DATA STRUCTURE

Lecture 05: Stacks and Queues

STACK: LAST-IN-FIRST-OUT (LIFO) LIST

- Push
  - Add an element into a stack
- Pop
  - Get and delete an element from a stack

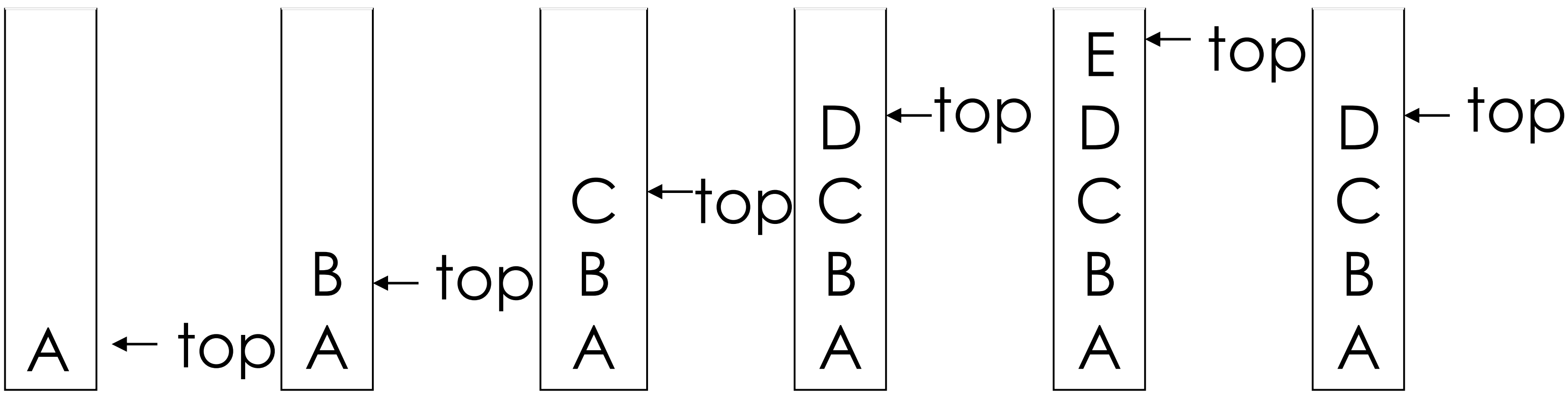
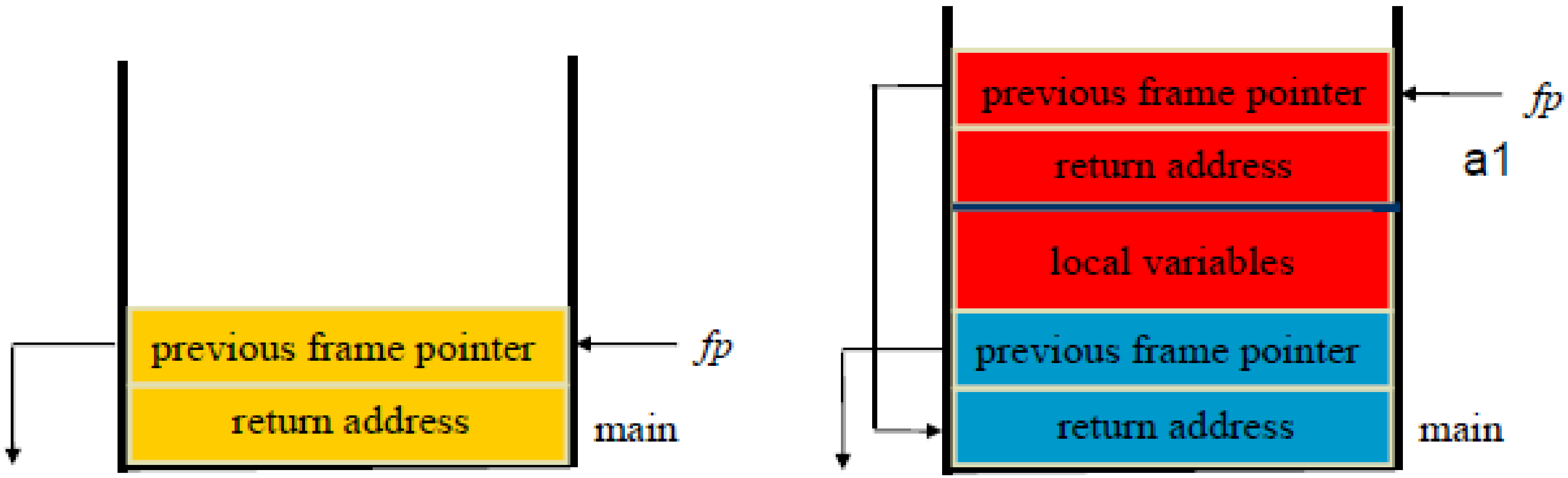


