EECS 560
DATA STRUCTURES

MODULE II: QUEUE AND STACK

DISCLAIMER

- This document is intended to be used for studying EECS 560 Data Structures, only by students who are currently enrolled in the course.

-

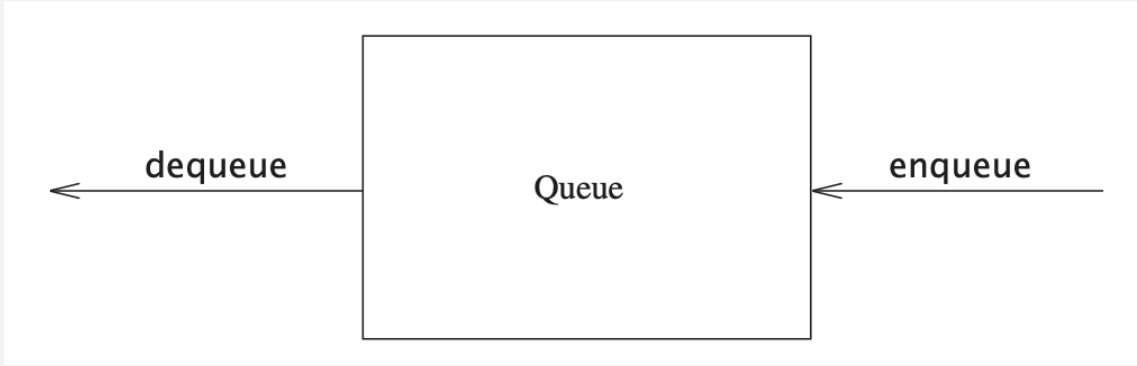
- If you disagree, please delete the document immediately.

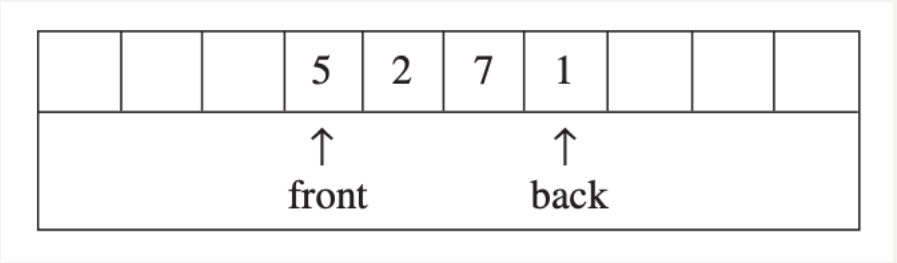
ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Data Structures and Algorithm Analysis in C++***, 4th edition, by Mark Allen Weiss.

QUEUE

- Queue is an extension of the list ADT that has the "first in first out" behavior (while the other operations remain the same as previously discussed)
- Two essential operations of queue are:
  - enqueue (or push): insert an element into the list at the last position (also called "back")
  - dequeue: remove the first element (also called "front") of the list (its content is also returned)





QUEUE APPLICATIONS

- It can serve as a "buffer" that resembles a real queue
  - Imagine you are waiting in a line for checkout
  - the line can be modeled as a queue
  - the first arriving customer goes to checkout first, representing the "first in first out" model
- More applications in graph traversal (discussed later in the graph ADT module)

