## Comparing Implementations

- All of the three implementations are ultimately array based or reference based
- Fixed size versus dynamic size
  - An array-based implementation
    - Uses fixed-sized arrays
      - Prevents the `push` operation from adding an item to the stack if the stack's size limit has been reached
  - A reference-based implementation
    - Does not put a limit on the size of the stack




## Comparing Implementations

- An implementation that uses a linked list versus one that uses a reference-based implementation of the ADT list
  - Linked list approach
    - More efficient (in reality not enough to be noticeable)
  - ADT list approach
    - Reuses an already implemented class
      - Much simpler to write
      - Saves time (for the programmer)




## The Java Collections Framework Class `Stack`

- JCF contains an implementation of a stack class called `Stack` (generic)
- Derived from `Vector`
- Includes methods: `peek, pop, push,` and `search`
- `search` returns the position of an object on the stack (1 is the top of the stack)




## An RPN (or postfix) calculator

- Reverse Polish Notation (named in honour of Jan Lukasiewicz) (sometimes called Zciweisakul notation)

3 5 6 + 2 * -

This evaluates (3 – (5 + 6) * 2)

- Operands are pushed on a stack.
- Operators pop off two elements, perform the operation and push back the result

# RPN Example

3 5 6 + 2 * -

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 6 | | 2 | | |
| | 5 | 5 | 11 | 11 | 22 | |
| 3 | 3 | 3 | 3 | 3 | 3 | -19 |




# A Simple RPN Calculator program

See RPNCalculator.java

```
Stack<Double> values = new Stack<Double>();
String input;
while ((input = Keyboard.readInput()).length() > 0) {
   try {
        double number = Double.parseDouble(input);
        values.push(number);
        System.out.println(values);
   } catch (NumberFormatException e) {
        if (values.size() < 2) {
                System.out.println("Not enough numbers on
                                                  the stack.");
                System.out.println(values);
                continue;
        }
```




# A Simple RPN Calculator program 2

```
        double x, y;
        double answer;
        char operator;
        operator = input.charAt(0);
        switch (operator) {
        case '+':
                y = values.pop();
                x = values.pop();
                answer = x + y;
                break;
        … // the other operations
        default:
                System.out.println("Incorrect operator.");
                System.out.println(values);
                continue;
        }
        values.push(answer);
        System.out.println(values);
   }
```




# Converting Infix Expressions to Equivalent Postfix Expressions

- An infix expression can be evaluated by first being converted into an equivalent postfix expression
- Facts about converting from infix to postfix
  - Operands always stay in the same order with respect to one another
  - An operator will move only “to the right” with respect to the operands
  - All parentheses are removed

# Converting Infix Expressions to Equivalent Postfix Expressions

| 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 |

Figure 7-9

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




# Converting Infix Expressions to Equivalent Postfix Expressions

| ch | stack (bottom to top) | postfixExp | |
|---|---|---|---|
| a | | a | |
| + | + | a | |
| ( | + ( | a | |
| b | + ( | ab | |
| * | + ( * | ab | |
| c | + ( * | abc | |
| - | + ( - | abc* | higher precedence operator popped off |
| d | + ( - | abc*d | |
| ) | + ( | abc*d- | |
| | + | abc*d- | |
| / | + / | abc*d- | |
| e | + / | abc*d-e | |
| | + | abc*d-e/ | |
| | | abc*d-e/+ | |

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




# The Relationship Between Stacks and Recursion

- The ADT stack has a hidden presence in the concept of recursion
- Typically, stacks are used by compilers to implement recursive methods
  - During execution, each recursive call generates an activation record that is pushed onto a stack
- Stacks can be used to implement a nonrecursive version of a recursive algorithm




# The Abstract Data Type Queue

- A queue
  - New items enter at the back, or rear, of the queue
  - Items leave from the front of the queue
  - First-in, first-out (FIFO) property
    - The first item inserted into a queue is the first item to leave

# The Abstract Data Type Queue

- ADT queue operations
  - Create an empty queue
  - Determine whether a queue is empty
  - Add a new item to the queue
  - Remove from the queue the item that was added earliest
  - Remove all the items from the queue
  - Retrieve from the queue the item that was added earliest




# The Abstract Data Type Queue

- Pseudocode for the ADT queue operations

```
createQueue()
// Creates an empty queue.

isEmpty()
// Determines whether a queue is empty

enqueue(newItem) throws QueueException
// Adds newItem at the back of a queue. Throws
// QueueException if the operation is not
// successful
```




# The Abstract Data Type Queue

- Pseudocode for the ADT queue operations (Continued)

```
dequeue() throws QueueException
// Retrieves and removes the front of a queue.
// Throws QueueException if the operation is
// not successful.

dequeueAll()
// Removes all items from a queue

peek() throws QueueException
// Retrieves the front of a queue. Throws
// QueueException if the retrieval is not
// successful
```




