STACKS AND QUEUES

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
"Fundamentals of Data Structures in C",
Computer Science Press, 1992.

Stack: a Last-In-First-Out (LIFO) list

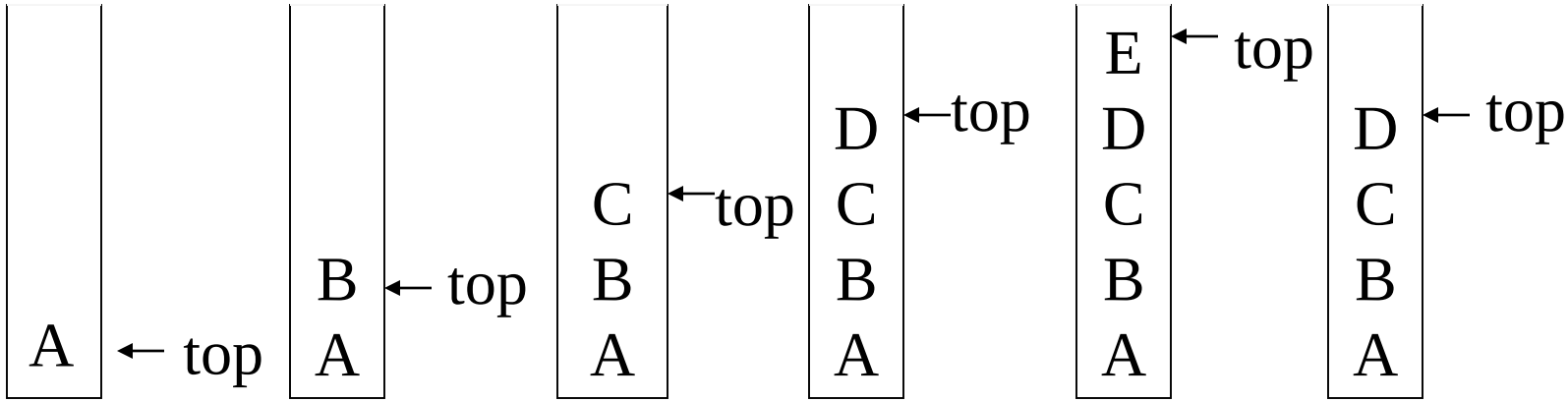


***Figure** Inserting and deleting elements in a stack

An application of stack: stack frame of function call (activation record)