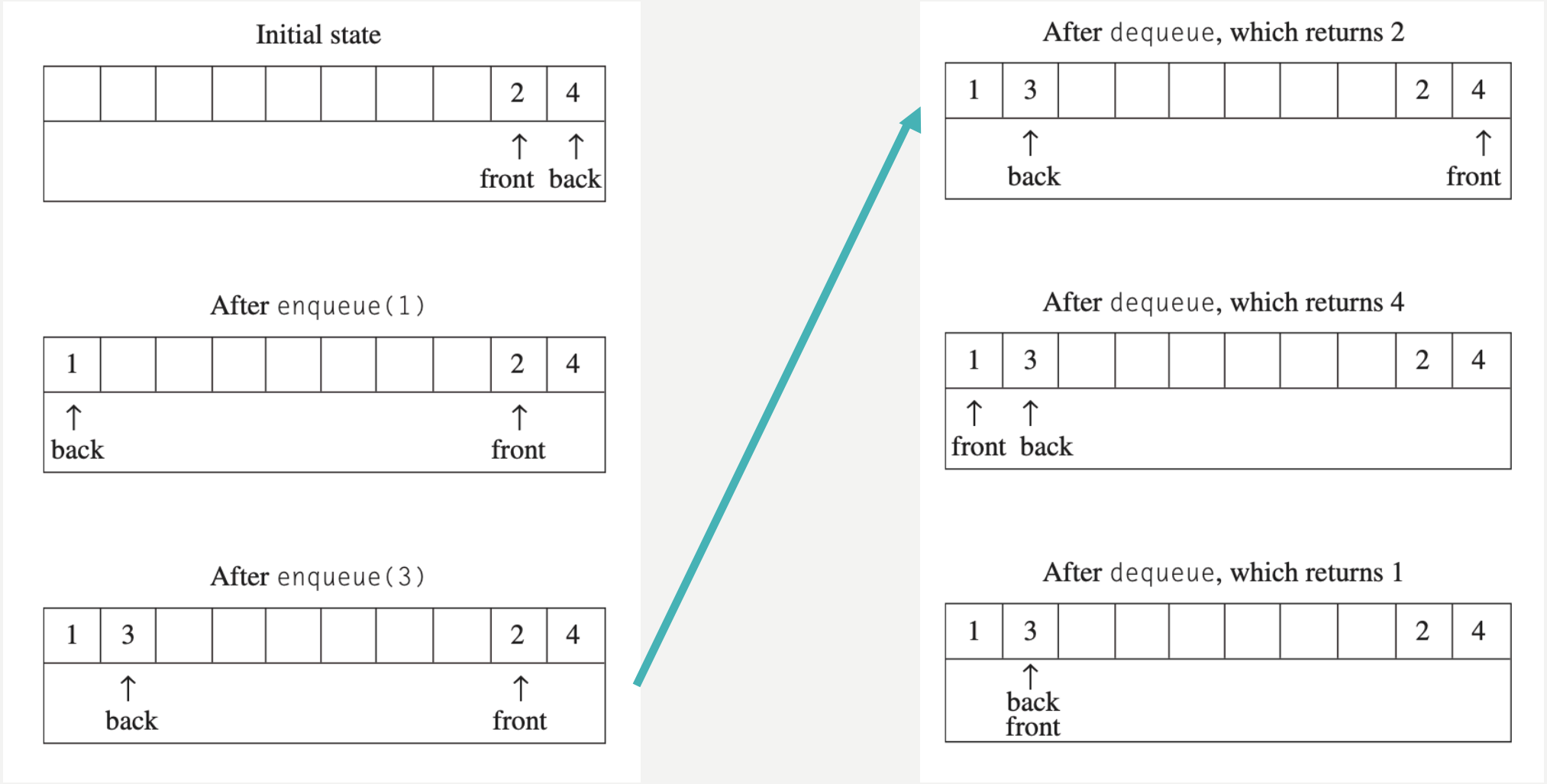
QUEUE IMPLEMENTATION (LINKED LIST)

- The implementation is straightforward (and we will do that as a lab)
  - enqueue(x): this can be implemented as insert(x, theSize), i.e., inserting x at the last position
  - dequeue(): this can be implemented as delete(1), i.e., deleting the element before the second element
  - both operations clearly take O(1) time

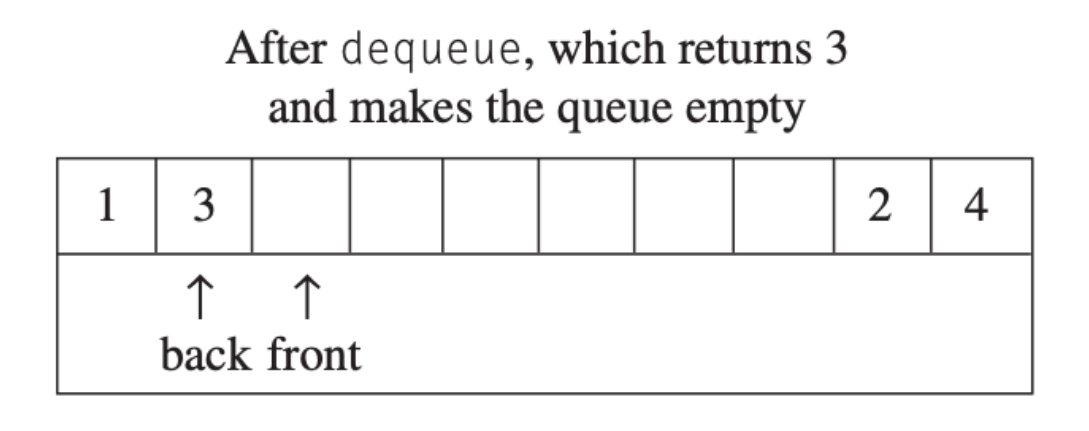
QUEUE IMPLEMENTATION (ARRAY)

