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

Element array

Allocated size: 4

Logical size: 1

Total size: 137
A Bounded Stack

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

The diagram shows a bounded stack with allocated sizes and logical sizes.
A Bounded Stack

- Element array: [137, 42, 2718]
- Allocated size: 4
- Logical size: 3
A Bounded Stack

137  42  2718  512

element array
allocated size
logical size
A Bounded Stack

- Element array: 137, 42, 2718, 512
- Allocated size: 4
- Logical size: 3
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

137 42 2718 512

- element array
- allocated size: 4
- logical size: 2
A Bounded Stack
A Bounded Stack

137  42  161  314

element array

allocated size

logical size

4

4

4
New Stuff!
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

- element array
- allocated size: 4
- logical size: 2

[Diagram showing an array with elements 137 and 42]
An Initial Idea

- Element array: 137, 42, 161
- Allocated size: 4
- Logical size: 3
An Initial Idea

allocated size

element array

allocated size

logical size
An Initial Idea

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

Values:
- 137
- 42
- 161
- 314
An Initial Idea

- **element array**
- **allocated size**
- **logical size**

Allocate size

137 42 161 314
An Initial Idea

137  42  

137  42  161  314  

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

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

137  42  161  [314]
An Initial Idea

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

4 137
4 42
161 161
314 314
An Initial Idea

Dynamic Deallocation!

allocated size

logical size

element array

4

4

delete[]
An Initial Idea

137  42  161  314

element array
allocated size  4
logical size  4
An Initial Idea

Element array

Allocated size: 4

Logical size: 4

Values:
- 137
- 42
- 161
- 314
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
An Initial Idea

Element array

Allocated size

Logical size

137 42 161 314 159
An Initial Idea

**Element array**: 137

**Allocated size**: 5

**Logical size**: 5

**Values in the array**: 137, 42, 161, 314, 159
An Initial Idea

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

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

- Element array
- Allocated size
- Logical size

```plaintext
137  42  161
137  42  161  314  159
```
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

137  42  161  314  159
An Initial Idea

Dynamic Deallocation!

allocated size

137  42  161  314  159

logical size

5

element array

5

delete[]
An Initial Idea

| 137 | 42  | 161 | 314 | 159 |

- element array
- allocated size
- logical size

- 5
- 5
An Initial Idea

Element array

Allocated size

Logical size

5

5

5

137  42  161  314  159
An Initial Idea

- Element array: 137, 42, 161, 314, 159
- Allocated size: 6
- Logical size: 5
An Initial Idea

Element array

Allocated size

Logical size

137  42  161  314  159  265
Ready... set... grow!
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int pop();
    int peek() const;

    int size() const;
    bool isEmpty() const;

private:

    int* elems;
    int allocatedSize;
    int logicalSize;
};
class OurStack {
public:
  OurStack();
  ~OurStack();

  void push(int value);
  int pop();
  int peek() const;

  int size() const;
  bool isEmpty() const;

private:
  void grow();

  int* elems;
  int allocatedSize;
  int logicalSize;
};

This is a private member function. It's a helper function only the implementation can call.
An Initial Idea

- **Elems**: 137, 42, 161, 314
- **Allocated Size**: 4
- **Logical Size**: 4
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;
}

137  42  161  314

elems
allocated size  5
logical size    4
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];
    }
}

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;
}

Dynamic Deallocation!
An Initial Idea

allocated size

logical size
elems

137  42  161  314

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

    int* newElems = new int[allocatedSize];

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

    delete[] elems;
}

delete[] 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

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:
    - 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:
  • 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:
  • $1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2)$
What This Means

• What is the complexity of pushing $n$ elements and then popping them?
• Cost of the pushes:
  • $1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2)$
• Cost of the pops:
  • $1 + 1 + 1 + 1 + \ldots + 1$
What This Means

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

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

• Cost of the pops:
  • $1 + 1 + 1 + 1 + \ldots + 1 = O(n)$
What This Means

- What is the complexity of pushing $n$ elements and then popping them?
- Cost of the pushes:
  - $1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2)$
- Cost of the pops:
  - $1 + 1 + 1 + 1 + \ldots + 1 = \mathcal{O}(n)$
- Total cost:
What This Means

- What is the complexity of pushing $n$ elements and then popping them?
- Cost of the pushes:
  - $1 + 2 + 3 + 4 + \ldots + n = \mathcal{O}(n^2)$
- Cost of the pops:
  - $1 + 1 + 1 + 1 + \ldots + 1 = \mathcal{O}(n)$
- Total cost: $\mathcal{O}(n^2)$
Validating Our Model
Time-Out for Announcements!
Assignment 4

• Assignment 4 is due on Friday.
• You can use a late day to extend the deadline to Wednesday (there’s no class on Monday), but we don’t recommend this.
  • That will eat into your time for studying for the exam.
  • Topics from Assignment 4 are fair game for the exam.
• YEAH Hours for Assignment 5 will be on Friday at 3:30PM in 380-380Y.
Midterm Exam

• The midterm exam is next **Tuesday, February 19** from **7:00PM - 10:00PM**. Locations are divvied up by last (family) name:
  • A – K: Go to **Bishop Auditorium**
  • L – Z: Go to **Hewlett 200**
• It covers topics from Lectures 01 – 12 (up through and including big-O notation) and Assignments 0 – 4.
• The exam is closed-book and limited-note. You may bring one double-sided sheet of 8.5” × 11” of notes to the exam with you.
Midterm Exam

• We will be administering the exam using a software tool called *BlueBook*.

