5 - STACKS AND QUEUES

Topics

- Stacks
- Queues
- Deques
- Priority Queues
- Example: Parsing Arithmetic Expressions
  - Translating Infix Expressions to Postfix Expressions
  - Evaluating Postfix Expressions

Stacks

- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
  - A stack allows access to only one data item: the last item inserted.
  - Placing a data item on the top of the stack is called *pushing* it.
  - Removing it from the top of the stack is called *popping* it.
- Stack can be used to:
  - check whether parentheses, braces, and brackets are balanced in a computer program source file.
  - parse (analyze) arithmetic expressions such as 3*(4+5)

Stacks

See Stack Workshop applet

- Top of stack
- Push (notice push to a full stack (top = N-1) -> overflow)
- Pop (notice pop from an empty stack (top = -1) -> underflow)
- Peek

Pushing and popping

	0	1	2	3	4	5	6	7	8	9
stk:	17	23	97	44						

top = 3 or count = 4

- If the bottom of the stack is at location 0, then an empty stack is represented by top = -1 or count = 0
- To add (push) an element, either:
  - Increment top and store the element in stk[top], or
  - Store the element in stk[count] and increment count
- To remove (pop) an element, either:
  - Get the element from stk[top] and decrement top, or
  - Decrement count and get the element in stk[count]

After popping

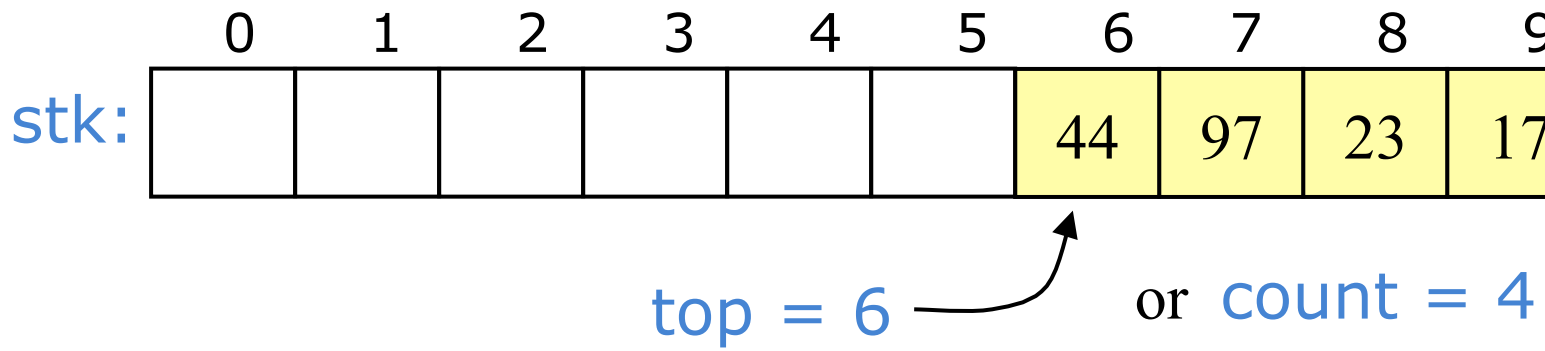
	0	1	2	3	4	5	6	7	8	9
stk:	17	23	97	44						

top = 2 or count = 3

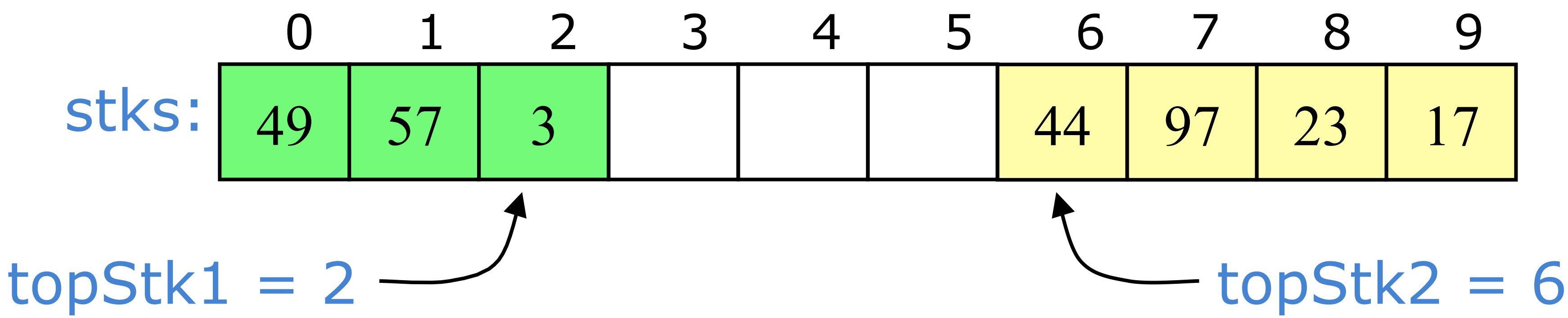
- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, "*it depends*"
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to null
  - Why? To allow it to be garbage collected!

