COSC 2006: Data Structures I

The IntArrayBag Collection Class

From Main's book

The IntArraySet Collection Class

(not in Main's book)

The DoubleArraySeq Collection Class

(Programming Project in Main's book)

COSC 2006: Data Structures I

The IntArrayBag Collection Class

From Main's book

The IntArraySet Collection Class

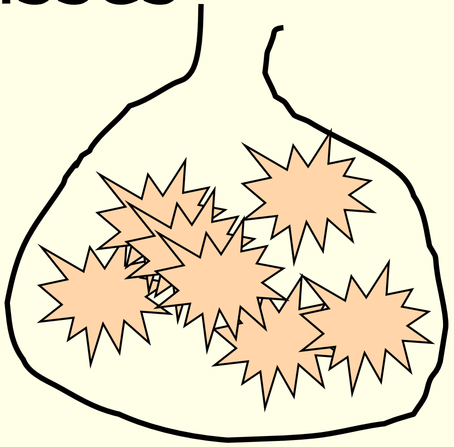
(not in Main's book)

The DoubleArraySeq Collection Class

(Programming Project in Main's book)

The Bag and Set ADTs

- They are container (collection) classes
- Set data type (Main doesn't do it)
  - no duplicates, order unimportant
- Bag data type
  - (multi-set): duplicates are allowed
- In both cases the order is unimportant



Main does an IntArrayBag class which is a Bag of integers implemented using a dynamic array

IntArrayBag ADT

Design and Documentation

IntArrayBag class design

```
public class IntArrayBag implements Cloneable
{
   public IntArrayBag() {...}
   public IntArrayBag(int initialCapacity) {...}

   public void add(int element) {...}
   public void addMany(int... elements) {...} // Java 5
   public void addAll(IntArrayBag addend) {...}
   public IntArrayBag clone() {...} // Java 5
   public int countOccurrences(int target) {...}
   public void ensureCapacity(int minimumCapacity) {...}
   public int getCapacity() {...}
   public boolean remove(int target) {...}
   public int size() {...}
   public static IntArrayBag union(
      IntArrayBag b1, IntArrayBag b2) {...}
   public void trimToSize() {...}
   public String toString() {...} // we added this
}
```

Constructors

- **`public IntArrayBag()`**
  - initialize an empty bag with an initial capacity of 10 integers
- **`public intArrayBag(`**
  `int initialCapacity)`
  - initialize an empty bag with the specified initial capacity. The add method works efficiently without needing more memory until this capacity is reached.

Methods (1)

- `public void add(int element)`
  - add a new element to this bag. If the current capacity would be exceeded then the capacity is increased before adding the new element

-
  ```
  public void addMany(
                  int ... elements)
  ```
  - Add the specified elements to this bag. This is a feature of Java 5 (variable argument list)

Methods (2)

- `public void addAll(`
  `IntArrayBag addend)`
  - Add the contents of the specified bag to this bag. The addend bag is unchanged.

Methods (3)

- **`public IntArrayBag clone()`**
  - return a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. In Java 5 the return type can be specified instead of using the **`Object`** type
- **`public int countOccurrences(`**
  `int target)`
  - Count the number of occurrences of a given element in this bag

Methods (4)

- `public void ensureCapacity(`
  `                    int minimumCapacity)`
  - Change the current capacity of this bag to at least minimumCapacity. If the capacity was already at or greater than minimumCapacity, then the capacity is left unchanged.
- `public int getCapacity()`
  - return the current capacity of this bag

Methods (5)

- `public boolean remove(`
  `                                int target)`
  - Remove one copy of the specified element from this bag. Return true if element was found else return false
- `public int size()`
  - return the current number of elements in this bag

Methods (6)

- **`public void trimToSize()`**
  - Reduce the current capacity of this bag to its actual size (number of elements it currently contains)
- **`public String toString()`**
  - return a string representation of the bag having the form IntArrayBag[a,b,c…]
  - This is not in Main's original class

Methods (7)

- `public static IntArrayBag union(`
  ```
      IntArrayBag b1,
      IntArrayBag b2)
  ```
  - Create a new bag that contains all the elements from the two specified bags
  - This is a static version of the addAll instance method.
  - Bags b1 and b2 are unchanged.

Pre and Post conditions

- They are documentation conventions
- Method Precondition (`@precondition`)
  - what must be true before the method is called in order that the method execute correctly
- Method Postcondition (`@postcondition`)
  - what must be true after the method executes correctly
- Postcondition can also be done with @return if there is a return value.
- We have custom javadoc tags (next slide)

Example: default constructor

- **`public IntArrayBag()`**
- precondition
  - none so it can be omitted
- postcondition
  - This bag is empty and has an initial capacity of 10

Example: constructor

-
  ```
  public IntArrayBag(
                  int initialCapacity)
  ```

- precondition
  - initialCapacity is non-negative
- postcondition
  - This bag is empty and has the given initial capacity

Example: add

- `public void add(int element)`
- precondition
  - none so it can be omitted. In a fixed size implementation this could be "bag is not full"
- postcondition
  - A new copy of element has been added to this bag

Example: addMany

-
  ```
  public void addMany(
                      int ... elements)
  ```

- precondition
  - none so it can be omitted. In a fixed size implementation this could be "bag is not full"
- postcondition
  - New copies of the specified elements have been added to this bag

Example: addAll

- `public void addAll(`
  `    IntArrayBag addend)`
- precondition
  - the parameter addend is not null
- postcondition
  - The elements from addend were added to this bag
- Note: we could have no precondition by agreeing that addAll does nothing if addend is null. (Main: NullPointerException)

Example: clone

- `public IntArrayBag clone()`
- precondition
  - none so it can be omitted
- postcondition
  - A copy of this bag is returned. Subsequent changes to the copy will not affect the original nor vice versa.
- Note: @postcondition can be done with @return

Example: countOccurrences

- `public int countOccurrences(`
  ```
                                int target)
  ```
- Precondition
  - none so it can be omitted.
- Postcondition (can be done with @return)
  - the number of times that target occurs in this bag is returned.

Example: ensureCapacity

-
  ```
  public int ensureCapacity(
                 int minimumCapacity)
  ```
- Precondition
  - minimumCapacity > 0
- Postcondition
  - This bag's capacity has been changed to at least minimumCapacity. If the capacity was already at or greater than minimumCapacity, then the capacity is left unchanged.

Example: getCapacity

- **`public int getCapacity()`**
- Precondition
  - none so it can be omitted
- Postcondition (can be done with @return)
  - the current capacity of this bag is returned.

Example: remove

- `public boolean remove(`
  `                        int target)`
- Precondition
  - none so it can be omitted
- Postcondition (can be done with @return)
  - If target was found in this bag, then one copy of target has been removed and the method returns true. Otherwise the bag remains unchanged and the method returns false.

Example: size

- **`public int size()`**
- Precondition
  - none so it can be omitted
- Postcondition (can be done with @return)
  - The current number of elements in this bag is returned.

Example: trimToSize

- `public void trimToSize()`
- Precondition
  - none
- Postcondition
  - This bag's capacity has been changed to its current size.

Example: union

- `public static IntArrayBag union(`
  ```
  IntArrayBag b1,
  IntArrayBag b2)
  ```
- Precondition
  - neither b1 nor b2 are null
- Postcondition
  - The bag returned is the union of b1 and b2

Example: toString

- **`public String toString()`**
- Precondition
  - none
- Postcondition
  - A String representation of this bag is returned in the form IntArrayBag[1,2,3,3,4,5] with no guaranteed order since there is no implied ordering for a bag

Javadoc

- Class comment describing the ADT
- Each Constructor/Method should have a javadoc comment containing
  - General sentence(s) describing method
  - @param (if there are parameters)
  - @precondition (custom tag)
  - @postcondition (void methods)
  - @postcondition or @return (non-void)
  - @throws (@exception) if appropriate
  - @note (custom tag) if appropriate
- Never refer to implementation details in javadoc comments

Using/Testing add

Create an empty bag and add some integers to it.
Display the results

```
IntArrayBag b = new IntArrayBag();
b.add(1); b.add(2); b.add(3);
b.add(4); b.add(3);
System.out.println(b);
```

Following result is displayed (may get a different order since order is nuimportant

```
IntArrayBag[1,2,3,4,3]
```

Using/Testing addMany

Create an empty bag and add some integers to it using the variable argument feature of Java 5.
Display the results

```
IntArrayBag b = new IntArrayBag();
b.addMany(1,2,3,4,3);
System.out.println(b);
```

Following result is displayed (may get a different order since order is unimportant

```
IntArrayBag[1,2,3,4,3]
```

Using/Testing remove

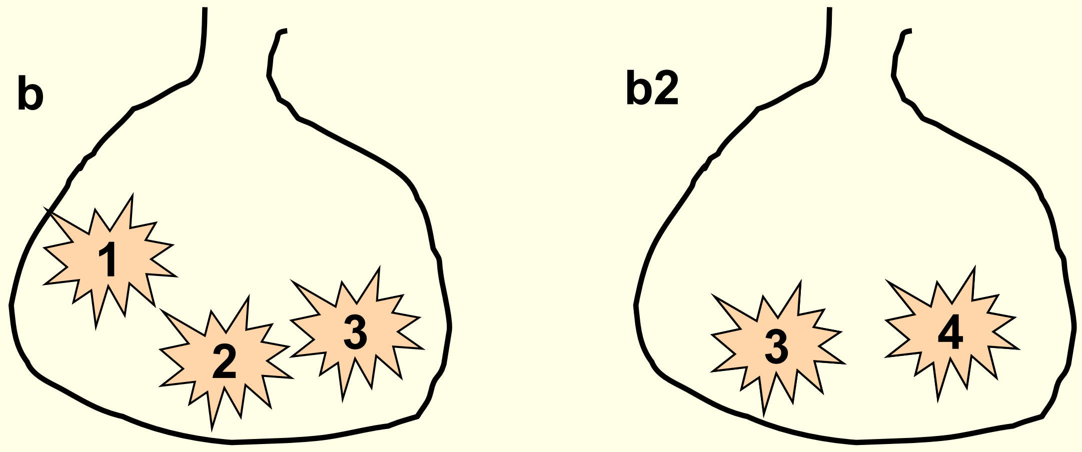
Create an empty bag and add some integers to it, remove some elements and display the results

```
IntArrayBag b = new IntArrayBag();
b.addMany(1,2,3,4,3);
System.out.println(b);
b.remove(3); System.out.println(b);
b.remove(1); System.out.println(b);
```

Following result is displayed

```
IntArrayBag[1,2,3,4,3]
IntArrayBag[1,2,3,4]
IntArrayBag[4,2,3]
```

