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 /2nd Edition”,
Silicon Press, 2008.

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

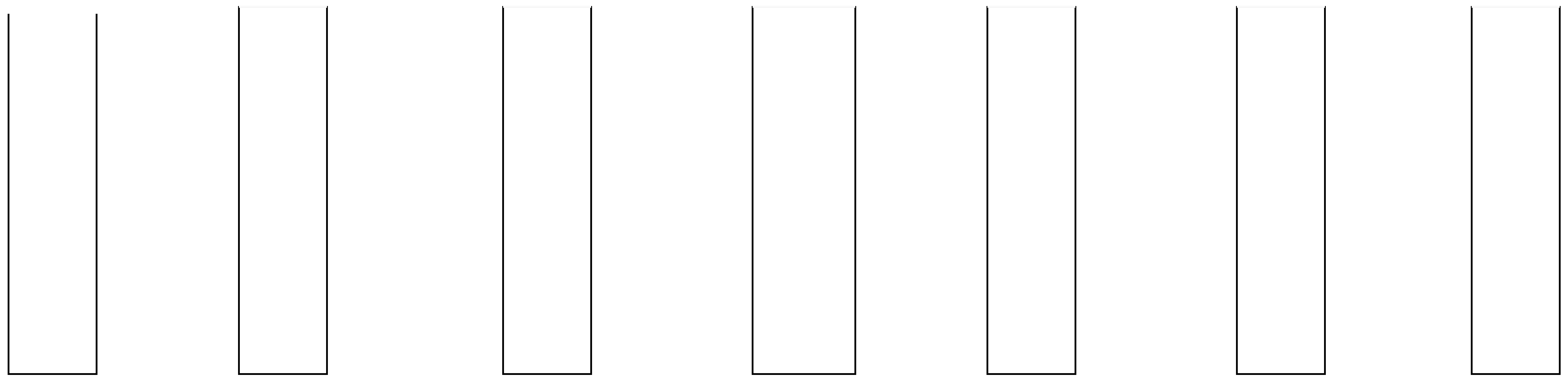
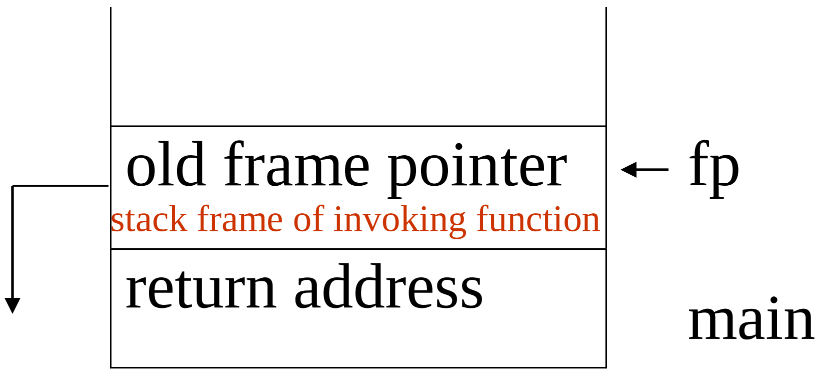


