Lecture 06:Bag Implementations That Use Arrays

CS 0445: Data Structures

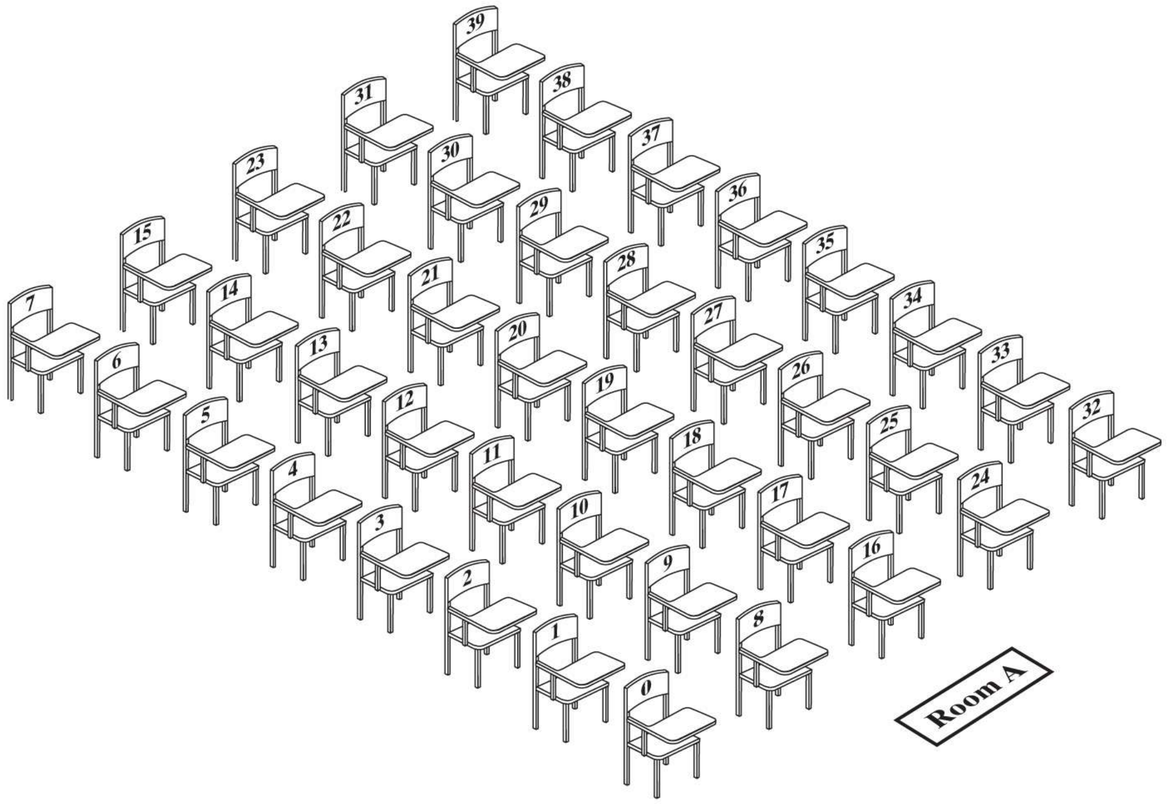
Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Sep 12, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

Fixed-Size Array to Implement the ADT Bag

classroom that contains desks in fixed positions



UML for a fixed size `ArrayBag`

```
ArrayBag
--------------------------------------
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
--------------------------------------
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T):
integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

The Class `ArrayBag` (Part 1)

```
/**A class of bags whose entries are stored in a fixed-size array.
   INITIAL, INCOMPLETE DEFINITION; no security checks */
public final class ArrayBag<T> implements BagInterface<T>
{
       private final T[] bag;
       private int numberOfEntries;
       private static final int DEFAULT_CAPACITY = 25;

       /** Creates an empty bag whose initial capacity is 25. */
       public ArrayBag()
       {
       this(DEFAULT_CAPACITY);
       } // end default constructor