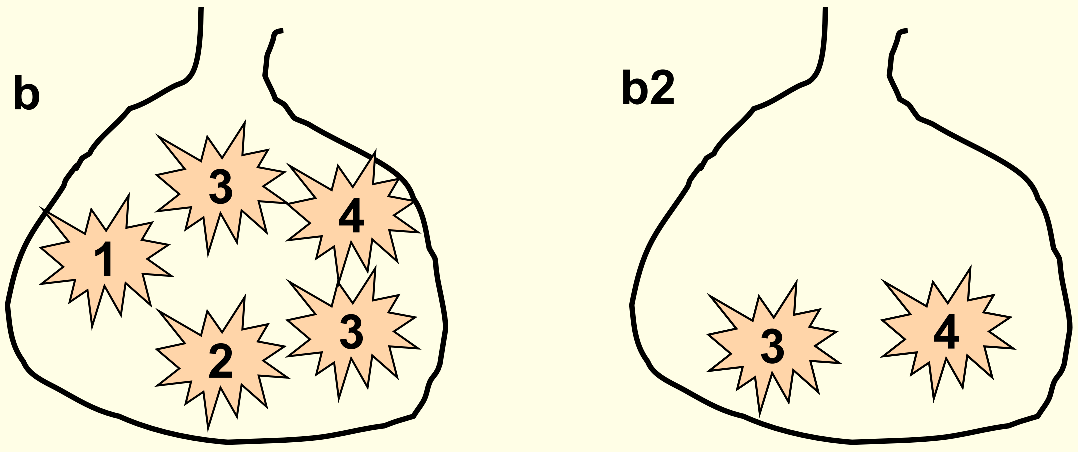
The addAll method (before)



```
IntArrayBag b = new IntArrayBag();
b.add(1); b.add(2); b.add(3);

IntArrayBag b2 = new IntArrayBag();
b2.add(3); b2.add(4);

b.addAll(b2);
```

b2 is unchanged by this operation

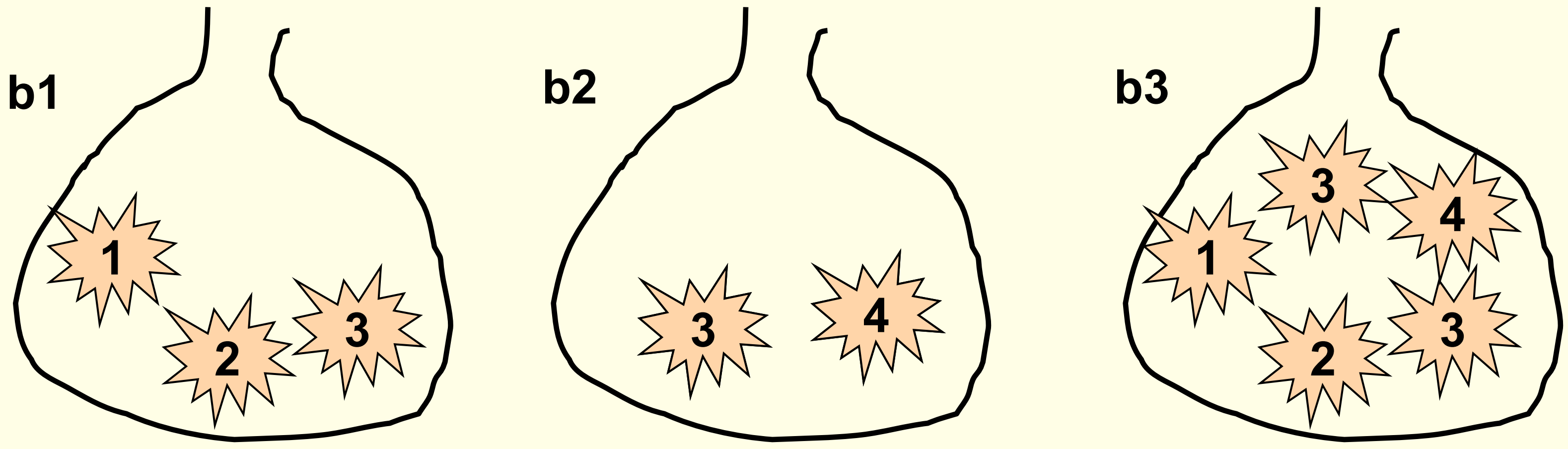
The addAll method (after)



```
IntArrayBag b = new IntArrayBag();
b.add(1); b.add(2); b.add(3);

IntArrayBag b2 = new IntArrayBag();
b2.add(3); b2.add(4);

b.addAll(b2);
```

b2 is unchanged by this operation

The static union method



```
IntArrayBag b1 = new IntArrayBag();
b1.add(1); b1.add(2); b1.add(3);

IntArrayBag b2 = new IntArrayBag();
b2.add(3); b2.add(4);

IntArrayBag b3 = IntArrayBag.union(b1,b2);
```

IntArrayBag

Implementation

Data fields

- Internally an IntArrayBag can use an array of type int[] to hold the elements and an integer variable to indicate how many elements are currently being used:

```
public class IntArrayBag
{
  private int[] data;
  private int manyItems;
  private static final int INITIAL_CAPACITY = 
10;
  ...
```

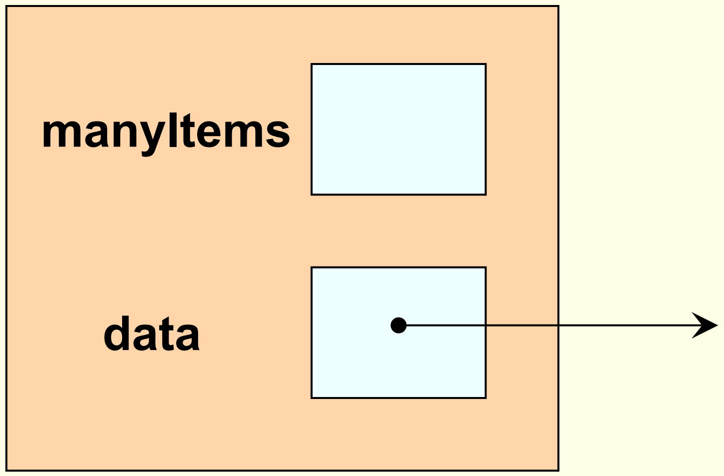
Invariant of an ADT

- The ADT invariant specifes relationships among the data fields (instance variables) that must always be true before and after the execution of any instance method.
- It provides documentation for the state of an object as defined by the instance variables

Invariant of IntArrayBag

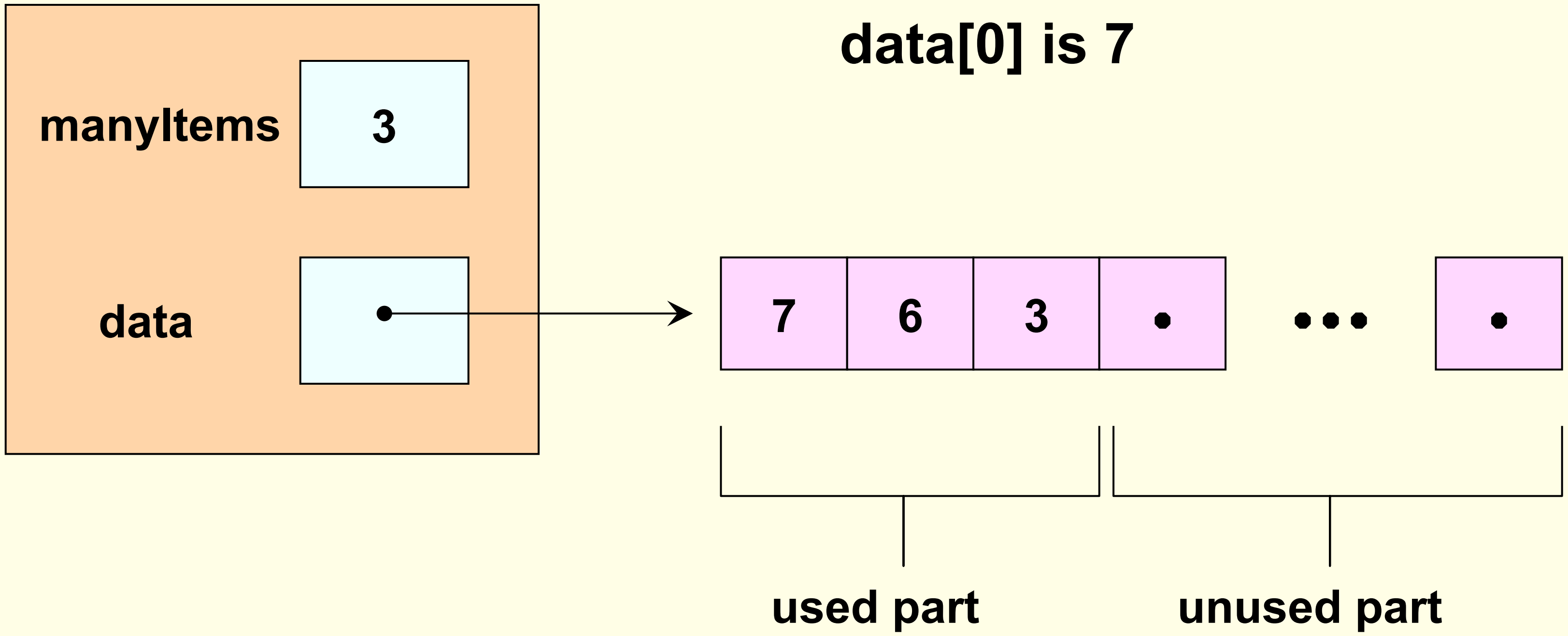
- Number of elements in the bag is stored in the instance variable manyItems which must satisfy
  `0 <= manyItems <= data.length`
- If the bag is empty then manyItems is 0 and contents of the data array are not relevant
- If the bag is nonempty the elements in the bag are stored in **`data[0]`** to **`data[manyItems-1]`** and we don't care what is stored in the unused part of the array

An IntArrayBag object (1)



Here there are two instance data fields:
(1) manyItems gives the number of elements currently in the bag
(2) data is a reference to an array of integers holding the bag elements

An IntArrayBag object (2)



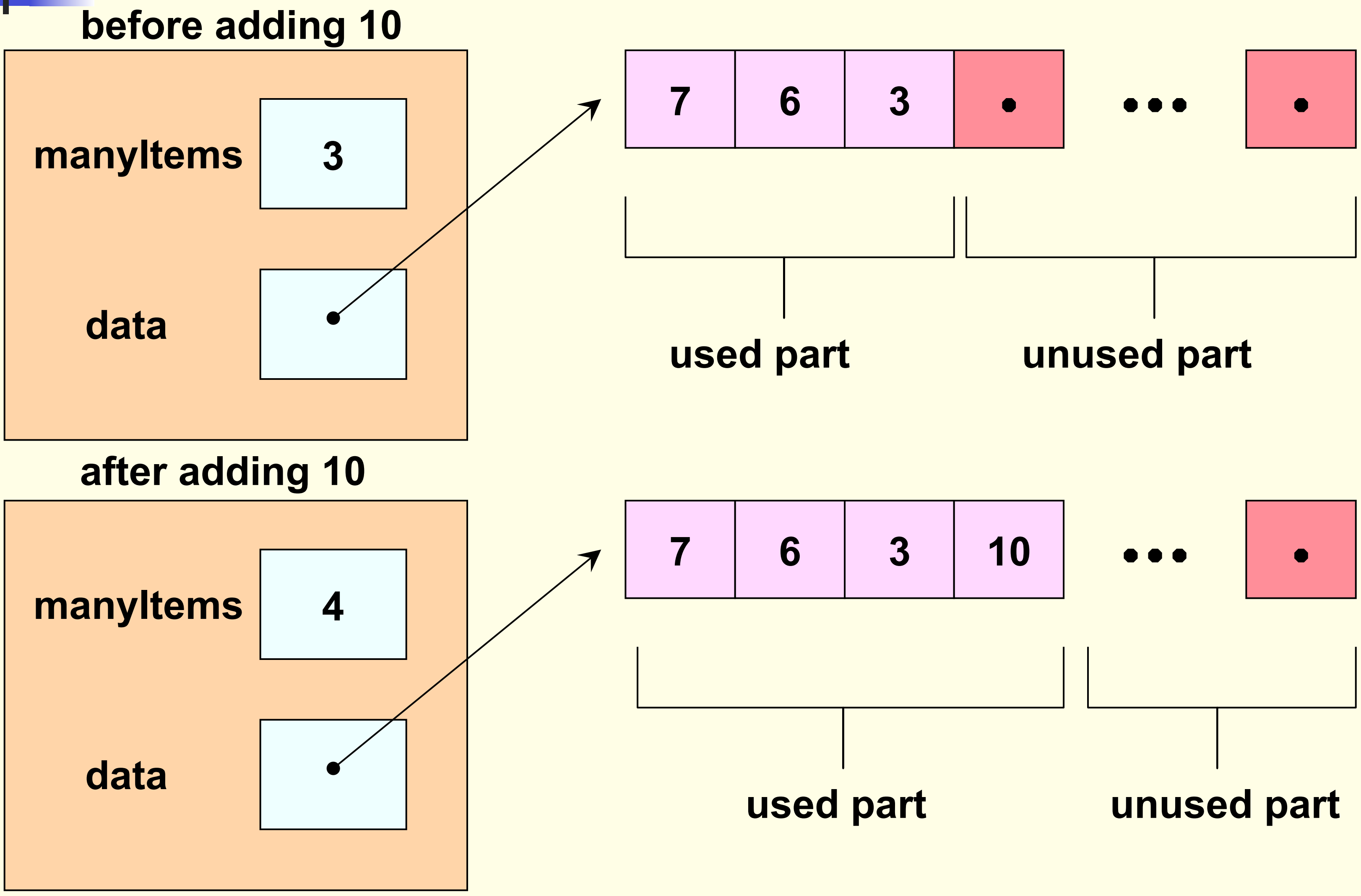
After an IntArrayBag object is constructed the array of integers exists and data is a reference to this array. At any time the first part of the array will hold the bag elements and the remaining part of the array is unused.