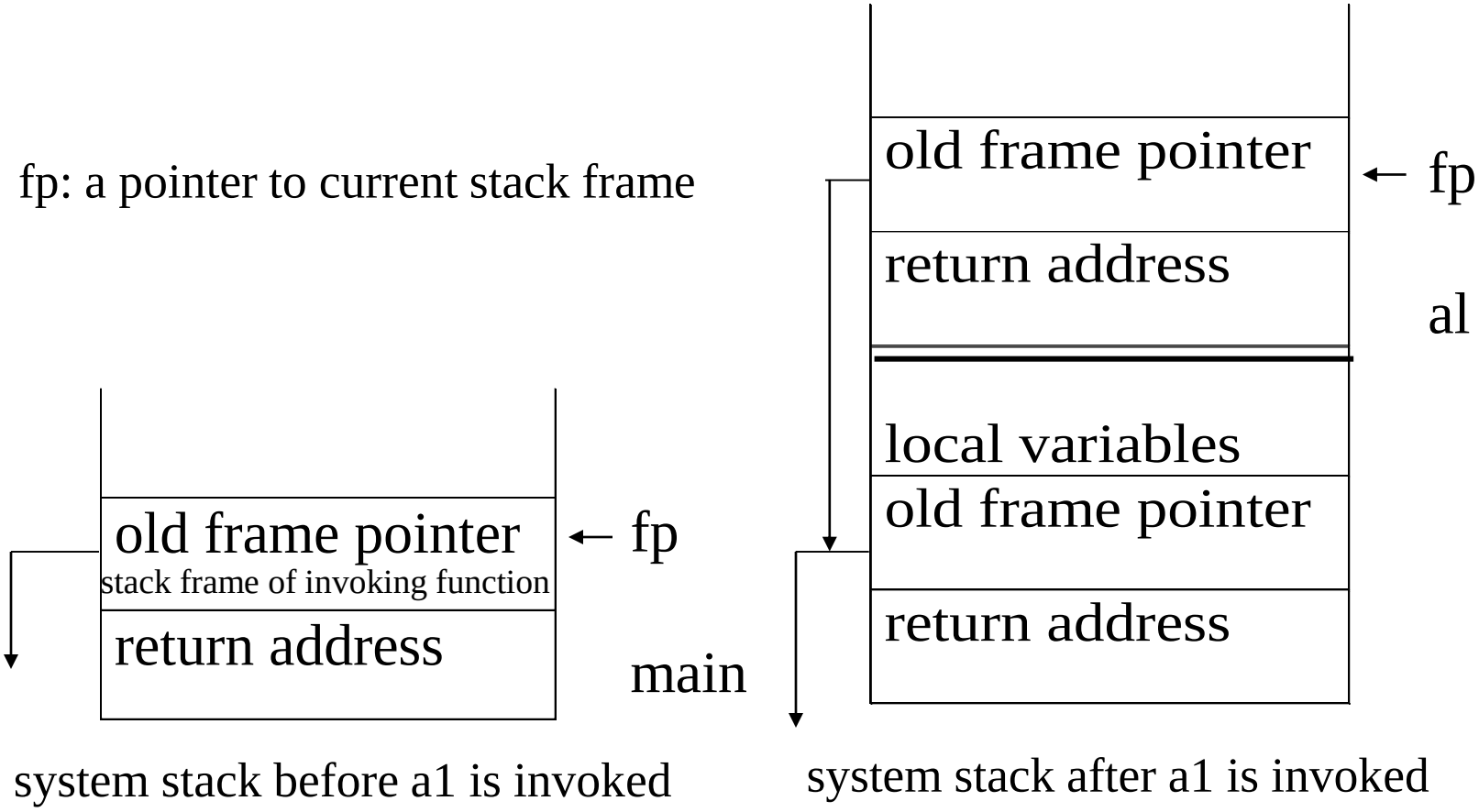


***Figure** System stack after function call

Abstract data type for stack

structure *Stack* is

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

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer

Stack CreateS(*max_stack_size*) ::=

create an empty stack whose maximum size is *max_stack_size*

Boolean IsFull(*stack*, *max_stack_size*) ::=

if (number of elements in *stack* == *max_stack_size*)

return TRUE

else return FALSE

Stack Add(*stack*, *item*) ::=

if (IsFull(*stack*)) *stack_full*

else insert *item* into top of *stack* and **return**

Boolean IsEmpty(*stack*) ::=
if(*stack* == CreateS(*max_stack_size*))
return TRUE
else return FALSE

Element Delete(*stack*) ::=
if(IsEmpty(*stack*)) **return**
else remove and return the *item* on the top of the stack.

***Structure :** Abstract data type *Stack*

Implementation: using array

Stack **CreateS(max_stack_size)** ::=

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
        int key;
        /* other fields */
        } element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

Boolean **IsEmpty(Stack)** ::= top< 0;

Boolean **IsFull(Stack)** ::= top >= MAX_STACK_SIZE-1;

Add to a stack

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

***program :** Add to a stack

Delete from a stack

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

***Program :** Delete from a stack

Stack Full with array doubling

```
Void stackFull()
{
        REALLOC(stack,2*capacity*sizeof(*stack));
        capacity*=2;
}
```

Stacks using Dynamic Arrays

```
Stack creates()::= typedef struct
                   {
                            int key;
                   }element;
                   element *stack;

          MALLOC(stack,sizeof(*stack));
                   int capacity=1;
                   int top=-1;
Boolean Isempty(Stack)::= top<0;
Boolean IsFull(Stack)::top>=capacity-1;
```

