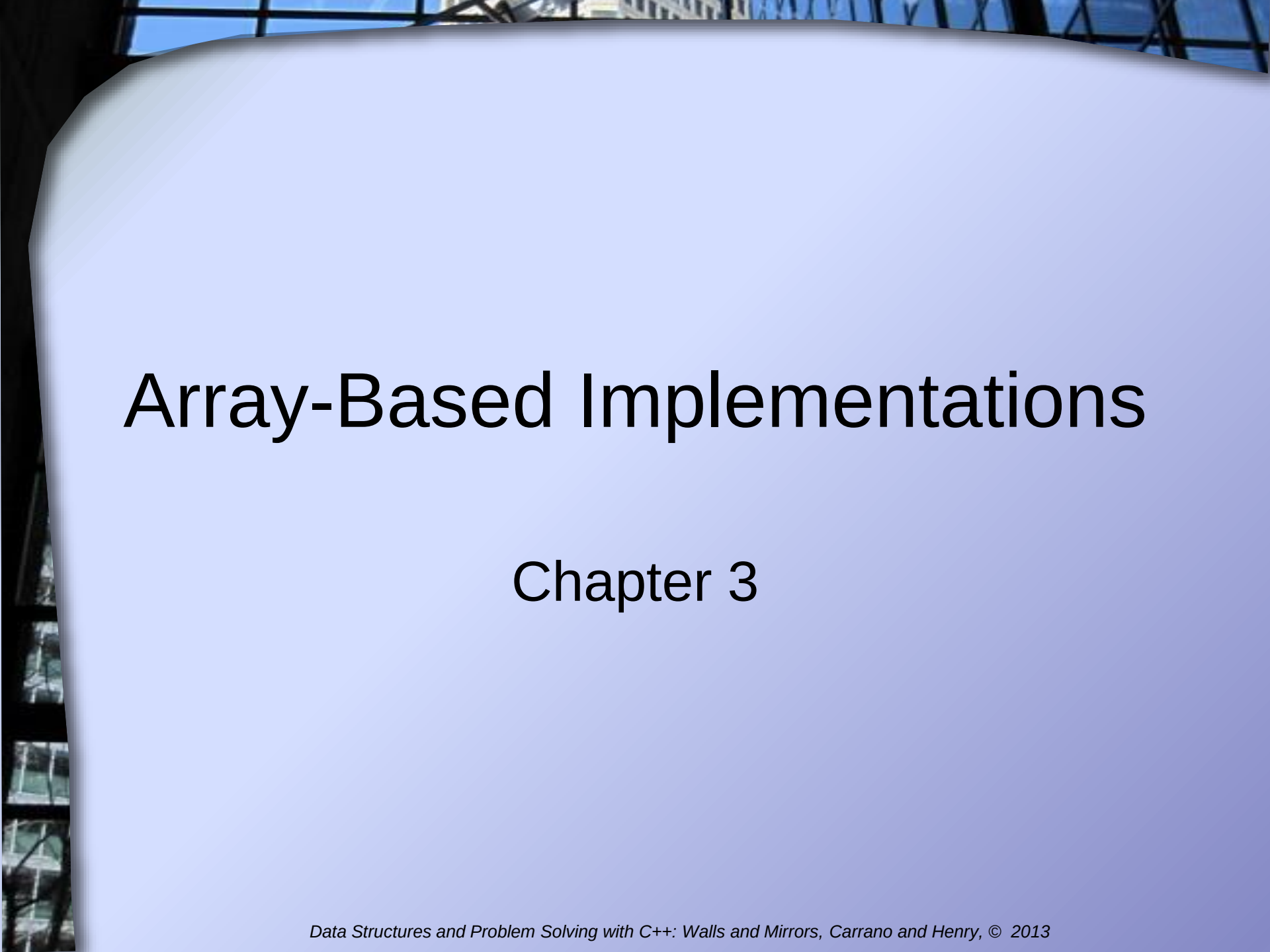


Array-Based Implementations

Chapter 3

Contents

- The Approach
- An Array-Based Implementation of the ADT Bag
- An Array-Based Implementation of the ADT Bag

The Approach

- Review of an ADT
  - A collection of data and a set of operations on that data
  - Specifications indicate *what* the operations do, *not how* to implement
- Implementing as class provides way to enforce a wall
  - Prevents access of data structure in any way other than by using the operations.