IntArrayBag constructors

Default constructor and one for specified capacity.

```
public IntArrayBag()
{
   this(INITIAL_CAPACITY);
}

public IntArrayBag(int initialCapacity)
{
   if (initialCapacity < 0)
      throw new IllegalArgumentException(
         "The initialCapacity is negative: "
         + initialCapacity);
   data = new int[initialCapacity];
   manyItems = 0;
}
```

add method (1)


before adding 10
manyItems
3
data
7
6
3
•
• • •
•
used part
unused part
after adding 10
manyItems
4
data
7
6
3
10
• • •
•
used part
unused part

add method (2)

Add a new element to this bag. If the new element would take this bag beyond its current capacity then the capacity is increased before adding the new element

```
public void add(int element)
{
   if (manyItems == data.length) // bag is full
   {
      ensureCapacity(manyItems*2 + 1);
   }
   data[manyItems] = element;
   manyItems++;
}
```

manyItems gives the next available position

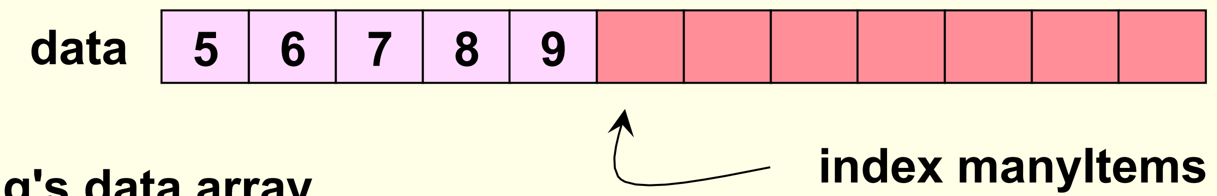
addMany method

Add new elements to this bag. If the new elements would take this bag beyond its current capacity then the capacity is increased before adding each new element

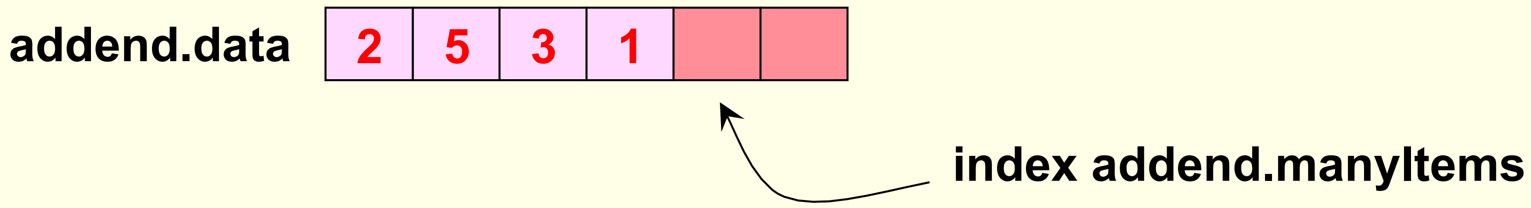
```
public void addMany(int... elements)
{
   int n = manyItems + elements.length;
   if (n > data.length)
   {
      ensureCapacity(n*2);
   }
   System.arraycopy(elements, 0, data, manyItems,
      elements.length;);
   manyItems = manyItems + elements.length;
}
```

addAll method (1)

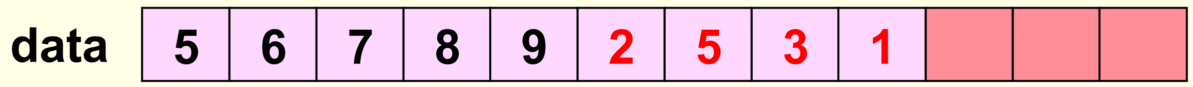
this bag's data array before addAll



addend bag's data array



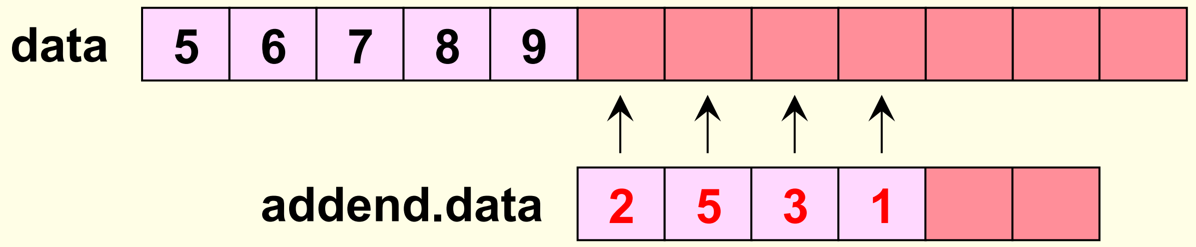
this bag's data array after addAll



addAll method (2)

Add the contents of another bag to this bag

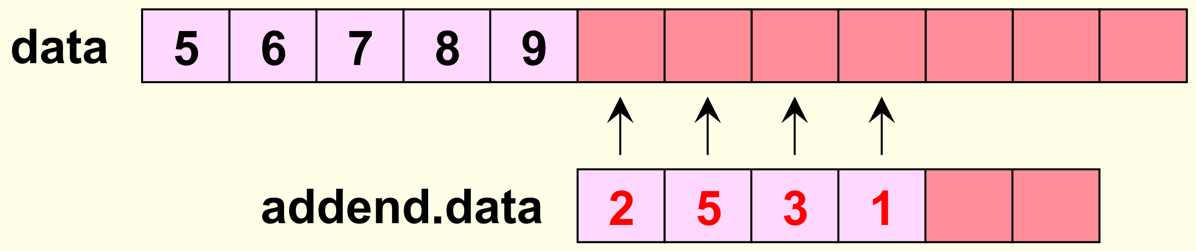
```
public void addAll(IntArrayBag addend)
{
   ensureCapacity(manyItems + addend.manyItems);
   System.arraycopy(
      addend.data, 0, // all of addend array
      data, manyItems, // start after end of data
      addend.manyItems // copy this many items
   )
   manyItems += addend.manyItems; // update
}
```



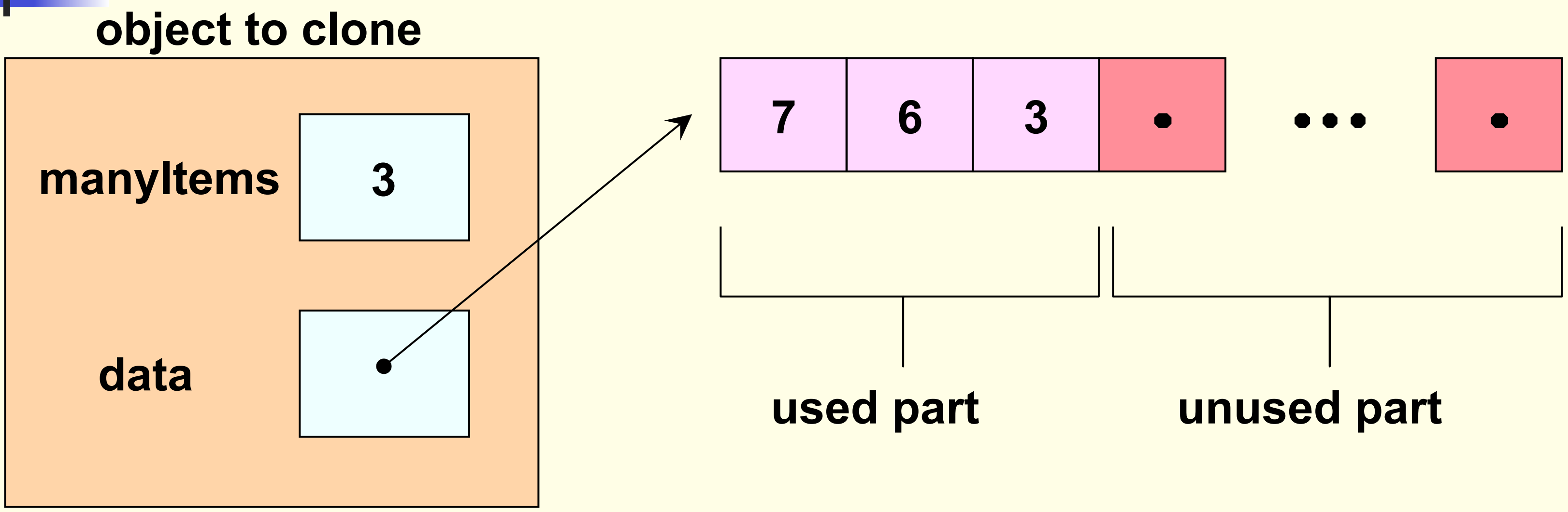
addAll method (3)

A less efficient for-loop version of the addAll method

```
public void addAll(IntArrayBag addend)
{
   ensureCapacity(manyItems + addend.manyItems);
   for (int k = 0; k < addend.manyItems; k++)
   {
      data[manyItems + k] = addend.data[k];
   }
   manyItems += addend.manyItems;
}
```