Sharing space

- Of course, the bottom of the stack could be at the *other* end



- Sometimes this is done to allow two stacks to share the *same storage area*



Array implementation of stacks

- To implement a stack, items are inserted and removed at the same end (called the top)
- Efficient array implementation requires that the top of the stack be towards the center of the array, not fixed at one end
- To use an array to implement a stack, you need both the array itself and an integer
  - The integer tells you either:
    - Which location is currently the top of the stack, or
    - How many elements are in the stack
- See Java code in Listing 4.1, page 120

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack
- For underflow, you should throw an exception
  - If you don't catch it yourself, Java will throw an ArrayIndexOutOfBounds exception
  - You could create your own, more informative exception
- For overflow, you could do the same things
  - Or, you could check for the problem, and copy everything into a new, larger array

Stack Examples - Reversing a word

- Stack Example 1: Reversing a word (Listing 4.2, page 124)
  - It displays the entered word with the letters in reverse order.

```
Enter a string: part
Reversed: trap
```

Stack Examples - Delimiter matching (1)

- Stack Example 2: Delimiter matching (Listing 4.3, page 128)
  - It checks the delimiters in a line of text typed by the user.
  - The delimiters are the braces { and }, brackets [and], and parentheses (and).
  - Each opening or left delimiter should be matched by a closing or right delimiter.
  - Also, opening delimiters that occur later in the string should be closed before those occurring earlier.
  - Here are some examples:
    - c [d] // correct
    - a { b [c] d } e // correct
    - a { b (c] d } e // not correct;] doesn't match (
    - a [b { c } d] e } // not correct; nothing matches final }
    - a { b (c) // not correct; nothing matches opening {

Stack Examples - Delimiter matching (2)

- RULE: Read characters from the string one at a time:
  - Place opening delimiters when it finds them, on a stack.
  - When it reads a closing delimiter from the input, it pops the opening delimiter from the top of the stack and attempts to match it with the closing delimiter.
    - If they're not the same, an error occurs. => **a { b (c] d } e**
    - Also, if there is no opening delimiter on the stack to match a closing one, an error occurs. => **a [b { c } d] e }**
    - A delimiter that hasn't been matched is discovered because it remains on the stack after all the characters in the string have been read. => **a { b (c)**

Stack Examples - Delimiter matching (3)

- Let's see what happens on the stack for a typical correct string:
- a { b (c [d] e) f }

Character Read	a	{	b	(	c	[	d	]	e	)	f	}
		Push		Push		Push		Pop		Pop		Pop
Stack Contents												
						[	[					
				(	(	(	(	(	(			
		{	{	{	{	{	{	{	{	{	{	

Efficiency of Stacks

- Items can be both pushed and popped from the stack in constant O(1) time.
- That is, the time is not dependent on how many items are in the stack and is therefore very quick.
- No comparisons or moves are necessary.

Queues

- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
  - The two basic queue operations are:
    - *inserting (enque)* an item, which is placed at the rear of the queue, and
    - *removing (deque)* an item, which is taken from the front of the queue.

FIGURE 4.4 A queue of people.

