Previously, on CS106B...
A Bounded Stack

- Element array
- Allocated size: 4
- Logical size: 0
A Bounded Stack

The stack’s allocated size is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.
A Bounded Stack

The stack’s **allocated size** is the number of slots in the array. Remember – arrays in C++ cannot grow or shrink.

The stack’s **logical size** is the number of elements actually stored in the stack. This lets us track how much space we’re actually using.
A Bounded Stack

The figure illustrates a bounded stack with an allocated size of 137. The stack contains an element array, an allocated size of 4, and a logical size of 1.
A Bounded Stack

Element array

Allocated size

Logical size

137 42

4

2
A Bounded Stack

137  42  2718

element array
allocated size
logical size

4

3
A Bounded Stack

element array
allocated size
logical size

137 42 2718 512
A Bounded Stack

**element array**

allocated size

logical size

137  42  2718  512

4

3
A Bounded Stack

Arrays cannot grow or shrink, so this older value is still technically there in the array. We’re just going to pretend it isn’t.
A Bounded Stack

Diagram showing:
- Element array
- Allocated size:
  - 4
- Logical size:
  - 2

Numbers:
- 137
- 42
- 2718
- 512
A Bounded Stack

- Element array: 137, 42, 161, 512
- Allocated size: 4
- Logical size: 3
A Bounded Stack

<table>
<thead>
<tr>
<th>element array</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated size</td>
</tr>
<tr>
<td>logical size</td>
</tr>
</tbody>
</table>

137  42  161  314
Running out of Space

- Our current implementation very quickly runs out of space to store elements.
- What should we do when this happens?
An Initial Idea

```
137  42
element array
allocated size
logical size
```

```
4
2
```
An Initial Idea

137  42  161

element array
allocated size
logical size

4

3
An Initial Idea

- Element array:
  - Allocated size: 4
  - Logical size: 4

- Array values:
  - 137
  - 42
  - 161
  - 314
An Initial Idea

Element array

Allocated size

Logical size

137  42  161  314
An Initial Idea

element array

allocated size

logical size

137

42

161

314

137
An Initial Idea

137  42

137  42  161  314

element array
allocated size  4
logical size  4
An Initial Idea

- Element array: 137, 42, 161
- Allocated size: 4
- Logical size: 4

137 42 161 314
An Initial Idea

element array
allocated size
logical size

137 42 161 314

137 42 161 314
An Initial Idea

- Element array: 137, 42, 161, 314
- Allocated size: 4
- Logical size: 4
An Initial Idea

137  42  161  314

- element array
- allocated size: 4
- logical size: 4
An Initial Idea

137  42  161  314

allocated size

element array

allocated size  4

logical size  4
An Initial Idea

137 42 161 314

element array
allocated size 5
logical size 4
An Initial Idea

element array
allocated size
logical size

137  42  161  314  159

5

5
An Initial Idea

Element array:
- Logical size: 5
- Allocated size: 5

Array:
- 137
- 42
- 161
- 314
- 159
An Initial Idea

137

137 42 161 314 159

- element array
- allocated size 5
- logical size 5
An Initial Idea

- Element array
- Allocated size: 5
- Logical size: 5

Arrays: 137, 42, 161, 314, 159
An Initial Idea

```
137  42  161
```

```
137  42  161  314  159
```

```
allocated size
logical size
```

```
element array
```

```
5
```

```
5
```
An Initial Idea

- Element array: [137, 42, 161, 314]
- Allocated size: 5
- Logical size: 5
An Initial Idea

allocated size

logical size

element array

allocated size

logical size

137 42 161 314 159
An Initial Idea

<table>
<thead>
<tr>
<th>137</th>
<th>42</th>
<th>161</th>
<th>314</th>
<th>159</th>
</tr>
</thead>
</table>

allocated size

logical size

element array

allocated size

logical size

5

5
An Initial Idea

### Element Array

- **Allocated Size**: 5
- **Logical Size**: 5

### Values

- 137
- 42
- 161
- 314
- 159
An Initial Idea

```
137  42  161  314  159
```

- **element array**
- **allocated size**: 5
- **logical size**: 5
An Initial Idea