Queue: a First-In-First-Out (FIFO) list

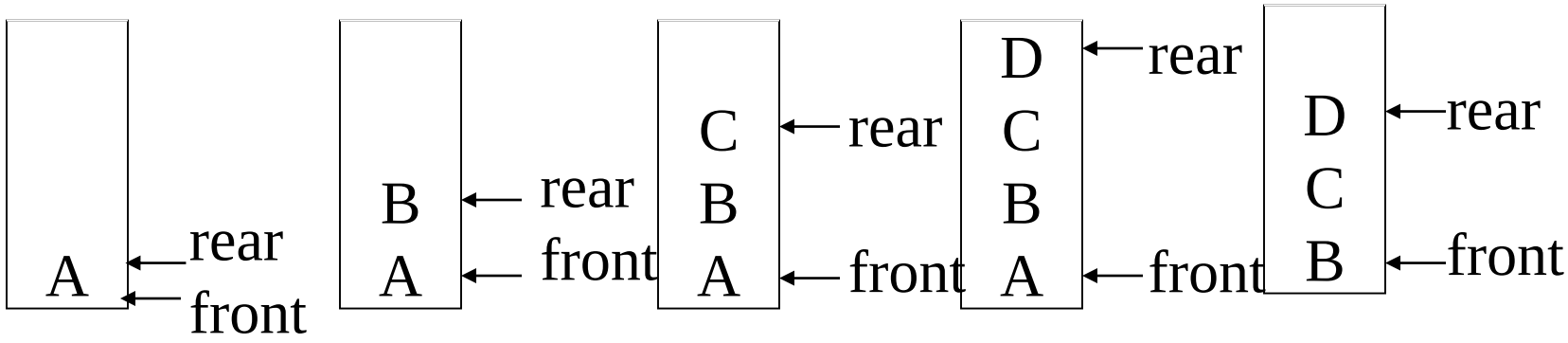


***Figure :** Inserting and deleting elements in a queue

Application: Job scheduling

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

***Figure :** Insertion and deletion from a sequential queue

Abstract data type of queue

structure *Queue* is

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

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*,

max_ queue_ size ∈ positive integer

Queue CreateQ(*max_queue_size*) ::=

create an empty queue whose maximum size is *max_queue_size*

Boolean IsFullQ(*queue*, *max_queue_size*) ::=

if(number of elements in *queue* == *max_queue_size*)

return *TRUE*

else return *FALSE*

Queue AddQ(*queue*, *item*) ::=

if (IsFullQ(*queue*)) *queue_full*

else insert *item* at rear of *queue* and return *queue*

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

***Structure :** Abstract data type *Queue*

Implementation 1: using array

Queue CreateQ(*max_queue_size*) ::=

```
# define MAX_QUEUE_SIZE 100/* Maximum queue size */
typedef struct {
            int key;
            /* other fields */
            } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

Boolean IsEmpty(queue) ::= front == rear

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

Add to a queue

```
void addq(int *rear, element item)
{
/* add an item to the queue */
    if (*rear == MAX_QUEUE_SIZE_1) {
       queue_full( );
       return;
    }
   queue [++*rear] = item;
}
```

***Program :** Add to a queue

Delete from a queue