*Figure 3.1: Inserting and deleting elements in a stack (p.102)

AN APPLICATION OF STACK: STACK FRAME OF FUNCTION CALL



(a) (b)

System stack after function call

STACK ADT

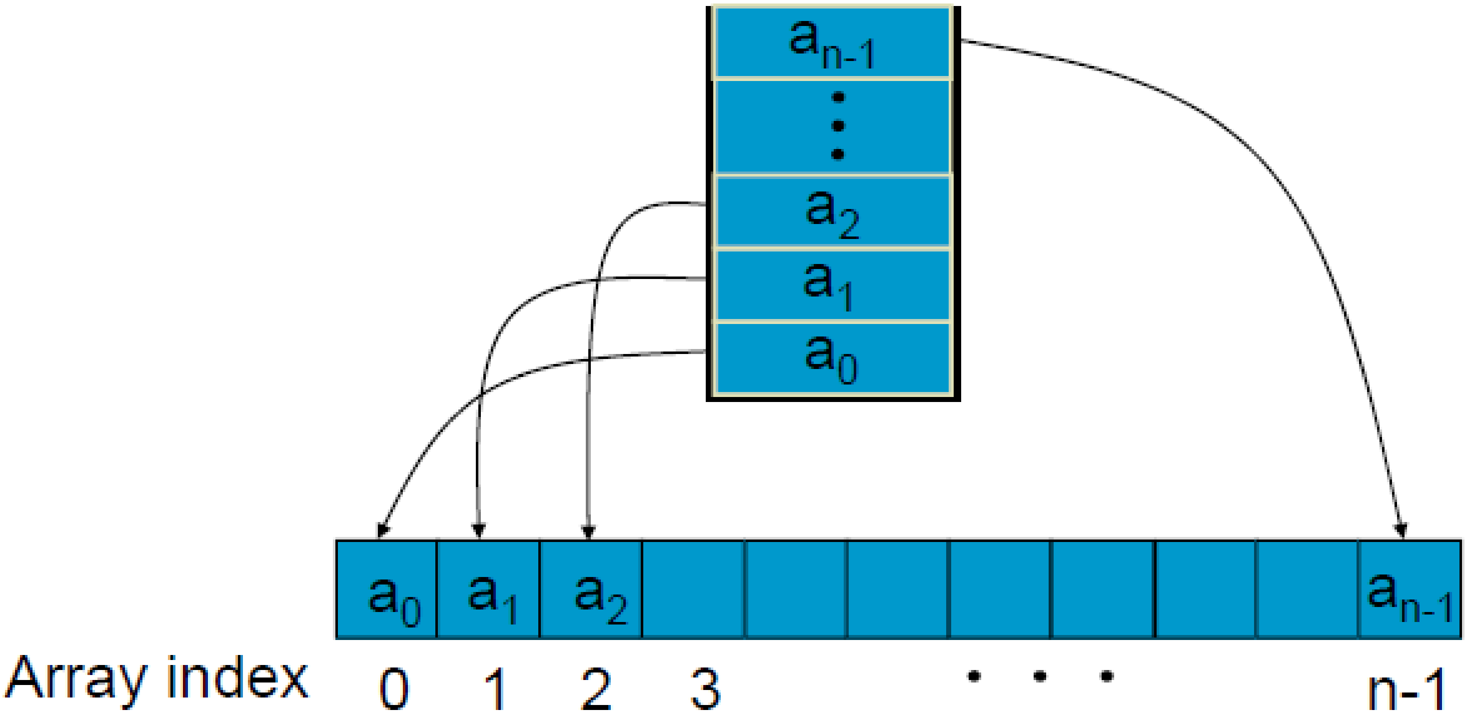
structure *Stack* is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer
   Stack CreateS(*max_stack_size*) ::=
      create an empty stack whose maximum size is *max_stack_size*
   Boolean IsFull(*stack, 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 3.1: Abstract data type *Stack* (p.104)

IMPLEMENTATION OF STACK BY ARRAY

- How to check whether a stack is full or empty?



