Please log onto https://edstem.org/us/courses/21257/discussion/1336583 to submit live lecture questions

Please log onto PollEv.com/champk to answer the daily lecture participation question

Lecture 2: List and Stacks

CSE 373: Data Structures and Algorithms
Agenda

- Dust off data structure cobwebs
- List Case Study
- Stack Case Study
Announcements

Things are live!
- course website – one stop for all things 373
- Canvas – Calendar + office hours
- Ed board – get your course content questions answered + connect with students
- Gradescope

HW 0 Assigned – Due Wednesday
- 143 review
Questions?

Clarification on syllabus, General complaining/moaning
Data Structures and Algorithms

What are they anyway?
Basic Definitions

Data Structure
- A way of organizing and storing data
- Examples from CSE 14X: arrays, linked lists, stacks, queues, trees

Algorithm
- A series of precise instructions to produce to a specific outcome
- Examples from CSE 14X: binary search, merge sort, recursive backtracking
Review: Clients vs Objects

CLIENT CLASSES

A class that is executable, in Java this means it contains a Main method

```
public static void main(String[] args)
```

OBJECT CLASSES

A coded structure that contains data and behavior

Start with the data you want to hold, organize the things you want to enable users to do with that data
Abstract Data Types (ADT)

Abstract Data Types
- An abstract definition for expected operations and behavior
- Defines the input and outputs, not the implementations

Review: List – a collection storing an ordered sequence of elements
- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- in Java, a list can be represented as an ArrayList object
**Review: Interfaces**

**interface**: A construct in Java that defines a set of methods that a class promises to implement
- Interfaces give you an is-a relationship *without* code sharing.
- A `Rectangle` object can be treated as a `Shape` but inherits no code.
- Analogous to non-programming idea of roles or certifications:
  - "I’m certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
  - "I’m ‘certified’ as a `Shape`, because I implement the `Shape` interface. This assures you I know how to compute my area and perimeter."

```java
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

**Example**

```java
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}
```

![Diagram of interfaces and their methods](image)

CSE 373 20 SP – CHUN & CHAMPION 9
Review: Java Collections

Java provides some implementations of ADTs for you!

<table>
<thead>
<tr>
<th>ADTs</th>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>List&lt;Integer&gt; a = new ArrayList&lt;Integer&gt;();</td>
</tr>
<tr>
<td>Stacks</td>
<td>Stack&lt;Character&gt; c = new Stack&lt;Character&gt;();</td>
</tr>
<tr>
<td>Queues</td>
<td>Queue&lt;String&gt; b = new LinkedList&lt;String&gt;();</td>
</tr>
<tr>
<td>Maps</td>
<td>Map&lt;String, String&gt; d = new TreeMap&lt;String, String&gt;();</td>
</tr>
</tbody>
</table>

But some data structures you made from scratch... why?

Linked Lists - LinkedHashMap was a collection of ListNode

Binary Search Trees – SearchTree was a collection of SearchTreeNode
Full Definitions

Abstract Data Type (ADT)
- A definition for expected operations and behavior
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

Data Structure
- A way of organizing and storing related data points
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList
ADTs we’ll discuss this quarter

- List
- Set
- Map
- Stack
- Queue
- Priority Queue
- Graph
- Disjoint Set
Questions?
Case Study: The List ADT

**list:** a collection storing an ordered sequence of elements.
- Each item is accessible by an index.
- A list has a size defined as the number of elements in the list

**Expected Behavior:**
- `get(index)`: returns the item at the given index
- `set(value, index)`: sets the item at the given index to the given value
- `append(value)`: adds the given item to the end of the list
- `insert(value, index)`: insert the given item at the given index maintaining order
- `delete(index)`: removes the item at the given index maintaining order
- `size()`: returns the number of elements in the list

```java
List<String> names = new ArrayList<>();
names.add("Anish");
names.add("Amanda");
names.add(0, "Brian");
```
Case Study: List Implementations

List ADT

**state**
- Set of ordered items
- Count of items

**behavior**
- `get(index)` return item at index
- `set(item, index)` replace item at index
- `append(item)` add item to end of list
- `insert(item, index)` add item at index
- `delete(index)` delete item at index
- `size()` count of items

ArrayList\<E\>

**state**
- `data[]`
- `size`

**behavior**
- `get` return `data[index]`
- `set` `data[index] = value`
- `append` `data[size] = value`, if out of space grow data
- `insert` shift values to make hole at index, `data[index] = value`, if out of space grow data
- `delete` shift following values forward
- `size` return `size`

LinkedList\<E\>

**state**
- `Node front`
- `size`

**behavior**
- `get` loop until index, return node’s value
- `set` loop until index, update node’s value
- `append` create new node, update next of last node
- `insert` create new node, loop until index, update next fields
- `delete` loop until index, skip node
- `size` return `size`

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>list</strong></td>
<td>88.6</td>
<td>26.1</td>
<td>94.4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>free space</strong></td>
<td>88.6</td>
<td>26.1</td>
<td>94.4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ArrayList uses an Array as underlying storage

LinkedList uses nodes as underlying storage
Implementing Insert

**ArrayList<E>**

insert(element, index) with shifting

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

numberOfItems = 4

**LinkedList<E>**

insert(element, index) with shifting

| 10 | 3 | 4 | 5 |

numberOfItems = 4
Implementing Delete

**ArrayList<E>**

`delete(index)` with shifting

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

`numberOfItems = 4`

**LinkedList<E>**

`delete(index)` with shifting

