Ulster University

COM498 Algorithms & Data Structures

2.3 Array Implementation of Bag

Fixed-Size Array to Implement Bag

- One way to implement the Bag ADT is to use a fixed-size array using a Java array to represent the entries in a bag (bag can become full like a real bag)
- Task is to define the methods previously specified in `BagInterface` where each public method corresponds to an ADT bag operation
- Will result in the class `ArrayBag` that implements the interface

Fixed-Size Array Implementation

- Recall the interface defines a generic type T for the objects in the bag

- `int getCurrentSize()`
- `boolean isEmpty`
- `boolean addNewEntry(T newEntry)`
- `T remove()`
- `boolean remove(T anEntry)`
- `void clear()`
- `int getFrequencyOf(T anEntry)`
- `boolean contains(T anEntry)`
- `T[] toArray`

- Same generic type `T` will be used in the definition of `ArrayBag` class
- Definition for the class will be fairly involved (lots of methods), so should not define the entire class then attempt to test it
- Instead, identify a group of core methods to first implement and test

Core Methods

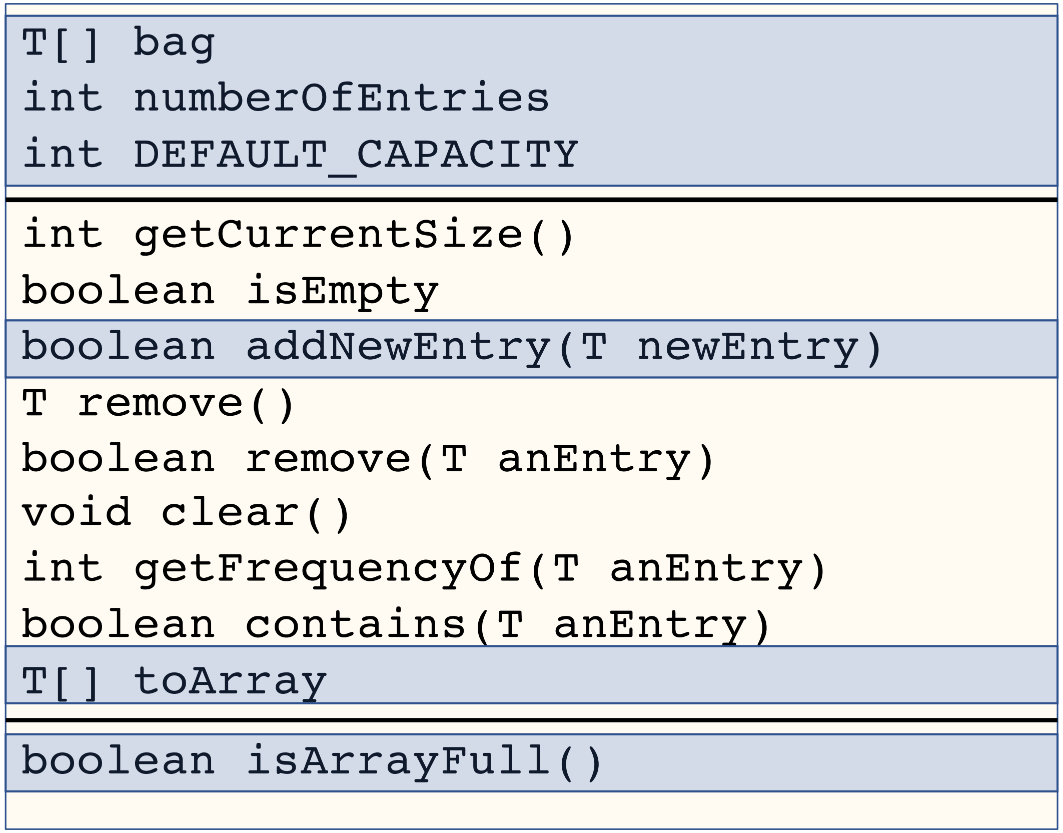
- When dealing with a collection such as a bag, you cannot test most methods until you have created the collection