# The Abstract Data Type Queue

| Operation | Queue after operation (Front at left) |
|---|---|
| `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) |

Figure 8-2

Some queue operations

# Recognizing Palindromes

- A nonrecursive recognition algorithm for palindromes
  – As you traverse the character string from left to right, insert each character into both a queue and a stack
  – Compare the characters at the front of the queue and the top of the stack

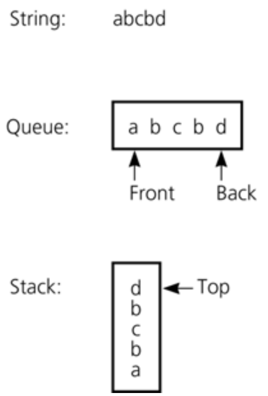


Figure 8-3
The results of inserting a string into both a queue and a stack




# A Reference-Based Implementation

- Possible implementations of a queue
  – A linear linked list with two external references
    - A reference to the front
    - A reference to the back

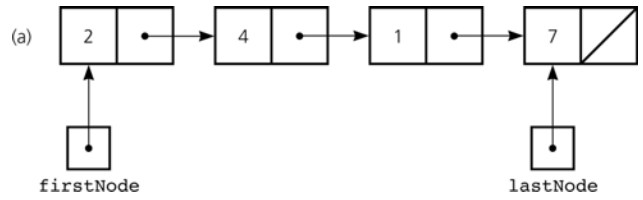


Figure 8-4a
A reference-based implementation of a queue: a) a linear linked list with two external references




# A Reference-Based Implementation

- Possible implementations of a queue (Continued)
  – A circular linked list with one external reference
    - A reference to the back

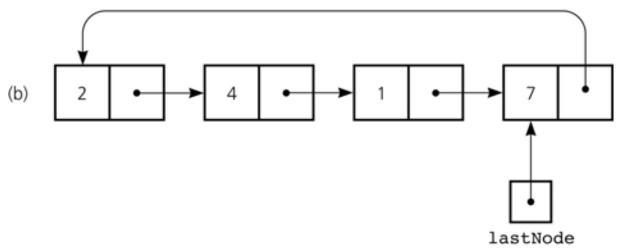


Figure 8-4b
A reference-based implementation of a queue: b) a circular linear linked list with one external reference




# A Reference-Based Implementation

Circular linked-list implementation

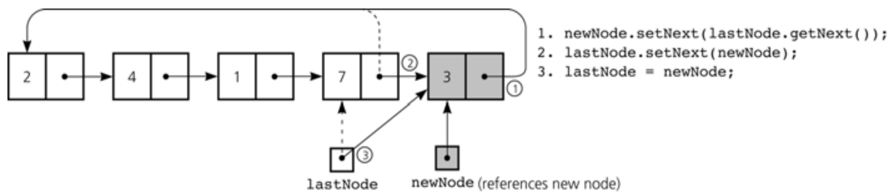


Figure 8-5
Inserting an item into a nonempty queue

# A Reference-Based Implementation

Circular linked-list implementation

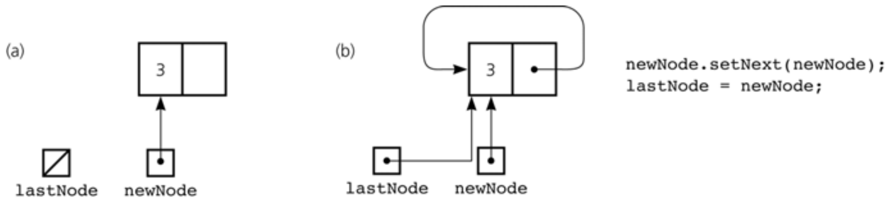


Figure 8-6

Inserting an item into an empty queue: a) before insertion; b) after insertion





# A Reference-Based Implementation

Circular linked-list implementation

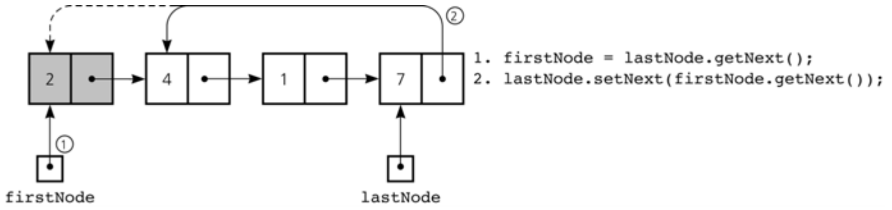


Figure 8-7

Deleting an item from a queue of more than one item