```
10 -> 3 -> 4 -> 5
```

`numberOfItems = 4`
Implementing Append

**ArrayList<E>**

append(element) with growth

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

append(2)

numberOfItems = 5

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**LinkedList<E>**

append(element) with growth

append(2)

numberOfItems = 5
Design Decisions

For every ADT there are lots of different ways to implement them

Based on your situation you should consider:
- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

This class is all about implementing ADTs based on making the right design tradeoffs!
> A common topic in interview questions
Design Decisions

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

   ArrayList – optimize for the ability to shuffle play on the playlist
   LinkedList – optimize for adding/removing/changing order of songs

Situation #2: Write a data structure that implements the List ADT that will be used to store the history of a bank customer’s transactions.

   ArrayList – optimize for accessing of elements and adding to back
   LinkedList – if structured backwards, could optimize for addition to front

Situation #3: Write a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a TA at a tutoring center

   LinkedList – optimize for removal from front
   ArrayList – optimize for addition to back
What is this class about?

CSE 143 – OBJECT ORIENTED PROGRAMMING
- Classes and Interfaces
- Methods, variables and conditionals
- Loops and recursion
- Linked lists and binary trees
- Sorting and Searching
- O(n) analysis
- Generics

CSE 373 – DATA STRUCTURES AND ALGORITHMS
- Design decisions
- Design analysis
- Implementations of data structures
- Debugging and testing
- Abstract Data Types
- Code Modeling
- Complexity Analysis
- Software Engineering Practices
Data Structures and Algorithms

What are they anyway?
Basic Definitions

Data Structure
- A way of organizing and storing data
- Examples from CSE 14X: arrays, linked lists, stacks, queues, trees

Algorithm
- A series of precise instructions to produce to a specific outcome
- Examples from CSE 14X: binary search, merge sort, recursive backtracking
A class that is executable, in Java this means it contains a Main method

```java
public static void main(String[] args)
```

A coded structure that contains data and behavior

Start with the data you want to hold, organize the things you want to enable users to do with that data
Abstract Data Types (ADT)

Abstract Data Types
- An abstract definition for expected operations and behavior
- Defines the input and outputs, not the implementations

Review: List – a collection storing an ordered sequence of elements
- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- in Java, a list can be represented as an ArrayList object
Review: Interfaces

**interface**: A construct in Java that defines a set of methods that a class promises to implement
- Interfaces give you an is-a relationship *without* code sharing.
  - A Rectangle object can be treated as a Shape but inherits no code.
- Analogous to non-programming idea of roles or certifications:
  - "I'm certified as a CPA accountant. This assures you I know how to do taxes, audits, and consulting."
  - "I'm 'certified' as a Shape, because I implement the Shape interface. This assures you I know how to compute my area and perimeter."

```java
public interface Name {
    public <type> name(<type> name, ..., <type> name);
    public <type> name(<type> name, ..., <type> name);
    ...
    public <type> name(<type> name, ..., <type> name);
}
```

Example

```java
// Describes features common to all shapes.
public interface Shape {
    public double area();
    public double perimeter();
}
```