- `int getCurrentSize()`
- `boolean isEmpty`
- `boolean addNewEntry(T newEntry)`
- `T remove()`
- `boolean remove(T anEntry)`
- `void clear()`
- `int getFrequencyOf(T anEntry)`
- `boolean contains(T anEntry)`
- `T[] toArray`

- If the `add()` method doesn’t work, then you can’t test the `remove()` methods
- To see if `add()` works you need a method that shows the contents of the bag
- Constructors are also needed, along with methods used by core methods

- The set of core methods should allow you to construct a bag, add objects to the bag, and look at the result

Core Methods



```
T[] bag
int numberOfEntries
int DEFAULT_CAPACITY

int getCurrentSize()
boolean isEmpty
boolean addNewEntry(T newEntry)
T remove()
boolean remove(T anEntry)
void clear()
int getFrequencyOf(T anEntry)
boolean contains(T anEntry)
T[] toArray

boolean isArrayFull()
```

- We identify `addNewEntry()` and its helper method `isArrayFull()` as core methods – along with `toArray()`
- These provide the minimum facilities to test a working application
- Also, identified an array to hold objects, the current number of entries and the length of the array as essential data components

Defining the Constructor

- Our constructor has 3 tasks
    1. Specify the array length
    2. Create the array
    3. Initialise the current number of entries to zero (initially the bag is empty)

- The decision to use a generic data type T in the declaration of the bag affects how we will allocate this array within the constructor

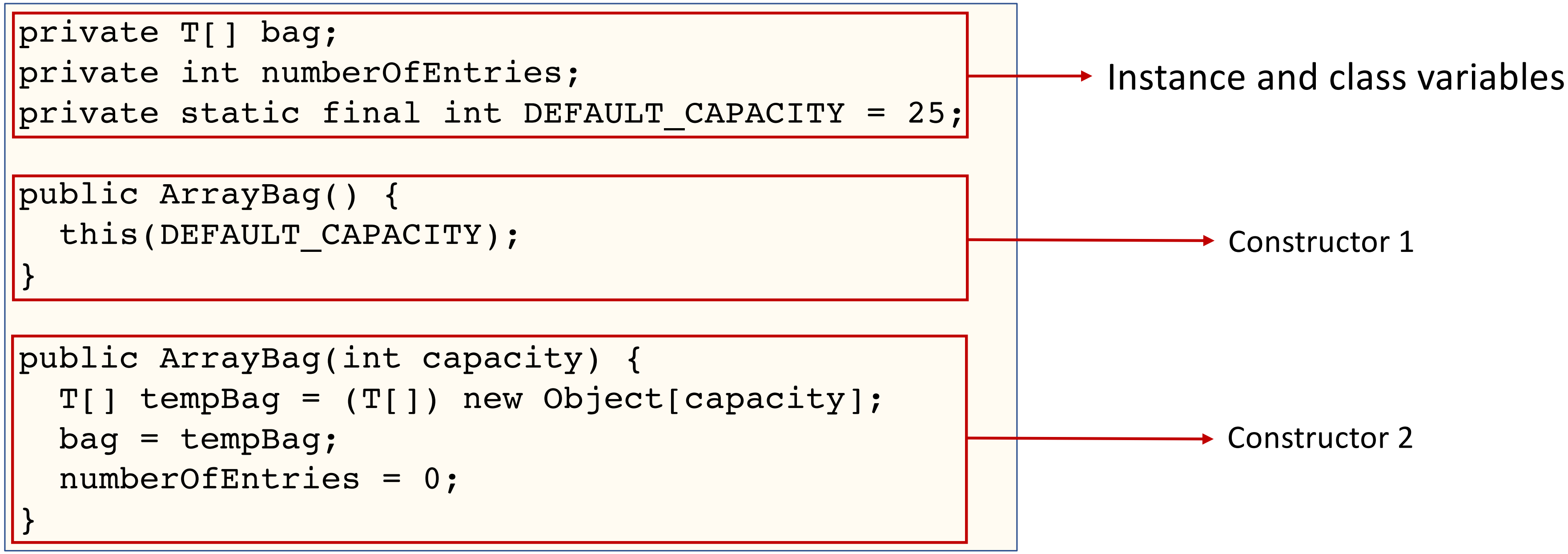
```
bag = new T[capacity]; // would produce a syntax error
```

