GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 04
STACKS AND QUEUES

Kettering
UNIVERSITY

HIGHLIGHTS

ADT Stack

- Introduction, Operations, LIFO vs FIFO, Pseudocode Definition
- **Stack Implementations:** Array/Reference/List-Based, Comparison, JCF Stack
- Stacks and Recursion
- **Stack Applications**
  - Check Balanced Braces, Check String in Language, Evaluate Expressions
  - Convert Expressions, Find Path in Digraph

ADT Queue

- Introduction, Operations
- **Queue Implementations:** Reference/Array/List-Based, JCF Queue
- **Queue Applications:** Reading a String, Check Palindromes

Summary of Position-Oriented ADTs

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

- **Set 1:** ex. 1-10.
- **Set 2:** ex. 1-3, ex. 7-10, ex. 12-14.
- **Set 3:** ex. 1, ex. 4, ex. 7, ex. 10, ex. 13, ex. 16, ex. 19, ex. 22, ex. 25, ex. 28, ex. 31.

ADDITIONAL RESOURCES

- **"Object-Oriented Data Structures Using Java (4th Ed.)", chap. 2, chap. 4.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 7-8.
- "Java Illuminated (5th Ed.)", chap. 14.
- visualgo.net/en/list

ADT STACK - INTRODUCTION 1

The specifications of an Abstract Data Type (ADT) for a particular problem, can emerge during the design of the solution for that problem.

Example: The **read-and-correct** algorithm, or the **display-backward** algorithm are examples of problems that lead to specifying an ADT while solving the problem.

ADT STACK - INTRODUCTION 2

THE READ-AND-CORRECT ALGORITHM A

Consider the following line: $\mathbf{abcc \leftarrow ddde \leftarrow\leftarrow\leftarrow ef \leftarrow fg = abcdefg}$

If a left-arrow represents a backspace character, then the left line equals the right line.

Question: How can a program read the original line (left) and get the correct input?

To design a solution to this problem, you need to decide how to store the input line. However, accordingly with ADT approach, you should postpone this decision until you understand of what operations you will need to perform on the data.

ADT STACK - INTRODUCTION 3

THE READ-AND-CORRECT ALGORITHM B

A first attempt at a solution should lead to the following **required operations**:

- **add a new item** to the ADT,
- **remove** from the ADT the **item that was added most recently**, and
- determine whether the ADT **is empty**.

As illustrated by the following **pseudocode** of the read-and-correct algorithm:

```
// Read the line, correcting mistakes along the way.
while( not end of line ) {
   Read a new character ch;
   if( ch is not a '<-' ) { Add ch to the ADT; }
   else if( the ADT is not empty ) { Remove from the ADT the item added most recently; }
   else { Ignore the '<-'; } }
```

ADT STACK - INTRODUCTION 4

THE DISPLAY-BACKWARD ALGORITHM

Now, using the same ADT, consider the additional problem of **displaying the input line backward**.

It is easy to realize, that this time you need a **new operation**:

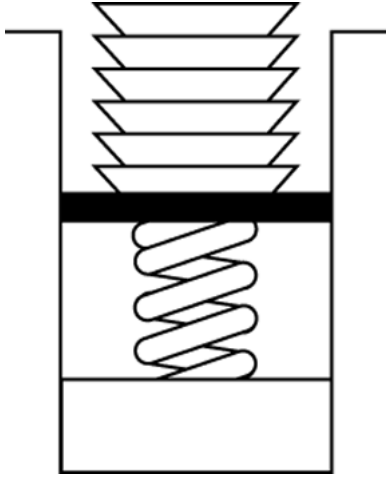
- **retrieve** from the ADT the **item that was added most recently**.

As illustrated by the following **pseudocode** of the display-backward algorithm:

```
// Write the line in reversed order.
while( the ADT is not empty ) {
   ch = Retrieve from the ADT the item that was added most recently;
   Display ch;
   Remove from the ADT the item added most recently. }
```

ADT STACK - OPERATIONS 1

The 4 operations that we need to solve our problems define the required ADT, which happens to be well known: it is usually called a **stack** (or pile).
This ADT works as a stack of cafeteria dishes.



THE ADT STACK: OPERATIONS

1. create an empty stack
2. determine whether a stack is empty
3. add a new item to the stack
4. remove from stack the item that was added most recently
5. remove all the items from the stack
6. retrieve from stack the item that was added most recently

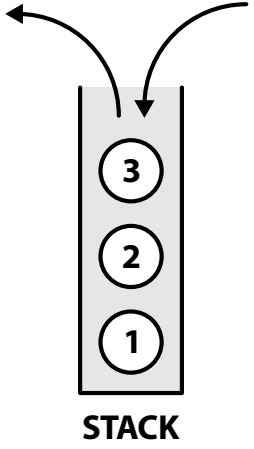
See: en.wikipedia.org/wiki/stack_(abstract_data_type)

ADT STACK - OPERATIONS 2

THE ADT STACK

A **stack** or **LIFO** (**last-in, first-out**) is an ADT that serves as a collection of elements, with two principal operations:

- **push:** adds an item to the top (last-in),
- **pop:** removes an item from the top (first-out).



Example: OpenGL matrix stacks.

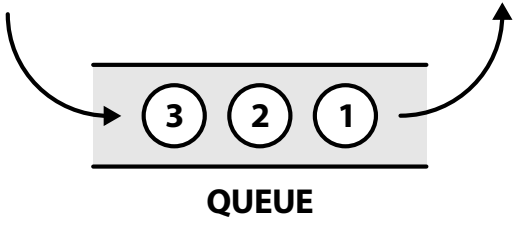
See: en.wikipedia.org/wiki/stack_(abstract_data_type)

ADT STACK - LIFO vs FIFO

THE ADT QUEUE

On the contrary, a **queue** uses a **FIFO** (**first-in, first-out**) strategy, with two principal operations:

- **enqueue:** adds an item to the rear (first-in),
- **dequeue:** removes an item from the front (first-out).



Example: FCFS (first-come, first-served) in OS scheduling

See: en.wikipedia.org/wiki/queue_(abstract_data_type)

ADT STACK - PSEUDOCODE DEFINITION

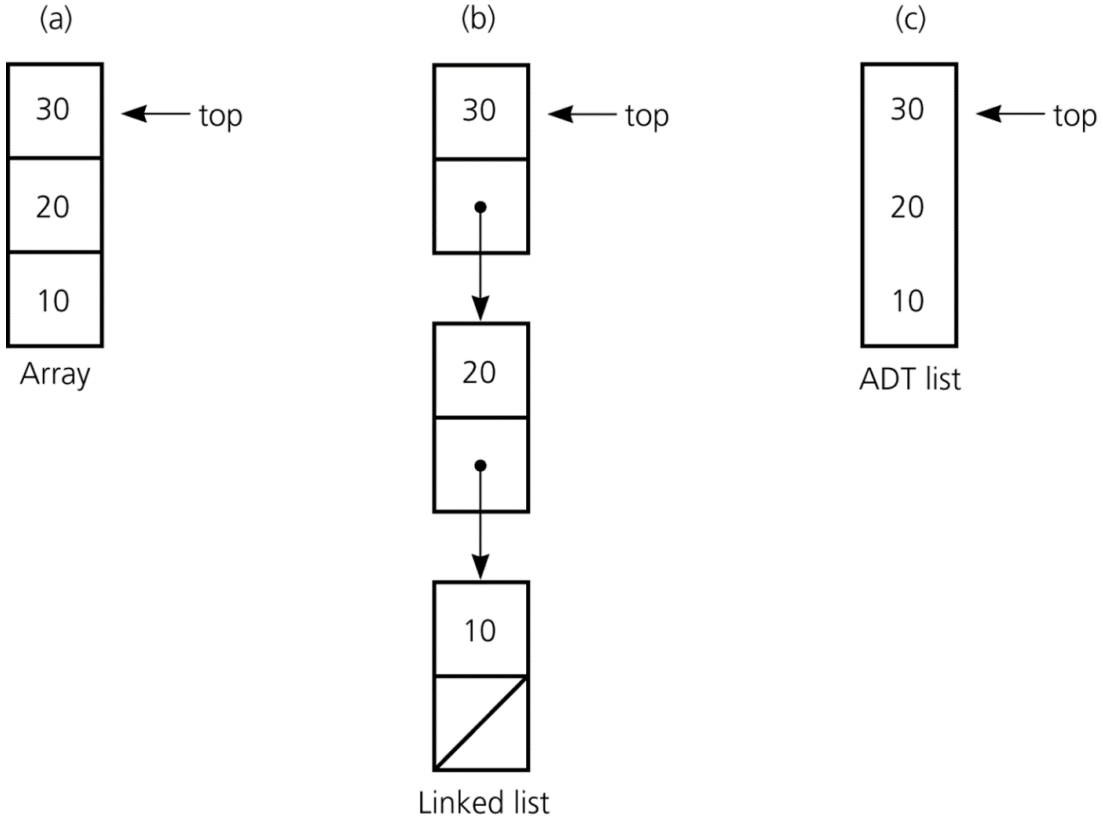
REFINING ADT STACK DEFINITION

The following **pseudocode** completes and details the ADT stack operations using their conventional names:

```
// StackItemType is the type of the items stored in the stack.

void createStack(); // Creates an empty stack.
boolean isEmpty(); // Determines whether a stack is empty.
void popAll(); // Removes all items from the stack.
void push( StackItemType i ) throws StackException; // Adds i to the top of the stack.
StackItemType pop() throws StackException; // Retrieves and removes top of the stack.
StackItemType peek() throws StackException; // Retrieves the top of the stack.
```

STACK IMPLEMENTATIONS - INTRODUCTION

ADT stacks implemented using: an array **(a)**, a linked list **(b)**, or an ADT list **(c)**.



