# Stacks and Queues

Data Structures

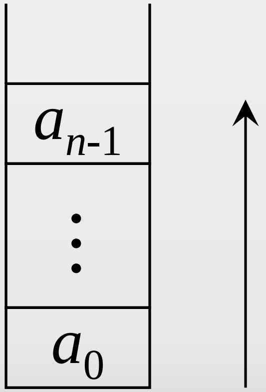
Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

# Introduction

- The stack and the queue are both special cases of ordered list.
- Given an ordered list $A = a_0, a_1, \ldots, a_{n\text{-}1}$, each $a_i$ is called an atom or an element.
  - The empty list is denoted by ().

# The Stack Abstraction Data Type

- A *stack* is an ordered list in which insertions and deletions are made at one end called *top*.
- Given a stack $S = (a_0, \ldots, a_{n-1})$, we say that $a_0$ is the bottom element and $a_{n-1}$ is the top element.



# The Stack Abstraction Data Type (contd.)

- Example: the sequence of insertion operations (p. 108, Fig. 3.1)
  - A *Last-In-First-Out* (*LIFO*) list
- The system stack is an application of the stack.
  - Used at run-time to process function calls
  - Activation records (AR) or stack frames: elements of the system stack

| |
|---|
| local variables |
| parameters |
| prev. AR ptr. |
| return address |

# The Stack Abstraction Data Type (contd.)

- Each time when a subprogram is invoked, the invoking subprogram creates an AR and places it on top of the system stack (p. 109, Fig. 3.2).
  - Initially, the AR for the invoked subprogram contains only a pointer to the previous AR and a return address.
    - prev. AR ptr. -- pointing to the caller′s AR
    - return address -- the location of the statement to be executed after the subprogram terminates
  - If this subprogram invokes another one, the local variables, except those declared static, and the parameters of the caller are added to its AR.
  - When this subprogram terminates, its AR is removed.

# The Stack Abstraction Data Type (contd.)

- The ADT specification of the stack structure (p. 110, ADT 3.1)
- The easiest way to implement a stack is by using an one-dimensional array.
  - e.g., *stack*[*MAX_STACK_SIZE*]
  - *stack*[0] is the bottom and the *i*th element is *stack*[*i*-1]
  - *top* -- an associated variable indicating the index of the top element in the stack; initial value is -1
- Relevant implementations (p.109~111)
  - push / pop

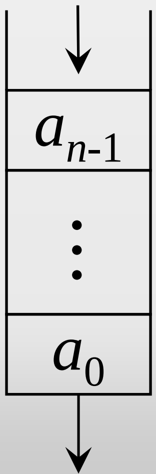
# The Stack Abstraction Data Type (contd.)

- push: *top* = *top* + 1; data insertion
- pop: retrieving data; *top* = *top* - 1
- Extraordinary cases
  - Underflow
  - Overflow

- Other applications of stacks
  - A mazing problem (backtracking)
  - Expressions evaluation

# The Queue Abstraction Data Type

- A *queue* is an ordered list in which all insertions take place at one end and all deletions take place at the opposite end.
- Example: insertions and deletions (p. 114, Fig. 3.4)
- The first element inserted into a queue is the first element removed.
  - First-In-First-Out (FIFO) lists



# The Queue Abstraction Data Type (contd.)

- The ADT specification of the queue structure (p. 115, ADT 3.2)
- The simplest way to implement a queue is by using an one-dimensional array and two variables, *front* and *rear*.
  - The *front* index is smaller than the index of the first-in element by one.
  - The *rear* index points to the current end of the queue.
  - The initial values are both -1 to indicate an empty state.

# The Queue Abstraction Data Type (contd.)

- Implementation of operations (p. 114~116)
  - insert (add) and delete
  - insert: *rear* = *rear* + 1; data insertion
  - delete: *front* = *front* +1; data retrieval
- When *rear* equals *MAX_QUEUE_SIZE* - 1, *queue_full* is triggered to move the entire queue to the left
  - The worst case complexity of *queue_full* is O(*MAX_QUEUE_SIZE*).

- A variant: circular queues
  - More efficient (p. 117, Fig 3.6, p.118~119)

# The Queue Abstraction Data Type (contd.)

- The initial values of *front* and *rear* are 0 instead of -1.
  - The *front* index always points one position counterclockwise from the first element in the queue.
- To distinguish between an empty and a full state, a circular queue of size *MAX_QUEUE_SIZE* can hold at most *MAX_QUEUE_SIZE*-1 elements.

- Other variants of queues
  - Double-ended queues (dequeue)
  - Priority queues
  - Double-ended priority queues

# Evaluation of Expressions

- Within any programming language, there is a precedence hierarchy of operators.
  - C (p.130, Fig. 3.12)
- Compilers typically use postfix notation for expressions evaluation.
  - Parenthesis-free
- Infix notation is the most common way of writing expressions, even for programmers.

