Chapter 3
Stacks and Queues

Yi-Fen Liu

Department of IECS, FCU

References:
- E. Horowitz, S. Sahni and S. Anderson-Freed, *Fundamentals of Data Structures (2nd Edition)*
- Slides are credited from Prof. Chung, NTHU

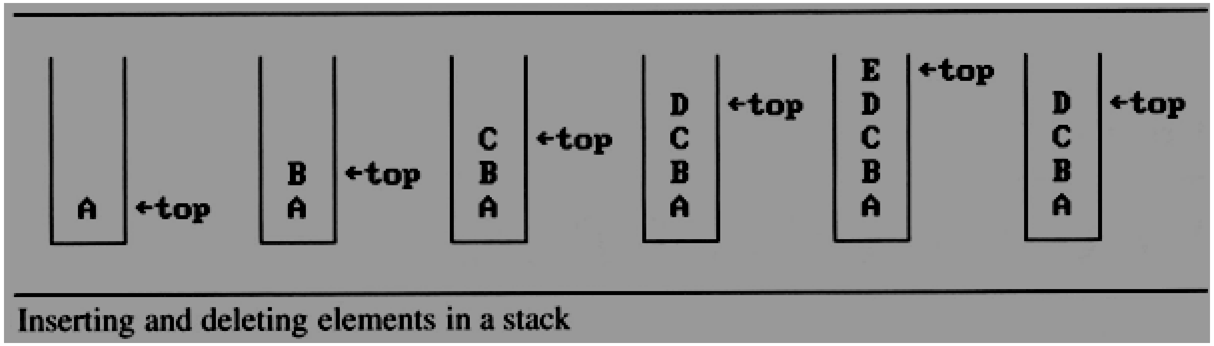
Outline

- The Stack Abstract Data Type
- The Queue Abstract Data Type
- A Mazing Problem
- Evaluation of Expressions

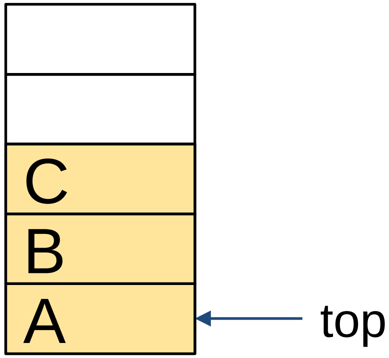
THE STACK ADT

The Stack ADT (1)

- A stack is an ordered list in which insertions and deletions are made at one end called the top
  - If we add the elements A, B, C, D, E to the stack, in that order, then E is the first element we delete from the stack
- A stack is also known as a Last-In-First-Out (LIFO) list