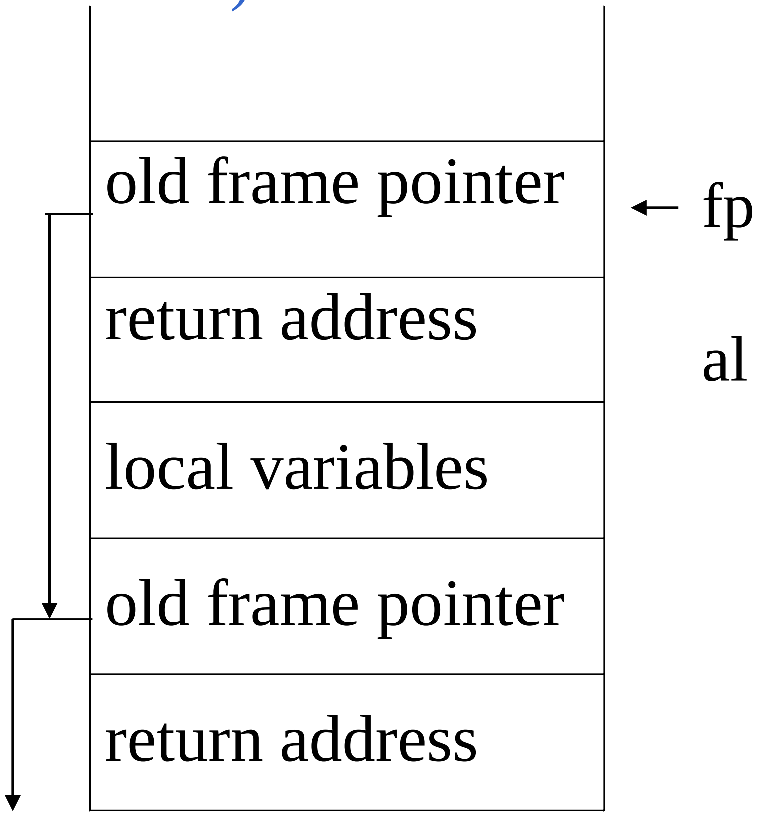
***Figure 3.1:** Inserting and deleting elements in a stack (p.108)

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

fp: a pointer to current stack frame



system stack before a1 is invoked



system stack after a1 is invoked

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

***Structure 3.1:** Abstract data type *Stack* (p.110)

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        (int *top, element item)
{
 /* add an item to the global stack */
    if (*top >= MAX_STACK_SIZE-1)  {
          stackFull( );
          return;
    }
                    = item;
}
```

<u>***program 3.1:** Add to a stack (p.111)</u>

Delete from a stack

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

***Program 3.2:** Delete from a stack (p.111)

Delete from a stack

```
void stackFull()
{
   fprintf(stderr, “Stack is full, cannot add element”);
   exit(EXIT_FAILRE);
 }
```

***Program 3.3:** Stack ful (p.111)

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

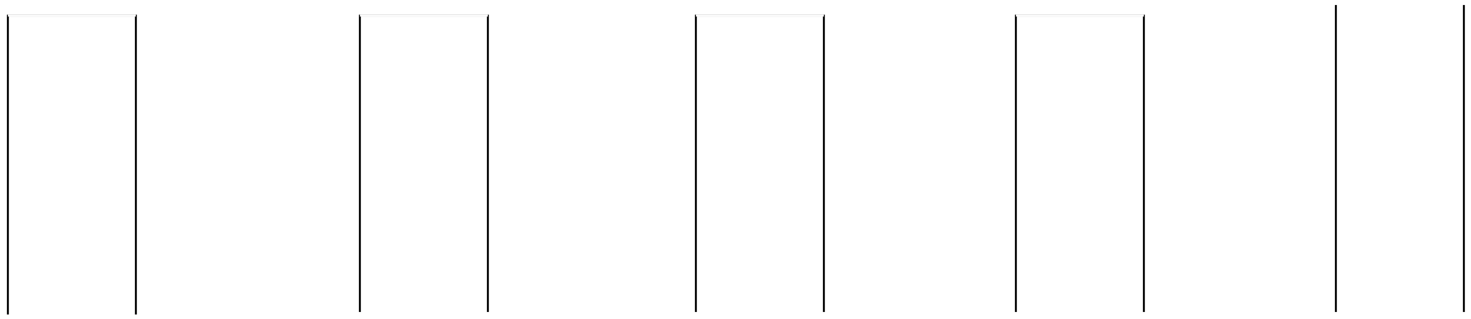


***Figure 3.4:** Inserting and deleting elements in a queue (p.114)

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

***Structure 3.2:** Abstract data type *Queue (p.115)*

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          (int *rear, element item)
{
/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE-1) {
      queueFull( );
      return;
   }
                          = item;
}
```