```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
   if ( *front == rear)
      return queue_empty( );     /* return an error key */
   return queue [++ *front];
}
```

***Program :** Delete from a queue

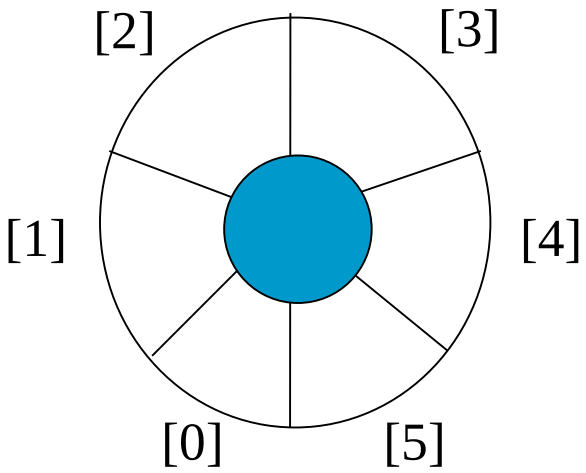
problem: there may be available space when IsFullQ is true
I.E. movement is required.

Implementation 2: regard an array as a circular queue

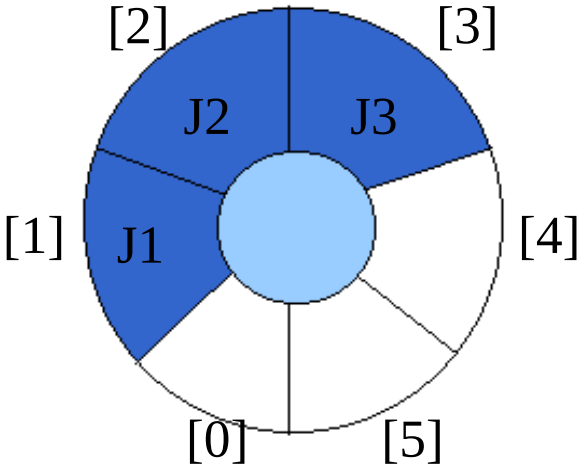
front: one position counterclockwise from the first element

rear: current end

EMPTY QUEUE



front = 0
rear = 0

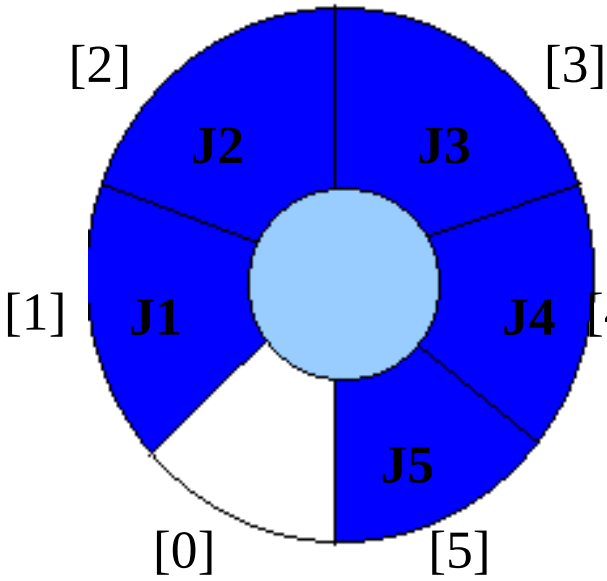


front = 0
rear = 3

***Figure :** Empty and nonempty circular queues

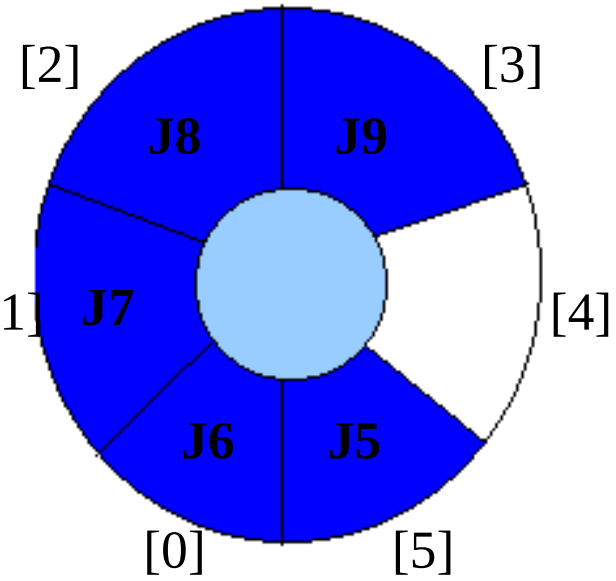
Problem: one space is left when queue is full

FULL QUEUE



front =0
rear = 5

FULL QUEUE



front =4
rear =3

***Figure :** Full circular queues and then we remove the item

Add to a circular queue

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

***Program :** Add to a circular queue

Delete from a circular queue