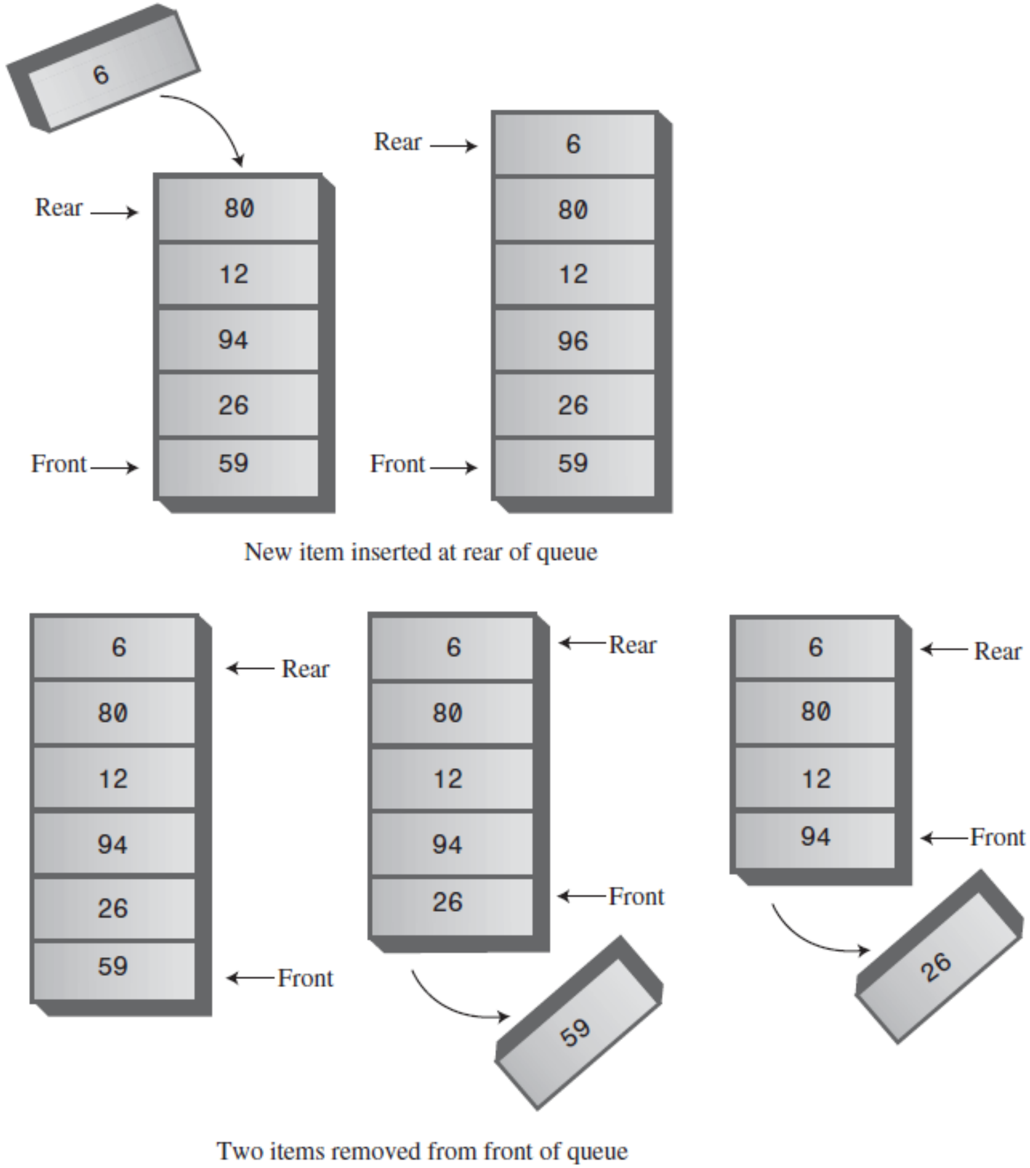


FIGURE 4.6 Operation of the `Queue` class methods.

Queues

- Model real-world situations such as People waiting in line at a bank, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet.
- Various queues doing their job in your computer's (or the network's) operating system.
  - There's a printer queue where print jobs wait for the printer to be available.
  - A queue also stores keystroke data as you type at the keyboard. Using a queue guarantees the keystrokes stay in order until they can be processed.

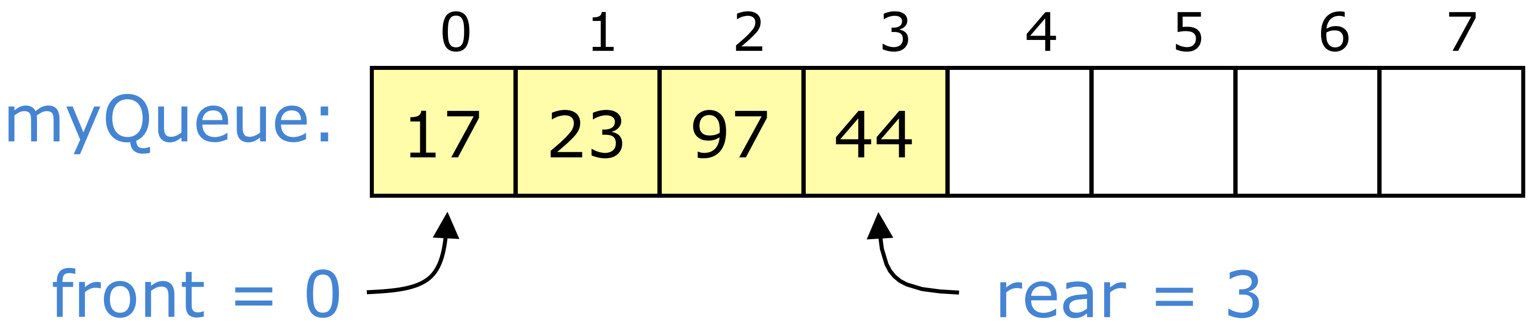
Queues

See Queue Workshop applet

- Front and Rear
- Insert (notice insert to a full Queue -> overflow)
- Remove (notice remove from an empty Queue -> underflow)
- Peek