***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;
```

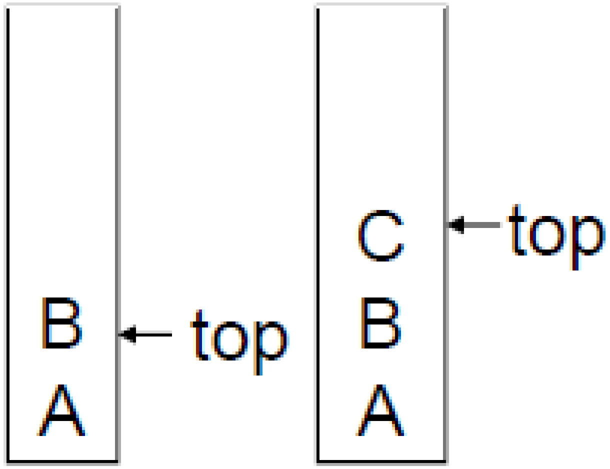
***Boolean* IsEmpty(Stack)** ::= top< 0;

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

ADDING TO A STACK

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

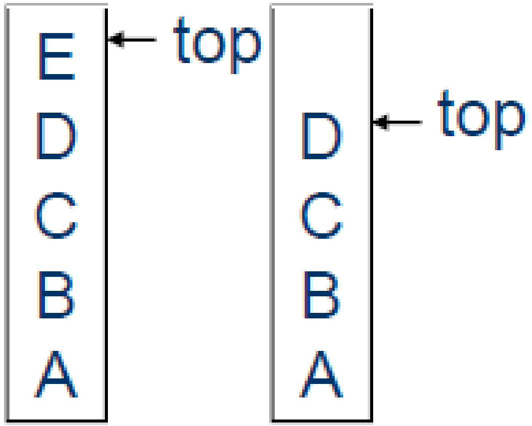
***program 3.1:** Add to a stack (p.104)



DELETING FROM A STACK

```
element delete(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 (p.105)



QUEUE:
FIRST-IN-FIRST-OUT (FIFO) LIST

- Add an element into a queue
- Get and delete an element from a queue
- Variation
    - Priority queue

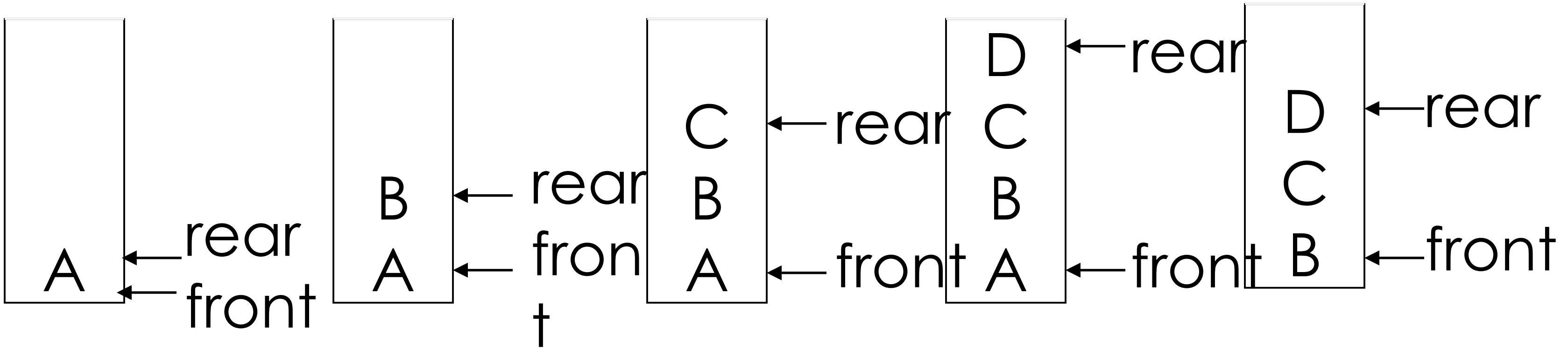


*Figure 3.4: Inserting and deleting elements in a queue (p.106)

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.108)

QUEUE ADT

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*

QUEUE ADT

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 (p.107)*

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 3.3:** Add to a queue (p.108)

DELETING FROM A QUEUE

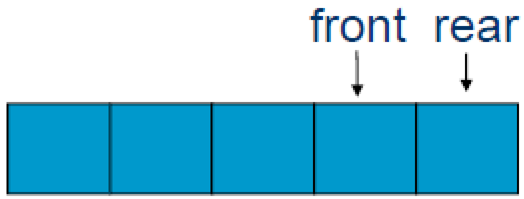
```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
   if ( *front == rear)
      return queue_empty( );     /* return an error key */
   return queue [++ *front];
}
```

***Program 3.4:** Delete from a queue(p.108)

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

PROBLEM

- As the elements enter and leave the queue, the queue gradually shifts to the right.
  - Eventually the rear index equals *MaxSize-1*, suggesting that the queue is full even though the underlying array is not full
- Solution:
  - Use a function to move the entire queue to the left so that *front*=-1
  - It is time-consuming
  - Time complexity=O(MaxSize)

