BACS2063 Data Structures and Algorithms

Array Implementations of Collection ADTs

Chapter 4

Learning Outcomes

At the end of this chapter, you should be able to

- Implement the ADTs list, stack and queue using arrays.
- Discuss the strengths and weaknesses of using arrays to implement the ADTs.
- Analyze the efficiency of the array implementations of the ADTs

Recall: Creating an ADT

1. Specify the ADT
   – Write an ADT specification which describes the characteristics of that data type and the set of operations for manipulating the data
2. Implement the ADT
   a) Create a Java interface
      - Include all the operations from the ADT specification

   b) Create a Java class which implements the interface
      - Determine how to represent the data
      - Implement all the operations from the interface
3. Use the ADT in a client program or application

Lists

Lists

- A list provides a way to organize data

Fig. 4-1 A to-do list.

Specifications for the ADT List

- Operations on lists
  - Add new entry – at end, or anywhere
  - Remove an item
  - Remove all items
  - Replace an entry
  - Look at any entry
  - Look for an entry of a specific value
  - Count how many entries
  - Check if list is empty, full

List ADT

- Refer to Appendix 4.1 for List ADT specification
  - Note that in this specification, entries in the list have positions that begin with 1 to be consistent with typical lists used in everyday life.
- Remember that at this point,
  - You should not think about how to represent the list in your program or how to implement its operations.
  - Instead, focus on what are the operations and what the operations do, not on how they do them.
  - *i.e.*, at this point, the list is an abstract data type.

Issues to consider for the operations

- **`add`**, **`remove`**, **`replace`**, **`getEntry`** work OK when valid position given
- **`remove`**, **`replace`** and **`getEntry`** not meaningful on empty lists
- A list could become full, what happens to **`add`**?

Design Issues

- When you specify an ADT, you need to decide how to handle special / unusual conditions, and the ADT specification should include details on how the special conditions would be handled by the various operations.

Possible Solutions

- Assume the invalid situations will not occur
- Ignore the invalid situations
- Make reasonable assumptions, act in predictable way
- Return boolean value indicating success or failure of the operation
- Throw an exception

Array List Implementation (1)

- Java interface
  - Contains the method declarations of all the operations listed in the List ADT specification.
  - All the methods are **abstract methods**, i.e. method headers without bodies.
- Refer to **`Chapter4\adt\`**
  - **`ListInterface.java`**

Array List Implementation (2)

- Data fields in the Java class:
  - An array – to store the elements of the list
  - An integer variable – to keep track of the current total number of elements in the list

```
T[] array;   // array of list entries
int length; // current no. of entries in the list
```

Note: `T` represents the data type of the entries in the list. It will be defined as generic type within the class.

Generic Types (1)

- Used in Java interface and classes which implement ADTs to specify and constrain the type of objects being stored in the collection.
- The interface name or class name is followed by an identifier enclosed in angle brackets:

```
public interface ListInterface<T>
```

- The identifier **T** – which can be any identifier but usually is a single capital letter – represents the data type within the class definition.

Generic Types (2)

- When you use the class, you supply an actual data type to replace **`T`**, e.g.:

```
ListInterface<String> taskList;
```

- Now, whenever **`T`** appears as a data type in the definition of **`ListInterface`**, **`String`** will be used.

- Note: A generic type must be a reference type, not a primitive type.

Array List Implementation (3)

- Java class
  - Implements the interface **`ListInterface`**
  - Operations – implemented as Java methods
    - Core operations – as appears in the list ADT specification and also the interface **`ListInterface`**
    - Utility operations – to support the core operations

Method `add(newEntry)`

- Adds a new item *at the end* of the list
  - Assign new value at end
  - Increment **`length`** of list

0	1	2	3	4	5	6		length
apple	orange	durian	lemon	plum				5

`add(newPosition, newEntry)`

- Adds an item in at a specified position
  - Requires a utility method `makeRoom()` to shift elements ahead

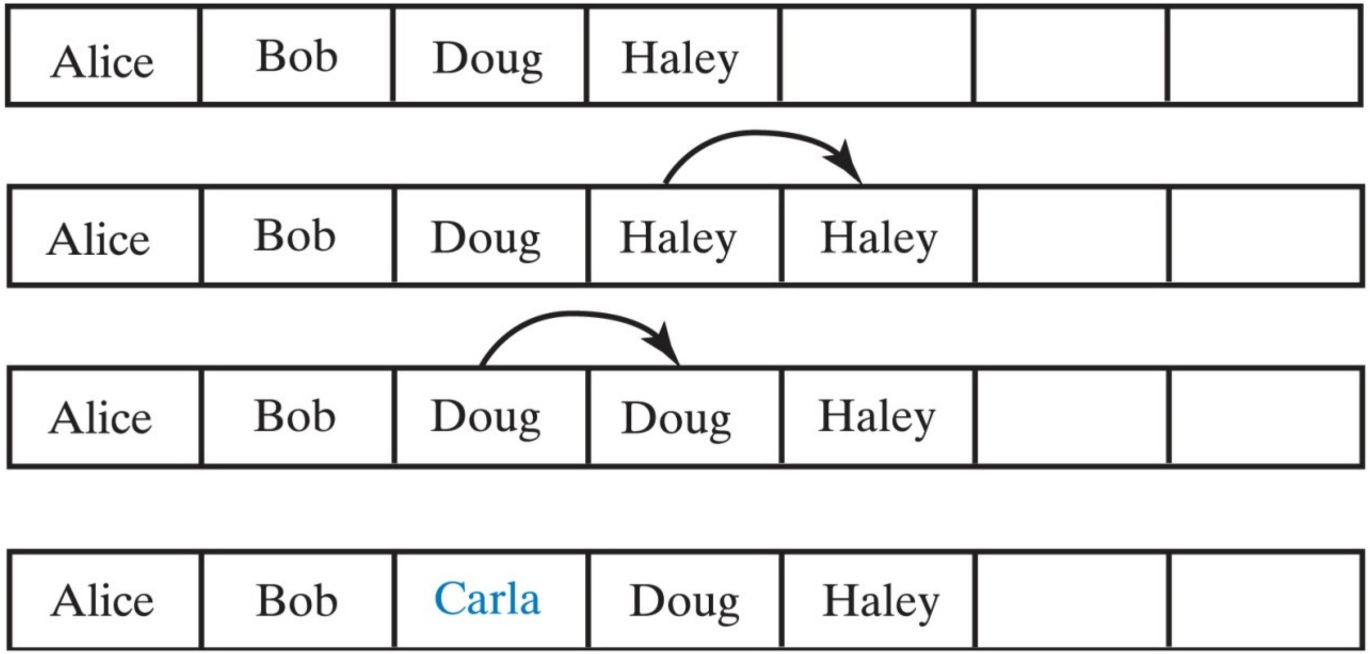
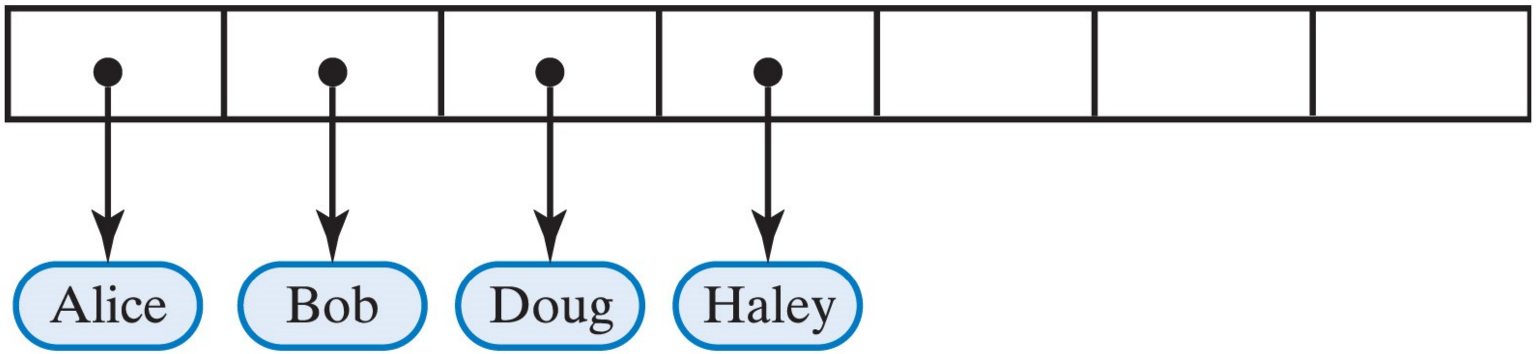


Fig. 5-3 Making room to insert Carla as third entry in an array.

Array of Objects

- An array of objects actually contains references to those objects



- For simplicity, figures portray arrays as if they actually contained objects

Method `remove(givenPosition)`

- Removes the item at the specified position.
- Must shift existing entries to avoid gap in the array – requires the utility method `removeGap()`
  - Except when removing last entry
- Method must also handle error situations
  - When position specified in the remove is invalid
  - When `remove()` is called and the list is empty
  - Invalid call returns null value

