CS 261 Lab #5

Is it really midterm time already?
Mix of multiple choice, matching, short answer, true/false, and code

Code needn’t be *perfect*, but should clearly be C
(missing a semi-colon is fine; only writing pseudo-code is not)

Will cover weeks 1 – 5
(everything up to and including binary search *trees*)

Header code will be provided
Be able to determine **Big O runtime** by examining an algorithm (in pseudocode and C) or equation.

Know the Big O runtimes of **common algorithms** (e.g., *binary search*).
What’s the **Big O runtime** for:

A method that takes $3n^2 + 6n + 50$ steps?
What’s the **Big O runtime** for:

A method that takes $3n^2 + 6n + 50$ steps? **$O(n^2)$** because the $n^2$ term dominates.
What’s the **Big O runtime** for:

A method that takes $3n^2 + 6n + 50$ steps?

**$O(n^2)$** because the $n^2$ term dominates

Binary search?
What’s the **Big O runtime** for:

A method that takes $3n^2 + 6n + 50$ steps?

**$O(n^2)$** because the $n^2$ term dominates

Binary search?

**$O(\log n)$** because it halves the search space on each iteration
What’s the Big O runtime for:

A method that takes $3n^2 + 6n + 50$ steps?
- $O(n^2)$ because the $n^2$ term dominates

Binary search?
- $O(\log n)$ because it halves the search space on each iteration

```java
for (int i = n; i > 0; i = i / 2) {
    // constant-time operations
}
```
What’s the **Big O runtime** for:

A method that takes $3n^2+6n+50$ steps?

**$O(n^2)$** because the $n^2$ term dominates.

Binary search?

**$O(\log n)$** because it halves the search space on each iteration.

```java
for (int i = n; i > 0; i = i / 2) {
    // constant-time operations
}
```

**$O(\log n)$** because the counter it halved on each iteration.
What’s the **Big O runtime** for:

```java
def for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        // constant-time operations
    }
}
```
What’s the **Big O runtime** for:

```java
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        // constant-time operations
    }
}
```

**$O(n^2)$** because the outer loop will run $n$ times, and each time the inner loop can run up to $n$ times.
What’s the **Big O runtime** for:

```java
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        // constant-time operations
    }
}
```

**$O(n^2)$** because the outer loop will run $n$ times, and each time the inner loop can run up to $n$ times.

```java
iterator = list->frontSentinel->next;
while (iterator != list->backSentinel) {
    if (iterator->value == value)
        return 1;
    iterator = iterator->next;
}
```
What’s the **Big O runtime** for:

```java
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        // constant-time operations
    }
}
```

**O(n²)** because the outer loop will run \( n \) times, and each time the inner loop can run up to \( n \) times.

```java
iterator = list->frontSentinel->next;
while (iterator != list->backSentinel) {
    if (iterator->value == value)
        return 1;
    iterator = iterator->next;
}
```

**O(n)** because it needs to check each element.
Know the **properties** and **operations** of the data types we’ve covered (e.g., *stack*, *queue*, *dynamic array*, etc.)

Be able to **compare the Big O runtimes** of common operations on different data types

Understand situations when **one data type is preferable** to another
What are the three operations of a **stack** ADT?

Which ADT would be good for **finite-length undo**?
What are the three operations of a stack ADT?

**push, pop, & top**

Which ADT would be good for **finite-length undo**?
What are the three operations of a stack ADT?

**push, pop, & top**

Which ADT would be good for finite-length undo?

**deque**

(need to remove old entries to have finite length)
What’s the **ordering** property of a **stack**?
What’s the **ordering** property of a **stack**?

last in, first out
What’s the **ordering** property of a **stack**?

*last in, first out*

What about a **queue**?
What’s the **ordering** property of a **stack**?

*last in, first out*

What about a **queue**?

*first in, first out*
What’s the **ordering** property of a **stack**?

*last in, first out*

What about a **queue**?

*first in, first out*

Does the **bag** ADT have an **ordering** property?
What’s the **ordering** property of a **stack**?

**last in, first out**

What about a **queue**?

**first in, first out**

Does the **bag** ADT have an **ordering** property?

