Core ArrayBag Methods

Pearson

Implementing the ADT Bag

- **Steps to Follow**
  - Decide on Data Fields
  - Implement Constructors
    - Initialize the data fields
  - Implement Core Functions
    - Methods critical to collection functionality
    - Methods to check status of collection
    - Test Your Implementation
  - Implement Additional Methods
    - Test Your Implementation

```
template<class ItemType>
class BagInterface
{
public:
   /** Gets the current number of entries in this bag.
    @return the integer number of entries currently in the bag */
   virtual int getCurrentSize() const = 0;

   /** Sees whether this bag is empty.
    @return true if the bag is empty, or false if not */
   virtual bool isEmpty() const = 0;

   /** Adds a new entry to this bag.
    @post if successful, newEntry in stored in bag and
          count of items in the bag is increased by 1
    @param someItem  the object to be added as a new entry
    @return true if addition is successful, or false if not */
   virtual bool add(const ItemType& someItem) = 0;

   /** Removes one occurrence of a given entry from this bag,
       if possible.
    @post if successful, anEntry has been removed from the bag
          and the count of items in the bag has decreased by 1
    @param target  the entry to be removed
    @return true if removal was successful, or false if not */
   virtual bool remove(const ItemType& target) = 0;

   /** Removes all entries from this bag.
       @post bag contains no items and the count of items is 0 */
   virtual void clear() = 0;

   /** Counts the number of times a given entry appears in bag.
    @param target  the entry to be counted
    @return the number of times anEntry appears in the bag */
   virtual int getFrequencyOf(const ItemType& target) const = 0;

   /** Tests whether this bag contains a given entry.
    @param target  the entry to locate
    @return true if bag contains anEntry, or false otherwise */
   virtual bool contains(const ItemType& target) const = 0;
   /** Vector with of all entries that are in this bag.

    @param bagContents  a vector
    @post bagContents contains all the entries in the bag */
   virtual std::vector<ItemType> toVector() const = 0;

   /** Destroys object & frees memory allocated by object. */
   virtual ~BagInterface() { }
};  // end BagInterface
#endif
```

Pearson

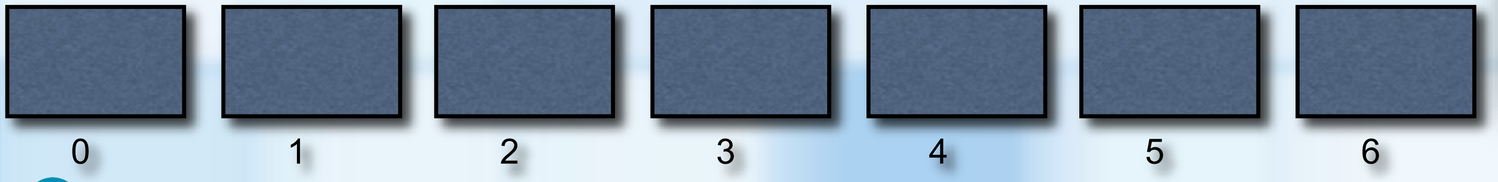
Decide on Data Fields

- **Implementation must store items**
  - Use an array of fixed size

Sumar
Maria
Ted
Jose
Nancy

Eventually the array will become full.

This limits the items our bag can hold.




```
template<class ItemType>
class BagInterface
{
public:
   /** Gets the current number of entries in this bag.
    @return the integer number of entries currently in the bag */
   virtual int getCurrentSize() const = 0;

   /** Sees whether this bag is empty.
```

```
template<class ItemType>
class BagInterface
{
public:
   virtual int getCurrentSize() const = 0;
   virtual bool isEmpty() const = 0;
   virtual bool add(const ItemType& someItem) = 0;
   virtual bool remove(const ItemType& target) = 0;
   virtual void clear() = 0;
   virtual int getFrequencyOf(const
                              ItemType& target) const = 0;
   virtual bool contains(const
                         ItemType& target) const = 0;
   virtual std::vector<ItemType> toVector() const = 0;
   virtual ~BagInterface() { }
};  // end BagInterface
```

```
    @return the number of times anEntry appears in the bag */
   virtual int getFrequencyOf(const ItemType& target) const = 0;

   /** Tests whether this bag contains a given entry.
    @param target  the entry to locate
    @return true if bag contains anEntry, or false otherwise */
   virtual bool contains(const ItemType& target) const = 0;
   /** Vector with of all entries that are in this bag.

    @param bagContents  a vector
    @post bagContents contains all the entries in the bag */
   virtual std::vector<ItemType> toVector() const = 0;

   /** Destroys object & frees memory allocated by object. */
   virtual ~BagInterface() { }
};  // end BagInterface
#endif
```

Pearson

Decide on Data Fields

- **Implementation must store items**
  - Use an array of fixed size
  - Default capacity for the bag
  - Current number of items in the bag
  - Maximum capacity of bag

```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType>
{
private:
   static const int DEFAULT_CAPACITY = 6;
   ItemType items[DEFAULT_CAPACITY]; // bag items
  int itemCount;                    // count of bag items
  int maxItems;                  // max capacity of the bag


public:
   ArrayBag();
   int getCurrentSize() const;
   bool isEmpty() const;
   bool add(const ItemType& someItem);
   bool remove(const ItemType& someItem);
   void clear();
   bool contains(const ItemType& target) const;
   int getFrequencyOf(const ItemType& target) const;
   std::vector<ItemType> toVector() const;
};  // end ArrayBag
```