- Need to declare an array of type Object and cast it to the desired type

```
T[] tempBag = (T[])new Object[capacity];
bag = tempBag;
```

Flexible Constructor

- Since our constructor needs to specify the length of the array (max size of the bag)
  - A good approach is to provide 2 versions - one that sets a default size if the client expresses no preference and another to set a size specified by the client



```
private T[] bag;
private int numberOfEntries;
private static final int DEFAULT_CAPACITY = 25;

public ArrayBag() {
  this(DEFAULT_CAPACITY);
}

public ArrayBag(int capacity) {
  T[] tempBag = (T[]) new Object[capacity];
  bag = tempBag;
  numberOfEntries = 0;
}
```

Instance and class variables

Constructor 1

Constructor 2

Scenario

- Implement a skeleton for the class ArrayBag.
  - Provide the class header to implement the BagInterface previously defined and populate it with the definition of the class and instance variables, the constructors and empty methods for each of the public methods defined in the interface class.

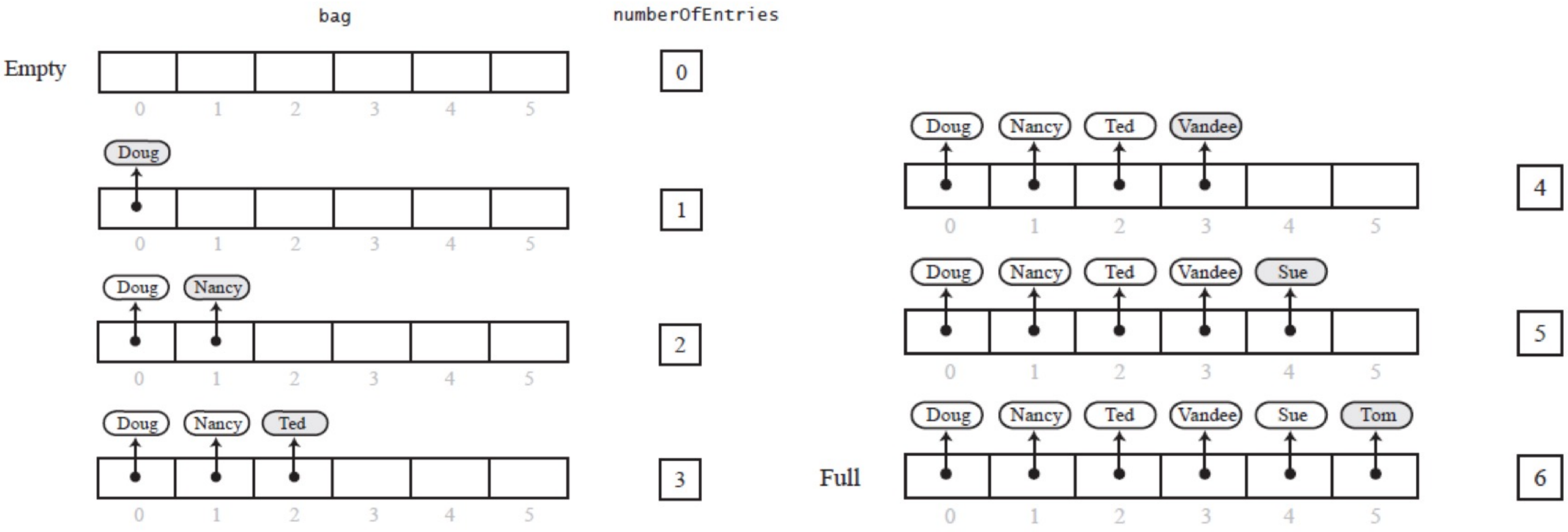
Core Methods – add()

```
Algorithm add (newEntry)
// Add a new entry into the bag, returning true if space available and false otherwise

if the array is full
   return false
else add the newEntry into the array at the position pointed at by numberOfEntries
   increment numberOfEntries
   return true
```

```
Algorithm isArrayFull ()
// Add a new entry into the bag, returning true if space available and false otherwise

if numberOfEntries equals the size of the bag array
   return true
else return false
```

