



CHAPTER 3

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”,

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

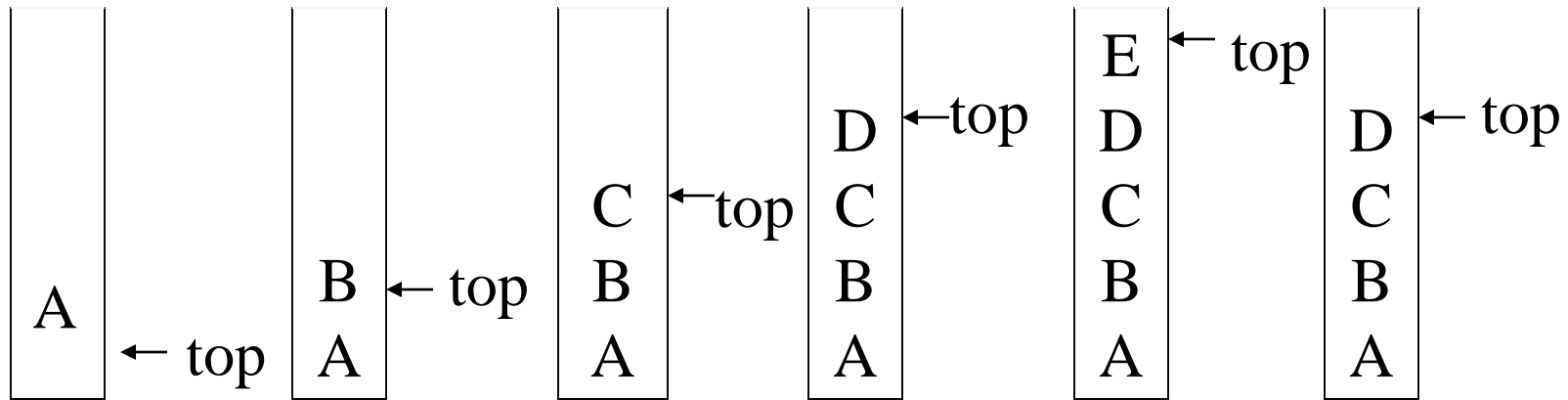
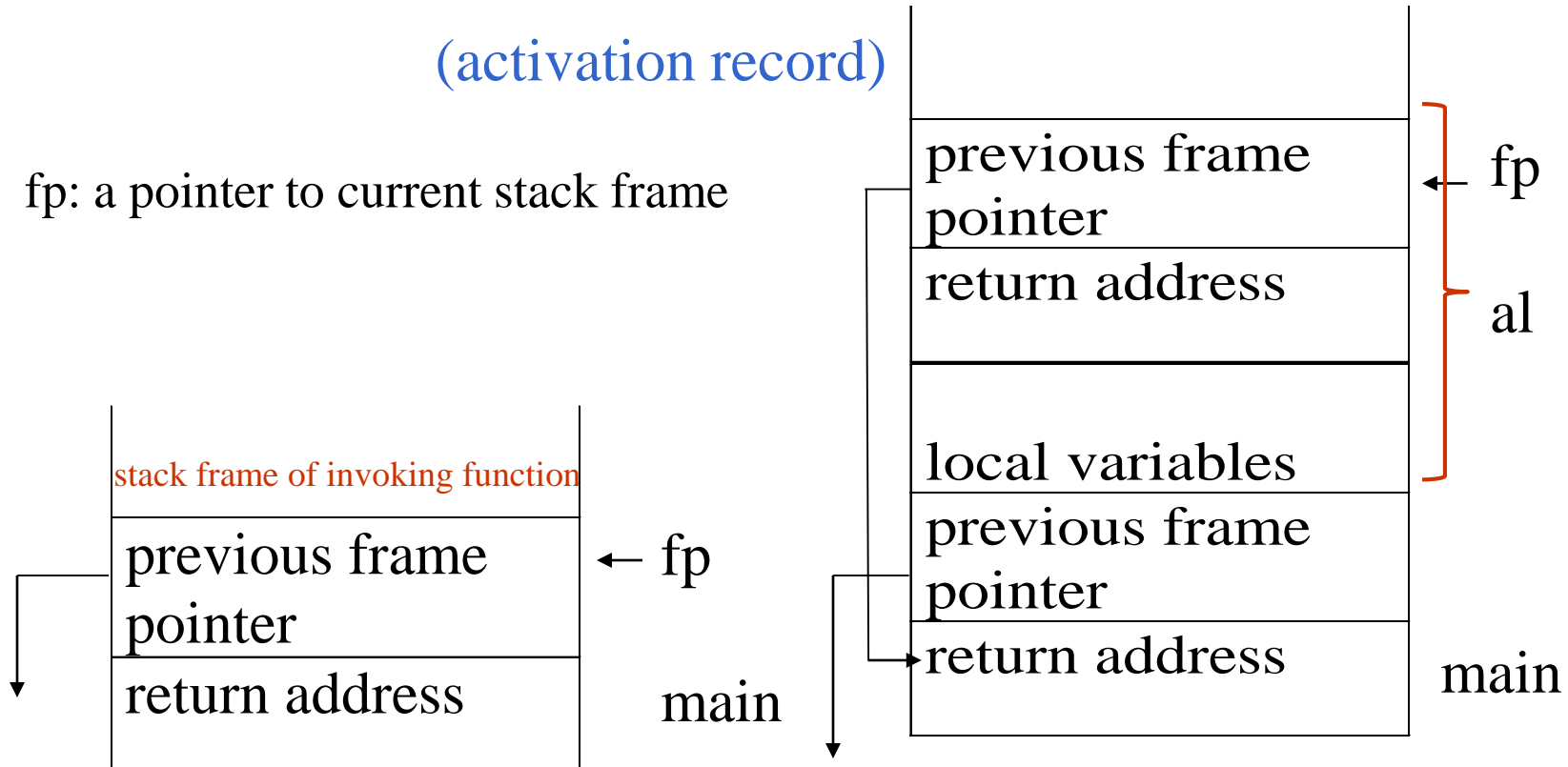


Figure 3.1: Inserting and deleting elements in a stack

An application of stack: stack frame of function call



system stack before a1 is invoked

system stack after a1 is invoked

(a)

(b)

Figure 3.2: System stack after function call a1

abstract data type for stack

structure *Stack* is

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

functions:

for all $stack \in Stack$, $item \in element$, $max_stack_size \in$ 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 $Push(stack, item) ::=$

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 Pop(*stack*) ::=

if(IsEmpty(*stack*)) **return**

else remove and return the *item* on the top
of the stack.

