

Data Structures

Array-Based Implementations

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **The ADT bag.**
- Implementation.
- Recursion and DMA.

The ADT “Bag”

- We want to implement an ADT “Bag” of items whose types are “ItemType:”
 - `getCurrentSize(): integer`
 - `isEmpty(): boolean`
 - `add(newEntry: ItemType): boolean // duplications are allowed`
 - `remove(anEntry: ItemType): boolean // remove only one copy`
 - `clear(): void`
 - `getFrequencyOf(anEntry: ItemType): integer`
 - `contains(anEntry: ItemType): boolean`
 - `print(): void`
- Specifications indicate **what** the operations do, **not how** to implement
- Let’s begin with a bag of **C++ strings**.

Implementing the ADT “Bag”

- Implementing the ADT as a **C++ class** provides ways to enforce a wall:
 - This prevents one to access the data without using the defined operations.

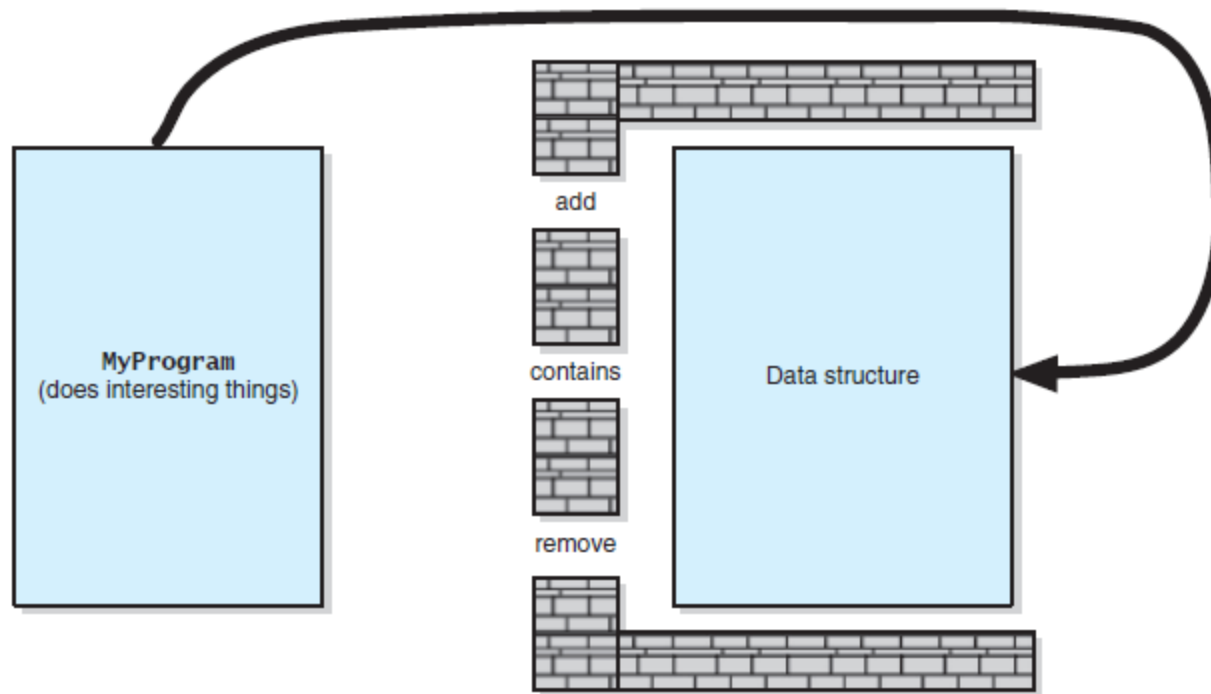


Figure 3-1 (Carrano and Henry, 2015)

Implementing the ADT “Bag”

