Stacks and Queues

Data Structures

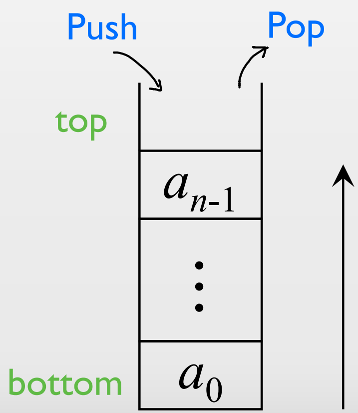
Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Introduction

- The stack and the queue are both special cases of ordered list.
- Given an ordered list $A = a_0, a_1, \ldots, a_{n-1}$, 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*.
  取拿都在同一端
- 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 後者進先出
- 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
    大小不固定

	local variables	隨著輸入改變
	parameters	傳入參數（個數、大小）
記前一個的位置	prev. AR ptr.	
可預測	return address	caller等下繼續執行的指令

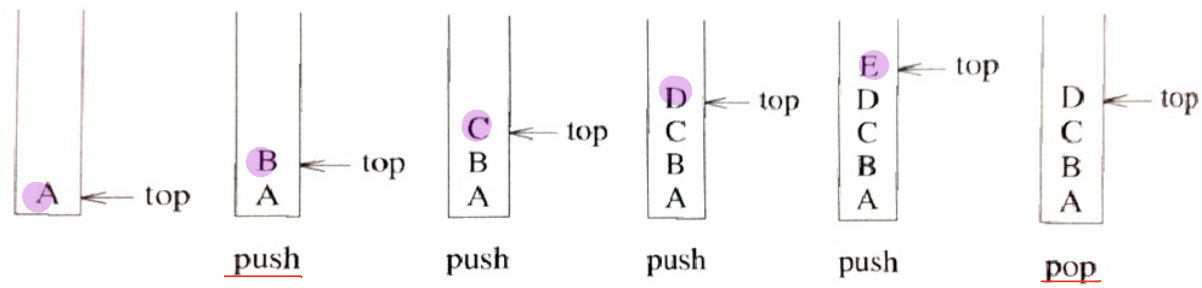


Figure 3.1: Inserting and deleting elements in a stack

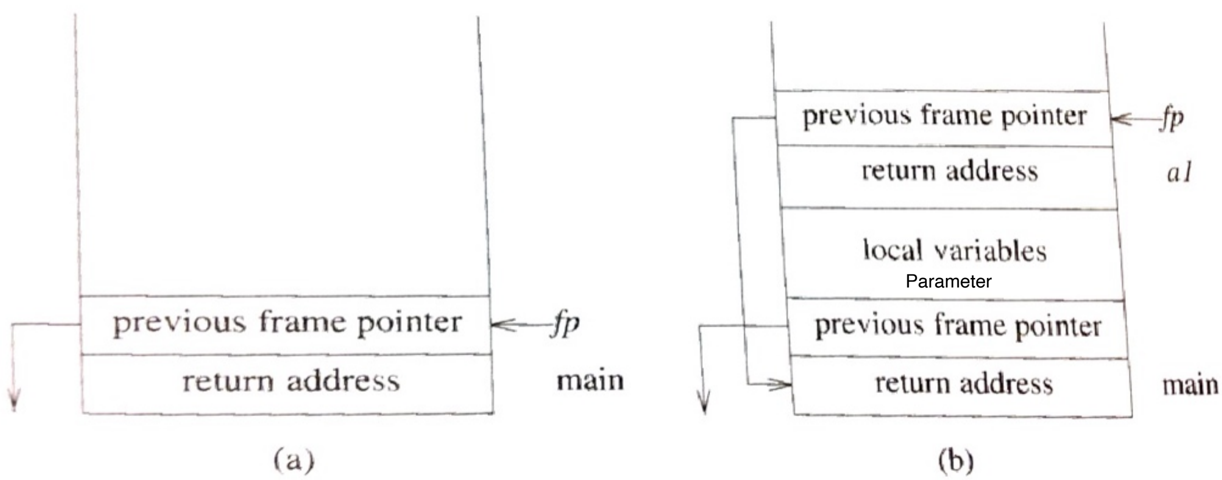


Figure 3.2: System stack after function call

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).