***Program 3.5:** Add to a queue (p.116)

Delete from a queue

```
element             (int *front, int rear)
{
/* remove element at the front of the queue */
   if (                           )
      return queueEmpty( );    /* return an error key */
   return                                ;
}
```

***Program 3.6:** Delete from a queue(p.116)*

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

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 (p.117)

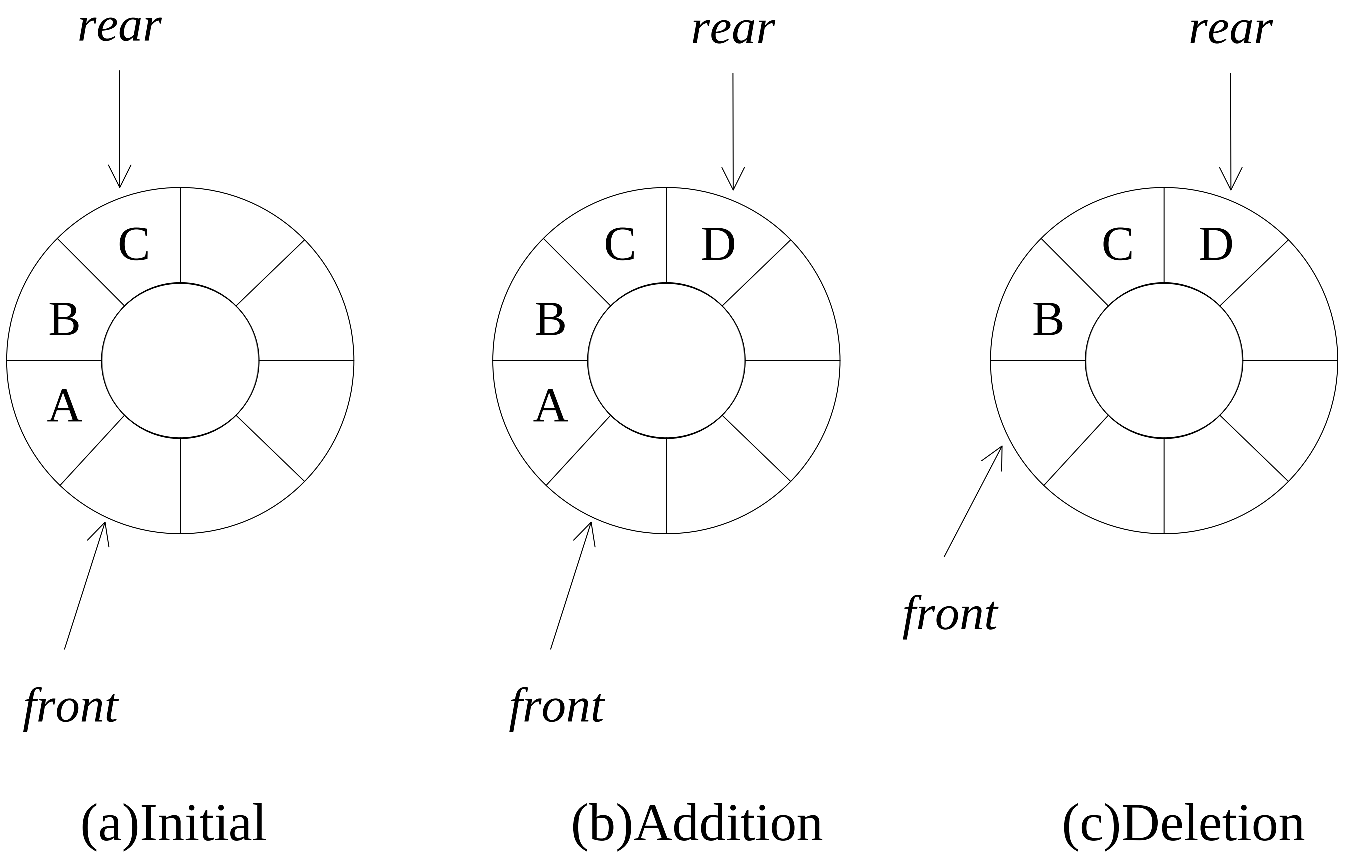
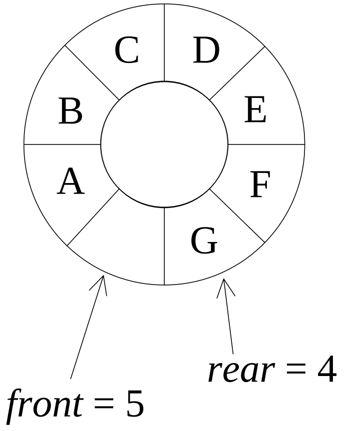