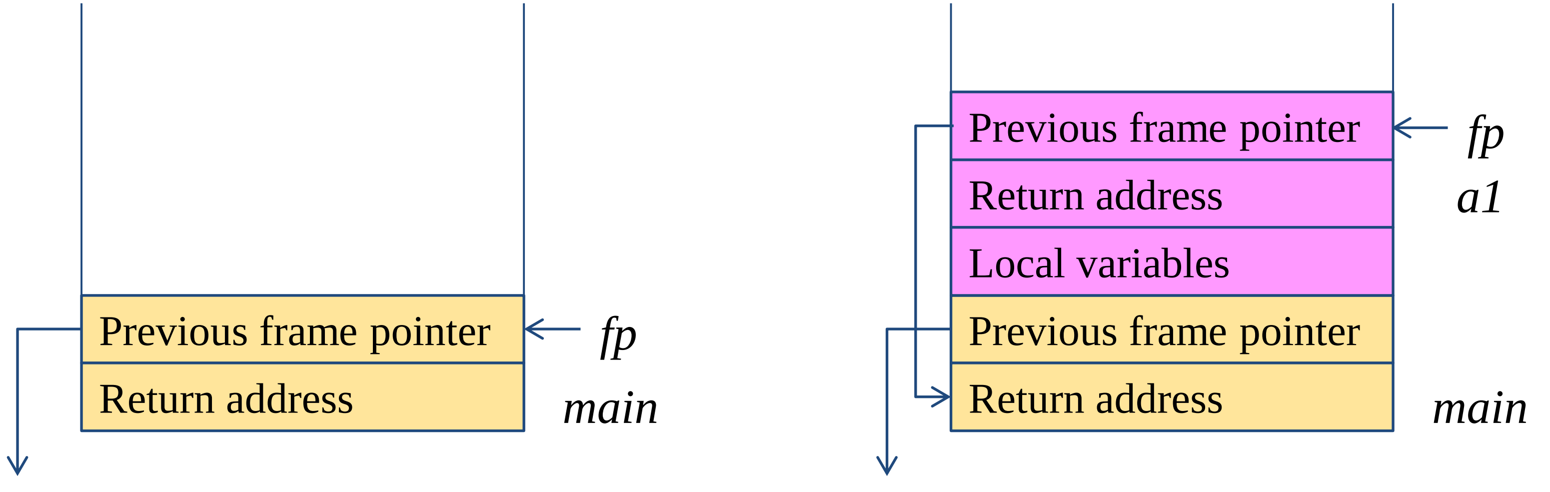
Inserting and deleting elements in a stack



The Stack ADT (2)

- System stack
  - Stack frame of function call

fp: a pointer to current stack frame



system stack before *a1* is invoked

system stack after *a1* is invoked

The Stack ADT (3)

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.

Abstract data type *Stack*

The Stack ADT (4)

- 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;
```

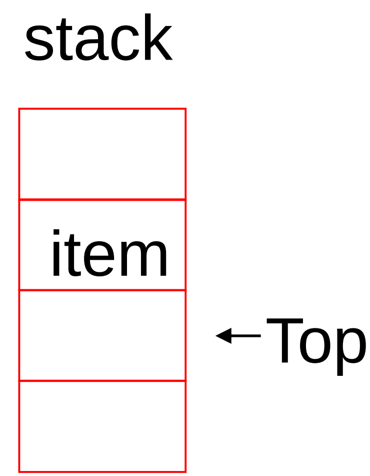
Boolean IsEmpty(Stack) ::= `top < 0;`

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

The Stack ADT (5)

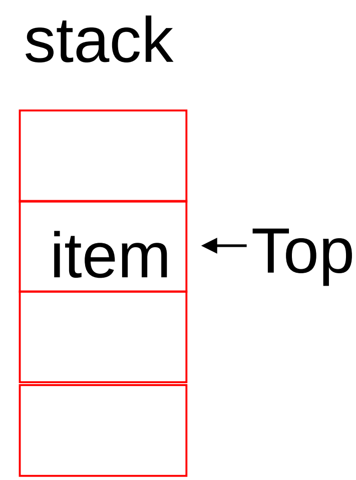
```c
void add(int *top, element item)
{
/* add an item to the global stack */
   if (*top >= MAX_STACK_SIZE-1) {
      stack_full();
      return;
   }
   stack[++*top] = item;
}
```

Add to a stack




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

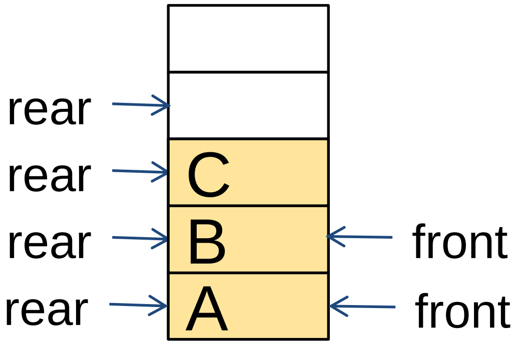
Delete from a stack

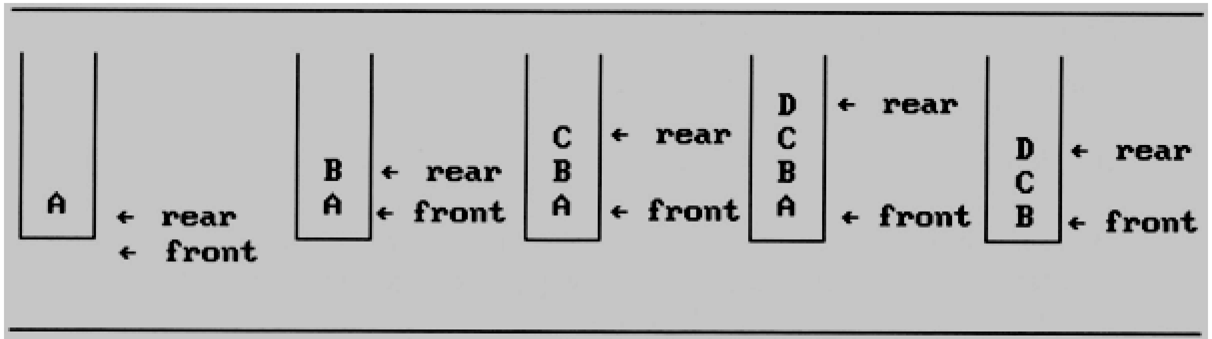


THE QUEUE ADT

The Queue ADT (1)

- A queue is an ordered list in which all insertion take place one end, called the rear and all deletions take place at the opposite end called the front
- First-In-First-Out (FIFO) list
- If we insert the elements A, B, C, D, E, in that order, then A is the first element we delete from the queue as a





Inserting and deleting elements in a queue

The Queue ADT (2)

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.

Abstract data type *Queue*

The Queue ADT (3)

- Using a one dimensional array and two variables, *front* and *rear*

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 IsEmptyQ(*queue*) ::= `front == rear`

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

The Queue ADT (4)

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