Structure 3.1: 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 3.1: 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 3.2: Delete from a stack

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

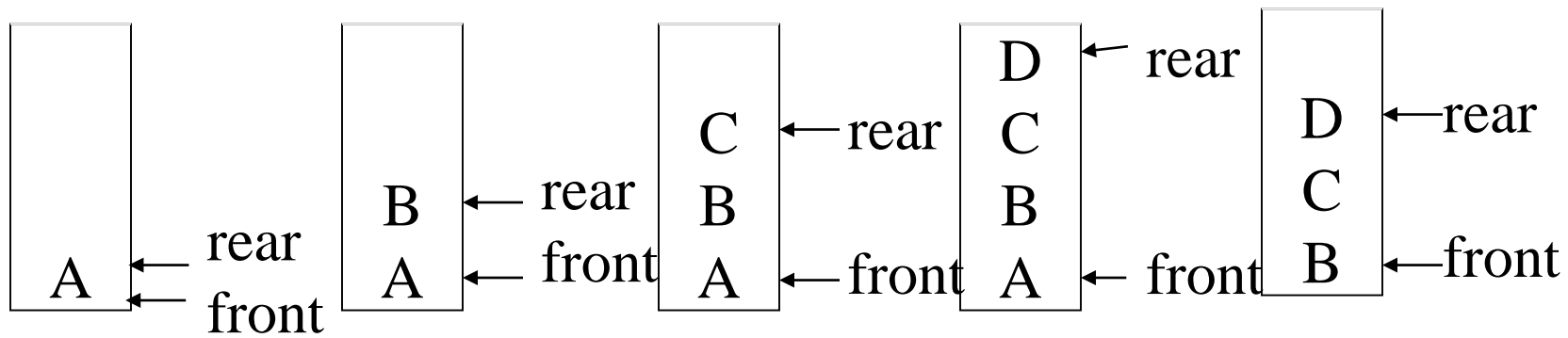


Figure 3.4: Inserting and deleting elements in a queue

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 3.5: 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 $queue \in Queue$, $item \in element$,

$max_queue_size \in$ 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 $queue == max_queue_size$)

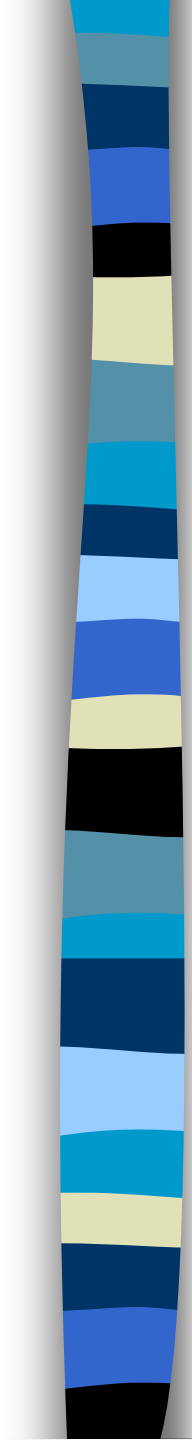
return *TRUE*

else return *FALSE*

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

if (IsFullQ($queue$)) $queue_full$

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



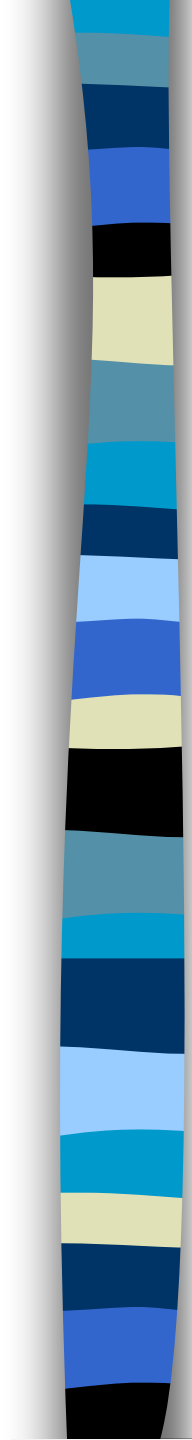
Boolean IsEmptyQ(*queue*) ::=
 if (*queue* == CreateQ(*max_queue_size*))
 return *TRUE*
 else return *FALSE*

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

Structure 3.2: 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 3.5: 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 3.6: Delete from a queue

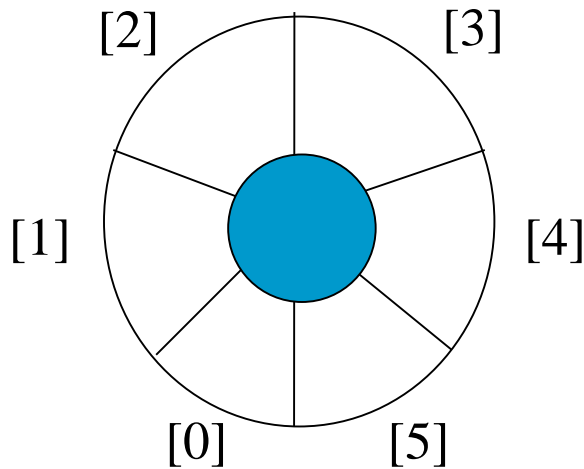
problem: there may be available space when IsFullQ is true i.e., movement is required.