Push
  - 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.

    不包括global variable(前面有static的視為全域變數)

Pop
  - When this subprogram terminates, its AR is removed.

    自己產生自己清除

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

Stack是空的記為-1（因為起始記為0）
- Relevant implementations (p.109~111)
  - push / pop

ADT *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *stack* ∈ *Stack*, *item* ∈ *element*, *maxStackSize* ∈ positive integer
    Stack CreateS(*maxStackSize*) ::=
        create an empty stack whose maximum size is *maxStackSize*
    Boolean IsFull(*stack*, *maxStackSize*) ::=
        if (number of elements in *stack* == *maxStackSize*)
        return *TRUE*
        else return *FALSE*
    Stack Push(*stack*, *item*) ::=
        if (IsFull(*stack*)) *stackFull* 檢查是否滿的才能push
        else insert *item* into top of *stack* and **return**
    Boolean IsEmpty(*stack*) ::=
        if (*stack* == CreateS(*maxStackSize*))
        return *TRUE*
        else return *FALSE*
    Element Pop(*stack*) ::=
        if (IsEmpty(*stack*)) **return** 檢查是否空的才能pop
        else remove and return the element at the top of the stack.

ADT 3.1: Abstract data type *Stack*

```
void push(element item)
{/* add an item to the global stack */
   if (top >= MAX_STACK_SIZE-1)
      stackFull();
   stack[++top] = item;
}
```

Program 3.1: Add an item to a stack

```
element pop()
{/* delete and return the top element from the stack */
   if (top == -1)
      return stackEmpty(); /* returns an error key */
   return stack[top--];
}
```

Program 3.2: Delete from a stack

```
void stackFull()
{
   fprintf(stderr, "Stack is full, cannot add element");
   exit(EXIT_FAILURE);
}
```

Program 3.3: Stack full

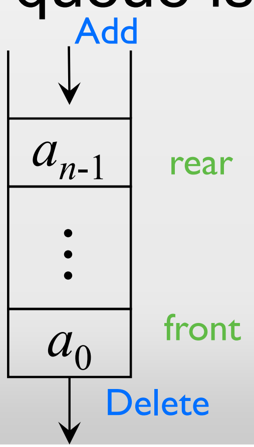
The Stack Abstraction Data Type (contd.)

- push: *top* = *top* + 1; data insertion先改top再放值
- pop: retrieving data; *top* = *top* - 1 先取值再改top
- Extraordinary cases 使用IsEmpty / IsFull 保護機制
  - Underflow 已經空了再做pop
  - Overflow 已經滿了再做push

- Other applications of stacks
  - A mazing problem (backtracking)
  - Expressions evaluation ex. a + b * c （compiler如何做運算）

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. 取拿在不同端
- 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
    先進先出



A	A B	A B C	A B C D	A B C D E	B C D E
↑	↑ ↑	↑ ↑	↑ ↑	↑ ↑	↑ ↑
f,r	f r	f r	f r	f r	f r
	add	add	add	add	delete

f = queue front 頭 r = queue rear 尾

Figure 3.4: Inserting and deleting elements in a queue

ADT *Queue* **is**

objects: a finite ordered list with zero or more elements.

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*, *maxQueueSize* ∈ positive integer

Queue CreateQ(*maxQueueSize*) ::=
    create an empty queue whose maximum size is *maxQueueSize*

Boolean IsFullQ(*queue*, *maxQueueSize*) ::=
    if (number of elements in *queue* == *maxQueueSize*)
    return *TRUE*
    else return *FALSE*

Queue AddQ(*queue*, *item*) ::=
    if (IsFullQ(*queue*)) *queueFull*
    else insert *item* at rear of *queue* and return *queue*

Boolean IsEmptyQ(*queue*) ::=
    if (*queue* == CreateQ(*maxQueueSize*))
    return *TRUE*
    else return *FALSE*

Element DeleteQ(*queue*) ::=
    if (IsEmptyQ(*queue*)) **return**
    else remove and return the *item* at front of queue.

ADT 3.2: Abstract data type *Queue*

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. 第1個元素前1個
  - 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*, *queue_full* is triggered to move the entire queue to the left
  - The worst case complexity of *queue_full* is O(*MAX_QUEUE_SIZE*).