- The abstract class **BagInterface** defines the **behaviors** of the ADT.
 - In many cases, **no member variable** is needed for the abstract class.

```
class BagInterface
{
public:
    virtual int getCurrentSize() const = 0;
    virtual bool isEmpty() const = 0;
    virtual bool add(const string& newEntry) = 0;
    virtual bool remove(const string& anEntry) = 0;
    virtual void clear() = 0;
    virtual int getFrequencyOf(const string& anEntry) const = 0;
    virtual bool contains(const string& anEntry) const = 0;
    virtual void print() const = 0;
};
```

Implementing the ADT “Bag”

- To implement the ADT, we write a (concrete) class to inherit **BagInterface** and **implement all the operations.**

```
class ArrayBag : public BagInterface
{
private:
    // ...
public:
    ArrayBag();
    int getCurrentSize() const;
    bool isEmpty() const;
    bool add(const string& newEntry);
    bool remove(const string& anEntry);
    void clear();
    bool contains(const string& anEntry) const;
    int getFrequencyOf(const string& anEntry) const;
    void print() const;
};
```

Implementing the ADT “Bag”

- Now it is time to choose a **data structure**.
- For the ADT bag, we will try two different data structures:
 - An **array** and a **linked list**.
- Let’s use a **fixed-size (static)** array first.
 - In a fixed-size array, each item occupies one entry of the array.
 - These items are **unsorted**. A **one-dimensional unsorted array** will be used.

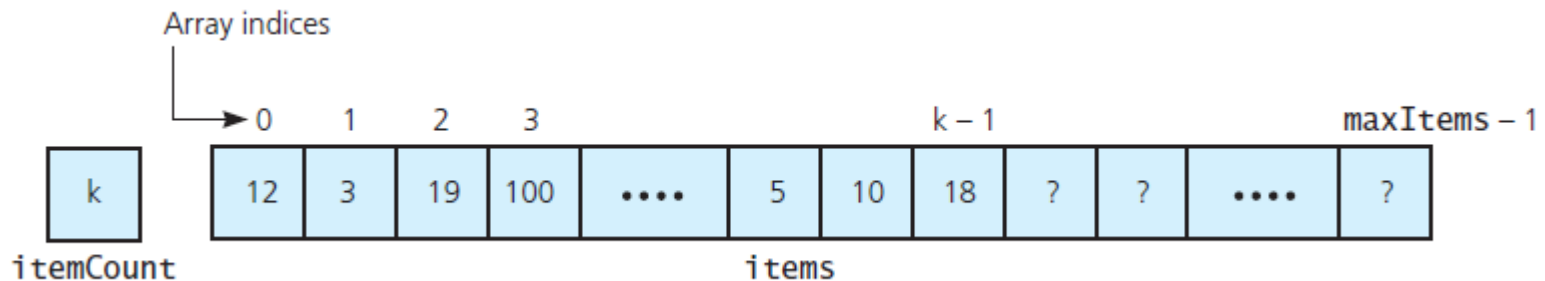


Figure 3-2 (Carrano and Henry, 2015)

Outline

- The ADT bag.
- **Implementation.**
- Recursion and DMA.

Using a fixed-size array

- What else should we keep track of when we use a fixed array?
 - The **maximum size** of the array.
 - The **number of items** currently stored.
- We add **member variable declarations** into the private section.

```
class ArrayBag : public BagInterface
{
private:
    static const int DEFAULT_CAPACITY = 6;
    string items[DEFAULT_CAPACITY];
    int itemCount;
    int maxItems;
    // may have something else...
public:
    // those member functions...
};
```

Implementing member functions

- Some member functions are pretty straightforward.
 - Note that these should be **constant** member functions.
- How would you add a constructor to choose a maximum number of items?

```
ArrayBag::ArrayBag()  
    : itemCount(0),  
      maxItems(DEFAULT_CAPACITY) {}  
int ArrayBag::getCurrentSize() const  
{  
    return itemCount;  
}  
bool ArrayBag::isEmpty() const  
{  
    return itemCount == 0;  
}  
void ArrayBag::print() const  
{  
    for(int i = 0; i < itemCount; i++)  
        cout << items[i] << " ";  
    cout << endl;  
}
```

The member function `add()`

- For adding an item:
 - If there is a room, store the item somewhere the array (where?)
 - Return true if there is a room or false otherwise.
- Let's put it in the **first empty cell**.
 - If the array has no “hole,” its index is exactly `itemCount`.
 - Otherwise, we may need to search from the beginning.

