AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Array-based Bag

COMP 2210 – Dr. Hendrix

A Bag collection

A **bag** or multiset is a collection of elements where there is no particular order and duplicates are allowed. This is essentially what `java.util.Collection` describes.

We will **specify the behavior** of this collection with an **interface**:

A Bag collection

A **bag** or multiset is a collection of elements where there is no particular order and duplicates are allowed. This is essentially what `java.util.Collection` describes.

We will **specify the behavior** of this collection with an **interface**:

A subset of the JCF Collection interface

```
import java.util.Iterator;
public interface Bag<T> extends Iterable<T>{
   boolean     add(T element);
   boolean     remove(T element);
   boolean     contains(T element);
   int         size();
   boolean     isEmpty();
   Iterator<T> iterator();
}
```

ArrayBag

We will **implement the behavior** of the collection with a **class**.

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {













}
```

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {







    public boolean add(T element) { . . . }
    public boolean remove(T element) { . . . }
    public boolean contains(T element) { . . . }
    public int size() { . . . }
    public boolean isEmpty() { . . . }
    public Iterator<T> iterator() { . . . }

}
```

Implement all interface methods

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {






   public ArrayBag() { . . . }

   public boolean add(T element) { . . . }
   public boolean remove(T element) { . . . }
   public boolean contains(T element) { . . . }
   public int size() { . . . }
   public boolean isEmpty() { . . . }
   public Iterator<T> iterator() { . . . }

}
```

Provide a constructor

Implement all interface methods

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {




    public ArrayBag() { . . . }

    public boolean add(T element) { . . . }
    public boolean remove(T element) { . . . }
    public boolean contains(T element) { . . . }
    public int size() { . . . }
    public boolean isEmpty() { . . . }
    public Iterator<T> iterator() { . . . }

}
```

Provide physical storage

Provide a constructor

Implement all interface methods

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {




    public ArrayBag() { . . . }

    public boolean add(T element) { . . . }
    public boolean remove(T element) { . . . }
    public boolean contains(T element) { . . . }
    public int size() { . . . }
    public boolean isEmpty() { . . . }
    public Iterator<T> iterator() { . . . }

}
```

Provide physical storage

Choose an appropriate data structure that will efficiently support the collection methods.

Provide a constructor

Implement all interface methods

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {

    private T[] elements;



    public ArrayBag() { . . . }

    public boolean add(T element) { . . . }
    public boolean remove(T element) { . . . }
    public boolean contains(T element) { . . . }
    public int size() { . . . }
    public boolean isEmpty() { . . . }
    public Iterator<T> iterator() { . . . }

}
```

Provide physical storage

Choose an appropriate data structure that will efficiently support the collection methods.

Provide a constructor

Implement all interface methods

ArrayBag

We will **implement the behavior** of the collection with a **class**.

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {

    private T[] elements;

    private int size;

    public ArrayBag() { . . . }

    public boolean add(T element) { . . . }
    public boolean remove(T element) { . . . }
    public boolean contains(T element) { . . . }
    public int size() { . . . }
    public boolean isEmpty() { . . . }
    public Iterator<T> iterator() { . . . }

}
```

Provide physical storage

Add a convenience field

Provide a constructor

Implement all interface methods

Choose an appropriate data structure that will efficiently support the collection methods.

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {


   private T[] elements;
   private int size;














}
```

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {


   private T[] elements;
   private int size;

   public ArrayBag() {


   }






}
```

ArrayBag – constructor

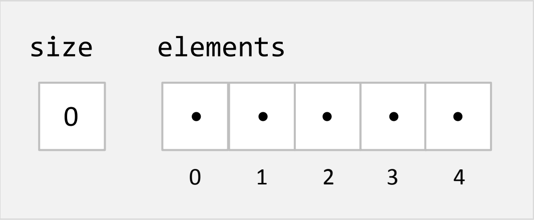
```
public class ArrayBag<T> implements Bag<T> {


  private T[] elements;
  private int size;

  public ArrayBag() {


  }






}
```

```
Bag bag = new ArrayBag();
```

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {


    private T[] elements;
    private int size;

    public ArrayBag() {



    }






}
```

```
Bag bag = new ArrayBag();
```



size

0

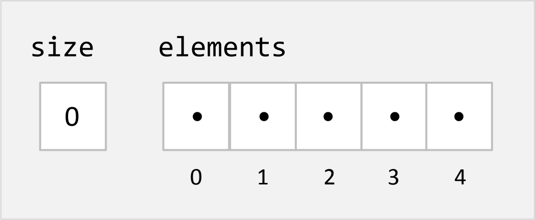
elements

•	•	•	•	•
0	1	2	3	4

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {


   private T[] elements;
   private int size;

   public ArrayBag() {


   }



   public ArrayBag(int capacity) {


   }
}
```

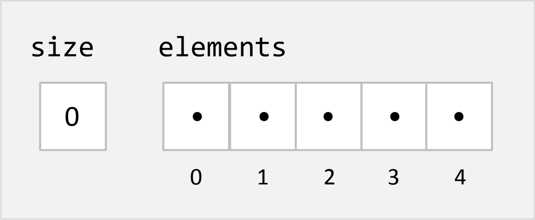
```
Bag bag = new ArrayBag();
```



ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {


   private T[] elements;
   private int size;

   public ArrayBag() {

      this(DEFAULT_CAPACITY);

   }



   public ArrayBag(int capacity) {



   }
}
```

```
Bag bag = new ArrayBag();
```



size elements

0

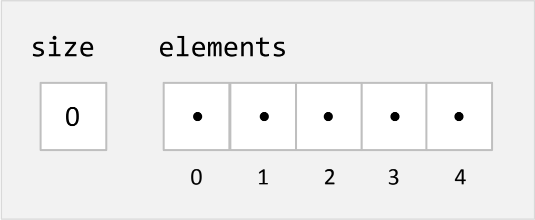
• • • • •

0 1 2 3 4

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
    private static final int DEFAULT_CAPACITY = 5;

    private T[] elements;
    private int size;

    public ArrayBag() {

        this(DEFAULT_CAPACITY);

    }



    public ArrayBag(int capacity) {



    }
}
```

```
Bag bag = new ArrayBag();
```



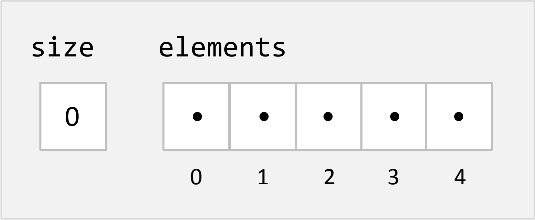
ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
    private static final int DEFAULT_CAPACITY = 5;
    private T[] elements;
    private int size;

    public ArrayBag() {

        this(DEFAULT_CAPACITY);

    }
```

Design decision: Should this constructor be public or private?

```
    public ArrayBag(int capacity) {


    }
}
```

```
Bag bag = new ArrayBag();
```



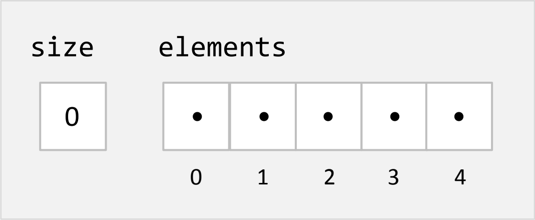
ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
    private static final int DEFAULT_CAPACITY = 5;

    private T[] elements;
    private int size;


    public ArrayBag() {

        this(DEFAULT_CAPACITY);
    }
```

Design decision: Should this constructor be public or private?

```
    public ArrayBag(int capacity) {


    }
}
```

```
Bag bag = new ArrayBag();
```



```
bag = new ArrayBag(3);
```

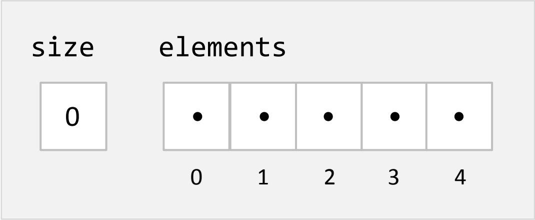
ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
    private static final int DEFAULT_CAPACITY = 5;

    private T[] elements;
    private int size;

    public ArrayBag() {

        this(DEFAULT_CAPACITY);
    }
```

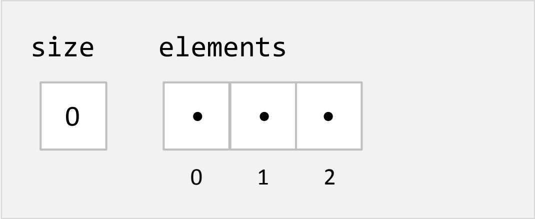
Design decision: Should this constructor be public or private?

```
    public ArrayBag(int capacity) {


    }
}
```

```
Bag bag = new ArrayBag();
```



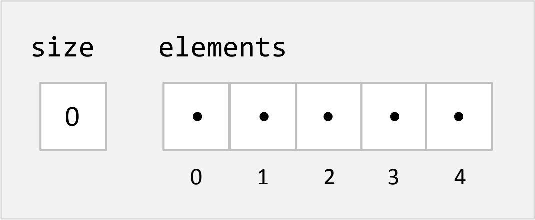
```
bag = new ArrayBag(3);
```



ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;




   public ArrayBag(int capacity) {


   }




}
```

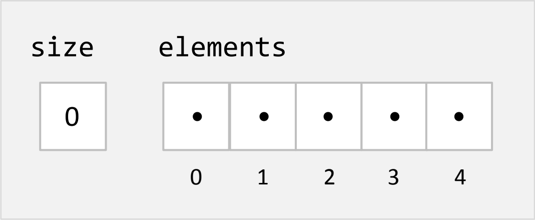
```
Bag bag = new ArrayBag(5);
```



ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;




   public ArrayBag(int capacity) {
      elements = (T[]) new Object[capacity];
      size = 0;
   }




}
```

```
Bag bag = new ArrayBag(5);
```

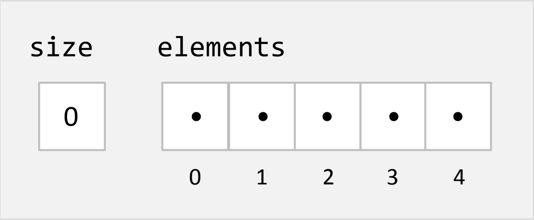


ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;



   public ArrayBag(int capacity) {
      elements = (T[]) new Object[capacity];
      size = 0;
   }


}
```

This will generate a type-safety warning that can’t be eliminated.