- The implementation will work on a fix-sized array
- We will use two pointers (front and back) to mark the boundaries of the list
  - enqueue: move forward (towards the right) the back pointer
  - dequeue: move forward (towards the right) the front pointer
  - note that the two pointers do not necessarily point to the starting and ending positions of the array

QUEUE IMPLEMENTATION (ARRAY)



QUEUE IMPLEMENTATION (ARRAY)



- Note: this also indicates the initial state when the list is empty. And we can represent an empty list by setting the front pointer one position after the back pointer (which position the back pointer is pointing at does not matter).

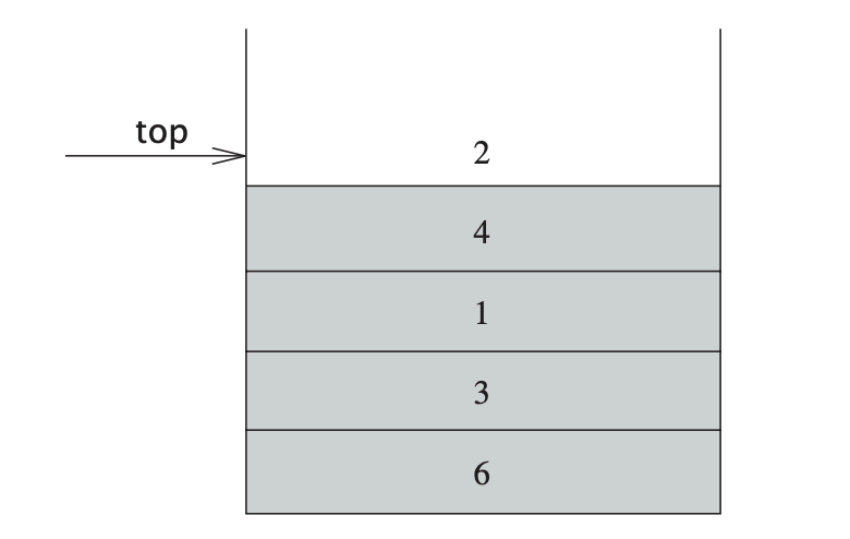
QUEUE IMPLEMENTATION (ARRAY)

- When resizing the array, we need to copy all elements in the current array (between the front and the back pointers, inclusively) to the new array.

- Other STL supported queue operations:
  - front(): returns the value pointed by the front pointer (without modifying it)
  - back(): returns the value pointed by the back pointer (without modifying it)

STACK

- Stack is also an extension of the list ADT.
- Unlike the queue's "first in first out" behavior, the stack supports the "first in last out"(or equivalently "last in first out") operations
  - push: inserting an element to the last position of the list
  - pop: deleting the last element in the list (and also return its value)