```
void addq(int *rear, element item)
{
/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE-1) {
      queue_full();
      return;
   }
   queue[++*rear] = item;
}
```

Add to 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];
}
```

Delete from a queue

Example: Job scheduling

- The figure illustrates how an operating system might process jobs if it used a sequential representation for its queue.

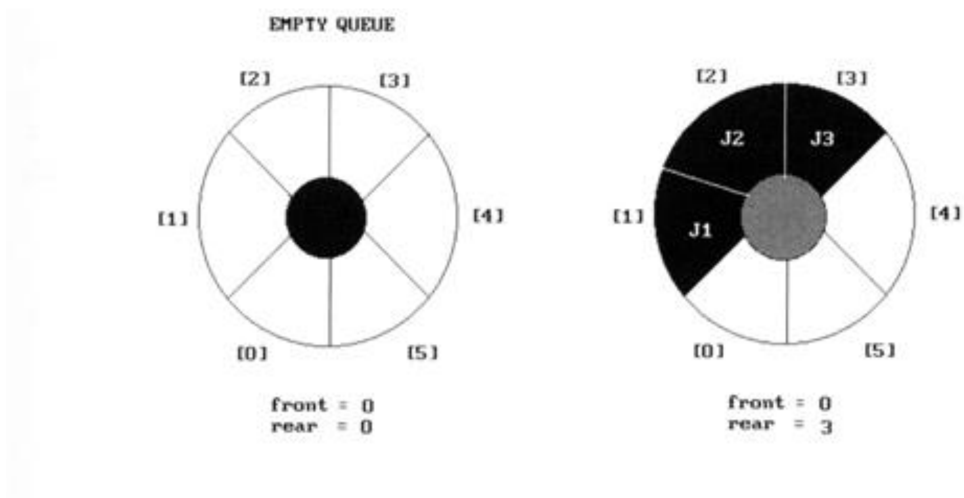
front	*rear*	*Q*[0]	*Q*[1]	*Q*[2]	*Q*[3]	Comments
−1	−1					queue is empty
−1	0	J1				Job 1 is added
−1	1	J1	J2			Job 2 is added
−1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Insertion and deletion from a sequential queue