Core Methods – add()

- Adding entries to an array that represents a bag with capacity 6 (until full):



- After each addition to the bag, increment the data field `numberOfEntries`

Core Methods – toArray()

- Retrieves the entries in the bag and returns them in a newly allocated array
  - The length of the new array can equal the number of entries in the bag (rather than the capacity of the bag)
  - However, we have the same problems in allocating an array that we had when implementing the constructors (so use the same approach)
- After `toArray()` creates the new array…
  - a loop can be used to copy the references in the array bag to the new array before returning it
  - OR use the built in Java method `System.arraycopy()`

Scenario

- Add the core methods to the class ArrayBag. Test your implementation by adding a `main()` method to the ArrayBag class that…

    i. Creates a new ArrayBag with capacity 5 to hold a collection of names as ArrayBag objects

    ii. Adds three names into the bag, confirming for each that the value returned by the `addNewEntry()` method is `true`

    iii. Uses `toArray()` to return the bag of names as an array and print its contents

    iv. Adds three more names into the bag, again reporting the value returned by the `addNewEntry()` method. Note that the value returned for the final `addNewEntry()` should be `false`, as the bag will have reached capacity

    v. Uses `toArray()` to return the bag of names as an array and prints its contents, ensuring that only the first 5 names were added.

Securing the Implementation

- Programmers must include fail-safe measures in their code to make programs secure for their users

- Java manages memory for you, checks validity of array indices, and is type-safe
- However, a mistake can make code vulnerable
- You can practice fail-safe programming by including checks for anticipated errors

- When implementing an ADT, ask the following questions:
  1. What happens if a constructor does not execute completely? (i.e. it throws an exception or error before it completes its initialization)
  2. What might happen if a client tries to create a bag that exceeds a given limit?

Securing the Implementation

- For the ArrayBag class we want to guard against both these situations

- We begin by refining the implementation to make code more secure by adding two additional data fields to the class:

```
...
private T[] bag;
private int numberOfEntries;
private static final int DEFAULT_CAPACITY = 25;
private boolean initialised = false;
private static final int MAX_CAPACITY = 10000;
...
```

New class variables

- To ensure a client cannot create a bag that is too large, the constructor should check the desired capacity (passed in as an argument) against MAX_CAPACITY

- If the request is too large, it can throw an exception

Securing the Implementation

- If the requested capacity is within range, the constructor could still fail (array allocation could fail due to insufficient memory)
- To prevent this, each vital method of the class can check the status field
- The constructor should set `initialised` to `true` for correctly initialised objects:

```
public ArrayBag(int capacity) {
   if (capacity <= MAX_CAPACITY) {
      T[] tempBag = (T[]) new Object[capacity];
      bag = tempBag;
      numberOfEntries = 0;
      initialised = true;
   } else throw new IllegalStateException(
                     "Attempt to create a bag where the capacity
                      exceeds the maximum");
}
```

Securing the Implementation

- Since `initialised` will be checked in several methods, repetitive code can be avoided by defining a private method:

```
private void checkInitialisation() {
   if (!initialised)
      throw new SecurityException(
                "ArrayBag object is not intialised properly");
}
```

- The method can be called by other methods in the class prior to performing their operations

```
public boolean addNewEntry(T newEntry) {
   checkInitialisation();
   ...
```

- Note: the exceptions `SecurityException` and `IllegalStateException` are standard runtime exceptions (in package `java.lang`.)