Implementation 2: regard an array as a circular queue

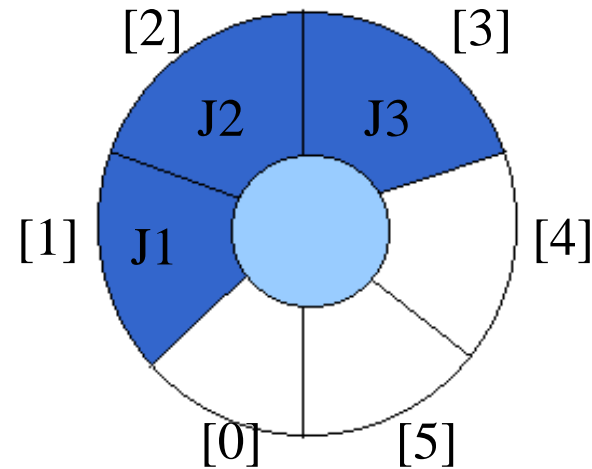
front: one position counterclockwise from the first element

rear: current end

EMPTY QUEUE



front = 0
rear = 0



front = 0
rear = 3

Figure 3.6: Empty and nonempty circular queues

Problem: one space is left when queue is full

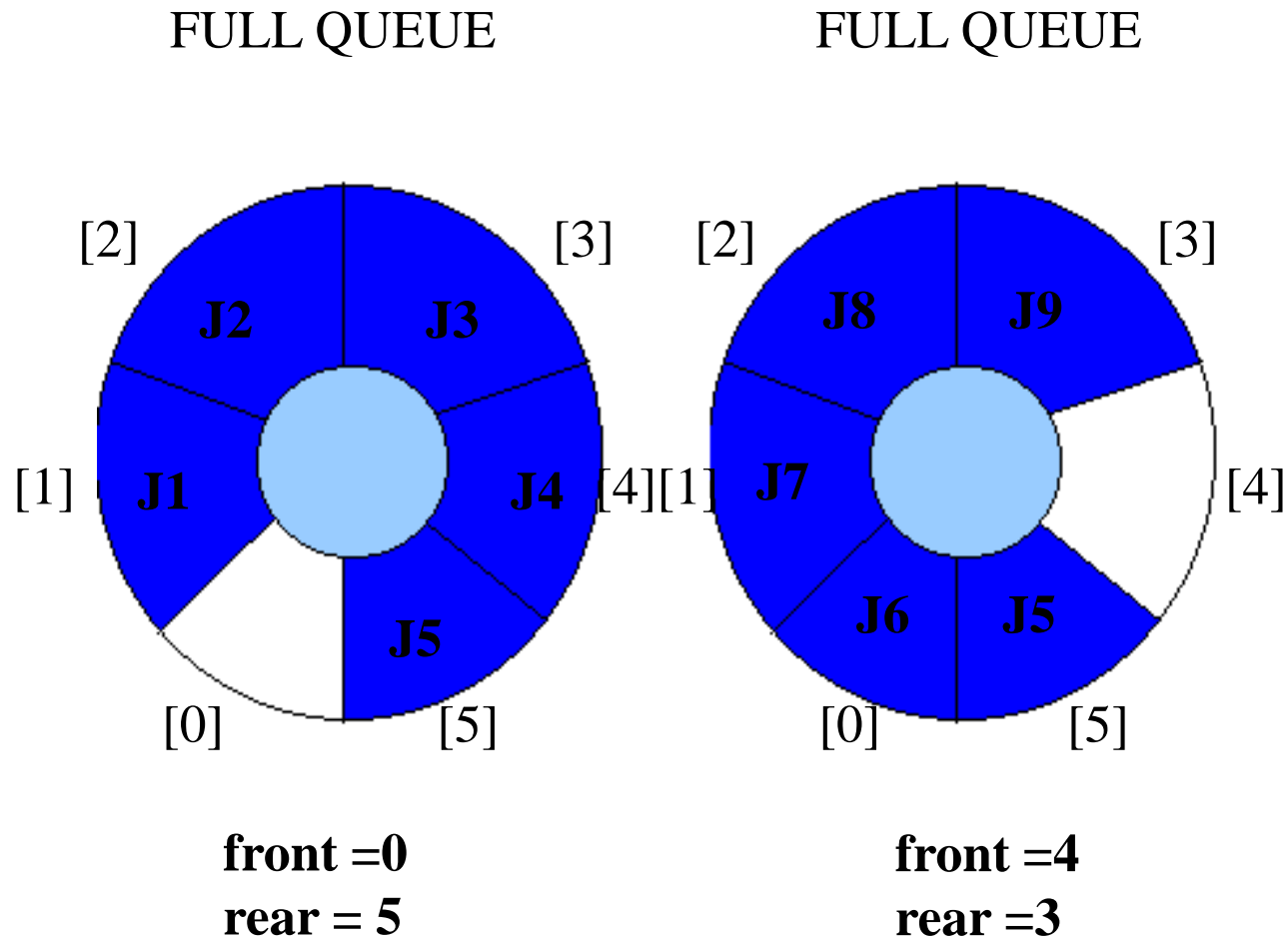


Figure 3.7: Full circular queues and then we remove the item



Add to a circular queue

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

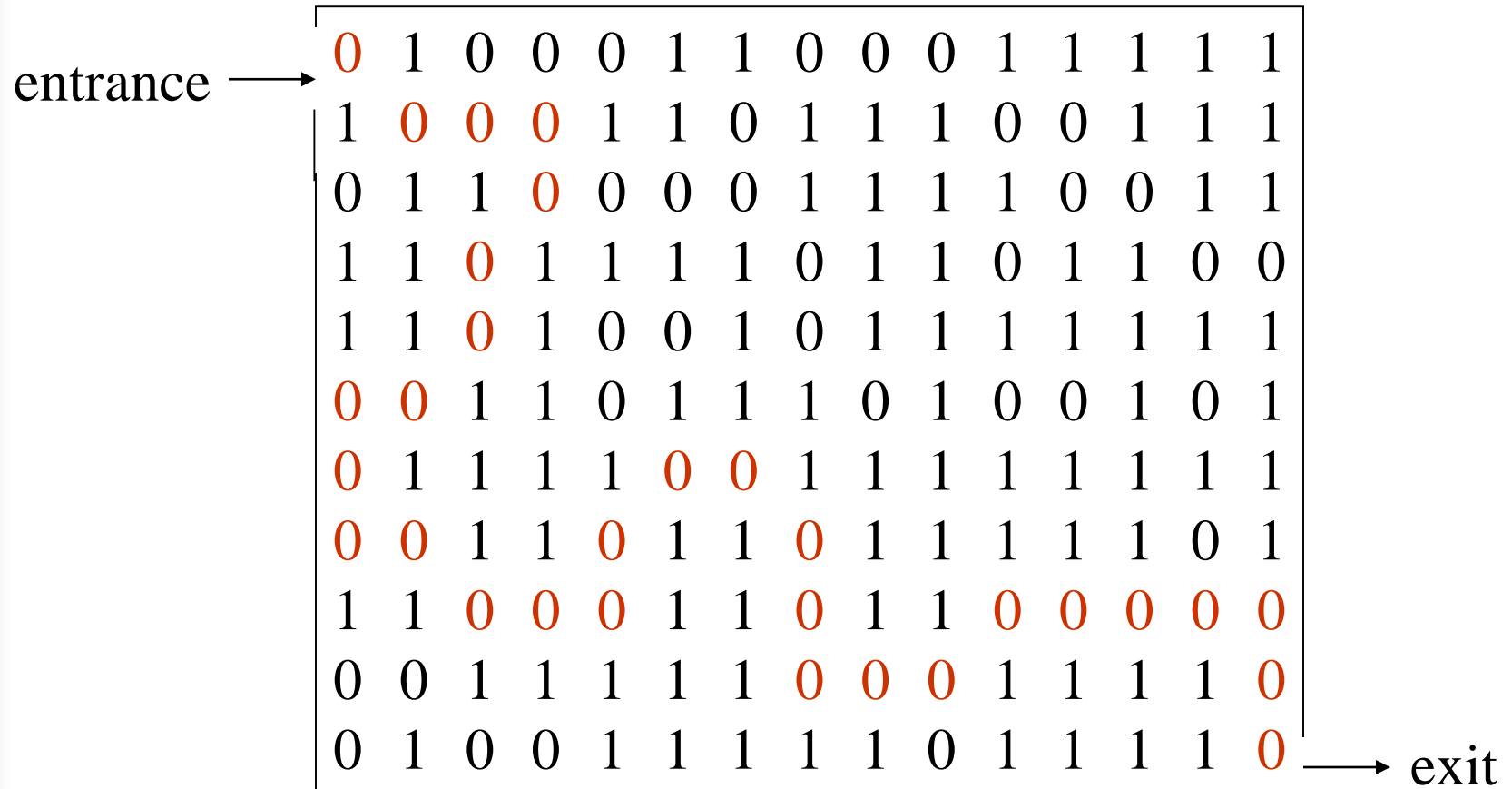