       /** Creates an empty bag having a given initial capacity.
      @param desiredCapacity  The integer capacity desired. */
       public ArrayBag(int desiredCapacity)
       {
     // The cast is safe because the new array contains null entries.
     @SuppressWarnings("unchecked")
     T[] tempBag = (T[])new Object[desiredCapacity]; // Unchecked cast
     bag = tempBag;
     numberOfEntries = 0;
       } // end constructor
```

The Class `ArrayBag` (Part 2)

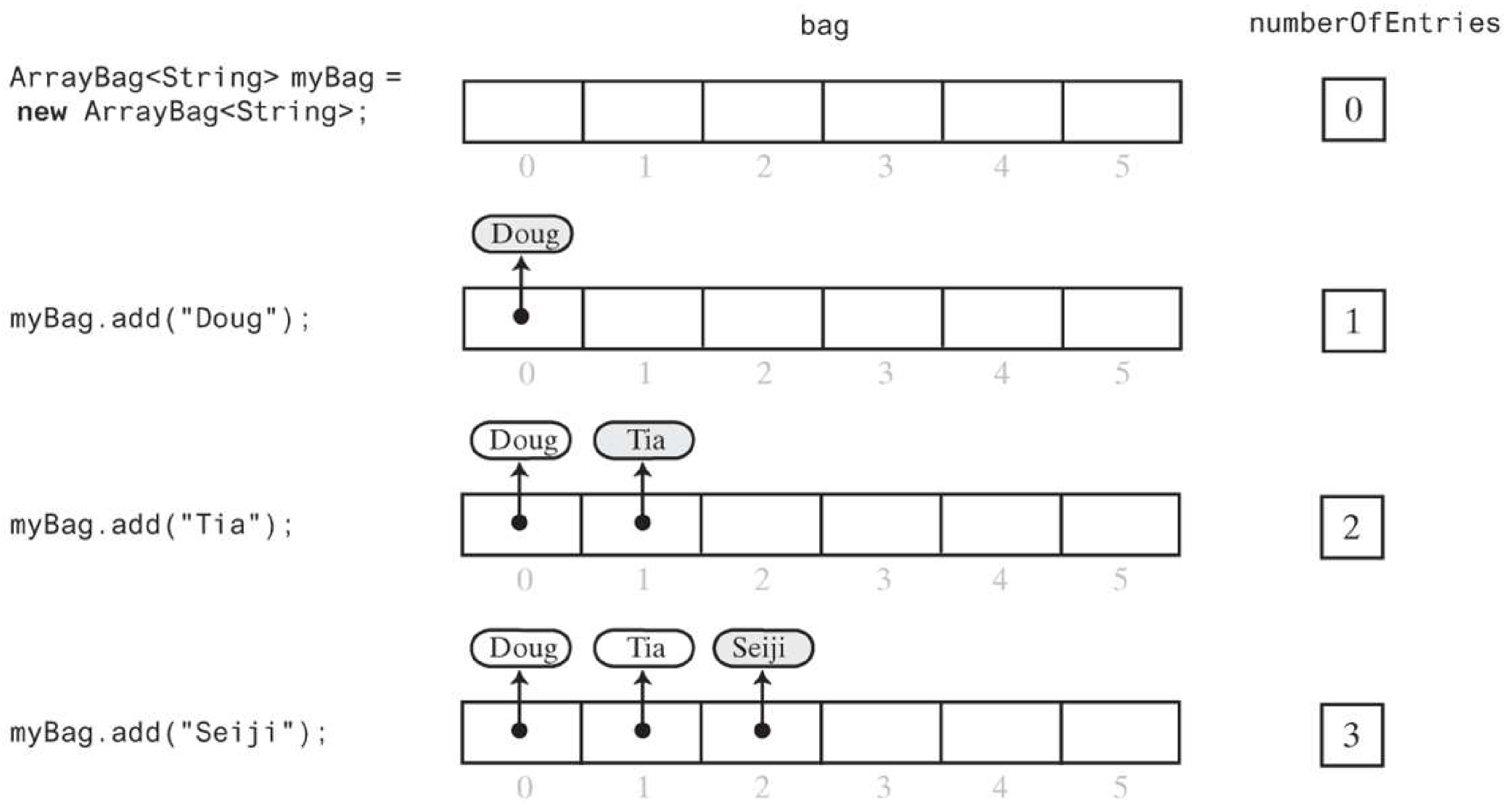
```
   /** Adds a new entry to this bag.
  @param newEntry  The object to be added as a new entry.
  @return  True if the addition is successful, or false if not. */
   public boolean add(T newEntry)
   {
   // To be defined
   } // end add

   /** Retrieves all entries that are in this bag.
  @return  A newly allocated array of all the entries in this bag. */
   public T[] toArray()
   {
   // To be defined
   } // end toArray

 // Returns true if the array bag is full, or false if not.
   private boolean isArrayFull()
   {
   // To be defined
   } // end isArrayFull
} // end ArrayBag
```

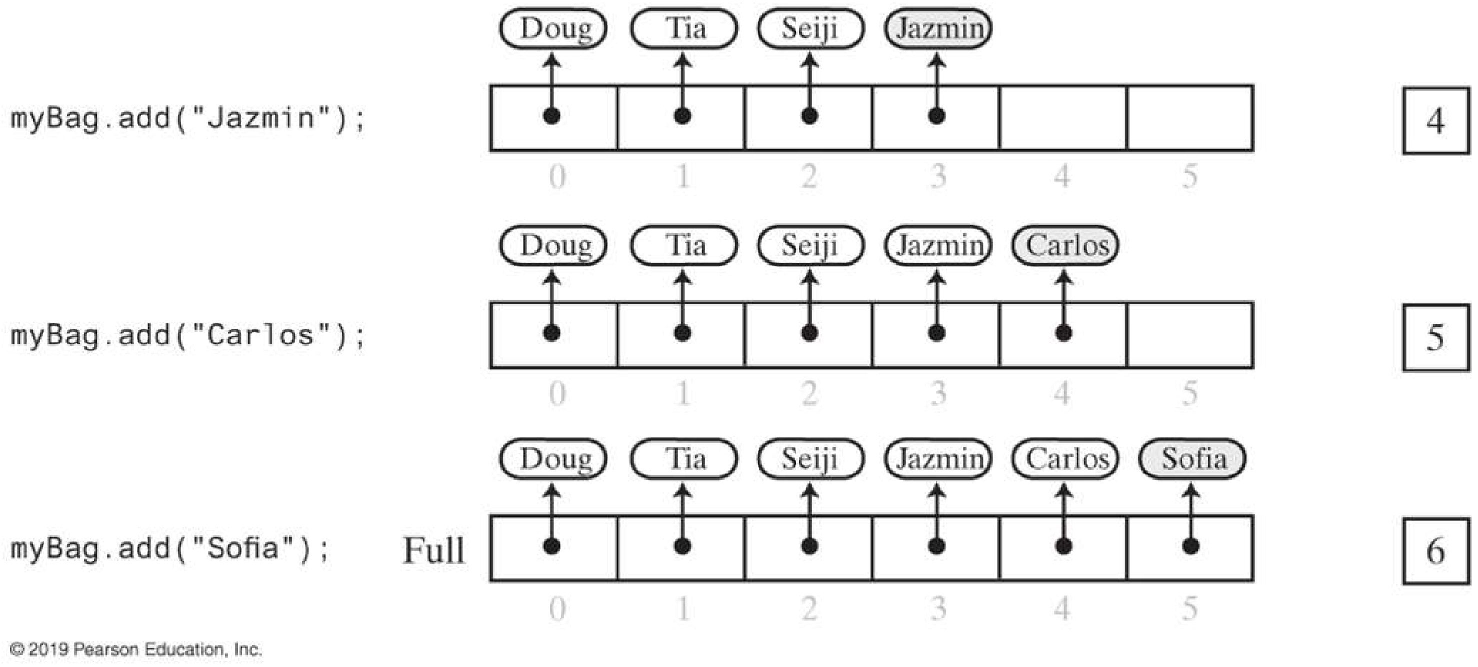
Adding to a fixed-size `ArrayBag` (Part 1)

Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



Adding to a fixed-size `ArrayBag` (Part 2)

Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



Fixed-Size `ArrayBag`