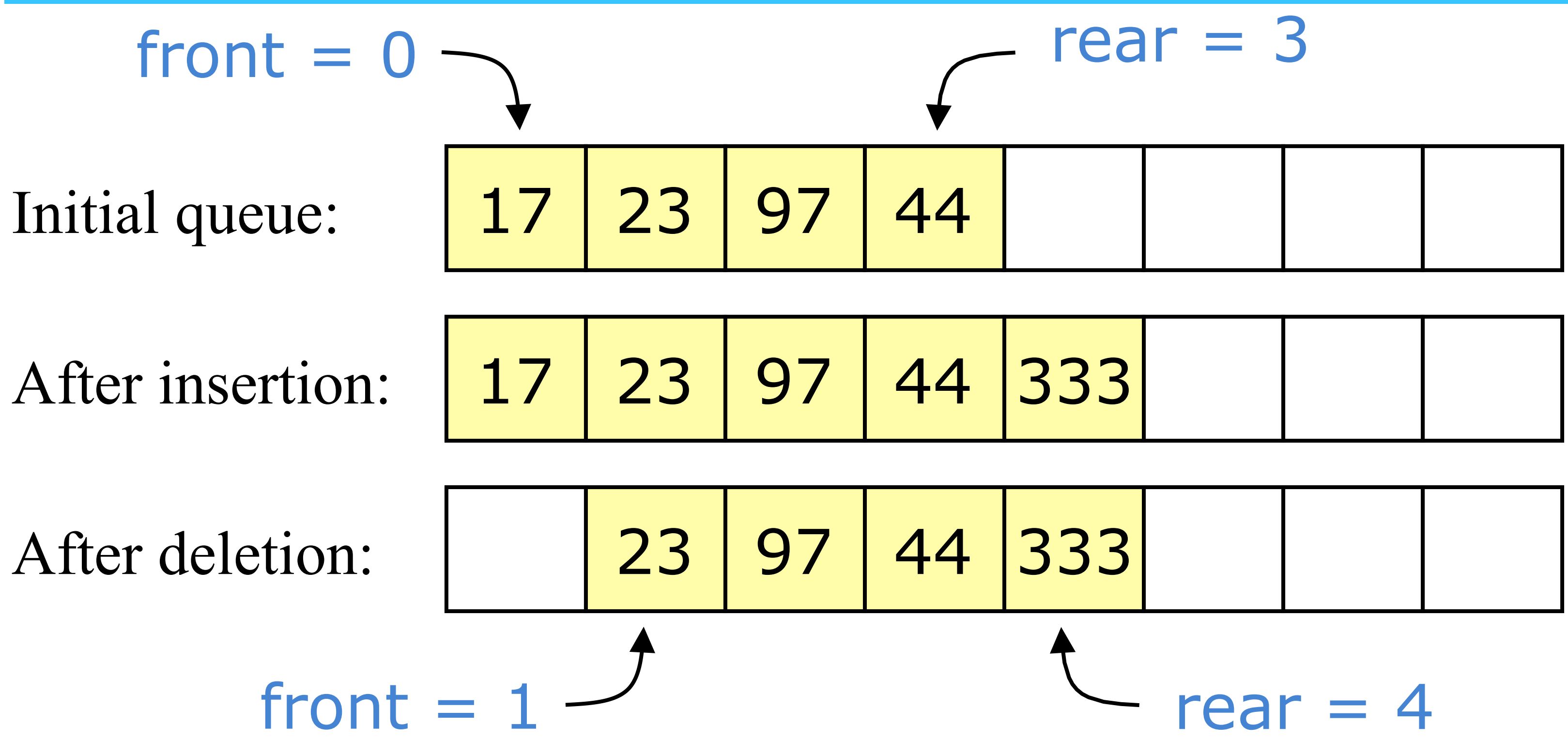
Array implementation of queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

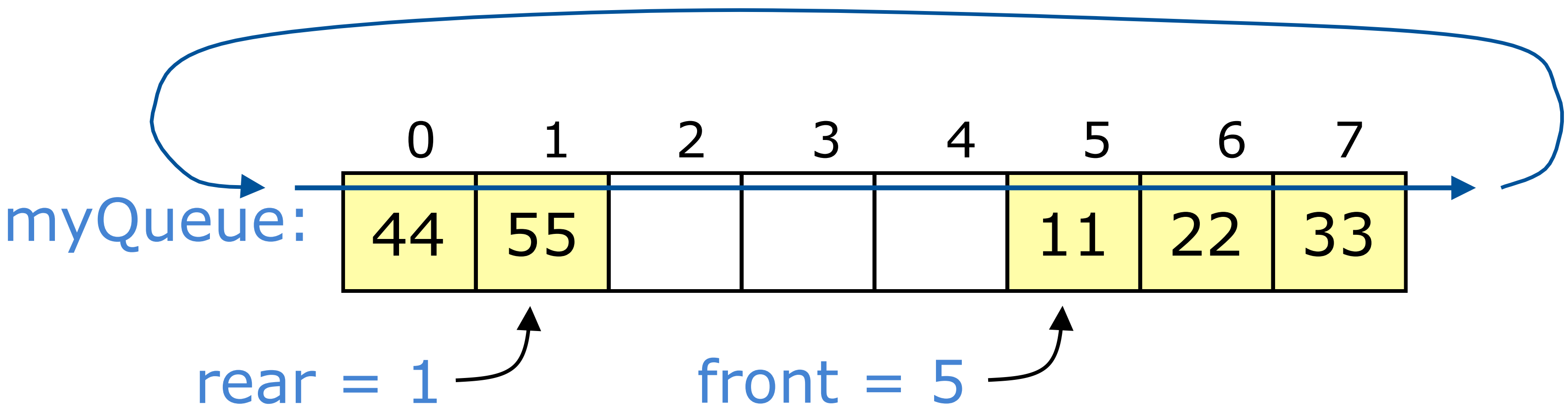
Array implementation of queues



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

Circular arrays

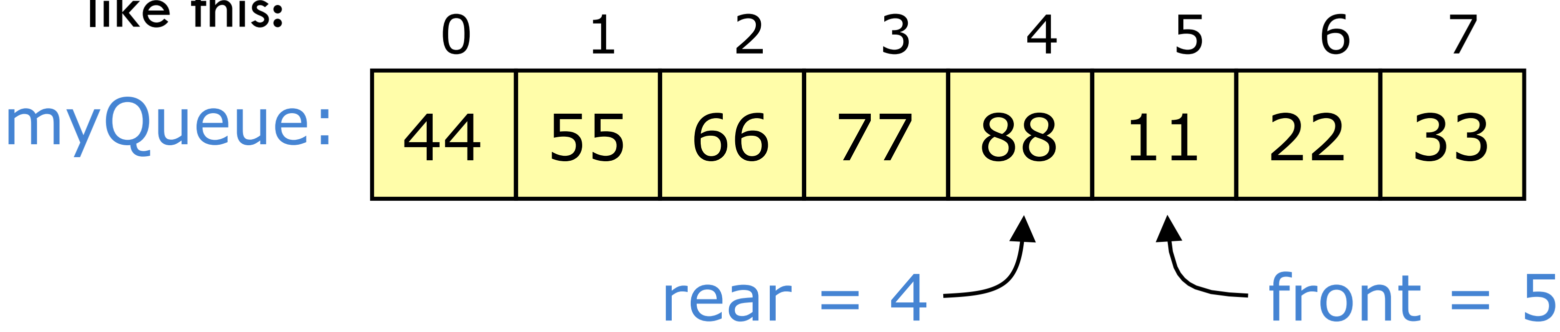
- We can treat the array holding the queue elements as circular (joined at the ends)