Removing a List Entry

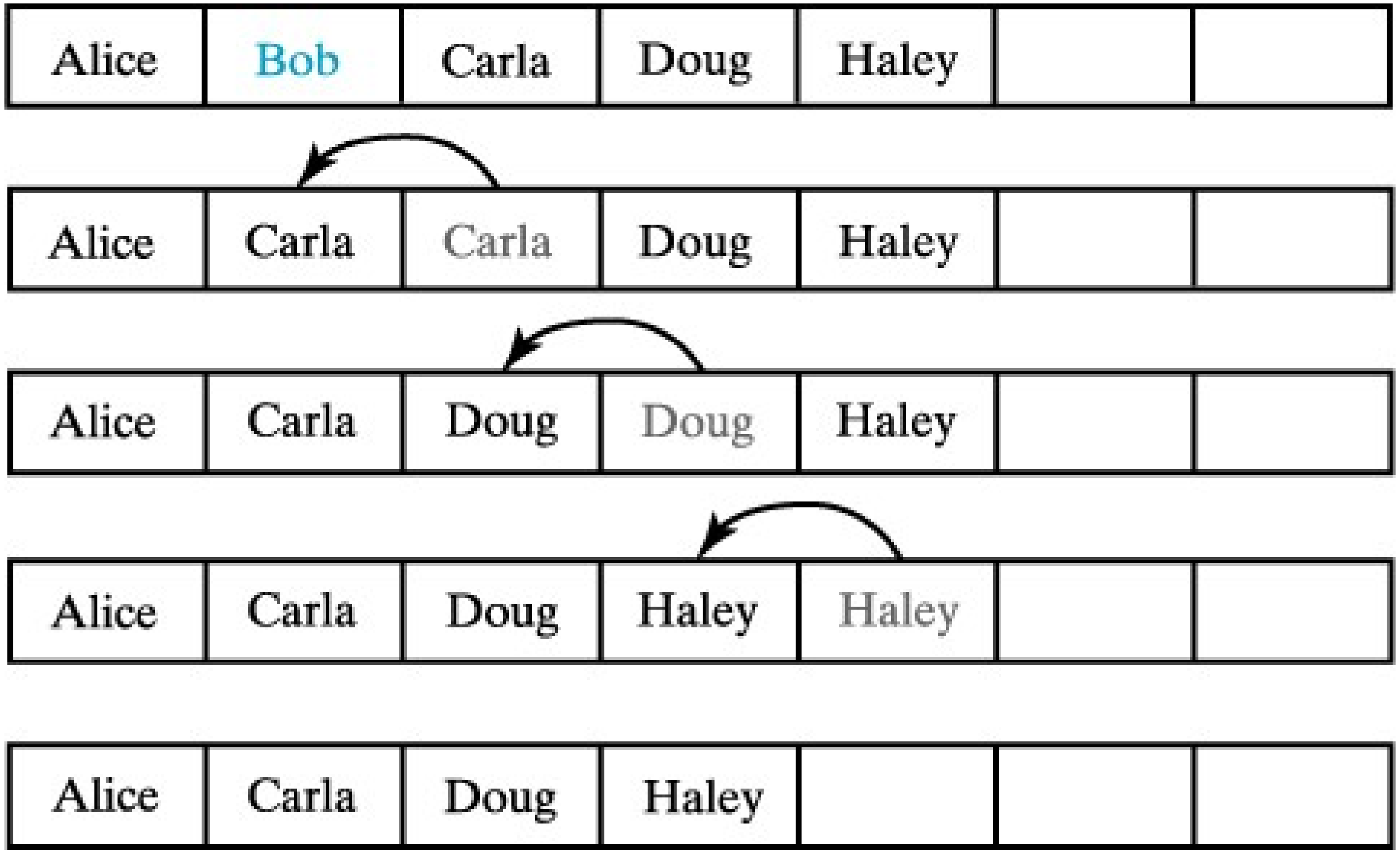


Fig. 5-5 Removing Bob by shifting array entries.

Question

Figure 5-5 in the previous slide shows Haley shifted one position toward the beginning of the array. Actually, the reference to Haley is copied, not moved, to its new location. Should we assign `null` to Haley's original location?

Algorithms for Other Methods?


```
public boolean replace (int givenPosition,
                        T newEntry);
public T getEntry (int givenPosition);
public boolean contains (T anEntry);
```

The Java Implementation

Refer to:

`Chapter4\adt\ArrList.java`

Note:

- In the interface **`ListInterface`**, each abstract method corresponds to an ADT list operation.
- Since the **`ArrList`** class implements **`ListInterface`**, **`ArrList`** contains the implementation of each abstract method of **`ListInterface`**.

Problem

- What if the array becomes full, i.e. all the array locations are assigned entries?

Expanding an Array (1/4)

- An array has a fixed size
  - If we need a larger list, we are in trouble
- When array becomes full, move its contents to a larger array (dynamic expansion)
  - Copy data from original to new location
  - Manipulate names so new location keeps name of original array
- Need two utility methods for expanding an array:
  - `isArrayFull()` to determine if the array is already full
  - `doubleArray()` to expand the array

Expanding an Array (2/4)

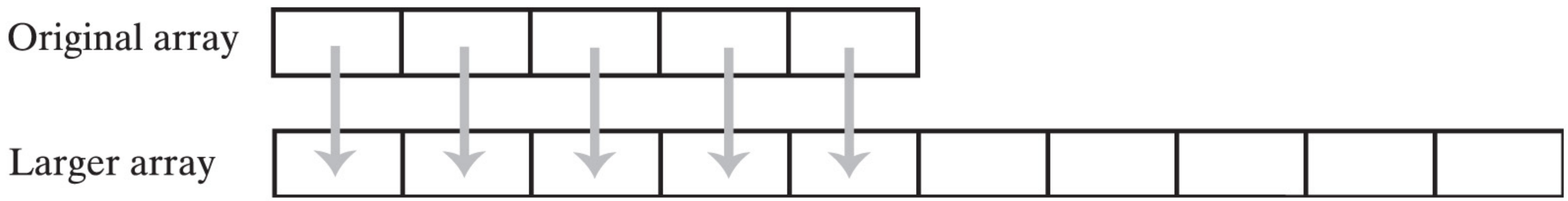


Fig. 5-6 The dynamic expansion of an array copies the array's contents to a larger second array.

Expanding an Array (3/4)

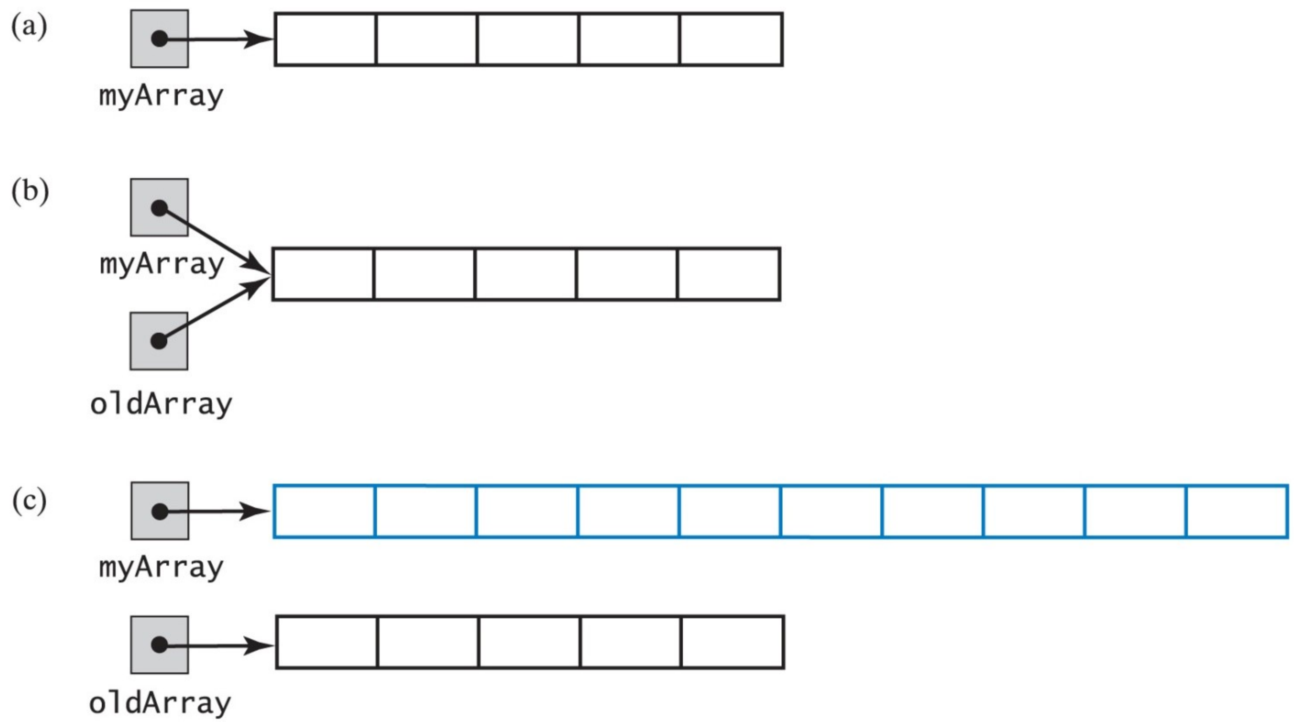


Fig. 5-7 (a) an array;
(b) the same array with two references;
(c) the two arrays, reference to original array now referencing a new, larger array

Expanding an Array (4/4)

- Code to accomplish the expansion shown in Fig. 5-7, previous slide

```
int [] myArray = new int [INITIAL_SIZE];
int [] oldArray = myArray;
     // save reference to myArray
myArray = new int [2 * oldArray.length];
     // double size of array
for (int index = 0 ;
         index < oldArray.length ;
         index++)
       myArray [index] = oldArray [index];
```

Expandable List Implementation

- Change the **`isFull`** to always return false
  - We will expand the array when it becomes full
  - We keep this function so that the original interface does not change
- The **`add()`** methods will double the size of the array when it becomes full
- Now declare a private method **`isArrayFull`**
  - Called by the **`add()`** methods

Pros and Cons of Array Implementation for the ADT List

- ☑ Retrieving an entry is fast
- ☑ Adding an entry at the end of the list is fast

- ✘ Adding or removing an entry that is between other entries requires shifting elements in the array
- ✘ Increasing the size of the array requires copying elements

Question

Consider the following statements that create a list of Name objects:

```
ListInterface<Name> nameList = new ArrList<Name>();
Name amy = new Name("Amy", "Tan");
nameList.add(amy);
nameList.add(new Name("Jack", "Bauer"));
nameList.add(new Name("Jim", "Taylor"));
nameList.display();
```

Suppose that the return type of `getEntry` was `Object` instead of a generic type. Would this change affect how you use the method? In particular, would the following statement for retrieving the second name in `nameList` be correct? Why?

```
Name secondName = nameList.getEntry(2);
```

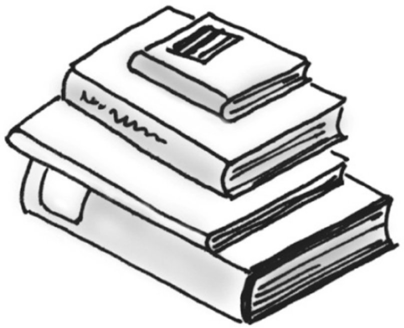
Sample Code

- **`Chapter4\adt\`**
  - `ListInterface.java`
  - `ArrList.java`
- **`Chapter4\entity\`**
  - `Runner.java`
- **`Chapter4\client\`**
  - `Registration.java`

Stacks

Stacks

- New items are added to the top of the stack, i.e. the item most recently added is always on the top. The most recently added item is always the next item to be removed.
- Exhibits LIFO behavior.