```
Bag bag = new ArrayBag(5);
```



ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;
```

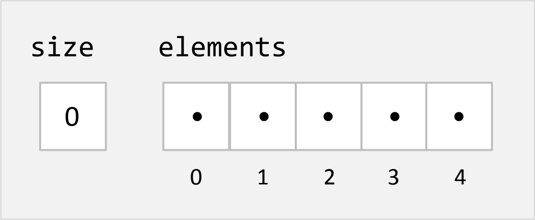
This annotation will suppress the notification.

```
   @SuppressWarnings(“unchecked”)
   public ArrayBag(int capacity) {
      elements = (T[]) new Object[capacity];
      size = 0;
   }
```

This will generate a type-safety warning that can’t be eliminated.

```
}
```

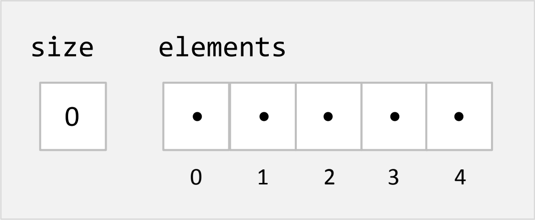
```
Bag bag = new ArrayBag(5);
```



ArrayBag – size and isEmpty

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;

   public int size() {


   }

   public boolean isEmpty() {


   }

}
```

```
Bag bag = new ArrayBag(5);
```



size

0

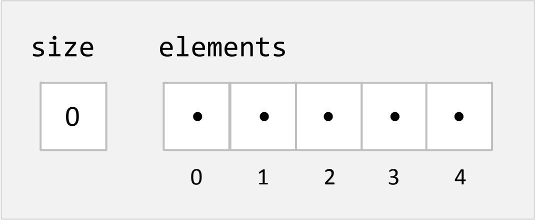
elements

•	•	•	•	•
0	1	2	3	4

ArrayBag – size and isEmpty

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;


   public int size() {


   }


   public boolean isEmpty() {


   }

}
```

```
Bag bag = new ArrayBag(5);
```



size

0

elements

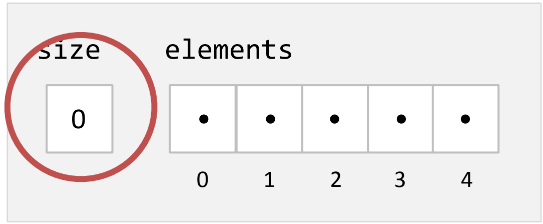
•	•	•	•	•
0	1	2	3	4

These can be fast and trivial with O(1) time complexity.

ArrayBag – size and isEmpty

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;


   public int size() {


   }


   public boolean isEmpty() {


   }

}
```

```
Bag bag = new ArrayBag(5);
```

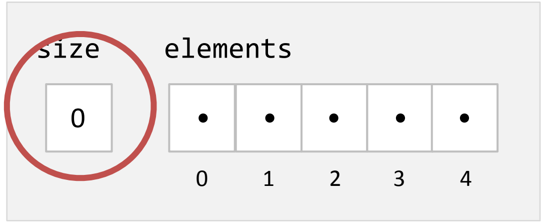


These can be fast and trivial with O(1) time complexity.

ArrayBag – size and isEmpty

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;

   public int size() {
      return size;
   }

   public boolean isEmpty() {


   }

}
```

```
Bag bag = new ArrayBag(5);
```

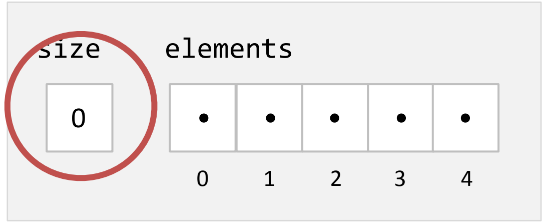


These can be fast and trivial with O(1) time complexity.

ArrayBag – size and isEmpty

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 5;
   private T[] elements;
   private int size;

   public int size() {
      return size;
   }

   public boolean isEmpty() {
      return size == 0;
   }

}
```

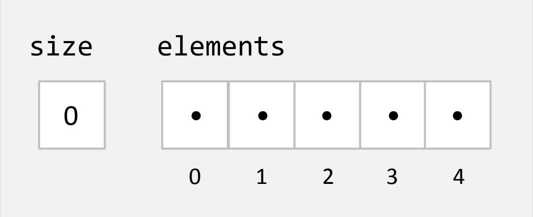
```
Bag bag = new ArrayBag(5);
```



These can be fast and trivial with O(1) time complexity.

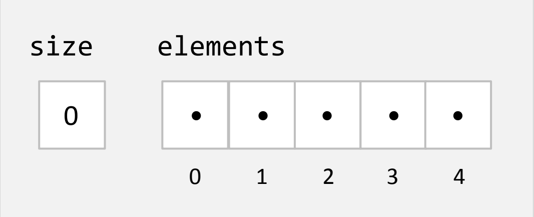
ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {









   }
}
```



ArrayBag – add()

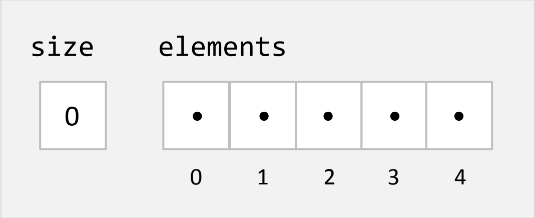
```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean add(T element) {










  }
}
```



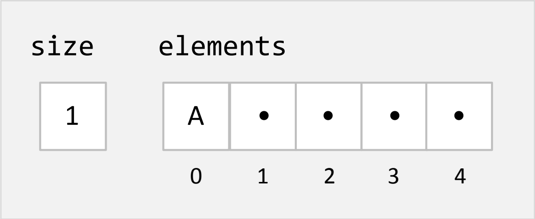
```
bag.add(“A”);
```

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {









   }
}
```

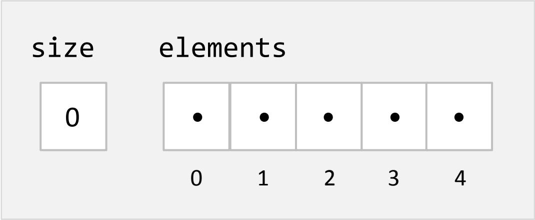


```
bag.add(“A”);
```

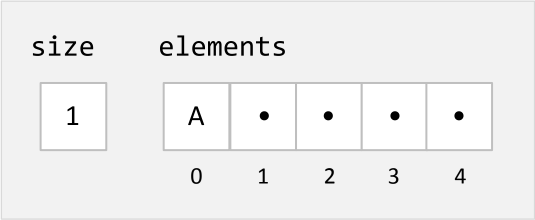


ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {









   }
}
```



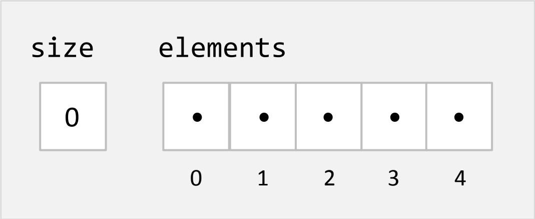
bag.add(“A”);



bag.add(“B”);

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean add(T element) {









  }
}
```



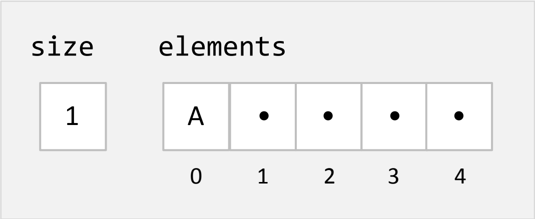
size

0

elements

•	•	•	•	•
0	1	2	3	4