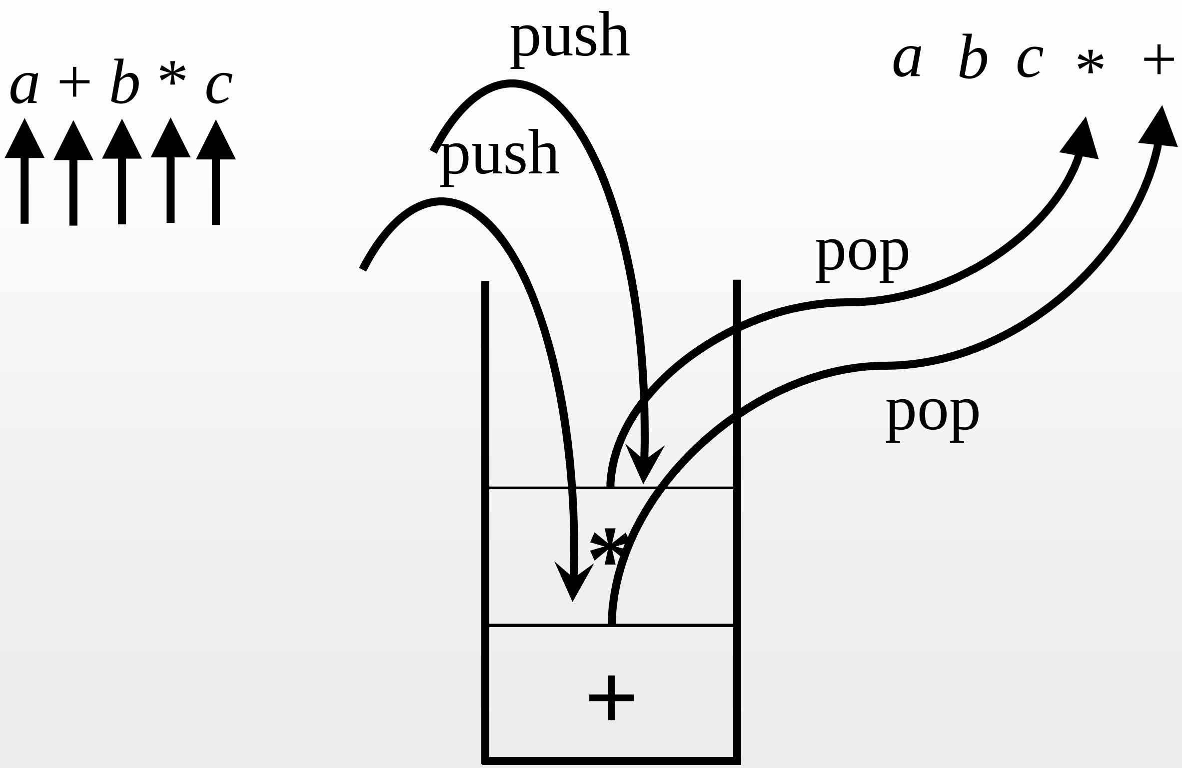
# Evaluation of Expressions (contd.)