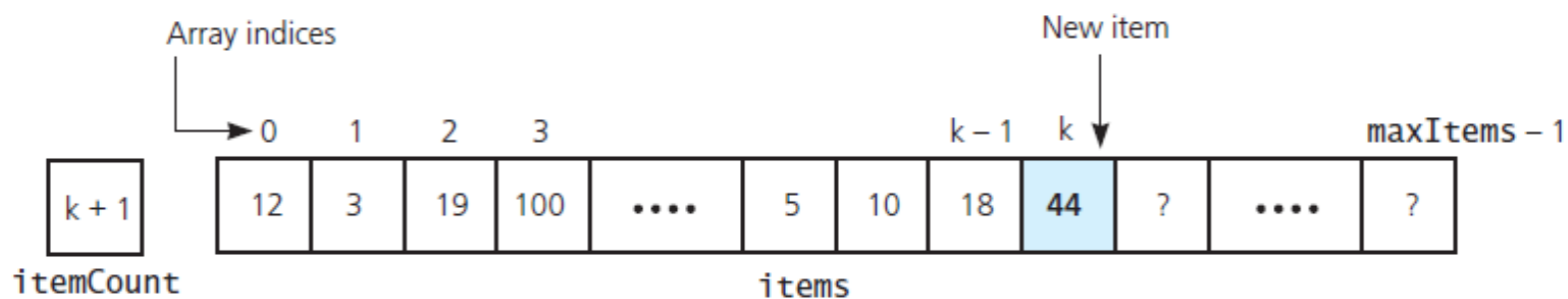


Figure 3-3 (Carrano and Henry, 2015): **After** adding an item

The member function `add()`

- As long as the array has no hole, adding an item is easy.
 - When we remove an item, we need to **maintain this property!**

```
bool ArrayBag::add(const string& newEntry)
{
    bool hasRoomToAdd = (itemCount < maxItems);
    if (hasRoomToAdd)
    {
        items[itemCount] = newEntry;
        itemCount++;
    }
    return hasRoomToAdd;
}
```

- Why using a **constant reference**? Why not just `string newEntry`?

Testing your implementations

- Before we try to implement something more, **test** your existing member functions.
- Once you have your **add()** function, you may test the constructor, **isEmpty()**, **getCurrentSize()**, and **print()**.
- In fact, **print()** is to test **add()** and other member functions.

```
int main()
{
    ArrayBag bag;
    bag.add("aaa");
    bag.print();
    cout << bag.isEmpty();
    cout << bag.getCurrentSize();
    bag.add("bbb");
    bag.print();
    cout << bag.isEmpty();
    cout << bag.getCurrentSize();

    return 0;
}
```

The member function `remove()`

- For removing a given item:
 - Remove one copy if it exists in the array.
 - Return true if the item exists or false otherwise.
- Note that we need to determine whether a given item exists.
 - This is exactly the function **`contains()`**.
 - Moreover, we need to **locate** that item.
- We may enhance **modularity** (and make the development more efficient) by implementing a member function to be a **building block**.

The member function `getIndexOf()`

- Let's implement a member function `getIndexOf()`.
 - `getIndexOf(anEntry: ItemType): integer`
 - Given an item, return the index of **its first copy** in the array or **-1** otherwise.