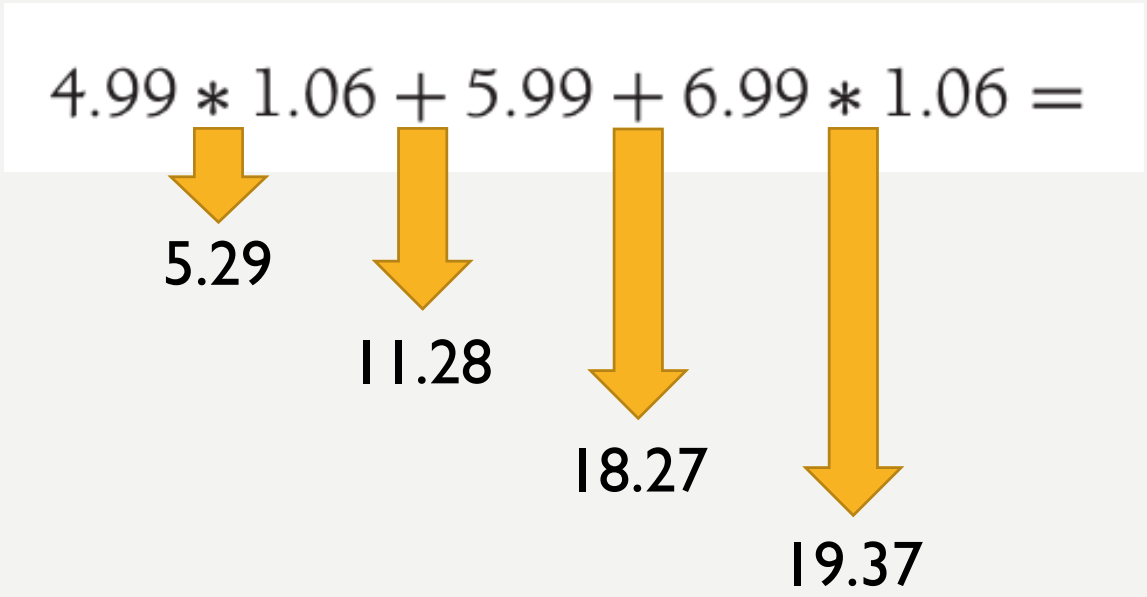


STACK APPLICATIONS

- Arithmetic computation
  - helps the computer parse arithmetic expressions to determine the correct execution order of different operators
- More applications in graph traversal (discussed later in the graph ADT module)

STACK APPLICATIONS

- Consider the following example. If we simply compute the operations from left to right, we are likely getting a wrong answer of 19.37



- But we know the correct execution order should be (4.99*1.06)+5.99+(6.99*1.06)=18.69

STACK APPLICATION

- The computer uses stack to determine the correct computation order, according to the priorities of different operators in two steps
  - first, convert the infix expression (what we regularly write) into the postfix expression
  - second, compute the value using the postfix expression
- For simplicity, we assume only three operations
  - (): parenthesis, which has the highest priority
  - *: multiplication, which has a lower priority
  - +: addition, which has the lowest priority
  - the minus operation can be viewed as adding the corresponding negative operand, while the division operation can be viewed as multiplying the corresponding reciprocal operand

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

- Consider the following infix expression

```
a + b * c + ( d * e + f ) * g
```

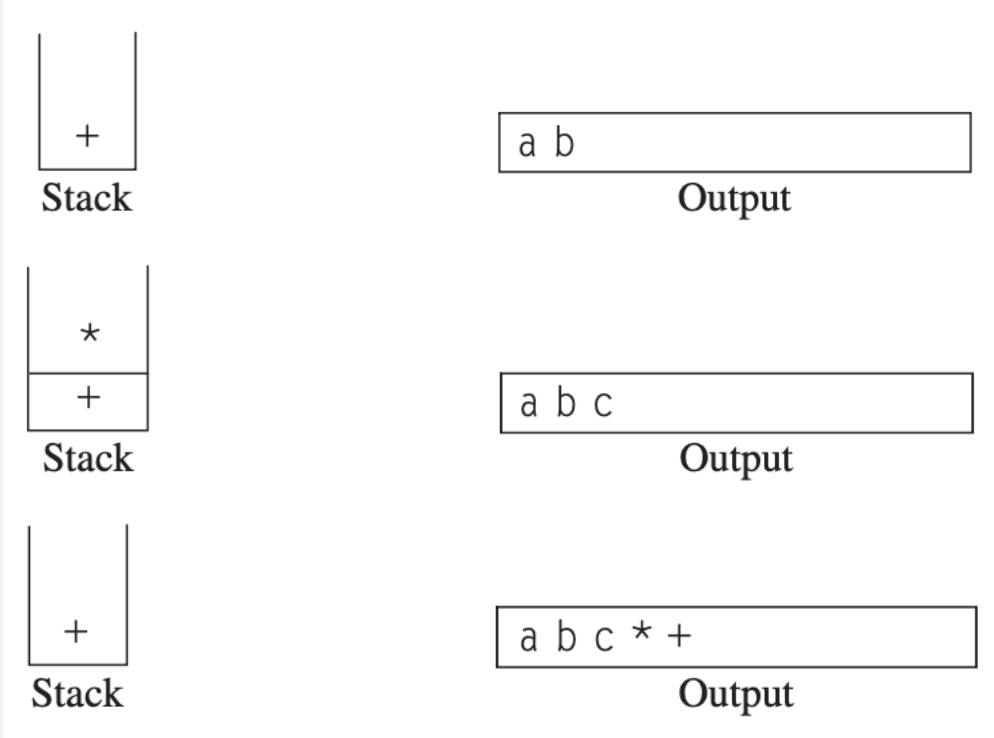
- We will convert it to the postfix form by scanning the expression from left to right, and apply the following rules:
  - if we see an operand, we output it
  - if we see an operator, we pop all operators in the stack that **have the same or higher priorities** than it and write them to the output. After that we push the current operator into the stack
    - exception: if the operator is a right parenthesis, we pop every operator until we see a left parenthesis (which will also be popped)
  - when all symbols are read, pop all symbols in the stack out and write to the output