STACK IMPLEMENTATIONS - INTERFACE

STACK INTERFACE

Provides a common specification for different ADT stack implementations.

```
// Interface providing a common specification for different ADT stack implementations.
public interface StackInterface {
   public boolean isEmpty(); // Determines whether the stack is empty.
   public void popAll(); // Removes all the items from the stack.

   // Adds an item to the top of the stack.
   // Throws StackException when newItem cannot be placed on the stack.
   public void push( Object newItem ) throws StackException;

   // Removes the top of a stack. Throws StackException if the stack is empty.
   public Object pop() throws StackException;

   // Retrieves the top of the stack. Throws StackException if the stack is empty.
   public Object peek() throws StackException; }
```

STACK IMPLEMENTATIONS - EXCEPTIONS

STACK EXCEPTION

Custom exception for the ADT stack (used by **StackInterface**), extending **java.lang.RuntimeException**.

```
import java.lang.RuntimeException;
import java.lang.String;
public class StackException extends java.lang.RuntimeException {
   public StackException( String s ) { super(s); }
}
```

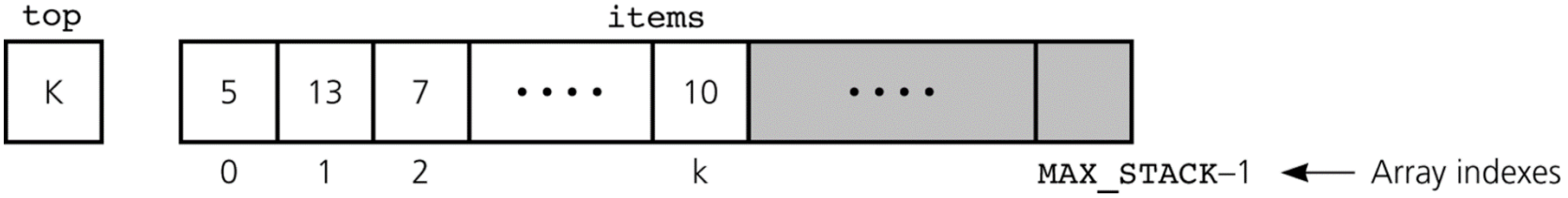
Note: Since **StackException** is a **RuntimeException** (i.e. non-critical), the calls to methods that throw **StackException** do not have to be enclosed in **try** blocks.

See: docs.oracle.com/javase/8/docs/api/java/lang/runtimeexception

STACK IMPLEMENTATIONS - ARRAY-BASED 1

The **StackArrayBased** class implements **StackInterface**, and includes the following private data fields:

- an array of elements of type **Object** called **items**,
- the index **top**.



STACK IMPLEMENTATIONS - ARRAY-BASED 2

```
public class StackArrayBased implements StackInterface {

   final int MAX_STACK = 50; // Max size (default access modifier: package-private).
   private Object items[];
   private int top;

   // Default constructor.
   public StackArrayBased() {
      items = new Object[ MAX_STACK ];
      top = -1;
   }

   public boolean isEmpty() { return top < 0; }

   // Note: isFull is not part of the StackInterface.
   public boolean isFull() { return top == ( MAX_STACK - 1 ); }

   // ...
```

STACK IMPLEMENTATIONS - ARRAY-BASED 3

```
// ...

public void push( Object newItem ) throws StackException {
   if( !isFull() ) { items[ ++top ] = newItem; }
   // Note: even if both X++ and ++X increment X by 1,
   //       X++ returns X before the increment, and ++X returns X after the increment.
   else { throw new StackException( "StackException on push: stack full!" ); }
}

public void popAll() { items = new Object[ MAX_STACK ]; top = -1; }

public Object pop() throws StackException {
   if( !isEmpty() ) { return items[ top-- ]; }
   else { throw new StackException( "StackException on pop: stack empty!" ); }
}

// ...
```

STACK IMPLEMENTATIONS - ARRAY-BASED 4

```
   // ...

   public Object peek() throws StackException {
      if( !isEmpty() ) { return items[ top ]; }
      else { throw new StackException( "StackException on peek: stack empty!" ); } }
}
```

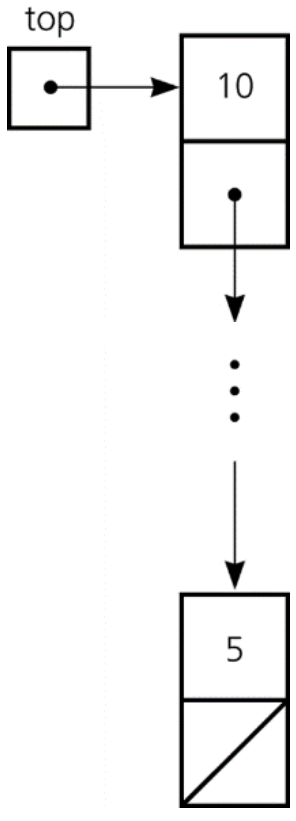
Note: Instances of the **StackArrayBased** cannot contain items of a primitive type (e.g. **int**), because primitive types are not derived from **Object**. So, if you need a stack of integers, you will have to use the corresponding wrapper class **Integer**.

Note: pop and **peek** return an item of type **Object**, so cast this item back to the subtype (e.g. **Integer**) used in the stack to use the methods of the subtype.

See: docs.oracle.com/javase/8/docs/api/java/lang/object

STACK IMPLEMENTATIONS - REFERENCE-BASED 1

A reference-based implementation is required when the stack needs to grow and shrink dynamically.



StackReferenceBased implements **StackInterface**, and in this case **top** is a reference to the head of a linked list of items.

STACK IMPLEMENTATIONS - REFERENCE-BASED 2

```
public class StackReferenceBased implements StackInterface {

   private Node top; // Reference at top node of the stack, or null.

   public StackReferenceBased() { top = null; }
   public boolean isEmpty() { return top == null; }
   public void push( Object newItem ) { top = new Node( newItem, top ); }
   public void popAll() { top = null; }

   public Object pop() throws StackException {
      if( !isEmpty() ) {
         Node temp = top;
         top = top.next;
         return temp.item; }
      else { throw new StackException( "StackException on pop: stack empty!" ); }
   }

   // ...
```

STACK IMPLEMENTATIONS - REFERENCE-BASED 3

```
   // ...

   public Object peek() throws StackException {
      if( !isEmpty() ) { return top.item; }
      else { throw new StackException( "StackException on peek: stack empty!" ); } }
}
```

Note: The implementation of the **StackReferenceBased** class uses the same **Node** class developed for the **ListReferenceBased** class.

Note: pop and **popAll** rely on the **Java Garbage Collector** to free the memory allocated for removed nodes, in fact both methods remove references to the removed nodes to mark them for garbage collection.

Note: As expected, the member variable **top** is completely hidden to the client.

STACK IMPLEMENTATIONS - LIST-BASED 1

The **ADT List** (implemented as the **ListReferenceBased** class)
can be used to represent the items in a stack.

If the item in position **0** of the ADT list represents the top of the ADT stack, then:

- **stack.push(item)** is implemented as: **list.add(0, item)**
- **stack.pop()** operation is implemented as: **list.get(0) + list.remove(0)**
- **stack.peek()** operation is implemented as: **list.get(0)**

STACK IMPLEMENTATIONS - LIST-BASED 2

```
import List.*; // List package including: ListReferenceBased class, and ListInterface.

public class StackListBased implements StackInterface {

   private ListInterface list; // list only needs to implement ListInterface interface!

   public StackListBased() { list = new ListReferenceBased(); } // list reference-based!
   public boolean isEmpty() { return list.isEmpty(); }
   public void push( Object newItem ) { list.add( 0, newItem ); }

   public Object pop() throws StackException {
      if( !list.isEmpty() ) {
         Object temp = list.get(0);
         list.remove(0);
         return temp; }
      else { throw new StackException( "StackException on pop: stack empty!" ); }
   }

   // ...
```

STACK IMPLEMENTATIONS - LIST-BASED 3

```
   // ...

   public void popAll() { list.removeAll(); }

   public Object peek() {
      if( !list.isEmpty() ) { return list.get(0); }
      else { throw new StackException( "StackException on peek: stack empty!" ); }
   }

}
```

Note: The data field **list** is declared using the **ListInterface**, but it is instantiated as a **ListReferenceBased** (that implements **ListInterface**) in the constructor of the **StackListBased** class.

Note: As expected, the member variable **list** is completely hidden to the client.

STACK IMPLEMENTATIONS - COMPARISON

The 3 implementations are substantially array-based or reference-based,
so the main properties to compare are **fixed-size versus dynamic-size**:

- **an array-based implementation** using fixed-sized arrays, prevents push from adding an item to the stack if the stack size limit has been reached;
- **a reference-based implementation** does not limit the size of the stack.
- **a list implementation** is less efficient than the reference-based implementation, but reuses code already implemented class, so it is simpler-quicker to code.

STACK IMPLEMENTATIONS - JCF STACK 1

The JCF contains an implementation of a stack called **Stack** (**generic**), derived from the **Vector** class (i.e. a growable array), and including: **peek**, **pop**, **push**, and **search**.

THE JCF STACK

```
public class Stack<E> extends Vector<E> {
   public Stack();
   public boolean empty();
   public E peek() throws EmptyStackException;
   public E pop() throws EmptyStackException;
   public E push( E item ); // Return value is the input item.
   public int search( Object o ); // Returns 1-based pos of o (comparison using equals).
}
```

See: docs.oracle.com/javase/8/docs/api/java/util/stack
See: docs.oracle.com/javase/8/docs/api/java/util/vector

STACK IMPLEMENTATIONS - JCF STACK 2

EXAMPLE OF USAGE OF THE JCF STACK

```
import java.util.Stack;
public class TestStack {
   static public void main( String[] args ) {
      Stack<Integer> aStack = new Stack<Integer>();
      if( aStack.empty() ) { System.out.println( "The stack is empty." ); }
      for( int i = 0; i < 5; i++ ) { aStack.push(i); } // Autoboxing!
      while( !aStack.empty() ) { System.out.print( aStack.pop() + " " ); } } }
```

AUTOBOXING

Java compiler automatic conversion of a primitive type into its corresponding object wrapper class (see also **unboxing**).