```
element deleteq(int* front, int rear)
{
   element item;
   /* remove front element from the queue and put it in item */
      if (*front == rear)
         return queue_empty( );
                  /* queue_empty returns an error key */
      *front = (*front+1) % MAX_QUEUE_SIZE;
      return queue[*front];
}
```

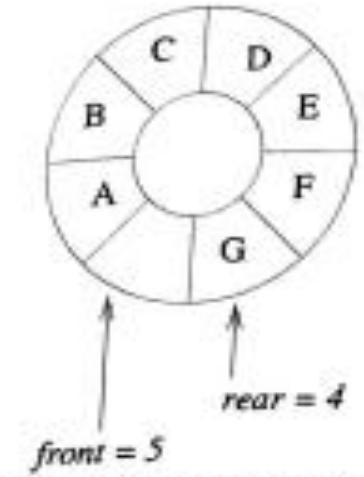
***Program :** Delete from a circular queue

Dynamic circular queue

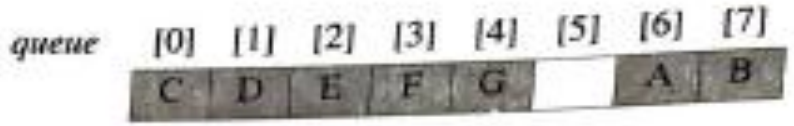
(1) Create a new array *newQueue* of twice the capacity.

(2) Copy the second segment (i.e., the elements *queue* [*front* +1] through *queue* [*capacity* −1]) to positions in *newQueue* beginning at 0.

(3) Copy the first segment (i.e., the elements *queue* [0] through *queue* [*rear*]) to positions in *newQueue* beginning at *capacity* −*front* −1.

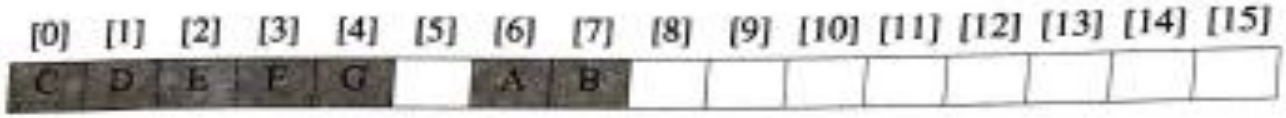


(a) A full circular queue



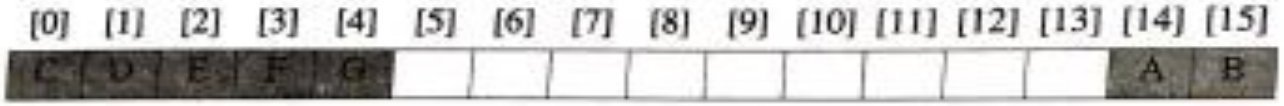
front = 5, *rear* = 4

(b) Flattened view of circular full queue



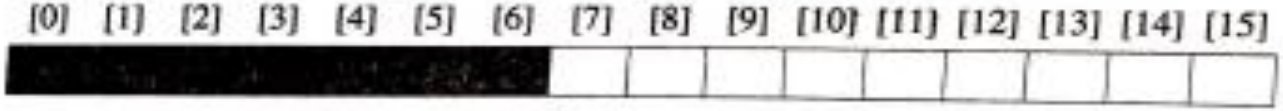
front = 5, *rear* = 4

(c) After array doubling



front = 13, *rear* = 4

(d) After shifting right segment



front = 15, *rear* = 6

(e) Alternative configuration

```
void addq(element item)
{/* add an item to the queue */
   rear = (rear+1) % capacity;
   if (front == rear)
      queueFull(); /* double capacity */
   queue[rear] = item;
}
```

Program 3.9: Add to a circular queue

```
void queueFull()
{
   /* allocate an array with twice the capacity */
   element* newQueue;
   MALLOC(newQueue, 2 * capacity * sizeof(*queue));

   /* copy from queue to newQueue */
   int start = (front+1)  % capacity;
   if (start  <  2)
      /* no wrap around */
      copy(queue+start, queue+start+capacity-1, newQueue);
   else
   {/* queue wraps around */
      copy(queue+start, queue+capacity, newQueue);
      copy(queue, queue+rear+1, newQueue+capacity-start);
   }