```
Circle<radius>
Circle(radius)
Circle(radius)
Radius(radius)
Circle(radius)
Circle(radius)
Circle(radius)
```
Review: Java Collections

Java provides some implementations of ADTs for you!

**ADTs**

<table>
<thead>
<tr>
<th>ADT</th>
<th>Data Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lists</td>
<td>List&lt;Integer&gt; a = new ArrayList&lt;Integer&gt;();</td>
</tr>
<tr>
<td>Stacks</td>
<td>Stack&lt;Character&gt; c = new Stack&lt;Character&gt;();</td>
</tr>
<tr>
<td>Queues</td>
<td>Queue&lt;String&gt; b = new LinkedList&lt;String&gt;();</td>
</tr>
<tr>
<td>Maps</td>
<td>Map&lt;String, String&gt; d = new TreeMap&lt;String, String&gt;();</td>
</tr>
</tbody>
</table>

But some data structures you made from scratch... why?

- **Linked Lists** - LinkedIntList was a collection of ListNode
- **Binary Search Trees** - SearchTree was a collection of SearchTreeNode
Full Definitions

Abstract Data Type (ADT)
- A definition for expected operations and behavior
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

Data Structure
- A way of organizing and storing related data points
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedListInt, ArrayIntList
ADTs we’ll discuss this quarter

- List
- Set
- Map
- Stack
- Queue
- Priority Queue
- Graph
- Disjoint Set
Case Study: The List ADT

**list:** a collection storing an ordered sequence of elements.
- Each item is accessible by an index.
- A list has a size defined as the number of elements in the list

List<String> names = new ArrayList<>();
names.add("Anish");
names.add("Amanda");
names.add(0, "Brian");

<table>
<thead>
<tr>
<th>Anish</th>
<th>Amanda</th>
<th>Amanda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brian</td>
<td>Anish</td>
<td>Amanda</td>
</tr>
</tbody>
</table>
Case Study: The List ADT

**list:** a collection storing an ordered sequence of elements.
- Each item is accessible by an index.
- A list has a size defined as the number of elements in the list

**Expected Behavior:**
- `get(index):` returns the item at the given index
- `set(value, index):` sets the item at the given index to the given value
- `append(value):` adds the given item to the end of the list
- `insert(value, index):` insert the given item at the given index maintaining order
- `delete(index):` removes the item at the given index maintaining order
- `size():` returns the number of elements in the list
Case Study: List Implementations

**List ADT**

**state**
- Set of ordered items
- Count of items

**behavior**
- `get(index)` return item at index
- `set(item, index)` replace item at index
- `append(item)` add item to end of list
- `insert(item, index)` add item at index
- `delete(index)` delete item at index
- `size()` count of items

**ArrayList**

**state**
- `data[]`
- `size`

**behavior**
- `get` return `data[index]`
- `set` `data[index] = value`
- `append` `data[size] = value`, if out of space grow data
- `insert` shift values to make hole at index, `data[index] = value`, if out of space grow data
- `delete` shift following values forward
- `size` return `size`

**LinkedList**

**state**
- Node front
- `size`

**behavior**
- `get` loop until index, return node’s value
- `set` loop until index, update node’s value
- `append` create new node, update next of last node
- `insert` create new node, loop until index, update next fields
- `delete` loop until index, skip node
- `size` return `size`

**ArrayList**

uses an Array as underlying storage

**LinkedList**

uses nodes as underlying storage

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>88.6</td>
<td>26.1</td>
<td>94.4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

list | free space

| node size | 88.6 | 26.1 | 94.4 | 0 | 0 |

CSE 373 WI - KASEY CHAMPION 32
Implementing ArrayList

**ArrayList**

**state**
- data[]
- size

**behavior**
- get return data[index]
- set data[index] = value
- append data[size] = value, if out of space grow data
- insert shift values to make hole at index, data[index] = value, if out of space grow data
- delete shift following values forward
- size return size

### insert(element, index) with shifting

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>size</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>number</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

### delete(index) with shifting

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
Implementing ArrayList

**ArrayList<E>**

**state**
- data[]
- size

**behavior**
- **get** return data[index]
- **set** data[index] = value
- **append** data[size] = value, if out of space grow data
- **insert** shift values to make hole at index, data[index] = value, if out of space grow data
- **delete** shift following values forward
- **size** return size

**append(element) with growth**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

numberOfItems = 5
Review: Complexity Class

**complexity class:** A category of algorithm efficiency based on the algorithm's relationship to the input size N.

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big-O</th>
<th>Runtime if you double N</th>
<th>Example Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
<td>Accessing an index of an array</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
<td>Binary search</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
<td>Looping over an array</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
<td>Merge sort algorithm</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
<td>Nested loops!</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>Fibonacci with recursion</td>
</tr>
</tbody>
</table>

Note: You don’t have to understand all of this right now – we’ll dive into it soon.
Case Study: List Implementations – Time and Space Complexities

Time

<table>
<thead>
<tr>
<th></th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>get()</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add()</td>
<td>O(1)</td>
<td>O(1) amortized</td>
</tr>
<tr>