ADT Stack Operations

- `push(newEntry)`
- `pop()`
- `peek()`
- `isEmpty()`
- `clear()`

Array Stack Implementation: Consideration

- When using an array to implement a stack
  - The array's first element should represent the bottom of the stack
  - The last occupied location in the array represents the stack's top
- This avoids shifting of elements of the array if it were done the other way around

Comparison of 2 approaches

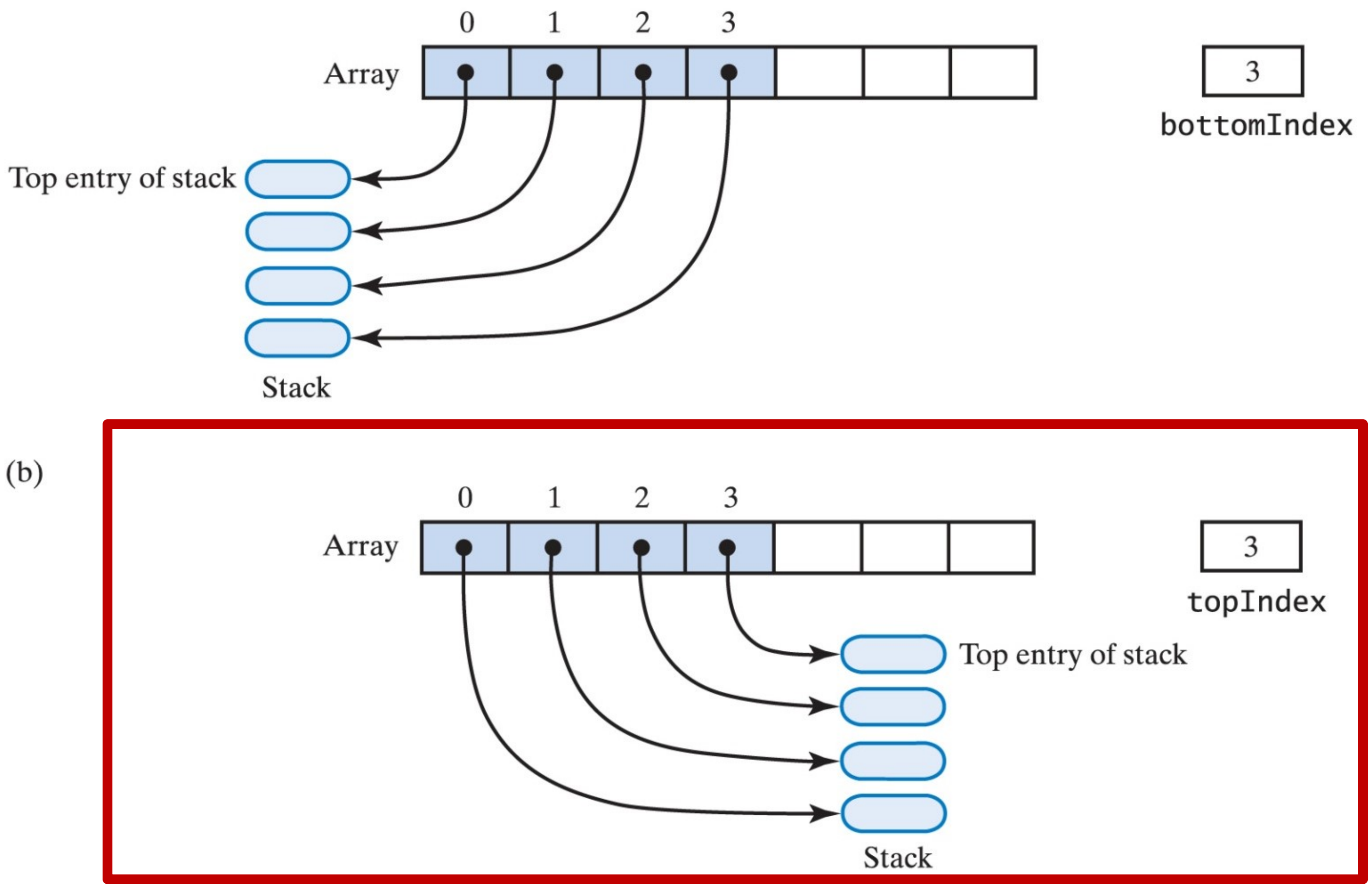


Fig. 22-4 An array that implements a stack; its first location references (a) the top of the stack; (b) the bottom of the stack

Array Stack Implementation

- Java interface: refer to **`StackInterface.java`**
- Data fields in the Java class:
  - An array – to store the entries of the stack
  - An integer variable – represents the array index of the top entry; initialized to -1 to indicate an empty stack

```
T[] array;      // array of stack entries
int topIndex;   // index of the top entry
```

  - Java class: refer to **`ArrayStack.java`**

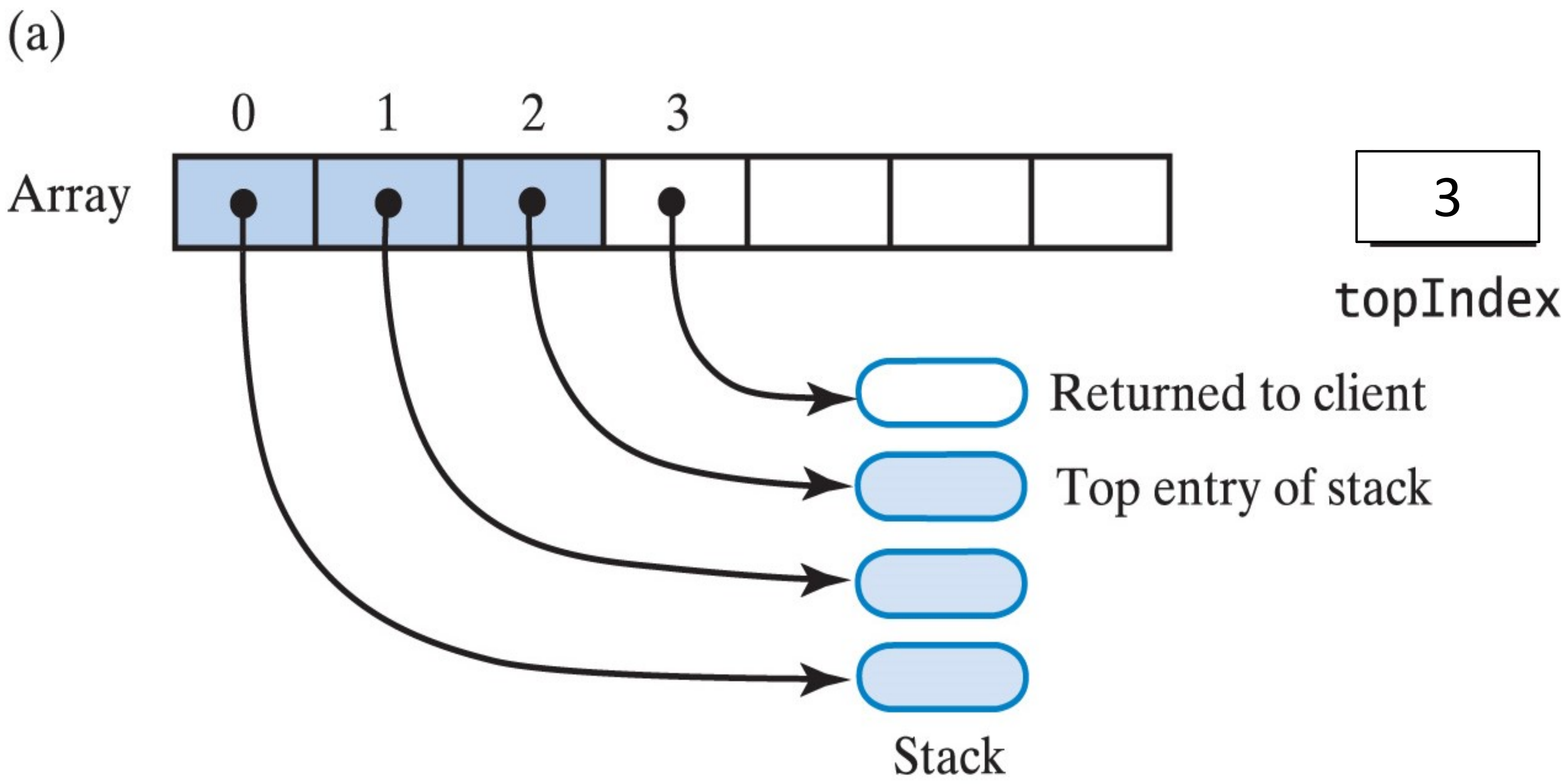
Method `push(newEntry)`

- Adds a new item *at the top* of the stack
  1. Increment **`topIndex`**
  2. Assign new value at the array location indicated by **`topIndex`**

Method `pop()` (1)