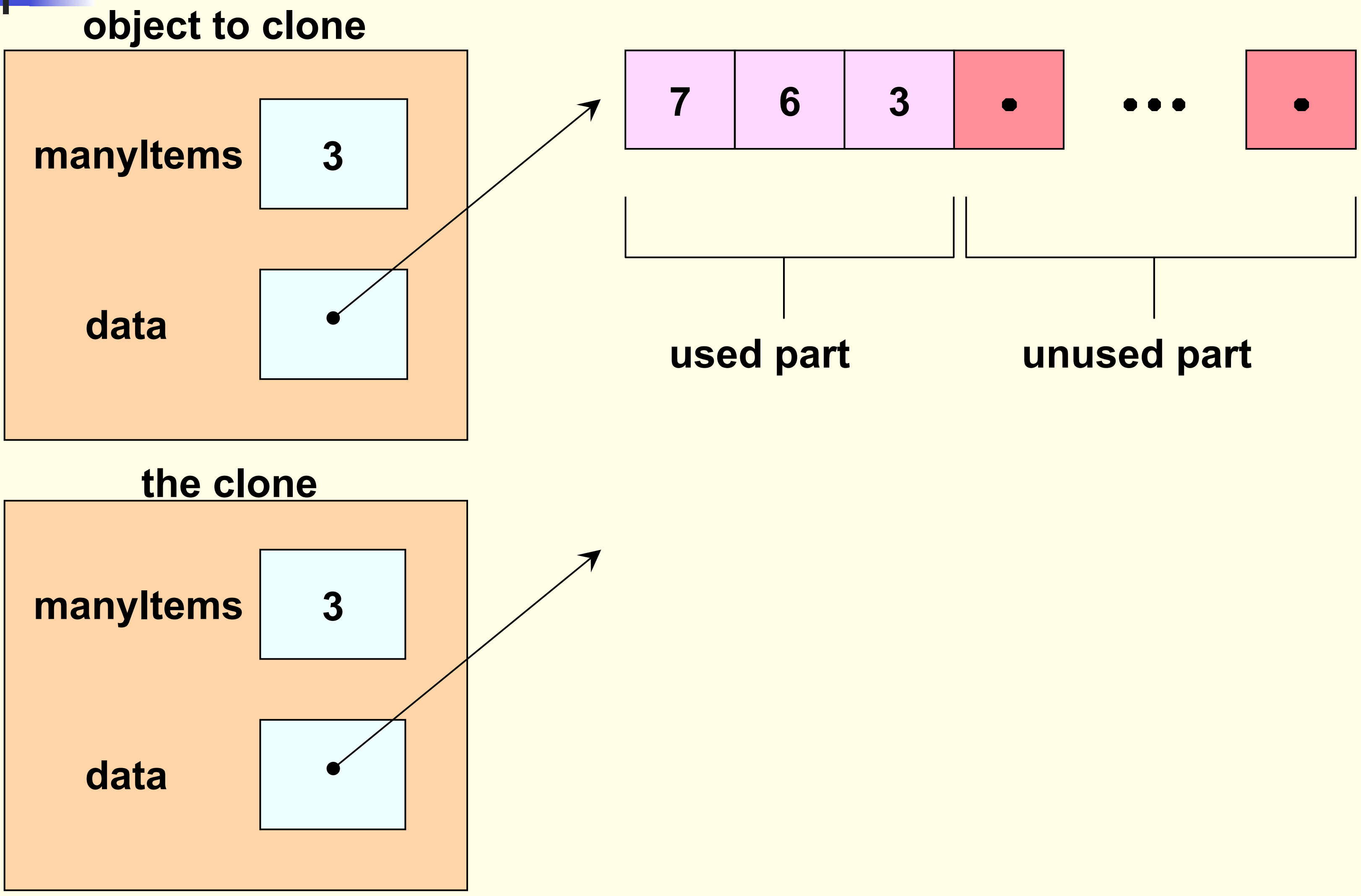
clone method (1-3)



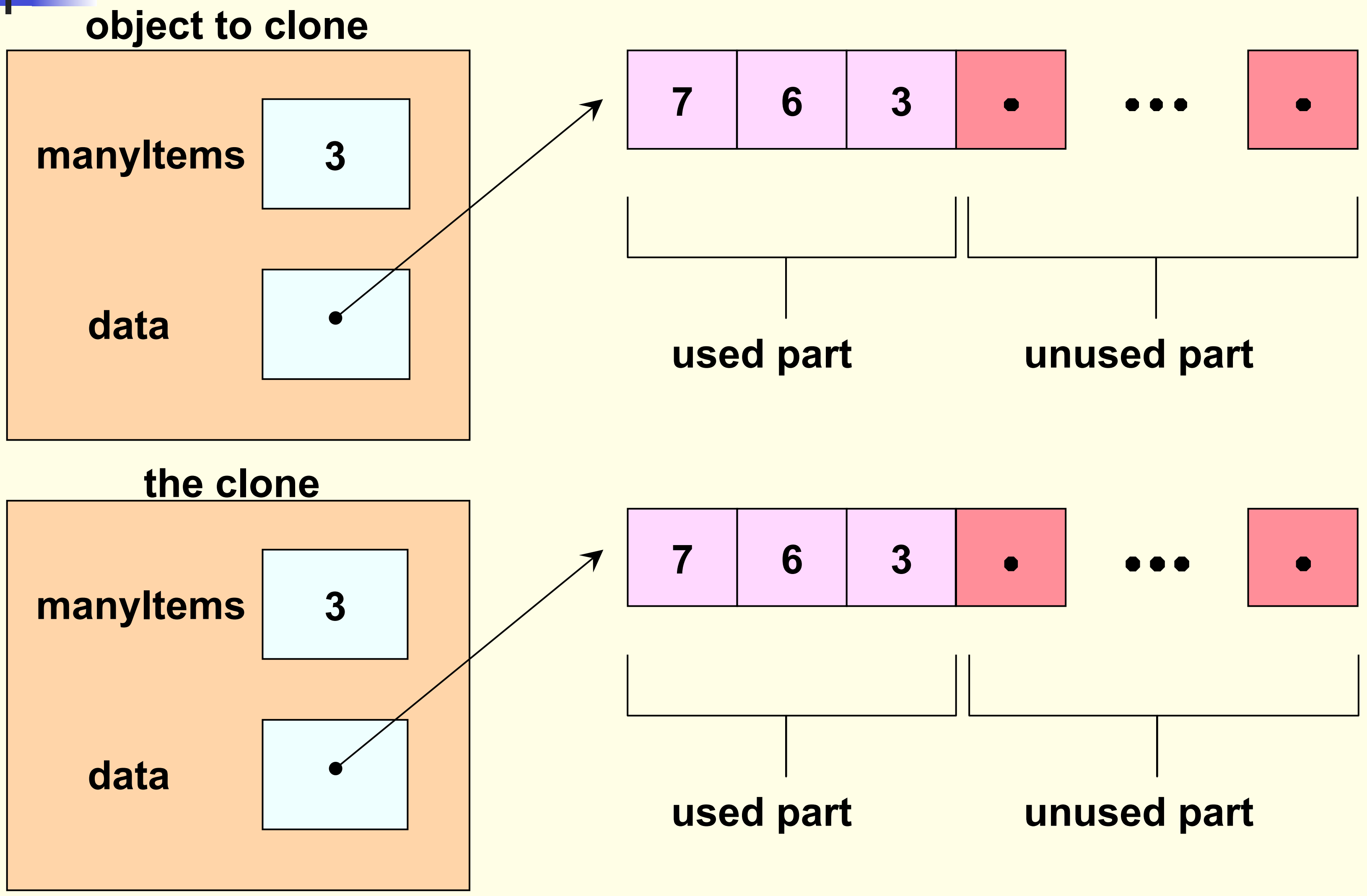
This will give a deep clone for primitive types

Clone for an array is alread y built-in to Java

clone method (2-3)



clone method (3-3)



clone method (2)

Make a clone of this bag.
It's actually a deep clone since the array is of primitive type.
First clone the instance data fields
Then clone the integer array referenced by data

```
public IntArrayBag clone()
{   IntArrayBag answer;
    try
    {   answer = (IntArrayBag) super.clone();
    } catch (CloneNotSupportedException e)
    {
        throw new RunTimeException("...");
    }
    answer.data = data.clone();
    return answer;
}
```

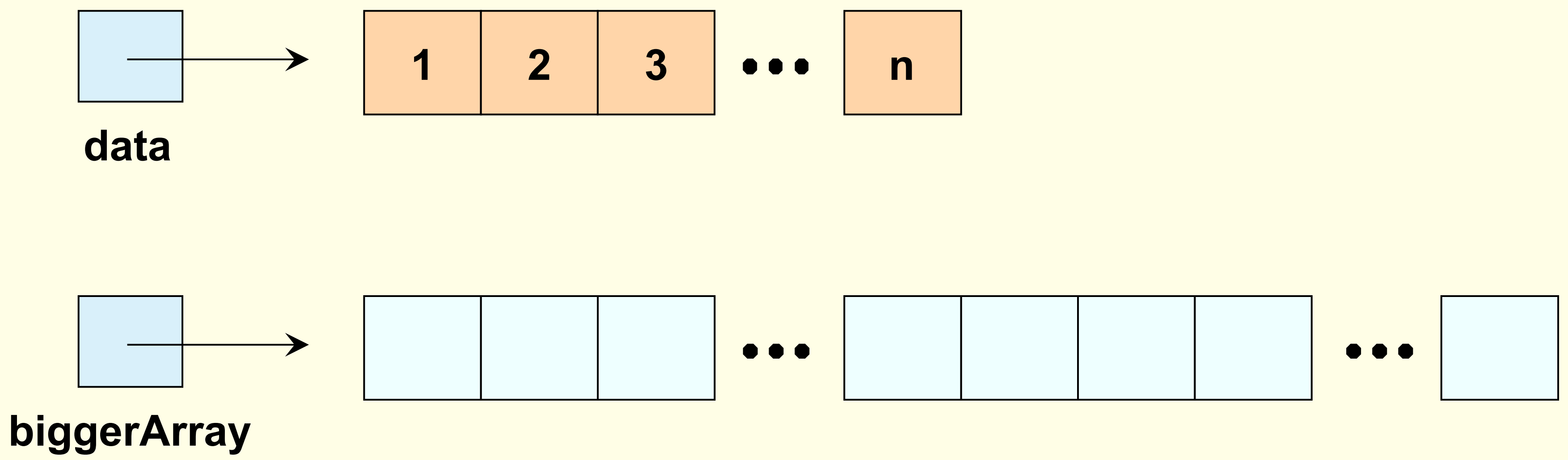
important

countOccurrences method

Process the array from index 0 to index manyItems - 1 and count the number of times the given target integers occurs