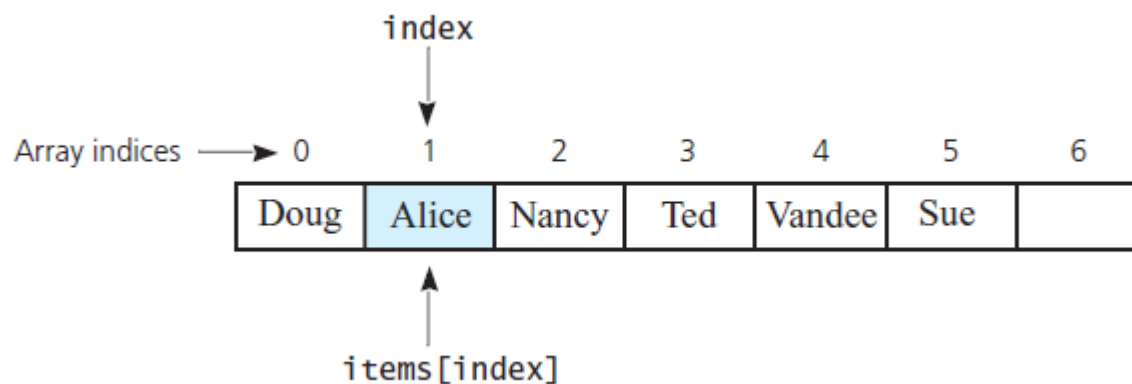


Figure 3-4 (Carrano and Henry, 2015)

- Define things **precisely**: Returning an array **index** or a **rank** of an item?

Implementation of getIndexOf ()

```
int ArrayBag::getIndexOf(const string& 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++;
    }
    return result;
}
```


Privatizing `getIndexOf ()`

- The function `getIndexOf ()` should be **private**!
- If it is public, clients will know some **details** of the private array.
 - Data hiding (encapsulation, the “wall”) would be damaged.
- As long as one thing is **useless** to clients, hide it!

```
class ArrayBag : public BagInterface
{
private:
    static const int DEFAULT_CAPACITY = 6;
    string items[DEFAULT_CAPACITY];
    int itemCount;
    int maxItems;
    int getIndexOf(const string& target) const;
public:
    // those member functions...
};
```

The member function `remove()`

- With `getIndexOf()`, `remove()` can be easily implemented.
- The pseudocode:
- How to “remove the item while ensuring no hole?”

```
remove(anEntry)
{
  Search the array items for anEntry
  if(anEntry is in the bag at items[index])
  {
    Decrement the counter itemCount
    remove the item while ensuring no hole
    return true
  }
  else
    return false
}
```

Ensuring no hold (idea 1)