```
bag.add(“A”);
```



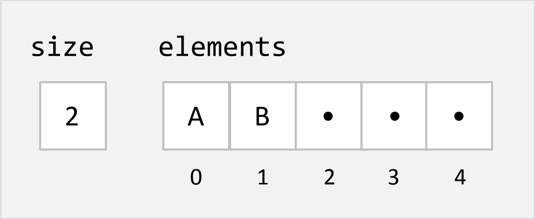
size

1

elements

A	•	•	•	•
0	1	2	3	4

```
bag.add(“B”);
```



size

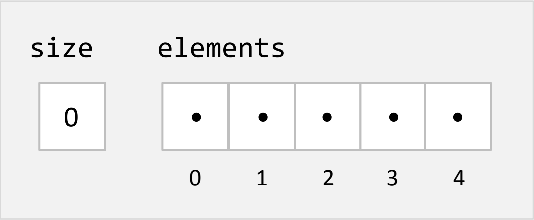
2

elements

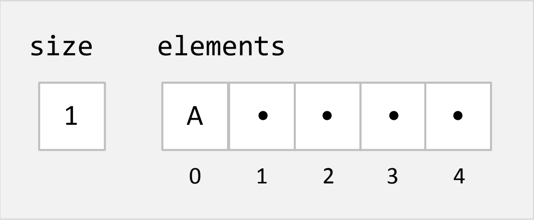
A	B	•	•	•
0	1	2	3	4

ArrayBag – add()

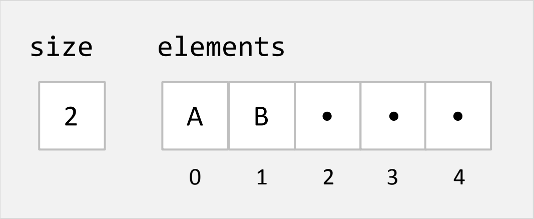
```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;





   }
}
```



bag.add(“A”);



bag.add(“B”);

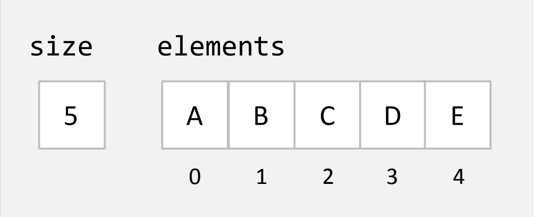


ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;






   }
}
```

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;





   }
}
```



size

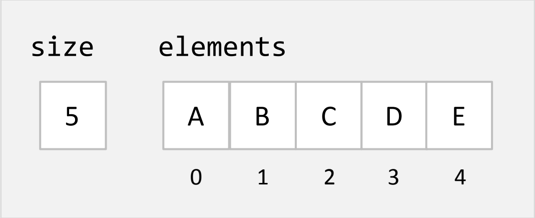
5

elements

A	B	C	D	E
0	1	2	3	4

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;





   }
}
```



size

5

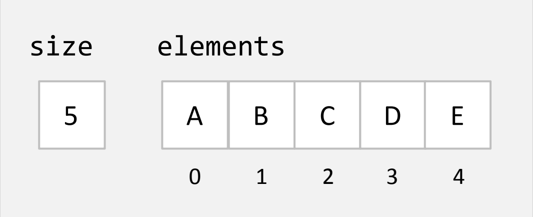
elements

A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean add(T element) {

     elements[size] = element;
     size++;
     return true;





  }
}
```



size

5

elements

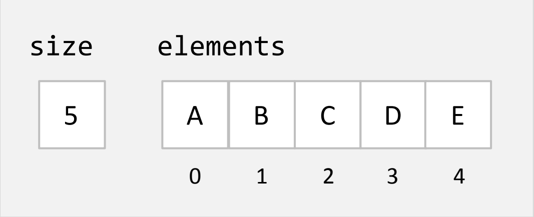
A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

What happens at this point?

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;





   }
}
```



	size	elements				
	5	A	B	C	D	E
		0	1	2	3	4

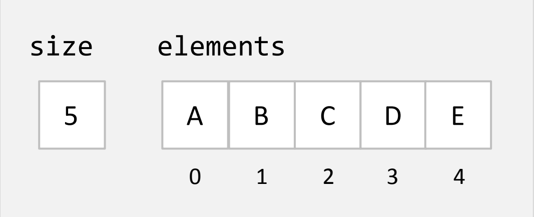
```
bag.add(“F”);
```

What happens at this point?

Options?

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;




   }
}
```



size

5

elements

A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

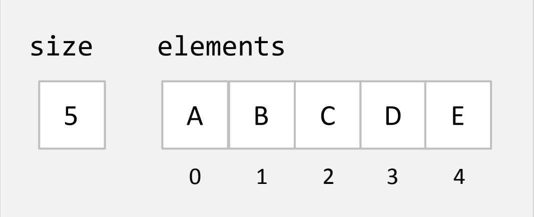
What happens at this point?

Options?

Ignore and return false

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;





   }
}
```



size

5

elements

A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

What happens at this point?

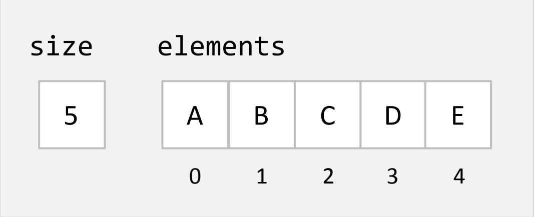
Options?

Ignore and return false

Throw an exception

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      elements[size] = element;
      size++;
      return true;




   }
}
```



size	elements				
5	A	B	C	D	E
	0	1	2	3	4

```
bag.add(“F”);
```

What happens at this point?

Options?

Ignore and return false

Throw an exception

Get a bigger array

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {




      elements[size] = element;
      size++;
      return true;

   }
}
```

size

5

elements

A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

What happens at this point?

Options?

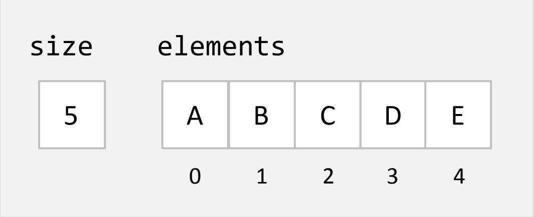
Ignore and return false

Throw an exception

Get a bigger array

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {

      }

      elements[size] = element;
      size++;
      return true;

   }
}
```



size

5

elements

A	B	C	D	E
0	1	2	3	4

`bag.add(“F”);`

What happens at this point?

Options?

Ignore and return false

Throw an exception

Get a bigger array

ArrayBag – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;

   }
}
```

size	elements				
5	A	B	C	D	E
	0	1	2	3	4

`bag.add(“F”);`

What happens at this point?

Options?

Ignore and return false

Throw an exception

Get a bigger array

ArrayBag – add() testing

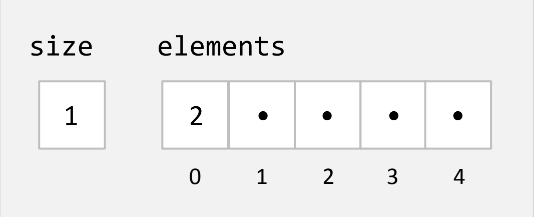
```
public class ArrayBagTest {













}
```

ArrayBag – add() testing

```
public class ArrayBagTest {

   @Test public void addTest1() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      boolean actual = bag.add(2);
      Assert.assertEquals(expected, actual);
   }







}
```

ArrayBag – add() testing

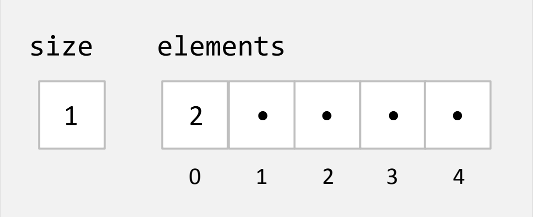
```
public class ArrayBagTest {

   @Test public void addTest1() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      boolean actual = bag.add(2);
      Assert.assertEquals(expected, actual);
   }







}
```



size	elements				
1	2	•	•	•	•
	0	1	2	3	4

ArrayBag – add() testing

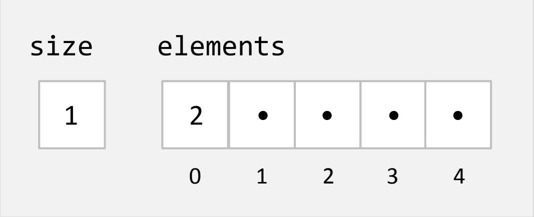
```
public class ArrayBagTest {

   @Test public void addTest1() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      boolean actual = bag.add(2);
      Assert.assertEquals(expected, actual);
   }



}
```



Note that we have no access to the fields size and elements from the test case methods.

ArrayBag – add() testing

```
public class ArrayBagTest {

   @Test public void addTest1() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      boolean actual = bag.add(2);
      Assert.assertEquals(expected, actual);
   }





}
```



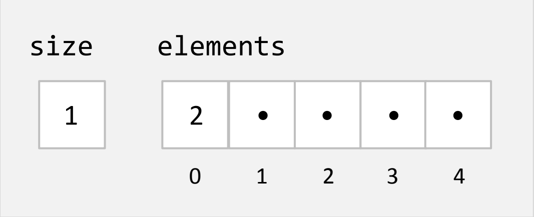
size	elements				
1	2	•	•	•	•
	0	1	2	3	4

Note that we have no access to the fields size and elements from the test case methods.