***Figure 3.6:** Circular queue (p.117)

Add to a circular queue



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

***Program 3.7:** Add to a circular queue (p.118)*

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 queueEmpty( );   /* returns an error key */

     return queue[*front];
}
```

***Program 3.8:** Delete from a circular queue (p.119)

A Mazing Problem

$$\begin{array}{r|ccccccccccccccc|l}
\text{entrance} \rightarrow & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & \\
 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & \\
 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & \\
 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & \\
 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & \\
 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & \\
 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \\
 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \\
 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & \rightarrow \text{exit}
\end{array}$$

***Figure 3.8:** An example maze(p.123)

a possible representation

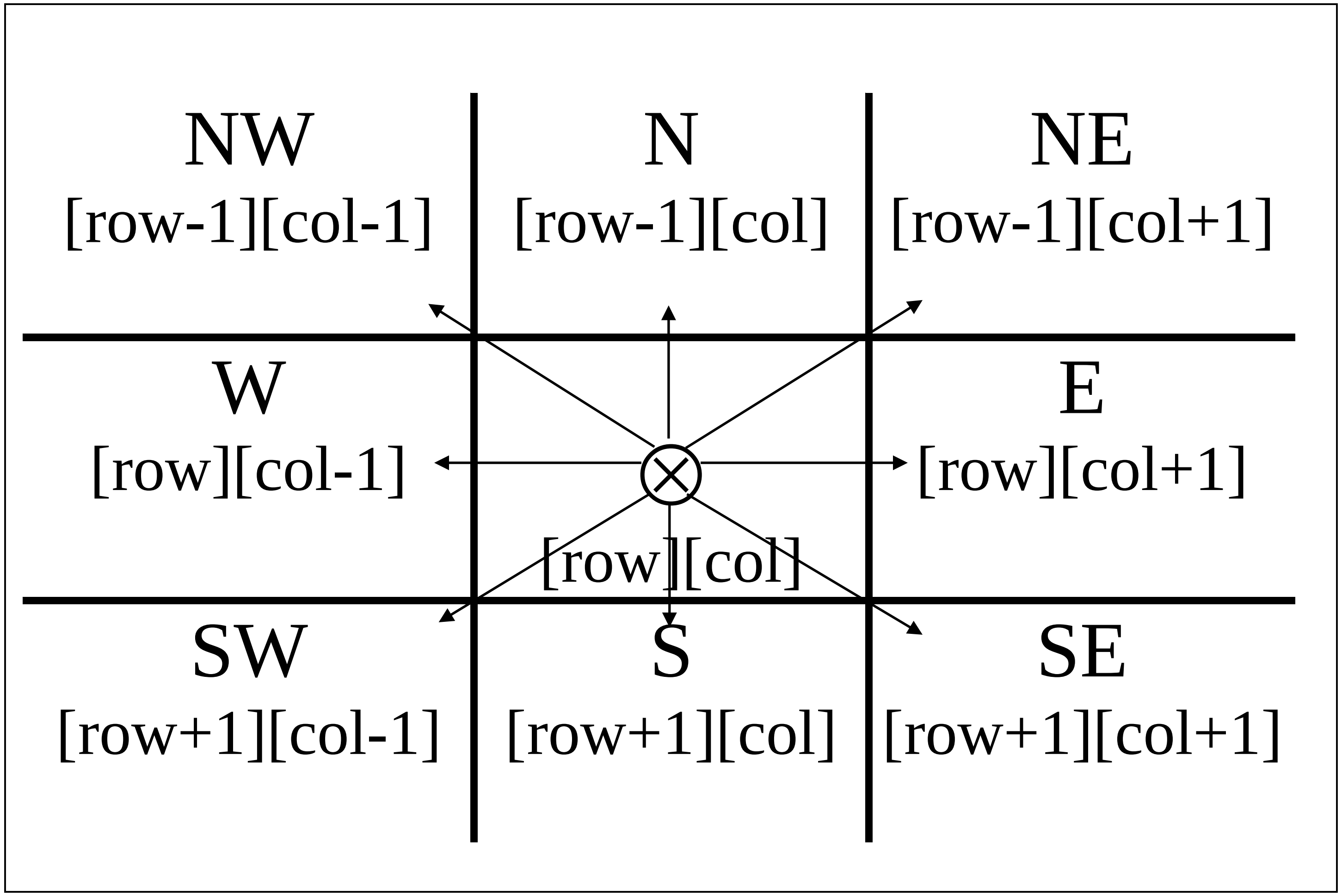


***Figure 3.9:** Allowable moves (p.124)

a possible implementation