See: docs.oracle.com/javase/tutorial/java/data/autoboxing

STACKS AND RECURSION 1

Stacks are linked to the concept of recursion. Consider the following analogies:

- **Box Trace and Stack:** box trace of recursive directed path search, and the stack storing visited cities;
- **Backtracking:** navigating the box trace backward, or popping from the stack;
- **Termination:** detected when all the box trace boxes have been crossed off, or when the stack is empty.

These similarities are far more than coincidence. In fact:

You can always capture the actions of a recursive method by using a stack.

STACKS AND RECURSION 2

Typically, **stacks are used by compilers to implement recursive methods** (in a way similar to the box trace) that can be summarized as follows:

- **Recursive Calls:** during execution, each recursive call generates an **activation record** (recursive call local environment, point of return of the execution etc.) that is pushed onto a stack (i.e. the **call stack**).

- **Returning from a Recursive Call:** the stack is popped, bringing the activation record containing the proper local environment etc., on top of the stack.

See: en.wikipedia.org/wiki/call_stack

Finally, **stacks can be used to implement a non-recursive version of a recursive algorithm**, for example to improve its efficiency.

STACK APPLICATIONS - BALANCED BRACES 1

CHECKING FOR BALANCED BRACES: PROBLEM

A stack can be used to verify whether a program contains balanced braces:

- **Balanced Braces:** abc { defg { ijk } { l { mn } } op } qr
- **Unbalanced Braces:** abc { def } } { ghij { kl } m

CHECKING FOR BALANCED BRACES: REQUIREMENTS

1. Each time you encounter a **}**, it matches an already encountered **{**.
2. When you reach the end of the string, you have matched each **{**.

The solution requires that you keep track of each unmatched **{** and discard one each time you encounter a **}**. One way to perform this task is to push each **{** encountered onto a stack and pop one off each time you encounter a **}**.

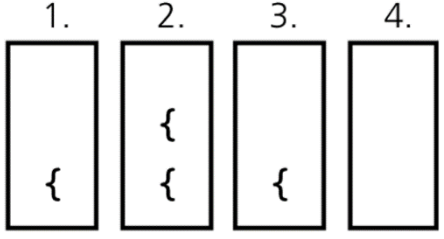
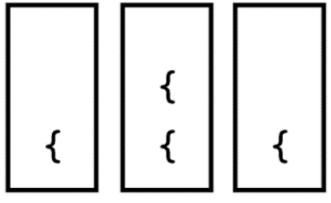
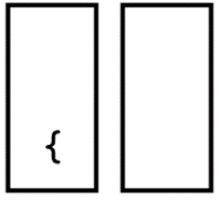
STACK APPLICATIONS - BALANCED BRACES 2

CHECKING FOR BALANCED BRACES: PSEUDOCODE OF THE ALGORITHM

```
// aString is the input string to be checked for balanced braces.
aStack.createStack();
boolean balancedSoFar = true;
int i = 0;
while( balancedSoFar && ( i < aString.length() ) ) {
   StackItemType ch = character at position i in aString;
   i++;
   if( ch == '{' ) { aStack.push( ch ); } // Push an open brace onto the stack.
   else if( ch == '}' ) { // Check if ch is a close brace.
      if( !aStack.isEmpty() ) {
      StackItemType openBrace = aStack.pop(); } // Pop a matching open brace from stack.
      else { balancedSoFar = false; } } // No matching open brace in the stack.
   // Ignoring all characters other than braces.
}
if( balancedSoFar && aStack.isEmpty() ) { aString has balanced braces; }
else { aString does not have balanced braces; }
```

STACK APPLICATIONS - BALANCED BRACES 3

CHECKING FOR BALANCED BRACES: TRACE OF THE ALGORITHM

Input string	Stack as algorithm executes	
`{a{b}c}`		1. push "{" 2. push "{" 3. pop 4. pop Stack empty ⟹ balanced
`{a{bc}`		1. push "{" 2. push "{" 3. pop Stack not empty ⟹ not balanced
`{ab}c}`		1. push "{" 2. pop Stack empty when last "}" encountered ⟹ not balanced

STACK APPLICATIONS - STRING IN LANGUAGE 1

RECOGNIZING STRINGS IN A LANGUAGE: PROBLEM

Given the language **L**:

$$L = \{\, w\$w' : w \text{ is a string of characters other than } \$,\ w' = reverse(w) \,\}$$

A stack can be used to determine whether a given string is in **L**:

1. traverse the first half of the string, **pushing** each character onto a stack;
2. once you reach the **$**, for each character in the second half of the string, **pop** a character off the stack;
   a. match the popped character with the current character in the string;
3. the stack must be empty when, and only when, you reach the end of the string, otherwise the string is not in **L**.

STACK APPLICATIONS - STRING IN LANGUAGE 2

RECOGNIZING STRINGS IN A LANGUAGE: PSEUDOCODE OF THE ALGORITHM

```
// aString is the input string to be checked as part of the L language.
aStack.createStack(); int i = 0; StackItemType ch = character at position i in aString;
while( ch != '$' ) { // Push characters before $ (characters in w), onto the stack.
   aStack.push( ch );
   i++;
   ch = character at position i in aString; }
i++; // Skip the $.
boolean inLanguage = true; // Assume aString is in language L.
while( inLanguage && ( i < aString.length() ) ) { // Match the reverse of w.
   ch = character at position i in aString;
   try {
      StackItemType stackTop = aStack.pop();
      if( stackTop == ch ) { i++; } // Characters match.
      else { inLanguage = false; } } // Characters do not match.
   catch( StackException e ) { inLanguage = false; } } // Pop failed.
if( inLanguage && aStack.isEmpty() ) { aString is in language L; }
else { aString is not in language L; }
```

STACK APPLICATIONS - EVALUATE EXPRESSIONS 1

EVALUATION OF INFIX EXPRESSIONS: STRATEGY

To evaluate an infix expression:

1. convert the infix expression to postfix form, and
2. evaluate the postfix expression.

EVALUATION OF POSTFIX EXPRESSIONS: IMPLEMENTATION USING A STACK A

To evaluate a postfix expression (i.e. a string) we need to simplify the problem:

- the string is a syntactically **correct postfix expression**,
- **no unary operators** are present,
- **no exponentiation operators** are present, and
- **operands are single lowercase letters that represent integer values**.

STACK APPLICATIONS - EVALUATE EXPRESSIONS 2

EVALUATION OF POSTFIX EXPRESSIONS: IMPLEMENTATION USING A STACK B

A postfix calculator requires you to enter expressions in postfix form (e.g. 2 3 4 + *):

1. when an **operand** is entered,

    a. the calculator **pushes it onto a stack**;

2. when an **operator** is entered,

    a. the calculator **applies it to the top two operands of the stack**,
    b. then **pops the operands from the stack**, and
    c. finally **pushes the result of the operation on the stack**.

STACK APPLICATIONS - EVALUATE EXPRESSIONS 3

EVALUATION OF POSTFIX EXPRESSIONS: IMPLEMENTATION USING A STACK C

Actions of a **postfix calculator** evaluating the expression: **2 3 4 + *** using a stack.

Key entered	Calculator action		Stack (bottom to top)
2	`push 2`		2
3	`push 3`		2 3
4	`push 4`		2 3 4
+	`operand2 = pop stack`	(4)	2 3
	`operand1 = pop stack`	(3)	2
	`result = operand1 + operand2`	(7)	2
	`push result`		2 7
*	`operand2 = pop stack`	(7)	2
	`operand1 = pop stack`	(2)	
	`result = operand1 * operand2`	(14)	
	`push result`		14

STACK APPLICATIONS - EVALUATE EXPRESSIONS 4

EVALUATION OF POSTFIX EXPRESSIONS: IMPLEMENTATION USING A STACK D

```
// Pseudocode of a stack-based postfix calculator.
for( each character ch in the string ) {
   if( ch is an operand ) {
      int val = convert ch into the relative operand val;
      aStack.push(val); }
   else {
      // ch is an operator named op.
      Operator op = convert ch into the relative operator op;
      // Evaluate op and push the result onto the stack.
      int operand2 = aStack.pop();
      int operand1 = aStack.pop();
      int result = operand1 op operand2; // Evaluating operator op.
      aStack.push(result); } }
```

Note: Once the algorithm terminates, the **final result is on the top of the stack**.

STACK APPLICATIONS - CONVERT EXPRESSIONS 1

CONVERSION OF INFIX EXPRESSIONS TO POSTFIX FORM

An infix expression can be easily converted into an equivalent postfix expression:

1. operands always stay in the same order with respect to one another,
2. an operator will move only to the right with respect to the operands, and
3. all parentheses are removed.

Note: a compiler usually performs this conversion to evaluate expressions in code!

CONVERSION OF INFIX EXPRESSIONS TO POSTFIX FORM: STRATEGY A

To convert an input infix expression in an equivalent postfix expression **postfixExp**:

1. if input character is an **operand**, append it to the output string **postfixExp**;
2. if input character is a **(** , push it onto the **stack**;

STACK APPLICATIONS - CONVERT EXPRESSIONS 2

CONVERSION OF INFIX EXPRESSIONS TO POSTFIX FORM: STRATEGY B

3. if input character is an **operator op**:
    a. if the stack is empty, push the operator onto the **stack**;
    b. if the stack is not empty:
        i. pop operators of greater or equal precedence from the stack, and append them to **postfixExp** (stop if you encounter a **(** or an operator of lower precedence or if the stack becomes empty);
        ii. push the **operator op** onto the stack;
4. if input character is a **)**:
    a. pop operators off the stack, and append them to **postfixExp** until you encounter the matching **(**;
5. if you reach the **end of the string**, append the rest of the stack to **postfixExp**.

STACK APPLICATIONS - CONVERT EXPRESSIONS 3

CONVERSION OF INFIX EXPRESSIONS TO POSTFIX FORM: TRACE EXAMPLE

Trace of the algorithm that converts infix expression **a - (b + c * d) / e** to postfix form.