Pearson

Implementing Constructors

- Must happen before other class methods can be called
- Ensure all data fields are initialized

```
template<class ItemType>
ArrayBag<ItemType>::ArrayBag()
    : itemCount(0), maxItems(DEFAULT_CAPACITY)
{
}  // end default constructor
```

Implementing Core Methods

- **Determine collection characteristics**
  - Number of items? Is the bag *empty*?
- **Place items into object**

```
template<class ItemType>
class BagInterface
{
public:
   virtual int getCurrentSize() const = 0;
   virtual bool isEmpty() const = 0;
   virtual bool add(const ItemType& someItem) = 0;
   virtual bool remove(const ItemType& someItem) = 0;
   virtual void clear() = 0;
   virtual int getFrequencyOf(const
                              ItemType& target) const = 0;
   virtual bool contains(const
                              ItemType& target) const = 0;
   virtual std::vector<ItemType> toVector() const = 0;
   virtual ~BagInterface() { }
};  // end BagInterface
```

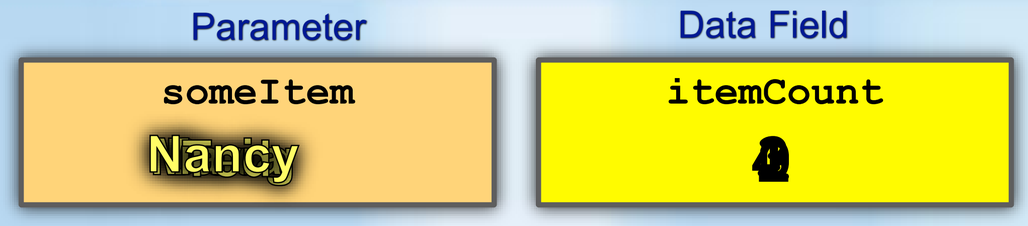
```
template<class ItemType>
int ArrayBag<ItemType>::getCurrentSize() const
{
   return itemCount;
}  // end getCurrentSize
```

```
template<class ItemType>
bool ArrayBag<ItemType>::isEmpty() const
{
   return itemCount == 0;
}  // end isEmpty
```

Pearson

Implementing Core Methods

- **Determine collection characteristics**
  - Number of items? Is the bag ***empty***?
- **Place items into object**
  - Start at first element




```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& someItem)
{
   bool hasRoomToAdd = (itemCount < maxItems);
   if (hasRoomToAdd)
   {
      items[itemCount] = someItem;
      itemCount++;
   }  // end if

   return hasRoomToAdd;
}  // end add
```

Pearson

Implementing Core Methods

- **Determine collection characteristics**
  - Number of items? Is the bag *empty*?
- **Place items into object**
  - Start at first element
- **Report on items in object**
  - Allows us to determine if the items were added properly.
  - Should it return the array or a copy?
    - Returning a copy keeps data **private**
    - Helps to prevent data being accidentally corrupted

```
template<class ItemType>
std::vector<ItemType> ArrayBag<ItemType>::toVector() const
{
    std::vector<ItemType> bagContents;
    for (int i = 0; i < itemCount; i++)
       bagContents.push_back(items[i]);

  return bagContents;
}  // end toVector
```

Other ArrayBag Methods

Test Core Methods

- **Must implement all interface methods**

```
template<class ItemType>
int ArrayBag<ItemType>::getFrequencyOf(const ItemType& target) const
{
  int frequency = 0;
  int loopCounter;
  for (loopCounter = 0; loopCounter < currentSize; loopCounter++)
    if (target == items[loopCounter])
      frequency++;
  return frequency;
} // end getFrequencyOf

template<class ItemType>
bool ArrayBag<ItemType>::contains(const ItemType& target) const
{
  int loopCounter;
  for (loopCounter = 0; (loopCounter < currentSize) && !found; loopCounter++)
    found = (target == items[loopCounter]);
  return found;
} // end contains
```

```
template<class ItemType>
bool ArrayBag<ItemType>::remove(const ItemType& target)
{
  int loopCounter = 0;
  bool found = false;
  while (!found && (loopCounter < currentSize))
  {
    if (target == items[loopCounter])
    {
      found = true;
      items[loopCounter] = items[currentSize - 1];
      currentSize--;
    }
    loopCounter++;
  }
  return found;
} // end remove

template<class ItemType>
void ArrayBag<ItemType>::clear()
{
  currentSize = 0;
} // end clear
// private
template<class ItemType>
int ArrayBag<ItemType>::getIndexOf(const ItemType& target) const
{
  int result = -1;
  return result;
} // end getIndexOf
```

