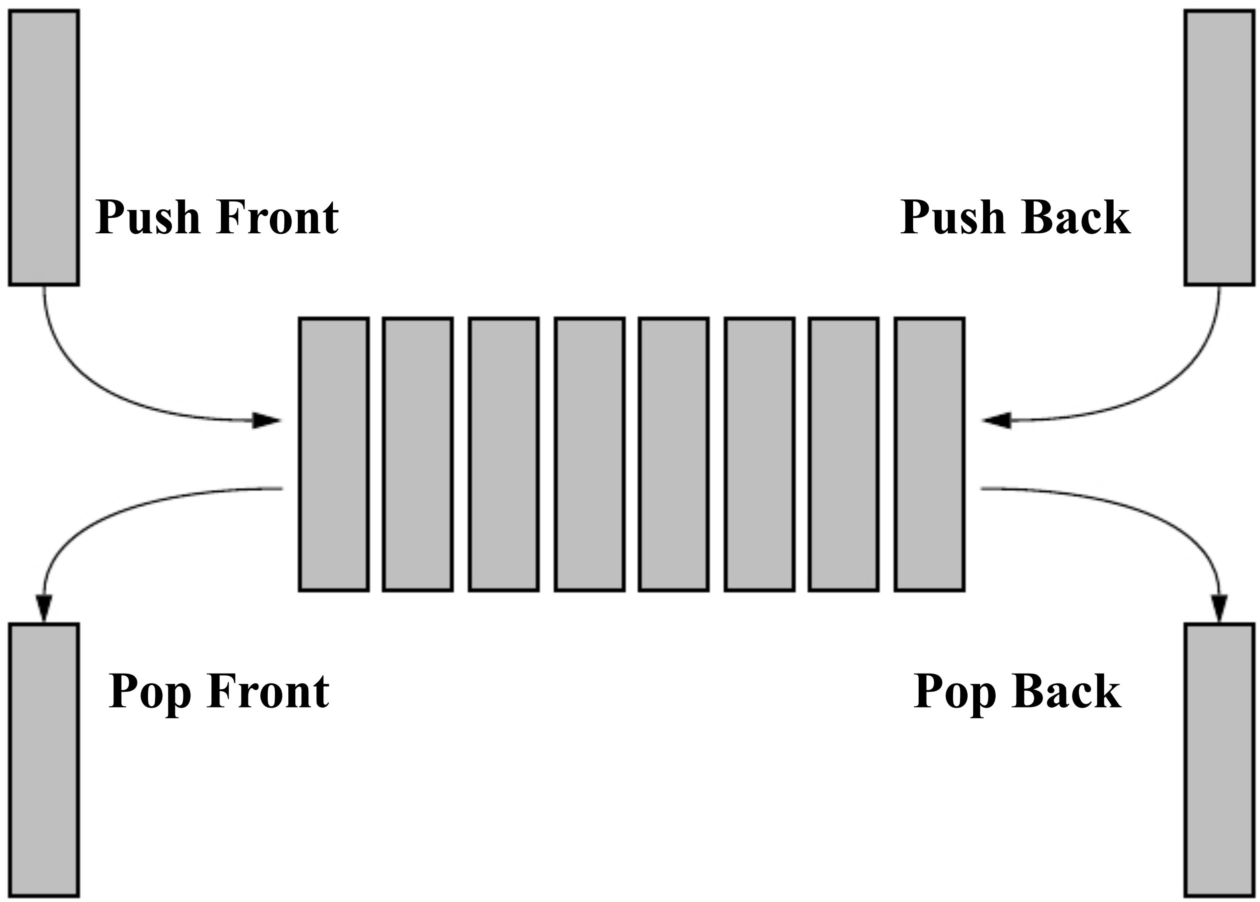
- **How should you implement a Queue?**
  - **Back it with an array**
  - **Back it with a vector**
  - **Back it with a linked list**
  - **Inherit from a linked list**
  - **Which is best?**

- **Queue Error Conditions**
  - ***Queue Underflow:*** **the name for the condition where you call *pop* on an *empty* Queue**
  - ***Queue Overflow:*** **the name for the condition where you call *push* on a *full* Queue (a Queue that can't grow any more)**