```
typedef struct {
        short int vert;
        short int horiz;
        } offsets;
offsets move[8]; /*array of moves for each direction*/
```

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

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

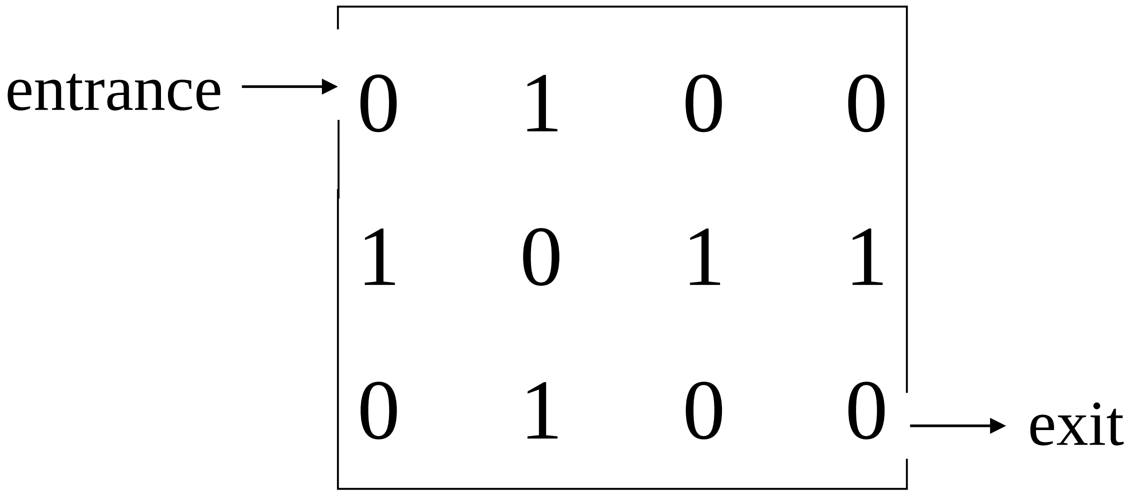
Use stack to keep pass history

```
#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;
     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 (p.126)


entrance
0 1 0 0
1 0 1 1
0 1 0 0
exit

The size of a stack?