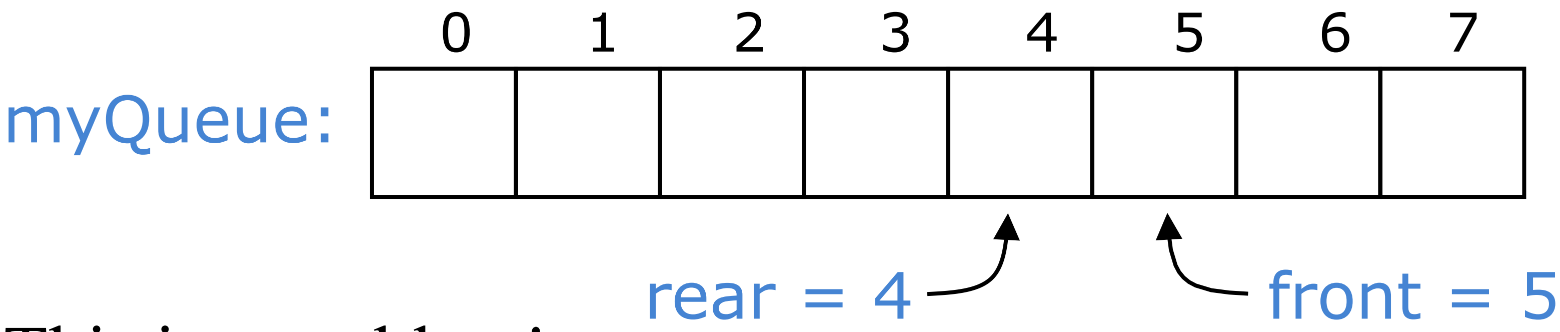
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;`
  and: `rear = (rear + 1) % myQueue.length;`

Full and empty queues

- If the queue were to become completely full, it would look like this:



- If we were then to remove all eight elements, making the queue completely empty, it would look like this:



This is a problem!

Full and empty queues: solutions

- **Solution #1:** Keep an additional variable

myQueue:

0	1	2	3	4	5	6	7
44	55	66	77	88	11	22	33

count = 8 rear = 4 front = 5

- **Solution #2:** (Slightly more efficient) Keep a gap between elements: consider the queue full when it has `n-1` elements

myQueue:

0	1	2	3	4	5	6	7
44	55	66	77		11	22	33

rear = 3 front = 5

Queue implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to null
- Java code implementation (see Listing 4.4, page 138).

Efficiency of Queues

- As with a stack, items can be inserted and removed from a queue in O(1) time.

Deques

- A *deque* is a double-ended queue.
- You can insert items at either end and delete them from either end.
- The methods might be called insertLeft() and insertRight(), and removeLeft() and removeRight().
- If you restrict yourself to insertRight() and removeRight() (or their equivalents on the right), the deque acts like a stack.
- If you restrict yourself to insertRight() and removeLeft() (or the opposite pair), it acts like a queue.

Priority Queues

- A **priority queue** is a more specialized data structure than a stack or a queue.
- Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front.
- However, in a priority queue, items are ordered by key value so that the item with the lowest key is always at the front (in case lowest key has highest priority).
- Items are inserted in the proper position to maintain the order.

Priority Queues

See PriorityQ
Workshop applet

- Ascending Priority

Queue -> minimum key always at the top.
(It can also be
a Descending Priority
Queue -> maximum on top)