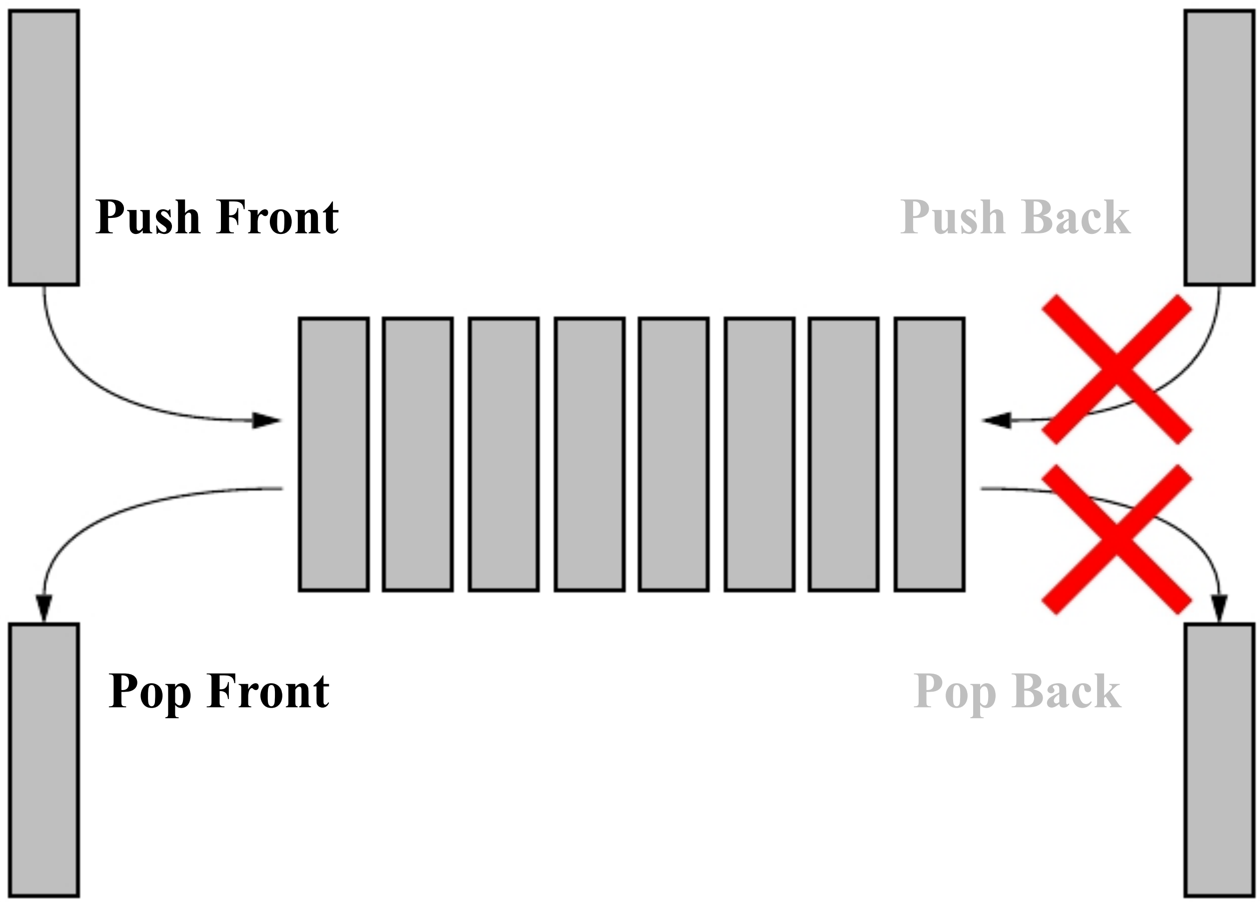
Deques

- ***Deque:*** **a combination of a Stack and a Queue where you can insert or remove at either end of the list (but not the middle)**
  - **Like books on a bookshelf**