Only testing the return value is not enough. We have to test the interactions among add and other methods.

ArrayBag – add() testing

```
public class ArrayBagTest {

   @Test public void addTest1() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      boolean actual = bag.add(2);
      Assert.assertEquals(expected, actual);
   }


   @Test public void addTest2() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      int expected = 1;
      bag.add(2);
      int actual = bag.size();
      Assert.assertEquals(expected, actual);
}  }
```

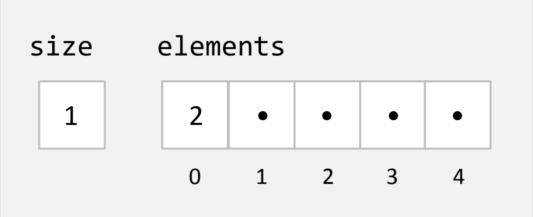


Note that we have no access to the fields size and elements from the test case methods.

Only testing the return value is not enough. We have to test the interactions among add and other methods.

ArrayBag – add() testing

```
public class ArrayBagTest {

   @Test public void addTest3() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      bag.add(2);
      boolean actual = bag.contains(2);
      Assert.assertEquals(expected, actual);
   }

   @Test public void addTest4() {
      Bag<Integer> bag = new ArrayBag<Integer>();
      boolean expected = true;
      bag.add(2);
      boolean actual = bag.remove(2);
      Assert.assertEquals(expected, actual);
}  }
```



Note that we have no access to the fields size and elements from the test case methods.

Only testing the return value is not enough. We have to test the interactions among add and other methods.

ArrayBag – add() efficiency

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

Time Complexity:

ArrayBag – add() efficiency

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

Time Complexity: **O(1)**

ArrayBag – add() efficiency

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

Time Complexity: **O(1)**

We can add a new element to the bag in constant time. That is, no matter how large the bag grows, it always takes the same amount of time to add a new element.

ArrayBag – add() refactoring

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean add(T element) {

     if (size == elements.length) {
        return false;
     }

     elements[size] = element;
     size++;
     return true;
  }

}
```

ArrayBag – add() refactoring

```java
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean add(T element) {

     if (size == elements.length) {
        return false;
     }

     elements[size] = element;
     size++;
     return true;
  }

}
```

REFACTORING

http://www.refactoring.com/

ArrayBag – add() refactoring

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

REFACTORING

http://www.refactoring.com/

Extract Method:

“Turn [a] fragment into a method whose name explains the purpose of the method.”

ArrayBag – add() refactoring

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (size == elements.length) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

}
```

REFACTORING

http://www.refactoring.com/

Extract Method:

“Turn [a] fragment into a method whose name explains the purpose of the method.”

ArrayBag – add() refactoring

```java
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean add(T element) {

     if (size == elements.length) {
        return false;
     }

     elements[size] = element;
     size++;
     return true;
  }

}
```

isFull

REFACTORING

http://www.refactoring.com/

Extract Method:

"Turn [a] fragment into a method whose name explains the purpose of the method."

ArrayBag – add() refactoring

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {

      if (isFull()) {
         return false;
      }

      elements[size] = element;
      size++;
      return true;
   }

   private boolean isFull() {
      return size == elements.length;
   }
}
```

REFACTORING

http://www.refactoring.com/

Extract Method:

“Turn [a] fragment into a method whose name explains the purpose of the method.”

This isn’t strictly necessary, but:

- It increases readability.
- It increases maintainability.

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
   boolean     add(T element);
   boolean     remove(T element);
   boolean     contains(T element);
   int         size();
   boolean     isEmpty();
   Iterator<T> iterator();
}
```

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
   boolean     add(T element);
   boolean     remove(T element);
   boolean     contains(T element);
  ✓int         size();
   boolean     isEmpty();
   Iterator<T> iterator();
}
```

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
   boolean      add(T element);
   boolean      remove(T element);
   boolean      contains(T element);
  ✓int          size();
  ✓boolean      isEmpty();
   Iterator<T> iterator();
}
```

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
  ✓boolean     add(T element);
   boolean     remove(T element);
   boolean     contains(T element);
  ✓int         size();
  ✓boolean     isEmpty();
   Iterator<T> iterator();
}
```

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean      add(T element);
  boolean      remove(T element);
  boolean      contains(T element);
 ✓int          size();
 ✓boolean      isEmpty();
  Iterator<T> iterator();
}
```

We’re taking a systematic approach to developing the ArrayBag class:

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean     add(T element);
  boolean     remove(T element);
  boolean     contains(T element);
 ✓int         size();
 ✓boolean     isEmpty();
  Iterator<T> iterator();
}
```

We’re taking a systematic approach to developing the ArrayBag class:

Develop one method at a time.

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean     add(T element);
  boolean     remove(T element);
  boolean     contains(T element);
 ✓int         size();
 ✓boolean     isEmpty();
  Iterator<T> iterator();
}
```

We’re taking a systematic approach to developing the ArrayBag class:

> Develop one method at a time.
>
> Run it against its full test suite (which will involve calls to other methods that may still be stubs).

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean      add(T element);
  boolean      remove(T element);
  boolean      contains(T element);
 ✓int          size();
 ✓boolean      isEmpty();
  Iterator<T> iterator();
}
```

We’re taking a systematic approach to developing the ArrayBag class:

Develop one method at a time.

Run it against its full test suite (which will involve calls to other methods that may still be stubs).

Analyze its time complexity, revise if appropriate.

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean      add(T element);
  boolean      remove(T element);
  boolean      contains(T element);
 ✓int          size();
 ✓boolean      isEmpty();
  Iterator<T> iterator();
}
```

We're taking a systematic approach to developing the ArrayBag class:

Develop one method at a time.

Run it against its full test suite (which will involve calls to other methods that may still be stubs).

Analyze its time complexity, revise if appropriate.

Consider refactoring, clean-up, and generality.

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean      add(T element);
  boolean      remove(T element);
→ boolean      contains(T element);
 ✓int          size();
 ✓boolean      isEmpty();
  Iterator<T> iterator();
}
```

We’re taking a systematic approach to developing the ArrayBag class:

Develop one method at a time.

Run it against its full test suite (which will involve calls to other methods that may still be stubs).

Analyze its time complexity, revise if appropriate.

Consider refactoring, clean-up, and generality.

ArrayBag – so far

```
import java.util.Iterator;
public interface Bag<T> ... {
 ✓boolean      add(T element);
  boolean      remove(T element);
 ➡boolean      contains(T element);
 ✓int          size();
 ✓boolean      isEmpty();
  Iterator<T> iterator();
}
```

We’re taking a systematic approach to developing the ArrayBag class:

Develop one method at a time.

Run it against its full test suite (which will involve calls to other methods that may still be stubs).

Analyze its time complexity, revise if appropriate.

Consider refactoring, clean-up, and generality.

Note that a given method in this class can’t be fully tested until all the methods have been written. Development and testing are necessarily iterative.

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {







   }





}
```

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {







   }




}
```

This is just linear search.

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < ___________; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }



}
```

This is just linear search.

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < ___________; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }



}
```

Q: What should go in the blank?

A. `elements.length`

B. `size`

C. `isFull()`

D. `DEFAULT_CAPACITY`

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < ___size____; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }



}
```

Q: What should go in the blank?

A. `elements.length`

B. `size` ⬅

C. `isFull()`

D. `DEFAULT_CAPACITY`

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean contains(T element) {

     for (int i = 0; i < ___size_____; i++) {
        if (elements[i].equals(element)) {
           return true;
        }
     }
     return false;

  }


}
```

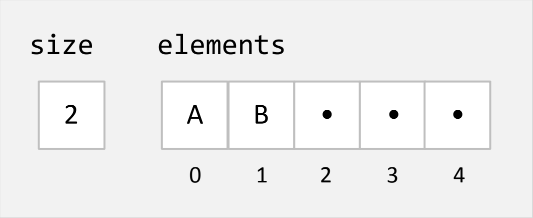
Q: What should go in the blank?

A. `elements.length`

B. `size`

C. `isFull()`

D. `DEFAULT_CAPACITY`



size	elements				
2	A	B	•	•	•
	0	1	2	3	4

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < ___size____; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }



}
```

Q: What should go in the blank?