直線型queue缺點：Array滿了但不一定代表queue真的滿了

- A variant: circular queues 在空間利用較有效率，直到完全無空間
  - More efficient (p. 117, Fig 3.6, p.118~119)

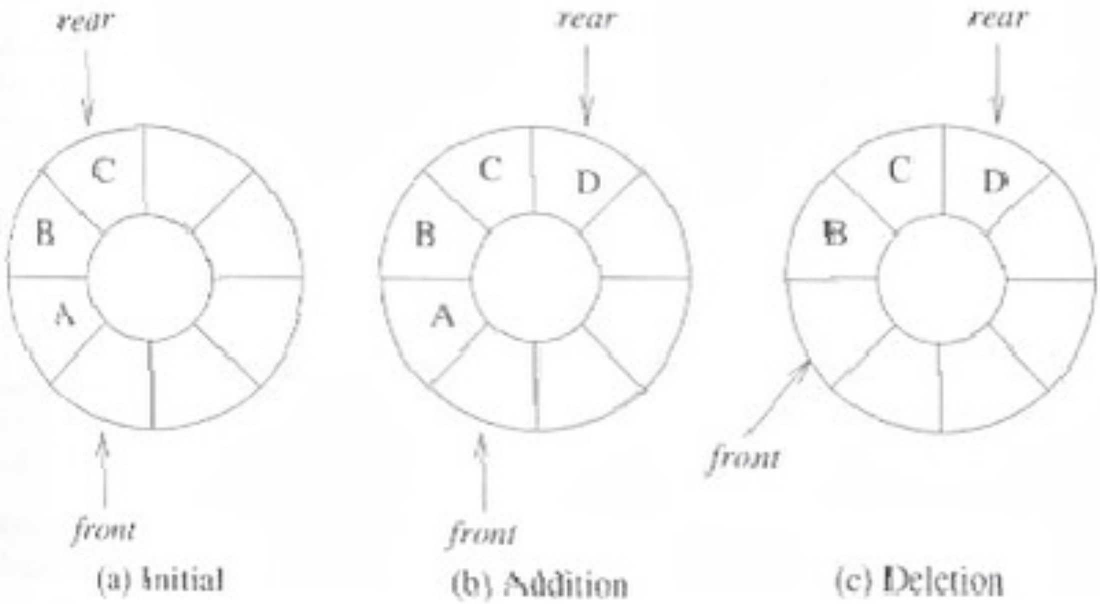


Figure 3.6: Circular queue

```
void addq(element item)
{/* add an item to the queue */
   rear = (rear+1) % MAX_QUEUE_SIZE;
   if (front == rear)
      queueFull(); /* print error and exit */
   queue[rear] = item;
}
```

Program 3.7: Add to a circular queue

```
element deleteq()
{/* remove front element from the queue */
  element item;
    if (front == rear)
      return queueEmpty(); /* return an error key */
    front = (front+1) % MAX_QUEUE_SIZE;
    return queue[front];
}
```

Program 3.8: Delete from a circular queue

The Queue Abstraction Data Type (contd.)

- The initial values of *front* and *rear* are 0 instead of -1.

  當queue是空的front和rear會疊在一起，
  當queue滿了front和rear會相鄰

  - 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.

  缺點：為了要鑑別空滿，要空下一格不放東西

- Other variants of queues

  當數量少時空下一格影響較大，
  反之數量多影響較小

  - Double-ended queues (deques)
  - Priority queues 先看元素間優先權高低，再依FIFO
  - 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. 不使用括號，即能表示先後運算次序
  - Parenthesis-free 不需括號表示法，將infix轉為postfix
- Infix notation is the most common way of writing expressions, even for programmers.

Token	Operator	Precedence[1]	Associativity
() [] → .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

Figure 3.12: Precedence hierarchy for C

Evaluation of Expressions (contd.)

- So, compilers use a two-stage processing for expressions evaluation.
  - Stage 1: Infix to postfix 表示法轉換
  - Stage 2: Evaluating postfix expressions
- Infix to postfix
  - The order of operands is the same in infix and postfix. Operand出現次序不會改變，看到就輸出
    - Operands are passed to the output expression.
  - The order in which the operators are output depends on their precedence. Operator處理，查表

Evaluation of Expressions (contd.)