`ch`	stack (bottom to top)	`postfixExp`	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	**`postfixExp`** until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to **`postfixExp`**

STACK APPLICATIONS - CONVERT EXPRESSIONS 4

CONVERSION OF INFIX EXPRESSIONS TO POSTFIX FORM: PSEUDOCOSE A

```
// Pseudocode of a stack-based infix-to-postfix (string) expression converter.
for( each character ch in the infix expression ) {
   switch(ch) {
      // Append operand to the end of postfix expression.
      case operand : { postfixExp = postfixExp + ch; break; }
      // Save '(' on the stack.
      case '(' : { aStack.push(ch); break; }
      // Pop stack until we find the matching '('.
      case ')' : {
         while( top of the stack != '(' ) { postfixExp = postfixExp + aStack.pop(); }
         openParenthesis = aStack.pop(); // Remove the open parenthesis.
         break; }

   // ...
```

STACK APPLICATIONS - CONVERT EXPRESSIONS 5

CONVERSION OF INFIX EXPRESSIONS TO POSTFIX FORM: PSEUDOCOSE B

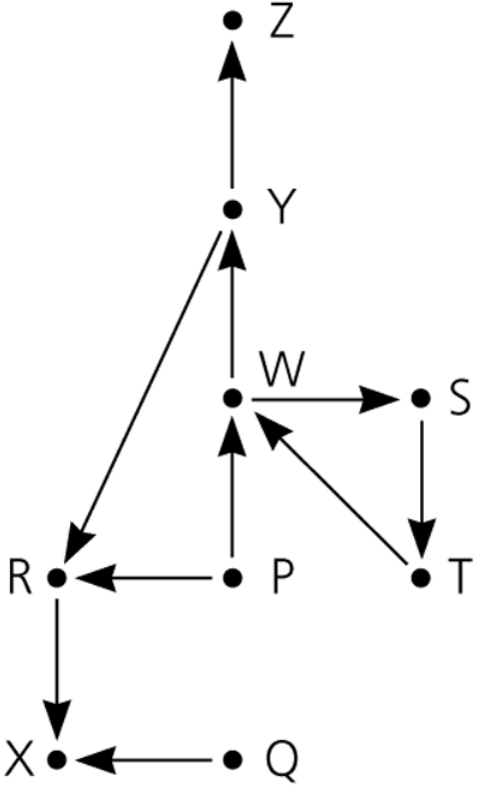
```
      // ...

      // Process stack operators of greater precedence.
      case operator : {
         while( !aStack.isEmpty() &&
                ( top of the stack != '(' ) &&
                ( precedence(ch) <= precedence( top of the stack ) ) ) {
            postfixExp = postfixExp + aStack.pop(); }
            aStack.push(ch); // Store the new operator on the stack.
         break; } } }
// Append to postfix expression the operators remaining in the stack.
while( !aStack.isEmpty() ) { postfixExp = postfixExp + aStack.pop(); }
```

STACK APPLICATIONS - FIND PATH IN DIGRAPH 1

The following problem is just a version of the problem "find a path in a digraph".

PATHPLANNING FOR AIRLINE COMPANY A

An The High Planes Airline Company (HPAir) has this problem:
for each customer request, indicate whether a sequence of HPAir flights exists from origin city to destination city.



The **flight map** for HPAir is a **directed graph**, where:

- **adjacent nodes** are nodes linked by a **directed link**,
- a **directed path** is a sequence of directed links.

We will solve this problem first by using **stacks**, and then by using **recursion**.

STACK APPLICATIONS - FIND PATH IN DIGRAPH 2

PATHPLANNING FOR AIRLINE COMPANY B

A **non-recursive solution that uses a stack** performs an exhaustive search beginning at the origin city, and then checks every possible directed path until either:

- it finds a directed path that gets to the destination city, or
- it determines that no such directed path exists.

In this case, the **ADT stack is useful in organizing the exhaustive search**.

Backtracking can be used to recover from a wrong choice of a city, but it requires to store information about the order in which cities have been visited during the search.

We implement backtracking storing the sequence of visited cities in a stack:

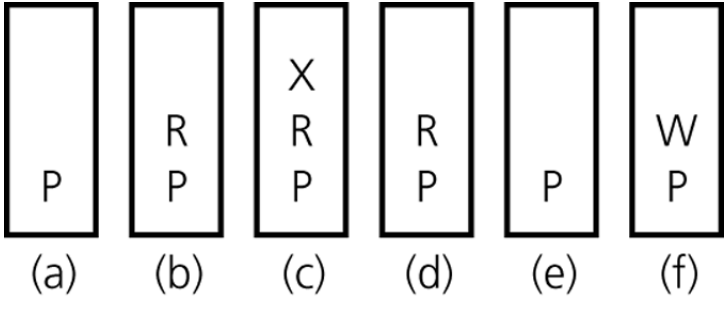
- each time we visit a city, we **push** the city onto the stack;
- we select the next city to visit from those adjacent to the **top** of the stack;
- if we need to backtrack, we simply **pop** a city from the stack.

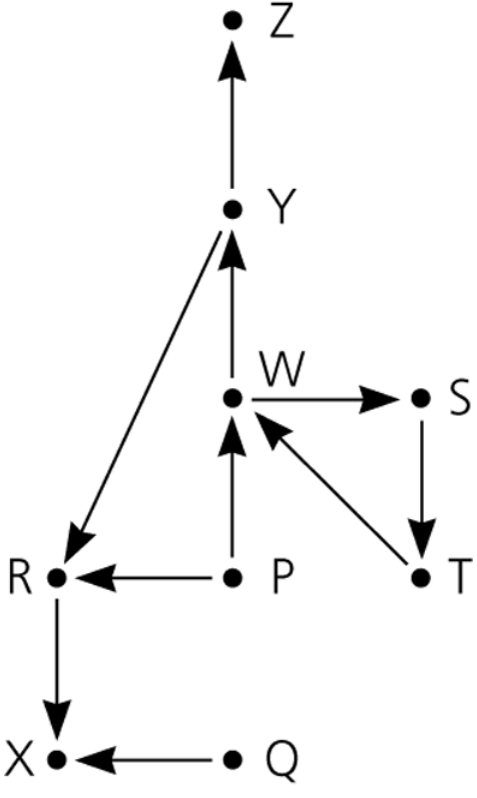
STACK APPLICATIONS - FIND PATH IN DIGRAPH

PATHPLANNING FOR AIRLINE COMPANY

The stack of cities as you travel from **P** to **W**:

- from **P (a)**,
- to **R (b)**,
- to **X (c)**,
- (backtracking) back to **R (d)**,
- (backtracking) back to **P (e)**, and
- to **W (f)**.





STACK APPLICATIONS - FIND PATH IN DIGRAPH 4

PATHPLANNING FOR AIRLINE COMPANY D

```
// Pseudocode of the search algorith with backtracking (version 1).
aStack.createStack();
aStack.push(origin); // Push origin city onto the stack.
while( a directed path from origin to destination has not been found ) {
   // Backtracking.
   if( you need to backtrack from city on top of the stack ) { temp = aStack.pop(); }
   else {
      Select a city C adjacent to the city on top of the stack;
      aStack.push(C); } }
```

Question: When we need to backtrack?

Observation: We don't want to visit a city that we have already visited (marked)!

Question: How we can understand if a directed path has been found?

Observation: Once all the possibilities have been exhausted, the stack will be empty!

STACK APPLICATIONS - FIND PATH IN DIGRAPH 5

PATHPLANNING FOR AIRLINE COMPANY E

```
// Pseudocode of the search algorith with backtracking (version 2).
aStack.createStack();
Clear marks on all cities; // Init all cities as non-visited.
aStack.push(origin); // Push origin city onto the stack.
Mark the origin; // Set the origin as visited.
while( !aStack.isEmpty() && ( destination is not on the top of the stack ) ) {
   if( we cannot find an adjacent city to the top of the stack that is not marked ) {
      temp = aStack.pop(); } // Backtracking.
   else {
      Select a non-marked city C adjacent to the top of the stack;
      aStack.push(C);
      Mark city C; } }
// Check if a directed path from origin to destination has been found.
if( aStack.isEmpty() ) { return false; } // No directed path exists.
else { return true; } // A directed path has been found.
```

STACK APPLICATIONS - FIND PATH IN DIGRAPH 6

PATHPLANNING FOR AIRLINE COMPANY F

Trace of the search algorithm for the flight map given.

Action	Reason	Contents of stack (bottom to top)
Push P	Initialize	P
Push R	Next unvisited adjacent city	P R
Push X	Next unvisited adjacent city	P R X
Pop X	No unvisited adjacent city	P R
Pop R	No unvisited adjacent city	P
Push W	Next unvisited adjacent city	P W
Push S	Next unvisited adjacent city	P W S
Push T	Next unvisited adjacent city	P W S T
Pop T	No unvisited adjacent city	P W S
Pop S	No unvisited adjacent city	P W
Push Y	Next unvisited adjacent city	P W Y
Push Z	Next unvisited adjacent city	P W Y Z

STACK APPLICATIONS - FIND PATH IN DIGRAPH 7

PATHPLANNING FOR AIRLINE COMPANY G

Now we know the operations required by the search algorithm on the flight map, so we can design an **ADT Flight Map**:

```
// Operations of the ADT for the flight map.
void createFlightMap(); // Creates an empty flight map.
void readFlightMap( String fileCities, String fileLinks ); // Init the flight map.
void displayFlightMap(); // Displays flight information.
void displayAllCities(); // Displays the names of all cities in the flight map.
void displayAdjacentCities( City c ); // Displays all cities adjacent to input city c.
void markVisited( City c ); // Marks a city as visited.
void unmarkAll(); // Clears mark on all cities.
boolean isVisited( City c ); // Determines whether a city was visited.
void insertAdjacent( City c, City a ); // Inserts city a adjacent to city c.
City getNextCity( City c ); // Returns next unvisited city adjacent to input city c.
boolean isPath( City o, City d ); // Search a directed path from city o to city d.
```