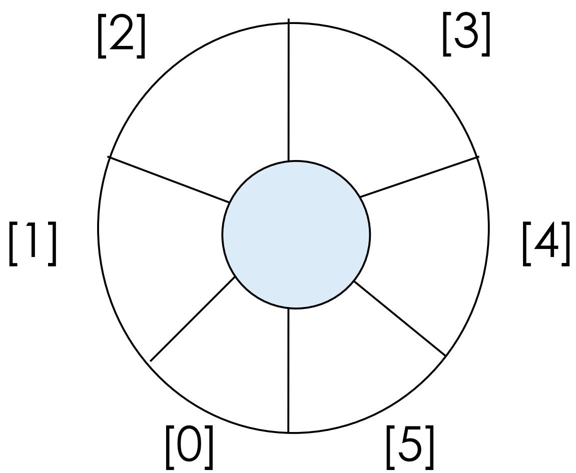


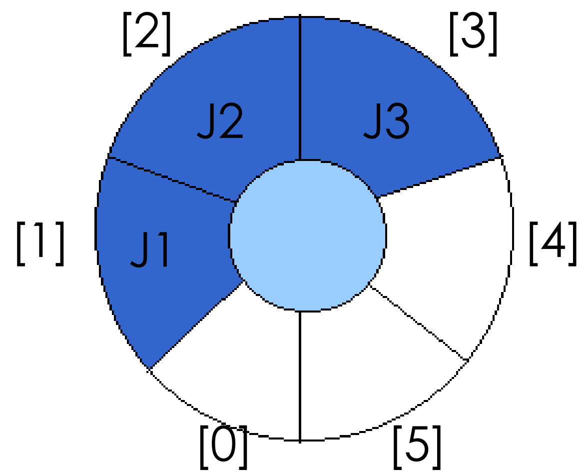
IMPLEMENTATION 2: REGARD AN ARRAY AS A CIRCULAR QUEUE

- Two indices
  - front: one position counterclockwise from the first element
  - rear: current end
- Problem
  - In order to distinguish whether a circular queue is full or empty, one space is left when queue is full

AN EXAMPLE OF CIRCULAR QUEUE

EMPTY QUEUE





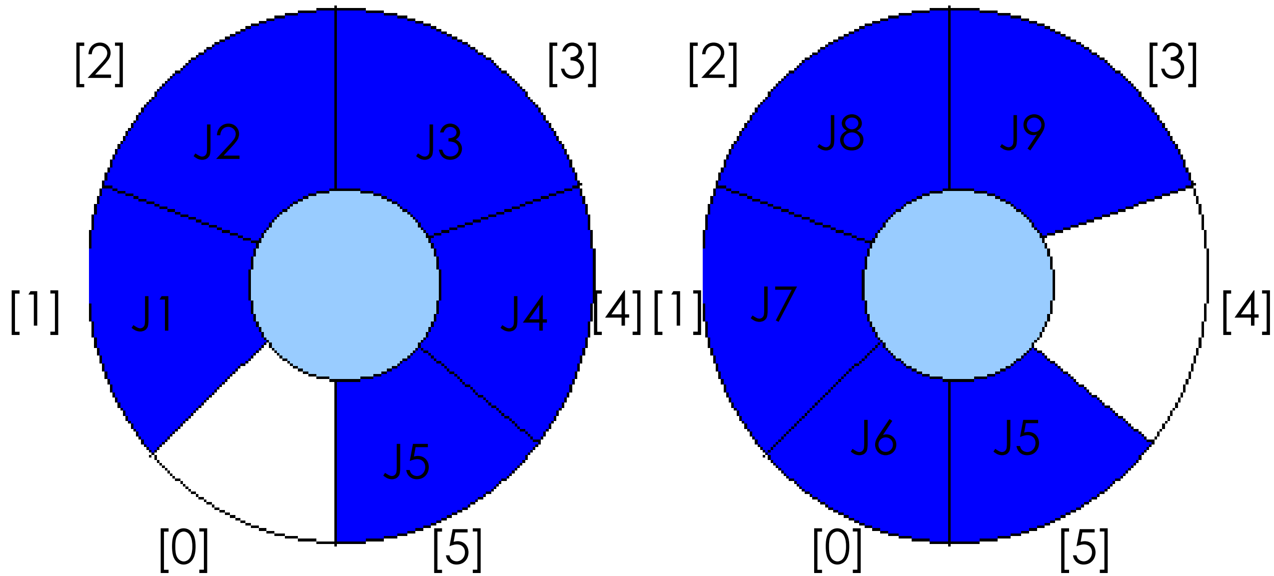
front = 0
rear = 0

front = 0
rear = 3

*Figure 3.6: Empty and nonempty circular queues (p.109)