**nope, but the ordered bag** does
What are the **average** and **worst-case** Big O runtimes for the the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contains</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addFirst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>O(1+)</td>
<td></td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addFirst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the deque and bag interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td><strong>O(1+)</strong></td>
<td><strong>O(1)</strong></td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td><strong>O(n)</strong></td>
<td><strong>O(1)</strong></td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td><strong>O(n)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td><strong>O(n)</strong></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td><strong>O(1+)</strong></td>
<td><strong>O(n)</strong></td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the deque and bag interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th>Method</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>$O(1+)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the deque and bag interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th></th>
<th>linked list</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>addFirst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the deque and bag interfaces on a dynamic array versus a linked list?

<table>
<thead>
<tr>
<th>Method</th>
<th>Dynamic Array (Average)</th>
<th>Dynamic Array (Worst)</th>
<th>Linked List (Average)</th>
<th>Linked List (Worst)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the deque and bag interfaces on a dynamic array versus a linked list?

<table>
<thead>
<tr>
<th>Method</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>$O(1+)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td>$O(1+)$</td>
<td></td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the deque and bag interfaces on a dynamic array versus a linked list?

<table>
<thead>
<tr>
<th>Method</th>
<th>Dynamic Array</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Worst</td>
</tr>
<tr>
<td>add</td>
<td>$O(1^+)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>contains</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>addFirst</td>
<td>$O(1^+)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the deque and bag interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addLast</td>
<td></td>
<td>O(n)</td>
</tr>
</tbody>
</table>

**ADTs**

**dynamic array**

**linked list**
What are the **average** and **worst-case** Big O runtimes for the deque and bag interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>average</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(1)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(1+)</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>worst</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the deque and bag interfaces on a dynamic array versus a linked list?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addLast</td>
<td>O(1+)</td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th></th>
<th>linked list</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(1+)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>addLast</td>
<td>O(1+)</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the **deque** and **bag** interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td><strong>add</strong></td>
<td>(O(1+))</td>
<td>(O(n))</td>
</tr>
<tr>
<td><strong>contains</strong></td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td><strong>addFirst</strong></td>
<td>(O(1+))</td>
<td>(O(n))</td>
</tr>
<tr>
<td><strong>addLast</strong></td>
<td>(O(1+))</td>
<td>(O(n))</td>
</tr>
</tbody>
</table>
What are the **average** and **worst-case** Big O runtimes for the the deque and bag interfaces on a **dynamic array** versus a **linked list**?

<table>
<thead>
<tr>
<th></th>
<th>dynamic array</th>
<th>linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>average</td>
<td>worst</td>
</tr>
<tr>
<td>add</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>contains</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
<tr>
<td>addLast</td>
<td>O(1+)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Be able to write and understand C code that uses or builds upon our ADTs.

Be able to show the state of an ADT after a series of operations have been performed on it.
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>0</td>
</tr>
<tr>
<td>beginning</td>
<td>0</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<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>capacity</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beginning</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capacity</th>
<th>count</th>
<th>beginning</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>1</td>
</tr>
<tr>
<td>beginning</td>
<td>0</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0</td>
<td>5.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>3</td>
</tr>
<tr>
<td>beginning</td>
<td>0</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

```
<table>
<thead>
<tr>
<th>count</th>
<th>beginning</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
</table>
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>capacity</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>beginning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<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>capacity</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beginning</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th></th>
<th>3.0</th>
<th>5.0</th>
<th>1.0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>capacity</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>count</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>beginning</td>
<td>1</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

```
<table>
<thead>
<tr>
<th></th>
<th>3.0</th>
<th>5.0</th>
<th>1.0</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>count</td>
<td>capacity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<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>capacity</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beginning</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th></th>
<th>8.0</th>
<th>5.0</th>
<th>1.0</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8.0</td>
<td>5.0</td>
<td>1.0</td>
<td>2.0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

capacity: 5
count: 3
beginning: 0
Show the state of a dynamic array after each of the following operations:

```c
dstruct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

```
<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4</td>
</tr>
<tr>
<td>beginning</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.0</td>
<td>5.0</td>
<td>1.0</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>4</td>
</tr>
<tr>
<td>beginning</td>
<td>0</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>count</th>
<th>8.0</th>
<th>5.0</th>
<th>1.0</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginning</td>
<td>0</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<table>
<thead>
<tr>
<th>capacity</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>3</td>
</tr>
<tr>
<td>beginning</td>
<td>1</td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```