A. `elements.length`

B. `size`

C. `isFull()`

D. `DEFAULT_CAPACITY`

size	elements				
2	A	B	•	•	•
	0	1	2	3	4

```
size = 2
elements.length = 5
```

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < size; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }




}
```

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean contains(T element) {

      for (int i = 0; i < size; i++) {
         if (elements[i].equals(element)) {
            return true;
         }
      }
      return false;

   }




}
```

Testing …

```
@Test
public void testContainsPresentMiddleFull() {
   BagInterface<String> bag =
   new ArrayBag<String>(5);
   bag.add("A"); bag.add("B");
   bag.add("C"); bag.add("D");
   bag.add("E");
   boolean expected = true;
   boolean actual = bag.contains("C");
   Assert.assertEquals(expected, actual);
}
```

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean contains(T element) {

     for (int i = 0; i < size; i++) {
        if (elements[i].equals(element)) {
           return true;
        }
     }
     return false;

  }



}
```

Testing …

```
@Test
public void testContainsPresentMiddleFull() {
   BagInterface<String> bag =
   new ArrayBag<String>(5);
   bag.add("A"); bag.add("B");
   bag.add("C"); bag.add("D");
   bag.add("E");
   boolean expected = true;
   boolean actual = bag.contains("C");
   Assert.assertEquals(expected, actual);
}
```

Time complexity …

O(N) *where N is the size of the bag, not the capacity of the array*

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {


   }
}
```

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {










   }
}
```

attempt to locate element

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {









   }
}
```

attempt to locate element

Linear search again …

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {









   }
}
```

Linear search from contains:

```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

attempt to locate element

Linear search again …

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }




   }
}
```

Linear search from contains:

```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

attempt to locate element

Linear search again …

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }









   }
}
```

Linear search from contains:

```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

attempt to locate element

Linear search again …

unable to locate

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }


   }
}
```

Linear search from contains:

```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

attempt to locate element

Linear search again …

unable to locate

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
```

located, so remove it

```
   }
}
```

attempt to locate element

Linear search again …

unable to locate

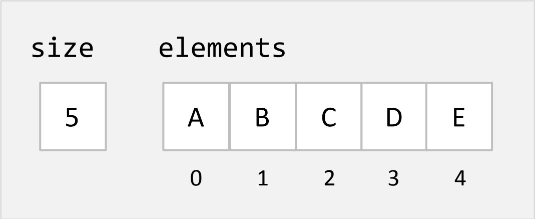
Linear search from contains:

```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }



   }
}
```

located, so remove it



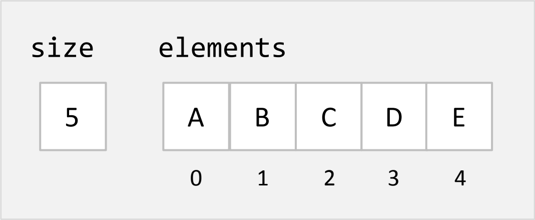
size	elements				
5	A	B	C	D	E
	0	1	2	3	4

```
bag.remove(“B”);
```

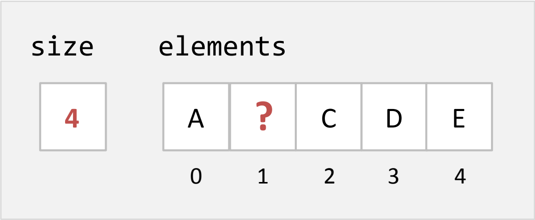
ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

      

   }
}
```

located, so remove it



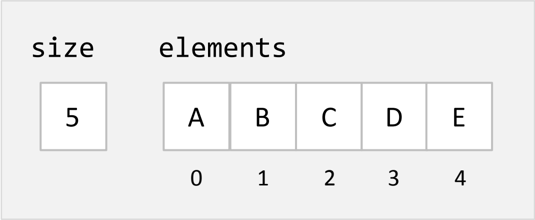
bag.remove(“B”);



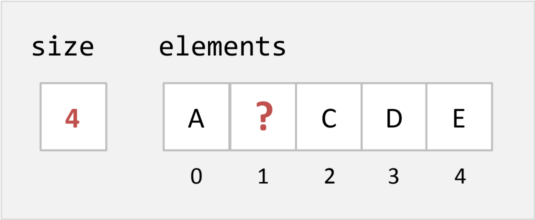
ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }



   }
}
```

located, so remove it



bag.remove("B");

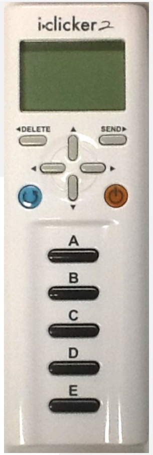


Must handle the array consistent with add() – left justified, no gaps.

Participation

Q: Which is the **correct and most efficient** option for removing element?

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

         located, so remove it

   }
}
```



A. Just set to null

size	elements				
4	A	●	C	D	E
	0	1	2	3	4

B. Shift to the left

size	elements				
4	A	C	D	E	●
	0	1	2	3	4

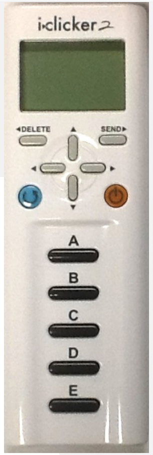
C. Replace with the last

size	elements				
4	A	E	C	D	●
	0	1	2	3	4

Participation

Q: Which is the **correct and most efficient** option for removing element?

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }

            located, so remove it

   }
}
```



A. Just set to null

size	elements				
4	A	●	C	D	E
	0	1	2	3	4

B. Shift to the left

size	elements				
4	A	C	D	E	●
	0	1	2	3	4

C. Replace with the last

size	elements				
4	A	E	C	D	●
	0	1	2	3	4

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }




   }
}
```

located, so remove it

size	elements				
5	A	B	C	D	E
	0	1	2	3	4

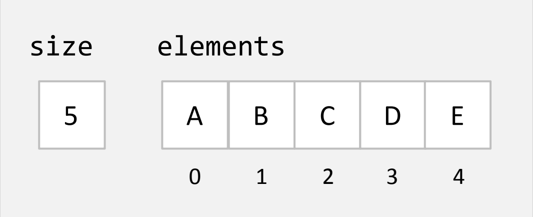
bag.remove(“B”);

size	elements				
4	A	E	C	D	●
	0	1	2	3	4

ArrayBag – remove()

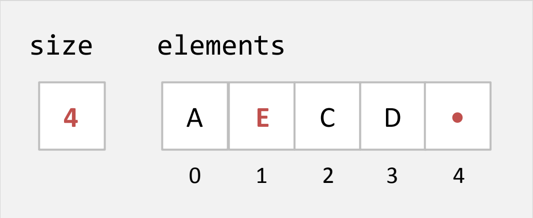
```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

located, so remove it



size	elements				
5	A	B	C	D	E
	0	1	2	3	4

bag.remove(“B”);



size	elements				
4	A	E	C	D	•
	0	1	2	3	4

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Time complexity:

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Time complexity:

O(1)

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Time complexity:

O(N)

O(1)

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Time complexity: **O(N)**

O(N)

O(1)

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Time complexity: O(N)

N = number of elements in the bag, not the capacity of the array

O(N)

O(1)

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Refactoring: Extract method

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Refactoring: Extract method

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean remove(T element) {
      int i = 0;
      while ((i < size) &&
            (!elements[i].equals(element))) {
         i++;
      }
      if (i >= size) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
      return true;
   }
}
```

Refactoring: Extract method

Refactor this for two reasons:
(1) Textbook “extract method” – it’s linear search.
(2) Linear search is used in two different methods – contains and remove.

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

  public boolean remove(T element) {
     int i = 0;
     while ((i < size) &&
           (!elements[i].equals(element))) {
        i++;
     }
     if (i >= size) {
        return false;
     }
     elements[i] = elements[--size];
     elements[size] = null;
     return true;
  }
}
```

Refactoring: Extract method

Refactor this for two reasons:
(1) Textbook "extract method" – it's linear search.
(2) Linear search is used in two different methods – contains and remove.

Note:
The remove() method needs the location of the element, but contains() doesn't. So, remove() can't use the linear search from contains(), but contains() can use the linear search from remove().

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;
   }  return true;
}
```

Refactoring: Extract method

ArrayBag – remove()

```
public class ArrayBag<T> implements Bag<T> {
  public boolean remove(T element) {
     int i = locate(element);
     if (i < 0) {
        return false;
     }
     elements[i] = elements[--size];
     elements[size] = null;
     return true;
  }

  private int locate(T element) {
     for (int i = 0; i < size; i++) {
        if (elements[i].equals(element))
           return i;
     }
     return -1;
  }
}
```

Refactoring: Extract method

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean contains(T element) {


   }



   private int locate(T element) {
      for (int i = 0; i < size; i++) {
         if (elements[i].equals(element))
            return i;
      }
      return -1;
   }
}
```

Refactoring: Extract method

ArrayBag – contains()

```java
public class ArrayBag<T> implements Bag<T> {

   public boolean contains(T element) {


   }


   private int locate(T element) {
      for (int i = 0; i < size; i++) {
         if (elements[i].equals(element))
            return i;
      }
      return -1;
   }
}
```

Refactoring: Extract method

```java
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

ArrayBag – contains()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean contains(T element) {

      return locate(element) >= 0;
   }


   private int locate(T element) {
      for (int i = 0; i < size; i++) {
         if (elements[i].equals(element))
            return i;
      }
      return -1;
   }
}
```

Refactoring: Extract method

```
for (int i = 0; i < size; i++) {
   if (elements[i].equals(element)) {
      return true;
   }
}
return false;
```

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {

   }








}
```

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {

   }




}
```

```
class ArrayIterator<T>
      implements Iterator<T>
```

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {

   }







}
```

```
class ArrayIterator<T>
      implements Iterator<T>
```

Top-level class

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {

   }

}
```

```
class ArrayIterator<T>
      implements Iterator<T>
```

Nested class

Top-level class

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {

   }

}
```

```
class ArrayIterator<T>
      implements Iterator<T>
```

