Chap 3. Stacks and Queues (3)

Contents

Contents

3.6 Evaluation of Expressions

3.6.1 Expressions

- Complex expressions
  - ((a+1==*b*)||((*d*==*MAX_SIZE*-1)&& !*flag*))
  - operators, operands, parentheses

- Complex assignment statements
  - x = a / b – c + d * e – a * c

- The order in which the operations are performed?

  - If a = 4, b = c = 2, d = e = 3,
    - x = ((a/b)-c)+(d*e)-(a*c) = ((4/2)-2)+(3*3)-(4*2) = 1
    - x = (a/(b-c+d))*(e-a)*c = (4/(2-2+3))*(3-4)*2 = -2.66666…

Token	Operator	Precedence[1]	Associativity
() [] → .	function call array element struct or union member	17	left-to-right
-- ++	decrement, increment[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

Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first.

3.6.2 Evaluating Postfix Expressions

- **Infix notation**
  - binary operator is in-between its two operands

- **Prefix notation**
  - operator appears before its operands

- **Postfix notation**
  - Each operator appears after its operands
  - Used by compiler
  - *Parentheses-free notation*
  - To evaluate expression, we make a single left-to-right scan of it (no precedence hierarchy)
  - Use *stack*

Infix	Postfix
2+3*4	2 3 4*+
*a*b +5	*ab*5+
(1+2)*7	1 2+7*
*a*b/c	*ab*c/
((*a*/(*b* −*c* +*d*))*(*e* −*a*)*c*)	*abc* −*d* +/*ea* −*c**
a/*b* −*c* +*d*e* −*a*c*	*ab*/*c* −*de** +*ac** −

Figure 3.13: Infix and postfix notation

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
−	6/2−3			0
4	6/2−3	4		1
2	6/2−3	4	2	2
*	6/2−3	4*2		1
+	6/2−3+4*2			0

Figure 3.14: Postfix evaluation

postfix expression
6 2/3-4 2*+

Stack 의 크기?
Push 횟수
Pop 횟수

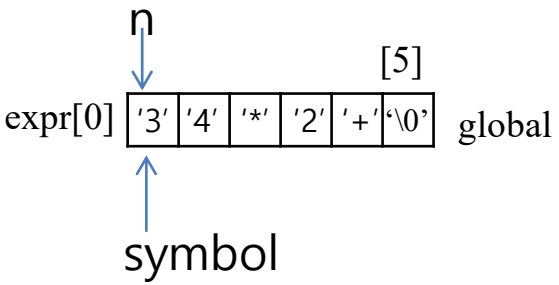
- Representation of stack and expression
  - Assumption that expression contains only
    - Binary operators : +, -, *, /, %
    - Operands : single digit integers

```
#define MAX_STACK_SIZE 100  /* maximum stack size */
#define MAX_EXPR_SIZE 100    /* max size of expression */

typedef enum { lparen, rparen, plus, minus, times, divide,
                                    mod, eos, operand } precedence;

int stack[MAX_STACK_SIZE];   /* global stack */
char expr[MAX_EXPR_SIZE];    /* global input string
                                    (a postfix expression)*/
```

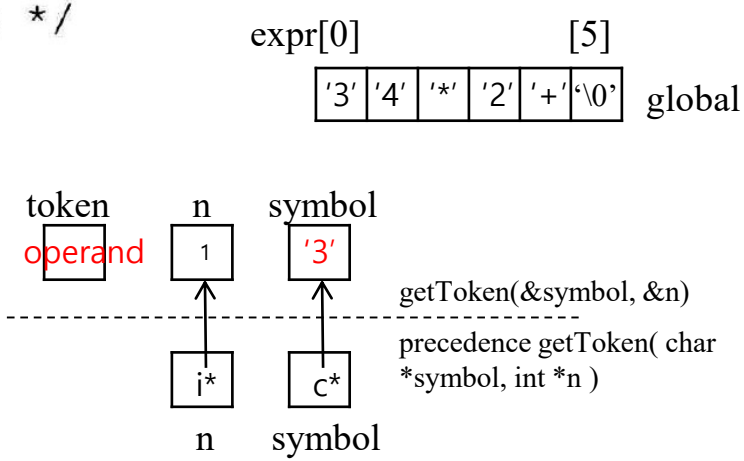
```
int eval(void)
{/* evaluate a postfix expression, expr, maintained as a
    global variable. '\0' is the the end of the expression.
    The stack and top of the stack are global variables.
    getToken is used to return the token type and
    the character symbol. Operands are assumed to be single
    character digits */
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0; /* counter for the expression string */
  top = -1;
  token = getToken(&symbol, &n);
  while (token != eos) {
    if (token == operand)
      push(symbol-'0'); /* stack insert */
    else {
      /* pop two operands, perform operation, and
         push result to the stack */
      op2 = pop(); /* stack delete */
      op1 = pop();
      switch(token) {
        case plus: push(op1+op2);
                   break;
        case minus: push(op1-op2);
                    break;
        case times: push(op1*op2);
                    break;
        case divide: push(op1/op2);
                     break;
        case mod: push(op1%op2);
      }
    }
    token = getToken(&symbol, &n);
  }
  return pop(); /* return result */
}
```