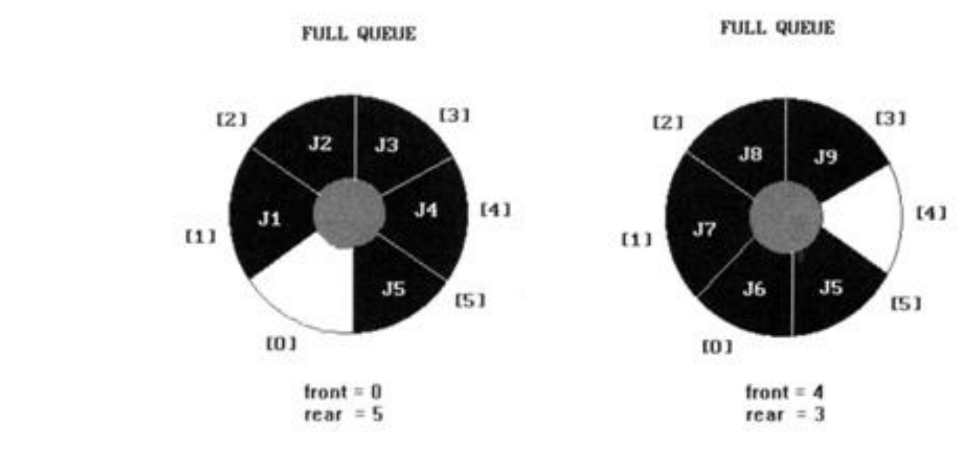
- As jobs enter and leave the system, the queue gradually shift to right.
- In this case, queue_full should move the entire queue to the left so that the first element is again at queue[0], front is at -1, and rear is correctly positioned.
  - Shifting an array is very time-consuming, queue_full has a worst case complexity of O(MAX_QUEUE_SIZE).

Circular Queue (1)

- We can obtain a more efficient representation if we regard the array queue[MAX_QUEUE_SIZE] as circular
  - front: one position counterclockwise from the first element
  - rear: current end
- Problem: one space is left when queue is full



Empty and nonempty circular queues



Full circular queues

Circular Queue (2)

- Implementing addq and deleteq for a circular queue is slightly more difficult since we must assure that a circular rotation occurs

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

Add to a circular queue

Circular Queue (3)

```
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];
}
```

A MAZING PROBLEM

A Mazing Problem (1)

- Representation of the maze
  - The most obvious choice is a two dimensional array
    - 0s the open paths and 1s the barriers
  - Notice that not every position has eight neighbors
    - To avoid checking for these border conditions we can surround the maze by a border of ones. Thus an $m \times p$ maze will require an $(m+2) \times (p+2)$ array
    - The entrance is at position [1][1] and the exit at [m][p]