- Element array
- Allocated size: 6
- Logical size: 5
An Initial Idea

- Element array:
  - Allocated size: 6
  - Logical size: 6

- Array:
  - 137
  - 42
  - 161
  - 314
  - 159
  - 265
Ready... set... grow!
An Initial Idea

137 42 161 314

elems 4
allocated size 4
logical size 4
An Initial Idea

OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}

void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
An Initial Idea

void OurStack::grow() {
  allocatedSize++;
  int* newElems = new int[allocatedSize];
  for (int i = 0; i < size(); i++) {
    newElems[i] = elems[i];
  }
  delete[] elems;
  elems = newElems;
}

void OurStack::grow() {
  allocatedSize++;
  int* newElems = new int[allocatedSize];
  for (int i = 0; i < size(); i++) {
    newElems[i] = elems[i];
  }
  delete[] elems;
  elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
}

newElems

allocated size

logical size

elems

137 42 161 314
An Initial Idea

void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}

137  42  161  314  
newElems

elems
allocated size
logical size

137  42  161  314
An Initial Idea

```cpp
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
```
An Initial Idea

void OurStack::grow() {
    allocatedSize++;
    int* newElems = new int[allocatedSize];
    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }
    delete[] elems;
    elems = newElems;
}
An Initial Idea

void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
An Initial Idea

```cpp
void OurStack::grow() {
    allocatedSize++;

    int* newElems = new int[allocatedSize];

    for (int i = 0; i < size(); i++) {
        newElems[i] = elems[i];
    }

    delete[] elems;
    elems = newElems;
}
```
Analyzing Our Approach

- We now have a working solution, but is it an efficient solution?
- Let's analyze the big-O complexity of the five operations.
  - size:
  - isEmpty:
  - push:
  - pop:
  - peek:
Analyzing Our Approach

- We now have a working solution, but is it an efficient solution?
- Let's analyze the big-O complexity of the five operations.
  
  - **size**: $O(1)$
  - **isEmpty**: $O(1)$
  - **push**: $O(n)$
  - **pop**: $O(1)$
  - **peek**: $O(1)$
What This Means

• What is the complexity of pushing $n$ elements and then popping them?
What This Means

- What is the complexity of pushing $n$ elements and then popping them?
- Cost of the pushes:
  - $1 + 2 + 3 + 4 + \ldots + n$
What This Means

• What is the complexity of pushing $n$ elements and then popping them?

• Cost of the pushes:
  • $1 + 2 + 3 + 4 + \ldots + n = O(n^2)$
What This Means

- What is the complexity of pushing $n$ elements and then popping them?
- Cost of the pushes:
  - $1 + 2 + 3 + 4 + ... + n = O(n^2)$
- Cost of the pops:
  - $1 + 1 + 1 + 1 + ... + 1$
What This Means

- What is the complexity of pushing \( n \) elements and then popping them?

- Cost of the pushes:
  - \( 1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2) \)

- Cost of the pops:
  - \( 1 + 1 + 1 + 1 + \ldots + 1 = \mathcal{O}(n) \)
What This Means

• What is the complexity of pushing \( n \) elements and then popping them?

• Cost of the pushes:
  • \( 1 + 2 + 3 + 4 + \ldots + n = O(n^2) \)

• Cost of the pops:
  • \( 1 + 1 + 1 + 1 + \ldots + 1 = O(n) \)

• Total cost:
What This Means

• What is the complexity of pushing \( n \) elements and then popping them?