```
/** Adds a new entry to this bag.
   @param newEntry  the object to be added as a new entry.
   @return  True if the addition is successful, or false if not.*/
public boolean add(T newEntry)
{
boolean result = true;
if (isArrayFull())
{
      result = false;
}
else
{  // Assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
} // end if

return result;
} // end add
```

Fixed-Size `ArrayBag`

```
// Returns true if this bag is full, or false if not.
	private boolean isArrayFull()
	{
	return numberOfEntries >= bag.length;
	} // end isArrayFull
```

Fixed-Size `ArrayBag`

```
/** Retrieves all entries that are in this bag.
      @return  A newly allocated array of all
                                   the entries in the bag. */
     public T[] toArray()
     {
    // The cast is safe because the new array
                                  //  contains null entries.
    @SuppressWarnings("unchecked")
    T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast

     for (int index = 0; index < numberOfEntries; index++)
     {
          result[index] = bag[index];
     } // end for

     return result;
     } // end toArray
```

Making the Implementation Secure

- Practice fail-safe programming by including checks for anticipated errors
- Validate input data and arguments to a method
- refine incomplete implementation of ArrayBag to make code more secure by adding the following two data fields

```
private boolean integrityOK = false;
private static final int MAX_CAPACITY = 10000;
```

Making the Implementation Secure

- **Revised constructor**

```
 /** Creates an empty bag having a given capacity.
@param desiredCapacity  The integer capacity desired. */
 public ArrayBag2(int desiredCapacity)
 {
if (desiredCapacity <= MAX_CAPACITY)
{
  // The cast is safe because the new array contains null entries
  @SuppressWarnings("unchecked")
  T[] tempBag = (T[])new Object[desiredCapacity]; // Unchecked cast
  bag = tempBag;
  numberOfEntries = 0;
  integrityOK = true;
}
else
  throw new IllegalStateException("Attempt to create a bag whose" +
                                "capacity exceeds allowed maximum.");
 } // end constructor
```

Making the Implementation Secure

- **Method to check initialization**

```
// Throws an exception if this object is not initialized.
private void checkIntegrity()
{
  if (!integrityOK)
    throw new SecurityException("ArrayBag object is corrupt.");
} // end checkIntegrity
```

Making the Implementation Secure

- Revised method `add`

```
 /** Adds a new entry to this bag.
 @param newEntry  The object to be added as a new entry.
 @return  True if the addition is successful, or false if not. */
 public boolean add(T newEntry)
 {
 checkIntegrity();
boolean result = true;
if (isArrayFull())
{
  result = false;
}
else
{  // Assertion: result is true here
  bag[numberOfEntries] = newEntry;
  numberOfEntries++;
} // end if

return result;
 } // end add
```

Testing the Core Methods

- **Stubs for `remove` and `clear`**

```
    /** Removes one unspecified entry from this bag, if possible.
   @return  Either the removed entry, if the removal
           was successful, or null */
    public T remove()
{
   return null; // STUB
} // end remove

    /** Removes one occurrence of a given entry from this bag.
   @param anEntry  The entry to be removed
   @return  True if the removal was successful, or false otherwise */
public boolean remove(T anEntry)
{
   return false; // STUB
} // end remove

    /** Removes all entries from this bag. */
    public void clear()
{
   // STUB
} // end clear
```

Testing the Core Methods (Part 1)

A program that tests core methods of the class `ArrayBag`

```
/**  A test of the constructors and the methods add and toArray,
   as defined in the first draft of the class ArrayBag. */
public class ArrayBagDemo1
{
      public static void main(String[] args)
      {
     // Adding to an initially empty bag with sufficient capacity
     System.out.println("Testing an initially empty bag with " +
                    " sufficient capacity:");
      BagInterface<String> aBag = new ArrayBag1<>();
      String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
      testAdd(aBag, contentsOfBag1);

     // Filling an initially empty bag to capacity
     System.out.println("\nTesting an initially empty bag that " +
                    " will be filled to capacity:");
      aBag = new ArrayBag1<>(7);
      String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D",
                          "another string"};
      testAdd(aBag, contentsOfBag2);
      } // end main
```

Testing the Core Methods (Part 2)

A program that tests core methods of the class `ArrayBag`