Securing the Implementation

- Some common guidelines for writing more secure Java code:
  1. Declare most (if not all) data fields of a class as private (any public data fields should be static and final and have constant values)
  2. Don't be too clever!
  3. Avoid duplicate code; instead, encapsulate such code into a private method that other methods can call
  4. When a constructor calls a method, ensure it cannot be overridden

- As such, we revise the declaration for class ArrayBag:

```
public final class ArrayBag<T> implements BagInterface<T> {
```

- As it is declared as final no other class can extend it
- A final class is more secure because it cannot be inherited to change its behaviour

Scenario

- Secure your implementation of the ArrayBag class by adding the following

  i. Add the `initialised` and `MAX_CAPACITY` variables

  ii. Update the constructor to check that the requested capacity is allowed and throw an exception if not. If the Bag is created, the `initialised` class variable should be set to `true`

  iii. Add the new private method `checkInitialisation()` to test the value of `initialised` and throw an exception if it is `false`

  iv. Call the new `checkInitialisation()` method as the first action in the `addNewEntry()` method

  v. Update the class header to define the ArrayBag class as `final`

- Check that your implementation still works as before following these changes

Implementing More Methods

- Once the core methods are successfully implemented and tested the remaining methods can be defined (starting with the easiest ones)

```
public int getCurrentSize() {
   return numberOfEntries;
}

public boolean isEmpty() {
   return numberOfEntries == 0;
}
```

- No need to perform unnecessary security checks
- Even if constructor is not complete, Java will initialize numberOfEntries to zero by default (a partially initialized bag will appear empty)

- Methods that access the array `bag` should always ensure it exists first

Implementing More Methods

`int getFrequencyOf(T anEntry)`

```
Algorithm getFrequencyOf(anEntry)
// returns the number of times that anEntry appears in the bag

set counter to zero
for array elements from position 0 to position numberOfEntries – 1
   if current array element equals anEntry add 1 to counter
end for
return counter
```

- To do the comparison, the `equals()` method must be used (instead of ==)

  `anEntry.equals(bag[index])` NOT `anEntry == bag[index]`

- We assume the class to which the objects belong defines its own version of `equals()`

Implementing More Methods

`boolean contains(T anEntry)`

```
Algorithm contains(anEntry)
// returns true if anEntry appears in the bag, false otherwise

set found to false
set index to 0
while not found and there are more elements to check
   if array element at index equals newEntry set found to true
   increment index
end while
return found
```

- Similar to `getFrequencyOf()` but search can stop when the first match is found

Methods That Remove Entries

- Two of the methods to remove entries are straightforward to define:

```
public void clear() {
   while(!isEmpty()) remove();
}
```

- Keep removing objects until the bag is empty

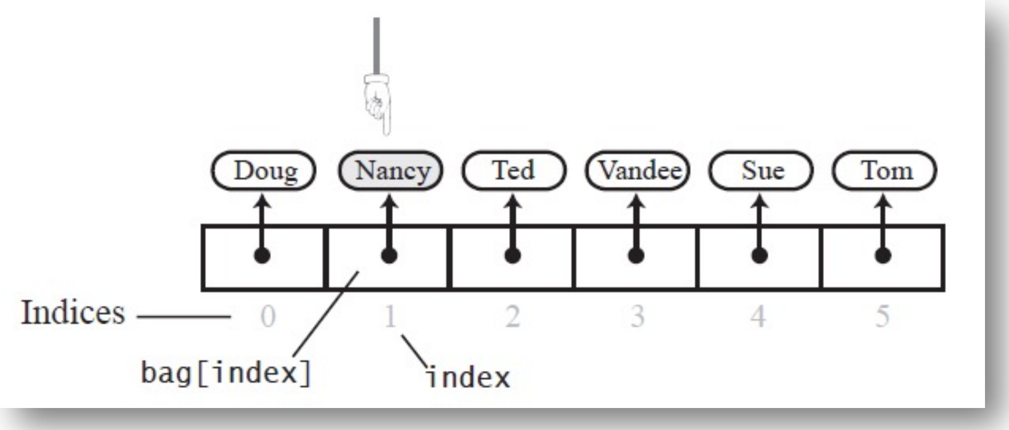
```
public T remove() {
   checkInitialisation();
   T result = null;
   if (numberOfEntries > 0) {
      result = bag[numberOfEntries – 1];
      bag[numberOfEntries – 1] = null;
      numberOfEntries--;
   }
   return result;
}
```