The Approach

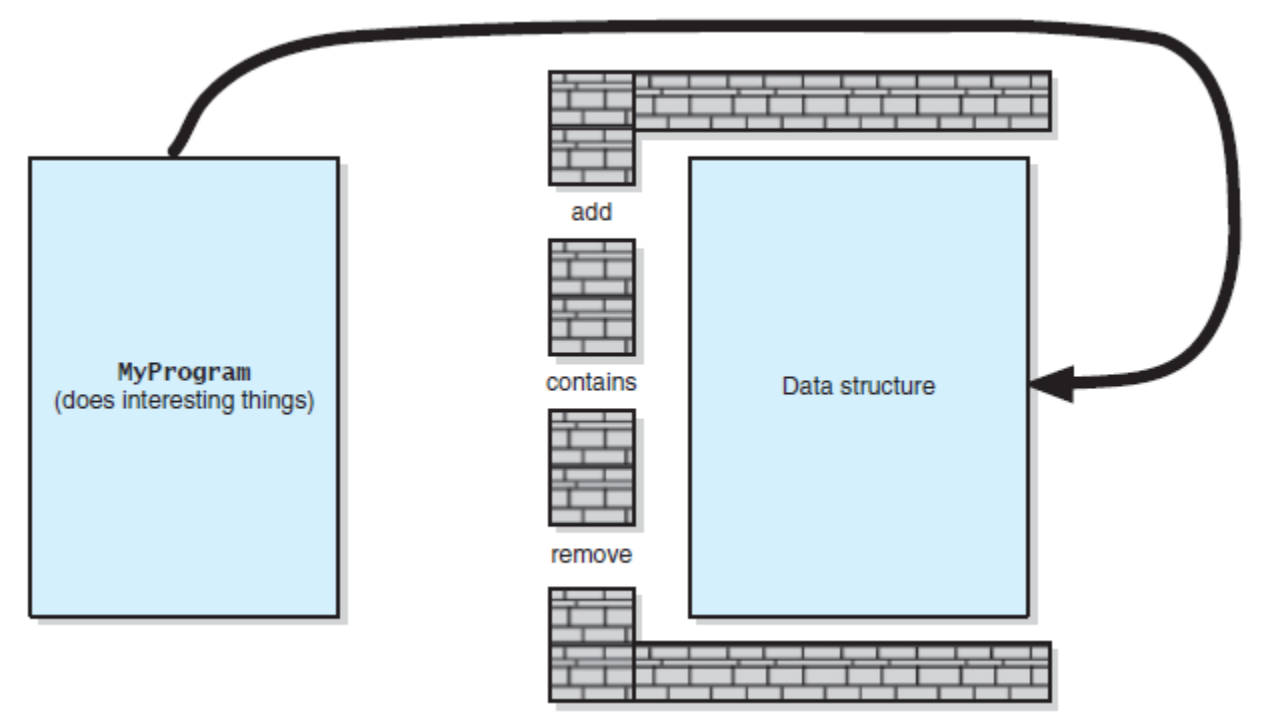


FIGURE 3-1 Violating the wall of ADT operations

Core Methods

- Methods which do basic tasks
  - Add
  - Remove
  - toVector (for display contents)
  - Constructors
- Add and remove methods designed first

Using Fixed-Size Arrays

- Must store
  - Data item
  - Its number in the array
- Keep track of
  - Max size of the array
  - Number of items currently stored
  - Where items were removed from array

Array-Based Implementation

- Private data, see header file, Listing 3-1

```
static const int DEFAULT_CAPACITY = 50;
ItemType items[DEFAULT_CAPACITY]; // Array of bag items
int itemCount;                    // Current count of bag items
int maxItems;                     // Max capacity of the bag
```