STACK APPLICATIONS - FIND PATH IN DIGRAPH 8

PATHPLANNING FOR AIRLINE COMPANY H

The **iterative implementation** of the search algorithm for the **ADT Flight Map**.

```
// ADT Flight Map method implementing the search for a directed path (iterative version).
public boolean isPathIterative( City orig, City dest ) {
   StackReferenceBased stack = new StackReferenceBased(); City top, next; // Variables.
   unmarkAll(); // Clear marks on all the cities.
   stack.push(orig); markVisited(orig); // Push origin on stack and mark it as visited.
   top = (City) ( stack.peek() );
   while( !stack.isEmpty() && ( top.compareTo(dest) != 0 ) ) {
      next = getNextCity(top);
      if( next == null ) { stack.pop(); } // No city found, backtrack.
      else { stack.push(next); markVisited(next); }
      top = (City) ( stack.peek() ); }
   if( stack.isEmpty() ) { return false; } // No directed path orig-dest exists.
   else { return true; } // A directed path between orig and dest has been found.
}
```

STACK APPLICATIONS - FIND PATH IN DIGRAPH 9

PATHPLANNING FOR AIRLINE COMPANY

The **recursive implementation** of the search algorithm for the **ADT Flight Map**.

```
// ADT Flight Map method implementing the search for a directed path (recursive version).
public boolean isPathRecursive( City orig, City dest ) {
   City next; boolean done; // Variables.
   markVisited(orig); // Mark origin city as visited.
   // Base case: the destination has been reached.
   if( orig.compareTo(dest) == 0 ) { return true }
   else { // Recursive case: try a flight to each unvisited city.
      done = false;
      next = getNextCity(orig);
      while( next != null && !done ) {
         done = isPathRecursive( next, dest );
         if( !done ) { next = getNextCity(orig); } }
      return done; }
}
```

ADT QUEUE - INTRODUCTION

THE ADT QUEUE

A **queue** or **FIFO** (**first-in, first-out**) is an ADT that serves as a collection of items, where the first item inserted is the first item to leave, and with 2 main operations:

- **enqueue** adds an item to the rear (first-in),
- **dequeue** removes an item from the front (first-out).

Example: FCFS (first-come, first-served) in OS scheduling

See: en.wikipedia.org/wiki/queue_(abstract_data_type)

ADT QUEUE - OPERATIONS 1

1. Create an empty queue.
2. Determine whether a queue is empty.
3. Add a new item to the queue.
4. Remove from the queue the item that was added earliest.
5. Remove all the items from the queue.
6. Retrieve from the queue the item that was added earliest.

```
// QueueItemType is the type of items stored in the queue.
void createQueue(); // Creates an empty queue.
boolean isEmpty(); // Determines whether a queue is empty.
void enqueue( QueueItemType i ) throws QueueException; // Adds i at the back of a queue.
QueueItemType dequeue() throws QueueException; // Retrieves and removes front of a queue.
void dequeueAll(); // Removes all items from a queue.
QueueItemType peek() throws QueueException; // Retrieves the front of a queue.
```

ADT QUEUE - OPERATIONS 2

An example of how to use an ADT queue and its operations.

Operation	Queue after operation
	Front ←
`queue.createQueue()`	
`queue.enqueue(5)`	5
`queue.enqueue(2)`	5 2
`queue.enqueue(7)`	5 2 7
`queueFront = queue.peek()`	5 2 7 (`queueFront` is 5)
`queueFront = queue.dequeue()`	5 2 7 (`queueFront` is 5)
`queueFront = queue.dequeue()`	2 7 (`queueFront` is 2)

QUEUE IMPLEMENTATIONS - INTERFACE

QUEUE INTERFACE

Provides a common specification for different ADT queue implementations.

```
// Interface providing a common specification for different ADT queue implementations.
public interface QueueInterface {
   public boolean isEmpty(); // Determines whether the queue is empty.
   public void dequeueAll(); // Removes all items of a queue.

   // Adds an item at the back of the queue.
   public void enqueue( Object newItem ) throws QueueException;

   // Retrieves and removes the front of the queue.
   public Object dequeue() throws QueueException;

   // Retrieves the top of the queue. Throws QueueException if the queue is empty.
   public Object peek() throws QueueException;
}
```

QUEUE IMPLEMENTATIONS - EXCEPTIONS

QUEUE EXCEPTION

Custom exception for the ADT queue (used by **QueueInterface**), extending **java.lang.RuntimeException**.

```
import java.lang.RuntimeException;
import java.lang.String;
public class QueueException extends java.lang.RuntimeException {
   public QueueException( String s ) { super(s); }
}
```

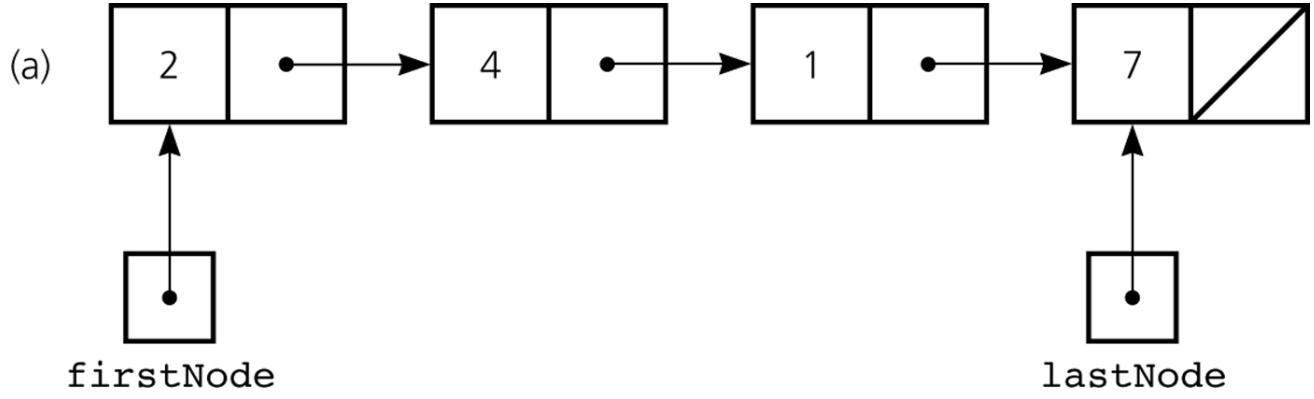
Note: Since **QueueException** is a **RuntimeException**, the calls to methods that throw **QueueException** do not have to be enclosed in **try** blocks.

See: docs.oracle.com/javase/8/docs/api/java/lang/runtimeexception

QUEUE IMPLEMENTATIONS - LL WITH TAIL

Queue implemented by using a **linked list with tail reference (a):**

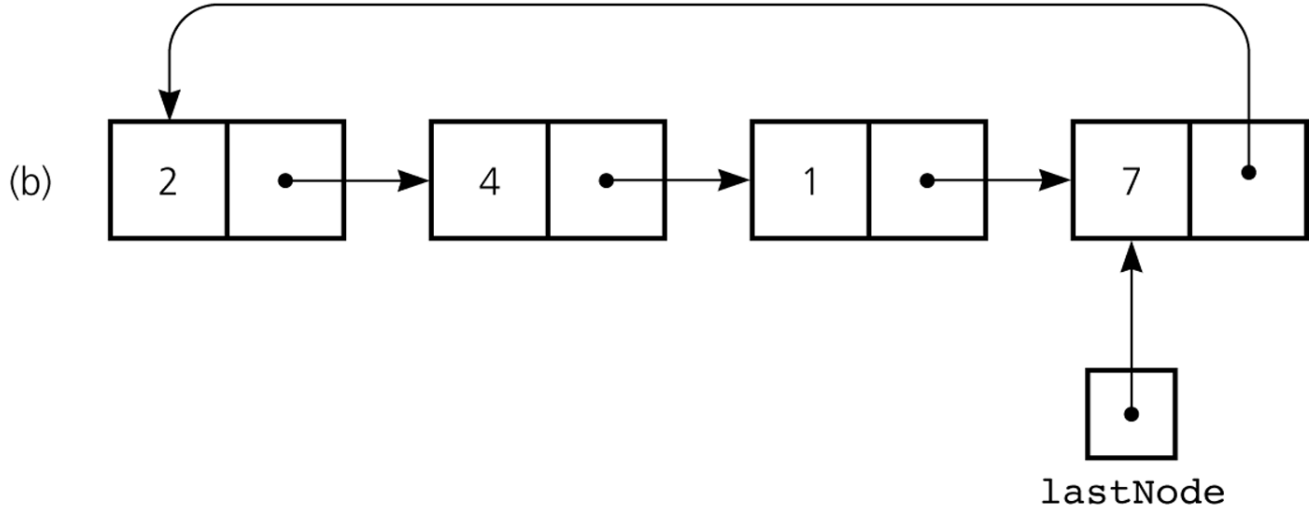
- **firstNode:** a reference to the front (head).
- **lastNode:** a reference to the back (tail).



QUEUE IMPLEMENTATIONS - CIRCULAR LL 1

Queue implemented by using a **circular linked list (b):**

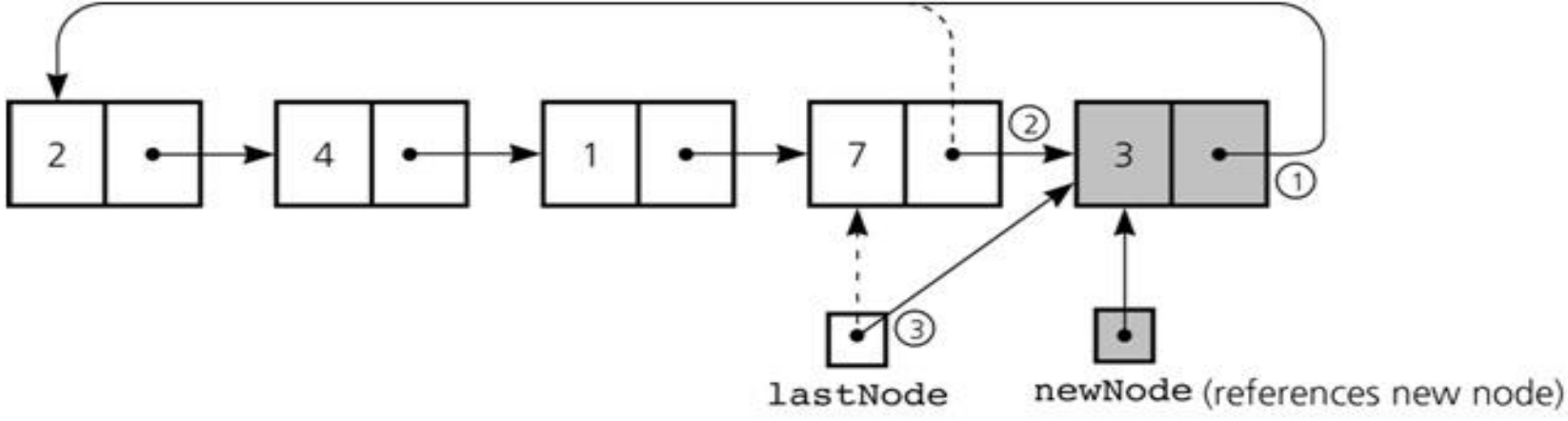
- **lastNode:** a reference to the back (tail).
- The reference to the front (head) is represented by **lastNode.next**.