- First version – we will return to this later!
- Doesn’t matter which object is removed, so remove the easiest one
- Setting entry to `null` flags it for Java garbage collection

Methods That Remove Entries

- ***“Remove an occurrence of a particular object from the bag”*** described as:
  `T remove(entry)`
- If an entry occurs more than once we simply remove the first occurrence

- Assuming the bag is not empty:
  - Search the array bag
  - If *anEntry* equals bag[index] then note the value of index
- Now remove the entry at the index

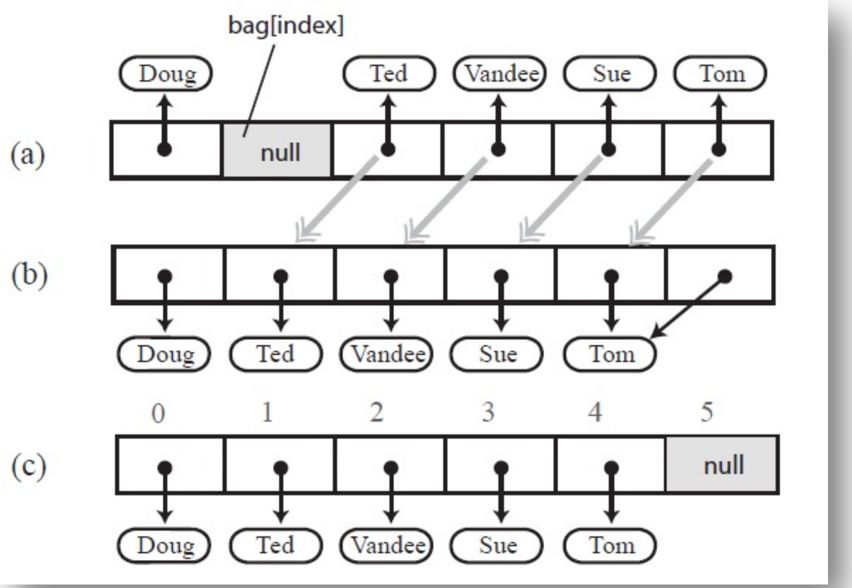


Methods That Remove Entries

- Setting the entry in the array to null will remove the reference to the entry, but it will leave a gap in the array (bag no longer stored in consecutive array locations)

- Could get rid of the gap by shifting the following entries and replacing the duplicate reference to the last entry with null:

  a) A gap in the array bag
  b) Array after shifting entries (after the gap)
  c) Replace duplicate reference to the last entry with null