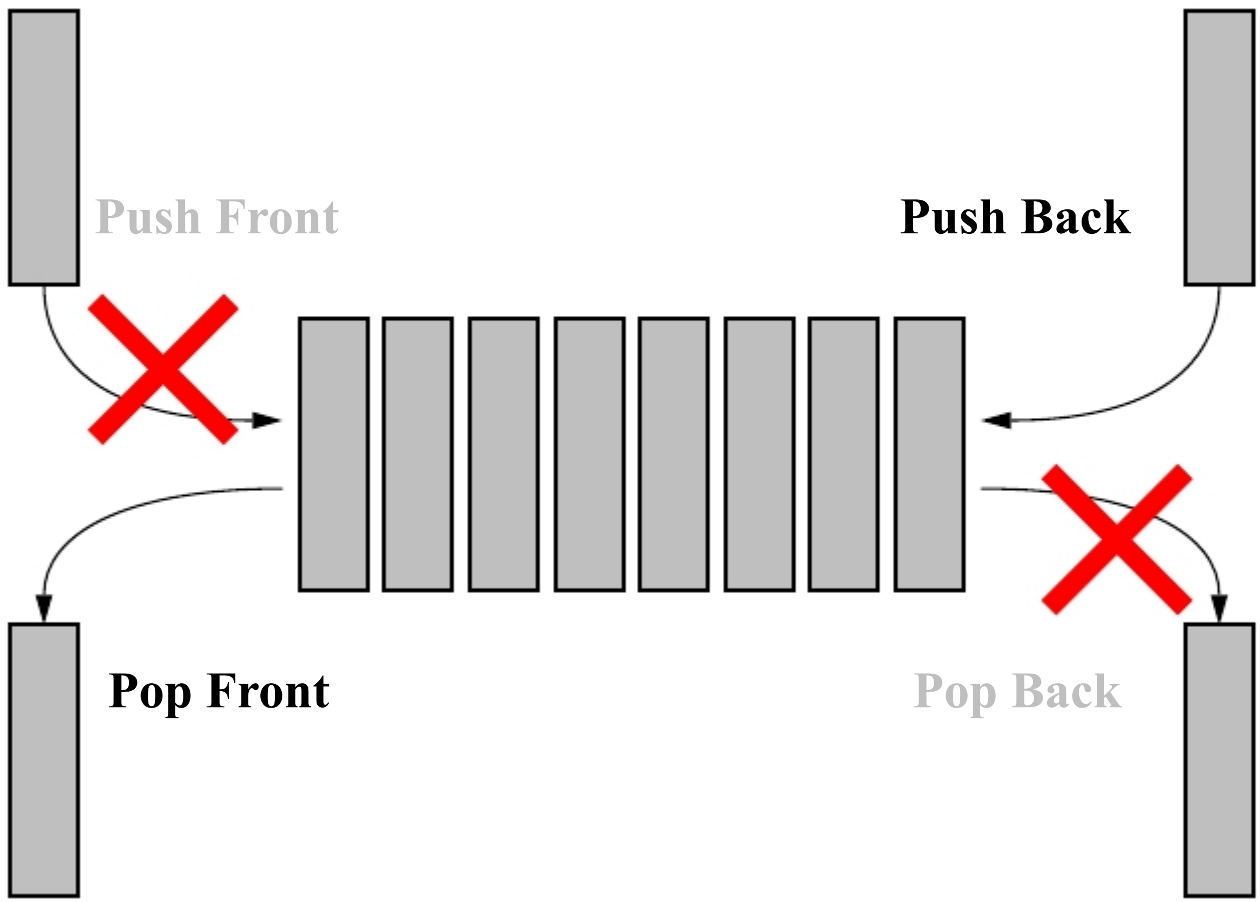
The Deque ADT



Implement Stack Using Deque



Implement Queue Using Deque



CS102
Searching

Search

- One of the best ways to understand Big O is by example
- If you were given a list of items and you wanted to see if an item is in that list, how would you do it?
  - How did you write your contains() function on your Linked List class?

Linear (a.k.a. Sequential) Search

- Start at the beginning (or end) of the list
- Compare search value with every element in the list one at a time
  - If you find what you're looking for, return true
  - If you look through all the items and don't find it, return false

Linear Search

```
int linearSearch(const vector<int>& list,
        const int& value) {
    for(int i=0;
            i < list.size();
            i++) {
        if (list[i] == value) {
            return true;
        }
    }
    return false;
}
```

- What is the Big O?