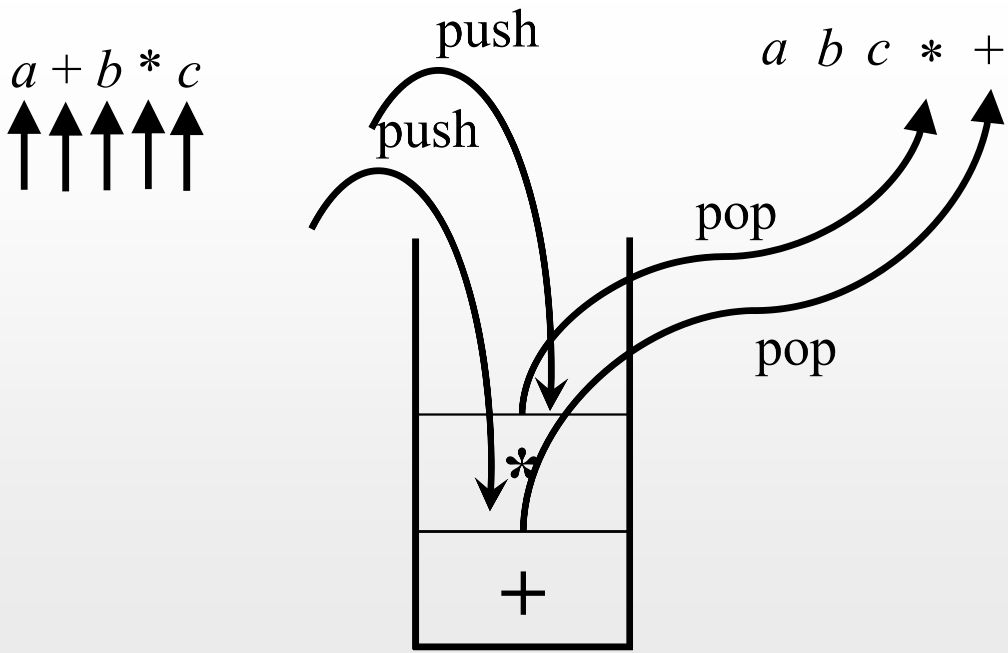
無括號 ◆ Without consideration of parentheses

- Compare the precedence of top operator and that of incoming operator. 查表比較，直到incoming operator precedence高於top operator為止,將top pop出
- 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 Pop直到是空的