Methods That Remove Entries

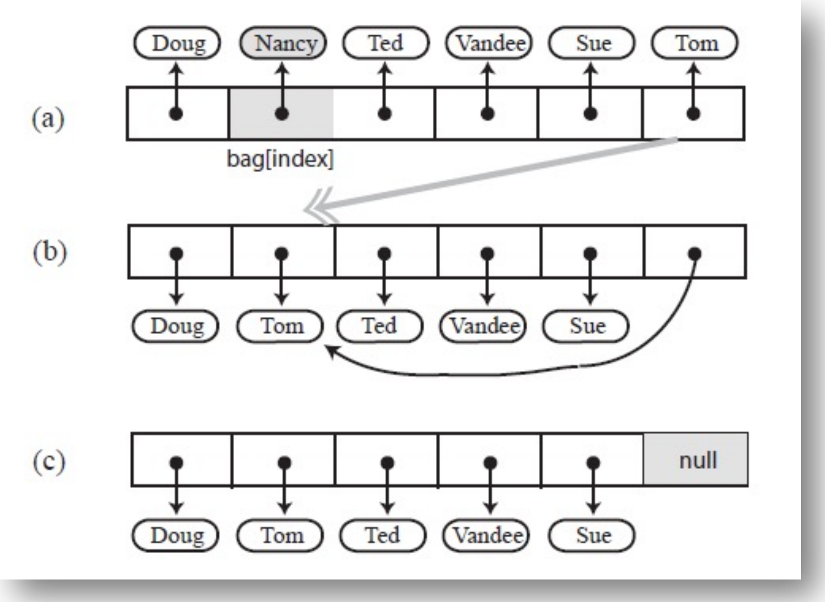
- Previous approach would work but is somewhat time consuming!
- Remember that we aren’t required to maintain any particular order for the elements

- Instead, we can replace the entry being removed with the last entry in the array:

  a) Locate the entry to be removed

  b) Copy the last entry in the array to the index of the entry to be removed

  c) Replace the last entry in the array with null



Methods That Remove Entries

- Both `remove()` and `remove(anElement)` can make use of a private helper method to remove and return an element from a specific position

```
private T removeElementAt(int index) {
   T result = null;
   int lastIndex = numberOfEntries - 1
   checkInitialisation()

   if (!isEmpty() && index >= 0 && index <= lastIndex) {
      result = bag[index];
      bag[index] = bag[lastIndex];
      bag[lastIndex] = null;
      numberOfEntries--;
   }
   return result;
}
```

Methods That Remove Entries

- Both `remove()` and `remove(anEntry)` can make use of a private helper method to remove and return an element from a specific position

```
 public T remove () {
    return removeElementAt(numberOfEntries – 1);
}
```

```
 public boolean remove(T anEntry) {
    boolean found = false;
    int index = 0;
    while (!found && index < numberOfEntries) {
       if (bag[index].equals(anEntry)) found = true;
       else index++;
    }
    if (found) removeElementAt(index);
    return found;
}
```

Scenario

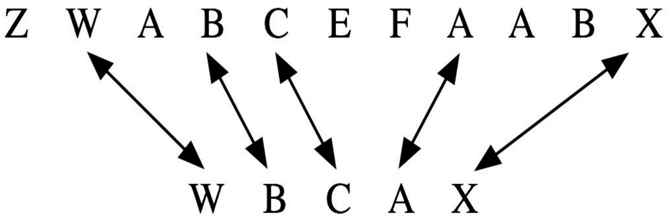
- Add the remaining methods to your `ArrayBag` class. All public methods outlined in the interface class should now be implemented.
  - Test your implementation by adding the file ***BagTest.java*** to your **Bag** project and providing suitable code at the locations marked as `// TODO`
  - Run the `main()` method in ***BagTest.java*** and trace the diagnostic comments provided to ensure that your `ArrayBag` implementation is working as expected.

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
- A fixed-size array is limited in its capacity
- Approaches to dynamically increase the size of the array are possible (but increasing the size of the array requires time to copy its entries)

Challenge

- **Longest Common Subsequence**
  - In the application created in this exercise, two strings of characters will be taken as input and the longest subsequence of characters common to both strings will be determined. We want to find the longest sequence of letters that is common between two strings.
  - For one string to be a subsequence of the other, all letters in the first string must match up uniquely with a letter in the second string.
  - The matches have to be in the same order, but they do not need to be consecutive. For example, “WBCAX” is a subsequence of “ZWABCEFAABX” as can be seen below


Z W A B C E F A A B X
W B C A X