Linear Search

- What is the *best case* scenario?

- What is the *worst case* scenario?

- What is the *average* scenario?
  - Assuming value in the list?

  - Assuming all possible inputs?

Linear Search

- What is the *best case* scenario?
  - Search item is first in list = *O(1)*
- What is the *worst case* scenario?
  - Search item not in list = *O(N)*
- What is the *average* scenario?
  - Assuming value in the list?
    - Look through half the list = *O(N/2) = O(N)*
  - Assuming all possible inputs?
    - Hard to say. Probably *O(N)*

Linear Search

- What data types does this work for?
  - Arrays
  - Vectors
  - Linked Lists

Linear Search

- Search is a very recursive problem. How would we write a recursive linear search?
- What's the algorithm?
- What are the base cases?
- What's the recursive case?

Linear Search

```
int recursiveSearch(const vector<int>& list,
        const int& value) {
    recSearchHelper(list, value, 0);
}

int recSearchHelper(const vector<int>& list,
        const int& value),
        const int index) {
    if (index >= list.size()) {
        return false;
    } else if (list[i] == value) {
        return true;
    }
    return recSearchHelper(list, value, index+1);
}
```

Search

- If we know nothing else about the data in the list we're searching, is there a better way?

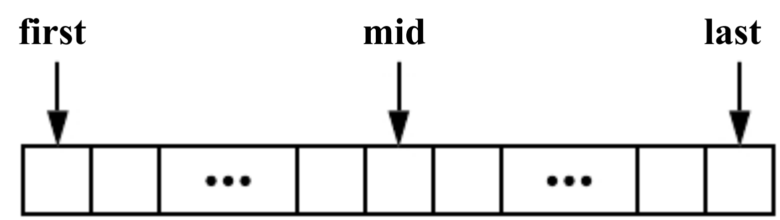
- What about if we could make assumptions about the data?
  - What assumptions would be helpful?
    - By the way, the word "assumption" means that this is *what we assume about the data*
      - If the data/input violates the assumption, it can break our algorithm
      - Sometimes, we should check if our assumption is correct before we proceed (of course, this is not free)

Binary Search

- ***Assumption:*** **List elements are *sorted* in ascending order**
- **Compare search value to the median and see if the search item is ==, < or >**
- **If search item is equal, return true**
- **Otherwise, cut the list in half and repeat by finding the median value in the valid half of the list**
- **Return false if the value is not found**
- **Ex: Looking up a word in a dictionary**

Binary Search

```
int binarySearch(const vector<int>& list,
        const int& value) {
    int first=0, last=list.size()-1;
    while (first <= last) {
        int mid=(first+last)/2;
        if (list[mid] == value) {
            return true;
        } else if (list[mid] > value) {
            last = mid-1;
        } else {
            first = mid+1;

        }
    }
    return false;
}
```



Binary Search

- What is the *best case* scenario?

- What is the *worst case* scenario?

- What is the *average* scenario?
  - Assuming value in the list?
  - Assuming all possible inputs?

Binary Search

- What is the *best case* scenario?
  - Value is in the middle of the list = $O(1)$
- What is the *worst case* scenario?
  - Value is not in the list = $O(\log(N))$
    - if $a = \log_2 n$, then $2^a = n$
    - therefore, $x = \log_2 2^x$
    - as n grows geometrically/exponentially, $\log_2 n$ grows linearly
    - the base of logarithm is usually omitted
      - $\log_b n = \log_x n \,/\, \log_x b$ for any base x
      - $\log_b n = \log_2 n \,/\, \log_2 b = c \cdot \log_2 n$