   /* switch to newQueue */
   front = 2 * capacity - 1;
   rear = capacity - 2;
   capacity *= 2;
   free(queue);
   queue = newQueue;
}
```

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:
((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:
(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

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
* / %	mutiplicative	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 (p.119)

user **compiler**

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+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 (p.120)

Postfix: no parentheses, no precedence

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 of expn : 6 2 / 3 – 4 2 * +

Goal: infix --> postfix

Assumptions:

operators: +, -, *, /, %

operands: single digit integer

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
#define MAX_EXPR_SIZE 100 /* max size of expression */
typedef enum{1paran, rparen, plus, minus, times, divide,
                    mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string */
```

```
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.
   get_token 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 */
  int top = -1;
  token = get_token(&symbol, &n);
  while (token != eos)  {
     if (token == operand)          exp: character array
          add(&top, symbol-'0');   /* stack insert */
```

```
    else {
        /* remove two operands, perform operation, and
           return result to the stack */
      op2 = delete(&top);  /* stack delete */
      op1 = delete(&top);
      switch(token) {
          case plus: add(&top, op1+op2); break;
          case minus: add(&top, op1-op2); break;
          case times: add(&top, op1*op2); break;
          case divide: add(&top, op1/op2); break;
          case mod: add(&top, op1%op2);
      }
    }
    token = get_token (&symbol, &n);
  }
  return delete(&top); /* return result */
}
```

***Program 3.9:** Function to evaluate a postfix expression (p.122)

```
precedence get_token(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 3.10:** Function to get a token from the input string (p.123)

Infix to Postfix Conversion
(Intuitive Algorithm)

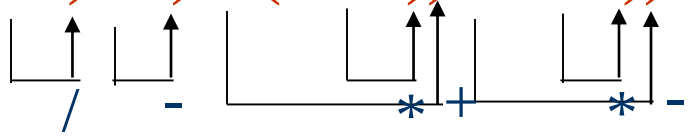
(1) Fully parenthesize expression

a / b - c + d * e - a * c -->

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

(2) All operators replace their corresponding right parentheses.

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



(3) Delete all parentheses.

ab/c-de*+ac*-

two passes

The orders of operands in infix and postfix are the same.

a + b * c, * > +

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

***Figure 3.15:** Translation of a+b*c to postfix (p.124)

a $*_1$ (b +c) $*_2$ d

Token	Stack [0]	[1]	[2]	Top	Output
a				-1	a
$*_1$	$*_1$			0	a
(	$*_1$	(		1	a
b	$*_1$	(		1	ab
+	$*_1$	(	+	2	ab
c	$*_1$	(	+	2	abc
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	abc+$*_1$
d	$*_2$			0	abc+$*_1$d
eos	$*_2$			0	abc+$*_1$d$*_2$

*** Figure 3.16:** Translation of a*(b+c)*d to postfix (p.124)

Rules

(1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.

(2) (has low in-stack precedence, and high incoming precedence.

	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

```
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};
```

isp: in-stack precedence

icp: incoming precedence

```
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 = get _token(&symbol, &n); token != eos;
                    token = get_token(&symbol, &n)) {
     if (token == operand)
        printf (“%c”, symbol);
     else if (token == rparen ){
```