```
// Tests the method add.
    private static void testAdd(BagInterface<String> aBag, String[] content)
    {
    System.out.print("Adding the following strings to the bag: ");
    for (int index = 0; index < content.length; index++)
    {
          if (aBag.add(content[index]))
      System.out.print(content[index] + " ");
    else
      System.out.print("\nUnable to add " + content[index] +
                  " to the bag.");
    } // end for
  System.out.println();

    displayBag(aBag);
    } // end testAdd
```

Testing the Core Methods (Part 3)

- **A program that tests core methods of the class `ArrayBag`**

```
// Tests the method toArray while displaying the bag.
    private static void displayBag(BagInterface<String> aBag)
    {
    System.out.println("The bag contains the following string(s):");
    Object[] bagArray = aBag.toArray();
    for (int index = 0; index < bagArray.length; index++)
    {
        System.out.print(bagArray[index] + " ");
    } // end for

    System.out.println();
    } // end displayBag
} // end ArrayBagDemo1
```

Program Output

```
Testing an initially empty bag with  sufficient capacity:
Adding the following strings to the bag: A A B A C A
The bag contains the following string(s):
A A B A C A

Testing an initially empty bag that  will be filled to capacity:
Adding the following strings to the bag: A B A C B C D
Unable to add another string to the bag.
The bag contains the following string(s):
A B A C B C D
```

Implementing More Methods

- Methods `isEmpty` and `getCurrentSize`

```
/** Sees whether this bag is empty.
 @return  True if this bag is empty, or false if not. */
public boolean isEmpty()
{
return numberOfEntries == 0;
} // end isEmpty

/** Gets the current number of entries in this bag.
 @return  The integer number of entries currently in this bag. */
public int getCurrentSize()
{
return numberOfEntries;
} // end getCurrentSize
```

Implementing More Methods

- **Method `getFrequencyOf`**

```
/** Counts the number of times a given entry appears in this bag.
@param anEntry  The entry to be counted.
@return  The number of times anEntry appears in this bag. */
public int getFrequencyOf(T anEntry)
{
checkIntegrity();
int counter = 0;

for (int index = 0; index < numberOfEntries; index++)
{
  if (anEntry.equals(bag[index]))
  {
    counter++;
  } // end if
} // end for
```

Implementing More Methods

- **Method `contains`**

```
  /** Tests whether this bag contains a given entry.
  @param anEntry  The entry to locate.
  @return  True if this bag contains anEntry, or false otherwise. */
public boolean contains(T anEntry)
  {
  checkIntegrity();
  boolean found = false;
  int index = 0;
  while (!found && (index < numberOfEntries))
  {
       if (anEntry.equals(bag[index]))
       {
            found = true;
       } // end if
   index++;
  } // end while
  return found;
} // end contains
```

Methods That Remove Entries

- The method `clear`

```
/** Removes all entries from this bag. */
public void clear()
{
while (!isEmpty())
  remove();
} // end clear
```

Methods That Remove Entries

The method `remove`

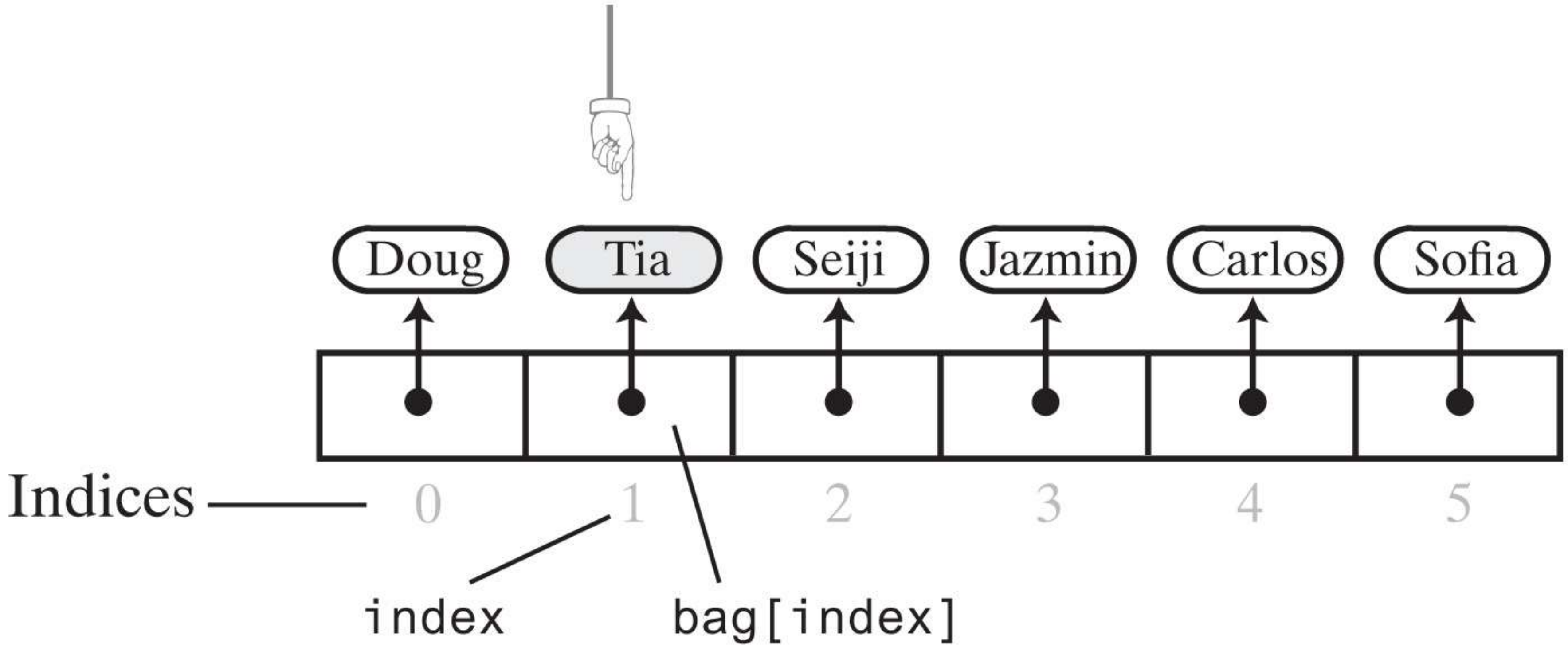
```
/** Removes one unspecified entry from this bag, if possible.
 @return  Either the removed entry, if the removal
        was successful, or null. */
public T remove()
{
checkIntegrity();
T result = null;