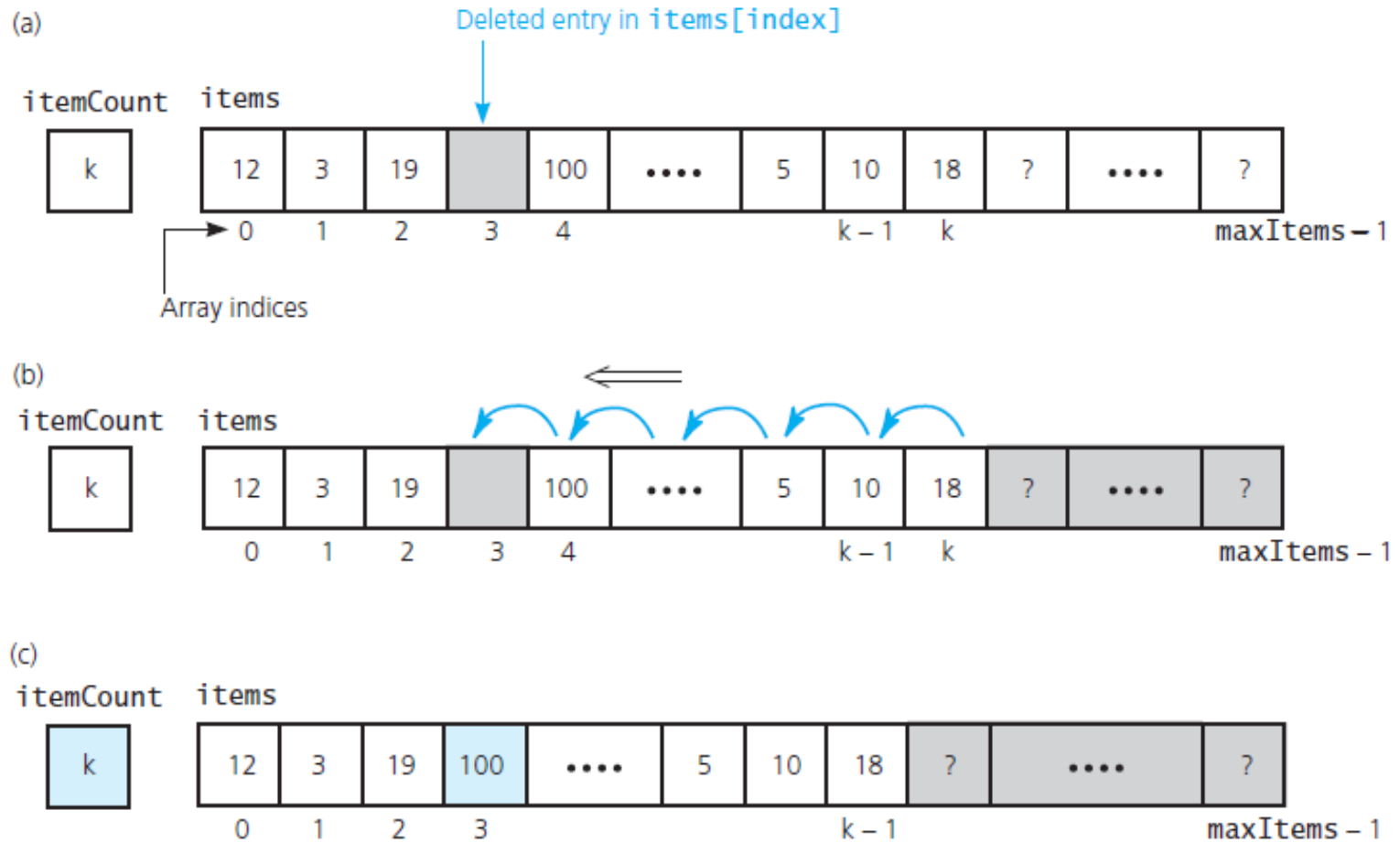


Figure 3-5 (Carrano and Henry, 2015)

Ensuring no hold (idea 2)