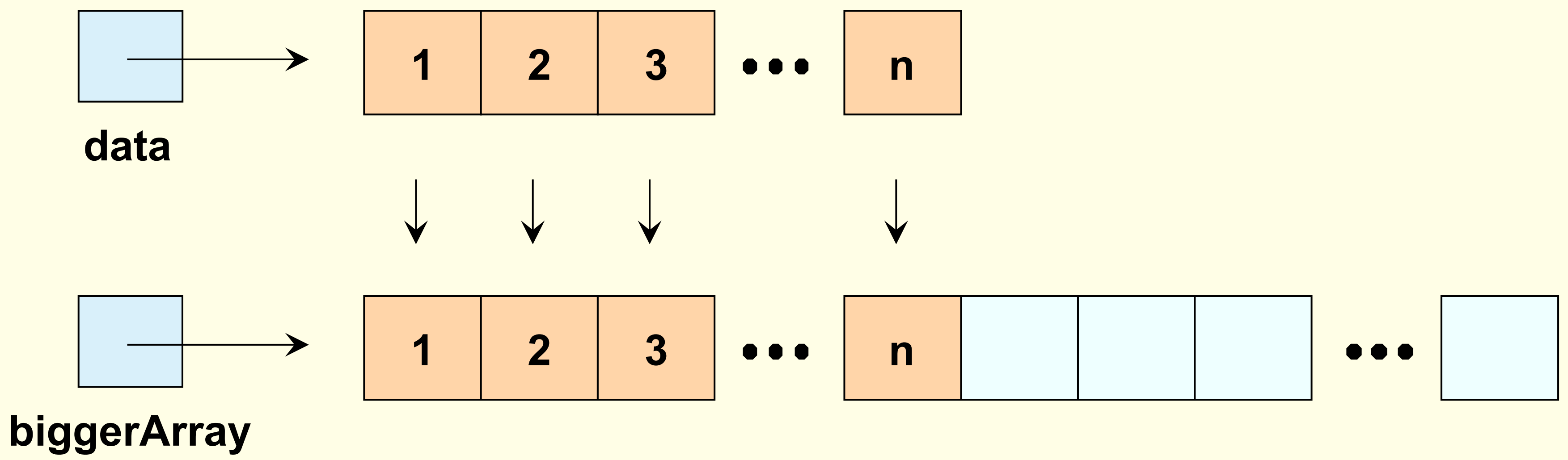
```
public int countOccurrences(int target)
{
   int answer = 0; // intialize counter

   for (int k = 0; k < manyItems; k++)
   {
      if (target == data[k])
         answer++;
   }
   return answer;
}
```

ensureCapacity method (1-5)



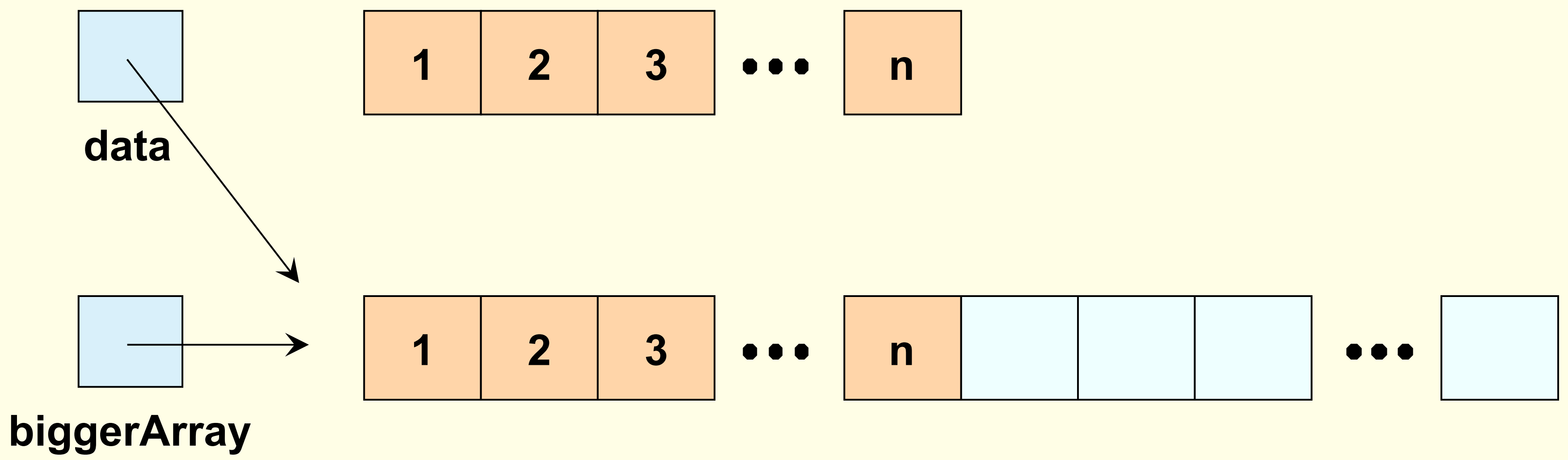
Make space for a bigger array twice the size

ensureCapacity method (2-5)



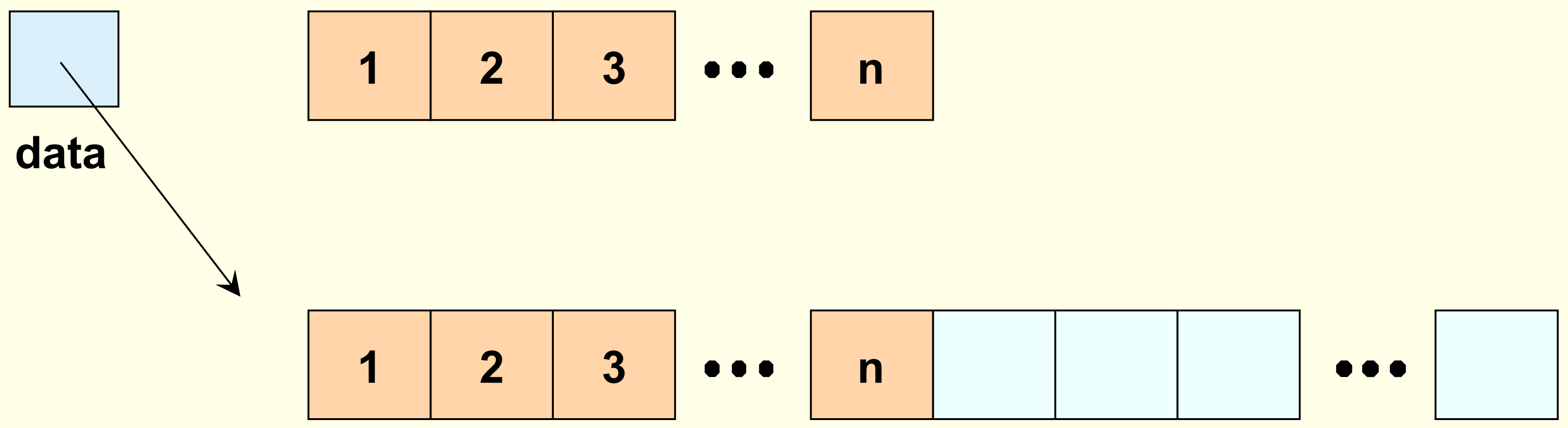
Copy the data to it

ensureCapacity method (3-5)



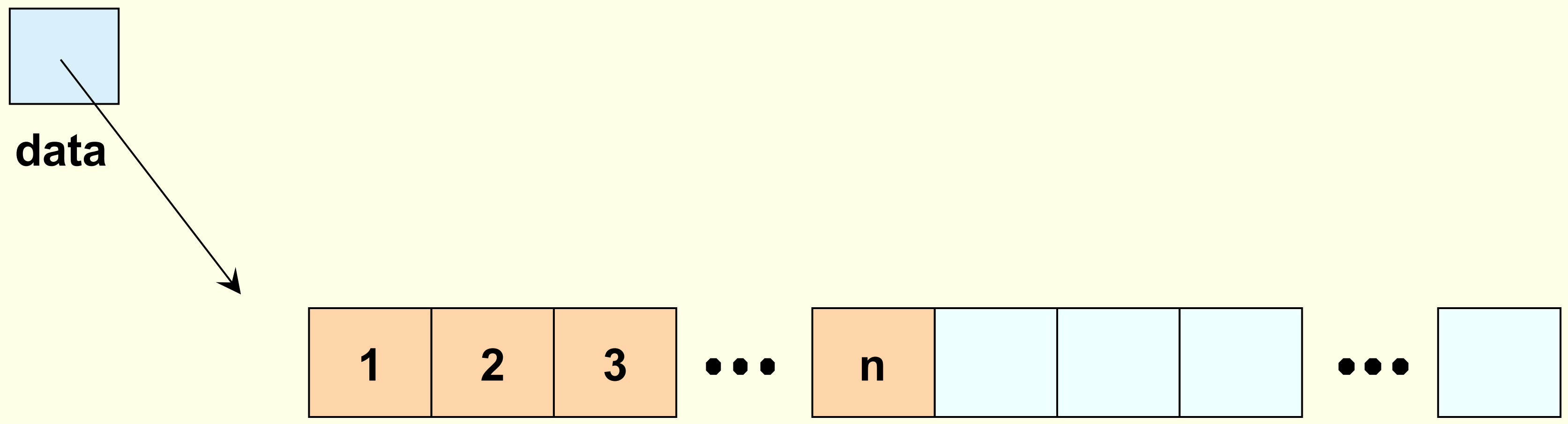
reset the reference (private data field) to reference the bigger array

ensureCapacity method (4-5)



local reference for copy disappears when method exits

ensureCapacity method (5-5)



original array is garbage collected

ensureCapacity method (2)

Change capacity of this bag to at least minimumCapacity.
If capacity was already at or greater than minimumCapacity
the capacity is left unchanged

```
public void ensureCapacity(int minimumCapacity)
{
   int[] biggerArray;
   if (data.length < minimumCapacity)
   {
      biggerArray = new int[minimumCapacity];
      System.arraycopy(data, 0, biggerArray,
         0, manyItems);
      data = biggerArray;
   }
}
```

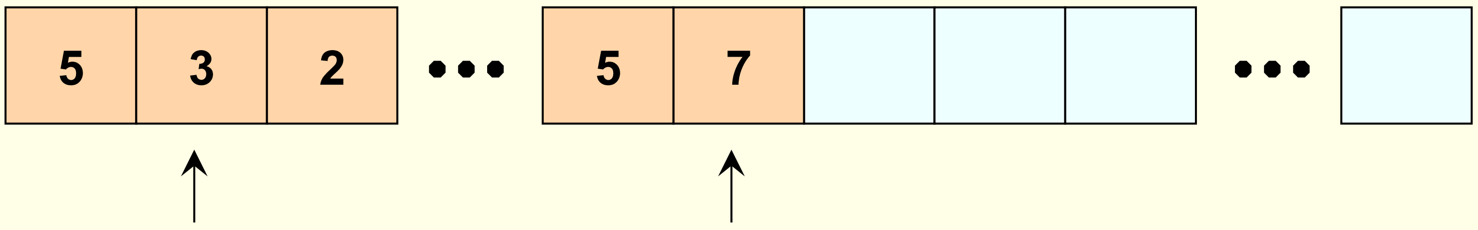
getCapacity method

Just return the current maximum size of the data array

```
public int getCapacity()
{
   return data.length;
}
```

remove method (1-1)

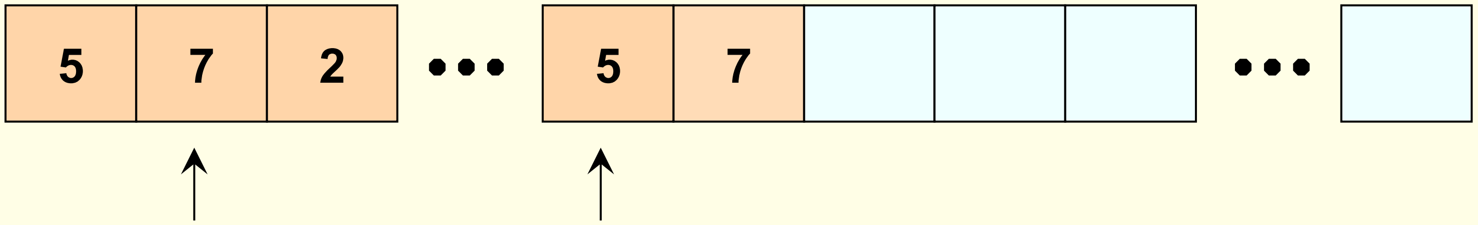
Remove integer 3 at index 1 in array of size manyItems



```
data[index] = data[manyItems - 1];
manyItems--;
```

remove method (1-2)