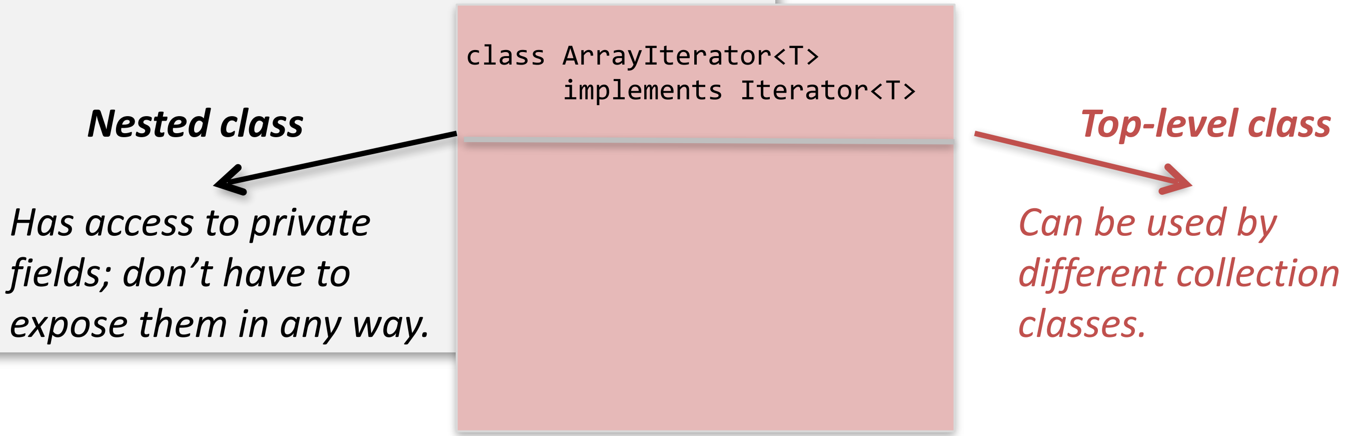
Nested class

Top-level class

Can be used by different collection classes.

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {

   }

}
```



```
class ArrayIterator<T>
      implements Iterator<T>
```

Nested class

Has access to private fields; don’t have to expose them in any way.

Top-level class

Can be used by different collection classes.

ArrayBag – iterator()

```
import java.util.Iterator;
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public Iterator<T> iterator() {
      return new ArrayIterator(elements, size);
   }

}
```

```
class ArrayIterator<T>
      implements Iterator<T>
```

Nested class

Has access to private fields; don't have to expose them in any way.

Top-level class

Can be used by different collection classes.

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  // the array of elements to be iterated over.
  private T[] items;

  // the number of elements in the array.
  private int count;

  // the current position in the iteration.
  private int current;


  public ArrayIterator(T[] elements, int size) {
     items = elements;
     count = size;
     current = 0;
  }
}
```

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public boolean hasNext() {

  }


  public void remove() {

  }

}
```

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public boolean hasNext() {
     return (current < count);
  }


  public void remove() {

  }

}
```

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public boolean hasNext() {
     return (current < count);
  }


  public void remove() {

  }

}
```

The remove method is listed as an “optional operation” in the Iterator API.

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public boolean hasNext() {
     return (current < count);
  }


  public void remove() {
     throw new UnsupportedOperationException();
  }

}
```

The remove method is listed as an “optional operation” in the Iterator API.

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public T next() {





  }


}
```

ArrayBag – iterator()

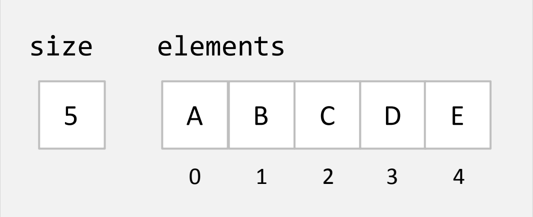
```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public T next() {
     if (!hasNext()) {
        throw new NoSuchElementException();
     }


  }


}
```

ArrayBag – iterator()

```
import java.util.Iterator;
import java.util.NoSuchElementException;
public class ArrayIterator<T> implements Iterator<T> {

  private T[] items;
  private int count;
  private int current;


  public T next() {
     if (!hasNext()) {
        throw new NoSuchElementException();
     }

     return items[current++];
  }


}
```

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {
      if (isFull()) {
         return false;
      }
      elements[size] = element;
      size++;
      return true;
   }



}
```



size

5

elements

0	1	2	3	4
A	B	C	D	E

```
bag.add(“F”);
```

What happens at this point?

Options?

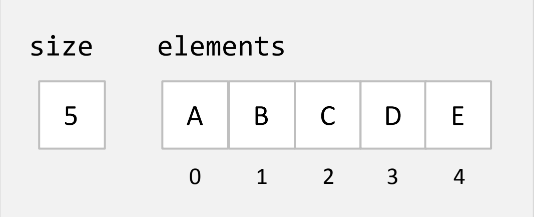
Ignore and return false

Throw an exception

Get a bigger array

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {
   private T[] elements;
   private int size;

   public boolean add(T element) {
      if (isFull()) {
         return false;
      }
      elements[size] = element;
      size++;
      return true;
   }


}
```



size

5

elements

A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

What happens at this point?

Options?

Ignore and return false

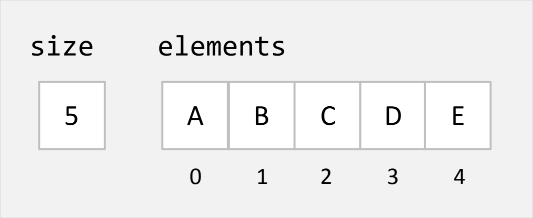
Throw an exception

Get a bigger array

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {

      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Strategy:



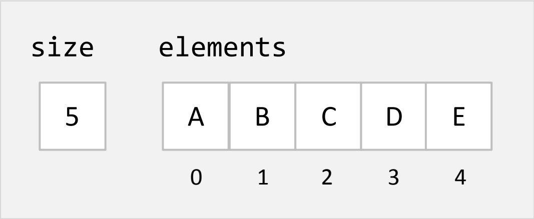
```
bag.add(“F”);
```

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {

      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Strategy:

When the array becomes full, double the capacity.



size

5

elements

A	B	C	D	E
0	1	2	3	4

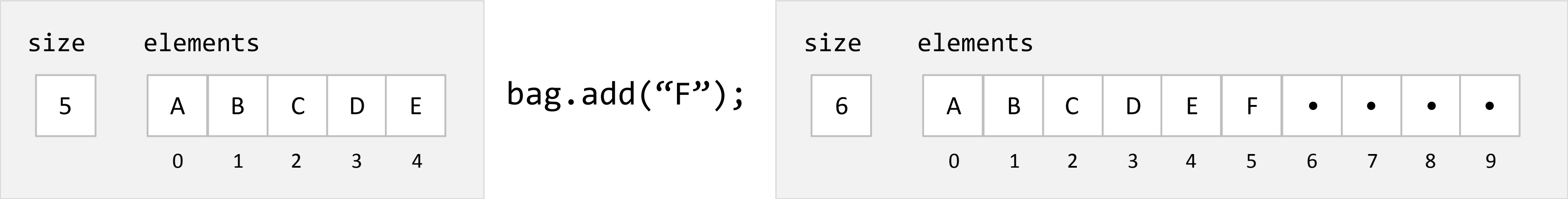
```
bag.add("F");
```

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {

      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Strategy:

When the array becomes full, double the capacity.



size

5

elements

A	B	C	D	E
0	1	2	3	4

`bag.add(“F”);`

size

6

elements

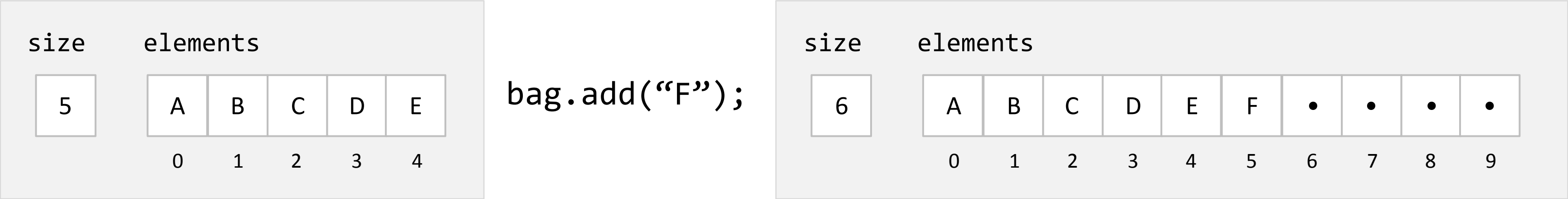
A	B	C	D	E	F	•	•	•	•
0	1	2	3	4	5	6	7	8	9

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Strategy:

When the array becomes full, double the capacity.



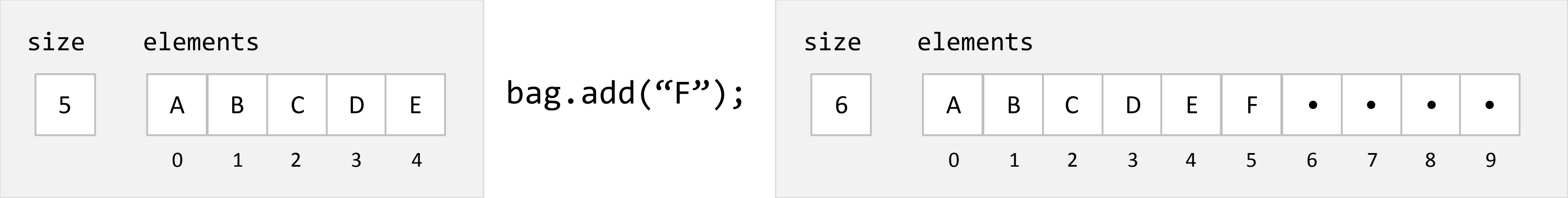
size	elements				
5	A	B	C	D	E
	0	1	2	3	4

bag.add(“F”);

size	elements									
6	A	B	C	D	E	F	•	•	•	•
	0	1	2	3	4	5	6	7	8	9

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   private void resize(int capacity) {
      T[] a = (T[]) new Object[capacity];
      for (int i = 0; i < size(); i++) {
         a[i] = elements[i];
      }
      elements = a;
   }
}
```