• Visit the CS106B website, click the “BlueBook” link under the “Resources” tab, then download the BlueBook software.

• This week’s section handout will be done through BlueBook so that you get a chance to test it out.

• Need a laptop for the exam? We can help out with that. Please contact us ASAP so we can make appropriate arrangements.
Practice Midterm Exam

• There’s a practice midterm exam up on the course website. It’s a minimally-modified version of the exam we gave out in Winter 2017.

• The password is `maplesyrup`

and you’ll see why when you start the exam. 😊
Back to the Stack!
Speeding up the Stack
Key Idea: *Plan for the Future*
A Better Idea

137  42  161  314

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

allocated size

logical size

element array

allocated size

logical size

137  42  161  314
A Better Idea

```
137
```

```
137  42  161  314
```

```
+-------------------+
| element array     |
| allocated size    |
| logical size      |
|                   |
+-------------------+

allocated size: 4
logical size: 4
A Better Idea

allocated size

logical size

element array

allocated size

logical size

137 42

137 42 161 314
A Better Idea

allocated size

logical size

element array

allocated size

logical size

137 42 161

137 42 161 314
A Better Idea

137  42  161  314

137  42  161  314

element array
allocated size
logical size

4

4

4
A Better Idea

Dynamic Deallocation!

element array
allocated size
logical size

137 42 161 314

delete[]
A Better Idea

137 42 161 314

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

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

137 42 161 314
A Better Idea

[Diagram showing an element array with allocated size 6 and logical size 4, with values 137, 42, 161, 314]
A Better Idea
A Better Idea

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

- Array:
  - 137
  - 42
  - 161
  - 314
  - 159
  - 265
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

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
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
logical size

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

allocated size
logical size
A Much Better Idea

element array

allocated size

logical size

137 42

137 42

2

2
A Much Better Idea

Dynamic Deallocation!

- element array
- allocated size
- logical size

```
137 42
```

```
delete[]
```
A Much Better Idea

```
137 42
```

- element array
- allocated size
- logical size

```
2
2
```
A Much Better Idea

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

```
137  42
```
A Much Better Idea

137 42

element array
allocated size
logical size

4 2
A Much Better Idea

- Element array: 137, 42, 271
- Allocated size: 4
- Logical size: 3
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**

<table>
<thead>
<tr>
<th>element array</th>
<th>allocated size</th>
<th>logical size</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>42</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>271</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>828</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The diagram illustrates an element array with allocated size 4, logical size 4, and values 137, 42, 271, and 828.
A Much Better Idea

<table>
<thead>
<tr>
<th>137</th>
<th>42</th>
</tr>
</thead>
</table>

allocated size

| 137 | 42 | 271 | 828 |

logical size

element array

allocated size

logical size

4
A Much Better Idea

allocated size

logical size

element array

allocated size

logical size

4

137  42  271

828
A Much Better Idea

element array
allocated size
logical size

137 | 42 | 271 | 828

137 | 42 | 271 | 828
A Much Better Idea

137  42  271  828

Dynamic Deallocation!

element array
allocated size
logical size

4

4
A Much Better Idea

| 137 | 42  | 271 | 828 |

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

element array
allocated size
logical size
A Much Better Idea

137  42  271  828

element array
allocated size
logical size
A Much Better Idea

```
137  42  271  828  182
```

- **element array**
- **allocated size**: 8
- **logical size**: 5
A Much Better Idea

<table>
<thead>
<tr>
<th></th>
<th>137</th>
<th>42</th>
<th>271</th>
<th>828</th>
<th>182</th>
<th>845</th>
</tr>
</thead>
</table>

allocated size

logical size

element array

allocated size

logical size

8

6
A Much Better Idea

element array
allocated size
logical size

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>42</td>
<td>271</td>
<td>828</td>
<td>182</td>
<td>845</td>
<td>904</td>
<td></td>
</tr>
</tbody>
</table>
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*.

• We reason about the total 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.
Your Action Items

- **Download BlueBook**
  - Hopefully you’ve already done this; if not, please do that soon.

- **Finish Assignment 4**
  - Need help? Stop by the LaIR!
Next Time

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

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