if (numberOfEntries > 0))
{
  result = bag[numberOfEntries - 1];
  bag[numberOfEntries - 1] = null;
  numberOfEntries--;
} // end if

return result;
} // end remove
```

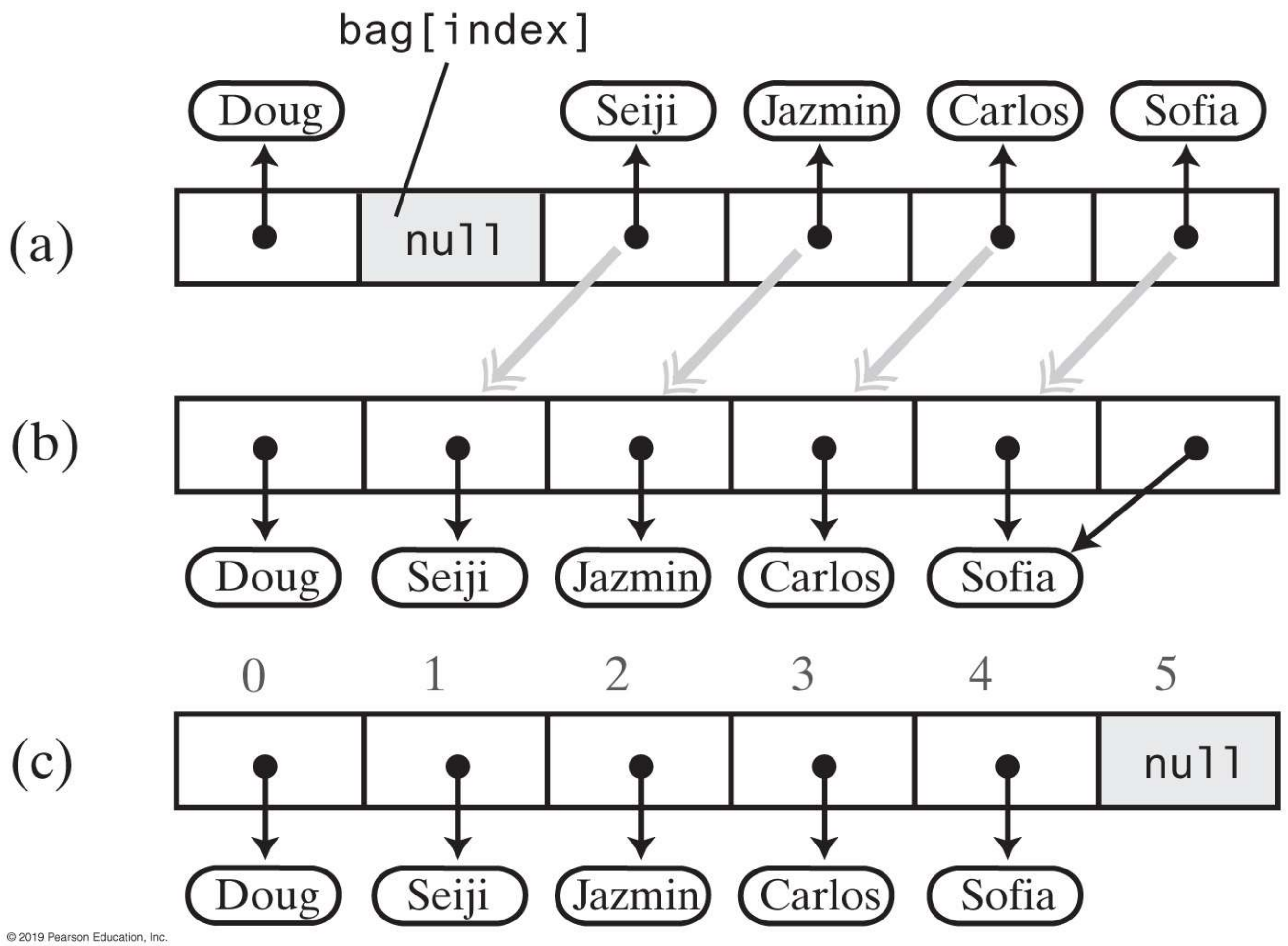
Methods That Remove Entries

The array bag after a successful search for the string "Tia"



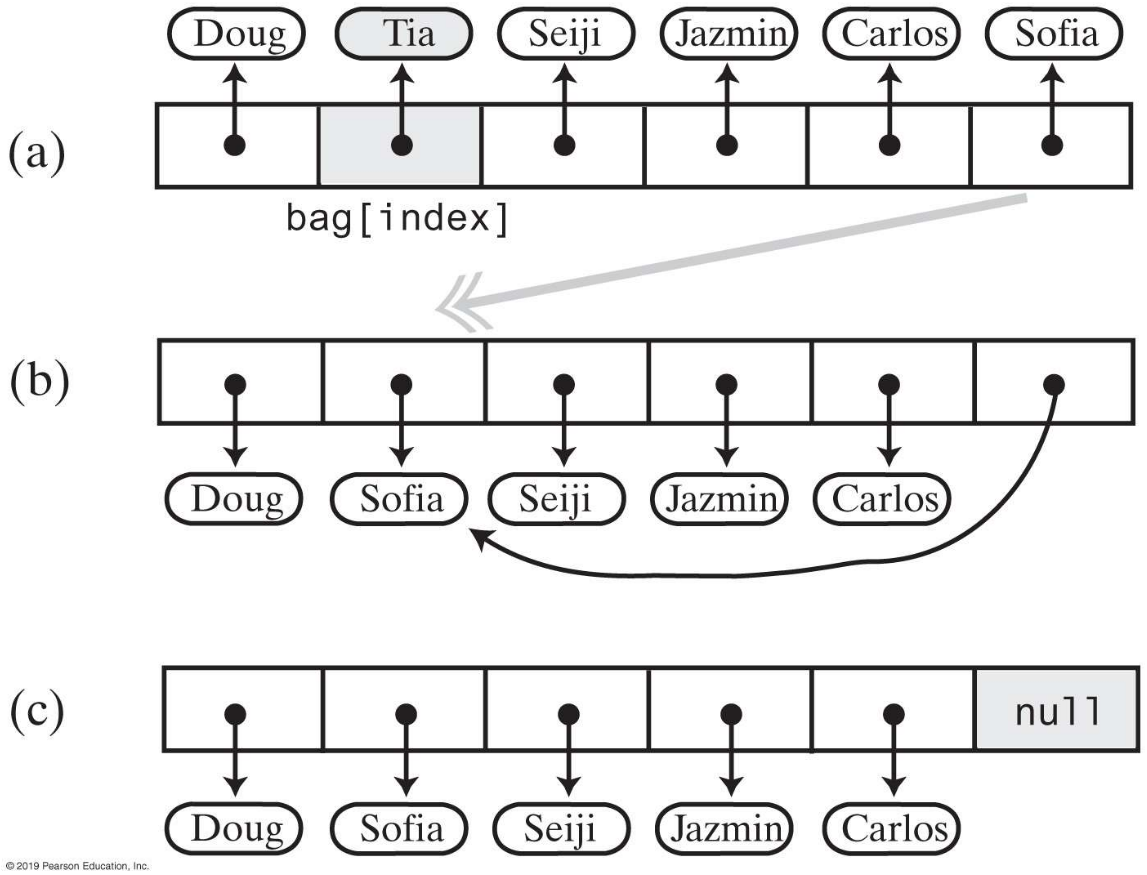
Methods That Remove Entries

Shifting entries to avoid a gap when removing an entry



Methods That Remove Entries

Avoiding a gap in the array while removing an entry



Methods That Remove Entries

- The private helper method `removeEntry`