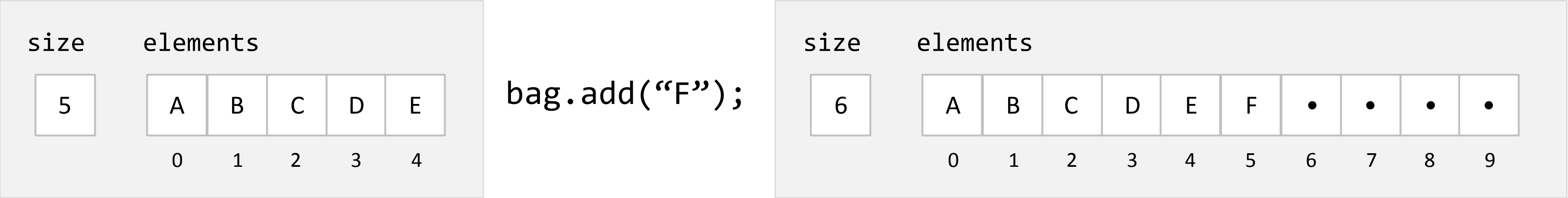
size	elements				
5	A	B	C	D	E
	0	1	2	3	4

bag.add(“F”);

size	elements									
6	A	B	C	D	E	F	•	•	•	•
	0	1	2	3	4	5	6	7	8	9

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   private void resize(int capacity) {
      T[] a = (T[]) new Object[capacity];
      System.arraycopy(elements, 0, a, 0, elements.length);
      elements = a;
   }


}
```



size

5

elements

A	B	C	D	E
0	1	2	3	4

bag.add(“F”);

size

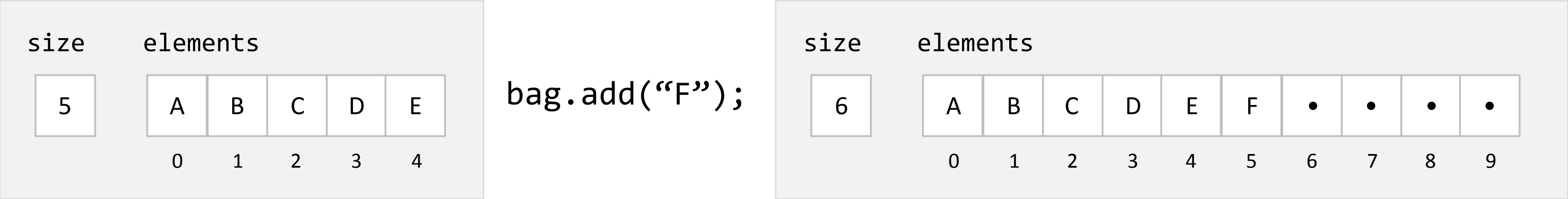
6

elements

A	B	C	D	E	F	•	•	•	•
0	1	2	3	4	5	6	7	8	9

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   private void resize(int capacity) {
      T[] a = Arrays.<T>copyOf(elements, capacity);
      elements = a;
   }



}
```



size

5

elements

A	B	C	D	E
0	1	2	3	4

```
bag.add(“F”);
```

size

6

elements

A	B	C	D	E	F	•	•	•	•
0	1	2	3	4	5	6	7	8	9

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #1: **O(N)**

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #1: **O(N)**

Although we won’t have to expand the array very often, it will be linear cost when we do. So, in a strict sense, the worst case is O(N).

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

add() 1:	1
add() 2:	1
add() 3:	1
add() N:	1
add() N+1:	N

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

add() 1:	1
add() 2:	1
add() 3:	1
add() N:	1
add() N+1:	N

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

add() 1:	1
add() 2:	1
add() 3:	1
add() N:	1
add() N+1:	N

∑ = ~2N

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

$$\left.\begin{array}{ll} \text{add() 1:} & 1 \\ \text{add() 2:} & 1 \\ \text{add() 3:} & 1 \\ \text{add() N:} & 1 \\ \text{add() N+1:} & N \end{array}\right\} \quad \sum = \begin{array}{c} \sim 2N \\ \div \\ \sim N \end{array}$$

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

   public boolean add(T element) {
      if (isFull()) {
         resize(elements.length * 2);
      }
      elements[size] = element;
      size++;
      return true;
   }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

$$\left.\begin{array}{rl} \text{add() 1:} & 1 \\ \text{add() 2:} & 1 \\ \text{add() 3:} & 1 \\ \text{add() N:} & 1 \\ \text{add() N+1:} & N \end{array}\right\} \quad \begin{array}{c} \sum = \sim 2N \\ \div \\ \sim N \\ = \sim 2 \end{array}$$

Dynamic resizing – add()

```
public class ArrayBag<T> implements Bag<T> {

  public boolean add(T element) {
     if (isFull()) {
        resize(elements.length * 2);
     }
     elements[size] = element;
     size++;
     return true;
  }
}
```

Time Complexity:

Answer #2: **O(1) amortized**

We can *amortize* the cost of expanding the capacity of the array over a sequence of N calls to add().

add() 1: 1

add() 2: 1

add() 3: 1

add() N: 1

add() N+1: N

∑ = ~2N

÷

~N

= ~2

O(1)

Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;




      return true;
   }
}
```

Strategy:

Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;



      return true;
   }
}
```

Strategy:

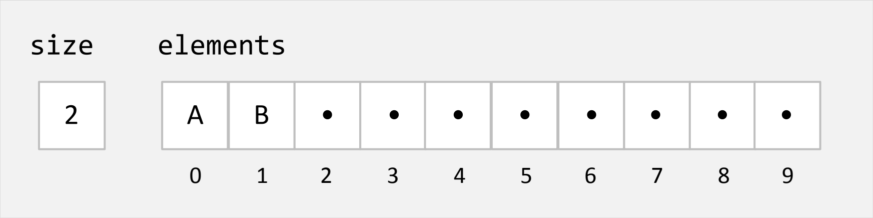
When the array becomes less than 25% full, reduce the capacity by half.

Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;




      return true;
   }
}
```

Strategy:

When the array becomes less than 25% full, reduce the capacity by half.



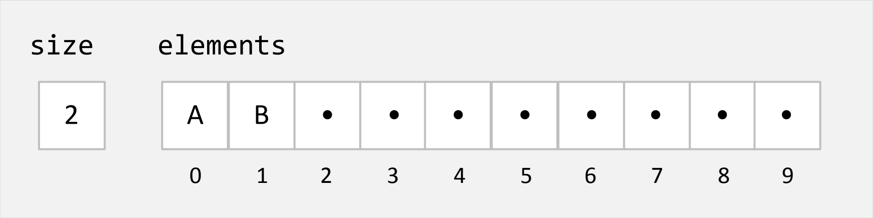
size	elements									
2	A	B	•	•	•	•	•	•	•	•
	0	1	2	3	4	5	6	7	8	9

Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;




      return true;
   }
}
```

Strategy:

When the array becomes less than 25% full, reduce the capacity by half.