Program 3.13: Function to evaluate a postfix expression

※ symbol-'0' makes a single digit integer ('0': ASCII value of 48).



```
precedence getToken(char *symbol, int *n)
{/* get the next token, symbol is the character
    representation, which is returned, the token is
    represented by its enumerated value, which
    is returned in the function name */
   *symbol = expr[(*n)++];
   switch (*symbol) {
      case '(' : return lparen;
      case ')' : return rparen;
      case '+' : return plus;
      case '-' : return minus;
      case '/' : return divide;
      case '*' : return times;
      case '%' : return mod;
      case '\0' : return eos;
      default  : return operand; /* no error checking,
                                    default is operand */
   }
}
```



Program 3.14: Function to get a token from the input string

3.6.3 Infix to Postfix

- An algorithm by hand

(1) Fully parenthesize the expression.

(2) Move all binary operators so that they replace their corresponding right parentheses.

(3) Delete all parentheses.

Infix : a/b-c+d*e-a*c

(1) ((((a/b)-c) + (d*e)) - (a*c))

(2) ((((a b / c- (de* + (ac*-

(3) a b / c- de* + ac*-

※ It is inefficient on a computer
because it requires 2 passes: (1)and(2)

- **Example:** Simple expression
  - Input : a+b*c
  - Output : abc*+

Token generation by ***scanning*** left to right.
- ***Operands*** are passed to the output expression.
- ***Operators*** are stacked and unstacked *by their precedence*.

icp[token]

isp[stack[top]]

Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+	* push		1	*ab*
c	+	*		1	*abc*
eos				−1	*abc**+

Figure 3.15: Translation of *a* +*b***c* to postfix

- **Example:** Parenthesized expression
  - Input : a*(b+c)*d
  - Output : abc+*d*

Token	Stack [0]	Stack [1]	Stack [2]	Top	Output
a				−1	*a*
*	*			0	*a*
(	*	(		1	*a*
b	*	(		1	*ab*
+	*	(	+	2	*ab*
c	*	(	+	2	*abc*
)	*			0	*abc +*
*	*			0	*abc +**
d	*			0	*abc +*d*
eos				-1	*abc +*d**

push

push

pop→print

push

Figure 3.16: Translation of *a*(b +c)*d* to postfix

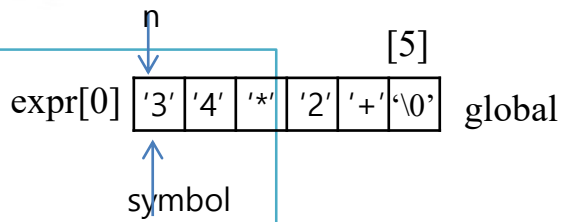
- ***isp***(in-stack precedence) and ***icp***(incoming precedence)

```
precedence stack[MAX_STACK_SIZE];

/* isp and icp arrays – index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */
static int isp[] =  { 0, 19, 12, 12, 13, 13, 13, 0 };
static int icp[] =  { 20, 19, 12, 12, 13, 13, 13, 0 };
```

```
typedef enum {
lparen, rparen, plus, minus, times, divide,
mod, eos, operand
} precedence;
```