<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>8.0</td>
<td>5.0</td>
<td>1.0</td>
<td>2.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>capacity</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>beginning</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Show the state of a dynamic array after each of the following operations:

```c
struct dynArrDeque d;
initDynArrDeque(&d, 5);
addBackArrDeque(&d, 3.0);
addBackArrDeque(&d, 5.0);
addBackArrDeque(&d, 1.0);
removeFrontArrDeque(&d);
addBackArrDeque(&d, 2.0);
addFrontArrDeque(&d, 8.0);
removeBackArrDeque(&d);
removeFrontArrDeque(&d);
```
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to `pop` to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to \texttt{pop} to get the value \texttt{8} off of the stack?

```c
struct dynArrStack s:
    dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;

dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

3
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
```

```c
  dynArrStackPush(&s, 7);  // Marked for deletion
```
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
    dynArrStackPush(&s, 7);
    dynArrStackPush(&s, 2);
    dynArrStackPush(&s, 8);
    dynArrStackTop(&s);
    dynArrStackPush(&s, 5);
    dynArrStackPush(&s, 1);
    dynArrStackPush(&s, 1);
    dynArrStackPop(&s);
    dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to `pop` to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
```

```c
dynArrStackPush(&s, 2);
```
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
**dynArrStackPush(&s, 2);**
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

The value 8 is on the stack after the 8th push. To get 8 off the stack, you would need to pop it once.
After the following instructions execute, how many times would you need to `pop` to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);

// dynArrStackTop(&s);
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
```

```
dynArrStackPush(&s, 5);
```

```
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to `pop` to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
```

| 5 |
| 8 |
| 2 |
| 7 |
| 3 |
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
```

```
5
8
2
7
3
```
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
```
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
```

After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
```

<table>
<thead>
<tr>
<th>Pop Count</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to \texttt{pop} to get the value 8 off of the stack?

```
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

3 7 2 8 5 1
After the following instructions execute, how many times would you need to `pop` to get the value `8` off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<table>
<thead>
<tr>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to \texttt{pop} to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<table>
<thead>
<tr>
<th>5</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
After the following instructions execute, how many times would you need to pop to get the value 8 off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

- pop
- pop
- pop
- 8
- 2
- 7
- 3
After the following instructions execute, how many times would you need to **pop** to get the value **8** off of the stack?

```c
struct dynArrStack s;
dynArrStackInit(&s, 8);
dynArrStackPush(&s, 3);
dynArrStackPush(&s, 7);
dynArrStackPush(&s, 2);
dynArrStackPush(&s, 8);
dynArrStackTop(&s);
dynArrStackPush(&s, 5);
dynArrStackPush(&s, 1);
dynArrStackPush(&s, 1);
dynArrStackPop(&s);
dynArrStackPush(&s, 9);
```

<table>
<thead>
<tr>
<th>3</th>
<th>7</th>
<th>2</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td>pop</td>
<td>pop</td>
<td>top</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Write a function to print each value of a linked list:

```c
struct SLink {
    TYPE value;
    struct SLink *next;
};

struct List {
    struct SLink *frontSntl;
    struct SLink *backSntl;
};

void _printList(struct List *list) {
}
```
Write a function to print each value of a linked list:

```c
struct SLink {
    TYPE value;
    struct SLink *next;
};

struct List {
    struct SLink *frontSntl;
    struct SLink *backSntl;
};

void _printList(struct List *list) {
    struct SLink *current;
}
```
Write a function to print each value of a linked list:

```c
struct SLink {
    char value;
    struct SLink *next;
};

struct List {
    struct SLink *frontSntl;
    struct SLink *backSntl;
};

void _printList(struct List *list) {
    struct SLink *current;
    current = list->frontSntl->next;
    // Function body here
}
```
Write a function to print each value of a linked list:

```c
struct SLink {
    TYPE value;
    struct SLink *next;
};

struct List {
    struct SLink *frontSntl;
    struct SLink *backSntl;
};

void _printList(struct List *list) {
    struct SLink *current;
    current = list->frontSntl->next;
    while (current != list->backSntl) {
        // Print the value of the current node
        current = current->next;
    }
}
```
Write a function to print each value of a linked list:

```c
struct SLink {
    TYPE value;
    struct SLink *next;
};

struct List {
    struct SLink *frontSntl;
    struct SLink *backSntl;
};

void _printList(struct List *list) {
    struct SLink *current;
    current = list->frontSntl->next;
    while (current != list->backSntl) {
        printf("Value = %d\n", current->value);
        current = current->next;
    }
}
```
Write a function to **print each value** of a linked list:

```c
struct SLink {
    TYPE value;
    struct SLink *next;
};

struct List {
    struct SLink *frontSntl;
    struct SLink *backSntl;
};

void _printList(struct List *list) {
    struct SLink *current;
    current = list->frontSntl->next;
    while (current != list->backSntl) {
        printf("Value = %d\n", current->value);
        current = current->next;
    }
}
```
Write the Iterator functions

\textbf{next()} and \textbf{hasNext()} for a linked list:

```c
struct ListIterator {
  struct List *list;
  struct SLink *current;
};

int hasNext(struct ListIterator *itr) {
}

TYPE next (struct ListIterator *itr) {
}
```
Write the Iterator functions `next()` and `hasNext()` for a linked list:

```c
struct List *list;
struct SLink *current;
};

int hasNext(struct ListIterator *itr) {
    if (itr->current->next != itr->list->backSntl) {
        // Your implementation here
    }
}

TYPE next (struct ListIterator *itr) {
    // Your implementation here
}
```
Write the Iterator functions `next()` and `hasNext()` for a linked list:

```c
struct ListIterator {
    struct List *list;
    struct SLink *current;
};

int hasNext(struct ListIterator *itr) {
    if (itr->current->next != itr->list->backSntl) {
        itr->current = itr->current->next;
    }
}
```

```c
TYPE next (struct ListIterator *itr) {
}
```
Write the Iterator functions `next()` and `hasNext()` for a linked list:

```c
struct ListIterator {
    struct List *list;
    struct SLink *current;
};

int hasNext(struct ListIterator *itr) {
    if (itr->current->next != itr->list->backSntl) {
        itr->current = itr->current->next;
        return 1;
    }
}

TYPE next (struct ListIterator *itr) {
}
```
Write the Iterator functions \texttt{next()} and \texttt{hasNext()} for a linked list:

```c
struct ListIterator {
    struct List *list;
    struct SLink *current;
};

int hasNext(struct ListIterator *itr) {
    if (itr->current->next != itr->list->backSntl) {
       (itr->current = itr->current->next;
        return 1;
    } else {
        return 0;
    }
}

TYPE next (struct ListIterator *itr) {
}
```
Write the Iterator functions **next()** and **hasNext()** for a linked list:

```c
struct ListIterator {
    struct List *list;
    struct SLink *current;
};

int hasNext(struct ListIterator *itr) {
    if (itr->current->next != itr->list->backSntl) {
        itr->current = itr->current->next;
        return 1;
    } else {
        return 0;
    }
}

TYPE next (struct ListIterator *itr) {
    return itr->current->value;
}
```
Use the provided _binarySearch() function to **implement a contains() function** that runs in **O(log n)** time

```c
int _binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {

}
```
Use the provided `_binarySearch()` function to **implement a contains() function** that runs in **O(log n)** time

```c
int _binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {
    int index;
}
```
Use the provided `_binarySearch()` function to **implement a contains() function** that runs in \(O(\log n)\) time.

```c
int _binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {
    int index;
    index = _binarySearch(da->data, da->size, value);
}
```
Use the provided \_binarySearch() function to implement a \texttt{contains()} function that runs in $O(\log n)$ time.

```c
int _binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {
    int index;
    index = _binarySearch(da->data, da->size, value);
    if (index < da->size) {
        
    }
}
```
Use the provided _binarySearch() function to implement a contains() function that runs in O(log n) time

```c
int _binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {
    int index;
    index = _binarySearch(da->data, da->size, value);

    if (index < da->size) {
        if (da->data[index] == value) {
        }
    }
}
```
Use the provided \_binarySearch() function to **implement a contains() function** that runs in \(O(\log n)\) time.

```c
int \_binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {
    int index;
    index = \_binarySearch(da->data, da->size, value);
    if (index < da->size) {
        if (da->data[index] == value) {
            return 1;
        }
    }
}
```
Use the provided `_binarySearch()` function to **implement a contains() function** that runs in \( O(\log n) \) time

```c
int _binarySearch(TYPE *data, int count, TYPE value);

int contains (struct DynArr *da, TYPE value) {
    int index;
    index = _binarySearch(da->data, da->size, value);

    if (index < da->size) {
        if (da->data[index] == value) {
            return 1;
        }
    }

    return 0;
}
```
This was practice... midterm questions will be different!

xkcd #135
That’s all!

Any questions?