This is done by copying last used array element to position one.
This effectively removes the 3 in O(1) time instead of O(n) time



```
data[index] = data[manyItems - 1];
manyItems--;
```

These are equivalent

```
manyItems—;
data[index] = data[manyItems];
```

```
data[index] = data[--manyItems];
```

remove method (2)

- An alternate approach would be to shift array elements left to close up the hole
- This would make remove an O(n) operation instead of O(1)
- We can simply copy the last element to the index of the element to remove because there is no ordering of bag elements that must be preserved.
- Could also have a removeAll method to remove all occurrences of a given element.

remove method (3)

- Internal order of bag
  - there is an internal order since we are using an array and an array has an order
- This internal order is not visible to the user of the class so we don't need to maintain the internal order of the elements.
- When we do array implementations of ordered collections the internal order will be important.

remove method (4)

Search for the given target. If it is not found return false else delete the target and return true. To keep the array contiguous we can simply copy the last element used to the position of the element deleted

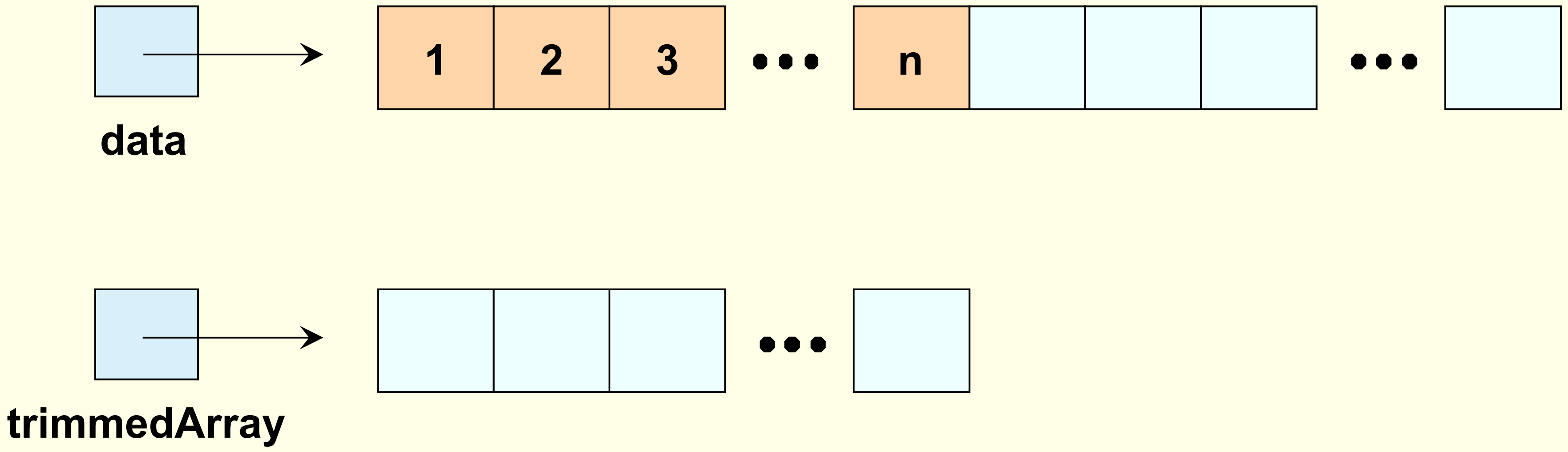
```
public boolean remove(int target)
{   int index = 0;
    while((index < manyItems) && (target != data[index]))
    {   index++;
    }
    if (index == manyItems) // didn't find target
        return false;
    else // found target at index
    {   manyItems--;
        data[index] = data[ manyItems];
        return true;
    }
}
```

size method

Return number of elements currently in the bag

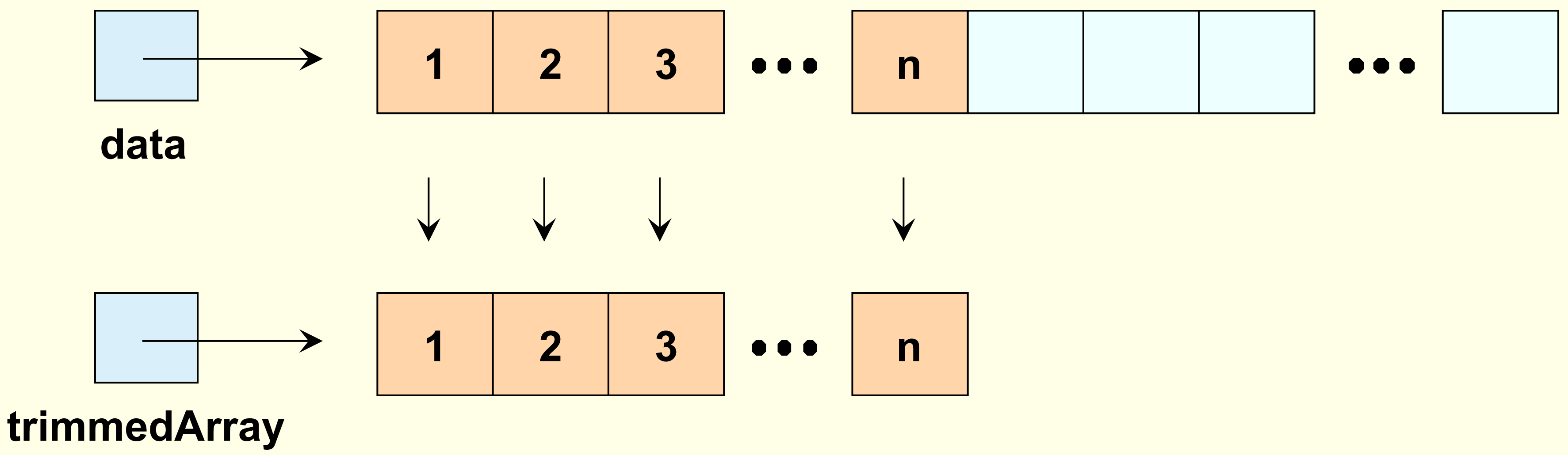
```
public int size()
{
    return manyItems;
}
```

trimToSize method (1-5)



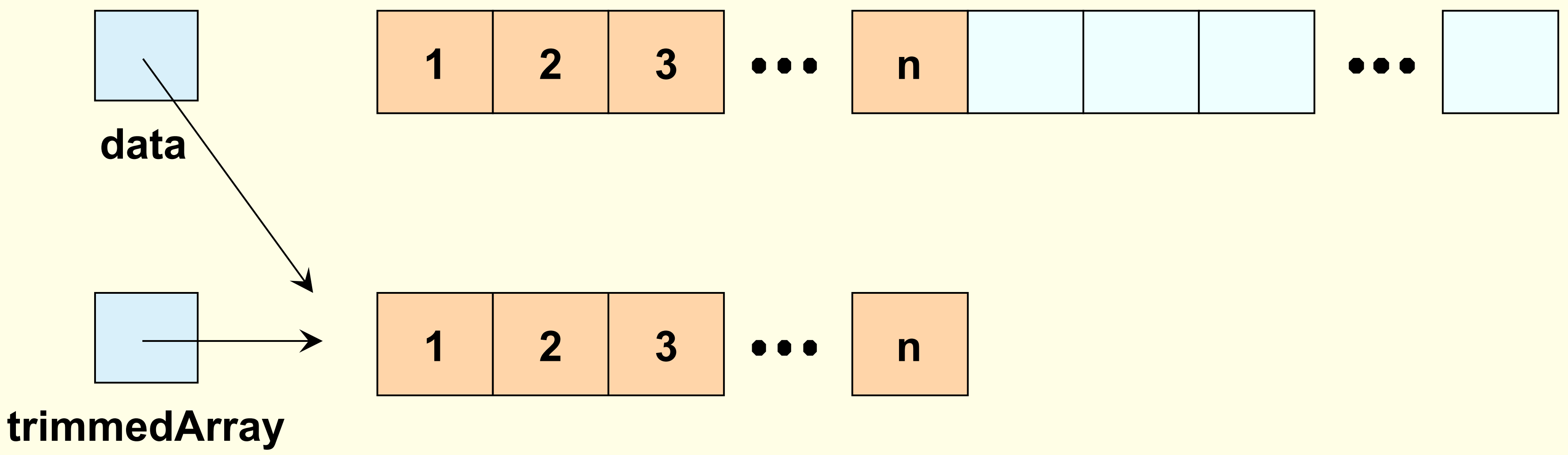
Make a smaller array that contains room for exactly the n array elements referenced by data. If data.length is manyItems we do nothing

trimToSize method (2-5)



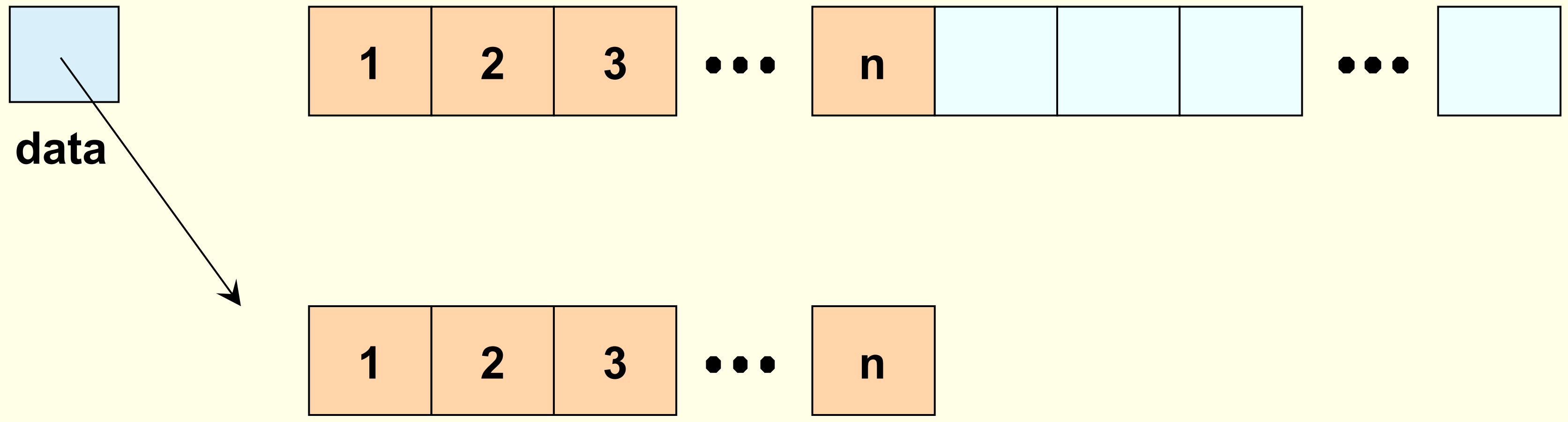
Now copy the elements from array referenced by data to the array referenced by trimmedArray

trimToSize method (3-5)