Problem: one space is left when queue is full

FULL QUEUE   FULL QUEUE



front =0
rear = 5

front =4
rear =3

*Figure 3.7: Full circular queues and then we remove the item (p.110)

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 3.5:** Add to a circular queue (p.110)

DELETING 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 3.6:** Delete from a circular queue (p.111)

A MAZING PROBLEM

entrance →

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

→ exit

1: blocked path 0: through path

*Figure 3.8: An example maze(p.113)

A POSSIBLE REPRESENTATION

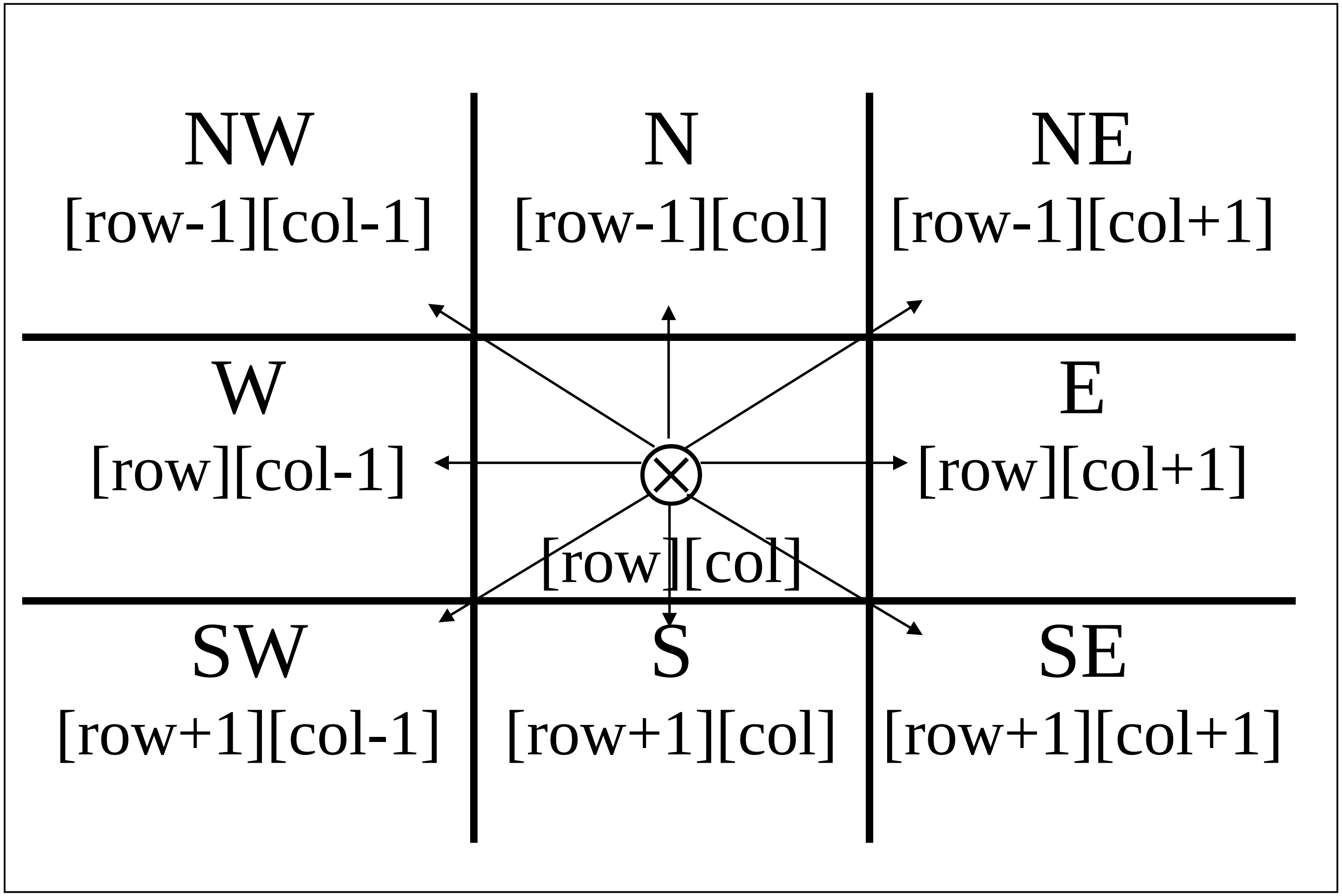


*Figure 3.9: Allowable moves (p.113)