有括號 ◆ With consideration of parentheses

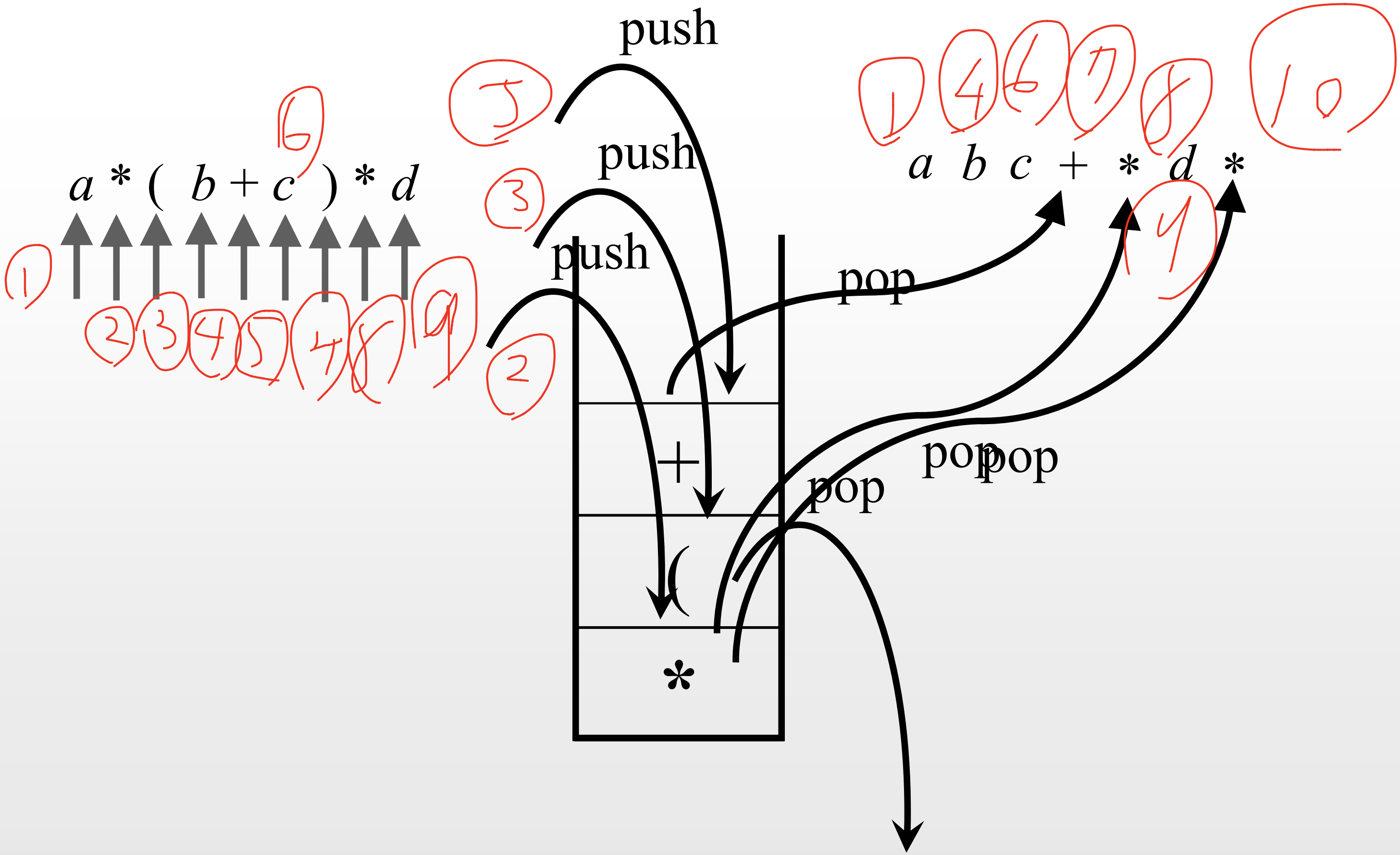
- *in-stack precedence* and *incoming precedence*
- The left parenthesis is a lowest-precedence operator on the stack while possessing highest precedence as an incoming one. 左括號唯一例外，在 in-stack precedence 和 incoming precedence 不同

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


$a + b * c$
push
push
$a\ b\ c\ *\ +$
pop
pop
*
+

```
void postfix(void)
{/* output the postfix of the expression. The expression
    string, the stack, and top are global */
  char symbol;
  precedence token;
  int n = 0;
  int top = 0;    /* place eos on stack */
  stack[0] = eos;
  for (token = getToken(&symbol, &n); token != eos;
                          token = getToken(&symbol,&n)) {
    if (token == operand)
      printf("%c", symbol);
    else if (token == rparen) {
      /* unstack tokens until left parenthesis */
      while (stack[top] != lparen)
        printToken(pop());
      pop();  /* discard the left parenthesis */
    }
    else {
      /* remove and print symbols whose isp is greater
         than or equal to the current token's icp */
      while(isp[stack[top]] >= icp[token])
        printToken(pop());
      push(token);
    }
  }
  while ( (token = pop()) != eos)
    printToken(token);
  printf("\n");
}
```