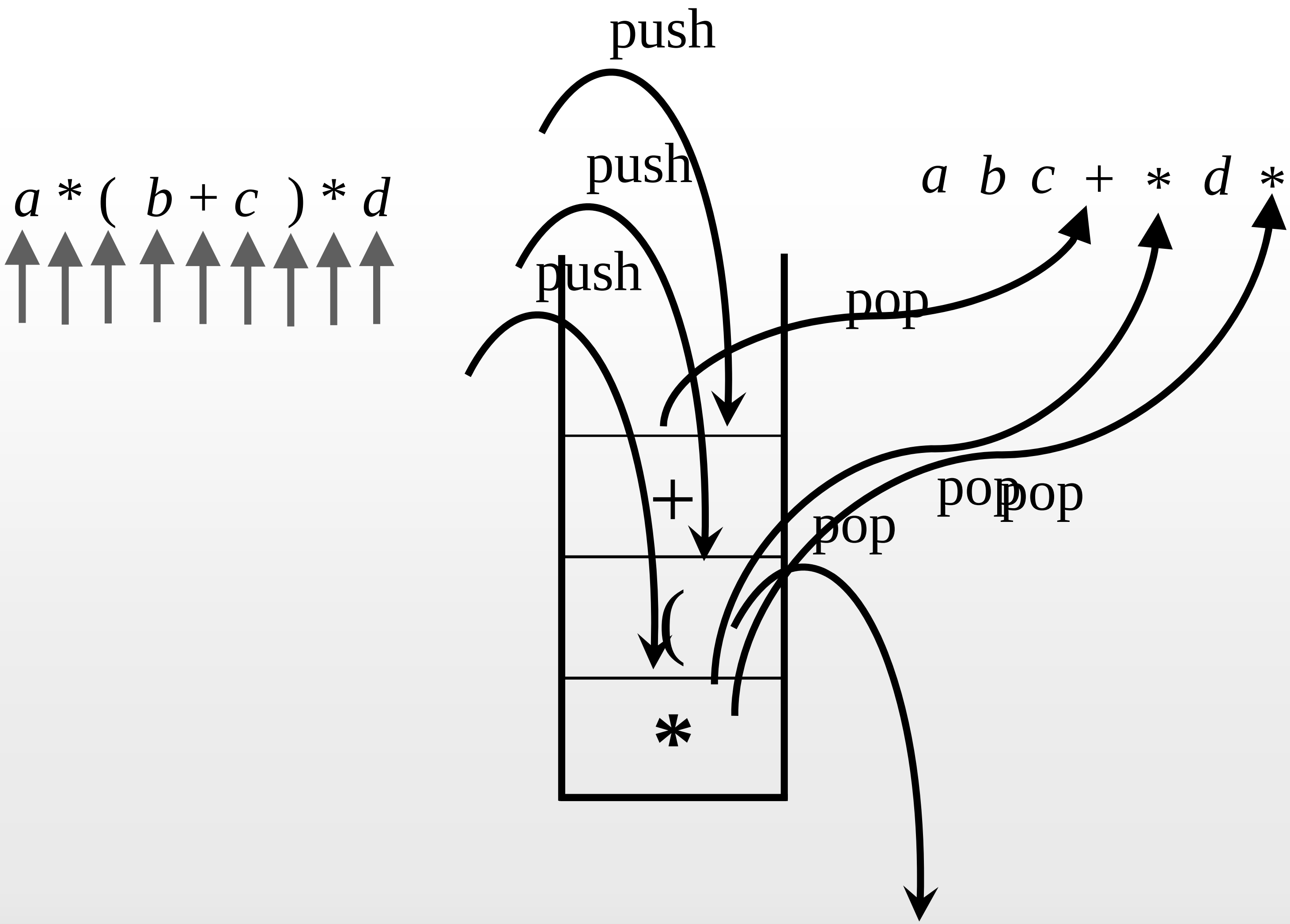
- So, compilers use a two-stage processing for expressions evaluation.
  - Stage 1: Infix to postfix
  - Stage 2: Evaluating postfix expressions
- Infix to postfix
  - The order of operands is the same in infix and postfix.
    - Operands are passed to the output expression.
  - The order in which the operators are output depends on their precedence.

# Evaluation of Expressions (contd.)

- Without consideration of parentheses
  - Compare the precedence of top operator and that of incoming operator.
  - If the latter is lower, pop the former and repeat this operation until the incoming operator has higher precedence than the stack top.
  - At last push the incoming one into the stack
- With consideration of parentheses
  - *in-stack precedence* and *incoming precedence*
  - The left parenthesis is a lowest-precedence operator on the stack while possessing highest precedence as an incoming one.

- p. 137, Program 3.15 ($\Theta(n)$, n: # of tokens)


a + b * c
push
push
a b c * +
pop
pop
*
+

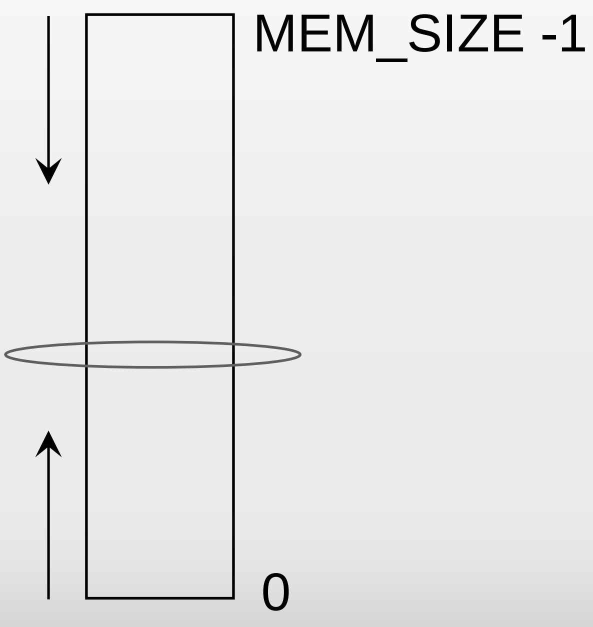

a * ( b + c ) * d
push
push
push
a b c + * d *
pop
pop
pop
pop
+
(
*

# Evaluation of Expressions (contd.)

- Evaluating postfix expressions
  - The operands are stored on a stack until they are needed.
  - For an operator, remove two operands from the stack, perform the specified operation, and then push the result back to the stack.

# Multiple Stacks

- Two stacks only
  - A stack grows upwards and the other one is toward the opposite direction.
  - Overflow check



# Multiple Stacks (contd.)

- More than two stacks
  - Divide the available memory into $n$ segments.
    - In proportion to the expected sizes of the various stacks
    - Equal segments (p. 140, Fig. 3.18)
  - Problem: Some stack $i$ overflows while there are free space in the array.
    - Local overflow, but not global overflow
    - Solution 1: Moving stacks with ID greater than $i$ to the right as possible (p.140, (1)).
    - Solution 2: Moving stacks with ID smaller than $i$ to the left as possible (p.141, (2)).