QUEUE IMPLEMENTATIONS - CIRCULAR LL 2

Inserting an item into a non-empty queue (implemented with a circular linked list).

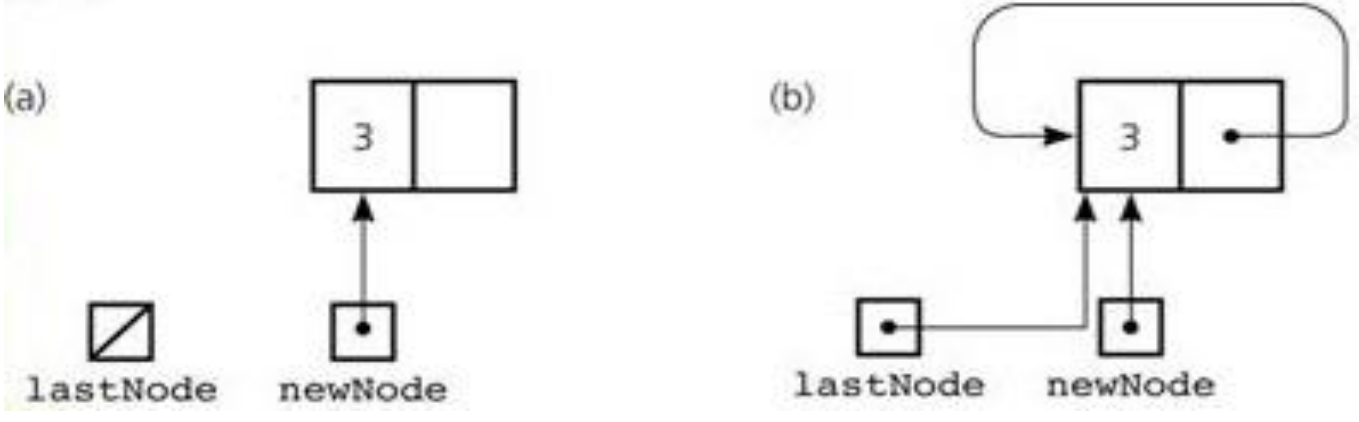
```
newNode.next = lastNode.next; // New node next reference set to front node (circular).
lastNode.next = newNode; // Last node next reference set to new node.
lastNode = newNode; // Last node is now the new node.
```



QUEUE IMPLEMENTATIONS - CIRCULAR LL 3

Inserting an item into an empty queue (implemented with a circular linked list): before insertion **(a)**, and after insertion **(b)**.

```
newNode.next = newNode; // New node next reference set to new node (circular).
lastNode = newNode; // Last node is now the new node.
```

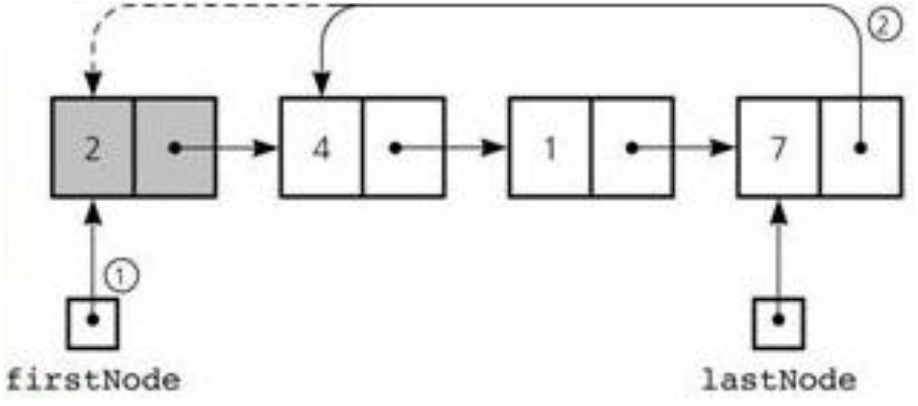


QUEUE IMPLEMENTATIONS - CIRCULAR LL 4

Delete from a queue with 2 or more items (implemented with a circular linked list).

```
firstNode = lastNode.next; // Store reference to 1st node in a temporary variable.
lastNode.next = firstNode.next; // Last node next set to 1st node next (2nd node).
```

If a queue has only one item, delete it setting **lastNode** to **null** (special case).



QUEUE IMPLEMENTATIONS - REFERENCE-BASED 1

```
public class QueueReferenceBased implements QueueInterface {
   private Node lastNode; // Reference to the last node of the queue.

   public QueueReferenceBased() { lastNode = null; } // Default constructor.
   public boolean isEmpty() { return lastNode == null; } // Checks if queue is empty.
   public void dequeueAll() { lastNode = null; } // Removes all items in queue.

   public void enqueue( Object newItem ) { // Inserts input item at back of queue.
      Node newNode = new Node( newItem );
      if( isEmpty() ) { newNode.next = newNode; }
      else {
         newNode.next = lastNode.next;
         lastNode.next = newNode; }
      lastNode = newNode;
   }

   // ...
```

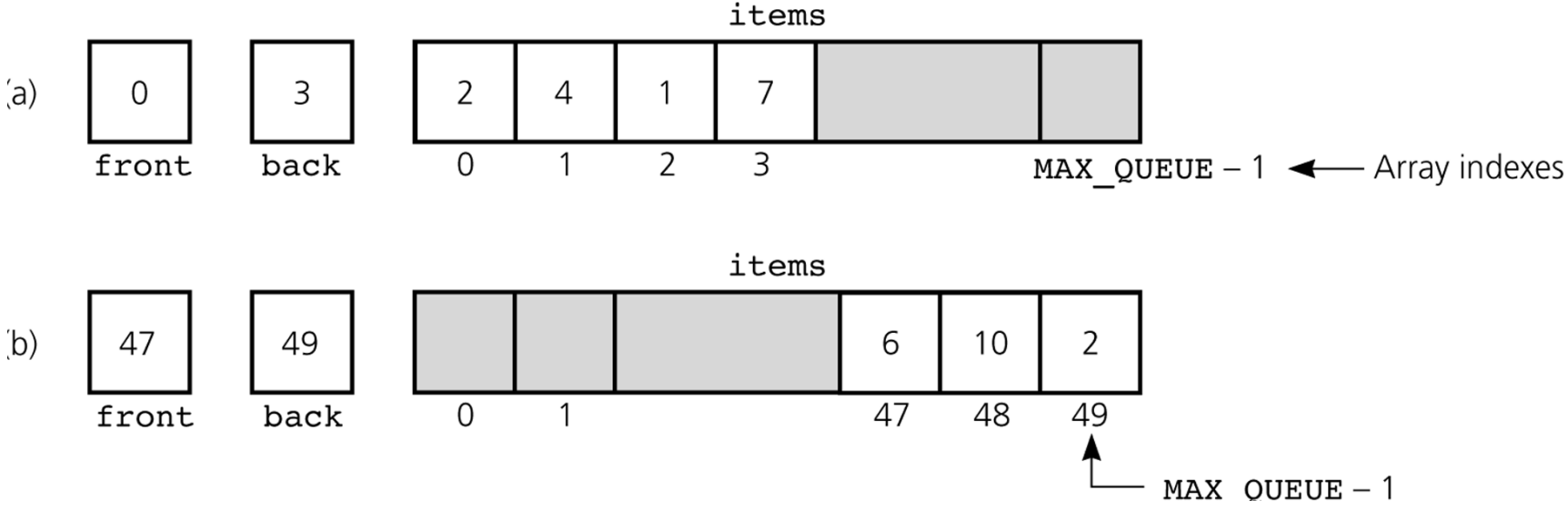
QUEUE IMPLEMENTATIONS - REFERENCE-BASED 2

```
   // ...

   public Object dequeue() throws QueueException { // Returns (and remove) item at front.
      if( !isEmpty() ) { // Queue is not empty, remove front.
         Node firstNode = lastNode.next;
         if( firstNode == lastNode ) { lastNode = null; } // Special case: 1 node queue.
         else { lastNode.next = firstNode.next; }
         return firstNode.item; }
      else { throw new QueueException( "QueueException on dequeue: queue empty!" ); }
   }

   public Object peek() throws QueueException { // Returns items at front (no remove).
      if( !isEmpty() ) { // Queue is not empty, retrieve front.
         Node firstNode = lastNode.next;
         return firstNode.item; }
      else { throw new QueueException( "QueueException on peek: queue empty!" ); }
   }
}
```

QUEUE IMPLEMENTATIONS - ARRAY-BASED

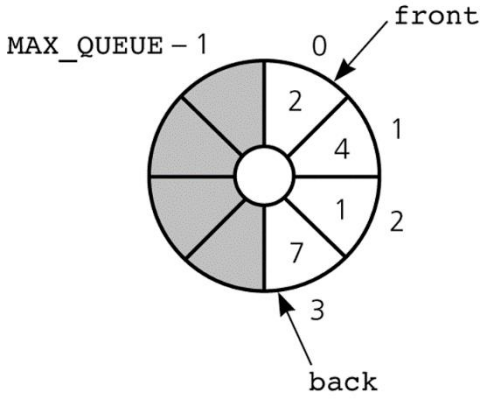
a. A simple array-based implementation of a queue.