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

- Why would the algorithm work?
  - we can interpret the postfix expression as an ordered execution of a set of operations. That is, the earlier we see an operator in the postfix expression, the earlier we execute the operation.
  - so, when we pop the operators, we ensure that the operators that are being popped have the same or higher priority than then current operator. It indicates that we allow operators that with higher priorities to execute earlier.
  - we will go through a concrete example

STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

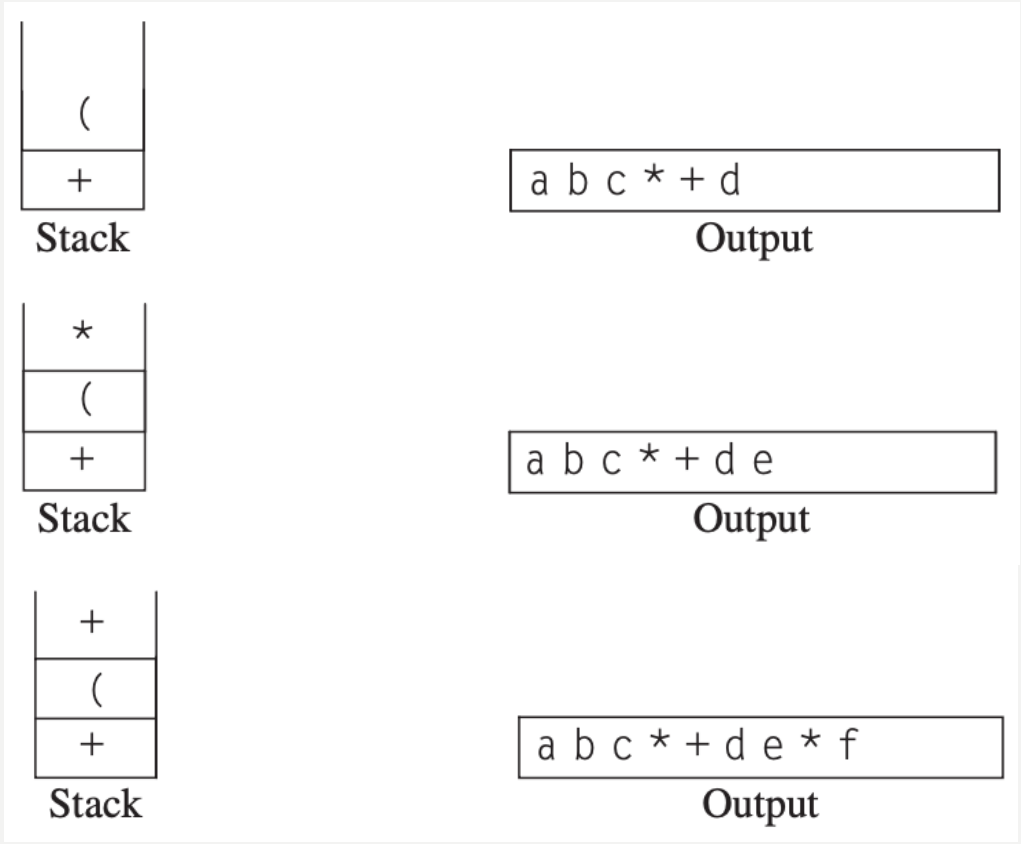
```
a + b * c + ( d * e + f ) * g
```



// when we see the third operator "+", we pop the second operator "*" and the first operator "+" out because they have the same or higher priority than "+"

