CS302 - Data Structures

using C++

Topic: Code ArrayBag Methods

Kostas Alexis

AUTONOMOUS ROBOTS LAB

Implementing the ADT Bag

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

```
// Start with myArray[first]
void displayArray(int myArray[], int first, int last)
{
     std::cout << myArray[first] << " ";
     if (first < last)
          displayArray(myArray, first + 1, last);
} // end displayArray
```

Implementing the ADT Bag

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

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

Decide on Data Fields

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

Doug
Maria
Ted
Jose
Nancy

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

Decide on Data Fields

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

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

Decide on Data Fields

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

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

Decide on Data Fields

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

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

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 bag capacity

UML Notation

ArrayBag

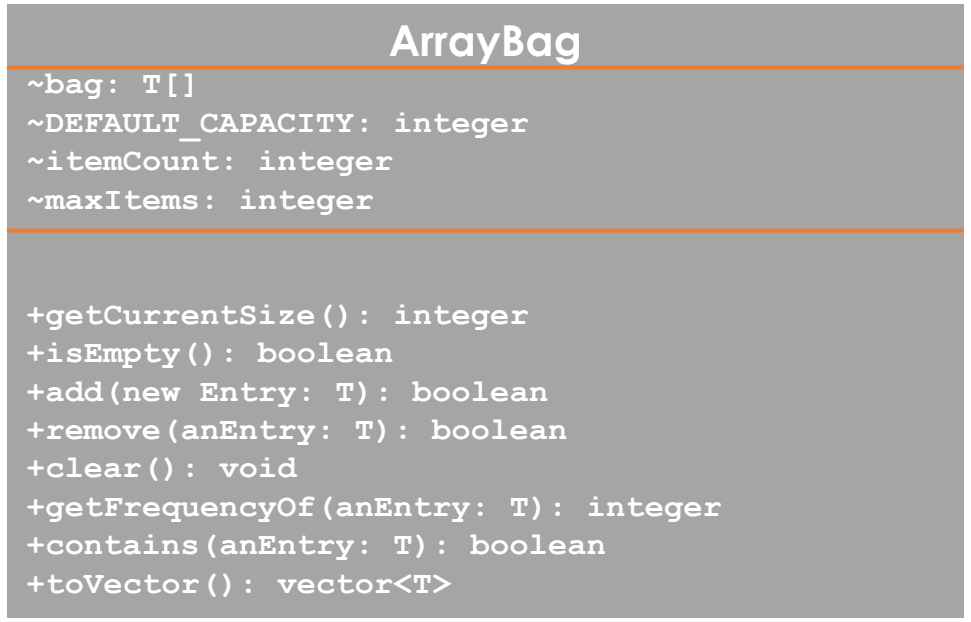
```
~bag: T[]
~DEFAULT_CAPACITY: integer
~itemCount: integer
~maxItems: integer
```

```
+getCurrentSize(): integer
+isEmpty(): boolean
+add(new Entry: T): boolean
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toVector(): vector<T>
```

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 bag capacity

UML Notation




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

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_BAG_SIZE)
{
} // end default constructor
```

Implementing Core Methods

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

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

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

Implementing Core Methods

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

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

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

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

Implementing Core Methods

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

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

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

Implementing Core Methods

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

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

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

```
template<class ItemType>
bool ArrayBag<ItemType>::add(const ItemType& newEntry)
{
     bool hasRoomToAdd = false;

      return hasRoomToAdd
} // getCurrentSize
```

stub

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
```

Thank you