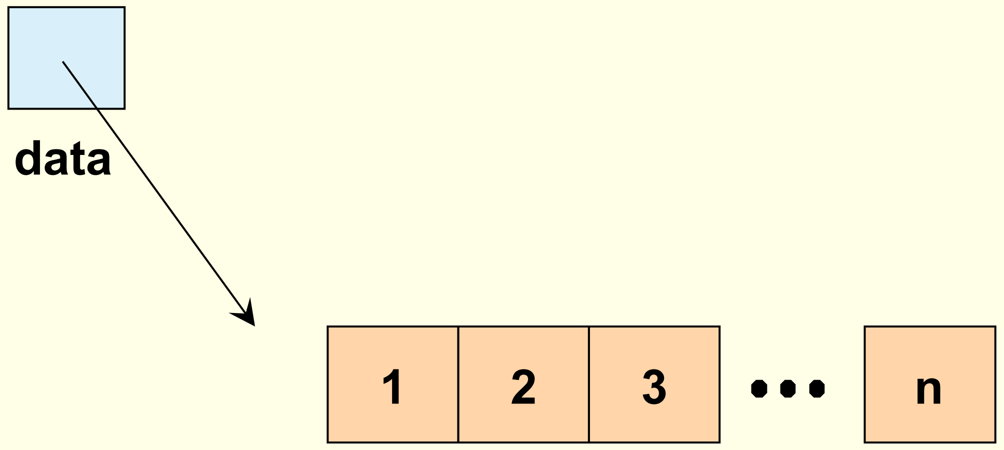
Assign the trimmedArray reference to the data reference

trimToSize method (4-5)



Local trimmedArray reference disappears when method exits

trimToSize method (5-5)



Original array is eventually garbage collected

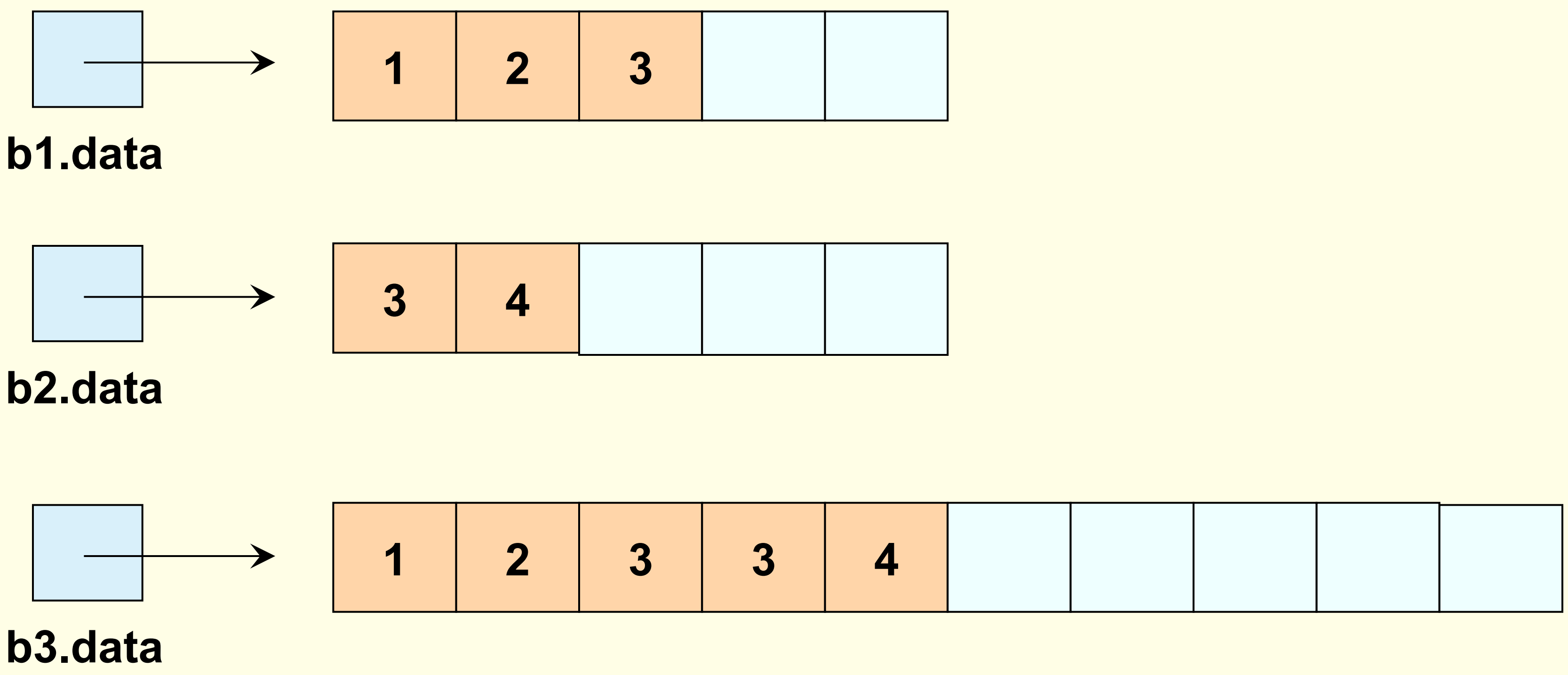
trimToSize method (2)

Reduce that capacity of the bag if necessary to its actual size as given by manyItems. Since manyItems <= data.length the size is reduced only if data.length != manyItems.

```
public void trimToSize()
{
   int[] trimmedArray;
   if (data.length != manyItems)
   {
      trimmedArray = new int[manyItems];
      System.arraycopy(data, 0, trimmedArray,0,
         manyItems);
      data = trimmedArray;
   }
}
```

could use < instead of !=

union method (1)



union method (2)

Make a new bag which is the union of two given bags

```
public static IntArrayBag union(IntArrayBag b1,
                                            IntArrayBag b2)
{
   IntArrayBag answer = new IntArrayBag(
      b1.getCapacity() + b2.getCapacity());

   System.arraycopy(b1.data, 0, answer.data, 0,
      b1.manyItems);

   System.arraycopy(b2.data, 0, answer.data, b1.manyItems,
      b2.manyItems);

   answer.manyItems = b1.manyItems + b2.manyItems);
   return answer;
}
```

toString method

Main does not do this: String returned has the form IntArrayBag[1,2,3,3,4]

```
public String toString()
{   StringBuffer s = new StringBuffer();
    s.append("IntArrayBag[");
    for (int k = 0; k < size(); k++)
    {
        s.append(data[k]);
        // don't put comma after last element
        if (k != size() - 1)
            s.append(",");
    }
    s.append("]");
    return s.toString();
}
```

use StringBuilder

Adding an equals method

- An equals method with prototype `public boolean equals(Object obj)` overriding `Object` equals could be added
- This method returns true if this bag contains exactly the same elements as the bag obj specified as the parameter
- programming project 1 in Main's book.

A random remove method

- We have a method for removing a given element from a bag
- We could also write a remove method with prototype

  ```
  public int remove()
  ```

  that removes a random element from the bag and returns it.
  - not in Main's book

COSC 2006: Data Structures I

The IntArraySet Collection Class

This is a programming project

in Main's book

IntArraySet class design

```
public class IntArraySet implements Cloneable
{
   public IntArraySet() {...}
   public IntArraySet(int initialCapacity) {...}

   public void add(int element) {...}
   public void addMany(int ... elements) {...}
   public void addAll(IntArraySet addend) {...}
   public IntArraySet clone() {...}
   public boolean contains(int target) {...}
   public void ensureCapacity(int minimumCapacity) {...}
   public int getCapacity() {...}
   public boolean remove(int target) {...}
   public int size() {...}
   public void trimToSize() {...}
   public static IntArraySet union(
      IntArraySet b1, IntArraySet b2) {...}
   public boolean equals(Object obj) {...} // added
   public String toString() {...} // added
}
```

replaces countOccurrences

IntArraySet methods (1)

- The IntArrayBag add, addMany, and addAll methods must be modifed to prevent duplicates:
  - add, addMany
    - If set already contains the element then it is not added to the set
  - addAll
    - When adding the addend Set to this set duplicates must be avoided
  - union
    - as above we must avoid duplicates
- Cannot use System.arraycopy in these cases

IntArraySet methods (2)

- The IntArrayBag countOccurrences method can only return 0 or 1 for a set so we can replace this method by the following one

```
public boolean contains(int target)
```

- This method returns true if target is found in the set else it returns false. It can also be used in add and addAll methods.

IntArraySet methods (3)

- The following methods are unchanged for the IntArraySet implementation
  - constructors
  - clone
  - ensureCapacity
  - getCapacity
  - size
  - trimToSize

IntArraySet methods (4)

- In IntArrayBag we provided a toString method to return a string representation of a bag
- We can also include a similar method in the IntArraySet class

IntArraySet methods (5)

- In IntArrayBag we can also have an equals method (programming project 1)
- Include a similar method in the IntArraySet class that overrides the Object equals method.
- Also include the random remove method with prototype **`public int remove()`** that removes a random element from the set.

COSC 2006: Data Structures I

The DoubleArraySeq Collection Class

This is a programming project

in Main's book

What is a sequence?

- A sequence is like a bag but with a definite order on the elements.
- A sequence has no direct access operations for locating an element with given index
- Element positions are all specified relative to a current position
  - Example: addBefore, addAfter, getCurrent
- An internal iterator can be used to go from the beginning to the end of the sequence.

Internal details (1)

- For a sequence implemented using a data array there will be both an internal and external order that must be the same.
- This is different from the Bag where there is no external order so the internal order does not need to be maintained.

Internal details (2)

- Since the bag had no order we were free to remove an element by moving the last used array element into the position for removal
  - 1,2,3,4,2,6,7 → 1,2,7,4,2,6 after removal of 3
- For the sequence this is not possible and the array elements beyond the position for removal must all be shifted one place
  - 1,2,3,4,2,6,7 → 1,2,4,2,6,7 after removal of 3

DoubleArraySeq class design

```
public class DoubleArraySeq implements Cloneable
{  public DoubleArraySeq() {...}
   public DoubleArraySeq(int initialCapacity) {...}
   public void addAfter(double element) {...}
   public void addBefore(double element) {...}
   public void addAll(DoubleArraySeq addend) {...}
   public DoubleArraySeq clone() {...}
   public void ensureCapacity(int minimumCapacity) {...}
   public int getCapacity() {...}
   public boolean removeCurrent() {...}
   public void trimToSize() {...}
   public void start() {...}
   public void advance() {...}
   public double getCurrent() {...}
   public boolean isCurrent() {...}
   public static DoubleArraySeq concatenate(
      DoubleArraySeq s1, DoubleArraySeq s2) {...}
   public int size() {...}
   public String toString() {...} // we added this
}
```

Internal Iterator for now

DoubleArraySeq methods (1)

- The following methods are essentially the same as for the bag
  - constructors
  - addAll
  - clone
  - ensureCapacity, trimToSize
  - getCapacity
  - size

DoubleArraySeq methods (2)

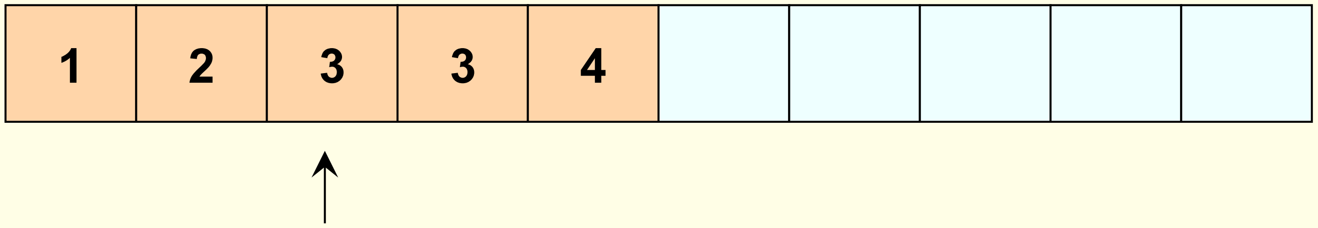
- There is no add method. Since the sequence is ordered we have two positional add methods
- public void addAfter(double element)
  - add an element after the current element
- public void addBefore(double element)
  - add an element before the current element

Operation of addAfter (1)

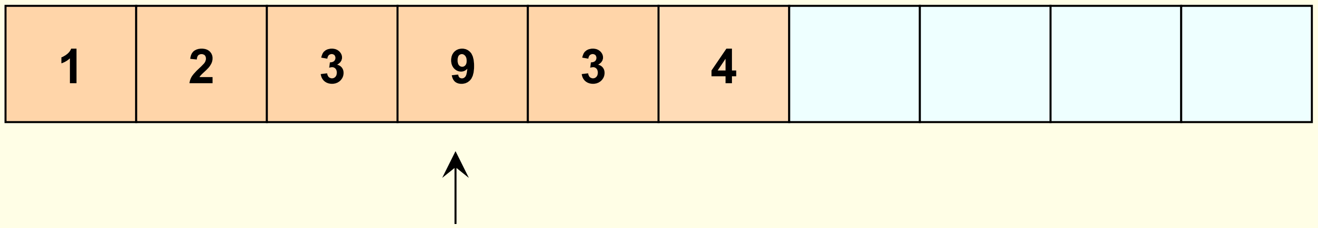
- public void addAfter(double element)
- postcondition
  - A new copy of element has been added to this sequence. If there was a current element, then the new element is placed after the current element.
  - If there was no current element, then the new element is placed at the end of the sequence
  - In either case the new element is now the current element.