Program 3.7: Add to a circular queue

Delete from a circular queue

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

Program 3.8: Delete from a circular queue

A Mazing Problem



1: blocked path

0: through path

***Figure 3.8:** An example maze

a possible representation

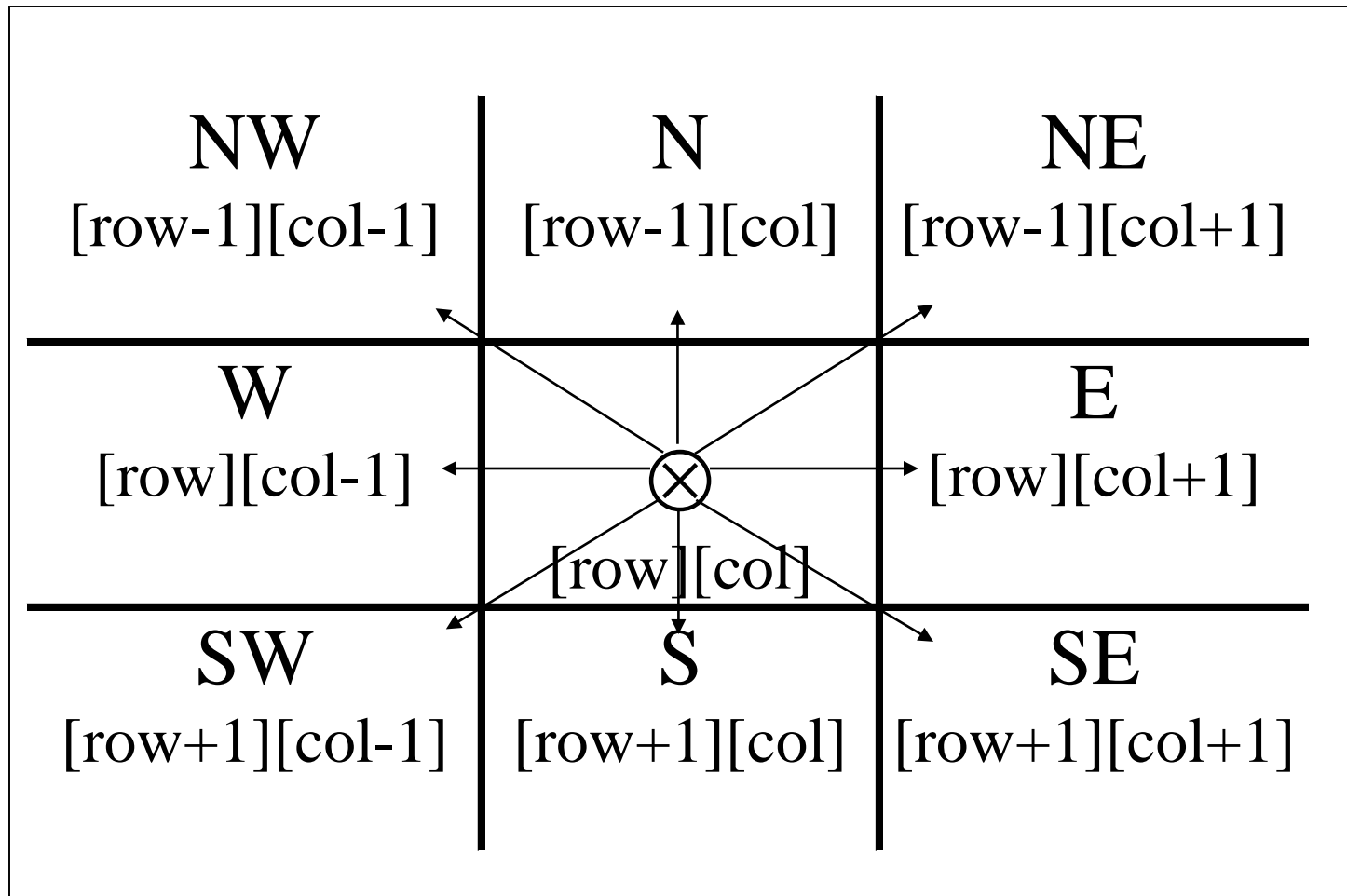


Figure 3.9: Allowable moves

a possible implementation

```
typedef struct {  
    short int vert; next_row = row + move[dir].vert;  
    short int horiz; next_col = col + move[dir].horiz;  
} offsets;  
offsets move[8]; /*array of moves for each direction*/
```