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");
```

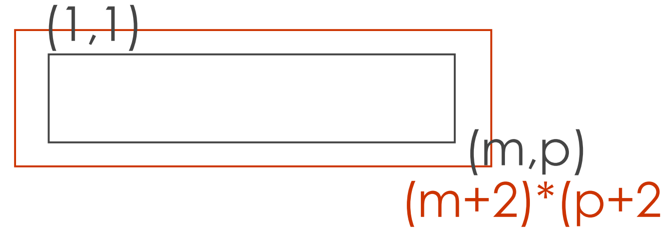
***Program 3.7:** Initial maze algorithm (p.115)

THE SIZE OF A STACK?

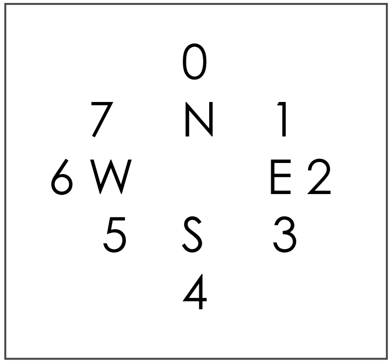
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{m*p}$$

$$mp \rightarrow [m/2]p, \quad mp \rightarrow [p/2]m$$

*Figure 3.11: Simple maze with a long path (p.116)




```
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 = delete(&top);
      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;
```



```
if (next_row==EXIT_ROW && next_col==EXIT_COL)
            found = TRUE;
       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;
            add(&top, 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 3.8:**Maze search function (p.117)

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

EVALUATION OF EXPRESSIONS (CONT'D)

- Infix:
  - Each operator comes in-between the operands
  - 2+3
- Postfix
  - Each operator appears after its operands
  - 23+
- Prefix
  - Each operator appears before its operands
  - +23

EVALUATION OF EXPRESSIONS (CONT'D)

user **computer**

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*-+

Postfix & prefix: no parentheses, no precedence

EVALUATION OF EXPRESSIONS (CONT'D)

- Phase 1: Infix to postfix conversion
  - 6/2-3+4*2 → 6 2 / 3 – 4 2 * +
- Phase 2: Postfix expression evaluation
  - 6 2 / 3 – 4 2 * + → 8

PHASE 2: POSTFIX EXPRESSION EVALUATION

6 2 / 3 – 4 2 * +

Token	Stack [0]	Stack [1]	Stack [2]	Top
6	6			0
2	6	2		1
/	3			0
3	3	3		1
-	0			0
4	0	4		1
2	0	4	2	2
*	0	8		1
+	8			0

POSTFIX EVALUATION

- 23-5*93/+

Token	Stack					Top
	[0]	[1]	[2]	[3]	[4]	
2						
3						
-						
5						
*						
9						
3						
/						
+						

GOAL: INFIX --> POSTFIX

```
#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 */
```

Assumptions:

operators: +, -, *, /, %

operands: single digit integer

```
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)
          add(&top, symbol-'0');   /* stack insert */
```