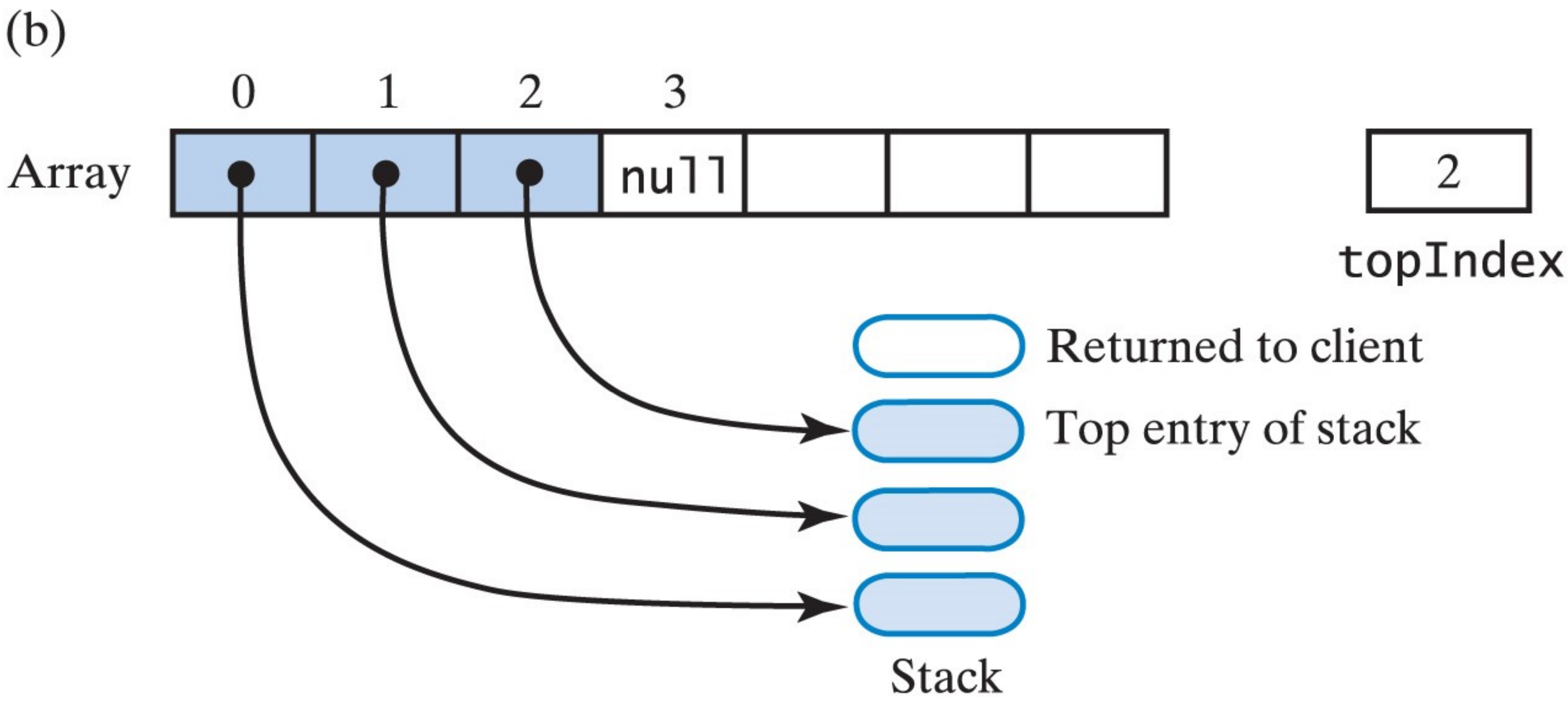
- Removes the entry *at the top* of the stack
  1. Assign the entry at the array location indicated by **`topIndex`** to a temporary variable (to be returned)
  2. Decrement **`topIndex`**

Method `pop()` (2)



Assign the value at the array location indicated by **`topIndex`** to a temporary variable to be returned to the calling method

Method **pop()** (3)



Setting **stack[topIndex]=null** and then decrement **topIndex**

Exercise

Write the algorithm for the method **`convertNumberToBinary(int number)`** to convert the given number to its equivalent binary (base-2) representation and returns the result as a string value.

Hint: use a stack.

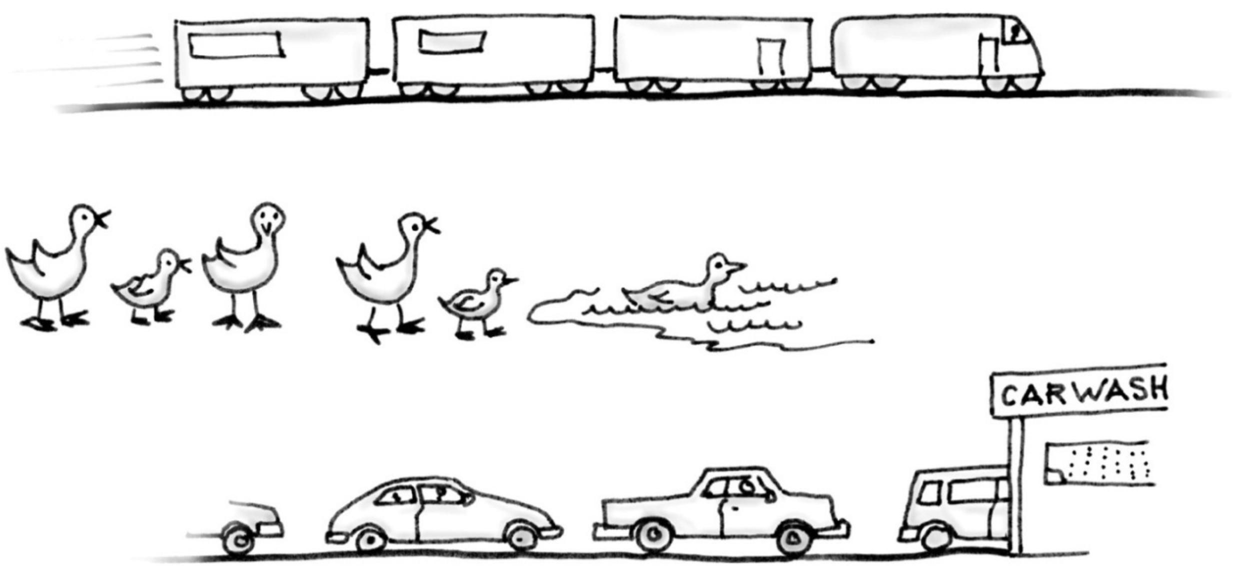
Sample Code

- `Chapter4\adt\`
  - `StackInterface.java`
  - `ArrayStack.java`
- `Chapter4\client\`
  - `StringReversal.java`

Queues

Queues

- Queue organizes entries according to order of entry - exhibits FIFO behavior
- All additions are at the back of the queue. Front of queue has items added first



ADT Queue Operations

- **`enqueue`**
- **`dequeue`**
- **`getFront`**
- **`isEmpty`**
- **`clear`**

Array Queue Implementation

- Java interface: refer to `QueueInterface.java`
- Data fields in the Java class:
  - An array – to store the entries of the queue
  - Two integer variables – to represent
    - the array index of the **front** of the queue and
    - the array index of the **back** of the queue

```
T[] array;        // array of queue entries
int frontIndex;  // index of the front entry
int backIndex;   // index of the back entry
```

Array Queue Implementation: Variations

1. Linear array with fixed front?
2. Linear array with dynamic front?
3. Circular array?

Method 1:
Linear Array with Fixed Front

Data Fields

Method 1:
Linear Array
Fixed Front

- The front of the queue is fixed to **`queue[0]`**, i.e., **`frontIndex`** is always 0.
- **`backIndex`** initialized to -1 to indicate an empty queue

Method `enqueue()`

Method 1:
Linear Array
Fixed Front

1. Increment **`backIndex`**
2. Assign new value at the array location indicated by **`backIndex`**

Method `dequeue()`

Method 1:
Linear Array
Fixed Front

1. Assign the entry at array location **0** to a temporary variable (to be returned)
2. Shift entries from array location **1** to **`backIndex`** one step towards the front of the array
3. Decrement **`backIndex`**

Method `isEmpty()`

Method 1:
Linear Array
Fixed Front

- The queue is empty if **`backIndex`** is equal to **-1**

Strength and Weakness

Method 1:
Linear Array
Fixed Front

- ☑ Easy to understand as it is similar to how everyone else in a queue moves forward a step
- ✘ The **`dequeue`** operation is inefficient: there's overhead incurred as must shift entries each time we remove an entry

Method 2:
Linear Array with Dynamic Front

Method 2:
Linear Array
Dynamic Front

Data Fields

- **`frontIndex`** is dynamic, *i.e.*, we instead "move" (i.e. update) **`frontIndex`**
- **`backIndex`** initialized to -1; **`frontIndex`** initialized to 0

Method `enqueue()`

Method 2:
Linear Array
Dynamic Front

1. Increment **`backIndex`**
2. Assign new value at the array location indicated by **`backIndex`**

Method `dequeue()`

Method 2:
Linear Array
Dynamic Front

1. Assign the entry at array location **`frontIndex`** to a temporary variable (to be returned)
2. Increment **`frontIndex`**

Method 2:
Linear Array
Dynamic Front

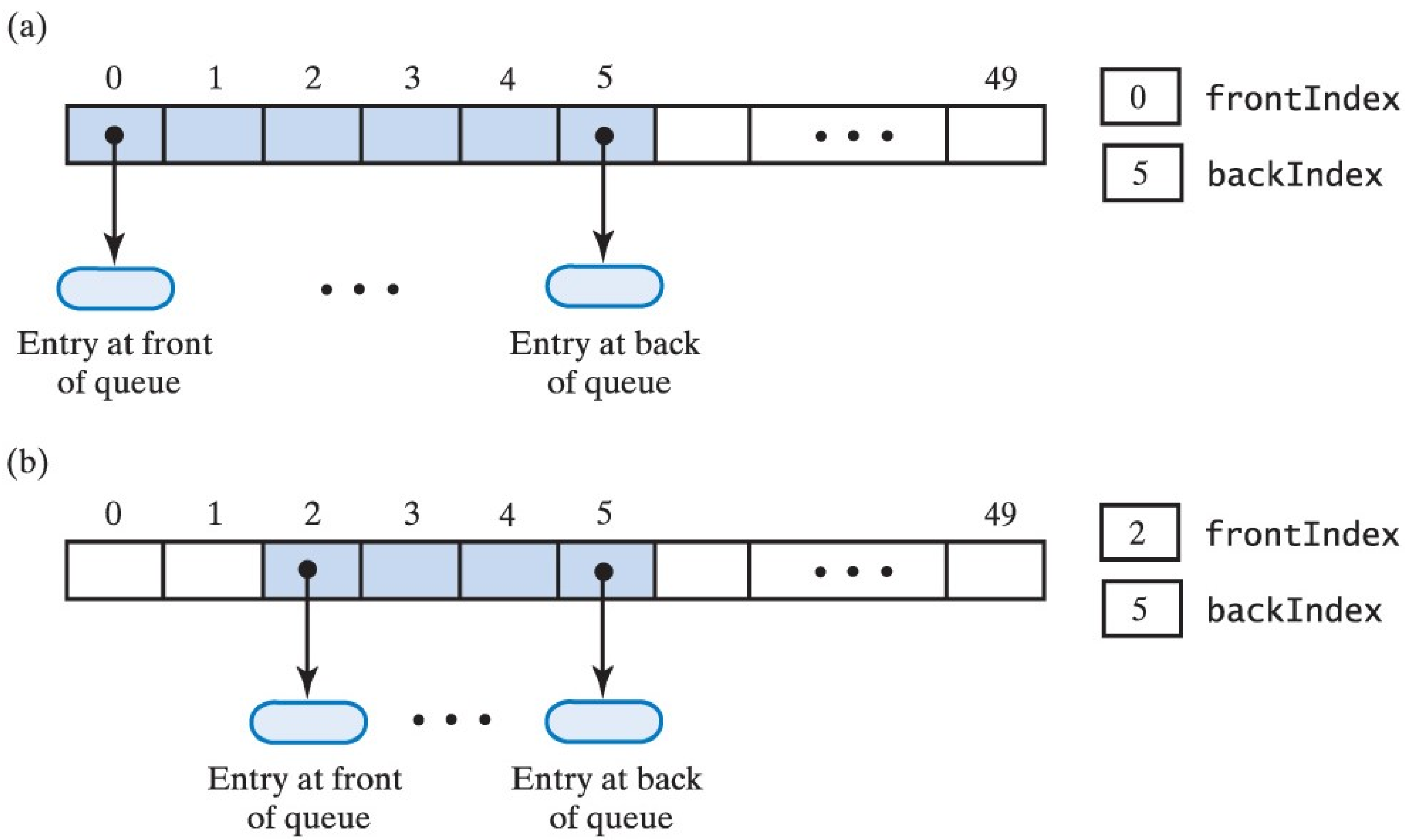


Fig. 24-6 An array that represents a queue without shifting its entries: (a) initially; (b) after removing the front twice;

Method `isEmpty()`

Method 2:
Linear Array
Dynamic Front

- The queue is empty if **`backIndex < frontIndex`**

Strength and Weakness

Method 2:
Linear Array
Dynamic Front

- ☑ Do not have to shift entries after each `dequeue` operation.
- ✘ Problem: *Rightward drift*, i.e. the array can become "full" when the last array location has been occupied but there are empty locations in the beginning part of the array.
  - How to use the empty locations?