Name	Dir	move[dir].vert	move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

Use stack to keep pass history

```
#define MAX_STACK_SIZE 100
    /*maximum stack size*/
typedef struct {
    short int row;
    short int col;
    short int dir;
} element;
element stack[MAX_STACK_SIZE];
```



Initialize a stack to the maze's entrance coordinates and direction to **north**;

```
while (stack is not empty){
```

```
    /* move to position at top of stack */
```

```
    <row, col, dir> = delete from top of stack;
```

```
    while (there are more moves from current position) {
```

```
        <next_row, next_col > = coordinates of next move;
```

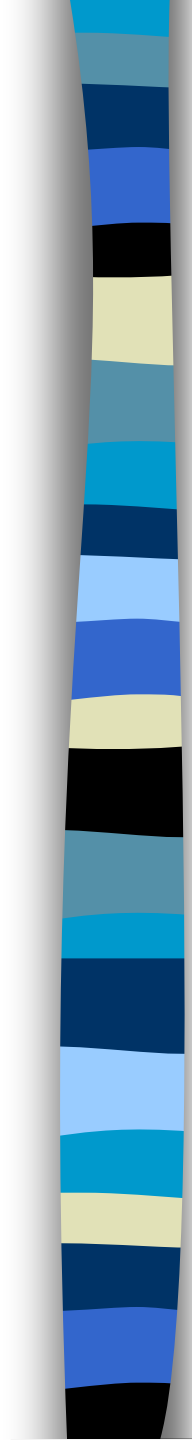
```
        dir = direction of move;
```

```
        if ((next_row == EXIT_ROW)&& (next_col == EXIT_COL))
```

```
            success; /* find out the destination */
```

```
        if (maze[next_row][next_col] == 0 &&
```

```
            mark[next_row][next_col] == 0) {
```

```
/* legal move and haven't been there */
mark[next_row][next_col] = 1;
/* save current position and direction */
add <row, col, dir> to the top of the stack;
row = next_row;
col = next_col;
dir = north;
}
}
}
printf("No path found\n");
```

***Program 3.11: Initial maze algorithm**

The size of a stack?

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{m \times p}$$

The worst case complexity of the algorithm is $O(mp)$

Figure 3.11: Simple maze with a long path

```
void path (void)
```

```
{
```

```
/* output a path through the maze if such a path exists */
```

```
int i, row, col, next_row, next_col, dir, found = FALSE;
```

```
element position;
```

```
mark[1][1] = 1; top = 0;
```

```
stack[0].row = 1; stack[0].col = 1; stack[0].dir = 1;
```

```
while (top > -1 && !found) {
```

```
    position = pop();
```

```
    row = position.row; col = position.col;
```

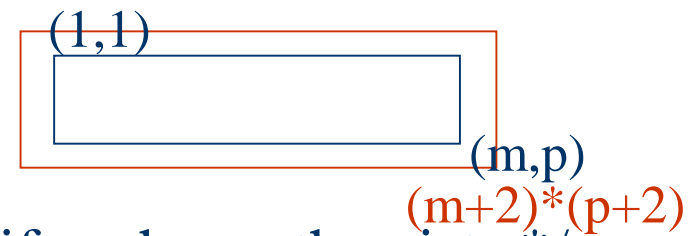
```
    dir = position.dir;
```

```
    while (dir < 8 && !found) {
```

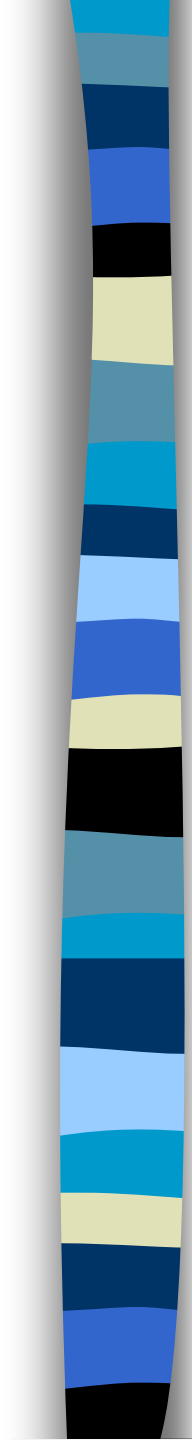
```
        /*move in direction dir */
```

```
        next_row = row + move[dir].vert;
```

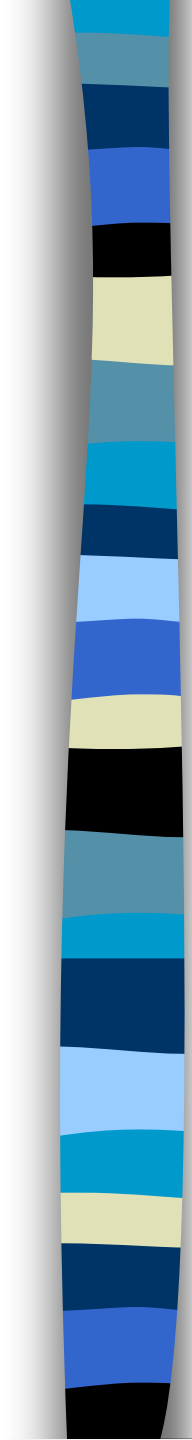
```
        next_col = col + move[dir].horiz;
```