Operation of addAfter (2)

The sequence <1,2,3,3,4> with current element 3 at index 2



Add 9 after the current element.
It becomes new current element with index 3



NOTE: If there is no current element the element is added at the end

Operation of addAfter (3)

The following statements make the sequence <1,2,3,4,..., 10>

```
DoubleArraySeq s = new DoubleArraySeq();
for (int k = 1; k <= 10; k++)
{
    s.addAfter(k);
}
```

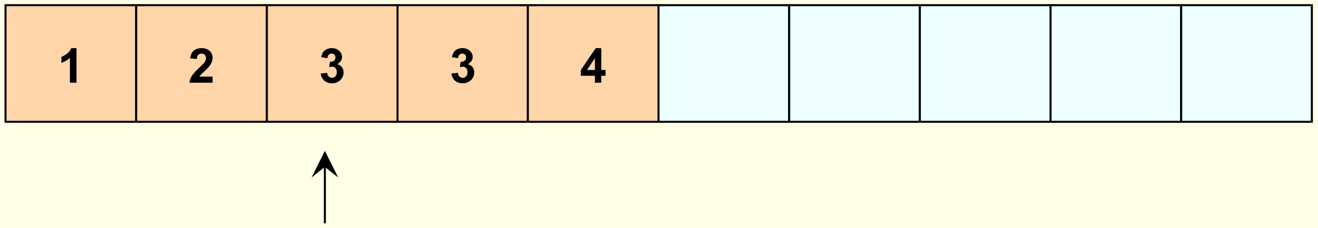
This works because there is no current element. Then each element is added at the end of the sequence and the last element is always the current element.
This is the way to construct sequences from the empty sequence.

Operation of addBefore (1)

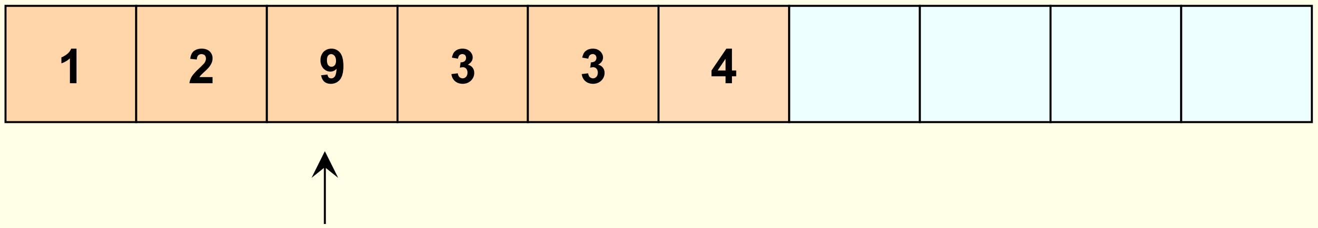
- public void addBefore(double element)
- postcondition
  - A new copy of element has been added to this sequence. If there was a current element, then the new element is placed before the current element.
  - If there was no current element, then the new element is placed at the start of the sequence
  - In either case the new element is now the current element.

Operation of addBefore (2)

The sequence <1,2,3,3,4> with current element 3 at index 2.



Add 9 before the current element.
It becomes new current element with index 2



NOTE: If there is no current element the element is added at the start

Operation of addBefore (3)

The following statements make the sequence <10,9,8,...,1>

```
DoubleArraySeq s = new DoubleArraySeq();
for (int k = 1; k <= 10; k++)
{
   s.addBefore(k);
}
```

It is faster to create a sequence like this using addAfter.

Iterator methods (1)

- There are two iterator methods that control the current element
  - public void start()
    - set the current element to be first element of sequence. If sequence is empty then the current element is undefined.
  - public void advance()
    - current element is now next element in sequence if there is one. Otherwise current element is undefined.

Iterator methods (2)

- There are two iterator methods that check for a current position and get the element at that position
  - public boolean isCurrent()
    - return true if there is a current position else return false. There is no current postion if sequence is empty or if advance() went off the end.
  - public double getCurrent()
    - returns the current element if there is one

Example

The following statements make the sequence <1,2,3,4,..., 10> and use the iterator to display the elements one per line

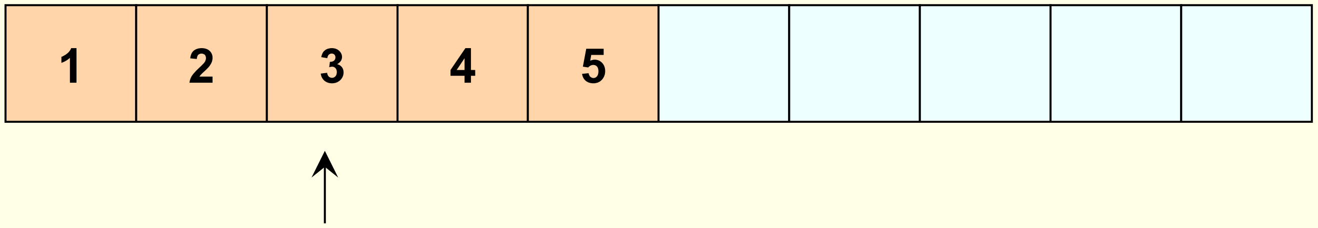
```
DoubleArraySeq s = new DoubleArraySeq();
for (int k = 1; k <= 10; k++)
{
   s.addAfter(k);
}


for (s.start(); s.isCurrent(); s.advance())
{
   System.out.println(s.getCurrent());
}
```

Operation of removeCurrent (1)

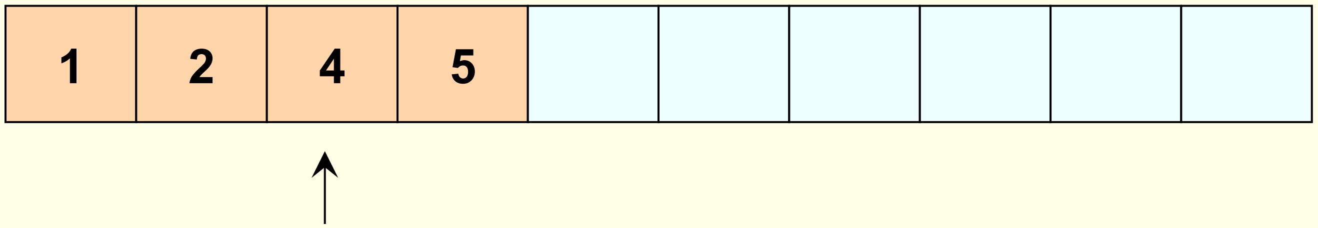
- public void removeCurrent()
- precondition
  - isCurrent() returns true
- postcondition
  - The current element is removed and the following element is the new current element. If there was no following element, then there is now no current element.

Operation of removeCurrent (2)

The sequence <1,2,3,4,5> with current element 3 at index 2

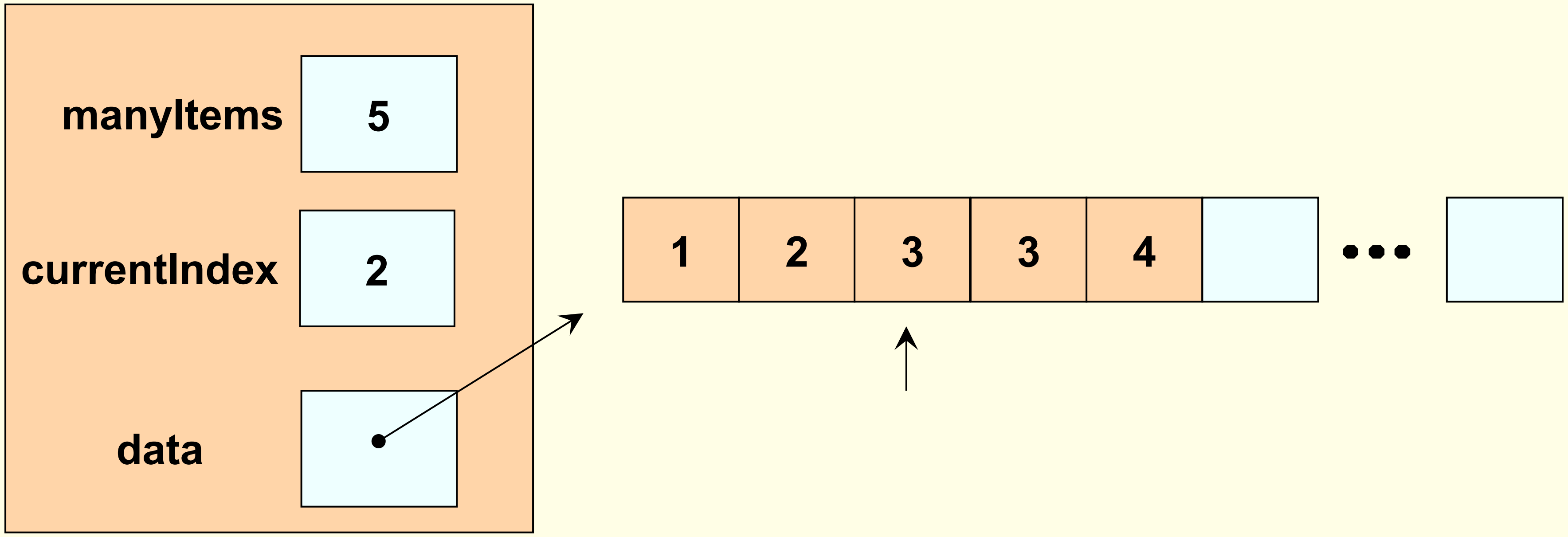


Remove the current element.
The one following it becomes the new current element



NOTE: If there is no following element then there is now no current element

A DoubleArraySeq object



manyItems is the size of the sequence, currentIndex is the index of the current element which is data[currentIndex].
currentIndex is not defined if manyItems is 0 or if currentIndex is manyItems - 1