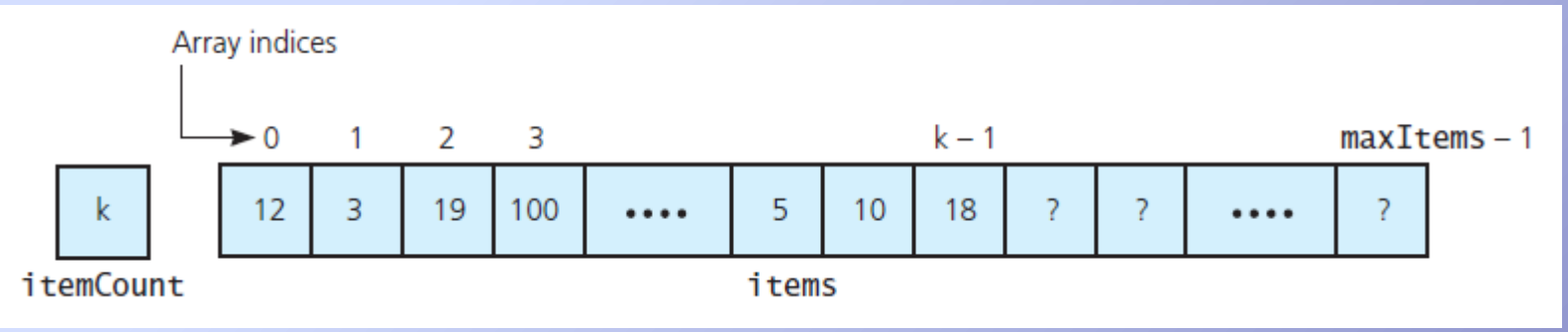


FIGURE 3-2 An array-based implementation of the ADT bag

Defining the Core Methods

- Core methods defined
  - `ArrayBag<ItemType>::ArrayBag(): itemCount(0), maxItems(DEFAULT_CAPACITY)add`
  - `bool ArrayBag<ItemType>::add( const ItemType& newEntry)`
  - `vector<ItemType> ArrayBag<ItemType>::toVector() const`
  - `int  ArrayBag<ItemType>::getCurrentSize() const`
  - `bool ArrayBag<ItemType>::isEmpty() const`

Defining the Core Methods

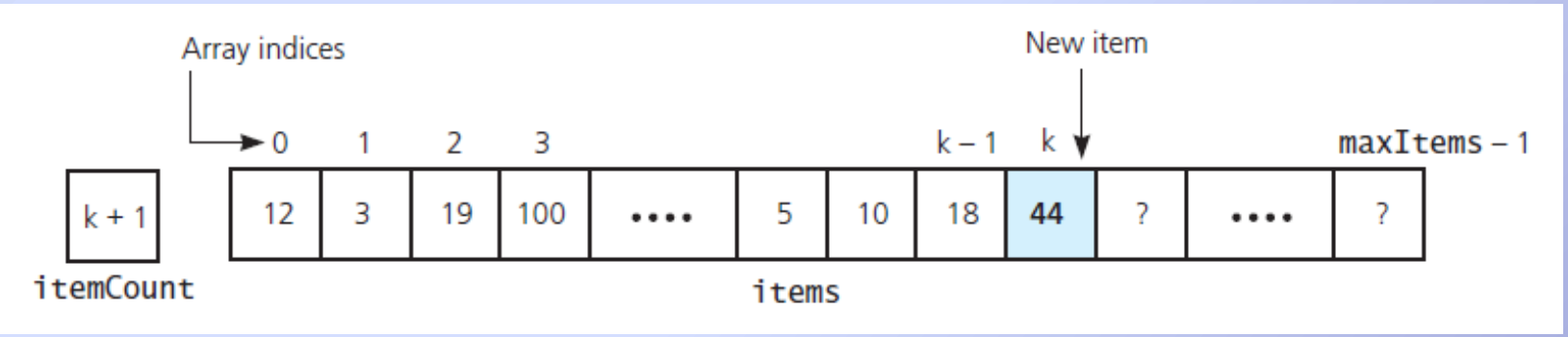


FIGURE 3-3 Inserting a new entry into an array-based bag

- View test program, Listing 3-2 and Output

Implementing More Methods

- `int ArrayBag<ItemType>:: getFrequencyOf ( const ItemType& anEntry) const`
- `bool ArrayBag<ItemType>::contains( const ItemType& target) const`