	0	
7	N	1
6 W		E 2
5	S	3
	4	



```
if (next_row==EXIT_ROW && next_col==EXIT_COL)
    found = TRUE; //Exit
else if ( !maze[next_row][next_col] &&
        !mark[next_row][next_col] {
    mark[next_row][next_col] = 1;
    position.row = row;
    position.col = col;
    position.dir = ++dir;
    push(position);
    row = next_row;
    col = next_col;
    dir = 0;
    }
else ++dir;
}
```



```
if (found) {
    printf("The path is :\n");
    printf("row col\n");
    for (i = 0; i <= top; i++)
        printf(" %2d%5d", stack[i].row, stack[i].col);
    printf("%2d%5d\n", row, col);
    printf("%2d%5d\n", EXIT_ROW, EXIT_COL);
}
else printf("The maze does not have a path\n");
}
```

Program 3.12:Maze search function

Evaluation of Expressions

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

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

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

Interpretation 1:

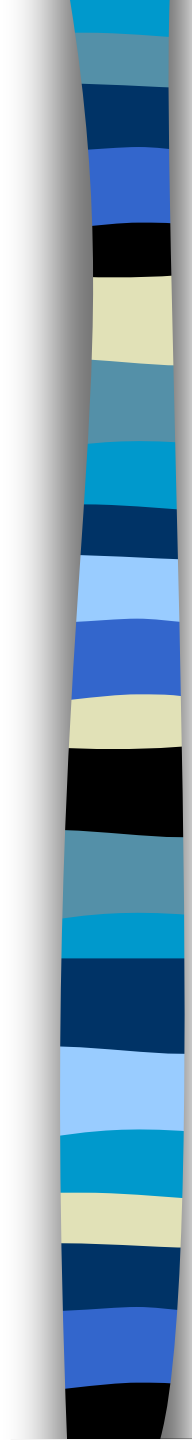
$$((4/2)-2)+(3*3)-(4*2)=0 + 9 - 8=1$$

Interpretation 2:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\cdots$$

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

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

Postfix: no parentheses, no precedence

Token	Stack			Top
	[0]	[1]	[2]	
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

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{ lparen, rparen, plus, minus, times, divide,
              mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string */
```

eos : end of 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) {
```

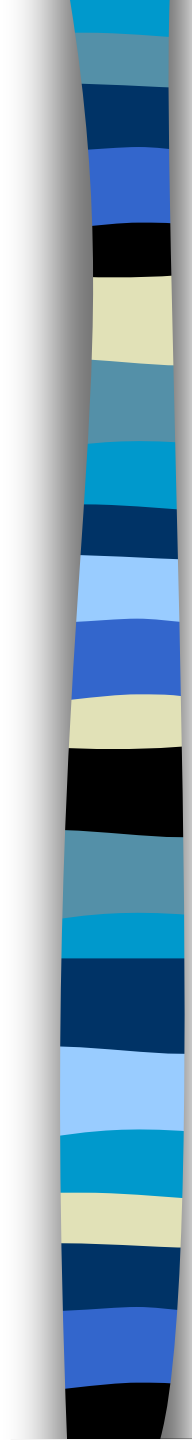
```
    if (token == operand)
```

```
        push(symbol-'0'); /* stack insert */
```

```
        /* ASCII 0*/
```

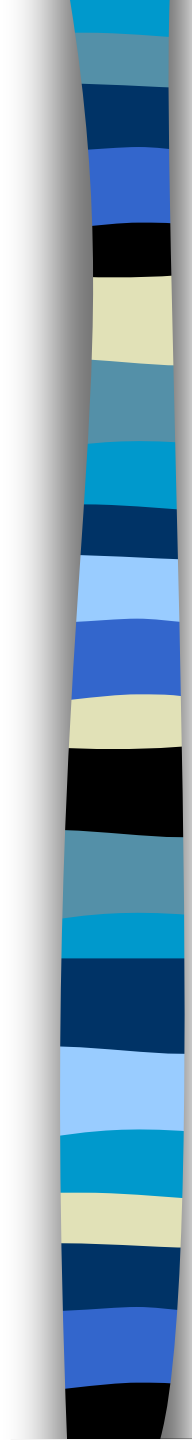
```
CHAPTER 3
```

exp: character array



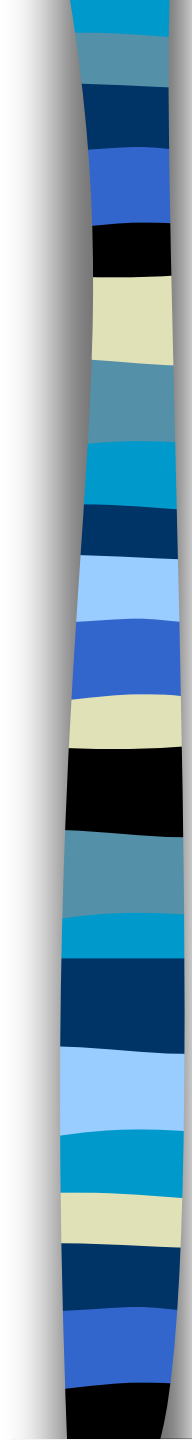
```
else {
    /* remove two operands, perform operation, and
       return result to the stack */
    op2 = pop(); /* stack delete */
    op1 = pop();
    switch(token) {
        case plus: add(&top, op1+op2); break;
        case minus: add(&top, op1-op2); break;
        case times: add(&top, op1*op2);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

Program 3.13: Function to evaluate a postfix expression



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

Infix to Postfix Conversion (Intuitive Algorithm)

- (1) Fully parenthesize expression

$$a / b - c + d * e - a * c \rightarrow$$

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

- (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			Top	Output
	[0]	[1]	[2]		
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

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

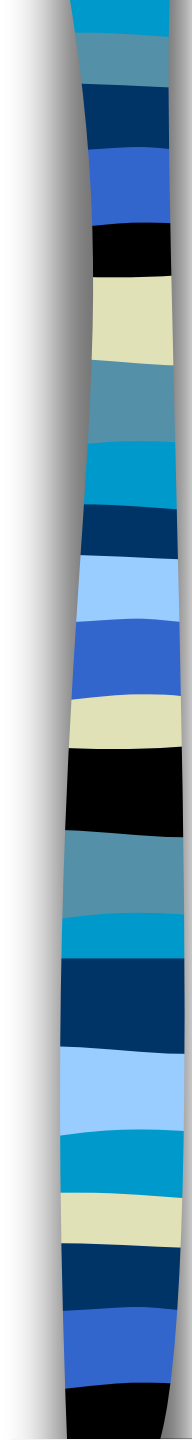
Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
* ₁	* ₁			0	a
(* ₁	(1	a
b	* ₁	(1	ab
+	* ₁	(+	2	ab
c	* ₁	(+	2	abc
)	* ₁	match)		0	abc+
* ₂	* ₂	* ₁ = * ₂		0	abc+* ₁
d	* ₂			0	abc+* ₁ d
eos	* ₂			0	abc+* ₁ d* ₂

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

Rules

- (1) Operators are taken out of the stack as long as their in-stack precedence (isp) is **higher than or equal to** the incoming precedence (icp) of the new operator.
- (2) “(” has **low** in-stack precedence, and **high** incoming precedence.

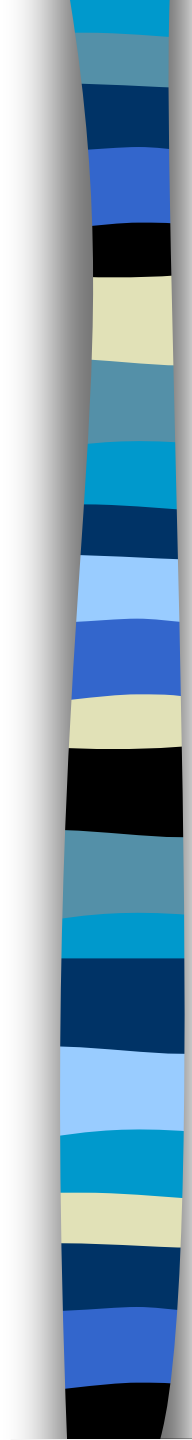
	()	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0