• Cost of the pushes:
  • \( 1 + 2 + 3 + 4 + \ldots + n = O(n^2) \)

• Cost of the pops:
  • \( 1 + 1 + 1 + 1 + \ldots + 1 = O(n) \)

• Total cost: \( O(n^2) \)
Validating Our Model
Time-Out for Announcements!
Assignment 4

- Assignment 4 is due on Friday.
- Recommendation: Aim to complete all the parts of the assignment by the end of this evening.
- We’ve posted a handy Assignment Submission Checklist up on the course website. Work through this before submitting – it’ll help make sure your code is ready to go!
Midterm Exam

- The midterm exam is next Tuesday, February 21 from 7:00PM – 10:00PM.
  - Location TBA
- Covers topics up through and including big-O notation, plus Assignments 0 – 4.
- Closed-book, closed-computer, limited-note. You get one double-sided sheet of 8.5” × 11” notes when you take the exam. We also provide a library reference sheet.
- Practice exam posted on the course website.
- Need some practice? Work through the section handouts, the chapter exercises in the textbook, and revisit old assignments. Need more practice? Let us know!
Want to check out Treehacks? A little nervous about it? Don't know anyone else who's doing it?

**Come to HACK 101!**

Learn how to be successful at a hackathon! Meet teammates for Treehacks! Start the ideation process for your project!

RSVP [HERE](#) hosted by Black in CS
Back to the Stack!
Speeding up the Stack
Key Idea: *Plan for the Future*
A Better Idea
A Better Idea

element array
allocated size
logical size

137 42 161 314
A Better Idea

- Element array
- Allocated size: 4
- Logical size: 4

137 42 161 314
A Better Idea

allocated size

logical size

element array

allocated size

logical size

137 42

137 42 161 314

4

4

4
A Better Idea

Element array

Allocated size

4

Logical size

4
A Better Idea

allocated size

logical size

element array

allocated size

logical size

4

137  42  161  314

137  42  161  314
A Better Idea

- Element array
  - Allocated size: 4
  - Logical size: 4
A Better Idea

137  42  161  314

allocated size

logical size

element array

allocated size

logical size

4
A Better Idea

Element array

Allocated size: 4

Logical size: 4
A Better Idea

137  42  161  314

element array
allocated size
logical size

6
4
A Better Idea

137  42  161  314  159

element array
allocated size
logical size
A Better Idea
What Just Happened?

- Half of our pushes are now “easy” pushes, and half of our pushes are now “hard” pushes.
- Hard pushes still take time $O(n)$.
- Easy pushes only take time $O(1)$.
- Worst-case is still $O(n)$.
- What about the average case?
Analyzing the Work
Analyzing the Work
Analyzing the Work

We cut down the amount of work by roughly one half!
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis
A Different Analysis

We cut down the amount of work by roughly one half!
How does it stack up?
A Much Better Idea

137  42

element array
allocated size  2
logical size  2
A Much Better Idea

- Element array
- Allocated size: 2
- Logical size: 2

137 42
A Much Better Idea

- Element array
- Allocated size
- Logical size

```
Array:
137
42
2
2
```
A Much Better Idea

element array
allocated size
logical size

137 42

allocated size
logical size

2

2
A Much Better Idea

element array

allocated size 2

logical size 2

137 42

137 x 42

137 42
A Much Better Idea

allocated size

logical size

element array

2

2

2
A Much Better Idea

137  42

element array
allocated size
logical size

2

2
A Much Better Idea

element array
allocated size
logical size

137
42

4
2
A Much Better Idea

element array
allocated size
logical size

137 42 271

4
3
A Much Better Idea

137  42  271  828

- element array
- allocated size: 4
- logical size: 4
A Much Better Idea

137  42  271  828

element
array

allocated
size

logical
size

4

4
A Much Better Idea

element array
allocated size
logical size

137
42
271
828
A Much Better Idea

137  42

137  42  271  828

element array
allocated size
4
logical size
4
A Much Better Idea

allocated size

logical size

element array

allocated size

logical size

137 | 42 | 271

828

4

4

4
A Much Better Idea

allocated size

logical size

element array

allocated size

logical size

137  42  271  828

137  42  271  828
A Much Better Idea

```
137  42  271  828
```

```
137  42  271  828
```

```
<table>
<thead>
<tr>
<th>element array</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated size</td>
</tr>
<tr>
<td>logical size</td>
</tr>
</tbody>
</table>
```
A Much Better Idea