Method 3:
Circular Array

Data Fields

Method 3: Circular Array

- When queue reaches end of array, add subsequent entries to beginning
- Array behaves as though it were circular
  - First location follows last one
- **`backIndex`** initialized to -1; **`frontIndex`** initialized to 0
- Use *modulo arithmetic* to update indices:

```
backIndex = (backIndex + 1) % array.length
frontIndex = (frontIndex + 1) % array.length
```

Method 3: Circular Array

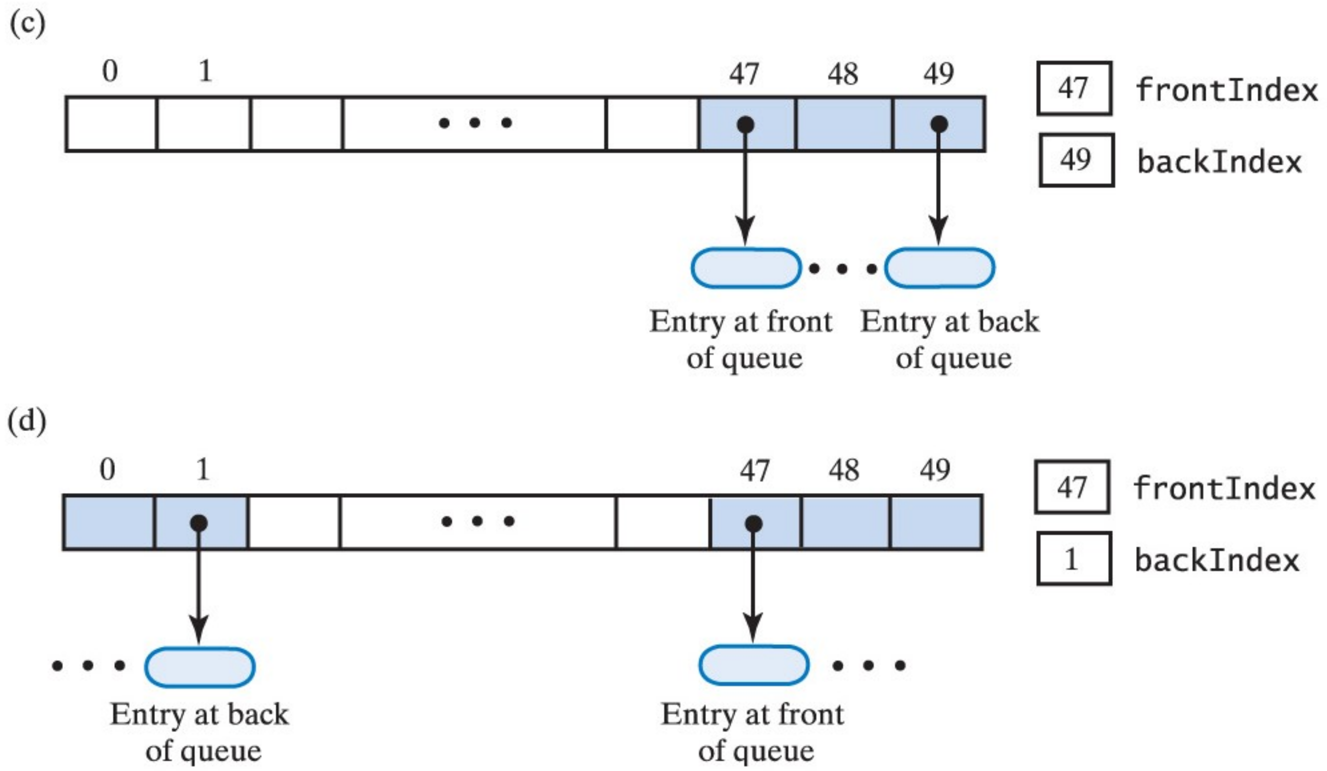


Fig. 24-6 An array that represents a queue without shifting its entries: (c) after several more additions & removals; (d) after two additions that wrap around to the beginning of the array

Method 3: Circular Array

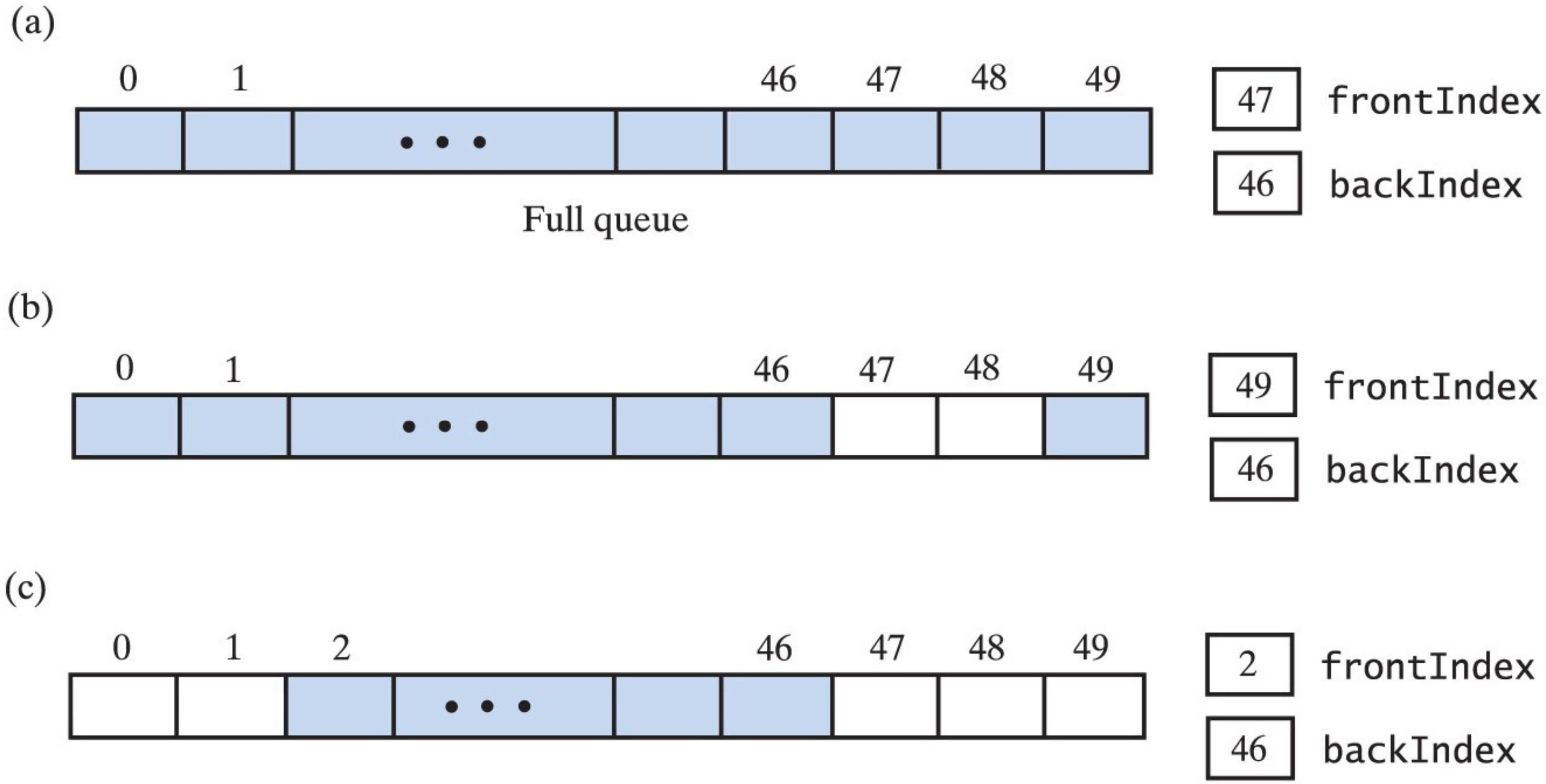


Fig. 24-7 A circular array that represents a queue: (a) when full; (b) after removing 2 entries; (c) after removing 3 more entries;