exp: character array

```
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)

PHASE 1: INFIX TO POSTFIX CONVERSION

- Assumptions:
  - operators: +, -, *, /, %
  - operands: single digit integer

PHASE 1: INFIX TO POSTFIX CONVERSION

Intuitive Algorithm

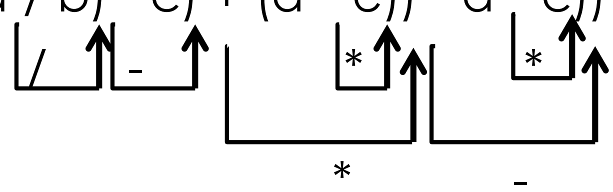
(1) Fully parenthesize expression

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*-

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*+

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	ab*c
<eos>		-1	ab*c+

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

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>				-1	abc+$*_1$d$*_2$

RULES

- Operators are taken out of the stack as long as their in-stack precedence is numerically less than or equal to the incoming precedence of the new operator, i.e., isp(y)<=icp(x)
- "(" has lowest in-stack precedence (i.e., 8), and highest incoming precedence (i.e.,0).
  - No operator other than the matching right parenthesis ")" should cause it to get unstacked

RULES (CONT'D)

Priority	Operator
1	Unary minus, !
2	*,/,%
3	+,-
4	<,<=,>=,>
5	==,!=
6	&&
7	||

EXERCISE

- Infix → Postfix
  - (1) a+b*c-d/e
  - (2) (a+b)*(c-d)/(e*f)^g
  - (3) A^B^C
  - (4) ~(A>B) and (C or D<E) or ~F

- Postfix → Infix
  - (1) abc-d+/ea-*c*
  - (2) ABCDE-+^*EF*-

(SUPPLEMENT) INFIX TO PREFIX CONVERSION

Intuitive Algorithm

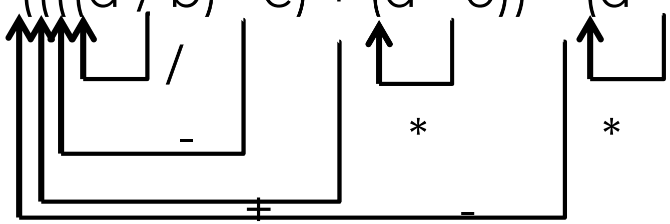
(1) Fully parenthesize expression

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.

-+/-/abc*de*ac

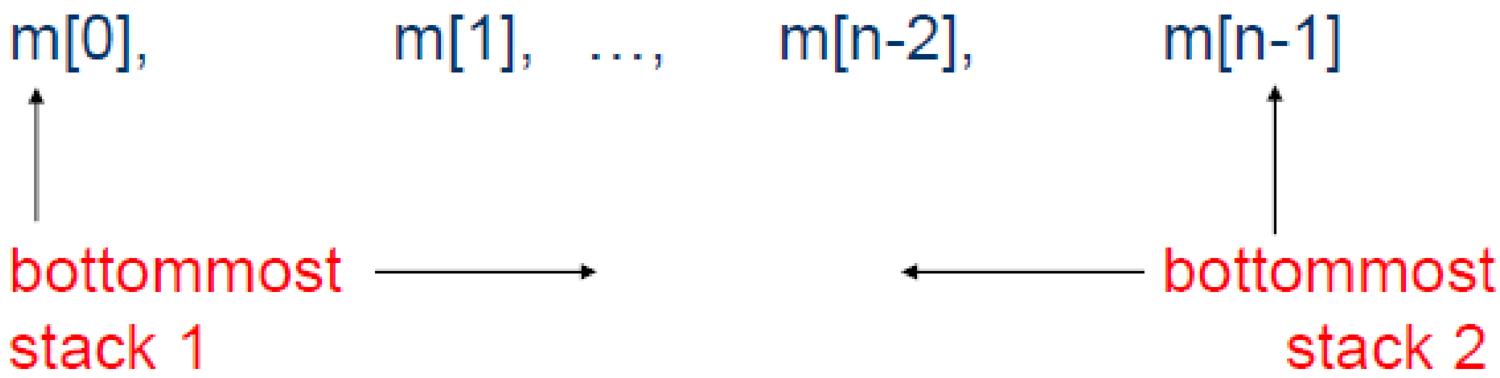
EXERCISE

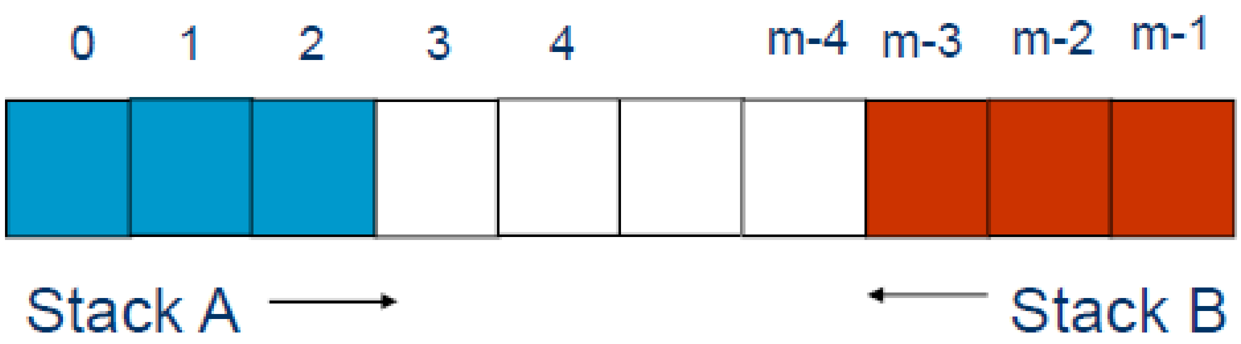
- Infix → Prefix
  - (1) a+b*c-d/e
  - (2) (a+b)*(c-d)/(e*f)^g
  - (3) A^B^C
  - (4) ~(A>B) and (C or D<E) or ~F

- Prefix → Infix
  - (1) +*/ab-+cde-fg
  - (2) -/*a+b*cdef

MULTIPLE STACKS AND QUEUES

- Two stacks



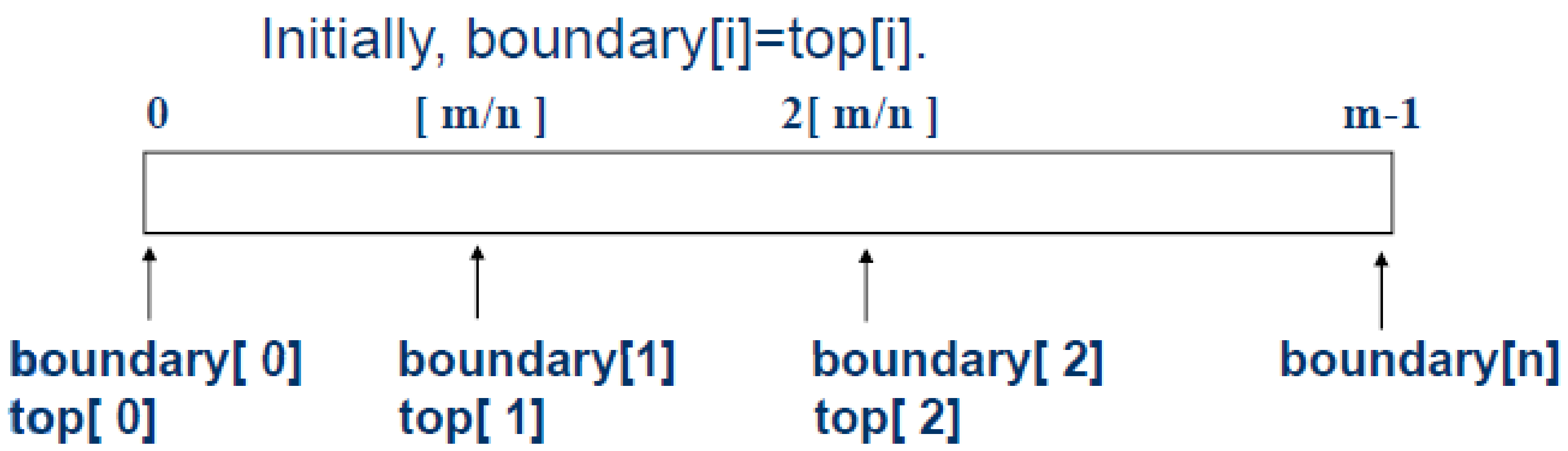


MULTIPLE STACKS AND QUEUES (CONT'D)

- More than two stacks (n)
- Memory is divided into n segments
  - The initial division of these segments may be done in proportion to expected sizes of these stacks if these are known
  - All stacks are empty and divided into roughly equal segments

MULTIPLE STACKS AND QUEUES (CONT'D)

- boundary[stack_no]
  - 0 ≤ stack_no < MAX_STACKS
- top[stack_no]
  - 0 ≤ stack_no < MAX_STACKS