Q. *isp* [*plus*] ?

```
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;
   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();
      }
      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

- Analysis of ***postfix***
  - n : number of tokens in the expression
  - extracting tokens and outputting them : $\Theta(n)$
  - in two while loop, the number of tokens that get *stacked* and *unstacked* is linear in n : $\Theta(n)$
  - So, the total complexity : $\boldsymbol{\Theta(n)}$

3.7 MULTIPLE STACKS AND QUEUES

- Representing more than two stacks within the same array
  - divide the memory into equal segments
- For stack *i*
  - *i* refers to the stack number of one of the *n* stacks.
  - *boundary [i],* 0 < =~ *i* <*MAX-STACKS,* points to the position immediately to the left of the bottom element of stack *i,*
  - *top[i],* 0 <= ~ *i* < *MAX-STACKS* points to the top element
  - Stack *i* is empty iff *boundary[i] = top[i]*.

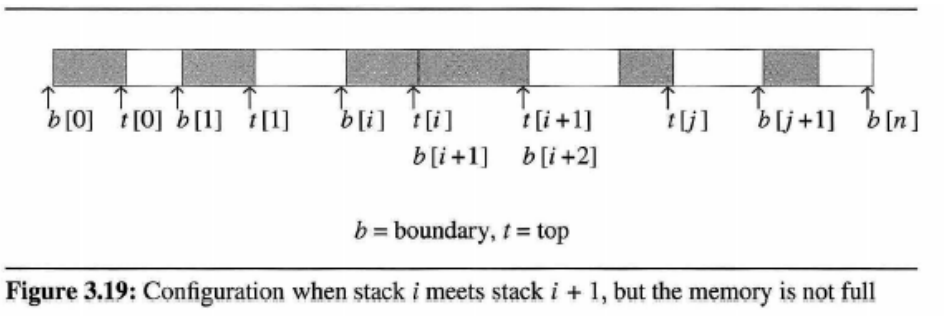


Figure 3.19: Configuration when stack *i* meets stack *i* + 1, but the memory is not full

- **multiple stacks declarations**

```
#define MEMORY_SIZE 100 /* size of memory */
#define MAX_STACKS 10 /* max number of stacks plus 1 */
/* global memory declaration */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n;  /* number of stacks entered by the user */
```

To divide the array into roughly equal segments

```
top[0] = boundary[0] = -1;
for (j = 1; j < n; j++)
   top[j] = boundary[j] = (MEMORY_SIZE/n)*j;
boundary[n] = MEMORY_SIZE-1;
```

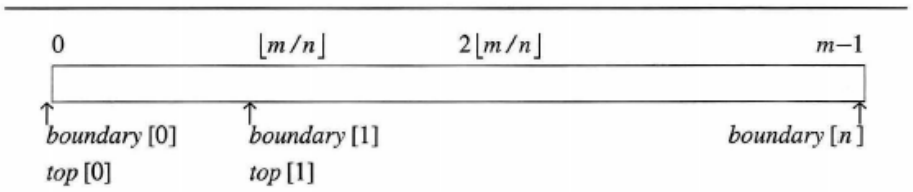


Figure 3.18: Initial configuration for n stacks in *memory* [m].

```
void push(int i, element item)
{/* add an item to the ith stack */
   if (top[i] == boundary[i+1])
      stackFull(i);
   memory[++top[i]] = item;
}
```

Program 3.16: Add an item to the *i*th stack

```
element pop(int i)
{/* remove top element from the ith stack */
   if (top[i] == boundary[i])
      return stackEmpty(i);
   return memory[top[i]--];
}
```

Program 3.17: Delete an item from the *i*th stack

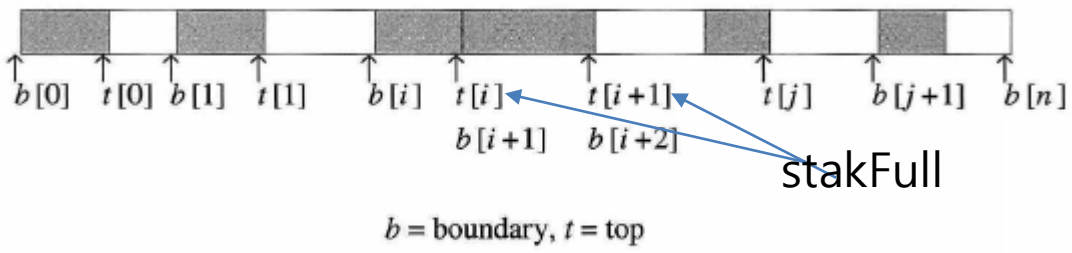


Figure 3.19: Configuration when stack *i* meets stack *i* + 1, but the memory is not full