Program 3.15: Function to convert from infix to postfix

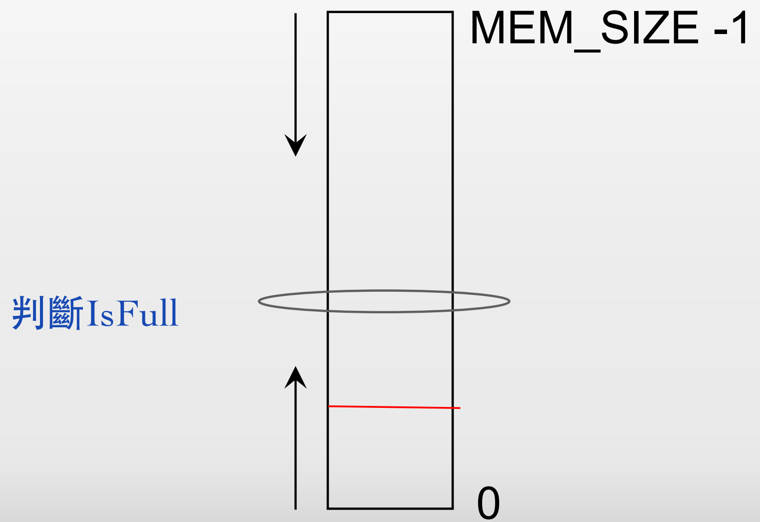

a * (b + c) * d
push
push
push
pop
pop
pop
pop
+
(
*
a b c + * d *

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) 分成等大的stack
  - Problem: Some stack i overflows while there are free space in the array.
    - Local overflow, but not global overflow 其它stack未放滿
    - 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)). bottom會移位

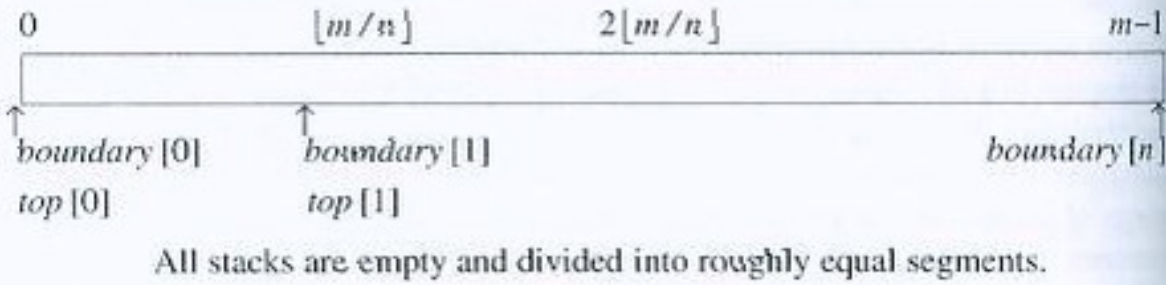


Figure 3.18: Initial configuration for n stacks in *memory* $[m]$.

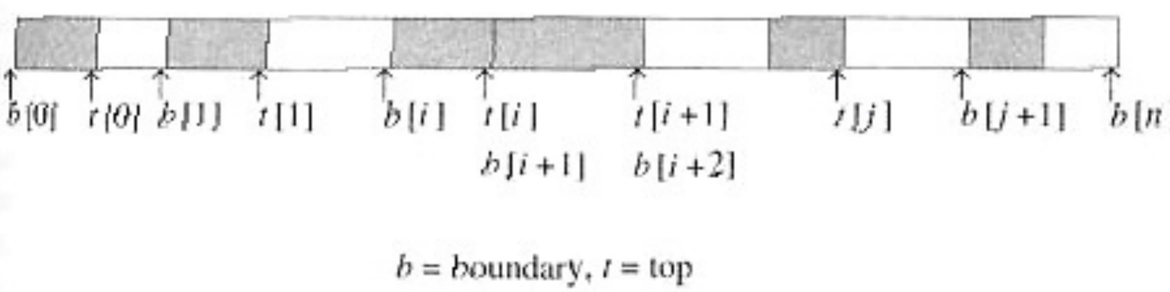


Figure 3.19: Configuration when stack i meets stack $i + 1$, but the memory is not full

(1) Determine the least, j, $i < j < n$, such that there is free space between stacks j and $j + 1$. That is, *top*$[j]$ < *boundary*$[j+1]$. If there is such a j, then move stacks $i+1$, $i+2$, $\cdots$, j one position to the right (treating *memory*[0] as leftmost and *memory*[*MEMORY_SIZE* − *1*] as rightmost). This creates a space between stacks i and $i+1$.

(2) If there is no j as in (1), then look to the left of stack i. Find the largest j such that $0 \le j < i$ and there is space between stacks j and $j+1$. That is, *top*$[j]$ < *boundary*$[j+1]$. If there is such a j, then move stacks $j+1$, $j+2$, $\cdots$, i one space to the left. This also creates a space between stacks i and $i+1$.