log n	*n*
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096

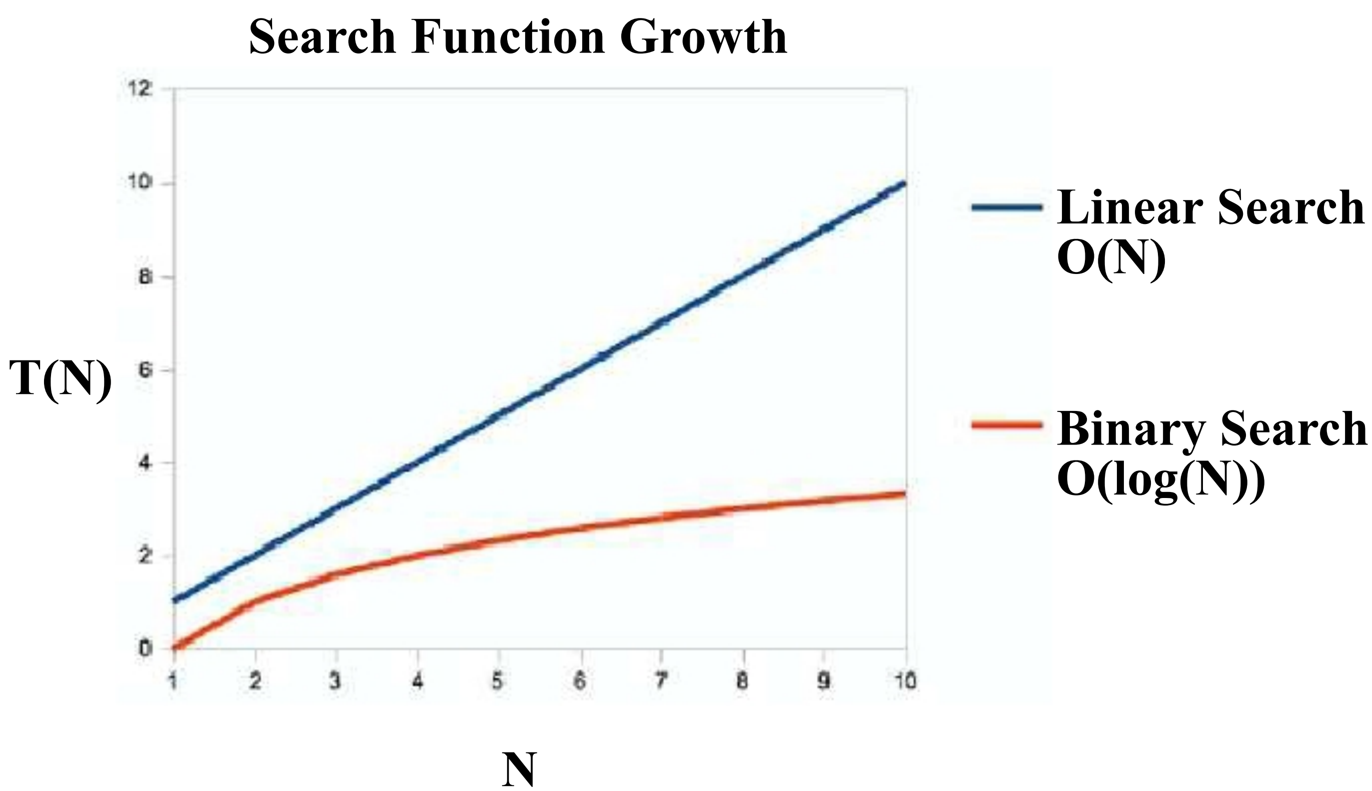
Binary Search

- What is the *best case* scenario?
  - Value is in the middle of the list = *O(1)*
- What is the *worst case* scenario?
  - Value is not in the list = *O(log(N))*
- What is the *average* scenario?
  - Assuming value in the list?
    - *O(2*log(N)-3) = O(log(N))*
  - Assuming all possible inputs?
    - *O(2*log(N)) = O(log(N))*

Binary Search

- **What data types does this work for?**
  - **Arrays**
  - **Vectors**
  - **Linked Lists (will *not* work)**

Binary Search vs. Linear Search



Stuff to Read

- Stack Overflow Question: "Plain English explanation of Big O"
  - http://stackoverflow.com/questions/487258/plain-english-explanation-of-big-o
    - Check out the accepted answer
- Slightly longer version:
  - http://www.cforcoding.com/2009/07/plain-english-explanation-of-big-o.html