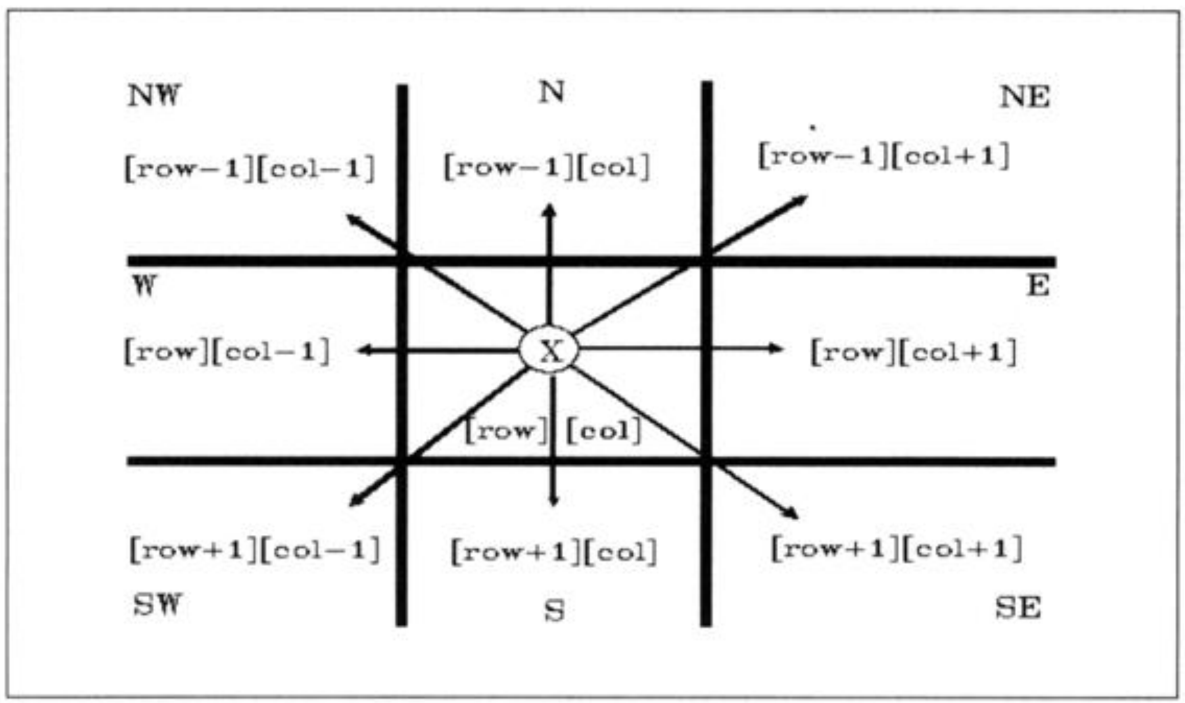
entrance

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1
1	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1
1	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	1
1	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0	1
1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	1
1	0	1	1	1	1	0	0	1	1	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	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	1
1	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

exit

A Mazing Problem (2)

- If X marks the spot of our current location, *maze*[*row*][*col*], then the figure shows the possible moves from this position



Allowable moves

A Mazing Problem (3)

- A possible implementation
  - Predefinition: the possible directions to move in an array as in the figure

```
typedef struct {
    short int vert;
    short int horiz;
} offsets;
```

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

Table of moves

```
offsets move[8]; /*array of moves for each direction*/
```

  - If we are at position, *maze*[*row*][*col*], and we wish to find the position of the next move
    - next_row = row + move[dir].vert;
    - next_col = col + move[dir].horiz;

A Mazing Problem (4)

- Initial attempt at a maze traversal algorithm
  - Maintain a second two-dimensional array, *mark*, to record the maze positions already checked
  - 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];
```

A Mazing Problem (5)

```
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");
```

Initial maze algorithm

Pop out

R 1 C 1 D 1

R: row
C: col
D: dir

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

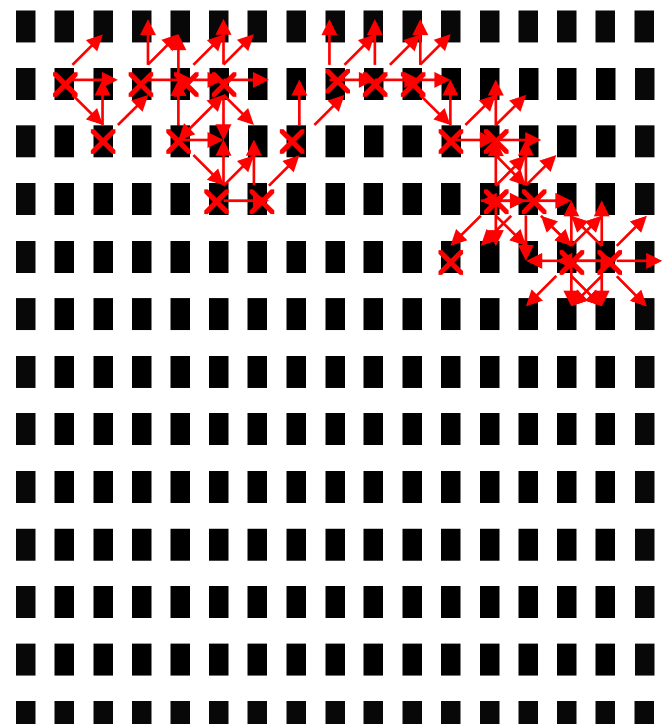
Figure 3.10: Table of moves

maze[1][1]: entrance

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1	1
1	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1	1
1	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	1
1	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0	1
1	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1	1
1	0	1	1	1	1	0	0	1	1	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	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	1
1	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

maze[11][15]: exit

Initially set mark[1][1]=1



A Mazing Problem (7)

- Review of *add* and *delete* 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;
}
```