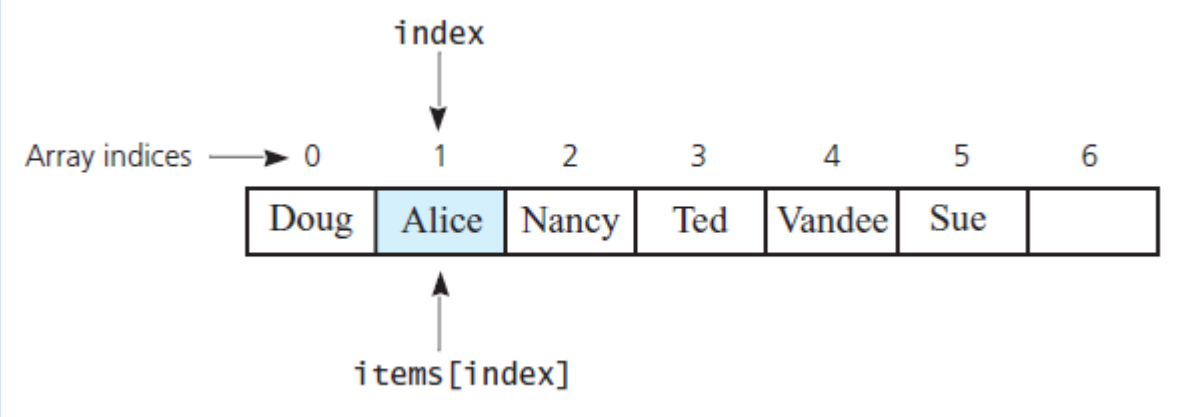


FIGURE 3-4 The array items after a successful search for the string "Alice"

Implementing More Methods

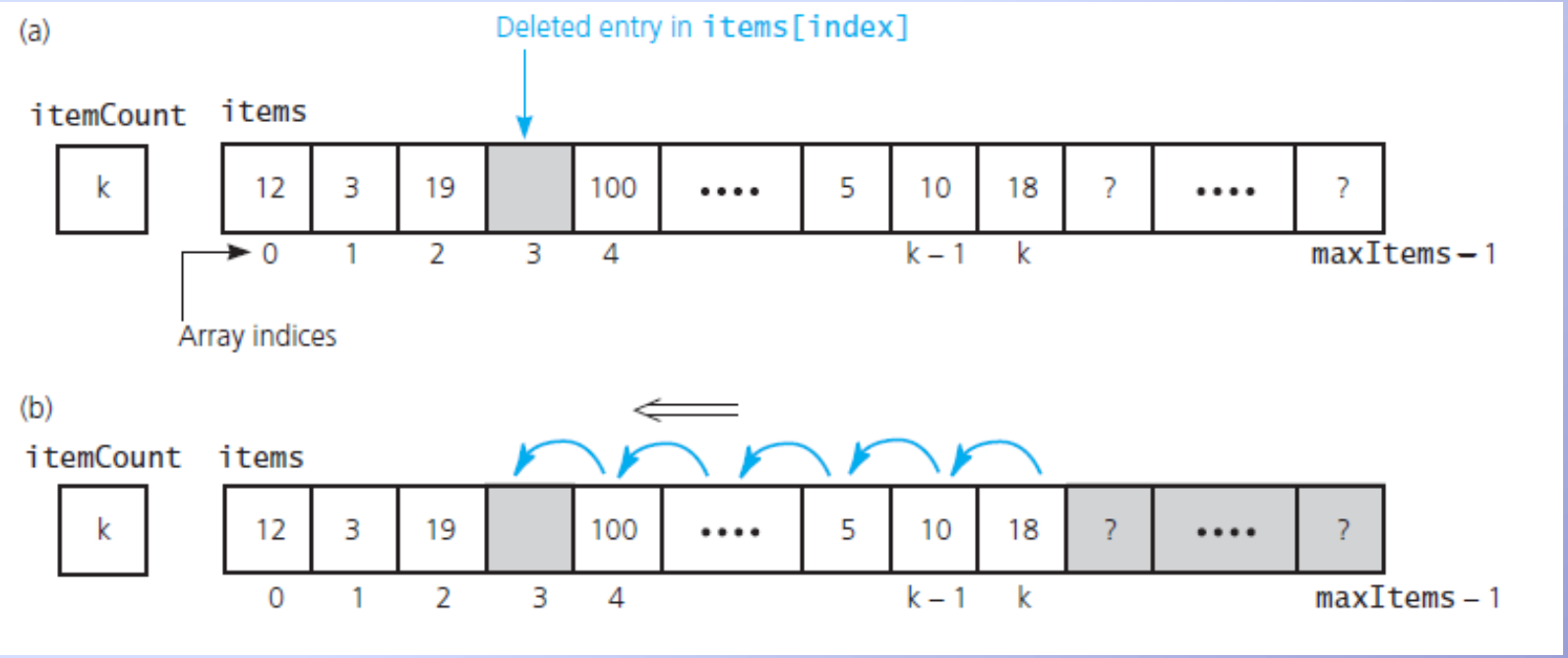


FIGURE 3-5 (a) A gap in the array items after deleting the entry in items[index] and decrementing itemCount;