```
#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)

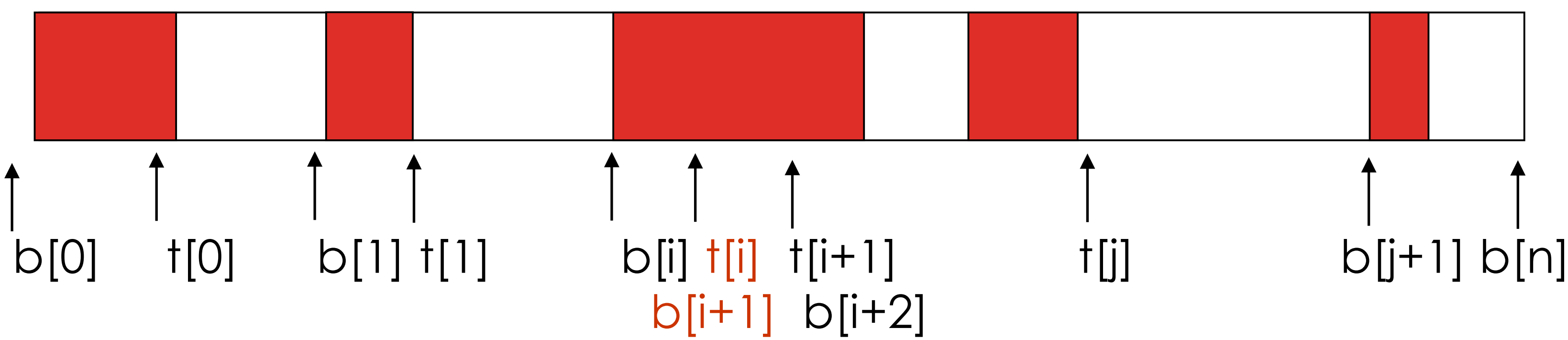
```
element delete(int i)
{
   /* remove top element from the ith stack */
   if (top[i] == boundary[i])
     return stack_empty(i);
   return memory[top[i]--];
}
```

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

Find j, stack_no < j < n (to right)

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

or, 0 ≤ j < stack_no (to left)



meet

b=boundary, t=top

Find an available space

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