Method 3: Circular Array

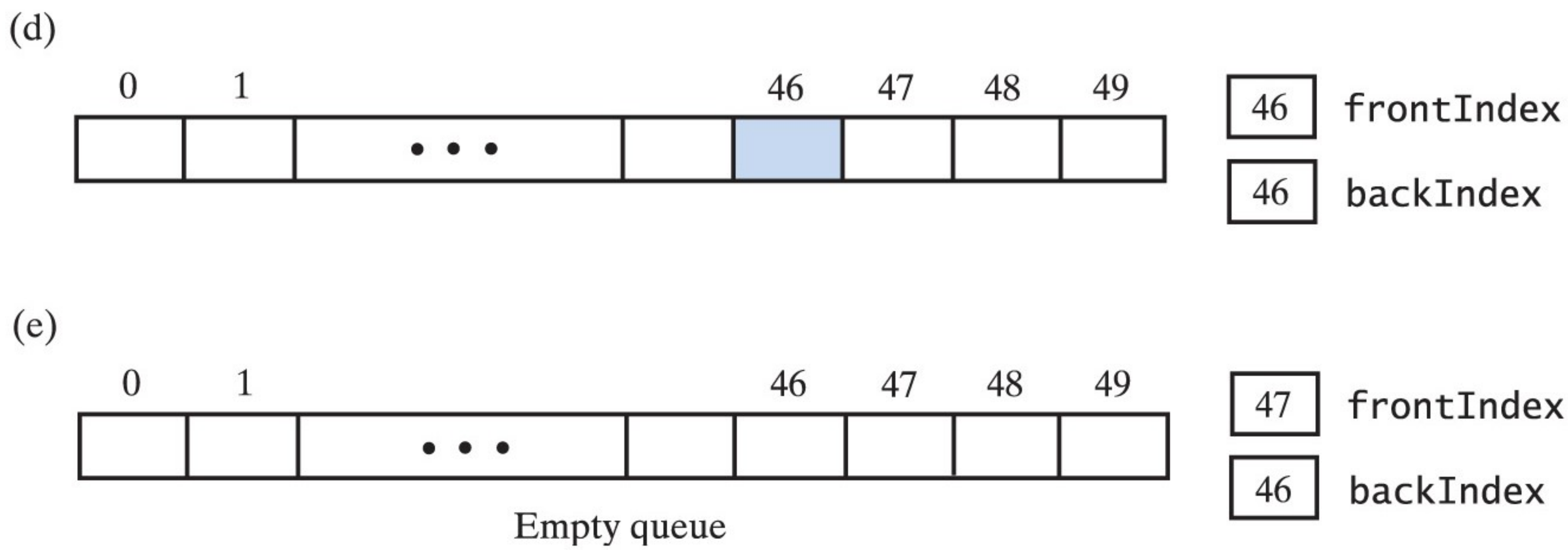


Fig. 24-7 A circular array that represents a queue:
(d) after removing all but one entry;
(e) after removing remaining entry.

Method **enqueue()**

Method 3:
Circular Array

1. Update **backIndex** using modulo arithmetic:

   ```
   backIndex = (backIndex + 1) % array.length
   ```

2. Assign new value at the array location indicated by **backIndex**

Method **dequeue()**

Method 3:
Circular Array

1. Assign the entry at array location **`frontIndex`** to a temporary variable (to be returned)
2. Update **`frontIndex`** using modulo arithmetic:

```
frontIndex = (frontIndex+1) % array.length
```

Circular Array Implementation of a Queue

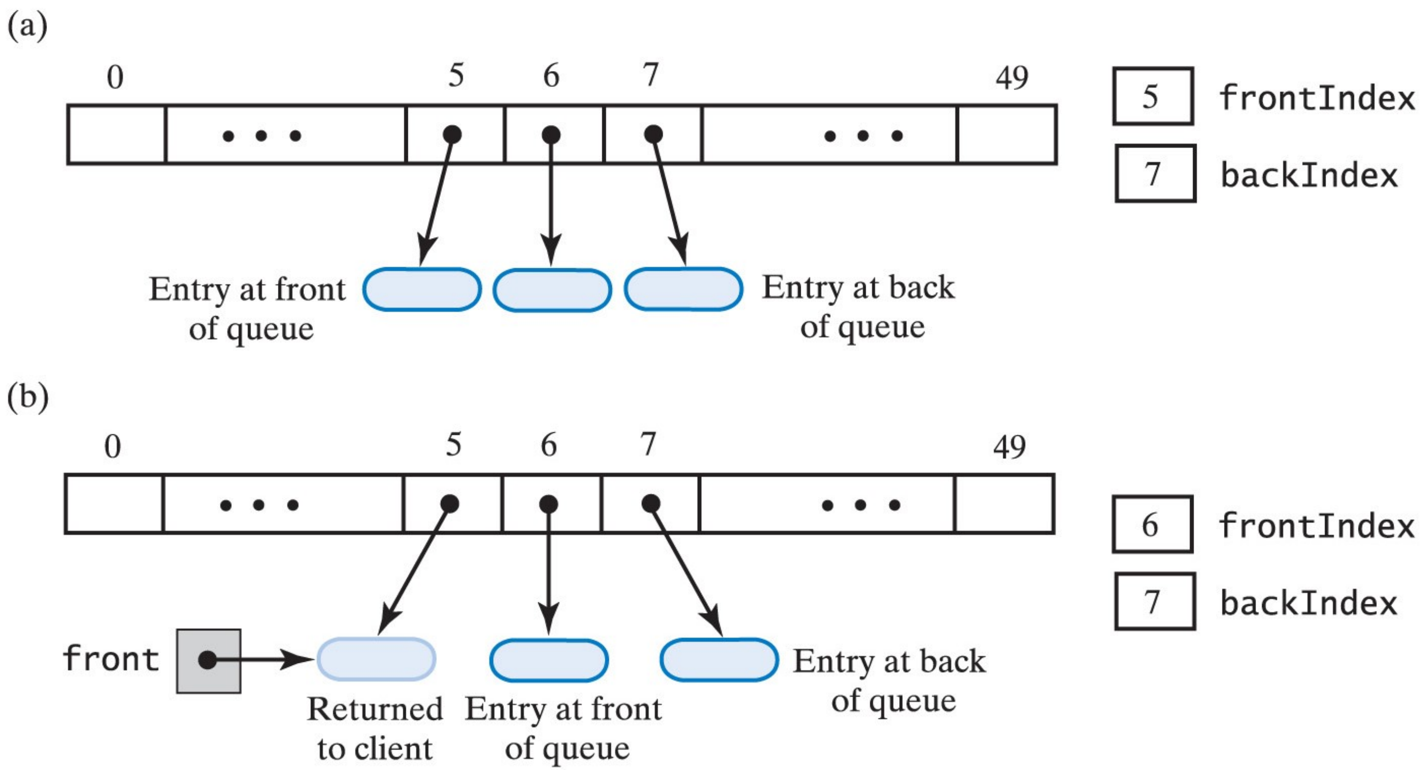


Fig. 24-9 An array-base queue: (a) initially; (b) after removing its front by incrementing **`frontIndex`**;

Circular Array Implementation of a Queue

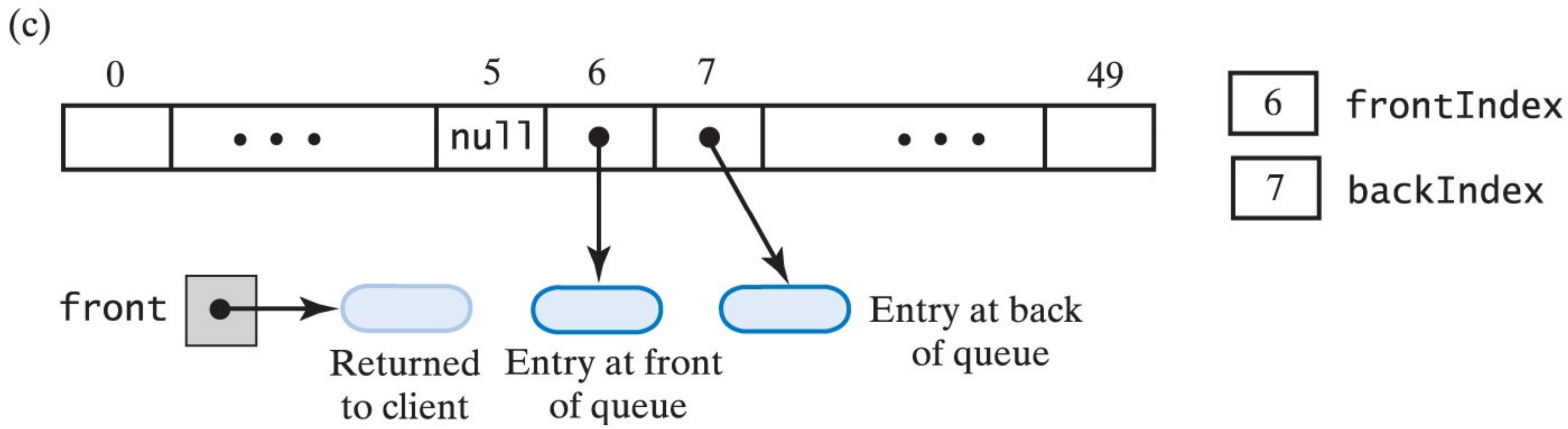


Fig. 24-9 An array-base queue: (c) after removing its front by setting **`queue[frontIndex]`** to **`null`**, then incrementing **`frontIndex`**.

Method 3: Circular Array

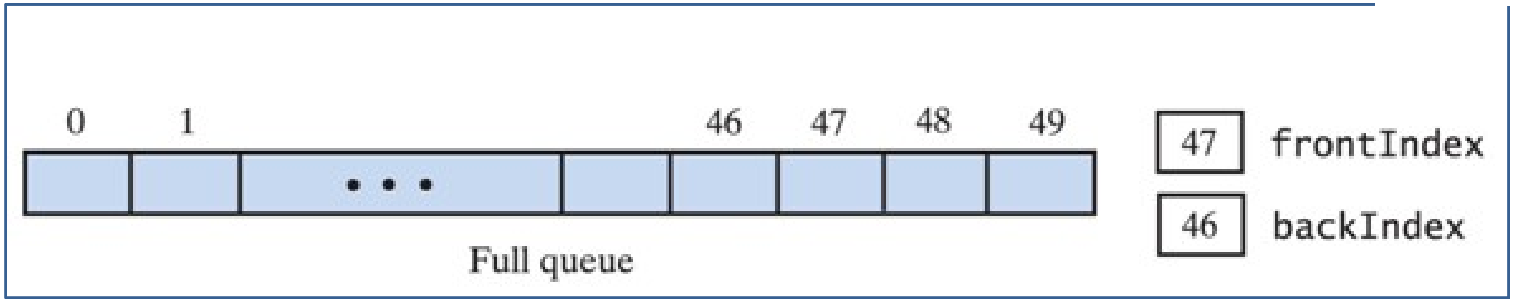
Strength and Weakness

- ☑ No rightward drift problem
  - – No wasted array locations
  - – Do not have to shift entries after the last array location is used
- ✘ Problem: *How to detect when the queue is empty and when the array is already full?*
  - – Note: with circular array