Add to a stack

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

Delete from a stack

A Mazing Problem (8)

```
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;
    }
  }
```

A Mazing Problem (9)

```
  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");
}
```

- Analysis:
  - The worst case of computing time of *path* is O(*mp*), where *m* and *p* are the number of rows and columns of the maze respectively

A Mazing Problem (10)

- The size of a stack

$$mp \rightarrow \lceil m/2 \rceil * p, \text{ or } \lceil p/2 \rceil * m$$

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

<u>Simple maze with a long path</u>

EVALUATION OF EXPRESSIONS

Evaluation of Expressions (1)

- The representation and evaluation of expressions

  ((rear+1==front) || ((rear==MAX_QUEUE_SIZE-1) && !front))

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

  – If we examine expressions, we notice that they contains
    - operators: ==, +, -, ||, &&, !
    - operands: rear, front, MAX_QUEUE_SIZE
    - parentheses: ( )

Evaluation of Expressions (2)

- Understanding the meaning of these or any other expressions and statements
  - Assume a = 4, b = c = 2, d = e = 3 in the statement x = a/b - c+d*e - a*c, finding out the value of x
    - 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…
  - We would have written the statement differently by using parentheses to change the order of evaluation
    - x = ((a/(b - c+d))*(e - a)*c

Precedence Hierarchy for C (1)

- 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
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right

Precedence Hierarchy for C (2)

<table>
<tr><td>> >=
< <=</td><td>relational</td><td>10</td><td>left-to-right</td></tr>
<tr><td>== !=</td><td>equality</td><td>9</td><td>left-to-right</td></tr>
<tr><td>&</td><td>bitwise and</td><td>8</td><td>left-to-right</td></tr>
<tr><td>^</td><td>bitwise exclusive or</td><td>7</td><td>left-to-right</td></tr>
<tr><td>|</td><td>bitwise or</td><td>6</td><td>left-to-right</td></tr>
<tr><td>&&</td><td>logical and</td><td>5</td><td>left-to-right</td></tr>
<tr><td>||</td><td>logical or</td><td>4</td><td>left-to-right</td></tr>
<tr><td>?:</td><td>conditional</td><td>3</td><td>right-to-left</td></tr>
<tr><td>= += -= /= *= %=
<<= >>= &= ^= |=</td><td>assignment</td><td>2</td><td>right-to-left</td></tr>
<tr><td>,</td><td>comma</td><td>1</td><td>left-to-right</td></tr>
</table>

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

Evaluating Postfix Expressions (1)

- The standard way of writing expressions is known as ***infix*** notation
  - binary operator in-between its two operands
- Infix notation is not the one used by compilers to evaluate expressions
- Instead compilers typically use a parenthesis-free notation referred to as ***postfix***
  - Postfix: no parentheses, no precedence

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

Evaluating Postfix Expressions (2)

- Evaluating postfix expressions is much simpler than the evaluation of infix expressions
  - There are no parentheses to consider
  - To evaluate an expression we make a single left-to-right scan of it
  - We can evaluate an expression easily by using a stack
  - The figure shows this processing when the input is nine character string 6 2/3-4 2*+