- Insert (ordered)
- Remove (from front)
- Peek

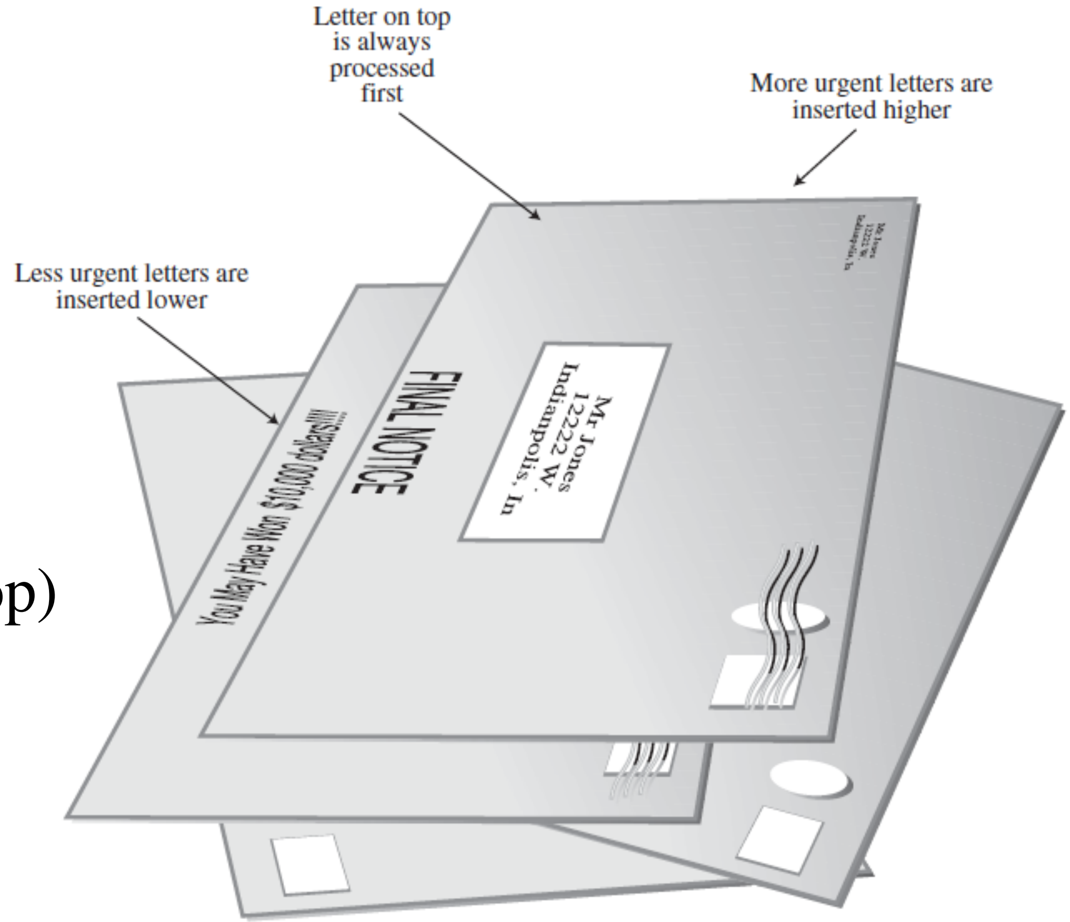
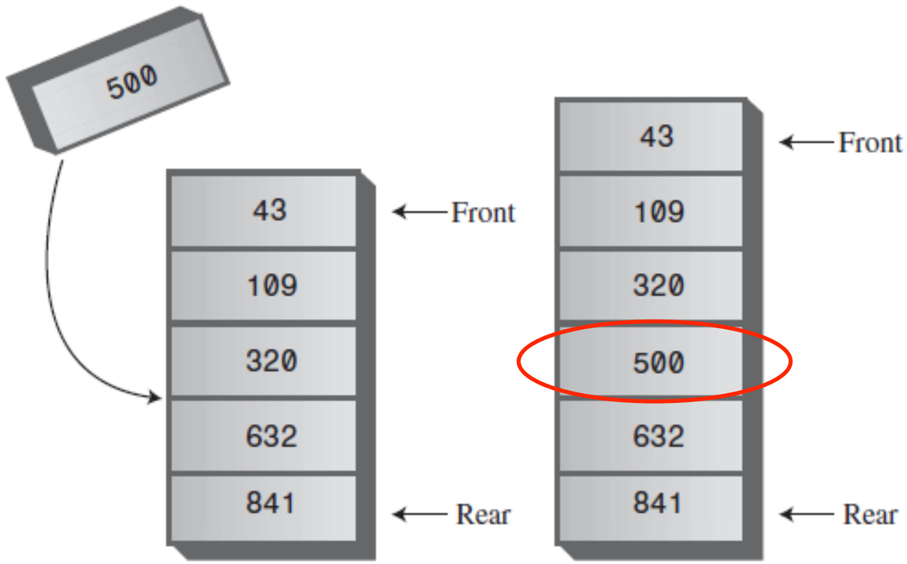
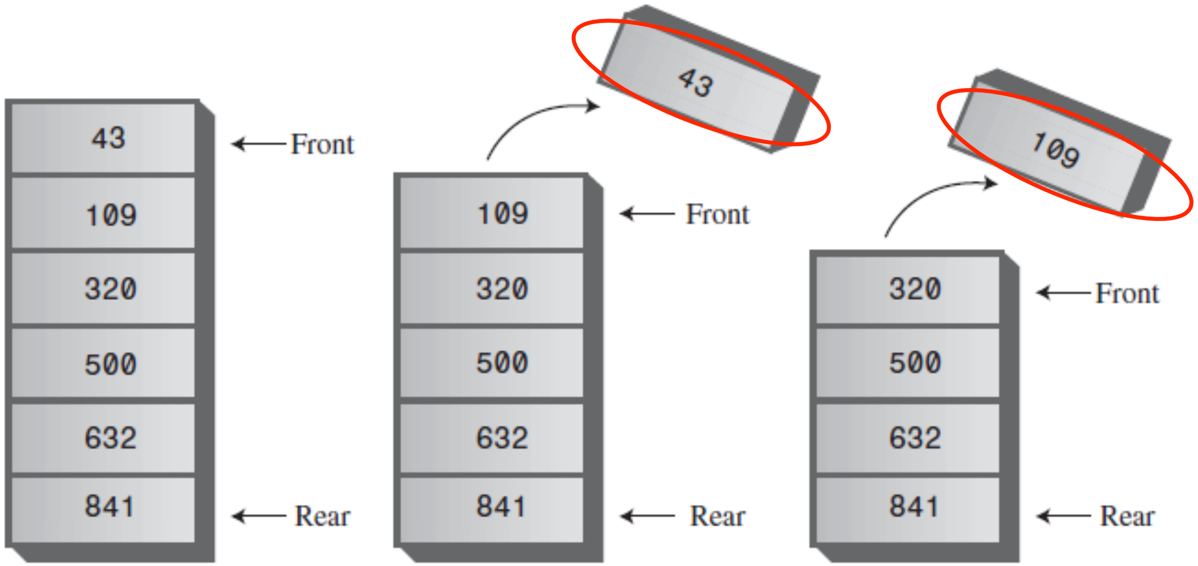