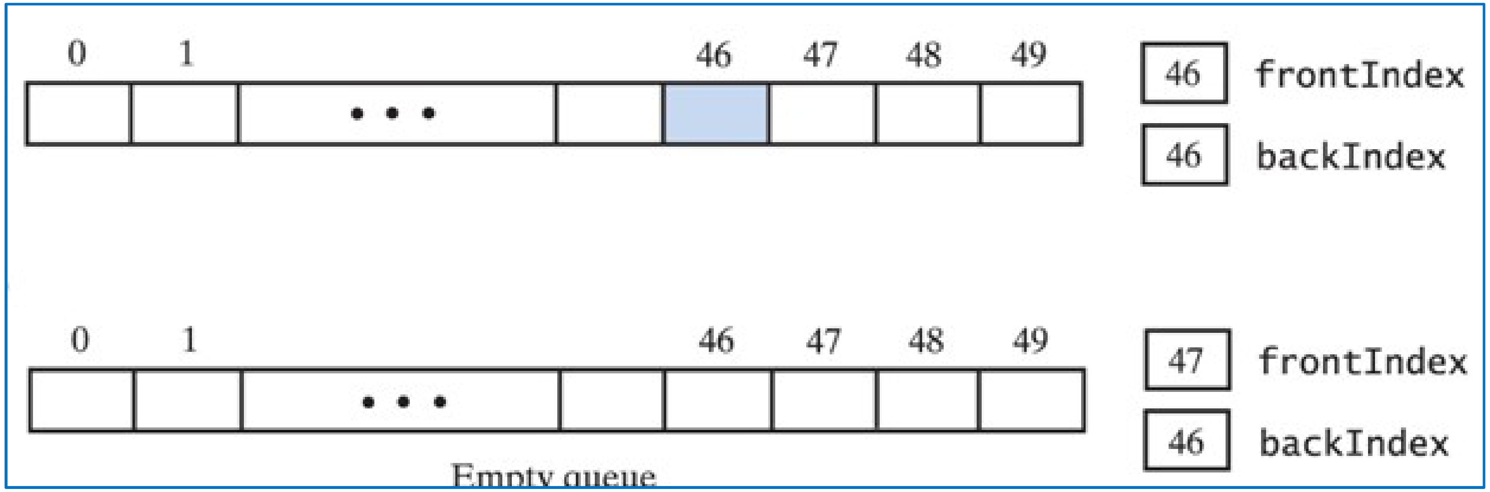
    `frontIndex == backIndex + 1`

    both when queue is empty and when full

Method 3: Circular Array



Observe that the relative positions of `frontIndex` and `backIndex` are the same for full queue (figure above) and empty queue (figure below) .



Solutions to Detect Empty and Full Queues

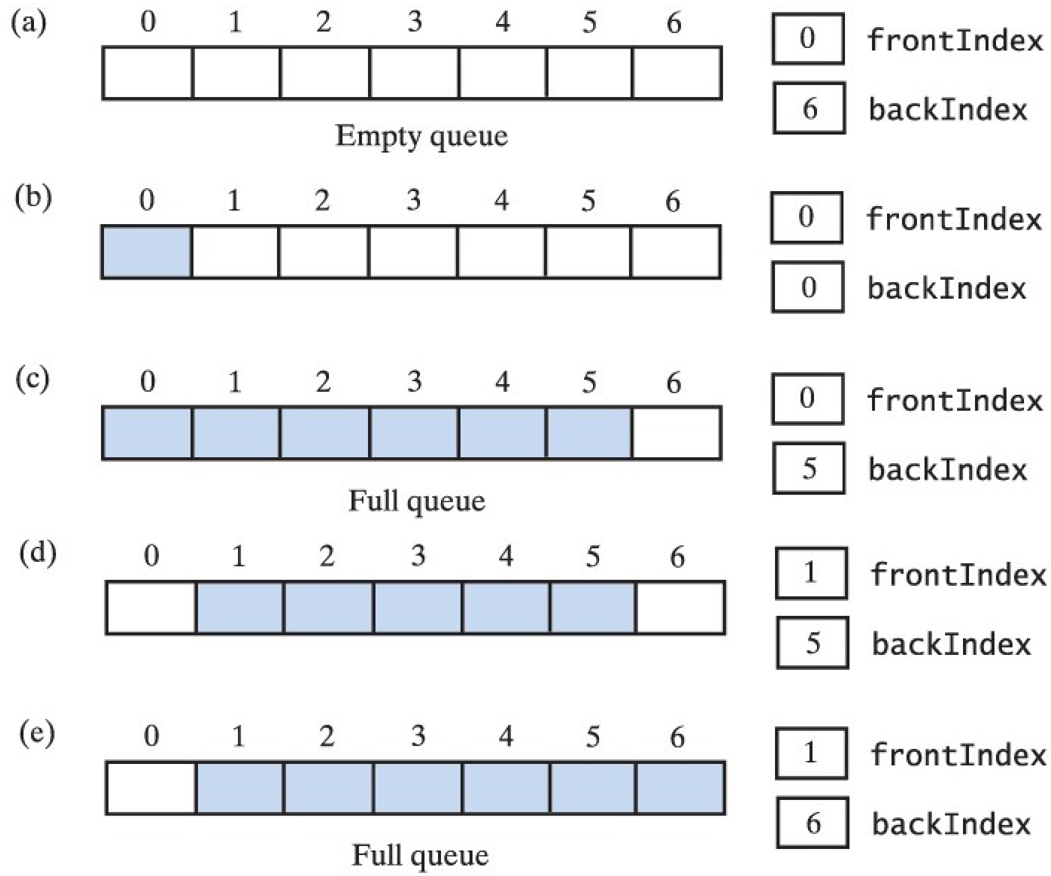
Method 3: Circular Array

1. Use a counter to keep track of the total entries in the queue
   – Empty queue detected when counter is 0
   – Full queue detected when counter equals array length
2. Leave one unused (vacant) location in the array
   – Empty queue detected when **`frontIndex`** is one location "in front" of **`backIndex`** (remember the wraparound action)
   – Full queue detected when only one vacant array location left.

A Circular Array with One Unused Location

Method 3: Circular Array

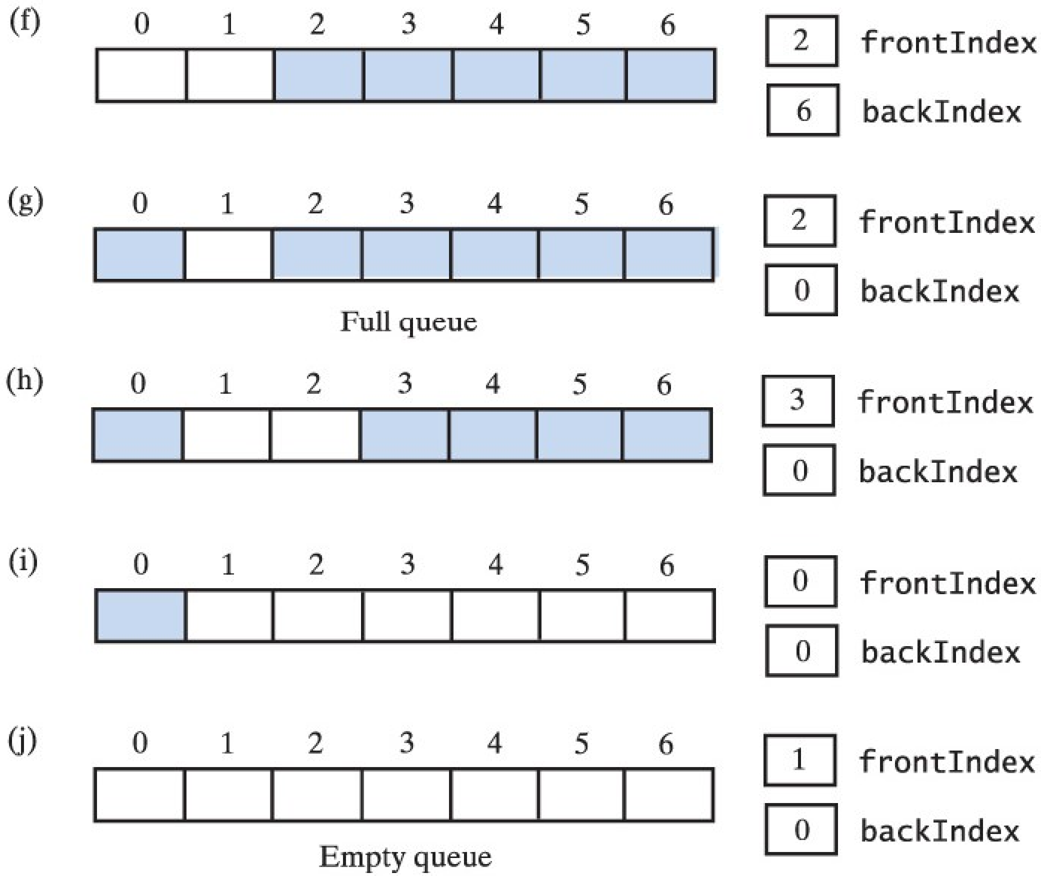
Fig. 24-8 A seven-location circular array that contains at most six entries of a queue … continued →



Allows us to distinguish between empty and full queue

A Circular Array with One Unused Location

Fig. 24-8 (ctd.) A seven-location circular array that contains at most six entries of a queue.