Token	Stack [0]	Stack [1]	Stack [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

Postfix evaluation

Evaluating Postfix Expressions (3)

- Representation
  - We now consider the representation of both the stack and the expression

```
#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 */
```

Evaluating Postfix Expressions (4)

- Get Token

```
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 ' ' : return eos;
    default  : return operand; /* no error checking,
                                   default is operand */
  }
}
```

Function to get a token from the input string

Evaluating Postfix Expressions (5)

- Function to evaluate a postfix expression

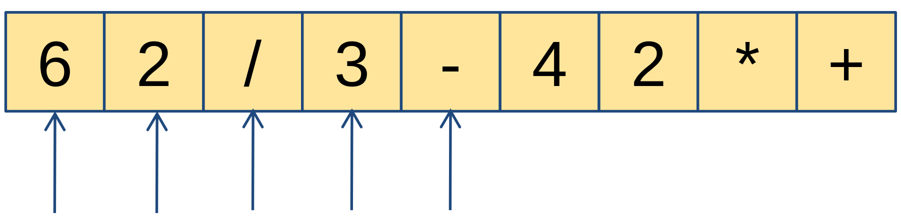
```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
global variable. '\0' is the the end of the expression.
The stack and top of the stack are global variables.
get_token is used to return the tokentype and
the character symbol. Operands are assumed to be single
character digits */
   precedence token;
   char symbol;
   int op1, op2;
   int n = 0; /* counter for the expression string */
   int top = -1;
```

Evaluating Postfix Expressions (6)

```
   token = get_token(&symbol, &n);
   while (token != eos) {
      if (token == operand)
         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 */
}
```

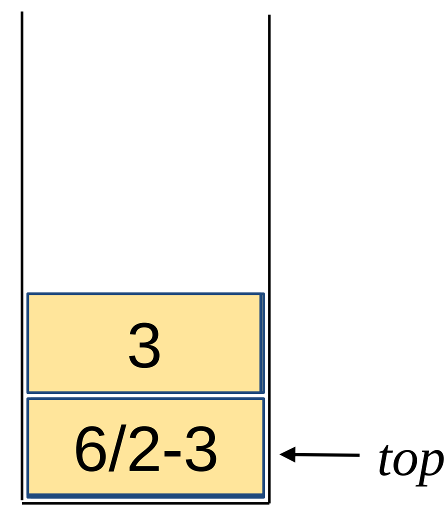
Evaluating Postfix Expressions (7)

- string: 6 2/3-4 2*+
- make a single left-to-right scan



not an operator, push it into the stack

an operator, pop out two values, calculate it and push the result into the stack



Evaluation of Expressions (1)

- The idea for producing a postfix expression from an infix one

Ex: Trans a / b - c + d * e - a * c To postfix

(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))

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

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

(3)Delete all parentheses

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

The order of operands is the same in infix and postfix

Evaluation of Expressions (2)

- Algorithm to convert from infix to postfix
  - Assumptions
    - operators: (,), +, -, *, /, %
    - operands: single digit integer or variable of one character
  - Scan string from left to right
  - Operands are taken out immediately
  - 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

Evaluation of Expressions (3)

- ‘(‘ has low isp, and high icp

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

Evaluation of Expressions (4)

Example [*Simple expression*]: Simple expression a+b*c, which yields abc*+ in postfix

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

Translation of $a+b*c$ to postfix

Evaluation of Expressions (5)

Example : The expression a*(b+c)*d, which yields abc+*d* in postfix

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	*			0	*abc* +**d**

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

Evaluation of Expressions (6)

- Complexity: Θ(n)
  - The total time spent here is Θ(n) as the number of tokens that get stacked and unstacked is linear in n
  - where n is the number of tokens in the expression

```
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);
   printf("\n");
}
```

Function to convert from infix to postfix