```
// Removes and returns the entry at a given index within the array bag.
// If no such entry exists, returns null.
    // Preconditions: 0 <= givenIndex < numberOfEntries;
//              checkIntegrity has been called.
private T removeEntry(int givenIndex)
    {
    T result = null;

    if (!isEmpty() && (givenIndex >= 0))
    {
     result = bag[givenIndex];                // Entry to remove
     bag[givenIndex] = bag[numberOfEntries - 1]; // Replace entry with last entry
     bag[numberOfEntries - 1] = null;          // Remove last entry
     numberOfEntries--;
    } // end if

   return result;
    } // end removeEntry
```

Methods That Remove Entries

- **The revised `remove` methods**

```
/** Removes one unspecified entry from this bag, if possible.
@return  Either the removed entry, if the removal was successful,
      or null otherwise. */
public T remove()
{
checkIntegrity();
T result = removeEntry(numberOfEntries - 1);
return result;
} // end remove

/** Removes one occurrence of a given entry from this bag.
@param anEntry  The entry to be removed.
@return  True if the removal was successful, or false if not. */
public boolean remove(T anEntry)
{
checkIntegrity();
int index = getIndexOf(anEntry);
T result = removeEntry(index);
return anEntry.equals(result);
} // end remove
```

Methods That Remove Entries

- **Definition for the method `getIndexOf`**

```
    // Locates a given entry within the array bag.
    // Returns the index of the entry, if located, or -1 otherwise.
    // Precondition: checkIntegrity has been called.
private int getIndexOf(T anEntry){
int where = -1;
 boolean found = false;
 int index = 0;

  while (!found && (index < numberOfEntries)){
    if (anEntry.equals(bag[index]))
    {
      found = true;
      where = index;
    } // end if
    index++;
  } // end while

  // Assertion: If where > -1, anEntry is in the array bag, and it
  // equals bag[where]; otherwise, anEntry is not in the array

   return where;
   } // end getIndexOf
```

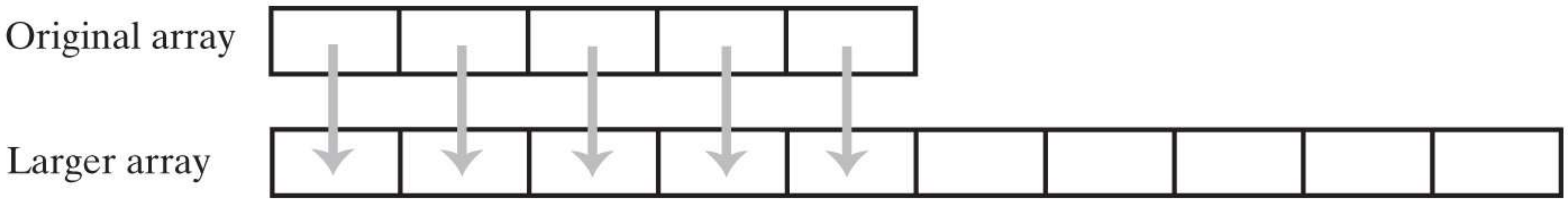
Revised Methods

- **Revised definition for the method `contains`**