Method `isEmpty()`

Method 3: Circular Array

- The queue is empty if

```
(backIndex + 1) % array.length ==  frontIndex
```

Method 3: Circular Array

Implementation of method `doubleArray()` in a circular array

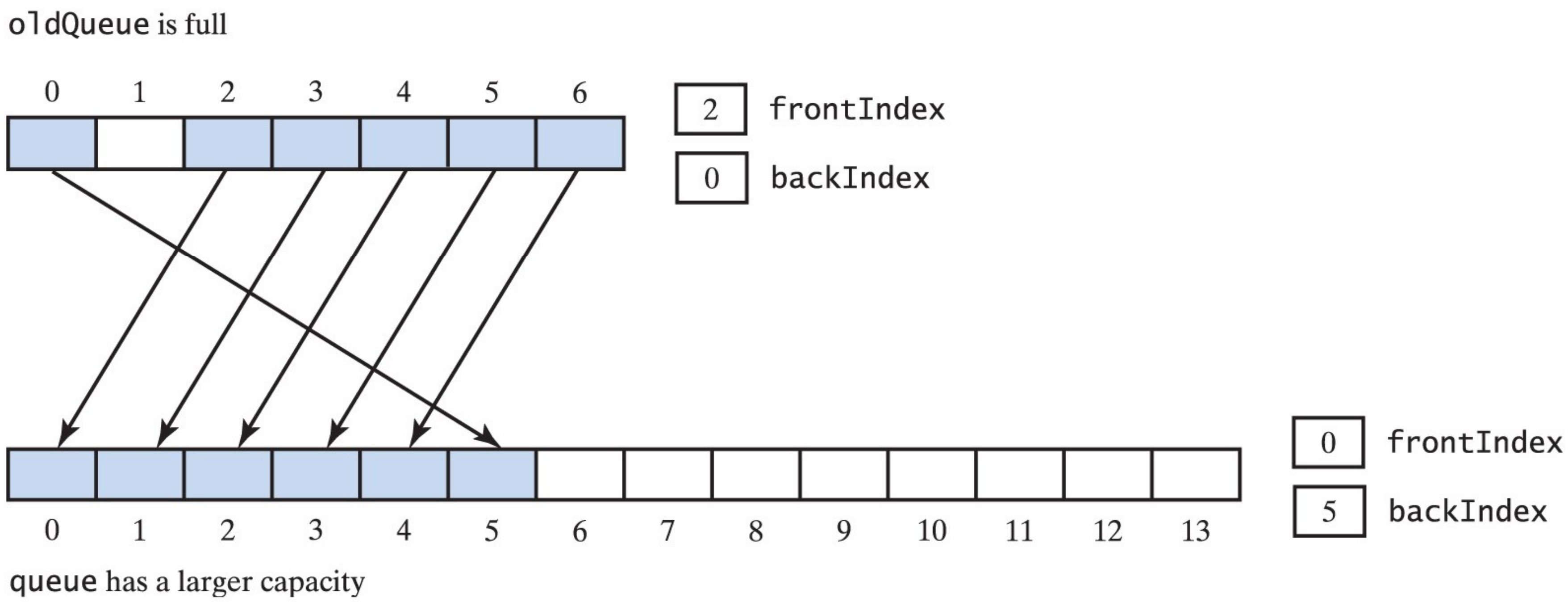


Fig. 24-10 Doubling the size of an array-based queue

Question

Queue ADT may be implemented using arrays in at least three different ways: linear array with a fixed front , linear array with a dynamic front and circular array

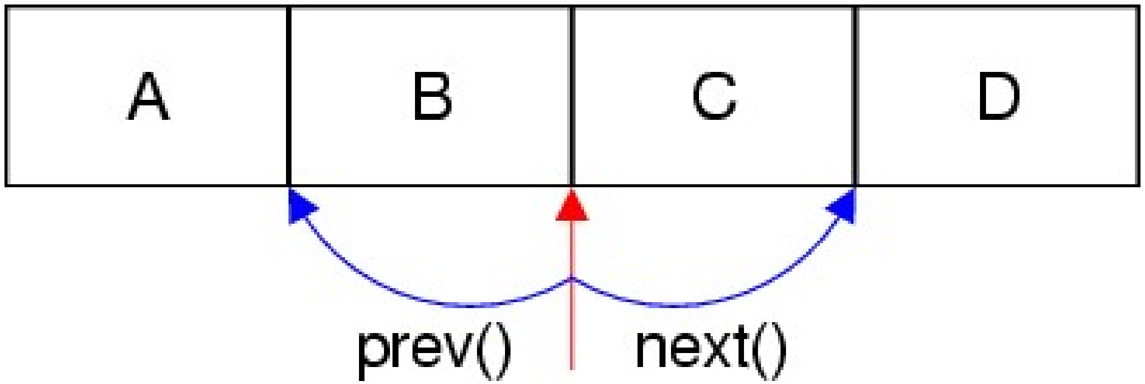
Compare & contrast *all three* of the above array implementations of queues in terms of

- The general idea behind each implementation
- How the *add* and *remove* operations are implemented in each approach
- How an *empty* queue and *full* queue is detected for each approach
- Advantages and disadvantages of each approach
- Solutions to the problems encountered with each approach

Sample Code

- `Chapter4\adt\`
  - `QueueInterface.java`
  - `ArrayQueue.java`
  - `CircularArrayQueue.java`
- `Chapter4\entity\`
  - `Customer.java`
  - `StockLedger.java`
  - `StockPurchase.java`
- `Chapter4\client\`
  - `SimulationDriver.java`
  - `StockLedgerDriver.java`
  - `WaitLine.java`

Iterators



Iterators (1)

- A typical process on a collection of entries is to go through its entries in order, one at a time, *e.g.*, to look for a specific entry.
- An iterator
  - is an object that enables you to *traverse* a collection of data, beginning with the first entry.
  - acts like a cursor or pointer, moving about on a data structure and locating individual elements for access.
  - is a software design pattern that abstracts the process of scanning through a collection of elements one element at a time.

Iterators (2)

- During one complete iteration, each entry is considered once
- Iterator may be manipulated
  - Check whether next entry exists
  - Asked to advance to next entry
  - Give a reference to current entry
  - Modify the list as you traverse it

Java's Iterator Interfaces

- As iteration is such a common operation, Java provides 2 interfaces for iterators for a uniform way for traversing elements in various types of collections:
  - `Iterator`
  - `ListIterator`
- These interfaces provide a uniform way for traversing elements in various types of collections.

  (Note: Collections include list, stack, queue, etc.)

`java.util.Iterator`

- This interface specifies a generic type for entries
- Includes 3 method headers:

`hasNext`	Checks if next entry exists .
`next`	Returns next entry and advances iterator to the next entry. Throws `NoSuchElementException` if there are no more elements.
`remove`	Removes the entry that was returned by the last call to **`next()`** .

<<interface>>

java.util.Iterator<T>

+hasNext(): boolean

+next(): T

+remove(): void

An Inner Class Iterator

- The iterator class is defined as an *inner class* of the ADT.
  - Thus, it has direct access to the ADT's data fields.

```
import java.util.Iterator;
public interface ListWithIteratorInterface < T >
                 extends ListInterface < T >
{
    public Iterator < T > getIterator ();
} // end ListWithIteratorInterface
```

- Note method **`getIterator`**
  - Enables the client to create an iterator
  - Includes a call to the inner class iterator's constructor

Sample Code

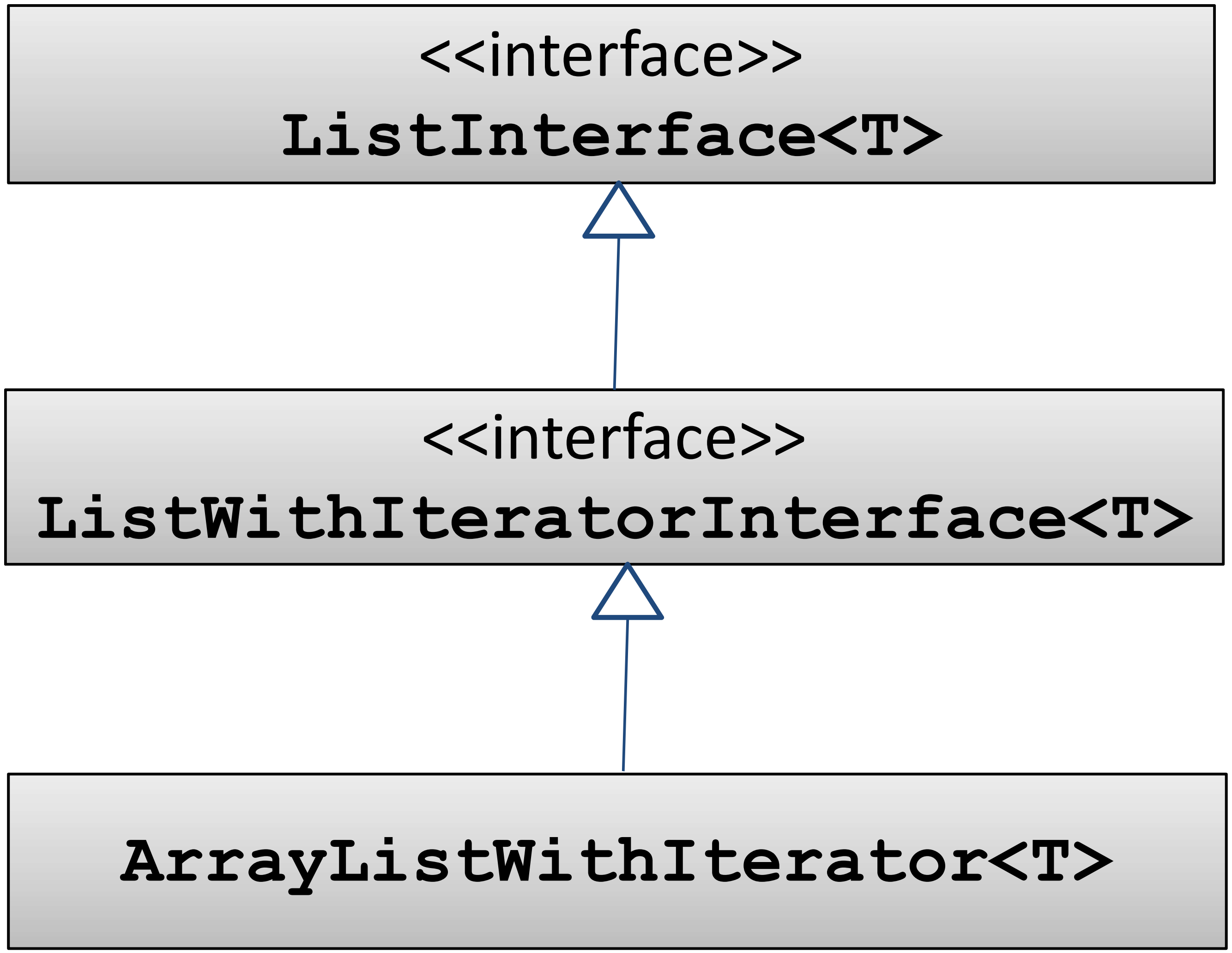
(a) **`Chapter4\adt\`**

- **`ListWithIteratorInterface.java`**:
  - An interface that extends the interface **`ListInterface`**.
  - Contains the abstract method **`getIterator()`** which returns an iterator to the list
- **`ArrayListWithIterator.java`**
  - A class that implements the interface **`ListWithIteratorInterface`**

(b) **`Chapter4\client\`**

- **`TestArrayListWIthIterator.java`**

UML Class Diagram for Example



Learning Outcomes

You should now be able to

- Describe the concept of recursion
- Solve a problem using recursion
- Trace a recursive method call
- Analyze the efficiency of a recursive solution as compared to other alternative solutions

References

- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson
- Malik DS and Nair PS, 2003, Data Structures using Java, Thomson Course Technology