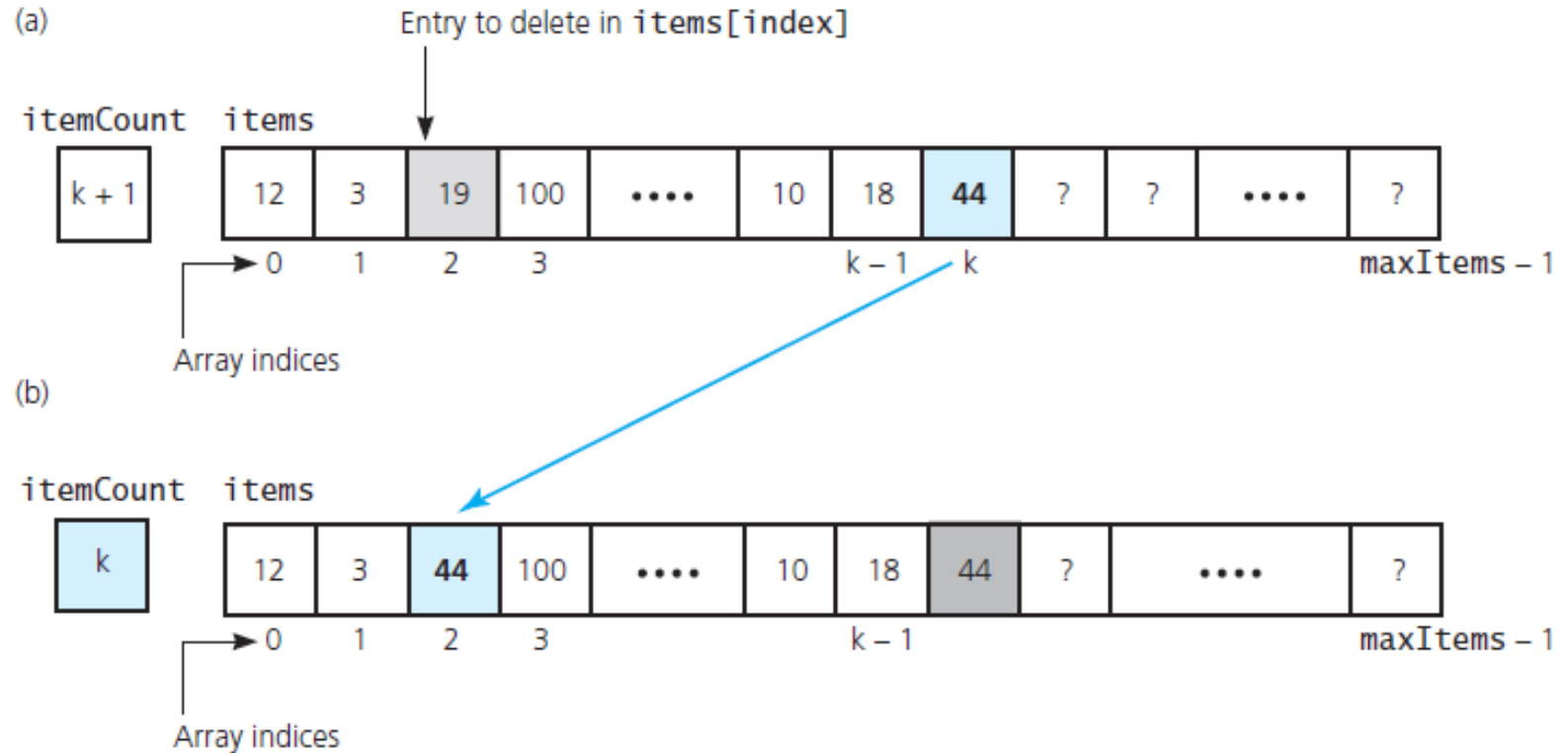


Figure 3-6 (Carrano and Henry, 2015)

Implementing `remove()`

```
bool ArrayBag::remove(const string& anEntry)
{
    int locatedIndex = getIndexOf(anEntry);
    bool canRemoveItem = (locatedIndex > -1);
    if(canRemoveItem)
    {
        itemCount--;
        items[locatedIndex] = items[itemCount];
    }
    return canRemoveItem;
}
```

```
remove(anEntry)
{
    Search the array items for anEntry
    if(anEntry is in the bag at items[index])
    {
        Decrement the counter itemCount
        remove the item while ensuring no hole
        return true
    }
    else
        return false
}
```

Functions `contains()` and `clear()`

- The functions `contains()` is straightforward.

```
bool ArrayBag::contains(const string& anEntry) const
{
    return getIndexof(anEntry) > -1;
}
```

- How about `clear()`? Which one is better?

```
void ArrayBag::clear()
{
    while(!isEmpty())
        remove(items[1]);
}
```

```
void ArrayBag::clear()
{
    itemCount = 0;
}
```

Member function `getFrequencyOf()`

- `getFrequencyOf()` returns the number of occurrences of a given item.

```
int ArrayBag::getFrequencyOf(const string& anEntry) const
{
    int frequency = 0;
    int curIndex = 0;
    while (curIndex < itemCount)
    {
        if (items[curIndex] == anEntry)
            frequency++;
        curIndex++;
    }
    return frequency;
}
```

Remarks

- The array-based implementation of the ADT bag has been completed!
- Before we start the implementation, always “**design**” your program first.
 - Design the ADT. In particular, design the operations and behaviors.
 - Write an **abstract class** to specify the operations.
 - Write a **concrete class** to inherit the abstract class. Implement the class.
 - For a specific function, **pseudocodes** are helpful.
 - **Test** your implementation.
- Do not forget **data hiding**:
 - If something is not useful for clients, hide it.
 - E.g., do not let clients know whether you starts at `items[0]` or `items[1]`.