```
public boolean contains(T anEntry)
{
   checkIntegrity();
   return getIndexOf(anEntry) > -1; // or >= 0
} // end contains
```

Resizing an Array

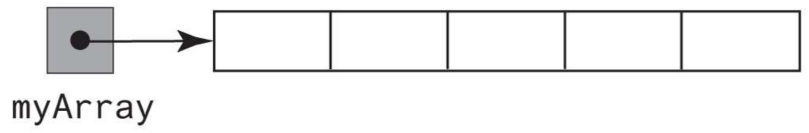
Resizing an array copies its contents to a larger second array



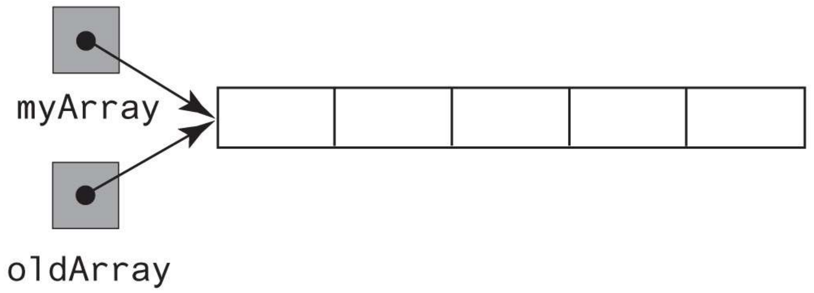
Steps to Resize an Array (Part 1)

Resizing an array

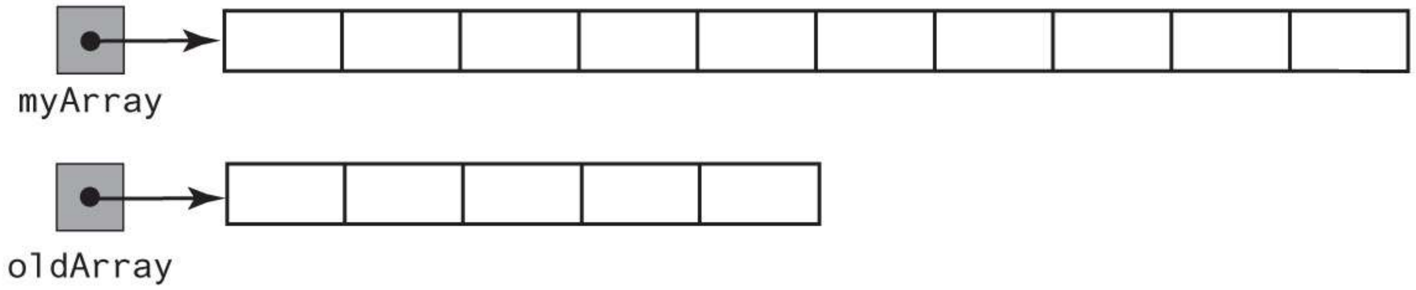
(a) An array



(b) Two references to the same array



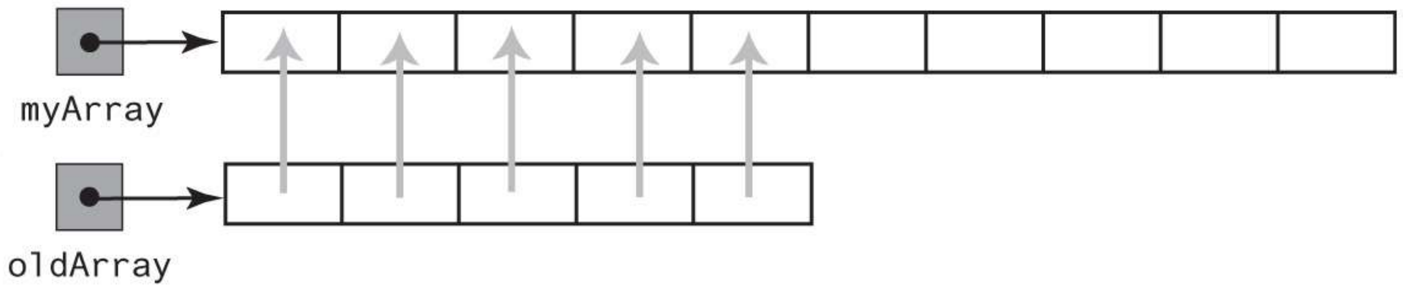
(c) The original array variable now references a new, larger array



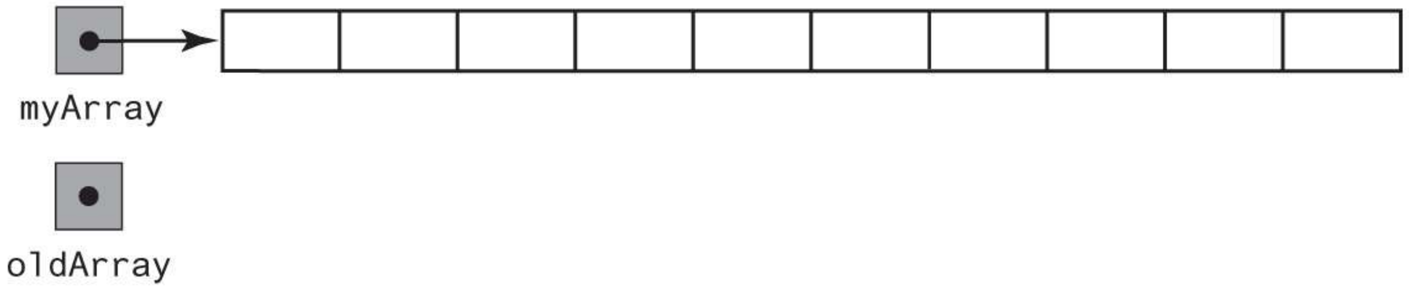
Steps to Resize an Array (Part 2)

Resizing an array

(d) The entries in the original array are copied to the new array

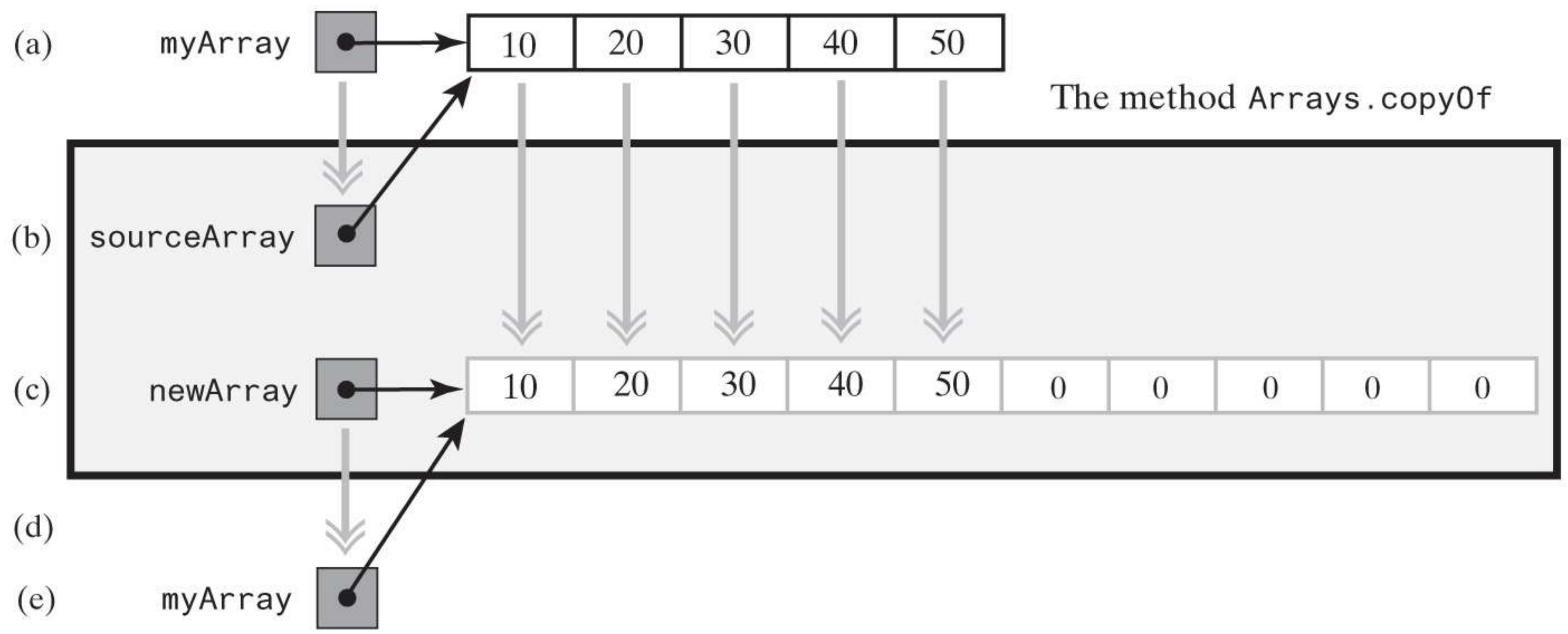


(e) The original array is discarded



Resizing Using `Arrays.copyOf`

Alternative steps to resize an array



New Implementation of a Bag

- **Revised definition of method `add`**