Implementing More Methods

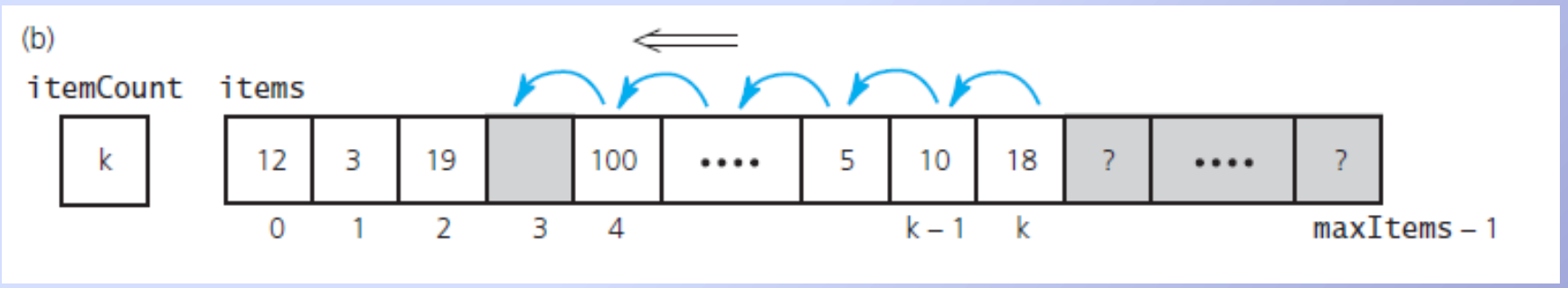


FIGURE 3-5 (b) shifting subsequent entries to avoid a gap;

Implementing More Methods

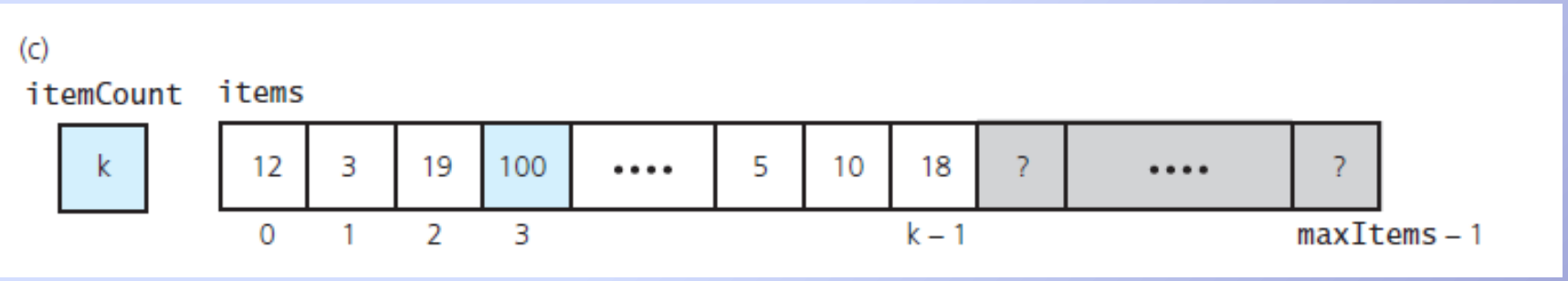


FIGURE 3-5 (c) the array after shifting;

Implementing More Methods

- `int ArrayBag<ItemType>::getIndexOf (const ItemType& target) const`
- `bool ArrayBag<ItemType>::remove( const ItemType& anEntry)`
- `void ArrayBag<ItemType>::clear()`