```
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(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] )
        print_token(pop(&top));
    push(token);
}
}
while ((token = pop(&top)) != eos)
    print_token(token);
print("\n");
}

```

$f(n)=\theta(g(n))$ iff there exist positive constants c_1 , c_2 , and n_0 such that $c_1g(n)\leq f(n)\leq c_2g(n)$ for all n , $n\geq n_0$.

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

$\theta(n)$

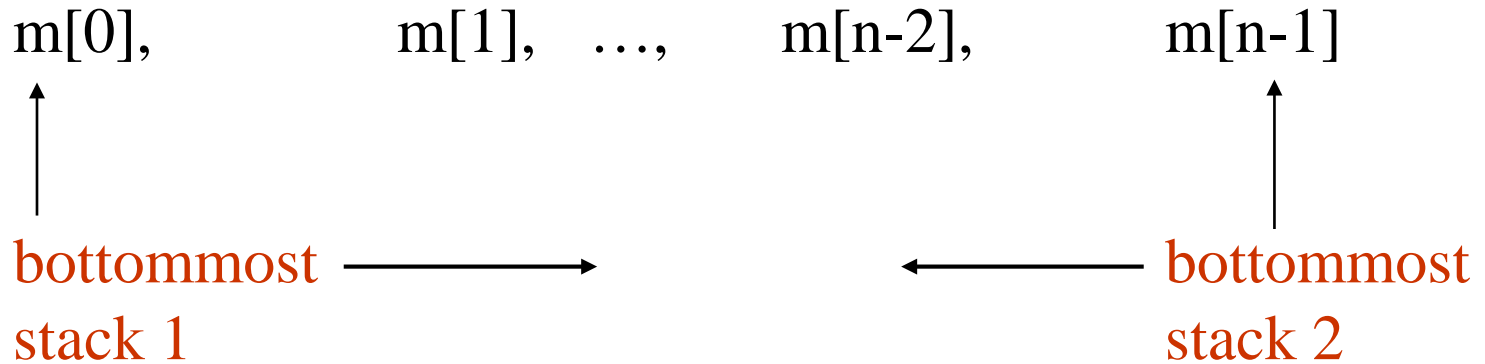
Infix	Prefix
$a*b/c$	<u>$/*abc$</u>
$a/b-c+d*e-a*c$	<u>$-+-/abc*de*ac$</u>
$a*(b+c)/d-g$	<u>$-/*a+bc$</u> <u><u><u><u>cdg</u></u></u></u>

- (1) evaluation
- (2) transformation

*Figure 3.17: Infix and postfix expressions

Multiple Stacks and Queues

Two stacks



More than two stacks (n)

memory is divided into n equal segments

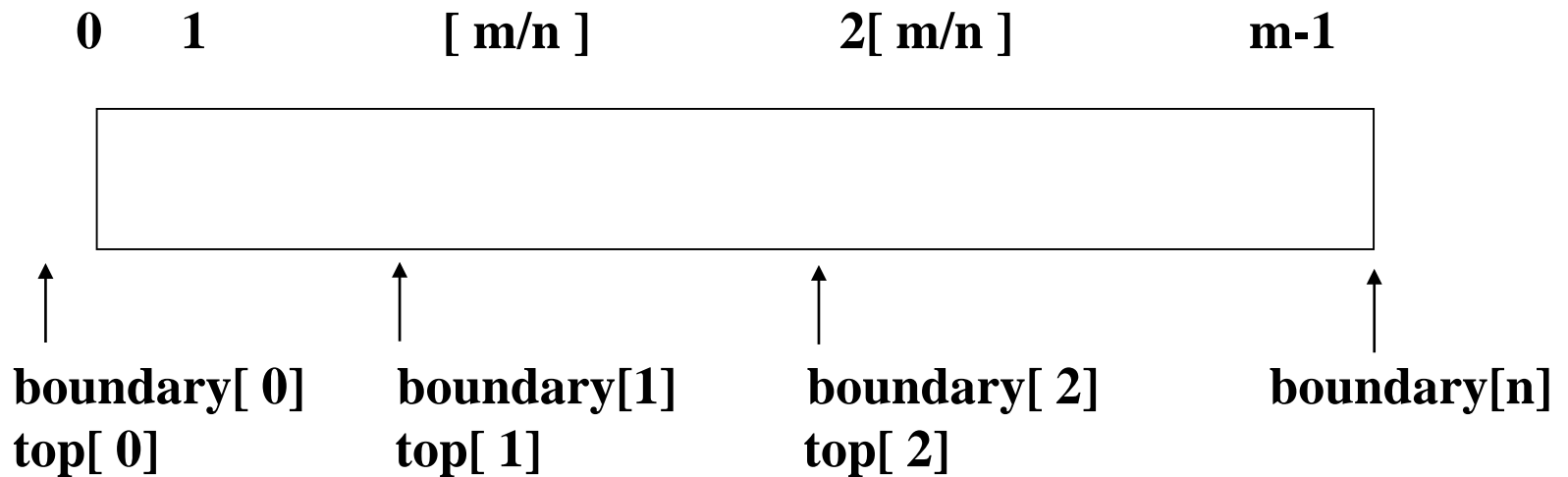
$\text{boundary}[\text{stack_no}]$

$0 \leq \text{stack_no} < \text{MAX_STACKS}$

$\text{top}[\text{stack_no}]$

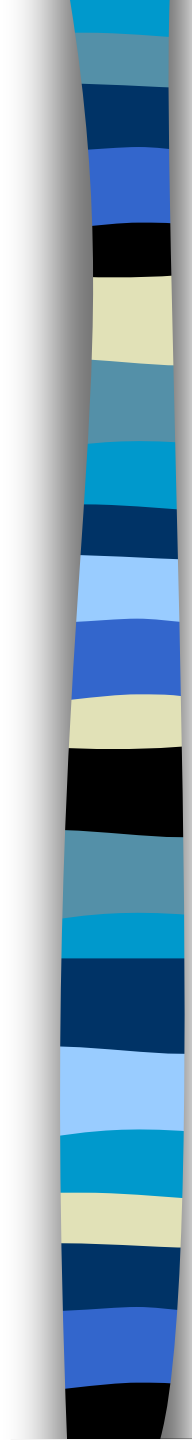
$0 \leq \text{stack_no} < \text{MAX_STACKS}$

Initially, $\text{boundary}[i]=\text{top}[i]$.



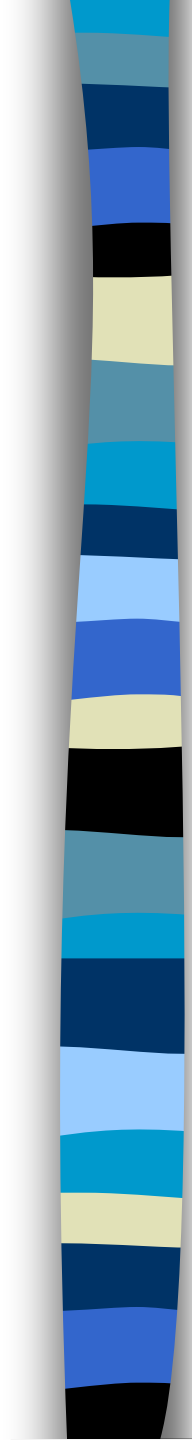
All stacks are empty and divided into roughly equal segments.

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