<td>remove()</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

Space

ArrayList – O(n)
LinkedList – O(n)

ArrayList uses “more” memory when there are many resizes

LinkedList uses more memory per element (holding node value, previous link, next link, etc.)

Which would you use if there were many insertions and deletions? What if there wasn’t?
Design Decisions

For every ADT there are lots of different ways to implement them

Based on your situation you should consider:
- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

This class is all about implementing ADTs based on making the right design tradeoffs!

> A common topic in interview questions
Design Decisions

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

ArrayList – I want to be able to shuffle play on the playlist

Situation #2: Write a data structure that implements the List ADT that will be used to store the history of a bank customer’s transactions.

ArrayList – optimize for addition to back and accessing of elements

Situation #3: Write a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a TA at a tutoring center

LinkedList – optimize for removal from front

ArrayList – optimize for addition to back
List ADT tradeoffs

Last time: we used “slow” and “fast” to describe running times. Let’s be a little more precise.

Recall these basic Big-O ideas from 14X: Suppose our list has N elements
- If a method takes a constant number of steps (like 23 or 5) its running time is $O(1)$
- If a method takes a linear number of steps (like $4N+3$) its running time is $O(N)$

For ArrayLists and Linked Lists, what is the $O()$ for each of these operations?
- Time needed to access $N^{th}$ element:
- Time needed to insert at end (the array is full!)

What are the memory tradeoffs for our two implementations?
- Amount of space used overall
- Amount of space used per element
List ADT tradeoffs

Time needed to access $N^{th}$ element:
- **ArrayList**: O(1) constant time
- **LinkedList**: O(N) linear time

Time needed to insert at $N^{th}$ element (the array is full!)
- **ArrayList**: O(N) linear time
- **LinkedList**: O(N) linear time

Amount of space used overall
- **ArrayList**: sometimes wasted space
- **LinkedList**: compact

Amount of space used per element
- **ArrayList**: minimal
- **LinkedList**: tiny extra
Real-World Scenarios – Lists

**LinkedList**
- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button
- Dynamic memory allocation – We use linked list of free blocks
- Implementations of other ADTs such as Stacks, Queues, Graphs, etc.

**ArrayList**
- Maintaining Database Records – List of records you want to add / delete from and maintain your order after
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list
Question Break
**Review: What is a Stack?**

**stack:** A collection based on the principle of adding elements and retrieving them in the opposite order.
- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").

![Stack Diagram]

### Stack ADT

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>push(item) add item to top</td>
</tr>
<tr>
<td>Number of items</td>
<td>pop() return and remove item at top</td>
</tr>
<tr>
<td></td>
<td>peek() look at item at top</td>
</tr>
<tr>
<td></td>
<td>size() count of items</td>
</tr>
<tr>
<td></td>
<td>isEmpty() count of items is 0?</td>
</tr>
</tbody>
</table>

### Supported Operations:
- **push(item):** Add an element to the top of stack
- **pop():** Remove the top element and returns it
- **peek():** Examine the top element without removing it
- **size():** How many items are in the stack?
- **isEmpty():** True if there are 1 or more items in stack, false otherwise
Implementing a Stack with an Array

Stack ADT

- **state**
  - Set of ordered items
  - Number of items

- **behavior**
  - `push(item)` add item to top
  - `pop()` return and remove item at top
  - `peek()` look at item at top
  - `size()` count of items
  - `isEmpty()` count of items is 0?

ArrayStack<E>

- **state**
  - `data[]`
  - `size`

- **behavior**
  - `push` `data[size] = value`, if out of room grow `data`
  - `pop` return `data[size - 1]`, size-1
  - `peek` return `data[size - 1]`
  - `size` return `size`
  - `isEmpty` return `size == 0`

<table>
<thead>
<tr>
<th>push(3)</th>
<th>push(4)</th>
<th>pop()</th>
<th>push(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

numberOfItems = 2

Big O Analysis

- `pop()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `push()` O(N) linear if you have to resize
  O(1) otherwise

Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the “push()” operation will be?
Implementing a Stack with Nodes

Stack ADT

**state**
- Set of ordered items
- Number of items

**behavior**
- `push(item)` add item to top
- `pop()` return and remove item at top
- `peek()` look at item at top
- `size()` count of items
- `isEmpty()` count of items is 0?

LinkedStack<E>

**state**
- Node top
- `size`

**behavior**
- `push` add new node at top
- `pop` return and remove node at top
- `peek` return node at top
- `size` return size
- `isEmpty` return size == 0

Big O Analysis

- `push()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `push()` O(1) Constant

Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the “push()” operation will be?
Real-World Scenarios – Stacks

- Undo/Redo Feature in editing software
- DNA Parsing Implementation
  - [https://www.nature.com/articles/s41467-021-25023-6](https://www.nature.com/articles/s41467-021-25023-6)
  - Stack is able to record combinations of two different DNA signals, release the signals into solution in reverse order, and then re-record
  - Social implications + ethical concerns?
    - Accuracy limits
    - Performance of stack dependent on efficiency of “washing steps” between stack operations
    - What if certain DNA needs more stack operations to parse than other? What kind of inequalities can this create between more common and more rare DNA? What are some social consequences of using a stack for DNA sequencing?