```
/** Adds a new entry to this bag.
   @param newEntry  The object to be added as a new entry.
   @return  True.  */
public boolean add(T newEntry)
{
  checkIntegrity();
  boolean result = true;
  if (isArrayFull())
  {
    doubleCapacity();
  } // end if

  bag[numberOfEntries] = newEntry;
  numberOfEntries++;

  return true;
} // end add
```

New Implementation of a Bag

- The methods `checkCapacity` and `doubleCapacity`

```
// Throws an exception if the client requests a capacity that is too large.
private void checkCapacity(int capacity)
{
  if (capacity > MAX_CAPACITY)
    throw new IllegalStateException("Attempt to create a bag whose " +
                                    "capacity exeeds allowed " +
                                    "maximum of " + MAX_CAPACITY);
} // end checkCapacity

// Doubles the size of the array bag.
// Precondition: checkIntegrity has been called.
private void doubleCapacity()
{
  int newLength = 2 * bag.length;
  checkCapacity(newLength);
  bag = Arrays.copyOf(bag, newLength);
} // end doubleCapacity
```

Pros and Cons of Using an Array

- Adding an entry to the bag is fast
- Removing an unspecified entry is fast
- Removing a particular entry requires time to locate the entry
- Increasing the size of the array requires time to copy its entries