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

mp --> $\lceil m/2 \rceil$ p,   mp --> $\lceil p/2 \rceil$ m

***Figure 3.11:** Simple maze with a long path (p.127)

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
x x	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= x=	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.130)

user **compiler**

Infix	Postfix
2+3*4	
a*b+5	
(1+2)*7	
a*b/c	
(a/(b-c+d))*(e-a)*c	
a/b-c+d*e-a*c	

***Figure 3.13:** Infix and postfix notation (p.131)

Postfix: no parentheses, no precedence

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

***Figure 3.14:** Postfix evaluation (p.131)

Infix to Postfix Conversion

(1) Fully parenthesize expression

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.

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

a + b * c, * > +

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

***Figure 3.15:** Translation of a+b*c to postfix (p.135)

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

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

*** Figure 3.16:** Translation of a*(b+c)*d to postfix (p.135)

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

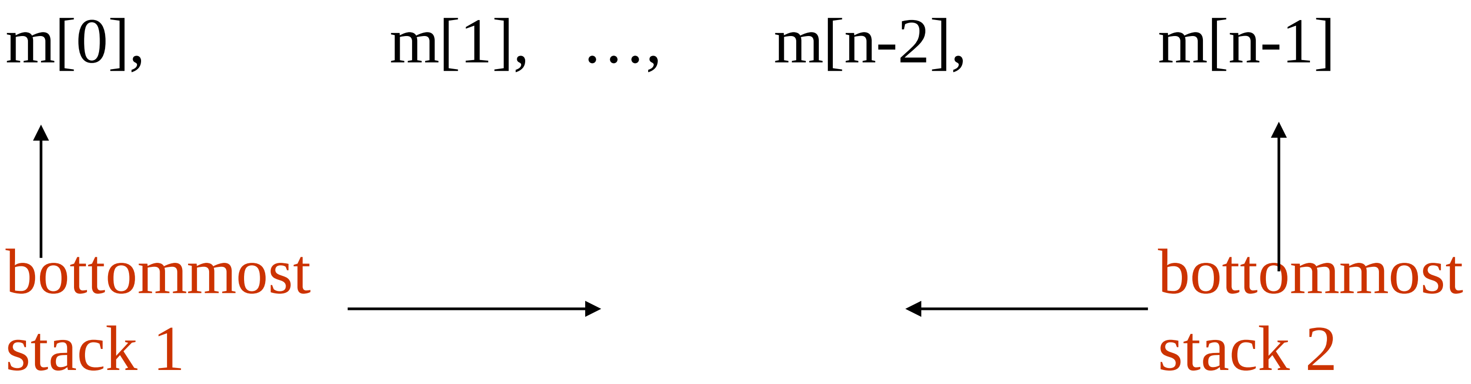
Infix	Prefix
a*b/c a/b-c+d*e-a*c a*(b+c)/d-g	

(1) evaluation
(2) transformation

***Figure 3.17:** Infix and postfix expressions (p.138)

Multiple stacks and queues

Two stacks



More than two stacks (n)