FIGURE 4.10 Letters in a priority queue.



New item inserted in priority queue



Two items removed from front of priority queue

Priority Queue Implementation Detail

- For small numbers of items, or situations in which speed isn't critical, implementing a priority queue with an array is satisfactory.
- For larger numbers of items, or when speed is critical, the heap is a better choice.
- Insertion is slow, but deletion is fast (delete always the front of the queue, at "count-1").
- See Java code in Listing 4.6, page 147.

Efficiency of Priority Queues

- In the priority-queue implementation we show here, insertion runs in O(N) time, while deletion takes O(1) time.
- We'll see how to improve insertion time with "heaps"

Parsing Arithmetic Expressions

- *Parsing* (analyzing) arithmetic expressions such as 2+3 or 2*(3+4) or ((2+4)*7)+3*(9–5)
- The storage structure it uses is the stack (like in case of checking brackets).
- As it turns out, it's fairly difficult, at least for a computer algorithm, to evaluate an arithmetic expression directly.
- It's easier to use a two-step process:
    1. Transform the arithmetic expression into a different format, called postfix notation.
    2. Evaluate the postfix expression.

Postfix notation

- Infix notation: A+B
  - Operator between operands
- Prefix notation: +AB
  - Operator before operands
- Postfix notation: AB+
  - Reverse Polish Notation (RPN): Operator follows the two operands.
- Operators: + , - , * , /

Important note about Operator’s precedence

- Both * and / have a higher precedence than + and –, so all multiplications and divisions must be carried out before any additions or subtractions (unless parentheses dictate otherwise)
- 2 + 3 * 4 = 2 + 12 = 14 (postfix 234*+)
  - “*” has higher precedence over “+”
- (2 + 3) * 4 = 5 * 4 = 20 (postfix 23+4*)
  - Parenthesis “ ( ) “ have precedence over other operators.

Postfix notation

TABLE 4.2 Infix and Postfix Expressions

Infix	Postfix
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C–D)	AB+CD–*
((A+B)*C)–D	AB+C*D–
A+B*(C–D/(E+F))	ABCDEF+/–*+

How Humans Evaluate Infix?

- 234+* = ?
  - 2*(3+4) = 14
- When "solving" an arithmetic expression, we follow rules like:
  - **1.** You read from left to right.
  - **2.** When you've read enough to evaluate two operands and an operator, you do the calculation and substitute the answer for these two operands and operator.
    - * and /, have higher precedence than + and -
  - **3.** You continue this process—going from left to right and evaluating when possible—until the end of the expression.

How Humans Evaluate Infix?

TABLE 4.3 Evaluating 3+4–5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
–	7	When you see the –, you can evaluate 3+4.
	7–	
5	7–5	
End	2	When you reach the end of the expression, you can evaluate 7–5.

Role of Precedence

TABLE 4.4 Evaluating 3+4*5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
*	3+4*	You can't evaluate 3+4 because * is higher precedence than +.
5	3+4*5	When you see the 5, you can evaluate 4*5.
	3+20	
End	23	When you see the end of the expression, you can evaluate 3+20.

Role of Parenthesis

TABLE 4.5 Evaluating 3*(4+5)

Item Read	Expression Parsed So Far	Comments
3	3	
*	3*	
(	3*(	
4	3*(4	You can't evaluate 3*4 because of the parenthesis.
+	3*(4+	
5	3*(4+5	You can't evaluate 4+5 yet.
)	3*(4+5)	When you see the), you can evaluate 4+5.
	3*9	After you've evaluated 4+5, you can evaluate 3*9.
	27	
End		Nothing left to evaluate.

Translating Infix to Postfix

TABLE 4.7 Translating A+B*C to Postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Comments
A	A	A	
+	A+	A	
B	A+B	AB	
*	A+B*	AB	You can't copy the + because * is higher precedence than +.
C	A+B*C	ABC	When you see the C, you can copy the *.
	A+B*C	ABC*	
End	A+B*C	ABC*+	When you see the end of the expression, you can copy the +.

TABLE 4.8 Translating A*(B+C) into Postfix

Character Read from Infix Expression	Infix Expression Parsed so Far	Postfix Expression Written So Far	Comments
A	A	A	
*	A*	A	
(	A*(	A	
B	A*(B	AB	You can't copy * because of the parenthesis.
+	A*(B+	AB	
C	A*(B+C	ABC	You can't copy the + yet.
)	A*(B+C)	ABC+	When you see the), you can copy the +.
	A*(B+C)	ABC+*	After you've copied the +, you can copy the *.
End	A*(B+C)	ABC+*	Nothing left to copy.