```
    /*unstack tokens until left parenthesis */
    while (stack[top] != lparen)
        print_token(delete(&top));
    delete(&top); /*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] )
        print_token(delete(&top));
    add(&top, token);
   }
 }
 while ((token = delete(&top)) != eos)
     print_token(token);
 print("\n");
}
```

$\theta(n)$

$f(n)=\theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \le f(n) \le c_2 g(n)$ for all n, $n \ge n_0$.

$f(n)=\theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

***Program 3.11**: Function to convert from infix to postfix (p.126)

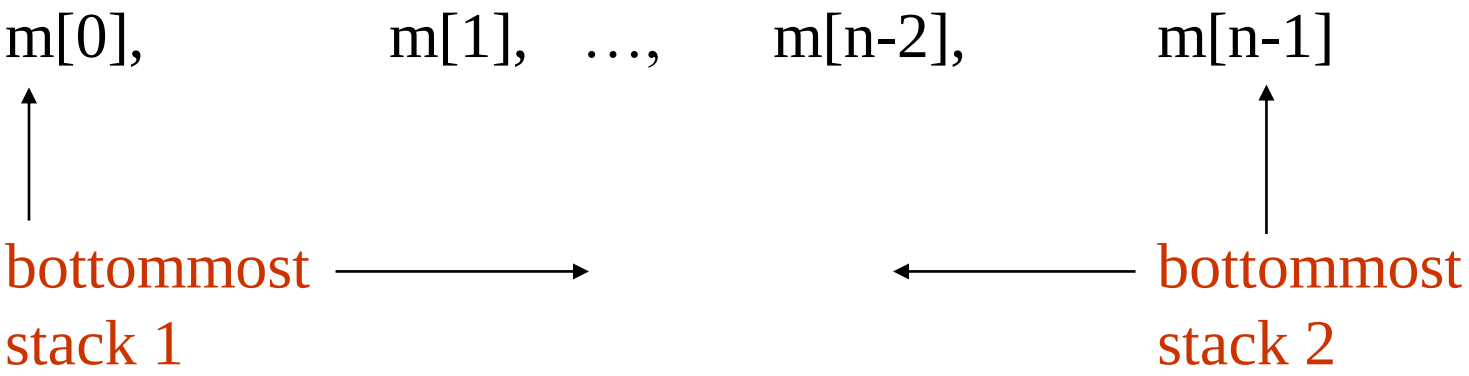
Infix	Prefix
a*b/c	/*abc
a/b-c+d*e-a*c	-+-/abc*de*ac
a*(b+c)/d-g	-/*a+bcdg

(1) evaluation
(2) transformation

***Figure 3.17:** Infix and postfix expressions (p.127)

Multiple stacks and queues

Two stacks



More than two stacks (n)

memory is divided into n equal segments

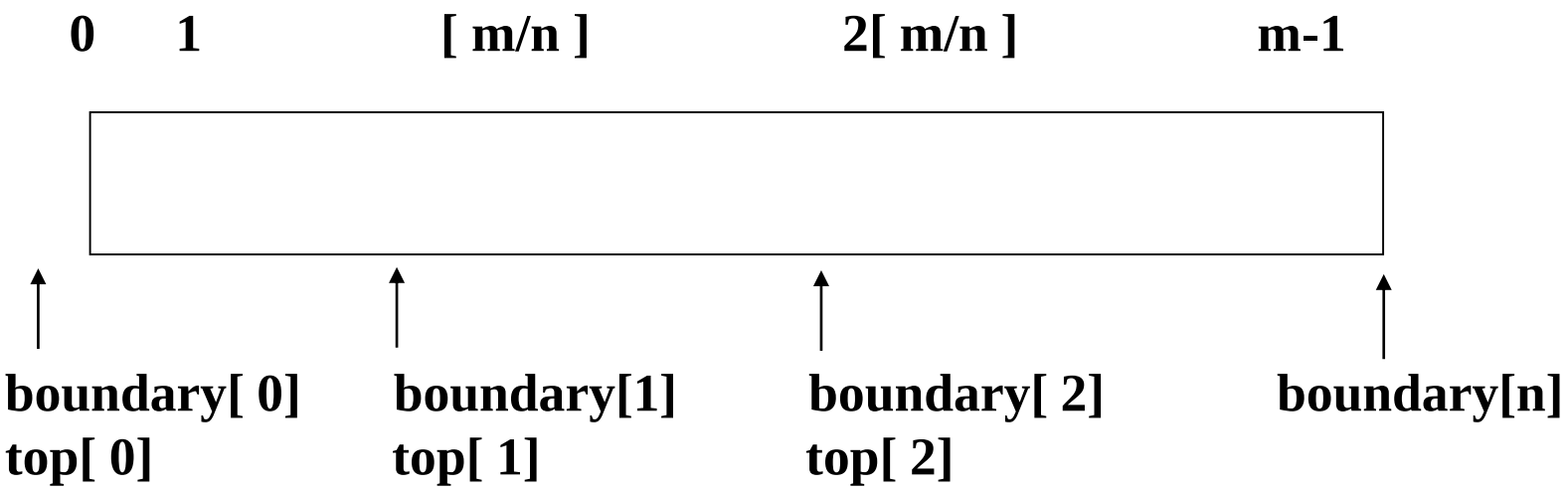
boundary[stack_no]

$0 \le stack\_no < MAX\_STACKS$

top[stack_no]

$0 \le stack\_no < MAX\_STACKS$

Initially, boundary[i]=top[i].



All stacks are empty and divided into roughly equal segments.

***Figure 3.18:** Initial configuration for *n* stacks in memory [m]. (p.129)