memory is divided into n equal segments

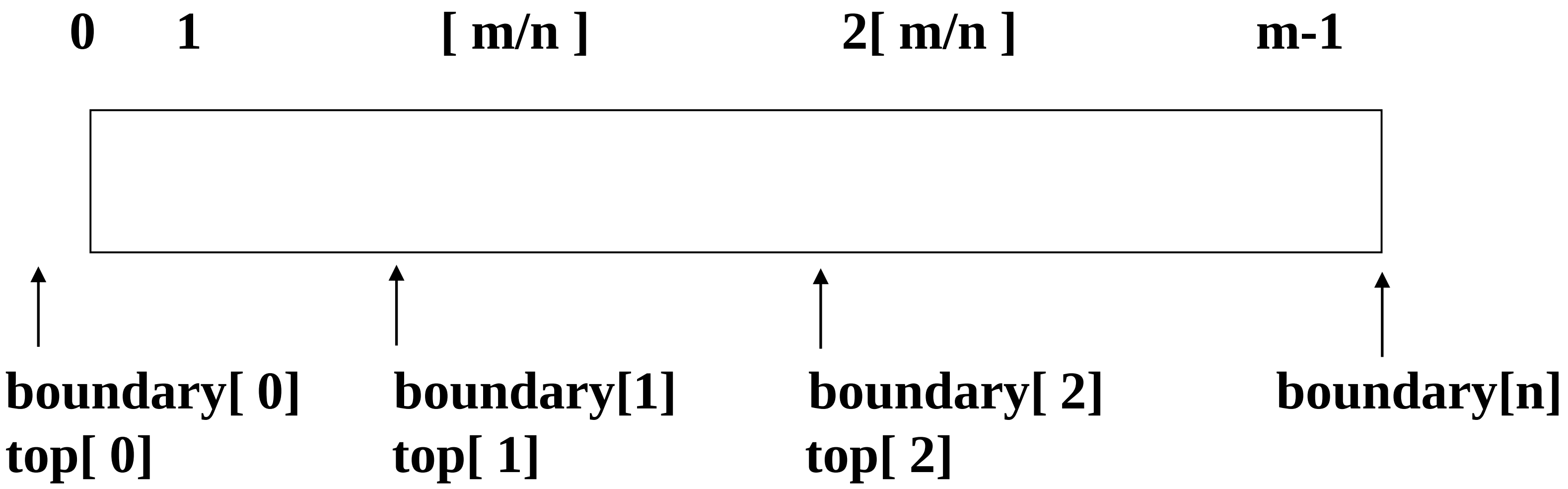
boundary[stack_no]

$0 \leq$ stack_no $<$ MAX_STACKS

top[stack_no]

$0 \leq$ stack_no $<$ MAX_STACKS

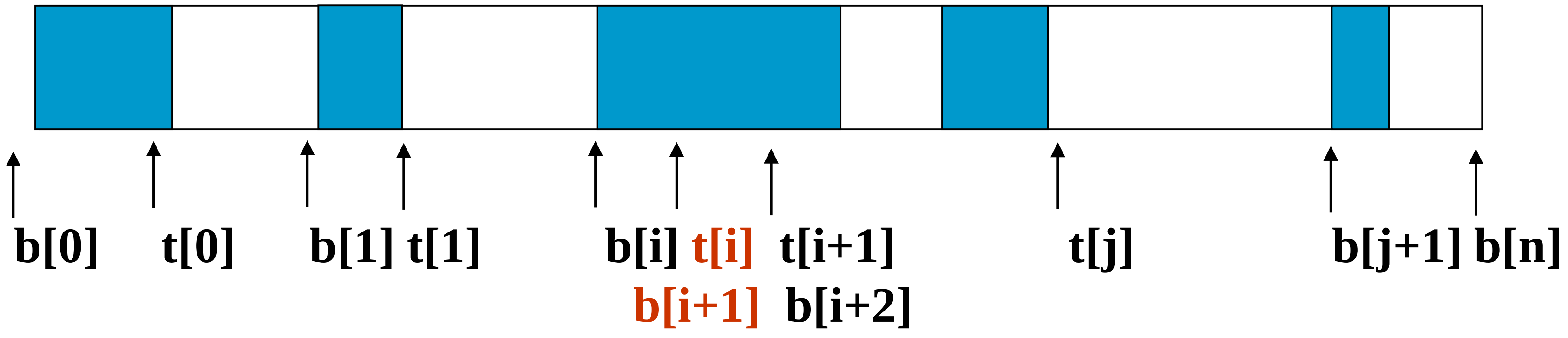
Initially, boundary[i]=top[i].



All stacks are empty and divided into roughly equal segments.

***Figure 3.18:** Initial configuration for *n* stacks in memory [m]. (p.140)

Find j, stack_no < j < n (往右)
such that top[j] < boundary[j+1]
or, 0 ≤ j < stack_no (往左)



meet
b=boundary, t=top

往左或右找一個空間

***Figure 3.19:** Configuration when stack i meets stack i+1,
but the memory is not full (p.141)