Test Core Methods

- **Must implement all interface methods**
  - Stub other methods
- **Test constructor**
  - Create an ArrayBag
  - Validate the bag is empty
- **Add items**
  - Validate that the items are in the bag
- **Fill bag**
  - Validate the bag is full
  - Validate additional adds fail

```
  ArrayBag<std::string> bag;
std::cout << "isEmpty: returns " << bag.isEmpty()
                          << "; should be 1 (true)" << std::endl;
std::cout << "The bag contains " << bag.getCurrentSize()
     << " items:" << std::endl;

std::vector<std::string> bagItems = bag.toVector();
int numberOfEntries = bagItems.size();
for (int i = 0; i < numberOfEntries; i++)
{
   std::cout << bagItems[i] << " ";
}  // end for

  std::string items[] = {"one", "two", "three",
                                "four", "five", "one"};

std::cout << "Add 6 items to the bag: " << std::endl;
  for (int i = 0; i < 6; i++)
{
       bag.add(items[i]);
  }  // end for

  std::cout << "The bag contains " << bag.getCurrentSize()
     << " items:" << std::endl;

bagItems = bag.toVector(bagItems);
numberOfEntries = bagItems.size();
for (int i = 0; i < numberOfEntries; i++)
{
   std::cout << bagItems[i] << " ";
}  // end for
```

Pearson

Additional Methods

- **Additional status methods**
  - Status of collection
  - Status of an item

```
template<class ItemType>
class BagInterface
{
public:
   virtual int getCurrentSize() const = 0;
   virtual bool isEmpty() const = 0;
   virtual bool add(const ItemType& someItem) = 0;
   virtual bool remove(const ItemType& target) = 0;
   virtual void clear() = 0;
   virtual int getFrequencyOf(const
                              ItemType& target) const = 0;
   virtual bool contains(const
                             ItemType& target) const = 0;
   virtual vector<ItemType> toVector() const = 0;
   virtual ~BagInterface() { }
};  // end BagInterface
```

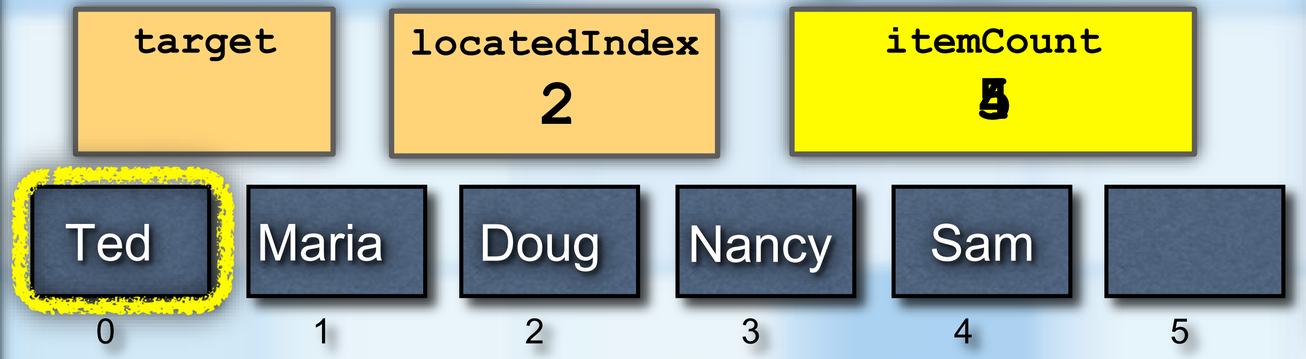
```
template<class ItemType>
int ArrayBag<ItemType>::getFrequencyOf(const
                                       ItemType& target) const
{
  int frequency = 0;
  int curIndex = 0;       // Current array index
  while (curIndex < itemCount)
  {
    if (items[curIndex] == target)
    {
      frequency++;
    }  // end if

    curIndex++;           // Increment to next entry
  }  // end while

  return frequency;
}  // end getFrequencyOf
```

```
template<class ItemType>
bool ArrayBag<ItemType>::contains(const
                                  ItemType& target) const
{
   return getIndexOf(target) > -1;
}  // end contains
```

Pearson

Additional Methods

- **Additional status methods**
  - Status of collection
  - Status of an item
- **Removing items from the collection**
  - All items
  - A specific item




```
template<class ItemType>
bool ArrayBag<ItemType>::remove(const ItemType& target)
{
  int locatedIndex = getIndexOf(target);
    bool canRemoveItem = !isEmpty() && (locatedIndex > -1);
    if (canRemoveItem)
    {
       items[locatedIndex] = items[itemCount-1];
       itemCount--;
    }  // end if

    return canRemoveItem;
}  // end remove
```

Pearson

Additional Methods

- **Additional status methods**
  - Status of collection
  - Status of an item
- **Removing items from the collection**
  - All items
  - A specific item

```
// private
template<class ItemType>
int ArrayBag<ItemType>::
          getIndexOf(const ItemType& target) const
{
    bool found = false;
  int result = -1;
  int searchIndex = 0;

  while (!found && (searchIndex < itemCount))
  {
    if (items[searchIndex] == target)
    {
      found = true;
      result = searchIndex;
    }
    else
    {
      searchIndex++;
    }  // end if
  }  // end while

  return result;
}  // end getIndexOf
```

Pearson