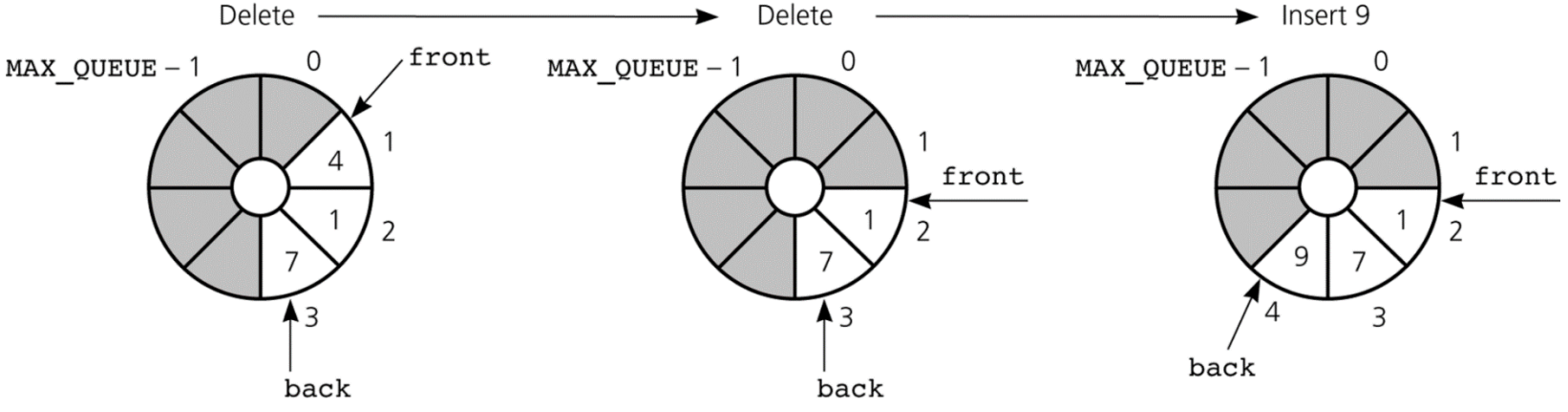
b. Usage involves a **rightward drift** causing the queue to appear full.



QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY

Array-based implementation of a queue with a **circular array** using two indices (**front** and **back**), eliminates the problem of rightward drift wrapping around 0 both indices.



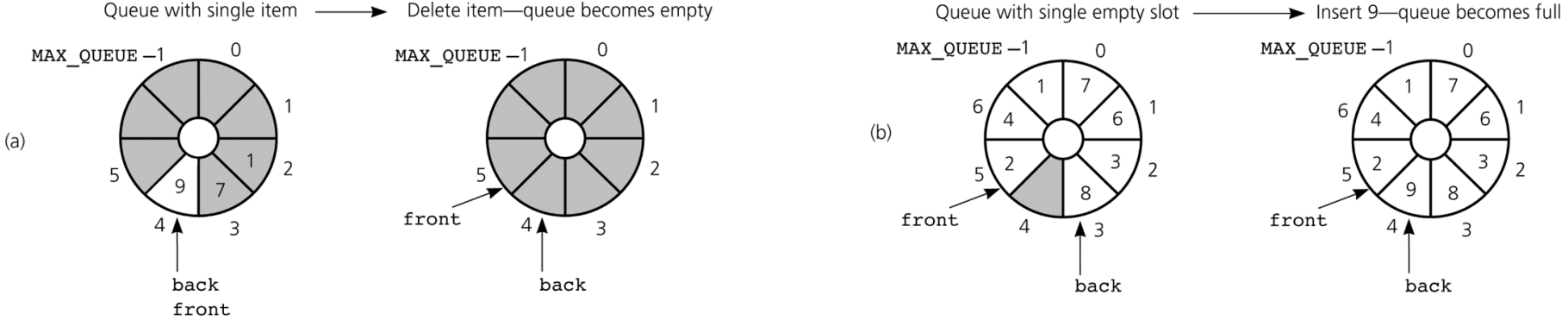


QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY 2

A problem with the circular array implementation is that **front** and **back** cannot be used to distinguish between **queue-full** and **queue-empty** conditions:

a. if the queue becomes empty, **front** passes **back**,
b. if the queue becomes full, **back** catches up to **front**.

To detect queue-full and queue-empty conditions, keep a **count** of the queue items.



QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY 3

Initialize the Queue:

```
front = 0; // Queue front initialized to 0.
back = MAX_QUEUE - 1; // Queue back initialized to MAX_QUEUE - 1.
count = 0; // Number of items in queue initialized to 0.
```

Inserting into a Queue:

```
back = ( back + 1 ) % MAX_QUEUE; // Increment queue back wrapping around 0.
items[back] = newItem; // Set the value of the new item in queue.
count++; // Increment the number of items in queue by 1.
```

Deleting from a Queue:

```
front = ( front + 1 ) % MAX_QUEUE; // Increment queue front wrapping around 0.
count--; // Decrement the number of items in queue by 1.
```

QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY 4

```
public class QueueArrayBased implements QueueInterface {

   private final int MAX_QUEUE = 50; // Maximum size of the queue.
   private Object[] items;
   private int front; // Index of the queue front.
   private int back; // Index of the queue back.
   private int count; // Number of items in queue.

   // Default constructor.
   public QueueArrayBased() {
      items = new Object[MAX_QUEUE];
      front = 0; back = MAX_QUEUE - 1; count = 0;
   }

   public boolean isEmpty() { return count == 0; }
   public boolean isFull() { return count == MAX_QUEUE; }

   // ...
```

QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY 5

```
// ...

public void enqueue( Object newItem ) throws QueueException {
   if( !isFull() ) {
      back = ( back + 1 ) % MAX_QUEUE;
      items[back] = newItem;
      count++; }
   else { throw new QueueException( "QueueException on enqueue: queue full!" ); } }

public Object dequeue() throws QueueException {
   if( !isEmpty() ) {
      Object queueFront = items[front];
      front = ( front + 1 ) % MAX_QUEUE;
      count--;
      return queueFront; }
   else { throw new QueueException( "QueueException on dequeue: queue empty!" ); } }

// ...
```

QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY 6

```
   // ...

   public void dequeueAll() {
      items = new Object[MAX_QUEUE];
      front = 0;
      back = MAX_QUEUE - 1;
      count = 0;
   }

   public Object peek() throws QueueException {
      if( !isEmpty() ) { return items[front]; }
      else { throw new QueueException( "QueueException on peek: queue empty!" ); }
   }

}
```

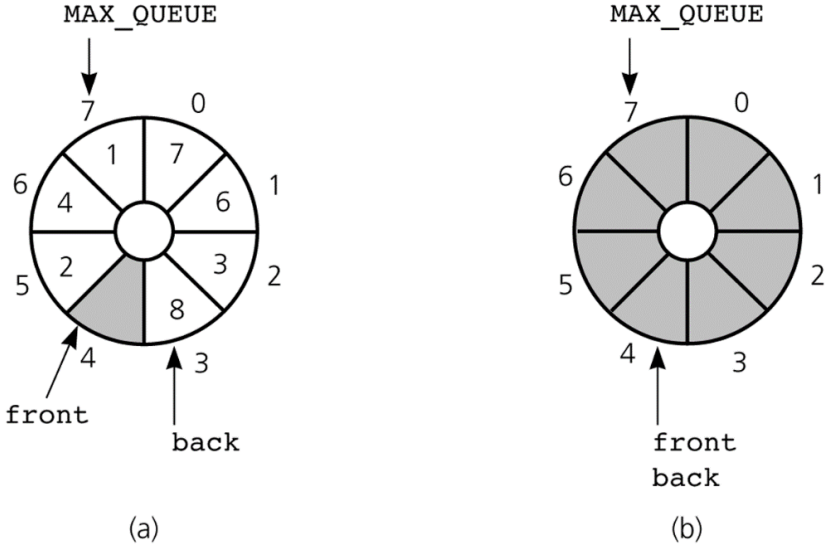
QUEUE IMPLEMENTATIONS - CIRCULAR ARRAY

VARIATIONS OF THE ARRAY-BASED IMPLEMENTATION:

Flag vs Counter: use a flag **full** to distinguish between full and empty conditions.

Dummy Item: declare **MAX_QUEUE+1** locations for array items, but use only **MAX_QUEUE** of them for queue items. In figure: a full queue **(a)**, and an empty queue **(b)**.

```
front == (back+1)%(MAX_QUEUE+1); // Full.
front == back; // Empty.
```

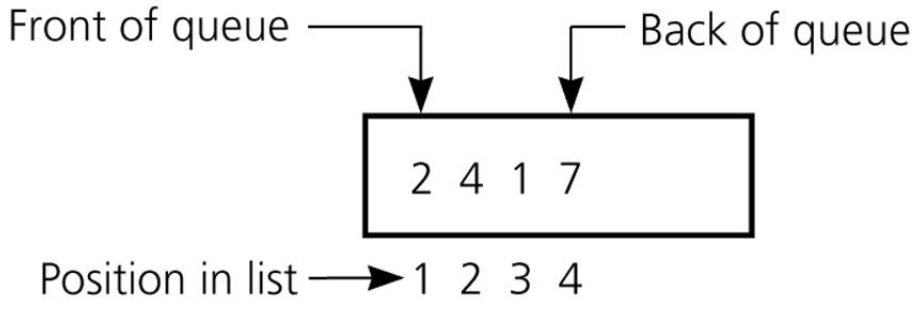


QUEUE IMPLEMENTATIONS - LIST-BASED 1

ADT QUEUE IMPLEMENTATION USING THE ADT LIST

If item in position **1** of the ADT list represents the front of the queue, **dequeue** and **peek** can be implemented in this way:

- **dequeue():** `list.remove(1);`
- **peek():** `list.get(1);`



If the item at the end of the ADT list represents the back of the queue, **enqueue** can be implemented in this way:

- **enqueue(newItem)**: `list.add( list.size() + 1, newItem );`

QUEUE IMPLEMENTATIONS - LIST-BASED 2

```
import List.*; // List package including: ListReferenceBased class, and ListInterface.

public class QueueListBased implements QueueInterface {

   private ListInterface list; // list only needs to implement ListInterface interface!

   // Default constructor.
   public QueueListBased() { list = new ListReferenceBased(); }

   public boolean isEmpty() { return list.isEmpty(); }

   public void enqueue( Object newItem ) { list.add( list.size(), newItem ); }

   // ...
```

QUEUE IMPLEMENTATIONS - LIST-BASED 3

```
   // ...

   public Object dequeue() throws QueueException {
      if( !isEmpty() ) {
         Object queueFront = list.get(0);
         list.remove(0);
         return queueFront; }
      else { throw new QueueException( "QueueException on dequeue: queue empty!" ); } }

   public void dequeueAll() { list.removeAll(); }

   public Object peek() throws QueueException {
      if( !isEmpty() ) { return list.get(0); }
      else { throw new QueueException( "QueueException on peek: queue empty!" ); } }

}
```

QUEUE IMPLEMENTATIONS - COMPARISON

The ADT queue implementations mentioned are either array-based or reference-based, so the main properties to compare are **fixed-size versus dynamic size**:

- **a statically allocated array** prevents the **enqueue** operation from adding an item to the queue if the array is full;
- **a resizable array or a reference-based implementation** does not impose this restriction on the enqueue operation;
- **the ADT list approach**, is slightly less efficient than the reference-based implementation, but reuses an already implemented class, so it is simpler to write and saves time.

QUEUE IMPLEMENTATIONS - JCF QUEUE

JCF has an interface **java.util.Queue** derived from interface **Collection** (+ **Iterable**).

JCF QUEUE INTERFACE

```
public interface Queue<E> extends Collection<E> {
   // Retrieves, but does not remove, head of queue. If queue empty, throws exception.
   E element() throws NoSuchElementException;
   boolean offer( E o ); // Inserts the specified element into the queue, if possible.
   E peek(); // Retrieves, but does not remove, head of queue. May return null...
   E poll(); // Retrieves and removes head of queue. Returns null if queue empty.
   // Retrieves and removes head of queue. If queue empty, throws exception.
   E remove() throws NoSuchElementException; }
```

See: docs.oracle.com/javase/8/docs/api/java/util/queue
See: docs.oracle.com/javase/8/docs/api/java/util/collection
See: docs.oracle.com/javase/8/docs/api/java/lang/iterable

QUEUE IMPLEMENTATIONS - JCF DEQUE 1

A **deque** (pronounced "deck") is a **double-ended queue** allowing us to insert and delete from either ends. Thus, **a deque may function as both a stack and a queue**.

JCF interface **java.util.Deque** derives interface **Queue** (+ **Collection** and **Iterable**).

Example: Implementing a **text editor**, we need **stack** functionality when handling input characters (**push** for each key pressed, except for backspace that causes a **pop**), but we need **queue** functionality when dealing with output characters (**FIFO** strategy to retrieve and display characters on the screen).

See: docs.oracle.com/javase/8/docs/api/java/util/deque
See: docs.oracle.com/javase/8/docs/api/java/util/queue
See: docs.oracle.com/javase/8/docs/api/java/util/collection
See: docs.oracle.com/javase/8/docs/api/java/lang/iterable

QUEUE IMPLEMENTATIONS - JCF DEQUE 2

java.util.Deque interface includes the following methods (partial display here):

```
// A sketch of the JCF Deque interface.
public interface Deque<E> extends Queue<E> {

   // 3 METHODS FOR ADDING AN ELEMENT, IF NO SPACE IS AVAILABLE, THEY THROW AN EXCEPTION.

   // Inserts the specified element at the back of the deque.
   boolean add( E e ) throws IllegalStateException;

   // Inserts the specified element at the front of the deque.
   void addFirst( E e ) throws IllegalStateException;

   // Inserts the specified element at the back of the deque.
   void addLast( E e ) throws IllegalStateException;

   // ...
```

QUEUE IMPLEMENTATIONS - JCF DEQUE 3

```
// ...

// 3 METHODS FOR ADDING AN ELEMENT, RETURN TRUE UPON SUCCESS OR FALSE OTHERWISE.

boolean offer( E e ); // Inserts the speficied element at the back of the deque.
boolean offerFirst( E e ); // Inserts the speficied element at the front of the deque.
boolean offerLast( E e ); // Inserts the speficied element at the back of the deque.

// 3 METHODS TO RETRIEVE BUT NOT REMOVE AN ELEMENT, THROW EXCEPTION IF DEQUE IS EMPTY.

E element() throws NoSuchElementException; // Get, but not remove, the deque front.
E getFirst() throws NoSuchElementException; // Get, but not remove, the deque front.
E getLast() throws NoSuchElementException; // Get, but not remove, the deque back.

// ...
```

QUEUE IMPLEMENTATIONS - JCF DEQUE 4

```
// ...

// 3 METHODS TO RETRIEVE BUT NOT REMOVE AN ELEMENT, RETURN NULL IF DEQUE IS EMPTY.

E peek(); // Retrieves, but does not remove, the front of the deque.
E peekFirst(); // Retrieves, but does not remove, the front of the deque.
E peekLast(); // Retrieves, but does not remove, the back of the deque.

// 3 METHODS TO RETRIEVE AND REMOVE AN ELEMENT, THROW EXCEPTION IF DEQUE IS EMPTY.

E remove() throws NoSuchElementException; // Get and removes the deque front.
E removeFirst() throws NoSuchElementException; // Get and removes the deque front.
E removeLast() throws NoSuchElementException; // Get and removes the deque back.

// ...
```

QUEUE IMPLEMENTATIONS - JCF DEQUE 5

```
    // ...

    // 3 METHODS TO RETRIEVE AND REMOVE AN ELEMENT, RETURN NULL IF DEQUE IS EMPTY.

    E poll() throws NoSuchElementException; // Get and removes the deque front.
    E pollFirst() throws NoSuchElementException; // Get and removes the deque front.
    E pollLast() throws NoSuchElementException; // Get and removes the deque back.

    // MISCELLANEOUS METHODS.

    boolean contains( Object o ); // Returns true if deque contains the specified element.
    int size(); // Returns the number of elements in the deque.

}
```

QUEUE APPLICATIONS - READING A STRING

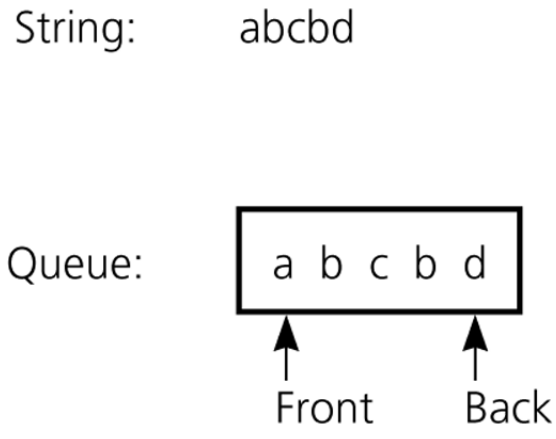
A queue can retain characters in the order in which they are typed. Then, once the characters are in the queue, the system can process them as necessary.

```
// Read a string of characters from a single line of input into a queue.
queue.createQueue()
while( not the end of the line ) {
   Read a new character ch of the line;
   queue.enqueue(ch); }
```

QUEUE APPLICATIONS - CHECK PALINDROME 1

CHECK PALINDROME: PROBLEM

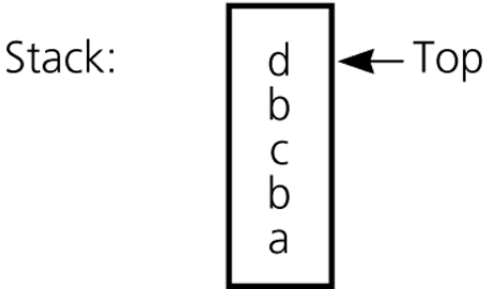
To recognize a **palindrome**, a queue can be used in conjunction with a stack: a **stack** can be used to **reverse** the order of occurrences, whereas a **queue** can be used to **preserve** the order of occurrences.



CHECK PALINDROME: ALGORITHM

Non-recursive recognition algorithm for palindromes:

1. traverse the string from left to right,
2. add current char to a queue and a stack (right),
3. compare first chars in queue and in stack.



QUEUE APPLICATIONS - CHECK PALINDROME 2

CHECK PALINDROME: PSEUDOCODE OF THE ALGORITHM

```
// Pseudocode of a non-recursive recognition algorithm for the language of palindromes.
public boolean isPalindrome( String str ) {
   aQueue.createQueue(); aStack.createStack(); // Set an empty queue and an empty stack.
   // Insert each character of the string into both the queue and the stack.
   int length = the length of str;
   for( int i = 1; i <= length; i++ ) {
      char nextChar = character at position i of str;
      aQueue.enqueue( nextChar );
      aStack.push( nextChar ); }
   // Compare the queue characters with the stack characters.
   boolean charactersAreEqual = true;
   while( !aQueue.isEmpty() && charactersAreEqual ) {
      char queueFront = aQueue.dequeue();
      char stackTop = aStack.pop();
      if( queueFront != stackTop ) { charactersAreEqual = false; } }
   return charactersAreEqual;
}
```

SUMMARY OF POSITION-ORIENTED ADTs

List: All positions can be accessed.
ADT list operations generalize stack and queue operations:
length, **add**, **remove**, and **get**

Stack and Queue: Only end position can be accessed.
Operations of stacks and queues can be paired off as:
createStack and **createQueue**
isEmpty (stack) and **isEmpty** (queue)
push and **enqueue**
pop and **dequeue**
peek (stack) and **peek** (queue)