```
#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 */
```

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



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

*Program 3.16: Add an item to the stack *stack-no*

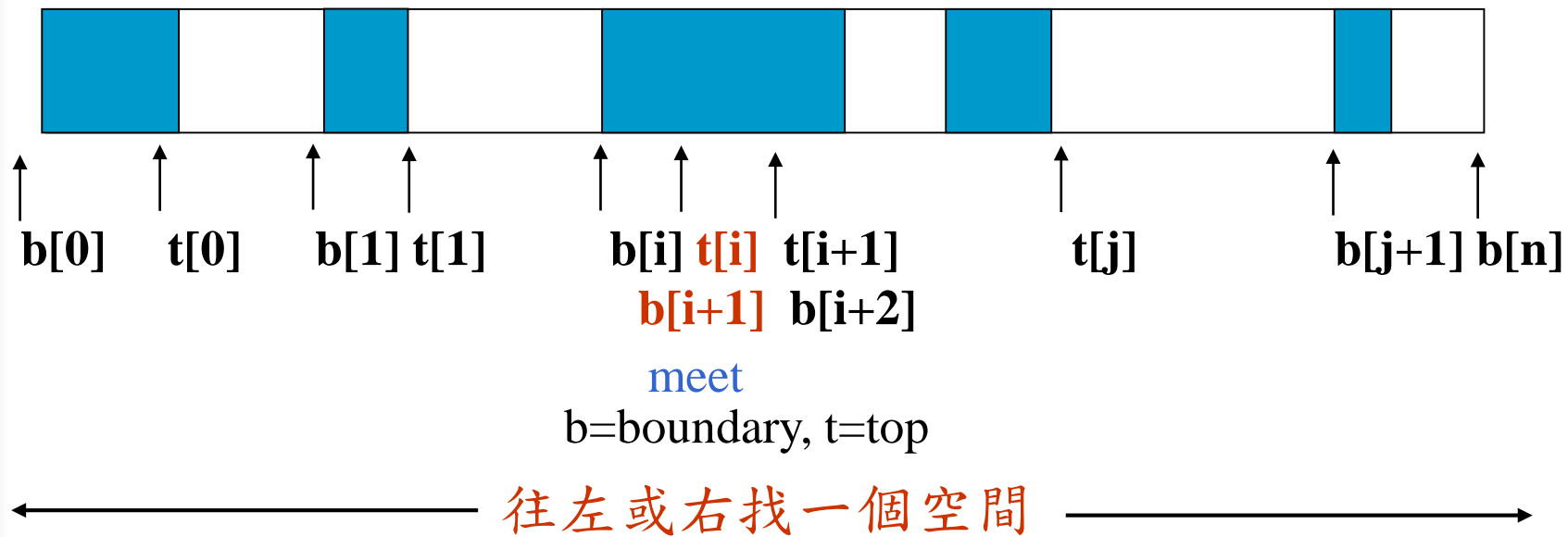
```
element pop(int i)
{
    /* remove top element from the ith stack */
    if (top[i] == boundary[i])
        return stackEmpty(i);
    return memory[top[i]--];
}
```

*Program 3.17: Delete an *item* from the stack *stack-no*

Find j , $\text{stack_no} < j < n$ (往右)

such that $\text{top}[j] < \text{boundary}[j+1]$

or, $0 \leq j < \text{stack_no}$ (往左)



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