137  42  271  828

element array
allocated size  4
logical size  4
A Much Better Idea
A Much Better Idea

Element array

Allocated size

Logical size

137 | 42 | 271 | 828
A Much Better Idea

Element array

Allocated size

Logical size

137  42  271  828  182
A Much Better Idea

<table>
<thead>
<tr>
<th>element array</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>allocated size</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical size</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 137 | 42 | 271 | 828 | 182 | 845 |
A Much Better Idea

Element array

Allocated size: 8

Logical size: 7

137  42  271  828  182  845  904
A Much Better Idea

```
137  42  271  828  182  845  904  5
```

- **element array**
- **allocated size**: 8
- **logical size**: 8
Let's Give it a Try!
How do we analyze this?
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work
Spreading the Work

On average, we do just 3 units of work!

This is $O(1)$ work on average!
Sharing the Burden

• We still have “heavy” pushes taking time $O(n)$ and “light” pushes taking time $O(1)$.
• Worst-case time for a push is $O(n)$.
• Heavy pushes become so rare that the average time for a push is $O(1)$.
• Can we confirm this?
Amortized Analysis

• The analysis we have just done is called an *amortized analysis*.

• Reason about the total amount of work done, not the word done per operation.

• In an amortized sense, our implementation of the stack is extremely fast!

• This is one of the most common approaches to implementing Stack.
Implementing Queue
Implementing Queue

- We've just used dynamic arrays to implement a stack. Could we use them to implement a queue?
- Yes, but here's a better idea: *could we use our stack to implement a queue?*
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Out

In

2

1
The Two-Stack Queue

Out

In

3
2
1
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

**Out**

**In**

4

1

2

3
The Two-Stack Queue
The Two-Stack Queue

Out

In

4

3
2
1
The Two-Stack Queue

In

Out

3

4

In

2

1
The Two-Stack Queue

```
3
4
```

```
2
1
```

Out

In
The Two-Stack Queue
The Two-Stack Queue

Out

2
3
4

In

1
The Two-Stack Queue
The Two-Stack Queue

Out

In

2
3
4

1
The Two-Stack Queue
The Two-Stack Queue

![Diagram showing a two-stack queue with items 1, 2, 3, and 4 in the Out stack and an empty In stack.]

- Out: 1, 2, 3, 4
- In: Empty
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Out

2

3

4

In

1
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Out

In

1 2 3

4

6 5
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

Out

1  2  3  4

In

5  6  7
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

1  2  3  4

Out

6

7

In

5
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue
The Two-Stack Queue

![Diagram of two stacks with numbers 6 and 7 in Out and 1, 2, 3, 4, 5 in In]
The Two-Stack Queue

- Maintain two stacks, an **In** stack and an **Out** stack.
- To enqueue an element, push it onto the **In** stack.
- To dequeue an element:
  - If the **Out** stack is empty, pop everything off the **In** stack and push it onto the **Out** stack.
  - Pop the **Out** stack and return its value.
Analyzing Efficiency

- How efficient is our two-stack queue?
- All enqueues just do one push.
- A dequeue might do a lot of pushes and a lot of pops.
- However, let's do an amortized analysis:
  - Each element is pushed at most twice and popped at most twice.
  - $n$ enqueues and $n$ dequeues thus do at most $4n$ pushes and pops.
  - Any $4n$ pushes / pops takes $O(n)$ amortized time.
  - Amortized cost: $O(1)$ per operation.
Next Time

- **Linked Lists**
  - A different way to represent sequences of elements.

- **Dynamic Allocation Revisited**
  - What else can we allocate?