Implementing More Methods

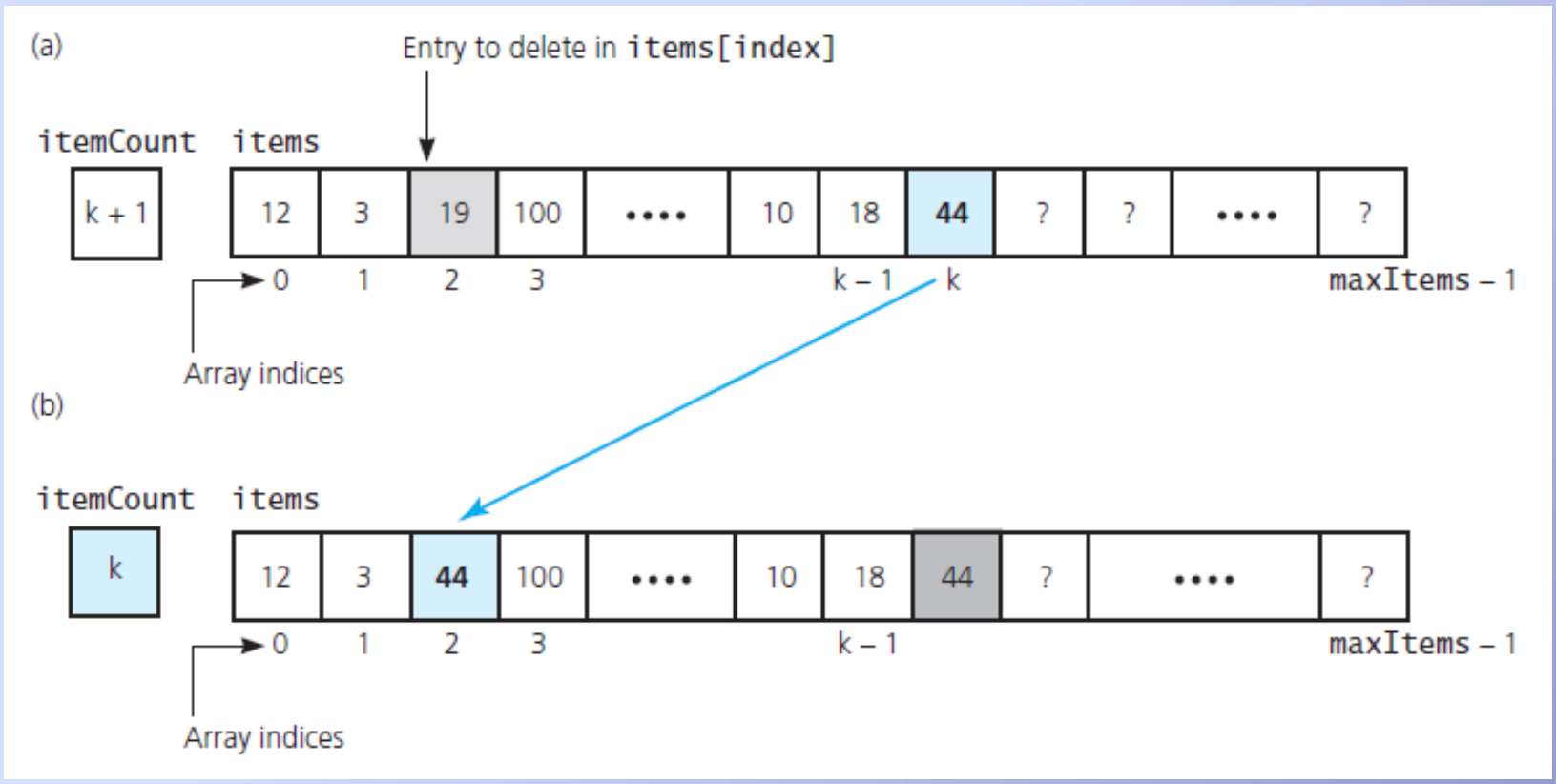


FIGURE 3-6 Avoiding a gap in the array while removing an entry

Using Recursion in the Implementation

- Recursive version of **`getIndexOf`**

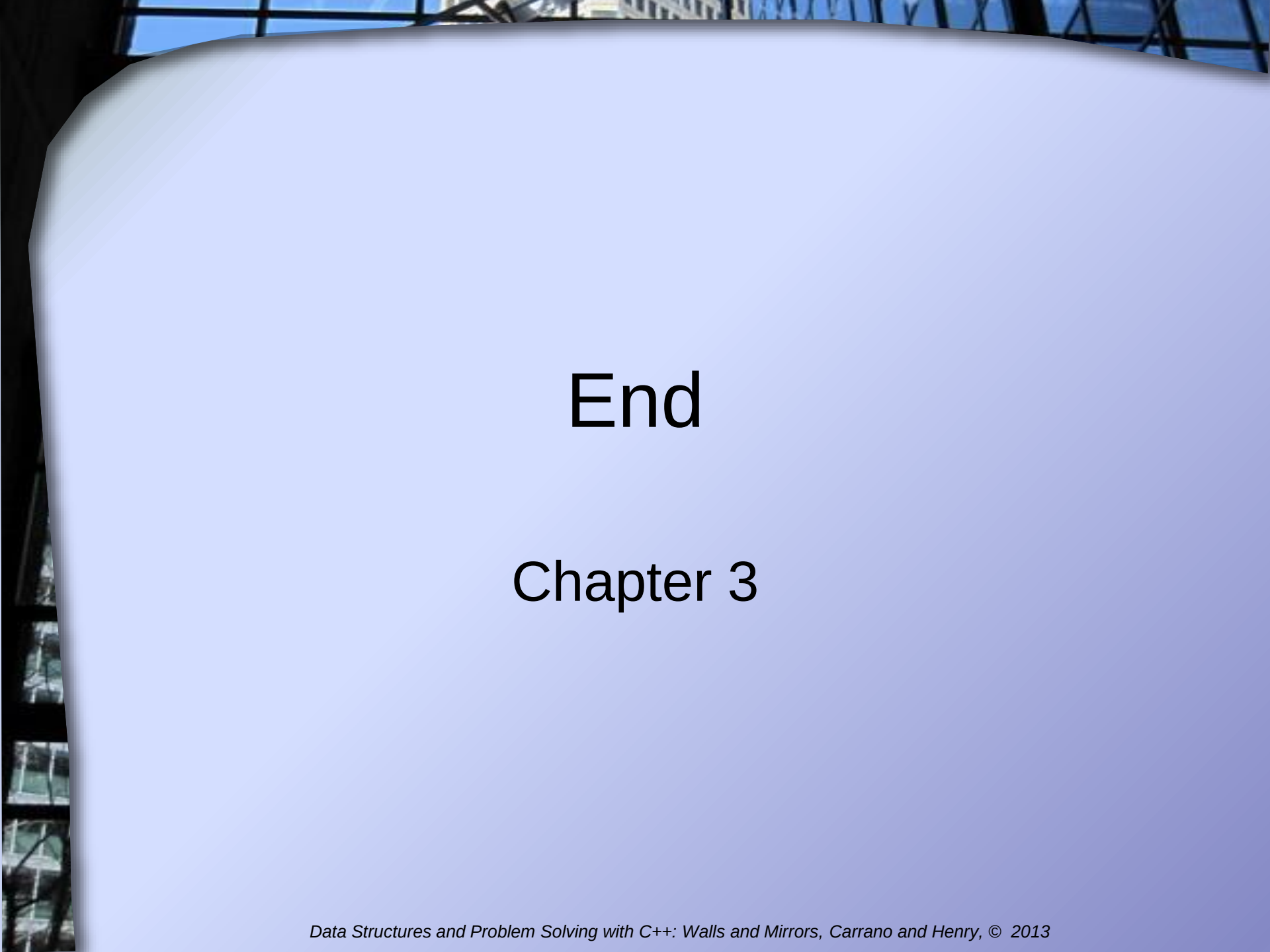
```
template<class ItemType>
int ArrayBag<ItemType>::getIndexOf(const ItemType& target, int searchIndex) const
{
    int result = -1;
    if (searchIndex < itemCount)
    {
        if (items[searchIndex] == target)
        {
           result = searchIndex;
        }
        else
        {
           result = getIndexOf(target, searchIndex + 1);
        } // end if
    } // end if

    return result;
}   // end getIndexOf
```

Using Recursion in the Implementation

- Recursive version of **`getFrequencyOf`**

```
template<class ItemType>
int ArrayBag<ItemType>::countFrequency(const ItemType& target,
                                       int searchIndex) const
{
   if (searchIndex < itemCount)
   {
      if (items[searchIndex] == target)
      {
         return 1 + countFrequency(target, searchIndex + 1);
      }
      else
      {
         return countFrequency(target, searchIndex + 1);
      }  // end if
   }
   else
      return 0;   // Base case
}  // end countFrequency
```



End

Chapter 3