```
#define MEMORY_SIZE 100    /* size of memory */
#define MAX_STACK_SIZE 100
                /* 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 */
```

*(p.128)

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

*(p.129)

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

*__Program 3.12:__ Add an item to the stack *stack-no* (p.129)

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

__*Program 3.13:__ Delete an *item* from the stack *stack-no* (p.130)

Find j, stack_no < j < n

such that top[j] < boundary[j+1]

or, 0 ≤ j < stack_no

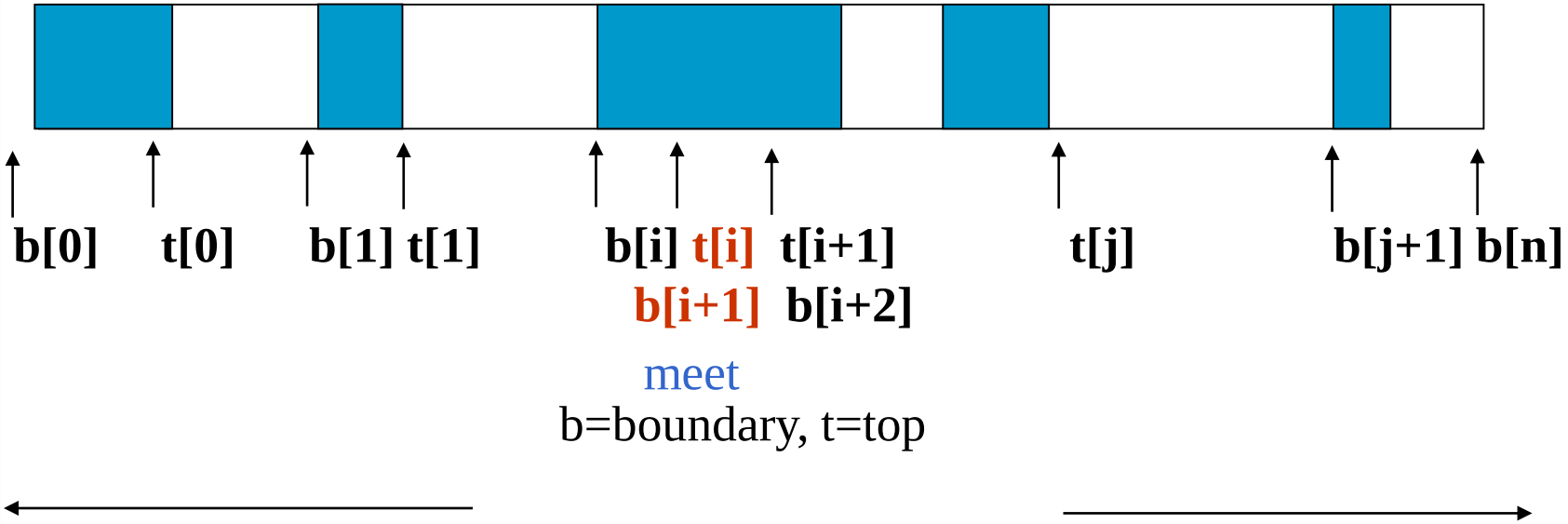


***Figure 3.19:** Configuration when stack i meets stack i+1, but the memory is not full (p.130)*