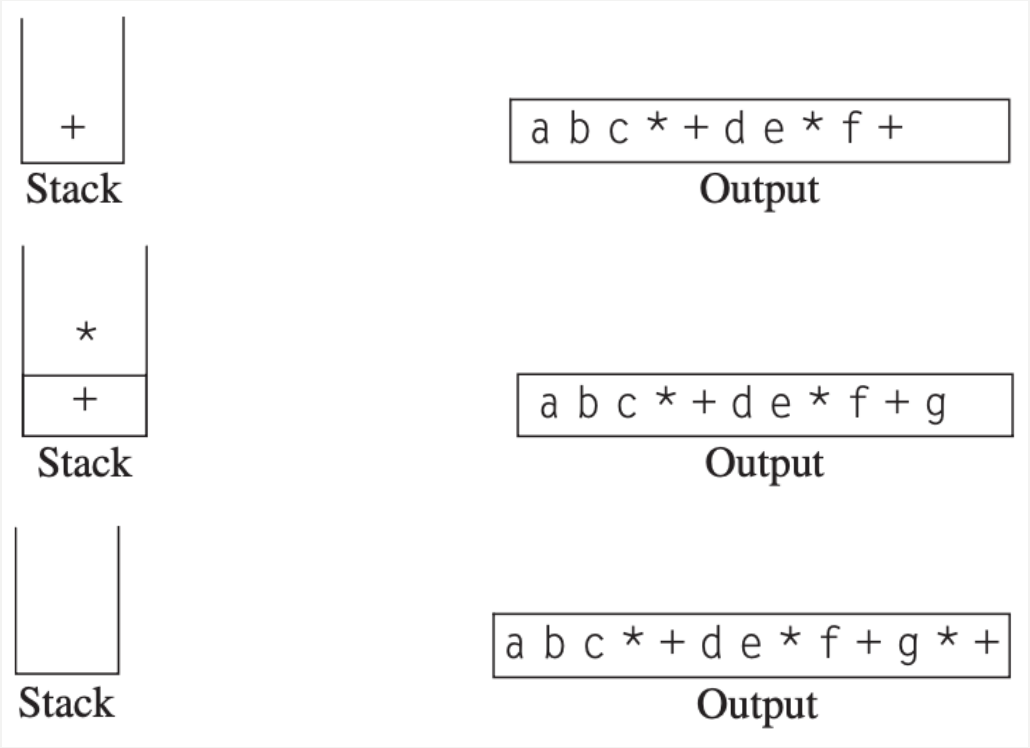
STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

a + b * c + (d * e + f) * g



STACK APPLICATION (INFIX TO POSTFIX CONVERSION)

a + b * c + (d * e + f) * g



STACK APPLICATION (POSTFIX COMPUTATION)

- Now we have converted the infix expression to postfix expression
  - infix: a+b*c+(d*e+f)*g
  - postfix: abc*+de*f+g*+

- The next step is to compute the actual value using the postfix expression using the following steps (also involve stack)
  - we scan the postfix expression from left to right
  - if we see an operand, we push it to the stack
  - if we see an operator, we pop two operands from the stack and perform the corresponding operation, and push the resulted number back to the stack

STACK APPLICATION (POSTFIX COMPUTATION)

- consider the postfix expression we just got: abc*+de*f+g*+

		c				e		f		g	
	b	b	(b*c)		d	d	(d*e)	(d*e)	(d*e)+f	(d*e)+f	((d*e)+f)*g
a	a	a	a	a+(b*c)	a+(b*c)	a+(b*c)	a+(b*c)	a+(b*c)	a+(b*c)	a+(b*c)	a+(b*c)

And finally we get: (a+(b*c))+((d*e)+f)*g

STACK APPLICATION (POSTFIX COMPUTATION)

- compare the results from parsing postfix expression and the original infix expression
  - postfix parsing: (a+(b*c))+((d*e)+f)*g
  - infix: a+b*c+(d*e+f)*g

- We can easily confirm that the computation is correct

SUMMARY

- Queue and stack are extensions of the original list data structure
  - queue: first in first out
  - stack: last in first out

- Many important applications, and are considered as two fundamental data structures