Outline

- The ADT bag.
- Implementation.
- **Recursion and DMA.**

Recursion

- Though not required, some functions can be implemented with recursion.
- In many cases, a task that need to **go through a data structure** can be implemented with recursion.
- Let's see two examples.

Using recursion for `getIndexOf()`

```
int ArrayBag::getIndexOf(const string& target, int searchIndex) const
{ // search within items[searchIndex..(itemCount - 1)]
  int result = -1;
  if(searchIndex < itemCount)
  {
    if(items[searchIndex] == target)
      result = searchIndex;
    else
      result = getIndexOf(target, searchIndex + 1);
  }
  return result;
}
```

Using recursion for `getFrequencyOf()`

```
int ArrayBag::getFrequencyOf(const string& anEntry) const
{
    return countFrequency(anEntry, 0);
}
```

```
int ArrayBag::countFrequency(const string& target, int searchIndex) const
{ // count within items[searchIndex..(itemCount - 1)]
    int frequency = 0;
    if (searchIndex < itemCount)
    {
        if (items[searchIndex] == target)
            frequency = 1 + countFrequency(target, searchIndex + 1);
        else
            frequency = countFrequency(target, searchIndex + 1);
    }
    return frequency;
}
```

Using recursion for the two functions

- The ways of applying recursion on `getIndexOf ()` and `getFrequencyOf ()` are different.
 - We add a second parameter to `getIndexOf ()` .
 - We create a new function `countFrequency ()` for `getFrequencyOf ()` .
- The key factor determining the difference is their **visibility**.
 - A private function is only used in the class. Its header can be modified.
 - A public function is called by a client. Its header should remain unchanged.
 - If we add `searchIndex` to the a public function header, we reveal some information about the implementation. This **violates data hiding**.

Dynamically adjusting the array size

- In the previous implementation, the array size is fixed.
- With **dynamic memory allocation** (DMA), we may make it **changeable**.
 - E.g., let's double the array length when it is full.

Modifying the class definition

- To do so, first we change the class definition.
 - **items** becomes (purely) a **pointer** rather than a (static) array.
 - When we implement DMA, we need a **destructor**.
 - All others are unchanged.

```
class ArrayBag : public BagInterface
{
private:
    static const int DEFAULT_CAPACITY = 6;
    string* items;
    int itemCount;
    int maxItems;
    int getIndexOf(const string& target) const;
public:
    ArrayBag();
    ~ArrayBag();
    // others are not changed
};
```

The constructor and destructor

- The constructor is modified to initialize **items**.
 - It points to a dynamic array.
- The destructor releases the dynamic array.

```
ArrayBag::ArrayBag()  
    : itemCount(0), maxItems(DEFAULT_CAPACITY)  
{  
    items = new string[DEFAULT_CAPACITY];  
}  
  
ArrayBag::~~ArrayBag()  
{  
    delete [] items;  
}
```


The member function add()

- **add()** is modified:
 - When the array is full, double the array length.
- We use a local pointer **oldArray** to point to the existing array.
 - Do this before you use **items** to point to the new array.
 - Otherwise you will have memory leak.
- Then release the old space.

```
bool ArrayBag::add(const string& newEntry)
{
    bool hasRoomToAdd = (itemCount < maxItems);
    if(!hasRoomToAdd)
    {
        string* oldArray = items;
        items = new string[2 * maxItems];
        for(int index = 0; index < maxItems; index++)
            items[index] = oldArray[index];
        delete [] oldArray;
        maxItems = 2 * maxItems;
    }
    items[itemCount] = newEntry;
    itemCount++;
    return true;
}
```

Should we make the array dynamic?

- A dynamic array is of course useful.
 - There is no need to worry about whether the array is full.
- Disadvantages:
 - It can **hurt efficiency**: Every time when the array is full, we need to allocate a new space, copy and paste, and releasing an old space.
 - It can **waste spaces** (as all static arrays do).
- Another way to prevent your bag from being full is to use a **link-based implementation**.