TABLE 4.9 Translating A+B*(C–D) to Postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Stack Contents
A	A	A	
+	A+	A	+
B	A+B	AB	+
*	A+B*	AB	+*
(	A+B*(	AB	+*(
C	A+B*(C	ABC	+*(
–	A+B*(C–	ABC	+*(–
D	A+B*(C–D	ABCD	+*(–
)	A+B*(C–D)	ABCD–	+*(
	A+B*(C–D)	ABCD–	+*(
	A+B*(C–D)	ABCD–	+*
	A+B*(C–D)	ABCD–*	+
	A+B*(C–D)	ABCD–*+	

TABLE 4.10 Infix to Postfix Translation Rules

Item Read from Input (Infix)	Action
Operand	Write it to output (postfix)
Open parenthesis (	Push it on stack
Close parenthesis)	While stack not empty, repeat the following:
	Pop an item,
	If item is not (, write it to output
	Quit loop if item is (
Operator (`opThis`)	If stack empty,
	Push `opThis`
	Otherwise,
	While stack not empty, repeat:
	Pop an item,
	If item is (, push it, or
	If item is an operator (`opTop`), and
	If `opTop < opThis`, push `opTop`, or
	If `opTop >= opThis`, output `opTop`
	Quit loop if `opTop < opThis` or item is (
	Push `opThis`
No more items	While stack not empty,
	Pop item, output it.

TABLE 4.11 Translation Rules Applied to A+B–C

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to output.
+	A+	A	+	If stack empty, push `opThis`.
B	A+B	AB	+	Write operand to output.
–	A+B–	AB		Stack not empty, so pop item.
	A+B–	AB+		`opThis` is –, `opTop` is +, `opTop>=opThis`, so output `opTop`.
	A+B–	AB+	–	Then push `opThis`.
C	A+B–C	AB+C	–	Write operand to output.
End	A+B–C	AB+C–		Pop leftover item, output it.

TABLE 4.12 Translation Rules Applied to A+B*C

Character Read From Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to postfix.
+	A+	A	+	If stack empty, push `opThis`.
B	A+B	AB	+	Write operand to output.
*	A+B*	AB	+	Stack not empty, so pop `opTop`.
	A+B*	AB	+	`opThis` is *, `opTop` is +, `opTop<opThis`, so push `opTop`.
	A+B*	AB	+*	Then push `opThis`.
C	A+B*C	ABC	+*	Write operand to output.
End	A+B*C	ABC*	+	Pop leftover item, output it.
	A+B*C	ABC*+		Pop leftover item, output it.

TABLE 4.13 Translation Rules Applied to A*(B+C)

Character Read From Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to postfix.
*	A*	A	*	If stack empty, push `opThis`.
(	A*(	A	*(	Push (on stack.
B	A*(B	AB	*(	Write operand to postfix.
+	A*(B+	AB	*	Stack not empty, so pop item.
	A*(B+	AB	*(	It's (, so push it.
	A*(B+	AB	*(+	Then push `opThis`.
C	A*(B+C	ABC	*(+	Write operand to postfix.
)	A*(B+C)	ABC+	*(	Pop item, write to output.
	A*(B+C)	ABC+	*	Quit popping if (.
End	A*(B+C)	ABC+*		Pop leftover item, output it.

Evaluating Postfix Expressions

TABLE 4.14 Evaluating a Postfix Expression

Item Read from Postfix Expression	Action
Operand	Push it onto the stack.
Operator	Pop the top two operands from the stack and apply the operator to them. Push the result.

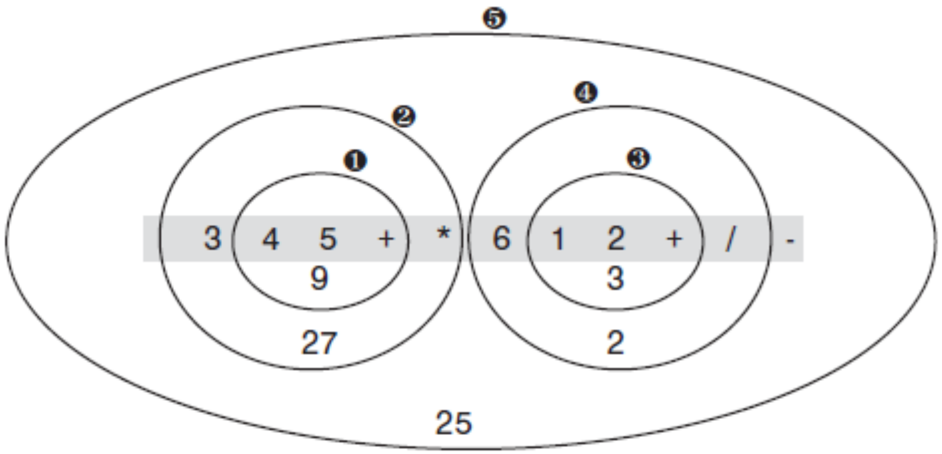


FIGURE 4.16 Visual approach to postfix evaluation of 345+*612+/–.

The End