size

2

elements

A	B	•	•	•	•	•	•	•	•
0	1	2	3	4	5	6	7	8	9

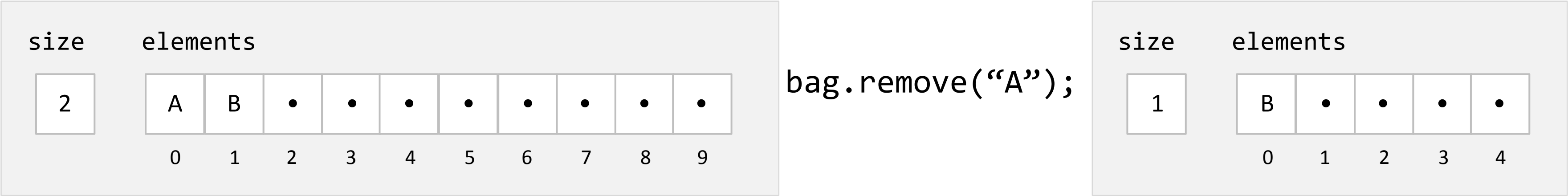
```
bag.remove(“A”);
```

Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;




      return true;
   }
}
```

Strategy:

When the array becomes less than 25% full, reduce the capacity by half.



size

2

elements

A	B	•	•	•	•	•	•	•	•
0	1	2	3	4	5	6	7	8	9

bag.remove(“A”);

size

1

elements

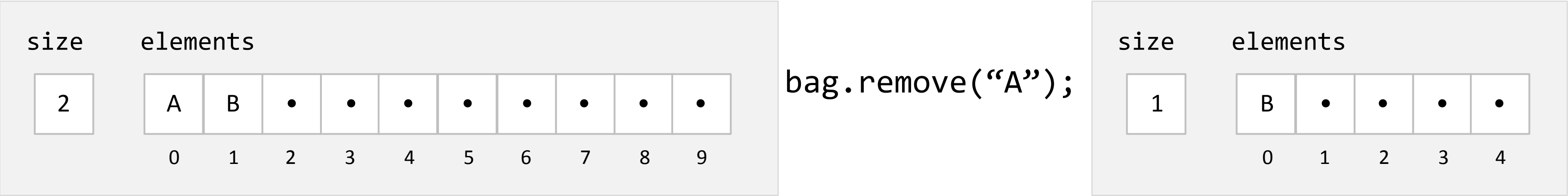
B	•	•	•	•
0	1	2	3	4

Dynamic resizing – remove()

```
public class ArrayBag<T> implements Bag<T> {
   public boolean remove(T element) {
      int i = locate(element);
      if (i < 0) {
         return false;
      }
      elements[i] = elements[--size];
      elements[size] = null;

      if (size > 0 && size < elements.length / 4) {
         resize(elements.length / 2);
      }
      return true;
   }
}
```

Strategy:

When the array becomes less than 25% full, reduce the capacity by half.



size	elements									
2	A	B	•	•	•	•	•	•	•	•
	0	1	2	3	4	5	6	7	8	9

bag.remove(“A”);

size	elements				
1	B	•	•	•	•
	0	1	2	3	4

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {


   private T[] elements;
   private int size;

   public ArrayBag() {
      this(DEFAULT_CAPACITY);
   }
}
```

```
Bag bag = new ArrayBag();
```

ArrayBag – constructor

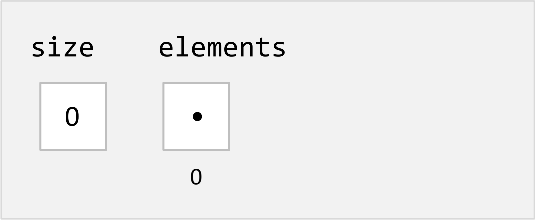
```
public class ArrayBag<T> implements Bag<T> {
    private static final int DEFAULT_CAPACITY = 1;

    private T[] elements;
    private int size;

    public ArrayBag() {

        this(DEFAULT_CAPACITY);

    }

}
```

```
Bag bag = new ArrayBag();
```

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
  private static final int DEFAULT_CAPACITY = 1;
  private T[] elements;
  private int size;

  public ArrayBag() {
    this(DEFAULT_CAPACITY);
  }
}
```

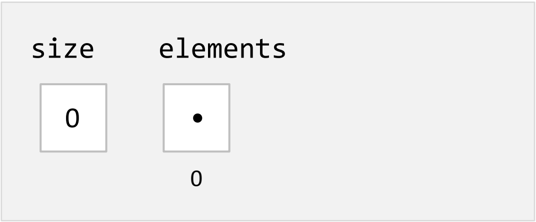
```
Bag bag = new ArrayBag();
```



ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 1;
   private T[] elements;
   private int size;

   public ArrayBag() {
      this(DEFAULT_CAPACITY);
   }
}
```

```
Bag bag = new ArrayBag();
```

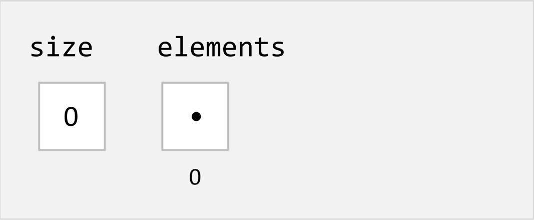


Starting an empty bag at capacity 1 and using the dynamic resizing strategies just described allows us to maintain the following invariant: the array is always between 25% and 100% full.

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 1;
   private T[] elements;
   private int size;

   public ArrayBag() {
      this(DEFAULT_CAPACITY);
   }
}
```

```
Bag bag = new ArrayBag();
```



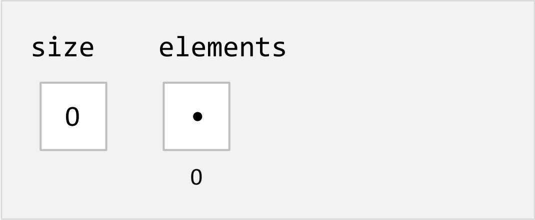
Starting an empty bag at capacity 1 and using the dynamic resizing strategies just described allows us to maintain the following invariant: the array is always between 25% and 100% full.

Thus, the amount of memory needed for the array is a constant times N, that is, O(N).

ArrayBag – constructor

```
public class ArrayBag<T> implements Bag<T> {
   private static final int DEFAULT_CAPACITY = 1;
   private T[] elements;
   private int size;

   public ArrayBag() {
      this(DEFAULT_CAPACITY);
   }
}
```

```
Bag bag = new ArrayBag();
```



Starting an empty bag at capacity 1 and using the dynamic resizing strategies just described allows us to maintain the following invariant: the array is always between 25% and 100% full.

Thus, the amount of memory needed for the array is a constant times N, that is, O(N).

We can guarantee that our implementation only needs a linear amount of memory.

Participation

Q: Assuming that the ArrayBag class implements the dynamic resizing strategy just described, what is the capacity of the internal array after the following sequence of statements has executed?

```
Bag<String> sb = new ArrayBag<String>();

sb.add("A"); sb.add("B"); sb.add("C"); sb.add("D"); sb.add("E");

sb.remove("A"); sb.remove("B"); sb.remove("C"); sb.remove("D");
```

A. 10
B. 8
C. 4
D. 2

Participation

Q: Assuming that the ArrayBag class implements the dynamic resizing strategy just described, what is the capacity of the internal array after the following sequence of statements has executed?

```
Bag<String> sb = new ArrayBag<String>();

sb.add("A"); sb.add("B"); sb.add("C"); sb.add("D"); sb.add("E");

sb.remove("A"); sb.remove("B"); sb.remove("C"); sb.remove("D");
```

A. 10
B. 8
C. 4 ←
D. 2

Things to think about

Things to think about

- **Consider different design/implementation choices.**

Things to think about

- **Consider different design/implementation choices.**
    - The current implementation optimized the add() method.

Things to think about

- **Consider different design/implementation choices.**
  - The current implementation optimized the add() method.
  - What if the ArrayBag was intended to be used in an application where the data is fairly stable but there will be a high volume of queries (calls to contain).

Things to think about

- **Consider different design/implementation choices.**
  - The current implementation optimized the add() method.
  - What if the ArrayBag was intended to be used in an application where the data is fairly stable but there will be a high volume of queries (calls to contain).
    - Optimize contains() instead => impose order on array => change add() and remove() => add() will become O(N) and contains will become O(log N).

Things to think about

- **Consider different design/implementation choices.**
  - The current implementation optimized the add() method.
  - What if the ArrayBag was intended to be used in an application where the data is fairly stable but there will be a high volume of queries (calls to contain).
    - Optimize contains() instead => impose order on array => change add() and remove() => add() will become O(N) and contains will become O(log N).
    - Tradeoffs like these are important to be able to describe, measure, and make informed choices.

Things to think about

- **Consider different design/implementation choices.**
  - The current implementation optimized the add() method.
  - What if the ArrayBag was intended to be used in an application where the data is fairly stable but there will be a high volume of queries (calls to contain).
    - Optimize contains() instead => impose order on array => change add() and remove() => add() will become O(N) and contains will become O(log N).
    - Tradeoffs like these are important to be able to describe, measure, and make informed choices.

- **What would change if we were implementing a Set collection instead of a Bag?**

Things to think about

- **Consider different design/implementation choices.**
  - The current implementation optimized the add() method.
  - What if the ArrayBag was intended to be used in an application where the data is fairly stable but there will be a high volume of queries (calls to contain).
    - Optimize contains() instead => impose order on array => change add() and remove() => add() will become O(N) and contains will become O(log N).
    - Tradeoffs like these are important to be able to describe, measure, and make informed choices.

- **What would change if we were implementing a Set collection instead of a Bag?**
  - add() must change to eliminate duplicates => must use linear search => increases to O(N) time.

Things to think about

- **Consider different design/implementation choices.**
  - The current implementation optimized the add() method.
  - What if the ArrayBag was intended to be used in an application where the data is fairly stable but there will be a high volume of queries (calls to contain).
    - Optimize contains() instead => impose order on array => change add() and remove() => add() will become O(N) and contains will become O(log N).
    - Tradeoffs like these are important to be able to describe, measure, and make informed choices.

- **What would change if we were implementing a Set collection instead of a Bag?**
  - add() must change to eliminate duplicates => must use linear search => increases to O(N) time.
  - Since add() must change anyway, should we impose an order?